# Unity Manual

Welcome to Unity.

Unity is made to empower you to create the best interactive entertainment or multimedia experience that you can. This manual is designed to help you learn how to use Unity, from basic to advanced techniques. It can be read from start to finish or used as a reference.

The manual is divided into different sections. The first section, User Guide, is an introduction to Unity's interface, asset workflow, and the basics of building a game. If you are new to Unity, you should start by reading the Unity Basics subsection.

The iOS Guide addresses iOS specific topics such as iOS-specific scripting API, optimizations, and general platform development questions.

The Android Guide addresses Android specific topics such as setting up the Android SDK and general development questions.

The next section, FAQ, is a collection of frequently asked questions about performing common tasks that require a few steps.

The last section, Advanced, addresses topics such as game optimization, shaders, file sizes, and deployment.

When you've finished reading, take a look at the Reference Manual and the Scripting Reference for further details about the different possibilities of constructing your games with Unity.

If you find that any question you have is not answered in this manual please ask on Unity Answers or Unity Forums. You will be able to find your answer there.

Happy reading,
*The Unity team*

The Unity Manual Guide contains some sections that apply only to certain platforms. Please select which platforms you want to see. Platform-specific information can always be seen by clicking on the disclosure triangles on each page.

## Table of Contents


- User Guide
  - Unity Basics
    - Learning the Interface
      - Project Browser
      - Hierarchy
      - Toolbar
      - Scene View
      - Game View
      - Inspector
      - Other Views
    - Customizing Your Workspace
    - Asset Workflow
    - Creating Scenes
    - Publishing Builds
    - Tutorials
    - Unity Hotkeys
    - Preferences
  - Building Scenes
    - GameObjects
      - The GameObject-Component Relationship

            - Using Components
            - The Component-Script Relationship
            - Deactivating GameObjects
        - Using the Inspector
            - Editing Value Properties
            - Assigning References
            - Multi-Object Editing
            - Inspector Options
        - Using the Scene View
            - Scene View Navigation
            - Positioning GameObjects
            - View Modes
            - Gizmo and Icon Display Controls
        - Searching
        - Prefabs
        - Lights
        - Cameras
        - Terrain Engine Guide
    - Asset Import and Creation
        - Importing Assets
        - Meshes
            - 3D formats
        - Legacy animation system
        - Materials and Shaders
        - Texture 2D
        - Procedural Materials
        - Movie Texture
        - Audio Files
            - Tracker Modules
        - Using Scripts
        - Asset Store
        - Asset Server (Pro Only)
            - Cache Server (Team license only)
            - Cache Server FAQ
        - Behind the Scenes
    - Creating Gameplay
        - Instantiating Prefabs at runtime
        - Input
        - Transforms
        - Physics
        - Adding Random Gameplay Elements
        - Particle Systems
            - Particle System Curve Editor
            - Colors and Gradients in the Particle System (Shuriken)
            - Gradient Editor
            - Particle System Inspector
            - Introduction to Particle System Modules (Shuriken)
            - Particle System Modules (Shuriken)
            - Particle Effects (Shuriken)
        - Mecanim Animation System
            - A Glossary of Animation and Mecanim terms
            - Asset Preparation and Import
                - Preparing your own character
                - Importing Animations
                - Splitting Animations
            - Working with humanoid animations
                - Creating the Avatar
                - Configuring the Avatar
                - Muscle setup
                - Avatar Body Mask
                - Retargeting of Humanoid animations

                - Inverse Kinematics (Pro only)
            - Generic Animations in Mecanim
            - Bringing Characters to Life
                - Looping animation clips
                - Animator Component and Animator Controller
                - Animation State Machines
                    - Animation States
                    - Animation Transitions
                    - Animation Parameters
                - Blend Trees
                - Mecanim Advanced topics
                    - Working with Animation Curves in Mecanim (Pro only)
                    - Sub-State Machines
                    - Animation Layers
                    - Animation State Machine Preview (solo and mute)
                    - Target Matching
                    - Root Motion - how it works
                        - Tutorial: Scripting Root Motion for "in-place" humanoid animations
        - Legacy animation system
            - Animation View Guide (Legacy)
            - Animation Scripting (Legacy)
        - Navmesh and Pathfinding (Pro only)
            - Navmesh Baking
        - Sound
        - Game Interface Elements
        - Networked Multiplayer
- Getting Started with iOS Development
    - Unity iOS Basics
    - Unity Remote
    - iOS Scripting
        - Input
        - Mobile Keyboard
        - Advanced Unity Mobile Scripting
        - Using .NET API 2.0 compatibility level
    - iOS Hardware Guide
    - Optimizing Performance in iOS.
        - iOS Specific Optimizations
        - Measuring Performance with the Built-in Profiler
        - Optimizing the Size of the Built iOS Player
    - Account Setup
    - Features currently not supported by Unity iOS
    - Building Plugins for iOS
    - Preparing your application for "In App Purchases"
    - Customizing the Splash screen of Your Mobile Application
    - Trouble Shooting
    - Reporting crash bugs on iOS
- Getting Started with Android Development
    - Android SDK Setup
    - Android Remote
    - Trouble Shooting
    - Reporting crash bugs under Android
    - Features currently not supported by Unity Android
    - android-OBBsupport
    - Player Settings
    - Android Scripting
        - Input
        - Mobile Keyboard
        - Advanced Unity Mobile Scripting
        - Using .NET API 2.0 compatibility level
    - Building Plugins for Android
    - Customizing the Splash screen of Your Mobile Application

- Getting Started with Native Client Development
- Getting Started with Flash Development
  - Flash: Setup
  - Flash: Building & Running
  - Flash: Debugging
  - Flash: What is and is not supported
  - Flash: Embedding Unity Generated Flash Content in Larger Flash Projects
  - Flash: Adobe Premium Features License
  - Example: Supplying Data from Flash to Unity
  - Example: Calling ActionScript Functions from Unity
  - Example: Browser JavaScript Communication
  - Example: Accessing the Stage
- FAQ
  - Upgrade Guide from Unity 3.5 to 4.0
  - Unity 3.5 upgrade guide
  - Upgrading your Unity Projects from 2.x to 3.x
    - Physics upgrade details
    - Mono Upgrade Details
    - Rendering upgrade details
    - Unity 3.x Shader Conversion Guide
  - Unity 4.0 Activation - Overview
    - Managing your Unity 4.x license
    - Step-by-Step Guide to Online Activation of Unity 4.0
    - Step-by-Step Guide to Manual Activation of Unity 4.0
  - Game Code Questions
    - How to make a simple first person walkthrough
  - Graphics Questions
    - How do I Import Alpha Textures?
    - How do I Use Normal Maps?
    - How do I use Detail Textures?
    - How do I Make a Cubemap Texture?
    - How do I Make a Skybox?
    - How do I make a Mesh Particle Emitter? (Legacy Particle System)
    - How do I make a Splash Screen?
    - How do I make a Spot Light Cookie?
    - How do I fix the rotation of an imported model?
    - How do I use Water?
  - FBX export guide
  - Art Asset Best-Practice Guide
  - How do I import objects from my 3D app?
    - Importing Objects From Maya
    - Importing Objects From Cinema 4D
    - Importing Objects From 3D Studio Max
    - Importing Objects From Cheetah3D
    - Importing Objects From Modo
    - Importing Objects From Lightwave
    - Importing Objects From Blender
  - Workflow Questions
    - Getting started with Mono Develop
    - How do I reuse assets between projects?
    - How do I install or upgrade Standard Assets?
    - Porting a Project Between Platforms
  - Mobile Developer Checklist
    - Crashes
    - Profiling
    - Optimizations
- Advanced
  - Vector Cookbook
    - Understanding Vector Arithmetic
    - Direction and Distance from One Object to Another
    - Computing a Normal/Perpendicular vector

        - The Amount of One Vector's Magnitude that Lies in Another Vector's Direction
    - AssetBundles (Pro only)
        - AssetBundles FAQ
        - Building AssetBundles
        - Downloading AssetBundles
        - Loading resources from AssetBundles
        - Keeping track of loaded AssetBundles
        - Storing and loading binary data in an AssetBundle
        - Protecting Content
        - Managing asset dependencies
        - Including scripts in AssetBundles
    - Graphics Features
        - HDR (High Dynamic Range) Rendering in Unity
        - Rendering Paths
        - Linear Lighting (Pro Only)
        - Level of Detail (Pro Only)
        - Shaders
            - Shaders: ShaderLab & Fixed Function shaders
            - Shaders: Vertex and Fragment Programs
        - Using DirectX 11 in Unity 4
        - Compute Shaders
        - Graphics Emulation
    - AssetDatabase
    - Build Player Pipeline
    - Profiler (Pro only)
    - Lightmapping Quickstart
        - Lightmapping In-Depth
        - Custom Beast Settings
        - Lightmapping UVs
        - Light Probes
    - Occlusion Culling (Pro only)
    - Camera Tricks
        - UnderstandingFrustum
        - The Size of the Frustum at a Given Distance from the Camera
        - Dolly Zoom (AKA the "Trombone" Effect)
        - Rays from the Camera
        - Using an Oblique Frustum
        - Creating an Impression of Large or Small Size
    - Loading Resources at Runtime
    - Modifying Source Assets Through Scripting
    - Generating Mesh Geometry Procedurally
        - Anatomy of a Mesh
        - Using the Mesh Class
        - Example - Creating a Billboard Plane
    - Rich Text
    - Using Mono DLLs in a Unity Project
    - Execution Order of Event Functions
    - Practical Guide to Optimization for Mobiles
        - Practical Guide to Optimization for Mobiles - Future & High End Devices
        - Practical Guide to Optimization for Mobiles - Graphics Methods
        - Practical Guide to Optimization for Mobiles - Scripting and Gameplay Methods
        - Practical Guide to Optimization for Mobiles - Rendering Optimizations
        - Practical Guide to Optimization for Mobiles - Optimizing Scripts
    - Optimizing Graphics Performance
        - Draw Call Batching
        - Modeling Characters for Optimal Performance
        - Rendering Statistics Window
    - Reducing File Size
    - Understanding Automatic Memory Management
    - Platform Dependent Compilation
    - Generic Functions

- Debugging
  - Console
  - Debugger
  - Log Files
    - Accessing hidden folders
- Plugins (Pro/Mobile-Only Feature)
  - Building Plugins for Desktop Platforms
  - Building Plugins for iOS
  - Building Plugins for Android
  - Low-level Native Plugin Interface
- Textual Scene File Format (Pro-only Feature)
  - Description of the Format
  - YAMLSceneExample
  - YAML Class ID Reference
- Streaming Assets
- Command line arguments
- Running Editor Script Code on Launch
- Network Emulation
- Security Sandbox of the Webplayer
- Overview of available .NET Class Libraries
- Visual Studio C# Integration
- Using External Version Control Systems with Unity
- Analytics
- Check For Updates
- Installing Multiple Versions of Unity
- Trouble Shooting
- Shadows in Unity
  - Directional Shadow Details
  - Troubleshooting Shadows
  - Shadow Size Computation
  - IME in Unity
  - Optimizing for integrated graphics cards
  - Web Player Deployment
    - HTML code to load Unity content
    - Working with UnityObject2
    - Customizing the Unity Web Player loading screen
    - Customizing the Unity Web Player's Behavior
    - Unity Web Player and browser communication
    - Using web player templates
    - Web Player Streaming
    - Webplayer Release Channels


Page last updated: 2012-11-14

---

# User Guide

This section of the Manual is focused on the features and functions of Unity. It discusses the interface, core Unity building blocks, asset workflow, and basic gameplay creation. By the time you are done reading the user guide, you will have a solid understanding of how to use Unity to put together an interactive scene and publish it.

We recommend that new users begin by reading the Unity Basics section.


- Unity Basics
  - Learning the Interface
    - Project Browser
    - Hierarchy
    - Toolbar
    - Scene View
    - Game View

            - Inspector
            - Other Views
        - Customizing Your Workspace
        - Asset Workflow
        - Creating Scenes
        - Publishing Builds
        - Tutorials
        - Unity Hotkeys
        - Preferences
- Building Scenes
    - GameObjects
        - The GameObject-Component Relationship
        - Using Components
        - The Component-Script Relationship
        - Deactivating GameObjects
    - Using the Inspector
        - Editing Value Properties
        - Assigning References
        - Multi-Object Editing
        - Inspector Options
    - Using the Scene View
        - Scene View Navigation
        - Positioning GameObjects
        - View Modes
        - Gizmo and Icon Display Controls
    - Searching
    - Prefabs
    - Lights
    - Cameras
    - Terrain Engine Guide
- Asset Import and Creation
    - Importing Assets
    - Meshes
        - 3D formats
    - Legacy animation system
    - Materials and Shaders
    - Texture 2D
    - Procedural Materials
    - Movie Texture
    - Audio Files
        - Tracker Modules
    - Using Scripts
    - Asset Store
    - Asset Server (Pro Only)
        - Cache Server (Team license only)
        - Cache Server FAQ
    - Behind the Scenes
- Creating Gameplay
    - Instantiating Prefabs at runtime
    - Input
    - Transforms
    - Physics
    - Adding Random Gameplay Elements
    - Particle Systems
        - Particle System Curve Editor
        - Colors and Gradients in the Particle System (Shuriken)
        - Gradient Editor
        - Particle System Inspector
        - Introduction to Particle System Modules (Shuriken)
        - Particle System Modules (Shuriken)
        - Particle Effects (Shuriken)

- Mecanim Animation System
  - A Glossary of Animation and Mecanim terms
  - Asset Preparation and Import
    - Preparing your own character
    - Importing Animations
    - Splitting Animations
  - Working with humanoid animations
    - Creating the Avatar
    - Configuring the Avatar
    - Muscle setup
    - Avatar Body Mask
    - Retargeting of Humanoid animations
    - Inverse Kinematics (Pro only)
  - Generic Animations in Mecanim
  - Bringing Characters to Life
    - Looping animation clips
    - Animator Component and Animator Controller
    - Animation State Machines
      - Animation States
      - Animation Transitions
      - Animation Parameters
    - Blend Trees
    - Mecanim Advanced topics
      - Working with Animation Curves in Mecanim (Pro only)
      - Sub-State Machines
      - Animation Layers
      - Animation State Machine Preview (solo and mute)
      - Target Matching
      - Root Motion - how it works
        - Tutorial: Scripting Root Motion for "in-place" humanoid animations
- Legacy animation system
  - Animation View Guide (Legacy)
  - Animation Scripting (Legacy)
- Navmesh and Pathfinding (Pro only)
  - Navmesh Baking
- Sound
- Game Interface Elements
- Networked Multiplayer


Page last updated: 2010-09-09

---

# Unity Basics

This section is your key to getting started with Unity. It will explain the Unity interface, menu items, using assets, creating scenes, and publishing builds.

When you are finished reading this section, you will understand how Unity works, how to use it effectively, and the steps to put a basic game together.

Learning the Interface
There is a lot to learn, so take the time you need to observe and understand the interface. We will walk through each interface element together.

Asset Workflow

Publishing Builds
At any time while you are creating your game, you might want to see how it looks when you build and run it outside of the editor as a standalone or web player. This section will explain how to access the Build Settings and how to create different builds of your games.

Here we'll explain the steps to use a single asset with Unity. These steps are general and are meant only as an overview for basic actions.

Creating Scenes

Scenes contain the objects of your game. In each Scene, you will place your environments, obstacles, and decorations, designing and building your game in pieces.

Tutorials

These online tutorials will let you work with Unity while you follow along, providing hands-on experience with building real projects.

Page last updated: 2010-09-10

---

## Learning the Interface

Take your time to look over the Unity Editor interface and familiarize yourself with it. The **Main Editor Window** is made up of several **Tabbed Windows**, called **Views**. There are several types of Views in Unity - they all have specific purposes which are described in the subsections below.

- Project Browser
- Hierarchy
- Toolbar
- Scene View
- Game View
- Inspector
- Other Views

Page last updated: 2012-10-17

---

## ProjectView40

In this view, you can access and manage the assets that belong to your project.

The left panel of the browser shows the folder structure of the project as a hierarchical list. When a folder is selected from the list by clicking, its contents will be shown in the panel to the right. The individual assets are shown as icons that indicate their type (script, material, sub-folder, etc). The icons can be resized using the slider at the bottom of the panel; they will be replaced by a hierarchical list view if the slider is moved to the extreme left. The space to the left of the slider shows the currently selected item, including a full path to the item if a search is being performed.

Above the project structure list is a **Favorites** section where you can keep frequently-used items for easy access. You can drag items from the project structure list to the Favourites and also save search queries there (see **Searching** below).

Just above the panel is a "breadcrumb trail" that shows the path to the folder currently being viewed. The separate elements of the trail can be clicked for easy navigation around the folder hierarchy. When searching, this bar changes to show the area being searched (the root Assets folder, the selected folder or the Asset Store) along with a count of free and paid assets available in the store, separated by a slash. There is an option in the *General* section of Unity's Preferences window to disable the display of Asset Store hit counts if they are not required.

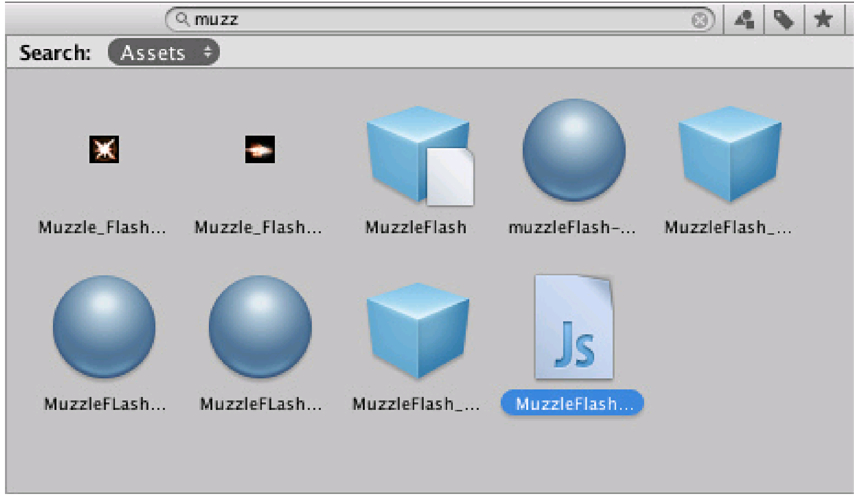
Along the top edge of the window is the browser's toolbar.

Located at the left side of the toolbar, the **Create** menu lets you add new assets and sub-folders to the current folder. To its right are a set of tools to allow you to search the assets in your project.

The Window menu provides the option of switching to a one-column version of the project view, essentially just the hierarchical structure list without the icon view. The lock icon next to the menu enables you to "freeze" the current contents of the view (ie, stop them being changed by events elsewhere) in a similar manner to the inspector lock.

## Searching

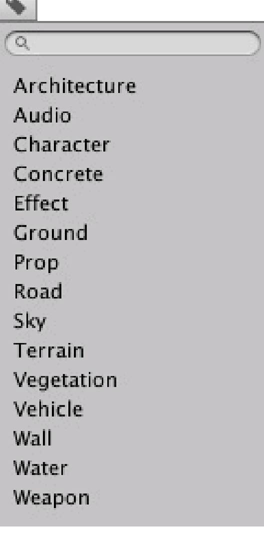
The browser has a powerful search facility that is especially useful for locating assets in large or unfamiliar projects. The basic search will filter assets according to the text typed in the search box




If you type more than one search term then the search is narrowed, so if you type *coastal scene* it will only find assets with both "coastal" and "scene" in the name (ie, terms are ANDed together).

To the right of the search bar are three buttons. The first allows you to further filter the assets found by the search according to their type.

Continuing to the right, the next button filters assets according to their Label (labels can be set for an asset in the Inspector). Since the number of labels can potentially be very large, the label menu has its own mini-search filter box.




Note that the filters work by adding an extra term in the search text. A term beginning with "t:" filters by the specified asset type, while "l:" filters by label. You can type these terms directly into the search box rather than use the menu if you know what you are looking for. You can search for more than one type or label at once. Adding several types will expand the search to include

all specified types (ie, types will be ORed together). Adding multiple labels will narrow the search to items that have all the specified labels (ie, labels are ANDed).

muzzle flash t:Material t:Texture l:Weapon

The rightmost button saves the search by adding an item to the Favourites section of the asset list.

**Searching the Asset Store**

The Project Browser's search can also be applied to assets available from the Unity **Asset Store**. If you choose **Asset Store** from the menu in the breadcrumb bar, all free and paid items from the store that match your query will be displayed. Searching by type and label works the same as for a Unity project. The search query words will be checked against the asset name first and then the package name, package label and package description in that order (so an item whose name contains the search terms will be ranked higher than one with the same terms in its package description).

If you select an item from the list, its details will be displayed in the inspector along with the option to purchase and/or download it. Some asset types have previews available in this section so you can, for example, play an audio clip or rotate a 3D model before buying. The inspector also gives the option of viewing the asset in the usual Asset Store window to see additional details.

## Shortcuts

The following keyboard shortcuts are available when the browser view has focus. Note that some of them only work when the view is using the two-column layout (you can switch between the one- and two-column layouts using the panel menu in the very top right corner).

| | |
|---|---|
| **F** | Frame selection |
| **Tab** | Shift focus between first column and second column (Two columns) |
| **Ctrl/Cmd + F** | Focus search field |
| **Ctrl/Cmd + A** | Select all visible items in list |
| **Ctrl/Cmd + D** | Duplicate selected assets |
| **Delete** | Delete with dialog |
| **Delete + Shift** | Delete without dialog |
| **Backspace + Cmd** | Delete without dialogs (OSX) |
| **Enter** | Begin rename selected (OSX) |
| **Cmd + down arrow** | Open selected assets (OSX) |
| **Cmd + up arrow** | Jump to parent folder (OSX, Two columns) |
| **F2** | Begin rename selected (Win) |
| **Enter** | Open selected assets (Win) |
| **Backspace** | Jump to parent folder (Win, Two columns) |
| **Right arrow** | Expand selected item (tree views and search results). If the item is already expanded, this will select its first child item. |
| **Left arrow** | Collapse selected item (tree views and search results). If the item is already collapsed, this will select its parent item. |
| **Alt + right arrow** | Expand item when showing assets as previews |
| **Alt + left arrow** | Collapse item when showing assets as previews |

Page last updated: 2012-11-15

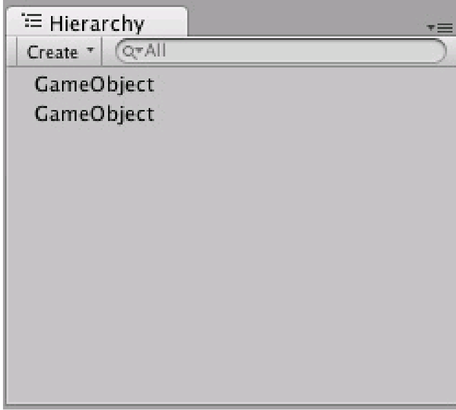
---

## Hierarchy

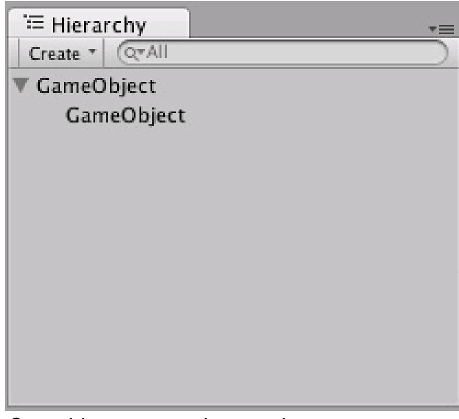
The **Hierarchy** contains every **GameObject** in the current Scene. Some of these are direct instances of asset files like 3D models, and others are instances of Prefabs, custom objects that will make up much of your game. You can select objects in the Hierarchy and drag one object onto another to make use of **Parenting** (see below). As objects are added and removed in the scene, they will appear and disappear from the Hierarchy as well.

### Parenting

Unity uses a concept called **Parenting**. To make any GameObject the child of another, drag the desired child onto the desired parent in the Hierarchy. A child will inherit the movement and rotation of its parent. You can use a parent object's foldout arrow to show or hide its children as necessary.




*Two unparented objects*



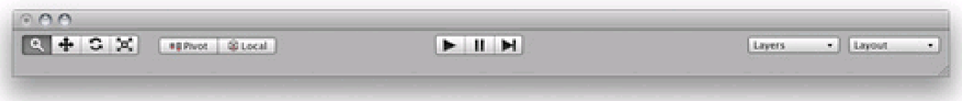

*One object parented to another*

To learn more about Parenting, please review the Parenting section of the Transform Component page.

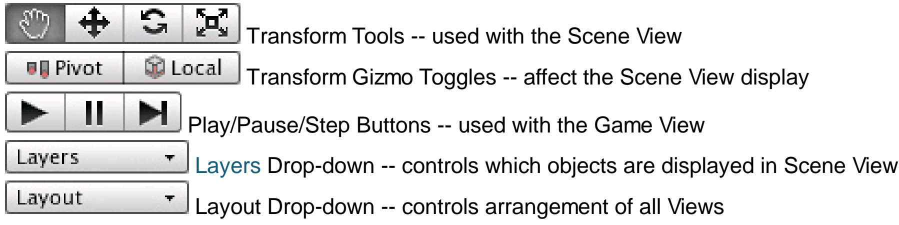
Page last updated: 2012-10-18

---

# Toolbar




The Toolbar consists of five basic controls. Each relate to different parts of the Editor.




Transform Tools -- used with the Scene View

Transform Gizmo Toggles -- affect the Scene View display

Play/Pause/Step Buttons -- used with the Game View

Layers Drop-down -- controls which objects are displayed in Scene View

Layout Drop-down -- controls arrangement of all Views

Page last updated: 2012-10-17

---

# SceneView40

*The **Scene View***

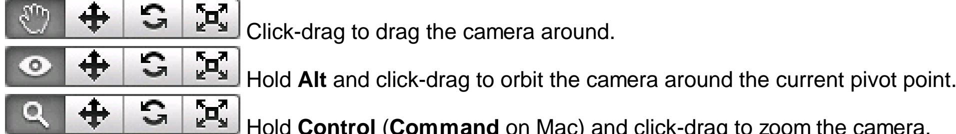
The **Scene View** is your interactive sandbox. You will use the Scene View to select and position environments, the player, the camera, enemies, and all other **GameObjects**. Maneuvering and manipulating objects within the Scene View are some of the most important functions in Unity, so it's important to be able to do them quickly. To this end, Unity provides keystrokes for the most common operations.

### Scene View Navigation

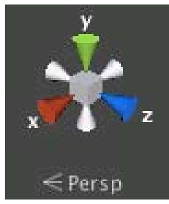
See Scene View Navigation for full details on navigating the scene view. Here's a brief overview of the essentials:

- Hold the right mouse button to enter **Flythrough** mode. This turns your mouse and **WASD** keys (plus **Q** and **E** for up and down) into quick first-person view navigation.
- Select any GameObject and press the **F** key. This will center the Scene View and pivot point on the selection.
- Use the arrow keys to move around on the X/Z plane.
- Hold **Alt** and click-drag to orbit the camera around the current pivot point.
- Hold **Alt** and middle click-drag to drag the Scene View camera around.
- Hold **Alt** and right click-drag to zoom the Scene View. This is the same as scrolling with your mouse wheel.

You might also find use in the **Hand Tool** (shortcut: **Q**), especially if you are using a one-button mouse. With the Hand tool is selected,


Click-drag to drag the camera around.
Hold **Alt** and click-drag to orbit the camera around the current pivot point.
Hold **Control** (**Command** on Mac) and click-drag to zoom the camera.

In the upper-right corner of the Scene View is the **Scene Gizmo**. This displays the Scene Camera's current orientation, and allows you to quickly modify the viewing angle.

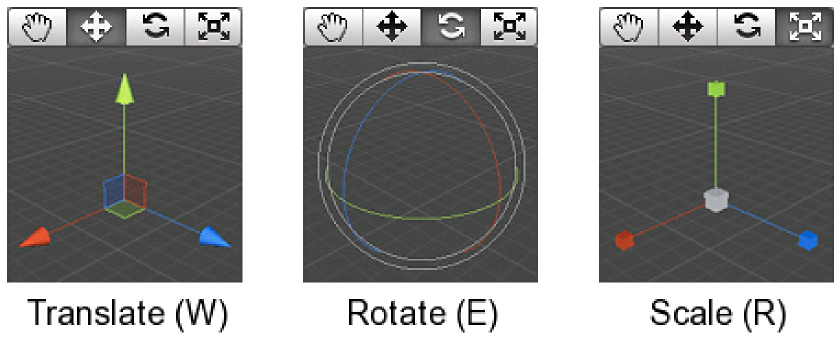



Each of the coloured "arms" of the gizmo represents a geometric axis. You can click on any of the arms to set the camera to an orthographic (i.e., perspective-free) view looking along the corresponding axis. You can click on the text underneath the gizmo to switch between the normal perspective view and an isometric view. While in isometric mode, you can right-click drag to orbit, and Alt-click drag to pan.

### Positioning GameObjects

See Positioning GameObjects for full details on positioning GameObjects in the scene. Here's a brief overview of the essentials:

When building your games, you'll place lots of different objects in your game world. To do this use the Transform Tools in the Toolbar to Translate, Rotate, and Scale individual GameObjects. Each has a corresponding Gizmo that appears around the selected GameObject in the Scene View. You can use the mouse and manipulate any Gizmo axis to alter the **Transform** Component of the GameObject, or you can type values directly into the number fields of the Transform Component in the Inspector.

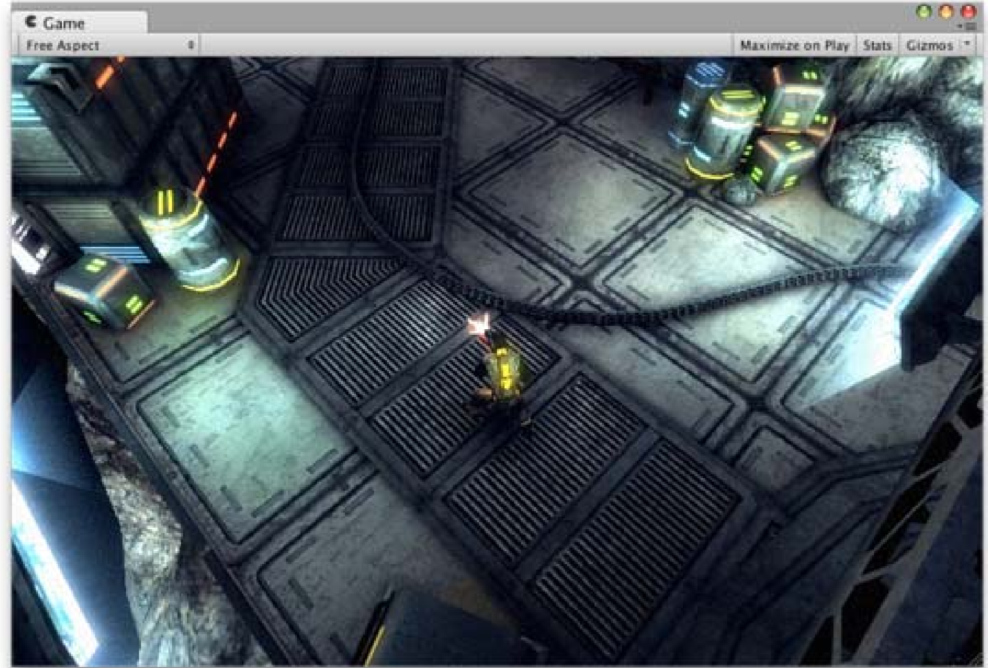



### Scene View Control Bar

The Scene View control bar lets you see the scene in various view modes - Textured, Wireframe, RGB, Overdraw, and many others. It will also enable you to see (and hear) in-game lighting, game elements, and sound in the Scene View. See View Modes for all the details.

Page last updated: 2012-10-19

## GameView40




The **Game View** is rendered from the Camera(s) in your game. It is representative of your final, published game. You will need to use one or more **Cameras** to control what the player actually sees when they are playing your game. For more information about Cameras, please view the Camera Component page.

### Play Mode

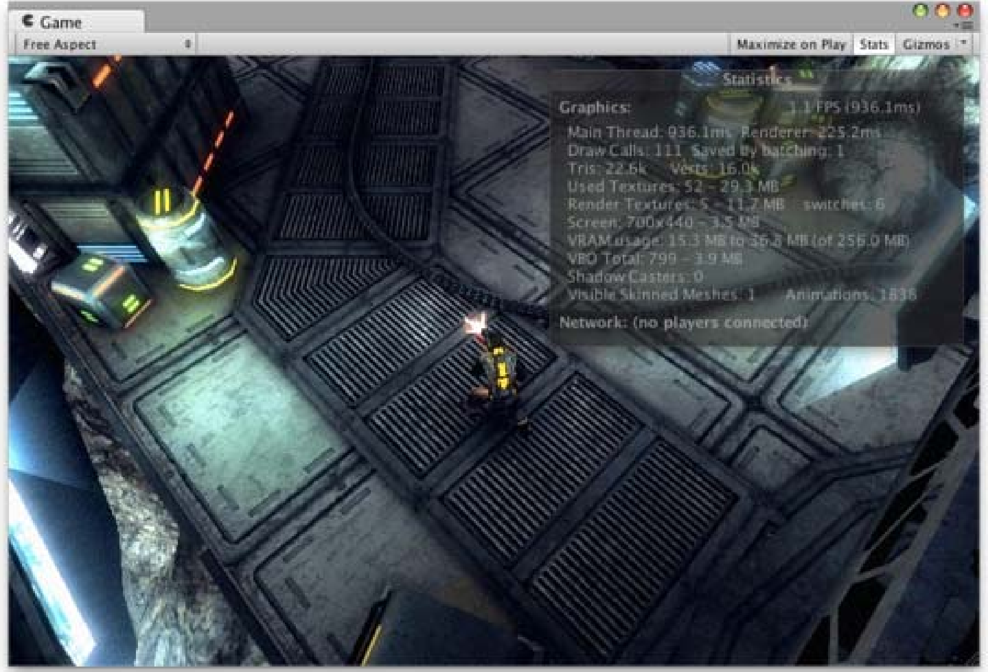
Use the buttons in the Toolbar to control the Editor **Play Mode** and see how your published game will play. While in Play mode, any changes you make are temporary, and will be reset when you exit Play mode. The Editor UI will darken to remind you of this.

### Game View Control Bar

The first drop-down on the Game View control bar is the **Aspect Drop-down**. Here, you can force the aspect ratio of the Game View window to different values. It can be used to test how your game will look on monitors with different aspect ratios.

Further to the right is the **Maximize on Play** toggle. While enabled, the Game View will maximize itself to 100% of your Editor Window for a nice full-screen preview when you enter Play mode.

Continuing to the right is the **Stats** button. This shows Rendering Statistics window that is very useful for monitoring the graphics performance of your game (see Optimizing Graphics Performance for further details).




The last button is the **Gizmos** toggle. While enabled, all Gizmos that appear in Scene View will also be drawn in Game View. This includes Gizmos drawn using any of the **Gizmos** class functions. The Gizmos button also has a popup menu showing the various different types of Components used in the game.

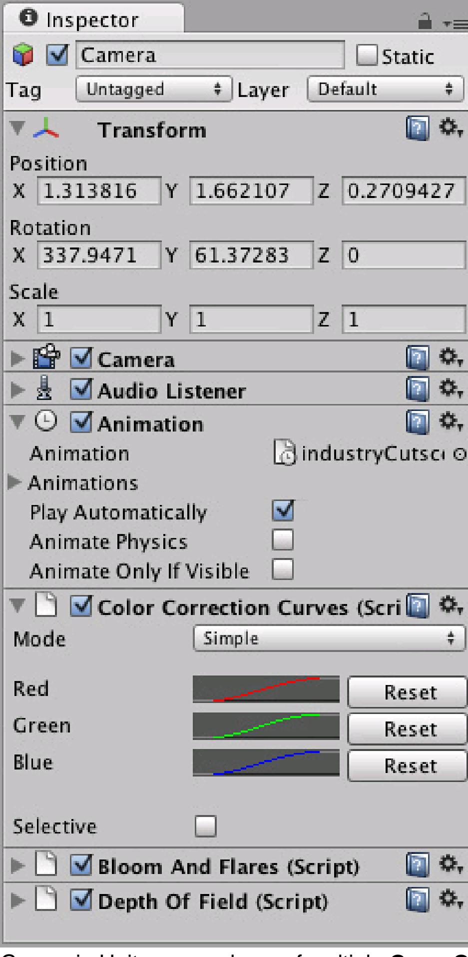
Next to each Component's name are the settings for the icon and gizmos associated with it. The *Icon* setting reveals another popup menu which lets you choose from a selection of preset icons or a custom icon defined by a texture.

The *Gizmo* setting enables you to selectively disable Gizmo drawing for specific components.

The *3D Gizmos* setting at the top of the menu refers to the Gizmo icons. With the setting enabled, the icons will show the perspective of the camera (ie, icons for nearby objects will be larger than those for distant objects), otherwise they will be the same size regardless of distance. The slider next to the checkbox allows you to vary the size of the icons, which can be useful for reducing clutter when there are a lot of gizmos visible.

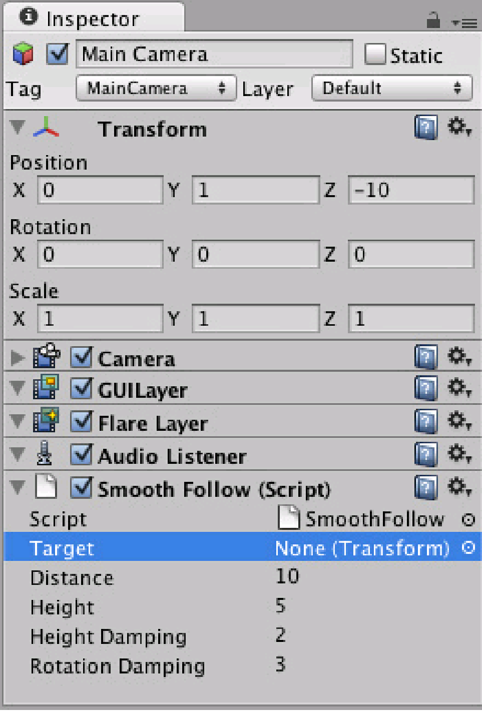
Page last updated: 2012-10-19

## Inspector

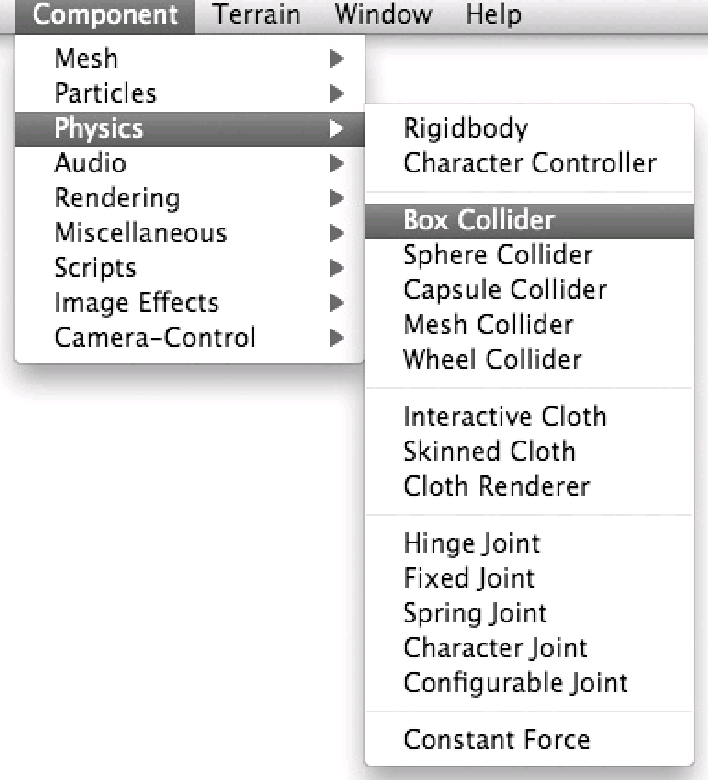



Games in Unity are made up of multiple **GameObjects** that contain meshes, scripts, sounds, or other graphical elements like **Lights**. The **Inspector** displays detailed information about your currently selected GameObject, including all attached **Components** and their properties. Here, you modify the functionality of GameObjects in your scene. You can read more about the GameObject-Component relationship, as it is very important to understand.

Any property that is displayed in the Inspector can be directly modified. Even script variables can be changed without modifying the script itself. You can use the Inspector to change variables at runtime to experiment and find the magic gameplay for your game. In a script, if you define a public variable of an object type (like GameObject or Transform), you can drag and drop a GameObject or Prefab into the Inspector to make the assignment.

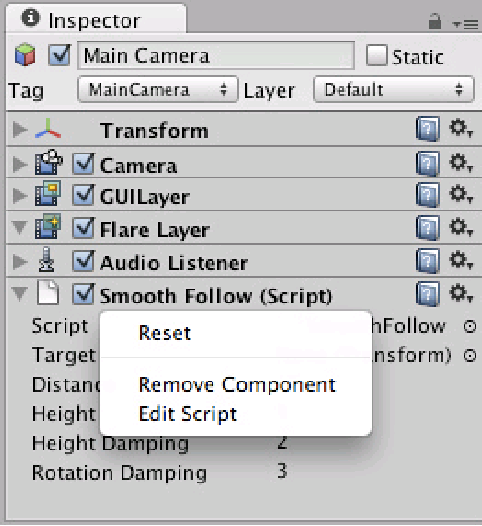



Click the question mark beside any Component name in the Inspector to load its Component Reference page. Please view the Component Reference for a complete and detailed guide to all of Unity's Components.

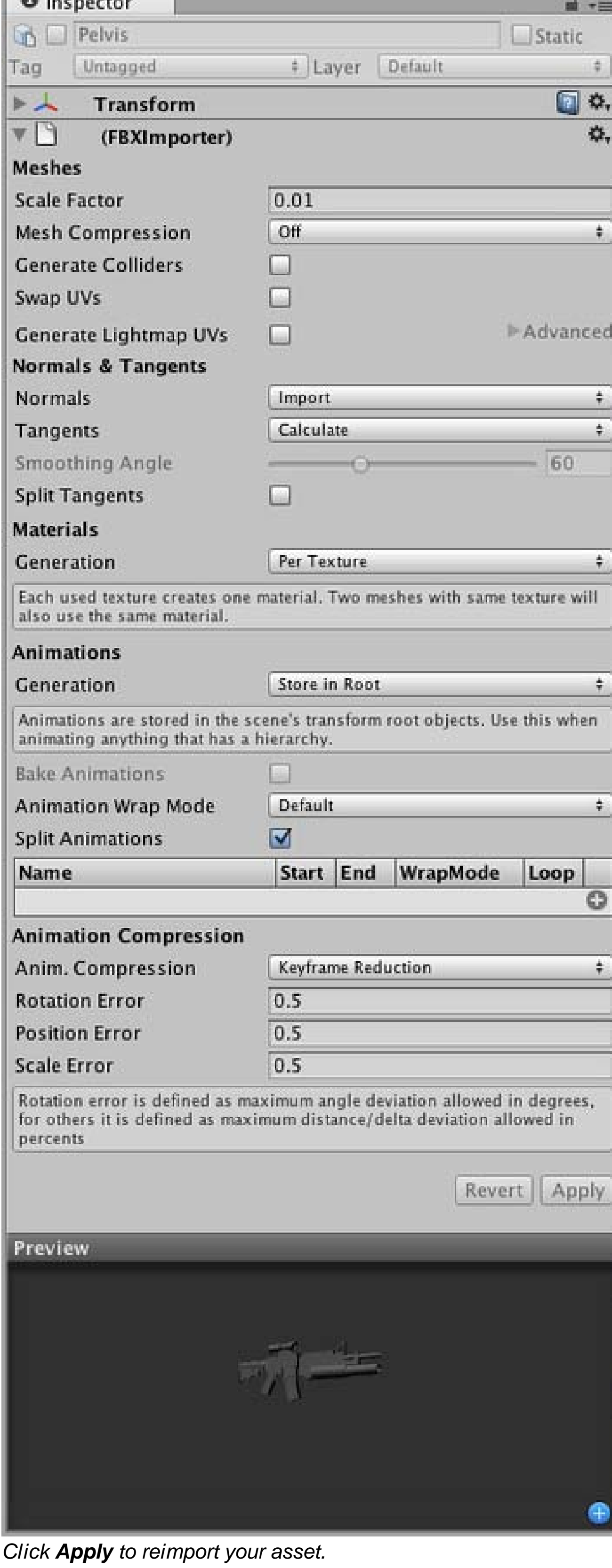



*Add Components from the **Component** menu*

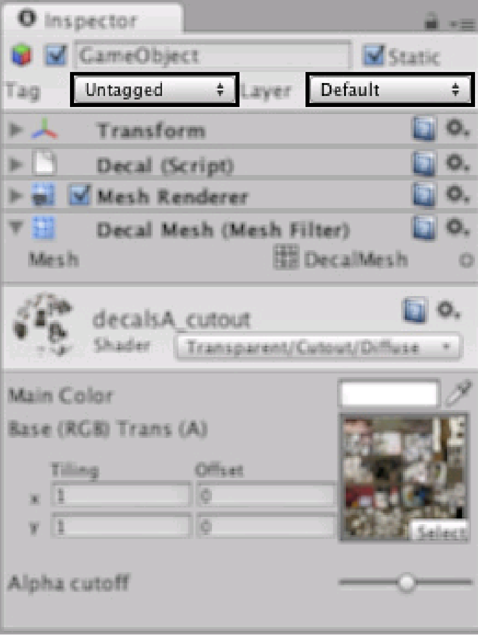
You can click the tiny gear icon (or right-click the Component name) to bring up a context menu for the specific Component.




The Inspector will also show any Import Settings for a selected asset file.




*Click **Apply** to reimport your asset.*



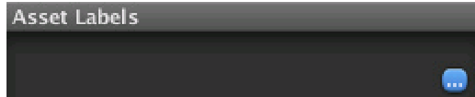

Use the **Layer** drop-down to assign a rendering Layer to the GameObject. Use the **Tag** drop-down to assign a Tag to this GameObject.

## Prefabs

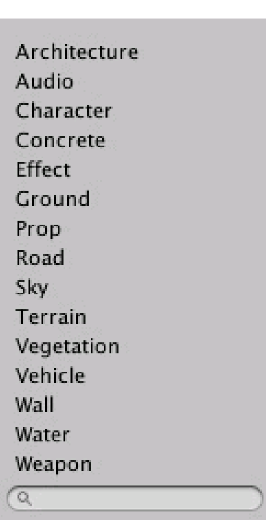
If you have a Prefab selected, some additional buttons will be available in the Inspector. For more information about Prefabs, please view the Prefab manual page.

## Labels

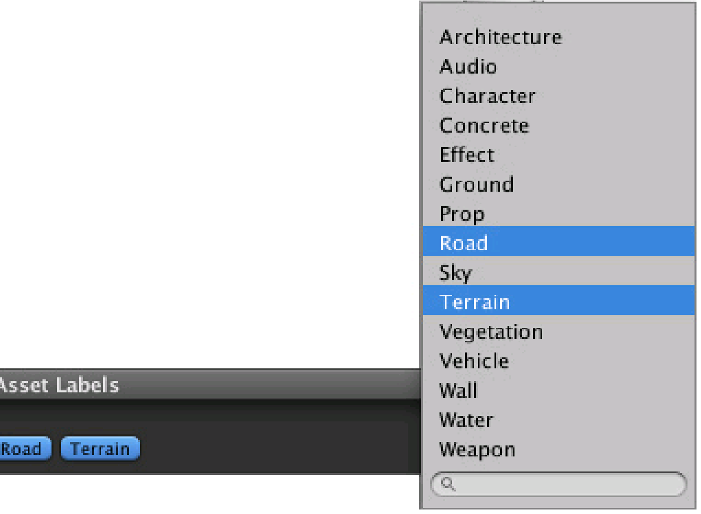
Unity allows assets to be marked with **Labels** to make them easier to locate and categorise. The bottom item on the inspector is the Asset Labels panel.




At the bottom right of this panel is a button titled with an ellipsis ("...") character. Clicking this button will bring up a menu of available labels




You can select one or more items from the labels menu to mark the asset with those labels (they will also appear in the Labels panel). If you click a second time on one of the active labels, it will be removed from the asset.




The menu also has a text box that you can use to specify a search filter for the labels in the menu. If you type a label name that does not yet exist and press return/enter, the new label will be added to the list and applied to the selected asset. If you remove a custom label from all assets in the project, it will disappear from the list.

Once you have applied labels to your assets, you can use them to refine searches in the **Project Browser** (see this page for further details). You can also access an asset's labels from an editor script using the AssetDatabase class.

Page last updated: 2012-11-15

---

# Other Views

The Views described on this page covers the basics of the interface in Unity. The other Views in Unity are described elsewhere on separate pages:

- The Console shows logs of messages, warnings, and errors.
- The Animation View can be used to animate objects in the scene.
- The Profiler can be used to investigate and find the performance bottle-necks in your game.
- The Asset Server View can be used to manage version control of the project using Unity's Asset Server.
- The Lightmapping View can be used to manage lightmaps using Unity's built-in lightmapping.
- The Occlusion Culling View can be used to manage Occlusion Culling for improved performance.

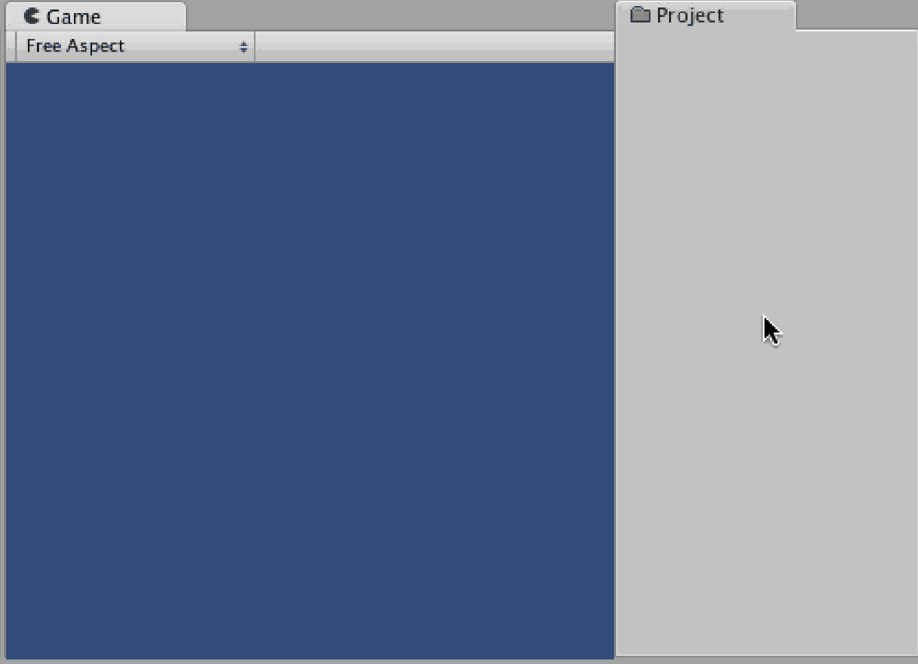
Page last updated: 2012-11-26

---

# Customizing Your Workspace

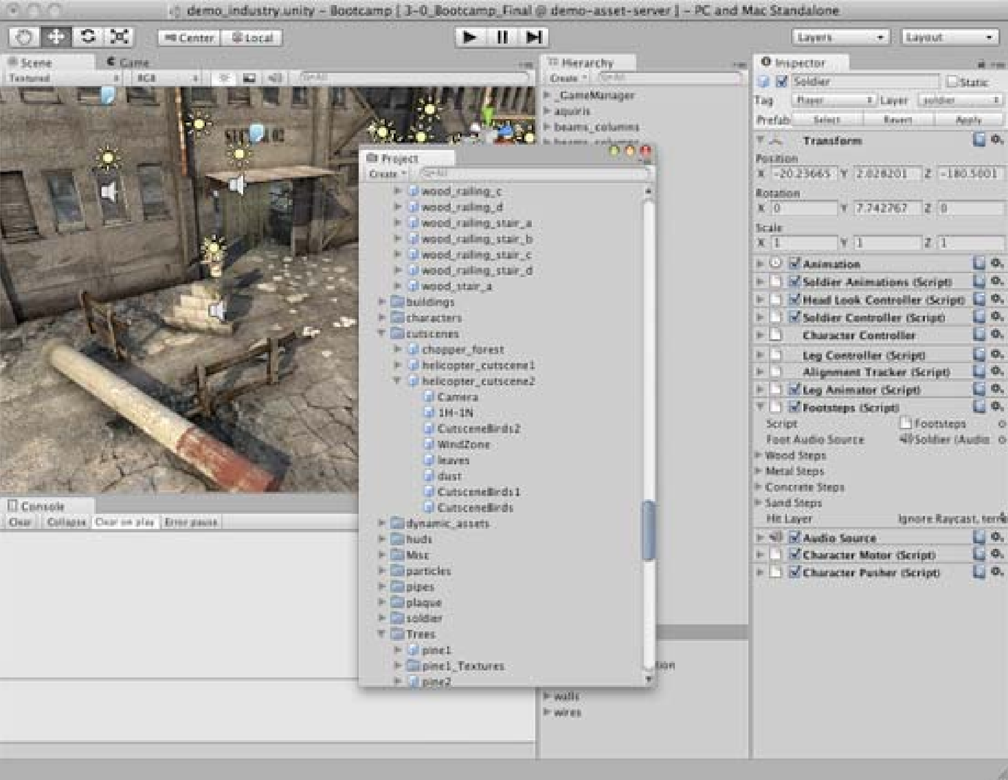
## Customizing Your Workspace

You can customize your **Layout** of Views by click-dragging the Tab of any View to one of several locations. Dropping a Tab in the **Tab Area** of an existing window will add the Tab beside any existing Tabs. Alternatively, dropping a Tab in any **Dock Zone** will add the View in a new window.



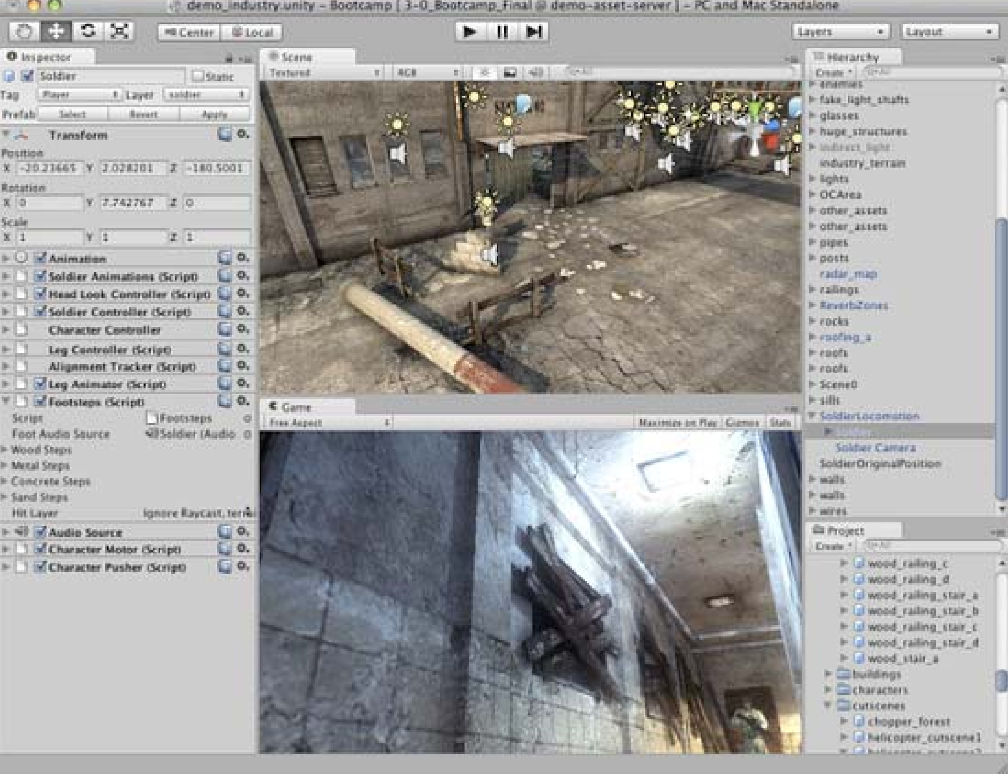

*Views can be docked to the sides or bottom of any existing window*

Tabs can also be detached from the Main Editor Window and arranged into their own floating Editor Windows. Floating Windows can contain arrangements of Views and Tabs just like the Main Editor Window.



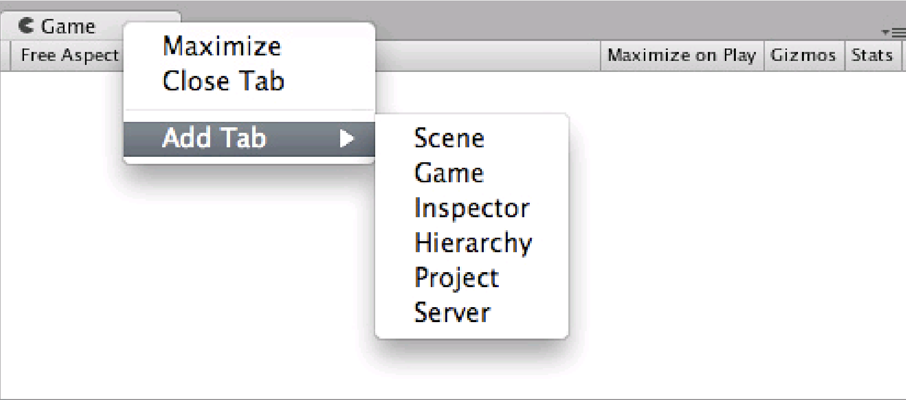

*Floating Editor Windows are the same as the Main Editor Window, except there is no Toolbar*

When you've created a Layout of Editor Windows, you can Save the layout and restore it any time. You do this by expanding the Layout drop-down (found on the Toolbar) and choosing **Save Layout...**. Name your new layout and save it, then restore it by simply choosing it from the Layout drop-down.



*A completely custom Layout*

At any time, you can right-click the tab of any view to view additional options like Maximize or add a new tab to the same window.




Page last updated: 2010-09-07

---

## Asset Workflow

Here we'll explain the steps to use a single asset with Unity. These steps are general and are meant only as an overview for basic actions. For the example, we'll talk about using a 3D mesh.

### Create Rough Asset

Use any supported 3D modeling package to create a rough version of your asset. Our example will use Maya. Work with the asset until you are ready to save. For a list of applications that are supported by Unity, please see this page.

### Import

When you save your asset initially, you should save it normally to the **Assets** folder in your Project folder. When you open the Unity project, the asset will be detected and imported into the project. When you look in the **Project View**, you'll see the asset located there, right where you saved it. Please note that Unity uses the FBX exporter provided by your modeling package to convert your models to the FBX file format. You will need to have the FBX exporter of your modeling package available for

Unity to use. Alternatively, you can directly export as FBX from your application and save in the Projects folder. For a list of applications that are supported by Unity, please see this page.

### Import Settings

If you select the asset in the **Project View** the import settings for this asset will appear in the **Inspector**. The options that are displayed will change based on the type of asset that is selected.

### Adding Asset to the Scene

Simply click and drag the mesh from the Project View to the **Hierarchy** or **Scene View** to add it to the Scene. When you drag a mesh to the scene, you are creating a **GameObject** that has a **Mesh Renderer Component**. If you are working with a texture or a sound file, you will have to add it to a GameObject that already exists in the Scene or Project.

### Putting Different Assets Together

Here is a brief description of the relationships between the most common assets

- A Texture is applied to a **Material**
- A Material is applied to a GameObject (with a Mesh Renderer Component)
- An **Animation** is applied to a GameObject (with an Animation Component)
- A sound file is applied to a GameObject (with an **Audio Source** Component)

### Creating a Prefab

**Prefabs** are a collection of GameObjects & Components that can be re-used in your scenes. Several identical objects can be created from a single Prefab, called instancing. Take trees for example. Creating a tree Prefab will allow you to instance several identical trees and place them in your scene. Because the trees are all linked to the Prefab, any changes that are made to the Prefab will automatically be applied to all tree instances. So if you want to change the mesh, material, or anything else, you just make the change once in the Prefab and all the other trees inherit the change. You can also make changes to an instance, and choose **GameObject->Apply Changes to Prefab** from the main menu. This can save you lots of time during setup and updating of assets.

When you have a GameObject that contains multiple Components and a hierarchy of child GameObjects, you can make a Prefab of the top-level GameObject (or **root**), and re-use the entire collection of GameObjects.

Think of a Prefab as a blueprint for a structure of GameObjects. All the Prefab clones are identical to the blueprint. Therefore, if the blueprint is updated, so are all the clones. There are different ways you can update the Prefab itself by changing one of its clones and applying those changes to the blueprint. To read more about using and updating Prefabs, please view the Prefabs page.

To actually create a Prefab from a GameObject in your scene, simply drag the GameObject from the scene into the project, and you should see the Game Object's name text turn blue. Name the new Prefab whatever you like. You have now created a re-usable prefab.

### Updating Assets

You have imported, instantiated, and linked your asset to a Prefab. Now when you want to edit your source asset, just double-click it from the Project View. The appropriate application will launch, and you can make any changes you want. When you're done updating it, just Save it. Then, when you switch back to Unity, the update will be detected, and the asset will be re-imported. The asset's link to the Prefab will also be maintained. So the effect you will see is that your Prefab will update. That's all you have to know to update assets. Just open it and save!

### Optional - Adding Labels to the Assets.

Is always a good idea to add labels to your assets if you want to keep organized all your assets, with this you can search for the labels associated to each asset in the search field in the project view or in the object selector.

Steps for adding a label to an asset:

- Select the asset you want to add the label to (From the project view).
- In the inspector click on the "Add Label" icon ( ) if you dont have any Labels associated to that asset.
  - If you have a label associated to an asset then just click where the labels are.
- Start writing your labels.

**Notes:**

- You can have more than one label for any asset.
- To separate/create labels, just press **space** or **enter** when writing asset label names.

Page last updated: 2012-09-14

---

# Creating Scenes

**Scenes** contain the objects of your game. They can be used to create a main menu, individual levels, and anything else. Think of each unique Scene file as a unique level. In each Scene, you will place your environments, obstacles, and decorations, essentially designing and building your game in pieces.

### Instancing Prefabs

Use the method described in the last section to create a **Prefab**. You can also read more details about Prefabs here. Once you've created a Prefab, you can quickly and easily make copies of the Prefab, called an **Instance**. To create an instance of any Prefab, drag the Prefab from the **Project View** to the **Hierarchy** or **Scene View**. Now you have a unique instance of your Prefab to position and tweak as you like.

### Adding Component & Scripts

When you have a Prefab or any **GameObject** highlighted, you can add additional functionality to it by using **Components**. Please view the Component Reference for details about all the different Components. **Scripts** are a type of Component. To add a Component, just highlight your GameObject and select a Component from the **Component** menu. You will then see the Component appear in the **Inspector** of the GameObject. Scripts are also contained in the **Component** menu by default.

If adding a Component breaks the GameObject's connection to its Prefab, you can always use **GameObject->Apply Changes to Prefab** from the menu to re-establish the link.

### Placing GameObjects

Once your GameObject is in the scene, you can use the **Transform Tools** to position it wherever you like. Additionally, you can use the **Transform** values in the Inspector to fine-tune placement and rotation. Please view the Transform Component page for more information about positioning and rotating GameObjects.

### Working with Cameras

**Cameras** are the eyes of your game. Everything the player will see when playing is through one or more cameras. You can position, rotate, and parent cameras just like any other GameObject. A camera is just a GameObject with a Camera Component attached to it. Therefore it can do anything a regular GameObject can do, and then some camera-specific functions too. There are also some helpful Camera scripts that are installed with the standard assets package when you create a new project. You can find them in **Components->Camera-Control** from the menu. There are some additional aspects to cameras which will be good to understand. To read about cameras, view the Camera component page.

### Lights

Except for some very few cases, you will always need to add **Lights** to your scene. There are three different types of lights, and all of them behave a little differently from each other. The important thing is that they add atmosphere and ambience to your game. Different lighting can completely change the mood of your game, and using lights effectively will be an important subject to learn. To read about the different lights, please view the Light component page.

Page last updated: 2009-02-16

---

# Publishing Builds

At any time while you are creating your game, you might want to see how it looks when you build and run it outside of the editor as a standalone or web player. This section will explain how to access the **Build Settings** and how to create different builds of your games.

**File->Build Settings...** is the menu item to access the Build Settings window. It pops up an editable list of the scenes that will be included when you build your game.

*The Build Settings window*

The first time you view this window in a project, it will appear blank. If you build your game while this list is blank, only the currently open scene will be included in the build. If you want to quickly build a test player with only one scene file, just build a player with a blank scene list.

It is easy to add scene files to the list for multi-scene builds. There are two ways to add them. The first way is to click the **Add Current** button. You will see the currently open scene appear in the list. The second way to add scene files is to drag them from the **Project View** to the list.

At this point, notice that each of your scenes has a different index value. **Scene 0** is the first scene that will be loaded when you build the game. When you want to load a new scene, use Application.LoadLevel() inside your scripts.

If you've added more than one scene file and want to rearrange them, simply click and drag the scenes on the list above or below others until you have them in the desired order.

If you want to remove a scene from the list, click to highlight the scene and press **Command-Delete**. The scene will disappear from the list and will not be included in the build.

When you are ready to publish your build, select a **Platform** and make sure that the Unity logo is next to the platform; if its not then click in the **Switch Platform** button to let Unity know which platform you want to build for. Finally press the **Build** button. You will be able to select a name and location for the game using a standard Save dialog. When you click **Save**, Unity builds your game pronto. It's that simple. If you are unsure where to save your built game to, consider saving it into the projects root folder. You cannot save the build into the Assets folder.

Enabling the **Development Build** checkbox on a player will enable Profiler functionality and also make the Autoconnect Profiler and Script Debugging options available.

# Desktop

### Web Player Streaming

Streaming Web Players allow your Web Player games to begin playing as soon as Scene 0 is finished loading. If you have a game with 10 levels, it doesn't make much sense to force the player to wait and download all assets for levels 2-10 before they can start playing level 1. When you publish a Streaming Web Player, the assets that must be downloaded will be sequenced in the order of the **Scene** file they appear in. As soon as all assets contained in Scene 0 are finished downloading, the Web Player will begin playing.

Put simply, Streaming Web Players will get players playing your game faster than ever.

The only thing you need to worry about is checking to make sure that the next level you want to load is finished streaming before you load it.

Normally, in a non-streamed player, you use the following code to load a level:

```
Application.LoadLevel("levelName");
```

In a Streaming Web Player, you must first check that the level is finished streaming. This is done through the CanStreamedLevelBeLoaded() function. This is how it works:

```
var levelToLoad = 1;

function LoadNewLevel () {
	if (Application.CanStreamedLevelBeLoaded (levelToLoad)) {
		Application.LoadLevel (levelToLoad);
	}
}
```

If you would like to display the level streaming progress to the player, for a loading bar or other representation, you can read the progress by accessing GetStreamProgressForLevel().

### Offline webplayer deployment

If the Offline Deployment option is enabled for a webplayer then the UnityObject.js file (used to interface the player with the host page) will be placed alongside the player during the build. This enables the player to work with the local script file even when there is no network connection; normally, UnityObject.js is downloaded from Unity's webserver so as to make use of the latest version.

### Building standalone players

With Unity you can build standalone applications for Windows and Mac (Intel, PowerPC or Universal, which runs on both architectures). It's simply a matter of choosing the build target in the build settings dialog, and hitting the 'Build' button. When building standalone players, the resulting files will vary depending on the build target. On Windows an executable file (.exe) will be built, along with a Data folder which contains all the resources for your application. On Mac an app bundle will be built, containing the file needed to run the application, as well as the resources.

Distributing your standalone on Mac is just to provide the app bundle (everything is packed in there). On Windows you need to provide both the .exe file and the Data folder for others to run it. Think of it like this: Other people must have the same files on their computer, as the resulting files that Unity builds for you, in order to run your game.

### Inside the build process

The building process will place a blank copy of the built game application wherever you specify. Then it will work through the scene list in the build settings, open them in the editor one at a time, optimize them, and integrate them into the application package. It will also calculate all the assets that are required by the included scenes and store that data in a separate file within the application package.

- Any **GameObject** in a scene that is tagged with 'EditorOnly' will be not be included in the published build. This is useful for debugging scripts that don't need to be included in the final game.

- When a new level loads, all the objects in the previous level are destroyed. To prevent this, use DontDestroyOnLoad() on any objects you don't want destroyed. This is most commonly used for keeping music playing while loading a level, or for game controller scripts which keep game state and progress.
- After the loading of a new level is finished, the message: OnLevelWasLoaded() will be sent to all active game objects.
- For more information on how to best create a game with multiple scenes, for instance a main menu, a high-score screen, and actual game levels, see the Scripting Tutorial.pdf

## iOS

### Inside the iOS build process

The iPhone/iPad application build process is a two step process:

1. XCode project is generated with all the required libraries, precompiled .NET code and serialized assets.
2. XCode project is built and deployed on the actual device.

When "Build" is hit on "Build settings" dialog only the first step is accomplished. Hitting "Build and Run" performs both steps. If in the project save dialog the user selects an already existing folder an alert is displayed. Currently there are two XCode project generation modes to select:

- **replace** - all the files from target folder are removed and the new content is generated
- **append** - the "Data", "Libraries" and project root folder are cleaned and filled with newly generated content. The XCode project file is updated according to the latest Unity project changes. XCode project "Classes" subfolder could be considered as safe place to place custom native code, but making regular backups is recommended. Append mode is supported only for the existing XCode projects generated with the same Unity iOS version.

If Cmd+B is hit then the automatic build and run process is invoked and the latest used folder is assumed as the build target. In this case **append** mode is assumed as default.

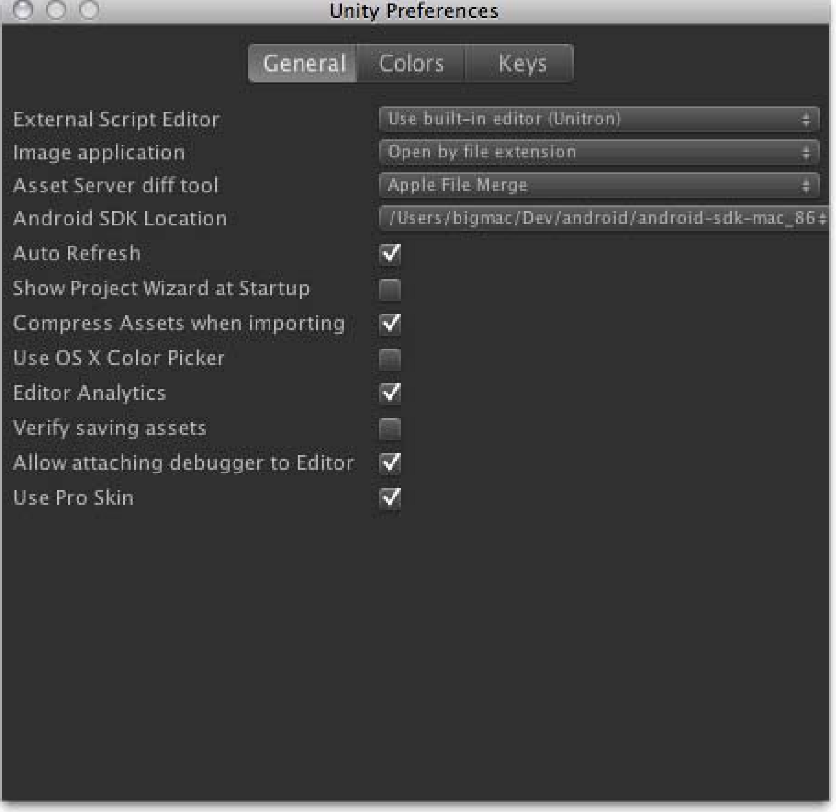
## Android

The Android application build process is performed in two steps:

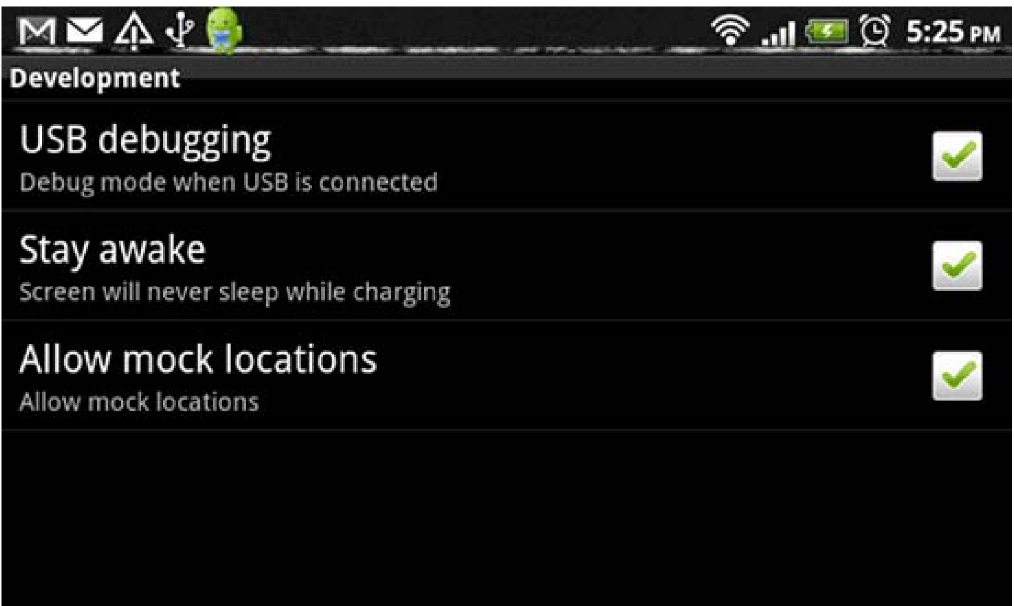
1. Application package (.apk file) is generated with all the required libraries and serialized assets.
2. Application package is deployed on the actual device.

When "Build" is hit on "Build settings" dialog only the first step is accomplished. Hitting "Build and Run" performs both steps. If Cmd+B is hit then the automatic build and run process is invoked and the latest used file is assumed as the build target.

Upon the first attempt to build an Android project, Unity would ask you to locate the Android SDK, that is required to build and install your Android application on the device. You can change this setting later in **Preferences**.




When building the app to the Android, be sure that the device has the "USB Debugging" and the "Allow mock locations" checkboxes checked in the device settings.




You can ensure that the operating system sees your device by running `adb devices` command found in your `Android SDK/platform-tools` folder. This should work both for Mac and Windows.

```
Terminal — bash — 72×8
Last login: Mon Feb 14 17:51:01 on ttys000
iMac-22:~ bigmac$ cd Dev/android/android-sdk-mac_86/platform-tools/
iMac-22:platform-tools bigmac$ ./adb devices
List of devices attached
HT0AGRT01900    device

iMac-22:platform-tools bigmac$
```

Unity builds an application archive (.apk file) for you and installs it on the connected device. In some cases your application cannot autostart like on iPhone, so you need to unlock the screen, and in some rare cases find the newly installed application in the menu.

### Texture Compression

Under **Build Settings** you'll also find the **Texture Compression** option. By default, Unity uses ETC1/RGBA16 texture format for textures that don't have individual texture format overrides (see Texture 2D / Per-Platform Overrides).

If you want to build an application archive (.apk file) targeting a specific hardware architecture, you can use the **Texture Compression** option to override this default behavior. Any texture that is set to not be compressed will be left alone; only textures using a compressed texture format will use the format selected in the **Texture Compression** option.

To make sure the application is only deployed on devices which support the selected texture compression, Unity will edit the AndroidManifest to include tags matching the particular format selected. This will enable the Android Market filtering mechanism to only serve the application to devices with the appropriate graphics hardware.

### Preloading

Published builds automatically preload all assets in a scene when the scene loads. The exception to this rule is scene 0. This is because the first scene is usually a splashscreen, which you want to display as quickly as possible.

To make sure all your content is preloaded, you can create an empty scene which calls **Application.LoadLevel(1)**. In the build settings make this empty scene's index 0. All subsequent levels will be preloaded.

### You're ready to build games

By now, you have learned how to use Unity's interface, how to use assets, how to create scenes, and how to publish your builds. There is nothing stopping you from creating the game of your dreams. You'll certainly learn much more along the way, and we're here to help.

To learn more details about using Unity itself, you can continue reading the manual or follow the Tutorials.

To learn more about Components, the nuts & bolts of game behaviors, please read the Component Reference.

To learn more about Scripting, please read the Scripting Reference.

To learn more about creating Art assets, please read the Assets section of the manual.

To interact with the community of Unity users and developers, visit the Unity Forums. You can ask questions, share projects, build a team, anything you want to do. Definitely visit the forums at least once, because we want to see the amazing games that you make.

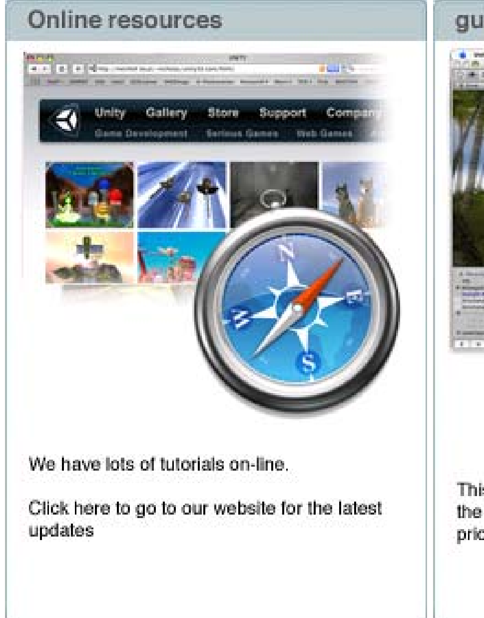
Page last updated: 2011-10-31

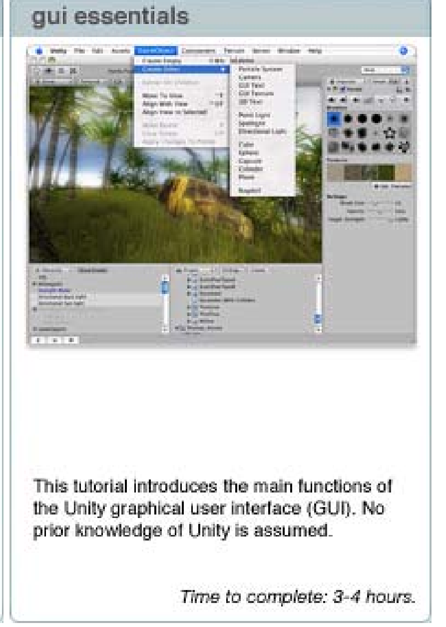
---

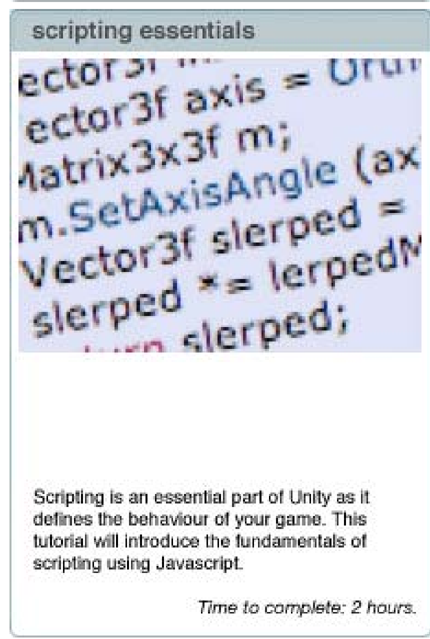
## Tutorials

These tutorials will let you work with Unity while you follow along. They will give you hands-on experience with building real projects. For new users, it is recommended that you follow the GUI Essentials and Scripting Essentials tutorials first. After that, you can follow any of them. They are all in PDF format, so you can print them out and follow along or read them alongside Unity.

**Note:** These Tutorials are intended for use with the Desktop version of Unity, these will not work with Android or iOS devices (iPhone/iPad).




Also if you are searching for other resources like presentations, articles, assets or extensions for Unity, then you can find them here.

You can also check the latest additions about tutorials just by checking our Unity3D Tutorial's Home Page.

Page last updated: 2010-09-10

---

## Unity Hotkeys

This page gives an overview of the default Unity Hotkeys. You can also download a PDF of the table for Windows and MacOSX. Where a command has *CTRL/CMD* as part of the keystroke, this indicates that the Control key should be used on Windows and the Command key on MacOSX.

**Tools**

| *Keystroke* | *Command* |
|---|---|
| Q | Pan |
| W | Move |
| E | Rotate |
| R | Scale |
| Z | Pivot Mode toggle |
| X | Pivot Rotation Toggle |
| V | Vertex Snap |
| CTRL/CMD+LMB | Snap |

**GameObject**

| | |
|---|---|
| CTRL/CMD+SHIFT+N | New game object |
| CTRL/CMD+ALT+F | Move to view |
| CTRL/CMD+SHIFT+F | Align with view |

**Window**

| | |
|---|---|
| CTRL/CMD+1 | Scene |
| CTRL/CMD+2 | Game |
| CTRL/CMD+3 | Inspector |
| CTRL/CMD+4 | Hierarchy |
| CTRL/CMD+5 | Project |
| CTRL/CMD+6 | Animation |
| CTRL/CMD+7 | Profiler |
| CTRL/CMD+9 | Asset store |
| CTRL/CMD+0 | Animation |
| CTRL/CMD+SHIFT+C | Console |

**Edit**

| | |
|---|---|
| CTRL/CMD+Z | Undo |
| CTRL+Y (Windows only) | Redo |
| CMD+SHIFT+Z (Mac only) | Redo |
| CTRL/CMD+X | Cut |
| CTRL/CMD+C | Copy |
| CTRL/CMD+V | Paste |
| CTRL/CMD+D | Duplicate |
| SHIFT+Del | Delete |
| F | Frame (centre) selection |
| CTRL/CMD+F | Find |
| CTRL/CMD+A | Select All |

**Selection**

| | |
|---|---|
| CTRL/CMD+SHIFT+1 | Load Selection 1 |
| CTRL/CMD+SHIFT+2 | Load Selection 2 |
| CTRL/CMD+SHIFT+3 | Load Selection 3 |

| | |
|---|---|
| CTRL/CMD+SHIFT+4 | Load Selection 4 |
| CTRL/CMD+SHIFT+5 | Load Selection 5 |
| CTRL/CMD+SHIFT+6 | Load Selection 6 |
| CTRL/CMD+SHIFT+7 | Load Selection 7 |
| CTRL/CMD+SHIFT+8 | Load Selection 8 |
| CTRL/CMD+SHIFT+9 | Load Selection 9 |
| CTRL/CMD+ALT+1 | Save Selection 1 |
| CTRL/CMD+ALT+2 | Save Selection 2 |
| CTRL/CMD+ALT+3 | Save Selection 3 |
| CTRL/CMD+ALT+4 | Save Selection 4 |
| CTRL/CMD+ALT+5 | Save Selection 5 |
| CTRL/CMD+ALT+6 | Save Selection 6 |
| CTRL/CMD+ALT+7 | Save Selection 7 |
| CTRL/CMD+ALT+8 | Save Selection 8 |
| CTRL/CMD+ALT+9 | Save Selection 9 |

**Assets**

| | |
|---|---|
| CTRL/CMD+R | Refresh |

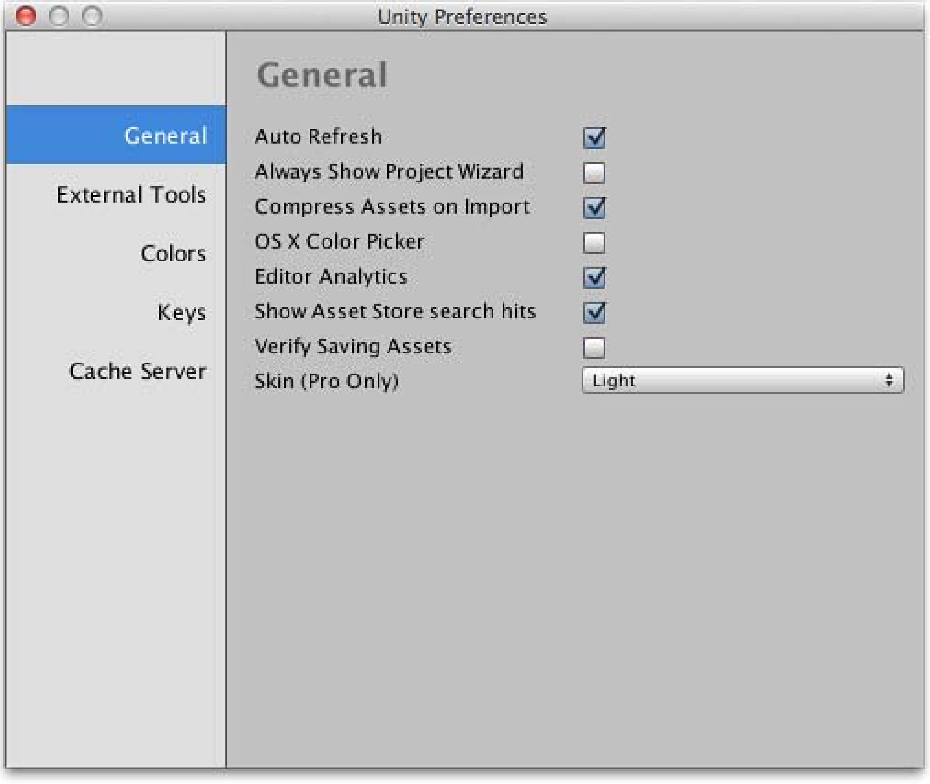
Page last updated: 2012-09-12

---

# Preferences

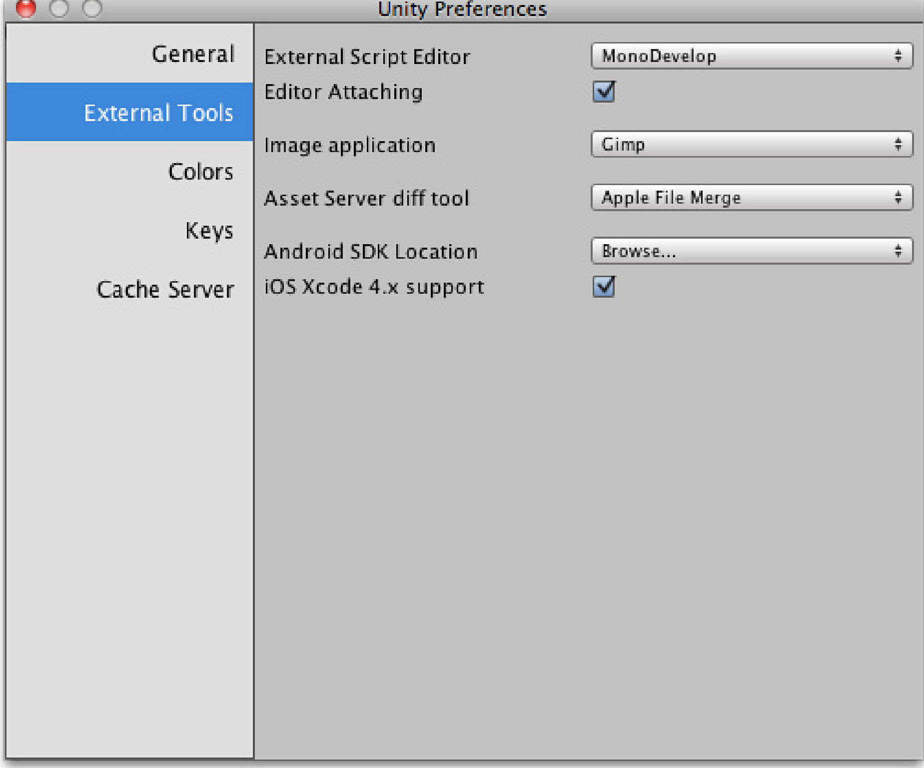
Unity provides a number of preference panels to allow you to customise the behaviour of the editor.

## General



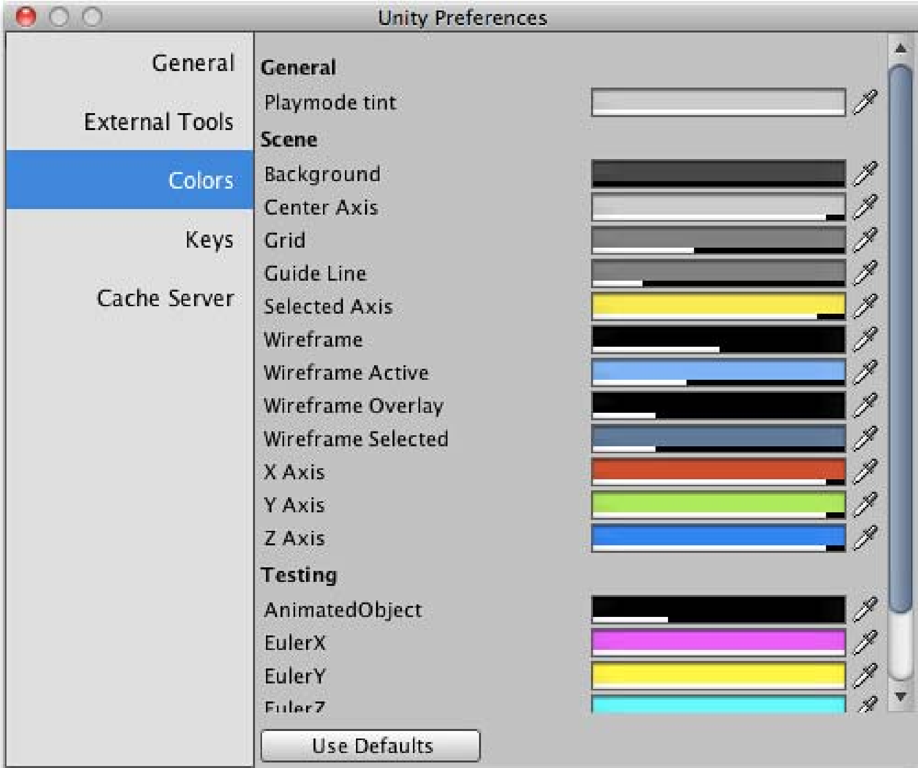

| | |
|---|---|
| **Auto Refresh** | Should the editor update assets automatically as they change? |
| **Always Show Project Wizard** | Should the project wizard be shown at startup? (By default, it is shown only when the alt key is held down during launch) |
| **Compress Assets On Import** | Should assets be compressed automatically during import? |
| **OSX Color Picker** | Should the native OSX color picker be used instead of Unity's own? |
| **Editor Analytics** | Can the editor send information back to Unity automatically? |
| **Show Asset Store search hits** | Should the number of free/paid assets from the store be shown in the Project Browser? |
| **Verify Saving Assets** | Should Unity verify which assets to save individually on quitting? |
| **Skin (Pro Only)** | Which color scheme should Unity use for the editor? Pro users have the option of dark grey in addition to the default light grey. |
| **Graphics Device** | This is set to Automatic on the Mac but has options for Direct3D 9, Direct3D 11 and OpenGL on Windows. |

## External Tools



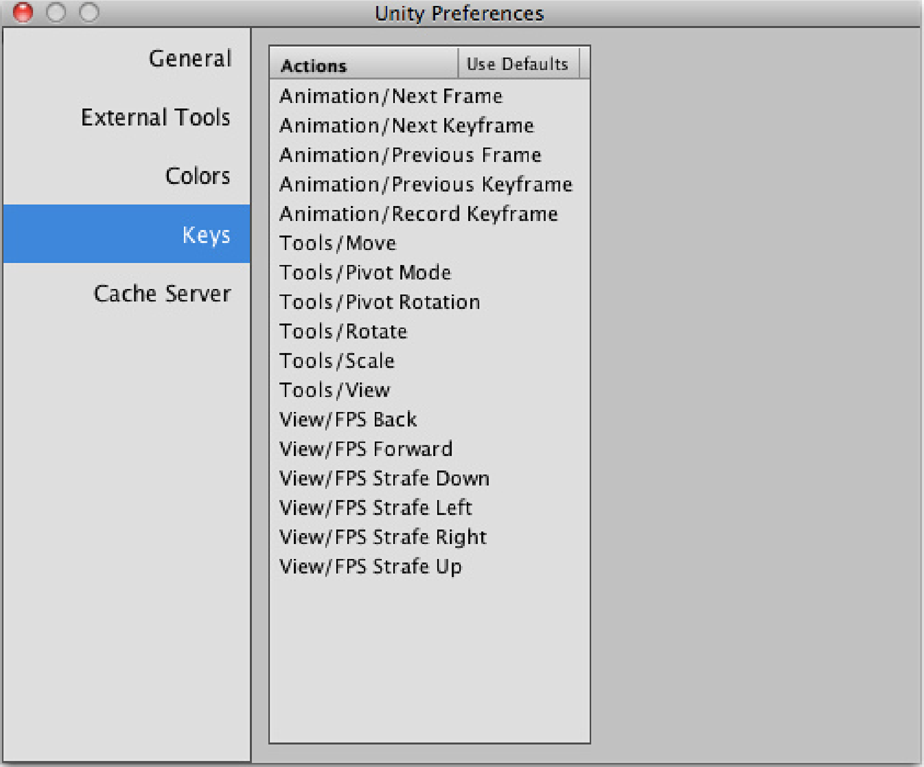

| | |
|---|---|
| **External Script Editor** | Which application should Unity use to open script files? |
| **Editor Attaching** | Should Unity allow debugging to be controlled from the external script editor? |
| **Image Application** | Which application should Unity use to open image files? |
| **Asset Server Diff Tool** | Which application should Unity use to resolve file differences with the asset server? |
| **Android SDK Location** | Where in the filesystem is the Android SDK folder located? |
| **iOS Xcode 4.x support** | Should support for Xcode 4.x be enabled for iOS build targets? |

## Colors



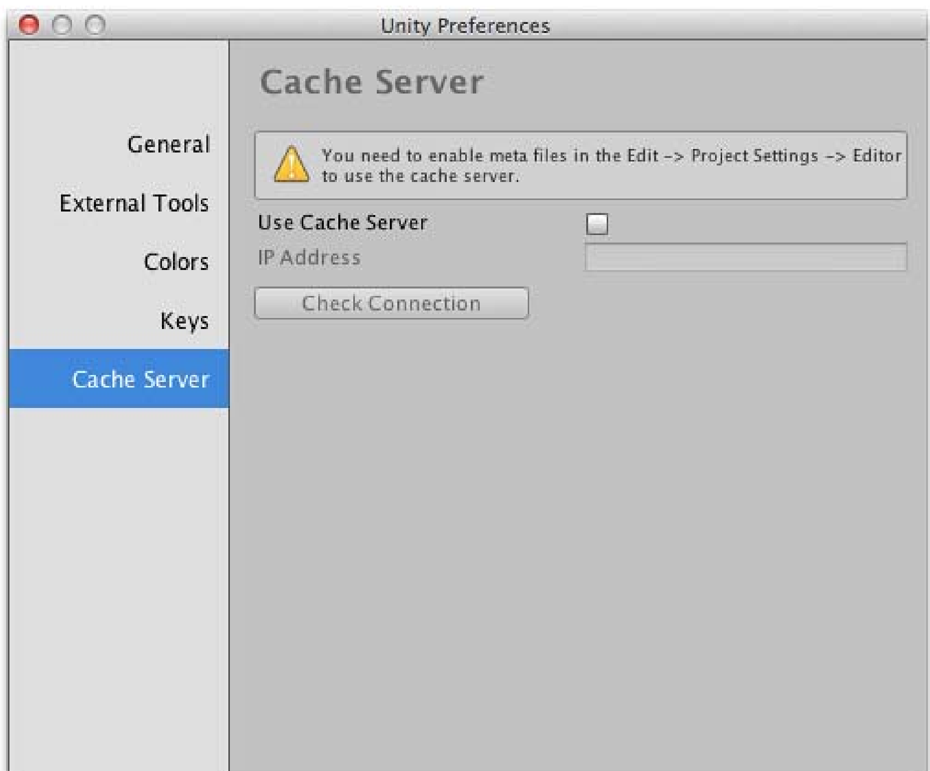

This panel allows you to choose the colors that Unity uses when displaying various user interface elements.

### Keys




This panel allows you to set the keystrokes that activate the various commands in Unity.

### Cache Server




| | |
|---|---|
| **Use Cache Server** | Should the cache server be enabled? |
| **IP Address** | IP address of the cache server, if enabled |

Page last updated: 2012-10-26

## Building Scenes

This section will explain the core elements you will work with to build scenes for complete games.


- GameObjects
  - The GameObject-Component Relationship
  - Using Components
  - The Component-Script Relationship
  - Deactivating GameObjects
- Using the Inspector
  - Editing Value Properties
  - Assigning References
  - Multi-Object Editing
  - Inspector Options
- Using the Scene View
  - Scene View Navigation
  - Positioning GameObjects
  - View Modes
  - Gizmo and Icon Display Controls
- Searching
- Prefabs
- Lights
- Cameras
- Terrain Engine Guide


Page last updated: 2007-11-16

## GameObjects

**GameObjects** are the most important objects in Unity. It is very important to understand what a GameObject is, and how it can be used. This page will explain all that for you.

### What are GameObjects?

Every object in your game is a GameObject. However, GameObjects don't do anything on their own. They need special properties before they can become a character, an environment, or a special effect. But every one of these objects does so many different things. If every object is a GameObject, how do we differentiate an interactive power-up object from a static room? What makes these GameObjects different from each other?

The answer to this question is that GameObjects are containers. They are empty boxes which can hold the different pieces that make up a lightmapped island or a physics-driven car. So to really understand GameObjects, you have to understand these pieces; they are called **Components**. Depending on what kind of object you want to create, you will add different combinations of Components to the GameObject. Think of a GameObject as an empty cooking pot, and Components as different ingredients that make up your recipe of gameplay. You can also make your own Components using Scripts.

You can read more about GameObjects, Components, and Script Components on the pages in this section:

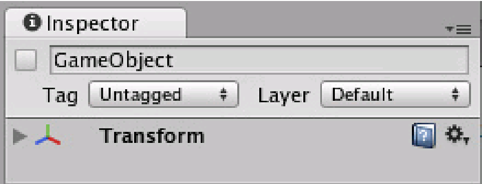

- The GameObject-Component Relationship
- Using Components
- The Component-Script Relationship
- Deactivating GameObjects


Page last updated: 2010-09-14

## The GameObject-Component Relationship

As described previously in GameObjects, a GameObject contains Components. We'll explore this relationship by discussing a

GameObject and its most common Component -- the **Transform** Component. With any Unity Scene open, create a new GameObject (using **Shift-Control-N** on Windows or **Shift-Command-N** on Mac), select it and take a look at the **Inspector**.




*The Inspector of an Empty GameObject*

Notice that an empty GameObject still contains a Name, a Tag, and a Layer. Every GameObject also contains a Transform Component.

## The Transform Component

It is impossible to create a GameObject in Unity without a Transform Component. The Transform Component is one of the most important Components, since all of the GameObject's Transform properties are enabled by its use of this Component. It defines the GameObject's position, rotation, and scale in the game world/Scene View. If a GameObject did not have a Transform Component, it would be nothing more than some information in the computer's memory. It effectively would not exist in the world.

The Transform Component also enables a concept called **Parenting**, which is utilized through the **Unity Editor** and is a critical part of working with GameObjects. To learn more about the Transform Component and Parenting, read the Transform Component Reference page.

## Other Components

The Transform Component is critical to all GameObjects, so each GameObject has one. But GameObjects can contain other Components as well.

*The Main Camera, added to each scene by default*

Looking at the Main Camera GameObject, you can see that it contains a different collection of Components. Specifically, a Camera Component, a GUILayer, a Flare Layer, and an Audio Listener. All of these Components provide additional functionality to the GameObject. Without them, there would be nothing rendering the graphics of the game for the person playing! Rigidbodies, Colliders, Particles, and Audio are all different Components (or combinations of Components) that can be

added to any given GameObject.

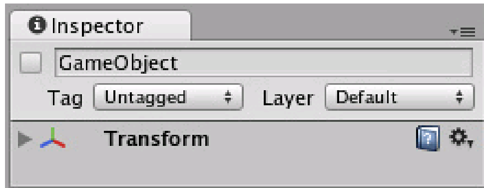
Page last updated: 2012-08-13

---

## Using Components40

**Components** are the nuts & bolts of objects and behaviors in a game. They are the functional pieces of every **GameObject**. If you don't yet understand the relationship between Components and GameObjects, read the GameObjects page before going any further.

A GameObject is a container for many different Components. By default, all GameObjects automatically have a **Transform** Component. This is because the Transform dictates where the GameObject is located, and how it is rotated and scaled. Without a Transform Component, the GameObject wouldn't have a location in the world. Try creating an empty GameObject now as an example. Click the **GameObject->Create Empty** menu item. Select the new GameObject, and look at the **Inspector**.

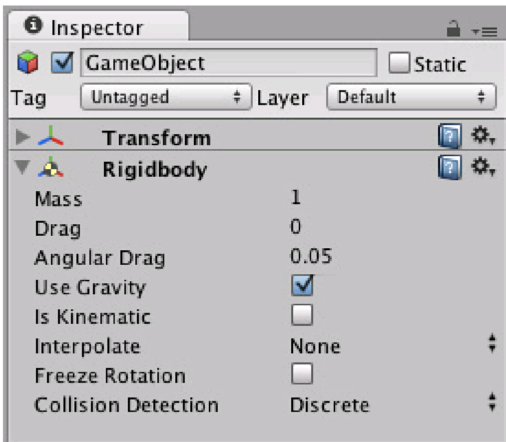



*Even empty GameObjects have a Transform Component*

Remember that you can always use the Inspector to see which Components are attached to the selected GameObject. As Components are added and removed, the Inspector will always show you which ones are currently attached. You will use the Inspector to change all the properties of any Component (including scripts)

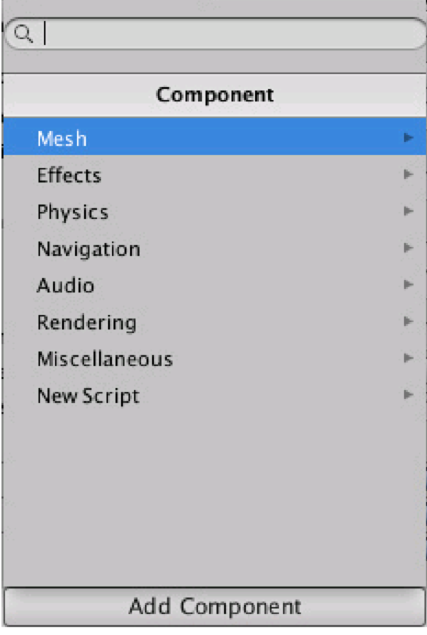
### Adding Components

You can add Components to the selected GameObject through the Components menu. We'll try this now by adding a **Rigidbody** to the empty GameObject we just created. Select it and choose **Component->Physics->Rigidbody** from the menu. When you do, you will see the Rigidbody's properties appear in the Inspector. If you press **Play** while the empty GameObject is still selected, you might get a little surprise. Try it and notice how the Rigidbody has added functionality to the otherwise empty GameObject. (The y-component of the GameObject starts to decrease. This is because the physics engine in Unity is causing the GameObject to fall under gravity.)




*An empty GameObject with a Rigidbody Component attached*

Another option is to use the **Component Browser**, which can be activated with the **Add Component** button in the object's inspector.




The browser lets you navigate the components conveniently by category and also has a search box that you can use to locate components by name.

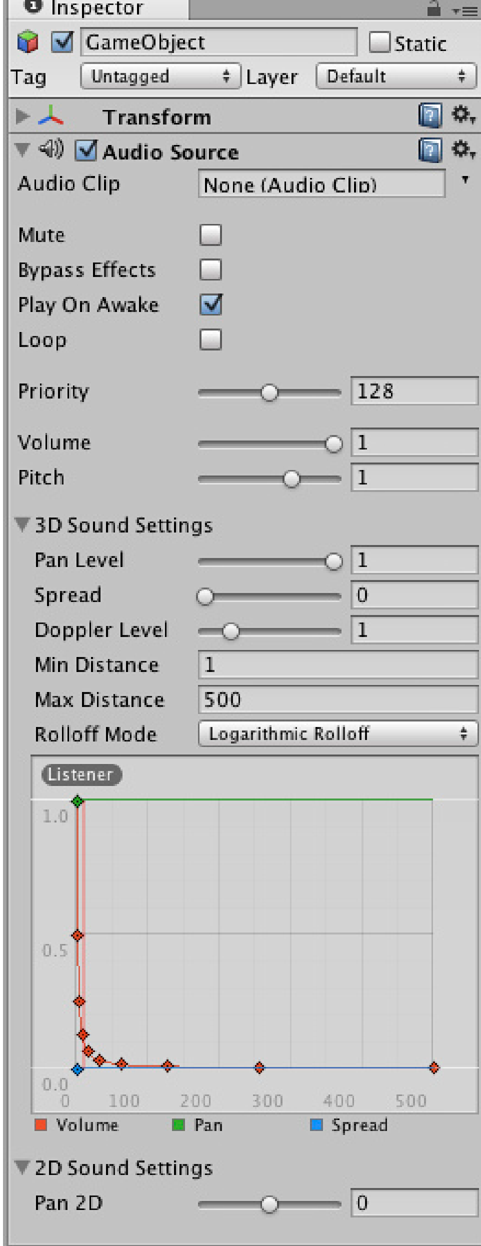
You can attach any number or combination of Components to a single GameObject. Some Components work best in combination with others. For example, the Rigidbody works with any Collider. The Rigidbody controls the Transform through the **NVIDIA PhysX** physics engine, and the Collider allows the Rigidbody to collide and interact with other Colliders.

If you want to know more about using a particular Component, you can read about any of them in the Component Reference. You can also access the reference page for a Component from Unity by clicking on the small **?** on the Component's header in the Inspector.

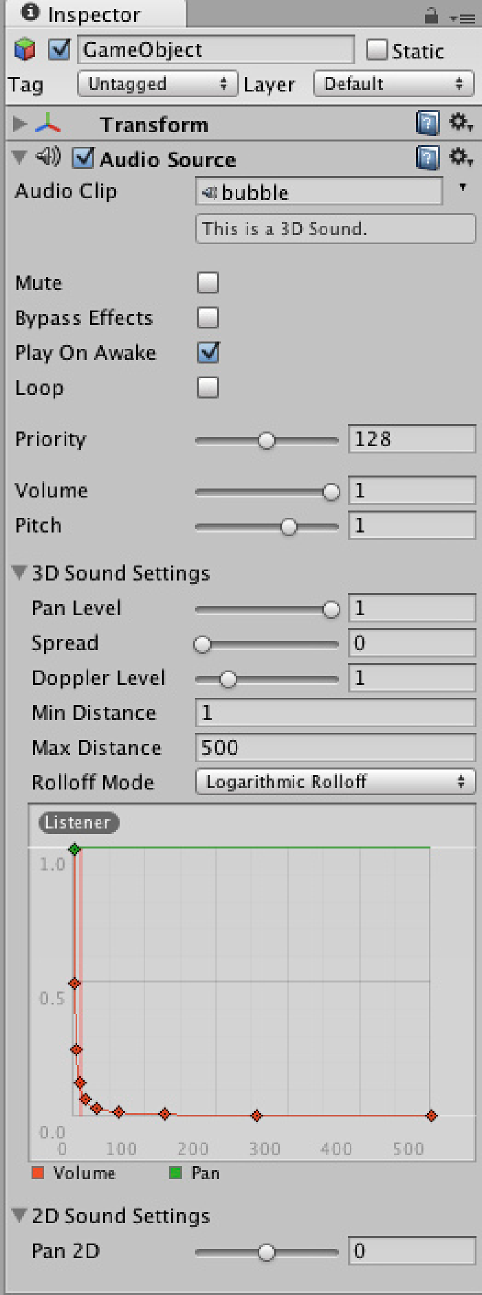
## Editing Components

One of the great aspects of Components is flexibility. When you attach a Component to a GameObject, there are different values or **Properties** in the Component that can be adjusted in the editor while building a game, or by scripts when running the game. There are two main types of Properties: **Values** and **References**.

Look at the image below. It is an empty GameObject with an **Audio Source** Component. All the values of the **Audio Source** in the Inspector are the default values.




This Component contains a single Reference property, and seven Value properties. **Audio Clip** is the Reference property. When this Audio Source begins playing, it will attempt to play the audio file that is referenced in the **Audio Clip** property. If no reference is made, an error will occur because there is no audio to be played. You must reference the file within the Inspector. This is as easy as dragging an audio file from the Project View onto the Reference Property or using the Object Selector.



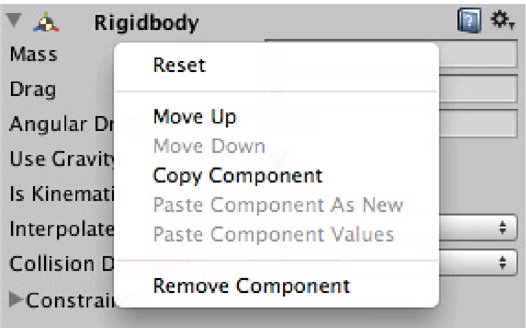
*Now a sound effect file is referenced in the **Audio Clip** property*

Components can include references to any other type of Component, GameObjects, or Assets. You can read more about assigning references on the Assigning References page.

The remaining properties on the Audio Clip are all Value properties. These can be adjusted directly in the Inspector. The Value properties on the Audio Clip are all toggles, numeric values, drop-down fields, but value properties can also be text strings, colors, curves, and other types. You can read more about these and about editing value properties on the Editing Value Properties page.

### Copying and pasting Component settings

The context menu for a Component has items for copying and pasting its settings.



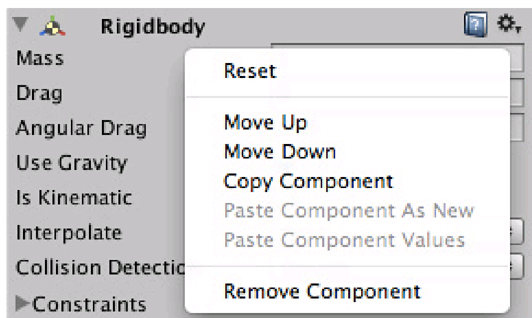

The copied values can be pasted to an existing component using the **Paste Component Values** menu item. Alternatively, you can use **Paste Component As New** to create a new Component with those values.

### Testing out Properties

While your game is in **Play Mode**, you are free to change properties in any GameObject's Inspector. For example, you might want to experiment with different heights of jumping. If you create a **Jump Height** property in a script, you can enter Play Mode, change the value, and press the jump button to see what happens. Then without exiting Play Mode you can change it again and see the results within seconds. When you exit Play Mode, your properties will revert to their pre-Play Mode values, so you don't lose any work. This workflow gives you incredible power to experiment, adjust, and refine your gameplay without investing a lot of time in iteration cycles. Try it out with any property in Play Mode. We think you'll be impressed.

### Changing the order of Components

The order in which components are listed in the Inspector doesn't matter in most cases. However, there are some Components, such as Image Effects where the ordering is significant. The context menu has **Move Up** and **Move Down** commands to let you reorder Components as necessary.




### Removing Components

If you want to remove a Component, option- or right-click on its header in the Inspector, and choose **Remove Component**. Or you can left-click the options icon next to the **?** on the Component header. All the property values will be lost and this cannot be undone, so be completely sure you want to remove the Component before you do.

Page last updated: 2012-09-11

---

## The Component-Script Relationship

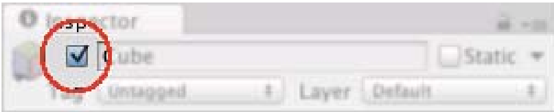
When you create a **script** and and attach it to a GameObject, the script appears in the GameObject's Inspector just like a Component. This is because scripts become Components when they are saved - a script is just a specific type of Component. In technical terms, a script compiles as a type of Component, and is treated like any other Component by the Unity engine. So basically, a script is a Component that you are creating yourself. You will define its members to be exposed in the Inspector, and it will execute whatever functionality you've written.

Read more about creating and using scripts on the Scripting page.

Page last updated: 2010-09-14

## DeactivatingGameObjects

A GameObject can be temporarily removed from the scene by marking it as inactive. This can be done using its activeSelf property from a script or with the activation checkbox in the inspector



*A GameObject's activation checkbox*

### Effect of deactivating a parent GameObject

When a parent object is deactivated, the deactivation also overrides the **activeSelf** setting on all its child objects, so the whole hierarchy from the parent down is made inactive. Note that this does *not* change the value of the **activeSelf** property on the child objects, so they will return to their original state once the parent is reactivated. This means that you can't determine whether or not a child object is currently active in the scene by reading its **activeSelf** property. Instead, you should use the activeInHierarchy property, which takes the overriding effect of the parent into account.

This overriding behaviour was introduced in Unity 4.0. In earlier versions, there was a function called **SetActiveRecursively** which could be used to activate or deactivate the children of a given parent object. However, this function worked differently in that the activation setting of each child object was changed - the whole hierarchy could be switched off and on but the child objects had no way to "remember" the state they were originally in. To avoid breaking legacy code, **SetActiveRecursively** has been kept in the API for 4.0 but its use is not recommended and it may be removed in the future. In the unusual case where you actually want the children's **activeSelf** settings to be changed, you can use code like the following:-

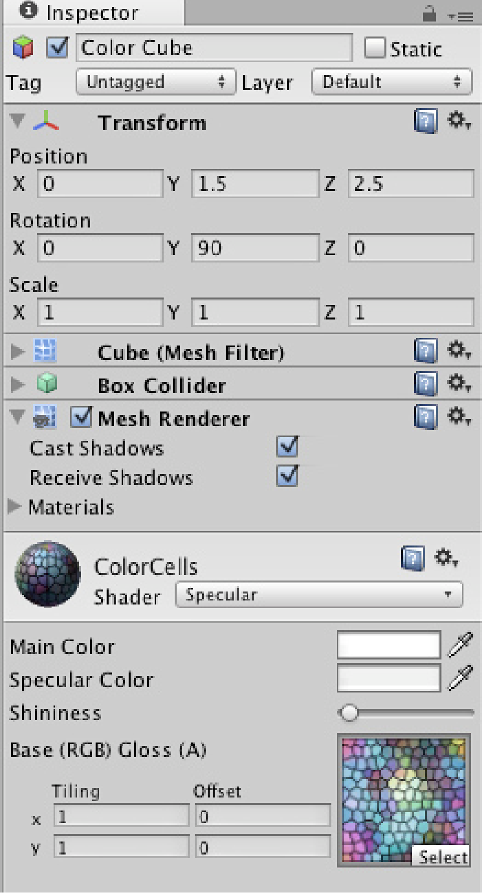
```
// JavaScript
function DeactivateChildren(g: GameObject, a: boolean) {
	g.activeSelf = a;

	for (var child: Transform in g.transform) {
		DeactivateChildren(child.gameObject, a);
	}
}


// C#
void DeactivateChildren(GameObject g, bool a) {
	g.activeSelf = a;

	foreach (Transform child in g.transform) {
		DeactivateChildren(child.gameObject, a);
	}
}
```

Page last updated: 2012-10-05

## Using The Inspector

The **Inspector** is used to view and edit **Properties** of many different types.

Games in Unity are made up of multiple **GameObjects** that contain meshes, scripts, sounds, or other graphical elements like Lights. When you select a **GameObject** in the Hierarchy or Scene View, the **Inspector** will show and let you modify the **Properties** of that GameObject and all the **Components** and **Materials** on it. The same will happen if you select a **Prefab** in the Project View. This way you modify the functionality of GameObjects in your game. You can read more about the GameObject-Component relationship, as it is very important to understand.




*Inspector shows the properties of a GameObject and the Components and Materials on it.*

When you create a **script** yourself, which works as a custom Component type, the member variables of that script are also exposed as **Properties** that can be edited directly in the Inspector when that script component has been added to a **GameObject**. This way script variables can be changed without modifying the script itself.

Furthermore, the Inspector is used for showing import options of assets such as textures, 3D models, and fonts when selected. Some scene and project-wide settings are also viewed in the **Inspector**, such as all the Settings Managers.

Any property that is displayed in the Inspector can be directly modified. There are two main types of Properties: **Values** and **References**.

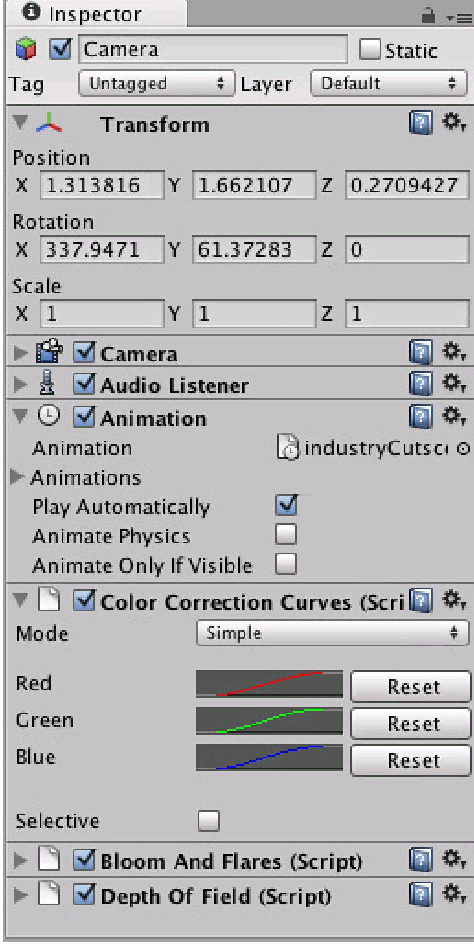
- Editing Value Properties
- Assigning References
- Multi-Object Editing
- Inspector Options

Page last updated: 2010-09-13

---

## Editing Value Properties40

Value properties do not reference anything and they can be edited right on the spot. Typical value properties are numbers, toggles, strings, and selection popups, but they can also be colors, vectors, curves, and other types.



*Value properties on the inspector can be numbers, checkboxes, strings...*

Many value properties have a text field and can be adjusted simply by clicking on them, entering a value using the keyboard, and pressing **Enter** to save the value.

- You can also put your mouse next to a numeric property, left-click and drag it to scroll values quickly
- Some numeric properties also have a slider that can be used to visually tweak the value.

Some Value Properties open up a small popup dialog that can be used to edit the value.

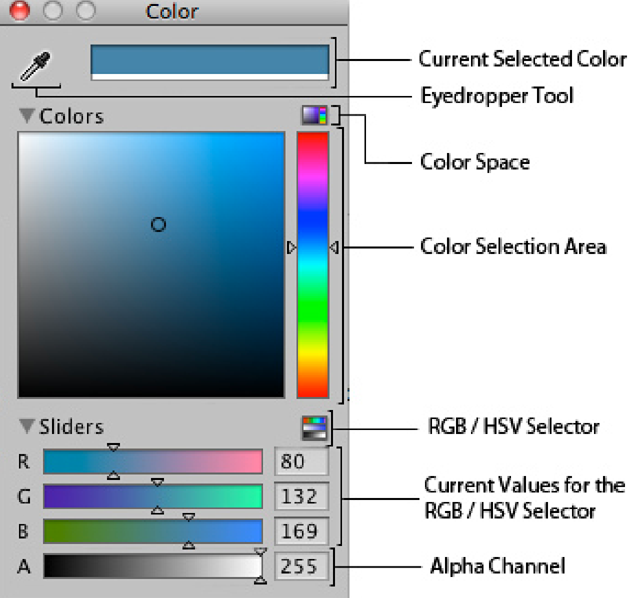
### Color Picker

Properties of the **Color** type will open up the **Color Picker**. (On Mac OS X this color picker can be changed to the native OS color picker by enabling **Use OS X Color Picker** under **Unity->Preferences**.)

The Color Picker reference in the inspector is represented by:

*Color Picker reference in the inspector.*

And opens the Color Picker just by clicking on it:




*Color Picker descriptions.*

Use the **Eyedropper Tool** when you want to find a value just by putting your mouse over the color you want to grab.
**RGB / HSV Selector** lets you switch your values from *Red, Green, Blue* to *Hue, Saturation and Value (Strength)* of your color.
Finally, the transparency of the Color selected can be controlled by the **Alpha Channel** value.

### Curve Editor

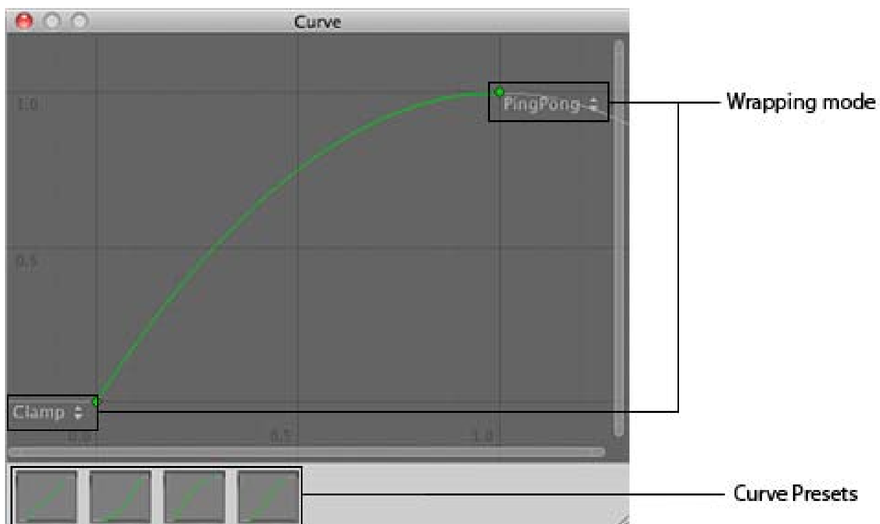
Properties of the **AnimationCurve** type will open up the **Curve Editor**. The Curve Editor lets you edit a curve or choose from one of the presets. For more information on editing curves, see the guide on Editing Curves.

The type is called AnimationCurve for legacy reasons, but it can be used to define any custom curve function. The function can then be evaluated at runtime from a script.

An AnimationCurve property is shown in the inspector as a small preview:

*A preview of an AnimationCurve in the Inspector.*

Clicking on it opens the Curve Editor:

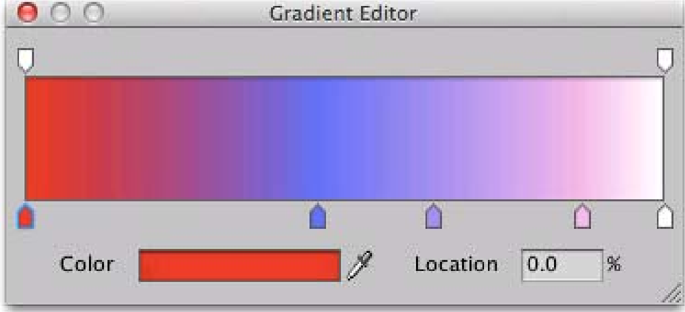



*The Curve Editor is for editing AnimationCurves.*

**Wrapping Mode** Lets you select between Ping Pong, Clamp and Loop for the Control Keys in your curve.
The **Presets** lets you modify your curve to default outlines the curves can have.

### Gradient editor

In graphics and animation, it is often useful to be able to blend one colour gradually into another, over space or time. A **gradient** is a visual representation of a colour progression, which simply shows the main colours (which are called **stops**) and all the intermediate shades between them. In Unity, gradients have their own special value editor, shown below.




The upward-pointing arrows along the bottom of the gradient bar denote the stops. You can select a stop by clicking on it; its value will be shown in the Color box which will open the standard colour picker when clicked. A new stop can be created by clicking just underneath the gradient bar. The position of any of the stops can be changed simply by clicking and dragging and a stop can be removed with **ctrl/cmd + delete**.

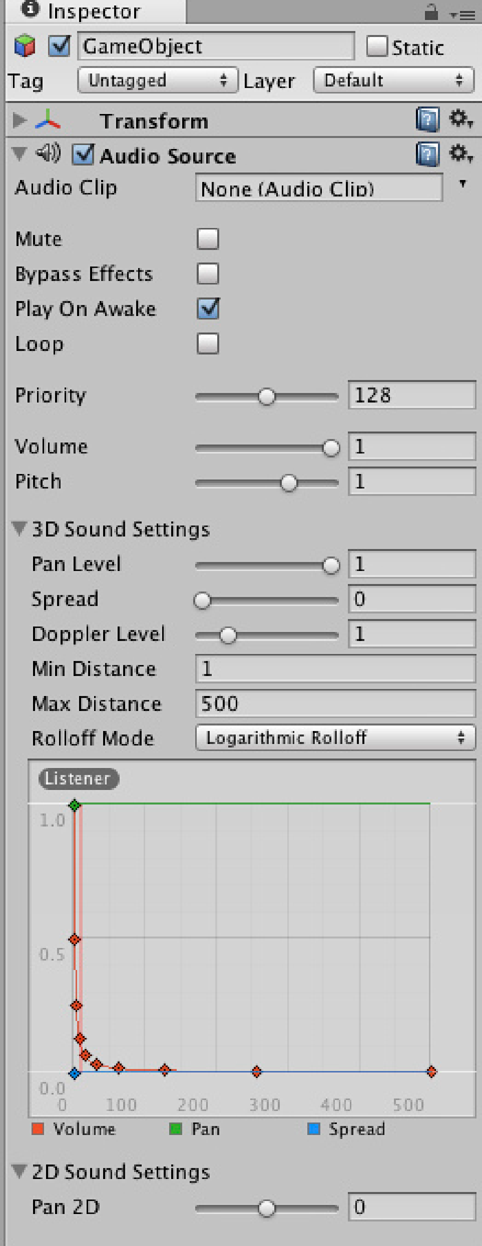
The downward-pointing arrows above the gradient bar are also stops but they correspond to the alpha (transparency) of the gradient at that point. By default, there are two stops set to 100% alpha (ie, fully opaque) but any number of stops can be added and edited in much the same way as the colour stops.

Page last updated: 2012-08-13

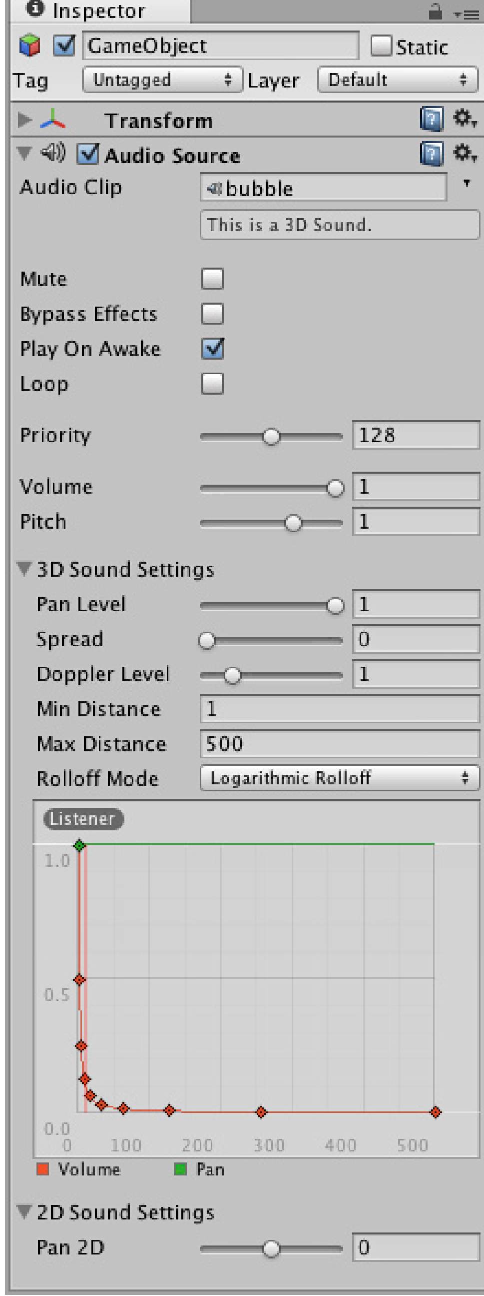
---

## Editing Reference Properties

Reference properties are properties that reference other objects such as GameObjects, Components, or Assets. The reference slot will show what kind of objects can be used for this reference.




*The **Audio Clip** property slot shows that it accept references to objects of type **Audio Clip***



*Now an **Audio Clip** file is referenced in the **Audio Clip** property.*

This type of referencing is very quick and powerful, especially when using scripting. To learn more about using scripts and properties, see the Scripting Tutorial on the Tutorials page.

Object references can be assigned to a reference property either by drag and drop or by using the **Object Picker**.

### Drag and Drop

You can use drag and drop simply by selecting the desired object in the Scene View, Hierarchy, or Project View and dragging it into the slot of the reference property.

If a reference property accepts a specific Component type (for example a Transform) then dragging a GameObject or a Prefab onto the reference property will work fine provided that the GameObject or Prefab contains a component of the correct type. The property will then reference the component in question, even though it was a GameObject or Prefab you dragged onto it.

If you drag an object onto an reference property, and the object is not of the correct type, or does not contain the right component, then you won't be able to assign the object to the reference property.

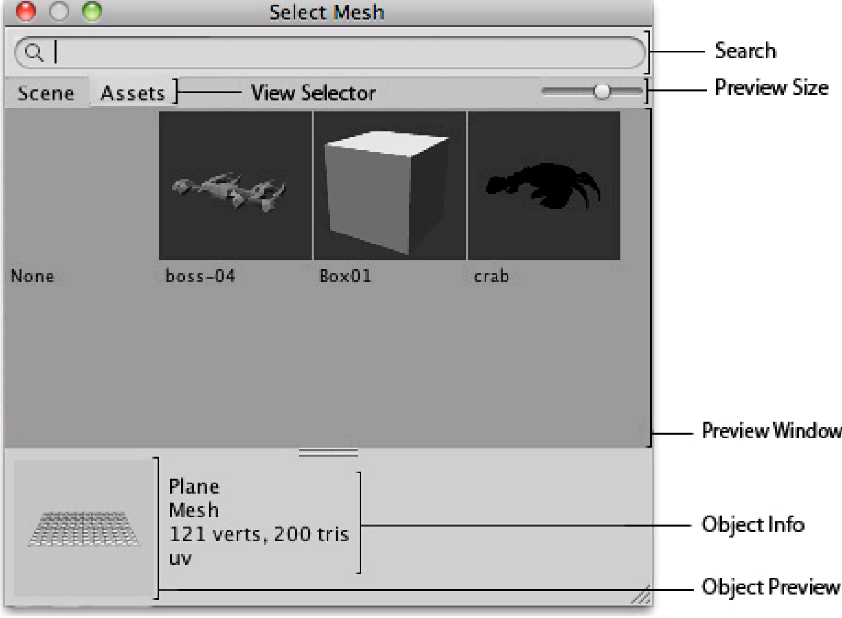
### The Object Picker

You can click on the small target icon next to a reference slot to open the Object Picker.

*References to the Object Picker from the Editor.*

The Object Picker is a simple window for assigning objects in the inspector after allowing you to preview and search those available.

Although the Object Picker is really easy to use, there are a few things you should be aware of. These are described below.




*Anatomy of the Object Picker.*

1. **Search:** When there are lots of objects in the picker, you can use the Search field to filter them. This search field can also search objects using their Labels.
2. **View Selector:** Switches the base of the search between objects in the scene and assets.
3. **Preview Size:** This horizontal scroll bar lets you increase/decrease the size of your preview objects in the preview window. With this you can see more or fewer objects in the preview window at any moment.
4. **Preview Window:** Here are all the objects that reside in your **Scene/Assets folder** filtered by the **Search** field.
5. **Object Info:** Displays information about the currently selected object. The content of this field depends on the type of object being viewed, so if for example you pick a mesh, it will tell you the number of vertices and triangles, and whether or not it has UVs and is skinned. However, if you pick an audio file it will give you information such as the bit rate of the audio, the length, etc.
6. **Object Preview:** This also depends on the type of object you are viewing. If you select a mesh, it will display you how the mesh looks, but if you select a script file, it will just display an icon of the file.

The Object Picker works on any asset you have in your project, which can be a video, a song, a terrain, a GUI skin, a scripting file, or a mesh; it is a tool you will use often.

**Hints**

- Use Labels on your Assets and you will be able to find them more easily by searching for them using the search field of the Object Picker.
- If you dont want to see the descriptions of the objects you can move the slider in the bottom middle of the preview window downward.
- If you want to see a detailed preview of the object, you can enlarge the object preview by dragging the slider in the bottom middle of the preview window.

Page last updated: 2012-08-13

---

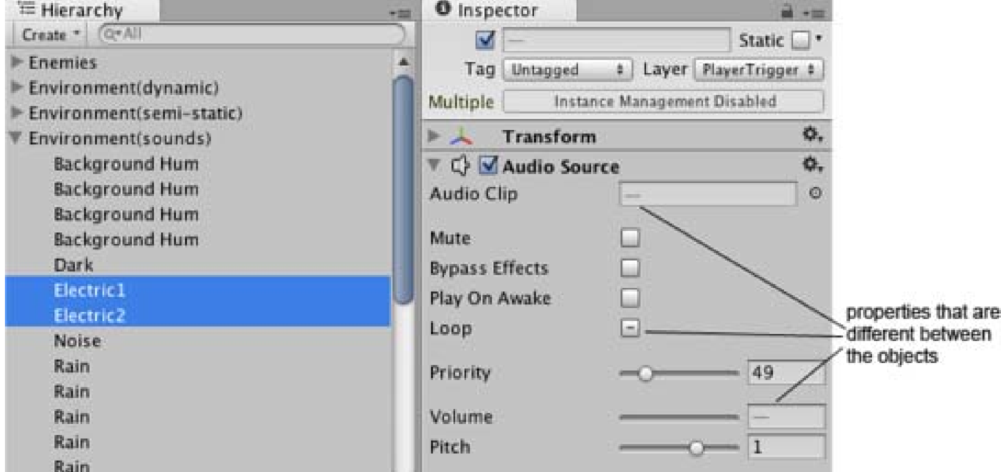
# Multi-Object Editing

Starting in Unity 3.5 you can select multiple objects of the same type and edit them simultaneously in the **Inspector**. Any changed properties will be applied to all of the selected objects. This is a big time saver if you want to make the same change to many objects.

When selecting multiple objects, a component is only shown in the **Inspector** if that component exists on all the selected objects. If it only exists on some of them, a small note will appear at the bottom of the Inspector saying that components that are only on some of the selected objects cannot be multi-edited.

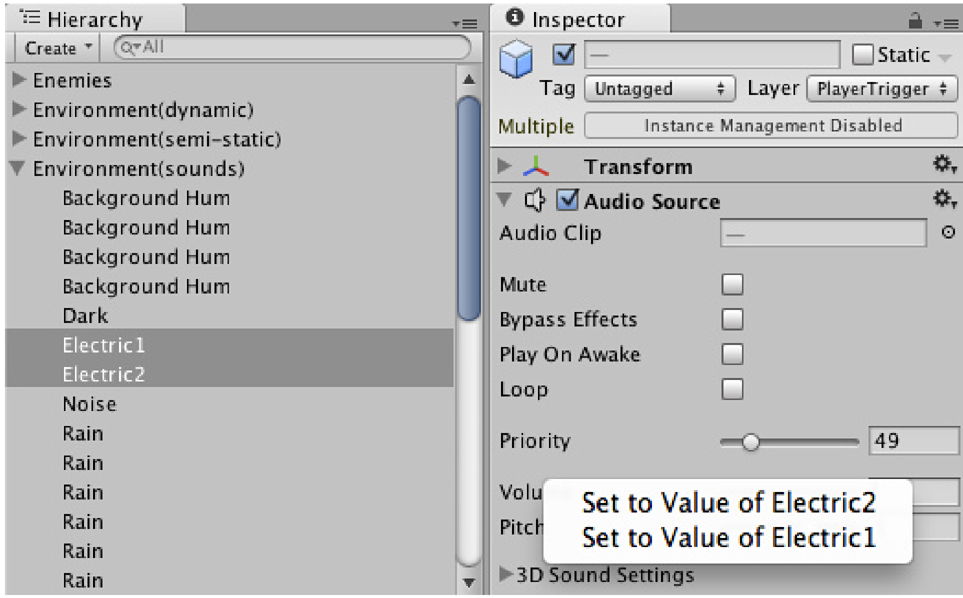
## Property Values

When multiple objects are selected, each property shown in the Inspector represents that property on each of the selected objects. If the value of the property is the same for all the objects, the value will be shown as normal, just like when editing a single object. If the value of the property is not the same for all the selected objects, no value is shown and a dash or similar is shown instead, indicating that the values are different.




*Multi-edit of two objects*

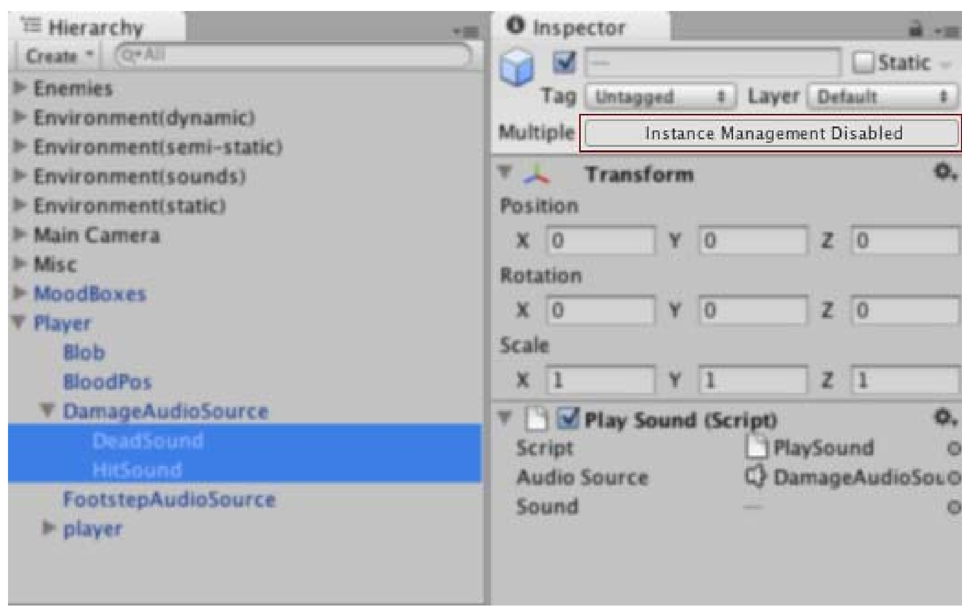
Regardless of whether a value is shown or a dash, the property value can be edited as usual and the changed value is applied to all the selected objects. If the values are different and a dash is thus shown, it's also possible to right-click on the label of the property. This brings up a menu that lets you choose from which of the objects to inherit the value.



*Selecting which object to get the value from*

## Multi-Editing Prefab or Model Instances

Prefabs can be multi-edited just like Game Objects in the scene. Instances of prefabs or of models can also be multi-edited; however certain restrictions apply: When editing a single prefab or model instance, any property that is different from the prefab or model will appear in bold, and when right clicking there's an option to revert the property to the value it has in the prefab or model. Furthermore, the Game Object has options to apply or revert all changes. None of these things are available when multi-object editing. Properties cannot be reverted or applied; nor will they appear in bold if different from the prefab or model. To remind you of this, the **Inspector** will show a note with *Instance Management Disabled* where the **Select**, **Revert**, and **Apply** buttons would normally appear.



*Instance Managment Disabled for multi-edit of prefabs*

## Non-Supported Objects

A few object types do not support multi-object editing. When you select multiple objects simultaneously, these objects will show a small note saying "Multi-object editing not supported".

If you have made a custom editor for one of your own scripts, it will also show this message if it doesn't support multi-object editing. See the script reference for the Editor class to learn how to implement support for multi-object editing for your own custom editors.

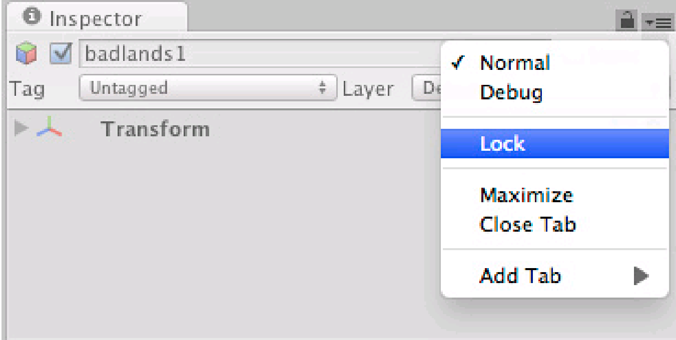
Page last updated: 2012-01-23

## Inspector Options

The Inspector Lock and the Inspector Debug Mode are two useful options that can help you in your workflow.

### Lock

The Lock lets you maintain focus on a specific GameObject in the Inspector while selecting other GameObjects. To toggle the lock of an Inspector click on the *lock/unlock* ( ) icon above the Inspector or open the tab menu and select **Lock**.



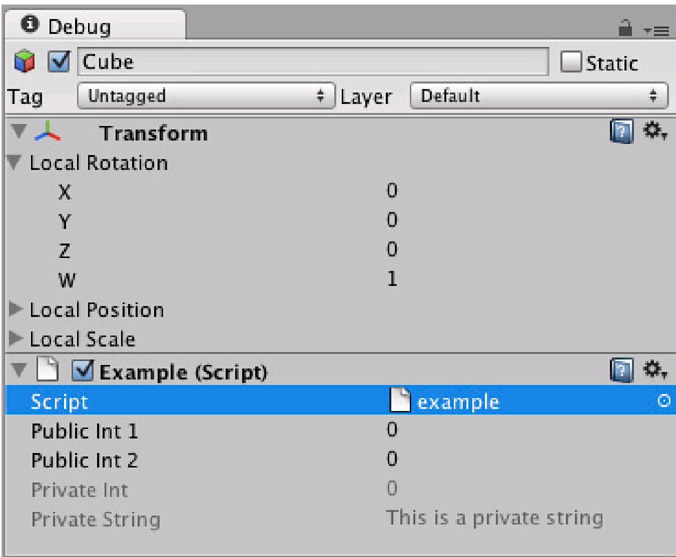

*Locking the Inspector from the tab menu.*

Note that you can have more than one Inspector open, and that you can for example lock one of them to a specific GameObject while keeping the other one unlocked to show whichever GameObject is selected.

### Debug

The Debug Mode lets you inspect private variables of components in the Inspector, which are normally not shown. To change to Debug Mode, open the tab menu and select **Debug**.

In Debug Mode, all components are shown using a default interface, rather than the custom interfaces that some components use in the Normal Mode. For example, the Transform component will in Debug Mode show the raw Quaternion values of the rotation rather than the Euler angles shown in the Normal Mode. You can also use the Debug Mode to inspect the values of private variables in your own script components.




*Debug Mode in the Inspector lets you inspect private variables in your scripts and in other components.*

The Debug mode is per Inspector and you can have one Inspector in Debug Mode while another one is not.

Page last updated: 2010-09-09

# Using The Scene View

The **Scene View** is your interactive sandbox. You will use the Scene View to select and position environments, the player, the camera, enemies, and all other **GameObjects**. Maneuvering and manipulating objects within the Scene View are some of the most important functions in Unity, so it's important to be able to do them quickly.


- Scene View Navigation
- Positioning GameObjects
- View Modes
- Gizmo and Icon Display Controls


Page last updated: 2010-09-06

---

# Scene View Navigation

The Scene View has a set of navigation controls to help you move around quickly and efficiently.

## Arrow Movement

You can use the **Arrow Keys** to move around the scene as though "walking" through it. The up and down arrows move the camera forward and backward in the direction it is facing. The left and right arrows pan the view sideways. Hold down the **Shift** key with an arrow to move faster.

## Focusing

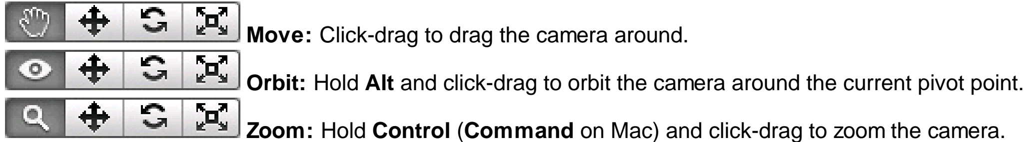
If you select a GameObject in the hierarchy, then move the mouse over the scene view and press the **F** key, the view will move so as to center on the object. This feature is referred to as **frame selection**.

## Move, Orbit and Zoom

Moving, orbiting and zooming are key operations in Scene View navigation, so Unity provides several alternative ways to perform them for maximum convenience.

### Using the Hand Tool

When the hand tool is selected (shortcut: **Q**), the following mouse controls are available:



**Move:** Click-drag to drag the camera around.

**Orbit:** Hold **Alt** and click-drag to orbit the camera around the current pivot point.

**Zoom:** Hold **Control** (**Command** on Mac) and click-drag to zoom the camera.

Holding down **Shift** will increase the rate of movement and zooming.

### Shortcuts Without Using the Hand Tool

For extra efficiency, all of these controls can also be used regardless of which transform tool is selected. The most convenient controls depend on which mouse or track-pad you are using:

| Action | 3-button mouse | 2-button mouse or track-pad | Mac with only one mouse button or track-pad |
|---|---|---|---|
| **Move** | Hold **Alt** and middle click-drag. | Hold **Alt-Control** and click-drag. | Hold **Alt-Command** and click-drag. |
| **Orbit** | Hold **Alt** and click-drag. | Hold **Alt** and click-drag. | Hold **Alt** and click-drag. |
| **Zoom** | Hold **Alt** and right click-drag or use scroll-wheel. | Hold **Alt** and right click-drag. | Hold **Alt-Control** and click-drag or use two-finger swipe. |

## Flythrough Mode

The **Flythrough** mode lets you navigate the Scene View by flying around in first person similar to how you would navigate in many games.

- Click and hold the right mouse button.
- Now you can move the view around using the mouse and use the **WASD** keys to move left/right forward/backward and the

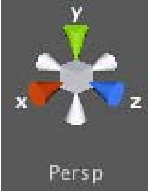
**Q** and **E** keys to move up and down.

- Holding down **Shift** will make you move faster.

Flythrough mode is designed for **Perspective Mode**. In **Isometric Mode**, holding down the right mouse button and moving the mouse will orbit the camera instead.

### Scene Gizmo

In the upper-right corner of the Scene View is the **Scene Gizmo**. This displays the Scene View Camera's current orientation, and allows you to quickly modify the viewing angle.




You can click on any of the arms to snap the Scene View Camera to that direction. Click the middle of the Scene Gizmo, or the text below it, to toggle between **Isometric Mode** and **Perspective Mode**. You can also always shift-click the middle of the Scene Gizmo to get a "nice" perspective view with an angle that is looking at the scene from the side and slightly from above.

*Perspective mode.*

*Isometric mode. Objects do not get smaller with distance here!*

**Mac Trackpad Gestures**
On a Mac with a trackpad, you can drag with two fingers to zoom the view.

You can also use three fingers to simulate the effect of clicking the arms of the **Scene Gizmo**: drag up, left, right or down to snap the Scene View Camera to the corresponding direction. In OS X 10.7 "Lion" you may have to change your trackpad settings in order to enable this feature:

- Open System Preferences and then Trackpad (or type trackpad into Spotlight).
- Click into the "More Gestures" option.
- Click the first option labelled "Swipe between pages" and then either set it to "Swipe left or right with three fingers" or "Swipe with two or three fingers".

Page last updated: 2012-11-16

---

# Positioning GameObjects

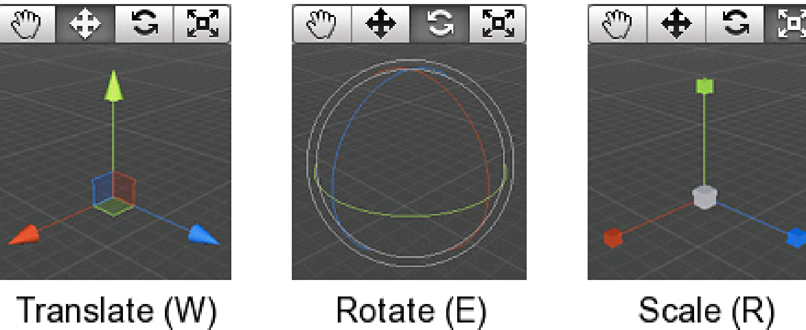
When building your games, you'll place lots of different objects in your game world.

## Focusing

It can be useful to focus the Scene View Camera on an object before manipulating it. Select any GameObject and press the **F** key. This will center the Scene View and pivot point on the selection. This is also known as Frame Selection.

## Translate, Rotate, and Scale

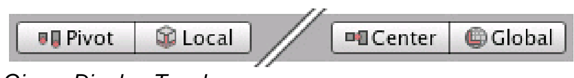
Use the Transform Tools in the Toolbar to Translate, Rotate, and Scale individual GameObjects. Each has a corresponding Gizmo that appears around the selected GameObject in the Scene View. You can use the mouse and manipulate any Gizmo axis to alter the **Transform** Component of the GameObject, or you can type values directly into the number fields of the Transform Component in the Inspector. Each of the three transform modes can be selected with a hotkey - **W** for Translate, **E** for Rotate and **R** for Scale.




- Click and drag in the center of the Gizmo to manipulate the object on all axes at once.
- At the center of the Translate gizmo, there are three small squares that can be used to drag the object within a single plane (ie, two axes can be moved at once while the third is kept still).
- If you have a three button mouse, you can click the middle button to adjust the last-adjusted axis (which turns yellow) without clicking directly on it.
- Be careful when using the scaling tool, as non-uniform scales (e.g. 1,2,1) can cause unusual scaling of child objects.
- For more information on transforming GameObjects, please view the Transform Component page.

### Gizmo Display Toggles

The **Gizmo Display Toggles** are used to define the location of any Transform Gizmo.



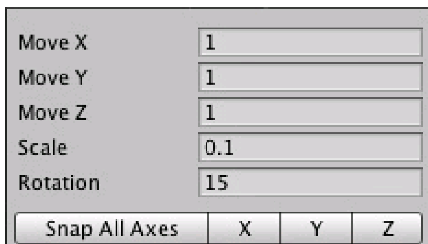

*Gizmo Display Toggles*

- Position:
  - **Center** will position the Gizmo at the center of the object's rendered bounds.
  - **Pivot** will position the Gizmo at the actual pivot point of a Mesh.
- Rotation:
  - **Local** will keep the Gizmo's rotation relative to the object's.
  - **Global** will clamp the Gizmo to world space orientation.

### Unit Snapping

While dragging any Gizmo Axis using the Translate Tool, you can hold the **Control** key (**Command** on Mac) to snap to increments defined in the **Snap Settings**.

You can change the unit distance that is used for the unit snapping using the menu **Edit->Snap Settings...**




*Scene View Unit Snapping settings.*

### Surface Snapping

While dragging in the center using the Translate Tool, you can hold **Shift** and **Control** (**Command** on Mac) to snap the object to the intersection of any **Collider**. This makes precise positioning of objects incredibly fast.

### Look-At Rotation

While using the Rotate Tool, you can hold **Shift** and **Control** (**Command** on Mac) to rotate the object towards a point on the surface of any **Collider**. This makes orientation of objects relative to one another simple.

### Vertex Snapping

You can assemble your worlds more easily with a feature called **Vertex Snapping**. This feature is a really simple but powerful

tool in Unity. It lets you take any vertex from a given mesh and with your mouse place that vertex in the same position as any vertex from any other mesh you choose.

With this you can assemble your worlds really fast. For example, you can place roads in a racing game with high precision and add power up items on the vertices of a mesh.

*Assembling roads with Vertex Snapping.*

Using vertex snapping in Unity is simple. Just follow these steps:

- Select the mesh you want to manipulate and make sure the Transform Tool is active.
- Press and **hold** the **V** key to activate the vertex snapping mode.
- Move your cursor over the vertex on your mesh that you want to use as the pivot point.
- **Hold** down the left button once your cursor is over the desired vertex and drag your mesh next to any other vertex on another mesh.
- Release your mouse button and the **V** key when you are happy with the results.
- **Shift-V** acts as a toggle of this functionality.
- You can snap vertex to vertex, vertex to surface and pivot to vertex.

A video on how to use vertex snapping can be found here.

Page last updated: 2011-10-26

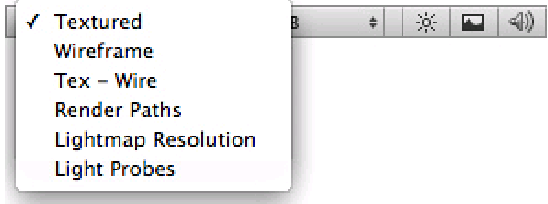
---

## View Modes

The Scene View control bar lets you choose various options for viewing the scene and also control whether lighting and audio are enabled. These controls only affect the scene view during development and have no effect on the built game.

### Draw Mode

The first drop-down menu selects which **Draw Mode** will be used to depict the scene.



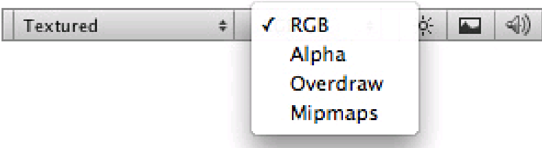

*Draw Mode drop-down*

- **Textured**: show surfaces with their textures visible.

- **Wireframe**: draw meshes with a wireframe representation.
- **Tex-Wire**: show meshes textured and with wireframes overlaid.
- **Render Paths**: show the rendering path for each object using a color code: Green indicates deferred lighting, yellow indicates forward rendering and red indicates vertex lit.
- **Lightmap Resolution**: overlay a checkered grid on the scene to show the resolution of the lightmaps.

### Render Mode

The next drop-down along selects which of four **Render Modes** will be used to render the scene.




*Render Mode drop-down*

- **RGB**: render the scene with objects normally colored.
- **Alpha**: render colors with alpha.
- **Overdraw**: render objects as transparent "silhouettes". The transparent colors accumulate, making it easy to spot places where one object is drawn over another.
- **Mipmaps:** show ideal texture sizes using a color code: red indicates that the texture is larger than necessary (at the current distance and resolution); blue indicates that the texture could be larger. Naturally, ideal texture sizes depend on the resolution at which the game will run and how close the camera can get to particular surfaces.

### Scene Lighting, Game Overlay, and Audition Mode

To the right of the dropdown menus are three buttons which control other aspects of the scene representation.

The first button determines whether the view will be lit using a default scheme or with the lights that have actually been added to the scene. The default scheme is used initially but this will change automatically when the first light is added. The second button controls whether skyboxes and GUI elements will be rendered in the scene view and also shows and hides the placement grid. The third button switches audio sources in the scene on and off.

Page last updated: 2011-11-10

---

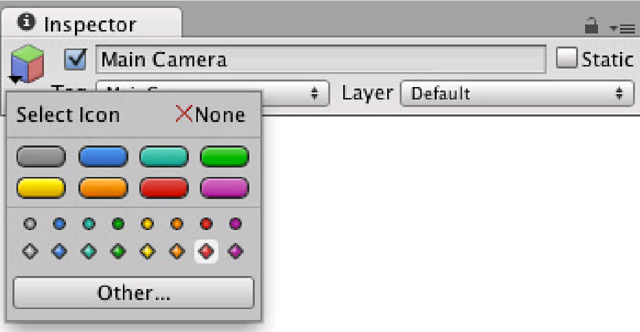
## Gizmo and Icon Visibility

Gizmos and icons have a few display options which can be used to reduce clutter and improve the visual clarity of the scene during development.

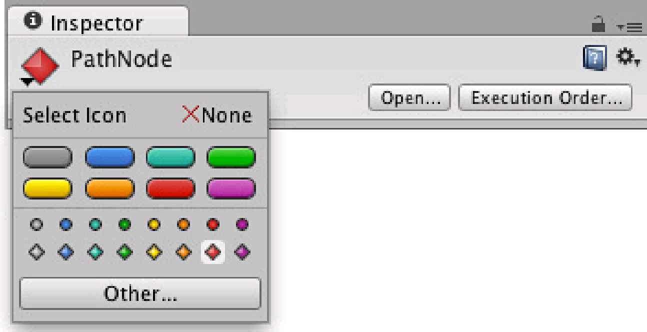
### The Icon Selector

Using the **Icon Selector**, you can easily set custom icons for GameObjects and scripts that will be used both in the Scene View and the Inspector. To change the icon for a GameObject, simply click on its icon in the Inspector. The icons of script assets can be changed in a similar way. In the Icon Selector is a special kind of icon called a **Label Icon**. This type of icon will show up in the Scene View as a text label using the name of the GameObject. Icons for built-in Components cannot be changed.

Note: When an asset's icon is changed, the asset will be marked as modified and therefore picked up by Revision Control Systems.




*Selecting an icon for a GameObject*




*Selecting an icon for a script*

### Showing and Hiding Icons and Gizmos

The visibility of an individual component's gizmos depends on whether the component is expanded or collapsed in the inspector (ie, collapsed components are invisible). However, you can use the **Gizmos dropdown** to expand or collapse every component of a given type at once. This is a useful way to reduce visual clutter when there are a large number of gizmos and icons in the scene.

To show the state of the current gizmo and icon, click on **Gizmos** in the control bar of the Scene or Game View. The toggles here are used to set which icons and gizmos are visible.

Note that the scripts that show up in the **Scripts** section are those that either have a custom icon or have an **OnDrawGizmos ()** or **OnDrawGizmosSelected ()** function implemented.

*The **Gizmos dropdown**, displaying the visibility state of icons and gizmos*

The **Icon Scaling** slider can be used to adjust the size used for icon display in the scene. If the slider is placed at the extreme right, the icon will always be drawn at its natural size. Otherwise, the icon will be scaled according to its distance from the scene view camera (although there is an upper limit on the display size in order that screen clutter be avoided).

Page last updated: 2011-11-09

---

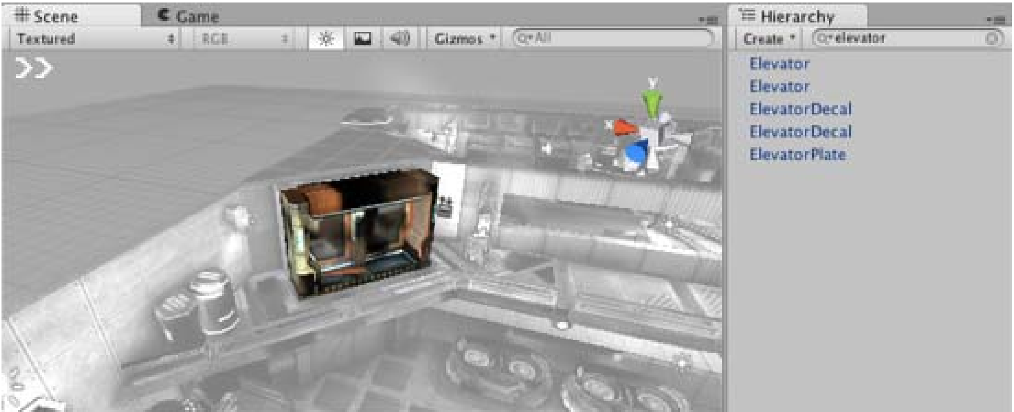
# Searching

When working with large complex scenes it can be useful to search for specific objects. By using the **Search** feature in Unity, you can filter out only the object or group of objects that you want to see. You can search assets by their name, by Component type, and in some cases by asset Labels. You can specify the search mode by choosing from the Search drop-down menu.

## Scene Search

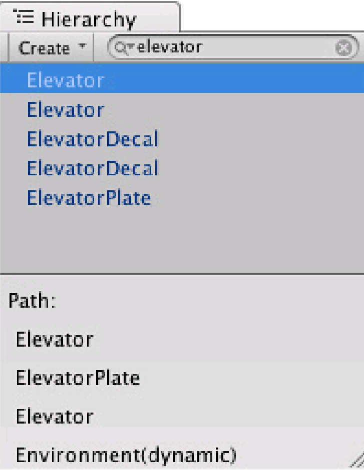
When a scene is loaded in the Editor, you can see the objects in both the Scene View and the Hierarchy. The specific assets are shared in both places, so if you type in a search term (eg, "elevator"), you'll see the the filter applied both visually in the Scene View and a more typical manner in the Hierarchy. There is also no difference between typing the search term into the search field in the Scene View or the Hierachy -- the filter takes effect in both views in either case.

*Scene View and Hierarchy with no search applied.*




*Scene View and Hierarchy with active filtering of search term.*

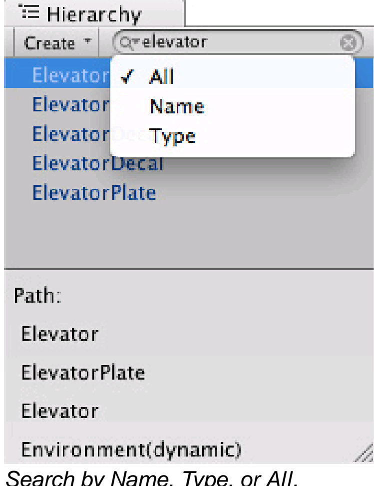
When a search term filter is active, the Hierarchy doesn't show hierarchical relationships between GameObjects, but you can select any GameObject, and it's hierarchical path in the scene will be shown at the bottom of the Hierarchy.




*Click on a GameObject in the filtered list to see its hierarchical path.*

When you want to clear the search filter, just click the small cross in the search field.

In the Scene search you can search either by Name or by Type. Click on the small magnifying glass in the search field to open the search drop-down menu and choose the search mode.

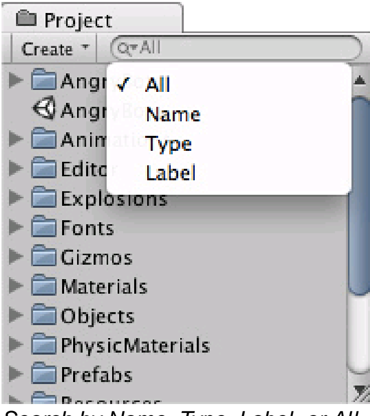



*Search by Name, Type, or All.*

## Project Search

The same fundamentals apply to searching of assets in the Project View -- just type in your search term and you'll see all the relevant assets appear in the filter.

In the Project search you can search by Name or by Type as in the Scene search, and additionally you can search by Label. Click on the small magnifying glass in the search field to open the search drop-down menu and choose the search mode.




*Search by Name, Type, Label, or All.*

### Object Picker Search

When assigning an object via the Object Picker, you can also enter a search term search to filter the objects you want to see.

Page last updated: 2011-11-10

---

# Prefabs

A **Prefab** is a type of asset -- a reusable **GameObject** stored in **Project View**. Prefabs can be inserted into any number of scenes, multiple times per scene. When you add a Prefab to a scene, you create an **instance** of it. All Prefab instances are linked to the original Prefab and are essentially clones of it. No matter how many instances exist in your project, when you make any changes to the Prefab you will see the change applied to all instances.

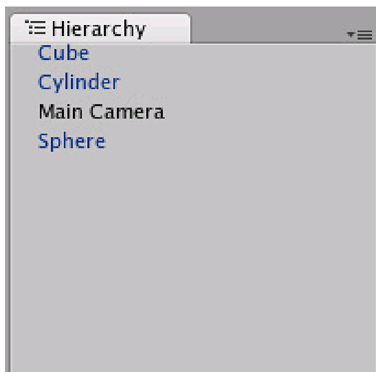
### Creating Prefabs

In order to create a Prefab, simply drag a GameObject that you've created in the scene into the Project View. The GameObject's name will turn blue to show that it is a Prefab. You can rename your new Prefab.

After you have performed these steps, the GameObject and all its children have been copied into the Prefab data. The Prefab can now be re-used in multiple instances. The original GameObject in the Hierarchy has now become an instance of the Prefab.

### Prefab Instances

To create a Prefab instance in the current scene, drag the Prefab from the Project View into the **Scene** or Hierarchy View. This instance is **linked** to the Prefab, as displayed by the blue text used for their name in the Hierarchy View.



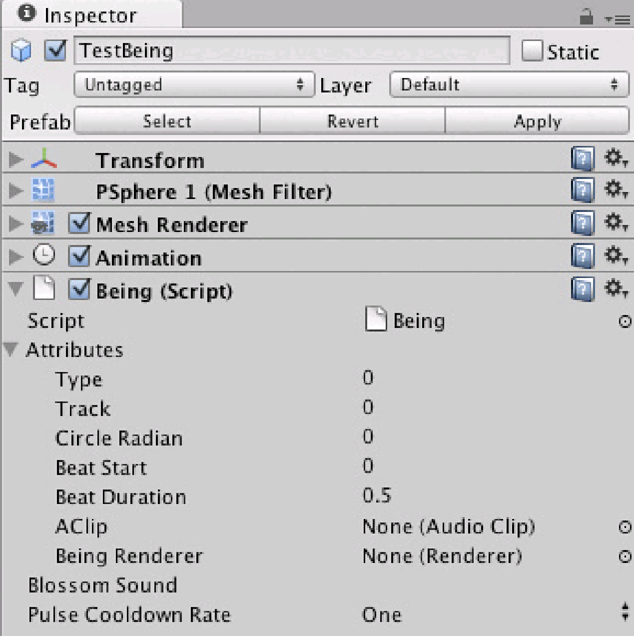

*Three of these GameObjects are linked to Prefabs. One of them is not.*

- If you have selected a Prefab instance, and want to make a change that affects all instances, you can click the **Select** button in the Inspector to select the source Prefab.
- Information about instantiating prefabs from scripts is in the Instantiating Prefabs page.

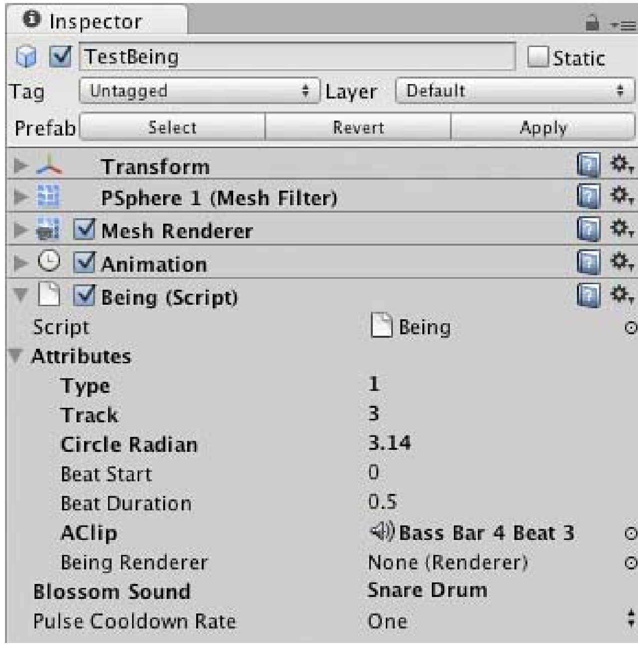
#### Inheritance

Inheritance means that whenever the source Prefab changes, those changes are applied to all linked GameObjects. For example, if you add a new script to a Prefab, all of the linked GameObjects will instantly contain the script as well. However, it is possible to change the properties of a single instance while keeping the link intact. Simply change any property of a prefab instance, and watch as the variable name becomes bold. The variable is now overridden. All overridden properties will not be affected by changes in the source Prefab.

This allows you to modify Prefab instances to make them unique from their source Prefabs without breaking the Prefab link.




*A linked GameObject with no overrides enabled.*

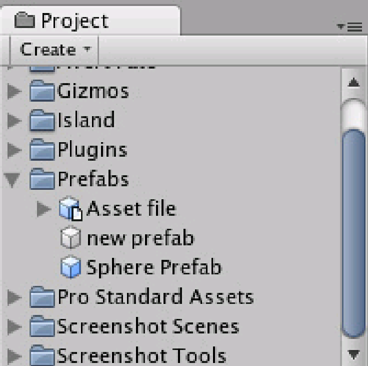



*A linked GameObject with several (bold) overrides enabled.*

- If you want to update the source Prefab and all instances with the new overridden values, you can click the **Apply** button in the Inspector.
  - Note that the root's position and rotation will *not* be applied, as that affects the instances absolute position and would put all instances in the same place. However position and rotation from any children or ancestors of the root *will* be applied as they are computed relative to the root's transform.
- If you want to discard all overrides on a particular instance, you can click the **Revert** button.

### Imported Prefabs

When you place a mesh asset into your Assets folder, Unity automatically imports the file and generates something that looks similar to a Prefab out of the mesh. This is not actually a Prefab, it is simply the asset file itself. Instancing and working with assets introduces some limitations that are not present when working with normal Prefabs.




*Notice the asset icon is a bit different from the Prefab icons*

The asset is instantiated in the scene as a GameObject, linked to the source asset instead of a normal Prefab. Components can be added and removed from this GameObject as normal. However, you cannot apply any changes to the asset itself since this would add data to the asset file itself! If you're creating something you want to re-use, you should make the asset instance into a Prefab following the steps listed above under "Creating Prefabs".

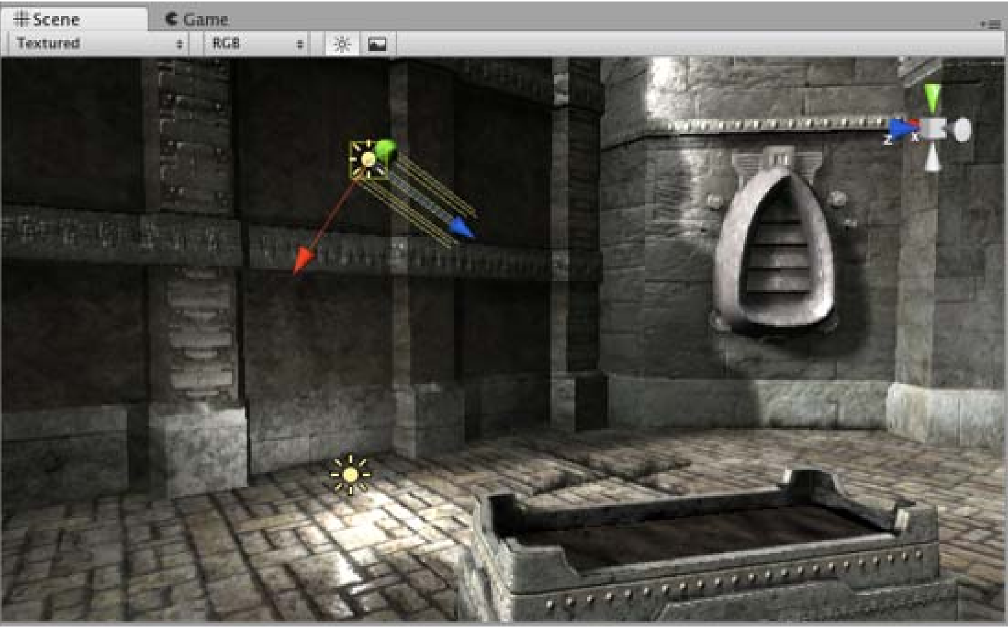
- When you have selected an instance of an asset, the **Apply** button in the Inspector is replaced with an **Edit** button. Clicking this button will launch the editing application for your asset (e.g. Maya or Max).

Page last updated: 2012-09-14

---

# Lights

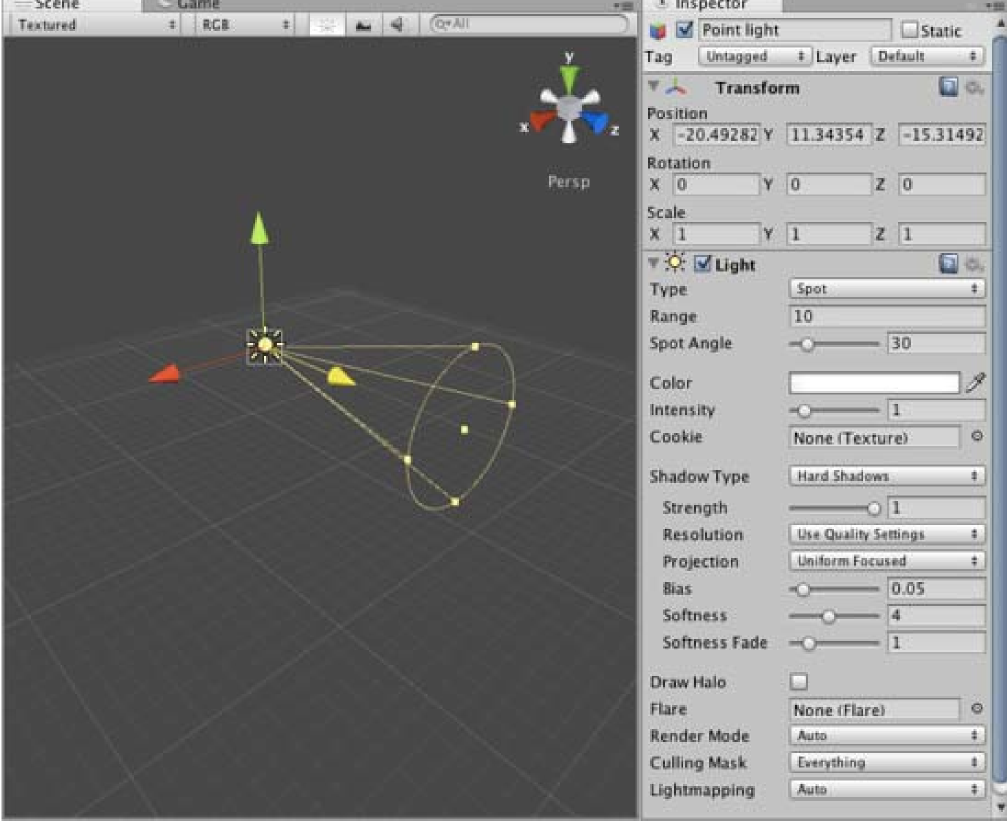
**Lights** are an essential part of every scene. While meshes and textures define the shape and look of a scene, lights define the color and mood of your 3D environment. You'll likely work with more than one light in each scene. Making them work together requires a little practice but the results can be quite amazing.




*A simple, two-light setup*

Lights can be added to your scene from the **GameObject->Create Other** menu. Once a light has been added, you can manipulate it like any other GameObject. Additionally, you can add a Light Component to any selected GameObject by using **Component->Rendering->Light**.

There are many different options within the Light Component in the **Inspector**.

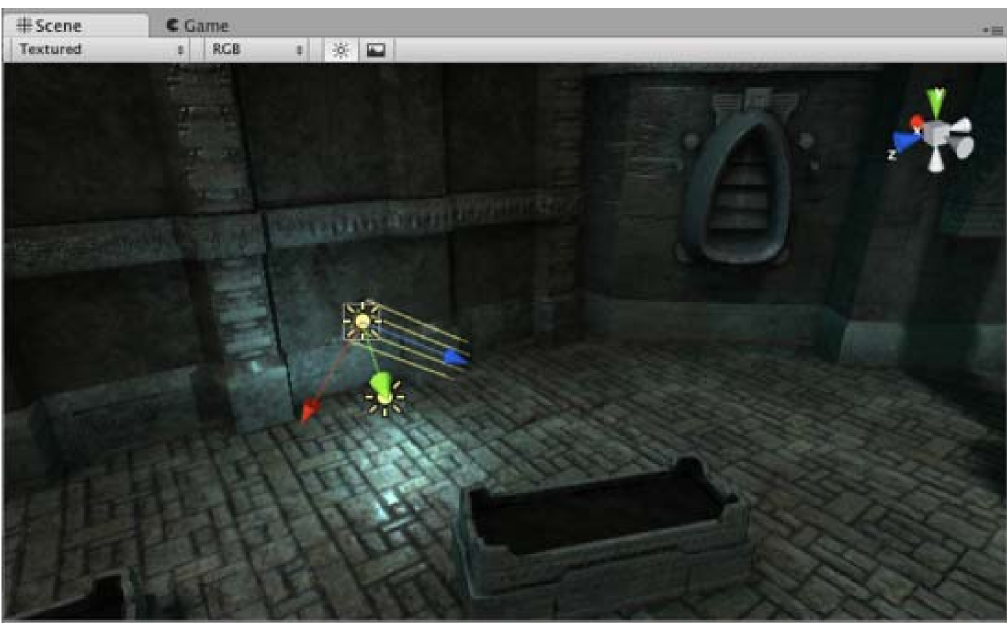


*Light Component properties in the Inspector*

By simply changing the **Color** of a light, you can give a whole different mood to the scene.

*Bright, sunny lights*

*Dark, medieval lights*




*Spooky night lights*

The lights you create this way are **realtime** lights - their lighting is calculated each frame while the game is running. If you know the light will not change, you can make your game faster and look much better by using Lightmapping.

### Rendering paths

Unity supports different Rendering Paths, these paths affect mainly Lights and Shadows, so choosing the correct rendering path depending on your game requirements can improve your project's performance. For more info about rendering paths you can visit the Rendering paths section.

### More information

For more information about using Lights, check the Lights page in the Reference Manual.

Page last updated: 2011-10-24

---

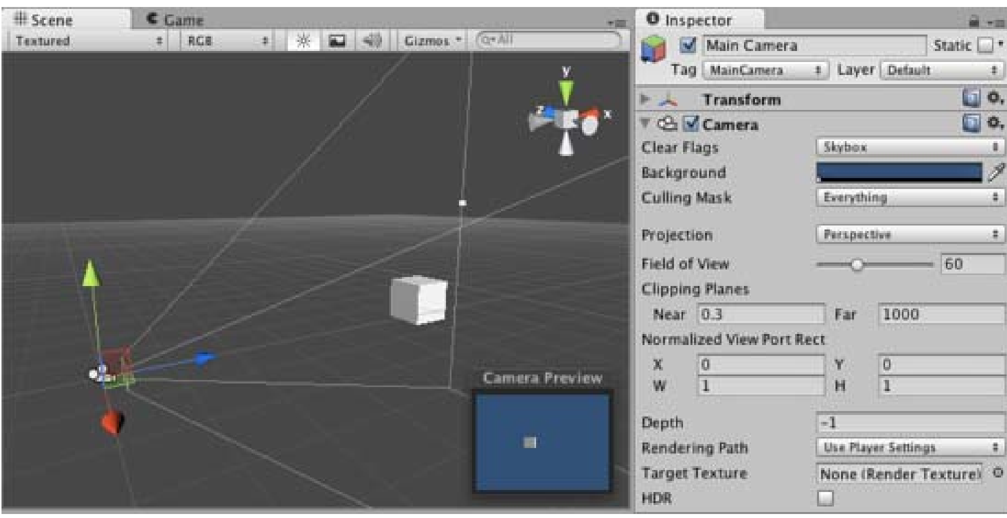
# Cameras

Just as cameras are used in films to display the story to the audience, **Cameras** in Unity are used to display the game world to the player. You will always have at least one camera in a scene, but you can have more than one. Multiple cameras can give you a two-player splitscreen or create advanced custom effects. You can animate cameras, or control them with physics. Practically anything you can imagine is possible with cameras, and you can use typical or unique cameras to fit your game's style.

The remaining text is from the Camera Component reference page.

# Camera

**Cameras** are the devices that capture and display the world to the player. By customizing and manipulating cameras, you can make the presentation of your game truly unique. You can have an unlimited number of cameras in a scene. They can be set to render in any order, at any place on the screen, or only certain parts of the screen.




*Unity's flexible Camera object*

## Properties

| | |
|---|---|
| **Clear Flags** | Determines which parts of the screen will be cleared. This is handy when using multiple Cameras to draw different game elements. |
| **Background** | Color applied to the remaining screen after all elements in view have been drawn and there is no skybox. |
| **Culling Mask** | Include or omit layers of objects to be rendered by the Camera. Assign layers to your objects in the Inspector. |
| **Projection** | Toggles the camera's capability to simulate perspective. |
| **Perspective** | Camera will render objects with perspective intact. |
| **Orthographic** | Camera will render objects uniformly, with no sense of perspective. |
| **Size** (when Orthographic is selected) | The viewport size of the Camera when set to Orthographic. |
| **Field of view** | Width of the Camera's view angle, measured in degrees along the local Y axis. |
| **Clipping Planes** | Distances from the camera to start and stop rendering. |
| **Near** | The closest point relative to the camera that drawing will occur. |
| **Far** | The furthest point relative to the camera that drawing will occur. |
| **Normalized View Port Rect** | Four values that indicate where on the screen this camera view will be drawn, in Screen Coordinates (values 0-1). |
| **X** | The beginning horizontal position that the camera view will be drawn. |
| **Y** | The beginning vertical position that the camera view will be drawn. |
| **W** (Width) | Width of the camera output on the screen. |
| **H** (Height) | Height of the camera output on the screen. |
| **Depth** | The camera's position in the draw order. Cameras with a larger value will be drawn on top of cameras with a smaller value. |
| **Rendering Path** | Options for defining what rendering methods will be used by the camera. |
| **Use Player Settings** | This camera will use whichever Rendering Path is set in the Player Settings. |

| | |
|---|---|
| **Vertex Lit** | All objects rendered by this camera will be rendered as Vertex-Lit objects. |
| **Forward** | All objects will be rendered with one pass per material, as was standard in Unity 2.x. |
| **Deferred Lighting** (Unity Pro only) | All objects will be drawn once without lighting, then lighting of all objects will be rendered together at the end of the render queue. |
| **Target Texture** (Unity Pro/Advanced only) | Reference to a Render Texture that will contain the output of the Camera view. Making this reference will disable this Camera's capability to render to the screen. |
| **HDR** | Enables High Dynamic Range rendering for this camera. |

## Details

Cameras are essential for displaying your game to the player. They can be customized, scripted, or parented to achieve just about any kind of effect imaginable. For a puzzle game, you might keep the Camera static for a full view of the puzzle. For a first-person shooter, you would parent the Camera to the player character, and place it at the character's eye level. For a racing game, you'd likely want to have the Camera follow your player's vehicle.

You can create multiple Cameras and assign each one to a different **Depth**. Cameras are drawn from low **Depth** to high **Depth**. In other words, a Camera with a **Depth** of 2 will be drawn on top of a Camera with a depth of 1. You can adjust the values of the **Normalized View Port Rectangle** property to resize and position the Camera's view onscreen. This can create multiple mini-views like missile cams, map views, rear-view mirrors, etc.

### Render Path

Unity supports different Rendering Paths. You should choose which one you use depending on your game content and target platform / hardware. Different rendering paths have different features and performance characteristics that mostly affect Lights and Shadows.
The rendering Path used by your project is chosen in Player Settings. Additionally, you can override it for each Camera.

For more info on rendering paths, check the rendering paths page.

### Clear Flags

Each Camera stores color and depth information when it renders its view. The portions of the screen that are not drawn in are empty, and will display the skybox by default. When you are using multiple Cameras, each one stores its own color and depth information in buffers, accumulating more data as each Camera renders. As any particular Camera in your scene renders its view, you can set the **Clear Flags** to clear different collections of the buffer information. This is done by choosing one of the four options:

### Skybox

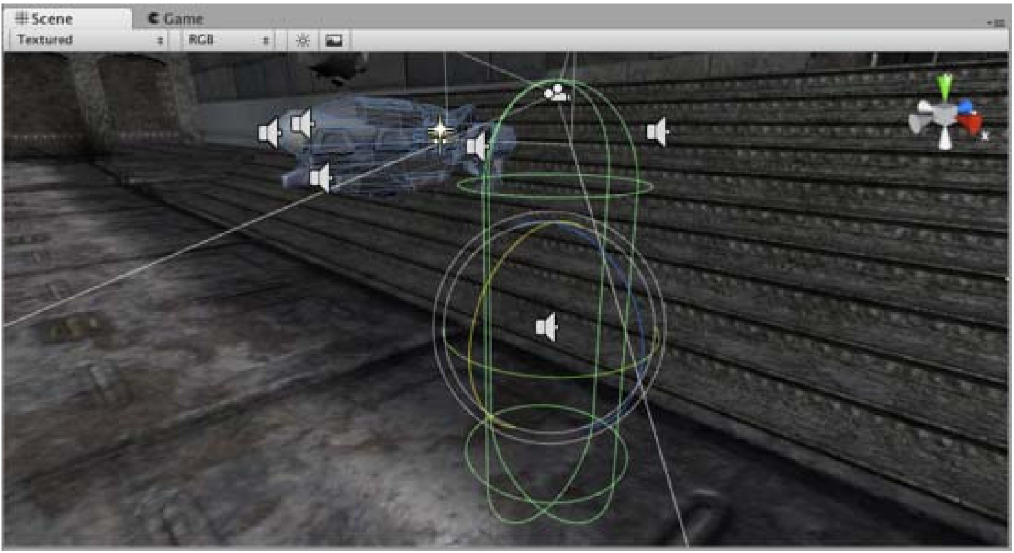
This is the default setting. Any empty portions of the screen will display the current Camera's skybox. If the current Camera has no skybox set, it will default to the skybox chosen in the Render Settings (found in **Edit->Render Settings**). It will then fall back to the **Background Color**. Alternatively a Skybox component can be added to the camera. If you want to create a new Skybox, you can use this guide.

### Solid Color

Any empty portions of the screen will display the current Camera's **Background Color**.

### Depth Only

If you wanted to draw a player's gun without letting it get clipped inside the environment, you would set one Camera at **Depth** 0 to draw the environment, and another Camera at **Depth** 1 to draw the weapon alone. The weapon Camera's **Clear Flags** should be set to to **depth only**. This will keep the graphical display of the environment on the screen, but discard all information about where each object exists in 3-D space. When the gun is drawn, the opaque parts will completely cover anything drawn, regardless of how close the gun is to the wall.



*The gun is drawn last, after clearing the depth buffer of the cameras before it*

**Don't Clear**
This mode does not clear either the color or the depth buffer. The result is that each frame is drawn over the next, resulting in a smear-looking effect. This isn't typically used in games, and would likely be best used with a custom shader.

**Clip Planes**
The **Near** and **Far Clip Plane** properties determine where the Camera's view begins and ends. The planes are laid out perpendicular to the Camera's direction and are measured from the its position. The **Near plane** is the closest location that will be rendered, and the **Far plane** is the furthest.

The clipping planes also determine how depth buffer precision is distributed over the scene. In general, to get better precision you should move the **Near plane** as far as possible.

Note that the near and far clip planes together with the planes defined by the field of view of the camera describe what is popularly known as the camera *frustum*. Unity ensures that when rendering your objects those which are completely outside of this frustum are not displayed. This is called Frustum Culling. Frustum Culling happens irrespective of whether you use Occlusion Culling in your game.

For performance reasons, you might want to cull small objects earlier. For example, small rocks and debris could be made invisible at much smaller distance than large buildings. To do that, put small objects into a separate layer and setup per-layer cull distances using Camera.layerCullDistances script function.

**Culling Mask**
The **Culling Mask** is used for selectively rendering groups of objects using Layers. More information on using layers can be found here.

Commonly, it is good practice to put your User Interface on a different layer, then render it by itself with a separate Camera set to render the UI layer by itself.

In order for the UI to display on top of the other Camera views, you'll also need to set the **Clear Flags** to **Depth only** and make sure that the UI Camera's **Depth** is higher than the other Cameras.

**Normalized Viewport Rectangle**
**Normalized Viewport Rectangles** are specifically for defining a certain portion of the screen that the current camera view will be drawn upon. You can put a map view in the lower-right hand corner of the screen, or a missile-tip view in the upper-left corner. With a bit of design work, you can use **Viewport Rectangle** to create some unique behaviors.

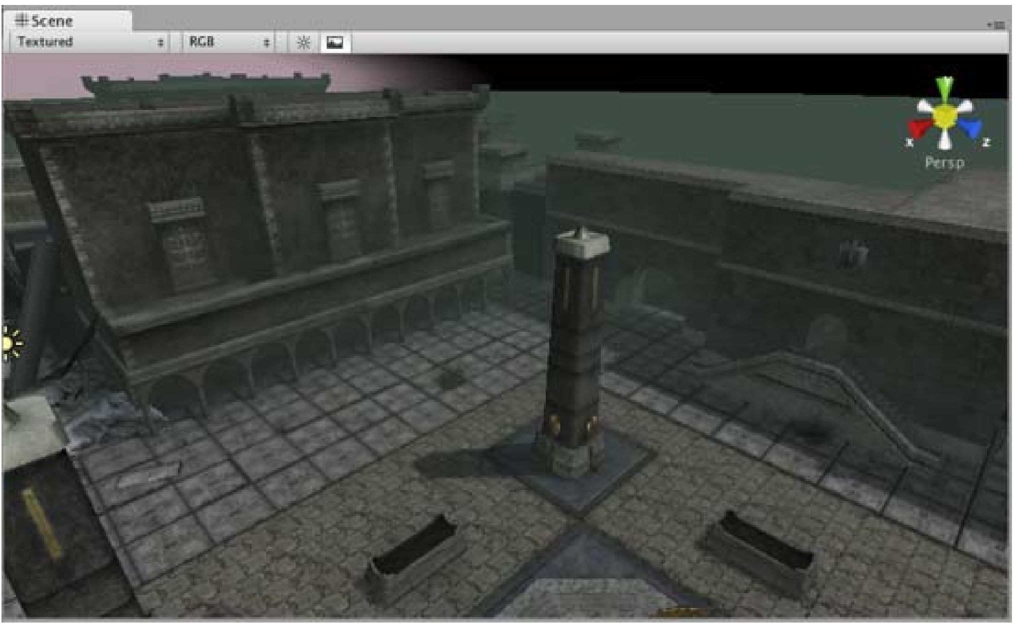
It's easy to create a two-player split screen effect using **Normalized Viewport Rectangle**. After you have created your two cameras, change both camera H value to be 0.5 then set player one's Y value to 0.5, and player two's Y value to 0. This will make player one's camera display from halfway up the screen to the top, and player two's camera will start at the bottom and stop halfway up the screen.

*Two-player display created with **Normalized Viewport Rectangle***

**Orthographic**

Marking a Camera as **Orthographic** removes all perspective from the Camera's view. This is mostly useful for making isometric or 2D games.

Note that fog is rendered uniformly in orthographic camera mode and may therefore not appear as expected. Read more about why in the component reference on Render Settings.




*Perspective camera.*

*Orthographic camera. Objects do not get smaller with distance here!*

**Render Texture**

This feature is only available for Unity Advanced licenses . It will place the camera's view onto a Texture that can then be applied to another object. This makes it easy to create sports arena video monitors, surveillance cameras, reflections etc.

*A Render Texture used to create a live arena-cam*

## Hints

- Cameras can be instantiated, parented, and scripted just like any other GameObject.
- To increase the sense of speed in a racing game, use a high **Field of View**.
- Cameras can be used in physics simulation if you add a **Rigidbody** Component.
- There is no limit to the number of Cameras you can have in your scenes.
- Orthographic cameras are great for making 3D user interfaces
- If you are experiencing depth artifacts (surfaces close to each other flickering), try setting **Near Plane** to as large as possible.
- Cameras cannot render to the Game Screen and a Render Texture at the same time, only one or the other.
- Pro license holders have the option of rendering a Camera's view to a texture, called Render-to-Texture, for even more unique effects.
- Unity comes with pre-installed Camera scripts, found in **Components->Camera Control**. Experiment with them to get a taste of what's possible.

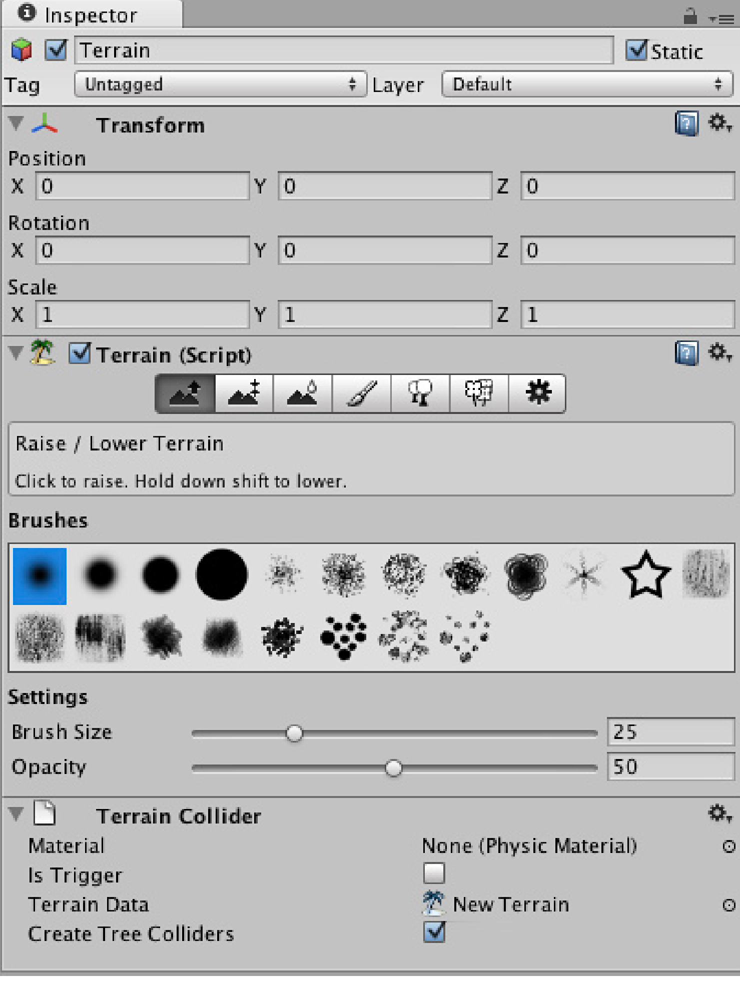
Page last updated: 2007-11-16

## Terrains

This section will explain how to use the **Terrain Engine**. It will cover creation, technical details, and other considerations. It is broken into the following sections:

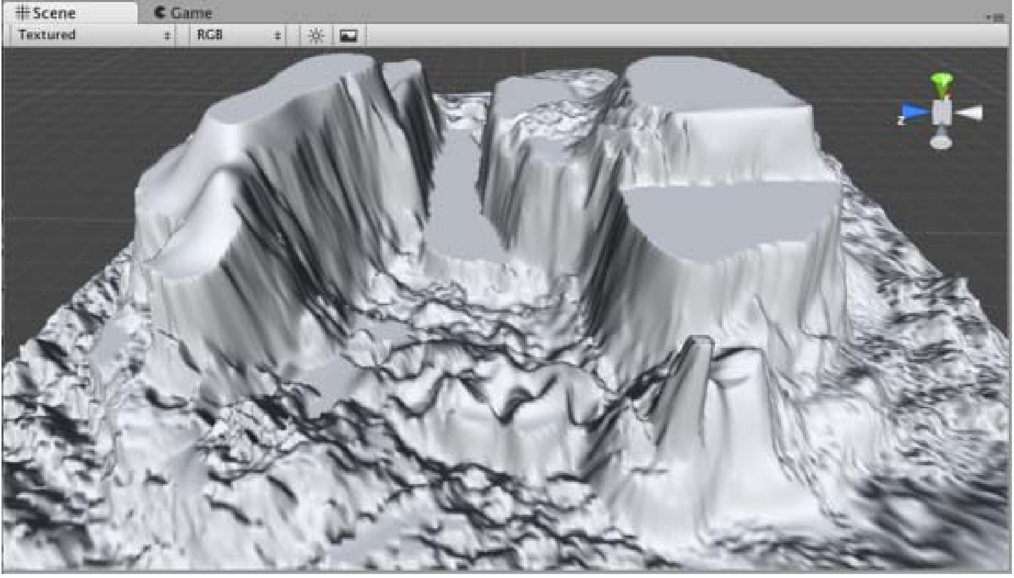
### Using Terrains

This section covers the most basic information about using Terrains. This includes creating Terrains and how to use the new Terrain tools & brushes.




### Height

This section explains how to use the different tools and brushes that alter the Height of the Terrain.

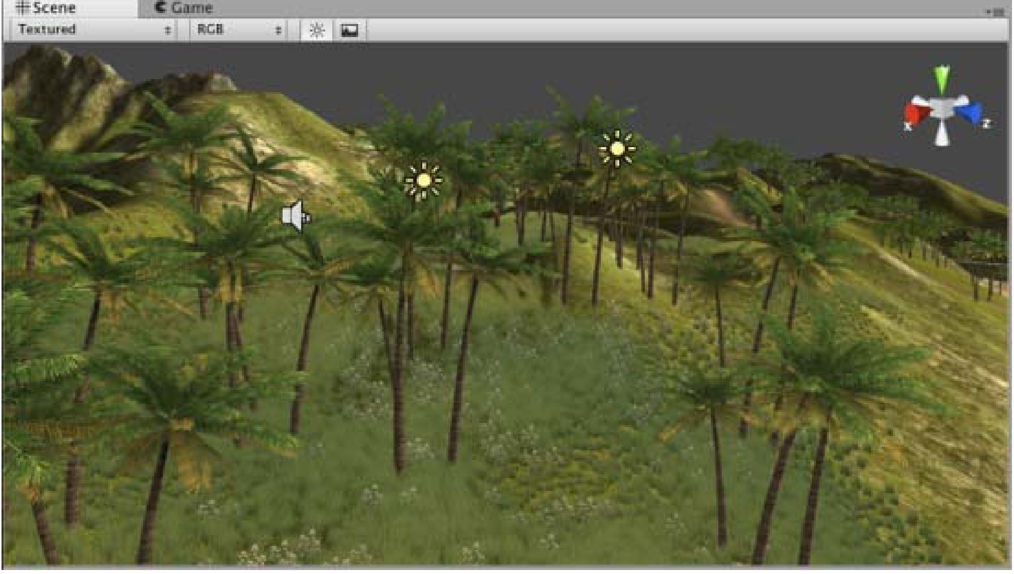



## Terrain Textures

This section explains how to add, paint and blend Terrain Textures using different brushes.

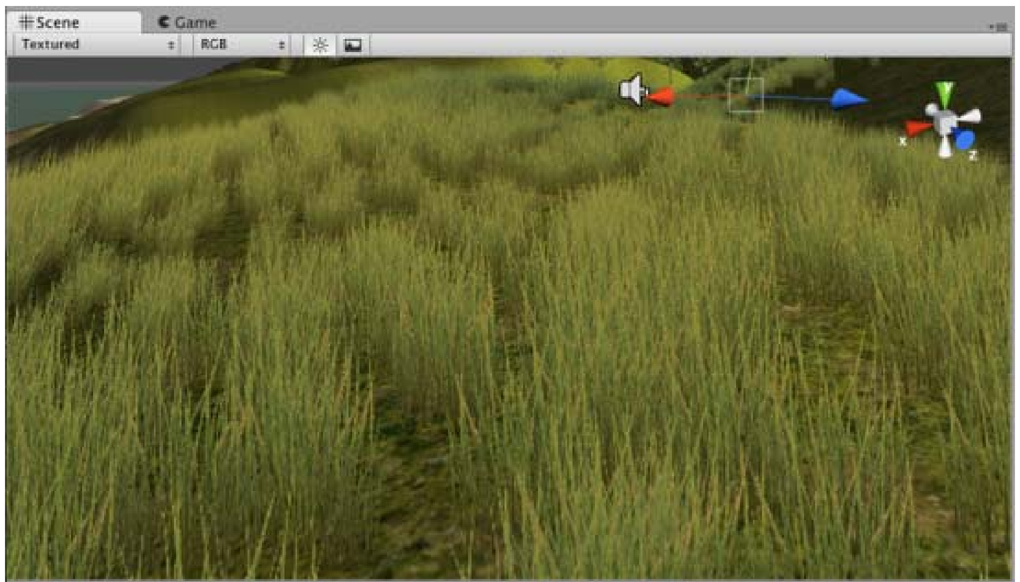
## Trees

This section contains important information for creating your own tree assets. It also covers adding and painting trees on your Terrain.




## Grass

This section explains how grass works, and how to use it.




## Detail Meshes

This section explains practical usage for detail meshes like rocks, haystacks, vegetation, etc.

### Lightmaps

You can lightmap terrains just like any other objects using Unity's built-in lightmapper. See Lightmapping Quickstart for help.

### Other Settings

This section covers all the other settings associated with Terrains.

#### Mobile performance note

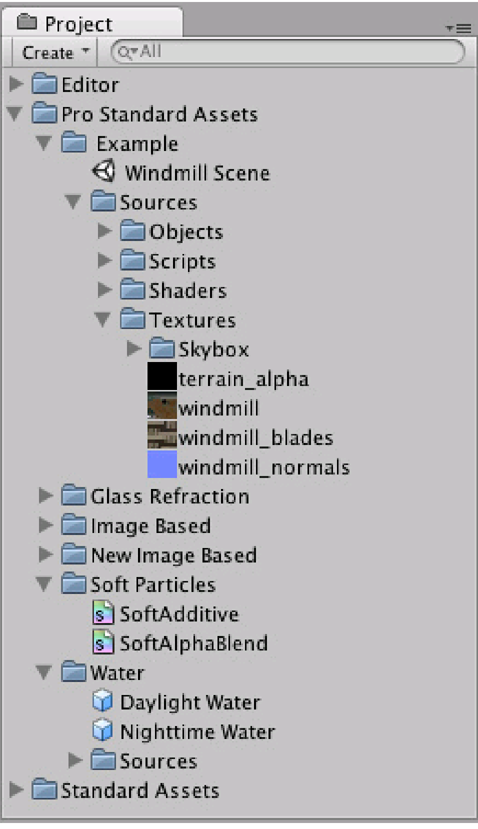
Rendering terrain is quite expensive, so terrain engine is not very practical on lower-end mobile devices.

Page last updated: 2010-06-03

---

## Asset Import and Creation

A large part of making a game is utilizing your asset source files in your **GameObjects**. This goes for textures, models, sound effects and behaviour scripts. Using the **Project View** inside Unity, you have quick access to all the files that make up your game:



*The Project View displays all source files and created **Prefabs***

This view shows the organization of files in your project's Assets folder. Whenever you update one of your asset files, the changes are immediately reflected in your game!

To import an asset file into your project, move the file into **(your Project folder)->Assets** in the Finder, and it will automatically be imported into Unity. To apply your assets, simply drag the asset file from the Project View window into the **Hierarchy** or **Scene View**. If the asset is meant to be applied to another object, drag the asset over the object.

## Hints

- It is always a good idea to add labels to your assets when you are working with big projects or when you want to keep organized all your assets, with this you can search for the labels associated to each asset in the *search field* in the project view.
- When backing up a project folder **always** back up *Assets*, *ProjectSettings* and *Library* folders. The Library folder contains all meta data and all the connections between objects, thus if the Library folder gets lost, you will lose references from scenes to assets. Easiest is just to back up the whole project folder containing the Assets, ProjectSettings and Library folders.
- Rename and move files to your heart's content inside Project View; nothing will break.
- **Never** rename or move anything from the Finder or another program; everything will break. In short, Unity stores lots of metadata for each asset (things like import settings, cached versions of compressed textures, etc.) and if you move a file externally, Unity can no longer associate metadata with the moved file.

Continue reading for more information:

- Importing Assets
- Meshes
  - 3D formats
- Legacy animation system
- Materials and Shaders
- Texture 2D
- Procedural Materials

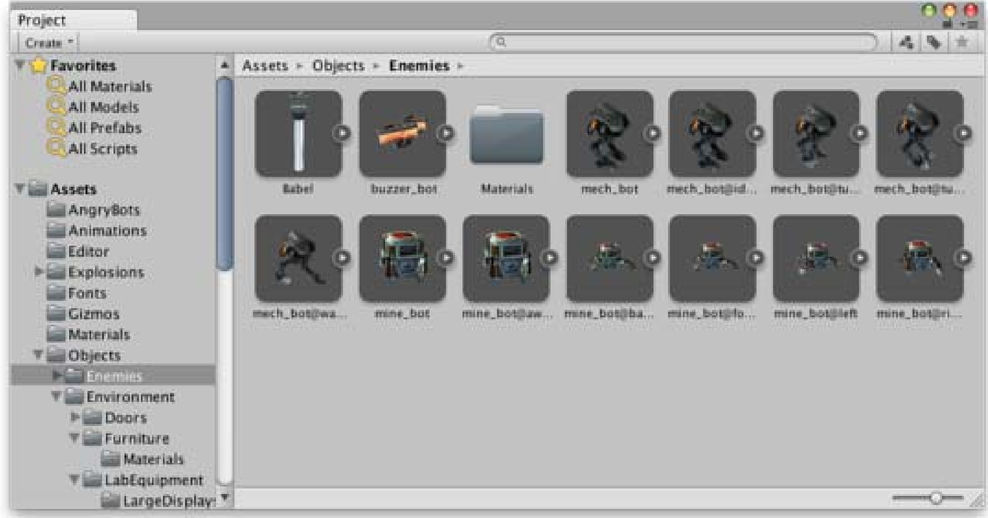
- Movie Texture
- Audio Files
  - Tracker Modules
- Using Scripts
- Asset Store
- Asset Server (Pro Only)
  - Cache Server (Team license only)
  - Cache Server FAQ
- Behind the Scenes


Page last updated: 2012-01-08

---

## Importing Assets

Unity will automatically detect files as they are added to your Project folder's **Assets** folder. When you put any asset into your Assets folder, you will see the asset appear in your **Project View**.




*The Project View is your window into the Assets folder, normally accessible from the file manager*

When you are organizing your Project View, there is one very important thing to remember:

**Never move any assets or organize this folder from the Explorer (Windows) or Finder (OS X). Always use the Project View!**

There is a lot of meta data stored about relationships between asset files within Unity. This data is all dependent on where Unity expects to find these assets. If you move an asset from within the Project View, these relationships are maintained. If you move them outside of Unity, these relationships are broken. You'll then have to manually re-link lots of dependencies, which is something you probably don't want to do.

So just remember to only save assets to the Assets folder from other applications, and never rename or move files outside of Unity. Always use Project View. You can safely open files for editing from anywhere, of course.

### Creating and Updating Assets

When you are building a game and you want to add a new asset of any type, all you have to do is create the asset and save it somewhere in the Assets folder. When you return to Unity or launch it, the added file(s) will be detected and imported.

Additionally, as you update and save your assets, the changes will be detected and the asset will be re-imported in Unity. This allows you to focus on refining your assets without struggling to make them compatible with Unity. Updating and saving your assets normally from its native application provides optimum, hassle-free workflow that feels natural.

### Asset Types

There are a handful of basic asset types that will go into your game. The types are:

- Mesh Files & Animations
- Texture Files
- Sound Files

We'll discuss the details of importing each of these file types and how they are used.

**Meshes & Animations**
Whichever 3D package you are using, Unity will import the meshes and animations from each file. For a list of applications that are supported by Unity, please see this page.

Your mesh file does not need to have an animation to be imported. If you do use animations, you have your choice of importing all animations from a single file, or importing separate files, each with one animation. For more information about importing animations, please see page about Animation Import .

Once your mesh is imported into Unity, you can drag it to the **Scene** or **Hierarchy** to create an instance of it. You can also add **Components** to the instance, which will not be attached to mesh file itself.

Meshes will be imported with UVs and a number of default **Materials** (one material per UV). You can then assign the appropriate texture files to the materials and complete the look of your mesh in Unity's game engine.

**Textures**
Unity supports all image formats. Even when working with layered Photoshop files, they are imported without disturbing the Photoshop format. This allows you to work with a single texture file for a very care-free and streamlined experience.

You should make your textures in dimensions that are to the power of two (e.g. 32x32, 64x64, 128x128, 256x256, etc.) Simply placing them in your project's Assets folder is sufficient, and they will appear in the Project View.

Once your texture has been imported, you should assign it to a Material. The material can then be applied to a mesh, **Particle System**, or **GUI Texture**. Using the **Import Settings**, it can also be converted to a **Cubemap** or **Normalmap** for different types of applications in the game. For more information about importing textures, please read the Texture Component page.

**Sounds**

## Desktop

Unity features support for two types of audio: **Uncompressed Audio** or **Ogg Vorbis**. Any type of audio file you import into your project will be converted to one of these formats.

**File Type Conversion**

| | |
|---|---|
| **.AIFF** | Converted to uncompressed audio on import, best for short sound effects. |
| **.WAV** | Converted to uncompressed audio on import, best for short sound effects. |
| **.MP3** | Converted to Ogg Vorbis on import, best for longer music tracks. |
| **.OGG** | Compressed audio format, best for longer music tracks. |

**Import Settings**
If you are importing a file that is not already compressed as Ogg Vorbis, you have a number of options in the **Import Settings** of the Audio Clip. Select the Audio Clip in the **Project View** and edit the options in the **Audio Importer** section of the **Inspector**. Here, you can compress the Clip into Ogg Vorbis format, force it into Mono or Stereo playback, and tweak other options. There are positives and negatives for both Ogg Vorbis and uncompressed audio. Each has its own ideal usage scenarios, and you generally should not use either one exclusively.

Read more about using Ogg Vorbis or Uncompressed audio on the Audio Clip Component Reference page.

## iOS

Unity features support for two types of audio: **Uncompressed Audio** or **MP3 Compressed audio**. Any type of audio file you import into your project will be converted to one of these formats.

**File Type Conversion**

| | |
|---|---|
| **.AIFF** | Imports as uncompressed audio for short sound effects. Can be compressed in Editor on demand. |
| **.WAV** | Imports as uncompressed audio for short sound effects. Can be compressed in Editor on demand. |

**.MP3** Imports as Apple Native compressed format for longer music tracks. Can be played on device hardware.
**.OGG** OGG compressed audio format, **incompatible** with the iPhone device. Please use MP3 compressed sounds on the iPhone.

**Import Settings**
When you are importing an audio file, you can select its final format and choose to force it to stereo or mono channels. To access the **Import Settings**, select the Audio Clip in the **Project View** and find the **Audio Importer** in the Inspector. Here, you can compress the Clip into Ogg Vorbis format, force it into Mono or Stereo playback, and tweak other options, such as the very important Decompress On Load setting.

Read more about using MP3 Compressed or Uncompressed audio on the Audio Clip Component Reference page.

## Android

Unity features support for two types of audio: **Uncompressed Audio** or **MP3 Compressed audio**. Any type of audio file you import into your project will be converted to one of these formats.

**File Type Conversion**
**.AIFF** Imports as uncompressed audio for short sound effects. Can be compressed in Editor on demand.
**.WAV** Imports as uncompressed audio for short sound effects. Can be compressed in Editor on demand.
**.MP3** Imports as MP3 compressed format for longer music tracks.
**.OGG** Note: the OGG compressed audio format is **incompatible** with some Android devices, so Unity does not support it for the Android platform. Please use MP3 compressed sounds instead.

**Import Settings**
When you are importing an audio file, you can select its final format and choose to force it to stereo or mono channels. To access the **Import Settings**, select the Audio Clip in the **Project View** and find the **Audio Importer** in the Inspector. Here, you can compress the Clip into Ogg Vorbis format, force it into Mono or Stereo playback, and tweak other options, such as the very important Decompress On Load setting.

Read more about using MP3 Compressed or Uncompressed audio on the Audio Clip Component Reference page.

Once sound files are imported, they can be attached to any **GameObject**. The Audio file will create an Audio Source Component automatically when you drag it onto a GameObject.

Page last updated: 2012-10-26

---

# Meshes

When a 3D model is imported, Unity represents it internally as a **Mesh**. A Mesh must be attached to a GameObject using a Mesh Filter component. For the mesh to be visible, the GameObject must also have a Mesh Renderer or other suitable rendering component attached. With these components in place, the mesh will be visible at the GameObject's position with its exact appearance dependent on the Material used by the renderer.

*A Mesh Filter together with Mesh Renderer makes the model appear on screen.*

Unity's mesh importer provides many options for controlling the generation of the mesh and associating it with its textures and materials. These options are covered by the following pages:-

- 3D formats

Page last updated: 2012-01-19

---

# 3D-formats

Importing meshes into Unity can be achieved from two main types of files:

1. **Exported 3D file formats**, such as .FBX or .OBJ
2. **Proprietary 3D application files**, such as .Max and .Blend file formats from 3D Studio Max or Blender for example.

Either should enable you to get your meshes into Unity, but there are considerations as to which type you choose:

### Exported 3D files

Unity can read .FBX, **.dae** (Collada), **.3DS**, **.dxf** and **.obj** files, FBX exporters can be found here and obj or Collada exporters can also be found for many applications

**Advantages:**

- Only export the data you need
- Verifiable data (re-import into 3D package before Unity)
- Generally smaller files
- Encourages modular approach - e.g different components for collision types or interactivity
- Supports other 3D packages whose Proprietary formats we don't have direct support for

**Disadvantages:**

- Can be a slower pipeline for prototyping and iterations
- Easier to lose track of versions between source(working file) and game data (exported FBX for example)

### Proprietary 3D application files

Unity can also import, *through conversion*: **Max**, **Maya**, **Blender**, **Cinema4D**, **Modo**, **Lightwave** & **Cheetah3D** files, e.g. **.MAX**, **.MB**, **.MA** etc.

**Advantages:**

- Quick iteration process (save the source file and Unity reimports)
- Simple initially

**Disadvantages:**

- A licensed copy of that software must be installed on all machines using the Unity project
- Files can become bloated with unnecessary data
- Big files can slow Unity updates
- Less validation � harder to troubleshoot problems

Page last updated: 2012-10-24

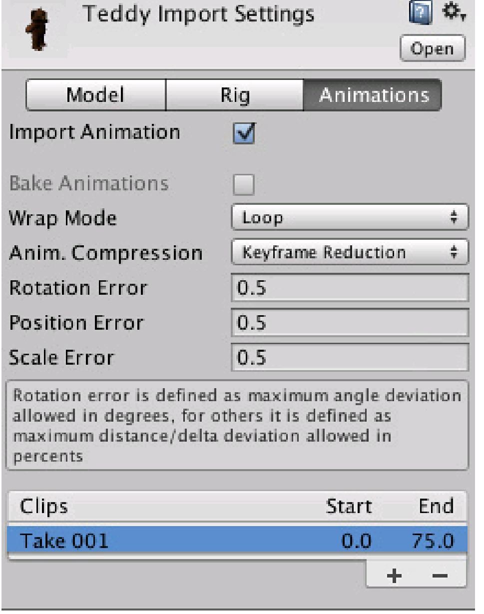
---

# Animations (Legacy)

Prior to the introduction of Mecanim, Unity used its own animation system and for backward compatiblity, this system is still available. The main reason for using legacy animation is to continue working with an old project without the work of updating it for Mecanim. However, it is not recommended that you use the legacy system for new projects.

### Working with legacy animations

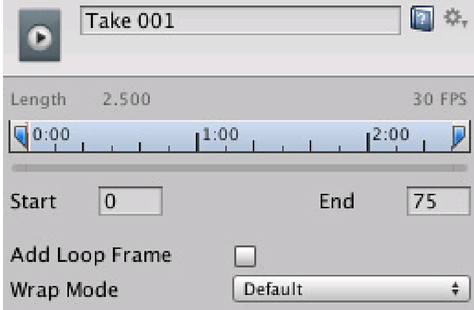
To import a legacy animation, you first need to mark it as such in the Mesh importer's Rig tab:-



The Animation tab on the importer will then look something like this:-

| | |
|---|---|
| **Import Animation** | Selects whether or not animation should be imported at all. |
| **Wrap Mode** | The method of handling what happens when the animation comes to an end:- |
| **Default** | Uses whatever setting is specified in the animation clip. |
| **Once** | Play the clip to the end and then finish. |
| **Loop** | Play to the end, then immediately restart from the beginning. |
| **PingPong** | Play to the end, then play from the end in reverse, and so on. |
| **Forever** | Play to the end, then loop the last frame indefinitely. |
| **Anim Compression** | Settings to attempt to remove redundant information from clips:- |
| **Off** | No compression. |
| **Keyframe reduction** | Attempt to remove keyframes where differences are too small to be seen |
| **Keyframe reduction and compression** | As for *Keyframe reduction*, but clip data is also compressed. |
| **Rotation error** | Minimum difference in rotation values (in degrees), below which two keyframes are counted as equal. |
| **Position error** | Minimum difference in position (as a percentage of coordinate values), below which two keyframes are counted as equal. |
| **Rotation error** | Minimum difference in scale (as a percentage of coordinate values), below which two keyframes are counted as equal. |

Below the properties in the inspector is a list of animation clips. When you click on a clip in the list, an additional panel will appear below it in the inspector:-



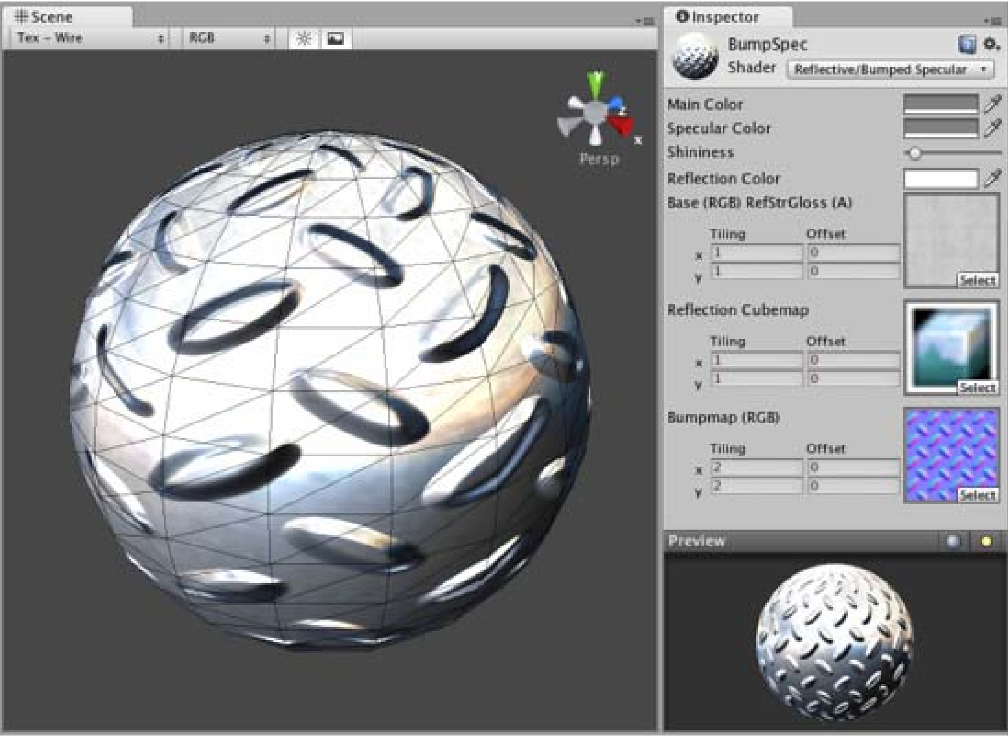

The Start and End values can be changed to allow you to use just a part of the original clip (see the page on |splitting animations for further details). The *Add Loop Frame* option adds an extra keyframe to the end of the animation that is exactly the same as the keyframe at the start. This enables the animation to loop smoothly even when the last frame doesn't exactly match up with the first. The *Wrap Mode* setting is identical to the master setting in the main animation properties but applies only to that specific clip.

Page last updated: 2012-11-09

---

# Materials

There is a close relationship between **Materials** and **Shaders** in Unity. Shaders contain code that defines what kind of properties and assets to use. Materials allow you to adjust properties and assign assets.




*A Shader is implemented through a Material*

To create a new Material, use **Assets->Create->Material** from the main menu or the **Project View** context menu. Once the Material has been created, you can apply it to an object and tweak all of its properties in the **Inspector**. To apply it to an object, just drag it from the **Project View** to any object in the **Scene** or **Hierarchy**.

### Setting Material Properties

You can select which Shader you want any particular Material to use. Simply expand the **Shader** drop-down in the Inspector, and choose your new Shader. The Shader you choose will dictate the available properties to change. The properties can be colors, sliders, textures, numbers, or vectors. If you have applied the Material to an active object in the **Scene**, you will see your property changes applied to the object in real-time.

There are two ways to apply a **Texture** to a property.

1. Drag it from the Project View on top of the Texture square
2. Click the **Select** button, and choose the texture from the drop-down list that appears

Two placement options are available for each **Texture**:

| | |
|---|---|
| **Tiling** | Scales the texture along the different. |
| **Offset** | Slides the texture around. |

### Built-in Shaders

There is a library of built-in Shaders that come standard with every installation of Unity. There are over 30 of these built-in Shaders, and six basic families.

- Normal: For opaque textured objects.
- Transparent: For partly transparent objects. The texture's alpha channel defines the level of transparency.
- TransparentCutOut: For objects that have only fully opaque and fully transparent areas, like fences.
- Self-Illuminated: For objects that have light emitting parts.
- Reflective: For opaque textured objects that reflect an environment **Cubemap**.

In each group, built-in shaders range by complexity, from the simple **VertexLit** to the complex **Parallax Bumped with Specular**. For more information about performance of Shaders, please read the built-in Shader performance page

This grid displays a thumbnail of all built-in Shaders:

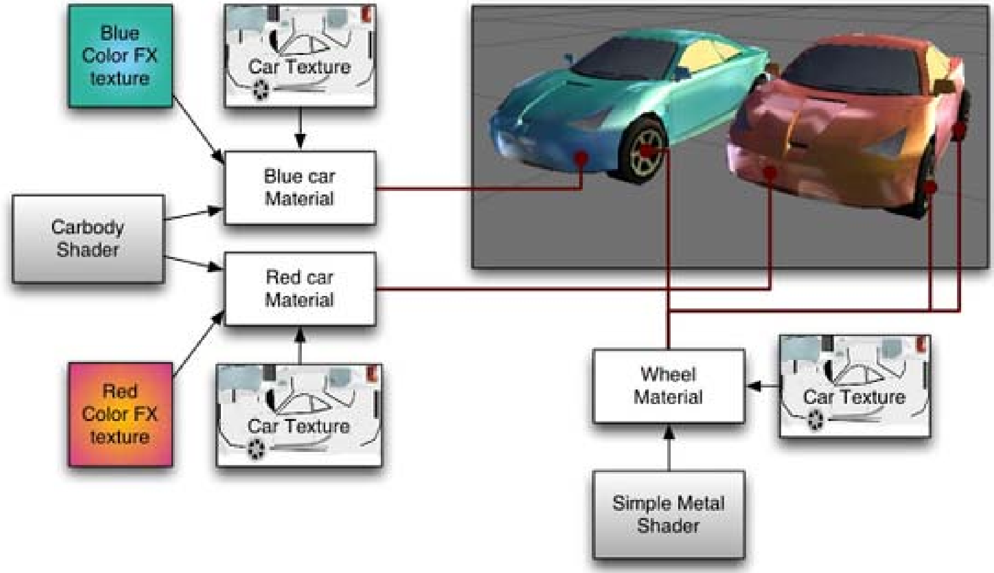
*The builtin Unity shaders matrix*

## Shader technical details

Unity has an extensive Shader system, allowing you to tweak the look of all in-game graphics. It works like this:

A Shader basically defines a formula for how the in-game shading should look. Within any given Shader is a number of properties (typically textures). Shaders are implemented through **Materials**, which are attached directly to individual **GameObjects**. Within a Material, you will choose a Shader, then define the properties (usually textures and colors, but properties can vary) that are used by the Shader.

This is rather complex, so let's look at a workflow diagram:




On the left side of the graph is the **Carbody Shader**. 2 different Materials are created from this: **Blue car Material** and **Red car Material**. Each of these Materials have 2 textures assigned; the **Car Texture** defines the main texture of the car, and a **Color FX texture**. These properties are used by the shader to make the car finish look like 2-tone paint. This can be seen on the front of the red car: it is yellow where it faces the camera and then fades towards purple as the angle increases. The car materials are attached to the 2 cars. The car wheels, lights and windows don't have the color change effect, and must hence use a different Material. At the bottom of the graph there is a **Simple Metal Shader**. The **Wheel Material** is using this Shader. Note that even though the same **Car Texture** is reused here, the end result is quite different from the car body, as the Shader used in the Material is different.

To be more specific, a Shader defines:

- The method to render an object. This includes using different methods depending on the graphics card of the end user.
- Any vertex and fragment programs used to render.
- Some texture properties that are assignable within Materials.
- Color and number settings that are assignable within Materials.

A Material defines:

- Which textures to use for rendering.
- Which colors to use for rendering.
- Any other assets, such as a Cubemap that is required by the shader for rendering.

Shaders are meant to be written by graphics programmers. They are created using the **ShaderLab** language, which is quite simple. However, getting a shader to work well on a variety graphics cards is an involved job and requires a fairly comprehensive knowledge of how graphics cards work.

A number of shaders are built into Unity directly, and some more come in the Standard Assets Library. If you like, there is plenty more shader information in the Built-in Shader Guide.

Page last updated: 2010-09-16

---

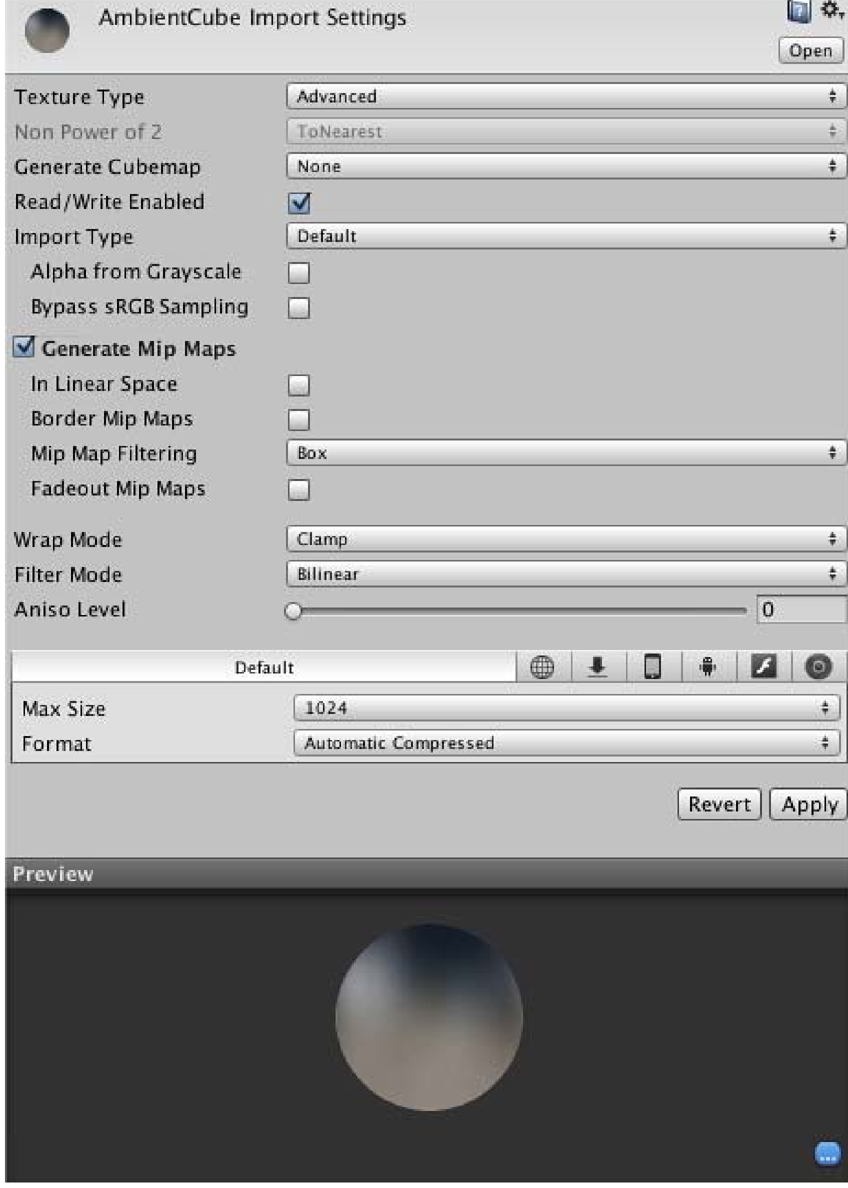
# Textures

**Textures** bring your **Meshes**, **Particles**, and interfaces to life! They are image or movie files that you lay over or wrap around your objects. As they are so important, they have a lot of properties. If you are reading this for the first time, jump down to Details, and return to the actual settings when you need a reference.

The shaders you use for your objects put specific requirements on which textures you need, but the basic principle is that you can put any image file inside your project. If it meets the size requirements (specified below), it will get imported and optimized for game use. This extends to multi-layer Photoshop or TIFF files - they are flattened on import, so there is no size penalty for your game.

## Properties

The **Texture Inspector** looks a bit different from most others:



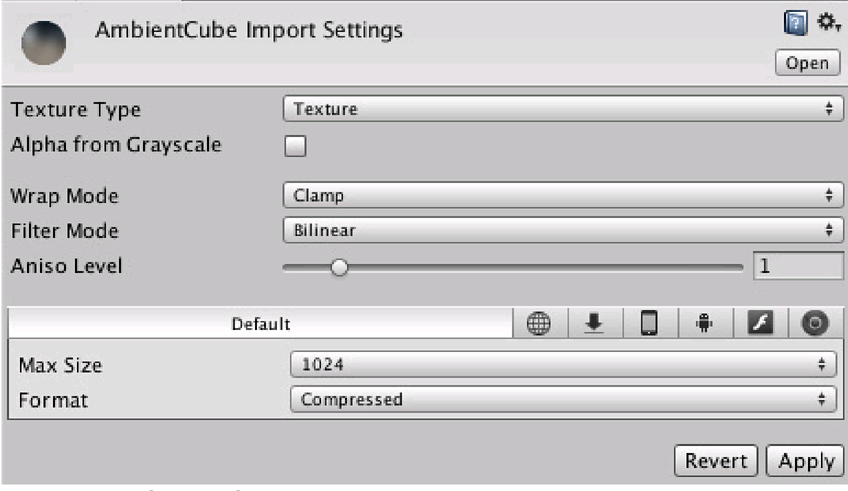

The inspector is split into two sections, the **Texture Importer** and the texture preview.

## Texture Importer

Textures all come from image files in your **Project Folder**. How they are imported is specified by the **Texture Importer**. You change these by selecting the file texture in the **Project View** and modifying the **Texture Importer** in the **Inspector**.

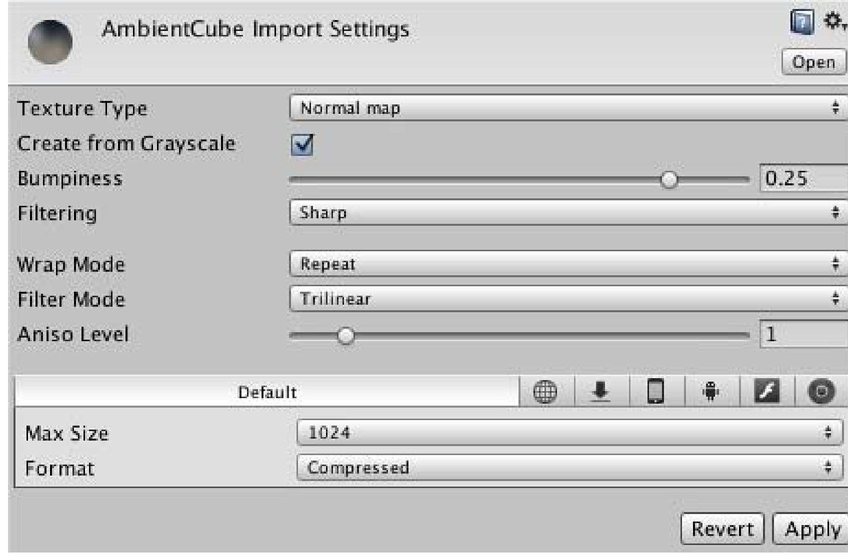
The topmost item in the inspector is the **Texture Type** menu that allows you to select the type of texture you want to create from the source image file.

| | |
|---|---|
| **Texture Type** | Select this to set basic parameters depending on the purpose of your texture. |
| **Texture** | This is the most common setting used for all the textures in general. |
| **Normal Map** | Select this to turn the color channels into a format suitable for real-time normal mapping. For more info, see Normal Maps |
| **GUI** | Use this if your texture is going to be used on any HUD/GUI Controls. |
| **Reflection** | Also known as Cube Maps, used to create reflections on textures. check Cubemap Textures for more info. |
| **Cookie** | This sets up your texture with the basic parameters used for the Cookies of your lights |
| **Advanced** | Select this when you want to have specific parameters on your texture and you want to have total control over your texture. |




*Basic Texture Settings Selected*

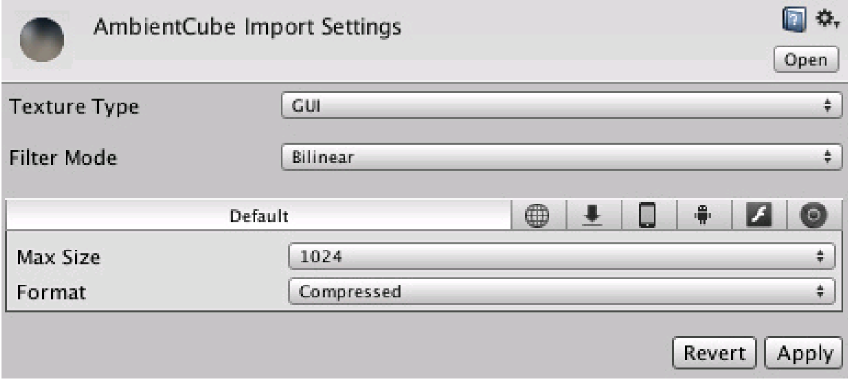
| | |
|---|---|
| **Alpha From Grayscale** | If enabled, an alpha transparency channel will be generated by the image's existing values of light & dark. |
| **Wrap Mode** | Selects how the Texture behaves when tiled: |
| **Repeat** | The Texture repeats (tiles) itself |
| **Clamp** | The Texture's edges get stretched |
| **Filter Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations: |
| **Point** | The Texture becomes blocky up close |
| **Bilinear** | The Texture becomes blurry up close |
| **Trilinear** | Like Bilinear, but the Texture also blurs between the different mip levels |
| **Aniso Level** | Increases texture quality when viewing the texture at a steep angle. Good for floor and ground textures, see below. |




*Normal Map Settings in the Texture Importer*

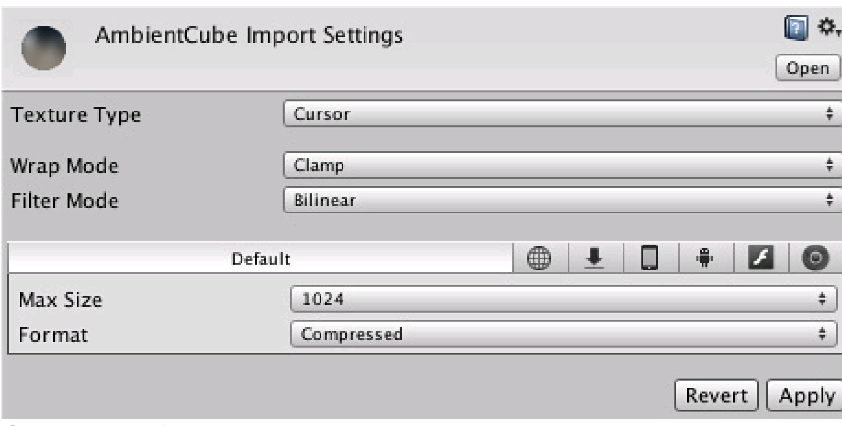
| | |
|---|---|
| **Create from Greyscale** | If this is enabled then Bumpiness and Filtering options will be shown. |
| **Bumpiness** | Control the amount of bumpiness. |
| **Filtering** | Determine how the bumpiness is calculated: |
| **Smooth** | This generates normal maps that are quite smooth. |
| **Sharp** | Also known as a Sobel filter. this generates normal maps that are sharper than Standard. |
| **Wrap Mode** | Selects how the Texture behaves when tiled: |
| **Repeat** | The Texture repeats (tiles) itself |
| **Clamp** | The Texture's edges get stretched |
| **Filter Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations: |
| **Point** | The Texture becomes blocky up close |
| **Bilinear** | The Texture becomes blurry up close |
| **Trilinear** | Like Bilinear, but the Texture also blurs between the different mip levels |

| | |
|---|---|
| **Aniso Level** | Increases texture quality when viewing the texture at a steep angle. Good for floor and ground textures, see below. |



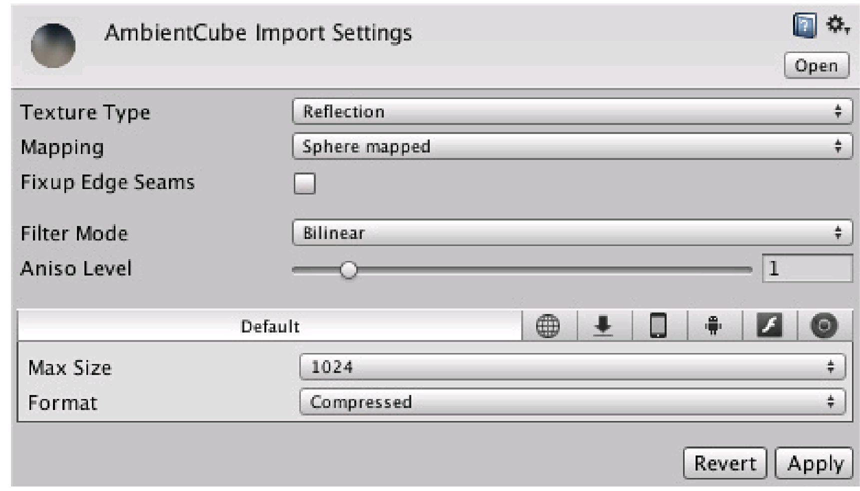

*GUI Settings for the Texture Importer*

| | |
|---|---|
| **Filter Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations: |
| **Point** | The Texture becomes blocky up close |
| **Bilinear** | The Texture becomes blurry up close |
| **Trilinear** | Like Bilinear, but the Texture also blurs between the different mip levels |




*Cursor settings for the Texture Importer*

| | |
|---|---|
| **Wrap Mode** | Selects how the Texture behaves when tiled: |
| **Repeat** | The Texture repeats (tiles) itself |
| **Clamp** | The Texture's edges get stretched |
| **Filter Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations: |
| **Point** | The Texture becomes blocky up close |
| **Bilinear** | The Texture becomes blurry up close |
| **Trilinear** | Like Bilinear, but the Texture also blurs between the different mip levels |

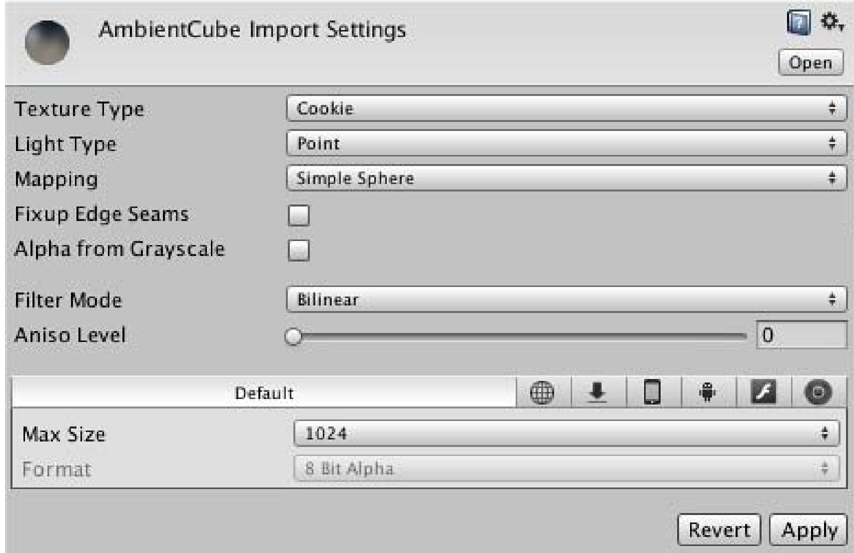



*Reflection Settings in the Texture Importer*

| | |
|---|---|
| **Mapping** | This determines how the texture will be mapped to a cubemap. |

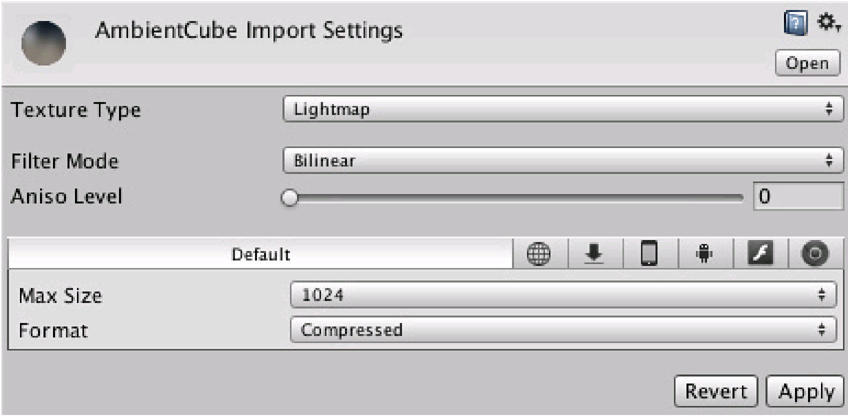
| | |
|---|---|
| **Sphere Mapped** | Maps the texture to a "sphere like" cubemap. |
| **Cylindrical** | Maps the texture to a cylinder, use this when you want to use reflections on objects that are like cylinders. |
| **Simple Sphere** | Maps the texture to a simple sphere, deforming the reflection when you rotate it. |
| **Nice Sphere** | Maps the texture to a sphere, deforming it when you rotate but you still can see the texture's wrap |
| **6 Frames Layout** | The texture contains six images arranged in one of the standard cubemap layouts, cross or sequence (+x -x +y -y +z -z) and the images can be in either horizontal or vertical orientation. |
| **Fixup edge seams** | Removes visual artifacts at the joined edges of the map image(s), which will be visible with glossy reflections. |
| **Filter Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations: |
| **Point** | The Texture becomes blocky up close |
| **Bilinear** | The Texture becomes blurry up close |
| **Trilinear** | Like Bilinear, but the Texture also blurs between the different mip levels |
| **Aniso Level** | Increases texture quality when viewing the texture at a steep angle. Good for floor and ground textures, see below. |

An interesting way to add a lot of visual detail to your scenes is to use **Cookies** - greyscale textures you use to control the precise look of in-game lighting. This is fantastic for making moving clouds and giving an impression of dense foliage. The Light page has more info on all this, but the main thing is that for textures to be usable for cookies you just need to set the **Texture Type** to Cookie.



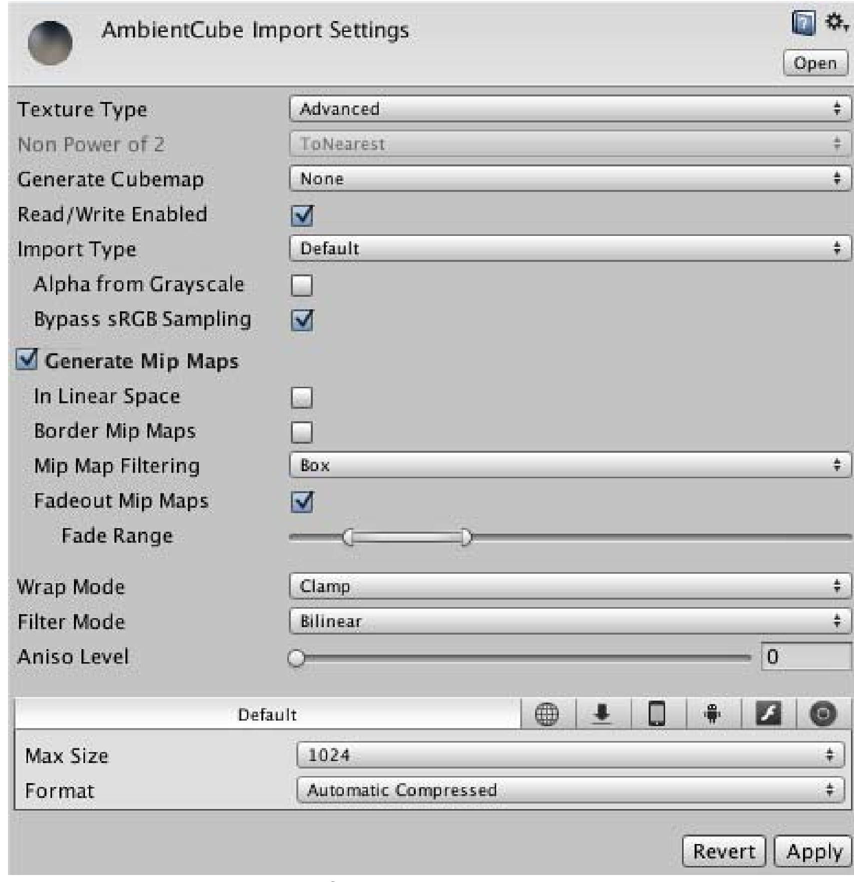

*Cookie Settings in the Texture Importer*

| | |
|---|---|
| **Light Type** | Type of light that the texture will be applied to. (This can be Spotlight, Point or Directional lights). For Directional Lights this texture will tile, so in the texture inspector, you must set the Edge Mode to **Repeat**; for SpotLights You should keep the edges of your cookie texture solid black in order to get the proper effect. In the Texture Inspector, set the Edge Mode to **Clamp**. |
| **Mapping** | (Point light only) Options for mapping the texture onto the spherical cast of the point light. |
| **Sphere Mapped** | Maps the texture to a "sphere like" cubemap. |
| **Cylindrical** | Maps the texture to a cylinder, use this when you want to use reflections on objects that are like cylinders. |
| **Simple Sphere** | Maps the texture to a simple sphere, deforming the reflection when you rotate it. |
| **Nice Sphere** | Maps the texture to a sphere, deforming it when you rotate but you still can see the texture's wrap |
| **6 Frames Layout** | The texture contains six images arranged in one of the standard cubemap layouts, cross or sequence (+x -x +y -y +z -z) and the images can be in either horizontal or vertical orientation. |
| **Fixup edge seams** | (Point light only) Removes visual artifacts at the joined edges of the map image(s). |
| **Alpha from Greyscale** | If enabled, an alpha transparency channel will be generated by the image's existing values of light & dark. |




*Lightmap settings in the Texture Importer*

| | |
|---|---|
| **Filter Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations: |
| **Point** | The Texture becomes blocky up close |
| **Bilinear** | The Texture becomes blurry up close |
| **Trilinear** | Like Bilinear, but the Texture also blurs between the different mip levels |
| **Aniso Level** | Increases texture quality when viewing the texture at a steep angle. Good for floor and ground textures, see below. |




*The Advanced Texture Importer Settings dialog*

| | |
|---|---|
| **Non Power of 2** | If texture has non-power-of-two size, this will define a scaling behavior at import time (for more info see the Texture Sizes section below): |
| **None** | Texture will be padded to the next larger power-of-two size for use with GUITexture component. |
| **To nearest** | Texture will be scaled to the nearest power-of-two size at import time. For instance 257x511 texture will become 256x512. Note that PVRTC formats require textures to be square (width equal to height), therefore final size will be upscaled to 512x512. |
| **To larger** | Texture will be scaled to the next larger power-of-two size at import time. For instance 257x511 texture will become 512x512. |
| **To smaller** | Texture will be scaled to the next smaller power-of-two size at import time. For instance 257x511 texture will become 256x256. |

| | |
|---|---|
| **Generate Cube Map** | Generates a cubemap from the texture using different generation methods. |
| **Spheremap** | Maps the texture to a "sphere like" cubemap. |
| **Cylindrical** | Maps the texture to a cylinder, use this when you want to use reflections on objects that are like cylinders. |
| **SimpleSpheremap** | Maps the texture to a simple sphere, deforming the reflection when you rotate it. |
| **NiceSpheremap** | Maps the texture to a sphere, deforming it when you rotate but you still can see the texture's wrap |
| **FacesVertical** | The texture contains the six faces of the cube arranged in a vertical strip in the order +x -x +y -y +z -z. |
| **FacesHorizontal** | The texture contains the six faces of the cube arranged in a horizontal strip in the order +x -x +y -y +z -z. |
| **CrossVertical** | The texture contains the six faces of the cube arranged in a vertically oriented cross. |
| **CrossHorizontal** | The texture contains the six faces of the cube arranged in a horizontally oriented cross. |
| **Read/Write Enabled** | Select this to enable access to the texture data from scripts (GetPixels, SetPixels and other Texture2D functions). Note however that a copy of the texture data will be made, doubling the amount of memory required for texture asset. Use only if absolutely necessary. This is only valid for uncompressed and DTX compressed textures, other types of compressed textures cannot be read from. Disabled by default. |
| **Import Type** | The way the image data is interpreted. |
| **Default** | Standard texture. |
| **Normal Map** | Texture is treated as a normal map (enables other options) |
| **Lightmap** | Texture is treated as a lightmap (disables other options) |
| **Alpha from grayscale** | (Default mode only) Generates the alpha channel from the luminance information in the image |
| **Create from grayscale** | (Normal map mode only) Creates the map from the luminance information in the image |
| **Bypass sRGB sampling** | (Default mode only) Use the exact colour values from the image rather than compensating for gamma (useful when the texture is for GUI or used as a way to encode non-image data) |
| **Generate Mip Maps** | Select this to enable mip-map generation. Mip maps are smaller versions of the texture that get used when the texture is very small on screen. For more info, see Mip Maps below. |
| **In Linear Space** | Generate mipmaps in linear colour space. |
| **Border Mip Maps** | Select this to avoid colors seeping out to the edge of the lower Mip levels. Used for light cookies (see below). |
| **Mip Map Filtering** | Two ways of mip map filtering are available to optimize image quality: |
| **Box** | The simplest way to fade out the mipmaps - the mip levels become smoother and smoother as they go down in size. |
| **Kaiser** | A sharpening Kaiser algorithm is run on the mip maps as they go down in size. If your textures are too blurry in the distance, try this option. |
| **Fade Out Mipmaps** | Enable this to make the mipmaps fade to gray as the mip levels progress. This is used for detail maps. The left most scroll is the first mip level to begin fading out at. The rightmost scroll defines the mip level where the texture is completely grayed out |
| **Wrap Mode** | Selects how the Texture behaves when tiled: |
| **Repeat** | The Texture repeats (tiles) itself |
| **Clamp** | The Texture's edges get stretched |
| **Filter Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations: |
| **Point** | The Texture becomes blocky up close |
| **Bilinear** | The Texture becomes blurry up close |
| **Trilinear** | Like Bilinear, but the Texture also blurs between the different mip levels |
| **Aniso Level** | Increases texture quality when viewing the texture at a steep angle. Good for floor and ground textures, see below. |

## Per-Platform Overrides

When you are building for different platforms, you have to think about the resolution of your textures for the target platform, the size and the quality. You can set default options and then override the defaults for a specific platform.

*Default settings for all platforms.*

| | |
|---|---|
| **Max Texture Size** | The maximum imported texture size. Artists often prefer to work with huge textures - scale the texture down to a suitable size with this. |
| **Texture Format** | What internal representation is used for the texture. This is a tradeoff between size and quality. In the examples below we show the final size of a in-game texture of 256 by 256 pixels: |

| | |
|---|---|
| **Compressed** | Compressed RGB texture. This is the most common format for diffuse textures. 4 bits per pixel (32 KB for a 256x256 texture). |
| **16 bit** | Low-quality truecolor. Has 16 levels of red, green, blue and alpha. |
| **Truecolor** | Truecolor, this is the highest quality. At 256 KB for a 256x256 texture. |

If you have set the **Texture Type** to **Advanced** then the **Texture Format** has different values.

## Desktop

| | |
|---|---|
| **Texture Format** | What internal representation is used for the texture. This is a tradeoff between size and quality. In the examples below we show the final size of an in-game texture of 256 by 256 pixels: |
| **RGB Compressed DXT1** | Compressed RGB texture. This is the most common format for diffuse textures. 4 bits per pixel (32 KB for a 256x256 texture). |
| **RGBA Compressed DXT5** | Compressed RGBA texture. This is the main format used for diffuse & specular control textures. 1 byte/pixel (64 KB for a 256x256 texture). |
| **RGB 16 bit** | 65 thousand colors with no alpha. Compressed DXT formats use less memory and usually look better. 128 KB for a 256x256 texture. |
| **RGB 24 bit** | Truecolor but without alpha. 192 KB for a 256x256 texture. |
| **Alpha 8 bit** | High quality alpha channel but without any color. 64 KB for a 256x256 texture. |
| **RGBA 16 bit** | Low-quality truecolor. Has 16 levels of red, green, blue and alpha. Compressed DXT5 format uses less memory and usually looks better. 128 KB for a 256x256 texture. |
| **RGBA 32 bit** | Truecolor with alpha - this is the highest quality. At 256 KB for a 256x256 texture, this one is expensive. Most of the time, **DXT5** offers sufficient quality at a much smaller size. The main way this is used is for normal maps, as DXT compression there often carries a visible quality loss. |

## iOS

| | |
|---|---|
| **Texture Format** | What internal representation is used for the texture. This is a tradeoff between size and quality. In the examples below we show the final size of a in-game texture of 256 by 256 pixels: |
| **RGB Compressed PVRTC 4 bits** | Compressed RGB texture. This is the most common format for diffuse textures. 4 bits per pixel (32 KB for a 256x256 texture) |
| **RGBA Compressed PVRTC 4 bits** | Compressed RGBA texture. This is the main format used for diffuse & specular control textures or diffuse textures with transparency. 4 bits per pixel (32 KB for a 256x256 texture) |
| **RGB Compressed PVRTC 2 bits** | Compressed RGB texture. Lower quality format suitable for diffuse textures. 2 bits per pixel (16 KB for a 256x256 texture) |
| **RGBA Compressed PVRTC 2 bits** | Compressed RGBA texture. Lower quality format suitable for diffuse & specular control textures. 2 bits per pixel (16 KB for a 256x256 texture) |
| **RGB Compressed DXT1** | Compressed RGB texture. This format is not supported on iOS, but kept for backwards compatibility with desktop projects. |
| **RGBA Compressed DXT5** | Compressed RGBA texture. This format is not supported on iOS, but kept for backwards compatibility with desktop projects. |
| **RGB 16 bit** | 65 thousand colors with no alpha. Uses more memory than PVRTC formats, but could be more suitable for UI or crisp textures without gradients. 128 KB for a 256x256 texture. |
| **RGB 24 bit** | Truecolor but without alpha. 192 KB for a 256x256 texture. |
| **Alpha 8 bit** | High quality alpha channel but without any color. 64 KB for a 256x256 texture. |
| **RGBA 16 bit** | Low-quality truecolor. Has 16 levels of red, green, blue and alpha. Uses more memory than PVRTC formats, but can be handy if you need exact alpha channel. 128 KB for a 256x256 texture. |
| **RGBA 32 bit** | Truecolor with alpha - this is the highest quality. At 256 KB for a 256x256 texture, this one is expensive. Most of the time, **PVRTC** formats offers sufficient quality at a much smaller size. |
| **Compression quality** | Choose Fast for quickest performance, Best for the best image quality and Normal for a balance between the two. |

## Android

| | |
|---|---|
| **Texture Format** | What internal representation is used for the texture. This is a tradeoff between size and quality. In the examples below we show the final size of a in-game texture of 256 by 256 pixels: |
| **RGB Compressed DXT1** | Compressed RGB texture. Supported by Nvidia Tegra. 4 bits per pixel (32 KB for a 256x256 texture). |

| | |
|---|---|
| **RGBA Compressed DXT5** | Compressed RGBA texture. Supported by Nvidia Tegra. 6 bits per pixel (64 KB for a 256x256 texture). |
| **RGB Compressed ETC 4 bits** | Compressed RGB texture. This is the default texture format for Android projects. ETC1 is part of OpenGL ES 2.0 and is supported by all OpenGL ES 2.0 GPUs. It does not support alpha. 4 bits per pixel (32 KB for a 256x256 texture) |
| **RGB Compressed PVRTC 2 bits** | Compressed RGB texture. Supported by Imagination PowerVR GPUs. 2 bits per pixel (16 KB for a 256x256 texture) |
| **RGBA Compressed PVRTC 2 bits** | Compressed RGBA texture. Supported by Imagination PowerVR GPUs. 2 bits per pixel (16 KB for a 256x256 texture) |
| **RGB Compressed PVRTC 4 bits** | Compressed RGB texture. Supported by Imagination PowerVR GPUs. 4 bits per pixel (32 KB for a 256x256 texture) |
| **RGBA Compressed PVRTC 4 bits** | Compressed RGBA texture. Supported by Imagination PowerVR GPUs. 4 bits per pixel (32 KB for a 256x256 texture) |
| **RGB Compressed ATC 4 bits** | Compressed RGB texture. Supported by Qualcomm Snapdragon. 4 bits per pixel (32 KB for a 256x256 texture). |
| **RGBA Compressed ATC 8 bits** | Compressed RGBA texture. Supported by Qualcomm Snapdragon. 6 bits per pixel (64 KB for a 256x256 texture). |
| **RGB 16 bit** | 65 thousand colors with no alpha. Uses more memory than the compressed formats, but could be more suitable for UI or crisp textures without gradients. 128 KB for a 256x256 texture. |
| **RGB 24 bit** | Truecolor but without alpha. 192 KB for a 256x256 texture. |
| **Alpha 8 bit** | High quality alpha channel but without any color. 64 KB for a 256x256 texture. |
| **RGBA 16 bit** | Low-quality truecolor. The default compression for the textures with alpha channel. 128 KB for a 256x256 texture. |
| **RGBA 32 bit** | Truecolor with alpha - this is the highest quality compression for the textures with alpha. 256 KB for a 256x256 texture. |
| **Compression quality** | Choose Fast for quickest performance, Best for the best image quality and Normal for a balance between the two. |

Unless you're targeting a specific hardware, like Tegra, we'd recommend using ETC1 compression. If needed you could store an external alpha channel and still benefit from lower texture footprint. If you absolutely want to store an alpha channel in a texture, RGBA16 bit is the compression supported by all hardware vendors.

Textures can be imported from DDS files but only DXT or uncompressed pixel formats are currently supported.

If your app utilizes an unsupported texture compression, the textures will be uncompressed to RGBA 32 and stored in memory along with the compressed ones. So in this case you lose time decompressing textures and lose memory storing them twice. It may also have a very negative impact on rendering performance.

# Flash

| | |
|---|---|
| **Format** | Image format |
| **RGB JPG Compressed** | RGB image data compressed in JPG format |
| **RGBA JPG Compressed** | RGBA image data (ie, with alpha) compressed in JPG format |
| **RGB 24-bit** | Uncompressed RGB image data, 8 bits per channel |
| **RGBA 32-bit** | Uncompressed RGBA image data, 8 bits per channel |

## Details

**Supported Formats**
Unity can read the following file formats: PSD, TIFF, JPG, TGA, PNG, GIF, BMP, IFF, PICT. It should be noted that Unity can import multi-layer PSD & TIFF files just fine. They are flattened automatically on import but the layers are maintained in the assets themselves, so you don't lose any of your work when using these file types natively. This is important as it allows you to just have one copy of your textures that you can use from Photoshop, through your 3D modelling app and into Unity.

**Texture Sizes**
Ideally texture sizes should be powers of two on the sides. These sizes are as follows: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 or 2048 pixels. The textures do not have to be square, i.e. width can be different from height.

It is possible to use other (non power of two) texture sizes with Unity. Non power of two texture sizes work best when used on GUI Textures, however if used on anything else they will be converted to an uncompressed RGBA 32 bit format. That means they will take up more video memory (compared to PVRT(iOS)/DXT(Desktop) compressed textures), will be slower to load and

slower to render (if you are on iOS mode). In general you'll use non power of two sizes only for GUI purposes.

Non power of two texture assets can be scaled up at import time using the **Non Power of 2** option in the advanced texture type in the import settings. Unity will scale texture contents as requested, and in the game they will behave just like any other texture, so they can still be compressed and very fast to load.

One potential problem with using non power of two textures this is that Unity will convert these textures internally to power of two, and this stretching process can introduce minor visual artefacts.

**UV Mapping**
When mapping a 2D texture onto a 3D model, some sort of wrapping is done. This is called **UV mapping** and is done in your 3D modelling app. Inside Unity, you can scale and move the texture using Materials. Scaling normal & detail maps is especially useful.

**Mip Maps**
Mip Maps are a list of progressively smaller versions of an image, used to optimise performance on real-time 3D engines. Objects that are far away from the camera use the smaller texture versions. Using mip maps uses 33% more memory, but not using them can be a huge performance loss. You should always use mipmaps for in-game textures; the only exceptions are textures that will never be minified (e.g. GUI textures).

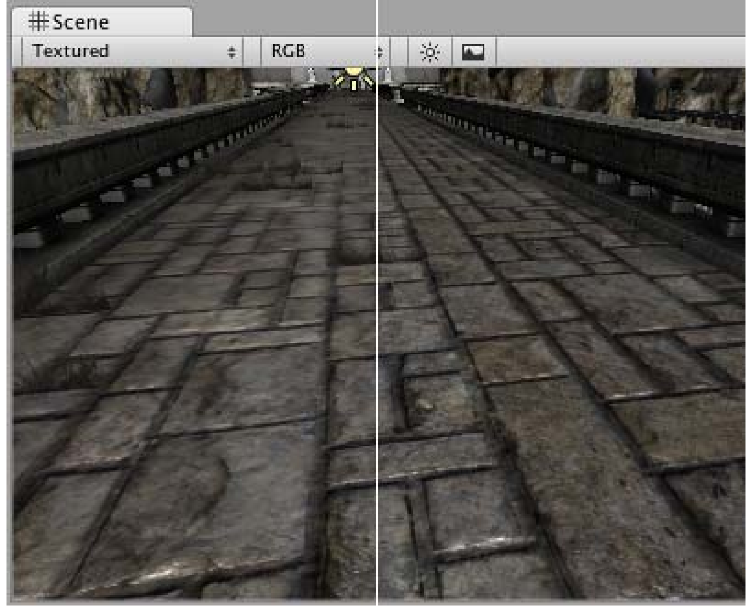
**Normal Maps**
Normal maps are used by normal map shaders to make low-polygon models look as if they contain more detail. Unity uses normal maps encoded as RGB images. You also have the option to generate a normal map from a grayscale height map image.

**Detail Maps**
If you want to make a terrain, you normally use your main texture to show where there are areas of grass, rocks sand, etc... If your terrain has a decent size, it will end up very blurry. Detail textures hide this fact by fading in small details as your main texture gets up close.

When drawing detail textures, a neutral gray is invisible, white makes the main texture twice as bright and black makes the main texture completely black.

**Reflections (Cube Maps)**
If you want to use texture for reflection maps (e.g. use the **Reflective** builtin shaders), you need to use Cubemap Textures.

**Anisotropic filtering**
Anisotropic filtering increases texture quality when viewed from a grazing angle, at some expense of rendering cost (the cost is entirely on the graphics card). Increasing anisotropy level is usually a good idea for ground and floor textures. In Quality Settings anisotropic filtering can be forced for all textures or disabled completely.



*No anisotropy (left) / Maximum anisotropy (right) used on the ground texture*

Page last updated: 2007-11-16

---

# Procedural Materials

Unity incorporates a new asset type known as **Procedural Materials**. These are essentially the same as standard Materials except that the textures they use can be generated at runtime rather than being predefined and stored.

The script code that generates a texture procedurally will typically take up much less space in storage and transmission than a bitmap image and so Procedural Materials can help reduce download times. Additionally, the generation script can be equipped with parameters that can be changed in order to vary the visual properties of the material at runtime. These properties can be anything from color variations to the size of bricks in a wall. Not only does this mean that many variations can be generated from a single Procedural Material but also that the material can be animated on a frame-by-frame basis. Many interesting visual effects are possible - imagine a character gradually turning to stone or acid damaging a surface as it touches.

Unity's Procedural Material system is based around an industry standard product called Substance, developed by Allegorithmic

## Supported Platforms

In Unity, Procedural Materials are fully supported for standalone and webplayer build targets only (Windows and Mac OS X). For all other platforms, Unity will pre-render or *bake* them into ordinary Materials during the build. Although this clearly negates the runtime benefits of procedural generation, it is still useful to be able to create variations on a basic material in the editor.

## Adding Procedural Materials to a Project

A Procedural Material is supplied as a Substance Archive file (SBSAR) which you can import like any other asset (drag and drop directly onto the Assets folder or use **Assets->Import New Asset...**). A Substance Archive asset contains one or more Procedural Materials and contains all the scripts and images required by these. Uncompiled SBS files are not supported.

Although they are implemented differently, Unity handles a Procedural Material just like any other Material. To assign a Procedural Material to a mesh, for example, you just drag and drop it onto the mesh exactly as you would with any other Material.

## Procedural Properties

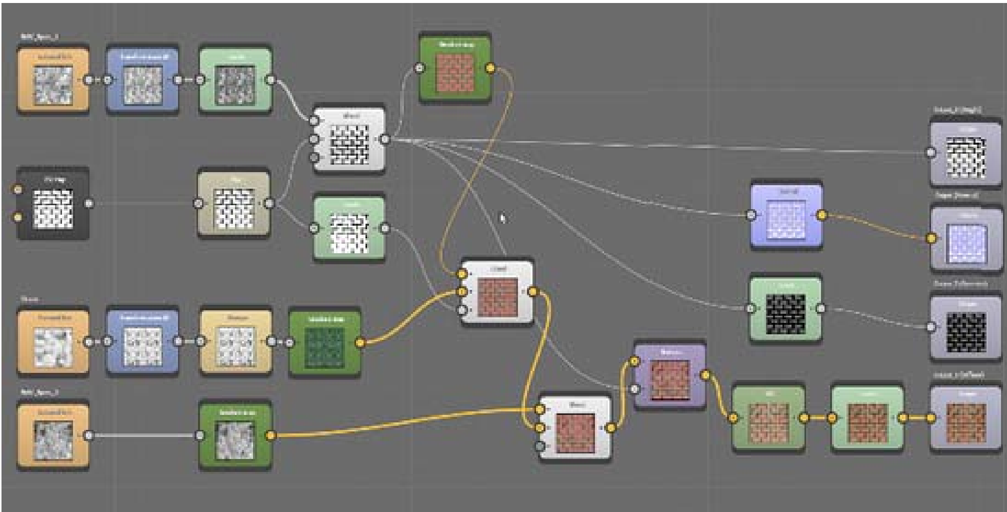
Each Procedural Material is a custom script which generates a particular type of material. These scripts are similar to Unity scripts in that they can have variables exposed for assignment in the inspector. For example, a "Brick Wall" Procedural Material could expose properties that let you set the number of courses of bricks, the colors of the bricks and the color of the mortar. This potentially offers infinite material variations from a single asset. These properties can also be set from a script at runtime in much the same way as the public variables of a MonoBehaviour script.

Procedural Materials can also incorporate complex texture animation. For example, you could animate the hands of the clock or cockroaches running across a floor.

### Creating Procedural Materials From Scratch

Procedural Materials can work with any combination of procedurally generated textures and stored bitmaps. Additionally, included bitmap images can be filtered and modified before use. Unlike a standard Material, a Procedural Material can use vector images in the form of SVG files which allows for resolution-independent textures.

The design tools available for creating Procedural Materials from scratch use visual, node-based editing similar to the kind found in artistic tools. This makes creation accessible to artists who may have little or no coding experience. As an example, here is a screenshot from Allegorithmic's Substance Designer which shows a "brick wall" Procedural Material under construction:



### Obtaining Procedural Materials

Since Unity's Procedural Materials are based on the industry standard Substance product, Procedural Material assets are readily available from internet sources, including Unity's own Asset Store. Allegorithmic's Substance Designer can be used to create Procedural Materials, but there are other applications (3D modelling apps, for example) that incorporate the Substance technology and work just as well with Unity.

### Performance and Optimization

Procedural Materials inherently tend to use less storage than bitmap images. However, the trade-off is that they are based around scripts and running those scripts to generate materials requires some CPU and GPU resources. The more complex your Procedural Materials are, the greater their runtime overhead.

Procedural Materials support a form of caching whereby the material is only updated if its parameters have changed since it was last generated. Further to this, some materials may have many properties that could theoretically be changed and yet only a few will ever need to change at runtime. In such cases, you can inform Unity about the variables that will not change to help it cache as much data as possible from the previous generation of the material. This will often improve performance significantly.

Procedural Materials can refer to hidden, system-wide, variables, such as elapsed time or number of Procedural Material instances (this data can be useful for animations). Changes in the values of these variables can still force a Procedural Material to update even if none of the explicitly defined parameters change.

Procedural Materials can also be used purely as a convenience in the editor (ie, you can generate a standard Material by setting the parameters of a Procedural Material and then "baking" it). This will remove the runtime overhead of material generation but naturally, the baked materials can't be changed or animated during gameplay.

### Using the Substance Player to Analyze Performance

Since the complexity of a Procedural Material can affect runtime performance, Allegorithmic incorporates profiling features in its *Substance Player* tool. This tool is available to download for free from Allegorithmic's website.

Substance Player uses the same optimized rendering engine as the one integrated into Unity, so its rendering measurement is more representative of performance in Unity than that of Substance Designer.

Page last updated: 2012-10-12

---

# Video Files

**Note:** This is a **Pro/Advanced** feature only.

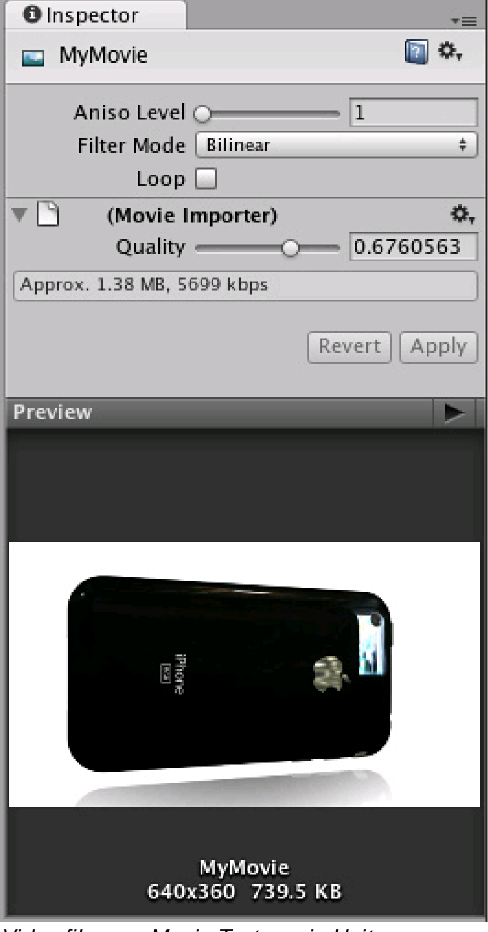
## Desktop

**Movie Textures** are animated **Textures** that are created from a video file. By placing a video file in your project's **Assets Folder**, you can import the video to be used exactly as you would use a regular Texture.

Video files are imported via Apple QuickTime. Supported file types are what your QuickTime installation can play (usually **.mov**, **.mpg**, **.mpeg**, **.mp4**, **.avi**, **.asf**). On Windows movie importing requires Quicktime to be installed (download here).

### Properties

The Movie Texture **Inspector** is very similar to the regular Texture Inspector.




*Video files are Movie Textures in Unity*

| | |
|---|---|
| **Aniso Level** | Increases Texture quality when viewing the texture at a steep angle. Good for floor and ground textures |
| **Filtering Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations |
| **Loop** | If enabled, the movie will loop when it finishes playing |
| **Quality** | Compression of the Ogg Theora video file. A higher value means higher quality, but larger file size |

## Details

When a video file is added to your Project, it will automatically be imported and converted to **Ogg Theora** format. Once your Movie Texture has been imported, you can attach it to any **GameObject** or **Material**, just like a regular Texture.

### Playing the Movie

Your Movie Texture will not play automatically when the game begins running. You must use a short script to tell it when to play.

```
// this line of code will make the Movie Texture begin playing
renderer.material.mainTexture.Play();
```

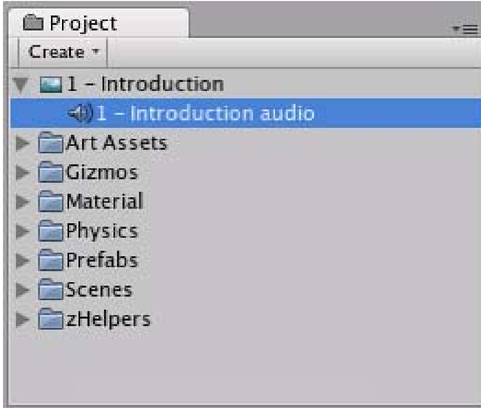
Attach the following script to toggle Movie playback when the space bar is pressed:

```
function Update () {
	if (Input.GetButtonDown ("Jump")) {
		if (renderer.material.mainTexture.isPlaying) {
			renderer.material.mainTexture.Pause();
		}
		else {
			renderer.material.mainTexture.Play();
		}
	}
```

```
}
```

For more information about playing Movie Textures, see the Movie Texture Script Reference page

**Movie Audio**
When a Movie Texture is imported, the audio track accompanying the visuals are imported as well. This audio appears as an **AudioClip** child of the Movie Texture.




*The video's audio track appears as a child of the Movie Texture in the **Project View***

To play this audio, the Audio Clip must be attached to a GameObject, like any other Audio Clip. Drag the Audio Clip from the Project View onto any GameObject in the Scene or Hierarchy View. Usually, this will be the same GameObject that is showing the Movie. Then use audio.Play() to make the the movie's audio track play along with its video.

## iOS

Movie Textures are not supported on iOS. Instead, full-screen streaming playback is provided using Handheld.PlayFullScreenMovie.

**You need to keep your videos inside the *StreamingAssets* folder located in your Project directory.**

Unity iOS supports any movie file types that play correctly on an iOS device, implying files with the extensions **.mov**, **.mp4**, **.mpv**, and **.3gp** and using one of the following compression standards:

- H.264 Baseline Profile Level 3.0 video
- MPEG-4 Part 2 video

For more information about supported compression standards, consult the iPhone SDK MPMoviePlayerController Class Reference.

As soon as you call iPhoneUtils.PlayMovie or iPhoneUtils.PlayMovieURL, the screen will fade from your current content to the designated background color. It might take some time before the movie is ready to play but in the meantime, the player will continue displaying the background color and may also display a progress indicator to let the user know the movie is loading. When playback finishes, the screen will fade back to your content.

**The video player does not respect switching to mute while playing videos**
As written above, video files are played using Apple's embedded player (as of SDK 3.2 and iPhone OS 3.1.2 and earlier). This contains a bug that prevents Unity switching to mute.

**The video player does not respect the device's orientation**
The Apple video player and iPhone SDK do not provide a way to adjust the orientation of the video. A common approach is to manually create two copies of each movie in landscape and portrait orientations. Then, the orientation of the device can be determined before playback so the right version of the movie can be chosen.

## Android

Movie Textures are not supported on Android. Instead, full-screen streaming playback is provided using Handheld.PlayFullScreenMovie.

**You need to keep your videos inside of the *StreamingAssets* folder located in your Project directory.**

Unity Android supports any movie file type supported by Android, (ie, files with the extensions **.mp4** and **.3gp**) and using one of the following compression standards:

- H.263
- H.264 AVC
- MPEG-4 SP

However, device vendors are keen on expanding this list, so some Android devices are able to play formats other than those listed, such as HD videos.

For more information about the supported compression standards, consult the Android SDK Core Media Formats documentation.

As soon as you call iPhoneUtils.PlayMovie or iPhoneUtils.PlayMovieURL, the screen will fade from your current content to the designated background color. It might take some time before the movie is ready to play. In the meantime, the player will continue displaying the background color and may also display a progress indicator to let the user know the movie is loading. When playback finishes, the screen will fade back to your content.

Page last updated: 2007-11-16

---

# Audio Files

As with Meshes or Textures, the workflow for **Audio File** assets is designed to be smooth and trouble free. Unity can import almost every common file format but there are a few details that are useful to be aware of when working with Audio Files.

Audio in Unity is either *Native* or *Compressed*. Unity supports most common formats (see the list below) and will import an audio file when it is added to the project. The default mode is *Native*, where the audio data from the original file is imported unchanged. However, Unity can also compress the audio data on import, simply by enabling the *Compressed* option in the importer. (iOS projects can make use of the hardware decoder - see the iOS documentation for further details). The difference between Native and Compressed modes are as follows:-

- **Native**: Use Native (WAV, AIFF) audio for short sound effects. The audio data will be larger but sounds won't need to be decoded at runtime.
- **Compressed**: The audio data will be small but will need to be decompressed at runtime, which entails a processing overhead. Depending on the target, Unity will encode the audio to either Ogg Vorbis(Mac/PC/Consoles) or MP3 (Mobile platforms). For the best sound quality, supply the audio in an uncompressed format such as WAV or AIFF (containing PCM data) and let Unity do the encoding. If you are targeting Mac and PC platforms only (including both standalones and webplayers) then importing an Ogg Vorbis file will not degrade the quality. However, on mobile platforms, Ogg Vorbis and MP3 files will be re-encoded to MP3 on import, which will introduce a slight quality degradation.

Any Audio File imported into Unity is available from scripts as an **Audio Clip** instance, which is effectively just a container for the audio data. The clips must be used in conjunction with **Audio Sources** and an **Audio Listener** in order to actually generate sound. When you attach your clip to an object in the game, it adds an Audio Source component to the object, which has **Volume**, **Pitch** and a numerous other properties. While a Source is playing, an Audio Listener can "hear" all sources within range, and the combination of those sources gives the sound that will actually be heard through the speakers. There can be only one Audio Listener in your scene, and this is usually attached to the **Main Camera**.

### Supported Formats

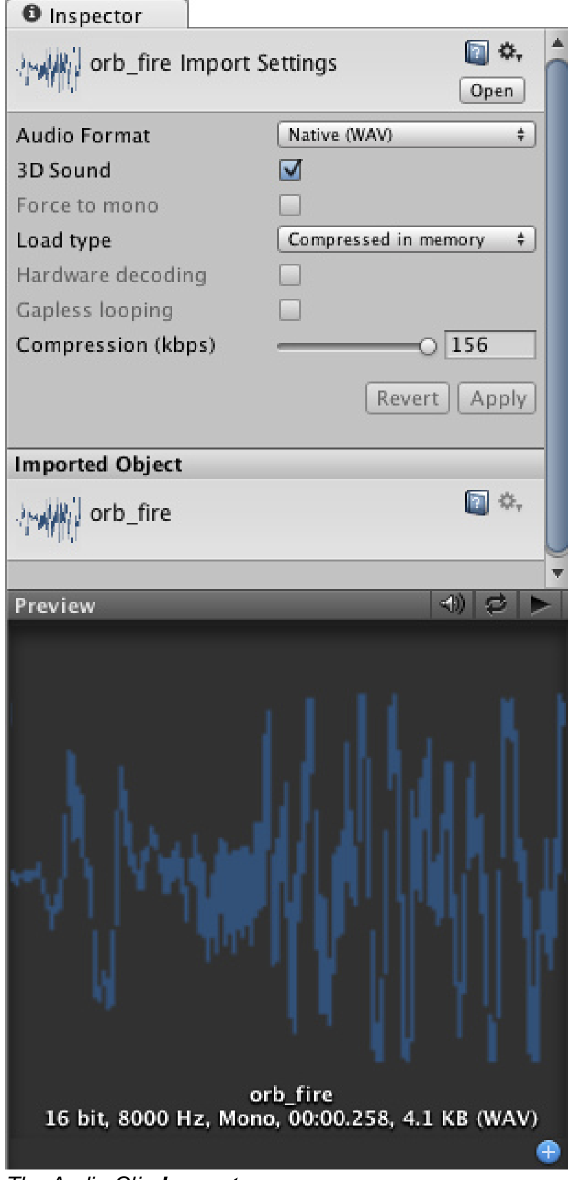
| Format | Compressed as (Mac/PC) | Compressed as (Mobile) |
|---|---|---|
| MPEG(1/2/3) | Ogg Vorbis | MP3 |
| Ogg Vorbis | Ogg Vorbis | MP3 |
| WAV | Ogg Vorbis | MP3 |
| AIFF | Ogg Vorbis | MP3 |
| MOD | - | - |
| IT | - | - |
| S3M | - | - |

| XM | - | - |
|---|---|---|

See the Sound chapter in the Creating Gameplay section of this manual for more information on using sound in Unity.

## Audio Clip

**Audio Clips** contain the audio data used by Audio Sources. Unity supports mono, stereo and multichannel audio assets (up to eight channels). The audio file formats that Unity can import are **.aif**, **.wav**, **.mp3**, and **.ogg**. Unity can also import tracker modules in the **.xm**, **.mod**, **.it**, and **.s3m** formats. The tracker module assets behave the same way as any other audio assets in Unity although no waveform preview is available in the asset import inspector.




*The Audio Clip* ***Inspector***

### Properties

| | |
|---|---|
| **Audio Format** | The specific format that will be used for the sound at runtime. |
| **Native** | This option offers higher quality at the expense of larger file size and is best for very short sound effects. |
| **Compressed** | The compression results in smaller files but with somewhat lower quality compared to native audio. This format is best for medium length sound effects and music. |
| **3D Sound** | If enabled, the sound will play back in 3D space. Both Mono and Stereo sounds can be played in 3D. |
| **Force to mono** | If enabled, the audio clip will be down-mixed to a single channel sound. |
| **Load Type** | The method Unity uses to load audio assets at runtime. |

| | |
|---|---|
| **Decompress on load** | Audio files will be decompressed as soon as they are loaded. Use this option for smaller compressed sounds to avoid the performance overhead of decompressing on the fly. Be aware that decompressing sounds on load will use about ten times more memory than keeping them compressed, so don't use this option for large files. |
| **Compressed in memory** | Keep sounds compressed in memory and decompress while playing. This option has a slight performance overhead (especially for Ogg/Vorbis compressed files) so only use it for bigger files where decompression on load would use a prohibitive amount of memory. Note that, due to technical limitations, this option will silently switch to *Stream From Disc* (see below) for Ogg Vorbis assets on platforms that use FMOD audio. |
| **Stream from disc** | Stream audio data directly from disc. The memory used by this option is typically a small fraction of the file size, so it is very useful for music or other very long tracks. For performance reasons, it is usually advisable to stream only one or two files from disc at a time but the of streams that can comfortably be handled depends on the hardware. |
| **Compression** | Amount of Compression to be applied to a **Compressed** clip. Statistics about the file size can be seen under the slider. A good approach to tuning this value is to drag the slider to a place that leaves the playback "good enough" while keeping the file small enough for your distribution requirements. |

| | |
|---|---|
| **Hardware Decoding** | (iOS only) On iOS devices, Apple's hardware decoder can be used resulting in lower CPU overhead during decompression. Check out platform specific details for more info. |

| | |
|---|---|
| **Gapless looping** | (Android/iOS only) Use this when compressing a seamless looping audio source file (in a non-compressed PCM format) to ensure perfect continuity is preserved at the seam. Standard MPEG encoders introduce a short silence at the loop point, which will be audible as a brief "click" or "pop". |

### Importing Audio Assets

Unity supports both *Compressed* and *Native* Audio. Any type of file (except MP3/Ogg Vorbis) will be initially imported as *Native*. Compressed audio files must be decompressed by the CPU while the game is running, but have smaller file size. If *Stream* is checked the audio is decompressed *on the fly*, otherwise it is decompressed completely as soon as it loads. Native PCM formats (WAV, AIFF) have the benefit of giving higher fidelity without increasing the CPU overhead, but files in these formats are typically much larger than compressed files. Module files (.mod,.it,.s3m..xm) can deliver very high quality with an extremely low footprint.

As a general rule of thumb, *Compressed* audio (or modules) are best for long files like background music or dialog, while *Native* is better for short sound effects. You should tweak the amount of Compression using the compression slider. Start with high compression and gradually reduce the setting to the point where the loss of sound quality is perceptible. Then, increase it again slightly until the perceived loss of quality disappears.

### Using 3D Audio

If an audio clip is marked as a **3D Sound** then it will be played back so as to simulate its position in the game world's 3D space. 3D sounds emulate the distance and location of sounds by attenuating volume and panning across speakers. Both mono and multiple channel sounds can be positioned in 3D. For multiple channel audio, use the *spread* option on the Audio Source to spread and split out the discrete channels in speaker space. Unity offers a variety of options to control and fine-tune the audio behavior in 3D space - see the Audio Source component reference for further details.

### Platform specific details

## iOS

On mobile platforms compressed audio is encoded as MP3 to take advantage of hardware decompression.

To improve performance, audio clips can be played back using the Apple hardware codec. To enable this option, check the "Hardware Decoding" checkbox in the Audio Importer. Note that only one hardware audio stream can be decompressed at a time, including the background iPod audio.

If the hardware decoder is not available, the decompression will fall back on the software decoder (on iPhone 3GS or later, Apple's software decoder is used in preference to Unity's own decoder (FMOD)).

## Android

On mobile platforms compressed audio is encoded as MP3 to take advantage of hardware decompression.

Page last updated: 2012-08-03

---

## TrackerModules

Tracker Modules are essentially just packages of audio samples that have been modeled, arranged and sequenced programatically. The concept was introduced in the 1980's (mainly in conjunction with the Amiga computer) and has been popular since the early days of game development and demo culture.

Tracker Module files are similar to MIDI files in many ways. The tracks are scores that contain information about when to play the instruments, and at what pitch and volume and from this, the melody and rhythm of the original tune can be recreated. However, MIDI has a disadvantage in that the sounds are dependent on the sound bank available in the audio hardware, so MIDI music can sound different on different computers. In contrast, tracker modules include high quality PCM samples that ensure a similar experience regardless of the audio hardware in use.

### Supported formats

Unity supports the four most common module file formats, namely Impulse Tracker (**.it**), Scream Tracker (**.s3m**), Extended Module File Format (**.xm**), and the original Module File Format (**.mod**).

### Benefits of Using Tracker Modules

Tracker module files differ from mainstream PCM formats (**.aif**, **.wav**, **.mp3**, and **.ogg**) in that they can be very small without a corresponding loss of sound quality. A single sound sample can be modified in pitch and volume (and can have other effects applied), so it essentially acts as an "instrument" which can play a tune without the overhead of recording the whole tune as a sample. As a result, tracker modules lend themselves to games, where music is required but where a large file download would be a problem.

### Third Party Tools and Further References

Currently, the most popular tools to create and edit Tracker Modules are MilkyTracker for OSX and OpenMPT for Windows. For more information and discussion, please see the blog post .mod in Unity from June 2010.

Page last updated: 2011-11-15

---

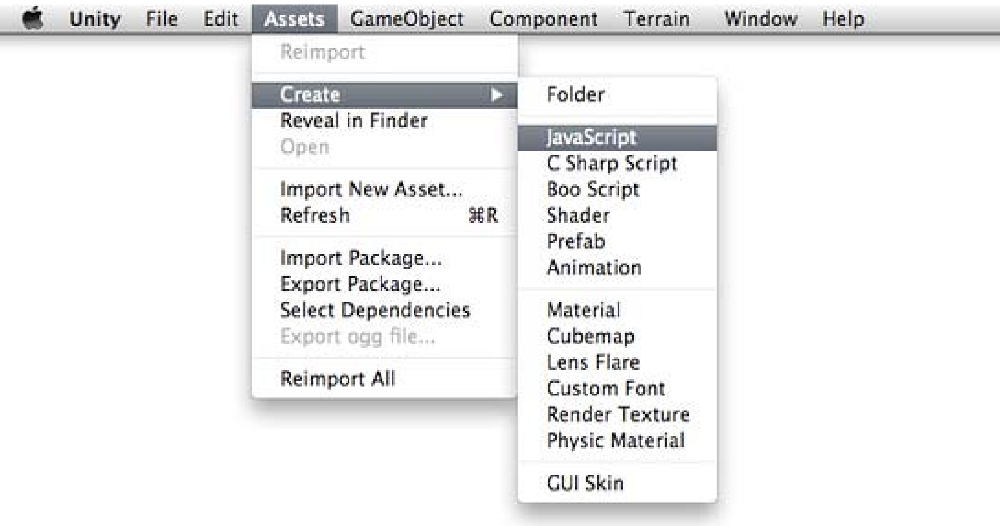
## Scripting

This brief introduction explains how to create and use scripts in a project. For detailed information about the Scripting API, please view the Scripting Reference. For detailed information about creating game play through scripting, please view the Creating Gameplay page of this manual.

Behaviour scripts in Unity can be written in **JavaScript**, **C#**, or **Boo**. It is possible to use any combination of the three languages in a single project, although there are certain restrictions in cases where one script incorporates classes defined in another script.

### Creating New Scripts

Unlike other assets like Meshes or Textures, Script files can be created from within Unity. To create a new script, open the **Assets->Create->JavaScript** (or **Assets->Create->C Sharp Script** or **Assets->Create->Boo Script**) from the main menu. This will create a new script called **NewBehaviourScript** and place it in the selected folder in **Project View**. If no folder is selected in Project View, the script will be created at the root level.




You can edit the script by double-clicking on it in the Project View. This will launch your default text editor as specified in Unity's preferences. To set the default script editor, change the drop-down item in **Unity->Preferences->External Script editor**.

These are the contents of a new, empty behaviour script:

```
function Update () {
}
```

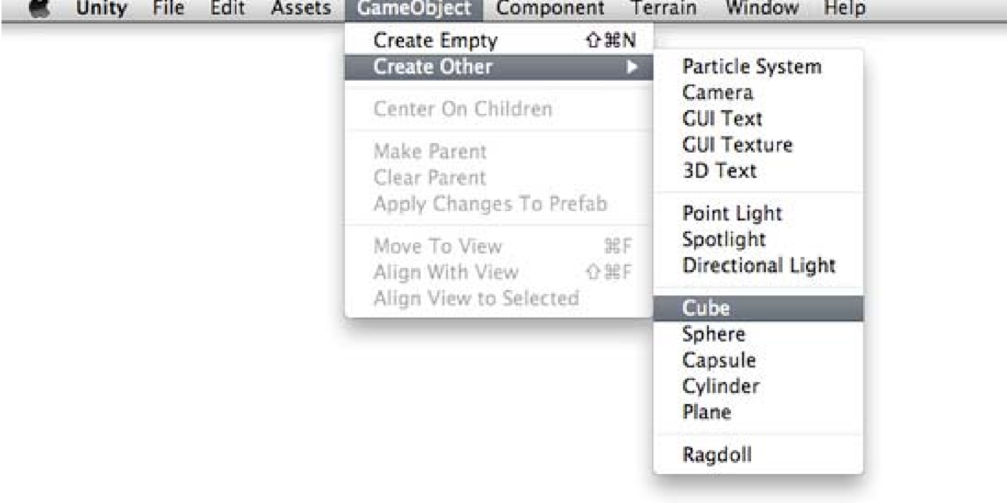
A new, empty script does not do a lot on its own, so let's add some functionality. Change the script to read the following:

```
function Update () {
    print("Hello World");
}
```

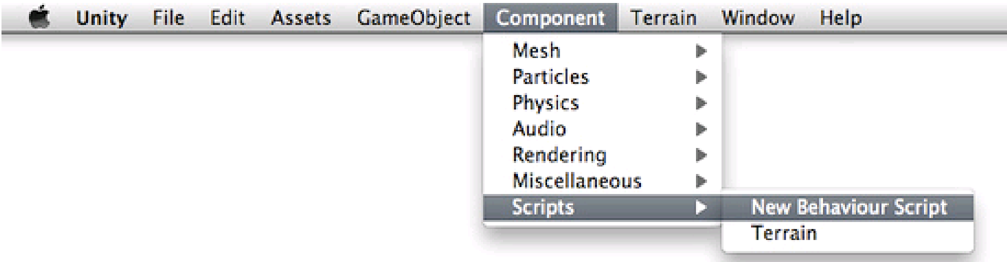
When executed, this code will print "Hello World" to the console. But there is nothing that causes the code to be executed yet. We have to attach the script to an active **GameObject** in the **Scene** before it will be executed.

### Attaching scripts to objects

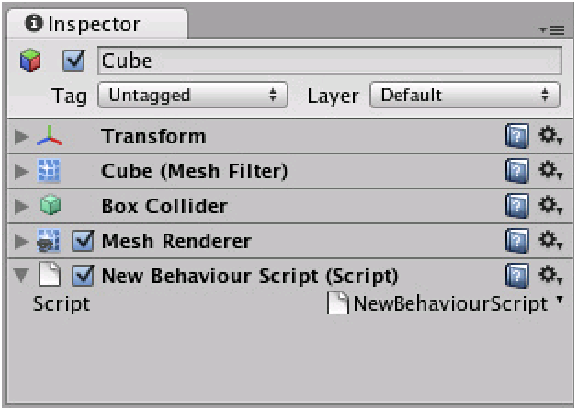
Save the above script and create a new object in the Scene by selecting **GameObject->Create Other->Cube**. This will create a new GameObject called "Cube" in the current Scene.



Now drag the script from the Project View to the Cube (in the Scene or **Hierarchy View**, it doesn't matter). You can also select the Cube and choose **Component->Scripts->New Behaviour Script**. Either of these methods will attach the script to the Cube. Every script you create will appear in the **Component->Scripts** menu.




If you select the Cube and look at the **Inspector**, you will see that the script is now visible. This means it has been attached.




Press **Play** to test your creation. You should see the text "Hello World" appear beside the Play/Pause/Step buttons. Exit play mode when you see it.

Hello World

## Manipulating the GameObject

A **print()** statement can be very handy when debugging your script, but it does not manipulate the GameObject it is attached to. Let's change the script to add some functionality:

```
function Update () {
   transform.Rotate(0, 5*Time.deltaTime, 0);
}
```

If you're new to scripting, it's okay if this looks confusing. These are the important concepts to understand:

1. **function Update () {}** is a container for code that Unity executes multiple times per second (once per frame).
2. **transform** is a reference to the GameObject's Transform Component.
3. **Rotate()** is a function contained in the Transform **Component**.
4. The numbers in-between the commas represent the degrees of rotation around each axis of 3D space: X, Y, and Z.
5. **Time.deltaTime** is a member of the Time class that evens out movement over one second, so the cube will rotate at the same speed no matter how many frames per second your machine is rendering. Therefore, **5 * Time.deltaTime** means 5 degrees per second.

With all this in mind, we can read this code as "every frame, rotate this GameObject's Transform component a small amount so that it will equal five degrees around the Y axis each second."

You can access lots of different Components the same way as we accessed **transform** already. You have to add Components to the GameObject using the **Component** menu. All the Components you can access directly are listed under **Variables** on the GameObject Scripting Reference Page.

For more information about the relationship between GameObjects, Scripts, and Components, please jump ahead to the GameObjects page or Using Components page of this manual.

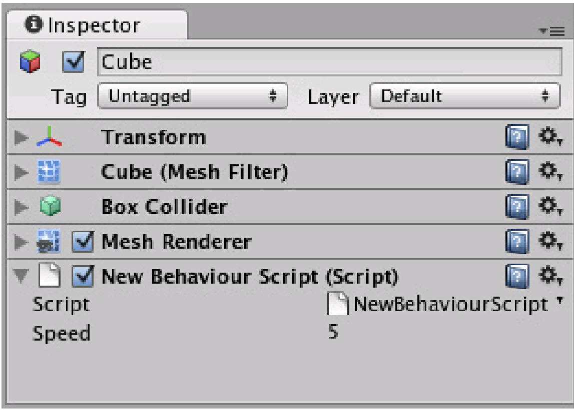
### The Power of Variables

Our script so far will always rotate the Cube 5 degrees each second. We might want it to rotate a different number of degrees per second. We could change the number and save, but then we have to wait for the script to be recompiled and we have to enter Play mode before we see the results. There is a much faster way to do it. We can experiment with the speed of rotation in real-time during Play mode, and it's easy to do.

Instead of typing **5** into the **Rotate()** function, we will declare a **speed** variable and use that in the function. Change the script to the following code and save it:

```
var speed = 5.0;

function Update () {
   transform.Rotate(0, speed*Time.deltaTime, 0);
}
```

Now, select the Cube and look at the Inspector. Notice how our **speed** variable appears.




This variable can now be modified directly in the Inspector. Select it, press **Return** and change the value. You can also right- or option-click on the value and drag the mouse up or down. You can change the variable at any time, even while the game is running.

Hit Play and try modifying the **speed** value. The Cube's rotation speed will change instantly. When you exit Play mode, you'll see that your changes are reverted back to their value before entering Play mode. This way you can play, adjust, and experiment to find the best value, then apply that value permanently.

The technique of changing a variable's value in the Inspector makes it easy to reuse one script on many objects, each with a different variable value. If you attach the script to multiple Cubes, and change the **speed** of each cube, they will all rotate at different speeds even though they use the same script.

### Accessing Other Components

When writing a script Component, you can access other components on the GameObject from within that script.

#### Using the GameObject members

You can directly access any member of the **GameObject** class. You can see a list of all the **GameObject** class members here. If any of the indicated classes are attached to the GameObject as a Component, you can access that Component directly through the script by simply typing the member name. For example, typing **transform** is equivalent to **gameObject.transform**. The **gameObject** is assumed by the compiler, unless you specifically reference a different GameObject.

Typing **this** will be accessing the script Component that you are writing. Typing **this.gameObject** is referring to the GameObject that the script is attached to. You can access the same GameObject by simply typing **gameObject**. Logically, typing **this.transform** is the same as typing **transform**. If you want to access a Component that is not included as a GameObject member, you have to use **gameObject.GetComponent()** which is explained on the next page.

There are many Components that can be directly accessed in any script. For example, if you want to access the **Translate** function of the Transform component, you can just write **transform.Translate()** or **gameObject.transform.Translate()**. This works because all scripts are attached to a GameObject. So when you write **transform** you are implicitly accessing the Transform Component of the GameObject that is being scripted. To be explicit, you write **gameObject.transform**. There is no advantage in one method over the other, it's all a matter of preference for the scripter.

To see a list of all the Components you can access implicitly, take a look at the GameObject page in the Scripting Reference.

**Using GetComponent()**

There are many Components which are not referenced directly as members of the **GameObject** class. So you cannot access them implicitly, you have to access them explicitly. You do this by calling the **GetComponent("component name")** and storing a reference to the result. This is most common when you want to make a reference to another script attached to the GameObject.

Pretend you are writing *Script B* and you want to make a reference to *Script A*, which is attached to the same GameObject. You would have to use **GetComponent()** to make this reference. In Script B, you would simply write:

**scriptA = GetComponent("ScriptA");**

For more help with using **GetComponent()**, take a look at the GetComponent() Script Reference page.

## Accessing variables in other script Components

All scripts attached to your GameObjects are Components. Therefore to get access to a public variable (and methods) in a script you make use of the GetComponent method. For example:

```
function Start () {
  // Print the position of the transform component, for the gameObject this script is attached to
  Debug.Log(gameObject.GetComponent<Transform>.().position);
}
```

In the previous example the GetComponent<T>. function is used to access the position property of the Transform component. The same technique can be used to access a variable in a custom script Component:

```
(MyClass.js)
public var speed : float = 3.14159;

(MyOtherClass.js)
function Start () {
  // Print the speed variable from the MyClass script Component attached to the gameObject
  Debug.Log(gameObject.GetComponent<MyClass>.().speed);
}
```

**Accessing a variable defined in C# from Javascript**

To access variables defined in C# scripts the compiled Assembly containing the C# code must exist when the Javascript code is compiled. Unity performs the compilation in different stages as described in the Script Compilation section in the Scripting Reference. If you want to create a Javascript that uses classes or variables from a C# script just place the C# script in the "Standard Assets", "Pro Standard Assets" or "Plugins" folder and the Javascript outside of these folders. The code inside the "Standard Assets", "Pro Standard Assets" or "Plugins" is compiled first and the code outside is compiled in a later step making the Types defined in the compilation step (your C# script) available to later compilation steps (your Javascript script).

In general the code inside the "Standard Assets", "Pro Standard Assets" or "Plugins" folders, regardless of the language (C#, Javascript or Boo), will be compiled first and available to scripts in subsequent compilation steps.

**Optimizing variable access**

In some circumstances you may be using GetComponent multiple times in your code, or multiple times per frame. Every call to GetComponent does a few extra steps internally to get the reference to the component you require. A more efficient approach is to store the reference to the component for example in your Start() function. As you will be storing the reference and not retrieving directly it is always good practice to check for null references:

```
(MyClass.js)
public var speed : float = 3.14159;

(MyOtherClass.js)
private var myClass : MyClass;
function Start () {
  // Get a reference to the MyClass script Component attached to the gameObject
  myClass = gameObject.GetComponent<MyClass>.();
}
function Update () {
  // Verify that the reference is still valid and print the speed variable
  if(myClass != null)
    Debug.Log (myClass.speed);
}
```

**Static Variables**

It is also possible to declare variables in your classes as static. There will exist one and only one instance of a static variable for a specific class and it can be modified without the need of an instance of a class object:

```
(MyClass.js)
static public var speed : float = 3.14159;

(MyOtherClass.js)
function Start () {
  Debug.Log (MyClass.speed);
}
```

It is recommended to **not** use static variables for object references to make sure unused objects are removed from memory.

### Where to go from here

This was just a short introduction on how to use scripts inside the Editor. For more examples, check out the Unity tutorials, available for free on our Asset Store. You should also read through the Scripting Overview in the Script Reference, which contains a more thorough introduction into scripting with Unity along with pointers to more in-depth information. If you're really stuck, be sure to visit the Unity Answers or Unity Forums and ask questions there. Someone is always willing to help.

Page last updated: 2012-06-28

---

## Asset Store

Unity's **Asset Store** is home to a growing library of free and commercial assets created both by Unity Technologies and also members of the community. A wide variety of assets is available, covering everything from textures, models and animations to whole project examples, tutorials and Editor extensions. The assets are accessed from a simple interface built into the Unity Editor and are downloaded and imported directly into your project.

### Access and Navigation

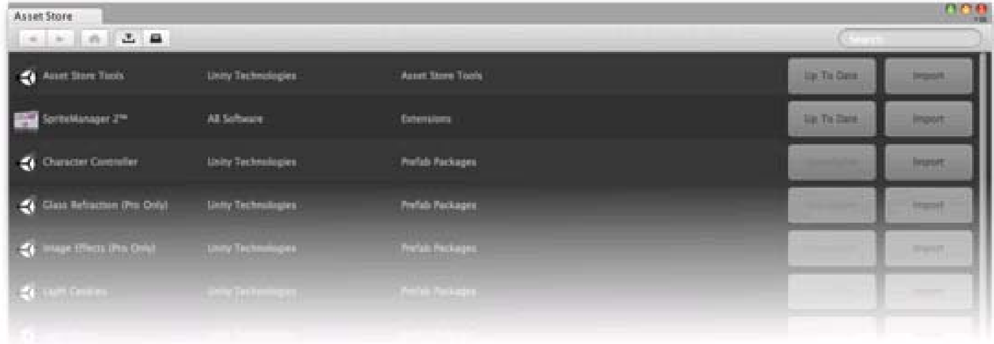
You can open the Asset Store window by selecting **Window->AssetStore** from the main menu. On your first visit, you will be prompted to create a free user account which you will use to access the Store subsequently.

*The Asset Store front page.*

The Store provides a browser-like interface which allows you to navigate either by free text search or by browsing packages and categories. To the left of the main tool bar are the familiar browsing buttons for navigating through the history of viewed items:-

To the right of these are buttons for viewing the Download Manager and for viewing the current contents of your shopping cart.

The Download Manager allows you to view the packages you have already bought and also to find and install any updates. Additionally, the standard packages supplied with Unity can be viewed and added to your project with the same interface.




*The Download Manager.*

### Location of Downloaded Asset Files

You will rarely, if ever, need to access the files downloaded from the Asset Store directly. However, if you do need to, you can find them in

```
~/Library/Unity/Asset Store
```

...on the Mac and in

```
C:\Users\accountName\AppData\Roaming\Unity\Asset Store
```

...on Windows. These folders contain subfolders that correspond to particular Asset Store vendors - the actual asset files are contained in the appropriate subfolders.

Page last updated: 2011-12-09

---

# Asset Server

### Unity Asset Server Overview

The **Unity Asset Server** is an asset and version control system with a graphical user interface integrated into Unity. It is meant to be used by team members working together on a project on different computers either in-person or remotely. The Asset Server is highly optimized for handling large binary assets in order to cope with large multi gigabyte project folders. When uploading assets, **Import Settings** and other meta data about each asset is uploaded to the asset server as well. Renaming and moving files is at the core of the system and well supported.

It is available only for Unity Pro, and is an additional license per client. To purchase an Asset Server Client License, please visit the Unity store at http://unity3d.com/store

### New to Source Control?

If you have never used Source Control before, it can be a little unfriendly to get started with any versioning system. Source Control works by storing an entire collection of all your assets - meshes, textures, materials, scripts, and everything else - in a database on some kind of server. That server might be your home computer, the same one that you use to run Unity. It might be a different computer in your local network. It might be a remote machine colocated in a different part of the world. It could even be a virtual machine. There are a lot of options, but the location of the server doesn't matter at all. The important thing is that you can access it somehow over your network, and that it stores your game data.

In a way, the Asset Server functions as a backup of your Project Folder. You do not directly manipulate the contents of the Asset Server while you are developing. You make changes to your Project locally, then when you are done, you **Commit Changes** to the Project on the Server. This makes your local Project and the Asset Server Project identical.

Now, when your fellow developers make a change, the Asset Server is identical to their Project, but not yours. To synchronize your local Project, you request to **Update from Server**. Now, whatever changes your team members have made will be downloaded from the server to your local Project.

This is the basic workflow for using the Asset Server. In addition to this basic functionality, the Asset Server allows for rollback to previous versions of assets, detailed file comparison, merging two different scripts, resolving conflicts, and recovering deleted assets.

### Setting up the Asset Server

The Asset Server requires a one time server setup and a client configuration for each user. You can read about how to do that in the **Asset Server Setup page**.

The rest of this guide explains how to deploy, administrate, and regularly use the Asset Server.

# Daily use of the Asset Server

This section explains the common tasks, workflow and best practices for using the Asset Server on a day-to-day basis.

### Getting Started

If you are joining a team that has a lot of work stored on the Asset Server already, this is the quickest way to get up and running correctly. If you are starting your own project from scratch, you can skip down to the Workflow Fundamentals section.

1. Create a new empty Project with no packages imported
2. Go to **Edit->Project Settings->Editor** and select **Asset Server** as the version control mode
3. From the menubar, select **Window->Version**
4. Click the **Connection** button

5. Enter your user name and password (provided by your Asset Server administrator)
6. Click **Show Projects** and select the desired project
7. Click **Connect**
8. Click the **Update** tab
9. Click the **Update** button
10. If a conflict occurs, discard all local versions
11. Wait for the update to complete
12. You are ready to go

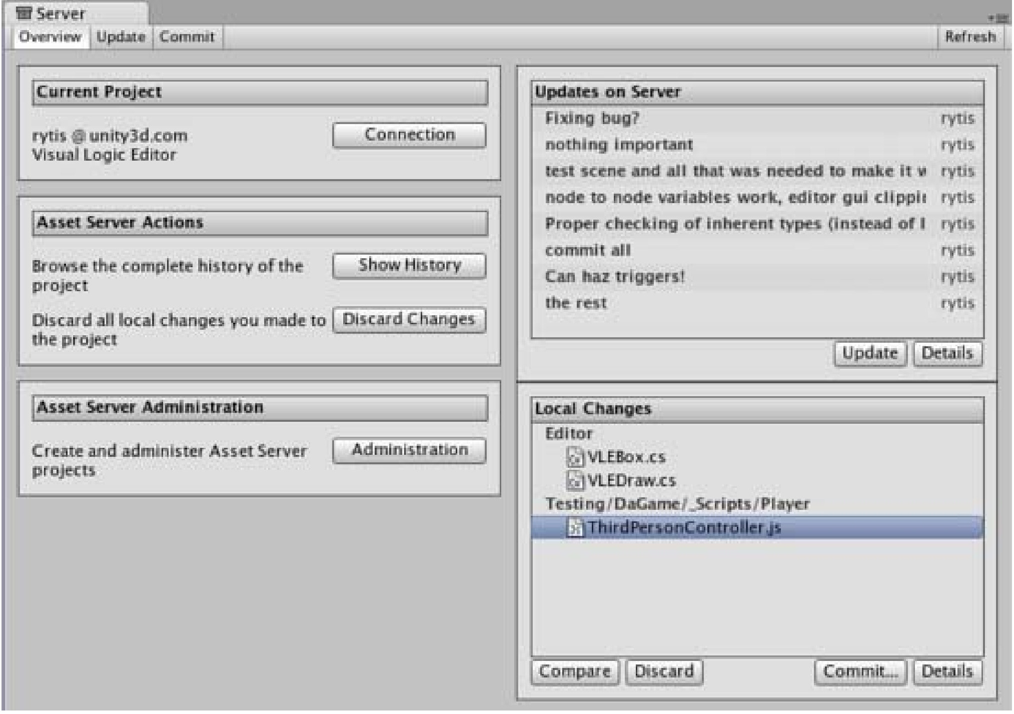
Continue reading for detailed information on how to use the Asset Server effectively every day.

### Workflow Fundamentals

When using the Asset Server with a multi-person team, it is generally good practice to Update all changed assets from the server when you begin working, and Commit your changes at the end of the day, or whenever you're done working. You should also commit changes when you have made significant progress on something, even if it is in the middle of the day. Committing your changes regularly and frequently is recommended.

### Understanding the Server View

The **Server View** is your window into the Asset Server you're connected to. You can open the Server View by selecting **Window->Version Control**.




*The **Overview** tab*

The Server View is broken into tabs: **Overview Update**, and **Commit**. **Overview** will show you any differences between your local project and the latest version on the server with options to quickly commit local changes or download the latest updates. **Update** will show you the latest remote changes on the server and allow you to download them to your local project. **Commit** allows you to create a **Changeset** and commit it to the server for others to download.

#### Connecting to the server

Before you can use the asset server, you must connect to it. To do this you click the **Connection** button, which takes you to the connection screen:

*The Asset Server connection screen*

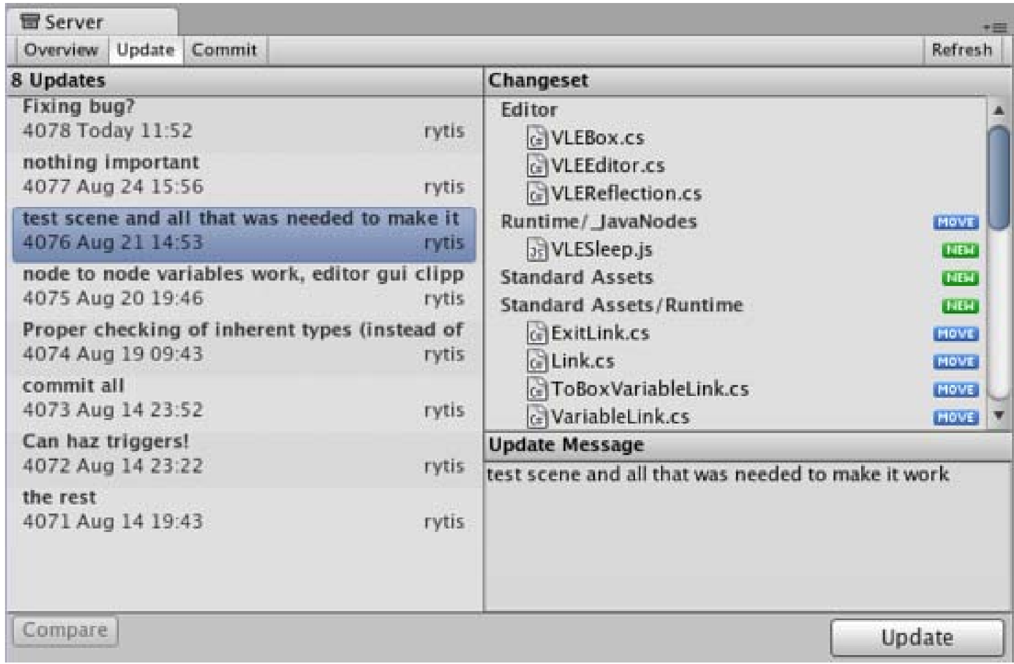
Here you need to fill in:

1. Server address
2. Username
3. Password

By clicking **Show projects** you can now see the available projects on the asset server, and choose which one to connect to by clicking **Connect**. Note that the username and password you use can be obtain from your system administrator. Your system administrator created accounts when they installed Asset Server.

### Updating from the Server

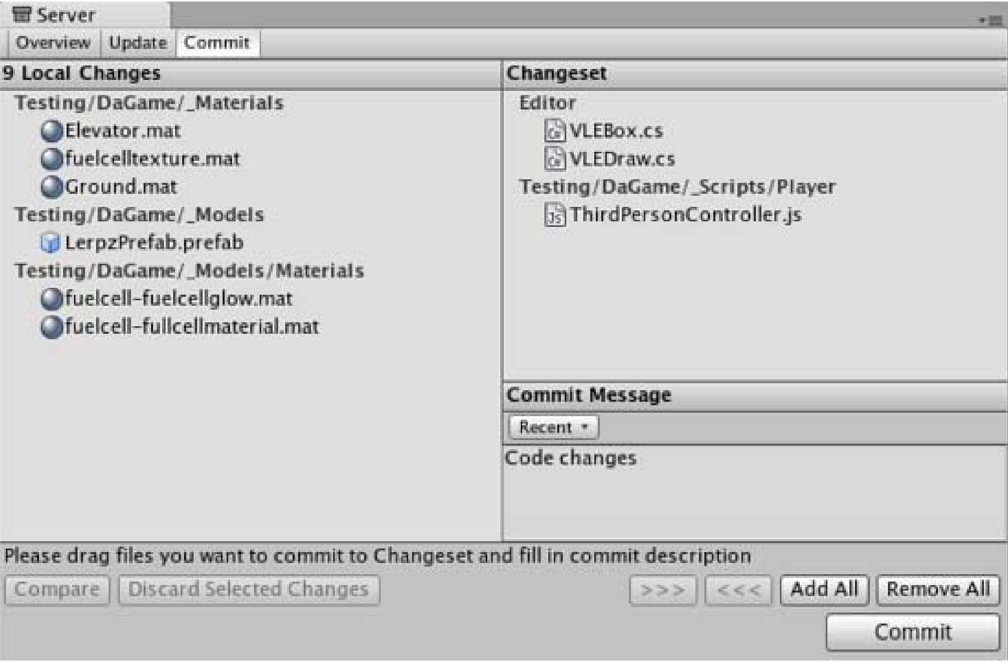
To download all updates from the server, select the **Update** tab from the Overview tab and you will see a list of the latest committed Changesets. By selecting one of these you can see what was changed in the project as well as the provided commit message. Click **Update** and you will begin downloading all Changeset updates.




*The **Update** Tab*

### Committing Changes to the Server

When you have made a change to your local project and you want to store those changes on the server, you use the top **Commit** tab.

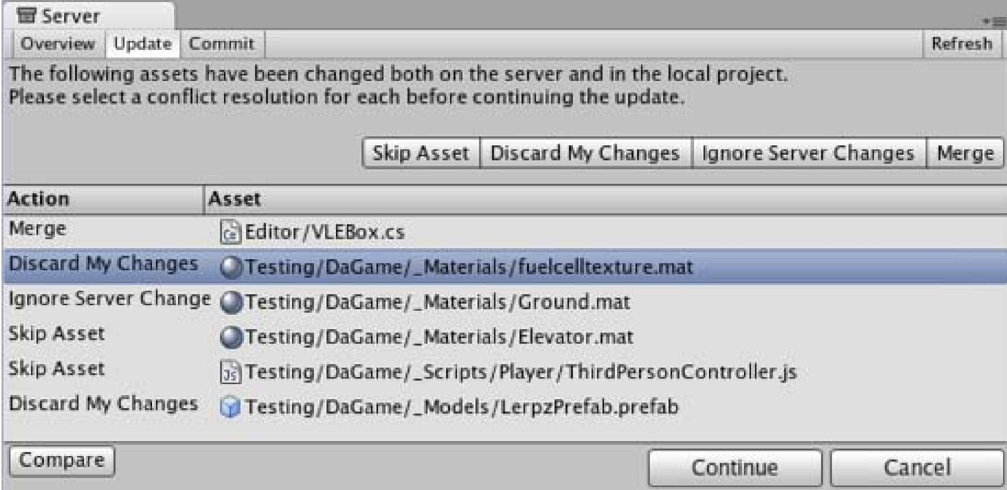



*The **Commit** tab*

Now you will be able to see all the local changes made to the project since your last update, and will be able to select which changes you wish to upload to the server. You can add changes to the changeset either by manually dragging them into the changeset field, or by using the buttons placed below the commit message field. Remember to type in a commit message which will help you when you compare versions or revert to an earlier version later on, both of which are discussed below.

**Resolving conflicts**

With multiple people working on the same collection of data, conflicts will inevitably arise. Remember, there is no need to panic! If a conflict exists, you will be presented with the **Conflict Resolution** dialog when updating your project.




*The **Conflict Resolution** screen*

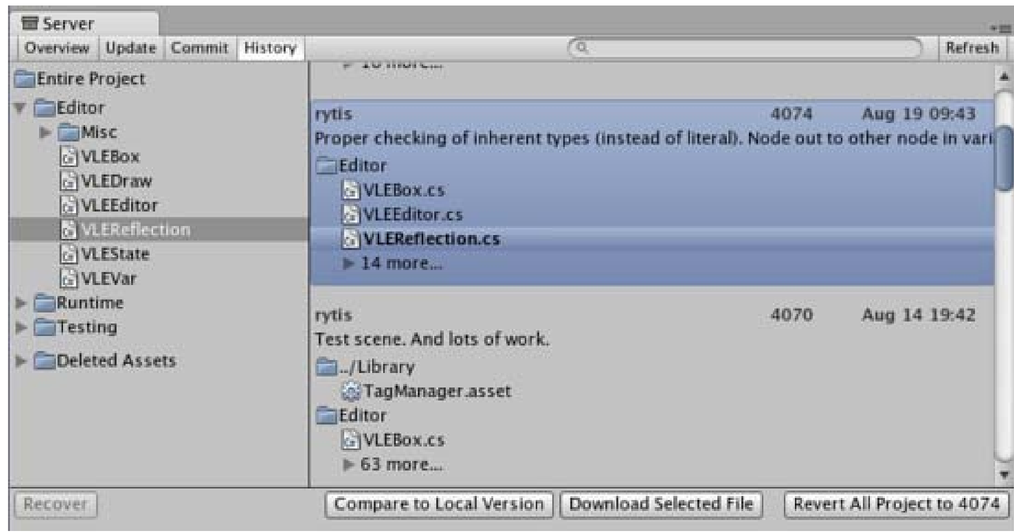
Here, you will be informed of each individual conflict, and be presented with different options to resolve each individual conflict. For any single conflict, you can select **Skip Asset** (which will not download that asset from the server), **Discard My Changes** (which will completely overwrite your local version of the asset) or **Ignore Server Changes** (which will ignore the changes others made to the asset and after this update you will be able to commit your local changes over server ones) for each individual conflict. Additionally, you can select **Merge** for text assets like scripts to merge the server version with the local version.

**Note:** If you choose to discard your changes, the asset will be updated to the latest version from the server (i.e., it will incorporate other users' changes that have been made while you were working). If you want to get the asset back as it was when you started working, you should revert to the specific version that you checked out. (See *Browsing revision history and reverting assets* below.)

If you run into a conflict while you are committing your local changes, Unity will refuse to commit your changes and inform you that a conflict exists. To resolve the conflicts, select **Update**. Your local changes will not automatically be overwritten. At this point you will see the **Conflict Resolution** dialog, and can follow the instructions in the above paragraph.

**Browsing revision history and reverting assets**

The Asset Server retains all uploaded versions of an asset in its database, so you can revert your local version to an earlier version at any time. You can either select to restore the entire project or single files. To revert to an older version of an asset or a project, select the Overview tab then click **Show History** listed under Asset Server Actions. You will now see a list of all commits and be able to select and restore any file or all project to an older version.




*The **History** dialog*

Here, you can see the version number and added comments with each version of the asset or project. This is one reason why descriptive comments are helpful. Select any asset to see its history or **Entire Project** for all changes made in project. Find revision you need. You can either select whole revision or particular asset in revision. Then click **Download Selected File** to get your local asset replaced with a copy of the selected revision. **Revert All Project** will revert entire project to selected revision.

Prior to reverting, if there are any differences between your local version and the selected server version, those changes will be lost when the local version is reverted.

If you only want to abandon the changes made to the local copy, you don't have to revert. You can discard those local modifications by selecting **Discard Changes** in the main asset server window. This will immediately download the current version of the project from the server to your local Project.

**Comparing asset versions**

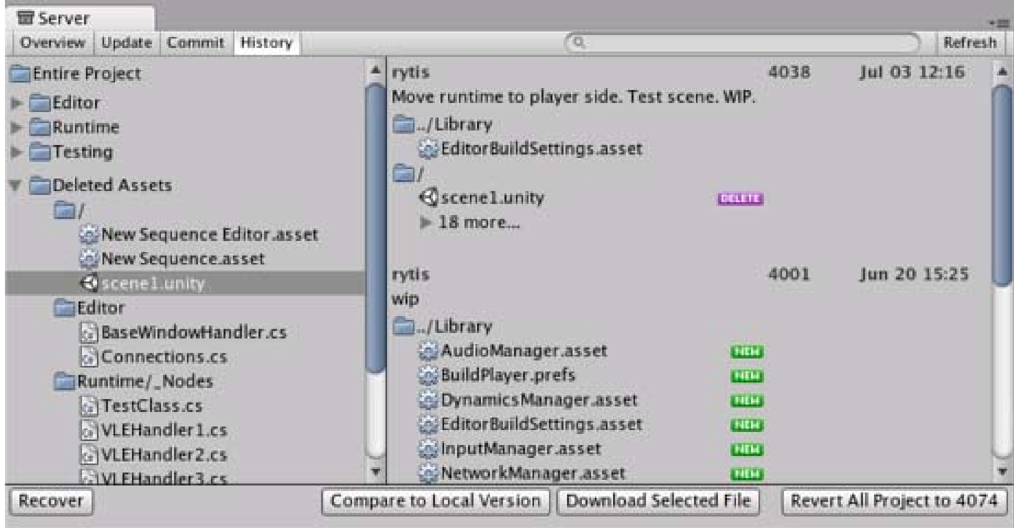
If you're curious to see the differences between two particular versions you can explicitly compare them. To do this, open **History** window, select revision and asset you want to compare and press **Compare to Local Version**. If you need to compare two different revisions of an asset - right click on it, in the context menu select **Compare to Another Revision** then find revision you want to compare to and select it.

*Note*: this feature requires that you have one of supported file diff/merge tools installed. Supported tools are:

- On Windows:
  - TortoiseMerge: part of TortoiseSVN or a separate download from the project site.
  - WinMerge.
  - SourceGear Diff/Merge.
  - Perforce Merge (p4merge): part of Perforce's visual client suite (P4V).
  - TkDiff.
- On Mac OS X:
  - SourceGear Diff/Merge.
  - FileMerge: part of Apple's XCode development tools.
  - TkDiff.
  - Perforce Merge (p4merge): part of Perforce's visual client suite (P4V).

**Recovering deleted assets**

Deleting a local asset and committing the delete to the server will in fact not delete an asset permanently. Just as any previous version of an asset can be restored through **History** window from the Overview tab.




*The **History** dialog*

Expand **Deleted Assets** item, find and select assets from the list and hit **Recover**, the selected assets will be downloaded and re-added to the local project. If the folder that the asset was located in before the deletion still exists, the asset will be restored to the original location, otherwise it will be added to the root of the Assets folder in the local project.

**Best Practices & Common Issues**

This is a compilation of best practices and solutions to problems which will help you when using the Asset Server:

1. Backup, Backup, Backup
   - Maintain a backup of your database. It is very important to do this. In the unfortunate case that you have a hardware problem, a virus, a user error, etc you may loose all of your work. Therefore make sure you have a backup system in place. You can find lots of resources online for setting up backup systems.
2. Stop the server before shutting the machine down
   - This can prevent "fast shutdowns" from being generated in the PostgreSQL (Asset Server) log. If this occurs the Asset Server has to do a recovery due to an improper shut down. This can take a very long time if you have a large project with many commits.
3. Resetting you password from Console
   - You can reset your password directly from a shell, console or command line using the following command:

     ```
     psql -U unitysrv -d template1 -c"alter role admin with password 'MYPASSWORD'"
     ```
4. Can't connect to Asset Server
   - The password may have expired. Try resetting your password.
   - Also the username is case sensitive: "Admin" != "admin". Make sure you are using the correct case.
   - Make sure the server is actually running:
     - On OS X or Linux you can type on the terminal: ps -aux
     - On Windows you can use the Task Manager.
   - Verify that the Asset Server is not running on more than one computer in your Network. You could be connecting to the wrong one.
5. The Asset Server doesn't work in 64-bit Linux
   - The asset server can run OK on 64-bit Linux machines if you install 32-bit versions of the required packages. You can use "dpkg -i --force-architecture" to do this.
6. Use the Asset Server logs to get more information
   - Windows:
     - \Unity\AssetServer\log
   - OS X:
     - /Library/UnityAssetServer/log

### Asset Server training complete

You should now be equipped with the knowledge you need to start using the Asset Server effectively. Get to it, and don't forget the good workflow fundamentals. Commit changes often, and don't be afraid of losing anything.

Page last updated: 2011-10-31

---

## Asset Cache Server

Unity has a completely automatic asset pipeline. Whenever a source asset like a `.psd` or an `.fbx` file is modified, Unity will detect the change and automatically reimport it. The imported data from the file is subsequently stored by Unity in its own internal format. The best parts about the asset pipeline are the "hot reloading" functionality and the guarantee that all your source assets are always in sync with what you see. This feature also comes at a cost. Any asset that is modified has to be reimported right away. When working in large teams, after getting latest from Source Control, you often have to wait for a long time to re-import all the assets modified or created by other team members. Also, switching your project platform back and forth between desktop and mobile will trigger a re-import of most assets.

The time it takes to import assets can be drastically reduced by caching the imported asset data on the **Cache Server**.

Each asset import is cached based on

- The asset file itself
- The import settings
- Asset importer version
- The current platform.

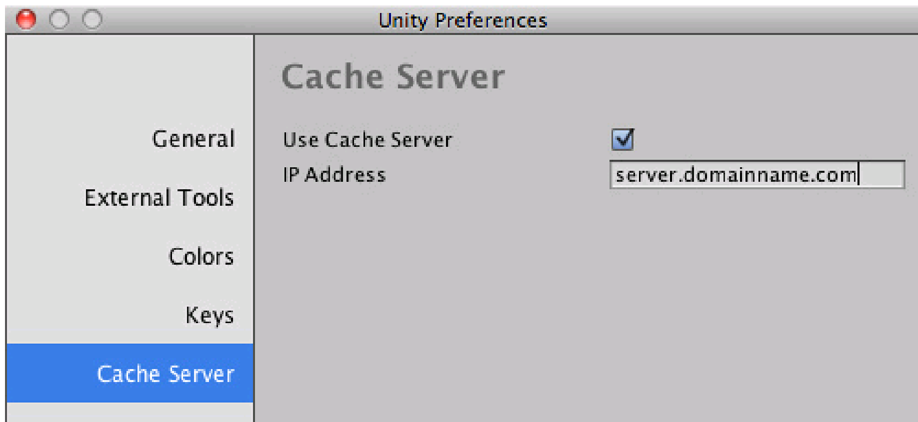
If any of the above change, the asset gets reimported, otherwise it gets downloaded from the Cache Server.

When you enable the cache server in the preferences, you can even share asset imports across multiple projects.

Note that once the cache server is set up, this process is *completely automatic*, which means there are no additional workflow requirements. It will simply reduce the time it takes to import projects without getting in your way.

### How to set up a Cache Server (user)

Setting up the Cache Server couldn't be easier. All you need to do is click Use Cache Server in the preferences and tell the local machine's **Unity Editor** where the **Cache Server** is.




This can be found in **Unity->Preferences** on the Mac or **Edit->Preferences** on the PC.

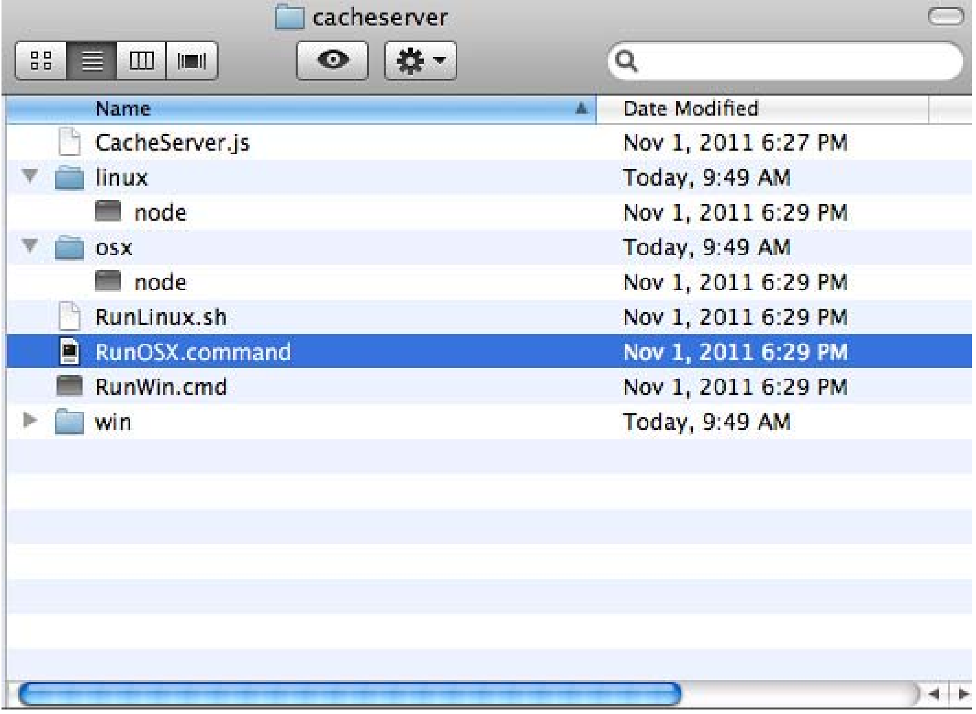
If you are hosting the Cache Server on your local machine, specify *localhost* for the server address. However, due to hard drive size limitations, it is recommended you host the Cache Server on separate machine.

### How to set up a Cache Server (admin)

Admins need to set up the **Cache Server** machine that will host the cached assets.

You need to:

- Download the Cache Server here
- Unzip the file, after which you should see something like this:




- Depending on your operating system, run the appropriate command script.
- You will see a terminal window, indicating that the Cache Server is running in the background

Terminal — node — 80×24

```
Last login: Mon Nov 14 09:21:37 on console
/Users/aleks/Downloads/cacheserver/RunOSX.command ; exit;
dyn154:~ aleks$ /Users/aleks/Downloads/cacheserver/RunOSX.command ; exit;
cache size 0
Cache Server on port 8125
Cache Server is ready
```

The **Cache Server** needs to be on a reliable machine with very large storage (much larger than the size of the project itself, as there will likely be multiple versions of imported resources stored). If the hard disk becomes full the **Cache Server** could perform slowly.

### Installing the Cache Server as a service

The provided `.sh` and `.cmd` scripts should be set-up as a service on the server. The cache server can be safely killed and restarted at any time, since it uses atomic file operations.

### Cache Server Configuration

If you simply start the Cache Server by double clicking the script, it will create a "cache" directory next to the script, and keep its data in there. The cache directory is allowed to grow to up to 50 GB. You can configure the size and the location of the data using command line options, like this:

```
./RunOSX.command --path ~/mycachePath --size 2000000000
```

`--path` lets you specify a cache location, and `--size` lets you specify the maximum cache size in bytes.

### Recommendations for the machine hosting the Cache Server

We recommend equipping the machine with a lot of RAM. For best performance there is enough RAM to hold an entire imported project folder. In addition, it is best to have a machine with a fast hard drive and fast Ethernet connection. The hard drive should also have sufficient free space. On the other hand, the Cache Server has very low CPU usage.

One of the main distinctions between the Cache Server and version control is that its cached data can always be rebuilt locally. It is simply a tool for improving performance. For this reason it doesn't make sense to use a Cache Server over the Internet. If you have a distributed team, we recommend that you place a separate cache server in each location.

We recommend that you run the cache server on a Linux or Mac OS X machine. The Windows file system is not particularly well optimized for how the Asset Cache Server stores data and problems with file locking on Windows can cause issues that don't occur on Linux or Mac OS X.

Page last updated: 2012-10-26

---

# Cache Server FAQ

**Will the size of my Cache Server database grow indefinitely as more and more resources get imported and stored?**
The Cache Server removes assets that have not been used for a period of time automatically (of course if those assets are needed again, they will be re-created during next usage).

**Does the cache server work only with the asset server?**
The cache server is designed to be transparent to source/version control systems and so you are not restricted to using Unity's asset server.

**What changes will cause the imported file to get regenerated?**
When Unity is about to import an asset, it generates an MD5 hash of all source data.

For a texture this consists of:

- The source asset: "myTexture.psd" file
- The meta file: "myTexture.psd.meta" (Stores all importer settings)
- The internal version number of the texture importer
- A hash of version numbers of all AssetPostprocessors

If that hash is different from what is stored on the Cache Server, the asset will be reimported, otherwise the cached version will be downloaded. The client Unity editor will only pull assets from the server as they are needed - assets don't get pushed to each project as they change.

**How do I work with Asset dependencies?**
The Cache Server does not handle dependencies. Unity's asset pipeline does not deal with the concept of dependencies. It is built in such a way as to avoid dependencies between assets. **AssetPostprocessors** are a common technique used to customize the Asset importer to fit your needs. For example, you might want to add MeshColliders to some GameObjects in an fbx file based on their name or tag.

It is also easy to use **AssetPostprocessors** to introduce dependencies. For example you might use data from a text file next to the asset to add additional components to the imported game objects. This is not supported in the Cache Server. If you want to use the Cache Server, you will have to remove dependency on other assets in the project folder. Since the Cache Server doesn't know anything about the dependency in your postprocessor, it will not know that anything has changed thus use an old cached version of the asset.

In practice there are plenty of ways you can do asset postprocessing to work well with the cache server. You can use:

- The Path of the imported asset
- Any import settings of the asset
- The source asset itself or any data generated from it passed to you in the asset postprocessor.

**Are there any issues when working with materials?**
Modifying materials that already exist might cause trouble. When using the Cache Server, Unity validates that the references to materials are maintained. But since no postprocessing calls will be invoked, the contents of the material can not be changed when a model is imported through the Cache Server. Thus you might get different results when importing with or without Cache Server. It is best to never modify materials that already exist on disk.

**Are there any asset types which will not be cached by the server?**
There are a few kinds of asset data which the server doesn't cache. There isn't really anything to be gained by caching script files and so the server will ignore them. Also, native files used by 3D modelling software (Maya, 3D Max, etc) are converted to FBX using the application itself. Currently, the asset server caches neither the native file nor the intermediate FBX file generated in the import process. However, it is possible to benefit from the server by exporting files as FBX from the modelling software and adding those to the Unity project.

Page last updated: 2012-09-04

---

## Behind the Scenes

Unity automatically imports assets and manages various kinds of additional data about them for you. Below is a description of how this process works.

When you place an Asset such as a texture in the Assets folder, Unity will first detect that a new file has been added (the editor frequently checks the contents of the Assets folder against the list of assets it already knows about). Once a unique ID value has been assigned to the asset to enable it to be accessed internally, it will be imported and processed. The asset that you actually see in the Project panel is the result of that processing and its data contents will typically be different to those of the original asset. For example, a texture may be present in the Assets folder as a PNG file but will be converted to an internal format after import and processing.

Using an internal format for assets allows Unity to keep additional data known as *metadata* which enables the asset data to be handled in a much more flexible way. For example, the Photoshop file format is convenient to work with, but you wouldn't expect it to support game engine features such as mip maps. Unity's internal format, however, can add extra functionality like this to any asset type. All metadata for assets is stored in the **Library** folder. As as user, you should never have to alter the Library folder manually and attempting to do so may corrupt the project.

Unity allows you to create folders in the Project view to help you organize assets, and those folders will be mirrored in the actual filesystem. However, you **must** move the files within Unity by dragging and dropping in the Project view. If you attempt to use the filesystem/desktop to move the files then Unity will misinterpret the change (it will appear that the old asset has been deleted and a new one created in its place). This will lose information, such as links between assets and scripts in the project.

When backing up a project, you should **always** back up the main Unity project folder, containing both the **Assets** and **Library** folders. All the information in the subfolders is crucial to the way Unity works.

Page last updated: 2011-11-15

---

## Creating Gameplay

Unity empowers game designers to make games. What's really special about Unity is that you don't need years of experience with code or a degree in art to make fun games. There are a handful of basic workflow concepts needed to learn Unity. Once understood, you will find yourself making games in no time. With the time you will save getting your games up and running, you will have that much more time to refine, balance, and tweak your game to perfection.

This section will explain the core concepts you need to know for creating unique, amazing, and fun gameplay. The majority of these concepts require you to write **Scripts**. For an overview of creating and working with Scripts, please read the Scripting page.

- Instantiating Prefabs at runtime
- Input
- Transforms
- Physics
- Adding Random Gameplay Elements
- Particle Systems

    - Particle System Curve Editor
    - Colors and Gradients in the Particle System (Shuriken)
    - Gradient Editor
    - Particle System Inspector
    - Introduction to Particle System Modules (Shuriken)
    - Particle System Modules (Shuriken)
    - Particle Effects (Shuriken)
- Mecanim Animation System
    - A Glossary of Animation and Mecanim terms
    - Asset Preparation and Import
        - Preparing your own character
        - Importing Animations
        - Splitting Animations
    - Working with humanoid animations
        - Creating the Avatar
        - Configuring the Avatar
        - Muscle setup
        - Avatar Body Mask
        - Retargeting of Humanoid animations
        - Inverse Kinematics (Pro only)
    - Generic Animations in Mecanim
    - Bringing Characters to Life
        - Looping animation clips
        - Animator Component and Animator Controller
        - Animation State Machines
            - Animation States
            - Animation Transitions
            - Animation Parameters
        - Blend Trees
        - Mecanim Advanced topics
            - Working with Animation Curves in Mecanim (Pro only)
            - Sub-State Machines
            - Animation Layers
            - Animation State Machine Preview (solo and mute)
            - Target Matching
            - Root Motion - how it works
                - Tutorial: Scripting Root Motion for "in-place" humanoid animations
- Legacy animation system
    - Animation View Guide (Legacy)
    - Animation Scripting (Legacy)
- Navmesh and Pathfinding (Pro only)
    - Navmesh Baking
- Sound
- Game Interface Elements
- Networked Multiplayer


Page last updated: 2010-06-30

---

# Instantiating Prefabs

By this point you should understand the concept of **Prefabs** at a fundamental level. They are a collection of predefined **GameObjects** & **Components** that are re-usable throughout your game. If you don't know what a Prefab is, we recommend you read the Prefabs page for a more basic introduction.

Prefabs come in very handy when you want to instantiate complicated GameObjects at runtime. The alternative to instantiating Prefabs is to create GameObjects from scratch using code. Instantiating Prefabs has many advantages over the alternative approach:

- You can instantiate a Prefab from one line of code, with complete functionality. Creating equivalent GameObjects from

code takes an average of five lines of code, but likely more.
- You can set up, test, and modify the Prefab quickly and easily in the Scene and Inspector.
- You can change the Prefab being instanced without changing the code that instantiates it. A simple rocket might be altered into a super-charged rocket, and no code changes are required.

## Common Scenarios

To illustrate the strength of Prefabs, let's consider some basic situations where they would come in handy:

1. Building a wall out of a single "brick" Prefab by creating it several times in different positions.
2. A rocket launcher instantiates a flying rocket Prefab when fired. The Prefab contains a Mesh, **Rigidbody**, **Collider**, and a child GameObject with its own trail **Particle System**.
3. A robot exploding to many pieces. The complete, operational robot is destroyed and replaced with a wrecked robot Prefab. This Prefab would consist of the robot split into many parts, all set up with Rigidbodies and Particle Systems of their own. This technique allows you to blow up a robot into many pieces, with just one line of code, replacing one object with a Prefab.

**Building a wall**

This explanation will illustrate the advantages of using a Prefab vs creating objects from code.

First, lets build a brick wall from code:

```
// JavaScript
function Start () {
    for (var y = 0; y < 5; y++) {
        for (var x = 0; x < 5; x++) {
            var cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
            cube.AddComponent(Rigidbody);
            cube.transform.position = Vector3 (x, y, 0);
        }
    }
}


// C#
public class Instantiation : MonoBehaviour {

    void Start() {
        for (int y = 0; y < 5; y++) {
            for (int x = 0; x < 5; x++) {
                GameObject cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
                cube.AddComponent<Rigidbody>();
                cube.transform.position = new Vector3(x, y, 0);
            }
        }
    }
}
```

- To use the above script we simply save the script and drag it onto an empty GameObject.
- Create an empty GameObject with **GameObject->Create Empty**.

If you execute that code, you will see an entire brick wall is created when you enter **Play Mode**. There are two lines relevant to the functionality of each individual brick: the **CreatePrimitive()** line, and the **AddComponent()** line. Not so bad right now, but each of our bricks is un-textured. Every additional action to want to perform on the brick, like changing the texture, the friction, or the Rigidbody **mass**, is an extra line.

If you create a Prefab and perform all your setup before-hand, you use one line of code to perform the creation and setup of each brick. This relieves you from maintaining and changing a lot of code when you decide you want to make changes. With a Prefab, you just make your changes and Play. No code alterations required.

If you're using a Prefab for each individual brick, this is the code you need to create the wall.

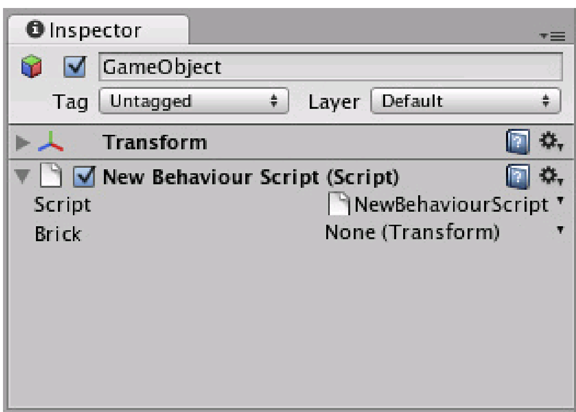
```
// JavaScript

var brick : Transform;
function Start () {
   for (var y = 0; y < 5; y++) {
      for (var x = 0; x < 5; x++) {
         Instantiate(brick, Vector3 (x, y, 0), Quaternion.identity);
      }
   }
}



// C#
public Transform brick;

void Start() {
      for (int y = 0; y < 5; y++) {
            for (int x = 0; x < 5; x++) {
                  Instantiate(brick, new Vector3(x, y, 0), Quaternion.identity);
            }
      }
}
```

This is not only very clean but also very reusable. There is nothing saying we are instantiating a cube or that it must contain a rigidbody. All of this is defined in the Prefab and can be quickly created in the Editor.

Now we only need to create the Prefab, which we do in the Editor. Here's how:

1. Choose **GameObject->Create Other->Cube**
2. Choose **Component->Physics->Rigidbody**
3. Choose **Assets->Create->Prefab**
4. In the **Project View**, change the name of your new Prefab to "Brick"
5. Drag the cube you created in the **Hierarchy** onto the "Brick" Prefab in the **Project View**
6. With the Prefab created, you can safely delete the Cube from the Hierarchy (**Delete** on Windows, **Command-Backspace** on Mac)

We've created our Brick Prefab, so now we have to attach it to the **brick** variable in our script. Select the empty GameObject that contains the script. Notice that a new variable has appeared in the Inspector, called "brick".




*This variable can accept any GameObject or Prefab*

Now drag the "Brick" Prefab from the Project View onto the **brick** variable in the Inspector. Press Play and you'll see the wall built using the Prefab.

This is a workflow pattern that can be used over and over again in Unity. In the beginning you might wonder why this is so much better, because the script creating the cube from code is only 2 lines longer.

But because you are using a Prefab now, you can adjust the Prefab in seconds. Want to change the mass of all those

instances? Adjust the Rigidbody in the Prefab only once. Want to use a different **Material** for all the instances? Drag the Material onto the Prefab only once. Want to change friction? Use a different **Physic Material** in the Prefab's collider. Want to add a Particle System to all those boxes? Add a child to the Prefab only once.

**Instantiating rockets & explosions**

Here's how Prefabs fit into this scenario:

1. A rocket launcher instantiates a rocket Prefab when the user presses fire. The Prefab contains a mesh, Rigidbody, Collider, and a child GameObject that contains a trail particle system.
2. The rocket impacts and instantiates an explosion Prefab. The explosion Prefab contains a Particle System, a light that fades out over time, and a script that applies damage to surrounding GameObjects.

While it would be possible to build a rocket GameObject completely from code, adding Components manually and setting properties, it is far easier to instantiate a Prefab. You can instantiate the rocket in just one line of code, no matter how complex the rocket's Prefab is. After instantiating the Prefab you can also modify any properties of the instantiated object (e.g. you can set the velocity of the rocket's Rigidbody).

Aside from being easier to use, you can update the prefab later on. So if you are building a rocket, you don't immediately have to add a Particle trail to it. You can do that later. As soon as you add the trail as a child GameObject to the Prefab, all your instantiated rockets will have particle trails. And lastly, you can quickly tweak the properties of the rocket Prefab in the Inspector, making it far easier to fine-tune your game.

This script shows how to launch a rocket using the **Instantiate()** function.

```
// JavaScript

// Require the rocket to be a rigidbody.
// This way we the user can not assign a prefab without rigidbody
var rocket : Rigidbody;
var speed = 10.0;

function FireRocket () {
    var rocketClone : Rigidbody = Instantiate(rocket, transform.position, transform.rotation);
    rocketClone.velocity = transform.forward * speed;
    // You can also acccess other components / scripts of the clone
    rocketClone.GetComponent(MyRocketScript).DoSomething();
}

// Calls the fire method when holding down ctrl or mouse
function Update () {
    if (Input.GetButtonDown("Fire1")) {
        FireRocket();
    }
}


// C#

// Require the rocket to be a rigidbody.
// This way we the user can not assign a prefab without rigidbody
public Rigidbody rocket;
public float speed = 10f;

void FireRocket () {
        Rigidbody rocketClone = (Rigidbody) Instantiate(rocket, transform.position, transform.rotation);
        rocketClone.velocity = transform.forward * speed;

        // You can also acccess other components / scripts of the clone
        rocketClone.GetComponent<MyRocketScript>().DoSomething();
}
```

```
// Calls the fire method when holding down ctrl or mouse
void Update () {
	if (Input.GetButtonDown("Fire1")) {
		FireRocket();
	}
}
```

**Replacing a character with a ragdoll or wreck**

Let's say you have a fully rigged enemy character and he dies. You could simply play a death animation on the character and disable all scripts that usually handle the enemy logic. You probably have to take care of removing several scripts, adding some custom logic to make sure that no one will continue attacking the dead enemy anymore, and other cleanup tasks.

A far better approach is to immediately delete the entire character and replace it with an instantiated wrecked prefab. This gives you a lot of flexibility. You could use a different material for the dead character, attach completely different scripts, spawn a Prefab containing the object broken into many pieces to simulate a shattered enemy, or simply instantiate a Prefab containing a version of the character.

Any of these options can be achieved with a single call to **Instantiate()**, you just have to hook it up to the right prefab and you're set!

The important part to remember is that the wreck which you **Instantiate()** can be made of completely different objects than the original. For example, if you have an airplane, you would model two versions. One where the plane consists of a single GameObject with **Mesh Renderer** and scripts for airplane physics. By keeping the model in just one GameObject, your game will run faster since you will be able to make the model with less triangles and since it consists of fewer objects it will render faster than using many small parts. Also while your plane is happily flying around there is no reason to have it in separate parts.

To build a wrecked airplane Prefab, the typical steps are:

1. Model your airplane with lots of different parts in your favorite modeler
2. Create an empty Scene
3. Drag the model into the empty Scene
4. Add Rigidbodies to all parts, by selecting all the parts and choosing **Component->Physics->Rigidbody**
5. Add Box Colliders to all parts by selecting all the parts and choosing **Component->Physics->Box Collider**
6. For an extra special effect, add a smoke-like Particle System as a child GameObject to each of the parts
7. Now you have an airplane with multiple exploded parts, they fall to the ground by physics and will create a Particle trail due to the attached particle system. Hit Play to preview how your model reacts and do any necessary tweaks.
8. Choose **Assets->Create Prefab**
9. Drag the root GameObject containing all the airplane parts into the Prefab

```
// JavaScript

var wreck : GameObject;

// As an example, we turn the game object into a wreck after 3 seconds automatically
function Start () {
    yield WaitForSeconds(3);
    KillSelf();
}

// Calls the fire method when holding down ctrl or mouse
function KillSelf () {
    // Instantiate the wreck game object at the same position we are at
    var wreckClone = Instantiate(wreck, transform.position, transform.rotation);

    // Sometimes we need to carry over some variables from this object
    // to the wreck
    wreckClone.GetComponent(MyScript).someVariable = GetComponent(MyScript).someVariable;
```

```
    // Kill ourselves
    Destroy(gameObject);


// C#

public GameObject wreck;

// As an example, we turn the game object into a wreck after 3 seconds automatically
IEnumerator Start() {
        yield return new WaitForSeconds(3);
        KillSelf();
}

// Calls the fire method when holding down ctrl or mouse
void KillSelf () {
        // Instantiate the wreck game object at the same position we are at
        GameObject wreckClone = (GameObject) Instantiate(wreck, transform.position, transform.rotation);

        // Sometimes we need to carry over some variables from this object
        // to the wreck
        wreckClone.GetComponent<MyScript>().someVariable = GetComponent<MyScript>().someVariable;

        // Kill ourselves
        Destroy(gameObject);
}

}
```

The First Person Shooter tutorial explains how to replace a character with a ragdoll version and also synchronize limbs with the last state of the animation. You can find that tutorial on the Tutorials page.

**Placing a bunch of objects in a specific pattern**

Lets say you want to place a bunch of objects in a grid or circle pattern. Traditionally this would be done by either:

1. Building an object completely from code. This is tedious! Entering values from a script is both slow, unintuitive and not worth the hassle.
2. Make the fully rigged object, duplicate it and place it multiple times in the scene. This is tedious, and placing objects accurately in a grid is hard.

So use **Instantiate()** with a Prefab instead! We think you get the idea of why Prefabs are so useful in these scenarios. Here's the code necessary for these scenarios:

```
// JavaScript

// Instantiates a prefab in a circle

var prefab : GameObject;
var numberOfObjects = 20;
var radius = 5;

function Start () {
    for (var i = 0; i < numberOfObjects; i++) {
        var angle = i * Mathf.PI * 2 / numberOfObjects;
        var pos = Vector3 (Mathf.Cos(angle), 0, Mathf.Sin(angle)) * radius;
        Instantiate(prefab, pos, Quaternion.identity);
    }
}
```

```
// C#
// Instantiates a prefab in a circle

public GameObject prefab;
public int numberOfObjects = 20;
public float radius = 5f;

void Start() {
	for (int i = 0; i < numberOfObjects; i++) {
		float angle = i * Mathf.PI * 2 / numberOfObjects;
		Vector3 pos = new Vector3(Mathf.Cos(angle), 0, Mathf.Sin(angle)) * radius;
		Instantiate(prefab, pos, Quaternion.identity);
	}
}
```

```
// JavaScript

// Instantiates a prefab in a grid

var prefab : GameObject;
var gridX = 5;
var gridY = 5;
var spacing = 2.0;

function Start () {
    for (var y = 0; y < gridY; y++) {
        for (var x=0;x<gridX;x++) {
            var pos = Vector3 (x, 0, y) * spacing;
            Instantiate(prefab, pos, Quaternion.identity);
        }
    }
}


// C#

// Instantiates a prefab in a grid

public GameObject prefab;
public float gridX = 5f;
public float gridY = 5f;
public float spacing = 2f;

void Start() {
	for (int y = 0; y < gridY; y++) {
		for (int x = 0; x < gridX; x++) {
			Vector3 pos = new Vector3(x, 0, y) * spacing;
			Instantiate(prefab, pos, Quaternion.identity);
		}
	}
}
```

Page last updated: 2012-10-09

---

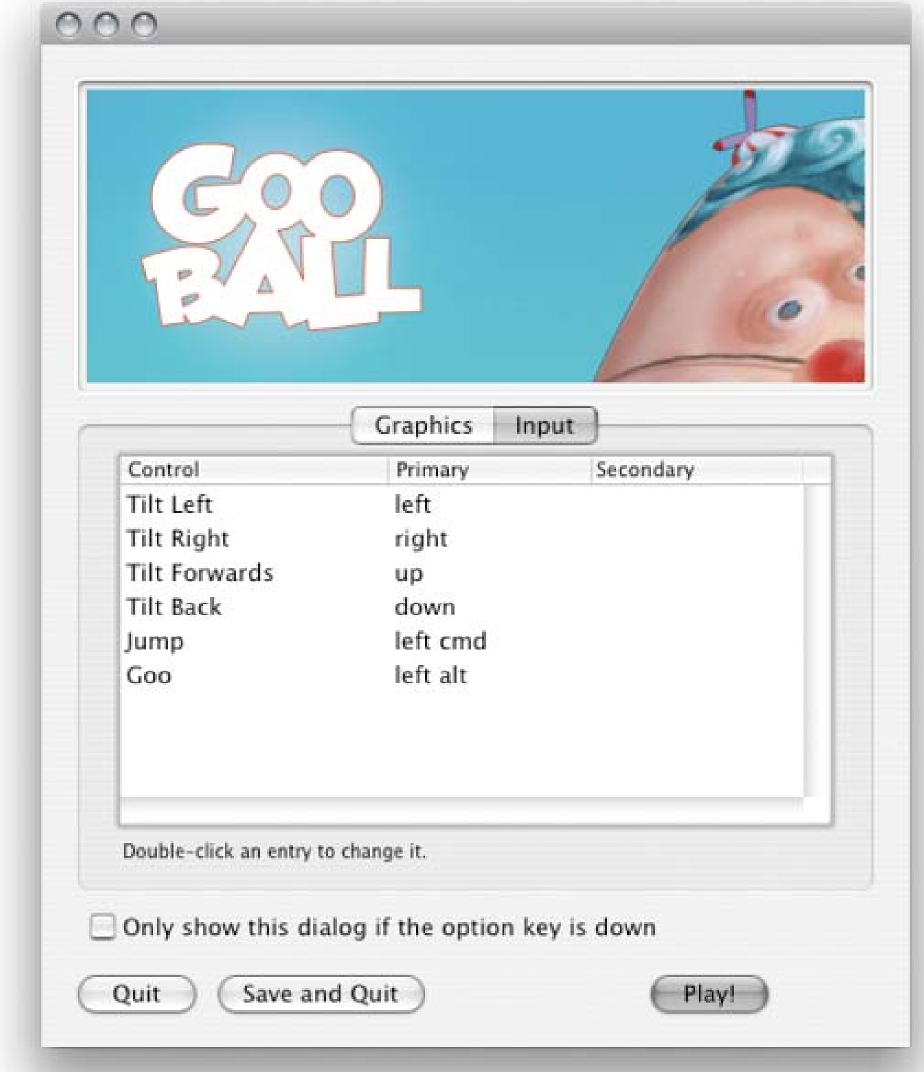
# Input

## Desktop

**Note:** Keyboard, joystick and gamepad input work on the desktop versions of Unity (including webplayer and Flash) but not on mobiles.

Unity supports keyboard, joystick and gamepad input.

Virtual axes and buttons can be created in the **Input Manager**, and end users can configure Keyboard input in a nice screen configuration dialog.




You can setup joysticks, gamepads, keyboard, and mouse, then access them all through one simple scripting interface.

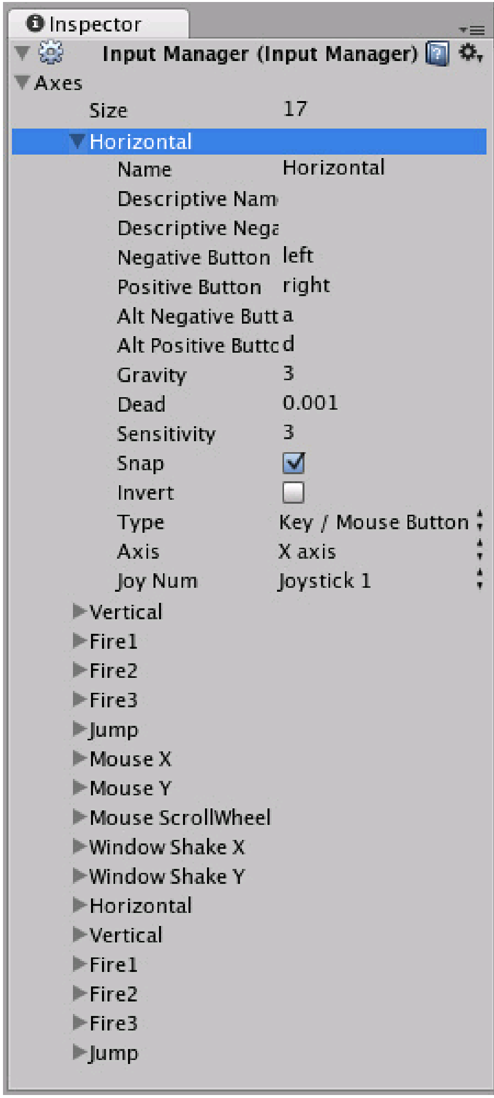
From scripts, all virtual axes are accessed by their name.

Every project has the following default input axes when it's created:

- **Horizontal** and **Vertical** are mapped to w, a, s, d and the arrow keys.
- **Fire1**, **Fire2**, **Fire3** are mapped to Control, Option (Alt), and Command, respectively.
- **Mouse X** and **Mouse Y** are mapped to the delta of mouse movement.
- **Window Shake X** and **Window Shake Y** is mapped to the movement of the window.

**Adding new Input Axes**

If you want to add new virtual axes go to the **Edit->Project Settings->Input** menu. Here you can also change the settings of each axis.




You map each axis to two buttons on a joystick, mouse, or keyboard keys.

| | |
|---|---|
| **Name** | The name of the string used to check this axis from a script. |
| **Descriptive Name** | Positive value name displayed in the input tab of the **Configuration** dialog for standalone builds. |
| **Descriptive Negative Name** | Negative value name displayed in the Input tab of the **Configuration** dialog for standalone builds. |
| **Negative Button** | The button used to push the axis in the negative direction. |
| **Positive Button** | The button used to push the axis in the positive direction. |
| **Alt Negative Button** | Alternative button used to push the axis in the negative direction. |
| **Alt Positive Button** | Alternative button used to push the axis in the positive direction. |
| **Gravity** | Speed in units per second that the axis falls toward neutral when no buttons are pressed. |
| **Dead** | Size of the analog dead zone. All analog device values within this range result map to neutral. |
| **Sensitivity** | Speed in units per second that the the axis will move toward the target value. This is for digital devices only. |
| **Snap** | If enabled, the axis value will reset to zero when pressing a button of the opposite direction. |
| **Invert** | If enabled, the **Negative Buttons** provide a positive value, and vice-versa. |
| **Type** | The type of inputs that will control this axis. |
| **Axis** | The axis of a connected device that will control this axis. |
| **Joy Num** | The connected Joystick that will control this axis. |

Use these settings to fine tune the look and feel of input. They are all documented with tooltips in the Editor as well.

**Using Input Axes from Scripts**

You can query the current state from a script like this:

```
value = Input.GetAxis ("Horizontal");
```

An axis has a value between -1 and 1. The neutral position is 0. This is the case for joystick input and keyboard input.

However, Mouse Delta and Window Shake Delta are how much the mouse or window moved during the last frame. This means it can be larger than 1 or smaller than -1 when the user moves the mouse quickly.

It is possible to create multiple axes with the same name. When getting the input axis, the axis with the largest absolute value will be returned. This makes it possible to assign more than one input device to one axis name. For example, create one axis for keyboard input and one axis for joystick input with the same name. If the user is using the joystick, input will come from the joystick, otherwise input will come from the keyboard. This way you don't have to consider where the input comes from when writing scripts.

**Button Names**
To map a key to an axis, you have to enter the key's name in the **Positive Button** or **Negative Button** property in the **Inspector**.

The names of keys follow this convention:

- Normal keys: "a", "b", "c" ...
- Number keys: "1", "2", "3", ...
- Arrow keys: "up", "down", "left", "right"
- Keypad keys: "[1]", "[2]", "[3]", "[+]", "[equals]"
- Modifier keys: "right shift", "left shift", "right ctrl", "left ctrl", "right alt", "left alt", "right cmd", "left cmd"
- Mouse Buttons: "mouse 0", "mouse 1", "mouse 2", ...
- Joystick Buttons (from any joystick): "joystick button 0", "joystick button 1", "joystick button 2", ...
- Joystick Buttons (from a specific joystick): "joystick 1 button 0", "joystick 1 button 1", "joystick 2 button 0", ...
- Special keys: "backspace", "tab", "return", "escape", "space", "delete", "enter", "insert", "home", "end", "page up", "page down"
- Function keys: "f1", "f2", "f3", ...

The names used to identify the keys are the same in the scripting interface and the Inspector.

```
value = Input.GetKey ("a");
```

## Mobile Input

On iOS and Android, the Input class offers access to touchscreen, accelerometer and geographical/location input.

Access to keyboard on mobile devices is provided via the iOS keyboard.

### Multi-Touch Screen

The iPhone and iPod Touch devices are capable of tracking up to five fingers touching the screen simultaneously. You can retrieve the status of each finger touching the screen during the last frame by accessing the Input.touches property array.

Android devices don't have a unified limit on how many fingers they track. Instead, it varies from device to device and can be anything from two-touch on older devices to five fingers on some newer devices.

Each finger touch is represented by an Input.Touch data structure:

| | |
|---|---|
| **fingerId** | The unique index for a touch. |
| **position** | The screen position of the touch. |
| **deltaPosition** | The screen position change since the last frame. |
| **deltaTime** | Amount of time that has passed since the last state change. |
| **tapCount** | The iPhone/iPad screen is able to distinguish quick finger taps by the user. This counter will let you know how many times the user has tapped the screen without moving a finger to the sides. Android devices do not count number of taps, this field is always 1. |

| | |
|---|---|
| **phase** | Describes so called "phase" or the state of the touch. It can help you determine if the touch just began, if user moved the finger or if he just lifted the finger. |

Phase can be one of the following:

| | |
|---|---|
| **Began** | A finger just touched the screen. |
| **Moved** | A finger moved on the screen. |
| **Stationary** | A finger is touching the screen but hasn't moved since the last frame. |
| **Ended** | A finger was lifted from the screen. This is the final phase of a touch. |
| **Canceled** | The system cancelled tracking for the touch, as when (for example) the user puts the device to her face or more than five touches happened simultaneously. This is the final phase of a touch. |

Following is an example script which will shoot a ray whenever the user taps on the screen:

```
var particle : GameObject;
function Update () {
	for (var touch : Touch in Input.touches) {
		if (touch.phase == TouchPhase.Began) {
			// Construct a ray from the current touch coordinates
			var ray = Camera.main.ScreenPointToRay (touch.position);
			if (Physics.Raycast (ray)) {
				// Create a particle if hit
				Instantiate (particle, transform.position, transform.rotation);
			}
		}
	}
}
```

**Mouse Simulation**
On top of native touch support Unity iOS/Android provides a mouse simulation. You can use mouse functionality from the standard Input class.

## Device Orientation

Unity iOS/Android allows you to get discrete description of the device physical orientation in three-dimensional space. Detecting a change in orientation can be useful if you want to create game behaviors depending on how the user is holding the device.

You can retrieve device orientation by accessing the Input.deviceOrientation property. Orientation can be one of the following:

| | |
|---|---|
| **Unknown** | The orientation of the device cannot be determined. For example when device is rotate diagonally. |
| **Portrait** | The device is in portrait mode, with the device held upright and the home button at the bottom. |
| **PortraitUpsideDown** | The device is in portrait mode but upside down, with the device held upright and the home button at the top. |
| **LandscapeLeft** | The device is in landscape mode, with the device held upright and the home button on the right side. |
| **LandscapeRight** | The device is in landscape mode, with the device held upright and the home button on the left side. |
| **FaceUp** | The device is held parallel to the ground with the screen facing upwards. |
| **FaceDown** | The device is held parallel to the ground with the screen facing downwards. |

## Accelerometer

As the mobile device moves, a built-in accelerometer reports linear acceleration changes along the three primary axes in three-dimensional space. Acceleration along each axis is reported directly by the hardware as G-force values. A value of 1.0 represents a load of about +1g along a given axis while a value of -1.0 represents -1g. If you hold the device upright (with the home button at the bottom) in front of you, the X axis is positive along the right, the Y axis is positive directly up, and the Z axis is positive pointing toward you.

You can retrieve the accelerometer value by accessing the Input.acceleration property.

The following is an example script which will move an object using the accelerometer:

```
var speed = 10.0;
```

```
function Update () {
      var dir : Vector3 = Vector3.zero;

      // we assume that the device is held parallel to the ground
      // and the Home button is in the right hand

      // remap the device acceleration axis to game coordinates:
      //  1) XY plane of the device is mapped onto XZ plane
      //  2) rotated 90 degrees around Y axis
      dir.x = -Input.acceleration.y;
      dir.z = Input.acceleration.x;

      // clamp acceleration vector to the unit sphere
      if (dir.sqrMagnitude > 1)
            dir.Normalize();

      // Make it move 10 meters per second instead of 10 meters per frame...
      dir *= Time.deltaTime;

      // Move object
      transform.Translate (dir * speed);
}
```

**Low-Pass Filter**

Accelerometer readings can be jerky and noisy. Applying low-pass filtering on the signal allows you to smooth it and get rid of high frequency noise.

The following script shows you how to apply low-pass filtering to accelerometer readings:

```
var AccelerometerUpdateInterval : float = 1.0 / 60.0;
var LowPassKernelWidthInSeconds : float = 1.0;

private var LowPassFilterFactor : float = AccelerometerUpdateInterval / LowPassKernelWidthInSeconds; // tweakable
private var lowPassValue : Vector3 = Vector3.zero;
function Start () {
      lowPassValue = Input.acceleration;
}

function LowPassFilterAccelerometer() : Vector3 {
      lowPassValue = Mathf.Lerp(lowPassValue, Input.acceleration, LowPassFilterFactor);
      return lowPassValue;
}
```

The greater the value of `LowPassKernelWidthInSeconds`, the slower the filtered value will converge towards the current input sample (and vice versa). You should be able to use the `LowPassFilter()` function instead of `avgSamples()`.

**I'd like as much precision as possible when reading the accelerometer. What should I do?**

Reading the Input.acceleration variable does not equal sampling the hardware. Put simply, Unity samples the hardware at a frequency of 60Hz and stores the result into the variable. In reality, things are a little bit more complicated -- accelerometer sampling doesn't occur at consistent time intervals, if under significant CPU loads. As a result, the system might report 2 samples during one frame, then 1 sample during the next frame.

You can access all measurements executed by accelerometer during the frame. The following code will illustrate a simple average of all the accelerometer events that were collected within the last frame:

```
var period : float = 0.0;
var acc : Vector3 = Vector3.zero;
for (var evnt : iPhoneAccelerationEvent  in iPhoneInput.accelerationEvents) {
```

```
        acc += evnt.acceleration * evnt.deltaTime;
        period += evnt.deltaTime;
}
if (period > 0)
        acc *= 1.0/period;
return acc;
```

### Further Reading

The Unity mobile input API is originally based on Apple's API. It may help to learn more about the native API to better understand Unity's Input API. You can find the Apple input API documentation here:

- Programming Guide: Event Handling (Apple iPhone SDK documentation)
- UITouch Class Reference (Apple iOS SDK documentation)

**Note:** The above links reference your locally installed iPhone SDK Reference Documentation and will contain native ObjectiveC code. It is not necessary to understand these documents for using Unity on mobile devices, but may be helpful to some!

## iOS

### Device geographical location

Device geographical location can be obtained via the iPhoneInput.lastLocation property. Before calling this property you should start location service updates using iPhoneSettings.StartLocationServiceUpdates() and check the service status via iPhoneSettings.locationServiceStatus. See the scripting reference for details.

Page last updated: 2012-06-28

---

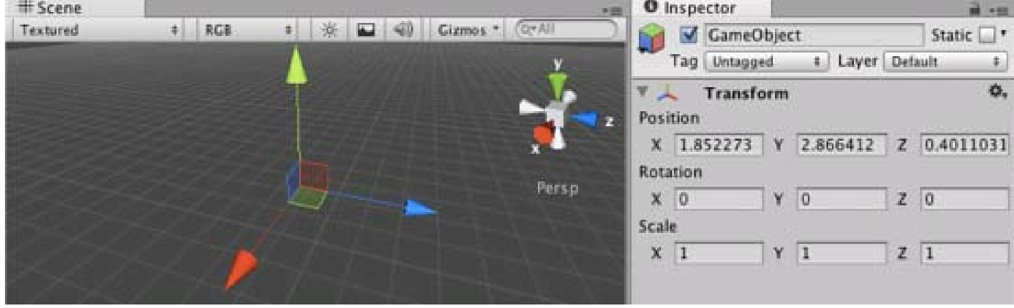
# Transforms

**Transforms** are a key **Component** in every **GameObject**. They dictate where the GameObject is positioned, how it is rotated, and its scale. It is impossible to have a GameObject without a Transform. You can adjust the Transform of any GameObject from the **Scene View**, the **Inspector**, or through Scripting.

The remainder of this page's text is from the Transform Component Reference page.

# Transform

The **Transform Component** determines the **Position**, **Rotation**, and **Scale** of each object in the scene. Every object has a Transform.




*The Transform Component is editable in the **Scene View** and in the **Inspector***

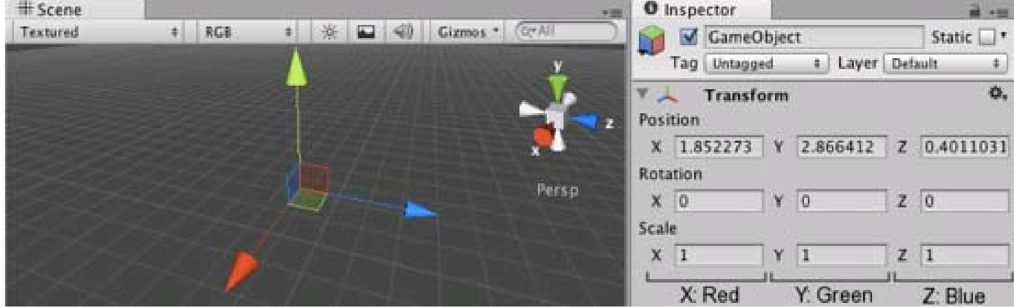
### Properties

| | |
|---|---|
| **Position** | Position of the Transform in X, Y, and Z coordinates. |
| **Rotation** | Rotation of the Transform around the X, Y, and Z axes, measured in degrees. |
| **Scale** | Scale of the Transform along X, Y, and Z axes. Value "1" is the original size (size at which the object was imported). |

All properties of a Transform are measured relative to the Transform's parent (see below for further details). If the Transform has no parent, the properties are measured relative to World Space.

## Using Transforms

Transforms are always manipulated in 3D space in the X, Y, and Z axes. In Unity, these axes are represented by the colors red, green, and blue respectively. Remember: XYZ = RGB.

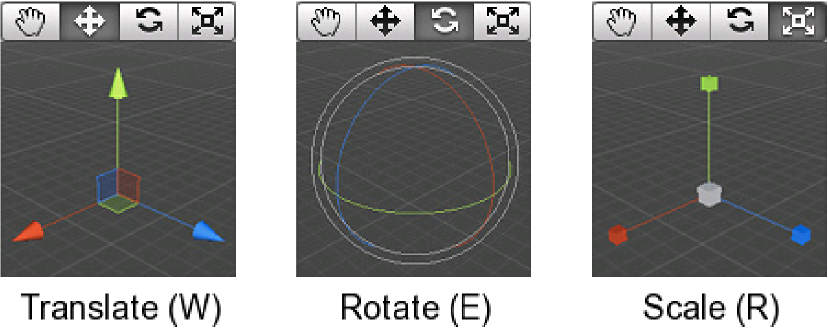



*Color-coded relationship between the three axes and Transform properties*

Transforms can be directly manipulated in the **Scene View** or by editing properties in the Inspector. In the scene, you can modify Transforms using the Move, Rotate and Scale tools. These tools are located in the upper left-hand corner of the Unity Editor.

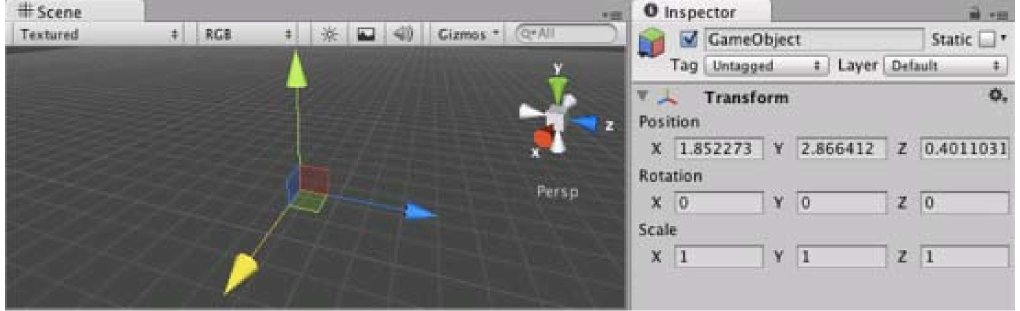
*The View, Translate, Rotate, and Scale tools*

The tools can be used on any object in the scene. When you click on an object, you will see the tool gizmo appear within it. The appearance of the gizmo depends on which tool is selected.




All three Gizmos can be directly edited in the Scene View.

When you click and drag on one of the three gizmo axes, you will notice that its color changes. As you drag the mouse, you will see the object translate, rotate, or scale along the selected axis. When you release the mouse button, the axis remains selected. You can click the middle mouse button and drag the mouse to manipulate the Transform along the selected axis.




*Any individual axis will become selected when you click on it*

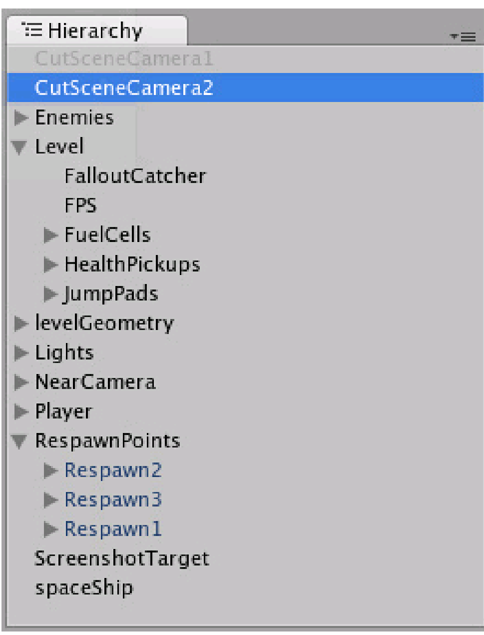
Around the centre of the Transform gizmo are three coloured squares. These allow you to drag the Transform in a single plane (ie, the object will move in two axes but be held still in the third axis).

*Dragging in the XZ plane*

### Parenting

Parenting is one of the most important concepts to understand when using Unity. When a GameObject is a **Parent** of another GameObject, the **Child** GameObject will move, rotate, and scale exactly as its Parent does. Just like your arms are attached to your body, when you turn your body, your arms move because they're attached. Any object can have multiple children, but only one parent.

You can create a Parent by dragging any GameObject in the **Hierarchy View** onto another. This will create a Parent-Child relationship between the two GameObjects.




*Example of a Parent-Child hierarchy. GameObjects with foldout arrows to the left of their names are parents.*

In the above example, we say that the arms are parented to the body and the hands are parented to the arms. The scenes you make in Unity will contain collections of these **Transform hierarchies**. The topmost parent object is called the **Root object**. When you move, scale or rotate a parent, all the changes in its Transform are applied to its children as well.

It is worth pointing out that the Transform values in the Inspector of any Child GameObject are displayed relative to the Parent's Transform values. These are also called the **Local Coordinates**. Through scripting, you can access the **Global Coordinates** as well as the local coordinates.

You can build compound objects by parenting several separate objects together, for example, the skeletal structure of a human ragdoll. You can also achieve useful effects with simple hierarchies. For example, if you have a horror game that takes place at night, you can create an effective atmosphere with a flashlight. To create this object, you would parent a spotlight Transform to the flashlight Transform. Then, any alteration of the flashlight Transform will affect the spotlight, creating a convincing flashlight effect.

### Performance Issues and Limitations with Non-Uniform Scaling

Non-uniform scaling is when the **Scale** in a Transform has different values for x, y, and z; for example (2, 4, 2). In contrast, uniform scaling has the same value for x, y, and z; for example (3, 3, 3). Non-uniform scaling can be useful in a few select cases but should be avoided whenever possible.

Non-uniform scaling has a negative impact on rendering performance. In order to transform vertex normals correctly, we transform the mesh on the CPU and create an extra copy of the data. Normally we can keep the mesh shared between instances in graphics memory, but in this case you pay both a CPU and memory cost per instance.

There are also certain limitations in how Unity handles non-uniform scaling:

- Certain components do not fully support non-uniform scaling. For example, for components with a **radius** property or similar, such as a **Sphere Collider**, **Capsule Collider**, **Light**, **Audio Source** etc., the shape will never become elliptical but remain circular/spherical regardless of non-uniform scaling.
- A child object that has a non-uniformly scaled parent and is rotated relative to that parent may have a non-orthogonal matrix, meaning that it may appear skewed. Some components that do support simple non-uniform scaling still do not support non-orthogonal matrices. For example, a **Box Collider** cannot be skewed so if its transform is non-orthogonal, the Box Collider will not match the shape of the rendered mesh accurately.
- For performance reasons, a child object that has a non-uniformly scaled parent will not have its scale/matrix automatically updated while rotating. This may result in popping of the scale once the scale is updated, for example if the object is detached from its parent.

### Importance of Scale

The scale of the Transform determines the difference between the size of your mesh in your modeling application and the size of your mesh in Unity. The mesh's size in Unity (and therefore the Transform's scale) is **very** important, especially during physics simulation. There are three factors that can affect the scale of your object:

- The size of your mesh in your 3D modeling application.
- The **Mesh Scale Factor** setting in the object's **Import Settings**.
- The **Scale** values of your Transform Component.

Ideally, you should not adjust the **Scale** of your object in the Transform Component. The best option is to create your models at real-life scale so you won't have to change your Transform's scale. The next best option is to adjust the scale at which your mesh is imported in the **Import Settings** for your individual mesh. Certain optimizations occur based on the import size, and instantiating an object that has an adjusted scale value can decrease performance. For more information, see the section about optimizing scale on the Rigidbody component reference page.

### Hints

- When parenting Transforms, set the parent's location to <0,0,0> before adding the child. This will save you many headaches later.
- **Particle Systems** are not affected by the Transform's **Scale**. In order to scale a Particle System, you need to modify the properties in the System's Particle Emitter, Animator and Renderer.
- If you are using **Rigidbodies** for physics simulation, there is some important information about the Scale property on the Rigidbody component reference page.
- You can change the colors of the Transform axes (and other UI elements) from the preferences (**Menu: Unity > Preferences** and then select the **Colors & keys** panel).
- It is best to avoid scaling within Unity if possible. Try to have the scales of your object finalized in your 3D modeling application, or in the **Import Settings** of your mesh.

Page last updated: 2007-11-16

---

# Physics

Unity has NVIDIA PhysX physics engine built-in. This allows for unique emergent behaviour and has many useful features.

### Basics

To put an object under physics control, simply add a **Rigidbody** to it. When you do this, the object will be affected by gravity, and can collide with other objects in the world.

#### Rigidbodies

Rigidbodies are physically simulated objects. You use Rigidbodies for things that the player can push around, for example crates or loose objects, or you can move Rigidbodies around directly by adding forces to it by scripting.

If you move the Transform of a non-Kinematic Rigidbody directly it may not collide correctly with other objects. Instead you should move a Rigidbody by applying forces and torque to it. You can also add Joints to rigidbodies to make the behavior more complex. For example, you could make a physical door or a crane with a swinging chain.

You also use Rigidbodies to bring vehicles to life, for example you can make cars using a Rigidbody, 4 Wheel Colliders and a script applying wheel forces based on the user's Input.

You can make airplanes by applying forces to the Rigidbody from a script. Or you can create special vehicles or robots by adding various Joints and applying forces via scripting.

Rigidbodies are most often used in combination with primitive colliders.

**Tips:**

- You should never have a parent and child rigidbody together
- You should never scale the parent of a rigidbody

**Kinematic Rigidbodies**

A **Kinematic Rigidbody** is a Rigidbody that has the isKinematic option enabled. Kinematic Rigidbodies are not affected by forces, gravity or collisions. They are driven explicitly by setting the position and rotation of the Transform or animating them, yet they can interact with other non-Kinematic Rigidbodies.

Kinematic Rigidbodies correctly wake up other Rigidbodies when they collide with them, and they apply friction to Rigidbodies placed on top of them.

These are a few example uses for Kinematic Rigidbodies:

1. Sometimes you want an object to be under physics control but in another situation to be controlled explicitly from a script or animation. For example you could make an animated character whose bones have Rigidbodies attached that are connected with joints for use as a Ragdoll. Most of the time the character is under animation control, thus you make the Rigidbody Kinematic. But when he gets hit you want him to turn into a Ragdoll and be affected by physics. To accomplish this, you simply disable the isKinematic property.
2. Sometimes you want a moving object that can push other objects yet not be pushed itself. For example if you have an animated platform and you want to place some Rigidbody boxes on top, you should make the platform a Kinematic Rigidbody instead of just a **Collider** without a Rigidbody.
3. You might want to have a Kinematic Rigidbody that is animated and have a real Rigidbody follow it using one of the available Joints.

**Static Colliders**

A **Static Collider** is a GameObject that has a Collider but not a Rigidbody. Static Colliders are used for level geometry which always stays at the same place and never moves around. You can add a **Mesh Collider** to your already existing graphical meshes (even better use the **Import Settings** Generate Colliders check box), or you can use one of the other Collider types.

You should never move a Static Collider on a frame by frame basis. Moving Static Colliders will cause an internal recomputation in PhysX that is quite expensive and which will result in a big drop in performance. On top of that the behaviour of waking up other Rigidbodies based on a Static Collider is undefined, and moving Static Colliders will not apply friction to Rigidbodies that touch it. Instead, Colliders that move should always be Kinematic Rigidbodies.

**Character Controllers**

You use Character Controllers if you want to make a humanoid character. This could be the main character in a third person platformer, FPS shooter or any enemy characters.

These Controllers don't follow the rules of physics since it will not feel right (in Doom you run 90 miles per hour, come to halt in one frame and turn on a dime). Instead, a Character Controller performs collision detection to make sure your characters can slide along walls, walk up and down stairs, etc.

Character Controllers are not affected by forces but they can push Rigidbodies by applying forces to them from a script. Usually, all humanoid characters are implemented using Character Controllers.

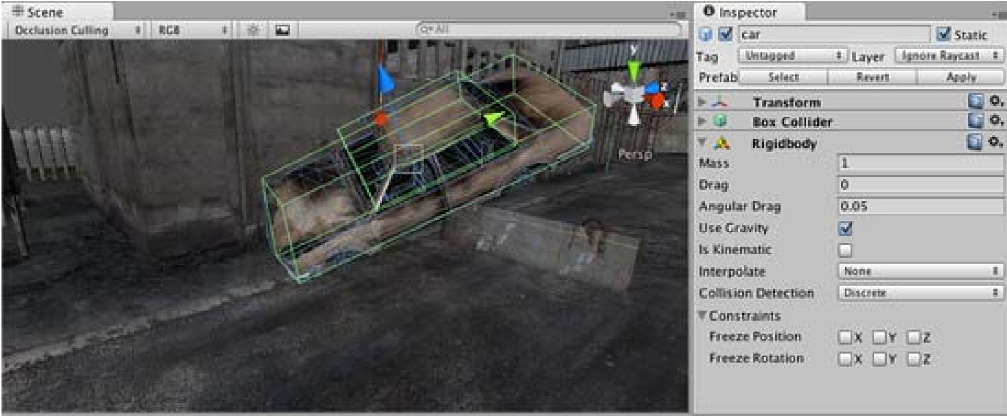
Character Controllers are inherently unphysical, thus if you want to apply real physics - Swing on ropes, get pushed by big rocks - to your character you have to use a Rigidbody, this will let you use joints and forces on your character. Character

Controllers are always aligned along the Y axis, so you also need to use a Rigidbody if your character needs to be able to change orientation in space (for example under a changing gravity). However, be aware that tuning a Rigidbody to feel right for a character is hard due to the unphysical way in which game characters are expected to behave. Another difference is that Character Controllers can slide smoothly over steps of a specified height, while Rigidbodies will not.

If you parent a Character Controller with a Rigidbody you will get a "Joint" like behavior.

# Rigidbody

**Rigidbodies** enable your **GameObjects** to act under the control of physics. The Rigidbody can receive forces and torque to make your objects move in a realistic way. Any GameObject must contain a Rigidbody to be influenced by gravity, act under added forces via scripting, or interact with other objects through the NVIDIA PhysX physics engine.




*Rigidbodies allow GameObjects to act under physical influence*

## Properties

| | |
|---|---|
| **Mass** | The mass of the object (arbitrary units). It is recommended to make masses not more or less than 100 times that of other Rigidbodies. |
| **Drag** | How much air resistance affects the object when moving from forces. 0 means no air resistance, and infinity makes the object stop moving immediately. |
| **Angular Drag** | How much air resistance affects the object when rotating from torque. 0 means no air resistance, and infinity makes the object stop rotating immediately. |
| **Use Gravity** | If enabled, the object is affected by gravity. |
| **Is Kinematic** | If enabled, the object will not be driven by the physics engine, and can only be manipulated by its **Transform**. This is useful for moving platforms or if you want to animate a Rigidbody that has a **HingeJoint** attached. |
| **Interpolate** | Try one of the options only if you are seeing jerkiness in your Rigidbody's movement. |
| **None** | No Interpolation is applied. |
| **Interpolate** | Transform is smoothed based on the Transform of the previous frame. |
| **Extrapolate** | Transform is smoothed based on the estimated Transform of the next frame. |
| **Collision Detection** | Used to prevent fast moving objects from passing through other objects without detecting collisions. |
| **Discrete** | Use Discreet collision detection against all other colliders in the scene. Other colliders will use Discreet collision detection when testing for collision against it. Used for normal collisions (This is the default value). |
| **Continuous** | Use Discrete collision detection against dynamic colliders (with a rigidbody) and continuous collision detection against static MeshColliders (without a rigidbody). Rigidbodies set to Continuous Dynamic will use continuous collision detection when testing for collision against this rigidbody. Other rigidbodies will use Discreet Collision detection. Used for objects which the Continuous Dynamic detection needs to collide with. (This has a big impact on physics performance, leave it set to Discrete, if you don't have issues with collisions of fast objects) |
| **Continuous Dynamic** | Use continuous collision detection against objects set to Continuous and Continuous Dynamic Collision. It will also use continuous collision detection against static MeshColliders (without a rigidbody). For all other colliders it uses discreet collision detection. Used for fast moving objects. |
| **Constraints** | Restrictions on the Rigidbody's motion:- |
| **Freeze Position** | Stops the Rigidbody moving in the world X, Y and Z axes selectively. |
| **Freeze Rotation** | Stops the Rigidbody rotating around the world X, Y and Z axes selectively. |

## Details

Rigidbodies allow your GameObjects to act under control of the physics engine. This opens the gateway to realistic collisions, varied types of joints, and other very cool behaviors. Manipulating your GameObjects by adding forces to a Rigidbody creates a very different feel and look than adjusting the Transform **Component** directly. Generally, you shouldn't manipulate the Rigidbody and the Transform of the same GameObject - only one or the other.

The biggest difference between manipulating the Transform versus the Rigidbody is the use of forces. Rigidbodies can receive forces and torque, but Transforms cannot. Transforms can be translated and rotated, but this is not the same as using physics. You'll notice the distinct difference when you try it for yourself. Adding forces/torque to the Rigidbody will actually change the object's position and rotation of the Transform component. This is why you should only be using one or the other. Changing the Transform while using physics could cause problems with collisions and other calculations.

Rigidbodies must be explicitly added to your GameObject before they will be affected by the physics engine. You can add a Rigidbody to your selected object from **Components->Physics->Rigidbody** in the menubar. Now your object is physics-ready; it will fall under gravity and can receive forces via scripting, but you may need to add a **Collider** or a Joint to get it to behave exactly how you want.

**Parenting**

When an object is under physics control, it moves semi-independently of the way its transform parents move. If you move any parents, they will pull the Rigidbody child along with them. However, the Rigidbodies will still fall down due to gravity and react to collision events.

**Scripting**

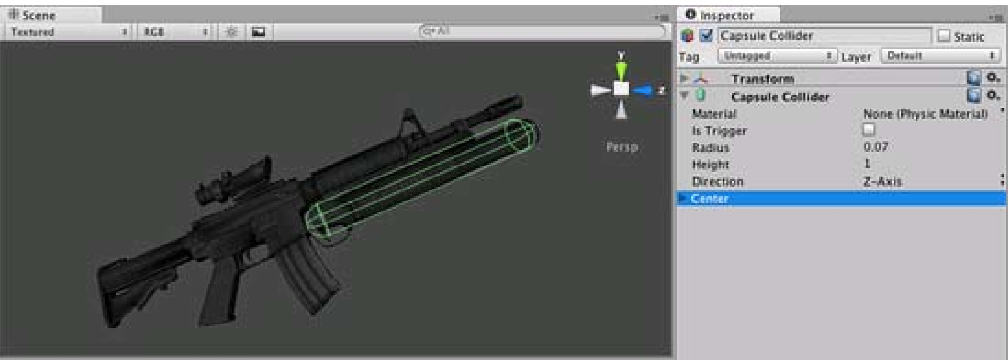
To control your Rigidbodies, you will primarily use scripts to add forces or torque. You do this by calling **AddForce()** and **AddTorque()** on the object's Rigidbody. Remember that you shouldn't be directly altering the object's Transform when you are using physics.

**Animation**

For some situations, mainly creating ragdoll effects, it is neccessary to switch control of the object between animations and physics. For this purpose Rigidbodies can be marked **isKinematic**. While the Rigidbody is marked **isKinematic**, it will not be affected by collisions, forces, or any other part of physX. This means that you will have to control the object by manipulating the Transform component directly. Kinematic Rigidbodies will affect other objects, but they themselves will not be affected by physics. For example, Joints which are attached to Kinematic objects will constrain any other Rigidbodies attached to them and Kinematic Rigidbodies will affect other Rigidbodies through collisions.

**Colliders**

Colliders are another kind of component that must be added alongside the Rigidbody in order to allow collisions to occur. If two Rigidbodies bump into each other, the physics engine will not calculate a collision unless both objects also have a Collider attached. Collider-less Rigidbodies will simply pass through each other during physics simulation.



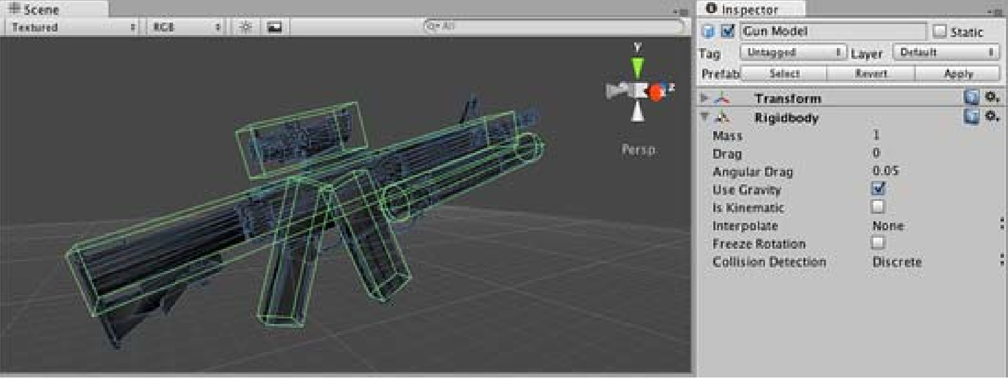

*Colliders define the physical boundaries of a Rigidbody*

Add a Collider with the **Component->Physics** menu. View the Component Reference page of any individual Collider for more specific information:

- Box Collider - primitive shape of a cube
- Sphere Collider - primitive shape of a sphere
- Capsule Collider - primitive shape of a capsule
- Mesh Collider - creates a collider from the object's mesh, cannot collide with another Mesh Collider
- Wheel Collider - specifically for creating cars or other moving vehicles

**Compound Colliders**

Compound Colliders are combinations of primitive Colliders, collectively acting as a single Collider. They come in handy when you have a complex mesh to use in collisions but cannot use a **Mesh Collider**. To create a Compound Collider, create child objects of your colliding object, then add a primitive Collider to each child object. This allows you to position, rotate, and scale each Collider easily and independently of one another.




*A real-world Compound Collider setup*

In the above picture, the *Gun Model* GameObject has a Rigidbody attached, and multiple primitive Colliders as child GameObjects. When the Rigidbody parent is moved around by forces, the child Colliders move along with it. The primitive Colliders will collide with the environment's Mesh Collider, and the parent Rigidbody will alter the way it moves based on forces being applied to it and how its child Colliders interact with other Colliders in the Scene.

Mesh Colliders can't normally collide with each other. If a Mesh Collider is marked as **Convex**, then it can collide with another Mesh Collider. The typical solution is to use primitive Colliders for any objects that move, and Mesh Colliders for static background objects.

**Continuous Collision Detection**

Continuous collision detection is a feature to prevent fast-moving colliders from passing each other. This may happen when using normal (**Discrete**) collision detection, when an object is one side of a collider in one frame, and already passed the collider in the next frame. To solve this, you can enable continuous collision detection on the rigidbody of the fast-moving object. Set the collision detection mode to **Continuous** to prevent the rigidbody from passing through any static (ie, non-rigidbody) MeshColliders. Set it to **Continuous Dynamic** to also prevent the rigidbody from passing through any other supported rigidbodies with collision detection mode set to **Continuous** or **Continuous Dynamic**. Continuous collision detection is supported for Box-, Sphere- and CapsuleColliders. Note that continuous collision detection is intended as a safety net to catch collisions in cases where objects would otherwise pass through each other, but will not deliver physically accurate collision results, so you might still consider decreasing the fixed Time step value in the TimeManager inspector to make the simulation more precise, if you run into problems with fast moving objects.

## Use the right size

The size of the your GameObject's mesh is much more important than the mass of the Rigidbody. If you find that your Rigidbody is not behaving exactly how you expect - it moves slowly, floats, or doesn't collide correctly - consider adjusting the scale of your mesh asset. Unity's default unit scale is 1 unit = 1 meter, so the scale of your imported mesh is maintained, and applied to physics calculations. For example, a crumbling skyscraper is going to fall apart very differently than a tower made of toy blocks, so objects of different sizes should be modeled to accurate scale.

If you are modeling a human make sure he is around 2 meters tall in Unity. To check if your object has the right size compare it to the default cube. You can create a cube using **GameObject->Create Other->Cube**. The cube's height will be exactly 1 meter, so your human should be twice as tall.

If you aren't able to adjust the mesh itself, you can change the uniform scale of a particular mesh asset by selecting it in **Project View** and choosing **Assets->Import Settings...** from the menubar. Here, you can change the scale and re-import your mesh.

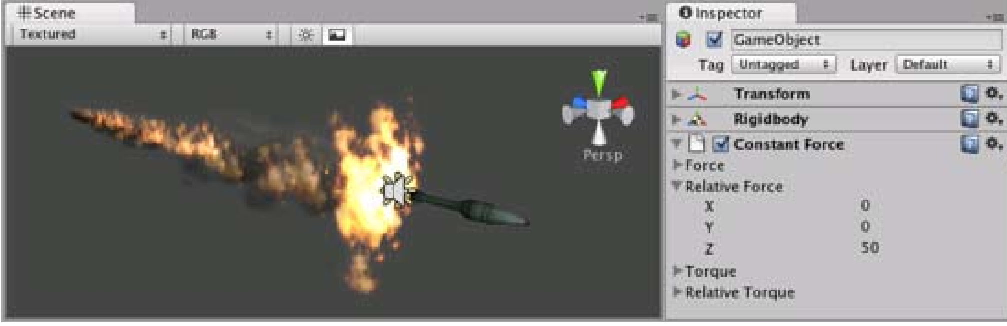
If your game requires that your GameObject needs to be instantiated at different scales, it is okay to adjust the values of your Transform's scale axes. The downside is that the physics simulation must do more work at the time the object is instantiated, and could cause a performance drop in your game. This isn't a terrible loss, but it is not as efficient as finalizing your scale with the other two options. Also keep in mind that non-uniform scales can create undesirable behaviors when Parenting is used. For these reasons it is always optimal to create your object at the correct scale in your modeling application.

### Hints

- The relative **Mass** of two Rigidbodies determines how they react when they collide with each other.
- Making one Rigidbody have greater **Mass** than another does not make it fall faster in free fall. Use **Drag** for that.
- A low **Drag** value makes an object seem heavy. A high one makes it seem light. Typical values for **Drag** are between .001 (solid block of metal) and 10 (feather).
- If you are directly manipulating the Transform component of your object but still want physics, attach a Rigidbody and make it Kinematic.
- If you are moving a GameObject through its Transform component but you want to receive Collision/Trigger messages, you must attach a Rigidbody to the object that is moving.

## Constant Force

**Constant Force** is a quick utility for adding constant forces to a **Rigidbody**. This works great for one shot objects like rockets, if you don't want it to start with a large velocity but instead accelerate.




*A rocket propelled forward by a Constant Force*

### Properties

| | |
|---|---|
| **Force** | The vector of a force to be applied in world space. |
| **Relative Force** | The vector of a force to be applied in the object's local space. |
| **Torque** | The vector of a torque, applied in world space. The object will begin spinning *around* this vector. The longer the vector is, the faster the rotation. |
| **Relative Torque** | The vector of a torque, applied in local space. The object will begin spinning *around* this vector. The longer the vector is, the faster the rotation. |

#### Details

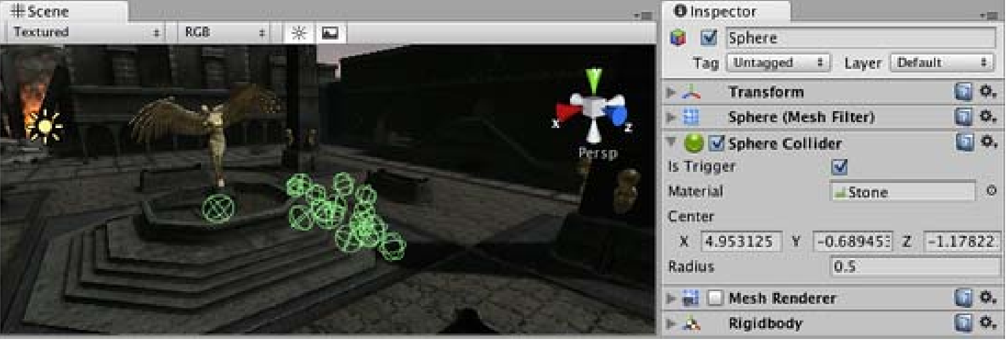
To make a rocket that accelerates forward set the **Relative Force** to be along the positive z-axis. Then use the Rigidbody's **Drag** property to make it not exceed some maximum velocity (the higher the drag the lower the maximum velocity will be). In the Rigidbody, also make sure to turn off gravity so that the rocket will always stay on its path.

### Hints

- To make an object flow upwards, add a Constant Force with the **Force** property having a positive Y value.
- To make an object fly forwards, add a Constant Force with the **Relative Force** property having a positive Z value.

## Sphere Collider

The **Sphere Collider** is a basic sphere-shaped collision primitive.



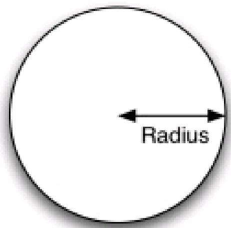

*A pile of Sphere Colliders*

## Properties

| | |
|---|---|
| **Is Trigger** | If enabled, this Collider is used for triggering events, and is ignored by the physics engine. |
| **Material** | Reference to the Physics Material that determines how this Collider interacts with others. |
| **Radius** | The size of the Collider. |
| **Center** | The position of the Collider in the object's local space. |

## Details

The Sphere Collider can be resized to uniform scale, but not along individual axes. It works great for falling boulders, ping pong balls, marbles, etc.




*A standard Sphere Collider*

Colliders work with Rigidbodies to bring physics in Unity to life. Whereas Rigidbodies allow objects to be controlled by physics, Colliders allow objects to collide with each other. Colliders must be added to objects independently of Rigidbodies. A Collider does not necessarily need a Rigidbody attached, but a Rigidbody **must** be attached in order for the object to move as a result of collisions.

When a collision between two Colliders occurs and if at least one of them has a Rigidbody attached, three collision messages are sent out to the objects attached to them. These events can be handled in scripting, and allow you to create unique behaviors with or without making use of the built-in NVIDIA PhysX engine.

#### Triggers

An alternative way of using Colliders is to mark them as a **Trigger**, just check the IsTrigger property checkbox in the Inspector. Triggers are effectively ignored by the physics engine, and have a unique set of three trigger messages that are sent out when a collision with a Trigger occurs. Triggers are useful for triggering other events in your game, like cutscenes, automatic door opening, displaying tutorial messages, etc. Use your imagination!

Be aware that in order for two Triggers to send out trigger events when they collide, one of them must include a Rigidbody as well. For a Trigger to collide with a normal Collider, one of them must have a Rigidbody attached. For a detailed chart of different types of collisions, see the collision action matrix in the Advanced section below.

#### Friction and bounciness

Friction, bounciness and softness are defined in the Physisc Material. The Standard Assets contain the most common physics materials. To use one of them click on the Physics Material drop-down and select one, eg. Ice. You can also create your own physics materials and tweak all friction values.

## Compound Colliders

Compound Colliders are combinations of primitive Colliders, collectively acting as a single Collider. They come in handy when you have a complex mesh to use in collisions but cannot use a **Mesh Collider**. To create a Compound Collider, create child objects of your colliding object, then add a primitive Collider to each child object. This allows you to position, rotate, and scale each Collider easily and independently of one another.

*A real-world Compound Collider setup*

In the above picture, the *Gun Model* GameObject has a Rigidbody attached, and multiple primitive Colliders as child GameObjects. When the Rigidbody parent is moved around by forces, the child Colliders move along with it. The primitive Colliders will collide with the environment's Mesh Collider, and the parent Rigidbody will alter the way it moves based on forces being applied to it and how its child Colliders interact with other Colliders in the Scene.

Mesh Colliders can't normally collide with each other. If a Mesh Collider is marked as **Convex**, then it can collide with another Mesh Collider. The typical solution is to use primitive Colliders for any objects that move, and Mesh Colliders for static background objects.

## Hints

- To add multiple Colliders for an object, create child GameObjects and attach a Collider to each one. This allows each Collider to be manipulated independently.
- You can look at the gizmos in the **Scene View** to see how the Collider is being calculated on your object.
- Colliders do their best to match the scale of an object. If you have a non-uniform scale (a scale which is different in each direction), only the Mesh Collider can match completely.
- If you are moving an object through its Transform component but you want to receive Collision/Trigger messages, you must attach a Rigidbody to the object that is moving.
- If you make an explosion, it can be very effective to add a rigidbody with lots of drag and a sphere collider to it in order to push it out a bit from the wall it hits.

## Advanced

### Collider combinations

There are numerous different combinations of collisions that can happen in Unity. Each game is unique, and different combinations may work better for different types of games. If you're using physics in your game, it will be very helpful to understand the different basic Collider types, their common uses, and how they interact with other types of objects.

### Static Collider

These are GameObjects that do **not** have a Rigidbody attached, but **do** have a Collider attached. These objects should remain still, or move very little. These work great for your environment geometry. They will not move if a Rigidbody collides with them.

### Rigidbody Collider

These GameObjects contain both a Rigidbody and a Collider. They are completely affected by the physics engine through scripted forces and collisions. They might collide with a GameObject that only contains a Collider. These will likely be your primary type of Collider in games that use physics.

### Kinematic Rigidbody Collider

This GameObject contains a Collider and a Rigidbody which is marked **IsKinematic**. To move this GameObject, you modify its Transform Component, rather than applying forces. They're similar to Static Colliders but will work better when you want to move the Collider around frequently. There are some other specialized scenarios for using this GameObject.

This object can be used for circumstances in which you would normally want a Static Collider to send a trigger event. Since a Trigger must have a Rigidbody attached, you should add a Rigidbody, then enable **IsKinematic**. This will prevent your Object from moving from physics influence, and allow you to receive trigger events when you want to.

Kinematic Rigidbodies can easily be turned on and off. This is great for creating ragdolls, when you normally want a character

to follow an animation, then turn into a ragdoll when a collision occurs, prompted by an explosion or anything else you choose. When this happens, simply turn all your Kinematic Rigidbodies into normal Rigidbodies through scripting.

If you have Rigidbodies come to rest so they are not moving for some time, they will "fall asleep". That is, they will not be calculated during the physics update since they are not going anywhere. If you move a Kinematic Rigidbody out from underneath normal Rigidbodies that are at rest on top of it, the sleeping Rigidbodies will "wake up" and be correctly calculated again in the physics update. So if you have a lot of Static Colliders that you want to move around and have different object fall on them correctly, use Kinematic Rigidbody Colliders.

**Collision action matrix**

Depending on the configurations of the two colliding Objects, a number of different actions can occur. The chart below outlines what you can expect from two colliding Objects, based on the components that are attached to them. Some of the combinations only cause one of the two Objects to be affected by the collision, so keep the standard rule in mind - physics will not be applied to objects that do not have Rigidbodies attached.

| **Collision detection occurs and messages are sent upon collision** | | | | | | |
|---|---|---|---|---|---|---|
| | Static Collider | Rigidbody Collider | Kinematic Rigidbody Collider | Static Trigger Collider | Rigidbody Trigger Collider | Kinematic Rigidbody Trigger Collider |
| Static Collider | | Y | | | | |
| Rigidbody Collider | Y | Y | Y | | | |
| Kinematic Rigidbody Collider | | Y | | | | |
| Static Trigger Collider | | | | | | |
| Rigidbody Trigger Collider | | | | | | |
| Kinematic Rigidbody Trigger Collider | | | | | | |

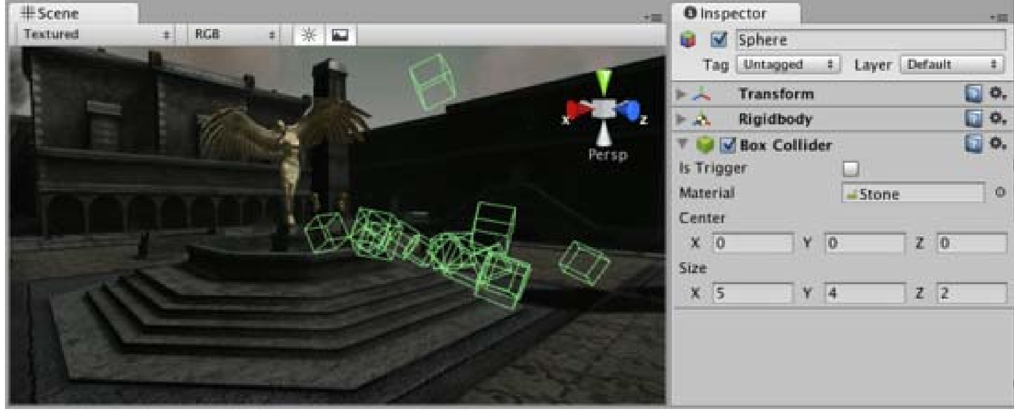
| **Trigger messages are sent upon collision** | | | | | | |
|---|---|---|---|---|---|---|
| | Static Collider | Rigidbody Collider | Kinematic Rigidbody Collider | Static Trigger Collider | Rigidbody Trigger Collider | Kinematic Rigidbody Trigger Collider |
| Static Collider | | | | | Y | Y |
| Rigidbody Collider | | | | Y | Y | Y |
| Kinematic Rigidbody Collider | | | | Y | Y | Y |
| Static Trigger Collider | | Y | Y | | Y | Y |
| Rigidbody Trigger Collider | Y | Y | Y | Y | Y | Y |
| Kinematic Rigidbody Trigger Collider | Y | Y | Y | Y | Y | Y |

### Layer-Based Collision Detection

In Unity 3.x we introduce something called **Layer-Based Collision Detection**, and you can now selectively tell Unity GameObjects to collide with specific layers they are attached to. For more information click here

## Box Collider

The **Box Collider** is a basic cube-shaped collision primitive.



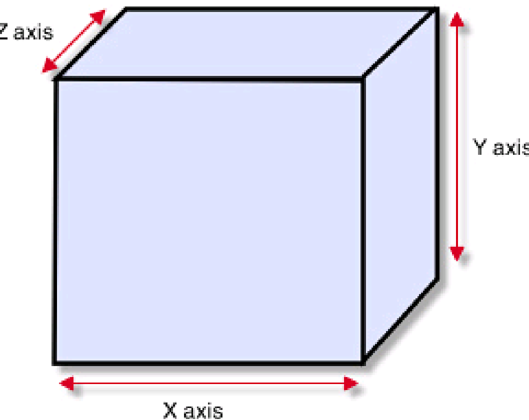
*A pile of Box Colliders*

### Properties

| | |
|---|---|
| **Is Trigger** | If enabled, this Collider is used for triggering events, and is ignored by the physics engine. |
| **Material** | Reference to the Physics Material that determines how this Collider interacts with others. |
| **Center** | The position of the Collider in the object's local space. |
| **Size** | The size of the Collider in the X, Y, Z directions. |

### Details

The Box Collider can be resized into different shapes of rectangular prisms. It works great for doors, walls, platforms, etc. It is also effective as a human torso in a ragdoll or as a car hull in a vehicle. Of course, it works perfectly for just boxes and crates as well!




*A standard Box Collider*

Colliders work with Rigidbodies to bring physics in Unity to life. Whereas Rigidbodies allow objects to be controlled by physics, Colliders allow objects to collide with each other. Colliders must be added to objects independently of Rigidbodies. A Collider does not necessarily need a Rigidbody attached, but a Rigidbody **must** be attached in order for the object to move as a result of collisions.

When a collision between two Colliders occurs and if at least one of them has a Rigidbody attached, three collision messages are sent out to the objects attached to them. These events can be handled in scripting, and allow you to create unique behaviors with or without making use of the built-in NVIDIA PhysX engine.

#### Triggers

An alternative way of using Colliders is to mark them as a **Trigger**, just check the IsTrigger property checkbox in the Inspector. Triggers are effectively ignored by the physics engine, and have a unique set of three trigger messages that are sent out when a collision with a Trigger occurs. Triggers are useful for triggering other events in your game, like cutscenes, automatic door opening, displaying tutorial messages, etc. Use your imagination!

Be aware that in order for two Triggers to send out trigger events when they collide, one of them must include a Rigidbody as well. For a Trigger to collide with a normal Collider, one of them must have a Rigidbody attached. For a detailed chart of different types of collisions, see the collision action matrix in the Advanced section below.

#### Friction and bounciness

Friction, bounciness and softness are defined in the Physisc Material. The Standard Assets contain the most common physics materials. To use one of them click on the Physics Material drop-down and select one, eg. Ice. You can also create your own physics materials and tweak all friction values.

## Mesh Collider

The **Mesh Collider** takes a Mesh Asset and builds its Collider based on that mesh. It is far more accurate for collision detection than using primitives for complicated meshes. Mesh Colliders that are marked as **Convex** can collide with other Mesh Colliders.

*A Mesh Collider used on level geometry*

## Properties

| | |
|---|---|
| **Is Trigger** | If enabled, this Collider is used for triggering events, and is ignored by the physics engine. |
| **Material** | Reference to the Physics Material that determines how this Collider interacts with others. |
| **Mesh** | Reference to the Mesh to use for collisions. |
| **Smooth Sphere Collisions** | When this is enabled, collision mesh normals are smoothed. You should enable this on smooth surfaces eg. rolling terrain without hard edges to make sphere rolling smoother. |
| **Convex** | If enabled, this Mesh Collider will collide with other Mesh Colliders. Convex Mesh Colliders are limited to 255 triangles. |

## Details

The Mesh Collider builds its collision representation from the Mesh attached to the GameObject, and reads the properties of the attached Transform to set its position and scale correctly.

Collision meshes use backface culling. If an object collides with a mesh that will be backface culled graphically it will also not collide with it physically.

There are some limitations when using the Mesh Collider. Usually, two Mesh Colliders cannot collide with each other. All Mesh Colliders can collide with any primitive Collider. If your mesh is marked as **Convex**, then it can collide with other Mesh Colliders.

Colliders work with Rigidbodies to bring physics in Unity to life. Whereas Rigidbodies allow objects to be controlled by physics, Colliders allow objects to collide with each other. Colliders must be added to objects independently of Rigidbodies. A Collider does not necessarily need a Rigidbody attached, but a Rigidbody **must** be attached in order for the object to move as a result of collisions.

When a collision between two Colliders occurs and if at least one of them has a Rigidbody attached, three collision messages are sent out to the objects attached to them. These events can be handled in scripting, and allow you to create unique behaviors with or without making use of the built-in NVIDIA PhysX engine.

### Triggers

An alternative way of using Colliders is to mark them as a **Trigger**, just check the IsTrigger property checkbox in the Inspector. Triggers are effectively ignored by the physics engine, and have a unique set of three trigger messages that are sent out when a collision with a Trigger occurs. Triggers are useful for triggering other events in your game, like cutscenes, automatic door opening, displaying tutorial messages, etc. Use your imagination!

Be aware that in order for two Triggers to send out trigger events when they collide, one of them must include a Rigidbody as well. For a Trigger to collide with a normal Collider, one of them must have a Rigidbody attached. For a detailed chart of different types of collisions, see the collision action matrix in the Advanced section below.

### Friction and bounciness

Friction, bounciness and softness are defined in the Physic Material. The Standard Assets contain the most common physics materials. To use one of them click on the Physics Material drop-down and select one, eg. Ice. You can also create your own physics materials and tweak all friction values.

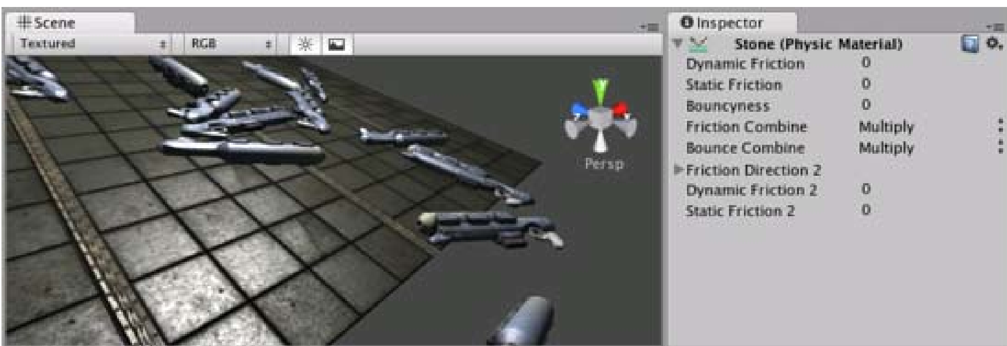
## Hints

- Mesh Colliders **cannot** collide with each other unless they are marked as **Convex**. Therefore, they are most useful for background objects like environment geometry.
- **Convex** Mesh Colliders must be fewer than 255 triangles.
- Primitive Colliders are less costly for objects under physics control.
- When you attach a Mesh Collider to a **GameObject**, its Mesh property will default to the mesh being rendered. You can change that by assigning a different Mesh.
- To add multiple Colliders for an object, create child GameObjects and attach a Collider to each one. This allows each Collider to be manipulated independently.
- You can look at the gizmos in the **Scene View** to see how the Collider is being calculated on your object.
- Colliders do their best to match the scale of an object. If you have a non-uniform scale (a scale which is different in each direction), only the Mesh Collider can match completely.
- If you are moving an object through its Transform component but you want to receive Collision/Trigger messages, you must attach a Rigidbody to the object that is moving.

# Physics Material

The **Physics Material** is used to adjust friction and bouncing effects of colliding objects.

To create a Physics Material select **Assets->Create->Physics Material** from the menu bar. Then drag the Physics Material from the Project View onto a **Collider** in the scene.




*The Physics Material* ***Inspector***

## Properties

| | |
|---|---|
| **Dynamic Friction** | The friction used when already moving. Usually a value from 0 to 1. A value of zero feels like ice, a value of 1 will make it come to rest very quickly unless a lot of force or gravity pushes the object. |
| **Static Friction** | The friction used when an object is laying still on a surface. Usually a value from 0 to 1. A value of zero feels like ice, a value of 1 will make it very hard to get the object moving. |
| **Bounciness** | How bouncy is the surface? A value of 0 will not bounce. A value of 1 will bounce without any loss of energy. |
| **Friction Combine Mode** | How the friction of two colliding objects is combined. |
| **Average** | The two friction values are averaged. |
| **Min** | The smallest of the two values is used. |
| **Max** | The largest of the two values is used. |
| **Multiply** | The friction values are multiplied with each other. |
| **Bounce Combine** | How the bounciness of two colliding objects is combined. It has the same modes as Friction Combine Mode |
| **Friction Direction 2** | The direction of anisotropy. Anisotropic friction is enabled if this direction is not zero. Dynamic Friction 2 and Static Friction 2 will be applied along Friction Direction 2. |
| **Dynamic Friction 2** | If anisotropic friction is enabled, DynamicFriction2 will be applied along Friction Direction 2. |
| **Static Friction 2** | If anisotropic friction is enabled, StaticFriction2 will be applied along Friction Direction 2. |

## Details

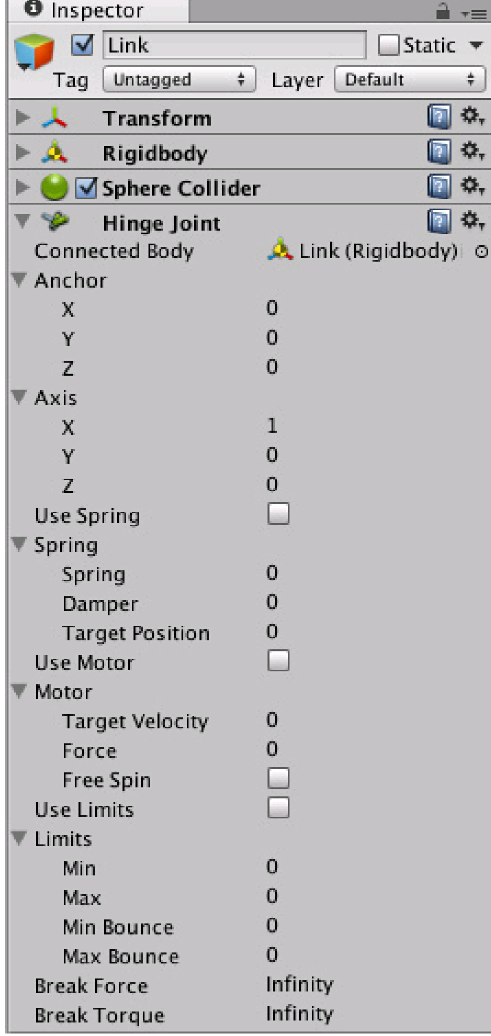
Friction is the quantity which prevents surfaces from sliding off each other. This value is critical when trying to stack objects. Friction comes in two forms, dynamic and static. **Static friction** is used when the object is lying still. It will prevent the object from starting to move. If a large enough force is applied to the object it will start moving. At this point **Dynamic Friction** will come into play. **Dynamic Friction** will now attempt to slow down the object while in contact with another.

### Hints

- Don't try to use a standard physics material for the main character. Make a customized one and get it perfect.

## Hinge Joint

The **Hinge Joint** groups together two Rigidbodies, constraining them to move like they are connected by a hinge. It is perfect for doors, but can also be used to model chains, pendulums, etc.




*The Hinge Joint **Inspector***

### Properties

| | |
|---|---|
| **Connected Body** | Optional reference to the Rigidbody that the joint is dependent upon. If not set, the joint connects to the world. |
| **Anchor** | The position of the axis around which the body swings. The position is defined in local space. |
| **Axis** | The direction of the axis around which the body swings. The direction is defined in local space. |
| **Use Spring** | Spring makes the Rigidbody reach for a specific angle compared to its connected body. |
| **Spring** | Properties of the Spring that are used if **Use Spring** is enabled. |
| **Spring** | The force the object asserts to move into the position. |
| **Damper** | The higher this value, the more the object will slow down. |
| **Target Position** | Target angle of the spring. The spring pulls towards this angle measured in degrees. |
| **Use Motor** | The motor makes the object spin around. |
| **Motor** | Properties of the Motor that are used if **Use Motor** is enabled. |
| **Target Velocity** | The speed the object tries to attain. |
| **Force** | The force applied in order to attain the speed. |
| **Free Spin** | If enabled, the motor is never used to brake the spinning, only accelerate it. |
| **Use Limits** | If enabled, the angle of the hinge will be restricted within the **Min** & **Max** values. |
| **Limits** | Properties of the Limits that are used if **Use Limits** is enabled. |

| | |
|---|---|
| **Min** | The lowest angle the rotation can go. |
| **Max** | The highest angle the rotation can go. |
| **Min Bounce** | How much the object bounces when it hits the minimum stop. |
| **Max Bounce** | How much the object bounces when it hits the maximum stop. |
| **Break Force** | The force that needs to be applied for this joint to break. |
| **Break Torque** | The torque that needs to be applied for this joint to break. |

### Details

A single Hinge Joint should be applied to a **GameObject**. The hinge will rotate at the point specified by the **Anchor** property, moving around the specified **Axis** property. You **do not** need to assign a GameObject to the joint's **Connected Body** property. You should only assign a GameObject to the **Connected Body** property if you want the joint's **Transform** to be dependent on the attached object's Transform.

Think about how the hinge of a door works. The **Axis** in this case is up, positive along the Y axis. The **Anchor** is placed somewhere at the intersection between door and wall. You would not need to assign the wall to the **Connected Body**, because the joint will be connected to the world by default.

Now think about a doggy door hinge. The doggy door's **Axis** would be sideways, positive along the relative X axis. The main door should be assigned as the **Connected Body**, so the doggy door's hinge is dependent on the main door's Rigidbody.

#### Chains

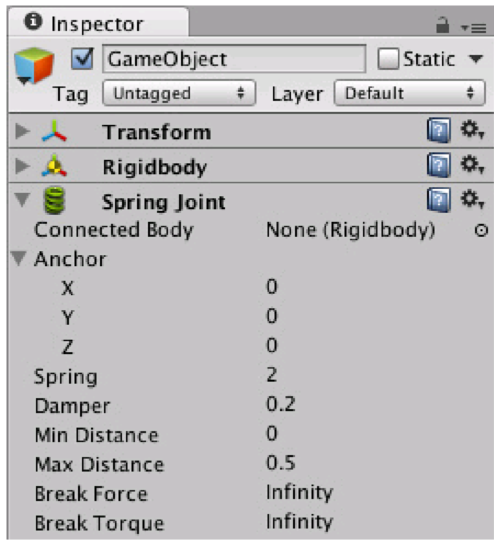
Multiple Hinge Joints can also be strung together to create a chain. Add a joint to each link in the chain, and attach the next link as the **Connected Body**.

### Hints

- You do not need to assign a **Connected Body** to your joint for it to work.
- Use **Break Force** in order to make dynamic damage systems. This is really cool as it allows the player to break a door off its hinge by blasting it with a rocket launcher or running into it with a car.
- The **Spring**, **Motor**, and **Limits** properties allow you to fine-tune your joint's behaviors.

## Spring Joint

The **Spring Joint** groups together two **Rigidbodies**, constraining them to move like they are connected by a spring.




*The Spring Joint **Inspector***

### Properties

| | |
|---|---|
| **Connected Body** | Optional reference to the Rigidbody that the joint is dependent upon. |
| **Anchor** | Position in the object's local space (at rest) that defines the center of the joint. This is not the point that the object will be drawn toward. |
| **X** | Position of the joint's local center along the X axis. |
| **Y** | Position of the joint's local center along the Y axis. |
| **Z** | Position of the joint's local center along the Z axis. |

| | |
|---|---|
| **Spring** | Strength of the spring. |
| **Damper** | Amount that the spring is reduced when active. |
| **Min Distance** | Distances greater than this will not cause the Spring to activate. |
| **Max Distance** | Distances less than this will not cause the Spring to activate. |
| **Break Force** | The force that needs to be applied for this joint to break. |
| **Break Torque** | The torque that needs to be applied for this joint to break. |

### Details

Spring Joints allows a Rigidbodied **GameObject** to be pulled toward a particular "target" position. This position will either be another Rigidbodied GameObject or the world. As the GameObject travels further away from this "target" position, the Spring Joint applies forces that will pull it back to its original "target" position. This creates an effect very similar to a rubber band or a slingshot.

The "target" position of the Spring is determined by the relative position from the **Anchor** to the **Connected Body** (or the world) when the Spring Joint is created, or when Play mode is entered. This makes the Spring Joint very effective at setting up Jointed characters or objects in the Editor, but is harder to create push/pull spring behaviors in runtime through scripting. If you want to primarily control a GameObject's position using a Spring Joint, it is best to create an empty GameObject with a Rigidbody, and set that to be the **Connected Rigidbody** of the Jointed object. Then in scripting you can change the position of the **Connected Rigidbody** and see your Spring move in the ways you expect.

#### Connected Rigidbody

You do not need to use a **Connected Rigidbody** for your joint to work. Generally, you should only use one if your object's position and/or rotation is dependent on it. If there is no **Connected Rigidbody**, your Spring will connect to the world.

#### Spring & Damper

**Spring** is the strength of the force that draws the object back toward its "target" position. If this is 0, then there is no force that will pull on the object, and it will behave as if no Spring Joint is attached at all.

**Damper** is the resistance encountered by the **Spring** force. The lower this is, the springier the object will be. As the **Damper** is increased, the amount of bounciness caused by the Joint will be reduced.

#### Min & Max Distance

If the position of your object falls in-between the **Min** & **Max Distances**, then the Joint will not be applied to your object. The position must be moved outside of these values for the Joint to activate.

### Hints

- You do not need to assign a **Connected Body** to your Joint for it to work.
- Set the ideal positions of your Jointed objects in the Editor prior to entering Play mode.
- Spring Joints require your object to have a Rigidbody attached.

## iOS

iOS physics optimization hints can be found here.

Page last updated: 2011-01-12

---

# RandomNumbers

Randomly chosen items or values are important in many games. This sections shows how you can use Unity's built-in random functions to implement some common game mechanics.

### Choosing a Random Item from an Array

Picking an array element at random boils down to choosing a random integer between zero and the array's maximum index value (which is equal to the length of the array minus one). This is easily done using the built-in Random.Range function:-

```
var element = myArray[Random.Range(0, myArray.Length)];
```

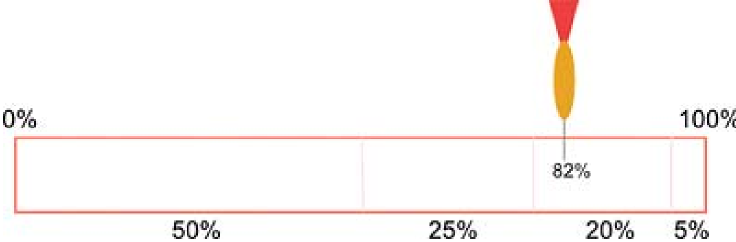
Note that Random.Range returns a value from a range that includes the first parameter but excludes the second, so using myArray.Length here gives the correct result.

### Choosing Items with Different Probabilities

Sometimes, you need to choose items at random but with some items more likely to be chosen than others. For example, an NPC may react in several different ways when it encounters a player:-

- 50% chance of friendly greeting
- 25% chance of running away
- 20% chance of immediate attack
- 5% chance of offering money as a gift

You can visualise these different outcomes as a paper strip divided into sections each of which occupies a fraction of the strip's total length. The fraction occupied is equal to the probability of that outcome being chosen. Making the choice is equivalent to picking a random point along the strip's length (say by throwing a dart) and then seeing which section it is in.




In the script, the paper strip is actually an array of floats that contain the different probabilities for the items in order. The random point is obtained by multiplying Random.value by the total of all the floats in the array (they need not add up to 1; the significant thing is the relative size of the different values). To find which array element the point is "in", firstly check to see if it is less than the value in the first element. If so, then the first element is the one selected. Otherwise, subtract the first element's value from the point value and compare that to the second element and so on until the correct element is found. In code, this would look something like the following:-

```
function Choose(probs: float[]) {
	var total = 0;

	for (elem in probs) {
		total += elem;
	}

	var randomPoint = Random.value * total;

	for (i = 0; i < probs.Length; i++) {
		if (randomPoint < probs[i])
			return i;
		else
			randomPoint -= probs[i];
	}

	return probs.Length - 1;
}
```

Note that the final return statement is necessary because Random.value can return a result of 1. In this case, the search will not find the random point anywhere. Changing the line

```
if (randomPoint < probs[i])
```

...to a less-than-or-equal test would avoid the extra return statement but would also allow an item to be chosen occasionally even when its probability is zero.

### Shuffling a List

A common game mechanic is to choose from a known set of items but have them arrive in random order. For example, a deck

of cards is typically shuffled so they are not drawn in a predictable sequence. You can shuffle the items in an array by visiting each element and swapping it with another element at a random index in the array:-

```
function Shuffle(deck: int[]) {
	for (i = 0; i < deck.Length; i++) {
		var temp = deck[i];
		var randomIndex = Random.Range(0, deck.Length);
		deck[i] = deck[randomIndex];
		deck[randomIndex] = temp;
	}
}
```

### Choosing from a Set of Items Without Repetition

A common task is to pick a number of items randomly from a set without picking the same one more than once. For example, you may want to generate a number of NPCs at random spawn points but be sure that only one NPC gets generated at each point. This can be done by iterating through the items in sequence, making a random decision for each as to whether or not it gets added to the chosen set. As each item is visited, the probability of its being chosen is equal to the number of items still needed divided by the number still left to choose from.

As an example, suppose that ten spawn points are available but only five must be chosen. The probability of the first item being chosen will be 5 / 10 or 0.5. If it is chosen then the probability for the second item will be 4 / 9 or 0.44 (ie, four items still needed, nine left to choose from). However, if the first was not chosen then the probability for the second will be 5 / 9 or 0.56 (ie, five still needed, nine left to choose from). This continues until the set contains the five items required. You could accomplish this in code as follows:-

```
var spawnPoints: Transform[];

function ChooseSet(numRequired: int) {
	var result = new Transform[numRequired];

	var numToChoose = numRequired;

	for (numLeft = spawnPoints.Length; numLeft > 0; numLeft--) {
		// Adding 0.0 is simply to cast the integers to float for the division.
		var prob = numToChoose + 0.0 / numLeft + 0.0;

		if (Random.value <= prob) {
			numToChoose--;
			result[numToChoose] = spawnPoints[numLeft - 1];

			if (numToChoose == 0)
				break;
		}
	}

	return result;
}
```

Note that although the selection is random, items in the chosen set will be in the same order they had in the original array. If the items are to be used one at a time in sequence then the ordering can make them partly predictable, so it may be necessary to shuffle the array before use.

### Random Points in Space

A random point in a cubic volume can be chosen by setting each component of a Vector3 to a value returned by Random.value:-

```
var randVec = Vector3(Random.value, Random.value, Random.value);
```

This gives a point inside a cube with sides one unit long. The cube can be scaled simply by multiplying the X, Y and Z

components of the vector by the desired side lengths. If one of the axes is set to zero, the point will always lie within a single plane. For example, picking a random point on the "ground" is usually a matter of setting the X and Z components randomly and setting the Y component to zero.

When the volume is a sphere (ie, when you want a random point within a given radius from a point of origin), you can use Random.insideUnitSphere multiplied by the desired radius:-

```
var randWithinRadius = Random.insideUnitSphere * radius;
```

Note that if you set one of the resulting vector's components to zero, you will *not* get a correct random point within a circle. Although the point is indeed random and lies within the right radius, the probability is heavily biased toward the edge of the circle and so points will be spread very unevenly. You should use Random.insideUnitCircle for this task instead:-

```
var randWithinCircle = Random.insideUnitCircle * radius;
```

Page last updated: 2011-09-12

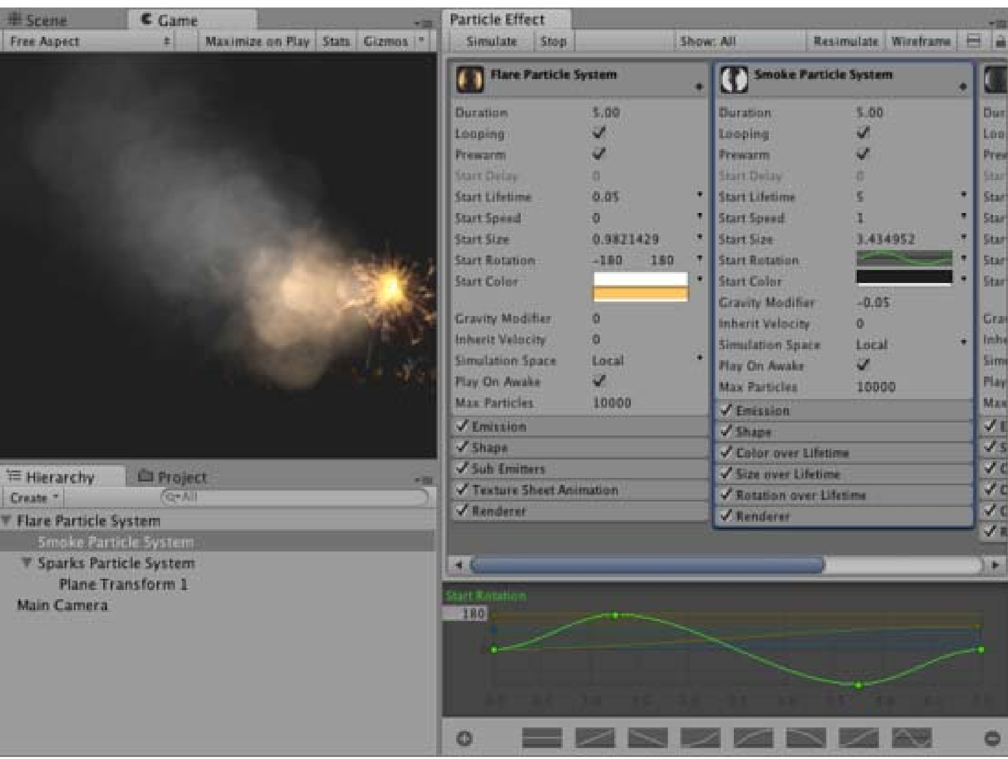
---

# Particle Systems

Note: This is the documentation for the new particle system (Shuriken). For documentation on the legacy particle system go to Legacy Particle System.

# Particle System (Shuriken)

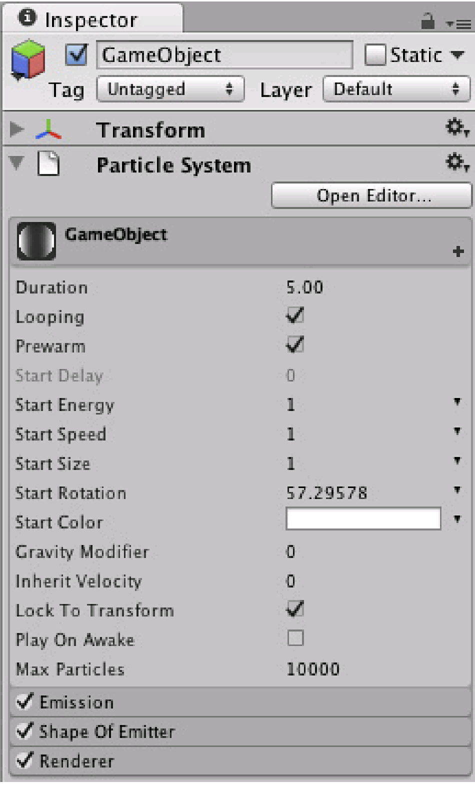
Particle Systems in Unity are used to make clouds of smoke, steam, fire and other atmospheric effects.




You can create a new particle system by creating a **Particle System** GameObject (menu **GameObject** -> **Create Other** -> **Particle System**) or by creating an empty **GameObject** and adding the **ParticleSystem** component to it (in **Component**->**Effects**)

### The Particle System Inspector (Shuriken)

The **Particle System Inspector** shows one particle system at a time (the currently selected one), and it looks like this:



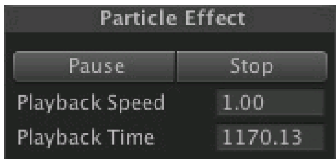

Individual particle systems can take on various complex behaviors by using Modules.

They can also be extended by being grouped together into **Particle Effects**.

If you press the button **Open Editor ...**, this will open up the Extended **Particle Editor**, that shows all of the particle systems under the same root in the scene tree. For more information on particle system grouping, see the section on Particle Effects.

### Scene View Editing

When creating and editing Particle Systems, you either work with the **Inspector** or the extended **Particle Editor**, and your changes are reflected in the **SceneView**. The scene view has a **Preview Panel**, where playback of the currently selected **Particle Effect** can be controlled in Edit Mode, with actions like **play**, **pause**, **stop** and **scrubbing playback time**




Scrubbing play back time can be performed by dragging on the label *Playback Time.* All Playback controls have shortcut keys which can be customized in the Preferences window

## Particle System Curve Editor

### MinMax curves

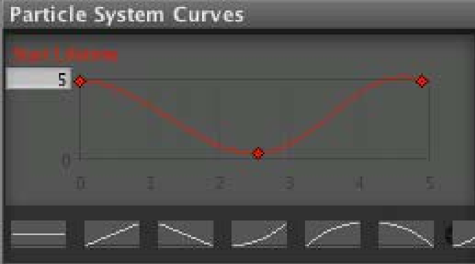
Many of the properties in the particle system modules describe a change of a value with time. That change is described via **MinMax Curves**. These time-animated properties (for example **size** and **speed**), will have a pull down menu on the right hand side, where you can choose between:

Constant
Random Between Two Constants
✓ Curve
Random Between Two Curves

**Constant**: The value of the property will not change with time, and will not be displayed in the **Curve Editor**

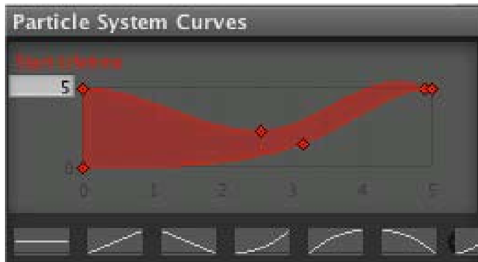
**Random between constants**: The value of the property will be selected at random between the two constants

**Curve**: The value of the property will change with time based on the curve specified in the **Curve Editor**



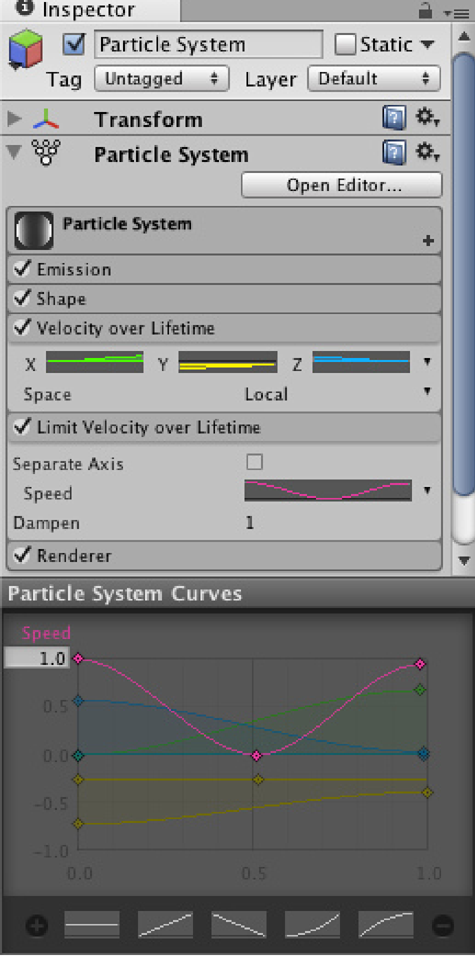
A property animated with a Curve

**Random between curves**: A curve will be generated at random between the min and the max curve, and the value of the property will change in time based on the generated curve



A property animated as **Random Between Two Curves**

In the **Curve Editor**, the *x*-axis spans time between 0 and the value specified by the **Duration** property, and the *y*-axis represents the value of the animated property at each point in time. The range of the *y*-axis can be adjusted in the number field in the upper right corner of the **Curve Editor**. The **Curve Editor** currently displays all of the curves for a particle system in the same window.



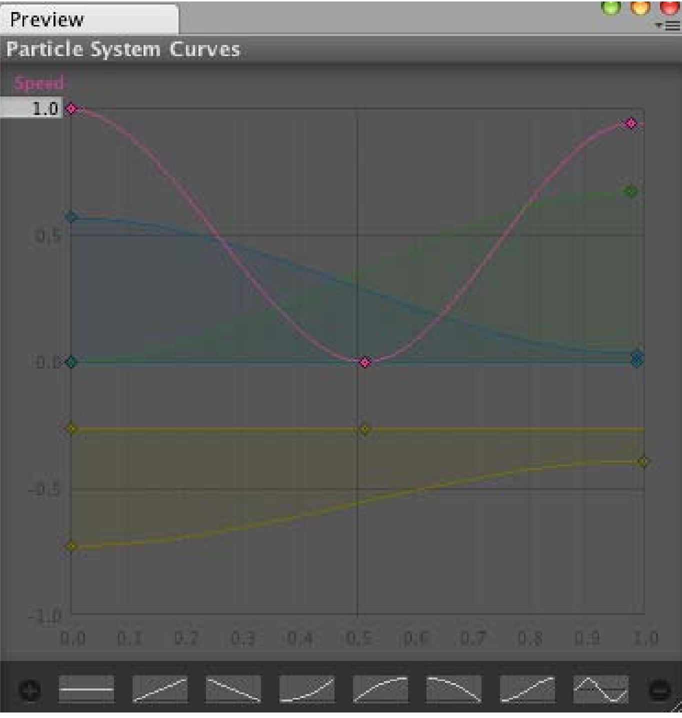

Multiple curves in the same curve editor

Note that the "-" in the bottom-right corner will remove the currently selected curve, while the "+" will *optimize* it (that is make it into a parametrized curve with at most 3 keys).

For animating properties that describe vectors in 3D space, we use the TripleMinMax Curves, which are simply curves for the x-,y-, and z- dimensions side by side, and it looks like this:

### Managing many curves in the curve editor

To avoid cluttering in the **Curve Editor**, it is possible to toggle curves on and off, by clicking on them in the inspector. The Particle System Curve Editor can also be detached from the Inspector by right-clicking on the **Particle System Curves** title bar, after which you should see something like this:



A detached Curve Editor that can be docked like any other window

For more information on working with curves, take a look at the Curve Editor documentation

## Colors and Gradients in the Particle System (Shuriken)

For properties that deal with color, the **Particle System** makes use of the **Color and Gradient Editor**. It works in a similar way to the Curve Editor.

The color-based properties will have a pull down menu on the right hand side, where you can choose between:

✓ Color
Gradient
Random Between Two Colors
Random Between Two Gradients

**Color**: The color will be the same throughout time (see Color Picker )

**Gradient**: The gradient (RGBA) will vary throughout time, edited in the Gradient Editor

**Random Between Two Colors**: The color varies with time and is chosen at random between two values specified in the Color Picker

**Random Between Two Gradients**: The gradient (RGBA) varies with time and is chosen at random between two values specified Gradient Editor

- Particle System Curve Editor
- Colors and Gradients in the Particle System (Shuriken)
- Gradient Editor
- Particle System Inspector
- Introduction to Particle System Modules (Shuriken)
- Particle System Modules (Shuriken)
- Particle Effects (Shuriken)

Page last updated: 2012-01-13

## Particle System Curve Editor

### MinMax curves

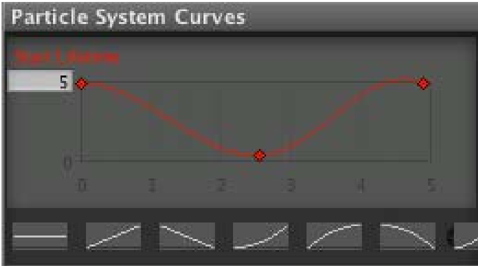
Many of the properties in the particle system modules describe a change of a value with time. That change is described via **MinMax Curves**. These time-animated properties (for example **size** and **speed**), will have a pull down menu on the right hand side, where you can choose between:

Constant
Random Between Two Constants
✓ Curve
Random Between Two Curves

**Constant**: The value of the property will not change with time, and will not be displayed in the **Curve Editor**

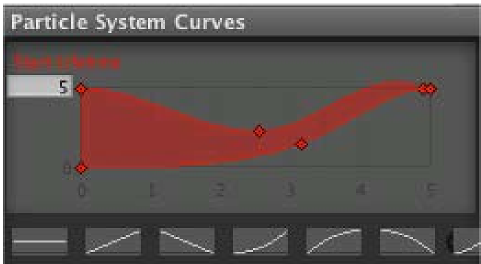
**Random between constants**: The value of the property will be selected at random between the two constants

**Curve**: The value of the property will change with time based on the curve specified in the **Curve Editor**



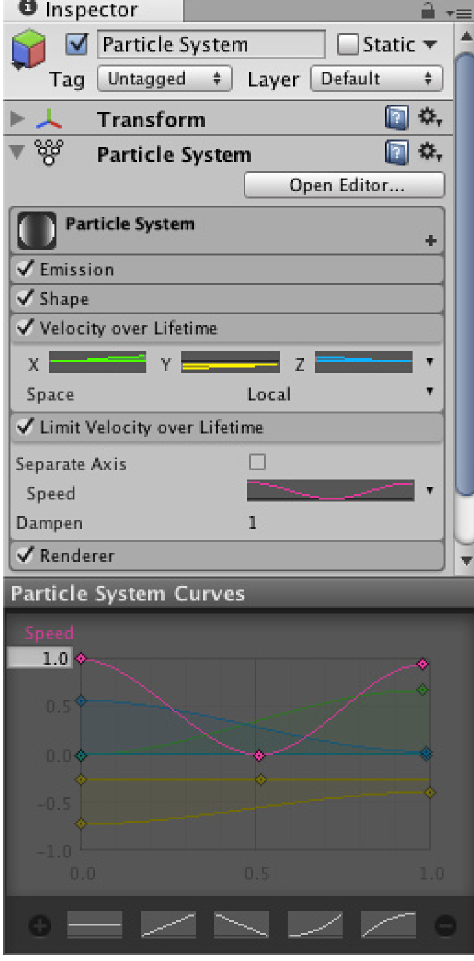
A property animated with a Curve

**Random between curves**: A curve will be generated at random between the min and the max curve, and the value of the property will change in time based on the generated curve



A property animated as **Random Between Two Curves**

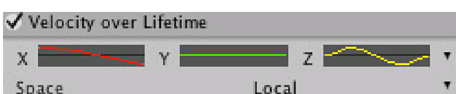
In the **Curve Editor**, the *x*-axis spans time between 0 and the value specified by the **Duration** property, and the *y*-axis represents the value of the animated property at each point in time. The range of the *y*-axis can be adjusted in the number field in the upper right corner of the **Curve Editor**. The **Curve Editor** currently displays all of the curves for a particle system in the same window.



Multiple curves in the same curve editor

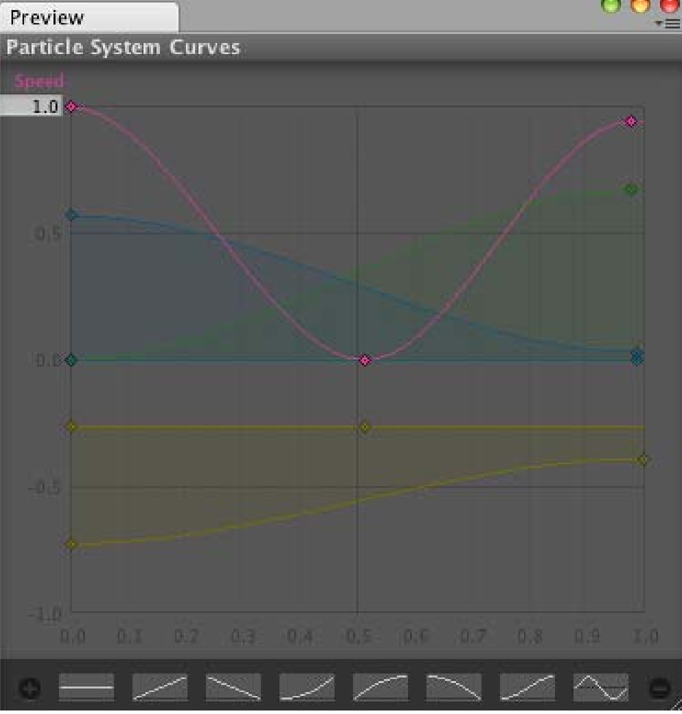
Note that the "-" in the bottom-right corner will remove the currently selected curve, while the "+" will *optimize* it (that is make it into a parametrized curve with at most 3 keys).

For animating properties that describe vectors in 3D space, we use the TripleMinMax Curves, which are simply curves for the x-,y-, and z- dimensions side by side, and it looks like this:




### Managing many curves in the curve editor

To avoid cluttering in the **Curve Editor**, it is possible to toggle curves on and off, by clicking on them in the inspector. The Particle System Curve Editor can also be detached from the Inspector by right-clicking on the **Particle System Curves** title bar, after which you should see something like this:



A detached Curve Editor that can be docked like any other window

For more information on working with curves, take a look at the Curve Editor documentation

Page last updated: 2012-01-16

---

# Particle System Color Editor

For properties that deal with color, the **Particle System** makes use of the **Color and Gradient Editor**. It works in a similar way to the Curve Editor.

The color-based properties will have a pull down menu on the right hand side, where you can choose between:

✓ Color
Gradient
Random Between Two Colors
Random Between Two Gradients

**Color**: The color will be the same throughout time (see Color Picker )

**Gradient**: The gradient (RGBA) will vary throughout time, edited in the Gradient Editor

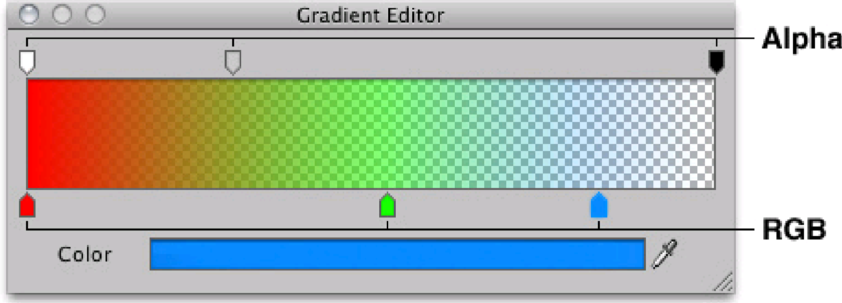
**Random Between Two Colors**: The color varies with time and is chosen at random between two values specified in the Color Picker

**Random Between Two Gradients**: The gradient (RGBA) varies with time and is chosen at random between two values specified Gradient Editor

Page last updated: 2012-01-13

---

## Particle System Gradient Editor




Gradient editor

The **Gradient Editor** is used for describing change of gradient with time. It animates the color (RGB-space, described by the markers at the bottom), and Alpha (described by the markers at the top).

You can add new *markers* for Alpha values by clicking near the top of the rectangle, and new ticks for Color by clicking near the bottom. The markers can be intuitively dragged along the timeline.

If an Alpha tick is selected, you can edit the value for that tick by dragging the alpha value.

If a Color tick is selected, the color can be modified by double clicking on the tick or clicking on the color bar.

To remove a *marker*, just drag it off the screen.

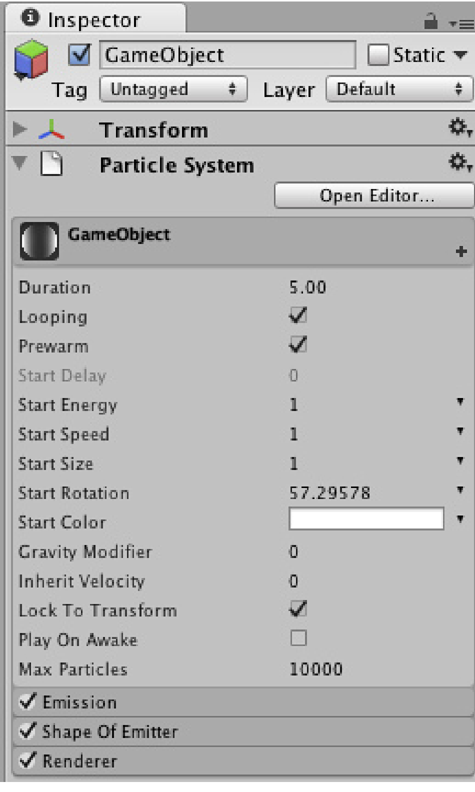
Page last updated: 2012-08-28

---

## Particle System Inspector

### The Particle System Inspector (Shuriken)

The **Particle System Inspector** shows one particle system at a time (the currently selected one), and it looks like this:




Individual particle systems can take on various complex behaviors by using Modules.

They can also be extended by being grouped together into **Particle Effects**.

If you press the button **Open Editor ...**, this will open up the Extended **Particle Editor**, that shows all of the particle systems under the same root in the scene tree. For more information on particle system grouping, see the section on Particle Effects.

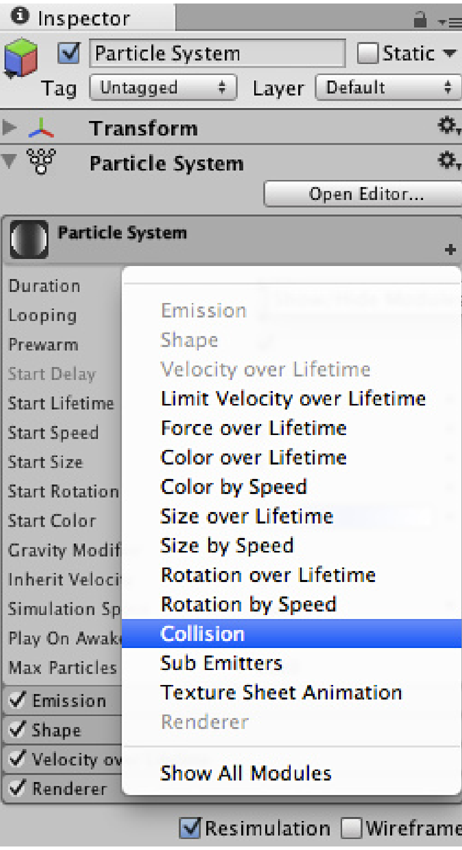
Page last updated: 2012-08-28

---

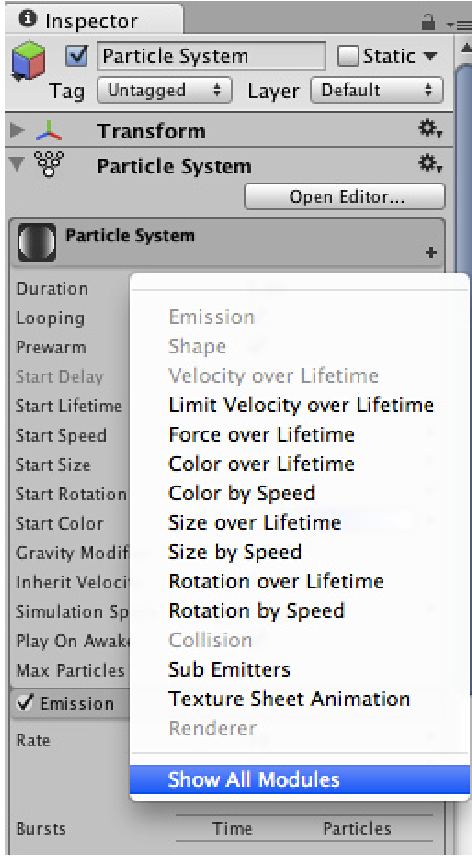
## Particle System Modules Intro

A Particle System consists of a predefined set of modules that can be enabled and disabled. These modules describe the behavior of particles in an individual particle system.

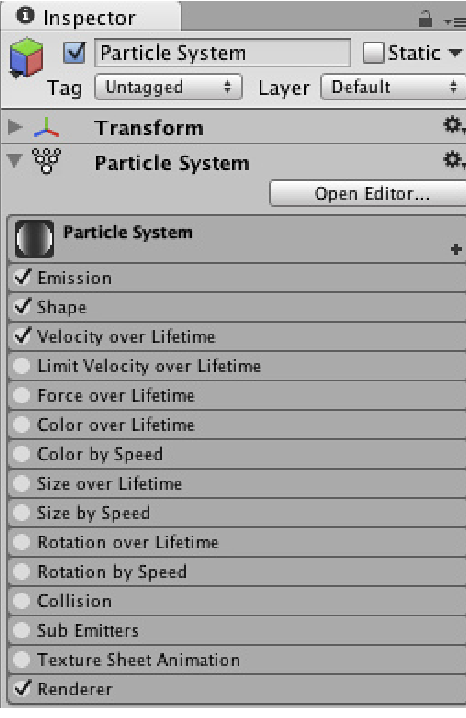
Initially only a few modules are enabled. Addding or removing modules changes the behavior of the particle system. You can add new modules by pressing the (+) sign in the top-right corner of the Particle System Inspector. This pops up a selection menu, where you can choose the module you want to enable.




An alternative way to work with modules is to select "Show All Modules", at which point all of the modules will show up in the inspector.




Then you can enable / disable modules directly from the inspector by clicking the checkbox to the left.




Most of the properties are controllable by curves (see Curve Editor). Color properties are controlled via gradients which define an animation for color (see Color Editor).

For details on individual modules and their properties, see Particle System Modules

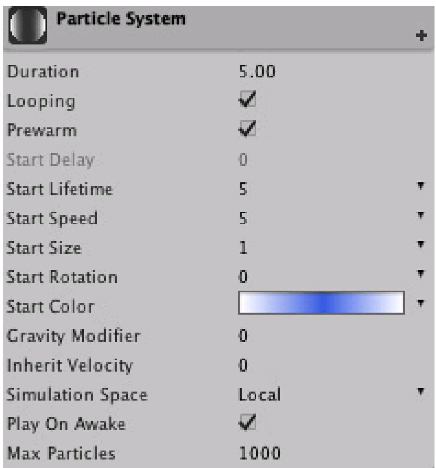
Page last updated: 2012-10-25

---

# Particle System Modules40

This page is dedicated to individual modules and their properties. For introduction to modules see this page

## Initial Module

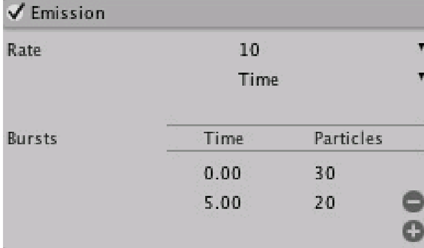



This module is always present, cannot be removed or disabled.

| | |
|---|---|
| **Duration** | The duration the Particle System will be emitting particles. |
| **Looping** | Is the Particle System looping. |
| **Prewarm** | Only looping systems can be prewarmed which means that the Particle System will have emitted particles at start as if it had already emitted particles one cycle. |
| **Start Delay** | Delay in seconds that this Particle System will wait before emitting particles. Note prewarmed looping systems cannot use a start delay. |

| | |
|---|---|
| **Start Lifetime** | The lifetime of particles in seconds (see MinMaxCurve). |
| **Start Speed** | The speed of particles when emitted.(see MinMaxCurve). |
| **Start Size** | The size of particles when emitted. (see MinMaxCurve). |
| **Start Rotation** | The rotation of particles when emitted. (see MinMaxCurve). |
| **Start Color** | The color of particles when emitted. (see MinMaxGradient). |
| **Gravity Modifier** | The amount of gravity that will affect particles during their lifetime. |
| **Inherit Velocity** | Factor for controlling the amount of velocity the particles should inherit of the transform of the Particle System (for moving Particle Systems). |
| **Simulation Space** | Simulate the Particle System in local space or world space. |
| **Play On Awake** | If enabled the Particle System will automatically start when it's created. |
| **Max Particles** | Max number of particles the Particle System will emit. |

### Emission Module




Controls the rate of particles being emitted and allows spawning large groups of particles at certain moments (over Particle System duration time). Useful for explosions when a bunch of particles need to be created at once.

| | |
|---|---|
| **Rate** | Amount of particles emitted over Time (per second) or Distance (per meter). (see MinMaxCurve) |
| **Bursts** (Time option only) | Add bursts of particles that occur within the duration of the Particle System |
| **Time and Number of Particles** | Specify time (in seconds within duration) that a specified amount of particles should be emitted. Use the + and - for adjusting number of bursts. |

### Shape Module

Defines the shape of the emitter: Sphere, Hemishpere, Cone, Box and Mesh. Can apply initial force along the surface normal or random direction.

| | |
|---|---|
| **Sphere** | |
| **Radius** | Radius of the sphere (can also be manipulated by handles in the Scene View) |
| **Emit from Shell** | Emit from shell of the sphere. If disabled, particles will be emitted from the volume of the sphere. |
| **Random Direction** | Should particles have have a random direction when emitted or a direction along the surface normal of the sphere |
| **Hemisphere** | |
| **Radius** | Radius of the hemisphere (can also be manipulated by handles in the Scene View) |
| **Emit from Shell** | Emit from shell of the hemisphere. If disabled particles will be emitted from the volume of the hemisphere. |
| **Random Direction** | Should particles have have a random direction when emitted or a direction along the surface normal of the hemisphere. |
| **Cone** | |
| **Angle** | Angle of the cone. If angle is 0 then particles will be emitted in one direction. (can also be manipulated by handles in the Scene View) |
| **Radius** | A value larger than 0 when basically create a capped cone, using this will change emission from a point to a disc.(can also be manipulated by handles in the Scene View) |
| **Emit From** | Determines where emission originates from. Possible values are Base, Base Shell, Volume and Volume Shell. |
| **Box** | |
| **Box X** | Scale of box in X (can also be manipulated by handles in the Scene View) |
| **Box Y** | Scale of box in Y (can also be manipulated by handles in the Scene View) |
| **Box Z** | Scale of box in Z (can also be manipulated by handles in the Scene View) |
| **Random Direction** | Should particles have have a random direction when emitted or a direction along the Z-axis of the box |

| | |
|---|---|
| **Mesh** | |
| **Type** | Particles can be emitted from either Vertex, Edge or Triangle |
| **Mesh** | Select Mesh that should be used as emission shape |
| **Random Direction** | Should particles have have a random direction when emitted or a direction along the surface of the mesh |

### Velocity Over Lifetime Module

Directly animates velocity of the particle. Mostly useful for particles which has complex physical, but simple visual behavior (like smoke with turbulence and temperature loss) and has little interaction with physical world.

| | |
|---|---|
| **XYZ** | Use either constant values for curves or random between curves for controlling the movement of the particles. See MinMaxCurve. |
| **Space** | Local / World: Are the velocity values in local space or world space |

### Limit Velocity Over Lifetime Module

Basically can be used to simulate drag. Dampens or clamps velocity, if it is over certain threshold. Can be configured per axis or per vector length.

| | |
|---|---|
| **Separate Axis** | Use for setting per axis control. |
| **Speed** | Specify magnitude as constant or by curve that will limit all axes of velocity. |
| **XYZ** | Control each axis seperately. See MinMaxCurve. |
| **Dampen** | (0-1) value that controls how much the exceeding velocity should be dampened. For example, a value of 0.5 will dampen exceeding velocity by 50% |

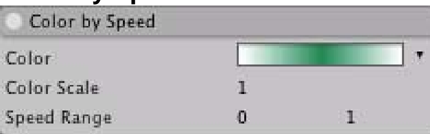
### Force Over Lifetime Module

| | |
|---|---|
| **XYZ** | Use either constant values for curves or random between curves for controlling the force applied to the particles. See MinMaxCurve. |
| **Randomize** | Randomize the force applied to the particles every frame |

### Color Over Lifetime Module

| | |
|---|---|
| **Color** | Controls the color of each particle during its lifetime. If some particles have a shorter lifetime than others, they will animate faster. Use constant color, random between two colors, animate it using gradient or specify a random color using two gradients (see Gradient). Note that this colour will be *multiplied* by the value in the **Start Color** property - if the Start Color is black then Color Over Lifetime will not affect the particle. |
| **Color Scale** | Use the color scale for easy adjustment of color or gradient. |

### Color By Speed Module



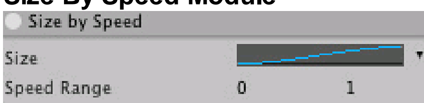

Animates particle color based on its speed. Remaps speed in the defined range to a color.

| | |
|---|---|
| **Color** | Color used for remapping of speed. Use gradients for varying colors. See MinMaxGradient. |
| **Color Scale** | Use the color scale for easy adjustment of color or gradient. |
| **Speed Range** | The min and max values for defining the speed range which is used for remapping a speed to a color. |

### Size Over Lifetime Module

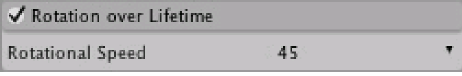
| | |
|---|---|
| **Size** | Controls the size of each particle during its lifetime. Use constant size, animate it using a curve or specify a random size using two curves. See MinMaxCurve. |

### Size By Speed Module




| | |
|---|---|
| **Size** | Size used for remapping of speed. Use curves for varying sizes. See MinMaxCurve. |
| **Speed Range** | The min and max values for defining the speed range which is used for remapping a speed to a size. |

### Rotation Over Lifetime Module




Specify values in degrees.

| | |
|---|---|
| **Rotational Speed** | Controls the rotational speed of each particle during its lifetime. Use constant rotational speed, animate it using a curve or specify a random rotational speed using two curves. See MinMaxCurve. |

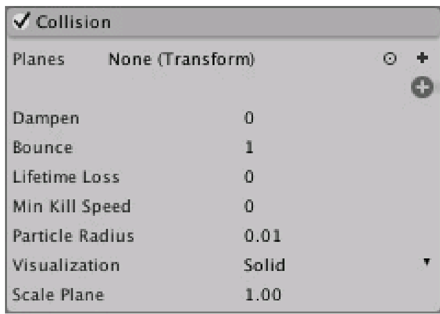
### Rotation By Speed Module

| | |
|---|---|
| **Rotational Speed** | Rotational speed used for remapping of a particle's speed. Use curves for varying rotational speeds. See MinMaxCurve. |
| **Speed Range** | The min and max values for defining the speed range which is used for remapping a speed to a rotational speed. |

### External Forces Module

| | |
|---|---|
| **Multiplier** | Scale factor that determines how much the particles are affected by wind zones (i.e., the wind force vector is multiplied by this value). |

### Collision Module




Set up collisions for the particles of this Particle System. World and planar collisions are supported. Planar collision is very efficient for simple collision detection. Planes are set up by referencing an existing transform in the scene or by creating a new empty GameObject for this purpose. Another benefit of planar collision is that particle systems with collision planes can be set up as prefabs. World collision uses raycasts so must be used with care in order to ensure good performance. However, for cases where approximate collisions are acceptable world collision in **Low** or **Medium** quality can be very efficient.

#### Properties common for any Collision Module

| | |
|---|---|
| **Planes/World** | Specify the collision type: **Planes** for planar collision or **World** for world collisions. |
| **Dampen** | (0-1) When the particle collides, it will keep this fraction of its speed. Unless it is set to 1.0, the particle will become slower after collision. |
| **Bounce** | (0-1) When the particle collides, it will keep this fraction of the component of the velocity, which is normal to the plane of collision. |
| **Lifetime Loss** | (0-1) The fraction of Start Lifetime lost on each collision. When lifetime reaches 0, the particle dies. For example if a particle should die on first collision, set this to 1.0. |

| | |
|---|---|
| **Min Kill Speed** | The minimum speed of a particle before it is killed. |

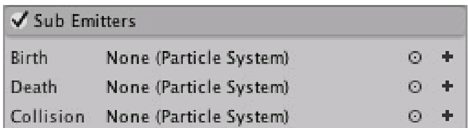
**Properties available only in the Planes Mode**

| | |
|---|---|
| **Planes** | Planes are defined by assigning a reference to a transform. This transform can be any transform in the scene and can be animated. Multiple planes can be used. Note: the Y-axis is used as the normal of a plane. |
| **Visualization** | Only used for visualizing the planes: Grid or Solid. |
| **Grid** | Rendered as gizmos and is useful for quick indication of position and orientation in the world. |
| **Solid** | Renders a plane in the scene which is useful for exact positioning of a plane. |
| **Scale Plane** | Resizes the visualization planes. |
| **Particle Radius** | The assumed radius of the particle for collision purposes. |

**Properties available only in the World Mode**

| | |
|---|---|
| **Collides With** | Filter for specifying colliders. Select **Everything** to colllide with the whole world. |
| **Collision Quality** | The quality of the world collision. |
| ***High*** | All particles performs a scene raycast per frame. Note: This is CPU intensive, it should only be used with 1000 simultaneous particles (scene wide) or less. |
| ***Medium*** | The particle system receives a share of the globally set **Particle Raycast Budget** (see Particle Raycast Budget) in each frame. Particles are updated in a round-robin fashion where particles that do not receive a raycast in a given frame will lookup and use older collisions stored in a cache. Note: This collision type is approximate and some particles will leak, particularly at corners. |
| ***Low*** | Same as **Medium** except the particle system is only awarded a share of the **Particle Raycast Budget** every fourth frame. |
| **Voxel Size** | Density of the voxels used for caching intersections used in the **Medium** and **Low** quality setting. The size of a voxel is given in scene units. Usually, 0.5 - 1.0 should be used (assuming metric units). |

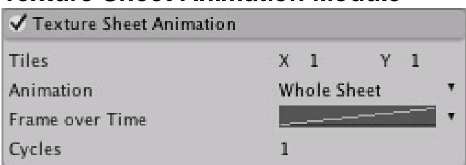
## Sub Emitter Module




This is a powerful module that enables spawning of other Particle Systems at the follwing particle events: birth, death or collision of a particle.

| | |
|---|---|
| **Birth** | Spawn another Particle System at birth of each particle in this Particle System |
| **Death** | Spawn another Particle System at death of each particle in this Particle System |
| **Collision** | Spawn another Particle System at collision of each particle in this Particle System. IMPORTANT: Collision needs to be set up using the Collision Module. See Collision Module |

## Texture Sheet Animation Module

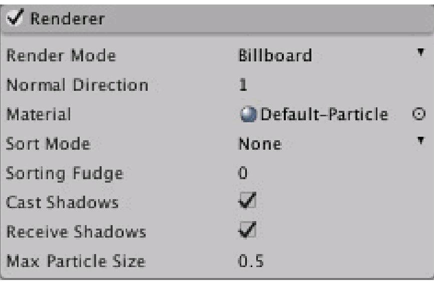



Animates UV coordinates of the particle over its lifetime. Animation frames can be presented in a form of a grid or every row in the sheet can be separate animation. The frames are animated with curves or can be a random frame between two curves. The speed of the animation is defined by "Cycles".

IMPORTANT: The texture used for animation is the one used by the material found in the Renderer module.

| | |
|---|---|
| **Tiles** | Define the tiling of the texture. |
| **Animation** | Specify the animation type: Whole Sheet or Single Row. |
| **Whole Sheet** | Uses the whole sheet for uv animation |
| **- Frame over Time** | Controls the uv animation frame of each particle during its lifetime over the whole sheet. Use constant, animate it using a curve or specify a random frame using two curves. See MinMaxCurve. |
| **Single Row** | Uses a single row of the texture sheet for uv animation |
| **- Random Row** | If checked the start row will be random and if unchecked the row index can be specified (first row is 0). |
| **- Frame over Time** | Controls the uv animation frame of each particle during its lifetime within the specified row. Use constant, animate it using a curve or specify a random frame using two curves. See MinMaxCurve. |
| **- Cycles** | Specify speed of animation. |

### Renderer Module




The renderer module exposes the **ParticleSystemRenderer** component's properties. Note that even though a GameObject has a **ParticleSystemRenderer** component, its properties are only exposed here, when this module is removed/added. It is actually the **ParticleSystemRenderer** component that is added or removed.

| | |
|---|---|
| **Render Mode** | Select one of the following particle render modes |
| **Billboard** | Makes the particles always face the camera |
| **Stretched Billboard** | Particles are stretched using the following parameters |
| **- Camera Scale** | How much the camera speed is factored in when determining particle stretching |
| **- Speed Scale** | Defines the length of the particle compared to its speed |
| **- Length Scale** | Defines the length of the particle compared to its width |
| **Horizontal Billboard** | Makes the particles align with the Y axis |
| **Vertical Billboard** | Makes the particles align with the XZ plane while facing the camera |
| **Mesh** | Particles are rendered using a mesh instead of a quad |
| **- Mesh** | The reference to the mesh used for rendering particles |
| **Normal Direction** | Value from 0 to 1 that determines how much normals point toward the camera (0) and how much sideways toward the centre of the view (1). |
| **Material** | Material used by billboarded or mesh particles. |
| **Sort Mode** | The draw order of particles can be sorted by distance, youngest first, or oldest first. |
| **Sorting Fudge** | Use this to affect the draw order. Particle systems with *lower* sorting fudge numbers are more likely to be drawn last, and thus appear in front of other transparent objects, including other particles. |
| **Cast Shadows** | Should particles cast shadows? May or may not be possible depending on the material |
| **Receive Shadows** | Should particles receive shadows? May or may not be possible depending on the material |
| **Max Particle Size** | Set max relative viewport size. Valid values: 0-1 |

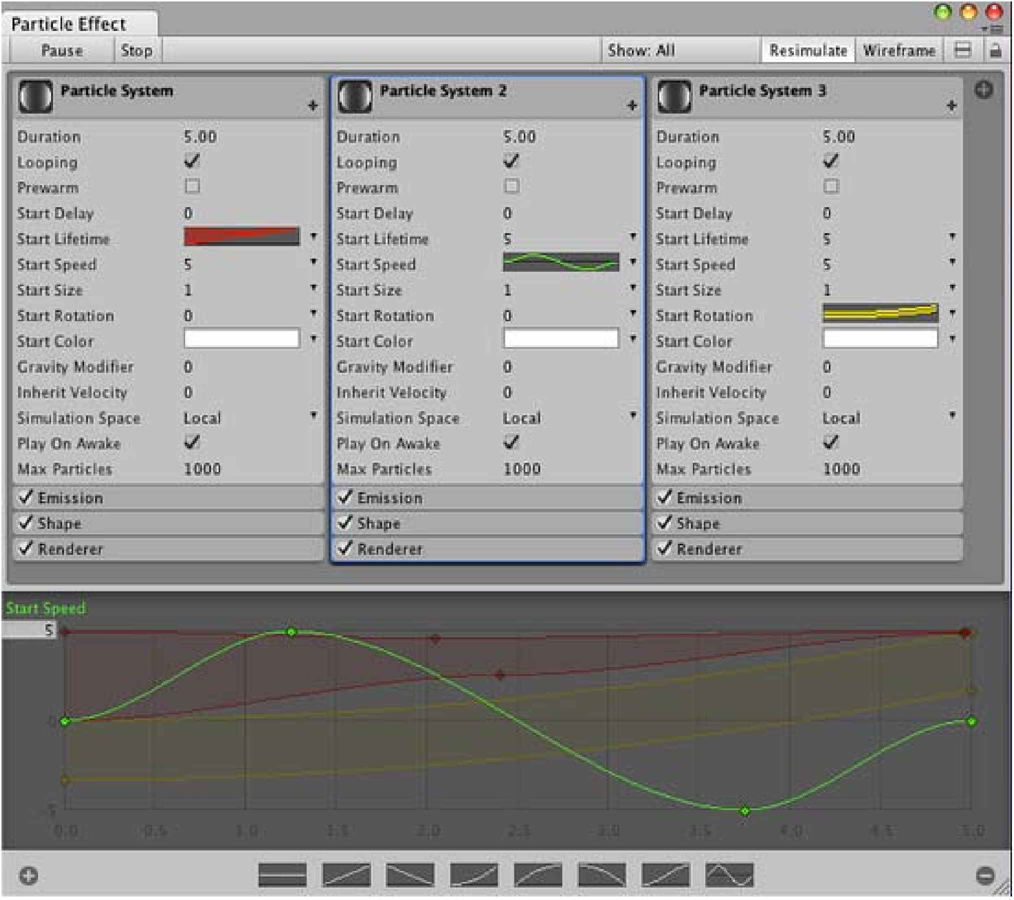
Page last updated: 2012-11-09

---

# Particle System Grouping

An important feature of Unity's Particle System is that individual Particle Systems can be grouped by being parented to the same root. We will use the term **Paricle Effect** for such a group. Particle Systems belonging to the same Particle Effect, are played, stopped and paused together.

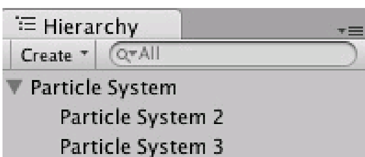
For managing complex particle effects, Unity provides a Particle Editor, which can be accessed from the Inspector, by pressing **Open Editor**



*Overview of the **Particle System Editor***

You can toggle between **Show: All** and **Show: Selected** in this Editor. **Show: All** will render the entire particle effect. **Show: Selected** will only render the selected particle systems. What is selected will be highlighted with a blue frame in the Particle Editor and also shown in blue in the Hierarchy view. You can also change the selection both from the Hierarchy View and the Particle Editor, by clicking the icon in the top-left corner of the Particle System. To do a multiselect, use Ctrl+click on windows and Command+click on the Mac.

You can explicitly control rendering order of grouped particles (or otherwise spatially close particle emitters) by tweaking **Sorting Fudge** property in the Renderer module.



*Particle Systems in the same hierarchy are considered as part of the same Particle Effect. This hierarchy shows the setup of the effect shown above.*

Page last updated: 2012-08-28

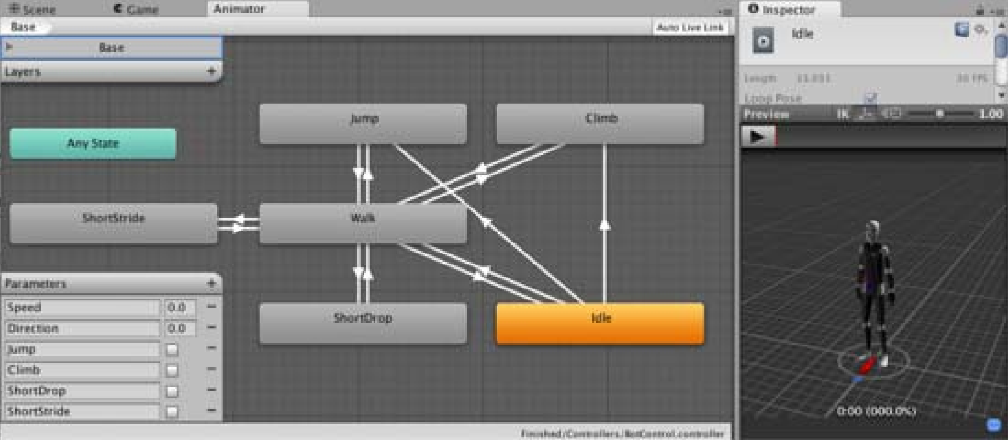
---

## Mecanim Animation System

Unity has a rich and sophisticated animation system called **Mecanim**. Mecanim provides:

- Easy workflow and setup of animations on humanoid characters.
- Animation retargeting - the ability to apply animations from one character model onto another.
- Simplified workflow for aligning animation clips.
- Convenient preview of animation clips, transitions and interactions between them. This allows animators to work more

independently of programmers, prototype and preview their animations before gameplay code is hooked in.
- Management of complex interactions between animations with a visual programming tool.
- Animating different body parts with different logic.



*Typical setup in the Visual Programming Tool and the Animation Preview window*

### Mecanim workflow

Workflow in Mecanim can be split into three major stages.

1. **Asset preparation and import**. This is done by artists or animators, with 3rd party tools, such as Max or Maya. This step is independent of Mecanim features.
2. Character setup for Mecanim, which can be done in 2 ways:

```
a. Humanoid character setup. Mecanim has a special workflow for humanoid models, with exte
b. Generic character setup. This is for anything like creatures, animated props, four-legge
```

3. **Bringing characters to life**. This involves setting up animation clips, as well as interactions between them, and involves setup of State Machines and Blend Trees, exposing Animation Parameters, and controlling animations from code.

Mecanim comes with a lot of new concepts and terminology. If at any point, you need to find out what something means, go to our Animation Glossary.


- A Glossary of Animation and Mecanim terms
- Asset Preparation and Import
  - Preparing your own character
  - Importing Animations
  - Splitting Animations
- Working with humanoid animations
  - Creating the Avatar
  - Configuring the Avatar
  - Muscle setup
  - Avatar Body Mask
  - Retargeting of Humanoid animations
  - Inverse Kinematics (Pro only)
- Generic Animations in Mecanim
- Bringing Characters to Life
  - Looping animation clips
  - Animator Component and Animator Controller
  - Animation State Machines
    - Animation States
    - Animation Transitions
    - Animation Parameters
  - Blend Trees
  - Mecanim Advanced topics
    - Working with Animation Curves in Mecanim (Pro only)
    - Sub-State Machines
    - Animation Layers

- Animation State Machine Preview (solo and mute)
- Target Matching
- Root Motion - how it works
  - Tutorial: Scripting Root Motion for "in-place" humanoid animations

### Legacy animation system

While Mecanim is recommended for use in most situations, especially for working humanoid animations, the Legacy animation system is still used in a variety of contexts. One of them is working legacy animations and code (content created before Unity 4.0). Another is controlling animation clips with parameters other than time (for example for controlling the aiming angle). For information on the Legacy animation system, see this section

Unity intends to phase out the Legacy animation system over time for all cases by merging the workflows into Mecanim.

Page last updated: 2012-11-08

---

# A glossary of animation and Mecanim terms

| Icon | Term | Description | Type of Concept | Usage/Comments |
|---|---|---|---|---|
| **Animation Clip related terms** | | | | |
| | Animation Clip | Animation data that can be used for animated characters or simple animations. It is a simple "unit" piece of motion, such as (one specific instance of) "Idle", "Walk" or "Run" | sub-Asset | |
| | Body Mask | A specification for which body parts to include or exclude for a skeleton | Asset (`.mask`) | Used in Animation Layers and in the importer |
| | Animation Curves | Curves can be attached to animation clips and controlled by various parameters from the game | | |
| **Avatar related terms** | | | | |
| | Avatar | An interface for retargeting one skeleton to another | sub-Asset | |
| | Retargeting | Applying animations created for one model to another | Process | |
| | Rigging | The prcoess of building a skeleton hierarchy of bone joints for your mesh | Process | done in an external tool, such as Max or Maya |
| | Skinning | The process of binding bone joints to the character's mesh or 'skin' | Process | done in an external tool, such as Max or Maya |
| | Muscle Definition | A Mecanim concept, which allows you to have a more intuitive control over the character's skeleton. When an Avatar is in place, Mecanim works in muscle space, which is more intuitive than bone space | | |
| | T-pose | The pose in which the character has his arms straight out to the sides, forming a "T". The required pose for the character to be in, in order to make an Avatar | | |
| | Bind-pose | The pose at which the character was modelled | | |
| | Human template | A pre-defined bone-mapping | Asset (`.ht`) | Used for matching bones from FBX files to the Avatar. |
| **Animator and Animator Controller related terms** | | | | |
| | Animator Component | Component on a model that animates that model using the Mecanim animation system. The component has a reference to an Animator Controller asset that controls the animation. | Component | |
| | Root Motion | Motion of character's root, whether it's controlled by the animation itself or externally. | | |
| | Animator Controller (Asset) | The Animator Controller controls animation through Animation Layers with Animation State Machines and Animation Blend Trees, controlled by Animation Parameters. The same Animator Controller can be | Asset (`.controller`) | |

| | | | | |
|---|---|---|---|---|
| | | referenced by multiple models with Animator components. | | |
| | Animator Controller (Window) | The window where the Animator Controller Asset is visualized and edited. | Window | |
| | Animation Layer | An Animation Layer contains an Animation State Machine that controls animations of a model or part of it. An example of this is if you have a full-body layer for walking / jumping and a higher layer for upper-body motions such as throwing object / shooting. The higher layers take precedence for the body parts they control. | | |
| | Animation State Machine | A graph controlling the interaction of Animation States. Each state references an Animation Blend Tree or a single Animation Clip. | | |
| | Animation Blend Tree | Used for continuous blending between similar Animation Clips based on float Animation Parameters. | | |
| | Animation Parameters | Used to communicate between scripting and the Animator Controller. Some parameters can be set in scripting and used by the controller, while other parameters are based on Custom Curves in Animation Clips and can be sampled using the scripting API. | | |
| | Inverse Kinematics (IK) | The ability to control the character's body parts based on various objects in the world. | | |
| **Non-Mecanim animation terms** | | | | |
| | Animation Component | The component needed for non-Mecanim animations | Component | |

Page last updated: 2012-11-07

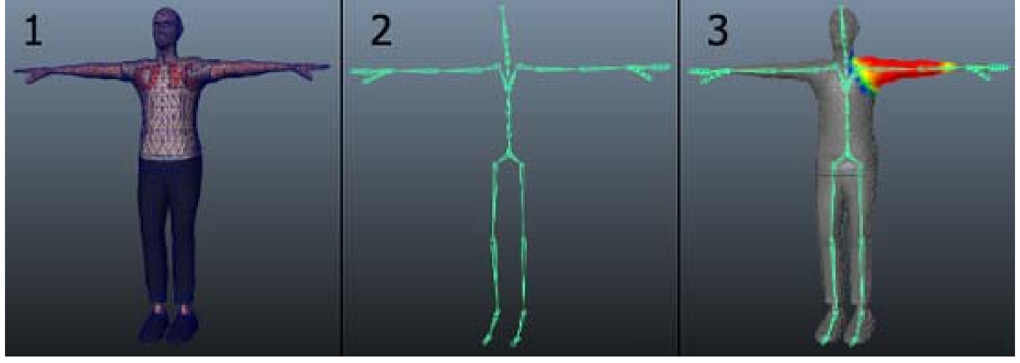
---

# Asset Preparation and Import

### Humanoid meshes

In order to take full advantage of Mecanim's humanoid animation system and retargeting, you need to have a **rigged** and **skinned** humanoid type mesh.

1. A character **model** is generally made up of polygons in a 3D package or converted to polygon or triangulated mesh, from a more complex mesh type before export.
2. A **joint hierarchy** or **skeleton** which defines the bones inside the mesh and their movement in relation to one another, must be created to control the movement of the character. The process for creating the joint hierarchy is known as **rigging**.
3. The mesh or *skin* must then be connected to the joint hierarchy in order to define which parts of the character mesh move when a given joint is animated. The process of connecting the skeleton to the mesh is known as **skinning**.



Stages for preparing a character (modeling, rigging, and skinning)

### How to obtain humanoid models

There are three main ways to obtain humanoid models for with the Mecanim Animation system:

1. Use a procedural character system or character generator such as *Poser*, *Makehuman* or *Mixamo*. Some of these systems will rig and skin your mesh (eg, Mixamo) while others will not. Furthermore, these methods may require that you reduce the number of polygons in your original mesh to make it suitable for use in Unity.
2. Purchase demo examples and character content from the Unity Asset Store.
3. Also, you can of course prepare your own character from scratch.

### Export & Verify

Unity imports a number of different generic and native 3D file formats. The format we recommend for exporting and verifying your model is FBX 2012 since it will allow you to:

- Export the mesh with the skeleton hierarchy, normals, textures and animation
- Re-import into your 3D package to verify your animated model has exported as you expected
- Export animations without meshes

### Further details

The following pages cover the stages of preparing and importing animation assets in greater depth

- Preparing your own character
- Importing Animations
- Splitting Animations

(back to Mecanim introduction)

Page last updated: 2012-11-01

---

## Preparing your own character

There are three main steps in creating an animated humanoid character from scratch: **modelling**, **rigging** and **skinning**.

### Modelling

This is the process of creating your own humanoid mesh in a 3D modelling package - 3DSMax, Maya, Blender, etc. Although this is a whole subject in its own right, there are a few guidelines you can follow to ensure a model works well with animation in a Unity project.

- Observe a **sensible topology**. The exact nature of a "sensible" structure for your mesh is rather subtle but generally, you should bear in mind how the vertices and triangles of the model will be distorted as it is animated. A poor topology will not allow the model to move without unsightly distortion of the mesh. A lot can be learned by studying existing 3D character meshes to see how the topology is arranged and why.
- Be mindful of the **scale** of your mesh. Do a test import and compare the size of your imported model with a "meter cube" (the standard Unity cube primitive has a side length of one unit, so it can be taken as a 1m cube for most purposes). Check the units your 3D package is using and adjust the export settings so that the size of the model is in correct proportion to the cube. Unless you are careful, it is easy to create models without any notion of their scale and consequently end up with a set of objects that are disproportionate in size when they are imported into Unity.
- Arrange the mesh so that the character's feet are standing on the local origin or "anchor point" of the model. Since a character typically walks upright on a floor, it is much easier to handle if its anchor point (ie, its transform position) is directly on that floor.
- Model in a **T-pose** if you can. This will help allow space to refine polygon detail where you need it (e.g. underarms). This will also make it easier to position your rig inside the mesh.
- **Clean up your model**. Where possible, cap holes, weld verts and remove hidden faces, this will help with skinning, especially automated skinning processes.

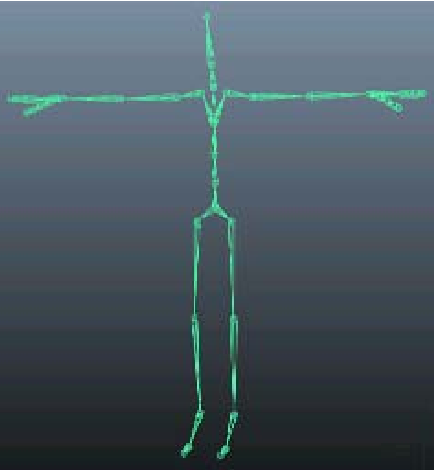
Skin Mesh - Modelled, textured and triangulated

### Rigging

This is the process of creating a skeleton of joints to control the movements of your model.

3D packages provide a number of ways to create joints for your humanoid rig. These range from ready-made biped skeletons that you can scale to fit your mesh, right through to tools for individual bone creation and parenting to create your own bone structure. Although the details are outside the scope of Unity, here are some general guidelines:

- Study existing humanoid skeletons hierarchies (eg, bipeds) and where possible use or mimic the bone structure.
- Make sure the hips are the parent bone for your skeleton hierarchy.
- A minimum of fifteen bones are required in the skeleton.
- The joint/bone hierachy should follow a natural structure for the character you are creating. Given that arms and legs come in pairs, you should use a consistent convention for naming them (eg, "arm_L" for the left arm, "arm_R" for the right arm, etc). Possible hierarchies include:
    - HIPS - spine - chest - shoulders - arm - forearm - hand
    - HIPS - spine - chest - neck - head
    - HIPS - UpLeg - Leg - foot - toe - toe_end



Biped Skeleton, positioned in T-pose

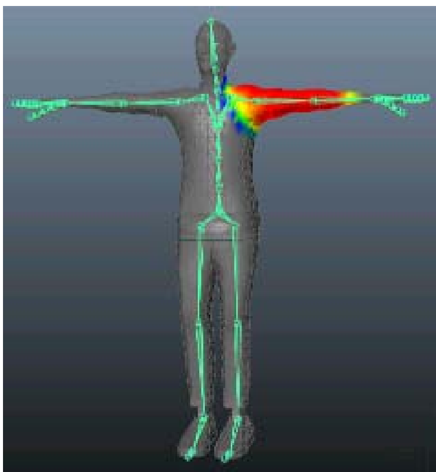
### Skinning

This is the process of attaching the mesh to the skeleton

Skinning involves binding vertices in your mesh to bones, either directly (rigid bind) or with blended influence to a number of bones (soft bind). Different software packages use different methods, eg, assigning individual vertices and painting the weighting of influence per bone onto the mesh. The initial setup is typically automated, say by finding the nearest influence or using "heatmaps". Skinning usually requires a fair amount of work and testing with animations in order to ensure satisfactory results for the skin deformation. Some general guidelines for this process include:

- Using an automated process initially to set up some of the skinning (see relevant tutorials on 3DMax, Maya, etc.)

- Creating a simple animation for your rig or importing some animation data to act as a test for the skinning. This should give you a quick way to evaluate whether or not the skinning looks good in motion.
- Incrementally editing and refining your skinning solution.
- Sticking to a maximum of four influences when using a soft bind, since this is the maximum number that Unity will handle. If more than four influences affect part of the mesh then at least some information will be lost when playing the animation in Unity.


Interactive Skin Bind, one of many skinning methods

(back to AssetPreparationandImport)

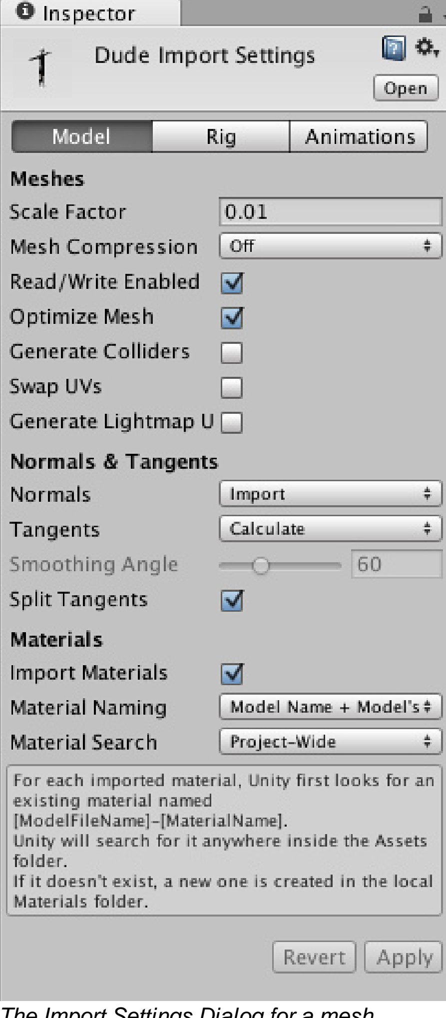
(back to Mecanim introduction)

Page last updated: 2012-11-01

---

## Importing Animations

Before a character model can be used, it must first be imported into your project. Unity can import native Maya (.mb or .ma) and Cinema 4D (.c4d) files, and also generic FBX files which can be exported from most animation packages (see this page for further details on exporting). To import an animation, simply drag the model file to the **Assets** folder of your project. When you select the file in the Project View you can edit the **Import Settings** in the inspector:-




*The Import Settings Dialog for a mesh*

See the FBX importer page for a full description of the available import options.

Splitting animations

(back to Mecanim introduction)

Page last updated: 2012-11-01

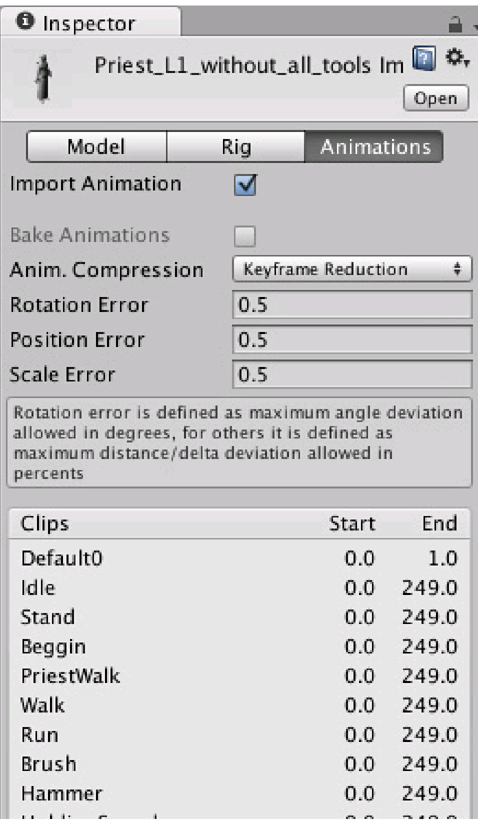
---

# Splitting animations

An animated character typically has a number of different movements that are activated in the game in different circumstances. These movements are called **Animation Clips**. For example, we might have separate animation clips for walking, running, jumping, throwing, dying, etc. Depending on the way the model was animated, these separate movements might be imported as distinct animation clips or as one single clip where each movement simply follows on from the previous one. In cases where there is only a single clip, the clip must be **split** into its component animation clips within Unity, which will involve some extra steps in your workflow.

### Working with models that have pre-split animations

The simplest types of models to work with are those that contain pre-split animations. If you have an animation like that, the Animations tab in the Animation Importer Inspector will look like this:




You will see a list available clips which you can preview by pressing Play in the Preview Window (lower down in the inspector). The frame ranges of the clips can be edited, if needed.

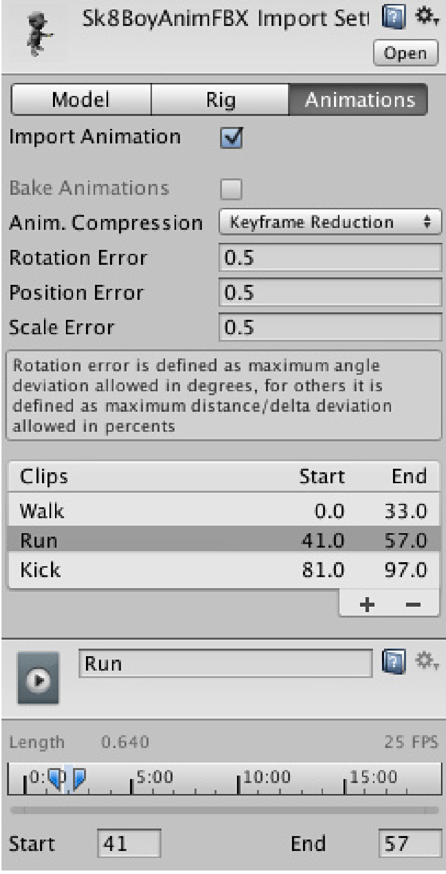
### Working with models that have unsplit animations

For models where the clips are supplied as one continuous animation, the Animation tab in the Animation Importer Inspector will look like this:

4In cases like this, you can define the frame ranges that correspond to each of the separate animation sequences (walking, jumping, etc). You can create a new animation clip by pressing (+) and selecting the range of frames that are included in it.

For example:

- walk animation during frames 1 - 33
- run animation during frames 41 - 57
- kick animation during frames 81 - 97




*The Import Settings Options for Animation*

In the Import Settings, the **Split Animations** table is where you tell Unity which frames in your asset file make up which Animation Clip. The names you specify here are used to activate them in your game.

For further information about the animation inspector, see the Animation Clip component reference page.

## Adding animations to models that do not contain them

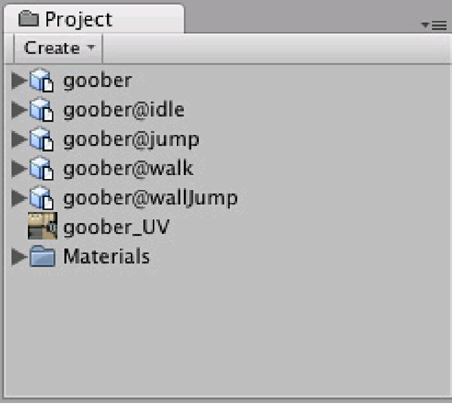
You can add animation clips to an **Animation** component even for models without muscle definitions (ie, non-Mecanim). You need to specify the default animation clip in the **Animation** property, and the available animation clips in the **Animations** property. The animation clips you add to such a non-Mecanim model should also be setup in a non-Mecanim way (ie, the **Muscle Definition** property should be set to **None**)

For models that have muscle definitions (Mecanim), the process is different:-

- Create a **New Animator Controller**
- Open the Animator Controller Window
- Drag the desired animation clip into the Animator Controller Window
- Drag the model asset into the Hierarchy.
- Add the animator controller to the **Animator component** of the asset.

### Importing Animations using multiple model files

Another way to import animations is to follow a naming scheme that Unity allows for the animation files. You create separate model files and name them with the convention 'modelName@animationName.fbx'. For example, for a model called "goober", you could import separate idle, walk, jump and walljump animations using files named "goober@idle.fbx", "goober@walk.fbx", "goober@jump.fbx" and "goober@walljump.fbx". Only the animation data from these files will be used, even if the original files are exported with mesh data.




*An example of four animation files for an animated character (note that the .fbx suffix is not shown within Unity)*

Unity automatically imports all four files and collects all animations to the file without the @ sign in. In the example above, the goober.mb file will be set up to reference idle, jump, walk and wallJump automatically.

For FBX files, simply export a model file with no animation ticked (eg, goober.fbx) and the 4 clips as goober@*animname*.fbx by exporting the desired keyframes for each (enable animation in the FBX dialog).

(back to Mecanim introduction)

Page last updated: 2012-11-12

---

## Avatar Creation and Setup

The Mecanim Animation System is particularly well suited for working with animations for humanoid skeletons. Since humanoid skeletons are a very common special case and are used extensively in games, Unity provides a specialized workflow, and an extended tool set for humanoid animations.

Because of the similarity in bone structure, it is possible to map animations from one humanoid skeleton to another, allowing **retargeting** and **inverse kinematics**.
With rare exceptions, humanoid models can be expected to have the same basic structure, representing the major articulate parts of the body, head and limbs. The Mecanim system makes good use of this idea to simplify the rigging and control of animations. A fundamental step in creating a animation is to set up a mapping between the simplified humanoid bone structure understood by Mecanim and the actual bones present in the skeleton; in Mecanim terminology, this mapping is called an **Avatar**. The pages in this section explain how to create an Avatar for your model.


- Creating the Avatar
- Configuring the Avatar
- Muscle setup
- Avatar Body Mask
- Retargeting of Humanoid animations
- Inverse Kinematics (Pro only)


Page last updated: 2012-11-08

---

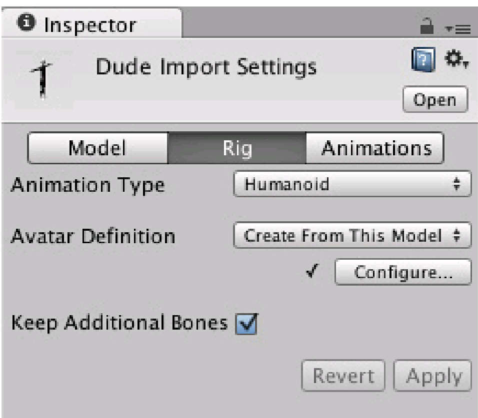
## Creating the Avatar

After an FBX file is imported, you can specify what kind of rig it is in the Rig tab of the FBX importer options.

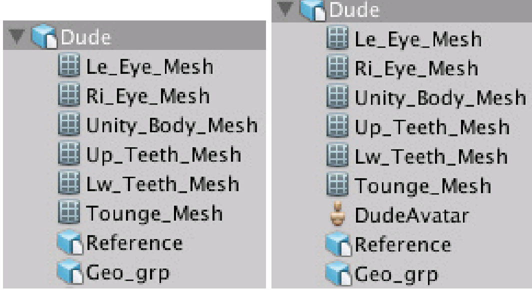
### Humanoid animations

For a Humanoid rig, select **Humanoid** and click **Apply**. Mecanim will attempt to match up your existing bone structure to the Avatar bone structure. In many cases, it can do this automatically by analysing the connections between bones in the rig.

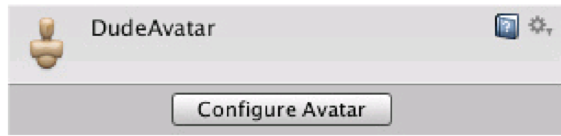
If the match has succeeded, you will see a check mark next to the **Configure...** menu




Also, in the case of a successful match, an Avatar sub-asset is added to the FBX asset, which you will be able to see in the project view hierarchy.




*Models with and without an Avatar sub-asset*




*The inspector for an Avatar asset*

If Mecanim was unable to create the Avatar, you will see a cross next to the **Configure ...** button, and no Avatar sub-asset will be added. When this happens, you need to configure the avatar manually.

### Non-humanoid animations

Two options for non-humanoid animation are provided: **Generic** and **Legacy**. Generic animations are imported using the Mecanim system but don't take advantage of the extra features available for humanoid animations. Legacy animations use the the animation system that was provided by Unity before Mecanim. There are some cases where it is still useful to work with legacy animations (most notably with legacy projects that you don't want to update fully) but they are seldom needed for new projects. See this section of the manual for further details on legacy animations.

(back to Avatar Creation and Setup)

(back to Mecanim introduction)

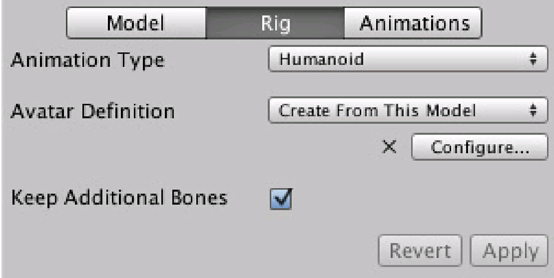
Page last updated: 2012-11-07

---

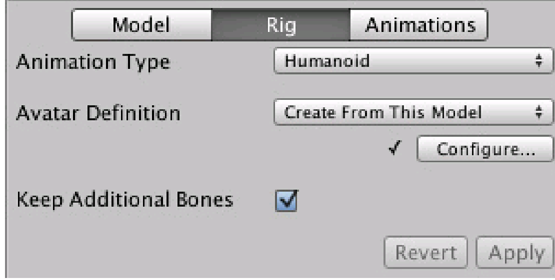
## Configuring the Avatar

Since the **Avatar** is such an important aspect of the Mecanim system, it is important that it is configured properly for your model. So, whether the automatic Avatar creation fails or succeeds, you need to go into the **Configure Avatar** mode to ensure your Avatar is valid and properly set up. It is important that your character's bone structure matches Mecanim's predefined bone structure *and* that the model is in T-pose.

If the automatic Avatar creation fails, you will see a cross next to the *Configure* button.

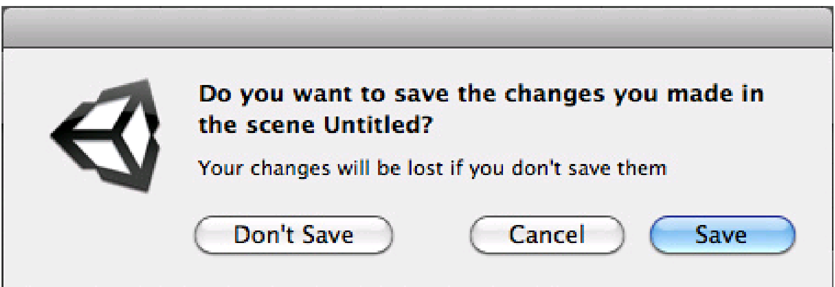



If it succeeds, you will see a check/tick mark:

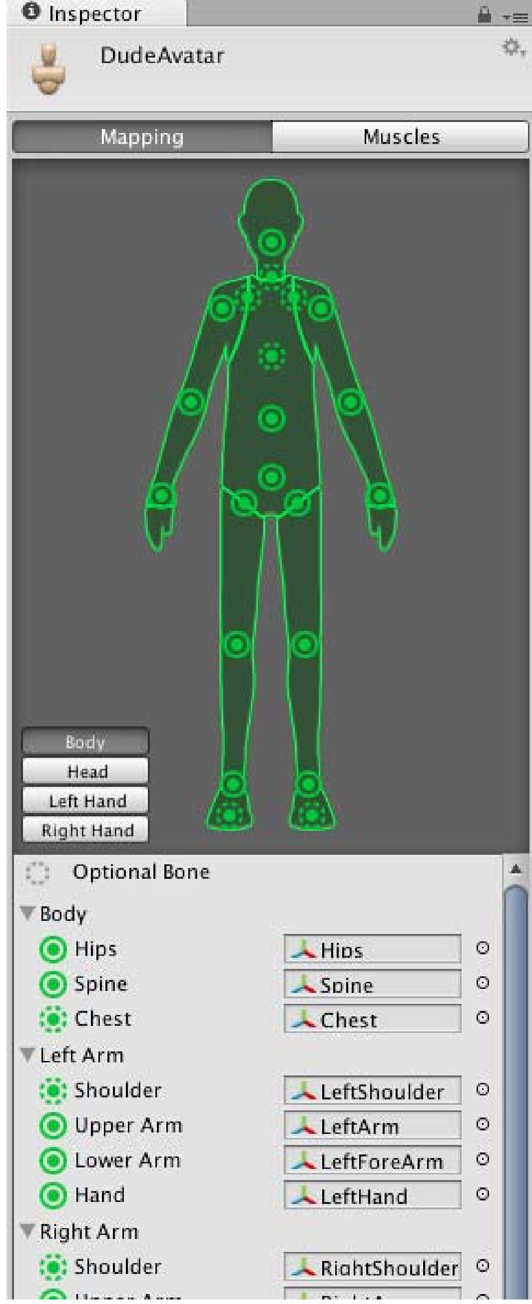



Here, success simply means all of the required bones have been matched but for better results, you might want to match the optional bones as well and get the model into a proper T-pose.

When you go to the **Configure ...** menu, the editor will ask you to save your scene. The reason for this is that in **Configure** mode, the Scene View is used to display bone, muscle and animation information for the selected model alone, without displaying the rest of the scene.



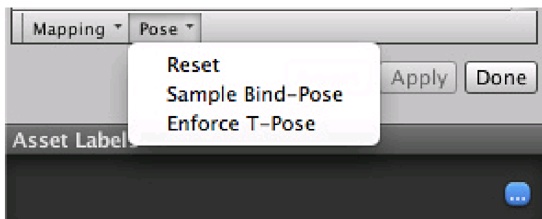

Once you have saved the scene, you will see a new Avatar Configuration inspector, with a bone mapping.




The inspector shows which of the bones are required and which are optional - the optional ones can have their movements interpolated automatically. For Mecanim to produce a valid match, your skeleton needs to have at least the required bones in place. In order to improve your chances for finding a match to the Avatar, name your bones in a way that reflects the body parts they represent (names like "LeftArm", "RightForearm" are suitable here).

If the model does NOT yield a valid match, you can manually follow a similar process to the one used internally by Mecanim:-

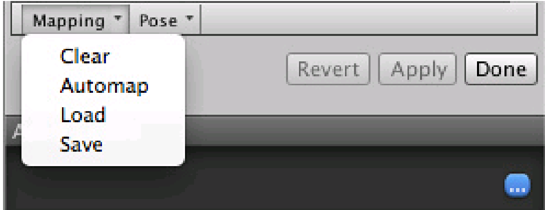
1. **Sample Bind-pose** (try to get the model closer to the pose with which it was modelled, a sensible initial pose)
2. **Automap** (create a bone-mapping from an initial pose)
3. **Enforce T-pose** (force the model closer to T-pose, which is the default pose used by Mecanim animations)




If the auto-mapping (**Mapping->Automap**) fails completely or partially, you can assign bones by either draging them from the

Scene or from the Hierarchy. If Mecanim thinks a bone fits, it will show up as green in the Avatar Inspector, otherwise it shows up in red.

Finally, if the bone assignment is correct, but the character is not in the correct *pose*, you will see the message "Character not in T-Pose". You can try to fix that with **Enforce T-Pose** or rotate the remaining bones into T-pose.




## Human Template files

You can save the mapping of bones in your skeleton to the Avatar on disk as a "human template file" (extention `*.ht`), which can be reused by any characters that use this mapping. This is useful, for example, if your animators use a consistent layout and naming convention for all skeleton but Mecanim doesn't know how to interpret it. You can then **Load** the .ht file for each model, so that manual remapping only needs to be done once.

(back to Avatar Creation and Setup)

(back to Mecanim introduction)

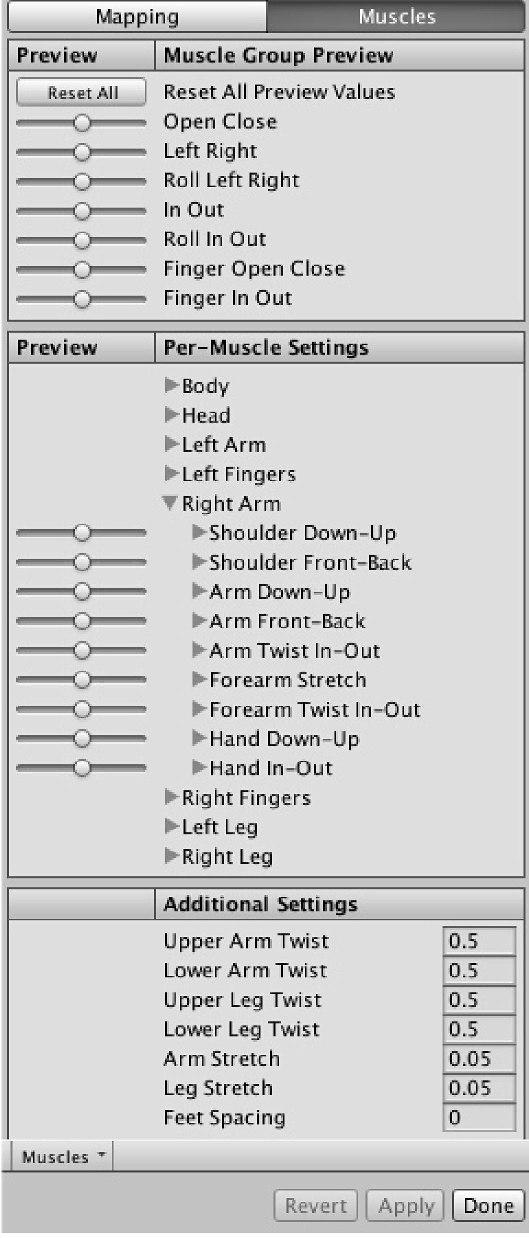
Page last updated: 2012-11-05

---

## Muscle Definitions

Mecanim allows you to control the range of motion of different bones using **Muscles**.

Once the Avatar has been properly configured, Mecanim will "understand" the bone structure and allow you to start working in the Muscles tab of the Avatar Inspector. Here, it is very easy to tweak the character's range of motion and ensure the character deforms in a convincing way, free from visual artifacts or self-overlaps.




You can either adjust individual bones in the body (lower part of the view) or manipulate the character using predefined deformations which operate on several bones at once (upper part of the view).

## Muscle Clips

In the Animation tab, you can set up **Muscle Clips**, which are animations for specific muscles and muscle groups.

You can also define which body parts these muscle clips apply to.

(back to Avatar Creation and Setup)

(back to Mecanim introduction)

Page last updated: 2012-11-01

---

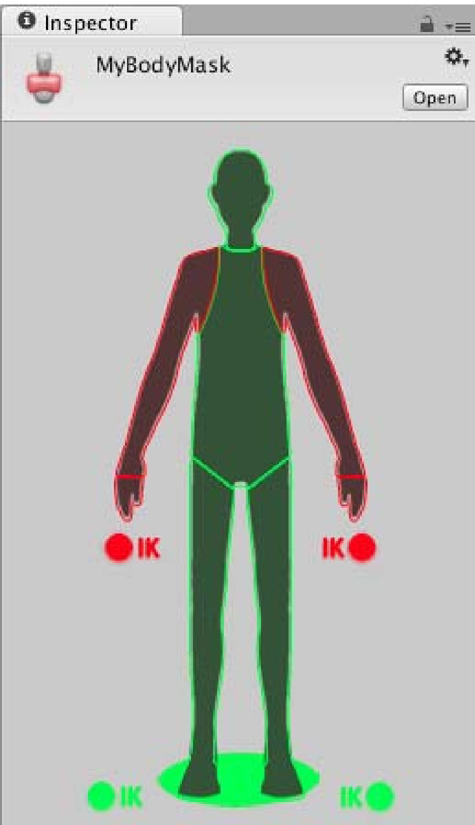
## Avatar Body Mask

Specific body parts can be selectively enabled or disabled in an animation using a so-called **Body Mask**. Body masks are used in the Animation tab of the mesh import inspector and Animation Layers. Body masks enable you to tailor an animation to fit the specific requirements of your character more closely. For example, you may have a standard walking animation that includes both arm and leg motion, but if a character is carrying a large object with both hands then you wouldn't want his arms to swing by his sides as he walks. However, you could still use the standard walking animation by switching off the arm

movements in the body mask.

The body parts included are: Head, Left Arm, Right Arm, Left Hand, Right Hand, Left Leg, Right Leg and Root (which is denoted by the "shadow" under the feet). In the body mask, you can also toggle **inverse kinematics** (IK) for hands and feet, which will determine whether or not IK curves will be included in animation blending.

- Click the avatar section to toggle inclusion or exclusion (green/red)
- Double click in empty space surrounding the avatar to toggle all



*Body mask in the Body Mask inspector (arms excluded)*

In the Animation tab of the mesh import inspector, you will see a list entitled *Clips* that contains all the object's animation clips. When you select an item from this list, options for the clip will be shown, including the body mask editor.

You can also create Body Mask Assets (**Assets->Create->Avatar Body Mask**), which show up as .mask files on disk.

The BodyMask assets can be reused in Animator Controllers, when specifying Animation Layers

A benefit of using body masks is that they tend to reduce memory overheads since body parts that are not active do not need their associated animation curves. Also, the unused curves need not be calculated during playback which will tend to reduce the CPU overhead of the animation.

(back to Mecanim introduction)

Page last updated: 2012-10-18

---

## Retargeting

One of the most powerful features of Mecanim is retargeting of humanoid animations. This means that with relative ease, you can apply the same set of animations to various character models. Retargeting is only possible for humanoid models, where an Avatar has been configured, because this gives us a correspondence between the models' bone structure.

### Recommended Hierarchy structure

When working with Mecanim animations, you can expect your scene to contain the following elements:-

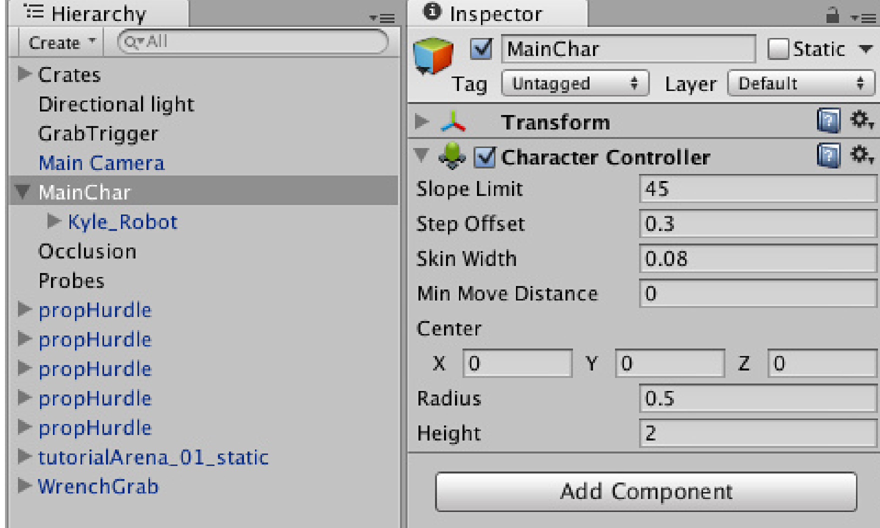
- The Imported character model, which has an Avatar on it.
- The Animator Component, referencing an Animator Controller asset.
- A set of animation clips, referenced from the Animator Controller.
- Scripts for the character.
- Character-related components, such as the Character Controller.

Your project should also contain another character model with a valid Avatar.

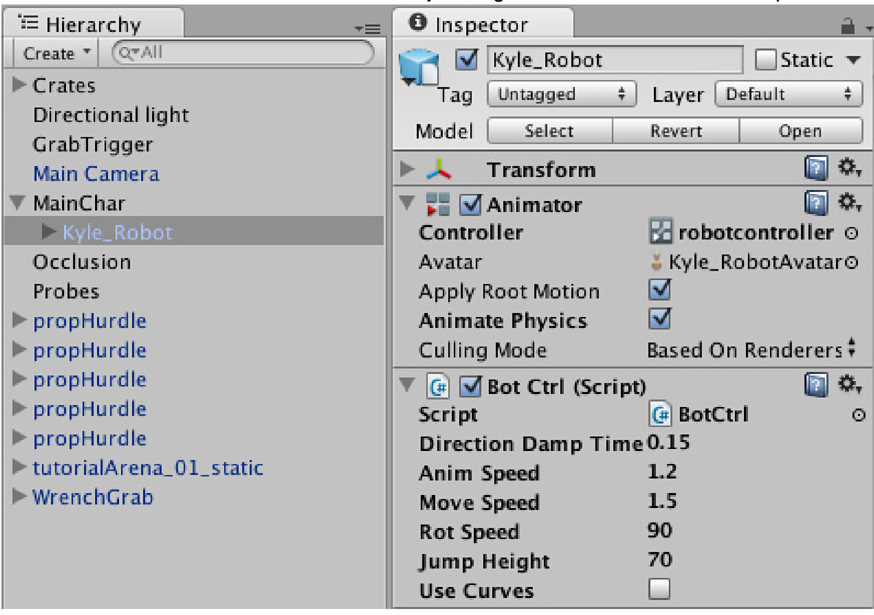
If in doubt about the terminology, consult the Animation Glossary

The recommended setup is to:

- Create a GameObject in the Hierarchy that contains Character-related components



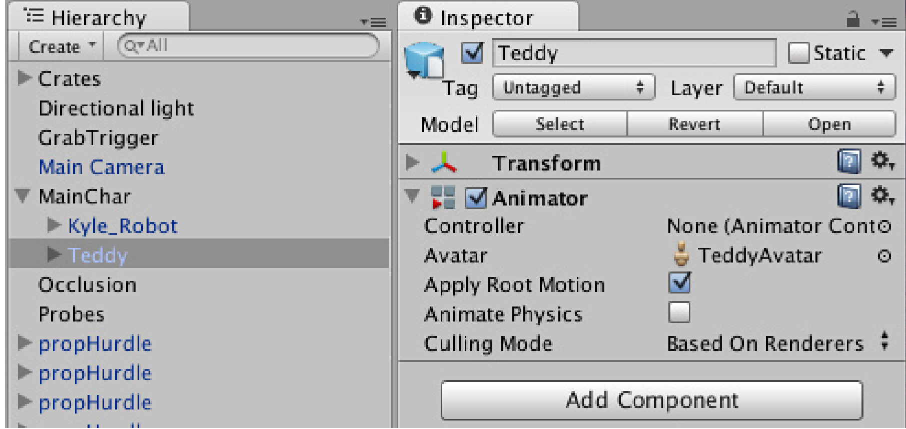

- Put the model as a child of the GameObject, together with the Animator component



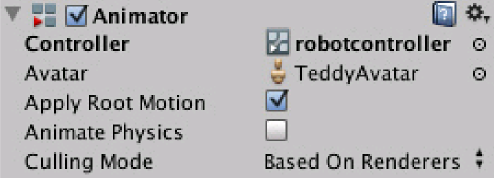

- Make sure scripts referencing the Animator are looking for the animator in the children instead of the root (use `GetComponentInChildren<Animator>()` instead of `GetComponent<Animator>()`)

Then in order to reuse the same animations on another model, you need to:

- Disable the original model
- Drop in the desired model as another child of GameObject




- Make sure the Animator Controller property for the new model is referencing the same controller asset




- Tweak the character controller, the transform, and other properties on the top-level GameObject, to make sure that the animations work smoothly with the new model.
- You're done!

(back to Mecanim introduction)

Page last updated: 2012-11-08

---

## Inverse Kinematics

Most animation is produced by rotating the angles of joints in a skeleton to predetermined values. The position of a child joint changes according to the rotation of its parent and so the end point of a chain of joints can be determined from the angles and relative positions of the individual joints it contains. This method of posing a skeleton is known as **forward kinematics**.

However, it is often useful to look at the task of posing joints from the opposite point of view - given a chosen position in space, work backwards and find a valid way of orienting the joints so that the end point lands at that position. This can be useful when you want a character to touch an object at a point selected by the user or plant its feet convincingly on an uneven surface. This approach is known as **Inverse Kinematics** (IK) and is supported in Mecanim for any humanoid character *with a correctly configured Avatar*.

To set up IK for a character, you typically have objects around the scene that a character interacts with, and then set up the IK thru script, in particular, Animator functions like SetIKPositionWeight, SetIKRotationWeight, SetIKPosition, SetIKRotation, SetLookAtPosition, bodyPosition, bodyRotation

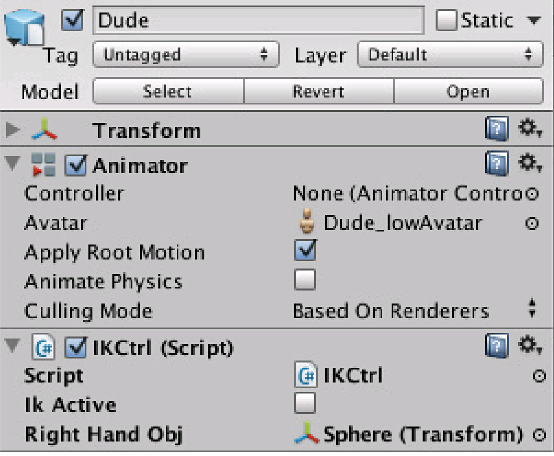
In the illustration above, we show a character grabbing a cylindrical object. How do we make this happen?

We start out with a character that has a valid Avatar, and attach to it a script that actually takes care of the IK, let's call it **IKCtrl**:

```
using UnityEngine;
using System;
using System.Collections;

[RequireComponent(typeof(Animator))]

public class IKCtrl : MonoBehaviour {

	protected Animator animator;

	public bool ikActive = false;
	public Transform rightHandObj = null;

	void Start ()
	{
		animator = GetComponent<Animator>();
	}

	//a callback for calculating IK
	void OnAnimatorIK()
	{
	    if(animator) {

			//if the IK is active, set the position and rotation directly to the goal.
			if(ikActive) {

				//weight = 1.0 for the right hand means position and rotation will be at the IK goal (the place the cha
				animator.SetIKPositionWeight(AvatarIKGoal.RightHand,1.0f);
				animator.SetIKRotationWeight(AvatarIKGoal.RightHand,1.0f);

				//set the position and the rotation of the right hand where the external object is
				if(rightHandObj != null) {
					animator.SetIKPosition(AvatarIKGoal.RightHand,rightHandObj.position);
					animator.SetIKRotation(AvatarIKGoal.RightHand,rightHandObj.rotation);
				}

			}

			//if the IK is not active, set the position and rotation of the hand back to the original position
			else {
				animator.SetIKPositionWeight(AvatarIKGoal.RightHand,0);
				animator.SetIKRotationWeight(AvatarIKGoal.RightHand,0);
			}
		}
	}
}
```

As we do not intend for the character to grab the entire object with his hand, we position a sphere where the hand should be on the cylinder, and rotate it accordingly.

This sphere should then be placed as the "Right Hand Obj" property of the IKCtrl script




Observe the character grabbing and ungrabbing the object as you click the **IKActive** checkbox

(back to Mecanim introduction)

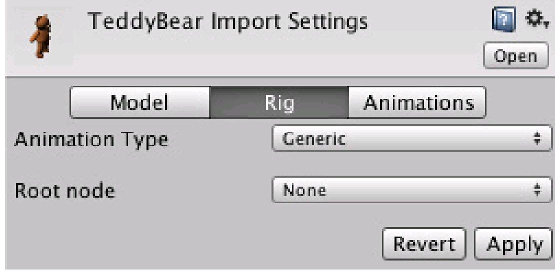
Page last updated: 2012-11-06

---

## Generic Animations

The full power of Mecanim is most evident when you are working with humanoid animations. However, non-humanoid animations are also supported although without the avatar system and other features. In Mecanim terminology, non-humanoid animations are referred to as **Generic Animations**.

To start working with a generic skeleton, go to the Rig tab in the FBX importer and choose **Generic** from the Animation Type menu.




### Root node in generic animations

While in the case of humanoid animations, we have the knowledge about the center of mass and orientation, in the case of Generic animations, the skeleton can be arbitrary, and we need to specify a reference bone, or the "root node". Selecting the root node allows us to establish correspondence between animation clips for a generic model, and blend properly between animations that are not "in place". The root node is also essential for separating animation of bones relative to reach other and motion of the root in the world (controlled from `OnAnimatorMove`)

Page last updated: 2012-11-06

---

## Bringing characters to life

Once the character mesh and animations are imported and the avatar is set up, you are ready to start making use of them in your game. The following sections cover the main features that Mecanim provides for controlling and sequencing your animations.

- Looping animation clips
- Animator Component and Animator Controller
- Animation State Machines
  - Animation States
  - Animation Transitions
  - Animation Parameters
- Blend Trees
- Mecanim Advanced topics
  - Working with Animation Curves in Mecanim (Pro only)
  - Sub-State Machines
  - Animation Layers
  - Animation State Machine Preview (solo and mute)
  - Target Matching
  - Root Motion - how it works
    - Tutorial: Scripting Root Motion for "in-place" humanoid animations

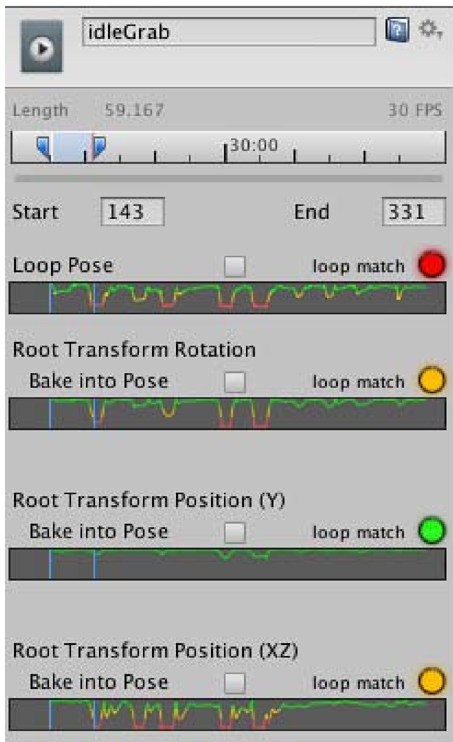

Page last updated: 2012-11-08

---

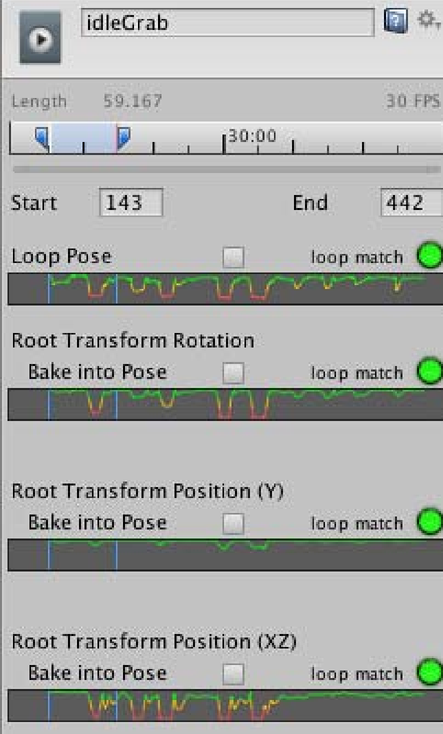
## Looping Animation Clips

A common operation for people working with animations is to make sure they loop properly. It is important, for example, that the animation clip representing the walk cycle, begins and ends in a similar pose (e.g. left foot on the ground), to ensure there is no foot sliding, or strange jerky motions. Mecanim provides convenient tools for this. Animation clips can loop based on pose, rotation, and position.

If you drag the Start or End points of the animation clip, you will see the Looping fitness curves for all of the paramers based on which it is possible to loop. If you place the Start / End marker in a place where the curve for the property is green, it is more likely that the clip can loop properly. The **loop match** indicator will show how good the looping is for the selected ranges.




Clip ranges with bad match for **Loop Pose**



Clip ranges with good match for **Loop Pose**

Once the **loop match** indicator is green, Enabling **Loop Pose** (for example) will make sure the looping of the pose is artifact-free.

For more details on animation clip options, see Animation Clip reference
(back to Mecanim introduction)

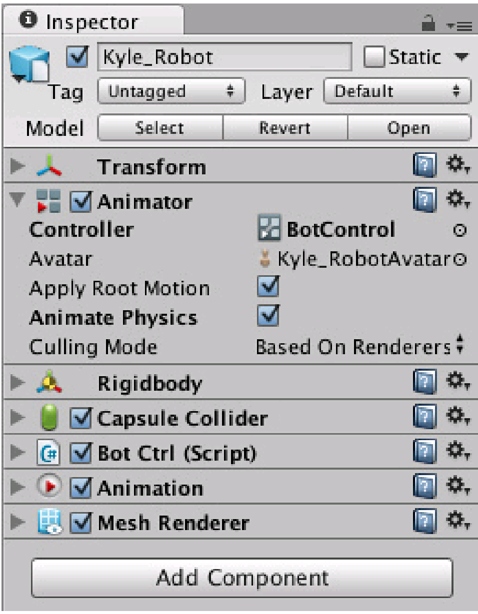
Page last updated: 2012-11-12

---

# Animator Component and Window

## Animator Component

Any GameObject that has an avatar will also have an **Animator** component, which is the link between the character and its behavior.




The **Animator** component references an **Animator Controller** which is used for setting up behavior on the character. This includes setup for State Machines, Blend Trees, and events to be controlled from script.

### Properties

| | |
|---|---|
| **Controller** | The animator controller attached to this character |
| **Avatar** | The Avatar for this character. |
| **Apply Root Motion** | Should we control the character's position from the animation itself or from script. |
| **Animate Physics** | Should the animation interact with physics? |
| **Culling Mode** | Culling mode for animations |
| **Always animate** | Always animate, don't do culling |
| **Based on Renderers** | When the renderers are invisible, only root motion is animated. All other body parts will remain static while the character is invisible. |

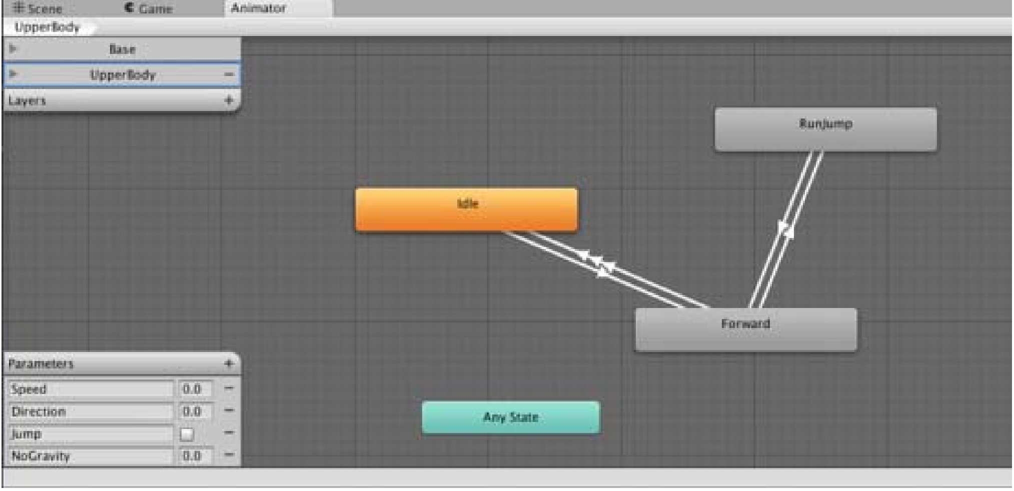
## Animator Controller

You can view and set up character behavior from the Animator Controller view (Menu: **Window > Animator Controller**).

An **Animator Controller** can be created from the Project View (Menu: **Create > Animator Controller**). This creates a `.controller` asset on disk, which looks like this in the Project Browser

Animator Controller asset on disk

After the state machine setup has been made, you can drop the controller onto the Animator component of any character with an Avatar in the Hierarchy View.




The Animator Controller Window

The Animator Controller Window will contain

- The Animation Layer Widget (top-left corner, see Animation Layers)
- The Event Parameters Widget (bottom-left, see Animation Parameters)
- The visualization of the State Machine itself.

Note that the Animator Controller Window will always display the state machine from the most recently selected `.controller` asset, regardless of what scene is currently loaded.

(back to Mecanim introduction)

Page last updated: 2012-10-18

---

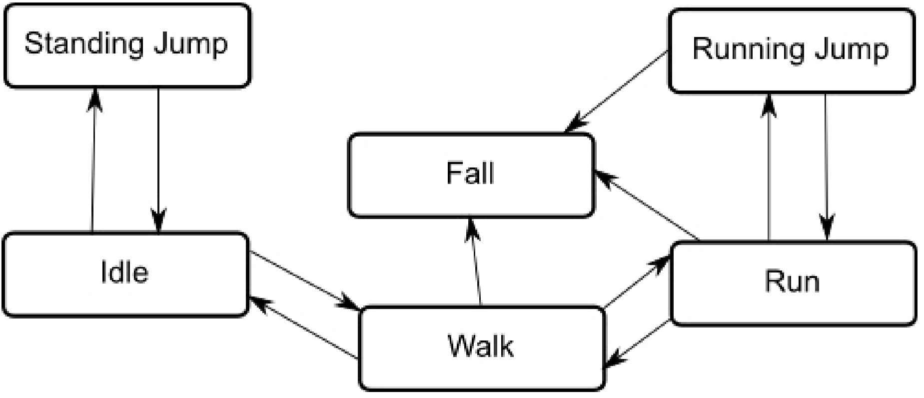
# Animation State Machines

It is common for a character to have several different animations that correspond to different actions it can perform in the game. For example, it may breathe or sway slightly while idle, walk when commanded to and raise its arms in panic as it falls from a platform. Controlling when these animations are played back is potentially quite a complicated scripting task. Mecanim borrows a computer science concept known as a **state machine** to simplify the control and sequencing of a character's animations.

## State machine basics

The basic idea is that a character is engaged in some particular kind of action at any given time. The actions available will depend on the type of gameplay but typical actions include things like idling, walking, running, jumping, etc. These actions are referred to as **states**, in the sense that the character is in a "state" where it is walking, idling or whatever. In general, the character will have restrictions on the next state it can go to rather than being able to switch immediately from any state to any other. For example, a running jump can only be taken when the character is already running and not when it is at a standstill, so it should never switch straight from the idle state to the running jump state. The options for the next state that a character can enter from its current state are referred to as **state transitions**. Taken together, the set of states, the set of transitions and the variable to remember the current state form a **state machine**.

The states and transitions of a state machine can be represented using a graph diagram, where the nodes represent the states and the arcs (arrows between nodes) represent the transitions. You can think of the current state as being a marker or highlight that is placed on one of the nodes and can then only jump to another node along one of the arrows.

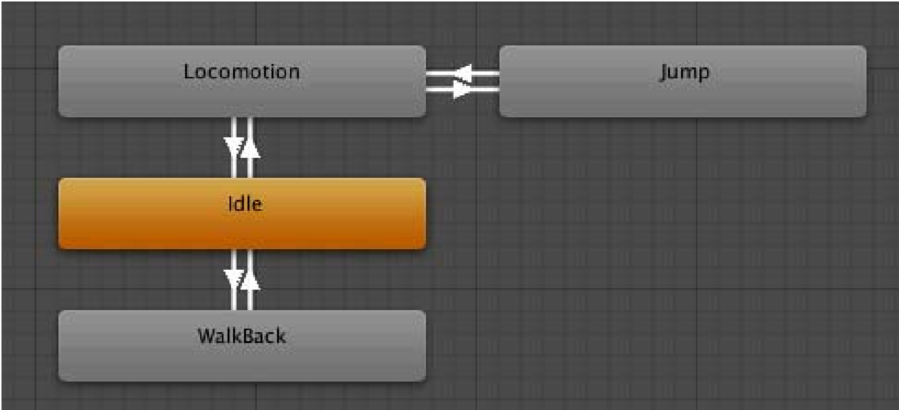



The importance of state machines for animation is that they can be designed and updated quite easily with relatively little coding. Each state has an animation sequence associated with it that will play whenever the machine is in that state. This enables an animator or designer to define the possible sequences of character actions and animations without being concerned about how the code will work.

### Mecanim state machines

Mecanim's Animation State Machines provide a way to overview all of the animation clips related to a particular character and allow various events in the game (for example user input) to trigger different animations.

Animation State Machines can be set up from the Animator Controller Window, and they look something like this:




State Machines consist of **States**, **Transitions** and **Events** and smaller Sub-State Machines can be used as components in larger machines.

- Animation States
- Animation Transitions
- Animation Parameters

(back to Mecanim introduction)

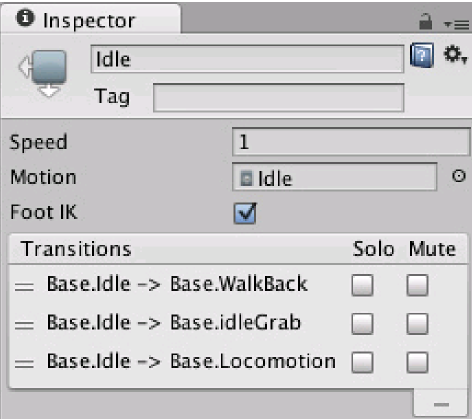
Page last updated: 2012-11-02

---

## Animation States

**Animation States** are the basic building blocks of an **Animation State Machine**. Each state contains an individual animation sequence (or blend tree) which will play while the character is in that state. When an event in the game triggers a state transition, the character will be left in a new state whose animation sequence will then take over.

When you select a state in the Animator Controller, you will see the properties for that state in the inspector:-



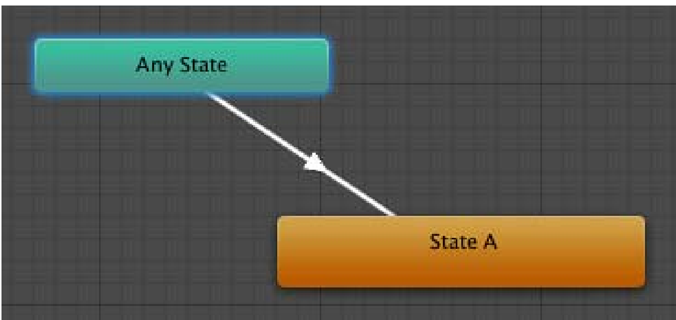

| | |
|---|---|
| **Speed** | The default speed of the animation |
| **Motion** | The animation clip assigned to this state |
| **Foot IK** | Should Foot IK be respected for this state |
| **Transitions** | The list of transitions originating from this state |

The default state, displayed in brown, is the state that the machine will be in when it is first activated. You can change the default state, if necessary, by right-clicking on another state and selecting **Set As Default** from the context menu. The *solo* and *mute* checkboxes on each transition are used to control the behaviour of **animation previews** - see this page for further details.

A new state can be added by right-clicking on an empty space in the Animator Controller Window and selecting **Create State->Empty** from the context menu. Alternatively, you can drag an animation into the Animator Controller Window to create a state containing that animation. (Note that you can only drag Mecanim animations into the Controller - non-Mecanim animations will be rejected.) States can also contain Blend Trees.

**Any State**

**Any State** is a special state which is always present. It exists for the situation where you want to go to a specific state regardless of which state you are currently in. This is a shorthand way of adding the same outward transition to all states in your machine. Note that the special meaning of **Any State** implies that it cannot be the end point of a transition (ie, jumping to "any state" cannot be used as a way to pick a random state to enter next).




(back to Animation State Machines)

Page last updated: 2012-10-18

## Animation Transitions

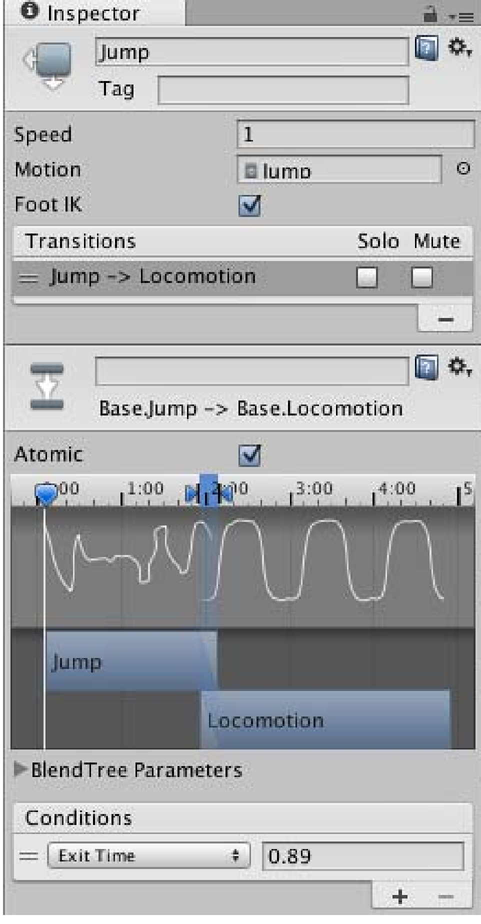
**Animation Transitions** define *what* happens when you switch from one **Animation State** to another. There can be only one transition active at any given time.

| | |
|---|---|
| **Atomic** | Is this transition atomic? (cannot be interrupted) |
| **Conditions** | Here we decide *when* transitions get triggered. |

A condition consists of:

- An event parameter
    - Instead of a parameter, you can also use **Exit Time**, and specify a number which represents the normalized time of the source state (e.g. 0.95 means the transition will trigger, when we've played the source clip 95% through).
- A conditional predicate, if needed (for example **Less**/**Greater** for floats).
- A parameter value (if needed).

You can adjust the transition between the two animation clips by dragging the start and end values of the overlap.




(See also Transition solo / mute)

(back to Animation State Machines)

Page last updated: 2012-10-18

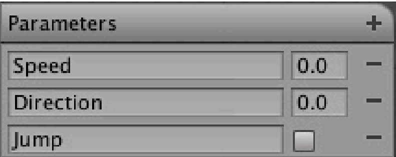
---

## Animation Parameters

Animation Parameters are variables that are defined within the animation system but can also be accessed and assigned values from scripts. For example, the value of a parameter can be updated by an animation curve and then accessed from a script so that, say, the pitch of a sound effect can be varied as if it were a piece of animation. Likewise, a script can set parameter values to be picked up by Mecanim. For example, a script can set a parameter to control a Blend Tree.

Default parameter values can be set up using the Parameters widget in the bottom left corner of the Animator window. They can be of four basic types:

- *Vector* - a point in space
- *Int* - an integer (whole number)
- *Float* - a number with a fractional part
- *Bool* - true or false value




Parameters can be assigned values from a script using functions in the Animator class: SetVector, SetFloat, SetInt, and SetBool.

Here's an example of a script that modifies parameters based on user input

```
using UnityEngine;
using System.Collections;


public class AvatarCtrl : MonoBehaviour {

	protected Animator animator;

	public float DirectionDampTime = .25f;

	void Start ()
	{
		animator = GetComponent<Animator>();
	}

	void Update ()
	{
		if(animator)
		{
			//get the current state
			AnimatorStateInfo stateInfo = animator.GetCurrentAnimatorStateInfo(0);

			//if we're in "Run" mode, respond to input for jump, and set the Jump parameter accordingly.
			if(stateInfo.nameHash == Animator.StringToHash("Base Layer.RunBT"))
			{
				if(Input.GetButton("Fire1"))
					animator.SetBool("Jump", true );
			}
			else
			{
				animator.SetBool("Jump", false);
			}

			float h = Input.GetAxis("Horizontal");
			float v = Input.GetAxis("Vertical");

			//set event parameters based on user input
			animator.SetFloat("Speed", h*h+v*v);
			animator.SetFloat("Direction", h, DirectionDampTime, Time.deltaTime);
		}
	}
}
```

(back to Animation State Machines)

Page last updated: 2012-11-09

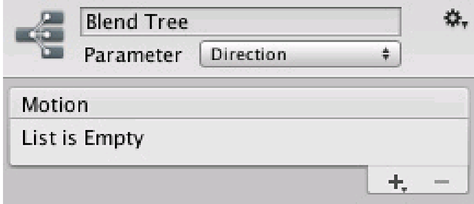
## Animation Blend Trees

A common task in game animation is to transition between two or more similar sequences. Perhaps the best-known example is the transition between walking and running animations according to the character's speed (ie, running movements are not just faster versions of walking movements, so they require separate animation clips). Another example is a character leaning to the left or right as he turns during a run. The most important detail of the transition is to ensure that it happens smoothly without a noticeable jerk where the animations are switched.

**Blend Trees** are Mecanim's method of allowing one animation to be blended smoothly with another. By tracking the bone movements of the two animations precisely, Mecanim can incorporate parts of both to varying degrees. The amount that each of the two animation clips contributes to the final effect is controlled using a **blending parameter**, which is just one of the numeric animation parameters associated with the character. To make for a smooth transition, Mecanim requires that the two clips to be blended are *aligned* so that the corresponding movements take place at the same points in normalized time. For example, walking and running animations can be aligned so that the individual footfalls take place at the same points in normalized time, even though the running cycle is faster in real time (the left foot hits at 0.0, the right foot at 0.5, say).

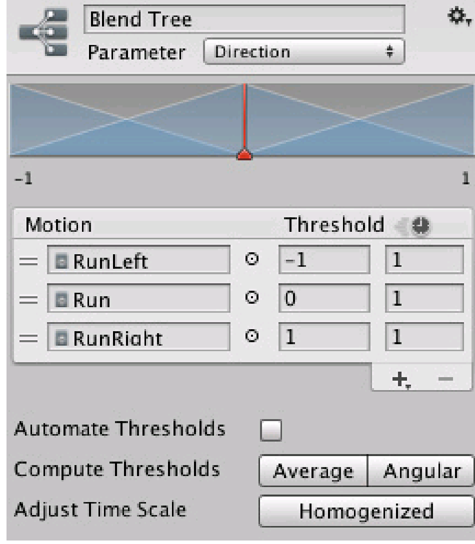
To start working with a new blend tree, you need to:

1. Right-click on empty space on the Animator Controller Window
2. Select **Create State > From New Blend Tree** from the context menu that appears.
3. Double-click on the Blend Tree to bring up the Blend Tree Inspector. (Note that you will just get the standard state inspector if you single click here.)

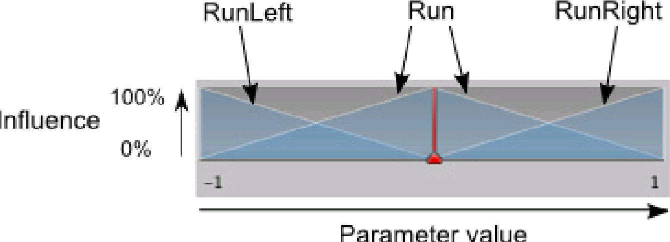



In the inspector, the first thing you need is to select the Animation Parameter that will control this Blend Tree. In this example, the parameter is *direction* which varies between -1.0 (left) and +1.0 (right), with 0.0 denoting a straight run without leaning.

Then you can add individual animations by clicking **+ -> Add Motion Field** to add an animation clip to the blend tree. When you're done, it should look something like this:




The lines in the diagram at the top of the inspector show the proportion of each animation that is incorporated as the parameter varies between its minimum and maximum values.

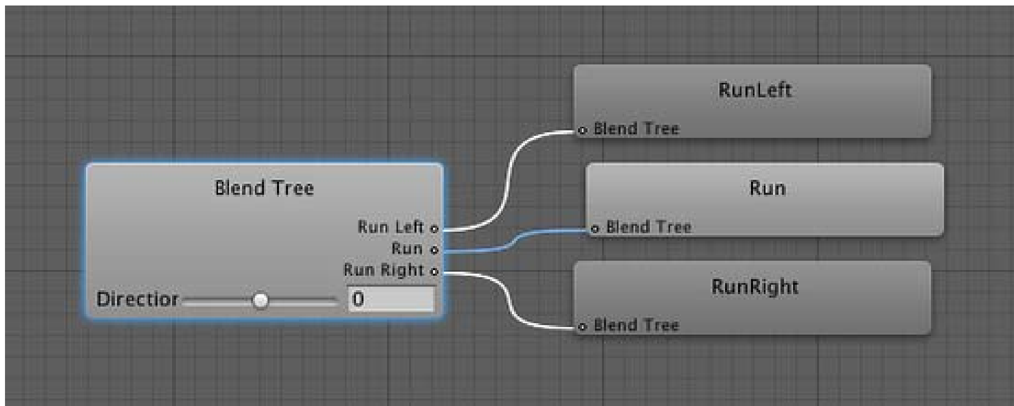



If you click and hold down the left mouse button on a line or the space beneath, the animation that it corresponds to will be highlighted in the list below. You can also drag the line left and right to change the parameter range over which it influences the animation (this also performs a live update of the Threshold values in the inspector).

The *Automate Thresholds* checkbox will distribute the clips' thresholds evenly across the numeric range of the parameter. For example, if there are five clips and the parameter ranges from -90 to +90, the thresholds will be set to -90, -45, 0, +45 and +90 in order. The *Compute Thresholds* buttons will set the thresholds from speed data obtained from the animation. Say, for example, you had a walk animation that covered 1.5 units per second, a jog at 2.3 units per second, and a run at 4 units per second, clicking the *Average* button would set the parameter range and thresholds for the three animations based on these values. So, if you set the speed parameter to 3.0, it would blend the jog and run with a slight bias toward the jog. The *Angular* button performs a similar calculation but based on angular speed (degrees per second) rather than linear speed. You can alter the "natural" speed of the animation clips using the time scaling text boxes (the column next to the threshold values), so you could make the walk twice as fast by using a value of 2.0 as its scale. The *Adjust Time Scale > Homogenized* button rescales the speeds of the clips so that they correspond with the chosen minimum and maximum values of the parameter but keep the same *relative* speeds they initially had.

The red vertical bar indicates the value of the event parameter. If you press **Play** in the Animation Preview panel and drag the bar left and right, you can see how the value of the parameter is controlling the blending of the different animations.

In the Animation view, a diagram of the blend tree complements the information shown in the inspector.




This gives a graphical display of how the animations are combined as the parameter value changes (as you drag the slider, the arrows from the tree root change their shading to show the dominant animation clip).

(back to Mecanim introduction)

Page last updated: 2012-11-08

---

# Advanced topics

The following section covers the features Mecanim provides for controlling and managing complex sets of animations.

- Working with Animation Curves in Mecanim (Pro only)
- Sub-State Machines
- Animation Layers
- Animation State Machine Preview (solo and mute)

- Target Matching
- Root Motion - how it works
  - Tutorial: Scripting Root Motion for "in-place" humanoid animations

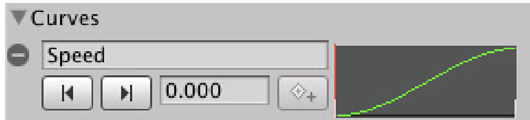
(back to Mecanim introduction)

Page last updated: 2012-11-08

---

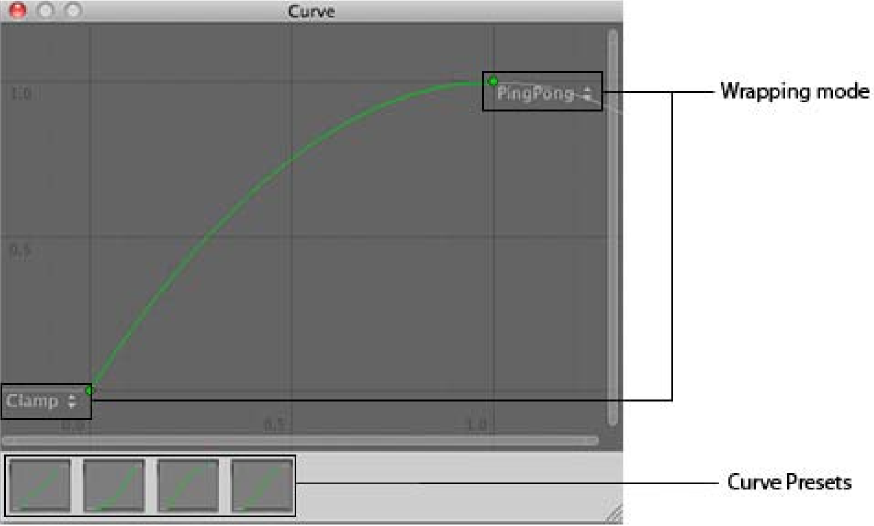
# Animation Curves in Mecanim

Animation curves can be attached to animation clips in the Animations tab of the **Animation Import Settings**.



*The curves on animation clips in Mecanim*

The curve's X-axis represents *normalized time* and always ranges between 0.0 and 1.0 (corresponding to the beginning and the end of the animation clip respectively, regardless of its duration).



*Unity Curve Editor*

Double-clicking an animation curve will bring up the standard Unity curve editor (see Editing Value Properties for further details) which you can use to add **keys** to the curve. Keys are points along the curve's timeline where it has a value explicitly set by the animator rather than just using an interpolated value. Keys are very useful for marking important points along the timeline of the animation. For example, with a walking animation, you might use keys to mark the points where the left foot is on the ground, then both feet on the ground, right foot on the ground, etc. Once the keys are set up, you can move conveniently between key frames by pressing the **Previous/Next Key Frame** buttons. This will move the vertical red line and show the *normalized time* at the keyframe; the value you enter in the text box will then set the value of the curve at that time.

### Animation Curves and Animator Controller parameters

If you have a curve with the same name as one of the parameters in the Animator Controller, then that parameter will take its value from the value of the curve at each point in the timeline. For example, if you make a call to GetFloat from a script, the returned value will be equal to the value of the curve at the time the call is made. Note that at any given point in time, there might be multiple animation clips attempting to set the same parameter from the same controller. In that case, the curve values from the multiple animation clips are blended. If an animation has no curve for a particular parameter then the blending will be done with the default value for that parameter.

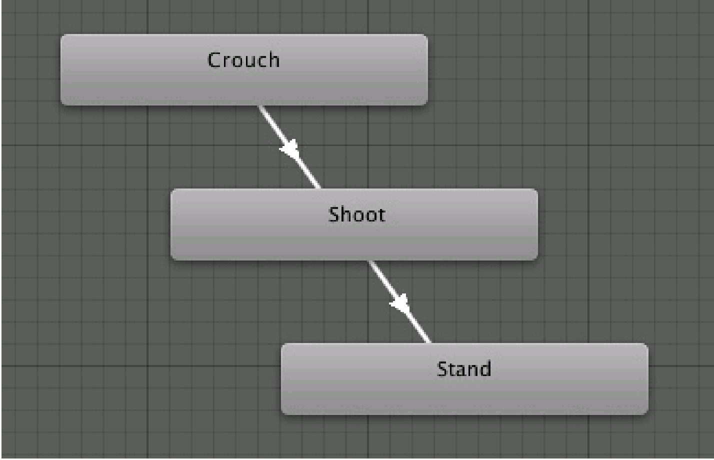
(back to Mecanim introduction)

Page last updated: 2012-11-08

---

## Nested State Machines

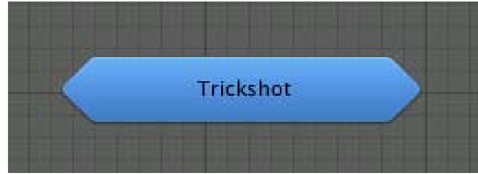
It is common for a character to have complex actions that consist of a number of stages. Rather than handle the entire action with a single state, it makes sense to identify the separate stages and use a separate state for each. For example, a character may have an action called "Trickshot" where it crouches to take a steady aim, shoots and then stands up again.



*The sequence of states in a "Trickshot" action*

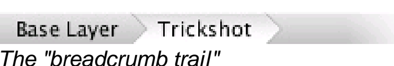
Although this is useful for control purposes, the downside is that the state machine will become large and unwieldy as more of these complex actions are added. You can simplify things somewhat just by separating the groups of states visually with empty space in the editor. However, Mecanim goes a step further than this by allowing you to collapse a group of states into a single named item in the state machine diagram. These collapsed groups of states are called **Sub-state machines**.

You can create a sub-state machine by right clicking on an empty space within the Animator Controller window and selecting **Create Sub-State Machine** from the context menu. A sub-state machine is represented in the editor by an elongated hexagon to distinguish it from normal states.



*A sub-state machine*

When you double-click the hexagon, the editor is cleared to let you edit the sub-state machine as though it were a completely separate state machine in its own right. The bar at the top of the window shows a "breadcrumb trail" to show which sub-state machine is currently being edited (and note that you can create sub-state machines within other sub-state machines, and so on). Clicking an item in the trail will focus the editor on that particular sub-state machine.



*The "breadcrumb trail"*

### External transitions

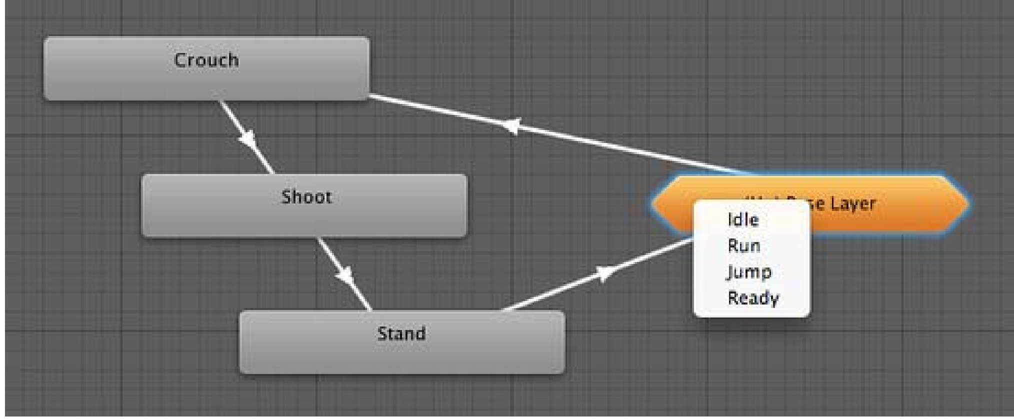
As noted above, a sub-state machine is just a way of visually collapsing a group of states in the editor, so when you make a transition to a sub-state machine, you have to choose which of its states you want to connect to.

*Choosing a target state within the "Trickshot" sub-state machine*

You will notice an extra state in the sub-state machine whose name begins with *Up*.

*The "Up" state*

The *Up* state represents the "outside world", the state machine that encloses the sub-state machine in the view. If you add a transition from a state in sub-state machine to the *Up* state, you will be prompted to choose one of the states of the enclosing machine to connect to.




*Connecting to a state in the enclosing machine*

(back to State Machines introduction)

(back to Mecanim introduction)

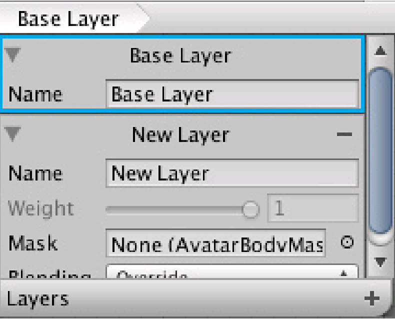
Page last updated: 2012-11-07

---

## Animation Layers

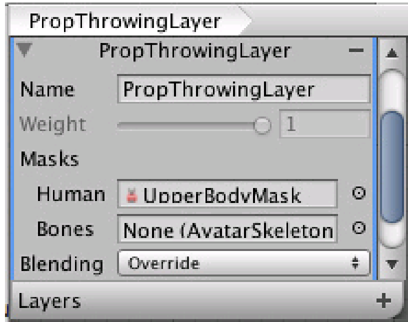
Unity uses **Animation Layers** for managing complex state machines for different body parts. An example of this is if you have a lower-body layer for walking-jumping, and an upper-body layer for throwing objects / shooting.

You can manage animation layers from the Layers Widget in the top-left corner of the Animator Controller.




You can add a new layer by pressing the **+** on the widget. On each layer, you can specify the body mask (the part of the body on which the animation would be applied), and the Blending type. **Override** means information from other layers will be ignored, while **Additive** means that the animation will be added on top of previous layers.

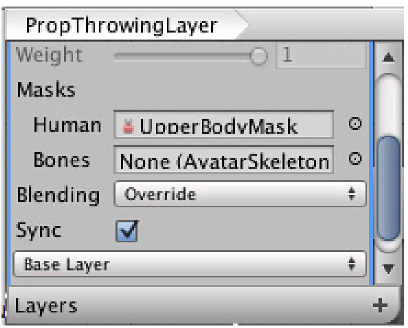
The **Mask** property is there to specify the body mask used on this layer. For example if you want to use upper body throwing animations, while having your character walk or run, you would use an upper body mask, like this:




For more on Avatar Body Masks, you can read this section

### Animation Layer syncing (Pro only)

Sometimes it is useful to be able to re-use the same state machine in different layers. For example if you want to simulate "wounded" behavior, and have "wounded" animations for walk / run / jump instead of the "healthy" ones. You can click the **Sync** checkbox on one of your layers, and then select the layer you want to sync with. The state machine structure will then be the same, but the actual animation clips used by the states will be distinct.




(back to Mecanim introduction)

Page last updated: 2012-11-06

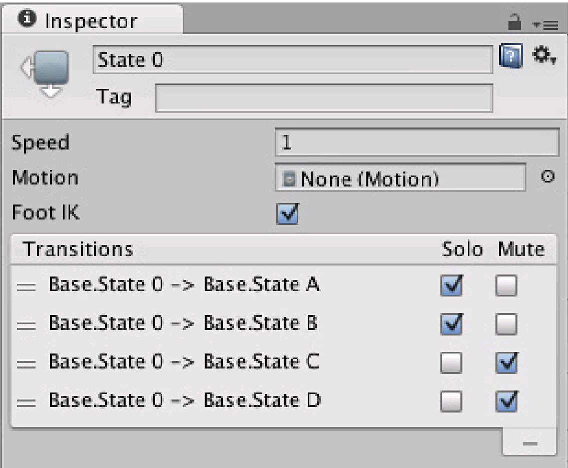
---

## Animation State Machine Preview (solo and mute)

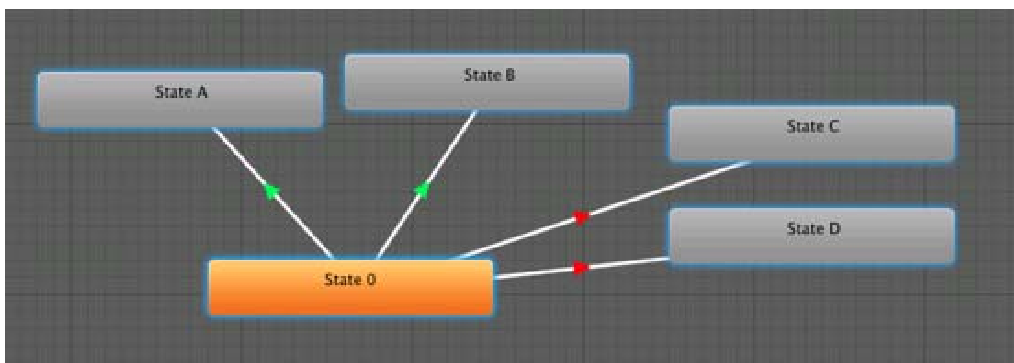
### Solo and Mute functionality

In complex state machines, it is useful to preview the operation of some parts of the machine separately. For this, you can use the Mute / Solo functionality. Muting means a transition will be disabled. Soloed transtions are enabled and with respect to other transitions originating from the same state. You can set up mute and solo states either from the Transition Inspector, or

the State Inspector (recommended), where you'll have an overview of all the transitions from that state.




Soloed transitions will be shown in green, while muted transitions in red, like this:




In the example above, if you are in `State 0`, only transitions to `State A` and `State B` will be available.

- The basic rule of thumb is that if one Solo is ticked, the rest of the transitions from that state will be muted.
- If both Solo and Mute are ticked, then Mute takes precedence.

Known issues:

- The controller graph currently doesn't always reflect the internal mute states of the engine.

(back to State Machines introduction)

(back to Mecanim introduction)

Page last updated: 2012-10-08

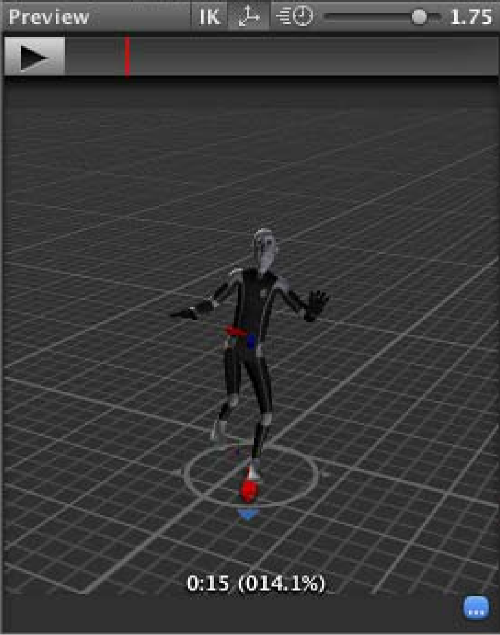
---

## Target Matching

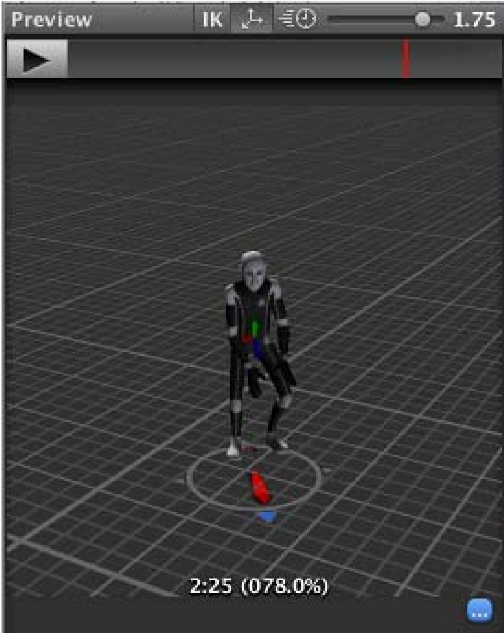
Often in games, a situation arises where a character must move in such a way that a hand or foot lands at a certain place at a certain time. For example, the character may need to jump across stepping stones or jump and grab an overhead beam.

You can use the Animator.MatchTarget function to handle this kind of situation. Say, for example, you want to arrange an situation where the character jumps onto a platform and you already have an animation clip for it called *Jump Up*. To do this, follow the steps below.

- Find the place in the animation clip at which the character is beginning to get off the ground, note in this case it is 14.1% or 0.141 into the animation clip in normalized time.




- Find the place in the animation clip at which the character is about to land on his feet, note in this case the value is 78.0% or 0.78.

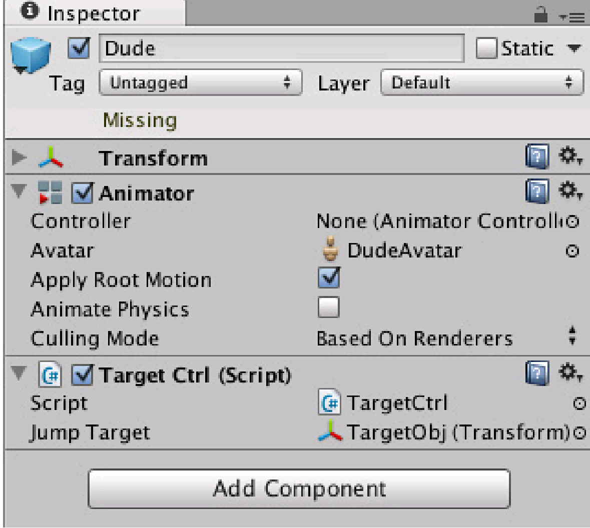



- Create a script (`TargetCtrl.cs`) that makes a call to MatchTarget, like this:

```
using UnityEngine;
using System;

[RequireComponent(typeof(Animator))]
public class TargetCtrl : MonoBehaviour {

	protected Animator animator;

	//the platform object in the scene
	public Transform jumpTarget = null;
	void Start () {
		animator = GetComponent<Animator>();
	}

	void Update () {
```

```
            if(animator) {
                  if(Input.GetButton("Fire1"))
                        animator.MatchTarget(jumpTarget.position, jumpTarget.rotation, AvatarTarget.LeftFoot,
                                                new MatchTargetWeightMask(Vector3.one, 1f), 0.141f, 0.78f);
            }
      }
}
```

Attach that script onto the Mecanim model.




The script will move the character so that it jumps from its current position and lands with its left foot at the target. Bear in mind that the result of using MatchTarget will generally only make sense if it is called at the right point in gameplay.

(back to Mecanim introduction)

Page last updated: 2012-11-08

---

# Root Motion

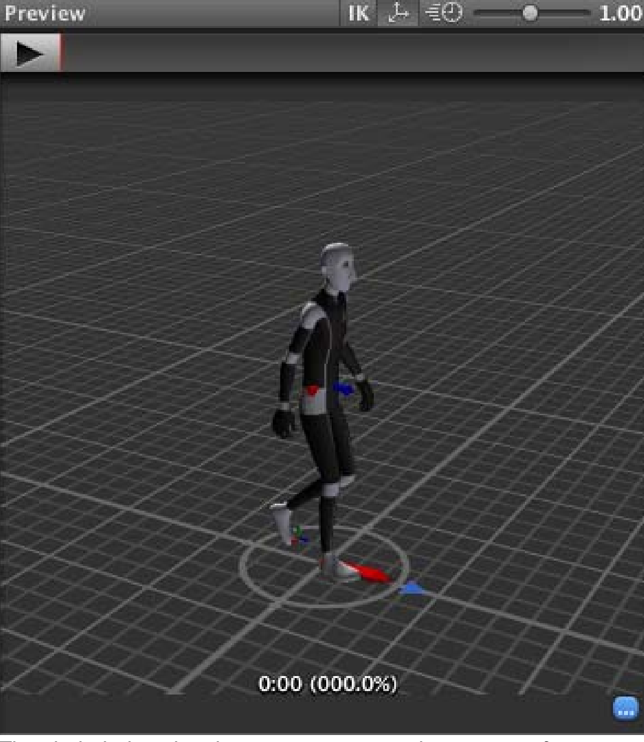
## Body Transform

The body transform must be set to be the same for all humanoid characters (from a retargeting standpoint). The body mass center should be used as the body position as this will nearly follow a straight line in most circumstances. The body orientation is an average of the lower and upper body orientation. Do not use the hips to store the world-space position and orientation of the animation as this can lead to unpredictable results. The body orientation is at identity for the Avatar T-Pose.

The body position and orientation are stored in the **Animation Clip** (using the Muscle definitions set up in the Avatar). They are the only world-space curves stored in the **Animation Clip**. Everything else: muscle curves and IK goals (Hands and Feet) are stored relative to the body transform.

## Root Transform

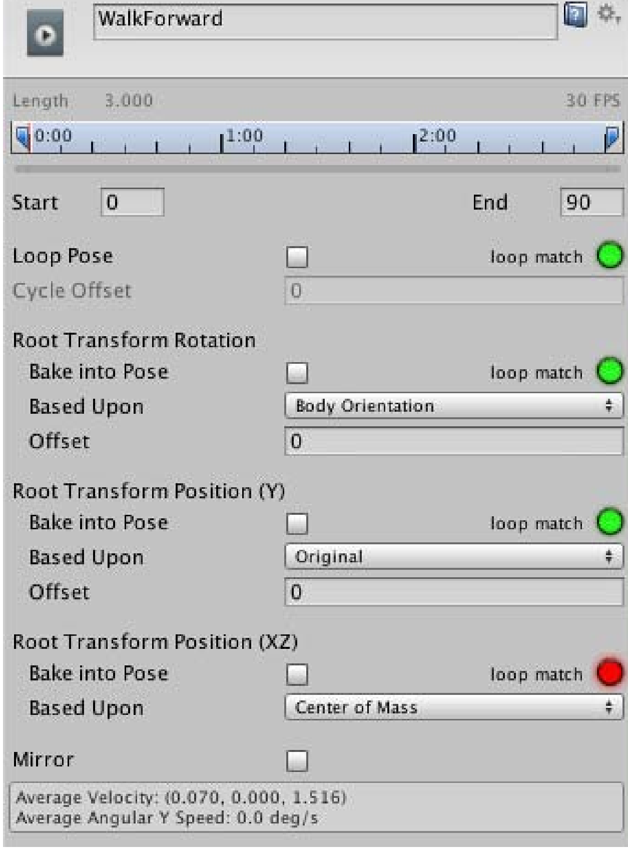
The Root Transform is a projection on the Y plane of the Body Transform and is computed at runtime. At every frame, a change in the Root Transform is computed. This change in transform is then applied to the Game Object to make it move.



*The circle below the character represents the root transform*

### Animation Clip Inspector

The Animation Clip Editor settings (**Root Transform Rotation**, **Root Transform Position (Y)** and **Root Transform Position (XZ)**) let you control the Root Transform projection from the Body Transform. Depending on these settings some parts of the Body Transform may be transferred Root Transform. For example you can decide if you want the motion Y position to be part of the Root Motion (trajectory) or part of the pose (body transform), which is known as Baked into Pose.




### Root Transform Rotation

Bake into Pose: The orientation will stay on the body transform (or Pose). The Root Orientation will be constant and delta Orientation will be identity. This means the the Game Object will not be rotated at all by that **AnimationClip**.

Only AnimationClips that have similar start and stop Root Orientation should use this option. You will have a Green Light in the UI telling you that an **AnimationClip** is a good candidate. A suitable candidate would be a straight walk or a run.

Based Upon: This let you set the orientation of the clip. Using **Body Orientation**, the clip will be oriented to follow the forward vector of body. This default setting works well for most Motion Capture (Mocap) data like walks, runs, and jumps, but it will fail with motion like strafing where the motion is perpendicular to the body's forward vector. In those cases you can manually adjust the orientation using the **Offset** setting. Finally you have **Original** that will automatically add the authored offset found in the imported clip. It is usually used with Keyframed data to respect orientation that was set by the artist.

Offset: used to enter the offset when that option is chosen for Based Upon.

### Root Transform Position (Y)

This uses the same concepts described in Root Transform Rotation.

Bake Into Pose: The Y component of the motion will stay on the Body Transform (Pose). The Y component of the Root Transform will be constant and Delta Root Position Y will be 0. This means that this clip won�t change the Game Object Height. Again you have a Green Light telling you that a clip is a good candidate for baking Y motion into pose.

Most of the **AnimationClips** will enable this setting. Only clips that will change the **GameObject** height should have this turned off, like jump up or down.

Note: the `Animator.gravityWeight` is driven by Bake Into Pose position Y. When enabled, `gravityWeight = 1`, when `disable = 0`. **gravityWeight** is blended for clips when transitioning between states.

Based Upon: In a similar way to **Root Transform Rotation** you can choose from **Original** or **Mass Center (Body)**. There is also a **Feet** option that is very convenient for **AnimationClips** that change height (Bake Into Pose disabled). When using **Feet** the Root Transform Position Y will match the lowest foot Y for all frames. Thus the blending point always remains around the feet which prevents floating problem when blending or transitioning.

Offset: In a similar way to **Root Transform Rotation**, you can manually adjust the **AnimationClip** height using the **Offset** setting.

### Root Transform Position (XZ)

Again, this uses same concepts described in **Root Transform Rotation** and **Root Motion Position (Y).**

Bake Into Pose will usually be used for �Idles� where you want to force the delta Position (XZ) to be 0. It will stop the accumulation of small deltas drifting after many evaluations. It can also be used for a Keyframed clip with Based Upon **Original** to force an authored position that was set by the artist.

### Loop Pose

Loop Pose (like Pose Blending in Blend Trees or Transitions) happens in the referential of Root Transform. Once the Root Transform is computed, the Pose becomes relative to it. The relative Pose difference between Start and Stop frame is computed and distributed over the range of the clip from 0-100%.

### Generic Root Motion and Loop Pose.

This works in essentially the same as Humanoid Root Motion, but instead of using the Body Transform to compute/project a Root Transform, the transform set in **Root Node** is used. The Pose (all the bones which transform below the Root Motion bone) is made relative to the Root Transform.

Page last updated: 2012-11-09

---

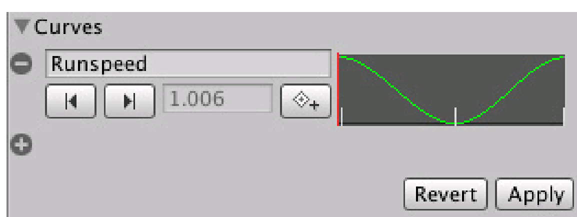
## Scripting Root Motion

Sometimes your animation comes as "in-place", which means if you put it in a scene, it will not move the character that it's on. In other words, the animation does not contain "root motion". For this, we can modify root motion from script. To put everything

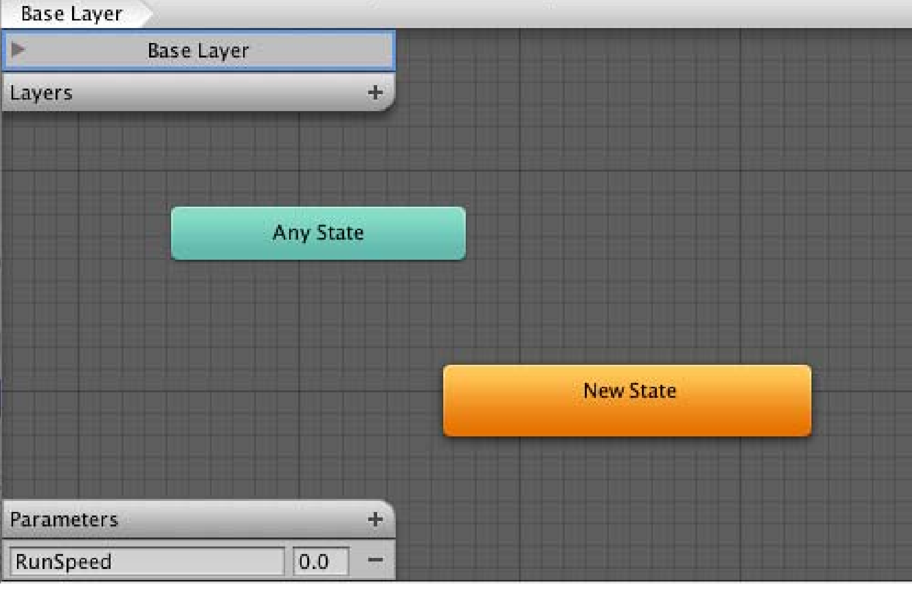
together follow the steps below (note there are many variations of achieving the same result, this is just one recipe).

- Open the inspector for the FBX file that contains the in-place animation, and go to the Animation tab
- Make sure the **Muscle Definition** is set to the Avatar you intend to control (let's say this avatar is called *Dude*, and he has already been added to the Hierarchy View).
- Select the animation clip from the available clips
- Make sure **Loop Pose** is properly aligned (the light next to it is green), and that the checkbox for **Loop Pose** is clicked

- Preview the animation in the animation viewer to make sure the beginning and the end of the animation align smoothly, and that the character is moving "in-place"
- On the animation clip create a curve that will control the speed of the character (you can add a curve from the Animation Import inspector **Curves-> +**)
- Name that curve something meaningful, like "Runspeed"




- Create a new **Animator Controller**, (let's call it **RootMotionController**)
- Drop the desired animation clip into it, this should create a state with the name of the animation (say **Run**)
- Add a parameter to the Controller with the same name as the curve (in this case, "Runspeed")

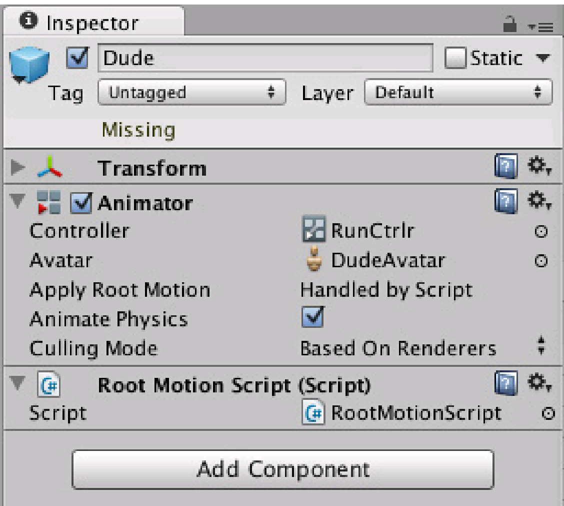



- Select the character **Dude** in the Hierarchy, whose inspector should already have an **Animator** component.
- Drag **RootMotionController** onto the **Controller** property of the Animator
- If you press play now, you should see the "Dude" running in place

- Finally, to control the motion, we will need to create a script (RootMotionScript.cs), that implements the `OnAnimatorMove` callback.

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Animator))]

public class RootMotionScript : MonoBehaviour {

        void OnAnimatorMove()
        {
            Animator animator = GetComponent<Animator>();

            if (animator)
            {
               Vector3 newPosition = transform.position;
               newPosition.z += animator.GetFloat("Runspeed") * Time.deltaTime;
               transform.position = newPosition;
            }
        }
}
```

- Attach RootMotionScript.cs to "Dude"
- Note that the Animator component detects there is a script with `OnAnimatorMove` and **Apply Root Motion** property shows up as *Handled by Script*




- Now you should see that the character is moving at the speed specified.

(back to Mecanim introduction)

Page last updated: 2012-11-07

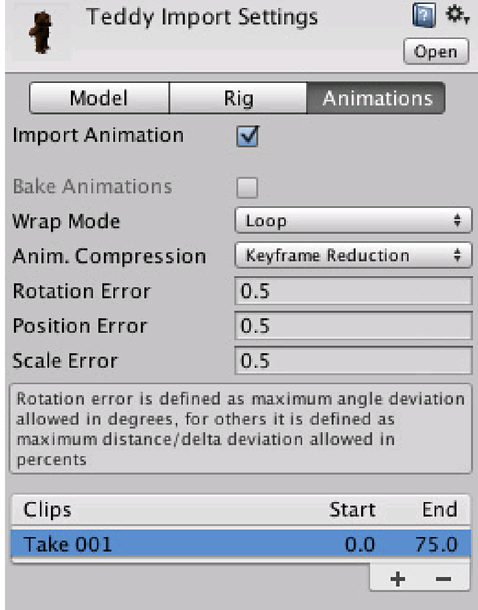
---

# Legacy Animation system

Prior to the introduction of Mecanim, Unity used its own animation system and for backward compatiblity, this system is still available. The main reason for using legacy animation is to continue working with an old project without the work of updating it for Mecanim. However, it is not recommended that you use the legacy system for new projects.

## Working with legacy animations

To import a legacy animation, you first need to mark it as such in the Mesh importer's Rig tab:-

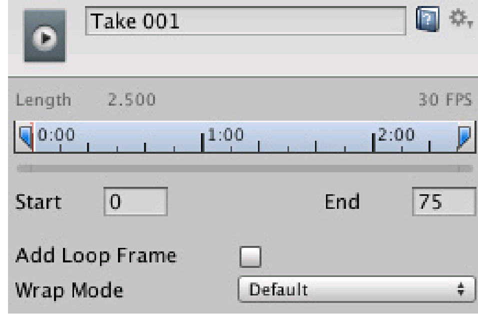



The Animation tab on the importer will then look something like this:-

| | |
|---|---|
| **Import Animation** | Selects whether or not animation should be imported at all. |
| **Wrap Mode** | The method of handling what happens when the animation comes to an end:- |
| **Default** | Uses whatever setting is specified in the animation clip. |
| **Once** | Play the clip to the end and then finish. |
| **Loop** | Play to the end, then immediately restart from the beginning. |

| | |
|---|---|
| **PingPong** | Play to the end, then play from the end in reverse, and so on. |
| **Forever** | Play to the end, then loop the last frame indefinitely. |
| **Anim Compression** | Settings to attempt to remove redundant information from clips:- |
| **Off** | No compression. |
| **Keyframe reduction** | Attempt to remove keyframes where differences are too small to be seen |
| **Keyframe reduction and compression** | As for *Keyframe reduction*, but clip data is also compressed. |
| **Rotation error** | Minimum difference in rotation values (in degrees), below which two keyframes are counted as equal. |
| **Position error** | Minimum difference in position (as a percentage of coordinate values), below which two keyframes are counted as equal. |
| **Rotation error** | Minimum difference in scale (as a percentage of coordinate values), below which two keyframes are counted as equal. |

Below the properties in the inspector is a list of animation clips. When you click on a clip in the list, an additional panel will appear below it in the inspector:-




The Start and End values can be changed to allow you to use just a part of the original clip (see the page on |splitting animations for further details). The *Add Loop Frame* option adds an extra keyframe to the end of the animation that is exactly the same as the keyframe at the start. This enables the animation to loop smoothly even when the last frame doesn't exactly match up with the first. The *Wrap Mode* setting is identical to the master setting in the main animation properties but applies only to that specific clip.

Page last updated: 2012-11-09

---

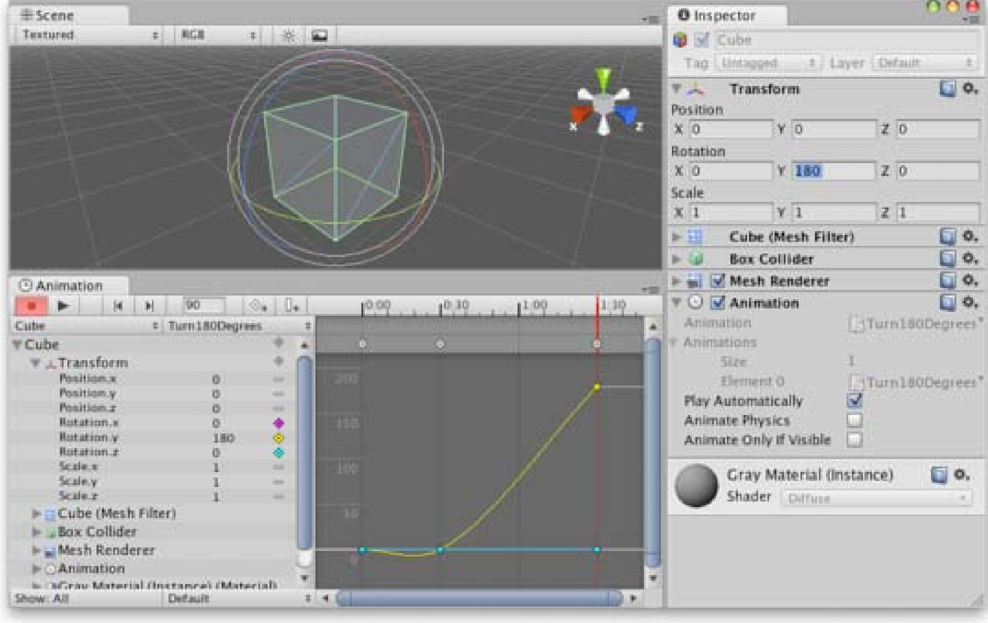
# Animation Editor Guide (Legacy)

The **Animation View** in Unity allows you to create and modify **Animation Clips** directly inside Unity. It is designed to act as a powerful and straightforward alternative to external 3D animation programs. In addition to animating movement, the editor also allows you to animate variables of materials and components and augment your Animation Clips with **Animation Events**, functions that are called at specified points along the timeline.

See the pages about Animation import and Animation Scripting for further information about these subject.

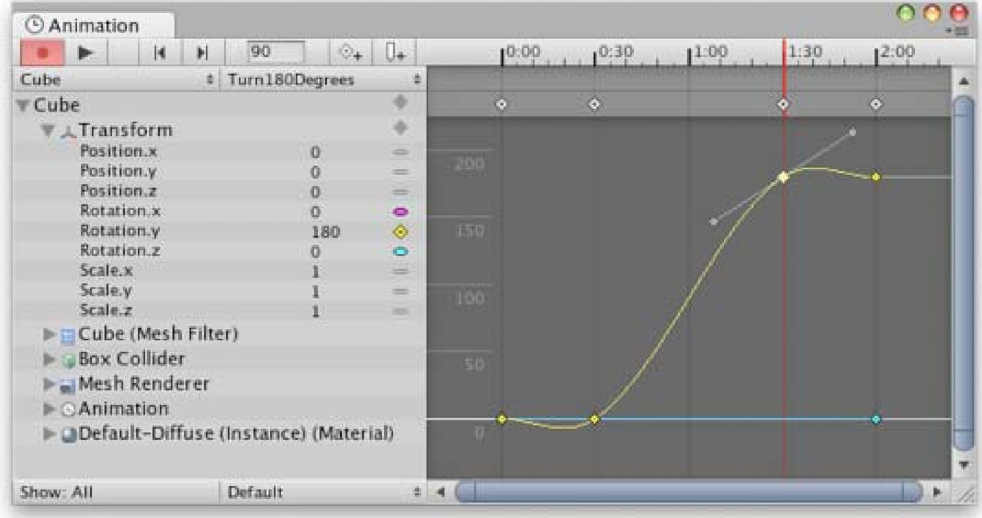
The Animation View Guide is broken up into several pages that each focus on different areas of the View:-

## Using the Animation View



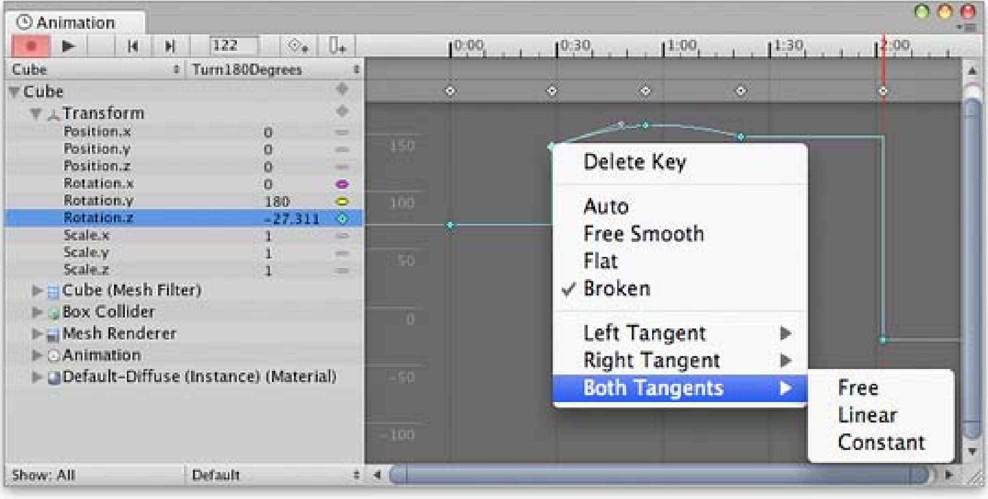

This section covers the basic operations of the **Animation View**, such as creating and editing **Animations Clips**.

## Using Animation Curves



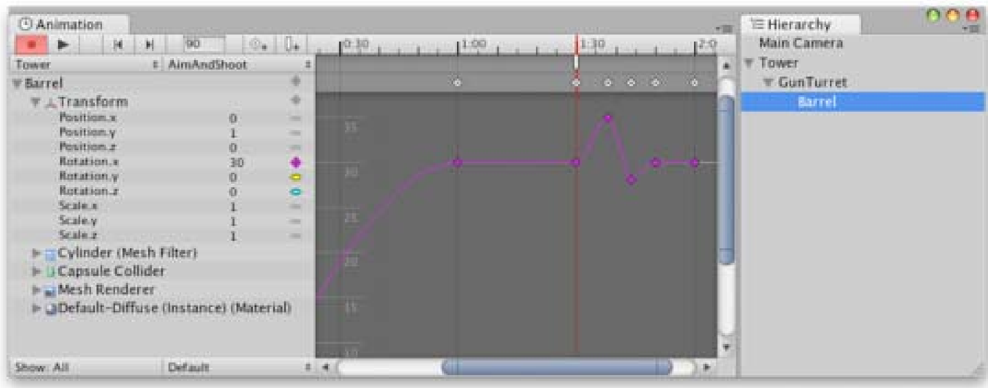

This section explains how to create **Animation Curves**, add and move **keyframes** and set WrapModes. It also offers tips for using **Animation Curves** to their full advantage.

## Editing Curves



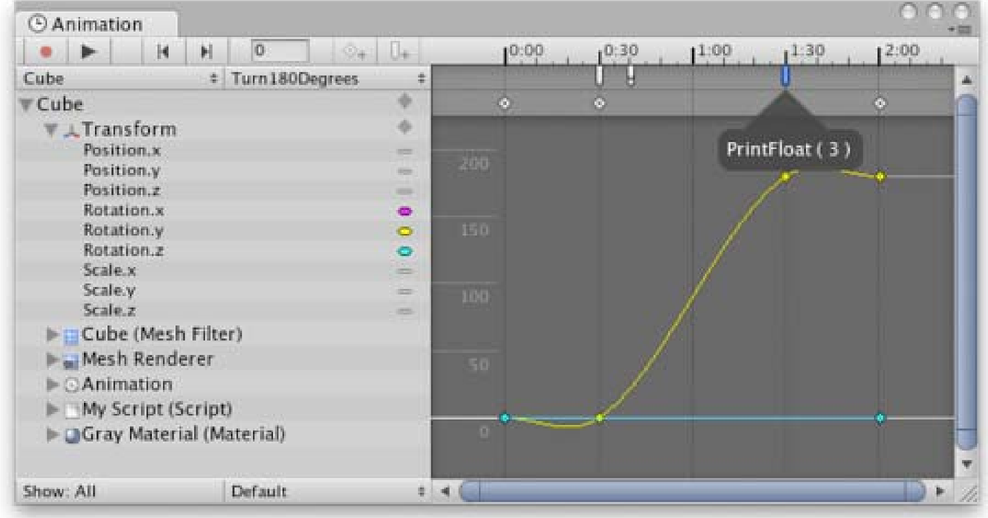

This section explains how to navigate efficienlty in the editor, create and move **keys**, and edit **tangents** and tangent types.

## Objects with Multiple Moving Parts




This section explains how to animate **Game Objects** with multiple moving parts and how to handle cases where there is more than one **Animation Component** that can control the selected **Game Object**.

## Using Animation Events




This section explains how to add **Animation Events** to an **Animation Clip**. Animation Events allow you call a script function at specified points in the animation's timeline.

Page last updated: 2012-09-10

# Animation Scripting (Legacy)

Unity's Animation System allows you to create beautifully animated skinned characters. The Animation System supports animation blending, mixing, additive animations, walk cycle time synchronization, animation layers, control over all aspects of the animation playback (time, speed, blend-weights), mesh skinning with 1, 2 or 4 bones per vertex as well as supporting physically based rag-dolls and procedural animation. To obtain the best results, it is recommended that you read about the best practices and techniques for creating a rigged character with optimal performance in Unity on the Modeling Optimized Characters page.

Making an animated character involves two things; *moving* it through the world and *animating* it accordingly. If you want to learn more about moving characters around, take a look at the Character Controller page. This page focuses on the animation. The actual animating of characters is done through Unity's scripting interface.

You can download example demos showing pre-setup animated characters. Once you have learned the basics on this page you can also see the animation script interface.

This page contains the following sections:-


- Animation Blending
- Animation Layers
- Animation Mixing
- Additive Animation
- Procedural Animation
- Animation Playback and Sampling


## Animation Blending

In today's games, animation blending is an essential feature to ensure that characters have smooth animations. Animators create separate animations, for example, a walk cycle, run cycle, idle animation or shoot animation. At any point in time during your game you need to be able to transition from the idle animation into the walk cycle and vice versa. Naturally, you want the transition to be smooth and avoid sudden jerks in the motion.

This is where animation blending comes in. In Unity you can have any number of animations playing on the same character. All animations are blended or added together to generate the final animation.

Our first step will be to make a character blend smoothly between the idle and walk animations. In order to make the scripter's job easier, we will first set the **Wrap Mode** of the animation to **Loop**. Then we will turn off **Play Automatically** to make sure our script is the only one playing animations.

Our first script for animating the character is quite simple; we only need some way to detect how fast our character is moving, and then fade between the walk and idle animations. For this simple test, we will use the standard input axes:-

```
function Update () {
  if (Input.GetAxis("Vertical") > 0.2)
     animation.CrossFade ("walk");
  else
    animation.CrossFade ("idle");
}
```

To use this script in your project:-

1. Create a Javascript file using **Assets->Create Other->Javascript**.
2. Copy and paste the code into it
3. Drag the script onto the character (it needs to be attached to the **GameObject** that has the animation)

When you hit the Play button, the character will start walking in place when you hold the up arrow key and return to the idle pose when you release it.

## Animation Layers

Layers are an incredibly useful concept that allow you to group animations and prioritize weighting.

Unity's animation system can blend between as many animation clips as you want. You can assign blend weights manually or simply use **animation.CrossFade()**, which will animate the weight automatically.

### Blend weights are always normalized before being applied

Let's say you have a walk cycle and a run cycle, both having a weight of 1 (100%). When Unity generates the final animation, it will normalize the weights, which means the walk cycle will contribute 50% to the animation and the run cycle will also contribute 50%.

However, you will generally want to prioritize which animation receives most weight when there are two animations playing. It is certainly possible to ensure that the weight sums up to 100% manually, but it is easier just to use layers for this purpose.

### Layering Example

As an example, you might have a shoot animation, an idle and a walk cycle. The walk and idle animations would be blended based on the player's speed but when the player shoots, you would want to show only the shoot animation. Thus, the shoot animation essentially has a higher priority.

The easiest way to do this is to simply keep playing the walk and idle animations while shooting. To do this, we need to make sure that the shoot animation is in a higher layer than the idle and walk animations, which means the shoot animation will receive blend weights first. The walk and idle animations will receive weights only if the shoot animation doesn't use all 100% of the blend weighting. So, when CrossFading the shoot animation in, the weight will start out at zero and over a short period become 100%. In the beginning the walk and idle layer will still receive blend weights but when the shoot animation is completely faded in, they will receive no weights at all. This is exactly what we need!

```
function Start () {
  // Set all animations to loop
  animation.wrapMode = WrapMode.Loop;
  // except shooting
  animation["shoot"].wrapMode = WrapMode.Once;

  // Put idle and walk into lower layers (The default layer is always 0)
  // This will do two things
  // - Since shoot and idle/walk are in different layers they will not affect
  //   each other's playback when calling CrossFade.
  // - Since shoot is in a higher layer, the animation will replace idle/walk
  //   animations when faded in.
  animation["shoot"].layer = 1;

  // Stop animations that are already playing
  //(In case user forgot to disable play automatically)
  animation.Stop();
}

function Update () {
  // Based on the key that is pressed,
  // play the walk animation or the idle animation
  if (Mathf.Abs(Input.GetAxis("Vertical")) > 0.1)
    animation.CrossFade("walk");
  else
    animation.CrossFade("idle");

  // Shoot
  if (Input.GetButtonDown ("Fire1"))
    animation.CrossFade("shoot");
}
```

By default the **animation.Play()** and **animation.CrossFade()** will stop or fade out animations that are in the same layer. This is exactly what we want in most cases. In our shoot, idle, run example, playing idle and run will not affect the shoot animation

and vice versa (you can change this behavior with an optional parameter to animation.CrossFade if you like).

## Animation Mixing

Animation mixing allow you to cut down on the number of animations you need to create for your game by having some animations apply to part of the body only. This means such animations can be used together with other animations in various combinations.

You add an animation mixing transform to an animation by calling **AddMixingTransform()** on the given AnimationState.

### Mixing Example

An example of mixing might be something like a hand-waving animation. You might want to make the hand wave either when the character is idle or when it is walking. Without animation mixing you would have to create separate hand waving animations for the idle and walking states. However, if you add the shoulder transform as a mixing transform to the hand waving animation, the hand waving animation will have full control only from the shoulder joint to the hand. Since the rest of the body will not be affected by he hand-waving, it will continue playing the idle or walk animation. Consequently, only the one animation is needed to make the hand wave while the rest of the body is using the idle or walk animation.

```
/// Adds a mixing transform using a Transform variable
var shoulder : Transform;
animation["wave_hand"].AddMixingTransform(shoulder);
```

Another example using a path.

```
function Start () {
   // Adds a mixing transform using a path instead
   var mixTransform : Transform = transform.Find("root/upper_body/left_shoulder");
   animation["wave_hand"].AddMixingTransform(mixTransform);
}
```

## Additive Animations

Additive animations and animation mixing allow you to cut down on the number of animations you have to create for your game, and are important for creating facial animations.

Suppose you want to create a character that leans to the sides as it turns while walking and running. This leads to four combinations (walk-lean-left, walk-lean-right, run-lean-left, run-lean-right), each of which needs an animation. Creating a separate animation for each combination clearly leads to a lot of extra work even in this simple case but the number of combinations increases dramatically with each additional action. Fortunately additive animation and mixing avoids the need to produce separate animations for combinations of simple movements.

### Additive Animation Example

Additive animations allow you to overlay the effects of one animation on top of any others that may be playing. When generating additive animations, Unity will calculate the difference between the first frame in the animation clip and the current frame. Then it will apply this difference on top of all other playing animations.

Referring to the previous example, you could make animations to lean right and left and Unity would be able to superimpose these on the walk, idle or run cycle. This could be achieved with code like the following:-

```
private var leanLeft : AnimationState;
private var leanRight : AnimationState;

function Start () {
   leanLeft = animation["leanLeft"];
   leanRight = animation["leanRight"];

   // Put the leaning animation in a separate layer
   // So that other calls to CrossFade won't affect it.
   leanLeft.layer = 10;
   leanRight.layer = 10;
```

```
    // Set the lean animation to be additive
    leanLeft.blendMode = AnimationBlendMode.Additive;
    leanRight.blendMode = AnimationBlendMode.Additive;

    // Set the lean animation ClampForever
    // With ClampForever animations will not stop
    // automatically when reaching the end of the clip
    leanLeft.wrapMode = WrapMode.ClampForever;
    leanRight.wrapMode = WrapMode.ClampForever;

    // Enable the animation and fade it in completely
    // We don't use animation.Play here because we manually adjust the time
    // in the Update function.
    // Instead we just enable the animation and set it to full weight
    leanRight.enabled = true;
    leanLeft.enabled = true;
    leanRight.weight = 1.0;
    leanLeft.weight = 1.0;

    // For testing just play "walk" animation and loop it
    animation["walk"].wrapMode = WrapMode.Loop;
    animation.Play("walk");
}

// Every frame just set the normalized time
// based on how much lean we want to apply
function Update () {
    var lean = Input.GetAxis("Horizontal");
    // normalizedTime is 0 at the first frame and 1 at the last frame in the clip
    leanLeft.normalizedTime = -lean;
    leanRight.normalizedTime = lean;
}
```

**Tip:** When using Additive animations, it is critical that you also play some other non-additive animation on every transform that is also used in the additive animation, otherwise the animations will add on top of the last frame's result. This is most certainly not what you want.

### Animating Characters Procedurally

Sometimes you want to animate the bones of your character procedurally. For example, you might want the head of your character to look at a specific point in 3D space which is best handled by a script that tracks the target point. Fortunately, Unity makes this very easy, since bones are just Transforms which drive the skinned mesh. Thus, you can control the bones of a character from a script just like the Transforms of a GameObject.

One important thing to know is that the animation system updates Transforms after the **Update()** function and before the **LateUpdate()** function. Thus if you want to do a **LookAt()** function you should do that in **LateUpdate()** to make sure that you are really overriding the animation.

Ragdolls are created in the same way. You simply have to attach Rigidbodies, Character Joints and Capsule Colliders to the different bones. This will then physically animate your skinned character.

### Animation Playback and Sampling

This section explains how animations in Unity are sampled when they are played back by the engine.

AnimationClips are typically authored at a fixed frame rate. For example, you may create your animation in 3ds Max or Maya at a frame rate of 60 frames per second (fps). When importing the animation in Unity, this frame rate will be read by the importer, so the data of the imported animation is also sampled at 60 fps.

However, games typically run at a variable frame rate. The frame rate may be higher on some computers than on others, and it may also vary from one second to the next based on the complexity of the view the camera is looking at at any given moment.

Basically this means that we can make no assumptions about the exact frame rate the game is running at. What this means is that even if an animation is authored at 60 fps, it may be played back at a different framerate, such as 56.72 fps, or 83.14 fps, or practically any other value.

As a result, Unity must sample an animation at variable framerates, and cannot guarantee the framerate for which it was originally designed. Fortunately, animations for 3D computer graphics do not consist of discrete frames, but rather of continuous curves. These curves can be sampled at any point in time, not just at those points in time that correspond to frames in the original animation. In fact, if the game runs at a higher frame rate than the animation was authored with, the animation will actually look smoother and more fluid in the game than it did in the animation software.

For most practical purposes, you can ignore the fact that Unity samples animations at variable framerates. However, if you have gameplay logic that relies on animations that animate transforms or properties into very specific configurations, then you need to be aware that the re-sampling takes place behind the scenes. For example, if you have an animation that rotates an object from 0 to 180 degrees over 30 frames, and you want to know from your code when it has reached half way there, you should not do it by having a conditional statement in your code that checks if the current rotation is 90 degrees. Because Unity samples the animation according to the variable frame rate of the game, it may sample it when the rotation is just below 90 degrees, and the next time right after it reached 90 degrees. If you need to be notified when a specific point in an animation is reached, you should use an AnimationEvent instead.

Note also that as a consequence of the variable framerate sampling, an animation that is played back using **WrapMode.Once** may not be sampled at the exact time of the last frame. In one frame of the game the animation may be sampled just before the end of the animation, and in the next frame the time can have exceeded the length of the animation, so it is disabled and not sampled further. If you absolutely need the last frame of the animation to be sampled exactly, you should use **WrapMode.ClampForever** which will keep sampling the last frame indefinitely until you stop the animation yourself.

Page last updated: 2012-09-05

---

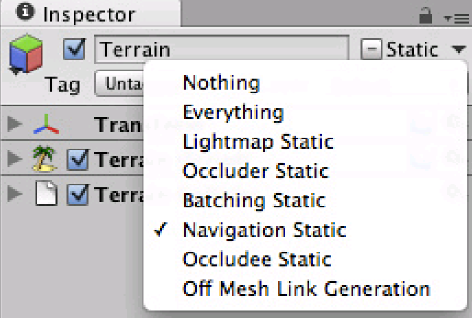
# Navmesh and Pathfinding

A navigation mesh (also known as the *Navmesh*) is a simplified representation of world geometry, which gameplay agents use to navigate the world. Typically an agent has a *goal*, or a *destination*, to which it is trying to find a path, and then navigate to that goal along the path. This process is called *pathfinding*. Note that *Navmesh generation* (or *baking*) is done by game developers inside the editor, while the *pathfinding* is done by agents at runtime based on that Navmesh.

In the complex world of games, there can be many agents, dynamic obstacles, and constantly changing accessibility levels for different areas in the world. Agents need to react dynamically to those changes. An agent's pathfinding task can be interrupted by or affected by things like collision avoidance with other characters, changing characteristics of the terrain, physical obstacles (such as closing doors), and an update to the actual destination.

Here is a simple example of how to set up a navmesh, and an agent that will do pathfinding on it:

- Create some geometry in the level, for example a **Plane** or a **Terrain**.
- In the Inspector Window's right hand corner click on **Static** and make sure that this geometry is marked up as **Navigation Static**



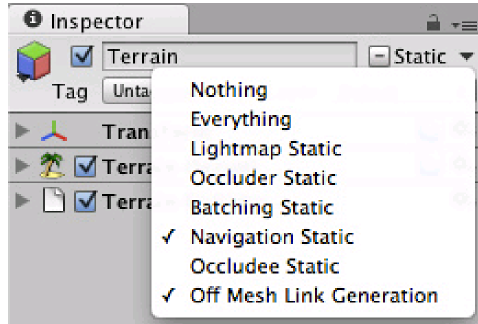

- Pull up the Navigation Mesh window (**Window**->**Navigation**).
- Bake the mesh. This will generate the navmesh for all **navigation-static** geometry.
- Create some dynamic geometry in the scene (such as characters).

- Set up an agent (or multiple agents), by adding a NavMeshAgent component to a dynamic geometry in the scene.
- Give the agent a destination (by setting the *destination* property) in a script attached to the agent.
- Press play and watch the magic.

Note that it is also possible to define custom NavMesh layers. These are needed for situations where some parts of the environment are easier for agents to pass through than others. For parts of the mesh that are not directly connected, it is possible to create Off Mesh Links.

### Automatic off-mesh links

Navmesh geometry can also be marked up for automatic off-mesh link generation, like this:




*Marking up geometry for automatic off-mesh link generation*

Geometry marked up in this way will be checked during the Navmesh Baking process for creating links to other Navmesh geometry. This way, we can control the auto-generation for each GameObject. Whether an off-mesh link will be auto-generated in the baking process is also determined by the **Jump distance** and the **Drop height** properties in the **Navigation Bake** settings.

The NavMeshLayer assigned to auto-generated off-mesh links, is the built-in layer **Jump**. This allows for global control of the auto-generated off-mesh links costs (see Navmesh layers).

Note, that there is also a possibility for setting up *manual* off-mesh links (described here).

Page last updated: 2012-04-24

---

# Navmesh Baking

Once the Navmesh geometry and layers are marked up, it's time to bake the Navmesh geometry.

Inside the Navigation window (**Window->Navigation**), go to the **Bake** tab (the upper-right corner), and click on the **Bake** button (the lower-right corner).

*Navigation Bake Window*

Here are the properties that affect Navmesh baking:

| | |
|---|---|
| **Radius** | radius of the "typical" agent (preferrably the smallest). |
| **Height** | height of the "typical" agent (the "clearance" needed to get a character through). |
| **Max Slope** | all surfaces with higher slope than this, will be discarded. |
| **Step height** | the height difference below which navmesh regions are considered connected. |
| **Drop height** | If the value of this property is positive, off-mesh links will be placed for adjacent navmesh surfaces where the height difference is below this value. |
| **Jump distance** | If the value of this property is positive, off-mesh links will be placed for adjacent navmesh surfaces where the horizontal distance is below this value. |
| **Advanced** | |
| **Min region area** | Regions with areas below this threshold will be discarded. |
| **Width inaccuracy %** | Allowable width inaccuracy |
| **Height inaccuracy %** | Allowable height inaccuracy |
| **Height mesh** | If this options is on, original height information is stored. This has performance implications for speed and memory usage. |

Note that the baked navmesh is part of the scene and agents will be able to traverse it. To remove the navmesh, click on **Clear** when you're in the **Bake** tab.

(back to Navigation and Pathfinding)

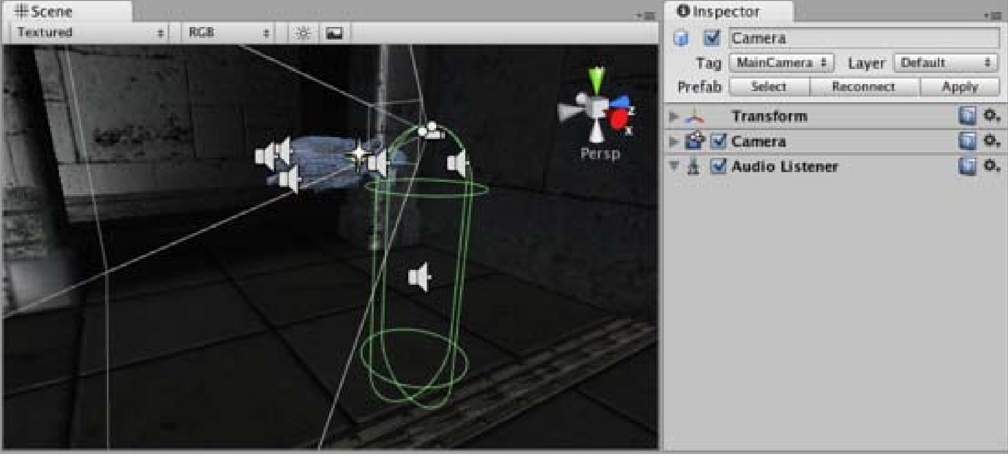
Page last updated: 2012-04-24

---

# Sound

## Audio Listener

The **Audio Listener** acts as a microphone-like device. It receives input from any given Audio Source in the scene and plays sounds through the computer speakers. For most applications it makes the most sense to attach the listener to the Main Camera. If an audio listener is within the boundaries of a Reverb Zone reverberation is applied to all audible sounds in the scene. (PRO only) Furthermore, Audio Effects can be applied to the listener and it will be applied to all audible sounds in the scene.



*The Audio Listener, attached to the Main Camera*

### Properties

The Audio Listener has no properties. It simply must be added to work. It is always added to the Main Camera by default.

### Details

The Audio Listener works in conjunction with Audio Sources, allowing you to create the aural experience for your games. When the Audio Listener is attached to a **GameObject** in your scene, any Sources that are close enough to the Listener will be picked up and output to the computer's speakers. Each scene can only have 1 Audio Listener to work properly.

If the Sources are 3D (see import settings in Audio Clip), the Listener will emulate position, velocity and orientation of the sound in the 3D world (You can tweak attenuation and 3D/2D behavior in great detail in Audio Source) . 2D will ignore any 3D processing. For example, if your character walks off a street into a night club, the night club's music should probably be 2D, while the individual voices of characters in the club should be mono with their realistic positioning being handled by Unity.

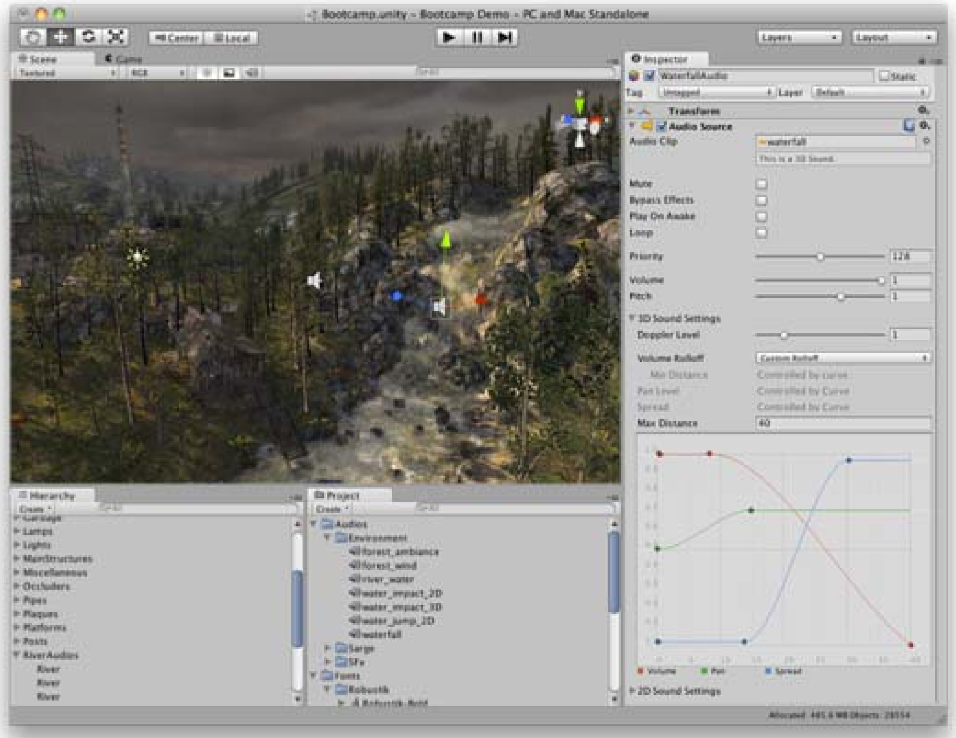
You should attach the Audio Listener to either the Main Camera or to the GameObject that represents the player. Try both to find what suits your game best.

### Hints

- Each scene can only have one Audio Listener.
- You access the project-wide audio settings using the Audio Manager, found in the **Edit->Project Settings->Audio** menu.
- View the Audio Clip Component page for more information about Mono vs Stereo sounds.

## Audio Source

The **Audio Source** plays back an Audio Clip in the scene. If the Audio Clip is a 3D clip, the source is played back at a given position and will attenuate over distance. The audio can be spread out between speakers (stereo to 7.1) (*Spread*) and morphed between 3D and 2D (*PanLevel*). This can be controlled over distance with falloff curves. Also, if the listener is within one or multiple Reverb Zones, reverberations is applied to the source. (PRO only) Individual filters can be applied to each audio source for an even richer audio experience. See Audio Effects for more details.



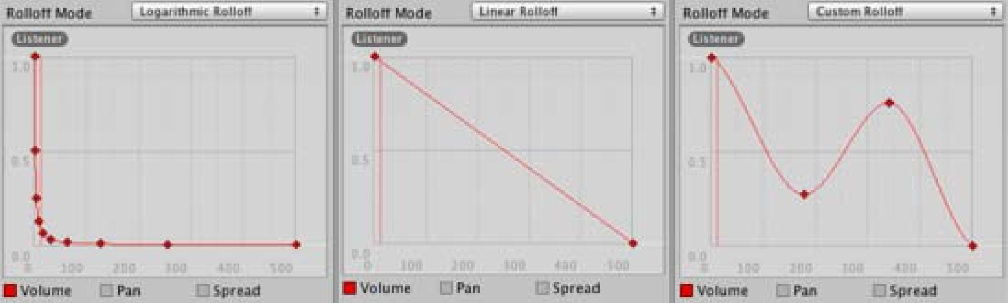
*The Audio Source gizmo in the **Scene View** and its settings in the **inspector**.*

## Properties

| | |
|---|---|
| **Audio Clip** | Reference to the sound clip file that will be played. |
| **Mute** | If enabled the sound will be playing but muted. |
| **Bypass Effects** | This Is to quickly "by-pass" filter effects applied to the audio source. An easy way to turn all effects on/off. |
| **Play On Awake** | If enabled, the sound will start playing the moment the scene launches. If disabled, you need to start it using the **Play()** command from scripting. |
| **Loop** | Enable this to make the **Audio Clip** loop when it reaches the end. |
| **Priority** | Determines the priority of this audio source among all the ones that coexist in the scene. (Priority: 0 = most important. 256 = least important. Default = 128.). Use 0 for music tracks to avoid it getting occasionally swapped out. |
| **Volume** | How loud the sound is at a distance of one world unit (one meter) from the **Audio Listener**. |
| **Pitch** | Amount of change in pitch due to slowdown/speed up of the **Audio Clip**. Value 1 is normal playback speed. |
| **3D Sound Settings** | Settings that are applied to the audio source if the Audio Clip is a 3D Sound. |
| **Pan Level** | Sets how much the 3d engine has an effect on the audio source. |
| **Spread** | Sets the spread angle to 3d stereo or multichannel sound in speaker space. |
| **Doppler Level** | Determines how much doppler effect will be applied to this audio source (if is set to 0, then no effect is applied). |
| **Min Distance** | Within the MinDistance, the sound will stay at loudest possible. Outside MinDistance it will begin to attenuate. Increase the MinDistance of a sound to make it 'louder' in a 3d world, and decrease it to make it 'quieter' in a 3d world. |
| **Max Distance** | The distance where the sound stops attenuating at. Beyond this point it will stay at the volume it would be at MaxDistance units from the listener and will not attenuate any more. |
| **Rolloff Mode** | How fast the sound fades. The higher the value, the closer the Listener has to be before hearing the sound.(This is determined by a Graph). |
| **Logarithmic Rolloff** | The sound is loud when you are close to the audio source, but when you get away from the object it decreases significantly fast. |
| **Linear Rolloff** | The further away from the audio source you go, the less you can hear it. |
| **Custom Rolloff** | The sound from the audio source behaves accordingly to how you set the graph of roll offs. |
| **2D Sound Settings** | Settings that are applied to the audio source if the Audio clip is a 2D Sound. |
| **Pan 2D** | Sets how much the engine has an effect on the audio source. |

## Types of Rolloff

There are three Rolloff modes: Logarithmic, Linear and Custom Rolloff. The Custom Rolloff can be modified by modifying the volume distance curve as described below. If you try to modify the volume distance function when it is set to Logarithmic or Linear, the type will automatically change to Custom Rolloff.

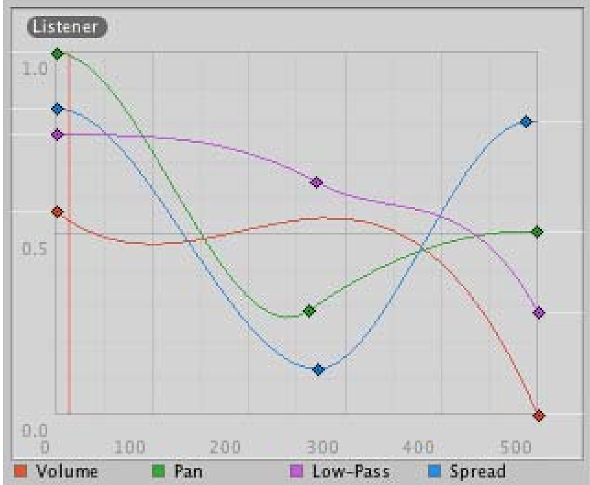


*Rolloff Modes that an audio source can have.*

### Distance Functions

There are several properties of the audio that can be modified as a function of the distance between the audio source and the audio listener.

**Volume**: Amplitude(0.0 - 1.0) over distance.
**Pan**: Left(-1.0) to Right(1.0) over distance.
**Spread**: Angle (degrees 0.0 - 360.0) over distance.
**Low-Pass** (only if LowPassFilter is attached to the AudioSource): Cutoff Frequency (22000.0-10.0) over distance.



*Distance functions for Volume, Pan, Spread and Low-Pass audio filter. The current distance to the Audio Listener is marked in the graph.*

To modify the distance functions, you can edit the curves directly. For more information, see the guide to Editing Curves.

### Creating Audio Sources

Audio Sources don't do anything without an assigned **Audio Clip**. The Clip is the actual sound file that will be played back. The Source is like a controller for starting and stopping playback of that clip, and modifying other audio properties.

To create a new Audio Source:

1. Import your audio files into your Unity Project. These are now Audio Clips.
2. Go to **GameObject->Create Empty** from the menubar.
3. With the new GameObject selected, select **Component->Audio->Audio Source**.
4. Assign the **Audio Clip** property of the Audio Source Component in the Inspector.

**Note:** If you want to create an **Audio Source** just for one **Audio Clip** that you have in the Assets folder then you can just drag that clip to the scene view - a GameObject with an **Audio Source** component will be created automatically for it. Dragging a clip onto on existing GameObject will attach the clip along with a new **Audio Source** if there isn't one already there. If the object does already have an **Audio Source** then the newly dragged clip will replace the one that the source currently uses.

#### Platform specific details

## iOS

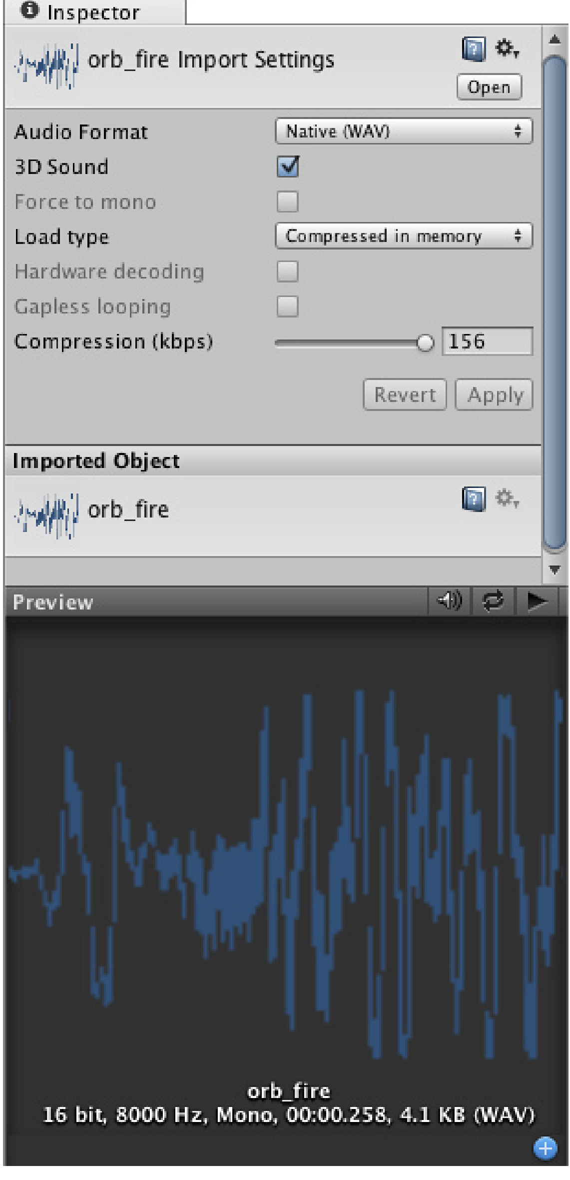
On mobile platforms compressed audio is encoded as MP3 for speedier decompression. Beware that this compression can remove samples at the end of the clip and potentially break a "perfect-looping" clip. Make sure the clip is right on a specific MP3 sample boundary to avoid sample clipping - tools to perform this task are widely available. For performance reasons audio clips can be played back using the Apple hardware codec. To enable this, check the "Use Hardware" checkbox in the import settings. See the Audio Clip documentation for more details.

## Android

On mobile platforms compressed audio is encoded as MP3 for speedier decompression. Beware that this compression can remove samples at the end of the clip and potentially break a "perfect-looping" clip. Make sure the clip is right on a specific MP3 sample boundary to avoid sample clipping - tools to perform this task are widely available.

# Audio Clip

**Audio Clips** contain the audio data used by Audio Sources. Unity supports mono, stereo and multichannel audio assets (up to eight channels). The audio file formats that Unity can import are **.aif**, **.wav**, **.mp3**, and **.ogg**. Unity can also import tracker modules in the **.xm**, **.mod**, **.it**, and **.s3m** formats. The tracker module assets behave the same way as any other audio assets in Unity although no waveform preview is available in the asset import inspector.



*The Audio Clip* ***Inspector***

### Properties

| | |
|---|---|
| **Audio Format** | The specific format that will be used for the sound at runtime. |
| **Native** | This option offers higher quality at the expense of larger file size and is best for very short sound effects. |
| **Compressed** | The compression results in smaller files but with somewhat lower quality compared to native audio. This format is best for medium length sound effects and music. |
| **3D Sound** | If enabled, the sound will play back in 3D space. Both Mono and Stereo sounds can be played in 3D. |
| **Force to mono** | If enabled, the audio clip will be down-mixed to a single channel sound. |
| **Load Type** | The method Unity uses to load audio assets at runtime. |
| **Decompress on load** | Audio files will be decompressed as soon as they are loaded. Use this option for smaller compressed sounds to avoid the performance overhead of decompressing on the fly. Be aware that decompressing sounds on load will use about ten times more memory than keeping them compressed, so don't use this option for large files. |
| **Compressed in memory** | Keep sounds compressed in memory and decompress while playing. This option has a slight performance overhead (especially for Ogg/Vorbis compressed files) so only use it for bigger files where decompression on load would use a prohibitive amount of memory. Note that, due to technical limitations, this option will silently switch to *Stream From Disc* (see below) for Ogg Vorbis assets on platforms that use FMOD audio. |
| **Stream from disc** | Stream audio data directly from disc. The memory used by this option is typically a small fraction of the file size, so it is very useful for music or other very long tracks. For performance reasons, it is usually advisable to stream only one or two files from disc at a time but the of streams that can comfortably be handled depends on the hardware. |
| **Compression** | Amount of Compression to be applied to a **Compressed** clip. Statistics about the file size can be seen under the slider. A good approach to tuning this value is to drag the slider to a place that leaves the playback "good enough" while keeping the file small enough for your distribution requirements. |
| **Hardware Decoding** | (iOS only) On iOS devices, Apple's hardware decoder can be used resulting in lower CPU overhead during decompression. Check out platform specific details for more info. |
| **Gapless looping** | (Android/iOS only) Use this when compressing a seamless looping audio source file (in a non-compressed PCM format) to ensure perfect continuity is preserved at the seam. Standard MPEG encoders introduce a short silence at the loop point, which will be audible as a brief "click" or "pop". |

### Importing Audio Assets

Unity supports both *Compressed* and *Native* Audio. Any type of file (except MP3/Ogg Vorbis) will be initially imported as *Native*. Compressed audio files must be decompressed by the CPU while the game is running, but have smaller file size. If *Stream* is checked the audio is decompressed *on the fly*, otherwise it is decompressed completely as soon as it loads. Native PCM formats (WAV, AIFF) have the benefit of giving higher fidelity without increasing the CPU overhead, but files in these formats are typically much larger than compressed files. Module files (.mod,.it,.s3m..xm) can deliver very high quality with an extremely low footprint.

As a general rule of thumb, *Compressed* audio (or modules) are best for long files like background music or dialog, while *Native* is better for short sound effects. You should tweak the amount of Compression using the compression slider. Start with high compression and gradually reduce the setting to the point where the loss of sound quality is perceptible. Then, increase it again slightly until the perceived loss of quality disappears.

### Using 3D Audio

If an audio clip is marked as a **3D Sound** then it will be played back so as to simulate its position in the game world's 3D space. 3D sounds emulate the distance and location of sounds by attenuating volume and panning across speakers. Both mono and multiple channel sounds can be positioned in 3D. For multiple channel audio, use the *spread* option on the Audio Source to spread and split out the discrete channels in speaker space. Unity offers a variety of options to control and fine-tune the audio behavior in 3D space - see the Audio Source component reference for further details.

### Platform specific details

## iOS

On mobile platforms compressed audio is encoded as MP3 to take advantage of hardware decompression.

To improve performance, audio clips can be played back using the Apple hardware codec. To enable this option, check the "Hardware Decoding" checkbox in the Audio Importer. Note that only one hardware audio stream can be decompressed at a

time, including the background iPod audio.

If the hardware decoder is not available, the decompression will fall back on the software decoder (on iPhone 3GS or later, Apple's software decoder is used in preference to Unity's own decoder (FMOD)).

## Android

On mobile platforms compressed audio is encoded as MP3 to take advantage of hardware decompression.

Page last updated: 2007-11-16

---

# Game Interface Elements

Unity gives you a number of options for creating your game's graphic user interface (GUI). You can use GUI Text and GUI Texture objects in the scene, or generate the interface from scripts using **UnityGUI**.

The rest of this page contains a detailed guide for getting up and running with UnityGUI.

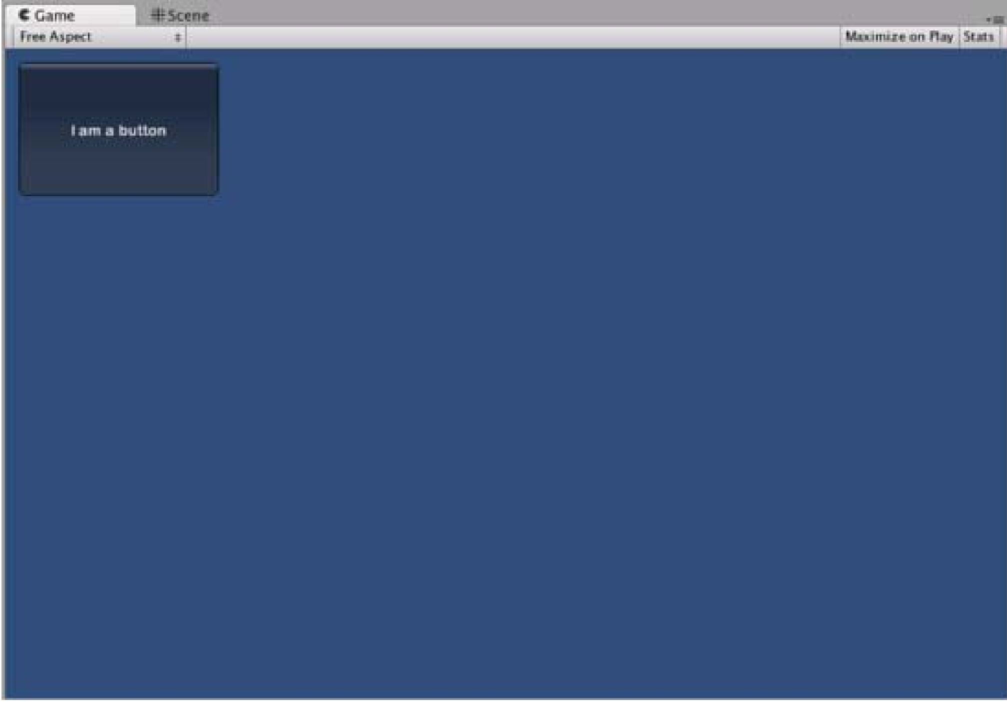
# GUI Scripting Guide

### Overview

UnityGUI allows you to create a wide variety of highly functional GUIs very quickly and easily. Rather than creating a GUI object, manually positioning it, and then writing a script that handles its functionality, you can do everything at once with just a few lines of code. The code produces **GUI controls** that are instantiated, positioned and handled with a single function call.

For example, the following code will create and handle a button with no additional work in the editor or elsewhere:-

```
// JavaScript
function OnGUI () {
	if (GUI.Button (Rect (10,10,150,100), "I am a button")) {
		print ("You clicked the button!");
	}
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	void OnGUI () {
		if (GUI.Button (new Rect (10,10,150,100), "I am a button")) {
			print ("You clicked the button!");
		}
	}
}
```




*This is the button created by the code above*

Although this example is very simple, there are very powerful and complex techniques available for use in UnityGUI. GUI construction is a broad subject but the following sections should help you get up to speed as quickly as possible. This guide can be read straight through or used as reference material.

### UnityGUI Basics

This section covers the fundamental concepts of UnityGUI, giving you an overview as well as a set of working examples you can paste into your own code. UnityGUI is very friendly to play with, so this is a good place to get started.

### Controls

This section lists every available Control in UnityGUI, along with code samples and images showing the results.

### Customization

It is important to be able to change the appearance of the GUI to match the look of your game. All controls in UnityGUI can be customized with **GUIStyles** and **GUISkins**, as explained in this section.

### Layout Modes

UnityGUI offers two ways to arrange your GUIs: you can manually place each control on the screen, or you can use an automatic layout system which works in a similar way to HTML tables. Either system can be used as desired and the two can be freely mixed. This section explains the functional differences between the two systems, including examples.

### Extending UnityGUI

UnityGUI is very easy to extend with new Control types. This chapter shows you how to make simple *compound* controls - complete with integration into Unity's event system.

### Extending Unity Editor

The GUI of the Unity editor is actually written using UnityGUI. Consequently, the editor is highly extensible using the same type of code you would use for in-game GUI. In addition, there are a number of Editor-specific GUI controls to help you create custom editor GUI.

Page last updated: 2011-11-17

---

## Networked Multiplayer

Realtime networking is a complex field but Unity makes it easy to add networking features to your game. Nevertheless, it is

useful to have some idea of the scope of networking before using it in a game. This section explains the fundamentals of networking along with the specifics of Unity's implementation. If you have never created a network game before then it is strongly recommended that you work through this guide before getting started.

### High Level Overview

This section outlines all the concepts involved in networking and serves as an introduction to deeper topics.

### Networking Elements in Unity

This section of the guide covers Unity's implementation of the concepts explained in the overview.

#### RPC Details

Remote Procedure Call or RPC is a way of calling a function on a remote machine. This may be a client calling a function on the server, or the server calling a function on some or all clients. This section explains RPC concepts in detail.

#### State Synchronization

State Synchronization is a method of regularly updating a specific set of data across two or more game instances running on the network.

#### Minimizing Bandwidth

Every choice you make about where and how to share data will affect the network bandwidth your game uses. This page explains how bandwidth is used and how to keep usage to a minimum.

#### Network View

Network Views are Components you use to share data across the network and are a fundamental aspect of Unity networking. This page explains them in detail.

#### Network Instantiate

A complex subject in networking is ownership of an object and determination of who controls what. Network Instantiation handles this task for you, as explained in this section. Also covered are some more sophisticated alternatives for situations where you need more control over object ownership.

#### Master Server

The Master Server is like a game lobby where servers can advertise their presence to clients. It can also enable communication from behind a firewall or home network using a technique called NAT punchthrough (with help from a facilitator) to make sure your players can always connect with each other. This page explains how to use the Master Server.

Page last updated: 2011-11-17

---

## iphone-GettingStarted

Building games for devices like the iPhone and iPad requires a different approach than you would use for desktop PC games. Unlike the PC market, your target hardware is standardized and not as fast or powerful as a computer with a dedicated video card. Because of this, you will have to approach the development of your games for these platforms a little differently. Also, the features available in Unity for iOS differ slightly from those for desktop PCs.

### Setting Up Your Apple Developer Account

Before you can run Unity iOS games on the actual device, you will need to have your Apple Developer account approved and set up. This includes establishing your team, adding your devices, and finalizing your provisioning profiles. All this setup is performed through Apple's developer website. Since this is a complex process, we have provided a basic outline of the tasks that must be completed before you can run code on your iOS devices. However, the best thing to do is follow the step-by-step instructions at Apple's iPhone Developer portal.

**Note**: We recommend that you set up your Apple Developer account before proceeding because you will need it to use Unity to its full potential with iOS.

### Accessing iOS Functionality

Unity provides a number of scripting APIs to access the multi-touch screen, accelerometer, device geographical location

system and much more. You can find out more about the script classes on the iOS scripting page.

### Exposing Native C, C++ or Objective-C Code to Scripts

Unity allows you to call custom native functions written in C, C++ or Objective-C directly from C# scripts. To find out how to bind native functions, visit the plugins page.

### Prepare Your Application for In-App Purchases

The Unity iOS runtime allows you to download new content and you can use this feature to implement in-app purchases. See the downloadable content manual page for further information.

### Occlusion Culling

Unity supports *occlusion culling* which is useful for squeezing high performance out of complex scenes with many objects. See the occlusion culling manual page for further information.

### Splash Screen Customization

See the splash screen customization page to find out how to change the image your game shows while launching.

### Troubleshooting and Reporting Crashes.

If you are experiencing crashes on the iOS device, please consult the iOS troubleshooting page for a list of common issues and solutions. If you can't find a solution here then please file a bug report for the crash (menu: **Help > Report A Bug** in the Unity editor).

### How Unity's iOS and Desktop Targets Differ

#### Statically Typed JavaScript

Dynamic typing in JavaScript is always turned off in Unity when targetting iOS (this is equivalent to `#pragma strict` getting added to all your scripts automatically). Static typing greatly improves performance, which is especially important on iOS devices. When you switch an existing Unity project to the iOS target, you will get compiler errors if you are using dynamic typing. You can easily fix these either by using explicitly declared types for the variables that are causing errors or taking advantage of type inference.

#### MP3 Instead of Ogg Vorbis Audio Compression

For performance reasons, MP3 compression is favored on iOS devices. If your project contains audio files with Ogg Vorbis compression, they will be re-compressed to MP3 during the build. Consult the audio clip documentation for more information on using compressed audio on the iPhone.

#### PVRTC Instead of DXT Texture Compression

Unity iOS does not support DXT textures. Instead, PVRTC texture compression is natively supported by iPhone/iPad devices. Consult the texture import settings documentation to learn more about iOS texture formats.

#### Movie Playback

MovieTextures are not supported on iOS. Instead, full-screen streaming playback is provided via scripting functions. To learn about the supported file formats and scripting API, consult the movie page in the manual.

### Further Reading

- Unity iOS Basics
- Unity Remote
- iOS Scripting
  - Input
  - Mobile Keyboard
  - Advanced Unity Mobile Scripting
  - Using .NET API 2.0 compatibility level
- iOS Hardware Guide
- Optimizing Performance in iOS.
  - iOS Specific Optimizations
  - Measuring Performance with the Built-in Profiler
  - Optimizing the Size of the Built iOS Player
- Account Setup
- Features currently not supported by Unity iOS
- Building Plugins for iOS
- Preparing your application for "In App Purchases"

- Customizing the Splash screen of Your Mobile Application
- Trouble Shooting
- Reporting crash bugs on iOS

Page last updated: 2012-06-06

---

# iphone-basic

This section covers the most common and important questions that come up when starting to work with iOS.

## Prerequisites

**I've just received iPhone Developer approval from Apple, but I've never developed for iOS before. What do I do first?**
**A**: Download the SDK, get up and running on the Apple developer site, and set up your team, devices, and provisioning. We've provided a basic list of steps to get you started.

**Can Unity-built games run in the iPhone Simulator?**
**A**: No, but Unity iOS can build to iPad Simulator if you're using the latest SDK. However the simulator itself is not very useful for Unity because it does not simulate all inputs from iOS or properly emulate the performance you get on the iPhone/iPad. You should test out gameplay directly inside Unity using the iPhone/iPad as a remote control while it is running the Unity Remote application. Then, when you are ready to test performance and optimize the game, you publish to iOS devices.

## Unity Features

**How do I work with the touch screen and accelerometer?**
**A**: In the scripting reference inside your Unity iOS installation, you will find classes that provide the hooks into the device's functionality that you will need to build your apps. Consult the Input System page for more information.

**My existing particle systems seem to run very slowly on iOS. What should I do?**
**A**: iOS has relatively low fillrate. If your particles cover a rather large portion of the screen with multiple layers, it will kill iOS performance even with the simplest shader. We suggest baking your particle effects into a series of textures off-line. Then, at run-time, you can use 1-2 particles to display them via animated textures. You can get fairly decent looking effects with a minimum amount of overdraw this way.

**Can I make a game that uses heavy physics?**
**A**: Physics can be expensive on iOS is it requires a lot of floating point number crunching. You should completely avoid MeshColliders if at all possible, but they can be used if they are really necessary. To improve performance, use a low fixed framerate using **Edit->Time->Fixed Delta Time**. A framerate of 10-30 is recommended. Enable rigidbody interpolation to achieve smooth motion while using low physics frame rates. In order to achieve completely fluid framerate without oscillations, it is best to pick fixed `deltaTime` value based on the average framerate your game is getting on iOS. Either 1:1 or half the frame rate is recommended. For example, if you get 30 fps, you should use 15 or 30 fps for fixed frame rate (0.033 or 0.066)

**Can I access the gallery, music library or the native iPod player in Unity iOS?**
**A**: Yes - if you implement it. Unity iPhone supports the native plugin system, where you can add any feature you need -- including access to Gallery, Music library, iPod Player and any other feature that the iOS SDK exposes. Unity iOS does not provide an API for accessing the listed features through Unity scripts.

## UnityGUI Considerations

**What kind of performance impact will UnityGUI make on my games?**
**A**: UnityGUI is fairly expensive when many controls are used. It is ideal to limit your use of UnityGUI to game menus or very minimal GUI Controls while your game is running. It is important to note that every object with a script containing an `OnGUI()` call will require additional processor time -- even if it is an empty `OnGUI()` block. It is best to disable any scripts that have an `OnGUI()` call if the GUI Controls are not being used. You can do this by marking the script as `enabled = false`.

**Any other tips for using UnityGUI?**
**A**: Try using GUILayout as little as possible. If you are not using GUILayout at all from one `OnGUI()` call, you can disable all GUILayout rendering using `MonoBehaviour.useGUILayout = false;` This doubles GUI rendering performance. Finally,

use as few GUI elements while rendering 3D scenes as possible.

Page last updated: 2011-10-29

---

## unity-remote

Unity Remote is an application that allows you to use your iOS device as a remote control for your project in Unity. This is useful during development since it is much quicker to test your project in the editor with remote control than to build and deploy it to the device after each change.

### Where can I find Unity Remote?

Unity remote is available for download from the AppStore at no charge. If you prefer to build and deploy the application yourself, you can download the source here at the Unity website.

### How do I build Unity Remote?

First, download the project source code here and unzip it to your preferred location. The zip file contains an XCode project to build Unity Remote and install it on your device.

Assuming you have already created the provisioning profile and successfully installed iOS builds on your device, you just need to open the Xcode project file UnityRemote.xcodeproj. Once XCode is launched, you should click "Build and Go" to install the app on your iOS device. If you have never built and run applications before, we recommend that you try building some of the Apple examples first to familiarize yourself with XCode and iOS.

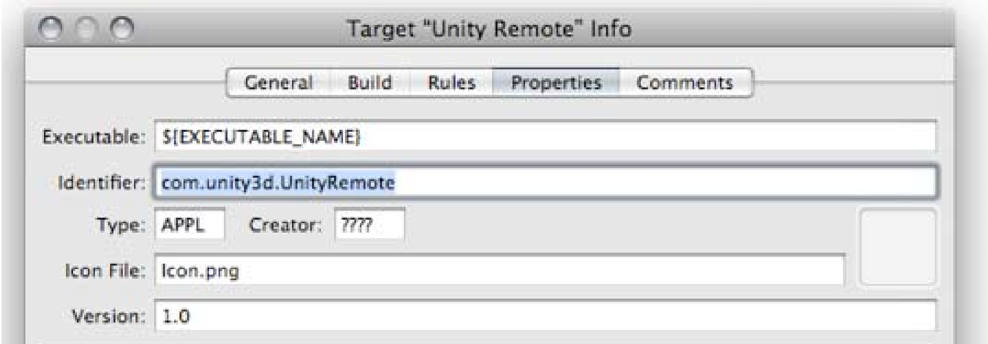
Once Unity Remote is installed, make sure your device is connected via Wi-Fi to the same network as your development machine. Launch Unity Remote on your iPhone/iPad while Unity is running on your computer and select your computer from the list that appears. Now, whenever you enter Play mode in the Editor, your device will act as a remote control that you can use for developing and testing your game. You can control the application with the device wirelessly and you will also see a low-res version of the app on the device's screen.

**Note:** The Unity iOS editor cannot emulate the device's hardware perfectly, so you may not get the exact behavior (graphics performance, touch responsiveness, sounds playback, etc) that you would on a real device.

### Xcode shows strange errors while deploying Unity Remote to my device. What should I do?

This indicates that the default Identifier in the Unity Remote project is not compatible with your provisioning profile. You will have to alter this Identifier manually in your XCode project. The Identifier must match your provisioning profile.

You will need to create an AppID with an trailing asterisk if you have not already done so; you can do this in the Program Portal on Apple's iPhone Developer Program. First, go to the Program Portal and choose the AppIDs tab. Then, click the Add ID button in the top right corner and type your usual bundle identifier followed by dot and asterisk (eg, com.mycompany.*) in the App ID Bundle Seed ID and Bundle Identifier field. Add the new AppID to your provisioning profile, then download and reinstall it. Don't forget to restart Xcode afterwards. If you have any problems creating the AppID, consult the Provisioning How-to section on Apple's website.




*Don't forget to change the Identifier before you install Unity Remote on your device.*

Open the Unity Remote project with XCode. From the menu, select **Project->Edit Active Target "Unity Remote"**. This will open a new window entitled Target "Unity Remote" Info. Select the Properties tab. Change the Identifier property field from

**com.unity3d.UnityRemote** to the bundle identifier in your AppID followed by "**.**" (dot) followed by "**UnityRemote**". For example, if your provisioning profile contains ##.com.mycompany.* AppID, then change the Identifier field to **com.mycompany.UnityRemote**.

Next, select **Build->Clean all targets** from the menu, and compile and install Unity Remote again. You may also need to change the active SDK from Simulator to Device - 2.0 | Release. There is no problem using SDK 2.0 even if your device runs a newer version of the OS.

### I'm getting really poor graphics quality when running my game in Unity Remote. What can I do to improve it?

When you use Unity Remote, the game actually runs on your Mac while its visual content is heavily compressed and streamed to the device. As a result, what you see on the device screen is just a low-res version of what the app would really look like. You should check how the game runs on the device occasionally by building and deploying the app (select **File->Build & Run** in the Unity editor).

### Unity Remote is laggy. Can I improve it?

The performance of Unity Remote depends heavily on the speed of the Wi-Fi network, the quality of the networking hardware and other factors. For the best experience, create an ad-hoc network between your Mac and iOS device. Click the Airport icon on your Mac and choose "Create Network". Then, enter a name and password and click OK. On the device, choose *Settings->Wi-Fi* and select the new Wi-Fi network you have just created. Remember that an ad-hoc network is really a wireless connection that does not involve a wireless access point. Therefore, you will usually not have internet access while using ad-hoc networking.

Turning Bluetooth off on both on your iPhone/iPad and on Mac should also improve connection quality.

If you do not need to see the game view on the device, you can turn image synchronization off in the Remote machine list. This will reduce the network traffic needed for the Remote to work.

### The connection to Unity Remote is easily lost

This can be due to a problem with the installation or other factors that prevent Unity Remote from functioning properly. Try the following steps in sequence, checking if the performance improves at each step before moving on to the next:-

1. First of all, check if Bluetooth is switched on. Both your Mac and iOS device should have Bluetooth disabled for best performance.
2. Delete the settings file located at ~/Library/Preferences/com.unity3d.UnityEditoriPhone.plist
3. Reinstall the game on your iPhone/iPad.
4. Reinstall Unity on your Mac.
5. As a last resort, performing a hard reset on the iOS device can sometimes improve the performance of Unity Remote.

If you still experience problems then try installing Unity Remote on another device (in another location if possible) and see if it gives you better results. There could be problems with RF interference or other software influencing the performance of the wireless adapter on your Mac or iOS device.

### Unity Remote doesn't see my Mac. What should I do?

- Check if Unity Remote and your Mac are connected to the same wireless network.
- Check your firewall settings, router security settings, and any other hardware/software that may filter packets on your network.
- Leave Unity Remote running, switch off your Mac's Airport for a minute or two, and switch on again.
- Restart both Unity and Unity Remote. Sometimes you also need to cold-restart your iPhone/iPad (hold down the menu and power buttons simultaneously).
- Unity Remote uses the Apple Bonjour service, so check that your Mac has it switched on.
- Reinstall Unity Remote from the latest Unity iOS package.

Page last updated: 2012-08-28

---

## iphone-API

Most features of the iOS devices are exposed through the Input and Handheld classes. For cross-platform projects, **UNITY_IPHONE** is defined for conditionally compiling iOS-specific C# code.

### Further Reading

- Input
- Mobile Keyboard
- Advanced Unity Mobile Scripting
- Using .NET API 2.0 compatibility level

Page last updated: 2012-11-23

---

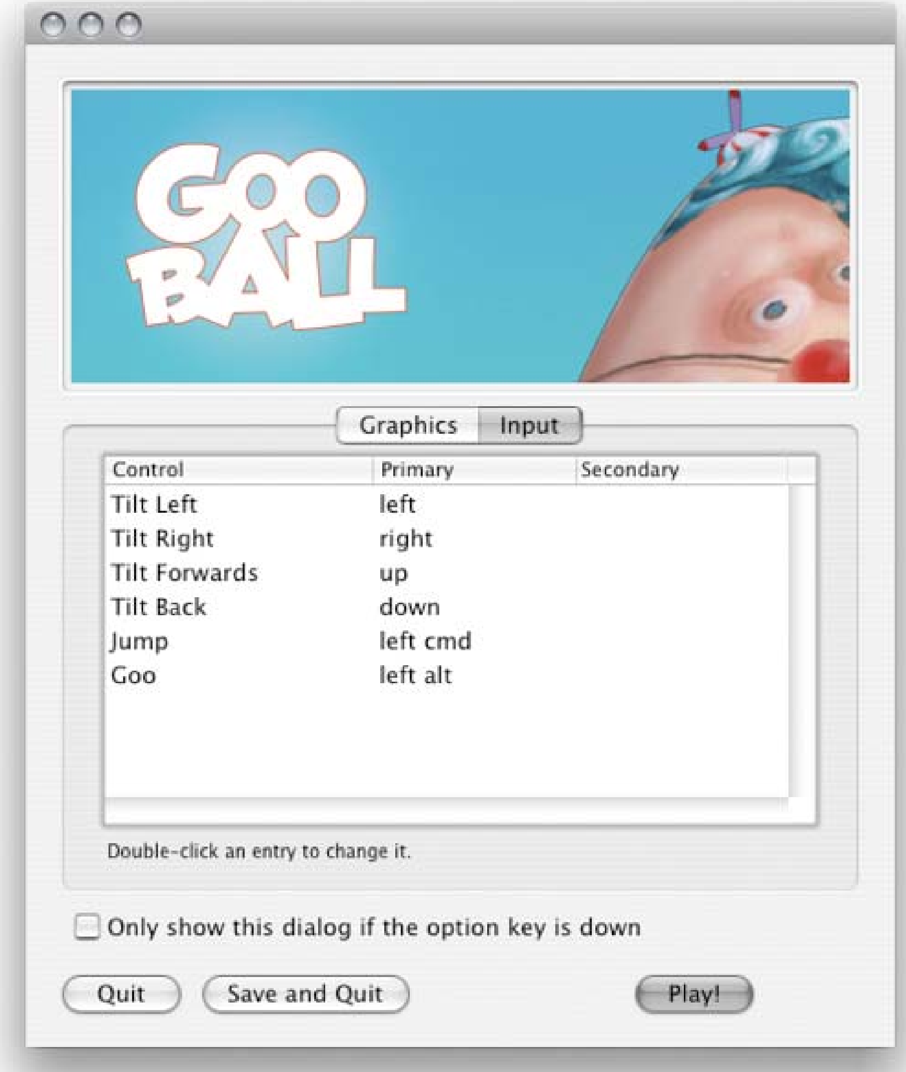
# iphone-Input

## Desktop

**Note:** Keyboard, joystick and gamepad input work on the desktop versions of Unity (including webplayer and Flash) but not on mobiles.

Unity supports keyboard, joystick and gamepad input.

Virtual axes and buttons can be created in the **Input Manager**, and end users can configure Keyboard input in a nice screen configuration dialog.




You can setup joysticks, gamepads, keyboard, and mouse, then access them all through one simple scripting interface.

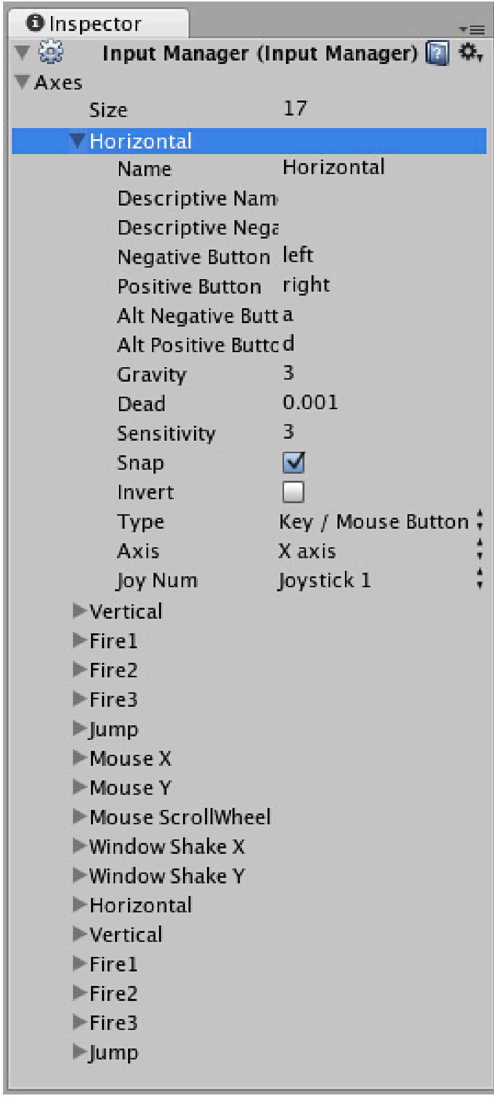
From scripts, all virtual axes are accessed by their name.

Every project has the following default input axes when it's created:

- **Horizontal** and **Vertical** are mapped to w, a, s, d and the arrow keys.
- **Fire1**, **Fire2**, **Fire3** are mapped to Control, Option (Alt), and Command, respectively.
- **Mouse X** and **Mouse Y** are mapped to the delta of mouse movement.
- **Window Shake X** and **Window Shake Y** is mapped to the movement of the window.

**Adding new Input Axes**

If you want to add new virtual axes go to the **Edit->Project Settings->Input** menu. Here you can also change the settings of each axis.




You map each axis to two buttons on a joystick, mouse, or keyboard keys.

| | |
|---|---|
| **Name** | The name of the string used to check this axis from a script. |
| **Descriptive Name** | Positive value name displayed in the input tab of the **Configuration** dialog for standalone builds. |
| **Descriptive Negative Name** | Negative value name displayed in the Input tab of the **Configuration** dialog for standalone builds. |
| **Negative Button** | The button used to push the axis in the negative direction. |
| **Positive Button** | The button used to push the axis in the positive direction. |
| **Alt Negative Button** | Alternative button used to push the axis in the negative direction. |
| **Alt Positive Button** | Alternative button used to push the axis in the positive direction. |
| **Gravity** | Speed in units per second that the axis falls toward neutral when no buttons are pressed. |
| **Dead** | Size of the analog dead zone. All analog device values within this range result map to neutral. |
| **Sensitivity** | Speed in units per second that the the axis will move toward the target value. This is for digital devices only. |
| **Snap** | If enabled, the axis value will reset to zero when pressing a button of the opposite direction. |
| **Invert** | If enabled, the **Negative Buttons** provide a positive value, and vice-versa. |
| **Type** | The type of inputs that will control this axis. |
| **Axis** | The axis of a connected device that will control this axis. |
| **Joy Num** | The connected Joystick that will control this axis. |

Use these settings to fine tune the look and feel of input. They are all documented with tooltips in the Editor as well.

**Using Input Axes from Scripts**

You can query the current state from a script like this:

```
value = Input.GetAxis ("Horizontal");
```

An axis has a value between -1 and 1. The neutral position is 0. This is the case for joystick input and keyboard input.

However, Mouse Delta and Window Shake Delta are how much the mouse or window moved during the last frame. This means it can be larger than 1 or smaller than -1 when the user moves the mouse quickly.

It is possible to create multiple axes with the same name. When getting the input axis, the axis with the largest absolute value will be returned. This makes it possible to assign more than one input device to one axis name. For example, create one axis for keyboard input and one axis for joystick input with the same name. If the user is using the joystick, input will come from the joystick, otherwise input will come from the keyboard. This way you don't have to consider where the input comes from when writing scripts.

**Button Names**
To map a key to an axis, you have to enter the key's name in the **Positive Button** or **Negative Button** property in the **Inspector**.

The names of keys follow this convention:

- Normal keys: "a", "b", "c" ...
- Number keys: "1", "2", "3", ...
- Arrow keys: "up", "down", "left", "right"
- Keypad keys: "[1]", "[2]", "[3]", "[+]", "[equals]"
- Modifier keys: "right shift", "left shift", "right ctrl", "left ctrl", "right alt", "left alt", "right cmd", "left cmd"
- Mouse Buttons: "mouse 0", "mouse 1", "mouse 2", ...
- Joystick Buttons (from any joystick): "joystick button 0", "joystick button 1", "joystick button 2", ...
- Joystick Buttons (from a specific joystick): "joystick 1 button 0", "joystick 1 button 1", "joystick 2 button 0", ...
- Special keys: "backspace", "tab", "return", "escape", "space", "delete", "enter", "insert", "home", "end", "page up", "page down"
- Function keys: "f1", "f2", "f3", ...

The names used to identify the keys are the same in the scripting interface and the Inspector.

```
value = Input.GetKey ("a");
```

## Mobile Input

On iOS and Android, the Input class offers access to touchscreen, accelerometer and geographical/location input.

Access to keyboard on mobile devices is provided via the iOS keyboard.

### Multi-Touch Screen

The iPhone and iPod Touch devices are capable of tracking up to five fingers touching the screen simultaneously. You can retrieve the status of each finger touching the screen during the last frame by accessing the Input.touches property array.

Android devices don't have a unified limit on how many fingers they track. Instead, it varies from device to device and can be anything from two-touch on older devices to five fingers on some newer devices.

Each finger touch is represented by an Input.Touch data structure:

| | |
|---|---|
| **fingerId** | The unique index for a touch. |
| **position** | The screen position of the touch. |
| **deltaPosition** | The screen position change since the last frame. |
| **deltaTime** | Amount of time that has passed since the last state change. |
| **tapCount** | The iPhone/iPad screen is able to distinguish quick finger taps by the user. This counter will let you know how many times the user has tapped the screen without moving a finger to the sides. Android devices do not count number of taps, this field is always 1. |
| **phase** | Describes so called "phase" or the state of the touch. It can help you determine if the touch just began, if user moved the finger or if he just lifted the finger. |

Phase can be one of the following:

| | |
|---|---|
| **Began** | A finger just touched the screen. |

| | |
|---|---|
| **Moved** | A finger moved on the screen. |
| **Stationary** | A finger is touching the screen but hasn't moved since the last frame. |
| **Ended** | A finger was lifted from the screen. This is the final phase of a touch. |
| **Canceled** | The system cancelled tracking for the touch, as when (for example) the user puts the device to her face or more than five touches happened simultaneously. This is the final phase of a touch. |

Following is an example script which will shoot a ray whenever the user taps on the screen:

```
var particle : GameObject;
function Update () {
	for (var touch : Touch in Input.touches) {
		if (touch.phase == TouchPhase.Began) {
			// Construct a ray from the current touch coordinates
			var ray = Camera.main.ScreenPointToRay (touch.position);
			if (Physics.Raycast (ray)) {
				// Create a particle if hit
				Instantiate (particle, transform.position, transform.rotation);
			}
		}
	}
}
```

**Mouse Simulation**

On top of native touch support Unity iOS/Android provides a mouse simulation. You can use mouse functionality from the standard Input class.

## Device Orientation

Unity iOS/Android allows you to get discrete description of the device physical orientation in three-dimensional space. Detecting a change in orientation can be useful if you want to create game behaviors depending on how the user is holding the device.

You can retrieve device orientation by accessing the Input.deviceOrientation property. Orientation can be one of the following:

| | |
|---|---|
| **Unknown** | The orientation of the device cannot be determined. For example when device is rotate diagonally. |
| **Portrait** | The device is in portrait mode, with the device held upright and the home button at the bottom. |
| **PortraitUpsideDown** | The device is in portrait mode but upside down, with the device held upright and the home button at the top. |
| **LandscapeLeft** | The device is in landscape mode, with the device held upright and the home button on the right side. |
| **LandscapeRight** | The device is in landscape mode, with the device held upright and the home button on the left side. |
| **FaceUp** | The device is held parallel to the ground with the screen facing upwards. |
| **FaceDown** | The device is held parallel to the ground with the screen facing downwards. |

## Accelerometer

As the mobile device moves, a built-in accelerometer reports linear acceleration changes along the three primary axes in three-dimensional space. Acceleration along each axis is reported directly by the hardware as G-force values. A value of 1.0 represents a load of about +1g along a given axis while a value of -1.0 represents -1g. If you hold the device upright (with the home button at the bottom) in front of you, the X axis is positive along the right, the Y axis is positive directly up, and the Z axis is positive pointing toward you.

You can retrieve the accelerometer value by accessing the Input.acceleration property.

The following is an example script which will move an object using the accelerometer:

```
var speed = 10.0;
function Update () {
	var dir : Vector3 = Vector3.zero;

	// we assume that the device is held parallel to the ground
	// and the Home button is in the right hand
```

```
        // remap the device acceleration axis to game coordinates:
        //  1) XY plane of the device is mapped onto XZ plane
        //  2) rotated 90 degrees around Y axis
        dir.x = -Input.acceleration.y;
        dir.z = Input.acceleration.x;

        // clamp acceleration vector to the unit sphere
        if (dir.sqrMagnitude > 1)
                dir.Normalize();

        // Make it move 10 meters per second instead of 10 meters per frame...
        dir *= Time.deltaTime;

        // Move object
        transform.Translate (dir * speed);
}
```

**Low-Pass Filter**
Accelerometer readings can be jerky and noisy. Applying low-pass filtering on the signal allows you to smooth it and get rid of high frequency noise.

The following script shows you how to apply low-pass filtering to accelerometer readings:

```
var AccelerometerUpdateInterval : float = 1.0 / 60.0;
var LowPassKernelWidthInSeconds : float = 1.0;

private var LowPassFilterFactor : float = AccelerometerUpdateInterval / LowPassKernelWidthInSeconds; // tweakable
private var lowPassValue : Vector3 = Vector3.zero;
function Start () {
        lowPassValue = Input.acceleration;
}

function LowPassFilterAccelerometer() : Vector3 {
        lowPassValue = Mathf.Lerp(lowPassValue, Input.acceleration, LowPassFilterFactor);
        return lowPassValue;
}
```

The greater the value of `LowPassKernelWidthInSeconds`, the slower the filtered value will converge towards the current input sample (and vice versa). You should be able to use the `LowPassFilter()` function instead of `avgSamples()`.

**I'd like as much precision as possible when reading the accelerometer. What should I do?**
Reading the Input.acceleration variable does not equal sampling the hardware. Put simply, Unity samples the hardware at a frequency of 60Hz and stores the result into the variable. In reality, things are a little bit more complicated -- accelerometer sampling doesn't occur at consistent time intervals, if under significant CPU loads. As a result, the system might report 2 samples during one frame, then 1 sample during the next frame.

You can access all measurements executed by accelerometer during the frame. The following code will illustrate a simple average of all the accelerometer events that were collected within the last frame:

```
var period : float = 0.0;
var acc : Vector3 = Vector3.zero;
for (var evnt : iPhoneAccelerationEvent  in iPhoneInput.accelerationEvents) {
        acc += evnt.acceleration * evnt.deltaTime;
        period += evnt.deltaTime;
}
if (period > 0)
        acc *= 1.0/period;
```

```
return acc;
```

### Further Reading

The Unity mobile input API is originally based on Apple's API. It may help to learn more about the native API to better understand Unity's Input API. You can find the Apple input API documentation here:

- Programming Guide: Event Handling (Apple iPhone SDK documentation)
- UITouch Class Reference (Apple iOS SDK documentation)

**Note:** The above links reference your locally installed iPhone SDK Reference Documentation and will contain native ObjectiveC code. It is not necessary to understand these documents for using Unity on mobile devices, but may be helpful to some!

## iOS

### Device geographical location

Device geographical location can be obtained via the iPhoneInput.lastLocation property. Before calling this property you should start location service updates using iPhoneSettings.StartLocationServiceUpdates() and check the service status via iPhoneSettings.locationServiceStatus. See the scripting reference for details.

Page last updated: 2012-06-28

---

# iOS-Keyboard

In most cases, Unity will handle keyboard input automatically for GUI elements but it is also easy to show the keyboard on demand from a script.

## iOS

### Using the Keyboard

#### GUI Elements

The keyboard will appear automatically when a user taps on editable GUI elements. Currently, GUI.TextField, GUI.TextArea and GUI.PasswordField will display the keyboard; see the GUI class documentation for further details.

#### Manual Keyboard Handling

Use the **iPhoneKeyboard.Open** function to open the keyboard. Please see the iPhoneKeyboard scripting reference for the parameters that this function takes.

### Keyboard Type Summary

The Keyboard supports the following types:

| | |
|---|---|
| **iPhoneKeyboardType.Default** | Letters. Can be switched to keyboard with numbers and punctuation. |
| **iPhoneKeyboardType.ASCIICapable** | Letters. Can be switched to keyboard with numbers and punctuation. |
| **iPhoneKeyboardType.NumbersAndPunctuation** | Numbers and punctuation. Can be switched to keyboard with letters. |
| **iPhoneKeyboardType.URL** | Letters with slash and .com buttons. Can be switched to keyboard with numbers and punctuation. |
| **iPhoneKeyboardType.NumberPad** | Only numbers from 0 to 9. |
| **iPhoneKeyboardType.PhonePad** | Keyboard used to enter phone numbers. |
| **iPhoneKeyboardType.NamePhonePad** | Letters. Can be switched to phone keyboard. |
| **iPhoneKeyboardType.EmailAddress** | Letters with @ sign. Can be switched to keyboard with numbers and punctuation. |

### Text Preview

By default, an edit box will be created and placed on top of the keyboard after it appears. This works as preview of the text that user is typing, so the text is always visible for the user. However, you can disable text preview by setting

**iPhoneKeyboard.hideInput** to true. Note that this works only for certain keyboard types and input modes. For example, it will not work for phone keypads and multi-line text input. In such cases, the edit box will always appear. **iPhoneKeyboard.hideInput** is a global variable and will affect all keyboards.

### Keyboard Orientation

By default, the keyboard automatically follows the device orientation. To disable or enable rotation to a certain orientation, use the following properties available in iPhoneKeyboard:

| | |
|---|---|
| **autorotateToPortrait** | Enable or disable autorotation to portrait orientation (button at the bottom). |
| **autorotateToPortraitUpsideDown** | Enable or disable autorotation to portrait orientation (button at top). |
| **autorotateToLandscapeLeft** | Enable or disable autorotation to landscape left orientation (button on the right). |
| **autorotateToLandscapeRight** | Enable or disable autorotation to landscape right orientation (button on the left). |

### Visibility and Keyboard Size

There are three keyboard properties in iPhoneKeyboard that determine keyboard visibility status and size on the screen.

| | |
|---|---|
| **visible** | Returns **true** if the keyboard is fully visible on the screen and can be used to enter characters. |
| **area** | Returns the position and dimensions of the keyboard. |
| **active** | Returns **true** if the keyboard is activated. This property is not static property. You must have a keyboard instance to use this property. |

Note that **iPhoneKeyboard.area** will return a rect with position and size set to 0 until the keyboard is fully visible on the screen. You should not query this value immediately after **iPhoneKeyboard.Open**. The sequence of keyboard events is as follows:

- **iPhoneKeyboard.Open** is called. **iPhoneKeyboard.active** returns true. **iPhoneKeyboard.visible** returns false. **iPhoneKeyboard.area** returns (0, 0, 0, 0).
- Keyboard slides out into the screen. All properties remain the same.
- Keyboard stops sliding. **iPhoneKeyboard.active** returns true. **iPhoneKeyboard.visible** returns true. **iPhoneKeyboard.area** returns real position and size of the keyboard.

### Secure Text Input

It is possible to configure the keyboard to hide symbols when typing. This is useful when users are required to enter sensitive information (such as passwords). To manually open keyboard with secure text input enabled, use the following code:

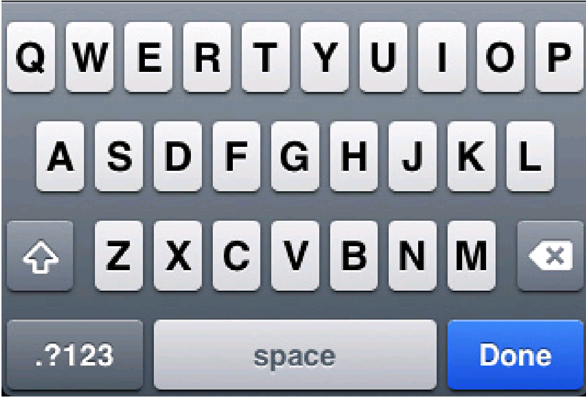
```
iPhoneKeyboard.Open("", iPhoneKeyboardType.Default, false, false, true);
```

*Hiding text while typing*

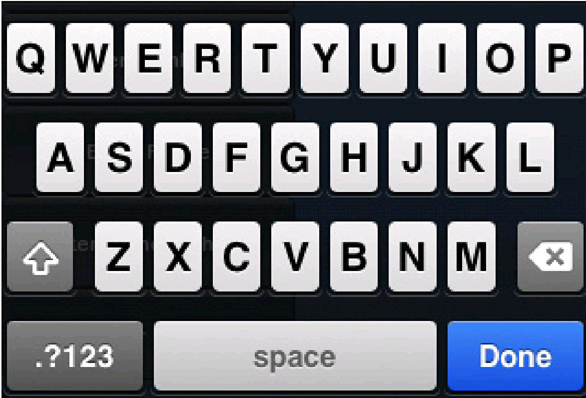
### Alert keyboard

To display the keyboard with a black semi-transparent background instead of the classic opaque, call **iPhoneKeyboard.Open** as follows:

```
iPhoneKeyboard.Open("", iPhoneKeyboardType.Default, false, false, true, true);
```



*Classic keyboard*



*Alert keyboard*

## Android

Unity Android reuses the iOS API to display system keyboard. Even though Unity Android supports most of the functionality of its iPhone counterpart, there are two aspects which are not supported:

- **iPhoneKeyboard.hideInput**
- **iPhoneKeyboard.area**

Please also note that the layout of a **iPhoneKeyboardType** can differ somewhat between devices.

Page last updated: 2011-11-02

---

# iOS-Advanced

## iOS

# Advanced iOS scripting

### Determining Device Generation

Different device generations support different functionality and have widely varying performance. You should query the device's generation and decide which functionality should be disabled to compensate for slower devices.

You can find the device generation from the iPhone.generation property. The reported generation can be one of the following:

- **iPhone**

- **iPhone3G**
- **iPhone3GS**
- **iPhone4**
- **iPodTouch1Gen**
- **iPodTouch2Gen**
- **iPodTouch3Gen**
- **iPodTouch4Gen**
- **iPad1Gen**

You can find more information about different device generations, performance and supported functionality in our iPhone Hardware Guide.

### Device Properties

There are a number of device-specific properties that you can access:-

| | |
|---|---|
| **SystemInfo.deviceUniqueIdentifier** | Unique device identifier. |
| **SystemInfo.deviceName** | User specified name for device. |
| **SystemInfo.deviceModel** | Is it iPhone or iPod Touch? |
| **SystemInfo.operatingSystem** | Operating system name and version. |

### Anti-Piracy Check

Pirates will often hack an application from the AppStore (by removing Apple DRM protection) and then redistribute it for free. Unity iOS comes with an anti-piracy check which allows you to determine if your application was altered **after** it was submitted to the AppStore.

You can check if your application is genuine (not-hacked) with the Application.genuine property. If this property returns **false** then you might notify the user that he is using a hacked application or maybe disable access to some functions of your application.

**Note:** accessing the Application.genuine property is a fairly expensive operation and so you shouldn't do it during frame updates or other time-critical code.

### Vibration Support

You can trigger a vibration by calling Handheld.Vibrate. Note that iPod Touch devices lack vibration hardware and will just ignore this call.

## Android

# Advanced Android scripting

### Determining Device Generation

Different Android devices support different functionality and have widely varying performance. You should target specific devices or device families and decide which functionality should be disabled to compensate for slower devices. There are a number of device specific properties that you can access to which device is being used.

**Note:** Android Marketplace does some additional compatibility filtering, so you should not be concerned if an ARMv7-only app optimised for OGLES2 is offered to some old slow devices.

### Device Properties

| | |
|---|---|
| **SystemInfo.deviceUniqueIdentifier** | Unique device identifier. |
| **SystemInfo.deviceName** | User specified name for device. |
| **SystemInfo.deviceModel** | Is it iPhone or iPod Touch? |
| **SystemInfo.operatingSystem** | Operating system name and version. |

### Anti-Piracy Check

Pirates will often hack an application (by removing Apple DRM protection) and then redistribute it for free. Unity Android comes with an anti-piracy check which allows you to determine if your application was altered **after** it was submitted to the AppStore.

You can check if your application is genuine (not-hacked) with the Application.genuine property. If this property returns **false**

then you might notify user that he is using a hacked application or maybe disable access to some functions of your application.

**Note:** Application.genuineCheckAvailable should be used along with **Application.genuine** to verify that application integrity can actually be confirmed. Accessing the Application.genuine property is a fairly expensive operation and so you shouldn't do it during frame updates or other time-critical code.

### Vibration Support

You can trigger a vibration by calling Handheld.Vibrate. However, devices lacking vibration hardware will just ignore this call.

Page last updated: 2012-07-11

---

# iOS-DotNet

## iOS

Now Unity iOS supports two .NET API compatibility levels: .NET 2.0 and a subset of .NET 2.0 .You can select the appropriate level in the Player Settings.

### .NET API 2.0

Unity supports the **.NET 2.0** API profile. This is close to the full .NET 2.0 API and offers the best compatibility with pre-existing .NET code. However, the application's build size and startup time will be relatively poor.

**Note:** Unity iOS does not support namespaces in scripts. If you have a third party library supplied as source code then the best approach is to compile it to a DLL outside Unity and then drop the DLL file into your project's Assets folder.

### .NET 2.0 Subset

Unity also supports the **.NET 2.0 Subset** API profile. This is close to the Mono "monotouch" profile, so many limitations of the "monotouch" profile also apply to Unity's .NET 2.0 Subset profile. More information on the limitations of the "monotouch" profile can be found here. The advantage of using this profile is reduced build size (and startup time) but this comes at the expense of compatibility with existing .NET code.

## Android

Unity Android supports two .NET API compatibility levels: .NET 2.0 and a subset of .NET 2.0 You can select the appropriate level in the Player Settings.

### .NET API 2.0

Unity supports the **.NET 2.0** API profile; It is close to the full .NET 2.0 API and offers the best compatibility with pre-existing .NET code. However, the application's build size and startup time will be relatively poor.

**Note:** Unity Android does not support namespaces in scripts. If you have a third party library supplied as source code then the best approach is to compile it to a DLL outside Unity and then drop the DLL file into your project's Assets folder.

### .NET 2.0 Subset

Unity also supports the **.NET 2.0 Subset** API profile. This is close to the Mono "monotouch" profile, so many limitations of the "monotouch" profile also apply to Unity's .NET 2.0 Subset profile. More information on the limitations of the "monotouch" profile can be found here. The advantage of using this profile is reduced build size (and startup time) but this comes at the expense of compatibility with existing .NET code.

Page last updated: 2012-07-11

---

# iphone-Hardware

## Hardware models

The following table summarizes iOS hardware available in devices of various generations:

### iPhone Models

**Original iPhone**

- Screen: 320x480 pixels, LCD at 163ppi
- ARM11, 412 Mhz CPU
- PowerVR MBX Lite 3D graphics processor
  - Slow
- 128MB of memory
- 2 megapixel camera

Fixed-function graphics (no fancy shaders), very slow CPU and GPU.

**iPhone 3G**

- Screen: 320x480 pixels, LCD at 163ppi
- ARM11, 412 Mhz CPU
- PowerVR MBX Lite 3D graphics processor
  - Slow
- 128MB of memory
- 2 megapixel camera
- GPS support

**iPhone 3GS**

- Screen: 320x480 pixels, LCD at 163ppi
- ARM Cortex A8, 600 MHz CPU
- PowerVR SGX535 graphics processor
  - Shader perfomance at native resolution, compared to iPad2:
    - Raw shader perfomance, compared to iPad3:
- 256MB of memory
- 3 megapixel camera with video capture capability
- GPS support
- Compass support

Shader-capable hardware, per-pixel-lighting (bumpmaps) can only be on small portions of the screen at once. Requires scripting optimization for complex games. This is the average hardware of the app market as of July 2012

**iPhone 4**

- Screen: 960x640 pixels, LCD at 326 ppi, 800:1 contrast ratio.
- Apple A4
  - 1Ghz ARM Cortex-A8 CPU
  - PowerVR SGX535 GPU
    - Shader perfomance at native resolution, compared to iPad2:
      - Raw shader perfomance, compared to iPad3:
- 512MB of memory
- Cameras
  - Rear 5.0 MP backside illuminated CMOS image sensor with 720p HD video at 30 fps and LED flash
  - Front 0.3 MP (VGA) with geotagging, tap to focus, and 480p SD video at 30 fps
- GPS support
- Compass Support

**iPhone 4S**

- Screen: 960x640 pixels, LCD at 326 ppi, 800:1 contrast ratio.
- Apple A5
  - Dual-Core 1Ghz ARM Cortex-A9 MPCore CPU
  - Dual-Core PowerVR SGX543MP2 GPU
    - Shader perfomance at native resolution, compared to iPad2:
      - Raw shader perfomance, compared to iPad3:
- 512MB of memory
- Cameras
  - Rear 5.0 MP backside illuminated CMOS image sensor with 720p HD

The iPhone 4S, with the new A5 chip, is capable of rendering complex shaders throughout the entire screen. Even image effects may be possible. However, optimizing your shaders is still crucial. But if your game isn't trying to push limits of the device, optimizing scripting and gameplay is probably as much of a waste of time on this generation of devices as it is on PC.

video at 30 fps and LED flash
    - Front 0.3 MP (VGA) with geotagging, tap to focus, and 480p SD video at 30 fps
- GPS support
- Compass Support

### iPod Touch Models

**iPod Touch 1st generation**

- Screen: 320x480 pixels, LCD at 163ppi
- ARM11, 412 Mhz CPU
- PowerVR MBX Lite 3D graphics processor
    - Slow
- 128MB of memory

Fixed-function graphics (no fancy shaders), very slow CPU and GPU.

**iPod Touch 2nd generation**

- Screen: 320x480 pixels, LCD at 163ppi
- ARM11, 533 Mhz CPU
- PowerVR MBX Lite 3D graphics processor
    - Slow
- 128MB of memory
- Speaker and microphone

**iPod Touch 3rd generation**

- Comparable to iPhone 3GS

**iPod Touch 4th generation**

- Comparable to iPhone 4

Shader-capable hardware, per-pixel-lighting (bumpmaps) can only be on small portions of the screen at once. Requires scripting optimization for complex games. This is the average hardware of the app market as of July 2012

### iPad Models

**iPad**

- Screen: 1024x768 pixels, LCD at 132 ppi, LED-backlit.
- Apple A4
    - 1Ghz MHz ARM Cortex-A8 CPU
    - PowerVR SGX535 GPU
        - Shader perfomance at native resolution, compared to iPad2:
            - Raw shader perfomance, compared to iPad3:
- Wifi + Blueooth + (3G Cellular HSDPA, 2G cellular EDGE on the 3G version)
- Accelerometer, ambient light sensor, magnetometer (for digital compass)
- Mechanical keys: Home, sleep, screen rotation lock, volume.

Similar to iPod Touch 4th Generation and iPhone 4.

**iPad 2**

- Screen: 1024x768 pixels, LCD at 132 ppi, LED-backlit.
- Apple A5
    - Dual-Core 1Ghz ARM Cortex-A9 MPCore CPU
    - Dual-Core PowerVR SGX543MP2 GPU
        - Shader perfomance at native resolution, compared to iPad2:
            - Raw shader perfomance, compared to iPad3:
- Same as Previous

The A5 can do full screen bumpmapping, assuming the shader is simple enough. However, it is likely that your game will perform best with bumpmapping only on crucial objects. Full screen image effects still out of reach. Scripting optimization less important.

**iPad 3**

- Screen: 2048 � 1536 pixels, LCD at 264 ppi, LED-backlit.
- Apple A5X
    - Dual-Core 1Ghz ARM Cortex-A9 MPCore CPU
    - Quad-Core PowerVR SGX543MP4 GPU
        - Shader perfomance at native resolution, compared to iPad2:

The iPad 3 has been shown to be capable of render-to-texture effects such as reflective water and fullscreen image effects. However, optimized shaders are still crucial. But if your game isn't trying to push limits of the device,

- Raw shader perfomance, compared to iPad3:

optimizing scripting and gameplay is probably as much of a waste of time on this generation of devices as it is on PC.

## Graphics Processing Unit and Hidden Surface Removal

The iPhone/iPad graphics processing unit (GPU) is a Tile-Based Deferred Renderer. In contrast with most GPUs in desktop computers, the iPhone/iPad GPU focuses on minimizing the work required to render an image as early as possible in the processing of a scene. That way, only the visible pixels will consume processing resources.

The GPU's frame buffer is divided up into tiles and rendering happens tile by tile. First, triangles for the whole frame are gathered and assigned to the tiles. Then, visible fragments of each triangle are chosen. Finally, the selected triangle fragments are passed to the rasterizer (triangle fragments occluded from the camera are rejected at this stage).

In other words, the iPhone/iPad GPU implements a **Hidden Surface Removal** operation at reduced cost. Such an architecture consumes less memory bandwidth, has lower power consumption and utilizes the texture cache better. Tile-Based Deferred Rendering allows the device to reject occluded fragments before actual rasterization, which helps to keep overdraw low.

For more information see also:-

- POWERVR MBX Technology Overview
- Apple Notes on iPhone/iPad GPU and OpenGL ES
- Apple Performance Advices for OpenGL ES in General
- Apple Performance Advices for OpenGL ES Shaders

#### MBX series

Older devices such as the original iPhone, iPhone 3G and iPod Touch 1st and 2nd Generation are equipped with the **MBX** series of GPUs. The MBX series supports only **OpenGL ES1.1**, the fixed function Transform/Lighting pipeline and two textures per fragment.

#### SGX series

Starting with the iPhone 3GS, newer devices are equipped with the **SGX** series of GPUs. The SGX series features support for the **OpenGL ES2.0** rendering API and vertex and pixel shaders. The Fixed-function pipeline is not supported natively on such GPUs, but instead is emulated by generating vertex and pixel shaders with analogous functionality on the fly.

The SGX series fully supports MultiSample anti-aliasing.

#### Texture Compression

The only texture compression format supported by iOS is **PVRTC**. PVRTC provides support for RGB and RGBA (color information plus an alpha channel) texture formats and can compress a single pixel to two or four bits.

The PVRTC format is essential to reduce the memory footprint and to reduce consumption of memory bandwidth (ie, the rate at which data can be read from memory, which is usually very limited on mobile devices).

#### Vertex Processing Unit

The iPhone/iPad has a dedicated unit responsible for vertex processing which runs calculations in parallel with rasterization. In order to achieve better parallelization, the iPhone/iPad processes vertices one frame ahead of the rasterizer.

## Unified Memory Architecture

Both the CPU and GPU on the iPhone/iPad share the same memory. The advantage is that you don't need to worry about running out of video memory for your textures (unless, of course, you run out of main memory too). The disadvantage is that you share the same memory bandwidth for gameplay and graphics. The more memory bandwidth you dedicate to graphics, the less you will have for gameplay and physics.

## Multimedia CoProcessing Unit

The iPhone/iPad main CPU is equipped with a powerful SIMD (Single Instruction, Multiple Data) coprocessor supporting either the **VFP** or the **NEON** architecture. The Unity iOS run-time takes advantage of these units for multiple tasks such as calculating skinned mesh transformations, geometry batching, audio processing and other calculation-intensive operations.

Page last updated: 2012-08-20

---

# iphone-performance

This section covers optimzations which are unique to iOS devices. For more information on optimizing for mobile devices, see the Practical Guide to Optimization for Mobiles.

- iOS Specific Optimizations
- Measuring Performance with the Built-in Profiler
- Optimizing the Size of the Built iOS Player

Page last updated: 2012-07-30

---

# iphone-iOS-Optimization

This page details optimizations which are unique to iOS deployment. For more information on optimizing for mobile devices, see the Practical Guide to Optimization for Mobiles.

## Script Call Optimization

Most of the functions in the **UnityEngine** namespace are implemented in C/C++. Calling a C/C++ function from a Mono script involves a performance overhead. You can use iOS Script Call optimization (menu: **Edit->Project Settings->Player**) to save about 1 to 4 milliseconds per frame. The options for this setting are:-

- **Slow and Safe** - the default Mono internal call handling with exception support.
- **Fast and Exceptions Unsupported** - a faster implementation of Mono internal call handling. However, this doesn't support exceptions and so should be used with caution. An app that doesn't explicitly handle exceptions (and doesn't need to deal with them gracefully) is an ideal candidate for this option.

## Setting the Desired Framerate

Unity iOS allows you to change the frequency with which your application will try to execute its rendering loop, which is set to 30 frames per second by default. You can lower this number to save battery power but of course this saving will come at the expense of frame updates. Conversely, you can increase the framerate to give the rendering priority over other activities such as touch input and accelerometer processing. You will need to experiment with your choice of framerate to determine how it affects gameplay in your case.

If your application involves heavy computation or rendering and can maintain only 15 frames per second, say, then setting the desired frame rate higher than fifteen wouldn't give any extra performance. The application has to be optimized sufficiently to allow for a higher framerate.

To set the desired framerate, open the XCode project generated by Unity and open the `AppController.mm` file. The line

```
#define kFPS 30
```

...determines the the current framerate, so you can just change to set the desired value. For example, if you change the define to:-

```
#define kFPS 60
```

...then the application will attempt to render at 60 FPS instead of 30 FPS.

## The Rendering Loop

When iOS version 3.1 or later is in use, Unity will use the **CADisplayLink** class to schedule the rendering loop. Versions before 3.1 need to use one of several fallback methods to handle the loop. However, the fallback methods can be activated even for iOS 3.1 and later by changing the line

```
#define USE_DISPLAY_LINK_IF_AVAILABLE 1
```

...and changing it to

```
#define USE_DISPLAY_LINK_IF_AVAILABLE 0
```

**Fallback Loop Types**

Apple recommends the system timer for scheduling the rendering operation on iOS versions before 3.1. This approach is good for applications where performance is not critical and favours battery life and correct processing of events over rendering performance. However, better rendering performance is often more important to games, so Unity provides several scheduling methods to tweak the performance of the rendering loop:-

- **System Timer:** this is the standard approach suggested by Apple. It uses the **NSTimer** class to schedule rendering and has the worst rendering performance but guarantees to process all input events.
- **Thread:** a separate thread is used to schedule rendering. This offers better rendering performance than the NSTimer approach, but sometimes could miss touch or accelerometer events. This method of scheduling is also the easiest to set up and is the default method used by Unity for iOS versions before 3.1.
- **Event Pump:** this uses a **CFRunLoop** object to dispatch events. It gives better rendering performance than the NSTimer approach and also allows you to set the amount of time the OS should spend processing touch and accelerometer events. This option must be used with care since touch and accelerometer events will be lost if there is not enough processor time available to handle them.

The different fallback loop types can be selected by changing defines in the AppController.mm file. The significant lines are the following:-

```
#define FALLBACK_LOOP_TYPE NSTIMER_BASED_LOOP
#define FALLBACK_LOOP_TYPE THREAD_BASED_LOOP
#define FALLBACK_LOOP_TYPE EVENT_PUMP_BASED_LOOP
```

The file should have all but one of these lines commented out. The uncommented line selects the rendering loop method that will be used by the application.

If you want to prioritize rendering over input processing with the NSTimer approach you should locate and change the line

```
#define kThrottleFPS 2.0
```

...in AppController.mm. Increasing this number will give higher priority to rendering. The result of changing this value varies among applications, so it is best to try it for yourself and see what happens in your specific case.

If you use the Event Pump rendering loop then you need to tweak the `kMillisecondsPerFrameToProcessEvents` constant precisely to achieve the desired responsiveness. The `kMillisecondsPerFrameToProcessEvents` constant allows you to specify exactly how much time (in milliseconds) you will allow the OS to process events. If you allocate insufficient time for this task then touch or accelerometer events might be lost, and while the application will be fast, it will also be less responsive.

To specify the amount of time (in milliseconds) that the OS will spend processing events, locate and change the line

```
#define kMillisecondsPerFrameToProcessEvents 7.0
```

...in AppController.mm.

## Tuning Accelerometer Processing Frequency

If accelerometer input is processed too frequently then the overall performance of your game may suffer as a result. By default, a Unity iOS application will sample the accelerometer 60 times per second. You may see some performance benefit by reducing the accelerometer sampling frequency and it can even be set to zero for games that don't use accelerometer input. You can change the accelerometer frequency from the **Other Settings** panel in the iOS Player Settings.

Page last updated: 2012-07-30

# iphone-InternalProfiler

## iOS

On iOS, it's disabled by default so to enable it, you need to open the Unity-generated XCode project, select the `iPhone_Profiler.h` file and change the line

```
#define ENABLE_INTERNAL_PROFILER 0
```

to

```
#define ENABLE_INTERNAL_PROFILER 1
```

Select **Run->Console** in the XCode menu to display the output console (GDB) and then run your project. Unity will output statistics to the console window every thirty frames.

## Android

On Android, it is enabled by default. Just make sure Development Build is checked in the player settings when building, and the statistics should show up in logcat when run on the device. To view logcat, you need `adb` or the Android Debug Bridge. Once you have that, simply run the shell command `adb logcat`.

Here's an example of the built-in profiler's output.

```
iPhone/iPad Unity internal profiler stats:
cpu-player>    min:  9.8   max: 24.0   avg: 16.3
cpu-ogles-drv> min:  1.8   max:  8.2   avg:  4.3
cpu-waits-gpu> min:  0.8   max:  1.2   avg:  0.9
cpu-present>   min:  1.2   max:  3.9   avg:  1.6
frametime>     min: 31.9   max: 37.8   avg: 34.1
draw-call #>   min:   4    max:   9    avg:   6     | batched:    10
tris #>        min:  3590  max:  4561  avg:  3871   | batched:  3572
verts #>       min:  1940  max:  2487  avg:  2104   | batched:  1900
player-detail> physx:  1.2 animation:  1.2 culling:  0.5 skinning:  0.0 batching:  0.2 render: 12.0 fixed-update-count: 1 .. 2
mono-scripts>  update:  0.5   fixedUpdate:  0.0 coroutines:  0.0
mono-memory>   used heap: 233472 allocated heap: 548864  max number of collections: 1 collection total duration:  5.7
```

All times are measured in milliseconds per frame. You can see the minimum, maximum and average times over the last thirty frames.

### General CPU Activity

**cpu-player** Displays the time your game spends executing code inside the Unity engine and executing scripts on the CPU.

**cpu-ogles-drv** Displays the time spent executing OpenGL ES driver code on the CPU. Many factors like the number of draw calls, number of internal rendering state changes, the rendering pipeline setup and even the number of processed vertices can have an effect on the driver stats.

**cpu-waits-gpu** Displays the time the CPU is idle while waiting for the GPU to finish rendering. If this number exceeds 2-3 milliseconds then your application is most probably fillrate/GPU processing bound. If this value is too small then the profile skips displaying the value.

**msaa-resolve** The time taken to apply anti-aliasiing.

**cpu-present** The amount of time spent executing the presentRenderbuffer command in OpenGL ES.

**frametime** Represents the overall time of a game frame. Note that iOS hardware is always locked at a 60Hz refresh rate, so you will always get multiples times of ~16.7ms (1000ms/60Hz = ~16.7ms).

### Rendering Statistics

**draw-call #** The number of draw calls per frame. Keep it as low as possible.

**tris #** Total number of triangles sent for rendering.

| | |
|---|---|
| **verts #** | Total number of vertices sent for rendering. You should keep this number below 10000 if you use only static geometry but if you have lots of skinned geometry then you should keep it much lower. |
| **batched** | Number of draw-calls, triangles and vertices which were automatically batched by the engine. Comparing these numbers with draw-call and triangle totals will give you an idea how well is your scene prepared for batching. Share as many materials as possible among your objects to improve batching. |

### Detailed Unity Player Statistics

The **player-detail** section provides a detailed breakdown of what is happening inside the engine:-

| | |
|---|---|
| **physx** | Time spent on physics. |
| **animation** | Time spent animating bones. |
| **culling** | Time spent culling objects outside the camera frustum. |
| **skinning** | Time spent applying animations to skinned meshes. |
| **batching** | Time spent batching geometry. Batching dynamic geometry is considerably more expensive than batching static geometry. |
| **render** | Time spent rendering visible objects. |
| **fixed-update-count** | Minimum and maximum number of FixedUpdates executed during this frame. Too many FixedUpdates will deteriorate performance considerably. There are some simple guidelines to set a good value for the fixed time delta here. |

### Detailed Scripts Statistics

The **mono-scripts** section provides a detailed breakdown of the time spent executing code in the Mono runtime:

| | |
|---|---|
| **update** | Total time spent executing all Update() functions in scripts. |
| **fixedUpdate** | Total time spent executing all FixedUpdate() functions in scripts. |
| **coroutines** | Time spent inside script coroutines. |

### Detailed Statistics on Memory Allocated by Scripts

The **mono-memory** section gives you an idea of how memory is being managed by the Mono garbage collector:

| | |
|---|---|
| **allocated heap** | Total amount of memory available for allocations. A garbage collection will be triggered if there is not enough memory left in the heap for a given allocation. If there is still not enough free memory even after the collection then the allocated heap will grow in size. |
| **used heap** | The portion of the **allocated heap** which is currently used up by objects. Every time you create a new class instance (not a struct) this number will grow until the next garbage collection. |
| **max number of collections** | Number of garbage collection passes during the last 30 frames. |
| **collection total duration** | Total time (in milliseconds) of all garbage collection passes that have happened during the last 30 frames. |

Page last updated: 2012-07-28

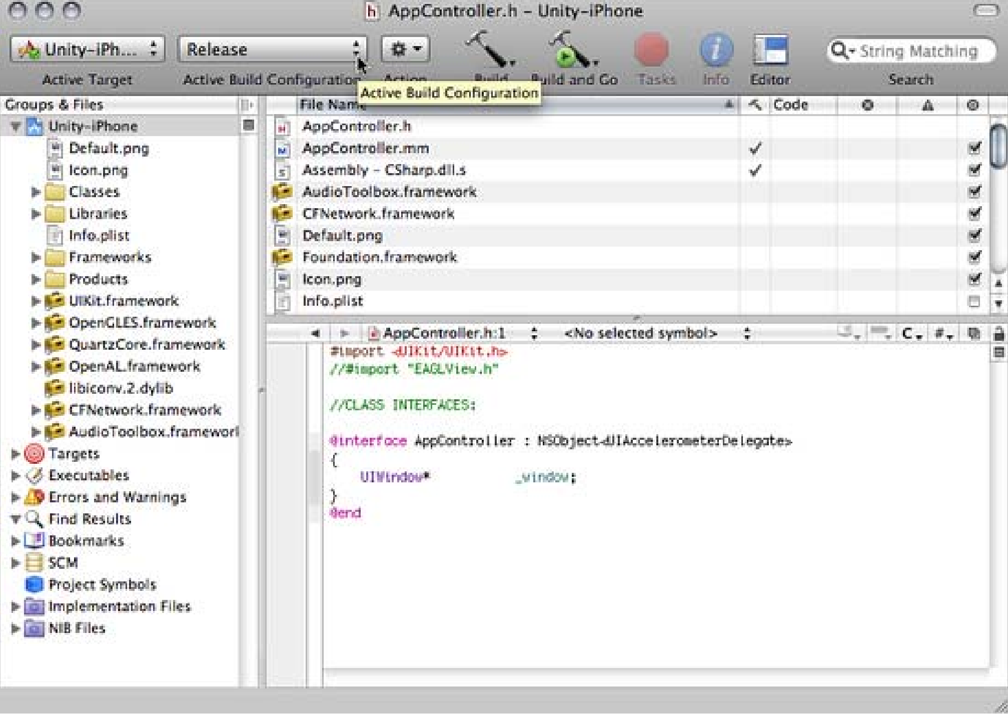
---

# iphone-playerSizeOptimization

The two main ways of reducing the size of the player are by changing the **Active Build Configuration** within Xcode and by changing the **Stripping Level** within Unity.

### Building in Release Mode

You can choose between the **Debug** and **Release** options on the **Active Build Configuration** drop-down menu in Xcode. Building as **Release** instead of **Debug** can reduce the size of the built player by as much as 2-3MB, depending on the game.




*The Active Build Configuration drop-down*

In Release mode, the player will be built without any debug information, so if your game crashes or has other problems there will be no stack trace information available for output. This is fine for deploying a finished game but you will probably want to use Debug mode during development.

## iOS Stripping Level (Advanced License feature)

The size optimizations activated by stripping work in the following way:-

1. **Strip assemblies** level: the scripts' bytecode is analyzed so that classes and methods that are not referenced from the scripts can be removed from the DLLs and thereby excluded from the AOT compilation phase. This optimization reduces the size of the main binary and accompanying DLLs and is safe as long as no reflection is used.
2. **Strip ByteCode** level: any .NET DLLs (stored in the Data folder) are stripped down to metadata only. This is possible because all the code is already precompiled during the AOT phase and linked into the main binary.
3. **Use micro mscorlib** level: a special, smaller version of mscorlib is used. Some components are removed from this library, for example, Security, Reflection.Emit, Remoting, non Gregorian calendars, etc. Also, interdependencies between internal components are minimized. This optimization reduces the main binary and mscorlib.dll size but it is not compatible with some System and System.Xml assembly classes, so use it with care.

These levels are cumulative, so level 3 optimization implicitly includes levels 2 and 1, while level 2 optimization includes level 1.

**Note: Micro mscorlib** is a heavily stripped-down version of the core library. Only those items that are required by the Mono runtime in Unity remain. Best practice for using micro mscorlib is not to use any classes or other features of .NET that are not required by your application. GUIDs are a good example of something you could omit; they can easily be replaced with custom made pseudo GUIDs and doing this would result in better performance and app size.

## Tips

### How to Deal with Stripping when Using Reflection

Stripping depends highly on static code analysis and sometimes this can't be done effectively, especially when dynamic features like reflection are used. In such cases, it is necessary to give some hints as to which classes shouldn't be touched. Unity supports a per-project custom stripping *blacklist*. Using the blacklist is a simple matter of creating a **link.xml** file and placing it into the **Assets** folder. An example of the contents of the **link.xml** file follows. Classes marked for preservation will not be affected by stripping:-

```
<linker>
```

```
    <assembly fullname="System.Web.Services">
        <type fullname="System.Web.Services.Protocols.SoapTypeStubInfo" preserve="all"/>
        <type fullname="System.Web.Services.Configuration.WebServicesConfigurationSectionHandler" preserve="all"/>
    </assembly>

    <assembly fullname="System">
        <type fullname="System.Net.Configuration.WebRequestModuleHandler" preserve="all"/>
        <type fullname="System.Net.HttpRequestCreator" preserve="all"/>
        <type fullname="System.Net.FileWebRequestCreator" preserve="all"/>
    </assembly>
</linker>
```

**Note:** it can sometimes be difficult to determine which classes are getting stripped in error even though the application requires them. You can often get useful information about this by running the stripped application on the simulator and checking the Xcode console for error messages.

**Simple Checklist for Making Your Distribution as Small as Possible**

1. Minimize your assets: enable PVRTC compression for textures and reduce their resolution as far as possible. Also, minimize the number of uncompressed sounds. There are some additional tips for file size reduction here.
2. Set the iOS Stripping Level to **Use micro mscorlib**.
3. Set the script call optimization level to **Fast but no exceptions**.
4. Don't use anything that lives in System.dll or System.Xml.dll in your code. These libraries are **not** compatible with micro mscorlib.
5. Remove unnecessary code dependencies.
6. Set the API Compatibility Level to **.Net 2.0 subset**. Note that .Net 2.0 subset has limited compatibility with other libraries.
7. Set the Target Platform to **armv6 (OpenGL ES1.1)**.
8. Don't use JS Arrays.
9. Avoid generic containers in combination with value types, including structs.

**Can I produce apps of less than 20 megabytes with Unity?**
Yes. An empty project would take about 13 MB in the AppStore if all the size optimizations were turned off. This gives you a budget of about 7MB for compressed assets in your game. If you own an Advanced License (and therefore have access to the stripping option), the empty scene with just the main camera can be reduced to about 6 MB in the AppStore (zipped and DRM attached) and you will have about 14 MB available for compressed assets.

**Why did my app increase in size after being released to the AppStore?**
When they publish your app, Apple first encrypt the binary file and then compresses it via zip. Most often Apple's DRM increases the binary size by about 4 MB or so. As a general rule, you should expect the final size to be approximately equal to the size of the zip-compressed archive of all files (except the executable) plus the size of the uncompressed executable file.

Page last updated: 2011-11-07

---

# iphone-accountsetup

There are some steps you must follow before you can build and run any code (including Unity-built games) on your iOs device. These steps are prerequisite to publishing your own iOS games.

## 1. Apply to Apple to Become a Registered iPhone/iPad Developer
You do this through Apple's website: http://developer.apple.com/iphone/program/

## 2. Upgrade your Operating System and iTunes Installation
Please note that these are Apple's requirements as part of using the iPhone SDK, but the requirements can change from time to time.

## 3. Download the iPhone SDK
Download the latest iOS SDK from the iOS dev center and install it. Do **not** download the beta version of the SDK - you should

use only the latest shipping version. Note that downloading and installing the iPhone SDK will also install XCode.

#### 4. Get Your Device Identifier

Connect your iOS device to the Mac with the USB cable and launch XCode. XCode will detect your phone as a new device and you should register it with the "Use For Development" button. This will usually open the Organizer window but if it doesn't then go to Window->Organizer. You should see your iOS device) in the devices list on the left; select it and note your device's identifier code (which is about 40 characters long).

#### 5. Add Your Device

Log in to the iPhone developer center and enter the program portal (button on the right). Go to the Devices page via the link on left side and then click the Add Device button on the right. Enter a name for your device (alphanumeric characters only) and your device's identifier code (noted in step 5 above). Click the Submit button when done.

#### 6. Create a Certificate

From the iPhone Developer Program Portal, click the Certificates link on the left side and follow the instructions listed under How-To...

#### 7. Download and Install the WWDR Intermediate Certificate

The download link is in the same "Certificates" section (just above the "Important Notice" rubric) as WWDR Intermediate Certificate. Once downloaded, double-click the certificate file to install it.

#### 8. Create a Provisioning File

Provisioning profiles are a bit complex, and need to be set up according to the way you have organized your team. It is difficult to give general instructions for provisioning, so we recommend that you look at the Provisioning How-to section on the Apple Developer website.

Page last updated: 2011-11-08

---

# iphone-unsupported

### Graphics

- DXT texture compression is not supported; use PVRTC formats instead. Please see the Texture2D Component page for more information.
- Rectangular textures can not be compressed to PVRTC formats.
- Movie Textures are not supported; use a full-screen streaming playback instead. Please see the Movie playback page for more information.
- Open GL ES2.0 is not supported on iPhone, iPhone 3G, iPod Touch 1st and iPod Touch 2nd Generation hardware.

### Audio

- Ogg audio compression is not supported. Ogg audio will be automatically converted to MP3 when you switch to iOS platform in the Editor. Please see the AudioClip Component page for more information about audio support in Unity iOS.

### Scripting

- **OnMouseDown**, **OnMouseEnter**, **OnMouseOver**, **OnMouseExit**, **OnMouseDown**, **OnMouseUp**, **OnMouseDrag** events are not supported.
- Dynamic features like Duck Typing are not supported. Use `#pragma strict` for your scripts to force the compiler to report dynamic features as errors.
- Video streaming via **WWW** class is not supported.
- FTP support by **WWW** class is limited.

### Features Restricted to Unity iOS Advanced License

- Static batching is only supported in **Unity iOS Advanced**.
- Video playback is only supported in **Unity iOS Advanced**.
- Splash-screen customization is only supported in **Unity iOS Advanced**.
- AssetBundles are only supported in **Unity iOS Advanced**.
- Code stripping is only supported in **Unity iOS Advanced**.
- .NET sockets are only supported in **Unity iOS Advanced**.

**Note:** it is recommended to minimize your references to external libraries, because 1 MB of .NET CIL code roughly translates to 3-4 MB of ARM code. For example, if your application references System.dll and System.Xml.dll then it means additional 6 MB of ARM code **if stripping is not used**. At some point application will reach limit when linker will have troubles linking the code. If you care a lot about application size you might find C# a more suitable language for your code as is has less dependencies than JavaScript.

Page last updated: 2011-10-29

---

# iphone-Plugins

This page describes Native Code Plugins for the iOS platform.

## Building an Application with a Native Plugin for iOS

1. Define your extern method in the C# file as follows:

```
[DllImport ("__Internal")]
private static extern float FooPluginFunction ();
```

2. Set the editor to the iOS build target
3. Add your native code source files to the generated XCode project's "Classes" folder (this folder is not overwritten when the project is updated, but don't forget to backup your native code).

If you are using C++ (.cpp) or Objective-C (.mm) to implement the plugin you must ensure the functions are declared with C linkage to avoid name mangling issues.

```
extern "C" {
  float FooPluginFunction ();
}
```

## Using Your Plugin from C#

iOS native plugins can be called only when deployed on the actual device, so it is recommended to wrap all native code methods with an additional C# code layer. This code should check Application.platform and call native methods only when the app is running on the device; dummy values can be returned when the app runs in the Editor. See the Bonjour browser sample application for an example.

## Calling C# / JavaScript back from native code

Unity iOS supports limited native-to-managed callback functionality via *UnitySendMessage*:

```
UnitySendMessage("GameObjectName1", "MethodName1", "Message to send");
```

This function has three parameters : the name of the target GameObject, the script method to call on that object and the message string to pass to the called method.

Known limitations:

1. Only script methods that correspond to the following signature can be called from native code: `function MethodName(message:string)`
2. Calls to *UnitySendMessage* are asynchronous and have a delay of one frame.

## Automated plugin integration

Unity iOS supports automated plugin integration in a limited way. All files with extensions **.a**,**.m**,**.mm**,**.c**,**.cpp** located in the Assets/**Plugins/iOS** folder will be merged into the generated Xcode project automatically. However, merging is done by symlinking files from Assets/**Plugins/iOS** to the final destination, which might affect some workflows. The **.h** files are not included in the Xcode project tree, but they appear on the destination file system, thus allowing compilation of .m/.mm/.c/.cpp files.

**Note:** subfolders are currently not supported.

### iOS Tips

1. Managed-to-unmanaged calls are quite processor intensive on iOS. Try to avoid calling multiple native methods per frame.
2. As mentioned above, wrap your native methods with an additional C# layer that calls native code on the device and returns dummy values in the Editor.
3. String values returned from a native method should be UTF-8 encoded and allocated on the heap. Mono marshaling calls are free for strings like this.
4. As mentioned above, the XCode project's "Classes" folder is a good place to store your native code because it is not overwritten when the project is updated.
5. Another good place for storing native code is the Assets folder or one of its subfolders. Just add references from the XCode project to the native code files: right click on the "Classes" subfolder and choose "Add->Existing files...".

### Examples

**Bonjour Browser Sample**
A simple example of the use of a native code plugin can be found here

This sample demonstrates how objective-C code can be invoked from a Unity iOS application. This application implements a very simple Bonjour client. The application consists of a Unity iOS project (Plugins/Bonjour.cs is the C# interface to the native code, while BonjourTest.js is the JS script that implements the application logic) and native code (Assets/Code) that should be added to the built XCode project.

Page last updated: 2011-11-01

---

# iphone-Downloadable-Content

This chapter does **not** aim to cover how to integrate your game with Apple's "StoreKit" API. It is assumed that you already have integration with "StoreKit" via a native code plugin.

Apple's "StoreKit" documentation defines four kinds of **Products** that could be sold via the "In App Purchase" process:

- Content
- Functionality
- Services
- Subscriptions

This chapter covers the first case only and focuses mainly on the downloadable content concept. AssetBundles are ideal candidates for use as downloadable content, and two scenarios will be covered:

- How to export asset bundles for use on iOS
- How download and cache them on iOS

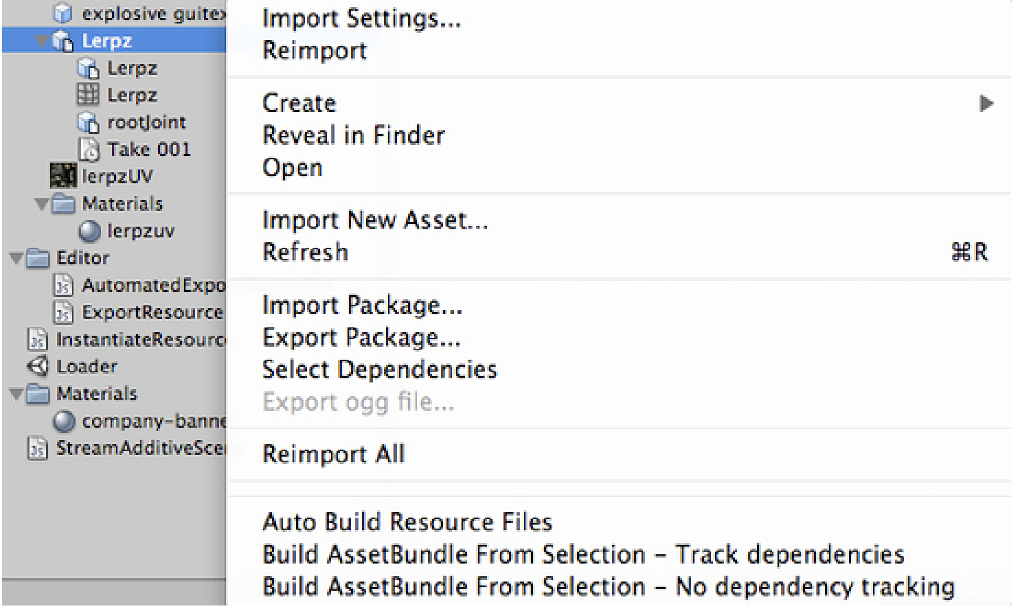
### Exporting your assets for use on iOS
Having separate projects for downloadable content can be a good idea, allowing better separation between content that comes with your main application and content that is downloaded later.

Please note: Any game scripts included in downloadable content must also be present in the main executable.

1. Create an **Editor** folder inside the Project View.
2. Create an **ExportBundle.js** script there and place the following code inside:

```
@MenuItem ("Assets/Build AssetBundle From Selection - Track dependencies")
static function ExportBundle(){

    var str : String = EditorUtility.SaveFilePanel("Save Bundle...", Application.dataPath, Selection.activeObject.name
    if (str.Length != 0){
        BuildPipeline.BuildAssetBundle(Selection.activeObject, Selection.objects, str, BuildAssetBundleOptions.Comp
    }
```

```
}
```

3. Design your objects that need to be downloadable as prefabs
4. Select a prefab that needs to be exported and mouse right click


If the first two steps were done properly, then the *Build AssetBundle From Selection - Track dependencies* context menu item should be visible.
5. Select it if you want to include everything that this asset uses.
6. A save dialog will be shown, enter the desired asset bundle file name. An **.assetbundle** extension will be added automatically. The Unity iOS runtime accepts only asset bundles built with the same version of the Unity editor as the final application. Read BuildPipeline.BuildAssetBundle for details.

### Downloading your assets on iOS

1. Asset bundles can be downloaded and loaded by using the WWW class and instantiating a main asset. Code sample:

```
var download : WWW;

var url = "http://somehost/somepath/someassetbundle.assetbundle";

download = new WWW (url);

yield download;

assetBundle = download.assetBundle;

if (assetBundle != null) {
	// Alternatively you can also load an asset by name (assetBundle.Load("my asset name"))
	var go : Object = assetBundle.mainAsset;

	if (go != null)
		instanced = Instantiate(go);
	else
		Debug.Log("Couldnt load resource");
} else {
	Debug.Log("Couldnt load resource");
}
```

2. You can save required files to a Documents folder next to your game's Data folder.

```
public static string GetiPhoneDocumentsPath () {
	// Your game has read+write access to /var/mobile/Applications/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
	// Application.dataPath returns
```

```
        // /var/mobile/Applications/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX/myappname.app/Data
        // Strip "/Data" from path
        string path = Application.dataPath.Substring (0, Application.dataPath.Length - 5);
        // Strip application name
        path = path.Substring(0, path.LastIndexOf('/'));
        return path + "/Documents";
    }
```

3. Cache a downloaded asset bundle using the .NET file API and for reuse it in the future by loading it via WWW class and **file:///pathtoyourapplication/Documents/savedassetbundle.assetbundle**. Sample code for caching:

```
// Code designed for caching on iPhone, cachedAssetBundle path must be different when running in Editor
// See code snippet above for getting the path to your Documents folder
private var cachedAssetBundle : String = "path to your Documents folder" + "/savedassetbundle.assetbundle";
var cache = new System.IO.FileStream(cachedAssetBundle, System.IO.FileMode.Create);
cache.Write(download.bytes, 0, download.bytes.Length);
cache.Close();
Debug.Log("Cache saved: " + cachedAssetBundle);
```

**Note:** You can test reading files from the Documents folder if you enable file sharing. Setting **UIFileSharingEnabled** to true in your **Info.plist** allows you to access the Documents folder from iTunes.

Page last updated: 2011-11-16

---

# MobileCustomizeSplashScreen

## iOS

Under iOS Basic, a default splash screen will be displayed while your game loads, oriented according to the **Default Screen Orientation** option in the Player Settings.

Users with an iOS Pro license can use any texture in the project as a splash screen. The size of the texture depends on the target device (320x480 pixels for 1-3rd gen devices, 1024x768 for iPad, 640x960 for 4th gen devices) and supplied textures will be scaled to fit if necessary. You can set the splash screen textures using the iOS Player Settings.

## Android

Under Android Basic, a default splash screen will be displayed while your game loads, oriented according to the **Default Screen Orientation** option in the Player Settings.

Android Pro users can use any texture in the project as a splash screen. You can set the texture from the Splash Image section of the Android Player Settings. You should also select the **Splash scaling** method from the following options:-

- **Center (only scale down)** will draw your image at its natural size unless it is too large, in which case it will be scaled down to fit.
- **Scale to fit (letter-boxed)** will draw your image so that the longer dimension fits the screen size exactly. Empty space around the sides in the shorter dimension will be filled in black.
- **Scale to fill (cropped)** will scale your image so that the shorter dimension fits the screen size exactly. The image will be cropped in the longer dimension.

Page last updated: 2011-11-08

---

# iphone-troubleshooting

This section addresses common problems that can arise when using Unity. Each platform is dealt with separately below.

# Desktop

### In MonoDevelop, the Debug button is greyed out!

- This means that MonoDevelop was unable to find the Unity executable. In the MonoDevelop preferences, go to the Unity/Debugger section and then browse to where your Unity executable is located.

### Is there a way to get rid of the welcome page in MonoDevelop?

- Yes. In the MonoDevelop preferences, go to the Visual Style section, and uncheck "Load welcome page on startup".

### Geforce 7300GT on OSX 10.6.4

- Deferred rendering is disabled because materials are not displayed correctly for Geforce 7300GT on OX 10.6.4; This happens because of buggy video drivers.

### On Windows x64, Unity crashes when my script throws a NullReferenceException

- Please apply Windows Hotfix #976038.

### Graphics

#### Slow framerate and/or visual artifacts.

- This may occur if your video card drivers are not up to date. Make sure you have the latest official drivers from your card vendor.

### Shadows

#### I see no shadows at all!

- Shadows are a **Unity Pro** only feature, so without Unity Pro you won't get shadows. Simpler shadow methods, like using a Projector, are still possible, of course.
- Shadows also require certain graphics hardware support. See Shadows page for details.
- Check if shadows are not completely disabled in Quality Settings.
- **Shadows are currently not supported for Android and iOS mobile platforms.**

#### Some of my objects do not cast or receive shadows

An object's Renderer must have **Receive Shadows** enabled for shadows to be rendered onto it. Also, an object must have **Cast Shadows** enabled in order to cast shadows on other objects (both are on by default).

Only opaque objects cast and receive shadows. This means that objects using the built-in Transparent or Particle shaders will not cast shadows. In most cases it is possible to use Transparent Cutout shaders for objects like fences, vegetation, etc. If you use custom written Shaders, they have to be pixel-lit and use the Geometry render queue. Objects using **VertexLit** shaders do not receive shadows but are able to cast them.

Only **Pixel lights** cast shadows. If you want to make sure that a light always casts shadows no matter how many other lights are in the scene, then you can set it to **Force Pixel** render mode (see the Light reference page).

# iOS

## Troubleshooting on iOS devices

There are some situations with iOS where your game can work perfectly in the Unity editor but then doesn't work or maybe doesn't even start on the actual device. The problems are often related to code or content quality. This section describes the most common scenarios.

### The game stops responding after a while. Xcode shows "interrupted" in the status bar.

There are a number of reasons why this may happen. Typical causes include:

1. Scripting errors such as using uninitialized variables, etc.
2. Using 3rd party Thumb compiled native libraries. Such libraries trigger a known problem in the iOS SDK linker and might cause random crashes.
3. Using generic types with value types as parameters (eg, List<int>, List<SomeStruct>, List<SomeEnum>, etc) for serializable script properties.
4. Using reflection when managed code stripping is enabled.

5. Errors in the native plugin interface (the managed code method signature does not match the native code function signature).

Information from the XCode Debugger console can often help detect these problems (Xcode menu: **View > Debug Area > Activate Console**).

### The Xcode console shows "Program received signal: “SIGBUS” or EXC_BAD_ACCESS error.

This message typically appears on iOS devices when your application receives a NullReferenceException. There two ways to figure out where the fault happened:

**Managed stack traces**

Since version 3.4 Unity includes software-based handling of the NullReferenceException. The AOT compiler includes quick checks for null references each time a method or variable is accessed on an object. This feature affects script performance which is why it is enabled only for development builds (for basic license users it is enough to enable the "development build" option in the Build Settings dialog, while iOS pro license users additionally need to enable the "script debugging" option). If everything was done right and the fault actually is occurring in .NET code then you won't see EXC_BAD_ACCESS anymore. Instead, the .NET exception text will be printed in the Xcode console (or else your code will just handle it in a "catch" statement). Typical output might be:

```
Unhandled Exception: System.NullReferenceException: A null value was found where an object instance was required.
  at DayController+$handleTimeOfDay$121+$.MoveNext () [0x0035a] in DayController.js:122
```

This indicates that the fault happened in the handleTimeOfDay method of the DayController class, which works as a coroutine. Also if it is script code then you will generally be told the exact line number (eg, "DayController.js:122"). The offending line might be something like the following:

```
Instantiate(_imgwww.assetBundle.mainAsset);
```

This might happen if, say, the script accesses an asset bundle without first checking that it was downloaded correctly.

**Native stack traces**

Native stack traces are a much more powerful tool for fault investigation but using them requires some expertise. Also, you generally can't continue after these native (hardware memory access) faults happen. To get a native stack trace, type **bt all** into the Xcode Debugger Console. Carefully inspect the printed stack traces - they may contain hints about where the error occurred. You might see something like:

```
...
Thread 1 (thread 11523):
#0 0x006267d0 in m_OptionsMenu_Start ()
#1 0x002e4160 in wrapper_runtime_invoke_object_runtime_invoke_void__this___object_intptr_intptr_intptr ()
#2 0x00a1dd64 in mono_jit_runtime_invoke (method=0x18b63bc, obj=0x5d10cb0, params=0x0, exc=0x2fffdd34) at /Users/m
#3 0x0088481c in MonoBehaviour::InvokeMethodOrCoroutineChecked ()
...
```

First of all you should find the stack trace for **"Thread 1"**, which is the main thread. The very first lines of the stack trace will point to the place where the error occurred. In this example, the trace indicates that the NullReferenceException happened inside the *"OptionsMenu"* script's *"Start"* method. Looking carefully at this method implementation would reveal the cause of the problem. Typically, NullReferenceExceptions happen inside the **Start** method when incorrect assumptions are made about initialization order. In some cases only a partial stack trace is seen on the Debugger Console:

```
Thread 1 (thread 11523):
#0 0x0062564c in start ()
```

This indicates that native symbols were stripped during the Release build of the application. The full stack trace can be obtained with the following procedure:

- Remove application from device.

- Clean all targets.
- Build and run.
- Get stack traces again as described above.

### EXC_BAD_ACCESS starts occurring when an external library is linked to the Unity iOS application.

This usually happens when an external library is compiled with the ARM Thumb instruction set. Currently such libraries are not compatible with Unity. The problem can be solved easily by recompiling the library without Thumb instructions. You can do this for the library's Xcode project with the following steps:

- in Xcode, select *"View"* > *"Navigators"* > *"Show Project Navigator"* from the menu
- select the *"Unity-iPhone"* project, activate *"Build Settings"* tab
- in the search field enter : *"Other C Flags"*
- add *-mno-thumb* flag there and rebuild the library.

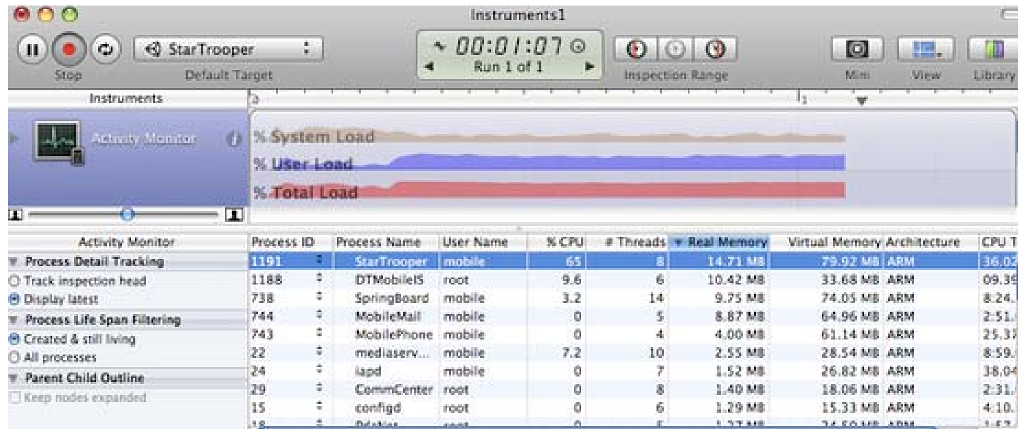
If the library source is not available you should ask the supplier for a non-thumb version of the library.

### The Xcode console shows "WARNING -> applicationDidReceiveMemoryWarning()" and the application crashes immediately afterwards

(Sometimes you might see a message like *Program received signal: �0�.*) This warning message is often not fatal and merely indicates that iOS is low on memory and is asking applications to free up some memory. Typically, background processes like Mail will free some memory and your application can continue to run. However, if your application continues to use memory or ask for more, the OS will eventually start killing applications and yours could be one of them. Apple does not document what memory usage is safe, but empirical observations show that applications using less than 50% MB of all device RAM (like ~200-256 MB for 2nd generation ipad) do not have major memory usage problems. The main metric you should rely on is how much RAM your application uses. Your application memory usage consists of three major components:

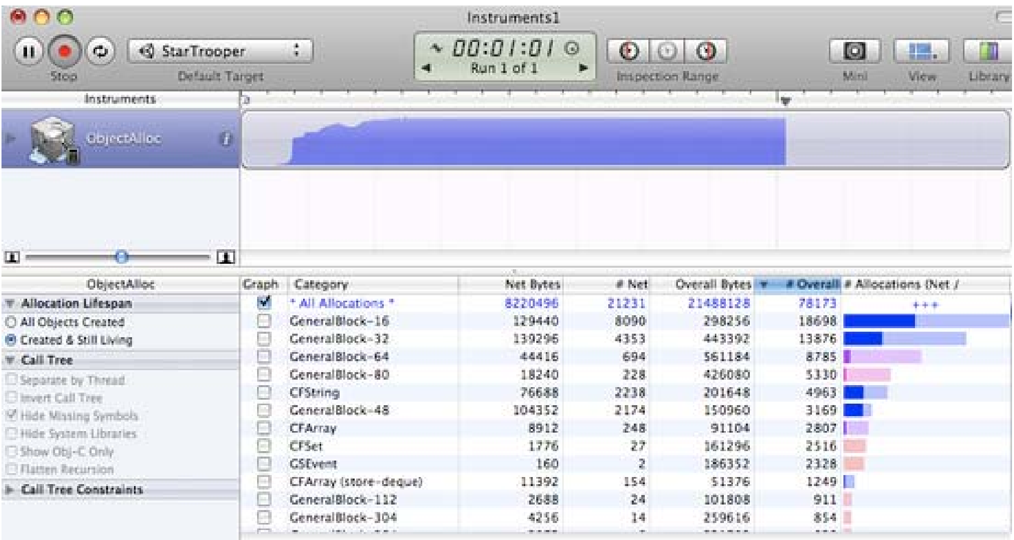
- application code (the OS needs to load and keep your application code in RAM, but some of it might be discarded if really needed)
- native heap (used by the engine to store its state, your assets, etc. in RAM)
- managed heap (used by your Mono runtime to keep C# or JavaScript objects)
- GLES driver memory pools: textures, framebuffers, compiled shaders, etc.

Your application memory usage can be tracked by two Xcode Instruments tools: **Activity Monitor**, **Object Allocations** and **VM Tracker**. You can start from the Xcode Run menu: **Product > Profile** and then select specific tool. **Activity Monitor** tool shows all process statistics including **Real memory** which can be regarded as the total amount of RAM used by your application. **Note:** OS and device HW version combination might noticeably affect memory usage numbers, so you should be careful when comparing numbers obtained on different devices.




**Note:** The internal profiler shows only the heap allocated by .NET scripts. Total memory usage can be determined via Xcode Instruments as shown above. This figure includes parts of the application binary, some standard framework buffers, Unity engine internal state buffers, the .NET runtime heap (number printed by internal profiler), GLES driver heap and some other miscellaneous stuff.

The other tool displays all allocations made by your application and includes both native heap and managed heap statistics (don't forget to check the **Created and still living** box to get the current state of the application). The important statistic is the **Net bytes** value.




To keep memory usage low:

- Reduce the application binary size by using the strongest iOS stripping options (Advanced license feature), and avoid unnecessary dependencies on different .NET libraries. See the player settings and player size optimization manual pages for further details.
- Reduce the size of your content. Use PVRTC compression for textures and use low poly models. See the manual page about reducing file size for more information.
- Don't allocate more memory than necessary in your scripts. Track mono heap size and usage with the internal profiler
- **Note:** with Unity 3.0, the scene loading implementation has changed significantly and now all scene assets are preloaded. This results in fewer hiccups when instantiating game objects. If you need more fine-grained control of asset loading and unloading during gameplay, you should use Resources.Load and Object.Destroy.

Querying the OS about the amount of free memory may seem like a good idea to evaluate how well your application is performing. However, the free memory statistic is likely to be unreliable since the OS uses a lot of dynamic buffers and caches. The only reliable approach is to keep track of memory consumption for your application and use that as the main metric. Pay attention to how the graphs from the tools described above change over time, especially after loading new levels.

### The game runs correctly when launched from Xcode but crashes while loading the first level when launched manually on the device.

There could be several reasons for this. You need to inspect the device logs to get more details. Connect the device to your Mac, launch Xcode and select **Window > Organizer** from the menu. Select your device in the Organizer's left toolbar, then click on the "Console" tab and review the latest messages carefully. Additionally, you may need to investigate crash reports. You can find out how to obtain crash reports here: http://developer.apple.com/iphone/library/technotes/tn2008/tn2151.html.

### The Xcode Organizer console contains the message "killed by SpringBoard".

There is a poorly-documented time limit for an iOS application to render its first frames and process input. If your application exceeds this limit, it will be killed by SpringBoard. This may happen in an application with a first scene which is too large, for example. To avoid this problem, it is advisable to create a small initial scene which just displays a splash screen, waits a frame or two with **yield** and then starts loading the real scene. This can be done with code as simple as the following:

```
function Start () {
    yield;
    Application.LoadLevel("Test");
}
```

### Type.GetProperty() / Type.GetValue() cause crashes on the device

Currently **Type.GetProperty()** and **Type.GetValue()** are supported only for the **.NET 2.0 Subset** profile. You can select the .NET API compatibility level in the Player Settings.

**Note: Type.GetProperty()** and **Type.GetValue()** might be incompatible with managed code stripping and might need to be excluded (you can supply a custom non-strippable type list during the stripping process to accomplish this). For further details,

see the iOS player size optimization guide.

### The game crashes with the error message "ExecutionEngineException: Attempting to JIT compile method 'SometType`1<SomeValueType>:.ctor ()' while running with --aot-only."

The Mono .NET implementation for iOS is based on AOT (ahead of time compilation to native code) technology, which has its limitations. It compiles only those generic type methods (where a value type is used as a generic parameter) which are explicitly used by other code. When such methods are used only via reflection or from native code (ie, the serialization system) then they get skipped during AOT compilation. The AOT compiler can be hinted to include code by adding a dummy method somewhere in the script code. This can refer to the missing methods and so get them compiled ahead of time.

```
void _unusedMethod()
{
   var tmp = new SomeType<SomeValueType>();
}
```

**Note:** value types are basic types, enums and structs.

### Various crashes occur on the device when a combination of System.Security.Cryptography and managed code stripping is used

.NET Cryptography services rely heavily on reflection and so are not compatible with managed code stripping since this involves static code analysis. Sometimes the easiest solution to the crashes is to exclude the whole **System.Security.Crypography** namespace from the stripping process.

The stripping process can be customized by adding a custom **link.xml** file to the **Assets** folder of your Unity project. This specifies which types and namespaces should be excluded from stripping. Further details can be found in the iOS player size optimization guide.

**link.xml**

```
<linker>
       <assembly fullname="mscorlib">
              <namespace fullname="System.Security.Cryptography" preserve="all"/>
       </assembly>
</linker>
```

**Application crashes when using System.Security.Cryptography.MD5 with managed code stripping**

You might consider advice listed above or can work around this problem by adding extra reference to specific class to your script code:

```
object obj = new MD5CryptoServiceProvider();
```

### "Ran out of trampolines of type 1/2" runtime error

This error usually happens if you use lots of recursive generics. You can hint to the AOT compiler to allocate more trampolines of type 1 or type 2. Additional AOT compiler command line options can be specified in the "Other Settings" section of the Player Settings. For type 1 trampolines, specify **nrgctx-trampolines=ABCD**, where ABCD is the number of new trampolines required (i.e. 4096). For type 2 trampolines specify **nimt-trampolines=ABCD**.

### After upgrading Xcode Unity iOS runtime fails with message "You are using Unity iPhone Basic. You are not allowed to remove the Unity splash screen from your game"

With some latest Xcode releases there were changes introduced in PNG compression and optimization tool. These changes might cause false positives in Unity iOS runtime checks for splash screen modifications. If you encounter such problems try upgrading Unity to the latest publicly available version. If it does not help you might consider following workaround:

- Replace your Xcode project from scratch when building from Unity (instead of appending it)
- Delete already installed project from device
- Clean project in Xcode (*Product->Clean*)
- Clear Xcode's Derived Data folders (*Xcode->Preferences->Locations*)

If this still does not help try disabling PNG re-compression in Xcode:

- Open your Xcode project
- Select "Unity-iPhone" project there
- Select "Build Settings" tab there
- Look for "Compress PNG files" option and set it to NO

### App Store submission fails with "iPhone/iPod Touch: application executable is missing a required architecture. At least one of the following architecture(s) must be present: armv6" message

You might get such message when updating already existing application, which previously was submitted with armv6 support. Unity 4.x and Xcode 4.5 does not support armv6 platform anymore. To solve submission problem just set **Target OS Version** in Unity **Player Settings** to **4.3** or higher.

### WWW downloads are working fine in Unity Editor and on Android, but not on iOS

Most common mistake is to assume that WWW downloads are always happening on separate thread. On some platforms this might be true, but you should not take it for granted. Best way to track WWW status is either to use *yield* statement or check status in *Update* method. You should **not** use busy *while* loops for that.

### "PlayerLoop called recursively!" error occurs when using Cocoa via a native function called from a script

Some operations with the UI will result in iOS redrawing the window immediately (the most common example is adding a UIView with a UIViewController to the main UIWindow). If you call a native function from a script, it will happen inside Unity's PlayerLoop, resulting in PlayerLoop being called recursively. In such cases, you should consider using the performSelectorOnMainThread method with waitUntilDone set to false. It will inform iOS to schedule the operation to run between Unity's PlayerLoop calls.

### Profiler or Debugger unable to see game running on iOS device

- Check that you have built a Development build, and ticked the "Enable Script Debugging" and "Autoconnect profiler" boxes (as appropriate).
- The application running on the device will make a multicast broadcast to 225.0.0.222 on UDP port 54997. Check that your network settings allow this traffic. Then, the profiler will make a connection to the remote device on a port in the range 55000 - 55511 to fetch profiler data from the device. These ports will need to be open for UDP access.

### Missing DLLs

If your application runs ok in editor but you get errors in your iOS project this may be caused by missing DLLs (e.g. I18N.dll, I19N.West.dll). In this case, try copying those dlls from within the Unity.app to your project's Assets/Plugins folder. The location of the DLLs within the unity app is:

```
Unity.app/Contents/Frameworks/Mono/lib/mono/unity
```

You should then also check the stripping level of your project to ensure the classes in the DLLs aren't being removed when the build is optimised. Refer to the iOS Optimisation Page for more information on iOS Stripping Levels.

### Xcode Debugger console reports: ExecutionEngineException: Attempting to JIT compile method '(wrapper native-to-managed) Test:TestFunc (int)' while running with --aot-only

Typically such message is received when managed function delegate is passed to the native function, but required wrapper code wasn't generated when building application. You can help AOT compiler by hinting which methods will be passed as delegates to the native code. This can be done by adding "MonoPInvokeCallbackAttribute" custom attribute. Currently only static methods can be passed as delegates to the native code.

Sample code:

```
using UnityEngine;
using System.Collections;
using System;
using System.Runtime.InteropServices;
using AOT;

public class NewBehaviourScript : MonoBehaviour {

    [DllImport ("__Internal")]
    private static extern void DoSomething (NoParamDelegate del1, StringParamDelegate del2);
```

```
	delegate void NoParamDelegate ();
	delegate void StringParamDelegate (string str);

	[MonoPInvokeCallback (typeof (NoParamDelegate))]
	public static void NoParamCallback()
	{
		Debug.Log ("Hello from NoParamCallback");
	}

	[MonoPInvokeCallback (typeof (StringParamDelegate))]
	public static void StringParamCallback(string str)
	{
		Debug.Log (string.Format ("Hello from StringParamCallback {0}", str));
	}

	// Use this for initialization
	void Start () {
		DoSomething(NoParamCallback, StringParamCallback);
	}
}
```

## Android

# Troubleshooting Android development

### Unity fails to install your application to your device

1. Verify that your computer can actually see and communicate with the device. See the Publishing Builds page for further details.
2. Check the error message in the Unity console. This will often help diagnose the problem.

If you get an error saying "Unable to install APK, protocol failure" during a build then this indicates that the device is connected to a low-power USB port (perhaps a port on a keyboard or other peripheral). If this happens, try connecting the device to a USB port on the computer itself.

### Your application crashes immediately after launch.

1. Ensure that you are not trying to use NativeActivity with devices that do not support it.
2. Try removing any native plugins you have.
3. Try disabling stripping.
4. Use **adb logcat** to get the crash report from your device.

### Building DEX Failed

This an error which will produce a message like the following:-

```
Building DEX Failed!
G:\Unity\JavaPluginSample\Temp/StagingArea> java -Xmx1024M
-Djava.ext.dirs="G:/AndroidSDK/android-sdk_r09-windows\platform-tools/lib/"
-jar "G:/AndroidSDK/android-sdk_r09-windows\platform-tools/lib/dx.jar"
--dex --verbose --output=bin/classes.dex bin/classes.jar plugins
Error occurred during initialization of VM
Could not reserve enough space for object heap
Could not create the Java virtual machine.
```

This is usually caused by having the wrong version of Java installed on your machine. Updating your Java installation to the latest version will generally solve this issue.

### The game crashes after a couple of seconds when playing video

Make sure `Settings->Developer Options->Don't keep activities` isn't enabled on the phone. The video player is its own activity and therefore the regular game activity will be destroyed if the video player is activated.

### My game quits when I press the sleep button

Change the `<activity>` tag in the `AndroidManifest.xml` to contain `<android:configChanges>` tag as described here.

An example activity tag might look something like this:-

```
<activity android:name=".AdMobTestActivity"
            android:label="@string/app_name"
            android:configChanges="fontScale|keyboard|keyboardHidden|locale|mnc|mcc|navigation|orientation|screenLayou
        <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
      </activity>
```

Page last updated: 2012-11-26

---

## iphone-bugreporting

Before submitting a bug report, please check the iOS Troubleshooting page, where you will find solutions to common crashes and other problems.

If your application crashes in the Xcode debugger then you can add valuable information to your bug report as follows:-

1. Click Continue (**Run->Continue**) twice
2. Open the debugger console (**Run->Console**) and enter (in the console): **thread apply all bt**
3. Copy **all** console output and send it together with your bugreport.

If your application crashes on the iOS device then you should retrieve the crash report as described here on Apple's website. Please attach the crash report, your built application and console log to your bug report before submitting.

Page last updated: 2011-11-08

---

## android-GettingStarted

Building games for a device running Android OS requires an approach similar to that for iOS development. However, the hardware is not completely standardized across all devices, and this raises issues that don't occur in iOS development. There are some feature differences in the Android version of Unity just as there are with the iOS version.

### Setting up your Android Developer environment

You will need to have your Android developer environment set up before you can test your Unity games on the device. This involves downloading and installing the Android SDK with the different Android plaforms and adding your physical device to your system (this is done a bit differently depending on whether you are developing on Windows or Mac). This setup process is explained on the Android developer website, and there may be additional information provided by the manufacturer of your device. Since this is a complex process, we've provided a basic outline of the tasks that must be completed before you can run code on your Android device or in the Android emulator. However, the best thing to do is follow the instructions step-by-step from the Android developer portal.

### Access Android Functionality

Unity Android provides scripting APIs to access various input data and settings. You can find out more about the available classes on the Android scripting page.

### Exposing Native C, C++ or Java Code to Scripts

Unity Android allows you to call custom functions written in C/C++ directly from C# scripts (Java functions can be called indirectly). To find out how to make functions from native code accessible from Unity, visit the plugins page.

### Occlusion Culling

Unity includes support for occlusion culling which is a particularly valuable optimization on mobile platforms. More information can be found on the occlusion culling page.

### Splash Screen Customization

The splash screen displayed while the game launches can be customized - see this page for further details.

### Troubleshooting and Bug Reports

There are many reasons why your application may crash or fail to work as you expected. Our Android troubleshooting guide will help you get to the bottom of bugs as quickly as possible. If, after consulting the guide, you suspect the problem is internal to Unity then you should file a bug report - see this page for details on how to do this.

### How Unity Android Differs from Desktop Unity

#### Strongly Typed JavaScript

For performance reasons, dynamic typing in JavaScript is always turned off in Unity Android, as if #pragma strict were applied automatically to all scripts. This is important to know if you start with a project originally developed for the desktop platforms since you may find you get unexpected compile errors when switching to Android; dynamic typing is the first thing to investigate. These errors are usually easy to fix if you make sure all variables are explicitly typed or use type inference on initialization.

#### ETC as Recommended Texture Compression

Although Unity Android does support DXT/PVRTC/ATC textures, Unity will decompress the textures into RGB(A) format at runtime if those compression methods are not supported by the particular device in use. This could have an impact on the GPU rendering speed and it is recommended to use the ETC format instead. ETC is the de facto standard compression format on Android, and should be supported on all post 2.0 devices. However, ETC does not support an alpha channel and RGBA 16-bit will sometimes be the best trade-off between size, quality and rendering speed where alpha is required.

It is also possible to create separate android distribution archives (.apk) for each of the DXT/PVRTC/ATC formats, and let the Android Market's filtering system select the correct archives for different devices (see Publishing Builds for Android).

#### Movie Playback

Movie textures are not supported on Android, but a full-screen streaming playback is provided via scripting functions. To learn about supported file formats and scripting API, consult the movie page or the Android supported media formats page.

#### Further Reading

- Android SDK Setup
- Android Remote
- Trouble Shooting
- Reporting crash bugs under Android
- Features currently not supported by Unity Android
- android-OBBsupport
- Player Settings
- Android Scripting
  - Input
  - Mobile Keyboard
  - Advanced Unity Mobile Scripting
  - Using .NET API 2.0 compatibility level
- Building Plugins for Android
- Customizing the Splash screen of Your Mobile Application

Page last updated: 2011-11-22

---

## android-sdksetup

There are some steps you must follow before you can build and run any code on your Android device. This is true regardless of whether you use Unity or write Android applications from scratch.

### 1. Download the Android SDK

Go to the Android Developer SDK webpage. Download and unpack the latest Android SDK.

### 2. Installing the Android SDK

Follow the instructions under Installing the SDK (although you can freely skip the optional parts relating to Eclipse). In step 4 of *Installing the SDK* be sure to add at least one **Android platform** with API level equal to or higher than 9 (Platform 2.3 or greater), the **Platform Tools**, and the **USB drivers** if you're using Windows.

### 3. Get the device recognized by your system

This can be tricky, especially under Windows based systems where drivers tend to be a problem. Also, your device may come with additional information or specific drivers from the manufacturer.

- For **Windows**: If the Android device is automatically recognized by the system you still might need to update the drivers with the ones that came with the Android SDK. This is done through the Windows Device Manager.
  If the device is not recognized automatically use the drivers from the Android SDK, or any specific drivers provided by the manufacturer.
  Additional info can be found here: USB Drivers for Windows
- For **Mac**: If you're developing on Mac OSX then no additional drivers are usually required.

**Note:** Don't forget to turn on "USB Debugging" on your device. You can do this from the home screen: press MENU, select Applications > Development, then enable USB debugging.

If you are unsure whether your device is properly installed on your system, please read the trouble-shooting page for details.

### 4. Add the Android SDK path to Unity

The first time you build a project for Android (or if Unity later fails to locate the SDK) you will be asked to locate the folder where you installed the Android SDK (you should select the root folder of the SDK installation). The location of the Android SDK can also be changed in the editor by selecting Unity > Preferences from the menu and then clicking on External Tools in the preferences window.

Page last updated: 2012-03-23

---

## android-remote

Android Remote is a Android application that makes your device act as a remote control for the project in Unity. This is useful for rapid development when you don't want to compile and deploy your project to device for each change.

### How to use Android remote

To use Android Remote, you should firstly make sure that you have the latest Android SDK installed (this is necessary to set up port-forwarding on the device). Then, connect the device to your computer with a USB cable and launch the Android Remote app. When you press Play in the Unity editor, the device will act as a remote control and will pass accelerometer and touch input events to the running game.

Page last updated: 2011-11-23

---

## android-troubleshooting

This section addresses common problems that can arise when using Unity. Each platform is dealt with separately below.

## Desktop

### In MonoDevelop, the Debug button is greyed out!

- This means that MonoDevelop was unable to find the Unity executable. In the MonoDevelop preferences, go to the Unity/Debugger section and then browse to where your Unity executable is located.

### Is there a way to get rid of the welcome page in MonoDevelop?

- Yes. In the MonoDevelop preferences, go to the Visual Style section, and uncheck "Load welcome page on startup".

### Geforce 7300GT on OSX 10.6.4

- Deferred rendering is disabled because materials are not displayed correctly for Geforce 7300GT on OX 10.6.4; This happens because of buggy video drivers.

### On Windows x64, Unity crashes when my script throws a NullReferenceException

- Please apply Windows Hotfix #976038.

### Graphics

#### Slow framerate and/or visual artifacts.

- This may occur if your video card drivers are not up to date. Make sure you have the latest official drivers from your card vendor.

### Shadows

#### I see no shadows at all!

- Shadows are a **Unity Pro** only feature, so without Unity Pro you won't get shadows. Simpler shadow methods, like using a Projector, are still possible, of course.
- Shadows also require certain graphics hardware support. See Shadows page for details.
- Check if shadows are not completely disabled in Quality Settings.
- **Shadows are currently not supported for Android and iOS mobile platforms.**

#### Some of my objects do not cast or receive shadows

An object's Renderer must have **Receive Shadows** enabled for shadows to be rendered onto it. Also, an object must have **Cast Shadows** enabled in order to cast shadows on other objects (both are on by default).

Only opaque objects cast and receive shadows. This means that objects using the built-in Transparent or Particle shaders will not cast shadows. In most cases it is possible to use Transparent Cutout shaders for objects like fences, vegetation, etc. If you use custom written Shaders, they have to be pixel-lit and use the Geometry render queue. Objects using **VertexLit** shaders do not receive shadows but are able to cast them.

Only **Pixel lights** cast shadows. If you want to make sure that a light always casts shadows no matter how many other lights are in the scene, then you can set it to **Force Pixel** render mode (see the Light reference page).

## iOS

# Troubleshooting on iOS devices

There are some situations with iOS where your game can work perfectly in the Unity editor but then doesn't work or maybe doesn't even start on the actual device. The problems are often related to code or content quality. This section describes the most common scenarios.

### The game stops responding after a while. Xcode shows "interrupted" in the status bar.

There are a number of reasons why this may happen. Typical causes include:

1. Scripting errors such as using uninitialized variables, etc.
2. Using 3rd party Thumb compiled native libraries. Such libraries trigger a known problem in the iOS SDK linker and might cause random crashes.
3. Using generic types with value types as parameters (eg, List<int>, List<SomeStruct>, List<SomeEnum>, etc) for serializable script properties.
4. Using reflection when managed code stripping is enabled.
5. Errors in the native plugin interface (the managed code method signature does not match the native code function signature).

Information from the XCode Debugger console can often help detect these problems (Xcode menu: **View > Debug Area > Activate Console**).

### The Xcode console shows "Program received signal: “SIGBUS” or EXC_BAD_ACCESS error.

This message typically appears on iOS devices when your application receives a NullReferenceException. There two ways to figure out where the fault happened:

**Managed stack traces**

Since version 3.4 Unity includes software-based handling of the NullReferenceException. The AOT compiler includes quick checks for null references each time a method or variable is accessed on an object. This feature affects script performance which is why it is enabled only for development builds (for basic license users it is enough to enable the "development build" option in the Build Settings dialog, while iOS pro license users additionally need to enable the "script debugging" option). If everything was done right and the fault actually is occurring in .NET code then you won't see EXC_BAD_ACCESS anymore. Instead, the .NET exception text will be printed in the Xcode console (or else your code will just handle it in a "catch" statement). Typical output might be:

```
Unhandled Exception: System.NullReferenceException: A null value was found where an object instance was required.
  at DayController+$handleTimeOfDay$121+$.MoveNext () [0x0035a] in DayController.js:122
```

This indicates that the fault happened in the handleTimeOfDay method of the DayController class, which works as a coroutine. Also if it is script code then you will generally be told the exact line number (eg, "DayController.js:122"). The offending line might be something like the following:

```
 Instantiate(_imgwww.assetBundle.mainAsset);
```

This might happen if, say, the script accesses an asset bundle without first checking that it was downloaded correctly.

**Native stack traces**

Native stack traces are a much more powerful tool for fault investigation but using them requires some expertise. Also, you generally can't continue after these native (hardware memory access) faults happen. To get a native stack trace, type **bt all** into the Xcode Debugger Console. Carefully inspect the printed stack traces - they may contain hints about where the error occurred. You might see something like:

```
...
Thread 1 (thread 11523):
#0 0x006267d0 in m_OptionsMenu_Start ()
#1 0x002e4160 in wrapper_runtime_invoke_object_runtime_invoke_void__this___object_intptr_intptr_intptr ()
#2 0x00a1dd64 in mono_jit_runtime_invoke (method=0x18b63bc, obj=0x5d10cb0, params=0x0, exc=0x2fffdd34) at /Users/m
#3 0x0088481c in MonoBehaviour::InvokeMethodOrCoroutineChecked ()
...
```

First of all you should find the stack trace for **"Thread 1"**, which is the main thread. The very first lines of the stack trace will point to the place where the error occurred. In this example, the trace indicates that the NullReferenceException happened inside the *"OptionsMenu"* script's *"Start"* method. Looking carefully at this method implementation would reveal the cause of the problem. Typically, NullReferenceExceptions happen inside the **Start** method when incorrect assumptions are made about initialization order. In some cases only a partial stack trace is seen on the Debugger Console:

```
Thread 1 (thread 11523):
#0 0x0062564c in start ()
```

This indicates that native symbols were stripped during the Release build of the application. The full stack trace can be obtained with the following procedure:

- Remove application from device.
- Clean all targets.
- Build and run.
- Get stack traces again as described above.

### EXC_BAD_ACCESS starts occurring when an external library is linked to the Unity iOS application.

This usually happens when an external library is compiled with the ARM Thumb instruction set. Currently such libraries are not compatible with Unity. The problem can be solved easily by recompiling the library without Thumb instructions. You can do this for the library's Xcode project with the following steps:

- in Xcode, select *"View"* > *"Navigators"* > *"Show Project Navigator"* from the menu
- select the *"Unity-iPhone"* project, activate *"Build Settings"* tab
- in the search field enter : *"Other C Flags"*
- add *-mno-thumb* flag there and rebuild the library.

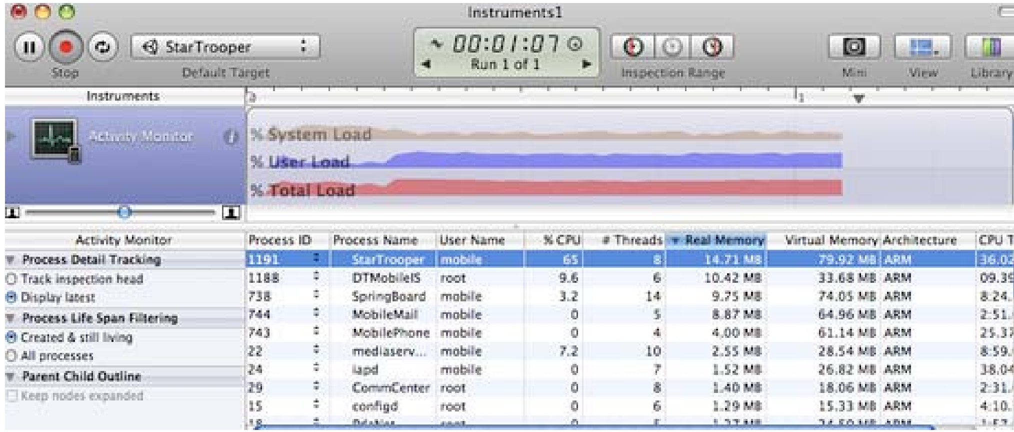
If the library source is not available you should ask the supplier for a non-thumb version of the library.

### The Xcode console shows "WARNING -> applicationDidReceiveMemoryWarning()" and the application crashes immediately afterwards

(Sometimes you might see a message like *Program received signal: �0�*.) This warning message is often not fatal and merely indicates that iOS is low on memory and is asking applications to free up some memory. Typically, background processes like Mail will free some memory and your application can continue to run. However, if your application continues to use memory or ask for more, the OS will eventually start killing applications and yours could be one of them. Apple does not document what memory usage is safe, but empirical observations show that applications using less than 50% MB of all device RAM (like ~200-256 MB for 2nd generation ipad) do not have major memory usage problems. The main metric you should rely on is how much RAM your application uses. Your application memory usage consists of three major components:

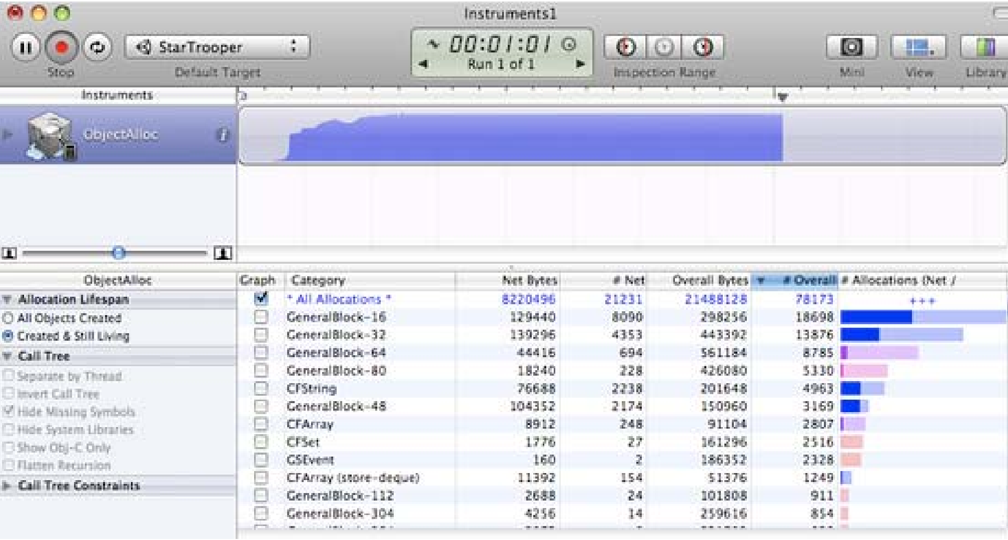
- application code (the OS needs to load and keep your application code in RAM, but some of it might be discarded if really needed)
- native heap (used by the engine to store its state, your assets, etc. in RAM)
- managed heap (used by your Mono runtime to keep C# or JavaScript objects)
- GLES driver memory pools: textures, framebuffers, compiled shaders, etc.

Your application memory usage can be tracked by two Xcode Instruments tools: **Activity Monitor**, **Object Allocations** and **VM Tracker**. You can start from the Xcode Run menu: **Product > Profile** and then select specific tool. **Activity Monitor** tool shows all process statistics including **Real memory** which can be regarded as the total amount of RAM used by your application. **Note:** OS and device HW version combination might noticeably affect memory usage numbers, so you should be careful when comparing numbers obtained on different devices.




**Note:** The internal profiler shows only the heap allocated by .NET scripts. Total memory usage can be determined via Xcode Instruments as shown above. This figure includes parts of the application binary, some standard framework buffers, Unity engine internal state buffers, the .NET runtime heap (number printed by internal profiler), GLES driver heap and some other miscellaneous stuff.

The other tool displays all allocations made by your application and includes both native heap and managed heap statistics (don't forget to check the **Created and still living** box to get the current state of the application). The important statistic is the **Net bytes** value.




To keep memory usage low:

- Reduce the application binary size by using the strongest iOS stripping options (Advanced license feature), and avoid unnecessary dependencies on different .NET libraries. See the player settings and player size optimization manual pages for further details.
- Reduce the size of your content. Use PVRTC compression for textures and use low poly models. See the manual page about reducing file size for more information.
- Don't allocate more memory than necessary in your scripts. Track mono heap size and usage with the internal profiler
- **Note:** with Unity 3.0, the scene loading implementation has changed significantly and now all scene assets are preloaded. This results in fewer hiccups when instantiating game objects. If you need more fine-grained control of asset loading and unloading during gameplay, you should use Resources.Load and Object.Destroy.

Querying the OS about the amount of free memory may seem like a good idea to evaluate how well your application is performing. However, the free memory statistic is likely to be unreliable since the OS uses a lot of dynamic buffers and caches. The only reliable approach is to keep track of memory consumption for your application and use that as the main metric. Pay attention to how the graphs from the tools described above change over time, especially after loading new levels.

### The game runs correctly when launched from Xcode but crashes while loading the first level when launched manually on the device.

There could be several reasons for this. You need to inspect the device logs to get more details. Connect the device to your Mac, launch Xcode and select **Window > Organizer** from the menu. Select your device in the Organizer's left toolbar, then click on the "Console" tab and review the latest messages carefully. Additionally, you may need to investigate crash reports. You can find out how to obtain crash reports here: http://developer.apple.com/iphone/library/technotes/tn2008/tn2151.html.

### The Xcode Organizer console contains the message "killed by SpringBoard".

There is a poorly-documented time limit for an iOS application to render its first frames and process input. If your application exceeds this limit, it will be killed by SpringBoard. This may happen in an application with a first scene which is too large, for example. To avoid this problem, it is advisable to create a small initial scene which just displays a splash screen, waits a frame or two with **yield** and then starts loading the real scene. This can be done with code as simple as the following:

```
function Start () {
    yield;
    Application.LoadLevel("Test");
}
```

### Type.GetProperty() / Type.GetValue() cause crashes on the device

Currently **Type.GetProperty()** and **Type.GetValue()** are supported only for the **.NET 2.0 Subset** profile. You can select the .NET API compatibility level in the Player Settings.

**Note: Type.GetProperty()** and **Type.GetValue()** might be incompatible with managed code stripping and might need to be excluded (you can supply a custom non-strippable type list during the stripping process to accomplish this). For further details,

see the iOS player size optimization guide.

### The game crashes with the error message "ExecutionEngineException: Attempting to JIT compile method 'SometType`1<SomeValueType>:.ctor ()' while running with --aot-only."

The Mono .NET implementation for iOS is based on AOT (ahead of time compilation to native code) technology, which has its limitations. It compiles only those generic type methods (where a value type is used as a generic parameter) which are explicitly used by other code. When such methods are used only via reflection or from native code (ie, the serialization system) then they get skipped during AOT compilation. The AOT compiler can be hinted to include code by adding a dummy method somewhere in the script code. This can refer to the missing methods and so get them compiled ahead of time.

```
void _unusedMethod()
{
   var tmp = new SomeType<SomeValueType>();
}
```

**Note:** value types are basic types, enums and structs.

### Various crashes occur on the device when a combination of System.Security.Cryptography and managed code stripping is used

.NET Cryptography services rely heavily on reflection and so are not compatible with managed code stripping since this involves static code analysis. Sometimes the easiest solution to the crashes is to exclude the whole **System.Security.Crypography** namespace from the stripping process.

The stripping process can be customized by adding a custom **link.xml** file to the **Assets** folder of your Unity project. This specifies which types and namespaces should be excluded from stripping. Further details can be found in the iOS player size optimization guide.

**link.xml**

```
<linker>
      <assembly fullname="mscorlib">
            <namespace fullname="System.Security.Cryptography" preserve="all"/>
      </assembly>
</linker>
```

**Application crashes when using System.Security.Cryptography.MD5 with managed code stripping**

You might consider advice listed above or can work around this problem by adding extra reference to specific class to your script code:

```
object obj = new MD5CryptoServiceProvider();
```

### "Ran out of trampolines of type 1/2" runtime error

This error usually happens if you use lots of recursive generics. You can hint to the AOT compiler to allocate more trampolines of type 1 or type 2. Additional AOT compiler command line options can be specified in the "Other Settings" section of the Player Settings. For type 1 trampolines, specify **nrgctx-trampolines=ABCD**, where ABCD is the number of new trampolines required (i.e. 4096). For type 2 trampolines specify **nimt-trampolines=ABCD**.

### After upgrading Xcode Unity iOS runtime fails with message "You are using Unity iPhone Basic. You are not allowed to remove the Unity splash screen from your game"

With some latest Xcode releases there were changes introduced in PNG compression and optimization tool. These changes might cause false positives in Unity iOS runtime checks for splash screen modifications. If you encounter such problems try upgrading Unity to the latest publicly available version. If it does not help you might consider following workaround:

- Replace your Xcode project from scratch when building from Unity (instead of appending it)
- Delete already installed project from device
- Clean project in Xcode (*Product->Clean*)
- Clear Xcode's Derived Data folders (*Xcode->Preferences->Locations*)

If this still does not help try disabling PNG re-compression in Xcode:

- Open your Xcode project
- Select "Unity-iPhone" project there
- Select "Build Settings" tab there
- Look for "Compress PNG files" option and set it to NO

### App Store submission fails with "iPhone/iPod Touch: application executable is missing a required architecture. At least one of the following architecture(s) must be present: armv6" message

You might get such message when updating already existing application, which previously was submitted with armv6 support. Unity 4.x and Xcode 4.5 does not support armv6 platform anymore. To solve submission problem just set **Target OS Version** in Unity **Player Settings** to **4.3** or higher.

### WWW downloads are working fine in Unity Editor and on Android, but not on iOS

Most common mistake is to assume that WWW downloads are always happening on separate thread. On some platforms this might be true, but you should not take it for granted. Best way to track WWW status is either to use *yield* statement or check status in *Update* method. You should **not** use busy *while* loops for that.

### "PlayerLoop called recursively!" error occurs when using Cocoa via a native function called from a script

Some operations with the UI will result in iOS redrawing the window immediately (the most common example is adding a UIView with a UIViewController to the main UIWindow). If you call a native function from a script, it will happen inside Unity's PlayerLoop, resulting in PlayerLoop being called recursively. In such cases, you should consider using the performSelectorOnMainThread method with waitUntilDone set to false. It will inform iOS to schedule the operation to run between Unity's PlayerLoop calls.

### Profiler or Debugger unable to see game running on iOS device

- Check that you have built a Development build, and ticked the "Enable Script Debugging" and "Autoconnect profiler" boxes (as appropriate).
- The application running on the device will make a multicast broadcast to 225.0.0.222 on UDP port 54997. Check that your network settings allow this traffic. Then, the profiler will make a connection to the remote device on a port in the range 55000 - 55511 to fetch profiler data from the device. These ports will need to be open for UDP access.

### Missing DLLs

If your application runs ok in editor but you get errors in your iOS project this may be caused by missing DLLs (e.g. I18N.dll, I19N.West.dll). In this case, try copying those dlls from within the Unity.app to your project's Assets/Plugins folder. The location of the DLLs within the unity app is:

```
Unity.app/Contents/Frameworks/Mono/lib/mono/unity
```

You should then also check the stripping level of your project to ensure the classes in the DLLs aren't being removed when the build is optimised. Refer to the iOS Optimisation Page for more information on iOS Stripping Levels.

### Xcode Debugger console reports: ExecutionEngineException: Attempting to JIT compile method '(wrapper native-to-managed) Test:TestFunc (int)' while running with --aot-only

Typically such message is received when managed function delegate is passed to the native function, but required wrapper code wasn't generated when building application. You can help AOT compiler by hinting which methods will be passed as delegates to the native code. This can be done by adding "MonoPInvokeCallbackAttribute" custom attribute. Currently only static methods can be passed as delegates to the native code.

Sample code:

```
using UnityEngine;
using System.Collections;
using System;
using System.Runtime.InteropServices;
using AOT;

public class NewBehaviourScript : MonoBehaviour {

	[DllImport ("__Internal")]
	private static extern void DoSomething (NoParamDelegate del1, StringParamDelegate del2);
```

```
        delegate void NoParamDelegate ();
        delegate void StringParamDelegate (string str);

        [MonoPInvokeCallback (typeof (NoParamDelegate))]
        public static void NoParamCallback()
        {
                Debug.Log ("Hello from NoParamCallback");
        }

        [MonoPInvokeCallback (typeof (StringParamDelegate))]
        public static void StringParamCallback(string str)
        {
                Debug.Log (string.Format ("Hello from StringParamCallback {0}", str));
        }

        // Use this for initialization
        void Start () {
                DoSomething(NoParamCallback, StringParamCallback);
        }
}
```

## Android

# Troubleshooting Android development

### Unity fails to install your application to your device

1. Verify that your computer can actually see and communicate with the device. See the Publishing Builds page for further details.
2. Check the error message in the Unity console. This will often help diagnose the problem.

If you get an error saying "Unable to install APK, protocol failure" during a build then this indicates that the device is connected to a low-power USB port (perhaps a port on a keyboard or other peripheral). If this happens, try connecting the device to a USB port on the computer itself.

### Your application crashes immediately after launch.

1. Ensure that you are not trying to use NativeActivity with devices that do not support it.
2. Try removing any native plugins you have.
3. Try disabling stripping.
4. Use **adb logcat** to get the crash report from your device.

### Building DEX Failed

This an error which will produce a message like the following:-

```
Building DEX Failed!
G:\Unity\JavaPluginSample\Temp/StagingArea> java -Xmx1024M
-Djava.ext.dirs="G:/AndroidSDK/android-sdk_r09-windows\platform-tools/lib/"
-jar "G:/AndroidSDK/android-sdk_r09-windows\platform-tools/lib/dx.jar"
--dex --verbose --output=bin/classes.dex bin/classes.jar plugins
Error occurred during initialization of VM
Could not reserve enough space for object heap
Could not create the Java virtual machine.
```

This is usually caused by having the wrong version of Java installed on your machine. Updating your Java installation to the latest version will generally solve this issue.

### The game crashes after a couple of seconds when playing video

Make sure `Settings->Developer Options->Don't keep activities` isn't enabled on the phone. The video player is its own activity and therefore the regular game activity will be destroyed if the video player is activated.

### My game quits when I press the sleep button

Change the `<activity>` tag in the `AndroidManifest.xml` to contain `<android:configChanges>` tag as described here.

An example activity tag might look something like this:-

```
<activity android:name=".AdMobTestActivity"
                  android:label="@string/app_name"
                  android:configChanges="fontScale|keyboard|keyboardHidden|locale|mnc|mcc|navigation|orientation|screenLayou
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
```

Page last updated: 2012-11-26

---

## android-bugreporting

Before submitting a bug with just "it crashes" in the message body, please look through the Troubleshooting Android development page first.

At this point there are no advanced debug tools to investigate on-device app crashes. However you can use **adb** application (found under Android-SDK/platform-tools) with **logcat** parameter. It prints status reports from your device. These reports may include information related to the occurred crash.

If you are sure that the crash you're experiencing happens due to a bug in Unity software, please save the **adb logcat** output, conduct a repro project and use the bugreporter (**Help/Report a bug**) to inform us about it. We will get back to you as soon as we can.

Page last updated: 2011-02-24

---

## android-unsupported

### Graphics

- Non-square textures are not supported by the ETC format.
- Movie Textures are not supported, use a full-screen streaming playback instead. Please see the Movie playback page for more information.

### Scripting

- **OnMouseEnter**, **OnMouseOver**, **OnMouseExit**, **OnMouseDown**, **OnMouseUp**, and **OnMouseDrag** events are not supported on Android.
- Dynamic features like Duck Typing are not supported. Use #pragma strict for your scripts to force the compiler to report dynamic features as errors.
- Video streaming via **WWW** class is not supported.

Page last updated: 2012-10-08

---

# android-OBBsupport

### Support for Split Application Binary (.OBB)

Under *Player Settings | Publishing Settings* you'll find the option to split the application binary (.apk) into expansion files (.apk + .obb).

This mechanism is only necessary when publishing to the Google Play Store, if the application is larger than 50 MB. See http://developer.android.com/guide/google/play/expansion-files.html for further information on APK Expansion Files.

When the *Split Application Binary* option is enabled the player executable and data will be split up, with a generated .apk (main application binary) consisting only of the executable (Java, Native) code (around 10MB), any and all script / plugin code, and the data for the first scene. Everything else (all additional scenes, resources, streaming assets ...) will be serialized separately to a APK Expansion File (.obb).

- When starting an .apk built with *Split Application Binary* enabled the application will check to see if it can access the .obb file from it's position on the sdcard (location explained in the Apk Expansion docs from Google).
- If the expansion file (.obb) cannot be found, only the first level can accessed (since the rest of the data is in the .obb).
- The first level is then required to make the .obb file available on sdcard, before the application can proceed to load subsequent scenes/data.
- If the .obb is found the Application.dataPath will switch from .apk path, to instead point to .obb. Downloading the .obb is then not necessary.
- The contents of the .obb are never used manually. Always treat the .apk+.obb as a unique bundle, the same way you would treat a single big .apk.

The *Split Application Binary* option is not the only way to split an .apk into .apk/.obb (other options include 3rd party plugins/asset bundles/etc), but it's the only **automatic** splitting mechanism officially supported.

#### Downloading of the expansion file (.OBB)

- The expansion file (.obb) may (but it's not required, in its current form at least) to be hosted on the Google Play servers.
- If the .obb is published together with the .apk on Google Play, you must also include code to download the .obb. (for those devices that require it, and for scenarios where the .obb is lost)
- The asset store has a plugin (adapted from the Google Apk Expansion examples) which does this for you. It will download the .obb and put it in the right place on the sdcard. See http://u3d.as/content/unity-technologies/google-play-obb-downloader/2Qq
- When using the asset store plugin you **need** to call that plugin from the first scene (because of the reasons explained above).
- The asset store plugin can also be used to download .obb's created in some other way (single data file, a zip of asset bundles, etc) - it's agnostic to how the .obb was created.

Page last updated: 2012-11-14

---

# Android Player Settings

**Player Settings** is where you define various parameters (platform specific) for the final game that you will build in Unity. Some of these values for example are used in the **Resolution Dialog** that launches when you open a standalone game, others are used by XCode when building your game for the iOS devices, so it's important to fill them out correctly.

To see the Player Settings choose **Edit->Project Settings->Player** from the menu bar.

PlayerSettings

Cross-Platform Settings
Company Name
Product Name TestProject
Default Icon None (Texture2D) Select
Default Cursor None (Texture2D) Select
Cursor Hotspot X 0 Y 0

*Global Settings that apply to any project you create.*

**Cross-Platform Properties**

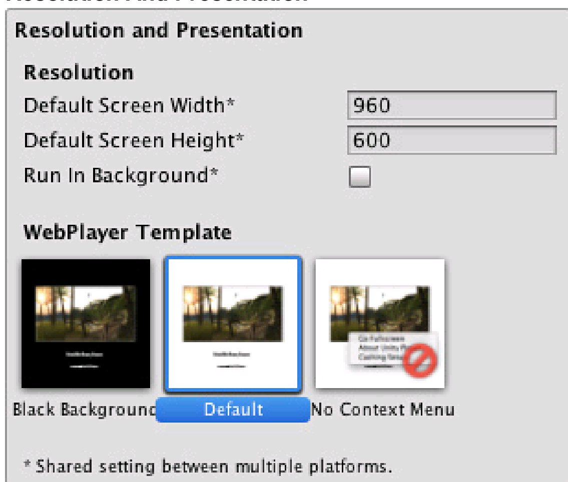
| | |
|---|---|
| **Company Name** | The name of your company. This is used to locate the preferences file. |
| **Product Name** | The name that will appear on the menu bar when your game is running and is used to locate the preferences file also. |
| **Default Icon** | Default icon the application will have on every platform (You can override this later for platform specific needs). |

## Per-Platform Settings

## Desktop

### Web-Player

***Resolution And Presentation***




***Resolution***

| | |
|---|---|
| **Default Screen Width** | Screen Width the player will be generated with. |
| **Default Screen Height** | Screen Height the plater will be generated with. |
| **Run in background** | Check this if you dont want to stop executing your game if the player looses focus. |
| ***WebPlayer Template*** | For more information you should check the "Using WebPlayer templates page", note that for each built-in and custom template there will be an icon in this section. |

***Icon***

Icons don't have any meaning for most webplayer builds but they are needed for Native Client builds used as Chrome applications. You can set these icons here.

***Other Settings***

| | |
|---|---|
| ***Rendering*** | |
| **Rendering Path** | This property is shared between Standalone and WebPlayer content. |
| **Vertex Lit** | Lowest lighting fidelity, no shadows support. Best used on old machines or limited mobile platforms. |
| **Forward with Shaders** | Good support for lighting features; limited support for shadows. |
| **Deferred Lighting** | Best support for lighting and shadowing features, but requires certain level of hardware support. Best used if you have many realtime lights. Unity Pro only. |
| **Color Space** | The color space to be used for rendering |
| **GammaSpace Rendering** | Rendering is gamma-corrected |
| **Linear Rendering Hardware Sampling** | Rendering is done in linear space |
| **Static Batching** | Set this to use Static batching on your build (Inactive by default in webplayers). Unity Pro only. |
| **Dynamic Batching** | Set this to use Dynamic Batching on your build (Activated by default). |
| ***Streaming*** | |
| **First Streamed Level** | If you are publishing a Streamed Web Player, this is the index of the first level that will have access to all Resources.Load assets. |
| **Optimization** | |
| **Optimize Mesh Data** | Remove any data from meshes that is not required by the material applied to them (tangents, normals, colors, UV). |
| **Debug Unload Mode** | Output debugging information regarding Resources.UnloadUnusedAssets. |

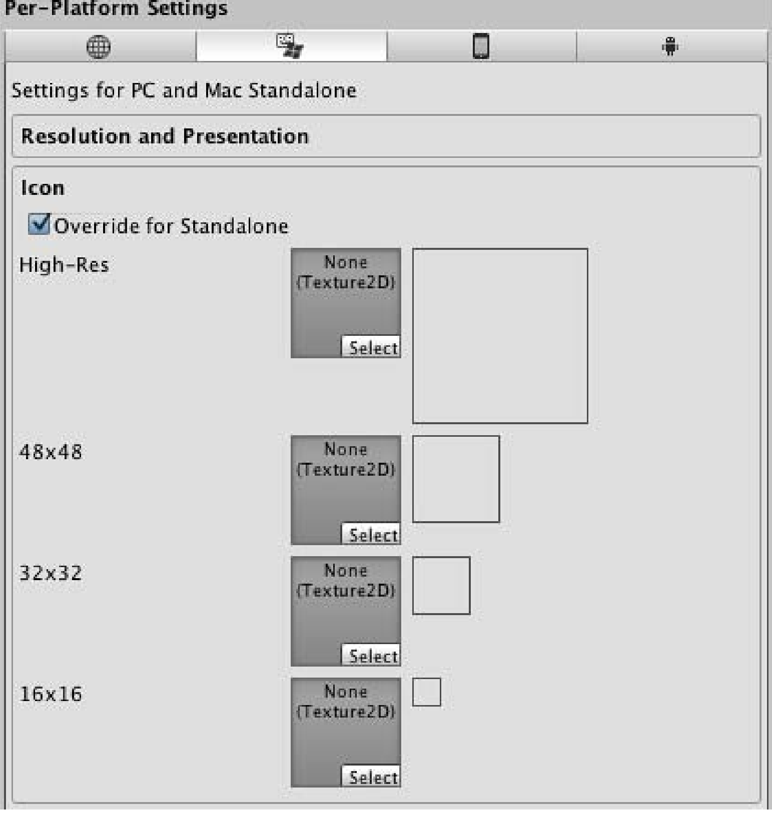
| | |
|---|---|
| **Disabled** | Don't output any debug data for UnloadUnusedAssets. |
| **Overview only** | Minimal stats about UnloadUnusedAssets usage. |
| **Full (slow)** | Output overview stats along with stats for all affected objects. This option can slow down execution due to the amount of data being displayed. |

### Standalone

***Resolution And Presentation***

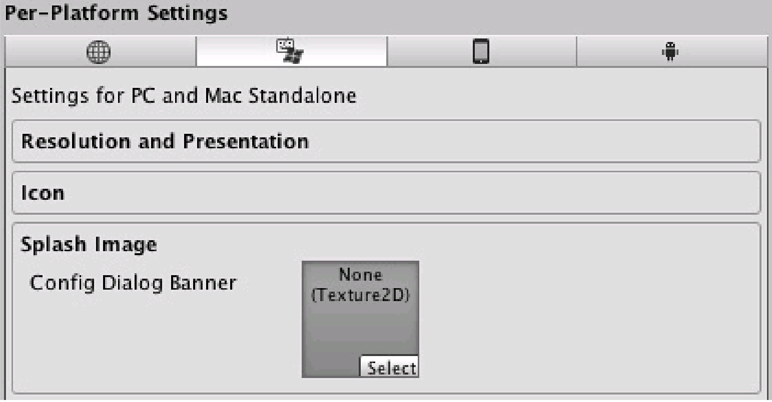
| | |
|---|---|
| ***Resolution*** | |
| **Default Screen Width** | Screen Width the stand alone game will be using by default. |
| **Default Screen Height** | Screen Height the plater will be using by default. |
| **Run in background** | Check this if you dont want to stop executing your game if it looses focus. |
| ***Standalone Player Options*** | |
| **Default is Full Screen** | Check this if you want to start your game by default in full screen mode. |
| **Capture Single Screen** | If enabled, standalone games in fullscreen mode will not darken the secondary monitor in multi-monitor setups. |
| **DisplayResolution Dialog** | |
| **Disabled** | No resolution dialog will appear when starting the game. |
| **Enabled** | Resolution dialog will always appear when the game is launched. |
| **Hidden by default** | The resolution player is possible to be opened only if you have pressed the "alt" key when starting the game. |
| **Use Player Log** | Write a log file with debugging information. If you plan to submit your application to the Mac App Store you will want to leave this option un-ticked. Ticked is the default. |
| **Mac App Store Validation** | Enable receipt validation for the Mac App Store. |
| **Supported Aspect Ratios** | Aspect Ratios selectable in the Resolution Dialog will be monitor-supported resolutions of enabled items from this list. |

***Icon***




| | |
|---|---|
| **Override for Standalone** | Check if you want to assign a custom icon you would like to be used for your standalone game. Different sizes of the icon should fill in the squares below. |

***Splash Image***




| | |
|---|---|
| **Config Dialog Banner** | Add your custom splash image that will be displayed when the game is starting. |

***Other Settings***

Other Settings
Rendering
Rendering Path* Forward
Color Space* Gamma
Static Batching
Dynamic Batching
Optimization
Api Compatibility Level .NET 2.0 Sub
Optimize Mesh Data*
Debug Unload Mode* Disabled
* Shared setting between multiple platforms.

| | |
|---|---|
| ***Rendering*** | |
| **Rendering Path** | This property is shared between Standalone and WebPlayer content. |
| **Vertex Lit** | Lowest lighting fidelity, no shadows support. Best used on old machines or limited mobile platforms. |
| **Forward with Shaders** | Good support for lighting features; limited support for shadows. |
| **Deferred Lighting** | Best support for lighting and shadowing features, but requires certain level of hardware support. Best used if you have many realtime lights. Unity Pro only. |
| **Color Space** | The color space to be used for rendering |
| **GammaSpace Rendering** | Rendering is gamma-corrected |
| **Linear Rendering Hardware Sampling** | Rendering is done in linear space |
| **Static Batching** | Set this to use Static batching on your build (Inactive by default in webplayers). Unity Pro only. |
| **Dynamic Batching** | Set this to use Dynamic Batching on your build (Activated by default). |
| **Optimization** | |
| **API Compatibility Level** | |
| **.Net 2.0** | .Net 2.0 libraries. Maximum .net compatibility, biggest file sizes |
| **.Net 2.0 Subset** | Subset of full .net compatibility, smaller file sizes |
| **Optimize Mesh Data** | Remove any data from meshes that is not required by the material applied to them (tangents, normals, colors, UV). |
| **Debug Unload Mode** | Output debugging information regarding Resources.UnloadUnusedAssets. |
| **Disabled** | Don't output any debug data for UnloadUnusedAssets. |
| **Overview only** | Minimal stats about UnloadUnusedAssets usage. |
| **Full (slow)** | Output overview stats along with stats for all affected objects. This option can slow down execution due to the amount of data being displayed. |

## iOS

***Resolution And Presentation***

Resolution and Presentation
Resolution
Default Orientation* Portrait
Status Bar
Status Bar Hidden
Status Bar Style Default
Use 32-bit Display Buffer*
Show Loading Indicator Don't Show
* Shared setting between multiple platforms.

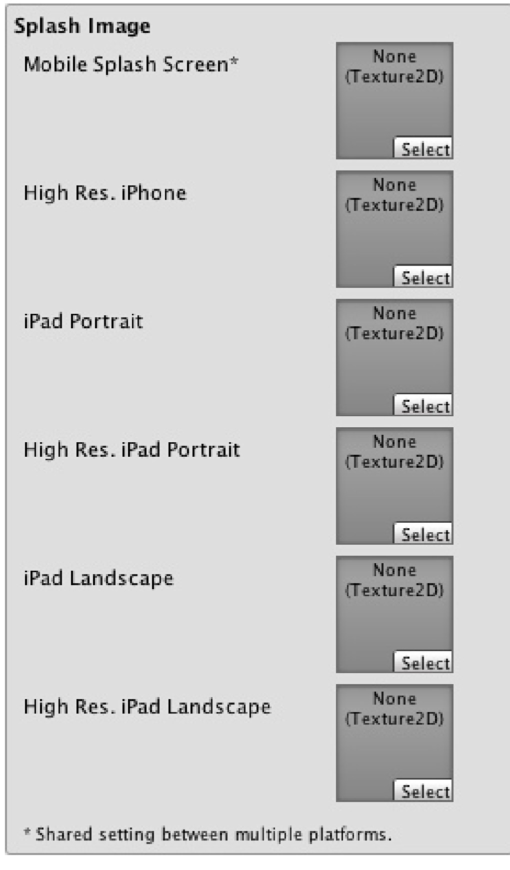
| | |
|---|---|
| ***Resolution*** | |
| **Default Orientation** | (This setting is shared between iOS and Android devices) |
| **Portrait** | The device is in portrait mode, with the device held upright and the home button at the bottom. |
| **Portrait Upside Down (iOS Only)** | The device is in portrait mode but upside down, with the device held upright and the home button at the top. |
| **Landscape Right (iOS Only)** | The device is in landscape mode, with the device held upright and the home button on the **left** side. |
| **Landscape Left** | The device is in landscape mode, with the device held upright and the home button on the **right** side. |
| **Auto Rotation** | The screen orientation is automatically set based on the physical device orientation. |
| ***Auto Rotation settings*** | |
| **Use Animated Autorotation** | When checked, orientation change is animated. This only applies when Default orientation is set to Auto Rotation. |
| ***Allowed Orientations for Auto Rotation*** | |
| **Portrait** | When checked, portrait orientation is allowed. This only applies when Default orientation is set to Auto Rotation. |
| **Portrait Upside Down** | When checked, portrait upside down orientation is allowed. This only applies when Default orientation is set to Auto Rotation. |
| **Landscape Right** | When checked, landscape right (home button on the **left** side) orientation is allowed. This only applies when Default orientation is set to Auto Rotation. |

| | |
|---|---|
| **Landscape Left** | When checked, landscape left (home button is on the **right** side) orientation is allowed. This only applies when Default orientation is set to Auto Rotation. |
| ***Status Bar*** | |
| **Status Bar Hidden** | Specifies whether the status bar is initially hidden when the application launches. |
| **Status Bar Style** | Specifies the style of the status bar as the application launches |
| **Default** | |
| **Black Translucent** | |
| **Black Opaque** | |
| **Use 32-bit Display Buffer** | Specifies if Display Buffer should be created to hold 32-bit color values (16-bit by default). Use it if you see banding, or need alpha in your ImageEffects, as they will create RTs in same format as Display Buffer. |
| **Show Loading Indicator** | Options for the loading indicator |
| **Don't Show** | No indicator |
| **White Large** | Indicator shown large and in white |
| **White** | Indicator shown at normal size in white |
| **Gray** | Indicator shown at normal size in gray |

***Icon***

| | |
|---|---|
| **Override for iOS** | Check if you want to assign a custom icon you would like to be used for your iPhone/iPad game. Different sizes of the icon should fill in the squares below. |
| **Prerendered icon** | If unchecked iOS applies sheen and bevel effects to the application icon. |

***Splash Image***




| | |
|---|---|
| **Mobile Splash Screen (Pro-only feature)** | Specifies texture which should be used for iOS Splash Screen. Standard Splash Screen size is 320x480.(This is shared between Android and iOS) |
| **High Res. iPhone (Pro-only feature)** | Specifies texture which should be used for iOS 4th gen device Splash Screen. Splash Screen size is 640x960. |
| **iPad Portrait (Pro-only feature)** | Specifies texture which should be used as iPad Portrait orientation Splash Screen. Standard Splash Screen size is 768x1024. |
| **High Res iPad Portrait (Pro-only feature)** | Specifies texture which should be used as iPad Portrait orientation Splash Screen. Standard Splash Screen size is 1536x2048. |
| **iPad Landscape (Pro-only feature)** | Specifies texture which should be used as iPad Landscape orientation Splash Screen. Standard Splash Screen size is 1024x768. |
| **High Res iPad Landscape (Pro-only feature)** | Specifies texture which should be used as iPad Portrait orientation Splash Screen. Standard Splash Screen size is 2048x1536. |

***Other Settings***

Other Settings
Rendering
Static Batching
Dynamic Batching
Identification
Bundle Identifier* com.Company.I
Bundle Version* 1.0
Configuration
Target Device iPhone + iPad
Target Platform* Universal armv6
Target Resolution Native (default c
Accelerometer Frequency 60 Hz
Override IPod Music
Requires Persistent WiFi
Exit on Suspend
Optimization
Api Compatibility Level .NET 2.0 Subset
AOT Compilation Options
SDK Version iOS latest
Target iOS Version 3.1.3
Stripping Level* Disabled
Script Call Optimization Slow and Safe
Optimize Mesh Data*
Debug Unload Mode* Disabled
* Shared setting between multiple platforms.

| | |
|---|---|
| ***Rendering*** | |
| **Static Batching** | Set this to use Static batching on your build (Activated by default). Pro-only feature. |
| **Dynamic Batching** | Set this to use Dynamic Batching on your build (Activated by default). |
| ***Identification*** | |
| **Bundle Identifier** | The string used in your provisioning certificate from your Apple Developer Network account(This is shared between iOS and Android) |
| **Bundle Version** | Specifies the build version number of the bundle, which identifies an iteration (released or unreleased) of the bundle. This is a monotonically increased string, comprised of one or more period-separated |
| ***Configuration*** | |
| **Target Device** | Specifies application target device type. |
| **iPhone Only** | Application is targeted for iPhone devices only. |
| **iPad Only** | Application is targeted for iPad devices only. |
| **iPhone + iPad** | Application is targeted for both iPad and iPhone devices. |
| **Target Platform** | Specifies the target arquitecture you are going to build for.(This setting is shared between iOS and Android Platforms) |
| **armv6 (OpenGL ES1.1)** | Application is optimized for armv6 chipsets |
| **Universal armv6+armv7 (OpenGL ES1.1+2.0)** | Application supports both armv6 and armv7 chipsets. *Note: increases application distribution size* |
| **armv7** | Application is optimized for armv7 chipsets. 1st-2nd gen. devices are not supported. There might be additional requirements for this build target imposed by Apple App Store. Defaults to OpenGL ES 2.0. |
| **Target Resolution** | Resolution you want to use on your deployed device.(This setting will not have any effect on devices with maximum resolution of 480x320) |
| **Native(Default Device Resolution)** | Will use the device native resolution. |
| **Standard(Medium or Low Resolution)** | Use the lowest resolution possible (480x320). |

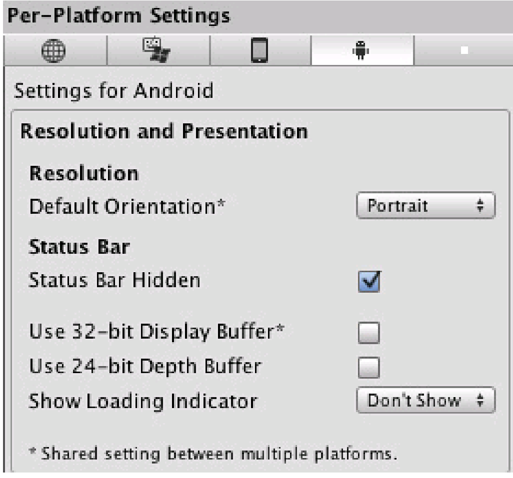
| | |
|---|---|
| **HD(Highest available resolution)** | Use the maximum resolution allowed on the device (960x640). |
| **Accelerometer Frequency** | How often the accelerometer is sampled |
| **Disabled** | Accelerometer is not sampled |
| **15Hz** | 15 samples per second |
| **30Hz** | 30 samples per second |
| **60Hz** | 60 samples per second |
| **100Hz** | 100 samples per second |
| **Override iPod Music** | If selected application will silence user's iPod music. Otherwise user's iPod music will continue playing in the background. |
| **UI Requires Persistent WiFi** | Specifies whether the application requires a Wi-Fi connection. iOS maintains the active Wi-Fi connection open while the application is running. |
| **Exit on Suspend** | Specifies whether the application should quit when suspended to background on iOS versions that support multitasking. |
| ***Optimization*** | |
| **Api Compatibility Level** | Specifies active .NET API profile |
| **.Net 2.0** | .Net 2.0 libraries. Maximum .net compatibility, biggest file sizes |
| **.Net 2.0 Subset** | Subset of full .net compatibility, smaller file sizes |
| **AOT compilation options** | Additional AOT compiler options. |
| **SDK Version** | Specifies iPhone OS SDK version to use for building in Xcode |
| **iOS 4.0** | iOS SDK 4.0. |
| **iOS Simulator 4.0** | iOS Simulator 4.0. Application built for this version of SDK will be able to run only on Simulator from the SDK 4. |
| **iOS 4.1** | iOS 4.1. |
| **iOS Simulator 4.1** | iOS Simulator 4.1. Application built for this version of SDK will be able to run only on Simulator from the SDK 4.x. |
| **iOS 4.2** | iOS 4.2. |
| **iOS Simulator 4.2** | iOS Simulator 4.2. Application built for this version of SDK will be able to run only on Simulator from the SDK 4.x. |
| **iOS 4.3** | iOS 4.3. |
| **iOS Simulator 4.3** | iOS Simulator 4.3. Application built for this version of SDK will be able to run only on Simulator from the SDK 4.x. |
| **iOS 5.0** | iOS 5.0 |
| **iOS Simulator 5.0** | iOS Simulator 5.0. Application built for this version of SDK will be able to run only on Simulator from the SDK 5.x. |
| **iOS latest** | Latest available iOS SDK. Available since iOS SDK 4.2. (default value) |
| **iOS Simulator latest** | Latest available iOS Simulator SDK. Available since iOS SDK 4.2. |
| **Unknown** | iOS SDK version is not managed by Unity Editor. |
| **Target iOS Version** | Specifies lowest iOS version where final application will able to run |
| **3.0** | iPhone OS 3.0. (default value) |
| **3.1** | iPhone OS 3.1. |
| **3.1.2** | iPhone OS 3.1.2. |
| **3.1.3** | iPhone OS 3.1.3. |
| **3.2** | iPhone OS 3.2. |
| **4.0** | iPhone OS 4.0. |
| **4.1** | iPhone OS 4.1. |
| **4.2** | iPhone OS 4.2. |
| **4.3** | iPhone OS 4.3. |
| **5.0** | iPhone OS 5.0 |
| **Unknown** | iPhone OS SDK version is not managed by Unity Editor. |
| **Stripping Level (Pro-only feature)** | Options to strip out scripting features to reduce built player size(This setting is shared between iOS and Android Platforms) |
| **Disabled** | No reduction is done. |
| **Strip Assemblies** | Level 1 size reduction. |
| **Strip ByteCode** | Level 2 size reduction (includes reductions from Level 1). |
| **Use micro mscorlib** | Level 3 size reduction (includes reductions from Levels 1 and 2). |
| **Script Call Optimization** | Optionally disable exception handling for a speed boost at runtime |

| | |
|---|---|
| **Slow and Safe** | Full exception handling will occur with some performance impact on the device |
| **Fast but no Exceptions** | No data provided for exceptions on the device, but the game will run faster |
| **Optimize Mesh Data** | Remove any data from meshes that is not required by the material applied to them (tangents, normals, colors, UV). |
| **Debug Unload Mode** | Output debugging information regarding Resources.UnloadUnusedAssets. |
| **Disabled** | Don't output any debug data for UnloadUnusedAssets. |
| **Overview only** | Minimal stats about UnloadUnusedAssets usage. |
| **Full (slow)** | Output overview stats along with stats for all affected objects. This option can slow down execution due to the amount of data being displayed. |

**Note:** If you build for example for iPhone OS 3.2, and then select Simulator 3.2 in Xcode you will get a ton of errors. So you **MUST** be sure to select a proper Target SDK in Unity Editor.

## Android

***Resolution And Presentation***

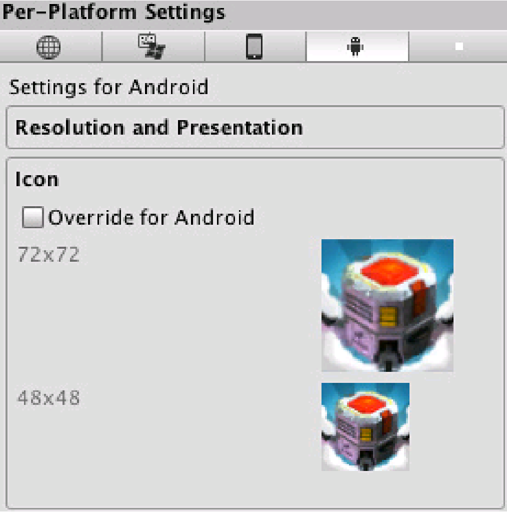



*Resolution and presentation for your Android project builds.*

***Resolution***

| | |
|---|---|
| **Default Orientation** | (This setting is shared between iOS and Android devices) |
| **Portrait** | The device is in portrait mode, with the device held upright and the home button at the bottom. |
| **Portrait Upside Down** | The device is in portrait mode but upside down, with the device held upright and the home button at the top (only available with Android OS 2.3 and later). |
| **Landscape Right** | The device is in landscape mode, with the device held upright and the home button on the **left** side (only available with Android OS 2.3 and later). |
| **Landscape Left** | The device is in landscape mode, with the device held upright and the home button on the **right** side. |
| **Use 32-bit Display Buffer** | Specifies if Display Buffer should be created to hold 32-bit color values (16-bit by default). Use it if you see banding, or need alpha in your ImageEffects, as they will create RTs in same format as Display Buffer. Not supported on devices running pre-Gingerbread OS (will be forced to 16-bit). |
| **Use 24-bit Depth Buffer** | If set Depth Buffer will be created to hold (at least) 24-bit depth values. Use it only if you see 'z-fighting' or other artifacts, as it may have performance implications. |

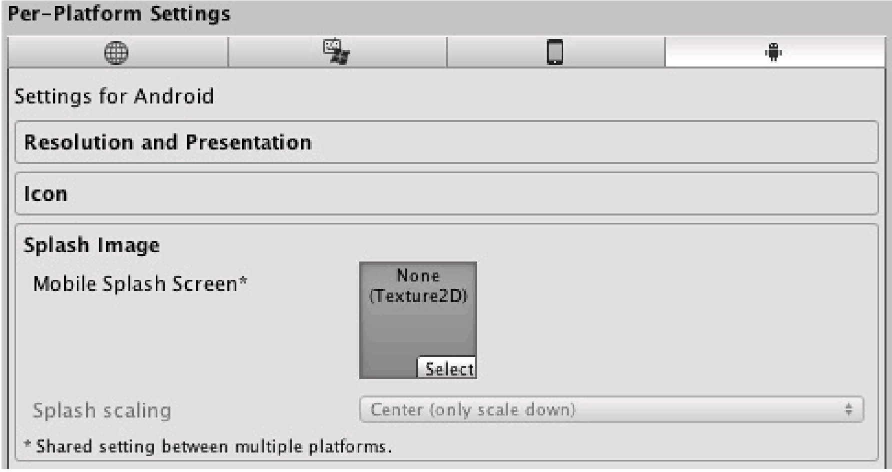
**Icon**




*Different icons that your project will have when built.*

| | |
|---|---|
| **Override for Android** | Check if you want to assign a custom icon you would like to be used for your Android game. Different sizes of the icon should fill in the squares below. |

**Splash Image**




*Splash image that is going to be displayed when your project is launched.*

| | |
|---|---|
| **Mobile Splash Screen (Pro-only feature)** | Specifies texture which should be used by the iOS Splash Screen. Standard Splash Screen size is 320x480.(This is shared between Android and iOS) |
| **Splash Scaling** | Specifies how will be the splash image scaling on the device. |

**Other Settings**

**Other Settings**

| Setting | Value |
|---|---|
| **Rendering** | |
| Static Batching | ☑ |
| Dynamic Batching | ☑ |
| **Identification** | |
| Bundle Identifier* | com.Company.I |
| Bundle Version* | 1.0 |
| Bundle Version Code | 1 |
| Minimum API Level | Android 2.0.1 'I |
| **Configuration** | |
| Device Filter | ARMv7 only |
| Graphics Level* | OpenGL ES 2.0 |
| Install Location | Prefer External |
| Internet Access | Auto |
| Write Access | Internal Only |
| **Optimization** | |
| Api Compatibility Level | .NET 2.0 Subset |
| Stripping Level* | Disabled |
| Enable 'logcat' Profiler | ☐ |
| Optimize Mesh Data* | ☐ |
| Debug Unload Mode* | Disabled |

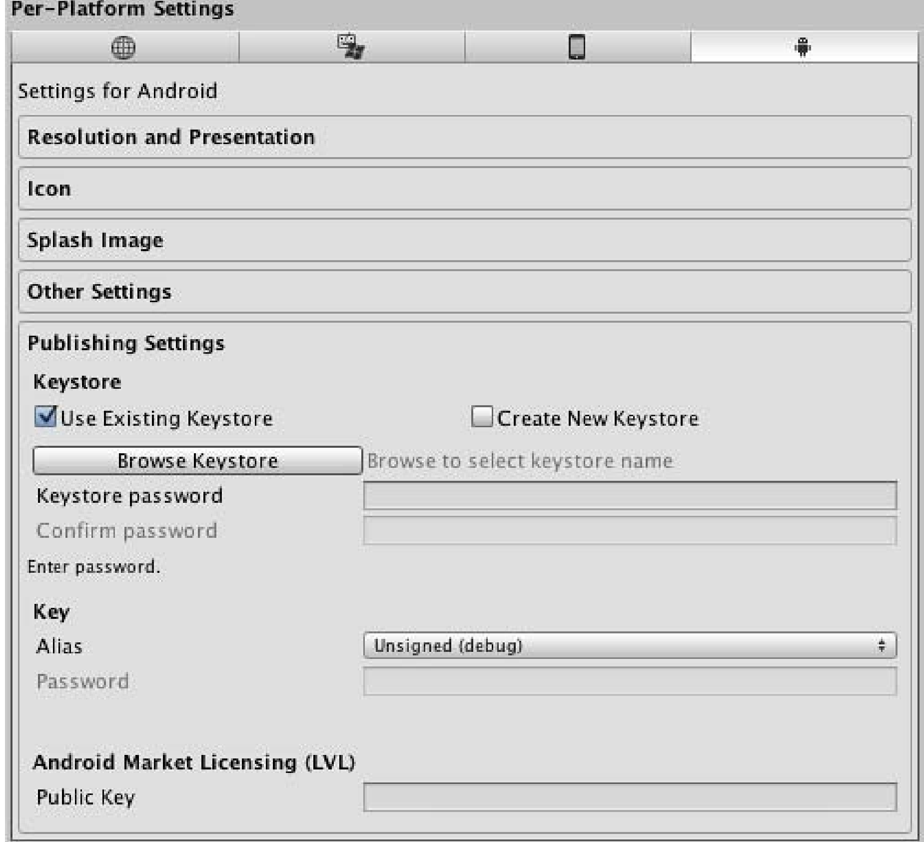
* Shared setting between multiple platforms.

| | |
|---|---|
| ***Rendering*** | |
| **Static Batching** | Set this to use Static batching on your build (Activated by default). Pro-only feature. |
| **Dynamic Batching** | Set this to use Dynamic Batching on your build (Activated by default). |
| ***Identification*** | |
| **Bundle Identifier** | The string used in your provisioning certificate from your Apple Developer Network account(This is shared between iOS and Android) |
| **Bundle Version** | Specifies the build version number of the bundle, which identifies an iteration (released or unreleased) of the bundle. This is a monotonically increased string, comprised of one or more period-separated(This is shared between iOS and Android) |
| **Bundle Version Code** | An internal version number. This number is used only to determine whether one version is more recent than another, with higher numbers indicating more recent versions. This is not the version number shown to users; that number is set by the versionName attribute. The value must be set as an integer, such as "100". You can define it however you want, as long as each successive version has a higher number. For example, it could be a build number. Or you could translate a version number in "x.y" format to an integer by encoding the "x" and "y" separately in the lower and upper 16 bits. Or you could simply increase the number by one each time a new version is released. |
| ***Configuration*** | |
| **Device Filter** | Specifies the target architecture you are going to build for. |
| **ARMv7 only** | Application optimized for ARMv7 CPU architecture. It will also enable correct Android Market device filtering, thus recommended for publishing to the Android Market (only devices supporting Unity Android will list the application on the Android Market). |

| | |
|---|---|
| **Graphics Level** | Select either ES 1.1 ('fixed function') or ES 2.0 ('shader based') Open GL level. When using the AVD (emulator) only ES 1.x is supported. |
| **Install Location** | Specifies application install location on the device (for detailed information, please refer to http://developer.android.com/guide/appendix/install-location.html). |
| **Automatic** | Let OS decide. User will be able to move the app back and forth. |
| **Prefer External** | Install app to external storage (SD-Card) if possible. OS does not guarantee that will be possible; if not, the app will be installed to internal memory. |
| **Force Internal** | Force app to be installed into internal memory. User will be unable to move the app to external storage. |

| | |
|---|---|
| **Internet Access** | When set to Require, will enable networking permissions even if your scripts are not using this. Automatically enabled for development builds. |
| **Write Access** | When set to External (SDCard), will enable write access to external storage such as the SD-Card. Automatically enabled for development builds. |
| ***Optimization*** | |
| **Api Compatibility Level** | Specifies active .NET API profile |
| **.Net 2.0** | .Net 2.0 libraries. Maximum .net compatibility, biggest file sizes |
| **.Net 2.0 Subset** | Subset of full .net compatibility, smaller file sizes |
| **Stripping Level (Pro-only feature)** | Options to strip out scripting features to reduce built player size(This setting is shared between iOS and Android Platforms) |
| **Disabled** | No reduction is done. |
| **Strip Assemblies** | Level 1 size reduction. |
| **Strip ByteCode (iOS only)** | Level 2 size reduction (includes reductions from Level 1). |
| **Use micro mscorlib** | Level 3 size reduction (includes reductions from Levels 1 and 2). |
| **Enable "logcat" profiler** | Enable this if you want to get feedback from your device while testing your projects. So adb logcat prints logs from the device to the console (only available in development builds). |
| **Optimize Mesh Data** | Remove any data from meshes that is not required by the material applied to them (tangents, normals, colors, UV). |
| **Debug Unload Mode** | Output debugging information regarding Resources.UnloadUnusedAssets. |
| **Disabled** | Don't output any debug data for UnloadUnusedAssets. |
| **Overview only** | Minimal stats about UnloadUnusedAssets usage. |
| **Full (slow)** | Output overview stats along with stats for all affected objects. This option can slow down execution due to the amount of data being displayed. |

### Publishing Settings




*Publishing settings for Android Market*

| | |
|---|---|
| ***Keystore*** | |
| **Use Existing Keystore / Create New Keystore** | Use this to choose whether to create a new Keystore or use an existing one. |
| **Browse Keystore** | Lets you select an existing Keystore. |
| **Keystore password** | Password for the Keystore. |

| | |
|---|---|
| **Confirm password** | Password confirmation, only enabled if the Create New Keystore option is chosen. |
| ***Key*** | |
| **Alias** | Key alias |
| **Password** | Password for key alias |

Note that for security reasons, Unity will save neither the keystore password nor the key password.

| | |
|---|---|
| **Split Application Binary** | Split application binary into expansion files, for use with Google Play Store if application is larger than 50 MB. When enabled the player executable and data will be split up, with a generated .apk consisting only of the executable (Java and Native) code (~10MB), and the data for the first scene. The application data will be serialized separately to an APK Expansion File (.obb). |

### Flash

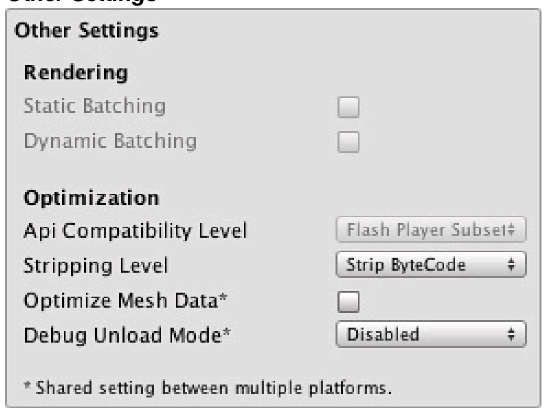
***Resolution And Presentation***

Resolution and Presentation
Resolution
Default Screen Width* 960
Default Screen Height* 600
* Shared setting between multiple platforms.

***Resolution***

| | |
|---|---|
| **Default Screen Width** | Screen Width the player will be generated with. |
| **Default Screen Height** | Screen Height the plater will be generated with. |

***Other Settings***




***Optimization***

| | |
|---|---|
| **Stripping** | Bytecode can optionally be stripped during the build. |
| **Optimize Mesh Data** | Remove any data from meshes that is not required by the material applied to them (tangents, normals, colors, UV). |
| **Debug Unload Mode** | Output debugging information regarding Resources.UnloadUnusedAssets. |
| **Disabled** | Don't output any debug data for UnloadUnusedAssets. |
| **Overview only** | Minimal stats about UnloadUnusedAssets usage. |
| **Full (slow)** | Output overview stats along with stats for all affected objects. This option can slow down execution due to the amount of data being displayed. |

# Details

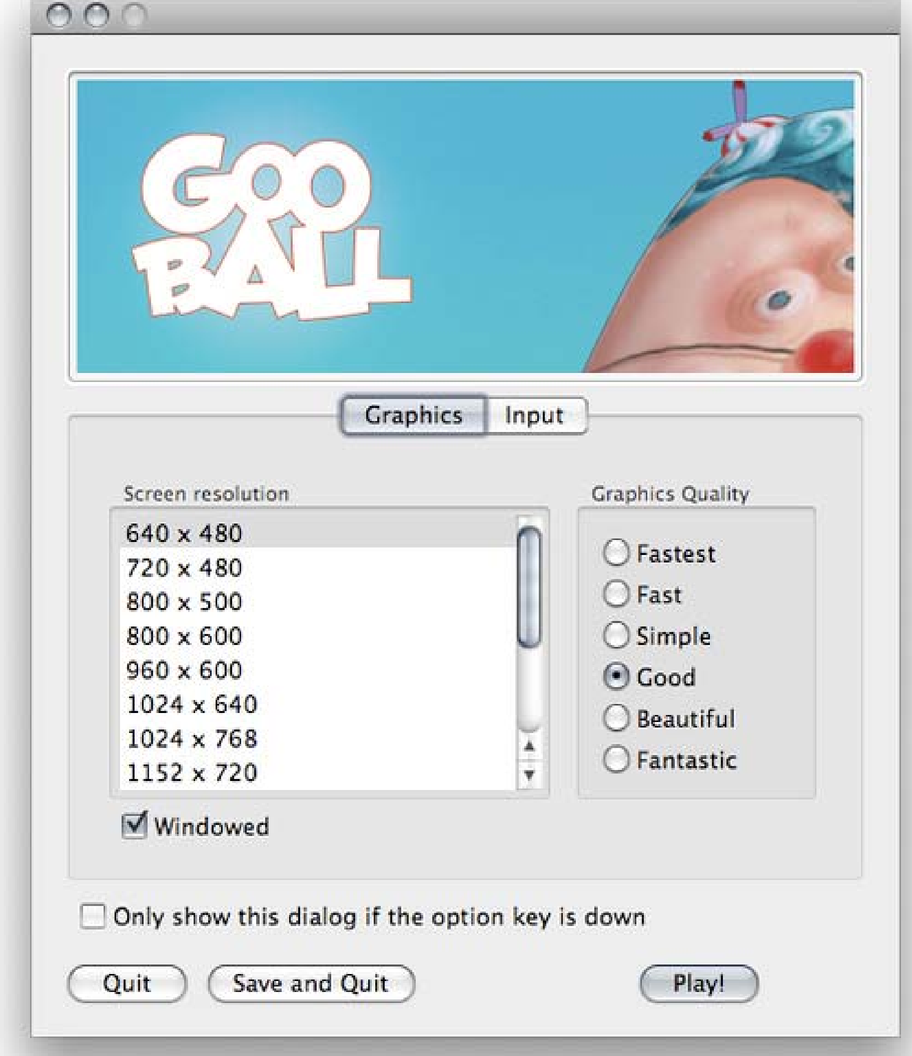
## Desktop

The Player Settings window is where many technical preference defaults are set. See also Quality Settings where the different graphics quality levels can be set up.

**Publishing a web player**
**Default Web Screen Width** and **Default Web Screen Height** determine the size used in the html file. You can modify the size in the html file later.

**Default Screen Width** and **Default Screen Height** are used by the Web Player when entering fullscreen mode through the context menu in the Web Player at runtime.

**Customizing your Resolution Dialog**




*The Resolution Dialog, presented to end-users*

You have the option of adding a custom banner image to the Screen Resolution Dialog in the Standalone Player. The maximum image size is 432 x 163 pixels. The image will not be scaled up to fit the screen selector. Instead it will be centered and cropped.

**Publishing to Mac App Store**
**Use Player Log** enables writing a log file with debugging information. This is useful to find out what happened if there are problems with your game. When publishing games for Apple's Mac App Store, it is recommended to turn this off, because Apple may reject your submission otherwise. See this manual page for further information about log files.

**Use Mac App Store Validation** enables receipt validation for the Mac App Store. If this is enabled, your game will only run when it contains a valid receipt from the Mac App Store. Use this when submitting games to Apple for publishing on the App Store. This prevents people from running the game on any computer then the one it was purchased on. Note that this feature does not implement any strong copy protection. In particular, any potential crack against one Unity game would work against any other Unity content. For this reason, it is recommended that you implement your own receipt validation code on top of this using Unity's plugin feature. However, since Apple requires plugin validation to initially happen before showing the screen setup dialog, you should still enable this check, or Apple might reject your submission.

## iOS

**Bundle Identifier**
The **Bundle Identifier** string must match the provisioning profile of the game you are building. The basic structure of the identifier is **com.CompanyName.GameName**. This structure may vary internationally based on where you live, so always default to the string provided to you by Apple for your Developer Account. Your GameName is set up in your provisioning certificates, that are manageable from the Apple iPhone Developer Center website. Please refer to the Apple iPhone Developer Center website for more information on how this is performed.

**Stripping Level (Pro-only)**
Most games don't use all necessary dlls. With this option, you can strip out unused parts to reduce the size of the built player on iOS devices. If your game is using classes that would normally be stripped out by the option you currently have selected, you'll be presented with a Debug message when you make a build.

**Script Call Optimization**
A good development practice on iOS is to never rely on exception handling (either internally or through the use of try/catch blocks). When using the default **Slow and Safe** option, any exceptions that occur on the device will be caught and a stack trace will be provided. When using the **Fast but no Exceptions** option, any exceptions that occur will crash the game, and no stack trace will be provided. However, the game will run faster since the processor is not diverting power to handle exceptions. When releasing your game to the world, it's best to publish with the **Fast but no Exceptions** option.

## Android

**Bundle Identifier**
The **Bundle Identifier** string is the unique name of your application when published to the Android Market and installed on the device. The basic structure of the identifier is **com.CompanyName.GameName**, and can be chosen arbitrarily. In Unity this field is shared with the iOS Player Settings for convenience.

**Stripping Level (Pro-only)**
Most games don't use all the functionality of the provided dlls. With this option, you can strip out unused parts to reduce the size of the built player on Android devices.

Page last updated: 2012-11-16

---

# android-API

Unity Android provides a number of scripting APIs unified with iOS APIs to access handheld device functionality. For cross-platform projects, UNITY_ANDROID is defined for conditionally compiling Android-specific C# code. The following scripting classes contain Android-related changes (some of the API is shared between Android and iOS):

| | |
|---|---|
| Input | Access to multi-touch screen, accelerometer and device orientation. |
| iPhoneSettings | Some of the Android settings, such as screen orientation, dimming and information about device hardware. |
| iPhoneKeyboard | Support for native on-screen keyboard. |
| iPhoneUtils | Useful functions for movie playback, anti-piracy protection and vibration. |

## Further Reading

- Input
- Mobile Keyboard
- Advanced Unity Mobile Scripting
- Using .NET API 2.0 compatibility level

Page last updated: 2010-09-08

---

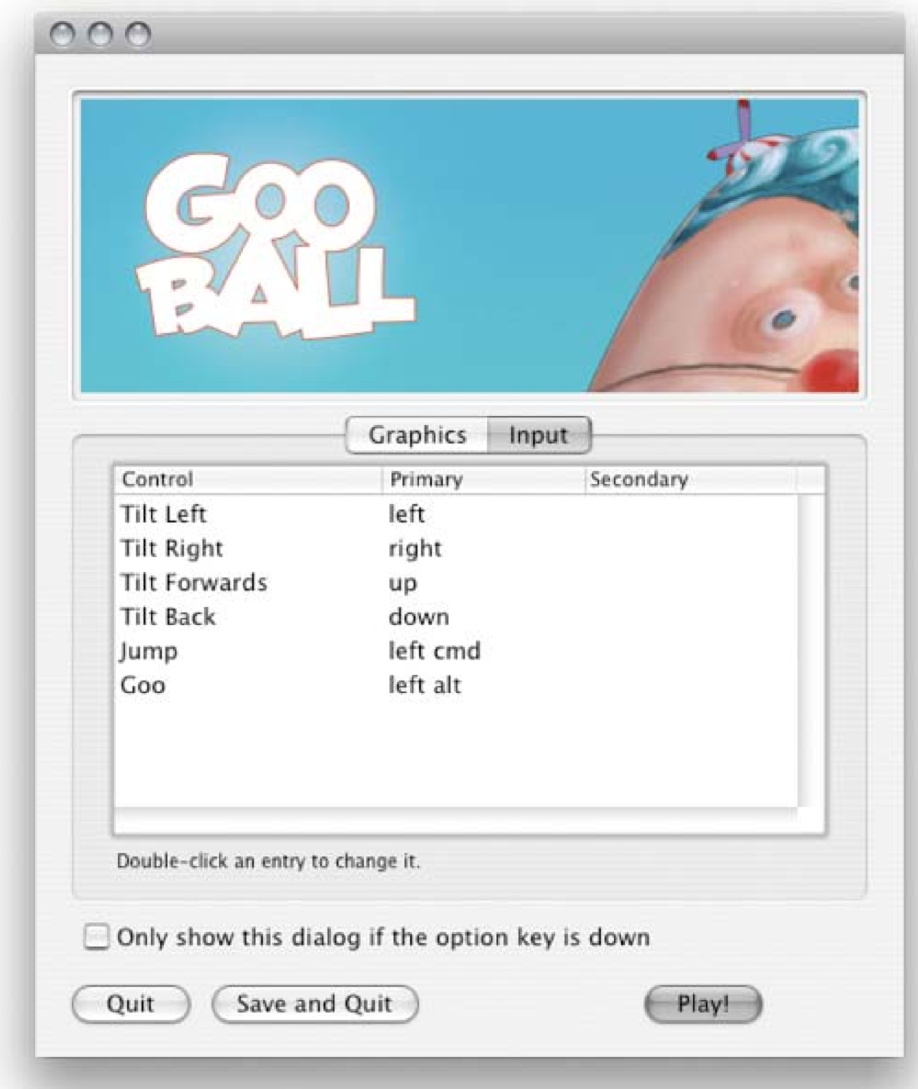
# Android-Input

## Desktop

**Note:** Keyboard, joystick and gamepad input work on the desktop versions of Unity (including webplayer and Flash) but not on mobiles.

Unity supports keyboard, joystick and gamepad input.

Virtual axes and buttons can be created in the **Input Manager**, and end users can configure Keyboard input in a nice screen configuration dialog.




You can setup joysticks, gamepads, keyboard, and mouse, then access them all through one simple scripting interface.

From scripts, all virtual axes are accessed by their name.

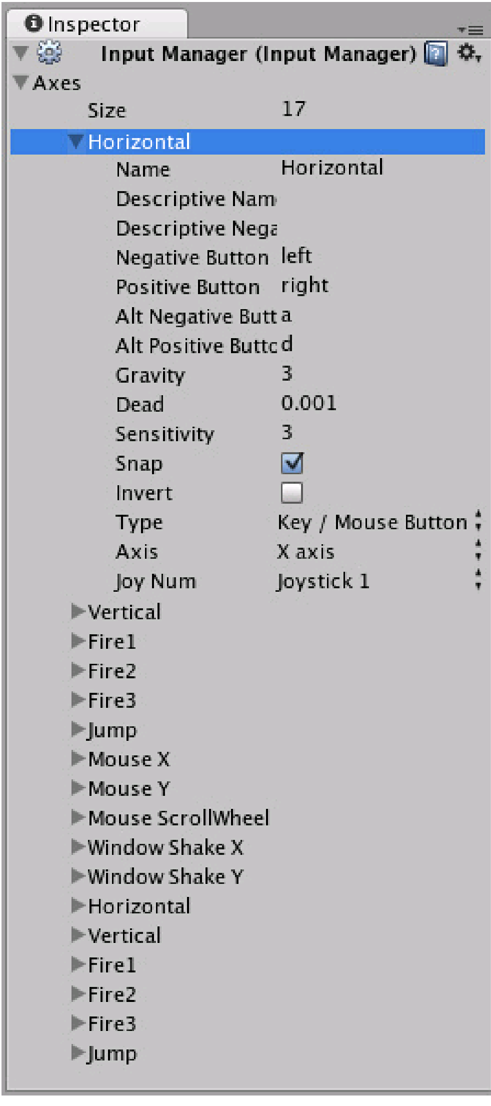
Every project has the following default input axes when it's created:

- **Horizontal** and **Vertical** are mapped to w, a, s, d and the arrow keys.
- **Fire1**, **Fire2**, **Fire3** are mapped to Control, Option (Alt), and Command, respectively.

- **Mouse X** and **Mouse Y** are mapped to the delta of mouse movement.
- **Window Shake X** and **Window Shake Y** is mapped to the movement of the window.

### Adding new Input Axes

If you want to add new virtual axes go to the **Edit->Project Settings->Input** menu. Here you can also change the settings of each axis.




You map each axis to two buttons on a joystick, mouse, or keyboard keys.

| | |
|---|---|
| **Name** | The name of the string used to check this axis from a script. |
| **Descriptive Name** | Positive value name displayed in the input tab of the **Configuration** dialog for standalone builds. |
| **Descriptive Negative Name** | Negative value name displayed in the Input tab of the **Configuration** dialog for standalone builds. |
| **Negative Button** | The button used to push the axis in the negative direction. |
| **Positive Button** | The button used to push the axis in the positive direction. |
| **Alt Negative Button** | Alternative button used to push the axis in the negative direction. |
| **Alt Positive Button** | Alternative button used to push the axis in the positive direction. |
| **Gravity** | Speed in units per second that the axis falls toward neutral when no buttons are pressed. |
| **Dead** | Size of the analog dead zone. All analog device values within this range result map to neutral. |
| **Sensitivity** | Speed in units per second that the the axis will move toward the target value. This is for digital devices only. |
| **Snap** | If enabled, the axis value will reset to zero when pressing a button of the opposite direction. |
| **Invert** | If enabled, the **Negative Buttons** provide a positive value, and vice-versa. |
| **Type** | The type of inputs that will control this axis. |
| **Axis** | The axis of a connected device that will control this axis. |
| **Joy Num** | The connected Joystick that will control this axis. |

Use these settings to fine tune the look and feel of input. They are all documented with tooltips in the Editor as well.

**Using Input Axes from Scripts**
You can query the current state from a script like this:

```
value = Input.GetAxis ("Horizontal");
```

An axis has a value between -1 and 1. The neutral position is 0. This is the case for joystick input and keyboard input.

However, Mouse Delta and Window Shake Delta are how much the mouse or window moved during the last frame. This means it can be larger than 1 or smaller than -1 when the user moves the mouse quickly.

It is possible to create multiple axes with the same name. When getting the input axis, the axis with the largest absolute value will be returned. This makes it possible to assign more than one input device to one axis name. For example, create one axis for keyboard input and one axis for joystick input with the same name. If the user is using the joystick, input will come from the joystick, otherwise input will come from the keyboard. This way you don't have to consider where the input comes from when writing scripts.

**Button Names**
To map a key to an axis, you have to enter the key's name in the **Positive Button** or **Negative Button** property in the **Inspector**.

The names of keys follow this convention:

- Normal keys: "a", "b", "c" ...
- Number keys: "1", "2", "3", ...
- Arrow keys: "up", "down", "left", "right"
- Keypad keys: "[1]", "[2]", "[3]", "[+]", "[equals]"
- Modifier keys: "right shift", "left shift", "right ctrl", "left ctrl", "right alt", "left alt", "right cmd", "left cmd"
- Mouse Buttons: "mouse 0", "mouse 1", "mouse 2", ...
- Joystick Buttons (from any joystick): "joystick button 0", "joystick button 1", "joystick button 2", ...
- Joystick Buttons (from a specific joystick): "joystick 1 button 0", "joystick 1 button 1", "joystick 2 button 0", ...
- Special keys: "backspace", "tab", "return", "escape", "space", "delete", "enter", "insert", "home", "end", "page up", "page down"
- Function keys: "f1", "f2", "f3", ...

The names used to identify the keys are the same in the scripting interface and the Inspector.

```
value = Input.GetKey ("a");
```

# Mobile Input

On iOS and Android, the Input class offers access to touchscreen, accelerometer and geographical/location input.

Access to keyboard on mobile devices is provided via the iOS keyboard.

## Multi-Touch Screen
The iPhone and iPod Touch devices are capable of tracking up to five fingers touching the screen simultaneously. You can retrieve the status of each finger touching the screen during the last frame by accessing the Input.touches property array.

Android devices don't have a unified limit on how many fingers they track. Instead, it varies from device to device and can be anything from two-touch on older devices to five fingers on some newer devices.

Each finger touch is represented by an Input.Touch data structure:

| | |
|---|---|
| **fingerId** | The unique index for a touch. |
| **position** | The screen position of the touch. |
| **deltaPosition** | The screen position change since the last frame. |
| **deltaTime** | Amount of time that has passed since the last state change. |

| | |
|---|---|
| **tapCount** | The iPhone/iPad screen is able to distinguish quick finger taps by the user. This counter will let you know how many times the user has tapped the screen without moving a finger to the sides. Android devices do not count number of taps, this field is always 1. |
| **phase** | Describes so called "phase" or the state of the touch. It can help you determine if the touch just began, if user moved the finger or if he just lifted the finger. |

Phase can be one of the following:

| | |
|---|---|
| **Began** | A finger just touched the screen. |
| **Moved** | A finger moved on the screen. |
| **Stationary** | A finger is touching the screen but hasn't moved since the last frame. |
| **Ended** | A finger was lifted from the screen. This is the final phase of a touch. |
| **Canceled** | The system cancelled tracking for the touch, as when (for example) the user puts the device to her face or more than five touches happened simultaneously. This is the final phase of a touch. |

Following is an example script which will shoot a ray whenever the user taps on the screen:

```
var particle : GameObject;
function Update () {
	for (var touch : Touch in Input.touches) {
		if (touch.phase == TouchPhase.Began) {
			// Construct a ray from the current touch coordinates
			var ray = Camera.main.ScreenPointToRay (touch.position);
			if (Physics.Raycast (ray)) {
				// Create a particle if hit
				Instantiate (particle, transform.position, transform.rotation);
			}
		}
	}
}
```

**Mouse Simulation**

On top of native touch support Unity iOS/Android provides a mouse simulation. You can use mouse functionality from the standard Input class.

## Device Orientation

Unity iOS/Android allows you to get discrete description of the device physical orientation in three-dimensional space. Detecting a change in orientation can be useful if you want to create game behaviors depending on how the user is holding the device.

You can retrieve device orientation by accessing the Input.deviceOrientation property. Orientation can be one of the following:

| | |
|---|---|
| **Unknown** | The orientation of the device cannot be determined. For example when device is rotate diagonally. |
| **Portrait** | The device is in portrait mode, with the device held upright and the home button at the bottom. |
| **PortraitUpsideDown** | The device is in portrait mode but upside down, with the device held upright and the home button at the top. |
| **LandscapeLeft** | The device is in landscape mode, with the device held upright and the home button on the right side. |
| **LandscapeRight** | The device is in landscape mode, with the device held upright and the home button on the left side. |
| **FaceUp** | The device is held parallel to the ground with the screen facing upwards. |
| **FaceDown** | The device is held parallel to the ground with the screen facing downwards. |

## Accelerometer

As the mobile device moves, a built-in accelerometer reports linear acceleration changes along the three primary axes in three-dimensional space. Acceleration along each axis is reported directly by the hardware as G-force values. A value of 1.0 represents a load of about +1g along a given axis while a value of -1.0 represents -1g. If you hold the device upright (with the home button at the bottom) in front of you, the X axis is positive along the right, the Y axis is positive directly up, and the Z axis is positive pointing toward you.

You can retrieve the accelerometer value by accessing the Input.acceleration property.

The following is an example script which will move an object using the accelerometer:

```
var speed = 10.0;
function Update () {
	var dir : Vector3 = Vector3.zero;

	// we assume that the device is held parallel to the ground
	// and the Home button is in the right hand

	// remap the device acceleration axis to game coordinates:
	//  1) XY plane of the device is mapped onto XZ plane
	//  2) rotated 90 degrees around Y axis
	dir.x = -Input.acceleration.y;
	dir.z = Input.acceleration.x;

	// clamp acceleration vector to the unit sphere
	if (dir.sqrMagnitude > 1)
		dir.Normalize();

	// Make it move 10 meters per second instead of 10 meters per frame...
	dir *= Time.deltaTime;

	// Move object
	transform.Translate (dir * speed);
}
```

**Low-Pass Filter**
Accelerometer readings can be jerky and noisy. Applying low-pass filtering on the signal allows you to smooth it and get rid of high frequency noise.

The following script shows you how to apply low-pass filtering to accelerometer readings:

```
var AccelerometerUpdateInterval : float = 1.0 / 60.0;
var LowPassKernelWidthInSeconds : float = 1.0;

private var LowPassFilterFactor : float = AccelerometerUpdateInterval / LowPassKernelWidthInSeconds; // tweakable
private var lowPassValue : Vector3 = Vector3.zero;
function Start () {
	lowPassValue = Input.acceleration;
}

function LowPassFilterAccelerometer() : Vector3 {
	lowPassValue = Mathf.Lerp(lowPassValue, Input.acceleration, LowPassFilterFactor);
	return lowPassValue;
}
```

The greater the value of `LowPassKernelWidthInSeconds`, the slower the filtered value will converge towards the current input sample (and vice versa). You should be able to use the `LowPassFilter()` function instead of `avgSamples()`.

**I'd like as much precision as possible when reading the accelerometer. What should I do?**
Reading the Input.acceleration variable does not equal sampling the hardware. Put simply, Unity samples the hardware at a frequency of 60Hz and stores the result into the variable. In reality, things are a little bit more complicated -- accelerometer sampling doesn't occur at consistent time intervals, if under significant CPU loads. As a result, the system might report 2 samples during one frame, then 1 sample during the next frame.

You can access all measurements executed by accelerometer during the frame. The following code will illustrate a simple average of all the accelerometer events that were collected within the last frame:

```
var period : float = 0.0;
var acc : Vector3 = Vector3.zero;
```

```
for (var evnt : iPhoneAccelerationEvent  in iPhoneInput.accelerationEvents) {
      acc += evnt.acceleration * evnt.deltaTime;
      period += evnt.deltaTime;
}
if (period > 0)
      acc *= 1.0/period;
return acc;
```

### Further Reading

The Unity mobile input API is originally based on Apple's API. It may help to learn more about the native API to better understand Unity's Input API. You can find the Apple input API documentation here:

- Programming Guide: Event Handling (Apple iPhone SDK documentation)
- UITouch Class Reference (Apple iOS SDK documentation)

**Note:** The above links reference your locally installed iPhone SDK Reference Documentation and will contain native ObjectiveC code. It is not necessary to understand these documents for using Unity on mobile devices, but may be helpful to some!

## iOS

### Device geographical location

Device geographical location can be obtained via the iPhoneInput.lastLocation property. Before calling this property you should start location service updates using iPhoneSettings.StartLocationServiceUpdates() and check the service status via iPhoneSettings.locationServiceStatus. See the scripting reference for details.

Page last updated: 2012-06-28

---

# Android-Keyboard

In most cases, Unity will handle keyboard input automatically for GUI elements but it is also easy to show the keyboard on demand from a script.

## iOS

### Using the Keyboard

#### GUI Elements

The keyboard will appear automatically when a user taps on editable GUI elements. Currently, GUI.TextField, GUI.TextArea and GUI.PasswordField will display the keyboard; see the GUI class documentation for further details.

#### Manual Keyboard Handling

Use the **iPhoneKeyboard.Open** function to open the keyboard. Please see the iPhoneKeyboard scripting reference for the parameters that this function takes.

### Keyboard Type Summary

The Keyboard supports the following types:

| | |
|---|---|
| **iPhoneKeyboardType.Default** | Letters. Can be switched to keyboard with numbers and punctuation. |
| **iPhoneKeyboardType.ASCIICapable** | Letters. Can be switched to keyboard with numbers and punctuation. |
| **iPhoneKeyboardType.NumbersAndPunctuation** | Numbers and punctuation. Can be switched to keyboard with letters. |
| **iPhoneKeyboardType.URL** | Letters with slash and .com buttons. Can be switched to keyboard with numbers and punctuation. |
| **iPhoneKeyboardType.NumberPad** | Only numbers from 0 to 9. |
| **iPhoneKeyboardType.PhonePad** | Keyboard used to enter phone numbers. |

| | |
|---|---|
| **iPhoneKeyboardType.NamePhonePad** | Letters. Can be switched to phone keyboard. |
| **iPhoneKeyboardType.EmailAddress** | Letters with @ sign. Can be switched to keyboard with numbers and punctuation. |

### Text Preview

By default, an edit box will be created and placed on top of the keyboard after it appears. This works as preview of the text that user is typing, so the text is always visible for the user. However, you can disable text preview by setting **iPhoneKeyboard.hideInput** to true. Note that this works only for certain keyboard types and input modes. For example, it will not work for phone keypads and multi-line text input. In such cases, the edit box will always appear. **iPhoneKeyboard.hideInput** is a global variable and will affect all keyboards.

### Keyboard Orientation

By default, the keyboard automatically follows the device orientation. To disable or enable rotation to a certain orientation, use the following properties available in iPhoneKeyboard:

| | |
|---|---|
| **autorotateToPortrait** | Enable or disable autorotation to portrait orientation (button at the bottom). |
| **autorotateToPortraitUpsideDown** | Enable or disable autorotation to portrait orientation (button at top). |
| **autorotateToLandscapeLeft** | Enable or disable autorotation to landscape left orientation (button on the right). |
| **autorotateToLandscapeRight** | Enable or disable autorotation to landscape right orientation (button on the left). |

### Visibility and Keyboard Size

There are three keyboard properties in iPhoneKeyboard that determine keyboard visibility status and size on the screen.

| | |
|---|---|
| **visible** | Returns **true** if the keyboard is fully visible on the screen and can be used to enter characters. |
| **area** | Returns the position and dimensions of the keyboard. |
| **active** | Returns **true** if the keyboard is activated. This property is not static property. You must have a keyboard instance to use this property. |

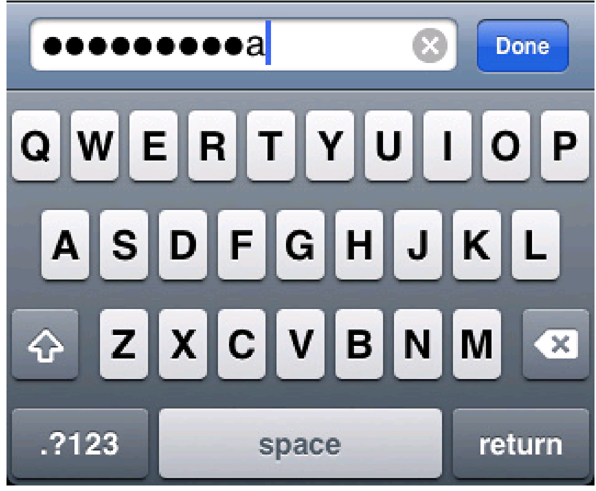
Note that **iPhoneKeyboard.area** will return a rect with position and size set to 0 until the keyboard is fully visible on the screen. You should not query this value immediately after **iPhoneKeyboard.Open**. The sequence of keyboard events is as follows:

- **iPhoneKeyboard.Open** is called. **iPhoneKeyboard.active** returns true. **iPhoneKeyboard.visible** returns false. **iPhoneKeyboard.area** returns (0, 0, 0, 0).
- Keyboard slides out into the screen. All properties remain the same.
- Keyboard stops sliding. **iPhoneKeyboard.active** returns true. **iPhoneKeyboard.visible** returns true. **iPhoneKeyboard.area** returns real position and size of the keyboard.

### Secure Text Input

It is possible to configure the keyboard to hide symbols when typing. This is useful when users are required to enter sensitive information (such as passwords). To manually open keyboard with secure text input enabled, use the following code:

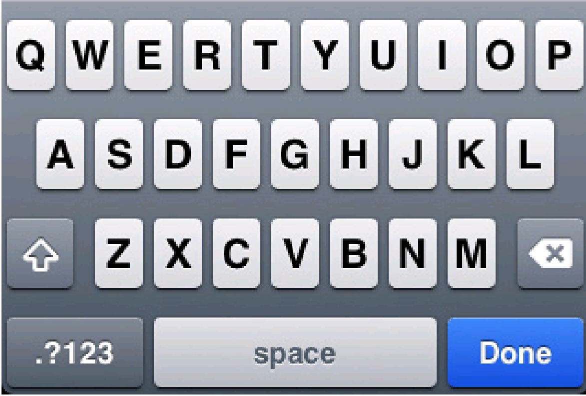
```
iPhoneKeyboard.Open("", iPhoneKeyboardType.Default, false, false, true);
```



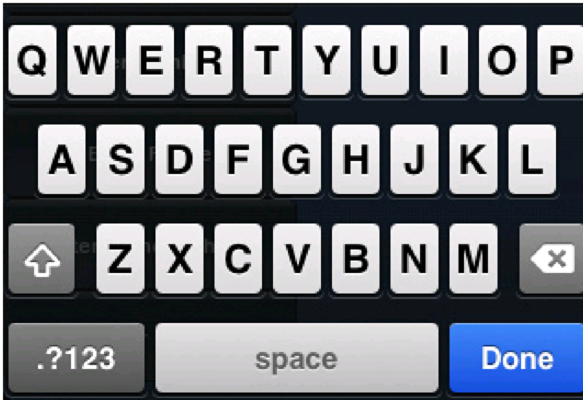

*Hiding text while typing*

**Alert keyboard**
To display the keyboard with a black semi-transparent background instead of the classic opaque, call **iPhoneKeyboard.Open** as follows:

```
iPhoneKeyboard.Open("", iPhoneKeyboardType.Default, false, false, true, true);
```



*Classic keyboard*



*Alert keyboard*

## Android

Unity Android reuses the iOS API to display system keyboard. Even though Unity Android supports most of the functionality of its iPhone counterpart, there are two aspects which are not supported:

- **iPhoneKeyboard.hideInput**
- **iPhoneKeyboard.area**

Please also note that the layout of a **iPhoneKeyboardType** can differ somewhat between devices.

Page last updated: 2011-11-02

---

# Android-Advanced

## iOS

# Advanced iOS scripting

### Determining Device Generation

Different device generations support different functionality and have widely varying performance. You should query the device's generation and decide which functionality should be disabled to compensate for slower devices.

You can find the device generation from the iPhone.generation property. The reported generation can be one of the following:

- **iPhone**
- **iPhone3G**
- **iPhone3GS**
- **iPhone4**
- **iPodTouch1Gen**
- **iPodTouch2Gen**
- **iPodTouch3Gen**
- **iPodTouch4Gen**
- **iPad1Gen**

You can find more information about different device generations, performance and supported functionality in our iPhone Hardware Guide.

### Device Properties

There are a number of device-specific properties that you can access:-

| | |
|---|---|
| **SystemInfo.deviceUniqueIdentifier** | Unique device identifier. |
| **SystemInfo.deviceName** | User specified name for device. |
| **SystemInfo.deviceModel** | Is it iPhone or iPod Touch? |
| **SystemInfo.operatingSystem** | Operating system name and version. |

### Anti-Piracy Check

Pirates will often hack an application from the AppStore (by removing Apple DRM protection) and then redistribute it for free. Unity iOS comes with an anti-piracy check which allows you to determine if your application was altered **after** it was submitted to the AppStore.

You can check if your application is genuine (not-hacked) with the Application.genuine property. If this property returns **false** then you might notify the user that he is using a hacked application or maybe disable access to some functions of your application.

**Note:** accessing the Application.genuine property is a fairly expensive operation and so you shouldn't do it during frame updates or other time-critical code.

### Vibration Support

You can trigger a vibration by calling Handheld.Vibrate. Note that iPod Touch devices lack vibration hardware and will just ignore this call.

## Android

# Advanced Android scripting

### Determining Device Generation

Different Android devices support different functionality and have widely varying performance. You should target specific devices or device families and decide which functionality should be disabled to compensate for slower devices. There are a number of device specific properties that you can access to which device is being used.

**Note:** Android Marketplace does some additional compatibility filtering, so you should not be concerned if an ARMv7-only app optimised for OGLES2 is offered to some old slow devices.

### Device Properties

| | |
|---|---|
| **SystemInfo.deviceUniqueIdentifier** | Unique device identifier. |
| **SystemInfo.deviceName** | User specified name for device. |
| **SystemInfo.deviceModel** | Is it iPhone or iPod Touch? |

| | |
|---|---|
| SystemInfo.operatingSystem | Operating system name and version. |

### Anti-Piracy Check

Pirates will often hack an application (by removing Apple DRM protection) and then redistribute it for free. Unity Android comes with an anti-piracy check which allows you to determine if your application was altered **after** it was submitted to the AppStore.

You can check if your application is genuine (not-hacked) with the Application.genuine property. If this property returns **false** then you might notify user that he is using a hacked application or maybe disable access to some functions of your application.

**Note:** Application.genuineCheckAvailable should be used along with **Application.genuine** to verify that application integrity can actually be confirmed. Accessing the Application.genuine property is a fairly expensive operation and so you shouldn't do it during frame updates or other time-critical code.

### Vibration Support

You can trigger a vibration by calling Handheld.Vibrate. However, devices lacking vibration hardware will just ignore this call.

Page last updated: 2012-07-11

---

# Android-DotNet

## iOS

Now Unity iOS supports two .NET API compatibility levels: .NET 2.0 and a subset of .NET 2.0 .You can select the appropriate level in the Player Settings.

### .NET API 2.0

Unity supports the **.NET 2.0** API profile. This is close to the full .NET 2.0 API and offers the best compatibility with pre-existing .NET code. However, the application's build size and startup time will be relatively poor.

**Note:** Unity iOS does not support namespaces in scripts. If you have a third party library supplied as source code then the best approach is to compile it to a DLL outside Unity and then drop the DLL file into your project's Assets folder.

### .NET 2.0 Subset

Unity also supports the **.NET 2.0 Subset** API profile. This is close to the Mono "monotouch" profile, so many limitations of the "monotouch" profile also apply to Unity's .NET 2.0 Subset profile. More information on the limitations of the "monotouch" profile can be found here. The advantage of using this profile is reduced build size (and startup time) but this comes at the expense of compatibility with existing .NET code.

## Android

Unity Android supports two .NET API compatibility levels: .NET 2.0 and a subset of .NET 2.0 You can select the appropriate level in the Player Settings.

### .NET API 2.0

Unity supports the **.NET 2.0** API profile; It is close to the full .NET 2.0 API and offers the best compatibility with pre-existing .NET code. However, the application's build size and startup time will be relatively poor.

**Note:** Unity Android does not support namespaces in scripts. If you have a third party library supplied as source code then the best approach is to compile it to a DLL outside Unity and then drop the DLL file into your project's Assets folder.

### .NET 2.0 Subset

Unity also supports the **.NET 2.0 Subset** API profile. This is close to the Mono "monotouch" profile, so many limitations of the "monotouch" profile also apply to Unity's .NET 2.0 Subset profile. More information on the limitations of the "monotouch" profile can be found here. The advantage of using this profile is reduced build size (and startup time) but this comes at the expense of compatibility with existing .NET code.

Page last updated: 2012-07-11

---

# Android-Plugins

This page describes Native Code Plugins for Android.

## Building a Plugin for Android

To build a plugin for Android, you should first obtain the Android NDK and familiarize yourself with the steps involved in building a shared library.

If you are using C++ (.cpp) to implement the plugin you must ensure the functions are declared with C linkage to avoid name mangling issues.

```
extern "C" {
  float FooPluginFunction ();
}
```

## Using Your Plugin from C#

Once built, the shared library should be copied to the **Assets->Plugins->Android** folder. Unity will then find it by name when you define a function like the following in the C# script:-

```
[DllImport ("PluginName")]
private static extern float FooPluginFunction ();
```

Please note that **PluginName** should not include the prefix ('lib') nor the extension ('.so') of the filename. It is advisable to wrap all native code methods with an additional C# code layer. This code should check Application.platform and call native methods only when the app is running on the actual device; dummy values can be returned from the C# code when running in the Editor. You can also use platform defines to control platform dependent code compilation.

## Deployment

For cross platform deployment, your project should include plugins for each supported platform (ie, libPlugin.so for Android, Plugin.bundle for Mac and Plugin.dll for Windows). Unity automatically picks the right plugin for the target platform and includes it with the player.

## Using Java Plugins

The Android plugin mechanism also allows Java to be used to enable interaction with the Android OS.

### Building a Java Plugin for Android

There are several ways to create a Java plugin but the result in each case is that you end up with a .jar file containing the .class files for your plugin. One approach is to download the JDK, then compile your .java files from the command line with *javac*. This will create .class files which you can then package into a .jar with the *jar* command line tool. Another option is to use the Eclipse IDE together with the ADT.

### Using Your Java Plugin from Native Code

Once you have built your Java plugin (.jar) you should copy it to the **Assets->Plugins->Android** folder in the Unity project. Unity will package your .class files together with the rest of the Java code and then access the code using the Java Native Interface (JNI). JNI is used both when calling native code from Java and when interacting with Java (or the JavaVM) from native code.

To find your Java code from the native side you need access to the Java VM. Fortunately, that access can be obtained easily by adding a function like this to your C/C++ code:

```
jint JNI_OnLoad(JavaVM* vm, void* reserved) {
  JNIEnv* jni_env = 0;
  vm->AttachCurrentThread(&jni_env, 0);
```

```
}
```

This is all that is needed to start using Java from C/C++. It is beyond the scope of this document to explain JNI completely. However, using it usually involves finding the class definition, resolving the constructor (<init>) method and creating a new object instance, as shown in this example:-

```
jobject createJavaObject(JNIEnv* jni_env) {
  jclass cls_JavaClass = jni_env->FindClass("com/your/java/Class");                // find class definition
  jmethodID mid_JavaClass = jni_env->GetMethodID (cls_JavaClass, "<init>", "()V");         // find constructor method
  jobject obj_JavaClass = jni_env->NewObject(cls_JavaClass, mid_JavaClass);           // create object instance
  return jni_env->NewGlobalRef(obj_JavaClass);                                    // return object with a global reference
}
```

**Using Your Java Plugin with helper classes**
**AndroidJNIHelper** and **AndroidJNI** can be used to ease some of the pain with raw JNI.

**AndroidJavaObject** and **AndroidJavaClass** automate a lot of tasks and also use cacheing to make calls to Java faster. The combination of **AndroidJavaObject** and **AndroidJavaClass** builds on top of **AndroidJNI** and **AndroidJNIHelper**, but also has a lot of logic in its own right (to handle the automation). These classes also come in a 'static' version to access static members of Java classes.

You can choose whichever approach you prefer, be it raw JNI through **AndroidJNI** class methods, or **AndroidJNIHelper** together with **AndroidJNI** and eventually **AndroidJavaObject/AndroidJavaClass** for maximum automation and convenience.

UnityEngine.AndroidJNI is a wrapper for the JNI calls available in C (as described above). All methods in this class are static and have a 1:1 mapping to the Java Native Interface. UnityEngine.AndroidJNIHelper provides helper functionality used by the next level, but is exposed as public methods because they may be useful for some special cases.

Instances of UnityEngine.AndroidJavaObject and UnityEngine.AndroidJavaClass have a 1:1 mapping to an instance of java.lang.Object and java.lang.Class (or subclasses thereof) on the Java side, respectively. They essentially provide 3 types of interaction with the Java side:

- Call a method
- Get the value of a field
- Set the value of a field

The **Call** is separated into two categories: **Call** to a 'void' method, and **Call** to a method with non-void return type. A generic type is used to represent the return type of those methods which return a non-void type. The **Get** and **Set** always take a generic type representing the field type.

**Example 1**

```
//The comments describe what you would need to do if you were using raw JNI
 AndroidJavaObject jo = new AndroidJavaObject("java.lang.String", "some_string");
 // jni.FindClass("java.lang.String");
 // jni.GetMethodID(classID, "<init>", "(Ljava/lang/String;)V");
 // jni.NewStringUTF("some_string");
 // jni.NewObject(classID, methodID, javaString);
 int hash = jo.Call<int>("hashCode");
 // jni.GetMethodID(classID, "hashCode", "()I");
 // jni.CallIntMethod(objectID, methodID);
```

Here, we're creating an instance of java.lang.String, initialized with a string of our choice and retrieving the hash value for that string.

The **AndroidJavaObject** constructor takes at least one parameter, the name of class for which we want to construct an instance. Any parameters after the class name are for the constructor call on the object, in this case the string "some_string". The subsequent **Call** to hashCode() returns an 'int' which is why we use that as the generic type parameter to the **Call** method.

**Note:** You cannot instantiate a nested Java class using dotted notation. Inner classes must use the $ separator, and it should work in both dotted and slashed format. So **android.view.ViewGroup$LayoutParams** or **android/view/ViewGroup$LayoutParams** can be used, where a **LayoutParams** class is nested in a **ViewGroup** class.

**Example 2**

One of the plugin samples above shows how to get the cache directory for the current application. This is how you would do the same thing from C# without any plugins:-

```
AndroidJavaClass jc = new AndroidJavaClass("com.unity3d.player.UnityPlayer");
// jni.FindClass("com.unity3d.player.UnityPlayer");
AndroidJavaObject jo = jc.GetStatic<AndroidJavaObject>("currentActivity");
// jni.GetStaticFieldID(classID, "Ljava/lang/Object;");
// jni.GetStaticObjectField(classID, fieldID);
// jni.FindClass("java.lang.Object");

Debug.Log(jo.Call<AndroidJavaObject>("getCacheDir").Call<string>("getCanonicalPath"));
// jni.GetMethodID(classID, "getCacheDir", "()Ljava/io/File;"); // or any baseclass thereof!
// jni.CallObjectMethod(objectID, methodID);
// jni.FindClass("java.io.File");
// jni.GetMethodID(classID, "getCanonicalPath", "()Ljava/lang/String;");
// jni.CallObjectMethod(objectID, methodID);
// jni.GetStringUTFChars(javaString);
```

In this case, we start with **AndroidJavaClass** instead of **AndroidJavaObject** because we want to access a static member of **com.unity3d.player.UnityPlayer** rather than create a new object (an instance is created automatically by the **Android UnityPlayer**). Then we access the static field "currentActivity" but this time we use **AndroidJavaObject** as the generic parameter. This is because the actual field type (android.app.Activity) is a subclass of java.lang.Object, and any non-primitive type must be accessed as **AndroidJavaObject**. The exceptions to this rule are strings, which can be accessed directly even though they don't represent a primitive type in Java.

After that it is just a matter of traversing the **Activity** through getCacheDir() to get the File object representing the cache directory, and then calling getCanonicalPath() to get a string representation.

Of course, nowadays you don't need to do that to get the cache directory since Unity provides access to the application's cache and file directory with Application.temporaryCachePath and Application.persistentDataPath.

**Example 3**

Finally, here is a trick for passing data from Java to script code using **UnitySendMessage**.

```
using UnityEngine;
public class NewBehaviourScript : MonoBehaviour {

	void Start () {
		JNIHelper.debug = true;
		using (JavaClass jc = new JavaClass("com.unity3d.player.UnityPlayer")) {
			jc.CallStatic("UnitySendMessage", "Main Camera", "JavaMessage", "whoowhoo");
		}
	}

	void JavaMessage(string message) {
		Debug.Log("message from java: " + message);
	}
}
```

The Java class **com.unity3d.player.UnityPlayer** now has a static method **UnitySendMessage**, equivalent to the iOS UnitySendMessage on the native side. It can be used in Java to pass data to script code.

Here though, we call it directly from script code, which essentially relays the message on the Java side. This then calls back to the native/Unity code to deliver the message to the object named "Main Camera". This object has a script attached which

contains a method called "JavaMessage".

**Best practice when using Java plugins with Unity**
As this section is mainly aimed at people who don't have comprehensive JNI, Java and Android experience, we assume that the **AndroidJavaObject/AndroidJavaClass** approach has been used for interacting with Java code from Unity.

The first thing to note is that any operation you perform on an **AndroidJavaObject** or **AndroidJavaClass** is computationally expensive (as is the raw JNI approach). It is highly advisable to keep the number of transitions between managed and native/Java code to a minimum, for the sake of performance and also code clarity.

You could have a Java method to do all the actual work and then use **AndroidJavaObject / AndroidJavaClass** to communicate with that method and get the result. However, it is worth bearing in mind that the JNI helper classes try to cache as much data as possible to improve performance.

```
//The first time you call a Java function like
AndroidJavaObject jo = new AndroidJavaObject("java.lang.String", "some_string");  // somewhat expensive
int hash = jo.Call<int>("hashCode");  // first time - expensive
int hash = jo.Call<int>("hashCode");  // second time - not as expensive as we already know the java method and can call it 
```

The Mono garbage collector should release all created instances of **AndroidJavaObject** and **AndroidJavaClass** after use, but it is advisable to keep them in a **using(){}** statement to ensure they are deleted as soon as possible. Without this, you cannot be sure when they will be destroyed. If you set **AndroidJNIHelper.debug** to true, you will see a record of the garbage collector's activity in the debug output.

```
//Getting the system language with the safe approach
void Start () {
	using (AndroidJavaClass cls = new AndroidJavaClass("java.util.Locale")) {
		using(AndroidJavaObject locale = cls.CallStatic<AndroidJavaObject>("getDefault")) {
			Debug.Log("current lang = " + locale.Call<string>("getDisplayLanguage"));

		}
	}
}
```

You can also call the **.Dispose()** method directly to ensure there are no Java objects lingering. The actual C# object might live a bit longer, but will be garbage collected by mono eventually.

## Extending the UnityPlayerActivity Java Code

With Unity Android it is possible to extend the standard **UnityPlayerActivity** class (the primary Java class for the Unity Player on Android, similar to AppController.mm on Unity iOS).

An application can override any and all of the basic interaction between Android OS and Unity Android. You can enable this by creating a new Activity which derives from UnityPlayerActivity (UnityPlayerActivity.java can be found at **/Applications/Unity /Unity.app/Contents/PlaybackEngines/AndroidPlayer/src/com/unity3d/player** on Mac and usually at **C:\Program Files\Unity\Editor\Data\PlaybackEngines\AndroidPlayer\src\com\unity3d\player** on Windows).

To do this, first locate the **classes.jar** shipped with Unity Android. It is found in the installation folder (usually **C:\Program Files\Unity\Editor\Data** (on Windows) or **/Applications/Unity** (on Mac)) in a sub-folder called **PlaybackEngines/AndroidPlayer/bin**. Then add **classes.jar** to the classpath used to compile the new Activity. The resulting .class file(s) should be compressed into a .jar file and placed in the **Assets->Plugins->Android** folder. Since the manifest dictates which activity to launch it is also necessary to create a new AndroidManifest.xml. The AndroidManifest.xml file should also be placed in the **Assets->Plugins->Android** folder.

The new activity could look like the following example, **OverrideExample.java**:

```
package com.company.product;

import com.unity3d.player.UnityPlayerActivity;
```

```
import android.os.Bundle;
import android.util.Log;

public class OverrideExample extends UnityPlayerActivity {

  protected void onCreate(Bundle savedInstanceState) {

    // call UnityPlayerActivity.onCreate()
    super.onCreate(savedInstanceState);

    // print debug message to logcat
    Log.d("OverrideActivity", "onCreate called!");
  }

  public void onBackPressed()
  {
    // instead of calling UnityPlayerActivity.onBackPressed() we just ignore the back button event
    // super.onBackPressed();
  }
}
```

And this is what the corresponding **AndroidManifest.xml** would look like:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.company.product">
  <application android:icon="@drawable/app_icon" android:label="@string/app_name">
        <activity android:name=".OverrideExample"
                  android:label="@string/app_name"
                  android:configChanges="fontScale|keyboard|keyboardHidden|locale|mnc|mcc|navigation|orientation|screen
        <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
  </application>
</manifest>
```

**UnityPlayerNativeActivity**

It is also possible to create your own subclass of `UnityPlayerNativeActivity`. This will have much the same effect as subclassing UnityPlayerActivity but with improved input latency. Be aware, though, that NativeActivity was introduced in Gingerbread and does not work with older devices. Since touch/motion events are processed in native code, Java views would normally not see those events. There is, however, a forwarding mechanism in Unity which allows events to be propagated to the DalvikVM. To access this mechanism, you need to modify the manifest file as follows:-

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.company.product">
  <application android:icon="@drawable/app_icon" android:label="@string/app_name">
        <activity android:name=".OverrideExampleNative"
                  android:label="@string/app_name"
                  android:configChanges="fontScale|keyboard|keyboardHidden|locale|mnc|mcc|navigation|orientation|screen
  <meta-data android:name="android.app.lib_name" android:value="unity" />
  <meta-data android:name="unityplayer.ForwardNativeEventsToDalvik" android:value="true" />
        <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
  </application>
```

```
</manifest>
```

Note the ".OverrideExampleNative" attribute in the activity element and the two additional meta-data elements. The first meta-data is an instruction to use the Unity library **libunity.so**. The second enables events to be passed on to your custom subclass of UnityPlayerNativeActivity.

### Examples

**Native Plugin Sample**
A simple example of the use of a native code plugin can be found here

This sample demonstrates how C code can be invoked from a Unity Android application. The package includes a scene which displays the sum of two values as calculated by the native plugin. Please note that you will need the Android NDK to compile the plugin.

**Java Plugin Sample**
An example of the use of Java code can be found here

This sample demonstrates how Java code can be used to interact with the Android OS and how C++ creates a bridge between C# and Java. The scene in the package displays a button which when clicked fetches the application cache directory, as defined by the Android OS. Please note that you will need both the JDK and the Android NDK to compile the plugins.

Here is a similar example but based on a prebuilt JNI library to wrap the native code into C#.

Page last updated: 2012-09-25

---

# Android Splash Screen

## iOS

Under iOS Basic, a default splash screen will be displayed while your game loads, oriented according to the **Default Screen Orientation** option in the Player Settings.

Users with an iOS Pro license can use any texture in the project as a splash screen. The size of the texture depends on the target device (320x480 pixels for 1-3rd gen devices, 1024x768 for iPad, 640x960 for 4th gen devices) and supplied textures will be scaled to fit if necessary. You can set the splash screen textures using the iOS Player Settings.

## Android

Under Android Basic, a default splash screen will be displayed while your game loads, oriented according to the **Default Screen Orientation** option in the Player Settings.

Android Pro users can use any texture in the project as a splash screen. You can set the texture from the Splash Image section of the Android Player Settings. You should also select the **Splash scaling** method from the following options:-

- **Center (only scale down)** will draw your image at its natural size unless it is too large, in which case it will be scaled down to fit.
- **Scale to fit (letter-boxed)** will draw your image so that the longer dimension fits the screen size exactly. Empty space around the sides in the shorter dimension will be filled in black.
- **Scale to fill (cropped)** will scale your image so that the shorter dimension fits the screen size exactly. The image will be cropped in the longer dimension.

Page last updated: 2011-11-08

---

# nacl-gettingstarted

Native Client (*NaCl*) is a new technology by Google which allows you to embed native executable code in web pages to allow deployment of very performant web apps without requiring the install of plugins. Currently, NaCl is only supported in Google Chrome on Windows, Mac OS X and Linux (with Chrome OS support being worked on), but the technology is open source, so it could be ported to other browser platforms in the future.

Unity 3.5 offers support to run Unity Web Player content (.unity3d files) using NaCl to allow content to be run without requiring a plugin install in Chrome. This is an early release - it should be stable to use, but it does not yet support all features supported in the Unity Web Player, because NaCl is an evolving platform, and does not support everything we can do in a browser plugin.

## Building and Testing games on NaCl

Building and testing games on NaCl is very simple. You need to have Google Chrome installed. Simply choose "Web Player" in Build Settings, and tick the "Enable NaCl" checkbox. This will make sure the generated unity3d file can be run on NaCl (by including GLSL ES shaders needed for NaCl, and by disabling dynamic fonts not supported by NaCl), and install the NaCl runtime and a html file to launch the game in NaCl. If you click Build & Run, Unity will install your player as an app in Chrome and launch it automatically.

## Shipping Games with NaCl

In its current state, NaCl is not enabled for generic web pages in Chrome by default. While you can embed a NaCl player into any web page, and direct your users to manually enable NaCl in chrome://flags, the only way to currently ship NaCl games and have them work out of the box is to deploy them on the Chrome Web Store (for which NaCl is enabled by default). Note that the Chrome Web Store is fairly unrestrictive, and allows you to host content embedded into your own web site, or to use your own payment processing system if you like. The plan is that this restriction will be lifted when Google has finished a new technology called portable NaCl (PNaCl), which lets you ship executables as LLVM bitcode, thus making NaCl apps independent of any specific CPU architectures. Then NaCl should be enabled for any arbitrary web site.

### Notes on Build size

When you make a NaCl build, you will probably notice that the *unity_nacl_files_3.x.x* folder is very large, over 100 MB. If you are wondering, if all this much data needs to be downloaded on each run for NaCl content, the answer is generally "no". There are two ways to serve apps on the Chrome Web Store, as a hosted or packaged app. If you serve your content as a packaged app, all data will be downloaded on install as a compressed archive, which will then be stored on the user's disk. If you serve your content as a hosted app, data will be downloaded from the web each time. But the nacl runtime will only download the relevant architecture (i686 or x86_64) from the *unity_nacl_files_3.x.x* folder, and when the web server is configured correctly, the data will be compressed on transfer, so the actual amount of data to be transferred should be around 10 MB (less when physics stripping is used). The *unity_nacl_files_3.x.x* folder contains a *.htaccess* file to set up Apache to compress the data on transfer. If you are using a different web server, you may have to set this up yourself.

## Limitations in NaCl

NaCl does not yet support all the features in the regular Unity Web Player. Support for many of these will be coming in future versions of Chrome and Unity. Currently, NaCl these features are unsupported by NaCl:

- Webcam Textures
- Joystick Input
- Caching
- Substances
- Dynamic Fonts
- Networking of any kind other then WWW class.
- The Profiler does not work, because it requires a network connection to the Editor.
- As with the standard webplayer plugin, native C/C++ plugins are not currently supported by NaCl.

The following features are supported, but have some limitations:

- Depth textures:

Depth textures are required for real-time shadows and other effects. Depth textures are supported in Unity NaCl, but Chrome's OpenGL ES 2.0 implementation does not support the required extensions on windows, so Depth textures will only work on OS X and Linux.

- Other graphics features:

NaCl uses OpenGL ES 2.0, which does not support all extensions included in the normal OpenGL. This means that some

features relying on extensions, such as linear and HDR lighting will not currently work on NaCl. Also Shaders need to be able to compile as GLSL shaders. Currently, not all built-in Unity shaders support this, for instance, the Screen Space Ambient Occlusion is not supported on GLSL.

- Cursor locking:

Cursor locking is supported, but only in fullscreen mode. Cursor locking in windowed mode is planned for a later Chrome release.

- NullReferenceExceptions:

NaCl does not have support for hardware exception handling. That means that a NullReferenceException in scripting code results in a crash in NaCl. You can, however pass `softexceptions="1"` to the embed parameters (set automatically by Unity when building a development player), to tell mono to do checking for NullReferences in software, which results in slower script execution but no crashes.

While Google does not give any system requirements for NaCl other then requiring at least OS X 10.6.7 on the Mac, we've found it to not work very well with old systems - especially when these systems have old GPUs or graphics drivers, or a low amount of installed main memory. If you need to target old hardware, you may find that the Web Player will give you a better experience.

### Fullscreen mode:

Fullscreen mode is supported by setting Screen.fullScreen, but you can only enter fullscreen mode in a frame where the user has released the mouse button. NaCl will not actually change the hardware screen resolution, which is why Screen.resolutions will only ever return the current desktop resolution. However, Chrome supports rendering into smaller back buffers, and scaling those up when blitting to the screen. So, requesting smaller resolutions then the desktop resolution is generally supported for fullscreen mode, but will result in GPU based scaling, instead of changing the screen mode.

### WWW class:

The WWW class is supported in NaCl, but follows different security policies then the Unity Web Player. While the Unity Web Player uses crossdomain.xml policy files, similar to flash, Unity NaCl has to follow the cross-origin security model followed by NaCl, documented here. Basically, in order to access html documents on a different domain then the player is hosted, you need to configure your web server to send a `Access-Control-Allow-Origin` respond header for the requests, which allows the domain hosting the player.

### Communicating with browser javascript in NaCl

Interacting with the web page using JavaScript is supported, and is very similar to using the Unity Web Player, with one exception: The syntax for sending messages to Unity from html javascript is different, because it has to go through the NaCl module. When you are using the default Unity-generated html, then this code will work:

```
document.getElementById('UnityEmbed').postMessage("GameObject.Message(parameter)");
```

### Logging

Since NaCl does not allow access to the user file system, it will not write log files. Instead it outputs all logging to stdout. To see the player logs from NaCl:

- Do a Build & Run in the edtior once to make sure your game is installed into Chrome as an app.
- On Mac OS X, start Chrome from a Terminal, and start the app by clicking on it's icon. You should see the Unity player log output in the terminal.
- On Windows it's the same, but you need to set the NACL_EXE_STDOUT and NACL_EXE_STDERR environment variables, and start Chrome with the --no-sandbox option. See Google's documentation.

Page last updated: 2012-09-20

---

## flash-gettingstarted

### What is Unity Flash?

The Flash build option allows Unity to publish swf (ShockWave Flash) files. These swf files can be played by a Flash plugin installed into your browser. Most computers in the world will either have a Flash Player installed, or can have one installed by visiting the Adobe Flash website. Just like a WebPlayer build creates a file with your 3d assets, audio, physics and scripts, Unity can build a SWF file. All the scripts from your game are automatically converted to ActionScript, which is the scripting

language that the Flash Player works with.

Note that the Unity Flash build option exports SWF files for playback in your browser. The SWF is not intended for playback on mobile platforms.

### Performance Comparison

We do not currently have direct comparisons of Unity webplayer content vs Flash SWF content. Much of our webplayer code is executed as native code, so for example, PhysX runs as native code. By comparison, when building a SWF file all of the physics runtime code (collision detection, newtonian physics) is converted to ActionScript. Typically you should expect the SWF version to run more slowly than the Unity webplayer version. We are, of course, doing everything we can to optimize for Flash.

### Further reading:

- Flash: Setup
- Flash: Building & Running
- Flash: Debugging
- Flash: What is and is not supported
- Flash: Embedding Unity Generated Flash Content in Larger Flash Projects
- Flash: Adobe Premium Features License
- Example: Supplying Data from Flash to Unity
- Example: Calling ActionScript Functions from Unity
- Example: Browser JavaScript Communication
- Example: Accessing the Stage

### Other Examples:

- Forums post - Loading Textures from Web (in AS3)

### Useful Resources:

- Scripting Reference: ActionScript
- Flash Development section on the Unity forums
- Flash questions on Unity Answers

Page last updated: 2012-10-24

---

## flash-setup

### Installing Unity for Flash

To view the SWF files that Unity creates, your web browser will need Adobe Flash Player 11.2 or newer, which you can obtain from http://get.adobe.com/flashplayer/. If you have Flash Player already installed, please visit http://kb2.adobe.com/cps/155/tn_15507.html to check that you have at least version 11.2. Adobe Flash Player 11 introduced the Stage 3D Accelerated Graphics Rendering feature that Unity requires for 3d rendering.

For system requirements see http://www.adobe.com/products/flashplayer/tech-specs.html

### Flash Player Switcher

This will allow you to switch between debug (slow) and regular (fast) versions of the Flash Player. Ensure you have Adobe AIR installed, or download it from http://get.adobe.com/air/. The Flash Player Switcher can be obtained from: https://github.com/jvanoostveen/Flash-Player-Switcher/downloads (select FlashPlayerSwitcher.air). Note: it currently supports only Mac OS X.

### Other Adobe Tools/Platforms

No other Adobe tools or platforms are required to develop with Unity and create SWF files. To embed the SWF that Unity builds into your own Flash Application you will need one of Adobe FlashBuilder/PowerFlasher FDT/FlashDeveloper/etc and be an experienced Flash developer. You will need to know:

- Your embedding application needs to be set to -swf-version=15 / fp11.2
- Your flash embeds wmode needs to be set to direct

Page last updated: 2012-10-24

## flash-building

The following is a step-by-step guide to build and run a new project exported to Flash.

1. Create your Unity content.
2. Choose File->Build Settings to bring up the Build Settings dialog and add your scene(s).
3. Change the Platform to Flash Player
4. Target Player can be left as the default. This option enables you to change the target Flash Player based on the features you require (see http://www.adobe.com/support/documentation/en/flashplayer/releasenotes.html for details).
5. Tick Development Build. (This causes Unity to **not** compress the final SWF file. Not compressing will make the build faster, and also, the SWF file will not have to be decompressed before being run in the Flash Player. Note that an empty scene built using the Development Build option will be around 16M in size, compared to around 2M compressed.)
6. Press the Build button.

Unity will build a SWF file at the location you choose. Additionally it will create the following files:

- an html file - Use this to view your Flash-built content.
- a swfobject.js file - Handles checking for the Flash Player and browser integration.
- an embeddingapi.swc file.

To view your Flash-built content open the html file. Do not open the SWF file directly.

Build-and-run will create the same files, launch your default browser and load the generated html file.

The embeddingapi.swc file created in the build allows you to load the SWF in your own project. Embedding the Unity content in a standard flash project allows you to do GUI in Flash. This type of Flash integration will of course not work in any of the other build targets.

As with the other build targets, there are Player settings that you can specify. Most of the Flash settings are shared with other platforms. Note that the resolution for the content is taken from the Standalone player settings.

We allow for a Flash API that gives you texture handles, which in combination with the swc embedding will give you means to do webcam, video, vector graphics from flash as textures.

### The Build Process

The Unity Flash Publisher attempts to convert scripts from C#/UnityScript into ActionScript. In this process, there can be two kinds of conversion errors:

- errors during conversion of unity code to ActionScript
- errors while compiling the converted code.

Errors during conversion will point to the original files and will have the familiar UnityScript error messages with file names and line numbers.

Errors during the compilation of the converted ActionScript will take you to the message in the generated ActionScript code (with filenames ending with `.as`).

### Debugging Converted ActionScript Code

During a build, the converted ActionScript (`.as`) files are stored within your project folder in:

- /Temp/StagingArea/Data/ConvertedDotNetCode/

If you encounter errors with your SWF (at runtime or during a build), it can be useful to look at this converted code.

It is possible that any ActionScript errors at compilation time will not be easily understood. Just remember that the ActionScript is generated from your game script code, so any changes you need to make will be in your original code and not the converted ActionScript files.

### Building for a specific Flash Player version

The dropdown box in the build settings window will enable you to choose which Flash Player version you wish to target. This will always default to the lowest supported Flash Player version (currently 11.2) upon creating/reopening your Unity project.

If you wish to build for a specific Flash Player version you can do so by creating an editor script to perform the build for you. In order to do this, you can specify a FlashBuildSubtarget in your EditorUserBuildSettings when building to Flash from an editor script. For example:

```
EditorUserBuildSettings.flashBuildSubtarget = FlashBuildSubtarget.Flash11dot2;
BuildPipeline.BuildPlayer(..., ..., BuildTarget.FlashPlayer, BuildOptions.Development);
```

### Example Build Errors and Warnings

Below are some common errors/warnings you may encounter when using the Flash export. We also have sections on the Forums and Answers dedicated to Flash export which may be of help if your error is not listed below.

#### Unable to find Java

```
Error building Player: Exception: Compiling SWF Failed: Unable to launch Java - is the Java Runtime Environment (JRE) installe
```

If you encounter the above error at build time, please install the 32-bit JRE and try again.

**'TerrainCollider' is not supported**

```
'TerrainCollider' is not supported when building for FlashPlayer.
'TerrainData' is not supported when building for FlashPlayer.
Asset: 'Assets/New Terrain.asset'
```

The terrain feature is not supported when building for the FlashPlayer target. All un-supported features will generate a similar warning. Note that the build will continue, however, the unsupported feature will be missing from the final SWF.

**Unboxing**

```
Error: Call to a possibly undefined method RuntimeServices_UnboxSingle_Object through a reference with static type Class.
```

This is likely because the conversion between types that is defined on the UnityScript side is not defined for our Flash Publisher. Any time you see an error that refers to Unbox it means a type conversion is required but cannot be found. In order to resolve these issues:

- Do not forget to use `#pragma strict`, and take care of all "implicit downcast" warning messages.
- The rule of thumb is to avoid runtime casts from Object to primitive types (int, float, etc.). Also prefer containers with explicit types to generic ones, for example:
  - System.Collections.Generic.List.<float> instead of Array
  - Dictionary<string, float> instead of Hashtable

**UnauthorizedAccessException**

```
Error building Player: UnauthorizedAccessException: Access to the path "Temp/StagingArea/Data/ConvertedDotNetCode/globa
```

If Unity-generated ActionScript files are open in a text editor, Unity may refuse to build issuing this error. To fix this, please close the ActionScript files and allow Unity to overwrite them.

Page last updated: 2012-11-06

---

# flash-debugging

### Where can I find my Flash Player log file?

Make sure you've done all of the following:

1) Install "content debugger" version of the Adobe Flash Player plugin from: http://www.adobe.com/support/flashplayer/downloads.html

2) Go to http://flashplayerversion.com/, and make sure that it says '`Debugger: Yes`'

3) Be careful using Chrome as it ships with its own Flash Player. If you wish to use Chrome with the debug Flash Player, you can do so by following these instructions: http://helpx.adobe.com/flash-player/kb/flash-player-google-chrome.html

4) Create a file called mm.cfg which will instruct the Flash Player to create a logfile. The mm.cfg file needs to be placed here:

| | |
|---|---|
| Macintosh OS X | /Library/Application Support/Macromedia/mm.cfg |
| XP | C:\Documents and Settings\username\mm.cfg |

| | |
|---|---|
| Windows Vista/Win7 | C:\Users\username\mm.cfg |
| Linux | /home/username/mm.cfg |

Write this text in the mm.cfg file:

```
ErrorReportingEnable=1
TraceOutputFileEnable=1
```

5) Find and open your flashlog.txt here:

| | |
|---|---|
| Macintosh OS X | /Users/username/Library/Preferences/Macromedia/Flash Player/Logs/ |
| XP | C:\Documents and Settings\username\Application Data\Macromedia\Flash Player\Logs |
| Windows Vista/Win7 | C:\Users\username\AppData\Roaming\Macromedia\Flash Player\Logs |
| Linux | /home/username/.macromedia/Flash_Player/Logs/ |

Note that whilst your content is running this flashlog.txt will constantly be updated as new debug messages are generated by your script code. You may need to reload the file or use an editor that can reload as the file grows in size.

More details about enabling debug logs when using SWFs is available at: http://livedocs.adobe.com/flex/3/html/help.html?content=logging_04.html.

Page last updated: 2012-11-06

---

# flash-whatssupported

## Supported

- Flash Player 11.2, 11.3 and 11.4
- Full ActionScript API Access
- Lightmapping
- Occlusion culling
- Editor Scripting (JavaScript / C# / Boo). Note: for JavaScript, use `#pragma strict`.
- Custom shaders
- Animation / skinning
- Basic types like int, string, List
- Basic audio features, such as AudioSource / AudioListener
- Physics
- Navigation Meshes
- Substance Textures, however the textures are baked at build time so cannot be dynamically changed at runtime
- PlayerPrefs - On Flash PlayerPrefs are stored per SWF per machine
- UnityGUI classes that do not require text input
- Particle System (Shuriken) works and is script accessible
- Asset bundles - These are supported but caching of bundles (i.e. use of LoadFromCacheOrDownload) is not currently supported
- WWW and WWWForm
- Mecanim

## Limited support

- Realtime shadows work, but do get affected by bugs in image effects
- Untyped variables in JavaScript and implicit type conversions
- Unity GUI / Immediate mode GUI
- Any .NET specific stuff. Do not use stuff from exotic class libraries (reflection, LINQ etc).
- GUIText wil have a dramatic impact on performance

### Not Currently Supported

- Image Effects
- Unity profiler
- UnityGUI classes that require text input
- Raknet networking (if you need networking, you can write it in Action Script 3 directly, using flash API)
- Cloth
- VertexLit shaders currently do not support Spot Lights (they are treated just like point lights).
- Advanced audio features, such as audio effects
- Terrain
- Texture mipMapBias
- Non-triangle MeshTopology and wireframe rendering
- AsyncOperation

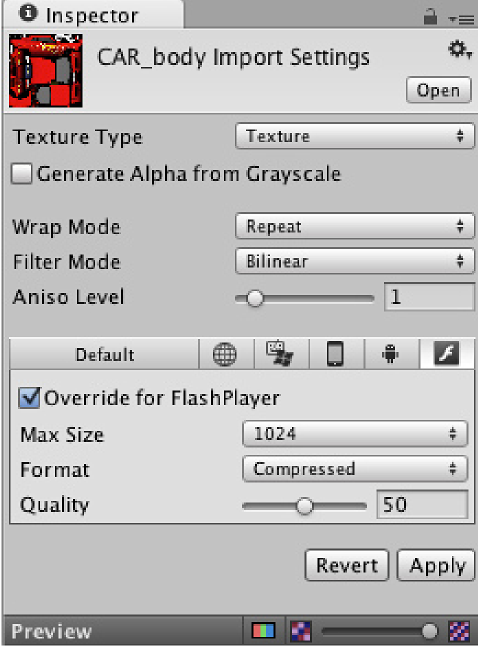
### Won't be supported

- Sockets - It is possible to use ActionScript sockets by implementing them in AS3.
- Deferred rendering

### Texture Support

We support j peg textures, as well as RGBA / Truecolor. Textures which are jpg-xr compressed are not readable and thus not supported.

The compression ratio can be specified in the texture import under 'Override for FlashPlayer' setting. Compressed textures get converted to j peg with the chosen compression ratio. The compression ratio is worth experimenting with since it can considerably reduce the size of the final SWF.




Texture quality ranges from 0 to 100, with 100 indicating no compression, and 0 the highest amount of compression possible.

The maximum supported texture resolution is 2048x2048.

### Unavailable APIs

- UnityEngine.AccelerationEvent
- UnityEngine.Achievement

- UnityEngine.AchievementDescription
- UnityEngine.GameCenter
- UnityEngine.GcLeaderboard
- UnityEngine.IDList
- UnityEngine.ISocial
- UnityEngine.Leaderboard
- UnityEngine.LocalServices
- UnityEngine.RectOffset
- UnityEngine.Score
- UnityEngine.Security
- UnityEngine.Serialization.ListSerializationSurrogate
- UnityEngine.Serialization.UnitySurrogateSelector
- UnityEngine.Social
- UnityEngine.StackTraceUtility
- UnityEngine.TextEditor
- UnityEngine.Types
- UnityEngine.UnityException
- UnityEngine.UnityLogWriter
- UnityEngine.UserProfile

Page last updated: 2012-11-06

---

# flash-embeddingapi

### embeddingapi.swc

If you want to embed your Unity generated Flash content within a larger Flash project, you can do so using the **embeddingapi.swc**. This SWC provides functionality to load and communicate with Unity published Flash content. In the **embeddingapi.swc** file, you will find two classes and two interfaces. Each of these, and their available functions, are described below.

When your Unity Flash project is built, a copy of the embeddingapi.swc file will be placed in the same location as your built SWF. You can then use this in your Flash projects as per other SWCs. For more details on what SWCs are and how to use them, see Adobe's documentation.

### Stage3D Restrictions

When embedding your Unity Flash content within another Flash project, it is useful to understand the Flash display model. All Stage3D content is displayed behind the Flash Stage. This means that any Flash display list content added to the Stage will always render in front of your 3D content. For more information on this, please refer to Adobe's "How Stage3D Works" page.

### IUnityContent

**IUnityContent** is implemented by Unity built Flash content. This interface is how you communicate with or modify the Untiy content.

**Methods:**

| | |
|---|---|
| getTextureFromNativeId(id : int) : TextureBase; | Enables retrieving of textures. A full example project using this can be found on the forums. |
| sendMessage(objectPath : String, methodName : String, value : Object = null) : Boolean; | The **sendMessage** function can be used to call a method on an object in the Unity content. |
| setContentHost(contentHost : IUnityContentHost) : void; | Sets the host (which must implement **IUnityContentHost**) for the Unity content. The host can then listen for when the Unity content has loaded/started. |
| setSize(width : int, height : int) : void; | Modifies the size of the Unity content |
| setPosition(x:int = 0, y:int = 0):void; | Enables you to reposition the Unity content within the content host. |
| startFrameLoop() : void; | Starts the Unity content. |

| | |
|---|---|
| stopFrameLoop() : void; | Stops the unity content. |
| forceUnload():void; | Unloads the Unity flash content. |

### IUnityContentHost

This must be implemented by whichever class will host the Unity content.

**Methods:**

| | |
|---|---|
| unityInitComplete() : void; | Called when the Unity engine is done initializing and the first level is loaded. |
| unityInitStart() : void; | Called when the content is loaded and the initialization of the Unity engine is started. |

### UnityContentLoader

The **UnityContentLoader** class can be used to load Unity published Flash content and extends the AS3 Loader class. As with standard AS3 Loader instances, you can add event listeners to its **contentLoaderInfo** in order to know the progress of the load and when it is complete.

**Constructor:**

```
UnityContentLoader(contentURL : String, contentHost : IUnityContentHost = null, params : UnityLoaderParams = null, autoLc
```

Creates a **UnityContentLoader** instance which you can attach event listeners to and use to load the unity content.

- **contentURL**: The URL of the Unity published SWF to load.
- **contentHost**: The host for the content. This should be your own ActionScript class that implements **IUnityContentHost**.
- **params**: Supply a **UnityLoaderParams** instance if you wish to override the default load details.
- **autoLoad**: If set to true, the load will begin as soon as the **UnityContentLoader** has been created (rather than needing to call **loadUnity()** separately). If you wish to track progress of the load using events, this should be set to false. You can then call **loadUnity()** manually once the relevant event listeners have been added.

**Accessible Properties:**

| | |
|---|---|
| unityContent : IUnityContent; | Once the content has finished loading, you can access the Unity content to perform functionality such as **sendMessage()**. |

**Methods:**

| | |
|---|---|
| loadUnity() : void; | Instructs the **UnityContentLoader** to load the Unity content from the URL supplied in the constructor. |
| forceUnload() : void; | Unloads the unity content from the host. |
| unload() : void; | Overrides the default unload() method of the AS3 Loader class and calls forceUnload. |
| unloadAndStop(gc:Boolean = true):void | Unloads the unity content then calls the default Loader implementation of unloadAndStop(gc). |

### UnityLoaderParams

**Constructor:**

Parameters can be supplied to the **UnityContentLoader** when created to provide additional loader configuration.

```
function UnityLoaderParams(scaleToStage : Boolean = false, width : int = 640, height : int = 480, usePreloader : Boolean = t
```

- **scaleToStage**: Whether the Unity content remains at a fixed size or whether it scales as the parent Flash window resizes.
- **width**: The width of the Unity content.
- **height**: The height of the Unity content.
- **usePreloader**: Whether or not to show the Unity preloader.
- **autoInit**: This is not currently used.
- **catchGlobalErrors**: Whether to catch errors and display them in a red box in the top left corner of the swf.

### Example

The following example shows how to load Unity published Flash content into a host SWF. It shows how to supply custom **UnityLoaderParams** and track progress of the file load. Once the Unity content has been added to the host, a function in the Unity content is called using the **sendMessage** function.

```
public class MyLoader extends Sprite implements IUnityContentHost
{
  private var unityContentLoader:UnityContentLoader;

  public function MyLoader()
  {
      var params:UnityLoaderParams = new UnityLoaderParams(false,720,400,false);
      unityContentLoader = new UnityContentLoader("UnityContent.swf", this, params, false);
      unityContentLoader.contentLoaderInfo.addEventListener(ProgressEvent.PROGRESS, onUnityContentLoaderProgress);
      unityContentLoader.contentLoaderInfo.addEventListener(Event.COMPLETE, onUnityContentLoaderComplete);
      unityContentLoader.loadUnity();
  }

  private function onUnityContentLoaderProgress(event:ProgressEvent):void
  {
      //Respond to load progress
  }

  private function onUnityContentLoaderComplete(event:Event):void
  {
     addChild(unityContentLoader);
     unityContentLoader.unityContent.setContentHost(this);
  }

  //unityInitStart has to be implemented by whatever implements IUnityContenthost
  //This is called when the content is loaded and the initialization of the unity engine is started.
  public function unityInitStart():void
  {
    //Unity engine started
  }

  //unityInitComplete has to be implemented by whatever implements IUnityContenthost
  //This is called when the unity engine is done initializing and the first level is loaded.
  public function unityInitComplete():void
  {
     unityContentLoader.unityContent.sendMessage("Main Camera","SetResponder",{responder:this});
  }

  ...

}
```

Page last updated: 2012-11-06

---

# flash-adobelicense

### What is the license and why is it needed?

When publishing your Unity project to Flash, you will need to acquire a license from Adobe in order for the content to work in the Flash Player. The Adobe documentation of premium features explains why a license is required for Unity built Flash games:

```
"Premium Features includes the XC APIs �(domain memory APIs in combination with Stage3D hardware acceleration APIs), w
```

For more information and the latest details on the license, please refer to the Adobe article which explains this in detail.

### How do I obtain a license?

To obtain a license, you will need to sign into https://www.adobefpl.com/ using your AdobeId and follow their instructions.

### Further reading

- Premium Features for Flash Player FAQs
- Adobe Premium Features for Flash Player
- Adobe gaming

Page last updated: 2012-11-06

---

## flashexamples-supplyingdata

If you wish to supply data from Flash to Unity, it must be one of the supported types. You can also create classes to represent the data (by providing a matching C# or JavaScript implementation).

First, create an AS3 implementation of your object and include the class in your project (in an folder called ActionScript):

```
public class ExampleObject
{
    public var anInt : int;
    public var someString : String;
    public var aBool : Boolean;
}
```

Now create a C# or JavaScript object which matches the AS3 implementation.

The NotRenamed attribute used below prevents name mangling of constructors, methods, fields and properties.

The NotConverted attribute instructs the build pipeline not to convert a type or member to the target platform. Normally when you build to Flash, each of your C#/JavaScript scripts are converted to an ActionScript (`.as`) script. Adding the [NotConverted] attribute overrides this process, allowing you to provide your own version of the .as script, manually. The dummy C#/JavaScript which you provide allows Unity to know the signature of the class (i.e. which functions it should be allowed to call), and your .as script provides the implementations of those functions. Note that the ActionScript version will only be used when you build to Flash. In editor or when built to other platforms, Unity will use your C#/JavaScript version.

**C#**

```
[NotConverted]
[NotRenamed]
public class ExampleObject
{
    [NotRenamed]
    public int anInt;

    [NotRenamed]
    public string someString;

    [NotRenamed]
    public bool aBool;

```

```
}
```

**JavaScript**

```
@NotConverted
@NotRenamed
class ExampleObject
{
   @NotRenamed
   public var anInt : int;

   @NotRenamed
   public var someString : String;

   @NotRenamed
   public var aBool : boolean;
}
```

Now you need a way in AS3 to retrieve your object, e.g.:

```
public static function getExampleObject() : ExampleObject
{
   return new ExampleObject();
}
```

Then you can then retrieve the object and access its data:

```
ExampleObject exampleObj = UnityEngine.Flash.ActionScript.Expression<ExampleObject>("MyStaticASClass.getExampleObjec
Debug.Log(exampleObj.someString);
```

Page last updated: 2012-10-24

---

# flashexamples-callingflashfunctions

This example shows how you can call different AS3 functions from Unity. You will encounter three scripts:

- An AS3 class (ExampleClass.as) containing different function examples. Any AS3 classes you create must be placed within an "ActionScript" folder in your project.
- A C#/JavaScript class (ExampleClass.cs/js) which mimics the AS3 implementation. You only need one of these.
- An example of how to call the functions from Unity.

When built to Flash, the AS3 implementation of ExampleClass is used. When run in-editor or built to any platform other than Flash the C#/JavaScript implementation will be used.

By creating an ActionScript version of your classes, this will enable you to use native AS3 libraries when building for Flash Player. This is particularly useful when you need to work around a .net library which isn't yet supported for Flash export.

## ExampleClass.as

```
public class ExampleClass
{
  public static function aStaticFunction() : void
  {
    trace("aStaticFunction - AS3 Implementation");
```

```
  }

  public static function aStaticFunctionWithParams(a : int) : void
  {
    trace("aStaticFunctionWithParams - AS3 Implementation");
��}

  public static function aStaticFunctionWithReturnType() : int
  {
    trace("aStaticFunctionWithReturnType - AS3 Implementation");
    return 1;
  }

  public function aFunction() : void
  {
    trace("aFunction - AS3 Implementation");
  }
}
```

### ExampleClass - C#/JavaScript Implementation

You can create the class to mimic the AS3 implementation in either C# or JavaScript. The implementations are very similar. Both examples are provided below.

#### C# Implementation (ExampleClass.cs)

```
using UnityEngine;

[NotRenamed]
[NotConverted]
public class ExampleClass
{
    [NotRenamed]
    public static void aStaticFunction()
    {
        Debug.Log("aStaticFunction - C# Implementation");
    }

    [NotRenamed]
    public static void aStaticFunctionWithParams(int a)
    {
        Debug.Log("aStaticFunctionWithParams - C# Implementation");
    }

    [NotRenamed]
    public static int aStaticFunctionWithReturnType()
    {
        Debug.Log("aStaticFunctionWithReturnType - C# Implementation");
        return 1;
    }

    [NotRenamed]
    public void aFunction()
    {
        Debug.Log("aFunction - C# Implementation");
    }
}
```

#### JavaScript Implementation (ExampleClass.js)

```
@NotConverted
@NotRenamed
class ExampleClass
{
   @NotRenamed
   static function aStaticFunction()
   {
      Debug.Log("aStaticFunction - JS Implementation");
   }

   @NotRenamed
   static function aStaticFunctionWithParams(a : int)
   {
      Debug.Log("aStaticFunctionWithParams - JS Implementation");
   }

   @NotRenamed
   static function aStaticFunctionWithReturnType() : int
   {
     Debug.Log("aStaticFunctionWithReturnType - JS Implementation");
     return 1;
   }

   @NotRenamed
   function aFunction()
   {
      Debug.Log("aFunction - JS Implementation");
   }
}
```

### How to Call the Functions

The below code will call the methods in the ActionScript (.as) implementation when building for Flash. This will allow you to use native AS3 libraries in your flash export projects. When building to a non-Flash platform or running in editor, the C#/JS implementation of the class will be used.

```
ExampleClass.aStaticFunction();
ExampleClass.aStaticFunctionWithParams(1);
int returnedValue = ExampleClass.aStaticFunctionWithReturnType();

ExampleClass exampleClass = new ExampleClass();
exampleClass.aFunction();
```

Page last updated: 2012-11-06

---

## flashexamples-browserjavascriptcommunication

This example shows how AS3 code can communicate JavaScript in the browser. This example makes use of the ExternalInterface ActionScript class.

When run, the BrowserCommunicator.TestCommunication() function will register a callback that the browser JavaScript can then call. The ActionScript will then call out to the browser JavaScript, causing an alert popup to be displayed. The exposed ActionScript function will then be invoked by the JavaScript, completing the two-way communication test.

### Required JavaScript

The following JavaScript needs to be added to the html page that serves the Unity published SWF. It creates the function which will be called from ActionScript:

```
<script type="text/javascript">

function calledFromActionScript()
{
    alert("ActionScript called Javascript function")

    var obj = swfobject.getObjectById("unityPlayer");
    if (obj)
    {
        obj.callFromJavascript();
    }
}

</script>
```

### BrowserCommunicator.as (and matching C# class)

```
package
{
    import flash.external.ExternalInterface;
    import flash.system.Security;

    public class BrowserCommunicator
    {
        //Exposed so that it can be called from the browser JavaScript.
        public static function callFromJavascript() : void
        {
            trace("Javascript successfully called ActionScript function.");
        }

        //Sets up an ExternalInterface callback and calls a Javascript function.
        public static function TestCommunication() : void
        {
            if (ExternalInterface.available)
            {
                try
                {
                    ExternalInterface.addCallback("callFromJavascript", callFromJavascript);
                }
                catch (error:SecurityError)
                {
                    trace("A SecurityError occurred: " + error.message);
                }
                catch (error:Error)
                {
                    trace("An Error occurred: " + error.message);
                }

                ExternalInterface.call('calledFromActionScript');
            }
            else
            {
```

```
            trace("External interface not available");
        }
    }
  }
}
```

C# dummy implementation of the class:

```
[NotConverted]
[NotRenamed]
public class BrowserCommunicator
{
   [NotRenamed]
   public static void TestCommunication()
   {
   }
}
```

### How to test

Simply call BrowserCommunicator.TestCommunication() and this will invoke the two-way communication test.

### Potential Issues

**Security Sandbox Violation**
*A SecurityError occurred: Error #2060: Security sandbox violation*

This happens when your published SWF does not have permission to access your html file. To fix this locally, you can either:

- Add the folder containing the SWF to the Flash Player's trusted locations in the Global Security Settings Panel.
- Host the file on localhost.

For more information on the Flash Security Sandboxes, please refer to the Adobe documentation.

Page last updated: 2012-10-24

---

## flashexamples-accessingthestage

You can access the Flash Stage from your C#/JS scripts in the following way:

```
ActionScript.Import("com.unity.UnityNative");
ActionScript.Statement("trace(UnityNative.stage);");
```

As an example, the following C# code will output the flashvars supplied to a SWF:

```
ActionScript.Import("flash.display.LoaderInfo");
ActionScript.Statement(
    "var params:Object = LoaderInfo(UnityNative.stage.loaderInfo).parameters;" +
    "var key:String;" +
    "for (key in params) {" +
        "trace(key + '=' + params[key]);" +
    "}"
```

```
);
```

Page last updated: 2012-11-06

# FAQ

The following is a list of common tasks in Unity and how to accomplish them.

- Upgrade Guide from Unity 3.5 to 4.0
- Unity 3.5 upgrade guide
- Upgrading your Unity Projects from 2.x to 3.x
  - Physics upgrade details
  - Mono Upgrade Details
  - Rendering upgrade details
  - Unity 3.x Shader Conversion Guide
- Unity 4.0 Activation - Overview
  - Managing your Unity 4.x license
  - Step-by-Step Guide to Online Activation of Unity 4.0
  - Step-by-Step Guide to Manual Activation of Unity 4.0
- Game Code Questions
  - How to make a simple first person walkthrough
- Graphics Questions
  - How do I Import Alpha Textures?
  - How do I Use Normal Maps?
  - How do I use Detail Textures?
  - How do I Make a Cubemap Texture?
  - How do I Make a Skybox?
  - How do I make a Mesh Particle Emitter? (Legacy Particle System)
  - How do I make a Splash Screen?
  - How do I make a Spot Light Cookie?
  - How do I fix the rotation of an imported model?
  - How do I use Water?
- FBX export guide
- Art Asset Best-Practice Guide
- How do I import objects from my 3D app?
  - Importing Objects From Maya
  - Importing Objects From Cinema 4D
  - Importing Objects From 3D Studio Max
  - Importing Objects From Cheetah3D
  - Importing Objects From Modo
  - Importing Objects From Lightwave
  - Importing Objects From Blender
- Workflow Questions
  - Getting started with Mono Develop
  - How do I reuse assets between projects?
  - How do I install or upgrade Standard Assets?
  - Porting a Project Between Platforms
- Mobile Developer Checklist
  - Crashes
  - Profiling
  - Optimizations

Page last updated: 2007-11-16

# Upgrade guide from 3.5 to 4.0

## GameObject active state

Unity 4.0 changes how the active state of GameObjects is handled. GameObject's active state is now inherited by child GameObjects, so that any GameObject which is inactive will also cause its children to be inactive. We believe that the new behavior makes more sense than the old one, and should have always been this way. Also, the upcoming new GUI system heavily depends on the new 4.0 behavior, and would not be possible without it. Unfortunately, this may require some work to fix existing projects to work with the new Unity 4.0 behavior, and here is the change:

**The old behavior:**

- Whether a GameObject is active or not was defined by its **.active** property.
- This could be queried and set by checking the **.active** property.
- A GameObject's active state had no impact on the active state of child GameObjects. If you want to activate or deactivate a GameObject and all of its children, you needed to call **GameObject.SetActiveRecursively**.
- When using **SetActiveRecursively** on a GameObject, the previous active state of any child GameObject would be lost. When you deactivate and then activated a GameObject and all its children using **SetActiveRecursively**, any child which had been inactive before the call to **SetActiveRecursively**, would become active, and you had to manually keep track of the active state of children if you want to restore it to the way it was.
- Prefabs could not contain any active state, and were always active after prefab instantiation.

**The new behavior:**

- Whether a GameObject is active or not is defined by its own **.activeSelf** property, and that of all of its parents. The GameObject is active if its own **.activeSelf** property and that of all of its parents is **true**. If any of them are **false**, the GameObject is inactive.
- This can be queried using the **.activeInHierarchy** property.
- The **.activeSelf** state of a GameObject can be changed by calling **GameObject.SetActive**. When calling **SetActive (false)** on a previously active GameObject, this will deactivate the GameObject and all its children. When calling **SetActive (true)** on a previously inactive GameObject, this will activate the GameObject, if all its parents are active. Children will be activated when all their parents are active (i.e., when all their parents have **.activeSelf** set to **true**).
- This means that **SetActiveRecursively** is no longer needed, as active state is inherited from the parents. It also means that, when deactivating and activating part of a hierarchy by calling **SetActive**, the previous active state of any child GameObject will be preserved.
- Prefabs can contain active state, which is preserved on prefab instantiation.

**Example:**

You have three GameObjects, A, B and C, so that B and C are children of A.

- Deactivate C by calling **C.SetActive(false)**.
- Now, **A.activeInHierarchy == true**, **B.activeInHierarchy == true** and **C.activeInHierarchy == false**.
- Likewise, **A.activeSelf == true**, **B.activeSelf == true** and **C.activeSelf == false**.
- Now we deactivate the parent A by calling **A.SetActive(false)**.
- Now, **A.activeInHierarchy == false**, **B.activeInHierarchy == false** and **C.activeInHierarchy == false**.
- Likewise, **A.activeSelf == false**, **B.activeSelf == true** and **C.activeSelf == false**.
- Now we activate the parent A again by calling **A.SetActive(true)**.
- Now, we are back to **A.activeInHierarchy == true**, **B.activeInHierarchy == true** and **C.activeInHierarchy == false**.
- Likewise, **A.activeSelf == true**, **B.activeSelf == true** and **C.activeSelf == false**.

**The new active state in the editor**

To visualize these changes, in the Unity 4.0 editor, any GameObject which is inactive (either because it's own **.activeSelf** property is set to **false**, or that of one of it's parents), will be greyed out in the hierarchy, and have a greyed out icon in the inspector. The GameObject's own **.activeSelf** property is reflected by it's active checkbox, which can be toggled regardless of parent state (but it will only activate the GameObject if all parents are active).

**How this affects existing projects:**

- To make you aware of places in your code where this might affect you, the **GameObject.active** property and the **GameObject.SetActiveRecursively()** function have been deprecated.
- They are, however still functional. Reading the value of **GameObject.active** is equivalent to reading **GameObject.activeInHierarchy**, and setting **GameObject.active** is equivalent to calling **GameObject.SetActive()**. Calling **GameObject.SetActiveRecursively()** is equivalent to calling **GameObject.SetActive()** on the GameObject and all of it's children.
- Exiting scenes from 3.5 are imported by setting the **selfActive** property of any GameObject in the scene to it's previous **active** property.
- As a result, any project imported from previous versions of Unity should still work as expected (with compiler warnings, though), as long as it does not rely on having active children of inactive GameObjects (which is no longer possible in Unity

4.0).
- If your project relies on having active children of inactive GameObjects, you need to change your logic to a model which works in Unity 4.0.

## Changes to the asset processing pipeline

During the development of 4.0 our asset import pipeline has changed in some significant ways internal in order to improve performance, memory usage and determinism. For the most part these changes does not have an impact on the user with one exception: Objects in assets are not made persistent until the very end of the import pipeline and any previously imported version of an assets will be completely replaced.

The first part means that during post processing you cannot get the correct references to objects in the asset and the second part means that if you use the references to a previously imported version of the asset during post processing do store modification those modifications will be lost.

**Example of references being lost because they are not persistent yet**
Consider this small example:

```
public class ModelPostprocessor : AssetPostprocessor
{
    public void OnPostprocessModel(GameObject go)
    {
        PrefabUtility.CreatePrefab("Prefabs/" + go.name, go);
    }
}
```

In Unity 3.5 this would create a prefab with all the correct references to the meshes and so on because all the meshes would already have been made persistent, but since this is not the case in Unity 4.0 the same post processor will create a prefab where all the references to the meshes are gone, simply because Unity 4.0 does not yet know how to resolve the references to objects in the original model prefab. To correctly copy a modelprefab in to prefab you should use **OnPostProcessAllAssets** to go through all imported assets, find the modelprefab and create new prefabs as above.

**Example of references to previously imported assets being discarded**
The second example is a little more complex but is actually a use case we have seen in 3.5 that broke in 4.0. Here is a simple **ScriptableObject** with a references to a mesh.

```
public class Referencer : ScriptableObject
{
    public Mesh myMesh;
}
```

We use this **ScriptableObject** to create an asset with references to a mesh inside a model, then in our post processor we take that reference and give it a different name, the end result being that when we have reimported the model the name of the mesh will be what the post processor determines.

```
public class Postprocess : AssetPostprocessor
{
       public void OnPostprocessModel(GameObject go)
       {
               Referencer myRef  = (Referencer)AssetDatabase.LoadAssetAtPath("Assets/MyRef.ass
               myRef.myMesh.name = "AwesomeMesh";
       }
}
```

This worked fine in Unity 3.5 but in Unity 4.0 the already imported model will be completely replaced, so changing the name of the mesh from a previous import will have no effect. The Solution here is to find the mesh by some other means and change its name. What is most important to note is that in Unity 4.0 you should ONLY modify the given input to the post processor and not rely on the previously imported version of the same asset.

### Mesh Read/Write option

Unity 4.0 adds a "Read/Write Enabled" option in Mesh import settings. When this option is turned off, it saves memory since Unity can unload a copy of mesh data in the game.

However, if you are scaling or instantiating meshes at runtime with a non-uniform scale, you may have to enable "Read/Write Enabled" in their import settings. The reason is that non-uniform scaling requires the mesh data to be kept in memory. Normally we detect this at build time, but when meshes are scaled or instantiated at runtime you need to set this manually. Otherwise they might not be rendered in game builds correctly.

### Mesh optimization

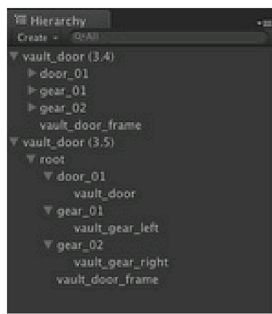
The Model Importer in Unity 4.0 has become better at mesh optimization. The "Mesh Optimization" checkbox in the Model Importer in Unity 4.0 is now enabled by default, and will reorder the vertices in your Mesh for optimal performance. You may have some post-processing code or effects in your project which depend on the vertex order of your meshes, and these might be broken by this change. In that case, turn off "Mesh Optimization" in the Mesh importer. Especially, if you are using the SkinnedCloth component, mesh optimization will cause your vertex weight mapping to change. So if you are using SkinnedCloth in a project imported from 3.5, you need to turn off "Mesh Optimization" for the affected meshes, or reconfigure your vertex weights to match the new vertex order.

Page last updated: 2012-11-12

---

## Upgrade guide from 3.4 to 3.5

If you have an FBX file with a root node marked up as a skeleton, it will be imported with an additional root node in 3.5, compared to 3.4




Unity 3.5 does this because when importing animated characters, the most common setup is to have one root node with all bones below and a skeleton next to it in the hierarchy. When creating additional animations, it is common to remove the skinned mesh from the fbx file. In that case the new import method ensures that the additional root node always exists and thus animations and the skinned mesh actually match.

If the connection between the instance and the FBX file's prefab has been broken in 3.4 the animation will not match in 3.5, and as a result your animation might not play.

In that case it is recommended that you recreate the prefabs or Game Object hierarchies by dragging your FBX file into your scene and recreating it.

Page last updated: 2012-02-03

---

## HowToUpgradeFrom2xTo3x

In our regular point releases of Unity, we make sure that projects from previous minor versions of the same major version are automatically upgraded when opened in the new editor for the first time. New properties are given default values, formats are converted and so on. However for major version changes such as 2.x to 3.x, we introduce several backwards-compatibility breaking changes.

While the primary visibility of this is the fact that content authored in the previous version will play back slightly differently when run in the new engine, some changes require more than a few property tweaks to play just like you want them to. These documents outlines those changes from 2.x to 3.x:

- Physics upgrade details
- Mono Upgrade Details
- Rendering upgrade details
- Unity 3.x Shader Conversion Guide

Page last updated: 2010-09-30

---

# PhysicsUpgradeDetails

For Unity 3.0, we upgraded the NVIDIA PhysX library from version 2.6 to 2.8.3, to give you access to many new features. Generally for existing projects, the behavior should be roughly the same as in Unity 2.x, but there may be slight differences in the outcome of physics simulation, so if your content depends on the exact behavior or on chain reactions of physical events, you may have to re-tweak your setup to work as expected in Unity 3.x.

If you are using Configurable Joints, the JointDrive.maximumForce property will now also be taken into consideration when JointDrive.mode is JointDriveMode.Position. If you have set this value to the default value of zero, the joint will not apply any forces. We will automatically change all JointDrive properties imported from old versions if JointDrive.mode is JointDriveMode.Position, but when you set up a joint from code, you may have to manually change this. Also, note that we have changed the default value for JointDrive.maximumForce to infinity.

Page last updated: 2010-09-25

---

# MonoUpgradeDetails

In Unity 3 we upgraded the mono runtime from 1.2.5 to 2.6 and on top of that, there are some JavaScript and Boo improvements. Aside from all bug fixes and improvements to mono between the two versions, this page lists some of the highlights.

### C# Improvements

Basically the differences betweeen C# 3.5 and C# 2.0, including:

- Variable type inference. More info here.
- Linq .
- Lambdas. More info here.

### JavaScript Improvements

- Compiler is now 4x faster;
- 'extends' no longer can be used with interfaces, unityscript now have 'implements' for that purpose (see below);
- Added support for consuming generic types such as generic collections:

```
var list = new System.Collections.Generic.List.<String>();
list.Add("foo");
```

- Added support for anonymous functions/closures:

```
list.Sort(function(x:String, y:String) {
     return x.CompareTo(y);
```

```
});
```

- Which include a simplified lambda expression form with type inference for the parameters and return value:

```
list.Sort(function(x, y) x.CompareTo(y));
```

- Function types:

```
function forEach(items, action: function(Object)) {
    for (var item in items) action(item);
}
```

- Type inferred javascript array comprehensions:

```
function printArray(a: int[]) {
    print("[" + String.Join(", ", [i.ToString() for (i in a)]) + "]");
}

var doubles = [i*2 for (i in range(0, 3))];
var odds = [i for (i in range(0, 6)) if (i % 2 != 0)];
printArray(doubles);
printArray(odds);
```

- Added support for declaring and implementing interfaces:

```
interface IFoo {
    function bar();
}

class Foo implements IFoo {
    function bar() {
        Console.WriteLine("Foo.bar");
    }
}
```

- All functions are now implicitly virtual, as a result the 'virtual' keyword has been deprecated and the 'final' keyword has been introduced to allow for non virtual methods to be defined as:

```
final function foo() {
}
```

- Value types (structs) can be defined as classes inheriting from System.ValueType:

```
class Pair extends System.ValueType {
    var First: Object;
    var Second: Object;

    function Pair(fst, snd) {
        First = fst;
        Second = snd;
    }

    override function ToString() {
        return "Pair(" + First + ", " + Second + ")";
    }
}
```

### Boo Improvements

- Boo upgrade to version 0.9.4.

Page last updated: 2011-11-08

## RenderingUpgradeDetails

Unity 3 brings a lot of graphics related changes, and some things might need to be tweaked when you upgrade existing Unity 2.x projects. For changes related to shaders, see Shader Upgrade Guide.

### Forward Rendering Path changes

Unity 2.x had one rendering path, which is called Forward in Unity 3. Major changes in it compared to Unity 2.x:

- Most common case (one directional per-pixel light) is drawn in one pass now! (used to be two passes)
- Point & Spot light shadows are not supported. Only one Directional light can cast shadows. Use Deferred Lighting path if you need more shadows.
- Most "Vertex" lights replaced with Spherical Harmonics lighting.
- Forward rendering path is purely shader based now, so it works on OpenGL ES 2.0, Xbox 360, PS3 (i.e. platforms that don't support fixed function rendering).

### Shader changes

See Shader Upgrade Guide for more details. Largest change is: if you want to write shaders that interact with lighting, you should use Surface Shaders.

### Obscure Graphics Changes That No One Will Probably Notice TM

- Removed Mac Radeon 9200 pixel shader support (`!!ATIfs` assembly shaders).
- Removed support for per-pixel lighting on pre-ShaderModel2.0 hardware. As a result, Diffuse Fast shader is just VertexLit now.
- Removed non-attenuated lights. All point and spot lights are attenuated now.
- Removed script callbacks: `OnPreCullObject` and `RenderBeforeQueues` attribute.
- Removed p-buffer based RenderTextures. RenderTextures on OpenGL require FBO support now.
- Most Pass LightMode tags are gone, and replaced with new tags. You should generally be using Surface Shaders for that stuff anyway.
- Texture instanceIDs are not OpenGL texture names anymore. Might affect C++ Plugins that were relying on that; use `texture.GetNativeTextureID()` instead.
- Rename shader keywords SHADOWS_NATIVE to SHADOWS_DEPTH; SHADOWS_PCF4 to SHADOWS_SOFT.
- Removed ambient boost on objects that were affected by more than 8 vertex lights.
- Removed `_ObjectSpaceCameraPos` and `_ObjectSpaceLightPos0` (added `_WorldSpaceCameraPos` and `_WorldSpaceLightPos0`).
- `LightmapMode` tag in shader texture property does nothing now.
- Skybox shaders do not write into depth buffer.
- `GrabPass` (i.e. refractive glass shader) now always grabs texture of the size of the screen.
- `#pragma multi_compile_vertex` and `#pragma multi_compile_fragment` are gone.
- Polygon offset in ShaderLab can't reference variables anymore (like `Offset [_Var1], [_Var2]`).
- Renamed `TRANSFER_EYEDEPTH/OUTPUT_EYEDEPTH` to `UNITY_TRANSFER_DEPTH/UNITY_OUTPUT_DEPTH`. They also work on a float2 in Unity 3.
- Removed special shader pass types: R2TPass, OffscreenPass.
- Removed `_Light2World0`, `_World2Light0` built-in shader matrices.
- Removed _SceneAmbient, _MultiModelAmbient, _MultiAmbient, _ModelAmbient, _MultiplyFog, _LightHackedDiffuse0, _ObjectCenterModelLightColor0 built-in shader vectors.
- Removed `_FirstPass` built-in shader float.
- Fog mode in shader files can't come from variable (like `Fog { Mode [_MyFogMode] }`). To use global fog mode, write `Fog { Mode Global }`.
- Removed `BlendColor` color from ShaderLab.
- Removed support for declaring texture matrix by-value in shader property.
- Removed support for "static" shader properties.
- Removed support for texture border color (`RenderTexture.SetBorderColor`).
- Removed `ColorMaterial Ambient, Diffuse, Specular` support (ColorMaterial AmbientAndDiffuse & Emission left). Support for the removed ones varied a lot depending on the platform causing confusion; and they didn't seem to be very useful anyway.
- Built-in `_CameraToWorld` and `_WorldToCamera` matrices now do what you'd expect them to do. Previously they only contained the rotation part, and camera-to-world was flipped on Y axis. Yeah, we don't know how that happened either :)
- Removed `Shader.ClearAll()`. Was deprecated since 2007, time to let it go.
- Vertex shaders are compiled to Shader Model 2.0 now (before was 1.1). If you want to compile to SM1.1, add `#pragma`

target 1.1 in the shader.

Page last updated: 2010-09-25

---

# SL-V3Conversion

Unity 3 has many new features and changes to its rendering system, and ShaderLab did update accordingly. Some advanced shaders that were used in Unity 2.x, especially the ones that used per-pixel lighting, will need update for Unity 3. If you have trouble updating them - just ask for our help!

For general graphics related Unity 3 upgrade details, see Rendering Upgrade Details.

When you open your Unity 2.x project in Unity 3.x, it will automatically upgrade your shader files as much as possible. The document below lists all the changes that were made to shaders, and what to do when you need manual shader upgrade.

### Per-pixel lit shaders

In Unity 2.x, writing shaders that were lit per-pixel was quite complicated. Those shaders would have multiple passes, with **LightMode** tags on each (usually **PixelOrNone**, **Vertex** and **Pixel**). With addition of Deferred Lighting in Unity 3.0 and changes in old forward rendering, we needed an easier, more robust and future proof way of writing shaders that interact with lighting. **All old per-pixel lit shaders need to be rewritten** to be Surface Shaders.

## Cg shader changes

#### Built-in "glstate" variable renames

In Unity 2.x, accessing some built-in variables (like model*view*projection matrix) was possible through built-in Cg names like `glstate.matrix.mvp`. However, that does not work on some platforms, so in Unity 3.0 we renamed those built-in variables. All these replacements will be done automatically when upgrading your project:

- `glstate.matrix.mvp` to UNITY_MATRIX_MVP
- `glstate.matrix.modelview[0]` to UNITY_MATRIX_MV
- `glstate.matrix.projection` to UNITY_MATRIX_P
- `glstate.matrix.transpose.modelview[0]` to UNITY_MATRIX_T_MV
- `glstate.matrix.invtrans.modelview[0]` to UNITY_MATRIX_IT_MV
- `glstate.matrix.texture[0]` to UNITY_MATRIX_TEXTURE0
- `glstate.matrix.texture[1]` to UNITY_MATRIX_TEXTURE1
- `glstate.matrix.texture[2]` to UNITY_MATRIX_TEXTURE2
- `glstate.matrix.texture[3]` to UNITY_MATRIX_TEXTURE3
- `glstate.lightmodel.ambient` to UNITY_LIGHTMODEL_AMBIENT
- `glstate.matrix.texture` to UNITY_MATRIX_TEXTURE

#### Semantics changes

Additionally, it is recommended to use `SV_POSITION` (instead of `POSITION`) semantic for position in vertex-to-fragment structures.

#### More strict error checking

Depending on platform, shaders might be compiled using a different compiler than Cg (e.g. HLSL on Windows) that has more strict error checking. Most common cases are:

- All vertex/fragment shader inputs and outputs need to have "semantics" assigned to them. Unity 2.x allowed to not assign any semantics (in which case some TEXCOORD would be used); in Unity 3.0 semantic is required.
- All shader output variables need to be written into. For example, if you have a `float4 color : COLOR` as your vertex shader output, you can't just write into `rgb` and leave alpha uninitialized.

## Other Changes

#### RECT textures are gone

In Unity 2.x, RenderTextures could be not power of two in size, so called "RECT" textures. They were designated by "RECT" texture type in shader properties and used as `samplerRECT`, `texRECT` and so on in Cg shaders. Texture coordinates for RECT textures were a special case in OpenGL: they were in pixels. In all other platforms, texture coordinates were just like for

any other texture: they went from 0.0 to 1.0 over the texture.

In Unity 3.0 we have decided to remove this OpenGL special case, and treat non power of two RenderTextures the same everywhere. It is recommended to replace `samplerRECT`, `texRECT` and similar uses with regular `sampler2D` and `tex2D`. Also, if you were doing any special pixel adressing for OpenGL case, you need to remove that from your shader, i.e. just keep the non-OpenGL part (look for `SHADER_API_D3D9` or `SHADER_API_OPENGL` macros in your shaders).

Page last updated: 2010-09-25

---

# Unity 4.x Activation - Overview

**What is the new Activation system?**

With our new Licensing System, we allow you, the user, to manage your Unity license independently. Contacting the Support Team when you need to switch machine is a thing of the past! The system allows instant, automated migration of your machine, with a single click. Please read our 'Managing your Unity 4.0 License' link for more information.

http://docs.unity3d.com/Documentation/Manual/ManagingyourUnity4xLicense.html

### If you're looking for step-by-step guides to Activation of Unity, please see the child pages.

## FAQ

**How many machines can I install my copy of Unity on?**

Every paid commercial Unity license allows a *single* person to use Unity on *two* machines that they have exclusive use of. Be it a Mac and a PC or your Home and Work machines. Educational licenses sold via Unity or any one of our resellers are only good for a single activation. The same goes for Trial licenses, unless otherwise stated.

The free version of Unity may not be licensed by a commercial entity with annual gross revenues (based on fiscal year) in excess of US$100,000, or by an educational, non-profit or government entity with an annual budget of over US$100,000.

If you are a Legal Entity, you may not combine files developed with the free version of Unity with any files developed by you (or by any third party) through the use of Unity Pro. Please see our EULA http://unity3d.com/company/legal/eula for further information regarding license usage.

**I need to use my license on another machine, but I get that message that my license has been 'Activated too many times'. What should I do?**

You�ll need to 'Return' your license. This enables you to return the license on the machine you no longer require, which in turn enables you to reactivate on a new machine. Please refer to the 'Managing your Unity 4.0 License' link at the top of the page, for more information.

**My account credentials aren�t recognised when logging in during the Activation process?**

Please ensure that your details are being entered correctly. Passwords ARE case sensitive, so ensure you�re typing exactly as you registered. You can reset your password using the link below:

https://accounts.unity3d.com/password/new

If you�re still having issues logging in, please contact 'support@unity3d.com'

**Can I use Unity 4.x with my 3.x Serial number?**

No, you can�t. In order to use Unity 4.x, you�ll need to upgrade to a 4.x license. You can do this Online, via our Web Store. https://store.unity3d.com/shop/

**I�m planning on replacing an item of hardware and/or my OS. What should I do?**

As with changing machine, you�ll need to 'Return' your license before making any hardware or OS changes to your machine. If you fail to �Return� the license, our server will see a request from another machine and inform you that you�ve reached your activation limit for the license. Please refer to the 'Managing your Unity 4.0 License' link at the top of the page, for more information regarding the return of a license.

**My machine died without me being able to 'Return' my license, what now?**

Please email 'support@unity3d.com' explaining your situation, including the details below.

```
 - The Serial number you were using on the machine.
 - The (local network) name of the machine that died
```

The Support Team will then be able to 'Return' your license manually.

**I have two licenses, each with an add-on I require, how do I activate them in unison on my machine?**

You can�t, unfortunately! A single license may only be used on one machine at any one time.

**Where is my Unity 4.x license file stored?**

- /Library/Application Support/Unity/Unity_v4.x.ulf (OS X)

- C:\ProgramData\Unity (Windows)

For any further assistance, please contact support@unity3d.com.

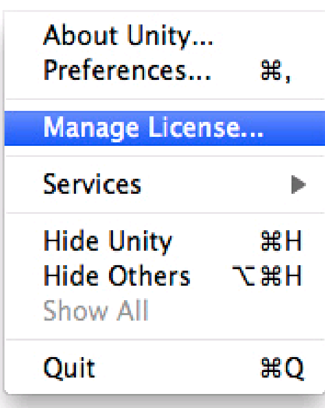
Page last updated: 2012-11-19

---

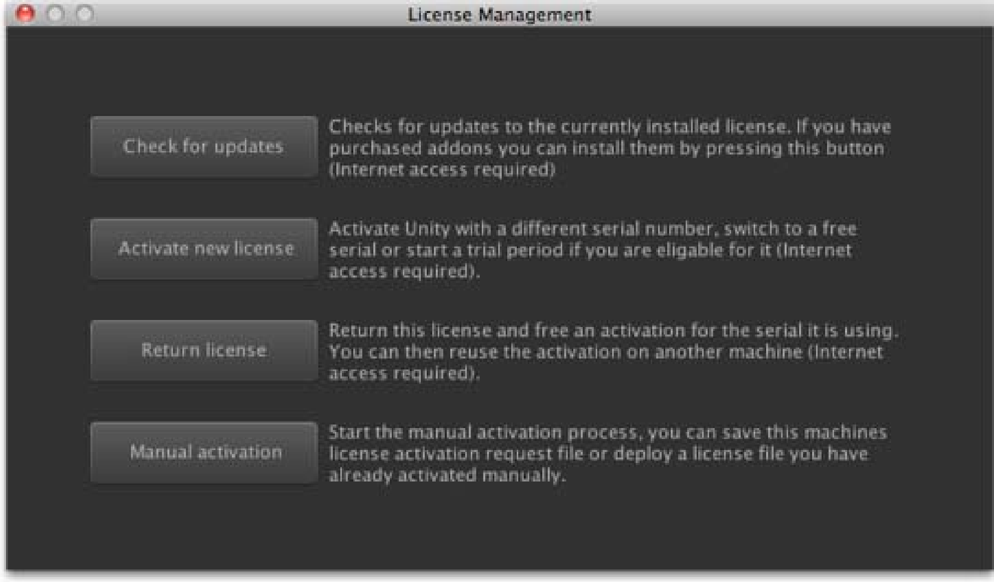
# Managing your Unity 4.x License

With Unity 4.0 you are now able to manage your license independently (no more contacting Support for migration to your shiny new machine). Below is a guide to how this new system works and performs.

You will notice a new option under the 'Unity' drop-down on your toolbar that reads 'Manage License�'. This is the unified place within the Editor for all your licensing needs.




Once you have clicked on the 'Manage License�' option you will be faced with the 'License Management' window. You then have four options (see image), explained below:




**'Check for updates'** cross-references the server, querying your Serial number for any changes that may have been made since you last activated. This is handy for updating your license to include new add-ons once purchased and added to your existing license via the Unity Store.

**'Activate a new license'** does what it says on the tin. This enables you to activate a new Serial number on the machine you�re using.

The **'Return license'** feature enables you to return the license on the machine in question, in return for a new activation that can be used on another machine. Once clicked the Editor will close and you will be able to activate your Serial number elsewhere. For more information on how many machines a single license enables use on, please see our EULA: http://unity3d.com/company/legal/eula.

**'Manual activation'** enables you to activate your copy of Unity offline. This is covered in more depth here: http://docs.unity3d.com/Documentation/Manual/ManualActivationGuide.html.

For any further assistance, please contact **support@unity3d.com**.

Page last updated: 2012-11-20

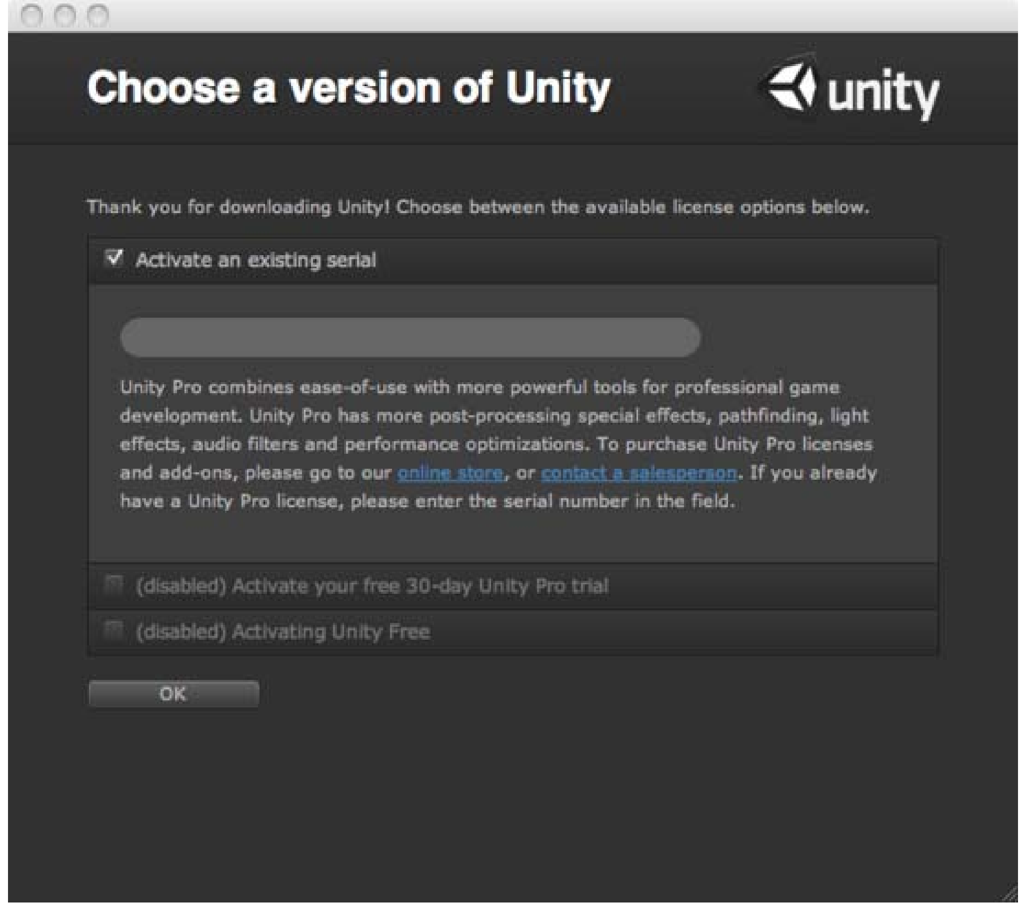
## Online Activation Guide

Online activation is the easiest and fastest way to get up and running with Unity. Below is a step-by-step guide on how to activate Unity online.

**1.** Download and install the Unity Editor. The latest version of Unity can be found at http://unity3d.com/unity/download/

**2.** Fire up the Editor from your Applications folder on OS X or the shortcut in the Start Menu on Windows.

**3.** You will be faced with a window titled 'Choose a version of Unity', you will then need to select the version of Unity you wish to activate by checking the tick box of the appropriate option and clicking 'OK' to proceed.

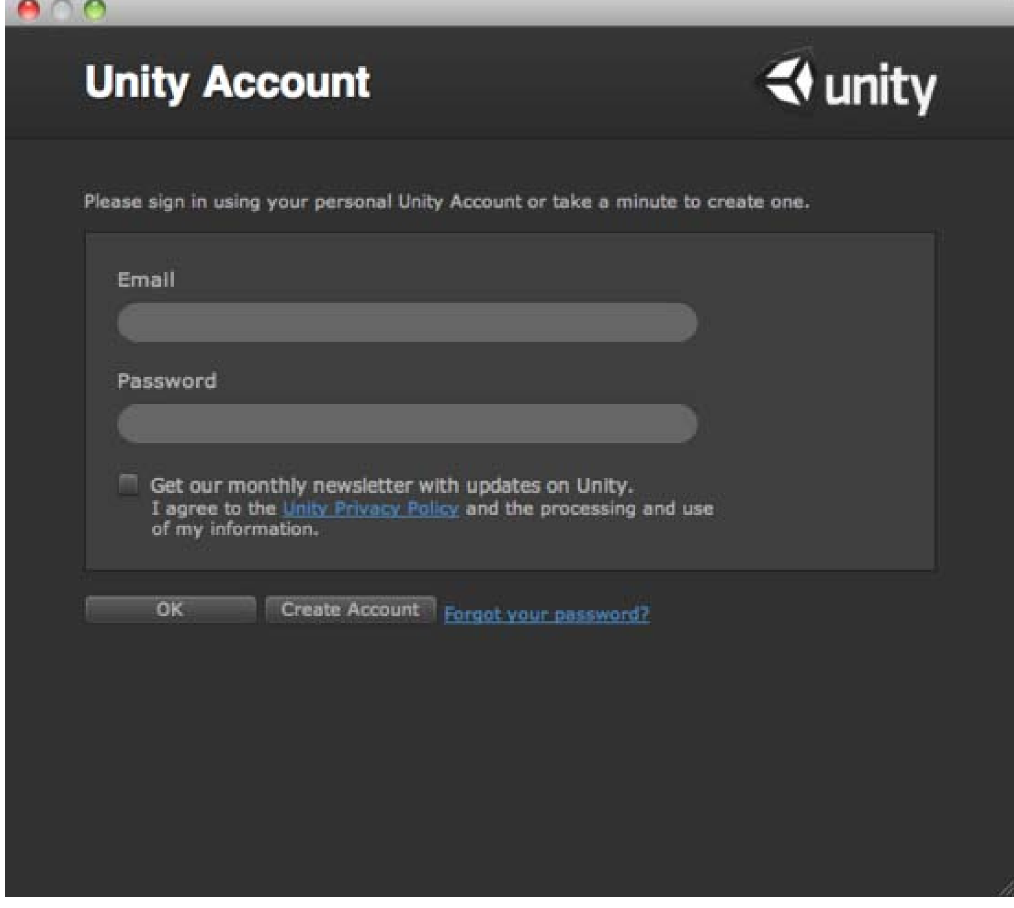



**a.** To activate an existing Unity 4.x Serial number generated by the Store or a member of our Sales Team, check the 'Activate an existing serial' box and enter the appropriate Serial number. Once the Serial number has been entered your license Type will be displayed on-screen.

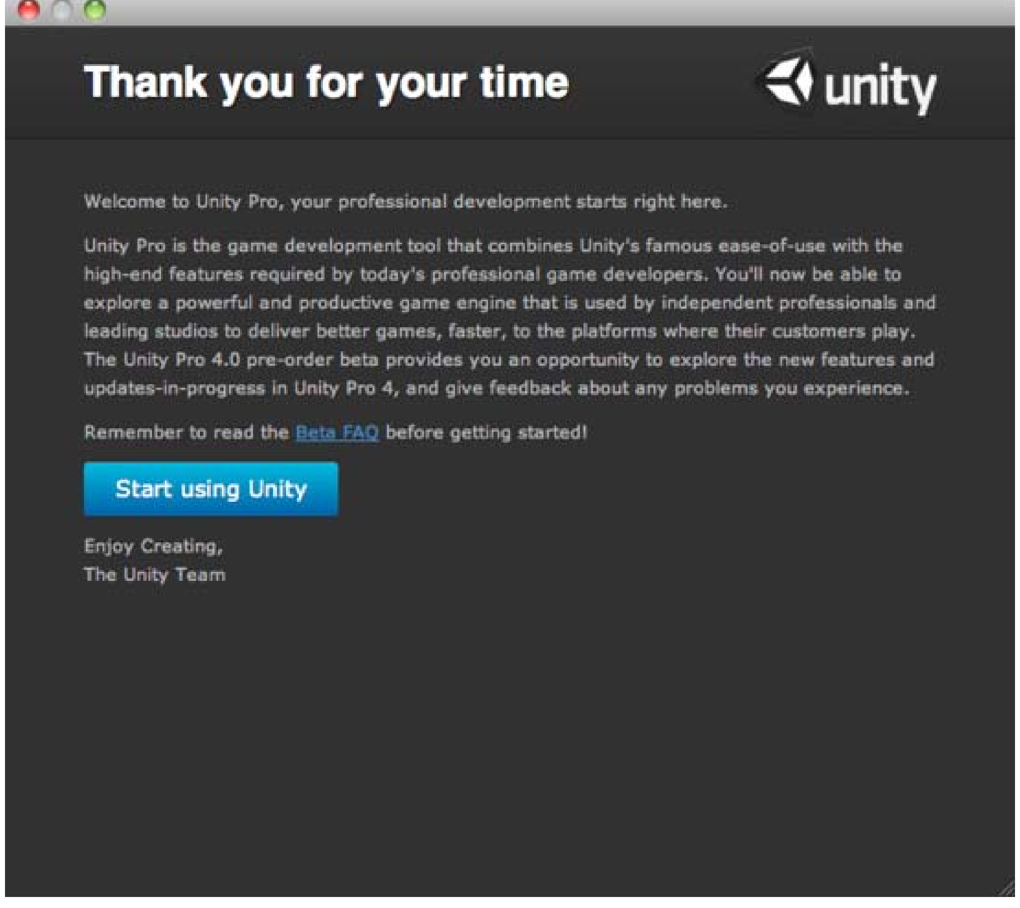
**b.** To Trial Unity Pro for 30 days Free-Of-Charge, check the 'Activate your free 30-day Unity Pro trial' box.

**c.** To activate the Free version of Unity, check the 'Activating Unity Free' box.

**4.** Next, you will encounter the 'Unity Account' window. Here you will need to enter your Unity Developer Network account credentials. (If you don�t have an existing account or have forgotten your password, simply click the respective 'Create account' and 'Forgot your password?' button and links. Follow the onscreen prompts to create or retrieve your account.) Once your credentials are entered you can proceed by clicking 'OK'.




**5.** 'Thank you for your time' you will now be able to proceed to the Unity Editor by clicking the 'Start using Unity' button.




**6.** You�re all done!

For any further assistance, please contact **support@unity3d.com**.

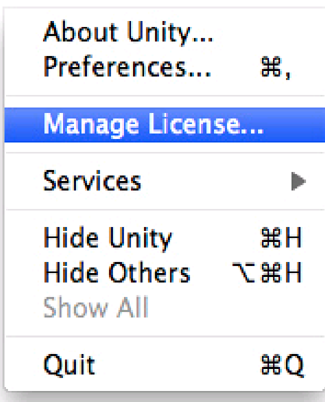
Page last updated: 2012-11-27

---

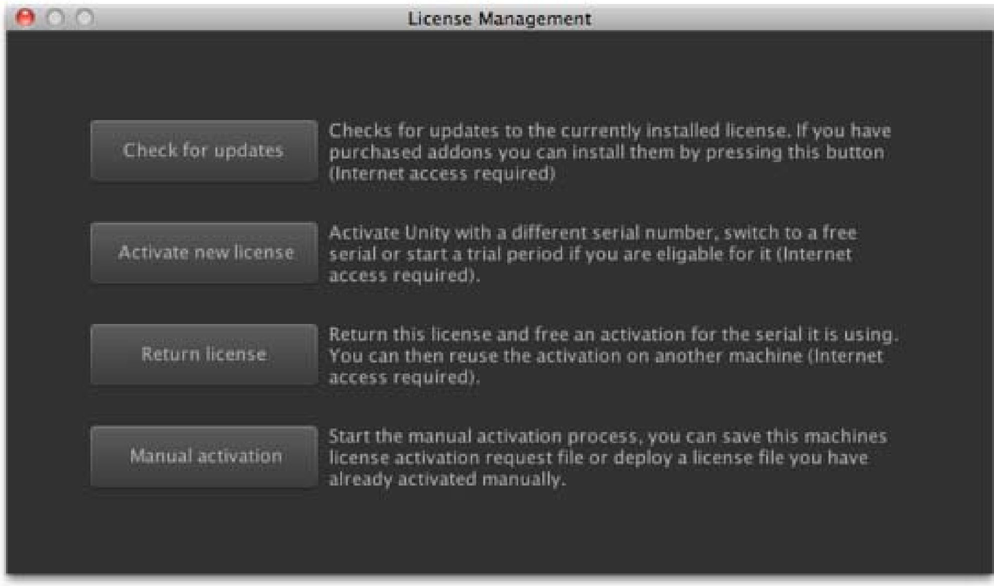
## Manual Activation Guide

With our new Licensing System, the Editor will automatically fall back to manual activation if Online Activation fails, or if you don�t have an internet connection. Please see the steps below for an outline on how to manually Activate Unity 4.0.

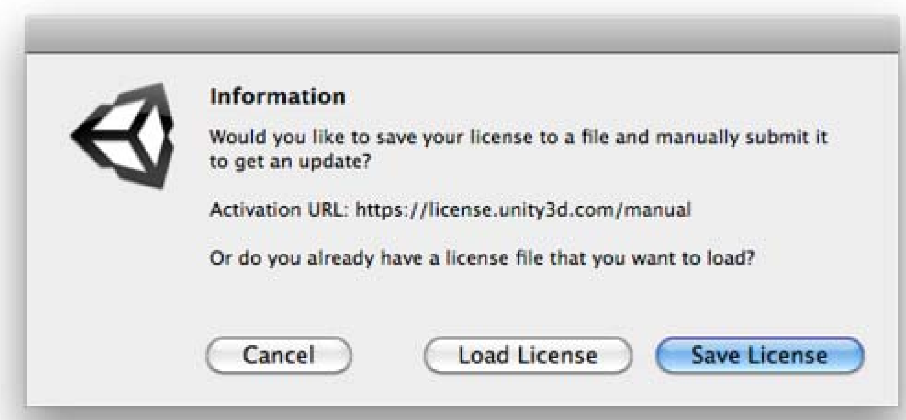
**1.** As above, Unity will fall back to Manual Activation, should the Online Activation fail. However, you can manually prompt Unity to start the Manual Activation procedure by navigating to 'Unity>Manage License�' within the Editor.




**2.** In the 'License Management' window, hit the 'Manual activation' button.




**3.** You should now be faced with a dialog displaying three buttons:



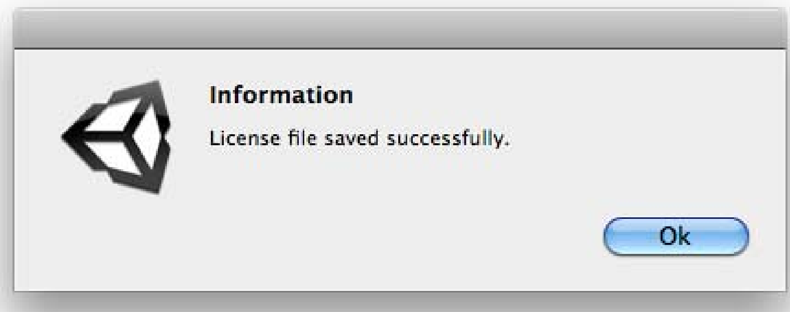

**a.** 'Cancel' will take you back to the 'License Management' window.

**b.** 'Save License' will generate you a license file specific to your machine, based on your HWID. This file can be saved in any location on your physical machine.

**c.** 'Load License' will load the activation file generated by the Manual Activation process.

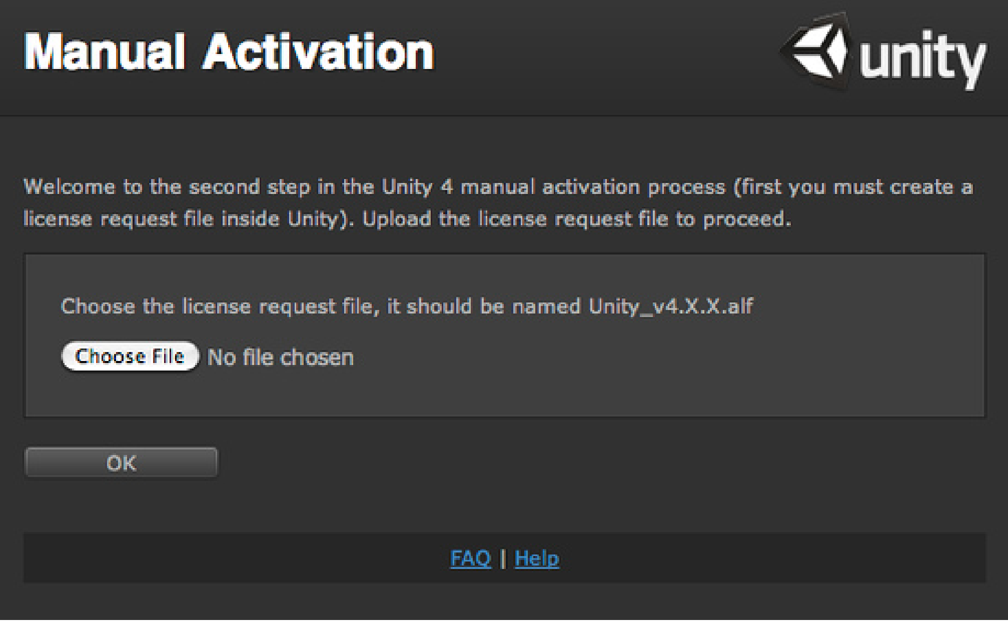
**4.** You will need to generate a license file; in order to do this, click the �**Save License**� button. Once clicked you will be faced with the window 'Save license information for offline activation'. Here you can select a directory on your machine to save the file.

**5.** Once saved, you will receive a message stating that 'License file saved successfully'. Click 'Ok' to proceed.




**6.** Now, you�ll need to minimise the Editor and navigate over to https://license.unity3d.com/manual within your Browser (if on a machine without an internet connection, you�ll need to copy the file to a machine that does and proceed there).

**7.** You now need to navigate to the file you generated in Step 4, uploading it in the appropriate field. When your file has been selected, click 'OK' to proceed.




**8.** Nearly done! You should have received a file in return, as with Step 4, save this to your machine in a directory of your choice.

**9.** Moving back into Unity, you can now select the 'Load License' button. Again, this will open up your directories within your hard drive. Now, select the file that you just saved via the Web form and click 'OK'.

**10.** Voila, you've just completed the Manual Activation process.

For any further assistance, please contact **support@unity3d.com**.

Page last updated: 2012-11-27

---

# Game Code How-to

- How to make a simple first person walkthrough

Page last updated: 2007-11-16

---

# HOWTO-First Person Walkthrough

Here's how you can make a simple first person walk-through with your own artwork:

1. Import your level. See here on how to import geometry from your art package into Unity.
2. Select the imported model file and enable **Generate Colliders** in the **Import Settings** in the **Inspector**.
3. Locate the **Standard Assets->Prefabs->First Person Controller** in the **Project View** and drag it into the **Scene View**.
4. Make sure that the scale of your level is correct. The First Person Controller is exactly 2 meters high, so if your level doesn't fit the size of the controller, you should adjust the scale of the level size within your modeling application. Getting scale right is critical for physical simulation, and other reasons documented at the bottom of this page. Using the wrong scale can make objects feel like they are floating or too heavy. If you can't change the scale in your modeling app, you can change the scale in the **Import Settings...** of the model file.
5. Move the First Person Controller to be at the start location using the **Transform** handles. It is critical that the first person controller does not intersect any level geometry, when starting the game (otherwise it will be stuck!).
6. Remove the default camera "Main Camera" in the hierarchy view. The First person controller already has its own camera.

7. Hit **Play** to walk around in your own level.

Page last updated: 2009-03-13

---

## Graphics how-tos

The following is a list of common graphics-related questions in Unity and how to accomplish them.

There is an excellent tutorial for creating textures, including color, bump, specular, and reflection mapping here.

- How do I Import Alpha Textures?
- How do I Use Normal Maps?
- How do I use Detail Textures?
- How do I Make a Cubemap Texture?
- How do I Make a Skybox?
- How do I make a Mesh Particle Emitter? (Legacy Particle System)
- How do I make a Splash Screen?
- How do I make a Spot Light Cookie?
- How do I fix the rotation of an imported model?
- How do I use Water?

Page last updated: 2007-11-16

---

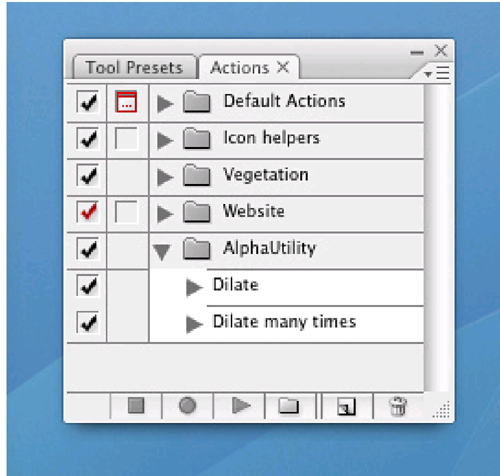
## HOWTO-alphamaps

Unity uses straight **alpha blending**. Hence, you need to expand the color layers... The alpha channel in Unity will be read from the first alpha channel in the Photoshop file.

### Setting Up

Before doing this, install these alpha utility photoshop actions: AlphaUtility.atn.zip

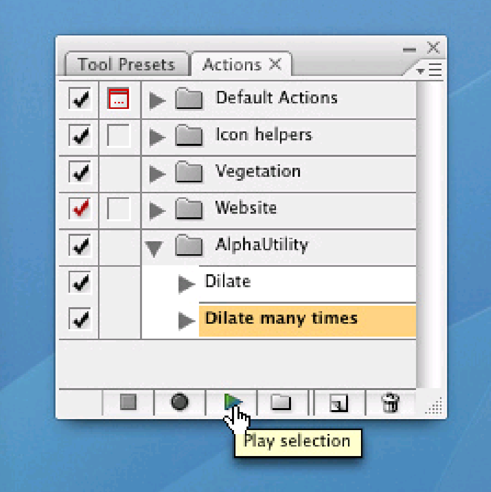
After installing, your Action Palette should contain a folder called AlphaUtility:




### Getting Alpha Right

Let's assume you have your alpha texture on a transparent layer inside photoshop. Something like this:

1. Duplicate the layer
2. Select the lowest layer. This will be source for the dilation of the background.
3. Select **Layer->Matting->Defringe** and apply with the default properties
4. Run the "Dilate Many" action a couple of times. This will expand the background into a new layer.




5. Select all the dilation layers and merge them with **Command-E**

6. Create a solid color layer at the bottom of your image stack. This should match the general color of your document (in this case, greenish). Note that without this layer Unity will take alpha from merged transparency of all layers.

Now we need to copy the transparency into the alpha layer.

1. Set the selection to be the contents of your main layer by Command-clicking on it in the Layer Palette.
2. Switch to the channels palette.
3. Create a new channel from the transparency.

Save your PSD file - you are now ready to go.

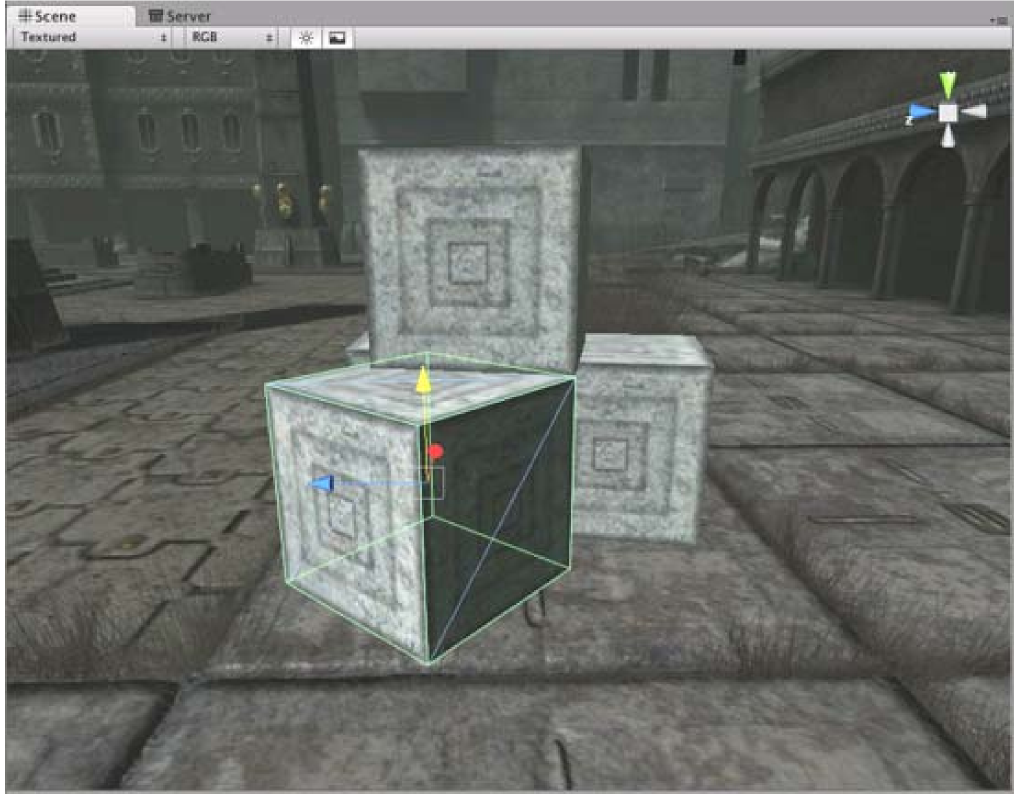
## Extra

Note that if your image contains transparency (after merging layers), then Unity will take alpha from merged transparency of all layers and it will ignore Alpha masks. A workaround for that is to create a layer with solid color as described in Item 6 on "Getting Alpha Right"

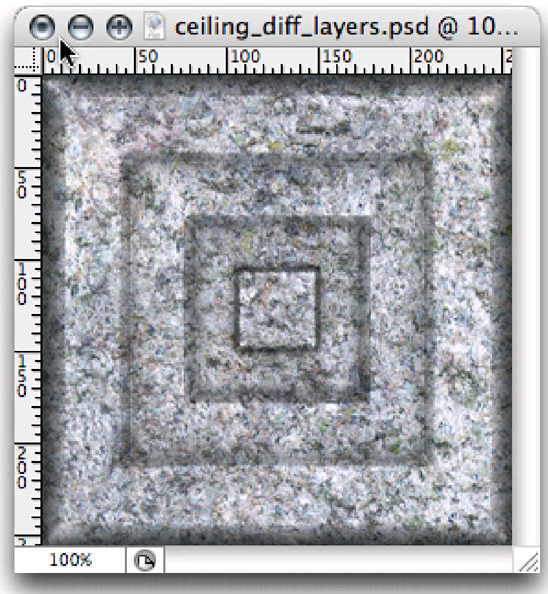
Page last updated: 2012-10-10

## HOWTO-Normalmap

**Normal maps** are grayscale images that you use as a height map on your objects in order to give an appearance of raised or recessed surfaces. Assuming you have a model that looks like this:



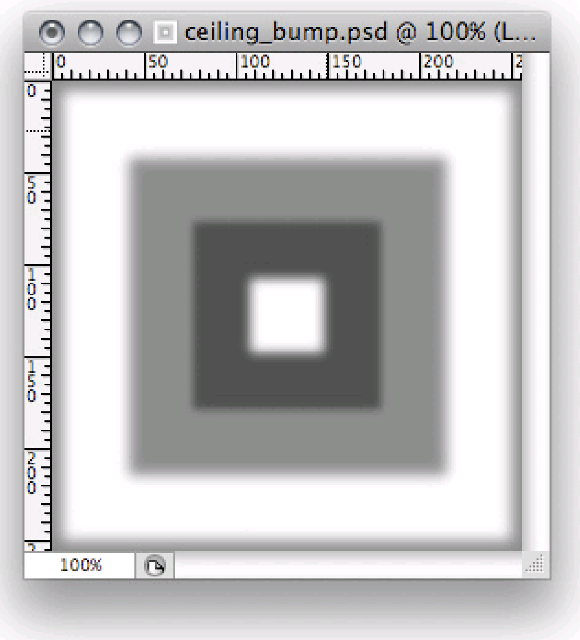
*The 3D Model*



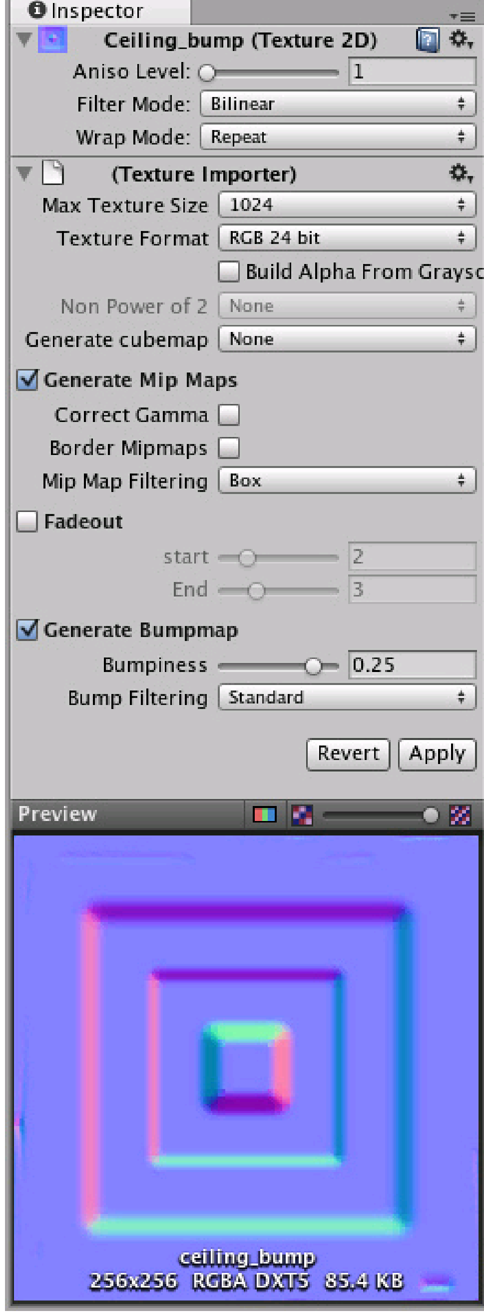

*The Texture*

We want to make the light parts of the object appear raised.

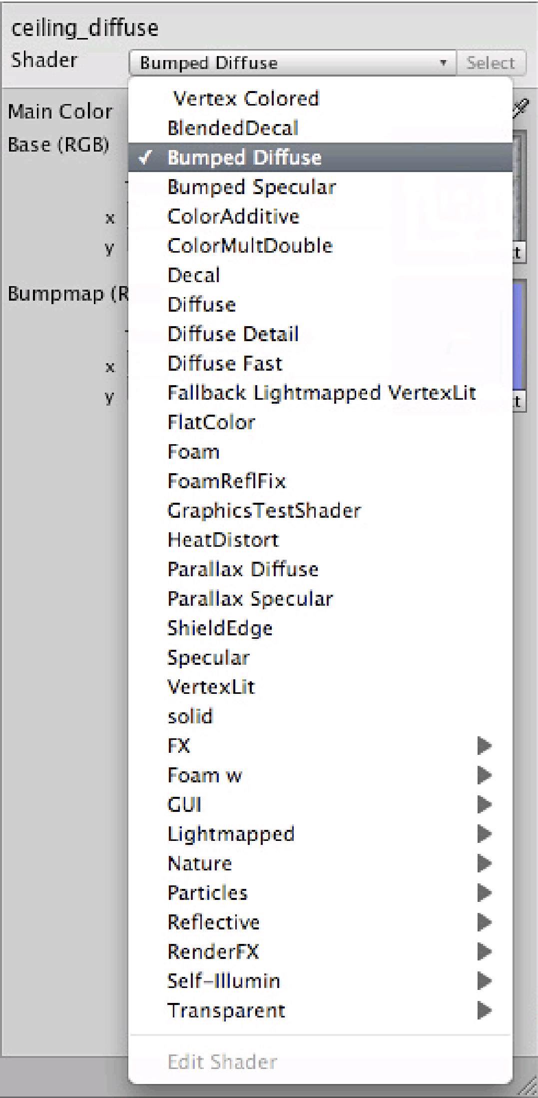
1. Draw a grayscale height map of your texture in Photoshop. White is high, black is low. Something like this:



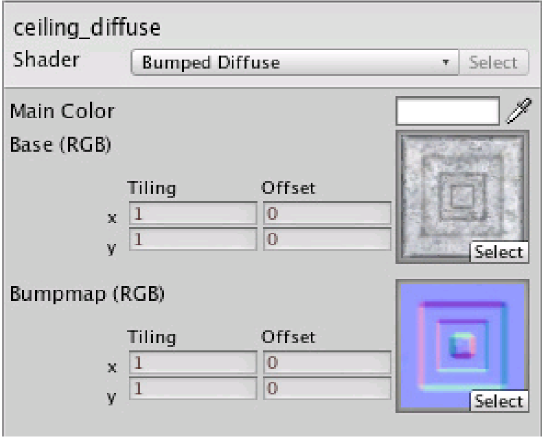

2. Save the image next to your main texture.
3. In Unity, select the image and select the 24 bit RGB format and enable **Generate Normal Map** in the **Import Settings** in the **Inspector**:




1. In the **Material Inspector** of your model, select 'Bumped Diffuse' from the Shader drop-down:




2. Drag your texture from the Project window to the 'Normalmap' texture slot:




Your object now has a normal map applied:

### Hints

- To make the bumps more noticable, either use the Bumpyness slider in the Texture Import Settings or blur the texture in Photoshop. Experiment with both approaches to get a feel for it.

Page last updated: 2010-09-10

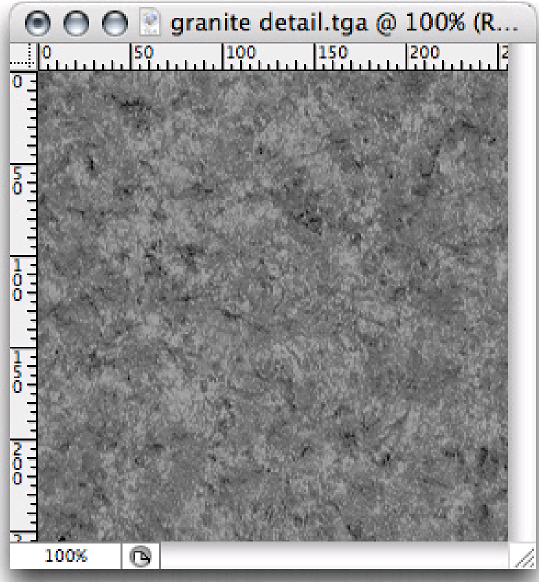
---

## HOWTO-UseDetailTexture

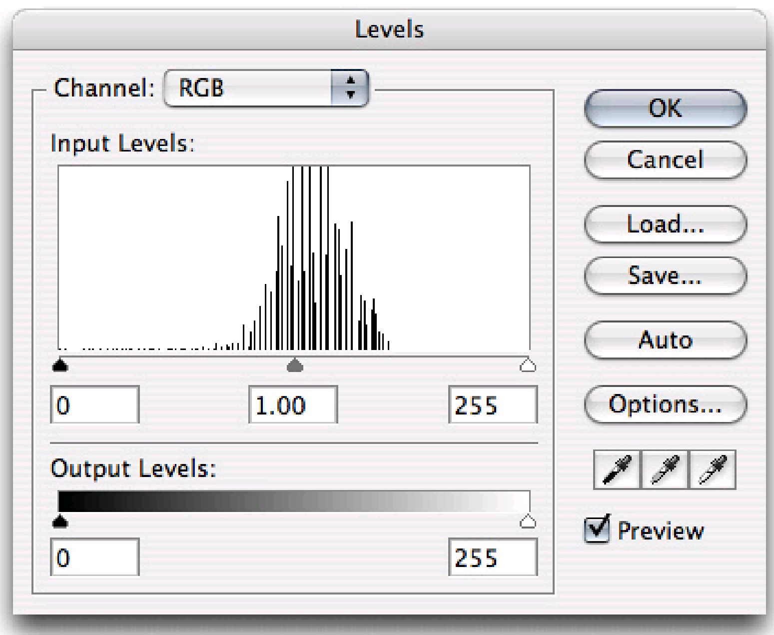
A **Detail texture** is a small, fine pattern which is faded in as you approach a surface, for example wood grain, imperfections in stone, or earthly details on a terrain. They are explicitly used with the Diffuse Detail shader.

Detail textures must tile in all directions. Color values from 0-127 makes the object it's applied to darker, 128 doesn't change anything, and lighter colors make the object lighter. It's very important that the image is centered around 128 - otherwise the object it's applied to will get lighter or darker as you approach.

1. Draw or find a grayscale image of the detail texture.




*The Detail Texture*

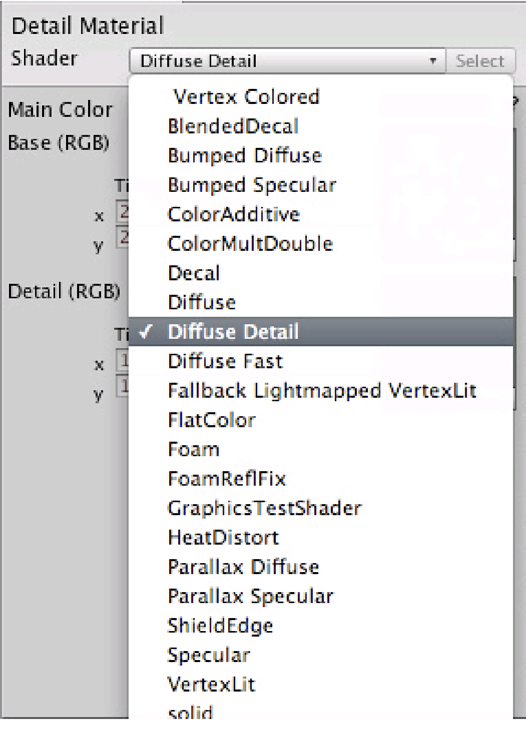



*The Levels*

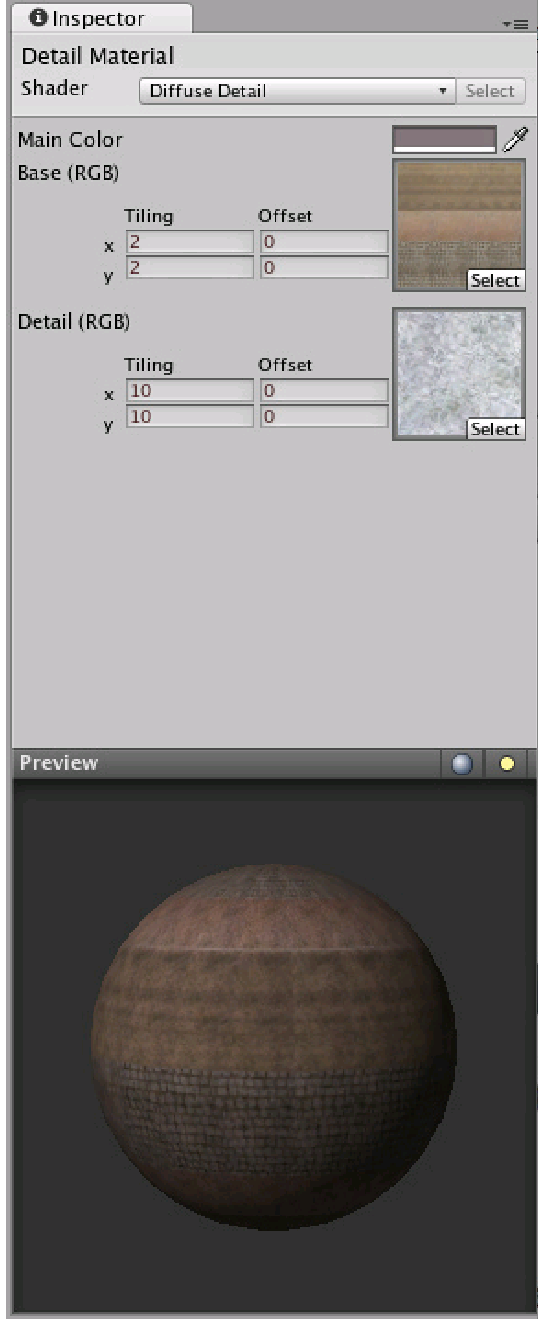
2. Save the image next to your main texture.
3. In Unity, select the image and under "Generate Mip Maps", enable **Fades Out** and set the sliders to something like this in the **Import Settings** in the **Inspector**.
4. The top slider determines how small the texture should before before beginning to fade out, and the bottom determines how far away it is before the detail texture completely disapear.

s.

5. In the **Material Inspector** on the right, select **Diffuse Detail** from the Shader drop-down:




6. Drag your texture from the **Project View** to the **Detail** texture slot.
7. Set the **Tiling** values to a high value




Page last updated: 2009-02-16

---

## HOWTO-MakeCubemap

**Cubemaps** are used by the Reflective built-in shaders. To build one, you either create six 2D textures and create a new Cubemap asset, or build the Cubemap from a single square texture. More details are in the Cubemap Texture documentation page.

Static and dynamic cubemap reflections can also be rendered from scripts. Code example in Camera.RenderToCubemap page contains a simple wizard script for rendering cubemaps straight from the editor.

Page last updated: 2010-09-10

---

## HOWTO-UseSkybox

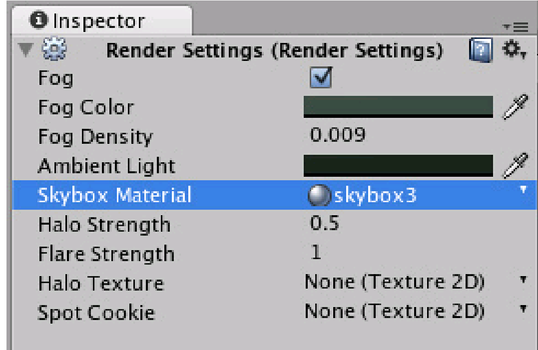
A **Skybox** is a 6-sided cube that is drawn behind all graphics in the game. Here are the steps to create one:

1. Make 6 textures that correspond to each of the 6 sides of the skybox and put them into your project's **Assets** folder.
2. For each texture you need to change the wrap mode from **Repeat** to **Clamp**. If you don't do this colors on the edges will not match up:

3. Create a new **Material** by choosing **Assets->Create->Material** from the menu bar.
4. Select the shader drop-down in the top of the **Inspector**, choose **RenderFX->Skybox**.
5. Assign the 6 textures to each texture slot in the material. You can do this by dragging each texture from the **Project View** onto the corresponding slots.

To Assign the skybox to the scene you're working on:

1. Choose **Edit->Render Settings** from the menu bar.
2. Drag the new Skybox Material to the **Skybox Material** slot in the Inspector.




Note that Standard Assets package contains several ready-to-use skyboxes - this is the quickest way to get started!

Page last updated: 2007-11-16

---

# HOWTO-MeshParticleEmitter

**Mesh Particle Emitters** are generally used when you need high control over where to emit particles.

For example, when you want to create a flaming sword:

1. Drag a mesh into the scene.
2. Remove the **Mesh Renderer** by right-clicking on the **Mesh Renderer's Inspector** title bar and choose **Remove Component**.
3. Choose **Mesh Particle Emitter** from the **Component->Effects->Legacy Particles** menu.
4. Choose **Particle Animator** from the **Component->Effects->Legacy Particles** menu.
5. Choose **Particle Renderer** from the **Component->Effects->Legacy Particles** menu.

You should now see particles emitting from the mesh.

Play around with the values in the Mesh Particle Emitter.

Especially enable **Interpolate Triangles** in the Mesh Particle Emitter Inspector and set **Min Normal Velocity** and **Max Normal Velocity** to 1.

To customize the look of the particles that are emitted:

1. Choose **Assets->Create->Material** from the menu bar.
2. In the Material Inspector, select **Particles->Additive** from the shader drop-down.
3. Drag & drop a texture from the **Project view** onto the texture slot in the Material Inspector.
4. Drag the Material from the Project View onto the Particle System in the **Scene View**.

You should now see textured particles emitting from the mesh.

### See Also

- Mesh Particle Emitter Component Reference page

Page last updated: 2012-01-17

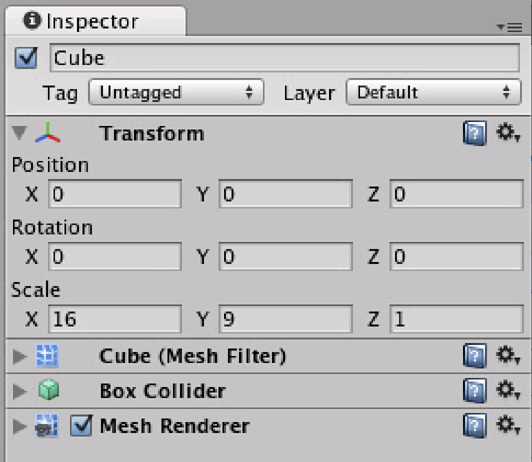
---

# HOWTO-SplashScreen

## Desktop

Here's how to do a splash screen or any other type of full-screen image in Unity. This method works for multiple resolutions and aspect ratios.

1. First you need a big texture. Ideally textures should be power of two in size. You might for example use 1024x512 as this fits most screens.
2. Make a box using the **GameObject->Create Other->Cube** menubar item.
3. Scale it to be in 16:9 format by entering 16 and 9 as the first two value in the Scale:




4. Drag the texture onto the cube and make the **Camera** point at it. Place the camera at such a distance so that the cube is still visible on a 16:9 aspect ratio. Use the **Aspect Ratio Selector** in the **Scene View** menu bar to see the end result.

## iOS

Customising IOS device Splash Screens

## Android

Customising Android device Splash Screens

Page last updated: 2011-02-22

---

# HOWTO-LightCookie

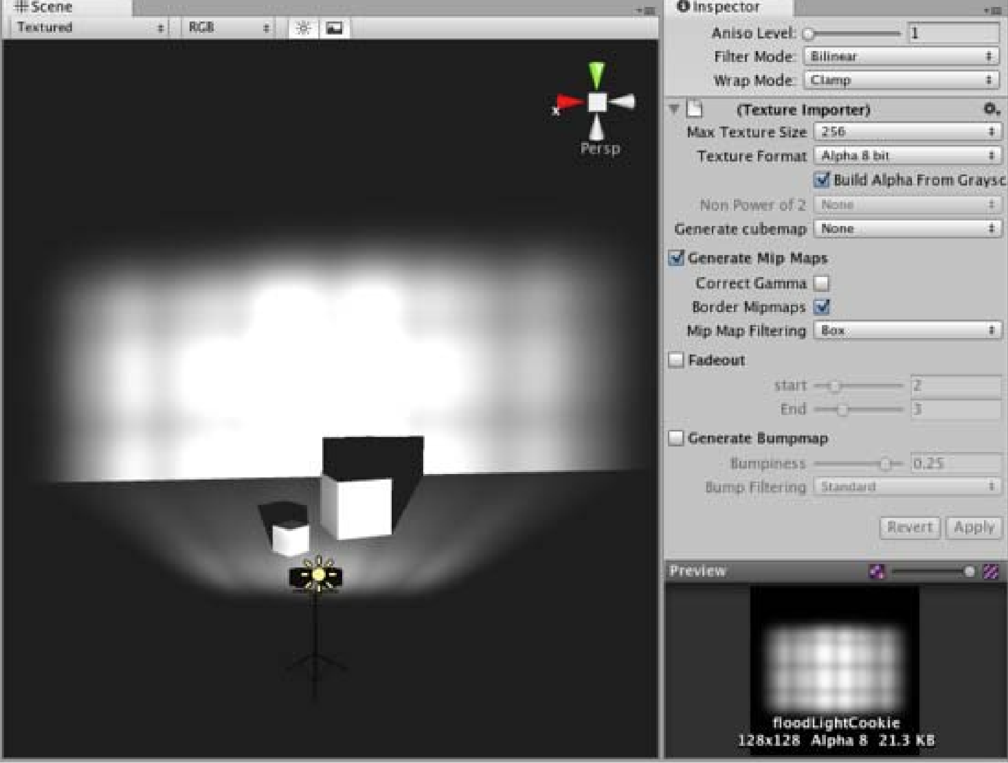
Unity ships with a few **Light Cookies** in the Standard Assets. When the Standard Assets are imported to your project, they can be found in **Standard Assets->Light Cookies**. This page shows you how to create your own.

A great way to add a lot of visual detail to your scenes is to use cookies - grayscale textures you use to control the precise look of in-game lighting. This is fantastic for making moving clouds and giving an impression of dense foilage. The Light Component Reference page has more info on all this, but the main thing is that for textures to be usable for cookies, the following properties need to be set:

To create a light cookie for a spot light:

1. Paint a cookie texture in Photoshop. The image should be greyscale. White pixels means full lighting intensity, black pixels mean no lighting. The borders of the texture need to be completely black, otherwise the light will appear to leak outside of the spotlight.
2. In the **Texture Inspector** change the **Repeat** Wrap mode to **Clamp**
3. Select the Texture and edit the following **Import Settings** in the **Inspector**.

4. Enable **Border Mipmaps**
5. Enable **Build Alpha From Grayscale** (this way you can make a grayscale cookie and Unity converts it to an alpha map automatically)
6. Set the Texture Format to **Alpha 8 Bit**




Page last updated: 2009-02-16

---

## HOWTO-FixZAxisIsUp

Some 3D art packages export their models so that the z-axis faces upward. Most of the standard scripts in Unity assume that the y-axis represents **up** in your 3D world. It is usually easier to fix the rotation in Unity than to modify the scripts to make things fit.

*Your model with z-axis points upwards*

If at all possible it is recommended that you fix the model in your 3D modelling application to have the y-axis face upwards before exporting.

If this is not possible, you can fix it in Unity by adding an extra parent transform:

1. Create an empty **GameObject** using the **GameObject->Create Empty** menu
2. Position the new GameObject so that it is at the center of your mesh or whichever point you want your object to rotate around.
3. Drag the mesh onto the empty GameObject

You have now made your mesh a **Child** of an empty GameObject with the correct orientation. Whenever writing scripts that make use of the y-axis as up, attach them to the **Parent** empty GameObject.

*The model with an extra empty transform*

Page last updated: 2007-11-16

---

## HOWTO-Water

**Note:** The content on this page applies to the desktop editor mode only.

Unity includes several water prefabs (including needed shaders, scripts and art assets) within the Standard Assets and Pro Standard Assets packages. Unity includes a basic water, while Unity Pro includes water with real-time reflections and refractions, and in both cases those are provided as separate daylight and nighttime water prefabs.

*Reflective daylight water (Unity Pro)*

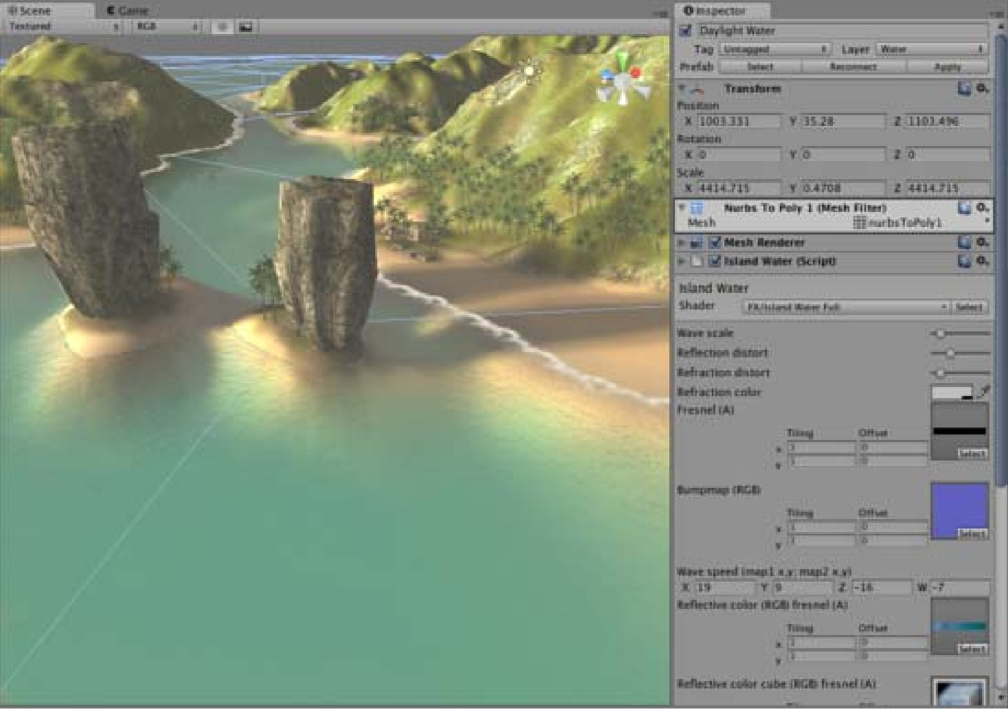
*Reflective/Refractive daylight water (Unity Pro)*

### Water setup

In most cases you just need to place one of the existing Prefabs into your scene (make sure to have the Standard Assets installed):

- Unity has **Daylight Simple Water** and **Nighttime Simple Water** in **Standard Assets->Water**.
- Unity Pro has **Daylight Water** and **Nighttime Water** in **Pro Standard Assets->Water** (but it needs some assets from **Standard Assets->Water** as well). Water mode (Simple, Reflective, Refractive) can be set in the Inspector.

The prefab uses an oval-shaped mesh for the water. If you need to use a different Mesh the easiest way is to simply change it in the **Mesh Filter** of the water object:




## Creating water from scratch (Advanced)

The simple water in Unity requires attaching a script to a plane-like mesh and using the water shader:

1. Have mesh for the water. This should be a flat mesh, oriented horizontally. UV coordinates are not required. The water GameObject should use the Water **layer**, which you can set in the **Inspector**.
2. Attach **WaterSimple** script (from **Standard Assets/Water/Sources**) to the object.
3. Use **FX/Water (simple)** shader in the material, or tweak one of provided water materials (**Daylight Simple Water** or **Nighttime Simple Water**).

The reflective/refractive water in Unity Pro requires similar steps to set up from scratch:

1. Have mesh for the water. This should be a flat mesh, oriented horizontally. UV coordinates are not required. The water GameObject should use the Water **layer**, which you can set in the **Inspector**.
2. Attach **Water** script (from **Pro Standard Assets/Water/Sources**) to the object.
   - Water rendering mode can be set in the Inspector: Simple, Reflective or Refractive.
3. Use **FX/Water** shader in the material, or tweak one of provided water materials (**Daylight Water** or **Nighttime Water**).

## Properties in water materials

These properties are used in Reflective & Refractive water shader. Most of them are used in simple water shader as well.

| | |
|---|---|
| **Wave scale** | Scaling of waves normal map. The smaller the value, the larger water waves. |
| **Reflection/refraction distort** | How much reflection/refraction is distorted by the waves normal map. |
| **Refraction color** | Additional tint for refraction. |
| **Environment reflection/refraction** | Render textures for real-time reflection and refraction. |
| **Normalmap** | Defines the shape of the waves. The final waves are produced by combining two these normal maps, each scrolling at different direction, scale and speed. The second normal map is half as large as the first one. |
| **Wave speed** | Scrolling speed for first normal map (1st and 2nd numbers) and the second normal map (3rd and 4th numbers). |
| **Fresnel** | A texture with alpha channel controlling the Fresnel efffect - how much reflection vs. refraction is visible, based on viewing angle. |

The rest of properties are not used by Reflective & Refractive shader by itself, but need to be set up in case user's video card does not suppor it and must fallback to the simpler shader:

| | |
|---|---|
| **Reflective color/cube and fresnel** | A texture that defines water color (RGB) and Fresnel effect (A) based on viewing angle. |
| **Horizon color** | The color of the water at horizon. (Used only in the simple water shader) |
| **Fallback texture** | Texture used to represent the water on really old video cards, if none of better looking shaders can't run on it. |

### Hardware support

- Reflective + Refractive water works on graphics cards with pixel shader 2.0 support (GeForce FX and up, Radeon 9500 and up, Intel 9xx). On older cards, Reflective water is used.
- Reflective water works on graphics cards with pixel shader 1.4 support (GeForce FX and up, Radeon 8500 and up, Intel 9xx). On older cards, Simple water is used.
- Simple water works about everywhere, with various levels of detail depending on hardware capabilities.

Page last updated: 2010-09-10

---

# HOWTO-exportFBX

Unity supports FBX files which can be generated from many popular 3D applications. Use these guidelines to help ensure the most best results.

### Select > Prepare > Check Settings > Export > Verify > Import

**What do you want to export? - be aware of export scope e.g. meshes, cameras, lights, animation rigs, etc. -**

- Applications often let you export ***selected objects*** or a ***whole scene***
- Make sure you are exporting only the objects you want to use from your scene by either exporting selected, or removing unwanted data from your scene.
- Good working practice often means keeping a working file with all lights, guides, control rigs etc. but only export the data you need with *export selected,* an export preset or even a custom scene exporter.

**What do you need to include? - prepare your assets:**

- Meshes - Remove construction history, Nurbs, Nurms, Subdiv surfaces must be converted to polygons - e.g. triangulate or quadrangulate
- Animation - Select the correct rig, check frame rate, animation length etc.
- Textures - Make sure your textures are sourced already from your Unity project or copied into a folder called \textures in your project
- Smoothing - Check if you want smoothing groups and/or smooth mesh

**How do I include those elements? - check the FBX export settings**

- Be aware of your settings in the export dialogue so that you know what to expect and can match up the fbx settings In Unity - see figs 1, 2 & 3 below
- Nodes, markers and their transforms can be exporte
- Cameras and Lights are not currently imported in to Unity

**Which version of FBX are you using? if in doubt use 2012.2**

- Autodesk update their FBX installer regularly and it can provide different results with different versions of their own software and other 3rd party 3D apps.
- ***See Advanced Options > FBX file format***

**Will it work? - Verify your export**

- Check your file size - do a sanity check on the file size (e.g. >10kb?)
- Re-import your FBX into a new scene in the 3D package you use to generate it - is it what you expected?

**Import!**

- Import into Unity
- Check FBX import settings in inspector : texures, animations, smoothing, etc.

See below for Maya FBX dialogue example:

Fig 1 General, Geometry & Animation

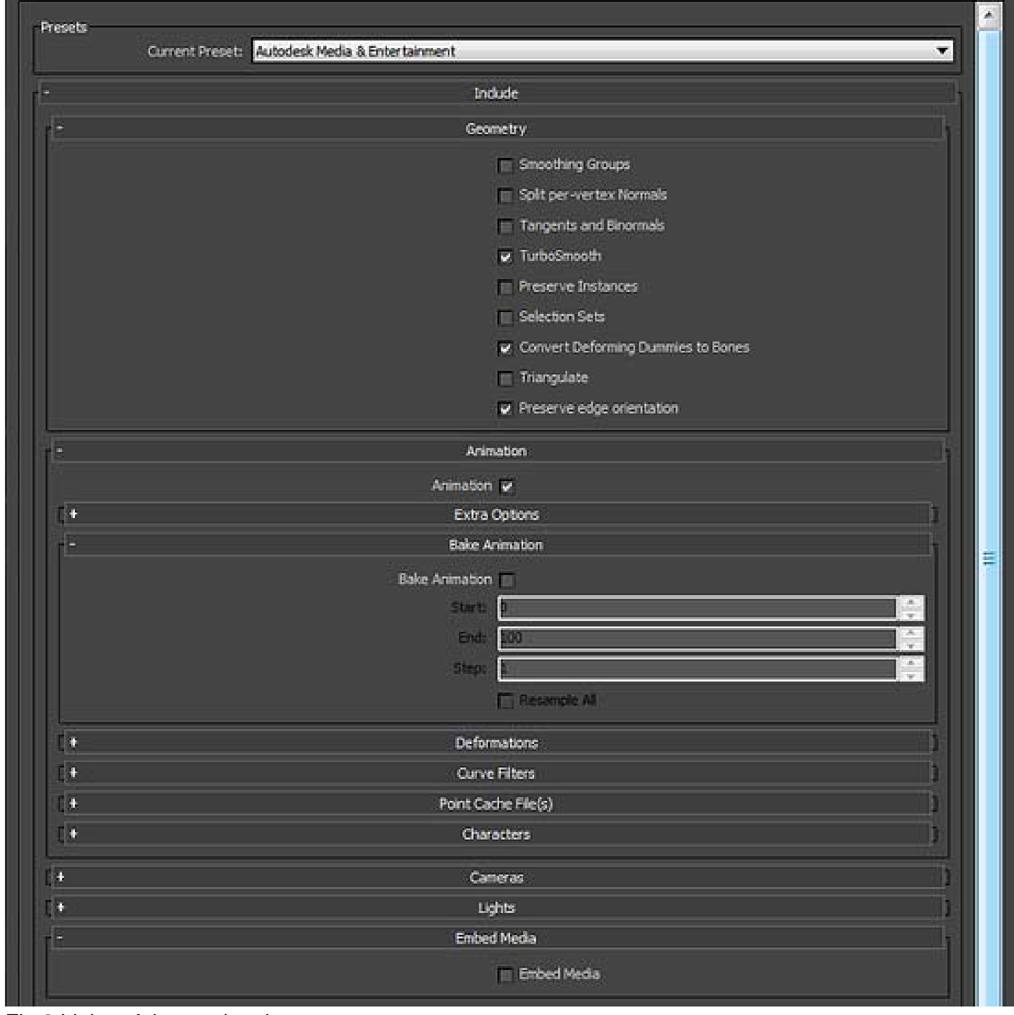
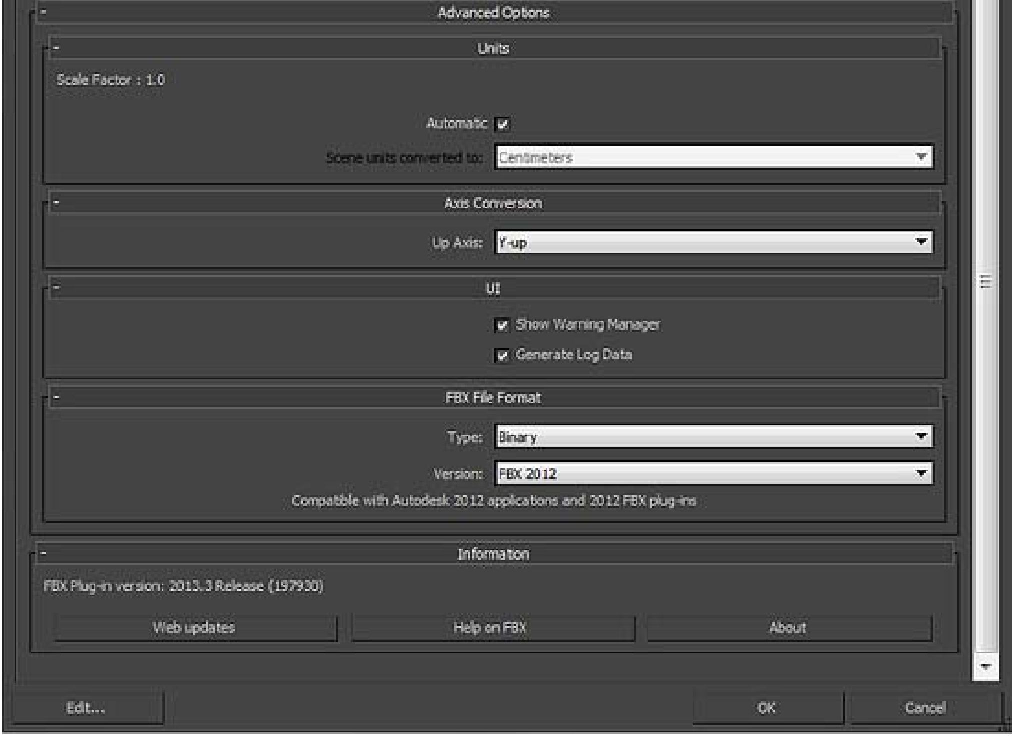



Fig 2 Lights, Advanced options




Page last updated: 2012-11-16

## HOWTO-ArtAssetBestPracticeGuide

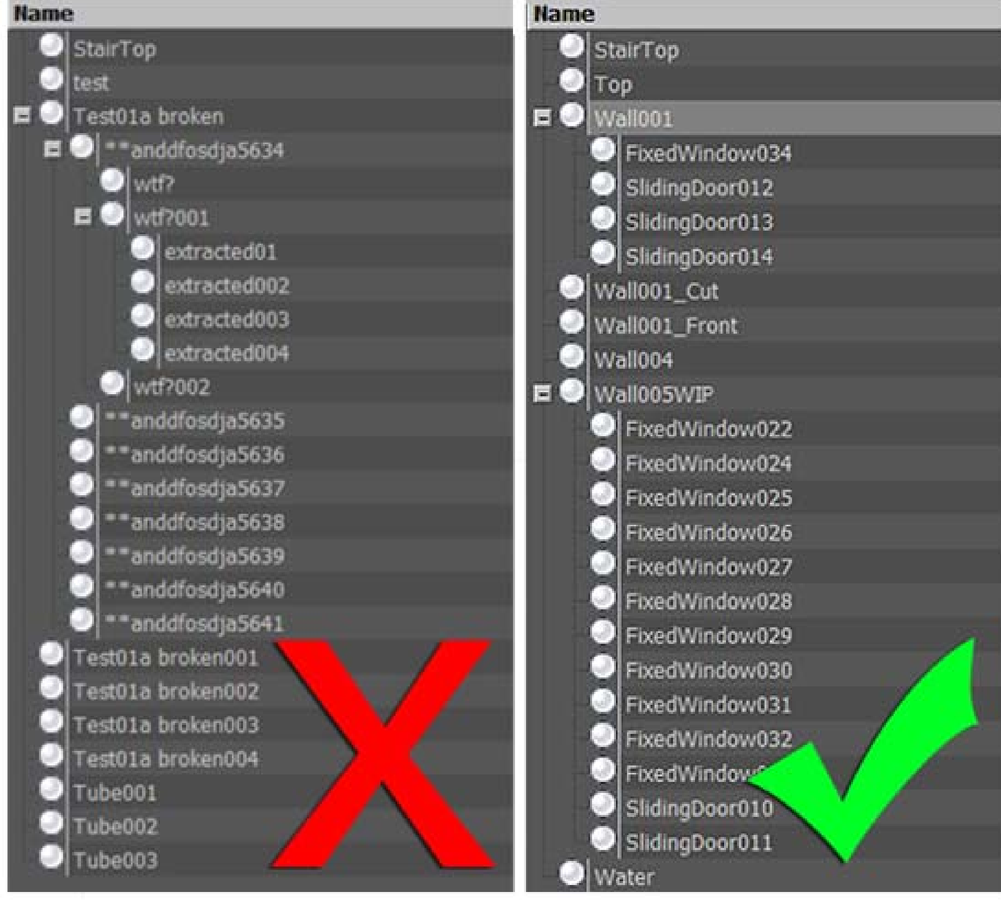
Unity supports textured 3D models from a variety of programs or sources. This short guide has been put together by games artists with developers at Unity, to help you create assets that work better and more efficiently in your Unity project.

### Scale & Units

- Set your system and project units for your software to work consistently with Unity e.g. Metric.
  - Working to scale can be important for both lighting and physics simulation.
  - Be aware that, for example, the Max system unit default is inches and Maya is centimetres.
  - Unity has different scaling for FBX and 3D application files on import; check the FBX import scale setting in Inspector.
  - If in doubt export a metre cube with your scene to match in Unity.
- Animation frame rate defaults can be different in packages, is a good idea to set consistently across your pipeline, for example 30fps.

### Files & Objects

- Name objects in your scene sensibly and uniquely. This can help you locate and troubleshoot specific meshes in your project.
- Avoid special characters * ( ) ? " # $ etc.
- Use simple but descriptive names for both objects and files (allow for duplication later).
- Keep your hierarchies as simple as you can.
- With big projects in your 3D application, consider having a working file outside your Unity project directory. This can often save time consuming updates and importing unnecessary data.

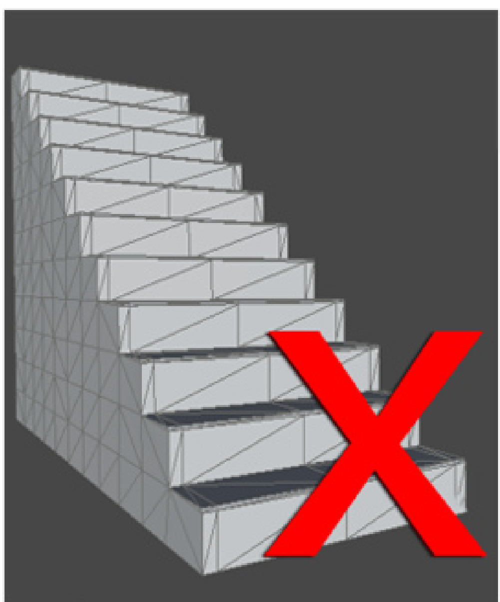
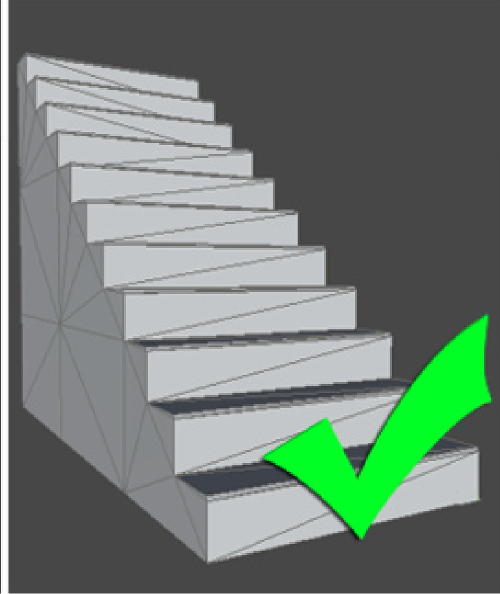



*Sensibly named objects help you find things quickly*

## Mesh

- Build with an efficient topology. Use polygons only where you need them.
- Optimise your geometry if it has too many polygons. Many character models need to be intelligently optimised or even rebuilt by an artist especially if sourced/built from:
  - 3D capture data

  - Poser
  - Zbrush
  - Other high density Nurbs/Patch models designed for render
- Where you can afford them, evenly spaced polygons in buildings, landscape and architecture will help spread lighting and avoid awkward kinks.
- Avoid really long thin triangles.

*Stairway to framerate heaven*

The method you use to construct objects can have a massive affect on the number of polygons, especially when not optimised. In this digram the same shape mesh has 156 triangles on the right and 726 on the left. 726 may not sound like a great deal of polygons, but if this is used 40 times in a level, you will really start to see the savings. A good rule of thumb is often to start simple and add detail where needed. It's always easier to add polygon than take them away.

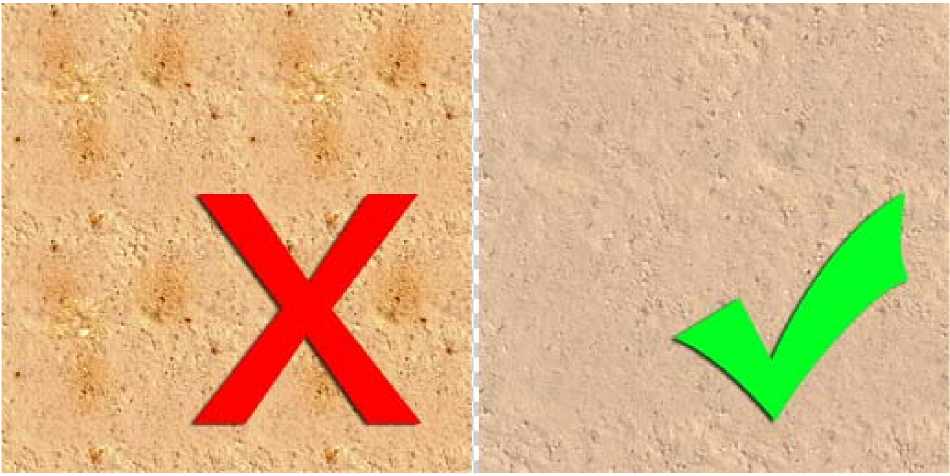
## Textures

If you author your textures to a power of two (e.g. 512�512 or 256�1024), the textures will be more efficient and won't need rescaling at build time. You can use up to 4096x4096 pixels, (although 2048x2048 is the highest available on many graphics cards/platforms). Search online for expert advice on creating good textures, but some of these guidelines can help you get the most efficient results from your project:

- Work with a high-resolution source file outside your unity project (such as a PSD or Gimp file). You can always downsize from source but not the other way round.
- Use the texture resolution output you require in your scene (save a copy, for example a 256x256 optimised PNG or a TGA file). You can make a judgement based on where the texture will be seen and where it is mapped.
- Store your output texture files together in your Unity project (for example: \Assets\textures).
- Make sure your 3D working file is referring to the same textures for consistency when you save/export.
- Make use of the available space in your texture, but be aware of different materials requiring different parts of the same texture. You can end up using/loading that texture multiple times.
- For alpha (cutout) and elements that may require different shaders, separate the textures. For example, the single texture below (left) has been replaced by three smaller textures below (right)

*One texture (left) vs three textures (right)*

- Make use of tiling textures (which seamlessly repeat) then you can use better resolution repeating over space.
- Remove easily noticeable repeating elements from your bitmap, and be careful with contrast. If you want to add details use decals and objects to break up the repeats.


*Tiling textures ftw*

- Unity takes care of compression for the output platform, so unless your source is already a JPG of the correct resolution it's better to use a lossless format for your textures.
- When creating a texture page from photographs, reduce the page to individual modular sections that can repeat. For example, you don't need twelve of the same windows using up texture space. That way you can have more pixel detail for that one window.

*Do you need ALL those windows?*

## Materials

- Organise and name the materials in your scene. This way you can find and edit your materials in Unity more easily when they�ve imported
- You can choose to create materials in Unity from either:
  - <modelname>-< material name> or:
  - <texture name> - make sure you are aware of which you want.
- Settings for materials in your native package will not all be imported to Unity:
  - Diffuse Colour, Diffuse texture and Names are usually supported
  - Shader model, specular, normal, other secondary textures and substance material settings will not be recognised/imported (coming in 3.5)

## Import/Export

Unity can use two types of files: Saved 3D application files and Exported 3D formats. Which you decide to use can be quite important:

### Saved application files

Unity can import, through conversion: Max, Maya, Blender, Cinema4D, Modo, Lightwave & cheetah3D files, e.g. .MAX, .MB, .MA etc. see more in Importing Objects.

**Advantages**:

- Quick iteration process (save and Unity updates)
- Simple initially

**Disadvantages**:

- A licensed copy of that software must be installed on all machines using the Unity project
- Files can become bloated with unnecessary data
- Big files can slow Unity updates
- Less Validation and harder to troubleshoot problems

### Exported 3D formats

Unity can also read FBX, OBJ, 3DS, DAE & DXF files. For a general export guide you can refer to this section this section

**Advantages**:

- Only export the data you need
- Verify your data (re-import into 3D package) before Unity
- Generally smaller files
- Encourages modular approach

**Disadvantages**:

- Can be slower pipeline or prototyping and iterations
- Easier to lose track of versions between source(working file) and game data (exported FBX)

Page last updated: 2012-11-16

---

## HOWTO-importObject

Unity supports importing from most popular 3D applications. Choose the one you're working with below:

- Maya
- Cinema 4D
- 3ds Max
- Cheetah3D
- Modo
- Lightwave

- Blender

### Other applications

Unity can read **.FBX**, **.dae**, **.3DS**, **.dxf** and **.obj** files, so check to see if your program can export to one of these formats. FBX exporters for popular 3D packages can be found here. Many packages also have a Collada exporter available.

### Hints

- Store textures in a folder called **Textures** next to the exported mesh. This will guarantee that Unity will always be able to find the Texture and automatically connect the Texture to the Material. For more information, see the Textures reference.

### See Also

- Modeling Optimized Characters
- How do I use normal maps?
- Mesh Import Settings
- How do I fix the rotation of an imported model?

Page last updated: 2012-08-06

---

# HOWTO-ImportObjectMaya

Unity natively imports Maya files. To get started, simply place your **.mb** or **.ma** file in your project's Assets folder. When you switch back into Unity, the scene is imported automatically and will show up in the Project view.

To see your model in Unity, simply drag it from the **Project View** into the **Scene View** or **Hierarchy View**.

### Unity currently imports from Maya:

1. All nodes with position, rotation and scale. Pivot points and Names are also imported.
2. Meshes with vertex colors, normals and up to 2 UV sets.
3. Materials with Texture and diffuse color. Multiple materials per mesh.
4. Animations FK & IK
5. Bone-based animations

Unity does not import blend shapes. Use Bone-based animations instead. Unity automatically triangulates polygonal meshes when importing, thus there is no need to do this manually in Maya.

If you are using IK to animate characters you have to select the imported .mb file in **Project View** and choose **Bake IK & Simulation** in the **Import Settings** dialog in the **Inspector**.

### Requirements

In order to import Maya **.mb** and **.ma** files, you need to have Maya installed on the machine you are using Unity to import the .mb/.ma file. Maya 8.0 and up is supported.

If you don't have Maya installed on your machine but want to import a Maya file from another machine, you can export to fbx format, which Unity imports natively. Please Install ->2011.3 for best results. To export see HOWTO_exportFBX.

Once exported Place the fbx file in the Unity project folder. Unity will now automatically import the fbx file. Check the FBX import setting in the inspector as mentioned in HOWTO_exportFBX

#### Behind the import process (Advanced)

When Unity imports a Maya file it will launch Maya in the background. Unity then communicates with Maya to convert the .mb file into a format Unity can read. The first time you import a Maya file in Unity, Maya has to launch in a command line process, this can take around 20 seconds, but subsequent imports will be very quick.

### Troubleshooting

- Keep your scene simple, try and work with a file which only contains the objects you need in Unity
- If your meshes cause problems, make sure you have converted any patches, nurbs surface etc into Polygons (Modify > Convert + also Mesh > Quadragulate/Triangulate) Unity only support Polygons.
- Maya in some rare cases messes up the node history, which sometimes results in models not exporting correctly. Fortunately you can very easily fix this by selecting **Edit->Delete by Type->Non-Deformer History**.

- Unity likes to keep up with the latest FBX where possible so if you have any issues with importing some models, check for the latest FBX exporters from Autodesk website or revert to FBX 2012
- Animation baking in Maya is now done with FBX instead of natively, which allows for more complex animations to be baked properly to FBX format. If you are using driven keys, then make sure to set at least one key on your drivers for the animation to bake properly

Page last updated: 2012-11-06

---

# HOWTO-ImportObjectCinema4D

Unity natively imports Cinema 4D files. To get started, simply place your **.c4d** file in your project's Assets folder. When you switch back into Unity, the scene is imported automatically and will show up in the **Project View**.

To see your model in Unity, simply drag it from the Project View into the **Scene View**.

If you modify your **.c4d** file, Unity will automatically update whenever you save.

### Unity currently imports

1. All objects with position, rotation and scale. Pivot points and Names are also imported.
2. Meshes with UVs and normals.
3. Materials with Texture and diffuse color. Multiple materials per mesh.
4. Animations FK (IK needs to be manually baked).
5. Bone-based animations.

Unity does not import Point Level Animations (PLA) at the moment. Use Bone-based animations instead.

### Animated Characters using IK

If you are using IK to animate your characters in Cinema 4D, you have to bake the IK before exporting using the **Plugins->Mocca->Cappucino** menu. If you don't bake your IK prior to importing into Unity, you will most likely only get animated locators but no animated bones.

### Requirements

- You need to have at least Cinema 4D version 8.5 installed to import **.c4d** files.

If you don't have Cinema 4D installed on your machine but want to import a Cinema 4D file from another machine, you can export to the FBX format, which Unity imports natively:

1. Open the Cinema 4D file
2. In Cinema 4D choose **File->Export->FBX 6.0**
3. Place the exported fbx file in the Unity project's Assets folder. Unity will now automatically import the fbx file.

### Hints

1. To maximize import speed when importing Cinema 4D files: go to the Cinema 4D preferences (**Edit->Preferences**) and select the FBX 6.0 preferences. Now uncheck **Embed Textures**.

### Behind the import process (Advanced)

When Unity imports a Cinema 4D file it will automatically install a Cinema 4D plugin and launch Cinema 4D in the background. Unity then communicates with Cinema 4D to convert the .c4d file into a format Unity can read. The first time you import a .c4d file and Cinema 4D is not open yet it will take a short while to launch it but afterwards importing **.c4d** files will be very quick.

### Cinema 4D 10 support

When importing **.c4d** files directly, Unity behind the scenes lets Cinema 4D convert its files to FBX. When Maxon shipped Cinema 4D 10.0, the FBX exporter was severly broken. With Cinema 4D 10.1 a lot of the issues have been fixed. Thus we strongly recommend everyone using Cinema 4D 10 to upgrade to 10.1.

Now there are still some issues left in Maxons FBX exporter. It seems that currently there is no reliable way of exporting animated characters that use the Joint's introduced in Cinema 4D 10. However the old bone system available in 9.6 exports perfectly fine. Thus when creating animated characters it is critical that you use the old bone system instead of joints.

Page last updated: 2007-11-16

---

# HOWTO-ImportObjectMax

If you make your 3D objects in 3dsMax, you can save your .max files directly into your **Project** or export them into Unity using the **Autodesk .FBX** or other genric formats.

### Unity imports meshes from 3ds Max. Saving a Max file or exporting a generic 3D file type each has advantages and disadvantages see Mesh

1. All nodes with position, rotation and scale. Pivot points and Names are also imported.
2. Meshes with vertex colors, normals and one or two UV sets.
3. Materials with diffuse texture and color. Multiple materials per mesh.
4. Animations.
5. Bone based animations.

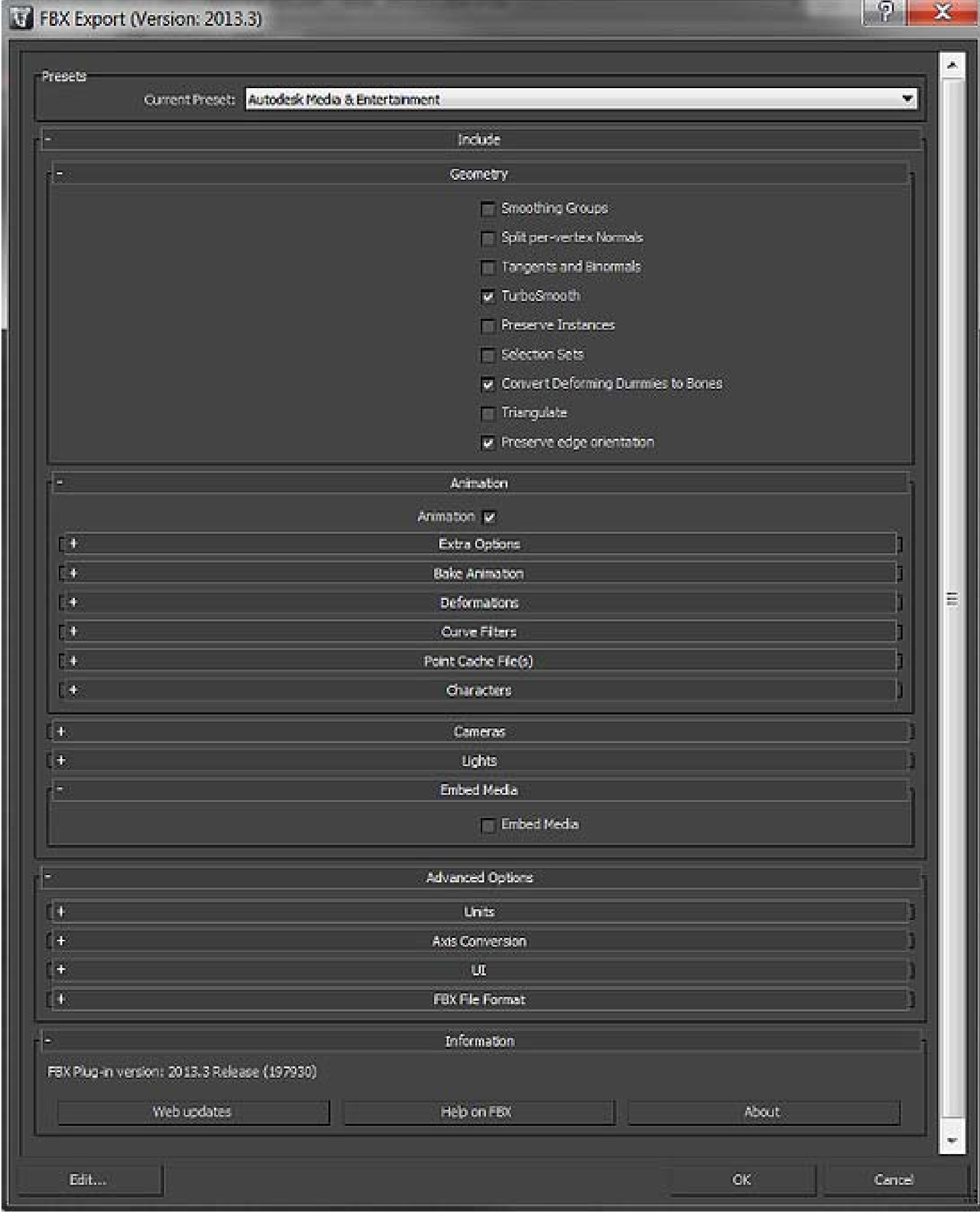
### To manually export to FBX from 3DS Max

1. Download the latest fbx exporter from Autodesk website and install it.
2. Export your scene or selected objects (**File->Export** or **File->Export Selected**) in **.fbx** format. Using default export options should be okay.
3. Copy the exported fbx file into your Unity project folder.
4. When you switch back into Unity, the **.fbx** file is imported automatically.
5. Drag the file from the **Project View** into the **Scene View**.

### Exporter options

Using default FBX exporter options (that basically export everything) you can choose:

Embed textures - this stores the image maps in the file, good for portability, not so good for file size




*Default FBX exporter options (for fbx plugin version 2013.3)*

## Exporting Bone-based Animations

There is a procedure you should follow when you want to export bone-based animations:

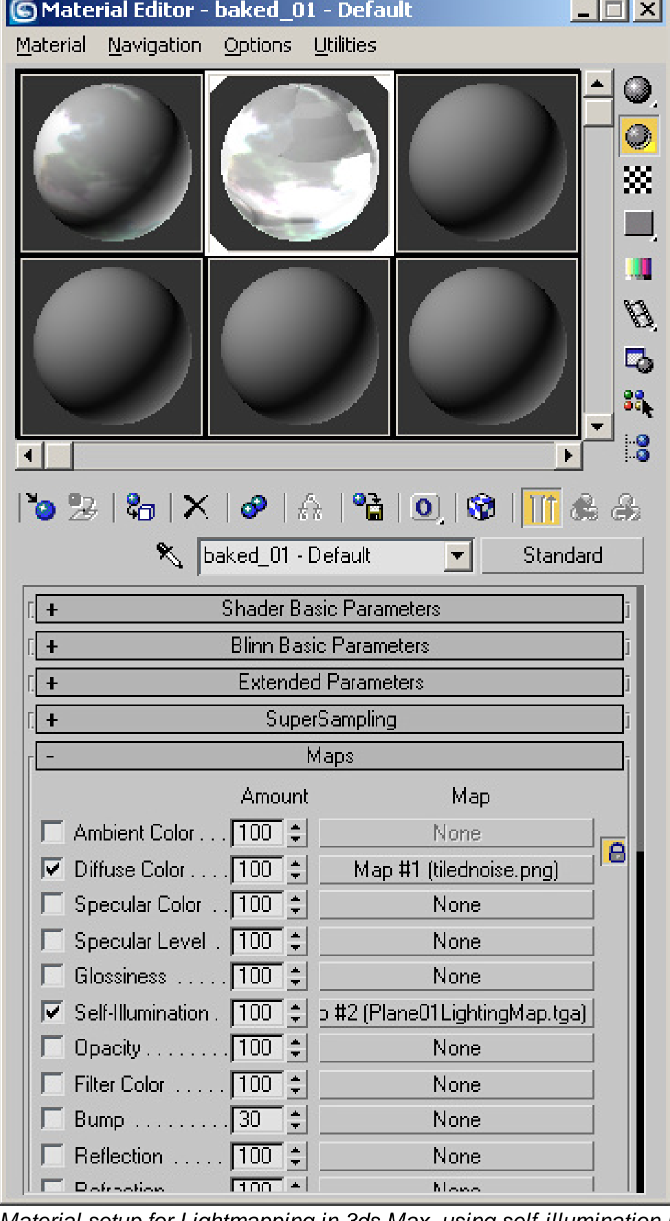
1. Set up the bone structure as you please.
2. Create the animations you want, using FK and/or IK
3. Select all bones and/or IK solvers
4. Go to **Motion->Trajectories** and press **Collapse**. Unity makes a key filter, so the amount of keys you export is irrelevant
5. "Export" or "Export selected" as newest FBX format
6. Drop the FBX file into **Assets** as usual
7. In Unity you must reassign the Texture to the Material in the root bone

When exporting a bone hierarchy with mesh and animations from 3ds Max to Unity, the GameObject hierarchy produced will correspond to the hierarchy you can see in "Schematic view" in 3ds Max. One difference is Unity will place a GameObject as the new root, containing the animations, and will place the mesh and material information in the root bone.

If you prefer to have animation and mesh information in the same Unity GameObject, go to the Hierarchy view in 3ds Max, and parent the mesh node to a bone in the bone hierarchy.

### Exporting Two UV Sets for Lightmapping

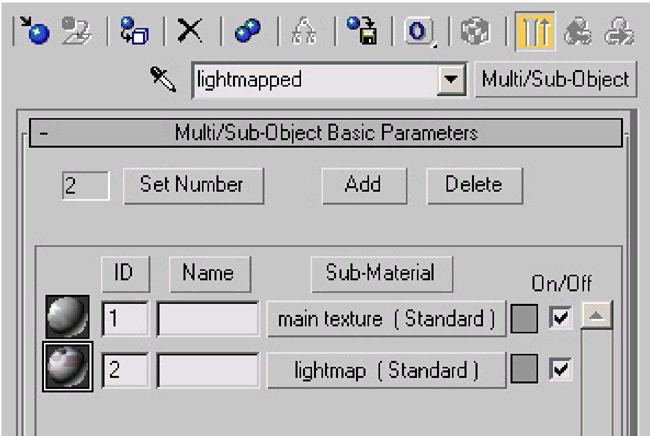
3ds Max' Render To Texture and automatic unwrapping functionality can be used to create lightmaps. Note that Unity has built-in lightmapper, but you might prefer using 3dsmax if that fits your workflow better. Usually one UV set is used for main texture and/or normal maps, and another UV set is used for the lightmap texture. For both UV sets to come through properly, the material in 3ds Max has to be Standard and both Diffuse (for main texture) and Self-Illumination (for lightmap) map slots have to be set up:




*Material setup for Lightmapping in 3ds Max, using self-illumination map*

Note that if object uses a Shell material type, then current Autodesk's FBX exporter will **not export UVs correctly**.

Alternatively, you can use Multi/Sub Object material type and setup two sub-materials, using the main texture and the lightmap in their diffuse map slots, like shown below. However, if faces in your model use different sub-material IDs, this will result in multiple materials being imported, which is not optimal for performance.




*Alternate Material setup for Lightmapping in 3ds Max, using multi/sub object material*

### Troubleshooting

If you have any issues with importing some models: ensure that you have the latest FBX plugin installed from Autodesk website or revert to FBX 2012.

Page last updated: 2012-11-06

---

# HOWTO-ImportObjectCheetah3D

Unity natively imports Cheetah3D files. To get started, simply place your **.jas** file in your project's Assets folder. When you switch back into Unity, the scene is imported automatically and will show up in the **Project View**.

To see your model in Unity, simply drag it from the Project View into the **Scene View**.

If you modify your **.jas** file, Unity will automatically update whenever you save.

### Unity currently imports from Cheetah3D

1. All nodes with position, rotation and scale. Pivot points and Names are also imported.
2. Meshes with vertices, polygons, triangles, UV's and Normals.
3. Animations.
4. Materials with diffuse color and textures.

### Requirements

- You need to have at least Cheetah3D 2.6 installed.

Page last updated: 2007-11-16

---

# HOWTO-ImportObjectModo

Unity natively imports modo files. This works under the hood by using the modo COLLADA exporter. Modo version 501 and later use this approach. To get started, save your .lxo file in the project's Assets folder. When you switch back into Unity, the file is imported automatically and will show up in the Project View.

For older modo versions prior to 501, simply save your Modo scene as an FBX or COLLADA file into the Unity project folder. When you switch back into Unity, the scene is imported automatically and will show up in the **Project View**.

To see your model in Unity, drag it from the Project View into the **Scene View**.

If you modify the lxo file, Unity will automatically update whenever you save.

### Unity currently imports

1. All nodes with position, rotation and scale. Pivot points and names are also imported.
2. Meshes with vertices, normals and UVs.
3. Materials with Texture and diffuse color. Multiple materials per mesh.
4. Animations

### Requirements

- modo 501 or later is required for native import of *.lxo files.

Page last updated: 2011-01-26

---

## HOWTO-importObjectLightwave

You can import meshes and animations from Lightwave using the FBX plugin for Lightwave.

### Unity currently imports

1. All nodes with position, rotation and scale. Pivot points and Names are also imported.
2. Meshes with UVs and normals.
3. Materials with Texture and diffuse color. Multiple materials per mesh.
4. Animations.
5. Bone-based animations.

### Installation

Download the latest Lightwave FBX exporter from:

- OS X lighwave 8.2 and 9.0 plugin
- OS X Lighwave 8.0 plugin
- Windows Lighwave 8.2 and 9.0 plugin
- Windows Lighwave 8.0 plugin

By downloading these plugins you automatically agree to this licence.

There are two versions of the plugin, one for LightWave 8.0 and one for LightWave 8.2 through 9.0. Make sure you install the correct version.

The plugin for Mac comes in an OS X package. If you double-click the package to install it, the installer will try to put it in the correct folder. If it can't find your LightWave plugin folder, it will create its own LightWave folder in your **Applications** folder and dump it there. If the latter occurs you should move it to your LightWave plugin folder (or any sub-folder). Once there you have to add the plugin to LightWave via the "Edit Plugins" panel (**Option-F11**) - see the LightWave manual for more details on how to add plugins.

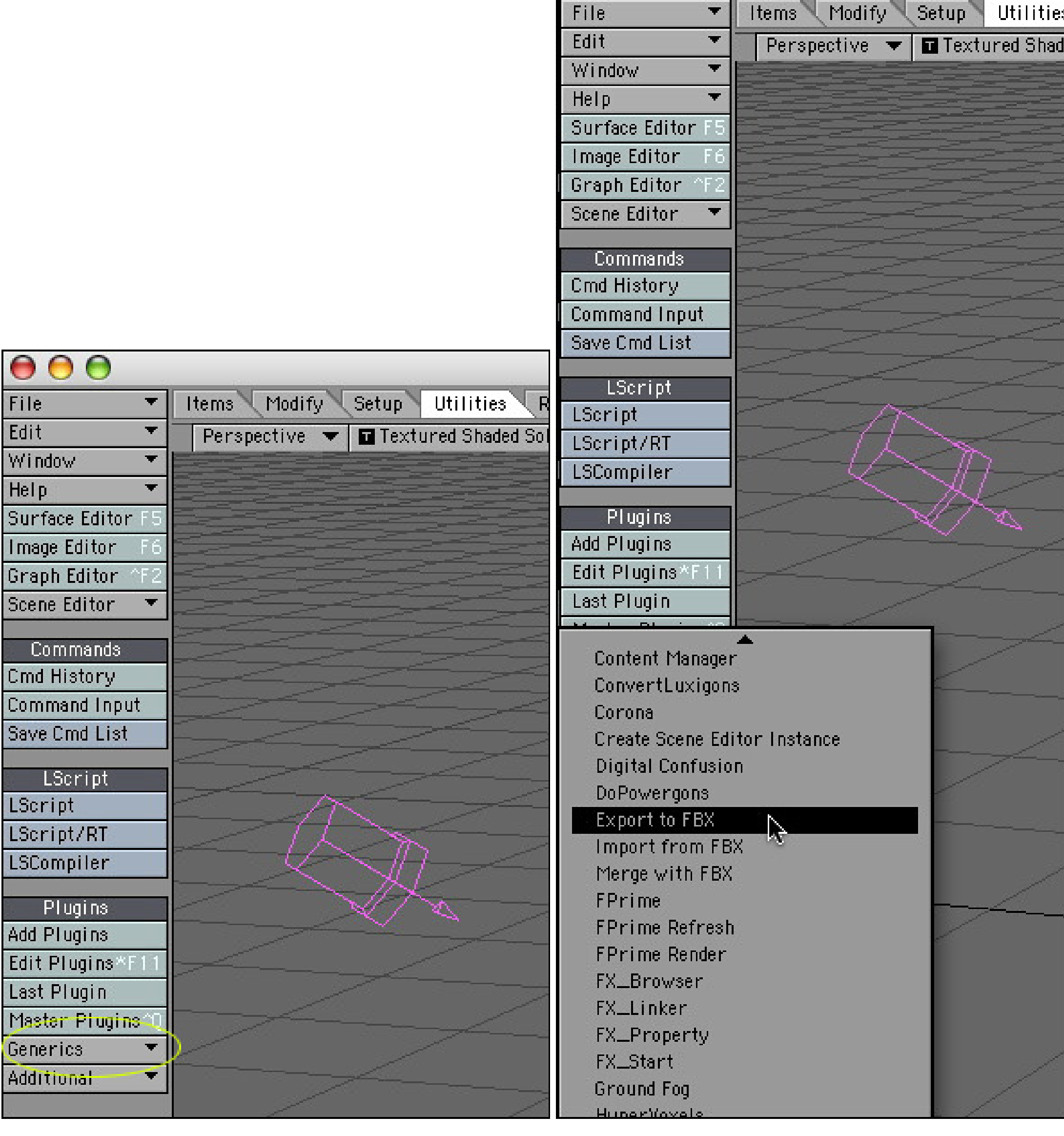
Once added to LightWave, the plugin is acessible via the Generics menu (on the Utiliies) tab. If the Generic menu is not present you will have to add it using the Config Menus panel. In the latter panel it can be found in the Plug-ins category and is called "Generic Plugins". Add it to any convenient menu (see the LightWave manual for more details on how to do this).

More information about installation can also be found in the release notes that can downloaded with the installer.

## Exporting

All objects and animations have to be exported from Layout (there is no Modeler FBX exporter).

**1. Select Export to FBX from the Generics menu**




**2. Select the appropriate settings in the fbx export dialog**

- Select the fbx file name. Make sure to save the exported fbx file in the Assets folder of your current Unity project.
- In the FBX dialogue panel you MUST select **Embed Textures** otherwise the exported object will have no UVs. This is a bug in the Lightwave fbx exporter and will be fixed in a future version according to Autodesk.
- If you want to export animations into Unity you must have "Animations" checked. You also need to have "Lights" or "Cameras" checked.
- To change the name of the exported animation clip in Unity, change the name from "LW Take 001" to your liking.

**3. Switch to Unity.**

- Unity will automatically import the fbx file and generate materials for the textures.
- Drag the imported fbx file from the Project view into the Scene view.

### Important notes

- You must select **Embed Textures** in the FBX panel when exporting or no UVs are exported
- If you want to export animations you must enable **Animations** and either **Camera** or **Lights**.
- It is strongly recommended to always place your textures in a folder called "Textures" next to the fbx file. This will guarantee that Unity can always find the Texture and automatically connect the texture to the material.

Page last updated: 2007-11-16

---

# HOWTO-ImportObjectBlender

Unity natively imports Blender files. This works under the hood by using the Blender FBX exporter, which was added to Blender in version 2.45. For this reason, you must update to Blender 2.45 or later (but see Requirements below).

To get started, save your **.blend** file in your project's Assets folder. When you switch back into Unity, the file is imported automatically and will show up in the **Project View**.

To see your model in Unity, drag it from the Project View into the **Scene View**.

If you modify your **.blend** file, Unity will automatically update whenever you save.

### Unity currently imports

1. All nodes with position, rotation and scale. Pivot points and Names are also imported.
2. Meshes with vertices, polygons, triangles, UVs, and normals.
3. Bones
4. Skinned Meshes
5. Animations

### Requirements

- You need to have Blender version 2.45-2.49 or 2.58 or later (versions 2.50-2.57 do not work, because FBX export was changed/broken in Blender).
- Textures and diffuse color are not assigned automatically. Manually assign them by dragging the texture onto the mesh in the Scene View in Unity.

Page last updated: 2011-08-10

## Workflow

- Getting started with Mono Develop
- How do I reuse assets between projects?
- How do I install or upgrade Standard Assets?
- Porting a Project Between Platforms

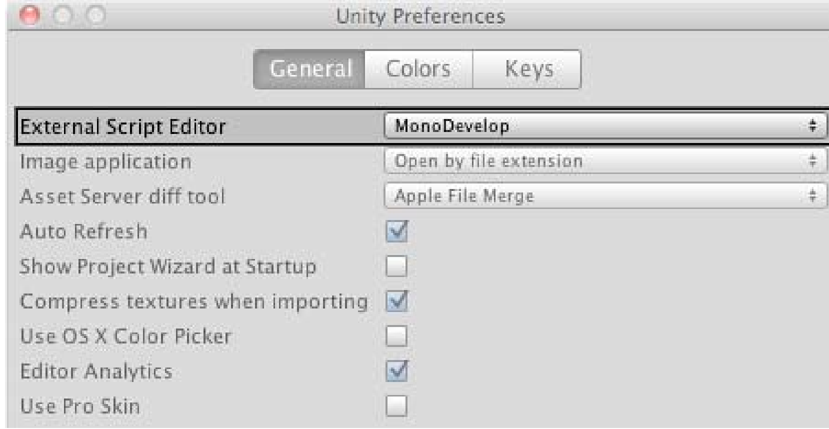
Page last updated: 2007-11-16

## HOWTO-MonoDevelop

Mono Develop comes now with Unity 3.x, this IDE will help you out taking care of the scripting part of your game and the debugging o it.

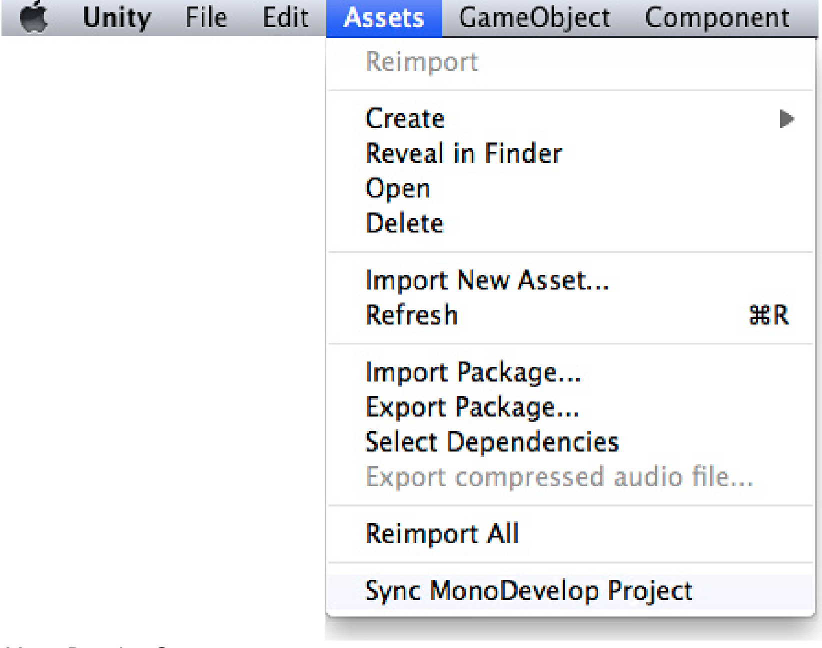
## Setting Up Mono Develop.

To set up Mono Develop to work with with Unity you just have to go to Unity Preferences and set it as your default editor.




*Setting Mono Develop as the Default Editor*

After this, create or open an existing project and make sure your project is synced with Mono Develop by clicking on **Assets -> Sync Mono Develop Project**.




*Mono Develop Sync.*

This will open your project (Only The scripting files, no Assets) in Mono Develop. Now you are ready to start debugging.

Also you might want to visit the troubleshooting page in case you have any problem setting your project.

Page last updated: 2010-09-24

---

# HOWTO-exportpackage

As you build your game, Unity stores a lot of metadata about your assets (import settings, links to other assets, etc.). If you want to take your assets into a different project, there is a specific way to do that. Here's how to easily move assets between projects and still preserve all this info.

1. In the **Project View**, select all the asset files you want to export.
2. Choose **Assets->Export Package...** from the menubar.
3. Name and save the package anywhere you like.
4. Open the project you want to bring the assets into.
5. Choose **Assets->Import Package...** from the menubar.
6. Select your package file saved in step 3.

## Hints

- When exporting a package Unity can export all dependencies as well. So for example if you select a **Scene** and export a package with all dependencies, then all models, textures and other assets that appear in the scene will be exported as well. This can be a quick way of exporting a bunch of assets without manually locating them all.
- If you store your Exported Package in the Standard Packages folder next to your Unity application, they will appear in the **Create New Project** dialog.

Page last updated: 2007-11-16

---

# HOWTO-InstallStandardAssets

Unity ships with multiple **Standard Assets** packages. These are collections of assets that are widely used by most Unity customers. When you create a new project from the Project Wizard you can optionally include these asset collections. These assets are copied from the Unity install folder into your new project. This means that if you upgrade Unity to a new version you will not get the new version of these assets and so upgrading them is needed. Also, consider that a newer version of e.g. an effect might behave differently for performance or quality reasons and thus requires retweaking of parameters. It's important to consider this before upgrading if you don't want your game to suddenly look or behave differently. Check with the package contents and Unity's release notes.

Standard Assets contain useful things like a first person controller, skyboxes, lens flares, Water prefabs, Image Effects and so on.

*The Standard Assets packages listed when creating new project*

### Upgrading

Sometimes you might want to upgrade your Standard Assets, for example because a new version of Unity ships with new Standard Assets:

1. Open your project.
2. Choose package you want to update from **Assets->Import Package** submenu.
3. A list of new or replaced assets will be presented, click **Import**.

For the cleanest possible upgrade, it should be considered to remove the old package contents first, as some scripts, effects or prefabs might have become deprecated or unneeded and Unity packages don't have a way of deleting (unneeded) files (but make sure to have a security copy of the old version available).

Page last updated: 2011-06-09

---

## HOWTO-PortingBetweenPlatforms

Most of Unity's API and project structure is identical for all supported platforms and in some cases a project can simply be rebuilt to run on different devices. However, fundamental differences in the hardware and deployment methods mean that some parts of a project may not port between platforms without change. Below are details of some common cross-platform issues and suggestions for solving them.

### Input

The most obvious example of different behaviour between platforms is in the input methods offered by the hardware.

**Keyboard and joypad**

The **Input.GetAxis** function is very convenient on desktop platforms as a way of consolidating keyboard and joypad input. However, this function doesn't make sense for the mobile platforms which rely on touchscreen input. Likewise, the standard desktop keyboard input doesn't port over to mobiles well for anything other than typed text. It is worthwhile to add a layer of abstraction to your input code if you are considering porting to other platforms in the future. As a simple example, if you were making a driving game then you might create your own input class and wrap the Unity API calls in your own functions:-

```
// Returns values in the range -1.0 .. +1.0 (== left .. right).
function Steering() {
	return Input.GetAxis("Horizontal");
}


// Returns values in the range -1.0 .. +1.0 (== accel .. brake).
function Acceleration() {
	return Input.GetAxis("Vertical");
}


var currentGear: int;

// Returns an integer corresponding to the selected gear.
function Gears() {
	if (Input.GetKeyDown("p"))
		currentGear++;
	else if (Input.GetKeyDown("l"))
		currentGear--;

	return currentGear;
}
```

One advantage of wrapping the API calls in a class like this is that they are all concentrated in a single source file and are consequently easy to locate and replace. However, the more important idea is that you should design your input functions according to the logical meaning of the inputs in your game. This will help to isolate the rest of the game code from the specific method of input used with a particular platform. For example, the Gears function above could be modified so that the actual input comes from touches on the screen of a mobile device. Using an integer to represent the chosen gear works fine for all platforms, but mixing the platform-specific API calls with the rest of the code would cause problems. You may find it convenient to use platform dependent compilation to combine the different implementation of the input functions in the same source file and avoid manual swaps.

**Touches and clicks**

The **Input.GetMouseButtonXXX** functions are designed so that they have a reasonably obvious interpretation on mobile devices even though there is no "mouse" as such. A single touch on the screen is reported as a left click and the **Input.mousePosition** property gives the position of the touch as long as the finger is touching the screen. This means that games with simple mouse interaction can often work transparently between the desktop and mobile platforms. Naturally, though, the conversion is often much less straightforward than this. A desktop game can make use of more than one mouse button and a mobile game can detect multiple touches on the screen at a time.

As with API calls, the problem can be managed partly by representing input with logical values that are then used by the rest of the game code. For example, a pinch gesture to zoom on a mobile device might be replaced by a plus/minus keystroke on the desktop; the input function could simply return a float value specifying the zoom factor. Likewise, it might be possible to use a two-finger tap on a mobile to replace a right button click on the desktop. However, if the properties of the input device are an integral part of the game then it may not be possible to remodel them on a different platform. This may mean that game cannot be ported at all or that the input and/or gameplay need to be modified extensively.

**Accelerometer, compass, gyroscope and GPS**

These inputs derive from the mobility of handheld devices and so may not have any meaningful equivalent on the desktop. However, some use cases simply mirror standard game controls and can be ported quite easily. For example, a driving game might implement the steering control from the tilt of a mobile device (determined by the accelerometer). In cases like this, the input API calls are usually fairly easy to replace, so the accelerometer input might be replaced by keystrokes, say. However, it may be necessary to recalibrate inputs or even vary the difficulty of the game to take account of the different input method.

Tilting a device is slower and eventually more strenuous than pressing keys and may also make it harder to concentrate on the display. This may result in the game's being more difficult to master on a mobile device and so it may be appropriate to slow down gameplay or allow more time per level. This will require the game code to be designed so that these factors can be adjusted easily.

### Memory, storage and CPU performance

Mobile devices inevitably have less storage, memory and CPU power available than desktop machines and so a game may be difficult to port simply because its performance is not acceptable on lower powered hardware. Some resource issues can be managed but if you are pushing the limits of the hardware on the desktop then the game is probably not a good candidate for porting to a mobile platform.

#### Movie playback

Currently, mobile devices are highly reliant on hardware support for movie playback. The result is that playback options are limited and certainly don't give the flexibility that the MovieTexture asset offers on desktop platforms. Movies can be played back fullscreen on mobiles but there isn't any scope for using them to texture objects within the game (so it isn't possible to display a movie on a TV screen within the game, for example). In terms of portability, it is fine to use movies for introductions, cutscenes, instructions and other simple pieces of presentation. However, if movies need to be visible within the game world then you should consider whether the mobile playback options will be adequate.

#### Storage requirements

Video, audio and even textures can use a lot of storage space and you may need to bear this in mind if you want to port your game. Storage space (which often also corresponds to download time) is typically not an issue on desktop machines but this is not the case with mobiles. Furthermore, mobile app stores often impose a limit on the maximum size of a submitted product. It may require some planning to address these concerns during the development of your game. For example, you may need to provide cut-down versions of assets for mobiles in order to save space. Another possibility is that the game may need to be designed so that large assets can be downloaded on demand rather than being part of the initial download of the application.

#### Automatic memory management

The recovery of unused memory from "dead" objects is handled automatically by Unity and often happens imperceptibly on desktop machines. However, the lower memory and CPU power on mobile devices means that garbage collections can be more frequent and the time they take can impinge more heavily on performance (causing unwanted pauses in gameplay, etc). Even if the game runs in the available memory, it may still be necessary to optimise code to avoid garbage collection pauses. More information can be found on our memory management page.

#### CPU power

A game that runs well on a desktop machine may suffer from poor framerate on a mobile device simply because the mobile CPU struggles with the game's complexity. Extra attention may therefore need to be paid to code efficiency when a project is ported to a mobile platform. A number of simple steps to improve efficiency are outlined on this page in our manual.

Page last updated: 2012-05-31

---

# MobileDeveloperChecklist

If you are having problems when developing for a mobile platform, this is a checklist to help you solve various problems.

- Crashes
- Profiling
- Optimizations

Page last updated: 2012-10-10

---

# MobileCrashes

### Checklist for crashes

- Disable code stripping (and set �slow with exceptions� for iOS)

- Follow the instructions on Optimizing the Size of the Built iOS Player (http://docs.unity3d.com/Documentation/Manual/iphone-playerSizeOptimization.html) to make sure your game does not crash with stripping on iOS.
- Verify it is not because of out of memory (restart your device, use the device with maximum RAM for the platform, be sure to watch the logs)

### Editor.log - on the editor

The Debug messages, warnings and errors all go to the console. Also Unity prints status reports to the console � loading assets, initializing mono, graphics driver info.

If you are trying to understand what is going on look at the editor.log. Here you will get the full picture, not just a console fragment. You can try to understand what�s happening, and watch the full log of your coding session. This will help you track down what has caused Unity crash to crash or find out what�s wrong with your assets.

Unity prints some tjings on the devices as well; Logcat console for android and Xcode gdb console on iOS devices

## Android

### Debugging on Android

1. Use the *DDMS* or *ADB* tool
2. Watch the stacktrace (Android 3 or newer). Either use *c++filt* (part of the *ndk*) or the other methods, like: http://slush.warosu.org/c++filtjs to decode the mangled function calls
3. Look at the *.so* file that the crash occurs on:
   1. *libunity.so* - the crash is in the Unity code or the user code
   2. *libdvm.so* - the crash is in the Java world, somewhere with Dalvik. So find Dalvik�s stacktrace, look at your JNI code or anything Java-related (including your possible changes to the *AndroidManifest.xml*).
   3. *libmono.so* - either a Mono bug or you're doing something Mono strongly dislikes
4. If the crashlog does not help you can disassemble it to get a rough understanding of what has happened.
   1. use ARM EABI tools from the Android NDK like this: *objdump.exe -S libmono.so >> out.txt*
   2. Look at the code around pc from the stacktrace.
   3. try to match that code within the fresh *out.txt* file.
   4. Scroll up to understand what is happening in the function it occurs in.

## iOS

### Debugging on iOS

1. Xcode has built in tools. Xcode 4 has a really nice GUI for debugging crashes, Xcode 3 has less.
2. Full gdb stack - thread apply all bt
3. Enable soft-null-check:

Enable development build and script debugging. Now uncaught null ref exceptions will be printed to the Xcode console with the appropriate managed call stack.

1. Try turning the "fast script call" and code stripping off. It may stop some random crashes, like those caused by using some rare *.Net* functions or reflection.

### Strategy

1. Try to figure out which script the crash happens in and debug it using mono develop on the device.
2. If the crash seems to not be in your code, take a closer look at the stacktrace, there should be a hint of something happening. Take a copy and submit it, and we�ll take a look.

Page last updated: 2012-10-10

---

# MobileProfiling

Ports that the Unity profiler uses:

```
MulticastPort : 54998
ListenPorts : 55000 - 55511
Multicast(unittests) : 55512 - 56023
```

They should be accessible from within the network node. That is, the devices that you�re trying to profile on should be able to see these ports on the machine with the Unity Editor with the Profiler on.

### First steps

Unity relies on the CPU (heavily optimized for the SIMD part of it, like SSE on x86 or NEON on ARM) for skinning, batching, physics, user scripts, particles, etc.

The GPU is used for shaders, drawcalls, image effects.

#### CPU or GPU bound

- Use the internal profiler to detect the CPU and GPU ms

#### Pareto analysis

A large majority of problems (80%) are produced by a few key causes (20%).

1. Use the Editor profiler to get the most problematic function calls and optimize them first.
2. Make sure the scripts run only when necessary.
    1. Use *OnBecameVisible/OnBecameInvisible* to disable inactive objects.
    2. Use coroutines if you don�t need some scripts to run every frame.

```
// Do some stuff every frame:
void Update () {
}

//Do some stuff every 0.2 seconds:
IEnumerator Start ()_ {
  while (true) {
    yield return new WaitForSeconds (0.2f);
  }
}
```

1. Use the *.NET System.Threading.Thread* class to put heavy calculations to the other thread. This allows you to run on multiple cores, but Unity API is not thread-safe. So buffer inputs and results and read and assign them on the main thread.

### CPU Profiling

#### Profile user code

Not all of the user code is shown in the Profiler. But you can use *Profiler.BeginSample* and *Profiler.EndSample* to make the required user code appear in the profiler.

### GPU Profiling

The Unity Editor profiler cannot show GPU data as of now. We�re working with hardware manufacturers to make it happen with the Tegra devices being the first to appear in the Editor profiler.

## iOS

#### Tools for iOS

- Unity internal profiler (not the Editor profiler). This shows the GPU time for the whole scene.
- PowerVR PVRUniSCo shader analyzer. See below.
- iOS: Xcode OpenGL ES Driver Instruments can show only high-level info:
    - �Device Utilization %� - GPU time spent on rendering in total. >95% means the app is GPU bound.
    - �Renderer Utilization %� - GPU time spent drawing pixels.
    - �Tiler Utilization %� - GPU time spent processing vertices.
    - �Split count� - the number of frame splits, where the vertex data didn�t fit into allocated buffers.

PowerVR is tile based deferred renderer, so it’s impossible to get GPU timings per draw call. However you can get GPU times

for the whole scene using Unity�s built-in profiler (the one that prints results to Xcode output). Apple�s tools currently can only tell you how busy the GPU and its parts are, but do not give times in milliseconds.

PVRUniSCo gives cycles for the whole shader, and approximate cycles for each line in the shader code. Windows & Mac! But it won�t match what Apple�s drivers are doing exactly anyway. Still, a good ballpark measure.

# Android

**Tools for Android**

- Adreno (Qualcomm)
- NVPerfHUD (NVIDIA)
- PVRTune, PVRUniSCo (PowerVR)

On Tegra, NVIDIA provides excellent performance tools which does everything you want - GPU time per draw call, Cycles per shader, Force 2x2 texture, Null view rectangle, runs on Windows, OSX, Linux. PerfHUD ES does not easily work with consumer devices, you need the development board from NVIDIA.

Qualcomm provides excellent Adreno Profiler (Windows only) which is Windows only, but works with consumer devices! It features Timeline graphs, frame capture, Frame debug, API calls, Shader analyzer, live editing.

**Graphics related CPU profiling**

The internal profiler gives a good overview per module:

- time spent in OpenGL ES API
- batching efficiency
- skinning, animations, particles

## Memory

Integrate this: http://docwiki.hq.unity3d.com/internal/index.php?n=Support.MemoryUsage

There is Unity memory and mono memory.

**Mono memory**

Mono memory handles script objects, wrappers for Unity objects (game objects, assets, components, etc). Garbage Collector cleans up when the allocation does not fit in the available memory or on a *System.GC.Collect()* call.

Memory is allocated in heap blocks. More can allocated if it cannot fit the data into the allocated block. Heap blocks will be kept in Mono until the app is closed. In other words, Mono does not release any memory used to the OS (Unity 3.x). Once you allocate a certain amount of memory, it is reserved for mono and not available for the OS. Even when you release it, it will become available internally for Mono only and not for the OS. The heap memory value in the Profiler will only increase, never decrease.

If the system cannot fit new data into the allocated heap block, the Mono calls a "GC" and can allocate a new heap block (for example, due to fragmentation).

�Too many heap sections� means you�ve run out of Mono memory (because of fragmentation or heavy usage).

Use *System.GC.GetTotalMemory* to get the total used Mono memory.

The general advice is, use as small an allocation as possible.

**Unity memory**

Unity memory handles Asset data (Textures, Meshes, Audio, Animation, etc), Game objects, Engine internals (Rendering, Particles, Physics, etc). Use *Profiler.usedHeapSize* to get the total used Unity memory.

**Memory map**

No tools yet but you can use the following.

- Unity Profiler - not perfect, skips stuff, but you can get an overview. It works on the device!
- Internal profiler

  - Shows Used heap and allocated heap - see mono memory.
  - Shows the number of mono allocations per frame.
- Xcode tools - iOS
  - Xcode Instruments Activity Monitor - Real Memory column.
  - Xcode Instruments Allocations - net allocations for created and living objects.
  - VM Tracker
    - textures usually get allocated with IOKit label.
    - meshes usually go into VM Allocate.
- Make your own tool
  - *FindObjectsOfTypeAll (type : Type) : Object[]*
  - *FindObjectsOfType (type : Type): Object[]*
  - *GetRuntimeMemorySize (o : Object) : int*
  - *GetMonoHeapSize*
  - *GetMonoUsedSize*
  - *Profiler.BeginSample/EndSample* - profile your own code
  - *UnloadUnusedAssets () : AsyncOperation*
  - *System.GC.GetTotalMemory/Profiler.usedHeapSize*
- References to the loaded objects - There is no way to figure this out. A workaround is to �Find references in scene� for public variables.

**Memory hiccups**

- Garbage collector
  - This fires when the system cannot fit new data into the allocated heap block.
  - Don�t use *OnGUI* on mobiles
    - It shoots several times per frame
    - It completely redraws the view.
    - It creates tons of memory allocation calls that require Garbage Collection to be invoked.
  - Creating/removing too many objects too quickly?
    - This may lead to fragmentation.
    - Use the Editor profiler to track the memory activity.
    - The internal profiler can be used to track the mono memory activity.
  - *System.GC.Collect()* You can use this *.Net* function when it�s ok to have a hiccup.
- New memory allocations
  - Allocation hiccups
    - Use lists of preallocated, reusable class instances to implement your own memory management scheme.
    - Don�t make huge allocations per frame, cache, preallocate instead
  - Problems with fragmentation?
    - Preallocate the memory pool.
    - Keep a List of inactive GameObjects and reuse them instead of Instantiating and Destroying them.
  - Out of mono memory
    - Profile memory activity - when does the first memory page fill up?
    - Do you really need so many gameobjects that a single memory page is not enough?
  - Use structs instead of classes for local data. Classes are stored on the heap; structs on the stack.

```
class MyClass {
  public int a, b, c;
}

struct MyStruct {
  public int a, b, c;
}

void Update () {
  //BAD
  // allocated on the heap, will be garbage collected later!
  MyClass c = new MyClass();

  //GOOD
  //allocated on the stack, no GC going to happen!
  MyStruct s = new MyStruct();
}
```

- Read the relevant section in the manual Link to http://docs.unity3d.com/Documentation/Manual/UnderstandingAutomaticMemoryManagement.html

**Out of memory crashes**
At some points a game may crash with "out of memory" though it in theory it should fit in fine. When this happens compare your normal game memory footprint and the allocated memory size when the crash happens. If the numbers are not similar, then there is a memory spike. This might be due to:

- Two big scenes being loaded at the same time - use an empty scene between two bigger ones to fix this.
- Additive scene loading - remove unused parts to maintain the memory size.
- Huge asset bundles loaded to the memory
- Loading via WWW or instantiating (a huge amount of) big objects like:
  - Textures without proper compression (a no go for mobiles).
  - Textures having Get/Set pixels enabled. This requires an uncompressed copy of the texture in memory.
  - Textures loaded from JPEG/PNGs at runtime are essentially uncompressed.
  - Big mp3 files marked as decompress on loading.
- Keeping unused assets in weird caches like static monobehavior fields, which are not cleared when changing scenes.

Page last updated: 2012-10-10

---

# MobileOptimisation

Just like on PCs, mobile platforms like iOS and Android have devices of various levels of performance. You can easily find a phone that�s 10x more powerful for rendering than some other phone. Quite easy way of scaling:

1. Make sure it runs okay on baseline configuration
2. Use more eye-candy on higher performing configurations:
   - Resolution
   - Post-processing
   - MSAA
   - Anisotropy
   - Shaders
   - Fx/particles density, on/off

## Focus on GPUs

Graphics performance is bound by fillrate, pixel and geometric complexity (vertex count). All three of these can be reduced if you can find a way to cull more renderers. Occlusion culling and could help here. Unity will automatically cull objects outside the viewing frustum.

On mobiles you�re essentially fillrate bound (fillrate = screen pixels * shader complexity * overdraw), and over-complex shaders is the most common cause of problems. So use mobile shaders that come with Unity or design your own but make them as simple as possible. If possible simplify your pixel shaders by moving code to vertex shader.

If reducing the Texture Quality in Quality Settings makes the game run faster, you are probably limited by memory bandwidth. So compress textures, use mipmaps, reduce texture size, etc.

LOD (Level of Detail) � make objects simpler or eliminate them completely as they move further away. The main goal would be to reduce the number of draw calls.

**Good practice**
Mobile GPUs have huge constraints in how much heat they produce, how much power they use, and how large or noisy they can be. So compared to the desktop parts, mobile GPUs have way less bandwidth, low ALU performance and texturing power. The architectures of the GPUs are also tuned to use as little bandwidth & power as possible.

Unity is optimized for OpenGL ES 2.0, it uses GLSL ES (similar to HLSL) shading language. Built in shaders are most often written in HLSL (also known as Cg). This is cross compiled into GLSL ES for mobile platforms. You can also write GLSL directly if you want to, but doing that limits you to OpenGL-like platforms (e.g. mobile + Mac) since there currently are no GLSL->HLSL translation tools. When you use float/half/fixed types in HLSL, they end up highp/mediump/lowp precision qualifiers in GLSL ES.

Here is the checklist for good practice:

1. Keep the number of materials as low as possible. This makes it easier for Unity to batch stuff.
2. Use texture atlases (large images containing a collection of sub-images) instead of a number of individual textures. These are faster to load, have fewer state switches, and are batching friendly.
3. Use *Renderer.sharedMaterial* instead of *Renderer.material* if using texture atlases and shared materials.
4. Forward rendered pixel lights are expensive.
    - Use light mapping instead of realtime lights where ever possible.
    - Adjust pixel light count in quality settings. Essentially only the directional light should be per pixel, everything else - per vertex. Certainly this depends on the game.
5. Experiment with Render Mode of Lights in the Quality Settings to get the correct priority.
6. Avoid Cutout (alpha test) shaders unless really necessary.
7. Keep Transparent (alpha blend) screen coverage to a minimum.
8. Try to avoid situations where multiple lights illuminate any given object.
9. Try to reduce the overall number of shader passes (Shadows, pixel lights, reflections).
10. Rendering order is critical. In general case:
    1. fully opaque objects roughly front-to-back.
    2. alpha tested objects roughly front-to-back.
    3. skybox.
    4. alpha blended objects (back to front if needed).
11. Post Processing is expensive on mobiles, use with care.
12. Particles: reduce overdraw, use the simplest possible shaders.
13. Double buffer for Meshes modified every frame:

```
void Update (){
  // flip between meshes
  bufferMesh = on ? meshA : meshB;
  on = !on;
  bufferMesh.vertices = vertices; // modification to mesh
  meshFilter.sharedMesh = bufferMesh;
}
```

**Sharer optimizations**

Checking if you are fillrate-bound is easy: does the game run faster if you decrease the display resolution? If yes, you are limited by fillrate.

Try reducing shader complexity by the following methods:

- Avoid alpha-testing shaders; instead use alpha-blended versions.
- Use simple, optimized shader code (such as the �Mobile� shaders that ship with Unity).
- Avoid expensive math functions in shader code (pow, exp, log, cos, sin, tan, etc). Consider using pre-calculated lookup textures instead.
- Pick lowest possible number precision format (float, half, fixedin Cg) for best performance.

## Focus on CPUs

It is often the case that games are limited by the GPU on pixel processing. So they end up having unused CPU power, especially on multicore mobile CPUs. So it is often sensible to pull some work off the GPU and put it onto the CPU instead (Unity does all of these): mesh skinning, batching of small objects, particle geometry updates.

These should be used with care, not blindly. If you are not bound by draw calls, then batching is actually worse for performance, as it makes culling less efficient and makes more objects affected by lights!

**Good practice**

- Don�t use more than a few hundred draw calls per frame on mobiles.
- FindObjectsOfType (and Unity getter properties in general) are very slow, so use them sensibly.
- Set the Static property on non-moving objects to allow internal optimizations like static batching.
- Spend lots of CPU cycles to do occlusion culling and better sorting (to take advantage of Early Z-cull).

**Physics**

Physics can be CPU heavy. It can be profiled via the Editor profiler. If Physics appears to take too much time on CPU:

- Tweak *Time.fixedDeltaTime* (in Project settings -> Time) to be as high as you can get away with. If your game is slow moving, you probably need less fixed updates than games with fast action. Fast paced games will need more frequent calculations, and thus *fixedDeltaTime* will need to be lower or a collision may fail.
- Physics.solverIterationCount (Physics Manager).
- Use as little Cloth objects as possible.
- Use Rigidbodies only where necessary.
- Use primitive colliders in preference mesh colliders.
- Never ever move a static collider (ie a collider without a Rigidbody) as it causes a big performance hit.
  - Shows up in Profiler as �Static Collider.Move� but actual processing is in *Physics.Simulate*
  - If necessary, add a RigidBody and set *isKinematic* to true.
- On Windows you can use NVidia�s AgPerfMon profiling tool set to get more details if needed.

## Android

**GPU**

These are the popular mobile architectures. This is both different hardware vendors than in PC/console space, and very different GPU architectures than the �usual� GPUs.

- ImgTec PowerVR SGX - Tile based, deferred: render everything in small tiles (as 16x16), shade only visible pixels
- NVIDIA Tegra - Classic: Render everything
- Qualcomm Adreno - Tiled: Render everything in tile, engineered in large tiles (as 256k). Adreno 3xx can switch to traditional.
- ARM Mali Tiled: Render everything in tile, engineered in small tiles (as 16x16)

Spend some time looking into different rendering approaches and design your game accordingly. Pay especial attention to sorting. Define the lowest end supported devices early in the dev cycle. Test on them with the profiler on as you design your game.

Use platform specific texture compression.

**Further reading**

- PowerVR SGX Architecture Guide http://imgtec.com/powervr/insider/powervr-sdk-docs.asp
- Tegra GLES2 feature guide http://developer.download.nvidia.com/tegra/docs/tegra_gles2_development.pdf
- Qualcomm Adreno GLES performance guide http://developer.qualcomm.com/file/607 /adreno200performanceoptimizationopenglestipsandtricksmarch10.pdf
- Engel, Rible http://altdevblogaday.com/2011/08/04/programming-the-xperia-play-gpu-by-wolfgang-engel-and-maurice-ribble/
- ARM Mali GPU Optimization guide http://www.malideveloper.com/developer-resources/documentation/index.php

**Screen resolution**

**Android version**

## iOS

**GPU**

Only PowerVR architecture (tile based deferred) to be concerned about.

- ImgTec PowerVR SGX. Tile based, deferred: render everything in tiles, shade only visible pixels
- ImgTec .PowerVR MBX. Tile based, deferred, fixed function - pre iPhone 4/iPad 1 devices

This means:

- Mipmaps are not so necessary.
- Antialiasing and aniso are cheap enough, not needed on iPad 3 in some cases

And cons:

- If vertex data per frame (number of vertices * storage required after vertex shader) exceeds the internal buffers allocated by the driver, the scene has to be �split� which costs performance. The driver might allocate a larger buffer after this point, or you might need to reduce your vertex count. This becomes apparent on iPad2 (iOS 4.3) at around 100 thousand

vertices with quite complex shaders.
- TBDR needs more transistors allocated for the tiling and deferred parts, leaving conceptually less transistors for �raw performance�. It�s very hard (i.e. practically impossible) to get GPU timing for a draw call on TBDR, making profiling hard.

#### Further reading

- PowerVR SGX Architecture Guide http://imgtec.com/powervr/insider/powervr-sdk-docs.asp

#### Screen resolution

#### iOS version

### Dynamic Objects

#### Asset Bundles

- Asset Bundles are cached on a device to a certain limit
- Create using the Editor API
- Load
  - Using WWW API: WWW.LoadFromCacheOrDownload
  - As a resource: AssetBundle.CreateFromMemory or AssetBundle.CreateFromFile
- Unload
  - AssetBundle.Unload
    - There is an option to unload the bundle, but keep the loaded asset from it
    - Also can kill all the loaded assets even if they�re referenced in the scene
  - Resources.UnloadUnusedAssets
    - Unloads all assets no longer referenced in the scene. So remember to kill references to the assets you don�t need.
    - Public and static variables are never garbage collected.
  - Resources.UnloadAsset
    - Unloads a specific asset from memory. It can be reloaded from disk if needed.

#### Is there any limitation for download numbers of Assetbundle at the same time on iOS? (e.g Can we download over 10 assetbundles safely at the same time(or every frame)? )

Downloads are implemented via async API provided by OS, so OS decides how many threads need to be created for downloads. When launching multiple concurrent downloads you should keep in mind total device bandwidth it can support and amount of free memory. Each concurrent download allocates its own temporal buffer, so you should be careful there to not run out of memory.

#### Resources

- Assets need to be recognized by Unity to be placed in a build.
- Add .bytes file extension to any raw bytes you want Unity to recognize as a binary data.
- Add .txt file extension to any text files you want Unity to recognize as a text asset
- Resources are converted to a platform format at a build time.
- Resources.Load()

### Silly issues checklist

- Textures without proper compression
  - Different solutions for different cases, but be sure to compress textures unless you�re sure you should not.
  - ETC/RGBA16 - default for android
    - but can tweak depending on the GPU vendor
    - best approach is to use ETC where possible
    - alpha textures can use two ETC files with one channel being for alpha
  - PVRTC - default for iOS
    - good for most cases
- Textures having Get/Set pixels enabled - doubles the footprint, uncheck unless Get/Set is needed
- Textures loaded from JPEG/PNGs on the runtime will be uncompressed
- Big mp3 files marked as decompress on load
- Additive scene loading
- Unused Assets that remain uncleaned in memory
  - Static fields
  - not unloaded asset bundles

- If it randomly crashes, try on a devkit or a device with 2 GB memory (like Ipad 3).

Sometimes there�s nothing in the console, just a random crash

- Fast script call and stripping may lead to random crashes on iOS. Try without them.

Page last updated: 2012-10-10

---

# Advanced


- Vector Cookbook
  - Understanding Vector Arithmetic
  - Direction and Distance from One Object to Another
  - Computing a Normal/Perpendicular vector
  - The Amount of One Vector's Magnitude that Lies in Another Vector's Direction
- AssetBundles (Pro only)
  - AssetBundles FAQ
  - Building AssetBundles
  - Downloading AssetBundles
  - Loading resources from AssetBundles
  - Keeping track of loaded AssetBundles
  - Storing and loading binary data in an AssetBundle
  - Protecting Content
  - Managing asset dependencies
  - Including scripts in AssetBundles
- Graphics Features
  - HDR (High Dynamic Range) Rendering in Unity
  - Rendering Paths
  - Linear Lighting (Pro Only)
  - Level of Detail (Pro Only)
  - Shaders
    - Shaders: ShaderLab & Fixed Function shaders
    - Shaders: Vertex and Fragment Programs
  - Using DirectX 11 in Unity 4
  - Compute Shaders
  - Graphics Emulation
- AssetDatabase
- Build Player Pipeline
- Profiler (Pro only)
- Lightmapping Quickstart
  - Lightmapping In-Depth
  - Custom Beast Settings
  - Lightmapping UVs
  - Light Probes
- Occlusion Culling (Pro only)
- Camera Tricks
  - UnderstandingFrustum
  - The Size of the Frustum at a Given Distance from the Camera
  - Dolly Zoom (AKA the "Trombone" Effect)
  - Rays from the Camera
  - Using an Oblique Frustum
  - Creating an Impression of Large or Small Size
- Loading Resources at Runtime
- Modifying Source Assets Through Scripting
- Generating Mesh Geometry Procedurally
  - Anatomy of a Mesh
  - Using the Mesh Class
  - Example - Creating a Billboard Plane
- Rich Text
- Using Mono DLLs in a Unity Project

- Execution Order of Event Functions
- Practical Guide to Optimization for Mobiles
  - Practical Guide to Optimization for Mobiles - Future & High End Devices
  - Practical Guide to Optimization for Mobiles - Graphics Methods
  - Practical Guide to Optimization for Mobiles - Scripting and Gameplay Methods
  - Practical Guide to Optimization for Mobiles - Rendering Optimizations
  - Practical Guide to Optimization for Mobiles - Optimizing Scripts
- Optimizing Graphics Performance
  - Draw Call Batching
  - Modeling Characters for Optimal Performance
  - Rendering Statistics Window
- Reducing File Size
- Understanding Automatic Memory Management
- Platform Dependent Compilation
- Generic Functions
- Debugging
  - Console
  - Debugger
  - Log Files
    - Accessing hidden folders
- Plugins (Pro/Mobile-Only Feature)
  - Building Plugins for Desktop Platforms
  - Building Plugins for iOS
  - Building Plugins for Android
  - Low-level Native Plugin Interface
- Textual Scene File Format (Pro-only Feature)
  - Description of the Format
  - YAMLSceneExample
  - YAML Class ID Reference
- Streaming Assets
- Command line arguments
- Running Editor Script Code on Launch
- Network Emulation
- Security Sandbox of the Webplayer
- Overview of available .NET Class Libraries
- Visual Studio C# Integration
- Using External Version Control Systems with Unity
- Analytics
- Check For Updates
- Installing Multiple Versions of Unity
- Trouble Shooting
- Shadows in Unity
  - Directional Shadow Details
  - Troubleshooting Shadows
  - Shadow Size Computation
- IME in Unity
- Optimizing for integrated graphics cards
- Web Player Deployment
  - HTML code to load Unity content
  - Working with UnityObject2
  - Customizing the Unity Web Player loading screen
  - Customizing the Unity Web Player's Behavior
  - Unity Web Player and browser communication
  - Using web player templates
  - Web Player Streaming
  - Webplayer Release Channels


Page last updated: 2007-11-16

---

# Vector Cookbook

Vector Cookbook

Although vector operations are easy to easy to describe, they are surprisingly subtle and powerful and have many uses in games programming. The following pages offer some suggestions about using vectors effectively in your code.


- Understanding Vector Arithmetic
- Direction and Distance from One Object to Another
- Computing a Normal/Perpendicular vector
- The Amount of One Vector's Magnitude that Lies in Another Vector's Direction


Page last updated: 2011-08-26

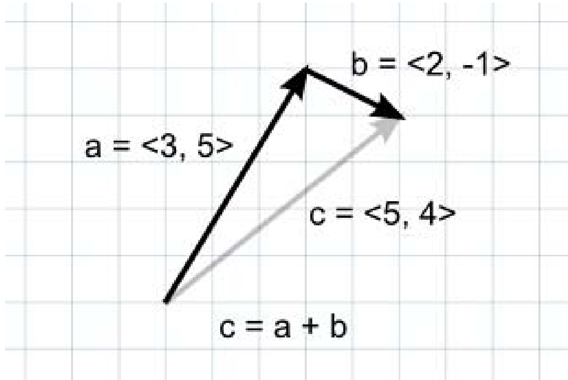
---

# UnderstandingVectorArithmetic

Vector arithmetic is fundamental to 3D graphics, physics and animation and it is useful to understand it in depth to get the most out of Unity. Below are descriptions of the main operations and some suggestions about the many things they can be used for.

### Addition

When two vectors are added together, the result is equivalent to taking the original vectors as "steps", one after the other. Note that the order of the two parameters doesn't matter, since the result is the same either way.



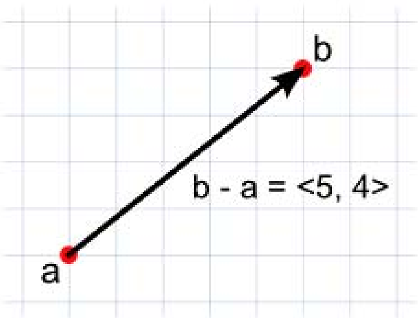

If the first vector is taken as a point in space then the second can be interpreted as an offset or "jump" from that position. For example, to find a point 5 units above a location on the ground, you could use the following calculation:-

```
var pointInAir = pointOnGround + new Vector3(0, 5, 0);
```

If the vectors represent forces then it is more intuitive to think of them in terms of their direction and magnitude (the magnitude indicates the size of the force). Adding two force vectors results in a new vector equivalent to the combination of the forces. This concept is often useful when applying forces with several separate components acting at once (eg, a rocket being propelled forward may also be affected by a crosswind).

### Subtraction

Vector subtraction is most often used to get the direction and distance from one object to another. Note that the order of the two parameters **does** matter with subtraction:-




```
// The vector d has the same magnitude as c but points in the opposite direction.
var c = b - a;
var d = a - b;
```

As with numbers, adding the negative of a vector is the same as subtracting the positive.

```
// These both give the same result.
var c = a - b;
var c = a + -b;
```

The negative of a vector has the same magnitude as the original and points along the same line but in the exact opposite direction.

### Scalar Multiplication and Division

When discussing vectors, it is common to refer to an ordinary number (eg, a float value) as a scalar. The meaning of this is that a scalar only has "scale" or magnitude whereas a vector has both magnitude and direction.

Multiplying a vector by a scalar results in a vector that points in the same direction as the original. However, the new vector's magnitude is equal to the original magnitude multiplied by the scalar value.

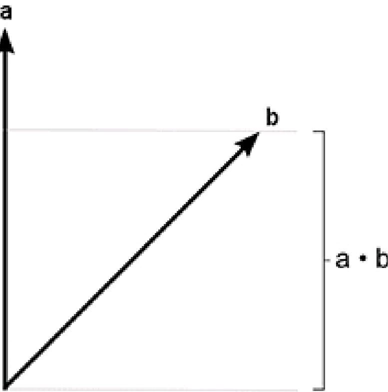
Likewise, scalar division divides the original vector's magnitude by the scalar.

These operations are useful when the vector represents a movement offset or a force. They allow you to change the magnitude of the vector without affecting its direction.

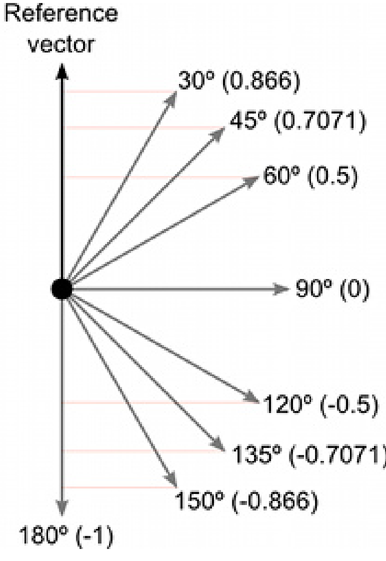
When any vector is divided by its own magnitude, the result is a vector with a magnitude of 1, which is known as a normalized vector. If a normalized vector is multiplied by a scalar then the magnitude of the result will be equal to that scalar value. This is useful when the direction of a force is constant but the strength is controllable (eg, the force from a car's wheel always pushes forwards but the power is controlled by the driver).

### Dot Product

The dot product takes two vectors and returns a scalar. This scalar is equal to the magnitudes of the two vectors multiplied together and the result multiplied by the cosine of the angle between the vectors. When both vectors are normalized, the cosine essentially states how far the first vector extends in the second's direction (or vice-versa - the order of the parameters doesn't matter).




It is easy enough to think in terms of angles and then find the corresponding cosines using a calculator. However, it is useful to get an intuitive understanding of some of the main cosine values as shown in the diagram below:-

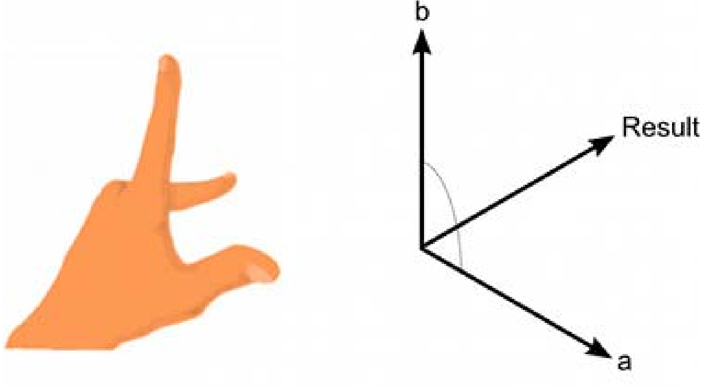



The dot product is a very simple operation that can be used in place of the Mathf.Cos function or the vector magnitude operation in some circumstances (it doesn't do exactly the same thing but sometimes the effect is equivalent). However, calculating the dot product function takes much less CPU time and so it can be a valuable optimization.

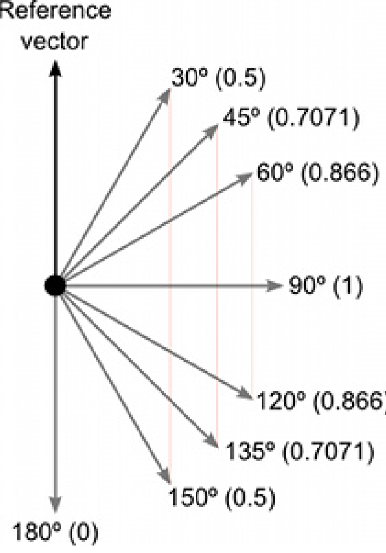
### Cross Product

The other operations are defined for 2D and 3D vectors and indeed vectors with any number of dimensions. The cross product, by contrast, is only meaningful for 3D vectors. It takes two vectors as input and returns another vector as its result.

The result vector is perpendicular to the two input vectors. The "left hand rule" can be used to remember the direction of the output vector from the ordering of the input vectors. If the first parameter is matched up to the thumb of the hand and the second parameter to the forefinger, then the result will point in the direction of the middle finger. If the order of the parameters is reversed then the resulting vector will point in the exact opposite direction but will have the same magnitude.



The magnitude of the result is equal to the magnitudes of the input vectors multiplied together and then that value multiplied by the sine of the angle between them. Some useful values of the sine function are shown below:-




The cross product can seem complicated since it combines several useful pieces of information in its return value. However, like the dot product, it is very efficient mathematically and can be used to optimize code that would otherwise depend on slow transcendental functions.

Page last updated: 2011-08-26

---

## DirectionDistanceFromOneObjectToAnother

If one point in space is subtracted from another then the result is a vector that "points" from one object to the other:

```
// Gets a vector that points from the player's position to the target's.
var heading = target.position - player.position;
```

As well as pointing in the direction of the target object, this vector's magnitude is equal to the distance between the two positions. It is common to need a normalized vector giving the direction to the target and also the distance to the target (say for directing a projectile). The distance between the objects is equal to the magnitude of the heading vector and this vector can be normalized by dividing it by its magnitude:-

```
var distance = heading.magnitude;
var direction = heading / distance;  // This is now the normalized direction.
```

This approach is preferable to using the both the magnitude and normalized properties separately, since they are both quite CPU-hungry (they both involve calculating a square root).

If you only need to use the distance for comparison (for a proximity check, say) then you can avoid the magnitude calculation altogether. The sqrMagnitude property gives the square of the magnitude value, and is calculated like the magnitude but without the time-consuming square root operation. Rather than compare the magnitude against a known distance, you can compare the squared magnitude against the squared distance:-

```
if (heading.sqrMagnitude < maxRange * maxRange) {
      // Target is within range.
}
```

This is much more efficient than using the true magnitude in the comparison.

Sometimes, the overground heading to a target is required. For example, imagine a player standing on the ground who needs to approach a target floating in the air. If you subtract the player's position from the target's then the resulting vector will point upwards towards the target. This is not suitable for orienting the player's transform since he will also point upwards; what is really needed is a vector from the player's position to the position on the ground directly below the target. This is easily obtained by taking the result of the subtraction and setting the Y coordinate to zero:-

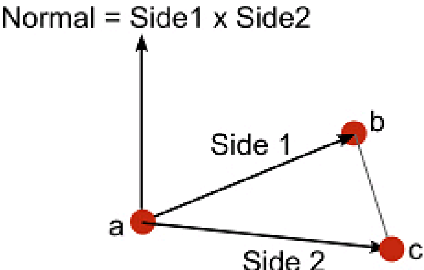
```
var heading = target.position - player.position;
heading.y = 0;    // This is the overground heading.
```

Page last updated: 2011-08-26

---

## ComputingNormalPerpendicularVector

A normal vector (ie, a vector perpendicular to a plane) is required frequently during mesh generation and may also be useful in path following and other situations. Given three points in the plane, say the corner points of a mesh triangle, it is easy to find the normal. Pick any of the three points and then subtract it from each of the two other points separately to give two vectors:-




```
var a: Vector3;
var b: Vector3;
var c: Vector3;

var side1: Vector3 = b - a;
var side2: Vector3 = c - a;
```

The cross product of these two vectors will give a third vector which is perpendicular to the surface. The "left hand rule" can be used to decide the order in which the two vectors should be passed to the cross product function. As you look down at the top side of the surface (from which the normal will point outwards) the first vector should sweep around clockwise to the second:-

```
var perp: Vector3 = Vector3.Cross(side1, side2);
```

The result will point in exactly the opposite direction if the order of the input vectors is reversed.

For meshes, the normal vector must also be normalized. This can be done with the normalized property, but there is another trick which is occasionally useful. You can also normalize the perpendicular vector by dividing it by its magnitude:-

```
var perpLength = perp.magnitude;
perp /= perpLength;
```

It turns out that the area of the triangle is equal to perpLength / 2. This is useful if you need to find the surface area of the whole mesh or want to choose triangles randomly with probability based on their relative areas.

Page last updated: 2011-08-26

---

## AmountVectorMagnitudeInAnotherDirection

A car's speedometer typically works by measuring the rotational speed of one of the unpowered wheels. The car may not be moving directly forward (it may be skidding sideways, for example) in which case part of the motion will not be in the direction the speedometer can measure. The magnitude of an object's rigidbody.velocity vector will give the speed in its direction of overall motion but to isolate the speed in the forward direction, you should use the dot product:-

```
var fwdSpeed = Vector3.Dot(rigidbody.velocity, transform.forward);
```

Naturally, the direction can be anything you like but the direction vector must always be normalized for this calculation. Not only is the result more correct than the magnitude of the velocity, it also avoids the slow square root operation involved in finding the magnitude.

Page last updated: 2011-08-26

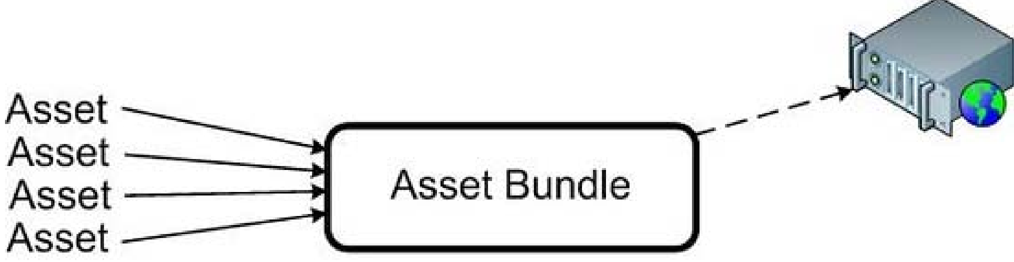
---

## AssetBundles

AssetBundles are files which you can export from Unity to contain assets of your choice. These files use a proprietary compressed format and can be loaded on demand by your application. This allows you to stream in content, such as models, textures, audio clips, or even entire scenes separately from the scene in which they will be used. AssetBundles have been designed to simplify downloading content to your application. AssetBundles can contain any kind of asset type recognized by Unity, as determined by the filename extension. If you want to include files with custom binary data, they should have the extension ".bytes". Unity will import these files as TextAssets.

When working with AssetBundles, here's the typical workflow:

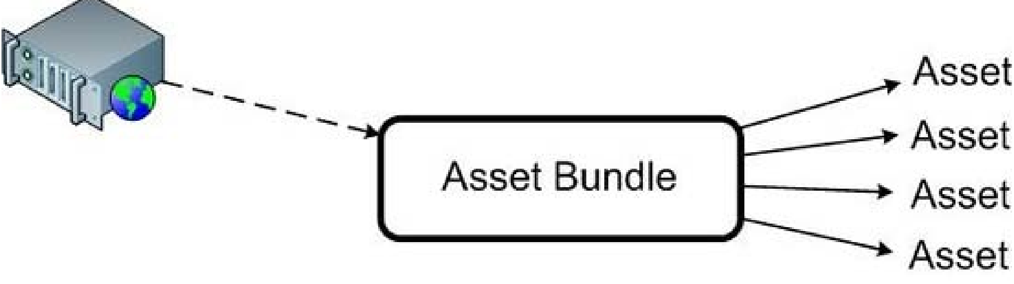
*During development*, the developer prepares AssetBundles and uploads them to a server.




Building and uploading asset bundles

1. **Building AssetBundles**. Asset bundles are created in the editor from assets in your scene. The Asset Bundle building process is described in more detail in the section for Building AssetBundles
2. **Uploading AssetBundles to external storage**. This step does not include the Unity Editor or any other Unity channels, but we include it for completeness. You can use an FTP client to upload your Asset Bundles to the server of your choice.

*At runtime*, on the user's machine, the application will load AssetBundles on demand and operate individual assets within each AssetBundle as needed.



```
Downloading AssetBundles and loading assets from them
```

1. **Downloading AssetBundles at runtime from your application**. This is done from script within a Unity scene, and Asset Bundles are loaded from the server on demand. More on that in Downloading Asset Bundles.
2. **Loading objects from AssetBundles**. Once the AssetBundle is downloaded, you might want to access its individual Assets from the Bundle. More on that in Loading Resources from AssetBundles

See also:


- Frequently Asked Questions
- Building AssetBundles
- Downloading Asset Bundles
- Loading Asset Bundles
- Keeping track of loaded AssetBundles
- Storing and loading binary data
- Protecting content
- Managing Asset Dependencies
- Including scripts in AssetBundles


Page last updated: 2012-10-10

---

## Frequently Asked Questions


1. What are AssetBundles?
2. What are they used for?
3. How do I create an AssetBundle?
4. How do I use an AssetBundle?
5. How do I use AssetBundles in the Editor?
6. How do I cache AssetBundles?
7. Are AssetBundles cross-platform?
8. How are assets in AssetBundles identified
9. Can I reuse my AssetBundles in another game?
10. Will an AssetBundle built now be usable with future versions of Unity?
11. How can I list the objects in an AssetBundle?


---

1. What are AssetBundles?

AssetBundles are a collection of assets, packaged for loading at runtime. With Asset Bundles, you can dynamically load and unload new content into your application. AssetBundles can be used to implement post-release DLC.

2. What are they used for?

They can be used to reduce the amount of space on disk used by your game, when first deployed. It can also be used to add new content to an already published game.

3. How do I create an AssetBundle?

To create an AssetBundle you need to use the BuildPipeline editor class. All scripts using Editor classes must be placed in a folder named Editor, anywhere in the Assets folder. Here is an example of such a script in C#:

```
+ Show [Creating an AssetBundle] +
```

4. How do I use an AssetBundle?

There are two main steps involved when working with AssetBundles. The first step is to download the AssetBundle from a server or disk location. This is done with the WWW class. The second step is to load the Assets from the AssetBundle, to be used in the application. Here is an example C# script:

```
+ Show [Using an AssetBundle] +
```

5. How do I use AssetBundles in the Editor?

As creating applications is an iterative process, you will very likely modify your Assets many times, which would require rebuilding the AssetBundles after every change to be able to test them. Even though it is possible to load AssetBundles in the Editor, that is not the recommended workflow. Instead, while testing in the Editor you should use the helper function Resources.LoadAssetAtPath to avoid having to use and rebuild AssetBundles. The function lets you load the Asset as if it were being loaded from an AssetBundle, but will skip the building process and your Assets are always up to date.

The following is an example helper script, that you can use to load your Assets depending on if you are running in the Editor or not. Put this code in C# script named AssetBundleLoader.cs:

```
+ Show [Using an AssetBundle in the Editor] +
```

6. How do I cache AssetBundles?

You can use WWW.LoadFromCacheOrDownload which automatically takes care of saving your AssetBundles to disk. Be aware that on the Webplayer you are limited to 50MB in total (shared between all webplayers). You can buy a separate caching license for your game if you require more space.

7. Are AssetBundles cross-platform?

AssetBundles are compatible between some platforms. Use the following table as a guideline.

**Platform compatibility for AssetBundles**

| | Standalone | Webplayer | iOS | Android |
|---|---|---|---|---|
| Editor | Y | Y | Y | Y |
| Standalone | Y | Y | | |
| Webplayer | Y | Y | | |
| iOS | | | Y | |
| Android | | | | Y |

For example, a bundle created while the Webplayer build target was active would be compatible with the editor and with standalone builds. However, it would not be compatible with apps built for the iOS or Android platforms.

8. How are assets in AssetBundles identified?

When you build AssetBundles the assets are identified internally by their filename without the extension. For example a Texture located in your Project folder at "Assets/Textures/myTexture.jpg" is identified and loaded using "myTexture" if you use the default method. You can have more control over this by supplying your own array of ids (strings) for each object when Building your AssetBundle with BuildPipeline.BuildAssetBundleExplicitAssetNames.

9. Can I reuse my AssetBundles in another game?

AssetBundles allow you to share content between different games. The requirement is that any Assets which are referenced by GameObjects in your AssetBundle must either be included in the AssetBundle or exist in the application (loaded in the current scene). To make sure the referenced Assets are included in the AssetBundle when they are built you can pass the BuildAssetBundleOptions.CollectDependencies option.

10. Will an AssetBundle built now be usable with future versions of Unity?

AssetBundles can contain a structure called a **type tree** which allows information about asset types to be understood correctly between different versions of Unity. On desktop platforms, the type tree is included by default but can be disabled by passing the BuildAssetBundleOptions.DisableWriteTypeTree to the BuildAssetBundle function. Webplayers intrinsically rely on the type tree and so it is always included (ie, the DisableWriteTypeTree option has no effect). Type trees are never included for mobile and console asset bundles and so you will need to rebuild these bundles whenever the serialization format changes. This can happen in new versions of Unity. (Except for bugfix releases) It also happens if you add or remove serialized fields in monobehaviour's that are included in the asset bundle. When loading an AssetBundle Unity will give you an error message if the AssetBundle must be rebuilt.

11. How can I list the objects in an AssetBundle?

You can use AssetBundle.LoadAll to retrieve an array containing all objects from the AssetBundle. It is not possible to get a list of the identifiers directly. A common workaround is to keep a separate TextAsset to hold the names of the assets in the

AssetBundle.

back to AssetBundles Intro

Page last updated: 2012-09-13

---

# Building AssetBundles

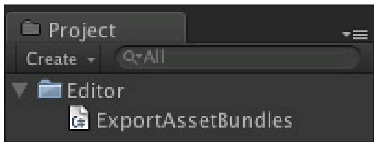
There are three class methods you can use to build AssetBundles:

- BuildPipeline.BuildAssetBundle allows you to build AssetBundles of any type of asset.
- BuildPipeline.BuildStreamedSceneAssetBundle is used when you want to include only scenes to be streamed and loaded as the data becomes available.
- BuildPipeline.BuildAssetBundleExplicitAssetNames is the same as BuildPipeline.BuildAssetBundle but has an extra parameter to specify a custom string identifier (name) for each object.

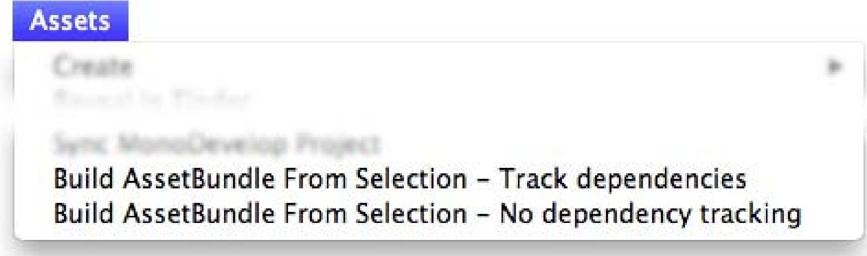
### An example of how to build an AssetBundle

Building asset bundles is done through editor scripting. There is basic example of this in the scripting documentation for BuildPipeline.BuildAssetBundle.

For the sake of this example, copy and paste the script from the link above into a new C# script called ExportAssetBundles. This script should be placed in a folder named Editor, so that it works inside the Unity Editor.



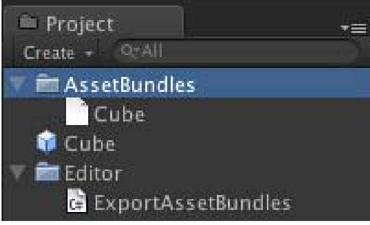

Now in the **Assets** menu, you should see two new menu options.




1. **Build AssetBundle From Selection - Track dependencies**. This will build the current object into an asset bundle and include all of its dependencies. For example if you have a prefab that consists of several hierarchical layers then it will recursively add all the child objects and components to the asset bundle.
2. **Build AssetBundle From Selection - No dependency tracking**. This is the opposite of the previous and will only include the single asset you have selected.

For this example, you should create a new prefab. First create a new Cube by going to **GameObject -> Create Other -> Cube**, which will create a new cube in the Hierarchy View. Then drag the Cube from the Hierarchy View into the Project View, which will create a prefab of that object.

You should then right click the Cube prefab in the project window and select **Build AssetBundle From Selection - Track dependencies**. At this point you will be presented with a window to save the �bundled� asset. If you created a new folder called "AssetBundles" and saved the cube as **Cube.unity3d**, your project window will now look something like this.




At this point you can move the AssetBundle **Cube.unity3d** elsewhere on your local storage, or upload it to a server of your choice.

### Building AssetBundles in a production enviroment

When first using AssetBundles it may seem enough to manually build them as seen in the previous example. But as a project grows in size and the number of Assets increases doing this process by hand is not efficient. A better approach is to write a function that builds all of the AssetBundles for a project. You can for example use a text file that maps Asset files to AssetBundle files.

back to AssetBundles Intro

Page last updated: 2012-09-04

---

# Downloading AssetBundles

### Downloading AssetBundles

This section assumes you already learned how to build asset bundles. If you have not, please see Building AssetBundles

There are two ways to download an AssetBundle

1. **Non-caching:** This is done using a creating a new WWW object. The AssetBundles are not cached to Unity�s Cache folder in the local storage device.
2. **Caching:** This is done using the WWW.LoadFromCacheOrDownload call. The AssetBundles are cached to Unity�s Cache folder in the local storage device. The WebPlayer shared cache allows up to 50 MB of cached AssetBundles. PC/Mac Standalone applications and iOS/Android applications have a limit of 4 GB. WebPlayer applications that make use of a dedicated cache are limited to the number of bytes specified in the caching license agreement. Please refer to the scripting documentation for other platforms.

Here's an example of a non-caching download:

```
using System;
using UnityEngine;
using System.Collections; class NonCachingLoadExample : MonoBehaviour {
   public string BundleURL;
   public string AssetName;
   IEnumerator Start() {
         // Download the file from the URL. It will not be saved in the Cache
         using (WWW www = new WWW(BundleURL)) {
               yield return www;
               if (www.error != null)
                     throw new Exception("WWW download had an error:" + www.error);
               AssetBundle bundle = www.assetBundle;
               if (AssetName == "")
                     Instantiate(bundle.mainAsset);
               else
                     Instantiate(bundle.Load(AssetName));
               // Unload the AssetBundles compressed contents to conserve memory
               bundle.Unload(false);
         }
```

```
    }
}
```

The recommended way to download AssetBundles is to use WWW.LoadFromCacheOrDownload. For example:

```
using System;
using UnityEngine;
using System.Collections;

public class CachingLoadExample : MonoBehaviour {
	public string BundleURL;
	public string AssetName;
	public int version;

	void Start() {
		StartCoroutine (DownloadAndCache());
	}

	IEnumerator DownloadAndCache (){
		// Wait for the Caching system to be ready
		while (!Caching.ready)
			yield return null;

		// Load the AssetBundle file from Cache if it exists with the same version or download and store it in the cache
		using(WWW www = WWW.LoadFromCacheOrDownload (BundleURL, version)){
			yield return www;
			if (www.error != null)
				throw new Exception("WWW download had an error:" + www.error);
			AssetBundle bundle = www.assetBundle;
			if (AssetName == "")
				Instantiate(bundle.mainAsset);
			else
				Instantiate(bundle.Load(AssetName));
			// Unload the AssetBundles compressed contents to conserve memory
			bundle.Unload(false);
		}
	}
}
```

When you access the `.assetBundle` property, the downloaded data is extracted and the AssetBundle object is created. At this point, you are ready to load the objects contained in the bundle. The second parameter passed to LoadFromCacheOrDownload specifies which version of the AssetBundle to download. If the AssetBundle doesn't exist in the cache or has a version lower than requested, LoadFromCacheOrDownload will download the AssetBundle. Otherwise the AssetBundle will be loaded from cache.

**Putting it all together**

Now that the components are in place you can build a scene that will allow you to load your AssetBundle and display the contents on screen.

Project
Create
All
AssetBundles
Cube
Cube
Editor
ExportAssetBundles
Scripts
CachingLoadExample
NonCachingLoadExample

Final project structure

First create an empty game object by going to **GameObject->CreateEmpty**. Drag the CachingLoadExample script onto the empty game object you just created. Then type the the URL of your AssetBundle in the BundleURL field. As we have placed this in the project directory you can copy the file directory location and add the prefix `file://`, for example `file://C:/UnityProjects/AssetBundlesGuide/Assets/AssetBundles/Cube.unity3d`

You can now hit play in the Editor and you should see the Cube prefab being loaded from the AssetBundle.

**Loading AssetBundles in the Editor**

When working in the Editor requiring AssetBundles to be built and loaded can slow down the development process. For instance, if an Asset from an AssetBundle is modified this will then require the AssetBundle to be rebuilt and in a production environment it is most likely that all AssetBundles are built together and therefore making the process of updating a single AssetBundle a lengthy operation. A better approach is to have a separate code path in the Editor that will load the Asset directly instead of loading it from an AssetBundle. To do this it is possible to use Resources.LoadAssetAtPath (Editor only).

```
// C# Example
// Loading an Asset from disk instead of loading from an AssetBundle
// when running in the Editor
using System.Collections;
using UnityEngine;

class LoadAssetFromAssetBundle : MonoBehaviour
{
	public Object Obj;

	public IEnumerator DownloadAssetBundle<T>(string asset, string url, int version) where T : Object {
		Obj = null;
#if UNITY_EDITOR
		Obj = Resources.LoadAssetAtPath("Assets/" + asset, typeof(T));
		yield return null;
#else
		// Wait for the Caching system to be ready
		while (!Caching.ready)
			yield return null;

		// Start the download
		using(WWW www = WWW.LoadFromCacheOrDownload (url, version)){
			yield return www;
			if (www.error != null)
				throw new Exception("WWW download:" + www.error);
			AssetBundle assetBundle = www.assetBundle;
			Obj = assetBundle.Load(asset, typeof(T));
			// Unload the AssetBundles compressed contents to conserve memory
			bundle.Unload(false);
		}
#endif
	}
}
```

back to AssetBundles Intro

Page last updated: 2012-08-16

---

## Loading resources from AssetBundles

### Loading and unloading objects from an AssetBundle

Having created an AssetBundle object from the downloaded data, you can load the objects contained in it using three different methods:

- AssetBundle.Load will load an object using its name identifier as a parameter. The name is the one visible in the Project view. You can optionally pass an object type as an argument to the Load method to make sure the object loaded is of a specific type.
- AssetBundle.LoadAsync works the same as the Load method described above but it will not block the main thread while the asset is loaded. This is useful when loading large assets or many assets at once to avoid pauses in your application.
- AssetBundle.LoadAll will load all the objects contained in your AssetBundle. As with AssetBundle.Load, you can optionally filter objects by their type.

To unload assets you need to use AssetBundle.Unload. This method takes a boolean parameter which tells Unity whether to unload all data (including the loaded asset objects) or only the compressed data from the downloaded bundle. If your application is using some objects from the AssetBundle and you want to free some memory you can pass false to unload the compressed data from memory. If you want to completely unload everything from the AssetBundle you should pass true which will destroy the Assets loaded from the AssetBundle.

### Loading objects from an AssetBundles asynchronously

You can use the AssetBundle.LoadAsync method to load objects Asynchronously and reduce the likelihood of having hiccups in your application.

```
using UnityEngine;

// Note: This example does not check for errors. Please look at the example in the DownloadingAssetBundles section for more
IEnumerator Start () {
	// Start a download of the given URL
	WWW www = WWW.LoadFromCacheOrDownload (url, 1);

	// Wait for download to complete
	yield return www;

	// Load and retrieve the AssetBundle
	AssetBundle bundle = www.assetBundle;

	// Load the object asynchronously
	AssetBundleRequest request = bundle.LoadAsync ("myObject", typeof(GameObject));

	// Wait for completion
	yield return request;

	// Get the reference to the loaded object
	GameObject obj = request.asset as GameObject;

	// Unload the AssetBundles compressed contents to conserve memory
	bundle.Unload(false);
}
```

back to AssetBundles Intro

Page last updated: 2012-08-14

---

## Keeping track of loaded AssetBundles

### Keeping Track of loaded AssetBundles

Unity will only allow you to have a single instance of a particular AssetBundle loaded at one time in your application. What this

means is that you can�t retrieve an AssetBundle from a WWW object if the same one has been loaded previously and has not been unloaded. In practical terms it means that when you try to access a previously loaded AssetBundle like this:

```
AssetBundle bundle = www.assetBundle;
```

the following error will be thrown

```
Cannot load cached AssetBundle. A file of the same name is already loaded from another AssetBundle
```

and the assetBundle property will return null. Since you can�t retrieve the AssetBundle during the second download if the first one is still loaded, what you need to do is to either *unload* the AssetBundle when you are no longer using it, or maintain a reference to it and avoid downloading it if it is already in memory. You can decide the right course of action based on your needs, but our recommendation is that you *unload* the AssetBundle as soon as you are done loading objects. This will free the memory and you will no longer get an error about loading cached AssetBundles.

If you do want to keep track of which AssetBundles you have downloaded, you could use a wrapper class to help you manage your downloads like the following:

```
using UnityEngine;
using System;
using System.Collections;
using System.Collections.Generic;

static public class AssetBundleManager {
   // A dictionary to hold the AssetBundle references
   static private Dictionary<string, AssetBundleRef> dictAssetBundleRefs;
   static AssetBundleManager (){
       dictAssetBundleRefs = new Dictionary<string, AssetBundleRef>();
   }
   // Class with the AssetBundle reference, url and version
   private class AssetBundleRef {
       public AssetBundle assetBundle = null;
       public int version;
       public string url;
       public AssetBundleRef(string strUrlIn, int intVersionIn) {
           url = strUrlIn;
           version = intVersionIn;
       }
   };
   // Get an AssetBundle
   public static AssetBundle getAssetBundle (string url, int version){
       string keyName = url + version.ToString();
       AssetBundleRef abRef;
       if (dictAssetBundleRefs.TryGetValue(keyName, out abRef))
           return abRef.assetBundle;
       else
           return null;
   }
   // Download an AssetBundle
   public static IEnumerator downloadAssetBundle (string url, int version){
       string keyName = url + version.ToString();
       if (dictAssetBundleRefs.ContainsKey(keyName))
           yield return null;
       else {
           using(WWW www = WWW.LoadFromCacheOrDownload (url, version)){
               yield return www;
               if (www.error != null)
                   throw new Exception("WWW download:" + www.error);
               AssetBundleRef abRef = new AssetBundleRef (url, version);
```

```
                abRef.assetBundle = www.assetBundle;
                dictAssetBundleRefs.Add (keyName, abRef);
            }
        }
    }
    // Unload an AssetBundle
    public static void Unload (string url, int version, bool allObjects){
        string keyName = url + version.ToString();
        AssetBundleRef abRef;
        if (dictAssetBundleRefs.TryGetValue(keyName, out abRef)){
            abRef.assetBundle.Unload (allObjects);
            abRef.assetBundle = null;
            dictAssetBundleRefs.Remove(keyName);
        }
    }
}
```

An example usage of the class would be:

```
using UnityEditor;

class ManagedAssetBundleExample : MonoBehaviour {
    public string url;
    public int version;
    AssetBundle bundle;
    void OnGUI (){
        if (GUILayout.Label ("Download bundle"){
            bundle = AssetBundleManager.getAssetBundle (url, version);
            if(!bundle)
                StartCoroutine (DownloadAB());
        }
    }
    IEnumerator DownloadAB (){
        yield return StartCoroutine(AssetBundleManager.downloadAssetBundle (url, version));
        bundle = AssetBundleManager.getAssetBundle (url, version);
    }
    void OnDisable (){
        AssetBundleManager.Unload (url, version);
    }
}
```

Please bear in mind, that the AssetBundleManager class in this example is static, and any AssetBundles that you are referencing will not be destroyed when loading a new scene. Use this class as a guide but as recommended initially it is best if you *unload* AssetBundles right after they have been used. You can always clone a previously Instantiated object, removing the need to load the AssetBundles again.

back to AssetBundles Intro

Page last updated: 2012-05-11

---

## Storing and loading binary data

The first step is to save your binary data file with the ".bytes" extension. Unity will treat this file as a TextAsset. As a TextAsset the file can be included when you build your AssetBundle. Once you have downloaded the AssetBundle in your application and loaded the TextAsset object, you can use the .bytes property of the TextAsset to retrieve your binary data.

```
string url = "http://www.mywebsite.com/mygame/assetbundles/assetbundle1.unity3d";
IEnumerator Start () {
    // Start a download of the given URL
    WWW www = WWW.LoadFromCacheOrDownload (url, 1);

    // Wait for download to complete
    yield return www;

    // Load and retrieve the AssetBundle
    AssetBundle bundle = www.assetBundle;

    // Load the TextAsset object
    TextAsset txt = bundle.Load("myBinaryAsText", typeof(TextAsset)) as TextAsset;

    // Retrieve the binary data as an array of bytes
    byte[] bytes = txt.bytes;
}
```

back to AssetBundles Intro

Page last updated: 2012-05-11

---

# Protecting content

Whilst it is possible to use encryption to secure your Assets as they are being transmitted, once the data is in the hands of the client it is possible to find ways to grab the content from them. For instance, there are tools out there which can record 3D data at the driver level, allowing users to extract models and textures as they are sent to the GPU. For this reason, our general stance is that if users are determined to extract your assets, they will be able to.

However, it is possible for you to use your own data encryption on AssetBundle files if you still want to.

One way to do this is making use of the TextAsset type to store your data as bytes. You can encrypt your data files and save them with a .bytes extension, which Unity will treat as a TextAsset type. Once imported in the Editor the files as TextAssets can be included in your AssetBundle to be placed in a server. In the client side the AssetBundle would be downloaded and the content decrypted from the bytes stored in the TextAsset. With this method the AssetBundles are not encrypted, but the data stored which is stored as TextAssets is.

```
string url = "http://www.mywebsite.com/mygame/assetbundles/assetbundle1.unity3d";
IEnumerator Start () {
    // Start a download of the encrypted assetbundle
    WWW www = new WWW.LoadFromCacheOrDownload (url, 1);

    // Wait for download to complete
    yield return www;

    // Load the TextAsset from the AssetBundle
    TextAsset textAsset = www.assetBundle.Load("EncryptedData", typeof(TextAsset));

    // Get the byte data
    byte[] encryptedData = textAsset.bytes;

    // Decrypt the AssetBundle data
    byte[] decryptedData = YourDecryptionMethod(encryptedData);

    // Use your byte array. The AssetBundle will be cached
}
```

An alternative approach is to fully encrypt the AssetBundles from source and then download them using the WWW class. You can give them whatever file extension you like as long as your server serves them up as binary data. Once downloaded you would then use your decryption routine on the data from the .bytes property of your WWW instance to get the decrypted AssetBundle file data and create the AssetBundle from memory using AssetBundle.CreateFromMemory.

```
string url = "http://www.mywebsite.com/mygame/assetbundles/assetbundle1.unity3d";
IEnumerator Start () {
    // Start a download of the encrypted assetbundle
    WWW www = new WWW (url);

    // Wait for download to complete
    yield return www;

    // Get the byte data
    byte[] encryptedData = www.bytes;

    // Decrypt the AssetBundle data
    byte[] decryptedData = YourDecryptionMethod(encryptedData);

    // Create an AssetBundle from the bytes array
    AssetBundle bundle = AssetBundle.CreateFromMemory(decryptedData);

    // You can now use your AssetBundle. The AssetBundle is not cached.
}
```

The advantage of this latter approach over the first one is that you can use any method (except AssetBundles.LoadFromCacheOrDownload) to transmit your bytes and the data is fully encrypted - for example sockets in a plugin. The drawback is that it won't be Cached using Unity's automatic caching. You can in all players except the WebPlayer store the file manually on disk and load it using AssetBundles.CreateFromFile

A third approach would combine the best of both approaches and store an AssetBundle itself as a TextAsset, inside another normal AssetBundles. The unencrypted AssetBundle containing the encrypted one would be cached. The original AssetBundle could then be loaded into memory, decrypted and instantiated using AssetBundle.CreateFromMemory.

```
string url = "http://www.mywebsite.com/mygame/assetbundles/assetbundle1.unity3d";
IEnumerator Start () {
    // Start a download of the encrypted assetbundle
    WWW www = new WWW.LoadFromCacheOrDownload (url, 1);

    // Wait for download to complete
    yield return www;

    // Load the TextAsset from the AssetBundle
    TextAsset textAsset = www.assetBundle.Load("EncryptedData", typeof(TextAsset));

    // Get the byte data
    byte[] encryptedData = textAsset.bytes;

    // Decrypt the AssetBundle data
    byte[] decryptedData = YourDecryptionMethod(encryptedData);

    // Create an AssetBundle from the bytes array
    AssetBundle bundle = AssetBundle.CreateFromMemory(decryptedData);

    // You can now use your AssetBundle. The wrapper AssetBundle is cached
}
```

back to AssetBundles Intro

Page last updated: 2012-09-04

## Managing Asset Dependencies

Any given asset in a bundle may depend on other assets. For example, a model may incorporate materials which in turn make use of textures and shaders. It is possible to include all an asset's dependencies along with it in its bundle. However, several assets from different bundles may all depend on a common set of other assets (eg, several different models of buildings may use the same brick texture). If a separate copy of a shared dependency is included in each bundle that has objects using it, then redundant instances of the assets will be created when the bundles are loaded. This will result in wasted memory.

To avoid such wastage, it is possible to separate shared dependencies out into a separate bundle and simply reference them from any bundles with assets that need them. First, the referencing feature needs to be enabled with a call to BuildPipeline.PushAssetDependencies. Then, the bundle containing the referenced dependencies needs to be built. Next, another call to PushAssetDependencies should be made before building the bundles that reference the assets from the first bundle. Additional levels of dependency can be introduced using further calls to PushAssetDependencies. The levels of reference are stored on a stack, so it is possible to go back a level using the corresponding BuildPipeline.PopAssetDependencies function. The push and pop calls need to be balanced including the initial push that happens before building.

At runtime, you need to load a bundle containing dependencies before any other bundle that references them. For example, you would need to load a bundle of shared textures before loading a separate bundle of materials that reference those textures.

Note that if you anticipate needing to rebuild asset bundles that are part of a dependency chain then you should build them with the BuildAssetBundleOptions.DeterministicAssetBundle option enabled. This guarantees that the internal ID values used to identify assets will be the same each time the bundle is rebuilt.

back to AssetBundles Intro

Page last updated: 2012-05-11

## Including scripts in AssetBundles

AssetBundles can contain scripts as TextAssets but as such they will not be actual executable code. If you want to include code in your AssetBundles that can be executed in your application it needs to be pre-compiled into an assembly and loaded using the Mono Reflection class (Note: Reflection is not available on iOS). You can create your assemblies in any normal C# IDE (e.g. Monodevelop, Visual Studio) or any text editor using the mono/.net compilers.

```
string url = "http://www.mywebsite.com/mygame/assetbundles/assetbundle1.unity3d";
IEnumerator Start () {
    // Start a download of the given URL
    WWW www = WWW.LoadFromCacheOrDownload (url, 1);

    // Wait for download to complete
    yield return www;

    // Load and retrieve the AssetBundle
    AssetBundle bundle = www.assetBundle;

    // Load the TextAsset object
    TextAsset txt = bundle.Load("myBinaryAsText", typeof(TextAsset)) as TextAsset;

    // Load the assembly and get a type (class) from it
    var assembly = System.Reflection.Assembly.Load(txt.bytes);
```

```
    var type = assembly.GetType("MyClassDerivedFromMonoBehaviour");

    // Instantiate a GameObject and add a component with the loaded class
    GameObject go = new GameObject();
    go.AddComponent(type);
}
```

back to AssetBundles Intro

Page last updated: 2012-05-11

---

# Graphics Features


- HDR (High Dynamic Range) Rendering in Unity
- Rendering Paths
- Linear Lighting (Pro Only)
- Level of Detail (Pro Only)
- Shaders
  - Shaders: ShaderLab & Fixed Function shaders
  - Shaders: Vertex and Fragment Programs
- Using DirectX 11 in Unity 4
- Compute Shaders
- Graphics Emulation


Page last updated: 2012-09-04

---

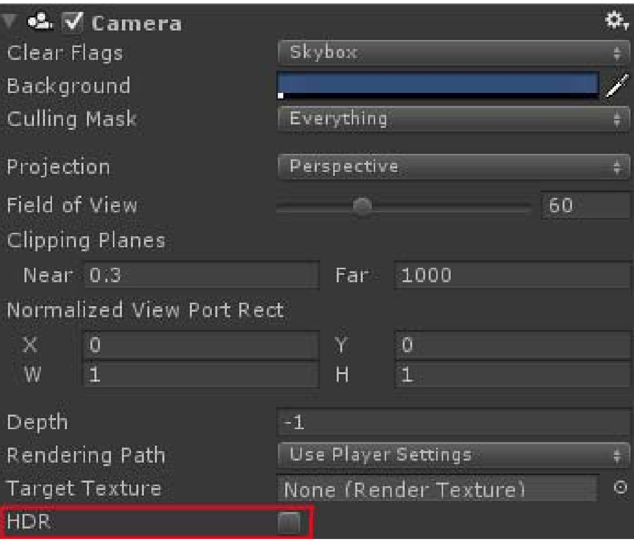
# HDR

In standard rendering, the red, green and blue values for a pixel are each represented by a fraction in the range 0..1, where 0 represents zero intensity and 1 represents the maximum intensity for the display device. While this is straightforward to use, it doesn't accurately reflect the way that lighting works in a real life scene. The human eye tends to adjust to local lighting conditions, so an object that looks white in a dimly lit room may in fact be less bright than an object that looks grey in full daylight. Additionally, the eye is more sensitive to brightness differences at the low end of the range than at the high end.

More convincing visual effects can be achieved if the rendering is adapted to let the ranges of pixel values more accurately reflect the light levels that would be present in a real scene. Although these values will ultimately need to be mapped back to the range available on the display device, any intermediate calculations (such as Unity's image effects) will give more authentic results. Allowing the internal representation of the graphics to use values outside the 0..1 range is the essence of **High Dynamic Range (HDR)** rendering.

### Working with HDR

HDR is enabled separately for each camera using a setting on the Camera component:-




When HDR is active, the scene is rendered into an HDR image buffer which can accommodate pixel values outside the 0..1 range. This buffer is then postprocessed using image effects such as HDR bloom. The tonemapping image effect is what converts the HDR image into the standard low dynamic range (LDR) image to be sent for display. The conversion to LDR must be applied at some point in the image effect pipeline but it need not be the final step if LDR-only image effects are to be applied afterwards. For convenience, some image effects can automatically convert to LDR after applying an HDR effect (see Scripting below).

**Tonemapping**

Tonemapping is the process of mapping HDR values back into the LDR range. There are many different techniques, and what is good for one project may not be the best for another. A variety of tonemapping image effects have been included in Unity. To use them select **Assets -> Import Package -> Image Effects (Pro Only)** select the camera in the scene then select **Component -> Image Effects ->ToneMapping** a detailed description of the tonemapping types can be found in the image effects documentation.

*An exceptionally bright scene rendered in HDR. Without tonemapping, most pixels seem out of range.*

*The same scene as above. But this time, the tonemapping effect is bringing most intensities into a more plausible range. Note that adaptive tonemapping can even blend between above and this image thus simulating the adaptive nature of capturing media (e.g. eyes, cameras).*

**HDR Bloom and Glow**

Using HDR allows for much more control in post processing. LDR bloom has an unfortunate side effect of blurring many areas of a scene even if their pixel intensity is less than 1.0. By using HDR it is possible to only bloom areas where the intensity is greater than one. This leads to a much more desiarable outcome with only super bright elements of a scene bleeding into neighboring pixels. The built in 'Bloom and Lens Flares' image effect now also supports HDR. To attach it to a camera select **Assets -> Import Package -> Image Effects (Pro Only)** select the camera in the scene then select **Component -> Image Effects ->Bloom** a detailed description of the 'Bloom' effect can be found in the image effects documentation.

*The car window sun reflections in this scene have intensity values far bigger than 1.0. Bloom can only pick up and glow these parts if the camera is HDR enabled thus capturing these intensities.*

*The car window will remain without glow if the camera is not HDR enabled. Only way to add glow is to lower the intensity threshhold but then unwanted parts of the image will start glowing as well.*

### Advantages of HDR

- Colors not being lost in high intensity areas
- Better bloom and glow support
- Reduction of banding in low frequency lighting areas

### Disadvantages of HDR

- Uses Floating Point buffers (rendering is slower and requires more VRAM)
- No hardware anti-aliasing support (but you can use Anti-Aliasing image effect to smooth out the edges)
- Not supported on all hardware

### Usage notes

#### Forward Rendering

In forward rendering mode HDR is only supported if you have an image effect present. This is due to performance considerations. If you have no image effect present then no tone mapping will exist and intensity truncation will occur. In this situation the scene will be rendered directly to the backbuffer where HDR is not supported.

#### Deferred Rendering

In HDR mode the light prepass buffer is also allocated as a floating point buffer. This reduces banding in the lighting buffer. HDR is supported in deferred rendering even if no image effects are present.

#### Scripting

The **ImageEffectTransformsToLDR** attribute can be added to an image effect script to indicate that the target buffer should be in LDR instead of HDR. Essentially, this means that a script can automatically convert to LDR after applying its HDR image effect.

Page last updated: 2012-09-05

---

# RenderingPaths

Unity supports different **Rendering Paths**. You should choose which one you use depending on your game content and target platform / hardware. Different rendering paths have different features and performance characteristics that mostly affect Lights and Shadows.

The rendering Path used by your project is chosen in Player Settings. Additionally, you can override it for each Camera.

If the graphics card can't handle a selected rendering path, Unity will automatically use a lower fidelity one. So on a GPU that can't handle Deferred Lighting, Forward Rendering will be used. If Forward Rendering is not supported, Vertex Lit will be used.

### Deferred Lighting

**Deferred Lighting** is the rendering path with the most lighting and shadow fidelity. It is best used if you have many realtime lights. It requires a certain level of hardware support, is for **Unity Pro** only and is not supported on **Mobile Devices**.

For more details see the **Deferred Lighting page**.

### Forward Rendering

**Forward** is a shader-based rendering path. It supports per-pixel lighting (including normal maps & light Cookies) and realtime shadows from one directional light. In the default settings, a small number of the brightest lights are rendered in per-pixel lighting mode. The rest of the lights are calculated at object vertices.

For more details see the **Forward Rendering page**.

### Vertex Lit

**Vertex Lit** is the rendering path with the lowest lighting fidelity and no support for realtime shadows. It is best used on old machines or limited mobile platforms.

For more details see the **Vertex Lit page**.

### Rendering Paths Comparison

| | Deferred Lighting | Forward Rendering | Vertex Lit |
|---|---|---|---|
| ***Features*** | | | |
| Per-pixel lighting (normal maps, light cookies) | Yes | Yes | - |
| Realtime shadows | Yes | 1 Directional Light | - |
| Dual Lightmaps | Yes | - | - |
| Depth&Normals Buffers | Yes | Additional render passes | - |
| Soft Particles | Yes | - | - |
| Semitransparent objects | - | Yes | Yes |
| Anti-Aliasing | - | Yes | Yes |
| Light Culling Masks | Limited | Yes | Yes |
| Lighting Fidelity | All per-pixel | Some per-pixel | All per-vertex |
| ***Performance*** | | | |
| Cost of a per-pixel Light | Number of pixels it illuminates | Number of pixels * Number of objects it illuminates | - |
| ***Platform Support*** | | | |
| PC (Windows/Mac) | Shader Model 3.0+ | Shader Model 2.0+ | Anything |
| Mobile (iOS/Android) | - | OpenGL ES 2.0 | OpenGL ES 2.0 & 1.1 |
| Consoles | 360, PS3 | 360, PS3 | - |

Page last updated: 2010-09-07

---

## Linear Lighting

### overview

Linear lighting refers to the process of illuminating a scene with all inputs being linear. Normally textures exist with gamma pre-applied to them this means that when the textures are sampled in a material that they are non linear. If these textures are used in the standard lighting equations it will lead to the result from the equation being incorrect as they expect all input to be linearized before use.

Linear lighting refers to the process of ensuring that both inputs and outputs of a shader are in the correct color space, this results in more correct lighting.

### Existing (Gamma) Pipeline

In the existing rendering pipeline all colors and textures are sampled in gamma space, that is gamma correction is not removed from images or colors before they are used in a shader. Due to this a situation arises where the inputs to the shader are in gamma space, the lighting equation uses these inputs as if they were in linear space and finally no gamma correction is

applied to the final pixel. Much of the time this looks acceptable as the two wrongs go some way to cancelling each other out. But it is not correct.

### Linear Lighting Pipeline

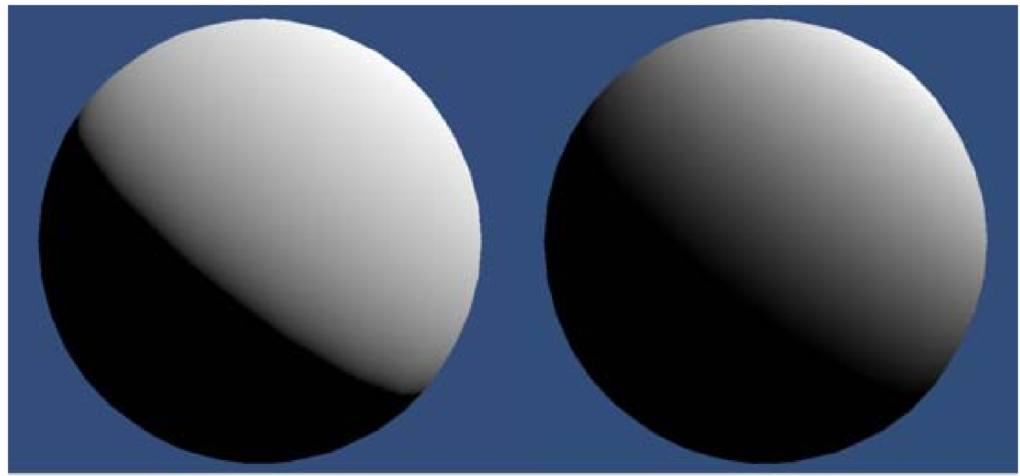
If linear lighting is enabled inputs to the shader program are supplied with the gamma correction removed from them. For colors this conversion is applied implicitly if you are in linear space. Textures are sampled using hardware sRGB reads, the source texture is supplied in gamma space and then on sampling in the graphics hardware the result is converted automatically. These inputs are then supplied to the shader and lighting occurs as it normally would. The resultant value is then written to the framebuffer. This value will either be gamma corrected and written to the framebuffer, of left in linear space for later gamma correction; this depends on the current rendering configuration.

### Differences Between Linear and Gamma Lighting

When using linear lighting input values to the lighting equations are different than in gamma space. This means that as lights striking surfaces will have a different response curve to what the existing Unity rendering pipeline has.

#### Light Falloff

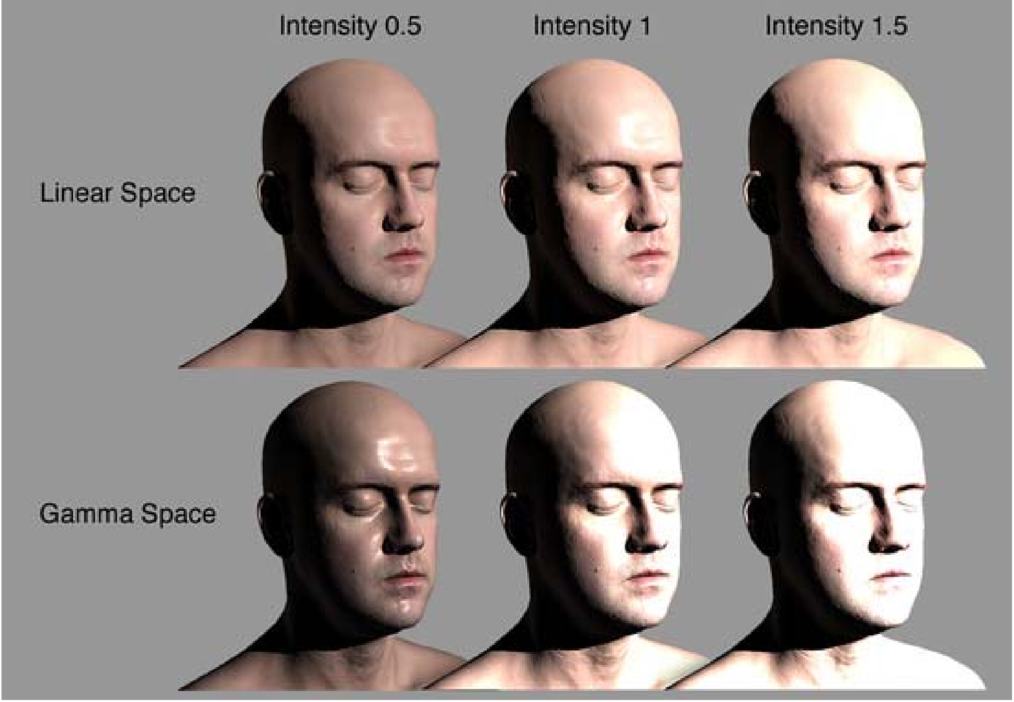
The falloff from distance and normal based lighting is changed in two ways. Firstly when rendering in linear mode the additional gamma correct that is performed will make light radius' appear larger. Secondly lighting edges will also be harsher. This more correctly models lighting intensity falloff on surfaces.



Linear Space                Gamma Space

#### Linear Intensity Response

When you are using gamma space lighting the colors and textures that are supplied to a shader have a gamma correction applied to them. When they are used in a shader the colors of high luminance are actually brighter then they should be for linear lighting. This means that as light intensity increases the surface will get brighter in a non linear way. This leads to lighting that can be too bright in many places, and can also give models and scenes a washed out feel. When you are using linear lighting, as light intensity increases the response from the surface remains linear. This leads to much more realistic surface shading and a much nicer color response in the surface.




Infinite, 3D Head Scan by Lee Perry-Smith is licensed under a Creative Commons Attribution 3.0 Unported License. Available from: http://www.ir-ltd.net/infinite-3d-head-scan-released

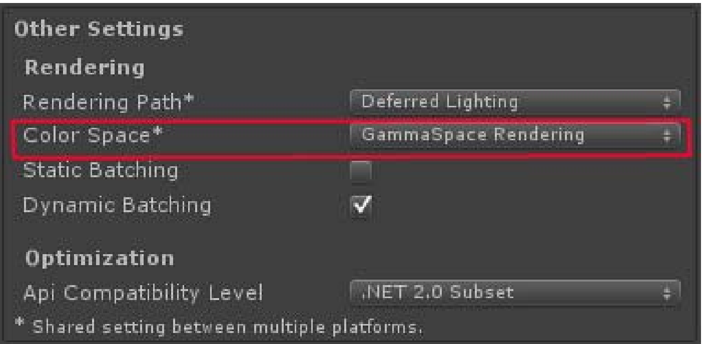
**Linear and Gamma Blending**

When performing blending into the framebuffer the blending occurs in the color space or the framebuffer. When using gamma rendering this means that non linear colors get blended together. This is incorrect. When using linear space rendering blending occurs in linear space, this is correct and leads to expected results.

## Using Linear Lighting

Linear lighting results in a different look to the rendered scene. If you author a project for linear lighting it will most likely not look correct if you change to gamma lighting. Because of this if you move to linear lighting from gamma lighting it may take some time to update the project so that it looks as good as before the switch. That being said enabling linear lighting in Unity is

quite simple. The feature is implemented on a per project level and is exposed in the Player Settings which can be located at **Edit -> Project Settings -> Player -> Other Settings**




**Lightmapping**
When you are using linear lighting all lighting and textures are linearized, this means that the values that are passed to the lightmapper also need to be modified. When you switch between linear lighting and gamma lighting or back you will need to rebake lightmaps. The Unity lightmapping interface will warn you when the lightmaps are in the incorrect color space.

**Supported Platforms**
Linear rendering is not supported on all platforms. The build targets that currently support the feature are:

- Windows & Mac (editor, standalone, web player)
- Xbox 360
- PlayStation 3

Even though these targets support linear lighting, it is not guaranteed that the graphics hardware on the device will be able to render the scene properly. You can check the desired color space and the active supported color space by looking at **QualitySettings.desiredColorSpace** and **QualitySettings.activeColorSpace** if the desired color space is linear but the active color space is gamma then the player has fallen back to gamma space. This can be used to show a warning box telling the user that the application will not look correct for them or to force an exit from the player.

**Linear and Non HDR**
When not using HDR a special framebuffer type is used that supports sRGB read and sRGB write (Degamma on read, Gamma on write). This means that just like a texture the values in the framebuffer are gamma corrected. When this framebuffer is used for blending or bound as texture the values have the gamma removed before being used. When these buffers are written to the value that is being written is converted from linear space to gamma space. If you are rendering in linear mode, all post process effects will have their source and target buffers created with sRGB read and write enabled so that post process and post process blending occurs in linear space.

**Linear and HDR**
When using HDR, rendering is performed into floating point buffers. These buffers have enough resolution to not require conversion to an from gamma space whenever the buffer is accessed, this means that when rendering in linear mode the render targets you use will store the colors in linear space. This means that all blending and post process effects will implicitly be performed in linear space. When the the backbuffer is written to, gamma correction is applied.

**GUI and Linear Authored Textures**
When rendering Unity GUI we do not perform the rendering in linear space. This means that GUI textures should not have their gamma removed on read. This can be achieved in two ways.

- Set the texture type to GUI in the texture importer
- Check the 'Bypass sRGB Sampling' checkbox int the advanced texture importer

It is also important that lookup textures and other textures which are authored to have their RGB values to mean something specific should bypass sRGB sampling.

This will force the sampled texture to not have gamma removed before being used by the graphics hardware. This is also useful for other texture types such as masks where you wish the value that is passed to the shader to be the exact same value that is in the authored texture.

Page last updated: 2012-01-18

# Level Of Detail

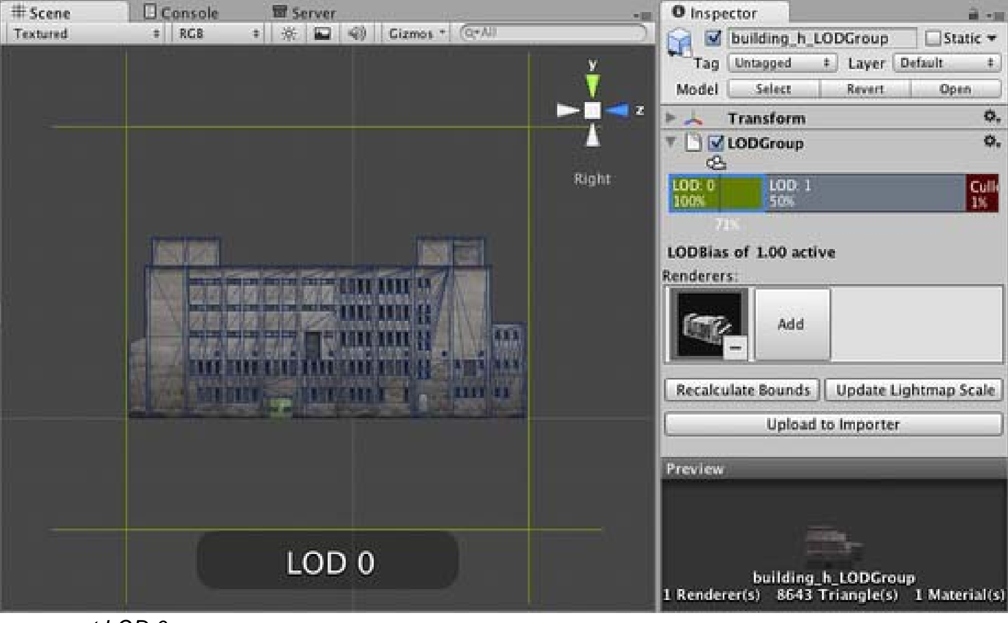
As your scenes get larger, performance becomes a bigger consideration. One of the ways to manage this is to have meshes with different levels of detail depending on how far the camera is from the object. This is called **Level of Detail** (abbreviated as **LOD**).

Here's one of the ways to set up an object with different **LODs**.

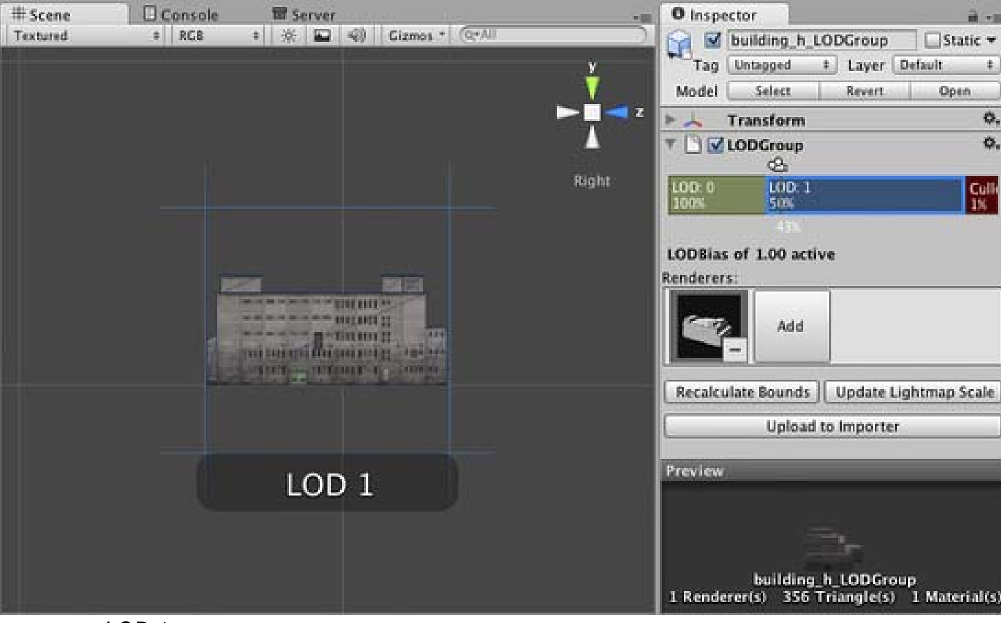
1. Create an empty **Game Object** in the scene
2. Create 2 versions of the mesh, a high-res mesh (for **L0D:0**, when camera is the closest), and a low-res mesh (for **L0D:1**, when camera is further away)
3. Add a **LODGroup** component to this object (**Component**->**Rendering**->**LOD Group**)
4. Drag in the object with the high-res mesh onto the first **Renderers** box for **L0D:0**. Say yes to the "Reparent game objects?" dialog
5. Drag in the object with the low-res mesh onto the first **Renderers** box for **LOD:1**. Say yes to the "Reparent game objects?" dialog
6. Right Click on **LOD:2** and remove it.

At this point the empty object should contain both versions of the mesh and "know" which mesh to show depending on how far away the camera is.

You can preview the effect of this by dragging the camera icon left and right in the window for the **LODGroup** component.




*camera at LOD 0*




*camera at LOD 1*

In the **Scene View**, you should be able to see

- Percentage of the view this object occupies
- What **LOD** is currently being displayed
- The number of triangles

### LOD-based naming conventions in the asset import pipeline

In order to simplify setup of LODs, Unity has a naming convention for models that are being imported.

Simply create your meshes in your modelling tool with names ending with _LOD0, _LOD1, _LOD2, etc., and the LOD group with appropriate settings will be created for you.

Note that the convention assumes that the LOD 0 is the highest resolution model.

### Setting up LODs for different platforms

You can tweak your LOD settings for each platform in Quality Settings, in particular the properties of **LOD bias** and **Maximum LOD Level**.

### Utilities

Here are some options that help you work with LODs

| | |
|---|---|
| Recalculate Bounds | If there is new geometry added to the LODGroup that is not reflected in the bounding volume then click this to update the bounds. One example where this is needed is when one of the geometries is part of a prefab, and new geometry is added to that prefab. Geometry added directly to the LODGroup will automatically update the bounds. |
| Update Lightmaps | Updates the **Scale in Lightmap** property in the lightmaps based on the LOD level boundaries. |
| Upload to Importer | Uploads the LOD level boundaries to the importer. |

Page last updated: 2012-02-01

---

# Shaders

All rendering in Unity is done with *Shaders* - small scripts that let you configure the how the graphics hardware is set up for rendering. Unity ships with 60+ built-in shaders (documented in the Built-in Shader Guide). You can extend this by making your own shaders.

Shaders in Unity can be written in one of three different ways:

**Surface Shaders**
Surface Shaders are your best option *if your shader needs to be affected by lights and shadows*. Surface shaders make it easy to write complex shaders in a compact way - it's a higher level of abstraction for interaction with Unity's lighting pipeline. Most surface shaders automatically support both forward and deferred lighting. You write surface shaders in a couple of lines of **Cg/HLSL** and a lot more code gets auto-generated from that.

Do *not* use surface shaders if your shader is not doing anything with lights. For Image Effects or many special-effect shaders, surface shaders are a suboptimal option, since they will do a bunch of lighting calculations for no good reason!

**Vertex and Fragment Shaders**
Vertex and Fragment Shaders will be required, if your shader doesn't need to interact with lighting, or if you need some very exotic effects that the surface shaders can't handle. Shader programs written this way are the most flexible way to create the effect you need (even surface shaders are automatically converted to a bunch of vertex and fragment shaders), but that comes at a price: you have to write more code and it's harder to make it interact with lighting. These shaders are written in **Cg/HLSL** as well.

**Fixed Function Shaders**
Fixed Function Shaders need to be written for old hardware that doesn't support programmable shaders. You will probably want to write fixed function shaders as an n-th fallback to your fancy fragment or surface shaders, to make sure your game still renders something sensible when run on old hardware or simpler mobile platforms. Fixed function shaders are entirely written in a language called **ShaderLab**, which is similar to Microsoft's .FX files or NVIDIA's CgFX.

## ShaderLab

Regardless of which type you choose, the actual meat of the shader code will always be wrapped in ShaderLab, which is used to organize the shader structure. It looks like this:

```
Shader "MyShader" {
    Properties {
        _MyTexture ("My Texture", 2D) = "white" { }
        // other properties like colors or vectors go here as well
    }
    SubShader {
        // here goes the 'meat' of your
        //  - surface shader or
        //  - vertex and fragment shader or
        //  - fixed function shader
    }
    SubShader {
        // here goes a simpler version of the SubShader above that can run on older graphics cards
    }
}
```

We recommend that you start by reading about some basic concepts of the ShaderLab syntax in the ShaderLab reference and then move on to the tutorials listed below.

The tutorials include plenty of examples for the different types of shaders. For even more examples of surface shaders in particular, you can get the source of Unity's built-in shaders from the Resources section. Unity's Image Effects package contains a lot of interesting vertex and fragment shaders.

Read on for an introduction to shaders, and check out the shader reference!

- Tutorial: ShaderLab & Fixed Function Shaders
- Tutorial: Vertex and Fragment Shaders
- Surface Shaders

Page last updated: 2012-08-13

---

# ShaderTut1

This tutorial will teach you how you can create your own shaders and make your game look a lot better!

Unity is equipped with a powerful shading and material language called **ShaderLab**. In style it is similar to CgFX and Direct3D Effects (.FX) languages - it describes everything needed to display a **Material**.

Shaders describe properties that are exposed in Unity's Material Inspector and multiple shader implementations (**SubShaders**) targeted at different graphics hardware capabilities, each describing complete graphics hardware rendering state, fixed function pipeline setup or vertex/fragment programs to use. Vertex and fragment programs are written in the high-level **Cg/HLSL** programming language.

In this tutorial we describe how to write shaders in using both fixed function and programmable pipelines. We assume that the reader has a basic understanding of OpenGL or Direct3D render states, fixed function and programmable pipelines and has some knowledge of Cg, HLSL or GLSL programming languages. Some shader tutorials and documentation can be found on NVIDIA and AMD developer sites.

## Getting started

To create a new shader, either choose **Assets->Create->Shader** from the menubar, or duplicate an existing shader, and work from that. The new shader can be edited by double-clicking it in the **Project View**.

We'll start with a very basic shader:

```
Shader "Tutorial/Basic" {
    Properties {
        _Color ("Main Color", Color) = (1,0.5,0.5,1)
    }
    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
            }
            Lighting On
        }
    }
}
```

This simple shader demonstrates one of the most basic shaders possible. It defines a color property called **Main Color** and assigns it a default value of rose-like color (red=100% green=50% blue=50% alpha=100%). It then renders the object by invoking a **Pass** and in that pass setting the diffuse material component to the property **_Color** and turning on per-vertex lighting.

To test this shader, create a new material, select the shader from the drop-down menu (**Tutorial->Basic**) and assign the Material to some object. Tweak the color in the Material Inspector and watch the changes. Time to move onto more complex things!

## Basic Vertex Lighting

If you open an existing complex shader, it can be a bit hard to get a good overview. To get you started, we will dissect the built-in **VertexLit** shader that ships with Unity. This shader uses fixed function pipeline to do standard per-vertex lighting.

```
Shader "VertexLit" {
    Properties {
        _Color ("Main Color", Color) = (1,1,1,0.5)
        _SpecColor ("Spec Color", Color) = (1,1,1,1)
        _Emission ("Emmisive Color", Color) = (0,0,0,0)
        _Shininess ("Shininess", Range (0.01, 1)) = 0.7
        _MainTex ("Base (RGB)", 2D) = "white" { }
    }
```

```
    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
                Ambient [_Color]
                Shininess [_Shininess]
                Specular [_SpecColor]
                Emission [_Emission]
            }
            Lighting On
            SeparateSpecular On
            SetTexture [_MainTex] {
                constantColor [_Color]
                Combine texture * primary DOUBLE, texture * constant
            }
        }
    }
}
```

All shaders start with the keyword **Shader** followed by a string that represents the name of the shader. This is the name that is shown in the **Inspector**. All code for this shader must be put within the curly braces after it: **{ }** (called a block).

- The name should be short and descriptive. It does not have to match the **.shader** file name.
- To put shaders in submenus in Unity, use slashes - e.g. **MyShaders/Test** would be shown as **Test** in a submenu called **MyShaders**, or **MyShaders->Test**.

The shader is composed of a **Properties** block followed by **SubShader** blocks. Each of these is described in sections below.

### Properties

At the beginning of the shader block you can define any properties that artists can edit in the Material Inspector. In the *VertexLit* example the properties look like this:

```
Shader " VertexLit" {
    Properties {
        _Color ("Main Color", Co
        _SpecColor ("Spec Color
        _Emission ("Emmisive C
        _Shininess ("Shininess",
        _MainTex ("Base (RGB)",
    }
```

The properties are listed on separate lines within the **Properties** block. Each property starts with the internal name (**Color**, **MainTex**). After this in parentheses comes the name shown in the inspector and the type of the property. After that, the default value for this property is listed:

_Color ("Main Color", Color) = (1,1,1,0)

Internal name | Inspector title | Property type | Default value

The list of possible types are in the Properties Reference. The default value depends on the property type. In the example of a color, the default value should be a four component vector.

We now have our properties defined, and are ready to start writing the actual shader.

### The Shader Body

Before we move on, let's define the basic structure of a shader file.

Different graphic hardware has different capabilities. For example, some graphics cards support fragment programs and others don't; some can lay down four textures per pass while the others can do only two or one; etc. To allow you to make full use of whatever hardware your user has, a shader can contain multiple **SubShaders**. When Unity renders a shader, it will go over all subshaders and use the first one that the hardware supports.

```
Shader "Structure Example" {
    Properties { /* ...shader properties... }
    SubShader {
        // ...subshader that uses vertex/fragment programs...
    }
    SubShader {
        // ...subshader that uses four textures per pass...
```

```
    }
    SubShader {
        // ...subshader that uses two textures per pass...
    }
    SubShader {
        // ...subshader that might look ugly but runs on anything :)
    }
}
```

This system allows Unity to support all existing hardware and maximize the quality on each one. It does, however, result in some long shaders.

Inside each SubShader block you set the rendering state shared by all passes; and define rendering passes themselves. A complete list of available commands can be found in the SubShader Reference.

### Passes

Each subshader is a collection of passes. For each pass, the object geometry is rendered, so there must be at least one pass. Our VertexLit shader has just one pass:

```
// ...snip...
Pass {
    Material {
        Diffuse [_Color]
        Ambient [_Color]
        Shininess [_Shininess]
        Specular [_SpecColor]
        Emission [_Emission]
    }
    Lighting On
    SeparateSpecular On
    SetTexture [_MainTex] {
        constantColor [_Color]
        Combine texture * primary DOUBLE, texture * constant
    }
}
// ...snip...
```

Any commands defined in a pass configures the graphics hardware to render the geometry in a specific way.

In the example above we have a **Material** block that binds our property values to the fixed function lighting material settings. The command **Lighting On** turns on the standard vertex lighting, and **SeparateSpecular On** enables the use of a separate color for the specular highlight.

All of these command so far map very directly to the fixed function OpenGL/Direct3D hardware model. Consult OpenGL red book for more information on this.

The next command, **SetTexture**, is very important. These commands define the textures we want to use and how to mix, combine and apply them in our rendering. **SetTexture** command is followed by the property name of the texture we would like to use (**_MainTex** here) This is followed by a **combiner block** that defines how the texture is applied. The commands in the combiner block are executed for each pixel that is rendered on screen.

Within this block we set a constant color value, namely the Color of the Material, **_Color**. We'll use this constant color below.

In the next command we specify how to mix the texture with the color values. We do this with the **Combine** command that specifies how to blend the texture with another one or with a color. Generally it looks like this:

```
Combine ColorPart, AlphaPart
```

Here **ColorPart** and **AlphaPart** define blending of color (RGB) and alpha (A) components respectively. If **AlphaPart** is omitted, then it uses the same blending as **ColorPart**.

In our VertexLit example:

```
Combine texture * primary DOUBLE, texture * constant
```

Here **texture** is the color coming from the current texture (here **_MainTex**). It is multiplied (*) with the **primary** vertex color. Primary color is the vertex lighting color, calculated from the Material values above. Finally, the result is multiplied by two to increase lighting intensity (**DOUBLE**). The alpha value (after the comma) is **texture** multiplied by **constant** value (set with **constantColor** above). Another often used combiner mode is called **previous** (not used in this shader). This is the result of any previous **SetTexture** step, and can be used to combine several textures and/or colors with each other.

### Summary

Our VertexLit shader configures standard vertex lighting and sets up the texture combiners so that the rendered lighting intensity is doubled.

We could put more passes into the shader, they would get rendered one after the other. For now, though, that is not nessesary as we have the desired effect. We only need one SubShader as we make no use of any advanced features - this particular shader will work on any graphics card that Unity supports.

The VertexLit shader is one of the most basic shaders that we can think of. We did not use any hardware specific operations, nor did we utilize any of the more special and cool commands that ShaderLab and Cg has to offer.

In the next chapter we'll proceed by explaining how to write custom vertex & fragment programs using Cg language.

Page last updated: 2010-09-25

---

## ShaderTut2

This tutorial will teach you how to write custom vertex and fragment programs in Unity shaders. For a basic introduction to **ShaderLab** see the Getting Started tutorial. If you want to write shaders that interact with lighting, read about **Surface Shaders** instead.

Lets start with a small recap of the general structure of a shader:

```
Shader "MyShaderName" {
    Properties {
        // ... properties here ...
    }
    SubShader {
        // ... subshader for graphics hardware A ...
        Pass {
            // ... pass commands ...
        }
        // ... more passes if needed ...
    }
    SubShader {
        // ... subshader for graphics hardware B ...
    }
    // ... Optional fallback ...
    FallBack "VertexLit"
}
```

Here at the end we introduce a new command:

```
FallBack "VertexLit"
```

The **Fallback** command can be used at the end of the shader; it tells which shader should be used if no **SubShaders** from the current shader can run on user's graphics hardware. The effect is the same as including all SubShaders from the fallback

shader at the end. For example, if you were to write a normal-mapped shader, then instead of writing a very basic non-normal-mapped subshader for old graphics cards you can just fallback to built-in **VertexLit** shader.

The basic building blocks of the shader are introduced in the first shader tutorial while the full documentation of Properties, SubShaders and Passes are also available.

A quick way of building SubShaders is to use passes defined in other shaders. The command **UsePass** does just that, so you can reuse shader code in a neat fashion. As an example the following command uses the pass with the name "BASE" from the built-in **Specular** shader:

```
UsePass "Specular/BASE"
```

In order for **UsePass** to work, a name must be given to the pass one wishes to use. The **Name** command inside the pass gives it a name:

```
Name "MyPassName"
```

### Vertex and fragment programs

We described a pass that used just a single texture combine instruction in the first tutorial. Now it is time to demonstrate how we can use vertex and fragment programs in our pass.

When you use vertex and fragment programs (the so called "programmable pipeline"), most of the hardcoded functionality ("fixed function pipeline") in the graphics hardware is switched off. For example, using a vertex program turns off standard 3D transformations, lighting and texture coordinate generation completely. Similarly, using a fragment program replaces any texture combine modes that would be defined in SetTexture commands; thus SetTexture commands are not needed.

Writing vertex/fragment programs requires a thorough knowledge of 3D transformations, lighting and coordinate spaces - because you have to rewrite the fixed functionality that is built into API's like OpenGL yourself. On the other hand, you can do much more than what's built in!

### Using Cg in ShaderLab

Shaders in ShaderLab are usually written in Cg programming language by embedding "Cg snippets" in the shader text. Cg snippets are compiled into low-level shader assembly by the Unity editor, and the final shader that is included in your game's data files only contains this low-level assembly. When you select a shader in the **Project View**, the **Inspector** shows shader text after Cg compilation, which might help as a debugging aid. Unity automatically compiles Cg snippets for both Direct3D, OpenGL, Flash and so on, so your shaders will just work on all platforms. Note that because Cg code is compiled by the editor, you can't create Cg shaders from scripts at runtime.

In general, Cg snippets are placed inside Pass blocks. They look like this:

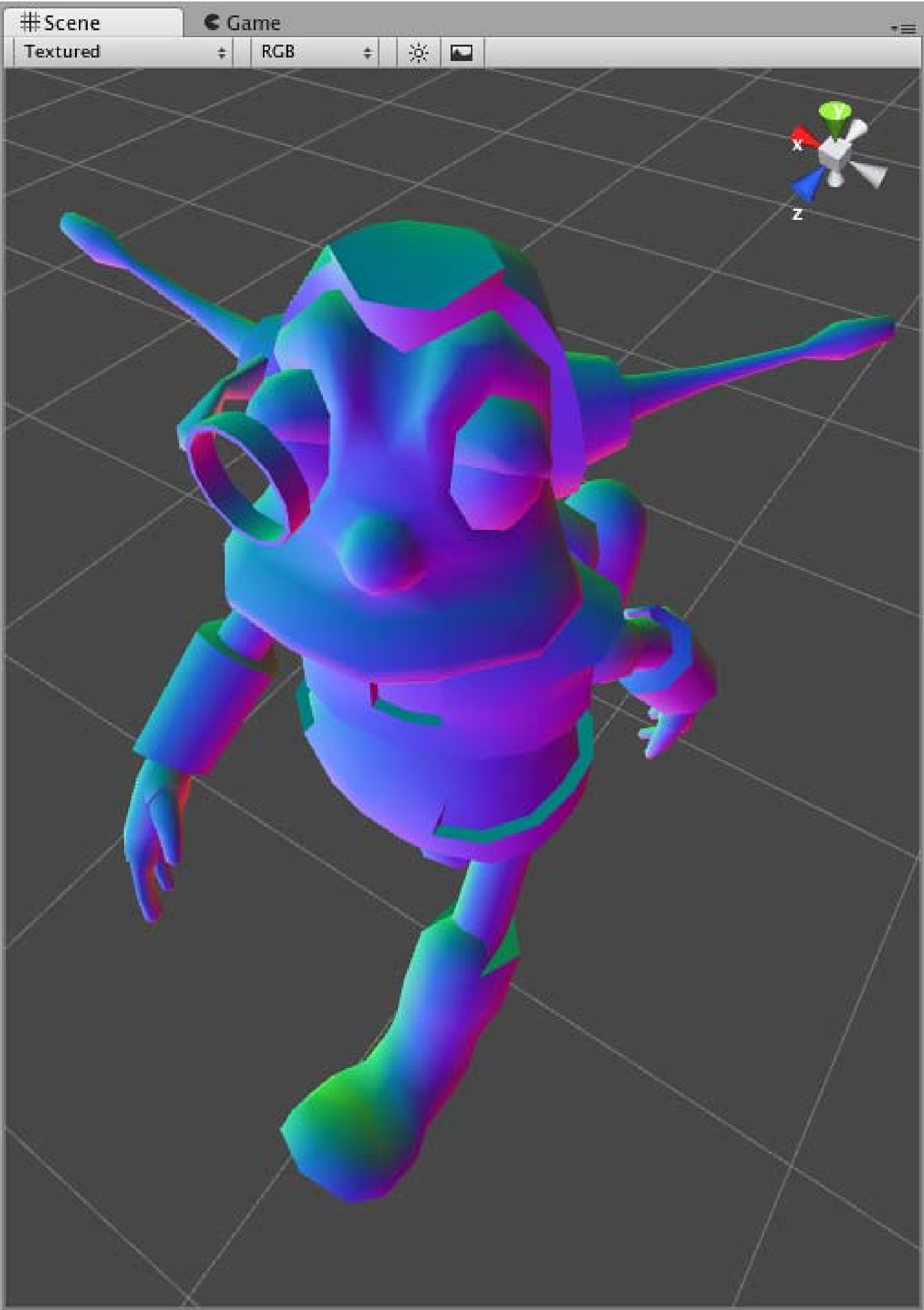
```
Pass {
    // ... the usual pass state setup ...

    CGPROGRAM
    // compilation directives for this snippet, e.g.:
    #pragma vertex vert
    #pragma fragment frag

    // the Cg code itself

    ENDCG
    // ... the rest of pass setup ...
}
```

The following example demonstrates a complete shader with Cg programs that renders object normals as colors:

```
Shader "Tutorial/Display Normals" {
SubShader {
  Pass {
```

```
CGPROGRAM
#pragma vertex vert
#pragma fragment frag
#include "UnityCG.cginc"

struct v2f {
    float4 pos : SV_POSITION;
    float3 color : COLOR0;
};

v2f vert (appdata_base v)
{
    v2f o;
    o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
    o.color = v.normal * 0.5 + 0.5;
    return o;
}

half4 frag (v2f i) : COLOR
{
    return half4 (i.color, 1);
}
ENDCG

    }
}
Fallback "VertexLit"
}
```

When applied on an object it will result in an image like this (if your graphics card supports vertex & fragment programs of course):




Our "Display Normals" shader does not have any properties, contains a single SubShader with a single Pass that is empty except for the Cg code. Finally, a fallback to the built-in **VertexLit** shader is defined. Let's dissect the Cg code part by part:

```
CGPROGRAM
#pragma vertex vert
#pragma fragment frag
// ... snip ...
ENDCG
```

The whole Cg snippet is written between **CGPROGRAM** and **ENDCG** keywords. At the start compilation directives are given as **#pragma** statements:

- **#pragma vertex name** tells that the code contains a vertex program in the given function (**vert** here).
- **#pragma fragment name** tells that the code contains a fragment program in the given function (**frag** here).

Following the compilation directives is just plain Cg code. We start by including a builtin Cg file:

```
#include UnityCg.cginc
```

The **UnityCg.cginc** file contains commonly used declarations and functions so that the shaders can be kept smaller (see shader include files page for details). Here we'll use **appdata_base** structure from that file. We could just define them directly in the shader and not include the file of course.

Next we define a "vertex to fragment" structure (here named **v2f**) - what information is passed from the vertex to the fragment program. We pass the position and color parameters. The color will be computed in the vertex program and just output in the fragment program.

We proceed by defining the vertex program - **vert** function. Here we compute the position and output input normal as a color:

```
o.color = v.normal * 0.5 + 0.5;
```

Normal components are in -1..1 range, while colors are in 0..1 range, so we scale and bias the normal in the code above. Next we define a fragment program - **frag** function that just outputs the calculated color and 1 as the alpha component:

```
half4 frag (v2f i) : COLOR
{
    return half4 (i.color, 1);
}
```

That's it, our shader is finished! Even this simple shader is very useful to visualize mesh normals.

Of course, this shader does not respond to lights at all, and that's where things get a bit more interesting; read about Surface Shaders for details.

### Using shader properties in Cg code

When you define properties in the shader, you give them a name like **_Color** or **_MainTex**. To use them in Cg you just have to define a variable of a matching name and type. Unity will automatically set Cg variables that have names matching with shader properties.

Here is a complete shader that displays a texture modulated by a color. Of course, you could easily do the same in a texture combiner call, but the point here is just to show how to use properties in Cg:

```
Shader "Tutorial/Textured Colored" {
Properties {
   _Color ("Main Color", Color) = (1,1,1,0.5)
   _MainTex ("Texture", 2D) = "white" { }
}
SubShader {
   Pass {

CGPROGRAM
#pragma vertex vert
#pragma fragment frag

#include "UnityCG.cginc"

float4 _Color;
sampler2D _MainTex;

struct v2f {
   float4  pos : SV_POSITION;
   float2  uv : TEXCOORD0;
};

float4 _MainTex_ST;

v2f vert (appdata_base v)
{
   v2f o;
```

```
    o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
    o.uv = TRANSFORM_TEX (v.texcoord, _MainTex);
    return o;
}

half4 frag (v2f i) : COLOR
{
    half4 texcol = tex2D (_MainTex, i.uv);
    return texcol * _Color;
}
ENDCG

    }
}
Fallback "VertexLit"
}
```

The structure of this shader is the same as in the previous example. Here we define two properties, namely **_Color** and **_MainTex**. Inside Cg code we define corresponding variables:

```
float4 _Color;
sampler2D _MainTex;
```

See Accessing Shader Properties in Cg for more information.

The vertex and fragment programs here don't do anything fancy; vertex program uses the **TRANSFORM_TEX** macro from UnityCG.cginc to make sure texture scale&offset is applied correctly, and fragment program just samples the texture and multiplies by the color property.

Note that because we're writing our own fragment program here, we don't need any SetTexture commands. How the textures are applied in the shader is entirely controlled by the fragment program.

### Summary

We have shown how custom shader programs can be generated in a few easy steps. While the examples shown here are very simple, there's nothing preventing you to write arbitrarily complex shader programs! This can help you to take the full advantage of Unity and achieve optimal rendering results.

The complete ShaderLab reference manual is here. We also have a forum for shaders at forum.unity3d.com so go there to get help with your shaders! Happy programming, and enjoy the power of Unity and Shaderlab.

Page last updated: 2012-09-04

---

## DirectX 11

Unity 4 introduces ability to use DirectX 11 graphics API, with all the goodies that you expect from it: compute shaders, tessellation shaders, shader model 5.0 and so on.

### Enabling DirectX 11

To enable DirectX 11 for your game builds and the editor, set "Use DX11" option in Player Settings. Unity editor needs to be restarted for this to take effect.

Note that DX11 requires Windows Vista or later and at least a DX10-level GPU (preferably DX11-level). Unity editor window title has "<DX11>" at the end when it is actually running in DX11 mode.

### Image Effects that can take advantage of DX11

- Depth of Field effect (optimized Bokeh texture splatting)
- Noise and Grain effect (higher quality noise patterns)
- Motion Blur effect (higher quality reconstruction filter)

### Compute Shaders

Compute shaders allow using GPU as a massively parallel processor. See Compute Shaders page for mode details.

### Tessellation & Geometry Shaders

Surface shaders have support for simple tessellation & displacement, see Surface Shader Tessellation page.

When manually writing shader programs, you can use full set of DX11 shader model 5.0 features, including geometry, hull & domain shaders.

### DirectX 11 Examples

The following screenshots show examples of what becomes possible with DirectX 11.

*The volumetric explosion in these shots is rendered using raymarching which becomes plausible with shader model 5.0. Moreover, as it generates and updates depth values, it becomes fully compatible with depth based image effects such as depth of field or motion blur.*

*The hair in this shot is implemented via DirectX 11 tessellation & geometry shaders to dynamically generate and animate individual strings of hair. Shading is based on a model proposed by Kajiya-Kai that enables a more believable diffuse and specular behaviour.*

*Similar to the hair technique above, the shown slippers fur is also based on generating geometry emitted from a simple base slippers mesh.*

*The blur effect in this image (known as **Bokeh**) is based on splatting a texture on top of very bright pixels. This can create very believable camera lens blurs, especially when used in conjunction with HDR rendering.*

*Exaggerated lens blur similar to the screenshot above. This is a possible result using the new Depth of Field effect*

Page last updated: 2012-10-30

---

## Compute Shaders

Compute Shaders are programs that run on the graphics card, outside of the normal rendering pipeline. They can be used for

massively parallel GPGPU algorithms, or to accelerate parts of game rendering. In order to efficiently use them, often an in-depth knowledge of GPU architectures and parallel algorithms is needed; as well as knowledge of DirectCompute, OpenCL or CUDA.

Compute shaders in Unity are built on top of DirectX 11 DirectCompute technology; and currently require Windows Vista or later and a GPU capable of Shader Model 5.0.

### Compute shader assets

Similar to normal shaders, Compute Shaders are asset files in your project, with `*.compute` file extension. They are written in DirectX 11 style HLSL language, with minimal amount of `#pragma` compilation directives to indicate which functions to compile as compute shader kernels.

Here's a minimal example of a compute shader file:

```
// test.compute
#pragma kernel FillWithRed

RWTexture2D<float4> res;

[numthreads(1,1,1)]
void FillWithRed (uint3 dtid : SV_DispatchThreadID)
{
    res[dtid.xy] = float4(1,0,0,1);
}
```

The example above does not do anything remotely interesting, just fills output texture with red color.

The language is standard DX11 HLSL, with the only exception of a `#pragma kernel FillWithRed` directive. One compute shader asset file must contain at least one "compute kernel" that can be invoked, and that function is indicated by the #pragma directive. There can be more kernels in the file; just add multiple `#pragma kernel` lines.

The `#pragma kernel` line can optionally be followed by a number of preprocessor macros to define while compiling that kernel, for example:

```
#pragma kernel KernelOne SOME_DEFINE DEFINE_WITH_VALUE=1337
#pragma kernel KernelTwo OTHER_DEFINE
// ...
```

### Invoking compute shaders

In your script, define a variable of `ComputeShader` type, assign a reference to the asset, and then you can invoke them with ComputeShader.Dispatch function. See scripting reference of ComputeShader class for more details.

Closely related to compute shaders is a ComputeBuffer class, which defines arbitrary data buffer ("structured buffer" in DX11 lingo). Render Textures can also be written into from compute shaders, if they have "random access" flag set ("unordered access view" in DX11), see RenderTexture.enableRandomWrite.

Page last updated: 2012-08-13

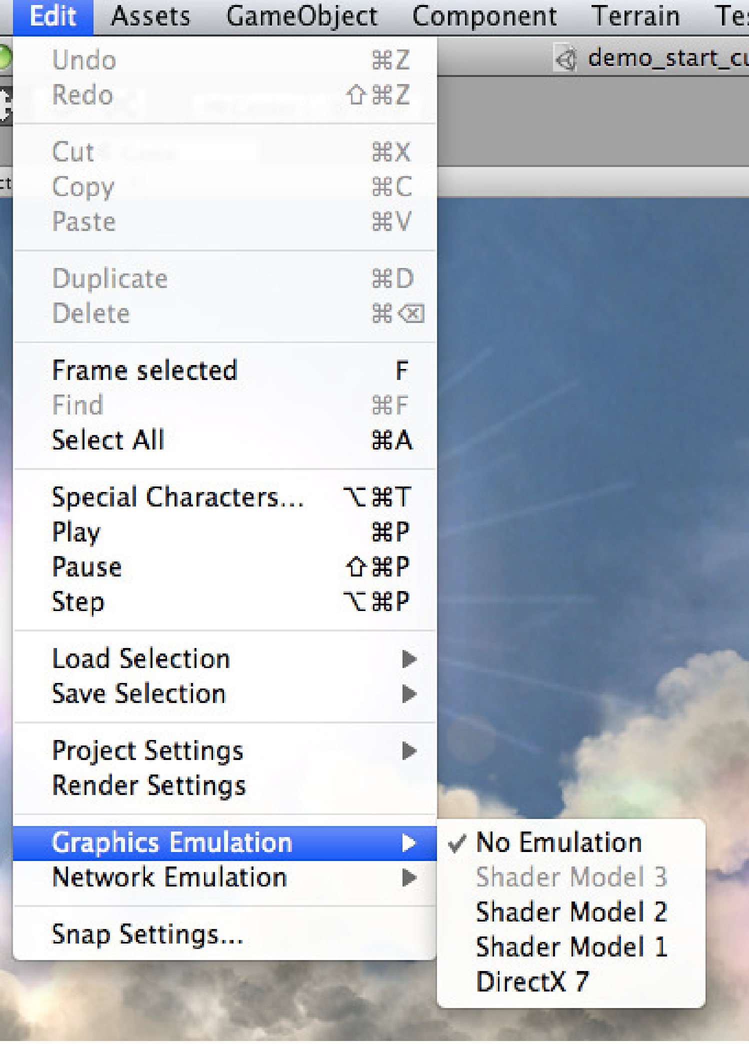
---

## GraphicsEmulation

You can choose to emulate less capable graphics hardware when working in the Unity editor. This is very handy when writing custom shaders and rendering effects, and is a quick way to test how your game will look on that eight year old graphics card that someone might have.

To enable Graphics emulation, go to **Edit->Graphics Emulation**, and choose your desired emulation level.

**Note:** The available graphic emulation options change depending on the platform you are currently targeting. More information can be found on the Publishing builds page.




*Enabling Graphics Emulation*

## Technical Details

Graphics emulation limits the graphics *capabilities* that are supported, but it does not emulate the *performance* of graphics hardware. Your game in the editor will still be rendered by your graphics card, more and more features will be disabled as you reduce emulation quality.

While emulation is a quick way to check out graphics capabilities, you should still test your game on actual hardware. This will reveal real performance and any peculiarities of the specific graphics card, operating system or driver version.

## Emulation Levels

Graphics emulation levels are the following:

**In web player or standalone mode:**

| | |
|---|---|
| **No Emulation** | No emulation is performed. |
| **Shader Model 3** | Emulates graphics card with Shader Model 3.0 level capabilities. Long vertex & fragment shader programs, realtime shadows, HDR. |
| **Shader Model 2** | Shader Model 2.0 capabilities. Vertex & fragment programs, realtime shadows. No HDR, maximum 4 texture combiner stages. |
| **Shader Model 1** | Shader Model 1.x capabilities. Vertex programs, 4 texture combiner stages. **Not supported**: fragment programs, shadows, HDR, depth textures, multiple render targets. |
| **DirectX 7** | DirectX 7 level capabilities. Vertex programs (usually in software mode), two texture combiner stages. **Not supported**: fragment programs, shadows, HDR, depth textures, 3D textures, min/max/sub blending. |

**In iOS or Android mode:**

| | |
|---|---|
| **No Emulation** | No emulation is performed. |
| **OpenGL ES 1.x** | OpenGL ES 1.1: Four texture combiner stages. **Not supported**: vertex or fragment programs, shadows and pretty much all other graphics features ;) |
| **OpenGL ES 2.0** | OpenGL ES 2.0: Vertex & fragment programs, four texture combiner stages. **Not supported**: HDR, 3D textures. |

When your graphics card does not support all the capabilities of some emulation level, that level will be disabled. For example, the Intel GMA950 (Intel 915/945/3000) card does not support Shader Model 3.0, so there's no way to emulate that level.

Page last updated: 2012-08-17

---

# AssetDatabase

AssetDatabase is an API which allows you to access the assets contained in your project. Among other things, it provides methods to find and load assets and also to create, delete and modify them. The Unity Editor uses the AssetDatabase internally to keep track of asset files and maintain the linkage between assets and objects that reference them. Since Unity needs to keep track of all changes to the project folder, you should always use the AssetDatabase API rather than the filesystem if you want to access or modify asset data.

The AssetDatabase interface is only available in the editor and has no function in the built player. Like all other editor classes, it is only available to scripts placed in the Editor folder (just create a folder named �Editor� in the main Assets folder of your project if there isn't one already).

### Importing an Asset

Unity normally imports assets automatically when they are dragged into the project but it is also possible to import them under script control. To do this you can use the AssetDatabase.ImportAsset method as in the example below.

```
using UnityEngine;
using UnityEditor;

public class ImportAsset {
	[MenuItem ("AssetDatabase/ImportExample")]
	static void ImportExample ()
	{
		AssetDatabase.ImportAsset("Assets/Textures/texture.jpg", ImportAssetOptions.Default);
	}
}
```

You can also pass an extra parameter of type AssetDatabase.ImportAssetOptions to the AssetDatabase.ImportAsset call. The scripting reference page documents the different options and their effects on the function's behaviour.

### Loading an Asset

The editor loads assets only as needed, say if they are added to the scene or edited from the Inspector panel. However, you can load and access assets from a script using AssetDatabase.LoadAssetAtPath, AssetDatabase.LoadMainAssetAtPath, AssetDatabase.LoadAllAssetRepresentationsAtPath and AssetDatabase.LoadAllAssetsAtPath. See the scripting documentation for further details.

```
using UnityEngine;
using UnityEditor;

public class ImportAsset {
	[MenuItem ("AssetDatabase/LoadAssetExample")]
	static void ImportExample ()
	{
		Texture2D t = AssetDatabase.LoadAssetAtPath("Assets/Textures/texture.jpg", typeof(Texture2D)) as Texture2D;
```

```
    }
}
```

### File Operations using the AssetDatabase

Since Unity keeps metadata about asset files, you should never create, move or delete them using the filesystem. Instead, you can use AssetDatabase.Contains, AssetDatabase.CreateAsset, AssetDatabase.CreateFolder, AssetDatabase.RenameAsset, AssetDatabase.CopyAsset, AssetDatabase.MoveAsset, AssetDatabase.MoveAssetToTrash and AssetDatabase.DeleteAsset.

```
public class AssetDatabaseIOExample {
    [MenuItem ("AssetDatabase/FileOperationsExample")]
    static void Example ()
    {
        string ret;

        // Create
        Material material = new Material (Shader.Find("Specular"));
        AssetDatabase.CreateAsset(material, "Assets/MyMaterial.mat");
        if(AssetDatabase.Contains(material))
            Debug.Log("Material asset created");

        // Rename
        ret = AssetDatabase.RenameAsset("Assets/MyMaterial.mat", "MyMaterialNew");
        if(ret == "")
            Debug.Log("Material asset renamed to MyMaterialNew");
        else
            Debug.Log(ret);

        // Create a Folder
        ret = AssetDatabase.CreateFolder("Assets", "NewFolder");
        if(AssetDatabase.GUIDToAssetPath(ret) != "")
            Debug.Log("Folder asset created");
        else
            Debug.Log("Couldn't find the GUID for the path");

        // Move
        ret = AssetDatabase.MoveAsset(AssetDatabase.GetAssetPath(material), "Assets/NewFolder/MyMaterialNew.mat")
        if(ret == "")
            Debug.Log("Material asset moved to NewFolder/MyMaterialNew.mat");
        else
            Debug.Log(ret);

        // Copy
        if(AssetDatabase.CopyAsset(AssetDatabase.GetAssetPath(material), "Assets/MyMaterialNew.mat"))
            Debug.Log("Material asset copied as Assets/MyMaterialNew.mat");
        else
            Debug.Log("Couldn't copy the material");
        // Manually refresh the Database to inform of a change
        AssetDatabase.Refresh();
        Material MaterialCopy = AssetDatabase.LoadAssetAtPath("Assets/MyMaterialNew.mat", typeof(Material)) as Mater

        // Move to Trash
        if(AssetDatabase.MoveAssetToTrash(AssetDatabase.GetAssetPath(MaterialCopy)))
            Debug.Log("MaterialCopy asset moved to trash");

        // Delete
        if(AssetDatabase.DeleteAsset(AssetDatabase.GetAssetPath(material)))
            Debug.Log("Material asset deleted");
        if(AssetDatabase.DeleteAsset("Assets/NewFolder"))
            Debug.Log("NewFolder deleted");
```

```
        // Refresh the AssetDatabase after all the changes
        AssetDatabase.Refresh();
    }
}
```

### Using AssetDatabase.Refresh

When you have finished modifying assets, you should call AssetDatabase.Refresh to commit your changes to the database and make them visible in the project.

Page last updated: 2012-06-27

---

# BuildPlayerPipeline

When building a player, you sometimes want to modify the built player in some way. For example you might want to add a custom icon, copy some documentation next to the player or build an Installer. Doing this manually can become tedious and if you know how to write sh or perl scripts you can automate this task.

### Mac OSX

After building a player Unity automatically looks for a sh or perl script called **PostprocessBuildPlayer** (without any extension) in your Project's Assets/Editor folder. If the file is found, it is invoked when the player finishes building.

In this script you can post process the player in any way you like. For example build an installer out of the player.

You can use perl, sh or any other commandline compatible language.

Unity passes some useful command line arguments to the script, so you know what kind of player it is and where it is stored.

The current directory will be set to be the project folder, that is the folder containing the Assets folder.

```
#!/usr/bin/perl

my $installPath = $ARGV[0];

# The type of player built:
# "dashboard", "standaloneWin32", "standaloneOSXIntel", "standaloneOSXPPC", "standaloneOSXUniversal", "webplayer"
my $target = $ARGV[1];

# What optimizations are applied. At the moment either "" or "strip" when Strip debug symbols is selected.
my $optimization = $ARGV[2];

# The name of the company set in the project settings
my $companyName = $ARGV[3];

# The name of the product set in the project settings
my $productName = $ARGV[4];

# The default screen width of the player.
my $width = $ARGV[5];

# The default screen height of the player
my $height = $ARGV[6];

print ("\n*** Building at '$installPath' with target: $target \n");
```

Note that some languages, such as Python, pass the name of the script as one of the command line arguments. If you are using one of these languages then the arguments will effectively be shifted along one place in the array (so the install path will be in ARGV[1], etc).

In order to see this feature in action please visit the Example Projects page on our website and download the PostprocessBuildPlayer example package file and import it for use into your own project. It uses the Build Player Pipeline feature to offer customized post-processing of web player builds in order to demonstrate the types of custom build behavior you can implement in your own PostprocessBuildPlayer script.

### Windows

On Windows, the PostprocessBuildPlayer script is not supported, but you can use editor scripting to achieve the same effect. You can use BuildPipeline.BuildPlayer to run the build and then follow it with whatever postprocessing code you need:-

```
using UnityEditor;
using System.Diagnostics;

public class ScriptBatch : MonoBehaviour
{
   [MenuItem("MyTools/Windows Build With Postprocess")]
   public static void BuildGame ()
   {
      // Get filename.
      string path = EditorUtility.SaveFolderPanel("Choose Location of Built Game", "", "");

      // Build player.
      BuildPipeline.BuildPlayer(levels, path + "BuiltGame.exe", BuildTarget.StandaloneWindows, BuildOptions.None);

      // Copy a file from the project folder to the build folder, alongside the built game.
      FileUtil.CopyFileOrDirectory("Assets/WebPlayerTemplates/Readme.txt", path + "Readme.txt");

      // Run the game (Process class from System.Diagnostics).
      Process proc = new Process();
      proc.StartInfo.FileName = path + "BuiltGame.exe";
      proc.Start();
   }
}
```

Page last updated: 2012-05-04

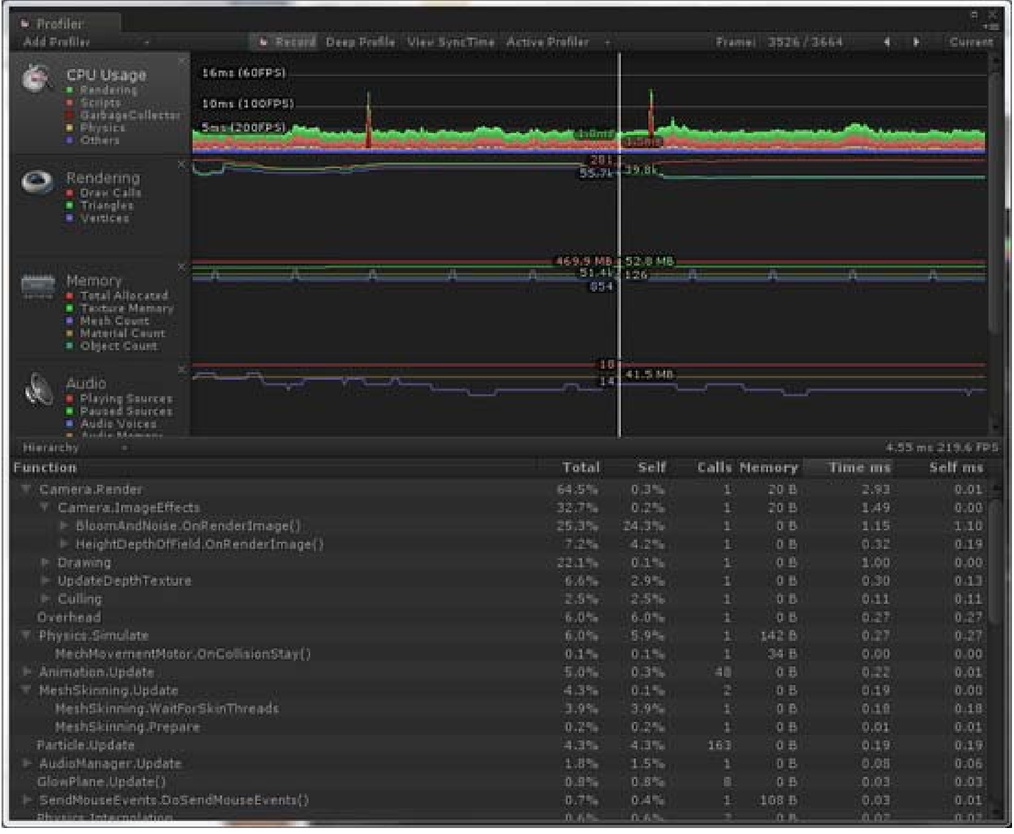
---

# Profiler

The Unity Profiler helps you to optimize your game. It reports for you how much time is spent in the various areas of your game. For example, it can report the percentage of time spent rendering, animating or in your game logic.

You can play your game in the Editor with Profiling on, and it will record performance data. The Profiler window then displays the data in a timeline, so you can see the frames or areas that spike (take more time) than others. By clicking anywhere in the timeline, the bottom section of the Profiler window will display detailed information for the selected frame.

Note that profiling has to *instrument* your code. This instrumentation has a small impact on the performance of your game. Typically this overhead is small enough to not affect the game framerate. When using profiling it is typical to consider only the ratio (or percentage) of time spent in certain areas. Also, to improve performance focus on those parts of the game that consume the most time. Compare profiling results before and after code changes and determine the improvements you measure. Sometimes changes you make to improve performance might have a negative effect on frame rate; unexpected consequences of code optimization should be expected.



*Profiler window*

### Attaching to Unity players

To profile your game running on an other device or a player running on another computer, it is possible to connect the editor to that other player. The dropdown **Active Profiler** will show all players running on the local network. These players are identified by player type and the host name running the player "iPhonePlayer (Toms iPhone)". To be able to connect to a player, the player must be launched with the **Development Build** checkbox found in the **Build Settings** dialog. From here it is also possible to tick a checkbox to make the Editor and Player Autoconnect at startup.

### Profiler Controls

Profiler controls are in the toolbar at the top of the window. Use these to turn profiling on and off, navigate through profiled frames and so on. The transport controls are at the far right end of the toolbar. Note that when the game is running and the profiler is collecting data clicking on any of these transport controls will pause the game. The controls go to the first recorded frame, step one frame back, step one frame forward and go to the last frame respectively. The profiler does not keep all recorded frames, so the notion of the *first* frame should really be though of as the oldest frame that is still kept in memory. The "current" transport button causes the profile statistics window to display data collected in real-time. The Active Profiler popup menu allows you to select whether profiling should be done in the editor or a separate player (for example, a game running on an attached iOS device).

#### Deep Profiling

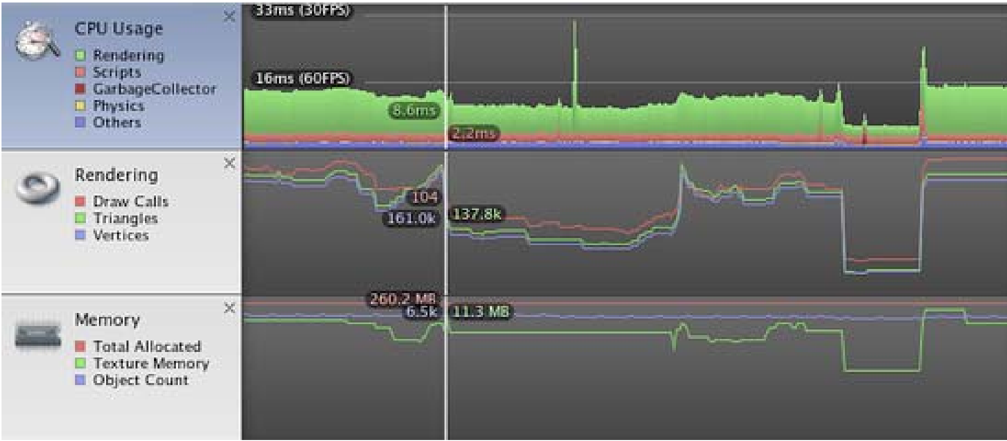
When you turn on **Deep Profile**, *all* your script code is profiled - that is, all function calls are recorded. This is useful to know where exactly time is spent in your game code.

Note that Deep Profiling incurs a **very large overhead** and uses a lot of memory, and as a result your game will run significantly slower while profiling. If you are using complex script code, Deep Profiling might not be possible at all. Deep profiling should work fast enough for small games with simple scripting. If you find that Deep Profiling for your entire game causes the frame rate to drop so much that the game barely runs, you should consider not using this approach, and instead use the approach described below. You may find deep profiling more helpful as you are designing your game and deciding how to best implement key features. Note that for large games deep profiling may cause Unity to run out of memory and so for this reason deep profiling may not be possible.

Manually profiling blocks of your script code will have a smaller overhead than using Deep Profiling. Use Profiler.BeginSample and Profiler.EndSample scripting functions to enable and disable profiling around sections of code.

**View SyncTime**
When running at a fixed framerate or running in sync with the vertical blank, Unity records the waiting time in "Wait For Target FPS". By default this amount of time is not shown in the profiler. To view how much time is spent waiting, you can toggle "View SyncTime". This is also a measure of how much headroom you have before losing frames.

## Profiler Timeline

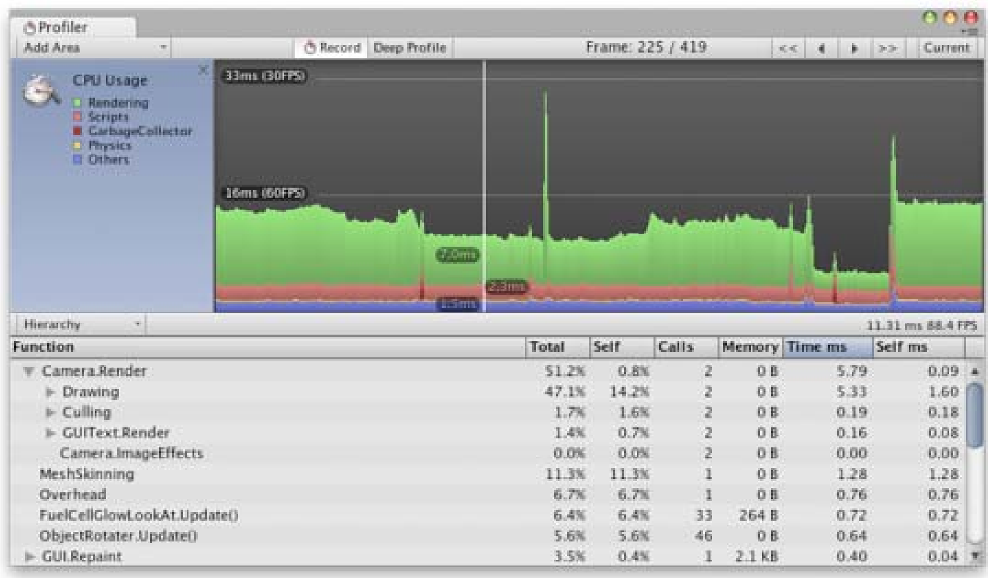



The upper part of the Profiler window displays performance data over time. When you run a game, data is recorded each frame, and the history of the last several hundred frames is displayed. Clicking on a particular frame will display it's details in the lower part of the window. Different details are displayed depending on which timeline area is currently selected.

The vertical scale of the timeline is managed automatically and will attempt to fill the vertical space of the window. Note that to get more detail in say the CPU Usage area you can remove the Memory and Rendering areas. Also, the splitter between the timeline and the statistics area can be selected and dragged downward to increase the screen area used for the timeline chart.

The timeline consists of several areas: CPU Usage, Rendering and Memory. These areas can be removed by clicking the close button in the panel, and re-added again using the *Add Area* drop down in the Profile Controls bar.

## CPU Usage Area

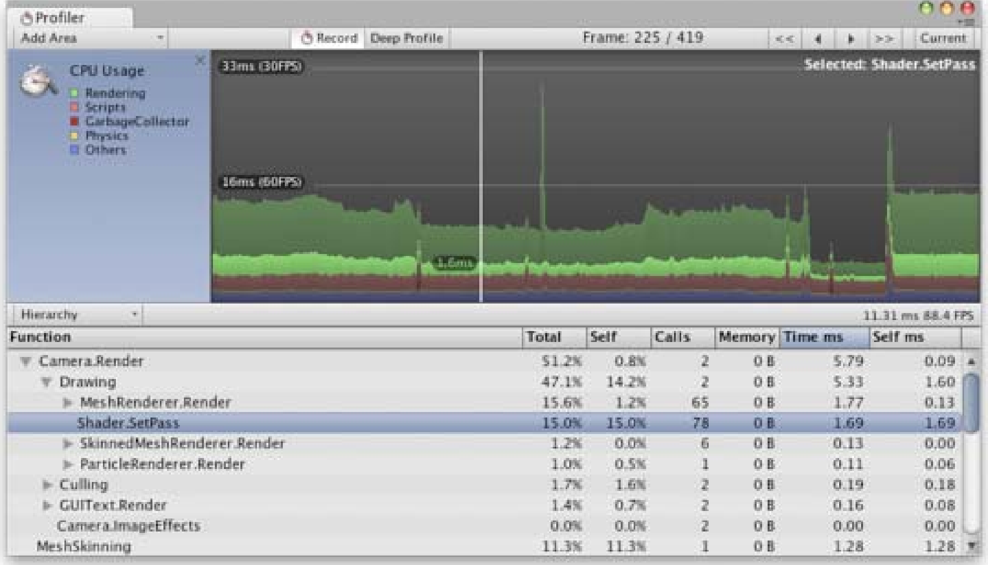



The CPU Usage area displays where time is spent in your game. When it is selected, the lower pane displays hierarchical time data for the selected frame.

- **Hierarchy mode:** Displays hierarchical time data.
- **Group Hierarchy mode:** Groups time data into logical groups (Rendering, Physics, Scripts etc.). Because children of any group can be in different group (e.g. some script might call rendering functions), the percentages of group times often add up to more than 100%. (This is not a bug.)

The way the CPU chart is stacked can be reordered by simply dragging chart labels up & down.

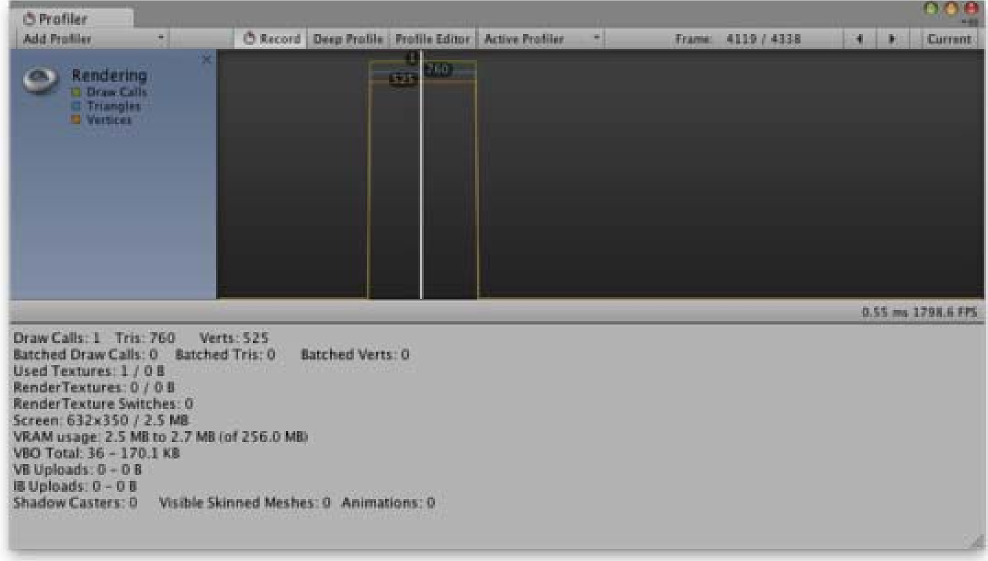
When an item is selected in the lower pane, it's contribution to the CPU chart is highlighted (and the rest are dimmed). Clicking on an item again de-selects it.




*Shader.SetPass is selected and it's contribution is highlighted in the chart.*

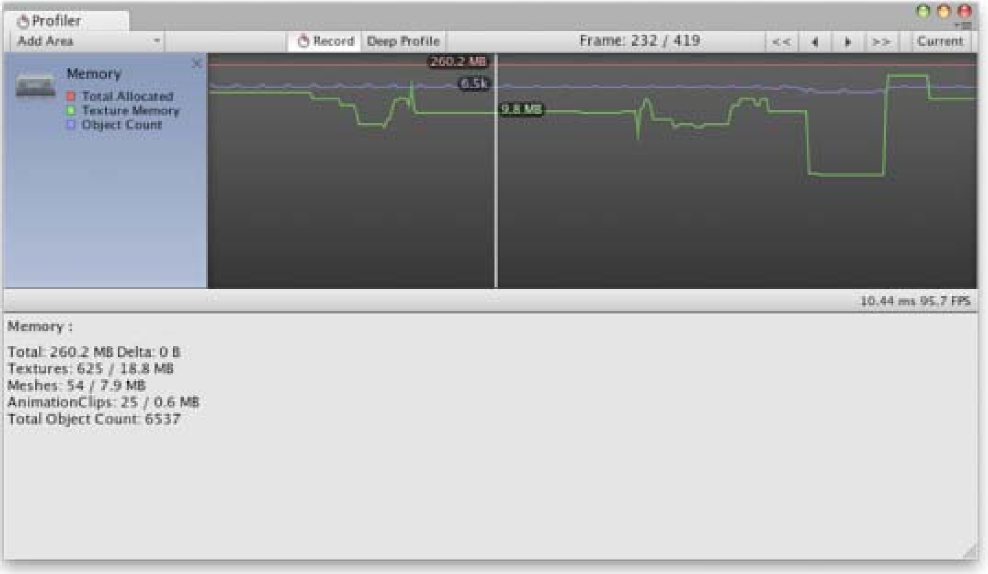
In the hierarchical time data the self time refers to the amount of time spent in a particular function not including the time spent calling sub-functions. In the screenshot above, for example 51.2% of time is spent in the Camera.Render function. This function does a lot of work and calls the various drawing and culling functions. Excluding all these functions only 0.8% of time is spent actually in the Camera.Render function.

### Rendering Area




The Rendering area displays rendering statistics. The Number of Draw Calls, Triangles and Vertices rendered is displayed graphical in the timeline. The Lower pane displays more rendering statistics and these more closely match the ones shown in the GameView Rendering Statistics window.

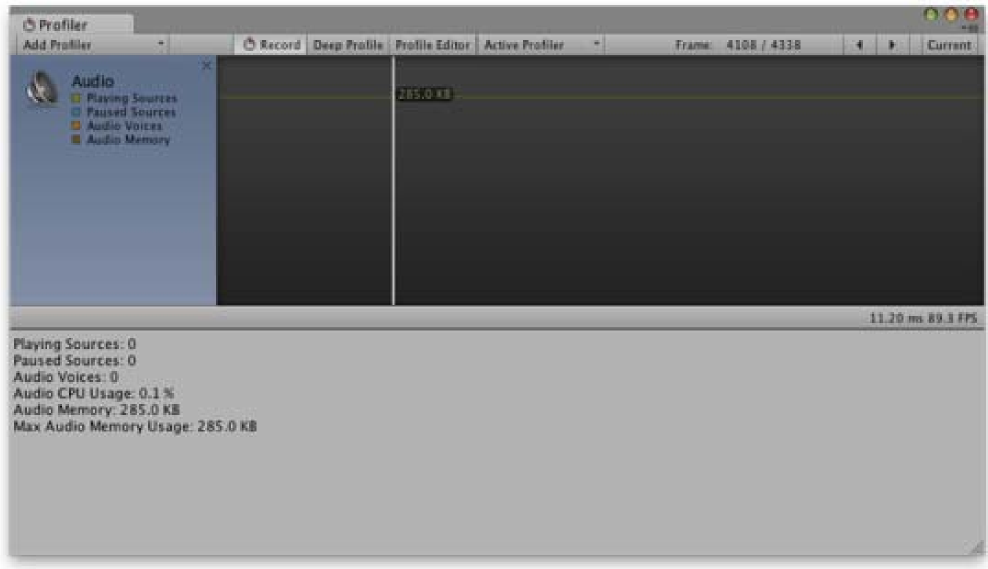
### Memory Area




The Memory area displays some memory usage data:

- **Total Allocated** is the total RAM used by the application. Note that in the Unity Editor this is memory used by everything in the editor; game builds will use much less.
- **Texture Memory** is the amount of video memory used by the textures in the current frame.
- **Object Count** is the total number of Objects that are created. If this number rises over time then it means your game is creating some objects that are never destroyed.

## Audio Area




The Audio area displays audio statistics:

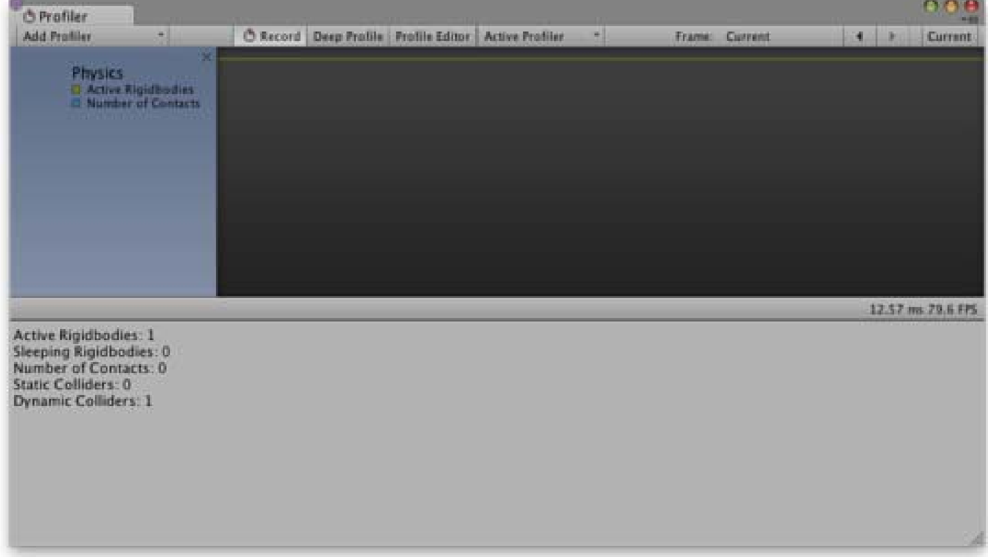
- **Playing Sources** is the total playing sources in the scene at a specific frame. Monitor this to see if audio is overloaded.
- **Paused Sources** is the total paused sources in the scene at a specific frame.
- **Audio Voice** is the actually number of audio (FMOD channels) voices used. PlayOneShot is using voices not shown in Playing Sources.
- **Audio Memory** is the total RAM used by the audio engine.

CPU usage can be seen in the bottom. Monitor this to see if Audio alone is taking up too much CPU.

**Note:** When an audio asset in Ogg Vorbis format is imported with the Compressed In Memory option, the memory usage reported by the profiler may be unexpectedly low. This happens for platforms that use FMOD audio - FMOD doesn't support Ogg Vorbis with the Compressed In Memory option, so the import setting is silently changed to Stream From Disk (which has

much lower memory overheads).

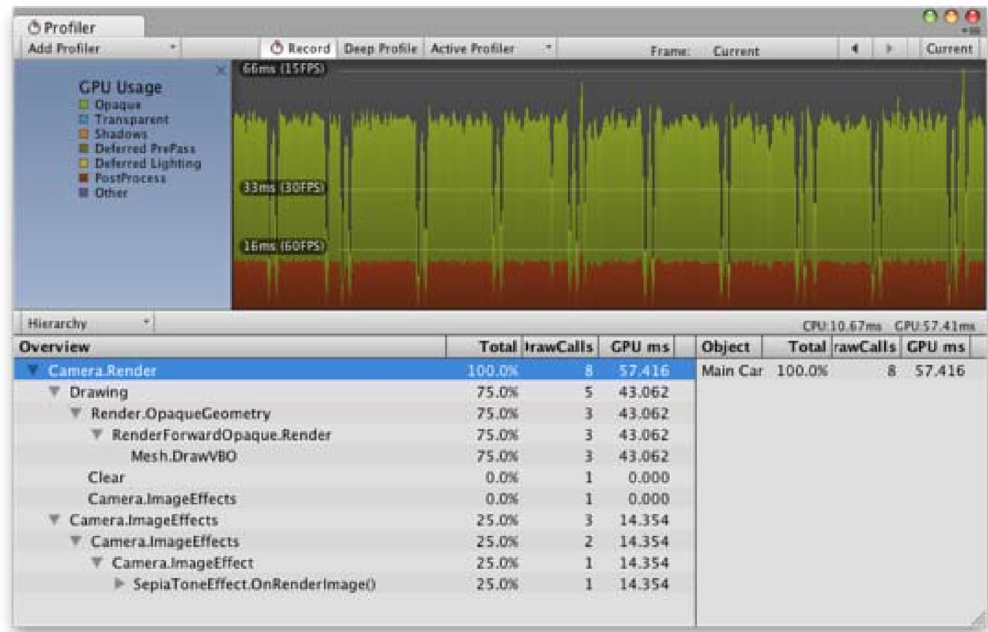
### Physics Area




The Physics area shows the following statistics about the physics in the scene:-

- **Active Rigidbodies** is the number of rigidbodies that are not currently sleeping (ie, they are moving or just coming to rest).
- **Sleeping Rigidbodies** is the number of rigidbodies that are completely at rest and therefore don't need to be updated actively by the physics engine (see Rigidbody Sleeping for further details).
- **Number of Contacts** is the total number of points of contact between all colliders in the scene.
- **Static Colliders** is the number of colliders attached to non-rigidbody objects (ie, objects which never move under physics).
- **Dynamic Colliders** is the number of colliders attached to rigidbody objects (ie, objects which do move under physics).

### GPU Area




The GPU profiler is similar to the CPU profiler with the various contributions to rendering time shown as a hierarchy in the bottom panel. Selecting an item from the hierarchy will show a breakdown in the panel to the right.

Please note that on the Mac, GPU profiling is only available under OSX 10.7 Lion and later versions.

### See Also

- Optimizing Graphics Performance page.

### iOS

Remote profiling can be enabled on iOS devices by following these steps:

1. Connect your iOS device to your WiFi network (local/adhoc WiFi network is used by profiler to send profiling data from device to the Unity Editor).
2. Check "Autoconnect Profiler" checkbox in Unity's build settings dialog.
3. Attach your device to your Mac via cable and hit "Build & Run" in Unity Editor.
4. When app launches on device open profiler window in Unity Editor (Window->Profiler)

If you are using a firewall, you need to make sure that ports 54998 to 55511 are open in the firewall's outbound rules - these are the ports used by Unity for remote profiling.

**Note:** Sometimes Unity Editor might not autoconnect to the device. In such cases profiler connection might be initiated from Profiler Window **Active Profiler** drop down menu by select appropriate device.

### Android

Remote profiling can be enabled on Android devices through two different paths : WiFi or ADB.

For WiFi profiling, follow these steps:

1. Make sure to disable Mobile Data on your Android device.
2. Connect your Android device to your WiFi network.
3. Check the "Autoconnect Profiler" checkbox in Unity's build settings dialog.
4. Attach your device to your Mac/PC via cable and hit "Build & Run" in Unity Editor.
5. When the app launches on the device, open the profiler window in Unity Editor (Window->Profiler)
6. If the Unity Editor fails to autoconnect to the device, select the appropriate device from the Profiler Window **Active Profiler** drop down menu.

**Note:** The Android device and host computer (running the Unity Editor) must both be on the same subnet for the device detection to work.

For ADB profiling, follow these steps:

- Attach your device to your Mac/PC via cable and make sure ADB recognizes the device (i.e. it shows in *adb devices* list).
- Open a Terminal window / CMD prompt and enter

```
adb forward tcp:54999 localabstract:Unity-<insert bundle identifier here>
```

- Check the "Development Build" checkbox in Unity's build settings dialog, and hit "Build & Run".
- When the app launches on the device, open the profiler window in Unity Editor (Window->Profiler)
- Select the *AndroidProfiler(ADB@127.0.0.1:54999)* from the Profiler Window **Active Profiler** drop down menu.

**Note:** The entry in the drop down menu is only visible when the selected target is Android.

If you are using a firewall, you need to make sure that ports 54998 to 55511 are open in the firewall's outbound rules - these are the ports used by Unity for remote profiling.

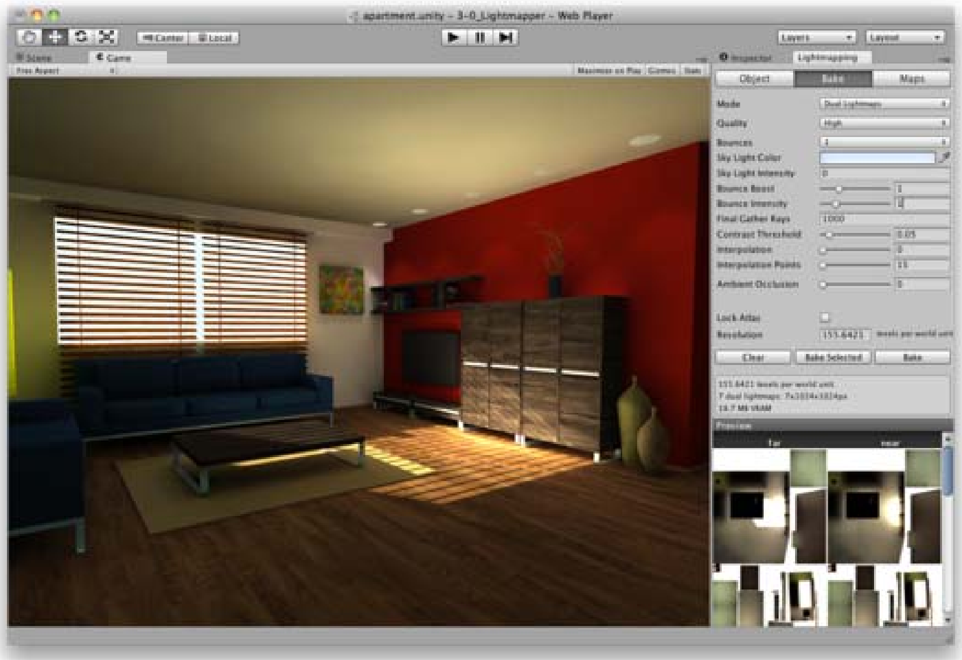
Page last updated: 2012-10-31

---

# Lightmapping

This an introductory description of lightmapping in Unity. For more advanced topics see in-depth description of lightmapping in Unity

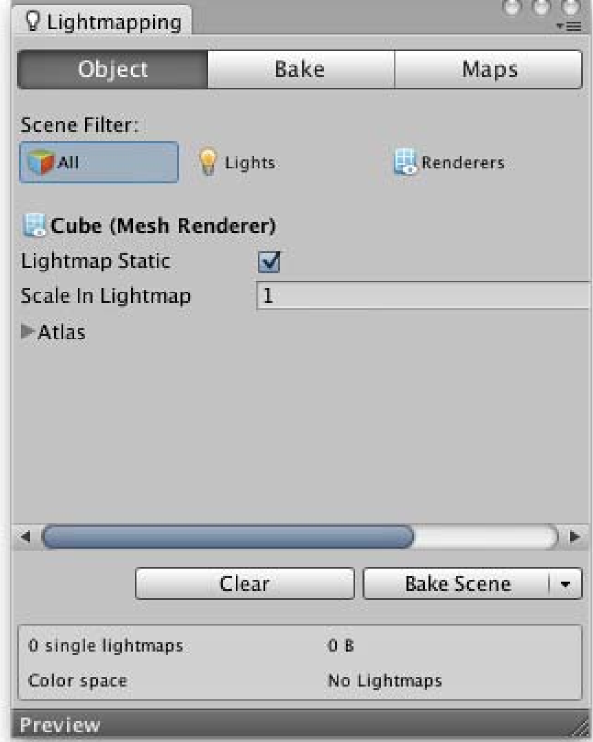
Unity has a built-in lightmapper: it's **Beast** by Illuminate Labs. Lightmapping is fully integrated in Unity. This means that Beast will bake lightmaps for your scene based on how your scene is set up within Unity, taking into account meshes, materials, textures and lights. It also means that lightmapping is now an integral part of the rendering engine - once your lightmaps are created you don't need to do anything else, they will be automatically picked up by the objects.



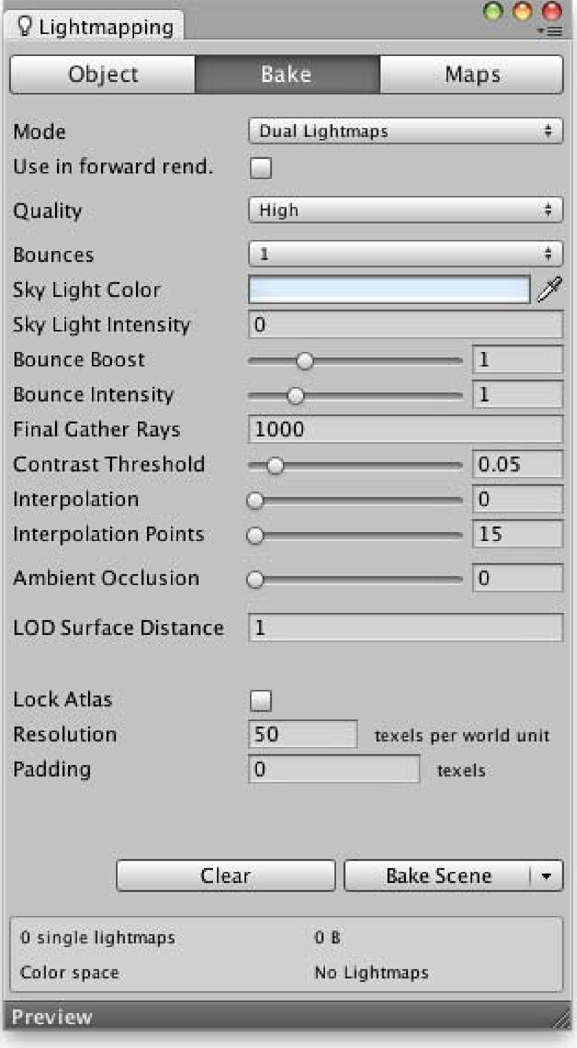
## Preparing the scene and baking the lightmaps

Selecting **Window – Lightmapping** from the menu will open the Lightmapping window:

1. Make sure any mesh you want to be lightmapped has proper UVs for lightmapping. The easiest way is to choose the **Generate Lightmap UVs** option in mesh import settings.
2. In the **Object** pane mark any Mesh Renderer, Skinned Mesh Renderer or Terrain as **static** – this will tell Unity, that those objects won't move nor change and they can be lightmapped.




3. To control the resolution of the lightmaps, go to the **Bake** pane and adjust the **Resolution** value. (To have a better understanding on how you spend your lightmap texels, look at the small **Lightmap Display** window within the **Scene View** and select **Show Resolution**).



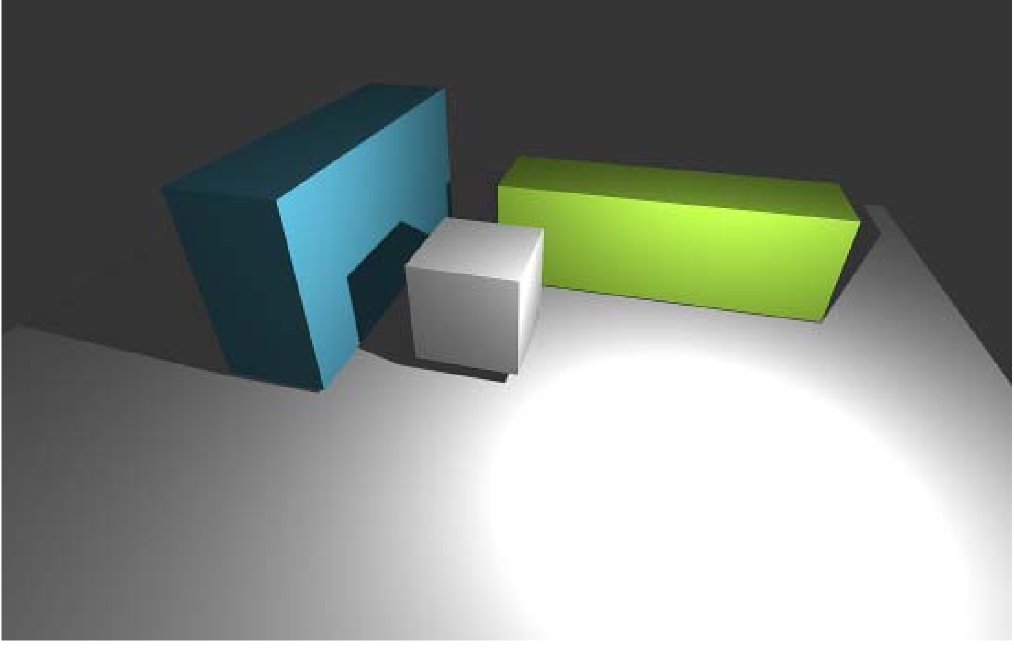

4. Press **Bake**
5. A progress bar appears in Unity Editor's status bar, in the bottom right corner.
6. When baking is done, you can see all the baked lightmaps at the bottom of the Lightmap Editor window.

Scene and game views will update - your scene is now lightmapped!

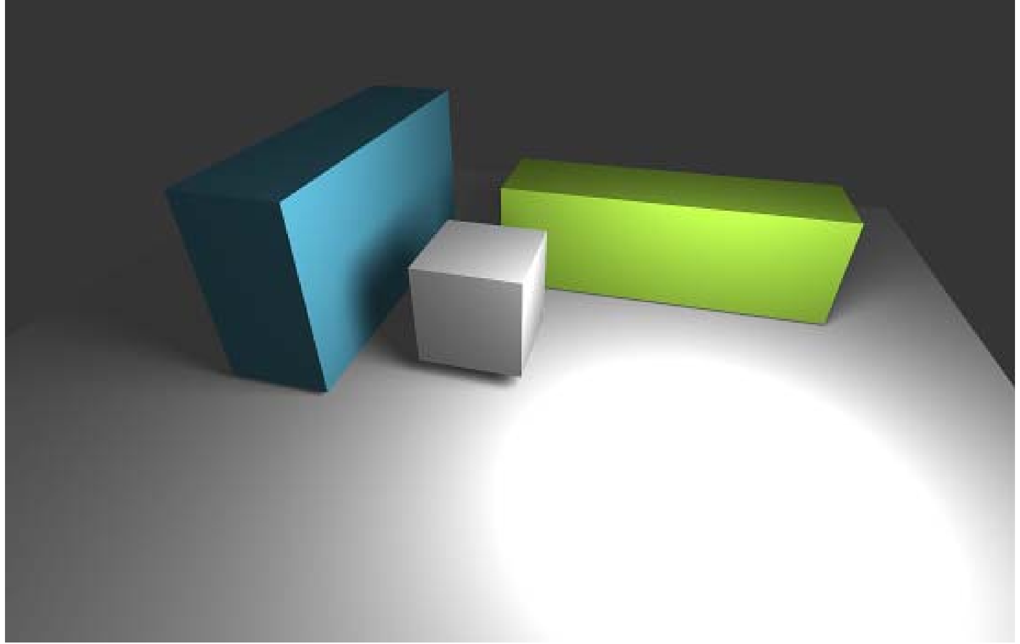
### Tweaking Bake Settings

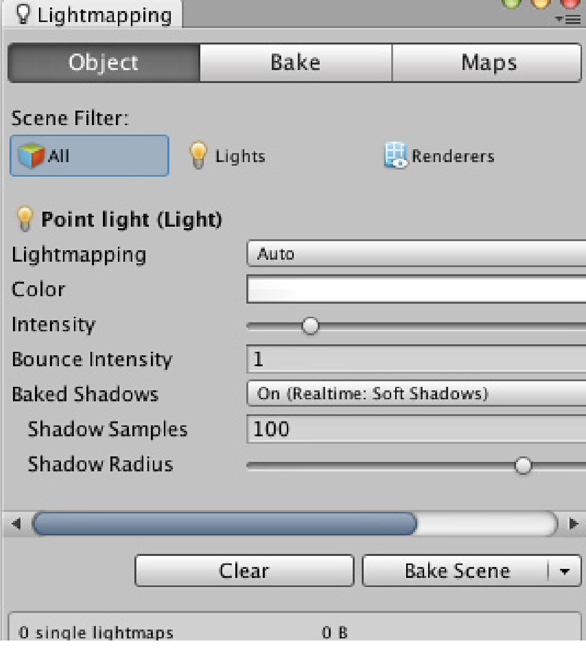
Final look of your scene depends a lot on your lighting setup and bake settings. Let's take a look at an example of some basic settings that can improve lighting quality.

This is a basic scene with a couple of cubes and one point light in the centre. The light is casting hard shadows and the effect is quite dull and artificial.

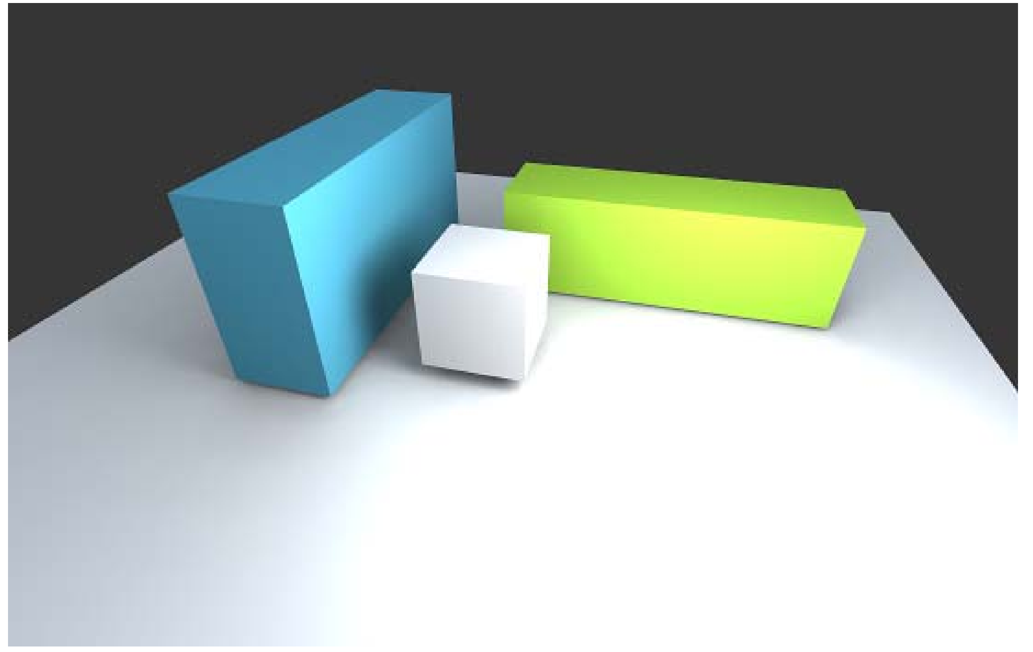


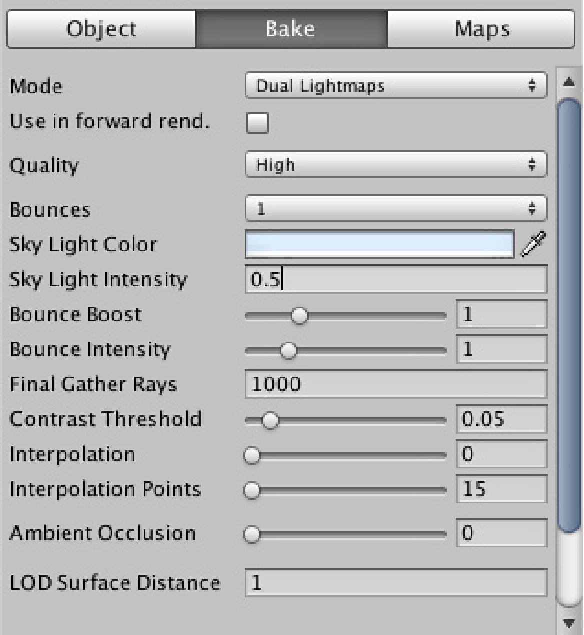
Selecting the light and opening the **Object** pane of the **Lightmapping** window exposes **Shadow Radius** and **Shadow Samples** properties. Setting Shadow Radius to 1.2, Shadow Samples to 100 and re-baking produces soft shadows with wide penumbra - our image already looks much better.






With Unity Pro we can take the scene one step further by enabling Global Illumination and adding a Sky Light. In the **Bake** pane we set the number of **Bounces** to 1 and the **Sky Light Intensity** to 0.5. The result is much softer lighting with subtle diffuse interreflection effects (color bleeding from the green and blue cubes) - much nicer and it's still only 3 cubes and a light!






### Lightmapping In-Depth

For more information about the various lightmapping-related settings, please refer to the in-depth description of lightmapping in Unity.

Page last updated: 2012-10-30

---

## LightmappingInDepth

If you are about to lightmap your first scene in Unity, this Quickstart Guide might help you out.

Lightmapping is fully integrated in Unity, so that you can build entire levels from within the Editor, lightmap them and have your materials automatically pick up the lightmaps without you having to worry about it. Lightmapping in Unity means that all your lights' properties will be mapped directly to the Beast lightmapper and baked into textures for great performance. Unity Pro extends this functionality by Global Illumination, allowing for baking realistic and beautiful lighting, that would otherwise be impossible in realtime. Additionally Unity Pro brings you sky lights and emissive materials for even more interesting scene lighting.

In this page you will find a more in-depth description of all the attributes that you can find in the Lightmapping window. To open the Lightmapping window select **Window – Lightmapping**.

### Scene filters

At the top of the inspector are three *Scene Filter* buttons that enable you to apply the operation to all objects or to restrict it to lights or renderers.

### Object

Per-object bake settings for lights, mesh renderers and terrains - depending on the current selection.

**Mesh Renderers and Terrains:**

| | |
|---|---|
| **Lightmap Static** | Mesh Renderers and Terrains have to marked as static to be lightmapped. |
| **Scale In Lightmap** | (Mesh Renderers only) Bigger value will result in more resolution to be dedicated to the give mesh renderer. The final resolution will be proportional (Scale in lightmap)*(Object's word-space surface area)*(Global bake settings Resolution value). A value of 0 will result in the object not being lightmapped (it will still affect other lightmapped objects). |
| **Lightmap Size** | (Terrains only) Lightmap size for this terrain instance. Terrains are not atlased as other objects - they get their individual lightmaps instead. |
| **Atlas** | Atlasing information – will be updated automatically, if **Lock Atlas** is disabled. If **Lock Atlas** is enabled, those parameters won't be modified automatically and can be edited manually. |
| **Lightmap Index** | An index into the lightmap array. |
| **Tiling** | (Mesh Renderers only) Tiling of object's lightmap UVs. |
| **Offset** | (Mesh Renderers only) Offset of object's lightmap UVs. |

**Lights:**

| | |
|---|---|
| **Lightmapping** | The Lightmapping mode: Realtime Only, Auto or Baked Only. See Dual Lightmaps description below. |
| **Color** | The color of the light. Same property is used for realtime rendering. |
| **Intensity** | The intensity of the light. Same property is used for realtime rendering. |
| **Bounce Intensity** | A multiplier to the intensity of indirect light emitted from this particular light source. |
| **Baked Shadows** | Controls whether shadows are casted from objects lit by this light (controls realtime shadows at the same time in case of Auto lights). |
| **Shadow Radius** | (Point and Spot lights only) Increase this value for soft direct shadows - it increases the size of the light for the shadowing (but not lighting) calculations. |
| **Shadow Angle** | (Directional lights only) Increase this value for soft direct shadows - it increases the angular coverage of the light for the shadowing (but not lighting) calculations. |
| **Shadow Samples** | If you've set Shadow Radius or Angle above zero, increase the number of Shadow Samples as well. Higher sample numbers remove noise from the shadow penumbra, but might increase rendering times. |

### Bake

Global bake settings.

| | |
|---|---|
| **Mode** | Controls both offline lightmap baking and runtime lightmap rendering modes. In Dual Lightmaps mode both near and far lightmaps will be baked; only deferred rendering path supports rendering dual lightmaps. Single Lightmaps mode results in only the far lightmap being baked; can also be used to force single lightmaps mode for the deferred rendering path. |
| **Use in forward rendering** | (Dual lightmaps only) Enables dual lightmaps in forward rendering. Note that this will require you to create your own shaders for the purpose. |
| **Quality** | Presets for high (good-looking) and low (but fast) quality bakes. They affect the number of final gather rays, contrast threshold and some other final gather and anti-aliasing settings. |
| **Bounces** | The number of light bounces in the Global Illumination simulation. At least one bounce is needed to give a soft, realistic indirect lighting. 0 means only direct light will be computed. |
| **Sky Light Color** | Sky light simulates light emitted from the sky from all the directions - great for outdoor scenes. |
| **Sky Light Intensity** | The intensity of the sky light - a value of 0 disables the sky light. |
| **Bounce Boost** | Boosts indirect light, can be used to increase the amount of bounced light within the scene without burning out the render too quickly. |
| **Bounce Intensity** | A multiplier to the intensity of the indirect light. |
| **Final Gather Rays** | The number of rays shot from every final gather point - higher values give better quality. |
| **Contrast Threshold** | Color contrast threshold, above which new final gather points will be created by the adaptive sampling algorithm. Higher values make Beast be more tolerant about illumination changes on the surface, thus producing smoother but less-detailed lightmaps. Lower numbers of final gather rays might need higher contrast threshold values not to force additional final gather points to be created. |
| **Interpolation** | Controls the way the color from final gather points will be interpolated. 0 for linear interpolation, 1 for advanced, gradient-based interpolation. In some cases the latter might introduce artifacts. |
| **Interpolation Points** | The number of final gather points to interpolate between. Higher values give more smooth results, but can also smooth out details in the lighting. |
| **Ambient Occlusion** | The amount of ambient occlusion to be baked into the lightmaps. Ambient occlusion is the visibility function integrated over the local hemisphere of size Max Distance, so doesn't take into account any lighting information. |
| **Lock Atlas** | When Lock Atlas is enabled, automatic atlasing won't be run and lightmap index, tiling and offset on the objects won't be modified. |
| **Resolution** | The resolution of the lightmaps in texels per world unit, so a value of 50 and a 10unit by 10unit plane will result in the plane occupying 500x500 pixels in the lightmap. |
| **Padding** | The blank space left between individual items on the atlas, given in texel units (0..1). |

### Maps

The editable array of all the lightmaps.

| | |
|---|---|
| **Compressed** | Toggles compression on all lightmap assets for this scene. |
| **Array Size** | Size of the lightmaps array (0 to 254). |
| **Lightmaps Array** | The editable array of all the lightmaps in the current scene. Unassigned slots are treated as black lightmaps. Indices correspond to the Lightmap Index value on Mesh Renderers and Terrains. Unless Lock Atlas is enabled, this array will get auto-resized and populated whenever you bake lightmaps. |

### Lightmap Display

Utilities for controlling how lightmaps are displayed in the editor. Lightmap Display is a sub window of the Scene View, visible whenever the Lightmapping window is visible.

| | |
|---|---|
| **Use Lightmaps** | Whether to use the lightmaps during the rendering or not. |
| **Shadow Distance** | The distance at which Auto lights and Close By lightmaps fade out to just Far Away lightmaps. This setting overrides but not overwrites the QualitySettings.shadowDistance setting. |
| **Show Resolution** | Toggles the scene view Lightmap Resolution mode, which allows you to preview how you spend your lightmap texels on objects marked as static. |

### Details

#### Dual Lightmaps

Dual lightmaps is Unity's approach to make lightmapping work with **specular**, **normal mapping** and proper blending of baked and realtime shadows. It's also a way to make your lightmaps look good even if the lightmap resolution is low.

Dual lightmaps by default can only be used in the Deferred Lighting rendering path. In Forward rendering path, it's possible to

enable Dual Lightmaps by writing custom shaders (use `dualforward` surface shader directive).

Dual lightmaps use two sets of lightmaps:

- **Far**: Contains full illumination
- **Near**: Contains indirect illumination from lights marked as **Auto**, full illumination from lights marked as **Bake Only**, emissive materials and sky lights.

**Realtime Only** lights are never baked. The **Near** lightmap set is used within the distance from the camera smaller than the Shadow Distance quality setting.
Within this distance **Auto** lights are rendered as realtime lights with specular bump and realtime shadows (this makes their shadows blend correctly with shadows from Realtime Only lights) and their indirect light is taken from the lightmap. Outside Shadow Distance **Auto** lights no longer render in realtime and full illumination is taken from the **Far** lightmap (**Realtime Only** lights are still there, but with disabled shadows).

The scene below contains one directional light with lightmapping mode set to the default Auto, a number of static lightmapped objects (buildings, obstacles, immovable details) and some dynamic moving or movable objects (dummies with guns, barrels). The scene is baked and rendered in dual lightmaps mode: behind the shadow distance buildings are fully lit only by lightmaps, while the two dummies are dynamically lit but don't cast shadows anymore; in front of the shadow distance both the dummy and static lightmapped buildings and ground are lit in realtime and cast realtime shadows, but the soft indirect light comes from the near lightmap.

Correct blend between shadows from static objects (the building) and dynamic objects (the dummy)

Bumps and specular highlights even from a light that is lightmapped

Good shadow resolution, even though lightmap resolution is much lower

**Single Lightmaps**

Single Lightmaps is a much simpler technique, but it can be used in any rendering path. All static illumination (i.e. from baked only and auto lights, sky lights and emissive materials) gets baked into one set of lightmaps. These lightmaps are used on all lightmapped objects regardless of shadow distance.

To match the strength of dynamic shadows to baked shadows, you need to manually adjust the **Shadow Strength** property of your light:

*Adjusting Shadow Strength of a light from the original value of 1.0 to 0.7.*

**Lightmapped Materials**

Unity doesn't require you to select special materials to use lightmaps. Any shader from the built-in shaders (and any Surface Shader you write, for that matter) already supports lightmaps out of box, without you having to worry about it - it just works.

**Lightmap Resolution**

With the *Resolution* bake setting you control how many texels per unit are needed for your scene to look good. If there's a 1x1 unit plane in your scene and the resolution is set to 10 texels per unit, your plane will take up 10x10 texels in the lightmap. Resolution bake setting is a global setting. If you want to modify it for a special object (make it very small or very big in the lightmap) you can use **Scale in Lightmap** property of Mesh Renderers. Setting Scale in Lightmap to 0 will result in the object not being lightmapped at all (it will still influence lightmaps on other objects). Use the *Lightmap Resolution* scene view render mode to preview how you spend your lightmap texels.

*Lightmap Resolution scene view mode visualising how the lightmap texels are spent (each square is one texel).*

**UVs**

A mesh that you're about to lightmap needs to have UVs suitable for lightmapping. The easiest way to ensure that is to enable the Generate Lightmap UVs option in Mesh Import Settings for a given mesh.

For more information see the Lightmap UVs page.

**Material Properties**

The following material properties are mapped to Beast's internal scene representation:

- Color

- Main Texture
- Specular Color
- Shininess
- Transparency
  - Alpha-based: when using a transparent shader, main texture's alpha channel will control the transparency
  - Color-based: Beast's RGB transparency can be enabled by adding a texture property called **_TransparencyLM** to the shader. Bear in mind that this transparency is defined in the opposite way compared to the alpha-based transparency: here a pixel with value (1, 0, 0) will be fully transparent to red light component and fully opaque to green and blue component, which will result in a red shadow; for the same reason white texture will be fully transparent, while black texture - fully opaque.
- Emission
  - Self Illuminated materials will emit light tinted by the Color and Main Texture and masked by the Illum texture. The intensity of emitted light is proportional to the Emission property (0 disables emission).
  - Generally large and dim light sources can be modeled as objects with emissive materials. For small and intense lights normal light types should be used, since emissive materials might introduce noise in the rendering.

*Note: When mapping materials to Beast, Unity detects the 'kind' of the shader by the shader's properties and path/name keywords such as: 'Specular', 'Transparent', 'Self-Illumin', etc.*

#### Skinned Mesh Renderers

Having skinned meshes that are static makes your content more flexible, since the shape of those meshes can be changed in Unity after import and can be tweaked per level. Skinned Mesh Renderers can be lightmapped in exactly the same way as Mesh Renderers and are sent to the lightmapper in their current pose.

Lightmapping can also be used if the vertices of a mesh are moved at runtime a bit -- the lighting won't be completely accurate, but in a lot of cases it will match well enough.

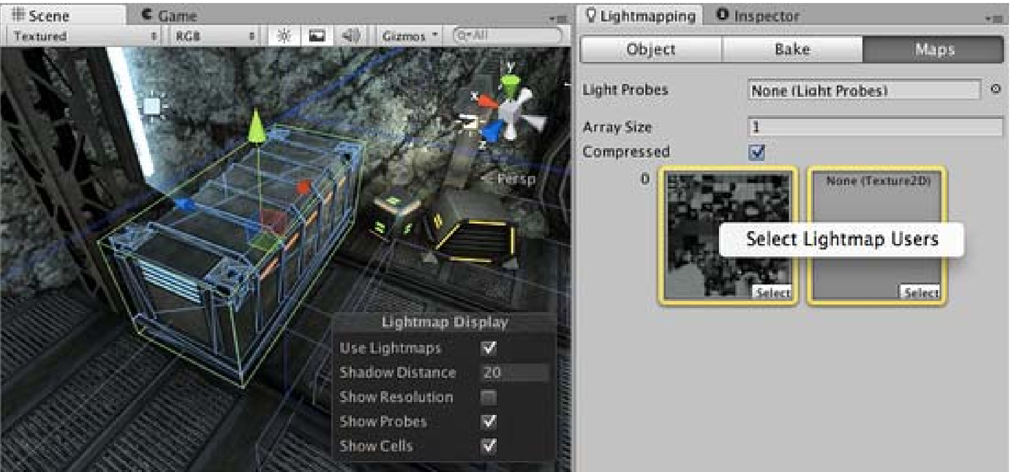
### Advanced

#### Automatic Atlasing

Atlasing (UV-packing) is performed automatically every time you perform a bake and normally you don't have to worry about it - it just works.

Object's world-space surface area is multiplied by the per-object Scale In Lightmap value and by the global Resolution and the result determines the size of the object's UV set (more precisely: the size of the [0,1]x[0,1] UV square) in the lightmap. Next, all objects are packed into as few lightmaps as possible, while making sure each of them occupies the amount of space calculated in the previous step. If a UV set for a given object occupies only part of the [0,1]x[0,1] square, in many cases atlasing will move neighboring UV sets closer, to make use of the empty space.

As a result of atlasing, every object to be lightmapped has it's place in one of the lightmaps and that space doesn't overlap with any other object's space. The atlasing information is stored as three values: Lightmap Index, Tiling (scale) and Offset in Mesh Renderers and as one value: Lightmap Index in Terrains and can be viewed and modified via the Object pane of the lightmapping window.



*Right-clicking a lightmap lets you select all game objects that use the chosen lightmap. Lightmaps of the active object from the current selection are highlighted in yellow.*

Atlasing can only modify per-object data which is Lightmap Index, Tiling and Offset and can not modify the UV set of an object, as the UV set is stored as part of the shared mesh. Lightmap UVs for a mesh can only be created at import time using Unity's built-in auto-unwrapper or in an external 3D package before importing to Unity.

**Lock Atlas**
When Lock Atlas is enabled, automatic atlasing won't be run and Lightmap Index, Tiling and Offset on the objects won't be modified. Beast will rely on whatever is the current atlasing, so it's the user's responsibility to maintain correct atlasing (e.g. no overlapping objects in the lightmaps, no objects referencing a lightmap slot past the lightmap array end, etc.).

Lock Atlas opens up the possibility for alternative workflows when sending your object's for lightmapping. You can then perform atlasing manually or via scripting to fit you specific needs; you can also lock the automatically generated atlasing if you are happy with the current atlasing, have baked more sets of lightmaps for your scene and want to make sure, that after adding one more object to the scene the atlasing won't change making the scene incompatible with other lightmap sets.

Remember that Lock Atlas locks only atlasing, not the mesh UVs. If you change your source mesh and the mesh importer is set to generate lightmap UVs, the UVs might be generated differently and your current lightmap will look incorrectly on the object - to fix this you will need to re-bake the lightmap.

**Custom Beast bake settings**
If you need even more control over the bake process, see the custom Beast settings page.

Page last updated: 2012-10-31

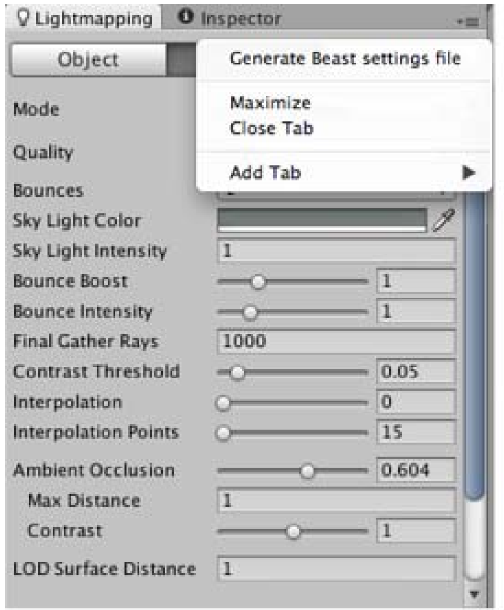
---

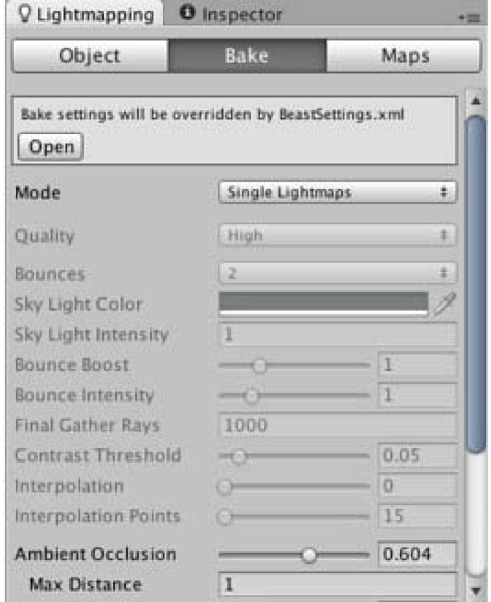
# LightmappingCustomSettings

If you need a different baking setup than the one Unity is using by default, you can specify it by using custom Beast settings.

Beast reads bake settings defined in XML format. Normally Unity generates the XML file based on the configuration you have chosen in Bake pane of the Lightmap Editor window and a number of other internal settings. You can override those settings by specifying your own settings in Beast's XML format.

To have Unity automatically generate the XML file for you, click the tab menu in the upper-right corner of the Lightmap Editor window and select *Generate Beast settings file*. You will notice that the BeastSettings.xml file appeared in the project next to your lightmaps and that the Lightmap Editor informs you, that your XML settings will override Unity's settings during the next bake. Click the *open* button to edit your custom settings.




A sample Beast configuration file is given below:-

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ILConfig>
  <AASettings>
    <samplingMode>Adaptive</samplingMode>
    <clamp>false</clamp>
    <contrast>0.1</contrast>
    <diagnose>false</diagnose>
    <minSampleRate>0</minSampleRate>
    <maxSampleRate>2</maxSampleRate>
    <filter>Gauss</filter>
    <filterSize>
      <x>2.2</x>
      <y>2.2</y>
    </filterSize>
  </AASettings>
  <RenderSettings>
    <bias>0</bias>
    <maxShadowRays>10000</maxShadowRays>
    <maxRayDepth>6</maxRayDepth>
  </RenderSettings>
  <EnvironmentSettings>
    <giEnvironment>SkyLight</giEnvironment>
    <skyLightColor>
      <r>0.86</r>
      <g>0.93</g>
      <b>1</b>
      <a>1</a>
    </skyLightColor>
    <giEnvironmentIntensity>0</giEnvironmentIntensity>
  </EnvironmentSettings>
  <FrameSettings>
    <inputGamma>1</inputGamma>
  </FrameSettings>
  <GISettings>
    <enableGI>true</enableGI>
    <fgPreview>false</fgPreview>
    <fgRays>1000</fgRays>
    <fgContrastThreshold>0.05</fgContrastThreshold>
    <fgGradientThreshold>0</fgGradientThreshold>
    <fgCheckVisibility>true</fgCheckVisibility>
    <fgInterpolationPoints>15</fgInterpolationPoints>
    <fgDepth>1</fgDepth>
    <primaryIntegrator>FinalGather</primaryIntegrator>
    <primaryIntensity>1</primaryIntensity>
    <primarySaturation>1</primarySaturation>
    <secondaryIntegrator>None</secondaryIntegrator>
    <secondaryIntensity>1</secondaryIntensity>
    <secondarySaturation>1</secondarySaturation>
    <fgAOInfluence>0</fgAOInfluence>
    <fgAOMaxDistance>0.223798</fgAOMaxDistance>
    <fgAOContrast>1</fgAOContrast>
    <fgAOScale>2.0525</fgAOScale>
  </GISettings>
  <SurfaceTransferSettings>
    <frontRange>0.0</frontRange>
    <frontBias>0.0</frontBias>
    <backRange>2.0</backRange>
    <backBias>-1.0</backBias>
    <selectionMode>Normal</selectionMode>
  </SurfaceTransferSettings>
```

```
  <TextureBakeSettings>
    <bgColor>
      <r>1</r>
      <g>1</g>
      <b>1</b>
      <a>1</a>
    </bgColor>
    <bilinearFilter>true</bilinearFilter>
    <conservativeRasterization>true</conservativeRasterization>
    <edgeDilation>3</edgeDilation>
  </TextureBakeSettings>
</ILConfig>
```

The toplevel XML elements are described in the sections below along with their subelements.

### Adaptive Sampling (*<AASettings>* element)

Beast uses an adaptive sampling scheme when sampling light maps. The light must differ more than a user set contrast threshold for Beast to place additional samples in an area. The sample area is defined by a Min and Max sample rate. The user sets the rate in the -4..4 range which means that Beast samples from 1/256 sample per pixel to 256 samples per pixel (the formula is: 4 to the power of samplerate). It is recommended to use at least one sample per pixel for production use (Min sample rate = 0). Undersampling is most useful when doing camera renders or baking textures with big UV-patches. When Beast has taken all necessary samples for an area, the final pixel value is weighed together using a filter. The look the filter produces is dependent on the filter type used and the size of the filter kernel. The available filters are:

- Box: Each sample is treated as equally important. The fastest filter to execute but it gives blurry results.
- Triangle: The filter kernel is a tent which means that distant samples are consideredless important.
- Gauss: Uses the Gauss function as filter kernel. This gives the best results (removes noise, preserves details).

There are more filters available, but these three are the most useful. The kernel (filter) size is given in pixels in the range 1..3. Beast actually uses all sub pixels when filtering, which yields better results than doing it afterwards in Photoshop.

**AASettings**

| | |
|---|---|
| **samplingMode** | The sampling strategy to use. Default is Adaptive. Adaptive: Adaptive anti-aliasing scheme for under/over sampling (from 1/256 up to 256 samples per pixel). SuperSampling: Anti-aliasing scheme for super sampling (from 1 up to 128 samples per pixel). |
| **minSampleRate** | Sets the min sample rate, default is 0 (ie one sample per pixel). |
| **maxSampleRate** | Sets the max sample rate, the formula used is 4^maxSampleRate (1, 4, 16, 64, 256 samples per pixel) |
| **contrast** | The contrast value which controls if more samples are necessary - a lower value forces more samples. |
| **filter** | Sets which filter type to use. Most useful ones for Baking are Box, Triangle and Gauss. |
| **filterSize** | Sets the filter size in pixels, from 1 to 3. |
| **diagnose** | Enable to diagnose the sampling. The brighter a pixel is, the more samples were taken at that position. |

### Texture Bake (*<TextureBakeSettings>* element)

These settings help getting rid of any artifacts that are purely related to how lightmaps are rasterized and read from a texture.

**TextureBakeSettings**

| | |
|---|---|
| **edgeDilation** | Expands the rendered region with the number of pixels specified. This is needed to prevent the artifacts occurring when GPU filters in empty pixels from around the rendered region. Should be set to 0 though, since a better algorithm is part of the import pipeline. |
| **bilinearFilter** | Is used to make sure that the data in the lightmap is "correct" when the GPU applies bilinear filtering. This is most noticable when the atlases are tightly packed. If there is only one pixel between two different UV patches, the bilinear functionality in Beast will make sure the that pixel is filled with the color from the correct patch. This minimizes light seams. |
| **conservativeRasterization** | Is used when the UV-chart does not cover the entire pixel. If such a layout is used, Beast may miss the texel by mistake. If conservative rasterization is used Beast will guarantee that it will find a UV-layout if present. Note that Beast will pick any UV-layout in the pixel. Conservative Rasterization often needs to be turned on if the UV atlases are tightly packed in low resolutions or if there are very thin objects present. |
| **bgColor** | The background color of the lightmap. Should be set to white (1,1,1,1). |

### Environment (<*EnvironmentSettings*> element)

The environment settings in Beast control what happens if a ray misses all geometry in the scene. The environment can either be a constant color or an HDR image in lat-long format for Image Based Lighting (IBL). Note that environments should only be used for effects that can be considered to be infinitely far away, meaning that only the directional component matters.

Defining an environment is usually a very good way to get very pleasing outdoor illumination results, but might also increase bake times.

**EnvironmentSettings**

| | |
|---|---|
| **giEnvironment** | The type of Environment: None, Skylight or IBL. |
| **giEnvironmentIntensity** | A scale factor for the intensity, used for avoiding gamma correction errors and to scale HDR textures to something that fits your scene. (in Unity: *Sky Light Intensity*) |
| **skyLightColor** | A constant environment color. Used if type is Skylight. It is often a good idea to keep the color below 1.0 in intensity to avoid boosting by gamma correction. Boost the intensity instead with the giEnvironmentIntensity setting. (in Unity: *Sky Light Color*) |
| **iblImageFile** | High-dynamic range IBL background image in Long-Lat format, .HDR or .EXR, absolute path. |

### Render Settings/Shadows (<*RenderSettings*> element)

Settings for ray-traced shadows.

**RenderSettings**

| | |
|---|---|
| **bias** | An error threshold to avoid double intersections of shadow rays. For example, a shadow ray should not intersect the same triangle as the primary ray did, but because of limited numerical precision this can happen. The bias value moves the intersection point to eliminate this problem. If set to zero this value is computed automatically depending on the scene size. |
| **maxShadowRays** | The maximum number of shadow rays per point that will be used to generate a soft shadow for any light source. Use this to shorten render times at the price of soft shadow quality. This will lower the maximum number of rays sent for any light sources that have a shadowSamples setting higher than this value, but will not raise the number if shadowSamples is set to a lower value. |
| **maxRayDepth** | The maximum amount of bounces a ray can have before being considered done. A bounce can be a reflection or a refraction. Increase the value if a ray goes through many transparent triangles before hitting an opaque object and you get light in areas that should be in the shadow. Common failure case: trees with alpha-tested leaves placed in a shadow of a mountain. |
| **giTransparencyDepth** | Maximum transparency depth for global illumination rays, i.e. the number of transparent surfaces the ray can go through, before you can assume it has been absorbed. Lower values speed up rendering, in scenes with, e.g. dense foliage, but may cause overlapping transparent objects to cast too much shadow. The default is 2. |

### Global Illumination (<*GISettings*> element)

The Global Illumination system allows you to use two separate algorithms to calculate indirect lighting. You can for instance calculate multiple levels of light bounces with a fast algorithm like the Path Tracer, and still calculate the final bounce with Final Gather to get a fast high-quality global illumination render. Both subsystems have individual control of Intensity and Saturation to boost the effects if necessary.

It's recommended to use FinalGather as the primary integrator and either None or PathTracer as the secondary integrator. Unity uses the first option (so final gather only) as the default, since it produces the best quality renders in most cases. Path Tracer should be used if many indirect bounces are needed and Final Gather-only solution with acceptable quality would take to much time to render.

**GISettings**

| | |
|---|---|
| **enableGI** | Setting to true enables Global Illumination. |
| **primaryIntegrator** | The integrator used for the final calculations of indirect light. FinalGather is default. |
| **secondaryIntegrator** | The integrator used for initial bounces of indirect light. Default is None, PathTracer is optional. |
| **primaryIntensity** | As a post process, converts the color of the primary integrator result from RGB to HSV and scales the V value. (in Unity: *Bounce Intensity*) |
| **primarySaturation** | As a post process, converts the color of the primary integrator result from RGB to HSV and scales the S value. |
| **secondaryIntensity** | As a post process, converts the color of the secondary integrator result from RGB to HSV and scales the V value. |
| **secondarySaturation** | As a post process, converts the color of the secondary integrator result from RGB to HSV and scales the S value. |

| | |
|---|---|
| **diffuseBoost** | This setting can be used to exaggerate light bouncing in dark scenes. Setting it to a value larger than 1 will push the diffuse color of materials towards 1 for GI computations. The typical use case is scenes authored with dark materials, this happens easily when doing only direct lighting since it is easy to compensate dark materials with strong light sources. Indirect light will be very subtle in these scenes since the bounced light will fade out quickly. Setting a diffuse boost will compensate for this. Note that values between 0 and 1 will decrease the diffuse setting in a similar way making light bounce less than the materials says, values below 0 is invalid. The actual computation taking place is a per component pow(colorComponent, (1.0 / diffuseBoost)). (in Unity: *Bounce Boost*) |
| **fgPreview** | Enable for a quick preview of the final image lighting. |

**Final Gather**

The settings below control the quality or correctness of the Final Gather solution. The normal usage scenario is this:

1. For each baking set up Contrast Threshold and Number of Rays may be adjusted. There are no perfect settings for these since they depend on the complexity of the geometry and light setup.
2. Check Visibility and Light Leakage reduction are expensive operations and should only be used to remedy actual light leakage problems. These settings will only help if the light leakage is caused by the Global Illumination calculations. A very common light leakage situation occurs with a wall as a single plane with no thickness. The light leaking through in that situation does not come from GI.
3. Gradient threshold should only be changed if there are white halos around corners.

Steps 2 and 3 should not need much tweaking in most scenes.

**GISettings**

| | |
|---|---|
| **fgContrastThreshold** | Controls how sensitive the final gather should be for contrast differences between the points during precalculation. If the contrast difference is above this threshold for neighbouring points, more points will be created in that area. This tells the algorithmto place points where they are really needed, e.g. at shadow boundaries or in areas where the indirect light changes quickly. Hence this threshold controls the number of points created in the scene adaptively. Note that if a low number of final gather rays are used, the points will have high variance and hence a high contrast difference. In that the case contrast threshold needs to be raised to prevent points from clumping together or using more rays per sample. (in Unity: *Contrast Threshold*) |
| **fgRays** | The maximum number of rays taken in each Final Gather sample. More rays gives better results but take longer to evaluate. (in Unity: *Final Gather Rays*) |
| **fgCheckVisibility** | Turn this on to reduce light leakage through walls. When points are collected to interpolate between, some of them can be located on the other side of geometry. As a result light will bleed through the geometry. To prevent this Beast can reject points that are not visible. |
| **fgCheckVisibilityDepth** | Controls for how many bounces the visibility checks should be performed. Adjust this only if experiencing light leakage when using multi bounce Final Gather. |
| **fgLightLeakReduction** | This setting can be used to reduce light leakage through walls when using final gather as primary GI and path tracing as secondary GI. Leakage, which can happen when e.g. the path tracer filters in values on the other side of a wall, is reduced by using final gather as a secondary GI fallback when sampling close to walls or corners. When this is enabled a final gather depth of 3 will be used automatically, but the higher depths will only be used close to walls or corners. Note that this is only usable when path tracing is used as secondary GI. |
| **fgLightLeakRadius** | Controls how far away from walls the final gather will be called again, instead of the secondary GI. If 0.0 is used Beast will try to estimate a good value. If this does not eliminate the leakage it can be set to a higher value manually. |
| **fgGradientThreshold** | Controls how the irradiance gradient is used in the interpolation. Each point stores its irradiance gradient which can be used to improve the interpolation. In some situations using the gradient can result in white "halos" and other artifacts. This threshold can be used to reduce those artifacts (set it low or to 0). (in Unity: *Interpolation*) |
| **fgInterpolationPoints** | Sets the number of final gather points to interpolate between. A higher value will give a smoother result, but can also smooth out details. If light leakage is introduced through walls when this value is increased, checking the sample visibility solves that problem. (in Unity: *Interpolation Points*) |
| **fgNormalThreshold** | Controls how sensitive the final gather should be for differences in the points normals. A lower value will give more points in areas of high curvature. |
| **fgDepth** | Controls the number of indirect light bounces. A higher value gives a more correct result, but the cost is increased rendering time. For cheaper multi bounce GI, use Path Tracer as the secondary integrator instead of increasing depth. (in Unity: *Bounces*) |

| | |
|---|---|
| **fgAttenuationStart** | The distance where attenuation is started. There is no attenuation before this distance. This can be used to add a falloff effect to the final gather lighting. When fgAttenuationStop is set higher than 0.0 this is enabled. |
| **fgAttenuationStop** | Sets the distance where attenuation is stopped (fades to zero). There is zero intensity beyond this distance. To enable attenuation set this value higher than 0.0. The default value is 0.0. |
| **fgFalloffExponent** | This can be used to adjust the rate by which lighting falls off by distance. A higher exponent gives a faster falloff. |
| **fgAOInfluence** | Blend the Final Gather with Ambient Occlusion. Range between 0..1. 0 means no occlusion, 1 is full occlusion. If Final Gather is used with multiple depths or with Path Tracing as Secondary GI the result can become a bit "flat". A great way to get more contrast into the lighting is to factor in a bit of ambient occlusion into the calculation. This Ambient Occlusion algorithm affects only final gather calculations. The Ambient Occlusion exposed in the Lightmapping window is calculated differently - by a separate, geometry-only pass. |
| **fgAOMaxDistance** | Max distance for the occlusion rays. Beyond this distance a ray is considered to be unoccluded. Can be used to avoid full occlusion for closed scenes such as rooms or to limit the AO contribution to creases. |
| **fgAOContrast** | Can be used to adjust the contrast for ambient occlusion. |
| **fgAOScale** | A scaling of the occlusion values. Can be used to increase or decrease the shadowing effect. |

**Path Tracer**

Use path tracing to get fast multi bounce global illumination. It should not be used as primary integrator for baking since the results are quite noisy which does not look good in light maps. It can be used as primary integrator to adjust the settings, to make sure the cache spacing and accuracy is good. The intended usage is to have it set as secondary integrator and have single bounce final gather as primary integrator. Accuracy and Point Size can be adjusted to make sure that the cache is sufficiently fine grained.

**GISettings**

| | |
|---|---|
| **ptAccuracy** | Sets the number of paths that are traced for each sample element (pixel, texel or vertex). For preview renderings, a low value like 0.5 to 0.1 can be used. This means that 1/2 to 1/10 of the pixels will generate a path. For production renderings values above 1.0 may be used, if necessary to get good quality. |
| **ptPointSize** | Sets the maximum distance between the points in the path tracer cache. If set to 0 a value will be calculated automatically based on the size of the scene. The automatic value will be printed out during rendering, which is a good starting value if the point size needs to be adjusted. |
| **ptCacheDirectLight** | When this is enabled the path tracer will also cache direct lighting from light sources. This increases performance since fewer direct light calculations are needed. It gives an approximate result, and hence can affect the quality of the lighting. For instance indirect light bounces from specular highlights might be lost. |
| **ptCheckVisibility** | Turn this on to reduce light leakage through walls. When points are collected to interpolate between, some of them can be located on the other side of geometry. As a result light will bleed through the geometry. To prevent this Beast can reject points that are not visible. Note: If using this turn off light leakage reduction for Final Gather. |

### Frame Settings (<*FrameSettings*> element)

Allow to control the amount of threads Beast uses and also the gamma correction of the input and output.

**FrameSettings**

| | |
|---|---|
| **inputGamma** | Keep at 1, as this setting is set appropriately per texture. |

### Surface Transfer (<*SurfaceTransferSettings*> element)

SurfaceTransferSettings are used to allow for transferring the lighting from LOD0 (the level of detail that is shown when the camera is close to an object) to LOD's with lower fidelity. Keep the settings at their defaults.

Page last updated: 2012-10-31

---

## LightmappingUV

Unity will use UV2 for lightmaps, if the channel is present. Otherwise it will use primary UVs.

Unity can unwrap your mesh for you to generate lightmap UVs. Just use the Generate Lightmap UVs setting in **Mesh Import Settings**.

Advanced Options for Generate Lightmap UVs:

| | |
|---|---|
| **Pack Margin** | The margin between neighboring patches, assuming the mesh will take entire 1024x1024 lightmap measured in pixels. That has great effect: to allow filtering, Lightmap will contain lighting information in texels near patch border. So to avoid light bleeding when applying Lightmap there should be some margin between patches. |
| **Hard Angle** | The angle between neighboring triangles, after which the edge between them will be considered hard edge and seam will be created. If you set it to 180 degrees all edges will be considered smooth: this is useful for organic models. The default value 88 degrees: this is useful for mechanical models |
| **Angle Error** | Maximum possible deviation of UVs angles from source geometry angles, in percentage. Basically it controls how similar triangles in uv space will be to triangles in original geometry (the value, the more similar triangles will be). Usually you wants it pretty low to avoid artifacts when applying Lightmap. Default is 8 percent. (This value goes from 0 to 100) |
| **Area Error** | Maximum possible deviation of UVs areas from source geometry areas, in percentage. Basically it controls how good relative triangle areas are preserved. Usually that is not very critical, and moving that up can allow to create less patches; although you should recheck that distortion do not deteriorate Lightmap quality, as that way triangles may have different resolution. Default is 15 percent. (This value goes from 0 to 100) |

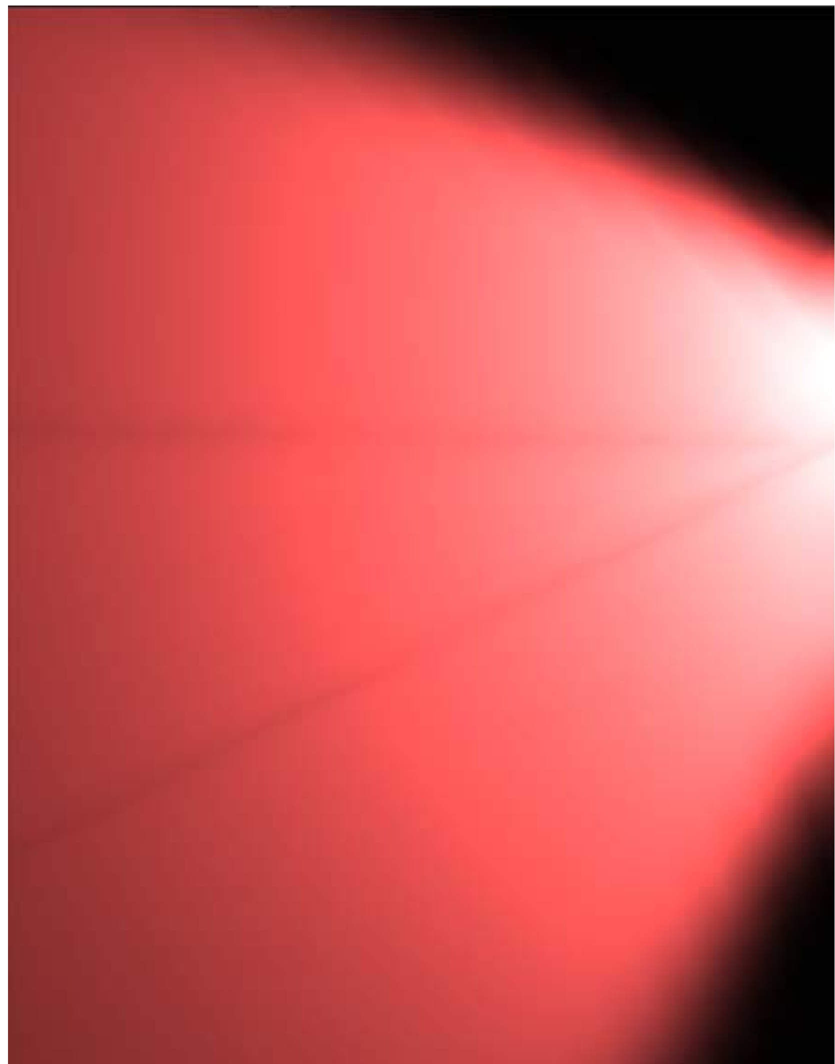
If you prefer to provide your own UVs for lightmapping, remember that a good UV set for lightmapping:

- Is contained within the [0,1]x[0,1] space
- Has no overlapping faces.
- Has low angle distortion, that is deviation of angles in UVs and in source geometry.
- Has low area distortion, that is, relative scale of triangles is mostly preserved, unless you really want some areas to have bigger Lightmap Resolution.
- Has enough margin between individual patches.

Some examples of the hints suggested above:

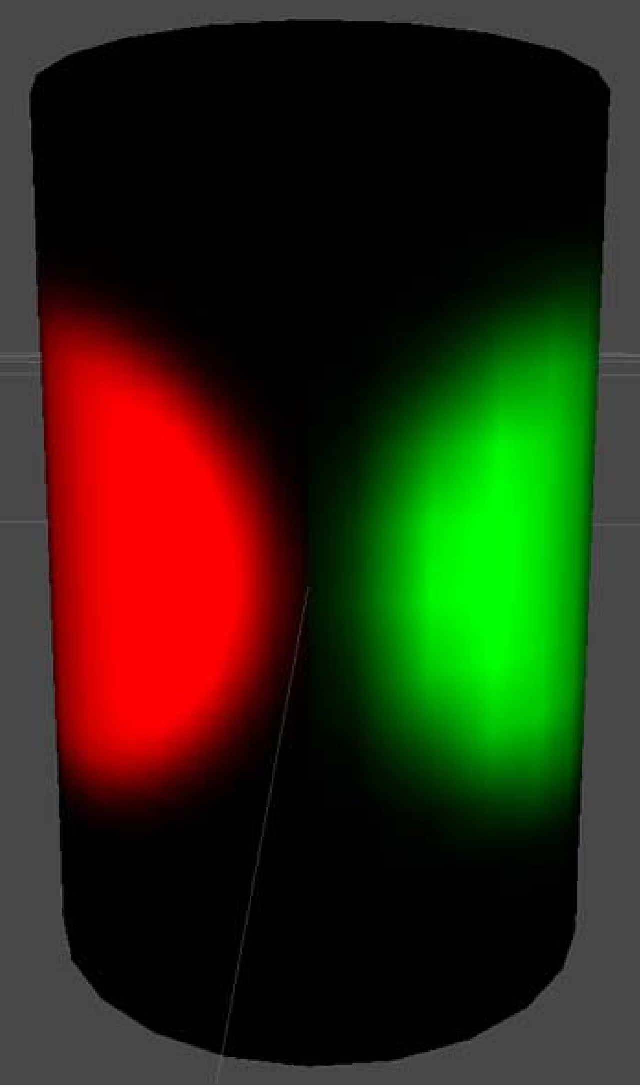
### Angle distortion

These screenshots were made for equal resolution, but with different uvs. Look at artefacts, and how the shape of light was slightly changed. There are only 4 triangles, actually, so shape distortion can be far uglier.



### Area distortion

There are 2 spotlight with same parameters, the difference being only pointing to areas with different lightmap resolution, due to relative triangle scale being not preserved



Page last updated: 2010-07-01

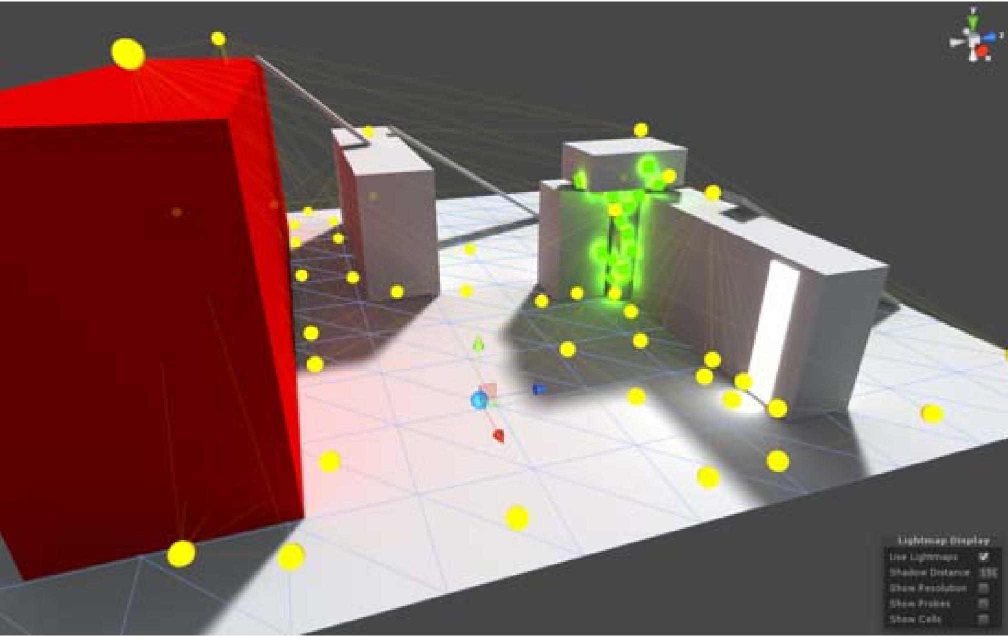
---

# LightProbes

Although lightmapping adds greatly to the realism of a scene, it has the disadvantage that non-static objects in the scene are less realistically rendered and can look disconnected as a result. It isn't possible to calculate lightmapping for moving objects in real time but it is possible to get a similar effect using **light probes**. The idea is that the lighting is sampled at strategic points in the scene, denoted by the positions of the probes. The lighting at any position can then be approximated by interpolating between the samples taken by the nearest probes. The interpolation is fast enough to be used during gameplay and helps avoid the disconnection between the lighting of moving objects and static lightmapped objects in the scene.

### Adding Light probes

The Light Probe Group component (menu: **Component -> Rendering -> Light Probe Group**) can be added to any available object in the scene. The inspector can be used to add new probes to the group. The probes appear in the scene as yellow spheres which can be positioned in the same manner as GameObjects. Selected probes can also be duplicated with the usual keyboard shortcut (ctrl+d/cmd+d).



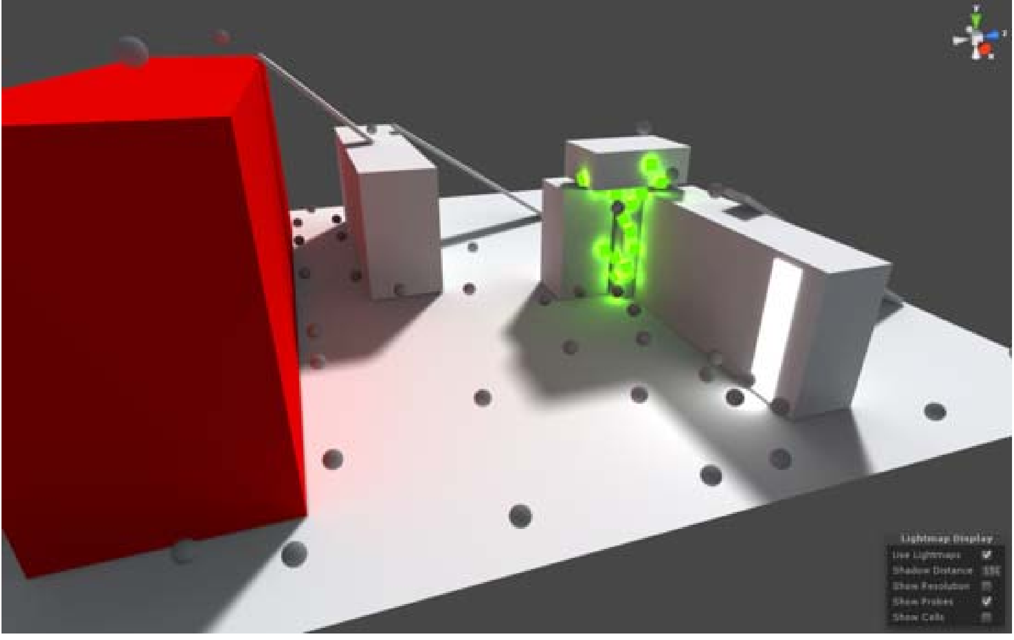
### Choosing Light Probe positions

Remember to **place probes where you want to sample light or sample darkness**. The probes need to **form a volume** within the scene for the space subdivision to work properly.

The simplest approach to positioning is to arrange them in a regular 3D grid pattern. While this setup is simple and effective, it is likely to consume a lot of memory (each light probe is essentially a spherical, panoramic HDR image of the view from the sample point). It is worth noting that probes are only needed for regions that players, NPCs or other dynamic objects can actually move to. Also, since lighting conditions are interpolated for positions between probes, it is not necessary to use lots of them across areas where the light doesn't change very much. For example, a large area of uniform shadow would not need a large number of probes and neither would a brightly lit area far away from reflective objects. Probes are generally needed where the lighting conditions change abruptly, for instance at the edge of a shadow area or in places where pieces of scenery have different colors.

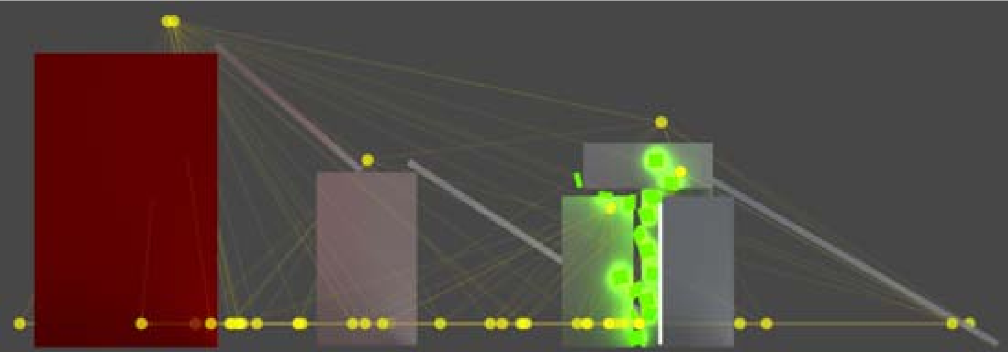
In some cases, the infrastructure of the game can be useful in choosing light probe positions. For example, a racing game typically uses waypoints around the track for AI and other purposes. These are likely to be good candidates for probe positions and it would likely be straightforward to set these positions from an editor script. Similarly, navigation meshes typically define the areas that can be reached by players and these also lend themselves to automated positioning of probes.

Here light probes have been baked over surfaces where our characters can walk on, but only where there are interesting lighting changes to capture:

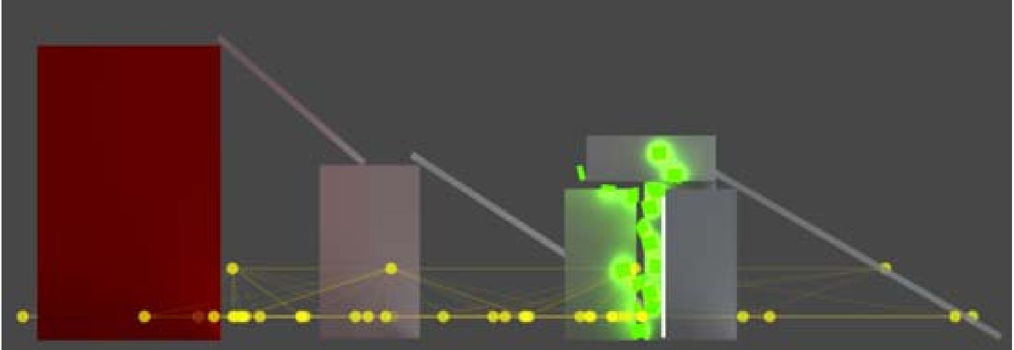



### Flat 2D levels

As it is now, the light probe system can't bake a completely flat probe cloud. So even if all your characters move only on a plane, you still have to take care to position at least some probes in a higher layer, so that a volume is formed and interpolation can work properly.



***Good***: *This is the original probe placement. The characters can move up the ramps and up onto the boxes, so it's good to sample lighting up there as well.*

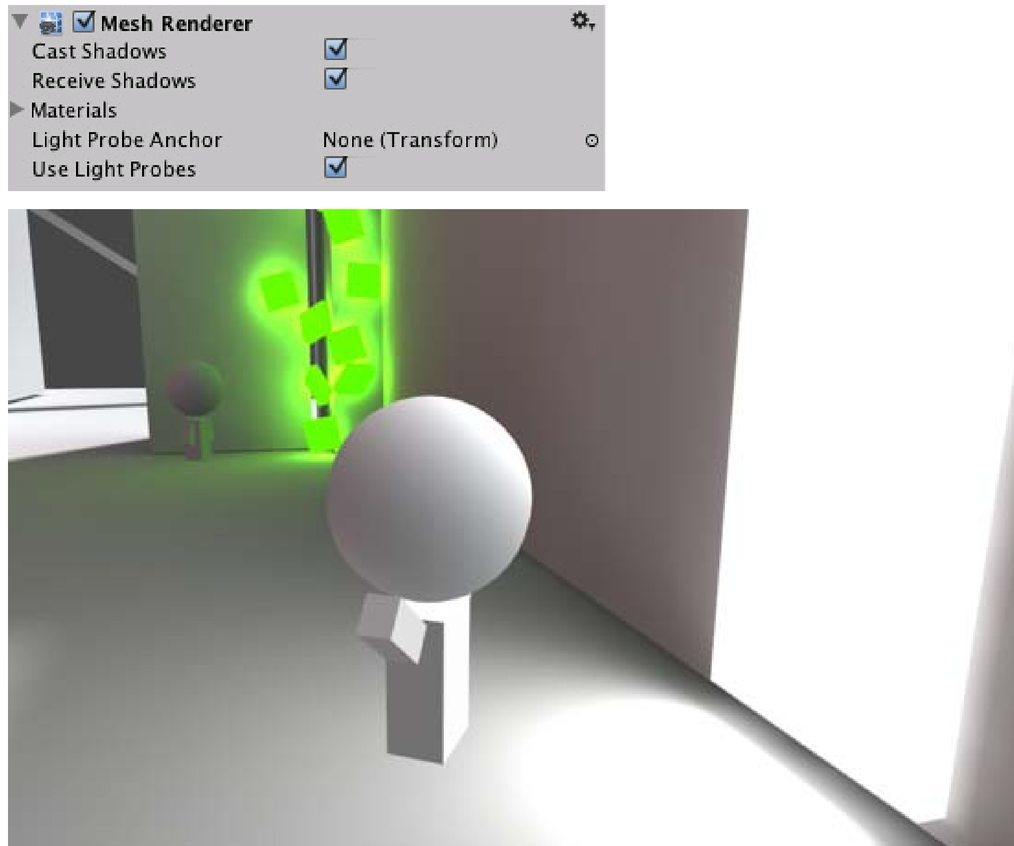


***Good***: *Here we assume the characters can only move on the plane. Still, there's a couple of probes placed a little bit higher, so that a volume is formed and thin cell are avoided.*

***Bad***: *The probes are placed too flat, which creates really long and thin cells and produces unintuitive interpolation results.*

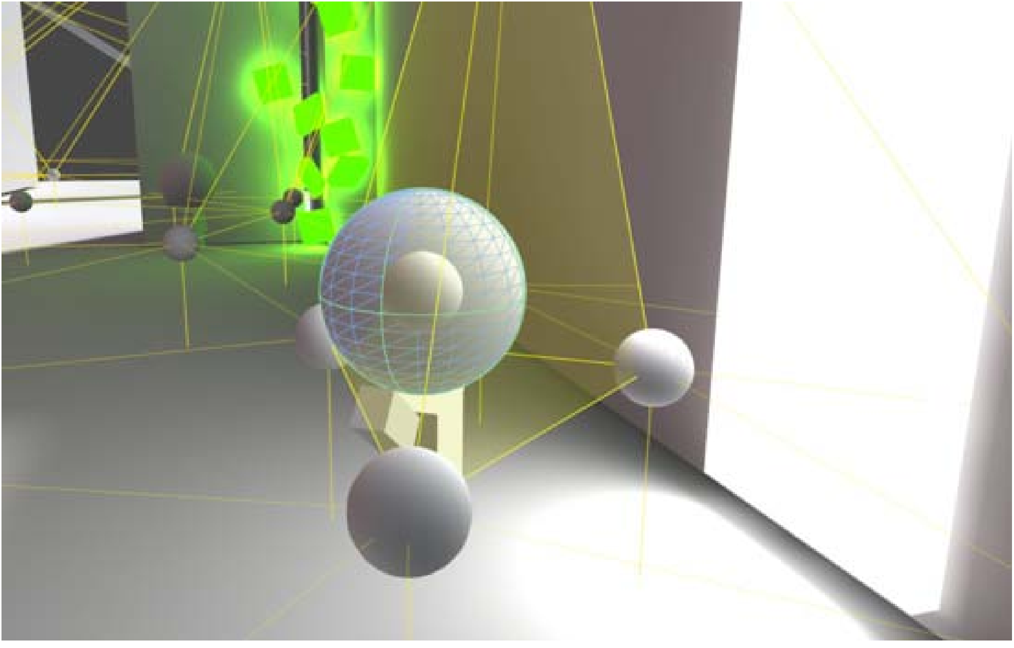
### Using Light Probes

To allow a mesh to receive lighting from the probe system, you should enable the Use Light Probes option on its Mesh Renderer:




The probe interpolation requires a point in space to represent the position of the mesh that is receiving light. By default, the centre of the mesh's bounding box is used but it is possible to override this by dragging a Transform to the Mesh Renderer's Light Probe Anchor property (this Transform's position will be used as the interpolation point instead). This may be useful when an object contains two separate adjoining meshes; if both meshes are lit individually according to their bounding box positions then the lighting will be discontinuous at the place where they join. This can be prevented by using the same Transform (for example the parent or a child object) as the interpolation point for both Mesh Renderers.

When an object using light probes is the active selected object in the Light Probes Scene View mode, its interpolated probe will be rendered on top of it for preview. The interpolated probe is the one used for rendering the object and is connected with 4 thin blue lines (3 when outside of the probe volume) to the probes it is being interpolated between:



**Dual Lightmaps vs. Single Lightmaps mode**

In Single Lightmaps mode all static lighting (including lights set to 'Auto' lightmapping mode) is baked into the light probes.

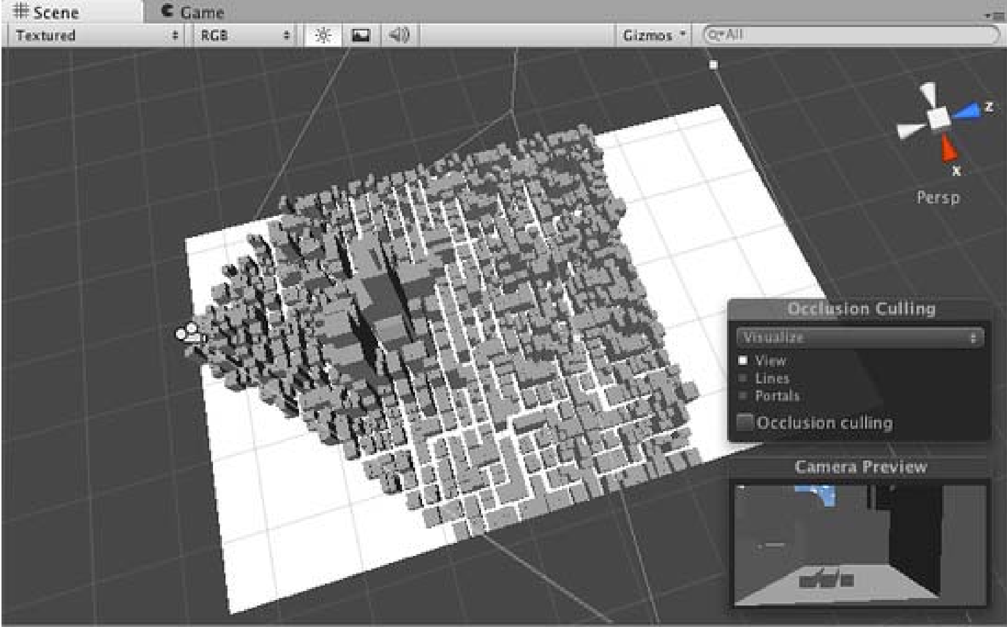
In Dual Lightmaps mode light probes will store lighting in the same configuration as 'Near' lightmaps, i.e. full illumination from sky lights, emissive materials, area lights and 'Baked Only' lights, but only indirect illumination from 'Auto' lights. Thanks to that the object can be lit in real-time with the 'Auto' lights and take advantage of dynamic elements such as real-time shadows, but at the same time receive indirect lighting added to the scene by these lights.

Page last updated: 2012-10-16

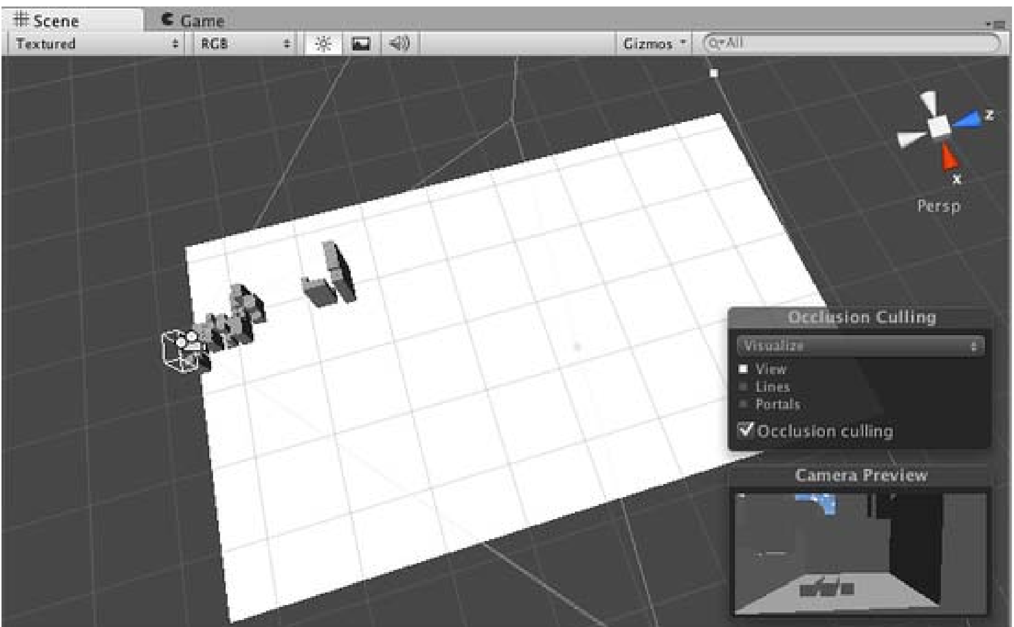
---

# Occlusion Culling

Occlusion Culling is a feature that disables rendering of objects when they are not currently seen by the camera because they are obscured by other objects. This does not happen automatically in 3D computer graphics since most of the time objects farthest away from the camera are drawn first and closer objects are drawn over the top of them (this is called "overdraw"). Occlusion Culling is different from Frustum Culling. Frustum Culling only disables the renderers for objects that are outside the camera's viewing area but does not disable anything hidden from view by overdraw. Note that when you use Occlusion Culling you will still benefit from Frustum Culling.




*The scene rendered without Occlusion Culling*




*The same scene rendered with Occlusion Culling*

The occlusion culling process will go through the scene using a virtual camera to build a hierarchy of potentially visible sets of objects. This data is used at runtime by each camera to identify what is visible and what is not. Equipped with this information, Unity will ensure only visible objects get sent to be rendered. This reduces the number of draw calls and increases the performance of the game.

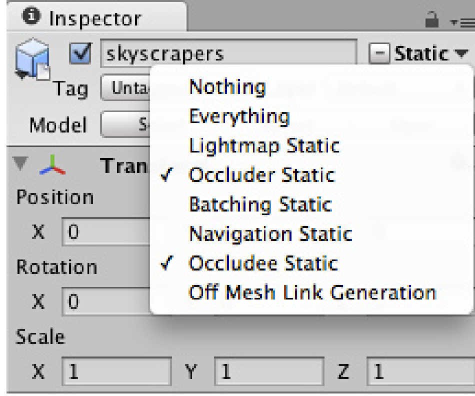
The data for occlusion culling is composed of cells. Each cell is a subdivision of the entire bounding volume of the scene. More specifically the cells form a binary tree. Occlusion Culling uses two trees, one for View Cells (Static Objects) and the other for Target Cells (Moving Objects). View Cells map to a list of indices that define the visible static objects which gives more accurate culling results for static objects.

It is important to keep this in mind when creating your objects because you need a good balance between the size of your objects and the size of the cells. Ideally, you shouldn't have cells that are too small in comparison with your objects but equally you shouldn't have objects that cover many cells. You can sometimes improve the culling by breaking large objects into smaller pieces. However, you can still merge small objects together to reduce draw calls and, as long as they all belong to the same cell, occlusion culling will not be affected. The collection of cells and the visibility information that determines which cells are visible from any other cell is known as a PVS (**P**otentially **V**isible **S**et).

## Setting up Occlusion Culling

In order to use Occlusion Culling, there is some manual setup involved. First, your level geometry must be broken into sensibly sized pieces. It is also helpful to lay out your levels into small, well defined areas that are occluded from each other by large objects such as walls, buildings, etc. The idea here is that each individual mesh will be turned on or off based on the occlusion data. So if you have one object that contains all the furniture in your room then either all or none of the entire set of furniture will be culled. This doesn't make nearly as much sense as making each piece of furniture its own mesh, so each can individually be culled based on the camera's view point.

You need to tag all scene objects that you want to be part of the occlusion to **Occlusion Static** in the **Inspector**. The fastest way to do this is to multi-select the objects you want to be included in occlusion calculations, and mark them as **Occlusion Static** and **Occludee Static**.




*Marking an object for Occlusion*

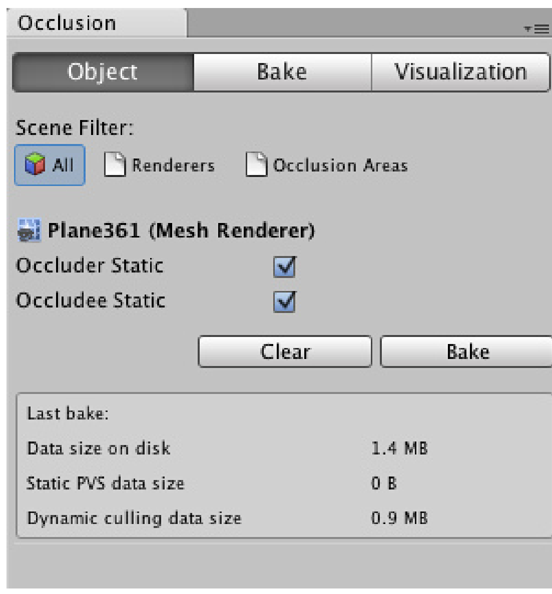
When should I use **Occludee Static**? Transparent objects that do not occlude, as well as small objects that are unlikely to occlude other things, should be marked as **Occludees**, but not **Occluders**. This means they will be considered in occlusion by other objects, but will not be considered as occluders themselves, which will help reduce computation.

## Occlusion Culling Window

For most operations dealing with Occlusion Culling, we recommend you use the Occlusion Culling Window (**Window->Occlusion Culling**)

In the Occlusion Culling Window, you can work with occluder meshes, and Occlusion Areas.

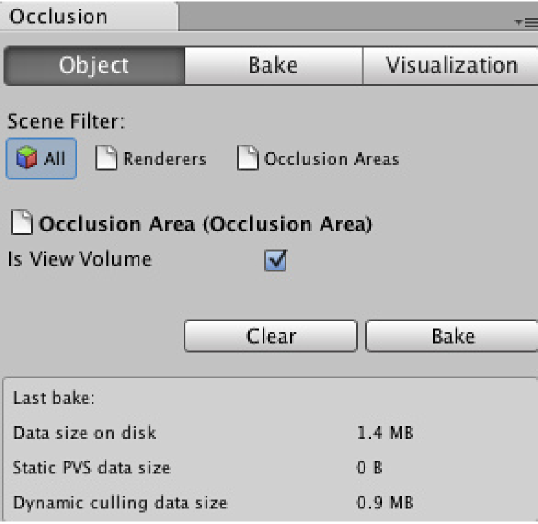
If you are in the **Object** tab of the **Occlusion Culling Window** and have some Mesh Renderer selected in the scene, you can modify the relevant Static flags:




*Occlusion Culling Window for a Mesh Renderer*

If you are in the **Object** tab of the **Occlusion Culling Window** and have an Occlusion Area selected, you can work with

relevant OcclusionArea properties (for more details go to the Occlusion Area section)



*Occlusion Culling Window for the Occlusion Area*

**NOTE:** By default if you don't create any occlusion areas, occlusion culling will be applied to the whole scene.

**NOTE:** Whenever your camera is outside occlusion areas, occlusion culling will not be applied. It is important to set up your Occlusion Areas to cover the places where the camera can potentially be, but making the areas too large, incurs a cost during baking.

### Occlusion Culling - Bake

Occl. Culling
Object | Bake | Visualization
Technique: Automatic Portal Generation
Portals are generated automatically. Static and dynamic objects are culled through portals. This allows you to open and close portals at runtime. This technique will cull objects most accurately but also has the most performance overhead on the CPU.
View Cell Size: 1
Near Clip Plane: 0.3
Far Clip Plane: 1000
Memory Limit: 100000
Clear | Bake
No occlusion data has been baked.

*Occlusion culling inspector bake tab.*

### Properties

| | |
|---|---|
| **Technique** | Select between the types of occlusion culling baking |
| **PVS only** | Only static objects will be occlusion culled. Dynamic objects will be culled based on the view frustrum only. this technique has the smallest overhead on the CPU, but since dynamic objects are not culled, it is only recommended for games with few moving objects and characters. Since all visibility is precomputed, you cannot open or close portals at runtime. |
| **PVS and dynamic objects** | Static objects are culled using precomputed visibility. Dynamic objects are culled using portal culling. this technique is a good balance between runtime overhead and culling efficiency. Since all visibility is precomputed, you cannot open or close a portal at runtime |
| **Automatic Portal Generation** | Portals are generated automatically. Static and dynamic objects are culled through portals. This allows you to open and close portals at runtime. This technique will cull objects most accurately, but also has the most performance overhead on the CPU. |
| **View Cell Size** | Size of each view area cell. A smaller value produces more accurate occlusion culling. The value is a tradeoff between occlusion accuracy and storage size |

| | |
|---|---|
| **Near Clip Plane** | Near clip plane should be set to the smallest near clip plane that will be used in the game of all the cameras. |
| **Far Clip Plane** | Far Clip Plane used to cull the objects. Any object whose distance is greater than this value will be occluded automatically.(Should be set to the largest far clip planed that will be used in the game of all the cameras) |
| **Memory limit** | This is a hint for the PVS-based baking, not available in *Automatic Portal Generation* mode |

When you have finished tweaking these values you can click on the **Bake** Button to start processing the Occlusion Culling data. If you are not satisfied with the results, you can click on the **Clear** button to remove previously calculated data.

### Occlusion Culling - Visualization

*Occlusion culling inspector visualization tab.*

The near and far planes define a virtual camera that is used to calculate the occlusion data. If you have several cameras with different near or far planes, you should use the smallest near plane and the largest far plane distance of all cameras for correct inclusion of objects.

All the objects in the scene affect the size of the bounding volume so try to keep them all within the visible bounds of the scene.

When you're ready to generate the occlusion data, click the **Bake** button. Remember to choose the **Memory Limit** in the **Bake** tab. Lower values make the generation quicker and less precise, higher values are to be used for production quality closer to release.

Bear in mind that the time taken to build the occlusion data will depend on the cell levels, the data size and the quality you have chosen. Unity will show the status of the PVS generation at the bottom of the main window.

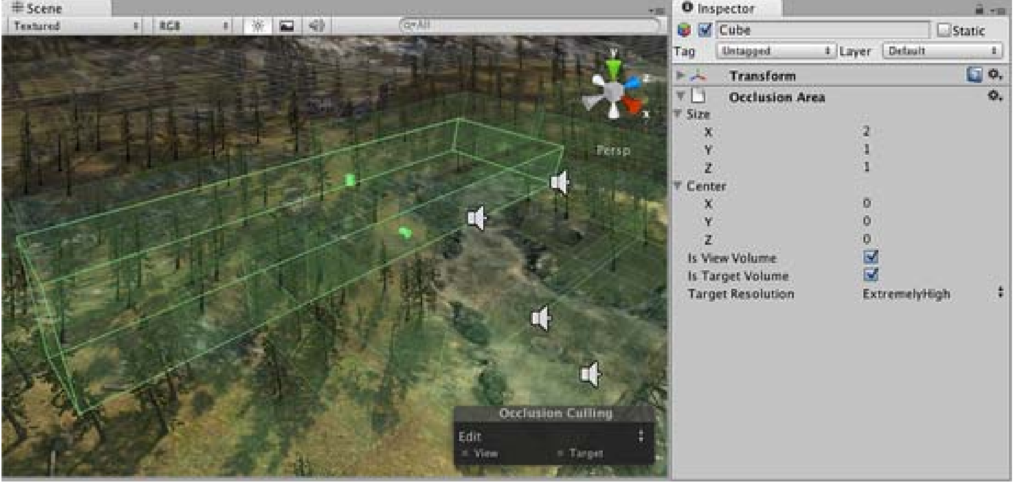
After the processing is done, you should see some colorful cubes in the View Area. The colored areas are regions that share the same occlusion data.

Click on **Clear** if you want to remove all the pre-calculated data for Occlusion Culling.

## Occlusion Area (Pro Only)

To apply occlusion culling to moving objects you have to create an **Occlusion Area** and then modify its size to fit the space where the moving objects will be located (of course the moving objects cannot be marked as static). You can create Occlusion Areas is by adding the **Occlusion Area** component to an empty game object (**Component->Rendering->Occlusion Area** in the menus)

After creating the **Occlusion Area**, just check the *Is Target Volume* checkbox to occlude moving objects.



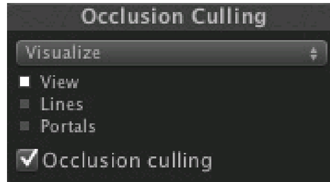
*Occlusion Area properties for moving objects.*

| | |
|---|---|
| **Size** | Defines the size of the Occlusion Area. |
| **Center** | Sets the center of the Occlusion Area. By default this is 0,0,0 and is located in the center of the box. |
| **Is View Volume** | Defines where the camera can be. Check this in order to occlude static objects that are inside this *Occlusion Area*. |
| **Is Target Volume** | **Select this when you want to occlude moving objects.** |
| **Target Resolution** | Determines how accurate the occlusion culling inside the area will be. This affects the size of the cells in an Occlusion Area. **NOTE: This only affects Target Areas.** |
| **Low** | This takes less time to calculate but is less accurate. |
| **Medium** | This gives a balance between accuracy and time taken to process the occlusion culling data. |
| **High** | This takes longer to calculate but has better accuracy. |
| **Very High** | Use this value when you want to have more accuracy than high resolutions, be aware it takes more time. |
| **Extremely High** | Use this value when you want to have almost exact occlusion culling on your moveable objects. Note: This setting takes a lot of time to calculate. |

After you have added the Occlusion Area, you need to see how it divides the box into cells. To see how the occlusion area will be calculated, Select **Edit** and toggle the **View** button in the **Occlusion Culling Preview Panel**.

### Testing the generated occlusion

After your occlusion is set up, you can test it by enabling the *Occlusion Culling* (in the **Occlusion Culling Preview Panel** in Visualize mode) and moving the **Main Camera** around in the scene view.



*The Occlusion View mode in Scene View*

As you move the Main Camera around (whether or not you are in Play mode), you'll see various objects disable themselves. The thing you are looking for here is any error in the occlusion data. You'll recognize an error if you see objects suddenly popping into view as you move around. If this happens, your options for fixing the error are either to change the resolution (if you are playing with target volumes) or to move objects around to cover up the error. To debug problems with occlusion, you can move the Main Camera to the problematic position for spot-checking.

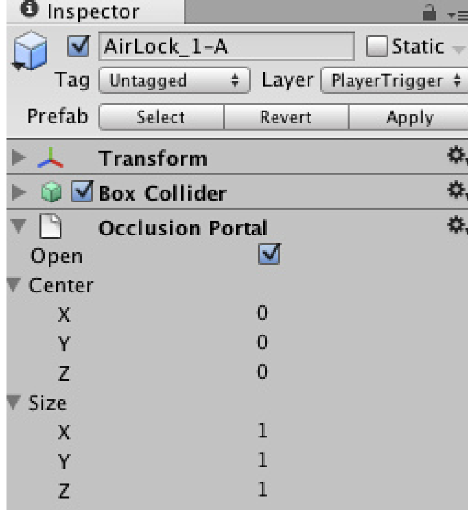
When the processing is done, you should see some colorful cubes in the View Area. The blue cubes represent the cell divisions for **Target Volumes**. The white cubes represent cell divisions for **View Volumes**. If the parameters were set correctly you should see some objects not being rendered. This will be because they are either outside of the view frustum of

the camera or else occluded from view by other objects.

After occlusion is completed, if you don't see anything being occluded in your scene then try breaking your objects into smaller pieces so they can be completely contained inside the cells.

## Occlusion Portals

In order to create occlusion primitive which are openable and closable at runtime, Unity uses **Occlusion Portals**.




| | |
|---|---|
| **Open** | Indicates if the portal is open (scriptable) |
| **Center** | Sets the center of the Occlusion Area. By default this is 0,0,0 and is located in the center of the box. |
| **Size** | Defines the size of the Occlusion Area. |

Page last updated: 2012-02-14

---

## CameraTricks

It is useful to understand how the camera works when designing certain visual effects or interactions with objects in the scene. This section explains the nature of the camera's view and how it can be used to enhance gameplay.

- UnderstandingFrustum
- The Size of the Frustum at a Given Distance from the Camera
- Dolly Zoom (AKA the "Trombone" Effect)
- Rays from the Camera
- Using an Oblique Frustum
- Creating an Impression of Large or Small Size

Page last updated: 2011-09-06

---

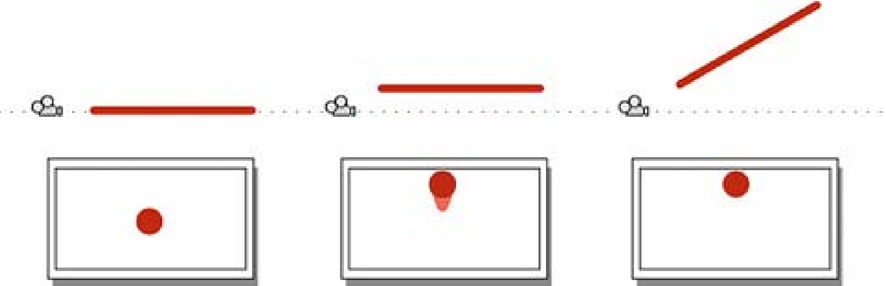
## UnderstandingFrustum

Understanding the View Frustum

The word **frustum** refers to a solid shape that looks like a pyramid with the top cut off parallel to the base. This is the shape of the region that can be seen and rendered by a perspective camera. The following thought experiment should help to explain why this is the case.

Imagine holding a straight rod (a broom handle or a pencil, say) end-on to a camera and then taking a picture. If the rod were held in the centre of the picture, perpendicular to the camera lens, then only its end would be visible as a circle on the picture; all other parts of it would be obscured. If you moved it upward, the lower side would start to become visible but you could hide

it again by angling the rod upward. If you continued moving the rod up and angling it further upward, the circular end would eventually reach the top edge of the picture. At this point, any object above the line traced by the rod in world space would not be visible on the picture.

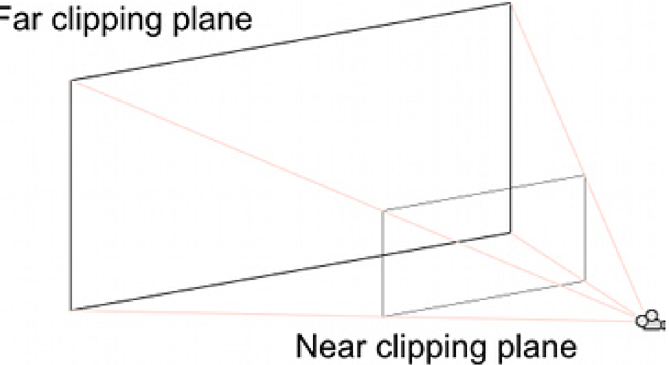


The rod could just as easily be moved and rotated left, right, or down or any combination of horizontal and vertical. The angle of the "hidden" rod simply depends on its distance from the centre of the screen in both axes.

The meaning of this thought experiment is that any point in a camera's image actually corresponds to a line in world space and only a single point along that line is visible in the image. Everything behind that position on the line is obscured.

The outer edges of the image are defined by the diverging lines that correspond to the corners of the image. If those lines were traced backwards towards the camera, they would all eventually converge at a single point. In Unity, this point is located exactly at the camera's transform position and is known as the centre of perspective. The angle subtended by the lines converging from the top and bottom centres of the screen at the centre of perspective is called the field of view (often abbreviated to FOV).

As stated above, anything that falls outside the diverging lines at the edges of the image will not be visible to the camera, but there are also two other restrictions on what it will render. The near and far clipping planes are parallel to the camera's XY plane and each set at a certain distance along its centre line. Anything closer to the camera than the near clipping plane and anything farther away than the far clipping plane will not be rendered.




The diverging corner lines of the image along with the two clipping planes define a truncated pyramid - the view frustum.

Page last updated: 2011-09-06

---

## FrustumSizeAtDistance

A cross-section of the view frustum at a certain distance from the camera defines a rectangle in world space that frames the visible area. It is sometimes useful to calculate the size of this rectangle at a given distance, or find the distance where the rectangle is a given size. For example, if a moving camera needs to keep an object (such as the player) completely in shot at all times then it must not get so close that part of that object is cut off.

The height of the frustum at a given distance (both in world units) can be obtained with the following formula:-

```
var frustumHeight = 2.0 * distance * Mathf.Tan(camera.fieldOfView * 0.5 * Mathf.Deg2Rad);
```

...and the process can be reversed to calculate the distance required to give a specified frustum height:-

```
var distance = frustumHeight * 0.5 / Mathf.Tan(camera.fieldOfView * 0.5 * Mathf.Deg2Rad);
```

It is also possible to calculate the FOV angle when the height and distance are known:-

```
var camera.fieldOfView = 2 * Mathf.Atan(frustumHeight * 0.5 / distance) * Mathf.Rad2Deg;
```

Each of these calculations involves the height of the frustum but this can be obtained from the width (and vice versa) very easily:-

```
var frustumWidth = frustumHeight * camera.aspect;
var frustumHeight = frustumWidth / camera.aspect;
```

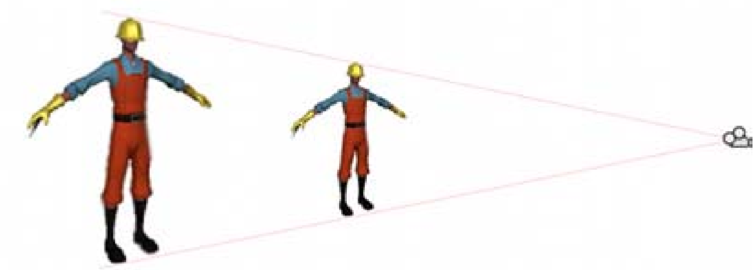
Page last updated: 2011-09-06

---

## DollyZoom

Dolly Zoom is the well-known visual effect where the camera simultaneously moves towards a target object and zooms out from it. The result is that the object appears roughly the same size but all the other objects in the scene change perspective. Done subtly, dolly zoom has the effect of highlighting the target object, since it is the only thing in the scene that isn't shifting position in the image. Alternatively, the zoom can be deliberately performed quickly to create the impression of disorientation.

An object that just fits within the frustum vertically will occupy the whole height of the view as seen on the screen. This is true whatever the object's distance from the camera and whatever the field of view. For example, you can move the camera closer to the object but then widen the field of view so that the object still just fits inside the frustum's height. That particular object will appear the same size onscreen but everything else will change size as the distance and FOV change. This is the essence of the dolly zoom effect.



Creating the effect in code is a matter of saving the height of the frustum at the object's position at the start of the zoom. Then, as the camera moves, its new distance is found and the FOV adjusted to keep it the same height at the object's position. This can be accomplished with the following code:-

```
var target: Transform;

private var initHeightAtDist: float;
private var dzEnabled: boolean;


// Calculate the frustum height at a given distance from the camera.
function FrustumHeightAtDistance(distance: float) {
	return 2.0 * distance * Mathf.Tan(camera.fieldOfView * 0.5 * Mathf.Deg2Rad);
}


// Calculate the FOV needed to get a given frustum height at a given distance.
```

```
function FOVForHeightAndDistance(height: float, distance: float) {
	return 2 * Mathf.Atan(height * 0.5 / distance) * Mathf.Rad2Deg;
}


// Start the dolly zoom effect.
function StartDZ() {
	var distance = Vector3.Distance(transform.position, target.position);
	initHeightAtDist = FrustumHeightAtDistance(distance);
	dzEnabled = true;
}


// Turn dolly zoom off.
function StopDZ() {
	dzEnabled = false;
}


function Start() {
	StartDZ();
}


function Update () {
	if (dzEnabled) {
		// Measure the new distance and readjust the FOV accordingly.
		var currDistance = Vector3.Distance(transform.position, target.position);
		camera.fieldOfView = FOVForHeightAndDistance(initHeightAtDist, currDistance);
	}

	// Simple control to allow the camera to be moved in and out using the up/down arrows.
	transform.Translate(Input.GetAxis("Vertical") * Vector3.forward * Time.deltaTime * 5);
}
```

Page last updated: 2011-09-06

---

## CameraRays

In the section Understanding the View Frustum, it was explained that any point in the camera's view corresponds to a line in world space. It is sometimes useful to have a mathematical representation of that line and Unity can provide this in the form of a Ray object. The Ray always corresponds to a point in the view, so the Camera class provides the ScreenPointToRay and ViewportPointToRay functions. The difference between the two is that ScreenPointToRay expects the point to be provided as a pixel coordinate, while ViewportPointToRay takes normalized coordinates in the range 0..1 (where 0 represents the bottom or left and 1 represents the top or right of the view). Each of these functions returns a Ray which consists of a point of origin and a vector which shows the direction of the line from that origin. The Ray originates from the near clipping plane rather than the Camera's transform.position point.

### Raycasting

The most common use of a Ray from the camera is to perform a raycast out into the scene. A raycast sends an imaginary "laser beam" along the ray from its origin until it hits a collider in the scene. Information is then returned about the object and the point that was hit in a RaycastHit object. This is a very useful way to locate an object based on its onscreen image. For example, the object at the mouse position can be determined with the following code:-

```
var hit: RaycastHit;
var ray: Ray = camera.ScreenPointToRay(Input.mousePosition);
```

```
if (Physics.Raycast(ray, hit)) {
	var objectHit: Transform = hit.transform;

	// Do something with the object that was hit by the raycast.
}
```

### Moving the Camera Along a Ray

It is sometimes useful to get a ray corresponding to a screen position and then move the camera along that ray. For example, you may want to allow the user to select an object with the mouse and then zoom in on it while keeping it "pinned" to the same screen position under the mouse (this might be useful when the camera is looking at a tactical map, say). The code to do this is fairly straightforward:-

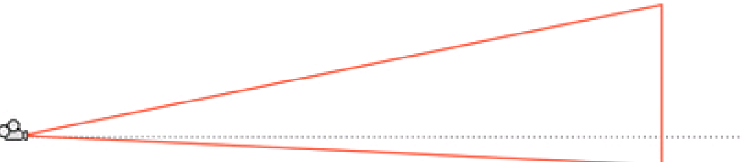
```
var zooming: boolean;
var zoomSpeed: float;

if (zooming) {
	var ray: Ray = camera.ScreenPointToRay(Input.mousePosition);
	zoomDistance = zoomSpeed * Input.GetAxis("Vertical") * Time.deltaTime;
	camera.transform.Translate(ray.direction * zoomDistance, Space.World);
}
```

Page last updated: 2011-09-06

---

## ObliqueFrustum

By default, the view frustum is arranged symmetrically around the camera's centre line but it doesn't necessarily need to be. The frustum can be made "oblique", which means that one side is at a smaller angle to the centre line than the opposite side. The effect is rather like taking a printed photograph and cutting one edge off. This makes the perspective on one side of the image seem more condensed giving the impression that the viewer is very close to the object visible at that edge. An example of how this can be used is a car racing game where the frustum might be flattened at its bottom edge. This would make the viewer seem closer to the road, accentuating the feeling of speed.



While the camera class doesn't have functions to set the obliqueness of the frustum, it can be done quite easily by altering the projection matrix:-

```
function SetObliqueness(horizObl: float, vertObl: float;) {
	var mat: Matrix4x4 = camera.projectionMatrix;
	mat[0, 2] = horizObl;
	mat[1, 2] = vertObl;
	camera.projectionMatrix = mat;
}
```

Mercifully, it is not necessary to understand how the projection matrix works to make use of this. The horizObl and vertObl values set the amount of horizontal and vertical obliqueness, respectively. A value of zero indicates no obliqueness. A positive value shifts the frustum rightwards or upwards, thereby flattening the left or bottom side. A negative value shifts leftwards or downwards and consequently flattens the right or top side of the frustum. The effect can be seen directly if this script is added to a camera and the game is switched to the scene view while the game runs; the wireframe depiction of the camera's frustum will change as you vary the values of horizObl and vertObl in the inspector. A value of 1 or -1 in either variable indicates that one side of the frustum is completely flat against the centreline. It is possible although seldom necessary to use values outside this range.

Page last updated: 2011-09-06

## ImpressionOfSize

From the graphical point of view, the units of distance in Unity are arbitrary and don't correspond to real world measurements. Although this gives flexibility and convenience for design, it is not always easy to convey the intended size of the object. For example, a toy car looks different to a full size car even though it may be an accurate scale model of the real thing.

A major element in the impression of an object's size is the way the perspective changes over the object's length. For example, if a toy car is viewed from behind then the front of the car will only be a short distance farther away than the back. Since the distance is small, perspective will have relatively little effect and so the front will appear little different in size to the back. With a full size car, however, the front will be several metres farther away from the camera than the back and the effect of perspective will be much more noticeable.

For an object to appear small, the lines of perspective should diverge only very slightly over its depth. You can achieve this by using a narrower field of view than the default 60� and moving the camera farther away to compensate for the increased onscreen size. Conversely, if you want to make an object look big, use a wide FOV and move the camera in close. When these perspective alterations are used with other obvious techniques (like looking down at a "small" object from higher-than-normal vantage point) the result can be quite convincing.

Page last updated: 2011-09-06

## Loading Resources at Runtime

In some situations, it is useful to make an asset available to a project without loading it in as part of a scene. For example, there may be a character or other object that can appear in any scene of the game but which will only be used infrequently (this might be a "secret" feature, an error message or a highscore alert, say). Furthermore, you may even want to load assets from a separate file or URL to reduce initial download time or allow for interchangeable game content.

Unity supports **Resource Folders** in the project to allow content to be supplied in the main game file yet not be loaded until requested. In Unity Pro, Unity iOS Advanced and Unity Android Advanced, you can also create **Asset Bundles**. These are files completely separate from the main game file which contain assets to be accessed by the game on demand from a file or URL.

### Asset Bundles (Unity Pro-only/iOS Advanced/Android Advanced licenses only)

An Asset Bundle is an external collection of assets. You can have many Asset Bundles and therefore many different external collections of assets. These files exist outside of the built Unity player, usually sitting on a web server for end-users to access dynamically.

To build an Asset Bundle, you call BuildPipeline.BuildAssetBundle() from inside an Editor script. In the arguments, you specify an array of **Objects** to be included in the built file, along with some other options. This will build a file that you can later load dynamically in the runtime by using AssetBundle.Load().

### Resource Folders

Resource Folders are collections of assets that are included in the built Unity player, but are not necessarily linked to any GameObject in the Inspector.

To put anything into a Resource Folder, you simply create a new folder inside the **Project View**, and name the folder "Resources". You can have multiple Resource Folders organized differently in your Project. Whenever you want to load an asset from one of these folders, you call Resources.Load().

If your target deployable is a **Streaming Web Player**, you can define which scene will include everything in your Resource Folders. You do this in the **Player Settings**, accessible via **Edit->Project Settings->Player**. Stream queue is determined by Build Settings' scene order.

**Note:**
All assets found in the Resources folders and their dependencies are stored in a file called *resources.assets*. If an asset is already used by another level it is stored in the *.sharedAssets* file for that level. The **Edit -> PlayerSettings First Streamed Level** setting determines the level at which the *resources.assets* will be collected and included in the build.

If a level prior to *"First streamed Level"* is including an asset in a Resource folder, the asset will be stored in assets for that level. if it is included afterwards, the level will reference the asset from the "resources.assets" file.

Only assets that are in the *Resources folder* can be accessed through Resources.Load. However many more assets might end up in the "resources.assets" file since they are dependencies. (For example a Material in the Resources folder might reference a Texture outside of the Resources folder)

### Resource Unloading

You can unload resources of an AssetBundle by calling AssetBundle.Unload(). If you pass **true** for the **unloadAllLoadedObjects** parameter, both the objects held internally by the AssetBundle and the ones loaded from the AssetBundle using AssetBundle.Load() will be destroyed and memory used by the bundle will be released.

Sometimes you may prefer to load an AssetBundle, instantiate the objects desired and release the memory used up by the bundle while keeping the objects around. The benefit is that you free up memory for other tasks, for instance loading another AssetBundle. In this scenario you would pass **false** as the parameter. After the bundle is destroyed you will not be able to load objects from it any more.

If you want to destroy scene objects loaded using Resources.Load() prior to loading another level, call Object.Destroy() on them. To release assets, use Resources.UnloadUnusedAssets().

Page last updated: 2012-11-28

---

## Modifying Source Assets Through Scripting

### Automatic Instantiation

Usually when you want to make a modification to any sort of game asset, you want it to happen at runtime and you want it to be temporary. For example, if your character picks up an invincibility power-up, you might want to change the **shader** of the **material** for the player character to visually demonstrate the invincible state. This action involves modifying the material that's being used. This modification is not permanent because we don't want the material to have a different shader when we exit **Play Mode**.

However, it is possible in Unity to write scripts that will permanently modify a source asset. Let's use the above material example as a starting point.

To temporarily change the material's shader, we change the **shader** property of the **material** component.

```
private var invincibleShader = Shader.Find ("Specular");

function StartInvincibility {
        renderer.material.shader = invincibleShader;
}
```

When using this script and exiting Play Mode, the state of the **material** will be reset to whatever it was before entering Play Mode initially. This happens because whenever renderer.material is accessed, the material is automatically instantiated and the instance is returned. This instance is simultaneously and automatically applied to the renderer. So you can make any changes that your heart desires without fear of permanence.

### Direct Modification

**IMPORTANT NOTE**
The method presented below will modify actual source asset files used within Unity. These modifications are not undoable. Use them with caution.

Now let's say that we don't want the material to reset when we exit play mode. For this, you can use renderer.sharedMaterial. The sharedMaterial property will return the actual asset used by this renderer (and maybe others).

The code below will permanently change the material to use the Specular shader. It will not reset the material to the state it was in before Play Mode.

```
private var invincibleShader = Shader.Find ("Specular");

function StartInvincibility {
        renderer.sharedMaterial.shader = invincibleShader;
}
```

As you can see, making any changes to a sharedMaterial can be both useful and risky. Any change made to a sharedMaterial will be permanent, and not undoable.

### Applicable Class Members

The same formula described above can be applied to more than just materials. The full list of assets that follow this convention is as follows:

- Materials: renderer.material and renderer.sharedMaterial
- Meshes: meshFilter.mesh and meshFilter.sharedMesh
- Physic Materials: collider.material and collider.sharedMaterial

### Direct Assignment

If you declare a public variable of any above class: Material, Mesh, or Physic Material, and make modifications to the asset using that variable instead of using the relevant class member, you will not receive the benefits of automatic instantiation before the modifications are applied.

### Assets that are not automatically instantiated

## Desktop

There are two different assets that are never automatically instantiated when modifying them.

- Texture2D
- TerrainData

Any modifications made to these assets through scripting are always permanent, and never undoable. So if you're changing your terrain's heightmap through scripting, you'll need to account for instantiating and assigning values on your own. Same goes for Textures. If you change the pixels of a texture file, the change is permanent.

## iOS

Texture2D assets are never automatically instantiated when modifying them. Any modifications made to these assets through scripting are always permanent, and never undoable. So if you change the pixels of a texture file, the change is permanent.

## Android

Texture2D assets are never automatically instantiated when modifying them. Any modifications made to these assets through scripting are always permanent, and never undoable. So if you change the pixels of a texture file, the change is permanent.

Page last updated: 2011-02-22

---

# Generating Mesh Geometry Procedurally

The Mesh class gives script access to an object's mesh geometry, allowing meshes to be created or modified at runtime. This technique is useful for graphical effects (eg, stretching or squashing an object) but can also be useful in level design and optimisation. The following sections explain the basic details of how a mesh is constructed along with an exploration of the API

and an example.


- Anatomy of a Mesh
- Using the Mesh Class
- Example - Creating a Billboard Plane


Page last updated: 2011-07-15

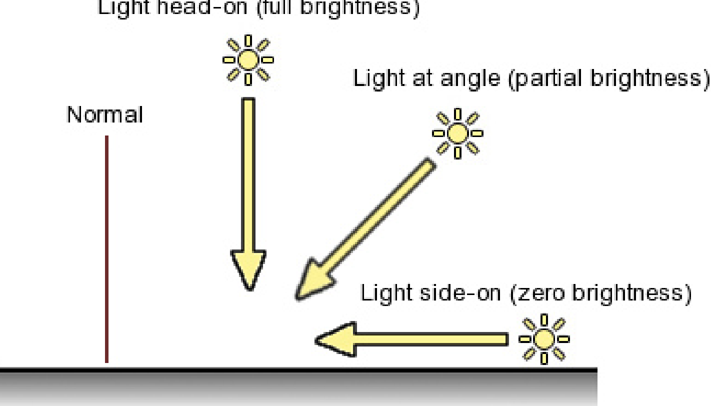
---

# Anatomy of a Mesh

A mesh consists of triangles arranged in 3D space to create the impression of a solid object. A triangle is defined by its three corner points or vertices. In the Mesh class, the vertices are all stored in a single array and each triangle is specified using three integers that correspond to indices of the vertex array. The triangles are also collected together into a single array of integers; the integers are taken in groups of three from the start of this array, so elements 0, 1 and 2 define the first triangle, 3, 4 and 5 define the second, and so on. Any given vertex can be reused in as many triangles as desired but there are reasons why you may not want to do this, as explained below.

### Lighting and Normals

The triangles are enough to define the basic shape of the object but extra information is needed to display the mesh in most cases. To allow the object to be shaded correctly for lighting, a normal vector must be supplied for each vertex. A normal is a vector that points outward, perpendicular to the mesh surface at the position of the vertex it is associated with. During the shading calculation, each vertex normal is compared with the direction of the incoming light, which is also a vector. If the two vectors are perfectly aligned, then the surface is receiving light head-on at that point and the full brightness of the light will be used for shading. A light coming exactly side-on to the normal vector will give no illumination to the surface at that position. Typically, the light will arrive at an angle to the normal and so the shading will be somewhere in between full brightness and complete darkness, depending on the angle.




Since the mesh is made up of triangles, it may seem that the normals at corners will simply be perpendicular to the plane of their triangle. However, normals are actually interpolated across the triangle to give the surface direction of the intermediate positions between the corners. If all three normals are pointing in the same direction then the triangle will be uniformly lit all over. The effect of having separate triangles uniformly shaded is that the edges will be very crisp and distinct. This is exactly what is required for a model of a cube or other sharp-edged solid but the interpolation of the normals can be used to create smooth shading to approximate a curved surface.

To get crisp edges, it is necessary to double up vertices at each edge since both of the two adjacent triangles will need their own separate normals. For curved surfaces, vertices will usually be shared along edges but a bit of intuition is often required to determine the best direction for the shared normals. A normal might simply be the average of the normals of the planes of the surrounding triangles. However, for an object like a sphere, the normals should just be pointing directly outward from the sphere's centre.

By calling Mesh.RecalculateNormals, you can get Unity to work out the normals' directions for you by making some assumptions about the "meaning" of the mesh geometry; it assumes that vertices shared between triangles indicate a smooth surface while doubled-up vertices indicate a crisp edge. While this is not a bad approximation in most cases,

RecalculateNormals will be tripped up by some texturing situations where vertices must be doubled even though the surface is smooth.

### Texturing

In addition to the lighting, a model will also typically make use of texturing to create fine detail on its surface. A texture is a bit like an image printed on a stretchable sheet of rubber. For each mesh triangle, a triangular area of the texture image is defined and that texture triangle is stretched and "pinned" to fit the mesh triangle. To make this work, each vertex needs to store the coordinates of the image position that will be pinned to it. These coordinates are two dimensional and scaled to the 0..1 range (0 means the bottom/left of the image and 1 means the right/top). To avoid confusing these coordinates with the Cartesian coordinates of the 3D world, they are referred to as U and V rather than the more familiar X and Y, and so they are commonly called UV coordinates.

Like normals, texture coordinates are unique to each vertex and so there are situations where you need to double up vertices purely to get different UV values across an edge. An obvious example is where two adjacent triangles use discontinuous parts of the texture image (eyes on a face texture, say). Also, most objects that are fully enclosed volumes will need a "seam" where an area of texture wraps around and joins together. The UV values at one side of the seam will be different from those at the other side.

Page last updated: 2011-07-15

---

## Using the Mesh Class

The Mesh class is the basic script interface to an object's mesh geometry. It uses arrays to represent the vertices, triangles, normals and texture coordinates and also supplies a number of other useful properties and functions to assist mesh generation.

### Accessing an Object's Mesh

The mesh data is attached to an object using the Mesh Filter component (and the object will also need a Mesh Renderer to make the geometry visible). This component is accessed using the familiar GetComponent function:-

```
var mf: MeshFilter = GetComponent(MeshFilter);
// Use mf.mesh to refer to the mesh itself.
```

### Adding the Mesh Data

The Mesh object has properties for the vertices and their associated data (normals and UV coordinates) and also for the triangle data. The vertices may be supplied in any order but the arrays of normals and UVs must be ordered so that the indices all correspond with the vertices (ie, element 0 of the normals array supplies the normal for vertex 0, etc). The vertices are Vector3s representing points in the object's local space. The normals are normalised Vector3s representing the directions, again in local coordinates. The UVs are specified as Vector2s, but since the Vector2 type doesn't have fields called U and V, you must mentally convert them to X and Y respectively.

The triangles are specified as triples of integers that act as indices into the vertex array. Rather than use a special class to represent a triangle the array is just a simple list of integer indices. These are taken in groups of three for each triangle, so the first three elements define the first triangle, the next three define the second triangle, and so on. An important detail of the triangles is the ordering of the corner vertices. They should be arranged so that the corners go around clockwise as you look down on the visible outer surface of the triangle, although it doesn't matter which corner you start with.

Page last updated: 2011-07-15

---

## Example - Creating a Billboard Plane

Unity comes with a Plane primitive object but a simpler plane may be useful in 2D games or GUI, and in any case makes a good starting example. A minimal plane will consist of four vertices to define the corners along with two triangles.

The first thing is to set the vertices array. We'll assume that the plane lies in the X and Y axes and let its width and height be determined by parameter variables. We'll supply the vertices in the order bottom-left, bottom-right, top-left, top-right.

```
var vertices: Vector3[] = new Vector3[4];

vertices[0] = new Vector3(0, 0, 0);
vertices[1] = new Vector3(width, 0, 0);
vertices[2] = new Vector3(0, height, 0);
vertices[3] = new Vector3(width, height, 0);

mesh.vertices = vertices;
```

(Since the Mesh data properties execute code behind the scenes, it is much more efficient to set up the data in your own array and then assign this to a property rather than access the property array element by element.)

Next come the triangles. Since we want two triangles, each defined by three integers, the triangles array will have six elements in total. Remembering the clockwise rule for ordering the corners, the lower left triangle will use 0, 2, 1 as its corner indices, while the upper right one will use 2, 3, 1.

```
var tri: int[] = new int[6];

//      Lower left triangle.
tri[0] = 0;
tri[1] = 2;
tri[2] = 1;

//      Upper right triangle.
tri[3] = 2;
tri[4] = 3;
tri[5] = 1;

mesh.triangles = tri;
```

A mesh with just the vertices and triangles set up will be visible in the editor but will not look very convincing since it is not correctly shaded without the normals. The normals for the flat plane are very simple - they are all identical and point in the negative Z direction in the plane's local space. With the normals added, the plane will be correctly shaded but remember that you need a light in the scene to see the effect.

```
var normals: Vector3[] = new Vector3[4];

normals[0] = -Vector3.forward;
normals[1] = -Vector3.forward;
normals[2] = -Vector3.forward;
normals[3] = -Vector3.forward;

mesh.normals = normals;
```

Finally, adding texture coordinates to the mesh will enable it to display a material correctly. Assuming we want to show the whole image across the plane, the UV values will all be 0 or 1, corresponding to the corners of the texture.

```
var uv: Vector2[] = new Vector2[4];

uv[0] = new Vector2(0, 0);
uv[1] = new Vector2(1, 0);
uv[2] = new Vector2(0, 1);
uv[3] = new Vector2(1, 1);

mesh.uv = uv;
```

The complete script might look a bit like this:-

```
var width: float;
var height: float;

function Start() {
	var mf: MeshFilter = GetComponent(MeshFilter);
	var mesh = new Mesh();
	mf.mesh = mesh;

	var vertices: Vector3[] = new Vector3[4];

	vertices[0] = new Vector3(0, 0, 0);
	vertices[1] = new Vector3(width, 0, 0);
	vertices[2] = new Vector3(0, height, 0);
	vertices[3] = new Vector3(width, height, 0);

	mesh.vertices = vertices;

	var tri: int[] = new int[6];

	tri[0] = 0;
	tri[1] = 2;
	tri[2] = 1;

	tri[3] = 2;
	tri[4] = 3;
	tri[5] = 1;

	mesh.triangles = tri;

	var normals: Vector3[] = new Vector3[4];
```

```
        normals[0] = -Vector3.forward;
        normals[1] = -Vector3.forward;
        normals[2] = -Vector3.forward;
        normals[3] = -Vector3.forward;

        mesh.normals = normals;

        var uv: Vector2[] = new Vector2[4];

        uv[0] = new Vector2(0, 0);
        uv[1] = new Vector2(1, 0);
        uv[2] = new Vector2(0, 1);
        uv[3] = new Vector2(1, 1);

        mesh.uv = uv;
}
```

Note that the if the code is executed once in the Start function then the mesh will stay the same throughout the game. However, you can just as easily put the code in the Update function to allow the mesh to be changed each frame (although this will increase the CPU overhead considerably).

Page last updated: 2011-08-15

---

# StyledText

The text for GUI elements and text meshes can incorporate multiple font styles and sizes. The GUIStyle, GUIText and TextMesh classes have a **Rich Text** setting which instructs Unity to look for markup tags within the text. These tags are not displayed but indicate style changes to be applied to the text.

## Markup format

The markup system is inspired by HTML but isn't intended to be strictly compatible with standard HTML. The basic idea is that a section of text can be enclosed inside a pair of matching tags:-

```
We are <b>not</b> amused
```

As the example shows, the tags are just pieces of text inside the "angle bracket" characters, < and >. The text inside the tag denotes its name (which in this case is just **b**). Note that the tag at the end of the section has the same name as the one at the start but with the slash / character added. The tags are not displayed to the user directly but are interpreted as instructions for styling the text they enclose. The b tag used in the example above applies boldface to the word "not", so the text will appear onscreen as:-

We are **not** amused

A marked up section of text (including the tags that enclose it) is referred to as an **element**.

### Nested elements

It is possible to apply more than one style to a section of text by "nesting" one element inside another

```
We are <b><i>definitely not</i></b> amused
```

The i tag applies italic style, so this would be presented onscreen as

We are ***definitely not*** amused

Note the ordering of the ending tags, which is in reverse to that of the starting tags. The reason for this is perhaps clearer when you consider that the inner tags need not span the whole text of the outermost element

```
We are <b>absolutely <i>definitely</i> not</b> amused
```

which gives

We are **absolutely *definitely* not** amused

**Tag parameters**
Some tags have a simple all-or-nothing effect on the text but others might allow for variations. For example, the **color** tag needs to know which colour to apply. Information like this is added to tags by the use of **parameters**:-

```
We are <color=green>green</color> with envy
```

Note that the ending tag doesn't include the parameter value. Optionally, the value can be surrounded by quotation marks but this isn't required.

## Supported tags

The following list describes all the styling tags supported by Unity.

**b**

Renders the text in boldface.

```
We are <b>not</b> amused
```

**i**

Renders the text in italics.

```
We are <i>usually</i> not amused
```

**size**

Sets the size of the text according to the parameter value, given in pixels.

```
We are <size=50>largely</size> unaffected
```

**color**

Sets the colour of the text according to the parameter value. The colour can be specified in the traditional HTML format

```
#rrggbbaa
```

...where the letters correspond to pairs of hexadecimal digits denoting the red, green, blue and alpha (transparency) values for the colour. For example, cyan at full opacity would be specified by

```
<color=#00ffffff>...
```

Another option is to use the name of the colour. This is easier to understand but naturally, the range of colours is limited and full opacity is always assumed.

```
<color=cyan>...
```

The available colour names are given in the table below.

| Colour name | Hex value | Swatch |
|---|---|---|
| aqua (same as cyan) | #00ffffff | |
| black | #000000ff | |
| blue | #0000ffff | |
| brown | #a52a2aff | |

| | | |
|---|---|---|
| cyan (same as aqua) | #00ffffff | |
| darkblue | #0000a0ff | |
| fuchsia (same as magenta) | #ff00ffff | |
| green | #008000ff | |
| grey | #808080ff | |
| lightblue | #add8e6ff | |
| lime | #00ff00ff | |
| magenta (same as fuchsia) | #ff00ffff | |
| maroon | #800000ff | |
| navy | #000080ff | |
| olive | #808000ff | |
| orange | #ffa500ff | |
| purple | #800080ff | |
| red | #ff0000ff | |
| silver | #c0c0c0ff | |
| teal | #008080ff | |
| white | #ffffffff | |
| yellow | #ffff00ff | |

**material**

This is only useful for text meshes and renders a section of text with a material specified by the parameter. The value is an index into the text mesh's array of materials as shown by the inspector.

```
We are <material=2>texturally</material> amused
```

**quad**

This is only useful for text meshes and renders an image inline with the text. It takes parameters that specify the material to use for the image, the image height in pixels, and a further four that denote a rectangular area of the image to display. Unlike the other tags, quad does not surround a piece of text and so there is no ending tag - the slash character is placed at the end of the initial tag to indicate that it is "self-closing".

```
<quad material=1 size=20 x=0.1 y=0.1 width=0.5 height=0.5 />
```

This selects the material at position in the renderer's material array and sets the height of the image to 20 pixels. The rectangular area of image starts at given by the x, y, width and height values, which are all given as a fraction of the unscaled width and height of the texture.

Page last updated: 2012-07-01

---

# UsingDLL

Usually, scripts are kept in a project as source files and compiled by Unity whenever the source changes. However, it is also possible to compile a script to a **dynamically linked library** (DLL) using an external compiler. The resulting DLL can then be added to the project and the classes it contains can be attached to objects just like normal scripts.

It is generally much easier to work with scripts than DLLs in Unity. However, you may have access to third party Mono code which is supplied in the form of a DLL. When developing your own code, you may be able to use compilers not supported by Unity (F#, for example) by compiling the code to a DLL and adding it to your Unity project. Also, you may want to supply Unity code without the source (for an Asset Store product, say) and a DLL is an easy way to do this.

### Creating a DLL

To create a DLL, you will first need a suitable compiler. Not all compilers that produce .NET code are guaranteed to work with Unity, so it may be wise to test the compiler with some available code before doing significant work with it. If the DLL contains no code that depends on the Unity API then you can simply compile it to a DLL using the appropriate compiler options. If you do want to use the Unity API then you will need to make Unity's own DLLs available to the compiler. On a Mac, these are contained in the application bundle (you can see the internal structure of the bundle by using the Show Package Contents command from the contextual menu; right click or ctrl-click the Unity application):-

The path to the Unity DLLs will typically be

```
/Applications/Unity/Unity.app/Contents/Frameworks/Managed/
```

...and the two DLLs are called UnityEngine.dll and UnityEditor.dll.

On Windows, the DLLs can be found in the folders that accompany the Unity application. The path will typically be

```
C:\Program Files (x86)\Unity\Editor\Data\Managed
```

...while the names of the DLLs are the same as for Mac OS.

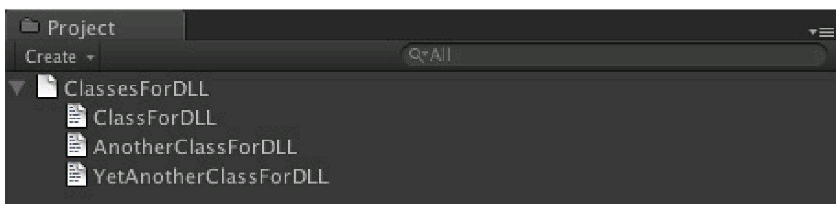
The exact options for compiling the DLL will vary depending on the compiler used. As an example, the command line for the Mono C# compiler, **mcs**, might look like this on Mac OS:-

```
mcs -r:/Applications/Unity/Unity.app/Contents/Frameworks/Managed/UnityEngine.dll -target
```

Here, the *-r* option specifies a path to a library to be included in the build, in this case the UnityEngine library. The *-target* option specifies which type of build is required; the word "library" is used to select a DLL build. Finally, the name of the source file to compile is *ClassesForDLL.cs* (it is assumed that this file is in the current working folder, but you could specify the file using a full path if necessary). Assuming all goes well, the resulting DLL file will appear shortly in the same folder as the source file.

### Using the DLL

Once compiled, the DLL file can simply be dragged into the Unity project like any other asset. The DLL asset has a foldout triangle which can be used to reveal the separate classes inside the library. Classes that derive from MonoBehaviour can be dragged onto Game Objects like ordinary scripts. Non-MonoBehaviour classes can be used directly from other scripts in the usual way.




*A folded-out DLL with the classes visible*

Page last updated: 2011-11-30

---

## Execution Order

In Unity scripting, there are a number of event functions that get executed in a predetermined order as a script executes. This

execution order is described below:

### First Scene Load

These functions get called when a scene starts (once for each object in the scene).

- **Awake:** This function is always called before any Start functions and also just after a prefab is instantiated. (If a GameObject is in-active during start up Awake is not called until it is made active, or a function in any script attached to it is called.)
- **OnEnable:** (only called if the Object is active): This function is called just after the object is enabled. This happens when a MonoBehaviour is instance is created, such as when a level is loaded or a GameObject with the script component is instantiated.

### Before the first frame update

- **Start:** Start is called before the first frame update only if the script instance is enabled.

### In between frames

- **OnApplicationPause:** This is called at the end of the frame where the pause is detected, effectively between the normal frame updates. One extra frame will be issued after **OnApplicationPause** is called to allow the game to show graphics that indicate the paused state.

### Update Order

When you're keeping track of game logic and interactions, animations, camera positions, etc., there are a few different events you can use. The common pattern is to perform most tasks inside the **Update()** function, but there are also other functions you can use.

- **FixedUpdate: FixedUpdate()** is often called more frequently than **Update()**. It can be called multiple times per frame, if the frame rate is low and it may not be called between frames at all if the frame rate is high. All physics calculations and updates occur immediately after **FixedUpdate()**. When applying movement calculations inside **FixedUpdate()**, you do not need to multiply your values by **Time.deltaTime**. This is because **FixedUpdate()** is called on a reliable timer, independent of the frame rate.

- **Update: Update()** is called once per frame. It is the main workhorse function for frame updates.

- **LateUpdate: LateUpdate()** is called once per frame, after **Update()** has finished. Any calculations that are performed in **Update()** will have completed when **LateUpdate()** begins. A common use for **LateUpdate()** would be a following third-person camera. If you make your character move and turn inside **Update()**, you can perform all camera movement and rotation calculations in **LateUpdate()**. This will ensure that the character has moved completely before the camera tracks its position.

### Rendering

- **OnPreCull:** Called before the camera culls the scene. Culling determines which objects are visible to the camera. OnPreCull is called just before culling takes place.
- **OnBecameVisible/OnBecameInvisible:** Called when an object becomes visible/invisible to any camera.
- **OnWillRenderObject:** Called **once** for each camera if the object is visible.
- **OnPreRender:** Called before the camera starts rendering the scene.
- **OnRenderObject:** Called after all regular scene rendering is done. You can use GL class or Graphics.DrawMeshNow to draw custom geometry at this point.
- **OnPostRender:** Called after a camera finishes rendering the scene.
- **OnRenderImage(Pro only):** Called after scene rendering is complete to allow postprocessing of the screen image.
- **OnGUI:** Called multiple times per frame in response to GUI events. The Layout and Repaint events are processed first, followed by a Layout and keyboard/mouse event for each input event.
- **OnDrawGizmos** Used for drawing Gizmos in the scene view for visualisation purposes.

### Coroutine

Normal coroutine updates are run after the Update function returns. A coroutine is function that can suspend its execution (yield) until the given given YieldInstruction finishes. Different uses of Coroutines:

- **yield;** The coroutine will continue after all Update functions have been called on the next frame.
- **yield WaitForSeconds(2);** Continue after a specified time delay, after all Update functions have been called for the frame
- **yield WaitForFixedUpdate();** Continue after all FixedUpdate has been called on all scripts
- **yield WWW** Continue after a WWW download has completed.
- **yield StartCoroutine(MyFunc);** Chains the coroutine, and will wait for the MyFunc coroutine to complete first.

### When the Object is Destroyed

- **OnDestroy:** This function is called after all frame updates for the last frame of the object's existence (the object might be destroyed in response to Object.Destroy or at the closure of a scene).

### When Quitting

These functions get called on all the active objects in your scene, :

- **OnApplicationQuit:** This function is called on all game objects before the application is quit. In the editor it is called when the user stops playmode. In the web player it is called when the web view is closed.
- **OnDisable:** This function is called when the behaviour becomes disabled or inactive.

**So in conclusion, this is the execution order for any given script:**

- All Awake calls
- All Start Calls
- **while** (stepping towards variable delta time)
  - All FixedUpdate functions
  - Physics simulation
  - OnEnter/Exit/Stay trigger functions
  - OnEnter/Exit/Stay collision functions
- Rigidbody interpolation applies transform.position and rotation
- OnMouseDown/OnMouseUp etc. events
- All Update functions
- Animations are advanced, blended and applied to transform
- All LateUpdate functions
- Rendering

### Hints

- Coroutines are executed after all Update functions.

Page last updated: 2012-10-10

---

## iphone-PracticalGuide

This guide is for developers new to mobile game development, who are probably feeling overwhelmed, and are either planning and prototyping a new mobile game or porting an existing project to run smoothly on a mobile device. It should also be useful as a reference for anyone making mobile games or browser games which target old PCs and netbooks.

Optimization is a broad topic, and how you do it depends a lot on your game, so this guide is best read as an introduction or reference rather than a step-by-step guide that guarantees a smooth product.

### All mobile devices are not created equal

The information here assumes hardware around the level of the Apple A4 chipset, which is used on the original iPad, the iPhone 3GS, and the 3rd generation iPod Touch. On the Android side, that would mean an Android phone such as the Nexus One, or most phones that run Android 2.3 Gingerbread. On average, these devices were released in early 2010. Out of the app-hungry market, these devices are the older, slower portion. But they should be supported, because they represent a large portion of the market.

There are much slower, and much faster phones out there as well. The computational capability of mobile devices is increasing at an alarming rate. It's not unheard of for a new generation of a mobile GPU to be five times faster than its predecessor. That's ***fast***, when compared to the PC industry.

For an overview of Apple mobile device tech specs, see the Hardware page.

If you want to develop for mobile devices which will be popular in the future, or exclusively for high end devices right now, you will be able to get away with doing more. See Future Mobile Devices.

The very low end, such as the iPhone 3G and the first and second generation iPod touches, are extremely limited and even more care must be taken to optimize for them. However, there is some question to whether consumers who have not upgraded their device will be buying apps. So unless you are making a free app, it might not be worthwhile to support the old hardware.

## Make optimization a design consideration, not a final step

British computer scientist Michael A. Jackson is often quoted for his Rules of Program Optimization:

*The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization (for experts only!): Don't do it yet.*

His rationale was that, considering how fast computers are, and how quickly their speed is increasing, there is a good chance that if you program something it will run fast enough. Besides that, if you try to optimize too heavily, you might over-complicate things, limit yourself, or create tons of bugs.

However, if you are developing mobile games, there is another consideration: The hardware that is on the market right now is very limited compared to the computers we are used to working with, so the risk of creating something that simply won't run on the device balances out the risk of over-complication that comes with optimizing from the start.

Throughout this guide we will try to point out situations where an optimization would help a lot, versus situations where it would be frivolous.

### Optimization: Not just for programmers

Artists also need to know the limitations of the platform and the methods that are used to get around them, so they can make creative choices that will pay off, and don't have to redo work.

- More responsibility can fall on the artist if the game design calls for atmosphere and lighting to be drawn into textures instead of being baked.
- Whenever anything can be baked, artists can produce content for baking, instead of real-time rendering. This allows them to ignore technical limitations and work freely.

### Design your game to make a smooth runtime fall into your lap

These two pages detail general trends in game performance, and will explain how you can best design your game to be optimized, or how you can intuitively figure out which things need to be optimized if you've already gone into production.

- Practical Methods for Optimized Rendering
- Practical Methods for Optimized Scripting and Gameplay

## Profile early and often

Profiling is important because it helps you discern which optimizations will pay off with big performance increases and which ones are a waste of your time. Because of the way that rendering is handled on a separate chip (GPU), the time it takes to render a frame is not the time that the CPU takes plus the time that the time that the GPU takes, instead it is the longer of the two. That means that if the CPU is slowing things down, optimizing your shaders won't increase the frame rate at all, and if the GPU is slowing things down, optimizing physics and scripts won't help at all.

Often different parts of the game and different situations perform differently as well, so one part of the game might cause 100 millisecond frames entirely due to a script, and another part of the game might cause the same slowdown, but because of something that is being rendered. So, at very least, you need to know where all the bottlenecks are if you are going to optimize your game.

### Unity Profiler (Pro only)

The main Profiler in Unity can be used when targeting iOS or Android. See the Profiler guide for basic instructions on how to use it.

### Internal Profiler

The internal profiler spews out text every 30 frames. It can help you figure out which aspects of your game are slowing things down, be it physics, scripts, or rendering, but it doesn't go into much detail, for example, which script or which renderer is the culprit.

See the Internal Profiler page for more details on how it works and how to turn it on.

### Profiler indicates most of time spent rendering

- Rendering Optimizations

### Profiler indicates most of time spent outside of rendering

- Scripting Optimizations

**Table of Contents**


- Practical Guide to Optimization for Mobiles - Future & High End Devices
- Practical Guide to Optimization for Mobiles - Graphics Methods
- Practical Guide to Optimization for Mobiles - Scripting and Gameplay Methods
- Practical Guide to Optimization for Mobiles - Rendering Optimizations
- Practical Guide to Optimization for Mobiles - Optimizing Scripts


Page last updated: 2012-11-02

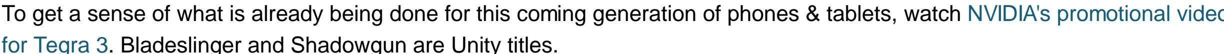
---

# iphone-FutureDevices

The graphical power of next-generation mobile devices is approaching that of the current generation of consoles (Wii, Xbox 360, and PS3). What will the consumer smartphone market look like in two years? It's hard to say for sure, but considering how things have been going, the average smartphone on the market will have a chipset about as fast as NVIDIA's Tegra 3 (Asus Transformer Prime, Google Nexus 7"), or Apple's A5X (iPad 3), and high-end tablets will pack graphical performance to rival today's consoles and consumer laptops.

### What can these new devices do?

- Bumpmaps everywhere
- Reflective water & simple image effects
- Realtime shadows (Unity 4.0 feature)
- HD video playback
- Faster script execution



To get a sense of what is already being done for this coming generation of phones & tablets, watch NVIDIA's promotional video for Tegra 3. Bladeslinger and Shadowgun are Unity titles.

Page last updated: 2012-11-06

---

# iphone-OptimizedGraphicsMethods

What are mobile devices capable of? How should you plan your game accordingly? If your game runs slow, and the profiler indicates that it's a rendering bottleneck, how do you know what to change, and how to make your game look good but still run fast? This page is dedicated to a general and non-technical exposition of the methods. If you are looking for the specifics, see the Rendering Optimizations page.

### What you can reasonably expect to run on current consumer mobiles:

- Lightmapped static geometry. But beware of:
  - Using a lot of alpha-test shaders
  - Bumpmapping, especially using built-in shaders.
  - High polygon count
- Animated characters, even with fancy shaders! But beware of:
  - Massive crowds or high-poly characters
- 2D games with sprites. But beware of:
  - Overdraw, or, lots of layers drawn on top of eachother.
- Particle effects. But beware of:
  - High density on large particles. (Lots of particles drawn on top of each other. This is another overdraw situation)
  - Ridiculous numbers of particles, or particle colliders.
- Physics. But beware of:

- Mesh colliders.
- Lots of active bodies.

### What you CANNOT reasonably expect to run on current consumer mobiles:

- Fullscreen screen image effects like glow and depth of field.
- Dynamic per-pixel lighting (multiple lights marked Important and not baked into the lightmap)
  - Every affected object is drawn an additional time for every dynamic light you use, and this gets slow quickly.
- Real time shadows on everything
  - Unity 4 offers native support for real time shadows on mobile platforms, but their use must be very judicious, and likely limited to higher-end devices.

## Examples - How top-notch mobile games are made

#### Shadowgun

Shadowgun is an impressive example of what can be done on current mobile hardware. But more specifically, it's a good example of what cannot be done, and how to get around the limitations. Especially because a small part of the game has been made **publicly available** in this blog post.

Here's a basic rundown of things that Shadowgun does in order to keep performance up:

- Dynamic lighting - barely used.
  - Blob shadows and Lightmaps are used instead of any real shadows.
  - Lightprobes, instead of real lights, are used on their characters.
    - Muzzle flashes added into the lightprobe data via script.
  - The only dynamic per-pixel lighting is an arbitrary light direction used to calculate a BRDF on the characters.
- Bumpmapping - barely used.
  - Real bumpmapping only used on characters.
  - As much contrast and detail as possible is baked into the diffuse texture maps. Lighting information from bumpmaps is baked in.
  - A good example is their statue texture, or their shiny wall, as seen on the right. No bumpmaps are used to render these, the specularity is faked by baking it into the texture. Lightmapping is combined with a vertex-lighting-based specular highlight to give these models a shiny look.
  - If you want to learn how to create textures like this one, check out the Rendering Optimizations page.

- Dense particles - avoided.
  - UV-scrolling textures used instead of dense particle effects.
- Fog effects - avoided.
  - Their god rays are hand-modeled.
  - Single planes that fade in and out are used to achieve cinematic fog effects without actually rendering any fog.
    - This is faster because the planes are few and far between, and it means that fog doesn't have to be calculated on every pixel and in every shader.
- Glow - avoided.
  - Blended sprite planes are used to give the appearance of a glow on certain objects.

### Sky Castle Demo

This demo was designed to show what Unity is capable of on high-end Android devices.

- Dynamic lighting - not used.
  - Lightmaps only.
- Bumpmapping - used
  - The bricks are all bumpmapped, lit by directional lightmaps. This is where the "high-end devices" part comes into play.
- Real time reflections - limited.
  - They carefully placed their real-time reflecting surfaces separately and in isolated areas, so that only one runs at a time, and the environment that needs to be rendered twice can be easily culled.

## Bottom line - What this means for your game

The more you respect and understand the limitations of the mobile devices, the better your game will look, and the smoother it will perform. If you want to make a high-class game for mobile, you will benefit from understanding Unity's graphics pipeline and being able to write your own shaders. But if you want something to grab to use right away, ShadowGun's shaders, available here, are a good place to start.

#### Don't Simulate It, Bake It !

There is no question that games attempt to follow the laws of nature. The movement of every parabolic projectile and the color of every pixel of shiny chrome is derived by formulas first written to mimic observations of the real world. But a game is one part scientific simulation and one part painting. You can't compete in the mobile market with physically accurate rendering; the hardware simply isn't there yet, if you try to imitate the real world all the way, your game will end up limited, drab, and laggy.

You have to pick up your polygons and your blend modes like they're paintbrushes.

The baked bumpmaps shown in Shadowgun are great examples of this. There are specular highlights already in the texture - the human eye doesn't notice that they don't actually line up with the reflected light and view directions - they are simply high-contrast details on the texture, completely faked, yet they end up looking great. This is a common cheating technique which has been used in many successful games. Compare the visor in the first Halo screenshot ever released with the visor from this release screenshot. It appears that the armor protrusions from the top of the helmet are reflected in the visor, but the

reflection is actually baked into the visor texture. In League of Legends, a spell effect appears to have a pixel-light attached to it, but it actually is a blended plane with a texture that was probably generated by taking a screenshot of a pixel light shining on the ground.

**What works well:**

- Lightmapped static geometry
  - Dramatic lighting and largely dynamic environments don't mix. Pick one or the other.
- Lightprobes for moving objects
  - Current mobile hardware is not really cut out for lots of dynamic lights, and it can't do shadows. Lightprobes are a really neat solution for complex game worlds with static lighting.
- Specialized shaders and detailed, high-contrast textures
  - The shaders in ShadowGun minimize per-pixel calculations and exploit complex and high-quality textures. See our Rendering Optimizations page for information on how to make textures that look great even when the shader is simple.
- Cartoon Graphics
  - Who says your game has to look like a photo? If you make lighting and atmosphere the responsibility of the texture artist, not the engine, you hardly even have to worry about optimizing rendering.

**What does not work:**

- Glow and other Post processing effects
  - Approximate such effects when possible by using blended quads, check out the Shadowgun project for an example of this.
- Bumpmapping, especially with the built-in shaders
  - Use it sparingly, only on the most important characters or objects. Anything that can take up the whole screen probably shouldn't be bumpmapped.
  - Instead of using bump maps, bake more detail and contrast into the diffuse texture. The effect from League of Legends is an interesting example of this being used successfully in the industry.

**But how do I actually *do* it?**
See our Rendering Optimizations page.

Page last updated: 2012-11-06

---

# iphone-OptimizedScriptingMethods

This section demonstrates ways that mobile developers write code and structure their games so that they run fast. The core idea here is that game design and optimization aren't really separate processes; decisions you make when you are designing your game can make it both fun and fast.

## A historical example

You may remember old games where the player was only allowed one shot on the screen at a time, and reload speed was controlled by whether the bullet missed or not, instead of a timer. This technique is called **object pooling**, and it simplifies memory management, making programs run smoother.

The creators of space invaders only had a small amount of RAM, and they had to ensure that their program would never need to allocate more than was available. If they let the player fire once every second, and they offered a powerup that decreased the reload time to a half a second, they would have to ensure that there was enough memory space to allocate a lot of projectiles in the case where the player fires as fast as possible and all of the shots live for the longest possible time. That would probably pose a problem for them, so instead, they just allocated one projectile and left it at that. As soon as the projectile dies, it is simply deactivated, and repositioned and activated when it is fired again. But it always lives in the same space in memory and doesn't have to move around or be constantly deleted and recreated.

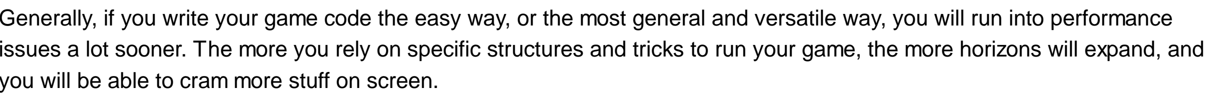
**An optimization, or a gameplay gem?**
This is hardly realistic, but it happens to be **fun**. Tension is released in a climactic moment when the alien invaders approach the ground, similar to a climax in film or literature. The invaders' close proximity gives the adept player near-instantaneous reload time, allowing them to miraculously defend earth by mashing the fire key in perfect time. Good game designs live in a

bizarre space between the interactive narrative and the background technology that powers it all. It's hard to plan out awesome, fun, efficient stuff like this, because code logistics and user interaction are two wildly different and deeply finicky things, and using them together to synthesize something fresh and fun takes a lot of thought and experimentation.

You probably can't plan out every aspect of your game in terms of interaction and playing nice with mobile hardware simultaneously. It's more likely that these "gems" where the two meet in harmony will pop up as accidents while you're experimenting. But having a solid understanding of the way your code runs on the hardware you intend to deploy on will help. If you want to see the detailed technical explanation of why object pooling is better, and learn about memory allocation, see our Scripting Optimizations page.

### Will X run fast on Mobiles?

Say you are beginning to work on a game, and you want to impress your players with lots of action and flashy stuff happening at once. How do you plan those things out? How do you know where the limits are, in game terms like how many coins, how many zombies, how many opponent cars, etc? It all depends on how you code your game.



Generally, if you write your game code the easy way, or the most general and versatile way, you will run into performance issues a lot sooner. The more you rely on specific structures and tricks to run your game, the more horizons will expand, and you will be able to cram more stuff on screen.

**Easy and versatile, but slow**

- Rigidbodies limited to 2 dimensions in a 2D game.
- Rigidbodies on projectiles.
- Using Instantiate and Destroy a lot.
- Lots of individual 3D objects for collectables or characters.
- Performing calculations every frame.
- Using OnGUI for your GUI or HUD.

**Complicated and limited, but faster**

- Writing your own physics code for a 2D game.
- Handling collision detection for projectiles yourself.
- Using Object Pooling instead of Instantiate and Destroy.
- Using animated sprites on particles to represent simple objects.
- Performing expensive calculations every few frames and caching the results.
- A custom GUI solution.

## Examples

**Hundreds of rotating, dynamically lit, collectable coins onscreen at once**

- NO: Each coin is a separate object with a rigidbody and a script that rotates it and allows it to be picked up.
- YES: The coins are a particle system with an animated texture, one script does the collision testing for all the coins and sets their color according to distance from a light.
  - This example is implemented in the Scripting Optimization page.

**Your custom-built soft-body simulation**

- NO: The world has pillows lying around everywhere, which you can throw around and make piles of.
- YES: Your character is a pillow, there is only one of them, and the situations it will be in are somewhat predictable (It only collides with spheres and axis-aligned cubes). You can probably code something which isn't a full-featured softbody

simulation, but looks really impressive and runs fast.

**30 enemy characters shooting at the player at once**

- NO: Each enemy has his own skinned mesh, a separate object for his weapon, and instantiates a rigidbody-based projectile every time he fires. Each enemy takes the state of all of his compatriots into account in a complicated AI script that runs every frame.
- YES: Most of the enemies are far away, and are represented by single sprites, or, the enemies are 2D and are just a couple sprites anyway. Every enemy bullet is drawn by the same particle system and simulated by a script which does only rudimentary physics. Each enemy updates his AI state twice per second according to the state of the other enemies in his sector.

### The how and why of script optimization

See our page on Optimizing Scripts.

Page last updated: 2012-11-06

---

# iphone-PracticalRenderingOptimizations

This section introduces the technicalities of rendering optimization. It shows how to bake lighting results for better performance, and how the developers of Shadowgun levered high-contrast textures, with lighting baked-in, to make their game look great. If you are looking for general information on what a mobile-optimized game looks like, check out the Graphics Methods page.

### Get Artsy!

Sometimes optimizing the rendering in your game requires some dirty work. All of the structure that Unity provides makes it easy to get something working fast, but if you require top notch fidelity on limited hardware, doing things yourself and sidestepping that structure is the way to go, provided that you can introduce a key structural change that makes things a lot faster. Your tools of choice are editor scripts, simple shaders, and good old-fashioned art production.

**Note for Unity Indie users:** The editor scripts referenced here use RenderTextures to make production smooth, so they wont work for you right away, but the principles behind them work with screenshotting as well, so nothing is stopping you from using these techniques for a few texture bakes of your own.

**How to Dive Under the Hood**

First of all, check out this introduction to how shaders are written.

- Built in shaders
  - Examine the source code of the built in shaders. Often, if you want to make a new shader that does something different, you can achieve it by taking parts of two already-existing shaders and putting them together.
- Surface Shader Debugging (#pragma debug)
  - A CG Shader is generated from every surface shader, and then fully compiled from there. If you add **#pragma debug** to the top of your surface shader, when you open the compiled shader via the inspector, you can see the intermediate CG code. This is useful for inspecting how a specific part of a shader is actually calculated, and it can also be useful for grabbing certain aspects you want from a surface shader and applying them to a CG shader.
- Shader Include Files
  - A lot of shader helper code is included in every shader, and usually it isn't used, but this is why you will sometimes see shaders calling functions like WorldReflectionVector which don't seem to be defined anywhere. Unity has several built-in shader include files that contain these helper definitions. To find a specific function, you will need to search through all of the different includes.
  - These files are a major part of internal structure that Unity uses to make it easy to write shaders; the files provide things like real time shadows, different light types, lightmaps, and multiple platform support.
- Hardware documentation
  - Take your time to study Apple documentations on

hardware and best practices for writing shaders. Note that we would suggest to be more aggressive with floating point precision hints however.

## Shadowgun in-depth

Shadowgun is a great graphical achievement considering the hardware it runs on. While the art quality seems to be the key to the puzzle, there are a couple tricks to achieving such quality that programmers can pull off to maximize their artists' potential.

In the Graphics Methods page we used the golden statue in Shadowgun as an example of a great optimization; instead of using a normal map to give their statue some solid definition, they just baked lighting detail into the texture. Here, we will show you how and why you should use a similar technique in your own game.

```
+ Show [Shader code for Real-Time vs Baked Golden Statue] +
```

Reflective Bumped Specular    Baked Light with Reflection

**Render to Texel**

The real-time light is definitely higher quality, but the performance gain from the baked version is massive. So how was this done? An editor tool called Render to Texel was created for this purpose. It bakes the light into the texture through the following process:

- Transform the tangent space normal map to world space via script.
- Create a world space position map via script.
- Render to Texture a fullscreen pass of a the entire texture using the two previous maps, with one additional pass per light.
- Average results from several different vantage points to yield something which looks plausible all around, or at least from common viewing angles in your game.

This is how the best graphics optimizations work. They sidestep tons of calculations by preforming them in the editor or before the game runs. In general, this is what you want to do:

- Create something that looks great, don't worry about performance.
- Use tools like Unity's lightmapper and editor extensions like Render to Texel and Sprite Packer to bake it down to something which is very simple to render.
    - Making your own tools is the best way to do this, you can create the perfect tool suited for whatever problem your game presents.
- Create shaders and scripts which modulate your baked output to give it some sort of "shine"; an eye-catching effect to create an illusion of dynamic light.

**Concept of Light Frequency**

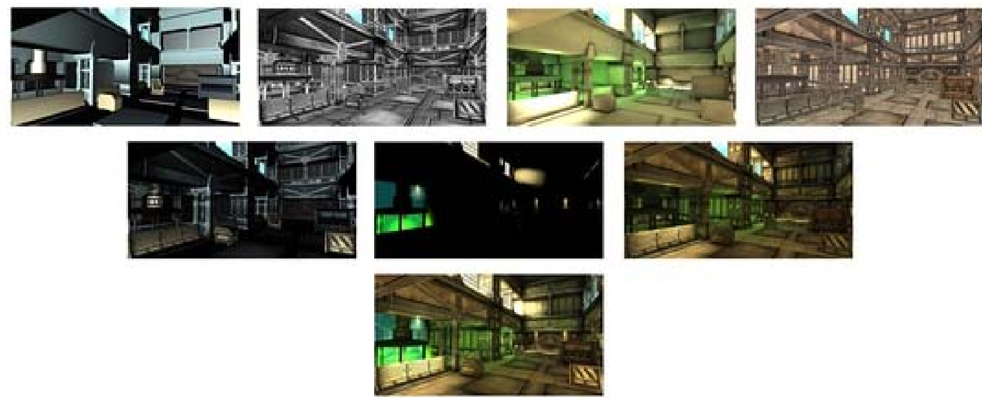
Just like the Bass and Treble of an audio track, images also have high-frequency and low-frequency components, and when you're rendering, it's best to handle them in different ways, similar to how stereos use subwoofers and tweeters to produce a full body of sound. One way to visualize the different frequencies of an image is to use the "High Pass" filter in Photoshop. **Filters->Other->High Pass**. If you have done audio work before, you will recognize the name High Pass. Essentially what it

does is cut off all frequencies lower than X, the parameter you pass to the filter. For images, Gaussian Blur is the equivalent of a Low Pass.

This applies to realtime graphics because frequency is a good way to separate things out and determine how to handle what. For example, in a basic lightmapped environment, the final image is obtained by composite of the lightmap, which is low frequency, and the textures, which are high-frequency. In Shadowgun, low frequency light is applied to characters quickly with light probes, high frequency light is faked through the use of a simple bumpmapped shader with an arbitrary light direction.

In general, by using different methods to render different frequencies of light, for example, baked vs dynamic, per-object vs per-level, per pixel vs per-vertex, etc, you can create full bodied images on limited hardware. Stylistic choices aside, it's generally a good idea to try to have strong variation colors or values at both high and low frequencies.

**Frequency in Practice: Shadowgun Decomposition**



- Top Row
  - Ultra-Low-Frequency Specular Vertex Light (Dynamic) | High Frequency Alpha Channel | Low Frequency Lightmap | High Frequency Albedo
- Mid Row
  - Specular Vertex Light * Alpha | High Frequency Additive Details | Lightmap * Color Channel
- Bottom
  - Final Sum

**Note**: Usually these decompositions refer to steps in a deferred renderer, but that's not the case here. All of this is done in just one pass. These are the two relevant shaders which this composition was based on:

```
+ Show [Lightmapped with Virtual Gloss Per-Vertex Additive] +
```

```
+ Show [Lightprobes with Virtual Gloss Per-Vertex Additive] +
```

## Best Practices

**GPU optimization: Alpha-Testing**
Some GPUs, particularly ones found in mobile devices, incur a high performance overhead for alpha-testing (or use of the **discard** and **clip** operations in pixel shaders). You should replace alpha-test shaders with alpha-blended ones if possible. Where alpha-testing cannot be avoided, you should keep the overall number of visible alpha-tested pixels to a minimum.

**iOS Texture Compression**
Some images, especially if using iOS/Android PVR texture compression, are prone to visual artifacts in the alpha channel. In such cases, you might need to tweak the PVRT compression parameters directly in your imaging software. You can do that by installing the **PVR export plugin** or using **PVRTexTool** from Imagination Tech, the creators of the PVRTC format. The resulting compressed image file with a **.pvr** extension will be imported by the Unity editor directly and the specified compression parameters will be preserved. If PVRT-compressed textures do not give good enough visual quality or you need especially crisp imaging (as you might for GUI textures) then you should consider using 16-bit textures instead of 32-bit. By doing so, you will reduce the memory bandwidth and storage requirements by half.

**Android Texture Compression**
All Android devices with support for OpenGL ES 2.0 also support the ETC1 compression format; it's therefore encouraged to

whenever possible use ETC1 as the prefered texture format.

If targeting a specific graphics architecture, such as the Nvidia Tegra or Qualcomm Snapdragon, it may be worth considering using the proprietary compression formats available on those architectures. The Android Market also allows filtering based on supported texture compression format, meaning a distribution archive (.apk) with for example DXT compressed textures can be prevented for download on a device which doesn't support it.

#### An Exercise

- Download Render to Texel.
- Bake lighting on your model.
- Run the High Pass filter on the result in Photoshop.
- Edit the "Mobile/Cubemapped" shader, included in the Render to Texel package, so that the missing low-frequency light details are replaced by vertex light.

Page last updated: 2012-11-06

---

# iphone-PracticalScriptingOptimizations

This section demonstrates how you would go about optimizing the actual scripts and methods your game uses, and it also goes into detail about the reasons why the optimizations work, and why applying them will benefit you in certain situations.

## Profiler is King (Unity Pro)

There is no such thing as a list of boxes to check that will ensure your project runs smoothly. To optimize a slow project, you have to profile to find specific offenders that take up a disproportionate amount of time. Trying to optimize without profiling or without thoroughly understanding the results that the profiler gives is like trying to optimize with a blindfold on.

So, if you want to make a technologically demanding game that runs on mobile platforms, you probably need Unity Pro for the Profiler.

#### What About Indie?

You can use the internal profiler to figure out what kind of process is slowing your game down, be it physics, scripts, or rendering, but you can't drill down into specific scripts and methods to find the actual offenders. However, by building switches into your game which enable and disable certain functionality, you can narrow down the worst offenders significantly. For example, if you remove the enemy characters' AI script and the framerate doubles, you know that the script, or something that it brings into the game, has to be optimized. The only problem is that you may have to try a lot of different things before you find the problem.

For more about profiling on mobile devices, see the profiling section.

## Optimized by Design

Attempting to develop something which is fast from the beginning is risky, because there is a trade-off between wasting time making things that would be just as fast if they weren't optimized and making things which will have to be cut or replaced later because they are too slow. It takes intuition and knowledge of the hardware to make good decisions in this regard, especially because every game is different and what might be a crucial optimization for one game may be a flop in another.

#### Object Pooling

We gave object pooling as an example of the intersection between good gameplay and good code design in our introduction to optimized scripting methods. Using object pooling for ephemeral objects is faster than creating and destroying them, because it makes memory allocation simpler and removes dynamic memory allocation overhead and Garbage Collection, or GC.

#### Memory Allocation

```
+ Show [Simple Explanation of what Automatic Memory Management is] +
```

- Read more about Automatic Memory Management and the Garbage Collector.

#### How to Avoid Allocating Memory

Every time an object is created, memory is allocated. Very often in code, you are creating objects without even knowing it.

- **Debug.Log("boo" + "hoo");** creates an object.
  - Use **System.String.Empty** instead of **""** when dealing with lots of strings.
- Immediate Mode GUI (UnityGUI) is slow and should not be used at any time when performance is an issue.
- Difference between **class** and **struct**:

  `+ Show [Class vs Struct] +`

- Objects which stick around for a long time should be classes, and objects which are ephemeral should be structs. Vector3 is probably the most famous struct. If it were a class, everything would be a lot slower.

**Why Object Pooling is Faster**

The upshot of this is that **using Instantiate and Destroy a lot gives the Garbage Collector a lot to do**, and this can cause a "hitch" in gameplay. As the Automatic Memory Management page explains, there are other ways to get around the common performance hitches that surround Instantiate and Destroy, such as triggering the Garbage Collector manually when nothing is going on, or triggering it very often so that a large backlog of unused memory never builds up.

Another reason is that, when a specific prefab is instantiated for the first time, sometimes additional things have to be loaded into RAM, or textures and meshes need to be uploaded to the GPU. This can cause a hitch as well, and with object pooling, this happens when the level loads instead of during gameplay.

Imagine a puppeteer who has an infinite box of puppets, where every time the script calls for a character to appear, he gets a new copy of its puppet out of the box, and every time the character exits the stage, he tosses the current copy. Object pooling is the equivalent of getting all the puppets out of the box before the show starts, and leaving them on the table behind the stage whenever they are not supposed to be visible.

**Why Object Pooling can be Slower**

One issue is that the creation of a pool reduces the amount of heap memory available for other purposes; so if you keep allocating memory on top of the pools you just created, you might trigger garbage collection even more often. Not only that, every collection will be slower, because the time taken for a collection increases with the number of live objects. With these issues in mind, it should be apparent that performance will suffer if you allocate pools that are too large or keep them active when the objects they contain will not be needed for some time. Furthermore, many types of objects don't lend themselves well to object pooling. For example, the game may include spell effects that persist for a considerable time or enemies that appear in large numbers but which are only killed gradually as the game progresses. In such cases, the performance overhead of an object pool greatly outweighs the benefits and so it should not be used.

**Implementation**

Here's a simple side by side comparison of a script for a simple projectile, one using Instantiation, and one using Object Pooling.

`+ Show [Object Pooling Example] +`

Of course, for a large, complicated game, you will want to make a generic solution that works for all your prefabs.

## Another Example: Coin Party!

The example of "Hundreds of rotating, dynamically lit, collectable coins onscreen at once" which was given in the Scripting Methods section will be used to demonstrate how script code, Unity components like the Particle System, and custom shaders can be used to create a stunning effect without taxing the weak mobile hardware.

Imagine that this effect lives in the context of a 2D sidescrolling game with tons of coins that fall, bounce, and rotate. The coins are dynamically lit by point lights. We want to capture the light glinting off the coins to make our game more impressive.

If we had powerful hardware, we could use a standard approach to this problem. Make every coin an object, shade the object with either vertex-lit, forward, or deferred lighting, and then add glow on top as an image effect to get the brightly reflecting coins to bleed light onto the surrounding area.

But mobile hardware would choke on that many objects, and a glow effect is totally out of the question. So what do we do?

**Animated Sprite Particle System**

If you want to display a lot of objects which all move in a similar way and can never be carefully inspected by the player, you might be able to render large amounts of them in no time using a particle system. Here are a few stereotypical applications of

this technique:

- Collectables or Coins
- Flying Debris
- Hordes or Flocks of Simple Enemies
- Cheering Crowds
- Hundreds of Projectiles or Explosions

There is a free editor extension called Sprite Packer that facilitates the creation of animated sprite particle systems. It renders frames of your object to a texture, which can then be used as an animated sprite sheet on a particle system. For our use case, we would use it on our rotating coin.

#### Reference Implementation

Included in the Sprite Packer project is an example that demonstrates a solution to this exact problem.

It uses a family of assets of all different kinds to achieve a dazzling effect on a low computing budget:

- A control script
- Specialized textures created from the output of the SpritePacker
- A specialized shader which is intimately connected with both the control script and the texture.

A readme file is included with the example which attempts to explain why and how the system works, outlining the process that was used to determine what features were needed and how they were implemented. This is that file:

`+ Show [Coin Party README] +`

The end goal of this example or "moral of the story" is that if there is something which your game really needs, and it causes lag when you try to achieve it through conventional means, that doesn't mean that it is impossible, it just means that you have to put in some work on a system of your own that runs much faster.

#### Techniques for Managing Thousands of Objects

These are specific scripting optimizations which are applicable in situations where hundreds or thousands of dynamic objects are involved. Applying these techniques to every script in your game is a terrible idea; they should be reserved as tools and design guidelines for large scripts which handle tons of objects or data at run time.

- Avoid or minimize $O(n^2)$ operations on large data sets
  `+ Show [Order N Squared] +`

- Cache references instead of performing unnecessary searches
  `+ Show [Reference Caching] +`

- Minimize expensive math functions
  `+ Show [Expensive Math Functions] +`

- Only execute expensive operations occasionally, e.g. Physics.Raycast()
  `+ Show [Infrequent Calling] +`

- Minimize callstack overhead in inner loops
  `+ Show [Callstack Overhead] +`

#### Optimizing Physics Performance

The NVIDIA PhysX physics engine used by Unity is available on mobiles, but the performance limits of the hardware will be reached more easily on mobile platforms than desktops.

Here are some tips for tuning physics to get better performance on mobiles:-

- You can adjust the **Fixed Timestep** setting (in the Time manager) to reduce the time spent on physics updates. Increasing the timestep will reduce the CPU overhead at the expense of the accuracy of the physics. Often, lower accuracy is an acceptable tradeoff for increased speed.
- Set the **Maximum Allowed Timestep** in the Time manager in the 8-10fps range to cap the time spent on physics in the

worst case scenario.
- Mesh colliders have a much higher performance overhead than primitive colliders, so use them sparingly. It is often possible to approximate the shape of a mesh by using child objects with primitive colliders. The child colliders will be controlled collectively as a single compound collider by the rigidbody on the parent.
- While wheel colliders are not strictly colliders in the sense of solid objects, they nonetheless have a high CPU overhead.

Page last updated: 2012-08-24

---

# Optimizing Graphics Performance

Good performance is critical to the success of many games. Below are some simple guidelines for maximizing the speed of your game's graphical rendering.

## Where are the graphics costs

The graphical parts of your game can primarily cost on two systems of the computer: the GPU or the CPU. The first rule of any optimization is to find **where the performance problem is**; because strategies for optimizing for GPU vs. CPU are quite different (and can even be opposite - it's quite common to make GPU do more work while optimizing for CPU, and vice versa).

Typical bottlenecks and ways to check for them:

- GPU is often limited by **fillrate** or memory bandwidth.
  - Does running the game at lower display resolution make it faster? If so, you're most likely limited by fillrate on the GPU.
- CPU is often limited by the number of things that need to be rendered, also known as "**draw calls**".
  - Check "draw calls" in Rendering Statistics window; if it's more than several thousand (for PCs) or several hundred (for mobile), then you might want to optimize the object count.

Of course, these are only the rules of thumb; the bottleneck could as well be somewhere else. Less typical bottlenecks:

- Rendering is not a problem, neither on the GPU nor the CPU! For example, your scripts or physics might be the actual problem. Use Profiler to figure this out.
- GPU has too many vertices to process. How many vertices are "ok" depends on the GPU and the complexity of vertex shaders. Typical figures are "not more than 100 thousand" on mobile, and "not more than several million" on PC.
- CPU has too many vertices to process, for things that do vertex processing on the CPU. This could be skinned meshes, cloth simulation, particles etc.

## CPU optimization - draw call count

In order to render any object on the screen, the CPU has some work to do - things like figuring out which lights affect that object, setting up the shader & shader parameters, sending drawing commands to the graphics driver, which then prepares the commands to be sent off to the graphics card. All this "per object" CPU cost is not very cheap, so if you have lots of visible objects, it can add up.

So for example, if you have a thousand triangles, it will be much, much cheaper if they are all in one mesh, instead of having a thousand individual meshes one triangle each. The cost of both scenarios on the GPU will be very similar, but the work done by the CPU to render a thousand objects (instead of one) will be significant.

In order to make CPU do less work, it's good to reduce the visible object count:

- Combine close objects together, either manually or using Unity's draw call batching.
- Use less materials in your objects, by putting separate textures into a larger texture atlas and so on.
- Use less things that cause objects to be rendered multiple times (reflections, shadows, per-pixel lights etc., see below).

Combine objects together so that each mesh has at least several hundred triangles and uses only one **Material** for the entire mesh. It is important to understand that combining two objects which don't share a material does not give you any performance increase at all. The most common reason for having multiple materials is that two meshes don't share the same textures, so to optimize CPU performance, you should ensure that any objects you combine share the same textures.

However, when using many pixel lights in the Forward rendering path, there are situations where combining objects may not make sense, as explained below.

### GPU: Optimizing Model Geometry

When optimizing the geometry of a model, there are two basic rules:

- Don't use any more triangles than necessary
- Try to keep the number of UV mapping seams and hard edges (doubled-up vertices) as low as possible

Note that the actual number of vertices that graphics hardware has to process is usually not the same as the number reported by a 3D application. Modeling applications usually display the geometric vertex count, i.e. the number of distinct corner points that make up a model. For a graphics card, however, some geometric vertices will need to be split into two or more logical vertices for rendering purposes. A vertex must be split if it has multiple normals, UV coordinates or vertex colors. Consequently, the vertex count in Unity is invariably higher than the count given by the 3D application.

While the amount of geometry in the models is mostly relevant for the GPU, some features in Unity also process models on the CPU, for example mesh skinning.

### Lighting Performance

Lighting which is not computed at all is always the fastest! Use Lightmapping to "bake" static lighting just once, instead of computing it each frame. The process of generating a lightmapped environment takes only a little longer than just placing a light in the scene in Unity, **but**:

- It is going to run a lot faster (2-3 times for 2 per-pixel lights)
- And it will look a lot better since you can bake global illumination and the lightmapper can smooth the results

In a lot of cases there can be simple tricks possible in shaders and content, instead of adding more lights all over the place. For example, instead of adding a light that shines straight into the camera to get "rim lighting" effect, consider adding a dedicated "rim lighting" computation into your shaders directly.

#### Lights in forward rendering

Per-pixel dynamic lighting will add significant rendering overhead to every affected pixel and can lead to objects being rendered in multiple passes. On less powerful devices, like mobile or low-end PC GPUs, avoid having more than one **Pixel Light** illuminating any single object, and use lightmaps to light static objects instead of having their lighting calculated every frame. Per-vertex dynamic lighting can add significant cost to vertex transformations. Try to avoid situations where multiple lights illuminate any given object.

If you use pixel lighting then each mesh has to be rendered as many times as there are pixel lights illuminating it. If you combine two meshes that are very far apart, it will increase the effective size of the combined object. All pixel lights that illuminate any part of this combined object will be taken into account during rendering, so the number of rendering passes that need to be made could be increased. Generally, the number of passes that must be made to render the combined object is the sum of the number of passes for each of the separate objects, and so nothing is gained by combining. For this reason, you should not combine meshes that are far enough apart to be affected by different sets of pixel lights.

During rendering, Unity finds all lights surrounding a mesh and calculates which of those lights affect it most. The Quality Settings are used to modify how many of the lights end up as pixel lights and how many as vertex lights. Each light calculates its importance based on how far away it is from the mesh and how intense its illumination is. Furthermore, some lights are more important than others purely from the game context. For this reason, every light has a **Render Mode** setting which can be set to **Important** or **Not Important**; lights marked as **Not Important** will typically have a lower rendering overhead.

As an example, consider a driving game where the player's car is driving in the dark with headlights switched on. The headlights are likely to be the most visually significant light sources in the game, so their Render Mode would probably be set to **Important**. On the other hand, there may be other lights in the game that are less important (other cars' rear lights, say) and which don't improve the visual effect much by being pixel lights. The Render Mode for such lights can safely be set to **Not Important** so as to avoid wasting rendering capacity in places where it will give little benefit.

Optimizing per-pixel lighting saves both CPU and the GPU: the CPU has less draw calls to do, and the GPU has less vertices to process and pixels to rasterize for all these additional object renders.

### GPU: Texture Compression and Mipmaps

Using Compressed Textures will decrease the size of your textures (resulting in faster load times and smaller memory footprint) and can also dramatically increase rendering performance. Compressed textures use only a fraction of the memory bandwidth needed for uncompressed 32bit RGBA textures.

#### Use Texture Mip Maps

As a rule of thumb, always have Generate Mip Maps enabled for textures used in a 3D scene. In the same way Texture Compression can help limit the amount of texture data transfered when the GPU is rendering, a mip mapped texture will enable the GPU to use a lower-resolution texture for smaller triangles.

The only exception to this rule is when a texel (texture pixel) is known to map 1:1 to the rendered screen pixel, as with UI elements or in a 2D game.

### LOD and Per-Layer Cull Distances

In some games, it may be appropriate to cull small objects more aggressively than large ones, in order to reduce both the CPU and GPU load. For example, small rocks and debris could be made invisible at long distances while large buildings would still be visible.

This can be either achieved by Level Of Detail system, or by setting manual per-layer culling distances on the camera. You could put small objects into a separate layer and setup per-layer cull distances using the Camera.layerCullDistances script function.

### Realtime Shadows

Realtime shadows are nice, but they can cost quite a lot of performance, both in terms of extra draw calls for the CPU, and extra processing on the GPU. For further details, see the Shadows page.

### GPU: Tips for writing high-performance shaders

A high-end PC GPU and a low-end mobile GPU can be literally hundreds of times performance difference apart. Same is true even on a single platform. On a PC, a fast GPU is dozens of times faster than a slow integrated GPU; and on mobile platforms you can see just as large difference in GPUs.

So keep in mind that GPU performance on mobile platforms and low-end PCs will be much lower than on your development machines. Typically, shaders will need to be hand optimized to reduce calculations and texture reads in order to get good performance. For example, some built-in Unity shaders have their "mobile" equivalents that are much faster (but have some limitations or approximations - that's what makes them faster).

Below are some guidelines that are most important for mobile and low-end PC graphics cards:

#### Complex mathematical operations

Transcendental mathematical functions (such as **pow**, **exp**, **log**, **cos**, **sin**, **tan**, etc) are quite expensive, so a good rule of thumb is to have no more than one such operation per pixel. Consider using lookup textures as an alternative where applicable.

It is not advisable to attempt to write your own **normalize**, **dot**, **inversesqrt** operations, however. If you use the built-in ones then the driver will generate much better code for you.

Keep in mind that alpha test (**discard**) operation will make your fragments slower.

#### Floating point operations

You should always specify the precision of floating point variables when writing custom shaders. It is **critical** to pick the smallest possible floating point format in order to get the best performance. Precision of operations is completely ignored on many desktop GPUs, but is critical for performance on many mobile GPUs.

If the shader is written in Cg/HLSL then precision is specified as follows:

- **float** - full 32-bit floating point format, suitable for vertex transformations but has the slowest performance.
- **half** - reduced 16-bit floating point format, suitable for texture UV coordinates and roughly twice as fast as **highp**.
- **fixed** - 10-bit fixed point format, suitable for colors, lighting calculation and other high-performance operations and roughly four times faster than **highp**.

If the shader is written in GLSL ES then the floating point precision is specified specified as **highp**, **mediump**, **lowp** respectively.

For further details about shader performance, please read the Shader Performance page.

### Simple Checklist to make Your Game Faster

- Keep vertex count below 200K..3M per frame when targetting PCs, depending on the target GPU
- If you're using built-in shaders, pick ones from Mobile or Unlit category. They work on non-mobile platforms as well; but are simplified and approximated versions of the more complex shaders.
- Keep the number of different materials per scene low - share as many materials between different objects as possible.
- Set **Static** property on a non-moving objects to allow internal optimizations like static batching.
- Do not use **Pixel Lights** when it is not necessary - choose to have only a single (preferably directional) pixel light affecting your geometry.
- Do not use dynamic lights when it is not necessary - choose to bake lighting instead.
- Use compressed texture formats when possible, otherwise prefer 16bit textures over 32bit.
- Do not use fog when it is not necessary.
- Learn benefits of Occlusion Culling and use it to reduce amount of visible geometry and draw-calls in case of complex static scenes with lots of occlusion. Plan your levels to benefit from ccclusion culling.
- Use skyboxes to "fake" distant geometry.
- Use pixel shaders or texture combiners to mix several textures instead of a multi-pass approach.
- If writing custom shaders, always use smallest possible floating point format:
  - **fixed** / **lowp** - for colors, lighting information and normals,
  - **half** / **mediump** - for texture UV coordinates,
  - **float** / **highp** - avoid in pixel shaders, fine to use in vertex shader for position calculations.
- Minimize use of complex mathematical operations such as **pow**, **sin**, **cos** etc. in pixel shaders.
- Choose to use less textures per fragment.

### See Also

- Draw Call Batching
- Modeling Characters for Optimal Performance
- Rendering Statistics Window

Page last updated: 2012-07-29

---

## Draw Call Batching

To draw an object on the screen, the engine has to issue a draw call to the graphics API (e.g. OpenGL or Direct3D). Every single draw call requires a significant amount of work on the part of the graphics API, causing significant performance overhead on the CPU side.

Unity combines a number of objects at runtime and draws them together with a single draw call. This operation is called "batching". The more objects Unity can batch together, the better rendering performance you will get.

Built-in batching support in Unity has significant benefit over simply combining geometry in the modeling tool (or using the **CombineChildren** script from the Standard Assets package). Batching in Unity happens **after** visibility determination step. The engine does culling on each object individually, and the amount of rendered geometry is going to be the same as without batching. Combining geometry in the modeling tool, on the other hand, prevents effecient culling and results in much higher amount of geometry being rendered.

### Materials

Only objects sharing the same material can be batched together. Therefore, if you want to achieve good batching, you need to share as many materials among different objects as possible.

If you have two identical materials which differ only in textures, you can combine those textures into a single big texture - a process often called *texture atlasing*. Once textures are in the same atlas, you can use single material instead.

If you need to access shared material properties from the scripts, then it is important to note that modifying Renderer.material will create a copy of the material. Instead, you should use Renderer.sharedMaterial to keep material shared.

### Dynamic Batching

Unity can automatically batch moving objects into the same draw call if they share the same material.

Dynamic batching is done automatically and does not require any additional effort on your side.

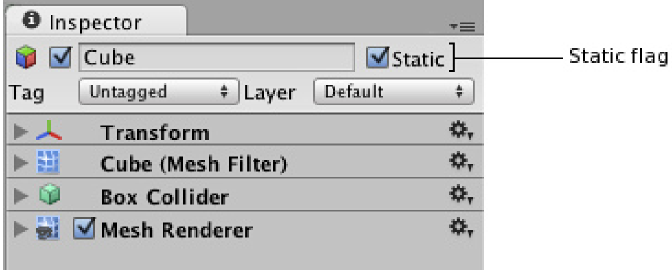
Tips:

- Batching dynamic objects has certain overhead **per vertex**, so batching is applied only to meshes containing less than **900** vertex attributes in total.
  - If your shader is using Vertex Position, Normal and single UV, then you can batch up to 300 verts and if your shader is using Vertex Position, Normal, UV0, UV1 and Tangent, then only 180 verts.
  - **Please note: attribute count limit might be changed in future**
- Don't use scale. Objects with scale (1,1,1) and (2,2,2) won't batch.
- Uniformly scaled objects won't be batched with non-uniformly scaled ones.
  - Objects with scale (1,1,1) and (1,2,1) won't be batched. On the other hand (1,2,1) and (1,3,1) will be.
- Using different material instances will cause batching to fail.
- Objects with lightmaps have additional (hidden) material parameter: offset/scale in lightmap, so lightmapped objects won't be batched (unless they point to same portions of lightmap)
- Multi-pass shaders will break batching. E.g. Almost all unity shaders supports several lights in forward rendering, effectively doing additional pass for them
- Using instances of a prefab automatically are using the same mesh and material.

### Static Batching

Static batching, on the other hand, allows the engine to reduce draw calls for geometry of any size (provided it does not move and shares the same material). Static batching is significantly more efficient than dynamic batching. You should choose static batching as it will require less CPU power.

In order to take advantage of static batching, you need explicitly specify that certain objects are static and will **not** move, rotate or scale in the game. To do so, you can mark objects as static using the Static checkbox in the Inspector:




Using static batching will require additional memory for storing the combined geometry. If several objects shared the same geometry before static batching, then a copy of geometry will be created for each object, either in the Editor or at runtime. This might not always be a good idea - sometimes you will have to sacrifice rendering performance by avoiding static batching for some objects to keep a smaller memory footprint. For example, marking trees as static in a dense forest level can have serious memory impact.

Static batching is only available in Unity Pro for each platform.

Page last updated: 2012-10-22

---

# Modeling Optimized Characters

Below are some tips for designing character models to give optimal rendering speed.

### Use a Single Skinned Mesh Renderer

You should use only a single skinned mesh renderer for each character. Unity optimizes animation using visibility culling and bounding volume updates and these optimizations are only activated if you use one animation component and one skinned mesh renderer in conjunction. The rendering time for a model could roughly double as a result of using two skinned meshes in place of a single mesh and there is seldom any practical advantage in using multiple meshes.

### Use as Few Materials as Possible

You should also keep the number of materials on each mesh as low as possible. The only reason why you might want to have more than one material on a character is that you need to use different shaders for different parts (eg, a special shader for the eyes). However, two or three materials per character should be sufficient in almost all cases.

### Use as Few Bones as Possible

A bone hierarchy in a typical desktop game uses somewhere between fifteen and sixty bones. The fewer bones you use, the better the performance will be. You can achieve very good quality on desktop platforms and fairly good quality on mobile platforms with about thirty bones. Ideally, keep the number below thirty for mobile devices and don't go too far above thirty for desktop games.

### Polygon Count

The number of polygons you should use depends on the quality you require and the platform you are targeting. For mobile devices, somewhere between 300 and 1500 polygons per mesh will give good results, whereas for desktop platforms the ideal range is about 1500 to 4000. You may need to reduce the polygon count per mesh if the game can have lots of characters onscreen at any given time. As an example, Half Life 2 used 2500-5000 triangles per character. Current AAA games running on the PS3 or Xbox 360 usually have characters with 5000-7000 triangles.

### Keep Forward and Inverse Kinematics Separate

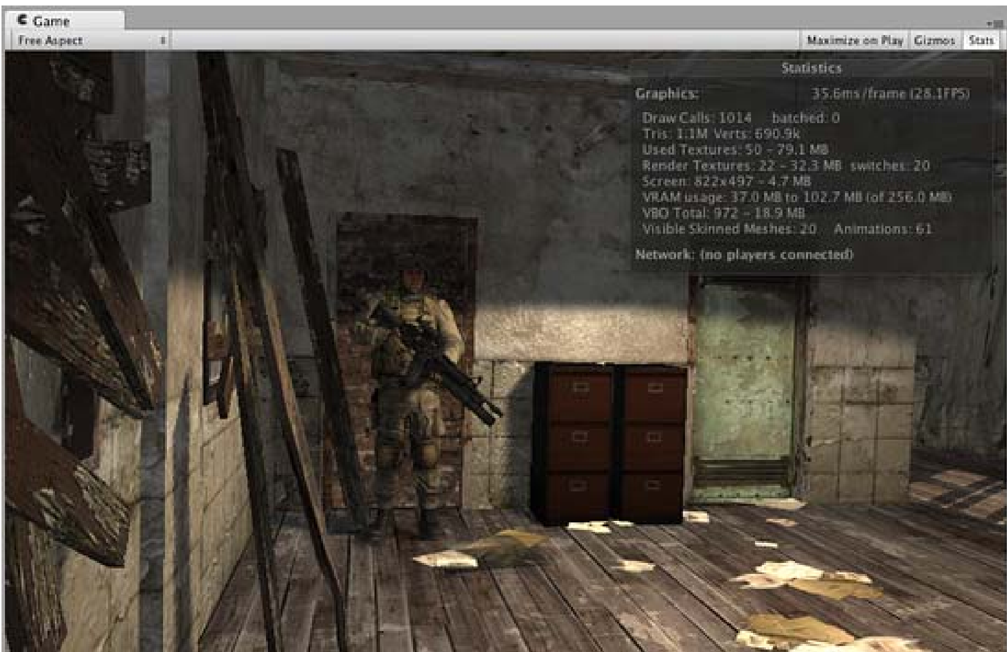
When animations are imported, a model's inverse kinematic (IK) nodes are baked into forward kinematics (FK) and as a result, Unity doesn't need the IK nodes at all. However, if they are left in the model then they will have a CPU overhead even though they don't affect the animation. You can delete the redundant IK nodes in Unity or in the modeling tool, according to your preference. Ideally, you should keep separate IK and FK hierarchies during modeling to make it easier to remove the IK nodes when necessary.

Page last updated: 2011-11-04

---

# RenderingStatistics

The **Game View** has a **Stats** button in the top right corner. When the button is pressed, an overlay window is displayed which shows realtime rendering statistics, which are useful for optimizing performance. The exact statistics displayed vary according to the build target.




*Rendering Statistics Window.*

The Statistics window contains the following information:-

| | |
|---|---|
| **Time per frame and FPS** | The amount of time taken to process and render one game frame (and its reciprocal, frames per second). Note that this number only includes the time taken to do the frame update and render the game view; it does not include the time taken in the editor to draw the scene view, inspector and other editor-only processing. |
| **Draw Calls** | The total number of meshes drawn after batching was applied. Note that where objects are rendered multiple times (for example, objects illuminated by pixel lights), each rendering results in a separate draw call. |

| | |
|---|---|
| **Batched (Draw Calls)** | The number of initially separate draw calls that were added to batches. "Batching" is where the engine attempts to combine the rendering of multiple objects into one draw call in order to reduce CPU overhead. To ensure good batching, you should share materials between different objects as often as possible. |
| **Tris** and **Verts** | The number of triangles and vertices drawn. This is mostly important when optimizing for low-end hardware |
| **Used Textures** | The number of textures used to draw this frame and their memory usage. |
| **Render Textures** | The number of Render Textures and their memory usage. The number of times the active Render Texture was switched each frame is also displayed. |
| **Screen** | The size of the screen, along with its anti-aliasing level and memory usage. |
| **VRAM usage** | Approximate bounds of current video memory (VRAM) usage. This also shows how much video memory your graphics card has. |
| **VBO total** | The number of unique meshes (Vertex Buffers Objects or VBOs) that are uploaded to the graphics card. Each different model will cause a new VBO to be created. In some cases scaled objects will cause additional VBOs to be created. In the case of a static batching, several different objects can potentially share the same VBO. |
| **Visible Skinned Meshes** | The number of skinned meshes rendered. |
| **Animations** | The number of animations playing. |

Page last updated: 2012-01-18

---

# Reducing File size

### Unity post-processes all imported assets

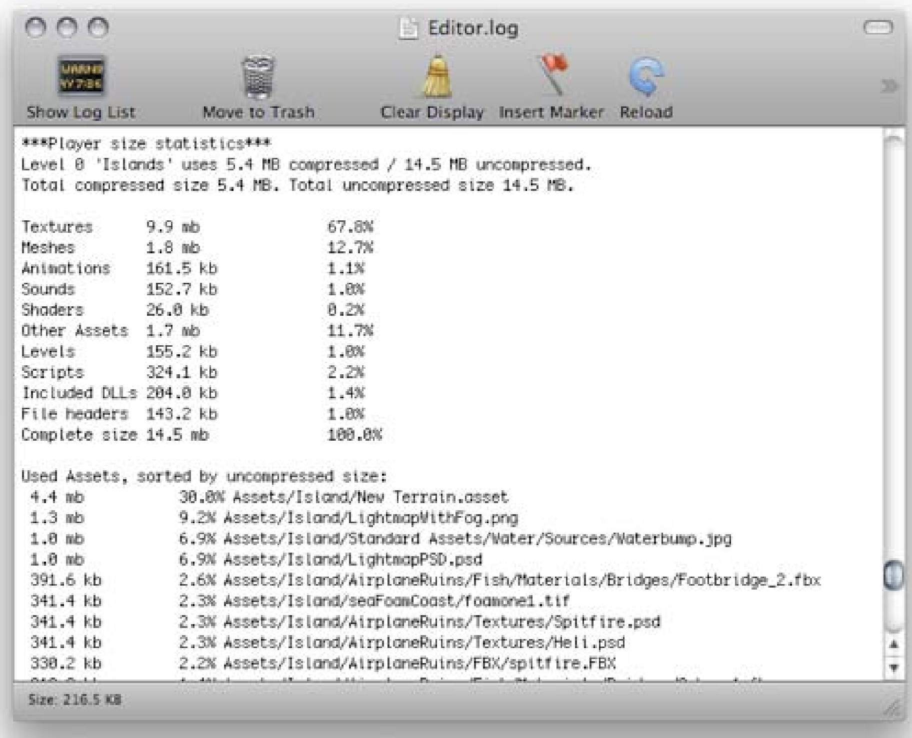
Unity always post-processes imported files, thus storing a file as a multi-layered psd file instead of a jpg will make absolutely zero difference in the size of the player you will deploy. Save your files in the format you are working with (eg. .mb files, .psd files, .tiff files) to make your life easier.

### Unity strips out unused assets

The amount of assets in your project folder does **not** influence the size of your built player. Unity is very smart about detecting which assets are used in your game and which are not. Unity follows all references to assets before building a game and generates a list of assets that need to be included in the game. Thus you can safely keep unused assets in your project folder.

### Unity prints an overview of the used file size

After Unity has completed building a player, it prints an overview of what type of asset took up the most file size, and it prints which assets were included in the build. To see it just open the editor console log: **Open Editor Log** button in the Console window (**Window -> Console**).

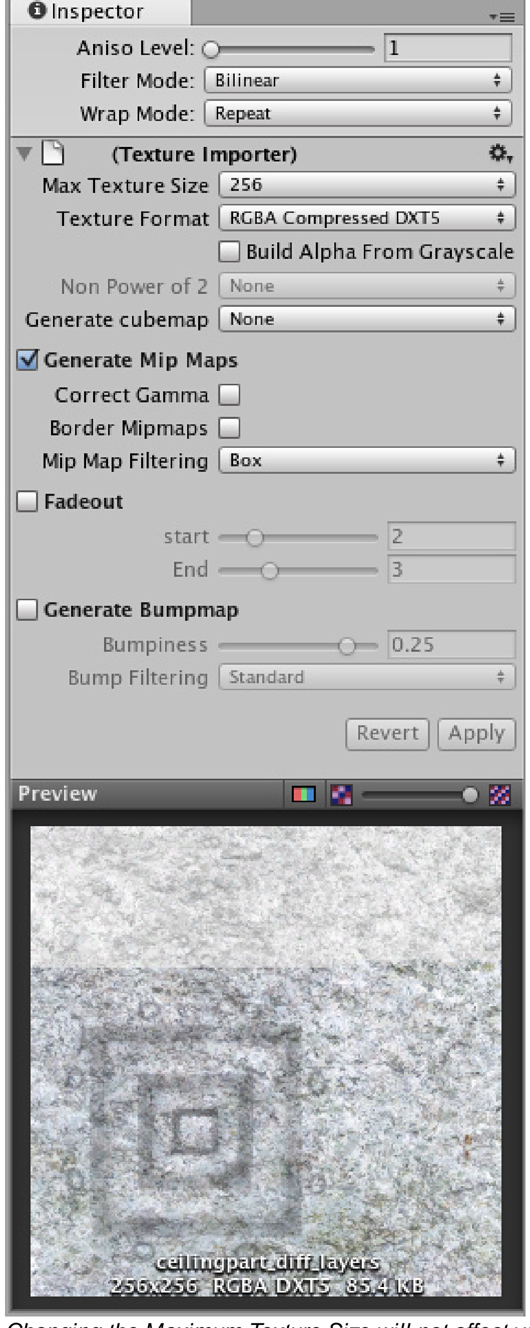



*An overview of what took up space*

### Optimizing texture size

Often textures take up most space in the build. The first to do is to use compressed texture formats (DXT(**Desktop platforms**) or PVRTC) where you can.

If that doesn't get the size down, try to reduce the size of the textures. The trick here is that you don't need to modfiy the actual source content. Simply select the texture in the Project view and set **Max Texture Size** in Import Settings. It is a good idea to zoom in on an object that uses the texture, then adjust the **Max Texture Size** until it starts looking worse in the **Scene View**.



*Changing the Maximum Texture Size will not affect your texture asset, just its resolution in the game*

**How much memory does my texture take up?**

## Desktop

| **Compression** | **Memory consumption** |
|---|---|
| **RGB Compressed DXT1** | 0.5 bpp (bytes/pixel) |
| **RGBA Compressed DXT5** | 1 bpp |
| **RGB 16bit** | 2 bpp |
| **RGB 24bit** | 3 bpp |
| **Alpha 8bit** | 1 bpp |
| **RGBA 16bit** | 2 bpp |
| **RGBA 32bit** | 4 bpp |

## iOS

| **Compression** | **Memory consumption** |
|---|---|
| **RGB Compressed PVRTC 2 bits** | 0.25 bpp (bytes/pixel) |

| | |
|---|---|
| **RGBA Compressed PVRTC 2 bits** | 0.25 bpp |
| **RGB Compressed PVRTC 4 bits** | 0.5 bpp |
| **RGBA Compressed PVRTC 4 bits** | 0.5 bpp |
| **RGB 16bit** | 2 bpp |
| **RGB 24bit** | 3 bpp |
| **Alpha 8bit** | 1 bpp |
| **RGBA 16bit** | 2 bpp |
| **RGBA 32bit** | 4 bpp |

## Android

| Compression | Memory consumption |
|---|---|
| **RGB Compressed DXT1** | 0.5 bpp (bytes/pixel) |
| **RGBA Compressed DXT5** | 1 bpp |
| **RGB Compressed ETC1** | 0.5 bpp |
| **RGB Compressed PVRTC 2 bits** | 0.25 bpp (bytes/pixel) |
| **RGBA Compressed PVRTC 2 bits** | 0.25 bpp |
| **RGB Compressed PVRTC 4 bits** | 0.5 bpp |
| **RGBA Compressed PVRTC 4 bits** | 0.5 bpp |
| **RGB 16bit** | 2 bpp |
| **RGB 24bit** | 3 bpp |
| **Alpha 8bit** | 1 bpp |
| **RGBA 16bit** | 2 bpp |
| **RGBA 32bit** | 4 bpp |

To figure out total texture size: width * height * bpp. Add 33% if you have Mipmaps.

By default Unity compresses all textures when importing. This can be turned off in the **Preferences** for faster workflow. But when building a game, all not-yet-compressed textures will be compressed.

### Optimizing mesh and animation size

Meshes and imported Animation Clips can be compressed so they take up less space in your game file. Compression can be turned on in Mesh Import Settings.

Mesh and Animation compression uses quantization, which means it takes less space but the compression can introduce some inaccuracies. Experiment with what level of compression is still acceptable for your models.

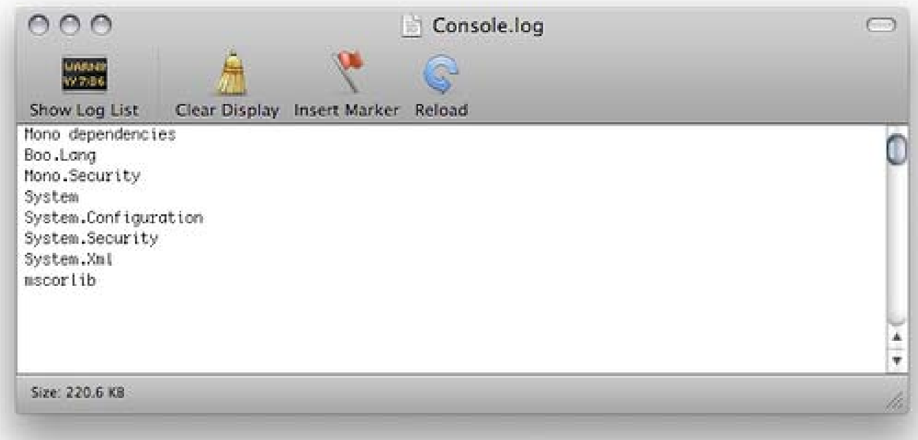
Note that mesh compression only produces smaller data files, and does not use less memory at run time. Animation **Keyframe reduction** produces smaller data files *and* uses less memory at run time, and generally you should always use keyframe reduction.

Additionally, you can choose not to store normals and/or tangents in your Meshes, to save space both in the game builds and memory at run time. This can be set in **Tangent Space Generation** drop down in Mesh Import Settings. Rules of thumb:

- Tangents are used for normal-mapping. If you don't use normal-mapping, you probably don't need to store tangents in those meshes.
- Normals are used for lighting. If you don't use realtime lighting on some of your meshes, you probably don't need to store normals in them.

### Reducing included dlls in the Players

When building a player (Desktop, Android or iOS) it is important to not depend on **System.dll** or **System.Xml.dll**. Unity does not include **System.dll** or **System.Xml.dll** in the players installation. That means, if you want to use Xml or some Generic containers which live in **System.dll** then the required dlls will be included in the players. This usually adds 1mb to the download size, obviously this is not very good for the distribution of your players and you should really avoid it. If you need to parse some Xml files, you can use a smaller xml library like this one Mono.Xml.zip. While most Generic containers are contained in mscorlib, Stack<> and few others are in **System.dll**. So you really want to avoid those.




*As you can see, Unity is including System.Xml.dll and System.dll, when building a player*

Unity includes the following DLLs with the players distribution **mscorlib.dll**, **Boo.Lang.dll**, **UnityScript.Lang.dll** and **UnityEngine.dll**.

Page last updated: 2012-07-26

---

# Understanding Automatic Memory Management

When an object, string or array is created, the memory required to store it is allocated from a central pool called the **heap**. When the item is no longer in use, the memory it once occupied can be reclaimed and used for something else. In the past, it was typically up to the programmer to allocate and release these blocks of heap memory explicitly with the appropriate function calls. Nowadays, runtime systems like Unity's Mono engine manage memory for you automatically. Automatic memory management requires less coding effort than explicit allocation/release and greatly reduces the potential for memory leakage (the situation where memory is allocated but never subsequently released).

## Value and Reference Types

When a function is called, the values of its parameters are copied to an area of memory reserved for that specific call. Data types that occupy only a few bytes can be copied very quickly and easily. However, it is common for objects, strings and arrays to be much larger and it would be very inefficient if these types of data were copied on a regular basis. Fortunately, this is not necessary; the actual storage space for a large item is allocated from the heap and a small "pointer" value is used to remember its location. From then on, only the pointer need be copied during parameter passing. As long as the runtime system can locate the item identified by the pointer, a single copy of the data can be used as often as necessary.

Types that are stored directly and copied during parameter passing are called value types. These include integers, floats, booleans and Unity's struct types (eg, **Color** and **Vector3**). Types that are allocated on the heap and then accessed via a pointer are called reference types, since the value stored in the variable merely "refers" to the real data. Examples of reference types include objects, strings and arrays.

## Allocation and Garbage Collection

The memory manager keeps track of areas in the heap that it knows to be unused. When a new block of memory is requested (say when an object is instantiated), the manager chooses an unused area from which to allocate the block and then removes the allocated memory from the known unused space. Subsequent requests are handled the same way until there is no free area large enough to allocate the required block size. It is highly unlikely at this point that all the memory allocated from the heap is still in use. A reference item on the heap can only be accessed as long as there are still reference variables that can locate it. If all references to a memory block are gone (ie, the reference variables have been reassigned or they are local variables that are now out of scope) then the memory it occupies can safely be reallocated.

To determine which heap blocks are no longer in use, the memory manager searches through all currently active reference variables and marks the blocks they refer to as "live". At the end of the search, any space between the live blocks is considered empty by the memory manager and can be used for subsequent allocations. For obvious reasons, the process of locating and freeing up unused memory is known as garbage collection (or GC for short).

### Optimization

Garbage collection is automatic and invisible to the programmer but the collection process actually requires significant CPU time behind the scenes. When used correctly, automatic memory management will generally equal or beat manual allocation for overall performance. However, it is important for the programmer to avoid mistakes that will trigger the collector more often than necessary and introduce pauses in execution.

There are some infamous algorithms that can be GC nightmares even though they seem innocent at first sight. Repeated string concatenation is a classic example:-

```
function ConcatExample(intArray: int[]) {
	var line = intArray[0].ToString();

	for (i = 1; i < intArray.Length; i++) {
		line += ", " + intArray[i].ToString();
	}

	return line;
}
```

The key detail here is that the new pieces don't get added to the string in place, one by one. What actually happens is that each time around the loop, the previous contents of the line variable become dead - a whole new string is allocated to contain the original piece plus the new part at the end. Since the string gets longer with increasing values of i, the amount of heap space being consumed also increases and so it is easy to use up hundreds of bytes of free heap space each time this function is called. If you need to concatenate many strings together then a much better option is the Mono library's System.Text.StringBuilder class.

However, even repeated concatenation won't cause too much trouble unless it is called frequently, and in Unity that usually implies the frame update. Something like:-

```
var scoreBoard: GUIText;
var score: int;

function Update() {
	var scoreText: String = "Score: " + score.ToString();
	scoreBoard.text = scoreText;
}
```

...will allocate new strings each time Update is called and generate a constant trickle of new garbage. Most of that can be saved by updating the text only when the score changes:-

```
var scoreBoard: GUIText;
var scoreText: String;
var score: int;
var oldScore: int;

function Update() {
	if (score != oldScore) {
		scoreText = "Score: " + score.ToString();
		scoreBoard.text = scoreText;
		oldScore = score;
	}
}
```

Another potential problem occurs when a function returns an array value:-

```
function RandomList(numElements: int) {
	var result = new float[numElements];

	for (i = 0; i < numElements; i++) {
		result[i] = Random.value;
	}

	return result;
}
```

This type of function is very elegant and convenient when creating a new array filled with values. However, if it is called repeatedly then fresh memory will be allocated each time. Since arrays can be very large, the free heap space could get used up rapidly, resulting in frequent garbage collections. One way to avoid this problem is to make use of the fact that an array is a reference type. An array passed into a function as a parameter can be modified within that function and the results will remain after the function returns. A function like the one above can often be replaced with something like:-

```
function RandomList(arrayToFill: float[]) {
	for (i = 0; i < arrayToFill.Length; i++) {
		arrayToFill[i] = Random.value;
	}
}
```

This simply replaces the existing contents of the array with new values. Although this requires the initial allocation of the array to be done in the calling code (which looks slightly inelegant), the function will not generate any new garbage when it is called.

### Requesting a Collection

As mentioned above, it is best to avoid allocations as far as possible. However, given that they can't be completely eliminated, there are two main strategies you can use to minimise their intrusion into gameplay:-

**Small heap with fast and frequent garbage collection**

This strategy is often best for games that have long periods of gameplay where a smooth framerate is the main concern. A game like this will typically allocate small blocks frequently but these blocks will be in use only briefly. The typical heap size when using this strategy on iOS is about 200KB and garbage collection will take about 5ms on an iPhone 3G. If the heap increases to 1MB, the collection will take about 7ms. It can therefore be advantageous sometimes to request a garbage collection at a regular frame interval. This will generally make collections happen more often than strictly necessary but they will be processed quickly and with minimal effect on gameplay:-

```
if (Time.frameCount % 30 == 0)
{
   System.GC.Collect();
}
```

However, you should use this technique with caution and check the profiler statistics to make sure that it is really reducing collection time for your game.

**Large heap with slow but infrequent garbage collection**

This strategy works best for games where allocations (and therefore collections) are relatively infrequent and can be handled during pauses in gameplay. It is useful for the heap to be as large as possible without being so large as to get your app killed by the OS due to low system memory. However, the Mono runtime avoids expanding the heap automatically if at all possible. You can expand the heap manually by preallocating some placeholder space during startup (ie, you instantiate a "useless" object that is allocated purely for its effect on the memory manager):-

```
function Start() {
	var tmp = new System.Object[1024];
```

```
        // make allocations in smaller blocks to avoid them to be treated in a special way, whi
         for (var i : int = 0; i < 1024; i++)
                tmp[i] = new byte[1024];

        // release reference
         tmp = null;
}
```

A sufficiently large heap should not get completely filled up between those pauses in gameplay that would accommodate a collection. When such a pause occurs, you can request a collection explicitly:-

```
System.GC.Collect();
```

Again, you should take care when using this strategy and pay attention to the profiler statistics rather than just assuming it is having the desired effect.

### Reusable Object Pools

There are many cases where you can avoid generating garbage simply by reducing the number of objects that get created and destroyed. There are certain types of objects in games, such as projectiles, which may be encountered over and over again even though only a small number will ever be in play at once. In cases like this, it is often possible to reuse objects rather than destroy old ones and replace them with new ones.

See here for more information on Object Pools and their implementation.

### Further Information

Memory management is a subtle and complex subject to which a great deal of academic effort has been devoted. If you are interested in learning more about it then memorymanagement.org is an excellent resource, listing many publications and online articles. Further information about object pooling can be found on the Wikipedia page and also at Sourcemaking.com.

Page last updated: 2012-07-30

---

## Platform Dependent Compilation

Unity includes a feature named "Platform Dependent Compilation". This consists of some preprocessor directives that let you *partition* your scripts to compile and execute a section of code exclusively for one of the supported platforms.

Furthermore, you can run this code within the Editor, so you can compile the code specifically for your mobile/console and test it in the Editor!

### Platform Defines

The platform defines that Unity supports for your scripts are:

| | |
|---|---|
| **UNITY_EDITOR** | Define for calling Unity Editor scripts from your game code. |
| **UNITY_STANDALONE_OSX** | Platform define for compiling/executing code specifically for Mac OS (This includes Universal, PPC and Intel architectures). |
| **UNITY_DASHBOARD_WIDGET** | Platform define when creating code for Mac OS dashboard widgets. |
| **UNITY_STANDALONE_WIN** | Use this when you want to compile/execute code for Windows stand alone applications. |
| **UNITY_STANDALONE_LINUX** | Use this when you want to compile/execute code for Linux stand alone applications. |
| **UNITY_WEBPLAYER** | Platform define for web player content (this includes Windows and Mac Web player executables). |
| **UNITY_WII** | Platform define for compiling/executing code for the Wii console. |

| | |
|---|---|
| **UNITY_IPHONE** | Platform define for compiling/executing code for the iPhone platform. |
| **UNITY_ANDROID** | Platform define for the Android platform. |
| **UNITY_PS3** | Platform define for running PlayStation 3 code. |
| **UNITY_XBOX360** | Platform define for executing Xbox 360 code. |
| **UNITY_NACL** | Platform define when compiling code for Google native client (this will be set additionally to **UNITY_WEBPLAYER**). |
| **UNITY_FLASH** | Platform define when compiling code for Adobe Flash. |

Also you can compile code selectively depending on the version of the engine you are working on. Currently the supported ones are:

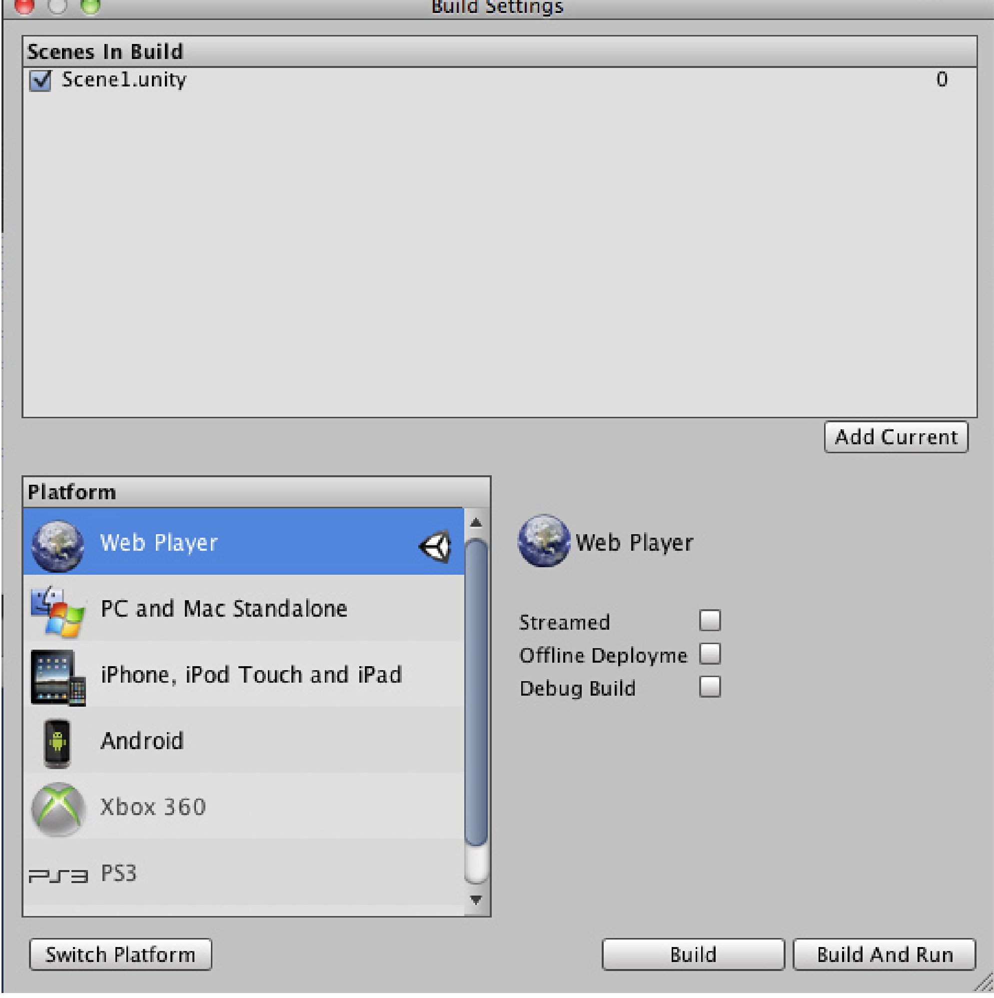
| | |
|---|---|
| **UNITY_2_6** | Platform define for the major version of Unity 2.6. |
| **UNITY_2_6_1** | Platform define for specific version 1 from the major release 2.6. |
| **UNITY_3_0** | Platform define for the major version of Unity 3.0. |
| **UNITY_3_0_0** | Platform define for the specific version 0 of Unity 3.0. |
| **UNITY_3_1** | Platform define for major version of Unity 3.1. |
| **UNITY_3_2** | Platform define for major version of Unity 3.2. |
| **UNITY_3_3** | Platform define for major version of Unity 3.3. |
| **UNITY_3_4** | Platform define for major version of Unity 3.4. |
| **UNITY_3_5** | Platform define for major version of Unity 3.5. |
| **UNITY_4_0** | Platform define for major version of Unity 4.0. |

**Note:** For versions before 2.6.0 there are no platform defines as this feature was first introduced in that version.

### Testing precompiled code.

We are going to show a small example of how to use the precompiled code. This will simply print a message that depends on the platform you have selected to build your target.

First of all, select the platform you want to test your code against by clicking on **File -> Build Settings**. This will bring the build settings window to select your target platform.




*Build Settings window with the WebPlayer Selected as Target platform.*

Select the platform you want to test your precompiled code against and press the **Switch Editor** button to tell Unity which platform you are targeting.

Create a script and copy/paste this code:

JavaScript Example:

```
function Awake() {
  #if UNITY_EDITOR
    Debug.Log("Unity Editor");
  #endif

  #if UNITY_IPHONE
    Debug.Log("Iphone");
  #endif

  #if UNITY_STANDALONE_OSX
    Debug.Log("Stand Alone OSX");
  #endif

  #if UNITY_STANDALONE_WIN
    Debug.Log("Stand Alone Windows");
  #endif
}
```

C# Example:

```
using UnityEngine;
using System.Collections;

public class PlatformDefines : MonoBehaviour {
  void Start () {

    #if UNITY_EDITOR
      Debug.Log("Unity Editor");
    #endif

    #if UNITY_IPHONE
      Debug.Log("Iphone");
    #endif

    #if UNITY_STANDALONE_OSX
        Debug.Log("Stand Alone OSX");
    #endif

    #if UNITY_STANDALONE_WIN
      Debug.Log("Stand Alone Windows");
    #endif

  }
}
```

Boo Example:

```
import UnityEngine

class PlatformDefines (MonoBehaviour):

        def Start ():
                ifdef UNITY_EDITOR:
                        Debug.Log("Unity Editor")

                ifdef UNITY_IPHONE:
                        Debug.Log("IPhone")

                ifdef UNITY_STANDALONE_OSX:
                        Debug.Log("Stand Alone OSX")

                ifdef not UNITY_IPHONE:
                        Debug.Log("not an iPhone")
```

Then, depending on which platform you selected, one of the messages will get printed on the Unity console when you press play.

In addition to the basic *#if* compiler directive, you can also use a multiway test in C# and JavaScript:-

```
#if UNITY_EDITOR
    Debug.Log("Unity Editor");
#elif UNITY_IPHONE
    Debug.Log("Unity iPhone");
#else
    Debug.Log("Any other platform");
#endif
```

However, Boo currently supports only the *ifdef* directive.

Page last updated: 2012-11-27

---

## Generic Functions

Some functions in the script reference (for example, the various GetComponent functions) are listed with a variant that has a letter T or a type name in angle brackets after the function name:-

```
function FuncName.<T>(): T;
```

These are known as generic functions. The significance they have for scripting is that you get to specify the types of parameters and/or the return type when you call the function. In JavaScript, this can be used to get around the limitations of dynamic typing:-

```
// The type is correctly inferred since it is defined in the function call.
var obj = GetComponent.<Rigidbody>();
```

In C#, it can save a lot of keystrokes and casts:-

```
Rigidbody rb = go.GetComponent<Rigidbody>();

// ...as compared with:-

Rigidbody rb = (Rigidbody) go.GetComponent(typeof(Rigidbody));
```

Any function that has a generic variant listed on its script reference page will allow this special calling syntax.

Page last updated: 2011-08-05

---

## Debugging

When creating a game, unplanned and undesired behaviors can (and inevitably will) appear due to errors in scripts or scene setup. Such undesired behaviors are commonly referred to as **bugs**, and the process of fixing them as **debugging**. Unity offers several methods you can use to debug your game. Read about them on the following pages.

- Console
- Debugger
- Log Files
  - Accessing hidden folders

Page last updated: 2010-09-03

---

## Console

Double-clicking an error in the Status Bar or choosing **Window->Console** will bring up the **Console**.

*Console in the editor.*

The Console shows messages, warnings, errors, or debug output from your game. You can define your own messages to be sent to the Console using **Debug.Log()**, **Debug.LogWarning**, or **Debug.LogError()**. You can double-click any message to be taken to the script that caused the message. You also have a number of options on the Console Toolbar.

*Console control toolbar helps your filter your debug output.*

- Pressing **Clear** will remove all current messages from the Console.
- When **Collapse** is enabled, identical messages will only be shown once.
- When **Clear on play** is enabled, all messages will be removed from the Console every time you go into Play mode.
- When **Error Pause** is enabled, **Debug.LogError()** will cause the pause to occur but **Debug.Log()** will not.
- Pressing **Open Player Log** will open the Player Log in a text editor (or using the Console app on Mac if set as the default app for .log files).
- Pressing **Open Editor Log** will open the Editor Log in a text editor (or using the Console app on Mac if set as the default app for .log files).

Page last updated: 2012-09-06

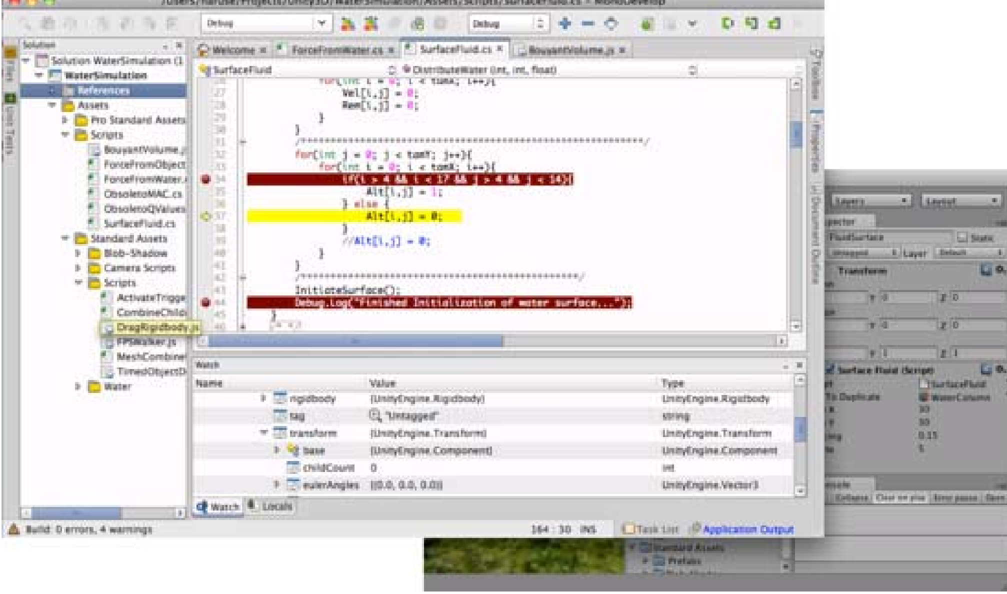
---

## Debugger

The Unity Debugger lets you inspect your code at runtime. For example, it can help you determine when a function is called and with which values. Furthermore, it allows you to look at the value of scripts' variables at a given time while running your game. You can locate bugs or logic problems in your scripts by executing them step by step.

Unity uses the MonoDevelop IDE to debug the scripts in your game. You can debug all the languages supported by the engine (JavaScript, C#, and Boo).

Note that the debugger has to load all your code and all symbols, so bear in mind that this can have a small impact on the performance of your game during execution. Typically, this overhead is not large enough to affect the game framerate.

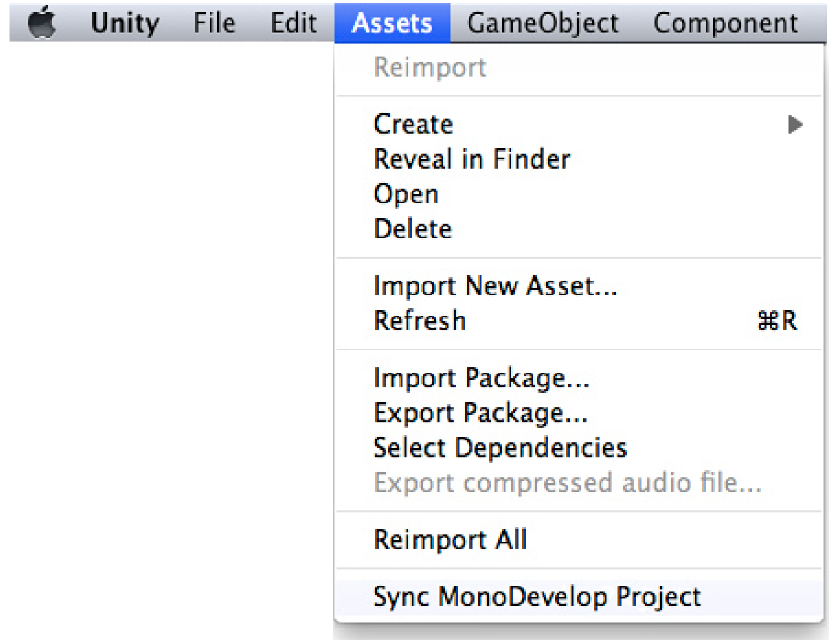


*MonoDevelop window debugging a script in unity.*

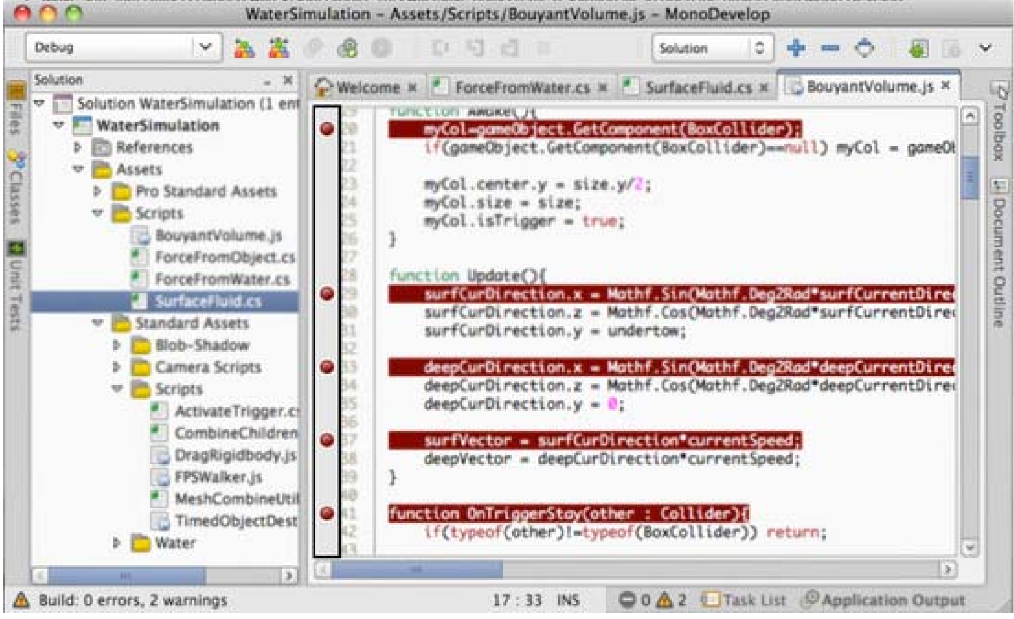
### Debugging in Unity.

On Windows, users must choose to install MonoDevelop as part of the Unity installation (selected by default).

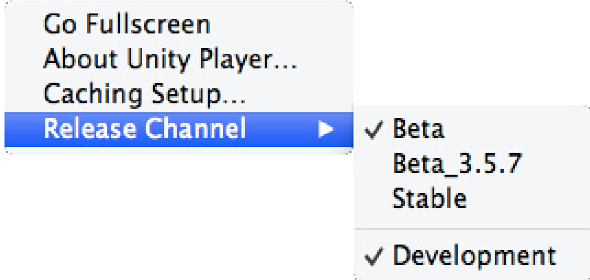
- If you haven't used MonoDevelop with your project before, synchronize your MonoDevelop project. This will open your project inside MonoDevelop.




- Set the necessary breakpoints on your scripts by clicking the lines that you want to analyze.

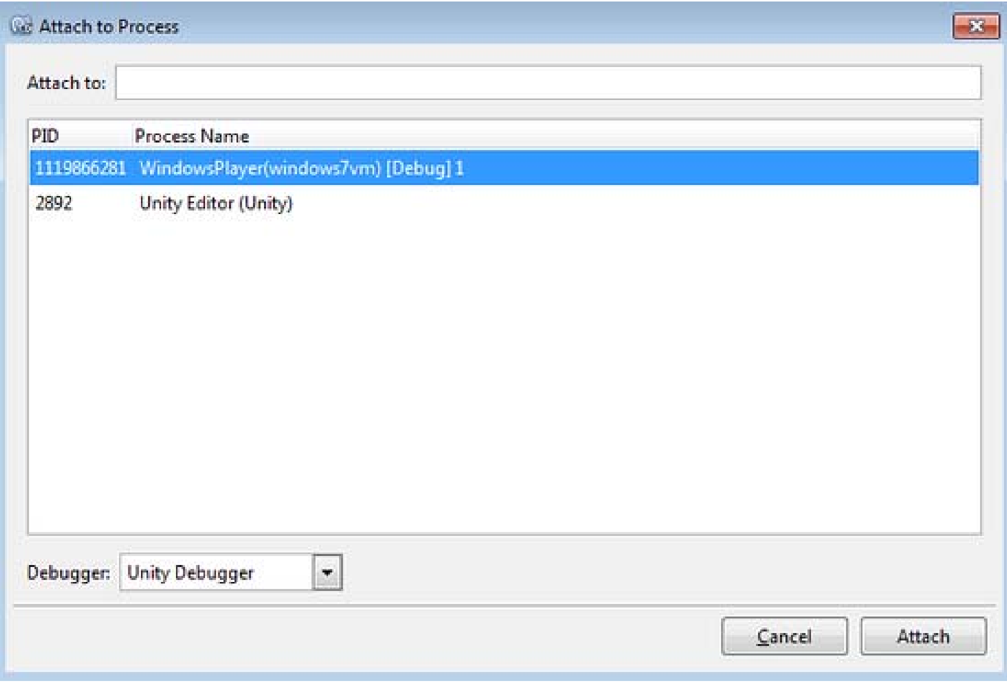



- Launch Unity or your player.
  - Unity: Ensure you have "Editor Attaching" checked in the Preferences window.
  - Players: Ensure that you have built your player with the "Development build" and "Allow script debugging" options enabled. For webplayers, additionally check that the development release channel setting is enabled on the player's context menu (right click on Windows or cmd-click on Mac OSX)

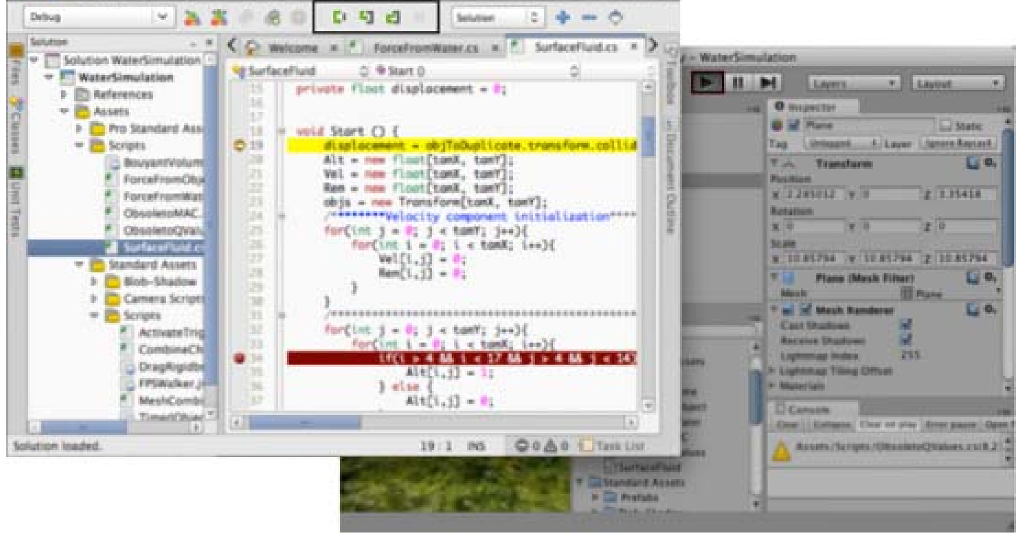


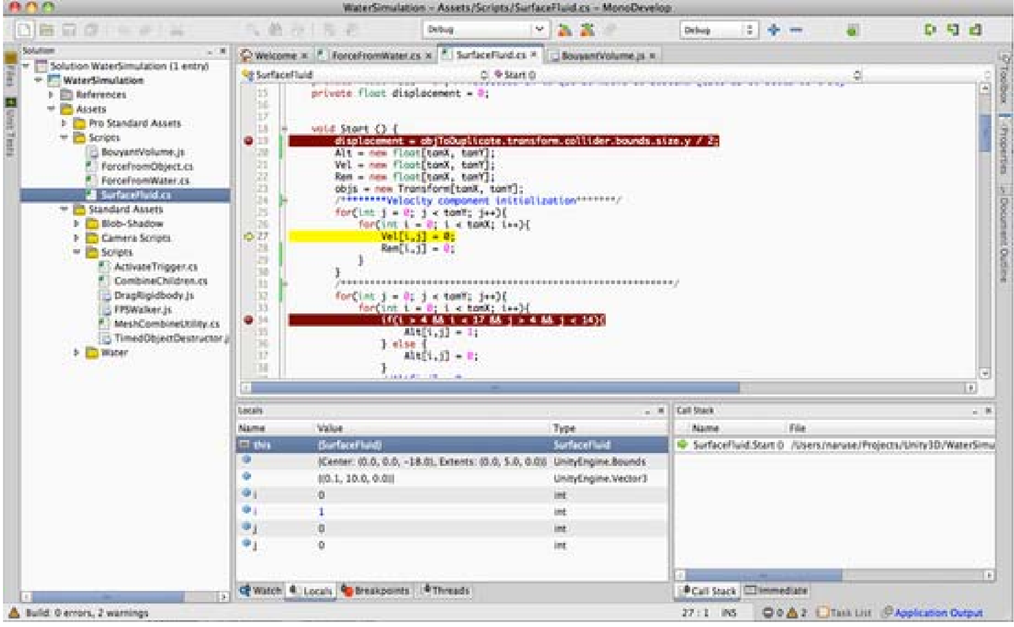

*Enabling debugging in the webplayer*

- Open your project in MonoDevelop.
- In MonoDevelop, click the Attach button in the toolbar, or choose **Attach** from the **Run** menu.
- From the dialog that appears, choose the item you wish to debug.
- **Notes:**
  - Currently supported debugging targets: Unity editors, desktop standalone players, Android and iOS players
  - If your player is set not to run in the background (the default), you may need to focus your player for a few seconds in order for it to appear in the list.
  - Android and iOS players need to have networking enabled when script debugging is enabled. All players need to be on the same network subnet as the computer running MonoDevelop.




- When you enter play mode, your script code will execute in the debugger.
- When a breakpoint occurs, script execution will stop, and you will be able to use MonoDevelop to step over, into, and out of your script methods, inspect your variables, examine the call stack, etc.
  - *Note:* When you're done debugging a toplevel method (e.g. `Update()`), or you just want to jump to the next breakpoint, you will experience better debugger performance by using the **Continue** command instead of stepping out or over the end of your function.





* When you're done debugging, click the Detach or Stop buttons in the toolbar, or choose **Detach** or **Stop** from the **Run** menu.

### Hints.

* If you add a watch to the ***this*** object, you can inspect the internal values (position, scale, rotation...) of the GameObject to which the script is attached.

### iOS remote debugging instructions

In addition to the instructions described above, Unity iOS applications require some additional steps for successful debugging:

1. Attach your iDevice to your WiFi network (the same requirement as for remote profiling).
2. Hit build & run in the Unity editor.
3. When the application builds, installs & launches via Xcode, click **Stop** in Xcode.
4. Manually find & launch your application on your iDevice. (**Note:** if the application is launched via Xcode you won't be able to resume after reaching a breakpoint).
5. When the app is running on the device, switch to MonoDevelop and click on the attach icon in the debugging toolbar. Select your device from the available instances list (if there are several instances shown, then select the bottom one).

Page last updated: 2012-10-30

---

## Log Files

There might be times during the development when you need to obtain information from the logs of the webplayer you've built, your standalone player, the target device or the editor. Usually you need to see these files when you have experienced a problem and you have to know where exactly the problem occurred.

On Mac the webplayer, player and editor logs can be accessed uniformly through the standard **Console.app** utility.

On Windows the webplayer and editor logs are place in folders there are not shown in the Windows Explorer by default. Please see the Accessing hidden folders page to resolve that situation.

### Editor

Editor log can be brought up through the **Open Editor Log** button in Unity's Console window.

| | |
|---|---|
| **Mac OS X** | ~/Library/Logs/Unity/Editor.log |
| **Windows XP** * | *C:\Documents and Settings\username\Local Settings\Application Data*\Unity\Editor\Editor.log |
| **Windows Vista/7** * | *C:\Users\username\AppData\Local*\Unity\Editor\Editor.log |

(*) On Windows the Editor log file is stored in the local application data folder: *%LOCALAPPDATA%*\Unity\Editor\Editor.log, where *LOCALAPPDATA* is defined by CSIDL_LOCAL_APPDATA.

## Desktop

On Mac all the logs can be accessed uniformly through the standard **Console.app** utility.

### Webplayer

| | |
|---|---|
| **Mac OS X** | ~/Library/Logs/Unity/WebPlayer.log |
| **Windows XP** * | *C:\Documents and Settings\username\Local Settings\Temp*\UnityWebPlayer\log\log_UNIQUEID.txt |
| **Windows Vista/7** * | *C:\Users\username\AppData\Local\Temp*\UnityWebPlayer\log\log_UNIQUEID.txt |
| **Windows Vista/7 + IE7 + UAC** * | *C:\Users\username\AppData\Local\Temp**Low***\UnityWebPlayer\log\log_UNIQUEID.txt |

(*) On Windows the webplayer log is stored in a temporary folder: *%TEMP%*\UnityWebPlayer\log\log_UNIQUEID.txt, where *TEMP* is defined by GetTempPath.

### Player

| | |
|---|---|
| **Mac OS X** | ~/Library/Logs/Unity/Player.log |
| **Windows** * | *EXECNAME_Data*\output_log.txt |

(*) *EXECNAME_Data* is a folder next to the executable with your game.

Note that on Windows standalones the location of the log file can be changed (or logging suppressed.) See the command line page for further details.

## iOS

The device log can be accessed in XCode via GDB console or the Organizer Console. The latter is useful for getting crashlogs when your application was not running through the XCode debugger.

Please see Debugging Applications in the iOS Development Guide. Also our Troubleshooting and Bugreporting guides may be useful for you.

## Android

The device log can be viewed by using the logcat console. Use the **adb** application found in **Android SDK/platform-tools directory** with a trailing **logcat** parameter:

```
$ adb logcat
```

Another way to inspect the LogCat is to use the Dalvik Debug Monitor Server (DDMS). DDMS can be started either from **Eclipse** or from inside the **Android SDK/tools**. DDMS also provides a number of other debug related tools.

Page last updated: 2012-06-15

---

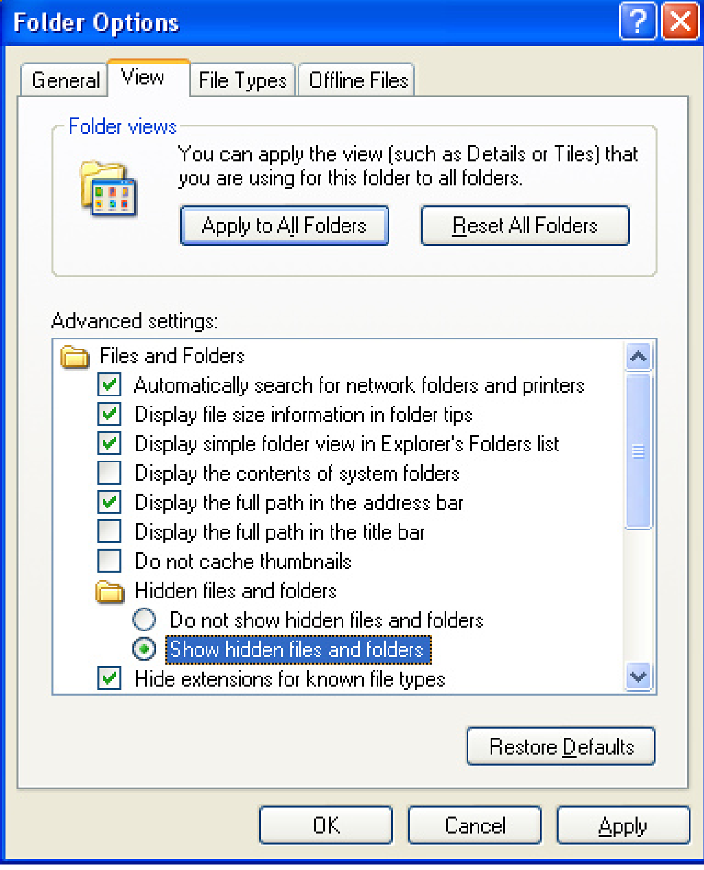
# Accessing Hidden Folders

On Windows the logs are stored in locations that are hidden by default. To enable navigating to them in the Windows Explorer

please perform the steps below.

### Show hidden folders on Windows XP

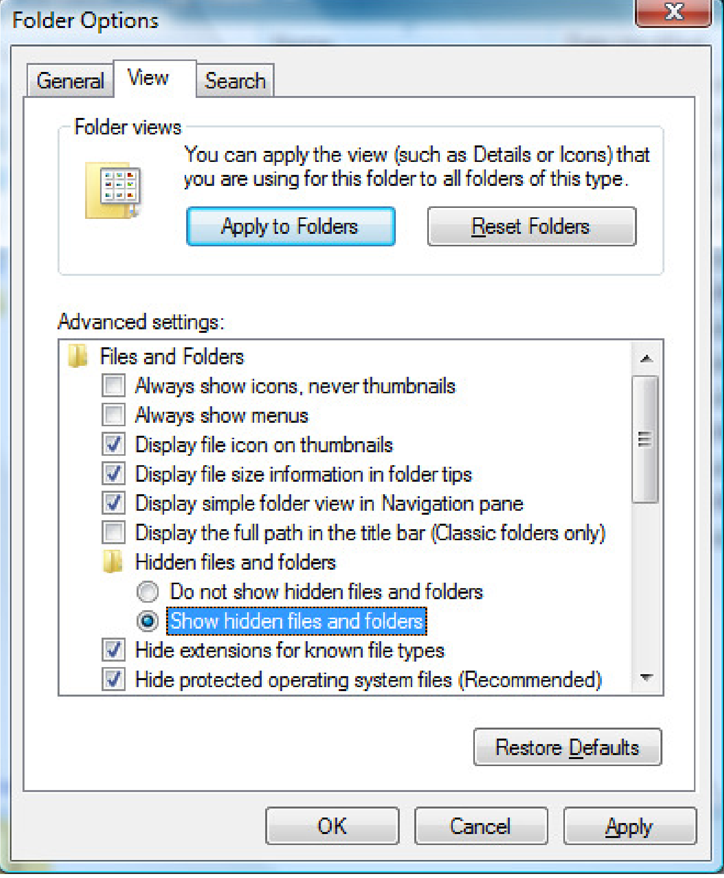
The Local Settings folder is hidden by default. In order to see it, you have to enable viewing of hidden folders in Windows Explorer from **Tools->Folder Options...->View (tab)**.




*Enabling viewing of hidden folders in Windows XP*

### Show hidden folders on Windows Vista/7

The AppData folder is hidden by default. In order to see it, you have to enable viewing of hidden folders in Windows Explorer from **Tools->Folder Options...->View (tab)**. The Tools menu is hidden by default, but can be displayed by pressing the Alt key once.




*Enabling viewing of hidden folders in Windows Vista*

Page last updated: 2010-09-17

---

# Plugins

Unity has extensive support for **Plugins**, which are libraries of native code written in C, C++, Objective-C, etc. Plugins allow your game code (written in Javascript, C# or Boo) to call functions from these libraries. This feature allows Unity to integrate with middleware libraries or existing C/C++ game code.

**Note:** On the desktop platforms, plugins are a pro-only feature. For security reasons, plugins are not usable with webplayers.

In order to use a plugin you need to do two things:-

- Write functions in a C-based language and compile them into a library.
- Create a C# script which calls functions in the library.

The plugin should provide a simple C interface which the C# script then exposes to other user scripts. It is also possible for Unity to call functions exported by the plugin when certain low-level rendering events happen (for example, when a graphics device is created), see the Native Plugin Interface page for details.

Here is a very simple example:

**C File of a Minimal Plugin:**

```
float FooPluginFunction () { return 5.0F; }
```

**C# Script that Uses the Plugin:**

```
using UnityEngine;
using System.Runtime.InteropServices;
```

```
class SomeScript : MonoBehaviour {

   #if UNITY_IPHONE || UNITY_XBOX360

   // On iOS and Xbox 360 plugins are statically linked into
   // the executable, so we have to use __Internal as the
   // library name.
   [DllImport ("__Internal")]

   #else

   // Other platforms load plugins dynamically, so pass the name
   // of the plugin's dynamic library.
   [DllImport ("PluginName")]

   #endif

   private static extern float FooPluginFunction ();

   void Awake () {
      // Calls the FooPluginFunction inside the plugin
      // And prints 5 to the console
      print (FooPluginFunction ());
   }
}
```

Note that when using Javascript you will need to use the following syntax, where DLLName is the name of the plugin you have written, or "__Internal" if you are writing statically linked native code:

```
@DllImport (DLLName)
static private function FooPluginFunction () : float {};
```

### Creating a Plugin

In general, plugins are built with native code compilers on the target platform. Since plugin functions use a C-based call interface, you must avoid name mangling issues when using C++ or Objective-C.

For further details and examples, see the following pages:-

- Building Plugins for Desktop Platforms
- Building Plugins for iOS
- Building Plugins for Android

### Further Information

- Native Plugin Interface - this is needed if you want to do rendering in your plugin.
- Mono Interop with native libraries.
- P-invoke documentation on MSDN.

Page last updated: 2012-02-02

---

## PluginsForDesktop

This page describes Native Code Plugins for desktop platforms (Windows/Mac OS X/Linux). Note that plugins are intentionally disabled in webplayers for security reasons.

### Building a Plugin for Mac OS X

On Mac OSX, plugins are deployed as bundles. You can create the bundle project with XCode by selecting **File->NewProject...** and then selecting Bundle - Carbon/Cocoa Loadable Bundle.

If you are using C++ (.cpp) or Objective-C (.mm) to implement the plugin then you must ensure the functions are declared with C linkage to avoid name mangling issues.

```
extern "C" {
  float FooPluginFunction ();
}
```

## Building a Plugin for Windows

Plugins on Windows are DLL files with exported functions. Practically any language or development environment that can create DLL files can be used to create plugins.
As with Mac OSX, you should declare any C++ functions with C linkage to avoid name mangling issues.

## Building a Plugin for Linux

Plugins on Linux are `.so` files with exported functions. These libraries are typically written in C or C++, but any language can be used.
As with the other platforms, you should declare any C++ functions with C linkage in order to avoid name mangling issues.

### 32-bit and 64-bit libraries

Currently, plugins for 32-bit and 64-bit players need to be managed manually, e.g, before building a 64-bit player, you need to copy the 64-bit library into the `Assets/Plugins` folder, and before building a 32-bit player, you need to copy the 32-bit library into the `Assets/Plugins` folder.

## Using your plugin from C#

Once built, the bundle should be placed in the **Assets->Plugins** folder in the Unity project. Unity will then find it by name when you define a function like this in the C# script:-

```
[DllImport ("PluginName")]
private static extern float FooPluginFunction ();
```

Please note that **PluginName** should not include the library prefix nor file extension. For example, the actual name of the plugin file would be PluginName.dll on Windows and libPluginName.so on Linux.
Be aware that whenever you change code in the Plugin you will need to recompile scripts in your project or else the plugin will not have the latest compiled code.

## Deployment

For cross platform plugins you must include the .bundle (for Mac), .dll (for Windows), and .so (for Linux) files in the Plugins folder.
No further work is then required on your side - Unity automatically picks the right plugin for the target platform and includes it with the player.

## Examples

### Simplest Plugin

This plugin project implements only some very basic operations (print a number, print a string, add two floats, add two integers). This example may be helpful if this is your first Unity plugin.
The project can be found here and includes Windows, Mac, and Linux project files.

### Rendering from C++ code

An example multiplatform plugin that works with multithreaded rendering in Unity can be found on the Native Plugin Interface page.

### Midi Plugin

A complete example of the Plugin interface can be found here.

This is a complete Midi plugin for OS X which uses Apple's CoreMidi API. It provides a simple C API and a C# class to access it from Unity. The C# class contains a high level API, with easy access to NoteOn and NoteOff events and their velocity.

### Texture Plugin

An example of how to assign image data to a texture directly in OpenGL (note that this will only work when Unity is using an

OpenGL renderer). This example includes both XCode and Visual Studio project files. The plugin, along with an accompanying Unity project, can be found here.

Page last updated: 2012-11-16

---

# PluginsForIOS

This page describes Native Code Plugins for the iOS platform.

## Building an Application with a Native Plugin for iOS

1. Define your extern method in the C# file as follows:
```
[DllImport ("__Internal")]
private static extern float FooPluginFunction ();
```
2. Set the editor to the iOS build target
3. Add your native code source files to the generated XCode project's "Classes" folder (this folder is not overwritten when the project is updated, but don't forget to backup your native code).

If you are using C++ (.cpp) or Objective-C (.mm) to implement the plugin you must ensure the functions are declared with C linkage to avoid name mangling issues.

```
extern "C" {
  float FooPluginFunction ();
}
```

## Using Your Plugin from C#

iOS native plugins can be called only when deployed on the actual device, so it is recommended to wrap all native code methods with an additional C# code layer. This code should check Application.platform and call native methods only when the app is running on the device; dummy values can be returned when the app runs in the Editor. See the Bonjour browser sample application for an example.

## Calling C# / JavaScript back from native code

Unity iOS supports limited native-to-managed callback functionality via *UnitySendMessage*:

```
UnitySendMessage("GameObjectName1", "MethodName1", "Message to send");
```

This function has three parameters : the name of the target GameObject, the script method to call on that object and the message string to pass to the called method.

Known limitations:

1. Only script methods that correspond to the following signature can be called from native code: `function MethodName(message:string)`
2. Calls to *UnitySendMessage* are asynchronous and have a delay of one frame.

## Automated plugin integration

Unity iOS supports automated plugin integration in a limited way. All files with extensions **.a**,**.m**,**.mm**,**.c**,**.cpp** located in the Assets/**Plugins/iOS** folder will be merged into the generated Xcode project automatically. However, merging is done by symlinking files from Assets/**Plugins/iOS** to the final destination, which might affect some workflows. The **.h** files are not included in the Xcode project tree, but they appear on the destination file system, thus allowing compilation of .m/.mm/.c/.cpp files.

**Note:** subfolders are currently not supported.

## iOS Tips

1. Managed-to-unmanaged calls are quite processor intensive on iOS. Try to avoid calling multiple native methods per frame.
2. As mentioned above, wrap your native methods with an additional C# layer that calls native code on the device and returns dummy values in the Editor.
3. String values returned from a native method should be UTF-8 encoded and allocated on the heap. Mono marshaling calls are free for strings like this.
4. As mentioned above, the XCode project's "Classes" folder is a good place to store your native code because it is not overwritten when the project is updated.
5. Another good place for storing native code is the Assets folder or one of its subfolders. Just add references from the XCode project to the native code files: right click on the "Classes" subfolder and choose "Add->Existing files...".

### Examples

#### Bonjour Browser Sample

A simple example of the use of a native code plugin can be found here

This sample demonstrates how objective-C code can be invoked from a Unity iOS application. This application implements a very simple Bonjour client. The application consists of a Unity iOS project (Plugins/Bonjour.cs is the C# interface to the native code, while BonjourTest.js is the JS script that implements the application logic) and native code (Assets/Code) that should be added to the built XCode project.

Page last updated: 2011-11-01

---

# PluginsForAndroid

This page describes Native Code Plugins for Android.

### Building a Plugin for Android

To build a plugin for Android, you should first obtain the Android NDK and familiarize yourself with the steps involved in building a shared library.

If you are using C++ (.cpp) to implement the plugin you must ensure the functions are declared with C linkage to avoid name mangling issues.

```
extern "C" {
  float FooPluginFunction ();
}
```

### Using Your Plugin from C#

Once built, the shared library should be copied to the **Assets->Plugins->Android** folder. Unity will then find it by name when you define a function like the following in the C# script:-

```
[DllImport ("PluginName")]
private static extern float FooPluginFunction ();
```

Please note that **PluginName** should not include the prefix ('lib') nor the extension ('.so') of the filename. It is advisable to wrap all native code methods with an additional C# code layer. This code should check Application.platform and call native methods only when the app is running on the actual device; dummy values can be returned from the C# code when running in the Editor. You can also use platform defines to control platform dependent code compilation.

### Deployment

For cross platform deployment, your project should include plugins for each supported platform (ie, libPlugin.so for Android, Plugin.bundle for Mac and Plugin.dll for Windows). Unity automatically picks the right plugin for the target platform and includes it with the player.

### Using Java Plugins

The Android plugin mechanism also allows Java to be used to enable interaction with the Android OS.

**Building a Java Plugin for Android**
There are several ways to create a Java plugin but the result in each case is that you end up with a .jar file containing the .class files for your plugin. One approach is to download the JDK, then compile your .java files from the command line with *javac*. This will create .class files which you can then package into a .jar with the *jar* command line tool. Another option is to use the Eclipse IDE together with the ADT.

**Using Your Java Plugin from Native Code**
Once you have built your Java plugin (.jar) you should copy it to the **Assets->Plugins->Android** folder in the Unity project. Unity will package your .class files together with the rest of the Java code and then access the code using the Java Native Interface (JNI). JNI is used both when calling native code from Java and when interacting with Java (or the JavaVM) from native code.

To find your Java code from the native side you need access to the Java VM. Fortunately, that access can be obtained easily by adding a function like this to your C/C++ code:

```
jint JNI_OnLoad(JavaVM* vm, void* reserved) {
  JNIEnv* jni_env = 0;
  vm->AttachCurrentThread(&jni_env, 0);
}
```

This is all that is needed to start using Java from C/C++. It is beyond the scope of this document to explain JNI completely. However, using it usually involves finding the class definition, resolving the constructor (<init>) method and creating a new object instance, as shown in this example:-

```
jobject createJavaObject(JNIEnv* jni_env) {
  jclass cls_JavaClass = jni_env->FindClass("com/your/java/Class");                // find class definition
  jmethodID mid_JavaClass = jni_env->GetMethodID (cls_JavaClass, "<init>", "()V");         // find constructor method
  jobject obj_JavaClass = jni_env->NewObject(cls_JavaClass, mid_JavaClass);           // create object instance
  return jni_env->NewGlobalRef(obj_JavaClass);                                    // return object with a global reference
}
```

**Using Your Java Plugin with helper classes**
**AndroidJNIHelper** and **AndroidJNI** can be used to ease some of the pain with raw JNI.

**AndroidJavaObject** and **AndroidJavaClass** automate a lot of tasks and also use cacheing to make calls to Java faster. The combination of **AndroidJavaObject** and **AndroidJavaClass** builds on top of **AndroidJNI** and **AndroidJNIHelper**, but also has a lot of logic in its own right (to handle the automation). These classes also come in a 'static' version to access static members of Java classes.

You can choose whichever approach you prefer, be it raw JNI through **AndroidJNI** class methods, or **AndroidJNIHelper** together with **AndroidJNI** and eventually **AndroidJavaObject/AndroidJavaClass** for maximum automation and convenience.

UnityEngine.AndroidJNI is a wrapper for the JNI calls available in C (as described above). All methods in this class are static and have a 1:1 mapping to the Java Native Interface. UnityEngine.AndroidJNIHelper provides helper functionality used by the next level, but is exposed as public methods because they may be useful for some special cases.

Instances of UnityEngine.AndroidJavaObject and UnityEngine.AndroidJavaClass have a 1:1 mapping to an instance of java.lang.Object and java.lang.Class (or subclasses thereof) on the Java side, respectively. They essentially provide 3 types of interaction with the Java side:

- Call a method
- Get the value of a field
- Set the value of a field

The **Call** is separated into two categories: **Call** to a 'void' method, and **Call** to a method with non-void return type. A generic type is used to represent the return type of those methods which return a non-void type. The **Get** and **Set** always take a generic type representing the field type.

**Example 1**

```
//The comments describe what you would need to do if you were using raw JNI
 AndroidJavaObject jo = new AndroidJavaObject("java.lang.String", "some_string");
 // jni.FindClass("java.lang.String");
 // jni.GetMethodID(classID, "<init>", "(Ljava/lang/String;)V");
 // jni.NewStringUTF("some_string");
 // jni.NewObject(classID, methodID, javaString);
 int hash = jo.Call<int>("hashCode");
 // jni.GetMethodID(classID, "hashCode", "()I");
 // jni.CallIntMethod(objectID, methodID);
```

Here, we're creating an instance of java.lang.String, initialized with a string of our choice and retrieving the hash value for that string.

The **AndroidJavaObject** constructor takes at least one parameter, the name of class for which we want to construct an instance. Any parameters after the class name are for the constructor call on the object, in this case the string "some_string". The subsequent **Call** to hashCode() returns an 'int' which is why we use that as the generic type parameter to the **Call** method.

**Note:** You cannot instantiate a nested Java class using dotted notation. Inner classes must use the $ separator, and it should work in both dotted and slashed format. So **android.view.ViewGroup$LayoutParams** or **android/view/ViewGroup$LayoutParams** can be used, where a **LayoutParams** class is nested in a **ViewGroup** class.

**Example 2**

One of the plugin samples above shows how to get the cache directory for the current application. This is how you would do the same thing from C# without any plugins:-

```
AndroidJavaClass jc = new AndroidJavaClass("com.unity3d.player.UnityPlayer");
// jni.FindClass("com.unity3d.player.UnityPlayer");
AndroidJavaObject jo = jc.GetStatic<AndroidJavaObject>("currentActivity");
// jni.GetStaticFieldID(classID, "Ljava/lang/Object;");
// jni.GetStaticObjectField(classID, fieldID);
// jni.FindClass("java.lang.Object");

Debug.Log(jo.Call<AndroidJavaObject>("getCacheDir").Call<string>("getCanonicalPath"));
// jni.GetMethodID(classID, "getCacheDir", "()Ljava/io/File;"); // or any baseclass thereof!
// jni.CallObjectMethod(objectID, methodID);
// jni.FindClass("java.io.File");
// jni.GetMethodID(classID, "getCanonicalPath", "()Ljava/lang/String;");
// jni.CallObjectMethod(objectID, methodID);
// jni.GetStringUTFChars(javaString);
```

In this case, we start with **AndroidJavaClass** instead of **AndroidJavaObject** because we want to access a static member of **com.unity3d.player.UnityPlayer** rather than create a new object (an instance is created automatically by the **Android UnityPlayer**). Then we access the static field "currentActivity" but this time we use **AndroidJavaObject** as the generic parameter. This is because the actual field type (android.app.Activity) is a subclass of java.lang.Object, and any non-primitive type must be accessed as **AndroidJavaObject**. The exceptions to this rule are strings, which can be accessed directly even though they don't represent a primitive type in Java.

After that it is just a matter of traversing the **Activity** through getCacheDir() to get the File object representing the cache directory, and then calling getCanonicalPath() to get a string representation.

Of course, nowadays you don't need to do that to get the cache directory since Unity provides access to the application's cache and file directory with Application.temporaryCachePath and Application.persistentDataPath.

**Example 3**

Finally, here is a trick for passing data from Java to script code using **UnitySendMessage**.

```
using UnityEngine;
public class NewBehaviourScript : MonoBehaviour {

	void Start () {
		JNIHelper.debug = true;
		using (JavaClass jc = new JavaClass("com.unity3d.player.UnityPlayer")) {
			jc.CallStatic("UnitySendMessage", "Main Camera", "JavaMessage", "whoowhoo");
		}
	}

	void JavaMessage(string message) {
		Debug.Log("message from java: " + message);
	}
}
```

The Java class **com.unity3d.player.UnityPlayer** now has a static method **UnitySendMessage**, equivalent to the iOS UnitySendMessage on the native side. It can be used in Java to pass data to script code.

Here though, we call it directly from script code, which essentially relays the message on the Java side. This then calls back to the native/Unity code to deliver the message to the object named "Main Camera". This object has a script attached which contains a method called "JavaMessage".

**Best practice when using Java plugins with Unity**

As this section is mainly aimed at people who don't have comprehensive JNI, Java and Android experience, we assume that the **AndroidJavaObject/AndroidJavaClass** approach has been used for interacting with Java code from Unity.

The first thing to note is that any operation you perform on an **AndroidJavaObject** or **AndroidJavaClass** is computationally expensive (as is the raw JNI approach). It is highly advisable to keep the number of transitions between managed and native/Java code to a minimum, for the sake of performance and also code clarity.

You could have a Java method to do all the actual work and then use **AndroidJavaObject / AndroidJavaClass** to communicate with that method and get the result. However, it is worth bearing in mind that the JNI helper classes try to cache as much data as possible to improve performance.

```
//The first time you call a Java function like
AndroidJavaObject jo = new AndroidJavaObject("java.lang.String", "some_string");  // somewhat expensive
int hash = jo.Call<int>("hashCode");  // first time - expensive
int hash = jo.Call<int>("hashCode");  // second time - not as expensive as we already know the java method and can call it 
```

The Mono garbage collector should release all created instances of **AndroidJavaObject** and **AndroidJavaClass** after use, but it is advisable to keep them in a **using(){}** statement to ensure they are deleted as soon as possible. Without this, you cannot be sure when they will be destroyed. If you set **AndroidJNIHelper.debug** to true, you will see a record of the garbage collector's activity in the debug output.

```
//Getting the system language with the safe approach
void Start () {
	using (AndroidJavaClass cls = new AndroidJavaClass("java.util.Locale")) {
		using(AndroidJavaObject locale = cls.CallStatic<AndroidJavaObject>("getDefault")) {
			Debug.Log("current lang = " + locale.Call<string>("getDisplayLanguage"));

		}
	}
}
```

You can also call the **.Dispose()** method directly to ensure there are no Java objects lingering. The actual C# object might live a bit longer, but will be garbage collected by mono eventually.

## Extending the UnityPlayerActivity Java Code

With Unity Android it is possible to extend the standard **UnityPlayerActivity** class (the primary Java class for the Unity Player on Android, similar to AppController.mm on Unity iOS).

An application can override any and all of the basic interaction between Android OS and Unity Android. You can enable this by creating a new Activity which derives from UnityPlayerActivity (UnityPlayerActivity.java can be found at **/Applications/Unity /Unity.app/Contents/PlaybackEngines/AndroidPlayer/src/com/unity3d/player** on Mac and usually at **C:\Program Files\Unity\Editor\Data\PlaybackEngines\AndroidPlayer\src\com\unity3d\player** on Windows).

To do this, first locate the **classes.jar** shipped with Unity Android. It is found in the installation folder (usually **C:\Program Files\Unity\Editor\Data** (on Windows) or **/Applications/Unity** (on Mac)) in a sub-folder called **PlaybackEngines/AndroidPlayer/bin**. Then add **classes.jar** to the classpath used to compile the new Activity. The resulting .class file(s) should be compressed into a .jar file and placed in the **Assets->Plugins->Android** folder. Since the manifest dictates which activity to launch it is also necessary to create a new AndroidManifest.xml. The AndroidManifest.xml file should also be placed in the **Assets->Plugins->Android** folder.

The new activity could look like the following example, **OverrideExample.java**:

```
package com.company.product;

import com.unity3d.player.UnityPlayerActivity;

import android.os.Bundle;
import android.util.Log;

public class OverrideExample extends UnityPlayerActivity {

  protected void onCreate(Bundle savedInstanceState) {

    // call UnityPlayerActivity.onCreate()
    super.onCreate(savedInstanceState);

    // print debug message to logcat
    Log.d("OverrideActivity", "onCreate called!");
  }

  public void onBackPressed()
  {
    // instead of calling UnityPlayerActivity.onBackPressed() we just ignore the back button event
    // super.onBackPressed();
  }
}
```

And this is what the corresponding **AndroidManifest.xml** would look like:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.company.product">
  <application android:icon="@drawable/app_icon" android:label="@string/app_name">
      <activity android:name=".OverrideExample"
                  android:label="@string/app_name"
                  android:configChanges="fontScale|keyboard|keyboardHidden|locale|mnc|mcc|navigation|orientation|screen
      <intent-filter>
                  <action android:name="android.intent.action.MAIN" />
                  <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
      </activity>
  </application>
</manifest>
```

**UnityPlayerNativeActivity**

It is also possible to create your own subclass of `UnityPlayerNativeActivity`. This will have much the same effect as subclassing UnityPlayerActivity but with improved input latency. Be aware, though, that NativeActivity was introduced in Gingerbread and does not work with older devices. Since touch/motion events are processed in native code, Java views would normally not see those events. There is, however, a forwarding mechanism in Unity which allows events to be propagated to the DalvikVM. To access this mechanism, you need to modify the manifest file as follows:-

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.company.product">
  <application android:icon="@drawable/app_icon" android:label="@string/app_name">
      <activity android:name=".OverrideExampleNative"
                  android:label="@string/app_name"
                  android:configChanges="fontScale|keyboard|keyboardHidden|locale|mnc|mcc|navigation|orientation|screen
  <meta-data android:name="android.app.lib_name" android:value="unity" />
  <meta-data android:name="unityplayer.ForwardNativeEventsToDalvik" android:value="true" />
      <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
      </activity>
  </application>
</manifest>
```

Note the ".OverrideExampleNative" attribute in the activity element and the two additional meta-data elements. The first meta-data is an instruction to use the Unity library **libunity.so**. The second enables events to be passed on to your custom subclass of UnityPlayerNativeActivity.

### Examples

**Native Plugin Sample**
A simple example of the use of a native code plugin can be found here

This sample demonstrates how C code can be invoked from a Unity Android application. The package includes a scene which displays the sum of two values as calculated by the native plugin. Please note that you will need the Android NDK to compile the plugin.

**Java Plugin Sample**
An example of the use of Java code can be found here

This sample demonstrates how Java code can be used to interact with the Android OS and how C++ creates a bridge between C# and Java. The scene in the package displays a button which when clicked fetches the application cache directory, as defined by the Android OS. Please note that you will need both the JDK and the Android NDK to compile the plugins.

Here is a similar example but based on a prebuilt JNI library to wrap the native code into C#.

Page last updated: 2012-09-25

---

# NativePluginInterface

In addition to the basic script interface, Native Code Plugins in Unity can receive callbacks when certain events happen. This is mostly used to implement low-level rendering in your plugin and enable it to work with Unity's multithreaded rendering.

**Note:** The rendering callbacks to plugins are not currently supported on mobile platforms.

### Access to the Graphics Device

A plugin can receive notification about events on the graphics device by exporting a `UnitySetGraphicsDevice` function. This will be called when the graphics device is created, before it is destroyed, and also before and after the device is "reset" (this only happens with Direct3D 9). The function has parameters which will receive the device pointer, device type and the

kind of event that is taking place.

```
// If exported by a plugin, this function will be called when graphics device is created, de
// and before and after it is reset (ie, resolution changed).
extern "C" void EXPORT_API UnitySetGraphicsDevice (void* device, int deviceType, int event
```

Possible values for deviceType:

```
enum GfxDeviceRenderer {
    kGfxRendererOpenGL = 0,              // OpenGL
    kGfxRendererD3D9 = 1,                // Direct3D 9
    kGfxRendererD3D11 = 2,               // Direct3D 11
    kGfxRendererGCM = 3,                 // Sony PlayStation 3 GCM
    kGfxRendererNull = 4,                // "null" device (used in batch mode)
    kGfxRendererHollywood = 5,           // Nintendo Wii
    kGfxRendererXenon = 6,               // Xbox 360
    kGfxRendererOpenGLES = 7,            // OpenGL ES 1.1
    kGfxRendererOpenGLES20Mobile = 8,    // OpenGL ES 2.0 mobile variant
    kGfxRendererMolehill = 9,            // Flash 11 Stage3D
    kGfxRendererOpenGLES20Desktop = 10,  // OpenGL ES 2.0 desktop variant (i.e. NaCl)
};
```

Possible values for eventType:

```
enum GfxDeviceEventType {
    kGfxDeviceEventInitialize = 0,
    kGfxDeviceEventShutdown = 1,
    kGfxDeviceEventBeforeReset = 2,
    kGfxDeviceEventAfterReset = 3,
};
```

### Plugin Callbacks on the Rendering Thread

Rendering in Unity can be multithreaded if the platform and number of available CPUs will allow for it. When multithreaded rendering is used, the rendering API commands happen on a thread which is completely separate from the one that runs MonoBehaviour scripts. Consequently, it is not always possible for your plugin to start doing some rendering immediately, since might interfere with whatever the render thread is doing at the time.

In order to do **any** rendering from the plugin, you should call GL.IssuePluginEvent from your script, which will cause your plugin to be called from the render thread. For example, if you call GL.IssuePluginEvent from the camera's OnPostRender function, you get a plugin callback immediately after the camera has finished rendering.

```
// If exported by a plugin, this function will be called for GL.IssuePluginEvent script call
// The function will be called on a rendering thread; note that when multithreaded rendering
// the render thread WILL BE DIFFERENT from the main thread, on which all scripts & other ga
// You have responsibility for ensuring any necessary synchronization with other plugin scri
extern "C" void EXPORT_API UnityRenderEvent (int eventID);
```

### Example

An example of a low-level rendering plugin **can be downloaded here**. It demonstrates two things:

- Renders a rotating triangle from C++ code after all regular rendering is done.
- Fills a procedural texture from C++ code, using Texture.GetNativeTexturePtr to access it.

The project works with Windows (Visual Studio 2008) and Mac OS X (Xcode 3.2) and uses Direct3D 9, Direct3D 11 or OpenGL depending on the platform. Direct3D 9 code part also demonstrates how to handle "lost" devices.

Page last updated: 2012-11-16

## TextualSceneFormat

As well as the default binary format, Unity also provides a textual format for scene data. This can be useful when working with version control software, since textual files generated separately can be merged more easily than binary files. Also, the text data can be generated and parsed by tools, making it possible to create and analyze scenes automatically. The pages in this section provide some reference material for working with the format.

- Description of the Format
- YAMLSceneExample
- YAML Class ID Reference

Page last updated: 2011-10-13

---

## FormatDescription

Unity's scene format is implemented with the YAML data serialization language. While we can't cover YAML in depth here, it is an open format and its specification is available for free at the YAML website. Each object in the scene is written to the file as a separate YAML document, which is introduced in the file by the --- sequence. Note that in this context, the term "object" refers to GameObjects, Components and other scene data collectively; each of these items requires its own YAML document in the scene file. The basic structure of a serialized object can be understood from an example:-

```
--- !u!1 &6
GameObject:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  importerVersion: 3
  m_Component:
  - 4: {fileID: 8}
  - 33: {fileID: 12}
  - 65: {fileID: 13}
  - 23: {fileID: 11}
  m_Layer: 0
  m_Name: Cube
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
```

The first line contains the string "!u!1 &6" after the document marker. The first number after the "!u!" part indicates the class of the object (in this case, it is a GameObject). The number following the ampersand is an object ID number which is unique within the file, although the number is assigned to each object arbitrarily. Each of the object's serializable properties is denoted by a line like the following:-

```
m_Name: Cube
```

Properties are typically prefixed with "m_" but otherwise follow the name of the property as defined in the script reference. A second object, defined further down in the file, might look something like this:-

```
--- !u!4 &8
Transform:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 6}
  m_LocalRotation: {x: 0.000000, y: 0.000000, z: 0.000000, w: 1.000000}
```

```
  m_LocalPosition: {x: -2.618721, y: 1.028581, z: 1.131627}
  m_LocalScale: {x: 1.000000, y: 1.000000, z: 1.000000}
  m_Children: []
  m_Father: {fileID: 0}
```

This is a Transform component attached to the GameObject defined by the YAML document above. The attachment is denoted by the line:-

```
m_GameObject: {fileID: 6}
```

...since the GameObject's object ID within the file was 6.

Floating point numbers can be represented in a decimal representation or as a hexadecimal number in IEE 754 format (denoted by a 0x prefix). The IEE 754 representation is used for lossless encoding of values, and is used by Unity when writing floating point values which don't have a short decimal representation. When Unity writes numbers in hexadecimal, it will always also write the decimal format in parentheses for debugging purposes, but only the hex is actually parsed when loading the file. If you wish to edit such values manually, simply remove the hex and enter only a decimal number. Here are some valid representations of floating point values (all representing the number one):

```
myValue: 0x3F800000
myValue: 1
myValue: 1.000
myValue: 0x3f800000(1)
myValue: 0.1e1
```

Page last updated: 2012-01-05

---

# YAMLSceneExample

An Example of a YAML Scene File

An example of a simple but complete scene is given below. The scene contains just a camera and a cube object. Note that the file **must** start with the two lines

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
```

...in order to be accepted by Unity. Otherwise, the import process is designed to be tolerant of omissions - default values will be supplied for missing property data as far as possible.

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!header
SerializedFile:
  m_TargetPlatform: 4294967294
  m_UserInformation:
--- !u!29 &1
Scene:
  m_ObjectHideFlags: 0
  m_PVSData:
  m_QueryMode: 1
  m_PVSObjectsArray: []
  m_PVSPortalsArray: []
  m_ViewCellSize: 1.000000
--- !u!104 &2
```

```
RenderSettings:
  m_Fog: 0
  m_FogColor: {r: 0.500000, g: 0.500000, b: 0.500000, a: 1.000000}
  m_FogMode: 3
  m_FogDensity: 0.010000
  m_LinearFogStart: 0.000000
  m_LinearFogEnd: 300.000000
  m_AmbientLight: {r: 0.200000, g: 0.200000, b: 0.200000, a: 1.000000}
  m_SkyboxMaterial: {fileID: 0}
  m_HaloStrength: 0.500000
  m_FlareStrength: 1.000000
  m_HaloTexture: {fileID: 0}
  m_SpotCookie: {fileID: 0}
  m_ObjectHideFlags: 0
--- !u!127 &3
GameManager:
  m_ObjectHideFlags: 0
--- !u!157 &4
LightmapSettings:
  m_ObjectHideFlags: 0
  m_LightProbeCloud: {fileID: 0}
  m_Lightmaps: []
  m_LightmapsMode: 1
  m_BakedColorSpace: 0
  m_UseDualLightmapsInForward: 0
  m_LightmapEditorSettings:
    m_Resolution: 50.000000
    m_LastUsedResolution: 0.000000
    m_TextureWidth: 1024
    m_TextureHeight: 1024
    m_BounceBoost: 1.000000
    m_BounceIntensity: 1.000000
    m_SkyLightColor: {r: 0.860000, g: 0.930000, b: 1.000000, a: 1.000000}
    m_SkyLightIntensity: 0.000000
    m_Quality: 0
    m_Bounces: 1
    m_FinalGatherRays: 1000
    m_FinalGatherContrastThreshold: 0.050000
    m_FinalGatherGradientThreshold: 0.000000
    m_FinalGatherInterpolationPoints: 15
    m_AOAmount: 0.000000
    m_AOMaxDistance: 0.100000
    m_AOContrast: 1.000000
    m_TextureCompression: 0
    m_LockAtlas: 0
--- !u!196 &5
NavMeshSettings:
  m_ObjectHideFlags: 0
  m_BuildSettings:
    cellSize: 0.200000
    cellHeight: 0.100000
    agentSlope: 45.000000
    agentClimb: 0.900000
    ledgeDropHeight: 0.000000
    maxJumpAcrossDistance: 0.000000
    agentRadius: 0.400000
    agentHeight: 1.800000
    maxEdgeLength: 12
    maxSimplificationError: 1.300000
    regionMinSize: 8
    regionMergeSize: 20
```

```
    tileSize: 500
    detailSampleDistance: 6.000000
    detailSampleMaxError: 1.000000
    accuratePlacement: 0
  m_NavMesh: {fileID: 0}
--- !u!1 &6
GameObject:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  importerVersion: 3
  m_Component:
  - 4: {fileID: 8}
  - 33: {fileID: 12}
  - 65: {fileID: 13}
  - 23: {fileID: 11}
  m_Layer: 0
  m_Name: Cube
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!1 &7
GameObject:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  importerVersion: 3
  m_Component:
  - 4: {fileID: 9}
  - 20: {fileID: 10}
  - 92: {fileID: 15}
  - 124: {fileID: 16}
  - 81: {fileID: 14}
  m_Layer: 0
  m_Name: Main Camera
  m_TagString: MainCamera
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!4 &8
Transform:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 6}
  m_LocalRotation: {x: 0.000000, y: 0.000000, z: 0.000000, w: 1.000000}
  m_LocalPosition: {x: -2.618721, y: 1.028581, z: 1.131627}
  m_LocalScale: {x: 1.000000, y: 1.000000, z: 1.000000}
  m_Children: []
  m_Father: {fileID: 0}
--- !u!4 &9
Transform:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 7}
  m_LocalRotation: {x: 0.000000, y: 0.000000, z: 0.000000, w: 1.000000}
  m_LocalPosition: {x: 0.000000, y: 1.000000, z: -10.000000}
```

```
  m_LocalScale: {x: 1.000000, y: 1.000000, z: 1.000000}
  m_Children: []
  m_Father: {fileID: 0}
--- !u!20 &10
Camera:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 7}
  m_Enabled: 1
  importerVersion: 2
  m_ClearFlags: 1
  m_BackGroundColor: {r: 0.192157, g: 0.301961, b: 0.474510, a: 0.019608}
  m_NormalizedViewPortRect:
    importerVersion: 2
    x: 0.000000
    y: 0.000000
    width: 1.000000
    height: 1.000000
  near clip plane: 0.300000
  far clip plane: 1000.000000
  field of view: 60.000000
  orthographic: 0
  orthographic size: 100.000000
  m_Depth: -1.000000
  m_CullingMask:
    importerVersion: 2
    m_Bits: 4294967295
  m_RenderingPath: -1
  m_TargetTexture: {fileID: 0}
  m_HDR: 0
--- !u!23 &11
Renderer:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 6}
  m_Enabled: 1
  m_CastShadows: 1
  m_ReceiveShadows: 1
  m_LightmapIndex: 255
  m_LightmapTilingOffset: {x: 1.000000, y: 1.000000, z: 0.000000, w: 0.000000}
  m_Materials:
  - {fileID: 10302, guid: 0000000000000000e000000000000000, type: 0}
  m_SubsetIndices:
  m_StaticBatchRoot: {fileID: 0}
  m_LightProbeAnchor: {fileID: 0}
  m_UseLightProbes: 0
  m_ScaleInLightmap: 1.000000
--- !u!33 &12
MeshFilter:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 6}
  m_Mesh: {fileID: 10202, guid: 0000000000000000e000000000000000, type: 0}
--- !u!65 &13
BoxCollider:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
```

```
  m_GameObject: {fileID: 6}
  m_Material: {fileID: 0}
  m_IsTrigger: 0
  m_Enabled: 1
  importerVersion: 2
  m_Size: {x: 1.000000, y: 1.000000, z: 1.000000}
  m_Center: {x: 0.000000, y: 0.000000, z: 0.000000}
--- !u!81 &14
AudioListener:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 7}
  m_Enabled: 1
--- !u!92 &15
Behaviour:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 7}
  m_Enabled: 1
--- !u!124 &16
Behaviour:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 7}
  m_Enabled: 1
--- !u!1026 &17
HierarchyState:
  m_ObjectHideFlags: 0
  expanded: []
  selection: []
  scrollposition_x: 0.000000
  scrollposition_y: 0.000000
```

Page last updated: 2011-10-13

---

# ClassIDReference

A reference of common class ID numbers used by the YAML file format is given below, both in numerical order of class IDs and alphabetical order of class names. Note that some ranges of numbers are intentionally omitted from the sequence - these may represent classes that have been removed from the API or may be reserved for future use. Classes defined from scripts will always have class ID 114 (MonoBehaviour).

### Classes Ordered by ID Number

**1** GameObject
**2** Component
**3** LevelGameManager
**4** Transform
**5** TimeManager
**6** GlobalGameManager
**8** Behaviour
**9** GameManager
**11** AudioManager
**12** ParticleAnimator
**13** InputManager
**15** EllipsoidParticleEmitter

**17** Pipeline
**18** EditorExtension
**20** Camera
**21** Material
**23** MeshRenderer
**25** Renderer
**26** ParticleRenderer
**27** Texture
**28** Texture2D
**29** Scene
**30** RenderManager
**33** MeshFilter
**41** OcclusionPortal
**43** Mesh
**45** Skybox
**47** QualitySettings
**48** Shader
**49** TextAsset
**52** NotificationManager
**54** Rigidbody
**55** PhysicsManager
**56** Collider
**57** Joint
**59** HingeJoint
**64** MeshCollider
**65** BoxCollider
**71** AnimationManager
**74** AnimationClip
**75** ConstantForce
**76** WorldParticleCollider
**78** TagManager
**81** AudioListener
**82** AudioSource
**83** AudioClip
**84** RenderTexture
**87** MeshParticleEmitter
**88** ParticleEmitter
**89** Cubemap
**92** GUILayer
**94** ScriptMapper
**96** TrailRenderer
**98** DelayedCallManager
**102** TextMesh
**104** RenderSettings
**108** Light
**109** CGProgram
**111** Animation
**114** MonoBehaviour
**115** MonoScript
**116** MonoManager
**117** Texture3D
**119** Projector
**120** LineRenderer
**121** Flare
**122** Halo
**123** LensFlare
**124** FlareLayer
**125** HaloLayer
**126** NavMeshLayers
**127** HaloManager
**128** Font

**129** PlayerSettings
**130** NamedObject
**131** GUITexture
**132** GUIText
**133** GUIElement
**134** PhysicMaterial
**135** SphereCollider
**136** CapsuleCollider
**137** SkinnedMeshRenderer
**138** FixedJoint
**140** RaycastCollider
**141** BuildSettings
**142** AssetBundle
**143** CharacterController
**144** CharacterJoint
**145** SpringJoint
**146** WheelCollider
**147** ResourceManager
**148** NetworkView
**149** NetworkManager
**150** PreloadData
**152** MovieTexture
**153** ConfigurableJoint
**154** TerrainCollider
**155** MasterServerInterface
**156** TerrainData
**157** LightmapSettings
**158** WebCamTexture
**159** EditorSettings
**160** InteractiveCloth
**161** ClothRenderer
**163** SkinnedCloth
**164** AudioReverbFilter
**165** AudioHighPassFilter
**166** AudioChorusFilter
**167** AudioReverbZone
**168** AudioEchoFilter
**169** AudioLowPassFilter
**170** AudioDistortionFilter
**180** AudioBehaviour
**181** AudioFilter
**182** WindZone
**183** Cloth
**184** SubstanceArchive
**185** ProceduralMaterial
**186** ProceduralTexture
**191** OffMeshLink
**192** OcclusionArea
**193** Tree
**194** NavMesh
**195** NavMeshAgent
**196** NavMeshSettings
**197** LightProbeCloud
**198** ParticleSystem
**199** ParticleSystemRenderer
**205** LODGroup
**220** LightProbeGroup
**1001** Prefab
**1002** EditorExtensionImpl
**1003** AssetImporter
**1004** AssetDatabase

**1005** Mesh3DSImporter
**1006** TextureImporter
**1007** ShaderImporter
**1020** AudioImporter
**1026** HierarchyState
**1027** GUIDSerializer
**1028** AssetMetaData
**1029** DefaultAsset
**1030** DefaultImporter
**1031** TextScriptImporter
**1034** NativeFormatImporter
**1035** MonoImporter
**1037** AssetServerCache
**1038** LibraryAssetImporter
**1040** ModelImporter
**1041** FBXImporter
**1042** TrueTypeFontImporter
**1044** MovieImporter
**1045** EditorBuildSettings
**1046** DDSImporter
**1048** InspectorExpandedState
**1049** AnnotationManager
**1050** MonoAssemblyImporter
**1051** EditorUserBuildSettings
**1052** PVRImporter
**1112** SubstanceImporter

### Classes Ordered Alphabetically

**Animation** 111
**AnimationClip** 74
**AnimationManager** 71
**AnnotationManager** 1049
**AssetBundle** 142
**AssetDatabase** 1004
**AssetImporter** 1003
**AssetMetaData** 1028
**AssetServerCache** 1037
**AudioBehaviour** 180
**AudioChorusFilter** 166
**AudioClip** 83
**AudioDistortionFilter** 170
**AudioEchoFilter** 168
**AudioFilter** 181
**AudioHighPassFilter** 165
**AudioImporter** 1020
**AudioListener** 81
**AudioLowPassFilter** 169
**AudioManager** 11
**AudioReverbFilter** 164
**AudioReverbZone** 167
**AudioSource** 82
**Behaviour** 8
**BoxCollider** 65
**BuildSettings** 141
**Camera** 20
**CapsuleCollider** 136
**CGProgram** 109
**CharacterController** 143
**CharacterJoint** 144
**Cloth** 183
**ClothRenderer** 161

**Collider** 56
**Component** 2
**ConfigurableJoint** 153
**ConstantForce** 75
**Cubemap** 89
**DDSImporter** 1046
**DefaultAsset** 1029
**DefaultImporter** 1030
**DelayedCallManager** 98
**EditorBuildSettings** 1045
**EditorExtension** 18
**EditorExtensionImpl** 1002
**EditorSettings** 159
**EditorUserBuildSettings** 1051
**EllipsoidParticleEmitter** 15
**FBXImporter** 1041
**FixedJoint** 138
**Flare** 121
**FlareLayer** 124
**Font** 128
**GameManager** 9
**GameObject** 1
**GlobalGameManager** 6
**GUIDSerializer** 1027
**GUIElement** 133
**GUILayer** 92
**GUIText** 132
**GUITexture** 131
**Halo** 122
**HaloLayer** 125
**HaloManager** 127
**HierarchyState** 1026
**HingeJoint** 59
**InputManager** 13
**InspectorExpandedState** 1048
**InteractiveCloth** 160
**Joint** 57
**LensFlare** 123
**LevelGameManager** 3
**LibraryAssetImporter** 1038
**Light** 108
**LightmapSettings** 157
**LightProbeCloud** 197
**LightProbeGroup** 220
**LineRenderer** 120
**LODGroup** 205
**MasterServerInterface** 155
**Material** 21
**Mesh** 43
**Mesh3DSImporter** 1005
**MeshCollider** 64
**MeshFilter** 33
**MeshParticleEmitter** 87
**MeshRenderer** 23
**ModelImporter** 1040
**MonoAssemblyImporter** 1050
**MonoBehaviour** 114
**MonoImporter** 1035
**MonoManager** 116
**MonoScript** 115
**MovieImporter** 1044

**MovieTexture** 152
**NamedObject** 130
**NativeFormatImporter** 1034
**NavMesh** 194
**NavMeshAgent** 195
**NavMeshLayers** 126
**NavMeshSettings** 196
**NetworkManager** 149
**NetworkView** 148
**NotificationManager** 52
**OcclusionArea** 192
**OcclusionPortal** 41
**OffMeshLink** 191
**ParticleAnimator** 12
**ParticleEmitter** 88
**ParticleRenderer** 26
**ParticleSystem** 198
**ParticleSystemRenderer** 199
**PhysicMaterial** 134
**PhysicsManager** 55
**Pipeline** 17
**PlayerSettings** 129
**Prefab** 1001
**PreloadData** 150
**ProceduralMaterial** 185
**ProceduralTexture** 186
**Projector** 119
**PVRImporter** 1052
**QualitySettings** 47
**RaycastCollider** 140
**Renderer** 25
**RenderManager** 30
**RenderSettings** 104
**RenderTexture** 84
**ResourceManager** 147
**Rigidbody** 54
**Scene** 29
**ScriptMapper** 94
**Shader** 48
**ShaderImporter** 1007
**SkinnedCloth** 163
**SkinnedMeshRenderer** 137
**Skybox** 45
**SphereCollider** 135
**SpringJoint** 145
**SubstanceArchive** 184
**SubstanceImporter** 1112
**TagManager** 78
**TerrainCollider** 154
**TerrainData** 156
**TextAsset** 49
**TextMesh** 102
**TextScriptImporter** 1031
**Texture** 27
**Texture2D** 28
**Texture3D** 117
**TextureImporter** 1006
**TimeManager** 5
**TrailRenderer** 96
**Transform** 4
**Tree** 193

**TrueTypeFontImporter** 1042
**WebCamTexture** 158
**WheelCollider** 146
**WindZone** 182
**WorldParticleCollider** 76

Page last updated: 2012-01-05

---

# StreamingAssets

Most assets in Unity are combined into the project when it is built. However, it is sometimes useful to place files into the normal filesystem on the target machine to make them accessible via a pathname. An example of this is the deployment of a movie file on iOS devices; the original movie file must be available from a location in the filesystem to be played by the PlayMovie function.

Any files placed in a folder called StreamingAssets in a Unity project will be copied verbatim to a particular folder on the target machine. On a desktop computer (Mac OS or Windows) the location of the files can be obtained with the following code:-

```
path = = Application.dataPath + "/StreamingAssets";
```

On iOS, you should use:-

```
path = Application.dataPath + "/Raw";
```

...while on Android, you should use:-

```
path = "jar:file://" + Application.dataPath + "!/assets/";
```

Note that on Android, the files are contained within a compressed .jar file (which is essentially the same format as standard zip-compressed files). This means that if you do not use Unity's WWW class to retrieve the file then you will need to use additional software to see inside the .jar archive and obtain the file.

Page last updated: 2012-01-18

---

# Command Line Arguments

Typically, Unity will be launched by double-clicking its icon from the desktop but it is also possible to run it from the command line (ie, the MacOS Terminal or the Windows Command Prompt). When launched in this way, Unity can receive commands and information on startup, which can be very useful for test suites, automated builds and other production tasks.

Under MacOS, you can launch Unity from the Terminal by typing:-

```
/Applications/Unity/Unity.app/Contents/MacOS/Unity
```

...while under Windows, you should type

```
"C:\Program Files (x86)\Unity\Editor\Unity.exe"
```

...at the command prompt.

Standalone Unity games can be launched in a similar way.

### Command Line Arguments

As mentioned above, the editor and also built games can optionally be supplied with additional commands and information on startup. This is done using the following command line arguments:-

**-batchmode**

Run Unity in batch mode. This should always be used in conjunction with the other command line arguments as it ensures no pop up windows appear and eliminates the need for any human intervention. When an exception occurs during execution of script code, asset server updates fail or other operations fail Unity will immediately exit with return code 1. Note that in batch mode, Unity will send a minimal version of its log output to the console. However, the Log Files still contain the full log information.

**-quit**

Quit the Unity editor after other commands have finished executing. Note that this can cause error messages to be hidden (but they will show up in the Editor.log file).

**-buildWindowsPlayer <pathname>**

Build a standalone Windows player (eg, -buildWindowsPlayer path/to/your/build.exe).

**-buildOSXPlayer <pathname>**

Build a standalone Mac OSX player (eg, -buildOSXPlayer path/to/your/build.app).

**-buildLinux32Player <pathname>**

Build a 32-bit standalone Linux player (eg, -buildLinux32Player path/to/your/build).

**-buildLinux64Player <pathname>**

Build a 64-bit standalone Linux player (eg, -buildLinux64Player path/to/your/build).

**-importPackage <pathname>**

Import the given package. No import dialog is shown.

**-createProject <pathname>**

Create an empty project at the given path.

**-projectPath <pathname>**

Open the project at the given path.

**-logFile <pathname>**

Specify where the Editor or Windows standalone log file will be written.

**-assetServerUpdate <IP[:port] projectName username password [r <revision>]>**

Force an update of the project in the Asset Server given by *IP:port*. The port is optional and if not given it is assumed to be the standard one (10733). It is advisable to use this command in conjunction with the *-projectPath* argument to ensure you are working with the correct project. If no project name is given then the last project opened by Unity is used. If no project exists at the path given by *-projectPath* then one is created automatically.

**-exportPackage <exportAssetPath1 exportAssetPath2 ExportAssetPath3 exportFileName>**

Exports a package given a path (or set of given paths). **exportAssetPath** is a folder (relative to to the Unity project root) to export from the Unity project and **exportFileName** is the package name. Currently, this option can only export whole folders at a time. This command normally needs to be used with the **-projectPath** argument.

**-nographics (Windows only)**

When running in batch mode, do not initialize graphics device at all. This makes it possible to run your automated workflows on machines that don't even have a GPU (automated workflows only work, when you have a window in focus, otherwise you can't send simulated input commands). A standalone player generated with this option will not feature any graphics.

**-executeMethod <ClassName.MethodName>**

Execute the static method as soon as Unity is started, the project is open and after the optional asset server update has been performed. This can be used to do continous integration, perform Unit Tests, make builds, prepare some data, etc. If you want to return an error from the commandline process you can either throw an exception which will cause Unity to exit with 1 or else call EditorApplication.Exit with a non-zero code. If you want to pass parameters you can add them to the command line and retrieve them inside the method using System.Environment.GetCommandLineArgs.

To use `-executeMethod` **you need to have a script in an Editor folder and a static function in the class**.

```
// C# example
using UnityEditor;
class MyEditorScript
{
```

```
    static void PerformBuild ()
    {
        string[] scenes = { "Assets/MyScene.unity" };
        BuildPipeline.BuildPlayer(scenes, ...);
    }
}
```

```
// JavaScript example
static void PerformBuild ()
{
    string[] scenes = { "Assets/MyScene.unity" };
    BuildPipeline.BuildPlayer(scenes, ...);
}
```

**Example usage**
Execute Unity in batch mode, execute `MyEditorScript.MyMethod` method, and quit upon completion.

**Windows:**
```
C:\program files\Unity\Editor>Unity.exe -quit -batchmode -executeMethod
MyEditorScript.MyMethod
```

**Mac OS:**
```
/Applications/Unity/Unity.app/Contents/MacOS/Unity -quit -batchmode -executeMethod
MyEditorScript.MyMethod
```

Execute Unity in batch mode. Use the project path given and update from the asset server. Execute the given method after all assets have been downloaded and imported from the asset server. After the method has finished execution, automatically quit Unity.

```
/Applications/Unity/Unity.app/Contents/MacOS/Unity -batchmode -projectPath ~/UnityProjects
/AutobuildProject -assetServerUpdate 192.168.1.1 MyGame AutobuildUser l33tpa33
-executeMethod MyEditorScript.PerformBuild -quit
```

## Unity Standalone Player command line arguments
Standalone players built with Unity also understand some command line arguments:

**-batchmode**
: Run the game in "headless" mode. The game will not display anything or accept user input. This is mostly useful for running servers for networked games.

**-force-opengl (Windows only)**
: Make the game use OpenGL for rendering, even if Direct3D is available. Normally Direct3D is used but OpenGL is used if Direct3D 9.0c is not available.

**-force-d3d9 (Windows only)**
: Make the game use Direct3D 9 for rendering. This is the default, so normally there's no reason to pass it.

**-force-d3d11 (Windows only)**
: Make the game use Direct3D 11 for rendering.

**-single-instance (Linux & Windows only)**
: Allow only one instance of the game to run at the time. If another instance is already running then launching it again with `-single-instance` will just focus the existing one.

**-nolog (Windows only)**
: Do not produce output log. Normally `output_log.txt` is written in the `*_Data` folder next to the game executable, where Debug.Log output is printed.

**-force-d3d9-ref (Windows only)**
: Make the game run using Direct3D's "Reference" software renderer. The DirectX SDK has to be installed for this to work. This is mostly useful for building automated test suites, where you want to ensure rendering is exactly the same no matter what graphics card is being used.

**-adapter N (Windows only)**
Allows the game to run full-screen on another display, where N denotes the display number.

**-popupwindow (Windows only)**
The window will be created as a a pop-up window (without a frame).

**-screen-width (Linux & Windows only)**
Overrides the default screen width. This must be an integer from a supported resolution.

**-screen-height (Linux & Windows only)**
Overrides the default screen height. This must be an integer from a supported resolution.

**-screen-quality (Linux only)**
Overrides the default screen quality. Example usage would be: `/path/to/myGame -screen-quality Beautiful`

### Editor Installer

The following options can be used when installing the Unity Editor from command line:

**/S (Windows only)**
Performs a silent (no questions asked) install.

**/D=PATH (Windows only)**
Sets the default install directory. Useful when combined with the silent install option.

**Example usage**
Install Unity silently to E:\Development\Unity.

**Windows:**
`UnitySetup.exe /S /D=E:\Development\Unity`

Page last updated: 2012-11-28

---

## RunningEditorCodeOnLaunch

Sometimes, it is useful to be able to run some editor script code in a project as soon as Unity launches without requiring action from the user. You can do this by applying the **InitializeOnLoad** attribute to a class which has a **static constructor**. A static constructor is a function with the same name as the class, declared static and without a return type or parameters (see here for more information):-

```
using UnityEngine;
using UnityEditor;

[InitializeOnLoad]
public class Startup {
    static Startup()
    {
        Debug.Log("Up and running");
    }
}
```

A static constructor is always guaranteed to be called before any static function or instance of the class is used, but the InitializeOnLoad attribute ensures that it is called as the editor launches.

An example of how this technique can be used is in setting up a regular callback in the editor (its "frame update", as it were). The EditorApplication class has a delegate called update which is called many times a second while the editor is running. To have this delegate enabled as the project launches, you could use code like the following:-

```
using UnityEditor;
```

```
using UnityEngine;

[InitializeOnLoad]
class MyClass
{
   static MyClass ()
   {
      EditorApplication.update += Update;
   }

   static void Update ()
   {
      Debug.Log("Updating");
   }
}
```

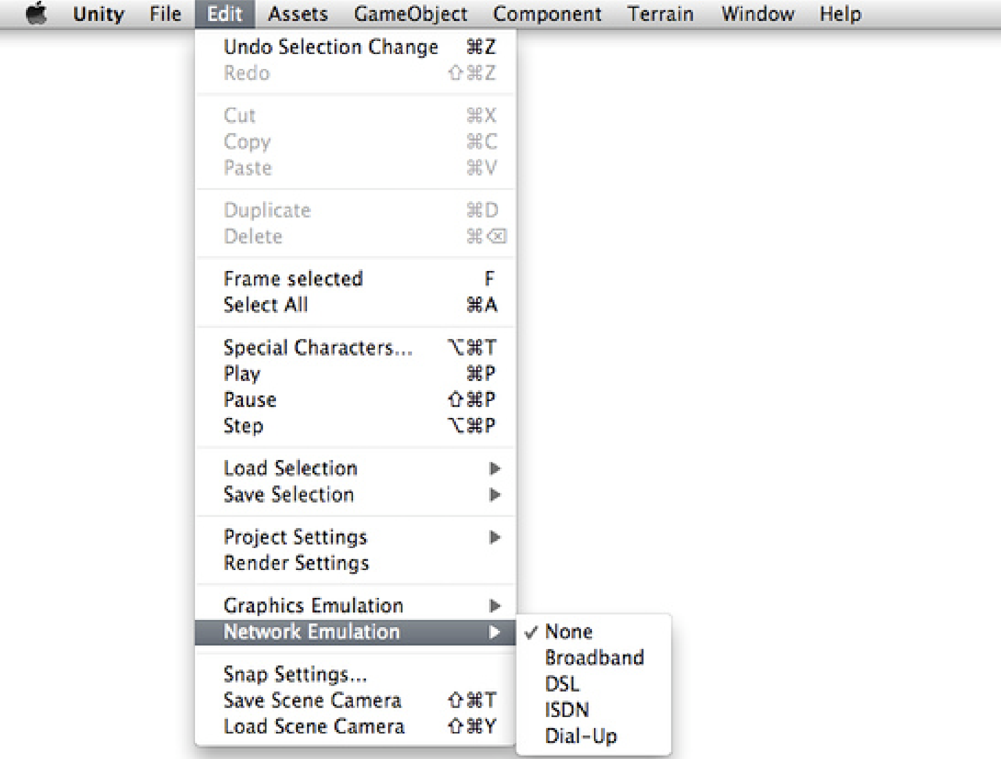
Page last updated: 2011-09-01

---

## NetworkEmulation

As part of Unity's Networking feature set, you can choose to emulate slower internet connection speeds to test out your game experience for users in low-bandwidth areas.

To enable Network emulation, go to **Edit->Network Emulation**, and choose your desired connection speed emulation.




*Enabling Network Emulation*

### Technical Details

Network emulation delays the sending of packets in networking traffic for the Network and NetworkView classes. The ping is artificially inflated for all options, the inflation value increasing as emulated connection speed gets slower. On the **Dial-Up** setting, packet dropping and variance is also introduced to simulate the worst possible connection ever. Emulation will persist whether you are serving the role of Server or Client.

Network emulation only affects the Network and NetworkView classes, and will not alter or emulate specialized networking code written using .NET sockets.

Page last updated: 2008-04-30

---

# Security Sandbox

## Desktop

In Unity 3.0, the webplayer implements a security model very similar to the one used by the Adobe Flash player�. This security restrictions apply only to the webplayer, and to the editor when the active build target is WebPlayer. The security model has several parts:

- Restrictions on accessing data on a domain other than the one hosting your .unity3d file.
- Some limitation on the usage of the Sockets.
- Disallowing invocation of any method we deemed off limits. (things like File.Delete, etc).
- Disallowing the usage of System.Reflection.* to call private/internal methods in classes you did not write yourself.

Currently only the first two parts of the security model are emulated in the Editor. Look here for a detailed list of which methods / classes are available in the webplayer.

The builtin mutiplayer networking functionality of Unity (`UnityEngine.Network`, `UnityEngine.NetworkView` classes etc) is not affected.

### This document describes how to make sure your content keeps working with version 3.0 of the Unity webplayer.

- See the Unity API reference for information about the WWW class.
- See the .NET API reference for information about the .NET Socket class.

The WWW class and sockets use the same policy schema but besides that they are completely separate systems. The WWW policy only defines permissions on the web service where the policy is hosted but socket policies apply to all TCP/UDP socket connections.

The Unity editor comes with an "Emulate Web Security" feature, that imposes the webplayer's security model. This makes it easy to detect problems from the comfort of the editor. You can find this setting in **Edit->Project Settings->Editor**.

## Implications for usage of the WWW class

The Unity webplayer expects a http served policy file named "**crossdomain.xml**" to be available on the domain you want to access with the WWW class, (although this is not needed if it is the same domain that is hosting the unity3d file).

For example, imagine a tetris game, hosted at the following url:

*http://gamecompany.com/games/tetris.unity3d*

needs to access a highscore list from the following url:

*http://highscoreprovider.net/gethighscore.php*

In this case, you would need to place a **crossdomain.xml** file at the root of the *highscoreprovider.net* domain like this: *http://highscoreprovider.net/crossdomain.xml*

The contents of the **crossdomain.xml** file are in the format used by the Flash player. It is very likely that you'll find the **crossdomain.xml** file already in place. The policy in the file look like this:

```
<?xml version="1.0"?>
```

```
<cross-domain-policy>
<allow-access-from domain="*"/>
</cross-domain-policy>
```

When this file is placed at http://highscoreprovider.net/crossdomain.xml, the owner of that domain declares that the contents of the webserver may be accessed by any webplayer coming from any domain.

The Unity webplayer does not support the `<allow-http-request-headers-from domain>` and `<site-control permitted-cross-domain-policies>` tags. Note that **crossdomain.xml** should be an ASCII file.

## Implications for usage of Sockets:

A Unity webplayer needs a socket served policy in order to connect to a particular host. This policy is by default hosted by the target host on port **843** but it can be hosted on other ports as well. The functional difference with a non-default port is that it must be manually fetched with Security.PrefetchSocketPolicy() API call and if it is hosted on a port higher than 1024 the policy can only give access to other ports higher than 1024.

When using the default port it works like this: A Unity webplayer tries to make a TCP socket connection to a host, it first checks that the host server will accept the connection. It does this by opening a TCP socket on port 843, issues a request, and expects to receive a socket policy over the new connection. The Unity webplayer then checks that the host's policy permits the connection to go ahead and it will proceed without error if so. This process happens transparently to the user's code, which does not need to be modified to use this security model. An example of a socket policy look like this:

```
<?xml version="1.0"?>
<cross-domain-policy>
   <allow-access-from domain="*" to-ports="1200-1220"/>
</cross-domain-policy>"
```

This policy effectively says "Content from any domain is free to make socket connections at ports 1200-1220". The Unity webplayer will respect this, and reject any attempted socket connection using a port outside that range (a SecurityException will be thrown).

When using UDP connections the policy can also be auto fetched when they need to be enforced in a similar manner as with TCP. The difference is that auto fetching with TCP happens when you Connect to something (ensures you are allowed to connect to a server), but with UDP, since it's connectionless, it also happens when you call any API point which sends or receives data (ensures you are allowed to send/receive traffic to/from a server).

The format used for the socket policy is the same as that used by the Flash player except some tags are not supported. The Unity webplayer only supports "*" as a valid value for the domain setting and the "to-ports" setting is mandatory.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT cross-domain-policy (allow-access-from*)>

<!ELEMENT allow-access-from EMPTY>
<!ATTLIST allow-access-from domain CDATA #REQUIRED>
<!ATTLIST allow-access-from to-ports CDATA #REQUIRED>
```

The socket policy applies to both TCP and UDP connection types so both UDP and TCP traffic can be controlled by one policy server.

For your convenience, we provide a small program which simply listens at port 843; when on a connection it receives a request string, it will reply with a valid socket policy. The server code can be found inside the Unity install folder, in Data/Tools /SocketPolicyServer on Windows or /Unity.app/Contents/Tools/SocketPolicyServer on OS X. Note that the pre-built executable can be run on Mac since it is a Mono executable. Just type "mono sockpol.exe" to run it. Note that this example code shows the correct behaviour of a socket policy server. Specifically the server expects to receive a zero-terminated string that contains `<policy-file-request/>`. It only sends to the client the socket policy xml document when this string (and exactly this string) has been received. Further, it is required that the xml header and xml body are sent with a single socket write. Breaking

the header and body into separate socket write operations can cause security exceptions due to Unity receiving an incomplete policy. If you experience any problems with your own server please consider using the example that we provide. This should help you diagnose whether you have server or network issues.

Third party networking libraries, commonly used for multiplayer game networking, should be able to work with these requirements as long as they do not depend on peer 2 peer functionality (see below) but utilize dedicated servers. These sometimes even come out of the box with support for hosting policies.

**Note:** Whilst the `crossdomain.xml` and socket policy files are both xml documents and are broadly similar, the way that these documents are served are very different. `Crossdomain.xml` (which applied to http requests) is fetched using http on port 80, where-as the socket policy is fetched from port 843 using a trivial server that implements the `<policy-file-request/>`. You cannot use an http server to issue the socket policy file, nor set up a server that simply sends the socket policy file in response to a socket connection on port 843. Note also that each server you connect to requires its own socket policy server.

**Debugging**

You can use `telnet` to connect to the socket policy server. An example session is shown below:

```
host$ telnet localhost 843
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
<policy-file-request/>
<?xml version='1.0'?>
<cross-domain-policy>
        <allow-access-from domain="*" to-ports="*" />
</cross-domain-policy>Connection closed by foreign host.
host$
```

In this example session, telnet is used to connect to the localhost on port 843. Telnet responds with the first three lines, and then sits waiting for the user to enter something. The user has entered the policy request string `<policy-file-request/>`, which the socket policy server receives and responds with the socket policy. The server then disconnects causing telnet to report that the connection has been closed.

## Listening sockets

You **cannot** create listening sockets in the webplayer, it cannot act as a server. Therefore webplayers cannot communicate with each other directly (peer 2 peer). When using TCP sockets you can only connect to remote endpoints provided it is allowed through the socket policy system. For UDP it works the same but the concept is a little bit different as it is a connectionless protocol, you don't have to connect/listen to send/receive packets. It works by enforcing that you can only receive packets from a server if he has responded first with a valid policy with the `allow-access-from domain` tag.

## This is all just so annoying, why does all this stuff exist?

The socket and WWW security features exist to protect people who install the Unity Web Player. Without these restrictions, an attack such as the following would be possible:

- Bob works at the white house.
- Frank is evil. He writes a unity webgame that pretends to be a game, but in the background does a WWW request to http://internal.whitehouse.gov/LocationOfNuclearBombs.pdf. internal.whitehouse.gov is a server that is not reachable from the internet, but is reachable from Bob's workstation because he works at the white house.
- Frank sends those pdf bytes to http://frank.com/secretDataUploader.php
- Frank places this game on http://www.frank.com/coolgame.unity3d
- Frank somehow convinces Bob to play the game.
- Bob plays the game.
- Game silently downloads the secret document, and sends it to Frank.

With the WWW and socket security features, this attack will fail, because before downloading the pdf, unity checks http://internal.whitehouse.gov/crossdomain.xml, with the intent to ask that server: "is the data you have on your server available for public usage?". Placing a crossdomain.xml on a webserver can be seen as the response to that question. In the case of this example, the system operator of internal.whitehouse.gov will not place a crossdomain.xml on its server, which will lead Unity to not download the pdf.

Unfortunately, in order to protect the people who install the Unity Web Player, people who develop in Unity need to take these security measures into account when developing content. The same restrictions are present in all major plugin technologies. (Flash, Silverlight, Shockwave)

### Exceptions

In order to find the right balance between protecting Web Player users and making life of content developers easy, we have implemented an exception to the security mechanism described above:

- You are allowed to download images from servers that do not have a crossdomain.xml file. However, the only thing you are allowed to do with these images is use them as textures in your scene. You are not allowed to use GetPixel() on them. You are also no longer allowed to read back from the screen. Both attempts will result in a SecurityException being thrown. The reasoning is here is that it's okay to download the image, as long as the content developer gets no access to it. So you can display it to the user, but you cannot send the bytes of the image back to some other server.

Page last updated: 2012-07-25

---

## VisualStudioIntegration

### What does this feature get me?

A more sophisticated C# development environment.
Think smart autocompletion, computer-assisted changes to source files, smart syntax highlighting and more.

### What's the difference between Express and Pro?

VisualStudio C# 2010 is a product from Microsoft. It comes in an Express and a Profesional edition.
The Express edition is free, and you can download it from here: http://www.microsoft.com/express/vcsharp/
The Professional edition is not free, you can find out more information about it here: http://www.microsoft.com/visualstudio/en-us/products/professional/default.mspx

Unity's VisualStudio integration has two components:
1) Unity creating and maintaining VisualStudio project files. **Works with Express and with Profesional.**
2) Unity automatically opening VisualStudio when you doubleclick on a script, or error in Unity. **Works with Professional only.**

### I've got Visual Studio Express, how do I use it?

- In Unity, select from the menu **Assets->Sync VisualStudio Project**
- Find the newly created .sln file in your Unity project (one folder up from your Assets folder)
- Open that file with Visual Studio Express.
- You can now edit all your script files, and switch back to Unity to use them.

### I've got Visual Studio Profesional, how do I use it?

- In Unity, go to Edit->Preferences, and make sure that Visual Studio is selected as your preferred external editor.
- Doubleclick a C# file in your project. Visual Studio should automatically open that file for you.
- You can edit the file, save, and switch back to Unity.

### A few things to watch out for:

- Even though Visual Studio comes with its own C# compiler, and you can use it to check if you have errors in your c# scripts, Unity still uses its own C# compiler to compile your scripts. Using the Visual Studio compiler is still quite useful, because it means you don't have to switch to Unity all the time to check if you have any errors or not.
- Visual Studio's C# compiler has some more features than Unity's C# compiler currently has. This means that some code (especially newer c# features) will not give an error in Visual Studio but will give an error in Unity.
- Unity automatically creates and maintains a Visual Studio .sln and .csproj file. Whenever somebody adds/renames/moves /deletes a file from within Unity, Unity regenerates the .sln and .csproj files. You can add files to your solution from Visual Studio as well. Unity will then import those new files, and the next time Unity creates the project files again, it will create them with this new file included.
- Unity does not regenerate the Visual Studio project files after an AssetServer update, or a SVN update. You can manually ask Unity to regenerate the Visual Studio project files trough the menu: **Assets->Sync VisualStudio Project**

Page last updated: 2011-08-03

---

# ExternalVersionControlSystemSupport

Unity offers an Asset Server add-on product for easy integrated versioning of your projects. If you for some reason are not able use the Unity Asset Server, it is possible to store your project in any other version control system, such as Subversion, Perforce or Bazaar. This requires some initial manual setup of your project.

Before checking your project in, you have to tell Unity to modify the project structure slightly to make it compatible with storing assets in an external version control system. This is done by selecting **Edit->Project Settings->Editor** in the application menu and enabling External Version Control support by selecting **Metafiles** in the dropdown for Version Control. This will create a text file for every asset in the `Assets` directory containing the necessary bookkeeping information required by Unity. The files will have a `.meta` file extension with the first part being the full file name of the asset it is associated with. Moving and renaming assets within Unity should also update the relevant `.meta` files. However, if you move or rename assets from an external tool, make sure to syncronize the relevant `.meta` files as well.

When checking the project into a version control system, you should add the `Assets` and the `ProjectSettings` directories to the system. The `Library` directory should be completely ignored - when using external version control, it's only a local cache of imported assets.

When creating new assets, make sure both the asset itself and the associated `.meta` file is added to version control.

### Example: Creating a new project and importing it to a Subversion repository.

First, let's assume that we have a subversion repository at `svn://my.svn.server.com/` and want to create a project at `svn://my.svn.server.com/MyUnityProject`. Then follow these steps to create the initial import in the system:

1. Create a new project inside Unity and lets call it `InitialUnityProject`. You can add any initial assets here or add them later on.
2. Enable **Meta files** in **Edit->Project Settings->Editor**
3. Quit Unity (We do this to assure that all the files are saved).
4. Delete the `Library` directory inside your project directory.
5. Import the project directory into Subversion. If you are using the command line client, this is done like this from the directory where your initial project is located:
   ```
   svn import -m"Initial project import" InitialUnityProject svn://my.svn.server.com/MyUnityProject
   ```
   If successful, the project should now be imported into subversion and you can delete the `InitialUnityProject` directory if you wish.
6. Check out the project back from subversion
   ```
   svn co svn://my.svn.server.com/MyUnityProject
   ```
   And check that the `Assets` and `ProjectSettings` directory are versioned.
7. Open the checked out project with Unity by launching it while holding down the **Option** or the left **Alt** key. Opening the project will recreate the `Library` directory in step 4 above.
8. **Optional:** Set up an ignore filter for the unversioned `Library` directory:
   ```
   svn propedit svn:ignore MyUnityProject/
   ```
   Subversion will open a text editor. Add the Library directory.
9. Finally commit the changes. The project should now be set up and ready:
   ```
   svn ci -m"Finishing project import" MyUnityProject
   ```

Page last updated: 2012-09-18

---

# Analytics

The Unity editor is configured to send anonymous usage data back to Unity. This information is used to help improve the

features of the editor. The analytics are collected using Google Analytics. Unity makes calls to a URI hosted by Google. The URN part of the URI contains details that describe what editor features or events have been used.

### Examples of collected data

The following are examples of data that Unity might collect.

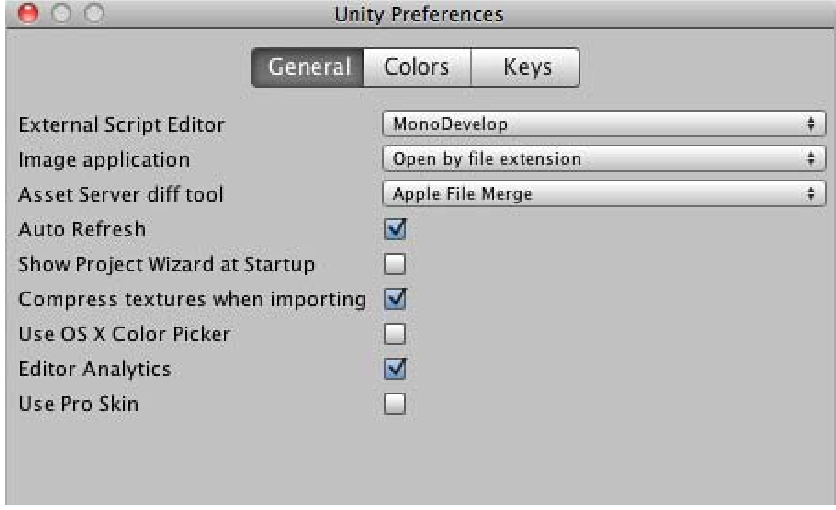
**Which menu items have been used.** If some menu items are used rarely or not at all we could in the future simplify the menuing system.

**Build times.** By collecting how long builds take to make we can focus engineering effort on optimizing the correct code.

**Lightmap baking.** Again, timing and reporting how long it takes for light maps to bake can help us decide how much effort to spend on optimizing this area.

### Disabling Analytics

If you do not want to send anonymous data to Unity then the sending of Analytics can be disabled. To do this untick the box in the Unity Preferences General tab.



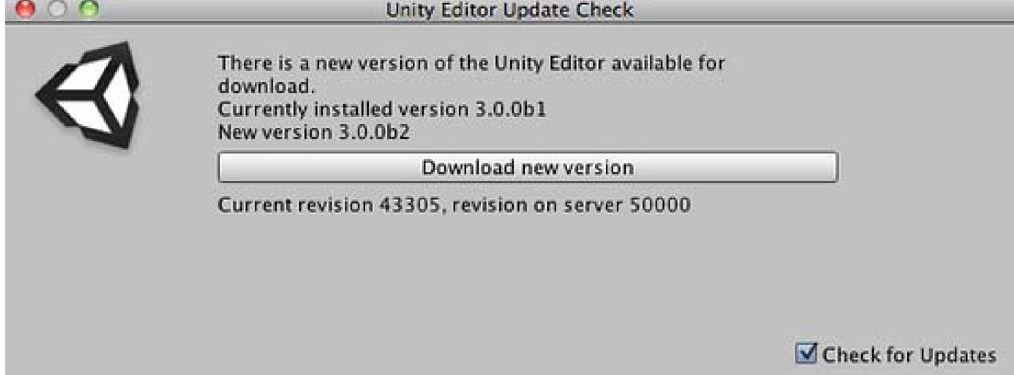

*Editor analytics in the preferences pane.*

Page last updated: 2010-09-10

---

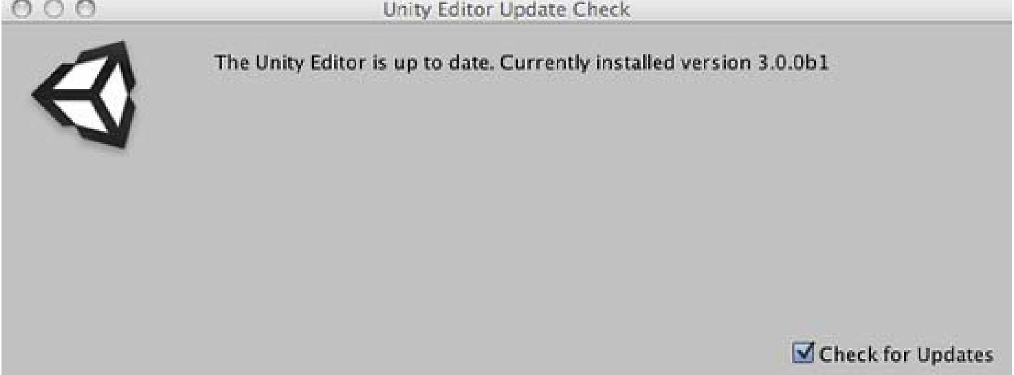
## Version Check

Unity checks whether updates are available. This check happens either when Unity is started, or when you choose the Help->Check for Updates menu item. The update check sends the current Unity revision number (the five digit number that appears in brackets after the version name in the About Unity dialog) to the update server where is it compared with the most-up-to-date released version. If a newer version of Unity is available the following dialog is shown:




*Window displayed when there is a newer version of Unity available for download.*

If the version in use is the most up-to-date then the following dialog is shown:




*Window displayed when Unity is updated to the latest version.*

Click the Download new version button to be taken to the website where you can download the new version.

### Update Check Frequency

The response from the server also contains a time interval which suggests when the next update check should be made. This allows the update check to be made less frequently when Unity is not expecting updates to be made available.

### Skipping Update Versions

If you are in the middle of a project you may not want to update to a new version of Unity. Ticking the Skip this version button on the Unity Editor Update Check dialog will prevent Unity from telling you about this update.

### Disabling the Update Check

It is not possible to disable the check for updates. The Check For Updates tick box on the dialog controls whether you are notified of updates (if they are available) when Unity starts. Even if you have unticked the Check for Updates option you can still check for updates by using the Help->Check for Updates menu item.

Page last updated: 2010-09-07

---

## Installing Multiple Versions of Unity

You can install more than one version of Unity on your machine as long as you follow the correct naming conventions for your folders. You need to rename each of the Unity folders themselves, so that the hierarchy looks like:

```
Unity_3.4.0
---Editor
---MonoDevelop
Unity_4.0b7
---Editor
---MonoDevelop
```

### PC

- Install Unity 4.0 (www.unity3d.com/download)
- When you install on PC it will select the previously installed directory - do not install here
- Create a new directory named sensibly e.g. Unity_4
- Name any shortcuts so you know which version you are launching
- Hold alt when you launch the beta to force unity to let you choose which project to open (otherwise it will try and upgrade the last opened project)
- Choose your projectname_4 directory to open your backed up project

**Do not** rename each Editor folder inside a single Unity folder! You will overwrite the MonoDevelop folder and this will cause **serious** stability problems and unexpected crashes.

### Mac

- Find your existing Unity application folder and rename appropriately e.g Unity35
- Install the Unity 4.0 (www.unity3d.com/download)
- Name any shortcuts so you know which version you are launching
- Hold alt when you launch the beta to force unity to let you choose which project to open (otherwise it will try and upgrade the last opened project)
- Choose your projectname_4 directory to open your backed up project

Page last updated: 2012-11-16

---

# TroubleShooting

This section addresses common problems that can arise when using Unity. Each platform is dealt with separately below.

## Desktop

### In MonoDevelop, the Debug button is greyed out!

- This means that MonoDevelop was unable to find the Unity executable. In the MonoDevelop preferences, go to the Unity/Debugger section and then browse to where your Unity executable is located.

### Is there a way to get rid of the welcome page in MonoDevelop?

- Yes. In the MonoDevelop preferences, go to the Visual Style section, and uncheck "Load welcome page on startup".

### Geforce 7300GT on OSX 10.6.4

- Deferred rendering is disabled because materials are not displayed correctly for Geforce 7300GT on OX 10.6.4; This happens because of buggy video drivers.

### On Windows x64, Unity crashes when my script throws a NullReferenceException

- Please apply Windows Hotfix #976038.

### Graphics

#### Slow framerate and/or visual artifacts.

- This may occur if your video card drivers are not up to date. Make sure you have the latest official drivers from your card vendor.

### Shadows

#### I see no shadows at all!

- Shadows are a **Unity Pro** only feature, so without Unity Pro you won't get shadows. Simpler shadow methods, like using a Projector, are still possible, of course.
- Shadows also require certain graphics hardware support. See Shadows page for details.
- Check if shadows are not completely disabled in Quality Settings.
- **Shadows are currently not supported for Android and iOS mobile platforms.**

#### Some of my objects do not cast or receive shadows

An object's Renderer must have **Receive Shadows** enabled for shadows to be rendered onto it. Also, an object must have **Cast Shadows** enabled in order to cast shadows on other objects (both are on by default).

Only opaque objects cast and receive shadows. This means that objects using the built-in Transparent or Particle shaders will not cast shadows. In most cases it is possible to use Transparent Cutout shaders for objects like fences, vegetation, etc. If you use custom written Shaders, they have to be pixel-lit and use the Geometry render queue. Objects using **VertexLit** shaders do not receive shadows but are able to cast them.

Only **Pixel lights** cast shadows. If you want to make sure that a light always casts shadows no matter how many other lights are in the scene, then you can set it to **Force Pixel** render mode (see the Light reference page).

## iOS

# Troubleshooting on iOS devices

There are some situations with iOS where your game can work perfectly in the Unity editor but then doesn't work or maybe doesn't even start on the actual device. The problems are often related to code or content quality. This section describes the most common scenarios.

## The game stops responding after a while. Xcode shows "interrupted" in the status bar.

There are a number of reasons why this may happen. Typical causes include:

1. Scripting errors such as using uninitialized variables, etc.
2. Using 3rd party Thumb compiled native libraries. Such libraries trigger a known problem in the iOS SDK linker and might cause random crashes.
3. Using generic types with value types as parameters (eg, List<int>, List<SomeStruct>, List<SomeEnum>, etc) for serializable script properties.
4. Using reflection when managed code stripping is enabled.
5. Errors in the native plugin interface (the managed code method signature does not match the native code function signature).

Information from the XCode Debugger console can often help detect these problems (Xcode menu: **View > Debug Area > Activate Console**).

## The Xcode console shows "Program received signal: “SIGBUS” or EXC_BAD_ACCESS error.

This message typically appears on iOS devices when your application receives a NullReferenceException. There two ways to figure out where the fault happened:

### Managed stack traces

Since version 3.4 Unity includes software-based handling of the NullReferenceException. The AOT compiler includes quick checks for null references each time a method or variable is accessed on an object. This feature affects script performance which is why it is enabled only for development builds (for basic license users it is enough to enable the "development build" option in the Build Settings dialog, while iOS pro license users additionally need to enable the "script debugging" option). If everything was done right and the fault actually is occurring in .NET code then you won't see EXC_BAD_ACCESS anymore. Instead, the .NET exception text will be printed in the Xcode console (or else your code will just handle it in a "catch" statement). Typical output might be:

```
Unhandled Exception: System.NullReferenceException: A null value was found where an object instance was required.
  at DayController+$handleTimeOfDay$121+$.MoveNext () [0x0035a] in DayController.js:122
```

This indicates that the fault happened in the handleTimeOfDay method of the DayController class, which works as a coroutine. Also if it is script code then you will generally be told the exact line number (eg, "DayController.js:122"). The offending line might be something like the following:

```
 Instantiate(_imgwww.assetBundle.mainAsset);
```

This might happen if, say, the script accesses an asset bundle without first checking that it was downloaded correctly.

### Native stack traces

Native stack traces are a much more powerful tool for fault investigation but using them requires some expertise. Also, you generally can't continue after these native (hardware memory access) faults happen. To get a native stack trace, type **bt all** into the Xcode Debugger Console. Carefully inspect the printed stack traces - they may contain hints about where the error occurred. You might see something like:

```
...
Thread 1 (thread 11523):
#0 0x006267d0 in m_OptionsMenu_Start ()
#1 0x002e4160 in wrapper_runtime_invoke_object_runtime_invoke_void__this___object_intptr_intptr_intptr ()
#2 0x00a1dd64 in mono_jit_runtime_invoke (method=0x18b63bc, obj=0x5d10cb0, params=0x0, exc=0x2fffdd34) at /Users/m
#3 0x0088481c in MonoBehaviour::InvokeMethodOrCoroutineChecked ()
...
```

First of all you should find the stack trace for **"Thread 1"**, which is the main thread. The very first lines of the stack trace will point to the place where the error occurred. In this example, the trace indicates that the NullReferenceException happened inside the *"OptionsMenu"* script's *"Start"* method. Looking carefully at this method implementation would reveal the cause of the problem. Typically, NullReferenceExceptions happen inside the **Start** method when incorrect assumptions are made about initialization order. In some cases only a partial stack trace is seen on the Debugger Console:

```
Thread 1 (thread 11523):
#0 0x0062564c in start ()
```

This indicates that native symbols were stripped during the Release build of the application. The full stack trace can be obtained with the following procedure:

- Remove application from device.
- Clean all targets.
- Build and run.
- Get stack traces again as described above.

### EXC_BAD_ACCESS starts occurring when an external library is linked to the Unity iOS application.

This usually happens when an external library is compiled with the ARM Thumb instruction set. Currently such libraries are not compatible with Unity. The problem can be solved easily by recompiling the library without Thumb instructions. You can do this for the library's Xcode project with the following steps:

- in Xcode, select *"View"* > *"Navigators"* > *"Show Project Navigator"* from the menu
- select the *"Unity-iPhone"* project, activate *"Build Settings"* tab
- in the search field enter : *"Other C Flags"*
- add *-mno-thumb* flag there and rebuild the library.

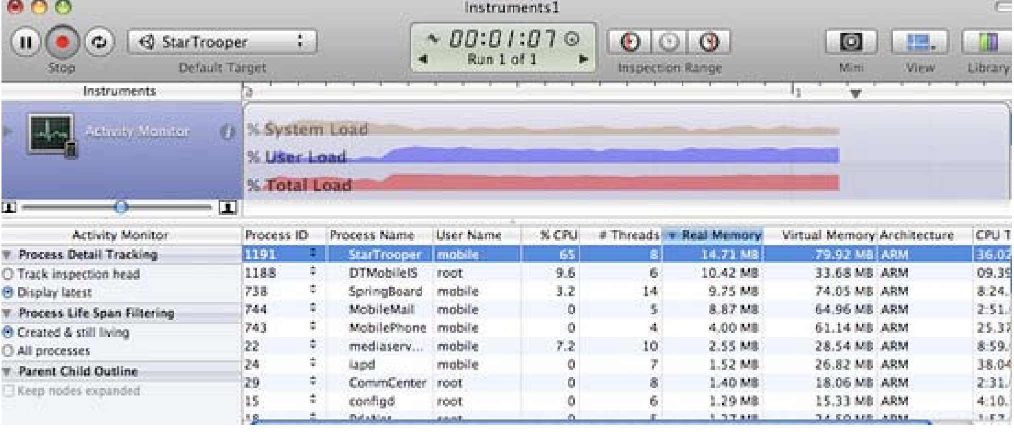
If the library source is not available you should ask the supplier for a non-thumb version of the library.

### The Xcode console shows "WARNING -> applicationDidReceiveMemoryWarning()" and the application crashes immediately afterwards

(Sometimes you might see a message like *Program received signal: �0�*.) This warning message is often not fatal and merely indicates that iOS is low on memory and is asking applications to free up some memory. Typically, background processes like Mail will free some memory and your application can continue to run. However, if your application continues to use memory or ask for more, the OS will eventually start killing applications and yours could be one of them. Apple does not document what memory usage is safe, but empirical observations show that applications using less than 50% MB of all device RAM (like ~200-256 MB for 2nd generation ipad) do not have major memory usage problems. The main metric you should rely on is how much RAM your application uses. Your application memory usage consists of three major components:

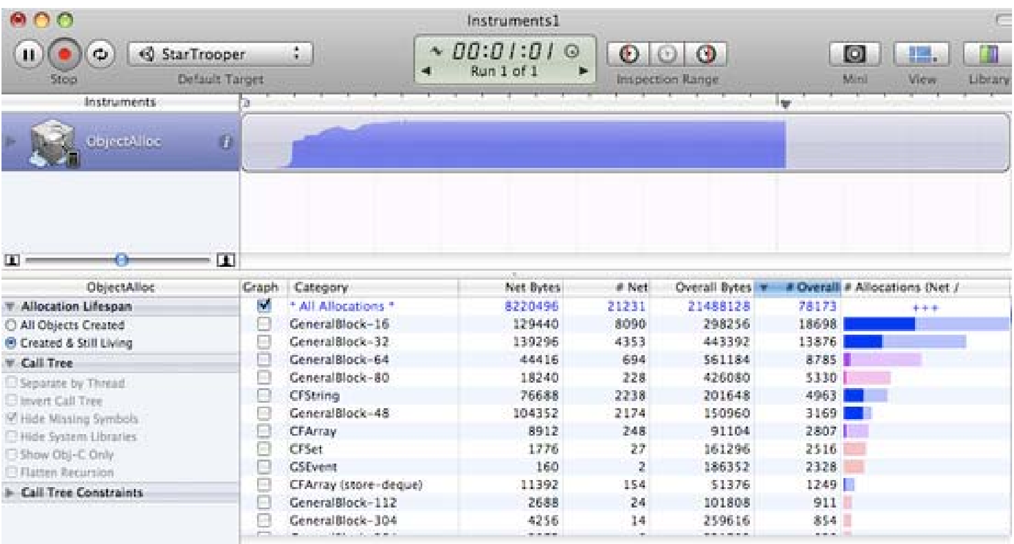
- application code (the OS needs to load and keep your application code in RAM, but some of it might be discarded if really needed)
- native heap (used by the engine to store its state, your assets, etc. in RAM)
- managed heap (used by your Mono runtime to keep C# or JavaScript objects)
- GLES driver memory pools: textures, framebuffers, compiled shaders, etc.

Your application memory usage can be tracked by two Xcode Instruments tools: **Activity Monitor**, **Object Allocations** and **VM Tracker**. You can start from the Xcode Run menu: **Product > Profile** and then select specific tool. **Activity Monitor** tool shows all process statistics including **Real memory** which can be regarded as the total amount of RAM used by your application. **Note:** OS and device HW version combination might noticeably affect memory usage numbers, so you should be careful when comparing numbers obtained on different devices.




**Note:** The internal profiler shows only the heap allocated by .NET scripts. Total memory usage can be determined via Xcode Instruments as shown above. This figure includes parts of the application binary, some standard framework buffers, Unity engine internal state buffers, the .NET runtime heap (number printed by internal profiler), GLES driver heap and some other miscellaneous stuff.

The other tool displays all allocations made by your application and includes both native heap and managed heap statistics (don't forget to check the **Created and still living** box to get the current state of the application). The important statistic is the **Net bytes** value.




To keep memory usage low:

- Reduce the application binary size by using the strongest iOS stripping options (Advanced license feature), and avoid unnecessary dependencies on different .NET libraries. See the player settings and player size optimization manual pages for further details.
- Reduce the size of your content. Use PVRTC compression for textures and use low poly models. See the manual page about reducing file size for more information.
- Don't allocate more memory than necessary in your scripts. Track mono heap size and usage with the internal profiler
- **Note:** with Unity 3.0, the scene loading implementation has changed significantly and now all scene assets are preloaded. This results in fewer hiccups when instantiating game objects. If you need more fine-grained control of asset loading and unloading during gameplay, you should use Resources.Load and Object.Destroy.

Querying the OS about the amount of free memory may seem like a good idea to evaluate how well your application is performing. However, the free memory statistic is likely to be unreliable since the OS uses a lot of dynamic buffers and caches. The only reliable approach is to keep track of memory consumption for your application and use that as the main metric. Pay attention to how the graphs from the tools described above change over time, especially after loading new levels.

### The game runs correctly when launched from Xcode but crashes while loading the first level when launched manually on the device.

There could be several reasons for this. You need to inspect the device logs to get more details. Connect the device to your

Mac, launch Xcode and select **Window > Organizer** from the menu. Select your device in the Organizer's left toolbar, then click on the "Console" tab and review the latest messages carefully. Additionally, you may need to investigate crash reports. You can find out how to obtain crash reports here: http://developer.apple.com/iphone/library/technotes/tn2008/tn2151.html.

### The Xcode Organizer console contains the message "killed by SpringBoard".

There is a poorly-documented time limit for an iOS application to render its first frames and process input. If your application exceeds this limit, it will be killed by SpringBoard. This may happen in an application with a first scene which is too large, for example. To avoid this problem, it is advisable to create a small initial scene which just displays a splash screen, waits a frame or two with **yield** and then starts loading the real scene. This can be done with code as simple as the following:

```
function Start () {
    yield;
    Application.LoadLevel("Test");
}
```

### Type.GetProperty() / Type.GetValue() cause crashes on the device

Currently **Type.GetProperty()** and **Type.GetValue()** are supported only for the **.NET 2.0 Subset** profile. You can select the .NET API compatibility level in the Player Settings.

**Note: Type.GetProperty()** and **Type.GetValue()** might be incompatible with managed code stripping and might need to be excluded (you can supply a custom non-strippable type list during the stripping process to accomplish this). For further details, see the iOS player size optimization guide.

### The game crashes with the error message "ExecutionEngineException: Attempting to JIT compile method 'SometType`1<SomeValueType>:.ctor ()' while running with --aot-only."

The Mono .NET implementation for iOS is based on AOT (ahead of time compilation to native code) technology, which has its limitations. It compiles only those generic type methods (where a value type is used as a generic parameter) which are explicitly used by other code. When such methods are used only via reflection or from native code (ie, the serialization system) then they get skipped during AOT compilation. The AOT compiler can be hinted to include code by adding a dummy method somewhere in the script code. This can refer to the missing methods and so get them compiled ahead of time.

```
void _unusedMethod()
{
    var tmp = new SomeType<SomeValueType>();
}
```

**Note:** value types are basic types, enums and structs.

### Various crashes occur on the device when a combination of System.Security.Cryptography and managed code stripping is used

.NET Cryptography services rely heavily on reflection and so are not compatible with managed code stripping since this involves static code analysis. Sometimes the easiest solution to the crashes is to exclude the whole **System.Security.Crypography** namespace from the stripping process.

The stripping process can be customized by adding a custom **link.xml** file to the **Assets** folder of your Unity project. This specifies which types and namespaces should be excluded from stripping. Further details can be found in the iOS player size optimization guide.

**link.xml**

```
<linker>
       <assembly fullname="mscorlib">
              <namespace fullname="System.Security.Cryptography" preserve="all"/>
       </assembly>
</linker>
```

**Application crashes when using System.Security.Cryptography.MD5 with managed code stripping**

You might consider advice listed above or can work around this problem by adding extra reference to specific class to your script code:

```
object obj = new MD5CryptoServiceProvider();
```

### "Ran out of trampolines of type 1/2" runtime error

This error usually happens if you use lots of recursive generics. You can hint to the AOT compiler to allocate more trampolines of type 1 or type 2. Additional AOT compiler command line options can be specified in the "Other Settings" section of the Player Settings. For type 1 trampolines, specify **nrgctx-trampolines=ABCD**, where ABCD is the number of new trampolines required (i.e. 4096). For type 2 trampolines specify **nimt-trampolines=ABCD**.

### After upgrading Xcode Unity iOS runtime fails with message "You are using Unity iPhone Basic. You are not allowed to remove the Unity splash screen from your game"

With some latest Xcode releases there were changes introduced in PNG compression and optimization tool. These changes might cause false positives in Unity iOS runtime checks for splash screen modifications. If you encounter such problems try upgrading Unity to the latest publicly available version. If it does not help you might consider following workaround:

- Replace your Xcode project from scratch when building from Unity (instead of appending it)
- Delete already installed project from device
- Clean project in Xcode (*Product->Clean*)
- Clear Xcode's Derived Data folders (*Xcode->Preferences->Locations*)

If this still does not help try disabling PNG re-compression in Xcode:

- Open your Xcode project
- Select "Unity-iPhone" project there
- Select "Build Settings" tab there
- Look for "Compress PNG files" option and set it to NO

### App Store submission fails with "iPhone/iPod Touch: application executable is missing a required architecture. At least one of the following architecture(s) must be present: armv6" message

You might get such message when updating already existing application, which previously was submitted with armv6 support. Unity 4.x and Xcode 4.5 does not support armv6 platform anymore. To solve submission problem just set **Target OS Version** in Unity **Player Settings** to **4.3** or higher.

### WWW downloads are working fine in Unity Editor and on Android, but not on iOS

Most common mistake is to assume that WWW downloads are always happening on separate thread. On some platforms this might be true, but you should not take it for granted. Best way to track WWW status is either to use *yield* statement or check status in *Update* method. You should **not** use busy *while* loops for that.

### "PlayerLoop called recursively!" error occurs when using Cocoa via a native function called from a script

Some operations with the UI will result in iOS redrawing the window immediately (the most common example is adding a UIView with a UIViewController to the main UIWindow). If you call a native function from a script, it will happen inside Unity's PlayerLoop, resulting in PlayerLoop being called recursively. In such cases, you should consider using the performSelectorOnMainThread method with waitUntilDone set to false. It will inform iOS to schedule the operation to run between Unity's PlayerLoop calls.

### Profiler or Debugger unable to see game running on iOS device

- Check that you have built a Development build, and ticked the "Enable Script Debugging" and "Autoconnect profiler" boxes (as appropriate).
- The application running on the device will make a multicast broadcast to 225.0.0.222 on UDP port 54997. Check that your network settings allow this traffic. Then, the profiler will make a connection to the remote device on a port in the range 55000 - 55511 to fetch profiler data from the device. These ports will need to be open for UDP access.

### Missing DLLs

If your application runs ok in editor but you get errors in your iOS project this may be caused by missing DLLs (e.g. I18N.dll, I19N.West.dll). In this case, try copying those dlls from within the Unity.app to your project's Assets/Plugins folder. The location of the DLLs within the unity app is:

```
Unity.app/Contents/Frameworks/Mono/lib/mono/unity
```

You should then also check the stripping level of your project to ensure the classes in the DLLs aren't being removed when the build is optimised. Refer to the iOS Optimisation Page for more information on iOS Stripping Levels.

### Xcode Debugger console reports: ExecutionEngineException: Attempting to JIT compile method '(wrapper native-to-managed) Test:TestFunc (int)' while running with --aot-only

Typically such message is received when managed function delegate is passed to the native function, but required wrapper code wasn't generated when building application. You can help AOT compiler by hinting which methods will be passed as delegates to the native code. This can be done by adding "MonoPInvokeCallbackAttribute" custom attribute. Currently only static methods can be passed as delegates to the native code.

Sample code:

```
using UnityEngine;
using System.Collections;
using System;
using System.Runtime.InteropServices;
using AOT;

public class NewBehaviourScript : MonoBehaviour {

	[DllImport ("__Internal")]
	private static extern void DoSomething (NoParamDelegate del1, StringParamDelegate del2);

	delegate void NoParamDelegate ();
	delegate void StringParamDelegate (string str);

	[MonoPInvokeCallback (typeof (NoParamDelegate))]
	public static void NoParamCallback()
	{
		Debug.Log ("Hello from NoParamCallback");
	}

	[MonoPInvokeCallback (typeof (StringParamDelegate))]
	public static void StringParamCallback(string str)
	{
		Debug.Log (string.Format ("Hello from StringParamCallback {0}", str));
	}

	// Use this for initialization
	void Start () {
		DoSomething(NoParamCallback, StringParamCallback);
	}
}
```

## Android

# Troubleshooting Android development

### Unity fails to install your application to your device

1. Verify that your computer can actually see and communicate with the device. See the Publishing Builds page for further details.
2. Check the error message in the Unity console. This will often help diagnose the problem.

If you get an error saying "Unable to install APK, protocol failure" during a build then this indicates that the device is connected to a low-power USB port (perhaps a port on a keyboard or other peripheral). If this happens, try connecting the device to a USB port on the computer itself.

### Your application crashes immediately after launch.

1. Ensure that you are not trying to use NativeActivity with devices that do not support it.
2. Try removing any native plugins you have.
3. Try disabling stripping.
4. Use **adb logcat** to get the crash report from your device.

### Building DEX Failed

This an error which will produce a message like the following:-

```
Building DEX Failed!
G:\Unity\JavaPluginSample\Temp/StagingArea> java -Xmx1024M
-Djava.ext.dirs="G:/AndroidSDK/android-sdk_r09-windows\platform-tools/lib/"
-jar "G:/AndroidSDK/android-sdk_r09-windows\platform-tools/lib/dx.jar"
--dex --verbose --output=bin/classes.dex bin/classes.jar plugins
Error occurred during initialization of VM
Could not reserve enough space for object heap
Could not create the Java virtual machine.
```

This is usually caused by having the wrong version of Java installed on your machine. Updating your Java installation to the latest version will generally solve this issue.

### The game crashes after a couple of seconds when playing video

Make sure `Settings->Developer Options->Don't keep activities` isn't enabled on the phone. The video player is its own activity and therefore the regular game activity will be destroyed if the video player is activated.

### My game quits when I press the sleep button

Change the `<activity>` tag in the `AndroidManifest.xml` to contain `<android:configChanges>` tag as described here.

An example activity tag might look something like this:-

```
<activity android:name=".AdMobTestActivity"
          android:label="@string/app_name"
          android:configChanges="fontScale|keyboard|keyboardHidden|locale|mnc|mcc|navigation|orientation|screenLayou
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
```

Page last updated: 2012-11-26

---

# Shadows

Unity Pro makes it possible to use real-time **shadows** on any light. Objects can cast shadows onto each other and onto parts of themselves ("self shadowing"). All types of Lights - Directional, Spot and Point - support shadows.

Using shadows can be as simple as choosing **Hard Shadows** or **Soft Shadows** on a Light. However, if you want optimal shadow quality and performance, there are some additional things to consider.

The Shadow Troubleshooting page contains solutions to common shadowing problems.

Curiously enough, the best shadows are non-realtime ones! Whenever your game level geometry and lighting is static, just precompute lightmaps in your 3D application. Computing shadows offline will always result in better quality and performance than displaying them in real time. *Now onto the realtime ones...*

### Tweaking shadow quality

Unity uses so called shadow maps to display shadows. Shadow mapping is a texture based approach, it's easiest to think of it as "shadow textures" projecting out from lights onto the scene. Thus much like regular texturing, quality of shadow mapping mostly depends on two factors:

- The **resolution** (size) of the shadow maps. The larger the shadow maps, the better the shadow quality.
- The **filtering** of the shadows. **Hard shadows** take the nearest shadow map pixel. **Soft shadows** average several shadow map pixels, resulting in smoother looking shadows (but soft shadows are more expensive to render).

Different **Light** types use different algorithms to calculate shadows.

- For Directional lights, the crucial settings for shadow quality are **Shadow Distance** and **Shadow Cascades**, found in Quality Settings. **Shadow Resolution** is also taken into account, but the first thing to try to improve directional shadow quality is reducing shadow distance. All the details about directional light shadows can be found here: Directional Shadow Details.
- For Spot and Point lights, **Shadow Resolution** determines shadow map size. Additionally, for lights that cover small area on the screen, smaller shadow map resolutions are used.

Details on how shadow map sizes are computed are in Shadow Size Details page.

### Shadow performance

Realtime shadows are quite performance hungry, so use them sparingly. For each light to render its shadows, first any potential shadow casters must be rendered into the shadow map, then all shadow receivers are rendered with the shadow map. This makes shadow casting lights even more expensive than **Pixel lights**, but hey, computers are getting faster as well!

**Soft shadows** are more expensive to render than **Hard shadows**. The cost is entirely on the graphics card though (it's only longer shaders), so Hard vs. Soft shadows don't make any impact on the CPU or memory.

Quality Settings contains a setting called **Shadow Distance** - this is how far from the camera shadows are drawn. Often it makes no sense to calculate and display shadows that are 500 meters away from the camera, so use as low shadow distance as possible for your game. This will help performance (and will improve quality of directional light shadows, see above).

### Hardware support for shadows

Built-in shadows require a fragment program (pixel shader 2.0) capable graphics card. This is the list of supported cards:

- On Windows:
  - ATI Radeon 9500 and up, Radeon X series, Radeon HD series.
  - NVIDIA GeForce 6xxx, 7xxx, 8xxx, 9xxx, GeForce GT, GTX series.
  - Intel GMA X3000 (965) and up.
- On Mac OS X:
  - Mac OS X 10.4.11 or later.
  - ATI Radeon 9500 and up, Radeon X, Radeon HD series.
  - NVIDIA GeForce FX, 6xxx, 7xxx, 8xxx, 9xxx, GT, GTX series.
  - Intel GMA 950 and later.
    - Soft shadows are disabled because of driver bugs (hard shadows will be used instead).
- Mobile (iOS & Android):
  - OpenGL ES 2.0
  - GL_OES_depth_texture support. Most notably, Tegra-based Android devices do not have it, so shadows are not supported there.

### Notes

- Forward rendering path supports only one directional shadow casting light. Vertex Lit rendering path does not support realtime shadows.
- Vertex-lit lights don't have shadows.
- Vertex-lit materials won't receive shadows (but do cast shadows).
- Transparent objects don't cast or receive shadows. Transparent Cutout objects do cast and receive shadows.

Page last updated: 2012-11-16

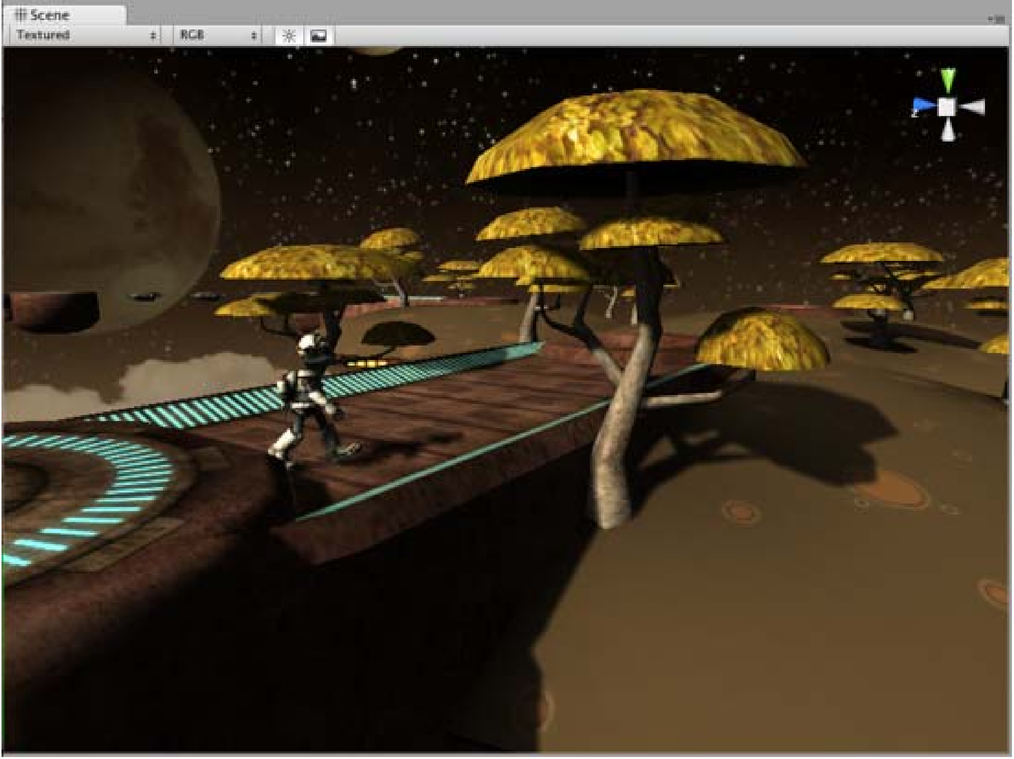
# DirectionalShadowDetails

This page explains shadows from **Directional** lights in detail.

**Note on Mobile platforms**: realtime shadows for directional lights **always** use 1 shadow cascade, and are always "hard shadows".

Directional lights are mostly used as a key light - sunlight or moonlight - in an outdoor game. Viewing distances can be huge, especially in first and third person games, and shadows often require some tuning to get the best quality vs. performance balance for your situation.

Let's start out with a good looking shadow setup for a 3rd person game:




*Shadows here look pretty good!*

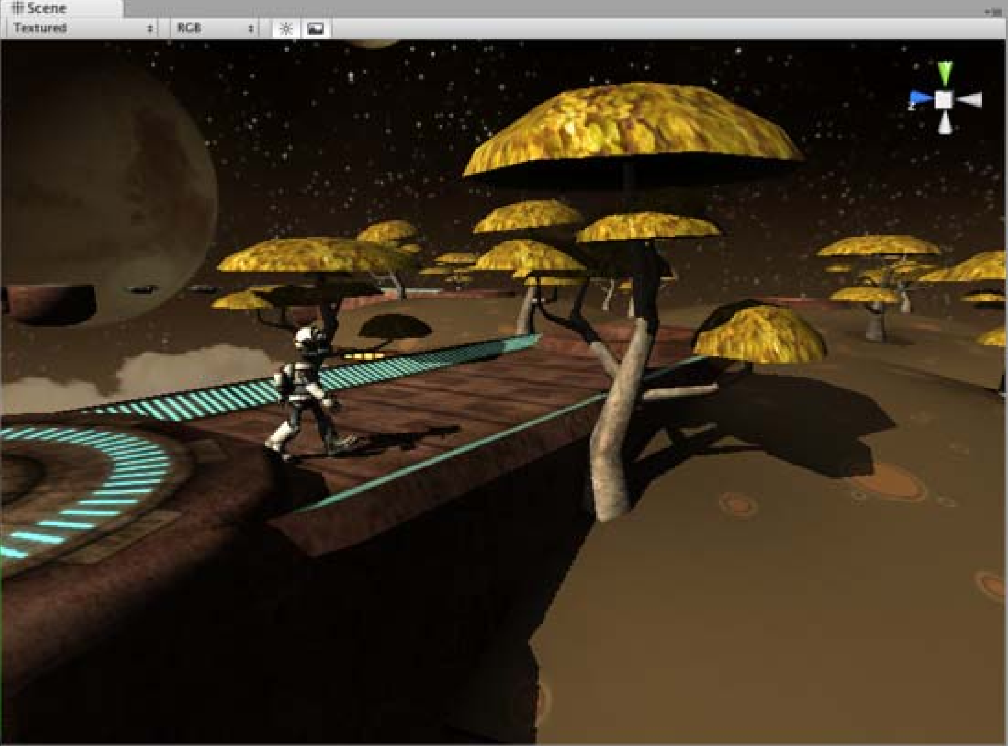
Here, visible distance is about 50 game units, so **Shadow Distance** was set to 50 in Quality Settings. Also, **Shadow Cascades** was set to 4, **Shadow Resolution** to High, and the light uses **Soft Shadows**.

Chapters below dissect each aspect of directional light shadows:

- Hard versus Soft shadows
- Shadow Cascade count
- Shadow Distance is Important!

### Hard versus Soft shadows

Using the same light setup, if we switch **Shadow Type** to **Hard Shadows**, then the transition from lit to shadowed regions is "hard" - either something is 100% in shadow, or 100% lit. Hard shadows are faster to render but often they look less realistic.



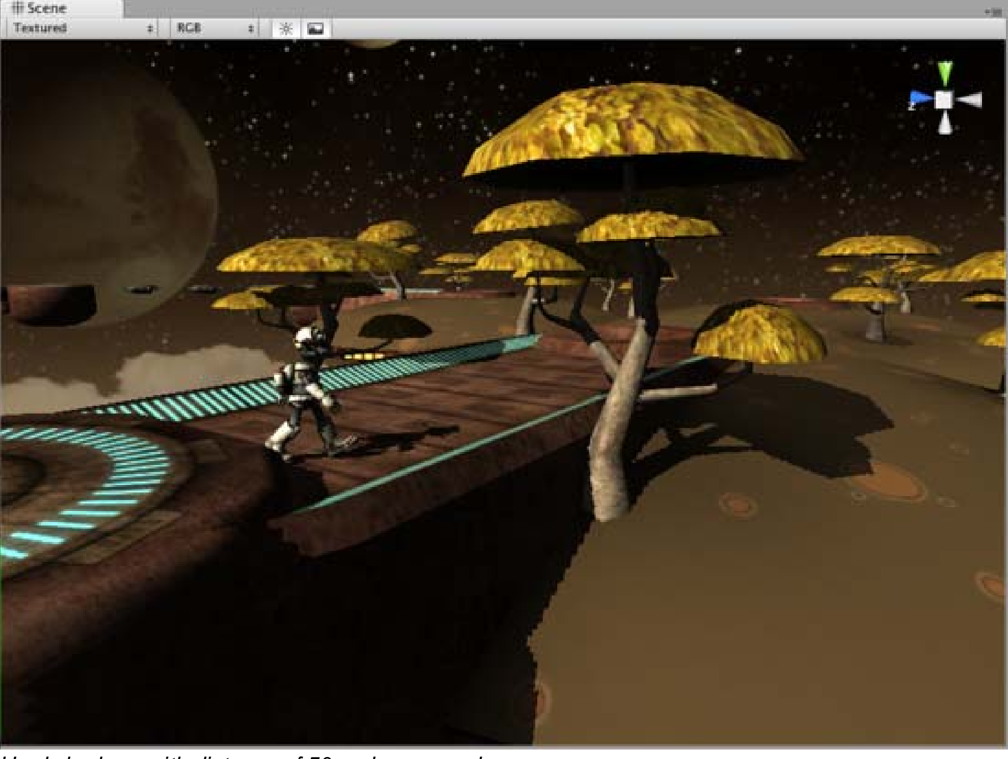

*Hard shadows with distance of 50 and four cascades.*

### Shadow Cascade count

For Directional lights Unity can use so called **Cascaded Shadow Maps** (alternatively called "Parallel Split Shadow Maps") which give very good shadow quality, especially for long viewing distances. Cascaded shadows work by dividing viewing area into progressively larger portions and using the same size shadow map on each. The result is that objects close to the viewer get more shadow map pixels than objects far away.

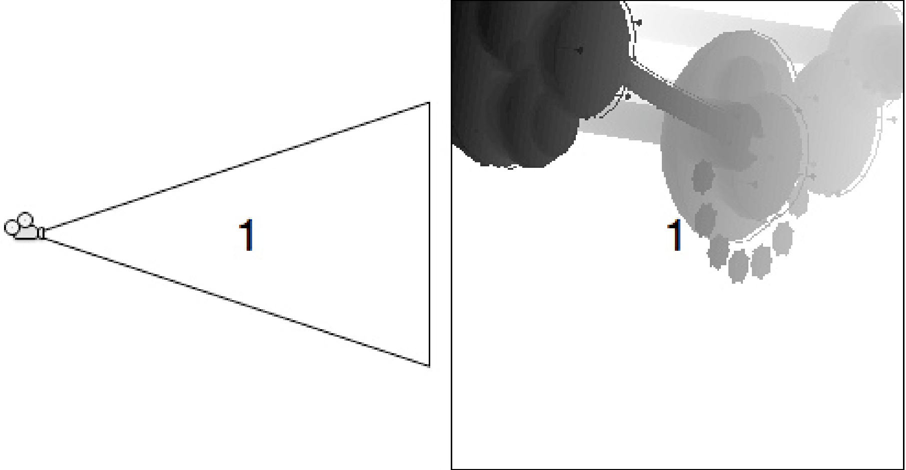
In the images below we'll use Hard shadows because shadow pixels are better visible there.

If no cascaded shadow maps were used, the entire shadow distance (still 50 units in our case) must be covered by the shadow texture uniformly. Hard shadows would look like this with no cascades:



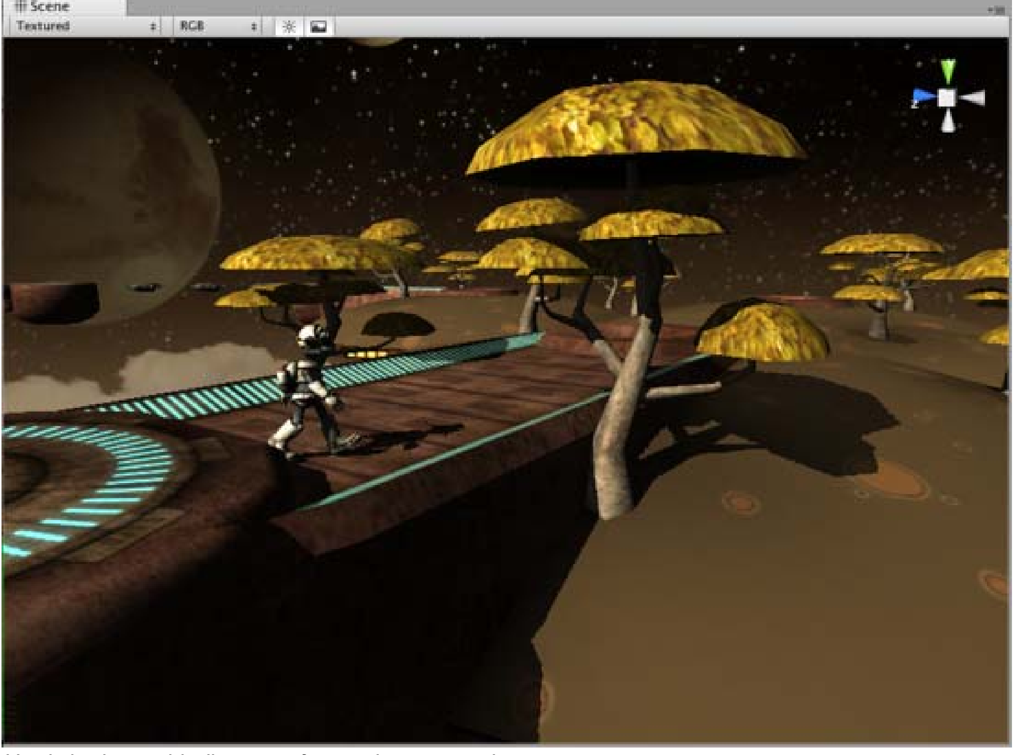

*Hard shadows with distance of 50 and no cascades.*

The pixels of the shadow texture are the same size everywhere, and while they look good in distance, the quality is not stellar up close. The shadow texture covers the entire viewing area, and if visualized it would look like this:



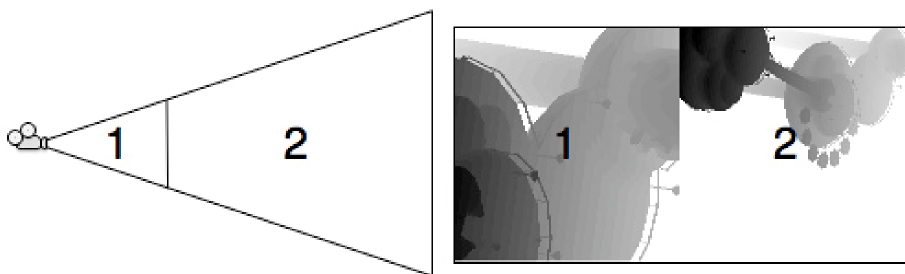

*With no cascades, shadow texture covers viewing area uniformly.*

When two shadow cascades are used, the entire shadow distance is divided into a smaller chunk near the viewer and a larger chunk far away. Hard shadows would look like this with two cascades:



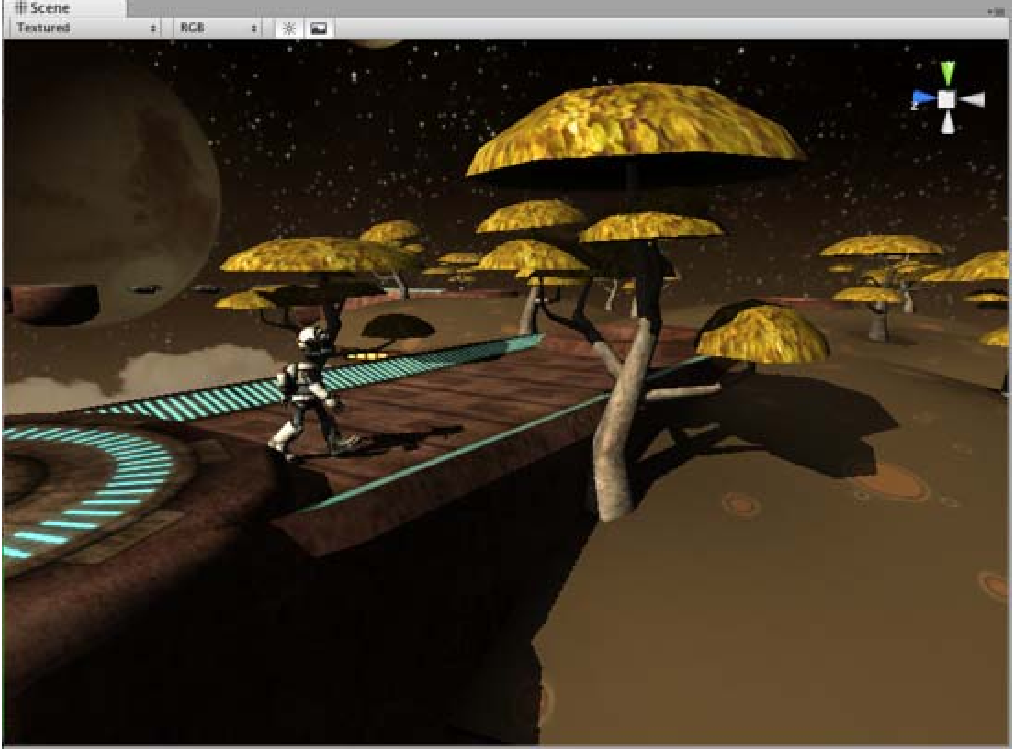

*Hard shadows with distance of 50 and two cascades.*

In exchange for some performance, we get better shadow resolution up close.

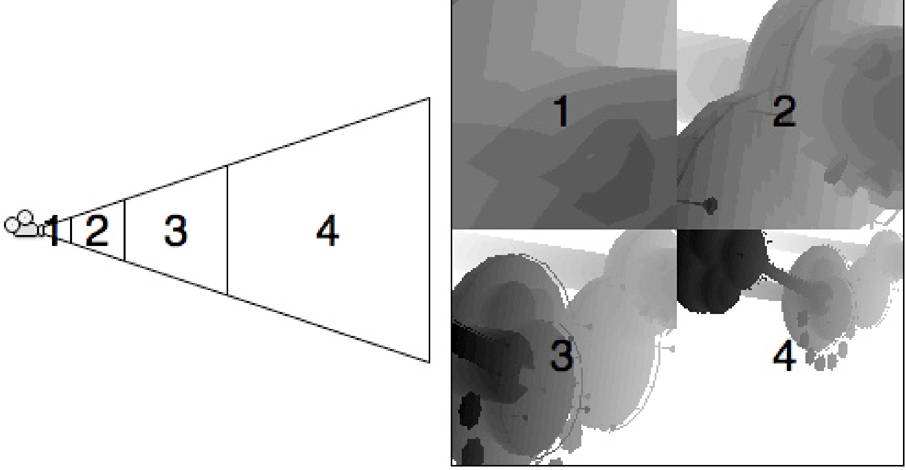



*With two cascades, two shadow textures cover different sized portions of viewing area.*

And finally when four shadow cascades are used, the shadow distance is divided into four progressively larger portions. Hard shadows would look like this with four cascades:




*Hard shadows with distance of 50 and four cascades. Hey, we've seen this already!*



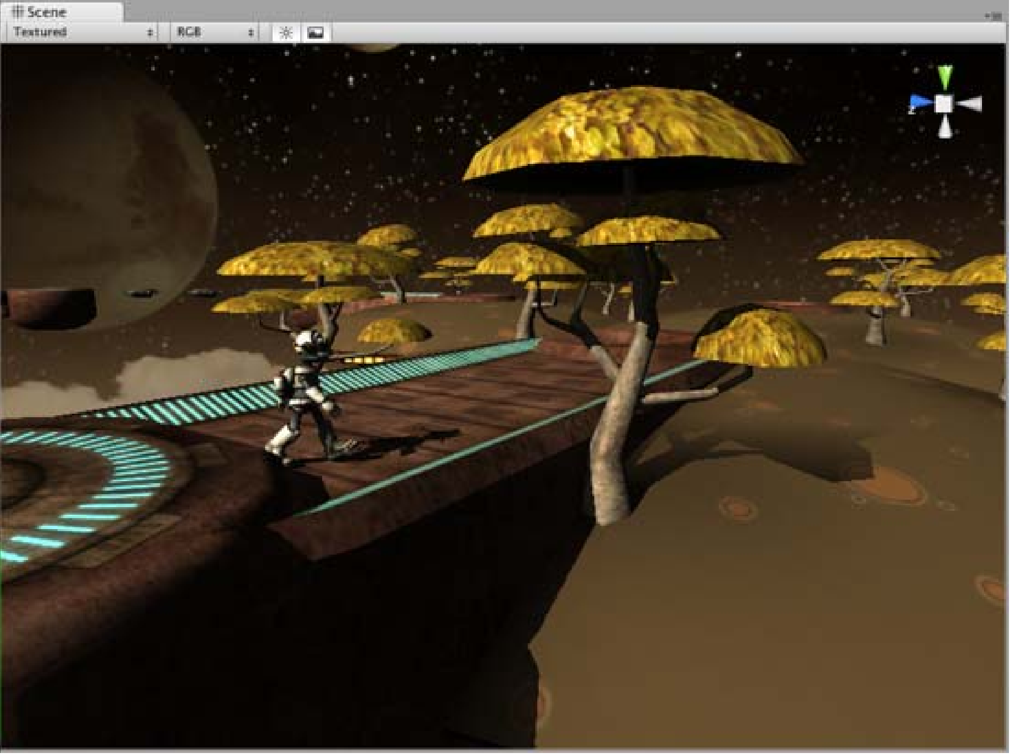

*With four cascades, four shadow textures cover different sized portions of viewing area.*

### Shadow Distance is Important!

**Shadow Distance** is extremely important for both quality and performance of directional light shadows. Just like shadow cascade count, shadow distance can be set in Quality Settings and allows an easy way to scale shadows down on less performant hardware.

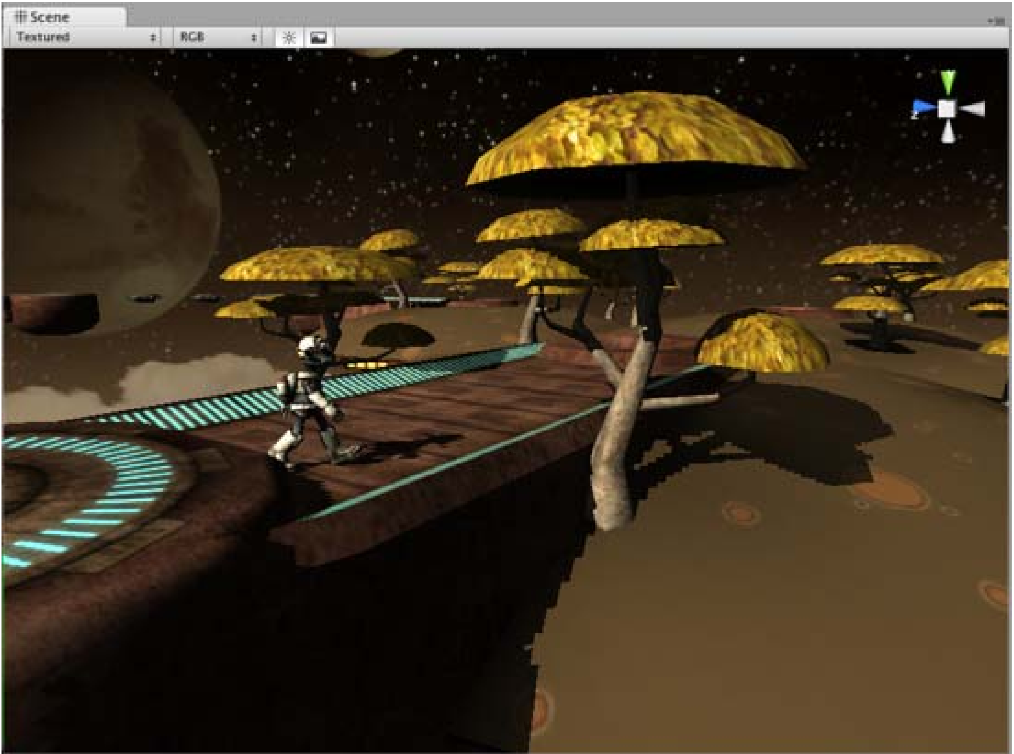
Shadows fade out at the end of shadow distance, and further than that objects are not shadowed. In most situations shadows further than some distance in the game would not be noticeable anyway!

With no shadow cascades, hard shadows and shadow distance set to 20 units our shadows look like picture below. Note that shadows do fade out in the distance, but at the same time shadow quality is much better than it was with no cascades and a distance of 50 units.



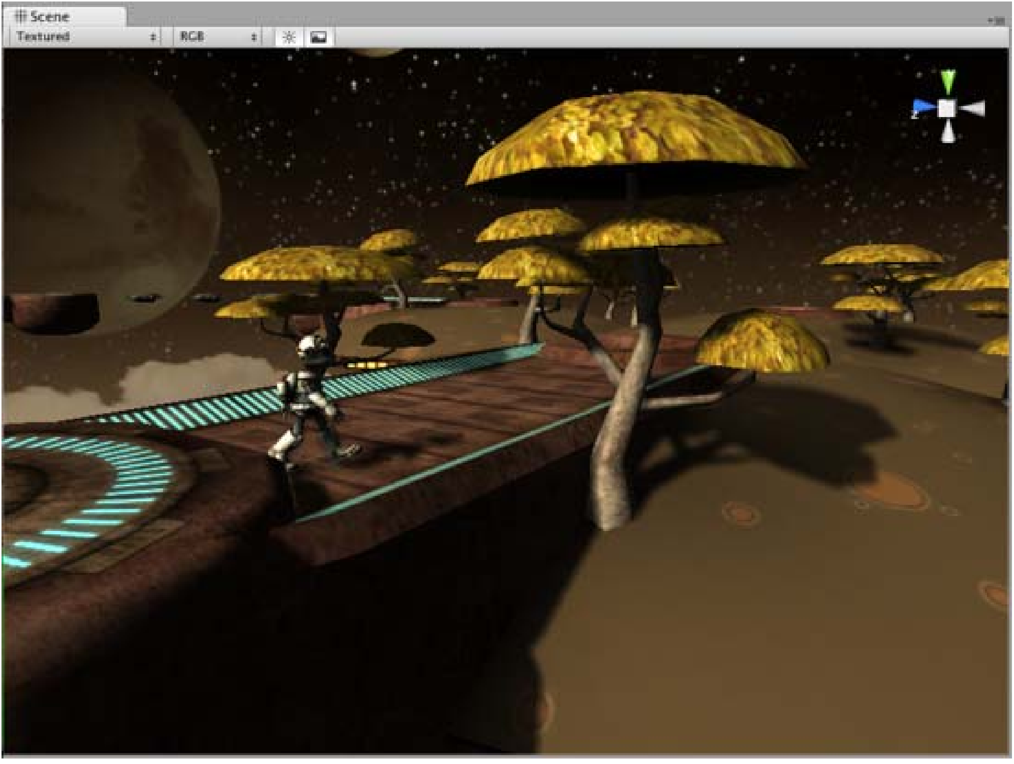

*Hard shadows with distance of 20 and no cascades.*

If on the other hand we set shadow distance too high, shadows won't look good at all. Setting distance to 100 here only decreases both performance and quality and does not make much sense - no objects in the scene are further than about 50 meters anyway!




*Hard shadows with distance of 100 and no cascades. Ouch!*

Shadow maps with cascades scale with distance much better. For example, four cascade soft shadows with covering 300 units in front of the camera look like picture below. It's somewhat worse than the picture at the top of this page, but not very bad either for a 6x increase in shadowing distance (of course in this scene that high shadow distance does not make much sense).



*Soft shadows with distance of 300 and four cascades.*

Page last updated: 2012-11-16

---

## Shadow Troubleshooting

This page lists solutions to common shadow problems.

**I see no shadows at all!**

- Shadows are a **Unity Pro** only feature, so without Unity Pro you won't get shadows. Simpler shadow methods, like using a Projector, are still possible of course.
- Shadows also require certain graphics hardware support. See Shadows page for details.
- Check if shadows are not completely disabled in Quality Settings.

**Some of my objects do not cast or receive shadows**

First, the Renderer has to have **Receive Shadows** on to have shadows on itself; and **Cast Shadows** on to cast shadows on other objects (both are on by default).

Next, only opaque objects cast and receive shadows; that means if you use built-in Transparent or Particle shaders then you'll get no shadows. In most cases it's possible to Transparent Cutout shaders (for objects like fences, vegetation etc.). If you use custom written Shaders, they have to be pixel-lit and use Geometry render queue. Objects using **VertexLit** shaders do not receive shadows either (but can cast shadows just fine).

Finally, in Forward rendering path, only the brightest directional light can cast shadows. If you want to have many shadow casting lights, you need to use Deferred Lighting rendering path.

Page last updated: 2012-08-17

---

## Shadow Size Details

Unity computes shadow map sizes this way:

First light's "coverage box" on the screen is computed. This is what rectangle on the screen the light possibly illuminates:

- For Directional lights that is the whole screen.
- For Spot lights it's the bounding rectangle of light's pyramid projected on the screen.
- For Point lights it's the bounding rectangle of light's sphere projected on the screen.

Then the larger value of this box' width & height is chosen; call that `pixel size`.

At "High" shadow resolution, the size of the shadow map then is:

- Directional lights: `NextPowerOfTwo( pixel size * 1.9 )`, but no more than 2048.
- Spot lights: `NextPowerOfTwo( pixel size )`, but no more than 1024.
- Point lights: `NextPowerOfTwo( pixel size * 0.5 )`, but no more than 512.

When graphics card has 512MB or more video memory, the upper shadow map limits are increased (4096 for Directional, 2048 for Spot, 1024 for Point lights).

At "Medium" shadow resolution, shadow map size is 2X smaller than at "High" resolution. And at "Low" resolution, it's 4X smaller than at "High" resolution.

The seemingly low limit on Point lights is because they use cubemaps for shadows. That means six cubemap faces at this resolution must be in video memory. They are also quite expensive to render, as potential shadow casters must be rendered into up to six cubemap faces.

### Shadow size computation when running close to memory limits

When running close to video memory limits, Unity will automatically drop shadow map resolution computed above.

Generally memory for the screen (backbuffer, frontbuffer, depth buffer) has to be in video memory; and memory for render textures has to be in video memory, Unity will use both to determine allowed memory usage of shadow maps. When allocating a shadow map according to size computed above, it's size will be reduced until it fits into `(TotalVideoMemory - ScreenMemory - RenderTextureMemory) / 3`.

Assuming all regular textures, vertex data and other graphics objects can be swapped out of video memory, maximum VRAM that could be used by a shadow map would be `(TotalVideoMemory-ScreenMemory-RenderTextureMemory)`. But exact amounts of memory taken by screen and render textures can never be determined, and some objects can not be swapped out, and performance would be horrible if all textures would be constantly swapping in and out. So Unity does not allow a shadow map to exceed one third of "generally available" video memory, which works quite well in practice.

Page last updated: 2012-08-17

---

# IME Input

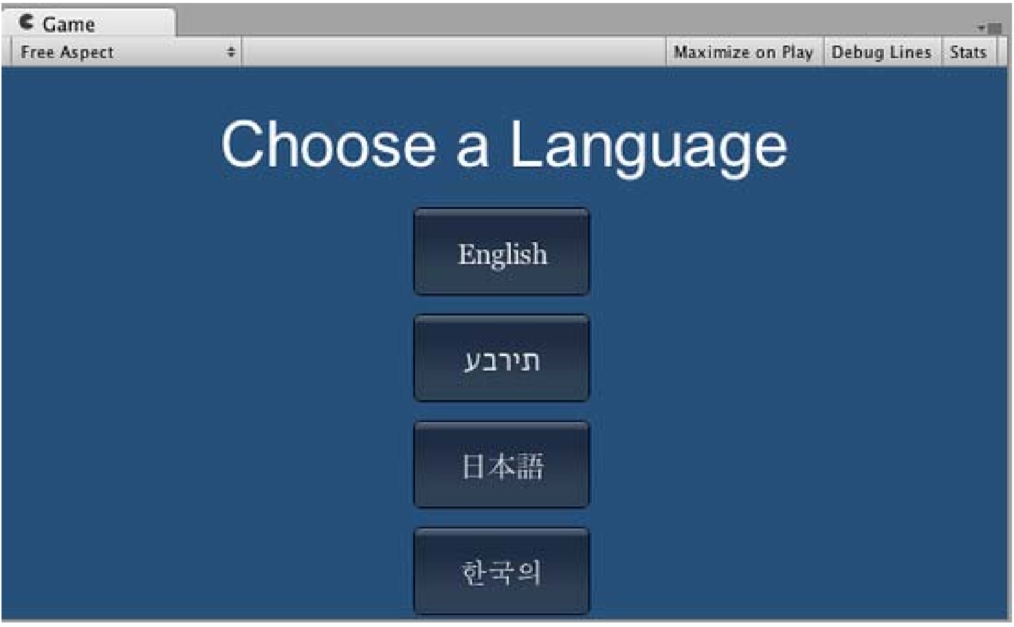
### What is Input Method Editor (IME)?

An input method is an operating system component or program that allows users to enter characters and symbols not found on their input device. For instance, on the computer, this allows the user of 'Western' keyboards to input Chinese, Japanese, Korean and Indic characters. On many hand-held devices, such as mobile phones, it enables using the numeric keypad to enter Latin alphabet characters.

The term input method generally refers to a particular way to use the keyboard to input a particular language, for example the Cangjie method, the pinyin method, or the use of dead keys.

# IME and Unity

# Desktop




Unity provides IME support, which means that you can write non-ASCII characters in all your graphical user interfaces. This Input Method is fully integrated with the engine so you do not need to do anything to activate it. In order to test it, just change your keyboard language to a non-ASCII language (e.g. Japanese) and just start writing your interface.

For more info and optimization when writting non-ASCII characters, check the ***character*** option in the font properties.

**Note:** IME on Unity is not supported on mac webplayers at the moment.

## iOS

This feature is not supported on iOS devices yet.

## Android

This feature is not supported on Android devices yet.

Page last updated: 2012-03-14

---

# OptimizeForIntegratedCards

### Polygon count matters

On most graphics cards today, polygon count does not really matter. The common knowledge is that object count and fillrate is much more important. Unfortunately, not so on the majority of older integrated chips (Intel 945 / GMA 950 and similar). How much it matters depends on the complexity of the vertex shaders or lighting and the speed of the CPU (thats right, most integrated cards transform & light vertices on the CPU).

Big Bang Brain Games never went above 25 thousand triangles in a scene using 1-2 per-vertex lights and no pixel lights (essentially a VertexLit rendering path). Quality Settings were used to speed up performance automatically when frame rate drops. So on higher end machines a higher quality setting was used which had pixel lights enabled.

What slows things down is drawing objects multiple times, using complex vertex shaders and lots of polygons. This means:

- Use VertexLit rendering path if possible. This will draw each object just once, no matter how many lights are in the scene.
- Try not to use lights at all, even vertex lights. Lights make sense if your geometry moves, or if your lights move. Otherwise bake the illumination using Lightmapper, it will run faster and look better.
- Optimize your geometry (see section below).

- Use Rendering Statistics window and Profiler!

### Optimize model geometry

When optimizing the geometry of a model, there are two basic rules:

- Don't use excessive amount of faces if you don't have to.
- Keep the number of UV mapping seams and hard edges as low as possible.

Note that the actual number of vertices that graphics hardware has to process is usually not the same as displayed in a 3D application. Modeling applications usually display the geometric vertex count, i.e. number of points that make up a model.

For a graphics card however, some vertices have to be split into separate ones. If a vertex has multiple normals (it's on a "hard edge"), or has multiple UV coordinates, or multiple vertex colors, it has to be split. So the vertex count you see in Unity is almost always different from the one displayed in 3D application.

### Bake lighting.

Bake ligthing either into lightmaps or vertex colors. Unity has a great Lightmapper built-in; also you can bake lightmaps in many 3D modeling packages.

The process of generating a lightmapped environment takes only a little longer than just placing a light in the scene in Unity, **but**:

- It usually will run a lot faster; especially if you have many lights.
- And look a lot better since you can bake global illumination.

Even next-gen games still rely on lightmapping a lot. Usually they use lightmapped environments and only use one or two realtime dynamic lights.

Page last updated: 2012-10-12

---

## Web Player Deployment

When building a **Web Player**, Unity automatically generates an HTML file next to the player data file. It contains the default HTML code to load the web player data file.

It is possible to further tweak and customize the generated HTML file to make it fit better with the containing site's design, to add more HTML content, etc. The following pages discuss the related subjects in depth:

- HTML code to load Unity content
- Working with UnityObject2
- Customizing the Unity Web Player loading screen
- Customizing the Unity Web Player's Behavior
- Unity Web Player and browser communication
- Using web player templates
- Web Player Streaming
- Webplayer Release Channels

Page last updated: 2012-10-12

---

## HTML code to load Unity Web Player content

Unity content is loaded in the browser by the Unity **Web Player** plugin. HTML code usually does not communicate with this plugin directly but via the script called **UnityObject2**. Its primary task is to make Unity content embedding very simple by shielding the user from various browser- and platform-specific issues. It also enables easy Web Player installation.

The HTML file generated by Unity when building a web player contains all the commonly required functionality. In most cases you don't have to modify the HTML file at all. The rest of the document explains the inner workings of this file.

The UnityObject2 script has to be loaded before it can be used. This is done at the top of the **<head>** section.

```
<script type="text/javascript">
<!--
var unityObjectUrl = "http://webplayer.unity3d.com/download_webplayer-3.x/3.0/uo/UnityObject2.js";
if (document.location.protocol == 'https:')
	unityObjectUrl = unityObjectUrl.replace("http://", "https://ssl-");
document.write('<script type="text/javascript" src="' + unityObjectUrl + '"></script>');
-->
</script>
```

You can now instantiate the **UnityObject2** class to assist you in various Unity-related tasks, the most important one being embedding Unity content. This is performed by instantiating **UnityObject2** and calling **initPlugin** on the new instance. **initPlugin** accepts several parameters. The first one specifies the **id** of the HTML element that will be replaced by Unity content. It could be any HTML element with **<div>** being the most common. Think of it as a temporary placeholder where Unity should be placed. The second parameter specifies the path to the web player data file to be displayed. See UnityObject2.initPlugin for more info.

```
var u = new UnityObject2();
u.initPlugin(jQuery("#unityPlayer")[0], "Example.unity3d");
```

Finally, the HTML placeholder is placed inside the **<body>** section. It could be as simple as **<div id="unityPlayer" />**. However for maximum compatibility it's best to place some warning message in case the browser doesn't support JavaScript and the placeholder isn't replaced by UnityObject.

```
<div id="unityPlayer">
	<div class="missing">
		<a href="http://unity3d.com/webplayer/" title="Unity Web Player. Install now!">
			<img alt="Unity Web Player. Install now!" src="http://webplayer.unity3d.com/installation/getunity.png" wid
		</a>
	</div>
</div>
```

Page last updated: 2012-11-16

---

# Working with UnityObject

**UnityObject2** is a JavaScript script that simplifies Unity content embedding into HTML and allows you to customize the install process. Having a custom install UI that matches your game and website, will create a more engaging and pleasurable experience for the end-user. It has functions to detect the Unity **Web Player** plugin, initiate Web Player installation and embed Unity content. Although it's possible to deploy **UnityObject2.js** file on the web server alongside the HTML file it's best to load it directly from the Unity server at **http://webplayer.unity3d.com/download_webplayer-3.x/3.0/uo/UnityObject2.js**. That way you will always reference the most up to date version of UnityObject2. Please note that the **UnityObject2.js** file hosted on the Unity server is minified to make it smaller and save traffic. If you want to explore the source code you can find the original file in the **Data\Resources** folder on Windows and the **Contents/Resources** folder on Mac OS X. UnityObject2 by default sends anonymous data to GoogleAnalytics which is used to help us identify installation type and conversion rate. UnityObject2 depends on jQuery.

## Constructor

You will need to create a new instance of the unityObject2 for each Unity content present on the page.

*Parameters:*

- **configuration** - A object containing the configuration for this instance. Those are the available members:
  - **width** - Default: 100%, Width of Unity content. Can be specified in pixel values (i.e. 600, "450") or in percentage values

(i.e. "50%", "100%"). Note that percentage values are relative to the parent element.

- **height** - Default: 100%, Height of Unity content. Can be specified in pixel values (i.e. 600, "450") or in percentage values (i.e. "50%", "100%"). Note that percentage values are relative to the parent element.
- **fullInstall** - Default: false, Installs the full Web Player if not available. Normally only a small part of the Web Player is installed and the remaining files are automatically downloaded later.
- **enableJava** - Default: true, Enables Java based installation. Some platforms doesn't support this feature.
- **enableClickOnce** - Default: true, Enables ClickOnce based installation. Only works on Internet Explorer browsers.
- **enableUnityAnalytics** - Default: true, Notifies Unity about Web Player installation. This doesn't do anything if the Web Player is already installed.
- **enableGoogleAnalytics** -Default: true, Notifies Unity about Web Player installation using Google Analytics. This doesn't do anything if the Web Player is already installed.
- **params** - Default: {}, Extra parameters to be used when embedding the Player. Those are usefull to customize the Unity experience:
  - **backgroundcolor** - Default: "FFFFFF", The background color of the web player content display region during loading, the default is white. **Pro Only**
  - **bordercolor** - Default: "FFFFFF", The color of the one pixel border drawn around the web player content display region during loading. **Pro Only**
  - **textcolor** - Default: "000000", The color of error message text (when data file fails to load for example). **Pro Only**
  - **logoimage** - Default: unity Logo, The path to a custom logo image, the logo image is drawn centered within the web player content display region during loading. **Pro Only**
  - **progressbarimage** - The path to a custom image used as the progress bar during loading. The progress bar image width is clipped based on the amount of file loading completed, therefore it starts with a zero pixel width and animates to its original width when the loading is complete. The progress bar is drawn beneath the logo image. **Pro Only**
  - **progressframeimage** - The path to a custom image used to frame the progress bar during loading. **Pro Only**
  - **disableContextMenu** - This parameter controls whether or not the Unity Web Player displays a context menu when the user right- or control-clicks on the content. Setting it to true prevents the context menu from appearing and allows content to utilize right-mouse behavior. To enable the context menu don't include this parameter.
  - **disableExternalCall** - This parameter controls whether or not the Unity Web Player allows content to communicate with browser-based JavaScript. Setting it to true prevents browser communication and so content cannot call or execute JavaScript in the browser, the default is false.
  - **disableFullscreen** - This parameter controls whether or not the Unity Web Player allows content to be viewed in fullscreen mode. Setting it to true prevents fullscreen viewing and removes the "Go Fullscreen" entry from the context menu, the default is false.
- **attributes** - Default: {}, Object containing list of attributes. These will be added to underlying **<object>** or **<embed>** tag depending on the browser.
- **debugLevel** - Default: 0, Enables/Disables logging, useful when developing to be able to see the progress on the browser console. Set it greater to 0 to enable.

*Notes*: All color values provided must be 6-digit hexidecimal colors, (eg. FFFFFF, 020F16, etc.). The image paths provided can be either relative or absolute links and all image files must be RGB (without transparency) or RGBA (with transparency) 8-bit/channel PNG files. Finally, the progressframeimage and the progressbarimage should be the same height.

## Functions

**observeProgress**
You can register a callback that will receive notifications during the plugin installation and/or initialization.

*Parameters:*

- **callback** - Callback function that will receive information about the plugin installation/initialization. This callback will receive an **progress** object.
  - **progress** - It contains information about the current step of the plugin installation/initialization.
    - **pluginStatus** - Will contain a string identifying the plugin status, can be one of those:
      - **unsupported**` - The current Browser/OS is not supported
      - **missing** - Supported platform, but the plugin haven't be installed yet.
      - **installed** - The plugin have finished installing, or was already installed.
      - **first** - called after the plugin have been installed at the first frame of the game is played (This will not be called if the plugin was already installed previously)
    - **targetEl** - The DOM Element serving as a container for the webplayer (This is the same element you pass to UnityObject2.initPlugin)
    - **bestMethod** - If the plugin is missing will contain the best installation path for the current platform, can be one of

the following strings:
  - **JavaInstall** - It will use our "One Click" Java Applet to install the plugin
  - **ClickOnceIE** - It will use Internet Explorer's Click Once install
  - **Manual** - It will ask the user to download a file and install it manually.
- **unityObj**- A reference to the previously created unityObject2 instance responsible for this callback
- **ua** - Contains some User Agent information used to decide the pluginStatus and bestMethod.

**initPlugin**
This will actually try to start up your game. And call the callback you previously registered at the appropriated moments. Note that

*Parameters:*

- **containerElement** - The DOM Element serving as a container for the webplayer
- **gameURL** - URL to the web player data file (.unity3d). Can be relative or absolute.

*Notes*: This function should be called after the containerElement is present on the DOM, to be safe, you could wait for the DOM to load completely before calling it.

**installPlugin**
Tries to install the plugin using the specified **method**. If no method is passed, it will try to use this.getBestInstallMethod().

*Parameters:*

- **method** - Default: this.**getBestInstallMethod**(). A string specifying which method to use for installation. It can be one of the following values: JavaInstall, ClickOnceIE, Manual.

*Notes*: Not all methods are available in every platform/browser. Manual will download an exe/dmg installer and the user will need to perform a manual installation.

**getUnity**
This will return a reference to the player, so you can Call SendMessage from it for instance.

*Notes*: Will return null if the WebPlayer has not been initialized yet.

*Example:* This exemplifies a very simple UnityObject2 usage. If the user already has the plugin installed, the WebPlayer will load normally, otherwise it will reveal a hidden `div.missing` and attach a click event to the install button. After installation is succeeded, the screen is hidden, and the webplayer is loaded normally.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Unity Web Player | "Sample"</title>
        <script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
        <script type="text/javascript">
                <!--
                var unityObjectUrl = "http://webplayer.unity3d.com/download_webplayer-3.x/3.0/uo/UnityObject2.js";
                if (document.location.protocol == 'https:')
                        unityObjectUrl = unityObjectUrl.replace("http://", "https://ssl-");
                document.write('<script type="text\/javascript" src="' + unityObjectUrl + '"><\/script>');
                -->
        </script>
        <script type="text/javascript">
                var u = new UnityObject2();
                u.observeProgress(function (progress) {
                        var $missingScreen = jQuery(progress.targetEl).find(".missing");
                        switch(progress.pluginStatus) {
                                case "unsupported":
                                        showUnsupported();
                                break;
```

```
                    case "broken":
                        alert("You will need to restart your browser after installation.");
                    break;
                    case "missing":
                        $missingScreen.find("a").click(function (e) {
                            e.stopPropagation();
                            e.preventDefault();
                            u.installPlugin();
                            return false;
                        });
                        $missingScreen.show();
                    break;
                    case "installed":
                        $missingScreen.remove();
                    break;
                    case "first":
                    break;
                }
            });
            jQuery(function(){
                u.initPlugin(jQuery("#unityPlayer")[0], "Example.unity3d");
            });
        </script>
    </head>
        <body>
            <p class="header">
                <span>Unity Web Player | </span>WebPlayer
            </p>
            <div class="content">
                <div id="unityPlayer">
                    <div class="missing">
                        <a href="http://unity3d.com/webplayer/" title="Unity Web Player. Install now!">
                            <img alt="Unity Web Player. Install now!" src="http://webplayer.unity3d.com/installation/
                        </a>
                    </div>
                </div>
            </div>
            <p class="footer">&laquo; created with <a href="http://unity3d.com/unity/" title="Go to unity3d.com">Unity</a
        </body>
</html>
```

Page last updated: 2012-11-16

---

# Customizing the Unity Web Player loading screen

By default the Unity **Web Player** displays a small Unity logo and a progress bar while loading web player content. It is possible to customize the appearance of that loading screen, including both the logo and progress bar display.

Please note that modifying the loader images is only possible with **Unity Pro**.

There are six optional parameters that can be passed to **UnityObject**, which can be used to customize the appearance of the Unity Web Player loading screen. Those optional parameters are:

- **backgroundcolor**: The background color of the web player content display region during loading, the default is white.
- **bordercolor**: The color of the one pixel border drawn around the web player content display region during loading, the default is white.
- **textcolor**: The color of error message text (when data file fails to load for example). The default is black or white, depending on the background color.

- **logoimage**: The path to a custom logo image, the logo image is drawn centered within the web player content display region during loading.
- **progressbarimage**: The path to a custom image used as the progress bar during loading. The progress bar image's width is clipped based on the amount of file loading completed, therefore it starts with a zero pixel width and animates to its original width when the loading is complete. The progress bar is drawn beneath the logo image.
- **progressframeimage**: The path to a custom image used to frame the progress bar during loading.

All color values provided must be 6-digit hexadecimal colors, (eg. FFFFFF, 020F16, etc.). The image paths provided can be either relative or absolute links. All images must be PNG files in RGB format (without transparency) or RGBA format (with transparency) stored at eight bits per channel. Finally, the **progressframeimage** and the **progressbarimage** should be the same height.

Here is an example script that customizes the appearance of the Unity Web Player loading screen. The background color is set to light gray (**A0A0A0**), border color to black (**000000**), text color to white (**FFFFFF**) and loader images to **MyLogo.png**, **MyProgressBar.png** and **MyProgressFrame.png**. All parameters are grouped into single **params** object and passed to UnityObject2 Constructor.

```
var params = {
	backgroundcolor: "A0A0A0",
	bordercolor: "000000",
	textcolor: "FFFFFF",
	logoimage: "MyLogo.png",
	progressbarimage: "MyProgressBar.png",
	progressframeimage: "MyProgressFrame.png"
};
var u = UnityObject2({ params: params });
u.initPlugin(jQuery("#unityPlayer")[0], "Example.unity3d");
```

See UnityObject2 for more details.

Example using the above snippet:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Unity Web Player | "Sample"</title>
        <script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
        <script type="text/javascript">
            <!--
            var unityObjectUrl = "http://webplayer.unity3d.com/download_webplayer-3.x/3.0/uo/UnityObject2.js";
            if (document.location.protocol == 'https:')
                unityObjectUrl = unityObjectUrl.replace("http://", "https://ssl-");
            document.write('<script type="text\/javascript" src="' + unityObjectUrl + '"><\/script>');
            -->
        </script>
        <script type="text/javascript">
            var params = {
                backgroundcolor: "A0A0A0",
                bordercolor: "000000",
                textcolor: "FFFFFF",
                logoimage: "MyLogo.png",
                progressbarimage: "MyProgressBar.png",
                progressframeimage: "MyProgressFrame.png"
            };
            var u = new UnityObject2({ params: params });
            u.observeProgress(function (progress) {
                var $missingScreen = jQuery(progress.targetEl).find(".missing");
                switch(progress.pluginStatus) {
                    case "unsupported":
```

```
                        showUnsupported();
                    break;
                    case "broken":
                        alert("You will need to restart your browser after installation.");
                    break;
                    case "missing":
                        $missingScreen.find("a").click(function (e) {
                            e.stopPropagation();
                            e.preventDefault();
                            u.installPlugin();
                            return false;
                        });
                        $missingScreen.show();
                    break;
                    case "installed":
                        $missingScreen.remove();
                    break;
                    case "first":
                    break;
                }
            });
            jQuery(function(){
                u.initPlugin(jQuery("#unityPlayer")[0], "Example.unity3d");
            });
        </script>
    </head>
        <body>
            <p class="header">
                <span>Unity Web Player | </span>WebPlayer
            </p>
            <div class="content">
                <div id="unityPlayer">
                    <div class="missing">
                        <a href="http://unity3d.com/webplayer/" title="Unity Web Player. Install now!">
                            <img alt="Unity Web Player. Install now!" src="http://webplayer.unity3d.com/installation/
                        </a>
                    </div>
                </div>
            </div>
            <p class="footer">&laquo; created with <a href="http://unity3d.com/unity/" title="Go to unity3d.com">Unity</a:
        </body>
```

Page last updated: 2012-11-16

---

# WebPlayerBehaviorTags

The Unity **Web Player** allows developers to use a few optional parameters to easily control its behavior in a few ways:

- **disableContextMenu**: This parameter controls whether or not the Unity Web Player displays a context menu when the user right- or control-clicks on the content. Setting it to true prevents the context menu from appearing and allows content to utilize right-mouse behavior. To enable the context menu don't include this parameter.
- **disableExternalCall**: This parameter controls whether or not the Unity Web Player allows content to communicate with browser-based JavaScript. Setting it to true prevents browser communication and so content cannot call or execute JavaScript in the browser, the default is false.
- **disableFullscreen**: This parameter controls whether or not the Unity Web Player allows content to be viewed in fullscreen mode. Setting it to true prevents fullscreen viewing and removes the "Go Fullscreen" entry from the context menu, the default is false.

Using **UnityObject2** you control those parameters like this:

```
var params = {
      disableContextMenu: true
};
var u = UnityObject2({ params: params });
u.initPlugin(jQuery("#unityPlayer")[0], "Example.unity3d");
```

In the above example you'll notice that neither **disableExternalCall** nor **disableFullscreen** are specified, therefore their default values are used.

See UnityObject2 for more details.

Example setting all the behavior options:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
      <title>Unity Web Player | "Sample"</title>
      <script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
      <script type="text/javascript">
            <!--
            var unityObjectUrl = "http://webplayer.unity3d.com/download_webplayer-3.x/3.0/uo/UnityObject2.js";
            if (document.location.protocol == 'https:')
                  unityObjectUrl = unityObjectUrl.replace("http://", "https://ssl-");
            document.write('<script type="text\/javascript" src="' + unityObjectUrl + '"><\/script>');
            -->
      </script>
      <script type="text/javascript">
            var params = {
                  disableContextMenu: true,
                  disableExternalCall: false,
                  disableFullscreen: false,
            };
            var u = new UnityObject2({ params: params });
            u.observeProgress(function (progress) {
                  var $missingScreen = jQuery(progress.targetEl).find(".missing");
                  switch(progress.pluginStatus) {
                        case "unsupported":
                              showUnsupported();
                        break;
                        case "broken":
                              alert("You will need to restart your browser after installation.");
                        break;
                        case "missing":
                              $missingScreen.find("a").click(function (e) {
                                    e.stopPropagation();
                                    e.preventDefault();
                                    u.installPlugin();
                                    return false;
                              });
                              $missingScreen.show();
                        break;
                        case "installed":
                              $missingScreen.remove();
                        break;
                        case "first":
                        break;
                  }
```

```
        });
        jQuery(function(){
            u.initPlugin(jQuery("#unityPlayer")[0], "Example.unity3d");
        });
    </script>
  </head>
    <body>
        <p class="header">
            <span>Unity Web Player | </span>WebPlayer
        </p>
        <div class="content">
            <div id="unityPlayer">
                <div class="missing">
                    <a href="http://unity3d.com/webplayer/" title="Unity Web Player. Install now!">
                        <img alt="Unity Web Player. Install now!" src="http://webplayer.unity3d.com/installation/
                    </a>
                </div>
            </div>
        </div>
        <p class="footer">&laquo; created with <a href="http://unity3d.com/unity/" title="Go to unity3d.com">Unity</a:
    </body>
```

Page last updated: 2012-11-16

---

# Unity Web Player and browser communication

The HTML page that contains **Unity Web Player** content can communicate with that content and vice versa. Basically there are two communication directions:

- The web page calls functions inside the Unity web player content.
- The Unity web player content calls functions in the web page.

Each of these communication directions is described in more detail below.

### Calling Unity web player content functions from the web page

The Unity Web Player object has a function, **SendMessage()**, that can be called from a web page in order to call functions within Unity web player content. This function is very similar to the GameObject.SendMessage function in the Unity scripting API. When called from a web page you pass an object name, a function name and a single argument, and **SendMessage()** will call the given function in the given game object.

In order to call the Unity Web Player's **SendMessage()** function you must first get a reference to the Unity web player object. You can use the **GetUnity()** function in the default html generated by Unity to obtain a reference to the object. Here is an example JavaScript function that would execute the **SendMessage()** function on the Unity web player; in turn **SendMessage()** will then call the function **MyFunction()** on the game object named *MyObject*, passing a piece of string data as an argument:

```
<script type="text/javascript" language="javascript">
<!--
//initializing the WebPlayer
var u = new UnityObject2();
u.initPlugin(jQuery("#unityPlayer")[0], "Example.unity3d");

function SaySomethingToUnity()
{
    u.getUnity().SendMessage("MyObject", "MyFunction", "Hello from a web page!");
}
-->
```

```
</script>
```

Inside of the Unity web player content you need to have a script attached to the **GameObject** named **MyObject**, and that script needs to implement a function named **MyFunction**:

```
function MyFunction(param : String)
{
    Debug.Log(param);
}
```

**Note:** keep in mind that if the function doesn't have any arguments, then an empty string ("") should be passed as an argument.

A single string, integer or float argument must be passed when using **SendMessage()**, the parameter is required on the calling side. If you don't need it then just pass a zero or other default value and ignore it on the Unity side. Additionally, the game object specified by the name can be given in the form of a path name. For example, **/MyObject/SomeChild** where **SomeChild** must be a child of **MyObject** and **MyObject** must be at the root level due to the '/' in front of its name.

**Note:** u.getUnity() might return null if the game isn't fully loaded yet, so it's a good idea to check if it's value is not null before using SendMessage(). Or wait for your game to be fully loaded before trying to communicate with it.

### Calling web page functions from Unity web player content

In order to call a web page function from within your Unity web player content you must use the **Application.ExternalCall()** function. Using that function you can call any JavaScript function defined in the web page, passing any number of parameters to it. Here is an example Unity script that uses the **Application.ExternalCall()** function to call a function named **SayHello()** found within the web page, passing a piece of string data as an argument:

```
Application.ExternalCall( "SayHello", "The game says hello!" );
```

The web page would need to define the **SayHello()** function, for example:

```
<script type="text/javascript" language="javascript">
<!--
function SayHello( arg )
{
    // show the message
    alert( arg );
}
-->
</script>
```

### Executing arbitrary browser code from Unity web player content

You don't even have to define functions in the embedding web page, instead you can use the **Application.ExternalEval()** function to execute arbitrary browser code from the web player content.

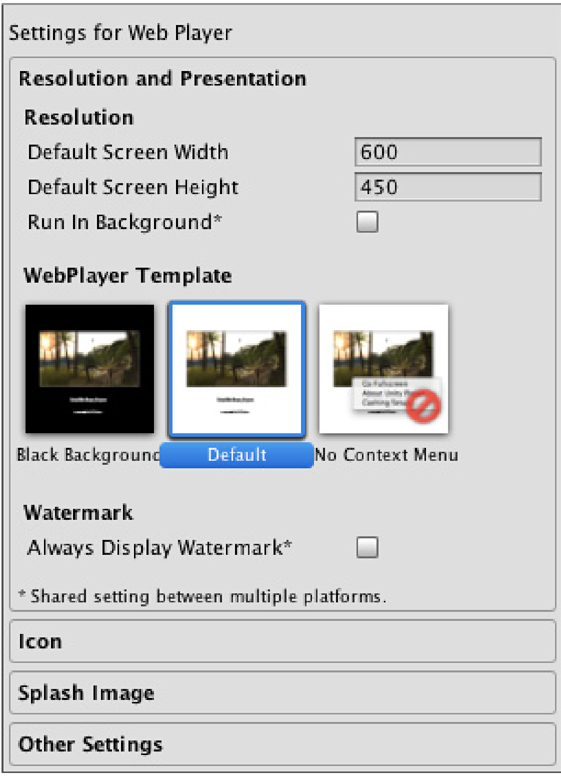
The following example checks that the page embedding the web player content is fetched from a certain host (unity3d.com), if that's not the case then it will redirect to another URL. This technique can be used to prevent deep linking to your web player content:

```
Application.ExternalEval(
    "if(document.location.host != 'unity3d.com') { document.location='http://unity3d.com'; }"
);
```

Page last updated: 2012-11-15

## Using Web Player templates

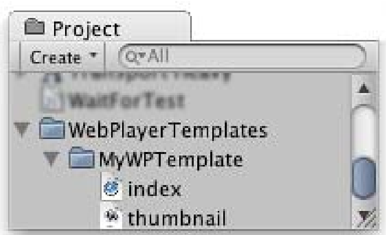
When you build a webplayer project, Unity embeds the player in an HTML page so that it can be played in the browser. The default page is very simple, with just a white background and some minimal text. There are actually three different variations of this page which can be selected from the Player Settings inspector (menu: Edit > Project Settings > Player).




The built-in HTML pages are fine for testing and demonstrating a minimal player but for production purposes, it is often desirable to see the player hosted in the page where it will eventually be deployed. For example, if the Unity content interacts with other elements in the page via the external call interface then it must be tested with a page that provides those interacting elements. Unity allows you to supply your own pages to host the player by using **webplayer templates**.

### Structure of a Webplayer Template

Custom templates are added to a project by creating a folder called "WebPlayerTemplates" in the Assets folder - the templates themselves are sub-folders within this folder. Each template folder contains an index.html or index.php file along with any other resources the page needs, such as images or stylesheets.




Once created, the template will appear among the options on the Player Settings inspector. (the name of the template will be the same as its folder). Optionally, the folder can contain a file named thumbnail.png, which should have dimensions of 128x128 pixels. The thumbnail image will be displayed in the inspector to hint at what the finished page will look like.

### Template Tags

During the build process, Unity will look for special tag strings in the page text and replace them with values supplied by the editor. These include the name, onscreen dimensions and various other useful information about the player.

The tags are delimited by percent signs (%) in the page source. For example, if the product name is defined as "MyPlayer" in the Player settings:-

```
<title>%UNITY_WEB_NAME%</title>
```

...in the template's index file will be replaced with

```
<title>MyPlayer</title>
```

...in the host page generated for the build. The complete set of tags is given below:-

**UNITY_WEB_NAME**
Name of the webplayer.

**UNITY_WIDTH**
**UNITY_HEIGHT**
Onscreen width and height of the player in pixels.

**UNITY_WEB_PATH**
Local path to the webplayer file.

**UNITY_UNITYOBJECT_LOCAL**
A browser JavaScript file called UnityObject2.js is generally used to embed the player in the host page and provide part of the interaction between Unity and the host's JavaScript. This is normally supplied to a page by downloading from Unity's website. However, this requires an internet connection and causes problems if the page is to be deployed offline from the user's hard drive. This tag provides the local path to the UnityObject.js file, which will be generated if the Offline Deployment option is enabled in the Build Settings.

**UNITY_UNITYOBJECT_URL**
In the usual case where the page will download UnityObject2.js from the Unity's website (ie, the Offline Deployment option is disabled), this tag will provide the download URL.

**UNITY_UNITYOBJECT_DEPENDENCIES**
The UnityObject2.js have dependencies and this tag will be replaced with the needed dependencies for it to work properly.

**UNITY_BETA_WARNING**
If the webplayer has been built with a beta version of Unity, this tag will be replaced with a short warning message about the fact. Otherwise, it is replaced with nothing.

**UNITY_CUSTOM_SOME_TAG**
If you add a tag to the index file with the form UNITY_CUSTOM_XXX, then this tag will appear in the Player Settings when your template is selected. For example, if something like

```
<title>Unity Web Player | %UNITY_CUSTOM_MYTAG%</title>
```

...is added to the source, the Player Settings will look like this:-

Resolution and Presentation
Resolution
Default Screen Width 600
Default Screen Height 450
Run In Background*
WebPlayer Template
MyWPTemplate Black Background Default
No Context Menu
Mytag
Watermark
Always Display Watermark*
* Shared setting between multiple platforms.

The textbox next to the tag's name contains the text that the custom tag will be replaced with during the build.

## Example

To illustrate the use of the template tags, here is the HTML source that Unity uses for its default webplayer build.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
	<head>
		<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
		<title>Unity Web Player | %UNITY_WEB_NAME%</title>
		%UNITY_UNITYOBJECT_DEPENDENCIES%
		<script type="text/javascript">
		<!--
		var unityObjectUrl = "%UNITY_UNITYOBJECT_URL%";
		if (document.location.protocol == 'https:')
			unityObjectUrl = unityObjectUrl.replace("http://", "https://ssl-");
		document.write('<script type="text\/javascript" src="' + unityObjectUrl + '"><\/script>');
		-->
		</script>
		<script type="text/javascript">
		<!--
			jQuery(function() {
				var config = {
					width: %UNITY_WIDTH%,
					height: %UNITY_HEIGHT%,
					params: %UNITY_PLAYER_PARAMS%
				};
				var u = new UnityObject2(config);

				var $missingScreen = jQuery("#unityPlayer").find(".missing");
				var $brokenScreen = jQuery("#unityPlayer").find(".broken");
				$missingScreen.hide();
				$brokenScreen.hide();
```

```
				u.observeProgress(function (progress) {
					switch(progress.pluginStatus) {
						case "broken":
							$brokenScreen.find("a").click(function (e) {
								e.stopPropagation();
								e.preventDefault();
								u.installPlugin();
								return false;
							});
							$brokenScreen.show();
						break;
						case "missing":
							$missingScreen.find("a").click(function (e) {
								e.stopPropagation();
								e.preventDefault();
								u.installPlugin();
								return false;
							});
							$missingScreen.show();
						break;
						case "installed":
							$missingScreen.remove();
						break;
						case "first":
						break;
					}
				});
				u.initPlugin(jQuery("#unityPlayer")[0], "%UNITY_WEB_PATH%");
			});
		-->
		</script>
		<style type="text/css">
		<!--
		body {
			font-family: Helvetica, Verdana, Arial, sans-serif;
			background-color: white;
			color: black;
			text-align: center;
		}
		a:link, a:visited {
			color: #000;
		}
		a:active, a:hover {
			color: #666;
		}
		p.header {
			font-size: small;
		}
		p.header span {
			font-weight: bold;
		}
		p.footer {
			font-size: x-small;
		}
		div.content {
			margin: auto;
			width: %UNITY_WIDTH%px;
		}
		div.broken,
		div.missing {
			margin: auto;
```

```
                position: relative;
                top: 50%;
                width: 193px;
            }
            div.broken a,
            div.missing a {
                height: 63px;
                position: relative;
                top: -31px;
            }
            div.broken img,
            div.missing img {
                border-width: 0px;
            }
            div.broken {
                display: none;
            }
            div#unityPlayer {
                cursor: default;
                height: %UNITY_HEIGHT%px;
                width: %UNITY_WIDTH%px;
            }
            -->
            </style>
        </head>
        <body>
            <p class="header"><span>Unity Web Player | </span>%UNITY_WEB_NAME%</p>%UNITY_BETA_WARNING%
            <div class="content">
                <div id="unityPlayer">
                    <div class="missing">
                        <a href="http://unity3d.com/webplayer/" title="Unity Web Player. Install now!">
                            <img alt="Unity Web Player. Install now!" src="http://webplayer.unity3d.com/installation/
                        </a>
                    </div>
                    <div class="broken">
                        <a href="http://unity3d.com/webplayer/" title="Unity Web Player. Install now! Restart your brow
                            <img alt="Unity Web Player. Install now! Restart your browser after install." src="http://w
                        </a>
                    </div>
                </div>
            </div>
            <p class="footer">&laquo; created with <a href="http://unity3d.com/unity/" title="Go to unity3d.com">Unity</a:
        </body>
</html>
```

Page last updated: 2012-11-15

---

# Web Player Streaming

Web Player Streaming is critical for providing a great web gaming experience for the end user. The idea behind web games is that the user can view your content almost immediately and start playing the game as soon as possible instead of making him wait for a progress bar. This is very achievable, and we will explain how.

### Tuning for Portals

This section will focus on publishing to online game portals. Streaming is useful for all kinds of contents, and it can easily be applied to many other situations.

Online game portals expect that some form of game play really starts after downloading at most 1 MB of data. If you don't

reach this makes it that much less likely for a portal to accept your content. From the user's perspective, the game needs to start quickly. Otherwise his time is being wasted and he might as well close the window.

On a 128 kilobit cable connection you can download 16 KB per second or 1 MB per minute. This is the low end of bandwidth online portals target.

The game would optimally be set up to stream something like this:

1. 50 KB display the logo and menu (4 seconds)
2. 320 KB let the user play a tiny tutorial level or let him do some fun interaction in the menu (20 seconds)
3. 800 KB let the user play the first small level (50 seconds)
4. Finish downloading the entire game within 1-5 MB (1-5 minutes)

The key point to keep in mind is to think in wait times for a user on a slow connection. Never let him wait.

Now, don't panic if your web player currently is 10 MB. It seems daunting to optimize it, but it turns out that with a little effort it is usually quite easy to structure your game in this fashion. Think of each above step as an individual scene. If you've made the game, you've done the hard part already. Structuring some scenes around this loading concept is a comparative breeze!

If you open the console log (**Open Editor Log** button in the Console window(**Desktop Platforms**); **Help -> Open Editor console log** menu **OSX**) after or during a build, you can see the size of each individual scene file. The console will show something like this:

```
***Player size statistics***
Level 0 'Main Menu' uses 95.0 KB compressed.
Level 1 'Character Builder' uses 111.5 KB compressed.
Level 2 'Level 1' uses 592.0 KB compressed.
Level 3 'Level 2' uses 2.2 MB compressed.
Level 4 'Level 3' uses 2.3 MB compressed.
Total compressed size 5.3 MB. Total decompressed size 9.9 MB.
```

This game could use a little more optimization! For more information, we recommend you read the reducing file size page.

### The Most Important Steps

1. Load the menu first. Showing an animated logo is a great way to make time go by unnoticed, thus letting the download progress further.

2. Make the first level be short and not use a lot of assets. This way, the first level can be loaded quickly, and by keeping the player occupied playing it for a minute or two you can be sure that the download of all remaining assets can be completed in the background. Why not have a mini tutorial level where the user can learn the controls of the game? No reason for high-res textures here or loads of objects, or having all your enemies in the first level. Use the one with the lowest poly-count. And yes, this means you might have to design your game with the web player experience in mind.

3. There is no reason why all music must be available when the game starts. Externalize the music and load it via the WWW class. Unity compresses audio with the high quality codec, Ogg Vorbis. However even when compressed, audio takes up a lot of space, and if you have to fit things into 3 MB, if you have 5 minutes of music all the compression in the world won't save you. Sacrifices are needed. Load a very short track that you can loop until more music has been downloaded. Only load more music when the player is hooked on the first level.

4. Optimize your textures using their Import Settings. After you externalize music, textures easily take up 90% of the game. Typical texture sizes are too big for web deployment. In a small browser window, sometimes big textures don't even increase the visual fidelity at all. Make sure you use textures that are only as big as they must be (and be ready for more sacrifices here). Halving the texture resolution actually makes the texture size a quarter of what it was. And of course all textures should be DXT compressed.

5. Generally reduce the size of your web players. There is a manual page committed to the utilities Unity offers for optimizing file size here. Although Unity uses cutting edge LZMA-based compression which usually compresses game data to anywhere from one half to a third of the uncompressed size, you'll need to try everything you can.

6. Try to avoid Resources.Load. While Resources.Load can be very handy, Unity will not be able to order your assets by when they are first used when you use Resources.Load, because any script could attempt to load the Resource. You

can set which level will include all assets that can be loaded through Resources.Load in the **Edit->Project Settings->Player** using the **First Streamed Level With Resources** property. Obviously you want to move Resources.Load assets as late as possible into the game or not use the feature at all.

### Publishing Streaming Web Players

Streaming in Unity is level based, and there is an easy workflow to set this up. Internally, Unity does all the dirty work of tracking assets and organizing them in the compressed data files optimally, ordering it by the first scene that uses them. You simply have to ensure that the first levels in the Build Settings use as few assets as possible. This comes naturally for a "menu level", but for a good web experience you really need make sure that the first actual game levels the player is going to play are small too.

In order to use streaming in Unity, you select **Web Player Streamed** in the Build Settings. Then the content automatically starts as soon as all assets used by the first level are loaded. Try to keep the "menu level" to something like 50-100 KB. The stream continues to load as fast as it can, and meanwhile live decompresses. When you look at the Console during/after a build, you can see how large

You can query the progress of the stream by level, and once a level is available it can be loaded. Use GetStreamProgressForLevel for displaying a progress bar and CanStreamedLevelBeLoaded to check if all the data is available to load a specific level.

This form of streaming is of course linear, which matches how games work in most cases. Sometimes that's not enough. Therefore Unity also provides you with API's to load a .unity3d file manually using the WWW class. Video and audio can be streamed as well, and can start playing almost immediately, without requiring the movie to be downloaded first. Finally Textures can easily be downloaded via the WWW class, as can any textual or binary data your game might depend on.

Page last updated: 2012-10-12

---

## WebPlayerChannels

Whenever a new major version of Unity is released, the webplayer plugin for the browser is also updated to take advantage of the latest features. The plugin checks for new releases as it starts up and will automatically upgrade itself when necessary.

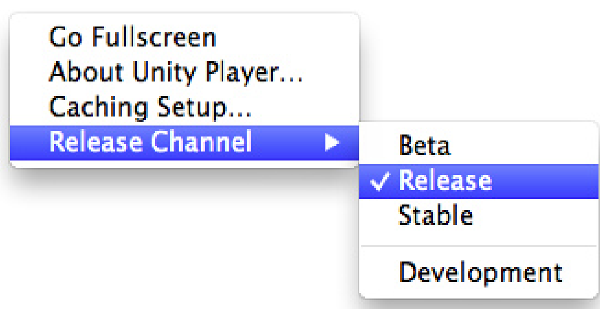
Although Unity does take backward compatibility very seriously, it is still possible that a new release could introduce bugs that cause problems with existing webplayers. Since the plugin updates automatically, such bugs could leave a user unable to play a web game until they are fixed and in the meantime, there may be no way to get the working version of the runtime back.

To avoid problems like this, the plugin is designed to keep a number of different versions of the Unity runtime, which are called **Release Channels**. When an update to the runtime is released, the plugin automatically retrieves it but also retains a copy of the previous version. The latest channel is named **Release** while the previous one is named **Stable**. By default, the plugin will use the oldest channel that supports all the features needed by a given webplayer. This enables new games to make use of new features but avoids the risk that new bugs will take down existing games.

Some time after a release, when the Release channel has been thoroughly tested for stability, it will be migrated to Stable status and it will be used as the default for all players from then on.

### Selecting a channel

Although the channel system avoids problems with upgrading, it is still useful to be able to test your game with the latest runtime. The Unity browser plugin lets you select the desired channel from the context menu (right-click on the content area in the web page).




The channels are listed in order with the most recent at the top. When you select an item from the menu, it will be used as the minimum version for running webplayer content (so if you select the Release channel, the older Stable channel will not be used). You can simply select the oldest version to restore the default channel selection. Normally, just the Release and Stable channels will be available but the system is open-ended so other channels (say for Beta testing) may appear from time to time.

Additionally, the menu gives you the option to switch to Development mode, which enables debugging and profiling in the webplayer and also offers an error console will appear whenever the player's code throws exceptions.

Page last updated: 2012-11-30

---

## Reference

Refer to the information on these pages for details on working in-depth with various aspects of Unity.

The Unity Manual Guide contains sections that apply only to certain platforms. Please select which platforms you want to see. Platform-specific information can always be seen by clicking on the disclosure triangles on each page.

- Components
  - Pathfinding
    - NavMesh Agent (Pro Only)
    - Off-mesh links (Pro only)
    - Navmesh Obstacle
  - Animation Components
    - Animation
    - Animation Clip
    - Animator Component
    - Animator Controller
    - Creating the Avatar
    - Avatar Body Mask
    - Avatar Skeleton Mask
    - Human Template files
    - Animation States
    - Animation Transitions
  - Asset Components
    - Audio Clip
    - Cubemap Texture
    - Meshes
      - Import settings for Meshes
      - FBX Importer, Rig options
      - FBX Importer - Animations Tab
    - Flare

        - Font
        - Material
        - Meshes
        - Movie Texture
        - Procedural Material Assets
        - Render Texture
        - Text Asset
        - Texture 2D
    - Audio Components
        - Audio Listener
        - Audio Source
        - Audio Filters (PRO only)
            - Audio Low Pass Filter (PRO only)
            - Audio High Pass Filter (PRO only)
            - Audio Echo Filter (PRO only)
            - Audio Distortion Filter (PRO only)
            - Audio Reverb Filter (PRO only)
            - Audio Chorus Filter (PRO only)
        - Reverb Zones.
        - Microphone
    - Physics Components
        - Box Collider
        - Capsule Collider
        - Character Controller
        - Character Joint
        - Configurable Joint
        - Constant Force
        - Fixed Joint
        - Hinge Joint
        - Mesh Collider
        - Physics Material
        - Rigidbody
        - Sphere Collider
        - Spring Joint
        - Interactive Cloth
        - Skinned Cloth
        - Wheel Collider
    - The GameObject
        - GameObject
    - Image Effect Scripts
        - Antialiasing (PostEffect)
        - Bloom
        - Camera Motion Blur
        - Depth of Field
        - Noise And Grain
        - Screen Overlay
        - Color Correction Lookup Texture
        - Bloom and Lens Flares
        - Color Correction Curves
        - Contrast Enhance
        - Crease
        - Depth of Field 3.4
        - Tonemapping
        - Edge Detect Effect Normals
        - Fisheye image effect
        - Global Fog
        - Sun Shafts
        - Tilt Shift
        - Vignetting (and Chromatic Aberration)
        - Blur image effect
        - Color Correction image effect

    - Contrast Stretch image effect
    - Edge Detection image effect
    - Glow image effect
    - Grayscale image effect
    - Motion Blur image effect
    - Noise image effect
    - Sepia Tone image effect
    - Screen Space Ambient Occlusion (SSAO) image effect
    - Twirl image effect
    - Vortex image effect
  - Settings Managers
    - Audio Manager
    - Editor settings
    - Input Manager
    - NavMesh Layers (Pro only)
    - Network Manager
    - Physics Manager
    - Player Settings
    - Quality Settings
    - Render Settings
    - Script Execution Order Settings
    - Tag Manager
    - Time Manager
  - Mesh Components
    - Mesh Filter
    - Mesh Renderer
    - Skinned Mesh Renderer
    - Text Mesh
  - Network Group
    - Network View
  - Effects
    - Particle System (Shuriken)
    - Halo
    - Lens Flare
    - Line Renderer
    - Trail Renderer
    - Projector
    - Particle Systems (Legacy, prior to release 3.5)
      - Ellipsoid Particle Emitter (Legacy)
      - Mesh Particle Emitter (Legacy)
      - Particle Animator (Legacy)
      - Particle Renderer (Legacy)
      - World Particle Collider (Legacy)
  - Rendering Components
    - Camera
    - Flare Layer
    - GUI Layer
    - GUI Text
    - GUI Texture
    - Light
    - Light Probe Group
    - Occlusion Area (Pro Only)
    - Occlusion Portals
    - Skybox
    - Level of Detail (Pro Only)
    - 3D Textures
  - Transform Component
    - Transform
  - UnityGUI Group
    - GUI Skin
    - GUI Style

  - Wizards
    - Ragdoll Wizard
- Terrain Engine Guide
  - Using Terrains
  - Height
  - Textures
  - Trees
  - Grass
  - Detail Meshes
  - Terrain Engine Guide
  - Other Settings
- Tree Creator Guide
  - Building Your First Tree
  - Tree Creator Structure
  - Branch Group Properties
  - Leaf Group Properties
  - Tree - Wind Zones
- Animation View Guide
  - Using the Animation View (Legacy)
  - Using Animation Curves (Legacy)
  - Editing Curves
  - Objects with Multiple Moving Parts
  - Using Animation Events
- GUI Scripting Guide
  - GUI Basics
  - Controls
  - Customization
  - Layout Modes
  - Extending UnityGUI
  - Extending the Editor
- Network Reference Guide
    - Networking on Mobile devices.
    - High Level Networking Concepts
    - Networking Elements in Unity
    - Network Views
    - RPC Details
    - State Synchronization Details
    - Network Instantiate
    - Network Level Loading
    - Master Server
      - Building the Unity Networking Servers on your own
    - Minimizing Network Bandwidth
    - Social API
- Built-in Shader Guide
  - Performance of Unity shaders
  - Normal Shader Family
    - Vertex-Lit
    - Diffuse
    - Specular
    - Bumped Diffuse
    - Bumped Specular
    - Parallax Diffuse
    - Parallax Bumped Specular
    - Decal
    - Diffuse Detail
  - Transparent Shader Family
    - Transparent Vertex-Lit
    - Transparent Diffuse
    - Transparent Specular
    - Transparent Bumped Diffuse
    - Transparent Bumped Specular

        - Transparent Parallax Diffuse
        - Transparent Parallax Specular
    - Transparent Cutout Shader Family
        - Transparent Cutout Vertex-Lit
        - Transparent Cutout Diffuse
        - Transparent Cutout Specular
        - Transparent Cutout Bumped Diffuse
        - Transparent Cutout Bumped Specular
    - Self-Illuminated Shader Family
        - Self-Illuminated Vertex-Lit
        - Self-Illuminated Diffuse
        - Self-Illuminated Specular
        - Self-Illuminated Normal mapped Diffuse
        - Self-Illuminated Normal mapped Specular
        - Self-Illuminated Parallax Diffuse
        - Self-Illuminated Parallax Specular
    - Reflective Shader Family
        - Reflective Vertex-Lit
        - Reflective Diffuse
        - Reflective Specular
        - Reflective Bumped Diffuse
        - Reflective Bumped Specular
        - Reflective Parallax Diffuse
        - Reflective Parallax Specular
        - Reflective Normal Mapped Unlit
        - Reflective Normal mapped Vertex-lit
- Unity's Rendering behind the scenes
    - Deferred Lighting Rendering Path
    - Forward Rendering Path Details
    - Vertex Lit Rendering Path Details
    - Hardware Requirements for Unity's Graphics Features
- Shader Reference
    - Writing Surface Shaders
        - Surface Shader Examples
        - Custom Lighting models in Surface Shaders
        - Surface Shader Lighting Examples
        - Surface Shaders with DX11 Tessellation
    - Writing vertex and fragment shaders
        - Accessing shader properties in Cg
        - Providing vertex data to vertex programs
        - Built-in shader include files
        - Predefined shader preprocessor macros
        - Built-in state variables in shader programs
        - GLSL Shader Programs
    - ShaderLab syntax: Shader
        - ShaderLab syntax: Properties
        - ShaderLab syntax: SubShader
            - ShaderLab syntax: Pass
                - ShaderLab syntax: Color, Material, Lighting
                - ShaderLab syntax: Culling & Depth Testing
                - ShaderLab syntax: Texture Combiners
                - ShaderLab syntax: Fog
                - ShaderLab syntax: Alpha testing
                - ShaderLab syntax: Blending
                - ShaderLab syntax: Pass Tags
                - ShaderLab syntax: Name
                - ShaderLab syntax: BindChannels
            - ShaderLab syntax: UsePass
            - ShaderLab syntax: GrabPass
            - ShaderLab syntax: SubShader Tags
        - ShaderLab syntax: Fallback

        - ShaderLab syntax: other commands
    - Advanced ShaderLab topics
        - Unity's Rendering Pipeline
        - Performance Tips when Writing Shaders
        - Rendering with Replaced Shaders
        - Using Depth Textures
        - Camera's Depth Texture
        - Platform Specific Rendering Differences
        - Shader Level of Detail
    - ShaderLab builtin values
- Scripting Concepts
    - Layers
        - Layer-Based Collision Detection.
    - What is a Tag?
    - Rigidbody Sleeping


Page last updated: 2011-04-05

---

## Components


- Pathfinding
    - NavMesh Agent (Pro Only)
    - Off-mesh links (Pro only)
    - Navmesh Obstacle
- Animation Components
    - Animation
    - Animation Clip
    - Animator Component
    - Animator Controller
    - Creating the Avatar
    - Avatar Body Mask
    - Avatar Skeleton Mask
    - Human Template files
    - Animation States
    - Animation Transitions
- Asset Components
    - Audio Clip
    - Cubemap Texture
    - Meshes
        - Import settings for Meshes
        - FBX Importer, Rig options
        - FBX Importer - Animations Tab
    - Flare
    - Font
    - Material
    - Meshes
    - Movie Texture
    - Procedural Material Assets
    - Render Texture
    - Text Asset
    - Texture 2D
- Audio Components
    - Audio Listener
    - Audio Source
    - Audio Filters (PRO only)
        - Audio Low Pass Filter (PRO only)
        - Audio High Pass Filter (PRO only)
        - Audio Echo Filter (PRO only)
        - Audio Distortion Filter (PRO only)

        - Audio Reverb Filter (PRO only)
        - Audio Chorus Filter (PRO only)
    - Reverb Zones.
    - Microphone
- Physics Components
    - Box Collider
    - Capsule Collider
    - Character Controller
    - Character Joint
    - Configurable Joint
    - Constant Force
    - Fixed Joint
    - Hinge Joint
    - Mesh Collider
    - Physics Material
    - Rigidbody
    - Sphere Collider
    - Spring Joint
    - Interactive Cloth
    - Skinned Cloth
    - Wheel Collider
- The GameObject
    - GameObject
- Image Effect Scripts
    - Antialiasing (PostEffect)
    - Bloom
    - Camera Motion Blur
    - Depth of Field
    - Noise And Grain
    - Screen Overlay
    - Color Correction Lookup Texture
    - Bloom and Lens Flares
    - Color Correction Curves
    - Contrast Enhance
    - Crease
    - Depth of Field 3.4
    - Tonemapping
    - Edge Detect Effect Normals
    - Fisheye image effect
    - Global Fog
    - Sun Shafts
    - Tilt Shift
    - Vignetting (and Chromatic Aberration)
    - Blur image effect
    - Color Correction image effect
    - Contrast Stretch image effect
    - Edge Detection image effect
    - Glow image effect
    - Grayscale image effect
    - Motion Blur image effect
    - Noise image effect
    - Sepia Tone image effect
    - Screen Space Ambient Occlusion (SSAO) image effect
    - Twirl image effect
    - Vortex image effect
- Settings Managers
    - Audio Manager
    - Editor settings
    - Input Manager
    - NavMesh Layers (Pro only)
    - Network Manager

    - Physics Manager
    - Player Settings
    - Quality Settings
    - Render Settings
    - Script Execution Order Settings
    - Tag Manager
    - Time Manager
- Mesh Components
    - Mesh Filter
    - Mesh Renderer
    - Skinned Mesh Renderer
    - Text Mesh
- Network Group
    - Network View
- Effects
    - Particle System (Shuriken)
    - Halo
    - Lens Flare
    - Line Renderer
    - Trail Renderer
    - Projector
    - Particle Systems (Legacy, prior to release 3.5)
        - Ellipsoid Particle Emitter (Legacy)
        - Mesh Particle Emitter (Legacy)
        - Particle Animator (Legacy)
        - Particle Renderer (Legacy)
        - World Particle Collider (Legacy)
- Rendering Components
    - Camera
    - Flare Layer
    - GUI Layer
    - GUI Text
    - GUI Texture
    - Light
    - Light Probe Group
    - Occlusion Area (Pro Only)
    - Occlusion Portals
    - Skybox
    - Level of Detail (Pro Only)
    - 3D Textures
- Transform Component
    - Transform
- UnityGUI Group
    - GUI Skin
    - GUI Style
- Wizards
    - Ragdoll Wizard

Page last updated: 2008-06-16

---

# comp-AIGroup

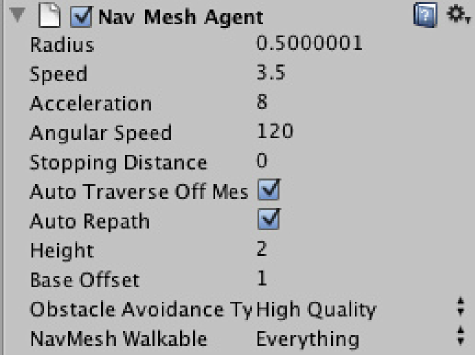
This section covers Unity's support for **pathfinding**, the process of planning an efficient route between two points while avoiding obstacles along the way.

- NavMesh Agent (Pro Only)
- Off-mesh links (Pro only)
- Navmesh Obstacle

Page last updated: 2011-12-01

## class-NavMeshAgent

The **NavMesh Agent** component is used in connection with pathfinding, and is the place to put information about how this agent navigates the NavMesh. You can access it in **Component->Navigation->Nav Mesh Agent**



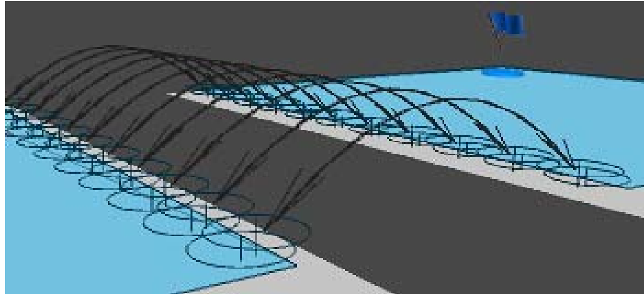

| | |
|---|---|
| **Radius** | Agent radius (used for pathfinding purposes only, and can differ from the actual object's radius, typically larger). |
| **Speed** | Maximum movement speed with which the agent can traverse the world toward its destination. |
| **Acceleration** | Maximum acceleration. |
| **Angular Speed** | Maximum rotation speed in (deg/s). |
| **Stopping distance** | Stopping distance. The agent will decelerate when within this distance to the destination. |
| **Auto Traverse OffMesh Link** | Automate movement onto and off of OffMeshLinks. |
| **Auto Repath** | Acquire new path if existing is partial or invalid. |
| **Height** | The height of the agent (used in debug graphics). |
| **Base offset** | Vertical offset of the collision geometry relative to the actual geometry. |
| **Obstacle Avoidance Type** | The level of quality of avoidance. |
| **NavMesh Walkable** | Specifies the types of Navmesh layers that the agent can traverse. |

(back to Navigation and Pathfinding)

Page last updated: 2012-01-27

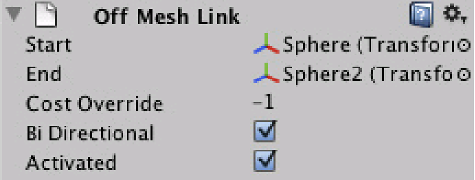
## class-OffMeshLink

Note that this section is primarily about the *manual* off-mesh links, namely those that are set up by the user via the **OffMeshLink** component. For automatically generated off-mesh links, see the the Navmesh intro



It is possible that the *navmesh static* geometry in the scene is disconnected, thus making it impossible for agents to get from one part of the world to the other.

To remedy this, Unity has a system of **Off-mesh links**



*The OffMeshLink component*

An off-mesh link is a component that can be placed on any object, and it has the following properties

| | |
|---|---|
| **Start** | The start object of the off-mesh link. |
| **End** | The end object of the off-mesh link. |
| **Cost Override** | If value is positive, use it when calculating path cost on processing a path request. Otherwise, we use the default cost (cost of the layer to which this game object belongs). If the Cost Override is set to the value 3.0, moving over the off-mesh link will be three times more expensive than moving the same distance on a default NavMesh area. *This property is runtime-editable and does not require a re-bake* |
| **Bi Directional** | If this is on, the link can be traversed both ways, if it's off, the link can only be traversed in the direction from Start to End. |
| **Activated** | Specifies if this link is actually used by the pathfinder. When this property is false, the off-mesh link will be disregarded. *This property is runtime-editable, and does not require a re-bake.* |

### Special notes on OffMeshLink properties

The "Activated" and "Cost Override" properties can be changed at runtime and have immediate effect. All other properties require a Navmesh re-bake before they effect.

If the start or end transforms are unassigned when baking, or if the position of either the start or end transforms is too far away from the NavMesh to find valid positions, the off-mesh links will not be generated. In this case, an error is displayed in the Console window.

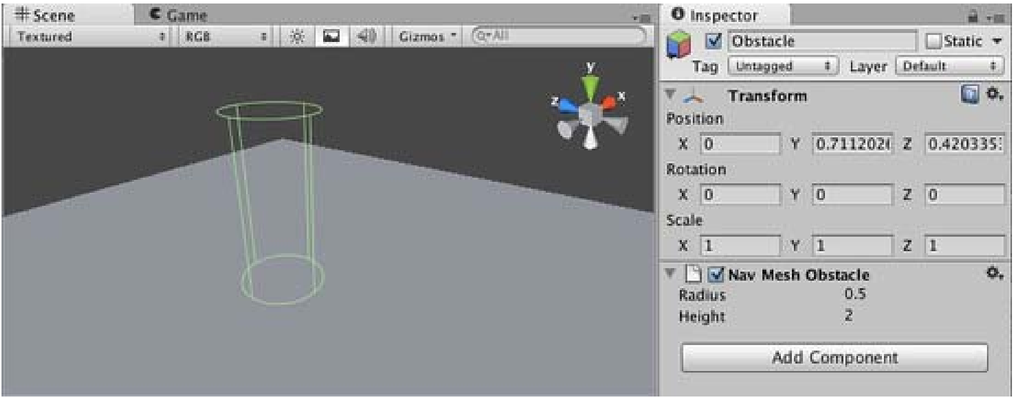
(back to Navigation and Pathfinding)

Page last updated: 2012-01-27

---

## class-NavMeshObstacle

Fixed obstacles on a navmesh can be set up as part of the baking process. However, it is also possible to have dynamic obstacles in a scene which will be avoided by agents as they move around. Such dynamic obstacles can be specified using the Navmesh Obstacle component. This can be added to any GameObject and will move as that object moves.




| | |
|---|---|
| **Radius** | Radius of the obstacle cylinder. |
| **Height** | Height of the obstacle cylinder. |

Page last updated: 2012-09-03

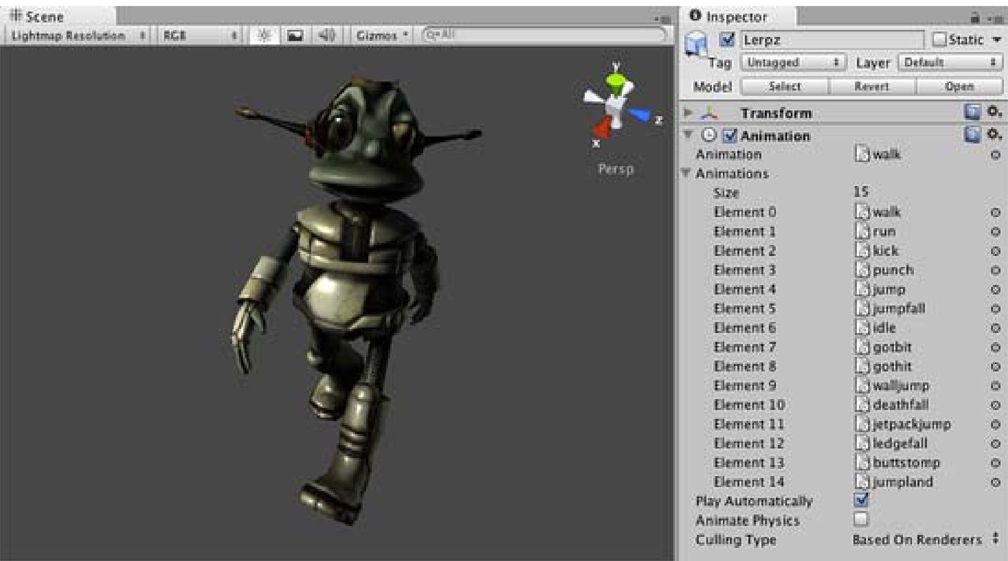
## comp-AnimationGroup

For a detailed explanation of the Mecanim Animation System, see Mecanim introduction

- Animation
- Animation Clip
- Animator Component
- Animator Controller
- Creating the Avatar
- Avatar Body Mask
- Avatar Skeleton Mask
- Human Template files
- Animation States
- Animation Transitions

Page last updated: 2012-11-12

## class-Animation




*The Animation* ***Inspector***

### Properties

| | |
|---|---|
| **Animation** | The default animation that will be played when Play Automatically is enabled. |
| **Animations** | A list of animations that can be accessed from scripts. |
| **Play Automatically** | Should the animation be played automatically when starting the game? |
| **Animate Physics** | Should the animation interact with physics. |
| **Culling Type** | Determines when the animation will not be played. |
| **Always Animate** | Always animate. |
| **Based on Renderers** | Cull based on the default animation pose. |
| **Based on Clip Bounds** | Cull based on clip bounds (calculated during import), if the clip bounds are out of view, the animation will not be played. |
| **Based on User Bounds** | Cull based on bounds defined by the user, if the user-defined bounds are out of view, the animation will not be played. |

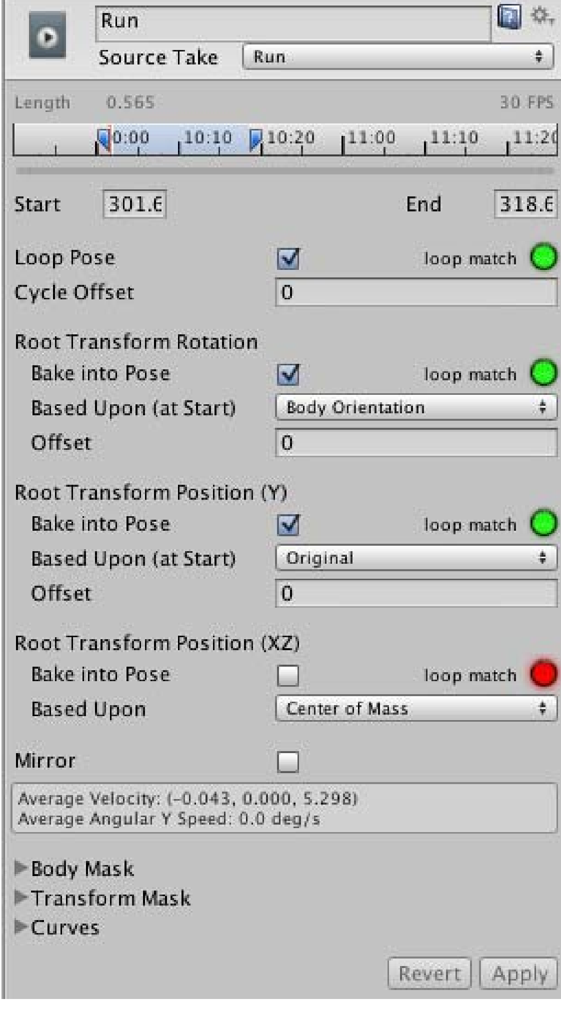
See the Animation View Guide for more information on how to create animations inside Unity. See the Animation Import page on how to import animated characters, or the Animation Scripting page on how to create animated behaviors for your game.

Page last updated: 2012-11-10

---

## class-AnimationClip

**Animation Clips** are the smallest building blocks of animation in Unity. They represent an isolated piece of motion, such as RunLeft, Jump, or Crawl, and can be manipulated and combined in various ways to produce lively end results (see Animation State Machines, Animator Controller, or Blend Trees).
Animation clips can be selected from imported FBX data (see FBXImporter settings for Animations), and when you click on the set of available animation clips you will see the following set of properties:




| | |
|---|---|
| **Name** | The name of the clip. |
| **Source Take** | The take in the source file to use as a source for this animation clip. (This option will not show up if there's only one take). This is what defines a set of animation as separated in Motionbuilder, Maya and other 3D packages. Unity can import these takes as individual clips or you can create them from the whole file or a take. |
| **Start** | Start frame of the clip. |
| **End** | End frame of the clip. |
| **Loop Pose** | Enable to make the motion loop seamlessly. |
| **Cycle Offset** | Offset to the cycle of a looping animation, if we want to start it at a different time. |
| **Root Transform Rotation** | |
| **Bake into Pose** | Enable to make root rotation be baked into the movement of the bones. Disable to make root rotation be stored as root motion. |
| **Based Upon** | What the root rotation is based upon. |
| - **Original** | Keeps the rotation as it is authored in the source file. |
| - **Body Orientation** | Keeps the upper body pointing forward. |
| **Offset** | Offset to the root rotation (in degrees). |
| **Root Transform Position (Y)** | |

| | |
|---|---|
| **Bake into Pose** | Enable to make vertical root motion be baked into the movement of the bones. Disable to make vertical root motion be stored as root motion. |
| **Based Upon** | What the vertical root position is based upon. |
| - **Original** | Keeps the vertical position as it is authored in the source file. |
| - **Center of Mass** | Keeps the center of mass aligned with root transform position. |
| - **Feet** | Keeps the feet aligned with the root transform position. |
| **Offset** | Offset to the vertical root position. |
| **Root Transform Position (XZ)** | |
| **Bake into Pose** | Enable to make horizontal root motion be baked into the movement of the bones. Disable to make horizontal root motion be stored as root motion. |
| **Based Upon** | What the horizontal root position is based upon. |
| - **Original** | Keeps the horizontal position as it is authored in the source file. |
| - **Center of Mass** | Keeps the center of mass aligned with the root transform position. |
| **Offset** | Offset to the horizontal root position. |
| **Mirror** | Mirror left and right in this clip. |
| **Body Mask** | The Body mask applied to this animation clip (see section on body masks). |
| **Curves** (Unity Pro only) | Parameter-related curves (see Curves in Mecanim). |

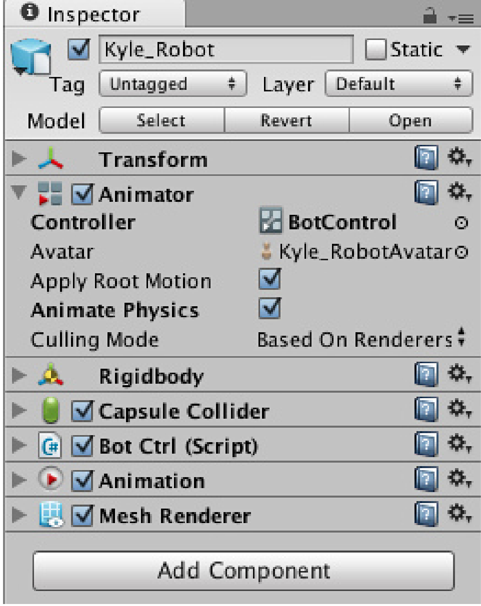
Creating clips is essentially defining the start and end points for segments of animation. In order for these clips to loop, they should be trimmed in such a way to match the first and last frame as best as possible for the desired loop. For more on this, see the section on Looping animation clips

Page last updated: 2012-11-12

---

## class-Animator

Any GameObject that has an avatar will also have an **Animator** component, which is the link between the character and its behavior.




The **Animator** component references an **Animator Controller** which is used for setting up behavior on the character. This includes setup for State Machines, Blend Trees, and events to be controlled from script.

### Properties

| | |
|---|---|
| **Controller** | The animator controller attached to this character |
| **Avatar** | The Avatar for this character. |
| **Apply Root Motion** | Should we control the character's position from the animation itself or from script. |
| **Animate Physics** | Should the animation interact with physics? |

| | |
|---|---|
| **Culling Mode** | Culling mode for animations |
| **Always animate** | Always animate, don't do culling |
| **Based on Renderers** | When the renderers are invisible, only root motion is animated. All other body parts will remain static while the character is invisible. |

Page last updated: 2012-11-01

---

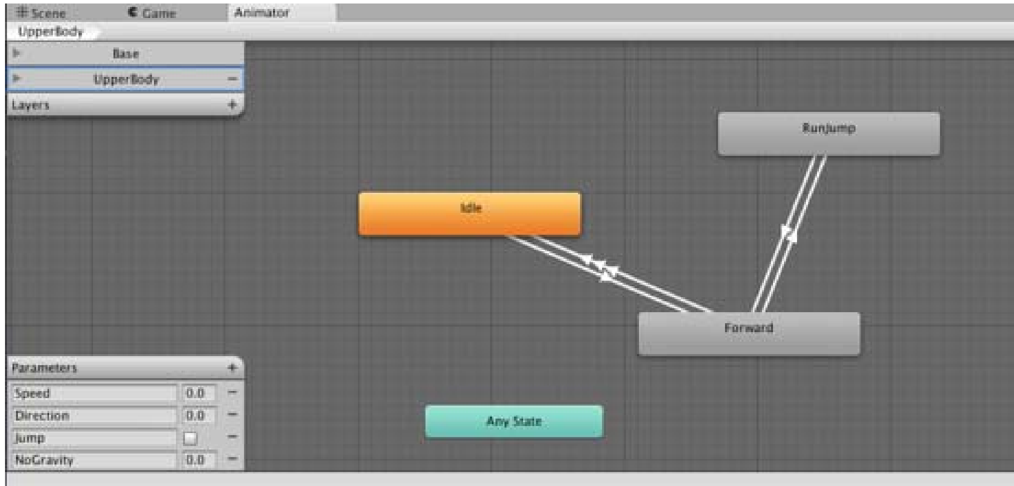
# class-AnimatorController

You can view and set up character behavior from the Animator Controller view (Menu: **Window > Animator Controller**).

An **Animator Controller** can be created from the Project View (Menu: **Create > Animator Controller**). This creates a `.controller` asset on disk, which looks like this in the Project Browser

Animator Controller asset on disk

After the state machine setup has been made, you can drop the controller onto the Animator component of any character with an Avatar in the Hierarchy View.




The Animator Controller Window

The Animator Controller Window will contain

- The Animation Layer Widget (top-left corner, see Animation Layers)
- The Event Parameters Widget (bottom-left, see Animation Parameters)
- The visualization of the State Machine itself.

Note that the Animator Controller Window will always display the state machine from the most recently selected `.controller` asset, regardless of what scene is currently loaded.

Page last updated: 2012-11-14

---

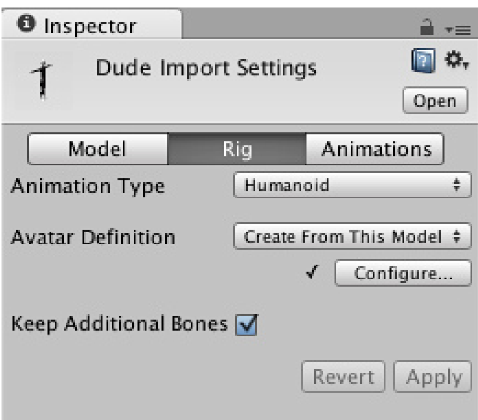
# class-Avatar

After an FBX file is imported, you can specify what kind of rig it is in the Rig tab of the FBX importer options.

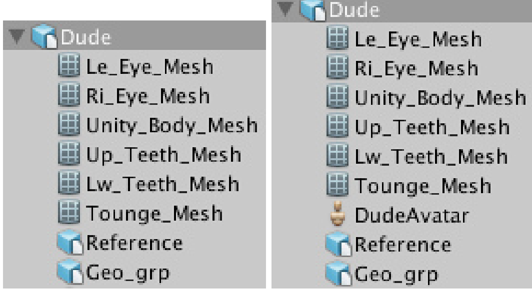
### Humanoid animations

For a Humanoid rig, select **Humanoid** and click **Apply**. Mecanim will attempt to match up your existing bone structure to the Avatar bone structure. In many cases, it can do this automatically by analysing the connections between bones in the rig.

If the match has succeeded, you will see a check mark next to the **Configure...** menu




Also, in the case of a successful match, an Avatar sub-asset is added to the FBX asset, which you will be able to see in the project view hierarchy.




*Models with and without an Avatar sub-asset*

*The inspector for an Avatar asset*

If Mecanim was unable to create the Avatar, you will see a cross next to the **Configure ...** button, and no Avatar sub-asset will be added. When this happens, you need to configure the avatar manually.

### Non-humanoid animations

Two options for non-humanoid animation are provided: **Generic** and **Legacy**. Generic animations are imported using the Mecanim system but don't take advantage of the extra features available for humanoid animations. Legacy animations use the the animation system that was provided by Unity before Mecanim. There are some cases where it is still useful to work with legacy animations (most notably with legacy projects that you don't want to update fully) but they are seldom needed for new projects. See this section of the manual for further details on legacy animations.

(back to Avatar Creation and Setup)

(back to Mecanim introduction)

Page last updated: 2012-10-18

---

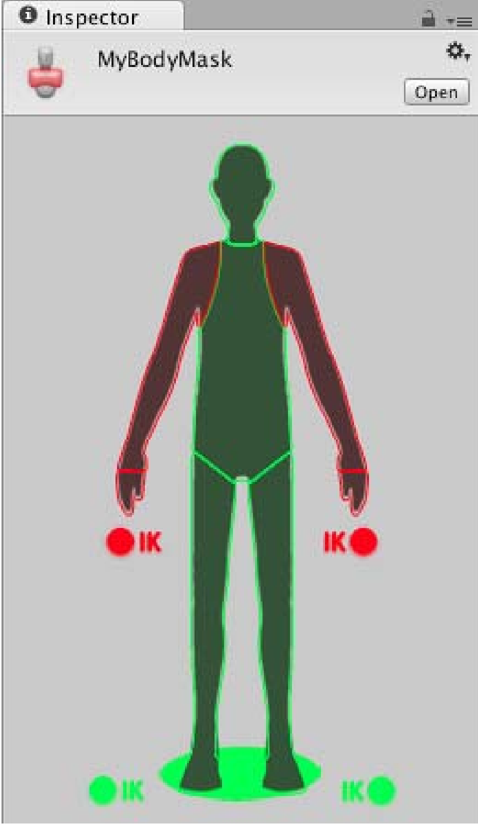
## class-AvatarBodyMask

Specific body parts can be selectively enabled or disabled in an animation using a so-called **Body Mask**. Body masks are used in the Animation tab of the mesh import inspector and Animation Layers. Body masks enable you to tailor an animation to fit the specific requirements of your character more closely. For example, you may have a standard walking animation that includes both arm and leg motion, but if a character is carrying a large object with both hands then you wouldn't want his arms

to swing by his sides as he walks. However, you could still use the standard walking animation by switching off the arm movements in the body mask.

The body parts included are: Head, Left Arm, Right Arm, Left Hand, Right Hand, Left Leg, Right Leg and Root (which is denoted by the "shadow" under the feet). In the body mask, you can also toggle **inverse kinematics** (IK) for hands and feet, which will determine whether or not IK curves will be included in animation blending.

- Click the avatar section to toggle inclusion or exclusion (green/red)
- Double click in empty space surrounding the avatar to toggle all



*Body mask in the Body Mask inspector (arms excluded)*

In the Animation tab of the mesh import inspector, you will see a list entitled *Clips* that contains all the object's animation clips. When you select an item from this list, options for the clip will be shown, including the body mask editor.

You can also create Body Mask Assets (**Assets->Create->Avatar Body Mask**), which show up as .mask files on disk.

The BodyMask assets can be reused in Animator Controllers, when specifying Animation Layers

A benefit of using body masks is that they tend to reduce memory overheads since body parts that are not active do not need their associated animation curves. Also, the unused curves need not be calculated during playback which will tend to reduce the CPU overhead of the animation.

(back to Mecanim introduction)

Page last updated: 2012-11-09

---

## class-AvatarSkeletonMask

similar to AvatarBodyMask, except used for generic animations.

Page last updated: 2012-10-18

## class-HumanTemplate

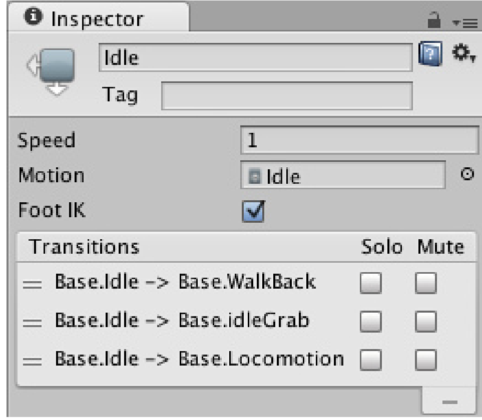
You can save the mapping of bones in your skeleton to the Avatar on disk as a "human template file" (extention `*.ht`), which can be reused by any characters that use this mapping. This is useful, for example, if your animators use a consistent layout and naming convention for all skeleton but Mecanim doesn't know how to interpret it. You can then **Load** the .ht file for each model, so that manual remapping only needs to be done once.

Page last updated: 2012-11-05

## class-State

**Animation States** are the basic building blocks of an **Animation State Machine**. Each state contains an individual animation sequence (or blend tree) which will play while the character is in that state. When an event in the game triggers a state transition, the character will be left in a new state whose animation sequence will then take over.

When you select a state in the Animator Controller, you will see the properties for that state in the inspector:-



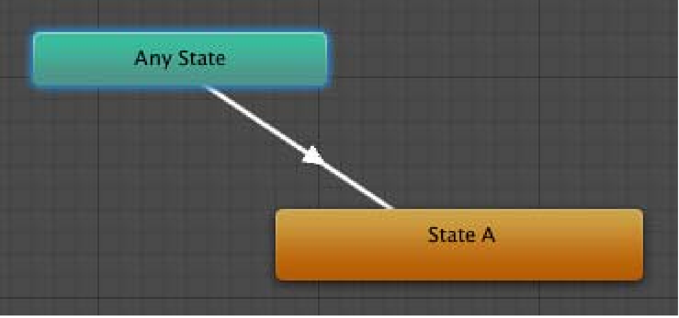

| | |
|---|---|
| **Speed** | The default speed of the animation |
| **Motion** | The animation clip assigned to this state |
| **Foot IK** | Should Foot IK be respected for this state |
| **Transitions** | The list of transitions originating from this state |

The default state, displayed in brown, is the state that the machine will be in when it is first activated. You can change the default state, if necessary, by right-clicking on another state and selecting **Set As Default** from the context menu. The *solo* and *mute* checkboxes on each transition are used to control the behaviour of **animation previews** - see this page for further details.

A new state can be added by right-clicking on an empty space in the Animator Controller Window and selecting **Create State->Empty** from the context menu. Alternatively, you can drag an animation into the Animator Controller Window to create a state containing that animation. (Note that you can only drag Mecanim animations into the Controller - non-Mecanim animations will be rejected.) States can also contain Blend Trees.

**Any State**

**Any State** is a special state which is always present. It exists for the situation where you want to go to a specific state regardless of which state you are currently in. This is a shorthand way of adding the same outward transition to all states in your machine. Note that the special meaning of **Any State** implies that it cannot be the end point of a transition (ie, jumping to "any state" cannot be used as a way to pick a random state to enter next).




(back to Animation State Machines)

Page last updated: 2012-11-14

---

## class-Transition

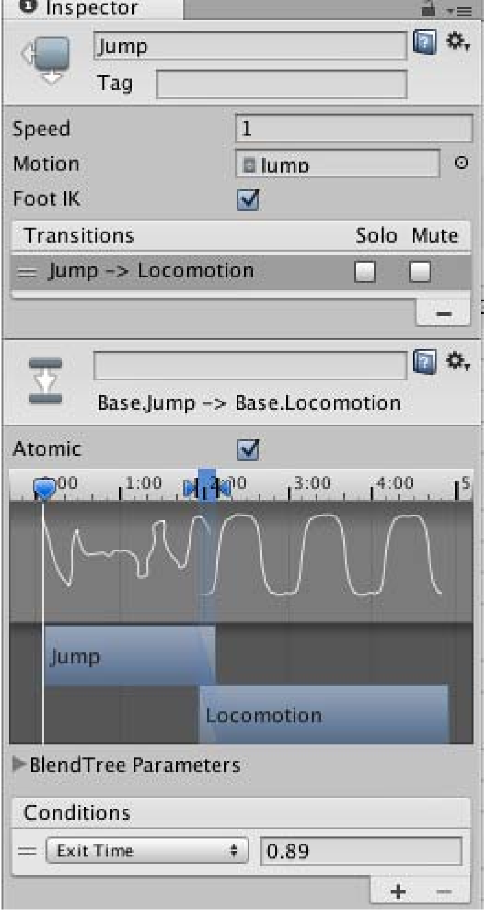
**Animation Transitions** define *what* happens when you switch from one **Animation State** to another. There can be only one transition active at any given time.

| | |
|---|---|
| **Atomic** | Is this transition atomic? (cannot be interrupted) |
| **Conditions** | Here we decide *when* transitions get triggered. |

A condition consists of:

- An event parameter
    - Instead of a parameter, you can also use **Exit Time**, and specify a number which represents the normalized time of the source state (e.g. 0.95 means the transition will trigger, when we've played the source clip 95% through).
- A conditional predicate, if needed (for example **Less**/**Greater** for floats).
- A parameter value (if needed).

You can adjust the transition between the two animation clips by dragging the start and end values of the overlap.




(See also Transition solo / mute)

(back to Animation State Machines)

Page last updated: 2012-11-14

---

# comp-AssetsGroup

**Assets** are the models, textures, sounds and all other "content" files from which you make your game.

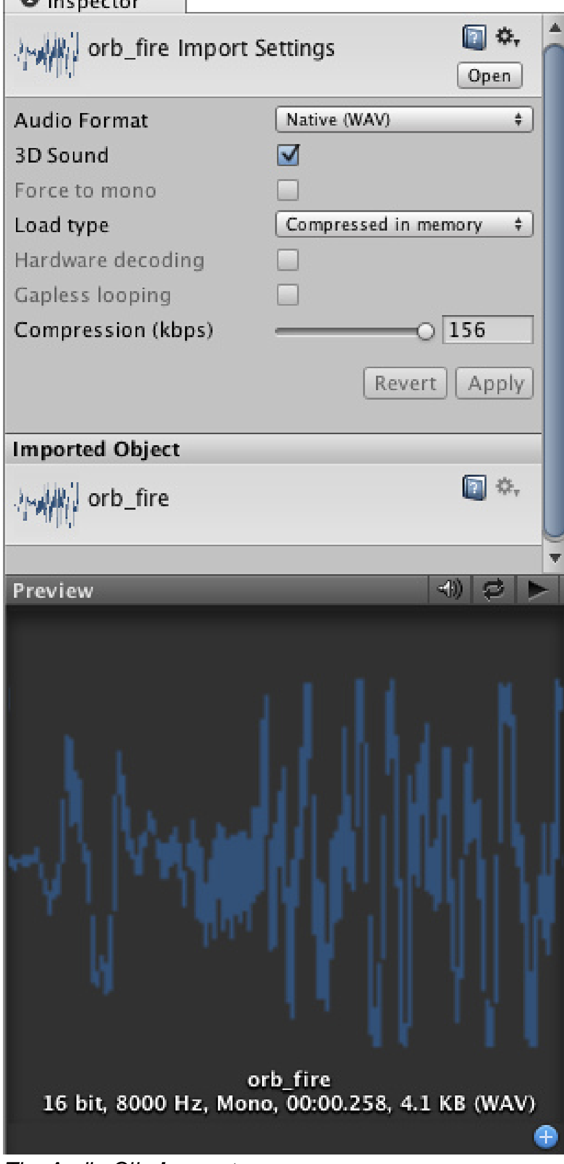
This section describes **Components** for all asset types. For a general overview of assets, see Assets overview page.

- Audio Clip
- Cubemap Texture
- Meshes
  - Import settings for Meshes
  - FBX Importer, Rig options
  - FBX Importer - Animations Tab
- Flare
- Font
- Material
- Meshes
- Movie Texture
- Procedural Material Assets
- Render Texture
- Text Asset
- Texture 2D

Page last updated: 2010-09-02

## class-AudioClip

**Audio Clips** contain the audio data used by Audio Sources. Unity supports mono, stereo and multichannel audio assets (up to eight channels). The audio file formats that Unity can import are **.aif**, **.wav**, **.mp3**, and **.ogg**. Unity can also import tracker modules in the **.xm**, **.mod**, **.it**, and **.s3m** formats. The tracker module assets behave the same way as any other audio assets in Unity although no waveform preview is available in the asset import inspector.




*The Audio Clip **Inspector***

### Properties

| | |
|---|---|
| **Audio Format** | The specific format that will be used for the sound at runtime. |
| **Native** | This option offers higher quality at the expense of larger file size and is best for very short sound effects. |
| **Compressed** | The compression results in smaller files but with somewhat lower quality compared to native audio. This format is best for medium length sound effects and music. |
| **3D Sound** | If enabled, the sound will play back in 3D space. Both Mono and Stereo sounds can be played in 3D. |
| **Force to mono** | If enabled, the audio clip will be down-mixed to a single channel sound. |
| **Load Type** | The method Unity uses to load audio assets at runtime. |
| **Decompress on load** | Audio files will be decompressed as soon as they are loaded. Use this option for smaller compressed sounds to avoid the performance overhead of decompressing on the fly. Be aware that decompressing sounds on load will use about ten times more memory than keeping them |

| | |
|---|---|
| | compressed, so don't use this option for large files. |
| **Compressed in memory** | Keep sounds compressed in memory and decompress while playing. This option has a slight performance overhead (especially for Ogg/Vorbis compressed files) so only use it for bigger files where decompression on load would use a prohibitive amount of memory. Note that, due to technical limitations, this option will silently switch to *Stream From Disc* (see below) for Ogg Vorbis assets on platforms that use FMOD audio. |
| **Stream from disc** | Stream audio data directly from disc. The memory used by this option is typically a small fraction of the file size, so it is very useful for music or other very long tracks. For performance reasons, it is usually advisable to stream only one or two files from disc at a time but the of streams that can comfortably be handled depends on the hardware. |
| **Compression** | Amount of Compression to be applied to a **Compressed** clip. Statistics about the file size can be seen under the slider. A good approach to tuning this value is to drag the slider to a place that leaves the playback "good enough" while keeping the file small enough for your distribution requirements. |
| **Hardware Decoding** | (iOS only) On iOS devices, Apple's hardware decoder can be used resulting in lower CPU overhead during decompression. Check out platform specific details for more info. |
| **Gapless looping** | (Android/iOS only) Use this when compressing a seamless looping audio source file (in a non-compressed PCM format) to ensure perfect continuity is preserved at the seam. Standard MPEG encoders introduce a short silence at the loop point, which will be audible as a brief "click" or "pop". |

### Importing Audio Assets

Unity supports both *Compressed* and *Native* Audio. Any type of file (except MP3/Ogg Vorbis) will be initially imported as *Native*. Compressed audio files must be decompressed by the CPU while the game is running, but have smaller file size. If *Stream* is checked the audio is decompressed *on the fly*, otherwise it is decompressed completely as soon as it loads. Native PCM formats (WAV, AIFF) have the benefit of giving higher fidelity without increasing the CPU overhead, but files in these formats are typically much larger than compressed files. Module files (.mod,.it,.s3m..xm) can deliver very high quality with an extremely low footprint.

As a general rule of thumb, *Compressed* audio (or modules) are best for long files like background music or dialog, while *Native* is better for short sound effects. You should tweak the amount of Compression using the compression slider. Start with high compression and gradually reduce the setting to the point where the loss of sound quality is perceptible. Then, increase it again slightly until the perceived loss of quality disappears.

### Using 3D Audio

If an audio clip is marked as a **3D Sound** then it will be played back so as to simulate its position in the game world's 3D space. 3D sounds emulate the distance and location of sounds by attenuating volume and panning across speakers. Both mono and multiple channel sounds can be positioned in 3D. For multiple channel audio, use the *spread* option on the Audio Source to spread and split out the discrete channels in speaker space. Unity offers a variety of options to control and fine-tune the audio behavior in 3D space - see the Audio Source component reference for further details.

### Platform specific details

## iOS

On mobile platforms compressed audio is encoded as MP3 to take advantage of hardware decompression.

To improve performance, audio clips can be played back using the Apple hardware codec. To enable this option, check the "Hardware Decoding" checkbox in the Audio Importer. Note that only one hardware audio stream can be decompressed at a time, including the background iPod audio.

If the hardware decoder is not available, the decompression will fall back on the software decoder (on iPhone 3GS or later, Apple's software decoder is used in preference to Unity's own decoder (FMOD)).

## Android

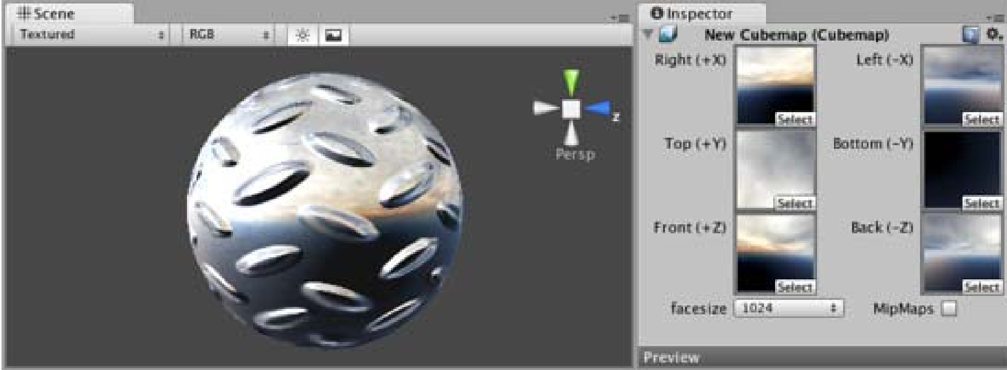
On mobile platforms compressed audio is encoded as MP3 to take advantage of hardware decompression.

Page last updated: 2012-10-31

---

## class-Cubemap

A **Cubemap Texture** is a collection of six separate square Textures that are put onto the faces of an imaginary cube. Most often they are used to display infinitely faraway reflections on objects, similar to how Skybox displays faraway scenery in the background. The Reflective built-in shaders in Unity use Cubemaps to display reflection.




*A mountain scene Cubemap displayed as a reflection on this sphere*

You create Cubemap in one of several ways:

1. Use **Assets->Create->Cubemap**, set its properties, and drag six Texture assets onto corresponding Cubemap "faces". Note that the textures must be re-applied if changed because the textures are baked into the Cubemap Asset and are in no way linked to the textures.
2. Use the Texture Import Settings to create a Cubemap from a single imported texture asset.
3. Render your scene into a cubemap from script. Code example in Camera.RenderToCubemap page contains a script for rendering cubemaps straight from the editor.

### Properties

| | |
|---|---|
| **Right (+X)** | Texture for the right global side of the Cubemap face. |
| **Left (-X)** | Texture for the up global side of the Cubemap face. |
| **Top (+Y)** | Texture for the top global side of the Cubemap face. |
| **Bottom (-Y)** | Texture for the bottom global side of the Cubemap face. |
| **Front (+Z)** | Texture for the forward global side of the Cubemap face. |
| **Back (-Z)** | Texture for the rear global side of the Cubemap face. |
| **Face Size** | Width and Height in pixels across each individual Cubemap face. Textures will be internally scaled to fit this size, there is no need to manually scale the assets. |
| **Mipmap** | Enable to create mipmaps. |
| **Format** | Format of the created cubemap. |

Page last updated: 2010-12-08

---

## class-FBXImporter

When a 3D model is imported, Unity represents it internally as a **Mesh**. A Mesh must be attached to a GameObject using a Mesh Filter component. For the mesh to be visible, the GameObject must also have a Mesh Renderer or other suitable rendering component attached. With these components in place, the mesh will be visible at the GameObject's position with its exact appearance dependent on the Material used by the renderer.

*A Mesh Filter together with Mesh Renderer makes the model appear on screen.*

Unity's mesh importer provides many options for controlling the generation of the mesh and associating it with its textures and materials. These options are covered by the following pages:-

- Import settings for Meshes
- FBX Importer, Rig options
- FBX Importer - Animations Tab

Page last updated: 2012-10-26

---

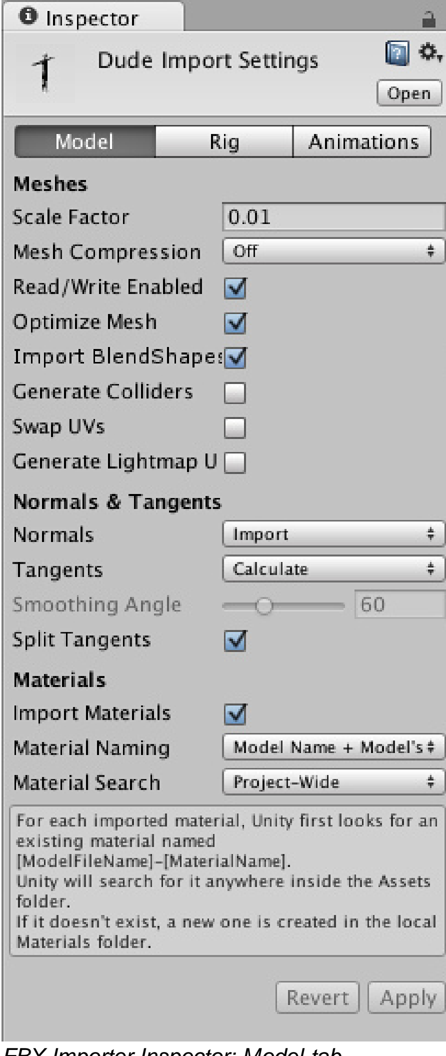
# FBXImporter-Model

The **Import Settings** for a model file will be displayed in the Model tab of the FBX importer inspector when the model is selected. These affect the **mesh**, it's **normals** and imported **materials**. Settings are applied per asset on disk so if you need assets with different settings make (and rename accordingly) a duplicate file.

Although defaults can suffice initially, it is worth studying the settings glossary below, as they can determine what you wish to do with the game object.

Some general adjustments to be made for example might be:

- Scale - this scale factor is used for compensating difference in units between Unity and 3d modeling tool - it rescales whole file. If you do not care about units you can simply set it to 1.
- Generate colliders - this will generate a collison mesh to allow your model to collide with other objects - see notes below.
- Material Naming and Search - this will help you automatically setup your materials and locate textures



*FBX Importer Inspector: Model tab*

| | |
|---|---|
| ***Meshes*** | |
| **Scale Factor** | Unity's physics system expects 1 meter in the game world to be 1 unit in the imported file. If you prefer to model at a different scale then you can compensate for it here. defaults for different 3D packages are as follows .fbx, .max, .jas, .c4d = 0.01, .mb, .ma, .lxo, .dxf, .blend, .dae = 1 .3ds = 0.1 |
| **Mesh Compression** | Increasing this value will reduce the file size of the mesh, but might introduce irregularities. It's best to turn it up as high as possible without the mesh looking too different from the uncompressed version. This is useful for optimizing game size. |
| **Read/Write Enabled** | Enables the mesh to be written at runtime so you can modify the data - makes a copy in memory. |
| **Optimize Mesh** | This option determines the order in which triangles will be listed in the mesh. |
| **Import BlendShapes** | Disable this if your file contains BlendShapes and you don't want them to be imported. |
| **Generate Colliders** | If this is enabled, your meshes will be imported with Mesh Colliders automatically attached. This is useful for quickly generating a collision mesh for environment geometry, but should be avoided for geometry you will be moving. For more info see Colliders below. |
| **Swap UVs** | Use this if lightmapped objects pick up the wrong UV channels. This will swap your primary and secondary UV channels. |
| **Generate Lightmap** | Use this to create the second UV channel to be used for Lightmapping. |
| **Advanced Options** | See Lightmapping UVs document. |
| ***Normals & Tangents*** | |
| **Normals** | Defines if and how normals should be calculated. This is useful for optimizing game size. |
| **Import** | Default option. Imports normals from the file. |

| | |
|---|---|
| **Calculate** | Calculates normals based on **Smoothing angle**. If selected, the **Smoothing Angle** becomes enabled. |
| **None** | Disables normals. Use this option if the mesh is neither normal mapped nor affected by realtime lighting. |
| **Tangents** | Defines if and how tangents and binormals should be calculated. This is useful for optimizing game size. |
| **Import** | Imports tangents and binormals from the file. This option is available only for FBX, Maya and 3dsMax files and only when normals are imported from the file. |
| **Calculate** | Default option. Calculates tangents and binormals. This option is available only when normals are either imported or calculated. |
| **None** | Disables tangents and binormals. The mesh will have no Tangents, so won't work with normal-mapped shaders. |
| **Smoothing Angle** | Sets how sharp an edge has to be in order to be treated as a hard edge. It is also used to split normal map tangents. |
| **Split Tangents** | Enable this if normal map lighting is broken by seams on your mesh. This usually only applies to characters. |
| ***Materials*** | |
| **Import Materials** | Disable this if you don't want materials to be generated. Default-Diffuse material will be used instead. |
| **Material Naming** | Controls how Unity materials are named: |
| **By Base Texture Name** | The name of the diffuse texture of the imported material that will be used to name the material in Unity. When a diffuse texture is not assigned to the material, Unity will use the name of the imported material. |
| **From Model's Material** | The name of the imported material will be used for naming the Unity material. |
| **Model Name + Model's Material** | The name of the model file in combination with the name of the imported material will be used for naming the Unity material. |
| **Texture Name or Model Name + Model's Material (Obsolete)** | The name of the diffuse texture of the imported material will be used for naming the Unity material. When a diffuse texture is not assigned or it cannot be located in one of the Textures folders, then the material will be named by Model Name + Model's Material instead. This option is backwards compatible with the behavior of Unity 3.4 (and earlier versions). We recommend using **By Base texture Name**, because it is less complicated and has more consistent behavior. |
| **Material Search** | Controls where Unity will try to locate existing materials using the name defined by the **Material Naming** option: |
| **Local** | Unity will try to find existing materials only in the "local" Materials folder, ie, the Materials subfolder which is the same folder as the model file. |
| **Recursive-Up** | Unity will try to find existing materials in all Materials subfolders in all parent folders up to the Assets folder. |
| **Everywhere** | Unity will try to find existing materials in all Unity project folders. |

Page last updated: 2012-12-03

---

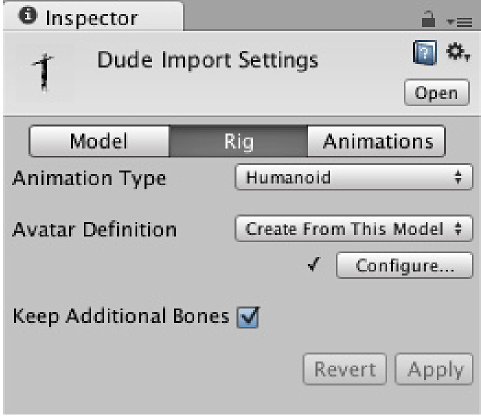
## FBXImporter-Rig

The Rig TAB allows you to assign or create an avatar definition to your imported skinned model so that you can animate it - see Asset Preparation and Import

If you have a **humanoid character** e.g. a biped (two legs) that has two arms and a head, then choose Humanoid and 'Create from this model' an Avatar will be created to best match the bone hierarchy - see Avatar Creation and Setup or you can pick an alternative avatar Definition that has already been set up.

If you have a non humanoid character e.g. a quadruped, or any animateable entity that you wish to use with Mecanim choose **Generic** after choosing you will then need to identify a bone in the drop down to choose as the root node.

Choose legacy if you wish to use the legacy animation system and import and use animations as with 3.x

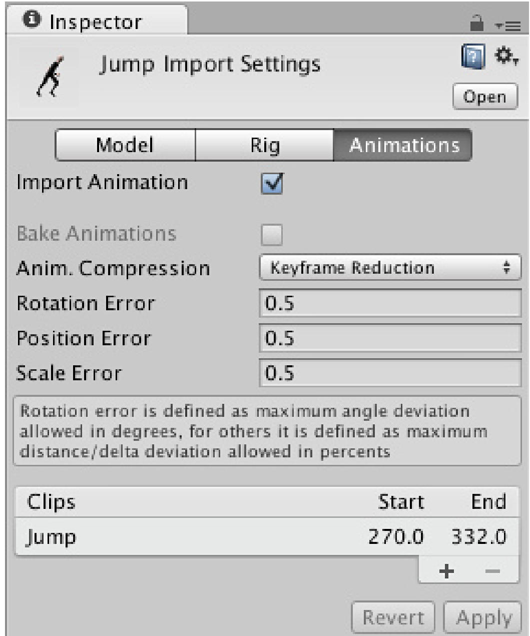



| | |
|---|---|
| **Animation Type** | The type of animation. |
| **None** | No animation present |
| **Legacy** | Legacy animation system |
| **Generic** | Generic Mecanim animation |
| **Humanoid** | Humanoid Mecanim animation system |
| **Avatar Definition** | Where to get the Avatar definition |
| **Create from this model** | The Avatar should be based on this model |
| **Copy from other Avatar** | Point to an Avatar config set up on another model. |
| **Configure...** | Go to the Avatar configuration |
| **Keep additional bones** | |

Page last updated: 2012-11-02

---

# FBXImporter-Animations




***Animations***

| | |
|---|---|
| **Generation** | Controls how animations are imported: |
| **Don't Import** | No animation or skinning is imported. |
| **Store in Original Roots** | Animations are stored in the root objects of your animation package (these might be different from the root objects in Unity). |
| **Store in Nodes** | Animations are stored together with the objects they animate. Use this when you have a complex animation setup and want full scripting control. |

| | |
|---|---|
| **Store in Root** | Animations are stored in the scene's transform root objects. Use this when animating anything that has a hierarchy. |
| **Bake Animations** | Enable this when using IK or simulation in your animation package. Unity will convert to forward kinematics on import. This option is available only for Maya, 3dsMax and Cinema4D files. |
| **Animation Wrap mode** | The default **Wrap Mode** for the animation in the mesh being imported |
| **Default** | The animation plays as specified in the animation splitting options below. |
| **Once** | The animation plays through to the end once and then stops. |
| **Loop** | The animation plays through and then restarts when the end is reached. |
| **PingPong** | The animation plays through and then plays in reverse from the end to the start, and so on. |
| **ClampForever** | The animation plays through but the last frame is repeated indefinitely. This is not the same as Once mode because playback does not technically stop at the last frame (which is useful when blending animations). |
| **Split Animations** | If you have multiple animations in a single file, you can split it into multiple clips. |
| **Name** | The name of the split animation clip |
| **Start** | The first frame of this clip in the model file |
| **End** | The last frame of this clip in the model file |
| **WrapMode** | What the split clip does when the end of the animation is reached (this is identical to the wrap mode option described above). |
| **Loop** | Depending on how the animation was created, one extra frame of animation may be required for the split clip to loop properly. If your looping animation doesn't look correct, try enabling this option. |
| ***Animation Compression*** | |
| **Anim. Compression** | The type of compression that will be applied to this mesh's animation(s) |
| **Off** | Disables animation compression. This means that Unity doesn't reduce keyframe count on import, which leads to the highest precision animations, but slower performance and bigger file and runtime memory size. It is generally not advisable to use this option - if you need higher precision animation, you should enable keyframe reduction and lower allowed **Animation Compression Error** values instead. |
| **Keyframe Reduction** | Reduces keyframes on import. If selected, the **Animation Compression Errors** options are displayed. |
| **Keyframe Reduction and Compression** | Reduces keyframes on import and compresses keyframes when storing animations in files. This affects only file size - the runtime memory size is the same as **Keyframe Reduction**. If selected, the **Animation Compression Errors** options are displayed. |
| **Animation Compression Errors** | These options are available only when keyframe reduction is enabled. |
| **Rotation Error** | Defines how much rotation curves should be reduced. The smaller value you use - the higher precision you get. |
| **Position Error** | Defines how much position curves should be reduced. The smaller value you use - the higher precision you get. |
| **Scale Error** | Defines how much scale curves should be reduced. The smaller value you use - the higher precision you get. |

For properties of AnimationClip, go to the AnimationClip reference page

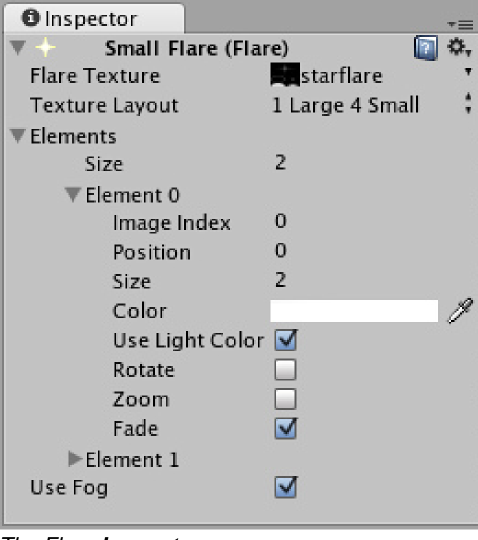
Page last updated: 2012-11-08

---

## class-Flare

**Flare** objects are the source assets that are used by Lens Flare Components. The Flare itself is a combination of a texture file and specific information that determines how the Flare behaves. Then when you want to use the Flare in a **Scene**, you reference the specific Flare from inside a **LensFlare Component** attached to a **GameObject**.

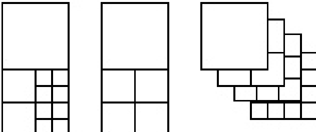
There are some sample Flares in the Standard Assets package. If you want to add one of these to your scene, attach a Lens Flare Component to a GameObject, and drag the Flare you want to use into the **Flare** property of the Lens Flare, just like assigning a **Material** to a **Mesh Renderer**.




*The Flare* ***Inspector***

Flares work by containing several Flare **Elements** on a single **Texture**. Within the Flare, you pick & choose which **Elements** you want to include from any of the Textures.

## Properties

| | |
|---|---|
| **Elements** | The number of Flare images included in the Flare. |
| **Image Index** | Which Flare image to use from the **Flare Texture** for this Element. See the Flare Textures section below for more information. |
| **Position** | The Element's offset along a line running from the containing GameObject's position through the screen center. 0 = GameObject position, 1 = screen center. |
| **Size** | The size of the element. |
| **Color** | Color tint of the element. |
| **Use Light Color** | If the Flare is attached to a Light, enabling this will tint the Flare with the Light's color. |
| **Rotate** | If enabled, bottom of the Element will always face the center of the screen, making the Element spin as the Lens Flare moves around on the screen. |
| **Zoom** | If enabled, the Element will scale up when it becomes visible and scale down again when it isn't. |
| **Fade** | If enabled, the Element will fade in to full strength when it becomes visible and fade out when it isn't. |
| **Flare Texture** | A texture containing images used by this Flare's **Elements**. It must be arranged according to one of the **TextureLayout** options. |
| **Texture Layout** | How the individual Flare Element images are laid out inside the **Flare Texture**.  |
| **Use Fog** | If enabled, the Flare will fade away with distance fog. This is used commonly for small Flares. |

## Details

A Flare consists of multiple **Elements**, arranged along a line. The line is calculated by comparing the position of the GameObject containing the Lens Flare to the center of the screen. The line extends beyond the containing GameObject and the screen center. All Flare **Elements** are strung out on this line.

## Flare Textures

For performance reasons, all **Elements** of one Flare must share the same Texture. This Texture contains a collection of the different images that are available as Elements in a single Flare. The **Texture Layout** defines how the **Elements** are laid out in the **Flare Texture**.

### Texture Layouts

These are the options you have for different Flare **Texture Layouts**. The numbers in the images correspond to the **Image Index** property for each **Element**.

| | |
|---|---|
| **1 Large 4 Small** | Designed for large sun-style Flares where you need one of the **Elements** to have a higher fidelity than the others. This is designed to be used with Textures that are twice as high as they are wide. |
| **1 Large 2 Medium 8 Small** | Designed for complex flares that require 1 high-definition, 2 medium & 8 small images. This is used in the standard assets "50mm Zoom Flare" where the two medium Elements are the rainbow-colored circles. This is designed to be used with textures that are twice as high as they are wide. |
| **1 Texture** | A single image. |
| **2x2 grid** | A simple 2x2 grid. |
| **3x3 grid** | A simple 3x3 grid. |
| **4x4 grid** | A simple 4x4 grid. |

### Hints

- If you use many different Flares, using a single **Flare Texture** that contains all the **Elements** will give you best rendering performance.
- Lens Flares are blocked by **Colliders**. A Collider in-between the Flare GameObject and the Camera will hide the Flare, even if the Collider does not have a **Mesh Renderer**.

Page last updated: 2007-09-10

---

## class-Font

**Fonts** can be created or imported for use in either the GUI Text or the Text Mesh **Components**.

### Importing Font files

To add a font to your project you need to place the font file in your Assets folder. Unity will then automatically import it. Supported Font formats are TrueType Fonts (.ttf or .dfont files) and OpenType Fonts (.otf files).

To change the **Size** of the font, highlight it in the **Project View** and you have a number of options in the **Import Settings** in the **Inspector**.

*Import Settings for a font*

| | |
|---|---|
| **Font Size** | The size of the font, based on the sizes set in any word processor |
| **Character** | The text encoding of the font. You can force the font to display only upper- or lower-case characters here<br>Setting this mode to Dynamic causes Unity to use the underlying OS font rendering routines (see below). |
| **2.x font placing** | Unity 3.x uses a more typographically correct vertical font placement compared to 2.x. We now use the font ascent stored in the truetype font data rather than computing it when we render the font texture. Ticking this Property causes the 2.x vertical positioning to be used. |

*Import Settings specific to non-dynamic fonts*

| | |
|---|---|
| **Font Rendering** | The amount of anti-aliasing applied to the font. |

*Import Settings specific to dynamic fonts*

| | |
|---|---|
| **Style** | The styling applied to the font, one of Normal, Bold, Italic or BoldAndItalic. |
| **Include Font Data** | This setting controls the packaging of the font when used with Dynamic font property. When selected the TTF is included in the output of the build. When not selected it is assumed that the end user will have the font already installed on their machine. Note that fonts are subject to copyright and you should only include fonts that you have licensed or created for yourself. |
| **Font Names** | Only available when Include Font Data is not selected. Enter a comma-separated list of font names. These fonts will be tried in turn from left to right and the first one found on the gamers machine will be used. |

After you import the font, you can expand the font in Project View to see that it has auto-generated some assets. Two assets are created during import: "font material" and "font texture".

## Dynamic fonts

Unity 3.0 adds support for dynamic font rendering. When you set the **Characters** drop-down in the Import Settings to **Dynamic**, Unity will not pre-generate a texture with all font characters. Instead, it will use the FreeType font rendering engine to create the texture on the fly. This has the advantage that it can save in download size and texture memory, especially when you are using a font which is commonly included in user systems, so you don't have to include the font data, or when you need to support asian languages or large font sizes (which would make the font textures very large using normal font textures).

## Unicode support

Unity has full unicode support. Unicode text allows you to display German, French, Danish or Japanese characters that are usually not supported in an ASCII character set. You can also enter a lot of different special purpose characters like arrow signs or the option key sign, if your font supports it.

To use unicode characters, choose either **Unicode** or **Dynamic** from the **Characters** drop-down in the Import Settings. You can now display unicode characters with this font. If you are using a **GUIText** or **Text Mesh**, you can enter unicode characters into the Component's **Text** field in the Inspector. Note that the Inspector on Mac may not show the unicode characters correctly.

You can also use unicode characters if you want to set the displayed text from scripting. The Javascript and C# compilers fully support Unicode based scripts. You simply have to save your scripts with UTF-16 encoding. In **Unitron**, this can be done by

opening the script and choosing **Text->Text Encoding->Unicode (UTF 16)**. Now you can add unicode characters to a string in your script and they will display as expected in **UnityGUI**, a GUIText, or a Text Mesh. On the PC where **UniSciTE** is used for script editing save scripts using the UCS-2 Little Endian encoding.

### Changing Font Color

There are different ways to change the color of your displayed font, depending on how the font is used.

#### GUIText & Text Mesh

If you are using a GUIText or a Text Mesh, you can change its color by using a custom **Material** for the font. In the Project View, click on **Create->Material**, and select and set up the newly created Material in the **Inspector**. Make sure you assign the texture from the font asset to the material. If you use the built-in **GUI/Text Shader** shader for the font material, you can choose the color in the **Text Color** property of the material.

#### UnityGUI

If you are using UnityGUI scripting to display your font, you have much more control over the font's color under different circumstances. To change the font's color, you create a **GUISkin** from **Assets->Create->GUI Skin**, and define the color for the specific control state, e.g. **Label->Normal->Text Color**. For more details, please read the GUI Skin page.

### Hints

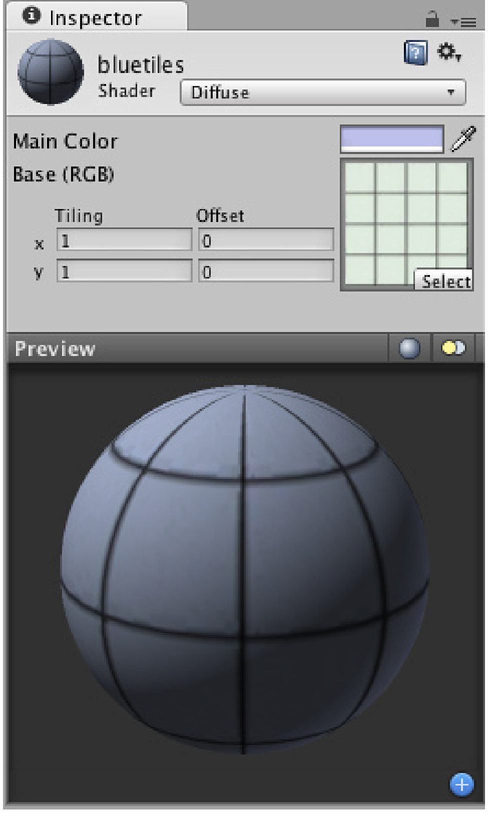
- To display an imported font select the font and choose **GameObject->Create Other->3D Text**.
- Using only lower or upper case characters reduces generated texture size.
- The default font that Unity supplies is Arial. This font is always available and does not appear in the Project window.

Page last updated: 2012-10-08

---

## class-Material

Materials are used in conjunction with Mesh or Particle Renderers attached to the **GameObject**. They play an essential part in defining how your object is displayed. Materials include a reference to the **Shader** used to render the **Mesh** or **Particles**, so these Components can not be displayed without some kind of Material.




*A Diffuse Shader Material has only two properties - a color and a texture.*

## Properties

The properties of any Material will change depending on the selected **Shader**. These are the most often used properties:

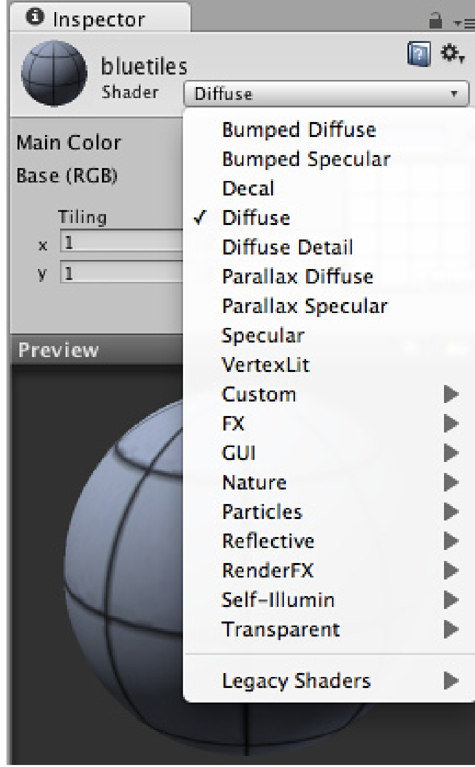
| | |
|---|---|
| **Shader** | The Shader that will be used by the Material. For more information, read the Built-in Shader Guide. |
| **Main Color** | Any kind of color tint can be applied. Use white for no tint. |
| **Base** | The **Texture** that will be displayed. |

## Details

Materials are used to place Textures onto your GameObjects. You cannot add a Texture directly without a Material, and doing so will implicitly create a new Material. The proper workflow is to create a Material, select a Shader, and choose the Texture asset(s) to display along with it. For more information on Materials, take a look at the Manual's page about Materials.

### Choosing Shaders

After you create your material, the first thing you should decide is which Shader to use. You choose it from the drop-down **Shader** menu.

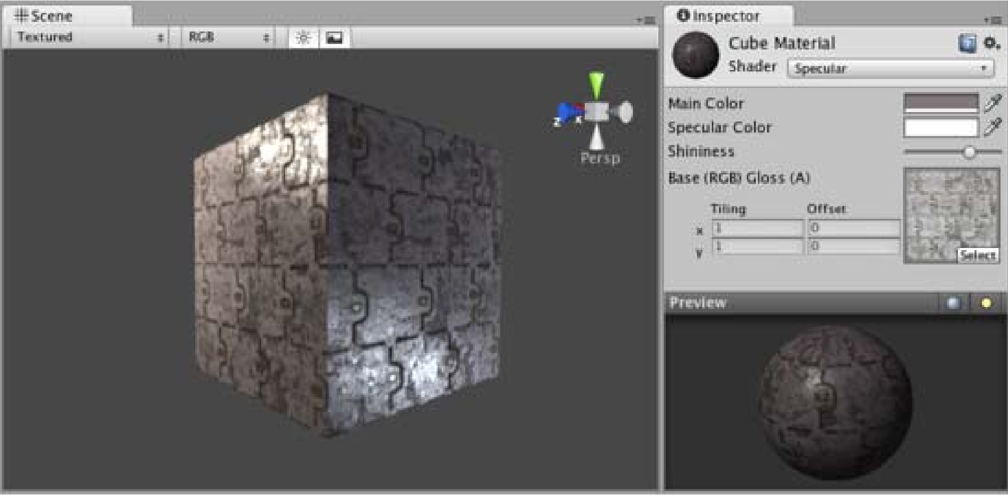


*The **Shader** drop-down menu*

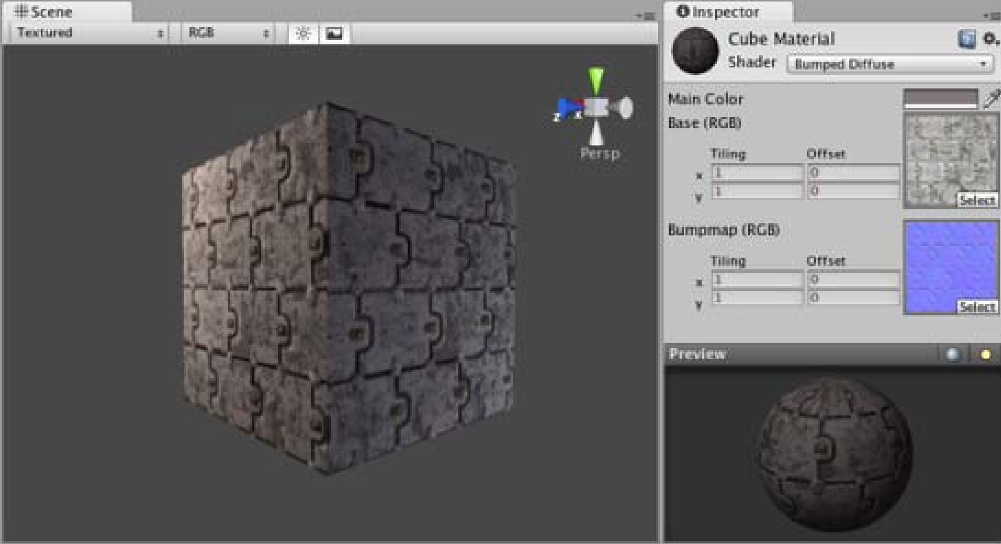
You can choose any Shader that exists in your project's assets folder or one of the built-in Shaders. You can also create your own Shaders. For more information on using the built-in Shaders, view the Built-in Shader Guide. For information on writing your own shaders, take a look at the Shaders section of the Manual and ShaderLab Reference.

### Setting shader properties

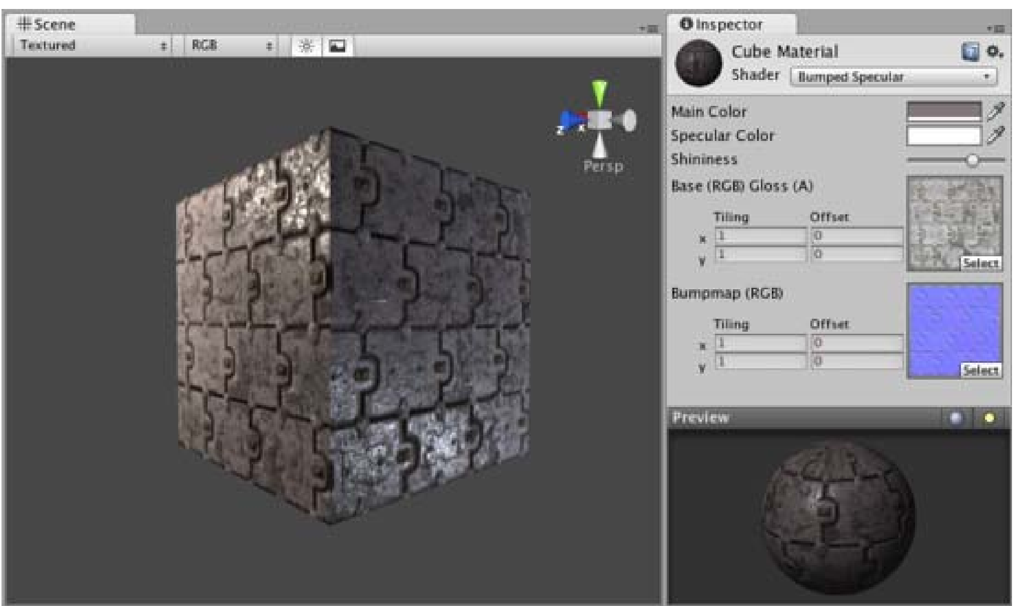
Depending on the type of shader selected, a number of different properties can appear in the **Inspector**.




*Properties of a Specular shader*




*Properties of a Normal mapped shader*




*Properties of a Normal mapped Specular shader*

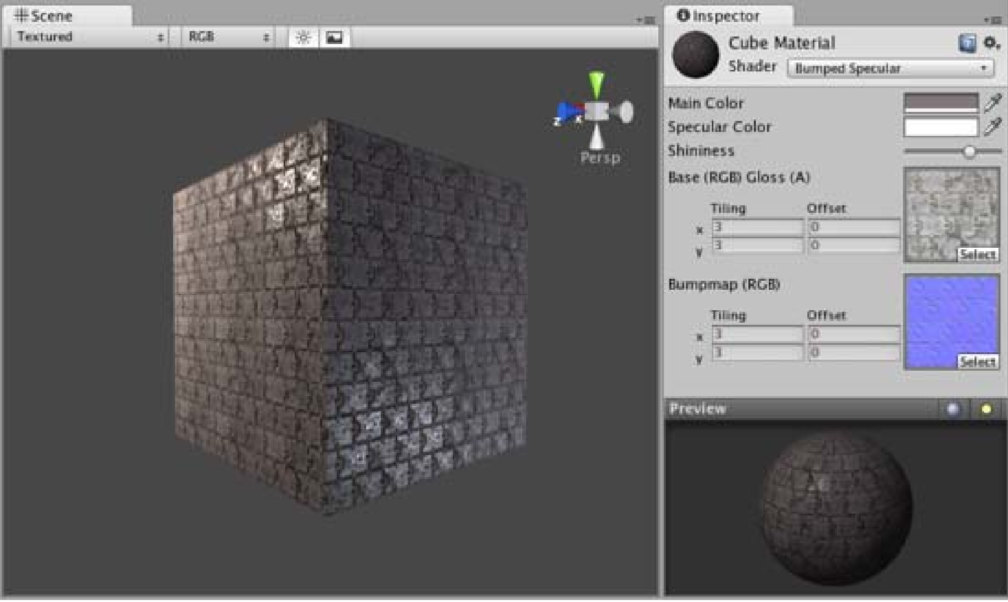
The different types of Shader properties are:

**Color pickers** Used to choose colors.

| | |
|---|---|
| **Sliders** | Used to tweak a number value across the allowed range. |
| **Textures** | Used to select textures. |

**Texture placement**

The placement of the textures can be altered by changing the **Offset** and **Tiling** properties.




*This texture is tiled 2x2 times by changing the **Tiling** properties*

| | |
|---|---|
| **Offset** | Slides the Texture around. |
| **Tiling** | Tiles the Texture along the different axes. |

### Hints

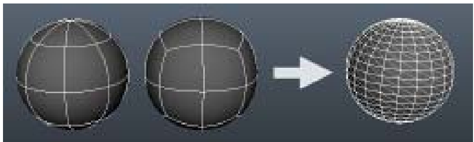
- It is a good practice to share a single Material across as many GameObjects as possible. This has great performance benefits.

Page last updated: 2010-07-13

---

## class-Mesh

**Meshes** make up a large part of your 3D worlds. Aside from some Asset store plugins, Unity does not include modelling tools. Unity does however have great interactivity with most 3D modelling packages. Unity supports triangulated or Quadrangulated polygon meshes. Nurbs, Nurms, Subdiv surfaces must be converted to polygons.



## 3D formats

Importing meshes into Unity can be achieved from two main types of files:

1. **Exported 3D file formats**, such as .FBX or .OBJ
2. **Proprietary 3D application files**, such as .Max and .Blend file formats from 3D Studio Max or Blender for example.

Either should enable you to get your meshes into Unity, but there are considerations as to which type you choose:

### Exported 3D files

Unity can read .FBX, **.dae** (Collada), **.3DS**, **.dxf** and **.obj** files, FBX exporters can be found here and obj or Collada exporters can also be found for many applications

**Advantages:**

- Only export the data you need
- Verifiable data (re-import into 3D package before Unity)
- Generally smaller files
- Encourages modular approach - e.g different components for collision types or interactivity
- Supports other 3D packages whose Proprietary formats we don't have direct support for

**Disadvantages:**

- Can be a slower pipeline for prototyping and iterations
- Easier to lose track of versions between source(working file) and game data (exported FBX for example)

## Proprietary 3D application files

Unity can also import, *through conversion*: **Max**, **Maya**, **Blender**, **Cinema4D**, **Modo**, **Lightwave** & **Cheetah3D** files, e.g. **.MAX**, **.MB**, **.MA** etc.

**Advantages:**

- Quick iteration process (save the source file and Unity reimports)
- Simple initially

**Disadvantages:**

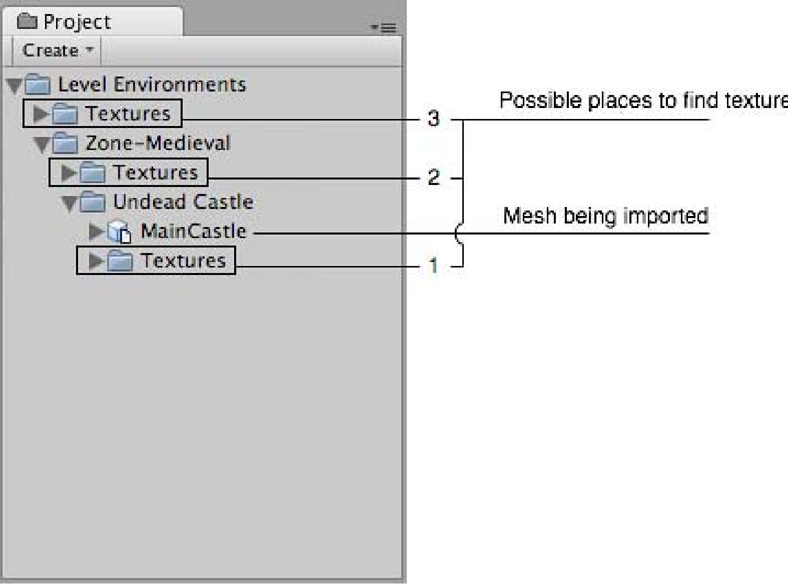
- A licensed copy of that software must be installed on all machines using the Unity project
- Files can become bloated with unnecessary data
- Big files can slow Unity updates
- Less validation � harder to troubleshoot problems

Here are some guidelines for directly supported 3D applications, others can most often export file type listed above.

- Maya
- Cinema 4D
- 3ds Max
- Cheetah3D
- Modo
- Lightwave
- Blender

## Textures

Unity will attempt to find the textures used by a mesh automatically on import by following a specific search plan. First, the importer will look for a sub-folder called Textures within the same folder as the mesh or in any parent folder. If this fails, an exhaustive search of all textures in the project will be carried out. Although slightly slower, the main disadvantage of the exhaustive search is that there could be two or more textures in the project with the same name. In this case, it is not guaranteed that the right one will be found.




*Place your textures in a **Textures** folder at or above the asset's level*

FBX importer options for the model

### Material Generation and Assignment

For each imported material Unity will apply the following rules:-

If material generation is disabled (i.e. **Import Materials** is unchecked), then it will assign the Default-Diffuse material. If it is enabled then it will do the following:

- Unity will pick a name for the Unity material based on the **Material Naming** setting
- Unity will try to find an existing material with that name. The scope of the Material search is defined by the **Material Search** setting.
- If Unity succeeds in finding an existing material then it will use it for the imported scene, otherwise it will generate a new material

### Colliders

Unity uses two main types of colliders: **Mesh Colliders** and **Primitive Colliders**. Mesh colliders are components that use imported mesh data and can be used for environment collision. When you enable **Generate Colliders** in the Import Settings, a Mesh collider is automatically added when the mesh is added to the Scene. It will be considered solid as far as the physics system is concerned.

If you are moving the object around (a car for example), you can not use Mesh colliders. Instead, you will have to use Primitive colliders. In this case you should disable the **Generate Colliders** setting.

### Animations

Animations are automatically imported from the scene. For more details about animation import options see the section on asset preparation and import in the Mecanim animation system.

### Normal mapping and characters

If you have a character with a normal map that was generated from a high-polygon version of the model, you should import the game-quality version with a **Smoothing angle** of 180 degrees. This will prevent odd-looking seams in lighting due to tangent splitting. If the seams are still present with these settings, enable **Split tangents across UV seams**.

If you are converting a greyscale image into a normal map, you don't need to worry about this.

### Blendshapes

Unity has support for BlendShapes (also called morph-targets or vertex level animation). Unity can import BlendShapes from **.FBX** (BlendShapes and controlling aninimation) and **.dae** (only BlendShapes) exported 3D files. Unity BlendShapes support vertex level animation on vertices, normals and tangents. Mesh can be affected by skin and BlendShapes at the same time. All meshes imported with BlendShapes will use SkinnedMeshRenderer (no mater if it does have skin or not). BlendShape animation is imported as part of regular animation - it simply animates BlendShape weigts on SkinnedMeshRenderer.

There are two ways to import BlendShapes with normals:

1. Set **Normals** import mode to **Calculate**, this way same logic will be used for calculating normals on a mesh and BlendShapes.
2. Export smoothing groups information to the source file, this ways Unity will calculate normals from smoothing groups for mesh and BlendShapes.

If you want tangents on your BlendShapes then set **Tangents** import mode to **Calculate**.

### Hints

- Merge your meshes together as much as possible. Make them share materials and textures. This has a huge performance benefit.
- If you need to set up your objects further in Unity (adding physics, scripts or other coolness), save yourself a world of pain and name your objects properly in your 3D application. Working with lots of *pCube17* or *Box42*-like objects is not fun.
- Make your meshes be centered on the world origin in your 3D app. This will make them easier to place in Unity.
- If a mesh does not have vertex colors, Unity will automatically add an array of all-white vertex colors to the mesh the first time it is rendered.

**The Unity Editor shows too many vertices or triangles (compared to what my 3D app says)**
This is correct. What you are looking at is the number of vertices/triangles actually being sent to the GPU for rendering. In addition to the case where the material requires them to be sent twice, other things like hard-normals and non-contiguous UVs increase vertex/triangle counts significantly compared to what a modeling app tells you. Triangles need to be contiguous in both 3D and UV space to form a strip, so when you have UV seams, degenerate triangles have to be made to form strips - this bumps up the count.

### See Also

- Modeling Optimized Characters
- How do I use normal maps?
- How do I fix the rotation of an imported model?

Page last updated: 2012-12-03

---

## class-MovieTexture

**Note:** This is a **Pro/Advanced** feature only.

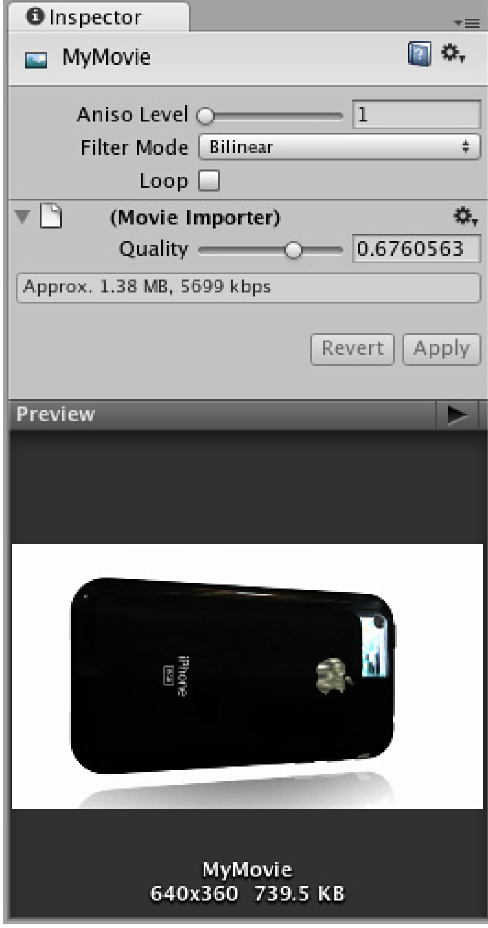
## Desktop

**Movie Textures** are animated **Textures** that are created from a video file. By placing a video file in your project's **Assets Folder**, you can import the video to be used exactly as you would use a regular Texture.

Video files are imported via Apple QuickTime. Supported file types are what your QuickTime installation can play (usually **.mov**, **.mpg**, **.mpeg**, **.mp4**, **.avi**, **.asf**). On Windows movie importing requires Quicktime to be installed (download here).

### Properties

The Movie Texture **Inspector** is very similar to the regular Texture Inspector.




*Video files are Movie Textures in Unity*

| | |
|---|---|
| **Aniso Level** | Increases Texture quality when viewing the texture at a steep angle. Good for floor and ground textures |
| **Filtering Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations |
| **Loop** | If enabled, the movie will loop when it finishes playing |
| **Quality** | Compression of the Ogg Theora video file. A higher value means higher quality, but larger file size |

## Details

When a video file is added to your Project, it will automatically be imported and converted to **Ogg Theora** format. Once your Movie Texture has been imported, you can attach it to any **GameObject** or **Material**, just like a regular Texture.

### Playing the Movie

Your Movie Texture will not play automatically when the game begins running. You must use a short script to tell it when to play.

```
// this line of code will make the Movie Texture begin playing
renderer.material.mainTexture.Play();
```

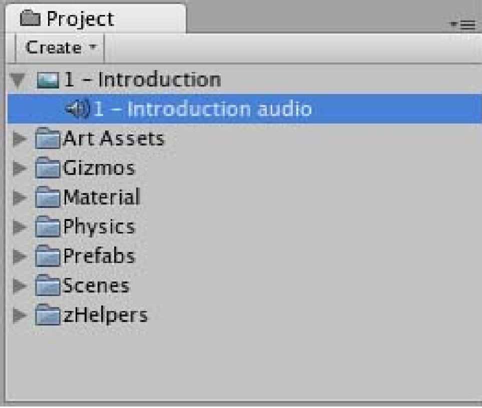
Attach the following script to toggle Movie playback when the space bar is pressed:

```
function Update () {
	if (Input.GetButtonDown ("Jump")) {
		if (renderer.material.mainTexture.isPlaying) {
			renderer.material.mainTexture.Pause();
		}
		else {
			renderer.material.mainTexture.Play();
		}
	}
```

```
}
```

For more information about playing Movie Textures, see the Movie Texture Script Reference page

**Movie Audio**
When a Movie Texture is imported, the audio track accompanying the visuals are imported as well. This audio appears as an **AudioClip** child of the Movie Texture.




*The video's audio track appears as a child of the Movie Texture in the **Project View***

To play this audio, the Audio Clip must be attached to a GameObject, like any other Audio Clip. Drag the Audio Clip from the Project View onto any GameObject in the Scene or Hierarchy View. Usually, this will be the same GameObject that is showing the Movie. Then use audio.Play() to make the the movie's audio track play along with its video.

## iOS

Movie Textures are not supported on iOS. Instead, full-screen streaming playback is provided using Handheld.PlayFullScreenMovie.

**You need to keep your videos inside the *StreamingAssets* folder located in your Project directory.**

Unity iOS supports any movie file types that play correctly on an iOS device, implying files with the extensions **.mov**, **.mp4**, **.mpv**, and **.3gp** and using one of the following compression standards:

- H.264 Baseline Profile Level 3.0 video
- MPEG-4 Part 2 video

For more information about supported compression standards, consult the iPhone SDK MPMoviePlayerController Class Reference.

As soon as you call iPhoneUtils.PlayMovie or iPhoneUtils.PlayMovieURL, the screen will fade from your current content to the designated background color. It might take some time before the movie is ready to play but in the meantime, the player will continue displaying the background color and may also display a progress indicator to let the user know the movie is loading. When playback finishes, the screen will fade back to your content.

**The video player does not respect switching to mute while playing videos**
As written above, video files are played using Apple's embedded player (as of SDK 3.2 and iPhone OS 3.1.2 and earlier). This contains a bug that prevents Unity switching to mute.

**The video player does not respect the device's orientation**
The Apple video player and iPhone SDK do not provide a way to adjust the orientation of the video. A common approach is to manually create two copies of each movie in landscape and portrait orientations. Then, the orientation of the device can be determined before playback so the right version of the movie can be chosen.

## Android

Movie Textures are not supported on Android. Instead, full-screen streaming playback is provided using Handheld.PlayFullScreenMovie.

**You need to keep your videos inside of the *StreamingAssets* folder located in your Project directory.**

Unity Android supports any movie file type supported by Android, (ie, files with the extensions **.mp4** and **.3gp**) and using one of the following compression standards:

- H.263
- H.264 AVC
- MPEG-4 SP

However, device vendors are keen on expanding this list, so some Android devices are able to play formats other than those listed, such as HD videos.

For more information about the supported compression standards, consult the Android SDK Core Media Formats documentation.

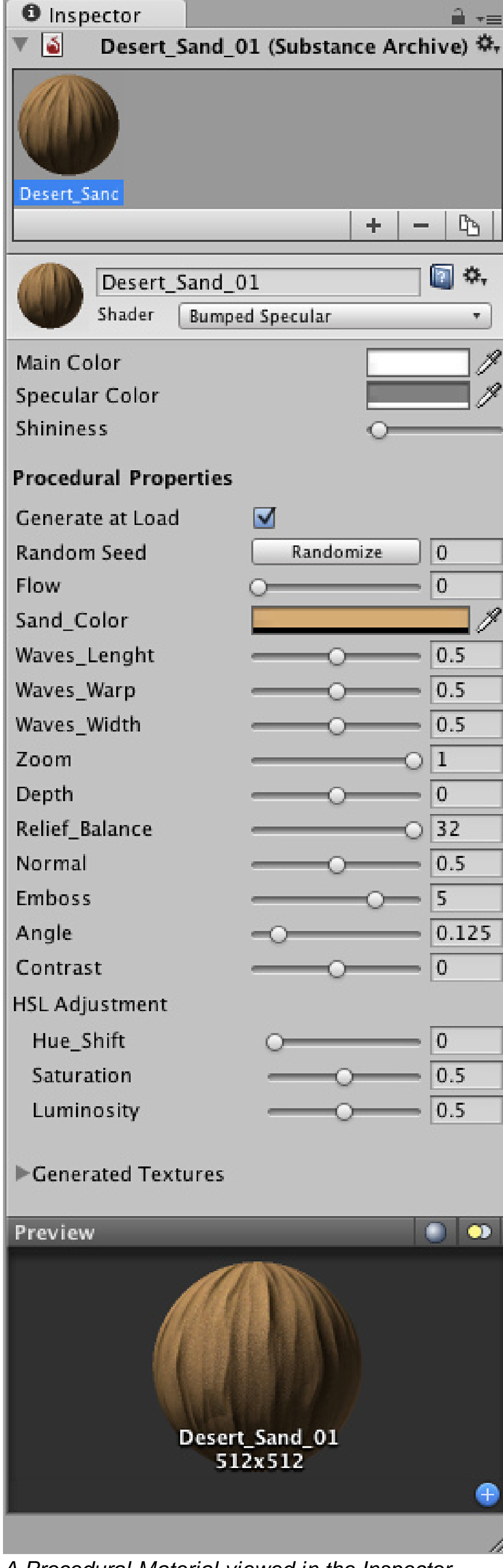
As soon as you call iPhoneUtils.PlayMovie or iPhoneUtils.PlayMovieURL, the screen will fade from your current content to the designated background color. It might take some time before the movie is ready to play. In the meantime, the player will continue displaying the background color and may also display a progress indicator to let the user know the movie is loading. When playback finishes, the screen will fade back to your content.

Page last updated: 2012-09-18

---

# class-ProceduralMaterial

Procedural Material Assets are textures that are generated for you at run-time. See Procedural Materials in the User Guide for more information. A Procedural Material asset can contain one or more procedural materials. These can be viewed in the Inspector just like regular materials. Note however that often procedural materials have many tweakable parameters. As with Material assets the Inspector shows a preview of the Procedural Material at the bottom of the window.




*A Procedural Material viewed in the Inspector.*

The Inspector window has 4 main panes.

1. Substance Archive Manager.
2. Properties.
3. Generated Textures.
4. Preview.

## Substance Archive Manager

The archive view shows you all the procedural materials that the Procedural Material asset contains. Select the procedural

material you are interested in from the row of previews. Procedural Materials can be Added and Deleted to the Procedural Material Asset archive using the plus and minus buttons. Adding a procedural material will create a new material using the prototype encoded in the archive. The third, Duplicate button will create a new procedural material that is a copy of the currently selected one, including all its settings. Procedural materials can be renamed by typing in a new name in the material header name field.

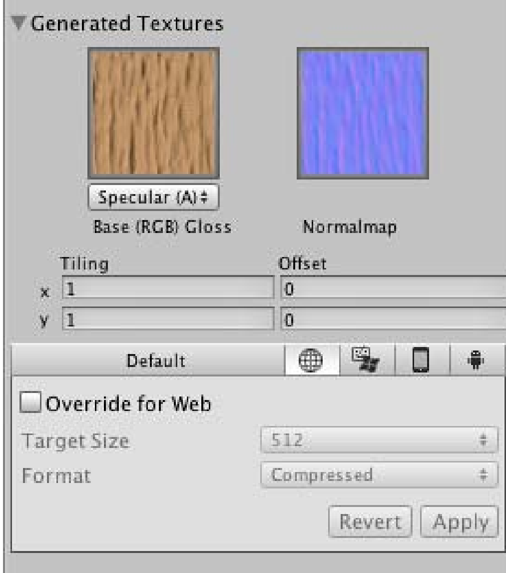
## Properties

**Material Properties**
These are the regular properties of the material, which are dependent on which shader is chosen. They work the same as for regular materials.

**Procedural Properties**
The properties of any Procedural Material will change according to how the procedural material was created.

| | |
|---|---|
| **Generate at Load** | Generate the substance when the scene loads. If disabled, it will only be generated when prompted from scripting. |
| **Random Seed** | Procedural materials often need some randomness. The Random Seed can be used to vary the generated appearance. Often this will be zero. Just click the Randomize button to get a different seed and observe how the material changes. |

## Generated Textures




*The Generated Textures pane.*

This area allows you to visualize textures that the procedural material generates. The dropdown below each of the generated textures allows you to choose which texture map should supply the alpha channel for that texture. You could, for example, specify that the base alpha comes from the Transparency image, or from the Specular image. The screen-shot below shows the base alpha channel coming from the Specular image.

**Per-Platform Overrides**
When you are building for different platforms, you have to think on the resolution of your textures for the target platform, the size and the quality. You can override these options and assign specific values depending on the platform you are deploying to. Note that if you don't select any value to override, the Editor will pick the default values when building your project.

| | |
|---|---|
| **Target Size** | The targeted size of the generated textures. Most procedural textures are designed to be resolution independent and will respect the chosen target size, but in rare cases they will use a fixed size instead, or have the possible sizes limited to within a certain range. The actual size of the generated textures can be read in the preview at the bottom of the Inspector. |
| **Texture Format** | What internal representation is used for the texture in memory, once it is generated. This is a tradeoff between size and quality: |
| **Compressed** | Compressed RGB texture. This will result in considerably less memory used. |
| **RAW** | Uncompressed truecolor, this is the highest quality. At 256 KB for a 256x256 texture. |

### Preview

The procedural material preview operates in an identical manner to the material preview. However, unlike the regular material preview it shows the pixel dimensions of the generated textures.

Page last updated: 2011-07-01

---

## class-RenderTexture

**Render Textures** are special types of **Textures** that are created and updated at runtime. To use them, you first create a new Render Texture and designate one of your Cameras to render into it. Then you can use the Render Texture in a **Material** just like a regular Texture. The Water prefabs in Unity Standard Assets are an example of real-world use of Render Textures for making real-time reflections and refractions.

Render Textures are a Unity Pro feature.

### Properties

The Render Texture **Inspector** is different from most Inspectors, but very similar to the Texture Inspector.

*The Render Texture Inspector is almost identical to the Texture Inspector*

The Render Texture inspector displays the current contents of Render Texture in realtime and can be an invaluable debugging tool for effects that use render textures.

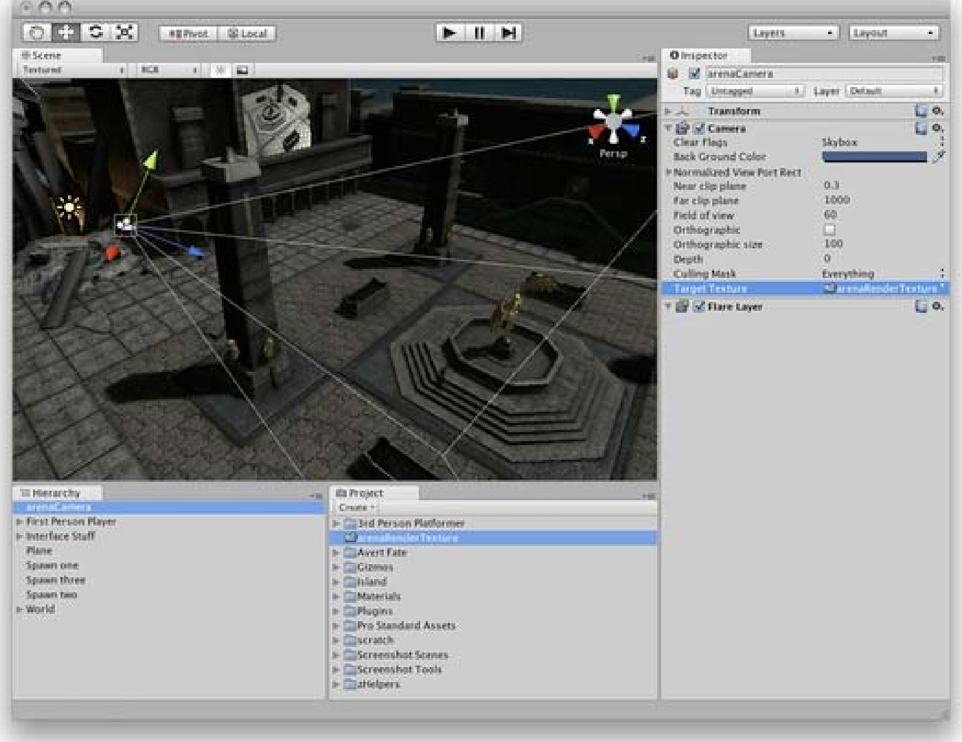
| | |
|---|---|
| **Size** | The size of the Render Texture in pixels. Observe that only power-of-two values sizes can be chosen. |
| **Aniso Level** | Increases Texture quality when viewing the texture at a steep angle. Good for floor and ground textures |
| **Filter Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations: |
| **No Filtering** | The Texture becomes blocky up close |
| **Bilinear** | The Texture becomes blurry up close |
| **Trilinear** | Like Bilinear, but the Texture also blurs between the different mip levels |
| **Wrap Mode** | Selects how the Texture behaves when tiled: |
| **Repeat** | The Texture repeats (tiles) itself |

| **Clamp** | The Texture's edges get stretched |
|---|---|

### Example

A very quick way to make a live arena-camera in your game:

1. Create a new Render Texture asset using **Assets->Create->Render Texture**.
2. Create a new Camera using **GameObject->Create Other->Camera**.
3. Assign the Render Texture to the **Target Texture** of the new Camera.
4. Create a wide, tall and thin box
5. Drag the Render Texture onto it to create a Material that uses the render texture.
6. Enter Play Mode, and observe that the box's texture is updated in real-time based on the new Camera's output.




*Render Textures are set up as demonstrated above*

## Hints

- Unity renders everything in the texture assigned to RenderTexture.active.

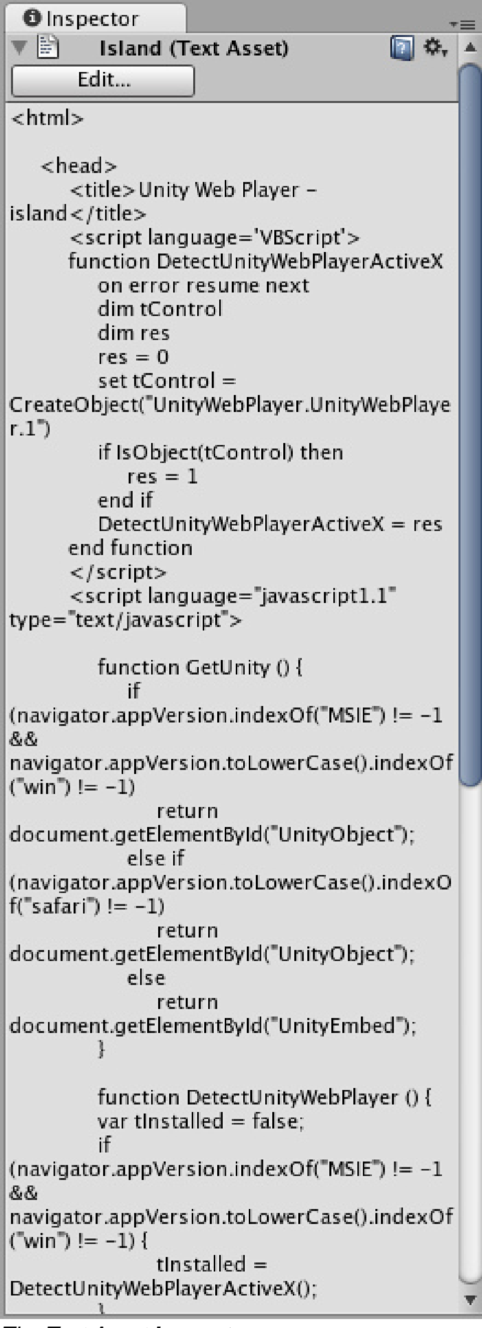
Page last updated: 2011-01-31

---

## class-TextAsset

**Text Assets** are a format for imported text files. When you drop a text file into your Project Folder, it will be converted to a Text Asset. The supported text formats are:

- **.txt**
- **.html**
- **.htm**
- **.xml**
- **.bytes**



*The Text Asset **Inspector***

## Properties

| | |
|---|---|
| **Text** | The full text of the asset as a single string. |

## Details

The Text Asset is a very specialized use case. It is extremely useful for getting text from different text files into your game while you are building it. You can write up a simple .txt file and bring the text into your game very easily. It is not intended for text file generation at runtime. For that you will need to use traditional Input/Output programming techniques to read and write external files.

Consider the following scenario. You are making a traditional text-heavy adventure game. For production simplicity, you want to break up all the text in the game into the different rooms. In this case you would make one text file that contains all the text that will be used in one room. From there it is easy to make a reference to the correct Text Asset for the room you enter. Then with some customized parsing logic, you can manage a large amount of text very easily.

### Binary data

A special feature of the text asset is that it can be used to store binary data. By giving a file the extension **.bytes** it can be loaded as a text asset and the data can be accessed through the **bytes** property.

For example put a jpeg file into the Resources folder and change the extension to **.bytes**, then use the following script code to read the data at runtime:

```
//Load texture from disk
TextAsset bindata= Resources.Load("Texture") as TextAsset;
Texture2D tex = new Texture2D(1,1);
tex.LoadImage(bindata.bytes);
```

### Hints

- Text Assets are serialized like all other assets in a build. There is no physical text file included when you publish your game.
- Text Assets are not intended to be used for text file generation at runtime.

Page last updated: 2011-05-24

---

## class-Texture2D

**Textures** bring your **Meshes**, **Particles**, and interfaces to life! They are image or movie files that you lay over or wrap around your objects. As they are so important, they have a lot of properties. If you are reading this for the first time, jump down to Details, and return to the actual settings when you need a reference.

The shaders you use for your objects put specific requirements on which textures you need, but the basic principle is that you can put any image file inside your project. If it meets the size requirements (specified below), it will get imported and optimized for game use. This extends to multi-layer Photoshop or TIFF files - they are flattened on import, so there is no size penalty for your game.

### Properties

The **Texture Inspector** looks a bit different from most others:

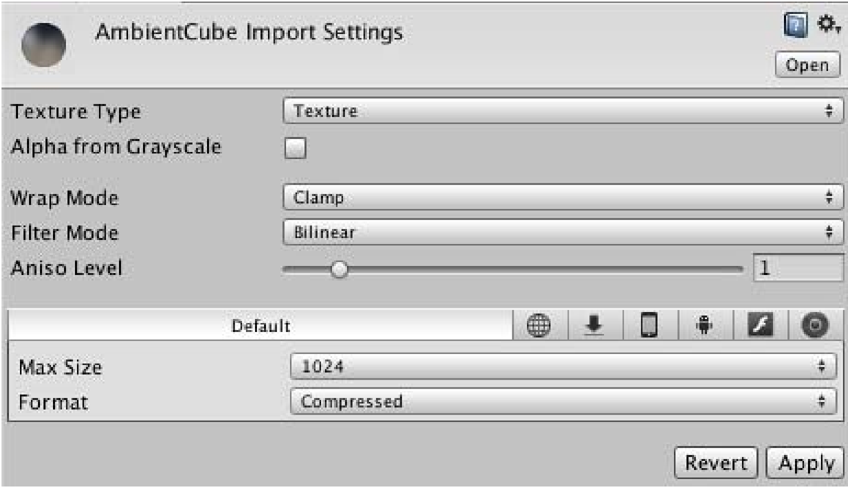
The inspector is split into two sections, the **Texture Importer** and the texture preview.

### Texture Importer

Textures all come from image files in your **Project Folder**. How they are imported is specified by the **Texture Importer**. You change these by selecting the file texture in the **Project View** and modifying the **Texture Importer** in the **Inspector**.

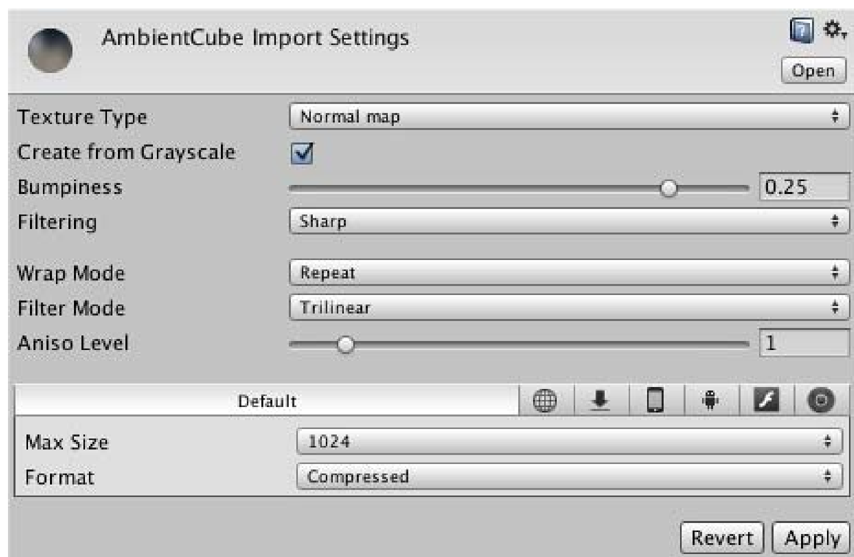
The topmost item in the inspector is the **Texture Type** menu that allows you to select the type of texture you want to create from the source image file.

| | |
|---|---|
| **Texture Type** | Select this to set basic parameters depending on the purpose of your texture. |
| **Texture** | This is the most common setting used for all the textures in general. |
| **Normal Map** | Select this to turn the color channels into a format suitable for real-time normal mapping. For more info, see Normal Maps |
| **GUI** | Use this if your texture is going to be used on any HUD/GUI Controls. |
| **Reflection** | Also known as Cube Maps, used to create reflections on textures. check Cubemap Textures for more info. |
| **Cookie** | This sets up your texture with the basic parameters used for the Cookies of your lights |
| **Advanced** | Select this when you want to have specific parameters on your texture and you want to have total control over your texture. |




*Basic Texture Settings Selected*

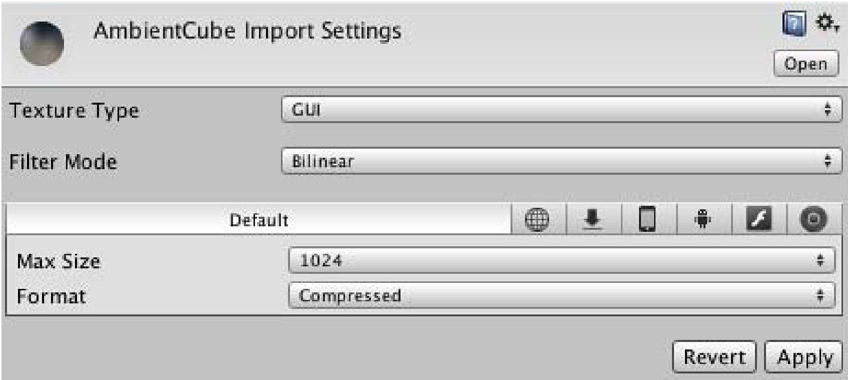
| | |
|---|---|
| **Alpha From Grayscale** | If enabled, an alpha transparency channel will be generated by the image's existing values of light & dark. |
| **Wrap Mode** | Selects how the Texture behaves when tiled: |
| **Repeat** | The Texture repeats (tiles) itself |
| **Clamp** | The Texture's edges get stretched |
| **Filter Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations: |
| **Point** | The Texture becomes blocky up close |
| **Bilinear** | The Texture becomes blurry up close |
| **Trilinear** | Like Bilinear, but the Texture also blurs between the different mip levels |
| **Aniso Level** | Increases texture quality when viewing the texture at a steep angle. Good for floor and ground textures, see below. |




*Normal Map Settings in the Texture Importer*

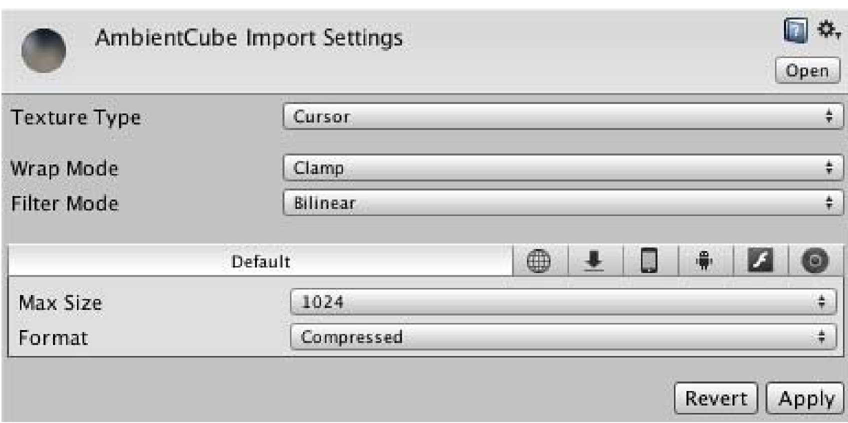
| | |
|---|---|
| **Create from Greyscale** | If this is enabled then Bumpiness and Filtering options will be shown. |
| **Bumpiness** | Control the amount of bumpiness. |
| **Filtering** | Determine how the bumpiness is calculated: |
| **Smooth** | This generates normal maps that are quite smooth. |
| **Sharp** | Also known as a Sobel filter. this generates normal maps that are sharper than Standard. |
| **Wrap Mode** | Selects how the Texture behaves when tiled: |
| **Repeat** | The Texture repeats (tiles) itself |
| **Clamp** | The Texture's edges get stretched |
| **Filter Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations: |
| **Point** | The Texture becomes blocky up close |
| **Bilinear** | The Texture becomes blurry up close |
| **Trilinear** | Like Bilinear, but the Texture also blurs between the different mip levels |

| | |
|---|---|
| **Aniso Level** | Increases texture quality when viewing the texture at a steep angle. Good for floor and ground textures, see below. |



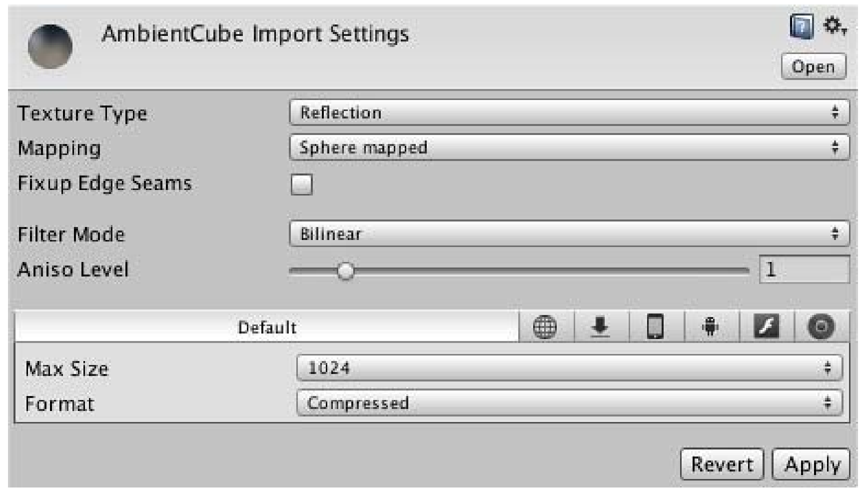

*GUI Settings for the Texture Importer*

| | |
|---|---|
| **Filter Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations: |
| **Point** | The Texture becomes blocky up close |
| **Bilinear** | The Texture becomes blurry up close |
| **Trilinear** | Like Bilinear, but the Texture also blurs between the different mip levels |




*Cursor settings for the Texture Importer*

| | |
|---|---|
| **Wrap Mode** | Selects how the Texture behaves when tiled: |
| **Repeat** | The Texture repeats (tiles) itself |
| **Clamp** | The Texture's edges get stretched |
| **Filter Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations: |
| **Point** | The Texture becomes blocky up close |
| **Bilinear** | The Texture becomes blurry up close |
| **Trilinear** | Like Bilinear, but the Texture also blurs between the different mip levels |

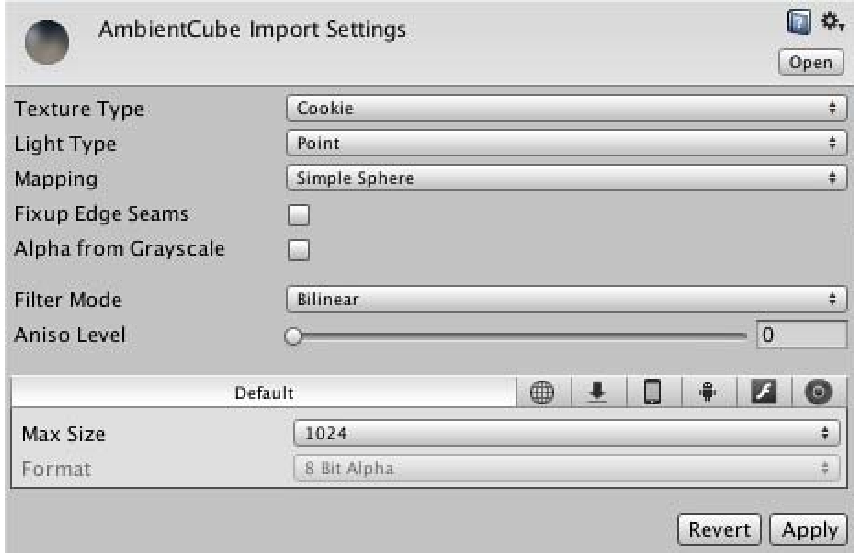



*Reflection Settings in the Texture Importer*

| | |
|---|---|
| **Mapping** | This determines how the texture will be mapped to a cubemap. |

| | |
|---|---|
| **Sphere Mapped** | Maps the texture to a "sphere like" cubemap. |
| **Cylindrical** | Maps the texture to a cylinder, use this when you want to use reflections on objects that are like cylinders. |
| **Simple Sphere** | Maps the texture to a simple sphere, deforming the reflection when you rotate it. |
| **Nice Sphere** | Maps the texture to a sphere, deforming it when you rotate but you still can see the texture's wrap |
| **6 Frames Layout** | The texture contains six images arranged in one of the standard cubemap layouts, cross or sequence (+x -x +y -y +z -z) and the images can be in either horizontal or vertical orientation. |
| **Fixup edge seams** | Removes visual artifacts at the joined edges of the map image(s), which will be visible with glossy reflections. |
| **Filter Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations: |
| **Point** | The Texture becomes blocky up close |
| **Bilinear** | The Texture becomes blurry up close |
| **Trilinear** | Like Bilinear, but the Texture also blurs between the different mip levels |
| **Aniso Level** | Increases texture quality when viewing the texture at a steep angle. Good for floor and ground textures, see below. |

An interesting way to add a lot of visual detail to your scenes is to use **Cookies** - greyscale textures you use to control the precise look of in-game lighting. This is fantastic for making moving clouds and giving an impression of dense foliage. The Light page has more info on all this, but the main thing is that for textures to be usable for cookies you just need to set the **Texture Type** to Cookie.



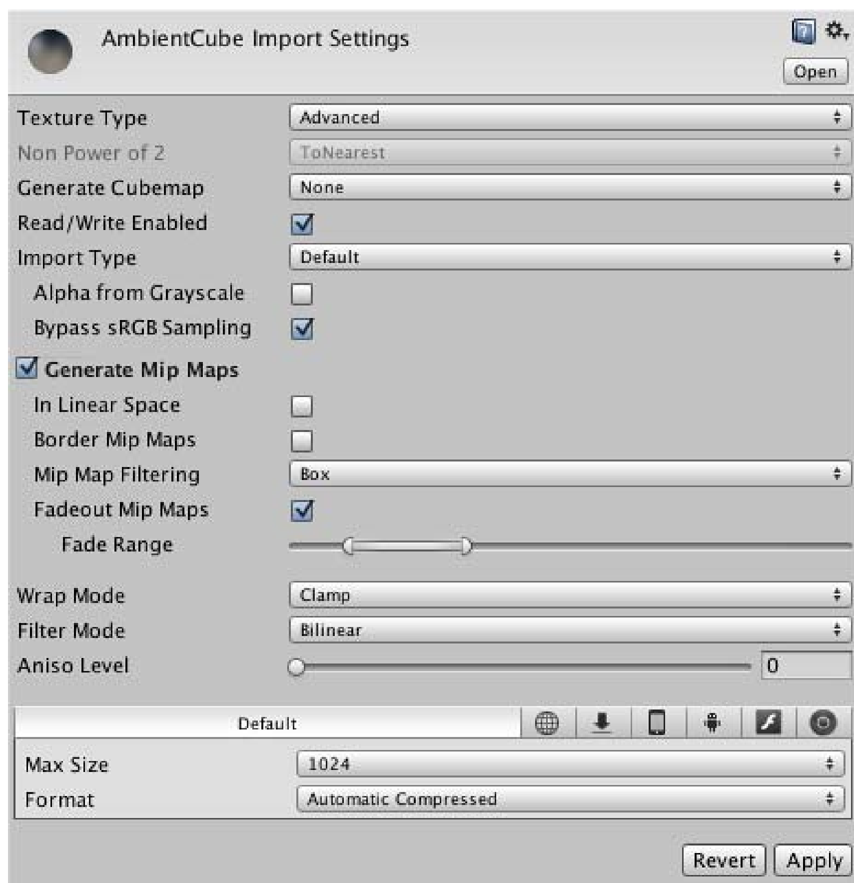

*Cookie Settings in the Texture Importer*

| | |
|---|---|
| **Light Type** | Type of light that the texture will be applied to. (This can be Spotlight, Point or Directional lights). For Directional Lights this texture will tile, so in the texture inspector, you must set the Edge Mode to **Repeat**; for SpotLights You should keep the edges of your cookie texture solid black in order to get the proper effect. In the Texture Inspector, set the Edge Mode to **Clamp**. |
| **Mapping** | (Point light only) Options for mapping the texture onto the spherical cast of the point light. |
| **Sphere Mapped** | Maps the texture to a "sphere like" cubemap. |
| **Cylindrical** | Maps the texture to a cylinder, use this when you want to use reflections on objects that are like cylinders. |
| **Simple Sphere** | Maps the texture to a simple sphere, deforming the reflection when you rotate it. |
| **Nice Sphere** | Maps the texture to a sphere, deforming it when you rotate but you still can see the texture's wrap |
| **6 Frames Layout** | The texture contains six images arranged in one of the standard cubemap layouts, cross or sequence (+x -x +y -y +z -z) and the images can be in either horizontal or vertical orientation. |
| **Fixup edge seams** | (Point light only) Removes visual artifacts at the joined edges of the map image(s). |
| **Alpha from Greyscale** | If enabled, an alpha transparency channel will be generated by the image's existing values of light & dark. |

AmbientCube Import Settings
Open
Texture Type: Lightmap
Filter Mode: Bilinear
Aniso Level: 0
Default
Max Size: 1024
Format: Compressed
Revert Apply

*Lightmap settings in the Texture Importer*

| | |
|---|---|
| **Filter Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations: |
| **Point** | The Texture becomes blocky up close |
| **Bilinear** | The Texture becomes blurry up close |
| **Trilinear** | Like Bilinear, but the Texture also blurs between the different mip levels |
| **Aniso Level** | Increases texture quality when viewing the texture at a steep angle. Good for floor and ground textures, see below. |



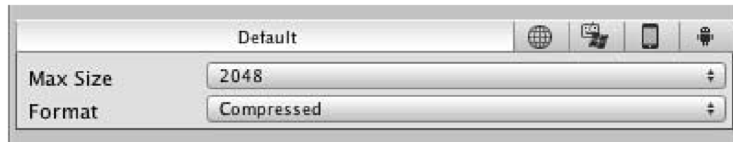

*The Advanced Texture Importer Settings dialog*

| | |
|---|---|
| **Non Power of 2** | If texture has non-power-of-two size, this will define a scaling behavior at import time (for more info see the Texture Sizes section below): |
| **None** | Texture will be padded to the next larger power-of-two size for use with GUITexture component. |
| **To nearest** | Texture will be scaled to the nearest power-of-two size at import time. For instance 257x511 texture will become 256x512. Note that PVRTC formats require textures to be square (width equal to height), therefore final size will be upscaled to 512x512. |
| **To larger** | Texture will be scaled to the next larger power-of-two size at import time. For instance 257x511 texture will become 512x512. |
| **To smaller** | Texture will be scaled to the next smaller power-of-two size at import time. For instance 257x511 texture will become 256x256. |

| | |
|---|---|
| **Generate Cube Map** | Generates a cubemap from the texture using different generation methods. |
| **Spheremap** | Maps the texture to a "sphere like" cubemap. |
| **Cylindrical** | Maps the texture to a cylinder, use this when you want to use reflections on objects that are like cylinders. |
| **SimpleSpheremap** | Maps the texture to a simple sphere, deforming the reflection when you rotate it. |
| **NiceSpheremap** | Maps the texture to a sphere, deforming it when you rotate but you still can see the texture's wrap |
| **FacesVertical** | The texture contains the six faces of the cube arranged in a vertical strip in the order +x -x +y -y +z -z. |
| **FacesHorizontal** | The texture contains the six faces of the cube arranged in a horizontal strip in the order +x -x +y -y +z -z. |
| **CrossVertical** | The texture contains the six faces of the cube arranged in a vertically oriented cross. |
| **CrossHorizontal** | The texture contains the six faces of the cube arranged in a horizontally oriented cross. |
| **Read/Write Enabled** | Select this to enable access to the texture data from scripts (GetPixels, SetPixels and other Texture2D functions). Note however that a copy of the texture data will be made, doubling the amount of memory required for texture asset. Use only if absolutely necessary. This is only valid for uncompressed and DTX compressed textures, other types of compressed textures cannot be read from. Disabled by default. |
| **Import Type** | The way the image data is interpreted. |
| **Default** | Standard texture. |
| **Normal Map** | Texture is treated as a normal map (enables other options) |
| **Lightmap** | Texture is treated as a lightmap (disables other options) |
| **Alpha from grayscale** | (Default mode only) Generates the alpha channel from the luminance information in the image |
| **Create from grayscale** | (Normal map mode only) Creates the map from the luminance information in the image |
| **Bypass sRGB sampling** | (Default mode only) Use the exact colour values from the image rather than compensating for gamma (useful when the texture is for GUI or used as a way to encode non-image data) |
| **Generate Mip Maps** | Select this to enable mip-map generation. Mip maps are smaller versions of the texture that get used when the texture is very small on screen. For more info, see Mip Maps below. |
| **In Linear Space** | Generate mipmaps in linear colour space. |
| **Border Mip Maps** | Select this to avoid colors seeping out to the edge of the lower Mip levels. Used for light cookies (see below). |
| **Mip Map Filtering** | Two ways of mip map filtering are available to optimize image quality: |
| **Box** | The simplest way to fade out the mipmaps - the mip levels become smoother and smoother as they go down in size. |
| **Kaiser** | A sharpening Kaiser algorithm is run on the mip maps as they go down in size. If your textures are too blurry in the distance, try this option. |
| **Fade Out Mipmaps** | Enable this to make the mipmaps fade to gray as the mip levels progress. This is used for detail maps. The left most scroll is the first mip level to begin fading out at. The rightmost scroll defines the mip level where the texture is completely grayed out |
| **Wrap Mode** | Selects how the Texture behaves when tiled: |
| **Repeat** | The Texture repeats (tiles) itself |
| **Clamp** | The Texture's edges get stretched |
| **Filter Mode** | Selects how the Texture is filtered when it gets stretched by 3D transformations: |
| **Point** | The Texture becomes blocky up close |
| **Bilinear** | The Texture becomes blurry up close |
| **Trilinear** | Like Bilinear, but the Texture also blurs between the different mip levels |
| **Aniso Level** | Increases texture quality when viewing the texture at a steep angle. Good for floor and ground textures, see below. |

## Per-Platform Overrides

When you are building for different platforms, you have to think about the resolution of your textures for the target platform, the size and the quality. You can set default options and then override the defaults for a specific platform.




*Default settings for all platforms.*

| | |
|---|---|
| **Max Texture Size** | The maximum imported texture size. Artists often prefer to work with huge textures - scale the texture down to a suitable size with this. |
| **Texture Format** | What internal representation is used for the texture. This is a tradeoff between size and quality. In the examples below we show the final size of a in-game texture of 256 by 256 pixels: |

| | |
|---|---|
| **Compressed** | Compressed RGB texture. This is the most common format for diffuse textures. 4 bits per pixel (32 KB for a 256x256 texture). |
| **16 bit** | Low-quality truecolor. Has 16 levels of red, green, blue and alpha. |
| **Truecolor** | Truecolor, this is the highest quality. At 256 KB for a 256x256 texture. |

If you have set the **Texture Type** to **Advanced** then the **Texture Format** has different values.

## Desktop

| | |
|---|---|
| **Texture Format** | What internal representation is used for the texture. This is a tradeoff between size and quality. In the examples below we show the final size of an in-game texture of 256 by 256 pixels: |
| **RGB Compressed DXT1** | Compressed RGB texture. This is the most common format for diffuse textures. 4 bits per pixel (32 KB for a 256x256 texture). |
| **RGBA Compressed DXT5** | Compressed RGBA texture. This is the main format used for diffuse & specular control textures. 1 byte/pixel (64 KB for a 256x256 texture). |
| **RGB 16 bit** | 65 thousand colors with no alpha. Compressed DXT formats use less memory and usually look better. 128 KB for a 256x256 texture. |
| **RGB 24 bit** | Truecolor but without alpha. 192 KB for a 256x256 texture. |
| **Alpha 8 bit** | High quality alpha channel but without any color. 64 KB for a 256x256 texture. |
| **RGBA 16 bit** | Low-quality truecolor. Has 16 levels of red, green, blue and alpha. Compressed DXT5 format uses less memory and usually looks better. 128 KB for a 256x256 texture. |
| **RGBA 32 bit** | Truecolor with alpha - this is the highest quality. At 256 KB for a 256x256 texture, this one is expensive. Most of the time, **DXT5** offers sufficient quality at a much smaller size. The main way this is used is for normal maps, as DXT compression there often carries a visible quality loss. |

## iOS

| | |
|---|---|
| **Texture Format** | What internal representation is used for the texture. This is a tradeoff between size and quality. In the examples below we show the final size of a in-game texture of 256 by 256 pixels: |
| **RGB Compressed PVRTC 4 bits** | Compressed RGB texture. This is the most common format for diffuse textures. 4 bits per pixel (32 KB for a 256x256 texture) |
| **RGBA Compressed PVRTC 4 bits** | Compressed RGBA texture. This is the main format used for diffuse & specular control textures or diffuse textures with transparency. 4 bits per pixel (32 KB for a 256x256 texture) |
| **RGB Compressed PVRTC 2 bits** | Compressed RGB texture. Lower quality format suitable for diffuse textures. 2 bits per pixel (16 KB for a 256x256 texture) |
| **RGBA Compressed PVRTC 2 bits** | Compressed RGBA texture. Lower quality format suitable for diffuse & specular control textures. 2 bits per pixel (16 KB for a 256x256 texture) |
| **RGB Compressed DXT1** | Compressed RGB texture. This format is not supported on iOS, but kept for backwards compatibility with desktop projects. |
| **RGBA Compressed DXT5** | Compressed RGBA texture. This format is not supported on iOS, but kept for backwards compatibility with desktop projects. |
| **RGB 16 bit** | 65 thousand colors with no alpha. Uses more memory than PVRTC formats, but could be more suitable for UI or crisp textures without gradients. 128 KB for a 256x256 texture. |
| **RGB 24 bit** | Truecolor but without alpha. 192 KB for a 256x256 texture. |
| **Alpha 8 bit** | High quality alpha channel but without any color. 64 KB for a 256x256 texture. |
| **RGBA 16 bit** | Low-quality truecolor. Has 16 levels of red, green, blue and alpha. Uses more memory than PVRTC formats, but can be handy if you need exact alpha channel. 128 KB for a 256x256 texture. |
| **RGBA 32 bit** | Truecolor with alpha - this is the highest quality. At 256 KB for a 256x256 texture, this one is expensive. Most of the time, **PVRTC** formats offers sufficient quality at a much smaller size. |
| **Compression quality** | Choose Fast for quickest performance, Best for the best image quality and Normal for a balance between the two. |

## Android

| | |
|---|---|
| **Texture Format** | What internal representation is used for the texture. This is a tradeoff between size and quality. In the examples below we show the final size of a in-game texture of 256 by 256 pixels: |
| **RGB Compressed DXT1** | Compressed RGB texture. Supported by Nvidia Tegra. 4 bits per pixel (32 KB for a 256x256 texture). |

| | |
|---|---|
| **RGBA Compressed DXT5** | Compressed RGBA texture. Supported by Nvidia Tegra. 6 bits per pixel (64 KB for a 256x256 texture). |
| **RGB Compressed ETC 4 bits** | Compressed RGB texture. This is the default texture format for Android projects. ETC1 is part of OpenGL ES 2.0 and is supported by all OpenGL ES 2.0 GPUs. It does not support alpha. 4 bits per pixel (32 KB for a 256x256 texture) |
| **RGB Compressed PVRTC 2 bits** | Compressed RGB texture. Supported by Imagination PowerVR GPUs. 2 bits per pixel (16 KB for a 256x256 texture) |
| **RGBA Compressed PVRTC 2 bits** | Compressed RGBA texture. Supported by Imagination PowerVR GPUs. 2 bits per pixel (16 KB for a 256x256 texture) |
| **RGB Compressed PVRTC 4 bits** | Compressed RGB texture. Supported by Imagination PowerVR GPUs. 4 bits per pixel (32 KB for a 256x256 texture) |
| **RGBA Compressed PVRTC 4 bits** | Compressed RGBA texture. Supported by Imagination PowerVR GPUs. 4 bits per pixel (32 KB for a 256x256 texture) |
| **RGB Compressed ATC 4 bits** | Compressed RGB texture. Supported by Qualcomm Snapdragon. 4 bits per pixel (32 KB for a 256x256 texture). |
| **RGBA Compressed ATC 8 bits** | Compressed RGBA texture. Supported by Qualcomm Snapdragon. 6 bits per pixel (64 KB for a 256x256 texture). |
| **RGB 16 bit** | 65 thousand colors with no alpha. Uses more memory than the compressed formats, but could be more suitable for UI or crisp textures without gradients. 128 KB for a 256x256 texture. |
| **RGB 24 bit** | Truecolor but without alpha. 192 KB for a 256x256 texture. |
| **Alpha 8 bit** | High quality alpha channel but without any color. 64 KB for a 256x256 texture. |
| **RGBA 16 bit** | Low-quality truecolor. The default compression for the textures with alpha channel. 128 KB for a 256x256 texture. |
| **RGBA 32 bit** | Truecolor with alpha - this is the highest quality compression for the textures with alpha. 256 KB for a 256x256 texture. |
| **Compression quality** | Choose Fast for quickest performance, Best for the best image quality and Normal for a balance between the two. |

Unless you're targeting a specific hardware, like Tegra, we'd recommend using ETC1 compression. If needed you could store an external alpha channel and still benefit from lower texture footprint. If you absolutely want to store an alpha channel in a texture, RGBA16 bit is the compression supported by all hardware vendors.

Textures can be imported from DDS files but only DXT or uncompressed pixel formats are currently supported.

If your app utilizes an unsupported texture compression, the textures will be uncompressed to RGBA 32 and stored in memory along with the compressed ones. So in this case you lose time decompressing textures and lose memory storing them twice. It may also have a very negative impact on rendering performance.

# Flash

| | |
|---|---|
| **Format** | Image format |
| **RGB JPG Compressed** | RGB image data compressed in JPG format |
| **RGBA JPG Compressed** | RGBA image data (ie, with alpha) compressed in JPG format |
| **RGB 24-bit** | Uncompressed RGB image data, 8 bits per channel |
| **RGBA 32-bit** | Uncompressed RGBA image data, 8 bits per channel |

## Details

#### Supported Formats

Unity can read the following file formats: PSD, TIFF, JPG, TGA, PNG, GIF, BMP, IFF, PICT. It should be noted that Unity can import multi-layer PSD & TIFF files just fine. They are flattened automatically on import but the layers are maintained in the assets themselves, so you don't lose any of your work when using these file types natively. This is important as it allows you to just have one copy of your textures that you can use from Photoshop, through your 3D modelling app and into Unity.

#### Texture Sizes

Ideally texture sizes should be powers of two on the sides. These sizes are as follows: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 or 2048 pixels. The textures do not have to be square, i.e. width can be different from height.

It is possible to use other (non power of two) texture sizes with Unity. Non power of two texture sizes work best when used on GUI Textures, however if used on anything else they will be converted to an uncompressed RGBA 32 bit format. That means they will take up more video memory (compared to PVRT(iOS)/DXT(Desktop) compressed textures), will be slower to load and

slower to render (if you are on iOS mode). In general you'll use non power of two sizes only for GUI purposes.

Non power of two texture assets can be scaled up at import time using the **Non Power of 2** option in the advanced texture type in the import settings. Unity will scale texture contents as requested, and in the game they will behave just like any other texture, so they can still be compressed and very fast to load.

One potential problem with using non power of two textures this is that Unity will convert these textures internally to power of two, and this stretching process can introduce minor visual artefacts.

**UV Mapping**

When mapping a 2D texture onto a 3D model, some sort of wrapping is done. This is called **UV mapping** and is done in your 3D modelling app. Inside Unity, you can scale and move the texture using Materials. Scaling normal & detail maps is especially useful.

**Mip Maps**

Mip Maps are a list of progressively smaller versions of an image, used to optimise performance on real-time 3D engines. Objects that are far away from the camera use the smaller texture versions. Using mip maps uses 33% more memory, but not using them can be a huge performance loss. You should always use mipmaps for in-game textures; the only exceptions are textures that will never be minified (e.g. GUI textures).

**Normal Maps**

Normal maps are used by normal map shaders to make low-polygon models look as if they contain more detail. Unity uses normal maps encoded as RGB images. You also have the option to generate a normal map from a grayscale height map image.

**Detail Maps**

If you want to make a terrain, you normally use your main texture to show where there are areas of grass, rocks sand, etc... If your terrain has a decent size, it will end up very blurry. Detail textures hide this fact by fading in small details as your main texture gets up close.

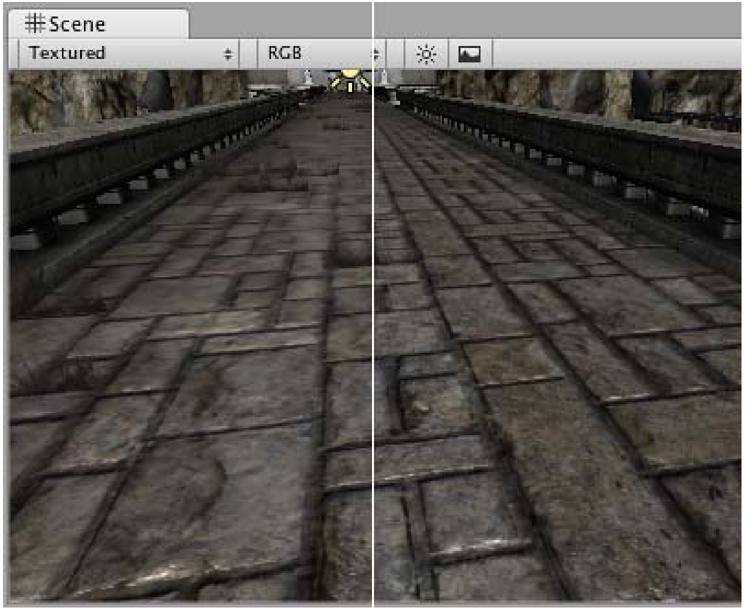
When drawing detail textures, a neutral gray is invisible, white makes the main texture twice as bright and black makes the main texture completely black.

**Reflections (Cube Maps)**

If you want to use texture for reflection maps (e.g. use the **Reflective** builtin shaders), you need to use Cubemap Textures.

**Anisotropic filtering**

Anisotropic filtering increases texture quality when viewed from a grazing angle, at some expense of rendering cost (the cost is entirely on the graphics card). Increasing anisotropy level is usually a good idea for ground and floor textures. In Quality Settings anisotropic filtering can be forced for all textures or disabled completely.



*No anisotropy (left) / Maximum anisotropy (right) used on the ground texture*

Page last updated: 2012-10-25

---

## comp-AudioGroup

These **Components** implement sound in Unity.

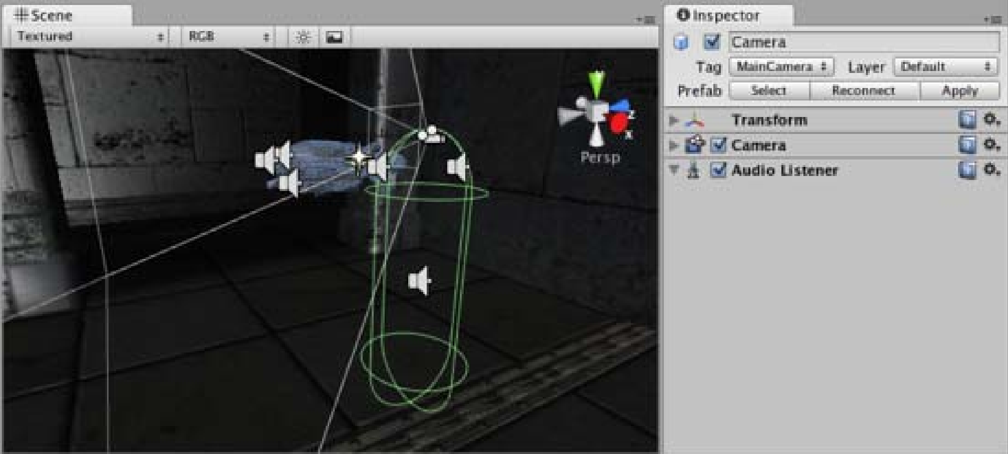
- Audio Listener - Add this to a **Camera** to get 3D positional sound.
- Audio Source - Add this Component to a **GameObject** to make it play a sound.

Page last updated: 2007-07-16

---

## class-AudioListener

The **Audio Listener** acts as a microphone-like device. It receives input from any given Audio Source in the scene and plays sounds through the computer speakers. For most applications it makes the most sense to attach the listener to the Main Camera. If an audio listener is within the boundaries of a Reverb Zone reverberation is applied to all audible sounds in the scene. (PRO only) Furthermore, Audio Effects can be applied to the listener and it will be applied to all audible sounds in the scene.




*The Audio Listener, attached to the Main Camera*

### Properties

The Audio Listener has no properties. It simply must be added to work. It is always added to the Main Camera by default.

### Details

The Audio Listener works in conjunction with Audio Sources, allowing you to create the aural experience for your games. When the Audio Listener is attached to a **GameObject** in your scene, any Sources that are close enough to the Listener will be picked up and output to the computer's speakers. Each scene can only have 1 Audio Listener to work properly.

If the Sources are 3D (see import settings in Audio Clip), the Listener will emulate position, velocity and orientation of the sound in the 3D world (You can tweak attenuation and 3D/2D behavior in great detail in Audio Source) . 2D will ignore any 3D processing. For example, if your character walks off a street into a night club, the night club's music should probably be 2D, while the individual voices of characters in the club should be mono with their realistic positioning being handled by Unity.

You should attach the Audio Listener to either the Main Camera or to the GameObject that represents the player. Try both to find what suits your game best.

### Hints

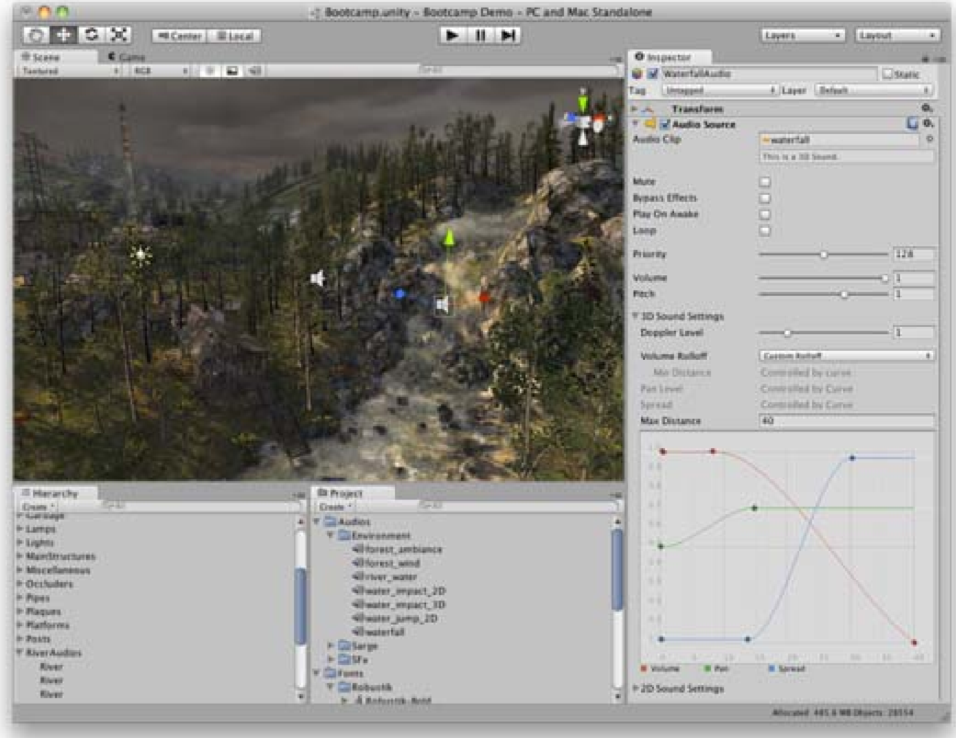
- Each scene can only have one Audio Listener.
- You access the project-wide audio settings using the Audio Manager, found in the **Edit->Project Settings->Audio** menu.
- View the Audio Clip Component page for more information about Mono vs Stereo sounds.

Page last updated: 2010-09-10

---

## class-AudioSource

The **Audio Source** plays back an Audio Clip in the scene. If the Audio Clip is a 3D clip, the source is played back at a given position and will attenuate over distance. The audio can be spread out between speakers (stereo to 7.1) (*Spread*) and morphed between 3D and 2D (*PanLevel*). This can be controlled over distance with falloff curves. Also, if the listener is within one or multiple Reverb Zones, reverberations is applied to the source. (PRO only) Individual filters can be applied to each audio source for an even richer audio experience. See Audio Effects for more details.




*The Audio Source gizmo in the **Scene View** and its settings in the **inspector**.*

### Properties

| | |
|---|---|
| **Audio Clip** | Reference to the sound clip file that will be played. |
| **Mute** | If enabled the sound will be playing but muted. |
| **Bypass Effects** | This Is to quickly "by-pass" filter effects applied to the audio source. An easy way to turn all effects on/off. |
| **Play On Awake** | If enabled, the sound will start playing the moment the scene launches. If disabled, you need to start it using the **Play()** command from scripting. |
| **Loop** | Enable this to make the **Audio Clip** loop when it reaches the end. |
| **Priority** | Determines the priority of this audio source among all the ones that coexist in the scene. (Priority: 0 = most important. 256 = least important. Default = 128.). Use 0 for music tracks to avoid it getting occasionally swapped out. |
| **Volume** | How loud the sound is at a distance of one world unit (one meter) from the **Audio Listener**. |
| **Pitch** | Amount of change in pitch due to slowdown/speed up of the **Audio Clip**. Value 1 is normal playback speed. |
| **3D Sound Settings** | Settings that are applied to the audio source if the Audio Clip is a 3D Sound. |
| **Pan Level** | Sets how much the 3d engine has an effect on the audio source. |
| **Spread** | Sets the spread angle to 3d stereo or multichannel sound in speaker space. |
| **Doppler Level** | Determines how much doppler effect will be applied to this audio source (if is set to 0, then no effect is applied). |

| | |
|---|---|
| **Min Distance** | Within the MinDistance, the sound will stay at loudest possible. Outside MinDistance it will begin to attenuate. Increase the MinDistance of a sound to make it 'louder' in a 3d world, and decrease it to make it 'quieter' in a 3d world. |
| **Max Distance** | The distance where the sound stops attenuating at. Beyond this point it will stay at the volume it would be at MaxDistance units from the listener and will not attenuate any more. |
| **Rolloff Mode** | How fast the sound fades. The higher the value, the closer the Listener has to be before hearing the sound.(This is determined by a Graph). |
| **Logarithmic Rolloff** | The sound is loud when you are close to the audio source, but when you get away from the object it decreases significantly fast. |
| **Linear Rolloff** | The further away from the audio source you go, the less you can hear it. |
| **Custom Rolloff** | The sound from the audio source behaves accordingly to how you set the graph of roll offs. |
| **2D Sound Settings** | Settings that are applied to the audio source if the Audio clip is a 2D Sound. |
| **Pan 2D** | Sets how much the engine has an effect on the audio source. |

### Types of Rolloff

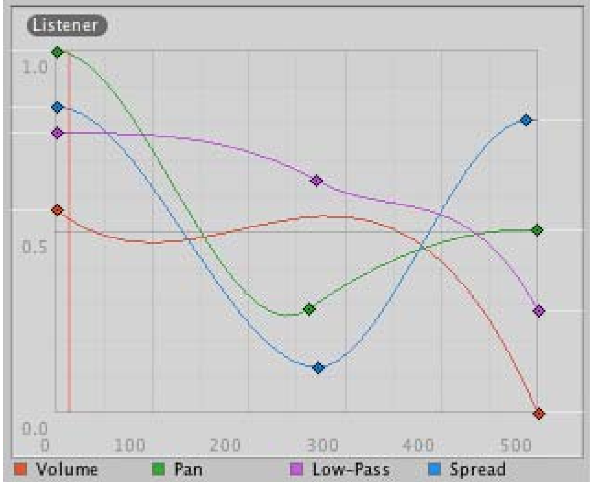
There are three Rolloff modes: Logarithmic, Linear and Custom Rolloff. The Custom Rolloff can be modified by modifying the volume distance curve as described below. If you try to modify the volume distance function when it is set to Logarithmic or Linear, the type will automatically change to Custom Rolloff.

*Rolloff Modes that an audio source can have.*

### Distance Functions

There are several properties of the audio that can be modified as a function of the distance between the audio source and the audio listener.

**Volume**: Amplitude(0.0 - 1.0) over distance.
**Pan**: Left(-1.0) to Right(1.0) over distance.
**Spread**: Angle (degrees 0.0 - 360.0) over distance.
**Low-Pass** (only if LowPassFilter is attached to the AudioSource): Cutoff Frequency (22000.0-10.0) over distance.




*Distance functions for Volume, Pan, Spread and Low-Pass audio filter. The current distance to the Audio Listener is marked in the graph.*

To modify the distance functions, you can edit the curves directly. For more information, see the guide to Editing Curves.

### Creating Audio Sources

Audio Sources don't do anything without an assigned **Audio Clip**. The Clip is the actual sound file that will be played back. The Source is like a controller for starting and stopping playback of that clip, and modifying other audio properties.

To create a new Audio Source:

1. Import your audio files into your Unity Project. These are now Audio Clips.
2. Go to **GameObject->Create Empty** from the menubar.
3. With the new GameObject selected, select **Component->Audio->Audio Source**.
4. Assign the **Audio Clip** property of the Audio Source Component in the Inspector.

**Note:** If you want to create an **Audio Source** just for one **Audio Clip** that you have in the Assets folder then you can just drag that clip to the scene view - a GameObject with an **Audio Source** component will be created automatically for it. Dragging a clip onto on existing GameObject will attach the clip along with a new **Audio Source** if there isn't one already there. If the object does already have an **Audio Source** then the newly dragged clip will replace the one that the source currently uses.

### Platform specific details

## iOS

On mobile platforms compressed audio is encoded as MP3 for speedier decompression. Beware that this compression can remove samples at the end of the clip and potentially break a "perfect-looping" clip. Make sure the clip is right on a specific MP3 sample boundary to avoid sample clipping - tools to perform this task are widely available. For performance reasons audio clips can be played back using the Apple hardware codec. To enable this, check the "Use Hardware" checkbox in the import settings. See the Audio Clip documentation for more details.

## Android

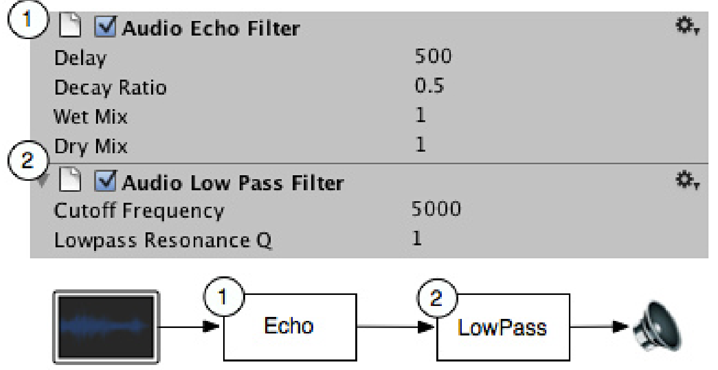
On mobile platforms compressed audio is encoded as MP3 for speedier decompression. Beware that this compression can remove samples at the end of the clip and potentially break a "perfect-looping" clip. Make sure the clip is right on a specific MP3 sample boundary to avoid sample clipping - tools to perform this task are widely available.

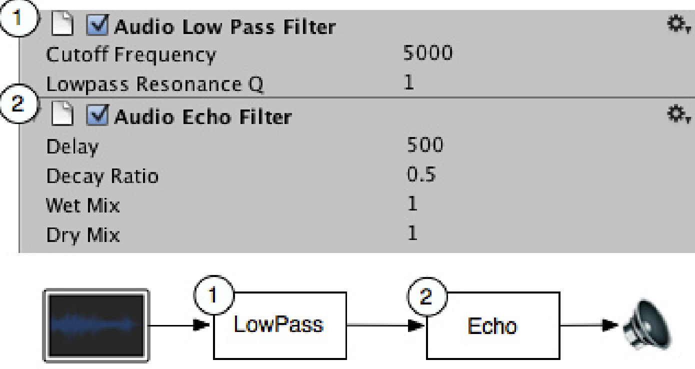
Page last updated: 2012-08-08

---

# class-AudioEffect

AudioSources and the AudioListener can have filter components applied, by adding the filter components to the same GameObject the AudioSource or AudioListener is on. Filter effects are serialized in the component order as seen here:






Enabling/Disabling a filter component will *bypass* the filter. Though highly optimized, some filters are still CPU intensive. Audio CPU usage can monitored in the profiler under the Audio Tab.

See these pages for more information on each filter type:

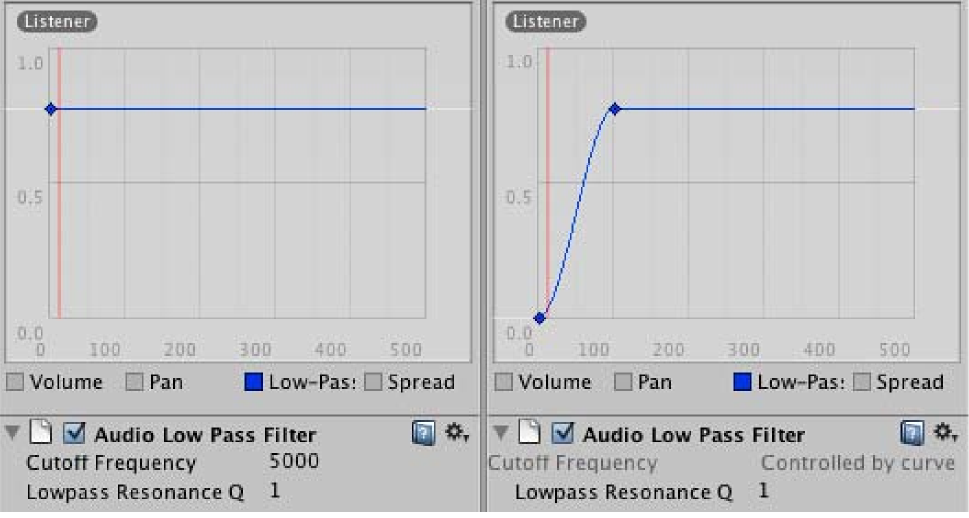
- Low Pass Filter
- High Pass Filter
- Echo Filter
- Distortion Filter
- Reverb Filter
- Chorus Filter

Page last updated: 2011-12-06

---

## class-AudioLowPassFilter

The **Audio Low Pass Filter** filter passes low frequencies of an AudioSource, or all sound reaching an AudioListener, and cuts frequencies higher than the **Cutoff Frequency**.
The **Lowpass Resonance Q** (known as Lowpass Resonance Quality Factor) determines how much the filter's self-resonance is dampened. Higher **Lowpass Resonance Q** indicates a lower rate of energy loss i.e. the oscillations die out more slowly.
The **Audio Low Pass Filter** has a Rolloff curve associated with it, making it possible to set **Cutoff Frequency** over distance between the AudioSource and the AudioListener.




*The Audio Low Pass filter properties in the inspector.*

### Properties

| | |
|---|---|
| **Cutoff Frequency** | Lowpass cutoff frequency in hz. 10.0 to 22000.0. Default = 5000.0. |
| **Lowpass Resonance Q** | Lowpass resonance Q value. 1.0 to 10.0. Default = 1.0. |

### Adding a low pass filter

To add a low pass filter to a given audio source just select the object in the inspector and then select **Component->Audio->Audio Low Pass Filter**.

### Hints

- Sounds propagates very differently given the environment. For example, to compliment a visual fog effect add a subtle low-pass to the Audio Listener.
- The high frequencies of a sound being emitted from behind a door will not reach the listener. To simulate this, simply change the **Cutoff Frequency** when opening the door.

Page last updated: 2012-08-08

---

## class-AudioHighPassFilter

The **Audio High Pass Filter** passes high frequencies of an AudioSource and cuts off signals with frequencies lower than the **Cutoff Frequency**.
The **Highpass resonance Q** (known as Highpass Resonance Quality Factor) determines how much the filter's self-resonance is dampened. Higher **Highpass resonance Q** indicates a lower rate of energy loss i.e. the oscillations die out more slowly.

*The Audio high Pass filter properties in the inspector.*

### Properties

| | |
|---|---|
| **Cutoff Frequency** | Highpass cutoff frequency in hz. 10.0 to 22000.0. Default = 5000.0. |
| **Highpass Resonance Q** | Highpass resonance Q value. 1.0 to 10.0. Default = 1.0. |

### Adding a high pass filter

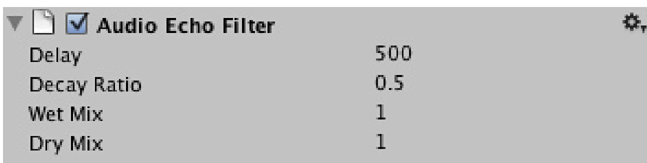
To add a high pass filter to a given audio source just select the object in the inspector and then select **Component->Audio->Audio High Pass Filter**.

Page last updated: 2010-09-17

---

## class-AudioEchoFilter

The **Audio Echo Filter** repeats a sound after a given **Delay**, attenuating the repetitions based on the **Decay Ratio**.
The **Wet Mix** determines the amplitude of the filtered signal, where the **Dry Mix** determines the amplitude of the unfiltered sound output.



*The **Audio Echo Filter** properties in the inspector.*

### Properties

| | |
|---|---|
| **Delay** | Echo delay in ms. 10 to 5000. Default = 500. |
| **Decay Ratio** | Echo decay per delay. 0 to 1. 1.0 = No decay, 0.0 = total decay (ie simple 1 line delay). Default = 0.5.L |
| **Wet Mix** | Volume of echo signal to pass to output. 0.0 to 1.0. Default = 1.0. |
| **Dry Mix** | Volume of original signal to pass to output. 0.0 to 1.0. Default = 1.0. |

### Adding an Echo filter

To add an Echo filter to a given audio source just select the object in the inspector and then select **Component->Audio->Audio Echo Filter**.

### Hints

- Hard surfaces reflects the propagation of sound. For example a large canyon can be made more convincing with the **Audio Echo Filter**.
- Sound propagates slower than light - we all know that from lightning and thunder. To simulate this, add an **Audio Echo Filter** to an event sound, set the **Wet Mix** to 0.0 and modulate the **Delay** to the distance between AudioSource and AudioListener.

Page last updated: 2012-08-08

---

## class-AudioDistorionFilter

The **Audio Distortion Filter** distorts the sound from an AudioSource or sounds reaching the AudioListener.

*The Audio Distortion Pass filter properties in the inspector.*

### Properties

| | |
|---|---|
| **Distortion** | Distortion value. 0.0 to 1.0. Default = 0.5. |

### Adding a Distortion filter

To add an **Audio Distortion Filter** to a selected AudioSource or AudioListener select the object in the inspector and then select **Component->Audio->Audio Distortion Filter**.

### Hints

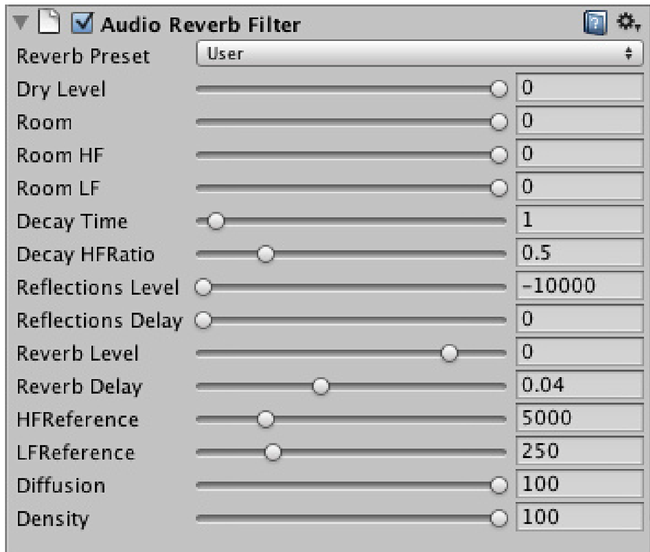
- Apply the **Audio Distortion Filter** to simulate the sound of a low quality radio transmission.

Page last updated: 2010-09-09

---

## class-AudioReverbFilter

The **Audio Reverb Filter** takes an Audio Clip and distorts it to create a personalized reverb effect.




*The Audio Reverb filter properties in the inspector.*

### Properties

| | |
|---|---|
| **Reverb Preset** | Custom reverb presets, select user to create your own customized reverbs. |
| **Dry Level** | Mix level of dry signal in output in mB. Ranges from -10000.0 to 0.0. Default is 0. |
| **Room** | Room effect level at low frequencies in mB. Ranges from -10000.0 to 0.0. Default is 0.0. |
| **Room HF** | Room effect high-frequency level in mB. Ranges from -10000.0 to 0.0. Default is 0.0. |

| | |
|---|---|
| **Room LF** | Room effect low-frequency level in mB. Ranges from -10000.0 to 0.0. Default is 0.0. |
| **Decay Time** | Reverberation decay time at low-frequencies in seconds. Ranges from 0.1 to 20.0. Default is 1.0. |
| **Decay HFRatio** | Decay HF Ratio : High-frequency to low-frequency decay time ratio. Ranges from 0.1 to 2.0. Default is 0.5. |
| **Reflections Level** | Early reflections level relative to room effect in mB. Ranges from -10000.0 to 1000.0. Default is -10000.0. |
| **Reflections Delay** | Early reflections delay time relative to room effect in mB. Ranges from -10000.0 to 2000.0. Default is 0.0. |
| **Reverb Level** | Late reverberation level relative to room effect in mB. Ranges from -10000.0 to 2000.0. Default is 0.0. |
| **Reverb Delay** | Late reverberation delay time relative to first reflection in seconds. Ranges from 0.0 to 0.1. Default is 0.04. |
| **HFReference** | Reference high frequency in Hz. Ranges from 20.0 to 20000.0. Default is 5000.0. |
| **LFReference** | Reference low-frequency in Hz. Ranges from 20.0 to 1000.0. Default is 250.0. |
| **Reflections Delay** | Late reverberation level relative to room effect in mB. Ranges from -10000.0 to 2000.0. Default is 0.0. |
| **Diffusion** | Reverberation diffusion (echo density) in percent. Ranges from 0.0 to 100.0. Default is 100.0. |
| **Density** | Reverberation density (modal density) in percent. Ranges from 0.0 to 100.0. Default is 100.0. |

**Note:** These values can only be modified if your **Reverb Preset** is set to **User**, else these values will be grayed out and they will have default values for each preset.

### Adding a reverb pass filter

To add a reverb pass filter to a given audio source just select the object in the inspector and then select **Component->Audio->Audio reverb Filter**.

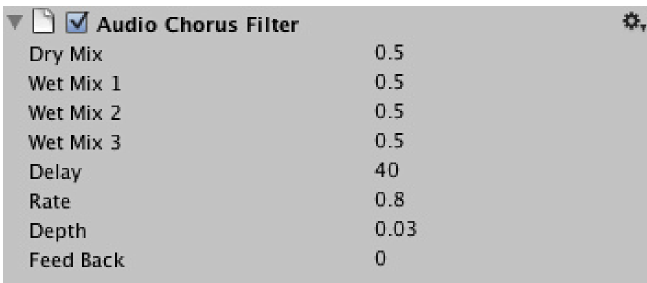
Page last updated: 2012-08-08

---

# class-AudioChorusFilter

The **Audio Chorus Filter** takes an Audio Clip and processes it creating a chorus effect.

The chorus effect modulates the original sound by a sinusoid low frequency oscillator (LFO). The output sounds like there are multiple sources emitting the same sound with slight variations - resembling a choir.




*The Audio high Pass filter properties in the inspector.*

### Properties

| | |
|---|---|
| **Dry Mix** | Volume of original signal to pass to output. 0.0 to 1.0. Default = 0.5. |
| **Wet Mix 1** | Volume of 1st chorus tap. 0.0 to 1.0. Default = 0.5. |
| **Wet Mix 2** | Volume of 2nd chorus tap. This tap is 90 degrees out of phase of the first tap. 0.0 to 1.0. Default = 0.5. |
| **Wet Mix 3** | Volume of 3rd chorus tap. This tap is 90 degrees out of phase of the second tap. 0.0 to 1.0. Default = 0.5. |
| **Delay** | The LFO's delay in ms. 0.1 to 100.0. Default = 40.0 ms |
| **Rate** | The LFO's modulation rate in Hz. 0.0 to 20.0. Default = 0.8 Hz. |
| **Depth** | Chorus modulation depth. 0.0 to 1.0. Default = 0.03. |
| **Feed Back** | Chorus feedback. Controls how much of the wet signal gets fed back into the filter's buffer. 0.0 to 1.0. Default = 0.0. |

### Adding a chorus filter

To add a chorus filter to a given audio source just select the object in the inspector and then select **Component->Audio->Audio Chorus Filter**.

### Hints

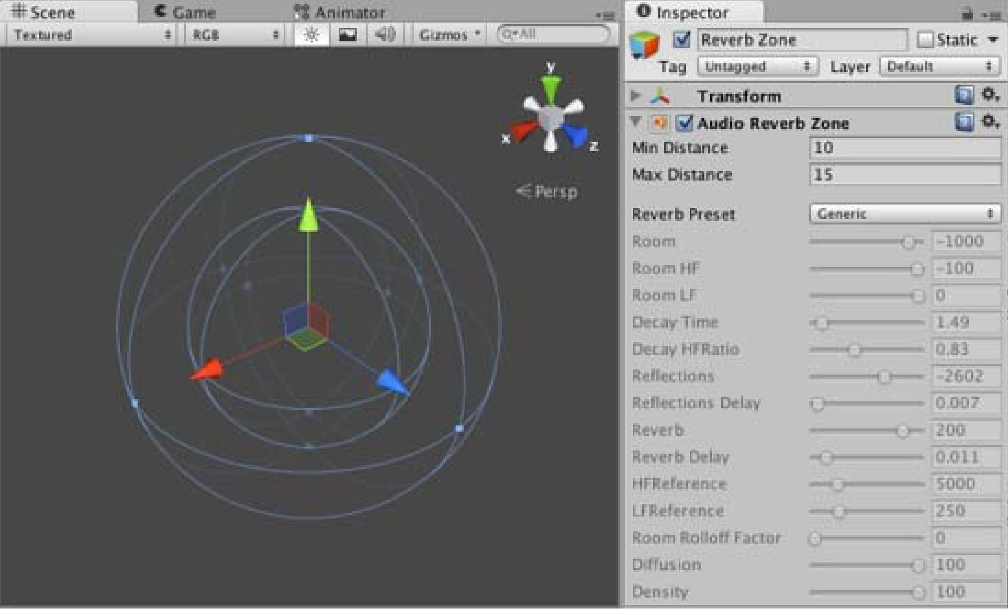
- You can tweak the chorus filter to create a flanger effect by lowering the feedback and decreasing the delay, as the flanger is a variant of the chorus.
- Creating a simple, dry echo is done by setting **Rate** and **Depth** to 0 and tweaking the mixes and **Delay**

Page last updated: 2012-08-08

---

## class-AudioReverbZone

**Reverb Zones** take an Audio Clip and distort it depending where the audio listener is located inside the reverb zone. They are used when you want to gradually change from a point where there is no ambient effect to a place where there is one, for example when you are entering a cavern.

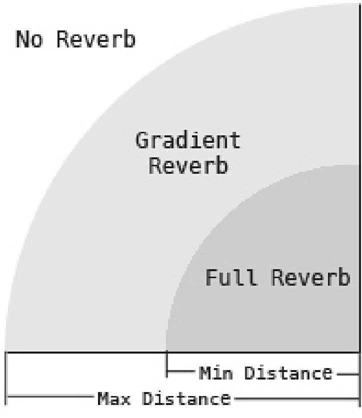



*The Audio Reverb Zone gizmo seen in the inspector.*

### Properties

| | |
|---|---|
| **Min Distance** | Represents the radius of the inner circle in the gizmo, this determines the zone where there is a gradually reverb effect and a full reverb zone. |
| **Max Distance** | Represents the radius of the outer circle in the gizmo, this determines the zone where there is no effect and where the reverb starts to get applied gradually. |
| **Reverb Preset** | Determines the reverb effect that will be used by the reverb zone. |

This diagram illustrates the properties of the reverb zone.




*How the sound works in a reverb zone*

### Adding a Reverb Zone

To add a Reverb Zone to a given audio source just select the object in the inspector and then select **Component->Audio->Audio Reverb Zone**.

### Hints.

- You can mix reverb zones to create combined effects

Page last updated: 2012-08-08

---

## class-Microphone

The **Microphone** class is useful for capturing input from a built-in (*physical*) microphone on your PC or mobile device.

With this class, you can start and end a recording from a built-in microphone, get a listing of available audio input devices (microphones), and find out the status of each such input device.

For more information on how to use this class, see the Scripting Reference.

Page last updated: 2011-10-18

---

## comp-DynamicsGroup

Unity has NVIDIA PhysX physics engine built-in. This allows for unique emergent behaviour and has many useful features.

### Basics

To put an object under physics control, simply add a **Rigidbody** to it. When you do this, the object will be affected by gravity,

and can collide with other objects in the world.

**Rigidbodies**

Rigidbodies are physically simulated objects. You use Rigidbodies for things that the player can push around, for example crates or loose objects, or you can move Rigidbodies around directly by adding forces to it by scripting.

If you move the Transform of a non-Kinematic Rigidbody directly it may not collide correctly with other objects. Instead you should move a Rigidbody by applying forces and torque to it. You can also add Joints to rigidbodies to make the behavior more complex. For example, you could make a physical door or a crane with a swinging chain.

You also use Rigidbodies to bring vehicles to life, for example you can make cars using a Rigidbody, 4 Wheel Colliders and a script applying wheel forces based on the user's Input.

You can make airplanes by applying forces to the Rigidbody from a script. Or you can create special vehicles or robots by adding various Joints and applying forces via scripting.

Rigidbodies are most often used in combination with primitive colliders.

**Tips:**

- You should never have a parent and child rigidbody together
- You should never scale the parent of a rigidbody

**Kinematic Rigidbodies**

A **Kinematic Rigidbody** is a Rigidbody that has the isKinematic option enabled. Kinematic Rigidbodies are not affected by forces, gravity or collisions. They are driven explicitly by setting the position and rotation of the Transform or animating them, yet they can interact with other non-Kinematic Rigidbodies.

Kinematic Rigidbodies correctly wake up other Rigidbodies when they collide with them, and they apply friction to Rigidbodies placed on top of them.

These are a few example uses for Kinematic Rigidbodies:

1. Sometimes you want an object to be under physics control but in another situation to be controlled explicitly from a script or animation. For example you could make an animated character whose bones have Rigidbodies attached that are connected with joints for use as a Ragdoll. Most of the time the character is under animation control, thus you make the Rigidbody Kinematic. But when he gets hit you want him to turn into a Ragdoll and be affected by physics. To accomplish this, you simply disable the isKinematic property.
2. Sometimes you want a moving object that can push other objects yet not be pushed itself. For example if you have an animated platform and you want to place some Rigidbody boxes on top, you should make the platform a Kinematic Rigidbody instead of just a **Collider** without a Rigidbody.
3. You might want to have a Kinematic Rigidbody that is animated and have a real Rigidbody follow it using one of the available Joints.

**Static Colliders**

A **Static Collider** is a GameObject that has a Collider but not a Rigidbody. Static Colliders are used for level geometry which always stays at the same place and never moves around. You can add a **Mesh Collider** to your already existing graphical meshes (even better use the **Import Settings** Generate Colliders check box), or you can use one of the other Collider types.

You should never move a Static Collider on a frame by frame basis. Moving Static Colliders will cause an internal recomputation in PhysX that is quite expensive and which will result in a big drop in performance. On top of that the behaviour of waking up other Rigidbodies based on a Static Collider is undefined, and moving Static Colliders will not apply friction to Rigidbodies that touch it. Instead, Colliders that move should always be Kinematic Rigidbodies.

**Character Controllers**

You use Character Controllers if you want to make a humanoid character. This could be the main character in a third person platformer, FPS shooter or any enemy characters.

These Controllers don't follow the rules of physics since it will not feel right (in Doom you run 90 miles per hour, come to halt in one frame and turn on a dime). Instead, a Character Controller performs collision detection to make sure your characters can slide along walls, walk up and down stairs, etc.

Character Controllers are not affected by forces but they can push Rigidbodies by applying forces to them from a script. Usually, all humanoid characters are implemented using Character Controllers.

Character Controllers are inherently unphysical, thus if you want to apply real physics - Swing on ropes, get pushed by big rocks - to your character you have to use a Rigidbody, this will let you use joints and forces on your character. Character Controllers are always aligned along the Y axis, so you also need to use a Rigidbody if your character needs to be able to change orientation in space (for example under a changing gravity). However, be aware that tuning a Rigidbody to feel right for a character is hard due to the unphysical way in which game characters are expected to behave. Another difference is that Character Controllers can slide smoothly over steps of a specified height, while Rigidbodies will not.

If you parent a Character Controller with a Rigidbody you will get a "Joint" like behavior.

### Component Details

#### Physics Control

- Rigidbody - Rigidbodies put objects under physics control.
- Constant Force - A utility component that adds a constant force to a rigidbody. Great for rockets and other quick functionality.

#### Colliders

- Sphere Collider - use for sphere-shaped objects.
- Box Collider - use for box-shaped objects.
- Capsule Collider - use for capsule-like (a cylinder with hemisphere ends) objects.
- Mesh Collider - takes the graphical mesh and uses it as a collision shape.
- Physic Material - contains settings allowing you to fine-tune your object's physical properties (friction, bounce, etc).

#### Joints

- Hinge Joint - Used to make door hinges.
- Spring Joint - A spring-like joint.
- Fixed Joint - Use to "lock" objects together.
- Configurable Joint - Use create complex joint behaviors of virtually any kind

#### Special Function

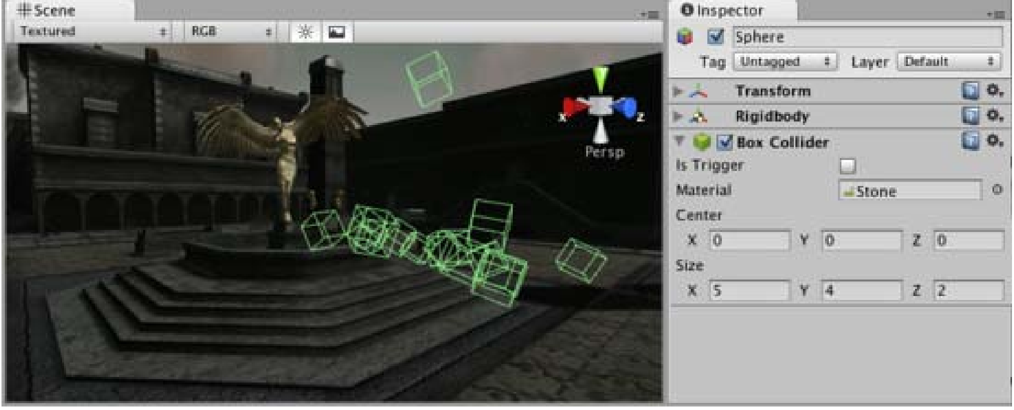
- Character Controller and Character Joint - Used to make character controllers.
- Wheel Collider - A special collider for grounded vehicles.
- Skinned Cloth - Used to create Skinned cloth
- Interactive Cloth - Used to create Interactive cloths, this is just normal cloth being simulated.

Page last updated: 2010-05-05

---

## class-BoxCollider

The **Box Collider** is a basic cube-shaped collision primitive.



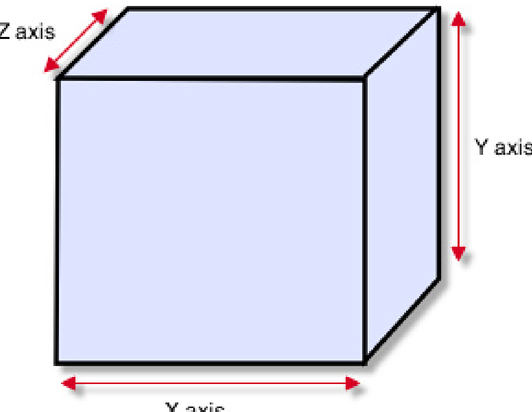

*A pile of Box Colliders*

### Properties

| | |
|---|---|
| **Is Trigger** | If enabled, this Collider is used for triggering events, and is ignored by the physics engine. |
| **Material** | Reference to the Physics Material that determines how this Collider interacts with others. |
| **Center** | The position of the Collider in the object's local space. |
| **Size** | The size of the Collider in the X, Y, Z directions. |

## Details

The Box Collider can be resized into different shapes of rectangular prisms. It works great for doors, walls, platforms, etc. It is also effective as a human torso in a ragdoll or as a car hull in a vehicle. Of course, it works perfectly for just boxes and crates as well!




*A standard Box Collider*

Colliders work with Rigidbodies to bring physics in Unity to life. Whereas Rigidbodies allow objects to be controlled by physics, Colliders allow objects to collide with each other. Colliders must be added to objects independently of Rigidbodies. A Collider does not necessarily need a Rigidbody attached, but a Rigidbody **must** be attached in order for the object to move as a result of collisions.

When a collision between two Colliders occurs and if at least one of them has a Rigidbody attached, three collision messages are sent out to the objects attached to them. These events can be handled in scripting, and allow you to create unique behaviors with or without making use of the built-in NVIDIA PhysX engine.

#### Triggers

An alternative way of using Colliders is to mark them as a **Trigger**, just check the IsTrigger property checkbox in the Inspector. Triggers are effectively ignored by the physics engine, and have a unique set of three trigger messages that are sent out when a collision with a Trigger occurs. Triggers are useful for triggering other events in your game, like cutscenes, automatic door opening, displaying tutorial messages, etc. Use your imagination!

Be aware that in order for two Triggers to send out trigger events when they collide, one of them must include a Rigidbody as well. For a Trigger to collide with a normal Collider, one of them must have a Rigidbody attached. For a detailed chart of different types of collisions, see the collision action matrix in the Advanced section below.

#### Friction and bounciness

Friction, bounciness and softness are defined in the Physisc Material. The Standard Assets contain the most common physics materials. To use one of them click on the Physics Material drop-down and select one, eg. Ice. You can also create your own physics materials and tweak all friction values.

## Compound Colliders

Compound Colliders are combinations of primitive Colliders, collectively acting as a single Collider. They come in handy when you have a complex mesh to use in collisions but cannot use a **Mesh Collider**. To create a Compound Collider, create child objects of your colliding object, then add a primitive Collider to each child object. This allows you to position, rotate, and scale each Collider easily and independently of one another.

*A real-world Compound Collider setup*

In the above picture, the *Gun Model* GameObject has a Rigidbody attached, and multiple primitive Colliders as child GameObjects. When the Rigidbody parent is moved around by forces, the child Colliders move along with it. The primitive Colliders will collide with the environment's Mesh Collider, and the parent Rigidbody will alter the way it moves based on forces being applied to it and how its child Colliders interact with other Colliders in the Scene.

Mesh Colliders can't normally collide with each other. If a Mesh Collider is marked as **Convex**, then it can collide with another Mesh Collider. The typical solution is to use primitive Colliders for any objects that move, and Mesh Colliders for static background objects.

## Hints

- To add multiple Colliders for an object, create child GameObjects and attach a Collider to each one. This allows each Collider to be manipulated independently.
- You can look at the gizmos in the **Scene View** to see how the Collider is being calculated on your object.
- Colliders do their best to match the scale of an object. If you have a non-uniform scale (a scale which is different in each direction), only the Mesh Collider can match completely.
- If you are moving an object through its Transform component but you want to receive Collision/Trigger messages, you must attach a Rigidbody to the object that is moving.

## Advanced

#### Collider combinations

There are numerous different combinations of collisions that can happen in Unity. Each game is unique, and different combinations may work better for different types of games. If you're using physics in your game, it will be very helpful to understand the different basic Collider types, their common uses, and how they interact with other types of objects.

#### Static Collider

These are GameObjects that do **not** have a Rigidbody attached, but **do** have a Collider attached. These objects should remain still, or move very little. These work great for your environment geometry. They will not move if a Rigidbody collides with them.

#### Rigidbody Collider

These GameObjects contain both a Rigidbody and a Collider. They are completely affected by the physics engine through scripted forces and collisions. They might collide with a GameObject that only contains a Collider. These will likely be your primary type of Collider in games that use physics.

#### Kinematic Rigidbody Collider

This GameObject contains a Collider and a Rigidbody which is marked **IsKinematic**. To move this GameObject, you modify its Transform Component, rather than applying forces. They're similar to Static Colliders but will work better when you want to move the Collider around frequently. There are some other specialized scenarios for using this GameObject.

This object can be used for circumstances in which you would normally want a Static Collider to send a trigger event. Since a Trigger must have a Rigidbody attached, you should add a Rigidbody, then enable **IsKinematic**. This will prevent your Object from moving from physics influence, and allow you to receive trigger events when you want to.

Kinematic Rigidbodies can easily be turned on and off. This is great for creating ragdolls, when you normally want a character to follow an animation, then turn into a ragdoll when a collision occurs, prompted by an explosion or anything else you choose. When this happens, simply turn all your Kinematic Rigidbodies into normal Rigidbodies through scripting.

If you have Rigidbodies come to rest so they are not moving for some time, they will "fall asleep". That is, they will not be calculated during the physics update since they are not going anywhere. If you move a Kinematic Rigidbody out from underneath normal Rigidbodies that are at rest on top of it, the sleeping Rigidbodies will "wake up" and be correctly calculated again in the physics update. So if you have a lot of Static Colliders that you want to move around and have different object fall on them correctly, use Kinematic Rigidbody Colliders.

**Collision action matrix**

Depending on the configurations of the two colliding Objects, a number of different actions can occur. The chart below outlines what you can expect from two colliding Objects, based on the components that are attached to them. Some of the combinations only cause one of the two Objects to be affected by the collision, so keep the standard rule in mind - physics will not be applied to objects that do not have Rigidbodies attached.

| **Collision detection occurs and messages are sent upon collision** | | | | | | |
|---|---|---|---|---|---|---|
| | Static Collider | Rigidbody Collider | Kinematic Rigidbody Collider | Static Trigger Collider | Rigidbody Trigger Collider | Kinematic Rigidbody Trigger Collider |
| Static Collider | | Y | | | | |
| Rigidbody Collider | Y | Y | Y | | | |
| Kinematic Rigidbody Collider | | Y | | | | |
| Static Trigger Collider | | | | | | |
| Rigidbody Trigger Collider | | | | | | |
| Kinematic Rigidbody Trigger Collider | | | | | | |

| **Trigger messages are sent upon collision** | | | | | | |
|---|---|---|---|---|---|---|
| | Static Collider | Rigidbody Collider | Kinematic Rigidbody Collider | Static Trigger Collider | Rigidbody Trigger Collider | Kinematic Rigidbody Trigger Collider |
| Static Collider | | | | | Y | Y |
| Rigidbody Collider | | | | Y | Y | Y |
| Kinematic Rigidbody Collider | | | | Y | Y | Y |
| Static Trigger Collider | | Y | Y | | Y | Y |
| Rigidbody Trigger Collider | Y | Y | Y | Y | Y | Y |
| Kinematic Rigidbody Trigger Collider | Y | Y | Y | Y | Y | Y |

### Layer-Based Collision Detection

In Unity 3.x we introduce something called **Layer-Based Collision Detection**, and you can now selectively tell Unity GameObjects to collide with specific layers they are attached to. For more information click here

Page last updated: 2009-07-16

---

# class-CapsuleCollider

The **Capsule Collider** is made of two half-spheres joined together by a cylinder. It is the same shape as the Capsule primitive.

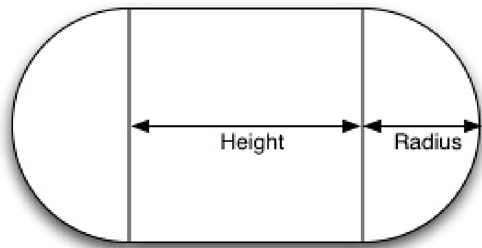
*A pile of Capsule Colliders*

## Properties

| | |
|---|---|
| **Is Trigger** | If enabled, this Collider is used for triggering events, and is ignored by the physics engine. |
| **Material** | Reference to the Physics Material that determines how this Collider interacts with others. |
| **Center** | The position of the Collider in the object's local space. |
| **Radius** | The radius of the Collider's local width. |
| **Height** | The total height of the Collider. |
| **Direction** | The axis of the capsule's lengthwise orientation in the object's local space. |

## Details

You can adjust the Capsule Collider's **Radius** and **Height** independently of each other. It is used in the Character Controller and works well for poles, or can be combined with other Colliders for unusual shapes.




*A standard Capsule Collider*

Colliders work with Rigidbodies to bring physics in Unity to life. Whereas Rigidbodies allow objects to be controlled by physics, Colliders allow objects to collide with each other. Colliders must be added to objects independently of Rigidbodies. A Collider does not necessarily need a Rigidbody attached, but a Rigidbody **must** be attached in order for the object to move as a result of collisions.

When a collision between two Colliders occurs and if at least one of them has a Rigidbody attached, three collision messages are sent out to the objects attached to them. These events can be handled in scripting, and allow you to create unique behaviors with or without making use of the built-in NVIDIA PhysX engine.

#### Triggers

An alternative way of using Colliders is to mark them as a **Trigger**, just check the IsTrigger property checkbox in the Inspector. Triggers are effectively ignored by the physics engine, and have a unique set of three trigger messages that are sent out when a collision with a Trigger occurs. Triggers are useful for triggering other events in your game, like cutscenes, automatic door opening, displaying tutorial messages, etc. Use your imagination!

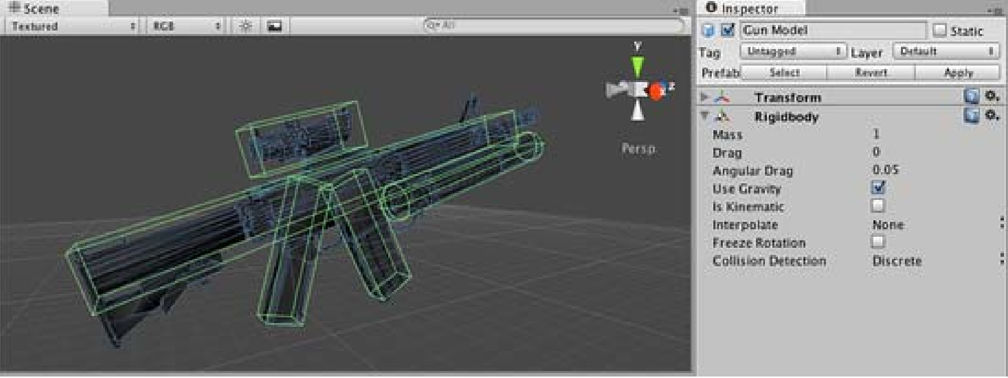
Be aware that in order for two Triggers to send out trigger events when they collide, one of them must include a Rigidbody as well. For a Trigger to collide with a normal Collider, one of them must have a Rigidbody attached. For a detailed chart of different types of collisions, see the collision action matrix in the Advanced section below.

#### Friction and bounciness

Friction, bounciness and softness are defined in the Physic Material. The Standard Assets contain the most common physics materials. To use one of them click on the Physics Material drop-down and select one, eg. Ice. You can also create your own physics materials and tweak all friction values.

## Compound Colliders

Compound Colliders are combinations of primitive Colliders, collectively acting as a single Collider. They come in handy when you have a complex mesh to use in collisions but cannot use a **Mesh Collider**. To create a Compound Collider, create child objects of your colliding object, then add a primitive Collider to each child object. This allows you to position, rotate, and scale each Collider easily and independently of one another.




*A real-world Compound Collider setup*

In the above picture, the *Gun Model* GameObject has a Rigidbody attached, and multiple primitive Colliders as child GameObjects. When the Rigidbody parent is moved around by forces, the child Colliders move along with it. The primitive Colliders will collide with the environment's Mesh Collider, and the parent Rigidbody will alter the way it moves based on forces being applied to it and how its child Colliders interact with other Colliders in the Scene.

Mesh Colliders can't normally collide with each other. If a Mesh Collider is marked as **Convex**, then it can collide with another Mesh Collider. The typical solution is to use primitive Colliders for any objects that move, and Mesh Colliders for static background objects.

## Hints

- To add multiple Colliders for an object, create child GameObjects and attach a Collider to each one. This allows each Collider to be manipulated independently.
- You can look at the gizmos in the **Scene View** to see how the Collider is being calculated on your object.
- Colliders do their best to match the scale of an object. If you have a non-uniform scale (a scale which is different in each direction), only the Mesh Collider can match completely.
- If you are moving an object through its Transform component but you want to receive Collision/Trigger messages, you must attach a Rigidbody to the object that is moving.

## Advanced

### Collider combinations

There are numerous different combinations of collisions that can happen in Unity. Each game is unique, and different combinations may work better for different types of games. If you're using physics in your game, it will be very helpful to understand the different basic Collider types, their common uses, and how they interact with other types of objects.

### Static Collider

These are GameObjects that do **not** have a Rigidbody attached, but **do** have a Collider attached. These objects should remain still, or move very little. These work great for your environment geometry. They will not move if a Rigidbody collides with them.

### Rigidbody Collider

These GameObjects contain both a Rigidbody and a Collider. They are completely affected by the physics engine through scripted forces and collisions. They might collide with a GameObject that only contains a Collider. These will likely be your primary type of Collider in games that use physics.

### Kinematic Rigidbody Collider

This GameObject contains a Collider and a Rigidbody which is marked **IsKinematic**. To move this GameObject, you modify its Transform Component, rather than applying forces. They're similar to Static Colliders but will work better when you want to move the Collider around frequently. There are some other specialized scenarios for using this GameObject.

This object can be used for circumstances in which you would normally want a Static Collider to send a trigger event. Since a

Trigger must have a Rigidbody attached, you should add a Rigidbody, then enable **IsKinematic**. This will prevent your Object from moving from physics influence, and allow you to receive trigger events when you want to.

Kinematic Rigidbodies can easily be turned on and off. This is great for creating ragdolls, when you normally want a character to follow an animation, then turn into a ragdoll when a collision occurs, prompted by an explosion or anything else you choose. When this happens, simply turn all your Kinematic Rigidbodies into normal Rigidbodies through scripting.

If you have Rigidbodies come to rest so they are not moving for some time, they will "fall asleep". That is, they will not be calculated during the physics update since they are not going anywhere. If you move a Kinematic Rigidbody out from underneath normal Rigidbodies that are at rest on top of it, the sleeping Rigidbodies will "wake up" and be correctly calculated again in the physics update. So if you have a lot of Static Colliders that you want to move around and have different object fall on them correctly, use Kinematic Rigidbody Colliders.

**Collision action matrix**

Depending on the configurations of the two colliding Objects, a number of different actions can occur. The chart below outlines what you can expect from two colliding Objects, based on the components that are attached to them. Some of the combinations only cause one of the two Objects to be affected by the collision, so keep the standard rule in mind - physics will not be applied to objects that do not have Rigidbodies attached.

| Collision detection occurs and messages are sent upon collision | | | | | | |
|---|---|---|---|---|---|---|
| | Static Collider | Rigidbody Collider | Kinematic Rigidbody Collider | Static Trigger Collider | Rigidbody Trigger Collider | Kinematic Rigidbody Trigger Collider |
| Static Collider | | Y | | | | |
| Rigidbody Collider | Y | Y | Y | | | |
| Kinematic Rigidbody Collider | | Y | | | | |
| Static Trigger Collider | | | | | | |
| Rigidbody Trigger Collider | | | | | | |
| Kinematic Rigidbody Trigger Collider | | | | | | |

| Trigger messages are sent upon collision | | | | | | |
|---|---|---|---|---|---|---|
| | Static Collider | Rigidbody Collider | Kinematic Rigidbody Collider | Static Trigger Collider | Rigidbody Trigger Collider | Kinematic Rigidbody Trigger Collider |
| Static Collider | | | | | Y | Y |
| Rigidbody Collider | | | | Y | Y | Y |
| Kinematic Rigidbody Collider | | | | Y | Y | Y |
| Static Trigger Collider | | Y | Y | | Y | Y |
| Rigidbody Trigger Collider | Y | Y | Y | Y | Y | Y |
| Kinematic Rigidbody Trigger Collider | Y | Y | Y | Y | Y | Y |

### Layer-Based Collision Detection

In Unity 3.x we introduce something called **Layer-Based Collision Detection**, and you can now selectively tell Unity GameObjects to collide with specific layers they are attached to. For more information click here

Page last updated: 2012-11-30

---

# class-CharacterController

The **Character Controller** is mainly used for third-person or first-person player control that does not make use of **Rigidbody** physics.

*The Character Controller*

## Properties

| | |
|---|---|
| **Height** | The Character's **Capsule Collider** height. Changing this will scale the collider along the Y axis in both positive and negative directions. |
| **Radius** | Length of the Capsule Collider's radius. This is essentially the width of the collider. |
| **Slope Limit** | Limits the collider to only climb slopes that are equal to or less than the indicated value. |
| **Step Offset** | The character will step up a stair only if it is closer to the ground than the indicated value. |
| **Min Move Distance** | If the character tries to move below the indicated value, it will not move at all. This can be used to reduce jitter. In most situations this value should be left at 0. |
| **Skin width** | Two colliders can penetrate each other as deep as their Skin Width. Larger Skin Widths reduce jitter. Low Skin Width can cause the character to get stuck. A good setting is to make this value 10% of the Radius. |
| **Center** | This will offset the Capsule Collider in world space, and won't affect how the Character pivots. |

## Details

The traditional Doom-style first person controls are not physically realistic. The character runs 90 miles per hour, comes to a halt immediately and turns on a dime. Because it is so unrealistic, use of Rigidbodies and physics to create this behavior is impractical and will feel wrong. The solution is the specialized Character Controller. It is simply a capsule shaped **Collider** which can be told to move in some direction from a script. The Controller will then carry out the movement but be constrained by collisions. It will slide along walls, walk up stairs (if they are lower than the **Step Offset**) and walk on slopes within the **Slope Limit**.

The Controller does not react to forces on its own and it does not automatically push Rigidbodies away.

If you want to push Rigidbodies or objects with the Character Controller, you can apply forces to any object that it collides with via the **OnControllerColliderHit()** function through scripting.

On the other hand, if you want your player character to be affected by physics then you might be better off using a Rigidbody instead of the Character Controller.

### Fine-tuning your character

You can modify the **Height** and **Radius** to fit your Character's mesh. It is recommended to always use around 2 meters for a human-like character. You can also modify the **Center** of the capsule in case your pivot point is not at the exact center of the Character.

**Step Offset** can affect this too, make sure that this value is between 0.1 and 0.4 for a 2 meter sized human.

**Slope Limit** should not be too small. Often using a value of 90 degrees works best. The Character Controller will not be able to climb up walls due to the capsule shape.

### Don't get stuck

The **Skin Width** is one of the most critical properties to get right when tuning your Character Controller. If your character gets stuck it is most likely because your **Skin Width** is too small. The **Skin Width** will let objects slightly penetrate the Controller but it removes jitter and prevents it from getting stuck.

It's good practice to keep your **Skin Width** at least greater than 0.01 and more than 10% of the **Radius**.

We recommend keeping **Min Move Distance** at 0.

See the Character Controller script reference here

You can download an example project showing pre-setup animated and moving character controllers from the Resources area on our website.

### Hints

- Try adjusting your **Skin Width** if you find your character getting stuck frequently.
- The Character Controller can affect objects using physics if you write your own scripts.
- The Character Controller can not be affected by objects through physics.
- Note that changing Character Controller properties in the inspector will recreate the controller in the scene, so any existing Trigger contacts will get lost, and you will not get any OnTriggerEntered messages until the controller is moved again.

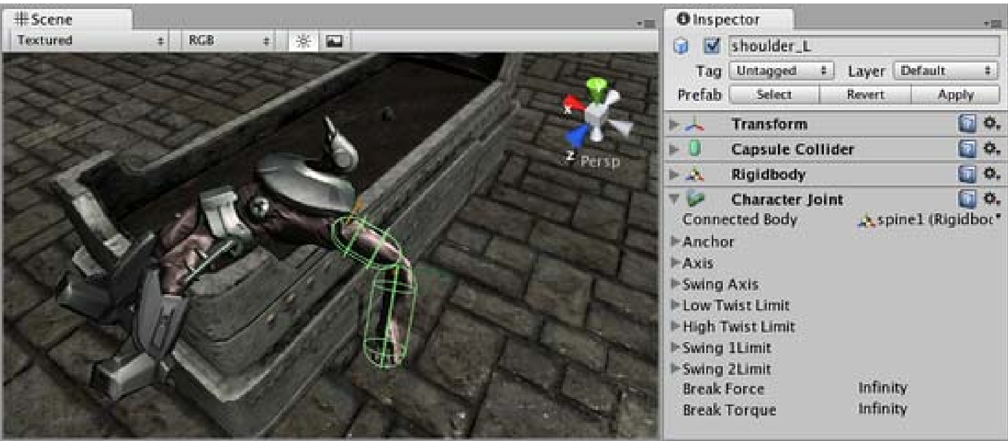
Page last updated: 2012-06-18

---

# class-CharacterJoint

**Character Joints** are mainly used for Ragdoll effects. They are an extended ball-socket joint which allows you to limit the joint on each axis.

If you just want to set up a ragdoll read about Ragdoll Wizard.




*The Character Joint on a Ragdoll*

### Properties

| | |
|---|---|
| **Connected Body** | Optional reference to the **Rigidbody** that the joint is dependent upon. If not set, the joint connects to the world. |
| **Anchor** | The point in the **GameObject's** local space where the joint rotates around. |
| **Axis** | The twist axes. Visualized with the orange gizmo cone. |
| **Swing Axis** | The swing axis. Visualized with the green gizmo cone. |
| **Low Twist Limit** | The lower limit of the joint. |
| **High Twist Limit** | The higher limit of the joint. |
| **Swing 1 Limit** | Lower limit around the defined **Swing Axis** |
| **Swing 2 Limit** | Upper limit around the defined **Swing Axis** |
| **Break Force** | The force that needs to be applied for this joint to break. |
| **Break Torque** | The torque that needs to be applied for this joint to break. |

### Details

Character joint's give you a lot of possibilities for constraining motion like with a universal joint.

The twist axis (visualized with the orange gizmo) gives you most control over the limits as you can specify a lower and upper limit in degrees (the limit angle is measured relative to the starting position). A value of -30 in **Low Twist Limit**->**Limit** and 60 in **High Twist Limit**->**Limit** limits the rotation around the twist axis (orange gizmo) between -30 and 60 degrees.

The **Swing 1 Limit** limits the rotation around the swing axis (green axis). The limit angle is symmetric. Thus a value of eg. 30 will limit the rotation between -30 and 30.

The **Swing 2 Limit** axis doesn't have a gizmo but the axis is orthogonal to the 2 other axes. Just like the previous axis the limit is symmetric, thus a value of eg. 40 will limit the rotation around that axis between -40 and 40 degrees.

**Breaking joints**

You can use the **Break Force** and **Break Torque** properties to set limits for the joint's strength. If these are less than infinity, and a force/torque greater than these limits are applied to the object, its Fixed Joint will be destroyed and will no longer be confined by its restraints.

### Hints

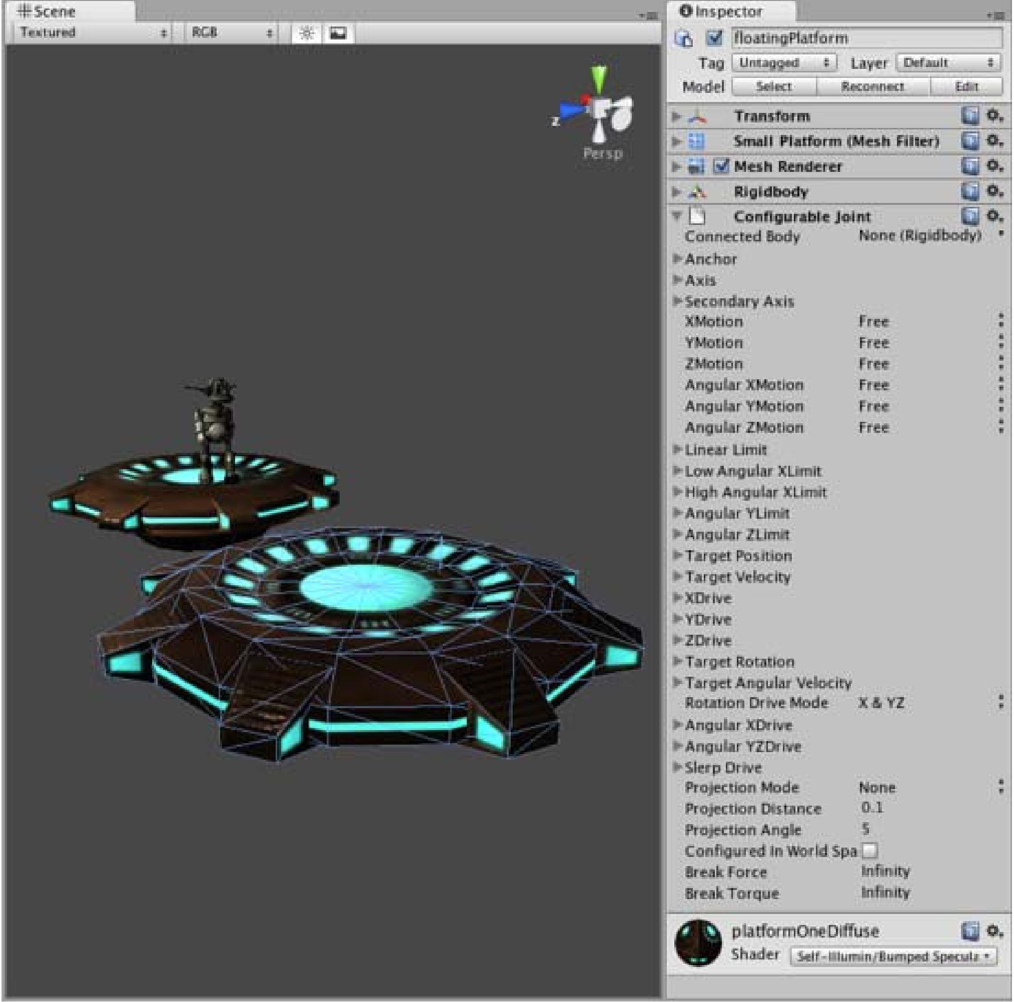
- You do not need to assign a **Connected Body** to your joint for it to work.
- Character Joints require your object to have a Rigidbody attached.

Page last updated: 2007-09-12

---

## class-ConfigurableJoint

**Configurable Joints** are extremely customizable. They expose all joint-related properties of PhysX, so they are capable of creating behaviors similar to all other joint types.




*Properties of the Configurable Joint*

### Details

There are two primary functions that the Configurable Joint can perform: movement/rotation restriction and movement/rotation acceleration. These functions depend on a number of inter-dependent properties. It may require some experimentation to

create the exact behavior you're trying to achieve. We'll now give you an overview of the Joint's functionality to make your experimentation as simple as possible.

## Movement/Rotation Restriction

You specify restriction per axis and per motion type. **XMotion**, **YMotion**, and **ZMotion** allow you to define translation along that axis. **Angular XMotion**, **Angular YMotion**, and **Angular ZMotion** allow you to define rotation around that axis. Each one of these properties can be set to **Free** (unrestricted), **Limited** (restricted based on limits you can define), or **Locked** (restricted to zero movement).

### Limiting Motion

When you have any of the **"Motion"** properties set to **Limited**, you can define the limitations of movement for that axis. You do this by changing the values of one of the **"Limit"** properties.

For translation of movement (non-angular), the **Linear Limit** property will define the maximum distance the object can move from its origin. Translation on any **"Motion"** properties set to **Limited** will be restricted according to **Linear Limit->Limit**. Think of this **Limit** property as setting a border around the axis for the object.

**Bouncyness**, **Spring**, and **Damper** will define the behavior of the object when it reaches the **Limit** on any of the **Limited "Motion"** axes. If all of these values are set to 0, the object will instantly stop moving when it reaches the border. **Bouncyness** will make the object bounce back away from the border. **Spring** and **Damper** will use springing forces to pull the object back to the border. This will soften the border, so the object will be able to pass through the border and be pulled back instead of stopping immediately.

### Limiting Rotation

Limiting rotation works almost the same as limiting motion. The difference is that the three **"Angular Motion"** properties all correspond to different **"Angular Limit"** properties. Translation restriction along all 3 axes are defined by the **Linear Limit** property, and rotation restriction along each of the 3 axes is defined by a separate **"Angular Limit"** property per axis.

**Angular XMotion** limitation is the most robust, as you can define a **Low Angular XLimit** and a **High Angular XLimit**. Therefore if you want to define a low rotation limit of -35 degrees and a high rotation limit of 180 degrees, you can do this. For the Y and Z axes, the low and high rotation limits will be identical, set together by the **Limit** property of **Angular YLimit** or **Angular ZLimit**.

The same rules about object behavior at the rotation limits from the Limiting Motion section applies here.

## Movement/Rotation Acceleration

You specify object movement or rotation in terms of moving the object toward a particular position/rotation, or velocity/angular velocity. This system works by defining the **"Target"** value you want to move toward, and using a **"Drive"** to provide acceleration which will move the object toward that target. Each **"Drive"** has a **Mode**, which you use to define which **"Target"** the object is moving toward.

### Translation Acceleration

The **XDrive**, **YDrive**, and **ZDrive** properties are what start the object moving along that axis. Each **Drive**'s **Mode** will define whether the object should be moving toward the **Target Position** or **Target Velocity** or both. For example, when **XDrive**'s mode is set to **Position**, then the object will try to move to the value of **Target Position->X**.

When a **Drive** is using **Position** in its **Mode**, its **Position Spring** value will define how the object is moved toward the **Target Position**. Similarly, when a **Drive** is using **Velocity** in its **Mode**, its **Maximum Force** value will define how the object is accelerated to the **Target Velocity**.

### Rotation Acceleration

Rotation acceleration properties: **Angular XDrive**, **Angular YZDrive**, and **Slerp Drive** function the same way as the translation **Drives**. There is one substantial difference. **Slerp Drive** behaves differently from the **Angular Drive** functionality. Therefore you can choose to use either both **Angular Drives** or **Slerp Drive** by choosing one from the **Rotation Drive Mode**. You cannot use both at once.

## Properties

| | |
|---|---|
| **Anchor** | The point where the center of the joint is defined. All physics-based simulation will use this point as the center in calculations |
| **Axis** | The local axis that will define the object's natural rotation based on physics simulation |

| | |
|---|---|
| **Secondary Axis** | Together, **Axis** and **Secondary Axis** define the local coordinate system of the joint. The third axis is set to be orthogonal to the other two. |
| **XMotion** | Allow movement along the X axis to be Free, completely Locked, or Limited according to **Linear Limit** |
| **YMotion** | Allow movement along the Y axis to be Free, completely Locked, or Limited according to **Linear Limit** |
| **ZMotion** | Allow movement along the Z axis to be Free, completely Locked, or Limited according to **Linear Limit** |
| **Angular XMotion** | Allow rotation around the X axis to be Free, completely Locked, or Limited according to **Low** and **High Angular XLimit** |
| **Angular YMotion** | Allow rotation around the Y axis to be Free, completely Locked, or Limited according to **Angular YLimit** |
| **Angular ZMotion** | Allow rotation around the Z axis to be Free, completely Locked, or Limited according to **Angular ZLimit** |
| **Linear Limit** | Boundary defining movement restriction, based on distance from the joint's origin |
| **Limit** | The distance in units from the origin to the wall of the boundary |
| **Bouncyness** | Amount of bounce-back force applied to the object when it reaches the **Limit** |
| **Spring** | Strength of force applied to move the object back to the **Limit**. Any value other than 0 will implicitly soften the boundary |
| **Damper** | Resistance strength against the **Spring** |
| **Low Angular XLimit** | Boundary defining lower rotation restriction, based on delta from original rotation |
| **Limit** | The rotation in degrees that the object's rotation should not drop below |
| **Bouncyness** | Amount of bounce-back torque applied to the object when its rotation reaches the **Limit** |
| **Spring** | Strength of force applied to move the object back to the **Limit**. Any value other than 0 will implicitly soften the boundary |
| **Damper** | Resistance strength against the **Spring** |
| **High Angular XLimit** | Boundary defining upper rotation restriction, based on delta from original rotation. |
| **Limit** | The rotation in degrees that the object's rotation should not exceed |
| **Bouncyness** | Amount of bounce-back torque applied to the object when its rotation reaches the **Limit** |
| **Spring** | Strength of force applied to move the object back to the **Limit**. Any value other than 0 will implicitly soften the boundary |
| **Damper** | Resistance strength against the **Spring** |
| **Angular YLimit** | Boundary defining rotation restriction, based on delta from original rotation |
| **Limit** | The rotation in degrees that the object's rotation should not exceed |
| **Bouncyness** | Amount of bounce-back torque applied to the object when its rotation reaches the **Limit** |
| **Spring** | Strength of torque applied to move the object back to the **Limit**. Any value other than 0 will implicitly soften the boundary |
| **Damper** | Resistance strength against the **Spring** |
| **Angular ZLimit** | Boundary defining rotation restriction, based on delta from original rotation |
| **Limit** | The rotation in degrees that the object's rotation should not exceed |
| **Bouncyness** | Amount of bounce-back torque applied to the object when its rotation reaches the **Limit** |
| **Spring** | Strength of force applied to move the object back to the **Limit**. Any value other than 0 will implicitly soften the boundary |
| **Damper** | Resistance strength against the **Spring** |
| **Target Position** | The desired position that the joint should move into |
| **Target Velocity** | The desired velocity that the joint should move along |
| **XDrive** | Definition of how the joint's movement will behave along its local X axis |
| **Mode** | Set the following properties to be dependent on **Target Position**, **Target Velocity**, or both |
| **Position Spring** | Strength of a rubber-band pull toward the defined direction. Only used if **Mode** includes **Position** |
| **Position Damper** | Resistance strength against the **Position Spring**. Only used if **Mode** includes **Position** |
| **Maximum Force** | Amount of strength applied to push the object toward the defined direction. Only used if **Mode** includes **Velocity** |
| **YDrive** | Definition of how the joint's movement will behave along its local Y axis |
| **Mode** | Set the following properties to be dependent on **Target Position**, **Target Velocity**, or both |
| **Position Spring** | Strength of a rubber-band pull toward the defined direction. Only used if **Mode** includes **Position**. |
| **Position Damper** | Resistance strength against the **Position Spring**. Only used if **Mode** includes **Position**. |
| **Maximum Force** | Amount of strength applied to push the object toward the defined direction. Only used if **Mode** includes **Velocity**. |
| **ZDrive** | Definition of how the joint's movement will behave along its local Z axis |
| **Mode** | Set the following properties to be dependent on **Target Position**, **Target Velocity**, or both. |
| **Position Spring** | Strength of a rubber-band pull toward the defined direction. Only used if **Mode** includes **Position** |

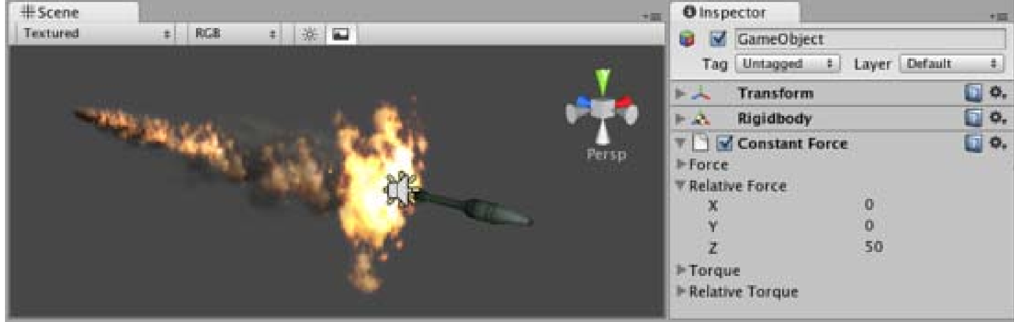
| | |
|---|---|
| **Position Damper** | Resistance strength against the **Position Spring**. Only used if **Mode** includes **Position** |
| **Maximum Force** | Amount of strength applied to push the object toward the defined direction. Only used if **Mode** includes **Velocity** |
| **Target Rotation** | This is a Quaternion. It defines the desired rotation that the joint should rotate into |
| **Target Angular Velocity** | This is a Vector3. It defines the desired angular velocity that the joint should rotate into |
| **Rotation Drive Mode** | Control the object's rotation with either **X & YZ** or **Slerp Drive** by itself |
| **Angular XDrive** | Definition of how the joint's rotation will behave around its local X axis. Only used if **Rotation Drive Mode** is **Swing & Twist** |
| **Mode** | Set the following properties to be dependent on **Target Rotation**, **Target Angular Velocity**, or both |
| **Position Spring** | Strength of a rubber-band pull toward the defined direction. Only used if **Mode** includes **Position** |
| **Position Damper** | Resistance strength against the **Position Spring**. Only used if **Mode** includes **Position** |
| **Maximum Force** | Amount of strength applied to push the object toward the defined direction. Only used if **Mode** includes **Velocity**. |
| **Angular YZDrive** | Definition of how the joint's rotation will behave around its local Y and Z axes. Only used if **Rotation Drive Mode** is **Swing & Twist** |
| **Mode** | Set the following properties to be dependent on **Target Rotation**, **Target Angular Velocity**, or both |
| **Position Spring** | Strength of a rubber-band pull toward the defined direction. Only used if **Mode** includes **Position** |
| **Position Damper** | Resistance strength against the **Position Spring**. Only used if **Mode** includes **Position** |
| **Maximum Force** | Amount of strength applied to push the object toward the defined direction. Only used if **Mode** includes **Velocity** |
| **Slerp Drive** | Definition of how the joint's rotation will behave around all local axes. Only used if **Rotation Drive Mode** is **Slerp Only** |
| **Mode** | Set the following properties to be dependent on **Target Rotation**, **Target Angular Velocity**, or both |
| **Position Spring** | Strength of a rubber-band pull toward the defined direction. Only used if **Mode** includes **Position** |
| **Position Damper** | Resistance strength against the **Position Spring**. Only used if **Mode** includes **Position** |
| **Maximum Force** | Amount of strength applied to push the object toward the defined direction. Only used if **Mode** includes **Velocity** |
| **Projection Mode** | Properties to track to snap the object back to its constrained position when it drifts off too much |
| **Projection Distance** | Distance from the **Connected Body** that must be exceeded before the object snaps back to an acceptable position |
| **Projection Angle** | Difference in angle from the **Connected Body** that must be exceeded before the object snaps back to an acceptable position |
| **Congfigure in World Space** | If enabled, all Target values will be calculated in World Space instead of the object's Local Space |
| **Break Force** | Applied Force values above this number will cause the joint to be destroyed |
| **Break Torque** | Applied Torque values above this number will cause the joint to be destroyed |

Page last updated: 2010-08-24

---

# class-ConstantForce

**Constant Force** is a quick utility for adding constant forces to a **Rigidbody**. This works great for one shot objects like rockets, if you don't want it to start with a large velocity but instead accelerate.




*A rocket propelled forward by a Constant Force*

## Properties

| | |
|---|---|
| **Force** | The vector of a force to be applied in world space. |

| | |
|---|---|
| **Relative Force** | The vector of a force to be applied in the object's local space. |
| **Torque** | The vector of a torque, applied in world space. The object will begin spinning *around* this vector. The longer the vector is, the faster the rotation. |
| **Relative Torque** | The vector of a torque, applied in local space. The object will begin spinning *around* this vector. The longer the vector is, the faster the rotation. |

#### Details

To make a rocket that accelerates forward set the **Relative Force** to be along the positive z-axis. Then use the Rigidbody's **Drag** property to make it not exceed some maximum velocity (the higher the drag the lower the maximum velocity will be). In the Rigidbody, also make sure to turn off gravity so that the rocket will always stay on its path.

### Hints

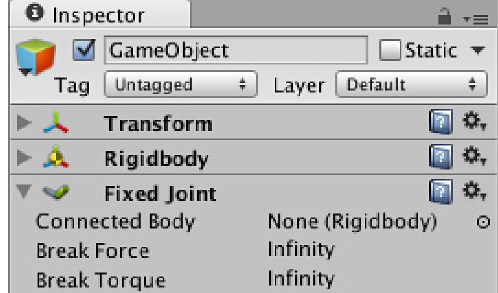
- To make an object flow upwards, add a Constant Force with the **Force** property having a positive Y value.
- To make an object fly forwards, add a Constant Force with the **Relative Force** property having a positive Z value.

Page last updated: 2007-10-02

---

## class-FixedJoint

**Fixed Joints** restricts an object's movement to be dependent upon another object. This is somewhat similar to **Parenting** but is implemented through physics rather than **Transform** hierarchy. The best scenarios for using them are when you have objects that you want to easily break apart from each other, or connect two object's movement without parenting.




*The Fixed Joint **Inspector***

### Properties

| | |
|---|---|
| **Connected Body** | Optional reference to the Rigidbody that the joint is dependent upon. If not set, the joint connects to the world. |
| **Break Force** | The force that needs to be applied for this joint to break. |
| **Break Torque** | The torque that needs to be applied for this joint to break. |

### Details

There may be scenarios in your game where you want objects to stick together permanently or temporarily. Fixed Joints may be a good **Component** to use for these scenarios, since you will not have to script a change in your object's hierarchy to achieve the desired effect. The trade-off is that you must use **Rigidbodies** for any objects that use a Fixed Joint.

For example, if you want to use a "sticky grenade", you can write a script that will detect collision with another Rigidbody (like an enemy), and then create a Fixed Joint that will attach itself to that Rigidbody. Then as the enemy moves around, the joint will keep the grenade stuck to them.

#### Breaking joints

You can use the **Break Force** and **Break Torque** properties to set limits for the joint's strength. If these are less than infinity, and a force/torque greater than these limits are applied to the object, its Fixed Joint will be destroyed and will no longer be confined by its restraints.

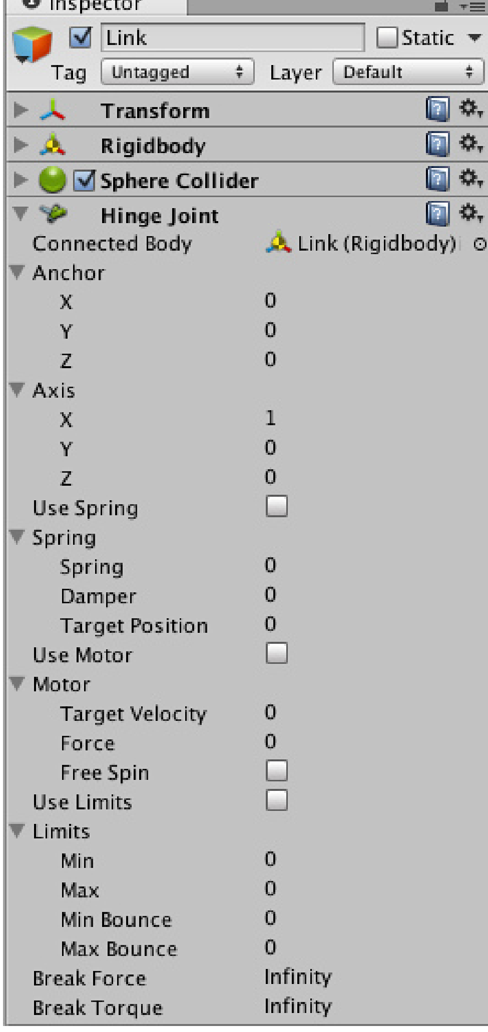
### Hints

- You do not need to assign a **Connected Body** to your joint for it to work.
- Fixed Joints require a Rigidbody.

Page last updated: 2007-09-14

## class-HingeJoint

The **Hinge Joint** groups together two Rigidbodies, constraining them to move like they are connected by a hinge. It is perfect for doors, but can also be used to model chains, pendulums, etc.



*The Hinge Joint **Inspector***

### Properties

| | |
|---|---|
| **Connected Body** | Optional reference to the Rigidbody that the joint is dependent upon. If not set, the joint connects to the world. |
| **Anchor** | The position of the axis around which the body swings. The position is defined in local space. |
| **Axis** | The direction of the axis around which the body swings. The direction is defined in local space. |
| **Use Spring** | Spring makes the Rigidbody reach for a specific angle compared to its connected body. |
| **Spring** | Properties of the Spring that are used if **Use Spring** is enabled. |
| **Spring** | The force the object asserts to move into the position. |
| **Damper** | The higher this value, the more the object will slow down. |
| **Target Position** | Target angle of the spring. The spring pulls towards this angle measured in degrees. |
| **Use Motor** | The motor makes the object spin around. |
| **Motor** | Properties of the Motor that are used if **Use Motor** is enabled. |
| **Target Velocity** | The speed the object tries to attain. |
| **Force** | The force applied in order to attain the speed. |
| **Free Spin** | If enabled, the motor is never used to brake the spinning, only accelerate it. |
| **Use Limits** | If enabled, the angle of the hinge will be restricted within the **Min** & **Max** values. |
| **Limits** | Properties of the Limits that are used if **Use Limits** is enabled. |
| **Min** | The lowest angle the rotation can go. |

| | |
|---|---|
| **Max** | The highest angle the rotation can go. |
| **Min Bounce** | How much the object bounces when it hits the minimum stop. |
| **Max Bounce** | How much the object bounces when it hits the maximum stop. |
| **Break Force** | The force that needs to be applied for this joint to break. |
| **Break Torque** | The torque that needs to be applied for this joint to break. |

## Details

A single Hinge Joint should be applied to a **GameObject**. The hinge will rotate at the point specified by the **Anchor** property, moving around the specified **Axis** property. You **do not** need to assign a GameObject to the joint's **Connected Body** property. You should only assign a GameObject to the **Connected Body** property if you want the joint's **Transform** to be dependent on the attached object's Transform.

Think about how the hinge of a door works. The **Axis** in this case is up, positive along the Y axis. The **Anchor** is placed somewhere at the intersection between door and wall. You would not need to assign the wall to the **Connected Body**, because the joint will be connected to the world by default.

Now think about a doggy door hinge. The doggy door's **Axis** would be sideways, positive along the relative X axis. The main door should be assigned as the **Connected Body**, so the doggy door's hinge is dependent on the main door's Rigidbody.

### Chains

Multiple Hinge Joints can also be strung together to create a chain. Add a joint to each link in the chain, and attach the next link as the **Connected Body**.

## Hints

- You do not need to assign a **Connected Body** to your joint for it to work.
- Use **Break Force** in order to make dynamic damage systems. This is really cool as it allows the player to break a door off its hinge by blasting it with a rocket launcher or running into it with a car.
- The **Spring**, **Motor**, and **Limits** properties allow you to fine-tune your joint's behaviors.

Page last updated: 2007-09-14

---

# class-MeshCollider

The **Mesh Collider** takes a Mesh Asset and builds its Collider based on that mesh. It is far more accurate for collision detection than using primitives for complicated meshes. Mesh Colliders that are marked as **Convex** can collide with other Mesh Colliders.

*A Mesh Collider used on level geometry*

## Properties

| | |
|---|---|
| **Is Trigger** | If enabled, this Collider is used for triggering events, and is ignored by the physics engine. |
| **Material** | Reference to the Physics Material that determines how this Collider interacts with others. |
| **Mesh** | Reference to the Mesh to use for collisions. |
| **Smooth Sphere Collisions** | When this is enabled, collision mesh normals are smoothed. You should enable this on smooth surfaces eg. rolling terrain without hard edges to make sphere rolling smoother. |

| | |
|---|---|
| **Convex** | If enabled, this Mesh Collider will collide with other Mesh Colliders. Convex Mesh Colliders are limited to 255 triangles. |

## Details

The Mesh Collider builds its collision representation from the Mesh attached to the GameObject, and reads the properties of the attached Transform to set its position and scale correctly.

Collision meshes use backface culling. If an object collides with a mesh that will be backface culled graphically it will also not collide with it physically.

There are some limitations when using the Mesh Collider. Usually, two Mesh Colliders cannot collide with each other. All Mesh Colliders can collide with any primitive Collider. If your mesh is marked as **Convex**, then it can collide with other Mesh Colliders.

Colliders work with Rigidbodies to bring physics in Unity to life. Whereas Rigidbodies allow objects to be controlled by physics, Colliders allow objects to collide with each other. Colliders must be added to objects independently of Rigidbodies. A Collider does not necessarily need a Rigidbody attached, but a Rigidbody **must** be attached in order for the object to move as a result of collisions.

When a collision between two Colliders occurs and if at least one of them has a Rigidbody attached, three collision messages are sent out to the objects attached to them. These events can be handled in scripting, and allow you to create unique behaviors with or without making use of the built-in NVIDIA PhysX engine.

### Triggers

An alternative way of using Colliders is to mark them as a **Trigger**, just check the IsTrigger property checkbox in the Inspector. Triggers are effectively ignored by the physics engine, and have a unique set of three trigger messages that are sent out when a collision with a Trigger occurs. Triggers are useful for triggering other events in your game, like cutscenes, automatic door opening, displaying tutorial messages, etc. Use your imagination!

Be aware that in order for two Triggers to send out trigger events when they collide, one of them must include a Rigidbody as well. For a Trigger to collide with a normal Collider, one of them must have a Rigidbody attached. For a detailed chart of different types of collisions, see the collision action matrix in the Advanced section below.

### Friction and bounciness

Friction, bounciness and softness are defined in the Physisc Material. The Standard Assets contain the most common physics materials. To use one of them click on the Physics Material drop-down and select one, eg. Ice. You can also create your own physics materials and tweak all friction values.

## Hints

- Mesh Colliders **cannot** collide with each other unless they are marked as **Convex**. Therefore, they are most useful for background objects like environment geometry.
- **Convex** Mesh Colliders must be fewer than 255 triangles.
- Primitive Colliders are less costly for objects under physics control.
- When you attach a Mesh Collider to a **GameObject**, its Mesh property will default to the mesh being rendered. You can change that by assigning a different Mesh.
- To add multiple Colliders for an object, create child GameObjects and attach a Collider to each one. This allows each Collider to be manipulated independently.
- You can look at the gizmos in the **Scene View** to see how the Collider is being calculated on your object.
- Colliders do their best to match the scale of an object. If you have a non-uniform scale (a scale which is different in each direction), only the Mesh Collider can match completely.
- If you are moving an object through its Transform component but you want to receive Collision/Trigger messages, you must attach a Rigidbody to the object that is moving.

## Advanced

### Collider combinations

There are numerous different combinations of collisions that can happen in Unity. Each game is unique, and different combinations may work better for different types of games. If you're using physics in your game, it will be very helpful to understand the different basic Collider types, their common uses, and how they interact with other types of objects.

### Static Collider

These are GameObjects that do **not** have a Rigidbody attached, but **do** have a Collider attached. These objects should remain still, or move very little. These work great for your environment geometry. They will not move if a Rigidbody collides with them.

**Rigidbody Collider**
These GameObjects contain both a Rigidbody and a Collider. They are completely affected by the physics engine through scripted forces and collisions. They might collide with a GameObject that only contains a Collider. These will likely be your primary type of Collider in games that use physics.

**Kinematic Rigidbody Collider**
This GameObject contains a Collider and a Rigidbody which is marked **IsKinematic**. To move this GameObject, you modify its Transform Component, rather than applying forces. They're similar to Static Colliders but will work better when you want to move the Collider around frequently. There are some other specialized scenarios for using this GameObject.

This object can be used for circumstances in which you would normally want a Static Collider to send a trigger event. Since a Trigger must have a Rigidbody attached, you should add a Rigidbody, then enable **IsKinematic**. This will prevent your Object from moving from physics influence, and allow you to receive trigger events when you want to.

Kinematic Rigidbodies can easily be turned on and off. This is great for creating ragdolls, when you normally want a character to follow an animation, then turn into a ragdoll when a collision occurs, prompted by an explosion or anything else you choose. When this happens, simply turn all your Kinematic Rigidbodies into normal Rigidbodies through scripting.

If you have Rigidbodies come to rest so they are not moving for some time, they will "fall asleep". That is, they will not be calculated during the physics update since they are not going anywhere. If you move a Kinematic Rigidbody out from underneath normal Rigidbodies that are at rest on top of it, the sleeping Rigidbodies will "wake up" and be correctly calculated again in the physics update. So if you have a lot of Static Colliders that you want to move around and have different object fall on them correctly, use Kinematic Rigidbody Colliders.

**Collision action matrix**
Depending on the configurations of the two colliding Objects, a number of different actions can occur. The chart below outlines what you can expect from two colliding Objects, based on the components that are attached to them. Some of the combinations only cause one of the two Objects to be affected by the collision, so keep the standard rule in mind - physics will not be applied to objects that do not have Rigidbodies attached.

| **Collision detection occurs and messages are sent upon collision** | | | | | | |
|---|---|---|---|---|---|---|
| | Static Collider | Rigidbody Collider | Kinematic Rigidbody Collider | Static Trigger Collider | Rigidbody Trigger Collider | Kinematic Rigidbody Trigger Collider |
| Static Collider | | Y | | | | |
| Rigidbody Collider | Y | Y | Y | | | |
| Kinematic Rigidbody Collider | | Y | | | | |
| Static Trigger Collider | | | | | | |
| Rigidbody Trigger Collider | | | | | | |
| Kinematic Rigidbody Trigger Collider | | | | | | |

| **Trigger messages are sent upon collision** | | | | | | |
|---|---|---|---|---|---|---|
| | Static Collider | Rigidbody Collider | Kinematic Rigidbody Collider | Static Trigger Collider | Rigidbody Trigger Collider | Kinematic Rigidbody Trigger Collider |
| Static Collider | | | | | Y | Y |
| Rigidbody Collider | | | | Y | Y | Y |
| Kinematic Rigidbody Collider | | | | Y | Y | Y |
| Static Trigger Collider | | Y | Y | | Y | Y |
| Rigidbody Trigger Collider | Y | Y | Y | Y | Y | Y |
| Kinematic Rigidbody Trigger Collider | Y | Y | Y | Y | Y | Y |

### Layer-Based Collision Detection
In Unity 3.x we introduce something called **Layer-Based Collision Detection**, and you can now selectively tell Unity

GameObjects to collide with specific layers they are attached to. For more information click here

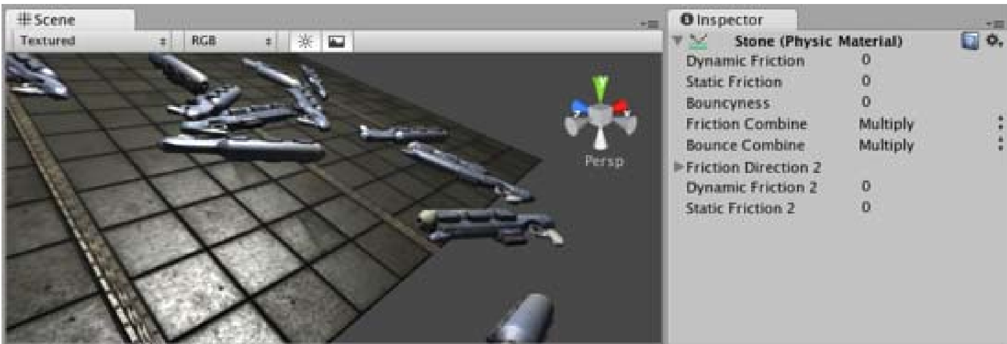
Page last updated: 2009-07-16

---

# class-PhysicMaterial

The **Physics Material** is used to adjust friction and bouncing effects of colliding objects.

To create a Physics Material select **Assets->Create->Physics Material** from the menu bar. Then drag the Physics Material from the Project View onto a **Collider** in the scene.



*The Physics Material* ***Inspector***

## Properties

| | |
|---|---|
| **Dynamic Friction** | The friction used when already moving. Usually a value from 0 to 1. A value of zero feels like ice, a value of 1 will make it come to rest very quickly unless a lot of force or gravity pushes the object. |
| **Static Friction** | The friction used when an object is laying still on a surface. Usually a value from 0 to 1. A value of zero feels like ice, a value of 1 will make it very hard to get the object moving. |
| **Bounciness** | How bouncy is the surface? A value of 0 will not bounce. A value of 1 will bounce without any loss of energy. |
| **Friction Combine Mode** | How the friction of two colliding objects is combined. |
| **Average** | The two friction values are averaged. |
| **Min** | The smallest of the two values is used. |
| **Max** | The largest of the two values is used. |
| **Multiply** | The friction values are multiplied with each other. |
| **Bounce Combine** | How the bounciness of two colliding objects is combined. It has the same modes as Friction Combine Mode |
| **Friction Direction 2** | The direction of anisotropy. Anisotropic friction is enabled if this direction is not zero. Dynamic Friction 2 and Static Friction 2 will be applied along Friction Direction 2. |
| **Dynamic Friction 2** | If anisotropic friction is enabled, DynamicFriction2 will be applied along Friction Direction 2. |
| **Static Friction 2** | If anisotropic friction is enabled, StaticFriction2 will be applied along Friction Direction 2. |

## Details

Friction is the quantity which prevents surfaces from sliding off each other. This value is critical when trying to stack objects. Friction comes in two forms, dynamic and static. **Static friction** is used when the object is lying still. It will prevent the object from starting to move. If a large enough force is applied to the object it will start moving. At this point **Dynamic Friction** will come into play. **Dynamic Friction** will now attempt to slow down the object while in contact with another.

## Hints

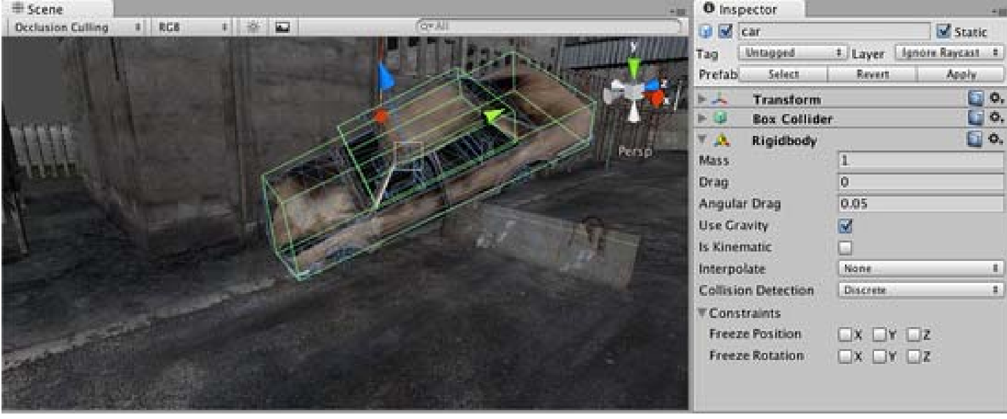
- Don't try to use a standard physics material for the main character. Make a customized one and get it perfect.

Page last updated: 2012-11-23

---

# class-Rigidbody

**Rigidbodies** enable your **GameObjects** to act under the control of physics. The Rigidbody can receive forces and torque to make your objects move in a realistic way. Any GameObject must contain a Rigidbody to be influenced by gravity, act under added forces via scripting, or interact with other objects through the NVIDIA PhysX physics engine.




*Rigidbodies allow GameObjects to act under physical influence*

## Properties

| | |
|---|---|
| **Mass** | The mass of the object (arbitrary units). It is recommended to make masses not more or less than 100 times that of other Rigidbodies. |
| **Drag** | How much air resistance affects the object when moving from forces. 0 means no air resistance, and infinity makes the object stop moving immediately. |
| **Angular Drag** | How much air resistance affects the object when rotating from torque. 0 means no air resistance, and infinity makes the object stop rotating immediately. |
| **Use Gravity** | If enabled, the object is affected by gravity. |
| **Is Kinematic** | If enabled, the object will not be driven by the physics engine, and can only be manipulated by its **Transform**. This is useful for moving platforms or if you want to animate a Rigidbody that has a **HingeJoint** attached. |
| **Interpolate** | Try one of the options only if you are seeing jerkiness in your Rigidbody's movement. |
| **None** | No Interpolation is applied. |
| **Interpolate** | Transform is smoothed based on the Transform of the previous frame. |
| **Extrapolate** | Transform is smoothed based on the estimated Transform of the next frame. |
| **Collision Detection** | Used to prevent fast moving objects from passing through other objects without detecting collisions. |
| **Discrete** | Use Discreet collision detection against all other colliders in the scene. Other colliders will use Discreet collision detection when testing for collision against it. Used for normal collisions (This is the default value). |
| **Continuous** | Use Discrete collision detection against dynamic colliders (with a rigidbody) and continuous collision detection against static MeshColliders (without a rigidbody). Rigidbodies set to Continuous Dynamic will use continuous collision detection when testing for collision against this rigidbody. Other rigidbodies will use Discreet Collision detection. Used for objects which the Continuous Dynamic detection needs to collide with. (This has a big impact on physics performance, leave it set to Discrete, if you don't have issues with collisions of fast objects) |
| **Continuous Dynamic** | Use continuous collision detection against objects set to Continuous and Continuous Dynamic Collision. It will also use continuous collision detection against static MeshColliders (without a rigidbody). For all other colliders it uses discreet collision detection. Used for fast moving objects. |
| **Constraints** | Restrictions on the Rigidbody's motion:- |
| **Freeze Position** | Stops the Rigidbody moving in the world X, Y and Z axes selectively. |
| **Freeze Rotation** | Stops the Rigidbody rotating around the world X, Y and Z axes selectively. |

## Details

Rigidbodies allow your GameObjects to act under control of the physics engine. This opens the gateway to realistic collisions, varied types of joints, and other very cool behaviors. Manipulating your GameObjects by adding forces to a Rigidbody creates a very different feel and look than adjusting the Transform **Component** directly. Generally, you shouldn't manipulate the Rigidbody and the Transform of the same GameObject - only one or the other.

The biggest difference between manipulating the Transform versus the Rigidbody is the use of forces. Rigidbodies can receive forces and torque, but Transforms cannot. Transforms can be translated and rotated, but this is not the same as using physics.

You'll notice the distinct difference when you try it for yourself. Adding forces/torque to the Rigidbody will actually change the object's position and rotation of the Transform component. This is why you should only be using one or the other. Changing the Transform while using physics could cause problems with collisions and other calculations.

Rigidbodies must be explicitly added to your GameObject before they will be affected by the physics engine. You can add a Rigidbody to your selected object from **Components->Physics->Rigidbody** in the menubar. Now your object is physics-ready; it will fall under gravity and can receive forces via scripting, but you may need to add a **Collider** or a Joint to get it to behave exactly how you want.

**Parenting**

When an object is under physics control, it moves semi-independently of the way its transform parents move. If you move any parents, they will pull the Rigidbody child along with them. However, the Rigidbodies will still fall down due to gravity and react to collision events.

**Scripting**

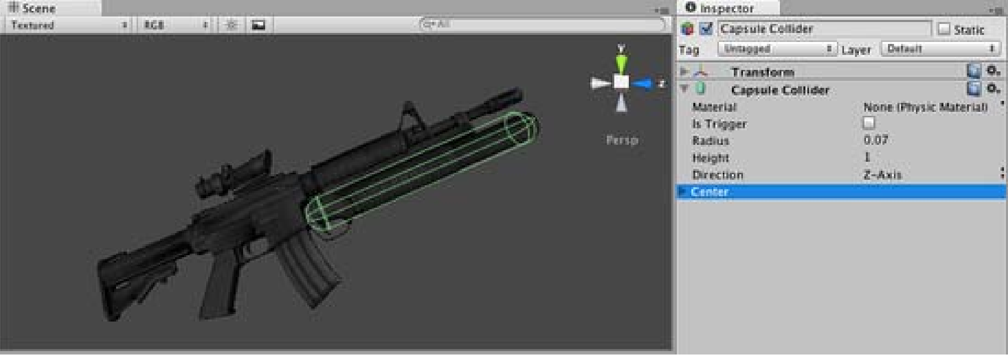
To control your Rigidbodies, you will primarily use scripts to add forces or torque. You do this by calling **AddForce()** and **AddTorque()** on the object's Rigidbody. Remember that you shouldn't be directly altering the object's Transform when you are using physics.

**Animation**

For some situations, mainly creating ragdoll effects, it is neccessary to switch control of the object between animations and physics. For this purpose Rigidbodies can be marked **isKinematic**. While the Rigidbody is marked **isKinematic**, it will not be affected by collisions, forces, or any other part of physX. This means that you will have to control the object by manipulating the Transform component directly. Kinematic Rigidbodies will affect other objects, but they themselves will not be affected by physics. For example, Joints which are attached to Kinematic objects will constrain any other Rigidbodies attached to them and Kinematic Rigidbodies will affect other Rigidbodies through collisions.

**Colliders**

Colliders are another kind of component that must be added alongside the Rigidbody in order to allow collisions to occur. If two Rigidbodies bump into each other, the physics engine will not calculate a collision unless both objects also have a Collider attached. Collider-less Rigidbodies will simply pass through each other during physics simulation.




*Colliders define the physical boundaries of a Rigidbody*

Add a Collider with the **Component->Physics** menu. View the Component Reference page of any individual Collider for more specific information:

- Box Collider - primitive shape of a cube
- Sphere Collider - primitive shape of a sphere
- Capsule Collider - primitive shape of a capsule
- Mesh Collider - creates a collider from the object's mesh, cannot collide with another Mesh Collider
- Wheel Collider - specifically for creating cars or other moving vehicles

**Compound Colliders**

Compound Colliders are combinations of primitive Colliders, collectively acting as a single Collider. They come in handy when you have a complex mesh to use in collisions but cannot use a **Mesh Collider**. To create a Compound Collider, create child objects of your colliding object, then add a primitive Collider to each child object. This allows you to position, rotate, and scale each Collider easily and independently of one another.

*A real-world Compound Collider setup*

In the above picture, the *Gun Model* GameObject has a Rigidbody attached, and multiple primitive Colliders as child GameObjects. When the Rigidbody parent is moved around by forces, the child Colliders move along with it. The primitive Colliders will collide with the environment's Mesh Collider, and the parent Rigidbody will alter the way it moves based on forces being applied to it and how its child Colliders interact with other Colliders in the Scene.

Mesh Colliders can't normally collide with each other. If a Mesh Collider is marked as **Convex**, then it can collide with another Mesh Collider. The typical solution is to use primitive Colliders for any objects that move, and Mesh Colliders for static background objects.

**Continuous Collision Detection**
Continuous collision detection is a feature to prevent fast-moving colliders from passing each other. This may happen when using normal (**Discrete**) collision detection, when an object is one side of a collider in one frame, and already passed the collider in the next frame. To solve this, you can enable continuous collision detection on the rigidbody of the fast-moving object. Set the collision detection mode to **Continuous** to prevent the rigidbody from passing through any static (ie, non-rigidbody) MeshColliders. Set it to **Continuous Dynamic** to also prevent the rigidbody from passing through any other supported rigidbodies with collision detection mode set to **Continuous** or **Continuous Dynamic**. Continuous collision detection is supported for Box-, Sphere- and CapsuleColliders. Note that continuous collision detection is intended as a safety net to catch collisions in cases where objects would otherwise pass through each other, but will not deliver physically accurate collision results, so you might still consider decreasing the fixed Time step value in the TimeManager inspector to make the simulation more precise, if you run into problems with fast moving objects.

## Use the right size

The size of the your GameObject's mesh is much more important than the mass of the Rigidbody. If you find that your Rigidbody is not behaving exactly how you expect - it moves slowly, floats, or doesn't collide correctly - consider adjusting the scale of your mesh asset. Unity's default unit scale is 1 unit = 1 meter, so the scale of your imported mesh is maintained, and applied to physics calculations. For example, a crumbling skyscraper is going to fall apart very differently than a tower made of toy blocks, so objects of different sizes should be modeled to accurate scale.

If you are modeling a human make sure he is around 2 meters tall in Unity. To check if your object has the right size compare it to the default cube. You can create a cube using **GameObject->Create Other->Cube**. The cube's height will be exactly 1 meter, so your human should be twice as tall.

If you aren't able to adjust the mesh itself, you can change the uniform scale of a particular mesh asset by selecting it in **Project View** and choosing **Assets->Import Settings...** from the menubar. Here, you can change the scale and re-import your mesh.

If your game requires that your GameObject needs to be instantiated at different scales, it is okay to adjust the values of your Transform's scale axes. The downside is that the physics simulation must do more work at the time the object is instantiated, and could cause a performance drop in your game. This isn't a terrible loss, but it is not as efficient as finalizing your scale with the other two options. Also keep in mind that non-uniform scales can create undesirable behaviors when Parenting is used. For these reasons it is always optimal to create your object at the correct scale in your modeling application.

## Hints

- The relative **Mass** of two Rigidbodies determines how they react when they collide with each other.
- Making one Rigidbody have greater **Mass** than another does not make it fall faster in free fall. Use **Drag** for that.
- A low **Drag** value makes an object seem heavy. A high one makes it seem light. Typical values for **Drag** are between .001 (solid block of metal) and 10 (feather).

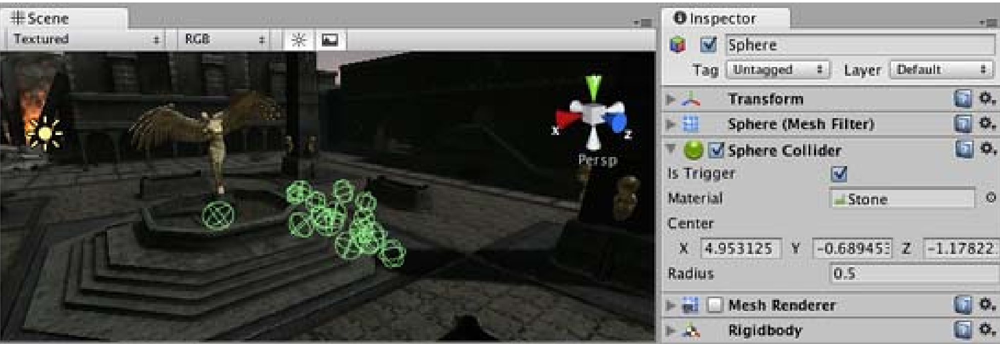
- If you are directly manipulating the Transform component of your object but still want physics, attach a Rigidbody and make it Kinematic.
- If you are moving a GameObject through its Transform component but you want to receive Collision/Trigger messages, you must attach a Rigidbody to the object that is moving.

Page last updated: 2012-11-29

---

## class-SphereCollider

The **Sphere Collider** is a basic sphere-shaped collision primitive.



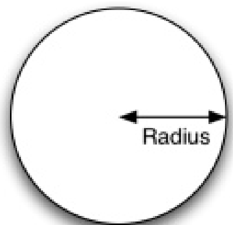

*A pile of Sphere Colliders*

### Properties

| | |
|---|---|
| **Is Trigger** | If enabled, this Collider is used for triggering events, and is ignored by the physics engine. |
| **Material** | Reference to the Physics Material that determines how this Collider interacts with others. |
| **Radius** | The size of the Collider. |
| **Center** | The position of the Collider in the object's local space. |

### Details

The Sphere Collider can be resized to uniform scale, but not along individual axes. It works great for falling boulders, ping pong balls, marbles, etc.




*A standard Sphere Collider*

Colliders work with Rigidbodies to bring physics in Unity to life. Whereas Rigidbodies allow objects to be controlled by physics, Colliders allow objects to collide with each other. Colliders must be added to objects independently of Rigidbodies. A Collider does not necessarily need a Rigidbody attached, but a Rigidbody **must** be attached in order for the object to move as a result of collisions.

When a collision between two Colliders occurs and if at least one of them has a Rigidbody attached, three collision messages are sent out to the objects attached to them. These events can be handled in scripting, and allow you to create unique behaviors with or without making use of the built-in NVIDIA PhysX engine.

#### Triggers

An alternative way of using Colliders is to mark them as a **Trigger**, just check the IsTrigger property checkbox in the Inspector. Triggers are effectively ignored by the physics engine, and have a unique set of three trigger messages that are sent out when a collision with a Trigger occurs. Triggers are useful for triggering other events in your game, like cutscenes, automatic door opening, displaying tutorial messages, etc. Use your imagination!

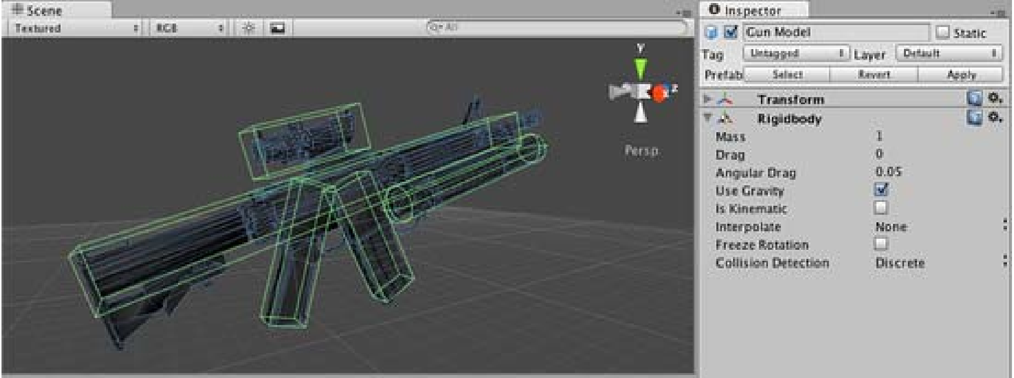
Be aware that in order for two Triggers to send out trigger events when they collide, one of them must include a Rigidbody as well. For a Trigger to collide with a normal Collider, one of them must have a Rigidbody attached. For a detailed chart of different types of collisions, see the collision action matrix in the Advanced section below.

#### Friction and bounciness

Friction, bounciness and softness are defined in the Physisc Material. The Standard Assets contain the most common physics materials. To use one of them click on the Physics Material drop-down and select one, eg. Ice. You can also create your own physics materials and tweak all friction values.

### Compound Colliders

Compound Colliders are combinations of primitive Colliders, collectively acting as a single Collider. They come in handy when you have a complex mesh to use in collisions but cannot use a **Mesh Collider**. To create a Compound Collider, create child objects of your colliding object, then add a primitive Collider to each child object. This allows you to position, rotate, and scale each Collider easily and independently of one another.




*A real-world Compound Collider setup*

In the above picture, the *Gun Model* GameObject has a Rigidbody attached, and multiple primitive Colliders as child GameObjects. When the Rigidbody parent is moved around by forces, the child Colliders move along with it. The primitive Colliders will collide with the environment's Mesh Collider, and the parent Rigidbody will alter the way it moves based on forces being applied to it and how its child Colliders interact with other Colliders in the Scene.

Mesh Colliders can't normally collide with each other. If a Mesh Collider is marked as **Convex**, then it can collide with another Mesh Collider. The typical solution is to use primitive Colliders for any objects that move, and Mesh Colliders for static background objects.

### Hints

- To add multiple Colliders for an object, create child GameObjects and attach a Collider to each one. This allows each Collider to be manipulated independently.
- You can look at the gizmos in the **Scene View** to see how the Collider is being calculated on your object.
- Colliders do their best to match the scale of an object. If you have a non-uniform scale (a scale which is different in each direction), only the Mesh Collider can match completely.
- If you are moving an object through its Transform component but you want to receive Collision/Trigger messages, you must attach a Rigidbody to the object that is moving.
- If you make an explosion, it can be very effective to add a rigidbody with lots of drag and a sphere collider to it in order to push it out a bit from the wall it hits.

### Advanced

#### Collider combinations

There are numerous different combinations of collisions that can happen in Unity. Each game is unique, and different combinations may work better for different types of games. If you're using physics in your game, it will be very helpful to understand the different basic Collider types, their common uses, and how they interact with other types of objects.

#### Static Collider

These are GameObjects that do **not** have a Rigidbody attached, but **do** have a Collider attached. These objects should remain still, or move very little. These work great for your environment geometry. They will not move if a Rigidbody collides with them.

**Rigidbody Collider**

These GameObjects contain both a Rigidbody and a Collider. They are completely affected by the physics engine through scripted forces and collisions. They might collide with a GameObject that only contains a Collider. These will likely be your primary type of Collider in games that use physics.

**Kinematic Rigidbody Collider**

This GameObject contains a Collider and a Rigidbody which is marked **IsKinematic**. To move this GameObject, you modify its Transform Component, rather than applying forces. They're similar to Static Colliders but will work better when you want to move the Collider around frequently. There are some other specialized scenarios for using this GameObject.

This object can be used for circumstances in which you would normally want a Static Collider to send a trigger event. Since a Trigger must have a Rigidbody attached, you should add a Rigidbody, then enable **IsKinematic**. This will prevent your Object from moving from physics influence, and allow you to receive trigger events when you want to.

Kinematic Rigidbodies can easily be turned on and off. This is great for creating ragdolls, when you normally want a character to follow an animation, then turn into a ragdoll when a collision occurs, prompted by an explosion or anything else you choose. When this happens, simply turn all your Kinematic Rigidbodies into normal Rigidbodies through scripting.

If you have Rigidbodies come to rest so they are not moving for some time, they will "fall asleep". That is, they will not be calculated during the physics update since they are not going anywhere. If you move a Kinematic Rigidbody out from underneath normal Rigidbodies that are at rest on top of it, the sleeping Rigidbodies will "wake up" and be correctly calculated again in the physics update. So if you have a lot of Static Colliders that you want to move around and have different object fall on them correctly, use Kinematic Rigidbody Colliders.

**Collision action matrix**

Depending on the configurations of the two colliding Objects, a number of different actions can occur. The chart below outlines what you can expect from two colliding Objects, based on the components that are attached to them. Some of the combinations only cause one of the two Objects to be affected by the collision, so keep the standard rule in mind - physics will not be applied to objects that do not have Rigidbodies attached.

| **Collision detection occurs and messages are sent upon collision** | | | | | | |
|---|---|---|---|---|---|---|
| | Static Collider | Rigidbody Collider | Kinematic Rigidbody Collider | Static Trigger Collider | Rigidbody Trigger Collider | Kinematic Rigidbody Trigger Collider |
| Static Collider | | Y | | | | |
| Rigidbody Collider | Y | Y | Y | | | |
| Kinematic Rigidbody Collider | | Y | | | | |
| Static Trigger Collider | | | | | | |
| Rigidbody Trigger Collider | | | | | | |
| Kinematic Rigidbody Trigger Collider | | | | | | |

| **Trigger messages are sent upon collision** | | | | | | |
|---|---|---|---|---|---|---|
| | Static Collider | Rigidbody Collider | Kinematic Rigidbody Collider | Static Trigger Collider | Rigidbody Trigger Collider | Kinematic Rigidbody Trigger Collider |
| Static Collider | | | | | Y | Y |
| Rigidbody Collider | | | | Y | Y | Y |
| Kinematic Rigidbody Collider | | | | Y | Y | Y |
| Static Trigger Collider | | Y | Y | | Y | Y |
| Rigidbody Trigger Collider | Y | Y | Y | Y | Y | Y |
| Kinematic Rigidbody Trigger Collider | Y | Y | Y | Y | Y | Y |

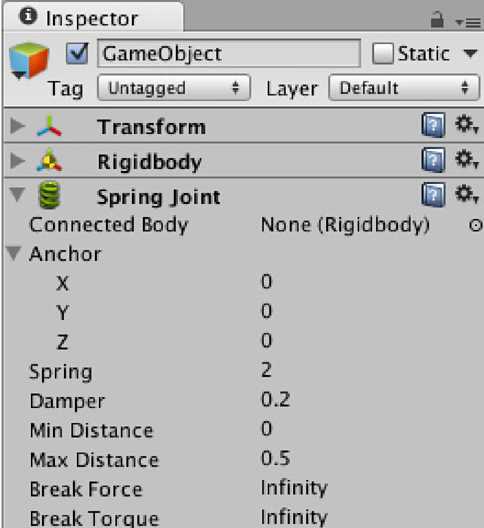
### Layer-Based Collision Detection

In Unity 3.x we introduce something called **Layer-Based Collision Detection**, and you can now selectively tell Unity GameObjects to collide with specific layers they are attached to. For more information click here

Page last updated: 2007-12-08

## class-SpringJoint

The **Spring Joint** groups together two **Rigidbodies**, constraining them to move like they are connected by a spring.



*The Spring Joint **Inspector***

### Properties

| | |
|---|---|
| **Connected Body** | Optional reference to the Rigidbody that the joint is dependent upon. |
| **Anchor** | Position in the object's local space (at rest) that defines the center of the joint. This is not the point that the object will be drawn toward. |
| **X** | Position of the joint's local center along the X axis. |
| **Y** | Position of the joint's local center along the Y axis. |
| **Z** | Position of the joint's local center along the Z axis. |
| **Spring** | Strength of the spring. |
| **Damper** | Amount that the spring is reduced when active. |
| **Min Distance** | Distances greater than this will not cause the Spring to activate. |
| **Max Distance** | Distances less than this will not cause the Spring to activate. |
| **Break Force** | The force that needs to be applied for this joint to break. |
| **Break Torque** | The torque that needs to be applied for this joint to break. |

### Details

Spring Joints allows a Rigidbodied **GameObject** to be pulled toward a particular "target" position. This position will either be another Rigidbodied GameObject or the world. As the GameObject travels further away from this "target" position, the Spring Joint applies forces that will pull it back to its original "target" position. This creates an effect very similar to a rubber band or a slingshot.

The "target" position of the Spring is determined by the relative position from the **Anchor** to the **Connected Body** (or the world) when the Spring Joint is created, or when Play mode is entered. This makes the Spring Joint very effective at setting up Jointed characters or objects in the Editor, but is harder to create push/pull spring behaviors in runtime through scripting. If you want to primarily control a GameObject's position using a Spring Joint, it is best to create an empty GameObject with a Rigidbody, and set that to be the **Connected Rigidbody** of the Jointed object. Then in scripting you can change the position of the **Connected Rigidbody** and see your Spring move in the ways you expect.

#### Connected Rigidbody

You do not need to use a **Connected Rigidbody** for your joint to work. Generally, you should only use one if your object's position and/or rotation is dependent on it. If there is no **Connected Rigidbody**, your Spring will connect to the world.

#### Spring & Damper

**Spring** is the strength of the force that draws the object back toward its "target" position. If this is 0, then there is no force that will pull on the object, and it will behave as if no Spring Joint is attached at all.

**Damper** is the resistance encountered by the **Spring** force. The lower this is, the springier the object will be. As the **Damper** is increased, the amount of bounciness caused by the Joint will be reduced.

#### Min & Max Distance

If the position of your object falls in-between the **Min** & **Max Distances**, then the Joint will not be applied to your object. The position must be moved outside of these values for the Joint to activate.

### Hints

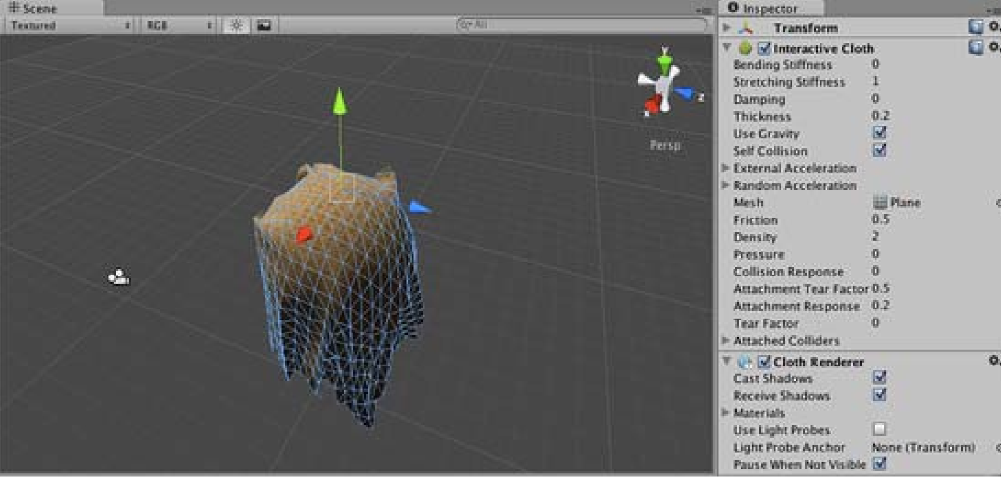
- You do not need to assign a **Connected Body** to your Joint for it to work.
- Set the ideal positions of your Jointed objects in the Editor prior to entering Play mode.
- Spring Joints require your object to have a Rigidbody attached.

Page last updated: 2007-09-15

---

## class-InteractiveCloth

The Interactive Cloth class is a Component that simulates a "cloth-like" behavior on a mesh. Use this Component if you want to use cloth in your scene.




*Interactive cloth in the scene view and its properties in the inspector.*

### Properties

### Interactive Cloth

| | |
|---|---|
| **Bending Stiffness** | Bending stiffness of the cloth. |
| **Stretching Stiffness** | Stretching Stiffness of the cloth. |
| **Damping** | Damp cloth motion. |
| **Thickness** | The thickness of the cloth surface. |
| **Use Gravity** | Should Gravity affect the cloth simulation?. |
| **Self Collision** | Will the cloth collide with itself?. |
| **External Acceleration** | A constant, external acceleration applied to the cloth |
| **Random Acceleration** | A random, external acceleration applied to the cloth |
| **Mesh** | Mesh that will be used by the interactive cloth for the simulation |
| **Friction** | The friction of the cloth. |
| **Density** | The density of the cloth. |
| **Pressure** | The pressure inside the cloth/ |
| **Collision Response** | How much force will be applied to colliding rigidbodies?. |
| **Attachment Tear Factor** | How far attached rigid bodies need to be stretched, before they will tear off. |
| **Attachment Response** | How much force will be applied to attached rigidbodies?. |
| **Tear Factor** | How far cloth vertices need to be stretched, before the cloth will tear. |
| **Attached Colliders** | Array that contains the attached colliders to this cloth |

The Interactive Cloth Component depends of the Cloth Renderer Component, this means that this component cannot be

removed if the Cloth Renderer is present in the Game Object.

### Cloth Renderer

| | |
|---|---|
| **Cast Shadows** | If selected the cloth will cast shadows |
| **Receive Shadows** | The cloth can receive Shadows if enabled |
| **Materials** | Materials that the cloth will use. |
| **Use Light Probes** | If selected, light probes will be enabled. |
| **Light Probe Anchor** | Light Probe lighting is interpolated at the center of the Renderer's bounds or at the position of the anchor, if assigned. |
| **Pause When Not Visible** | If selected, the simulation will not be calculated when the cloth is not being rendered by the camera. |

### Adding an Interactive Cloth Game Object

To add an Interactive Cloth in the scene just select **GameObject->Create Other->Cloth**.

### Hints.

- Using lots of clothes in your game will reduce exponentially the performance of your game.
- If you want to simulate clothing on characters, check out the Skinned Cloth component instead, which interacts with the SkinnedMeshRenderer component and is much faster then InteractiveCloth.
- To attach the cloth to other objects, use the *Attached Colliders* property to assign other objects to attach to. The colliders must overlap some vertices of the cloth mesh for this to work.
- Attached Colliders' objects must intersect with the cloth you are attaching to.

### Notes.

- Cloth simulation will generate normals but not tangents. If the source mesh has tangents, these will be passed to the shader unmodified - so if you are using a shader which depends on tangents (such as bump mapped shaders), the lighting will look wrong for cloth which has been moved from its initial position.

Page last updated: 2012-12-10

---

# class-SkinnedCloth

*Skinned cloth in the scene view and in the inspector.*

The **SkinnedCloth** component works together with the **SkinnedMeshRenderer** to simulate clothing on a character. If you have an animated character which uses the SkinnedMeshRenderer, you can add a SkinnedCloth component to the game object with the SkinnedMeshRenderer to make him appear more life-like. Just select the GameObject with the SkinnedMeshRender, and add a SkinnedCloth component using **Component->Physics->Skinned Cloth**.

What the SkinnedCloth component does, is to take the vertex output from the SkinnedMeshRenderer and apply clothing simulation to that. The SkinnedCloth component has a set of per-vertex coefficients, which define how free the simulated cloth can move with respect to the skinned mesh.

These coefficients can be visually edited using the scene view and the inspector, when the game object with the SkinnedCloth component is selected. There are two editing modes, selection and vertex painting. In selection mode, you click on vertices in the scene view to select them, and then edit their coefficients in the inspector. In vertex paining mode, you set the coefficient values you want in the inspector, enable the "paint" button next to those coefficients you want to change, and click on the

vertices to apply the values to those.

Note that skinned cloth simulation is only driven by the vertices skinned by the SkinnedMeshRenderer, and will not otherwise interact with any colliders. This makes skinned cloth simulation much faster then the fully physical Interactive Cloth component, as it does not need to be simulated on the same frame rate and the same thread as the rest of the physics simulation.

You can disable or enable the skinned cloth component at any time to turn it on or off. Turning it off will switch rendering to the normal SkinnedMeshRenderer, so you can switch between these whenever needed to dynamically adjust for varying performance. You can also smoothly cross-fade between the two modes from a script using the SkinnedCloth.SetEnabledFading() method, to make the transition unnoticeable to the player.

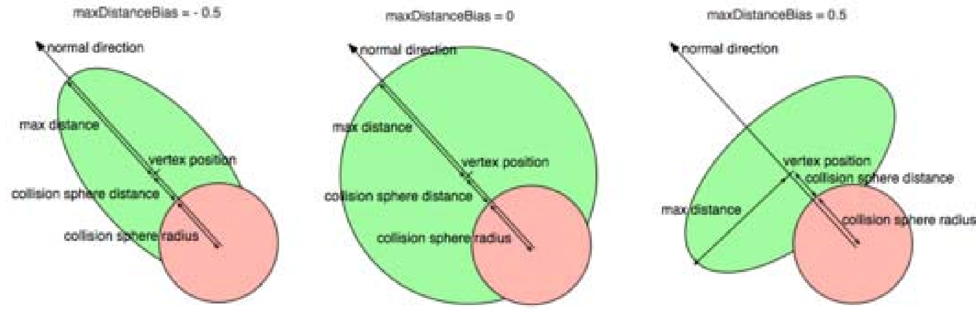
Note that cloth simulation will generate normals but not tangents. If the source mesh has tangents, these will be passed to the shader unmodified - so if you are using a shader which depends on tangents (such as bump mapped shaders), the lighting will look wrong for cloth which has been moved from it's initial position.

### Cloth Coefficients

There a four coefficients per vertex, which define how cloth vertices can move with respect to the skinned vertices and normals. These are:

| | |
|---|---|
| **Max Distance** | Distance a vertex is allowed to travel from the skinned mesh vertex position. The SkinnedCloth component makes sure that the cloth vertices stay within maxDistance from the skinned mesh vertex positions. If maxDistance is zero, the vertex is not simulated but set to the skinned mesh vertex position. This behavior is useful for fixing the cloth vertex to the skin of an animated character - you will want to do that for any vertices which shouldn't be skinned, or for parts which are somehow fixed to the character's body (such as the waist of trousers, fixed by a belt). However, if you have large parts of the character which should not use cloth simulation (such as the face or hands), for best performance, set those up as a separate skinned mesh, which does not have a SkinnedCloth component. |
| **Distance Bias** | Distorts the sphere defined by the maxDistance based on skinned mesh normals. The feature is disabled for a value of 0.0 (default). In this case the max distance sphere is undistorted. Decreasing the maxDistanceBias towards -1.0 reduces the distance the vertex is allowed to travel in the tangential direction. For -1.0 the vertex has to stay on the normal through the skinned mesh vertex position and within maxDistance to the skinned mesh vertex position. Increasing maxDistanceBias towards 1.0 reduces the discance the vertex is allowed to travel in the normal direction. At 1.0 the vertex can only move inside the tangental plane within maxDistance from the skinned mesh vertex position. |
| **Collision Sphere Radius and Collision Sphere Distance** | Definition of a sphere a vertex is not allowed to enter. This allows collision against the animated cloth. The pair (collisionSphereRadius, collisionSphereDistance) define a sphere for each cloth vertex. The sphere's center is located at the position constrainPosition - constrainNormal * (collisionSphereRadius + collisionSphereDistance) and its radius is collisionSphereRadius, where constrainPosition and constrainNormal are the vertex positions and normals generated by the SkinnedMeshRenderer. The SkinnedCloth makes sure that the cloth vertex does not enter this sphere. This means that collisionSphereDistance defines how deeply the skinned mesh may be penetrated by the cloth. It is typically set to zero. collisionSphereRadius should be set to a value larger then the distance between the neighboring vertices to make sure the cloth vertices will not be able to slip around the collision spheres. In such a setup, the cloth will appear to collides against the skinned mesh. |

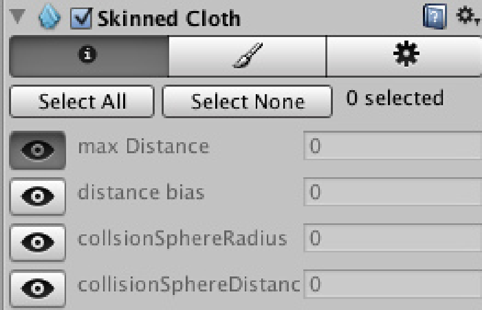
Refer to this image for a visual representation on how these coefficients work with respect to a skinned vertex and normal for different values of maxDistanceBias. The red area is the collision sphere defined by collisionSphereRadius and collisionSphereDistance, which the cloth vertex cannot enter. Thus, the green area, defined by maxDistance and maxDistanceBias subtracted by the red area defines the space in which the cloth vertex can move.



### The SkinnedCloth inspector

When you select a GameObject with a SkinnedCloth component, you can use the SkinnedCloth inspector to edit cloth vertex coefficients, and other properties. The inspector has three tabs:

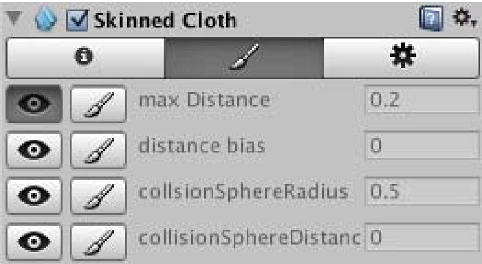
**Vertex Selection Tool**




In this mode you can select vertices in the scene view, and the set their coefficients in the inspector (see the previous section for an explanation on how the cloth coefficients work). It is possible to set multiple coefficients by holding the shift key, or by dragging a rectangle with the mouse. When multiple vertices are selected, the inspector will display average values for the vertices coefficients. When you change the values, however, that coefficient will be set to the same value for all vertices. If you switch the scene view to wireframe mode, you will also be able to see and to select back-facing vertices, this can be useful when you want to select full parts of the character.

To help you understand which values the coefficients have for all the vertices, you can click the eye icon next to a coefficient field, to make the editor visualize that coefficient in the scene view. This shows the vertices with the lowest value of that coefficient in a green tint, mid-range values will be yellow, and the highest values get a blue tint. The colors scale is always chosen relative to the used value range of that coefficient, and is independent of absolute values.

**Vertex Painting Tool**




Similar to the vertex selection, this is a tool to help you configure the vertex coefficient values. Unlike vertex selection, you don't need to click on a vertex before changing a value - in this mode, you just enter the values you want to set, enable the paintbrush toggle next to the coefficients you want to change, and then click on all vertices you want to set that value for.

**Configuration**

The third tab lets you configure various properties of the skinned cloth:

| | |
|---|---|
| **Bending** | Bending stiffness of the cloth. |

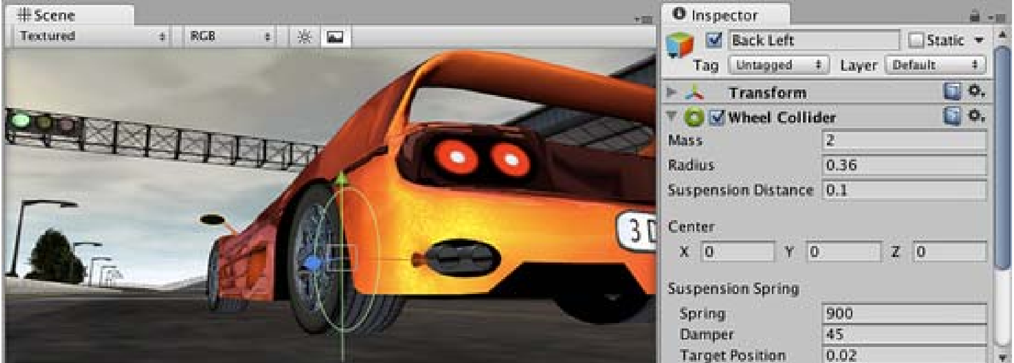
| | |
|---|---|
| **Stiffness** | |
| **Stretching Stiffness** | Streching stiffness of the cloth. |
| **Damping** | Damp cloth motion |
| **Thickness** | Thickness of the cloth surface. (0.001 - 10000) |
| **Use Gravity** | If enabled, gravity will affect the cloth simulation. |
| **Self Collision** | If enabled, the cloth can can collide with itself. |
| **External Acceleration** | A constant, external acceleration applied to the cloth. |
| **Random Acceleration** | A random, external acceleration applied to the cloth. |
| **World Velocity Scale** | How much world-space movement of the character will affect cloth vertices. The higher this value is, the more the cloth will move as a reaction to world space movement of the GameObject. Basically, this defines the air friction of the SkinnedCloth. |
| **World Acceleration Scale** | How much world-space acceleration of the character will affect cloth vertices. The higher this value is, the more the cloth will move as a reaction to world space acceleration of the GameObject. If the cloth appears lifeless, try increasing this value. If it appears to get unstable when the character accelerates, try decreasing the value. |

Page last updated: 2012-12-10

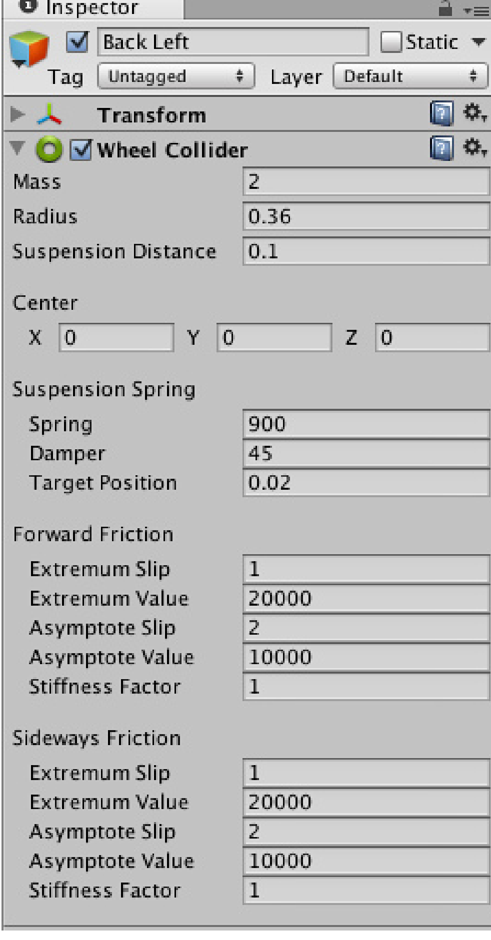
---

## class-WheelCollider

The **Wheel Collider** is a special collider for grounded vehicles. It has built-in collision detection, wheel physics, and a slip-based tire friction model. It can be used for objects other than wheels, but it is specifically designed for vehicles with wheels.



*The Wheel Collider Component. Car model courtesy of ATI Technologies Inc.*




## Properties

| | |
|---|---|
| **Center** | Center of the wheel in object local space. |
| **Radius** | Radius of the wheel. |
| **Suspension Distance** | Maximum extension distance of wheel suspension, measured in local space. Suspension always extends downwards through the local Y-axis. |
| **Suspension Spring** | The suspension attempts to reach a **Target Position** by adding spring and damping forces. |
| **Spring** | Spring force attempts to reach the **Target Position**. A larger value makes the suspension reach the **Target Position** faster. |
| **Damper** | Dampens the suspension velocity. A larger value makes the **Suspension Spring** move slower. |
| **Target Position** | The suspension's rest distance along Suspension Distance. 0 maps to fully extended suspension, and 1 maps to fully compressed suspension. Default value is zero, which matches the behavior of a regular car's suspension. |
| **Mass** | The Mass of the wheel. |
| **Forward/Sideways Friction** | Properties of tire friction when the wheel is rolling forward and sideways. See Wheel Friction Curves section below. |

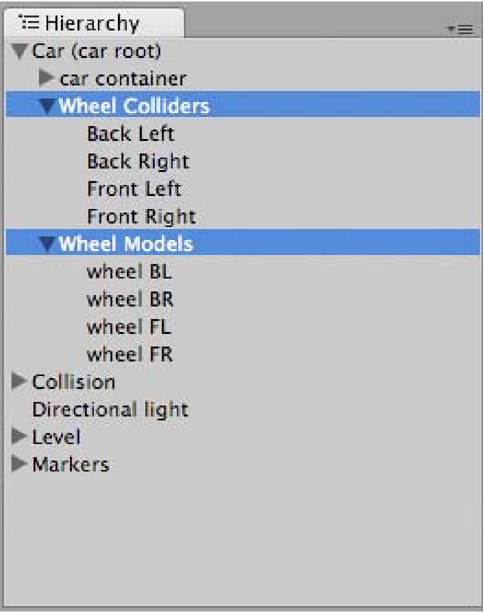
## Details

The wheel's collision detection is performed by casting a ray from **Center** downwards through the local Y-axis. The wheel has a **Radius** and can extend downwards according to the **Suspension Distance**. The vehicle is controlled from scripting using different properties: **motorTorque**, **brakeTorque** and **steerAngle**. See the Wheel Collider scripting reference for more information.

The Wheel Collider computes friction separately from the rest of physics engine, using a slip-based friction model. This allows for more realistic behaviour but also causes Wheel Colliders to ignore standard Physic Material settings.

### Wheel collider setup

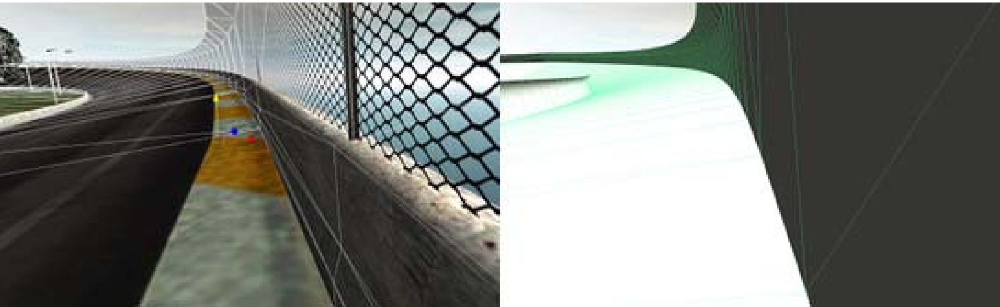
You do not turn or roll WheelCollider objects to control the car - the objects that have WheelCollider attached should always be fixed relative to the car itself. However, you might want to turn and roll the graphical wheel representations. The best way to do this is to setup separate objects for Wheel Colliders and visible wheels:




*Wheel Colliders are separate from visible Wheel Models*

**Collision geometry**

Because cars can achieve large velocities, getting race track collision geometry right is very important. Specifically, the collision mesh should not have small bumps or dents that make up the visible models (e.g. fence poles). Usually a collision mesh for the race track is made separately from the visible mesh, making the collision mesh as smooth as possible. It also should not have thin objects - if you have a thin track border, make it wider in a collision mesh (or completely remove the other side if the car can never go there).



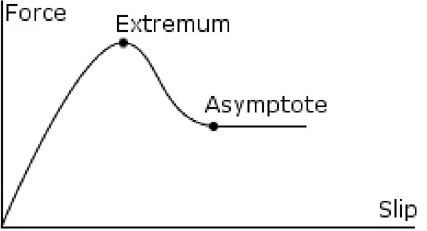
*Visible geometry (left) is much more complex than collision geometry (right)*

Wheel Friction Curves

**Wheel Friction Curves**

Tire friction can be described by the *Wheel Friction Curve* shown below. There are separate curves for the wheel's forward (rolling) direction and sideways direction. In both directions it is first determined how much the tire is slipping (based on the speed difference between the tire's rubber and the road). Then this slip value is used to find out tire force exerted on the contact point.

The curve takes a measure of tire slip as an input and gives a force as output. The curve is approximated by a two-piece spline. The first section goes from *(0 , 0)* to ***(ExtremumSlip , ExtremumValue)***, at which point the curve's tangent is zero. The second section goes from ***(ExtremumSlip , ExtremumValue)*** to ***(AsymptoteSlip , AsymptoteValue)***, where curve's tangent is again zero:



*Typical shape of a wheel friction curve*

The property of real tires is that for low slip they can exert high forces, since the rubber compensates for the slip by stretching. Later when the slip gets really high, the forces are reduced as the tire starts to slide or spin. Thus, tire friction curves have a shape like in the image above.

| | |
|---|---|
| **Extremum Slip/Value** | Curve's extremum point. |
| **Asymptote Slip/Value** | Curve's asymptote point. |
| **Stiffness** | Multiplier for the **Extremum Value** and **Asymptote Value** (default is 1). Changes the stiffness of the friction. Setting this to zero will completely disable all friction from the wheel. Usually you modify stiffness at runtime to simulate various ground materials from scripting. |

### Hints

- You might want to decrease physics timestep length in Time Manager to get more stable car physics, especially if it's a racing car that can achieve high velocities.
- To keep a car from flipping over too easily you can lower its Rigidbody center of mass a bit from script, and apply "down pressure" force that depends on car velocity.

Page last updated: 2012-12-10

---

## comp-GameObjectGroup

**GameObjects** are containers for all other Components. All objects in your game are GameObjects that contain different Components. Technically you can create a Component without GameObject, but you won't be able to use it until you apply it to a GameObject.

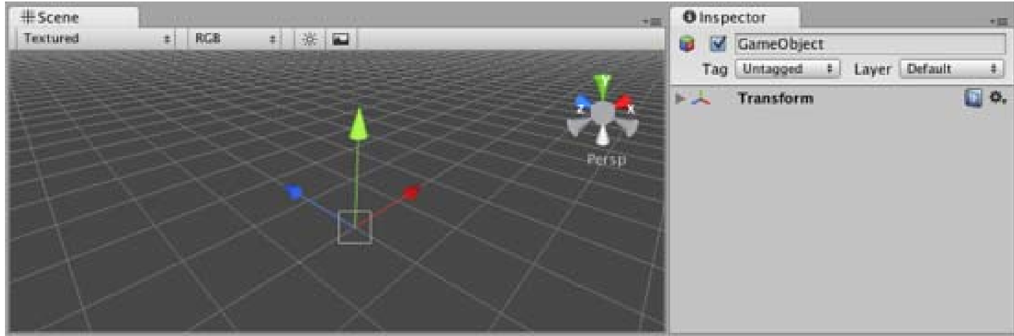
- GameObject

Page last updated: 2009-05-04

---

## class-GameObject

**GameObjects** are containers for all other Components. All objects in your game are inherently GameObjects.




*An empty GameObject*

### Creating GameObjects

GameObjects do not add any characteristics to the game by themselves. Rather, they are containers that hold Components, which implement actual functionality. For example, a Light is a Component which is attached to a GameObject.

If you want to create a Component from script, you should create and empty GameObject and then add required Component using **gameObject.AddComponent(*ClassName*)** function. You can't create Component and then make a reference from object to it.

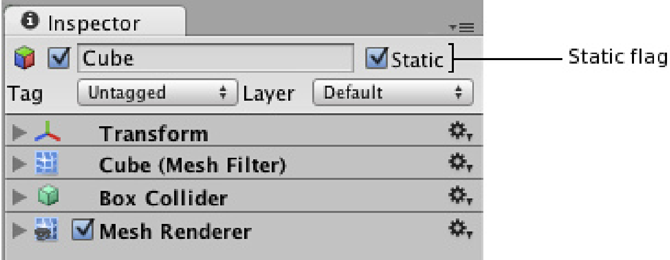
From scripts, Components can easily communicate with each other through message sending or the **GetComponent(*TypeName*)** function. This allows you to write small, reusable scripts that can be attached to multiple GameObjects and reused for different purposes.

### Details
Aside from being a container for the components, GameObjects have a **Tag**, a **Layer** and a **Name**.

Tags are used to quickly find objects, utilizing the Tag name. Layers can be used to cast rays, render, or apply lighting to certain groups of objects only. Tags and Layers can be set with the Tag Manager, found in **Edit->Project Settings->Tags**.

### Static Checkbox
In Unity, there is a new checkbox in the GameObject called **Static**. This checkbox is used for:

- preparing static geometry for automatic batching
- calculating Occlusion Culling



*The Static checkbox is used when generating Occlusion data*

When generating Occlusion data, marking a GameObject as **Static** will enable it to cull (or disable) mesh objects that cannot be seen behind the Static object. Therefore, any environmental objects that are not going to be moving around in your scene **should be marked** as Static.

For more information about how Occlusion Culling works in Unity please read the Occlusion Culling page.

### Hints
- For more information see the GameObject scripting reference page.
- More information about how to use layers can be found here.
- More information about how to use tags can be found here.

Page last updated: 2010-09-03

---

## comp-ImageEffects

This group handles all **Render Texture**-based fullscreen image postprocessing effects. They are only available with Unity Pro. They add a lot to the look and feel of your game without spending much time on artwork.

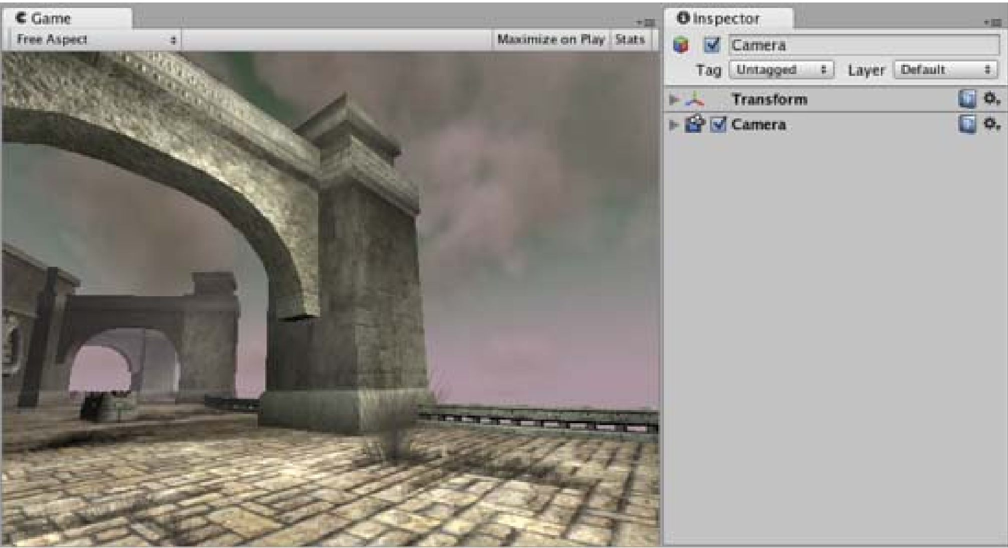
All image effects make use of Unity's **OnRenderImage** function which any MonoBehavior attached to a camera can overwrite to accomplish a wide range of custom effects.

Image effects can be executed directly after the opaque pass or after opaque and transparent passes (default). The former behavior can very easily be acquired by adding the attribute **ImageEffectOpaque** to the OnRenderImage function of the effect in question. For an example of an effect doing this, have a look at the Edge Detection effect.

- Antialiasing (PostEffect)
- Bloom
- Camera Motion Blur
- Depth of Field
- Noise And Grain
- Screen Overlay
- Color Correction Lookup Texture
- Bloom and Lens Flares
- Color Correction Curves
- Contrast Enhance
- Crease
- Depth of Field 3.4

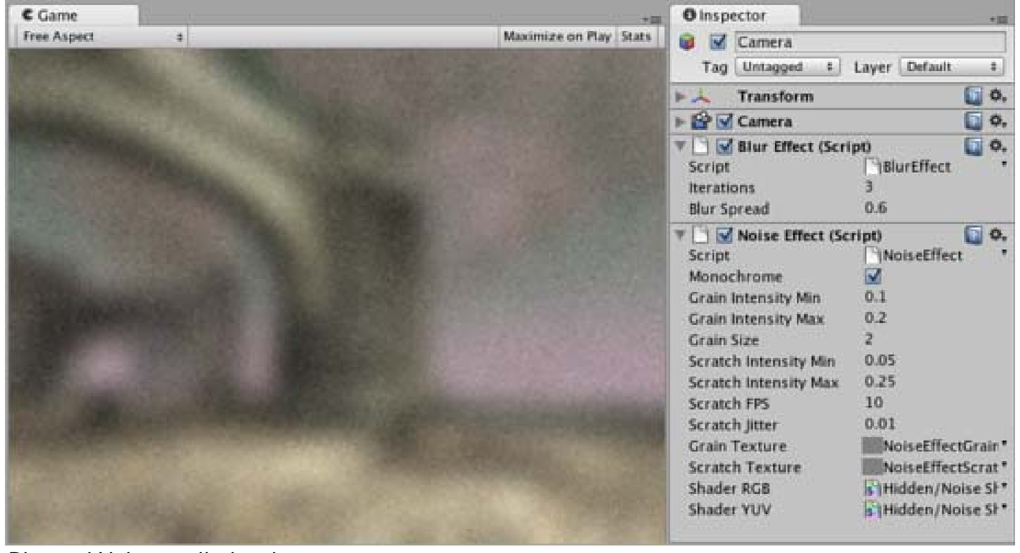
- Tonemapping
- Edge Detect Effect Normals
- Fisheye image effect
- Global Fog
- Sun Shafts
- Tilt Shift
- Vignetting (and Chromatic Aberration)
- Blur image effect
- Color Correction image effect
- Contrast Stretch image effect
- Edge Detection image effect
- Glow image effect
- Grayscale image effect
- Motion Blur image effect
- Noise image effect
- Sepia Tone image effect
- Screen Space Ambient Occlusion (SSAO) image effect
- Twirl image effect
- Vortex image effect

The scene used in above pages looks like this without any image effects applied:



*Scene with no image postprocessing effects.*

Multiple image effects can be "stacked" on the same camera. Just add them and it will work.



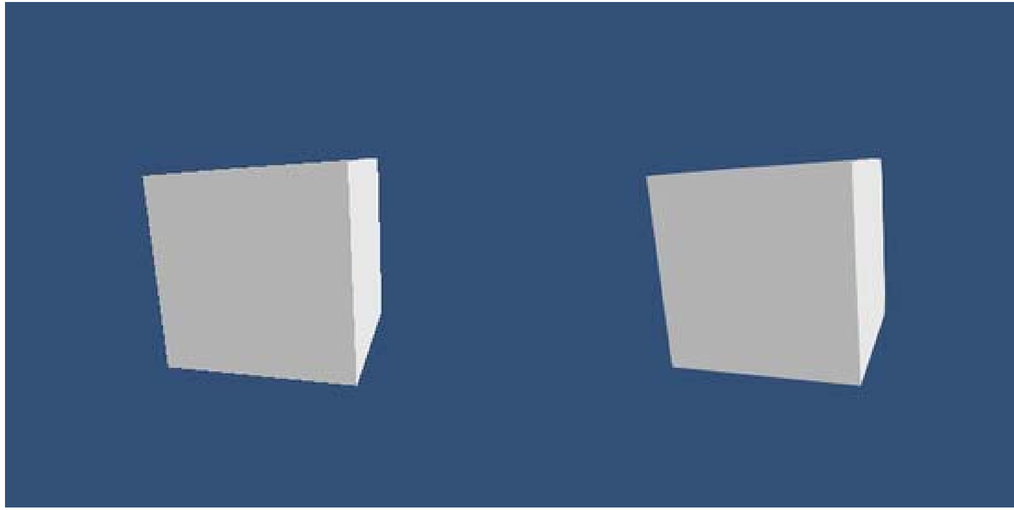
*Blur and Noise applied to the same camera.*

Page last updated: 2012-11-16

---

## script-AntialiasingAsPostEffect

The **Antialiasing (PostEffect)** offers a set of algorithms designed to give a smoother appearance to graphics. When two areas of different colour adjoin in an image, the shape of the pixels can form a very distinctive "staircase" along the boundary. This effect is known as **aliasing** and hence antialiasing refers to any measure which reduces the effect.



**The cube on the left is rendered without antialiasing while the one on the right shows the effect of the FXAA1PresetB algorithm**

The antialiasing algorithms are image based, which is very useful for deferred rendering where traditional multisampling is not properly supported. The algorithms currently supported are NVIDIA's FXAA, FXAA II, FXAA III (tweakable and console optimized), simpler edge blurs (NFAA, SSAA) that blur only local edges and an adaption of the DLAA algorithm that also addresses long edges. SSAA is the fastest technique, followed by NFAA, FXAAII, FXAA II, DLAA and the the other FXAA's. Typically, the quality of antialiasing trades off against the speed of the algorithm but there may be situations where the choice of algorithm makes little difference.

For those especially interested in console and NaCl deployment, the optimized **FXAA III** implementation offers the best tradeoff between quality and performance and can furthermore be tweaked towards sharper or blurrier looks.

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

### Properties

| | |
|---|---|
| **AA Technique** | The algorithm to be used. |

### Hardware support

This effect requires a graphics card with pixel shaders (3.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2004 (GeForce 6), AMD cards since 2005 (Radeon X1300), Intel cards since 2006 (GMA X3000); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2012-01-06

---

## script-Bloom

**Blooming** is the optical effect where light from a bright source (such as a glint) appears to leak into surrounding objects. The

**Bloom** image effect adds bloom and also automatically generates lens flares in a highly efficient way. Bloom is a very distinctive effect that can make a big difference to a scene and may suggest a magical or dreamlike environment especially when used in conjunction with HDR rendering. On the other hand, given proper settings it's also possible to enhance photorealism using this effect. Glow around very bright objects is a common phenomena observed in film and photography, where luminance values differ vastly. Bloom is an enhanced version of the Glow and Bloom And Flares image effects.

*Example showing proper HDR glow as created by the **Bloom** effect. In this scene, bloom uses a threshhold of 1.0 indicating that only HDR reflections, highlights or emissive surfaces glow, but common lighting is generally unaffected. In this particular example, only the car window (sporting the reflection of HDR sun values) glows.*

*Example showing **Anamorphic Lens Flares** result as created by the **Bloom** effect*

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

## Properties

| | |
|---|---|
| **Quality** | High quality preserves high frequencies and reduces aliasing. |
| **Mode** | Choose complex mode to show advanced options. |
| **Blend** | The method used to add bloom to the color buffer. The softer **Screen** mode is better for preserving bright image details but doesn't work with HDR. |
| **HDR** | Whether bloom is using HDR buffers. This will result in a different look as pixel intensities may leave the [0,1] range, see details in tonemapping and HDR. |
| **Cast lens flares** | Enable or disable automatic screen based lens flare generation. |

| | |
|---|---|
| **Intensity** | The global light intensity of the added light (affects bloom and lens flares). |
| **Threshhold** | Regions of the image brighter than this threshold receive blooming (and potentially lens flares). |
| **RGB Threshhold** | Chose different threshholds for R, G and B. |

**Blur iterations** The number of times gaussian blur is applied. More iterations improve smoothness but take extra time to

| | |
|---|---|
| | process and hide small frequencies. |
| **Sample distance** | The max radius of the blur. Does not affect performance. |
| **Use alpha mask** | The degree to which the alpha channel acts as a mask for the bloom effect. |

| | |
|---|---|
| **Lens Flares** | The type of lens flare. The options are **Ghosting**, **Anamorphic** or a mix of the two. |
| **Local intensity** | Local intensity used only for lens flares. 0 disables lens flares entirely. |
| **Local threshold** | The accumulative light intensity threshold that defines which image parts are candidates for lens flares. |
| **Stretch width** | The width for anamorphic lens flares. |
| **Rotation** | The orientation for anamorphic lens flares. |
| **Blur iterations** | The number of times blurring is applied to anamorphic lens flares. More iterations improve smoothness but take more processing time. |
| **Saturation** | (De-)saturates lens flares. If 0, lens flares will fully receive the **Tint Color**. |
| **Tint Color** | Color modulation for the anamorphic flare type. |
| **1st-4th Color** | Color modulation for all lens flares when **Ghosting** or **Combined** is chosen. |
| **Lens flare mask** | Mask used to prevent lens flare artifacts at screen edges. |

### Blend Modes: Add and Screen

Blend modes determine the way that two images will be combined when overlaid. Each pixel from the base image is combined mathematically with the pixel in the corresponding position in the overlay image. Two blend modes are available for this image effect, Add and Screen.

#### Add Mode

When the images are blended in Add mode, the values of the color channels (red, green and blue) are simply added together and clamped to the maximum value of 1. The overall effect is that areas of each image that aren't especially bright can easily blend to maximum brightness in the result. The final image tends to lose color and detail and so Add mode is useful when a dazzling "white out" effect is required.

#### Screen Mode

Screen mode is so named because it simulates the effect of projecting the two source images onto a white screen simultaneously. Each color channel is combined separately but identically to the others. Firstly, the channel values of the two source pixels are inverted (ie, subtracted from 1). Then, the two inverted values are multiplied together and the result is inverted. The result is brighter than either of the two source pixels but it will be at maximum brightness only if one of the source colors was also. The overall effect is that more color variation and detail from the source images is preserved, leading to a gentler effect than Add mode.

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2012-10-30

---

## script-CameraMotionBlur

**Motion Blur** is a common postprocessing effect simulating the fact that for most camera systems 'light' gets accumulated over time (instead of just taking discrete snapshots). Fast camera or object motion will hence produce blurred images.

*Example of a standard camera motion blur when the camera is moving sideways. Also, notice how the background areas blur less than the foreground ones which is a typical side effect of motion blur.*

The current Motion Blur implementation only supports blur due to camera motion with the option to exclude certain layers (useful for excluding characters and/or dynamic objects, especially when those are following the camera movement). It can however be extended to support dynamic objects if an additional script keeps track of each objects model matrix and updates the velocity buffer.

*Example showing Camera Motion Blur with dynamic objects (canisters, bus) being excluded*

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

## Properties

| | |
|---|---|
| **Technique** | Motion Blur algorithm. **Reconstruction** filters will generally give best results at the expense of performance and a limited blur radius of 10 pixels unless a DirectX11 enabled graphics device is used. |
| **Velocity Scale** | Higher scale makes image more likely to blur. |
| **Velocity Max** | Maximum pixel distance blur will be clamped to and tile size for reconstruction filters (see below). |
| **Velocity Min** | Minimum pixel distance at which blur is removed entirely. |

**Camera Motion** specific:

| | |
|---|---|
| **Camera Rotation** | Scales strength of blurs due to camera rotations. |
| **Camera Movement** | Scales strength of blurs due to camera translations. |

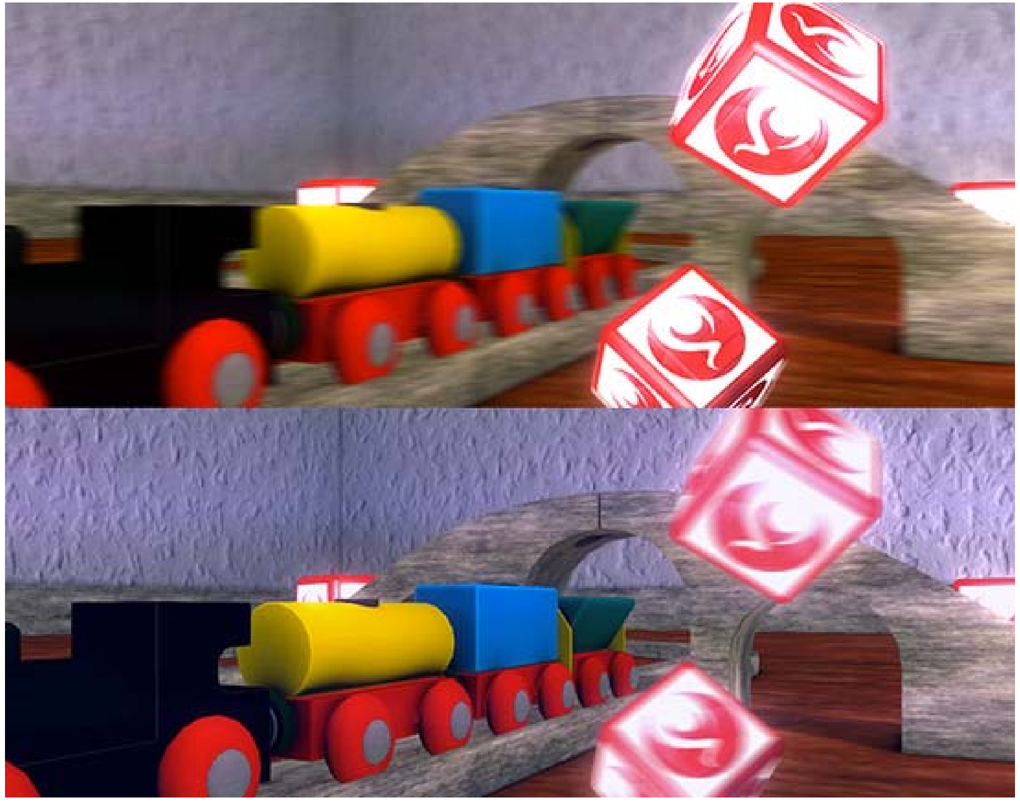
**Local Blur**, **Reconstruction** and **ReconstructionDX11** specific:

| | |
|---|---|
| **Exclude layers** | Objects in this layer will remain unaffected. |
| **Velocity downsample** | Lower resolution velocity buffers might help performance but will heavily degrade blur quality. Might still be a valid option for simple scenes. |
| **Sampler Jitter** | Adding noise helps prevent ghosting for the **Reconstruction** filter. |

| | |
|---|---|
| **Max Sample Count** | Number of samples used to determine the blur. Affects performance a lot. |
| **Preview (Scale)** | Preview how blur might look like given artificial camera motion values. |

### Motion Blur Filters (Technique)

**Local Blur** simply performs a directional blur along the current's pixel velocity. Being essentially a *gather* operation, it is suited for scenes with a low geometric complexity (e.g. vast terrains), large blur radii or when 'realism' is not the governing factor. One shortcoming is that it can't produce proper 'overlaps' of blurred objects onto focused background areas. Another one that excluded objects 'smear' onto blurred areas.

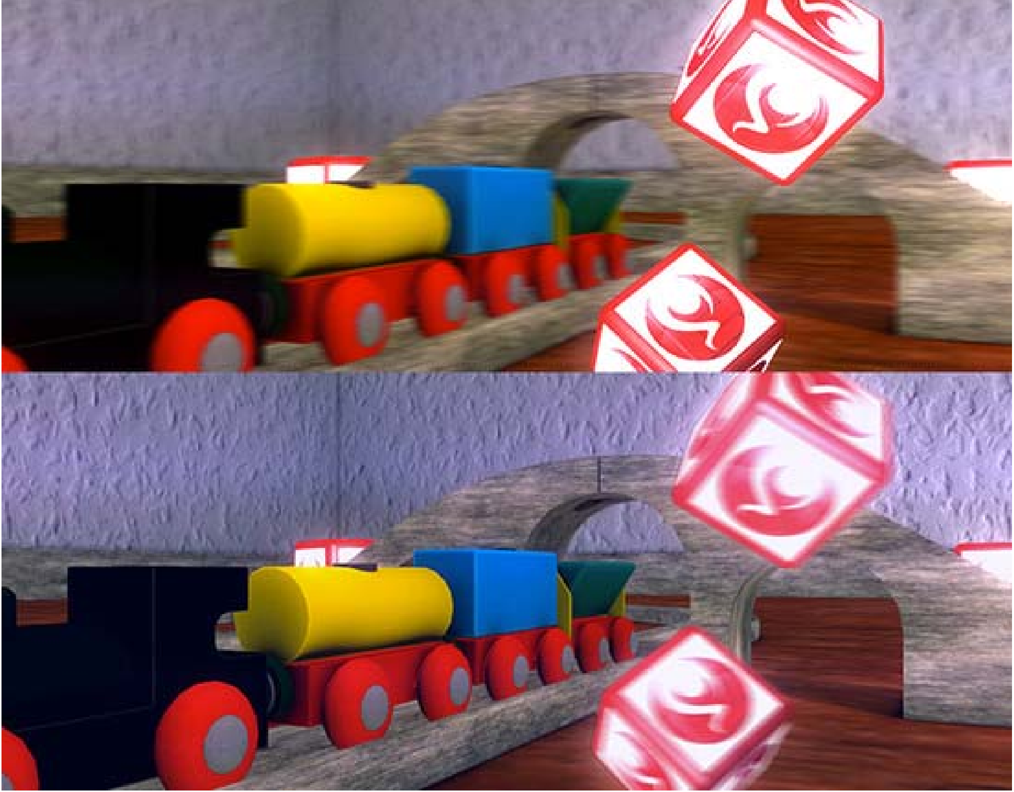


*Example using the **Local Blur** technique while the camera is translating sideways and either foreground (top) or background is excluded (bottom). Notice that both of the above mentioned artifacts apply, typically degrading image quality. If those are not important in your case, this motion blur technique is a fast and effective option.*

**Reconstruction** filters can produce more realistic blur results. The name Reconstruction is derived from the fact that the filter tries to estimate backgrounds, even if there is no information available in the given color and depth buffers. The results can be of higher quality and shortcomings of the *Local Blur*'s gather filter can be avoided (it can e.g. produce proper overlaps).

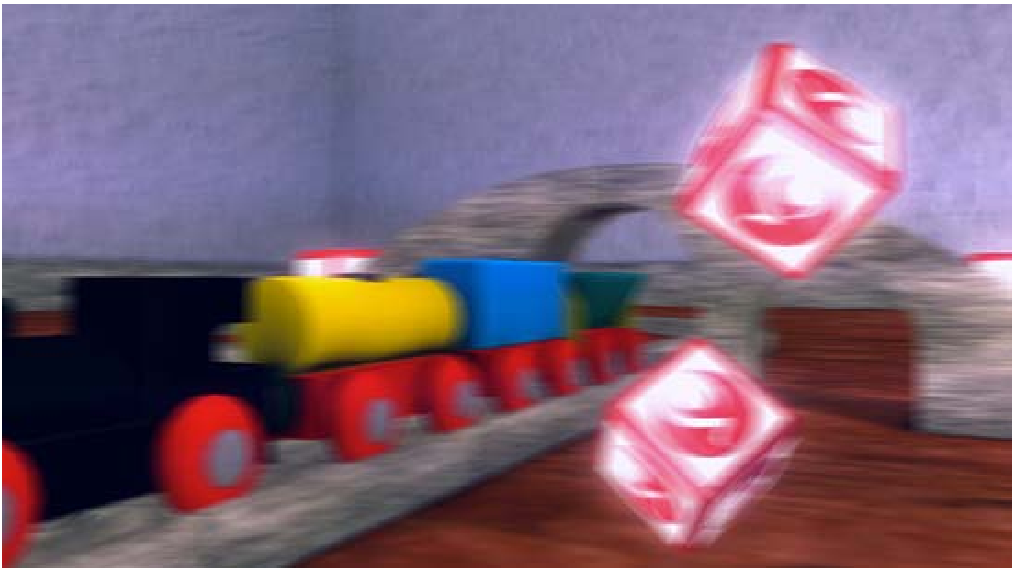
It is based on the paper *A Reconstruction Filter for Plausible Motion Blur* (http://graphics.cs.williams.edu/papers/MotionBlurI3D12/). The algorithm chops the image into tiles of the size **Velocity Max** and uses the maximum velocity in the area to simulate a blurry pixel scattering onto neighbouring areas. Artifacts can arise if the velocity is highly varying while the mentioned tile size is large.

The DirectX11 exclusive filter **ReconstructionDX11** allows arbitrary blur distances (aka tile size or **Velocity Max**) and a flexible number of samples.



*Example using the **Reconstruction** technique while the camera is translating sideways. Notice that this time, the mentioned artifacts are less severe as the Reconstruction filter tries to solve those cases (cubes overlapping when background is excluded (bottom) or excluded cubes not smearing onto blurred background (top)).*

While all of the above filters need a prepass to generate a velocity buffer, the **Camera Motion** filter solely works on the camera motion. It generates a global filter direction based on camera change and blurs the screen along that direction (see http://www.valvesoftware.com/publications/2008/GDC2008_PostProcessingInTheOrangeBox.pdf for more details). It is especially suited for smoothing fast camera rotations, for instance in first person shooter games.



*Example using the **Camera Motion** technique. Notice that the blur is uniform across the entire screen.*

### Hardware support

This effect requires a graphics card with pixel shaders (3.0) or OpenGL ES 2.0. Additionally, depth texture support is required. PC: NVIDIA cards since 2004 (GeForce 6), AMD cards since 2005 (Radeon X1300), Intel cards since 2006 (GMA X3000); Mobile: OpenGL ES 2.0 with depth texture support; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2012-10-03

---

## script-DepthOfFieldScatter

**Depth of Field** is a common postprocessing effect that simulates the properties of a camera lens. This version is a more modern and sophisticated version of the old Depth of Field 3.4 effect that works especially well with HDR rendering and a DirectX 11 compatible graphics device.

In real life, a camera can only focus sharply on an object at a specific distance; objects nearer or farther from the camera will be somewhat out of focus. The blurring not only gives a visual cue about an object's distance but also introduces **Bokeh** which is the term for pleasing visual artifacts that appear around bright areas of the image as they fall out of focus. Common **Bokeh** shapes are discs, hexagons and other shapes of higher level dihedral groups.

While the regular version only supports disc shapes (generated via circular texture sampling), the DirectX 11 version is able to splat any shape as defined by the **Bokeh Texture**.

An example of Depth of Field effect can be seen in the following image, displaying the results of a focused foreground and a defocused background..

*The DirectX11 version of this effect can create nicely defined bokeh shapes at a very reasonable cost.*

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

### Properties

| **Focal Settings** | |
|---|---|
| **Visualize** | Overlay color indicating camera focus. |
| **Focal distance** | The distance to the focal plane from the camera position in world space. |
| **Focal Size** | Increase the total focal area. |
| **Focus on Transform** | Determine the focal distance using a target object in the scene. |
| **Aperture** | The camera's aperture defining the transition between focused and defocused areas. It is good practice to keep this value as high as possible, as otherwise sampling artifacts might occur, especially when the **Max Blur Distance** is big. Bigger Aperture values will automatically downsample the image to produce a better defocus. |

| | |
|---|---|
| **Defocus Type** | Algorithm used to produce defocused areas. **DX11** is effectively a bokeh splatting technique while **DiscBlur** indicates a more traditional (scatter as gather) based blur. |
| **Sample Count** | Amount of filter taps. Greatly affects performance. |

| | |
|---|---|
| **Max Blur Distance** | Max distance for filter taps. Affects texture cache and can cause undersampling artifacts if value is too big. A value smaller than 4.0 should produce decent results. |
| **High Resolution** | Perform defocus operations in full resolution. Affects performance but might help reduce unwanted artifacts and produce more defined bokeh shapes. |

| | |
|---|---|
| **Near Blur** | Foreground areas will overlap at a performance cost. |
| **Overlap Size** | Increase foreground overlap dilation if needed. |

**DX11 Bokeh Settings**

| | |
|---|---|
| **Bokeh Texture** | Texture defining bokeh shape. |
| **Bokeh Scale** | Size of bokeh texture. |
| **Bokeh Intensity** | Blend strength of bokeh shapes. |
| **Min Luminance** | Only pixels brighter than this value will cast bokeh shapes. Affects performance as it limits overdraw to a more reasonable amount. |
| **Spawn Heuristic** | Bokeh shapes will only be cast if pixel in questions passes a frequency check. A threshhold around 0.1 seems like a good tradeoff between performance and quality. |

### Comparison between DirectX11 and DiscBlur settings

*Smooth transitions are possible with the high resolution DX11 version (albeit at a high performance cost).*

*Due to the nature of the standard DiscBlur texture sampling approach, the maximum blur radius is limited before sampling artifacts become appearant. Also, only spherical Bokeh shapes are possible.*

### About DirectX 11 Bokeh Splatting

This powerful technique enables proper Scattering, however due to high demands on fillrate, it should be used with care. The parameters **Spawn Heuristic** and **Min Luminance** control when and where Bokeh Sprites will be placed. If pixels don't pass a luminance and frequency check, a simple Box Blur will be used instead. It's however hard to notice as it uses the same kernel width as the Bokeh sprites.

The following pictures show that the road, that is neither bright nor bears great frequency changes can just be blurred with a simple box filter without ruining the overall **Bokeh** experience.

*Example with small **Max Blur Distance***

*Example with big **Max Blur Distance***

### Hardware support

This effect requires a graphics card with pixel shaders (3.0) or OpenGL ES 2.0. Additionally, depth texture support is required. PC: NVIDIA cards since 2004 (GeForce 6), AMD cards since 2005 (Radeon X1300), Intel cards since 2006 (GMA X3000); Mobile: OpenGL ES 2.0 with depth texture support; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2012-09-26

---

## script-NoiseAndGrain

The **Noise And Grain** image effect simulates noise and film grain which is a typical effect happening in film or photography. This special noise implementation can even be used to enhance image contrast as it's using a special blend mode. It also enables typical noise scenarios, such as as low level light noise or softening glowing halo's or bloom borders.

The DirectX 11 implementation is totally independent of any texture reads and thus a good fit for modern graphics hardware.

The standard version uses a noise texture that should have an average luminance of 0.5 to prevent unwanted brightness changes of the resulting image. The used default texture is an example for this.

*Example screenshot of the effect. Notice its smoothness, how it sticks mostly to bright and dark areas and that it has a distinct blue tint.*

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

### Properties

| | |
|---|---|
| **DirectX11 Grain** | Enable high quality noise and grain (DX11 only). |
| **Monochrome** | Use greyscale noise only. |
| **Intensity Multiplier** | Global intensity adjustment. |
| **General** | Add noise equally for all luminance ranges. |
| **Black Boost** | Add extra low luminance noise. |
| **White Boost** | Add extra high luminance noise. |
| **Mid Grey** | Defines ranges for high-level and low-level noise ranges above. |
| **Color Weights** | Additionally tint noise. |

| | |
|---|---|
| **Texture** | Texture used for non-DX11 mode. |
| **Filter** | Texture filtering. |
| **Softness** | Defines noise or grain crispness. Higher values might yield better performance but require temporary a render target. |

**Advanced**

| | |
|---|---|
| **Tiling** | Noise pattern tiling (can be tweaked for all color channels individually when in non-DX11 texture mode). |

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2012-09-05

---

## script-ScreenOverlay

The **Screen Overlay** image effect introduces an easy way to blend different kinds of textures over the entire screen to create custom looks or effects.

*Example using overlay to create a low quality camera **light leak** effect*

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

### Properties

| | |
|---|---|
| **Blend Mode** | Blend mode used when applying texture. |
| **Intensity** | Strength or opacity the overlay texture will be applied with. |
| **Texture** | Overlay texture itself. |

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

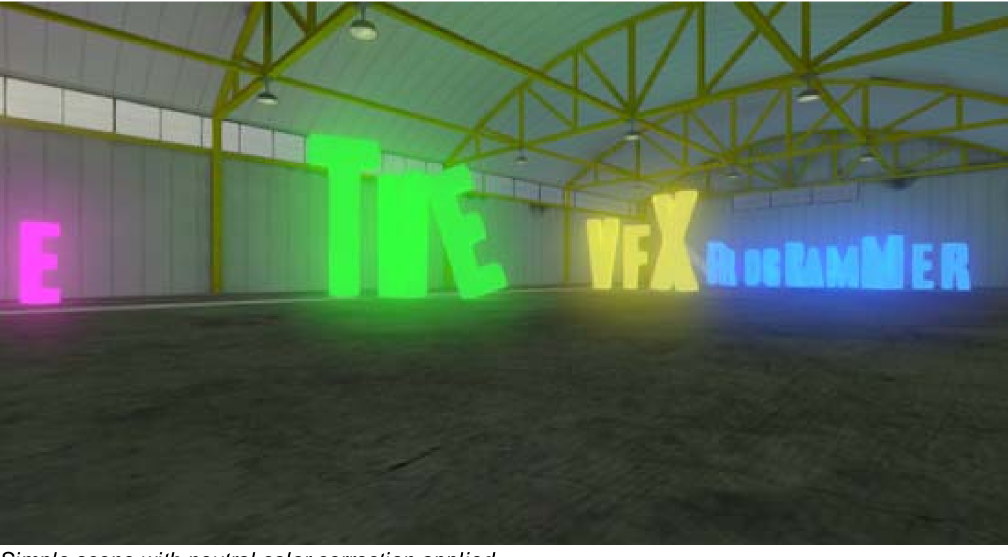
Page last updated: 2012-09-04

---

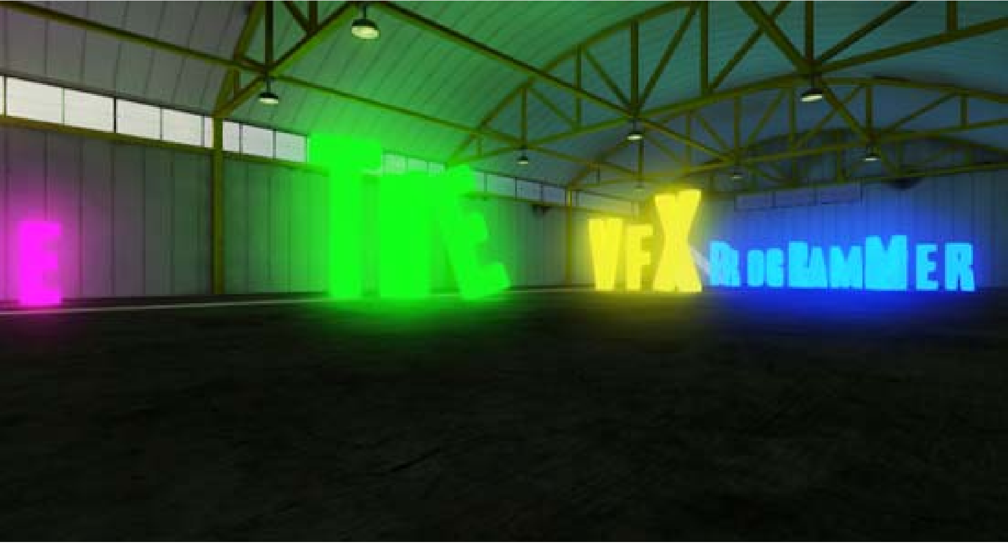
## script-ColorCorrectionLut

**Color Correction Lut** (**Lut** stands for *lookup texture*) is an optimized way of performing color grading in a post effect. Instead of tweaking the individual color channels via curves as in Color Correction Curves, only a single texture is used to produce the corrected image. The lookup will be performed by using the original image color as a vector that is used to address the lookup texture.

Advantages include better performance and more professional workflow opportunities, where all color transforms can be defined in professional image manipulation software (such as Photoshop or Gimp) and thus a more precise result can be achieved.



*Simple scene with neutral color correction applied.*



*Same scene using the included "ContrastEnhanced" lookup texture.*

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

## Properties

| | |
|---|---|
| **Based On** | A 2D representation of the 3D lookup texture that will be used to generate the corrected image. |

## Lookup Texture Requirements

The 2D texture representation is required to be laid out in a certain way that it represents an unwrapped volume texture (imagine an image sequence of "depth slices").

The following image shows an example of such an unwrapped texture which effectively enhances image contrast. It should be included in the standard packages.

*The image shows a texture of the dimension 256x16, yielding a 16x16x16 color lookup texture (lut). If the resulting quality is too low, a 1024x32 texture might yield better results (at the cost of memory).*

Texture importer requirements include enabling Read/Write support and disabling texture compression. Otherwise, unwanted image artifcats will likely occur.

## Example Workflow

Always keep the basic neutral lookup texture (lut) ready as this will be the basis for generating all other corrective lut's.

- Take a screenshot of your game
- Import into e.g. Photoshop and apply color adjustments (such as contrast, brightness, color levels adjustments) until a satisfying result has been reached
- Perform the same steps to the neutral lut and save as a new lut
- Assign new lut to the effect and hit **Convert & Apply**

### Hardware support

This effect requires a graphics card with pixel shaders (3.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2004 (GeForce 6), AMD cards since 2005 (Radeon X1300), Intel cards since 2006 (GMA X3000); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2012-09-12

---

## script-BloomAndLensFlares

**Blooming** is the optical effect where light from a bright source (such as a glint) appears to leak into surrounding objects. The **Bloom and Lens Flares** image effect adds bloom and also automatically generates lens flares in a highly efficient way. Bloom is a very distinctive effect that can make a big difference to a scene and may suggest a magical or dreamlike environment especially when used in conjunction with HDR rendering. Bloom and Lens Flares is actually an enhanced version of the Glow image effect which offers greater control over the bloom at the expense of rendering performance.

Note that this version is deprecated: A more flexible Bloom effect has been introduced with 4.0.

*Example showing how **Bloom and Lens Flares** can give a soft glow using the new **Screen** blend mode. The new anamorphic lens flare type helps evoke a cinematic feeling.*

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

### Properties

| | |
|---|---|
| **Tweak Mode** | Choose complex mode for additional options such as lens flares. |
| **Blend mode** | The method used to add bloom to the color buffer. The softer **Screen** mode is better for preserving bright image details but doesn't work with HDR. |
| **HDR** | Whether bloom is using HDR buffers. This will result in a different look as pixel intensities may leave the [0,1] range, see details in tonemapping and HDR. |
| **Cast lens flares** | Enable or disable automatic lens flare generation. |

| | |
|---|---|
| **Intensity** | The global light intensity of the added light (affects bloom and lens flares). |
| **Threshhold** | Regions of the image brighter than this threshold receive blooming (and potentially lens flares). |
| **Blur iterations** | The number of times gaussian blur is applied. More iterations improve smoothness but take extra time to process and hide small frequencies. |
| **Blur spread** | The max radius of the blur. Does not affect performance. |
| **Use alpha mask** | The degree to which the alpha channel acts as a mask for the bloom effect. |

| | |
|---|---|
| **Lens flare mode** | The type of lens flare. The options are **Ghosting**, **Anamorphic** or a mix of the two. |
| **Lens flare mask** | Mask used to prevent lens flare artifacts at screen edges. |
| **Local intensity** | Local intensity used only for lens flares. |
| **Local threshold** | The accumulative light intensity threshold that defines which image parts are candidates for lens flares. |
| **Stretch width** | The width for anamorphic lens flares. |
| **Blur iterations** | The number of times blurring is applied to anamorphic lens flares. More iterations improve smoothness but take more processing time. |
| **Tint Color** | Color modulation for the anamorphic flare type. |
| **1st-4th Color** | Color modulation for all lens flares when **Ghosting** or **Combined** is chosen. |

### Blend Modes: Add and Screen

Blend modes determine the way that two images will be combined when overlaid. Each pixel from the base image is combined mathematically with the pixel in the corresponding position in the overlay image. Two blend modes are available for Unity image effects, Add and Screen.

#### Add Mode

When the images are blended in Add mode, the values of the color channels (red, green and blue) are simply added together and clamped to the maximum value of 1. The overall effect is that areas of each image that aren't especially bright can easily blend to maximum brightness in the result. The final image tends to lose color and detail and so Add mode is useful when a dazzling "white out" effect is required.

#### Screen Mode

Screen mode is so named because it simulates the effect of projecting the two source images onto a white screen simultaneously. Each color channel is combined separately but identically to the others. Firstly, the channel values of the two source pixels are inverted (ie, subtracted from 1). Then, the two inverted values are multiplied together and the result is inverted. The result is brighter than either of the two source pixels but it will be at maximum brightness only if one of the source colors was also. The overall effect is that more color variation and detail from the source images is preserved, leading to a gentler effect than Add mode.

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2012-11-16

---

## script-ColorCorrectionCurves

**Color Correction Curves** make color adjustments using curves for each color channel. Depth based adjustments allow you to vary the color adjustment according to a pixel's distance from the camera. For example, objects on a landscape typically get more desaturated with distance due to the effect of particles in the atmosphere scattering.

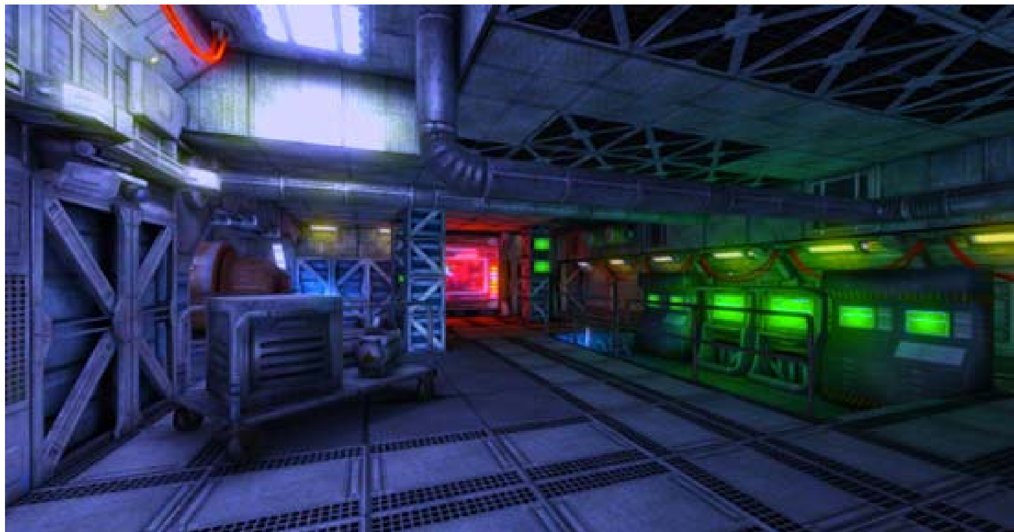
Selective adjustments can also be applied, so you can swap a target color in the scene for another color of your own choosing.

Lastly, Saturation is an easy way to adjust all color saturation or desaturation (until image turns black & white) which is an

effect that is not achievable with curves only.

See also the new Color Correction Lut effect for lookup texture based color grading.

The following images demonstrate how by simply enhancing the saturation slider and the blue channel curve can make a scene drastically different



As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

## Properties

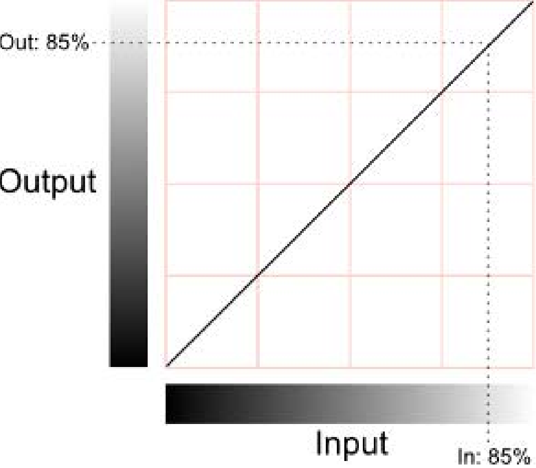
| | |
|---|---|
| **Mode** | Chose between advanced or simple configuration modes. |
| **Saturation** | Saturation level (0 creates a black & white image). |
| **Red** | The red channel curve. |
| **Green** | The green channel curve. |
| **Blue** | The blue channel curve. |

| | |
|---|---|
| **Red (Depth)** | The red channel curve for depth based correction. |
| **Green (Depth)** | The green channel curve for depth based correction. |
| **Blue (Depth)** | The blue channel curve for depth based correction. |
| **Blend Curve** | Defines how blending between the foreground and background color correction is performed. |

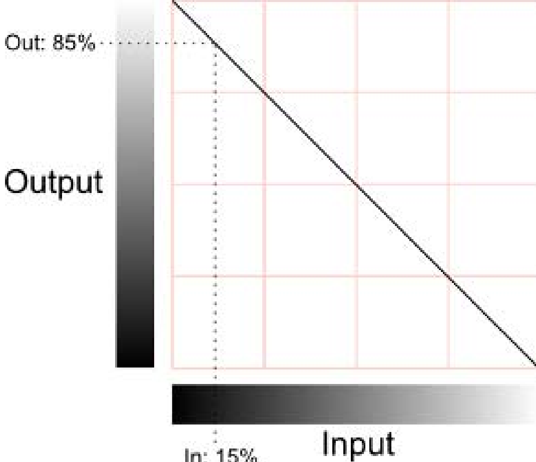
| **Selective Color Correction** | |
|---|---|
| **Enable** | Enables the optional selective color correction. |
| **Key** | The key color for selective color correction. |
| **Target** | The target color for selective color correction. |

### Understanding Curves

Curves offer a powerful way to enhance an image and can be used to increase or decrease contrast, add a tint or create psychedelic color effects. Curves work on each of the red, green and blue color channels separately and are based around the idea of mapping each input brightness level (ie, the original brightness value of a pixel) to an output level of your choosing. The relationship between the input and output levels can be shown on a simple graph:-




The horizontal axis represents the input level and the vertical represents the output level. Any point on the line specifies the output value that a given input is mapped to during processing. When the "curve" is the default straight line running from bottom-left to top-right, the input value is mapped to an identical output value, which will leave the pixel unchanged. However, the curve can be redrawn to re-map the brightness levels as required. A simple example is when the line goes diagonally from top-left to bottom-right:-

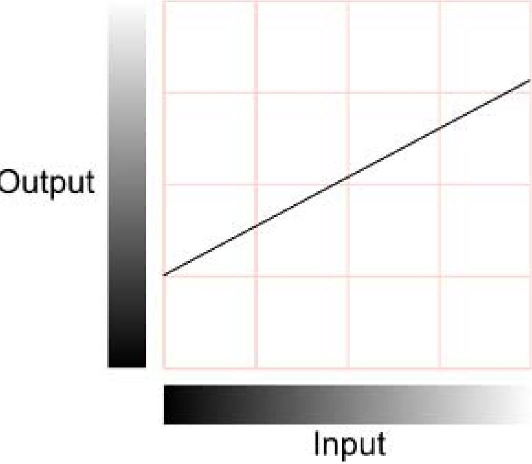



In this case, the pixel's brightness will be inverted; 0% will map to 100%, 25% to 75% and vice versa. If this is applied to all color channels then the image will be like a photographic negative of the original.

**Contrast**

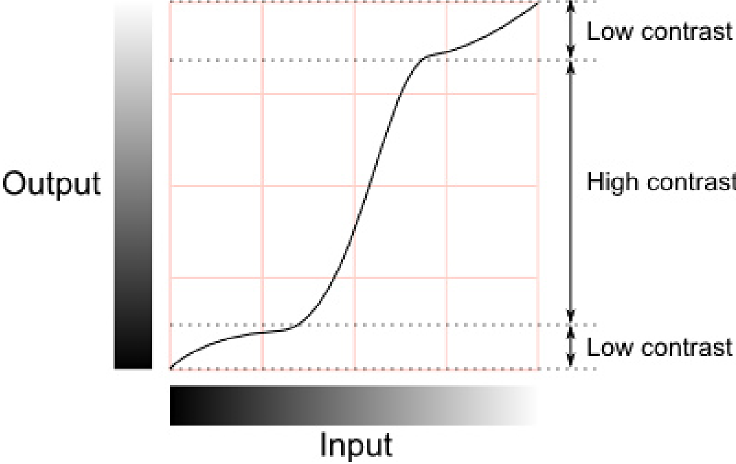
Most of the detail in an image is conveyed by the difference in brightness levels between pixels, regardless of their colour. Pixels that differ by less than about 2% brightness are likely to be indistinguishable but above this, the greater the difference, the greater the impression of detail. The spread of brightness values in the image is referred to as its contrast.

If a shallow slope is used for the curve, rather than the corner-to-corner diagonal then the full range of input values will be squeezed into a narrower range of output values:-



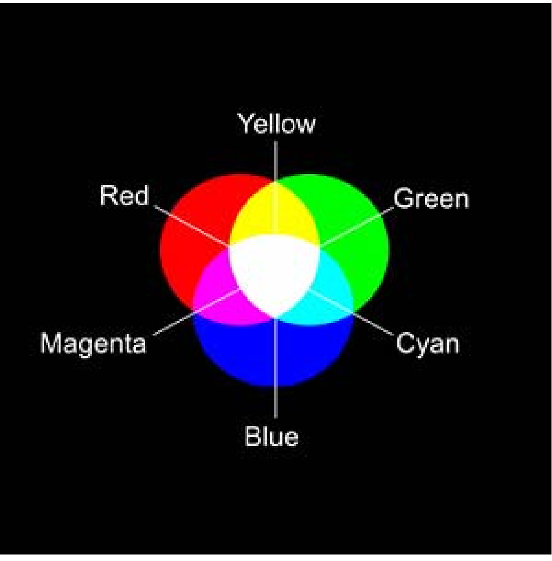

This has the effect of reducing the contrast, since the differences between pixel values in the output are necessarily smaller than those in the input (indeed, two slightly different input values may actually get mapped to the same output value). Note that since the image no longer spans the full range of output values, it is possible to slide the curve up and down the range, resulting in an image which is brighter or darker overall (the average brightness is sometimes called the "sit" point and is the parameter adjusted by the brightness control on a TV set). Reduced contrast can give the impression of gloom, fog or a dazzling light source in a scene, depending on the overall brightness.

It is not necessary to reduce the contrast across the whole range of brightness levels. The curve's slope can vary along its length, with the shallower parts corresponding to ranges of reduced contrast. In between the shallow parts, the slope may be steeper than the default, and the contrast will actually increase in these ranges. Changing the curve like this gives a useful way to increase contrast in some parts of the image while reducing it in areas where the detail is less important:-




**Colour Effects**

If the curves are set identically for each color channel (red, green and blue) then the changes will mainly affect the brightness of pixels while their colors remain relatively unchanged. However, if the curves are set differently for each channel then the colors can change dramatically. Many complex interactions between the color channels are possible but some basic insight can be gained from the following diagram:-

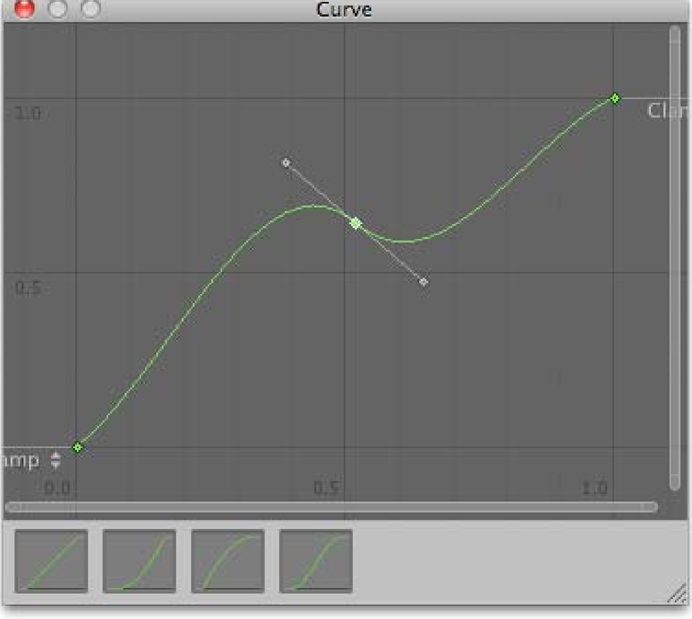



As explained in the section above, a reduction of contrast can accompany an increase or reduction in the overall brightness. If, say, the red channel is brightened then a red tint will be visible in the image. If it is darkened then the image will be tinted towards cyan (since this color is obtained by combining the other two primaries, green and blue).

**Depth-Based Color Correction**
Colors often appear slightly different when viewed at a distance. For example, in a landscape scene, colors tend to get desaturated by atmospheric light scattering. This kind of effect can be created using depth-based color correction. When this is enabled, two sets of color curves become available, one for the camera's near clipping plane and the other for the far clipping plane. The actual correction applied to an object depends on its distance from the camera; the normalized distance between the two clipping planes is used as an interpolation parameter between the two sets of color curves. The exact type of interpolation is specified by an additional blending curve, which maps the normalised distance to an interpolation value in much the same way that a color curve maps an input to an output. By default, this curve is a straight diagonal which results in linear interpolation between the two color corrections. However, it can be modified to bias the correction according to distance.

**Selective Color Correction**
Using this setting, it is possible to replace a particular color in the original image (referred to as the "key") and replace it with a chosen target color. Using a single exact color for the key would tend to introduce visual artifacts and so a range is used instead. The resulting color is an interpolation between the key and target colors, depending on how close the original image pixel is to the specified key color.

## Editing Curves
Clicking on one of the curves in the inspector will open an editing window:-




At the bottom of the window are a number of presets for common curves. However, you can also alter the curve by manipulating the key points. Right clicking on the curve line will add a new key point which can be dragged around with the mouse. If you right click on one of the points, you will see a contextual menu that gives several options for editing the curve. As well as allowing you to delete the key, there are four options that determine how it will affect the shape of the curve:-

- Auto: the curve will pass through the point and its shape will be adjusted to keep the curvature smooth between neighbouring points.
- Free Smooth: the tangent of the curve can be edited using handles attached to the key point.
- Flat: Free Smooth mode is enabled and the tangent is set horizontally.
- Broken: The key point has tangent handles as with Free Smooth mode but the handles on the left and right of the curve can be moved separately to create a sharp break rather than a smooth curve.

Below these options are a few settings that control how a point's tangent handles behave:-

- Free: Broken mode is enabled for the curve at the specified tangent.
- Linear: The curve between the key point and its neighbour is set to a straight line.
- Constant: A flat horizontal line is drawn from the curve to its neighbour and the vertical displacement occurs as a sharp step.

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. Additionally, depth texture support is required. PC: NVIDIA cards since 2004 (GeForce 6), AMD cards since 2004 (Radeon 9500), Intel cards since 2006 (GMA X3000); Mobile: OpenGL ES 2.0 with depth texture support; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2012-11-16

---

# script-ContrastEnhance

The **Contrast Enhance** image effect enhances the impression of contrast for a given camera. It uses the well-known **unsharp mask** process available in image processing applications.

When blurring is applied to an image, the colors of adjacent pixels are averaged to some extent, resulting in a reduction of sharp edge detail. However, areas of flat color remain relatively unchanged. The idea behind unsharp masking is that an

image is compared with a blurred (or "unsharp") version of itself. The difference in brightness between each pixel in the original and the corresponding pixel in the blurred image is an indication of how much constrast the pixel has against its neighbours. The brightness of that pixel is then changed in proportion to the local contrast. A pixel which is darker after blurring must be brighter than its neighbours, so its brightness is further increased while if the pixel is darker after blurring then it will be darkened even more. The effect of this is to increase contrast selectively in areas of the image where the detail is most noticeable. The parameters of unsharp masking are the pixel radius over which colors are blurred, the degree to which brightness will be altered by the effect and a "threshold" of contrast below which no change of brightness will be made.

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

*No Contrast Enhance*

*Contrast enhance enabled*

## Properties

| | |
|---|---|
| **Intensity** | The intensity of contrast enhancement. |
| **Threshhold** | The constrast threshold below which no enhancement is applied. |
| **Blur Spread** | The radius over which contrast comparisons are made. |

## Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2011-09-22

---

## script-Crease

The **Crease** is a common non-photorealistic (NPR) rendering technique that enhances the visibility of objects by adding outlines of variable thickness.

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

*Crease shading creates depth based outlines.*

### Properties

| | |
|---|---|
| **Intensity** | The intensity of the crease shading. |
| **Softness** | The smoothness and softness of the applied crease shading. |
| **Spread** | The blur radius, which also determines the thickness of outlines. |

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. Additionally, depth texture support is required. PC: NVIDIA cards since 2004 (GeForce 6), AMD cards since 2004 (Radeon 9500), Intel cards since 2006 (GMA X3000); Mobile: OpenGL ES 2.0 with depth texture support; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2011-09-21

---

## script-DepthOfField34

**Depth of Field 3.4** is a common postprocessing effect that simulates the properties of a camera lens. The name refers to the fact that the effect was added in Unity 3.4, but now is superseded by a more modern Depth Of Field Scatter effect which uses optimized techniques to simulate lens blurs and enables better transitions between focal areas. However, depending on the use case, performance might be a lot better in the old 3.4 version as it was developed for older hardware.

In real life, a camera can only focus sharply on an object at a specific distance; objects nearer or farther from the camera will be somewhat out of focus. The blurring not only gives a visual cue about an object's distance but also introduces **bokeh** which is the term for pleasing visual artifacts that appear around bright areas of the image as they fall out of focus.

An example of the new Depth of Field effect can be seen in the following images, displaying the results of a defocused foreground and a defocused background. Notice how the foreground blur overlaps with the rest while the background doesn't.

*Only the nearby pipes are in the focal area*

*Foreground vs Background blurring with Depth of Field*

You might also consider using the Tilt Shift effect for a more straightforward but less sophisticated depth-of-field effect.

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

## Properties

**General Settings**

| | |
|---|---|
| **Resolution** | Determines the internal render target sizes. A lower resolution will result in faster rendering and lower memory requirements. |
| **Quality** | The quality level. Choose between the faster **OnlyBackground** or the higher-quality **BackgroundAndForeground** which calculates the depth-of-field defocus for both areas separately. |
| **Simple tweak** | Switches to a simpler focal model. |
| **Visualize focus** | This shows the focal plane in the game view to assist learning and debugging. |
| **Enable bokeh** | This will generate more realistic lens blurs where very bright parts are scaled and overlap. |

**Focal Settings**

| | |
|---|---|
| **Focal distance** | The distance to the focal plane from the camera position in world space. |
| **Object Focus** | Determine the focal distance using a target object in the scene. |
| **Smoothness** | The smoothness when transitioning from out-of-focus to in-focus areas. |
| **Focal size** | The size of the in-focus area. |

**Blur**

| | |
|---|---|
| **Blurriness** | How many iterations are used when blurring the various buffers (each iteration requires processing time). |
| **Blur spread** | The blur radius. This is resolution-independent, so you may need to readjust the value for each required resolution. |

**Bokeh Settings**

| | |
|---|---|
| **Destination** | Enabling foreground and background blur increases rendering time but gives more realistic results. |
| **Intensity** | Blend intensity used as bokeh shapes are being accumulated. This is a critical value that always needs to be carefully adjusted. |

| | |
|---|---|
| **Min luminance** | The luminance threshold below which pixels will not have bokeh artifacts applied. |
| **Min contrast** | The contrast threshold below which pixels will not have bokeh artifacts applied. The significance of this is that you usually only need bokeh shapes in areas of high frequency (ie, cluttered or "noisy" areas of image) since they are otherwise nearly invisible. Performance will be improved if you use this parameter to avoid generating unnecessary bokeh artifacts. |
| **Downsample** | The size of the internal render target used for accumulating bokeh shapes. |
| **Size** | The maximum bokeh size. Will be modulated by the amount of defocus (Circle of Confusion). |
| **Bokeh Texture** | The texture defining the bokeh shapes. |

Note that since the bokeh effect is created by drawing triangles per pixel, it can drastically affect your framerate, especially if it's not adjusted optimally. Adjust the **Size**, **Min luminance**, **Min contrast**, **Downsample** and **Resolution** to improve performance. Also, since the screen is darkened before the bokeh shapes are applied, you should use an appropriate **Blurriness** level to remove possible artefacts.

### Hardware support

This effect requires a graphics card with pixel shaders (3.0) or OpenGL ES 2.0. Additionally, depth texture support is required. PC: NVIDIA cards since 2004 (GeForce 6), AMD cards since 2005 (Radeon X1300), Intel cards since 2006 (GMA X3000); Mobile: OpenGL ES 2.0 with depth texture support; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2012-11-16

---

## script-Tonemapping

**Tonemapping** is usually understood as the process of mapping color values from HDR (high dynamic range) to LDR (low dynamic range). In Unity, this means for most platforms that arbitrary 16-bit floating point color values will be mapped to traditional 8-bit values in the [0,1] range.

Note that Tonemapping will only properly work if the used camera is HDR enabled. It is also recommended to give light sources higher than normal intensity values to make use of the bigger range. Just as in real life there is huger differences in Luminance and our eyes or any capturing medium is only able to sample a certain range of that.

Tonemapping works well in conjunction with the HDR-enabled Bloom image effect. Make sure that Bloom should be applied before Tonemapping as otherwise all high ranges will be lost. Generally, any effect that can benefit from higher luminances should be scheduled before the Tonemapper (one more example being the Depth of Field image effect).

There are different ways on how to map intensities to LDR (as selected by **Mode**). This effect provides several techniques, two of them being adaptive (**AdaptiveReinhard** and **AdaptiveReinhardAutoWhite**), which means that color changes are carried out delayed as the change in intensities is fully registered. Cameras and the human eye have this effect. This enables interesting dynamic effects such as a simulation of the natural adaption happening when entering or leaving a dark tunnel into bright sunlight.

The following two screenshots show Photographic Tonemapping with different exposure values. Note how banding is avoided by using HDR cameras.

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

## Properties

| | |
|---|---|
| **Mode** | Choose the desired tonemapping algorithm. |

| | |
|---|---|
| **Exposure** | Simulated exposure, defining the actual range of luminances. |
| **Average grey** | Average grey value of the scene that defines the intensity of the result. |
| **White** | Smallest value that will be mapped white. |
| **Adaption speed** | Adjustment speed for all adaptive tonemappers. |
| **Texture size** | Size of the internal texture for all adaptive tonemappers. Bigger values capture more details when calculating the new intensity and lower performance. |

## Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2012-11-16

## script-EdgeDetectEffectNormals

This version of the **Edge Detect** image effect creates outlines around edges by taking the scene geometry into account. Edges are not determined by colour differences but by the surface normals and distance from camera of neighbouring pixels (the surface normal is an "arrow" that indicates the direction the surface is facing at a given pixel position). Generally, where two adjacent pixels have significantly different normals and/or distances from the camera, there is an edge in the scene.

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

*Example Edge Detection. Note how edge outlines can be smoothed out by adding an Antialiasing effect to follow Edge Detection.*

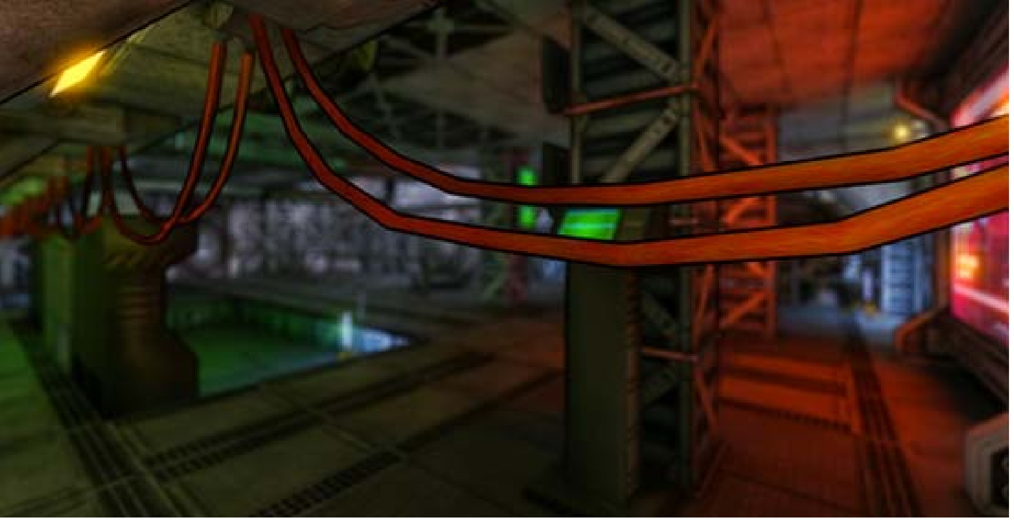
This effect uses the **ImageEffectOpaque** attribute which enables image effects to be executed before the transparent render passes. By default, image effects are executed after both opaque and transparent passes have been fully rendered.

### Properties

| | |
|---|---|
| **Mode** | Chose the filter type (see below). |
| **Depth Sensitivity** | The minimum difference between the distances of adjacent pixels that will indicate an edge. |
| **Normals Sensitivity** | The minimum difference between the normals of adjacent pixels that will indicate an edge. |
| **Sampling Distance** | Bigger sampling distances (default is 1.0) create thicker edges but also introduce haloing artifacts. |
| **Edges exponent** | Exponent used for Sobel filter. Smaller values detect smaller depth differences as edges. |
| **Background options** | |
| **Edges only** | Blend the background with a fixed color. |
| **Background** | The color used when **Edges only** is > 0. |

### Filter Types

The new **SobelDepthThin** filter offers a way to make edge detection work with other depth based image effects such as Depth of Field, Fog or Motion Blur as edges don't cross an object's silhouette:



*Edges don't leak into the defocused background and at the same time, the background blur doesn't remove the created edges.*

Note that as only depth is used for edge detection, this filter discards edges inside silhouettes.

**SobelDepth** works similarly but doesn't discard edges outside the silhouette of an object. Hence, the ede detection is more precise but doesn't work well with other depth-based effects.

**TriangleDepthNormals** is likely the cheapest available filter even though it examines both depth and normals to decide if a pixel resides on an edge, i.e. it detects more than just object silhouettes. A high amount of normal map details however might break this filter.

**RobertsCrossDepthNormals** shares its properties with the *Triangle* filter but looks at slightly more samples to determine edges. As a natural byproduct, the resulting edges tend to be thicker.

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. Additionally, depth texture support is required. PC: NVIDIA cards since 2004 (GeForce 6), AMD cards since 2004 (Radeon 9500), Intel cards since 2006 (GMA X3000); Mobile: OpenGL ES 2.0 with depth texture support; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2012-11-16

---

## script-Fisheye

The **Fisheye** image effect creates distorts the image as if viewed through a fisheye lens (although any lens will distort the image to some extent).

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

*Fisheye distorts the image at its corners. The use of image antialiasing techniques is highly recommended because of sampling artefacts.*

### Properties

| | |
|---|---|
| **Strength X** | The horizontal distortion. |
| **Strength Y** | The vertical distortion. |

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2011-09-19

---

## script-GlobalFog

The **Global Fog** image effect creates camera-based exponential fog. All calculations are done in world space which makes it possible to have height-based fog modes that can be used for sophisticated effects (see example).

*Example of global fog, demonstrating both distance and height based fog*

*Example of "cheating" at atmospheric effects using global fog*

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

### Properties

| | |
|---|---|
| **Fog Mode** | The available types of fog, based on distance, height or both |
| **Start Distance** | The distance at which the fog starts fading in, in world space units. |
| **Global Density** | The degree to which the **Fog Color** accumulates with distance. |
| **Height Scale** | The degree to which the fog density reduces with height (when height-based fog is enabled). |
| **Height** | The world space Y coordinate where fog starts to fade in. |
| **Global Fog Color** | The color of the fog. |

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. Additionally, depth texture support is required. PC: NVIDIA cards since 2004 (GeForce 6), AMD cards since 2004 (Radeon 9500), Intel cards since 2006 (GMA X3000); Mobile: OpenGL ES 2.0 with depth texture support; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2011-09-19

---

## script-SunShafts

The **Sun Shafts** image effect simulates the radial light scattering (also known as the "god ray" effect) that arises when a very bright light source is partly obscured.

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

*Example of the Sun Shafts effect*

## Properties

| | |
|---|---|
| **Rely on Z Buffer** | This option can be used when no depth textures are available or they are too expensive to calculate (eg, in forward rendering with a large number of objects). Note that if this option is disabled then Sun Shafts must be the very first image effect applied to the camera. |
| **Resolution** | The resolution at which the shafts are generated. Lower resolutions are faster to calculate and create softer results. |
| **Blend Mode** | Chose between the softer **Screen** mode and the simpler **Add** mode. |

| | |
|---|---|
| **Sun Transform** | The transform of the light source that casts the Sun Shafts. Only the position is significant. |
| **Center on ...** | Within the editor, position the **Sun Transform** object at the center of the game view camera. |

| | |
|---|---|
| **Shafts color** | The tint color of the shafts. |
| **Distance falloff** | The degree to which the shafts' brightness diminishes with distance from the **Sun Transform** object. |

| | |
|---|---|
| **Blur size** | The radius over which pixel colours are combined during blurring. |
| **Blur iterations** | The number of repetitions of the blur operation. More iterations will give smoother blurring but each has a cost in processing time. |

| | |
|---|---|
| **Intensity** | The brightness of the generated shafts. |
| **Use alpha mask** | Defines how much the alpha channel of the color buffer should be used when generating Sun Shafts. This is useful when your skybox has a proper alpha channel that defines a mask (eg, for clouds blocking the sun shafts). |

## Blend Modes: Add and Screen

Blend modes determine the way that two images will be combined when overlaid. Each pixel from the base image is combined mathematically with the pixel in the corresponding position in the overlay image. Two blend modes are available for Unity image effects, Add and Screen.

#### Add Mode

When the images are blended in Add mode, the values of the color channels (red, green and blue) are simply added together and clamped to the maximum value of 1. The overall effect is that areas of each image that aren't especially bright can easily blend to maximum brightness in the result. The final image tends to lose color and detail and so Add mode is useful when a dazzling "white out" effect is required.

#### Screen Mode

Screen mode is so named because it simulates the effect of projecting the two source images onto a white screen simultaneously. Each color channel is combined separately but identically to the others. Firstly, the channel values of the two source pixels are inverted (ie, subtracted from 1). Then, the two inverted values are multiplied together and the result is inverted. The result is brighter than either of the two source pixels but it will be at maximum brightness only if one of the source colors was also. The overall effect is that more color variation and detail from the source images is preserved, leading to a gentler effect than Add mode.

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. Additionally, depth texture support is required. PC: NVIDIA cards since 2004 (GeForce 6), AMD cards since 2004 (Radeon 9500), Intel cards since 2006 (GMA X3000); Mobile: OpenGL ES 2.0 with depth texture support; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2011-09-27

---

## script-TiltShift

**Tilt Shift** is a specialized version of the Depth of Field effect that allows for very smooth transitions between focused and defocused areas. It is is easier to use and is generally less prone to unsightly image artifacts. However, since it relies on dependent texture lookups, it can also have a higher processing overhead.

*Tilt shift example. Observe the overall smoothness the effect achieves.*

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

### Properties

**Focal Settings**

| | |
|---|---|
| **Visualize** | Visualizes the focal plane in the game view with a green tint (useful for learning or debugging). |
| **Distance** | The distance to the focal plane from the camera position in world space units. |
| **Smoothness** | The smoothness when transitioning from out-of-focus to in-focus areas. |

**Background Blur**

| | |
|---|---|
| **Downsample** | Downsamples most internal buffers (this makes the effect faster but more blurry). |
| **Iterations** | Number of iterations for blurring the background areas (ie, everything behind the focal plane). Each iteration requires processing time. |
| **Max Blur spread** | The maximum blur distance for the defocused areas. Makes out-of-focus areas increasingly blurred. |

**Foreground Blur**

| | |
|---|---|
| **Enable** | Enables foreground blurring. This typically gives a better result but with a cost in processing time. |
| **Iterations** | Number of iterations for blurring the foreground areas (ie, everything in front of the focal area). Each iteration requires processing time. |

### Hardware support

This effect requires a graphics card with pixel shaders (3.0) or OpenGL ES 2.0. Additionally, depth texture support is required.

PC: NVIDIA cards since 2004 (GeForce 6), AMD cards since 2005 (Radeon X1300), Intel cards since 2006 (GMA X3000); Mobile: OpenGL ES 2.0 with depth texture support; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2011-09-19

---

## script-Vignetting

The **Vignetting** image effect introduces darkening, blur and chromatic aberration (spectral color separation) at the edges and corners of the image. This is usually used to simulate a view through a camera lens but can also be used to create abstract effects.

*Example of Vignetting and chromatic Aberration. Notice how the screen corners darken and color separation (aberration) creates purple and slightly green color fringing.*

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

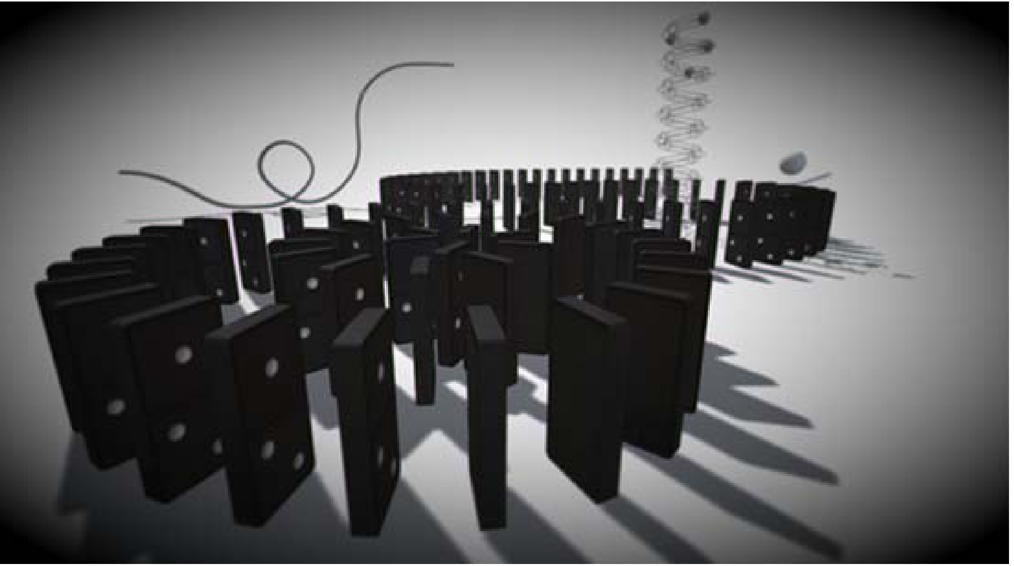
### Properties

| | |
|---|---|
| **Vignetting** | The degree of darkening applied to the screen edges and corners. Choose 0 to disable this feature and save on performance. |
| **Blurred Corners** | The amount of blur that is added to the screen corners. Choose 0 to disable this feature and save on performance. |
| **Blur Distance** | The blur filter sample distance used when blurring corners. |

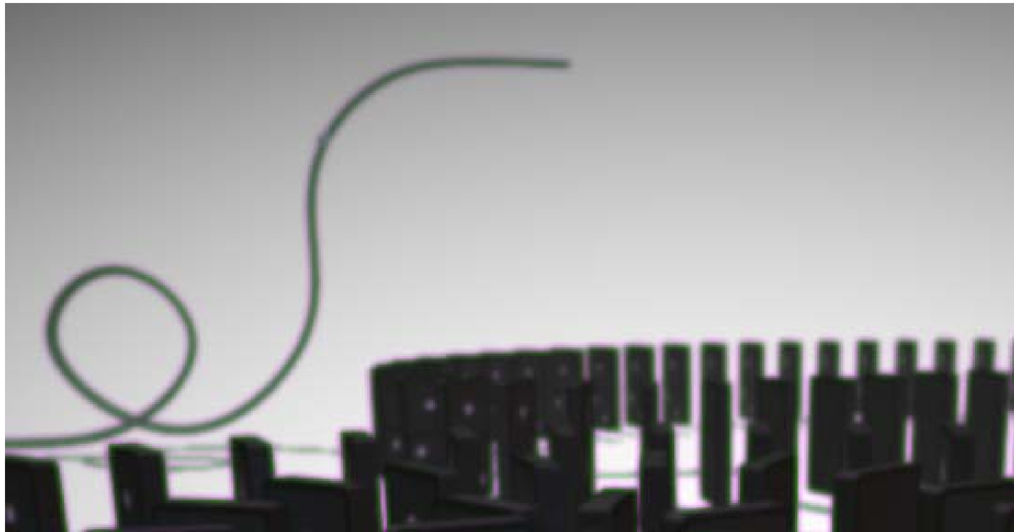
| | |
|---|---|
| **Aberration Mode** | **Advanced** tries to model more aberration effects (the constant axial aberration existant on the entire image plane) while **Simple** only models tangential aberration (limited to corners). |
| **Strength** | Overall aberration intensity (not to confuse with color offset distance), defaults to 1.0. |
| **Tangential Aberration** | The degree of tangential chromatic aberration: Uniform on the entire image plane. |
| **Axial Aberration** | The degree of axial chromatic aberration: Scales with smaller distance to the image plane's corners. |
| **Contrast Dependency** | The bigger this value, the more contrast is needed for the aberration to trigger. Higher values are more realistic (in this case, an HDR input is recommended). |

### Advanced Mode

Advanced mode is more expensive but offers a more realistic implementation of Chromatic Aberration.



*The **Advanced** mode offers great control over our model of chromatic aberration -- also known as green or purple color fringing -- a common phenomenon in photography (also see image below).*



*Closeup view of color fringing. Note how around areas of great contrast purple and green shimmers seem to appear. This effect depends depends on the kind of camera or lens system being used. The effect is based on the fact that different wavelengths will be projected on different focal planes.*

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

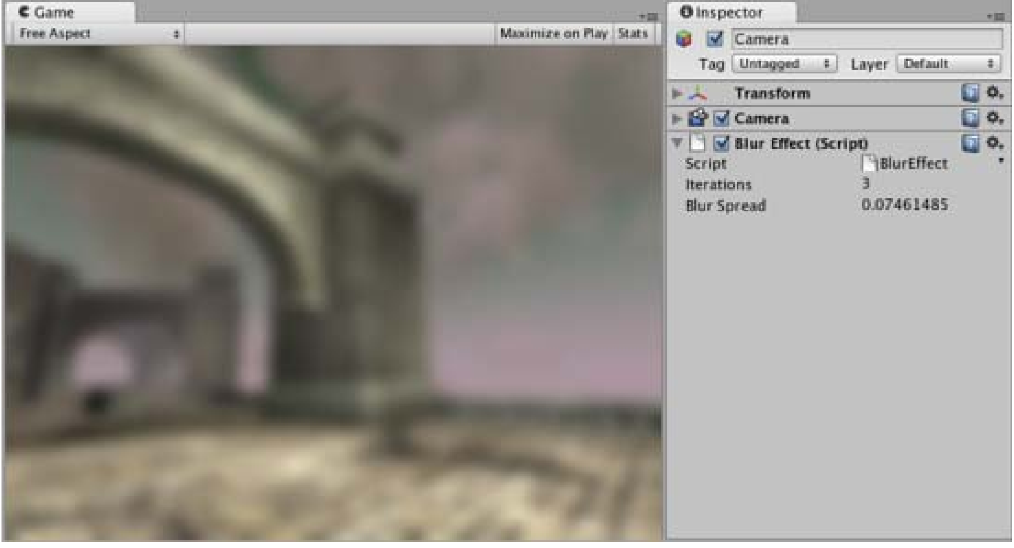
Page last updated: 2012-11-16

---

## script-BlurEffect

The **Blur** image effect blurs the rendered image in real-time.

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.



*Blur effect applied to the scene*

### Properties

| | |
|---|---|
| **Iterations** | The number of times the basic blur operation will be repeated. More iterations typically give a better result but each has a cost in processing time. |
| **Blur Spread** | Higher values will spread out the blur more at the same iteration count but at some cost in quality. Usually values from 0.6 to 0.7 are a good compromise between quality and speed. |

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

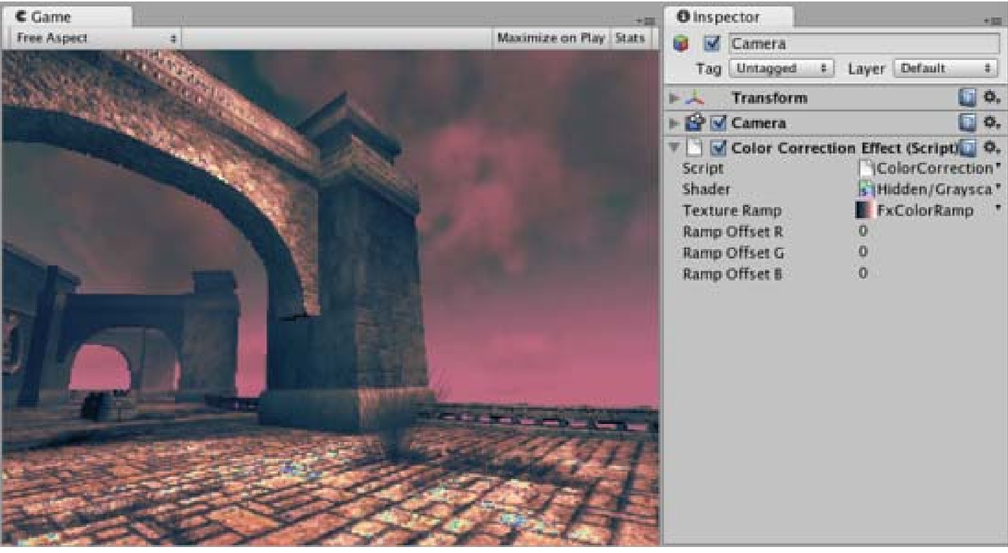
Page last updated: 2011-09-19

---

## script-ColorCorrectionEffect

**Color Correction** allows you apply arbitrary color correction to your scene as a postprocessing effect (just like the Curves tool in Photoshop or Gimp). This page explains how to setup color correction in Photoshop and then apply *exactly* the same color correction at runtime in Unity.

Like all image effects, Color Correction is only available in Pro version of Unity. Make sure to have the Pro Standard Assets installed.



*Color correction applied to the scene. Color ramp used (magnified) is shown at the right.*

*Color ramp used for the image above.*

### Getting color correction from Photoshop into Unity

1. Take a screenshot of a typical scene in your game
2. Open it in Photoshop and color correct using the **Image->Adjustments->Curves**
3. Save the **.acv** file file from the dialog using **Save...**
4. Open **Pro Standard Assets->Image Based->color correction ramp.png** in Photoshop
5. Now apply color correction to the ramp image: open **Image->Adjustments->Curves** again and load your saved **.acv** file
6. Select your camera in Unity and select **Component->Image Effects->Color Correction** to add color correction effect. Select your modified color ramp.
7. Hit Play to see the effect in action!

### Details

Color correction works by remapping the original image colors through the color ramp image (sized 256x1):

1. result.red = pixel's red value in ramp image at (original.red + **RampOffsetR**) index
2. result.green = pixel's green value in ramp image at (original.green + **RampOffsetG**) index
3. result.blue = pixel's blue value in ramp image at (original.blue + **RampOffsetB**) index

So for example, to invert the colors in the image you only need to flip the original color ramp horizontally (so that it goes from white to black instead of from black to white).

A simpler version of color remapping that only remaps based on luminance can be achieved with Grayscale image effect.

**Tips:**

- The color correction ramp image should not have mip-maps. Turn them off in **Import Settings**. It should also be set to **Clamp** mode.

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2011-05-12

## script-ContrastStretchEffect

**Contrast Stretch** dynamically adjusts the contrast of the image according to the range of brightness levels it contains. The adjustment takes place gradually over a period of time, so the player can be briefly dazzled by bright outdoor light when emerging from a dark tunnel, say. Equally, when moving from a bright scene to a dark one, the "eye" takes some time to adapt.

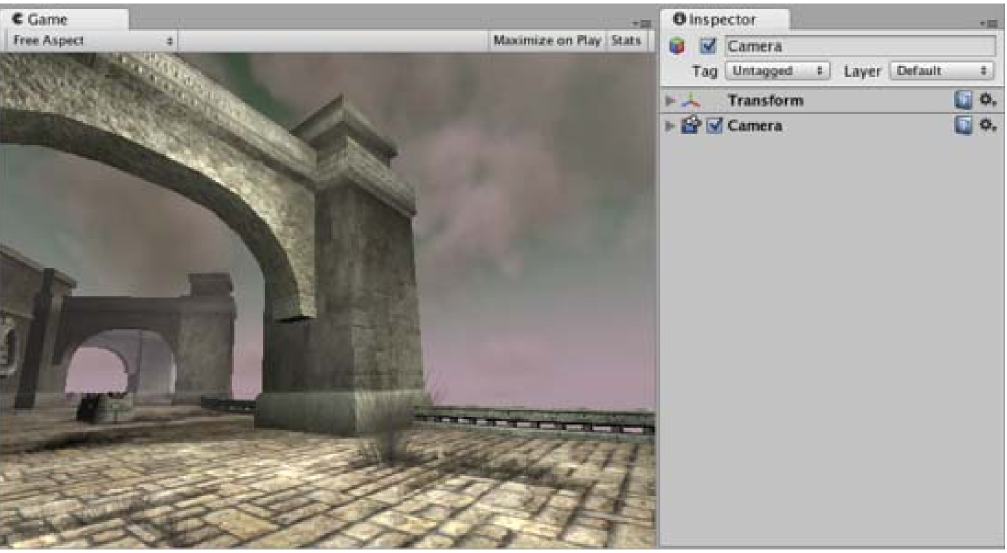
As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.

### Understanding Contrast Stretch

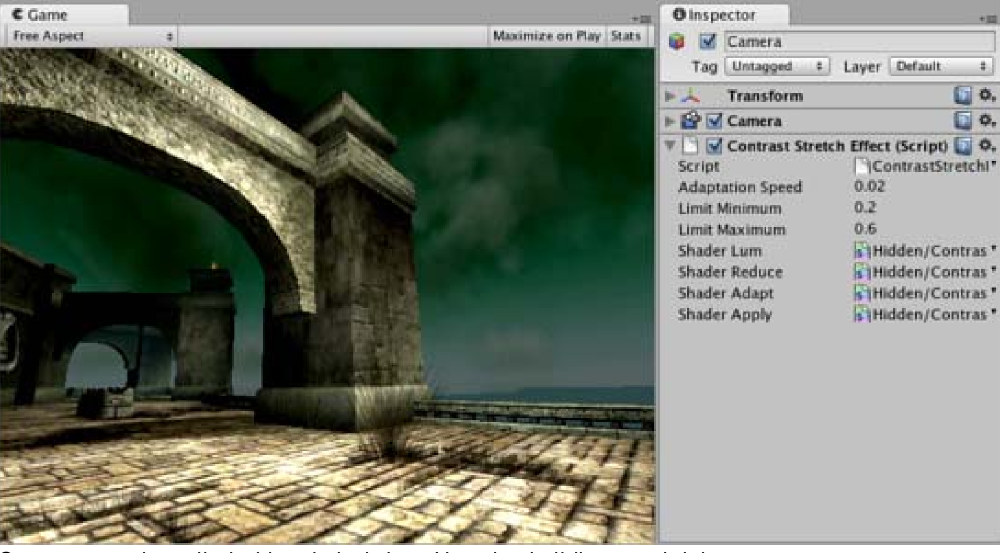
The clarity of detail in an image is largely determined by the range of different brightness values it contains. It is difficult for the eye to distinguish between two brightness levels that differ by less than about 2% and above that, the greater the difference, the stronger the detail. The overall separation between the lightest and darkest values in an image is referred to as the **contrast** of that image.

It is common for an image to use less than the full range of available brightness values. One way to increase the contrast is to redistribute the pixels' values so as to make better use of the range. The darkest level in the original image is remapped to a even darker level, the brightest to a brighter level and all the levels in between are moved farther apart in proportion. The distribution of levels is then "stretched" out farther across the available range and thus this effect is known as **contrast stretch**.

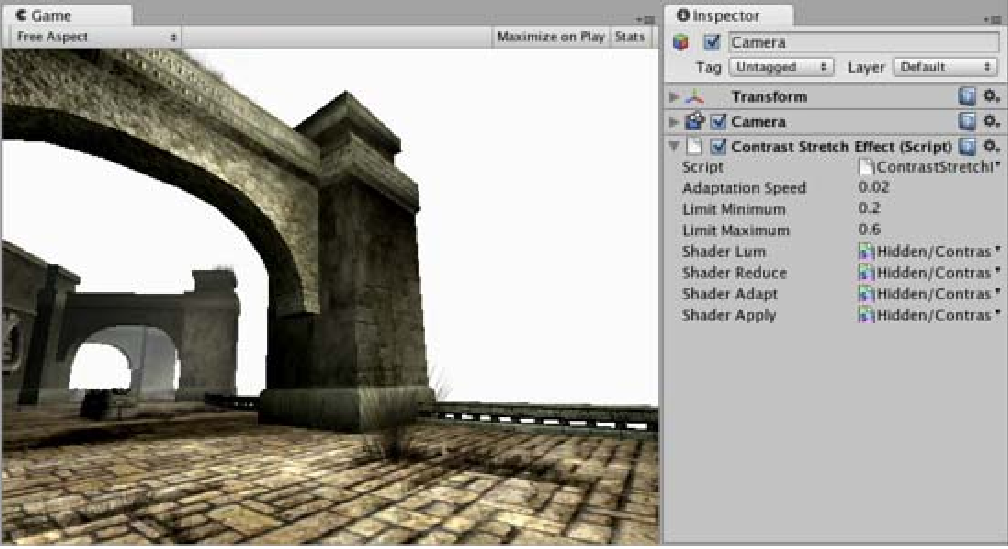
Contrast stretching is evocative of the way the eye adapts to different light conditions. When walking from an outdoor area to a dimly lit building, the view will briefly appear indistinct until the contrast is stretched to reveal the detail. When emerging from the building, the contrast stretch will have the effect of making the outdoor scene appear dazzling bright until the "eye" of the player adjusts.



*No Contrast Stretch applied.*



*Contrast stretch applied with a dark skybox. Note that buildings get brighter.*



*Contrast stretch applied with a very bright skybox. Note that buildings get darker.*

### Properties

| | |
|---|---|
| **Adaptation Speed** | The speed of the transition. The lower this number, the slower the transition |
| **Limit Minimum** | The darkest level in the image after adjustment. |
| **Limit Maximum** | The brightest level in the image after adjustment. |

**Tips:**

- Since Constrast Stretch is applied over a period of time, the full effect is only visible in Play mode.

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2011-09-20

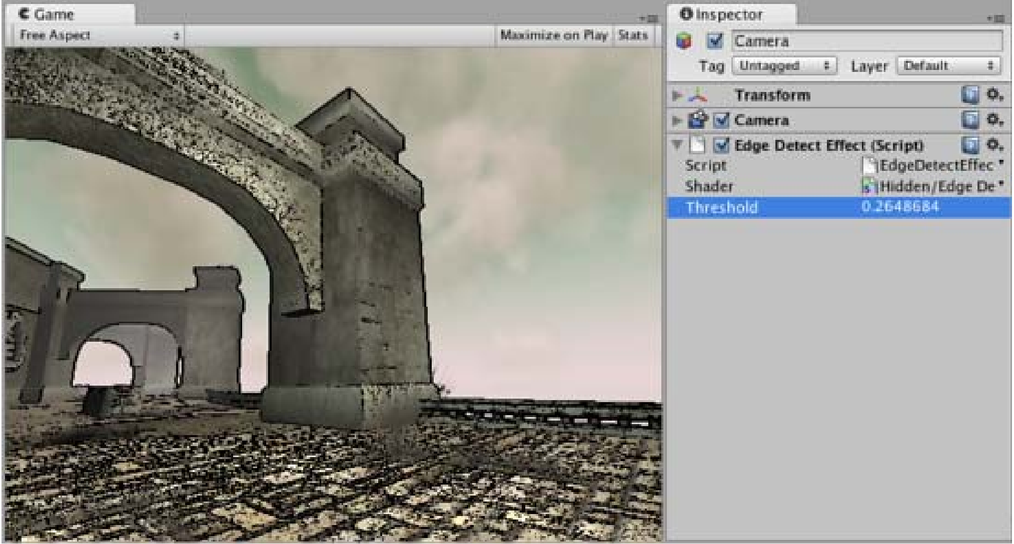
---

## script-EdgeDetectEffect

**Edge Detect** image effect adds black edges to the image wherever color differences exceed some threshold.

If more sophisticated geometry-based edge detection is required, the Standard Assets also provide such a normals and depth-based edge detection effect.

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.




*Edge Detect image effect applied to the scene*

| | |
|---|---|
| **Threshold** | Edges will be displayed wherever the color difference between neighboring pixels exceeds this value. Increasing the value will make edges less sensitive to texture or lighting changes. |

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

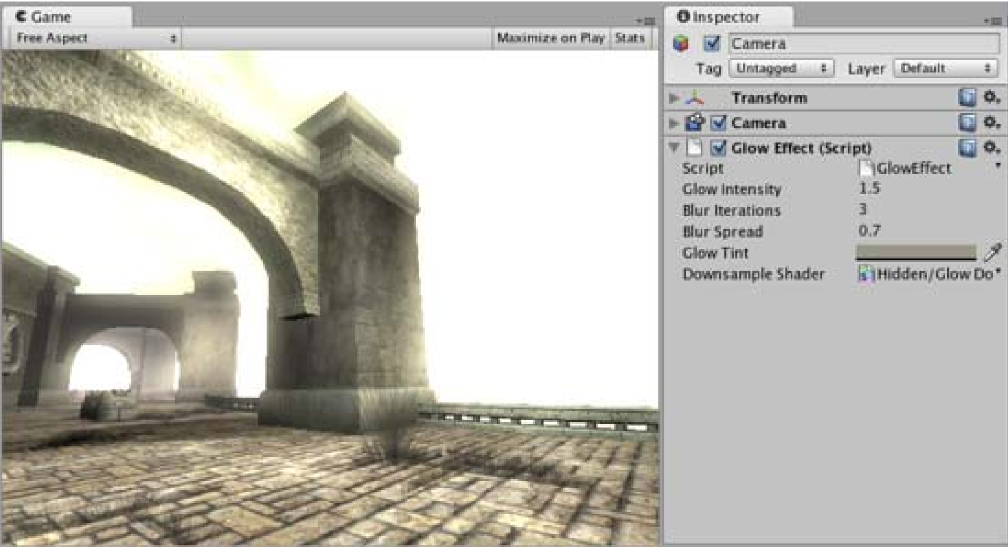
Page last updated: 2012-09-25

---

## script-GlowEffect

**Glow** (sometimes called "Bloom") can dramatically enhance the rendered image by making overbright parts "glow" (e.g. sun, light sources, strong highlights). The Bloom image effect gives greater control over the glow but has a bit higher processing overhead.

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.




*Glow effect applied to the scene*

## Properties

| | |
|---|---|
| **Glow Intensity** | Total brightness at the brightest spots of the glowing areas. |
| **Blur Iterations** | Number of times the glow is blurred when being drawn. Each iteration requires processing time. |
| **Blur Spread** | The pixel distance over which pixels are combined to produce blurring. |
| **Glow Tint** | Color tint applied to the glow. |
| **Downsample Shader** | The shader used for the glow. You generally should not have to change this. |

## Details

Glow uses the alpha channel of the final image to represent "color brightness". All colors are treated as RGB, multiplied by the alpha channel. You can view the contents of the alpha channel in **Scene View**.

All built-in shaders write the following information to alpha:

- Main texture's alpha multiplied by main color's alpha (not affected by lighting).
- Specular shaders add specular highlight multiplied by specular color's alpha.
- Transparent shaders do not modify alpha channel at all.
- Particle shaders do not modify alpha channel, except for Particles/Multiply which darkens anything that is in alpha.
- Skybox shaders write alpha of the texture multiplied by tint alpha

Most of the time you'll want to do this to get reasonable glow:

- Set material's main color alpha to zero or use a texture with zero alpha channel. In the latter case, you can put non-zero alpha in the texture to cause these parts to glow.
- Set the specular color alpha for Specular shaders to be 100%.
- Keep in mind what alpha the camera clears to (if it clears to a solid color), or what alpha the skybox material uses.
- Add the Glow image effect to the camera. Tweak *Glow Intensity* and *Blur Iterations* values, you can also take a look at the comments in the shader script source.
- The alpha channel on the Skybox can be used to great effect to add more glow when looking at the sun

**Tips:**

- Use the alpha rendering mode in the scene view toolbar to quickly see which objects output different values to the alpha channel.

## Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

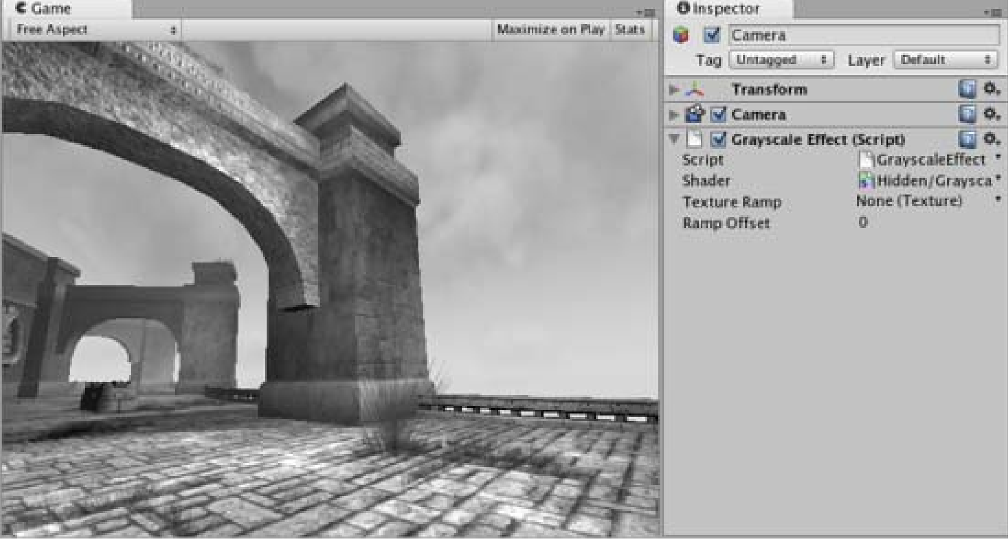
Page last updated: 2012-11-16

---

## script-GrayscaleEffect

**Grayscale** is a simple image effect that changes colors to grayscale by default. It can also use a **Texture Ramp** texture to remap luminance to arbitrary colors.

Like all image effects, Grayscale is available in Unity Pro only. Make sure to have the Pro Standard Assets installed.



*Grayscale image effect applied to the scene*

### Remapping colors

Grayscale can do a simple version of color correction, i.e. remap grayscale image into arbitrary colors. This can be used for effects like heat vision.

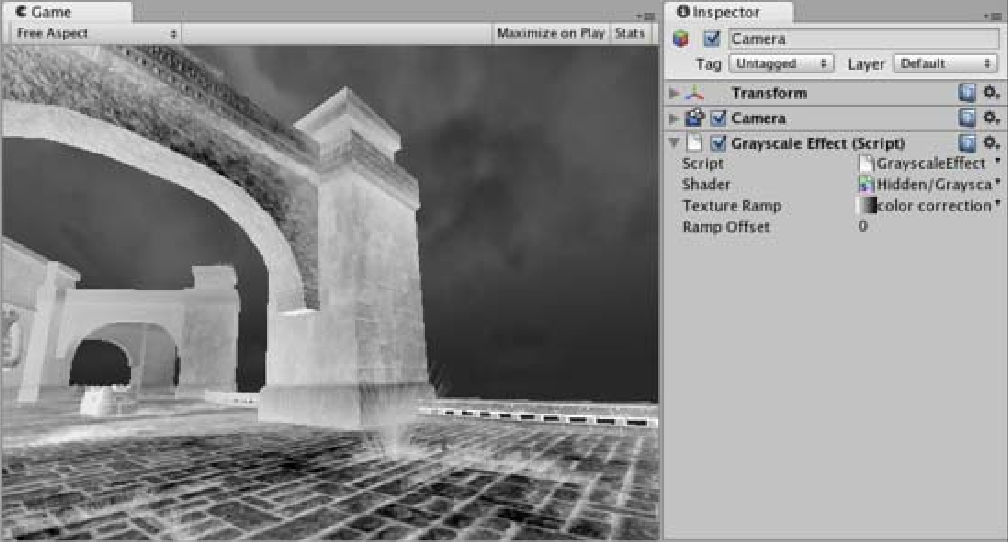
The process of color remapping is very similar to ColorCorrection effect:

1. Take a screenshot of a typical scene in your game.
2. Open it in Photoshop and convert to grayscale.
3. Color correct it using the **Image->Adjustments->Curves**.
4. Save the **.acv** file file from the dialog using **Save...**
5. Open **Pro Standard Assets->Image Based->color correction ramp.png** in Photoshop
6. Now apply color correction to the ramp image: open **Image->Adjustments->Curves** again and load your saved **.acv** file
7. Select your camera in Unity and select **Component->Image Effects->Grayscale** to add the effect. Select your modified color ramp.
8. Hit Play to see the effect in action!

#### Details

Color remapping works by remapping the original image luminance through the color ramp image (sized 256x1):

- result color = pixel's color in the ramp image at (OriginalLuminance + **RampOffset**) index. For example, to invert the colors in the image you only need to flip the original color ramp horizontally (so that it goes from white to black instead of from black to white):



*Grayscale applied to the scene with color ramp that goes from white to black.*

A more complex version of color remapping that does arbitrary color correction can be achieved with ColorCorrection image effect.

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

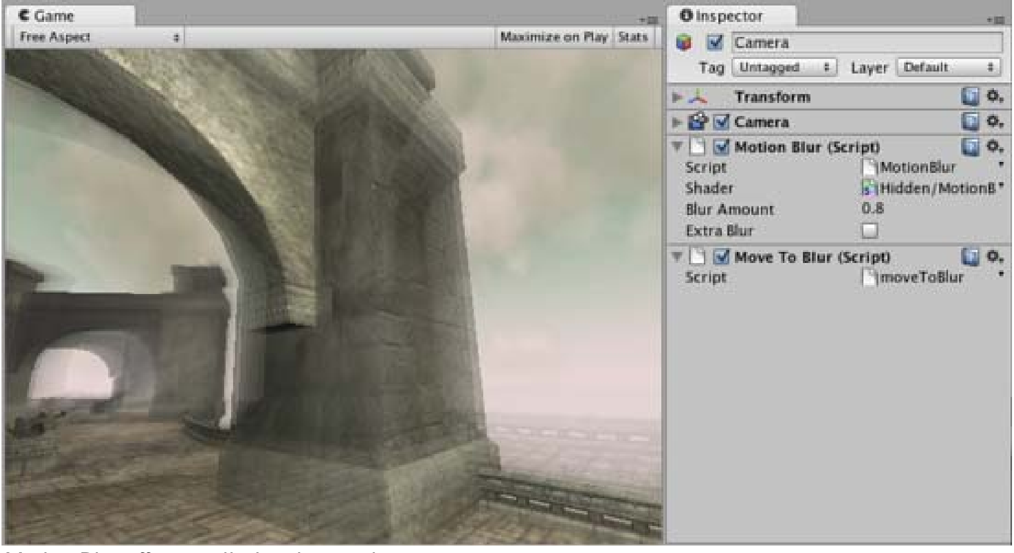
Page last updated: 2011-05-12

---

## script-MotionBlur

**Motion Blur** image effect enhances fast-moving scenes by leaving "motion trails" of previously rendered frames. For a modern implementation of Motion Blur, please refer to the new Camera Motion Blur Effect.

Like all image effects, Motion Blur is only available in Unity Pro. Make sure to have the Pro Standard Assets installed.



*Motion Blur effect applied to the rotating scene*

| | |
|---|---|
| **Blur Amount** | How much of the previous frames to leave in the image. Higher values make longer motion trails. |

**Extra Blur** If checked, this makes motion trails more blurry, by applying some extra blur to previous frames.

**Tips:**

- Motion Blur only works while in play mode because it's time based.

**Hardware support**

Motion Blur effect works all graphics cards that support rendering to a texture. E.g. GeForce2, Radeon 7000 and up. All image effects automatically disable themselves when they can not run on an end-users graphics card.

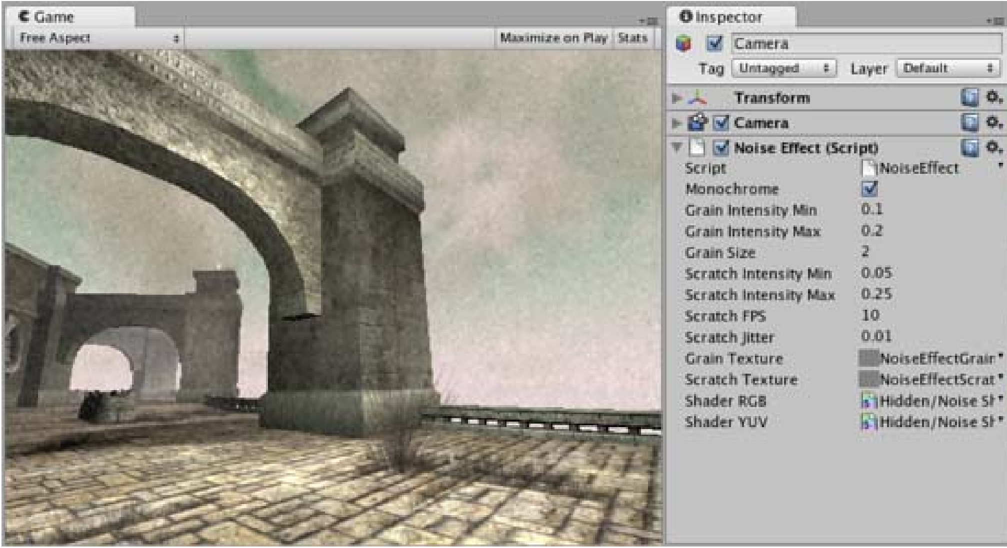
Page last updated: 2012-11-16

---

# script-NoiseEffect

**Noise** is an image postprocessing effect that can simulate both TV and VCR noise.

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.




*Noise effect with high intensity applied to the scene*

**Monochrome** If enabled, Noise is similar to TV noise. If disabled, it more closely resembles VCR noise - it distorts color values in YUV space, so you also get hue changes, mostly towards magenta/green gues.

**Grain Intensity Min/Max** The intensity of noise takes random values between **Min** and **Max**.

**Grain Size** The size of a single grain texture pixel in screen pixels. Increasing this will make noise grains larger.

**Scratch Intensity Min/Max** The intensity of additional scratch/dust takes random values between **Min** and **Max**.

**Scratch FPS** Scratches jump to different positions on the screen at this framerate.

**Scratch Jitter** Scratches can jitter slightly while remaining close to their original positions.

## Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

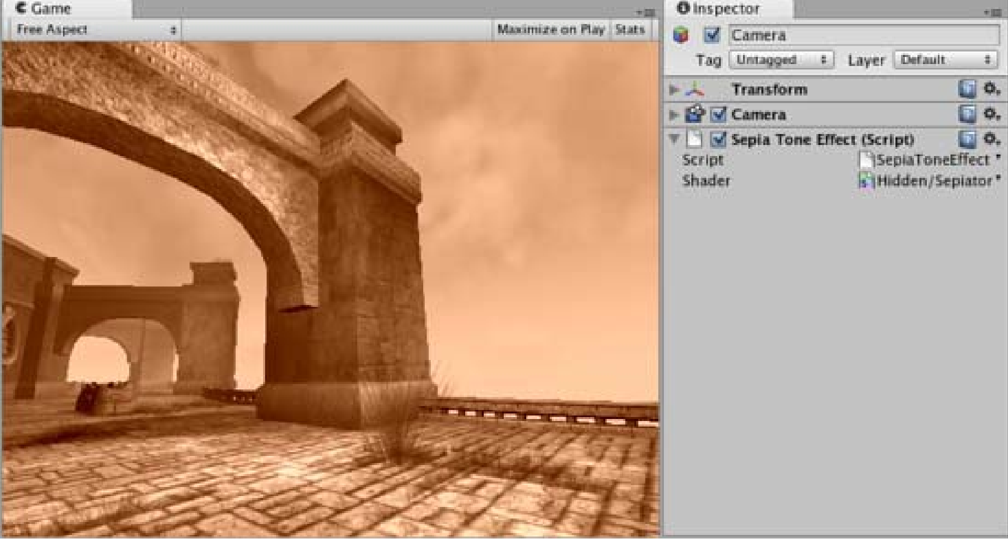
Page last updated: 2011-09-20

---

## script-SepiaToneEffect

**Sepia Tone** is a simple image effect that tints an image to resemble an old photograph.

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.



*Sepia Tone image effect applied to the scene*

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

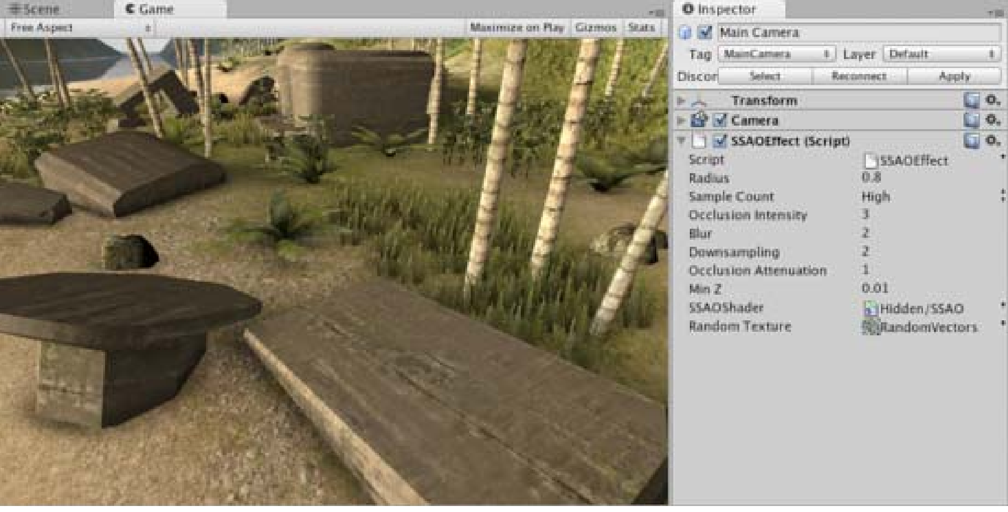
Page last updated: 2011-09-20

---

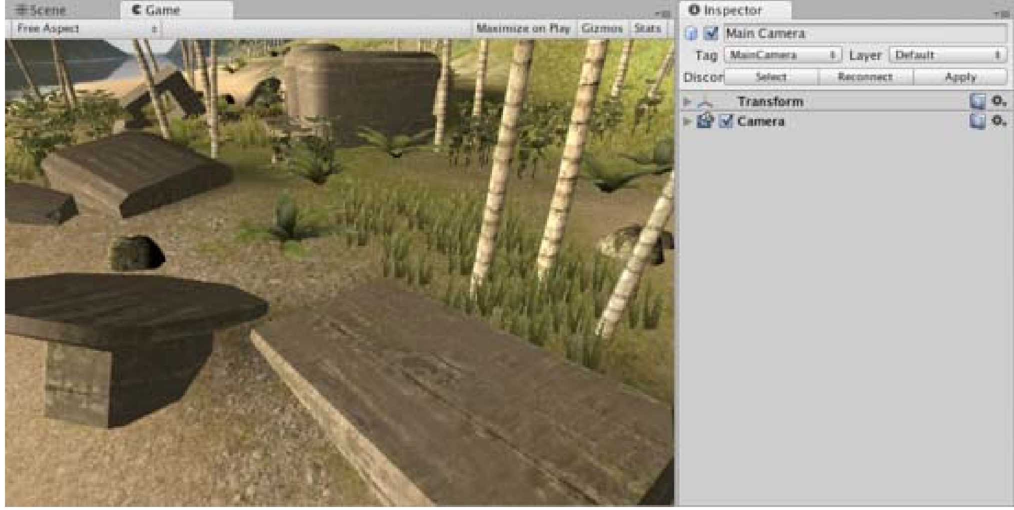
## script-SSAOEffect

**Screen Space Ambient Occlusion (SSAO)** approximates Ambient Occlusion in realtime, as an image post-processing effect. It darkens creases, holes and surfaces that are close to each other. In real life, such areas tend to block out or **occlude** ambient light, hence they appear darker.

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.




*SSAO applied to the scene.*




*The same scene without SSAO for comparison. Note the differences at the corners where structures or grass meet the ground.*

## Properties

| | |
|---|---|
| **Radius** | The maximum "radius" of a gap that will introduce ambient occlusion. |
| **Sample Count** | Number of ambient occlusion samples. A higher count will give better quality but with a higher processing overhead. |
| **Occlusion Intensity** | The degree of darkness added by ambient occlusion. |
| **Blur** | Amount of blur to apply to the darkening. No blur (0) is much faster but the darkened areas will be noisy. |
| **Downsampling** | The resolution at which calculations should be performed (for example, a downsampling value of 2 will work at half the screen resolution). Downsampling increases rendering speed at the cost of quality. |
| **Occlusion Attenuation** | How fast occlusion should attenuate with distance. |
| **Min Z** | Try increasing this value if there are artifacts. |

## Details

SSAO approximates ambient occlusion using an image processing effect. Its cost depends purely on screen resolution and SSAO parameters and does not depend on scene complexity as true AO would. However, the approximation tends to introduce artifacts. For example, objects that are outside of the screen do not contribute to occlusion and the amount of occlusion is dependent on viewing angle and camera position.

Note that SSAO is quite expensive in terms of processing time and generally should only be used on high-end graphics cards. Using SSAO will cause Unity to render the depth+normals texture of the camera which increases the number of draw calls and has a CPU processing overhead. However, the depth+normals texture then can be used for other effects as well (eg, Depth of Field). Once the texture is generated, the remainder of the SSAO effect is performed on the graphics card.

### Hardware support

SSAO works on graphics cards with Shader Model 3.0 support (eg, GeForce 6 and later, Radeon X1300 and later). All image effects automatically disable themselves when they can not run on a particular graphics card. Due to the complexity of the effect, SSAO is not supported on mobile devices.

Page last updated: 2011-09-21

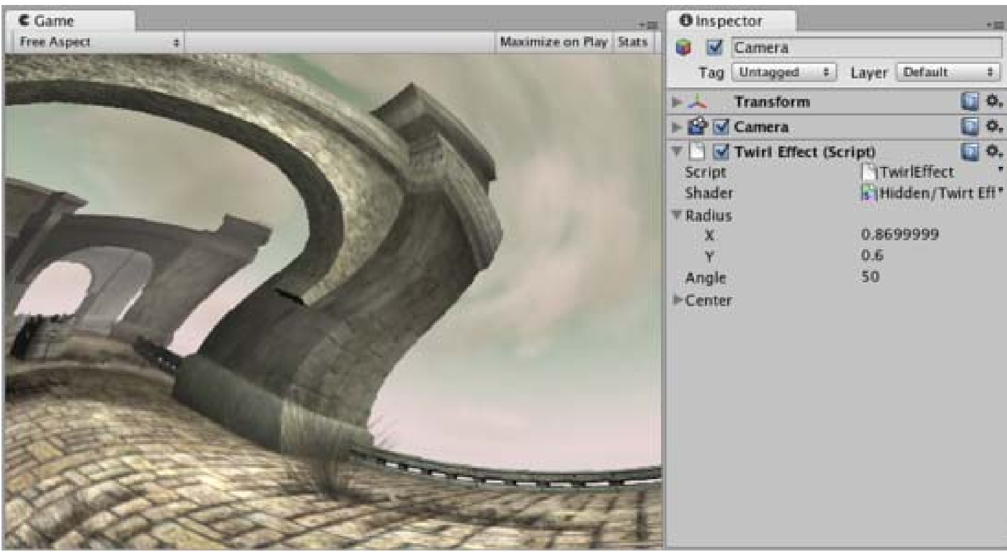
---

## script-TwirlEffect

The **Twirl** image effect distorts the rendered image within a circular region. The pixels at the centre of the circle are rotated by a specified angle; the rotation for other pixels in the circle decreases with distance from the centre, diminishing to zero at the circle's edge.

Twirl is similar to another image effect called Vortex, although vortex distorts the image around a central circle rather than a single point.

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.




*Twirl image effect applied to the scene*

| | |
|---|---|
| **Radius** | The radius of the ellipse where image distortion occurs, given in normalized screen coordinates (ie, a radius of 0.5 is half the size of the screen). |
| **Angle** | The angle of rotation at the centre point. |
| **Center** | The point at the centre of the circle where distortion occurs. |

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2011-09-20

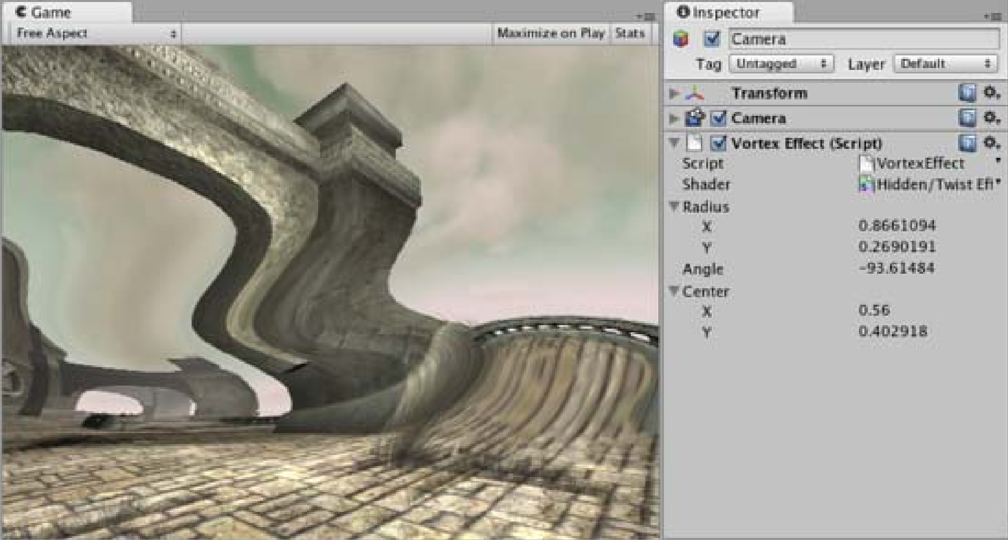
---

## script-VortexEffect

The **Vortex** image effect distorts the rendered image within a circular region. Pixels in the image are displaced around a central circular area by a specified angle; the amount of displacement reduces with distance from the centre, diminishing to

zero at the circle's edge. Vortex is similar to another image effect called Twirl, although Twirl distorts the image around a point rather than a circle.

As with the other image effects, this effect is only available in Unity Pro and you must have the Pro Standard Assets installed before it becomes available.



*Vortex image effect applied to the scene*

| | |
|---|---|
| **Radius** | The radius of the circle where distortion occurs, given in normalized screen coordinates (ie, a radius of 0.5 is half the size of the screen). |
| **Angle** | The angle by which pixels are displaced around the central circle. |
| **Center** | The center of the circular region of distortion. |

### Hardware support

This effect requires a graphics card with pixel shaders (2.0) or OpenGL ES 2.0. PC: NVIDIA cards since 2003 (GeForce FX), AMD cards since 2004 (Radeon 9500), Intel cards since 2005 (GMA 900); Mobile: OpenGL ES 2.0; Consoles: Xbox 360, PS3.

All image effects automatically disable themselves when they can not run on end-users graphics card.

Page last updated: 2011-09-20

---

## comp-ManagerGroup

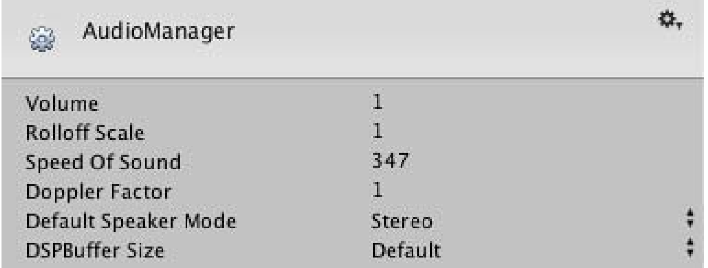
- Audio Manager
- Editor settings
- Input Manager
- NavMesh Layers (Pro only)
- Network Manager
- Physics Manager
- Player Settings
- Quality Settings
- Render Settings
- Script Execution Order Settings
- Tag Manager
- Time Manager

Page last updated: 2007-07-16

---

# class-AudioManager

The **Audio Manager** allows you to tweak the maximum volume of all sounds playing in the scene. To see it choose **Edit->Project Settings->Audio**.




## Properties

| | |
|---|---|
| **Volume** | The volume of all sounds playing. |
| **Rolloff Scale** | Sets the global attenuation rolloff factor for Logarithmic rolloff based sources (see Audio Source). The higher the value the faster the volume will attenuate, conversely the lower the value, the slower it attenuate (value of 1 will simulate the "real world"). |
| **Speed of Sound** | The speed of sound. 343 is the real world speed of sound, if you are using a meters as your units. You can adjust this value to make your objects have more or less Doppler effect with larger or smaller speed. |
| **Doppler Factor** | How audible the Doppler effect is. When it is zero it is turned off. 1 means it should be quite audible for fast moving objects. |
| **Default Speaker Mode** | Defines which speaker mode should be the default for your project. Default is 2 for stereo speaker setups (see AudioSpeakerMode in the scripting API reference for a list of modes). |
| **DSP Buffer Size** | The size of the DSP buffer can be set to optimise for latency or performance |
| **Default** | Default buffer size |
| **Best Latency** | Trades off performance in favour of latency |
| **Good Latency** | Balance between latency and performance |
| **Best Performance** | Trades off latency in favour of performance |

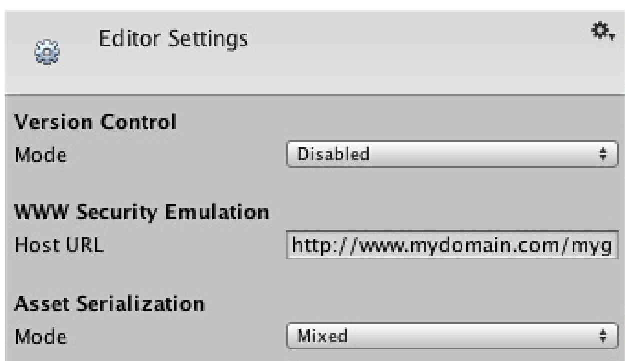
## Details

If you want to use Doppler Effect set **Doppler Factor** to 1. Then tweak both **Speed of Sound** and **Doppler Factor** until you are satisfied.
Speaker mode can be changed runtime from your application through scripting. See Audio Settings.

Page last updated: 2011-10-24

---

# class-EditorManager




## Properties

| | |
|---|---|
| **Version Control** | Which version control system should be used. |
| **WWW Security Emulation** | For webplayer testing, the editor can "pretend" that the game is a webplayer hosted at this URL. |

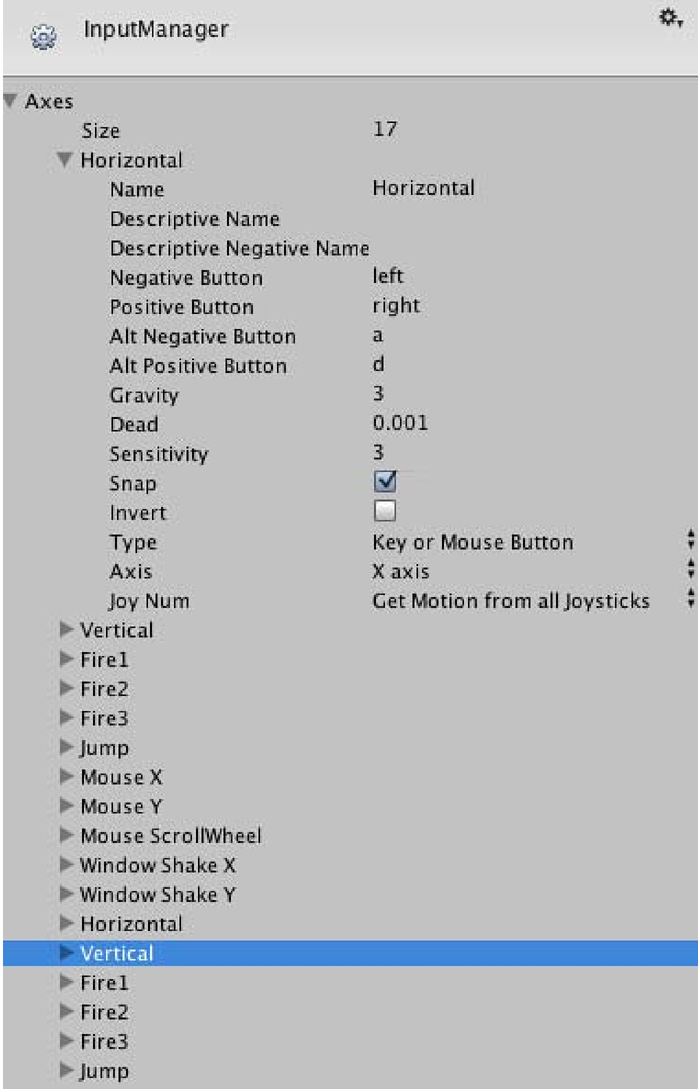
| | |
|---|---|
| **Asset Serialization** | To assist with version control merges, Unity can store scene files in a textual format (see the textual scene format pages for further details). If merges will not be performed then Unity can store scenes in a more space efficient binary format or allow both text and binary scene files to exist at the same time. |

Page last updated: 2012-01-31

# class-InputManager

## Desktop

The **Input Manager** is where you define all the different input axes and game actions for your project.




*The Input Manager*

To see the Input Manager choose: **Edit->Project Settings->Input**.

### Properties

| | |
|---|---|
| **Axes** | Contains all the defined input axes for the current project: **Size** is the number of different input axes in this project, **Element 0, 1, ...** are the particular axes to modify. |
| **Name** | The string that refers to the axis in the game launcher and through scripting. |
| **Descriptive Name** | A detailed definition of the **Positive Button** function that is displayed in the game launcher. |

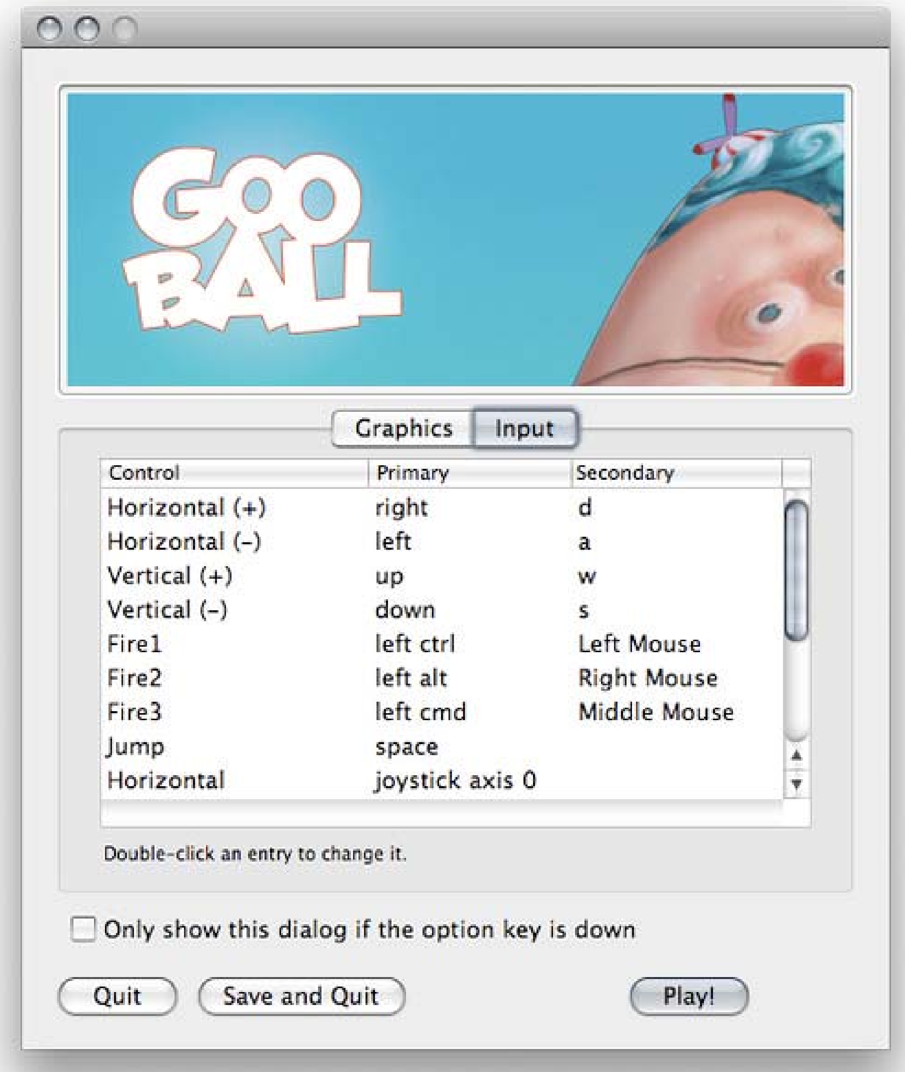
| | |
|---|---|
| **Descriptive Negative Name** | A detailed definition of the **Negative Button** function that is displayed in the game launcher. |
| **Negative Button** | The button that will send a negative value to the axis. |
| **Positive Button** | The button that will send a positive value to the axis. |
| **Alt Negative Button** | The secondary button that will send a negative value to the axis. |
| **Alt Positive Button** | The secondary button that will send a positive value to the axis. |
| **Gravity** | How fast will the input recenter. Only used when the **Type** is **key / mouse button**. |
| **Dead** | Any positive or negative values that are less than this number will register as zero. Useful for joysticks. |
| **Sensitivity** | For keyboard input, a larger value will result in faster response time. A lower value will be more smooth. For Mouse delta the value will scale the actual mouse delta. |
| **Snap** | If enabled, the axis value will be immediately reset to zero after it receives opposite inputs. Only used when the **Type** is **key / mouse button**. |
| **Invert** | If enabled, the positive buttons will send negative values to the axis, and vice versa. |
| **Type** | Use **Key / Mouse Button** for any kind of buttons, Mouse Movement for mouse delta and scrollwheels, **Joystick Axis** for analog joystick axes and **Window Movement** for when the user shakes the window. |
| **Axis** | Axis of input from the device (joystick, mouse, gamepad, etc.) |
| **Joy Num** | Which joystick should be used. By default this is set to retrieve the input from all joysticks. This is only used for input axes and not buttons. |

### Details

All the axes that you set up in the Input Manager serve two purposes:

- They allow you to reference your inputs by axis name in scripting
- They allow the players of your game to customize the controls to their liking

All defined axes will be presented to the player in the game launcher, where they will see its name, detailed description, and default buttons. From here, they will have the option to change any of the buttons defined in the axes. Therefore, it is best to write your scripts making use of axes instead of individual buttons, as the player may want to customize the buttons for your game.




*The game launcher's Input window is displayed when your game is run*

See also: Input.

### Hints

- Axes are not the best place to define "hidden" or secret functions, as they will be displayed very clearly to the player in the game launcher.

## iOS

This section is not supported on iOS devices.

For more info on how to work with input on iOS devices, please refer to the iOS Input page.

## Android

This section is not supported on Android devices.

For more info on how to work with input on Android devices, please refer to the Android Input page.

Page last updated: 2011-10-21

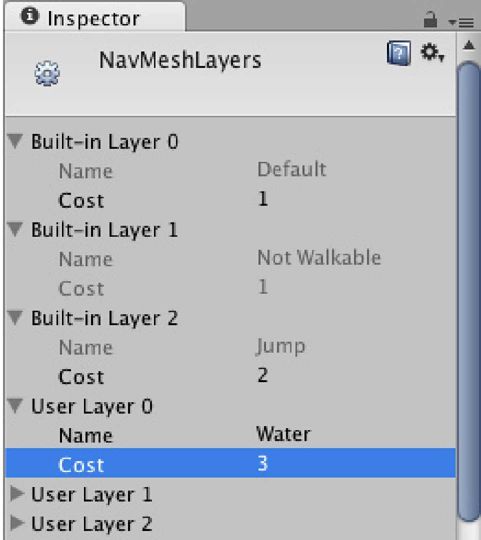
## class-NavMeshLayers

The primary task of the navigation system is finding the *optimal* path between two points in navigation-space. In the simplest case, the optimal path is the shortest path. However, in many complex environments, some areas are harder to move thru than others (for example, crossing a river can be more costly than running across a bridge). To model this, Unity utilizes the concept of *cost* and the *optimal path* is defined as the path with the lowest cost. To manage costs, Unity has the concept of **Navmesh Layers**. Each geometry marked up as **Navmesh Static** will belong to a Navmesh Layer.

During pathfinding, instead of comparing lengths of potential path segments, the cost of each segment is evaluated. This is a done by scaling the length of each segment by the cost of the navmesh layer for that particular segment. Note that when all costs are set to 1, the optimal path is equivalent to the shortest path.

To define custom layers per project

- Go to **Edit**->**Project Settings**->**Navmesh Layers**




- Go to one of the user layers, and set up name and cost
  - The name is what will be used everywhere in the scene for identifying the navmesh layer
  - The cost indicates how difficult it is to traverse the NavMesh layer. 1 is default, 2.0 is twice as difficult, 0.5 is half as difficult, etc.
- There are 3 built-in layers
  - Default - specifies the cost for everything not otherwise specified
  - Not walkable - the cost is ignored
  - Jump - the cost of automatically generated off-mesh links

To apply custom layers to specific geometry

- Select the geometry in the editor
- Pull up the **Navigation Mesh** window (**Window**->**Navigation**)
- Go to the **Object** tab, and select the desired **Navigation layer** for that object

- If you have **Show NavMesh** enabled in the **Navmesh Display** window, the different layers should show up in different colors in the editor.

To tell an agent what layers he can or cannot traverse

- Go to the **NavMeshAgent** component of the agent's geometry
- Modify **NavMesh Walkable** property
- Don't forget to set the agent's **destination** property from a script

*Note: Setting the cost value below 1 is not recommended, as the underlying pathfinding method does not guarantee an optimal path in this case*

One good use case for using Navmesh Layers is:

- You have a road that pedestrians (NavmeshAgents) need to cross.
- The pedestrian walkway in the middle is the preferred place for them to go
- Set up a navmesh layer with high cost for most of the road, and a navmesh layer with a low cost for the pedestrian walkway.
- This will cause agents to prefer paths that go thru the pedestrian walkway.

Another relevant topic for advanced pathfinding is Off-mesh links

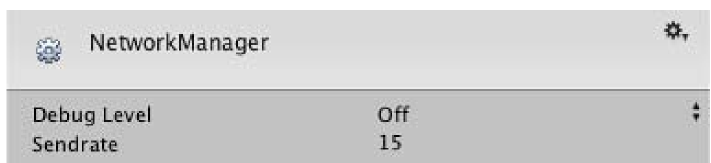
(back to Navigation and Pathfinding)

Page last updated: 2012-04-24

---

# class-NetworkManager

The **Network Manager** contains two very important properties for making Networked multiplayer games.



*The Network Manager*

You can access the Network Manager by selecting **Edit->Project Settings->Network** from the menu bar.

## Properties

| | |
|---|---|
| **Debug Level** | The level of messages that are printed to the console |
| **Off** | Only errors will be printed |
| **Informational** | Significant networking events will be printed |
| **Full** | All networking events will be printed |
| **Sendrate** | Number of times per second that data is sent over the network |

## Details

Adjusting the Debug Level can be enormously helpful in fine-tuning or debugging your game's networking behaviors. At first, setting it to **Full** will allow you to see every single network action that is performed. This will give you an overall sense of how frequently you are using network communication and how much bandwidth you are using as a result.

When you set it to **Informational**, you will see major events, but not every individual activity. Assigning unique **Network IDs** and buffering **RPC** calls will be logged here.

When it is **Off**, only errors from networking will be printed to the console.

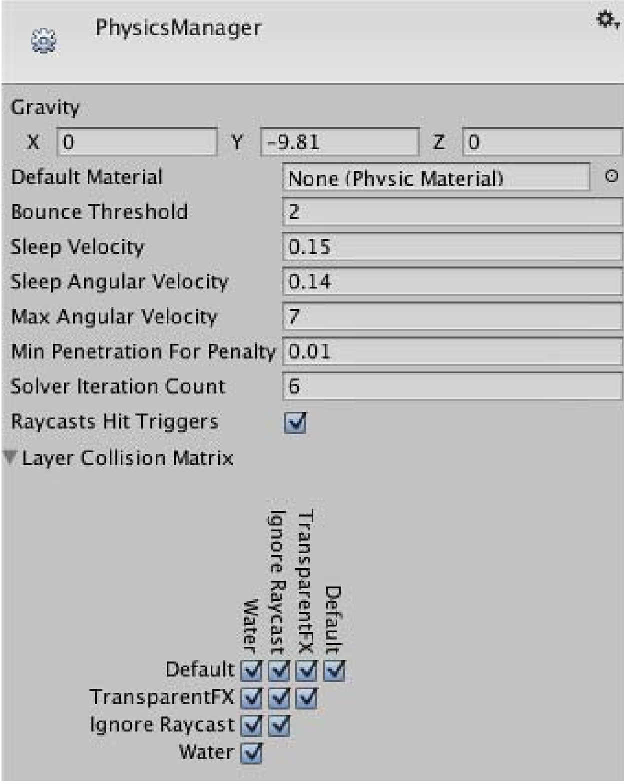
The data that is sent at the **Sendrate** intervals (1 second / **Sendrate** = interval) will vary based on the **Network View** properties of each broadcasting object. If the Network View is using **Unreliable**, its data will be send at each interval. If the Network View is using **Reliable Delta Compressed**, Unity will check to see if the Object being watched has changed since the last interval. If it has changed, the data will be sent.

Page last updated: 2011-10-21

---

## class-PhysicsManager

You can access the **Physics Manager** by selecting **Edit->Project Settings->Physics** from the menu bar.




*The Physics Manager*

### Properties

| | |
|---|---|
| **Gravity** | The amount of gravity applied to all **Rigidbodies**. Usually gravity acts only on the Y-axis (negative is down). Gravity is meters/(seconds^2). |
| **Default Material** | The default **Physics Material** that will be used if none has been assigned to an individual **Collider**. |
| **Bounce Threshold** | Two colliding objects with a relative velocity below this value will not bounce. This value also reduces jitter so it is not recommended to set it to a very low value. |
| **Sleep Velocity** | The default linear velocity, below which objects start going to sleep. |
| **Sleep Angular Velocity** | The default angular velocity, below which objects start going to sleep. |
| **Max Angular Velocity** | The default maximimum angular velocity permitted for any Rigidbodies. The angular velocity of Rigidbodies is clamped to stay within **Max Angular Velocity** to avoid numerical instability with quickly rotating bodies. Because this may prevent intentional fast rotations on objects such as wheels, you can override this value for any Rigidbody by scripting **Rigidbody.maxAngularVelocity**. |
| **Min Penetration For Penalty** | How deep in meters are two objects allowed to penetrate before the collision solver pushes them apart. A higher value will make objects penetrate more but reduces jitter. |
| **Solver Iteration Count** | Determines how accurately joints and contacts are resolved. Usually a value of 7 works very well for almost all situations. |
| **Raycasts Hit Triggers** | If enabled, any Raycast that intersects with a Collider marked as a Trigger will return a hit. If disabled, these intersections will not return a hit. |
| **Layer Collision Matrix** | Defines how the layer-based collision detection system will behave. |

### Details

The Physics Manager is where you define the default behaviors of your world. For an explanation of Rigidbody Sleeping, read this page about sleeping.

### Hints

- If you are having trouble with connected bodies oscillating and behaving eratically, setting a higher **Solver Iteration Count** may improve their stability, but will require more processing power.

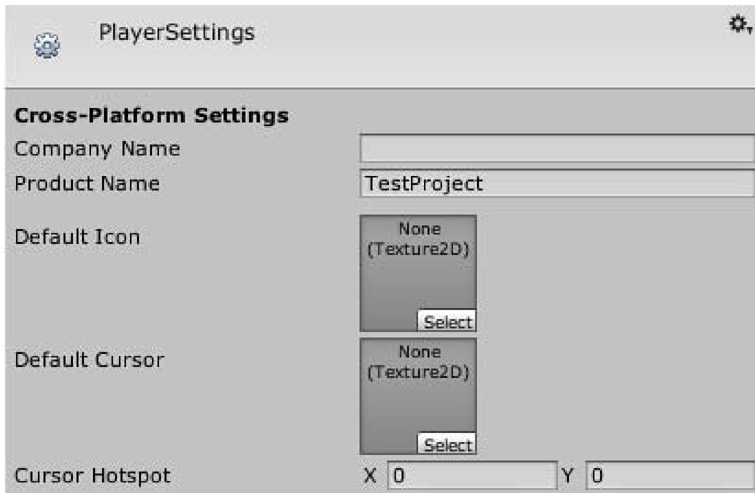
Page last updated: 2012-09-03

---

## class-PlayerSettings40

**Player Settings** is where you define various parameters (platform specific) for the final game that you will build in Unity. Some of these values for example are used in the **Resolution Dialog** that launches when you open a standalone game, others are used by XCode when building your game for the iOS devices, so it's important to fill them out correctly.

To see the Player Settings choose **Edit->Project Settings->Player** from the menu bar.




*Global Settings that apply to any project you create.*

### Cross-Platform Properties

| | |
|---|---|
| **Company Name** | The name of your company. This is used to locate the preferences file. |
| **Product Name** | The name that will appear on the menu bar when your game is running and is used to locate the preferences file also. |
| **Default Icon** | Default icon the application will have on every platform (You can override this later for platform specific needs). |
| **Default Cursor** | Default cursor the application will have on every supported platform. |
| **Cursor Hotspot** | Cursor hotspot in pixels from the top left of the default cursor |

## Per-Platform Settings

## Desktop

### Web-Player

***Resolution And Presentation***

Resolution and Presentation
Resolution
Default Screen Width* 960
Default Screen Height* 600
Run In Background*
WebPlayer Template
Black Background | Default | No Context Menu
* Shared setting between multiple platforms.

***Resolution***

| | |
|---|---|
| **Default Screen Width** | Screen Width the player will be generated with. |
| **Default Screen Height** | Screen Height the plater will be generated with. |
| **Run in background** | Check this if you dont want to stop executing your game if the player looses focus. |
| ***WebPlayer Template*** | For more information you should check the "Using WebPlayer templates page", note that for each built-in and custom template there will be an icon in this section. |

***Icon***

Icons don't have any meaning for webplayer builds (you can set icons for Native Client builds in the Native Client section of the Player Settings).

***Other Settings***

Other Settings
Rendering
Rendering Path* Forward
Color Space* Gamma
Use Direct3D 11*
Static Batching
Dynamic Batching
Streaming
First Streamed Level 0
Configuration
Scripting Define Symbols
Optimization
Api Compatibility Level Webplayer Subset
Optimize Mesh Data*
* Shared setting between multiple platforms.

***Rendering***

| | |
|---|---|
| **Rendering Path** | This property is shared between Standalone and WebPlayer content. |
| **Vertex Lit** | Lowest lighting fidelity, no shadows support. Best used on old machines or limited mobile platforms. |
| **Forward with Shaders** | Good support for lighting features; limited support for shadows. |
| **Deferred Lighting** | Best support for lighting and shadowing features, but requires certain level of hardware support. Best used if you have many realtime lights. Unity Pro only. |
| **Color Space** | The color space to be used for rendering |

| | |
|---|---|
| **GammaSpace Rendering** | Rendering is gamma-corrected |
| **Linear Rendering** | Rendering is done in linear space |
| **Hardware Sampling** | |
| **Use Direct3D 11** | Use Direct3D 11 for rendering. |
| **Static Batching** | Set this to use Static batching on your build (Inactive by default in webplayers). Unity Pro only. |
| **Dynamic Batching** | Set this to use Dynamic Batching on your build (Activated by default). |
| ***Streaming*** | |
| **First Streamed Level** | If you are publishing a Streamed Web Player, this is the index of the first level that will have access to all Resources.Load assets. |
| ***Configuration*** | |
| **Scripting Define Symbols** | Custom compilation flags (see the platform dependent compilation page for details). |
| ***Optimization*** | |
| **Optimize Mesh Data** | Remove any data from meshes that is not required by the material applied to them (tangents, normals, colors, UV). |

## Standalone

***Resolution And Presentation***

Resolution and Presentation
Resolution
Default Screen Width 1024
Default Screen Height 768
Run In Background*
Standalone Player Options
Default Is Full Screen
Capture Single Screen
Display Resolution Dialog Enabled
Use Player Log
Mac App Store Validation
Mac Fullscreen Mode FullscreenWindowWith
Supported Aspect Ratios
4:3
5:4
16:10
16:9
Others
* Shared setting between multiple platforms.

| | |
|---|---|
| ***Resolution*** | |
| **Default Screen Width** | Screen Width the stand alone game will be using by default. |
| **Default Screen Height** | Screen Height the plater will be using by default. |
| **Run in background** | Check this if you dont want to stop executing your game if it looses focus. |
| ***Standalone Player Options*** | |
| **Default is Full Screen** | Check this if you want to start your game by default in full screen mode. |
| **Capture Single Screen** | If enabled, standalone games in fullscreen mode will not darken the secondary monitor in multi-monitor setups. |
| **DisplayResolution Dialog** | |
| **Disabled** | No resolution dialog will appear when starting the game. |
| **Enabled** | Resolution dialog will always appear when the game is launched. |
| **Hidden by default** | The resolution player is possible to be opened only if you have pressed the "alt" key when starting the game. |
| **Use Player Log** | Write a log file with debugging information. If you plan to submit your application to the Mac App Store you will want to leave this option un-ticked. Ticked is the default. |
| **Resizable Window** | Allow user to resize the standalone player window. |
| **Mac App Store Validation** | Enable receipt validation for the Mac App Store. |
| **Mac Fullscreen Mode** | Options for fullscreen mode on Mac builds |

| | |
|---|---|
| **Capture Display** | Unity will take over the whole display (ie, GUI from other apps will not appear and the user cannot switch apps until fullscreen mode is exited). |
| **Fullscreen Window** | Unity runs in a window that covers the whole screen at desktop resolution. Other apps' GUI will display correctly and it is possible to switch apps with cmd + tab or trackpad gestures on OSX 10.7 and above. |
| **Fullscreen Window with Menu Bar and Dock** | As fullscreen window mode but the standard menu bar and dock will also be shown. |
| **Supported Aspect Ratios** | Aspect Ratios selectable in the Resolution Dialog will be monitor-supported resolutions of enabled items from this list. |

***Icon***

Icon
Override for Standalone
1024x1024
512x512
256x256
128x128
48x48
32x32
16x16

| | |
|---|---|
| **Override for Standalone** | Check if you want to assign a custom icon you would like to be used for your standalone game. Different sizes of the icon should fill in the squares below. |

***Splash Image***

Splash Image
Config Dialog Banner
None (Texture2D)
Select

**Config Dialog Banner** Add your custom splash image that will be displayed when the game is starting.

***Other Settings***

Other Settings
Rendering
Rendering Path* Forward
Color Space* Gamma
Use Direct3D 11*
Static Batching
Dynamic Batching
Configuration
Scripting Define Symbols
Optimization
Api Compatibility Level .NET 2.0 Subset
Optimize Mesh Data*
* Shared setting between multiple platforms.

| | |
|---|---|
| ***Rendering*** | |
| **Rendering Path** | This property is shared between Standalone and WebPlayer content. |
| **Vertex Lit** | Lowest lighting fidelity, no shadows support. Best used on old machines or limited mobile platforms. |
| **Forward with Shaders** | Good support for lighting features; limited support for shadows. |
| **Deferred Lighting** | Best support for lighting and shadowing features, but requires certain level of hardware support. Best used if you have many realtime lights. Unity Pro only. |
| **Color Space** | The color space to be used for rendering |
| **GammaSpace Rendering** | Rendering is gamma-corrected |
| **Linear Rendering Hardware Sampling** | Rendering is done in linear space |
| **Static Batching** | Set this to use Static batching on your build (Inactive by default in webplayers). Unity Pro only. |
| **Dynamic Batching** | Set this to use Dynamic Batching on your build (Activated by default). |
| ***Configuration*** | |
| **Scripting Define Symbols** | Custom compilation flags (see the platform dependent compilation page for details). |
| ***Optimization*** | |
| **API Compatibility Level** | |
| **.Net 2.0** | .Net 2.0 libraries. Maximum .net compatibility, biggest file sizes |
| **.Net 2.0 Subset** | Subset of full .net compatibility, smaller file sizes |
| **Optimize Mesh Data** | Remove any data from meshes that is not required by the material applied to them (tangents, normals, colors, UV). |

## iOS

***Resolution And Presentation***

Resolution and Presentation
Resolution
Default Orientation* Portrait
Status Bar
Status Bar Hidden
Status Bar Style Default
Use 32-bit Display Buffer*
Show Loading Indicator Don't Show
* Shared setting between multiple platforms.

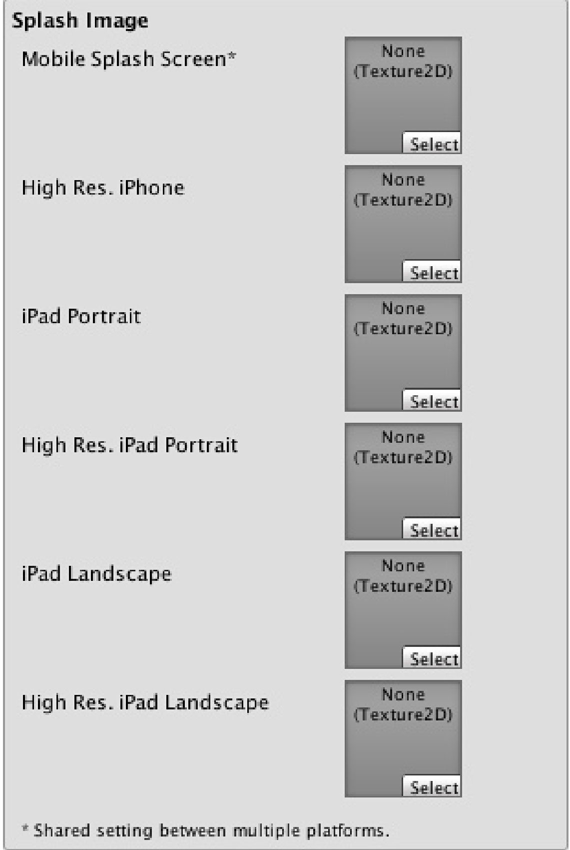
| | |
|---|---|
| ***Resolution*** | |
| **Default Orientation** | (This setting is shared between iOS and Android devices) |
| **Portrait** | The device is in portrait mode, with the device held upright and the home button at the bottom. |
| **Portrait Upside Down** | The device is in portrait mode but upside down, with the device held upright and the home button at the top. |
| **Landscape Right** | The device is in landscape mode, with the device held upright and the home button on the **left** side. |
| **Landscape Left** | The device is in landscape mode, with the device held upright and the home button on the **right** side. |
| **Auto Rotation** | The screen orientation is automatically set based on the physical device orientation. |
| ***Auto Rotation settings*** | |
| **Use Animated Autorotation** | When checked, orientation change is animated. This only applies when Default orientation is set to Auto Rotation. |
| ***Allowed Orientations for Auto Rotation*** | |
| **Portrait** | When checked, portrait orientation is allowed. This only applies when Default orientation is set to Auto Rotation. |
| **Portrait Upside Down** | When checked, portrait upside down orientation is allowed. This only applies when Default orientation is set to Auto Rotation. |
| **Landscape Right** | When checked, landscape right (home button on the **left** side) orientation is allowed. This only applies when Default orientation is set to Auto Rotation. |
| **Landscape Left** | When checked, landscape left (home button is on the **right** side) orientation is allowed. This only applies when Default orientation is set to Auto Rotation. |
| ***Status Bar*** | |
| **Status Bar Hidden** | Specifies whether the status bar is initially hidden when the application launches. |
| **Status Bar Style** | Specifies the style of the status bar as the application launches |
| **Default** | |
| **Black Translucent** | |
| **Black Opaque** | |
| **Use 32-bit Display Buffer** | Specifies if Display Buffer should be created to hold 32-bit color values (16-bit by default). Use it if you see banding, or need alpha in your ImageEffects, as they will create RTs in same format as Display Buffer. |
| **Show Loading Indicator** | Options for the loading indicator |
| **Don't Show** | No indicator |
| **White Large** | Indicator shown large and in white |
| **White** | Indicator shown at normal size in white |
| **Gray** | Indicator shown at normal size in gray |

***Icon***

Icon
Override for iPhone
144x144
114x114
72x72
57x57
Prerendered Icon

| | |
|---|---|
| **Override for iOS** | Check if you want to assign a custom icon you would like to be used for your iPhone/iPad game. Different sizes of the icon should fill in the squares below. |
| **Prerendered icon** | If unchecked iOS applies sheen and bevel effects to the application icon. |

***Splash Image***




| | |
|---|---|
| **Mobile Splash Screen (Pro-only feature)** | Specifies texture which should be used for iOS Splash Screen. Standard Splash Screen size is 320x480.(This is shared between Android and iOS) |

| | |
|---|---|
| **High Res. iPhone (Pro-only feature)** | Specifies texture which should be used for iOS 4th gen device Splash Screen. Splash Screen size is 640x960. |
| **iPad Portrait (Pro-only feature)** | Specifies texture which should be used as iPad Portrait orientation Splash Screen. Standard Splash Screen size is 768x1024. |
| **High Res. iPad Portrait** | Specifies texture which should be used as the high res iPad Portrait orientation Splash Screen. Standard Splash Screen size is 1536x2048. |
| **iPad Landscape (Pro-only feature)** | Specifies texture which should be used as iPad Landscape orientation Splash Screen. Standard Splash Screen size is 1024x768. |
| **High res. iPad Landscape (Pro-only feature)** | Specifies texture which should be used as the high res iPad Landscape orientation Splash Screen. Standard Splash Screen size is 2048x1536. |

***Other Settings***

Other Settings
Rendering
Static Batching
Dynamic Batching
Identification
Bundle Identifier* com.Company.Produ
Bundle Version* 1.0
Configuration
Target Device iPhone + iPad
Target Resolution Native (default device
Graphics Level* OpenGL ES 2.0
Accelerometer Frequency 60 Hz
Override IPod Music
Requires Persistent WiFi
Exit on Suspend
Scripting Define Symbols
Optimization
Api Compatibility Level .NET 2.0 Subset
AOT Compilation Options
SDK Version Device SDK
Target iOS Version 4.0
Stripping Level* Disabled
Script Call Optimization Slow and Safe
Optimize Mesh Data*
* Shared setting between multiple platforms.

| | |
|---|---|
| ***Rendering*** | |
| **Static Batching** | Set this to use Static batching on your build (Activated by default). Pro-only feature. |
| **Dynamic Batching** | Set this to use Dynamic Batching on your build (Activated by default). |
| ***Identification*** | |
| **Bundle Identifier** | The string used in your provisioning certificate from your Apple Developer Network account(This is shared between iOS and Android) |
| **Bundle Version** | Specifies the build version number of the bundle, which identifies an iteration (released or unreleased) of the bundle. This is a monotonically increased string, comprised of one or more period-separated |
| ***Configuration*** | |
| **Target Device** | Specifies application target device type. |
| **iPhone Only** | Application is targeted for iPhone devices only. |
| **iPad Only** | Application is targeted for iPad devices only. |
| **iPhone + iPad** | Application is targeted for both iPad and iPhone devices. |

| | |
|---|---|
| **Target Resolution** | Resolution you want to use on your deployed device.(This setting will not have any effect on devices with maximum resolution of 480x320) |
| **Native(Default Device Resolution)** | Will use the device native resolution. |
| **Auto (Best Performance)** | Chooses resolution automatically, favouring performance over graphic quality. |
| **Auto (Best Quality)** | Chooses resolution automatically, favouring graphic quality over performance. |
| **320p (iPhone)** | Pre-Retina iPhone display. |
| **640p (iPhone Retina Display)** | iPhone with Retina. |
| **768p (iPad)** | iPad display. |
| **Graphics Level** | OpenGL version. |
| **OpenGL ES 1.x** | OpenGL ES 1.x versions. |
| **OpenGL ES 2.0** | OpenGL ES 2.0. |
| **Accelerometer Frequency** | How often the accelerometer is sampled |
| **Disabled** | Accelerometer is not sampled |
| **15Hz** | 15 samples per second |
| **30Hz** | 30 samples per second |
| **60Hz** | 60 samples per second |
| **100Hz** | 100 samples per second |
| **Override iPod Music** | If selected, the application will silence user's iPod music. Otherwise user's iPod music will continue playing in the background. |
| **UI Requires Persistent WiFi** | Specifies whether the application requires a Wi-Fi connection. iOS maintains the active Wi-Fi connection open while the application is running. |
| **Exit on Suspend** | Specifies whether the application should quit when suspended to background on iOS versions that support multitasking. |
| **Scripting Define Symbols** | Custom compilation flags (see the platform dependent compilation page for details). |
| ***Optimization*** | |
| **Api Compatibility Level** | Specifies active .NET API profile |
| **.Net 2.0** | .Net 2.0 libraries. Maximum .net compatibility, biggest file sizes |
| **.Net 2.0 Subset** | Subset of full .net compatibility, smaller file sizes |
| **AOT compilation options** | Additional AOT compiler options. |
| **SDK Version** | Specifies iPhone OS SDK version to use for building in Xcode |
| **Device SDK** | SDK to run on actual hardware. |
| **Simulator SDK** | SDK to run only on the simulator. |
| **Target iOS Version** | Specifies lowest iOS version where final application will able to run; ranges from iOS 4.0 to 6.0. |
| **Stripping Level (Pro-only feature)** | Options to strip out scripting features to reduce built player size(This setting is shared between iOS and Android Platforms) |
| **Disabled** | No reduction is done. |
| **Strip Assemblies** | Level 1 size reduction. |
| **Strip ByteCode** | Level 2 size reduction (includes reductions from Level 1). |
| **Use micro mscorlib** | Level 3 size reduction (includes reductions from Levels 1 and 2). |
| **Script Call Optimization** | Optionally disable exception handling for a speed boost at runtime |
| **Slow and Safe** | Full exception handling will occur with some performance impact on the device |
| **Fast but no Exceptions** | No data provided for exceptions on the device, but the game will run faster |
| **Optimize Mesh Data** | Remove any data from meshes that is not required by the material applied to them (tangents, normals, colors, UV). |

**Note:** If you build for example for iPhone OS 3.2, and then select Simulator 3.2 in Xcode you will get a ton of errors. So you **MUST** be sure to select a proper Target SDK in Unity Editor.

## Android

***Resolution And Presentation***

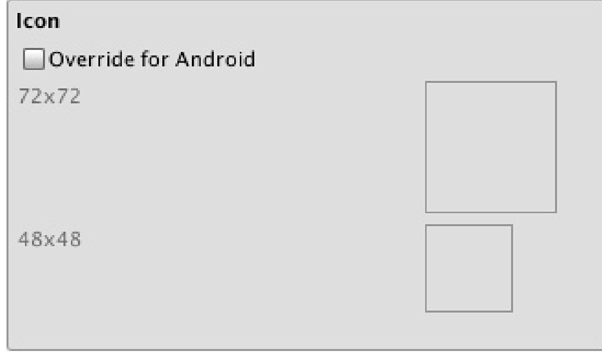
Resolution and Presentation
Resolution
Default Orientation* Portrait
Status Bar
Status Bar Hidden
Use 32-bit Display Buffer*
Use 24-bit Depth Buffer
Show Loading Indicator Don't Show
* Shared setting between multiple platforms.

*Resolution and presentation for your Android project builds.*

***Resolution***

| | |
|---|---|
| **Default Orientation** | (This setting is shared between iOS and Android devices) |
| **Portrait** | The device is in portrait mode, with the device held upright and the home button at the bottom. |
| **Portrait Upside Down** | The device is in portrait mode but upside down, with the device held upright and the home button at the top (only available with Android OS 2.3 and later). |
| **Landscape Right** | The device is in landscape mode, with the device held upright and the home button on the **left** side (only available with Android OS 2.3 and later). |
| **Landscape Left** | The device is in landscape mode, with the device held upright and the home button on the **right** side. |
| **Use 32-bit Display Buffer** | Specifies if Display Buffer should be created to hold 32-bit color values (16-bit by default). Use it if you see banding, or need alpha in your ImageEffects, as they will create RTs in same format as Display Buffer. Not supported on devices running pre-Gingerbread OS (will be forced to 16-bit). |
| **Use 24-bit Depth Buffer** | If set Depth Buffer will be created to hold (at least) 24-bit depth values. Use it only if you see 'z-fighting' or other artifacts, as it may have performance implications. |

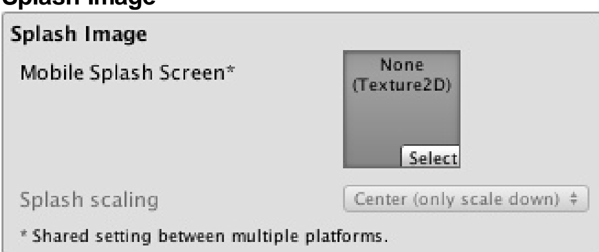
### Icon




*Different icons that your project will have when built.*

| | |
|---|---|
| **Override for Android** | Check if you want to assign a custom icon you would like to be used for your Android game. Different sizes of the icon should fill in the squares below. |

### Splash Image




*Splash image that is going to be displayed when your project is launched.*

| | |
|---|---|
| **Mobile Splash Screen (Pro-only feature)** | Specifies texture which should be used by the iOS Splash Screen. Standard Splash Screen size is 320x480.(This is shared between Android and iOS) |
| **Splash Scaling** | Specifies how will be the splash image scaling on the device. |

### Other Settings

**Other Settings**

**Rendering**
Static Batching ☑
Dynamic Batching ☑

**Identification**
Bundle Identifier* com.Company.Produ
Bundle Version* 1.0
Bundle Version Code 1
Minimum API Level Android 2.0.1 'Eclair' (

**Configuration**
Device Filter ARMv7 only
Install Location Prefer External
Graphics Level* OpenGL ES 2.0
Internet Access Auto
Write Access Internal Only
Scripting Define Symbols

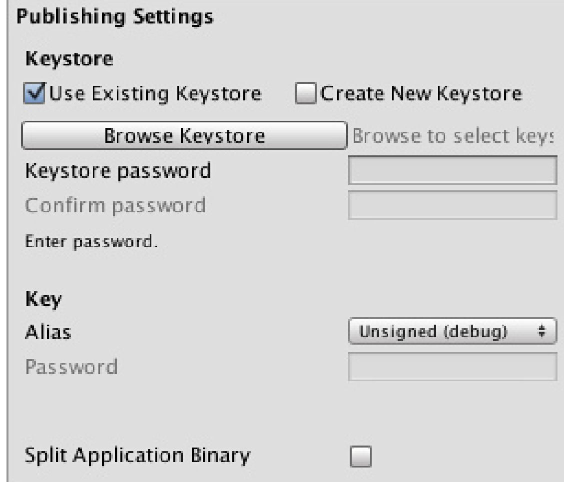
**Optimization**
Api Compatibility Level .NET 2.0 Subset
Stripping Level* Disabled
Enable 'logcat' Profiler ☐
Optimize Mesh Data* ☐

* Shared setting between multiple platforms.

| | |
|---|---|
| ***Rendering*** | |
| **Static Batching** | Set this to use Static batching on your build (Activated by default). Pro-only feature. |
| **Dynamic Batching** | Set this to use Dynamic Batching on your build (Activated by default). |
| ***Identification*** | |
| **Bundle Identifier** | The string used in your provisioning certificate from your Apple Developer Network account(This is shared between iOS and Android) |
| **Bundle Version** | Specifies the build version number of the bundle, which identifies an iteration (released or unreleased) of the bundle. This is a monotonically increased string, comprised of one or more period-separated(This is shared between iOS and Android) |
| **Bundle Version Code** | An internal version number. This number is used only to determine whether one version is more recent than another, with higher numbers indicating more recent versions. This is not the version number shown to users; that number is set by the versionName attribute. The value must be set as an integer, such as "100". You can define it however you want, as long as each successive version has a higher number. For example, it could be a build number. Or you could translate a version number in "x.y" format to an integer by encoding the "x" and "y" separately in the lower and upper 16 bits. Or you could simply increase the number by one each time a new version is released. |
| **Minimum API Level** | Minimum API version required to support the build. |
| ***Configuration*** | |
| **Graphics Level** | Select either ES 1.1 ('fixed function') or ES 2.0 ('shader based') Open GL level. When using the AVD (emulator) only ES 1.x is supported. |
| **Install Location** | Specifies application install location on the device (for detailed information, please refer to http://developer.android.com/guide/appendix/install-location.html). |
| **Automatic** | Let OS decide. User will be able to move the app back and forth. |
| **Prefer External** | Install app to external storage (SD-Card) if possible. OS does not guarantee that will be possible; if not, the app will be installed to internal memory. |
| **Force Internal** | Force app to be installed into internal memory. User will be unable to move the app to external storage. |
| **Internet Access** | When set to Require, will enable networking permissions even if your scripts are not using this. Automatically enabled for development builds. |

| | |
|---|---|
| **Write Access** | When set to External (SDCard), will enable write access to external storage such as the SD-Card. Automatically enabled for development builds. |
| **Scripting Define Symbols** | Custom compilation flags (see the platform dependent compilation page for details). |
| ***Optimization*** | |
| **Api Compatibility Level** | Specifies active .NET API profile |
| **.Net 2.0** | .Net 2.0 libraries. Maximum .net compatibility, biggest file sizes |
| **.Net 2.0 Subset** | Subset of full .net compatibility, smaller file sizes |
| **Stripping Level (Pro-only feature)** | Options to strip out scripting features to reduce built player size(This setting is shared between iOS and Android Platforms) |
| **Disabled** | No reduction is done. |
| **Strip Assemblies** | Level 1 size reduction. |
| **Strip ByteCode (iOS only)** | Level 2 size reduction (includes reductions from Level 1). |
| **Use micro mscorlib** | Level 3 size reduction (includes reductions from Levels 1 and 2). |
| **Enable "logcat" profiler** | Enable this if you want to get feedback from your device while testing your projects. So adb logcat prints logs from the device to the console (only available in development builds). |
| **Optimize Mesh Data** | Remove any data from meshes that is not required by the material applied to them (tangents, normals, colors, UV). |

### Publishing Settings




*Publishing settings for Android Market*

| | |
|---|---|
| ***Keystore*** | |
| **Use Existing Keystore / Create New Keystore** | Use this to choose whether to create a new Keystore or use an existing one. |
| **Browse Keystore** | Lets you select an existing Keystore. |
| **Keystore password** | Password for the Keystore. |
| **Confirm password** | Password confirmation, only enabled if the Create New Keystore option is chosen. |
| ***Key*** | |
| **Alias** | Key alias |
| **Password** | Password for key alias |
| **Split Application Binary** | Flag to split the application into expansion files. Useful only with Google Play Store when the finished build exceeds 50MB. |

Note that for security reasons, Unity will save neither the keystore password nor the key password. Also, note that the signing must be done from Unity's player settings - using jarsigner will not work.

### Flash

***Resolution And Presentation***

Resolution and Presentation
Resolution
Default Screen Width* 960
Default Screen Height* 600
* Shared setting between multiple platforms.

***Resolution***

| | |
|---|---|
| **Default Screen Width** | Screen Width the player will be generated with. |
| **Default Screen Height** | Screen Height the plater will be generated with. |

***Other Settings***

Other Settings
Rendering
Static Batching
Dynamic Batching
Optimization
Api Compatibility Level Flash Player Subset
Stripping Level Strip ByteCode
Strip Physics Code
Optimize Mesh Data*
* Shared setting between multiple platforms.

***Optimization***

| | |
|---|---|
| **Stripping** | Bytecode can optionally be stripped during the build. |
| **Strip Physics Code** | Remove physics engine code from the build when not required. |
| **Optimize Mesh Data** | Remove any data from meshes that is not required by the material applied to them (tangents, normals, colors, UV). |

## Google Native Client

***Resolution and Presentation***

Resolution and Presentation
Resolution
Default Screen Width* 960
Default Screen Height* 600
* Shared setting between multiple platforms.

***Resolution***

| | |
|---|---|
| **Default Screen Width** | Screen Width the player will be generated with. |
| **Default Screen Height** | Screen Height the plater will be generated with. |

***Icon***

*Different icons that your project will have when built.*

| | |
|---|---|
| **Override for Web** | Check if you want to assign a custom icon you would like to be used for your Native Client game. Different sizes of the icon should fill in the squares. |

***Other Settings***

| | |
|---|---|
| ***Rendering*** | |
| **Static Batching** | Set this to use Static batching on your build (Inactive by default in webplayers). Unity Pro only. |
| **Dynamic Batching** | Set this to use Dynamic Batching on your build (Activated by default). |
| ***Configuration*** | |
| **Scripting Define Symbols** | Custom compilation flags (see the platform dependent compilation page for details). |
| ***Optimization*** | |
| **API Compatibility Level** | |
| **.Net 2.0** | .Net 2.0 libraries. Maximum .net compatibility, biggest file sizes |
| **.Net 2.0 Subset** | Subset of full .net compatibility, smaller file sizes |
| **Strip Physics Code** | Remove physics engine code from the build when not required. |
| **Optimize Mesh Data** | Remove any data from meshes that is not required by the material applied to them (tangents, normals, colors, UV). |

## Details

## Desktop

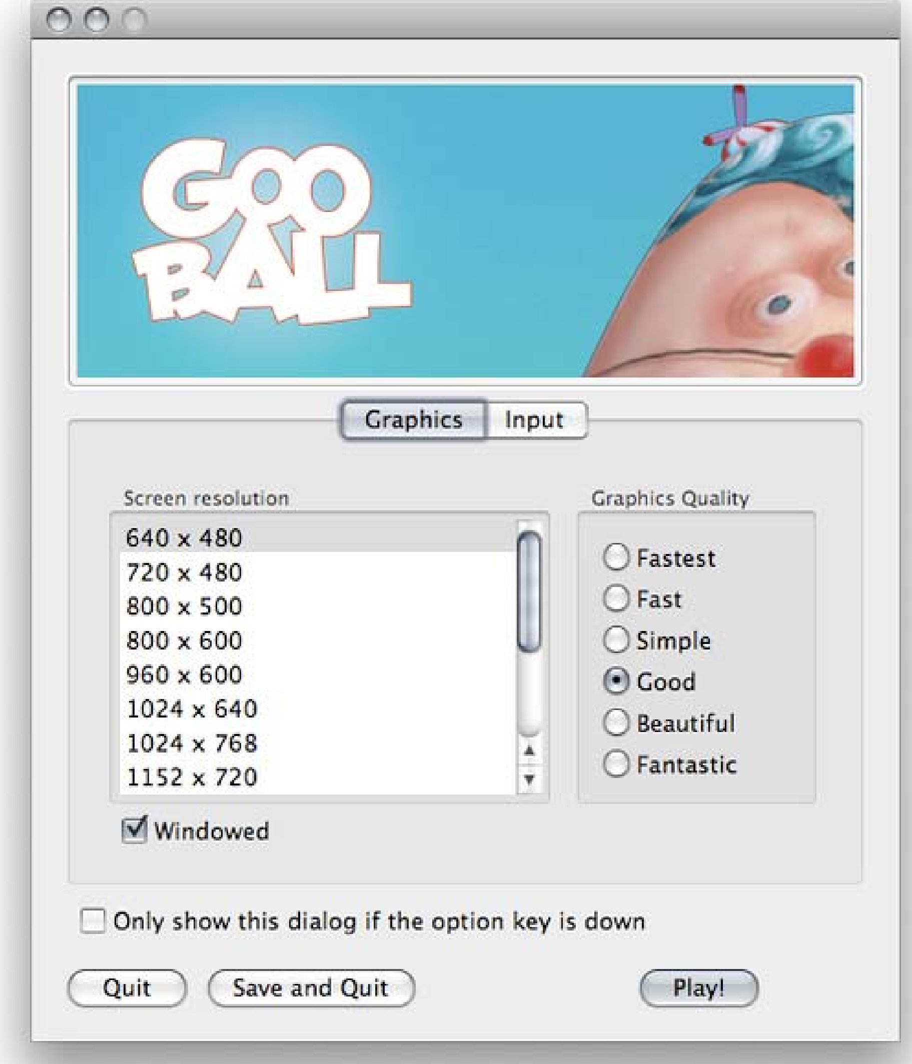
The Player Settings window is where many technical preference defaults are set. See also Quality Settings where the different graphics quality levels can be set up.

**Publishing a web player**

**Default Web Screen Width** and **Default Web Screen Height** determine the size used in the html file. You can modify the size in the html file later.

**Default Screen Width** and **Default Screen Height** are used by the Web Player when entering fullscreen mode through the context menu in the Web Player at runtime.

**Customizing your Resolution Dialog**




*The Resolution Dialog, presented to end-users*

You have the option of adding a custom banner image to the Screen Resolution Dialog in the Standalone Player. The maximum image size is 432 x 163 pixels. The image will not be scaled up to fit the screen selector. Instead it will be centered and cropped.

**Publishing to Mac App Store**

**Use Player Log** enables writing a log file with debugging information. This is useful to find out what happened if there are problems with your game. When publishing games for Apple's Mac App Store, it is recommended to turn this off, because Apple may reject your submission otherwise. See this manual page for further information about log files.

**Use Mac App Store Validation** enables receipt validation for the Mac App Store. If this is enabled, your game will only run when it contains a valid receipt from the Mac App Store. Use this when submitting games to Apple for publishing on the App Store. This prevents people from running the game on any computer then the one it was purchased on. Note that this feature does not implement any strong copy protection. In particular, any potential crack against one Unity game would work against any other Unity content. For this reason, it is recommended that you implement your own receipt validation code on top of this

using Unity's plugin feature. However, since Apple requires plugin validation to initially happen before showing the screen setup dialog, you should still enable this check, or Apple might reject your submission.

## iOS

**Bundle Identifier**
The **Bundle Identifier** string must match the provisioning profile of the game you are building. The basic structure of the identifier is **com.CompanyName.GameName**. This structure may vary internationally based on where you live, so always default to the string provided to you by Apple for your Developer Account. Your GameName is set up in your provisioning certificates, that are manageable from the Apple iPhone Developer Center website. Please refer to the Apple iPhone Developer Center website for more information on how this is performed.

**Stripping Level (Pro-only)**
Most games don't use all necessary dlls. With this option, you can strip out unused parts to reduce the size of the built player on iOS devices. If your game is using classes that would normally be stripped out by the option you currently have selected, you'll be presented with a Debug message when you make a build.

**Script Call Optimization**
A good development practice on iOS is to never rely on exception handling (either internally or through the use of try/catch blocks). When using the default **Slow and Safe** option, any exceptions that occur on the device will be caught and a stack trace will be provided. When using the **Fast but no Exceptions** option, any exceptions that occur will crash the game, and no stack trace will be provided. However, the game will run faster since the processor is not diverting power to handle exceptions. When releasing your game to the world, it's best to publish with the **Fast but no Exceptions** option.

## Android

**Bundle Identifier**
The **Bundle Identifier** string is the unique name of your application when published to the Android Market and installed on the device. The basic structure of the identifier is **com.CompanyName.GameName**, and can be chosen arbitrarily. In Unity this field is shared with the iOS Player Settings for convenience.

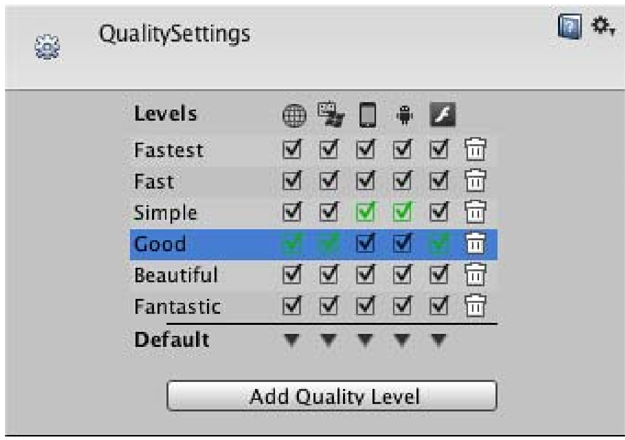
**Stripping Level (Pro-only)**
Most games don't use all the functionality of the provided dlls. With this option, you can strip out unused parts to reduce the size of the built player on Android devices.

Page last updated: 2012-09-25

---

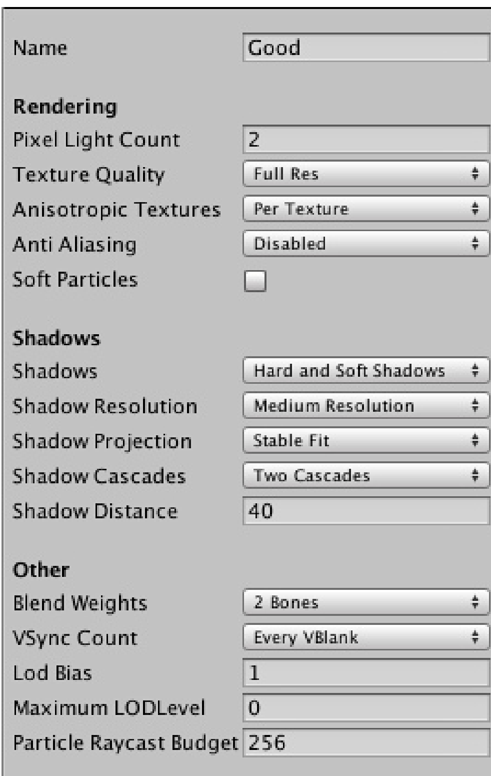
# class-QualitySettings

Unity allows you to set the level of graphical quality it will attempt to render. Generally speaking, quality comes at the expense of framerate and so it may be best not to aim for the highest quality on mobile devices or older hardware since it will have a detrimental effect on gameplay. The **Quality Settings** inspector (menu: **Edit->Project Settings->Quality**) is split into two main areas. At the top, there is the following matrix:-




Unity lets you assign a name to a given combination of quality options for easy reference. The rows of the matrix let you choose which of the different platforms each quality level will apply to. The Default row at the bottom of the matrix is not a quality level in itself but rather sets the default quality level used for each platform (a green checkbox in a column denotes the level currently chosen for that platform). Unity comes with six quality levels pre-enabled but you can add your own levels using the button below the matrix. You can use the trashcan icon (the rightmost column) to delete an unwanted quality level.

You can click on the name of a quality level to select it for editing, which is done in the panel below the settings matrix:-



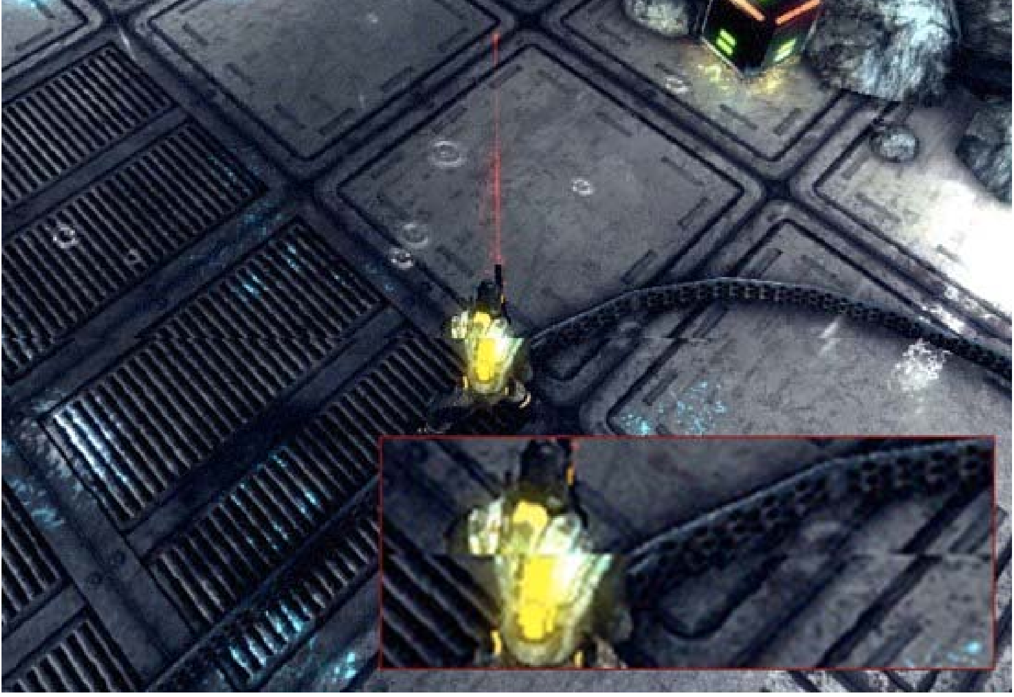

The quality options you can choose for a quality level are as follows:-

| | |
|---|---|
| Name | The name that will be used to refer to this quality level |
| **Pixel Light Count** | The maximum number of pixel lights when Forward Rendering is used. |
| **Texture Quality** | This lets you choose whether to display textures at maximum resolution or at a fraction of this (lower resolution has less processing overhead). The options are **Full Res**, **Half Res**, **Quarter Res** and **Eighth Res**. |
| **Anisotropic Textures** | This enables if and how anisotropic textures will be used. |
| **Disabled** | Anisotropic textures are not used. |
| **Per Texture** | Anisotropic rendering will be enabled separately for each Texture. |
| **Forced On** | Anisotropic textures are always used. |
| **AntiAliasing** | This sets the level of antialiasing that will be used. The options are **2x**, **4x** and **8x** multi-sampling. |
| **Soft Particles** | Should soft blending be used for particles? |

| | |
|---|---|
| **Shadows** | This determines which type of shadows should be used |
| **Hard and Soft Shadows** | Both hard and soft shadows will be rendered. |
| **Hard Shadows Only** | Only hard shadows will be rendered. |
| **Disable Shadows** | No shadows will be rendered. |
| **Shadow resolution** | Shadows can be rendered at several different resolutions: **Low**, **Medium**, **High** and **Very High**. The higher the resolution, the greater the processing overhead. |
| **Shadow Projection** | There are two different methods for projecting shadows from a directional light. **Close Fit** renders higher resolution shadows but they can sometimes wobble slightly if the camera moves. **Stable Fit** renders lower resolution shadows but they don't wobble with camera movements. |
| **Shadow Cascades** | The number of shadow cascades can be set to zero, two or four. A higher number of cascades gives better quality but at the expense of processing overhead (see the Directional Shadows page for further details). |
| **Shadow Distance** | The maximum distance from camera at which shadows will be visible. Shadows that fall beyond this distance will not be rendered. |
| **Blend Weights** | The number of bones that can affect a given vertex during an animation. The available options are one, two or four bones. |
| **VSync Count** | Rendering can be synchronised with the refresh rate of the display device to avoid "tearing" artifacts (see below). You can choose to synchronise with every vertical blank (VBlank), every second vertical blank or not to synchronise at all. |
| **LOD Bias** | LOD levels are chosen based on the onscreen size of an object. When the size is between two LOD levels, the choice can be biased toward the less detailed or more detailed of the two models available. This is set as a fraction from 0 to 1 - the closer it is to zero, the more the bias is toward the less detailed model. |
| **Maximum LOD Level** | The highest LOD that will be used by the game. Models which have a LOD above this level will not be used and omitted from the build (which will save storage and memory space). |
| **Particle Raycast Budget** | The maximum number of raycasts to use for approximate particle system collisions (those with **Medium** or **Low** quality). See Particle System Collision Module. |

## Tearing

The picture on the display device is not continuously updated but rather the updates happen at regular intervals much like frame updates in Unity. However, Unity's updates are not necessarily synchronised with those of the display, so it is possible for Unity to issue a new frame while the display is still rendering the previous one. This will result in a visual artifact called "tearing" at the position onscreen where the frame change occurs.



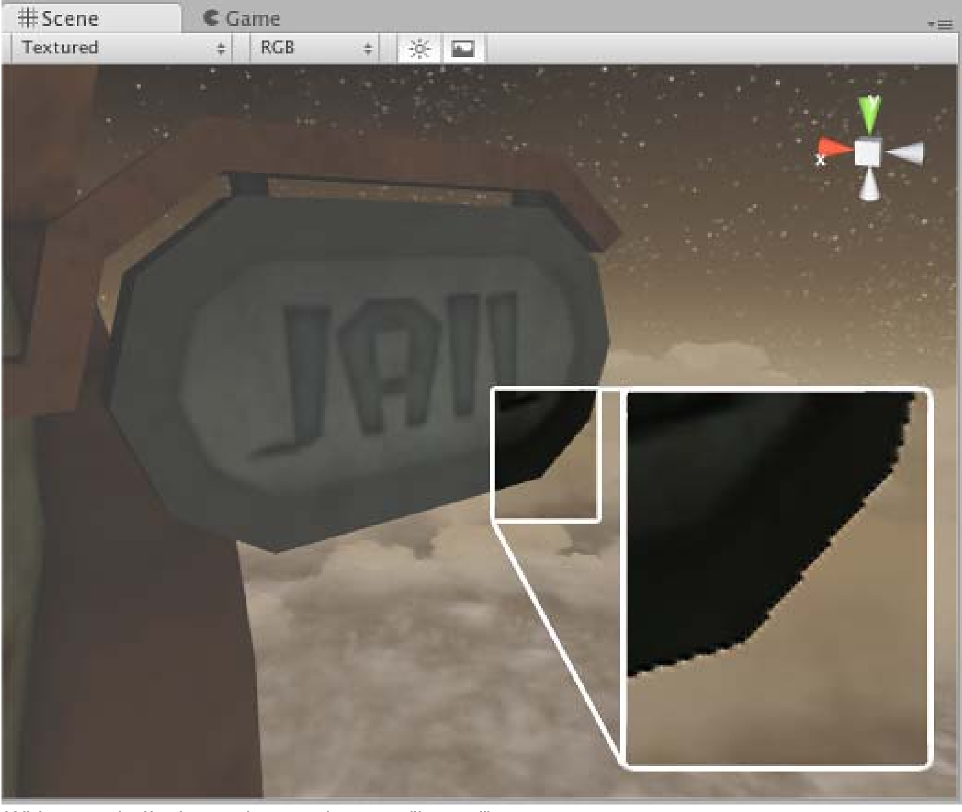
*Simulated example of tearing. The shift in the picture is clearly visible in the magnified portion.*

It is possible to set Unity to switch frames only during the period where the display device is not updating, the so-called "vertical blank". The VSync option on the Quality Settings synchronises frame switches with the device's vertical blank or

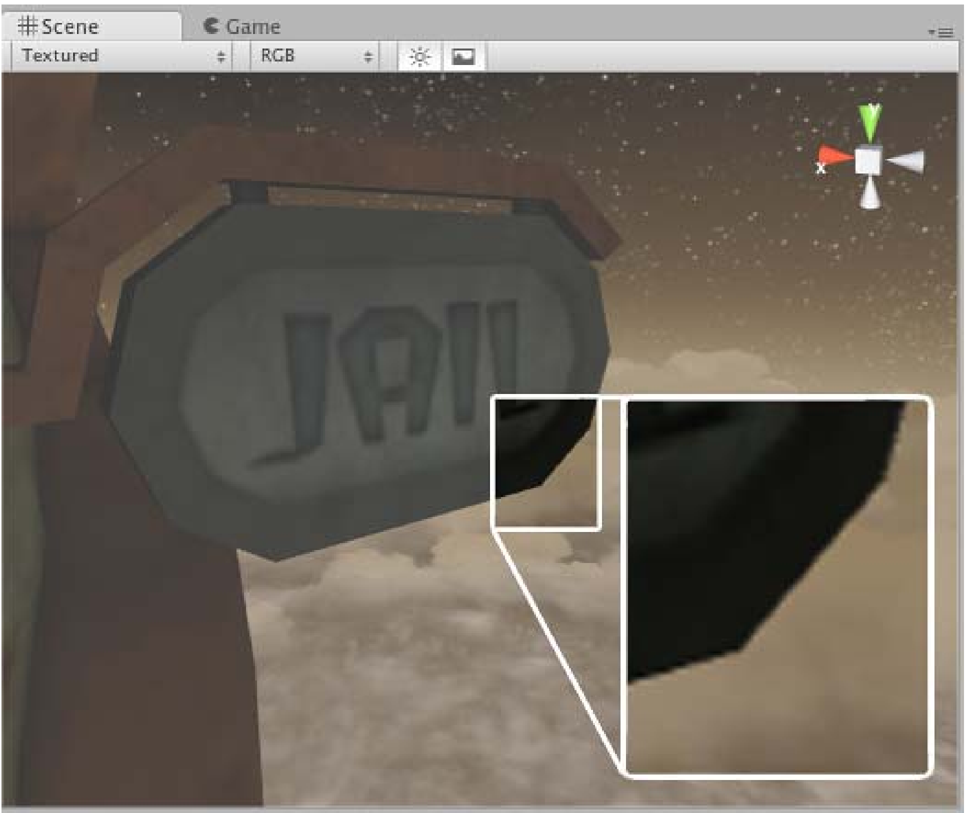
optionally with every other vertical blank. The latter may be useful if the game requires more than one device update to complete the rendering of a frame.

**Anti-aliasing**

Anti aliasing improves the appearance of polygon edges, so they are not "jagged", but smoothed out on the screen. However, it incurs a performance cost for the graphics card and uses more video memory (there's no cost on the CPU though). The level of anti-aliasing determines how smooth polygon edges are (and how much video memory does it consume).



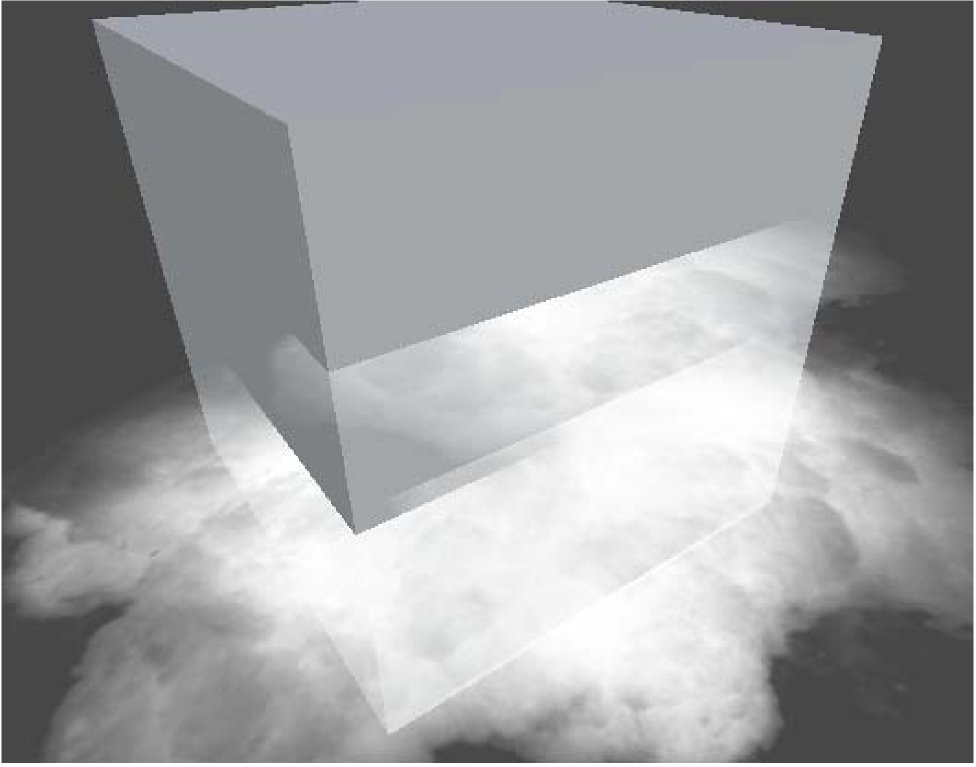

*Without anti-aliasing, polygon edges are "jagged".*



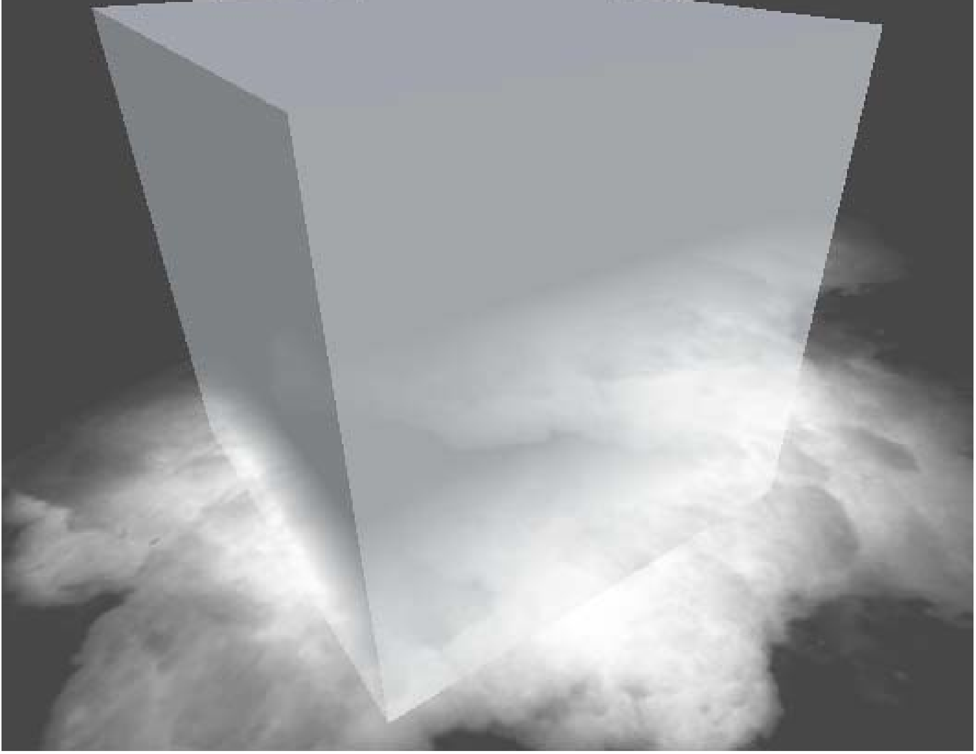
*With 6x anti-aliasing, polygon edges are smoothed out.*

**Soft Particles**

Soft Particles fade out near intersections with other scene geometry. This looks much nicer, however it's more expensive to compute (more complex pixel shaders), and only works on platforms that support depth textures. Furthermore, you have to use Deferred Lighting rendering path, or make the camera render depth textures from scripts.



*Without Soft Particles - visible intersections with the scene.*



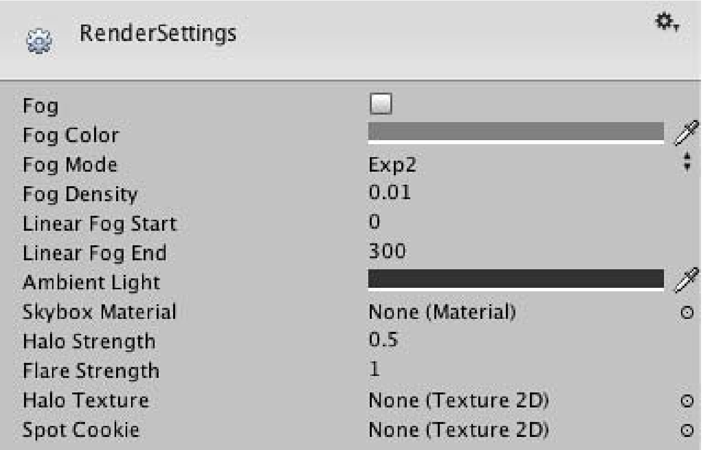
*With Soft Particles - intersections fade out smoothly.*

Page last updated: 2012-11-16

## class-RenderSettings

The **Render Settings** contain default values for a range of visual elements in your scene, like **Lights** and **Skyboxes**.

To see the Render Settings choose **Edit->Render Settings** from the menu bar.




### Properties

| | |
|---|---|
| **Fog** | If enabled, fog will be drawn throughout your scene. |
| **Fog Color** | Color of the fog. |
| **Fog Mode** | Fog mode: Linear, Exponential (Exp) or Exponential Squared (Exp2). This controls the way fog fades in with distance. |
| **Fog Density** | Density of the fog; only used by Exp and Exp2 fog modes. |
| **Linear Fog Start/End** | Start and End distances of the fog; only used by Linear fog mode. |
| **Ambient Light** | Color of the scene's ambient light. |
| **Skybox Material** | Default skybox that will be rendered for cameras that have no skybox attached. |
| **Halo Strength** | Size of all light halos in relation to their **Range**. |
| **Flare Strength** | Intensity of all flares in the scene. |
| **Halo Texture** | Reference to a Texture that will appear as the glow for all Halos in lights. |
| **Spot Cookie** | Reference to a Texture2D that will appear as the cookie mask for all Spot lights. |

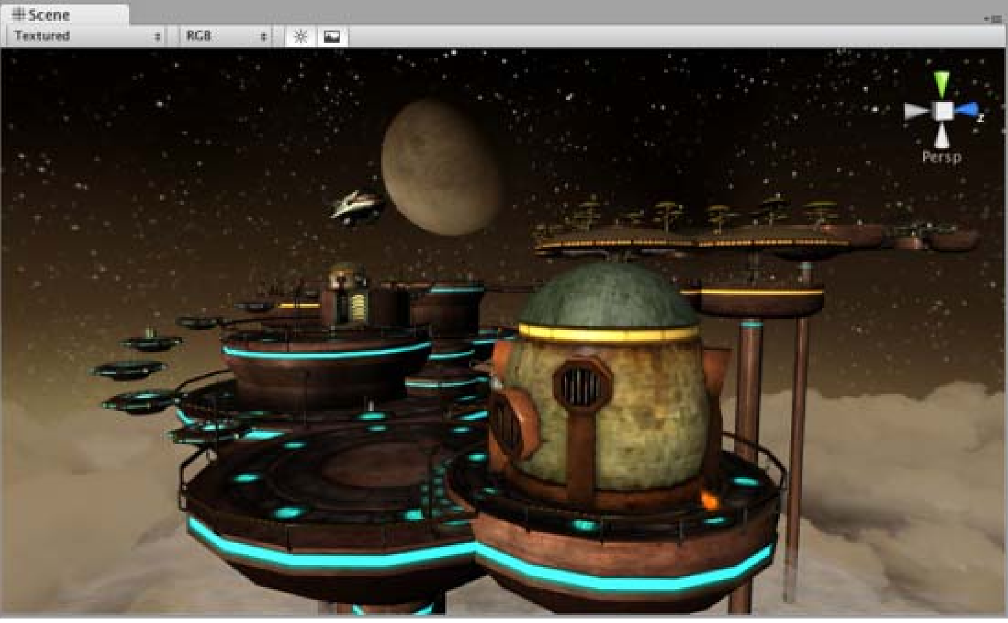
### Details

The Render Settings is used to define some basic visual commonalities of any individual scene in your project. Maybe you have two levels in the same environment: one at daytime and one at nighttime. You can use the same meshes and Prefabs to populate the scene, but you can change the **Ambient Light** to be much brighter at daytime, and much darker at night.

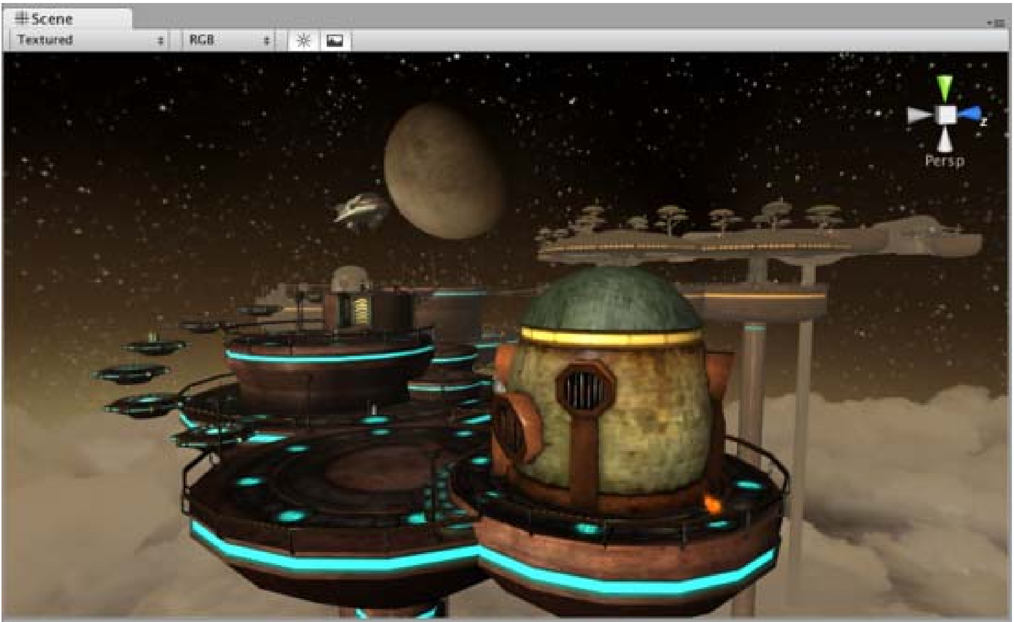
#### Fog

Enabling **Fog** will give a misty haze to your scene. You can adjust the look and color of the Fog with **Fog Density** and **Fog Color**, respectively.

Adding fog is often used to optimize performance by making sure that far away objects fade out and are not drawn. Please note that enabling fog is not enough to enable this performance optimization. To do that you also need to adjust your Camera's **Far Clip Plane**, so that geometry far away will not be drawn. It is best to tweak the fog to look correct first. Then make the Camera's far clip plane smaller until you see the geometry being clipped away before the fog fades it out.



*A scene with Fog turned off*



*The same scene with Fog turned on*

Note that fog is rendered uniformly in orthographic camera mode. This is because in our shaders, we output post-perspective space Z coordinate as the fog coordinate. But post-perspective Z is not really suitable for fog in orthographic cameras. Why do we do this? Because it's fast and does not need any extra computations; handling orthographic cameras would make all shaders be a bit slower.

### Hints

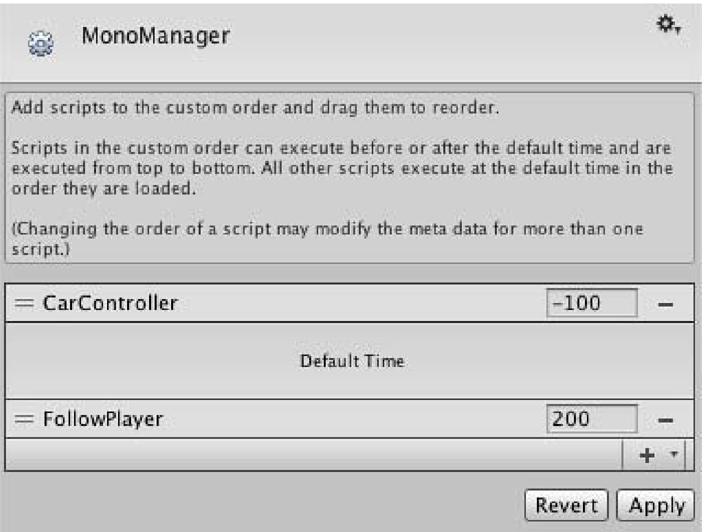
- Don't under-estimate the visual impact your game can make by thoughtfully tweaking the Render Settings!
- Render Settings are per-scene: each scene in your game can have different Render Settings.

Page last updated: 2011-10-24

---

## class-ScriptExecution

By default, the Awake, OnEnable and Update functions of different scripts are called in the order the scripts are loaded (which is arbitrary). However, it is possible to modify this order using the **Script Execution Order** settings.



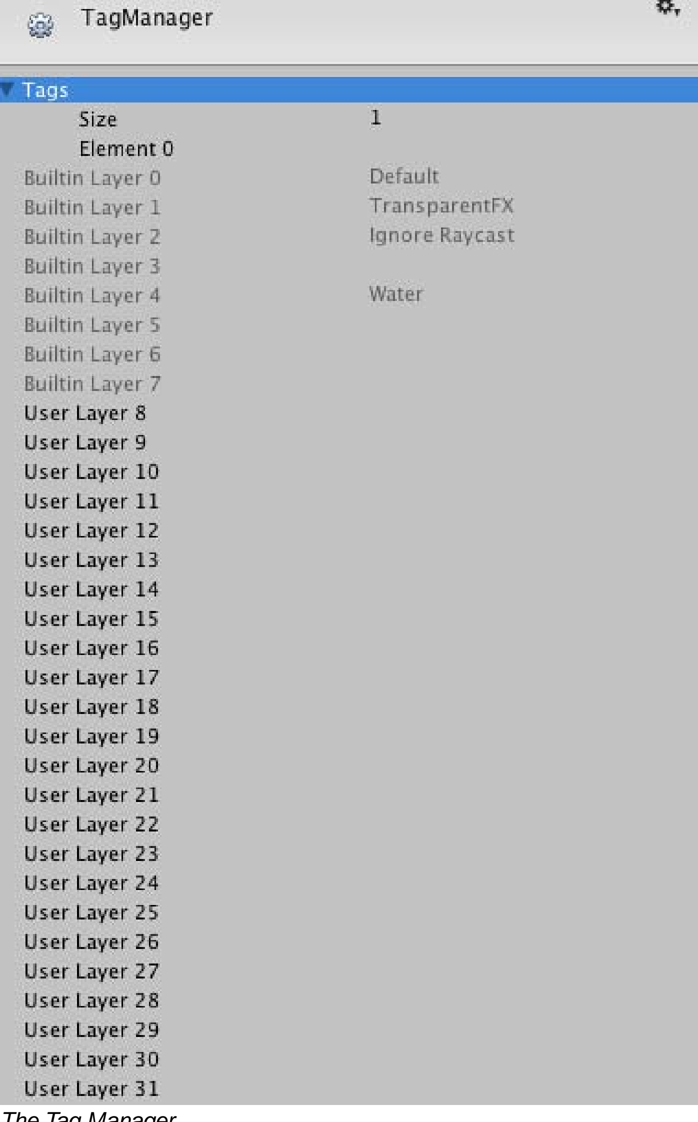

Scripts can be added to the inspector using the Plus "+" button and dragged to change their relative order. Note that it is possible to drag a script either above or below the **Default Time** bar; those above will execute ahead of the default time while those below will execute after. The ordering of scripts in the dialog from top to bottom determines their execution order. All scripts not in the dialog execute in the default time slot in arbitrary order.

Page last updated: 2011-10-21

---

## class-TagManager

The **Tag Manager** allows you to set up **Layers** and **Tags**. To see it choose **Edit->Project Settings->Tags**.



*The Tag Manager*

### Properties

| | |
|---|---|
| **Tags** | You can add new elements by typing in the last element. |
| **User Layer 8-31** | You can add custom named User Layers |

### Details

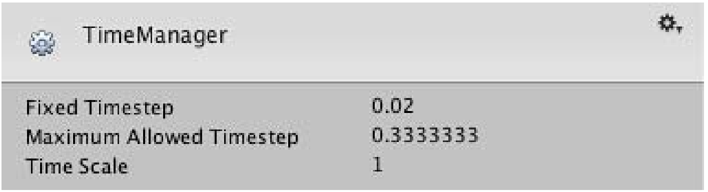
Layers can be used to cast rays, render, or apply lighting to certain groups of objects only. You can choose the Layer in the GameObject inspector. More information about how to use Layers can be found here, while Tags are covered here.

Tags are used to quickly find objects from scripts, utilizing the Tag name. When a new tag is added, you can choose it from the GameObject tag popup.

Page last updated: 2011-10-21

---

## class-TimeManager




*The Time Manager*

## Properties

**Fixed Timestep** A framerate-independent interval that dictates when physics calculations and **FixedUpdate()** events are performed.

**Maximum Allowed Timestep** A framerate-independent interval that caps the worst case scenario when frame-rate is low. Physics calculations and **FixedUpdate()** events will not be performed for longer time than specified.

**Time Scale** The speed at which time progress. Change this value to simulate bullet-time effects. A value of 1 means real-time. A value of .5 means half speed; a value of 2 is double speed.

## Details

**Fixed Timestep**
Fixed time stepping is very important for stable physics simulation. Not all computers are made equal, and different hardware configurations will run Unity games with varying performance. Therefore, physics must be calculated independently of the game's frame rate. Physics calculations like collision detection and Rigidbody movement are performed in discrete fixed time steps that are not dependent on frame rate. This makes the simulation more consistent across different computers or when changes in the frame rate occur. For example, the frame rate can drop due to an appearance of many game onscreen, or because the user launched another application in the background.

Here's how the fixed time step is calculated. Before every frame is drawn onscreen, Unity advances the fixed time by fixed delta time and performs physics calculations until it reaches the current time. This directly correlates to the **Fixed Timestep** property. The smaller the value of **Fixed Timestep**, the more frequently physics will be calculated. The number of Fixed frames per second can be calculated by dividing 1 by **Fixed Timestep**. Therefore, 1 / 0.02 = 50 fixed frames per second and 1 / 0.05 = 20 fixed frames per second.

Simply put, a smaller fixed update value leads to more accurate physics simulation but is heavier on the CPU.

**Maximum Allowed Timestep**
Fixed time stepping ensures stable physics simulation. However it can cause negative impact on performance if game is heavy on physics and is already running slow or occasionally dips to low frame rate. Longer the frame takes to process - more fixed update steps will have to be executed for the next frame. This results in performance degradation. To prevent such scenario Unity iOS introduced **Maximum Allowed Timestep** which ensures that physics calculations will not run longer than specified threshold.

If frame takes longer to process than time specified in **Maximum Allowed Timestep**, then physics will "pretend" that frame took only **Maximum Allowed Timestep** seconds. In other words if frame rate drops below some threshold, then rigid bodies will slow down a bit allowing CPU to catch up.

**Maximum Allowed Timestep** affects both physics calculation and **FixedUpdate()** events.

**Maximum Allowed Timestep** is specified in seconds as **Fixed Timestep**. Therefore setting 0.1 will make physics and **FixedUpdate()** events to slow down, if frame rate dips below 1 / 0.1 = 10 frames per second.

**Typical scenario**

1. Let's assume **Fixed Timestep** is 0.01, which means that physx, fixedUpdate and animations should be processed every 10 ms.
2. When your frame time is ~33 ms then fixed loop is executed 3 times per visual frame on average.
3. But frametime isn't fixed constant and depends on many factors including your scene state, OS background taks, etc.
4. Because of 3. reasons frametime sometimes can reach 40-50 ms, which means that fixed step loop will be executed 4-5 times.
5. When your fixed timestep tasks are pretty heavy then time spent on physx, fixedUpdates and animations extend your frametime by another 10 ms, which means one more additional iteration of all these fixed timestep tasks.
6. In some unlucky cases process described in 5. could extend to 10 and more times of processing fixed step loop.

7. That's why **Maximum Allowed Timestep** was introduced, it is the method to limit how much times physx, fixedUpdates and animations can be processed during single visual frame. If you have **Maximum Allowed Timestep** set to 100 ms and your **Fixed Timestep** is 10 ms, then for fixed step tasks will be executed up to 10 times per visual frame. So sometimes small performance hitch could trigger big performance hitch because of increased fixed timestep iteration count. By decreasing **Maximum Allowed Timestep** to 30 ms, you are limiting max fixed step iteration count to 3 and this means that your physx, fixedUpdate and animation won't blow your frametime up very much, but there is some negative effect of this limiting. Your animations and physics will slow down a bit when performance hitch occurs.

### Hints

- Give the player control over time by changing **Time Scale** dynamically through scripting.
- If your game is physics heavy or spends significant amount of time in **FixedUpdate()** events, then set **Maximum Allowed Timestep** to 0.1. This will prevent physics from driving your game below 10 frames per second.

Page last updated: 2011-10-21

---

## comp-MeshGroup

3D **Meshes** are the main graphics primitive of Unity. Various components exist in Unity to render regular or skinned meshes, trails or 3D lines.

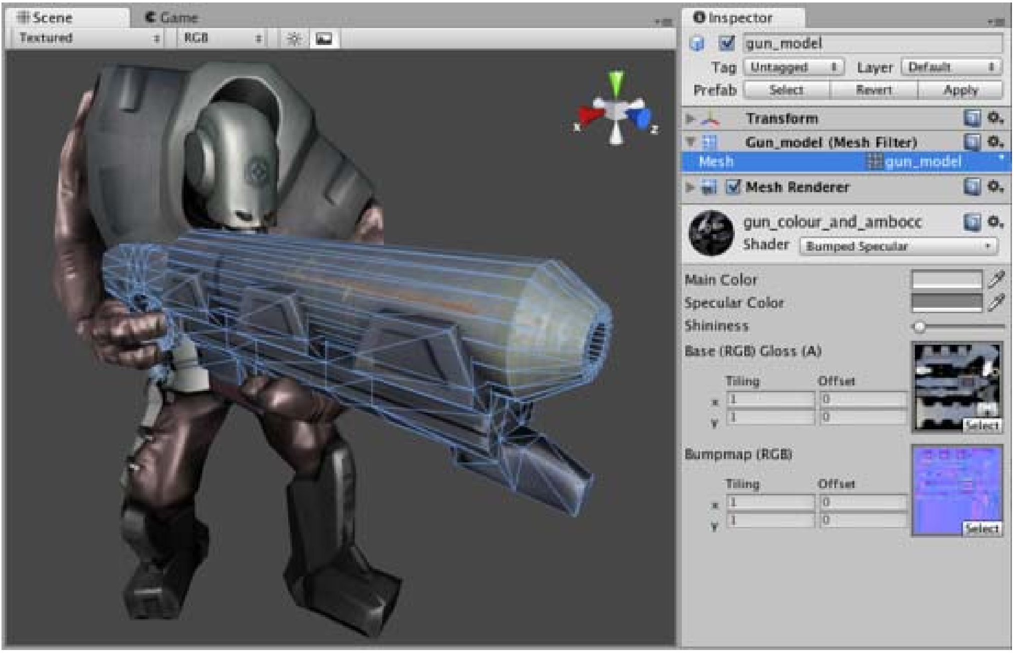
- Mesh Filter
- Mesh Renderer
- Skinned Mesh Renderer
- Text Mesh

Page last updated: 2007-07-26

---

## class-MeshFilter

The **Mesh Filter** takes a mesh from your assets and passes it to the Mesh Renderer for rendering on the screen.




*A Mesh Filter combined with a Mesh Renderer makes the model appear on screen.*

### Properties

| | |
|---|---|
| **Mesh** | Reference to a mesh that will be rendered. The **Mesh** is located in your Project Folder. |

### Details

When importing mesh assets, Unity automatically creates a Skinned Mesh Renderer if the mesh is skinned, or a Mesh Filter along with a Mesh Renderer, if it is not.

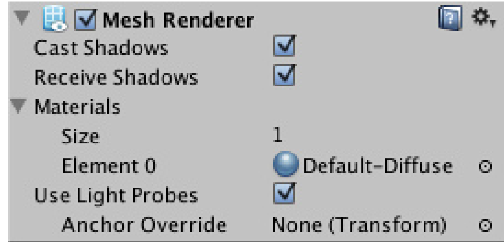
To see the Mesh in your scene, add a Mesh Renderer to the GameObject. It should be added automatically, but you will have to manually re-add it if you remove it from your object. If the Mesh Renderer is not present, the Mesh will still exist in your scene (and computer memory) but it will not be drawn.

Page last updated: 2010-09-14

---

# class-MeshRenderer

The **Mesh Renderer** takes the geometry from the Mesh Filter and renders it at the position defined by the object's Transform component.




### Properties

| | |
|---|---|
| **Cast Shadows** (Pro only) | If enabled, this **Mesh** will create shadows when a shadow-creating Light shines on it |
| **Receive Shadows** (Pro only) | If enabled, this Mesh will display any shadows being cast upon it |
| **Materials** | A list of **Materials** to render model with |
| **Use Light Probes** | Enable probe-based lighting for this mesh |
| **Anchor Override** | A **Transform** used to determine the interpolation position when the light probe system is used |

### Details

Meshes imported from 3D packages can use multiple Materials. For each Material there is an entry in Mesh Renderer's Materials list, so each submesh in the Mesh is rendered with a different material. If there are more materials assigned to the MeshRenderer then submeshes in the Mesh, then the first submesh will be rendered with each of the remaining materials - this lets you set up multi-pass rendering with multiple materials.

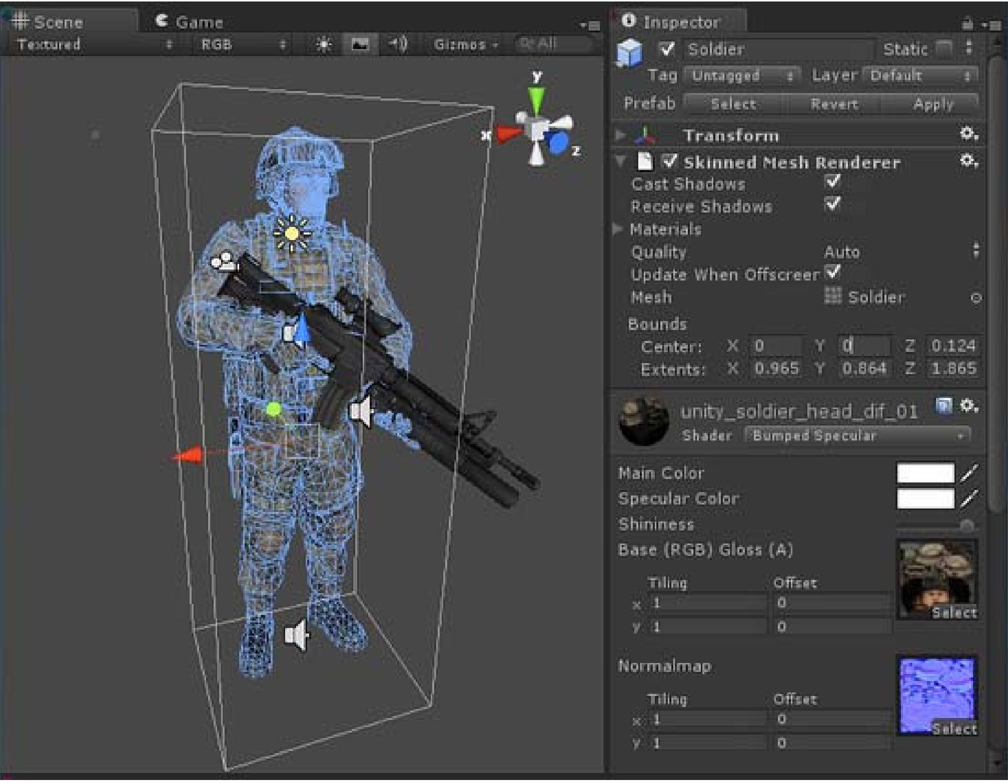
A mesh can receive light from the **light probe** system if the **Use Light Probes** option is enabled (see the light probes manual page for further details). A single point is used as the mesh's notional position for light probe interpolation. By default, this is the centre of the mesh's bounding box, but you can override this by dragging a **Transform** to the **Anchor Override** property. It may be useful to set the anchor in cases where an object contains two adjoining meshes; since each mesh has a separate bounding box, the two will be lit discontinuously at the join by default. However, if you set both meshes to use the same anchor point, they will be consistently lit.

Page last updated: 2012-11-29

---

# class-SkinnedMeshRenderer

The **Skinned Mesh Renderer** is automatically added to imported meshes when the imported mesh is skinned.




*An animated character rendered using the Skinned Mesh Renderer*

## Properties

| | |
|---|---|
| **Cast Shadows** (Pro only) | If enabled, this **Mesh** will create shadows when a shadow-creating Light shines on it |
| **Receive Shadows** (Pro only) | If enabled, this Mesh will display any shadows being cast upon it |
| **Materials** | A list of **Materials** to render model with. |
| **Quality** | The maximum amount of bones affecting every vertex. |
| **Update When Offscreen** | If enabled, the Skinned Mesh will be updated when offscreen. If disabled, this also disables updating animations. |
| **Bounds** | These bounds are use for determining when skinned mesh is offscreen. Bounding box is also displayed in the SceneView. Bounds are precalculated on import based on Mesh and animations in the model file. |
| **Mesh** | Meshed used by this renderer. |

## Details

Skinned Meshes are used for rendering characters. Characters are animated using bones, and every bone affects a part of the mesh. Multiple bones can affect the same vertex and are weighted. The main advantage to using boned characters in Unity is you can enable the bones to be affected by physics, making your characters into ragdolls. You can enable/disable bones via scripting, so your character instantly goes ragdoll when it is hit by an explosion.

*A Skinned Mesh enabled as a Ragdoll*

**Quality**
Unity can skin every vertex with either 1, 2, or 4 bones. 4 bone weights look nicest and are most expensive. 2 Bone weights is a good compromise and can be commonly used in games.

If **Quality** is set to **Automatic**, the Quality Settings **Blend Weights** value will be used. This allows end-users to choose a quality setting that gives them optimum performance.

**Update When Offscreen and Bounds**
By default, skinned meshes that are not visible are not updated. The skinning is not updated until the mesh comes back on screen. This is an important performance optimization - it allows you to have a lot of characters running around not taking up any processing power when they are not visible.

However, visibility is determined from the Mesh's Bounds, which is precalculated on import. Unity takes into account all attached animations for precalcualating bounding volume, but there are cases when Unity can't precalculate Bounds to fit all user's needs, for example (each of these become a problem when they push bones or vertices out of precalculated bounding volume):

- adding animations at run-time;
- using additive animations;
- proceduraly affecting positions of bones;
- using vertex shaders which can push vertices out of precalculated bounds;
- using ragdolls.

In those cases there are two solutions:

1. modify Bounds to match potential bounding volume of your mesh;
2. enable **Update When Offscreen** to skin and render skinned mesh all the time.

You should use fist option most of the time since it has better performance and use second option only if performance is not important in your case or you can't predict the size of your bounding volume (for example when using ragdolls).

In order to make SkinnedMeshes work better with Ragdolls Unity will automatically remap the SkinnedMeshRenderer to the rootbone on import. However Unity only does this if there is a single SkinnedMeshRenderer in the model file. So if you can't attach all SkinnedMeshRenderers to the root bone or a child and you use ragdolls, you should turn off this optimization.

### Hints

- Skinned Meshes currently can be imported from:
  - Maya
  - Cinema4D
  - 3D Studio Max
  - Blender
  - Cheetah 3D
  - XSI
  - Any other tool that supports the FBX format

Page last updated: 2011-08-17

---

## class-TextMesh

The **Text Mesh** generates 3D geometry that displays text strings.

*The Text Mesh **Inspector***

You can create a new Text Mesh from **GameObject->Create Other->3D Text**.

### Properties

| | |
|---|---|
| **Text** | The text that will be rendered |
| **Offset Z** | How far should the text be offset from the transform.position.z when drawing |
| **Character Size** | The size of each character (This scales the whole text) |
| **Line Spacing** | How much space will be in-between lines of text. |
| **Anchor** | Which point of the text shares the position of the Transform. |
| **Alignment** | How lines of text are aligned (Left, Right, Center). |
| **Tab Size** | How much space will be inserted for a tab '\t' character. This is a multiplum of the 'spacebar' character offset. |
| **Font** | The TrueType Font to use when rendering the text. |

### Details

Text Meshes can be used for rendering road signs, graffiti etc. The Text Mesh places text in the 3D scene. To make generic 2D text for GUIs, use a GUI Text component instead.

Follow these steps to create a Text Mesh with a custom Font:

1. Import a font by dragging a TrueType Font - a **.ttf** file - from the Explorer (Windows) or Finder (OS X) into the **Project View**.
2. Select the imported font in the Project View.
3. Choose **GameObject->Create Other->3D Text**.

You have now created a text mesh with your custom TrueType Font. You can scale the text and move it around using the **Scene View's Transform** controls.

**Note:** If you want to change the font for a Text Mesh, need to set the component's font property and also set the texture of the font material to the correct font texture. This texture can be located using the font asset's foldout. If you forget to set the texture then the text in the mesh will appear blocky and misaligned.

### Hints

- When entering text into the **Text** property, you can create a line break by holding **Alt** and pressing **Return**.
- You can download free TrueType Fonts from 1001freefonts.com (download the Windows fonts since they contain TrueType Fonts).
- If you are scripting the **Text** property, you can add line breaks by inserting the escape character "\n" in your strings.

Page last updated: 2012-01-11

---

## comp-NetworkGroup

This group contains all the **Components** that relate to Networked Multiplayer games.

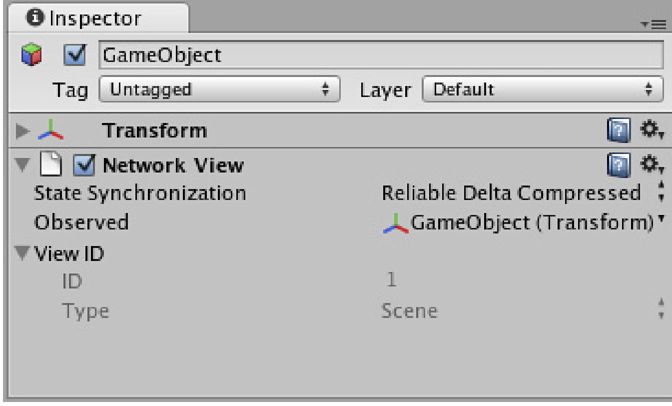
- Network View

Page last updated: 2007-08-23

---

## class-NetworkView

**Network Views** are the gateway to creating networked multiplayer games in Unity. They are simple to use, but they are extremely powerful. For this reason, it is recommended that you understand the fundamental concepts behind networking before you start experimenting with Network Views. You can learn and discover the fundamental concepts in the Network Reference Guide.




*The Network View* ***Inspector***

In order to use any networking capabilities, including **State Synchronization** or **Remote Procedure Calls**, your **GameObject** must have a Network View attached.

### Properties

| | |
|---|---|
| **State Synchronization** | The type of State Synchronization used by this Network View |
| **Off** | No State Synchronization will be used. This is the best option if you only want to send RPCs |
| **Reliable Delta Compressed** | The difference between the last state and the current state will be sent, if nothing has changed nothing will be sent. This mode is ordered. In the case of packet loss, the lost packet is re-sent automatically |
| **Unreliable** | The complete state will be sent. This uses more bandwidth, but the impact of packet loss is minimized |
| **Observed** | The **Component** data that will be sent across the network |

| | |
|---|---|
| **View ID** | The unique identifier for this Network View. These values are read-only in the Inspector |
| **Scene ID** | The number id of the Network View in this particular scene |
| **Type** | Either saved to the **Scene** or **Allocated** at runtime |

## Details

When you add a Network View to a GameObject, you must decide two things

1. What kind of data you want the Network View to send
2. How you want to send that data

### Choosing data to send

The **Observed** property of the Network View can contain a single Component. This can be a **Transform**, an **Animation**, a **RigidBody**, or a script. Whatever the **Observed** Component is, data about it will be sent across the network. You can select a Component from the drop-down, or you can drag any Component header directly to the variable. If you are not directly sending data, just using RPC calls, then you can turn off synchronization (no data directly sent) and nothing needs to be set as the Observed property. RPC calls just need a single network view present so you don't need to add a view specifically for RPC if a view is already present.

### How to send the data

You have 2 options to send the data of the **Observed** Component: **State Synchronization** and **Remote Procedure Calls**.

To use State Synchronization, set **State Synchronization** of the Network View to **Reliable Delta Compressed** or **Unreliable**. The data of the **Observed** Component will now be sent across the network automatically.

**Reliable Delta Compressed** is ordered. Packets are always received in the order they were sent. If a packet is dropped, that packet will be re-sent. All later packets are queued up until the earlier packet is received. Only the difference between the last transmissions values and the current values are sent and nothing is sent if there is no difference.

If it is observing a Script, you must explicitly Serialize data within the script. You do this within the **OnSerializeNetworkView()** function.

```
function OnSerializeNetworkView (stream : BitStream, info : NetworkMessageInfo) {
	var horizontalInput : float = Input.GetAxis ("Horizontal");
	stream.Serialize (horizontalInput);
}
```

The above function always writes (an update from the stream) into horizontalInput, when receiving an update and reads from the variable writing into the stream otherwise. If you want to do different things when receiving updates or sending you can use the **isWriting** attribute of the BitStream class.

```
function OnSerializeNetworkView (stream : BitStream, info : NetworkMessageInfo) {
	var horizontalInput : float = 0.0;
	if (stream.isWriting) {
		// Sending
		horizontalInput = Input.GetAxis ("Horizontal");
		stream.Serialize (horizontalInput);
	} else {

		// Receiving
		stream.Serialize (horizontalInput);
		// ... do something meaningful with the received variable
	}
}
```

**OnSerializeNetworkView** is called according to the **sendRate** specified in the network manager project settings. By default this is 15 times per second.

If you want to use Remote Procedure Calls in your script all you need is a NetworkView component present in the same GameObject the script is attached to. The NetworkView can be used for something else, or in case it's only used for sending

RPCs it can have no script observed and state synchronization turned off. The function which is to be callable from the network must have the **@RPC** attribute. Now, from any script attached to the same GameObject, you call networkView.RPC() to execute the Remote Procedure Call.

```
var playerBullet : GameObject;

function Update () {
	if (Input.GetButtonDown ("Fire1")) {
		networkView.RPC ("PlayerFire", RPCMode.All);
	}
}

@RPC
function PlayerFire () {
	Instantiate (playerBullet, playerBullet.transform.position, playerBullet.transform.rotation);
}
```

RPCs are transmitted reliably and ordered. For more information about RPCs, see the RPC Details page.

### Hints

- Read through the Network Reference Guide if you're still unclear about how to use Network Views
- State Synchronization does not need to be disabled to use Remote Procedure Calls
- If you have more than one Network View and want to call an RPC on a specific one, use **GetComponents(NetworkView)[i].RPC()**.

Page last updated: 2010-09-20

---

## comp-Effects

The effects group contains components that relate to visual effects.

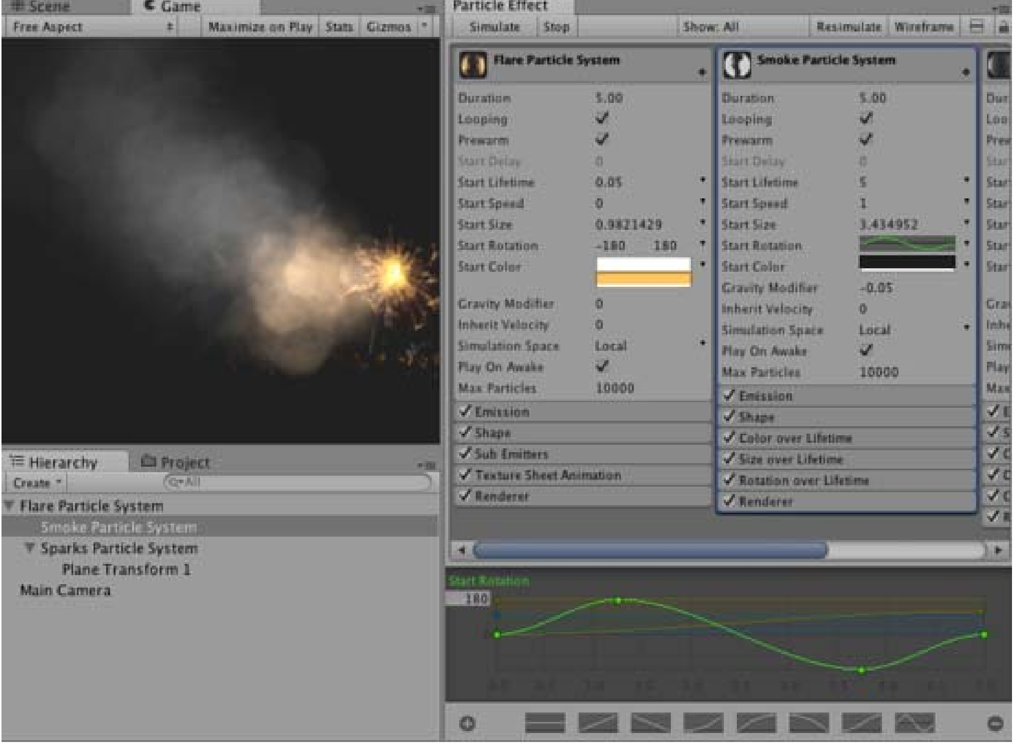
- Particle System (Shuriken)
- Halo
- Lens Flare
- Line Renderer
- Trail Renderer
- Projector
- Particle Systems (Legacy, prior to release 3.5)
  - Ellipsoid Particle Emitter (Legacy)
  - Mesh Particle Emitter (Legacy)
  - Particle Animator (Legacy)
  - Particle Renderer (Legacy)
  - World Particle Collider (Legacy)

Page last updated: 2012-01-12

---

## class-ParticleSystem

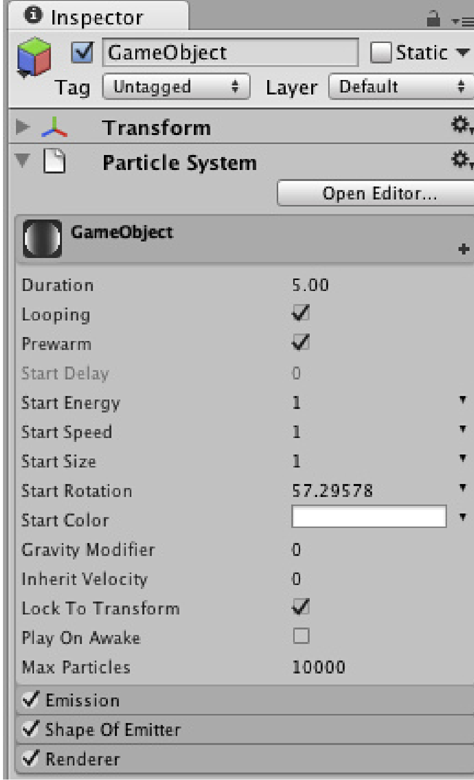
Particle Systems in Unity are used to make clouds of smoke, steam, fire and other atmospheric effects.




You can create a new particle system by creating a **Particle System** GameObject (menu **GameObject** -> **Create Other** -> **Particle System**) or by creating an empty **GameObject** and adding the **ParticleSystem** component to it (in **Component**->**Effects**)

### The Particle System Inspector (Shuriken)

The **Particle System Inspector** shows one particle system at a time (the currently selected one), and it looks like this:



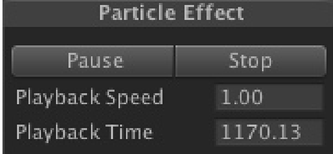

Individual particle systems can take on various complex behaviors by using Modules.

They can also be extended by being grouped together into **Particle Effects**.

If you press the button **Open Editor ...**, this will open up the Extended **Particle Editor**, that shows all of the particle systems under the same root in the scene tree. For more information on particle system grouping, see the section on Particle Effects.

### Scene View Editing

When creating and editing Particle Systems, you either work with the **Inspector** or the extended **Particle Editor**, and your changes are reflected in the **SceneView**. The scene view has a **Preview Panel**, where playback of the currently selected **Particle Effect** can be controlled in Edit Mode, with actions like **play**, **pause**, **stop** and **scrubbing playback time**

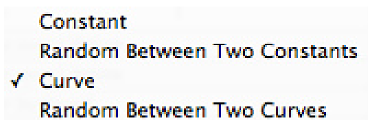



Scrubbing play back time can be performed by dragging on the label *Playback Time*. All Playback controls have shortcut keys which can be customized in the Preferences window

## Particle System Curve Editor

### MinMax curves

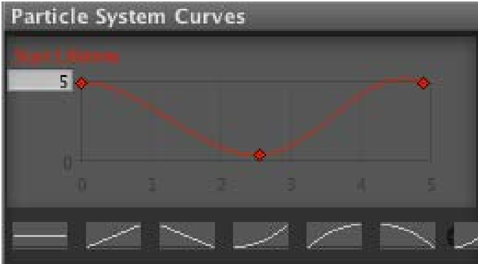
Many of the properties in the particle system modules describe a change of a value with time. That change is described via **MinMax Curves**. These time-animated properties (for example **size** and **speed**), will have a pull down menu on the right hand side, where you can choose between:




**Constant**: The value of the property will not change with time, and will not be displayed in the **Curve Editor**

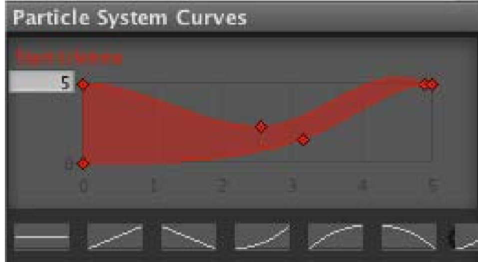
**Random between constants**: The value of the property will be selected at random between the two constants

**Curve**: The value of the property will change with time based on the curve specified in the **Curve Editor**




A property animated with a Curve

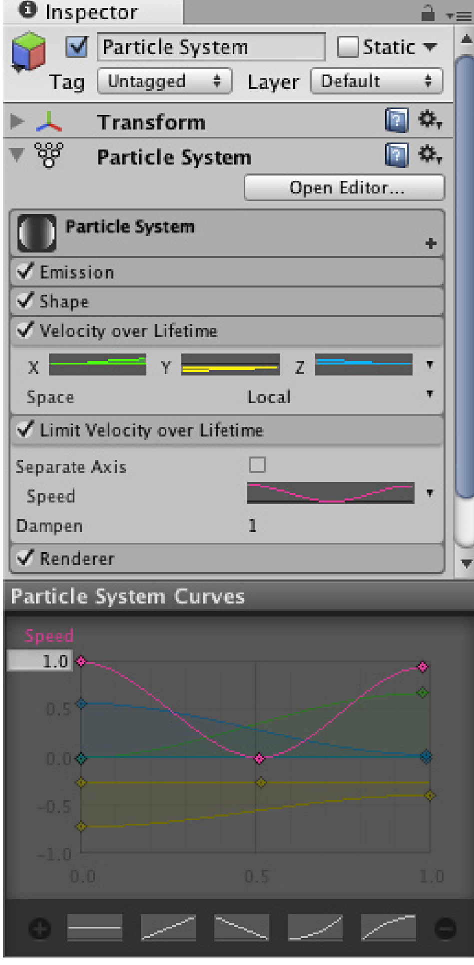
**Random between curves**: A curve will be generated at random between the min and the max curve, and the value of the property will change in time based on the generated curve




A property animated as **Random Between Two Curves**

In the **Curve Editor**, the *x*-axis spans time between 0 and the value specified by the **Duration** property, and the *y*-axis represents the value of the animated property at each point in time. The range of the *y*-axis can be adjusted in the number field

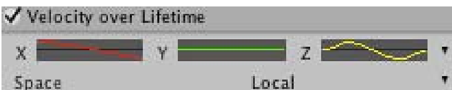
in the upper right corner of the **Curve Editor**. The **Curve Editor** currently displays all of the curves for a particle system in the same window.



Multiple curves in the same curve editor

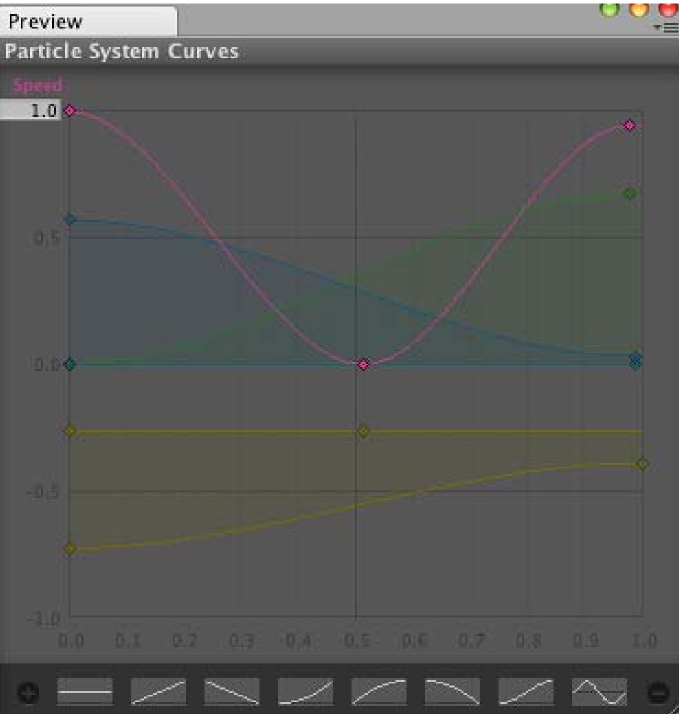
Note that the "-" in the bottom-right corner will remove the currently selected curve, while the "+" will *optimize* it (that is make it into a parametrized curve with at most 3 keys).

For animating properties that describe vectors in 3D space, we use the TripleMinMax Curves, which are simply curves for the x-,y-, and z- dimensions side by side, and it looks like this:




### Managing many curves in the curve editor

To avoid cluttering in the **Curve Editor**, it is possible to toggle curves on and off, by clicking on them in the inspector. The Particle System Curve Editor can also be detached from the Inspector by right-clicking on the **Particle System Curves** title bar, after which you should see something like this:



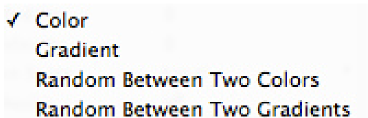
A detached Curve Editor that can be docked like any other window

For more information on working with curves, take a look at the Curve Editor documentation

## Colors and Gradients in the Particle System (Shuriken)

For properties that deal with color, the **Particle System** makes use of the **Color and Gradient Editor**. It works in a similar way to the Curve Editor.

The color-based properties will have a pull down menu on the right hand side, where you can choose between:




**Color**: The color will be the same throughout time (see Color Picker )

**Gradient**: The gradient (RGBA) will vary throughout time, edited in the Gradient Editor

**Random Between Two Colors**: The color varies with time and is chosen at random between two values specified in the Color Picker

**Random Between Two Gradients**: The gradient (RGBA) varies with time and is chosen at random between two values specified Gradient Editor

Page last updated: 2012-01-25

---

## class-Halo

**Halos** are light areas around light sources, used to give the impression of small dust particles in the air.

*A Light with a separate Halo* ***Component***

### Properties

Halos use the **Halo Texture** set up in the Render Settings. If none is assigned, it uses a default one. A Light component can be setup to automatically show halo, without a separate Halo component.

| | |
|---|---|
| **Color** | Color of the Halo. |
| **Size** | Size of the Halo. |

### Hints

- To see Halos in the scene view, check **Fx** button in the **Scene View** Toolbar.

Page last updated: 2011-04-12

---

## class-LensFlare

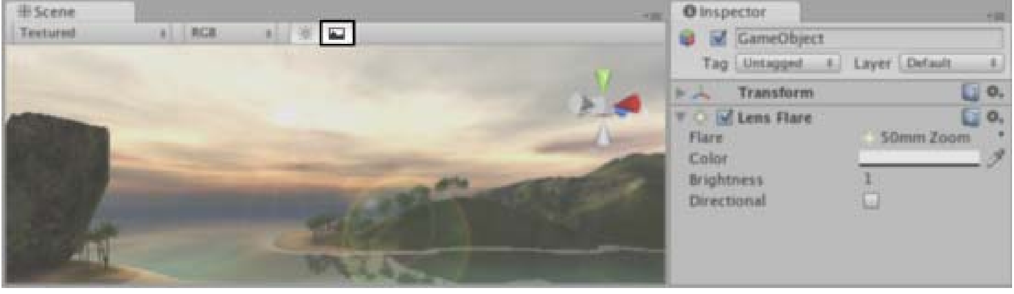
**Lens Flares** simulate the effect of lights refracting inside camera lens. They are used to represent really bright lights or, more subtly, just to add a bit more atmosphere to your scene.

*The Lens Flare* ***Inspector***

The easiest way to setup a Lens Flare is just to assign the Flare property of the Light. Unity contains a couple of pre-configured Flares in the Standard Assets package.

Otherwise, create an empty **GameObject** with **GameObject->Create Empty** from the menu bar and add the Lens Flare **Component** to it with **Component->Rendering->Lens Flare**. Then and choose the **Flare** in the Inspector.

To see the effect of Lens Flare in the **Scene View**, check the **Fx** button in the Scene View toolbar:




*Enable the* ***Fx*** *button to view Lens Flares in the Scene View*

### Properties

| | |
|---|---|
| **Flare** | The Flare to render. The flare defines all aspects of the lens flare's appearance. |
| **Color** | Some flares can be colorized to better fit in with your scene's mood. |
| **Brightness** | How large and bright the Lens Flare is. |
| **Directional** | If set, the flare will be oriented along positive Z axis of the game object. It will appear as if it was infinitely far away, and won't track object's position, only the direction of Z axis. |

### Details

You can directly set flares as a property of a Light Component, or set them up separately as Lens Flare component. If you attach them to a light, they will automatically track the position and direction of the light. To get more precise control, use this Component.

A Camera has to have a Flare Layer Component attached to make Flares visible (this is true by default, so you don't have to do any set-up).

### Hints

- Be discrete about your usage of Lens Flares.
- If you use a very bright Lens Flare, make sure its direction fits with your scene's primary light source.
- To design your own Flares, you need to create some Flare Assets. Start by duplicating some of the ones we provided in the the **Lens Flares** folder of the Standard Assets, then modify from that.
- Lens Flares are blocked by **Colliders**. A Collider in-between the Flare GameObject and the Camera will hide the Flare, even if the Collider does not have a **Mesh Renderer**.

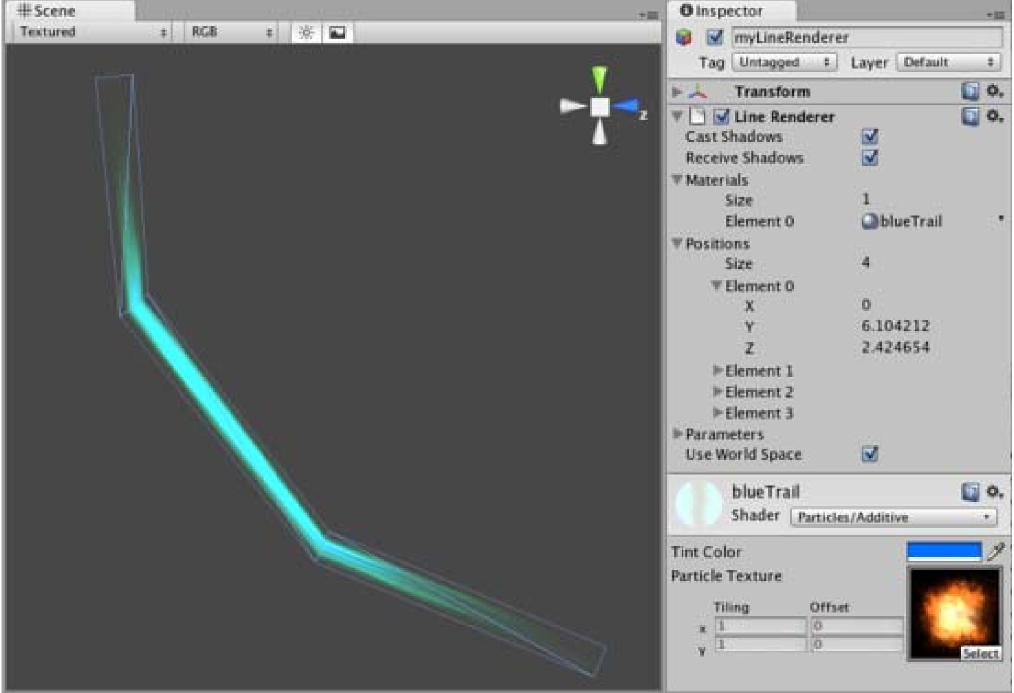
Page last updated: 2008-05-30

---

## class-LineRenderer

The **Line Renderer** takes an array of two or more points in 3D space and draws a straight line between each one. A single Line Renderer Component can thus be used to draw anything from a simple straight line, to a complex spiral. The line is always continuous; if you need to draw two or more completely separate lines, you should use multiple GameObjects, each with its own Line Renderer.

The Line Renderer does not render one pixel thin lines. It renders billboard lines that have width and can be textured. It uses the same algorithm for line rendering as the Trail Renderer.




*The Line Renderer **Inspector***

### Properties

| | |
|---|---|
| **Materials** | The first material from this list is used to render the lines. |
| **Positions** | Array of Vector3 points to connect. |
| **Size** | The number of segments in this line. |
| **Parameters** | List of parameters for each line: |
| **StartWidth** | Width at the first line position. |
| **EndWidth** | Width at the last line position. |
| **Start Color** | Color at the first line position. |
| **End Color** | Color at the last line position. |
| **Use World Space** | If enabled, the object's position is ignored, and the lines are rendered around world origin. |

### Details

To create a line renderer:

1. Choose **GameObject->Create Empty**
2. Choose **Component->Miscellaneous->Line Renderer**
3. Drag a texture or **Material** on the Line Renderer. It looks best if you use a particle shader in the Material.

### Hints

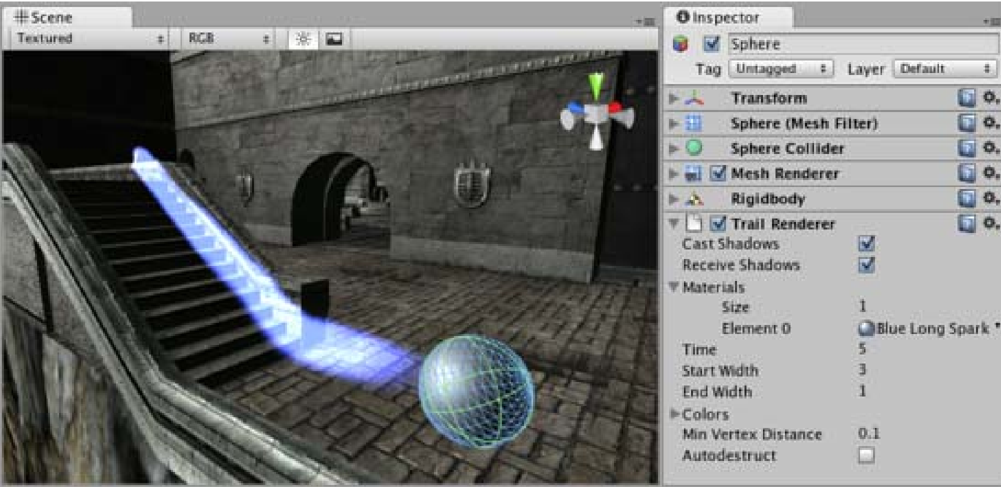
- Line Renderers are good to use for effects when you need to lay out all the vertices in one frame.
- The lines may seem to rotate as you move the **Camera**. This is intentional.
- The Line Renderer should be the only Renderer on a GameObject.

Page last updated: 2012-01-12

---

## class-TrailRenderer

The **Trail Renderer** is used to make trails behind objects in the scene as they move about.




*The Trail Renderer **Inspector***

### Properties

| | |
|---|---|
| **Materials** | An array of materials used for rendering the trail. Particle shaders work the best for trails. |
| **Size** | The total number of elements in the **Material** array. |
| **Element 0** | Reference to the **Material** used to render the trail. The total number of elements is determined by the **Size** property. |
| **Time** | Length of the trail, measured in seconds. |
| **Start Width** | Width of the trail at the object's position. |
| **End Width** | Width of the trail at the end. |
| **Colors** | Array of colors to use over the length of the trail. You can also set alpha transparency with the colors. |
| **Color0** to **Color4** | The trail's colors, initial to final. |
| **Min Vertex Distance** | The minimum distance between anchor points of the trail. |
| **AutoDestruct** | Enable this to make the object be destroyed when the object has been idle for **Time** seconds. |

### Details

The Trail Renderer is great for a trail behind a projectile, or contrails from the tip of a plane's wings. It is good when trying to add a general feeling of speed.

When using a Trail Renderer, no other renderers on the **GameObject** are used. It is best to create an empty GameObject, and attach a Trail Renderer as the only renderer. You can then parent the Trail Renderer to whatever object you would like it to follow.

**Materials**

Trail Renderers should use a material that has a Particle **Shader**. The **Texture** used for the Material should be of square dimensions (e.g. 256x256 or 512x512).

**Trail Width**

By setting the Trail's **Start** and **End Width**, along with the **Time** property, you can tune the way it is displayed and behaves. For example, you could create the wake behind a boat by setting the **Start Width** to 1, and the **End Width** to 2. These values would probably need to be fine-tuned for your game.

**Trail Colors**

You can cycle your trail through 5 different color/opacity combinations. Using colors could make a bright green plasma trail gradually dim down to a dull grey dissipation, or cycle through the other colors of the rainbow. If you don't want to change the color, it can be very effective to change only the opacity of each color to make your trail fade in and out at the head and/or tail.

**Min Vertex Distance**

The **Min Vertex Distance** value determines how far the object that contains the trail must travel before a segment of the trail is solidified. Low values like 0.1 will create trail segments more often, creating smoother trails. Higher values like 1.5 will create segments that are more jagged in appearance. There is a slight performance trade off when using lower values/smoother trails, so try to use the largest possible value to achieve the effect you are trying to create.

### Hints

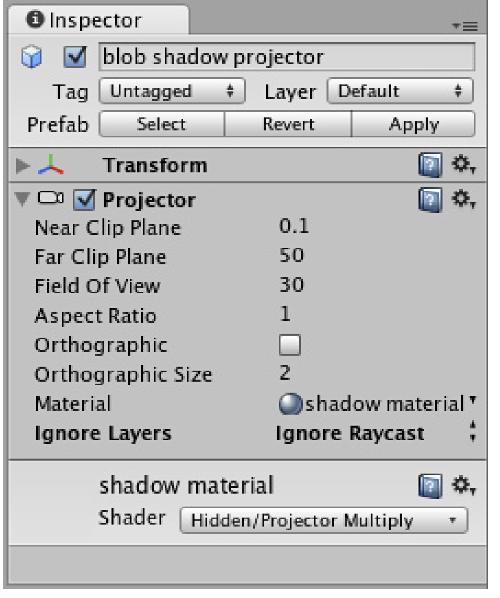
- Use Particle Materials with the Trail Renderer.
- Trail Renderers must be laid out over a sequence of frames, they can't appear instantaneously.
- Trail Renderers rotate to display the face toward the camera, similar to other **Particle Systems**.

Page last updated: 2011-12-28

---

## class-Projector

A **Projector** allows you to project a **Material** onto all objects that intersect its frustum. The material must use a special type of shader for the projection effect to work correctly - see the projector prefabs in Unity's standard assets for examples of how to use the supplied Projector/Light and Projector/Multiply shaders.



*The Projector **Inspector***

### Properties

| | |
|---|---|
| **Near Clip Plane** | Objects in front of the near clip plane will not be projected upon. |
| **Far Clip Plane** | Objects beyond this distance will not be affected. |
| **Field Of View** | The field of view in degrees. This is only used if the Projector is not Ortho Graphic. |
| **Aspect Ratio** | The Aspect Ratio of the Projector. This allows you to tune the height vs width of the Projector. |
| **Is Ortho Graphic** | If enabled, the Projector will be Ortho Graphic instead of perspective. |
| **Ortho Graphic Size** | The Ortho Graphic size of the Projection. this is only used if Is Ortho Graphic is turned on. |
| **Material** | The Material that will be Projected onto Objects. |
| **Ignore Layers** | Objects that are in one of the Ignore Layers will not be affected. By default, Ignore Layers is none so all geometry that intersects the Projector frustum will be affected. |

### Details

With a projector you can:

1. Create shadows.
2. Make a real world projector on a tripod with another Camera that films some other part of the world using a **Render Texture**.
3. Create bullet marks.
4. Funky lighting effects.

*A Projector is used to create a Blob Shadow for this Robot*

If you want to create a simple shadow effect, simply drag the **StandardAssets->Blob-Shadow->Blob shadow projector Prefab** into your scene. You can modify the Material to use a different Blob shadow texture.

**Note:** When creating a projector, always be sure to set the wrap mode of the texture's material of the projector to *clamp*. else the projector's texture will be seen repeated and you will not achieve the desired effect of shadow over your character.

### Hints

- Projector Blob shadows can create very impressive Splinter Cell-like lighting effects if used to shadow the environment properly.
- When no **Falloff** Texture is used in the projector's Material, it can project both forward and backward, creating "double projection". To fix this, use an alpha-only Falloff texture that has a black leftmost pixel column.

Page last updated: 2011-07-08

---

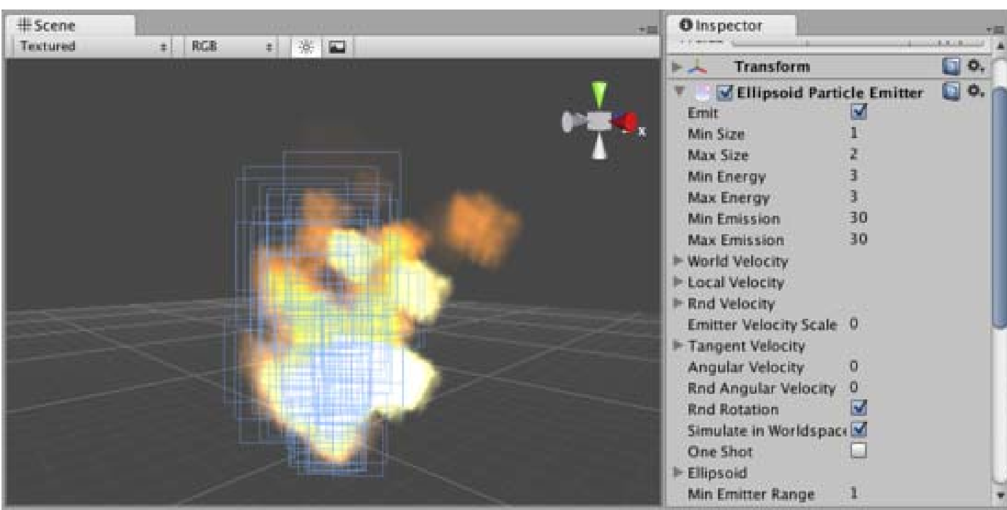
# comp-ParticlesLegacy

**Particles** are essentially 2D images rendered in 3D space. They are primarily used for effects such as smoke, fire, water droplets, or leaves. A **Particle System** is made up of three separate Components: **Particle Emitter**, **Particle Animator**, and a **Particle Renderer**. You can use a Particle Emitter and Renderer together if you want static particles. The Particle Animator will move particles in different directions and change colors. You also have access to each individual particle in a particle system via scripting, so you can create your own unique behaviors that way if you choose.

Please view the Particle Scripting Reference here.

# Ellipsoid Particle Emitter (Legacy)

The **Ellipsoid Particle Emitter** spawns particles inside a sphere. Use the **Ellipsoid** property below to scale & stretch the sphere.




*The Ellipsoid Particle Emitter* ***Inspector***

## Properties

| | |
|---|---|
| **Emit** | If enabled, the emitter will emit particles. |
| **Min Size** | The minimum size each particle can be at the time when it is spawned. |
| **Max Size** | The maximum size each particle can be at the time when it is spawned. |
| **Min Energy** | The minimum lifetime of each particle, measured in seconds. |
| **Max Energy** | The maximum lifetime of each particle, measured in seconds. |
| **Min Emission** | The minimum number of particles that will be spawned every second. |
| **Max Emission** | The maximum number of particles that will be spawned every second. |
| **World Velocity** | The starting speed of particles in world space, along X, Y, and Z. |
| **Local Velocity** | The starting speed of particles along X, Y, and Z, measured in the object's orientation. |
| **Rnd Velocity** | A random speed along X, Y, and Z that is added to the velocity. |
| **Emitter Velocity Scale** | The amount of the emitter's speed that the particles inherit. |
| **Tangent Velocity** | The starting speed of particles along X, Y, and Z, across the Emitter's surface. |
| **Angular Velocity** | The angular velocity of new particles in degrees per second. |
| **Rnd Angular Velocity** | A random angular velocity modifier for new particles. |
| **Rnd Rotation** | If enabled, the particles will be spawned with random rotations. |
| **Simulate In World Space** | If enabled, the particles don't move when the emitter moves. If false, when you move the emitter, the particles follow it around. |
| **One Shot** | If enabled, the particle numbers specified by min & max emission is spawned all at once. If disabled, the particles are generated in a long stream. |
| **Ellipsoid** | Scale of the sphere along X, Y, and Z that the particles are spawned inside. |
| **MinEmitterRange** | Determines an empty area in the center of the sphere - use this to make particles appear on the edge of the sphere. |

## Details

Ellipsoid Particle Emitters (EPEs) are the basic emitter, and are included when you choose to add a **Particle System** to your scene from **Components->Particles->Particle System**. You can define the boundaries for the particles to be spawned, and give the particles an initial velocity. From here, use the Particle Animator to manipulate how your particles will change over time to achieve interesting effects.

Particle Emitters work in conjunction with Particle Animators and Particle Renderers to create, manipulate, and display Particle Systems. All three Components must be present on an object before the particles will behave correctly. When particles are being emitted, all different velocities are added together to create the final velocity.

**Spawning Properties**

Spawning properties like **Size**, **Energy**, **Emission**, and **Velocity** will give your particle system distinct personality when trying to achieve different effects. Having a small **Size** could simulate fireflies or stars in the sky. A large **Size** could simulate dust clouds in a musky old building.

**Energy** and **Emission** will control how long your particles remain onscreen and how many particles can appear at any one time. For example, a rocket might have high **Emission** to simulate density of smoke, and high **Energy** to simulate the slow dispersion of smoke into the air.

**Velocity** will control how your particles move. You might want to change your **Velocity** in scripting to achieve interesting effects, or if you want to simulate a constant effect like wind, set your X and Z **Velocity** to make your particles blow away.

**Simulate in World Space**

If this is disabled, the position of each individual particle will always translate relative to the **Position** of the emitter. When the emitter moves, the particles will move along with it. If you have **Simulate in World Space** enabled, particles will not be affected by the translation of the emitter. For example, if you have a fireball that is spurting flames that rise, the flames will be spawned and float up in space as the fireball gets further away. If **Simulate in World Space** is disabled, those same flames will move across the screen along with the fireball.

**Emitter Velocity Scale**

This property will only apply if **Simulate in World Space** is enabled.

If this property is set to 1, the particles will inherit the exact translation of the emitter at the time they are spawned. If it is set to 2, the particles will inherit double the emitter's translation when they are spawned. 3 is triple the translation, etc.

**One Shot**

**One Shot** emitters will create all particles within the **Emission** property all at once, and cease to emit particles over time. Here are some examples of different particle system uses with **One Shot Enabled** or **Disabled**:

**Enabled**:

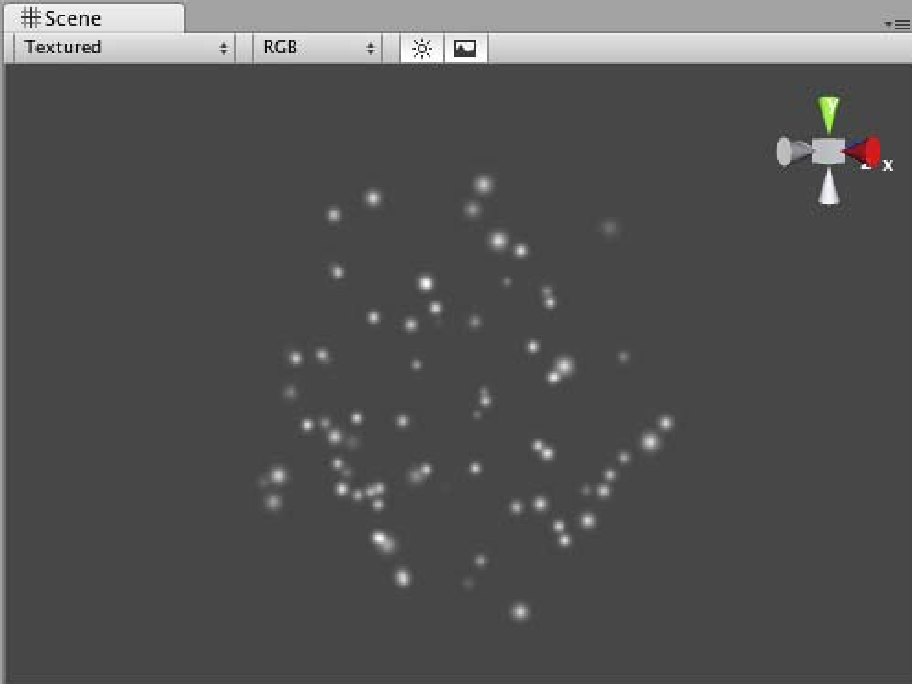
- Explosion
- Water splash
- Magic spell

**Disabled**:

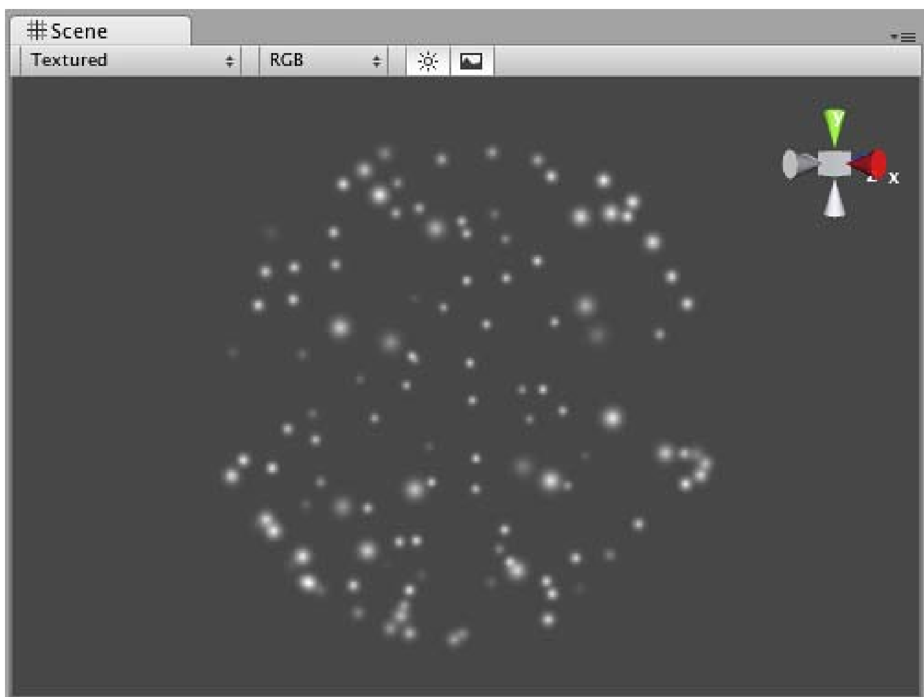
- Gun barrel smoke
- Wind effect
- Waterfall

**Min Emitter Range**

The **Min Emitter Range** determines the depth within the ellipsoid that particles can be spawned. Setting it to 0 will allow particles to spawn anywhere from the center core of the ellipsoid to the outer-most range. Setting it to 1 will restrict spawn locations to the outer-most range of the ellipsoid.




***Min Emitter Range*** *of 0*



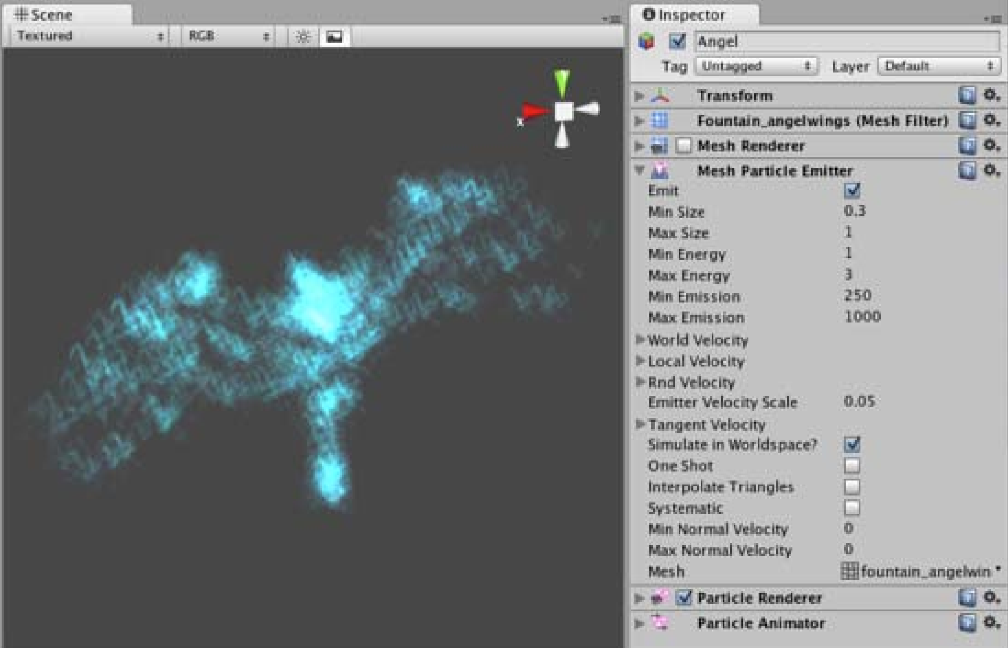

***Min Emitter Range*** *of 1*

### Hints

- Be careful of using many large particles. This can seriously hinder performance on low-level machines. Always try to use the minimum number of particles to attain an effect.
- The **Emit** property works in conjunction with the **AutoDestruct** property of the Particle Animator. Through scripting, you can cease the emitter from emitting, and then **AutoDestruct** will automatically destroy the Particle System and the GameObject it is attached to.

## Mesh Particle Emitter (Legacy)

The **Mesh Particle Emitter** emits particles around a mesh. Particles are spawned from the surface of the mesh, which can be necessary when you want to make your particles interact in a complex way with objects.




*The Mesh Particle Emitter* ***Inspector***

## Properties

| | |
|---|---|
| **Emit** | If enabled, the emitter will emit particles. |
| **Min Size** | The minimum size each particle can be at the time when it is spawned. |
| **Max Size** | The maximum size each particle can be at the time when it is spawned. |
| **Min Energy** | The minimum lifetime of each particle, measured in seconds. |
| **Max Energy** | The maximum lifetime of each particle, measured in seconds. |
| **Min Emission** | The minimum number of particles that will be spawned every second. |
| **Max Emission** | The maximum number of particles that will be spawned every second. |
| **World Velocity** | The starting speed of particles in world space, along X, Y, and Z. |
| **Local Velocity** | The starting speed of particles along X, Y, and Z, measured in the object's orientation. |
| **Rnd Velocity** | A random speed along X, Y, and Z that is added to the velocity. |
| **Emitter Velocity Scale** | The amount of the emitter's speed that the particles inherit. |
| **Tangent Velocity** | The starting speed of particles along X, Y, and Z, across the Emitter's surface. |
| **Angular Velocity** | The angular velocity of new particles in degrees per second. |
| **Rnd Angular Velocity** | A random angular velocity modifier for new particles. |
| **Rnd Rotation** | If enabled, the particles will be spawned with random rotations. |
| **Simulate In World Space** | If enabled, the particles don't move when the emitter moves. If false, when you move the emitter, the particles follow it around. |
| **One Shot** | If enabled, the particle numbers specified by min & max emission is spawned all at once. If disabled, the particles are generated in a long stream. |
| **Interpolate Triangles** | If enabled, particles are spawned all over the mesh's surface. If disabled, particles are only spawned from the mesh's vertrices. |
| **Systematic** | If enabled, particles are spawned in the order of the vertices defined in the mesh. Although you seldom have direct control over vertex order in meshes, most 3D modelling applications have a very systematic setup when using primitives. It is important that the mesh contains no faces in order for this to work. |
| **Min Normal Velocity** | Minimum amount that particles are thrown away from the mesh. |
| **Max Normal Velocity** | Maximum amount that particles are thrown away from the mesh. |

## Details

Mesh Particle Emitters (MPEs) are used when you want more precise control over the spawn position & directions than the simpler **Ellipsoid Particle Emitter** gives you. They can be used for making advanced effects.

MPEs work by emitting particles at the vertices of the attached mesh. Therefore, the areas of your mesh that are more dense with polygons will be more dense with particle emission.

Particle Emitters work in conjunction with Particle Animators and Particle Renderers to create, manipulate, and display Particle Systems. All three Components must be present on an object before the particles will behave correctly. When particles are

being emitted, all different velocities are added together to create the final velocity.

**Spawning Properties**

Spawning properties like **Size**, **Energy**, **Emission**, and **Velocity** will give your particle system distinct personality when trying to achieve different effects. Having a small **Size** could simulate fireflies or stars in the sky. A large **Size** could simulate dust clouds in a musky old building.

**Energy** and **Emission** will control how long your particles remain onscreen and how many particles can appear at any one time. For example, a rocket might have high **Emission** to simulate density of smoke, and high **Energy** to simulate the slow dispersion of smoke into the air.

**Velocity** will control how your particles move. You might want to change your **Velocity** in scripting to achieve interesting effects, or if you want to simulate a constant effect like wind, set your X and Z **Velocity** to make your particles blow away.

**Simulate in World Space**

If this is disabled, the position of each individual particle will always translate relative to the **Position** of the emitter. When the emitter moves, the particles will move along with it. If you have **Simulate in World Space** enabled, particles will not be affected by the translation of the emitter. For example, if you have a fireball that is spurting flames that rise, the flames will be spawned and float up in space as the fireball gets further away. If **Simulate in World Space** is disabled, those same flames will move across the screen along with the fireball.

**Emitter Velocity Scale**

This property will only apply if **Simulate in World Space** is enabled.

If this property is set to 1, the particles will inherit the exact translation of the emitter at the time they are spawned. If it is set to 2, the particles will inherit double the emitter's translation when they are spawned. 3 is triple the translation, etc.

**One Shot**

**One Shot** emitters will create all particles within the **Emission** property all at once, and cease to emit particles over time. Here are some examples of different particle system uses with **One Shot Enabled** or **Disabled**:

**Enabled**:

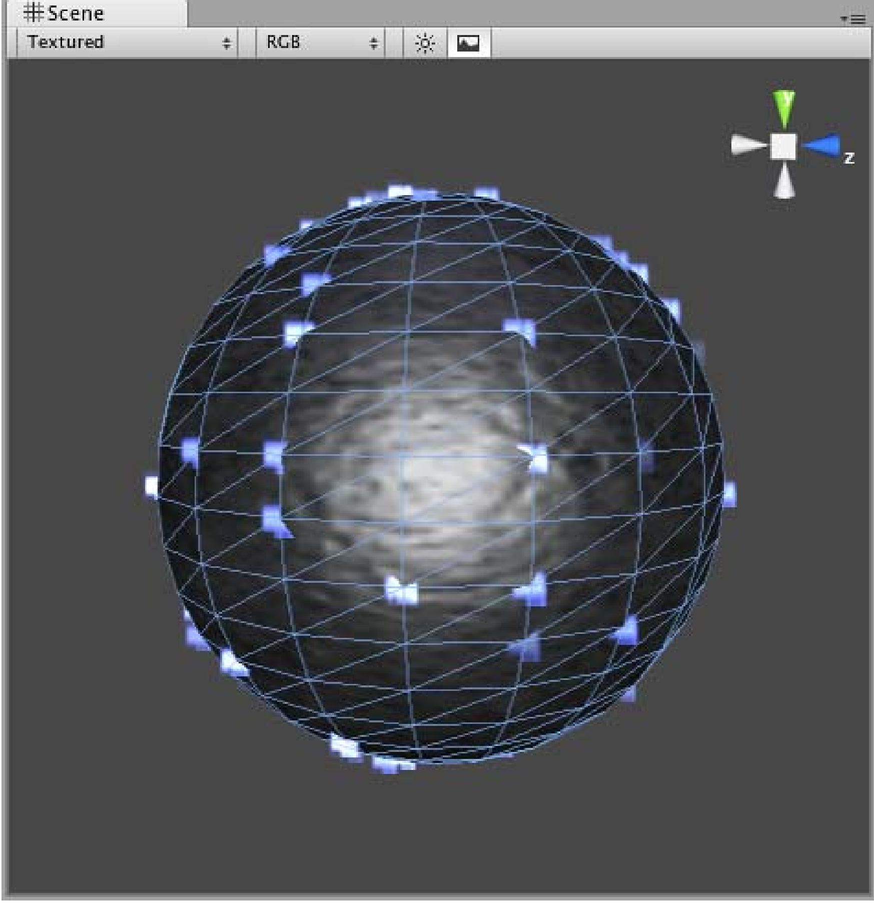
- Explosion
- Water splash
- Magic spell

**Disabled**:

- Gun barrel smoke
- Wind effect
- Waterfall

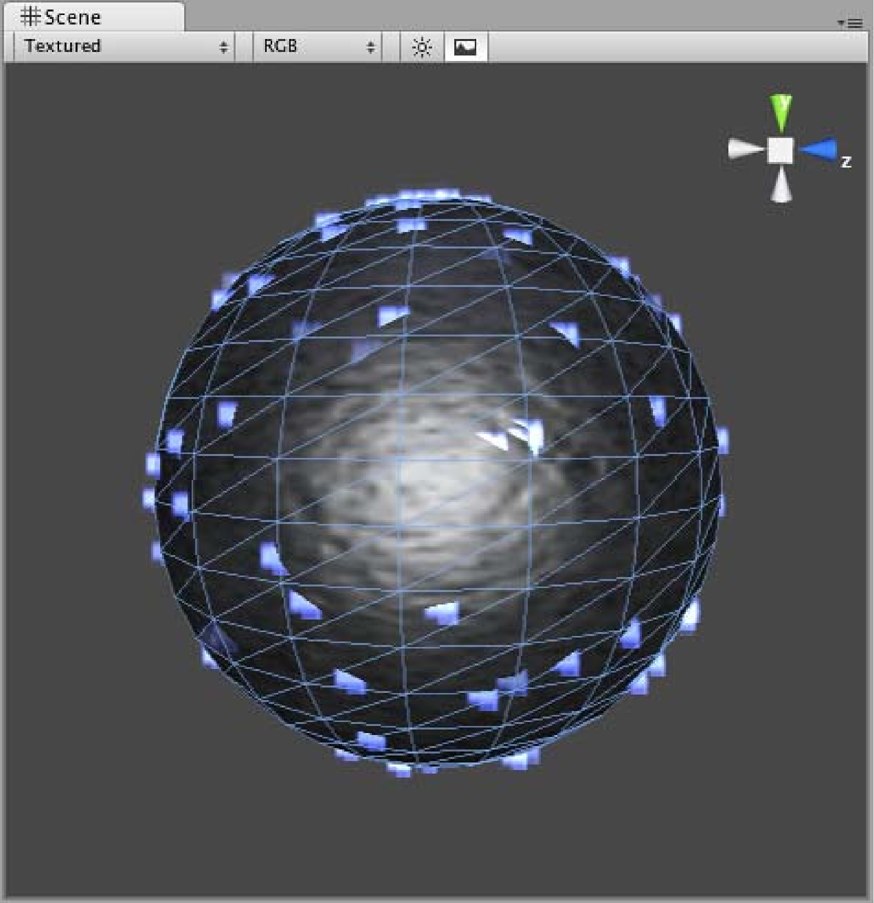
### Interpolate Triangles

Enabling your emitter to **Interpolate Triangles** will allow particles to be spawned between the mesh's vertices. This option is off by default, so particles will only be spawned at the vertices.




*A sphere with **Interpolate Triangles** off (the default)*

Enabling this option will spawn particles on and in-between vertices, essentially all over the mesh's surface (seen below).

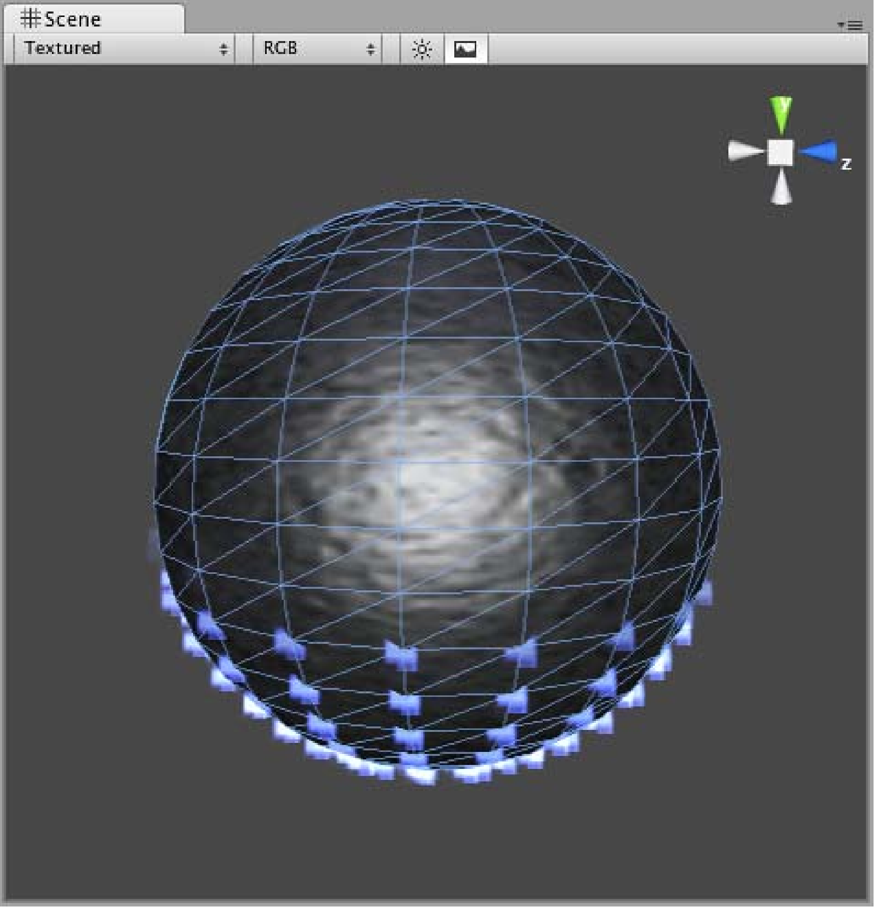



*A sphere with **Interpolate Triangles** on*

It bears repeating that even with **Interpolate Triangles** enabled, particles will still be denser in areas of your mesh that are more dense with polygons.

### Systematic

Enabling **Systematic** will cause your particles to be spawned in your mesh's vertex order. The vertex order is set by your 3D modeling application.



*An MPE attached to a sphere with **Systematic** enabled*

### Normal Velocity

**Normal Velocity** controls the speed at which particles are emitted along the normal from where they are spawned.

For example, create a Mesh Particle System, use a cube mesh as the emitter, enable **Interpolate Triangles**, and set **Normal Velocity Min** and **Max** to 1. You will now see the particles emit from the faces of the cube in a straight line.

### See Also

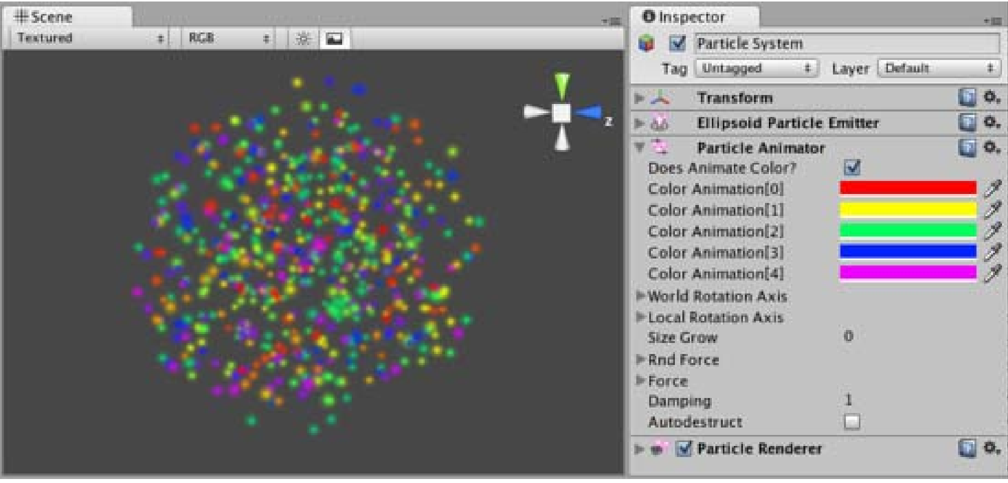
- How to make a Mesh Particle Emitter

### Hints

- Be careful of using many large particles. This can seriously hinder performance on low-level machines. Always try to use the minimum number of particles to attain an effect.
- The **Emit** property works in conjunction with the **AutoDestruct** property of the Particle Animator. Through scripting, you can cease the emitter from emitting, and then **AutoDestruct** will automatically destroy the Particle System and the GameObject it is attached to.
- MPEs can also be used to make glow from a lot of lamps placed in a scene. Simply make a mesh with one vertex in the center of each lamp, and build an MPE from that with a halo material. Great for evil sci-fi worlds.

## Particle Animator (Legacy)

**Particle Animators** move your particles over time, you use them to apply wind, drag & color cycling to your particle systems.




*The Particle Animator **Inspector***

## Properties

| | |
|---|---|
| **Does Animate Color** | If enabled, particles cycle their color over their lifetime. |
| **Color Animation** | The 5 colors particles go through. All particles cycle over this - if some have a shorter life span than others, they will animate faster. |
| **World Rotation Axis** | An optional world-space axis the particles rotate around. Use this to make advanced spell effects or give caustic bubbles some life. |
| **Local Rotation Axis** | An optional local-space axis the particles rotate around. Use this to make advanced spell effects or give caustic bubbles some life. |
| **Size Grow** | Use this to make particles grow in size over their lifetime. As randomized forces will spread your particles out, it is often nice to make them grow in size so they don't fall apart. Use this to make smoke rise upwards, to simulate wind, etc. |
| **Rnd Force** | A random force added to particles every frame. Use this to make smoke become more alive. |
| **Force** | The force being applied every frame to the particles, measure relative to the world. |
| **Damping** | How much particles are slowed every frame. A value of 1 gives no damping, while less makes them slow down. |
| **Autodestruct** | If enabled, the GameObject attached to the Particle Animator will be destroyed when all particles disappear. |

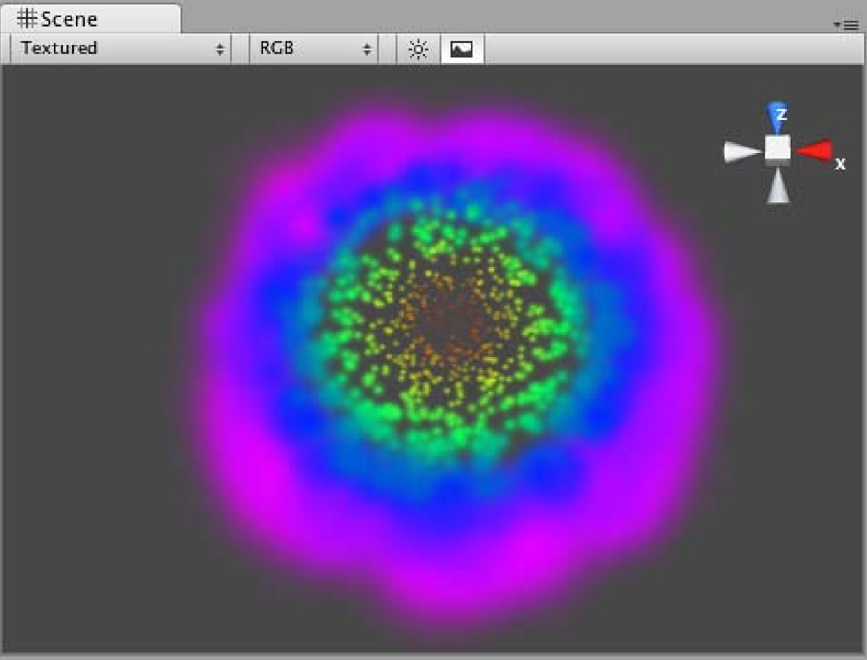
## Details

Particle Animators allow your particle systems to be dynamic. They allow you to change the color of your particles, apply forces and rotation, and choose to destroy them when they are finished emitting. For more information about Particle Systems, reference Mesh Particle Emitters, Ellipsoid Particle Emitters, and Particle Renderers.

### Animating Color

If you would like your particles to change colors or fade in/out, enable them to **Animate Color** and specify the colors for the cycle. Any particle system that animates color will cycle through the 5 colors you choose. The speed at which they cycle will be determined by the Emitter's **Energy** value.

If you want your particles to fade in rather than instantly appear, set your first or last color to have a low Alpha value.




*An **Animating Color** Particle System*

**Rotation Axes**
Setting values in either the Local or World **Rotation Axes** will cause all spawned particles to rotate around the indicated axis (with the **Transform's** position as the center). The greater the value is entered on one of these axes, the faster the rotation will be.

Setting values in the Local Axes will cause the rotating particles to adjust their rotation as the Transform's rotation changes, to match its local axes.

Setting values in the World Axes will cause the particles' rotation to be consistent, regardless of the Transform's rotation.

**Forces & Damping**
You use force to make particles accelerate in the direction specified by the force.

**Damping** can be used to decelerate or accelerate without changing their direction:

- A value of 1 means no **Damping** is applied, the particles will not slow down or accelerate.
- A value of 0 means particles will stop immediately.
- A value of 2 means particles will double their speed every second.

**Destroying GameObjects attached to Particles**
You can destroy the Particle System and any attached **GameObject** by enabling the **AutoDestruct** property. For example, if you have an oil drum, you can attach a Particle System that has **Emit** disabled and **AutoDestruct** enabled. On collision, you enable the Particle Emitter. The explosion will occur and after it is over, the Particle System and the oil drum will be destroyed and removed from the scene.

Note that automatic destruction takes effect only after some particles have been emitted. The precise rules for when the object is destroyed when **AutoDestruct** is on:

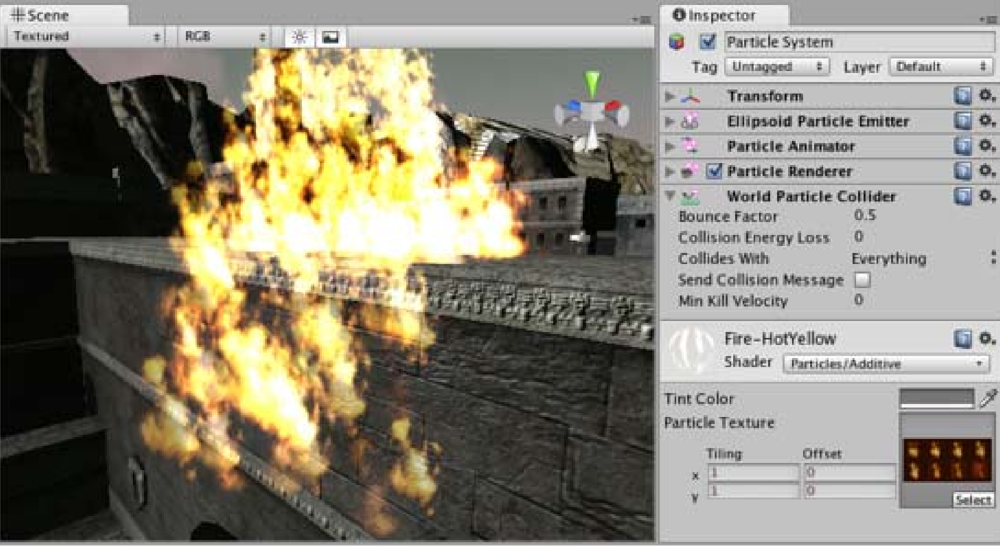
- If there have been some particles emitted already, but all of them are dead now, *or*
- If the emitter did have **Emit** on at some point, but now **Emit** is off.

## Hints

- Use the **Color Animation** to make your particles fade in & out over their lifetime - otherwise, you will get nasty-looking pops.
- Use the **Rotation Axes** to make whirlpool-like swirly motions.

# World Particle Collider (Legacy)

The **World Particle Collider** is used to collide particles against other **Colliders** in the scene.




*A **Particle System** colliding with a **Mesh Collider***

### Properties

| | |
|---|---|
| **Bounce Factor** | Particles can be accelerated or slowed down when they collide against other objects. This factor is similar to the **Particle Animator's Damping** property. |
| **Collision Energy Loss** | Amount of energy (in seconds) a particle should lose when colliding. If the energy goes below 0, the particle is killed. |
| **Min Kill Velocity** | If a particle's **Velocity** drops below **Min Kill Velocity** because of a collision, it will be eliminated. |
| **Collides with** | Which Layers the particle will collide against. |
| **Send Collision Message** | If enabled, every particle sends out a collision message that you can catch through scripting. |

### Details

To create a Particle System with Particle Collider:

1. Create a Particle System using **GameObject->Create Other->Particle System**
2. Add the Particle Collider using **Component->Particles->World Particle Collider**

#### Messaging

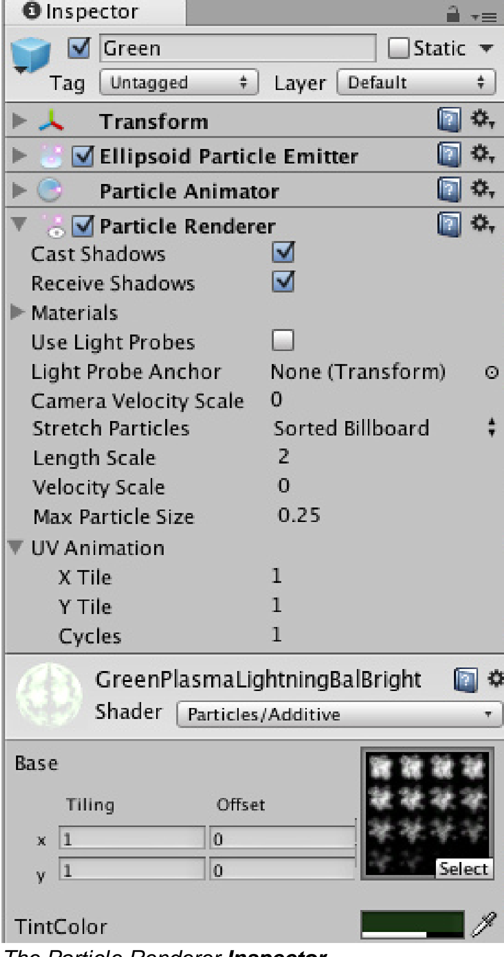
If **Send Collision Message** is enabled, any particles that are in a collision will send the message **OnParticleCollision()** to both the particle's **GameObject** and the GameObject the particle collided with.

### Hints

- **Send Collision Message** can be used to simulate bullets and apply damage on impact.
- Particle Collision Detection is slow when used with a lot of particles. Use Particle Collision Detection wisely.
- Message sending introduces a large overhead and shouldn't be used for normal Particle Systems.

## Particle Renderer (Legacy)

The **Particle Renderer** renders the **Particle System** on screen.



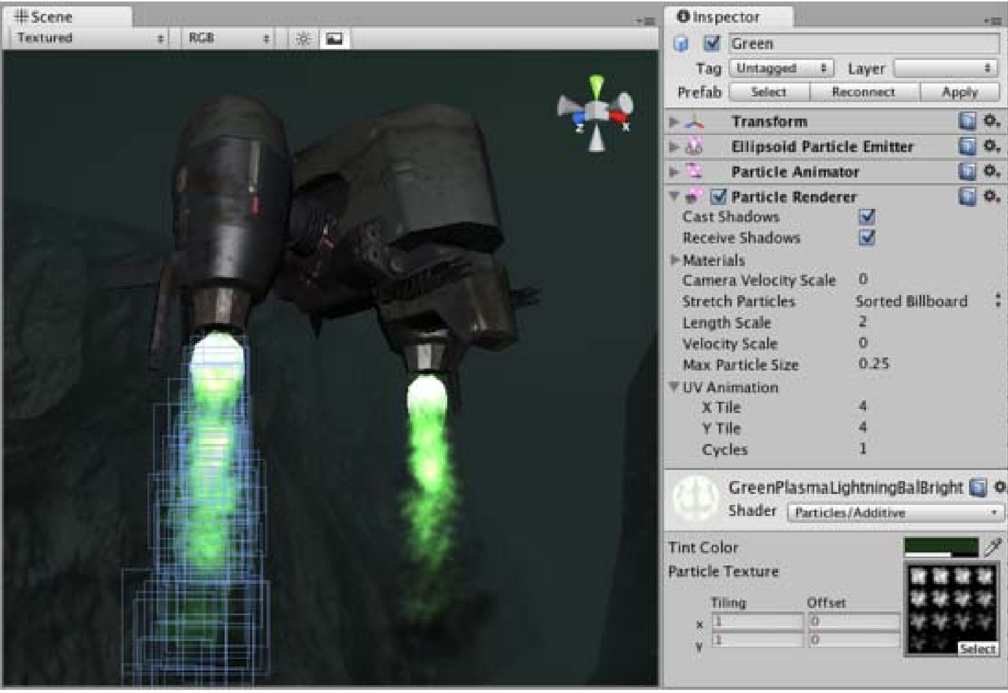

*The Particle Renderer* ***Inspector***

## Properties

| | |
|---|---|
| **Cast Shadows** | If enabled, allows object to cast shadows. |
| **Receive Shadows** | If enabled, allows object to receive shadows. |
| **Materials** | Reference to a list of **Materials** that will be displayed in the position of each individual particle. |
| **Use Light Probes** | If enabled and baked light probes are present in the scene, an interpolated light probe. |
| **Light Probe Anchor** | If set, Renderer will use this Transform's position to find the interpolated light probe. |
| **Camera Velocity Scale** | The amount of stretching that is applied to the Particles based on Camera movement. |
| **Stretch Particles** | Determines how the particles are rendered. |
| **Billboard** | The particles are rendered as if facing the camera. |
| **Stretched** | The particles are facing the direction they are moving. |
| **SortedBillboard** | The particles are sorted by depth. Use this when using a blending material. |
| **VerticalBillboard** | All particles are aligned flat along the X/Z axes. |
| **HorizontalBillboard** | All particles are aligned flat along the X/Y axes. |
| **Length Scale** | If **Stretch Particles** is set to **Stretched**, this value determines how long the particles are in their direction of motion. |
| **Velocity Scale** | If **Stretch Particles** is set to **Stretched**, this value determines the rate at which particles will be stretched, based on their movement speed. |
| **UV Animation** | If either of these are set, the UV coordinates of the particles will be generated for use with a tile animated texture. See the section on Animated Textures below. |
| **X Tile** | Number of frames located across the X axis. |
| **Y Tile** | Number of frames located across the Y axis. |
| **Cycles** | How many times to loop the animation sequence. |

## Details

Particle Renderers are required for any Particle Systems to be displayed on the screen.



*A Particle Renderer makes the Gunship's engine exhaust appear on the screen*

### Choosing a Material

When setting up a Particle Renderer it is very important to use an appropriate material and shader that renders both sides of the material. Most of the time you want to use a Material with one of the built-in Particle Shaders. There are some premade materials in the **Standard Assets->Particles->Sources** folder that you can use.

Creating a new material is easy:

1. Select **Assets->Create Other->Material** from the menu bar.
2. The Material has a shader popup, choose one of the shaders in the Particles group. Eg. **Particles->Multiply**.
3. Now assign a Texture. The different shaders use the alpha channel of the textures slightly differently, but most of the time a value of black will make it invisible and white in the alpha channel will display it on screen.

### Distorting particles

By default particles are rendered billboarded. That is simple square sprites. This is good for smoke and explosions and most other particle effects.

Particles can be made to either stretch with the velocity. This is useful for sparks, lightning or laser beams. **Length Scale** and **Velocity Scale** affects how long the stretched particle will be.

**Sorted Billboard** can be used to make all particles sort by depth. Sometimes this is necessary, mostly when using **Alpha Blended** particle shaders. This can be expensive and should only be used if it really makes a quality difference when rendering.

### Animated textures

Particle Systems can be rendered with an animated tile texture. To use this feature, make the texture out of a grid of images. As the particles go through their life cycle, they will cycle through the images. This is good for adding more life to your particles, or making small rotating debris pieces.

## Hints

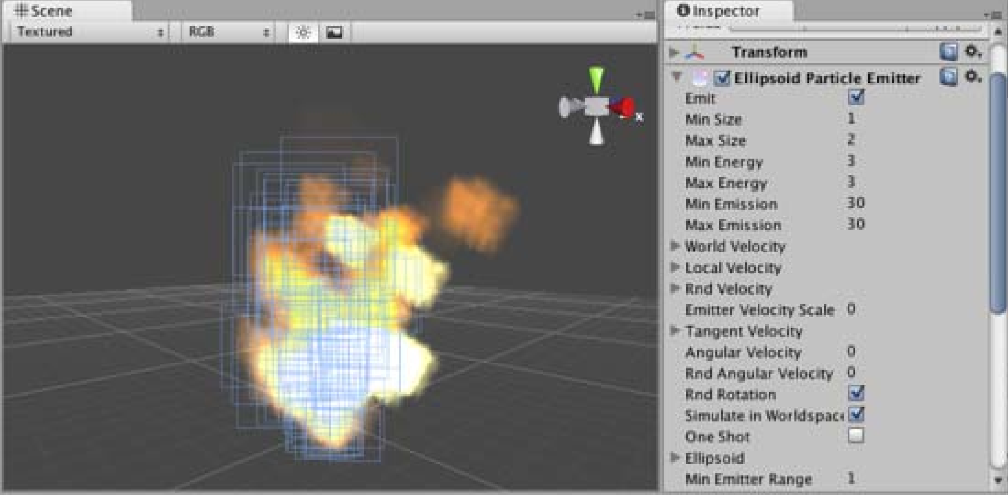
- Use Particle Shaders with the Particle Renderer.

Page last updated: 2012-02-03

---

# class-EllipsoidParticleEmitter

The **Ellipsoid Particle Emitter** spawns particles inside a sphere. Use the **Ellipsoid** property below to scale & stretch the sphere.



*The Ellipsoid Particle Emitter **Inspector***

## Properties

| | |
|---|---|
| **Emit** | If enabled, the emitter will emit particles. |
| **Min Size** | The minimum size each particle can be at the time when it is spawned. |
| **Max Size** | The maximum size each particle can be at the time when it is spawned. |
| **Min Energy** | The minimum lifetime of each particle, measured in seconds. |
| **Max Energy** | The maximum lifetime of each particle, measured in seconds. |
| **Min Emission** | The minimum number of particles that will be spawned every second. |
| **Max Emission** | The maximum number of particles that will be spawned every second. |
| **World Velocity** | The starting speed of particles in world space, along X, Y, and Z. |
| **Local Velocity** | The starting speed of particles along X, Y, and Z, measured in the object's orientation. |
| **Rnd Velocity** | A random speed along X, Y, and Z that is added to the velocity. |
| **Emitter Velocity Scale** | The amount of the emitter's speed that the particles inherit. |
| **Tangent Velocity** | The starting speed of particles along X, Y, and Z, across the Emitter's surface. |
| **Angular Velocity** | The angular velocity of new particles in degrees per second. |
| **Rnd Angular Velocity** | A random angular velocity modifier for new particles. |
| **Rnd Rotation** | If enabled, the particles will be spawned with random rotations. |
| **Simulate In World Space** | If enabled, the particles don't move when the emitter moves. If false, when you move the emitter, the particles follow it around. |
| **One Shot** | If enabled, the particle numbers specified by min & max emission is spawned all at once. If disabled, the particles are generated in a long stream. |
| **Ellipsoid** | Scale of the sphere along X, Y, and Z that the particles are spawned inside. |
| **MinEmitterRange** | Determines an empty area in the center of the sphere - use this to make particles appear on the edge of the sphere. |

## Details

Ellipsoid Particle Emitters (EPEs) are the basic emitter, and are included when you choose to add a **Particle System** to your scene from **Components->Particles->Particle System**. You can define the boundaries for the particles to be spawned, and give the particles an initial velocity. From here, use the Particle Animator to manipulate how your particles will change over time to achieve interesting effects.

Particle Emitters work in conjunction with Particle Animators and Particle Renderers to create, manipulate, and display Particle Systems. All three Components must be present on an object before the particles will behave correctly. When particles are being emitted, all different velocities are added together to create the final velocity.

### Spawning Properties

Spawning properties like **Size**, **Energy**, **Emission**, and **Velocity** will give your particle system distinct personality when trying to achieve different effects. Having a small **Size** could simulate fireflies or stars in the sky. A large **Size** could simulate dust clouds in a musky old building.

**Energy** and **Emission** will control how long your particles remain onscreen and how many particles can appear at any one

time. For example, a rocket might have high **Emission** to simulate density of smoke, and high **Energy** to simulate the slow dispersion of smoke into the air.

**Velocity** will control how your particles move. You might want to change your **Velocity** in scripting to achieve interesting effects, or if you want to simulate a constant effect like wind, set your X and Z **Velocity** to make your particles blow away.

**Simulate in World Space**
If this is disabled, the position of each individual particle will always translate relative to the **Position** of the emitter. When the emitter moves, the particles will move along with it. If you have **Simulate in World Space** enabled, particles will not be affected by the translation of the emitter. For example, if you have a fireball that is spurting flames that rise, the flames will be spawned and float up in space as the fireball gets further away. If **Simulate in World Space** is disabled, those same flames will move across the screen along with the fireball.

**Emitter Velocity Scale**
This property will only apply if **Simulate in World Space** is enabled.

If this property is set to 1, the particles will inherit the exact translation of the emitter at the time they are spawned. If it is set to 2, the particles will inherit double the emitter's translation when they are spawned. 3 is triple the translation, etc.

**One Shot**
**One Shot** emitters will create all particles within the **Emission** property all at once, and cease to emit particles over time. Here are some examples of different particle system uses with **One Shot Enabled** or **Disabled**:

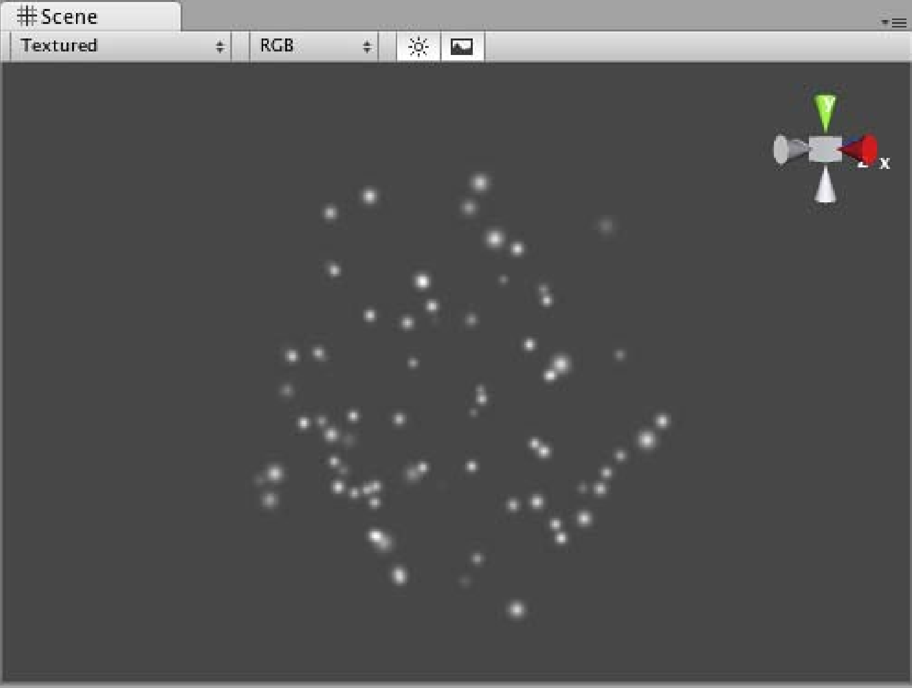
**Enabled**:

- Explosion
- Water splash
- Magic spell

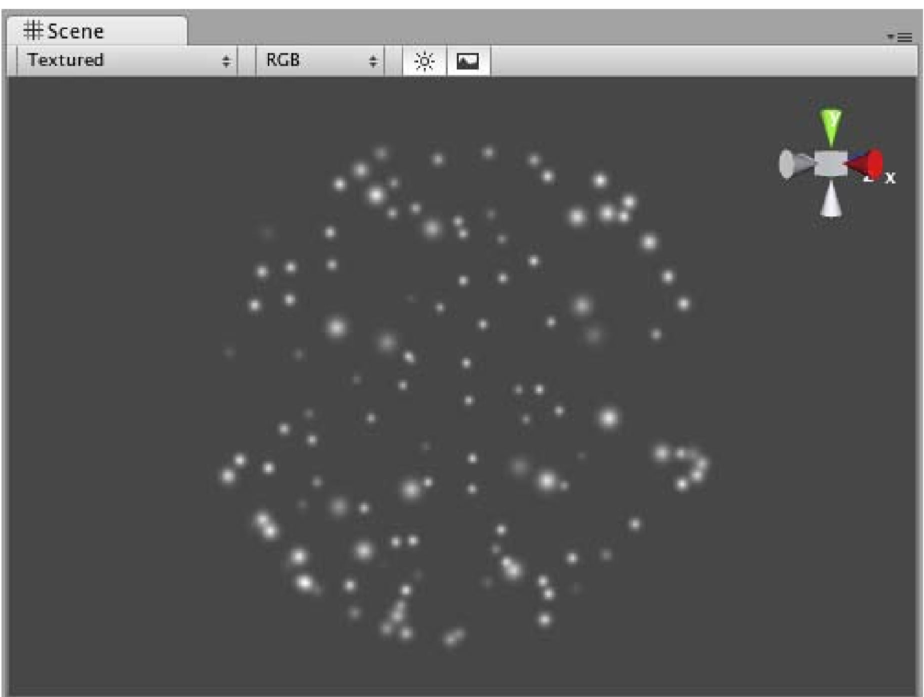
**Disabled**:

- Gun barrel smoke
- Wind effect
- Waterfall

**Min Emitter Range**
The **Min Emitter Range** determines the depth within the ellipsoid that particles can be spawned. Setting it to 0 will allow particles to spawn anywhere from the center core of the ellipsoid to the outer-most range. Setting it to 1 will restrict spawn locations to the outer-most range of the ellipsoid.




***Min Emitter Range*** *of 0*




***Min Emitter Range*** *of 1*

### Hints

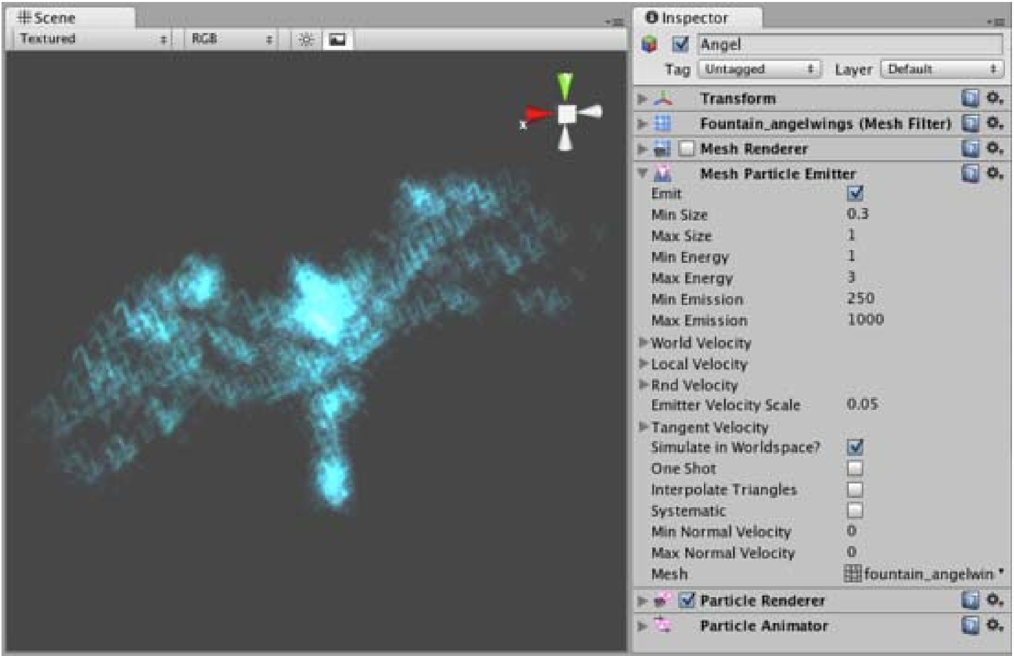
- Be careful of using many large particles. This can seriously hinder performance on low-level machines. Always try to use the minimum number of particles to attain an effect.
- The **Emit** property works in conjunction with the **AutoDestruct** property of the Particle Animator. Through scripting, you can cease the emitter from emitting, and then **AutoDestruct** will automatically destroy the Particle System and the GameObject it is attached to.

Page last updated: 2011-12-01

---

## class-MeshParticleEmitter

The **Mesh Particle Emitter** emits particles around a mesh. Particles are spawned from the surface of the mesh, which can be necessary when you want to make your particles interact in a complex way with objects.



*The Mesh Particle Emitter **Inspector***

## Properties

| | |
|---|---|
| **Emit** | If enabled, the emitter will emit particles. |
| **Min Size** | The minimum size each particle can be at the time when it is spawned. |
| **Max Size** | The maximum size each particle can be at the time when it is spawned. |
| **Min Energy** | The minimum lifetime of each particle, measured in seconds. |
| **Max Energy** | The maximum lifetime of each particle, measured in seconds. |
| **Min Emission** | The minimum number of particles that will be spawned every second. |
| **Max Emission** | The maximum number of particles that will be spawned every second. |
| **World Velocity** | The starting speed of particles in world space, along X, Y, and Z. |
| **Local Velocity** | The starting speed of particles along X, Y, and Z, measured in the object's orientation. |
| **Rnd Velocity** | A random speed along X, Y, and Z that is added to the velocity. |
| **Emitter Velocity Scale** | The amount of the emitter's speed that the particles inherit. |
| **Tangent Velocity** | The starting speed of particles along X, Y, and Z, across the Emitter's surface. |
| **Angular Velocity** | The angular velocity of new particles in degrees per second. |
| **Rnd Angular Velocity** | A random angular velocity modifier for new particles. |
| **Rnd Rotation** | If enabled, the particles will be spawned with random rotations. |
| **Simulate In World Space** | If enabled, the particles don't move when the emitter moves. If false, when you move the emitter, the particles follow it around. |
| **One Shot** | If enabled, the particle numbers specified by min & max emission is spawned all at once. If disabled, the particles are generated in a long stream. |
| **Interpolate Triangles** | If enabled, particles are spawned all over the mesh's surface. If disabled, particles are only spawned from the mesh's vertrices. |
| **Systematic** | If enabled, particles are spawned in the order of the vertices defined in the mesh. Although you seldom have direct control over vertex order in meshes, most 3D modelling applications have a very systematic setup when using primitives. It is important that the mesh contains no faces in order for this to work. |
| **Min Normal Velocity** | Minimum amount that particles are thrown away from the mesh. |
| **Max Normal Velocity** | Maximum amount that particles are thrown away from the mesh. |

## Details

Mesh Particle Emitters (MPEs) are used when you want more precise control over the spawn position & directions than the simpler **Ellipsoid Particle Emitter** gives you. They can be used for making advanced effects.

MPEs work by emitting particles at the vertices of the attached mesh. Therefore, the areas of your mesh that are more dense with polygons will be more dense with particle emission.

Particle Emitters work in conjunction with Particle Animators and Particle Renderers to create, manipulate, and display Particle Systems. All three Components must be present on an object before the particles will behave correctly. When particles are being emitted, all different velocities are added together to create the final velocity.

#### Spawning Properties

Spawning properties like **Size**, **Energy**, **Emission**, and **Velocity** will give your particle system distinct personality when trying to achieve different effects. Having a small **Size** could simulate fireflies or stars in the sky. A large **Size** could simulate dust clouds in a musky old building.

**Energy** and **Emission** will control how long your particles remain onscreen and how many particles can appear at any one time. For example, a rocket might have high **Emission** to simulate density of smoke, and high **Energy** to simulate the slow dispersion of smoke into the air.

**Velocity** will control how your particles move. You might want to change your **Velocity** in scripting to achieve interesting effects, or if you want to simulate a constant effect like wind, set your X and Z **Velocity** to make your particles blow away.

#### Simulate in World Space

If this is disabled, the position of each individual particle will always translate relative to the **Position** of the emitter. When the emitter moves, the particles will move along with it. If you have **Simulate in World Space** enabled, particles will not be affected by the translation of the emitter. For example, if you have a fireball that is spurting flames that rise, the flames will be spawned and float up in space as the fireball gets further away. If **Simulate in World Space** is disabled, those same flames will move across the screen along with the fireball.

#### Emitter Velocity Scale

This property will only apply if **Simulate in World Space** is enabled.

If this property is set to 1, the particles will inherit the exact translation of the emitter at the time they are spawned. If it is set to 2, the particles will inherit double the emitter's translation when they are spawned. 3 is triple the translation, etc.

#### One Shot

**One Shot** emitters will create all particles within the **Emission** property all at once, and cease to emit particles over time. Here are some examples of different particle system uses with **One Shot Enabled** or **Disabled**:

**Enabled**:

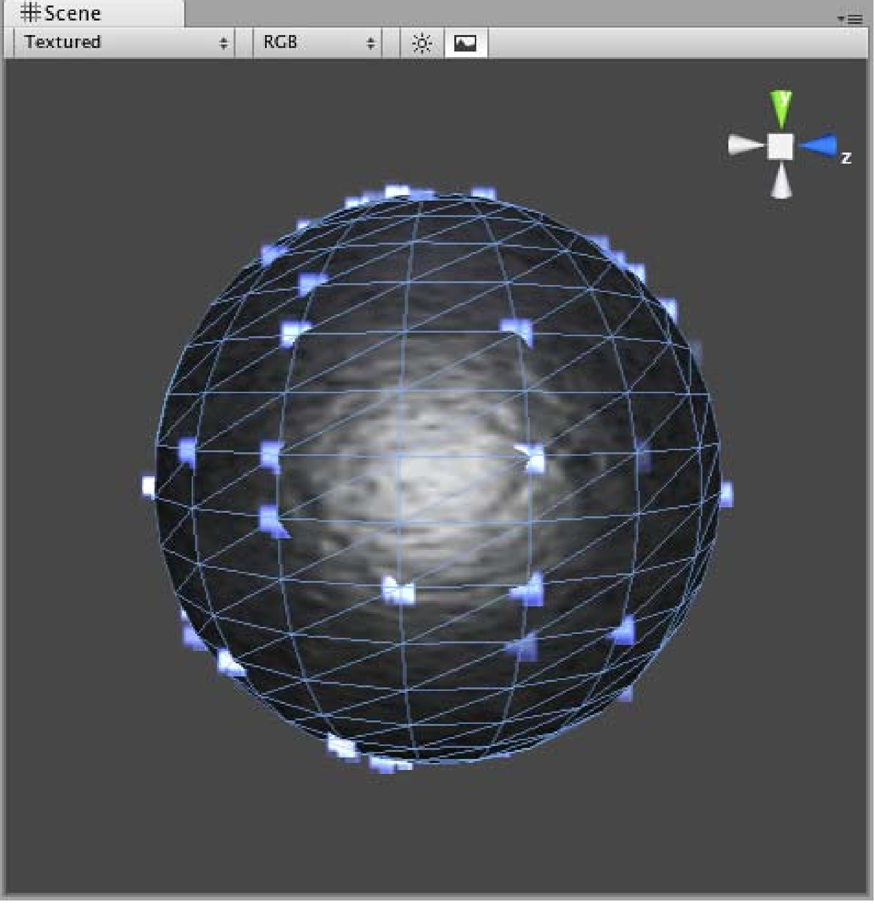
- Explosion
- Water splash
- Magic spell

**Disabled**:

- Gun barrel smoke
- Wind effect
- Waterfall

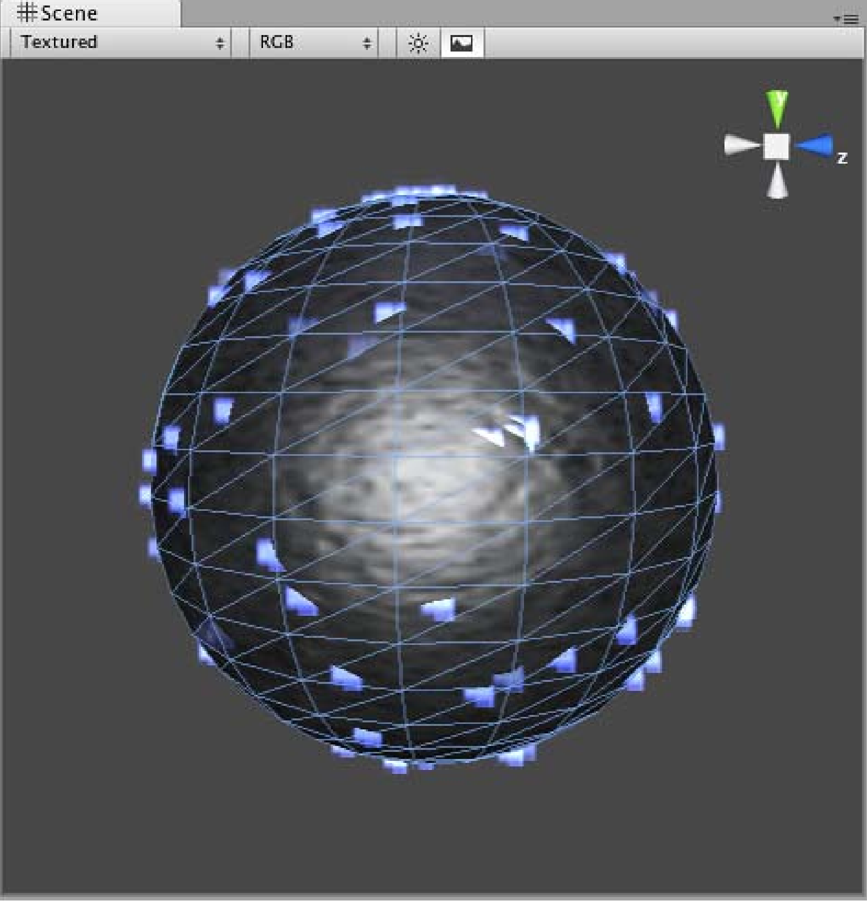
### Interpolate Triangles

Enabling your emitter to **Interpolate Triangles** will allow particles to be spawned between the mesh's vertices. This option is off by default, so particles will only be spawned at the vertices.




*A sphere with **Interpolate Triangles** off (the default)*

Enabling this option will spawn particles on and in-between vertices, essentially all over the mesh's surface (seen below).

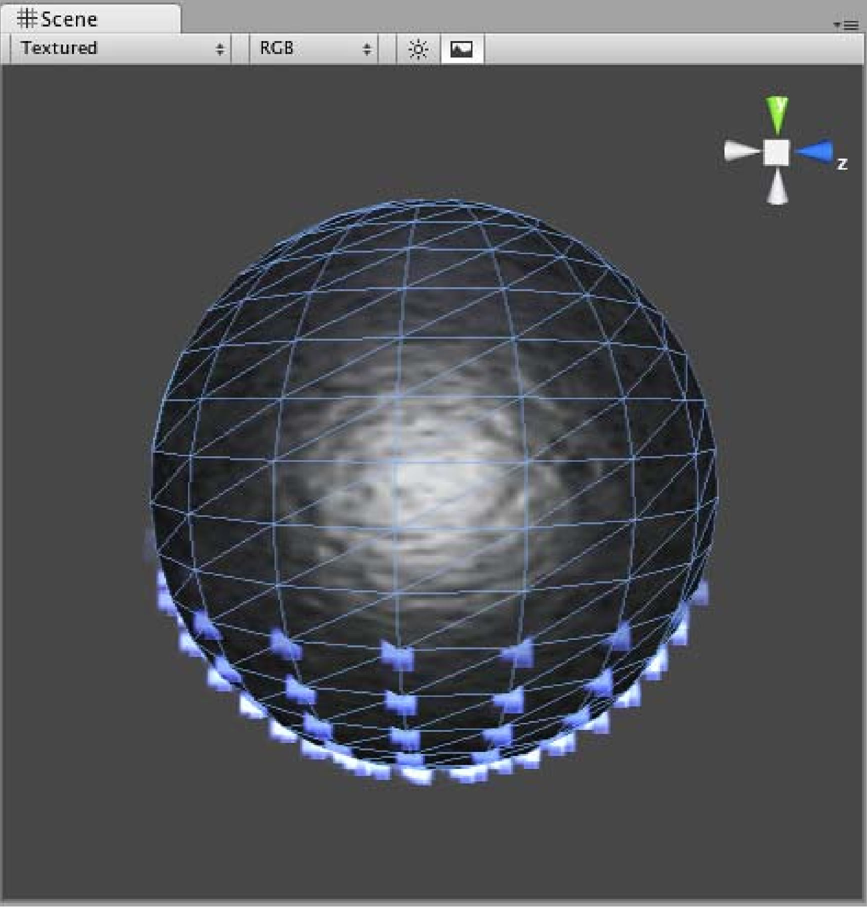


*A sphere with **Interpolate Triangles** on*

It bears repeating that even with **Interpolate Triangles** enabled, particles will still be denser in areas of your mesh that are more dense with polygons.

### Systematic

Enabling **Systematic** will cause your particles to be spawned in your mesh's vertex order. The vertex order is set by your 3D modeling application.




*An MPE attached to a sphere with* ***Systematic*** *enabled*

### Normal Velocity

**Normal Velocity** controls the speed at which particles are emitted along the normal from where they are spawned.

For example, create a Mesh Particle System, use a cube mesh as the emitter, enable **Interpolate Triangles**, and set **Normal Velocity Min** and **Max** to 1. You will now see the particles emit from the faces of the cube in a straight line.

### See Also

- How to make a Mesh Particle Emitter

### Hints

- Be careful of using many large particles. This can seriously hinder performance on low-level machines. Always try to use the minimum number of particles to attain an effect.
- The **Emit** property works in conjunction with the **AutoDestruct** property of the Particle Animator. Through scripting, you can cease the emitter from emitting, and then **AutoDestruct** will automatically destroy the Particle System and the GameObject it is attached to.
- MPEs can also be used to make glow from a lot of lamps placed in a scene. Simply make a mesh with one vertex in the center of each lamp, and build an MPE from that with a halo material. Great for evil sci-fi worlds.

Page last updated: 2011-12-01

---

## class-ParticleAnimator

**Particle Animators** move your particles over time, you use them to apply wind, drag & color cycling to your particle systems.

*The Particle Animator* ***Inspector***

## Properties

| | |
|---|---|
| **Does Animate Color** | If enabled, particles cycle their color over their lifetime. |
| **Color Animation** | The 5 colors particles go through. All particles cycle over this - if some have a shorter life span than others, they will animate faster. |
| **World Rotation Axis** | An optional world-space axis the particles rotate around. Use this to make advanced spell effects or give caustic bubbles some life. |
| **Local Rotation Axis** | An optional local-space axis the particles rotate around. Use this to make advanced spell effects or give caustic bubbles some life. |
| **Size Grow** | Use this to make particles grow in size over their lifetime. As randomized forces will spread your particles out, it is often nice to make them grow in size so they don't fall apart. Use this to make smoke rise upwards, to simulate wind, etc. |
| **Rnd Force** | A random force added to particles every frame. Use this to make smoke become more alive. |
| **Force** | The force being applied every frame to the particles, measure relative to the world. |
| **Damping** | How much particles are slowed every frame. A value of 1 gives no damping, while less makes them slow down. |
| **Autodestruct** | If enabled, the GameObject attached to the Particle Animator will be destroyed when all particles disappear. |

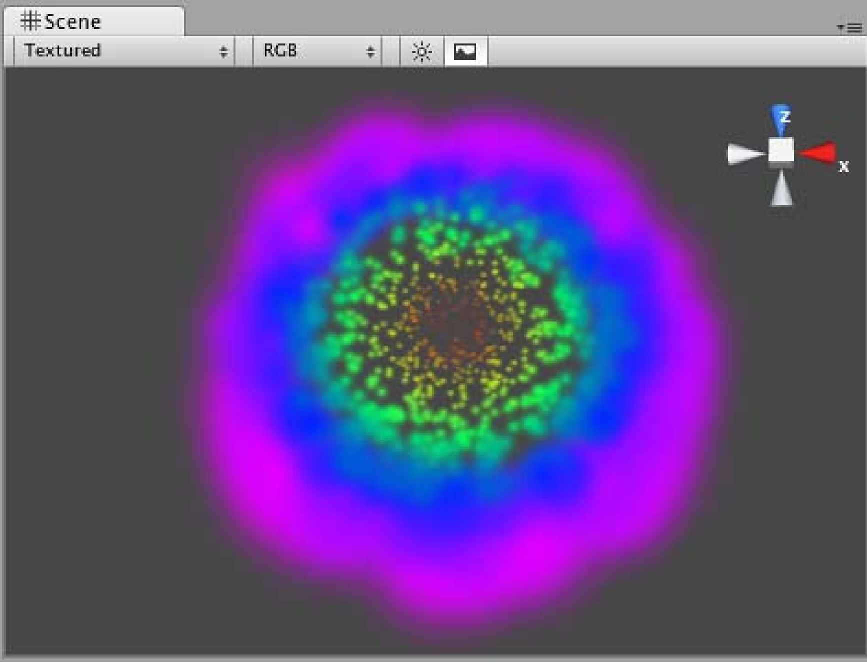
## Details

Particle Animators allow your particle systems to be dynamic. They allow you to change the color of your particles, apply forces and rotation, and choose to destroy them when they are finished emitting. For more information about Particle Systems, reference Mesh Particle Emitters, Ellipsoid Particle Emitters, and Particle Renderers.

### Animating Color

If you would like your particles to change colors or fade in/out, enable them to **Animate Color** and specify the colors for the cycle. Any particle system that animates color will cycle through the 5 colors you choose. The speed at which they cycle will be determined by the Emitter's **Energy** value.

If you want your particles to fade in rather than instantly appear, set your first or last color to have a low Alpha value.




*An **Animating Color** Particle System*

**Rotation Axes**
Setting values in either the Local or World **Rotation Axes** will cause all spawned particles to rotate around the indicated axis (with the **Transform's** position as the center). The greater the value is entered on one of these axes, the faster the rotation will be.

Setting values in the Local Axes will cause the rotating particles to adjust their rotation as the Transform's rotation changes, to match its local axes.

Setting values in the World Axes will cause the particles' rotation to be consistent, regardless of the Transform's rotation.

**Forces & Damping**
You use force to make particles accelerate in the direction specified by the force.

**Damping** can be used to decelerate or accelerate without changing their direction:

- A value of 1 means no **Damping** is applied, the particles will not slow down or accelerate.
- A value of 0 means particles will stop immediately.
- A value of 2 means particles will double their speed every second.

**Destroying GameObjects attached to Particles**
You can destroy the Particle System and any attached **GameObject** by enabling the **AutoDestruct** property. For example, if you have an oil drum, you can attach a Particle System that has **Emit** disabled and **AutoDestruct** enabled. On collision, you enable the Particle Emitter. The explosion will occur and after it is over, the Particle System and the oil drum will be destroyed and removed from the scene.

Note that automatic destruction takes effect only after some particles have been emitted. The precise rules for when the object is destroyed when **AutoDestruct** is on:

- If there have been some particles emitted already, but all of them are dead now, *or*
- If the emitter did have **Emit** on at some point, but now **Emit** is off.

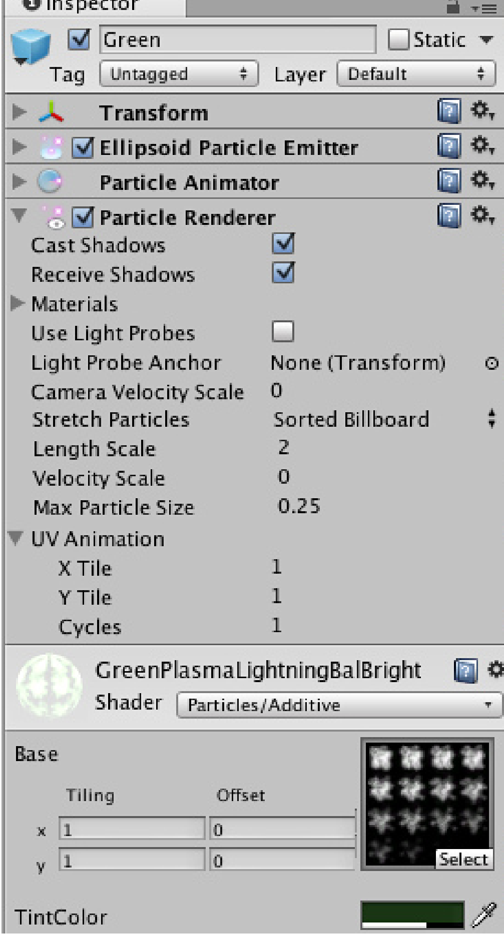
## Hints

- Use the **Color Animation** to make your particles fade in & out over their lifetime - otherwise, you will get nasty-looking pops.
- Use the **Rotation Axes** to make whirlpool-like swirly motions.

Page last updated: 2011-12-01

# class-ParticleRenderer

The **Particle Renderer** renders the **Particle System** on screen.



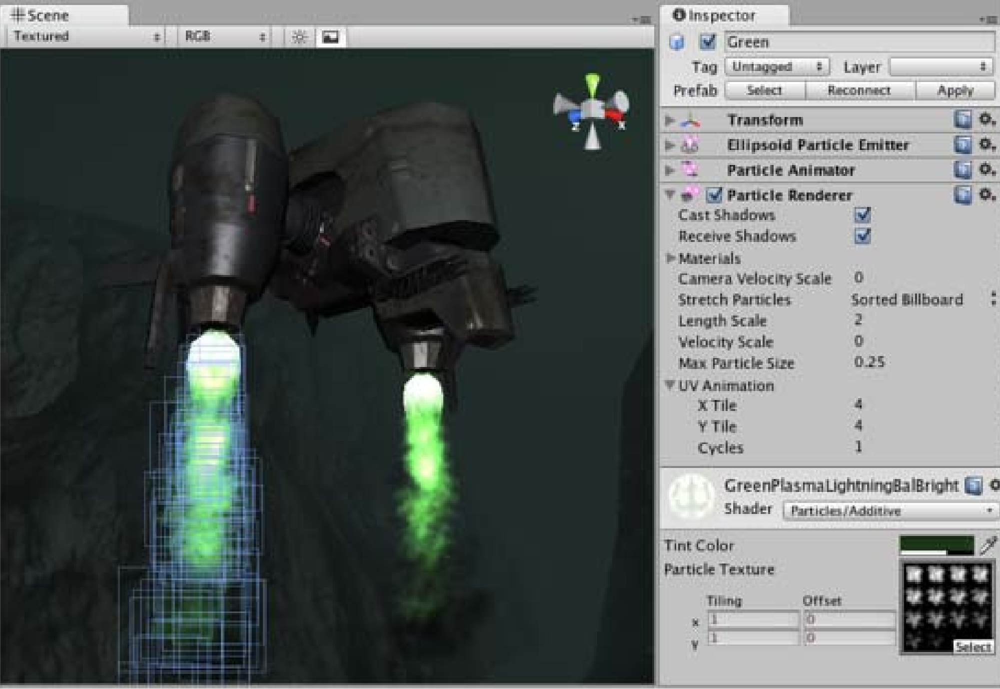

*The Particle Renderer **Inspector***

## Properties

| | |
|---|---|
| **Cast Shadows** | If enabled, allows object to cast shadows. |
| **Receive Shadows** | If enabled, allows object to receive shadows. |
| **Materials** | Reference to a list of **Materials** that will be displayed in the position of each individual particle. |
| **Use Light Probes** | If enabled and baked light probes are present in the scene, an interpolated light probe. |
| **Light Probe Anchor** | If set, Renderer will use this Transform's position to find the interpolated light probe. |
| **Camera Velocity Scale** | The amount of stretching that is applied to the Particles based on Camera movement. |
| **Stretch Particles** | Determines how the particles are rendered. |
| **Billboard** | The particles are rendered as if facing the camera. |
| **Stretched** | The particles are facing the direction they are moving. |
| **SortedBillboard** | The particles are sorted by depth. Use this when using a blending material. |
| **VerticalBillboard** | All particles are aligned flat along the X/Z axes. |
| **HorizontalBillboard** | All particles are aligned flat along the X/Y axes. |
| **Length Scale** | If **Stretch Particles** is set to **Stretched**, this value determines how long the particles are in their direction of motion. |
| **Velocity Scale** | If **Stretch Particles** is set to **Stretched**, this value determines the rate at which particles will be stretched, based on their movement speed. |
| **UV Animation** | If either of these are set, the UV coordinates of the particles will be generated for use with a tile animated texture. See the section on Animated Textures below. |
| **X Tile** | Number of frames located across the X axis. |
| **Y Tile** | Number of frames located across the Y axis. |
| **Cycles** | How many times to loop the animation sequence. |

## Details

Particle Renderers are required for any Particle Systems to be displayed on the screen.



*A Particle Renderer makes the Gunship's engine exhaust appear on the screen*

### Choosing a Material

When setting up a Particle Renderer it is very important to use an appropriate material and shader that renders both sides of the material. Most of the time you want to use a Material with one of the built-in Particle Shaders. There are some premade materials in the **Standard Assets->Particles->Sources** folder that you can use.

Creating a new material is easy:

1. Select **Assets->Create Other->Material** from the menu bar.
2. The Material has a shader popup, choose one of the shaders in the Particles group. Eg. **Particles->Multiply**.
3. Now assign a Texture. The different shaders use the alpha channel of the textures slightly differently, but most of the time a value of black will make it invisible and white in the alpha channel will display it on screen.

### Distorting particles

By default particles are rendered billboarded. That is simple square sprites. This is good for smoke and explosions and most other particle effects.

Particles can be made to either stretch with the velocity. This is useful for sparks, lightning or laser beams. **Length Scale** and **Velocity Scale** affects how long the stretched particle will be.

**Sorted Billboard** can be used to make all particles sort by depth. Sometimes this is necessary, mostly when using **Alpha Blended** particle shaders. This can be expensive and should only be used if it really makes a quality difference when rendering.

### Animated textures

Particle Systems can be rendered with an animated tile texture. To use this feature, make the texture out of a grid of images. As the particles go through their life cycle, they will cycle through the images. This is good for adding more life to your particles, or making small rotating debris pieces.

## Hints

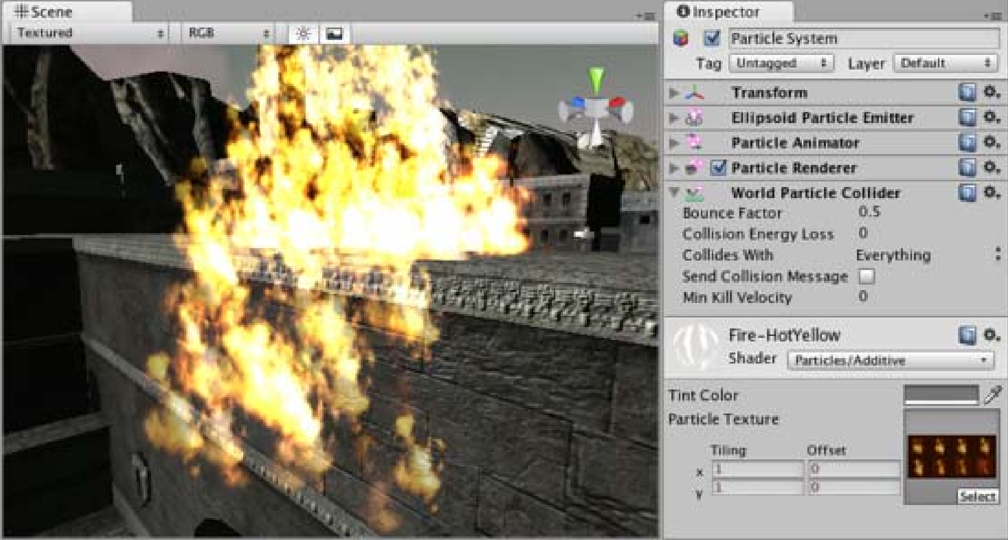
- Use Particle Shaders with the Particle Renderer.

Page last updated: 2012-11-30

---

# class-WorldParticleCollider

The **World Particle Collider** is used to collide particles against other **Colliders** in the scene.




*A **Particle System** colliding with a **Mesh Collider***

## Properties

| | |
|---|---|
| **Bounce Factor** | Particles can be accelerated or slowed down when they collide against other objects. This factor is similar to the **Particle Animator's Damping** property. |
| **Collision Energy Loss** | Amount of energy (in seconds) a particle should lose when colliding. If the energy goes below 0, the particle is killed. |
| **Min Kill Velocity** | If a particle's **Velocity** drops below **Min Kill Velocity** because of a collision, it will be eliminated. |
| **Collides with** | Which Layers the particle will collide against. |
| **Send Collision Message** | If enabled, every particle sends out a collision message that you can catch through scripting. |

## Details

To create a Particle System with Particle Collider:

1. Create a Particle System using **GameObject->Create Other->Particle System**
2. Add the Particle Collider using **Component->Particles->World Particle Collider**

### Messaging

If **Send Collision Message** is enabled, any particles that are in a collision will send the message **OnParticleCollision()** to both the particle's **GameObject** and the GameObject the particle collided with.

## Hints

- **Send Collision Message** can be used to simulate bullets and apply damage on impact.
- Particle Collision Detection is slow when used with a lot of particles. Use Particle Collision Detection wisely.
- Message sending introduces a large overhead and shouldn't be used for normal Particle Systems.

Page last updated: 2011-12-01

---

# comp-RenderingGroup

This group contains all **Components** that have to do with rendering in-game and user interface elements. Lighting and special effects are also included in this group.

- Camera
- Flare Layer
- GUI Layer
- GUI Text

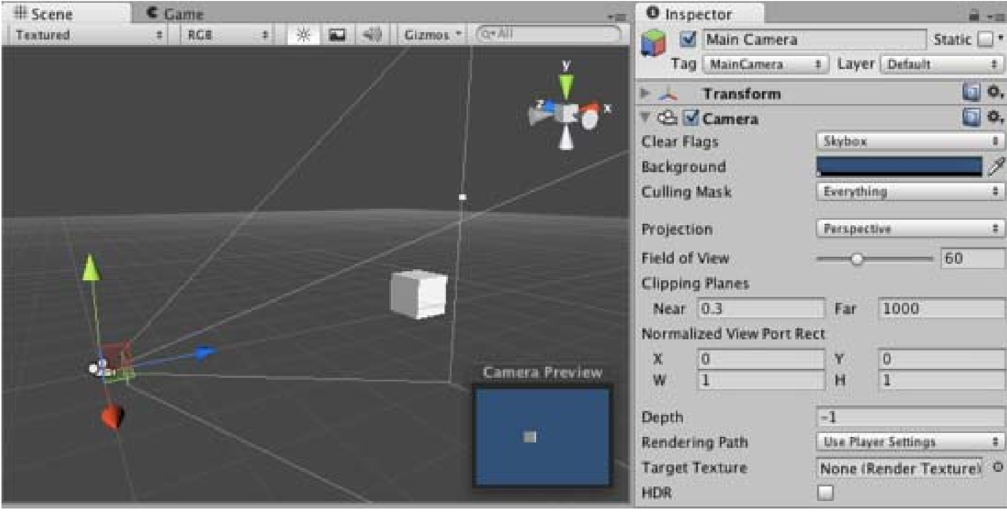
- GUI Texture
- Light
- Light Probe Group
- Occlusion Area (Pro Only)
- Occlusion Portals
- Skybox
- Level of Detail (Pro Only)
- 3D Textures

Page last updated: 2007-07-16

---

# class-Camera

**Cameras** are the devices that capture and display the world to the player. By customizing and manipulating cameras, you can make the presentation of your game truly unique. You can have an unlimited number of cameras in a scene. They can be set to render in any order, at any place on the screen, or only certain parts of the screen.




*Unity's flexible Camera object*

## Properties

| | |
|---|---|
| **Clear Flags** | Determines which parts of the screen will be cleared. This is handy when using multiple Cameras to draw different game elements. |
| **Background** | Color applied to the remaining screen after all elements in view have been drawn and there is no skybox. |
| **Culling Mask** | Include or omit layers of objects to be rendered by the Camera. Assign layers to your objects in the Inspector. |
| **Projection** | Toggles the camera's capability to simulate perspective. |
|     **Perspective** | Camera will render objects with perspective intact. |
|     **Orthographic** | Camera will render objects uniformly, with no sense of perspective. |
| **Size** (when Orthographic is selected) | The viewport size of the Camera when set to Orthographic. |
| **Field of view** | Width of the Camera's view angle, measured in degrees along the local Y axis. |
| **Clipping Planes** | Distances from the camera to start and stop rendering. |
|     **Near** | The closest point relative to the camera that drawing will occur. |
|     **Far** | The furthest point relative to the camera that drawing will occur. |
| **Normalized View Port Rect** | Four values that indicate where on the screen this camera view will be drawn, in Screen Coordinates (values 0-1). |
|     **X** | The beginning horizontal position that the camera view will be drawn. |
|     **Y** | The beginning vertical position that the camera view will be drawn. |
|     **W** (Width) | Width of the camera output on the screen. |
|     **H** (Height) | Height of the camera output on the screen. |
| **Depth** | The camera's position in the draw order. Cameras with a larger value will be drawn on top of cameras with a smaller value. |

| | |
|---|---|
| **Rendering Path** | Options for defining what rendering methods will be used by the camera. |
| **Use Player Settings** | This camera will use whichever Rendering Path is set in the Player Settings. |
| **Vertex Lit** | All objects rendered by this camera will be rendered as Vertex-Lit objects. |
| **Forward** | All objects will be rendered with one pass per material, as was standard in Unity 2.x. |
| **Deferred Lighting** (Unity Pro only) | All objects will be drawn once without lighting, then lighting of all objects will be rendered together at the end of the render queue. |
| **Target Texture** (Unity Pro/Advanced only) | Reference to a Render Texture that will contain the output of the Camera view. Making this reference will disable this Camera's capability to render to the screen. |
| **HDR** | Enables High Dynamic Range rendering for this camera. |

## Details

Cameras are essential for displaying your game to the player. They can be customized, scripted, or parented to achieve just about any kind of effect imaginable. For a puzzle game, you might keep the Camera static for a full view of the puzzle. For a first-person shooter, you would parent the Camera to the player character, and place it at the character's eye level. For a racing game, you'd likely want to have the Camera follow your player's vehicle.

You can create multiple Cameras and assign each one to a different **Depth**. Cameras are drawn from low **Depth** to high **Depth**. In other words, a Camera with a **Depth** of 2 will be drawn on top of a Camera with a depth of 1. You can adjust the values of the **Normalized View Port Rectangle** property to resize and position the Camera's view onscreen. This can create multiple mini-views like missile cams, map views, rear-view mirrors, etc.

### Render Path

Unity supports different Rendering Paths. You should choose which one you use depending on your game content and target platform / hardware. Different rendering paths have different features and performance characteristics that mostly affect Lights and Shadows.
The rendering Path used by your project is chosen in Player Settings. Additionally, you can override it for each Camera.

For more info on rendering paths, check the rendering paths page.

### Clear Flags

Each Camera stores color and depth information when it renders its view. The portions of the screen that are not drawn in are empty, and will display the skybox by default. When you are using multiple Cameras, each one stores its own color and depth information in buffers, accumulating more data as each Camera renders. As any particular Camera in your scene renders its view, you can set the **Clear Flags** to clear different collections of the buffer information. This is done by choosing one of the four options:

#### Skybox

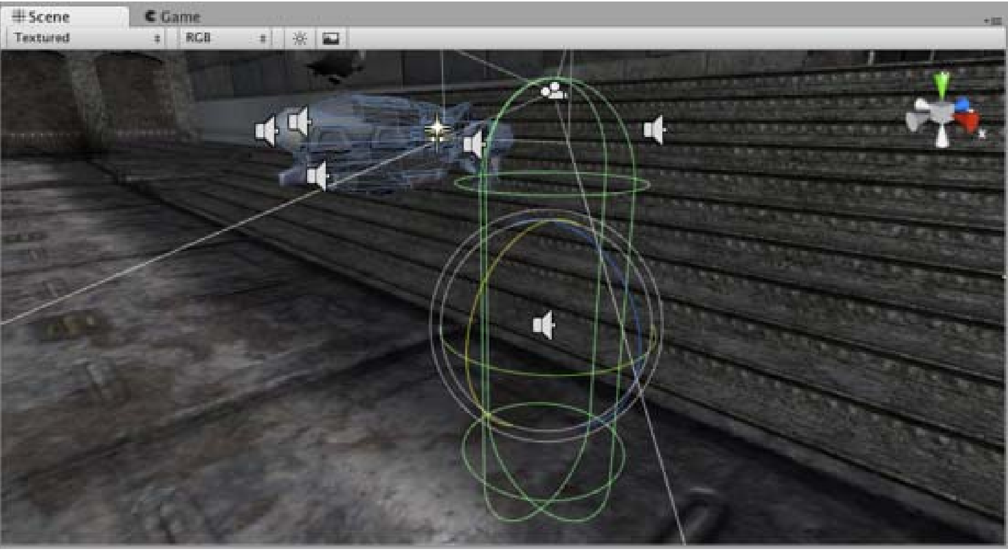
This is the default setting. Any empty portions of the screen will display the current Camera's skybox. If the current Camera has no skybox set, it will default to the skybox chosen in the Render Settings (found in **Edit->Render Settings**). It will then fall back to the **Background Color**. Alternatively a Skybox component can be added to the camera. If you want to create a new Skybox, you can use this guide.

#### Solid Color

Any empty portions of the screen will display the current Camera's **Background Color**.

#### Depth Only

If you wanted to draw a player's gun without letting it get clipped inside the environment, you would set one Camera at **Depth** 0 to draw the environment, and another Camera at **Depth** 1 to draw the weapon alone. The weapon Camera's **Clear Flags** should be set to to **depth only**. This will keep the graphical display of the environment on the screen, but discard all information about where each object exists in 3-D space. When the gun is drawn, the opaque parts will completely cover anything drawn, regardless of how close the gun is to the wall.



*The gun is drawn last, after clearing the depth buffer of the cameras before it*

**Don't Clear**
This mode does not clear either the color or the depth buffer. The result is that each frame is drawn over the next, resulting in a smear-looking effect. This isn't typically used in games, and would likely be best used with a custom shader.

**Clip Planes**
The **Near** and **Far Clip Plane** properties determine where the Camera's view begins and ends. The planes are laid out perpendicular to the Camera's direction and are measured from the its position. The **Near plane** is the closest location that will be rendered, and the **Far plane** is the furthest.

The clipping planes also determine how depth buffer precision is distributed over the scene. In general, to get better precision you should move the **Near plane** as far as possible.

Note that the near and far clip planes together with the planes defined by the field of view of the camera describe what is popularly known as the camera *frustum*. Unity ensures that when rendering your objects those which are completely outside of this frustum are not displayed. This is called Frustum Culling. Frustum Culling happens irrespective of whether you use Occlusion Culling in your game.

For performance reasons, you might want to cull small objects earlier. For example, small rocks and debris could be made invisible at much smaller distance than large buildings. To do that, put small objects into a separate layer and setup per-layer cull distances using Camera.layerCullDistances script function.

**Culling Mask**
The **Culling Mask** is used for selectively rendering groups of objects using Layers. More information on using layers can be found here.

Commonly, it is good practice to put your User Interface on a different layer, then render it by itself with a separate Camera set to render the UI layer by itself.

In order for the UI to display on top of the other Camera views, you'll also need to set the **Clear Flags** to **Depth only** and make sure that the UI Camera's **Depth** is higher than the other Cameras.

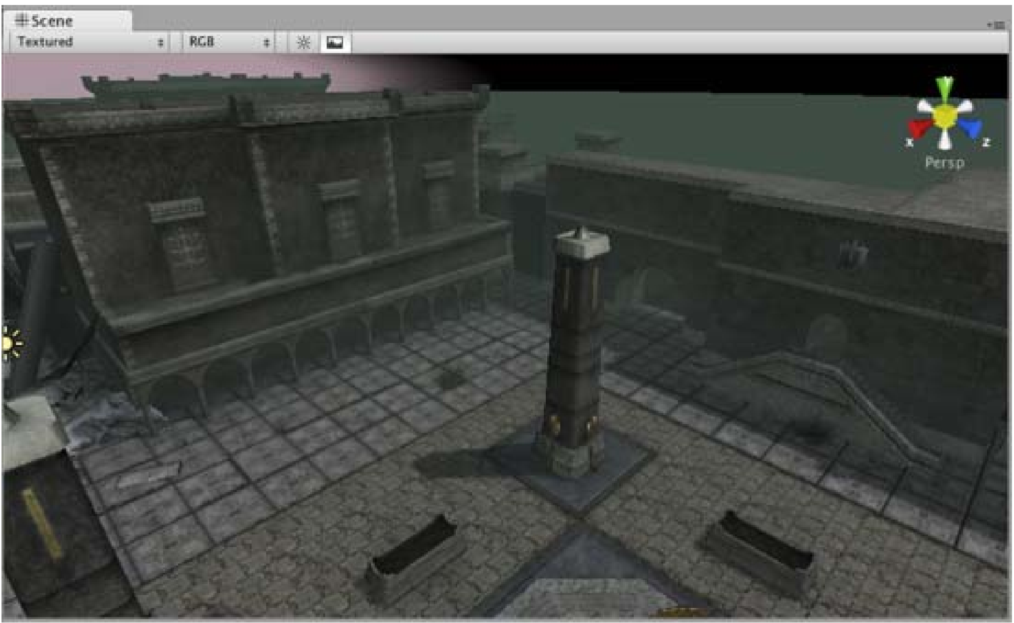
**Normalized Viewport Rectangle**
**Normalized Viewport Rectangles** are specifically for defining a certain portion of the screen that the current camera view will be drawn upon. You can put a map view in the lower-right hand corner of the screen, or a missile-tip view in the upper-left corner. With a bit of design work, you can use **Viewport Rectangle** to create some unique behaviors.

It's easy to create a two-player split screen effect using **Normalized Viewport Rectangle**. After you have created your two cameras, change both camera H value to be 0.5 then set player one's Y value to 0.5, and player two's Y value to 0. This will make player one's camera display from halfway up the screen to the top, and player two's camera will start at the bottom and stop halfway up the screen.

*Two-player display created with **Normalized Viewport Rectangle***

**Orthographic**
Marking a Camera as **Orthographic** removes all perspective from the Camera's view. This is mostly useful for making isometric or 2D games.

Note that fog is rendered uniformly in orthographic camera mode and may therefore not appear as expected. Read more about why in the component reference on Render Settings.



*Perspective camera.*

*Orthographic camera. Objects do not get smaller with distance here!*

**Render Texture**
This feature is only available for Unity Advanced licenses . It will place the camera's view onto a Texture that can then be applied to another object. This makes it easy to create sports arena video monitors, surveillance cameras, reflections etc.

*A Render Texture used to create a live arena-cam*

## Hints

- Cameras can be instantiated, parented, and scripted just like any other GameObject.
- To increase the sense of speed in a racing game, use a high **Field of View**.
- Cameras can be used in physics simulation if you add a **Rigidbody** Component.
- There is no limit to the number of Cameras you can have in your scenes.
- Orthographic cameras are great for making 3D user interfaces
- If you are experiencing depth artifacts (surfaces close to each other flickering), try setting **Near Plane** to as large as possible.
- Cameras cannot render to the Game Screen and a Render Texture at the same time, only one or the other.
- Pro license holders have the option of rendering a Camera's view to a texture, called Render-to-Texture, for even more unique effects.
- Unity comes with pre-installed Camera scripts, found in **Components->Camera Control**. Experiment with them to get a taste of what's possible.

Page last updated: 2011-11-10

## class-FlareLayer

The **Flare Layer** Component can be attached to Cameras to make Lens Flares appear in the image. By default, Cameras have a Flare Layer already attached.

Page last updated: 2007-07-30

## class-GUILayer

A **GUI Layer** Component is attached to a Camera to enable rendering of 2D GUIs.

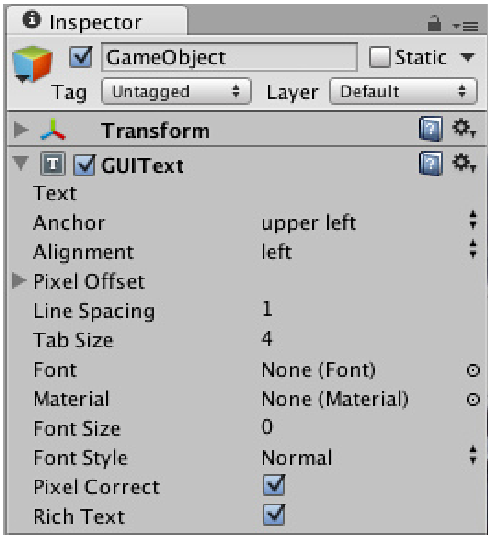
When a GUI Layer is attached to a Camera it will render all GUI Textures and GUI Texts in the scene. GUI Layers do not affect UnityGUI in any way.

You can enable and disable rendering GUI in a single camera by clicking on the check box of the GUI Layer in the **Inspector**.

Page last updated: 2007-09-21

## class-GuiText

**GUI Text** displays text of any font you import in screen coordinates.



*The GUI Text **Inspector***

**Please Note**: Unity 2.0 introduced **UnityGUI**, a GUI Scripting system. You may prefer creating user interface elements with UnityGUI instead of GUI Texts. Read more about how to use UnityGUI in the GUI Scripting Guide.

### Properties

| | |
|---|---|
| **Text** | The string to display. |
| **Anchor** | The point at which the **Text** shares the position of the **Transform**. |
| **Alignment** | How multiple lines are aligned within the **GUIText**. |
| **Pixel Offset** | Offset of the text relative to the position of the GUIText in the screen. |
| **Line Spacing** | How much space will be in-between lines of **Text**. |
| **Tab Size** | How much space will be inserted for a tab ('\t') character. As a multiplum of the space character offset. |
| **Font** | The Font to use when rendering the text. |

| | |
|---|---|
| **Material** | Reference to the **Material** containing the characters to be drawn. If set, this property overrides the one in the Font asset. |
| **Font Size** | The font size to use. Set to 0 to use the default font size. Only applicable for dynamic fonts. |
| **Font Style** | The font style to use. (Normal, Bold, Italic or Bold and Italic). Only applicable for dynamic fonts. |
| **Pixel Correct** | If enabled, all **Text** characters will be drawn in the size of the imported font texture. If disabled, the characters will be resized based on the Transform's **Scale**. |
| **Rich Text** | If enabled, allows HTML-style tags for text formatting. |

## Details

GUI Texts are used to print text onto the screen in 2D. The **Camera** has to have a GUI Layer attached in order to render the text. Cameras include a GUI Layer by default, so don't remove it if you want to display a GUI Text. GUI Texts are positioned using only the X and Y axes. Rather than being positioned in World Coordinates, GUI Texts are positioned in Screen Coordinates, where (0,0) is the bottom-left and (1,1) is the top-right corner of the screen

To import a font see the Font page.

### Pixel Correct

By default, GUI Texts are rendered with **Pixel Correct** enabled. This makes them look crisp and they will stay the same size in pixels independent of the screen resolution.

## Hints

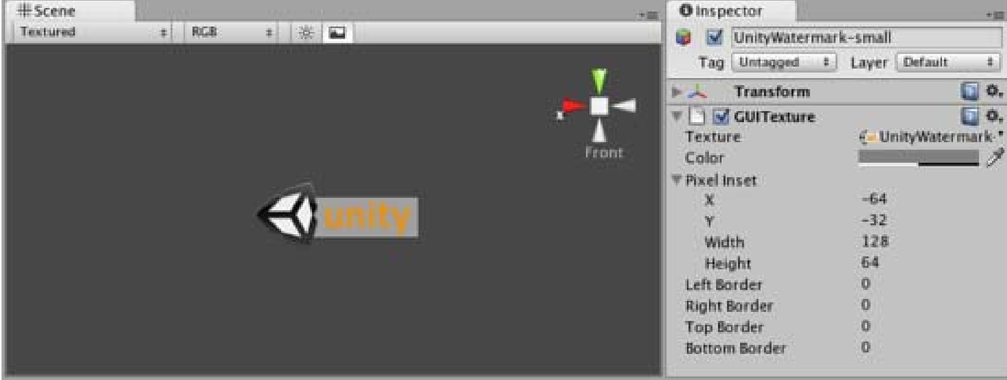
- When entering text into the **Text** property, you can create a line break by holding **Alt** and pressing **Return**.
- If you are scripting the **Text** property, you can add line breaks by inserting the escape character "\n" in your strings.
- You can download free true type fonts from 1001freefonts.com (download the Windows fonts since they contain TrueType fonts).

Page last updated: 2012-11-20

---

# class-GuiTexture

**GUI Textures** are displayed as flat images in 2D. They are made especially for user interface elements, buttons, or decorations. Their positioning and scaling is performed along the x and y axes only, and they are measured in **Screen Coordinates**, rather than **World Coordinates**.




*The GUI Texture* ***Inspector***

**Please Note**: Unity 2.0 introduced **UnityGUI**, a GUI Scripting system. You may prefer creating user interface elements with UnityGUI instead of GUI Textures. Read more about how to use UnityGUI in the GUI Scripting Guide.

## Properties

| | |
|---|---|
| **Texture** | Reference to the Texture that will be used as the texture's display. |
| **Color** | Color that will tint the **Texture** drawn on screen. |
| **Pixel Inset** | Used for pixel-level control of the scaling and positioning of the GUI Texture. All values are measured relative to the position of the GUI Texture's **Transform**. |
| **X** | Left-most pixel position of the texture. |
| **Y** | Bottom-most pixel position of the texture. |
| **Width** | Right-most pixel position of the texture. |

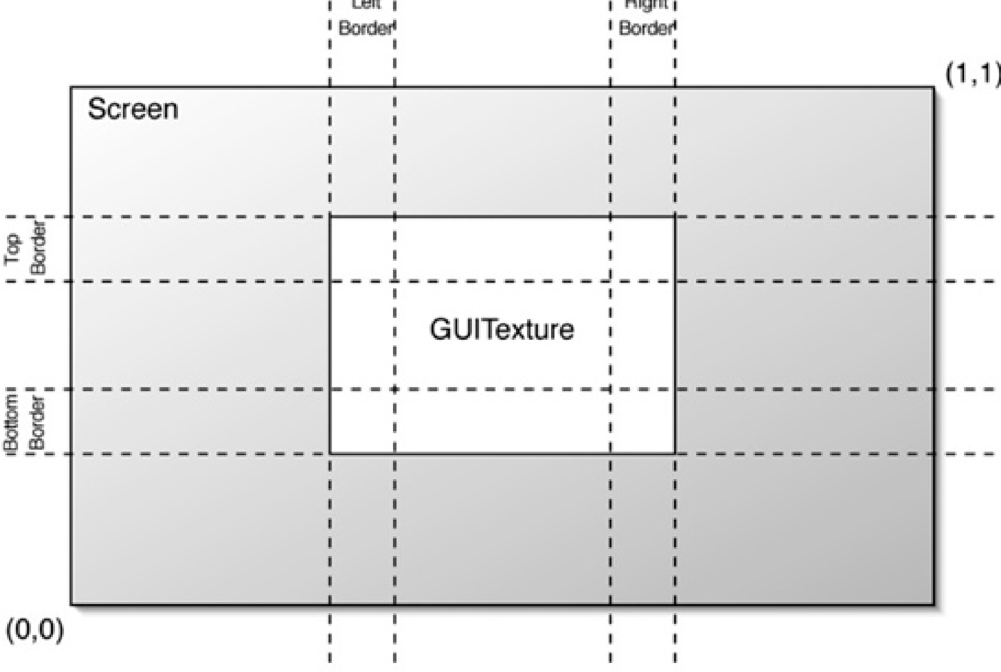
| | |
|---|---|
| **Height** | Top-most pixel position of the texture. |
| **Left Border** | Number of pixels from the left that are not affected by scale. |
| **Right Border** | Number of pixels from the right that are not affected by scale. |
| **Top Border** | Number of pixels from the top that are not affected by scale. |
| **Bottom Border** | Number of pixels from the bottom that are not affected by scale. |

## Details

To create a GUITexture:

1. Select a Texture in the **Project View**
2. Choose **GameObject->Create Other->GUI Texture** from the menu bar

GUI Textures are perfect for presenting game interface backgrounds, buttons, or other elements to the player. Through scripting, you can easily provide visual feedback for different "states" of the texture -- when the mouse is hovering over the texture, or is actively clicking it for example. Here is the basic breakdown of how the GUI Texture is calculated:




*GUI Textures are laid out according to these rules*

*The GUI elements seen here were all created with GUI Textures*

**Borders**
The number of pixels that will not scale with the texture at each edge of the image. As you rarely know the resolution your game runs in, chances are your GUI will get scaled. Some GUI textures have a border at the edge that is meant to be an exact number of pixels. In order for this to work, set the border sizes to match those from the texture.

**Pixel Inset**
The purpose of the **Pixel Inset** is to prevent textures from scaling with screen resolution, and keeping thim in a fixed pixel size. This allows you to render a texture without any scaling. This means that players who run your game in higher resolutions will see your textures in smaller areas of the screen, allowing them to have more screen real-estate for your gameplay graphics.

To use it effectively, you need to set the scale of the GUI Texture's Transform to (0, 0, 0). Now, the **Pixel Inset** is in full control of the texture's size and you can set the **Pixel Inset** values to be the exact pixel size of your Texture.

### Hints

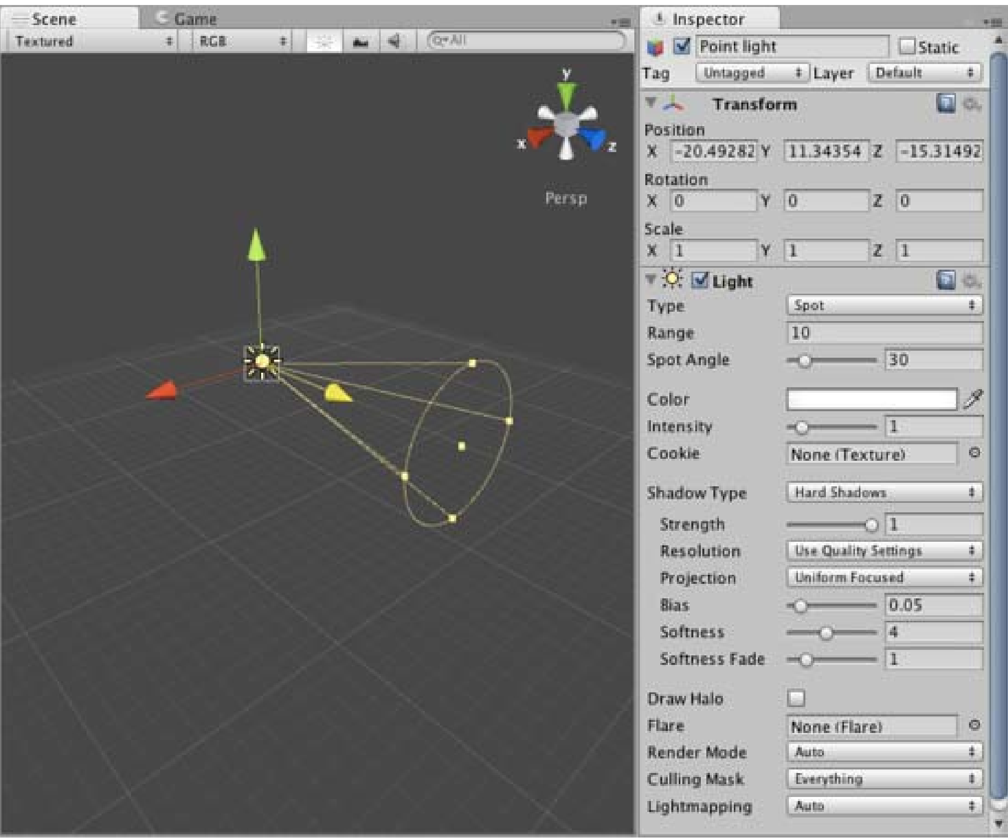
- The depth of each layered GUI Texture is determined by its individual Z Transform position, not the global Z position.
- GUI Textures are great for making menu screens, or pause/escape menu screens.
- You should use **Pixel Inset** on any GUI Textures that you want to be a specific number of pixels for the width and height.

Page last updated: 2010-06-24

---

## class-Light

**Lights** will bring personality and flavor to your game. You use lights to illuminate the scenes and objects to create the perfect visual mood. Lights can be used to simulate the sun, burning match light, flashlights, gun-fire, or explosions, just to name a few.



*The Light **Inspector***

There are four types of lights in Unity:

- **Point lights** shine from a location equally in all directions, like a light bulb.
- **Directional lights** are placed infinitely far away and affect everything in the scene, like the sun.
- **Spot lights** shine from a point in a direction and only illuminate objects within a cone - like the headlights of a car.

- **Area lights** (only available for lightmap baking) shine in all directions to one side of a rectangular section of a plane.

Lights can also cast **Shadows**. Shadows are a Pro-only feature. Shadow properties can be adjusted on a per-light basis.

## Properties

| | |
|---|---|
| **Type** | The current type of light object: |
| **Directional** | A light placed infinitely far away. It affects everything in the scene. |
| **Point** | A light that shines equally in all directions from its location, affecting all objects within its **Range**. |
| **Spot** | A light that shines everywhere within a cone defined by **Spot Angle** and **Range**. Only objects within this region are affected by the light. |
| **Area** | A light that shines in all directions to one side of a rectangular area of a plane, affecting all objects within its **Range**. The rectangle is defined by the X and Y properties. Area lights are only available during lightmap baking and have no effect on objects at runtime. |
| **Range** | How far light is emitted from the center of the object. **Point**/**Spot** light only. |
| **Spot Angle** | Determines the angle of the cone in degrees. **Spot** light only. |
| **Color** | The color of the light emitted. |
| **Intensity** | Brightness of the light. Default value for a **Point**/**Spot** light is 1. Default value for a **Directional** light is 0.5 |
| **Cookie** | The alpha channel of this texture is used as a mask that determines how bright the light is at different places. If the light is a **Spot** or a **Directional** light, this must be a 2D texture. If the light is a **Point** light, it must be a **Cubemap**. |
| **Cookie Size** | Scales the projection of a Cookie. **Directional** light only. |
| **Shadow Type** (Pro only) | **No**, **Hard** or **Soft** shadows that will be cast by this light. Only applicable to desktop build targets. Soft shadows are more expensive. |
| **Strength** | The darkness of the shadows. Values are between 0 and 1. |
| **Resolution** | Detail level of the shadows. |
| **Bias** | Offset used when comparing the pixel position in light space with the value from the shadow map. See *Shadow Mapping and the Bias Property* below |
| **Softness** | Scales the penumbra region (the offset of blur samples). **Directional** light only. |
| **Softness Fade** | Shadow softness fade based on the distance from the camera. **Directional** light only. |
| **Draw Halo** | If checked, a spherical halo of light will be drawn with a radius equal to **Range**. |
| **Flare** | Optional reference to the Flare that will be rendered at the light's position. |
| **Render Mode** | Importance of this light. This can affect lighting fidelity and performance, see *Performance Considerations* below. Options include: |
| **Auto** | The rendering method is determined at runtime depending on the brightness of nearby lights and current Quality Settings for desktop build target. |
| **Important** | This light is always rendered at per-pixel quality. Use this for very important effects only (e.g. headlights of a player's car). |
| **Not Important** | This light is always rendered in a faster, vertex/object light mode. |
| **Culling Mask** | Use to selectively exclude groups of objects from being affected by the light; see Layers. |
| **Lightmapping** | The Lightmapping mode: **RealtimeOnly**, **Auto** or **BakedOnly**; see the Dual Lightmaps description. |
| **X** | (Area lights only) The width of the rectangular light area. |
| **Y** | (Area lights only) The height of the rectangular light area. |

## Details

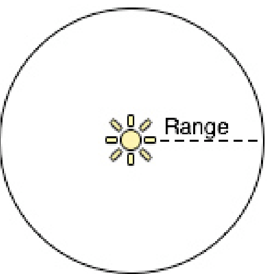
There are four basic light types in Unity. Each type can be customized to fit your needs.

You can create a texture that contains an alpha channel and assign it to the **Cookie** variable of the light. The Cookie will be projected from the light. The Cookie's alpha mask modulates the light amount, creating light and dark spots on surfaces. They are a great way af adding lots of complexity or atmosphere to a scene.

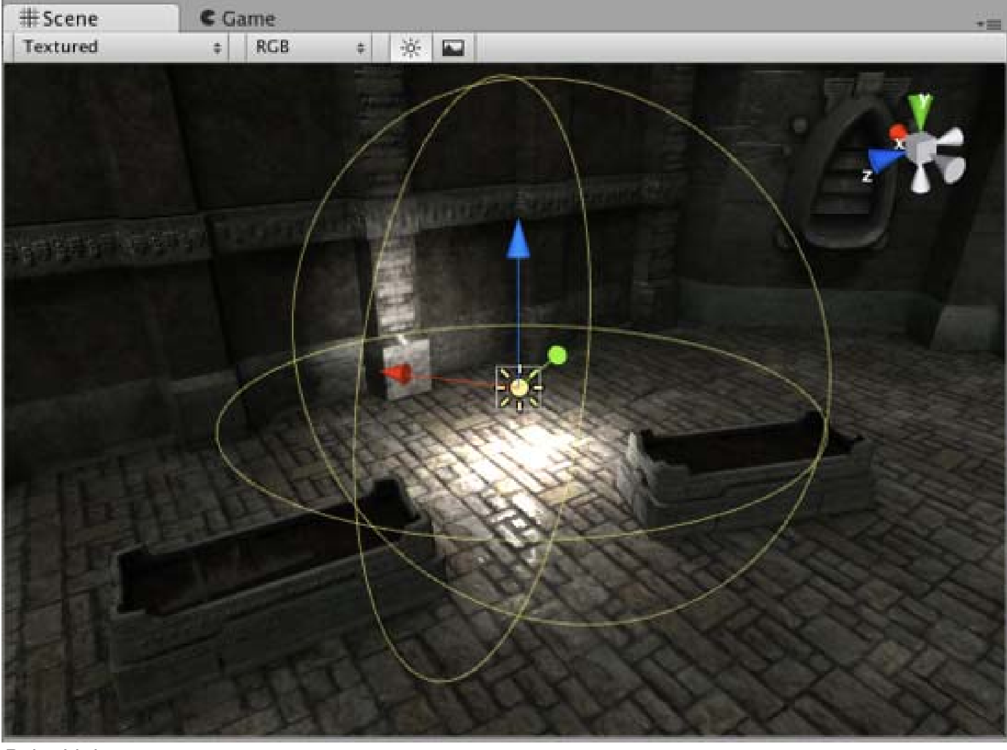
All built-in shaders in Unity seamlessly work with any type of light. **VertexLit** shaders cannot display Cookies or Shadows, however.

In Unity Pro with a build target of webplayer or standalone, all Lights can optionally cast Shadows. This is done by selecting either **Hard Shadows** or **Soft Shadows** for the **Shadow Type** property of each individual Light. For more information about shadows, please read the Shadows page.

#### Point Lights

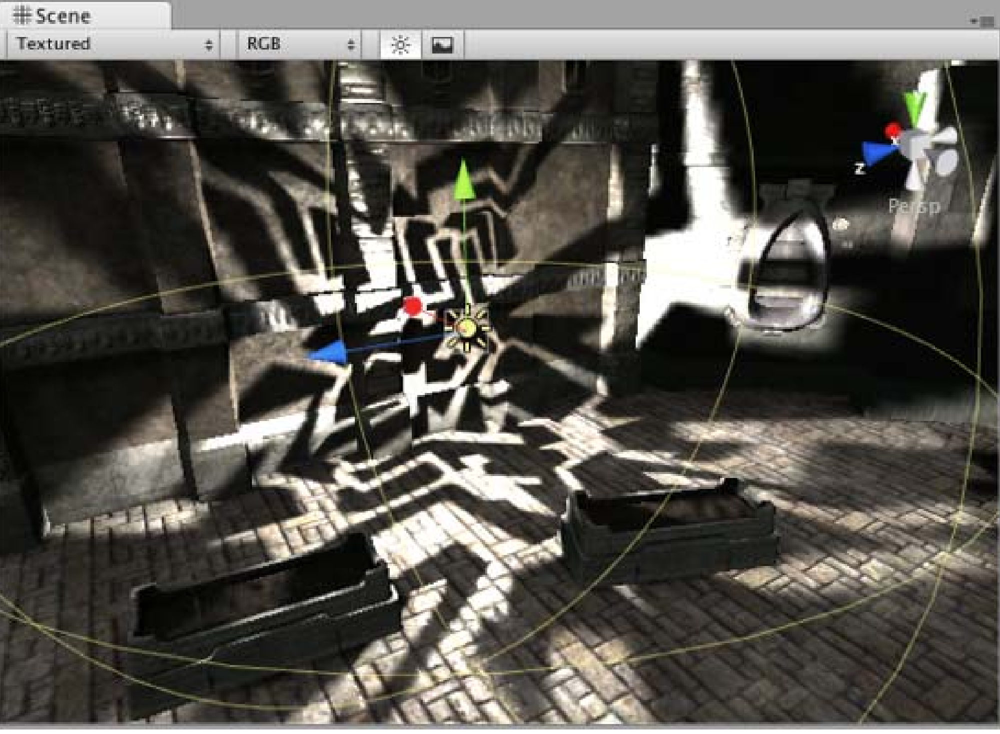



Point lights shine out from a point in all directions. They are the most common lights in computer games - typically used for explosions, light bulbs, etc. They have an average cost on the graphics processor (though point light shadows are the most expensive).



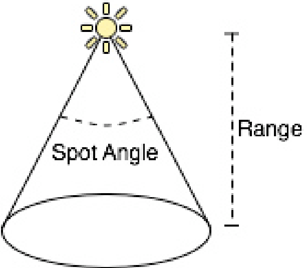

*Point Light*

Point lights can have cookies - Cubemap texture with alpha channel. This Cubemap gets projected out in all directions.

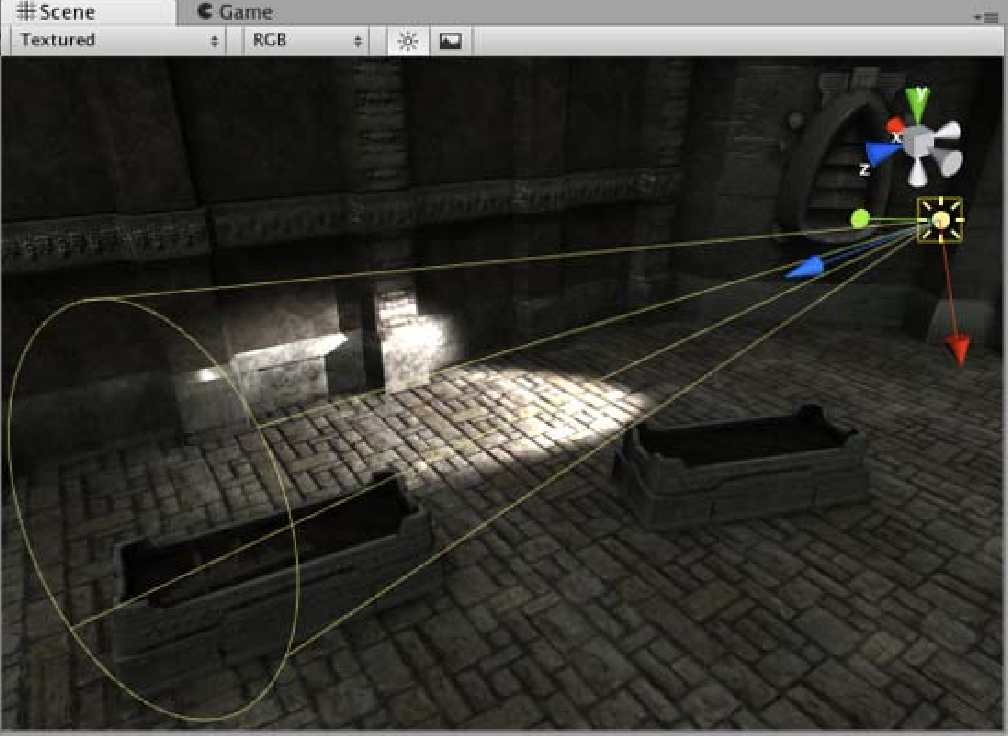



*Point Light with a Cookie*

**Spot Lights**

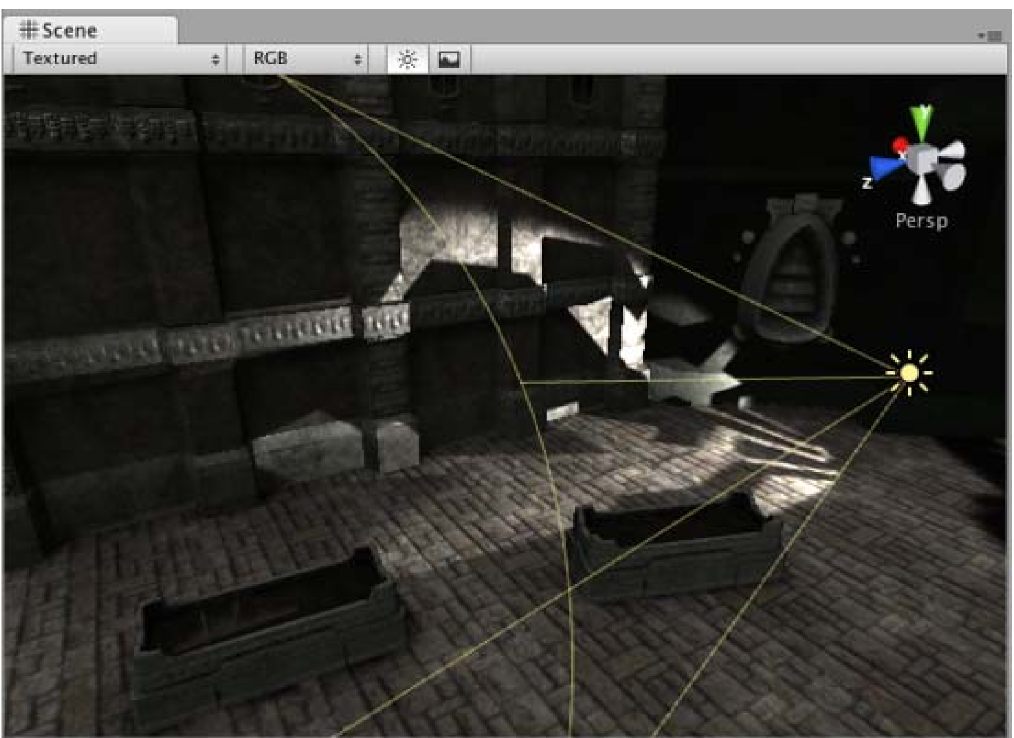



Spot lights only shine in one direction, in a cone. They are perfect for flashlights, car headlights or lamp posts. They are the most expensive on the graphics processor.



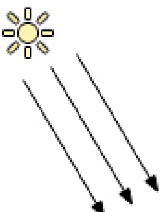

*Spot Light*

Spot lights can also have cookies - a texture projected down the cone of the light. This is good for creating an effect of light shining through the window. It is very important that the texture is black at the edges, has **Border Mipmaps** option on and its wrapping mode is set to **Clamp**. For more info on this, see Textures.




*Spot Light with a Cookie*

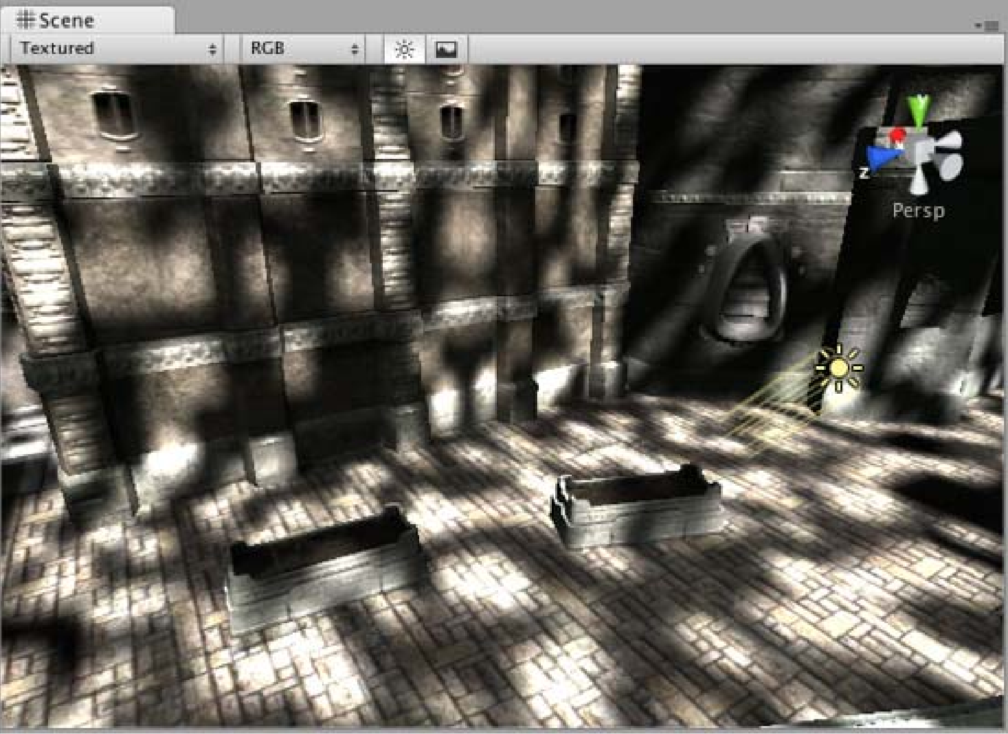
**Directional Lights**



Directional lights are used mainly in outdoor scenes for sun & moonlight. The light affect all surfaces of objects in your scene. They are the least expensive on the graphics processor. Shadows from directional lights (for platforms that support shadows) are explained in depth on this page.

*Directional Light*

When directional light has a cookie, it is projected down the center of the light's Z axis. The size of the cookie is controlled with **Cookie Size** property. Set the cookie texture's wrapping mode to **Repeat** in the **Inspector**.

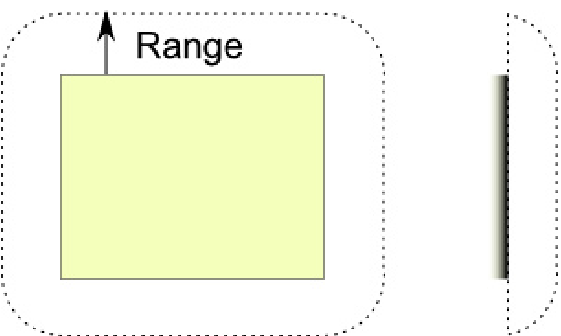


*Directional Light projecting a cloud-like cookie texture*

A cookie is a great way to add some quick detail to large outdoor scenes. You can even slide the light slowly over the scene to give the impression of moving clouds.

### Area Lights

Area lights cast light from one side of a rectangular area of a plane.



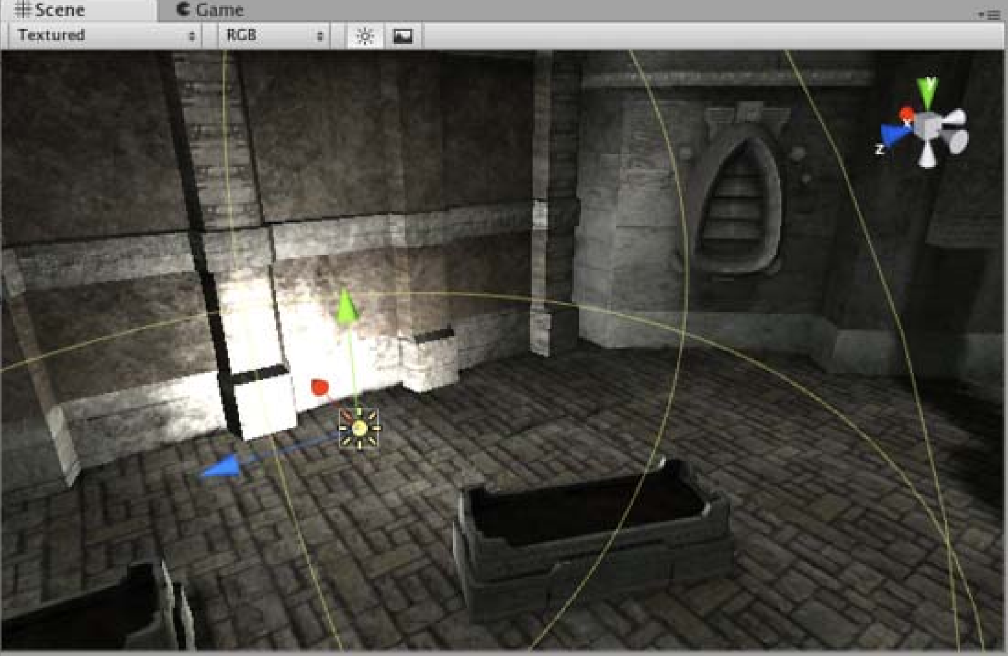

Light is cast on all objects within the light's range. The size of the rectangle is determined by the X and Y properties and and the plane's normal (ie, the side to which light is cast) is the same as the light's positive Z direction. Light is emitted from the whole surface of the rectangle, so shading and shadows from affected object tend to be much softer than with point or directional light sources.

Since the lighting calculation is quite processor-intensive, area lights are not available at runtime and can only be used as a lightmap effect.

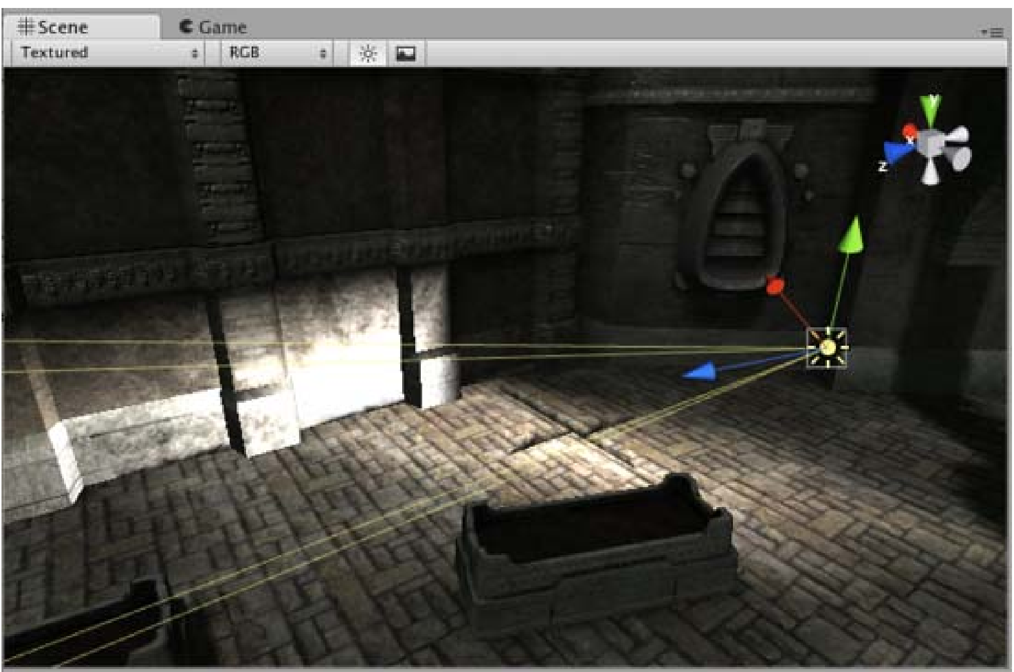
### Performance considerations

Lights can be rendered in one of two methods: **vertex** lighting and **pixel** lighting. Vertex lighting only calculates the lighting at the vertices of the game models and interpolates the lighting over the surfaces of the models. Pixel lights are calculated at every screen pixel, and hence are much more expensive. Some older graphics cards only support vertex lighting.

While pixel lighting is slower to render, it does allow some effects that are not possible with vertex lighting. Normal-mapping, light cookies and realtime shadows are only rendered for pixel lights. Spotlight shapes and Point light highlights are much better when rendered in pixel mode as well. The three light types above would look like this when rendered in vertex light mode:




*Point light in Vertex lighting mode.*




*Spot light in Vertex lighting mode.*

*Directional light in Vertex lighting mode.*

Lights have a big impact on rendering speed - therefore a tradeoff has to be made betwen lighting quality and game speed. Since pixel lights are much more expensive than vertex lights, Unity will only render the brightest lights at per-pixel quality. The actual number of pixel lights can be set in the Quality Settings for webplayer and standalone build targets.

You can explicitly control if a light should be rendered as a vertex or pixel light using the **Render Mode** property. By default Unity will classify the light automatically based on how much the object is affected by the light.

The actual lights that are rendered as pixel lights are determined on an object-by-object case. This means:

- Huge objects with bright lights could use all the pixel lights (depending on the quality settings). If the player is far from these, nearby lights will be rendered as vertex lights. Therefore, it is better to split huge objects up in a couple of small ones.

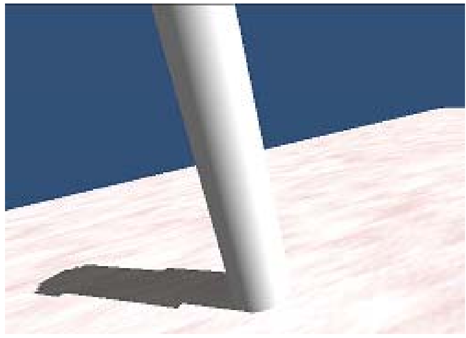
See Optimizing Graphics performance on Desktop, iOS or Android page for more information.

### Creating Cookies

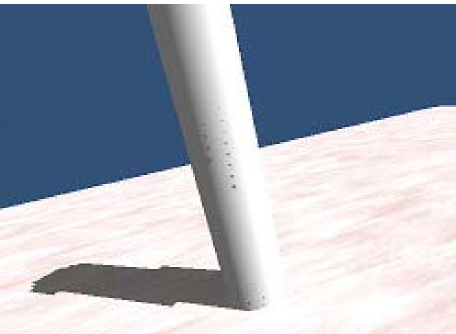
For more information on creating cookies, please see the tutorial on how to create a Spot light cookie.

### Shadow Mapping and the Bias Property

Shadows are implemented using a technique known as shadow mapping. This is analogous to the depth mapping used by a camera to determine which surfaces are obscured by others. The scene is internally rendered by a camera at the position of the light to create a depth map which stores the distance to each surface illuminated by the light. This kind of depth map is referred to as a shadow map, for obvious reasons. When the scene is rendered to the main view camera, each pixel position in the view is transformed into the light's space so that its distance can be compared to the corresponding pixel in the shadow map. If the pixel is more distant than the shadow map pixel then it is presumably obscured from the light by another object and so it will get no illumination.

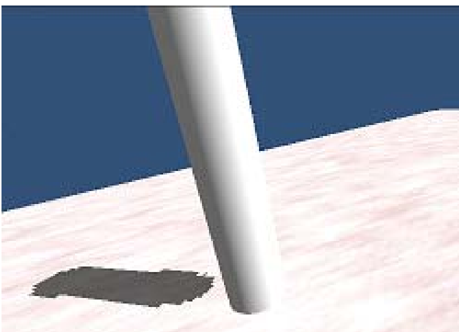

*Cylinder with correct shadowing*

A surface directly illuminated by a light can sometimes appear to be partly in shadow. This is because pixels that should be exactly at the distance specified in the shadow map will sometimes be deemed farther away (a consequence of using a low resolution image for the map). The result is arbitrary patterns of pixels in shadow when they should really be lit, giving a visual effect known as "shadow acne".



*Shadow acne in the form of small dots on the cylinder*

To prevent shadow acne, a bias value can be added to the distance in the shadow map to ensure that pixels on the borderline will definitely pass the comparison as they should. This is the value set by the Bias property associated with a light when it has shadows enabled. It is a mistake to set the bias too high, however, since areas of a shadow near to the object casting it can then sometimes be falsely illuminated. This effect is known as "Peter Panning" (ie, the disconnected shadow makes the object look as if it is flying some way above the ground like Peter Pan).



*Peter Panning makes the object look raised above the ground*

The bias value for a light may need a bit of tweaking to make sure that neither shadow acne nor Peter Panning occur. It is generally easier to gauge the right value by eye rather than attempt to calculate it.

### Hints

- Spot lights with cookies can be extremely effective for making light coming in from windows.
- Low-intensity point lights are good for providing depth to a scene.
- For maximum performance, use a VertexLit shader. This shader only does per-vertex lighting, giving a much higher throughput on low-end cards.
- Auto lights can cast dynamic shadows over lightmapped objects without adding extra illumination. For this to work the Auto lights must be active when the Lightmap is baked. Otherwise they render as real time lights.

Page last updated: 2012-09-21

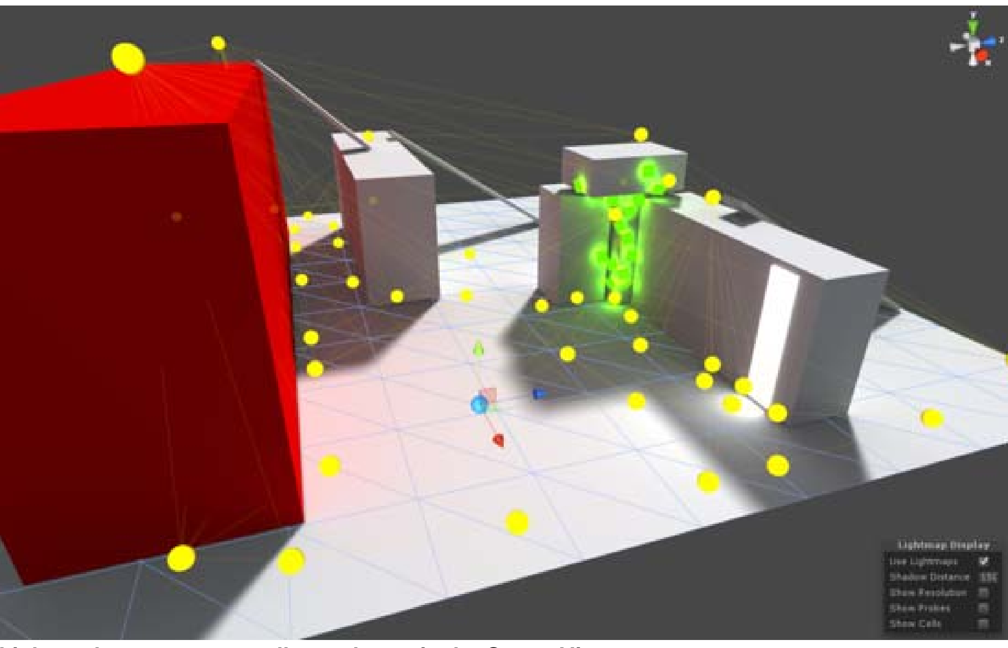
---

## class-LightProbeGroup

A **Light Probe Group** adds one or more light probes to a scene.

A new probe can be created by clicking the **Add Probe** button in the inspector. Once created, the probe can be selected and

moved in much the same way as a GameObject and can be deleted by typing Ctrl/Cmd + Backspace.



**Light probes appear as yellow spheres in the Scene View**

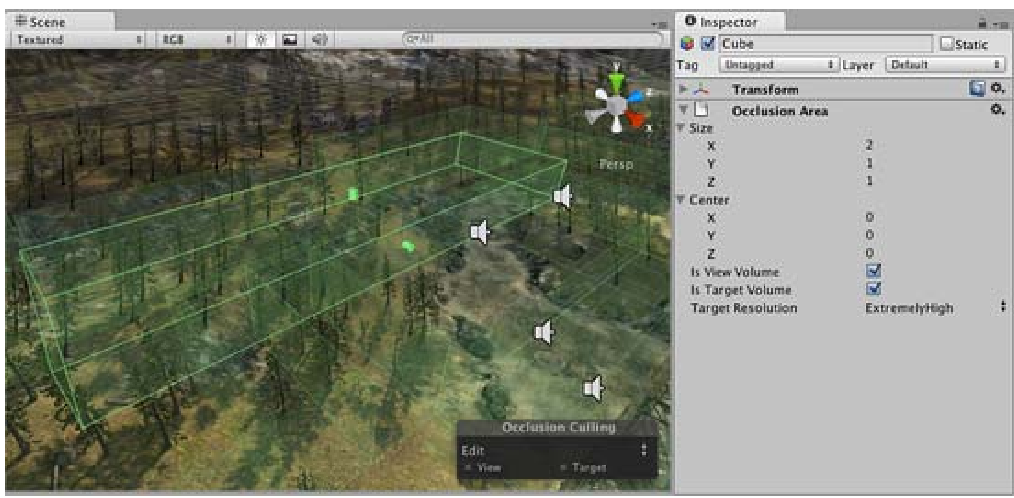
Page last updated: 2011-10-25

---

## class-OcclusionArea

To apply occlusion culling to moving objects you have to create an **Occlusion Area** and then modify its size to fit the space where the moving objects will be located (of course the moving objects cannot be marked as static). You can create Occlusion Areas is by adding the **Occlusion Area** component to an empty game object (**Component->Rendering->Occlusion Area** in the menus)

After creating the **Occlusion Area**, just check the *Is Target Volume* checkbox to occlude moving objects.

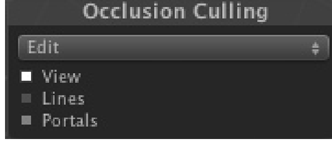



*Occlusion Area properties for moving objects.*

| | |
|---|---|
| **Size** | Defines the size of the Occlusion Area. |
| **Center** | Sets the center of the Occlusion Area. By default this is 0,0,0 and is located in the center of the box. |
| **Is View Volume** | Defines where the camera can be. Check this in order to occlude static objects that are inside this *Occlusion Area*. |
| **Is Target Volume** | **Select this when you want to occlude moving objects.** |

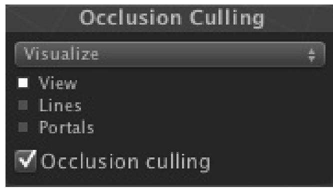
| | |
|---|---|
| **Target Resolution** | Determines how accurate the occlusion culling inside the area will be. This affects the size of the cells in an Occlusion Area. **NOTE: This only affects Target Areas.** |
| **Low** | This takes less time to calculate but is less accurate. |
| **Medium** | This gives a balance between accuracy and time taken to process the occlusion culling data. |
| **High** | This takes longer to calculate but has better accuracy. |
| **Very High** | Use this value when you want to have more accuracy than high resolutions, be aware it takes more time. |
| **Extremely High** | Use this value when you want to have almost exact occlusion culling on your moveable objects. Note: This setting takes a lot of time to calculate. |

After you have added the Occlusion Area, you need to see how it divides the box into cells. To see how the occlusion area will be calculated, Select **Edit** and toggle the **View** button in the **Occlusion Culling Preview Panel**.




### Testing the generated occlusion

After your occlusion is set up, you can test it by enabling the *Occlusion Culling* (in the **Occlusion Culling Preview Panel** in Visualize mode) and moving the **Main Camera** around in the scene view.



*The Occlusion View mode in Scene View*

As you move the Main Camera around (whether or not you are in Play mode), you'll see various objects disable themselves. The thing you are looking for here is any error in the occlusion data. You'll recognize an error if you see objects suddenly popping into view as you move around. If this happens, your options for fixing the error are either to change the resolution (if you are playing with target volumes) or to move objects around to cover up the error. To debug problems with occlusion, you can move the Main Camera to the problematic position for spot-checking.

When the processing is done, you should see some colorful cubes in the View Area. The blue cubes represent the cell divisions for **Target Volumes**. The white cubes represent cell divisions for **View Volumes**. If the parameters were set correctly you should see some objects not being rendered. This will be because they are either outside of the view frustum of the camera or else occluded from view by other objects.

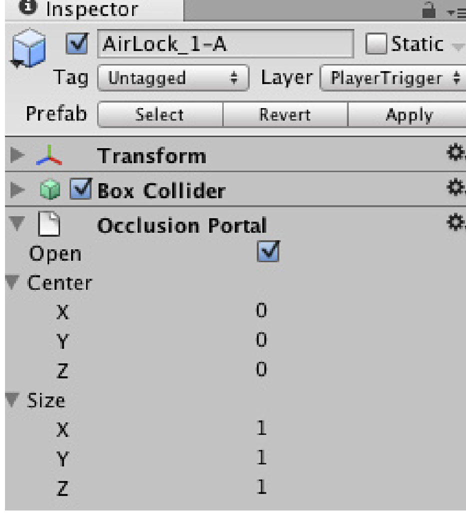
After occlusion is completed, if you don't see anything being occluded in your scene then try breaking your objects into smaller pieces so they can be completely contained inside the cells.

Page last updated: 2012-02-14

---

# class-OcclusionPortal

In order to create occlusion primitive which are openable and closable at runtime, Unity uses **Occlusion Portals**.



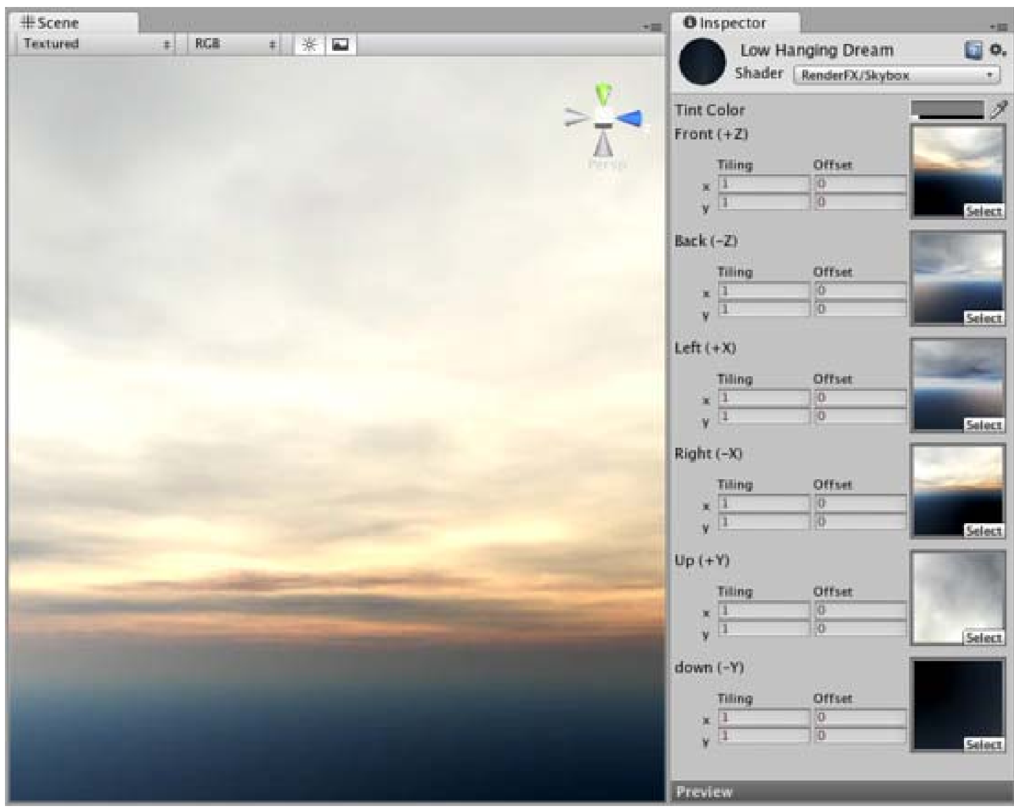

| | |
|---|---|
| **Open** | Indicates if the portal is open (scriptable) |
| **Center** | Sets the center of the Occlusion Area. By default this is 0,0,0 and is located in the center of the box. |
| **Size** | Defines the size of the Occlusion Area. |

Page last updated: 2011-11-17

---

## class-Skybox

**Skyboxes** are a wrapper around your entire scene that display the vast beyond of your world.




*One of the default Skyboxes found under **Standard Assets->Skyboxes***

### Properties

| | |
|---|---|
| **Material** | The **Material** used to render the Skybox, which contains 6 **Textures**. This Material should use the Skybox Shader, and each of the textures should be assigned to the proper global direction. |

### Details

Skyboxes are rendered before anything else in the scene in order to give the impression of complex scenery at the horizon. They are a box of 6 textures, one for each primary direction (+/-X, +/-Y, +/-Z).

You have two options for implementing Skyboxes. You can add them to an individual Camera (usually the main Camera) or you can set up a default Skybox in Render Settings's **Skybox Material** property. The Render Settings is most useful if you want all Cameras in your scene to share the same Skybox.

Adding the Skybox **Component** to a Camera is useful if you want to override the default Skybox set up in the Render Settings. E.g. You might have a split screen game using two Cameras, and want the Second camera to use a different Skybox. To add a Skybox Component to a Camera, click to highlight the Camera and go to **Component->Rendering->Skybox**.

Unity's Standard Assets contain 2 pre-setup Skybox materials in **Standard Assets->Skyboxes**.

If you want to create a new Skybox, use this guide.

### Hints

- If you have a Skybox assigned to a Camera, make sure to set the Camera's **Clear mode** to Skybox.
- It's a good idea to match your Fog color to the skybox's color. Fog color can be set in Render Settings.

Page last updated: 2011-01-19

---

## class-LODGroup

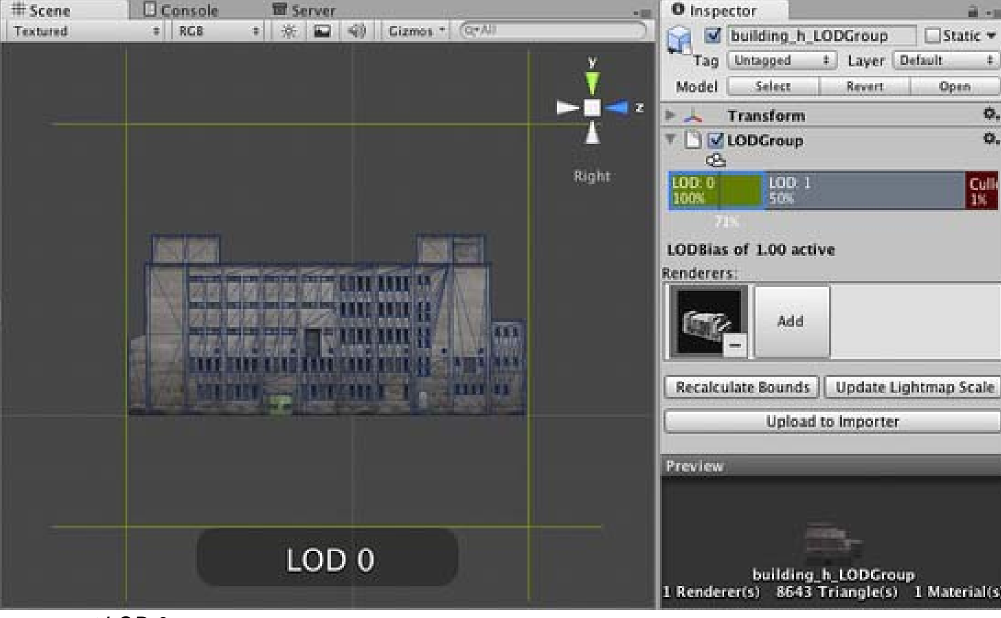
As your scenes get larger, performance becomes a bigger consideration. One of the ways to manage this is to have meshes with different levels of detail depending on how far the camera is from the object. This is called **Level of Detail** (abbreviated as **LOD**).

Here's one of the ways to set up an object with different **LODs**.

1. Create an empty **Game Object** in the scene
2. Create 2 versions of the mesh, a high-res mesh (for **L0D:0**, when camera is the closest), and a low-res mesh (for **L0D:1**, when camera is further away)
3. Add a **LODGroup** component to this object (**Component**->**Rendering**->**LOD Group**)
4. Drag in the object with the high-res mesh onto the first **Renderers** box for **L0D:0**. Say yes to the "Reparent game objects?" dialog
5. Drag in the object with the low-res mesh onto the first **Renderers** box for **LOD:1**. Say yes to the "Reparent game objects?" dialog
6. Right Click on **LOD:2** and remove it.

At this point the empty object should contain both versions of the mesh and "know" which mesh to show depending on how far away the camera is.

You can preview the effect of this by dragging the camera icon left and right in the window for the **LODGroup** component.




*camera at LOD 0*

*camera at LOD 1*

In the **Scene View**, you should be able to see

- Percentage of the view this object occupies
- What **LOD** is currently being displayed
- The number of triangles

### LOD-based naming conventions in the asset import pipeline

In order to simplify setup of LODs, Unity has a naming convention for models that are being imported.

Simply create your meshes in your modelling tool with names ending with _LOD0, _LOD1, _LOD2, etc., and the LOD group with appropriate settings will be created for you.

Note that the convention assumes that the LOD 0 is the highest resolution model.

### Setting up LODs for different platforms

You can tweak your LOD settings for each platform in Quality Settings, in particular the properties of **LOD bias** and **Maximum LOD Level**.

### Utilities

Here are some options that help you work with LODs

| | |
|---|---|
| Recalculate Bounds | If there is new geometry added to the LODGroup that is not reflected in the bounding volume then click this to update the bounds. One example where this is needed is when one of the geometries is part of a prefab, and new geometry is added to that prefab. Geometry added directly to the LODGroup will automatically update the bounds. |
| Update Lightmaps | Updates the **Scale in Lightmap** property in the lightmaps based on the LOD level boundaries. |
| Upload to Importer | Uploads the LOD level boundaries to the importer. |

Page last updated: 2012-11-13

---

## class-Texture3D

Unity supports 3D Texture use and creation from shader and script. While use cases of 3D Textures might not seem as straightforward at first, they can be an integral part of implementing specific kinds of effects such as 3D Color Correction.

Currently, 3D Textures can only be created from script. The following snippet creates a "neutral" 3D texture where, if used as a lookup texture in 3D Color Correction, the performed correction will be the identity.

```
function CreateIdentityLut (dim : int, tex3D : Texture3D)
{
  var newC : Color[] = new Color[dim * dim * dim];
  var oneOverDim : float = 1.0f / (1.0f * dim - 1.0f);
  for(var i : int = 0; i < dim; i++) {
    for(var j : int = 0; j < dim; j++) {
      for(var k : int = 0; k < dim; k++) {
        newC[i + (j*dim) + (k*dim*dim)] = new Color((i*1.0f)*oneOverDim, (j*1.0f)*oneOverDim, (k*1.0f)*oneOverDim, 1.0f);
      }
    }
  }
  tex3D.SetPixels (newC);
  tex3D.Apply ();
}
```

Page last updated: 2012-09-11

---

## comp-TransformationGroup

This group is for all **Components** that handle object positioning outside of Physics.

- Transform

Page last updated: 2007-07-16

---

## class-Transform

The **Transform Component** determines the **Position**, **Rotation**, and **Scale** of each object in the scene. Every object has a Transform.

*The Transform Component is editable in the **Scene View** and in the **Inspector***

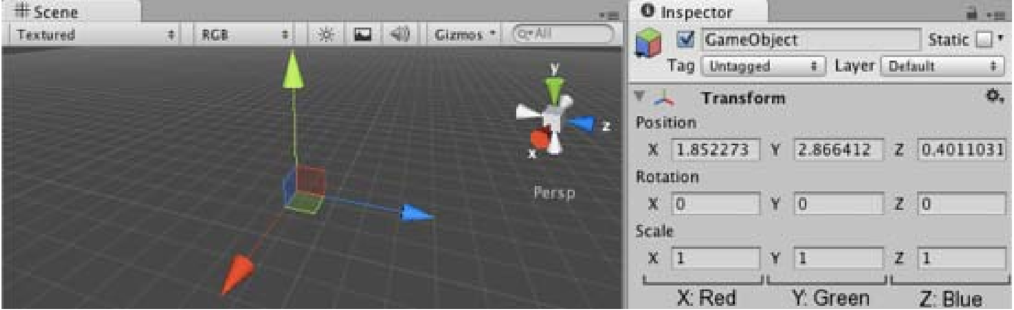
### Properties

| | |
|---|---|
| **Position** | Position of the Transform in X, Y, and Z coordinates. |
| **Rotation** | Rotation of the Transform around the X, Y, and Z axes, measured in degrees. |
| **Scale** | Scale of the Transform along X, Y, and Z axes. Value "1" is the original size (size at which the object was imported). |

All properties of a Transform are measured relative to the Transform's parent (see below for further details). If the Transform has no parent, the properties are measured relative to World Space.

### Using Transforms

Transforms are always manipulated in 3D space in the X, Y, and Z axes. In Unity, these axes are represented by the colors red, green, and blue respectively. Remember: XYZ = RGB.

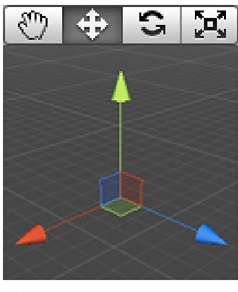


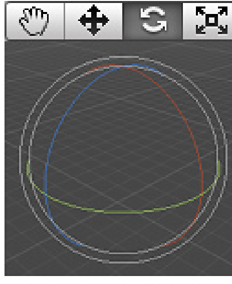

*Color-coded relationship between the three axes and Transform properties*

Transforms can be directly manipulated in the **Scene View** or by editing properties in the Inspector. In the scene, you can modify Transforms using the Move, Rotate and Scale tools. These tools are located in the upper left-hand corner of the Unity Editor.

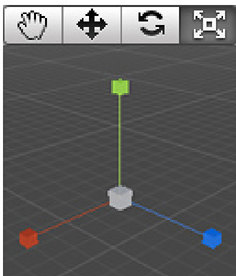
*The View, Translate, Rotate, and Scale tools*

The tools can be used on any object in the scene. When you click on an object, you will see the tool gizmo appear within it. The appearance of the gizmo depends on which tool is selected.





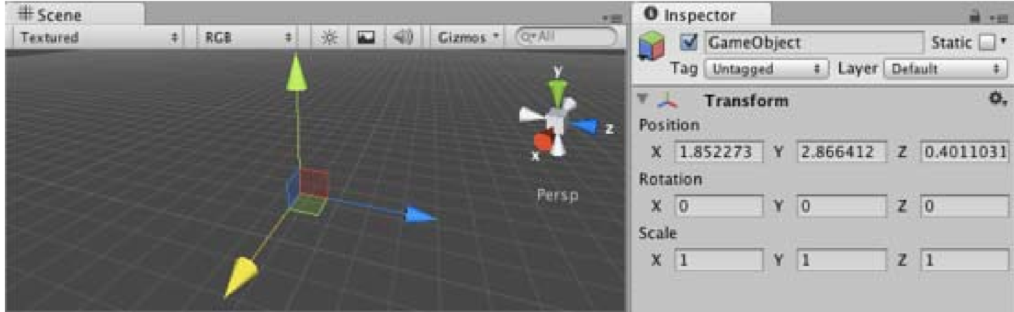
Rotate (E)



Scale (R)

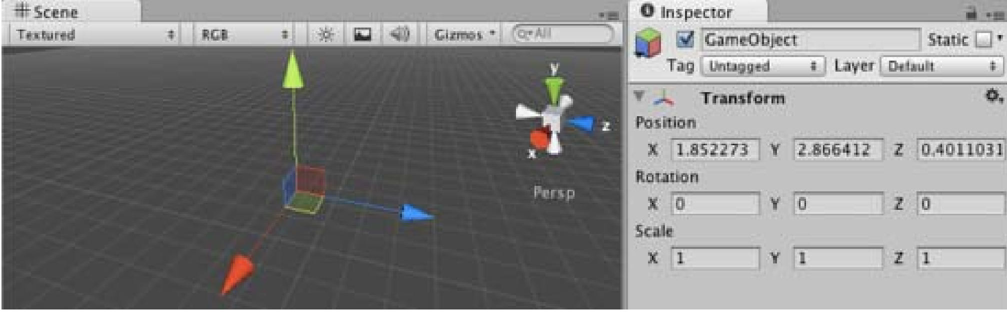
All three Gizmos can be directly edited in the Scene View.

When you click and drag on one of the three gizmo axes, you will notice that its color changes. As you drag the mouse, you will see the object translate, rotate, or scale along the selected axis. When you release the mouse button, the axis remains selected. You can click the middle mouse button and drag the mouse to manipulate the Transform along the selected axis.



*Any individual axis will become selected when you click on it*

Around the centre of the Transform gizmo are three coloured squares. These allow you to drag the Transform in a single plane (ie, the object will move in two axes but be held still in the third axis).

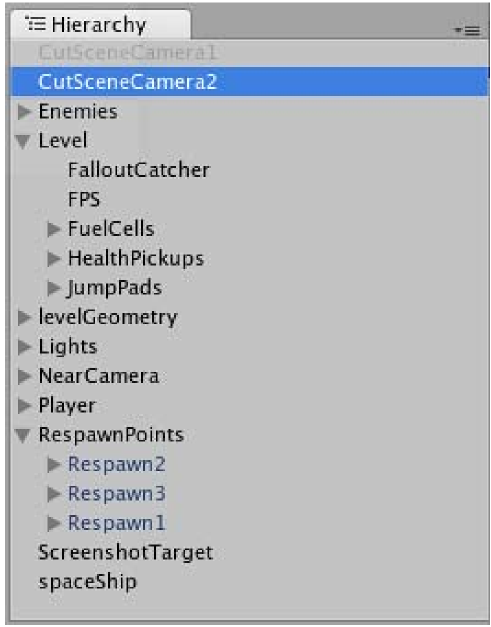


*Dragging in the XZ plane*

## Parenting

Parenting is one of the most important concepts to understand when using Unity. When a GameObject is a **Parent** of another GameObject, the **Child** GameObject will move, rotate, and scale exactly as its Parent does. Just like your arms are attached to your body, when you turn your body, your arms move because they're attached. Any object can have multiple children, but only one parent.

You can create a Parent by dragging any GameObject in the **Hierarchy View** onto another. This will create a Parent-Child relationship between the two GameObjects.



*Example of a Parent-Child hierarchy. GameObjects with foldout arrows to the left of their names are parents.*

In the above example, we say that the arms are parented to the body and the hands are parented to the arms. The scenes you make in Unity will contain collections of these **Transform hierarchies**. The topmost parent object is called the **Root object**. When you move, scale or rotate a parent, all the changes in its Transform are applied to its children as well.

It is worth pointing out that the Transform values in the Inspector of any Child GameObject are displayed relative to the Parent's Transform values. These are also called the **Local Coordinates**. Through scripting, you can access the **Global Coordinates** as well as the local coordinates.

You can build compound objects by parenting several separate objects together, for example, the skeletal structure of a human ragdoll. You can also achieve useful effects with simple hierarchies. For example, if you have a horror game that takes place at night, you can create an effective atmosphere with a flashlight. To create this object, you would parent a spotlight Transform to the flashlight Transform. Then, any alteration of the flashlight Transform will affect the spotlight, creating a convincing flashlight effect.

### Performance Issues and Limitations with Non-Uniform Scaling

Non-uniform scaling is when the **Scale** in a Transform has different values for x, y, and z; for example (2, 4, 2). In contrast, uniform scaling has the same value for x, y, and z; for example (3, 3, 3). Non-uniform scaling can be useful in a few select cases but should be avoided whenever possible.

Non-uniform scaling has a negative impact on rendering performance. In order to transform vertex normals correctly, we transform the mesh on the CPU and create an extra copy of the data. Normally we can keep the mesh shared between instances in graphics memory, but in this case you pay both a CPU and memory cost per instance.

There are also certain limitations in how Unity handles non-uniform scaling:

- Certain components do not fully support non-uniform scaling. For example, for components with a **radius** property or similar, such as a **Sphere Collider**, **Capsule Collider**, **Light**, **Audio Source** etc., the shape will never become elliptical but remain circular/spherical regardless of non-uniform scaling.
- A child object that has a non-uniformly scaled parent and is rotated relative to that parent may have a non-orthogonal matrix, meaning that it may appear skewed. Some components that do support simple non-uniform scaling still do not support non-orthogonal matrices. For example, a **Box Collider** cannot be skewed so if its transform is non-orthogonal, the Box Collider will not match the shape of the rendered mesh accurately.
- For performance reasons, a child object that has a non-uniformly scaled parent will not have its scale/matrix automatically updated while rotating. This may result in popping of the scale once the scale is updated, for example if the object is detached from its parent.

### Importance of Scale

The scale of the Transform determines the difference between the size of your mesh in your modeling application and the size of your mesh in Unity. The mesh's size in Unity (and therefore the Transform's scale) is **very** important, especially during physics simulation. There are three factors that can affect the scale of your object:

- The size of your mesh in your 3D modeling application.
- The **Mesh Scale Factor** setting in the object's **Import Settings**.
- The **Scale** values of your Transform Component.

Ideally, you should not adjust the **Scale** of your object in the Transform Component. The best option is to create your models at real-life scale so you won't have to change your Transform's scale. The next best option is to adjust the scale at which your mesh is imported in the **Import Settings** for your individual mesh. Certain optimizations occur based on the import size, and instantiating an object that has an adjusted scale value can decrease performance. For more information, see the section about optimizing scale on the Rigidbody component reference page.

### Hints

- When parenting Transforms, set the parent's location to <0,0,0> before adding the child. This will save you many headaches later.
- **Particle Systems** are not affected by the Transform's **Scale**. In order to scale a Particle System, you need to modify the properties in the System's Particle Emitter, Animator and Renderer.
- If you are using **Rigidbodies** for physics simulation, there is some important information about the Scale property on the Rigidbody component reference page.
- You can change the colors of the Transform axes (and other UI elements) from the preferences (**Menu: Unity > Preferences** and then select the **Colors & keys** panel).

- It is best to avoid scaling within Unity if possible. Try to have the scales of your object finalized in your 3D modeling application, or in the **Import Settings** of your mesh.

Page last updated: 2012-01-18

---

# comp-UnityGUIGroup

**UnityGUI** is the GUI creation system built into Unity. It consists of creating different **Controls**, and defining the content and appearance of those controls.

The **Components** of UnityGUI allow you to define the appearance of Controls.

- GUI Skin
- GUI Style

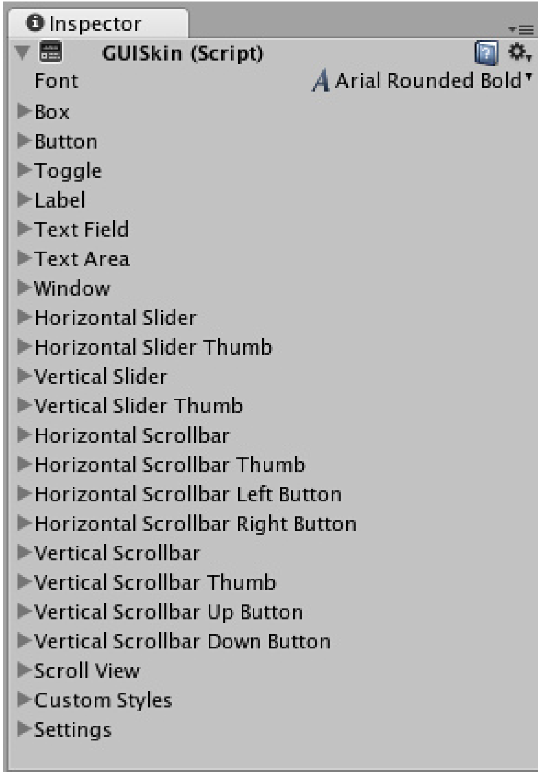
For information about using UnityGUI to create Controls and define their content, please read the GUI Scripting Guide.

Page last updated: 2007-08-22

---

# class-GUISkin

**GUISkins** are a collection of **GUIStyles** that can be applied to your GUI. Each **Control** type has its own Style definition. Skins are intended to allow you to apply style to an entire UI, instead of a single Control by itself.




*A GUI Skin as seen in the **Inspector***

To create a GUISkin, select **Assets->Create->GUI Skin** from the menubar.

GUISkins are part of the **UnityGUI** system. For more detailed information about UnityGUI, please take a look at the GUI Scripting Guide.

### Properties

All of the properties within a GUI Skin are an individual GUIStyle. Please read the GUIStyle page for more information about how to use Styles.

| | |
|---|---|
| **Font** | The global Font to use for every Control in the GUI |
| **Box** | The Style to use for all Boxes |
| **Button** | The Style to use for all Buttons |
| **Toggle** | The Style to use for all Toggles |
| **Label** | The Style to use for all Labels |
| **Text Field** | The Style to use for all Text Fields |
| **Text Area** | The Style to use for all Text Areas |
| **Window** | The Style to use for all Windows |
| **Horizontal Slider** | The Style to use for all Horizontal Slider bars |
| **Horizontal Slider Thumb** | The Style to use for all Horizontal Slider Thumb Buttons |
| **Vertical Slider** | The Style to use for all Vertical Slider bars |
| **Vertical Slider Thumb** | The Style to use for all Vertical Slider Thumb Buttons |
| **Horizontal Scrollbar** | The Style to use for all Horizontal Scrollbars |
| **Horizontal Scrollbar Thumb** | The Style to use for all Horizontal Scrollbar Thumb Buttons |
| **Horizontal Scrollbar Left Button** | The Style to use for all Horizontal Scrollbar scroll Left Buttons |
| **Horizontal Scrollbar Right Button** | The Style to use for all Horizontal Scrollbar scroll Right Buttons |
| **Vertical Scrollbar** | The Style to use for all Vertical Scrollbars |
| **Vertical Scrollbar Thumb** | The Style to use for all Vertical Scrollbar Thumb Buttons |
| **Vertical Scrollbar Up Button** | The Style to use for all Vertical Scrollbar scroll Up Buttons |
| **Vertical Scrollbar Down Button** | The Style to use for all Vertical Scrollbar scroll Down Buttons |
| **Custom 1-20** | Additional custom Styles that can be applied to any Control |
| **Custom Styles** | An array of additional custom Styles that can be applied to any Control |
| **Settings** | Additional Settings for the entire GUI |
| **Double Click Selects Word** | If enabled, double-clicking a word will select it |
| **Triple Click Selects Line** | If enabled, triple-clicking a word will select the entire line |
| **Cursor Color** | Color of the keyboard cursor |
| **Cursor Flash Speed** | The speed at which the text cursor will flash when editing any Text Control |
| **Selection Color** | Color of the selected area of Text |

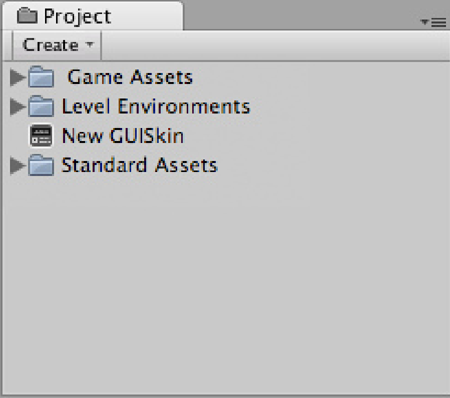
## Details

When you are creating an entire GUI for your game, you will likely need to do a lot of customization for every different Control type. In many different game genres, like real-time strategy or role-playing, there is a need for practically every single Control type.

Because each individual Control uses a particular Style, it does not make sense to create a dozen-plus individual Styles and assign them all manually. GUI Skins take care of this problem for you. By creating a GUI Skin, you have a pre-defined collection of Styles for every individual Control. You then apply the Skin with a single line of code, which eliminates the need to manually specify the Style of each individual Control.

### Creating GUISkins

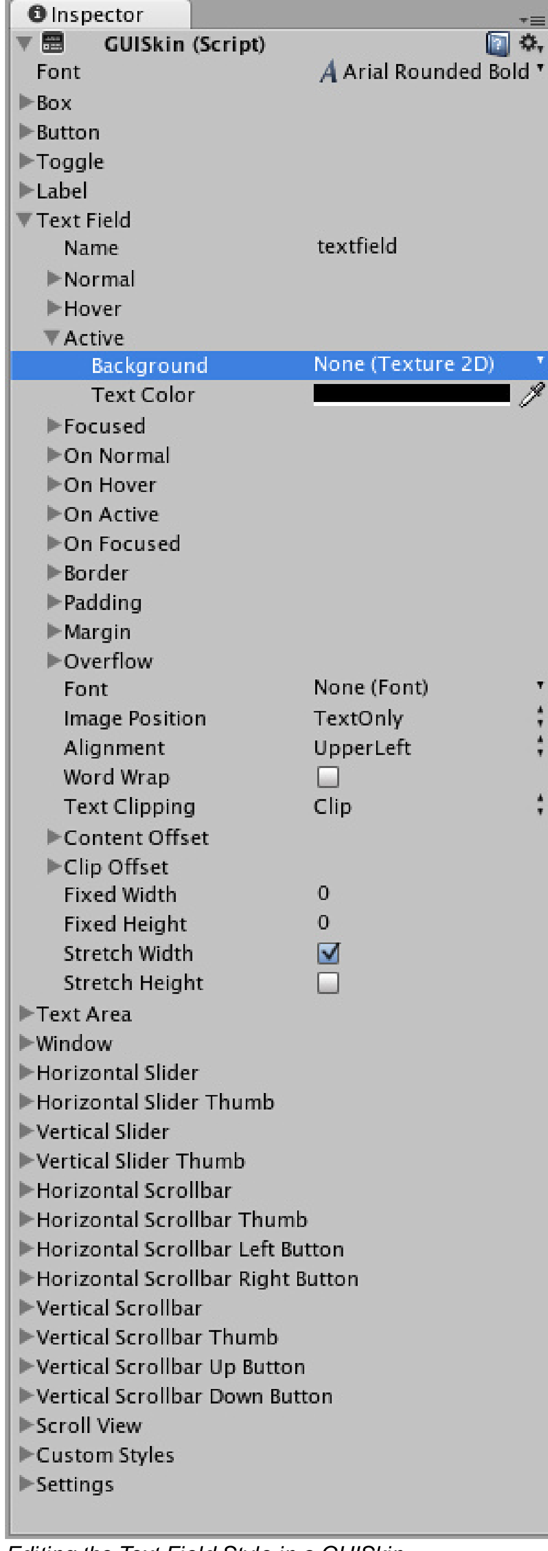
GUISkins are asset files. To create a GUI Skin, select **Assets->Create->GUI Skin** from the menubar. This will put a new GUISkin in your **Project View**.



*A new GUISkin file in the Project View*

**Editing GUISkins**

After you have created a GUISkin, you can edit all of the Styles it contains in the Inspector. For example, the **Text Field** Style will be applied to all Text Field Controls.



*Editing the Text Field Style in a GUISkin*

No matter how many Text Fields you create in your script, they will all use this Style. Of course, you have control over changing the styles of one Text Field over the other if you wish. We'll discuss how that is done next.

**Applying GUISkins**

To apply a GUISkin to your GUI, you must use a simple script to read and apply the Skin to your Controls.

```
// Create a public variable where we can assign the GUISkin
var customSkin : GUISkin;
```

```
// Apply the Skin in our OnGUI() function
function OnGUI () {
	GUI.skin = customSkin;

	// Now create any Controls you like, and they will be displayed with the custom Skin
	GUILayout.Button ("I am a re-Skinned Button");

	// You can change or remove the skin for some Controls but not others
	GUI.skin = null;

	// Any Controls created here will use the default Skin and not the custom Skin
	GUILayout.Button ("This Button uses the default UnityGUI Skin");
}
```

In some cases you want to have two of the same Control with different Styles. For this, it does not make sense to create a new Skin and re-assign it. Instead, you use one of the **Custom** Styles in the skin. Provide a **Name** for the custom Style, and you can use that name as the last argument of the individual Control.

```
// One of the custom Styles in this Skin has the name "MyCustomControl"
var customSkin : GUISkin;

function OnGUI () {
	GUI.skin = customSkin;

	// We provide the name of the Style we want to use as the last argument of the Control function
	GUILayout.Button ("I am a custom styled Button", "MyCustomControl");

	// We can also ignore the Custom Style, and use the Skin's default Button Style
	GUILayout.Button ("I am the Skin's Button Style");
}
```

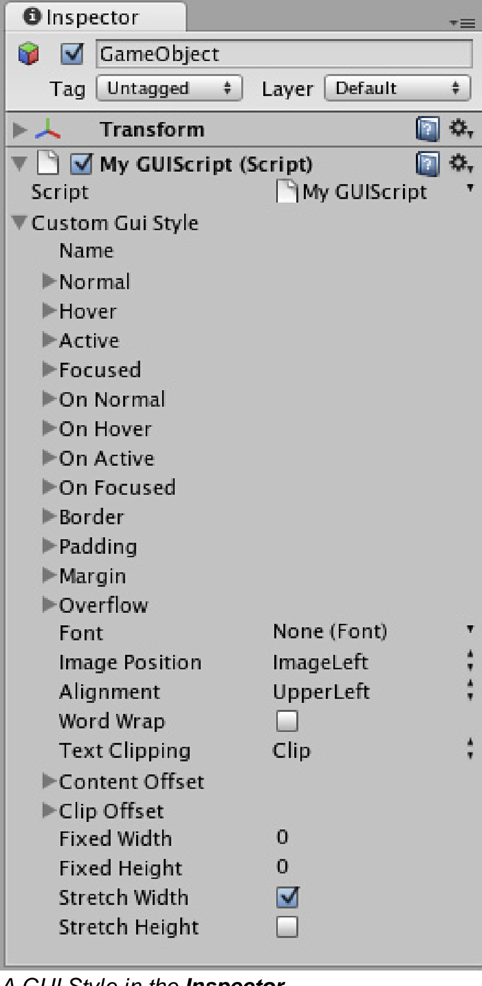
For more information about working with GUIStyles, please read the GUIStyle page. For more information about using UnityGUI, please read the GUI Scripting Guide.

Page last updated: 2007-10-01

---

## class-GUIStyle

**GUI Styles** are a collection of custom attributes for use with **UnityGUI**. A single GUI Style defines the appearance of a single UnityGUI **Control**.




*A GUI Style in the **Inspector***

If you want to add style to more than one control, use a GUI Skin instead of a GUI Style. For more information about UnityGUI, please read the GUI Scripting Guide.

## Properties

| | |
|---|---|
| **Name** | The text string that can be used to refer to this specific Style |
| **Normal** | Background image & Text Color of the Control in default state |
| **Hover** | Background image & Text Color when the mouse is positioned over the Control |
| **Active** | Background image & Text Color when the mouse is actively clicking the Control |
| **Focused** | Background image & Text Color when the Control has keyboard focus |
| **On Normal** | Background image & Text Color of the Control in enabled state |
| **On Hover** | Background image & Text Color when the mouse is positioned over the enabled Control |
| **On Active** | Properties when the mouse is actively clicking the enabled Control |
| **On Focused** | Background image & Text Color when the enabled Control has keyboard focus |
| **Border** | Number of pixels on each side of the **Background** image that are not affected by the scale of the Control' shape |
| **Padding** | Space in pixels from each edge of the Control to the start of its contents. |
| **Margin** | The margins between elements rendered in this style and any other GUI Controls. |
| **Overflow** | Extra space to be added to the background image. |
| **Font** | The Font used for all text in this style |
| **Image Position** | The way the background image and text are combined. |
| **Alignment** | Standard text alignment options. |
| **Word Wrap** | If enabled, text that reaches the boundaries of the Control will wrap around to the next line |
| **Text Clipping** | If **Word Wrap** is enabled, choose how to handle text that exceeds the boundaries of the Control |
| **Overflow** | Any text that exceeds the Control boundaries will continue beyond the boundaries |
| **Clip** | Any text that exceeds the Control boundaries will be hidden |
| **Content Offset** | Number of pixels along X and Y axes that the Content will be displaced in addition to all other properties |
| **X** | Left/Right Offset |

| | |
|---|---|
| **Y** | Up/Down Offset |
| **Fixed Width** | Number of pixels for the width of the Control, which will override any provided **Rect()** value |
| **Fixed Height** | Number of pixels for the height of the Control, which will override any provided **Rect()** value |
| **Stretch Width** | If enabled, Controls using this style can be stretched horizontally for a better layout. |
| **Stretch Height** | If enabled, Controls using this style can be stretched vertically for a better layout. |

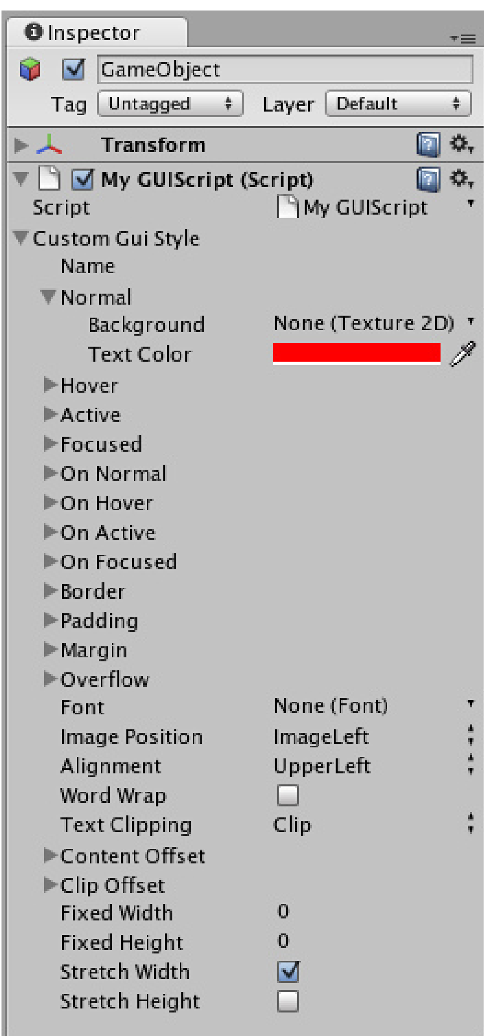
## Details

GUIStyles are declared from scripts and modified on a per-instance basis. If you want to use a single or few Controls with a custom Style, you can declare this custom Style in the script and provide the Style as an argument of the Control function. This will make these Controls appear with the Style that you define.

First, you must declare a GUI Style from within a script.

```
/* Declare a GUI Style */
var customGuiStyle : GUIStyle;

...
```

When you attach this script to a GameObject, you will see the custom Style available to modify in the **Inspector**.

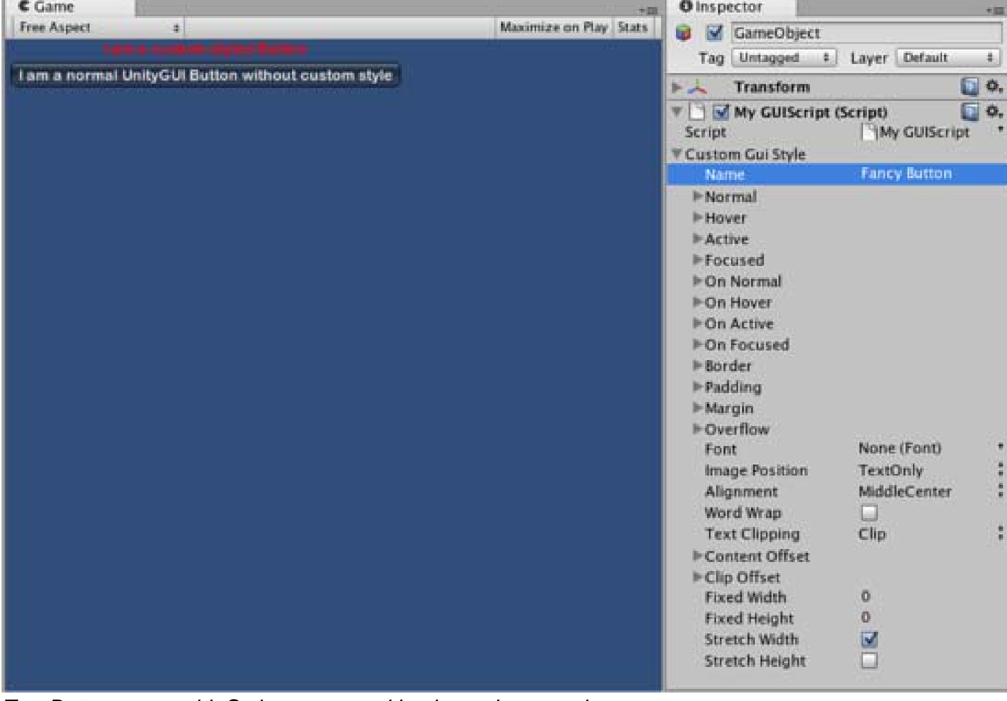



*A Style declared in a script can be modified in each instance of the script*

Now, when you want to tell a particular Control to use this Style, you provide the name of the Style as the last argument in the Control function.

```
...
```

```
function OnGUI () {
	// Provide the name of the Style as the final argument to use it
	GUILayout.Button ("I am a custom-styled Button", customGuiStyle);

	// If you do not want to apply the Style, do not provide the name
	GUILayout.Button ("I am a normal UnityGUI Button without custom style");
}
```




*Two Buttons, one with Style, as created by the code example*

For more information about using UnityGUI, please read the GUI Scripting Guide.

Page last updated: 2007-10-01

---

## comp-Wizards

- Ragdoll Wizard

Page last updated: 2007-07-16

---

## wizard-RagdollWizard

Unity has a simple wizard that lets you quickly create your own ragdoll. You simply have to drag the different limbs on the respective properties in the wizard. Then select create and Unity will automatically generate all **Colliders**, **Rigidbodies** and **Joints** that make up the Ragdoll for you.

### Creating the Character

Ragdolls make use of **Skinned Meshes**, that is a character mesh rigged up with bones in the 3D modeling application. For this reason, you must build ragdoll characters in a 3D package like Maya or Cinema4D.

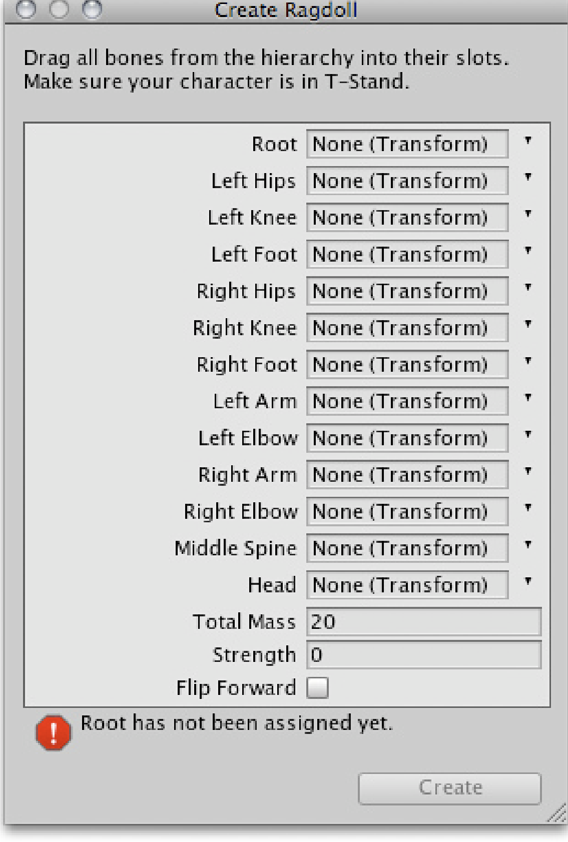
When you've created you character and rigged it, save the asset normally in your **Project Folder**. When you switch to Unity, you'll see the character asset file. Select that file and the **Import Settings** dialog will appear inside the inspector. Make sure that **Mesh Colliders** is not enabled.

### Using the Wizard

It's not possible to make the actual source asset into a ragdoll. This would require modifying the source asset file, and is therefore impossible. You will make an instance of the character asset into a ragdoll, which can then be saved as a **Prefab** for re-use.

Create an instance of the character by dragging it from the **Project View** to the **Hierarchy View**. Expand its **Transform Hierarchy** by clicking the small arrow to the left of the instance's name in the Hierarchy. Now you are ready to start assigning your ragdoll parts.

Open the Ragdoll Wizard by choosing **GameObject->Create Other->Ragdoll** from the menu bar. You will now see the Wizard itself.




*The Ragdoll Wizard*

Assigning parts to the wizard should be self-explanatory. Drag the different Transforms of your character instance to the appropriate property on the wizard. This should be especially easy if you created the character asset yourself.

When you are done, click the **Create Button**. Now when you enter **Play Mode**, you will see your character go limp as a ragdoll.

The final step is to save the setup ragdoll as a Prefab. Choose **Assets->Create->Prefab** from the menu bar. You will see a New Prefab appear in the Project View. Rename it to "Ragdoll Prefab". Drag the ragdoll character instance from the Hierarchy on top of the "Ragdoll Prefab". You now have a completely set-up, re-usable ragdoll character to use as much as you like in your game.

Page last updated: 2012-08-09

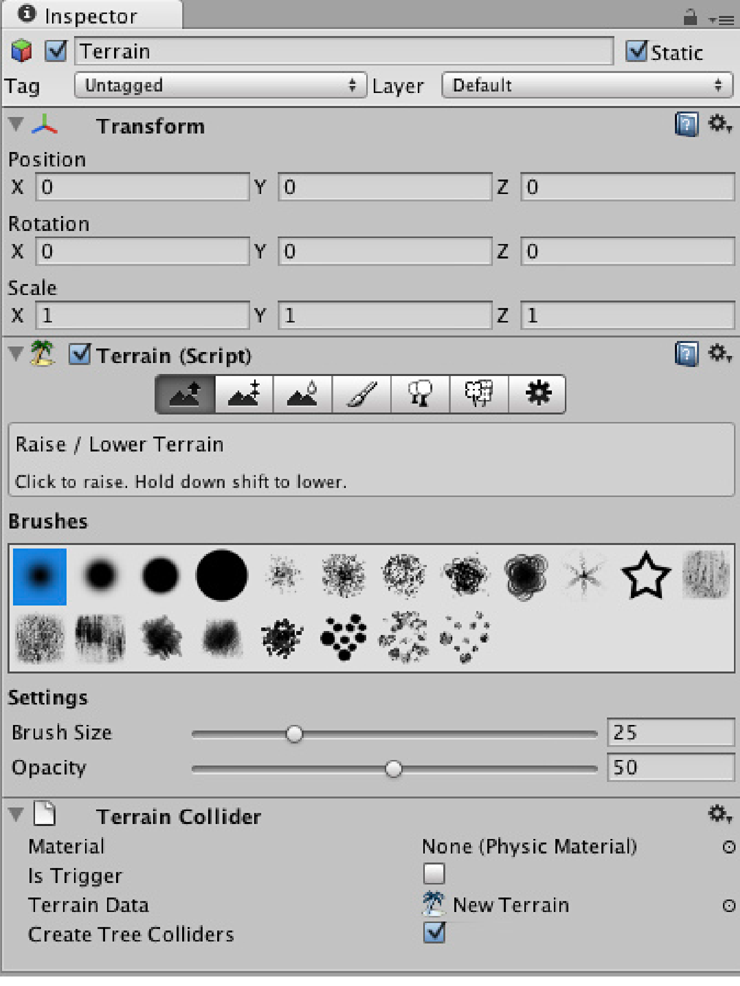
---

# script-Terrain

This section will explain how to use the **Terrain Engine**. It will cover creation, technical details, and other considerations. It is broken into the following sections:

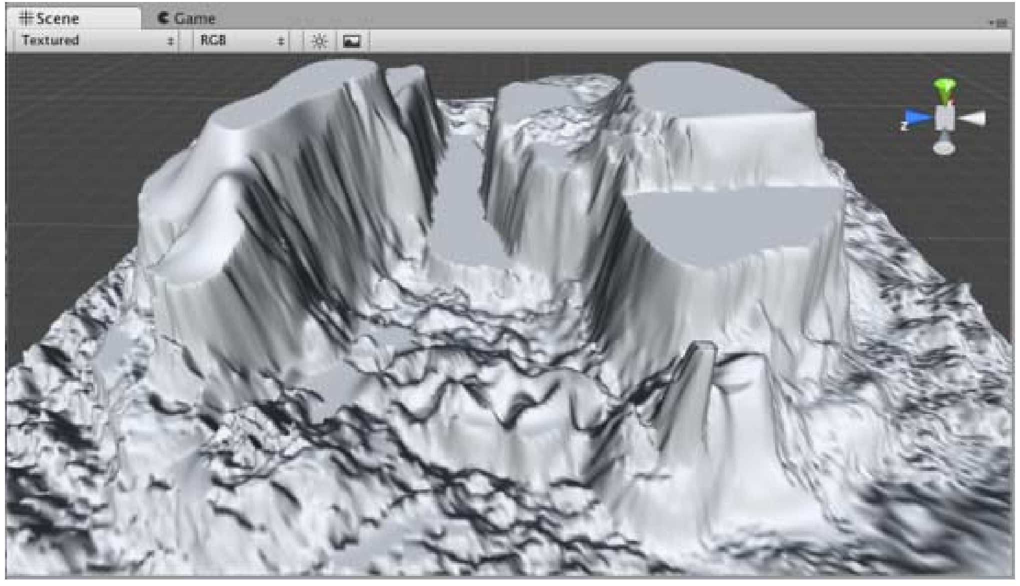
## Using Terrains

This section covers the most basic information about using Terrains. This includes creating Terrains and how to use the new Terrain tools & brushes.




## Height

This section explains how to use the different tools and brushes that alter the Height of the Terrain.

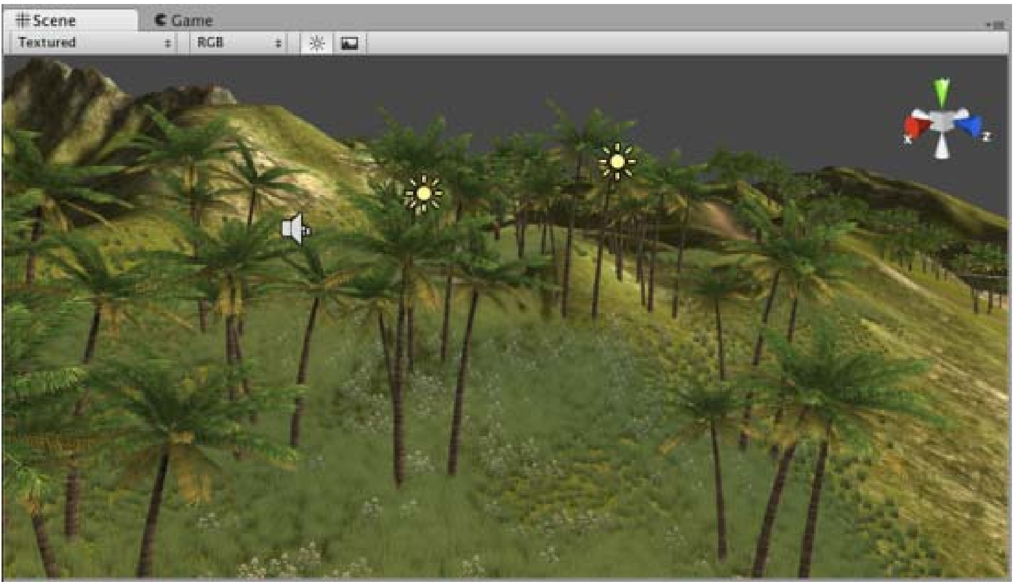



## Terrain Textures

This section explains how to add, paint and blend Terrain Textures using different brushes.

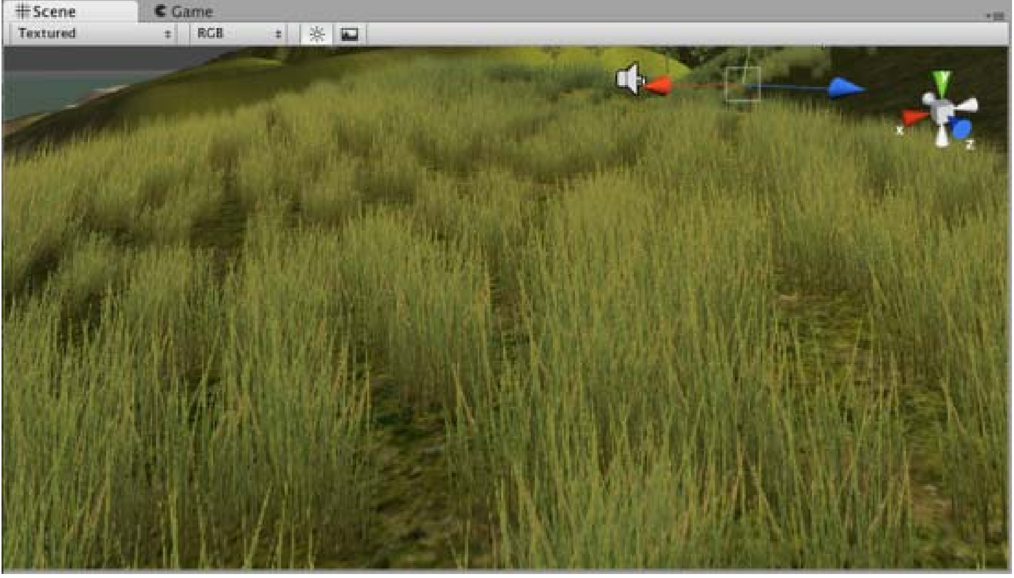
## Trees

This section contains important information for creating your own tree assets. It also covers adding and painting trees on your Terrain.




## Grass

This section explains how grass works, and how to use it.




## Detail Meshes

This section explains practical usage for detail meshes like rocks, haystacks, vegetation, etc.

## Lightmaps

You can lightmap terrains just like any other objects using Unity's built-in lightmapper. See Lightmapping Quickstart for help.

### Other Settings

This section covers all the other settings associated with Terrains.

### Mobile performance note

Rendering terrain is quite expensive, so terrain engine is not very practical on lower-end mobile devices.

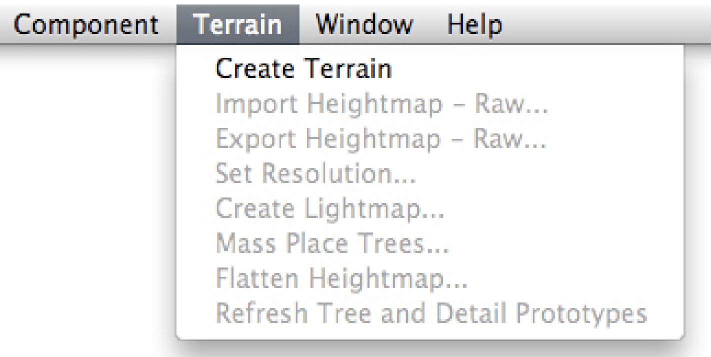
Page last updated: 2012-08-17

---

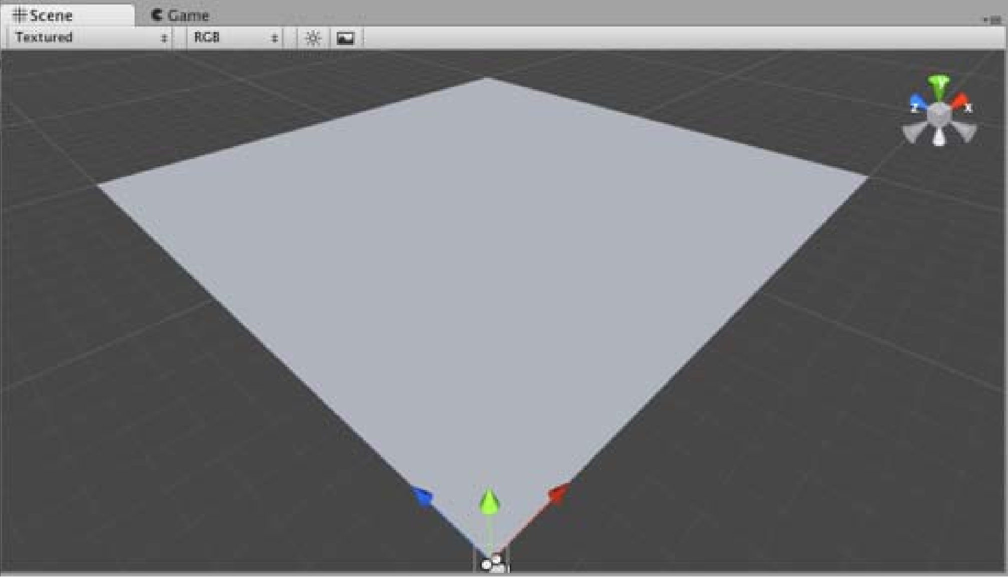
## terrain-UsingTerrains

### Creating a new Terrain

A new **Terrain** can be created from **Terrain->Create Terrain**. This will add a Terrain to your **Project** and **Hierarchy Views**.




Your new Terrain will look like this in the **Scene View**:




*A new Terrain in Scene View*

If you would like a differently sized Terrain, choose **Terrain->Set Resolution** from the menu bar. There are a number of settings that related to Terrain size which you can change from this dialog.

Set Heightmap resolution

Please note that modifying the resolution will clear the heightmap, detail map or splatmap.

| | |
|---|---|
| Terrain Width | 2000 |
| Terrain Height | 600 |
| Terrain Length | 2000 |
| Heightmap Resolution | 513 |
| Detail Resolution | 1024 |
| Detail Resolution Per Patch | 16 |
| Control Texture Resolution | 512 |
| Base Texture Resolution | 1024 |

Set Resolution

*Setting the resolution of your terrain.*

On the above image, there are values that can be changed.
These values are:

- Terrain Width: The width of the Terrain in units.
- Terrain Height: The height of the Terrain in units.
- Terrain Length: The length of the Terrain in units.
- HeightMap Resolution: The HeightMap resolution for the selected Terrain.
- Detail Resolution: The resolution of the map that controls grass and detail meshes. For performance reasons (to save on draw calls) the lower you set this number the better.
- Control Texture Resolution: The resolution of the splat map used to layer the different textures painted onto the Terrain.
- Base Texture Resolution: The resolution of the composite texture that is used in place of the splat map at certain distances.

### Navigating the Terrain

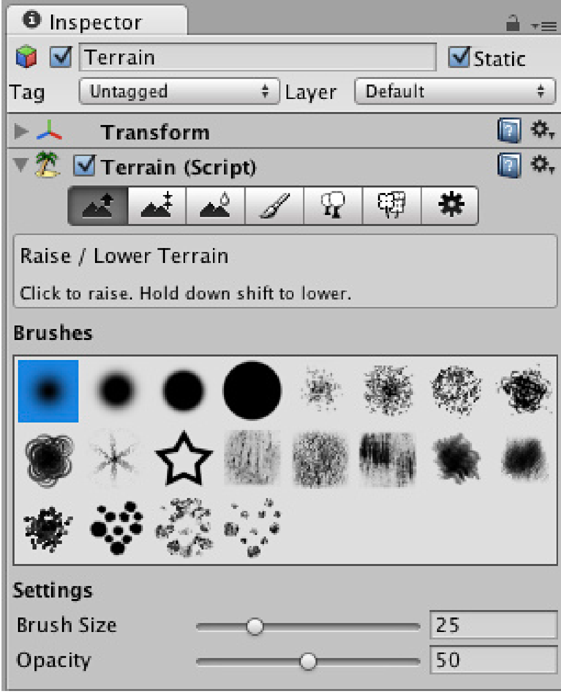
Terrains work a bit differently than other GameObjects. You can use **Brushes** to paint and manipulate your Terrain. If you want to reposition a Terrain, you can modify its **Transform Position** values in the **Inspector**. This allows you to move your Terrain around, but you cannot rotate or scale it.

While your Terrain is selected in the Hierarchy, you can gracefully navigate the terrain with the F (focus) key. When you press F, wherever your mouse is positioned will be moved to the center of the Scene View. This allows you to touch up an area, and quickly zoom over to a different area and change something else. If your mouse is not hovering over an area of the Terrain

when you press the F key, the entire Terrain will be centered in your Scene View.

### Editing the Terrain

With the Terrain selected, you can look at the **Inspector** to see some incredible new Terrain editing tools.




*Terrain Editing Tools appear in the Inspector*

Each rectangular button is a different Terrain tool. There are tools to change the height, paint splat maps, or attach details like trees or rocks. To use a specific tool, click on it. You will then see a short description of the tool appear in text below the tool buttons.

Most of the tools make use of a brush. Many different brushes are displayed for any tool that uses a brush. To select a brush, just click on it. The currently selected brush will display a preview when you hover the mouse over the terrain, at the size you have specified.

You will use all of these brushes in the **Scene View** to paint directly onto your Terrain. Simply choose the tool and brush you want, then click & drag on the Terrain to alter it in real-time. To paint height, textures, or decorations, you must have the Terrain selected in the **Hierarchy View**.

**Note:** When you have a brush selected, move your mouse over the Terrain in the Scene View and press **F**. This will center the Scene View over the mouse pointer position and automatically zoom in to the **Brush Size** distance. This is the quickest & easiest way to navigate around your Terrain while creating it.

#### Terrain Keyboard Shortcuts

While Terrain Inspector is active, those keyboard shortcuts can be used for fast editing (all of them customizable in Unity Preferences):

- Shift-Q to Shift-Y selects active terrain tool.
- Comma (,) and dot (.) cycle through active brush.
- Shift-comma (<) and Shift-dot (>) cycle through active tree/texture/detail object.

Page last updated: 2011-11-03

---

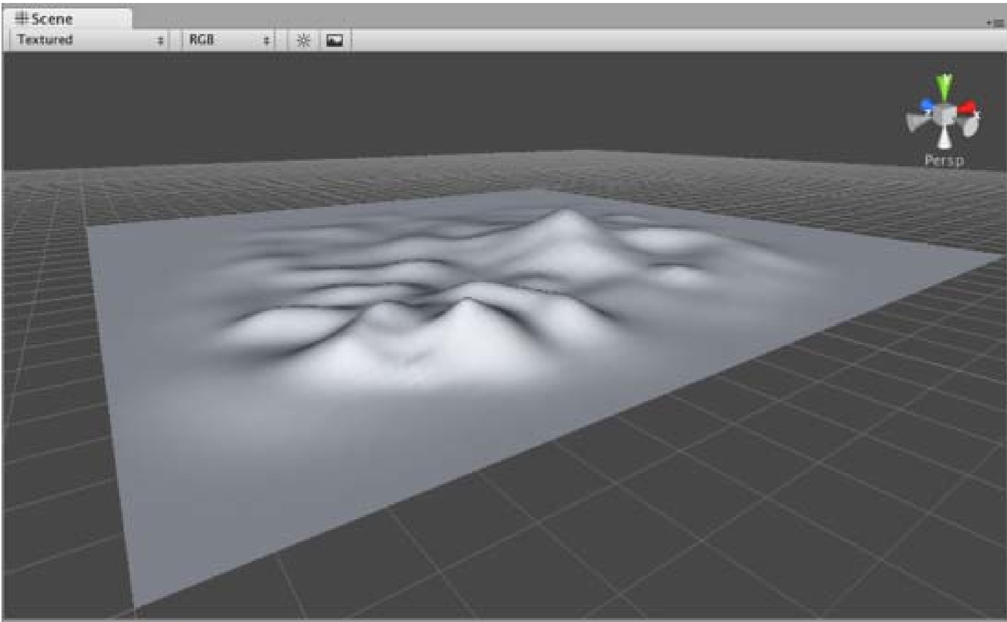
## terrain-Height

Using any of the Terrain editing tools is very simple. You will literally paint the Terrain from within the **Scene View**. For the height tools and all others, you just have to select the tool, and click the Terrain in Scene View to manipulate it in real-time.

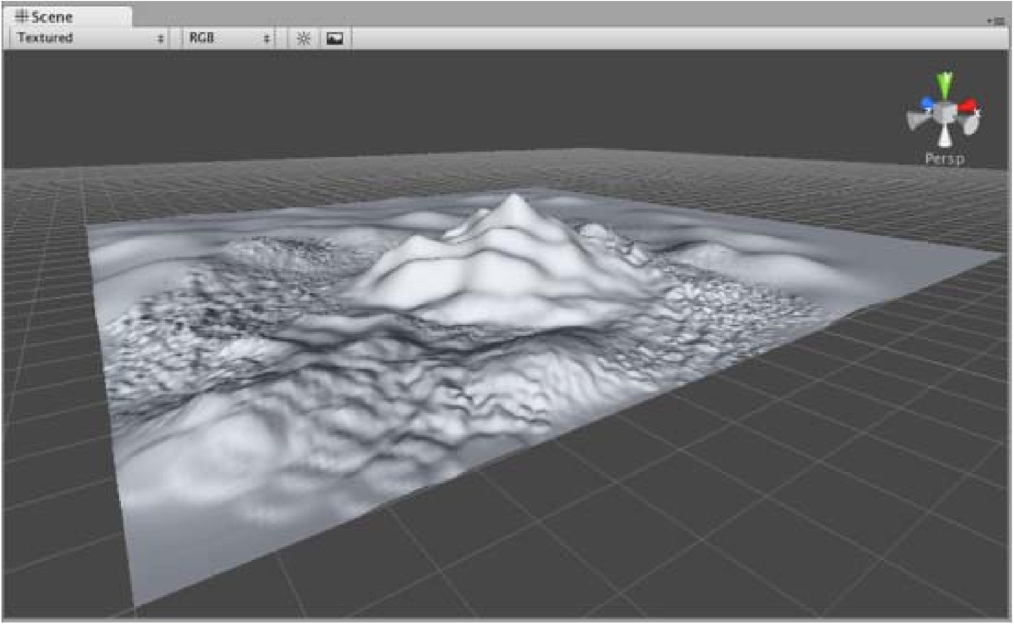
### Raising & Lowering Height

The first tool on the left is the **Raise Height** tool .

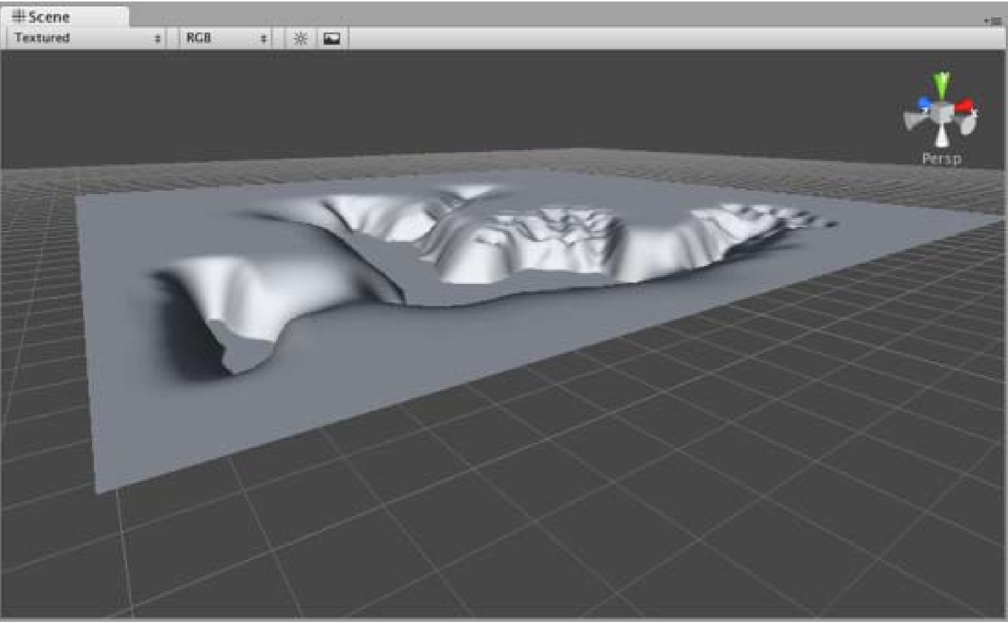
With this tool, you paint brush strokes that will raise the height of the **Terrain**. Clicking the mouse once will increment the height. Keeping the mouse button depressed and moving the mouse will continually raise the height until the maximum height is reached.




You can use any of the brushes to achieve different results




If you want to lower the height when you click, hold the **Shift** key.




**Note:** When you have a brush selected, move your mouse over the Terrain in the Scene View and press **F**. This will center the Scene View over the mouse pointer position and automatically zoom in to the **Brush Size** distance. This is the quickest & easiest way to navigate around your Terrain while creating it.

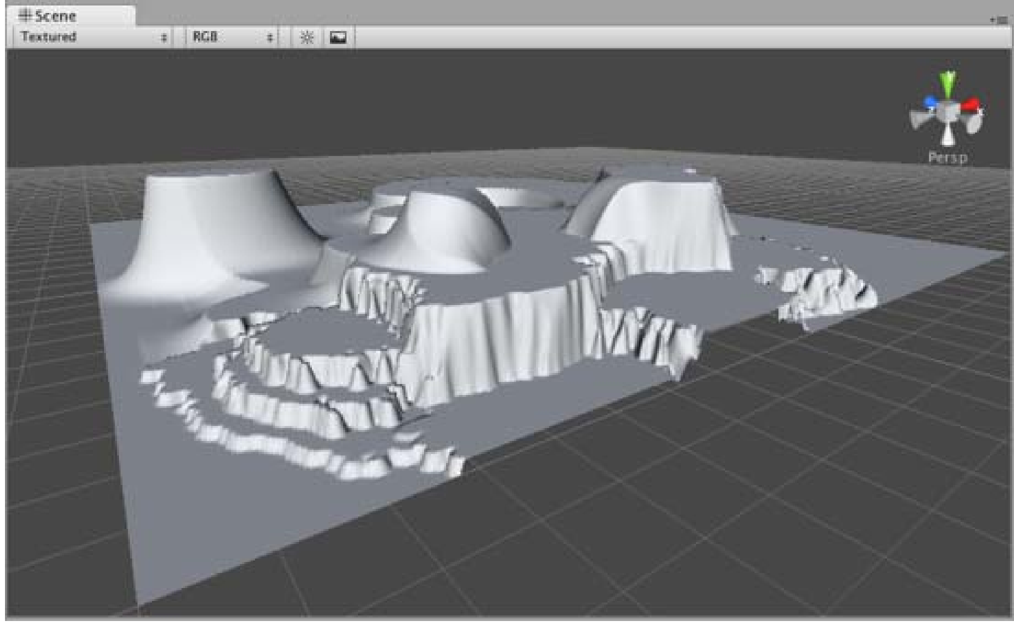
**Paint Height**

The second tool from the left is the **Paint Height** tool .

This tool allows you to specify a target height, and move any part of the terrain toward that height. Once the terrain reaches the target height, it will stop moving and rest at that height.

To specify the target height, hold **Shift** and click on the terrain at the height you desire. You can also manually adjust the **Height** slider in the Inspector.

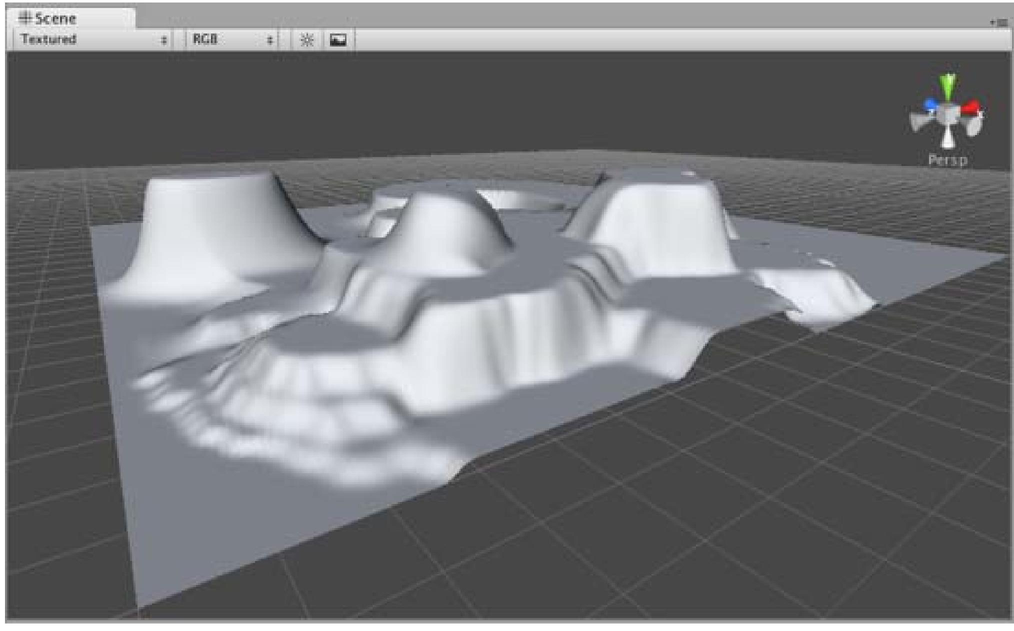
Now you've specified the target height, and any clicks you make on the terrain will move the terrain up or down to reach that height.




**Smoothing Height**

The third tool from the left is the **Smoothing Height** tool .

This tool allows you to soften any height differences within the area you're painting. Like the other brushes, paint the areas you want to smooth in the Scene View.




### Working with Heightmaps

If you like, you can import a greyscale Heightmap created in Photoshop, or from real-world geography data and apply it to your Terrain. To do this, choose **Terrain->Import Heightmap - Raw...**, then select the desired RAW file. You'll then see some import settings. These will be set for you, but you have the option of changing the size of your Terrain from this dialog if you like. When you're ready, click the **Import** button. Once the Heightmap has been applied to the Terrain, you can edit it normally with all the Tools described above. Note that the Unity Heightmap importer can only import grayscale raw files. Thus you can't create a raw heightmap using RGB channels, you must use grayscale.

Unity works with RAW files which make use of full 16-bit resolution. Any other heightmap editing application like Bryce, Terragen, or Photoshop can work with a Unity Heightmap at full resolution.

You also have the option of exporting your Heightmap to RAW format. Choose **Terrain->Export Heightmap - Raw...** and you'll see a export settings dialog. Make any changes you like, and click **Export** to save your new Heightmap.

Unity also provides an easy way to flatten your terrain. Choose **Terrain->Flatten...**. This lets you flatten your terrain to a height you specify in the wizard.

Page last updated: 2011-10-29

---

## terrain-Textures

Decorate the landscape of your terrain by tiling **Terrain Textures** across the entire terrain. You can blend and combine Terrain Textures to make smooth transitions from one map to another, or to keep the surroundings varied.

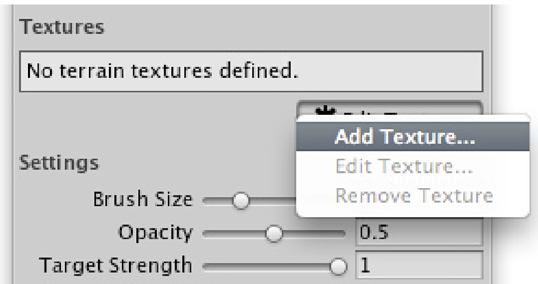
Terrain Textures are also called splat maps. What this means is you can define several repeating high resolution textures and blend between them arbitrarily, using alpha maps which you paint directly onto the Terrain. Because the textures are not large compared to the size of the terrain, the distribution size of the Textures is very small.

**Note:** Using an amount of textures in a multiple of four provides the greatest benefit for performance and storage of the Terrain alpha maps.

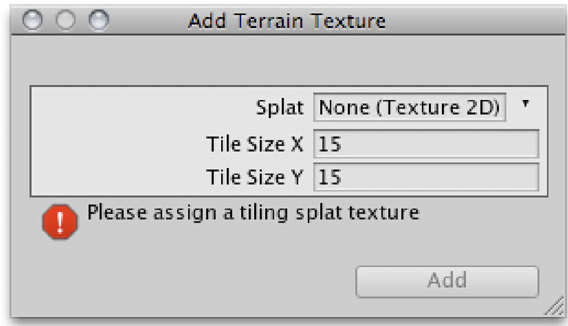
To being working with textures, click on the **Paint Textures** button in the Inspector.

### Adding a Terrain Texture

Before you can begin painting Terrain Textures, you will add at least one to the Terrain from your Project folder. Click the **Options Button->Add Texture...**.




This will bring up the Add Terrain Texture dialog.

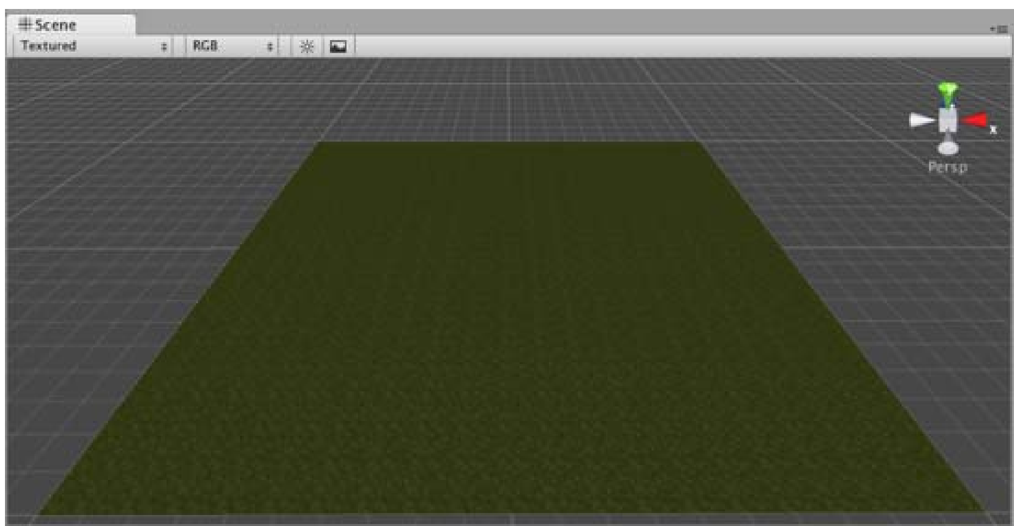



*The Add Terrain Texture dialog*

From here, select a tileable texture in the **Splat** property. You can either drag a texture to the property from the Project View, or choose one from the drop-down.

Now, set the **Tile Size X** and **Tile Size Y** properties. The larger the number, the larger each "tile" of the texture will be scaled. Textures with large **Tile Sizes** will be repeated fewer times across the entire Terrain. Smaller numbers will repeat the texture more often with smaller tiles.

Click the **Add** Button and you'll see your first Terrain Texture tile across the entire Terrain.




Repeat this process for as many Terrain Textures as you like.

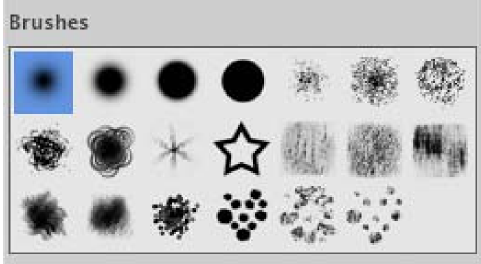
### Painting Terrain Textures

Once you've added at least two Terrain Textures, you can blend them together in various ways. This part gets really fun, so

let's jump right to the good stuff.

Select the Terrain Texture you want to use. The currently selected Terrain Texture will be highlighted in blue.

Select the Brush you want to use. The currently selected Brush will be highlighted in blue.

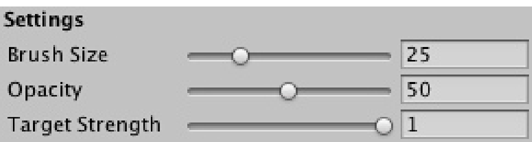



Select the Brush **Size**, **Opacity**, and **Target Strength**.

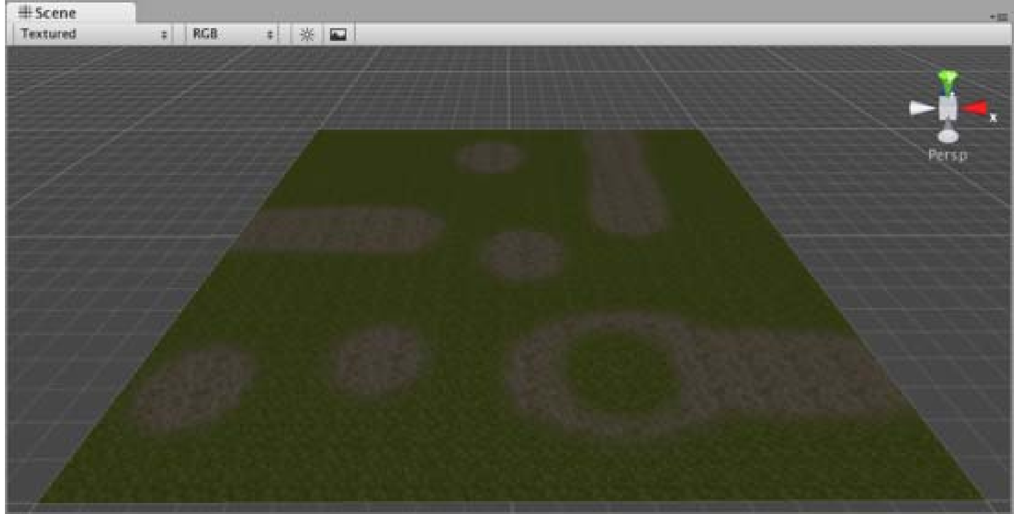
**Size** refers to the width of the brush relative to your terrain grid squares

**Opacity** is the transparency or amount of texture applied for a given amount of time you paint

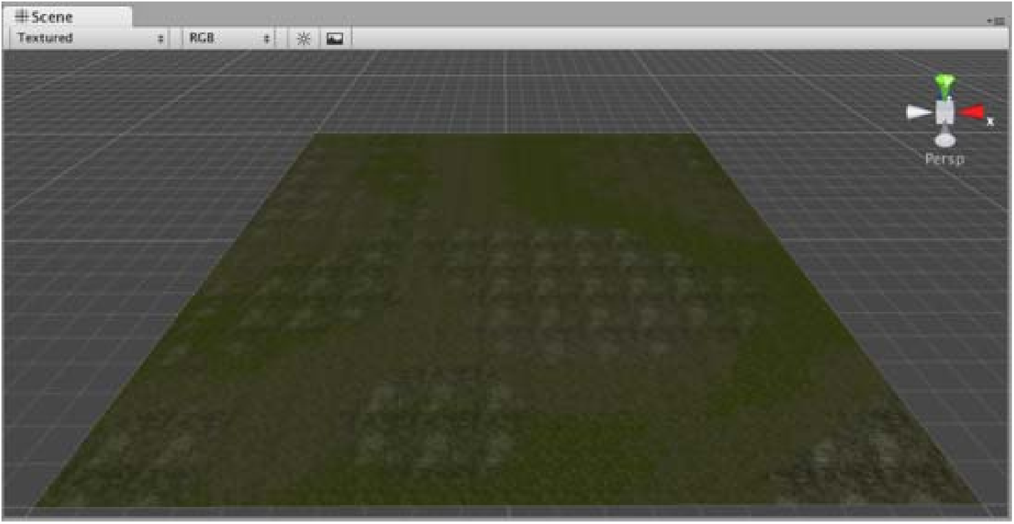
**Target Strength** is the maximum opacity you can reach by painting continuously.




Click and drag on the terrain to draw the Terrain Texture.




Use a variety of Textures, Brushes, Sizes, and Opacities to create a great variety of blended styles.



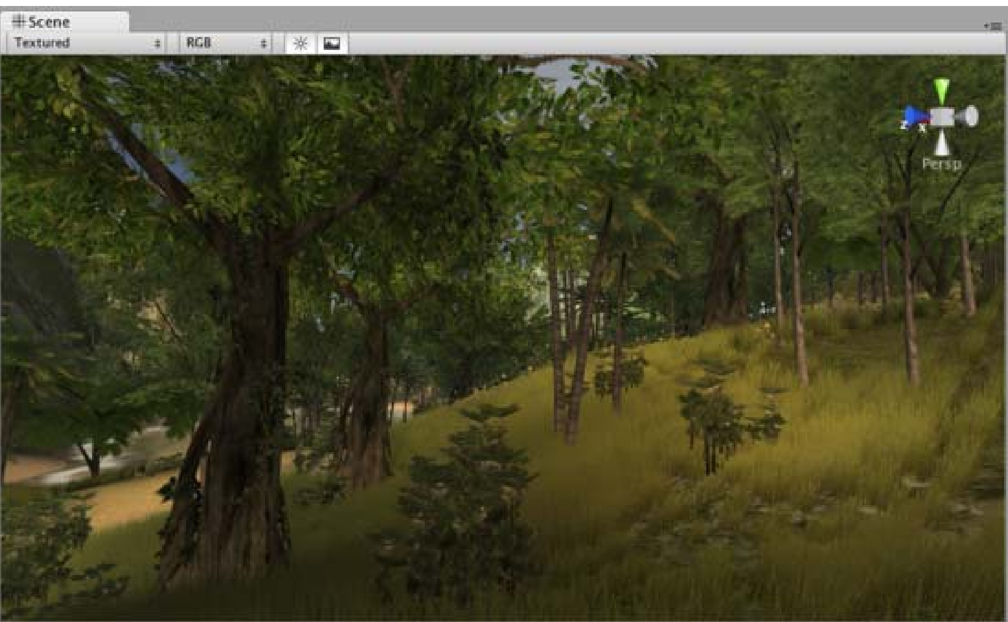

**Note:** When you have a brush selected, move your mouse over the Terrain in the Scene View and press **F**. This will center the Scene View over the mouse pointer position and automatically zoom in to the **Brush Size** distance. This is the quickest & easiest way to navigate around your Terrain while creating it.

Page last updated: 2012-08-17

---

## terrain-Trees

Unity's Terrain Engine has special support for Trees. You can put thousands of trees onto a Terrain, and render them in-game with a practical frame rate. This works by rendering trees near the camera in full 3D, and transitioning far-away trees to 2D billboards. Billboards in the distance will automatically update to orient themselves correctly as they are viewed from different angles. This transition system makes a detailed tree environment very simple for performance. You have complete control over tweaking the parameters of the mesh-to-billboard transition so you can get the best performance you need.




*You can easily paint lots of trees for beautiful environments like this*

### Adding Trees

Select the **Place Trees** button in the **Inspector**.

Before you can place trees on your terrain, you have to add them to the library of available trees. To do this, click the **Edit Trees button->Add Tree**. You'll see the **Add Tree** dialog appear.

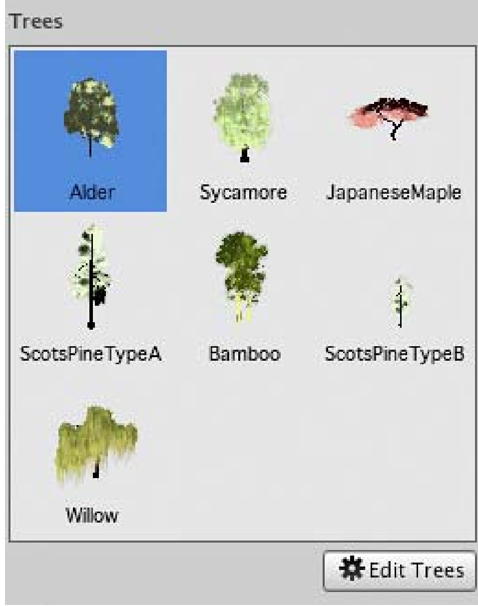
Add Tree
Tree None (Game Object)
Bend Factor 0
Please assign a tree
Add

*The Add Tree dialog*

Select the tree from your **Project View** and drag it to the **Tree** variable. You can also edit the **Bend Factor** if you want to add an additional bit of animated "bending in the wind" effect to the trees. When you're ready, click **Add**. The tree will now appear selected in the Inspector.

*The newly added tree appears selected in the Inspector*

You can add as many trees as you like. Each one will be selectable in the Inspector for you to place on your Terrain.




*The currently selected tree will always be highlighted in blue*

### Painting Trees

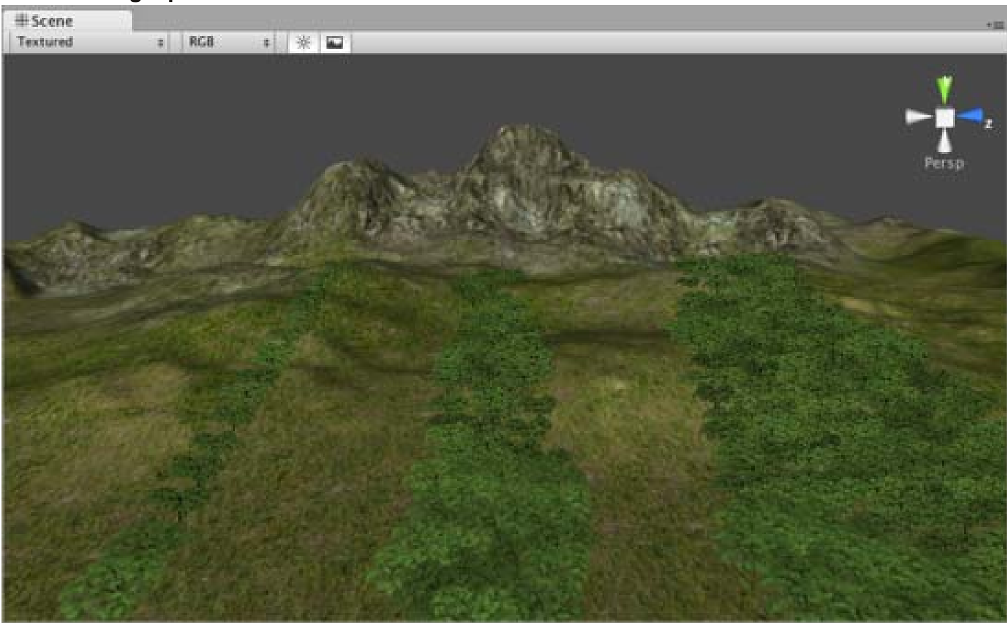
While still using the Place Trees tool, click anywhere on the Terrain to place your trees. To erase trees, hold the **Shift** button and click on the Terrain.

*Painting trees is as easy as using a paintbrush tool*

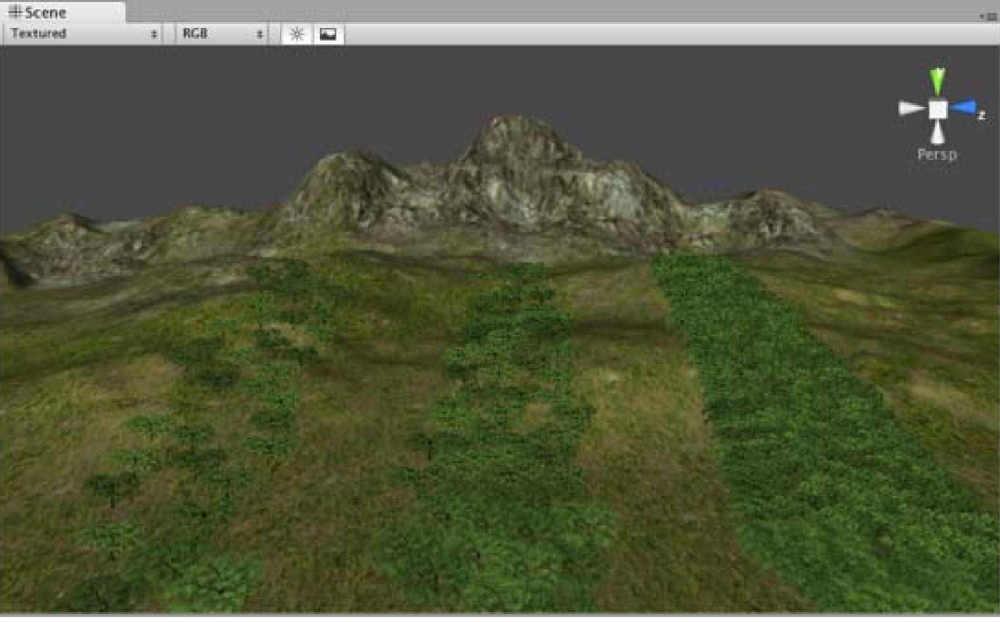
There are a number of options at your disposal when placing trees.

| | |
|---|---|
| **Brush Size** | Radius in meters of the tree placing brush. |
| **Tree Spacing** | Percentage of tree width between trees. |
| **Color Variation** | Allowed amount of color difference between each tree. |
| **Tree Height** | Height adjustment of each tree compared to the asset. |
| **Height Variation** | Allowed amount of difference in height between each tree. |
| **Tree Width** | Width adjustment of each tree compared to the asset. |
| **Width Variation** | Allowed amount of difference in width between each tree. |

**Tree Painting Tips**




*Different Brush sizes cover different area sizes*



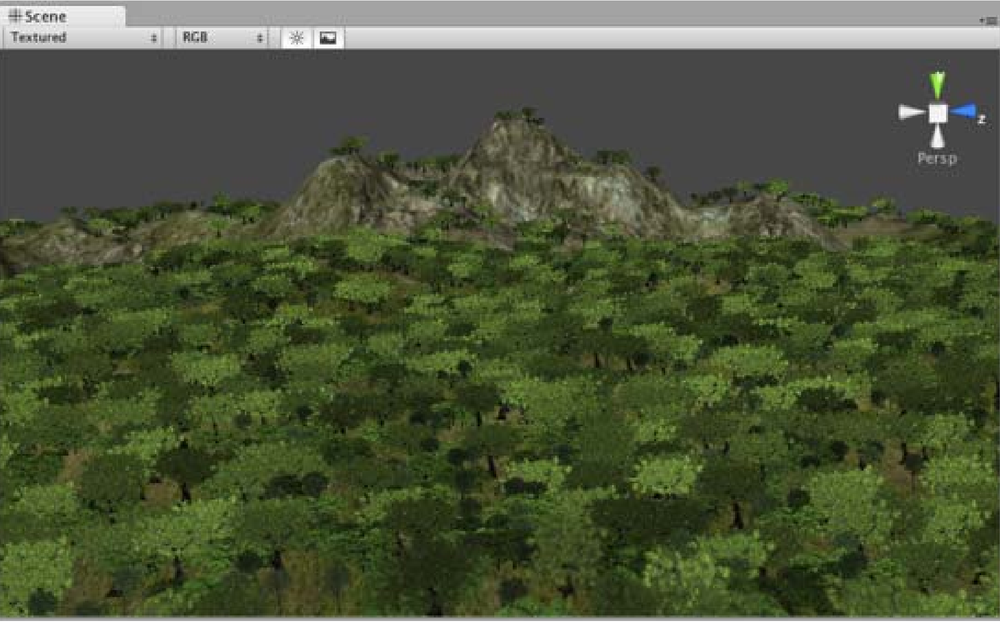

*Adjust **Tree Spacing** to change the density of the trees you're painting*

### Editing Trees

To change any import parameters for an added tree, select the detail and choose **Edit Trees button->Edit Detail**. Or double-click the tree you want to edit. You will then see the **Edit Tree** dialog, and you can change any of the settings.

### Mass Placement

If you don't want to paint your trees and you just want a whole forest created, you can use **Terrain->Mass Place Trees**. Here, you will see the **Mass Place Trees** dialog. You can set the number of trees you want placed, and they'll be instantly positioned. All the trees added to your Terrain will be used in this mass placement.




*10,000 Trees placed at once*

### Refreshing Source Assets

If you make any updates to your tree asset source file, it must be manually re-imported into the Terrain. To do this, use **Terrain->Refresh Tree and Detail Prototypes**. This is done after you've changed your source asset and saved it, and will refresh the trees in your Terrain immediately.

### Creating Trees

Trees can be created in two ways to the terrain engine:

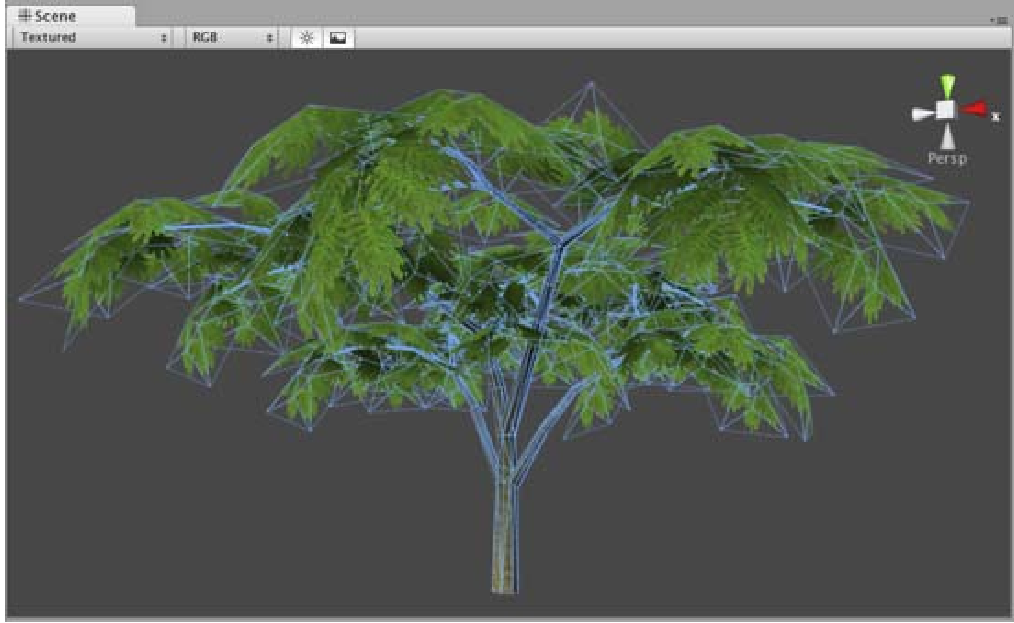
The first one is by using the Tree creator that Unity comes with, and the second one is by using a 3rd party modeling program compatible with Unity, in this case every tree should consist of a single mesh with two **Materials**. One for the trunk and one for the leaves.

For performance reasons, triangle count should be kept below 2000 for an average tree. The fewer triangles the better. The pivot point of the tree mesh must be exactly at the root of the tree, that is at the point where the tree should meet the surface it is placed on. This makes it the easiest to import into Unity and other modelling applications.

Trees must use the **Nature/Soft Occlusion Leaves** and **Nature/Soft Occlusion Bark shader**. In order to use those shaders you also have to place the tree in a special folder that contains the name "Ambient-Occlusion". When you place a model in that folder and reimport it, Unity will calculate soft ambient occlusion specialized for trees. The "Nature/Soft Occlusion" shaders need this information. If you don't follow the naming conventions the tree will look weird with completely black parts.

Unity also ships with several high quality trees in the "Terrain Demo.unitypackage". You can use those trees readily in your game.

### Using Low Poly Trees




One branch with leaves is done with only six triangles and shows quite a bit of curvature. You can add more triangles for even more curvature. But the main point is: When making trees, work with triangles not with quads. If you use quads you basically need twice as many triangles to get the same curvature on branches.

The tree itself wastes a lot of fillrate by having large polygons but almost everything is invisible due to the alpha. This should be avoided for performance reasons and of course because the goal is to make dense trees. This is one of the things that makes Oblivion's trees look great. They are so dense you cant even see through the leaves.

### Setting up Tree Collisions

If you'd like your trees to make use of colliders, it's very easy. When you've imported your Tree asset file, all you need to do is instantiate it in the Scene View, add a **Capsule Collider** and tweak it, and make the GameObject into a new Prefab. Then when you're adding trees to your Terrain, you add the tree Prefab with the Capsule Collider attached. You can only use Capsule Colliders when adding collisions with trees.

### Making your Trees Collide.

*Terrain Collider Inspector*

If you want to make your trees collide with rigid bodies, make sure you you check the *Create Tree Colliders* box. else your

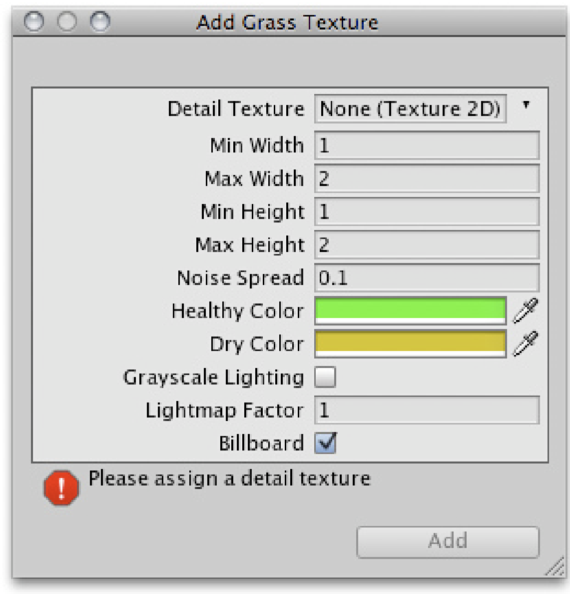
objects will go trough the trees you create. Note that the PhysX engine used by Unity only handles a maximum of 65536 colliders in a scene. If you use more trees then that (minus the other colliders already in the scene), then enabling tree colliders will fail with an error.

Page last updated: 2012-01-24

---

## terrain-Grass

The **Paint Foliage** button allows you to paint grass, rocks, or other decorations around the Terrain. To paint grass, choose **Edit Details button->Add Grass Texture**. You don't need to create a mesh for grass, just a texture.

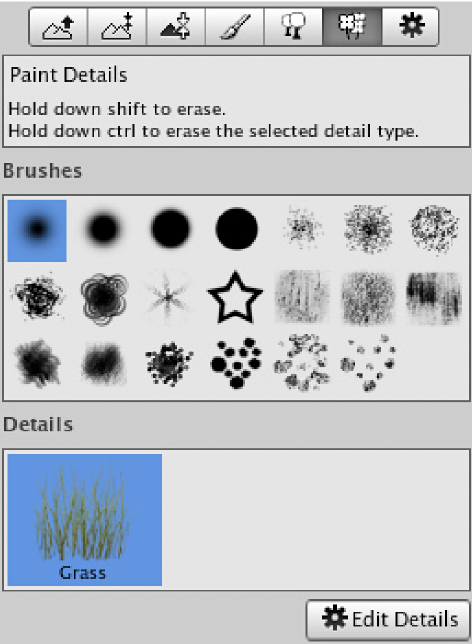



*The Add Grass Texture dialog*

At this dialog, you can fine-tune the appearance of the grass with the following options:

| | |
|---|---|
| **Detail Texture** | The texture to be used for the grass. |
| **Min Width** | Minimum width of each grass section in meters. |
| **Max Width** | Maximum width of each grass section in meters. |
| **Min Height** | Minimum height of each grass section in meters. |
| **Max Height** | Maximum height of each grass section in meters. |
| **Noise Spread** | The size of noise-generated clusters of grass. Lower numbers mean less noise. |
| **Healthy Color** | Color of healthy grass, prominent in the center of **Noise Spread** clusters. |
| **Dry Color** | Color of dry grass, prominent on the outer edges of **Noise Spread** clusters. |
| **Grayscale Lighting** | If enabled, grass textures will not be tinted by any colored light shining on the Terrain. |
| **Lightmap Factor** | How much the grass will be influenced by the Lightmap. |
| **Billboard** | If checked, this grass will always rotate to face the main **Camera**. |

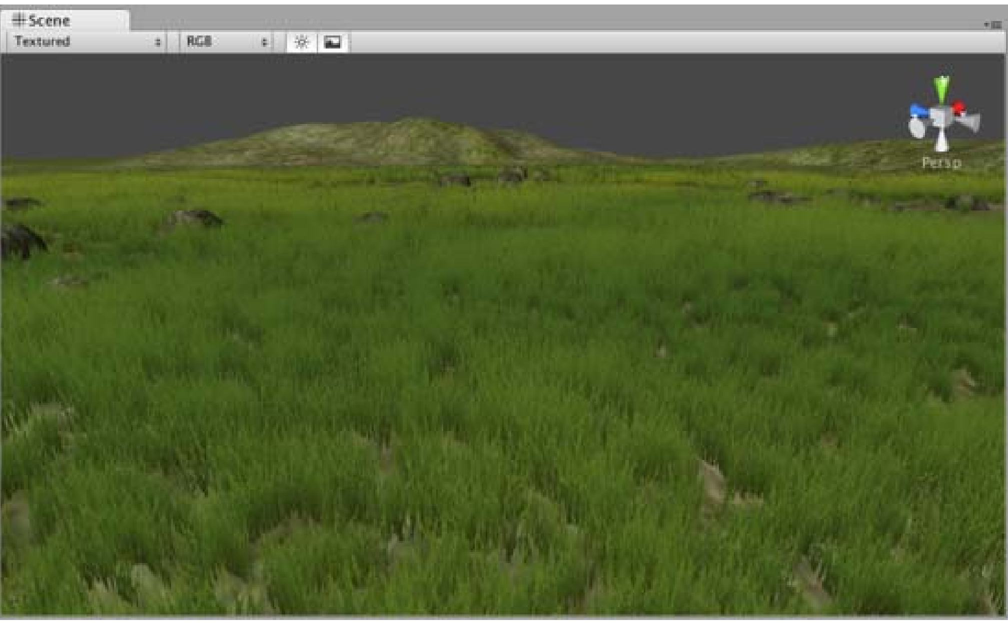
After you've clicked the **Add** button, you'll see the grass appear selectable in the **Inspector**.




*The added grass appears in the Inspector*

### Painting Grass

Painting grass works the same as painting textures or trees. Select the grass you want to paint, and paint right onto the Terrain in the **Scene View**



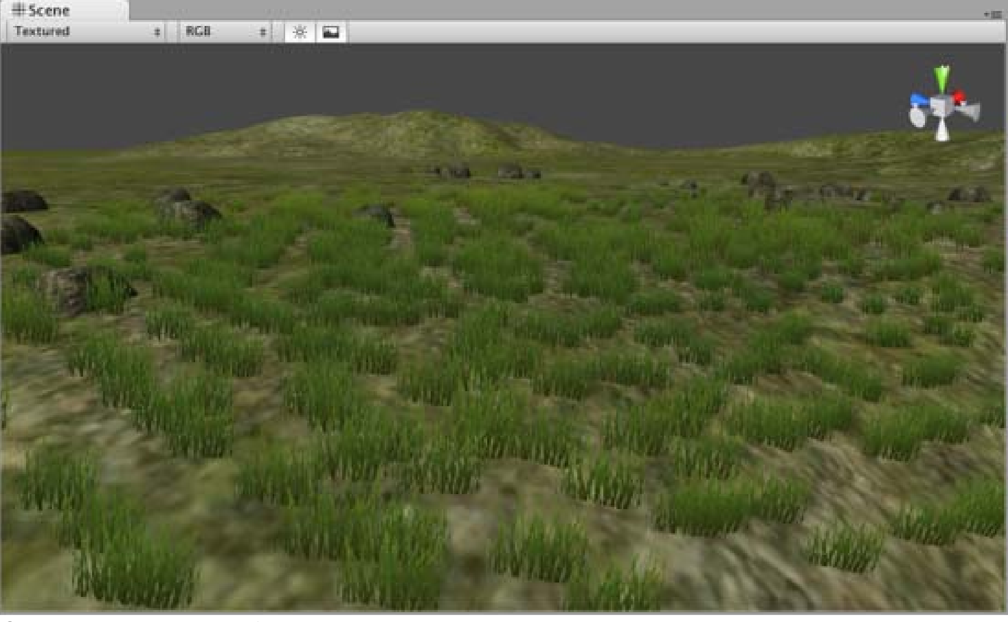

*Painting grass is easy as pie*

**Note:** When you have a brush selected, move your mouse over the Terrain in the Scene View and press **F**. This will center the Scene View over the mouse pointer position and automatically zoom in to the **Brush Size** distance. This is the quickest & easiest way to navigate around your Terrain while creating it.

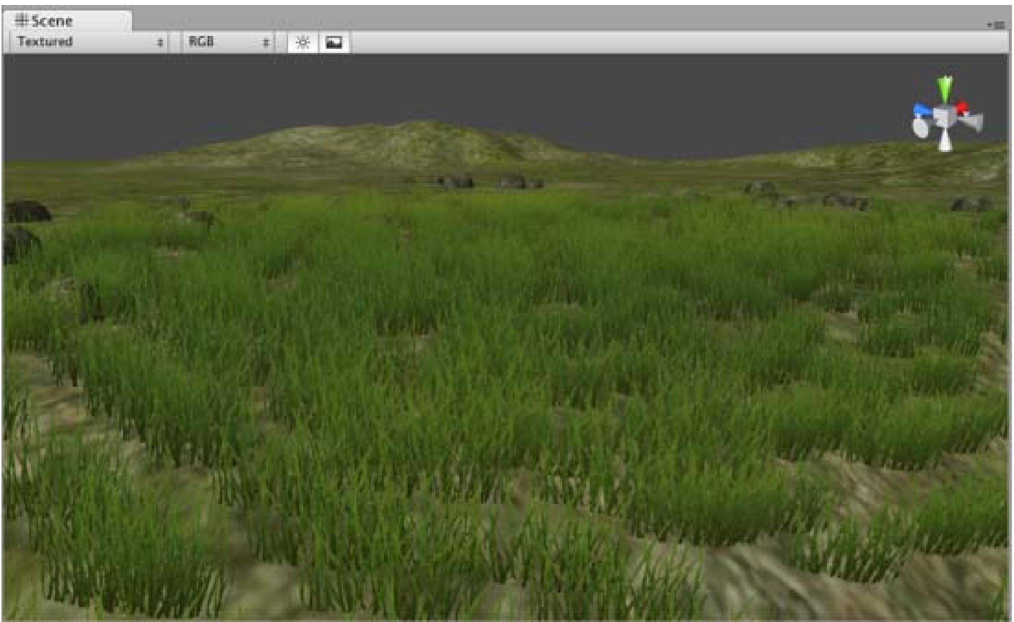
### Editing Grass

To change any import parameters for a particular Grass Texture, select it choose **Edit Details button->Edit**. You can also double-click it. You will then see the **Edit Grass** dialog appear, and be able to adjust the parameters described above.

You'll find that changing a few parameters can make a world of difference. Even changing the **Max/Min Width** and **Height** parameters can vastly change the way the grass looks, even with the same number of grass objects painted on the Terrain.




*Grass created with the default parameters*




*The same number of painted grass objects, now wider and taller*

Page last updated: 2011-10-29

---

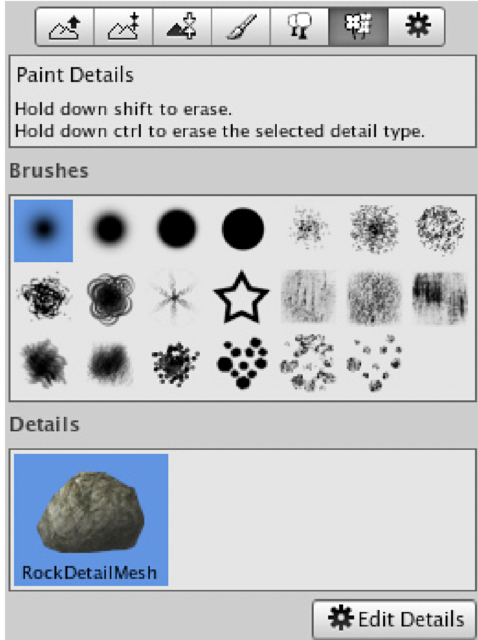
## terrain-DetailMeshes

Any Terrain decoration that is not trees or grass should be created as a **Detail Mesh**. This is perfect for things like rocks, 3D shrubbery, or other static items. To add these, use the **Paint Foliage** button . Then choose **Edit Details button->Add Detail Mesh**. You will see the **Add Detail Mesh** dialog appear.

*The Add Detail Mesh dialog*

| | |
|---|---|
| **Detail** | The mesh to be used for the detail. |
| **Noise Spread** | The size of noise-generated clusters of the **Detail**. Lower numbers mean less noise. |
| **Random Width** | Limit for the amount of width variance between all detail objects. |
| **Random Height** | Limit for the amount of height variance between all detail objects. |
| **Healthy Color** | Color of healthy detail objects, prominent in the center of **Noise Spread** clusters. |
| **Dry Color** | Color of dry detail objects, prominent on the outer edges of **Noise Spread** clusters. |
| **Grayscale Lighting** | If enabled, detail objects will not be tinted by any colored light shining on the Terrain. |
| **Lightmap Factor** | How much the detail objects will be influenced by the Lightmap. |
| **Render Mode** | Select whether this type of detail object will be lit using Grass lighting or normal Vertex lighting. Detail objects like rocks should use Vertex lighting. |

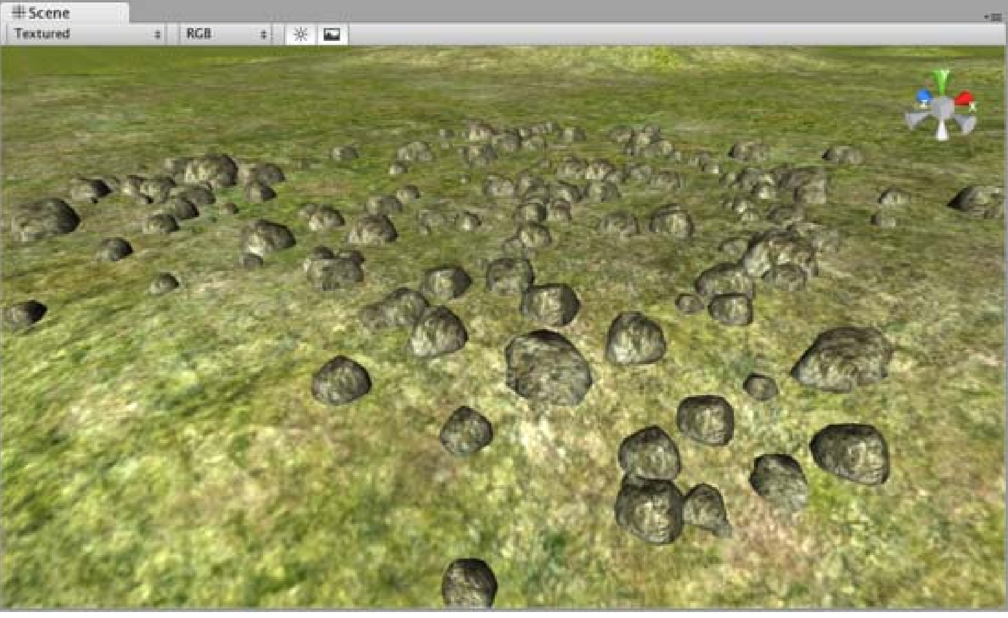
After you've clicked the **Add** button, you'll see the Detail mesh appear in the **Inspector**. Detail meshes and grass will appear next to each other.




*The added Detail mesh appears in the Inspector, beside any Grass objects*

### Painting Detail Meshes

Painting a Detail mesh works the same as painting textures, trees, or grass. Select the Detail you want to paint, and paint right onto the Terrain in the Scene View.




*Painting Detail meshes is very simple*

**Note:** When you have a brush selected, move your mouse over the Terrain in the Scene View and press **F**. This will center the Scene View over the mouse pointer position and automatically zoom in to the **Brush Size** distance. This is the quickest & easiest way to navigate around your Terrain while creating it.

### Editing Details

To change any import parameters for a Detail Mesh, select it and choose **Edit Details button->Edit**. You will then see the **Edit Detail Mesh** dialog appear, and be able to adjust the parameters described above.

### Refreshing Source Assets

If you make any updates to your Detail Mesh asset source file, it must be manually re-imported into the Terrain. To do this, use **Terrain->Refresh Tree and Detail Prototypes**. This is done after you've changed your source asset and saved it, and will refresh the Detail Meshes in your Terrain immediately.

### Hints:

- The UVs of the detail mesh objects need to be in the 0-1 range because all the separate textures used for all the detail meshes are packed into a single texture atlas.

Page last updated: 2011-10-29

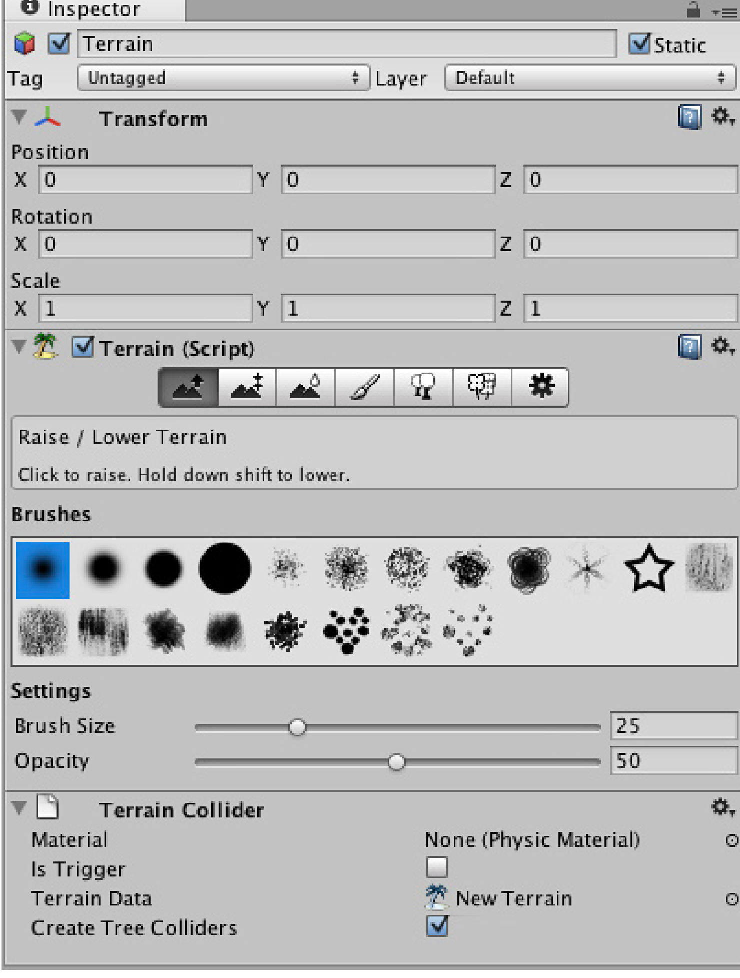
---

# Terrain Lightmapping

This section will explain how to use the **Terrain Engine**. It will cover creation, technical details, and other considerations. It is broken into the following sections:

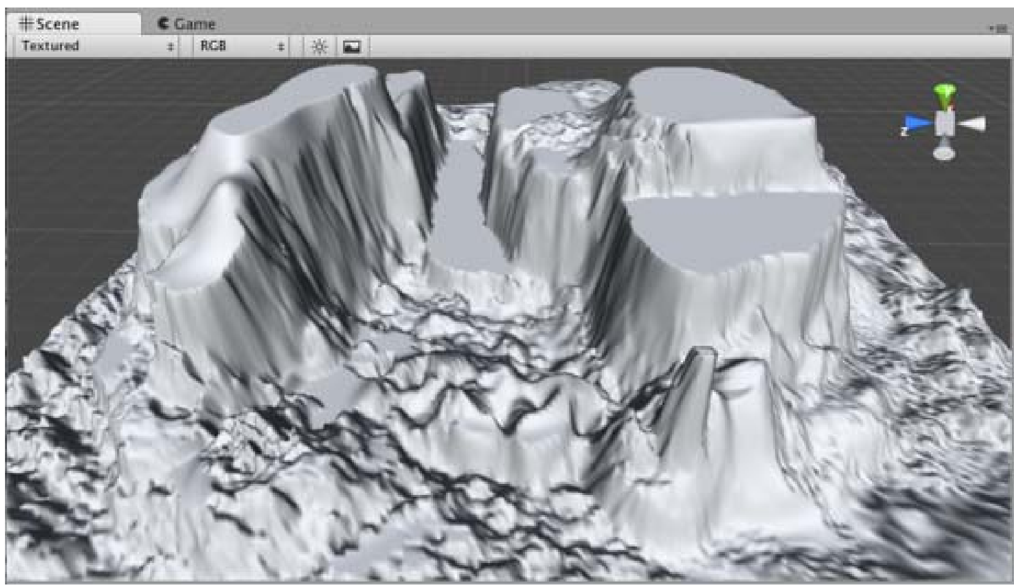
### Using Terrains

This section covers the most basic information about using Terrains. This includes creating Terrains and how to use the new Terrain tools & brushes.




## Height

This section explains how to use the different tools and brushes that alter the Height of the Terrain.

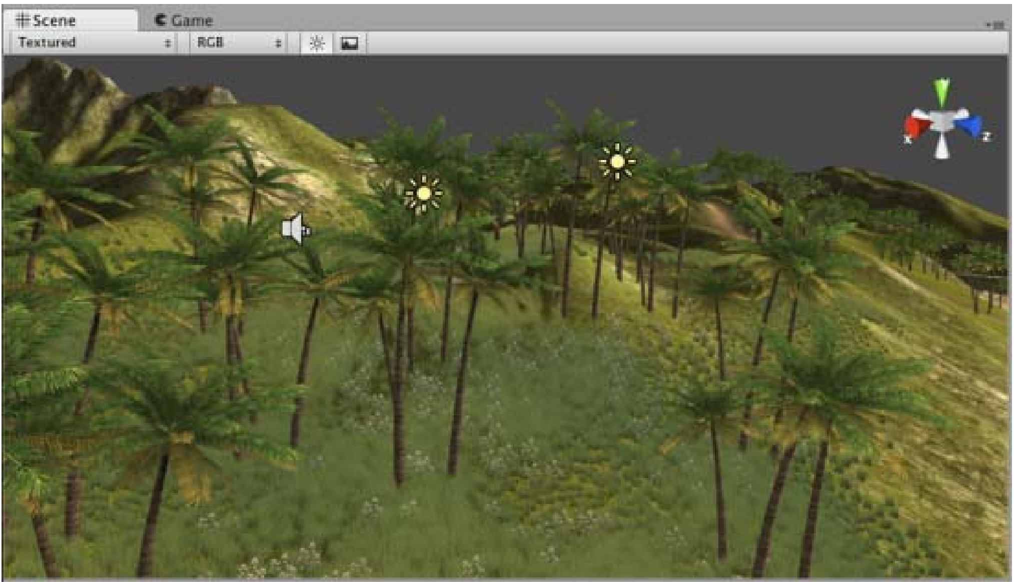



## Terrain Textures

This section explains how to add, paint and blend Terrain Textures using different brushes.

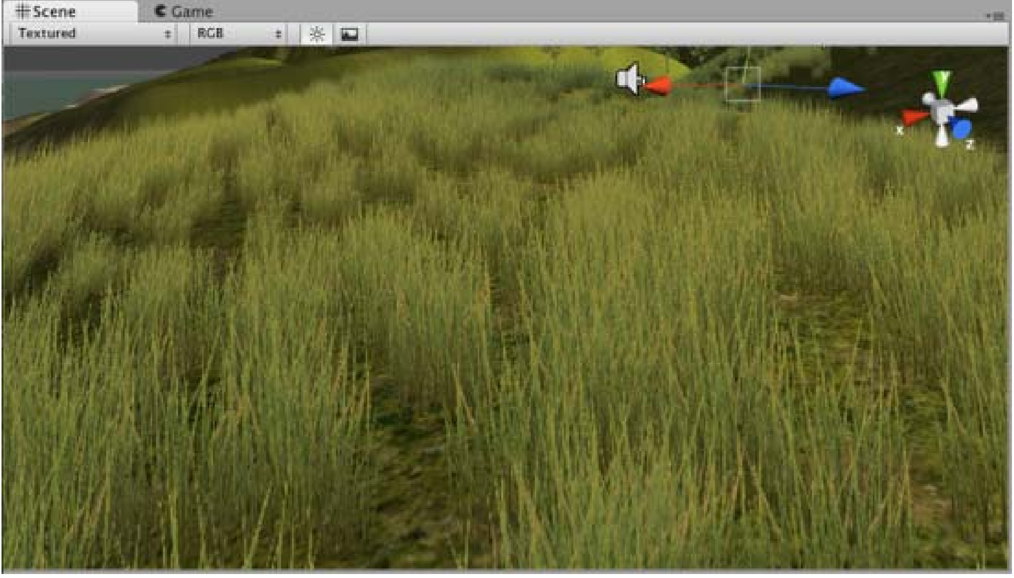
## Trees

This section contains important information for creating your own tree assets. It also covers adding and painting trees on your Terrain.




## Grass

This section explains how grass works, and how to use it.




## Detail Meshes

This section explains practical usage for detail meshes like rocks, haystacks, vegetation, etc.

## Lightmaps

You can lightmap terrains just like any other objects using Unity's built-in lightmapper. See Lightmapping Quickstart for help.

### Other Settings

This section covers all the other settings associated with Terrains.

#### Mobile performance note

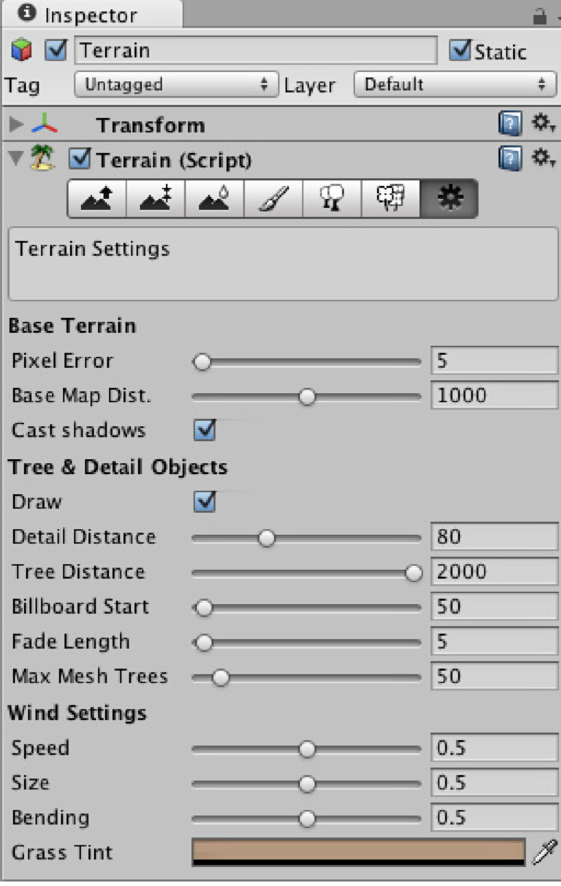
Rendering terrain is quite expensive, so terrain engine is not very practical on lower-end mobile devices.

Page last updated: 2012-08-17

---

## terrain-OtherSettings

There are a number of options under the **Terrain Settings** button in the Terrain Inspector.




*All additional Terrain Settings*

### Base Terrain

- **Pixel Error** controls the amount of allowable errors in the display of Terrain Geometry. This is essentially a geometry LOD setting; the higher the value, the less dense terrain geometry will be.
- **Base Map Dist.**: The distance that Terrain Textures will be displayed in high-resolution. After this distance, a low-resolution composited texture will be displayed.
- **Cast Shadows**: Should terrain cast shadows?

**Material** slot allows assigning a custom material for the terrain. The material should be using a shader that is capable of rendering terrain, for example **Nature/Terrain/Diffuse** (this shader is used if no material is assigned) or **Nature/Terrain /Bumped Specular**.

### Tree & Detail Settings

- **Draw**: if enabled, all trees, grass, and detail meshes will be drawn.
- **Detail Distance** distance from the camera that details will stop being displayed.
- **Tree Distance**: distance from the camera that trees will stop being displayed. The higher this is, the further-distance trees can be seen.
- **Billboard Start**: distance from the camera that trees will start appearing as Billboards instead of Meshes.
- **Fade Length**: total distance delta that trees will use to transition from Billboard orientation to Mesh orientation.
- **Max Mesh Trees**: total number of allowed mesh trees to be capped in the Terrain.

### Wind Settings

- **Speed**: the speed that wind blows through grass.
- **Size**: the areas of grass that are affected by wind all at once.
- **Bending**: amount that grass will bend due to wind.
- **Grass Tint**: overall tint amount for all Grass and Detail Meshes.

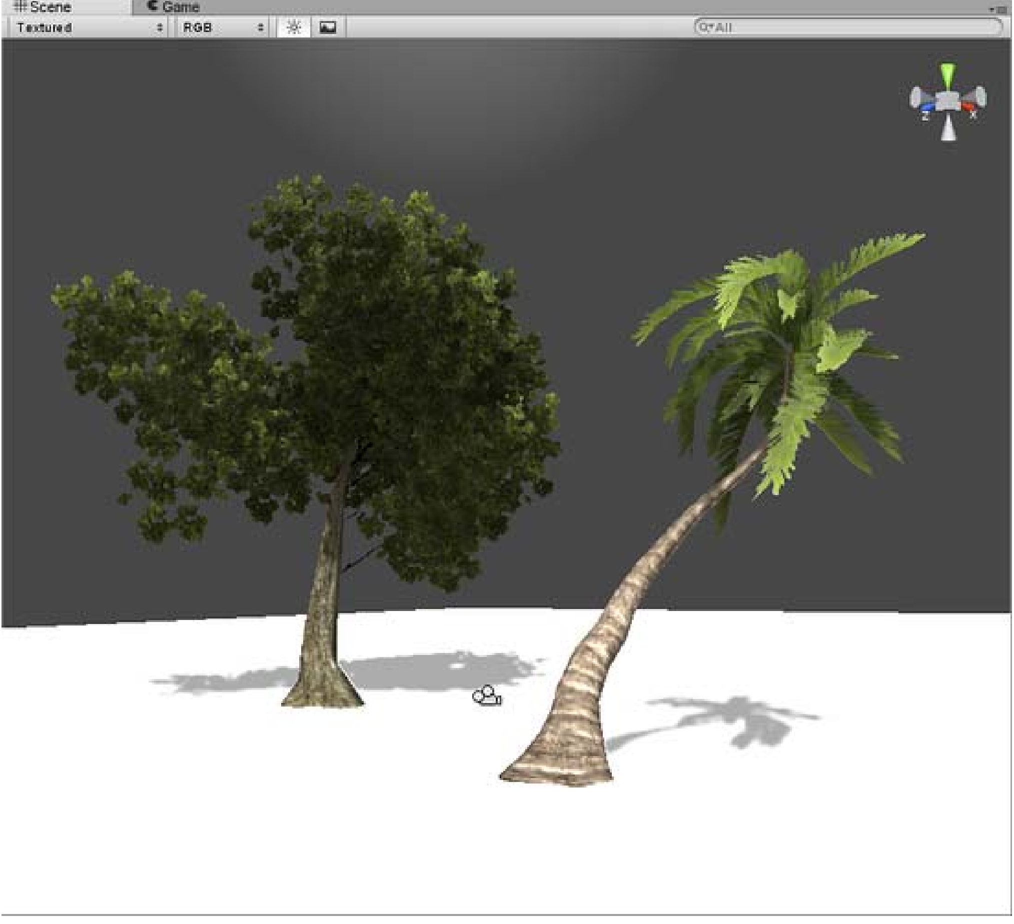
Page last updated: 2012-11-16

# class-Tree

The **Tree Creator** allows you to create and edit trees procedurally. The resulting trees can be used as normal **GameObjects** or integrated into the Terrain Engine. With this tool, you can create great looking trees quickly and fine tune your trees using a number of design tools to achieve the perfect look in your game. The Tree Creator is very useful when you want to create a forest or a jungle where you need different tree forms.

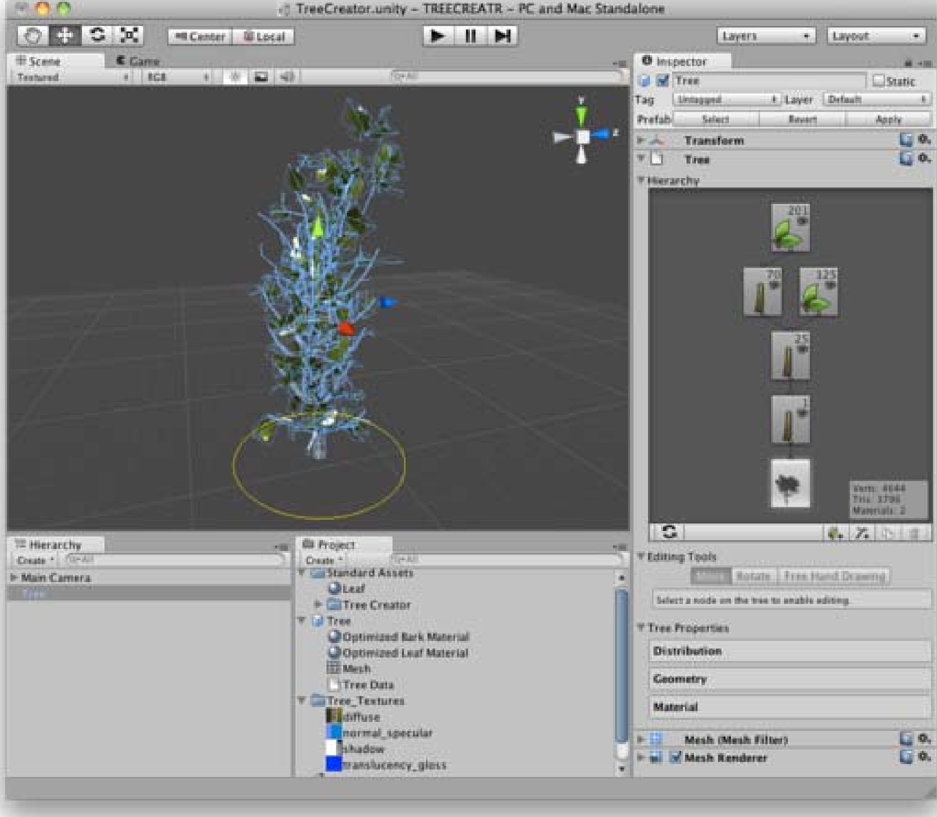
## Building Your First Tree

In this section we walk you step by step to create your first tree in Unity.



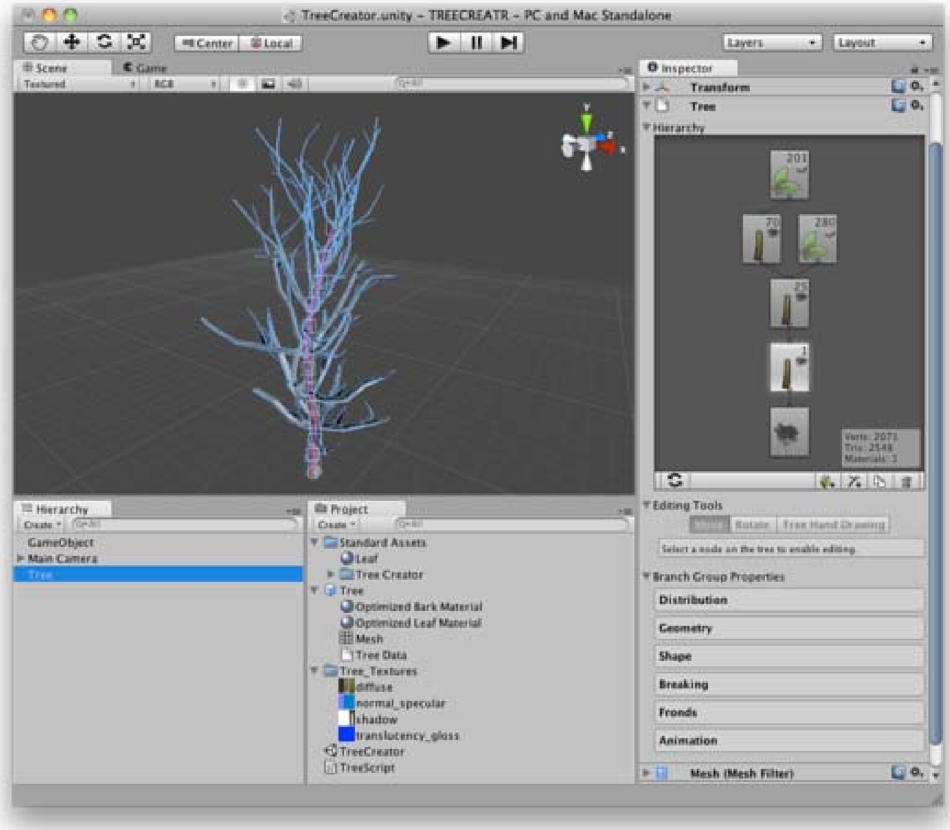

## Tree Creator Structure

This section provides a general overview of the Tree Creator's user interface.



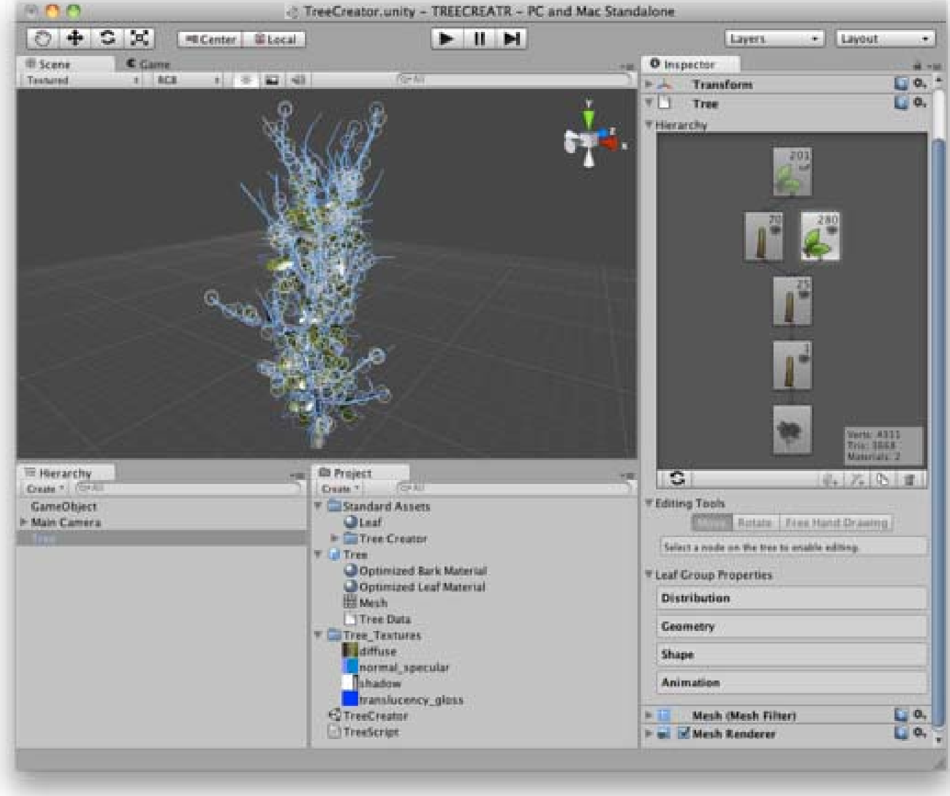

## Branches

This section focuses on explaining the specific properties of branches.



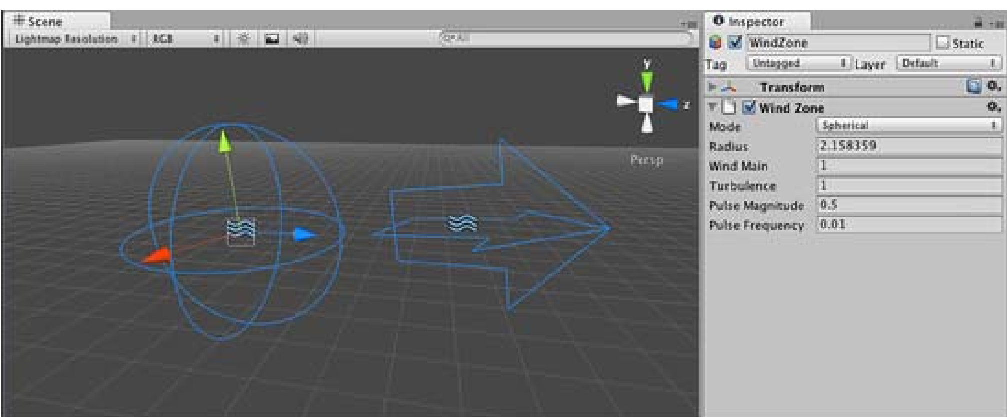
## Leaves

This section focuses on explaining the specific properties of leaves.



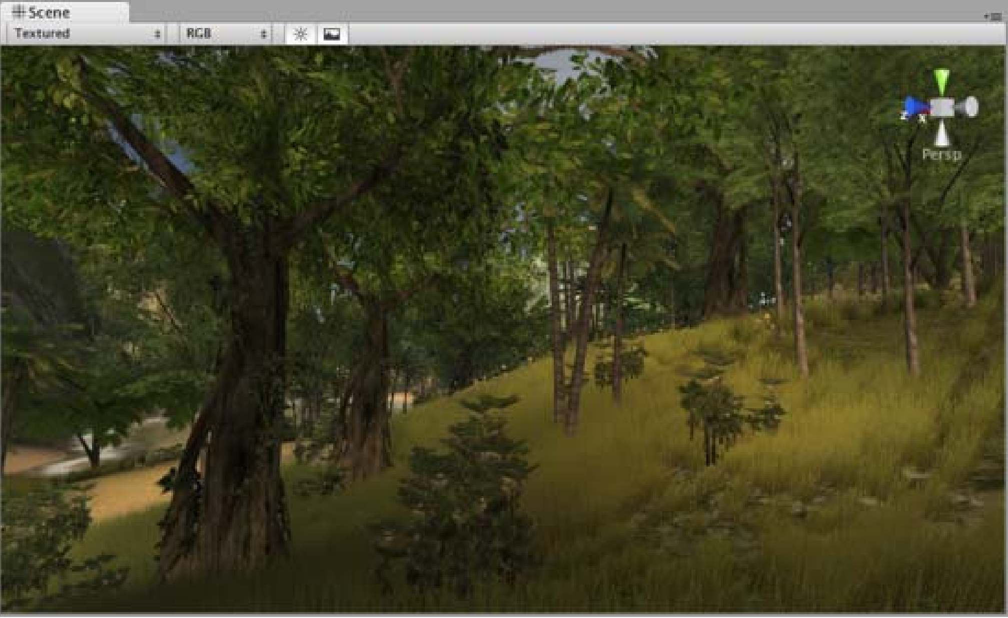

## Wind Zones

This section explains Wind Zones and how to apply them to your trees.




## Trees in the Terrain Engine

This section of the Terrain Engine Guide covers all the basics of integrating your trees into the Terrain Engine.




Page last updated: 2011-10-29

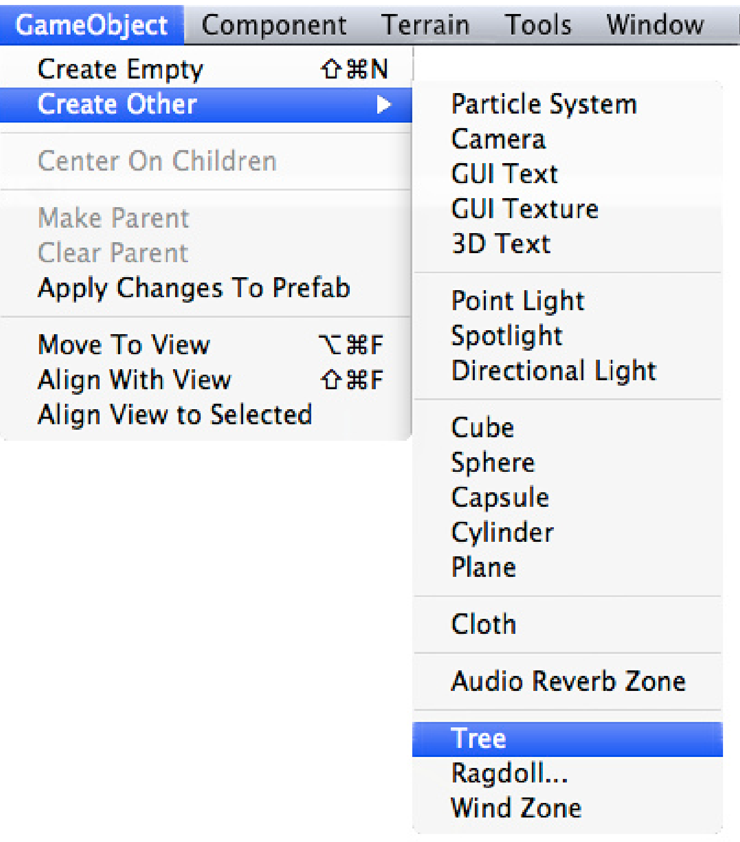
---

## tree-FirstTree

We'll now walk you through the creation of your first Tree Creator Tree in Unity. First, make sure you have included the tree creator package in your project. If you don't, select **Assets->Import Package...**, navigate to your Unity installation folder, and open the folder named *Standard Packages*. Select the *Tree Creator.unityPackage* package to get the needed assets into your project.

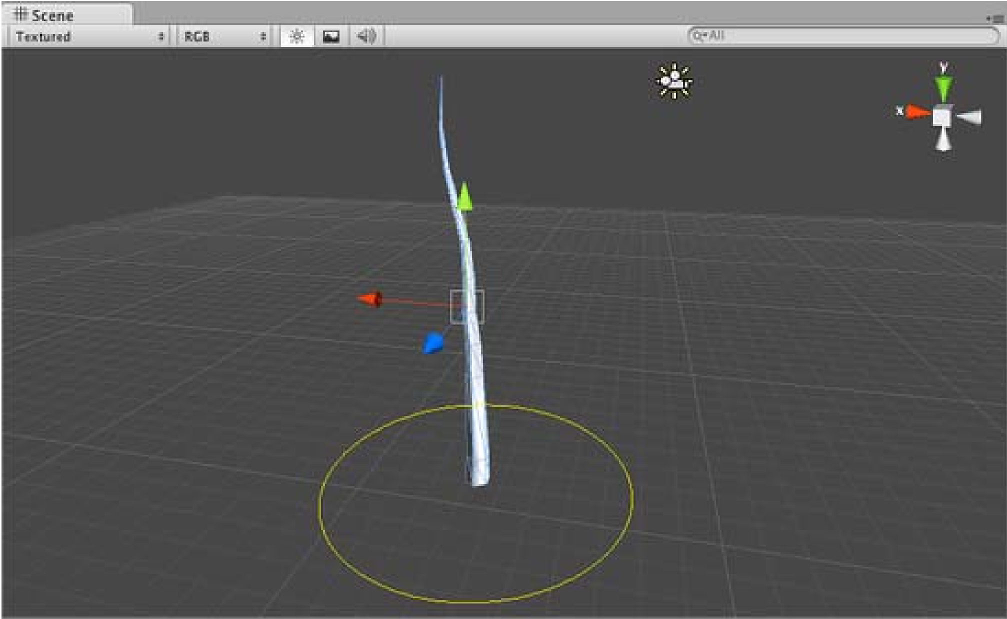
### Adding a new Tree

To create a new **Tree** asset, select **GameObject->Create Other->Tree**.




You'll see a new Tree asset is created in your Project View, and instantiated in the currently open Scene. This new Tree is very basic with only a single branch, so let's add some character to it.

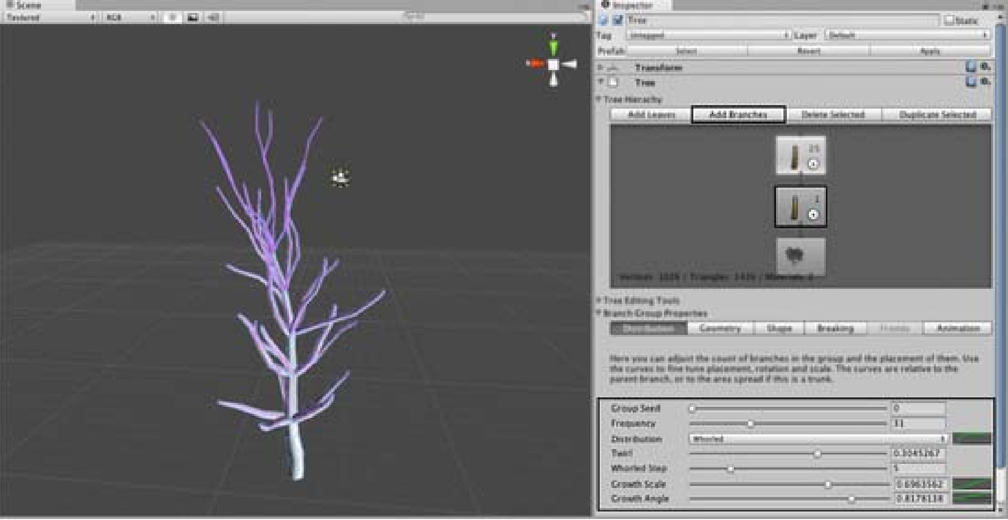
### Adding Branches




*A brand new tree in your scene*

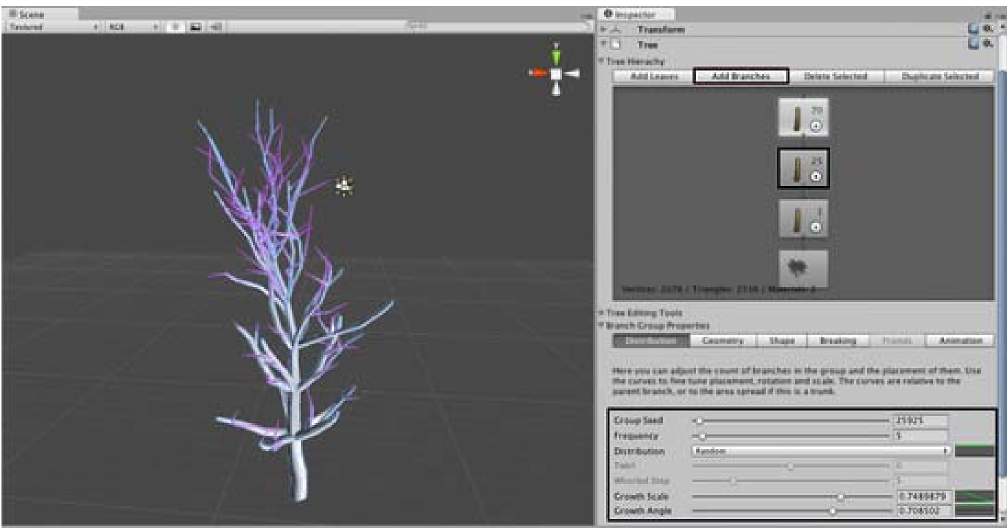
Select the tree to view the **Tree Creator** in the **Inspector**. This interface provides all the tools for shaping and sculpting your trees. You will see the **Tree Hierarchy** with two nodes present: the **Tree Root** node and a single **Branch Group** node, which we'll call the trunk of the tree.

In the **Tree Hierarchy**, select the **Branch Group**, which acts as the trunk of the tree. Click on the **Add Branch Group** button and you'll see a new **Branch Group** appear connected to the Main Branch. Now you can play with the settings in the **Branch Group Properties** to see alterations of the branches attached to the tree trunk.



*Adding branches to the tree trunk.*

After creating the branches that are attached to the trunk, we can now add smaller twigs to the newly created branches by attaching another **Branch Group** node. Select the secondary **Branch Group** and click the **Add Branch Group** button again. Tweak the values of this group to create more branches that are attached to the secondary branches.

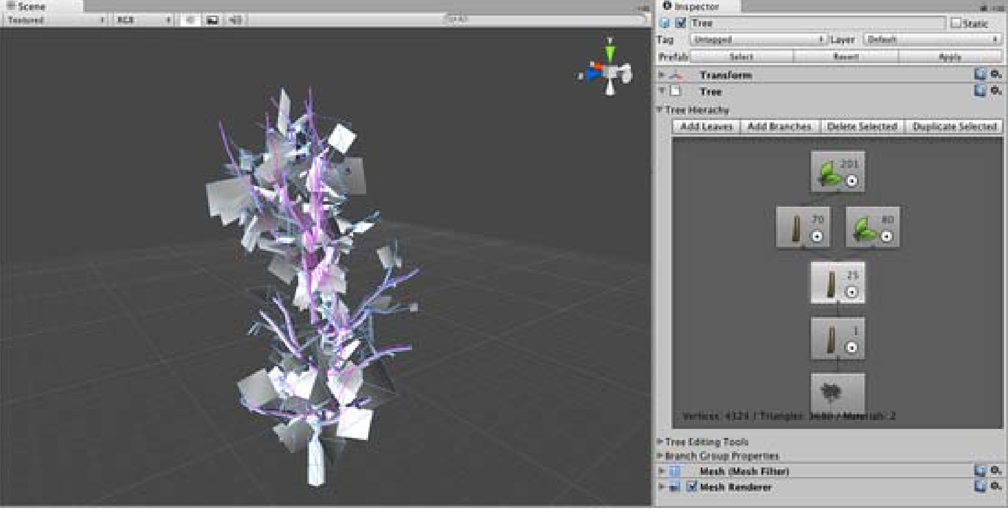


*Adding branches to the secondary branches.*

Now the tree's branch structure is in place. Our game doesn't take place in the winter time, so we should also add some **Leaves** to the different branches, right?

### Adding Leaves

We decorate our tree with leaves by adding **Leaf Groups**, which basically work the same as the Branch groups we've already used. Select your secondary Branch Group node and then click the **Add Leaf Group** button. If you're really hardcore, you can add another leaf group to the tiniest branches on the tree as well.



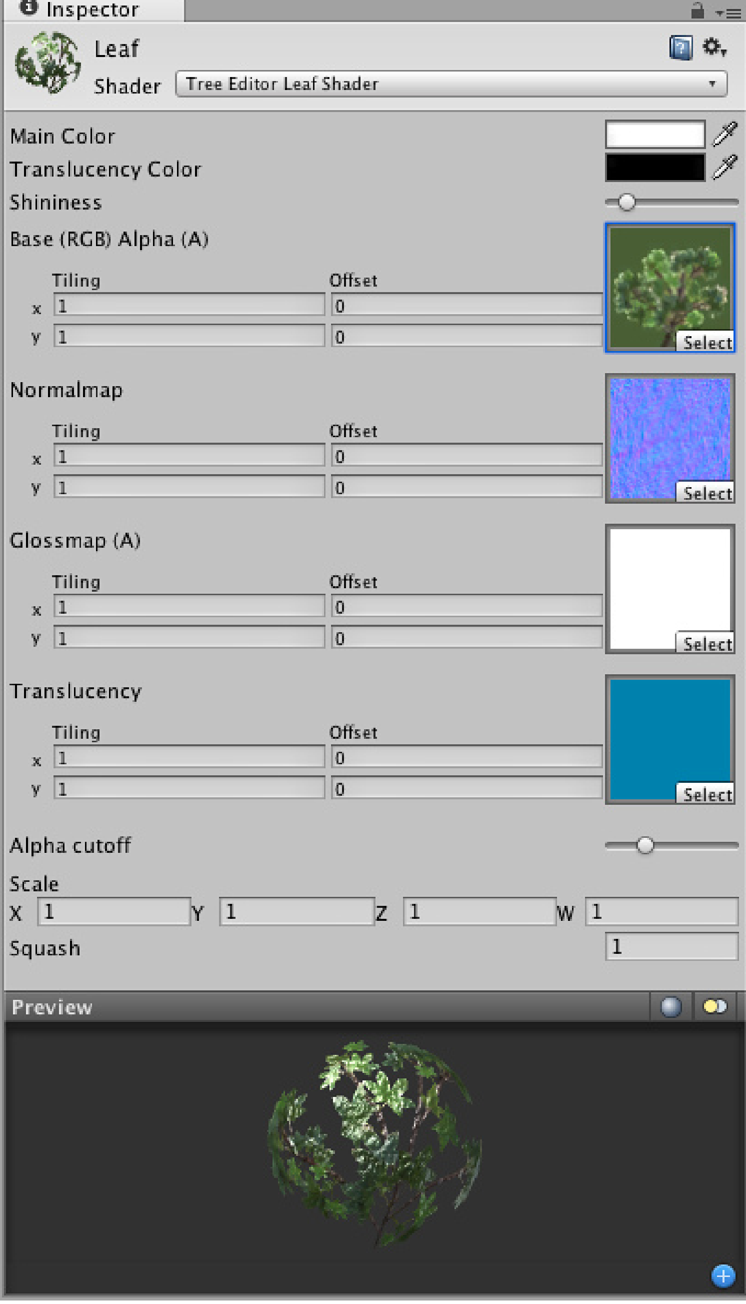
*Leaves added to the secondary and smallest branches*

Right now the leaves are rendered as opaque planes. This is because we want to adjust the leaves' values (size, position, rotation, etc.) before we add a material to them. Tweak the Leaf values until you find some settings you like.

## Adding Materials

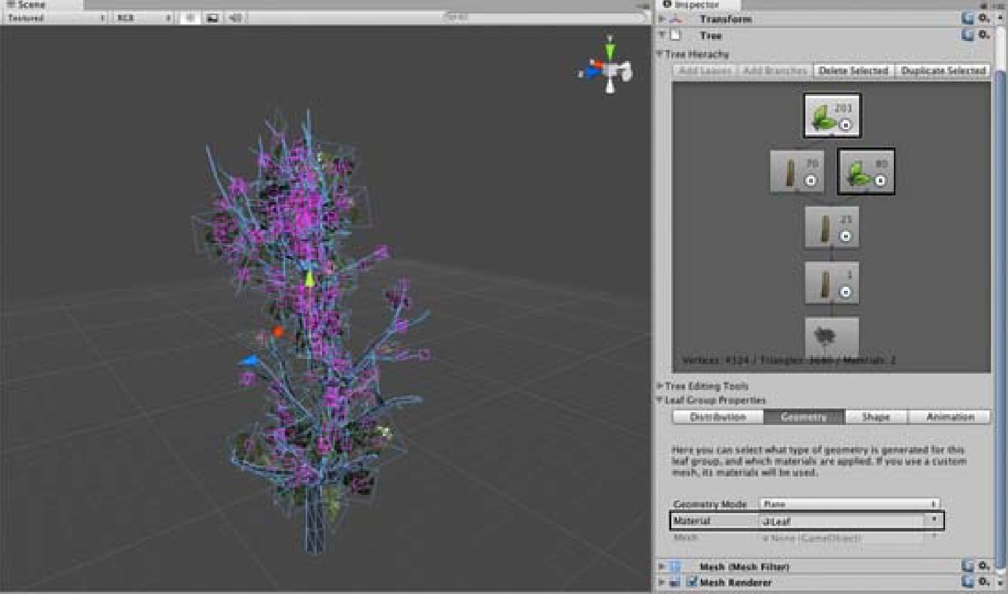
In order to make our tree realistic looking, we need to apply **Materials** for the branches and the leaves. Create a new Material in your project using ^Assets->Create->Material**. Rename it to "My Tree Bark", and choose** Nature->Tree Creator Bark^^ from the Shader drop-down. From here you can assign the **Textures** provided in the Tree Creator Package to the Base, Normalmap, and Gloss properties of the Bark Material. We recommend using the texture "BigTree_bark_diffuse" for the Base and Gloss properties, and "BigTree_bark_normal" for the Normalmap property.

Now we'll follow the same steps for creating a Leaf Material. Create a new Material and assign the shader as **Nature->Tree Creator Leaves**. Assign the texture slots with the leaf textures from the Tree Creator Package.



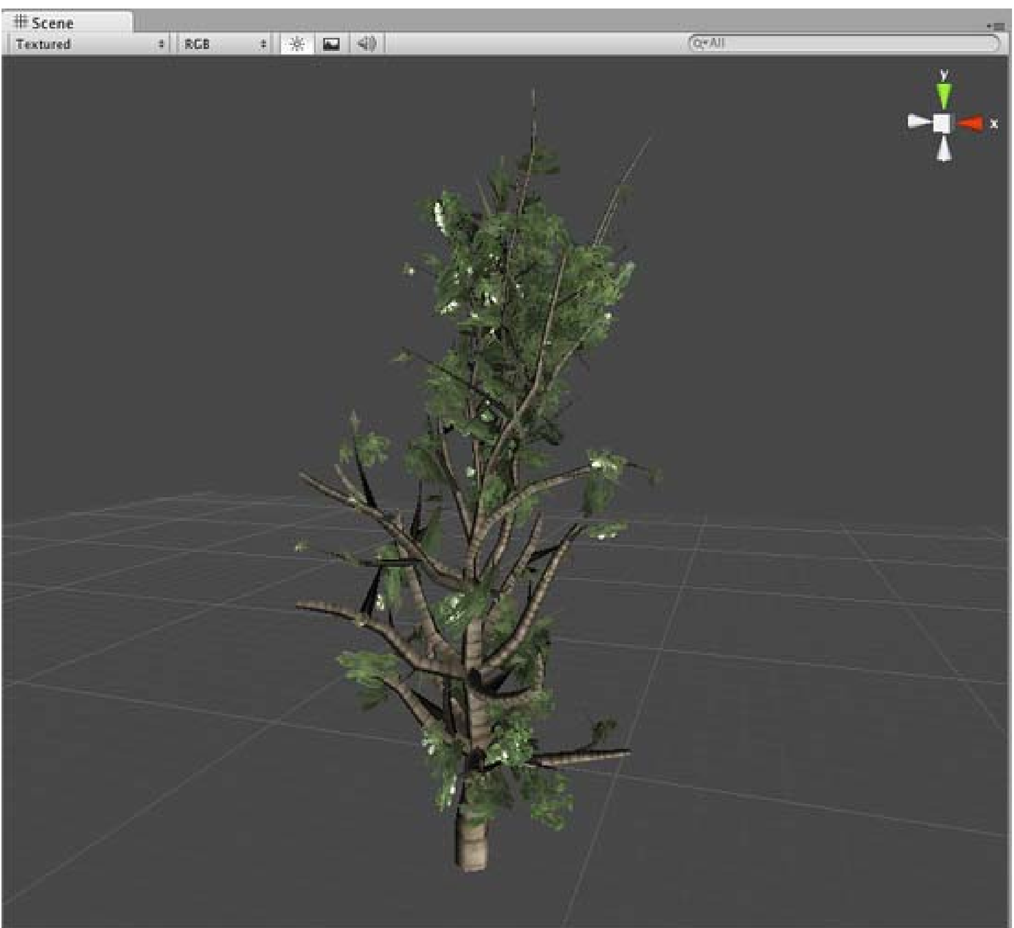

*Material for the Leaves*

When both Materials are created, we'll assign them to the different Group Nodes of the Tree. Select your Tree and click any Branch or Leaf node, then expand the **Geometry** section of the **Branch Group Properties**. You will see a Material assignment slot for the type of node you've selected. Assign the relevant Material you created and view the results.




*Setting the leaves material*

To finish off the tree, assign your Materials to all the **Branch** and **Leaf Group** nodes in the Tree. Now you're ready to put your first tree into a game!




*Tree with materials on leaves and branches.*

### Hints.

- Creating trees is a trial and error process.
- Don't create too many leaves/branches as this can affect the performance of your game.
- Check the alpha maps guide for creating custom leaves.

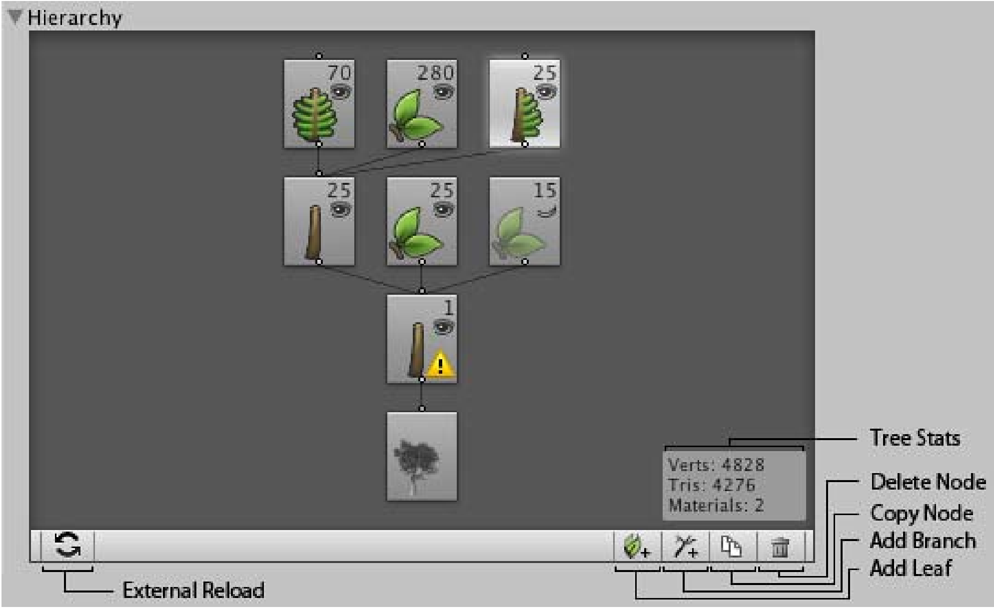
Page last updated: 2011-10-29

## tree-Structure

The Tree Creator's inspector is split into three different panes: Hierarchy, Editing Tools and Properties.

### Hierarchy

The hierarchy view is where you start building your tree. It shows a schematic representation of the tree, where each box is a group of nodes. By selecting one of the groups in the hierarchy you can modify its properties. You may also add or remove groups by clicking one of the buttons in the toolbar below the hierarchy.

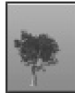



This hierarchy represents a tree with one trunk, 25 child branches. Child branches have in total 70 fronds attached, 280 leaves and 25 fronds with branches. Node representing the last group is selected. Also the trunk has 25 leaves of one type and 15 of another type, the last group is hidden.

| | |
|---|---|
| **Tree Stats** | Status information about your tree, tells you how many vertices, triangles and materials the tree has. |
| **Delete Node** | Deletes the currently selected group in the Hierarchy or a node or a spline point in the Scene View. |
| **Copy Node** | Copies the currently selected group. |
| **Add Branch** | Adds a branch group node to the currently selected group node. |
| **Add Leaf** | Adds a leaf group node to the currently selected group node. |
| **External Reload** | Recomputes the whole tree, should be used when the source materials have changed or when a mesh used in Mesh **Geometry Mode** on leaves has changed. |

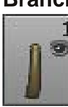
Nodes in the tree hierarchy represent groups of elements in the tree itself, i.e. branches, leaves or fronds. There are 5 types of nodes:

**Root Node:**



This is the starting point of a tree. It determines the global parameters for a tree, such as: quality, seed to diversify the trees, ambient occlusion and some material properties.

**Branch Nodes**




First branch group node attached to the root node creates trunk(s). The following branch nodes will create child branches. Various shape, growth and breaking parameters can be set on this type of node.

**Leaf Node**

Leaves can be attached to a root node (e.g. for small bushes) or to branch nodes. Leaves are final nodes, meaning no other nodes can be attached to them. Various geometry and distribution parameters can be set on this type of a node.

**Frond Node**

It has a similar behavior as the branch node, with some of the shape properties disabled and frond specific ones added.

**Branch + Frond Node**

This kind of node is a mixture of a branch and a frond, with properties of both types being accessible.

### Node Parameters

- Number at the top right of each node represents the number of elements this node created in the tree. The value is related to the frequency parameter from the Distribution tab.
- A node can be visible ( ) or invisible ( ).
- If a node was edited manually (branch splines or leaves were manipulated in the Scene View) a warning sign will appear over the node ( ). In such a case some procedural properties will be disabled.

## Editing Tools

While the Tree Creator works with procedural elements, you can decide at any time to modify them by hand to achieve the exact positioning and shape of elements you want.
Once you have edited a group by hand, certain procedural properties will no longer be available. You can, however, always revert it to a procedural group by clicking the button displayed in the **Branch/Leaf Group Properties**.

| | |
|---|---|
| **Move** | Select a node or a spline point in the Scene View. Dragging the node allows you to move it along and around its parent. Spline points can be moved using the normal move handles. |
| **Rotate** | Select a node or a spline point in the Scene View. Both will show the normal rotation handles. |
| **Free Hand Drawing** | Click on a spline point and drag the mouse to draw a new shape. Release the mouse to complete the drawing. Drawing always happens on a plane perpendicular to the viewing direction. |

## Global Tree Properties

Every tree has a root node which contains the global properties. This is the least complex group type, but it contains some important properties that control the rendering and generation of the entire tree.

### Distribution

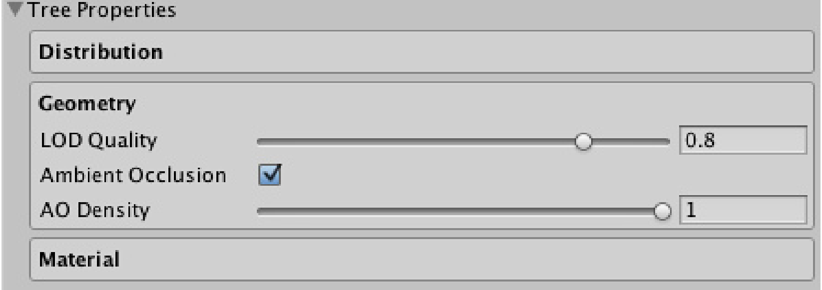
Allows for diversifying trees by changing the **Tree Seed** and making groups of trees by adjusting the **Area Spread** when the frequency on the branch node connected to the root node is higher than 1.

Tree Properties
Distribution
Tree Seed 0
Area Spread 5
Ground Offset 0
Geometry
Material

| | |
|---|---|
| **Tree Seed** | The global seed that affects the entire tree. Use it to randomize your tree, while keeping the general structure of it. |
| **Area Spread** | Adjusts the spread of trunk nodes. Has an effect only if you have more than one trunk. |
| **Ground Offset** | Adjusts the offset of trunk nodes on Y axis. |

### Geometry

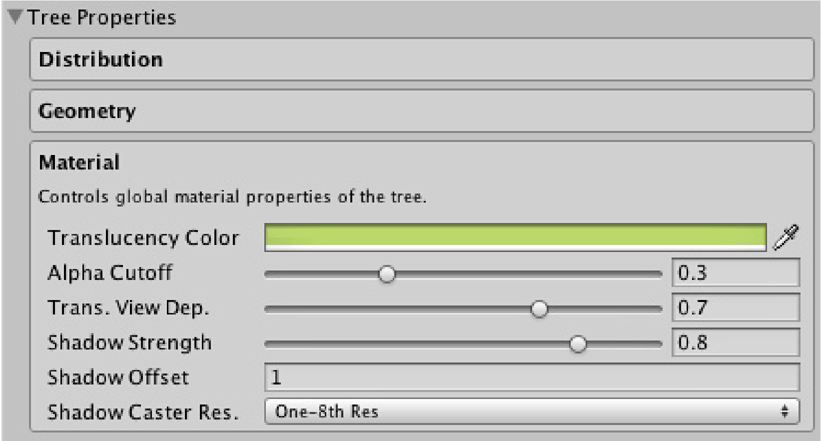
Allows to set the overall quality of the tree geometry and to control ambient occlusion.




| | |
|---|---|
| **LOD Quality** | Defines the level-of-detail for the entire tree. A low value will make the tree less complex, a high value will make the tree more complex. Check the statistics in the hierarchy view to see the current complexity of the mesh. Depending on the type of the tree and your target platform, you may need to adjust this property to make the tree fit within your polygon budget. With careful creation of art assets you can produce good looking trees with relatively few polygons. |
| **Ambient Occlusion** | Toggles ambient occlusion on or off. While modifying tree properties ambient occlusion is always hidden and won't be recomputed until you finish your changes, e.g. let go a slider. Ambient occlusion can greatly improve the visual quality of your tree, but its computation takes time, so you may want to disable it until you are happy with the shape of your tree. |
| **AO Density** | Adjusts the density of ambient occlusion. The higher the value the darker the effect. |

### Material

Controls the global material properties of the tree.




Translucency is one of the effects you can control in the Material properties. That property has an effect on leaves, which are translucent meaning that they permit the light to pass through them, but they diffuse it on the way.

| | |
|---|---|
| **Translucency Color** | The color that will be multiplied in when the leaves are backlit. |
| **Translucency View Dependency** | Fully view dependent translucency is relative to the angle between the view direction and the light direction. View independent is relative to the angle between the leaf's normal vector and the light direction. |
| **Alpha Cutoff** | Alpha values from the base texture smaller than the alpha cutoff are clipped creating a cutout. |
| **Shadow Strength** | Makes the shadows on the leaves less harsh. **Note:** Since it scales all the shadowing that the leaves receive, it should be used with care for trees that are e.g. in a shadow of a mountain. |
| **Shadow Offset** | Scales the values from the Shadow Offset texture set in the source material. It is used to offset the position of the leaves when collecting the shadows, so that the leaves appear as if they were not placed on one quad. It's especially important for billboarded leaves and they should have brighter values in the center of the texture and darker ones at the border. Start out with a black texture and add different shades of gray per leaf. |
| **Shadow Caster Resolution** | Defines the resolution of the texture atlas containing alpha values from source diffuse textures. The atlas is used when the leaves are rendered as shadow casters. Using lower resolution might increase performance. |

## Branches

This section focuses on explaining the specific Branch Group Properties.

## Leaves

This section focuses on explaining the specific Leaf Group Properties.

Page last updated: 2011-10-29

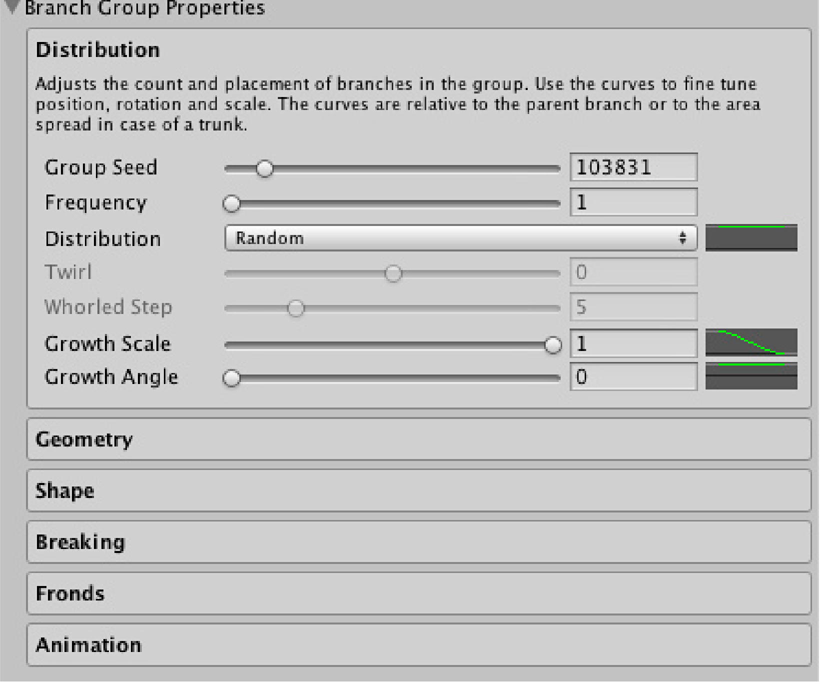
---

# tree-Branches

Branch groups node is responsible for generating branches and fronds. Its properties appear when you have selected a branch, frond or branch + frond node.

### Distribution

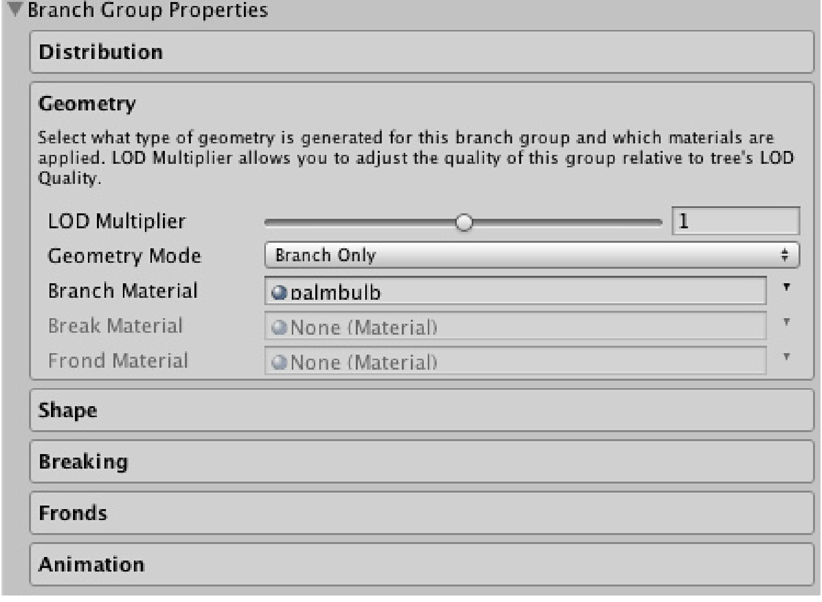
Adjusts the count and placement of branches in the group. Use the curves to fine tune position, rotation and scale. The curves are relative to the parent branch or to the area spread in case of a trunk.



| | |
|---|---|
| **Group Seed** | The seed for this group of branches. Modify to vary procedural generation. |
| **Frequency** | Adjusts the number of branches created for each parent branch. |
| **Distribution** | The way the branches are distributed along their parent. |
| **Twirl** | Defines how many nodes are in each whorled step when using Whorled distribution. For real plants this is normally a Fibonacci number. |
| **Growth Scale** | Defines the scale of nodes along the parent node. Use the curve to adjust and the slider to fade the effect in and out. |
| **Growth Angle** | Defines the initial angle of growth relative to the parent. Use the curve to adjust and the slider to fade the effect in and out. |

### Geometry

Select what type of geometry is generated for this branch group and which materials are applied. **LOD Multiplier** allows you to adjust the quality of this group relative to tree's **LOD Quality**.




| | |
|---|---|
| **LOD Multiplier** | Adjusts the quality of this group relative to tree's LOD Quality, so that it is of either higher or lower quality than the rest of the tree. |
| **Geometry Mode** | Type of geometry for this branch group: Branch Only, Branch + Fronds, Fronds Only. |
| **Branch Material** | The primary material for the branches. |
| **Break Material** | Material for capping broken branches. |
| **Frond Material** | Material for the fronds. |

### Shape

Adjusts the shape and growth of the branches. Use the curves to fine tune the shape, all curves are relative to the branch itself.

Branch Group Properties
Distribution
Geometry
Shape
Adjusts the shape and growth of the branches. Use the curves to fine tune the shape, all curves are relative to the branch itself.
Length
Relative Length
Radius 0.5
Cap Smoothing 0
Crinkliness 0.1526316
Seek Sun 0
Noise 0.1
Noise Scale U 0.2
Noise Scale V 0.1
Flare Radius 0
Flare Height 0.1
Flare Noise 0.3
Breaking
Fronds
Animation

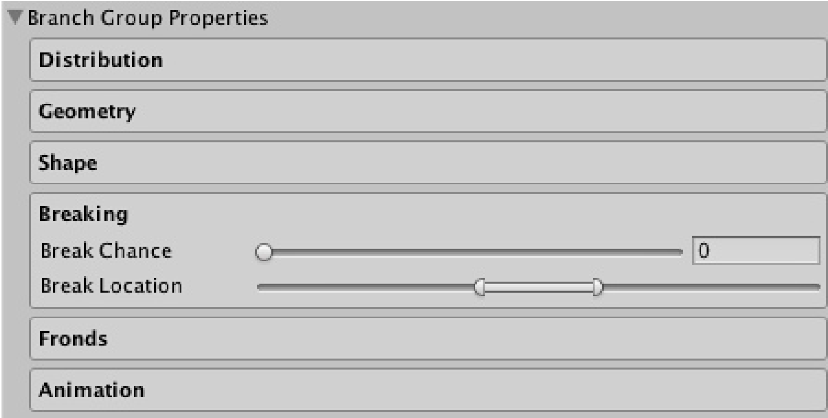
| | |
|---|---|
| **Length** | Adjusts the length of the branches. |
| **Relative Length** | Determines whether the radius of a branch is affected by its length. |
| **Radius** | Adjusts the radius of the branches, use the curve to fine-tune the radius along the length of the branches. |
| **Cap Smoothing** | Defines the roundness of the cap/tip of the branches. Useful for cacti. |
| **Growth** | **Adjusts the growth of the branches.** |
| **Crinkliness** | Adjusts how crinkly/crooked the branches are, use the curve to fine-tune. |
| **Seek Sun** | Use the curve to adjust how the branches are bent upwards/downwards and the slider to change the scale. |
| **Surface Noise** | **Adjusts the surface noise of the branches.** |
| **Noise** | Overall noise factor, use the curve to fine-tune. |
| **Noise Scale U** | Scale of the noise around the branch, lower values will give a more wobbly look, while higher values gives a more stochastic look. |
| **Noise Scale V** | Scale of the noise along the branch, lower values will give a more wobbly look, while higher values gives a more stochastic look. |
| **Flare** | **Defines a flare for the trunk.** |
| **Flare Radius** | The radius of the flares, this is added to the main radius, so a zero value means no flares. |
| **Flare Height** | Defines how far up the trunk the flares start. |
| **Flare Noise** | Defines the noise of the flares, lower values will give a more wobbly look, while higher values gives a more stochastic look. |

**These properties are for child branches only, not trunks.**

| | |
|---|---|
| **Welding** | **Defines the welding of branches onto their parent branch. Only valid for secondary branches.** |
| **Weld Length** | Defines how far up the branch the weld spread starts. |
| **Spread Top** | Weld's spread factor on the top-side of the branch, relative to it's parent branch. Zero means no spread. |
| **Spread Bottom** | Weld's spread factor on the bottom-side of the branch, relative to it's parent branch. Zero means no spread. |

### Breaking

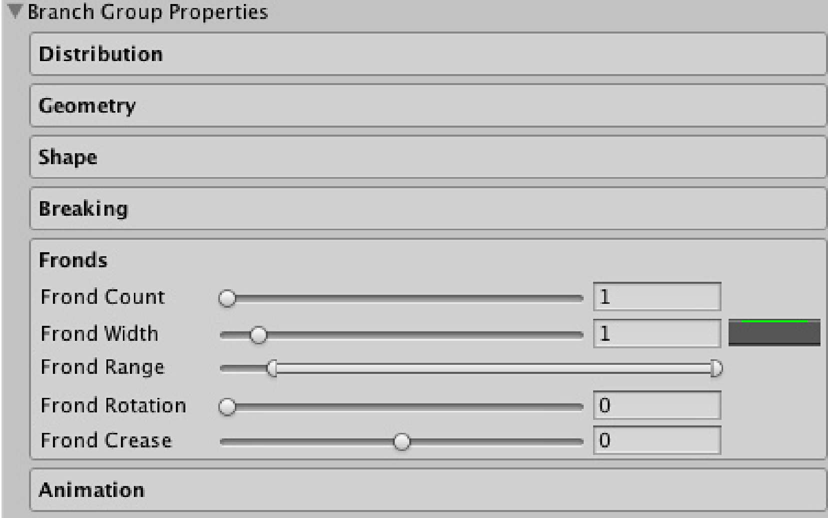
Controls the breaking of branches.




| | |
|---|---|
| **Break Chance** | Chance of a branch breaking, i.e. 0 = no branches are broken, 0.5 = half of the branches are broken, 1.0 = all the branches are broken. |
| **Break Location** | This range defines where the branches will be broken. Relative to the length of the branch. |

### Fronds

Here you can adjust the number of fronds and their properties. This tab is only available if you have Frond geometry enabled in the **Geometry** tab.




| | |
|---|---|
| **Frond Count** | Defines the number of fronds per branch. Fronds are always evenly spaced around the branch. |
| **Frond Width** | The width of the fronds, use the curve to adjust the specific shape along the length of the branch. |
| **Frond Range** | Defines the starting and ending point of the fronds. |
| **Frond Rotation** | Defines rotation around the parent branch. |
| **Frond Crease** | Adjust to crease / fold the fronds. |

### Animation

Adjusts the parameters used for animating this group of branches. The wind zones are only active in Play Mode.

Branch Group Properties
Distribution
Geometry
Shape
Breaking
Fronds
Animation
Adjusts the parameters used for animating this group of branches. The wind zones are only active in Play Mode.
Main Wind 0.5
Main Turbulence 0.5
Edge Turbulence 1
Create Wind Zone

| | |
|---|---|
| **Main Wind** | Primary wind effect. This creates a soft swaying motion and is typically the only parameter needed for primary branches. |
| **Main Turbulence** | Secondary turbulence effect. Produces more stochastic motion, which is individual per branch. Typically used for branches with fronds, such as ferns and palms. |
| **Edge Turbulence** | Turbulence along the edge of fronds. Useful for ferns, palms, etc. |
| **Create Wind Zone** | Creates a Wind Zone. |

Page last updated: 2011-10-29

---

## tree-Leaves

Leaf groups generate leaf geometry. Either from primitives or from user created meshes.

### Distribution

Adjusts the count and placement of leaves in the group. Use the curves to fine tune position, rotation and scale. The curves are relative to the parent branch.

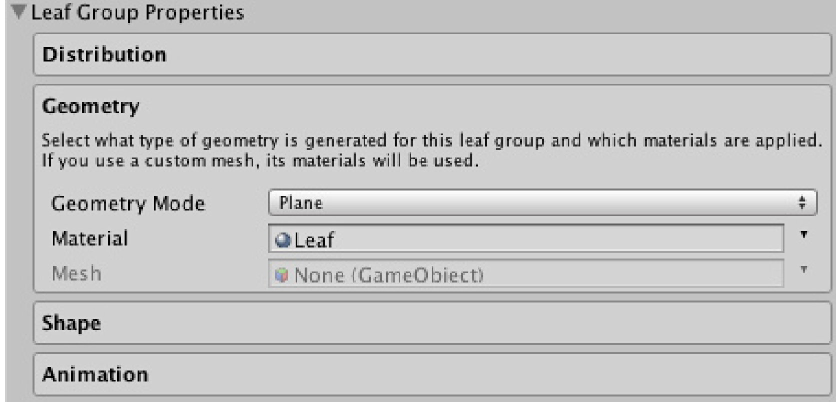
Leaf Group Properties
Distribution
Adjusts the count and placement of leaves in the group. Use the curves to fine tune position, rotation and scale. The curves are relative to the parent branch.
Group Seed 1234
Frequency 20
Distribution Random
Twirl 0
Whorled Step 5
Growth Scale 1
Growth Angle 0
Geometry
Shape
Animation

| | |
|---|---|
| **Group Seed** | The seed for this group of leaves. Modify to vary procedural generation. |
| **Frequency** | Adjusts the number of leaves created for each parent branch. |
| **Distribution** | Select the way the leaves are distributed along their parent. |

| | |
|---|---|
| **Twirl** | Twirl around the parent branch. |
| **Whorled Step** | Defines how many nodes are in each whorled step when using Whorled distribution. For real plants this is normally a Fibonacci number. |
| **Growth Scale** | Defines the scale of nodes along the parent node. Use the curve to adjust and the slider to fade the effect in and out. |
| **Growth Angle** | Defines the initial angle of growth relative to the parent. Use the curve to adjust and the slider to fade the effect in and out. |

### Geometry

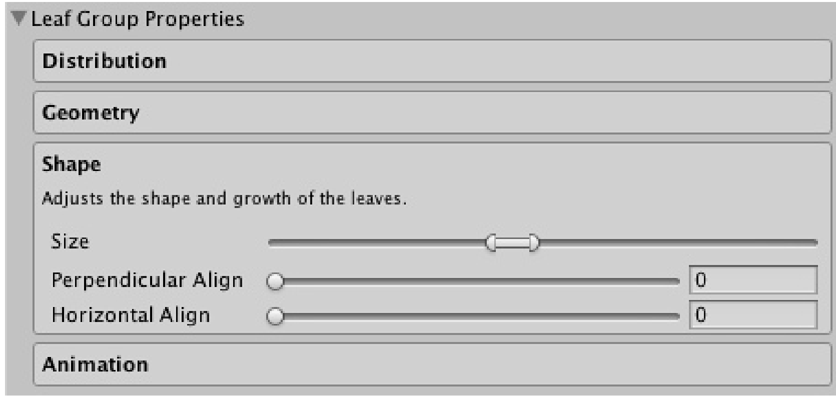
Select what type of geometry is generated for this leaf group and which materials are applied. If you use a custom mesh, its materials will be used.




| | |
|---|---|
| **Geometry Mode** | The type of geometry created. You can use a custom mesh, by selecting the Mesh option, ideal for flowers, fruits, etc. |
| **Material** | Material used for the leaves. |
| **Mesh** | Mesh used for the leaves. |

### Shape

Adjusts the shape and growth of the leaves.




| | |
|---|---|
| **Size** | Adjusts the size of the leaves, use the range to adjust the minimum and the maximum size. |
| **Perpendicular Align** | Adjusts whether the leaves are aligned perpendicular to the parent branch. |
| **Horizontal Align** | Adjusts whether the leaves are aligned horizontally. |

### Animation

Adjusts the parameters used for animating this group of leaves. Wind zones are only active in Play Mode. If you select too high values for Main Wind and Main Turbulence the leaves may float away from the branches.

Leaf Group Properties
Distribution
Geometry
Shape
Animation
Adjusts the parameters used for animating this group of leaves. Wind zones are only active in Play Mode. If you select too high values for Main Wind and Main Turbulence the leaves may float away from the branches.
Main Wind 0.5
Main Turbulence 0.5
Edge Turbulence 1
Create Wind Zone

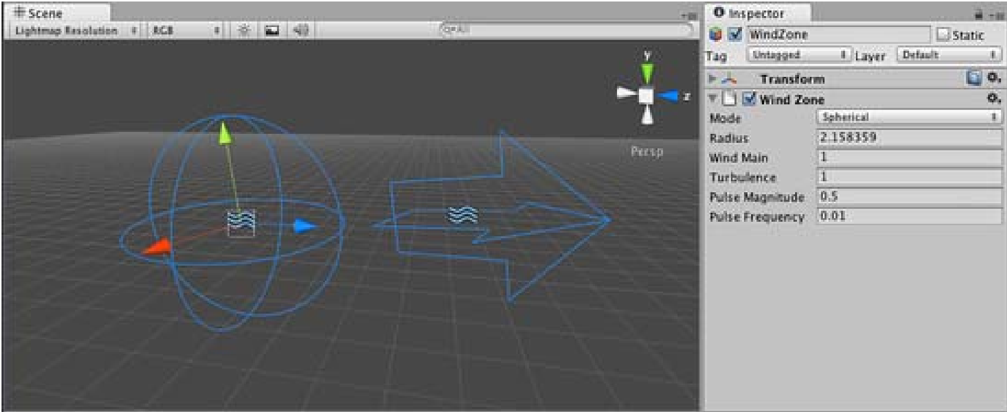
| | |
|---|---|
| **Main Wind** | Primary wind effect. Usually this should be kept as a low value to avoid leaves floating away from the parent branch. |
| **Main Turbulence** | Secondary turbulence effect. For leaves this should usually be kept as a low value. |
| **Edge Turbulence** | Defines how much wind turbulence occurs along the edges of the leaves. |

Page last updated: 2011-10-29

---

## class-WindZone

**Wind Zones** add realism to the trees you create by making them wave their branches and leaves as if blown by the wind.




*To the left a Spherical Wind Zone, to the right a Directional Wind zone.*

### Properties

| | |
|---|---|
| **Mode** | |
| **Spherical** | Wind zone only has an effect inside the radius, and has a falloff from the center towards the edge. |
| **Directional** | Wind zone affects the entire scene in one direction. |
| **Radius** | Radius of the Spherical Wind Zone (only active if the mode is set to Spherical). |
| **Wind Main** | The primary wind force. Produces a softly changing wind pressure. |
| **Turbulence** | The turbulence wind force. Produces a rapidly changing wind pressure. |
| **Pulse Magnitude** | Defines how much the wind changes over time. |
| **Pulse Frequency** | Defines the frequency of the wind changes. |

### Details

**Wind Zones** are used only by the tree creator for animating leaves and branches. This can help your scenes appear more natural and allows forces (such as explosions) within the game to look like they are interacting with the trees. For more information about how a tree works, just visit the tree class page.

### Using Wind Zones in Unity.

Using **Wind Zones** in Unity is really simple.

First of all, to create a new **wind zone** just click on **Game Object -> Create Other -> Wind Zone**.
Place the wind zone (depending on the type) near the trees created with the tree creator and watch it interact with your trees!.

**Note:** If the wind zone is Spherical you should place it so that the trees you want to blow are within the sphere's radius. If the wind zone is directional it doesn't matter where in the scene you place it.

### Hints

- To produce a softly changing general wind:
  - Create a directional wind zone.
  - Set Wind Main to 1.0 or less, depending on how powerful the wind should be.
  - Set Turbulence to 0.1.
  - Set Pulse Magnitude to 1.0 or more.
  - Set Pulse Frequency to 0.25.
- To create the effect of a helicopter passing by:
  - Create a spherical wind zone.
  - Set Radius to something that fits the size of your helicopter
  - Set Wind Main to 3.0
  - Set Turbulence to 5.0
  - Set Pulse Magnitude to 0.1
  - Set Pulse Frequency to 1.0
  - Attach the wind zone to a GameObject resembling your helicopter.
- To create the effect of an explosion:
  - Do the same as with the helicopter, but fade the Wind Main and Turbulence quickly to make the effect wear off.

Page last updated: 2011-10-29

---

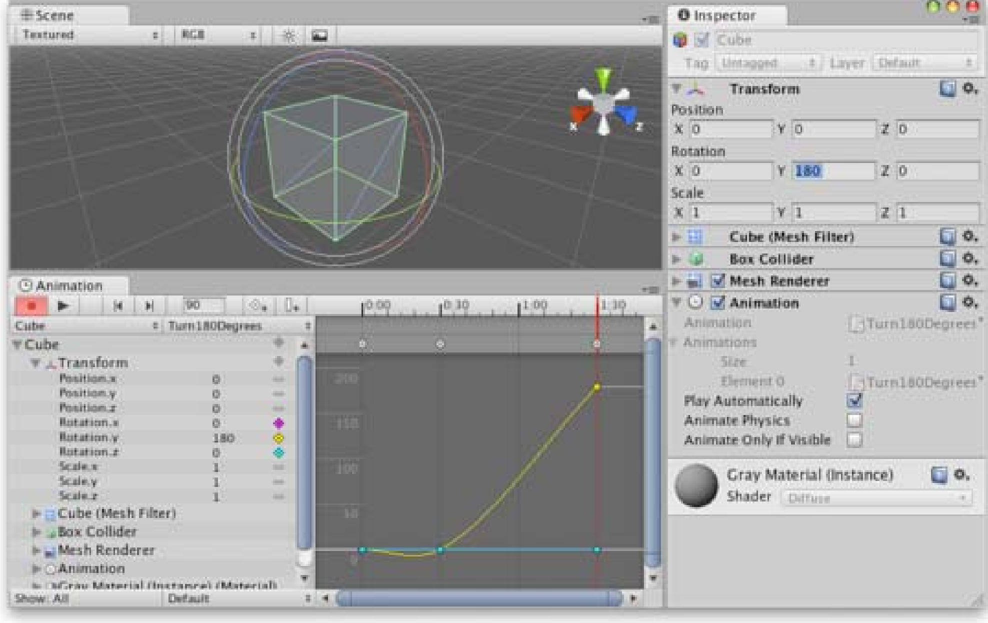
## AnimationEditorGuide

The **Animation View** in Unity allows you to create and modify **Animation Clips** directly inside Unity. It is designed to act as a powerful and straightforward alternative to external 3D animation programs. In addition to animating movement, the editor also allows you to animate variables of materials and components and augment your Animation Clips with **Animation Events**, functions that are called at specified points along the timeline.

See the pages about Animation import and Animation Scripting for further information about these subject.

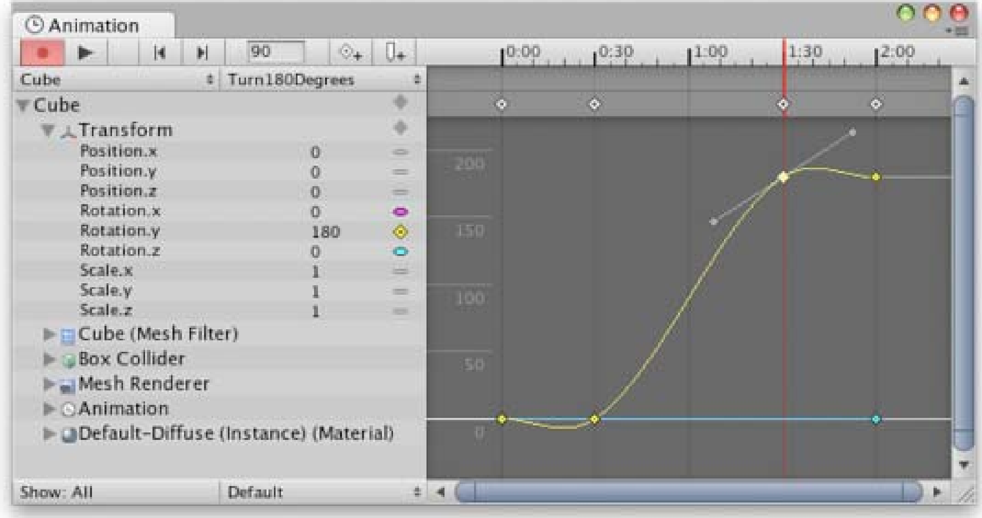
The Animation View Guide is broken up into several pages that each focus on different areas of the View:-

### Using the Animation View



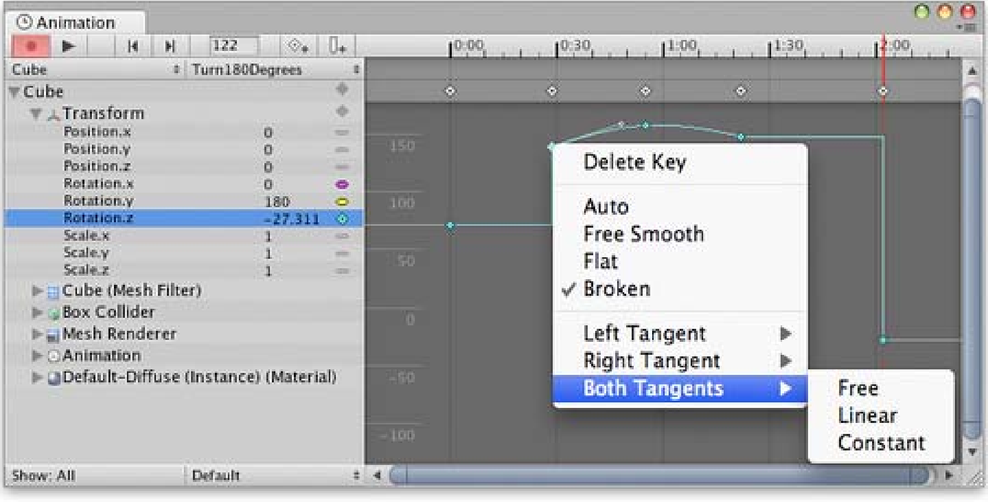

This section covers the basic operations of the **Animation View**, such as creating and editing **Animations Clips**.

## Using Animation Curves



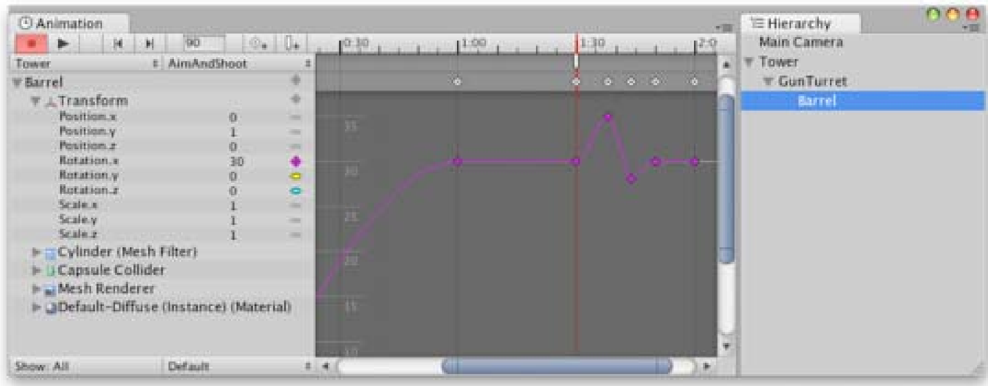

This section explains how to create **Animation Curves**, add and move **keyframes** and set WrapModes. It also offers tips for using **Animation Curves** to their full advantage.

## Editing Curves



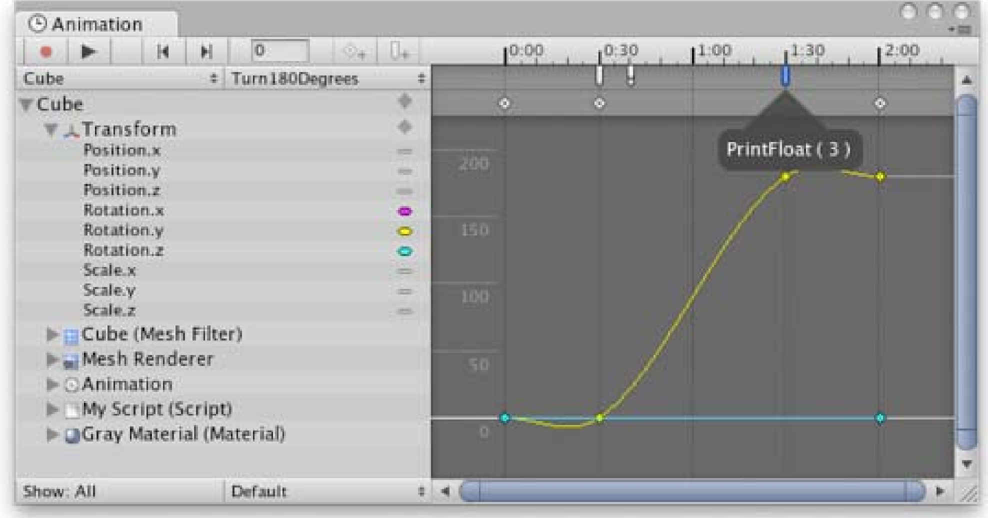

This section explains how to navigate efficienlty in the editor, create and move **keys**, and edit **tangents** and tangent types.

## Objects with Multiple Moving Parts




This section explains how to animate **Game Objects** with multiple moving parts and how to handle cases where there is more than one **Animation Component** that can control the selected **Game Object**.

## Using Animation Events

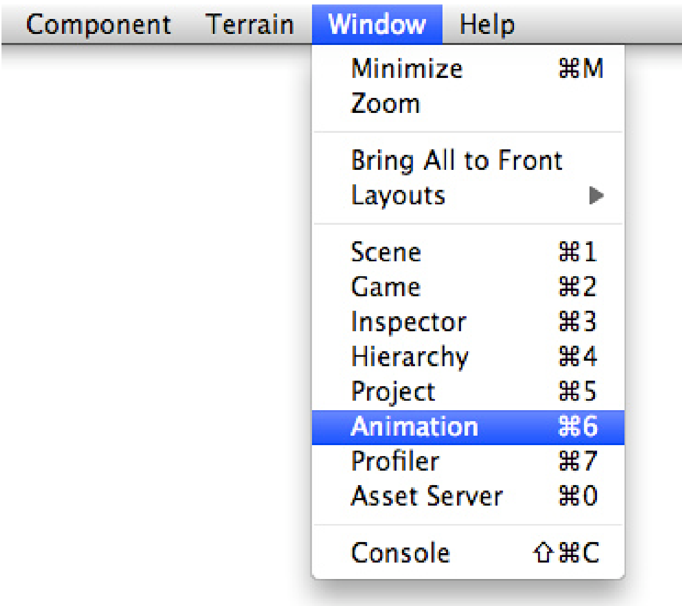



This section explains how to add **Animation Events** to an **Animation Clip**. Animation Events allow you call a script function at specified points in the animation's timeline.

Page last updated: 2012-12-04

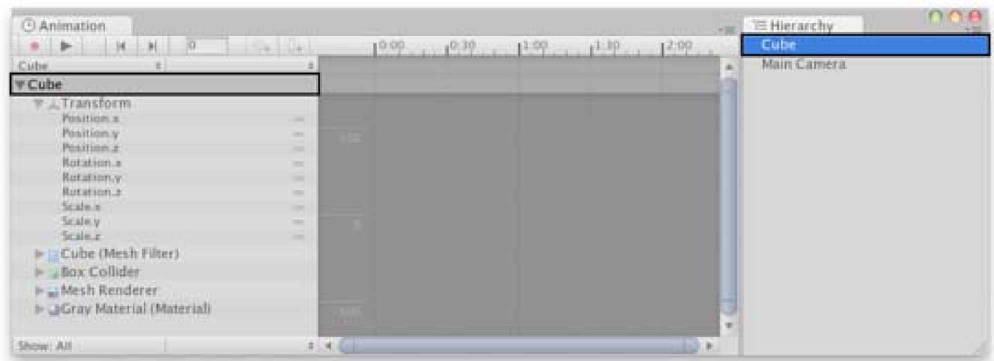
## animeditor-UsingAnimationEditor

The **Animation View** can be used to preview and edit **Animation Clips** for animated **Game Objects** in Unity. The **Animation View** can be opened from the **Window->Animation** menu.



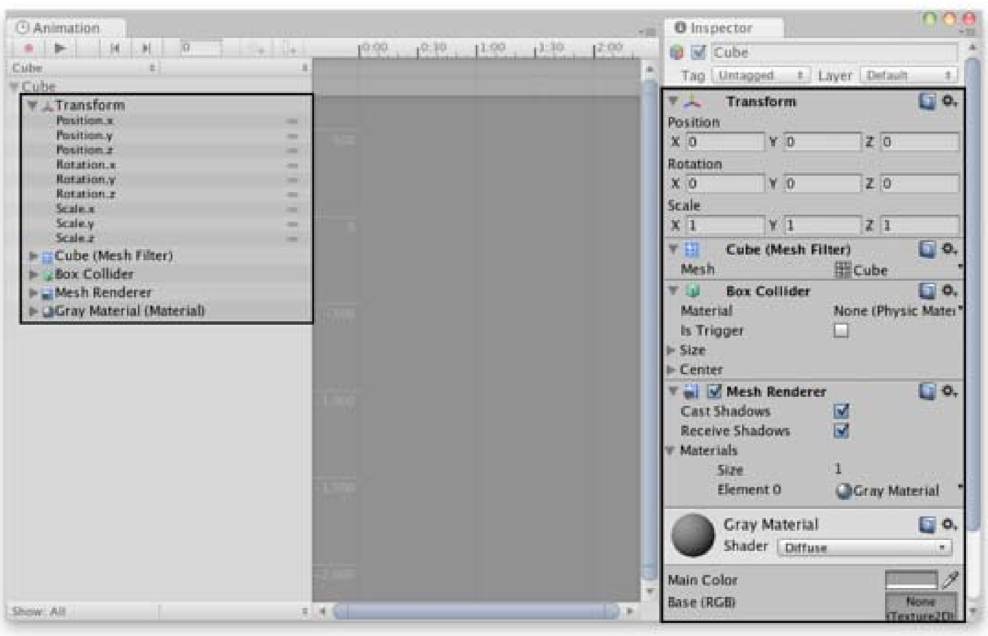

### Viewing Animations on a GameObject

The **Animation View** is tightly integrated with the **Hierarchy View**, the **Scene View**, and the **Inspector**. Like the Inspector, the Animation View will show whatever **Game Object** is selected. You can select a Game Object to view using the **Hierarchy View** or the **Scene View**. (If you select a Prefab in the **Project View** you can inspect its animation curves as well, but you have to drag the Prefab into the Scene View in order to be able to edit the curves.)




*The **Animation View** shows the **Game Object** selected in the **Hierarchy View**.*

At the left side of the **Animation View** is a hierarchical list of the animatable properties of the selected **Game Object**. The list is ordered by the **Components** and **Materials** attached to the Game Object, just like in the **Inspector**. Components and Materials can be folded and unfolded by clicking the small triangles next to them. If the selected Game Object has any child Game Objects, these will be shown after all of the Components and Materials.

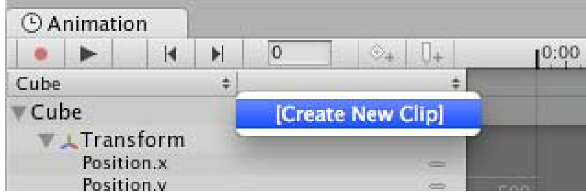


*The property list to the left in the **Animation View** shows the Components and Materials of the selected **Game Object**, just like the **Inspector**.*

### Creating a New Animation Clip

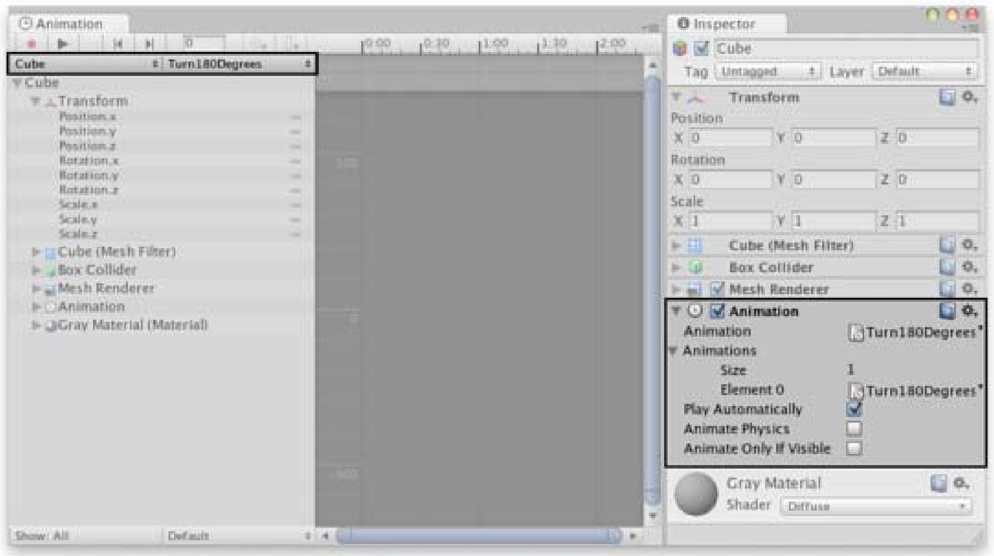
Animated **Game Objects** in Unity need an **Animation Component** that controls the animations. If a Game Object doesn't already have an Animation Component, the **Animation View** can add one for you automatically when creating a new **Animation Clip** or when entering **Animation Mode**.

To create a new **Animation Clip** for the selected **Game Object**, click the right of the two selection boxes at the upper right of the **Animation View** and select **[Create New Clip]**. You will then be prompted to save an Animation Clip somewhere in your **Assets** folder. If the Game Object doesn't have an **Animation Component** already, it will be automatically added at this point. The new **Animation Clip** will automatically be added to the list of Animations in the Animation Component.




*Create a new **Animation Clip**.*

In the **Animation View** you can always see which **Game Object** you are animating and which **Animation Clip** you are editing. There are two selection boxes in the upper left of the Animation View. The left selection box shows the **Game Object** with the **Animation Component** attached, and the right selection box shows the **Animation Clip** you are editing.



*The left selection box shows the **Game Object** with the **Animation Component** attached, and the right selection box shows the **Animation Clip** you are editing.*

### Animating a Game Object

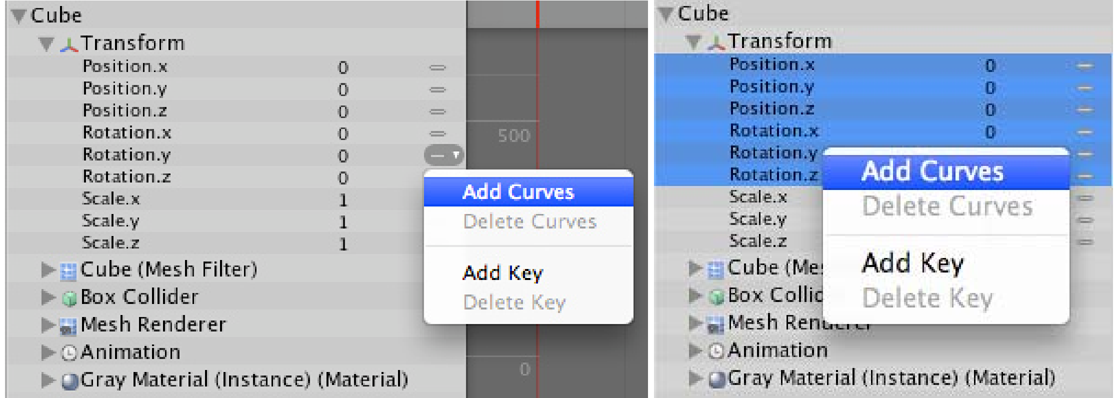
To begin editing an Animation Clip for the selected Game Object, click on the **Animation Mode button**.

Click the **Animation Mode button** to enter **Animation Mode**.

This will enter **Animation Mode**, where changes to the Game Object are stored into the **Animation Clip**. (If the Game Object doesn't have an **Animation Component** already, it will be automatically added at this point. If there is not an existing **Animation Clip**, you will be prompted to save one somewhere in your **Assets** folder.)

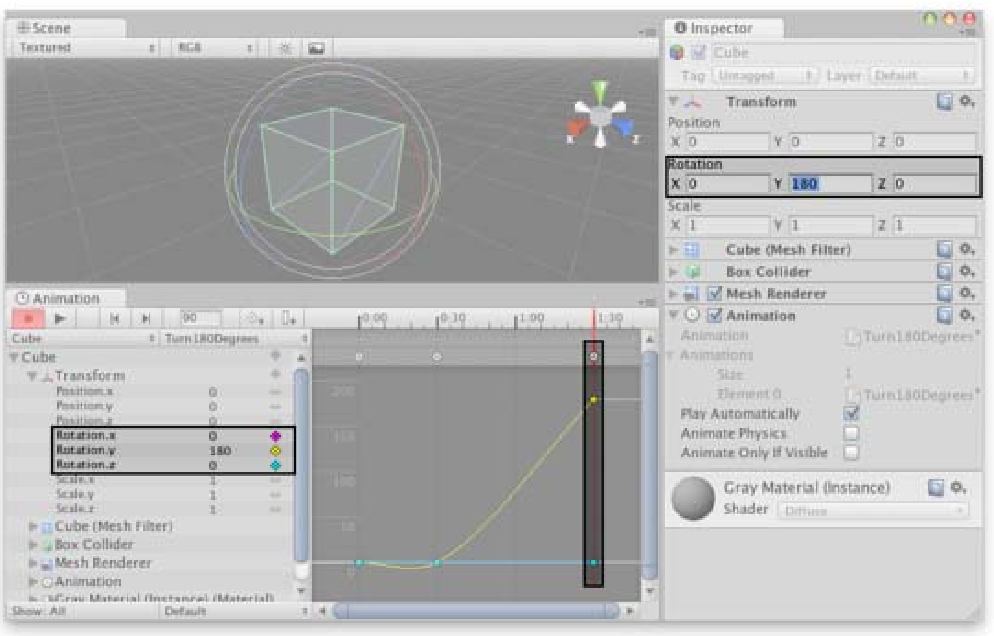
You can stop the **Animation Mode** at any time by clicking the **Animation Mode button** again. This will revert the **Game Object** to the state it was in prior to entering the Animation Mode.

You can animate any of the properties shown in the property list of the **Animation View**. To animate a property, click on the **Key Indicator** for that property and choose **Add Curve** from the menu. You can also select multiple properties and right click on the selection to add curves for all the selected properties at once. (**Transform** properties are special in that the **.x**, **.y**, and **.z** properties are linked, so that curves are added three at a time.)



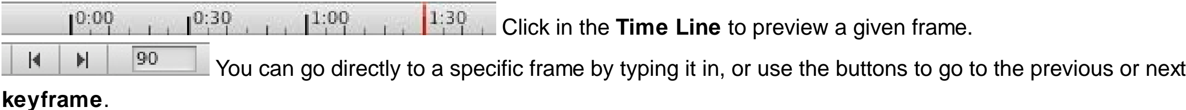

*Any property can be animated by clicking on its **Key Indicator** or by right clicking on its name. For **Transform** properties, curves for **.x**, **.y**, and **.z** are added together.*

When in **Animation Mode**, a red vertical line will show which frame of the **Animation Clip** is currently previewed. The **Inspector** and **Scene View** will show the Game Object at that frame of the Animation Clip. The values of the animated properties at that frame are also shown in a column to the right of the property names.




*In **Animation Mode** a red vertical line shows the currently previewed frame. The animated values at that frame are previewed in the **Inspector** and **Scene View** and to the right of the property names in the **Animation View**.*

You can click anywhere on the **Time Line** to preview or modify that frame in the Animation Clip. The numbers in the **Time Line** are shown as seconds and frames, so 1:30 means 1 second and 30 frames.




Click in the **Time Line** to preview a given frame.

You can go directly to a specific frame by typing it in, or use the buttons to go to the previous or next **keyframe**.

You can also use the following keyboard shortcuts to navigate between frames:

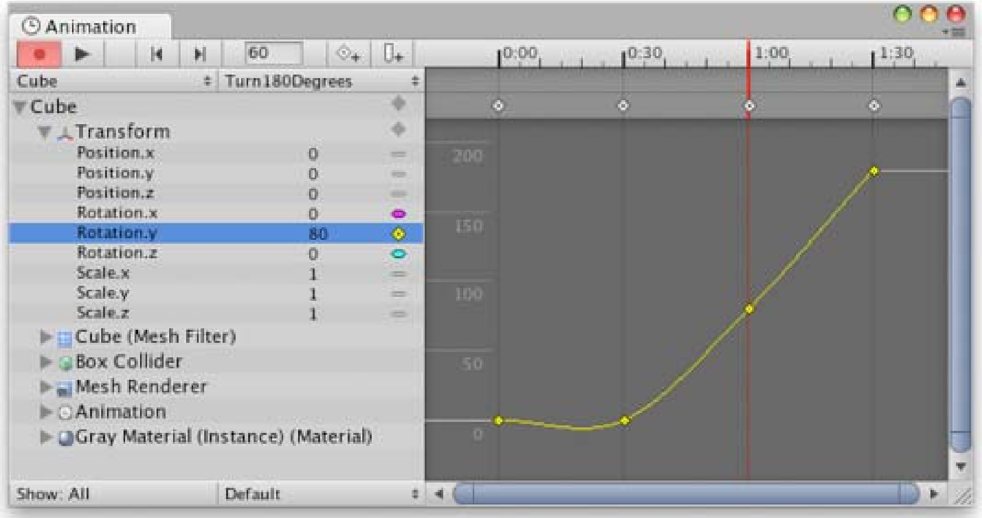
- Press **Comma** (**,**) to go to the previous frame.
- Press **Period** (**.**) to go to the next frame.
- Hold Alt and press **Comma** (**,**) to go to the previous **keyframe**.
- Hold Alt and press **Period** (**.**) to go to the next **keyframe**.

In **Animation Mode** you can move, rotate, or scale the **Game Object** in the **Scene View**. This will automatically create **Animation Curves** for the position, rotation, and scale properties of the **Animation Clip** if they didn't already exist, and keys on those Animation Curves will automatically be created at the currently previewed frame to store the respective **Transform** values you changed.

You can also use the **Inspector** to modify any of the animatable properties of the **Game Object**. This too will create **Animation Curves** as needed, and create **keys** on those Animation Curves at the currently previewed frame to store your changed values. Properties that are not animatable are grayed out in the **Inspector** while in Animation Mode.

The **Keyframe button** creates a **keyframe** for the shown curves at the currently previewed frame (shortcut: **K**).

You can also manually create a **keyframe** using the **Keyframe button**. This will create a key for all the curves that are currently shown in the **Animation View**. If you want to only show curves for a subset of the properties in the property list, you can select those properties. This is useful for selectively adding keys to specific properties only.




*When selecting a property in the property list, only the curve for that property is shown.*

### Playback

The **Animation Clip** can be played back at anytime by clicking the **Play button** in the **Animation View**.

▶ Click the **Play button** to play back the **Animation Clip**.

The playback will loop within the time range that is shown in the **Time Line**. This makes it possible to focus on refining a small part of the **Animation Clip** that is being worked on, without having to play back the entire length of the clip. To play back the whole length of the **Animation Clip**, zoom out to view the entire time range, or press **F** to Frame Select with no **keys** selected. To learn more about navigating the **Curve View**, see the section on Editing Animation Curves.

Page last updated: 2012-09-05

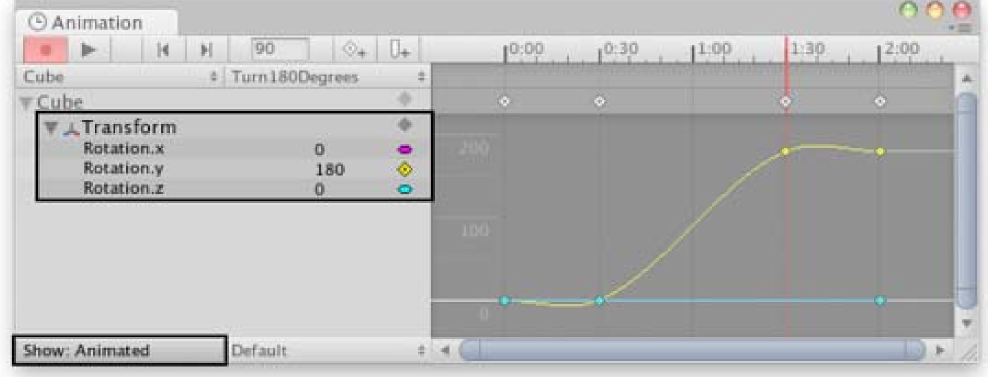
---

## animeditor-AnimationCurves

### The Property List

In an **Animation Clip**, any animatable property can have an **Animation Curve**, which means that the Animation Clip controls that property. In the property list of the **Animation View** properties with **Animation Curves** have colored curve indicators. For information on how to add curves to an animation property, see the section on Using the Animation View.

A **Game Object** can have quite a few components and the property list in the **Animation View** can get very long. To show only the properties that have **Animation Curves**, click the lower left button in the **Animation View** to set its state to **Show: Animated**.

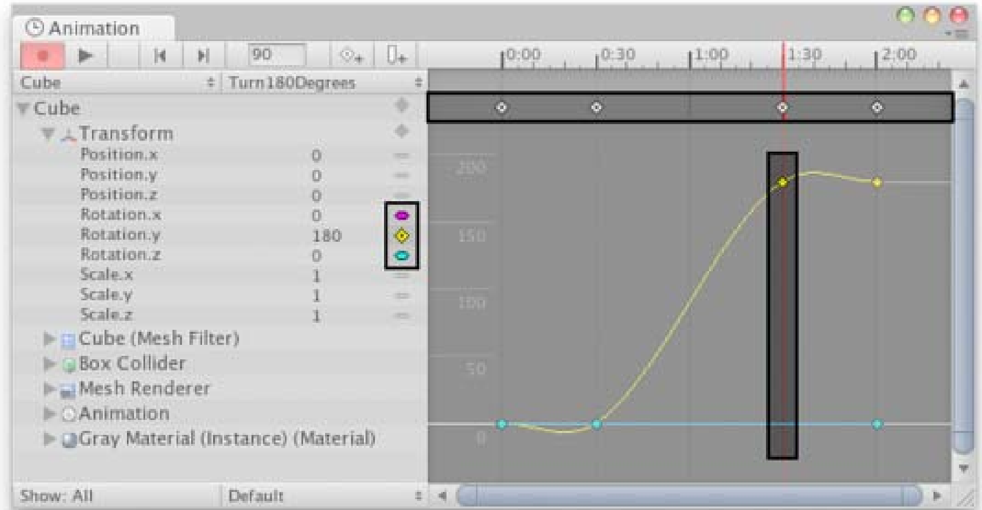



*Set the toggle button in the lower left corner to **Show: Animated** to hide all the properties without **Animation Curves** from the property list.*

### Understanding Curves, Keys and Keyframes

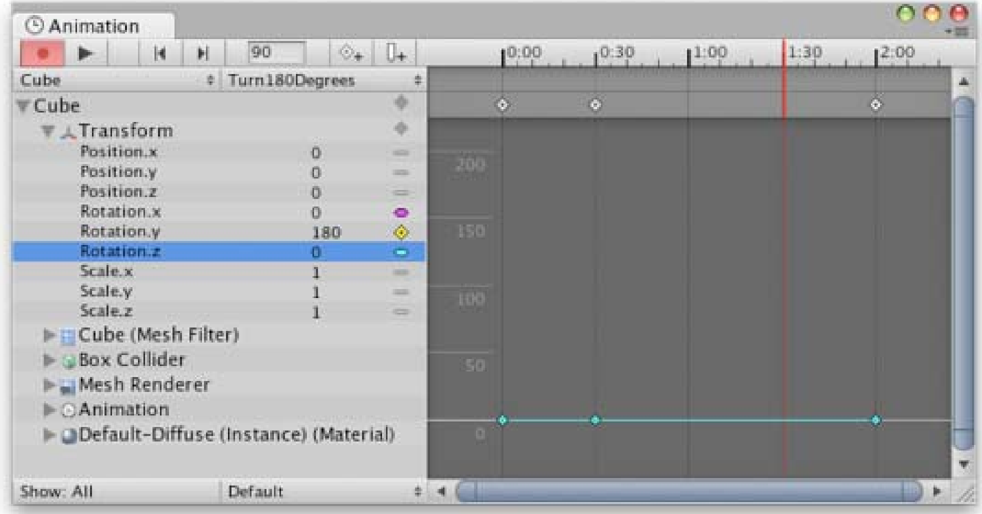
An **Animation Curve** has multiple **keys** which are control points that the curve passes through. These are visualized in the **Curve Editor** as small diamond shapes on the curves. A frame in which one or more of the shown curves have a **key** is called a **keyframe**. The **keyframes** are shown as white diamonds in the **Keyframe Line**.

If a property has a **key** in the currently previewed frame, the curve indicator will have a diamond shape.



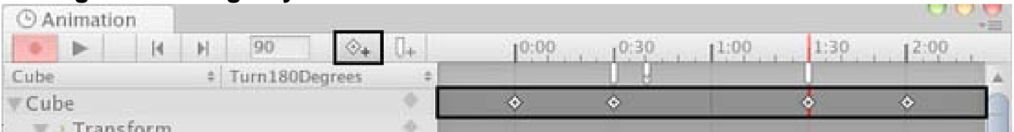

*The **Rotation.y** property has a **key** at the currently previewed frame. The **Keyframe Line** marks the **keyframes** for all shown curves.*

The **Keyframe Line** only shows keyframes for the curves that are shown. If a property is selected in the property list, only that property is shown, and the **Keyframe Line** will not mark the keys of the curves that are not shown.




*When a property is selected, other properties are not shown and the keys of their curves are not shown in the **Keyframe Line**.*

### Adding and Moving Keyframes



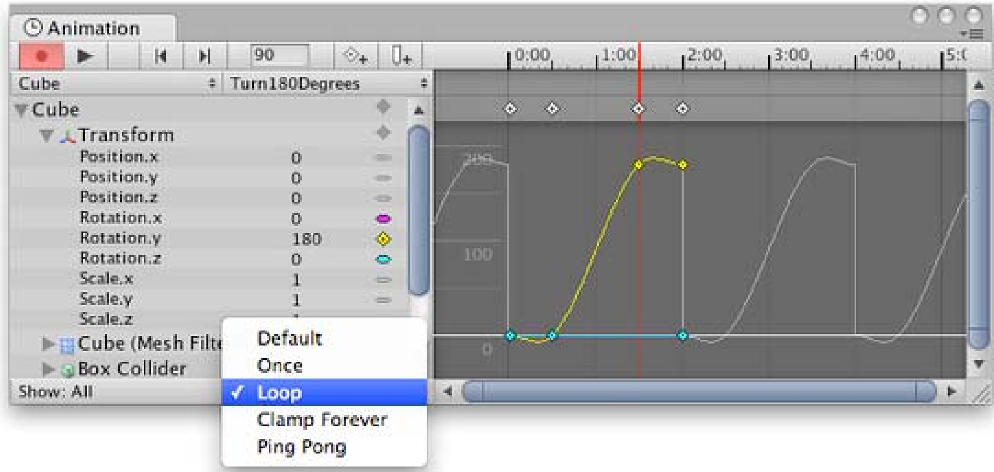

The **Keyframe Line** shows the **keyframes** of the currently shown curves. You can add a **keyframe** by double-clicking the **Keyframe Line** or by using the **Keyframe button**.

A **keyframe** can be added at the currently previewed frame by clicking the **Keyframe button** or at any given frame by double-clicking the **Keyframe Line** at the frame where the **keyframe** should be. This will add a **key** to all the shown curves at once. It is also possible to add a **keyframe** by right-clicking the **Keyframe Line** and select **Add Keyframe** from the

context menu. Once placed, **keyframes** can be dragged around with the mouse. It is also possible to select multiple **keyframes** to drag at once. **Keyframes** can be deleted by selecting them and pressing **Delete**, or by right-clicking on them and selecting **Delete Keyframe** from the context menu.

### Wrap Mode

An **Animation Clip** in Unity can have various **Wrap Modes** that can for example set the Animation Clip to loop. See WrapMode in the Scripting Reference to learn more. The Wrap Mode of an Animation Clip can be set in the **Animation View** in the lower right selection box. The **Curve View** will preview the selected **Wrap Mode** as white lines outside of the time range of the Animation Clip.




Setting the **Wrap Mode** of an **Animation Clip** will preview that Wrap Mode in the **Curve View**.

### Supported Animatable Properties

The **Animation View** can be used to animate much more than just the position, rotation, and scale of a **Game Object**. The properties of any **Component** and **Material** can be animated - even the public variables of your own scripts components. Making animations with complex visual effects and behaviors is only a matter of adding **Animation Curves** for the relevant properties.

The following types of properties are supported in the animation system:

- Float
- Color
- Vector2
- Vector3
- Vector4
- Quaternion

Arrays are not supported and neither are structs or objects other than the ones listed above.

Booleans in script components are not supported by the animation system, but booleans in certain built-in components are. For those booleans, a value of **0** equals **False** while any other value equals **True**.

Here are a few examples of the many things the **Animation View** can be used for:

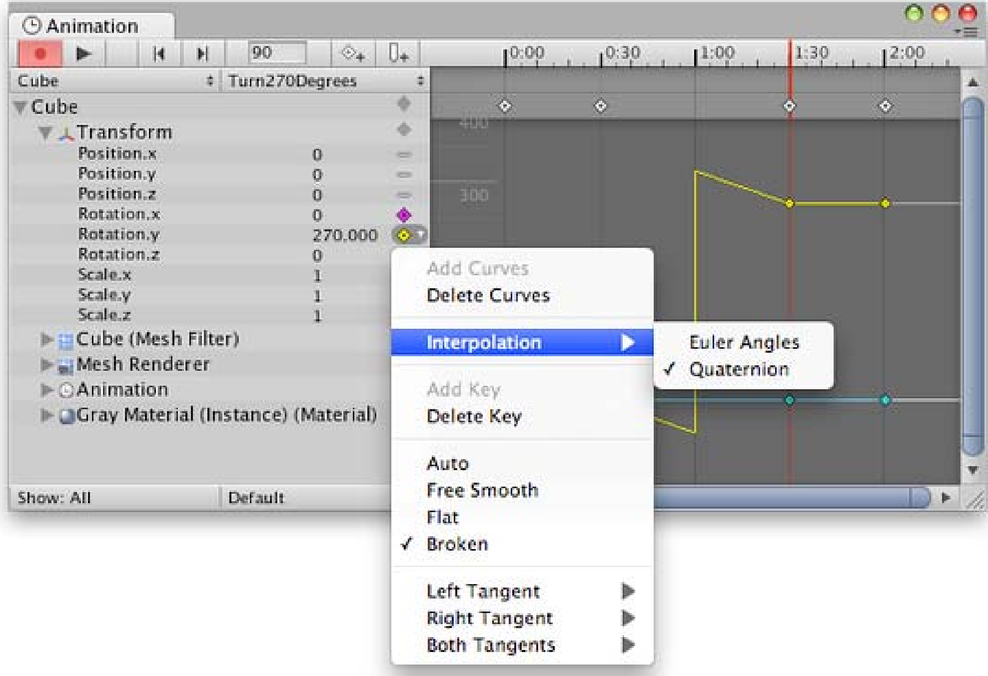
- Animate the **Color** and **Intensity** of a **Light** to make it blink, flicker, or pulsate.
- Animate the **Pitch** and **Volume** of a looping **Audio Source** to bring life to blowing wind, running engines, or flowing water while keeping the sizes of the sound assets to a minimum.
- Animate the **Texture Offset** of a **Material** to simulate moving belts or tracks, flowing water, or special effects.
- Animate the **Emit** state and **Velocities** of multiple **Ellipsoid Particle Emitters** to create spectacular fireworks or fountain displays.
- Animate variables of your own script components to make things behave differently over time.

When using **Animation Curves** to control game logic, please be aware of the way animations are played back and sampled in Unity.

### Rotation Interpolation Types

In Unity rotations are internally represented as **Quaternions**. Quaternions consist of **.x**, **.y**, **.z**, and **.w** values that should generally not be modified manually except by people who know exactly what they're doing. Instead, rotations are typically manipulated using **Euler Angles** which have **.x**, **.y**, and **.z** values representing the rotations around those three respective axes.

When interpolating between two rotations, the interpolation can either be performed on the **Quaternion** values or on the **Euler Angles** values. The **Animation View** lets you choose which form of interpolation to use when animating **Transform** rotations. However, the rotations are always shown in the form of **Euler Angles** values no matter which interpolation form is used.




*Transform rotations can use **Euler Angles** interpolation or **Quaternion** interpolation.*

**Quaternion Interpolation**

Quaternion interpolation always generates nice interpolations along the shortest path between two rotations. This avoids rotation interpolation artifacts such as Gimbal Lock. However, Quaternion interpolation cannot represent rotations larger than 180 degrees, because it is then shorter to go the other way around. If you use Quaternion interpolation and place two keys further apart than 180 degrees, the curve will look discontinuous, even though the actual rotation is still smooth - it simply goes the other way around, because it is shorter. If rotations larger than 180 degrees are desired, additional keys must be placed in between. When using Quaternion interpolation, changing the keys or tangents of one curve may also change the shapes of the other two curves, since all three curves are created from the internal Quaternion representation. When using Quaternion interpolation, keys are always linked, so that creating a key at a specific time for one of the three curves will also create a key at that time for the other two curves.

*Placing two keys 270 degrees apart when using Quaternion interpolation will cause the interpolated value to go the other way around, which is only 90 degrees.*

**Euler Angles Interpolation**

Euler Angles interpolation is what most people are used to working with. Euler Angles can represent arbitrary large rotations and the **.x**, **.y**, and **.z** curves are independent from each other. Euler Angles interpolation can be subject to artifacts such as Gimbal Lock when rotating around multiple axes at the same time, but are intuitive to work with for simple rotations around one axis at a time. When Euler Angles interpolation is used, Unity internally bakes the curves into the Quaternion representation used internally. This is similar to what happens when importing animation into Unity from external programs. Note that this curve baking may add extra keys in the process and that tangents with the **Constant** tangent type may not be completely precise at a sub-frame level.

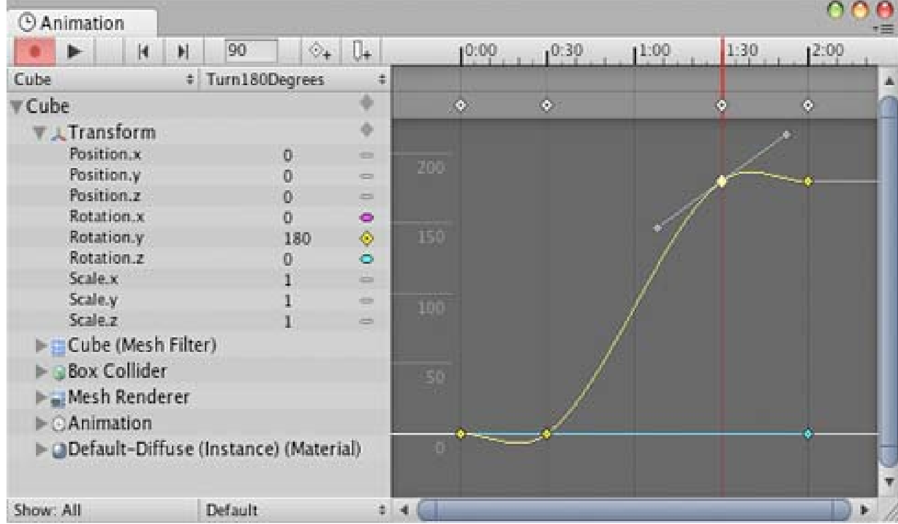
Page last updated: 2012-12-04

---

# EditingCurves

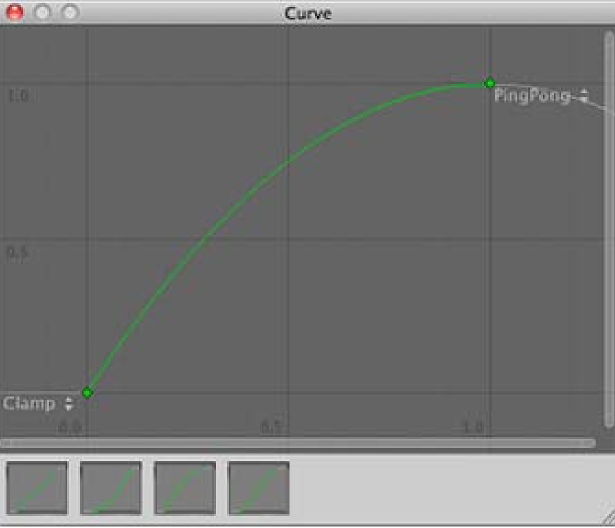
**Curves** can be used for many different things and there are several different controls in Unity that use curves that can be edited.

- The Animation View uses curves to animate properties over time in an **Animation Clip**.



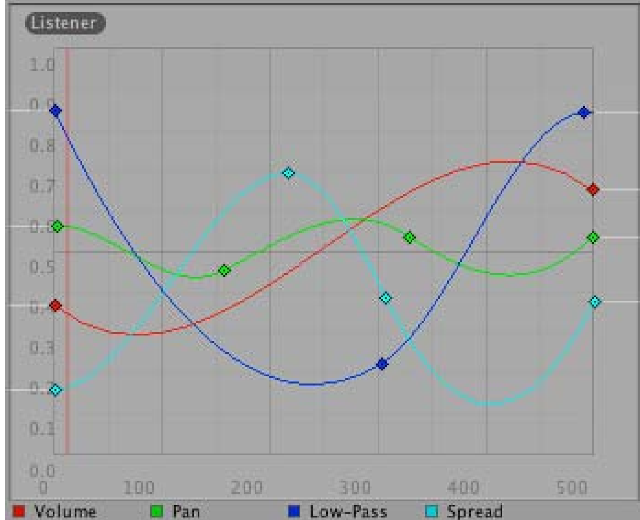

*The Animation View.*

- Script components can have member variables of type AnimationCurve that can be used for all kinds of things. Clicking on those in the Inspector will open up the **Curve Editor**.



*The Curve Editor.*

- The Audio Source component uses curves to control rolloff and other properties as a function of distance to the Audio Source.



*Distance function curves in the AudioSource component in the Inspector.*

While these controls have subtle differences, the **curves** can be edited in the exact same way in all of them. This page explains how to navigate and edit curves in those controls.

### Adding and Moving Keys on a Curve

A **key** can be added to a curve by double-clicking on the curve at the point where the **key** should be placed. It is also possible to add a **key** by right-clicking on a curve and select **Add Key** from the context menu.

Once placed, **keys** can be dragged around with the mouse:

- Click on a **key** to select it. Drag the selected **key** with the mouse.
- To snap the **key** to the grid while dragging it around, hold down **Command** on Mac / **Control** on Windows while dragging.

It is also possible to select multiple **keys** at once:

- To select multiple **keys** at once, hold down **Shift** while clicking the keys.
- To deselect a selected **key**, click on it again while holding down **Shift**.
- To select all **keys** within a rectangular area, click on an empty spot and drag to form the rectangle selection.
- The rectangle selection can also be added to existing selected keys by holding down **Shift**.

**Keys** can be deleted by selecting them and pressing **Delete**, or by right-clicking on them and selecting **Delete Key** from the context menu.

### Navigating the Curve View

When working with the **Animation View** you can easily zoom in on details of the curves you want to work with or zoom out to get the full picture.

You can always press **F** to frame-select the shown curves or selected keys in their entirely.

**Zooming**
You can **zoom** the Curve View using the scroll-wheel of your mouse, the zoom functionality of your trackpad, or by holding **Alt** while right-dragging with your mouse.

You can zoom on only the horizontal or vertical axis:

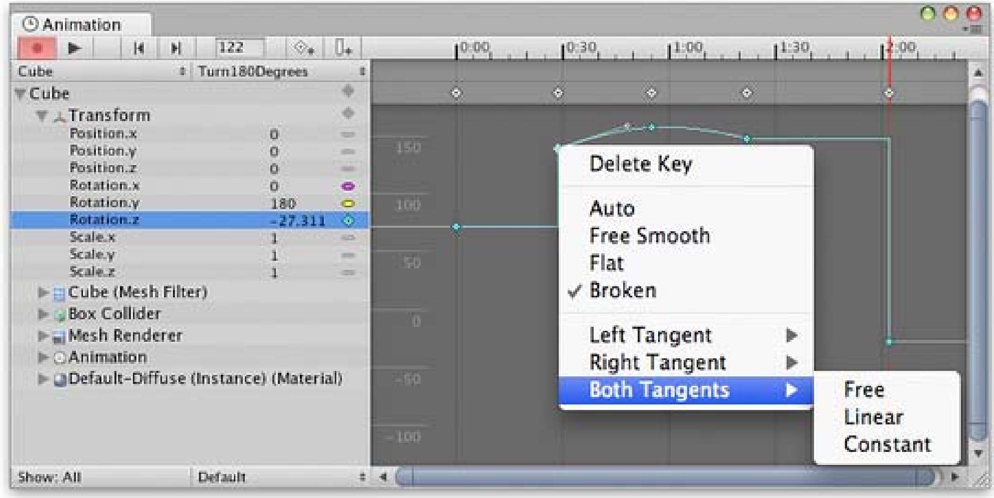
- **zoom** while holding down **Command** on Mac / **Control** on Windows to zoom horizontally.
- **zoom** while holding down **Shift** to zoom vertically.

Furthermore, you can drag the end caps of the scrollbars to shrink or expand the area shown in the Curve View.

**Panning**
You can **pan** the Curve View by middle-dragging with your mouse or by holding **Alt** while left-dragging with your mouse.

## Editing Tangents

A **key** has two **tangents** - one on the left for the ingoing slope and one on the right for the outgoing slope. The tangents control the shape of the curve between the keys. The **Animation View** have multiple tangent types that can be used to easily control the curve shape. The tangent types for a **key** can be chosen by right-clicking the key.

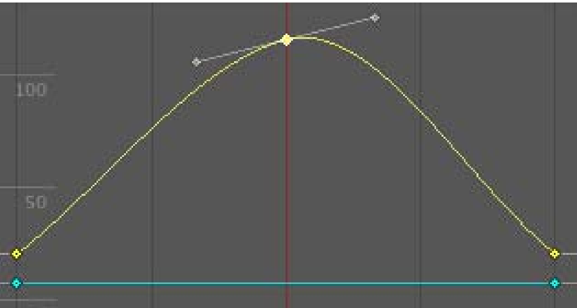



*Right-click a **key** to select the tangent type for that key.*

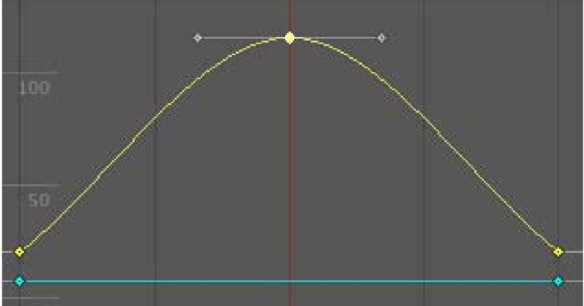
In order for animated values to change smoothly when passing a key, the left and right tangent must be co-linear. The following tangent types ensure smoothness:

- **Auto**: The tangents are automatically set so make the curve go smoothly through the key.

- **Free Smooth**: The tangents can be freely set by dragging the tangent handles. They are locked to be co-linear to ensure smoothness.

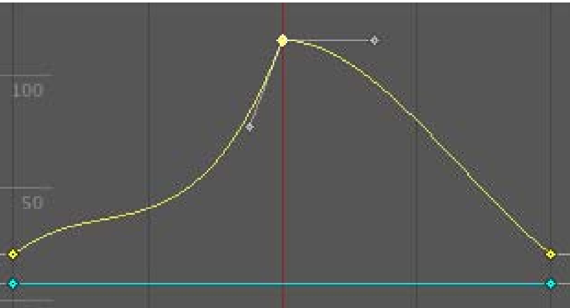



- **Flat**: The tangents are set to be horizontal. (This is a special case of **Free Smooth**.)

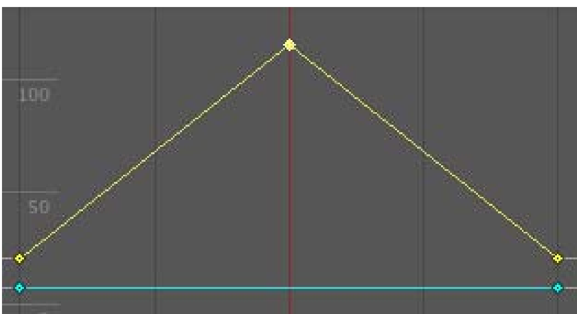



Sometimes smoothness is not desired. The left and right tangent can be set individually when the tangents are **Broken**. The left and right tangent can each be set to one of the following tangent types:

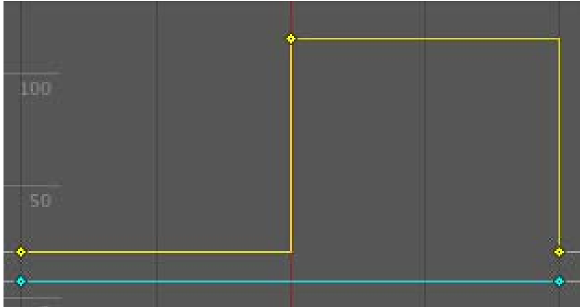
- **Free**: The tangent can be freely set by dragging the tangent handle.




- **Linear**: The tangent points towards the neighboring key. A linear curve segment can be made by setting the tangents at both ends to be **Linear**.




- **Constant**: The curve retains a constant value between two keys. The value of the left key determines the value of the curve segment.




Page last updated: 2012-12-04

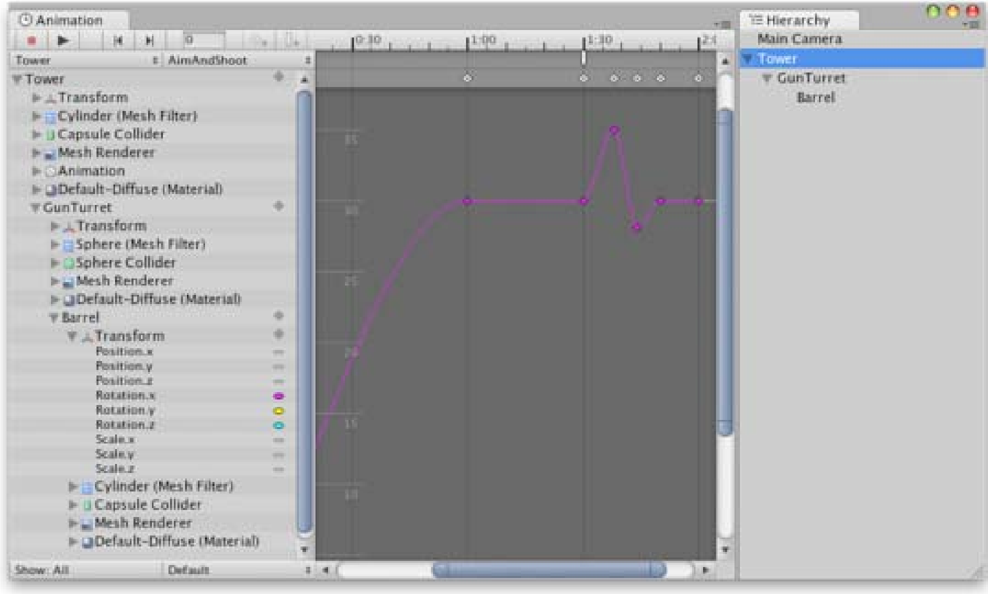
## animeditor-MultipleParts

You may want to animate **Game Objects** that have multiple moving parts, such as a gun turret with a moving barrel, or a character with many body parts. All the parts can be animated by a single Animation component on the parent, although it is useful to have additional Animation components on the children in some cases.

### Animating Child Game Objects

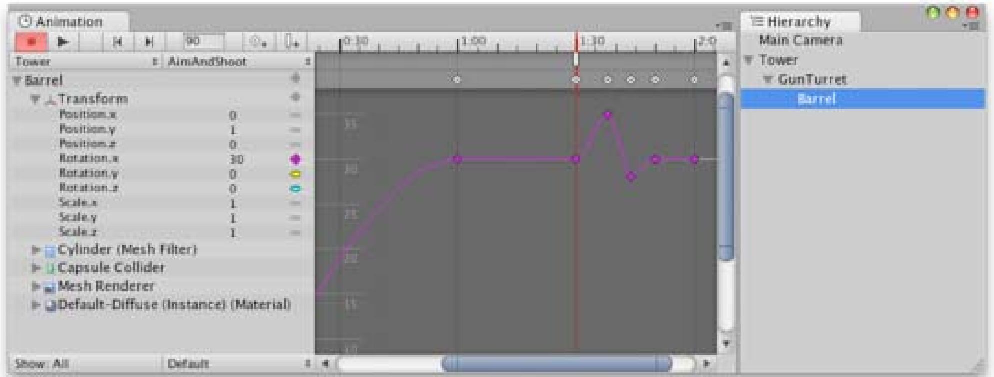
The Game Object hierarchy is shown in the panel to the left of the **Animation View**.

You can access the children of a Game Object by using the foldout triangle next to the object's name. The properties of child objects can be animated just like those of the parent.




*Child **Game Objects** can be folded out in the **Animation View**.*

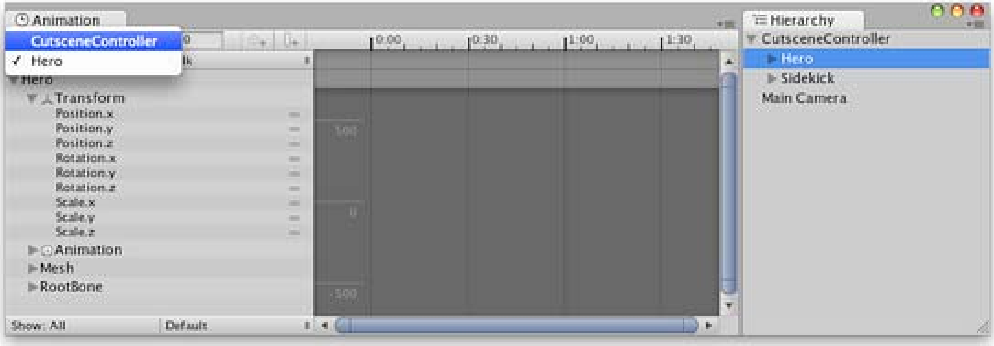
Alternatively you can select just the child Game Object you want to animate from the Hierarchy panel or the scene view. When you do this, only the child object is shown in the property list, although the animation data is still handled by the Animation component on the parent.




*The child **Game Objects** selected in the **Hierarchy View** are shown in the **Animation View**.*

### Handling Multiple Animation Components

If both a parent object and one of its children both have an Animation component then either component can animate the child object. The property list can be used to select which one you want to use.




*Select the Animation component you want to edit from the property list*

As an example, you may have a multiple characters (a hero and sidekick, say) that each have their own Animation component. You could have another Game Object in the same scene whose Animation component is used for controlling cutscenes. The cutscene might involve both the hero and sidekick characters walking around, which could be achieved by animating their positions from the cutscene controller. However, both characters would need to be children of the cutscene object to be controlled by its Animation component.

Page last updated: 2011-11-16

---

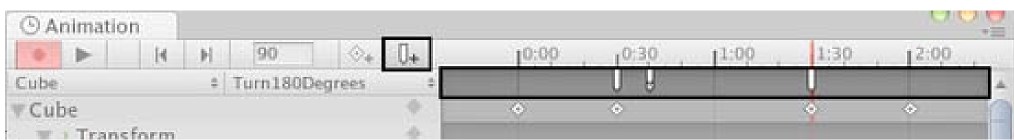
## animeditor-AnimationEvents

The power of animation clips can be increased by using **Animation Events**, which allow you to call functions in the object's script at specified points in the timeline.

The function called by an animation event can optionally take one parameter. The parameter can be a float, string, int, object reference or an AnimationEvent object. The AnimationEvent object has member variables that allow a float, string, integer and object reference to be passed into the function all at once, along with other information about the event that triggered the function call.

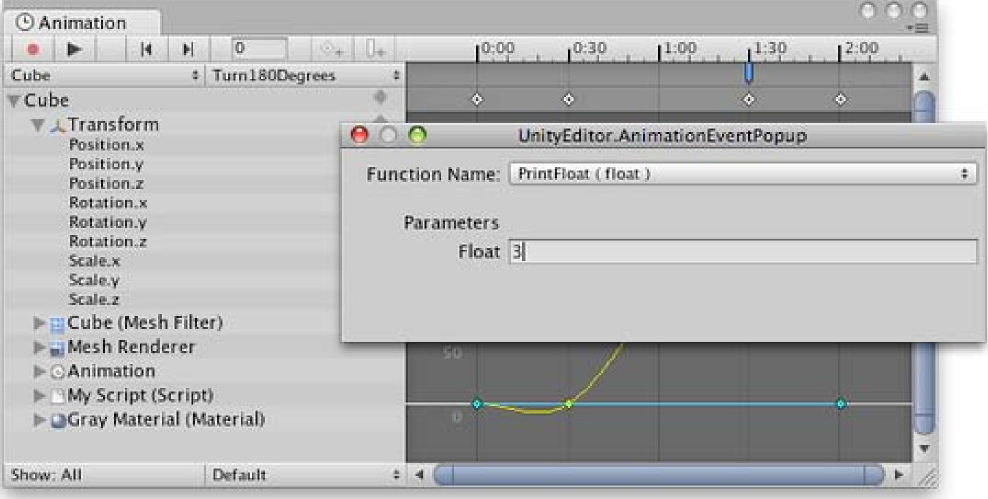
```
// This JavaScript function can be called by an Animation Event
function PrintFloat (theValue : float) {
	Debug.Log ("PrintFloat is called with a value of " + theValue);
}
```

You can add an animation event to a clip at the current playhead position by clicking the **Event button** or at any point in the animation by double-clicking the **Event Line** at the point where you want the event to be triggered. Once added, an event can be repositioned by dragging with the mouse. You can delete an event by selecting it and pressing **Delete**, or by right-clicking on it and selecting **Delete Event** from the contextual menu.



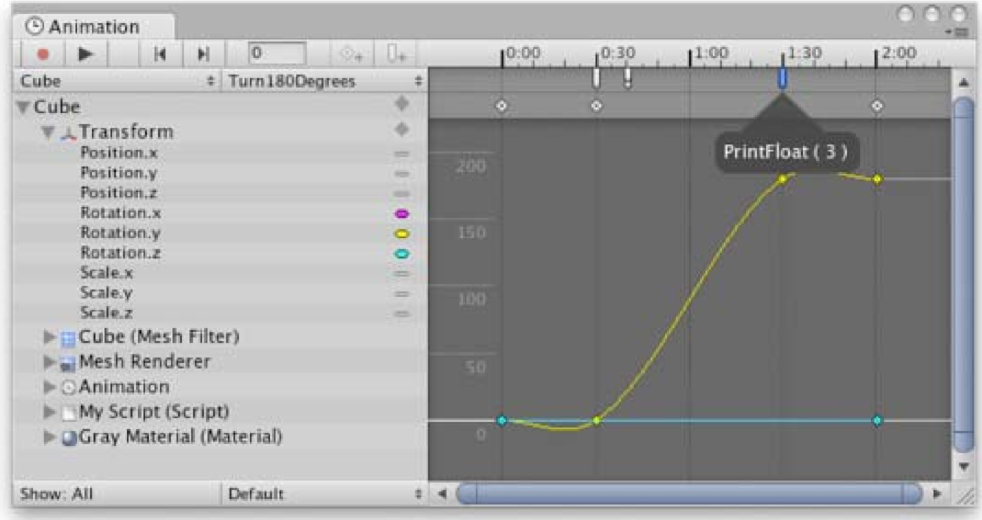

***Animation Events** are shown in the **Event Line**. Add a new **Animation Event** by double-clicking the **Event Line** or by using the **Event button**.*

When you add an event, a dialog box will appear to prompt you for the name of the function and the value of the parameter you want to pass to it.




*The **Animation Event** popup dialog lets you specify which function to call with which parameter value.*

The events added to a clip are shown as markers in the event line. Holding the mouse over a marker will show a tooltip with the function name and parameter value.




*Holding the mouse cursor over an **Animation Event marker** will show which function it calls as well as the parameter value.*

Page last updated: 2011-11-16

---

# GUI Scripting Guide

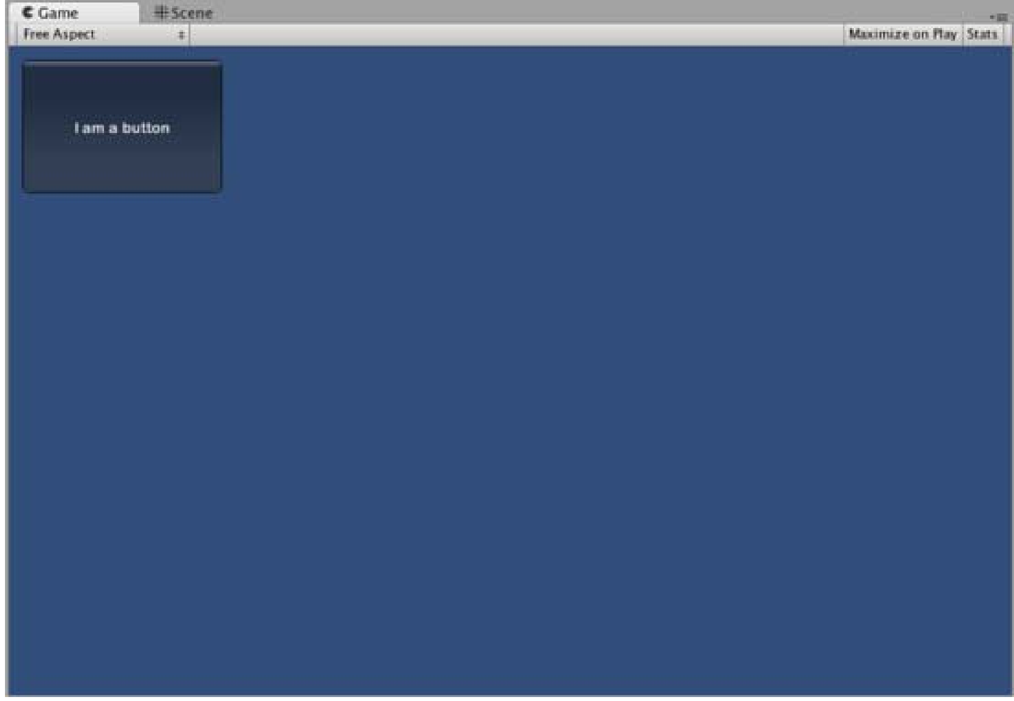
## Overview

UnityGUI allows you to create a wide variety of highly functional GUIs very quickly and easily. Rather than creating a GUI object, manually positioning it, and then writing a script that handles its functionality, you can do everything at once with just a few lines of code. The code produces **GUI controls** that are instantiated, positioned and handled with a single function call.

For example, the following code will create and handle a button with no additional work in the editor or elsewhere:-

```
// JavaScript
function OnGUI () {
	if (GUI.Button (Rect (10,10,150,100), "I am a button")) {
		print ("You clicked the button!");
	}
```

```
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	void OnGUI () {
		if (GUI.Button (new Rect (10,10,150,100), "I am a button")) {
			print ("You clicked the button!");
		}
	}
}
```




*This is the button created by the code above*

Although this example is very simple, there are very powerful and complex techniques available for use in UnityGUI. GUI construction is a broad subject but the following sections should help you get up to speed as quickly as possible. This guide can be read straight through or used as reference material.

### UnityGUI Basics

This section covers the fundamental concepts of UnityGUI, giving you an overview as well as a set of working examples you can paste into your own code. UnityGUI is very friendly to play with, so this is a good place to get started.

### Controls

This section lists every available Control in UnityGUI, along with code samples and images showing the results.

### Customization

It is important to be able to change the appearance of the GUI to match the look of your game. All controls in UnityGUI can be customized with **GUIStyles** and **GUISkins**, as explained in this section.

### Layout Modes

UnityGUI offers two ways to arrange your GUIs: you can manually place each control on the screen, or you can use an automatic layout system which works in a similar way to HTML tables. Either system can be used as desired and the two can be freely mixed. This section explains the functional differences between the two systems, including examples.

### Extending UnityGUI

UnityGUI is very easy to extend with new Control types. This chapter shows you how to make simple *compound* controls - complete with integration into Unity's event system.

### Extending Unity Editor

The GUI of the Unity editor is actually written using UnityGUI. Consequently, the editor is highly extensible using the same type of code you would use for in-game GUI. In addition, there are a number of Editor-specific GUI controls to help you create custom editor GUI.

Page last updated: 2012-01-17

---

# gui-Basics

This section will explain the bare necessities for scripting **Controls** with **UnityGUI**.

### Making Controls with UnityGUI

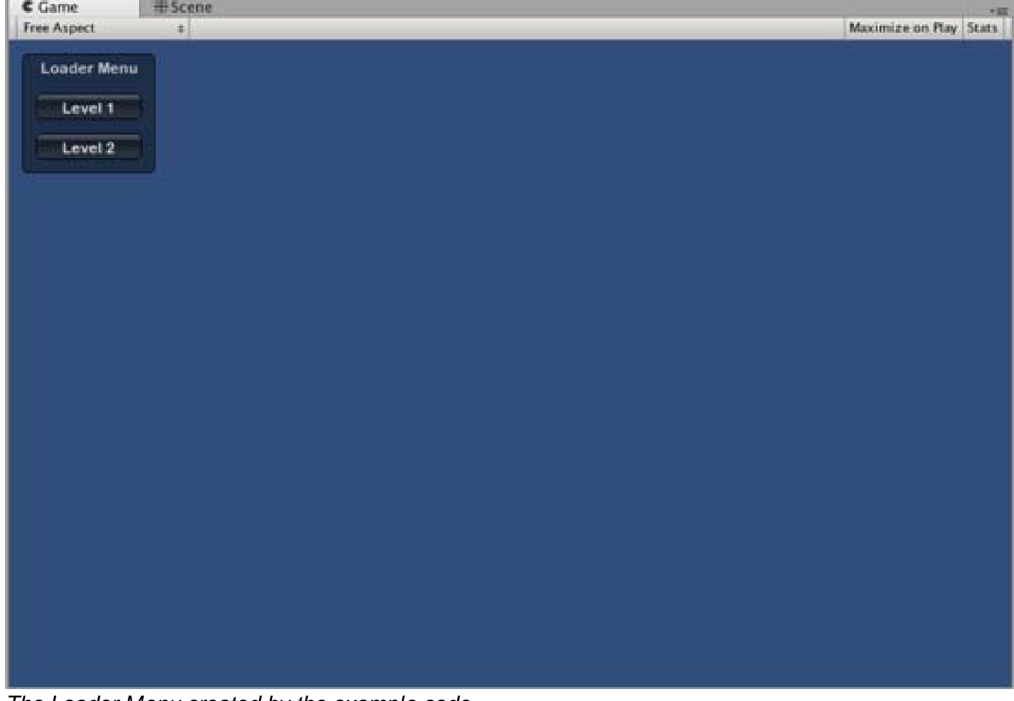
UnityGUI controls make use of a special function called **OnGUI()**. The **OnGUI()** function gets called every frame as long as the containing script is enabled - just like the **Update()** function.

GUI controls themselves are very simple in structure. This structure is evident in the following example.

```
/* Example level loader */


// JavaScript
function OnGUI () {
	// Make a background box
	GUI.Box (Rect (10,10,100,90), "Loader Menu");

	// Make the first button. If it is pressed, Application.Loadlevel (1) will be executed
	if (GUI.Button (Rect (20,40,80,20), "Level 1")) {
		Application.LoadLevel (1);
	}

	// Make the second button.
	if (GUI.Button (Rect (20,70,80,20), "Level 2")) {
		Application.LoadLevel (2);
	}
}


//C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	void OnGUI () {
		// Make a background box
		GUI.Box(new Rect(10,10,100,90), "Loader Menu");

		// Make the first button. If it is pressed, Application.Loadlevel (1) will be executed
		if(GUI.Button(new Rect(20,40,80,20), "Level 1")) {
			Application.LoadLevel(1);
		}

		// Make the second button.
```

```
        if(GUI.Button(new Rect(20,70,80,20), "Level 2")) {
            Application.LoadLevel(2);
        }
    }
}
```

This example is a complete, functional level loader. If you copy/paste this script and attach it a **GameObject**, you'll see the following menu appear in when you enter **Play Mode**:




*The Loader Menu created by the example code*

Let's take a look at the details of the example code:

The first GUI line, **GUI.Box (Rect (10,10,100,90), "Loader Menu");** displays a **Box** Control with the header text "Loader Menu". It follows the typical GUI Control declaration scheme which we'll explore momentarily.

The next GUI line is a **Button** Control declaration. Notice that it is slightly different from the Box Control declaration. Specifically, the entire Button declaration is placed inside an **if** statement. When the game is running and the Button is clicked, this **if** statement returns true and any code inside the **if** block is executed.

Since the **OnGUI()** code gets called every frame, you don't need to explicitly create or destroy GUI controls. The line that declares the Control is the same one that creates it. If you need to display Controls at specific times, you can use any kind of scripting logic to do so.

```
/* Flashing button example */



// JavaScript
function OnGUI () {
    if (Time.time % 2 < 1) {
        if (GUI.Button (Rect (10,10,200,20), "Meet the flashing button")) {
            print ("You clicked me!");
        }
    }
}
```

```
// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	void OnGUI () {
		if (Time.time % 2 < 1) {
			if (GUI.Button (new Rect (10,10,200,20), "Meet the flashing button")) {
				print ("You clicked me!");
			}
		}
	}
}
```

Here, **GUI.Button()** only gets called every other second, so the button will appear and disappear. Naturally, the user can only click it when the button is visible.

As you can see, you can use any desired logic to control when GUI Controls are displayed and functional. Now we will explore the details of each Control's declaration.

## Anatomy of a Control

There are three key pieces of information required when declaring a GUI Control:

**Type** (**Position**, **Content**)

Observe that this structure is a function with two arguments. We'll explore the details of this structure now.

#### Type

**Type** is the **Control Type**, and is declared by calling a function in Unity's GUI class or the GUILayout class, which is discussed at length in the Layout Modes section of the Guide. For example, **GUI.Label()** will create a non-interactive label. All the different control types are explained later, in the Controls section of the Guide.

#### Position

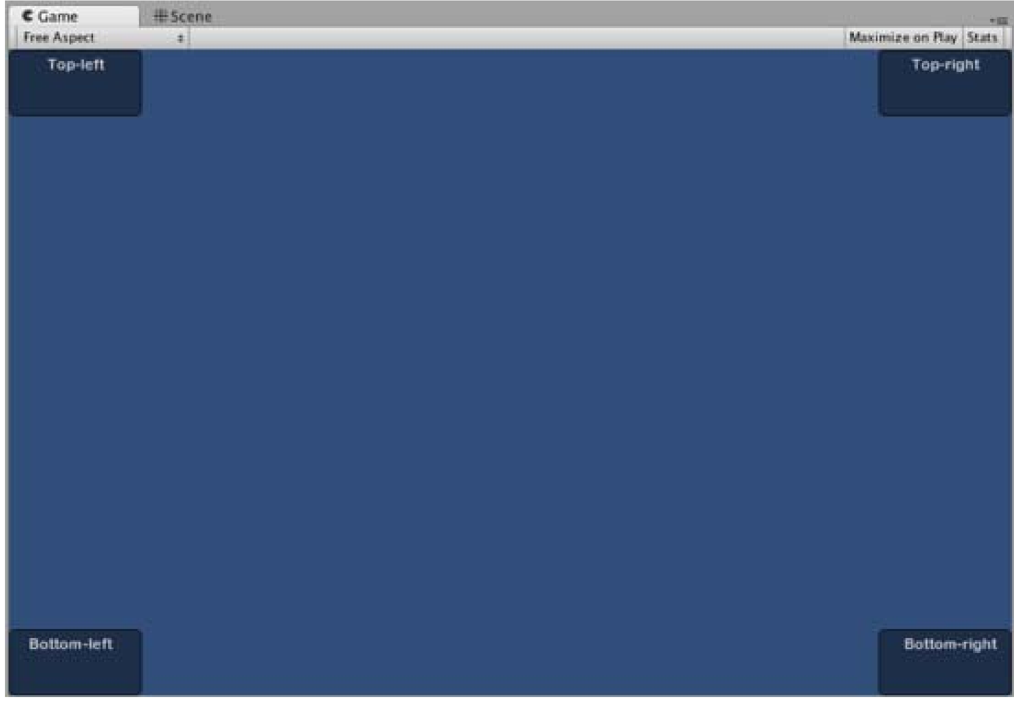
The **Position** is the first argument in any **GUI** Control function. The argument itself is provided with a **Rect()** function. **Rect()** defines four properties: **left-most position**, **top-most position**, **total width**, **total height**. All of these values are provided in **integers**, which correspond to pixel values. All UnityGUI controls work in **Screen Space**, which is the resolution of the published player in pixels.

The coordinate system is top-left based. **Rect(10, 20, 300, 100)** defines a Rectangle that starts at coordinates: 10,20 and ends at coordinates 310,120. It is worth repeating that the second pair of values in **Rect()** are total width and height, not the coordinates where the controls end. This is why the example mentioned above ends at 310,120 and not 300,100.

You can use the **Screen.width** and **Screen.height** properties to get the total dimensions of the screen space available in the player. The following example may help clarify how this is done:

```
/* Screen.width & Screen.height example */


// JavaScript
function OnGUI () {
	GUI.Box (Rect (0,0,100,50), "Top-left");
	GUI.Box (Rect (Screen.width - 100,0,100,50), "Top-right");
	GUI.Box (Rect (0,Screen.height - 50,100,50), "Bottom-left");
	GUI.Box (Rect (Screen.width - 100,Screen.height - 50,100,50), "Bottom-right");
}


// C#
```

```
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	void OnGUI(){
		GUI.Box (new Rect (0,0,100,50), "Top-left");
		GUI.Box (new Rect (Screen.width - 100,0,100,50), "Top-right");
		GUI.Box (new Rect (0,Screen.height - 50,100,50), "Bottom-left");
		GUI.Box (new Rect (Screen.width - 100,Screen.height - 50,100,50), "Bottom-right");
	}

}
```



*The Boxes positioned by the above example*

**Content**

The second argument for a GUI Control is the actual content to be displayed with the Control. Most often you will want to display some text or an image on your Control. To display text, pass a string as the Content argument like this:

```
/* String Content example */


// JavaScript
function OnGUI () {
	GUI.Label (Rect (0,0,100,50), "This is the text string for a Label Control");
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	void OnGUI () {
		GUI.Label (new Rect (0,0,100,50), "This is the text string for a Label Control");
	}
```

```
}
```

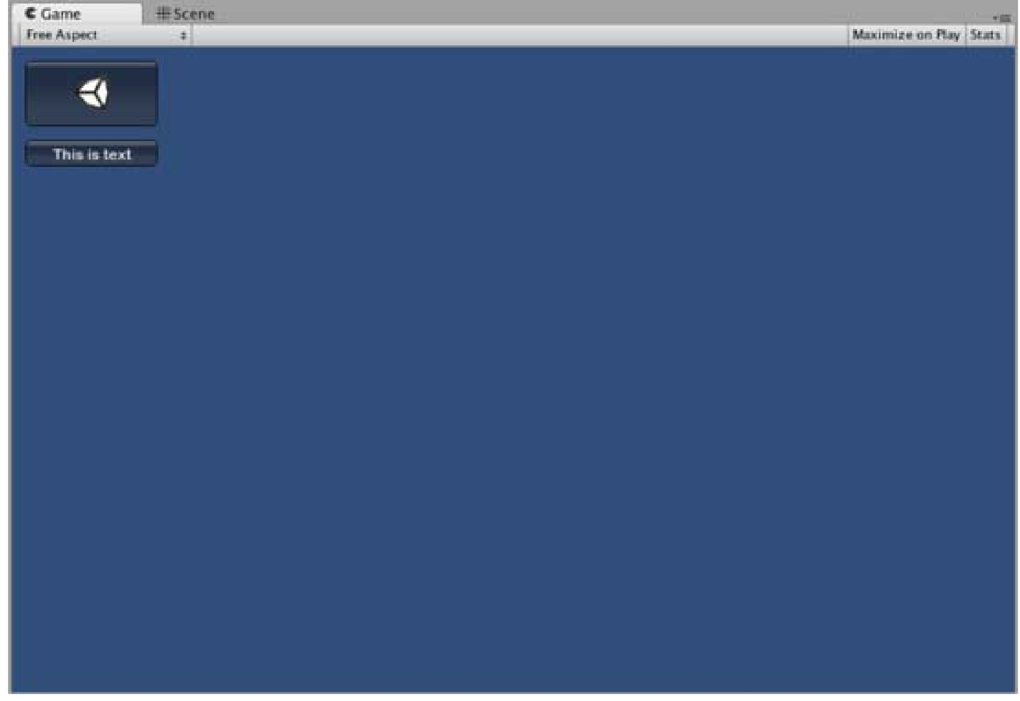
To display an image, declare a **Texture2D** public variable, and pass the variable name as the content argument like this:

```
/* Texture2D Content example */


// JavaScript
var controlTexture : Texture2D;

function OnGUI () {
	GUI.Label (Rect (0,0,100,50), controlTexture);
}


// C#
public Texture2D controlTexture;
  ...

void OnGUI () {
	GUI.Label (new Rect (0,0,100,50), controlTexture);
}
```

Here is an example closer to a real-world scenario:

```
/* Button Content examples */


// JavaScript
var icon : Texture2D;

function OnGUI () {
	if (GUI.Button (Rect (10,10, 100, 50), icon)) {
		print ("you clicked the icon");
	}

	if (GUI.Button (Rect (10,70, 100, 20), "This is text")) {
		print ("you clicked the text button");
	}
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	public Texture2D icon;

	void OnGUI () {
		if (GUI.Button (new Rect (10,10, 100, 50), icon)) {
			print ("you clicked the icon");
		}

		if (GUI.Button (new Rect (10,70, 100, 20), "This is text")) {
			print ("you clicked the text button");
		}
```

```
    }

}
```




*The Buttons created by the above example*

There is a third option which allows you to display images and text together in a GUI Control. You can provide a **GUIContent** object as the Content argument, and define the string and image to be displayed within the GUIContent.

```
/* Using GUIContent to display an image and a string */


// JavaScript
var icon : Texture2D;

function OnGUI () {
    GUI.Box (Rect (10,10,100,50), GUIContent("This is text", icon));
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

    public Texture2D icon;

    void OnGUI () {
        GUI.Box (new Rect (10,10,100,50), new GUIContent("This is text", icon));
    }

}
```

You can also define a **Tooltip** in the GUIContent, and display it elsewhere in the GUI when the mouse hovers over it.

```
/* Using GUIContent to display a tooltip */
```

```
// JavaScript
function OnGUI () {
	// This line feeds "This is the tooltip" into GUI.tooltip
	GUI.Button (Rect (10,10,100,20), GUIContent ("Click me", "This is the tooltip"));
	// This line reads and displays the contents of GUI.tooltip
	GUI.Label (Rect (10,40,100,20), GUI.tooltip);
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	void OnGUI () {
		// This line feeds "This is the tooltip" into GUI.tooltip
		GUI.Button (new Rect (10,10,100,20), new GUIContent ("Click me", "This is the tooltip"));

		// This line reads and displays the contents of GUI.tooltip
		GUI.Label (new Rect (10,40,100,20), GUI.tooltip);
	}

}
```

If you're daring you can also use GUIContent to display a string, an icon, and a tooltip!

```
/* Using GUIContent to display an image, a string, and a tooltip */


// JavaScript
var icon : Texture2D;

function OnGUI () {
	GUI.Button (Rect (10,10,100,20), GUIContent ("Click me", icon, "This is the tooltip"));
	GUI.Label (Rect (10,40,100,20), GUI.tooltip);
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	public Texture2D icon;

	void OnGUI () {
		GUI.Button (new Rect (10,10,100,20), new GUIContent ("Click me", icon, "This is the tooltip"));
		GUI.Label (new Rect (10,40,100,20), GUI.tooltip);
	}

}
```

The scripting reference page for GUIContent's constructor for an extensive list of examples.

Page last updated: 2012-01-17

## gui-Controls

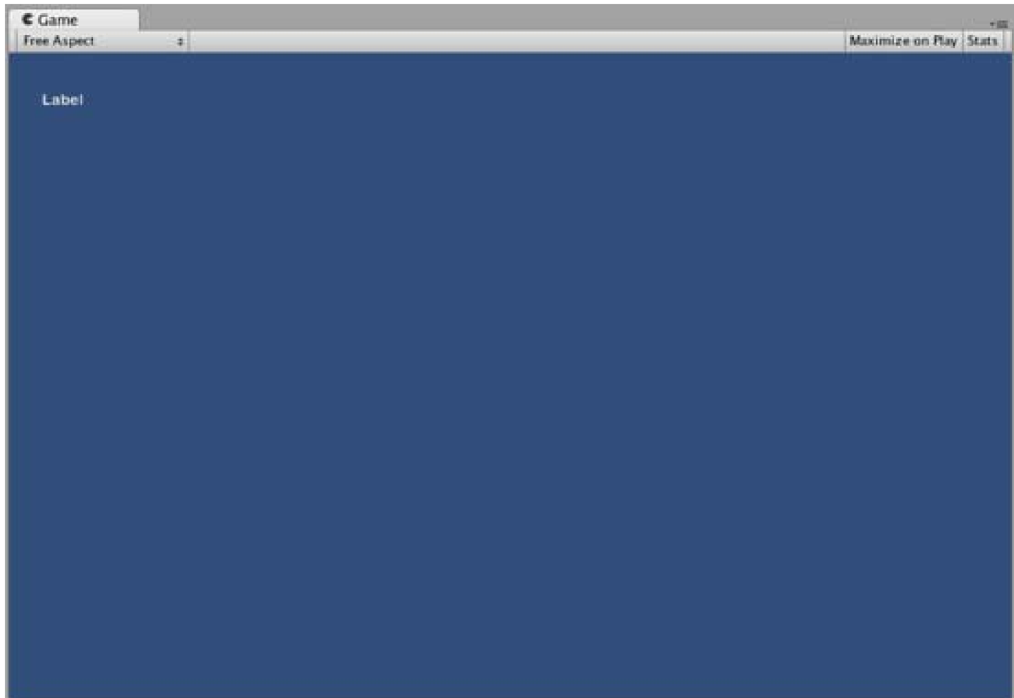
### Control Types

There are a number of different GUI **Controls** that you can create. This section lists all of the available display and interactive Controls. There are other GUI functions that affect layout of Controls, which are described in the Layout section of the Guide.

#### Label

The **Label** is non-interactive. It is for display only. It cannot be clicked or otherwise moved. It is best for displaying information only.

```
/* GUI.Label example */


// JavaScript
function OnGUI () {
	GUI.Label (Rect (25, 25, 100, 30), "Label");
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	void OnGUI () {
		GUI.Label (new Rect (25, 25, 100, 30), "Label");
	}

}
```




*The Label created by the example code*

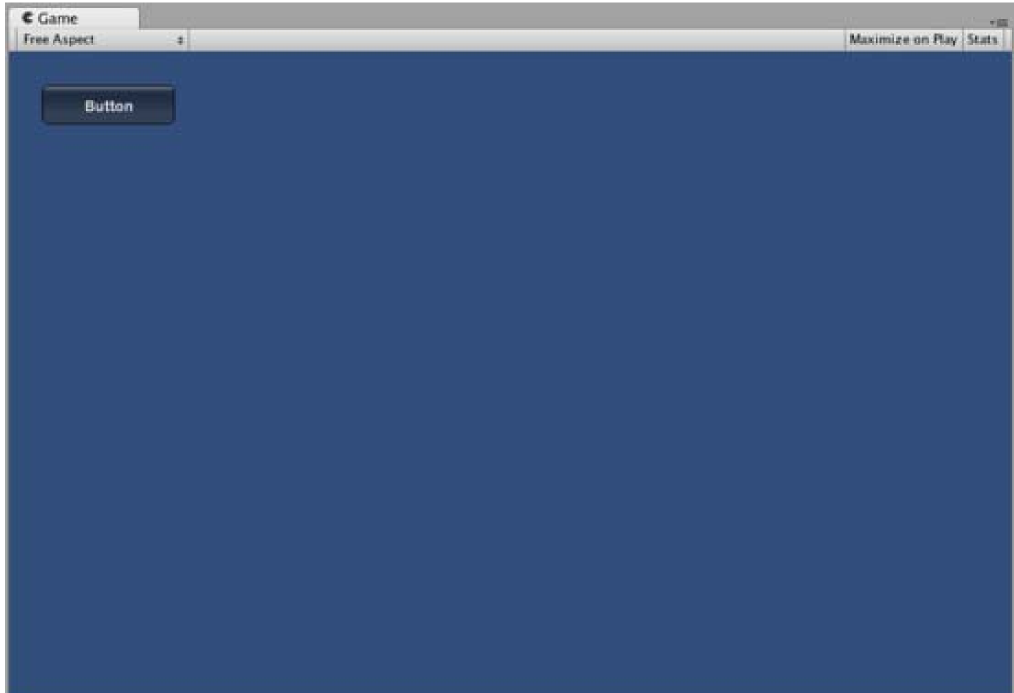
#### Button

The **Button** is a typical interactive button. It will respond a single time when clicked, no matter how long the mouse remains depressed. The response occurs as soon as the mouse button is released.

#### Basic Usage

In UnityGUI, Buttons will return **true** when they are clicked. To execute some code when a Button is clicked, you wrap the the GUI.Button function in an **if** statement. Inside the **if** statement is the code that will be executed when the Button is clicked.

```
/* GUI.Button example */


// JavaScript
function OnGUI () {
	if (GUI.Button (Rect (25, 25, 100, 30), "Button")) {
		// This code is executed when the Button is clicked
	}
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	void OnGUI () {
		if (GUI.Button (new Rect (25, 25, 100, 30), "Button")) {
			// This code is executed when the Button is clicked
		}
	}

}
```




*The Button created by the example code*

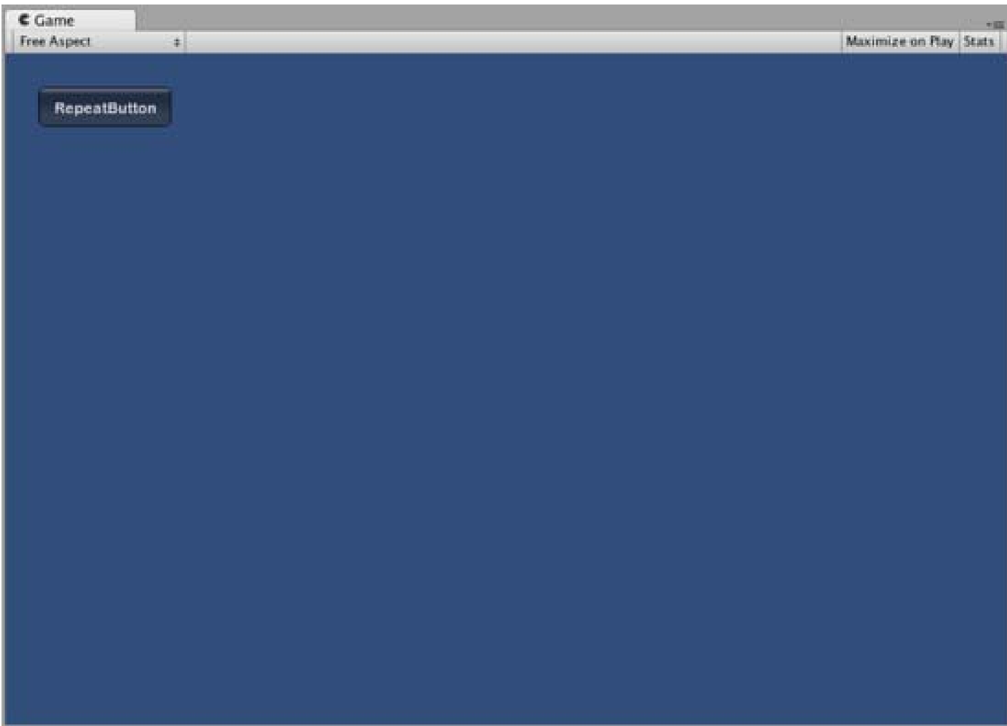
### RepeatButton

**RepeatButton** is a variation of the regular **Button**. The difference is, **RepeatButton** will respond every frame that the mouse button remains depressed. This allows you to create click-and-hold functionality.

### Basic Usage

In UnityGUI, RepeatButtons will return **true** for every frame that they are clicked. To execute some code while the Button is being clicked, you wrap the the GUI.RepeatButton function in an **if** statement. Inside the **if** statement is the code that will be executed while the RepeatButton remains clicked.

```
/* GUI.RepeatButton example */


// JavaScript
function OnGUI () {
	if (GUI.RepeatButton (Rect (25, 25, 100, 30), "RepeatButton")) {
		// This code is executed every frame that the RepeatButton remains clicked
	}
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	void OnGUI () {
		if (GUI.RepeatButton (new Rect (25, 25, 100, 30), "RepeatButton")) {
			// This code is executed every frame that the RepeatButton remains clicked
		}
	}

}
```




*The Repeat Button created by the example code*

## TextField

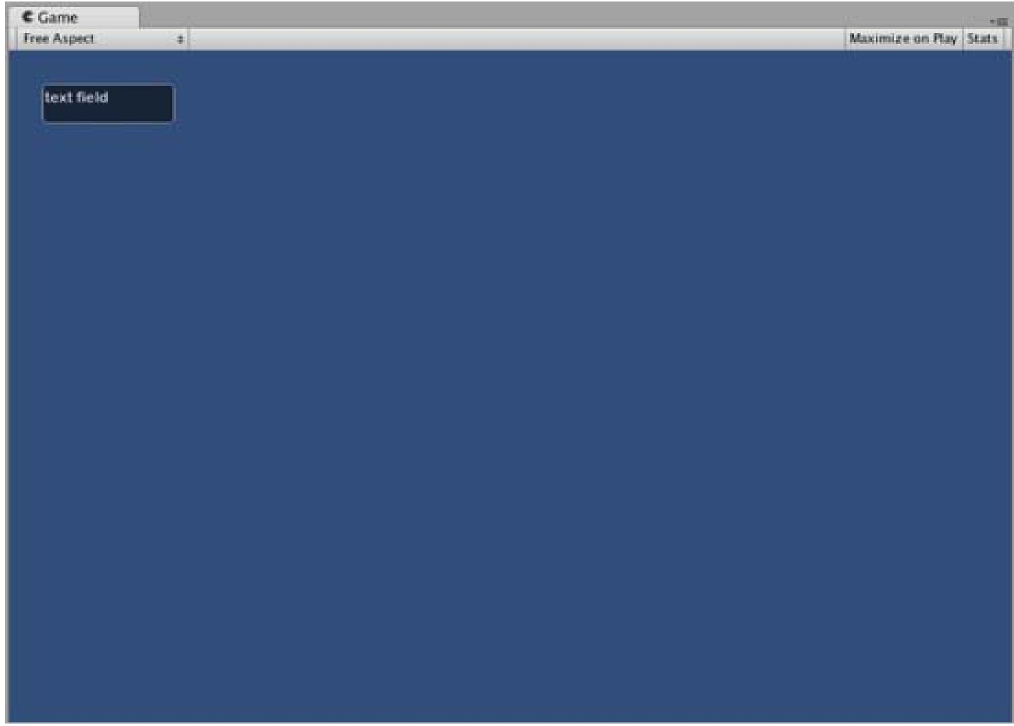
The **TextField** Control is an interactive, editable single-line field containing a text string.

### Basic Usage

The TextField will always display a string. You must provide the string to be displayed in the TextField. When edits are made to

the string, the TextField function will return the edited string.

```
/* GUI.TextField example */


// JavaScript
var textFieldString = "text field";

function OnGUI () {
	textFieldString = GUI.TextField (Rect (25, 25, 100, 30), textFieldString);
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	private string textFieldString = "text field";

	void OnGUI () {
		textFieldString = GUI.TextField (new Rect (25, 25, 100, 30), textFieldString);
	}

}
```



*The TextField created by the example code*

## TextArea

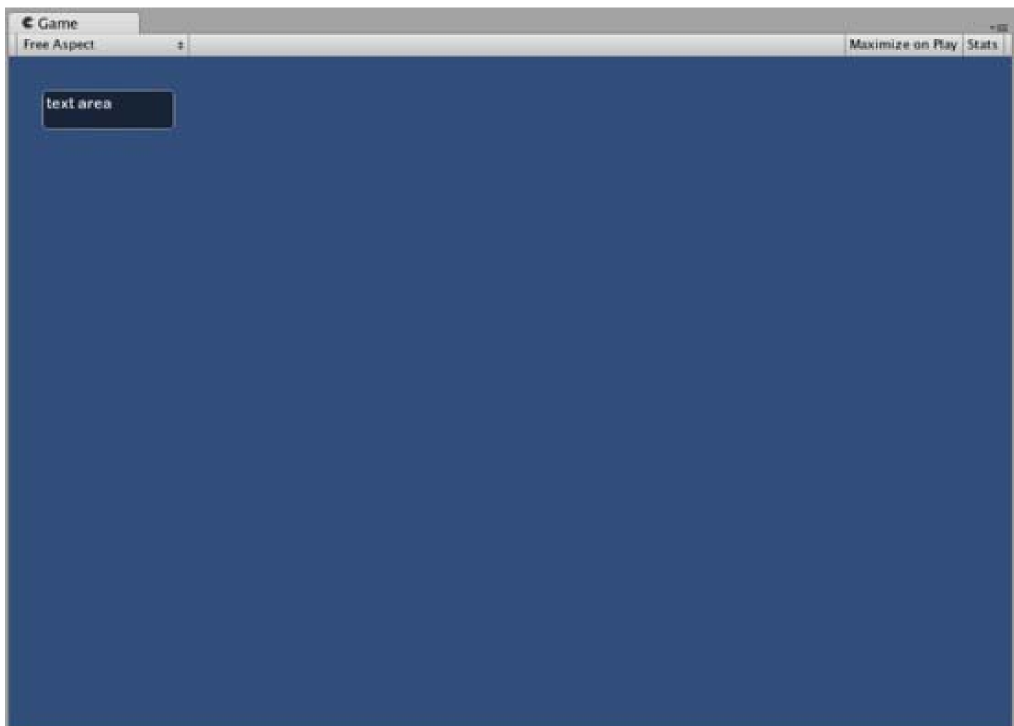
The **TextArea** Control is an interactive, editable multi-line area containing a text string.

### Basic Usage

The TextArea will always display a string. You must provide the string to be displayed in the TextArea. When edits are made to the string, the TextArea function will return the edited string.

```
/* GUI.TextArea example */
```

```
// JavaScript
var textAreaString = "text area";

function OnGUI () {
	textAreaString = GUI.TextArea (Rect (25, 25, 100, 30), textAreaString);
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	private string textAreaString = "text area";

	void OnGUI () {
		textAreaString = GUI.TextArea (new Rect (25, 25, 100, 30), textAreaString);
	}

}
```



*The TextArea created by the example code*

### Toggle

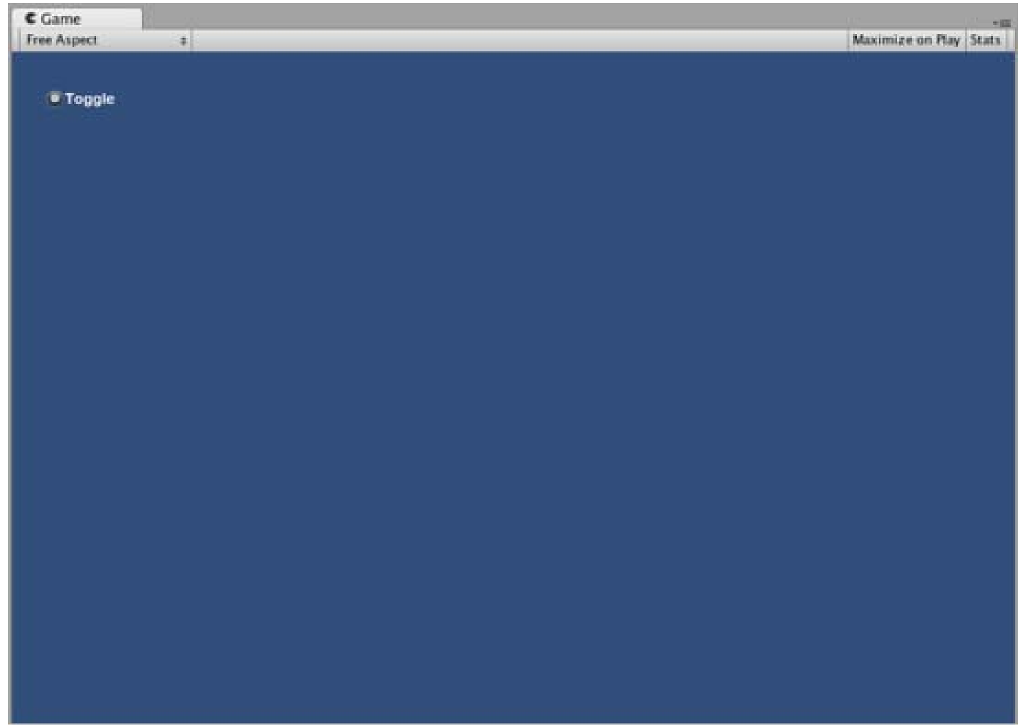
The **Toggle** Control creates a checkbox with a persistent on/off state. The user can change the state by clicking on it.

#### Basic Usage

The Toggle on/off state is represented by a true/false boolean. You must provide the boolean as a parameter to make the Toggle represent the actual state. The Toggle function will return a new boolean value if it is clicked. In order to capture this interactivity, you must assign the boolean to accept the return value of the Toggle function.

```
/* GUI.Toggle example */
```

```
// JavaScript
var toggleBool = true;

function OnGUI () {
	toggleBool = GUI.Toggle (Rect (25, 25, 100, 30), toggleBool, "Toggle");
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	private bool toggleBool = true;

	void OnGUI () {
		toggleBool = GUI.Toggle (new Rect (25, 25, 100, 30), toggleBool, "Toggle");
	}

}
```




*The Toggle created by the example code*

### Toolbar

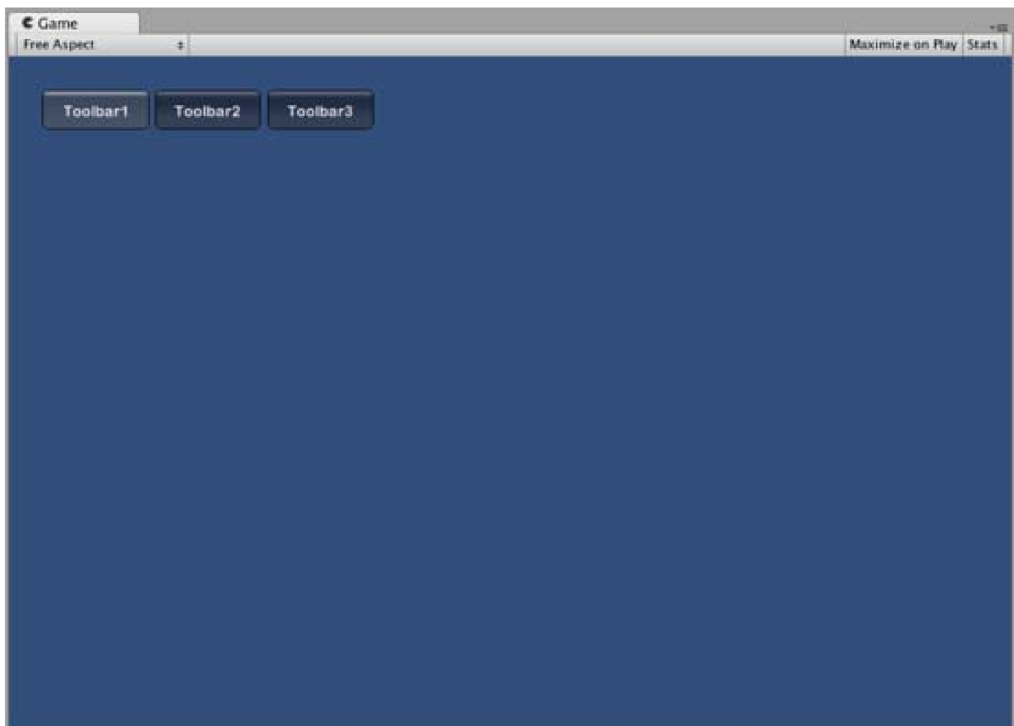
The **Toolbar** Control is essentially a row of **Buttons**. Only one of the Buttons on the Toolbar can be active at a time, and it will remain active until a different Button is clicked. This behavior emulates the behavior of a typical Toolbar. You can define an arbitrary number of Buttons on the Toolbar.

#### Basic Usage

The active Button in the Toolbar is tracked through an integer. You must provide the integer as an argument in the function. To make the Toolbar interactive, you must assign the integer to the return value of the function. The number of elements in the content array that you provide will determine the number of Buttons that are shown in the Toolbar.

```
/* GUI.Toolbar example */
```

```
// JavaScript
var toolbarInt = 0;
var toolbarStrings : String[] = ["Toolbar1", "Toolbar2", "Toolbar3"];

function OnGUI () {
	toolbarInt = GUI.Toolbar (Rect (25, 25, 250, 30), toolbarInt, toolbarStrings);
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	private int toolbarInt = 0;
	private string[] toolbarStrings = {"Toolbar1", "Toolbar2", "Toolbar3"};

	void OnGUI () {
		toolbarInt = GUI.Toolbar (new Rect (25, 25, 250, 30), toolbarInt, toolbarStrings);
	}

}
```



*The Toolbar created by the example code*

### SelectionGrid

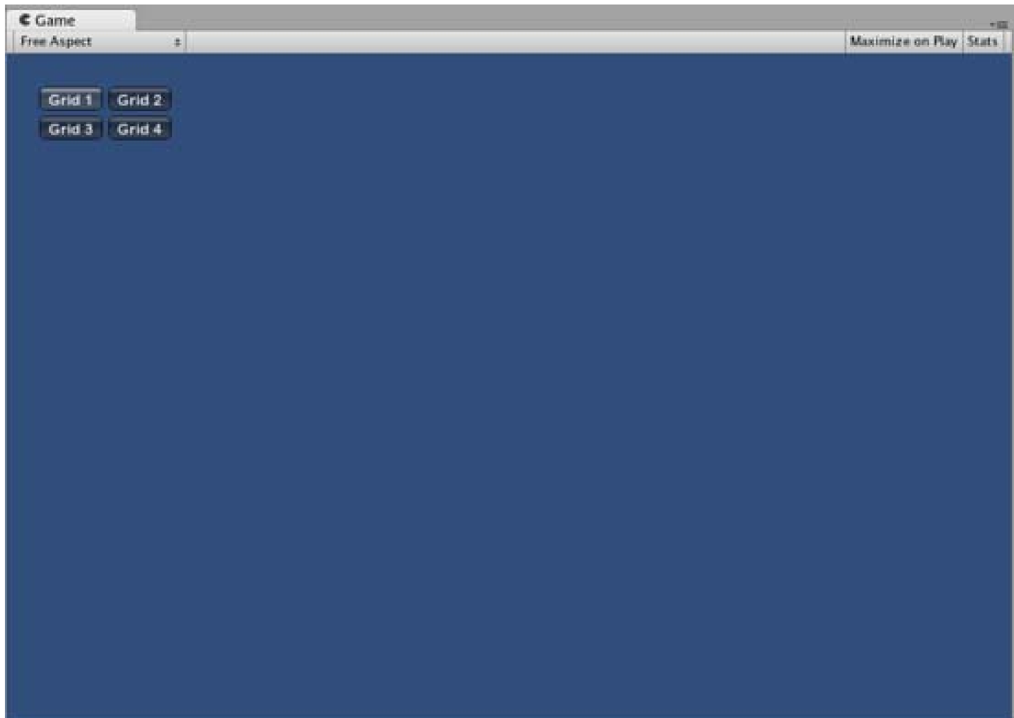
The **SelectionGrid** Control is a multi-row **Toolbar**. You can determine the number of columns and rows in the grid. Only one Button can be active at time.

#### Basic Usage

The active Button in the SelectionGrid is tracked through an integer. You must provide the integer as an argument in the function. To make the SelectionGrid interactive, you must assign the integer to the return value of the function. The number of elements in the content array that you provide will determine the number of Buttons that are shown in the SelectionGrid. You also can dictate the number of columns through the function arguments.

```
/* GUI.SelectionGrid example */
```

```
// JavaScript
var selectionGridInt : int = 0;
var selectionStrings : String[] = ["Grid 1", "Grid 2", "Grid 3", "Grid 4"];

function OnGUI () {
	selectionGridInt = GUI.SelectionGrid (Rect (25, 25, 100, 30), selectionGridInt, selectionStrings, 2);

}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	private int selectionGridInt = 0;
	private string[] selectionStrings = {"Grid 1", "Grid 2", "Grid 3", "Grid 4"};

	void OnGUI () {
		selectionGridInt = GUI.SelectionGrid (new Rect (25, 25, 300, 60), selectionGridInt, selectionStrings, 2);

	}

}
```



*The SelectionGrid created by the example code*

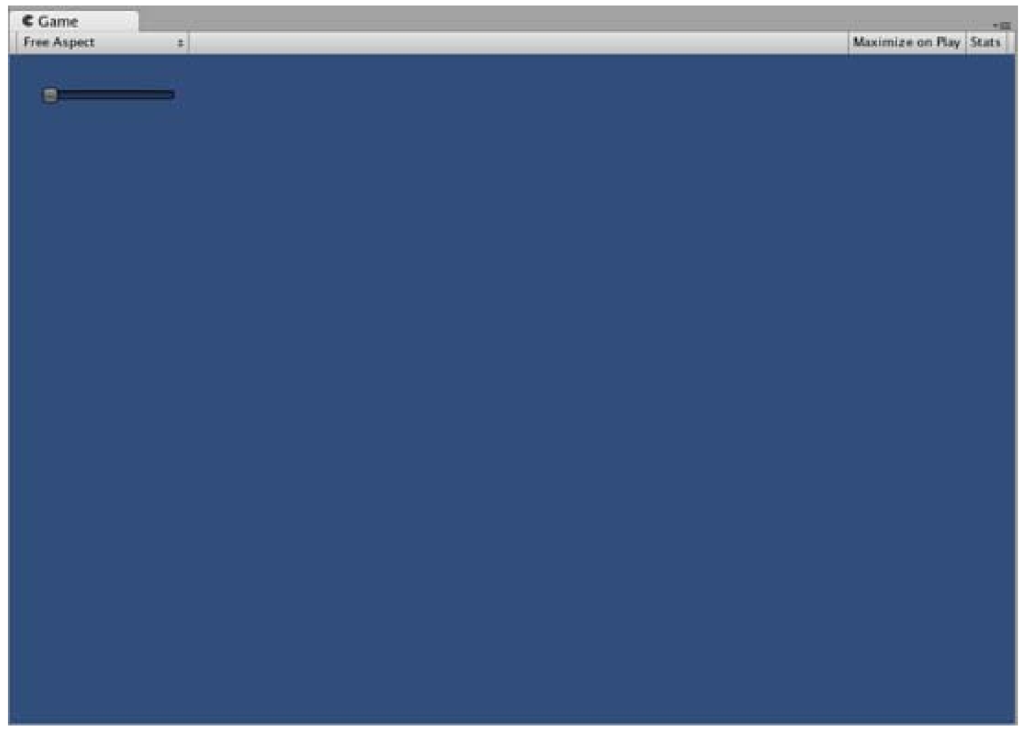
### HorizontalSlider

The **HorizontalSlider** Control is a typical horizontal sliding knob that can be dragged to change a value between predetermined min and max values.

#### Basic Usage

The position of the Slider knob is stored as a float. To display the position of the knob, you provide that float as one of the arguments in the function. There are two additional values that determine the minimum and maximum values. If you want the slider knob to be adjustable, assign the slider value float to be the return value of the Slider function.

```
/* Horizontal Slider example */


// JavaScript
var hSliderValue : float = 0.0;

function OnGUI () {
	hSliderValue = GUI.HorizontalSlider (Rect (25, 25, 100, 30), hSliderValue, 0.0, 10.0);
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	private float hSliderValue = 0.0f;

	void OnGUI () {
		hSliderValue = GUI.HorizontalSlider (new Rect (25, 25, 100, 30), hSliderValue, 0.0f, 10.0f);
	}

}
```




*The Horizontal Slider created by the example code*

### VerticalSlider

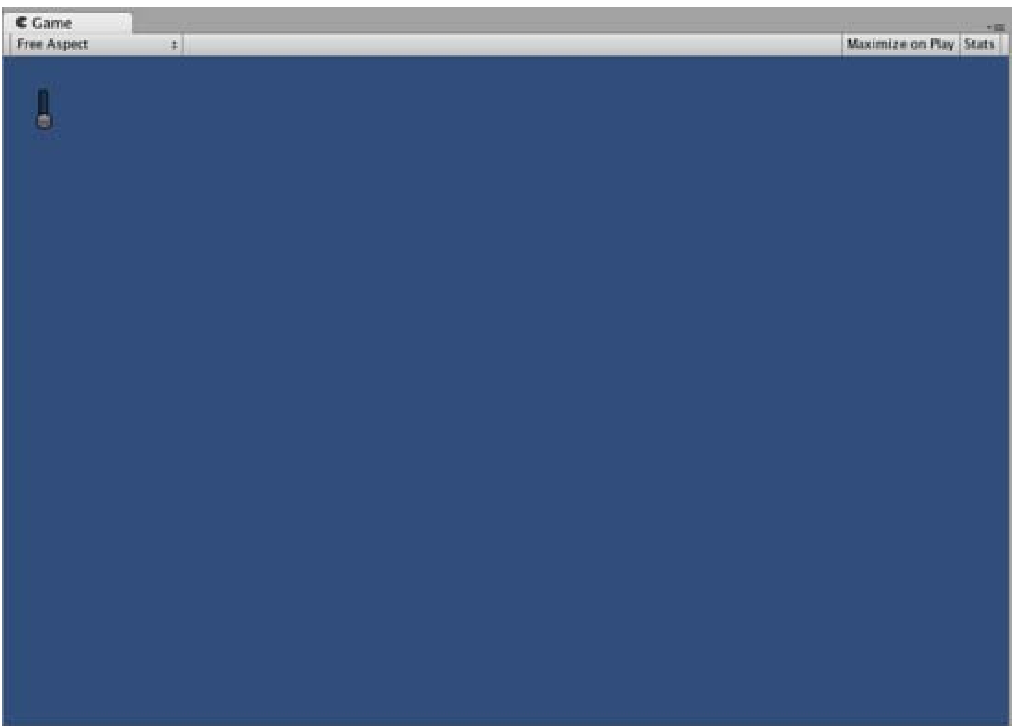
The **VerticalSlider** Control is a typical vertical sliding knob that can be dragged to change a value between predetermined min and max values.

#### Basic Usage

The position of the Slider knob is stored as a float. To display the position of the knob, you provide that float as one of the arguments in the function. There are two additional values that determine the minimum and maximum values. If you want the slider knob to be adjustable, assign the slider value float to be the return value of the Slider function.

```
/* Vertical Slider example */
```

```
// JavaScript
var vSliderValue : float = 0.0;

function OnGUI () {
	vSliderValue = GUI.VerticalSlider (Rect (25, 25, 100, 30), vSliderValue, 10.0, 0.0);
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	private float vSliderValue = 0.0f;

	void OnGUI () {
		vSliderValue = GUI.VerticalSlider (new Rect (25, 25, 100, 30), vSliderValue, 10.0f, 0.0f);
	}

}
```




*The Vertical Slider created by the example code*

### HorizontalScrollbar

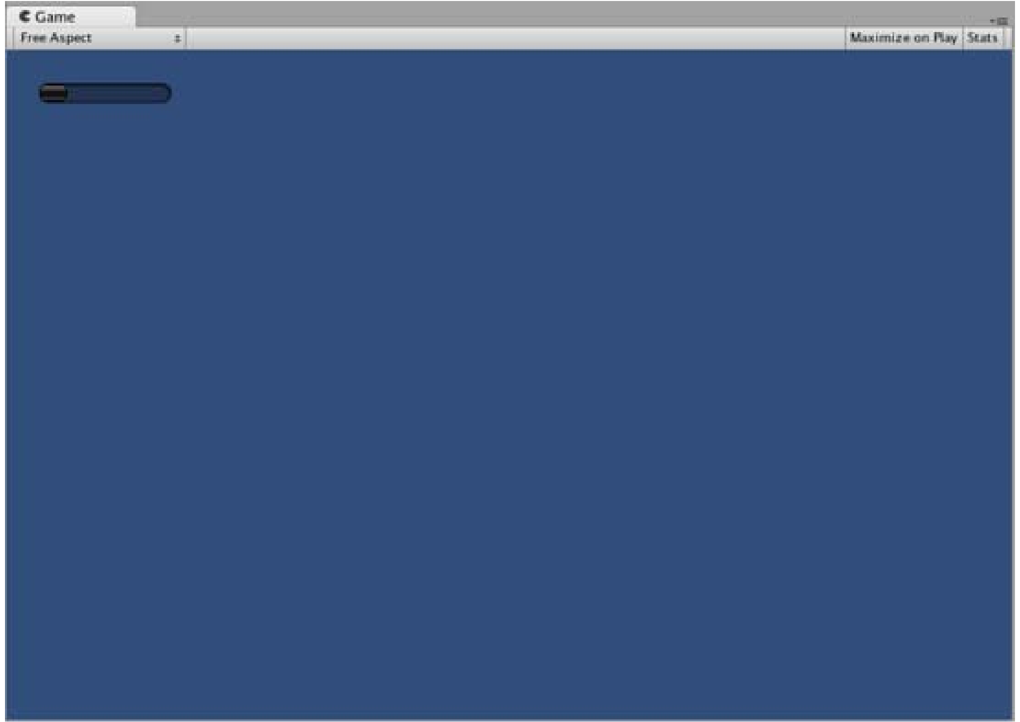
The **HorizontalScrollbar** Control is similar to a **Slider** Control, but visually similar to Scrolling elements for web browsers or word processors. This control is used to navigate the **ScrollView** Control.

#### Basic Usage

Horizontal Scrollbars are implemented identically to Horizontal Sliders with one exception: There is an additional argument which controls the width of the Scrollbar knob itself.

```
/* Horizontal Scrollbar example */
```

```
// JavaScript
var hScrollbarValue : float;

function OnGUI () {
	hScrollbarValue = GUI.HorizontalScrollbar (Rect (25, 25, 100, 30), hScrollbarValue, 1.0, 0.0, 10.0);
}



// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	private float hScrollbarValue;

	void OnGUI () {
		hScrollbarValue = GUI.HorizontalScrollbar (new Rect (25, 25, 100, 30), hScrollbarValue, 1.0f, 0.0f, 10.0f);
	}

}
```




*The Horizontal Scrollbar created by the example code*

### VerticalScrollbar

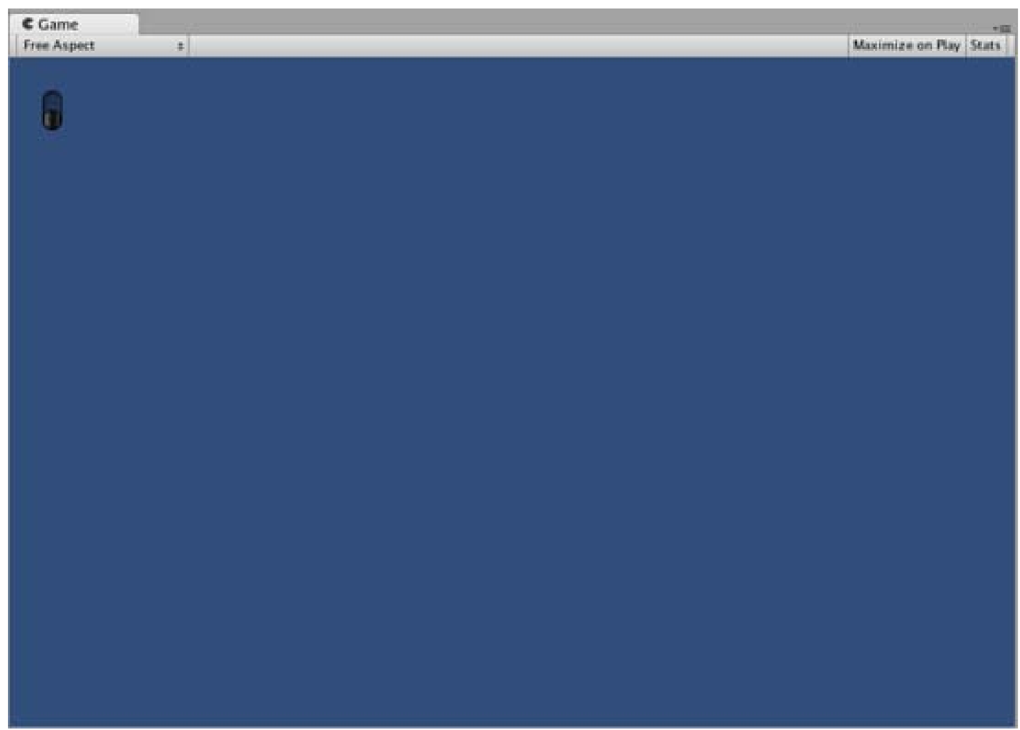
The **VerticalScrollbar** Control is similar to a **Slider** Control, but visually similar to Scrolling elements for web browsers or word processors. This control is used to navigate the **ScrollView** Control.

#### Basic Usage

Vertical Scrollbars are implemented identically to Vertical Sliders with one exception: There is an additional argument which controls the height of the Scrollbar knob itself.

```
/* Vertical Scrollbar example */



// JavaScript
var vScrollbarValue : float;
```

```
function OnGUI () {
	vScrollbarValue = GUI. VerticalScrollbar (Rect (25, 25, 100, 30), vScrollbarValue, 1.0, 10.0, 0.0);
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	private float vScrollbarValue;

	void OnGUI () {
		vScrollbarValue = GUI. VerticalScrollbar (new Rect (25, 25, 100, 30), vScrollbarValue, 1.0f, 10.0f, 0.0f);
	}

}
```




*The Vertical Scrollbar created by the example code*

### ScrollView

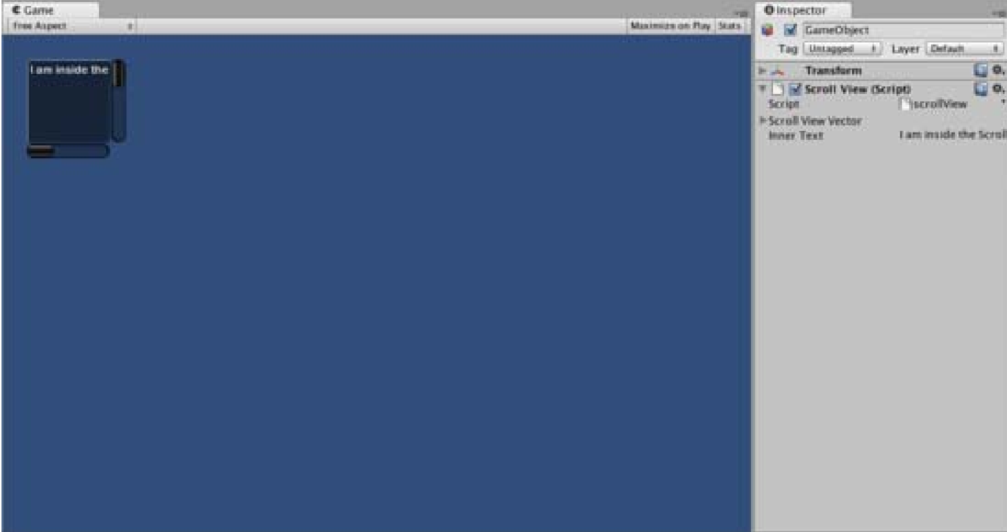
**ScrollViews** are Controls that display a viewable area of a much larger set of Controls.

#### Basic Usage

ScrollViews require two **Rects** as arguments. The first **Rect** defines the location and size of the viewable ScrollView area on the screen. The second **Rect** defines the size of the space contained inside the viewable area. If the space inside the viewable area is larger than the viewable area, Scrollbars will appear as appropriate. You must also assign and provide a 2D Vector which stores the position of the viewable area that is displayed.

```
/* ScrollView example */


// JavaScript
var scrollViewVector : Vector2 = Vector2.zero;
var innerText : String = "I am inside the ScrollView";
```

```
function OnGUI () {
	// Begin the ScrollView
	scrollViewVector = GUI.BeginScrollView (Rect (25, 25, 100, 100), scrollViewVector, Rect (0, 0, 400, 400));

	// Put something inside the ScrollView
	innerText = GUI.TextArea (Rect (0, 0, 400, 400), innerText);

	// End the ScrollView
	GUI.EndScrollView();
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	private Vector2 scrollViewVector = Vector2.zero;
	private string innerText = "I am inside the ScrollView";

	void OnGUI () {
		// Begin the ScrollView
		scrollViewVector = GUI.BeginScrollView (new Rect (25, 25, 100, 100), scrollViewVector, new Rect (0, 0, 400, 400))

		// Put something inside the ScrollView
		innerText = GUI.TextArea (new Rect (0, 0, 400, 400), innerText);

		// End the ScrollView
		GUI.EndScrollView();
	}

}
```




*The ScrollView created by the example code*

### Window

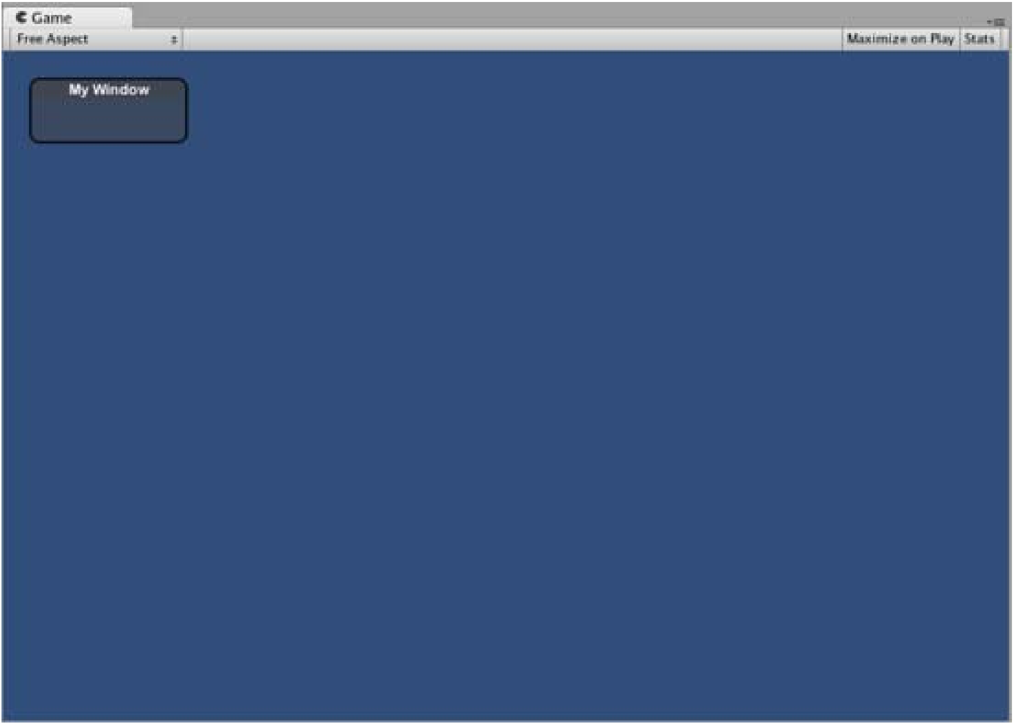
**Windows** are drag-able containers of Controls. They can receive and lose focus when clicked. Because of this, they are implemented slightly differently from the other Controls. Each Window has an **id** number, and its contents are declared inside a separate function that is called when the Window has focus.

#### Basic Usage

Windows are the only Control that require an additional function to work properly. You must provide an **id** number and a

function name to be executed for the Window. Inside the Window function, you create your actual behaviors or contained Controls.

```
/* Window example */


// JavaScript
var windowRect : Rect = Rect (20, 20, 120, 50);

function OnGUI () {
	windowRect = GUI.Window (0, windowRect, WindowFunction, "My Window");
}

function WindowFunction (windowID : int) {
	// Draw any Controls inside the window here
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	private Rect windowRect = new Rect (20, 20, 120, 50);

	void OnGUI () {
		windowRect = GUI.Window (0, windowRect, WindowFunction, "My Window");
	}

	void WindowFunction (int windowID) {
		// Draw any Controls inside the window here
	}

}
```




*The Window created by the example code*

### GUI.changed

To detect if the user did any action in the GUI (clicked a button, dragged a slider, etc), read the **GUI.changed** value from your script. This gets set to true when the user has done something, making it easy to validate the user input.

A common scenario would be for a Toolbar, where you want to change a specific value based on which Button in the Toolbar was clicked. You don't want to assign the value in every call to **OnGUI()**, only when one of the Buttons has been clicked.

```
/* GUI.changed example */


// JavaScript
private var selectedToolbar : int = 0;
private var toolbarStrings = ["One", "Two"];

function OnGUI () {
	// Determine which button is active, whether it was clicked this frame or not
	selectedToolbar = GUI.Toolbar (Rect (50, 10, Screen.width - 100, 30), selectedToolbar, toolbarStrings);

	// If the user clicked a new Toolbar button this frame, we'll process their input
	if (GUI.changed)
	{
		print ("The toolbar was clicked");

		if (selectedToolbar == 0)
		{
			print ("First button was clicked");
		}
		else
		{
			print ("Second button was clicked");
		}
	}
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	private int selectedToolbar = 0;
	private string[] toolbarStrings = {"One", "Two"};

	void OnGUI () {
		// Determine which button is active, whether it was clicked this frame or not
		selectedToolbar = GUI.Toolbar (new Rect (50, 10, Screen.width - 100, 30), selectedToolbar, toolbarStrings);

		// If the user clicked a new Toolbar button this frame, we'll process their input
		if (GUI.changed)
		{
			Debug.Log("The toolbar was clicked");

			if (0 == selectedToolbar)
			{
				Debug.Log("First button was clicked");
			}
			else
			{
				Debug.Log("Second button was clicked");
```

```
                }
            }
        }

}
```

**GUI.changed** will return true if any GUI Control placed before it was manipulated by the user.

Page last updated: 2012-01-17

---

# gui-Customization

## Customizing your GUI Controls

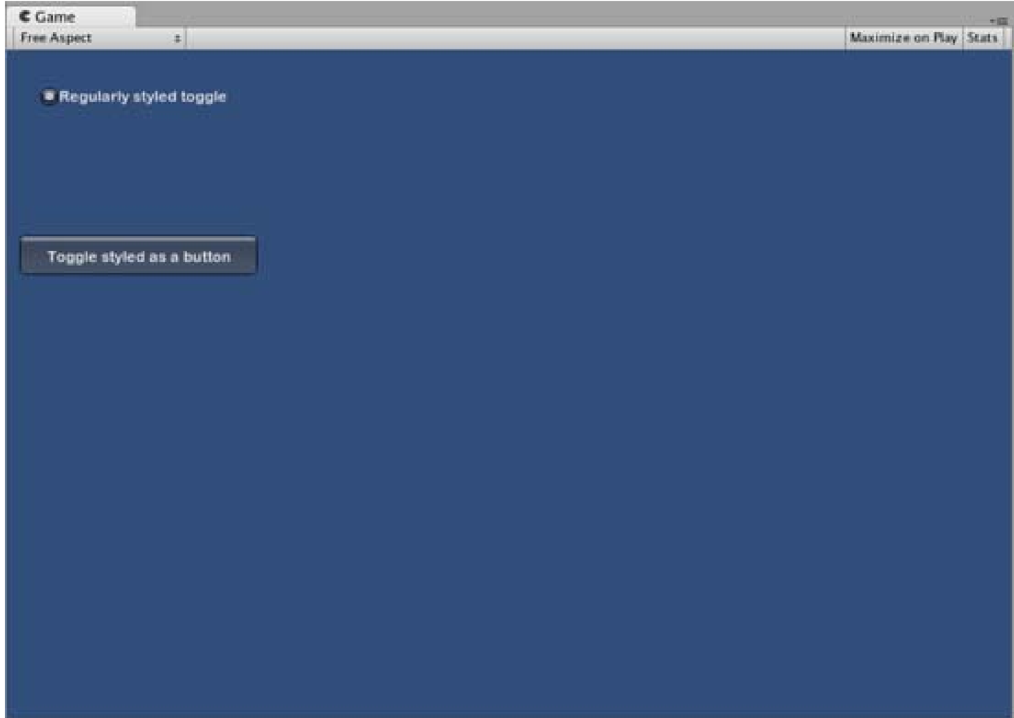
Functional Controls are necessary for your game, and the appearance of those controls is very important for the aesthetics of your game. In UnityGUI, you can fine-tune the appearance of your Controls with many details. Control appearances are dictated with **GUIStyles**. By default, when you create a Control without defining a GUIStyle, Unity's default GUIStyle is applied. This style is internal in Unity and can be used in published games for quick prototyping, or if you choose not to stylize your Controls.

When you have a large number of different GUIStyles to work with, you can define them all within a single **GUISkin**. A GUISkin is no more than a collection of GUIStyles.

### How Styles change the look of your GUI Controls

GUIStyles are designed to mimic Cascading Style Sheets (CSS) for web browsers. Many different CSS methodologies have been adapted, including differentiation of individual state properties for styling, and separation between the content and the appearance.

Where the Control defines the content, the Style defines the appearance. This allows you to create combinations like a functional **Toggle** which looks like a normal **Button**.



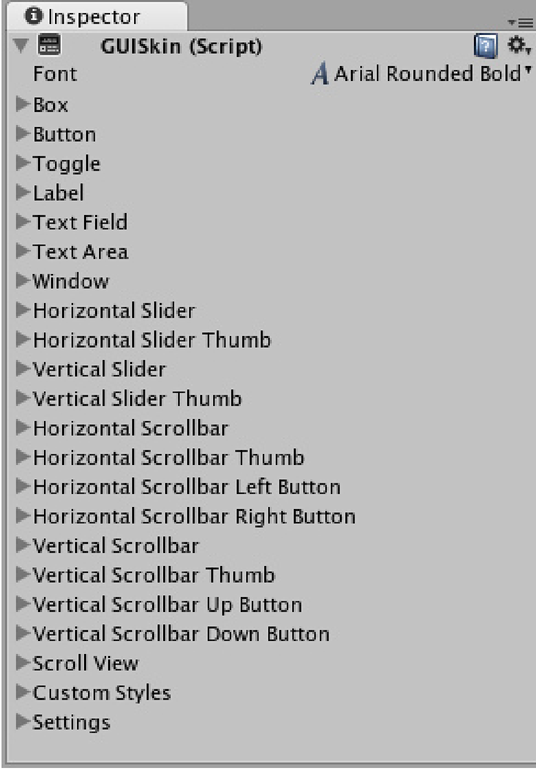

*Two Toggle Controls styled differently*

### The difference between Skins and Styles

As stated earlier, GUISkins are a collection of GUIStyles. Styles define the appearance of a GUI Control. You do not have to

use a Skin if you want to use a Style.

*A single GUIStyle shown in the Inspector*

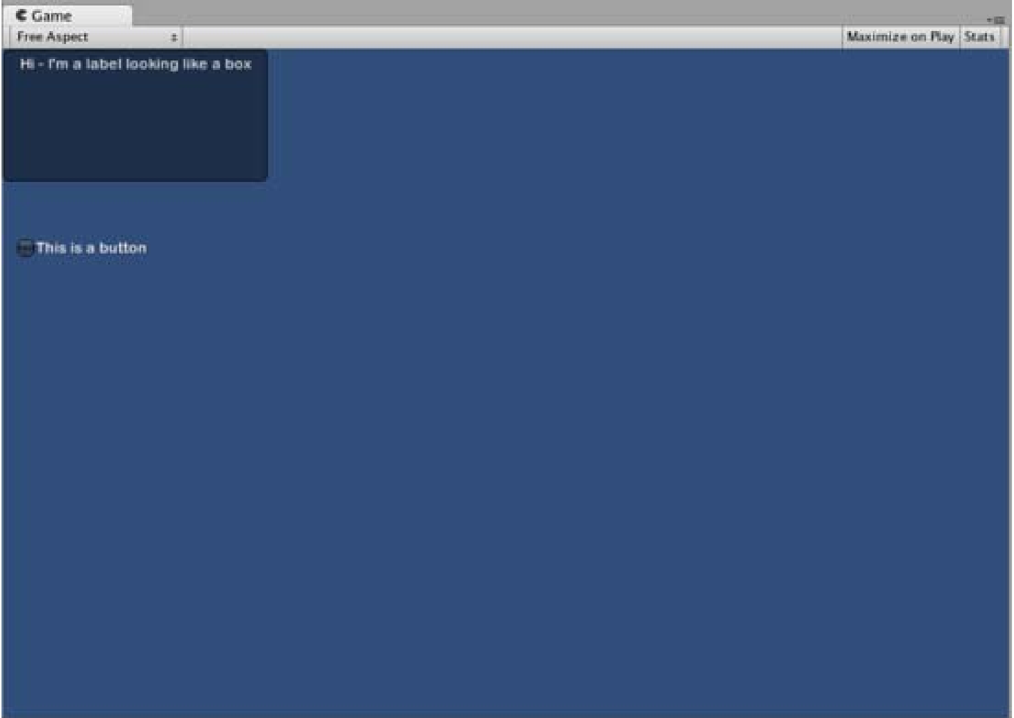



*A single GUISkin shown in the Inspector - observe that it contains multiple GUIStyles*

### Working with Styles

All GUI Control functions have an optional last parameter: the GUIStyle to use for displaying the Control. If this is omitted, Unity's default GUIStyle will be used. This works internally by applying the name of the control type as a string, so **GUI.Button()** uses the "button" style, **GUI.Toggle()** uses the "toggle" style, etc. You can override the default GUIStyle for a control by specifying it as the last parameter.

```
/* Override the default Control Style with a different style in the UnityGUI default Styles */


// JavaScript
function OnGUI () {
	// Make a label that uses the "box" GUIStyle.
	GUI.Label (Rect (0,0,200,100), "Hi - I'm a label looking like a box", "box");

	// Make a button that uses the "toggle" GUIStyle
	GUI.Button (Rect (10,140,180,20), "This is a button", "toggle");
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	void OnGUI () {
		// Make a label that uses the "box" GUIStyle.
		GUI.Label (new Rect (0,0,200,100), "Hi - I'm a label looking like a box", "box");

		// Make a button that uses the "toggle" GUIStyle
		GUI.Button (new Rect (10,140,180,20), "This is a button", "toggle");
	}

}
```




*The controls created by the code example above*

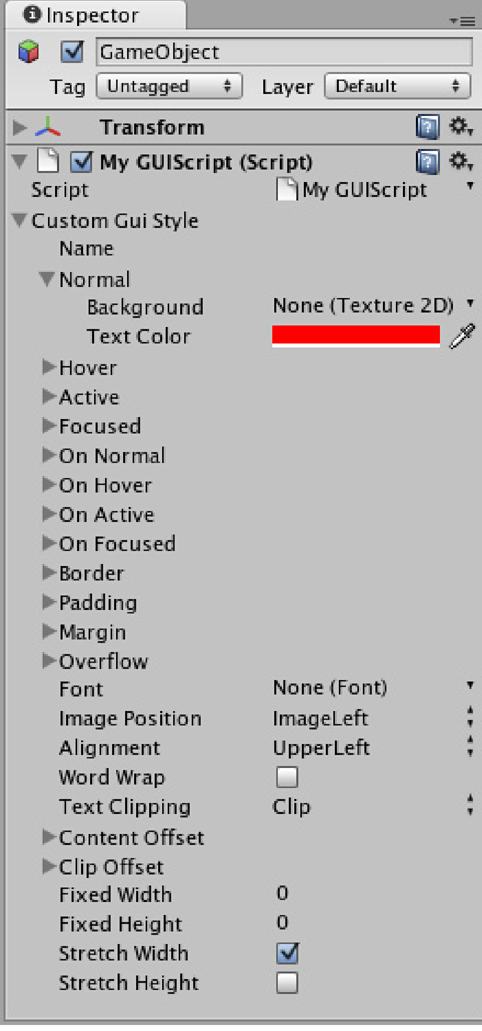
#### Making a public variable GUIStyle

When you declare a public GUIStyle variable, all elements of the Style will show up in the **Inspector**. You can edit all of the different values there.

```
/* Overriding the default Control Style with one you've defined yourself */


// JavaScript
var customButton : GUIStyle;

function OnGUI () {
	// Make a button. We pass in the GUIStyle defined above as the style to use
	GUI.Button (Rect (10,10,150,20), "I am a Custom Button", customButton);
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	public GUIStyle customButton;

	void OnGUI () {
		// Make a button. We pass in the GUIStyle defined above as the style to use
		GUI.Button (new Rect (10,10,150,20), "I am a Custom Button", customButton);
	}

}
```

#### Changing the different style elements

When you have declared a GUIStyle, you can modify that style in the Inspector. There are a great number of States you can define, and apply to any type of Control.




*Styles are modified on a per-script, per-GameObject basis*

Any Control State must be assigned a **Background** Color before the specified **Text Color** will be applied.

For more information about individual GUIStyles, please read the GUIStyle Component Reference page.

## Working with Skins

For more complicated GUI systems, it makes sense to keep a collection of styles in one place. This is what a GUISkin does. A GUISkin contains multiple different Styles, essentially providing a complete face-lift to all GUI Controls.

### Creating a new GUISkin

To create a GUISkin, select **Assets->Create->GUI Skin** from the menu bar. This will create a GUI Skin in your Project Folder. Select it to see all GUIStyles defined by the Skin in the Inspector.

### Applying the skin to a GUI

To use a skin you've created, assign it to **GUI.skin** in your **OnGUI()** function.

```
/* Make a property containing a reference to the skin you want to use */


// JavaScript
var mySkin : GUISkin;

function OnGUI () {
	// Assign the skin to be the one currently used.
	GUI.skin = mySkin;
```

```
        // Make a button. This will get the default "button" style from the skin assigned to mySkin.
        GUI.Button (Rect (10,10,150,20), "Skinned Button");
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

        public GUISkin mySkin;

        void OnGUI () {
                // Assign the skin to be the one currently used.
                GUI.skin = mySkin;

                // Make a button. This will get the default "button" style from the skin assigned to mySkin.
                GUI.Button (new Rect (10,10,150,20), "Skinned Button");
        }

}
```

You can switch skins as much as you like throughout a single **OnGUI()** call.

```
/* Example of switching skins in the same OnGUI() call */


// JavaScript
var mySkin : GUISkin;

var toggle = true;

function OnGUI () {
        // Assign the skin to be the one currently used.
        GUI.skin = mySkin;

        // Make a toggle. This will get the "button" style from the skin assigned to mySkin.
        toggle = GUI.Toggle (Rect (10,10,150,20), toggle, "Skinned Button", "button");

        // Assign the currently skin to be Unity's default.
        GUI.skin = null;

        // Make a button. This will get the default "button" style from the built-in skin.
        GUI.Button (Rect (10,35,150,20), "Built-in Button");
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

        public GUISkin mySkin;
        private bool toggle = true;

        void OnGUI () {
                // Assign the skin to be the one currently used.
                GUI.skin = mySkin;
```

```
        // Make a toggle. This will get the "button" style from the skin assigned to mySkin.
        toggle = GUI.Toggle (new Rect (10,10,150,20), toggle, "Skinned Button", "button");

        // Assign the currently skin to be Unity's default.
        GUI.skin = null;

        // Make a button. This will get the default "button" style from the built-in skin.
        GUI.Button (new Rect (10,35,150,20), "Built-in Button");
    }

}
```

Page last updated: 2012-01-17

---

# gui-Layout

### Fixed Layout vs Automatic Layout

There are two different modes you can use to arrange and organize your GUIs: Fixed and Automatic. Up until now, every UnityGUI example provided in this guide has used Fixed Layout. To use Automatic Layout, write **GUILayout** instead of **GUI** when calling control functions. You do not have to use one Layout mode over the other, and you can use both modes at once in the same **OnGUI()** function.

Fixed Layout makes sense to use when you have a pre-designed interface to work from. Automatic Layout makes sense to use when you don't know how many elements you need up front, or don't want to worry about hand-positioning each Control. For example, if you are creating a number of different buttons based on Save Game files, you don't know exactly how many buttons will be drawn. In this case Automatic Layout might make more sense. It is really dependent on the design of your game and how you want to present your interface.

There are two key differences when using Automatic Layout:

- **GUILayout** is used instead of **GUI**
- No **Rect()** function is required for Automatic Layout Controls

```
/* Two key differences when using Automatic Layout */


// JavaScript
function OnGUI () {
    // Fixed Layout
    GUI.Button (Rect (25,25,100,30), "I am a Fixed Layout Button");

    // Automatic Layout
    GUILayout.Button ("I am an Automatic Layout Button");
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

    void OnGUI () {
        // Fixed Layout
        GUI.Button (new Rect (25,25,100,30), "I am a Fixed Layout Button");
```

```
		// Automatic Layout
		GUILayout.Button ("I am an Automatic Layout Button");
	}

}
```

### Arranging Controls

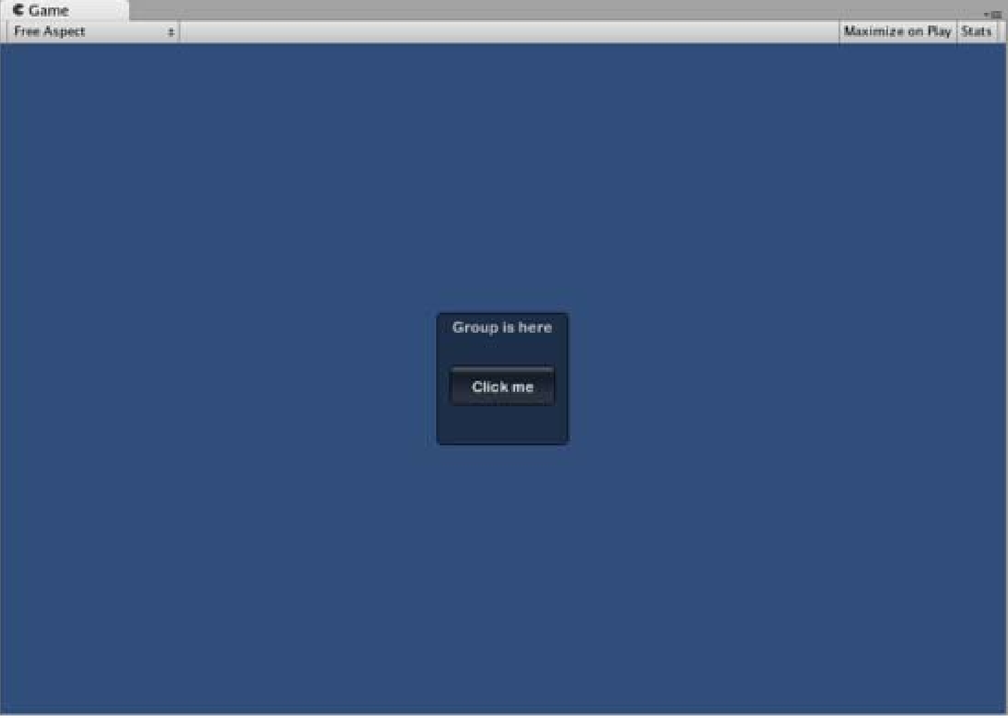
Depending on which Layout Mode you're using, there are different hooks for controlling where your Controls are positioned and how they are grouped together. In Fixed Layout, you can put different Controls into **Groups**. In Automatic Layout, you can put different Controls into **Areas**, **Horizontal Groups**, and **Vertical Groups**

#### Fixed Layout - Groups

Groups are a convention available in Fixed Layout Mode. They allow you to define areas of the screen that contain multiple Controls. You define which Controls are inside a Group by using the **GUI.BeginGroup()** and **GUI.EndGroup()** functions. All Controls inside a Group will be positioned based on the Group's top-left corner instead of the screen's top-left corner. This way, if you reposition the group at runtime, the relative positions of all Controls in the group will be maintained.

As an example, it's very easy to center multiple Controls on-screen.

```
/* Center multiple Controls on the screen using Groups */


// JavaScript
function OnGUI () {
	// Make a group on the center of the screen
	GUI.BeginGroup (Rect (Screen.width / 2 - 50, Screen.height / 2 - 50, 100, 100));
	// All rectangles are now adjusted to the group. (0,0) is the topleft corner of the group.

	// We'll make a box so you can see where the group is on-screen.
	GUI.Box (Rect (0,0,100,100), "Group is here");
	GUI.Button (Rect (10,40,80,30), "Click me");

	// End the group we started above. This is very important to remember!
	GUI.EndGroup ();
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	void OnGUI () {
		// Make a group on the center of the screen
		GUI.BeginGroup (new Rect (Screen.width / 2 - 50, Screen.height / 2 - 50, 100, 100));
		// All rectangles are now adjusted to the group. (0,0) is the topleft corner of the group.

		// We'll make a box so you can see where the group is on-screen.
		GUI.Box (new Rect (0,0,100,100), "Group is here");
		GUI.Button (new Rect (10,40,80,30), "Click me");

		// End the group we started above. This is very important to remember!
		GUI.EndGroup ();
	}

}
```

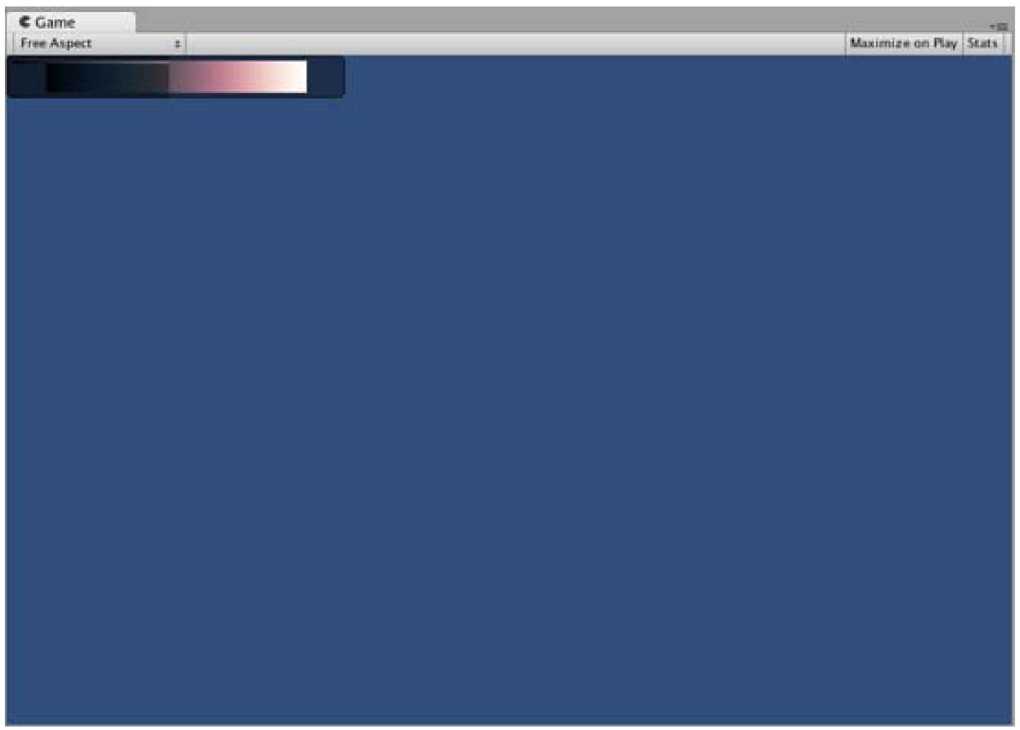



*The above example centers controls regardless of the screen resolution*

You can also nest multiple Groups inside each other. When you do this, each group has its contents clipped to its parent's space.

```
/* Using multiple Groups to clip the displayed Contents */



// JavaScript
var bgImage : Texture2D; // background image that is 256 x 32
var fgImage : Texture2D; // foreground image that is 256 x 32
var playerEnergy = 1.0; // a float between 0.0 and 1.0

function OnGUI () {
	// Create one Group to contain both images
	// Adjust the first 2 coordinates to place it somewhere else on-screen
	GUI.BeginGroup (Rect (0,0,256,32));

	// Draw the background image
	GUI.Box (Rect (0,0,256,32), bgImage);

	// Create a second Group which will be clipped
	// We want to clip the image and not scale it, which is why we need the second Group
	GUI.BeginGroup (Rect (0,0,playerEnergy * 256, 32));

	// Draw the foreground image
	GUI.Box (Rect (0,0,256,32), fgImage);

	// End both Groups
	GUI.EndGroup ();
	GUI.EndGroup ();
}



// C#
using UnityEngine;
using System.Collections;
```

```
public class GUITest : MonoBehaviour {

	// background image that is 256 x 32
	public Texture2D bgImage;

	// foreground image that is 256 x 32
	public Texture2D fgImage;

	// a float between 0.0 and 1.0
	public float playerEnergy = 1.0f;

	void OnGUI () {
		// Create one Group to contain both images
		// Adjust the first 2 coordinates to place it somewhere else on-screen
		GUI.BeginGroup (new Rect (0,0,256,32));

		// Draw the background image
		GUI.Box (new Rect (0,0,256,32), bgImage);

			// Create a second Group which will be clipped
			// We want to clip the image and not scale it, which is why we need the second Group
			GUI.BeginGroup (new Rect (0,0,playerEnergy * 256, 32));

			// Draw the foreground image
			GUI.Box (new Rect (0,0,256,32), fgImage);

			// End both Groups
			GUI.EndGroup ();

		GUI.EndGroup ();
	}

}
```



*You can nest Groups together to create clipping behaviors*

**Automatic Layout - Areas**

Areas are used in Automatic Layout mode only. They are similar to Fixed Layout Groups in functionality, as they define a finite portion of the screen to contain GUILayout Controls. Because of the nature of Automatic Layout, you will nearly always use Areas.

In Automatic Layout mode, you do not define the area of the screen where the Control will be drawn at the Control level. The Control will automatically be placed at the upper-leftmost point of its containing area. This might be the screen. You can also create manually-positioned Areas. GUILayout Controls inside an area will be placed at the upper-leftmost point of that area.

```
/* A button placed in no area, and a button placed in an area halfway across the screen. */



// JavaScript
function OnGUI () {
	GUILayout.Button ("I am not inside an Area");
	GUILayout.BeginArea (Rect (Screen.width/2, Screen.height/2, 300, 300));
	GUILayout.Button ("I am completely inside an Area");
	GUILayout.EndArea ();
}



// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	void OnGUI () {
		GUILayout.Button ("I am not inside an Area");
		GUILayout.BeginArea (new Rect (Screen.width/2, Screen.height/2, 300, 300));
		GUILayout.Button ("I am completely inside an Area");
		GUILayout.EndArea ();
	}

}
```

Notice that inside an Area, Controls with visible elements like Buttons and Boxes will stretch their width to the full length of the Area.

**Automatic Layout - Horizontal and Vertical Groups**

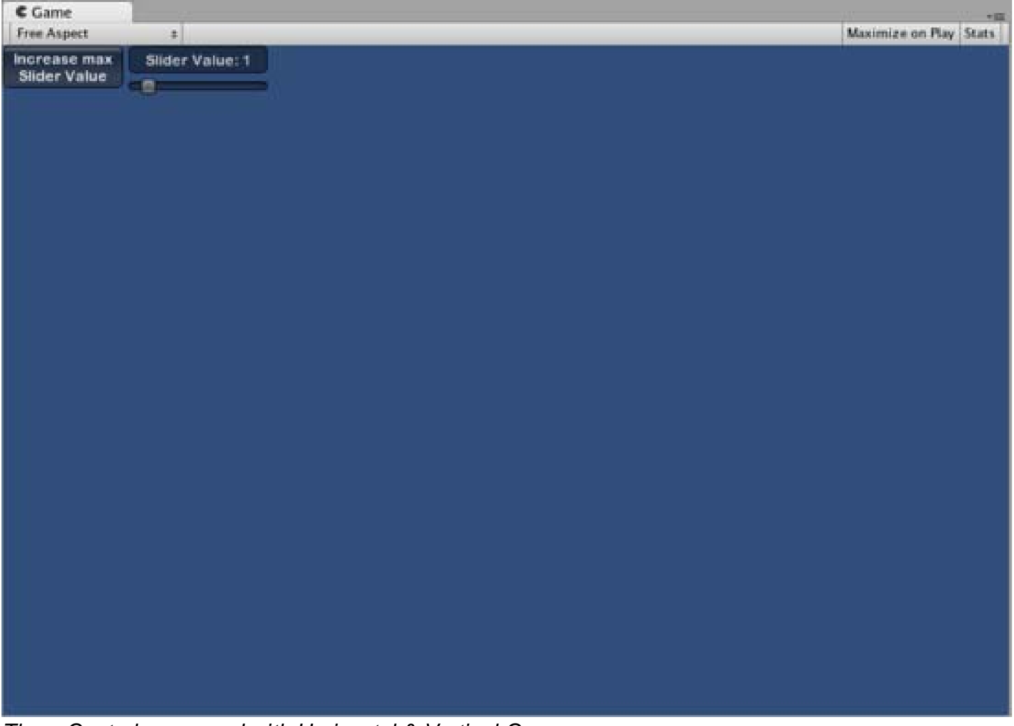
When using Automatic Layout, Controls will by default appear one after another from top to bottom. There are plenty of occasions you will want finer level of control over where your Controls are placed and how they are arranged. If you are using the Automatic Layout mode, you have the option of Horizontal and Vertical Groups.

Like the other layout Controls, you call separate functions to start or end these groups. The specific functions are **GUILayout.BeginHoriztontal()**, **GUILayout.EndHorizontal()**, **GUILayout.BeginVertical()**, and **GUILayout.EndVertical()**.

Any Controls inside a Horizontal Group will always be laid out horizontally. Any Controls inside a Vertical Group will always be laid out vertically. This sounds plain until you start nesting groups inside each other. This allows you to arrange any number of controls in any imaginable configuration.

```
/* Using nested Horizontal and Vertical Groups */



// JavaScript
var sliderValue = 1.0;
var maxSliderValue = 10.0;

function OnGUI()
{
```

```
	// Wrap everything in the designated GUI Area
	GUILayout.BeginArea (Rect (0,0,200,60));

	// Begin the singular Horizontal Group
	GUILayout.BeginHorizontal();

	// Place a Button normally
	if (GUILayout.RepeatButton ("Increase max\nSlider Value"))
	{
		maxSliderValue += 3.0 * Time.deltaTime;
	}

	// Arrange two more Controls vertically beside the Button
	GUILayout.BeginVertical();
	GUILayout.Box("Slider Value: " + Mathf.Round(sliderValue));
	sliderValue = GUILayout.HorizontalSlider (sliderValue, 0.0, maxSliderValue);

	// End the Groups and Area
	GUILayout.EndVertical();
	GUILayout.EndHorizontal();
	GUILayout.EndArea();
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	private float sliderValue = 1.0f;
	private float maxSliderValue = 10.0f;

	void OnGUI()
	{
		// Wrap everything in the designated GUI Area
		GUILayout.BeginArea (new Rect (0,0,200,60));

		// Begin the singular Horizontal Group
		GUILayout.BeginHorizontal();

		// Place a Button normally
		if (GUILayout.RepeatButton ("Increase max\nSlider Value"))
		{
			maxSliderValue += 3.0f * Time.deltaTime;
		}

		// Arrange two more Controls vertically beside the Button
		GUILayout.BeginVertical();
		GUILayout.Box("Slider Value: " + Mathf.Round(sliderValue));
		sliderValue = GUILayout.HorizontalSlider (sliderValue, 0.0f, maxSliderValue);

		// End the Groups and Area
		GUILayout.EndVertical();
		GUILayout.EndHorizontal();
		GUILayout.EndArea();
	}

}
```




*Three Controls arranged with Horizontal & Vertical Groups*

### Using GUILayoutOptions to define some controls

You can use GUILayoutOptions to override some of the Automatic Layout parameters. You do this by providing the options as the final parameters of the GUILayout Control.

Remember in the Areas example above, where the button stretches its width to 100% of the Area width? We can override that if we want to.

```
/* Using GUILayoutOptions to override Automatic Layout Control properties */


//JavaScript
function OnGUI () {
	GUILayout.BeginArea (Rect (100, 50, Screen.width-200, Screen.height-100));
	GUILayout.Button ("I am a regular Automatic Layout Button");
	GUILayout.Button ("My width has been overridden", GUILayout.Width (95));
	GUILayout.EndArea ();
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	void OnGUI () {
		GUILayout.BeginArea (new Rect (100, 50, Screen.width-200, Screen.height-100));
		GUILayout.Button ("I am a regular Automatic Layout Button");
		GUILayout.Button ("My width has been overridden", GUILayout.Width (95));
		GUILayout.EndArea ();
	}

}
```

For a full list of possible GUILayoutOptions, please read the GUILayoutOption Scripting Reference page.

Page last updated: 2012-01-18

---

# gui-Extending

There are a number of ways to leverage and extend UnityGUI to meet your needs. Controls can be mixed and created, and you have a lot of leverage into dictating how user input into the GUI is processed.

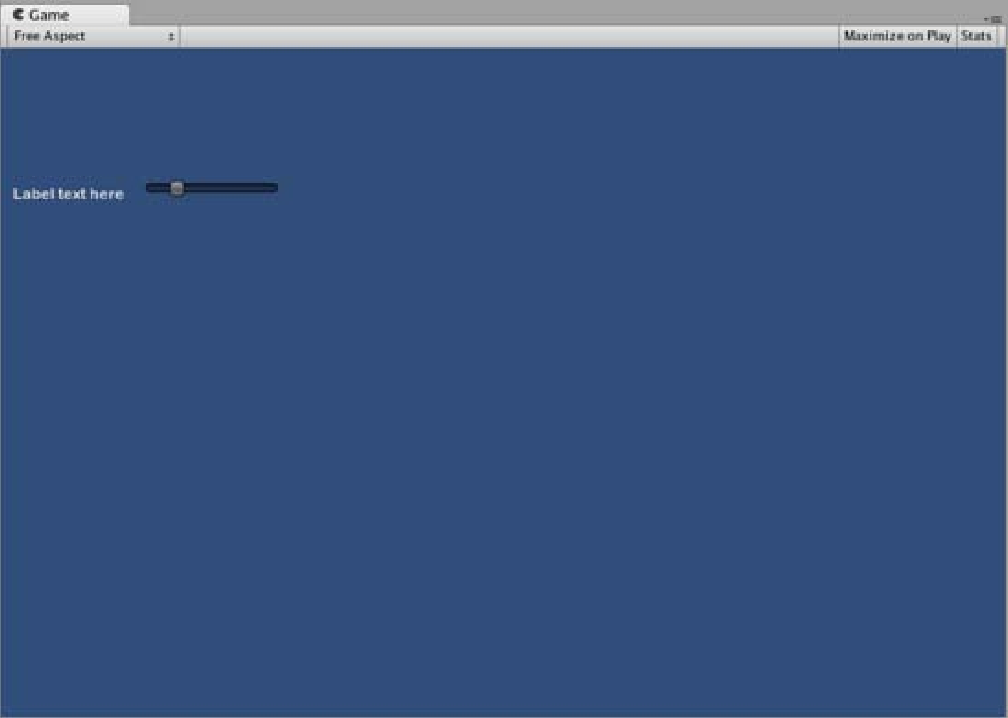
### Compound Controls

There might be situations in your GUI where two types of Controls always appear together. For example, maybe you are creating a Character Creation screen, with several Horizontal Sliders. All of those Sliders need a Label to identify them, so the player knows what they are adjusting. In this case, you could partner every call to **GUI.Label()** with a call to **GUI.HorizontalSlider()**, or you could create a **Compound Control** which includes both a Label and a Slider together.

```
/* Label and Slider Compound Control */



// JavaScript
var mySlider : float = 1.0;

function OnGUI () {
	mySlider = LabelSlider (Rect (10, 100, 100, 20), mySlider, 5.0, "Label text here");
}

function LabelSlider (screenRect : Rect, sliderValue : float, sliderMaxValue : float, labelText : String) : float {
	GUI.Label (screenRect, labelText);
	screenRect.x += screenRect.width; // <- Push the Slider to the end of the Label
	sliderValue = GUI.HorizontalSlider (screenRect, sliderValue, 0.0, sliderMaxValue);
	return sliderValue;
}



// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	private float mySlider = 1.0f;

	void OnGUI () {
		mySlider = LabelSlider (new Rect (10, 100, 100, 20), mySlider, 5.0f, "Label text here");
	}

	float LabelSlider (Rect screenRect, float sliderValue, float sliderMaxValue, string labelText) {
		GUI.Label (screenRect, labelText);

		// &lt;- Push the Slider to the end of the Label
		screenRect.x += screenRect.width;

		sliderValue = GUI.HorizontalSlider (screenRect, sliderValue, 0.0f, sliderMaxValue);
		return sliderValue;
	}

}
```

In this example, calling **LabelSlider()** and passing the correct arguments will provide a Label paired with a Horizontal Slider. When writing Compound Controls, you have to remember to return the correct value at the end of the function to make it

interactive.




*The above Compound Control always creates this pair of Controls*

**Static Compound Controls**

By using **Static** functions, you can create an entire collection of your own Compound Controls that are self-contained. This way, you do not have to declare your function in the same script you want to use it.

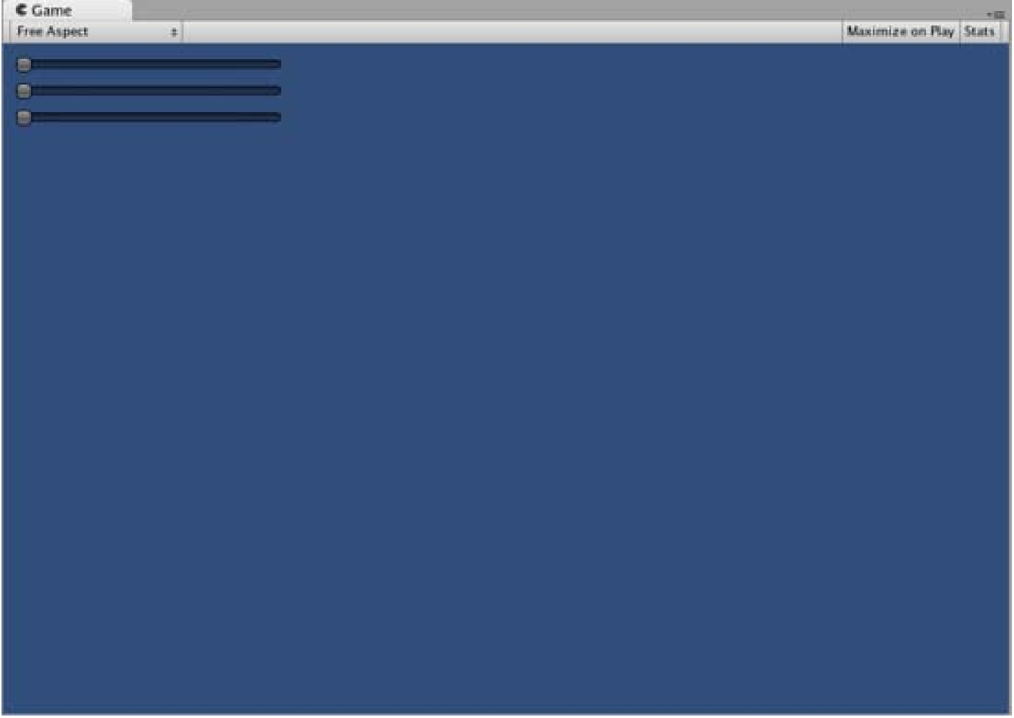
```
/* This script is called CompoundControls */


// JavaScript
static function LabelSlider (screenRect : Rect, sliderValue : float, sliderMaxValue : float, labelText : String) : float {
	GUI.Label (screenRect, labelText);
	screenRect.x += screenRect.width; // <- Push the Slider to the end of the Label
	sliderValue = GUI.HorizontalSlider (screenRect, sliderValue, 0.0, sliderMaxValue);
	return sliderValue;
}


// C#
using UnityEngine;
using System.Collections;

public class CompoundControls : MonoBehaviour {

	public static float LabelSlider (Rect screenRect, float sliderValue, float sliderMaxValue, string labelText) {
		GUI.Label (screenRect, labelText);

		// &lt;- Push the Slider to the end of the Label
		screenRect.x += screenRect.width;

		sliderValue = GUI.HorizontalSlider (screenRect, sliderValue, 0.0f, sliderMaxValue);
		return sliderValue;
	}

}
```

By saving the above example in a script called **CompoundControls**, you can call the **LabelSlider()** function from any other script by simply typing **CompoundControls.LabelSlider()** and providing your arguments.

**Elaborate Compound Controls**

You can get very creative with Compound Controls. They can be arranged and grouped in any way you like. The following example creates a re-usable RGB Slider.

```
/* RGB Slider Compound Control */



// JavaScript
var myColor : Color;

function OnGUI () {
	myColor = RGBSlider (Rect (10,10,200,10), myColor);
}

function RGBSlider (screenRect : Rect, rgb : Color) : Color {
	rgb.r = GUI.HorizontalSlider (screenRect, rgb.r, 0.0, 1.0);
	screenRect.y += 20; // <- Move the next control down a bit to avoid overlapping
	rgb.g = GUI.HorizontalSlider (screenRect, rgb.g, 0.0, 1.0);
	screenRect.y += 20; // <- Move the next control down a bit to avoid overlapping
	rgb.b = GUI.HorizontalSlider (screenRect, rgb.b, 0.0, 1.0);
	return rgb;
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	public Color myColor;

	void OnGUI () {
		myColor = RGBSlider (new Rect (10,10,200,10), myColor);
	}

	Color RGBSlider (Rect screenRect, Color rgb) {
		rgb.r = GUI.HorizontalSlider (screenRect, rgb.r, 0.0f, 1.0f);

		// &lt;- Move the next control down a bit to avoid overlapping
		screenRect.y += 20;
		rgb.g = GUI.HorizontalSlider (screenRect, rgb.g, 0.0f, 1.0f);

		// &lt;- Move the next control down a bit to avoid overlapping
		screenRect.y += 20;

		rgb.b = GUI.HorizontalSlider (screenRect, rgb.b, 0.0f, 1.0f);
		return rgb;
	}
}
```

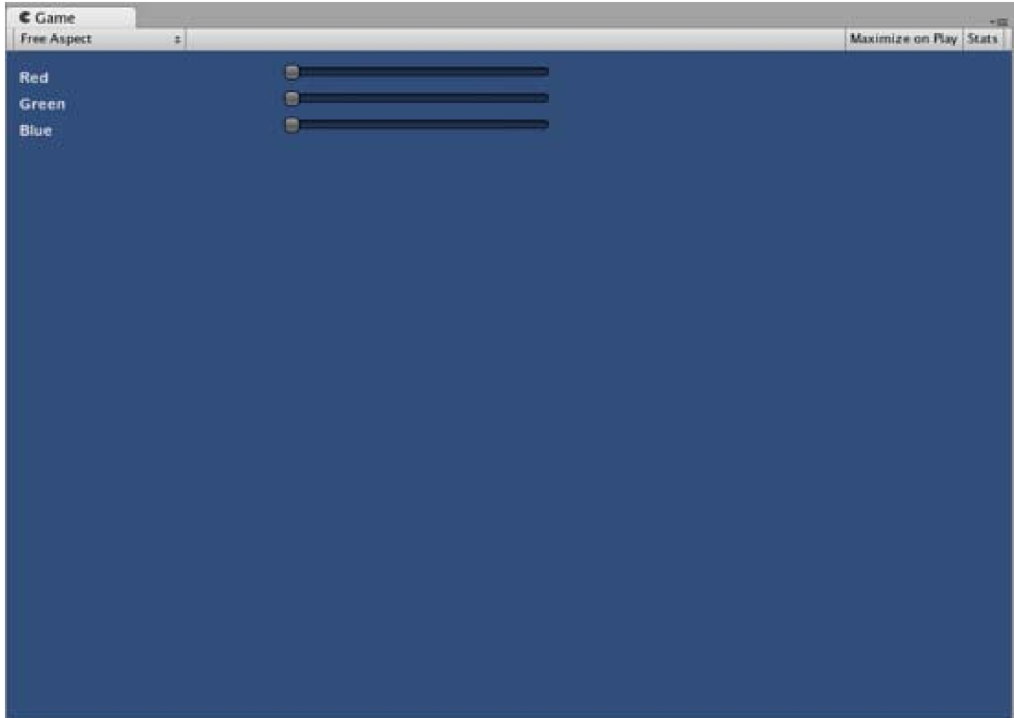



*The RGB Slider created by the example above*

Now let's build Compound Controls on top of each other, in order to demonstrate how Compound Controls can be used within other Compound Controls. To do this, we will create a new RGB Slider like the one above, but we will use the LabelSlider to do so. This way we'll always have a Label telling us which slider corresponds to which color.

```
/* RGB Label Slider Compound Control */


// JavaScript
var myColor : Color;

function OnGUI () {
	myColor = RGBLabelSlider (Rect (10,10,200,20), myColor);
}

function RGBLabelSlider (screenRect : Rect, rgb : Color) : Color {
	rgb.r = CompoundControls.LabelSlider (screenRect, rgb.r, 1.0, "Red");
	screenRect.y += 20; // <- Move the next control down a bit to avoid overlapping
	rgb.g = CompoundControls.LabelSlider (screenRect, rgb.g, 1.0, "Green");
	screenRect.y += 20; // <- Move the next control down a bit to avoid overlapping
	rgb.b = CompoundControls.LabelSlider (screenRect, rgb.b, 1.0, "Blue");
	return rgb;
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

	public Color myColor;

	void OnGUI () {
		myColor = RGBSlider (new Rect (10,10,200,30), myColor);
	}
```

```
    Color RGBSlider (Rect screenRect, Color rgb) {
        rgb.r =  CompoundControls.LabelSlider (screenRect, rgb.r, 1.0f, "Red");

        // &lt;- Move the next control down a bit to avoid overlapping
        screenRect.y += 20;
        rgb.g = CompoundControls.LabelSlider (screenRect, rgb.g, 1.0f, "Green");

        // &lt;- Move the next control down a bit to avoid overlapping
        screenRect.y += 20;

        rgb.b = CompoundControls.LabelSlider (screenRect, rgb.b, 1.0f, "Blue");

        return rgb;
    }

}
```




*The Compound RGB Label Slider created by the above code*

Page last updated: 2012-01-17

---

## gui-ExtendingEditor

### Introduction

You can create your own custom design tools inside Unity through **Editor Windows**. Scripts that derive from EditorWindow instead of MonoBehaviour can leverage both GUI/GUILayout and EditorGUI/EditorGUILayout controls. Alternatively, you can use **Custom Inspectors** to expose these GUI controls in your GameObject Inspector.

### Editor Windows

You can create any number of custom windows in your app. These behave just like the Inspector, Scene or any other built-in ones. This is a great way to add a user interface to a sub-system for your game.

*Custom Editor Interface by Serious Games Interactive used for scripting cutscene actions*

Making a custom Editor Window involves the following simple steps:

- Create a script that derives from EditorWindow.
- Use code to trigger the window to display itself.
- Implement the GUI code for your tool.

### Derive From EditorWindow

In order to make your Editor Window, your script must be stored inside a folder called "Editor". Make a class in this script that derives from EditorWindow. Then write your GUI controls in the inner OnGUI function.

```
class MyWindow extends EditorWindow {
    function OnGUI () {
        // The actual window code goes here
    }
}
```

*MyWindow.js - placed in a folder called 'Editor' within your project.*

### Showing the window

In order to show the window on screen, make a menu item that displays it. This is done by creating a function which is activated by the **MenuItem** property.

The default behavior in Unity is to recycle windows (so selecting the menu item again would show existing windows. This is done by using the function EditorWindow.GetWindow Like this:

```
class MyWindow extends EditorWindow {
    @MenuItem ("Window/My Window")
    static function ShowWindow () {
        EditorWindow.GetWindow (MyWindow);
    }

    function OnGUI () {
        // The actual window code goes here
    }
}
```

*Showing the MyWindow*

This will create a standard, dockable editor window that saves its position between invocations, can be used in custom layouts, etc. To have more control over what gets created, you can use GetWindowWithRect

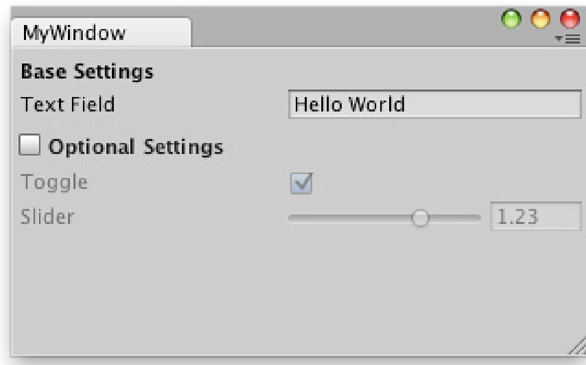
### Implementing Your Window's GUI

The actual contents of the window are rendered by implementing the OnGUI function. You can use the same UnityGUI classes you use for your ingame GUI (**GUI** and **GUILayout**). In addition we provide some additional GUI controls, located in the editor-only classes **EditorGUI** and **EditorGUILayout**. These classes add to the controls already available in the normal classes, so you can mix and match at will.

The following C# code shows how you can add GUI elements to your custom EditorWindow:

```
using UnityEditor;
using UnityEngine;

public class MyWindow : EditorWindow
{
	string myString = "Hello World";
	bool groupEnabled;
	bool myBool = true;
	float myFloat = 1.23f;

	// Add menu item named "My Window" to the Window menu
	[MenuItem("Window/My Window")]
	public static void ShowWindow()
	{
		//Show existing window instance. If one doesn't exist, make one.
		EditorWindow.GetWindow(typeof(MyWindow));
	}

	void OnGUI()
	{
		GUILayout.Label ("Base Settings", EditorStyles.boldLabel);
		myString = EditorGUILayout.TextField ("Text Field", myString);

		groupEnabled = EditorGUILayout.BeginToggleGroup ("Optional Settings", groupEnabled);
			myBool = EditorGUILayout.Toggle ("Toggle", myBool);
			myFloat = EditorGUILayout.Slider ("Slider", myFloat, -3, 3);
		EditorGUILayout.EndToggleGroup ();
	}
}
```

This example results in a window which looks like this:




*Custom Editor Window created using supplied example.*

For more info, take a look at the example and documentation on the EditorWindow page.

## Custom Inspectors

A key to increasing the speed of game creation is to create custom inspectors for commonly used components. For the sake of example, we'll use this very simple script that always keeps an object looking at a point.

```
var lookAtPoint = Vector3.zero;

function Update () {
   transform.LookAt (lookAtPoint);
}
```

*LookAtPoint.js*

This will keep an object oriented towards a world-space point. Let's make it cool!

The first step to making it work nicely in the editor is to make the script run even when you're not testing the game. We do this by adding an ExecuteInEditMode attribute to it:

```
@script ExecuteInEditMode()

var lookAtPoint = Vector3.zero;

function Update () {
   transform.LookAt (lookAtPoint);
}
```

Try adding the script to your main camera and drag it around in the Scene view.

**Making a Custom Editor**

This is all well and good, but we can make the inspector for it a lot nicer by customizing the inspector. To do that we need to create an **Editor** for it. Create a JavaScript called LookAtPointEditor in a folder called Editor.

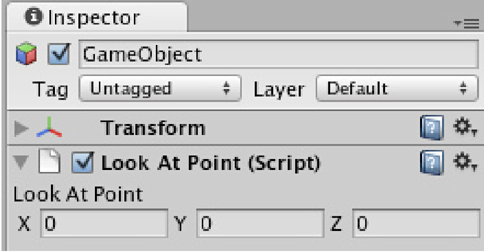
```
@CustomEditor (LookAtPoint)
class LookAtPointEditor extends Editor {
   function OnInspectorGUI () {
      target.lookAtPoint = EditorGUILayout.Vector3Field ("Look At Point", target.lookAtPoint);
      if (GUI.changed)
         EditorUtility.SetDirty (target);
   }
}
```

This class has to derive from Editor. The @CustomEditor attribute informs Unity which component it should act as an editor for.

The code in OnInspectorGUI is exectued whenever Unity displays the inspector. You can put any GUI code in here - it works just like OnGUI does for games, but is run inside the inspector. Editor defines the target property that you can use to access the object being inspected.

The EditorUtility.SetDirty code is executed if the user has changed any of the values by checking GUI.changed.

In this case, we make one of the Vector3 fields like is used in the Transform Inspector - like so:




*Yay for shiny inspectors*

There's a lot more that can be done here, but this will do for now - We've got bigger fish to fry...

**Scene View Additions**

You can add extra code to the Scene View by implementing an OnSceneGUI in your custom editor. In this case, we'll add a second set of position handles, letting users drag the look-at point around in the Scene view.

```
@CustomEditor (LookAtPoint)
class LookAtPointEditor extends Editor {
```

```
    function OnInspectorGUI () {
        target.lookAtPoint = EditorGUILayout.Vector3Field ("Look At Point", target.lookAtPoint);
        if (GUI.changed)
            EditorUtility.SetDirty (target);
    }

    function OnSceneGUI () {
        target.lookAtPoint = Handles.PositionHandle (target.lookAtPoint, Quaternion.identity);
        if (GUI.changed)
            EditorUtility.SetDirty (target);
    }
}
```

OnSceneGUI works just like OnInspectorGUI - except it gets run in the scene view. To help you make your editing interface, you can use the functions defined in Handles class. All functions in there are designed for working in 3D Scene views.

If you want to put 2D GUI objects (GUI, EditorGUI and friends), you need to wrap them in calls to Handles.BeginGUI() and Handles.EndGUI().

Page last updated: 2012-05-31

---

# Network Reference Guide

Networking is a very large, detailed topic. In Unity, it is extremely simple to create network functionality. However, it is still best to understand the breadth and depth involved with creating any kind of network game. The following page will explain the fundamentals of networking concepts, and the Unity-specific executions of these concepts for you to use. If you have never created a network game before, it is highly recommended that you read this guide in detail before attempting to create one.

### High Level Overview
This section will outline all the concepts involved in networking. It will serve as an introduction to deeper topics.

### Networking Elements in Unity
This section of the guide will cover Unity's execution of the ideas discussed above.

### Network View
Network Views are Components you use to share data across the network. They are extremely important to understand. This page will explain them in detail.

### RPC Details
RPC stands for Remote Procedure Call. It is a way of calling a function on a remote machine. This may be a client calling a function on the server, or the server calling a function on all or specific clients, etc. This page explains RPC concepts in detail.

### State Synchronization
State Synchronization is a method of regularly updating a specific set of data across two or more game instances running on the network.

### Network Instantiate
One difficult subject in networking is ownership of an object. Who controls what? Network Instantiation will determine this logic for you. This page will explain how to do this. It will also explain the complex alternatives, for situations when you just need more control.

### Master Server
The Master Server is like a game lobby where servers can advertise their presence to clients. It is also a solution to enabling communication from behind a firewall or home network. When needed it makes it possible to use a technique called NAT punchthrough (with help from a facilitator) to make sure your players can always connect with each other. This page will explain how to use the Master Server.

### Minimizing Bandwidth

Every choice you make about where and how to share data will affect the bandwidth your game uses. This page will share some details about bandwidth usage and how to keep it to a minimum.

## iOS

### Special details about networking on iOS

Boot up with Networking for iOS.

## Android

### Special details about networking on Android

Page last updated: 2011-02-22

---

# Networking on iOS

# iOS and Android

### Networking for mobile devices (iOS / Android)

The Unity iOS/Android Networking engine is fully compatible with networking for desktop devices, so your existing networking code should work on iOS/Android devices. However, you may want to re-engineer your code if it is mainly to be used with Wi-Fi or cellular networks. Moreover, depending on the mobile, the networking chip may also be the bottleneck since pings between mobile devices (or between the mobile device and the desktop) are about 40-60 ms, even in high performance Wi-Fi networks.

Using Networking you can create a game that can be played simultaneously from desktop and iOS over Wi-Fi or cellular networks. In the latter case, your game server should have a public IP address (accessible through the internet).

**Note:** EDGE / 3G data connections go to sleep very quickly when no data is sent. Thus sometimes you may need to "wake-up" networking. Just make the WWW class connect to your site (and yield until it finishes) before making the Unity networking connection..

Page last updated: 2011-11-08

---

# net-HighLevelOverview

This section covers general networking concepts that should be understood before developing a game with Unity's networking architecture.

### What is Networking?

Networking is communication between two or more computers. A fundamental idea is that of the relationship between the **client** (the computer that is requesting information) and the **server** (the computer responding to the information request). The server can either be a dedicated host machine used by all clients, or simply a player machine running the game (client) but also acting as the server for other players. Once a server has been established and a client has connected to it, the two computers can exchange data as demanded by gameplay.

Creating a network game requires a lot of attention to some very specific details. Even though network actions are easy to design and create in Unity, networking remains rather complex. A major design decision in Unity is to make networking as robust and flexible as possible. This means that you, as the game creator, are responsible for things that might be handled in an automatic but less robust way in other engines. The choices you make potentially have a major effect on the design of your game, so it is best to make them as early in the design stage as possible. Understanding networking concepts will help you

plan your game design well and avoid problems during the implementation.

## Networking Approaches

There are two common and proven approaches to structuring a network game which are known as **Authoritative Server** and **Non-Authoritative Server**. Both approaches rely on a server connecting clients and passing information between them. Both also offer privacy for end users since clients never actually connect directly with each other or have their IP addresses revealed to other clients.

### Authoritative Server

The authoritative server approach requires the server to perform all world simulation, application of game rules and processing of input from the player clients. Each client sends their input (in the form of keystrokes or requested actions) to the server and continuously receives the current state of the game from the server. The client never makes any changes to the game state itself. Instead, it tells the server what it wants to do, and the server then handles the request and replies to the client to explain what happened as a result.

Fundamentally, there is a layer of separation between what the player wants to do and what actually happens. This allows the server to listen to every client's requests before deciding how to update the game state.

An advantage of this approach is that it makes cheating much harder for clients. For example, clients do not have the possibility of cheating by telling the server (and thereby other clients) that an enemy has been killed, since they don't get to make that decision by themselves. They can only tell the server that a weapon was fired and from there, it is up to the server to determine whether or not a kill was made.

Another example of an authoritative server would be a multiplayer game that relies on physics. If each client is allowed to run its own physics simulation then small variations between clients could cause them to drift out of sync with each other gradually. However, if the simulation of all physics objects is handled on a central server then the updated state can be sent back to the clients, guaranteeing they are all consistent.

A potential disadvantage with authoritative servers is the time it takes for the messages to travel over the network. If the player presses a control to move forward and it takes a tenth of a second for the response to return from the server then the delay will be perceptible to the player. One solution to this is to use so-called **client-side prediction**. The essence of this technique is that the client is allowed to update its own local version of the game state but it must be able to receive corrections from the server's authoritative version where necessary. Typically, this should only be used for simple game actions and not significant logical changes to game state. For example, it would be unwise to report to a player that an enemy has been killed only for the server to override this decision.

Since client-side prediction is an advanced subject, we don't attempt to cover it in this guide but books and web resources are available if you want to investigate it further.

An authoritative server has a greater processing overhead than a non-authoritative one. When the server is not required to handle all changes to game state, a lot of this load can be distributed between the clients.

### Non-Authoritative Server

A non-authoritative server does not control the outcome of every user input. The clients themselves process user input and game logic locally, then send the result of any determined actions to the server. The server then synchronizes all actions with the world state. This is easier to implement from a design perspective, as the server really just relays messages between the clients and does no extra processing beyond what the clients do.

There is no need for any kind of *prediction* methods as the clients handle all physics and events themselves and relay what happened to the server. They are the *owners* of their objects and are the only agents permitted to send local modifications of those objects over the network.

## Methods of Network Communication

Now that we've covered the basic architectures of networked games, we will explore the lower-levels of how clients and servers can talk to each other.

There are two relevant methods: **Remote Procedure Calls** and **State Synchronization**. It is not uncommon to use both methods at different points in any particular game.

### Remote Procedure Calls

Remote Procedure Calls (RPCs) are used to invoke functions on other computers across the network, although the "network" can also mean the message channel between the client and server when they are both running on the same computer. Clients can send RPCs to the server, and the server can send RPCs to one or more clients. Most commonly, they are used for actions that happen infrequently. For example, if a client flips a switch to open a door, it can send an RPC to the server telling it that the door has been opened. The server can then send another RPC to all clients, invoking their local functions to open that same door. They are used for managing and executing individual events.

**State Synchronization**
State Synchronization is used to share data that is constantly changing. The best example of this would be a player's position in an action game. The player is always moving, running around, jumping, etc. All the other players on the network, even the ones that are not controlling this player locally, need to know where he is and what he is doing. By constantly relaying data about this player's position, the game can accurately represent that position to the other players.

This kind of data is regularly and frequently sent across the network. Since this data is time-sensitive, and it requires time to travel across the network from one machine to another, it is important to reduce the amount of data that is sent as far as possible. In simpler terms, state synchronization naturally requires a lot of bandwidth, so you should aim to use as little bandwidth as possible.

**Connecting servers and clients together**
Connecting servers and clients together can be a complex process. Machines can have private or public IP addresses and they can have local or external firewalls blocking access. Unity networking aims to handle as many situations as possible but there is no universal solution.

Private addresses are IP addresses which are not accessible directly from the internet (they are also called Network Address Translation or NAT addresses after the method used to implement them). Simply explained, the private address goes through a local router which translates the address into a public address. By doing this, many machines with private addresses can use a single public IP address to communicate with the internet. This is fine until someone elsewhere on the internet wants to initiate contact with one of the private addresses. The communication must happen via the public address of the router, which must then pass the message on to the private address. A technique called NAT punchthrough uses a shared server known as a *facilitator* to mediate communication in such a way that the private address can be reached from the public address. This works by having the private address first contact the facilitator, which "punches" a hole through the local router. The facilitator can now see the public IP address and port which the private address is using. Using this information, any machine on the internet can now connect directly with the otherwise unreachable private address. (Note that the details of NAT punchthrough are somewhat more complicated than this in practice.)

Public addresses are more straightforward. Here, the main issue is that connectivity can be blocked by an internal or external firewall (an internal firewall is one that runs locally on the computer it is protecting). For an internal firewall, the user can be asked to remove restrictions from a particular port so as to make the game server accessible. An external firewall, by contrast, is not under the control of the users. Unity can attempt to use NAT punchthrough to get access through an external firewall but this technique is not guaranteed to succeed. Our testing suggests that it generally works in practice but there doesn't appear to be any formal research that confirms this finding.

The connectivity issues just mentioned affect servers and clients differently. Client requests involve only outgoing network traffic which is relatively straightforward. If the client has a public address then this almost always works since outgoing traffic is typically only blocked on corporate networks that impose severe access restrictions. If the client has a private address it can connect to all servers except servers with private addresses which cannot do NAT punchthrough (more will be said about this later). The server end is more complicated because the server needs to be able to accept incoming connections from unknown sources. With a public address, the server needs to have the game port open to the internet (ie, not blocked by a firewall). or else it cannot accept any connections from clients and is thus unusable. If the server has a private address it must be able to do NAT punchthrough to allow connections and clients must also permit NAT punchthrough in order to connect to it.

Unity provides tools to test all these different connectivity situations. When it is established that a connection can be made, there are two methods by which it can happen: direct connections (where a client needs to know the DNS name or IP address of the server) and connections via the Master Server. The Master Server allows servers to advertise their presence to clients which need not know anything about particular game servers beforehand.

## Minimizing Network Bandwidth

When working with State Synchronization across multiple clients, you don't necessarily need to synchronize every single detail in order to make objects appear synchronized. For example, when synchronizing a character avatar you only need to send its position and rotation between clients. Even though the character itself is much more complex and might contain a deep **Transform** hierarchy, data about the entire hierarchy does not need to be shared.

A lot of data in your game is effectively static, and clients need neither transfer it initially nor synchronize it. Using infrequent or one-time RPC calls should be sufficient to make a lot of your functionality work. Take advantage of the data you know will exist in every installation of your game and keep the client working by itself as much as possible. For example, you know that assets like textures and meshes exist on all installations and they usually don't change, so they will never have to be synchronized. This is a simple example but it should get you thinking about what data is absolutely critical to share from one client to another. This is the only data that you should ever share.

It can be difficult to work out exactly what needs to be shared and what doesn't, especially if you have never made a network game before. Bear in mind that you can use a single RPC call with a level name to make all clients load the entire specified level and add their own networked elements automatically. Structuring your game to make each client as self-sufficient as possible will result in reduced bandwidth.

### Multiplayer Game Performance

The physical location and performance of the server itself can greatly affect the playability of a game running on it. Clients which are located a continent away from the server may experience a great deal of lag. This is a physical limitation of the internet and the only real solution is to arrange for the server to be as close as possible to the clients who will use it, or at least on the same continent.

### Extra Resources

We've collected the following links to additional resources about networking:-

- http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking
- http://developer.valvesoftware.com/wiki/Lag_Compensation
- http://developer.valvesoftware.com/wiki/Working_With_Prediction
- http://www.gamasutra.com/resource_guide/20020916/lambright_01.htm

Page last updated: 2011-11-18

---

## net-UnityNetworkElements

Unity's native networking supports everything discussed on the previous page. Server creation and client connection, sharing data between connected clients, determining which player controls which objects, and punching through network configuration variations are all supported out of the box. This page will walk you through the Unity-specific implementation of these networking practices.

### Creating a Server

Before you can begin playing a networked game, you have to determine the different computers you will be communicating with. To do this, you have to create a server. This can be a machine that is also running the game or it can be a dedicated machine that is not participating in the game. To create the server, you simply call Network.InitializeServer() from a script. When you want to connect to an existing server as a client, you call Network.Connect() instead.

In general, you will find it very useful to familiarize yourself with the entire Network class.

### Communicating using Network Views

The **Network View** is a Component that sends data across the network. Network Views make your GameObject capable of sending data using RPC calls or State Synchronization. The way you use Network Views will determine how your game's networking behaviors will work. Network Views have few options, but they are incredibly important for your networked game.

For more information on using Network Views, please read the Network View Guide page and Component Reference page.

### Remote Procedure Calls

Remote Procedure Calls (RPCs) are functions declared in scripts that are attached to a GameObject that contains a Network View. The Network View must point to the script which contains the RPC function. The RPC function can then be called from any script within that GameObject.

For more information on using RPCs in Unity, please read the RPC Details page.

### State Synchronization

State Synchronization is the continual sharing of data across all game clients. This way a player's position can be

synchronized over all clients, so it seems it is controlled locally when data is actually being delivered over a network. To synchronize state within a GameObject you just need to add a NetworkView to that object and tell it what to observe. The observed data is then synchronized across all clients in the game.

For more information on using State Synchronization in Unity, please read the State Synchronization page.

### Network.Instantiate()

**Network.Instantiate()** lets you instantiate a prefab on all clients in a natural and easy way. Essentially this is an **Instantiate()** call, but it performs the instantiation on all clients.

Internally Network.Instantiate is simply a buffered RPC call which is executed on all clients (also locally). It allocates a NetworkViewID and assigns it to the instantiated prefab which makes sure it synchronizes across all clients correctly.

For more information please read the Network Instantiate page.

### NetworkLevelLoad()

Dealing with sharing data, state of client players, and loading levels can be a bit overwhelming. The Network Level Load page contains a helpful example for managing this task.

### Master Server

The **Master Server** helps you match games. When you start a server you connect to the master server, and it provides a list of all the active servers.

The **Master Server** is a meeting place for servers and clients where servers are advertised and compatible clients can connect to running games. This prevents the need for fiddling with IP addresses for all parties involved. It can even help users host games without them needing to mess with their routers where, under normal circumstances, that would be required. It can help clients bypass the server's firewall and get to private IP addresses which are normally not accessible through the public internet. This is done with help from a facilitator which *facilitates* connection establishment.

For more information please read the Master Server page.

### Minimizing Bandwidth

Using the minimum amount of bandwidth to make your game run correctly is essential. There are different methods for sending data, different techniques for deciding *what* or *when* to send and other tricks at your disposal.

For tips and tricks to reduce bandwidth usage, please read the Minimizing Bandwith page.

### Debugging Networked Games

Unity comes with several facilities to help you debug your Networked game.

1. The Network Manager can be used for logging all incoming and outgoing network traffic.
2. Using the Inspector and Hierarchy View effectively you can track object creation and inspect view id's etc.
3. You can launch Unity two times on the same machine, and open different projects in each. On Windows, this can be done by just launching another Unity instance and opening the project from the project wizard. On Mac OS X, multiple Unity instances can be opened from the terminal, and a **-projectPath** argument can be specified:

```
/Applications/Unity/Unity.app/Contents/MacOS/Unity -projectPath "/Users/MyUser/MyProjectFol
/Applications/Unity/Unity.app/Contents/MacOS/Unity -projectPath "/Users/MyUser/MyOtherProjec
```

Make sure you make the player run in the background when debugging networking because, for example, if you have two instances running at once, one of them doesn't have focus. This will break the networking loop and cause undesirable results. You can enable this in Edit->Project Settings->Player in the editor or with Application.runInBackground

Page last updated: 2010-07-19

---

## net-NetworkView

Network Views are the main component involved in sharing data across the network. They allow two kinds of network

communication: **State Synchronization** and **Remote Procedure Calls**.

Network Views keep watch on particular objects to detect changes. These changes are then shared to the other clients on the network to ensure the change of state is noted by all of them. This concept is known as *state synchronization* and you can read about it further on the State Synchronization page.

There are some situations where you would not want the overhead of synchronizing state between clients, for example, when sending out the position of a new object or respawned player. Since events like this are infrequent, it does not make sense to synchronize the state of the involved objects. Instead, you can use a *remote procedure call* to tell the clients or server to perform operations like this. More information about Remote Procedure Calls can be found on the RPC manual page.

### Technical Details

A Network View is identified across the network by its **NetworkViewID** which is basically just a identifier which is negotiated to be unique among the networked machines. It is represented as a 128 bit number but is automatically compressed down to 16 bits when transferred over the network if possible.

Each packet that arrives on the client side needs to be applied to a specific Network View as specified by the NetworkViewID. Using this, Unity can find the right Network View, unpack the data and apply the incoming packet to the Network View's observed object.

More details about using Network Views in the Editor can be found on the Network View Component Reference page.

If you use Network.Instantiate() to create your Networked objects, you do not need to worry about allocating Network Views and assigning them yourself appropriately. It will all work automatically behind the scenes.

However, you can manually set the **NetworkViewID** values for each Network View by using Network.AllocateViewID. The Scripting Reference documentation shows an example of how an object can be instantiated manually on every client with an RPC function and then the NetworkViewID manually set with **AllocateViewID**.

Page last updated: 2011-11-22

---

## net-RPCDetails

**Remote Procedure Calls** (RPCs) let you call functions on a remote machine. Invoking an RPC is similar to calling a normal function and almost as easy but there are some important differences to understand.

1. An RPC call can have as many parameters as you like but the network bandwidth involved will increase with the number and size of parameters. You should keep parameters to a minimum in order to get the best performance.
2. Unlike a normal function call, an RPC needs an additional parameter to denote the recipients of the RPC request. There are several possible RPC call modes to cover all common use cases. For example, you can easily invoke the RPC function on all connected machines, on the server alone, on all clients but the one sending the RPC call or on a specific client.

RPC calls are usually used to execute some event on all clients in the game or pass event information specifically between two parties, but you can be creative and use them however you like. For example, a server for a game which only starts after four clients have connected could send an RPC call to all clients as soon as the fourth one connects, thus starting the game. A client could send RPC calls to everyone to signal that he picked up an item. A server could send an RPC to a particular client to initialize him right after he connects, for example, to give him his player number, spawn location, team color, etc. A client could in turn send an RPC only to the server to specify his starting options, such as the color he prefers or the items he has bought.

### Using RPCs

A function must be marked as an RPC before it can be invoked remotely. This is done by prefixing the function in the script with an RPC attribute:-

```
// All RPC calls need the @RPC attribute!
```

```
@RPC
function PrintText (text : String)
{
   Debug.Log(text);
}
```

All network communication is handled by NetworkView components, so you must attach one to the object whose script declares the RPC functions before they can be called.

### Parameters

You can use the following variable types as parameters to RPCs:-

- int
- float
- string
- NetworkPlayer
- NetworkViewID
- Vector3
- Quaternion

For example, the following code invokes an RPC function with a single string parameter:-

```
networkView.RPC ("PrintText", RPCMode.All, "Hello world");
```

The first parameter of **RPC()** is the name of the function to be invoked while the second determines the targets on which it will be invoked. In this case we invoke the RPC call on everyone who is connected to the server (but the call will not be buffered to wait for clients who connect later - see below for further details about buffering).

All parameters after the first two are the ones that will be passed to the RPC function and be sent across the network. In this case, "Hello World" will be sent as a parameter and be passed as the text parameter in the PrintText function.

You can also access an extra internal parameter, a NetworkMessageInfo struct which holds additional information, such as where the RPC call came from. This information will be passed automatically, so the PrintText function shown above will be can be declared as:-

```
@RPC
function PrintText (text : String, info : NetworkMessageInfo)
{
   Debug.Log(text + " from " + info.sender);
}
```

...while being invoked the same way as before.

As mentioned above, a Network View must be attached to any GameObject which has a script containing RPC functions. If you are using RPCs exclusively (ie, without state synchronisation) then the Network View's **State Synchronization** can be set to **Off**.

### RPC Buffer

RPC calls can also be buffered. Buffered RPC calls are stored up and executed in the order they were issued for each new client that connects. This can be a useful way to ensure that a latecoming player gets all necessary information to start. A common scenario is that every player who joins a game should first load a specific level. You could send the details of this level to all connected players but also buffer it for any who join in the future. By doing this, you ensure that the new player receives the level information just as if he had been present from the start.

You can also remove calls from the RPC buffer when necessary. Continuing the example above, the game may have moved on from the starting level by the time a new player joins, so you could remove the original buffered RPC and send a new one to request the new level.

Page last updated: 2011-11-18

---

# net-StateSynchronization

You can enable State Synchronization for a given Network View by choosing either **Reliable Delta Compressed** or **Unreliable** from the **State Synchronization** drop-down. You must then choose what kind of data will be synchronized using the *Observed* property.

Unity can synchronize Transform, Animation, Rigidbody and MonoBehaviour components.

Transforms are serialized by storing position, rotation and scale. Parenting information is not transferred over the network.

Animation serializes each running animation state, that is the time, weight, speed and enabled properties.

Rigidbody serializes position, rotation, velocity and angular velocity.

Scripts (MonoBehaviours) call the function OnSerializeNetworkView().

## Reliability and bandwidth

Network Views currently support two reliability levels Reliable Delta Compressed and Unreliable.

Both have their advantages and disadvantages, and the specific details of the game will determine which is the best to use.

For additional information about minimizing bandwidth, please read the Minimizing Bandwidth page.

### Reliable Delta Compressed

**Reliable Delta Compressed** mode will automatically compare the data that was last received by the client to the current state. If no data has changed since the last update then no data will be sent. However, the data will be compared on a per property basis. For example, if the Transform's position has changed but its rotation has not then only the position will be sent across the network. Bandwidth is saved by transmitting only the changed data.

Unity will also ensure that every UDP packet arrives reliably by resending it until receipt is determined. This means that if a packet is dropped, any packets sent later will not be applied until the dropped packet is re-sent and received. Until then, all later packets will be kept waiting in a buffer.

### Unreliable

In **Unreliable** mode, Unity will send packets without checking that they have been received. This means that it doesn't know which information has been received and so it is not safe to send only the changed data - the whole state will be sent with each update.

## Deciding which method to use

The Network layer uses UDP, which is an unreliable, unordered protocol but it can used to send ordered packets reliably, just like TCP does. To do this, Unity internally uses ACKs and NACKs to control packet transmission, ensuring no packets are dropped. The downside to using reliable ordered packets is that if a packet is dropped or delayed, everything stops until that packet has arrived safely. This can cause transmission delays where there is significant network lag.

Unreliable transmission is useful when you know that data will change every frame anyway. For example, in a racing game, you can practically rely that the player's car is always moving, so the effects of a missed packet will soon be fixed by the next one.

In general, you should use Unreliable sending where quick, frequent updates are more important than missed packets. Conversely, when data doesn't change so frequently, you can use reliable delta compression to save bandwidth.

## Prediction

When the server has full authority over the world state, the clients only change the game state according to updates they receive from the server. One problem with this is that the delay introduced by waiting for the server to respond can affect gameplay. For example, when a player presses a key to move forward, he won't actually move until the updated state is received from the server. This delay depends on the latency of the connection so the worse the connection the less snappy the control system becomes.

One possible solution to this is **Client-side Prediction** which means the client predicts the expected movement update from the server by using approximately the same internal model. So the player responds immediately to input but the server sees its position from the last update. When the state update finally arrives from the server, the client will compare what he predicted with what actually happened. This might differ because the server might know more about the environment around the player, the client just knows what he needs to know. Errors in prediction are corrected as they happen and if they are corrected continuously then the result will smoother and less noticeable.

### Dead reckoning or interpolation/extrapolation

It is possible to apply the basic principle of client-side prediction to the opponents of the player. **Extrapolation** is the process of storing the last few known values of position, velocity and direction for an opponent and use these to predict where he should be in the immediate future. When the next state update finally arrives with the correct position, the client state will be updated with the exact information, which may make the character jump suddenly if the prediction was bad. In FPS games the behavior of players can be very erratic so this kind of prediction has limited success. If the lag gets high enough the opponent will jump severely as the prediction errors accumulate.

**Interpolation** can be used when packets get dropped on the way to the client. This would normally cause the NPC's movement to pause and then jump to the newest position when the new packet finally arrives. By delaying the world state by some set amount of time (like 100 ms) and then interpolating between the last known position and the new one, the movement between these two points, where packets were dropped, will be smooth.

Page last updated: 2011-11-18

---

## net-NetworkInstantiate

The Network.Instantiate function offers a straightforward way to instantiate a prefab on all clients, similar to the effect of Object.Instantiate on a single client. The instantiating client is the one that controls the object (ie, the Input class is only accessible from scripts running on the client instance) but changes will be reflected across the network.

The argument list for **Network.Instantiate()** is as follows:

```
static function Instantiate (prefab : Object, position : Vector3, rotation : Quaternion, group : int) : Object
```

As with Object.Instantiate, the first three parameters describe the prefab to be instantiated along with its desired position and rotation. The *group* parameter allows you to define subgroups of objects to control the filtering of messages and can be set to zero if filtering is not required (see the *Communication Groups* section below).

### Technical Details

Behind the scenes, network instantiation is built around an RPC call which contains an identifier for the prefab along with the position and other details. The RPC call is always buffered in the same manner as other RPC calls, so that instantiated objects will appear on new clients when they connect. See the RPC page for further details about buffering.

### Communication Groups

Communication groups can be used to select the clients that will receive particular messages. For example, two connected players might be in separate areas of the game world and may never be able to meet. There is thus no reason to transfer game state between the two player clients but you may still want to allow chat communication between them. In this case, instantiation would need to be restricted for gameplay objects but not for the objects that implement the chat feature and so they would be put in separate groups.

Page last updated: 2011-11-22

---

## net-NetworkLevelLoad

Below is a simple example of a way to load a level in a multiplayer game. It makes sure no network messages are being processed while the level is being loaded. It also makes sure no messages are sent, until everything is ready. Lastly, when the

level is loaded it sends a message to all scripts so that they know the level is loaded and can react to that. The **SetLevelPrefix** function helps with keeping unwanted networks updates out of a new loaded level. Unwanted updates might be updates from the previous level for example. The example also uses groups to separate game data and level load communication into groups. Group 0 is used for game data traffic and group 1 for level loading. Group 0 is blocked while the level is being loaded but group 1 kept open, it could also ferry chat communication so that can stay open during level loading.

```
var supportedNetworkLevels : String[] = [ "mylevel" ];
var disconnectedLevel : String = "loader";
private var lastLevelPrefix = 0;

function Awake ()
{
   // Network level loading is done in a separate channel.
   DontDestroyOnLoad(this);
   networkView.group = 1;
   Application.LoadLevel(disconnectedLevel);
}

function OnGUI ()
{
      if (Network.peerType != NetworkPeerType.Disconnected)
      {
            GUILayout.BeginArea(Rect(0, Screen.height - 30, Screen.width, 30));
            GUILayout.BeginHorizontal();

            for (var level in supportedNetworkLevels)
            {
                  if (GUILayout.Button(level))
                  {
                        Network.RemoveRPCsInGroup(0);
                        Network.RemoveRPCsInGroup(1);
                        networkView.RPC( "LoadLevel", RPCMode.AllBuffered, level, lastLevelPrefix + 1);
                  }
            }
            GUILayout.FlexibleSpace();
            GUILayout.EndHorizontal();
            GUILayout.EndArea();
      }
}

@RPC
function LoadLevel (level : String, levelPrefix : int)
{
      lastLevelPrefix = levelPrefix;

            // There is no reason to send any more data over the network on the default channel,
            // because we are about to load the level, thus all those objects will get deleted anyway
            Network.SetSendingEnabled(0, false);

            // We need to stop receiving because first the level must be loaded first.
            // Once the level is loaded, rpc's and other state update attached to objects in the level are allowed to fire
            Network.isMessageQueueRunning = false;

            // All network views loaded from a level will get a prefix into their NetworkViewID.
            // This will prevent old updates from clients leaking into a newly created scene.
            Network.SetLevelPrefix(levelPrefix);
            Application.LoadLevel(level);
            yield;
            yield;
```

```
            // Allow receiving data again
            Network.isMessageQueueRunning = true;
            // Now the level has been loaded and we can start sending out data to clients
            Network.SetSendingEnabled(0, true);


            for (var go in FindObjectsOfType(GameObject))
                go.SendMessage("OnNetworkLoadedLevel", SendMessageOptions.DontRequireReceiver);
}

function OnDisconnectedFromServer ()
{
    Application.LoadLevel(disconnectedLevel);
}

@script RequireComponent(NetworkView)
```

Page last updated: 2009-07-24

---

## net-MasterServer

The Master Server is a meeting place that puts game instances in touch with the player clients who want to connect to them. It can also hide port numbers and IP addresses and perform other technical tasks that arise when setting up network connections, such as firewall handling and NAT punchthrough.

Each individual running game instance provides a **Game Type** to the Master Server. When a player connects and queries the Master Server for their matching **Game Type**, the server responds with the list of running games along with the number of players in each and whether or not a password is needed to play. The two functions used to exchange this data are MasterServer.RegisterHost() for the Server, and MasterServer.RequestHostList() for the player client.

When calling **RegisterHost**, you need to pass three arguments - *gameTypeName* (which is the previously mentioned **Game Type**), *gameName* and *comment* - for the host being registered. **RequestHostList** takes as an argument the *gameTypeName* of the hosts you are interested in connecting to. All the registered hosts of that type will then be returned to the requesting client. This is an asynchronous operation and the complete list can be retrieved with **PollHostList()** after it has arrived in full.

The NAT punchthrough duty of the Master Server is actually handled by a separate process called the **Facilitator** but Unity's Master Server runs both services in parallel.

The **Game Type** is an identifying name that should be unique for each game (although Unity does not offer any central registration system to guarantee this). It makes sense to choose a distinctive name that is unlikely to be used by anyone else. If there are several different versions of you game then you may need to warn a user that their client is not compatible with the running server version. The version information can be passed in the comment field (this is actually binary data and so the version can be passed in any desired form). The game name is simply the name of the particular game instance as supplied by whoever set it up.

The comment field can be used in more advanced ways if the Master Server is suitably modified (see below for further information on how to do this). For example, you could reserve the first ten bytes of the comment field for a password and then extract the password in the Master Server when it receives the host update. It can then reject the host update if a password check fails.

### Registering a game

Before registering a game, it is important to enable or disable the NAT functionality depending on whether or not it is supported by the host; you can do this with the **useNat** parameter of Network.InitializeServer.

A server might be started with code similar to this:-

```
function OnGUI() {
```

```
        if (GUILayout.Button ("Start Server"))
        {
            // Use NAT punchthrough if no public IP present
            Network.InitializeServer(32, 25002, !Network.HavePublicAddress());
            MasterServer.RegisterHost("MyUniqueGameType", "JohnDoes game", "l33t game for all");
        }
}
```

Here we just decide if NAT punchthrough is needed by checking whether or not the machine has a public address. There is a more sophisticated function available called Network.TestConnection which can tell you if the host machine can do NAT or not. It also does connectivity testing for public IP addresses to see if a firewall is blocking the game port. Machines which have public IP addresses always pass the NAT test but if the test fails then the host will **not** be able to connect to NAT clients. In such a case, the user should be informed that port forwarding must be enabled for the game to work. Domestic broadband connections will usually have a NAT address but will not be able to set up port forwarding (since they don't have a personal public IP address). In these cases, if the NAT test fails, the user should be informed that running a server is inadvisable given that only clients on the same local network will be able to connect.

If a host enables NAT functionality without needing it then it will still be accessible. However, clients which cannot do NAT punchthrough might incorrectly think they cannot connect on the basis that the server has NAT enabled.

## Connecting to a game

A **HostData** object is sent during host registrations or queries. It contains the following information about the host:-

| | | |
|---|---|---|
| boolean | **useNat** | Indicates if the host uses NAT punchthrough. |
| String | **gameType** | The game type of the host. |
| String | **gameName** | The game name of the host. |
| int | **connectedPlayers** | The number of currently connected players/clients. |
| int | **playerLimit** | The maximum number of concurrent players/clients allowed. |
| String[] | **IP** | The internal IP address of the host. On a server with a public address the external and internal addresses are the same. This field is defined as an array since all the IP addresses associated with all the active interfaces of the machine need to be checked when connecting internally. |
| int | **port** | The port of the host. |
| boolean | **passwordProtected** | Indicates whether you need to supply a password to be able to connect to this host. |
| String | **comment** | Any comment which was set during host registration. |
| String | **guid** | The network GUID of the host. This is needed to connect using NAT punchthrough. |

This information can be used by clients to see the connection capabilities of the hosts. When NAT is enabled you need to use the GUID of the host when connecting. This is automatically handled for you when the **HostData** is retrieved as you connect. The connection routine might look something like this:

```
function Awake() {
        MasterServer.RequestHostList("MadBubbleSmashGame");
}

function OnGUI() {
        var data : HostData[] = MasterServer.PollHostList();
        // Go through all the hosts in the host list
        for (var element in data)
        {
            GUILayout.BeginHorizontal();
            var name = element.gameName + " " + element.connectedPlayers + " / " + element.playerLimit;
            GUILayout.Label(name);
            GUILayout.Space(5);
            var hostInfo;
            hostInfo = "[";
            for (var host in element.ip)
                hostInfo = hostInfo + host + ":" + element.port + " ";
            hostInfo = hostInfo + "]";
            GUILayout.Label(hostInfo);
            GUILayout.Space(5);
```

```
            GUILayout.Label(element.comment);
            GUILayout.Space(5);
            GUILayout.FlexibleSpace();
            if (GUILayout.Button("Connect"))
            {
                // Connect to HostData struct, internally the correct method is used (GUID when using NAT).
                Network.Connect(element);
            }
            GUILayout.EndHorizontal();
        }
    }
```

This example prints out all of the relevant host information returned by the Master Server. Other useful data like ping information or geographic location of hosts can be added to this.

### NAT punchthrough

The availability of NAT punchthrough may determine whether or not a particular computer is suitable to use as a server. While some clients might be able to connect, there are others that might have trouble connecting to any NAT server.

By default, NAT punchthrough is provided with the help of the Master Server but it need not be done this way. The Facilitator is the process that is actually used for the NAT punchthrough service. If two machines are connected to the Facilitator then it will appear as if they can both connect to each other as long as it uses the external IP and port. The Master Server is used to provide this external IP and port information which is otherwise hard to determine. That is why the Master Server and Facilitator are so tightly integrated. The Master Server and Facilitator have the same IP address by default, to change either one use MasterServer.ipAddress, MasterServer.port, Network.natFacilitatorIP and Network.natFacilitatorPort

### Advanced

Unity Technologies also has a fully deployed Master Server available for testing purposes and this is actually the server that will be used by default. However, the source code is freely available for anyone to use and the server can be deployed on Windows, Linux and Mac OS. In addition to simply building the project from source, there might be cases where you want to modify the way in which the Master Server handles information and how it communicates. For example, you may be able to optimize the handling of host data or limit the number of clients returned on the host list. Such changes will require modifications to the source code; information about how to go about this can be found on the Master Server Build page.

Page last updated: 2011-11-22

---

## net-MasterServerBuild

The source code for all the individual networking servers can be downloaded from the Unity website. This includes the connection tester, facilitator, master server and proxy server.

All source packages include the RakNet 3.732 networking library which handles the basic networking functions and provides plugins used by the networking servers.

The packages include three different types of project files, ready for compilation:

- An Xcode 3.0 project for Mac OS X
- A Makefile for Linux and Mac OS X
- A Visual Studio 2008 solution

The Xcode and Visual Studio projects can just be opened, compiled and built. To build with the Makefile just run "make". It should work with a standard compilation setup on Linux and Mac OS X, if you have *gcc* then it should work. On Linux you might need to install the ncurses library.

### Structure

#### The Master Server

The Master Server uses an internal *database* structure to keep track of host information.

Hosts send messages with the RUM_UPDATE_OR_ADD_ROW message identifier and all their host information embedded. This is processed in the *OnReceive()* function in the LightweightDatabaseServer.cpp file. This is where all message initially appear and therefore a good place to start if you want to trace how a message is processed. A table is created within the database structure for each *game type* which is set when you use MasterServer.RegisterHost function. All game types are grouped together in a table, if the table does not exist it is dynamically created in the *CreateDefaultTable()* function.

The host information data is modified by the master server. The IP and port of the game which is registering, as seen by the master server, is injected into the host data. This way we can for sure detect the correct external IP and port in cases where the host has a private address (NAT address). The IP and port in the host data sent by the game server is the private address and port and this is stored for later use. If the master server detects that a client is requesting the host data for a game server and the server has the *same* IP address then he uses the private address of the server instead of the external one. This is to handle cases where the client and server are on the same local network, using the same router with NAT addresses. Thus they will have the same external address and cannot connect to each other through it, they need to use the private addresses and those will work in this case.

Clients send messages with the ID_DATABASE_QUERY_REQUEST message identifier and what game type they are looking for. The table or host list is fetched from the database structure and sent to the client. If it isn't found and empty host list is sent.

All messages sent to the master server must contain version information which is checked in the *CheckVersion()* function. At the moment each version of Unity will set a new master server version internally and this is checked here. So if the master server communication routine will change at any point it will be able to detect older versions here and possibly refer to another version of the master server (if at all needed) or modify the processing of that message to account for differences.

**The Facilitator**
The facilitator uses the NAT punchthrough plugin from RakNet directly with no modifications. This is essentially just a peer listening on a port with the NAT punchthrough plugin loaded. When a server and a client with NAT addresses are both connected to this peer, they will be able to perform NAT punchthrough to connect to each other. When the Network.InitializeServer uses NAT, the connection is set up automatically for you.

Page last updated: 2011-02-04

---

# net-MinimizingBandwidth

Since network communication is potentially slow compared to other aspects of a game, it is important to reduce it to a minimum. It is therefore very important to consider how much data you are exchanging and how frequently the exchanges take place.

## How data is synchronized

The amount of network bandwidth used depends heavily on whether you use the **Unreliable** or the **Reliable Delta Compression** mode to synchronize data (the mode is set from the Network View component).

In **Unreliable** mode, the whole of the object being synchronized will be transmitted on each iteration of the network update loop. The frequency of this update is determined by the value of Network.sendRate, which is set to 15 updates per second by default. **Unreliable** mode ensures frequent updates but any dropped or delayed packets will simply be ignored. This is often the best synchronization mode to use when objects change state very frequently and the effect of a missed update is very short-lived. However, you should bear in mind the amount of data that might be sent during each update. For example, synchronizing a Transform involves transmitting nine float values (three Vector3s with three floats each), which equates to 36 Bytes per update. If the server is running with eight clients and using the default update frequency then it will receive 4,320 KBytes/s (8*36*15) or 34.6Kbits/s and transmit 30.2 KBytes/s (8*7*36*15) or 242Kbits/s. You can reduce the bandwidth consumption considerably by lowering the frequency of updates, but the default value of 15 is about right for a game where the action moves quickly.

In **Reliable Delta Compressed** mode, the data is guaranteed to be sent reliably and arrive in the right order. If packets are lost then they get retransmitted and if they arrive out of order, they will be buffered until all packets in the sequence have arrived. Although this ensures that transmitted data is received correctly, the waiting and retransmission tend to increase bandwidth usage. However, the data is also delta compressed which means only the differences between the last state and the current state are transmitted. If the state is exactly the same then nothing is sent. The benefit of delta compression thus depends on how much the state changes and in which properties.

### What data is synchronized

There is plenty of opportunity for creativity in designing the game so that the state *appears* to be the same on all clients even though it may not be, strictly. An example of this is where animations are synchronized. If an Animation component is observed by a Network View then its properties will be synchronized exactly, so the frames of animation will appear exactly the same on all clients. Although this may be desirable in some cases, typically it will be enough for the character to be seen as walking, running, jumping, etc. The animations can thus be crudely synchronized simply by sending an integer value to denote which animation sequence to play. This will save a great deal of bandwidth compared to synchronizing the whole Animation component.

### When to synchronize data

It is often unnecessary to keep the game perfectly in sync on all clients, for example, in cases where the players are temporarily in different areas of the game world where they won't encounter each other. This can reduce the bandwidth as well as the load on the server since only the clients that can interact need to be kept in sync. This concept is sometimes referred to as **Relevant Sets** (ie, there is a subset of the total game that is relevant to any particular client at any particular time). Synchronizing clients according to their relevant sets can be handled with RPCs, since they can give greater control over the destination of a state update.

#### Level loading

When loading levels, it is seldom necessary to worry about the bandwidth being used since each client can simply wait until all the others have initialized the level to be played. Level loading can often involve transmitting even quite large data items (such as images or audio data).

Page last updated: 2011-11-21

---

## net-SocialAPI

Social API is Unity's point of access to social features, such as:

- User profiles
- Friends lists
- Achievements
- Statistics / Leaderboards

It provides a unified interface to different social back-ends, such as **XBox Live** or **GameCenter**, and is meant to be used primarily by programmers on the game project.

The Social API is mainly an asynchronous API, and the typical way to use it is by making a function call and registering for a callback to when that function completes. The asynchronous function may have side effects, such as populating certain state variables in the API, and the callback could contain data from the server to be processed.

The Social class resides in the UnityEngine namespace and so is always available but the other Social API classes are kept in their own namespace, UnityEngine.SocialPlatforms. Furthermore, implementations of the Social API are in a sub-namespace, like SocialPlatforms.GameCenter.

Here is an example (JavaScript) of how one might use the Social API:

```
import UnityEngine.SocialPlatforms;

function Start () {
    // Authenticate and register a ProcessAuthentication callback
    // This call needs to be made before we can proceed to other calls in the Social API
    Social.localUser.Authenticate (ProcessAuthentication);
}

// This function gets called when Authenticate completes
// Note that if the operation is successful, Social.localUser will contain data from the server.
function ProcessAuthentication (success: boolean) {
```

```
    if (success) {
        Debug.Log ("Authenticated, checking achievements");

        // Request loaded achievements, and register a callback for processing them
        Social.LoadAchievements (ProcessLoadedAchievements);
    }
    else
        Debug.Log ("Failed to authenticate");
}

// This function gets called when the LoadAchievement call completes
function ProcessLoadedAchievements (achievements: IAchievement[]) {
    if (achievements.Length == 0)
        Debug.Log ("Error: no achievements found");
    else
        Debug.Log ("Got " + achievements.Length + " achievements");

    // You can also call into the functions like this
    Social.ReportProgress ("Achievement01", 100.0, function(result) {
        if (result)
            Debug.Log ("Successfully reported achievement progress");
        else
            Debug.Log ("Failed to report achievement");
    });
}
```

Here is the same example using C#.

```
using UnityEngine;
using UnityEngine.SocialPlatforms;

public class SocialExample : MonoBehaviour {

    void Start () {
        // Authenticate and register a ProcessAuthentication callback
        // This call needs to be made before we can proceed to other calls in the Social API
        Social.localUser.Authenticate (ProcessAuthentication);
    }

    // This function gets called when Authenticate completes
    // Note that if the operation is successful, Social.localUser will contain data from the server.
    void ProcessAuthentication (bool success) {
        if (success) {
            Debug.Log ("Authenticated, checking achievements");

            // Request loaded achievements, and register a callback for processing them
            Social.LoadAchievements (ProcessLoadedAchievements);
        }
        else
            Debug.Log ("Failed to authenticate");
    }

    // This function gets called when the LoadAchievement call completes
    void ProcessLoadedAchievements (IAchievement[] achievements) {
        if (achievements.Length == 0)
            Debug.Log ("Error: no achievements found");
        else
            Debug.Log ("Got " + achievements.Length + " achievements");

        // You can also call into the functions like this
```

```
        Social.ReportProgress ("Achievement01", 100.0, result => {
            if (result)
                Debug.Log ("Successfully reported achievement progress");
            else
                Debug.Log ("Failed to report achievement");
        });
    }
}
```

For more info on the Social API, check out the Social API Scripting Reference

Page last updated: 2012-01-25

---

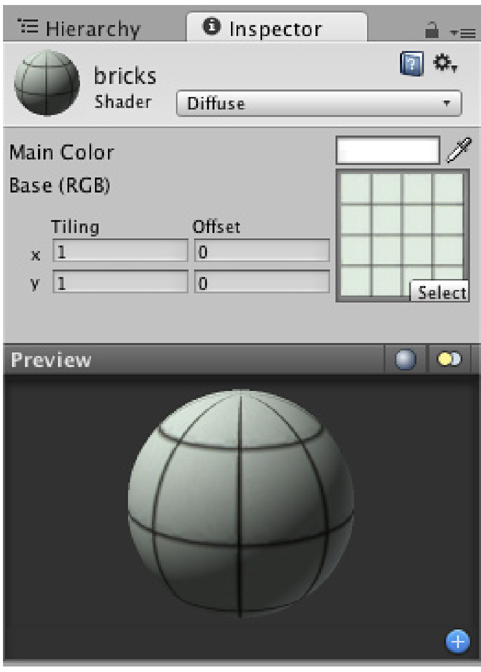
# Built-in Shader Guide

If you're looking for the best information for using Unity's built-in shaders, you've come to the right place. Unity includes more than 40 built-in shaders, and of course you can write many more on your own! This guide will explain each family of the built-in shaders, and go into detail for each specific shader. With this guide, you'll be able to make the most out of Unity's shaders, to achieve the effect you're aiming for.

### Using Shaders

Shaders in Unity are used through **Materials**, which essentially combine shader code with parameters like textures. An in-depth explanation of the Shader/Material relationship can be read here.

Material properties will appear in the **Inspector** when either the Material itself or a **GameObject** that uses the Material is selected. The Material Inspector looks like this:




Each Material will look a little different in the Inspector, depending on the specific shader it is using. The shader iself determines what kind of properties will be available to adjust in the Inspector. Material inspector is described in detail in Material reference page. Remember that a shader is implemented through a Material. So while the shader defines the properties that will be shown in the Inspector, each Material actually contains the adjusted data from sliders, colors, and textures. The most important thing to remember about this is that a single shader can be used in multiple Materials, but a single Material cannot use multiple shaders.

### Built-in Unity Shaders

- Performance of Unity shaders

- Normal Shader Family
  - Vertex-Lit
  - Diffuse
  - Specular
  - Bumped Diffuse
  - Bumped Specular
  - Parallax Diffuse
  - Parallax Bumped Specular
  - Decal
  - Diffuse Detail
- Transparent Shader Family
  - Transparent Vertex-Lit
  - Transparent Diffuse
  - Transparent Specular
  - Transparent Bumped Diffuse
  - Transparent Bumped Specular
  - Transparent Parallax Diffuse
  - Transparent Parallax Specular
- Transparent Cutout Shader Family
  - Transparent Cutout Vertex-Lit
  - Transparent Cutout Diffuse
  - Transparent Cutout Specular
  - Transparent Cutout Bumped Diffuse
  - Transparent Cutout Bumped Specular
- Self-Illuminated Shader Family
  - Self-Illuminated Vertex-Lit
  - Self-Illuminated Diffuse
  - Self-Illuminated Specular
  - Self-Illuminated Normal mapped Diffuse
  - Self-Illuminated Normal mapped Specular
  - Self-Illuminated Parallax Diffuse
  - Self-Illuminated Parallax Specular
- Reflective Shader Family
  - Reflective Vertex-Lit
  - Reflective Diffuse
  - Reflective Specular
  - Reflective Bumped Diffuse
  - Reflective Bumped Specular
  - Reflective Parallax Diffuse
  - Reflective Parallax Specular
  - Reflective Normal Mapped Unlit
  - Reflective Normal mapped Vertex-lit

Page last updated: 2011-01-20

---

# shader-Performance

There are a number of factors that can affect the overall performance of your game. This page will talk specifically about the performance considerations for Built-in Shaders. Performance of a shader mostly depends on two things: shader itself and which Rendering Path is used by the project or specific camera. For performance tips when writing your own shaders, see ShaderLab Shader Performance page.

## Rendering Paths and shader performance

From the rendering paths Unity supports, Deferred Lighting and Vertex Lit paths have the most predictable performance. In Deferred lighting, each object is generally drawn twice, no matter what lights are affecting it. Similarly, in Vertex Lit each object is generally drawn once. So then, the performance differences in shaders mostly depend on how many textures they use and what calculations they do.

**Shader Performance in Forward rendering path**

In Forward rendering path, performance of a shader depends on **both** the shader itself and the lights in the scene. The following section explains the details. There are two basic categories of shaders from a performance perspective, **Vertex-Lit**, and **Pixel-Lit**.

**Vertex-Lit** shaders in Forward rendering path are always cheaper than Pixel-Lit shaders. These shaders work by calculating lighting based on the mesh vertices, using all lights at once. Because of this, no matter how many lights are shining on the object, it will only have to be drawn once.

**Pixel-Lit** shaders calculate final lighting at each pixel that is drawn. Because of this, the object has to be drawn once to get the ambient & main directional light, and once for each additional light that is shining on it. Thus the formula is N rendering passes, where N is the final number of pixel lights shining on the object. This increases the load on the CPU to process and send off commands to the graphics card, and on the graphics card to process the vertices and draw the pixels. The size of the Pixel-lit object on the screen will also affect the speed at which it is drawn. The larger the object, the slower it will be drawn.

So pixel lit shaders come at performance cost, but that cost allows for some spectacular effects: shadows, normal-mapping, good looking specular highlights and light cookies, just to name a few.

Remember that lights can be forced into a pixel ("important") or vertex/SH ("not important") mode. Any vertex lights shining on a Pixel-Lit shader will be calculated based on the object's vertices or whole object, and will not add to the rendering cost or visual effects that are associated with pixel lights.

### General shader performance

Out of Built-in Shaders, they come roughly in this order of increasing complexity:

- **Unlit**. This is just a texture, not affected by any lighting.
- **VertexLit**.
- **Diffuse**.
- **Normal mapped**. This is a bit more expensive than Diffuse: it adds one more texture (normal map), and a couple of shader instructions.
- **Specular**. This adds specular highlight calculation.
- **Normal Mapped Specular**. Again, this is a bit more expensive than Specular.
- **Parallax Normal mapped**. This adds parallax normal-mapping calculation.
- **Parallax Normal Mapped Specular**. This adds both parallax normal-mapping and specular highlight calculation.

Additionally, Unity has several simplified shaders targeted at mobile platforms, under "Mobile" category. These shaders work on other platforms as well, so if you can live with their simplifications (e.g. approximate specular, no per-material color support etc.), try using them!

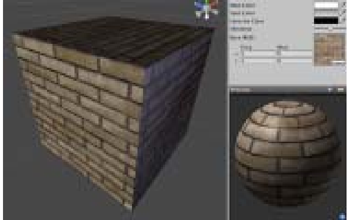
Page last updated: 2011-01-26

---

## shader-NormalFamily

These shaders are the basic shaders in Unity. They are not specialized in any way and should be suitable for most opaque objects. They are not suitable if you want your object to be transparent, emitting light etc.

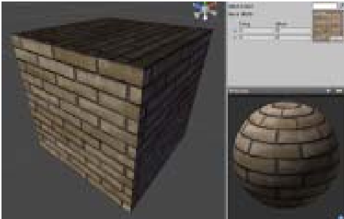
### Vertex Lit



**Assets needed:**

- One **Base** texture, no alpha channel required

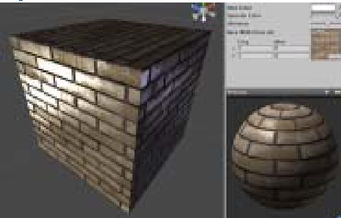
### Diffuse



**Assets needed:**

- One **Base** texture, no alpha channel required

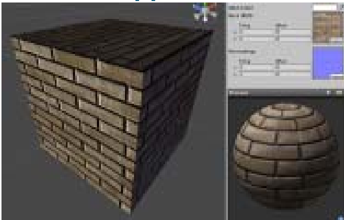
## Specular



**Assets needed:**

- One **Base** texture with alpha channel for Specular Map

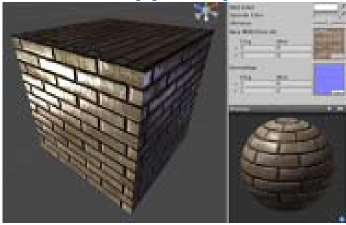
## Normal mapped



**Assets needed:**

- One **Base** texture, no alpha channel required
- One **Normal map**

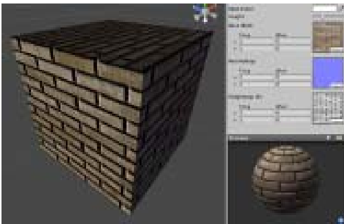
## Normal mapped Specular



**Assets needed:**

- One **Base** texture with alpha channel for Specular Map
- One **Normal map**

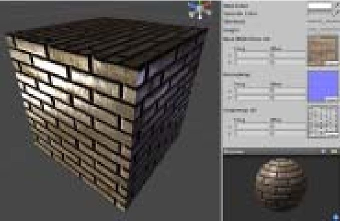
## Parallax



**Assets needed:**

- One **Base** texture, no alpha channel required
- One **Normal map**
- One **Height** texture with Parallax Depth in the alpha channel

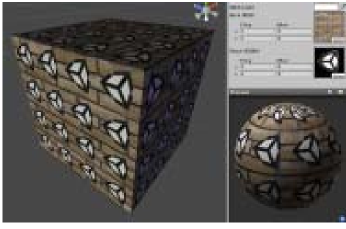
### Parallax Specular



**Assets needed:**

- One **Base** texture with alpha channel for Specular Map
- One **Normal map**
- One **Height** texture with Parallax Depth in the alpha channel

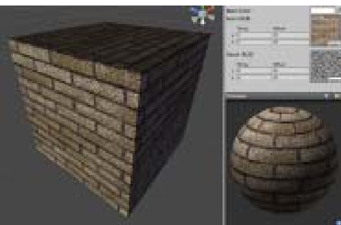
### Decal



**Assets needed:**

- One **Base** texture, no alpha channel required
- One **Decal** texture with alpha channel for Decal transparency

### Diffuse Detail



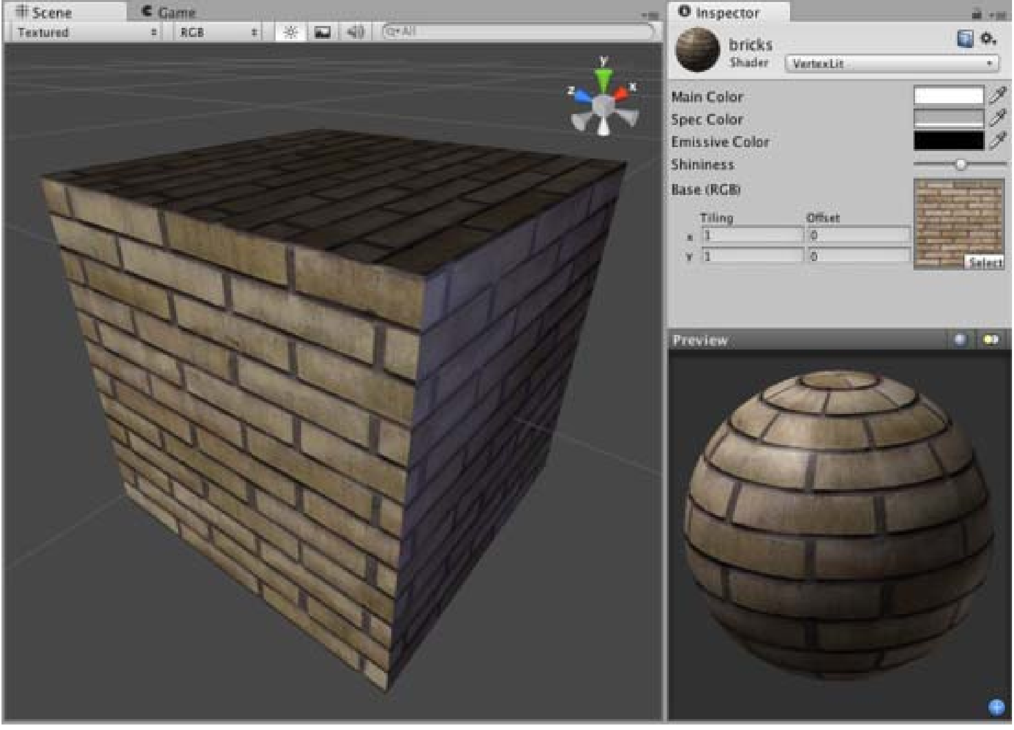
**Assets needed:**

- One **Base** texture, no alpha channel required
- One **Detail** grayscale texture; with 50% gray being neutral color

Page last updated: 2011-01-26

---

## shader-NormalVertexLit




### Vertex-Lit Properties

This shader is **Vertex-Lit**, which is one of the simplest shaders. All lights shining on it are rendered in a single pass and calculated at vertices only.

Because it is vertex-lit, it won't display any pixel-based rendering effects, such as light cookies, normal mapping, or shadows. This shader is also much more sensitive to tesselation of the models. If you put a point light very close to a cube using this shader, the light will only be calculated at the corners. Pixel-lit shaders are much more effective at creating a nice round highlight, independent of tesselation. If that's an effect you want, you may consider using a pixel-lit shader or increase tesselation of the objects instead.

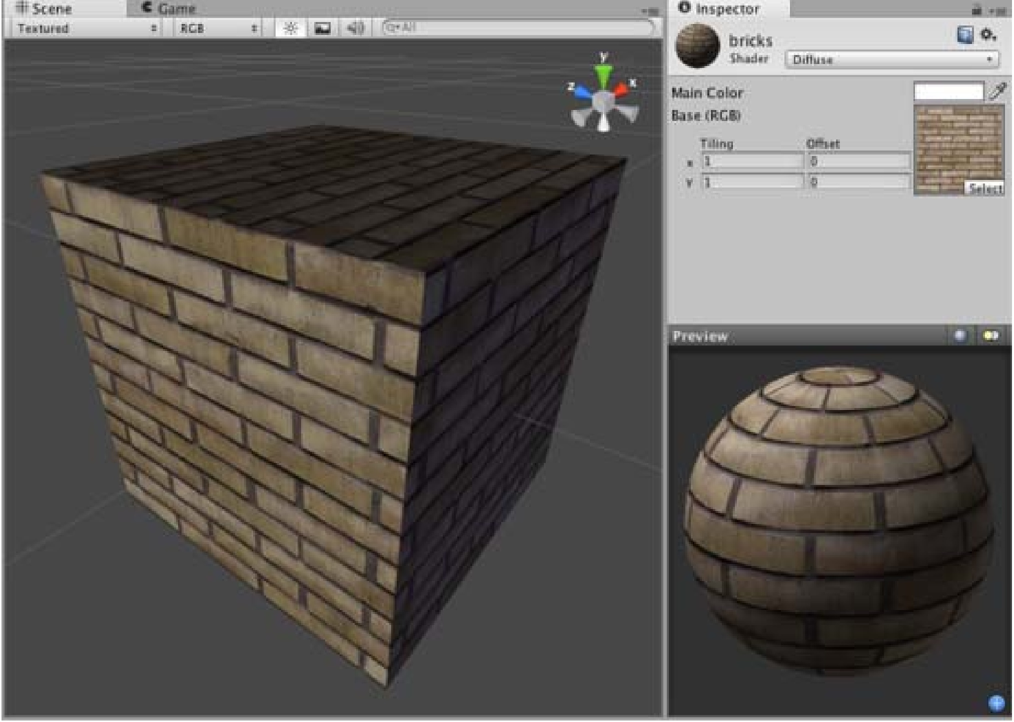
### Performance

Generally, this shader is very cheap to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-NormalDiffuse




### Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

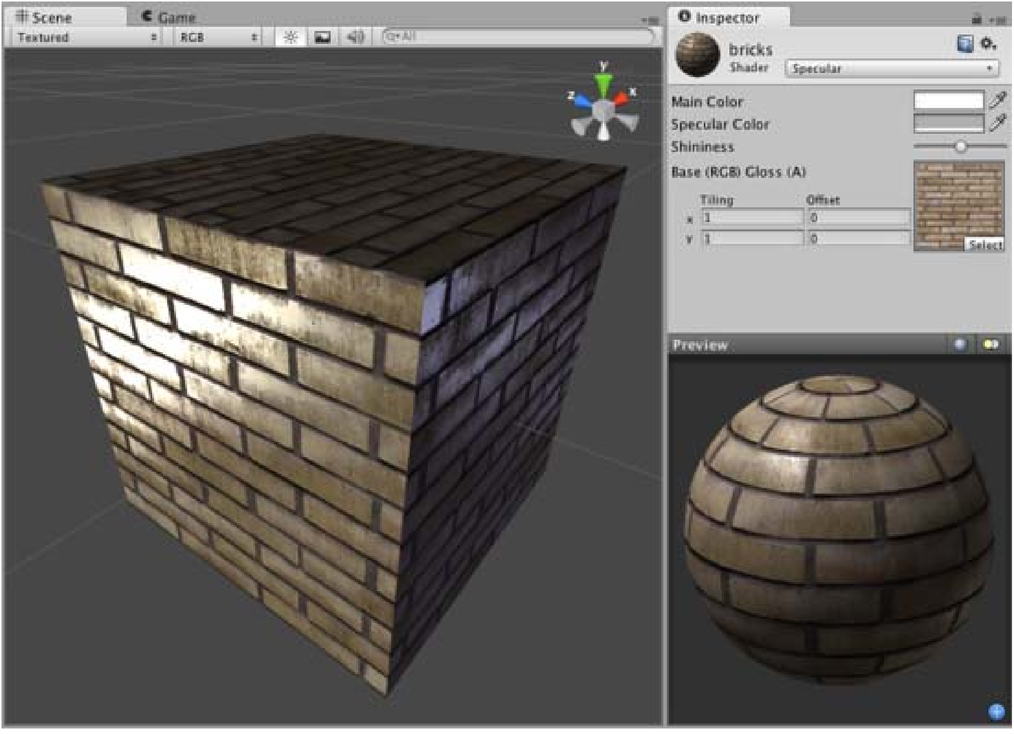
### Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-NormalSpecular



### Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

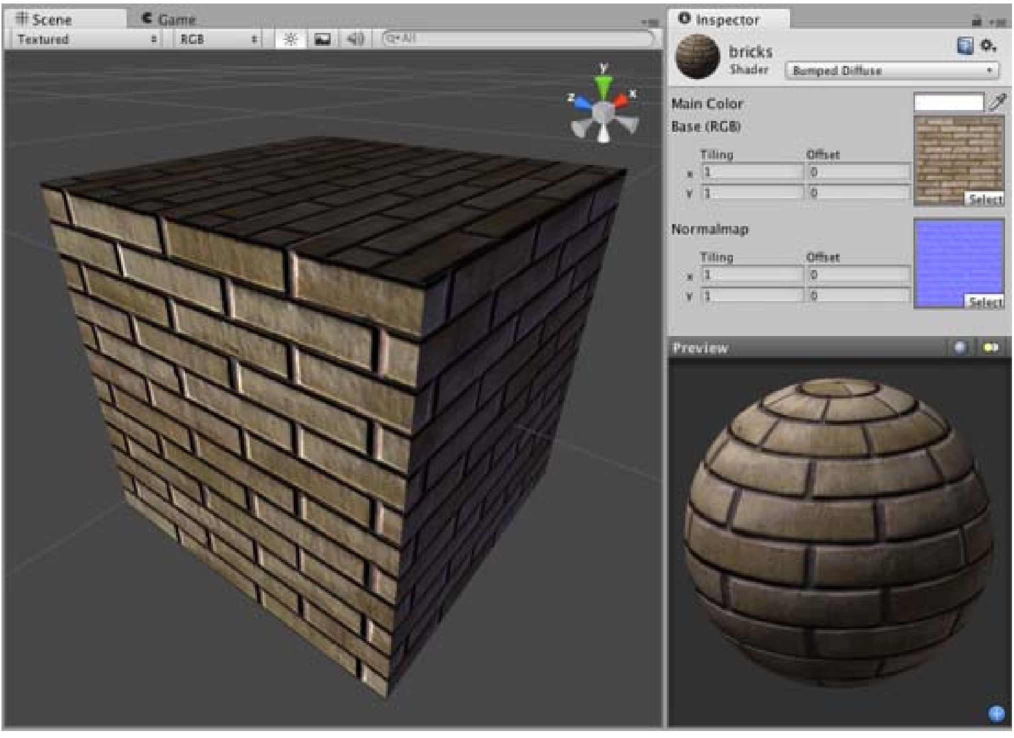
### Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-NormalBumpedDiffuse




### Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

**Normal mapping** simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each pixel's color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the mesh's geometry when calculating lighting of the object.

#### Creating Normal maps

You can import a regular grayscale image and convert it to a Normal Map from within Unity. To learn how to do this, please read the Normal map FAQ page.

#### Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

1. It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
2. It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

### Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

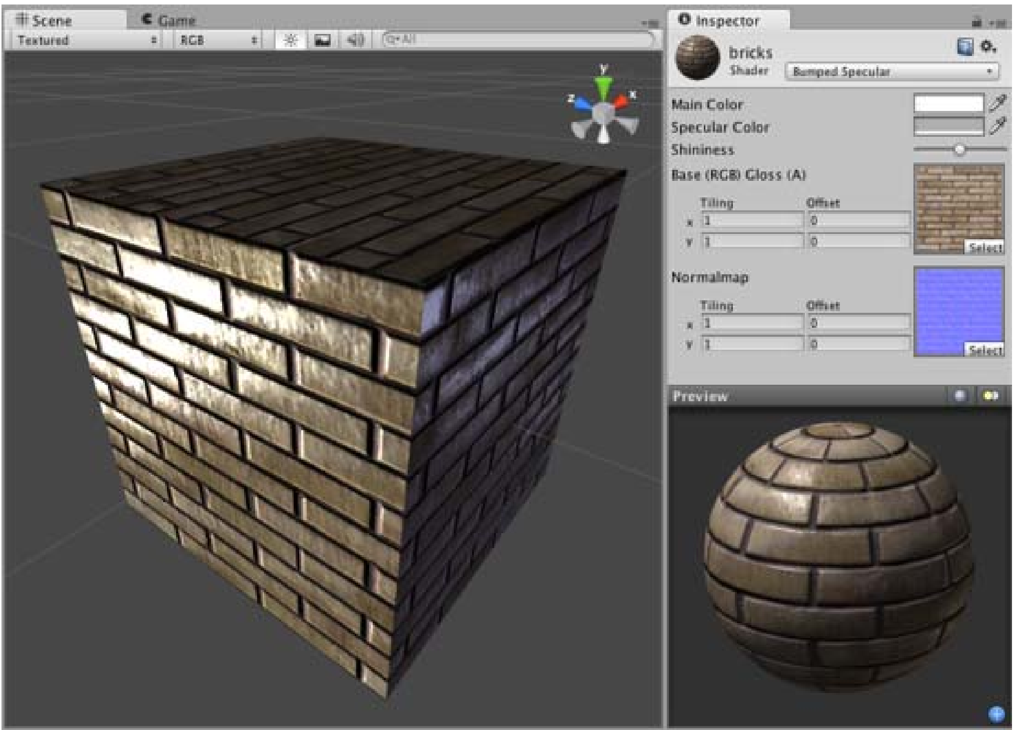
### Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-NormalBumpedSpecular




### Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

**Normal mapping** simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each pixel's color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the mesh's geometry when calculating lighting of the object.

#### Creating Normal maps

You can import a regular grayscale image and convert it to a Normal Map from within Unity. To learn how to do this, please read the Normal map FAQ page.

#### Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

1. It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
2. It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

### Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

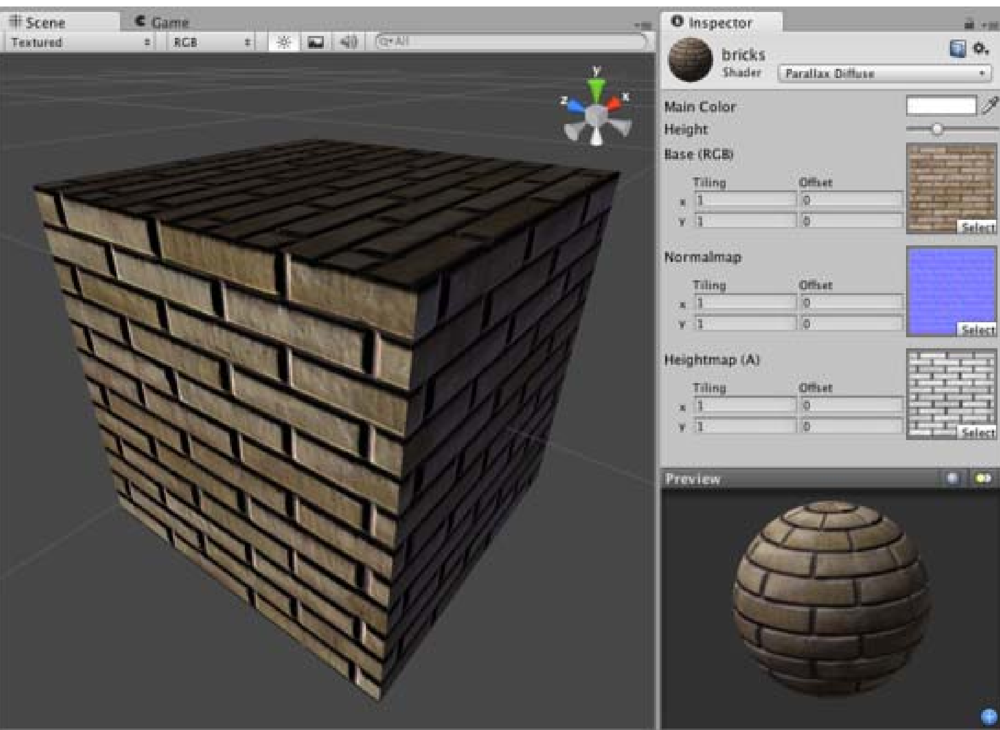
### Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-NormalParallaxDiffuse




### Parallax Normal mapped Properties

**Parallax Normal mapped** is the same as regular **Normal mapped**, but with a better simulation of "depth". The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

### Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

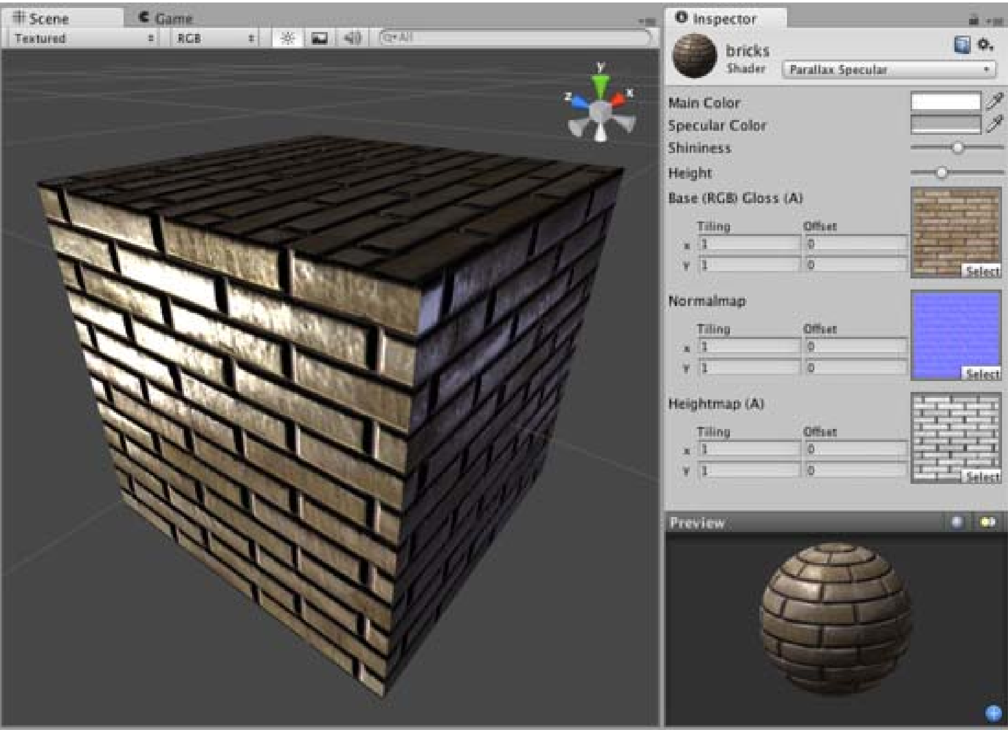
### Performance

Generally, this shader is on the more expensive rendering side. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-NormalParallaxSpecular




### Parallax Normal mapped Properties

**Parallax Normal mapped** is the same as regular **Normal mapped**, but with a better simulation of "depth". The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

### Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which

areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

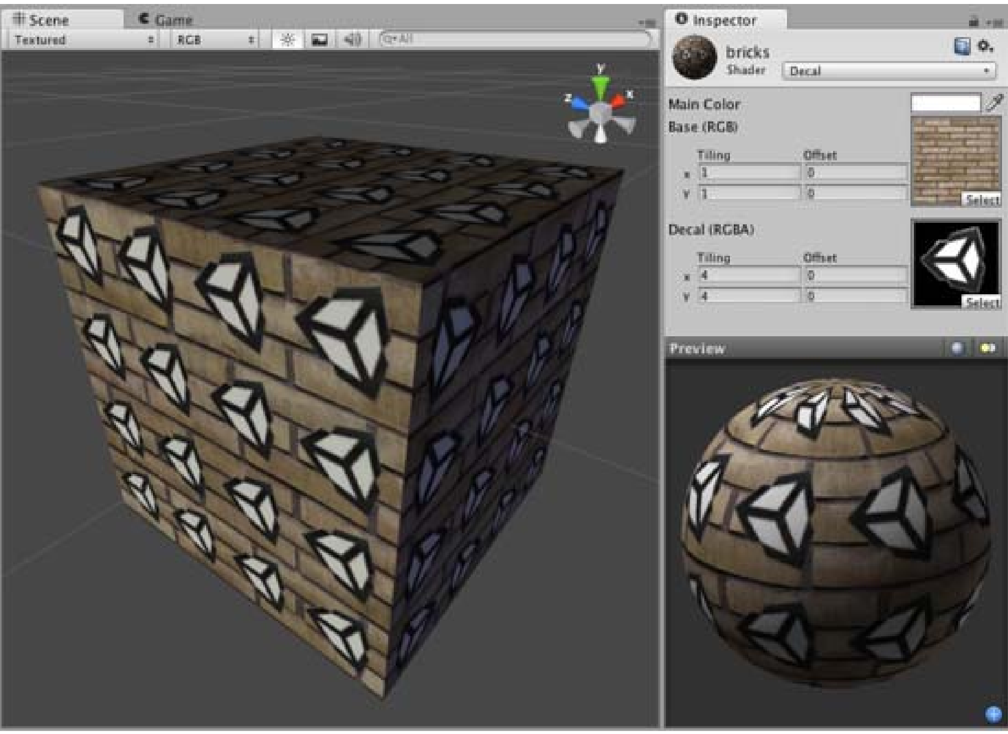
### Performance

Generally, this shader is on the more expensive rendering side. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-NormalDecal




### Decal Properties

This shader is a variation of the VertexLit shader. All lights that shine on it will be rendered as vertex lights by this shader. In addition to the main texture, this shader makes use of a second texture for additional details. The second "Decal" texture uses an alpha channel to determine visible areas of the main texture. The decal texture should be supplemental to the main texture. For example, if you have a brick wall, you can tile the brick texture as the main texture, and use the decal texture with alpha channel to draw graffiti at different places on the wall.

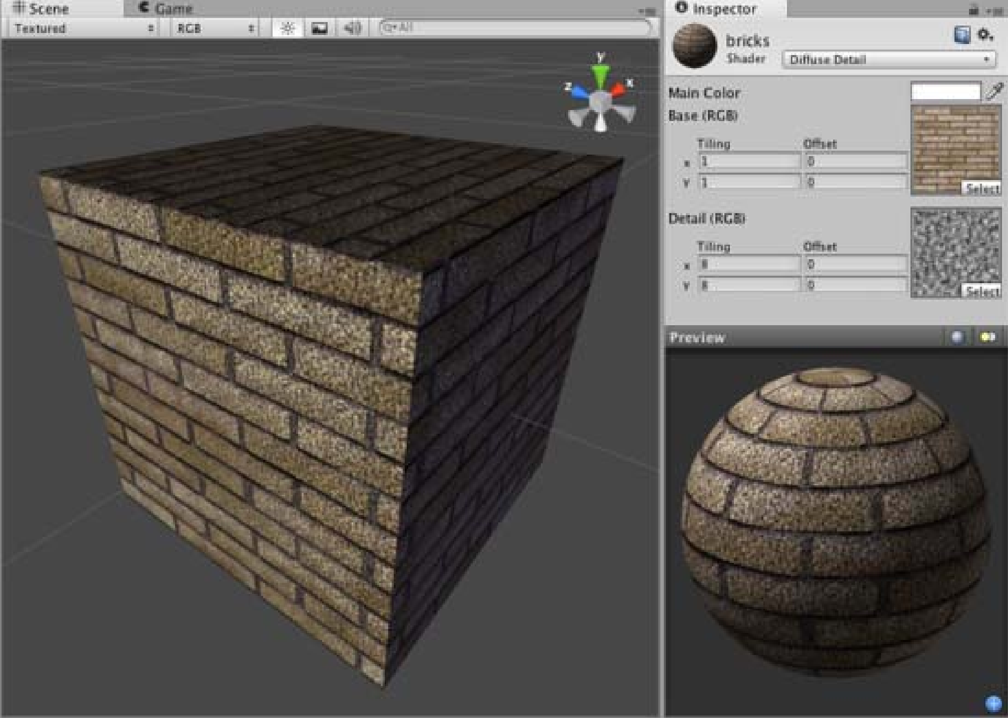
### Performance

This shader is approximately equivalent to the VertexLit shader. It is marginally more expensive due to the second decal texture, but will not have a noticeable impact.

Page last updated: 2007-09-15

---

## shader-NormalDiffuseDetail




### Diffuse Detail Properties

This shader is a version of the regular Diffuse shader with additional data. It allows you to define a second "Detail" texture that will gradually appear as the camera gets closer to it. It can be used on terrain, for example. You can use a base low-resolution texture and stretch it over the entire terrain. When the camera gets close the low-resolution texture will get blurry, and we don't want that. To avoid this effect, create a generic Detail texture that will be tiled over the terrain. This way, when the camera gets close, the additional details appear and the blurry effect is avoided.

The Detail texture is put "on top" of the base texture. Darker colors in the detail texture will darken the main texture and lighter colors will brighten it. Detail texture are usually gray-ish. For more information on effectively creating Detail textures, please view this page.

### Performance

This shader is pixel-lit, and approximately equivalent to the Diffuse shader. It is marginally more expensive due to additional texture.

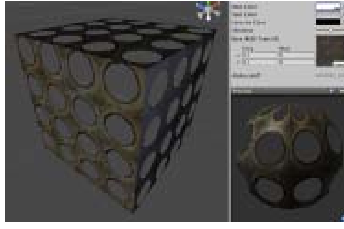
Page last updated: 2007-10-07

---

## shader-TransparentFamily

The Transparent shaders are used for fully- or semi-transparent objects. Using the alpha channel of the **Base** texture, you can determine areas of the object which can be more or less transparent than others. This can create a great effect for glass, HUD interfaces, or sci-fi effects.

### Transparent Vertex-Lit

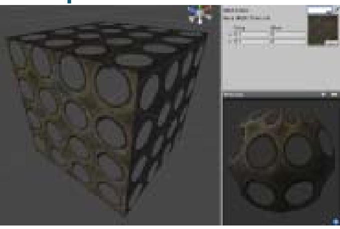


**Assets needed:**

- One **Base** texture with alpha channel for Transparency Map

» More details

## Transparent Diffuse

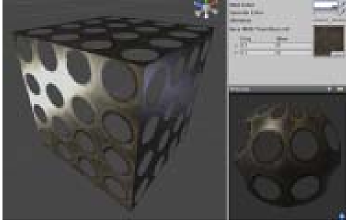


**Assets needed:**

- One **Base** texture with alpha channel for Transparency Map

» More details

## Transparent Specular



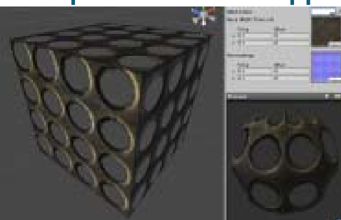
**Assets needed:**

- One **Base** texture with alpha channel for combined Transparency Map/Specular Map

**Note:** One limitation of this shader is that the **Base** texture's alpha channel doubles as a Specular Map for the Specular shaders in this family.

» More details

## Transparent Normal mapped

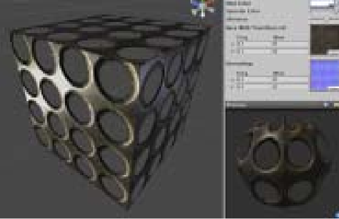


**Assets needed:**

- One **Base** texture with alpha channel for Transparency Map
- One **Normal map** normal map, no alpha channel required

» More details

## Transparent Normal mapped Specular



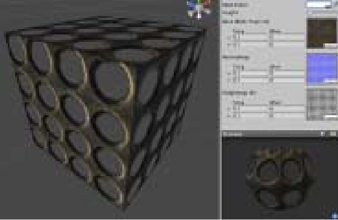
**Assets needed:**

- One **Base** texture with alpha channel for combined Transparency Map/Specular Map
- One **Normal map** normal map, no alpha channel required

**Note:** One limitation of this shader is that the **Base** texture's alpha channel doubles as a Specular Map for the Specular shaders in this family.

» More details

### Transparent Parallax

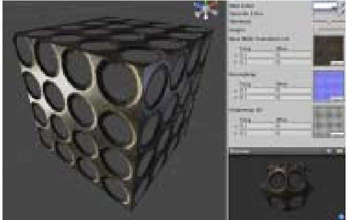


**Assets needed:**

- One **Base** texture with alpha channel for Transparency Map
- One **Normal map** normal map with alpha channel for Parallax Depth

» More details

### Transparent Parallax Specular



**Assets needed:**

- One **Base** texture with alpha channel for combined Transparency Map/Specular Map
- One **Normal map** normal map with alpha channel for Parallax Depth

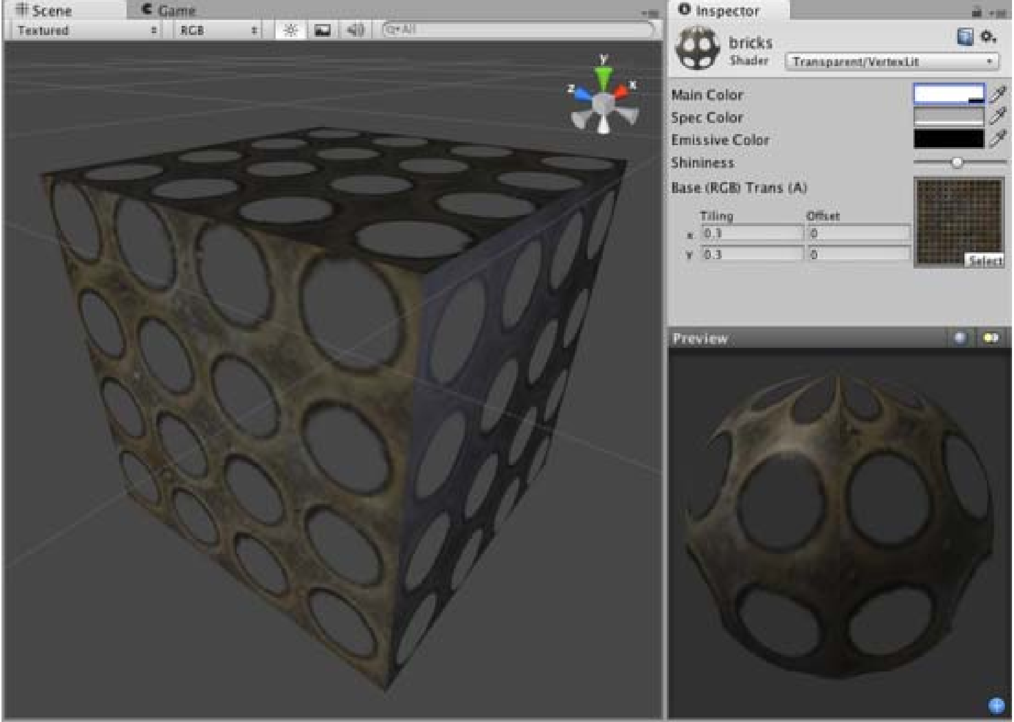
**Note:** One limitation of this shader is that the **Base** texture's alpha channel doubles as a Specular Map for the Specular shaders in this family.

» More details

Page last updated: 2010-07-13

---

## shader-TransVertexLit




### Transparent Properties

This shader can make mesh geometry partially or fully transparent by reading the alpha channel of the main texture. In the alpha, 0 (black) is completely transparent while 255 (white) is completely opaque. If your main texture does not have an alpha channel, the object will appear completely opaque.

Using transparent objects in your game can be tricky, as there are traditional graphical programming problems that can present sorting issues in your game. For example, if you see odd results when looking through two windows at once, you're experiencing the classical problem with using transparency. The general rule is to be aware that there are some cases in which one transparent object may be drawn in front of another in an unusual way, especially if the objects are intersecting, enclose each other or are of very different sizes. For this reason, you should use transparent objects if you need them, and try not to let them become excessive. You should also make your designer(s) aware that such sorting problems can occur, and have them prepare to change some design to work around these issues.

### Vertex-Lit Properties

This shader is **Vertex-Lit**, which is one of the simplest shaders. All lights shining on it are rendered in a single pass and calculated at vertices only.

Because it is vertex-lit, it won't display any pixel-based rendering effects, such as light cookies, normal mapping, or shadows. This shader is also much more sensitive to tesselation of the models. If you put a point light very close to a cube using this shader, the light will only be calculated at the corners. Pixel-lit shaders are much more effective at creating a nice round highlight, independent of tesselation. If that's an effect you want, you may consider using a pixel-lit shader or increase tesselation of the objects instead.

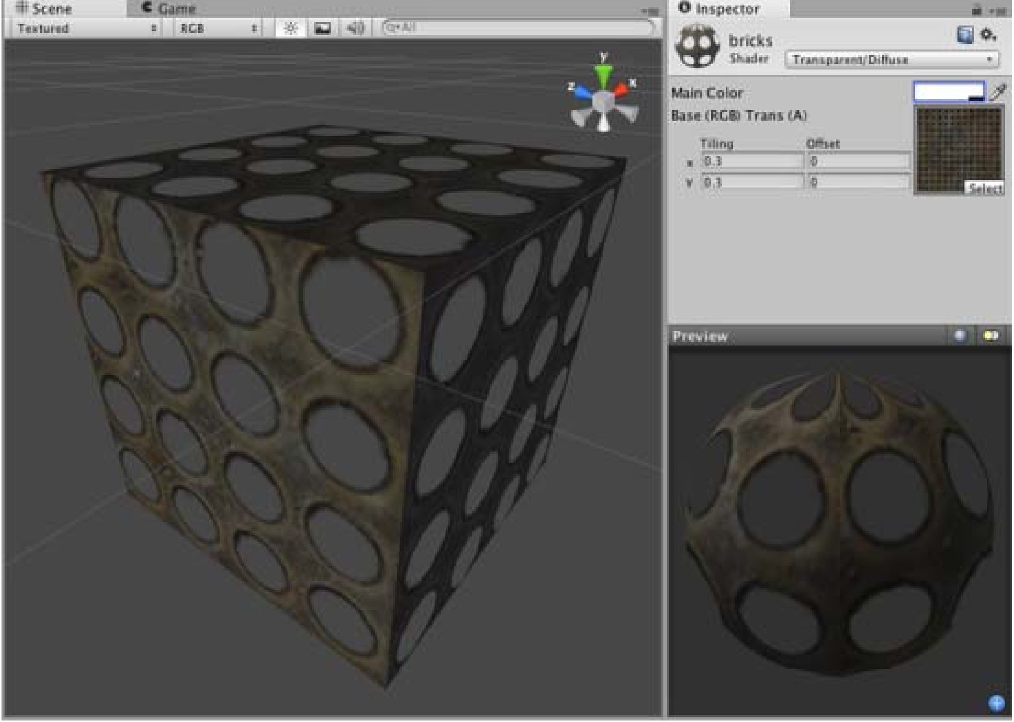
### Performance

Generally, this shader is very cheap to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-TransDiffuse




### Transparent Properties

This shader can make mesh geometry partially or fully transparent by reading the alpha channel of the main texture. In the alpha, 0 (black) is completely transparent while 255 (white) is completely opaque. If your main texture does not have an alpha channel, the object will appear completely opaque.

Using transparent objects in your game can be tricky, as there are traditional graphical programming problems that can present sorting issues in your game. For example, if you see odd results when looking through two windows at once, you're experiencing the classical problem with using transparency. The general rule is to be aware that there are some cases in which one transparent object may be drawn in front of another in an unusual way, especially if the objects are intersecting, enclose each other or are of very different sizes. For this reason, you should use transparent objects if you need them, and try not to let them become excessive. You should also make your designer(s) aware that such sorting problems can occur, and have them prepare to change some design to work around these issues.

### Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

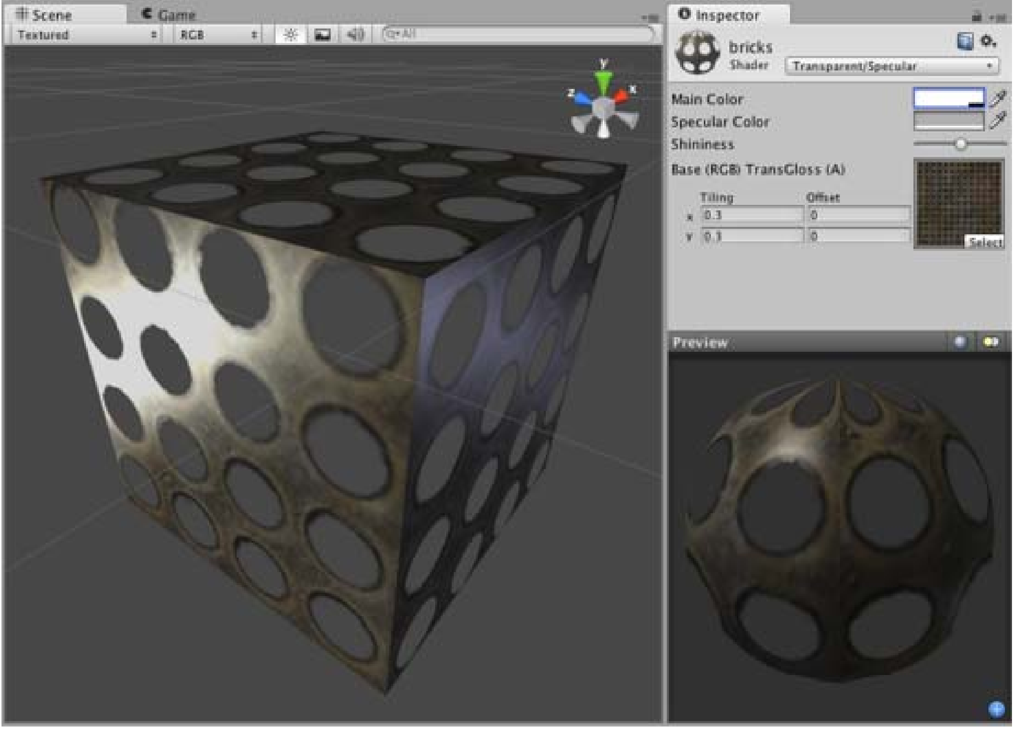
### Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-TransSpecular




One consideration for this shader is that the Base texture's alpha channel defines both the Transparent areas as well as the Specular Map.

### Transparent Properties

This shader can make mesh geometry partially or fully transparent by reading the alpha channel of the main texture. In the alpha, 0 (black) is completely transparent while 255 (white) is completely opaque. If your main texture does not have an alpha channel, the object will appear completely opaque.

Using transparent objects in your game can be tricky, as there are traditional graphical programming problems that can present sorting issues in your game. For example, if you see odd results when looking through two windows at once, you're experiencing the classical problem with using transparency. The general rule is to be aware that there are some cases in which one transparent object may be drawn in front of another in an unusual way, especially if the objects are intersecting, enclose each other or are of very different sizes. For this reason, you should use transparent objects if you need them, and try not to let them become excessive. You should also make your designer(s) aware that such sorting problems can occur, and have them prepare to change some design to work around these issues.

### Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

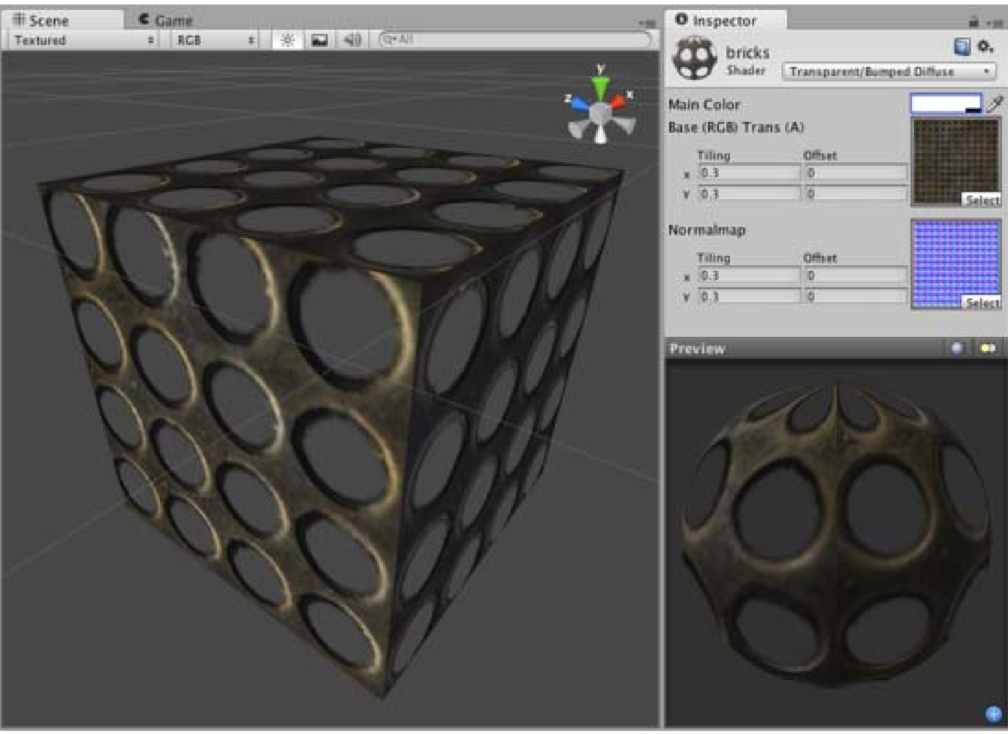
### Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

# shader-TransBumped Diffuse




## Transparent Properties

This shader can make mesh geometry partially or fully transparent by reading the alpha channel of the main texture. In the alpha, 0 (black) is completely transparent while 255 (white) is completely opaque. If your main texture does not have an alpha channel, the object will appear completely opaque.

Using transparent objects in your game can be tricky, as there are traditional graphical programming problems that can present sorting issues in your game. For example, if you see odd results when looking through two windows at once, you're experiencing the classical problem with using transparency. The general rule is to be aware that there are some cases in which one transparent object may be drawn in front of another in an unusual way, especially if the objects are intersecting, enclose each other or are of very different sizes. For this reason, you should use transparent objects if you need them, and try not to let them become excessive. You should also make your designer(s) aware that such sorting problems can occur, and have them prepare to change some design to work around these issues.

## Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

**Normal mapping** simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each pixel's color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the mesh's geometry when calculating lighting of the object.

### Creating Normal maps

You can import a regular grayscale image and convert it to a Normal Map from within Unity. To learn how to do this, please read the Normal map FAQ page.

### Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

1. It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.

2. It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

### Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

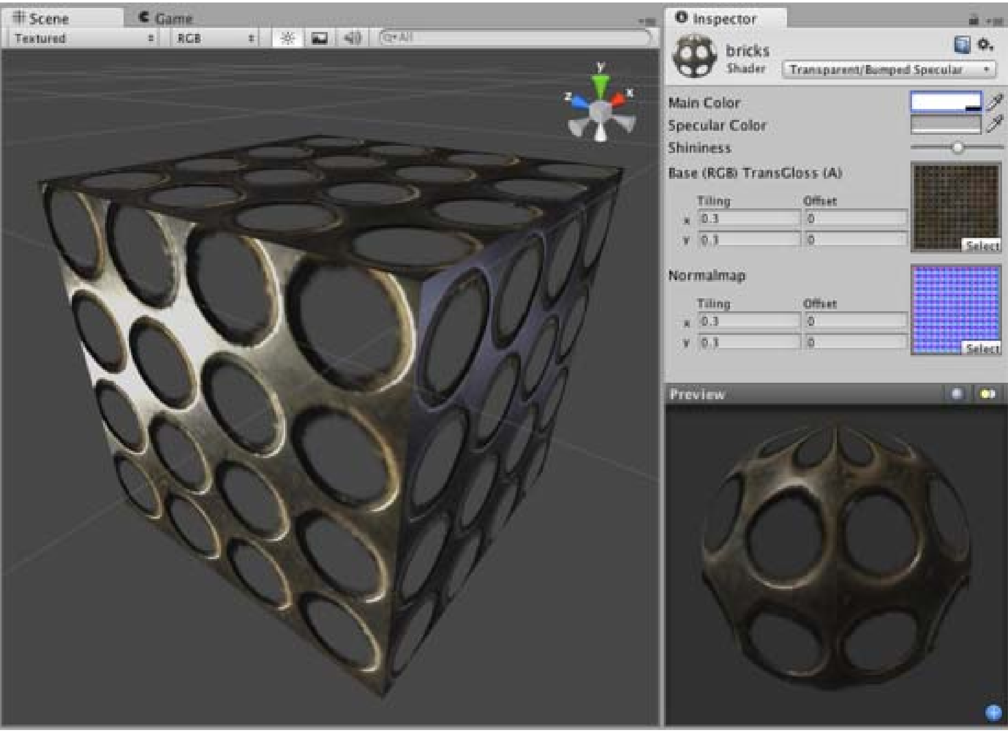
### Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-TransBumped Specular




One consideration for this shader is that the Base texture's alpha channel defines both the Transparent areas as well as the Specular Map.

### Transparent Properties

This shader can make mesh geometry partially or fully transparent by reading the alpha channel of the main texture. In the alpha, 0 (black) is completely transparent while 255 (white) is completely opaque. If your main texture does not have an alpha channel, the object will appear completely opaque.

Using transparent objects in your game can be tricky, as there are traditional graphical programming problems that can present sorting issues in your game. For example, if you see odd results when looking through two windows at once, you're experiencing the classical problem with using transparency. The general rule is to be aware that there are some cases in which one transparent object may be drawn in front of another in an unusual way, especially if the objects are intersecting, enclose each other or are of very different sizes. For this reason, you should use transparent objects if you need them, and try not to let them become excessive. You should also make your designer(s) aware that such sorting problems can occur, and have them prepare to change some design to work around these issues.

### Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

**Normal mapping** simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each pixel's color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the mesh's geometry when calculating lighting of the object.

#### Creating Normal maps

You can import a regular grayscale image and convert it to a Normal Map from within Unity. To learn how to do this, please read the Normal map FAQ page.

#### Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

1. It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
2. It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

### Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

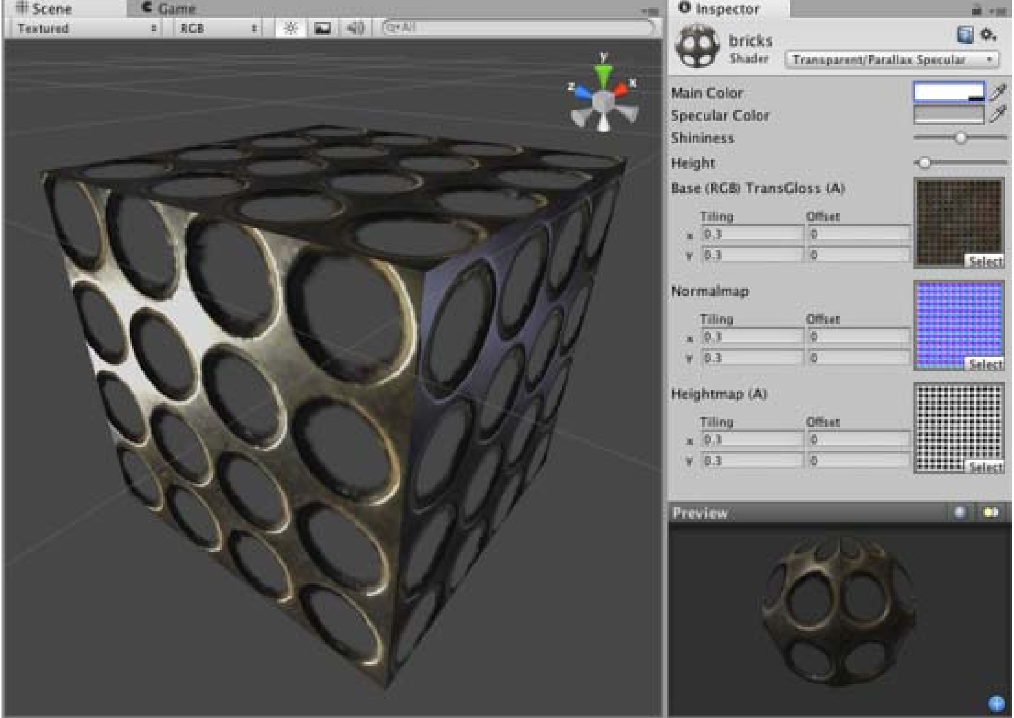
### Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-TransParallax Diffuse




## Transparent Properties

This shader can make mesh geometry partially or fully transparent by reading the alpha channel of the main texture. In the alpha, 0 (black) is completely transparent while 255 (white) is completely opaque. If your main texture does not have an alpha channel, the object will appear completely opaque.

Using transparent objects in your game can be tricky, as there are traditional graphical programming problems that can present sorting issues in your game. For example, if you see odd results when looking through two windows at once, you're experiencing the classical problem with using transparency. The general rule is to be aware that there are some cases in which one transparent object may be drawn in front of another in an unusual way, especially if the objects are intersecting, enclose each other or are of very different sizes. For this reason, you should use transparent objects if you need them, and try not to let them become excessive. You should also make your designer(s) aware that such sorting problems can occur, and have them prepare to change some design to work around these issues.

## Parallax Normal mapped Properties

**Parallax Normal mapped** is the same as regular **Normal mapped**, but with a better simulation of "depth". The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

## Diffuse Properties

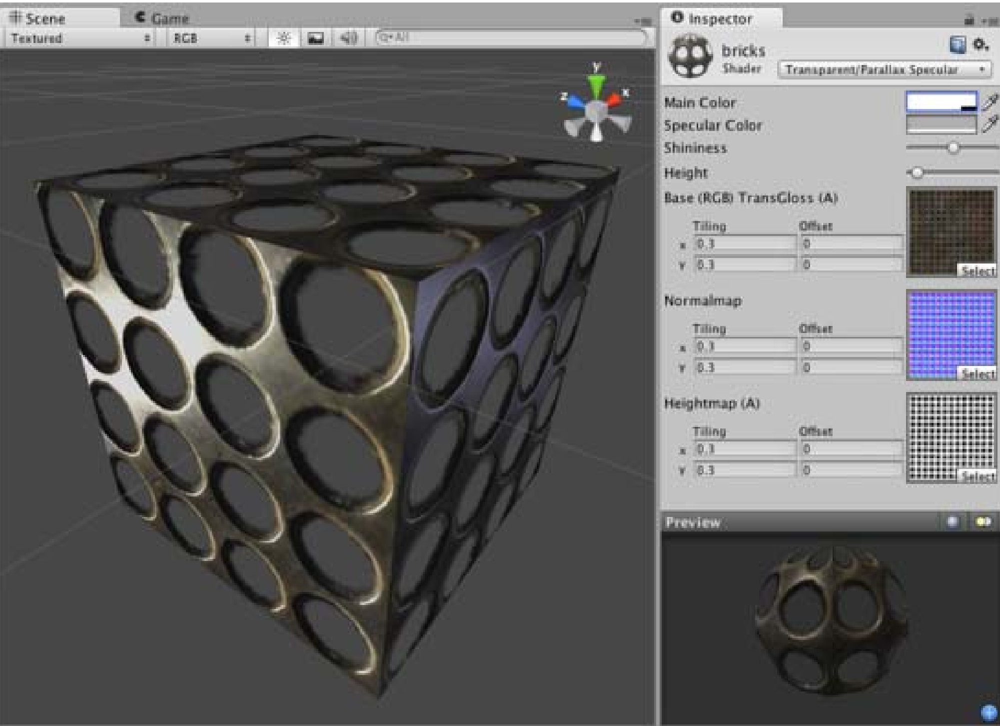
Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

## Performance

Generally, this shader is on the more expensive rendering side. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

## shader-TransParallax Specular




One consideration for this shader is that the Base texture's alpha channel defines both the Transparent areas as well as the Specular Map.

### Transparent Properties

This shader can make mesh geometry partially or fully transparent by reading the alpha channel of the main texture. In the alpha, 0 (black) is completely transparent while 255 (white) is completely opaque. If your main texture does not have an alpha channel, the object will appear completely opaque.

Using transparent objects in your game can be tricky, as there are traditional graphical programming problems that can present sorting issues in your game. For example, if you see odd results when looking through two windows at once, you're experiencing the classical problem with using transparency. The general rule is to be aware that there are some cases in which one transparent object may be drawn in front of another in an unusual way, especially if the objects are intersecting, enclose each other or are of very different sizes. For this reason, you should use transparent objects if you need them, and try not to let them become excessive. You should also make your designer(s) aware that such sorting problems can occur, and have them prepare to change some design to work around these issues.

### Parallax Normal mapped Properties

**Parallax Normal mapped** is the same as regular **Normal mapped**, but with a better simulation of "depth". The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

### Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas

will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

### Performance

Generally, this shader is on the more expensive rendering side. For more details, please view the Shader Peformance page.

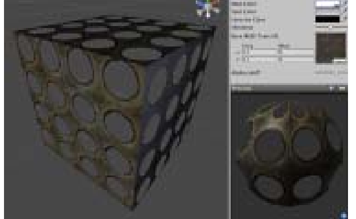
Page last updated: 2007-05-08

---

## shader-TransparentCutoutFamily

The Transparent Cutout shaders are used for objects that have fully opaque and fully transparent parts (no partial transparency). Things like chain fences, trees, grass, etc.

### Transparent Cutout Vertex-Lit

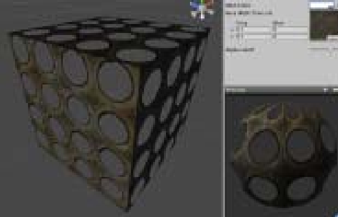


**Assets needed:**

- One **Base** texture with alpha channel for Transparency Map

» More details

### Transparent Cutout Diffuse

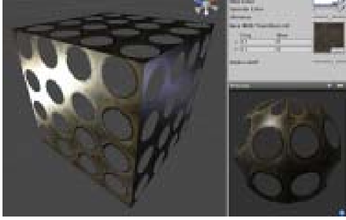


**Assets needed:**

- One **Base** texture with alpha channel for Transparency Map

» More details

### Transparent Cutout Specular



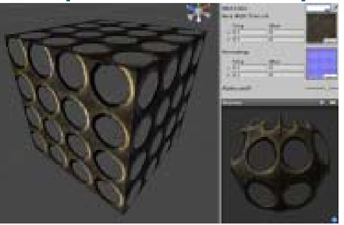
**Assets needed:**

- One **Base** texture with alpha channel for combined Transparency Map/Specular Map

**Note:** One limitation of this shader is that the **Base** texture's alpha channel doubles as a Specular Map for the Specular shaders in this family.

» More details

### Transparent Cutout Bumped

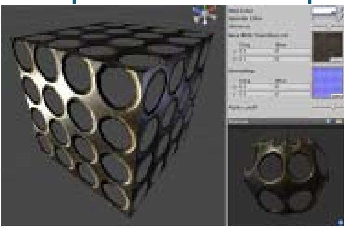


**Assets needed:**

- One **Base** texture with alpha channel for Transparency Map
- One **Normal map** normal map, no alpha channel required

» More details

### Transparent Cutout Bumped Specular



**Assets needed:**

- One **Base** texture with alpha channel for combined Transparency Map/Specular Map
- One **Normal map** normal map, no alpha channel required

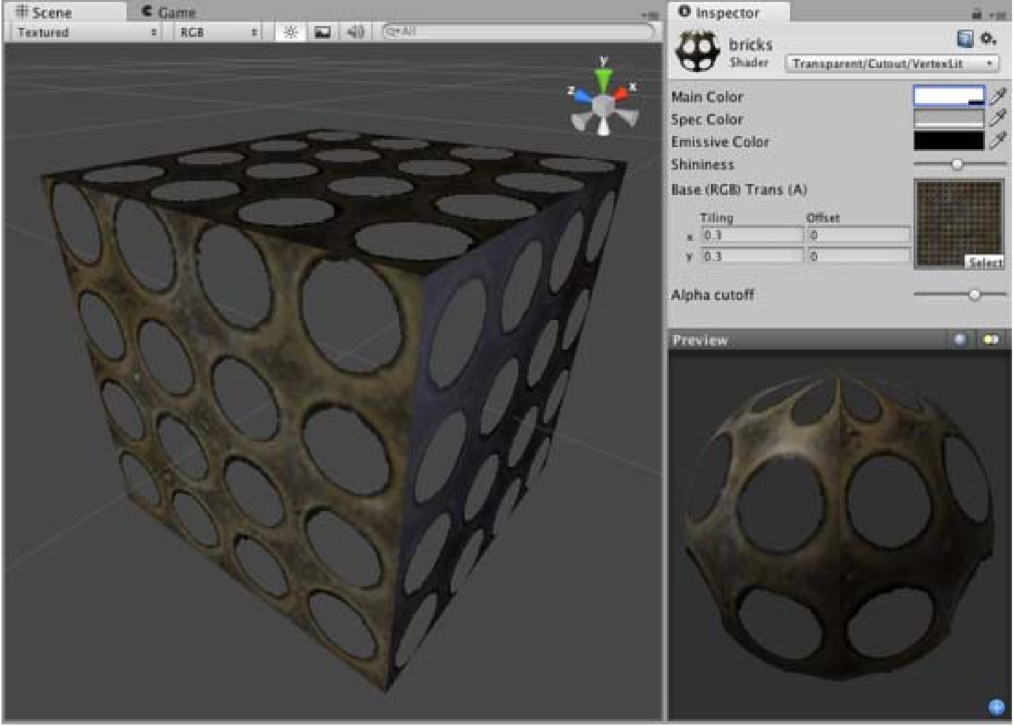
**Note:** One limitation of this shader is that the **Base** texture's alpha channel doubles as a Specular Map for the Specular shaders in this family.

» More details

Page last updated: 2010-07-13

---

# shader-TransCutVertexLit




### Transparent Cutout Properties

Cutout shader is an alternative way of displaying transparent objects. Differences between Cutout and regular Transparent shaders are:

- This shader cannot have partially transparent areas. Everything will be either fully opaque or fully transparent.
- Objects using this shader can cast and receive shadows!
- The graphical sorting problems normally associated with Transparent shaders do not occur when using this shader.

This shader uses an alpha channel contained in the **Base** Texture to determine the transparent areas. If the alpha contains a blend between transparent and opaque areas, you can manually determine the cutoff point for the which areas will be shown. You change this cutoff by adjusting the **Alpha Cutoff** slider.

### Vertex-Lit Properties

This shader is **Vertex-Lit**, which is one of the simplest shaders. All lights shining on it are rendered in a single pass and calculated at vertices only.

Because it is vertex-lit, it won't display any pixel-based rendering effects, such as light cookies, normal mapping, or shadows. This shader is also much more sensitive to tesselation of the models. If you put a point light very close to a cube using this shader, the light will only be calculated at the corners. Pixel-lit shaders are much more effective at creating a nice round highlight, independent of tesselation. If that's an effect you want, you may consider using a pixel-lit shader or increase tesselation of the objects instead.

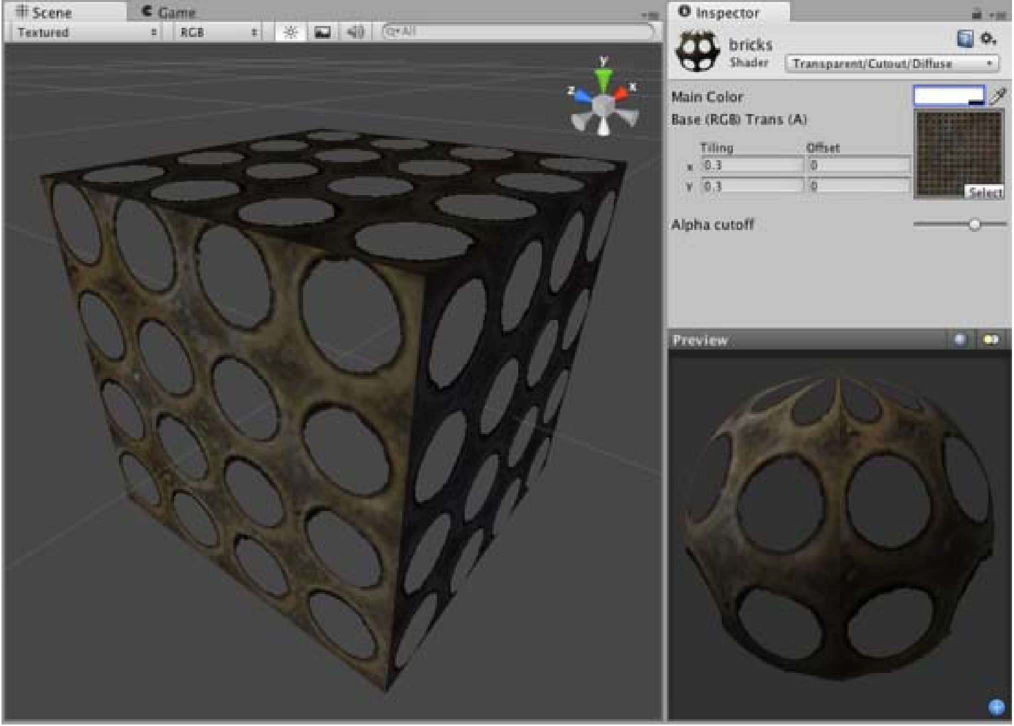
### Performance

Generally, this shader is very cheap to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-18

---

## shader-TransCutDiffuse




### Transparent Cutout Properties

Cutout shader is an alternative way of displaying transparent objects. Differences between Cutout and regular Transparent shaders are:

- This shader cannot have partially transparent areas. Everything will be either fully opaque or fully transparent.
- Objects using this shader can cast and receive shadows!
- The graphical sorting problems normally associated with Transparent shaders do not occur when using this shader.

This shader uses an alpha channel contained in the **Base** Texture to determine the transparent areas. If the alpha contains a blend between transparent and opaque areas, you can manually determine the cutoff point for the which areas will be shown. You change this cutoff by adjusting the **Alpha Cutoff** slider.

### Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

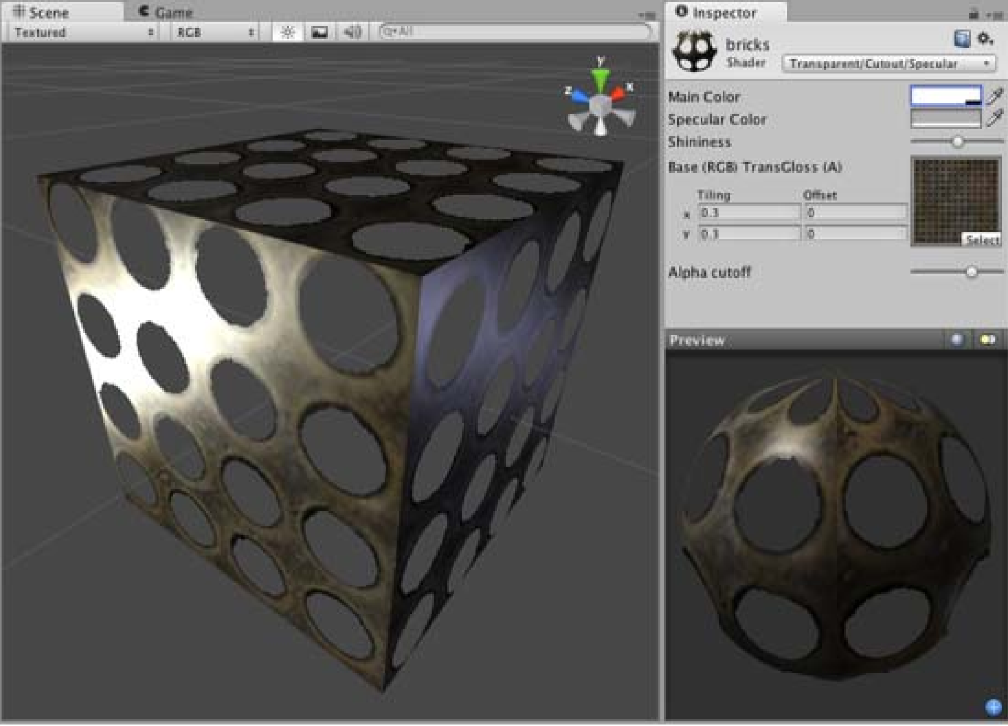
### Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-18

---

## shader-TransCutSpecular




One consideration for this shader is that the Base texture's alpha channel defines both the Transparent areas as well as the Specular Map.

### Transparent Cutout Properties

Cutout shader is an alternative way of displaying transparent objects. Differences between Cutout and regular Transparent shaders are:

- This shader cannot have partially transparent areas. Everything will be either fully opaque or fully transparent.
- Objects using this shader can cast and receive shadows!
- The graphical sorting problems normally associated with Transparent shaders do not occur when using this shader.

This shader uses an alpha channel contained in the **Base** Texture to determine the transparent areas. If the alpha contains a blend between transparent and opaque areas, you can manually determine the cutoff point for the which areas will be shown. You change this cutoff by adjusting the **Alpha Cutoff** slider.

### Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

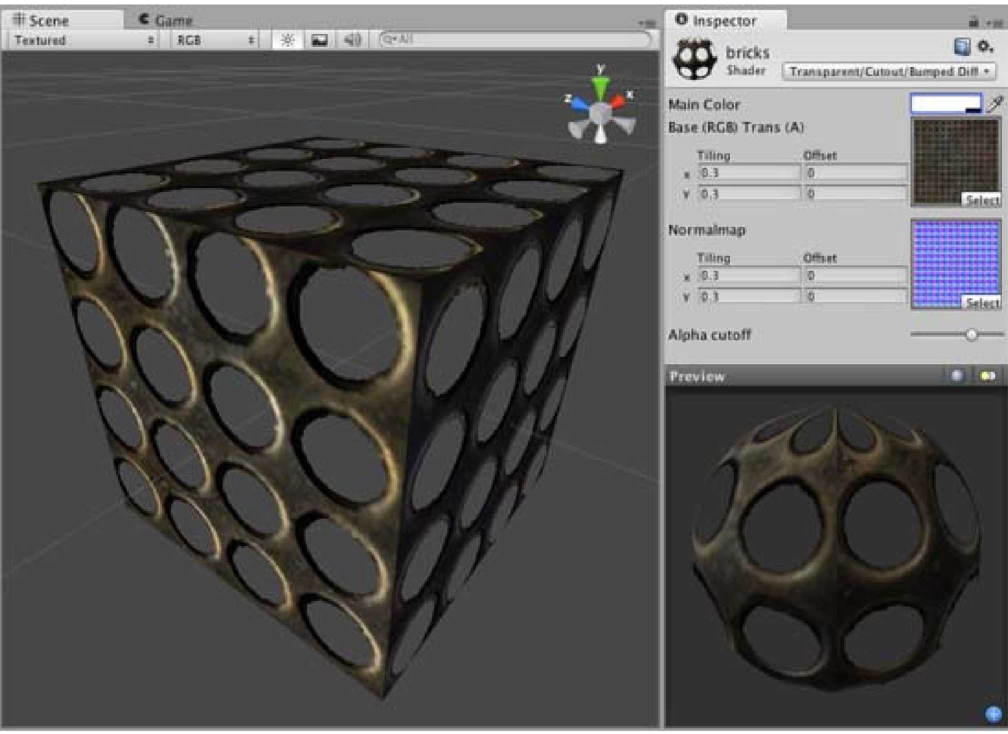
### Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-18

---

# shader-TransCutBumpedDiffuse




## Transparent Cutout Properties

Cutout shader is an alternative way of displaying transparent objects. Differences between Cutout and regular Transparent shaders are:

- This shader cannot have partially transparent areas. Everything will be either fully opaque or fully transparent.
- Objects using this shader can cast and receive shadows!
- The graphical sorting problems normally associated with Transparent shaders do not occur when using this shader.

This shader uses an alpha channel contained in the **Base** Texture to determine the transparent areas. If the alpha contains a blend between transparent and opaque areas, you can manually determine the cutoff point for the which areas will be shown. You change this cutoff by adjusting the **Alpha Cutoff** slider.

## Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

**Normal mapping** simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each pixel's color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the mesh's geometry when calculating lighting of the object.

### Creating Normal maps

You can import a regular grayscale image and convert it to a Normal Map from within Unity. To learn how to do this, please read the Normal map FAQ page.

### Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

1. It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
2. It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

### Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

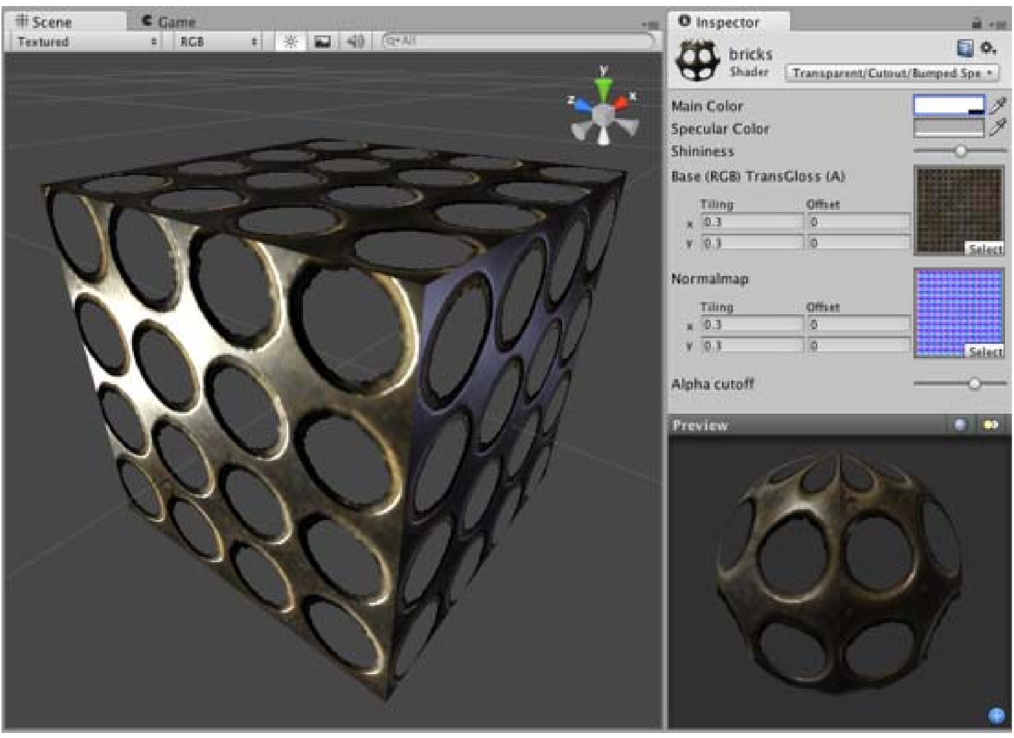
### Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-18

---

## shader-TransCutBumpedSpecular




One consideration for this shader is that the Base texture's alpha channel defines both the Transparent areas as well as the Specular Map.

### Transparent Cutout Properties

Cutout shader is an alternative way of displaying transparent objects. Differences between Cutout and regular Transparent shaders are:

- This shader cannot have partially transparent areas. Everything will be either fully opaque or fully transparent.
- Objects using this shader can cast and receive shadows!
- The graphical sorting problems normally associated with Transparent shaders do not occur when using this shader.

This shader uses an alpha channel contained in the **Base** Texture to determine the transparent areas. If the alpha contains a blend between transparent and opaque areas, you can manually determine the cutoff point for the which areas will be shown. You change this cutoff by adjusting the **Alpha Cutoff** slider.

### Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

**Normal mapping** simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each pixel's color value represents the angle of the surface normal. Then by using this value instead

of the one from geometry, lighting is computed. The normal map effectively overrides the mesh's geometry when calculating lighting of the object.

#### Creating Normal maps

You can import a regular grayscale image and convert it to a Normal Map from within Unity. To learn how to do this, please read the Normal map FAQ page.

#### Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

1. It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
2. It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

### Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

### Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-18

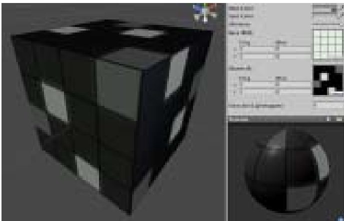
---

## shader-SelfIllumFamily

The **Self-Illuminated** shaders will emit light only onto themselves based on an attached alpha channel. They do not require any Lights to shine on them to emit this light. Any vertex lights or pixel lights will simply add more light on top of the self-illumination.

This is mostly used for light emitting objects. For example, parts of the wall texture could be self-illuminated to simulate lights or displays. It can also be useful to light power-up objects that should always have consistent lighting throughout the game, regardless of the lights shining on it.

### Self-Illuminated Vertex-Lit

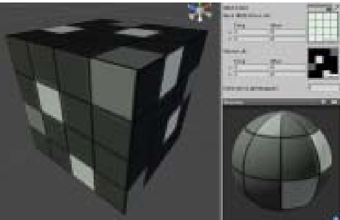


**Assets needed:**

- One **Base** texture, no alpha channel required
- One **Illumination** texture with alpha channel for Illumination Map

» More details

## Self-Illuminated Diffuse

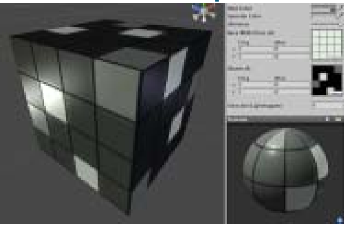


**Assets needed:**

- One **Base** texture, no alpha channel required
- One **Illumination** texture with alpha channel for Illumination Map

» More details

## Self-Illuminated Specular

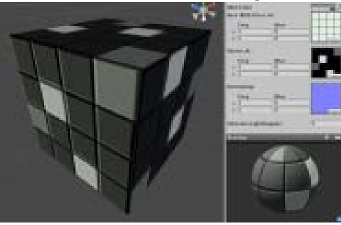


**Assets needed:**

- One **Base** texture with alpha channel for Specular Map
- One **Illumination** texture with alpha channel for Illumination Map

» More details

## Self-Illuminated Bumped

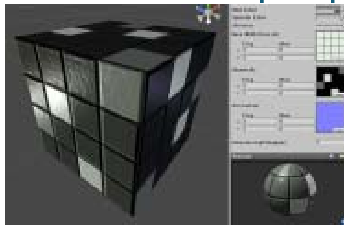


**Assets needed:**

- One **Base** texture, no alpha channel required
- One **Normal map** normal map with alpha channel for Illumination

» More details

## Self-Illuminated Bumped Specular

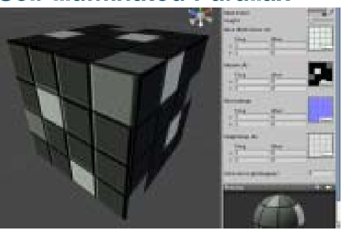


**Assets needed:**

- One **Base** texture with alpha channel for Specular Map
- One **Normal map** normal map with alpha channel for Illumination Map

» More details

### Self-Illuminated Parallax



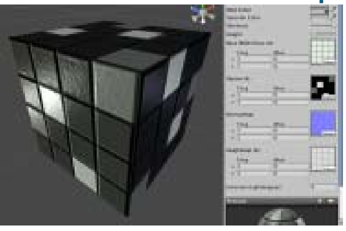
**Assets needed:**

- One **Base** texture, no alpha channel required
- One **Normal map** normal map with alpha channel for Illumination Map & Parallax Depth combined

**Note:** One consideration of this shader is that the **Bumpmap** texture's alpha channel doubles as a Illumination and the Parallax Depth.

» More details

### Self-Illuminated Parallax Specular



**Assets needed:**

- One **Base** texture with alpha channel for Specular Map
- One **Normal map** normal map with alpha channel for Illumination Map & Parallax Depth combined

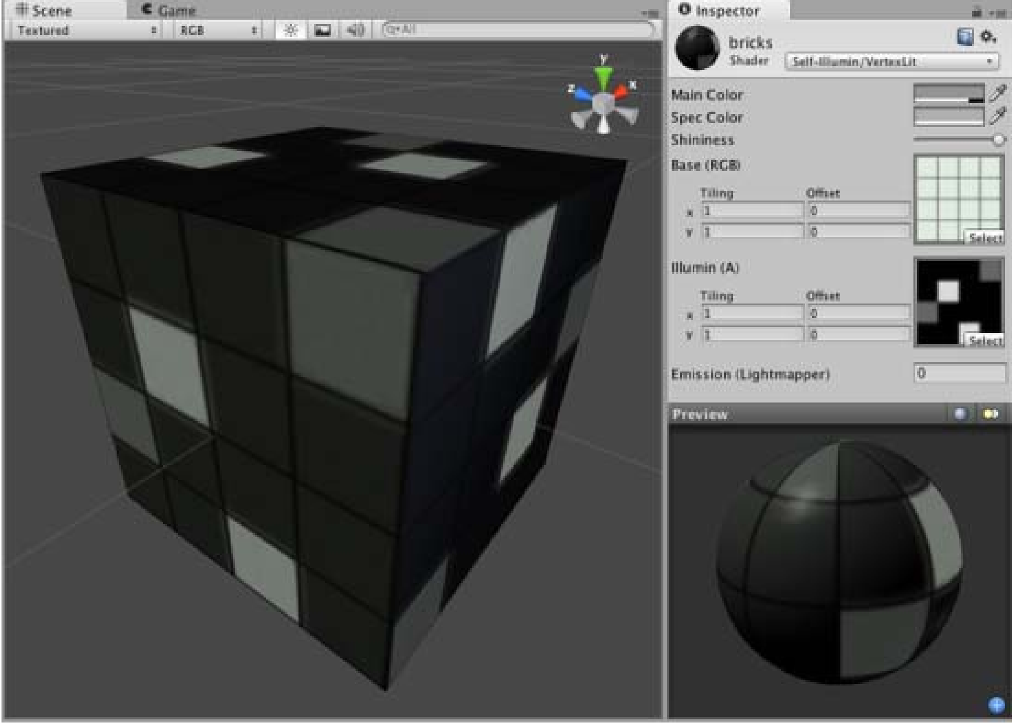
**Note:** One consideration of this shader is that the **Bumpmap** texture's alpha channel doubles as a Illumination and the Parallax Depth.

» More details

Page last updated: 2010-07-13

---

# shader-SelfIllumVertexLit




### Self-Illuminated Properties

This shader allows you to define bright and dark parts of the object. The alpha channel of a secondary texture will define areas of the object that "emit" light by themselves, even when no light is shining on it. In the alpha channel, black is zero light, and white is full light emitted by the object. Any scene lights will add illumination on top of the shader's illumination. So even if your object does not emit any light by itself, it will still be lit by lights in your scene.

### Vertex-Lit Properties

This shader is **Vertex-Lit**, which is one of the simplest shaders. All lights shining on it are rendered in a single pass and calculated at vertices only.

Because it is vertex-lit, it won't display any pixel-based rendering effects, such as light cookies, normal mapping, or shadows. This shader is also much more sensitive to tesselation of the models. If you put a point light very close to a cube using this shader, the light will only be calculated at the corners. Pixel-lit shaders are much more effective at creating a nice round highlight, independent of tesselation. If that's an effect you want, you may consider using a pixel-lit shader or increase tesselation of the objects instead.

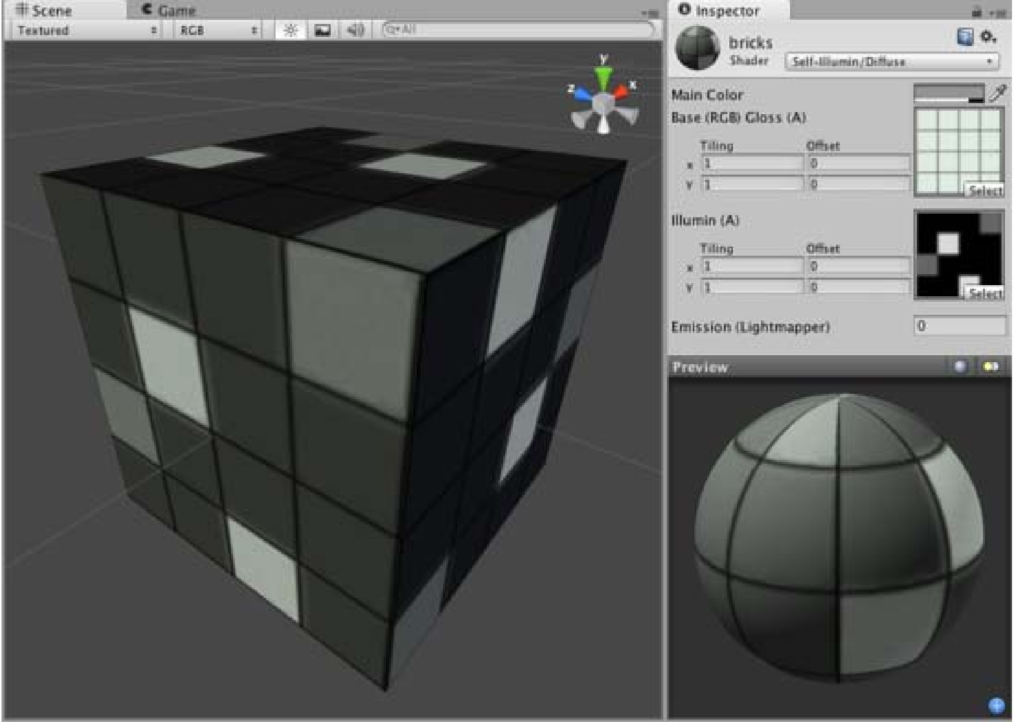
### Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-SelfIllumDiffuse




### Self-Illuminated Properties

This shader allows you to define bright and dark parts of the object. The alpha channel of a secondary texture will define areas of the object that "emit" light by themselves, even when no light is shining on it. In the alpha channel, black is zero light, and white is full light emitted by the object. Any scene lights will add illumination on top of the shader's illumination. So even if your object does not emit any light by itself, it will still be lit by lights in your scene.

### Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

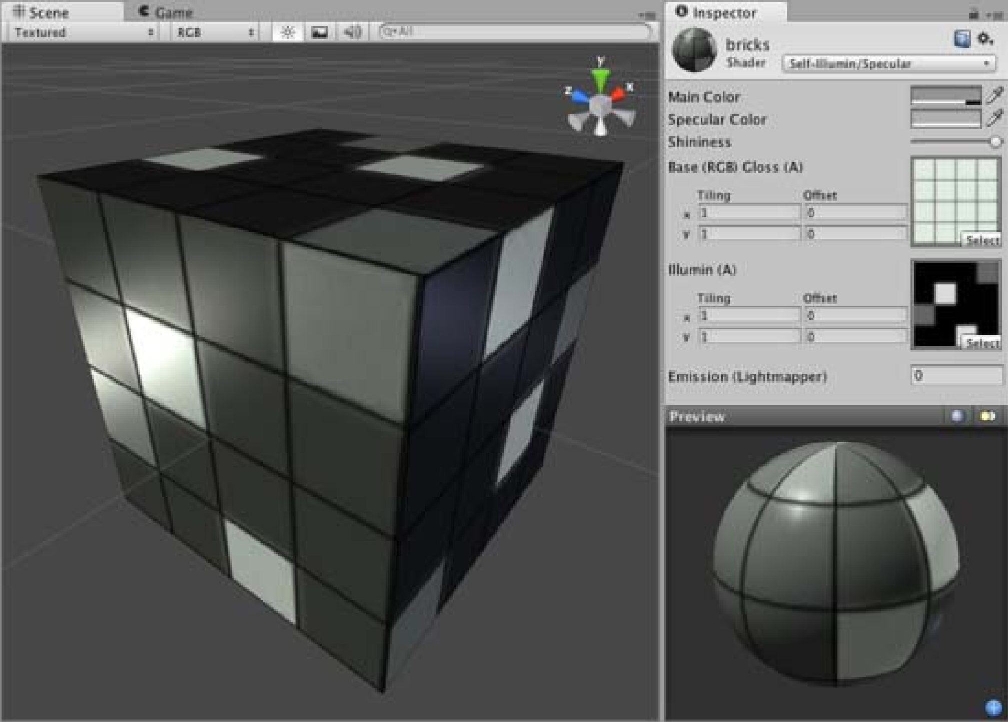
### Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-SelfIllumSpecular




### Self-Illuminated Properties

This shader allows you to define bright and dark parts of the object. The alpha channel of a secondary texture will define areas of the object that "emit" light by themselves, even when no light is shining on it. In the alpha channel, black is zero light, and white is full light emitted by the object. Any scene lights will add illumination on top of the shader's illumination. So even if your object does not emit any light by itself, it will still be lit by lights in your scene.

### Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

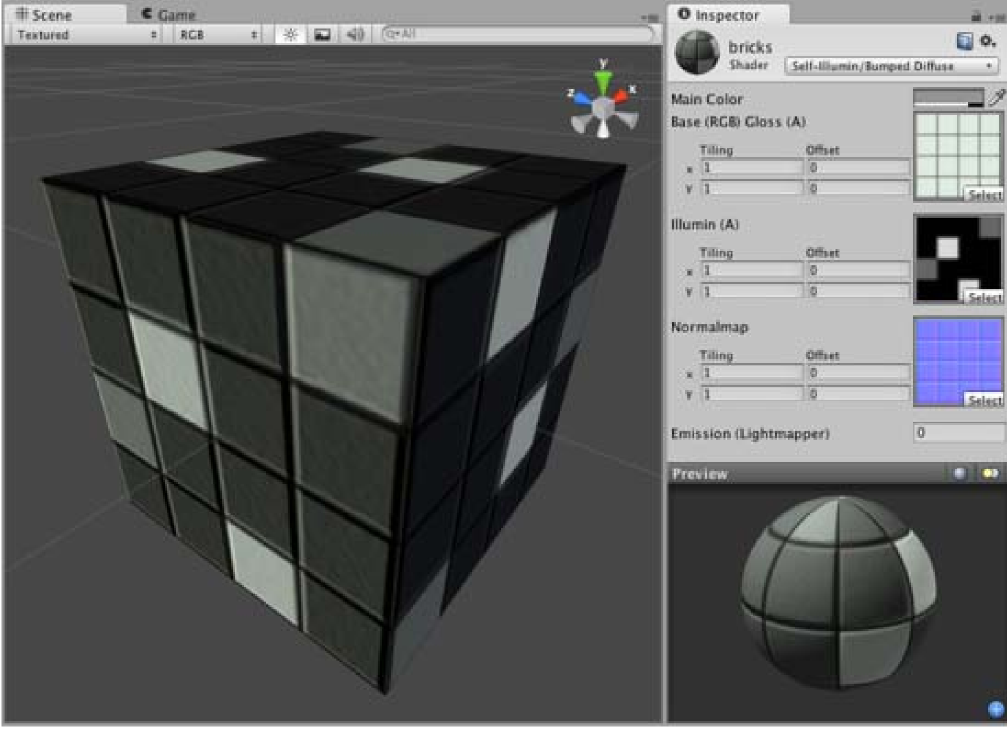
### Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-SelfIllumBumpedDiffuse




## Self-Illuminated Properties

This shader allows you to define bright and dark parts of the object. The alpha channel of a secondary texture will define areas of the object that "emit" light by themselves, even when no light is shining on it. In the alpha channel, black is zero light, and white is full light emitted by the object. Any scene lights will add illumination on top of the shader's illumination. So even if your object does not emit any light by itself, it will still be lit by lights in your scene.

## Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

**Normal mapping** simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each pixel's color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the mesh's geometry when calculating lighting of the object.

### Creating Normal maps

You can import a regular grayscale image and convert it to a Normal Map from within Unity. To learn how to do this, please read the Normal map FAQ page.

### Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

1. It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
2. It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

## Diffuse Properties

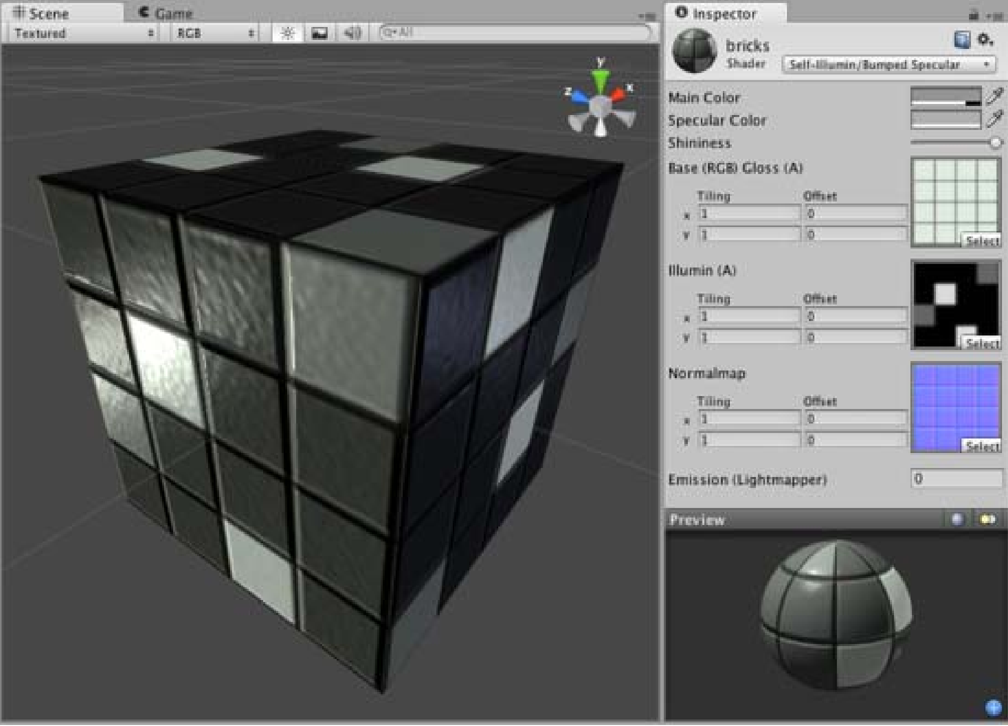
Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

## Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Page last updated: 2012-08-23

## shader-SelfIllumBumpedSpecular




### Self-Illuminated Properties

This shader allows you to define bright and dark parts of the object. The alpha channel of a secondary texture will define areas of the object that "emit" light by themselves, even when no light is shining on it. In the alpha channel, black is zero light, and white is full light emitted by the object. Any scene lights will add illumination on top of the shader's illumination. So even if your object does not emit any light by itself, it will still be lit by lights in your scene.

### Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

**Normal mapping** simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each pixel's color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the mesh's geometry when calculating lighting of the object.

#### Creating Normal maps

You can import a regular grayscale image and convert it to a Normal Map from within Unity. To learn how to do this, please read the Normal map FAQ page.

#### Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

1. It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
2. It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

### Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle.

The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

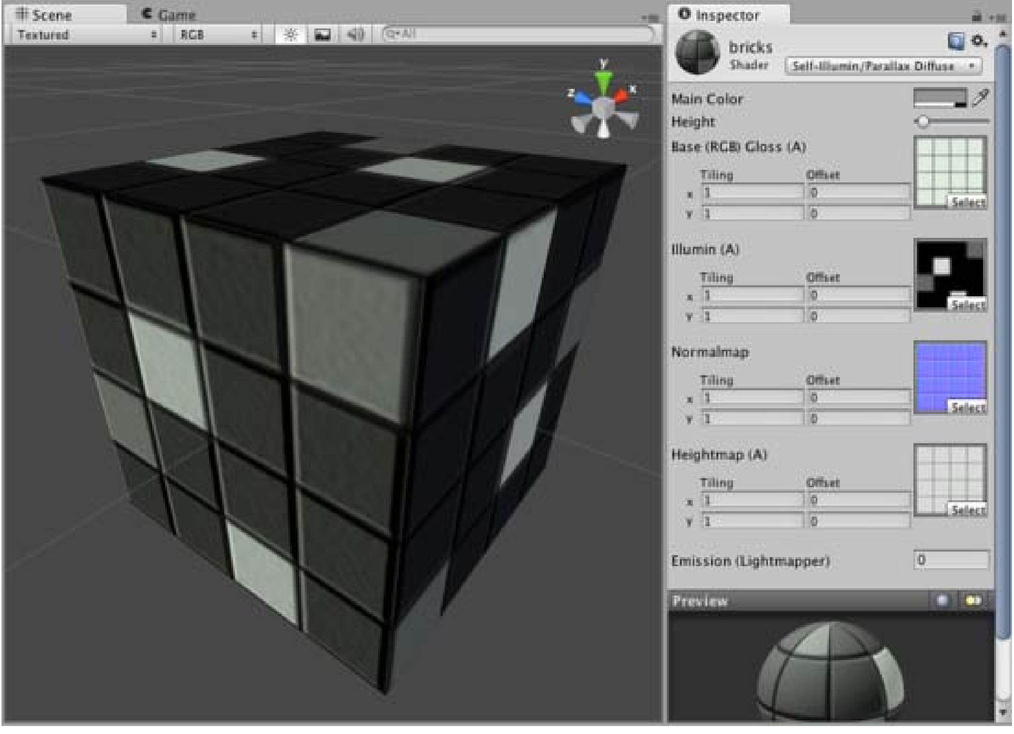
### Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Page last updated: 2012-08-23

---

## shader-SelfIllumParallaxDiffuse




### Self-Illuminated Properties

This shader allows you to define bright and dark parts of the object. The alpha channel of a secondary texture will define areas of the object that "emit" light by themselves, even when no light is shining on it. In the alpha channel, black is zero light, and white is full light emitted by the object. Any scene lights will add illumination on top of the shader's illumination. So even if your object does not emit any light by itself, it will still be lit by lights in your scene.

### Parallax Normal mapped Properties

**Parallax Normal mapped** is the same as regular **Normal mapped**, but with a better simulation of "depth". The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

### Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

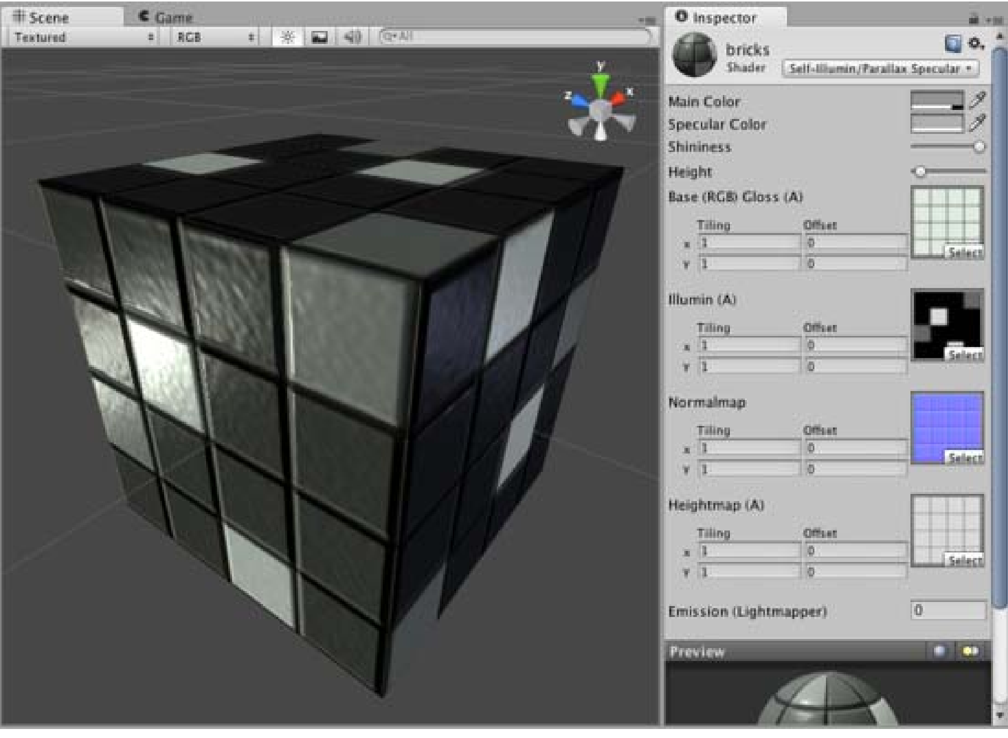
### Performance

Generally, this shader is on the more expensive rendering side. For more details, please view the Shader Peformance page.

Page last updated: 2012-08-23

---

## shader-SelfIllumParallaxSpecular




### Self-Illuminated Properties

This shader allows you to define bright and dark parts of the object. The alpha channel of a secondary texture will define areas of the object that "emit" light by themselves, even when no light is shining on it. In the alpha channel, black is zero light, and white is full light emitted by the object. Any scene lights will add illumination on top of the shader's illumination. So even if your object does not emit any light by itself, it will still be lit by lights in your scene.

### Parallax Normal mapped Properties

**Parallax Normal mapped** is the same as regular **Normal mapped**, but with a better simulation of "depth". The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

### Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

### Performance

Generally, this shader is on the more expensive rendering side. For more details, please view the Shader Peformance page.

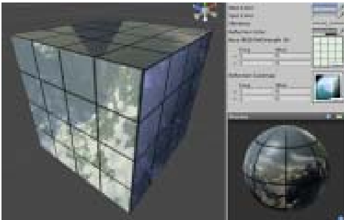
Page last updated: 2012-08-23

---

## shader-ReflectiveFamily

**Reflective** shaders will allow you to use a Cubemap which will be reflected on your mesh. You can also define areas of more or less reflectivity on your object through the alpha channel of the **Base** texture. High relectivity is a great effect for glosses, oils, chrome, etc. Low reflectivity can add effect for metals, liquid surfaces, or video monitors.

### Reflective Vertex-Lit

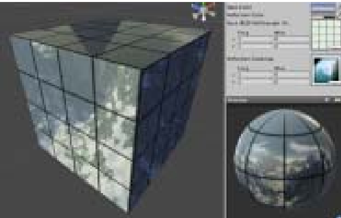


**Assets needed:**

- One **Base** texture with alpha channel for defining reflective areas
- One **Reflection** Cubemap for Reflection Map

» More details

### Reflective Diffuse



**Assets needed:**

- One **Base** texture with alpha channel for defining reflective areas
- One **Reflection** Cubemap for Reflection Map

» More details

### Reflective Specular

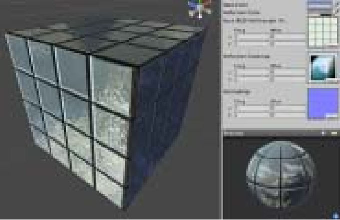
**Assets needed:**

- One **Base** texture with alpha channel for defining reflective areas & Specular Map combined
- One **Reflection** Cubemap for Reflection Map

**Note:** One consideration for this shader is that the **Base** texture's alpha channel will double as both the reflective areas and the Specular Map.

» More details

## Reflective Normal mapped



**Assets needed:**

- One **Base** texture with alpha channel for defining reflective areas
- One **Reflection** Cubemap for Reflection Map
- One **Normal map** normal map, no alpha channel required

» More details

## Reflective Normal Mapped Specular

**Assets needed:**

- One **Base** texture with alpha channel for defining reflective areas & Specular Map combined
- One **Reflection** Cubemap for Reflection Map
- One **Normal map** normal map, no alpha channel required

**Note:** One consideration for this shader is that the **Base** texture's alpha channel will double as both the reflective areas and the Specular Map.

» More details

## Reflective Parallax

**Assets needed:**

- One **Base** texture with alpha channel for defining reflective areas
- One **Reflection** Cubemap for Reflection Map
- One **Normal map** normal map, with alpha channel for Parallax Depth

» More details

## Reflective Parallax Specular

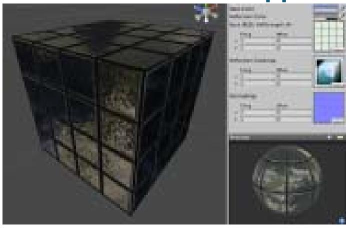
**Assets needed:**

- One **Base** texture with alpha channel for defining reflective areas & Specular Map
- One **Reflection** Cubemap for Reflection Map
- One **Normal map** normal map, with alpha channel for Parallax Depth

**Note:** One consideration for this shader is that the **Base** texture's alpha channel will double as both the reflective areas and the Specular Map.

» More details

### Reflective Normal mapped Unlit

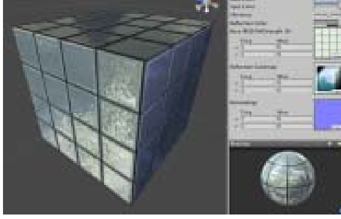


**Assets needed:**

- One **Base** texture with alpha channel for defining reflective areas
- One **Reflection** Cubemap for Reflection Map
- One **Normal map** normal map, no alpha channel required

» More details

### Reflective Normal mapped Vertex-Lit



**Assets needed:**

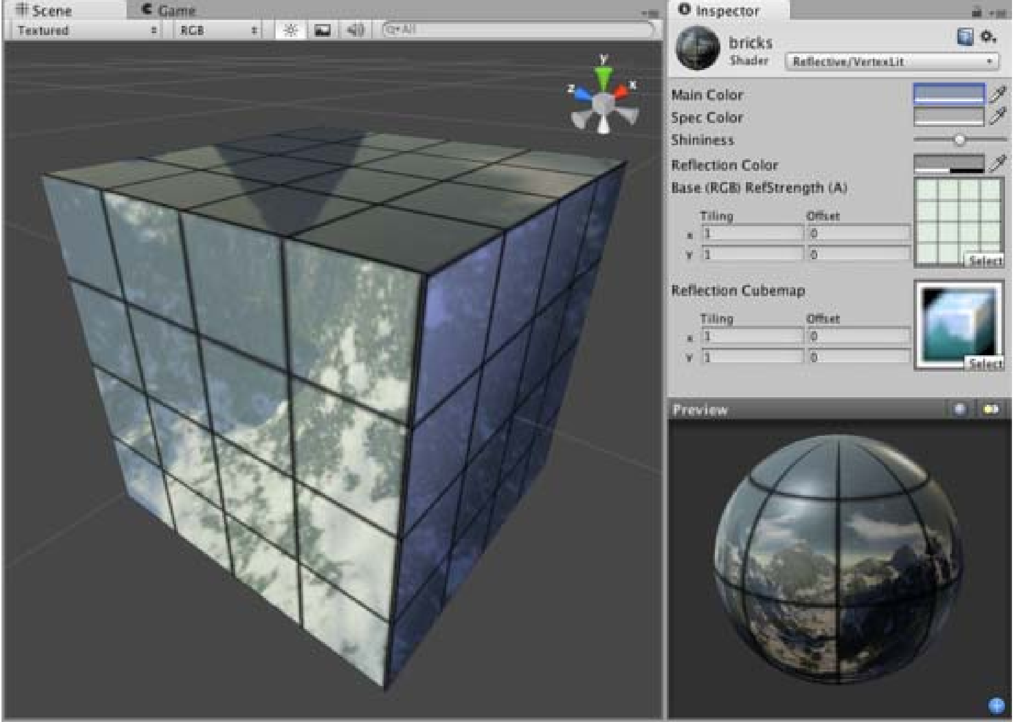
- One **Base** texture with alpha channel for defining reflective areas
- One **Reflection** Cubemap for Reflection Map
- One **Normal map** normal map, no alpha channel required

» More details

Page last updated: 2010-07-13

---

## shader-ReflectiveVertexLit




### Reflective Properties

This shader will simulate reflective surfaces such as cars, metal objects etc. It requires an environment Cubemap which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any scene lights will add illumination on top of what is reflected.

### Vertex-Lit Properties

This shader is **Vertex-Lit**, which is one of the simplest shaders. All lights shining on it are rendered in a single pass and calculated at vertices only.

Because it is vertex-lit, it won't display any pixel-based rendering effects, such as light cookies, normal mapping, or shadows. This shader is also much more sensitive to tesselation of the models. If you put a point light very close to a cube using this shader, the light will only be calculated at the corners. Pixel-lit shaders are much more effective at creating a nice round highlight, independent of tesselation. If that's an effect you want, you may consider using a pixel-lit shader or increase tesselation of the objects instead.

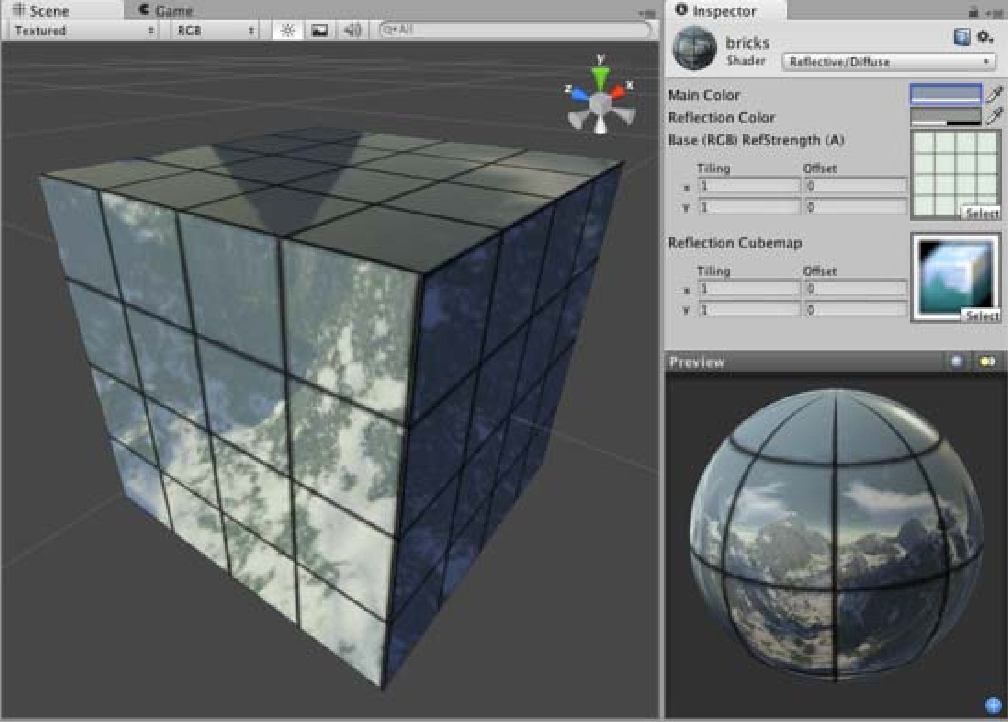
### Performance

Generally, this shader is not too expensive to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-ReflectiveDiffuse




### Reflective Properties

This shader will simulate reflective surfaces such as cars, metal objects etc. It requires an environment Cubemap which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any scene lights will add illumination on top of what is reflected.

### Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

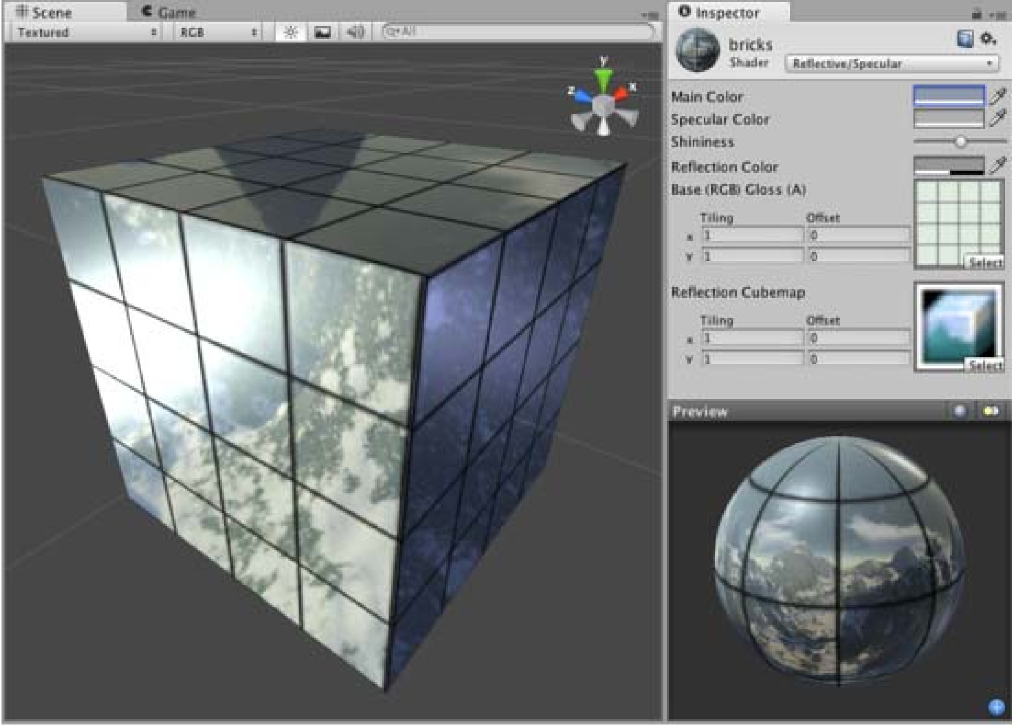
### Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-ReflectiveSpecular




One consideration for this shader is that the Base texture's alpha channel will double as both the Reflection Map and the Specular Map.

### Reflective Properties

This shader will simulate reflective surfaces such as cars, metal objects etc. It requires an environment Cubemap which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any scene lights will add illumination on top of what is reflected.

### Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

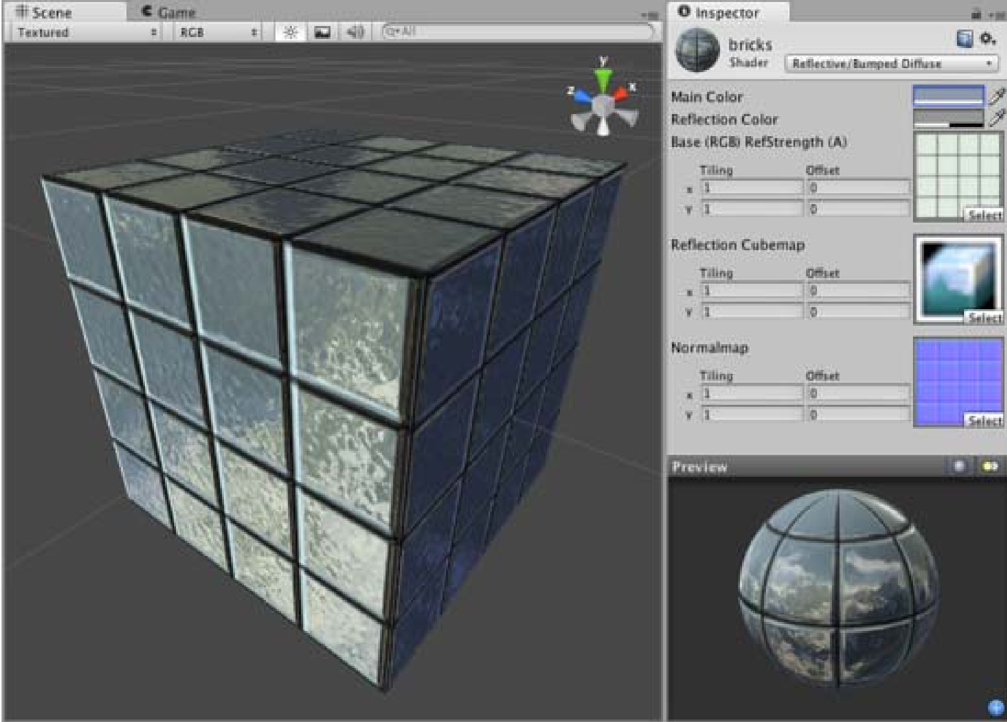
### Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-ReflectiveBumpedDiffuse




### Reflective Properties

This shader will simulate reflective surfaces such as cars, metal objects etc. It requires an environment Cubemap which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any scene lights will add illumination on top of what is reflected.

### Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

**Normal mapping** simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each pixel's color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the mesh's geometry when calculating lighting of the object.

#### Creating Normal maps

You can import a regular grayscale image and convert it to a Normal Map from within Unity. To learn how to do this, please read the Normal map FAQ page.

#### Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

1. It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
2. It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

### Diffuse Properties

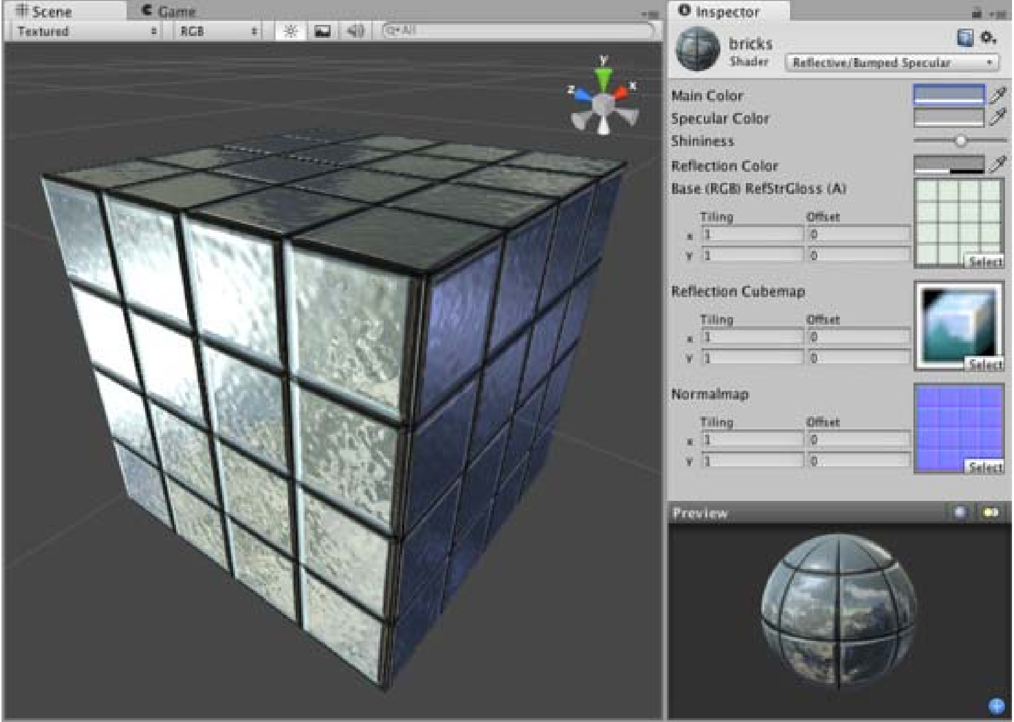
Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

### Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

## shader-ReflectiveBumpedSpecular




One consideration for this shader is that the Base texture's alpha channel will double as both the Reflection Map and the Specular Map.

### Reflective Properties

This shader will simulate reflective surfaces such as cars, metal objects etc. It requires an environment Cubemap which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any scene lights will add illumination on top of what is reflected.

### Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

**Normal mapping** simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each pixel's color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the mesh's geometry when calculating lighting of the object.

#### Creating Normal maps

You can import a regular grayscale image and convert it to a Normal Map from within Unity. To learn how to do this, please read the Normal map FAQ page.

#### Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

1. It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
2. It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

### Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

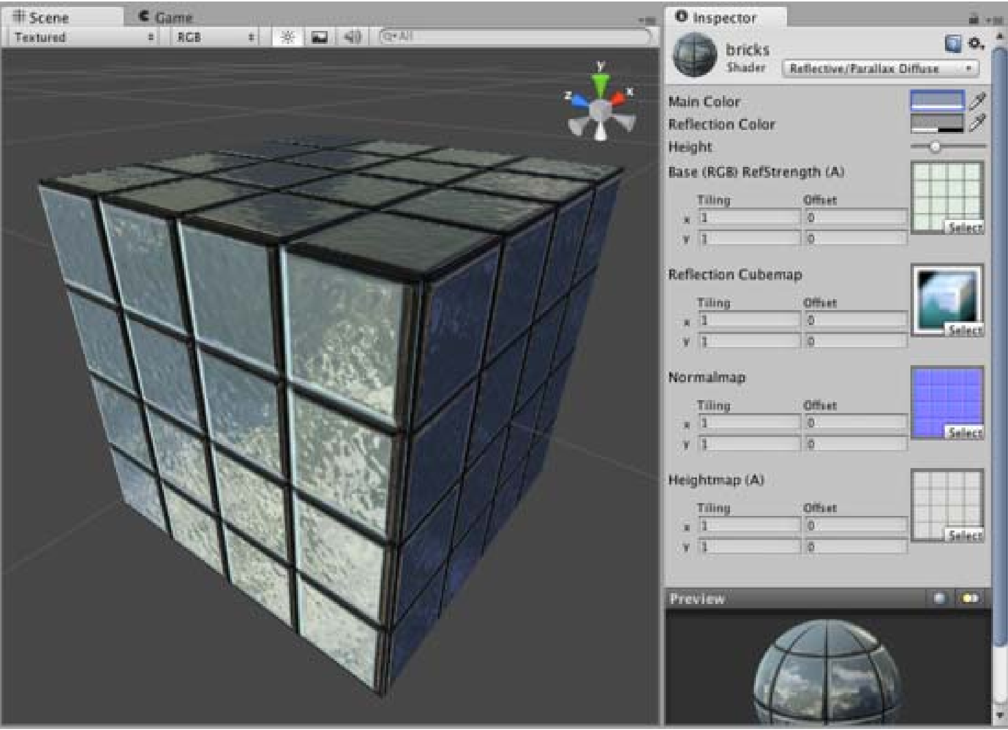
### Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Page last updated: 2010-07-14

---

## shader-ReflectiveParallaxDiffuse




### Reflective Properties

This shader will simulate reflective surfaces such as cars, metal objects etc. It requires an environment Cubemap which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any scene lights will add illumination on top of what is reflected.

### Parallax Normal mapped Properties

**Parallax Normal mapped** is the same as regular **Normal mapped**, but with a better simulation of "depth". The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

### Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the this angle, and does not change as the camera moves or rotates around.

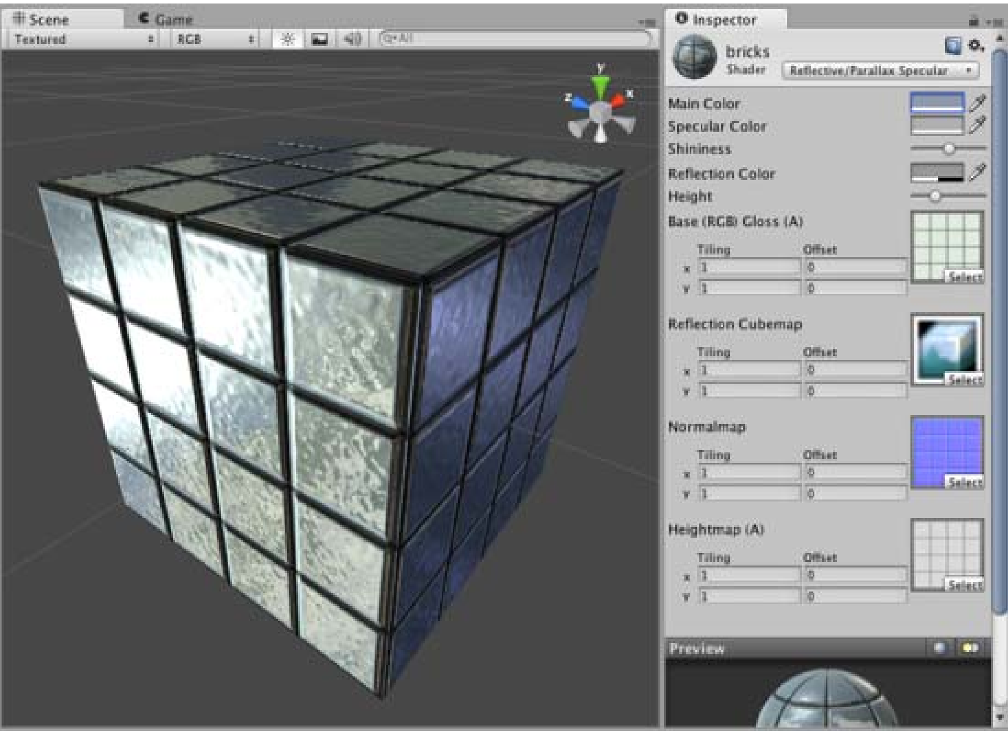
### Performance

Generally, this shader is on the more expensive rendering side. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-ReflectiveParallaxSpecular




One consideration for this shader is that the Base texture's alpha channel will double as both the Reflection Map and the Specular Map.

### Reflective Properties

This shader will simulate reflective surfaces such as cars, metal objects etc. It requires an environment Cubemap which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any scene lights will add illumination on top of what is reflected.

### Parallax Normal mapped Properties

**Parallax Normal mapped** is the same as regular **Normal mapped**, but with a better simulation of "depth". The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

### Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the

highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

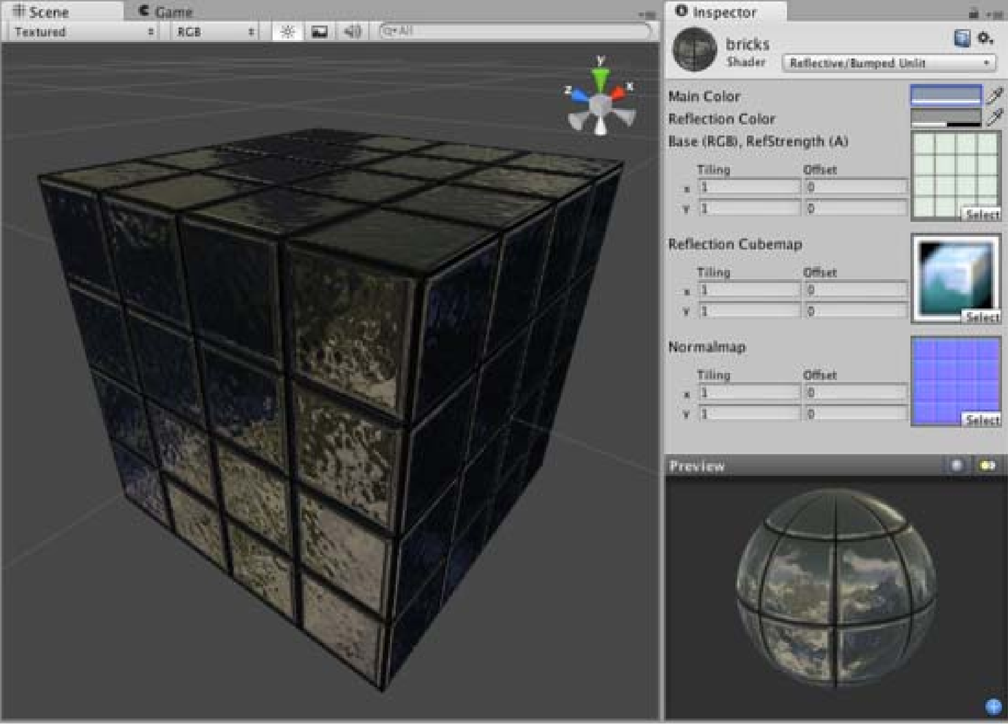
### Performance

Generally, this shader is on the more expensive rendering side. For more details, please view the Shader Peformance page.

Page last updated: 2007-05-08

---

## shader-ReflectiveBumpedUnlit




### Reflective Properties

This shader will simulate reflective surfaces such as cars, metal objects etc. It requires an environment Cubemap which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any scene lights will add illumination on top of what is reflected.

### Normal mapped Properties

This shader does not use normal-mapping in the traditional way. The normal map does not affect any lights shining on the object, because the shader does not use lights at all. The normal map will only distort the reflection map.

### Special Properties

This shader is special because it does not respond to lights at all, so you don't have to worry about performance reduction from use of multiple lights. It simply displays the reflection cube map on the model. The reflection is distorted by the normal map, so you get the benefit of detailed reflection. Because it does not respond to lights, it is quite cheap. It is somewhat of a specialized use case, but in those cases it does exactly what you want as cheaply as possible.

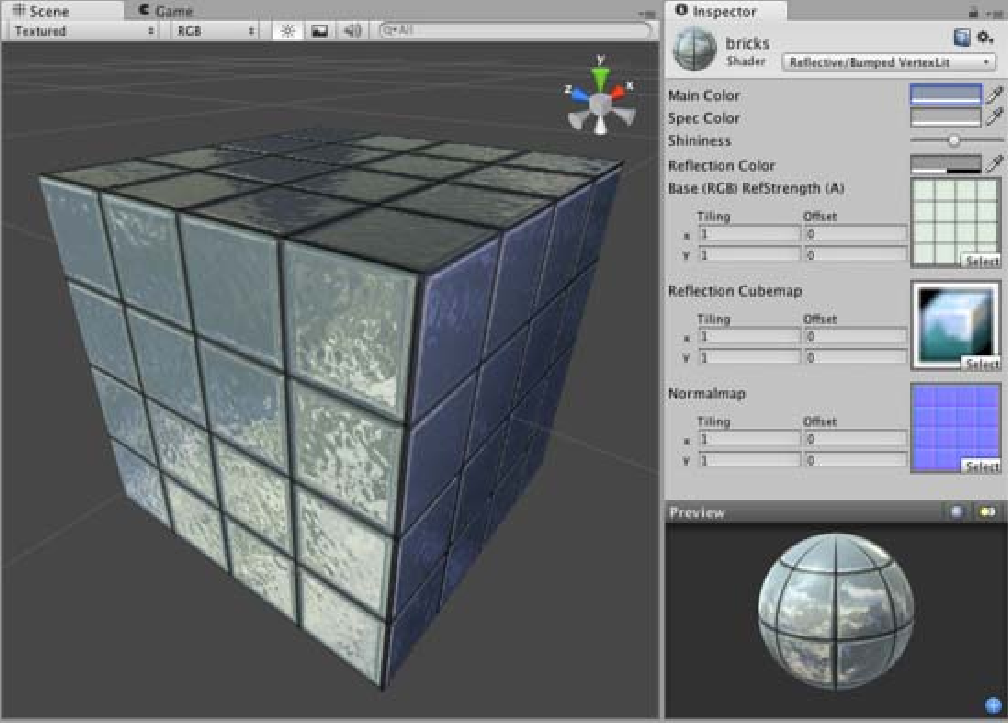
### Performance

Generally, this shader is quite cheap to render. For more details, please view the Shader Peformance page.

Page last updated: 2010-07-13

---

## shader-ReflectiveBumpedVertexLit




### Reflective Properties

This shader will simulate reflective surfaces such as cars, metal objects etc. It requires an environment Cubemap which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any scene lights will add illumination on top of what is reflected.

### Vertex-Lit Properties

This shader is **Vertex-Lit**, which is one of the simplest shaders. All lights shining on it are rendered in a single pass and calculated at vertices only.

Because it is vertex-lit, it won't display any pixel-based rendering effects, such as light cookies, normal mapping, or shadows. This shader is also much more sensitive to tesselation of the models. If you put a point light very close to a cube using this shader, the light will only be calculated at the corners. Pixel-lit shaders are much more effective at creating a nice round highlight, independent of tesselation. If that's an effect you want, you may consider using a pixel-lit shader or increase tesselation of the objects instead.

### Special Properties

This shader is a good alternative to Reflective Normal mapped. If you do not need the object itself to be affected by pixel lights, but do want the reflection to be affected by a normal map, this shader is for you. This shader is vertex-lit, so it is rendered more quickly than its Reflective Normal mapped counterpart.

### Performance

Generally, this shader is not expensive to render. For more details, please view the Shader Peformance page.

Page last updated: 2010-07-13

---

## Rendering-Tech

This section explains the technical details behind various aspects of Unity's rendering engine.

- Deferred Lighting Rendering Path
- Forward Rendering Path Details
- Vertex Lit Rendering Path Details
- Hardware Requirements for Unity's Graphics Features

Page last updated: 2011-11-08

---

# RenderTech-DeferredLighting

This page details the **Deferred Lighting** rendering path. See this article for a technical overview of deferred lighting.

The **Deferred Lighting** rendering path is the one with the highest lighting and shadow fidelity. There is no limit on the number of lights that can affect an object and all lights are evaluated per-pixel, which means that they all interact correctly with normal maps, etc. Additionally, all lights can have cookies and shadows.

Deferred lighting has the advantage that the processing overhead of lighting is proportional to the size of the light onscreen, no matter how many objects it illuminates. Therefore, performance can be improved by keeping lights small. Deferred lighting also has highly consistent and predictable behaviour. The effect of each light is computed per-pixel, so there are no lighting computations that break down on large triangles etc.

On the downside, deferred lighting has no real support for anti-aliasing and can't handle semi-transparent objects (these must be rendered using Forward Rendering). There is also no support for the Mesh Renderer's Receive Shadows flag and culling masks are only supported in a limited way.

## Requirements

Deferred lighting is only available in **Unity Pro**. It requires a graphics card with Shader Model 3.0 (or later), support for Depth render textures and two-sided stencil buffers. Most graphics cards made after 2004 support deferred lighting, including GeForce FX and later, Radeon X1300 and later, Intel 965 / GMA X3100 and later. However, it is not currently available on mobile platforms nor Flash.

## Performance Considerations

The rendering overhead of realtime lights in deferred lighting is proportional to the number of pixels illuminated by the light and *not* dependent on scene complexity. So small point or spot lights are very cheap to render and if they are fully or partially occluded by scene objects then they are even cheaper.

Of course, lights with shadows are much more expensive than lights without shadows. In deferred lighting, shadow-casting objects still need to be rendered once or more for each shadow-casting light. Furthermore, the lighting shader that applies shadows has a higher rendering overhead than the one used when shadows are disabled.

## Implementation Details

When Deferred Lighting is used, the rendering process in Unity happens in three passes:-

1. Base Pass: objects are rendered to produce screen-space buffers with depth, normals, and specular power.
2. Lighting pass: the previously generated buffers are used to compute lighting into another screen-space buffer.
3. Final pass: objects are rendered again. They fetch the computed lighting, combine it with color textures and add any ambient/emissive lighting.

Objects with shaders that can't handle deferred lighting are rendered after this process is complete, using the forward rendering path.

### Base Pass

The base pass renders each object once. View space normals and specular power are rendered into a single ARGB32 Render Texture (with normals in RGB channels and specular power in A). If the platform and hardware allow the Z buffer to be read as a texture then depth is not explicitly rendered. If the Z buffer can't be accessed as a texture then depth is rendered in an additional rendering pass using shader replacement.

The result of the base pass is a Z buffer filled with the scene contents and a Render Texture with normals and specular power.

**Lighting Pass**

The lighting pass computes lighting based on depth, normals and specular power. Lighting is computed in screen space, so the time it takes to process is independent of scene complexity. The lighting buffer is a single ARGB32 Render Texture, with diffuse lighting in the RGB channels and monochrome specular lighting in the A channel. Lighting values are logarithmically encoded to provide greater dynamic range than is usually possible with an ARGB32 texture. The only lighting model available with deferred rendering is Blinn-Phong.

Point and spot lights that do not cross the camera's near plane are rendered as 3D shapes, with the Z buffer's test against the scene enabled. This makes partially or fully occluded point and spot lights very cheap to render. Directional lights and point/spot lights that cross the near plane are rendered as fullscreen quads.

If a light has shadows enabled then they are also rendered and applied in this pass. Note that shadows do not come for "free"; shadow casters need to be rendered and a more complex light shader must be applied.

The only lighting model available is Blinn-Phong. If a different model is wanted you can modify the lighting pass shader, by placing the modified version of the Internal-PrePassLighting.shader file from the Built-in shaders into a folder named "Resources" in your "Assets" folder.

**Final Pass**

The final pass produces the final rendered image. Here all objects are rendered again with shaders that fetch the lighting, combine it with textures and add any emissive lighting. Lightmaps are also applied in the final pass. Close to the camera, realtime lighting is used, and only baked indirect lighting is added. This crossfades into fully baked lighting further away from the camera.

Page last updated: 2012-08-17

---

# RenderTech-ForwardRendering

This page describes details of **Forward** rendering path.

Forward Rendering path renders each object in one or more passes, depending on lights that affect the object. Lights themselves are also treated differently by Forward Rendering, depending on their settings and intensity.

## Implementation Details

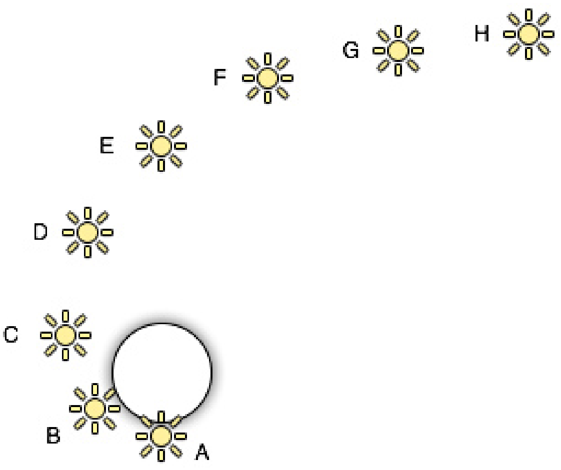
In Forward Rendering, some number of brightest lights that affect each object are rendered in fully per-pixel lit mode. Then, up to 4 point lights are calculated per-vertex. The other lights are computed as Spherical Harmonics (SH), which is much faster but is only an approximation. Whether a light will be per-pixel light or not is dependent on this:

- Lights that have their Render Mode set to **Not Important** are always per-vertex or SH.
- Brightest directional light is always per-pixel.
- Lights that have their Render Mode set to **Important** are always per-pixel.
- If the above results in less lights than current **Pixel Light Count** Quality Setting, then more lights are rendered per-pixel, in order of decreasing brightness.

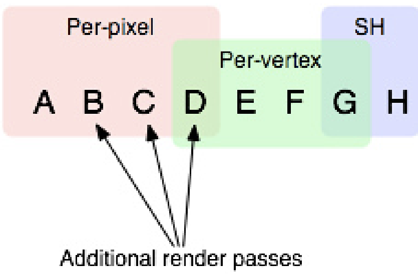
Rendering of each object happens as follows:

- Base Pass applies one per-pixel directional light and all per-vertex/SH lights.
- Other per-pixel lights are rendered in additional passes, one pass for each light.

For example, if there is some object that's affected by a number of lights (a circle in a picture below, affected by lights A to H):




Let's assume lights A to H have the same color & intensity, all all of them have Auto rendering mode, so they would be sorted in exactly this order for this object. The brightest lights will be rendered in per-pixel lit mode (A to D), then up to 4 lights in per-vertex lit mode (D to G), and finally the rest of lights in SH (G to H):




Note that light groups overlap; for example last per-pixel light blends into per-vertex lit mode so there are less "light popping" as objects and lights move around.

#### Base Pass

Base pass renders object with one per-pixel directional light and all SH lights. This pass also adds any lightmaps, ambient and emissive lighting from the shader. Directional light rendered in this pass can have Shadows. Note that Lightmapped objects do not get illumination from SH lights.

#### Additional Passes

Additional passes are rendered for each additional per-pixel light that affect this object. Lights in these passes can't have shadows (so in result, Forward Rendering supports one directional light with shadows).

### Performance Considerations

Spherical Harmonics lights are *very* fast to render. They have a tiny cost on the CPU, and are *actually free* for the GPU to apply (that is, base pass always computes SH lighting; but due to the way SH lights work, the cost is exactly the same no matter how many SH lights are there).

The downsides of SH lights are:

- They are computed at object's vertices, not pixels. This means they do not support light Cookies or normal maps.
- SH lighting is very low frequency. You can't have sharp lighting transitions with SH lights. They are also only affecting the diffuse lighting (too low frequency for specular highlights).
- SH lighting is is not local; point or spot SH lights close to some surface will "look wrong".

In summary, SH lights are often good enough for small dynamic objects.

Page last updated: 2010-07-08

---

# RenderTech-VertexLit

This page describes details of **Vertex Lit** rendering path.

Vertex Lit path generally renders each object in one pass, with lighting from all lights calculated at object vertices.

It's the fastest rendering path and has widest hardware support (however, keep in mind: it does not work on consoles).

Since all lighting is calculated at vertex level, this rendering path does not support most of per-pixel effects: shadows, normal mapping, light cookies, highly detailed specular highlights are not supported.

Page last updated: 2010-09-15

---

# RenderTech-HardwareRequirements

### Summary

| | PC/Mac | iOS/Android | Flash | 360/PS3 |
|---|---|---|---|---|
| Deferred lighting | SM3.0, GPU support | - | - | Yes |
| Forward rendering | SM2.0 | OpenGL ES 2.0 | Yes | Yes |
| Vertex Lit rendering | Yes | Yes | Yes | - |
| Realtime Shadows | SM2.0, GPU support | - | Kind-of | Yes |
| Image Effects | Most need SM2.0 | Most need OpenGL ES 2.0 | Kind-of | Yes |
| Vertex Shaders | SM1.1 | OpenGL ES 2.0 | Yes | Yes |
| Pixel Shaders | SM2.0 | OpenGL ES 2.0 | Yes | Yes |
| Fixed Function Shaders | Yes | Yes | Yes | - |

### Realtime Shadows

Realtime Shadows currently work on desktop & console platforms. On desktops, they generally need Shader Model 2.0 capable GPU. On Windows (Direct3D), the GPU also needs to support shadow mapping features; most discrete GPUs support that since 2003 and most integrated GPUs support that since 2007. Technically, on Direct3D 9 the GPU has to support D16/D24X8 or DF16/DF24 texture formats; and on OpenGL it has to support GL_ARB_depth_texture extension.

Flash does support realtime shadows, but due to lack of depth bias and shader limitations, they can have self-shadowing artifacts (increase light's shadow bias) and somewhat more "hard" edges.

Mobile shadows (iOS/Android) require OpenGL ES 2.0 and GL_OES_depth_texture extension. Most notably, the extension is **not** present on Tegra-based Android devices, so shadows do not work there.

### Image Effects

Image Effects require render-to-texture functionality, which is generally supported on anything made in this millenium. However, all except the simplest effects require quite programmable pixel shaders, so for all practical purposes they require Shader Model 2.0 on desktop (discrete GPUs since 2003; integrated GPUs since 2005) and OpenGL ES 2.0 on mobile platforms.

Some image effects work on Flash, but quite a lot of them do not; either due to no support for non-power-of-two textures, shader limitations or lacking features like depth texture support.

### Shaders

In Unity, you can write fixed function or programmable shaders. Fixed function is supported everywhere except consoles (Xbox 360 & Playstation 3). Programmable shaders default to Shader Model 2.0 (desktop) and OpenGL ES 2.0 (mobile). On desktop platforms, it is possible to target Shader Model 1.1 for vertex shaders.

Page last updated: 2012-11-16

---

## SL-Reference

Shaders in Unity can be written in one of three different ways:

- as **surface shaders**,
- as **vertex and fragment shaders** and
- as **fixed function shaders**.

The shader tutorial can guide you on choosing the right type for your needs.

Regardless of which type you choose, the actual meat of the shader code will always be wrapped in a language called ShaderLab, which is used to organize the shader structure. It looks like this:

```
Shader "MyShader" {
    Properties {
        _MyTexture ("My Texture", 2D) = "white" { }
        // other properties like colors or vectors go here as well
    }
    SubShader {
        // here goes the 'meat' of your
        //  - surface shader or
        //  - vertex and program shader or
        //  - fixed function shader
    }
    SubShader {
        // here goes a simpler version of the SubShader above than can run on older graphics cards
    }
}
```

We recommend that you start by reading about some basic concepts of the ShaderLab syntax in the sections listed below and then to move on to read about surface shaders or vertex and fragment shaders in other sections. Since fixed function shaders are written using ShaderLab only, you will find more information about them in the ShaderLab reference itself.

The reference below includes plenty of examples for the different types of shaders. For even more examples of surface shaders in particular, you can get the source of Unity's built-in shaders from the Resources section. Unity's Image Effects package contains a lot of interesting vertex and fragment shaders.

Read on for shader reference, and check out the shader tutorial as well!


- Writing Surface Shaders
  - Surface Shader Examples
  - Custom Lighting models in Surface Shaders
  - Surface Shader Lighting Examples
  - Surface Shaders with DX11 Tessellation
- Writing vertex and fragment shaders
  - Accessing shader properties in Cg
  - Providing vertex data to vertex programs
  - Built-in shader include files
  - Predefined shader preprocessor macros
  - Built-in state variables in shader programs
  - GLSL Shader Programs
- ShaderLab syntax: Shader
  - ShaderLab syntax: Properties
  - ShaderLab syntax: SubShader
    - ShaderLab syntax: Pass
      - ShaderLab syntax: Color, Material, Lighting
      - ShaderLab syntax: Culling & Depth Testing
      - ShaderLab syntax: Texture Combiners
      - ShaderLab syntax: Fog
      - ShaderLab syntax: Alpha testing

            - ShaderLab syntax: Blending
            - ShaderLab syntax: Pass Tags
            - ShaderLab syntax: Name
            - ShaderLab syntax: BindChannels
        - ShaderLab syntax: UsePass
        - ShaderLab syntax: GrabPass
        - ShaderLab syntax: SubShader Tags
    - ShaderLab syntax: Fallback
    - ShaderLab syntax: other commands
- Advanced ShaderLab topics
    - Unity's Rendering Pipeline
    - Performance Tips when Writing Shaders
    - Rendering with Replaced Shaders
    - Using Depth Textures
    - Camera's Depth Texture
    - Platform Specific Rendering Differences
    - Shader Level of Detail
- ShaderLab builtin values


Page last updated: 2011-01-14

---

# SL-SurfaceShaders

Writing shaders that interact with lighting is complex. There are different light types, different shadow options, different rendering paths (forward and deferred rendering), and the shader should somehow handle all that complexity.

**Surface Shaders** in Unity is a code generation approach that makes it much easier to write lit shaders than using low level vertex/pixel shader programs. Note that there is no custom languages, magic or ninjas involved in Surface Shaders; it just generates all the repetitive code that would have to be written by hand. You still write shader code in Cg / HLSL.

For some examples, take a look at **Surface Shader Examples** and **Surface Shader Custom Lighting Examples**.

### How it works

You define a "surface function" that takes any UVs or data you need as input, and fills in output structure `SurfaceOutput`. SurfaceOutput basically describes *properties of the surface* (it's albedo color, normal, emission, specularity etc.). You write this code in Cg / HLSL.

Surface Shader compiler then figures out what inputs are needed, what outputs are filled and so on, and generates actual vertex&pixel shaders, as well as rendering passes to handle forward and deferred rendering.

Standard output structure of surface shaders is this:

```
struct SurfaceOutput {
    half3 Albedo;
    half3 Normal;
    half3 Emission;
    half Specular;
    half Gloss;
    half Alpha;
};
```

### Samples

See **Surface Shader Examples**, **Surface Shader Custom Lighting Examples** and **Surface Shader Tessellation** pages.

### Surface Shader compile directives

Surface shader is placed inside `CGPROGRAM..ENDCG` block, just like any other shader. The differences are:

- It must be placed inside SubShader block, not inside Pass. Surface shader will compile into multiple passes itself.

- It uses `#pragma surface ...` directive to indicate it's a surface shader.

The `#pragma surface` directive is:

```
#pragma surface surfaceFunction lightModel [optionalparams]
```

Required parameters:

- surfaceFunction - which Cg function has surface shader code. The function should have the form of `void surf (Input IN, inout SurfaceOutput o)`, where Input is a structure you have defined. Input should contain any texture coordinates and extra automatic variables needed by surface function.
- lightModel - lighting model to use. Built-in ones are `Lambert` (diffuse) and `BlinnPhong` (specular). See Custom Lighting Models page for how to write your own.

Optional parameters:

- `alpha` - Alpha blending mode. Use this for semitransparent shaders.
- `alphatest:VariableName` - Alpha testing mode. Use this for transparent-cutout shaders. Cutoff value is in float variable with VariableName.
- `vertex:VertexFunction` - Custom vertex modification function. See Tree Bark shader for example.
- `finalcolor:ColorFunction` - Custom final color modification function. See Surface Shader Examples.
- `exclude_path:prepass` or `exclude_path:forward` - Do not generate passes for given rendering path.
- `addshadow` - Add shadow caster & collector passes. Commonly used with custom vertex modification, so that shadow casting also gets any procedural vertex animation.
- `dualforward` - Use dual lightmaps in forward rendering path.
- `fullforwardshadows` - Support all shadow types in Forward rendering path.
- `decal:add` - Additive decal shader (e.g. terrain AddPass).
- `decal:blend` - Semitransparent decal shader.
- `softvegetation` - Makes the surface shader only be rendered when Soft Vegetation is on.
- `noambient` - Do not apply any ambient lighting or spherical harmonics lights.
- `novertexlights` - Do not apply any spherical harmonics or per-vertex lights in Forward rendering.
- `nolightmap` - Disables lightmap support in this shader (makes a shader smaller).
- `nodirlightmap` - Disables directional lightmaps support in this shader (makes a shader smaller).
- `noforwardadd` - Disables Forward rendering additive pass. This makes the shader support one full directional light, with all other lights computed per-vertex/SH. Makes shaders smaller as well.
- `approxview` - Computes normalized view direction per-vertex instead of per-pixel, for shaders that need it. This is faster, but view direction is not entirely correct when camera gets close to surface.
- `halfasview` - Pass half-direction vector into the lighting function instead of view-direction. Half-direction will be computed and normalized per vertex. This is faster, but not entirely correct.
- `tessellate:TessFunction` - use DX11 GPU tessellation; the function computes tessellation factors. See Surface Shader Tessellation for details.

Additionally, you can write `#pragma debug` inside CGPROGRAM block, and then surface compiler will spit out a lot of comments of the generated code. You can view that using Open Compiled Shader in shader inspector.

### Surface Shader input structure

The input structure `Input` generally has any texture coordinates needed by the shader. Texture coordinates must be named "uv" followed by texture name (or start it with "uv2" to use second texture coordinate set).

Additional values that can be put into Input structure:

- `float3 viewDir` - will contain view direction, for computing Parallax effects, rim lighting etc.
- `float4` with `COLOR` semantic - will contain interpolated per-vertex color.
- `float4 screenPos` - will contain screen space position for reflection effects. Used by WetStreet shader in Dark Unity for example.
- `float3 worldPos` - will contain world space position.
- `float3 worldRefl` - will contain world reflection vector *if surface shader does not write to o.Normal*. See Reflect-Diffuse shader for example.
- `float3 worldNormal` - will contain world normal vector *if surface shader does not write to o.Normal*.
- `float3 worldRefl; INTERNAL_DATA` - will contain world reflection vector *if surface shader writes to o.Normal*. To

get the reflection vector based on per-pixel normal map, use `WorldReflectionVector (IN, o.Normal)`. See Reflect-Bumped shader for example.

- `float3 worldNormal; INTERNAL_DATA` - will contain world normal vector *if surface shader writes to o.Normal*. To get the normal vector based on per-pixel normal map, use `WorldNormalVector (IN, o.Normal)`.

### Further Documentation

- Surface Shader Examples
- Custom Lighting models in Surface Shaders
- Surface Shader Lighting Examples
- Surface Shaders with DX11 Tessellation

Page last updated: 2012-11-16

---

## SL-SurfaceShaderExamples

Here are some examples of **Surface Shaders**. The examples below focus on using built-in lighting models; examples on how to implement custom lighting models are in Surface Shader Lighting Examples.

### Simple

We'll start with a very simple shader and build up on that. Here's a shader that just sets surface color to "white". It uses built-in Lambert (diffuse) lighting model.

```
Shader "Example/Diffuse Simple" {
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
        float4 color : COLOR;
    };
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = 1;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

Here's how it looks like on a model with two lights set up:

### Texture

An all-white object is quite boring, so let's add a texture. We'll add a Properties block to the shader, so we get a texture selector in our Material. Other changes are in bold below.

```
Shader "Example/Diffuse Texture" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
        float2 uv_MainTex;
    };
    sampler2D _MainTex;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

### Normal mapping

Let's add some normal mapping:

```
Shader "Example/Diffuse Bump" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _BumpMap ("Bumpmap", 2D) = "bump" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
      float2 uv_MainTex;
      float2 uv_BumpMap;
    };
    sampler2D _MainTex;
    sampler2D _BumpMap;
    void surf (Input IN, inout SurfaceOutput o) {
      o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
      o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

### Rim Lighting

Now, try to add some Rim Lighting to highlight the edges of an object. We'll add some emissive light based on angle between surface normal and view direction. For that, we'll use `viewDir` built-in surface shader variable.

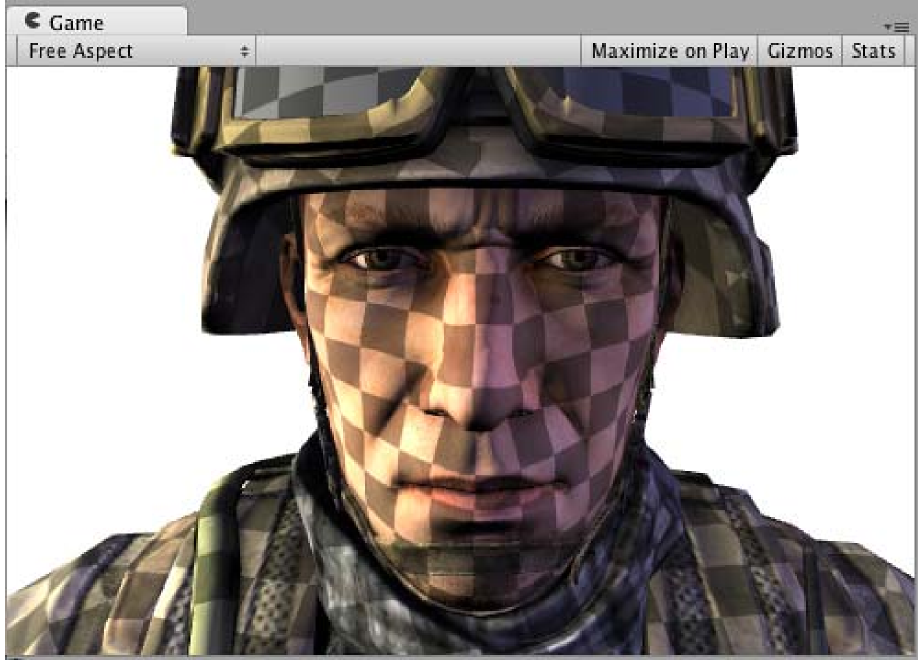
```
Shader "Example/Rim" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _BumpMap ("Bumpmap", 2D) = "bump" {}
    _RimColor ("Rim Color", Color) = (0.26,0.19,0.16,0.0)
    _RimPower ("Rim Power", Range(0.5,8.0)) = 3.0
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
        float2 uv_MainTex;
        float2 uv_BumpMap;
        float3 viewDir;
    };
    sampler2D _MainTex;
    sampler2D _BumpMap;
    float4 _RimColor;
    float _RimPower;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
        o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
        half rim = 1.0 - saturate(dot (normalize(IN.viewDir), o.Normal));
        o.Emission = _RimColor.rgb * pow (rim, _RimPower);
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

### Detail Texture

For a different effect, let's add a detail texture that is combined with the base texture. Detail texture uses the same UVs, but usually different Tiling in the Material, so we have to use different input UV coordinates.

```
Shader "Example/Detail" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _BumpMap ("Bumpmap", 2D) = "bump" {}
    _Detail ("Detail", 2D) = "gray" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
        float2 uv_MainTex;
        float2 uv_BumpMap;
        float2 uv_Detail;
    };
    sampler2D _MainTex;
    sampler2D _BumpMap;
    sampler2D _Detail;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
        o.Albedo *= tex2D (_Detail, IN.uv_Detail).rgb * 2;
        o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

Using a checker texture does not make much practical sense, but illustrates what happens:

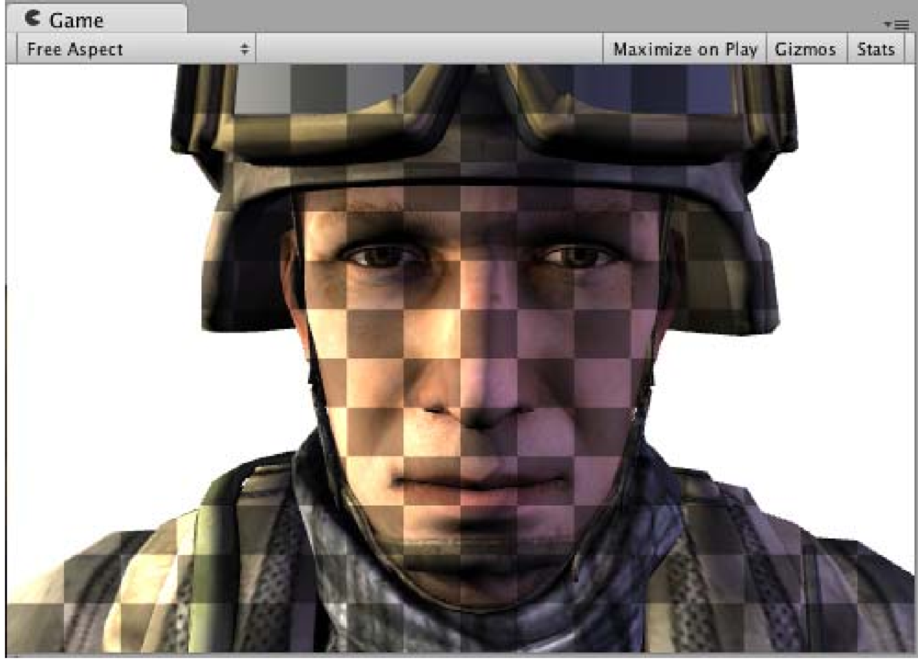



### Detail Texture in Screen Space

How about a detail texture in screen space? It does not make much sense for a soldier head model, but illustrates how a built-in `screenPos` input might be used:

```
Shader "Example/ScreenPos" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _Detail ("Detail", 2D) = "gray" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
        float2 uv_MainTex;
        float4 screenPos;
    };
    sampler2D _MainTex;
    sampler2D _Detail;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
        float2 screenUV = IN.screenPos.xy / IN.screenPos.w;
        screenUV *= float2(8,6);
        o.Albedo *= tex2D (_Detail, screenUV).rgb * 2;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

I removed normal mapping from the shader above, just to make it shorter:




### Cubemap Reflection

Here's a shader that does cubemapped reflection using built-in `worldRefl` input. It's actually very similar to built-in Reflective/Diffuse shader:

```
Shader "Example/WorldRefl" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _Cube ("Cubemap", CUBE) = "" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
        float2 uv_MainTex;
        float3 worldRefl;
    };
    sampler2D _MainTex;
    samplerCUBE _Cube;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb * 0.5;
        o.Emission = texCUBE (_Cube, IN.worldRefl).rgb;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

And since it assigns the reflection color as Emission, we get a very shiny soldier:

If you want to do reflections that are affected by normal maps, it needs to be slightly more involved: `INTERNAL_DATA` needs to be added to the Input structure, and `WorldReflectionVector` function used to compute per-pixel reflection vector after you've written the Normal output.

```
Shader "Example/WorldRefl Normalmap" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _BumpMap ("Bumpmap", 2D) = "bump" {}
    _Cube ("Cubemap", CUBE) = "" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
        float2 uv_MainTex;
        float2 uv_BumpMap;
        float3 worldRefl;
        INTERNAL_DATA
    };
    sampler2D _MainTex;
    sampler2D _BumpMap;
    samplerCUBE _Cube;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb * 0.5;
        o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
        o.Emission = texCUBE (_Cube, WorldReflectionVector (IN, o.Normal)).rgb;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

Here's a normal mapped shiny soldier:

### Slices via World Space Position

Here's a shader that "slices" the object by discarding pixels in nearly horizontal rings. It does that by using `clip()` Cg/HLSL function based on world position of a pixel. We'll use `worldPos` built-in surface shader variable.

```
Shader "Example/Slices" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _BumpMap ("Bumpmap", 2D) = "bump" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    Cull Off
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
        float2 uv_MainTex;
        float2 uv_BumpMap;
        float3 worldPos;
    };
    sampler2D _MainTex;
    sampler2D _BumpMap;
    void surf (Input IN, inout SurfaceOutput o) {
        clip (frac((IN.worldPos.y+IN.worldPos.z*0.1) * 5) - 0.5);
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
        o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

### Normal Extrusion with Vertex Modifier

It is possible to use a "vertex modifier" function that will modify incoming vertex data in the vertex shader. This can be used for procedural animation, extrusion along normals and so on. Surface shader compilation directive `vertex:functionName` is used for that, with a function that takes `inout appdata_full` parameter.

Here's a shader that moves vertices along their normals by the amount specified in the material:

```
Shader "Example/Normal Extrusion" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _Amount ("Extrusion Amount", Range(-1,1)) = 0.5
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert vertex:vert
    struct Input {
        float2 uv_MainTex;
    };
    float _Amount;
    void vert (inout appdata_full v) {
        v.vertex.xyz += v.normal * _Amount;
    }
    sampler2D _MainTex;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

Moving vertices along their normals makes a fat soldier:

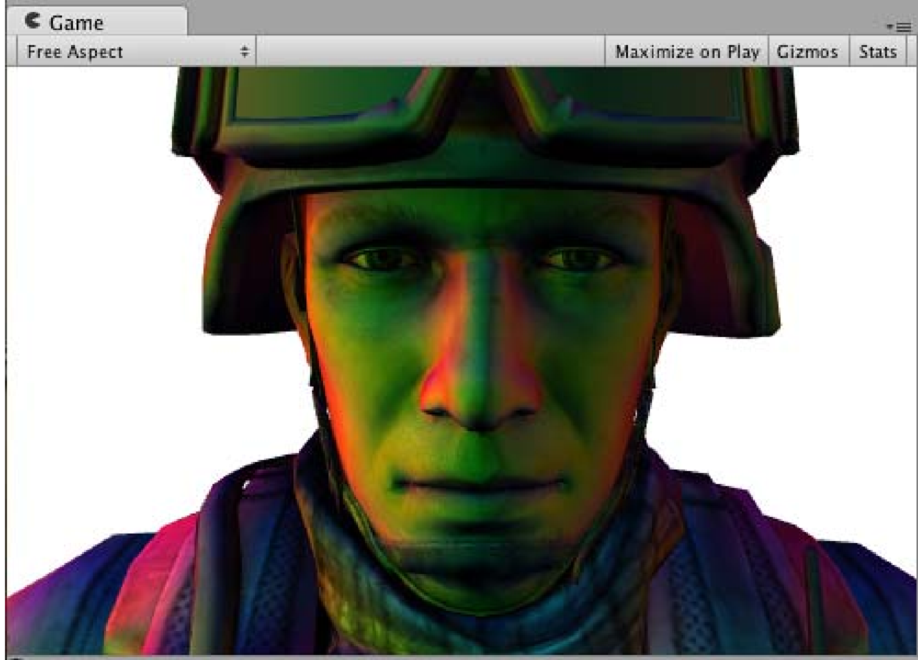
### Custom data computed per-vertex

Using a vertex modifier function it is also possible to compute custom data in a vertex shader, which then will be passed to the surface shader function per-pixel. The same compilation directive `vertex:functionName` is used, but the function should take two parameters: `inout appdata_full` and `out Input`. You can fill in any Input member that is not a built-in value there.

Example below defines a custom `float3 customColor` member, which is computed in a vertex function:

```
Shader "Example/Custom Vertex Data" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert vertex:vert
    struct Input {
        float2 uv_MainTex;
        float3 customColor;
    };
    void vert (inout appdata_full v, out Input o) {
        UNITY_INITIALIZE_OUTPUT(Input,o);
        o.customColor = abs(v.normal);
    }
    sampler2D _MainTex;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
        o.Albedo *= IN.customColor;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

In this example `customColor` is set to the absolute value of the normal:




More practical uses could be computing any per-vertex data that is not provided by built-in Input variables; or optimizing shader computations. For example, it's possible to compute Rim lighting at object's vertices, instead of doing that in the surface shader per-pixel.

### Final Color Modifier

It is possible to use a "final color modifier" function that will modify final color computed by the shader. Surface shader compilation directive `finalcolor:functionName` is used for that, with a function that takes `Input IN, SurfaceOutput o, inout fixed4 color` parameters.

Here's a simple shader that applies tint to final color. This is different from just applying tint to surface Albedo color: this tint will also affect any color that came from lightmaps, light probes and similar extra sources.

```
Shader "Example/Tint Final Color" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _ColorTint ("Tint", Color) = (1.0, 0.6, 0.6, 1.0)
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert finalcolor:mycolor
    struct Input {
        float2 uv_MainTex;
    };
    fixed4 _ColorTint;
    void mycolor (Input IN, SurfaceOutput o, inout fixed4 color)
    {
        color *= _ColorTint;
    }
    sampler2D _MainTex;
    void surf (Input IN, inout SurfaceOutput o) {
         o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

### Custom Fog with Final Color Modifier

Common use case for final color modifier (see above) would be implementing completely custom Fog. Fog needs to affect the final computed pixel shader color, which is exactly what the `finalcolor` modifier does.

Here's a shader that applies fog tint based on distance from screen center. This combines both the vertex modifier with custom vertex data (`fog`) and final color modifier. When used in forward rendering additive pass, Fog needs to fade to black color, and this example handles that as well with a check for `UNITY_PASS_FORWARDADD`.

```
Shader "Example/Fog via Final Color" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _FogColor ("Fog Color", Color) = (0.3, 0.4, 0.7, 1.0)
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert finalcolor:mycolor vertex:myvert
    struct Input {
        float2 uv_MainTex;
        half fog;
    };
    void myvert (inout appdata_full v, out Input data)
    {
        UNITY_INITIALIZE_OUTPUT(Input,v);
        float4 hpos = mul (UNITY_MATRIX_MVP, v.vertex);
        data.fog = min (1, dot (hpos.xy, hpos.xy) * 0.1);
    }
    fixed4 _FogColor;
    void mycolor (Input IN, SurfaceOutput o, inout fixed4 color)
    {
        fixed3 fogColor = _FogColor.rgb;
        #ifdef UNITY_PASS_FORWARDADD
        fogColor = 0;
        #endif
        color.rgb = lerp (color.rgb, fogColor, IN.fog);
    }
    sampler2D _MainTex;
    void surf (Input IN, inout SurfaceOutput o) {
         o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
```

```
        }
        ENDCG
    }
    Fallback "Diffuse"
}
```

Page last updated: 2012-11-16

---

# SL-SurfaceShaderLighting

When writing Surface Shaders, you're describing properties of a surface (albedo color, normal, ...) and the lighting interaction is computed by a **Lighting Model**. Built-in lighting models are **Lambert** (diffuse lighting) and **BlinnPhong** (specular lighting).

Sometimes you might want to use a custom lighting model, and it is possible to do that in Surface Shaders. Lighting model is nothing more than a couple of Cg/HLSL functions that match some conventions. The built-in `Lambert` and `BlinnPhong` models are defined in `Lighting.cginc` file inside Unity (**{unity install path}/Data/CGIncludes/Lighting.cginc** on Windows, **/Applications/Unity/Unity.app/Contents/CGIncludes/Lighting.cginc** on Mac).

## Lighting Model declaration

Lighting model is a couple of regular functions with names starting with `Lighting`. They can be declared anywhere in your shader file or one of included files. The functions are:

1. `half4 LightingName (SurfaceOutput s, half3 lightDir, half atten);` This is used in forward rendering path for light models that *are not* view direction dependent (e.g. diffuse).
2. `half4 LightingName (SurfaceOutput s, half3 lightDir, half3 viewDir, half atten);` This is used in forward rendering path for light models that are view direction dependent.
3. `half4 LightingName_PrePass (SurfaceOutput s, half4 light);` This is used in deferred lighting path.

Note that you don't need to declare all functions. A lighting model either uses view direction or it does not. Similarly, if the lighting model will not work in deferred lighting, you just do not declare `_PrePass` function, and all shaders that use it will compile to forward rendering only.

Decoding directional lightmaps needs to be customized in some circumstances in a similar fashion as the lighting function for forward and deferred lighting. Use one of the functions below depending on whether your light model is view direction dependent or not. Both functions handle forward and deferred lighting rendering paths automatically.

1. `half4 LightingName_DirLightmap (SurfaceOutput s, fixed4 color, fixed4 scale, bool`

`surfFuncWritesNormal);` This is used for light models that are not view direction dependent (e.g. diffuse).

2. `half4 LightingName_DirLightmap (SurfaceOutput s, fixed4 color, fixed4 scale, half3 viewDir, bool surfFuncWritesNormal, out half3 specColor);` This is used for light models that are view direction dependent.

### Examples

**Surface Shader Lighting Examples**

Page last updated: 2012-01-23

---

## SL-SurfaceShaderLightingExamples

Here are some examples of **custom lighting models** in Surface Shaders. General Surface Shader examples are in this page.

Because Deferred Lighting does not play well with some custom per-material lighting models, in most examples below we make the shaders compile to Forward Rendering only.

### Diffuse

We'll start with a shader that uses built-in Lambert lighting model:

```
Shader "Example/Diffuse Texture" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
        float2 uv_MainTex;
    };
    sampler2D _MainTex;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

Here's how it looks like with a texture and without an actual texture (one directional light is in the scene):

Now, let's do exactly the same, but write out *our own lighting model* instead of using built-in Lambert one. Surface Shader Lighting Models are just some functions that we need to write. Here's a simple Lambert one. Note that the "shader part" itself did not change at all (grayed out):

```
Shader "Example/Diffuse Texture" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf SimpleLambert

    half4 LightingSimpleLambert (SurfaceOutput s, half3 lightDir, half atten) {
        half NdotL = dot (s.Normal, lightDir);
        half4 c;
        c.rgb = s.Albedo * _LightColor0.rgb * (NdotL * atten * 2);
        c.a = s.Alpha;
```

```
            return c;
        }

        struct Input {
            float2 uv_MainTex;
        };
        sampler2D _MainTex;
        void surf (Input IN, inout SurfaceOutput o) {
            o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
        }
        ENDCG
      }
      Fallback "Diffuse"
    }
```

So our simple Diffuse lighting model is `LightingSimpleLambert` function. It computes lighting by doing a dot product between surface normal and light direction, and then applies light attenuation and color.

### Diffuse Wrap

Here's Wrapped Diffuse - a modification of Diffuse lighting, where illumination "wraps around" the edges of objects. It's useful for faking subsurface scattering effect. Again, the surface shader itself did not change at all, we're just using different lighting function.

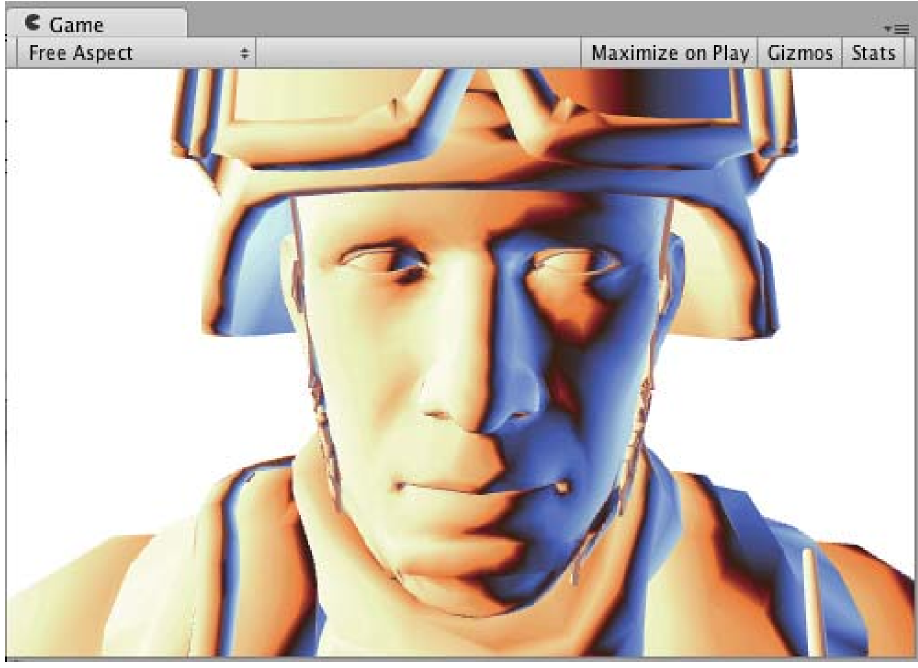
```
    Shader "Example/Diffuse Wrapped" {
      Properties {
        _MainTex ("Texture", 2D) = "white" {}
      }
      SubShader {
        Tags { "RenderType" = "Opaque" }
        CGPROGRAM
        #pragma surface surf WrapLambert

        half4 LightingWrapLambert (SurfaceOutput s, half3 lightDir, half atten) {
            half NdotL = dot (s.Normal, lightDir);
            half diff = NdotL * 0.5 + 0.5;
            half4 c;
            c.rgb = s.Albedo * _LightColor0.rgb * (diff * atten * 2);
            c.a = s.Alpha;
            return c;
        }

        struct Input {
            float2 uv_MainTex;
        };
        sampler2D _MainTex;
        void surf (Input IN, inout SurfaceOutput o) {
            o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
        }
        ENDCG
      }
      Fallback "Diffuse"
    }
```

### Toon Ramp

Here's a "Ramp" lighting model that uses a texture ramp to define how surface responds to angle between light and the normal. This can be used for variety of effects, including Toon lighting.

```
Shader "Example/Toon Ramp" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _Ramp ("Shading Ramp", 2D) = "gray" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Ramp

    sampler2D _Ramp;

    half4 LightingRamp (SurfaceOutput s, half3 lightDir, half atten) {
        half NdotL = dot (s.Normal, lightDir);
```

```
            half diff = NdotL * 0.5 + 0.5;
            half3 ramp = tex2D (_Ramp, float2(diff)).rgb;
            half4 c;
            c.rgb = s.Albedo * _LightColor0.rgb * ramp * (atten * 2);
            c.a = s.Alpha;
            return c;
        }

        struct Input {
            float2 uv_MainTex;
        };
        sampler2D _MainTex;
        void surf (Input IN, inout SurfaceOutput o) {
            o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
        }
        ENDCG
    }
    Fallback "Diffuse"
}
```




### Simple Specular

Here's a simple specular lighting model. It's quite simple to what built-in BlinnPhong actually does; we just put it here to illustrate how it's done.

```
Shader "Example/Simple Specular" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf SimpleSpecular

    half4 LightingSimpleSpecular (SurfaceOutput s, half3 lightDir, half3 viewDir, half atten)
        half3 h = normalize (lightDir + viewDir);

        half diff = max (0, dot (s.Normal, lightDir));

        float nh = max (0, dot (s.Normal, h));
        float spec = pow (nh, 48.0);

        half4 c;
        c.rgb = (s.Albedo * _LightColor0.rgb * diff + _LightColor0.rgb * spec) * (atten * 2);
        c.a = s.Alpha;
        return c;
    }

    struct Input {
        float2 uv_MainTex;
    };
    sampler2D _MainTex;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

Page last updated: 2010-09-14

---

# SL-SurfaceShaderTessellation

Surface Shaders have some support for DirectX 11 GPU Tessellation. Idea is:

- Tessellation is indicated by `tessellate:FunctionName` modifier. That function computes triangle edge and inside tessellation factors.
- When tessellation is used, "vertex modifier" (`vertex:FunctionName`) is invoked *after* tessellation, for each generated vertex in the domain shader. Here you'd typically to displacement mapping.
- Surface shaders can optionally compute phong tessellation to smooth model surface even without any displacement mapping.

Current limitations of tessellation support:

- Only triangle domain - no quads, no isoline tessellation.

- When tessellation is used, shader is automatically compiled into Shader Model 5.0 target, which means it will only work on DX11.

### No GPU tessellation, displacement in the vertex modifier

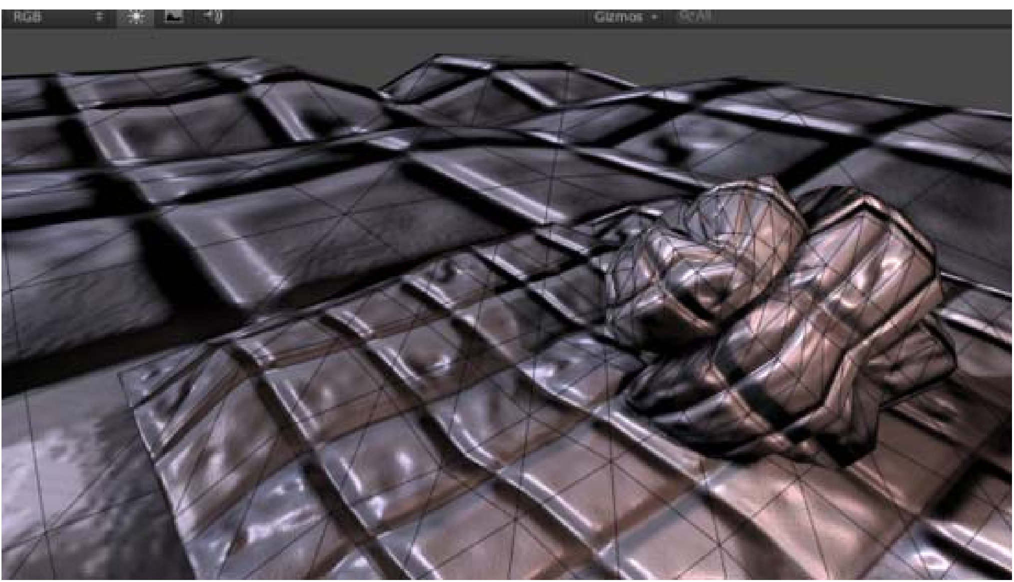
Let's start with a surface shader that does some displacement mapping *without* using tessellation. It just moves vertices along their normals based on amount coming from a displacement map:

```
Shader "Tessellation Sample" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _DispTex ("Disp Texture", 2D) = "gray" {}
        _NormalMap ("Normalmap", 2D) = "bump" {}
        _Displacement ("Displacement", Range(0, 1.0)) = 0.3
        _Color ("Color", color) = (1,1,1,0)
        _SpecColor ("Spec color", color) = (0.5,0.5,0.5,0.5)
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM
        #pragma surface surf BlinnPhong addshadow fullforwardshadows vertex:disp nolightmap
        #pragma target 5.0

        struct appdata {
            float4 vertex : POSITION;
            float4 tangent : TANGENT;
            float3 normal : NORMAL;
            float2 texcoord : TEXCOORD0;
        };

        sampler2D _DispTex;
        float _Displacement;

        void disp (inout appdata v)
        {
            float d = tex2Dlod(_DispTex, float4(v.texcoord.xy,0,0)).r * _Displacement;
            v.vertex.xyz += v.normal * d;
        }

        struct Input {
            float2 uv_MainTex;
        };

        sampler2D _MainTex;
        sampler2D _NormalMap;
        fixed4 _Color;

        void surf (Input IN, inout SurfaceOutput o) {
            half4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Specular = 0.2;
            o.Gloss = 1.0;
            o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_MainTex));
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

The above shader is fairly standard, points of intetest:

- Vertex modifier `disp` samples the displacement map and moves vertices along their normals.
- It uses custom "vertex data input" structure (`appdata`) instead of default `appdata_full`. This is not needed yet, but it's more efficient for tessellation to use as small structure as possible.
- Since our vertex data does not have 2nd UV coordinate, we add `nolightmap` directive to exclude lightmaps.

Here's how some simple objects would look like with this shader:



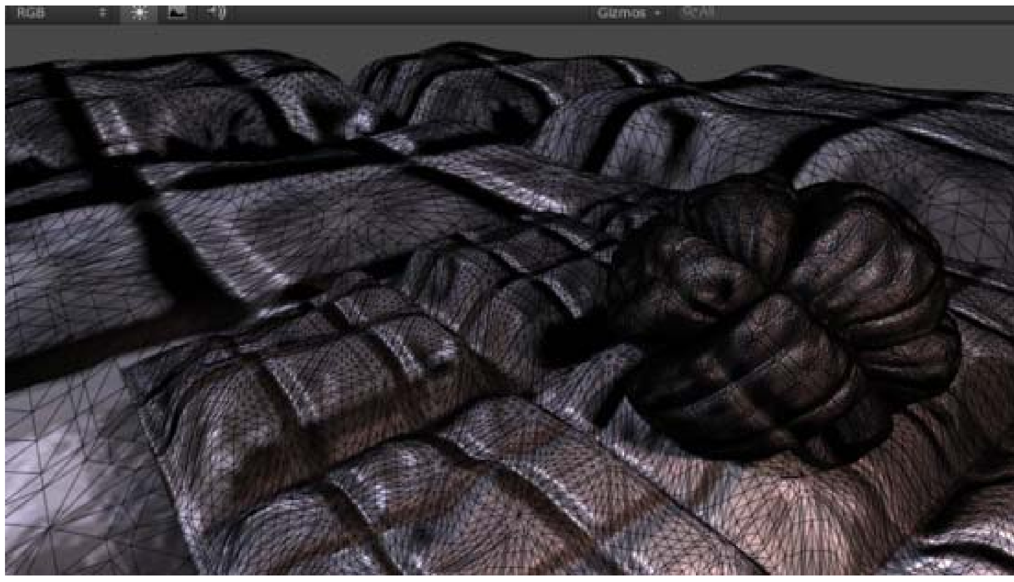

### Fixed amount of tessellation

Let's add fixed amount of tessellation, i.e. the same tessellation level for the whole mesh. This approach is suitable if your model's faces are roughly the same size on screen. Some script could then change the tessellation level from code, based on distance to the camera.

```
Shader "Tessellation Sample" {
    Properties {
        _Tess ("Tessellation", Range(1,32)) = 4
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _DispTex ("Disp Texture", 2D) = "gray" {}
        _NormalMap ("Normalmap", 2D) = "bump" {}
        _Displacement ("Displacement", Range(0, 1.0)) = 0.3
        _Color ("Color", color) = (1,1,1,0)
        _SpecColor ("Spec color", color) = (0.5,0.5,0.5,0.5)
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM
        #pragma surface surf BlinnPhong addshadow fullforwardshadows vertex:disp tessellate
        #pragma target 5.0

        struct appdata {
            float4 vertex : POSITION;
            float4 tangent : TANGENT;
            float3 normal : NORMAL;
            float2 texcoord : TEXCOORD0;
        };

        float _Tess;

        float4 tessFixed()
        {
            return _Tess;
```

```
        }

        sampler2D _DispTex;
        float _Displacement;

        void disp (inout appdata v)
        {
            float d = tex2Dlod(_DispTex, float4(v.texcoord.xy,0,0)).r * _Displacement;
            v.vertex.xyz += v.normal * d;
        }

        struct Input {
            float2 uv_MainTex;
        };

        sampler2D _MainTex;
        sampler2D _NormalMap;
        fixed4 _Color;

        void surf (Input IN, inout SurfaceOutput o) {
            half4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Specular = 0.2;
            o.Gloss = 1.0;
            o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_MainTex));
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

The tessellation function, `tessFixed` in our shader, returns four tessellation factors as a single float4 value: tree factors for each edge of the triangle, and one factor for the inside of the triangle. Here, we just return a constant value that is set in material properties.



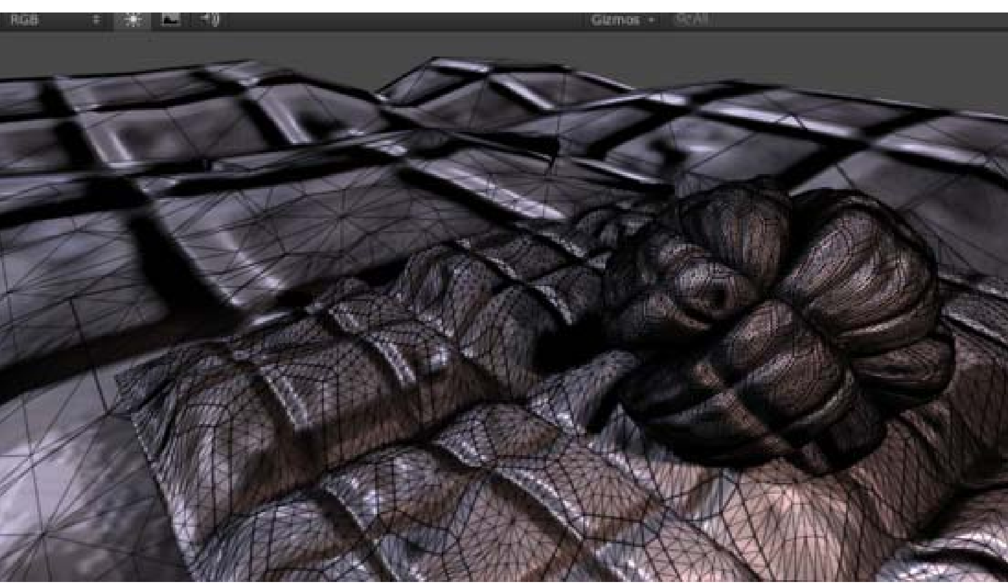

### Distance-based tessellation

We can also change tessellation level based on distance from the camera. For example, we could define two distance values; distance at which tessellation is at maximum (say, 10 meters), and distance towards which tessellation level gradually decreases (say, 20 meters).

```
Shader "Tessellation Sample" {
    Properties {
```

```
        _Tess ("Tessellation", Range(1,32)) = 4
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _DispTex ("Disp Texture", 2D) = "gray" {}
        _NormalMap ("Normalmap", 2D) = "bump" {}
        _Displacement ("Displacement", Range(0, 1.0)) = 0.3
        _Color ("Color", color) = (1,1,1,0)
        _SpecColor ("Spec color", color) = (0.5,0.5,0.5,0.5)
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM
        #pragma surface surf BlinnPhong addshadow fullforwardshadows vertex:disp tessellate
        #pragma target 5.0
        #include "Tessellation.cginc"

        struct appdata {
            float4 vertex : POSITION;
            float4 tangent : TANGENT;
            float3 normal : NORMAL;
            float2 texcoord : TEXCOORD0;
        };

        float _Tess;

        float4 tessDistance (appdata v0, appdata v1, appdata v2) {
            float minDist = 10.0;
            float maxDist = 25.0;
            return UnityDistanceBasedTess(v0.vertex, v1.vertex, v2.vertex, minDist, m
        }

        sampler2D _DispTex;
        float _Displacement;

        void disp (inout appdata v)
        {
            float d = tex2Dlod(_DispTex, float4(v.texcoord.xy,0,0)).r * _Displacement;
            v.vertex.xyz += v.normal * d;
        }

        struct Input {
            float2 uv_MainTex;
        };

        sampler2D _MainTex;
        sampler2D _NormalMap;
        fixed4 _Color;

        void surf (Input IN, inout SurfaceOutput o) {
            half4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Specular = 0.2;
            o.Gloss = 1.0;
            o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_MainTex));
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

Here the tessellation function takes three parameters; the vertex data of three triangle corners before tessellation. This is needed to compute tessellation levels, which depend on vertex positions now. We include a built-in helper file `Tessellation.cginc` and call `UnityDistanceBasedTess` function from it to do all the work. That function computes distance of each vertex to the camera and derives final tessellation factors.



### Edge length based tessellation

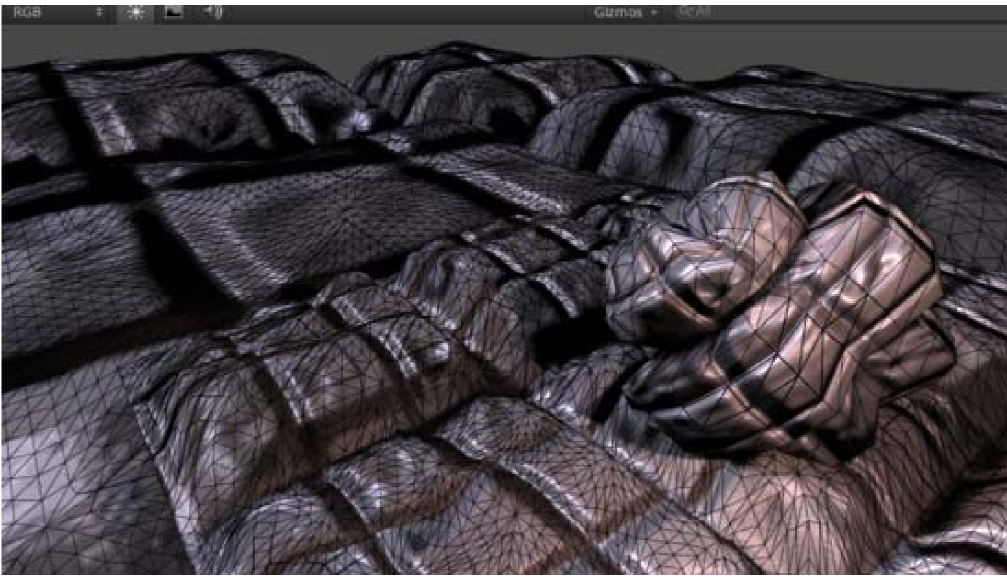
Purely distance based tessellation is good only when triangle sizes are quite similar. In the image above, you can see that objects that have small triangles are tessellated too much, while objects that have large triangles aren't tessellated enough.

Instead, tessellation levels could be computed based on triangle edge length on the screen - the longer the edge, the larger tessellation factor should be applied.

```
Shader "Tessellation Sample" {
    Properties {
        _EdgeLength ("Edge length", Range(2,50)) = 15
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _DispTex ("Disp Texture", 2D) = "gray" {}
        _NormalMap ("Normalmap", 2D) = "bump" {}
        _Displacement ("Displacement", Range(0, 1.0)) = 0.3
        _Color ("Color", color) = (1,1,1,0)
        _SpecColor ("Spec color", color) = (0.5,0.5,0.5,0.5)
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM
        #pragma surface surf BlinnPhong addshadow fullforwardshadows vertex:disp tesselate
        #pragma target 5.0
        #include "Tessellation.cginc"

        struct appdata {
            float4 vertex : POSITION;
            float4 tangent : TANGENT;
            float3 normal : NORMAL;
            float2 texcoord : TEXCOORD0;
        };

        float _EdgeLength;

        float4 tessEdge (appdata v0, appdata v1, appdata v2)
        {
```

```
            return UnityEdgeLengthBasedTess (v0.vertex, v1.vertex, v2.vertex, _EdgeLe
        }

        sampler2D _DispTex;
        float _Displacement;

        void disp (inout appdata v)
        {
            float d = tex2Dlod(_DispTex, float4(v.texcoord.xy,0,0)).r * _Displacement;
            v.vertex.xyz += v.normal * d;
        }

        struct Input {
            float2 uv_MainTex;
        };

        sampler2D _MainTex;
        sampler2D _NormalMap;
        fixed4 _Color;

        void surf (Input IN, inout SurfaceOutput o) {
            half4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Specular = 0.2;
            o.Gloss = 1.0;
            o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_MainTex));
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

Here again, we just call `UnityEdgeLengthBasedTess` function from `Tessellation.cginc` to do all the actual work.




For performance reasons, it's advisable to call `UnityEdgeLengthBasedTessCull` function instead, which will do patch frustum culling. This makes the shader a bit more expensive, but saves a lot of GPU work for parts of meshes that are outside of camera's view.

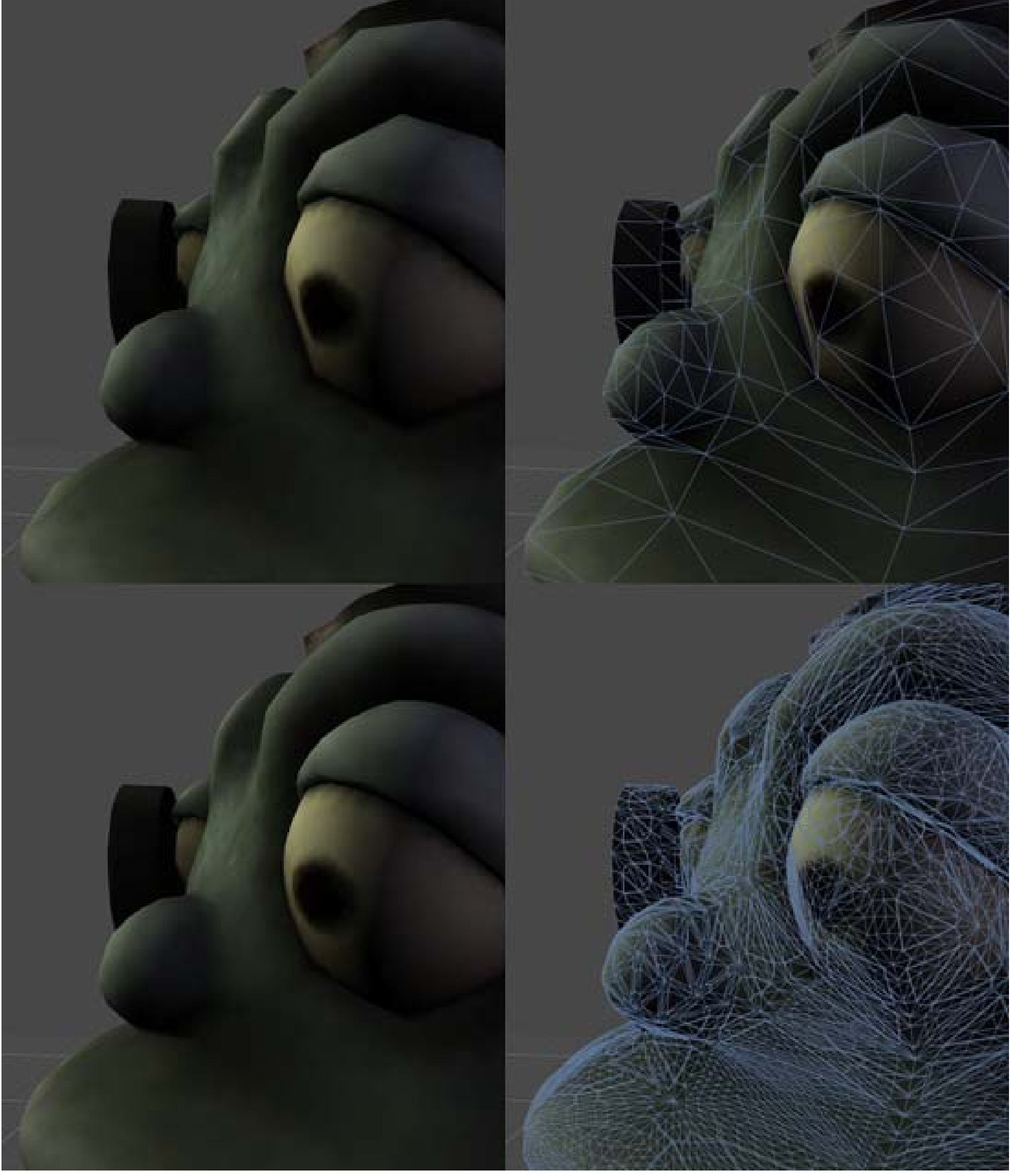
### Phong Tessellation

Phong Tessellation modifies positions of the subdivided faces so that the resulting surface follows the mesh normals a bit. It's quite an effective way of making low-poly meshes become more smooth.

Unity's surface shaders can compute Phong tessellation automatically using `tessphong:VariableName` compilation directive. Here's an example shader:

```
Shader "Phong Tessellation" {
    Properties {
        _EdgeLength ("Edge length", Range(2,50)) = 5
        _Phong ("Phong Strengh", Range(0,1)) = 0.5
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _Color ("Color", color) = (1,1,1,0)
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM
        #pragma surface surf Lambert vertex:dispNone tessellate:tessEdge tessphong:_Phong
        #include "Tessellation.cginc"

        struct appdata {
            float4 vertex : POSITION;
            float3 normal : NORMAL;
            float2 texcoord : TEXCOORD0;
        };

        void dispNone (inout appdata v) { }

        float _Phong;
        float _EdgeLength;

        float4 tessEdge (appdata v0, appdata v1, appdata v2)
        {
            return UnityEdgeLengthBasedTess (v0.vertex, v1.vertex, v2.vertex, _EdgeLength);
        }

        struct Input {
            float2 uv_MainTex;
        };

        fixed4 _Color;
        sampler2D _MainTex;

        void surf (Input IN, inout SurfaceOutput o) {
            half4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Alpha = c.a;
        }

        ENDCG
    }
    FallBack "Diffuse"
}
```

Here's a comparison between regular shader (top row) and one that uses Phong tessellation (bottom row). You can see that even without any displacement mapping, the surface becomes more round.



Page last updated: 2012-09-18

---

## SL-ShaderPrograms

**ShaderLab** shaders encompass more than just "hardware shaders". They do many things. They describe properties that are displayed in the Material Inspector, contain multiple shader implementations for different graphics hardware, configure fixed function hardware state and so on. The actual programmable shaders - like vertex and fragment programs - are just a part of the whole ShaderLab's "shader" concept. Take a look at shader tutorial for a basic introduction. Here we'll call the low-level hardware shaders **shader programs**.

**If you want to write shaders that interact with lighting, take a look at Surface Shaders documentation**. The rest of this page will assume shaders that do not interact with Unity lights (e.g. special effects, Image Effects etc.)

Shader programs are written in Cg / HLSL language, by embedding "snippets" in the shader text, somewhere inside the Pass command. They usually look like this:

```
Pass {
    // ... the usual pass state setup ...

    CGPROGRAM
```

```
        // compilation directives for this snippet, e.g.:
        #pragma vertex vert
        #pragma fragment frag

        // the Cg code itself

        ENDCG
        // ... the rest of pass setup ...
    }
```

## Cg snippets

Cg program snippets are written between **CGPROGRAM** and **ENDCG**.

At the start of the snippet compilation directives can be given as **#pragma** statements. Directives indicating which shader functions to compile:

- **#pragma vertex *name*** - compile function *name* as the vertex shader.
- **#pragma fragment *name*** - compile function *name* as the fragment shader.
- **#pragma geometry *name*** - compile function *name* as DX10 geometry shader. Having this option automatically turns on **#pragma target 4.0**, see below.
- **#pragma hull *name*** - compile function *name* as DX11 hull shader. Having this option automatically turns on **#pragma target 5.0**, see below.
- **#pragma domain *name*** - compile function *name* as DX11 domain shader. Having this option automatically turns on **#pragma target 5.0**, see below.

Other compilation directives:

- **#pragma target *name*** - which shader target to compile to. See shader targets for details.
- **#pragma only_renderers *space separated names*** - compile shader only for given renderers. By default shaders are compiled for all renderers. See renderers for details.
- **#pragma exclude_renderers *space separated names*** - do not compile shader for given renderers. By default shaders are compiled for all renderers. See renderers for details.
- **#pragma glsl** - when compiling shaders for desktop OpenGL platforms, convert Cg/HLSL into GLSL (instead of default setting which is ARB vertex/fragment programs). Use this to enable derivative instructions, texture sampling with explicit LOD levels, etc.
- **#pragma glsl_no_auto_normalization** - when compiling shaders for mobile GLSL (iOS/Android), turn off automatic normalization of normal & tangent vectors. By default, normals and tangents are normalized in the vertex shader on iOS/Android platforms.
- **#pragma fragmentoption *option*** - adds *option* to the compiled OpenGL fragment program. See the ARB fragment program specification for a list of allowed options. This directive has no effect on vertex programs or programs that are compiled to non-OpenGL targets.

Each snippet must contain a vertex program, a fragment program, or both. Thus a **#pragma vertex** or **#pragma fragment** directive is required, or both.

## Shader targets

By default, Unity compiles shaders into roughly shader model 2.0 equivalent. Using **#pragma target** allows shaders to be compiled into other capability levels. Currently these targets are supported:

- **#pragma target 2.0** *(default)* - roughly shader model 2.0
  - Shader Model 2.0 on Direct3D 9.
  - ARB_vertex_program with 256 instruction limit and ARB_fragment_program with 96 instruction limit (32 texture + 64 arithmetic), 16 temporary registers and 4 texture indirections.
- **#pragma target 3.0** - compile to shader model 3.0:
  - Shader Model 3.0 on Direct3D 9.
  - ARB_vertex_program with no instruction limit and ARB_fragment_program with 1024 instruction limit (512 texture + 512 arithmetic), 32 temporary registers and 4 texture indirections. It is possible to override these limits using **#pragma profileoption** directive. E.g. `#pragma profileoption MaxTexIndirections=256` raises texture indirections limit to 256. Note that some shader model 3.0 features, like derivative instructions, aren't supported by ARB_vertex_program/ARB_fragment_program. You can use **#pragma glsl** to translate to GLSL instead which has fewer restrictions.

  When compiling to 3.0 or larger target, both vertex and fragment programs need to be present.

- **#pragma target 4.0** - compile to DX10 shader model 4.0. This target is currently only supported by DirectX 11 renderer.
- **#pragma target 5.0** - compile to DX11 shader model 5.0. This target is currently only supported by DirectX 11 renderer.

### Rendering platforms

Unity supports several rendering APIs (e.g. Direct3D 9 and OpenGL), and by default all shader programs are compiled into for supported renderers. You can indicate which renderers to compile to using **#pragma only_renderers** or **#pragma exclude_renderers** directives. This is useful if you know you will only target Mac OS X (where there's no Direct3D), or only Windows (where Unity defaults to D3D), or if some particular shader is only possible in one renderer and not others. Currently supported renderer names are:

- **d3d9** - Direct3D 9.
- **d3d11** - Direct3D 11.
- **opengl** - OpenGL.
- **gles** - OpenGL ES 2.0.
- **xbox360** - Xbox 360.
- **ps3** - PlayStation 3.
- **flash** - Flash.

For example, this line would only compile shader into D3D9 mode:

```
#pragma only_renderers d3d9
```

### Subsections

- Accessing shader properties in Cg
- Providing vertex data to vertex programs
- Built-in shader include files
- Predefined shader preprocessor macros
- Built-in state variables in shader programs
- GLSL Shader Programs

Page last updated: 2012-11-16

---

## SL-PropertiesInPrograms

Shader declares Material properties in a Properties block. If you want to access some of those properties in a shader program, you need to declare a Cg/HLSL variable with the same name and a matching type. An example is provided in Shader Tutorial: Vertex and Fragment Programs.

For example these shader properties:

```
_MyColor ("Some Color", Color) = (1,1,1,1)
_MyVector ("Some Vector", Vector) = (0,0,0,0)
_MyFloat ("My float", Float) = 0.5
_MyTexture ("Texture", 2D) = "white" {}
_MyCubemap ("Cubemap", CUBE) = "" {}
```

would be declared for access in Cg/HLSL code as:

```
fixed4 _MyColor; // low precision type is enough for colors
float4 _MyVector;
float _MyFloat;
sampler2D _MyTexture;
samplerCUBE _MyCubemap;
```

Cg can also accept **uniform** keyword, but it is not necessary:

```
uniform float4 _MyColor;
```

Property types in ShaderLab map to Cg/HLSL variable types this way:

- Color and Vector properties map to **float4**, **half4** or **fixed4** variables.
- Range and Float properties map to **float**, **half** or **fixed** variables.
- Texture properties map to **sampler2D** variables for regular (2D) textures; Cubemaps map to **samplerCUBE**; and 3D textures map to **sampler3D**.

Page last updated: 2012-09-04

---

## SL-VertexProgramInputs

For Cg/HLSL vertex programs, the vertex data must be passed in as a structure. Several commonly used vertex structures are defined in **UnityCG.cginc** include file, and in most cases it's enough just to use them. The structures are:

- **appdata_base**: vertex consists of position, normal and one texture coordinate.
- **appdata_tan**: vertex consists of position, tangent, normal and one texture coordinate.
- **appdata_full**: vertex consists of position, tangent, normal, two texture coordinates and color.

For example, this shader colors the mesh based on it's normals and just uses **appdata_base** as vertex program input:

```
Shader "VertexInputSimple" {
  SubShader {
    Pass {
      CGPROGRAM
      #pragma vertex vert
      #pragma fragment frag
      #include "UnityCG.cginc"

      struct v2f {
          float4 pos : SV_POSITION;
          fixed4 color : COLOR;
      };

      v2f vert (appdata_base v)
      {
          v2f o;
          o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
          o.color.xyz = v.normal * 0.5 + 0.5;
          o.color.w = 1.0;
          return o;
      }

      fixed4 frag (v2f i) : COLOR0 { return i.color; }
      ENDCG
    }
  }
}
```

If you want to access different vertex data, you have to declare vertex structure yourself. The structure **members must be** from the following list:

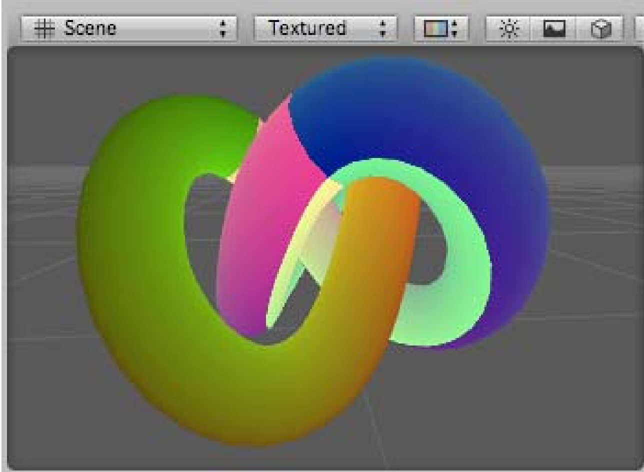
- **float4 vertex** is the vertex position
- **float3 normal** is the vertex normal
- **float4 texcoord** is the first UV coordinate
- **float4 texcoord1** is the second UV coordinate
- **float4 tangent** is the tangent vector (used for normal mapping)
- **float4 color** is per-vertex color

### Examples

### Visualizing UVs

The following shader example uses vertex position and first texture coordinate as vertex shader input (defined in structure **appdata**). It is very useful to debug UV coordinates of the mesh. UV coordinates are visualized as red and green colors, and coordinates outside of 0..1 range have additional blue tint applied.

```
Shader "!Debug/UV 1" {
SubShader {
    Pass {
        Fog { Mode Off }
CGPROGRAM
#pragma vertex vert
#pragma fragment frag

// vertex input: position, UV
struct appdata {
    float4 vertex : POSITION;
    float4 texcoord : TEXCOORD0;
};

struct v2f {
    float4 pos : SV_POSITION;
    float4 uv : TEXCOORD0;
};
v2f vert (appdata v) {
    v2f o;
    o.pos = mul( UNITY_MATRIX_MVP, v.vertex );
    o.uv = float4( v.texcoord.xy, 0, 0 );
    return o;
}
half4 frag( v2f i ) : COLOR {
    half4 c = frac( i.uv );
    if (any(saturate(i.uv) - i.uv))
        c.b = 0.5;
    return c;
}
ENDCG
    }
}
}
```




*Debug UV1 shader applied to a torus knot model*

Similarly, this shader vizualizes the second UV set of the model:

```
Shader "!Debug/UV 2" {
```

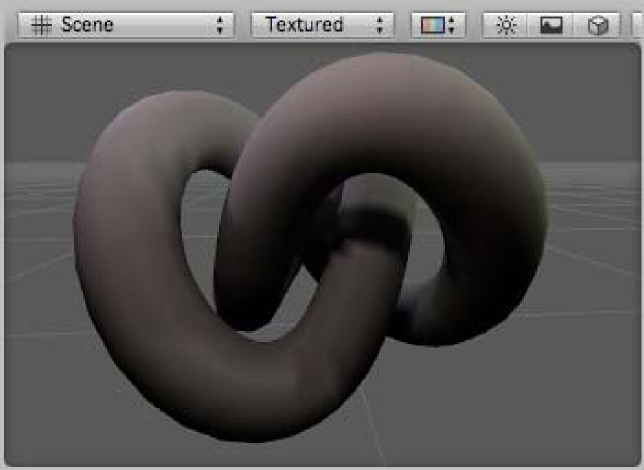
```
SubShader {
    Pass {
        Fog { Mode Off }
CGPROGRAM
#pragma vertex vert
#pragma fragment frag

// vertex input: position, second UV
struct appdata {
    float4 vertex : POSITION;
    float4 texcoord1 : TEXCOORD1;
};

struct v2f {
    float4 pos : SV_POSITION;
    float4 uv : TEXCOORD0;
};
v2f vert (appdata v) {
    v2f o;
    o.pos = mul( UNITY_MATRIX_MVP, v.vertex );
    o.uv = float4( v.texcoord1.xy, 0, 0 );
    return o;
}
half4 frag( v2f i ) : COLOR {
    half4 c = frac( i.uv );
    if (any(saturate(i.uv) - i.uv))
        c.b = 0.5;
    return c;
}
ENDCG
    }
}
}
```

**Visualizing vertex colors**

The following shader uses vertex position and per-vertex colors as vertex shader input (defined in structure **appdata**).

```
Shader "!Debug/Vertex color" {
SubShader {
    Pass {
        Fog { Mode Off }
CGPROGRAM
#pragma vertex vert
#pragma fragment frag

// vertex input: position, color
struct appdata {
    float4 vertex : POSITION;
    fixed4 color : COLOR;
};

struct v2f {
    float4 pos : SV_POSITION;
    fixed4 color : COLOR;
};
v2f vert (appdata v) {
    v2f o;
    o.pos = mul( UNITY_MATRIX_MVP, v.vertex );
    o.color = v.color;
    return o;
```

```
}
fixed4 frag (v2f i) : COLOR0 { return i.color; }
ENDCG
   }
}
}
```

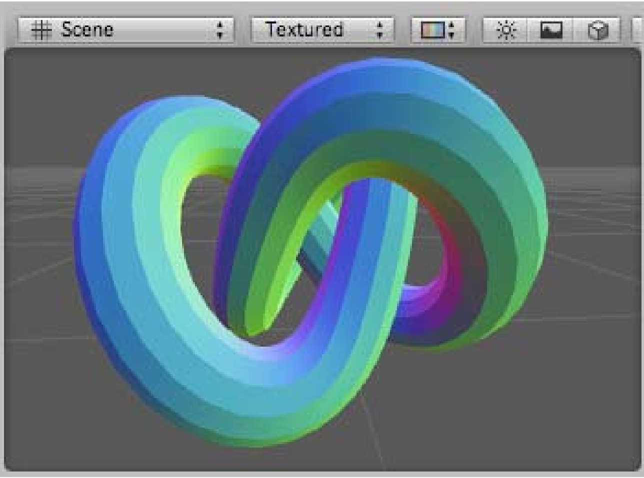


*Debug Colors shader applied to a model that has illumination baked into colors*

**Visualizing normals**

The following shader uses vertex position and normal as vertex shader input (defined in structure **appdata**). Normal's X,Y,Z components are visualized as R,G,B colors. Because the normal components are in -1..1 range, we scale and bias them so that the output colors are in displayable 0..1 range.

```
Shader "!Debug/Normals" {
SubShader {
   Pass {
      Fog { Mode Off }
CGPROGRAM
#pragma vertex vert
#pragma fragment frag

// vertex input: position, normal
struct appdata {
   float4 vertex : POSITION;
   float3 normal : NORMAL;
};

struct v2f {
   float4 pos : SV_POSITION;
   fixed4 color : COLOR;
};
v2f vert (appdata v) {
   v2f o;
   o.pos = mul( UNITY_MATRIX_MVP, v.vertex );
   o.color.xyz = v.normal * 0.5 + 0.5;
   o.color.w = 1.0;
   return o;
}
fixed4 frag (v2f i) : COLOR0 { return i.color; }
ENDCG
   }
}
}
```



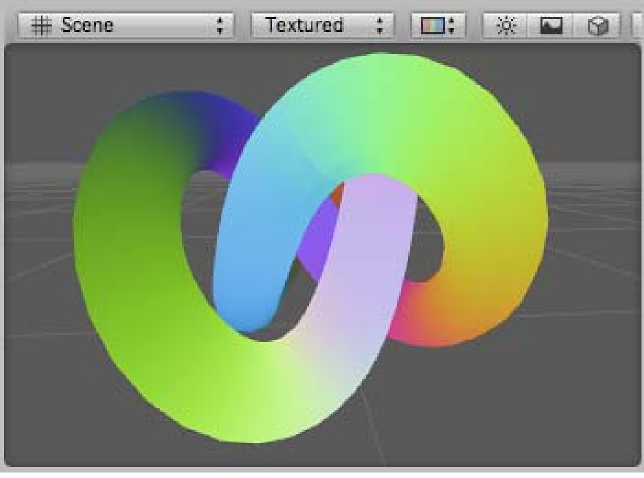

*Debug Normals shader applied to a model. You can see that the model has hard shading edges.*

**Visualizing tangents and binormals**

Tangent and binormal vectors are used for normal mapping. In Unity only the tangent vector is stored in vertices, and binormal is derived from normal and tangent.

The following shader uses vertex position and tangent as vertex shader input (defined in structure **appdata**). Tangent's X,Y,Z components are visualized as R,G,B colors. Because the normal components are in -1..1 range, we scale and bias them so that the output colors are in displayable 0..1 range.

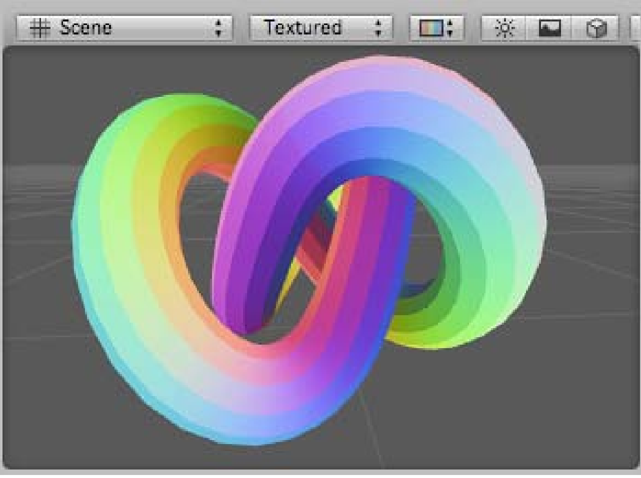
```
Shader "!Debug/Tangents" {
SubShader {
   Pass {
      Fog { Mode Off }
CGPROGRAM
#pragma vertex vert
#pragma fragment frag

// vertex input: position, tangent
struct appdata {
   float4 vertex : POSITION;
   float4 tangent : TANGENT;
};

struct v2f {
   float4    pos : SV_POSITION;
   fixed4    color : COLOR;
};
v2f vert (appdata v) {
   v2f o;
   o.pos = mul( UNITY_MATRIX_MVP, v.vertex );
   o.color = v.tangent * 0.5 + 0.5;
   return o;
}
fixed4 frag (v2f i) : COLOR0 { return i.color; }
ENDCG
   }
}
}
```



*Debug Tangents shader applied to a model.*

The following shader visualizes binormals. It uses vertex position, normal and tangent as vertex input. Binormal is calculated from normal and tangent. Just like normal or tangent, it needs to be scaled and biased into a displayable 0..1 range.

```
Shader "!Debug/Binormals" {
SubShader {
    Pass {
        Fog { Mode Off }
CGPROGRAM
#pragma vertex vert
#pragma fragment frag

// vertex input: position, normal, tangent
struct appdata {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 tangent : TANGENT;
};

struct v2f {
    float4    pos : SV_POSITION;
    float4    color : COLOR;
};
v2f vert (appdata v) {
    v2f o;
    o.pos = mul( UNITY_MATRIX_MVP, v.vertex );
    // calculate binormal
    float3 binormal = cross( v.normal, v.tangent.xyz ) * v.tangent.w;
    o.color.xyz = binormal * 0.5 + 0.5;
    o.color.w = 1.0;
    return o;
}
fixed4 frag (v2f i) : COLOR0 { return i.color; }
ENDCG
    }
}
}
```




*Debug Binormals shader applied to a model. Pretty!*

Page last updated: 2012-09-04

---

# SL-BuiltinIncludes

Unity contains several files that can be used by your shader programs to bring in predefined variables and helper functions. This is done by the standard `#include` directive, e.g.:

```
CGPROGRAM
// ...
#include "UnityCG.cginc"
// ...
ENDCG
```

Shader include files in Unity are with `.cginc` extension, and the built-in ones are:

- `HLSLSupport.cginc` - *(automatically included)* Helper macros and definitions for cross-platform shader compilation.
- `UnityCG.cginc` - commonly used global variables and helper functions.
- `AutoLight.cginc` - lighting & shadowing functionality, e.g. surface shaders use this file internally.
- `Lighting.cginc` - standard surface shader lighting models; automatically included when you're writing surface shaders.
- `TerrainEngine.cginc` - helper functions for Terrain & Vegetation shaders.

These files are found inside Unity application (**{unity install path}/Data/CGIncludes/UnityCG.cginc** on Windows, **/Applications/Unity/Unity.app/Contents/CGIncludes/UnityCG.cginc** on Mac), if you want to take a look at what exactly is done in any of the helper code.

## HLSLSupport.cginc

This file is *automatically included* when compiling shaders. It mostly declares various preprocessor macros to aid in multi-platform shader development.

## UnityCG.cginc

This file is often included in Unity shaders to bring in many helper functions and definitions.

#### Data structures in UnityCG.cginc

- struct `appdata_base`: vertex shader input with position, normal, one texture coordinate.
- struct `appdata_tan`: vertex shader input with position, normal, tangent, one texture coordinate.
- struct `appdata_full`: vertex shader input with position, normal, tangent, vertex color and two texture coordinates.
- struct `appdata_img`: vertex shader input with position and one texture coordinate.

#### Generic helper functions in UnityCG.cginc

- `float3 WorldSpaceViewDir (float4 v)` - returns world space direction (not normalized) from given object space vertex position towards the camera.
- `float3 ObjSpaceViewDir (float4 v)` - returns object space direction (not normalized) from given object space

vertex position towards the camera.
- `float2 ParallaxOffset (half h, half height, half3 viewDir)` - calculates UV offset for parallax normal mapping.
- `fixed Luminance (fixed3 c)` - converts color to luminance (grayscale).
- `fixed3 DecodeLightmap (fixed4 color)` - decodes color from Unity lightmap (RGBM or dLDR depending on platform).
- `float4 EncodeFloatRGBA (float v)` - encodes [0..1) range float into RGBA color, for storage in low precision render target.
- `float DecodeFloatRGBA (float4 enc)` - decodes RGBA color into a float.
- Similarly, `float2 EncodeFloatRG (float v)` and `float DecodeFloatRG (float2 enc)` that use two color channels.
- `float2 EncodeViewNormalStereo (float3 n)` - encodes view space normal into two numbers in 0..1 range.
- `float3 DecodeViewNormalStereo (float4 enc4)` - decodes view space normal from enc4.xy.

**Forward rendering helper functions in UnityCG.cginc**
These functions are only useful when using forward rendering (ForwardBase or ForwardAdd pass types).

- `float3 WorldSpaceLightDir (float4 v)` - computes world space direction (not normalized) to light, given object space vertex position.
- `float3 ObjSpaceLightDir (float4 v)` - computes object space direction (not normalized) to light, given object space vertex position.
- `float3 Shade4PointLights (...)` - computes illumination from four point lights, with light data tightly packed into vectors. Forward rendering uses this to compute per-vertex lighting.

**Vertex-lit helper functions in UnityCG.cginc**
These functions are only useful when using per-vertex lit shaders ("Vertex" pass type).

- `float3 ShadeVertexLights (float4 vertex, float3 normal)` - computes illumination from four per-vertex lights and ambient, given object space position & normal.

Page last updated: 2012-09-04

---

# SL-BuiltinMacros

When compiling shader programs, Unity defines several preprocessor macros.

## Target platform

- `SHADER_API_OPENGL` - desktop OpenGL
- `SHADER_API_D3D9` - Direct3D 9
- `SHADER_API_XBOX360` - Xbox 360
- `SHADER_API_PS3` - PlayStation 3
- `SHADER_API_D3D11` - desktop Direct3D 11
- `SHADER_API_GLES` - OpenGL ES 2.0 (desktop or mobile), use presence of `SHADER_API_MOBILE` to determine.
- `SHADER_API_FLASH` - Flash Stage3D
- `SHADER_API_D3D11_9X` - Direct3D 11 target for Windows RT

Additionally, `SHADER_TARGET_GLSL` is defined when the target shading language is GLSL (always true when `SHADER_API_GLES` is defined; and can be true for `SHADER_API_OPENGL` when `#pragma glsl` is used).

`SHADER_API_MOBILE` is defined for `SHADER_API_GLES` when compiling for "mobile" platform (iOS/Android); and not defined when compiling for "desktop" (NativeClient).

## Platform difference helpers

Direct use of these platform macros is discouraged, since it's not very future proof. For example, if you're writing a shader that checks for D3D9, then maybe in the future the check should be extended to include D3D11. Instead, Unity defines several helper macros (in `HLSLSupport.cginc`) to help with that.

- `UNITY_ATTEN_CHANNEL` - which channel of light attenuation texture contains the data; used in per-pixel lighting code.

Defined to either 'r' or 'a'.

- `UNITY_HALF_TEXEL_OFFSET` - defined on platforms that need a half-texel offset adjustment in mapping texels to pixels (e.g. Direct3D 9).
- `UNITY_UV_STARTS_AT_TOP` - always defined with value or 1 or 0; value of one is on platforms where texture V coordinate is zero at "top of the texture". Direct3D-like platforms use value of 1; OpenGL-like platforms use value of 0.
- `UNITY_MIGHT_NOT_HAVE_DEPTH_TEXTURE` - defined if a platform might emulate shadow maps or depth textures by manually rendering depth into a texture.
- `UNITY_PROJ_COORD(a)` - given a 4-component vector, return a texture coordinate suitable for projected texture reads. On most platforms this returns the given value directly.
- `UNITY_NEAR_CLIP_VALUE` - defined to the value of near clipping plane; Direct3D-like platforms use 0.0 while OpenGL-like platforms use -1.0.
- `UNITY_COMPILER_CG`, `UNITY_COMPILER_HLSL` or `UNITY_COMPILER_HLSL2GLSL` determine which underlying shader compiler is used; use in case of subtle syntax differences force you to write different shader code.
- `UNITY_CAN_COMPILE_TESSELLATION` - defined when the shader compiler "understands" tessellation shader HLSL syntax (currently only D3D11).
- `UNITY_INITIALIZE_OUTPUT(type, name)` - initialize variable *name* of given *type* to zero.

### Constant buffer macros

Direct3D 11 groups all shader variables into "constant buffers". Most of Unity's built-in variables are already grouped, but for variables in your own shaders it might be more optimal to put them into separate constant buffers depending on expected frequency of updates.

Use `CBUFFER_START(name)` and `CBUFFER_END` macros for that:

```
CBUFFER_START(MyRarelyUpdatedVariables)
float4 _SomeGlobalValue;
CBUFFER_END
```

### Surface shader pass indicators

When Surface Shaders are compiled, they end up generating a lot of code for various passes to do lighting. When compiling each pass, one of the following macros is defined:

- `UNITY_PASS_FORWARDBASE` - forward rendering base pass (main directional light, lightmaps, SH).
- `UNITY_PASS_FORWARDADD` - forward rendering additive pass (one light per pass).
- `UNITY_PASS_PREPASSBASE` - deferred lighting base pass (renders normals & specular exponent).
- `UNITY_PASS_PREPASSFINAL` - deferred lighting final pass (applies lighting & textures).
- `UNITY_PASS_SHADOWCASTER` - shadow caster rendering pass.
- `UNITY_PASS_SHADOWCOLLECTOR` - shadow "gathering" pass for directional light shadows.

Page last updated: 2012-11-16

---

## SL-BuiltinStateInPrograms

Often in shader programs you need to access some global state, for example, the current model*view*projection matrix, the current ambient color, and so on. There's no need to declare these variables for the built-in state, you can just use them in shader programs.

### Built-in matrices

Matrices (float4x4) supported:

**UNITY_MATRIX_MVP**
: Current model*view*projection matrix

**UNITY_MATRIX_MV**
: Current model*view matrix

**UNITY_MATRIX_P**
: Current projection matrix

**UNITY_MATRIX_T_MV**
: Transpose of model*view matrix

**UNITY_MATRIX_IT_MV**
Inverse transpose of model*view matrix
**UNITY_MATRIX_TEXTURE0 to UNITY_MATRIX_TEXTURE3**
Texture transformation matrices

### Built-in vectors

Vectors (float4) supported:

**UNITY_LIGHTMODEL_AMBIENT**
Current ambient color.

Page last updated: 2010-08-18

---

## SL-GLSLShaderPrograms

In addition to using Cg/HSL shader programs, OpenGL Shading Language (GLSL) shaders can be written directly.

However, **use of raw GLSL is only recommended for testing**, or when you know you will only target Mac OS X or OpenGL ES 2.0 compatible mobile devices. In majority of normal cases, Unity will cross-compile Cg/HLSL into optimized GLSL (this is done by default for mobile platforms, and can be optionally turned on for desktop platforms via `#pragma glsl`).

### GLSL snippets

GLSL program snippets are written between **GLSLPROGRAM** and **ENDGLSL** keywords.

In GLSL, all shader function entry points have to be called **main()**. When Unity loads the GLSL shader, it loads the source once for the vertex program, with **VERTEX** preprocessor define, and once more for the fragment program, with **FRAGMENT** preprocessor define. So the way to separate vertex and fragment program parts in GLSL snippet is to surround them with **#ifdef VERTEX** .. **#endif** and **#ifdef FRAGMENT** .. **#endif**. Each GLSL snippet must contain both a vertex program and a fragment program.

Standard include files match those provided for Cg shaders; they just have **.glslinc** extension: **UnityCG.glslinc**.

Vertex shader inputs come from predefined GLSL variables (gl_Vertex, gl_MultiTexCoord0, ...) or are user defined attributes. Usually only the tangent vector needs a user defined attribute:

```
attribute vec4 Tangent;
```

Data from vertex to fragment programs is passed through *varying* variables, for example:

```
varying vec3 lightDir; // vertex shader computes this, fragment shader uses this
```

Page last updated: 2012-01-02

---

## SL-Shader

**Shader** is the root command of a shader file. Each file must define one (and only one) Shader. It specifies how any objects whose material uses this shader are rendered.

### Syntax

**Shader** "*name*" **{** [Properties] Subshaders [Fallback] **}** Defines a shader. It will appear in the material inspector listed under **name**. Shaders optionally can define a list of **properties** that show up as material settings. After this comes a list of SubShaders, and optionally a fallback.

### Details

#### Properties

Shaders can have a list of properties. Any properties declared in a shader are shown in the material inspector inside Unity.

Typical properties are the object color, textures, or just arbitrary values to be used by the shader.

**SubShaders & Fallback**
Each shader is comprised of a list of sub-shaders. You must have at least one. When loading a shader, Unity will go through the list of subshaders, and pick the first one that is supported by the end user's machine. If no subshaders are supported, Unity will try to use fallback shader.

Different graphic cards have different capabilities. This raises an eternal issue for game developers; you want your game to look great on the latest hardware, but don't want it to be available only to those 3% of the population. This is where subshaders come in. Create one subshader that has all the fancy graphic effects you can dream of, then add more subshaders for older cards. These subshaders may implement the effect you want in a slower way, or they may choose not to implement some details.

## Examples

Here is one of the simplest shaders possible:

```
// colored vertex lighting
Shader "Simple colored lighting" {
    // a single color property
    Properties {
        _Color ("Main Color", Color) = (1,.5,.5,1)
    }
    // define one subshader
    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
            }
            Lighting On
        }
    }
}
```

This shader defines a color property **_Color** (that shows up in material inspector as *Main Color*) with a default value of **(1, 0.5, 0.5, 1)**. Then a single subshader is defined. The subshader consists of one Pass that turns on vertex lighting and sets up basic material for it.

## Subsections

- ShaderLab syntax: Properties
- ShaderLab syntax: SubShader
  - ShaderLab syntax: Pass
    - ShaderLab syntax: Color, Material, Lighting
    - ShaderLab syntax: Culling & Depth Testing
    - ShaderLab syntax: Texture Combiners
    - ShaderLab syntax: Fog
    - ShaderLab syntax: Alpha testing
    - ShaderLab syntax: Blending
    - ShaderLab syntax: Pass Tags
    - ShaderLab syntax: Name
    - ShaderLab syntax: BindChannels
  - ShaderLab syntax: UsePass
  - ShaderLab syntax: GrabPass
  - ShaderLab syntax: SubShader Tags
- ShaderLab syntax: Fallback
- ShaderLab syntax: other commands

Page last updated: 2008-06-16

# SL-Properties

Shaders can define a list of parameters to be set by artists in Unity's material inspector. The Properties block in the shader file defines them.

## Syntax

**Properties { *Property* [*Property ...*] }**
: Defines the property block. Inside braces multiple properties are defined as follows.

***name* ("*display name*", Range (*min*, *max*)) = *number***
: Defines a float property, represented as a slider from *min* to *max* in the inspector.

***name* ("*display name*", Color) = (*number*,*number*,*number*,*number*)**
: Defines a color property.

***name* ("*display name*", 2D) = "*name*" { *options* }**
: Defines a 2D texture property.

***name* ("*display name*", Rect) = "*name*" { *options* }**
: Defines a rectangle (non power of 2) texture property.

***name* ("*display name*", Cube) = "*name*" { *options* }**
: Defines a cubemap texture property.

***name* ("*display name*", Float) = *number***
: Defines a float property.

***name* ("*display name*", Vector) = (*number*,*number*,*number*,*number*)**
: Defines a four component vector property.

## Details

Each property inside the shader is referenced by **name** (in Unity, it's common to start shader property names with underscore). The property will show up in material inspector as **display name**. For each property a default value is given after equals sign:

- For *Range* and *Float* properties it's just a single number.
- For *Color* and *Vector* properties it's four numbers in parentheses.
- For texture (*2D*, *Rect*, *Cube*) the default value is either an empty string, or one of builtin default textures: "*white*", "*black*", "*gray*" or "*bump*".

Later on in the shader, property values are accessed using property name in square brackets: **[name]**.

## Example

```
Properties {
    // properties for water shader
    _WaveScale ("Wave scale", Range (0.02,0.15)) = 0.07 // sliders
    _ReflDistort ("Reflection distort", Range (0,1.5)) = 0.5
    _RefrDistort ("Refraction distort", Range (0,1.5)) = 0.4
    _RefrColor ("Refraction color", Color)  = (.34, .85, .92, 1) // color
    _ReflectionTex ("Environment Reflection", 2D) = "" {} // textures
    _RefractionTex ("Environment Refraction", 2D) = "" {}
    _Fresnel ("Fresnel (A) ", 2D) = "" {}
    _BumpMap ("Bumpmap (RGB) ", 2D) = "" {}
}
```

### Texture property options

The *options* inside curly braces of the texture property are optional. The available options are:

- **TexGen** *texgenmode*: Automatic texture coordinate generation mode for this texture. Can be one of *ObjectLinear*, *EyeLinear*, *SphereMap*, *CubeReflect*, *CubeNormal*; these correspond directly to OpenGL texgen modes. Note that TexGen is ignored if custom vertex programs are used.
- **LightmapMode** If given, this texture will be affected by per-renderer lightmap parameters. That is, the texture to use can be not in the material, but taken from the settings of the Renderer instead, see Renderer scripting documentation.

### Example

```
// EyeLinear texgen mode example
Shader "Texgen/Eye Linear" {
```

```
    Properties {
        _MainTex ("Base", 2D) = "white" { TexGen EyeLinear }
    }
    SubShader {
        Pass {
            SetTexture [_MainTex] { combine texture }
        }
    }
}
```

Page last updated: 2012-02-29

---

# SL-SubShader

Each shader in Unity consists of a list of subshaders. When Unity has to display a mesh, it will find the shader to use, and pick the first subshader that runs on the user's graphics card.

### Syntax

**Subshader { [*Tags*] [*CommonState*] *Passdef* [*Passdef ...*] }**

Defines the subshader as optional tags, common state and a list of pass definitions.

### Details

A subshader defines a list of rendering passes and optionally setup any state that is common to all passes. Additionally, subshader specific Tags can be set up.

When Unity chooses which subshader to render with, it renders an object once for each Pass defined (and possibly more due to light interactions). As each render of the object is an expensive operation, you want to define the shader in minimum amount of passes possible. Of course, sometimes on some graphics hardware the needed effect can't be done in a single pass; then you have no choice but to use multiple passes.

Each pass definition can be a regular Pass, a Use Pass or a Grab Pass.

Any statements that are allowed in a Pass definition can also appear in Subshader block. This will make all passes use this "shared" state.

### Example

```
// ...
SubShader {
    Pass {
        Lighting Off
        SetTexture [_MainTex] {}
    }
}
// ...
```

This subshader defines a single Pass that turns off any lighting and just displays a mesh with texture named **_MainTex**.

Page last updated: 2009-10-19

---

# SL-Pass

The Pass block causes the geometry of an object to be rendered once.

### Syntax

**Pass { *[Name and Tags] [RenderSetup] [TextureSetup]* }**
: The basic pass command contains an optional list of render setup commands, optionally followed by a list of textures to use.

## Name and tags

A Pass can define its Name and arbitrary number of Tags - name/value strings that communicate Pass' intent to the rendering engine.

## Render Setup

A pass sets up various states of the graphics hardware, for example should alpha blending be turned on, should fog be used, and so on. The commands are these:

**Material { *Material Block* }**
: Defines a material to use in a vertex lighting pipeline. See material page for details.

**Lighting On | Off**
: Turn vertex lighting on or off. See material page for details.

**Cull Back | Front | Off**
: Set polygon culling mode.

**ZTest (Less | Greater | LEqual | GEqual | Equal | NotEqual | Always)**
: Set depth testing mode.

**ZWrite On | Off**
: Set depth writing mode.

**Fog { *Fog Block* }**
: Set fog parameters.

**AlphaTest (Less | Greater | LEqual | GEqual | Equal | NotEqual | Always) *CutoffValue***
: Turns on alpha testing.

**Blend *SourceBlendMode DestBlendMode***
: Sets alpha blending mode.

**Color *Color value***
: Sets color to use if vertex lighting is turned off.

**ColorMask RGB | A | 0 | any combination of R, G, B, A**
: Set color writing mask. Writing *ColorMask 0* turns off rendering to all color channels.

**Offset *OffsetFactor* , *OffsetUnits***
: Set depth offset. Note that this command intentionally only accepts constants (i.e., not shader parameters) as of Unity 3.0.

**SeparateSpecular On | Off**
: Turns separate specular color for vertex lighting on or off. See material page for details.

**ColorMaterial AmbientAndDiffuse | Emission**
: Uses per-vertex color when computing vertex lighting. See material page for details.

## Texture Setup

After the render state setup, you can specify a number of textures and their combining modes to apply using SetTexture commands:

**SetTexture** *texture property* { *[Combine options]* }

The texture setup configures fixed function multitexturing pipeline, and is ignored if custom fragment shaders are used.

## Details

### Per-pixel Lighting

The per-pixel lighting pipeline works by rendering objects in multiple passes. Unity renders the object once to get ambient and any vertex lights in. Then it renders each pixel light affecting the object in a separate additive pass. See Render Pipeline for details.

### Per-vertex Lighting

Per-vertex lighting is the standard Direct3D/OpenGL lighting model that is computed for each vertex. **Lighting on** turns it on. Lighting is affected by **Material** block, **ColorMaterial** and **SeparateSpecular** commands. See material page for details.

## See Also

There are several special passes available for reusing common functionality or implementing various high-end effects:

- UsePass includes named passes from another shader.
- GrabPass grabs the contents of the screen into a texture, for use in a later pass.

### Subsections

- ShaderLab syntax: Color, Material, Lighting
- ShaderLab syntax: Culling & Depth Testing
- ShaderLab syntax: Texture Combiners
- ShaderLab syntax: Fog
- ShaderLab syntax: Alpha testing
- ShaderLab syntax: Blending
- ShaderLab syntax: Pass Tags
- ShaderLab syntax: Name
- ShaderLab syntax: BindChannels

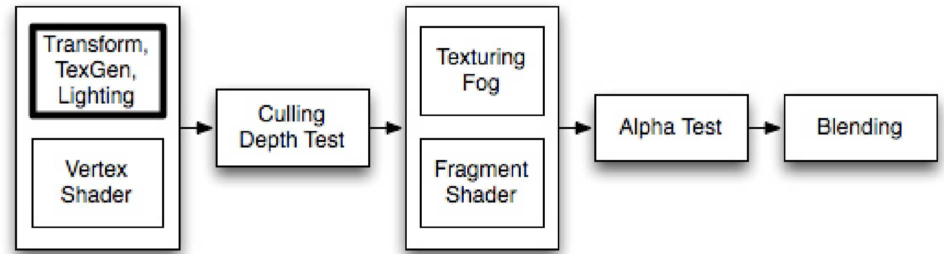
Page last updated: 2012-01-17

---

## SL-Material

The material and lighting parameters are used to control the built-in vertex lighting. Vertex lighting is the standard Direct3D/OpenGL lighting model that is computed for each vertex. **Lighting on** turns it on. Lighting is affected by **Material** block, **ColorMaterial** and **SeparateSpecular** commands.

Per-pixel lights are usually implemented with custom vertex/fragment programs and don't use vertex lighting. For these you don't use any of the commands described here, instead you define your own vertex and fragment programs where you do all lighting, texturing and anything else yourself.




Vertex Coloring & Lighting is the first effect to gets calculated for any rendered geometry. It operates on the vertex level, and calculates the base color that is used before textures are applied.

### Syntax

The top level commands control whether to use fixed function lighting or not, and some configuration options. The main setup is in the **Material Block**, detailed further below.

**Color *Color***
: Sets the object to a solid color. A color is either four RGBA values in parenthesis, or a color property name in square brackets.

**Material { *Material Block* }**
: The Material block is used to define the material properties of the object.

**Lighting On | Off**
: For the settings defined in the Material block to have any effect, you must enable Lighting with the *Lighting On* command. If lighting is off instead, the color is taken straight from the *Color* command.

**SeparateSpecular On | Off**
: This command makes specular lighting be added to the end of the shader pass, so specular lighting is unaffected by texturing. Only has effect when *Lighting On* is used.

**ColorMaterial AmbientAndDiffuse | Emission**
: makes per-vertex color be used instead of the colors set in the material. **AmbientAndDiffuse** replaces Ambient and Diffuse values of the material; **Emission** replaces Emission value of the material.

**Material Block**

This contains settings for how the material reacts to the light. Any of these properties can be left out, in which case they

default to black (i.e. have no effect).

**Diffuse *Color***
: The diffuse color component. This is an object's base color.

**Ambient *Color***
: The ambient color component. This is the color the object has when it's hit by the ambient light set in the RenderSettings.

**Specular *Color***
: The color of the object's specular highlight.

**Shininess *Number***
: The sharpness of the highlight, between 0 and 1. At 0 you get a huge highlight that looks a lot like diffuse lighting, at 1 you get a tiny speck.

**Emission *Color***
: The color of the object when it is not hit by any light.

The full color of lights hitting the object is:

```
Ambient * RenderSettings ambient setting +
(Light Color * Diffuse + Light Color * Specular) + Emission
```

The light parts of the equation (within parenthesis) is repeated for all lights that hit the object.

Typically you want to keep the Diffuse and Ambient colors the same (all builtin Unity shaders do this).

## Examples

Always render object in pure red:

```
Shader "Solid Red" {
    SubShader {
        Pass { Color (1,0,0,0) }
    }
}
```

Basic Shader that colors the object white and applies vertex lighting:

```
Shader "VertexLit White" {
    SubShader {
        Pass {
            Material {
                Diffuse (1,1,1,1)
                Ambient (1,1,1,1)
            }
            Lighting On
        }
    }
}
```

An extended version that adds material color as a property visible in Material Inspector:

```
Shader "VertexLit Simple" {
    Properties {
        _Color ("Main Color", COLOR) = (1,1,1,1)
    }
    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
                Ambient [_Color]
            }
```

```
            Lighting On
        }
    }
}
```

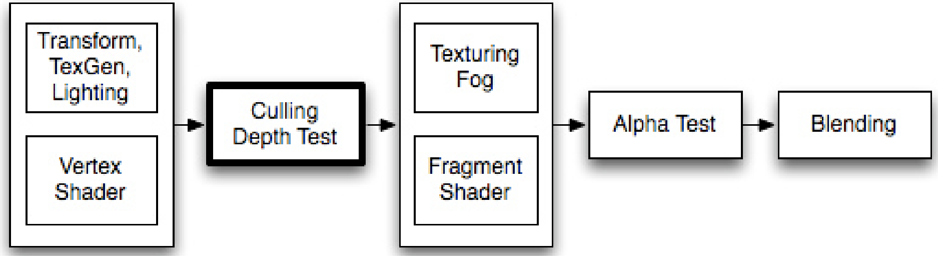
And finally, a full fledged vertex-lit shader (see also SetTexture reference page):

```
Shader "VertexLit" {
    Properties {
        _Color ("Main Color", Color) = (1,1,1,0)
        _SpecColor ("Spec Color", Color) = (1,1,1,1)
        _Emission ("Emmisive Color", Color) = (0,0,0,0)
        _Shininess ("Shininess", Range (0.01, 1)) = 0.7
        _MainTex ("Base (RGB)", 2D) = "white" {}
    }
    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
                Ambient [_Color]
                Shininess [_Shininess]
                Specular [_SpecColor]
                Emission [_Emission]
            }
            Lighting On
            SeparateSpecular On
            SetTexture [_MainTex] {
                Combine texture * primary DOUBLE, texture * primary
            }
        }
    }
}
```

Page last updated: 2009-07-27

---

# SL-CullAndDepth




Culling is an optimization that does not render polygons facing away from the viewer. All polygons have a front and a back side. Culling makes use of the fact that most objects are closed; if you have a cube, you will never see the sides facing away from you (there is always a side facing you in front of it) so we don't need to draw the sides facing away. Hence the term: Backface culling.

The other feature that makes rendering looks correct is Depth testing. Depth testing makes sure that only the closest surfaces objects are drawn in a scene.

### Syntax

**Cull Back | Front | Off**

Controls which sides of polygons should be culled (not drawn)

**Back** Don't render polygons facing away from the viewer *(default)*.

**Front** Don't render polygons facing towards the viewer. Used for turning objects inside-out.

**Off** Disables culling - all faces are drawn. Used for special effects.

**ZWrite On | Off**

Controls whether pixels from this object are written to the depth buffer (default is *On*). If you're drawng solid objects, leave this on. If you're drawing semitransparent effects, switch to *ZWrite Off*. For more details read below.

**ZTest Less | Greater | LEqual | GEqual | Equal | NotEqual | Always**

How should depth testing be performed. Default is *LEqual* (draw objects in from or at the distance as existing objects; hide objects behind them).

**Offset *Factor* , *Units***

Allows you specify a depth offset with two parameters. *factor* and *units*. *Factor* scales the maximum Z slope, with respect to X or Y of the polygon, and *units* scale the minimum resolvable depth buffer value. This allows you to force one polygon to be drawn on top of another although they are actually in the same position. For example *Offset 0, -1* pulls the polygon closer to the camera ignoring the polygon's slope, whereas *Offset -1, -1* will pull the polygon even closer when looking at a grazing angle.

## Examples

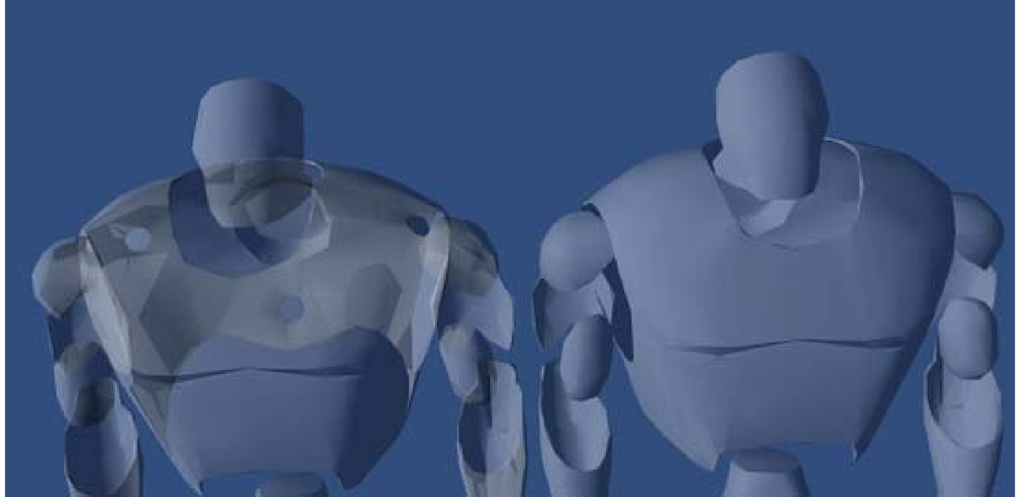
This object will render only the backfaces of an object:

```
Shader "Show Insides" {
   SubShader {
      Pass {
         Material {
            Diffuse (1,1,1,1)
         }
         Lighting On
         Cull Front
      }
   }
}
```

Try to apply it to a cube, and notice how the geometry feels all wrong when you orbit around it. This is because you're only seeing the inside parts of the cube.

#### Transparent shader with depth writes

Usually semitransparent shaders do not write into the depth buffer. However, this can create draw order problems, especially with complex non-convex meshes. If you want to fade in & out meshes like that, then using a shader that fills in the depth buffer before rendering transparency might be useful.



*Semitransparent object; left: standard Transparent/Diffuse shader; right: shader that writes to depth buffer.*

```
Shader "Transparent/Diffuse ZWrite" {
Properties {
```

```
    _Color ("Main Color", Color) = (1,1,1,1)
    _MainTex ("Base (RGB) Trans (A)", 2D) = "white" {}
}
SubShader {
    Tags {"Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent"}
    LOD 200

    // extra pass that renders to depth buffer only
    Pass {
        ZWrite On
        ColorMask 0
    }

    // paste in forward rendering passes from Transparent/Diffuse
    UsePass "Transparent/Diffuse/FORWARD"
}
Fallback "Transparent/VertexLit"
}
```

**Debugging Normals**

The next one is more interesting; first we render the object with normal vertex lighting, then we render the backfaces in bright pink. This has the effects of highlighting anywhere your normals need to be flipped. If you see physically-controlled objects getting 'sucked in' by any meshes, try to assign this shader to them. If any pink parts are visible, these parts will pull in anything unfortunate enough to touch it.

Here we go:

```
Shader "Reveal Backfaces" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" { }
    }
    SubShader {
        // Render the front-facing parts of the object.
        // We use a simple white material, and apply the main texture.
        Pass {
            Material {
                Diffuse (1,1,1,1)
            }
            Lighting On
            SetTexture [_MainTex] {
                Combine Primary * Texture
            }
        }

        // Now we render the back-facing triangles in the most
        // irritating color in the world: BRIGHT PINK!
        Pass {
            Color (1,0,1,1)
            Cull Front
        }
    }
}
```

**Glass Culling**

Controlling Culling is useful for more than debugging backfaces. If you have transparent objects, you quite often want to show the backfacing side of an object. If you render without any culling (**Cull Off**), you'll most likely have some rear faces overlapping some of the front faces.

Here is a simple shader that will work for convex objects (spheres, cubes, car windscreens).

```
Shader "Simple Glass" {
    Properties {
        _Color ("Main Color", Color) = (1,1,1,0)
        _SpecColor ("Spec Color", Color) = (1,1,1,1)
        _Emission ("Emmisive Color", Color) = (0,0,0,0)
        _Shininess ("Shininess", Range (0.01, 1)) = 0.7
        _MainTex ("Base (RGB)", 2D) = "white" { }
    }

    SubShader {
        // We use the material in many passes by defining them in the subshader.
        // Anything defined here becomes default values for all contained passes.
        Material {
            Diffuse [_Color]
            Ambient [_Color]
            Shininess [_Shininess]
            Specular [_SpecColor]
            Emission [_Emission]
        }
        Lighting On
        SeparateSpecular On

        // Set up alpha blending
        Blend SrcAlpha OneMinusSrcAlpha

        // Render the back facing parts of the object.
        // If the object is convex, these will always be further away
        // than the front-faces.
        Pass {
            Cull Front
            SetTexture [_MainTex] {
                Combine Primary * Texture
            }
        }
        // Render the parts of the object facing us.
        // If the object is convex, these will be closer than the
        // back-faces.
        Pass {
            Cull Back
            SetTexture [_MainTex] {
                Combine Primary * Texture
            }
        }
    }
}
```

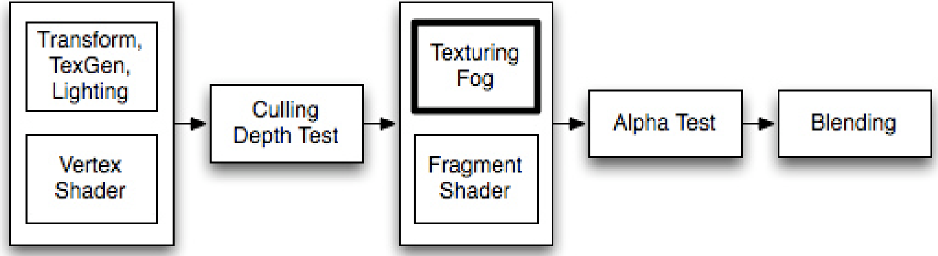
Page last updated: 2012-09-05

---

# SL-SetTexture

After the basic vertex lighting has been calculated, textures are applied. In ShaderLab this is done using **SetTexture** command.

*SetTexture commands have no effect when fragment programs are used; as in that case pixel operations are completely described in the shader.*




Texturing is the place to do old-style combiner effects. You can have multiple SetTexture commands inside a pass - all textures are applied in sequence, like layers in a painting program. SetTexture commands must be placed at the end of a Pass.

### Syntax

**SetTexture *[TexturePropertyName]* { *Texture Block* }**
: Assigns a texture. *TextureName* must be defined as a texture property. How to apply the texture is defined inside the *TextureBlock*.

The texture block controls how the texture is applied. Inside the texture block can be up to three commands: `combine`, `matrix` and `constantColor`.

### Texture block `combine` command

`combine` ***src1* * *src2***
: Multiplies src1 and src2 together. The result will be darker than either input.

`combine` ***src1* + *src2***
: Adds src1 and src2 together. The result will be lighter than either input.

`combine` ***src1* - *src2***
: Subtracts src2 from src1.

`combine` ***src1* +- *src2***
: Adds src1 to src2, then subtracts 0.5 (a signed add).

`combine` ***src1*** `lerp` **(*src2*) *src3***
: Interpolates between src3 and src1, using the alpha of src2. Note that the interpolation is opposite direction: src1 is used when alpha is one, and src3 is used when alpha is zero.

`combine` ***src1* * *src2* + *src3***
: Multiplies src1 with the alpha component of src2, then adds src3.

`combine` ***src1* * *src2* +- *src3***
: Multiplies src1 with the alpha component of src2, then does a signed add with src3.

`combine` ***src1* * *src2* - *src3***
: Multiplies src1 with the alpha component of src2, then subtracts src3.

All the **src** properties can be either one of *previous*, *constant*, *primary* or *texture*.

- **Previous** is the the result of the previous SetTexture.
- **Primary** is the color from the lighting calculation or the vertex color if it is bound.
- **Texture** is the color of the texture specified by *[_TextureName]* in the SetTexture (see above).
- **Constant** is the color specified in **ConstantColor**.

Modifiers:

- The formulas specified above can optionally be followed by the keywords **Double** or **Quad** to make the resulting color 2x or 4x as bright.
- All the **src** properties, except `lerp` argument, can optionally be preceded by **one -** to make the resulting color negated.
- All the **src** properties can be followed by **alpha** to take only the alpha channel.

### Texture block `constantColor` command

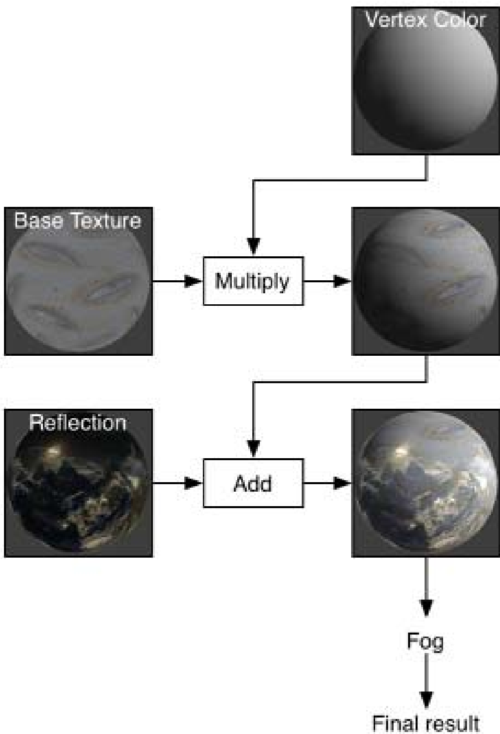
**ConstantColor *color***
: Defines a constant color that can be used in the combine command.

### Texture block `matrix` command

**matrix *[MatrixPropertyName]***
: Transforms texture coordinates used in this command with the given matrix.

## Details

Before fragment programs existed, older graphics cards used a layered approach to textures. The textures are applied one after each other, modifying the color that will be written to the screen. For each texture, the texture is typically combined with the result of the previous operation.




Note that on "true fixed function" devices (OpenGL, OpenGL ES 1.1, Wii) the value of each SetTexture stage is clamped to 0..1 range. Everywhere else (Direct3D, OpenGL ES 2.0) the range may or may not be higher. This might affect SetTexture stages that can produce values higher than 1.0.

### Separate Alpha & Color computation

By default, the combiner formula is used for calculating both the RGB and alpha component of the color. Optionally, you can specify a separate formula for the alpha calculation. This looks like this:

```
SetTexture [_MainTex] { combine previous * texture, previous + texture }
```

Here, we multiply the RGB colors and add the alpha.

### Specular highlights

By default the **primary** color is the sum of the diffuse, ambient and specular colors (as defined in the Lighting calculation). If you specify **SeparateSpecular On** in the pass options, the specular color will be added in *after* the combiner calculation, rather than before. This is the default behavior of the built-in VertexLit shader.

### Graphics hardware support

Modern graphics cards with fragment shader support ("shader model 2.0" on desktop, OpenGL ES 2.0 on mobile) support all **SetTexture** modes and at least 4 texture stages (many of them support 8). If you're running on really old hardware (made before 2003 on PC, or before iPhone3GS on mobile), you might have as low as two texture stages. The shader author should write separate SubShaders for the cards he or she wants to support.

## Examples

### Alpha Blending Two Textures

This small examples takes two textures. First it sets the first combiner to just take the **_MainTex**, then is uses the alpha channel of **_BlendTex** to fade in the RGB colors of **_BlendTex**

```
Shader "Examples/2 Alpha Blended Textures" {
   Properties {
      _MainTex ("Base (RGB)", 2D) = "white" {}
      _BlendTex ("Alpha Blended (RGBA) ", 2D) = "white" {}
   }
   SubShader {
      Pass {
         // Apply base texture
         SetTexture [_MainTex] {
            combine texture
         }
         // Blend in the alpha texture using the lerp operator
         SetTexture [_BlendTex] {
            combine texture lerp (texture) previous
         }
      }
   }
}
```

### Alpha Controlled Self-illumination

This shader uses the alpha component of the **_MainTex** to decide where to apply lighting. It does this by applying the texture to two stages; In the first stage, the alpha value of the texture is used to blend between the vertex color and solid white. In the second stage, the RGB values of the texture are multiplied in.

```
Shader "Examples/Self-Illumination" {
   Properties {
      _MainTex ("Base (RGB) Self-Illumination (A)", 2D) = "white" {}
   }
   SubShader {
      Pass {
         // Set up basic white vertex lighting
         Material {
            Diffuse (1,1,1,1)
            Ambient (1,1,1,1)
         }
         Lighting On

         // Use texture alpha to blend up to white (= full illumination)
         SetTexture [_MainTex] {
            constantColor (1,1,1,1)
            combine constant lerp(texture) previous
         }
         // Multiply in texture
         SetTexture [_MainTex] {
            combine previous * texture
         }
      }
   }
}
```

We can do something else for free here, though; instead of blending to solid white, we can add a self-illumination color and blend to that. Note the use of **ConstantColor** to get a _SolidColor from the properties into the texture blending.

```
Shader "Examples/Self-Illumination 2" {
   Properties {
      _IlluminCol ("Self-Illumination color (RGB)", Color) = (1,1,1,1)
      _MainTex ("Base (RGB) Self-Illumination (A)", 2D) = "white" {}
   }
```

```
    SubShader {
        Pass {
            // Set up basic white vertex lighting
            Material {
                Diffuse (1,1,1,1)
                Ambient (1,1,1,1)
            }
            Lighting On

            // Use texture alpha to blend up to white (= full illumination)
            SetTexture [_MainTex] {
                // Pull the color property into this blender
                constantColor [_IlluminCol]
                // And use the texture's alpha to blend between it and
                // vertex color
                combine constant lerp(texture) previous
            }
            // Multiply in texture
            SetTexture [_MainTex] {
                combine previous * texture
            }
        }
    }
}
```

And finally, we take all the lighting properties of the vertexlit shader and pull that in:

```
Shader "Examples/Self-Illumination 3" {
    Properties {
        _IlluminCol ("Self-Illumination color (RGB)", Color) = (1,1,1,1)
        _Color ("Main Color", Color) = (1,1,1,0)
        _SpecColor ("Spec Color", Color) = (1,1,1,1)
        _Emission ("Emmisive Color", Color) = (0,0,0,0)
        _Shininess ("Shininess", Range (0.01, 1)) = 0.7
        _MainTex ("Base (RGB)", 2D) = "white" {}
    }

    SubShader {
        Pass {
            // Set up basic vertex lighting
            Material {
                Diffuse [_Color]
                Ambient [_Color]
                Shininess [_Shininess]
                Specular [_SpecColor]
                Emission [_Emission]
            }
            Lighting On

            // Use texture alpha to blend up to white (= full illumination)
            SetTexture [_MainTex] {
                constantColor [_IlluminCol]
                combine constant lerp(texture) previous
            }
            // Multiply in texture
            SetTexture [_MainTex] {
                combine previous * texture
            }
        }
    }
```

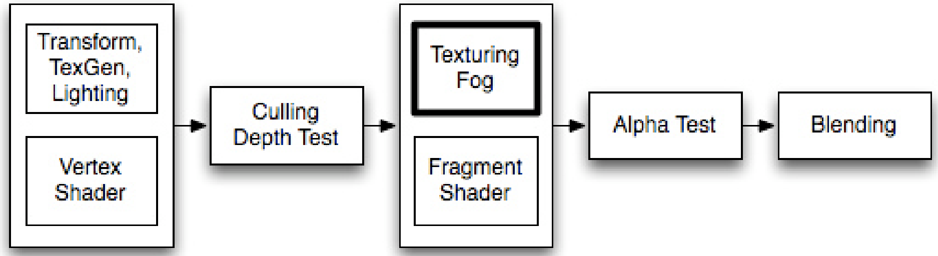
```
}
```

Page last updated: 2012-08-17

---

## SL-Fog

Fog parameters are controlled with Fog command.




Fogging blends the color of the generated pixels down towards a constant color based on distance from camera. Fogging does not modify a blended pixel's alpha value, only its RGB components.

### Syntax

**Fog { *Fog Commands* }**
: Specify fog commands inside curly braces.

**Mode Off | Global | Linear | Exp | Exp2**
: Defines fog mode. Default is global, which translates to Off or Exp2 depending whether fog is turned on in Render Settings.

**Color *ColorValue***
: Sets fog color.

**Density *FloatValue***
: Sets density for exponential fog.

**Range *FloatValue* , *FloatValue***
: Sets near & far range for linear fog.

### Details

Default fog settings are based on Render Settings: fog mode is either **Exp2** or **Off**; density & color taken from settings as well.

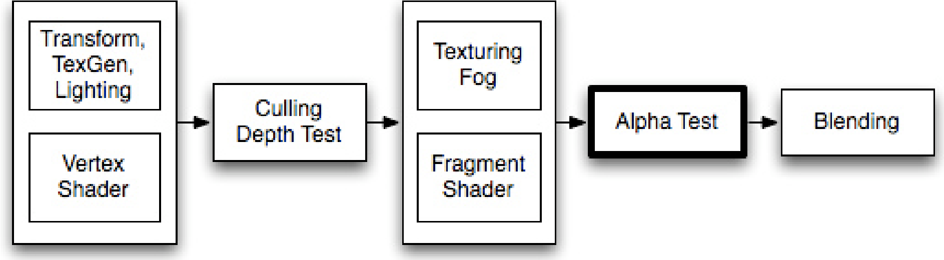
Note that if you use fragment programs, Fog settings of the shader will still be applied. On platforms where there is no fixed function Fog functionality, Unity will patch shaders at runtime to support the requested Fog mode.

Page last updated: 2010-08-18

---

## SL-AlphaTest

The alpha test is a last chance to reject a pixel from being written to the screen.



After the final output color has been calculated, the color can optionally have its alpha value compared to a fixed value. If the test fails, the pixel is not written to the display.

## Syntax

**AlphaTest Off**
: Render all pixels (default).

**AlphaTest *comparison AlphaValue***
: Set up the alpha test to only render pixels whose alpha value is within a certain range.

### Comparison

Comparison is one of the following words:

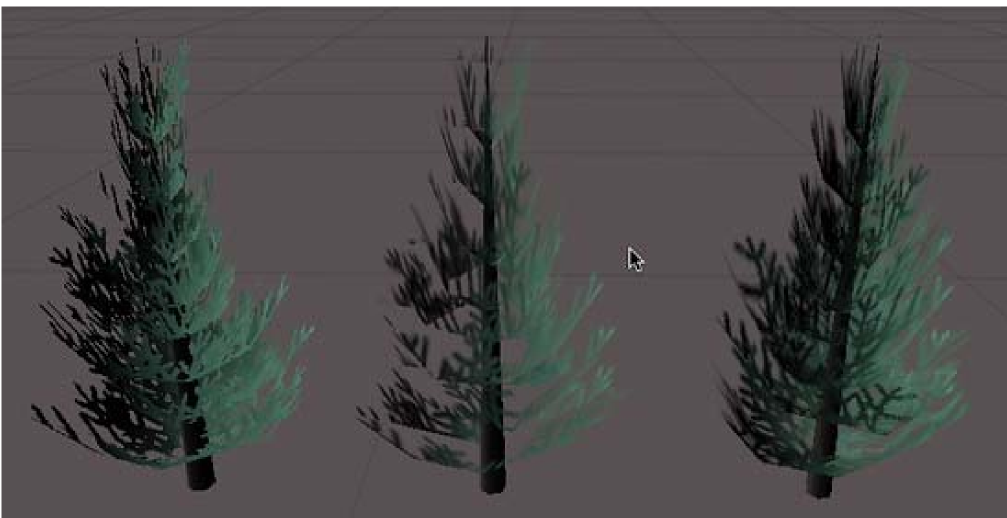
| | |
|---|---|
| **Greater** | Only render pixels whose alpha is greater than *AlphaValue*. |
| **GEqual** | Only render pixels whose alpha is greater than or equal to *AlphaValue*. |
| **Less** | Only render pixels whose alpha value is less than *AlphaValue*. |
| **LEqual** | Only render pixels whose alpha value is less than or equal to from *AlphaValue*. |
| **Equal** | Only render pixels whose alpha value equals *AlphaValue*. |
| **NotEqual** | Only render pixels whose alpha value differs from *AlphaValue*. |
| **Always** | Render all pixels. This is functionally equivalent to *AlphaTest Off*. |
| **Never** | Don't render any pixels. |

### AlphaValue

A floating-point number between 0 and 1. This can also be a variable reference to a float or range property, in which case it should be written using the standard square bracket notation (*[VariableName]*).

## Details

The alpha test is important when rendering concave objects with transparent parts. The graphics card maintains a record of the depth of every pixel written to the screen. If a new pixel is further away than one already rendered, the new pixel is not written to the display. This means that even with Blending, objects will not show through.



In this figure, the tree on the left is rendered using **AlphaTest**. Note how the pixels in it are either completely transparent or opaque. The center tree is rendered using only **Alpha Blending** - notice how transparent parts of nearby branches cover the distant leaves because of the depth buffer. The tree on the right is rendered using the last example shader - which implements a combination of blending and alpha testing to hide any artifacts.

## Examples

The simplest possible example, assign a texture with an alpha channel to it. The object will only be visible where alpha is greater than 0.5

```
Shader "Simple Alpha Test" {
	Properties {
		_MainTex ("Base (RGB) Transparency (A)", 2D) = "" {}
	}
```

```
	SubShader {
		Pass {
			// Only render pixels with an alpha larger than 50%
			AlphaTest Greater 0.5
			SetTexture [_MainTex] { combine texture }
		}
	}
}
```

This is not much good by itself. Let us add some lighting and make the cutoff value tweakable:

```
Shader "Cutoff Alpha" {
	Properties {
		_MainTex ("Base (RGB) Transparency (A)", 2D) = "" {}
		_Cutoff ("Alpha cutoff", Range (0,1)) = 0.5
	}
	SubShader {
		Pass {
			// Use the Cutoff parameter defined above to determine
			// what to render.
			AlphaTest Greater [_Cutoff]
			Material {
				Diffuse (1,1,1,1)
				Ambient (1,1,1,1)
			}
			Lighting On
			SetTexture [_MainTex] { combine texture * primary }
		}
	}
}
```

When rendering plants and trees, many games have the hard edges typical of alpha testing. A way around that is to render the object twice. In the first pass, we use alpha testing to only render pixels that are more than 50% opaque. In the second pass, we alpha-blend the graphic in the parts that were cut away, without recording the depth of the pixel. We might get a bit of confusion as further away branches overwrite the nearby ones, but in practice, that is hard to see as leaves have a lot of visual detail in them.

```
Shader "Vegetation" {
	Properties {
		_Color ("Main Color", Color) = (.5, .5, .5, .5)
		_MainTex ("Base (RGB) Alpha (A)", 2D) = "white" {}
		_Cutoff ("Base Alpha cutoff", Range (0,.9)) = .5
	}
	SubShader {
		// Set up basic lighting
		Material {
			Diffuse [_Color]
			Ambient [_Color]
		}
		Lighting On

		// Render both front and back facing polygons.
		Cull Off

		// first pass:
		//   render any pixels that are more than [_Cutoff] opaque
		Pass {
			AlphaTest Greater [_Cutoff]
			SetTexture [_MainTex] {
```

```
                combine texture * primary, texture
            }
        }

        // Second pass:
        //   render in the semitransparent details.
        Pass {
            // Dont write to the depth buffer
            ZWrite off
            // Don't write pixels we have already written.
            ZTest Less
            // Only render pixels less or equal to the value
            AlphaTest LEqual [_Cutoff]

            // Set up alpha blending
            Blend SrcAlpha OneMinusSrcAlpha

            SetTexture [_MainTex] {
                combine texture * primary, texture
            }
        }
    }
}
```

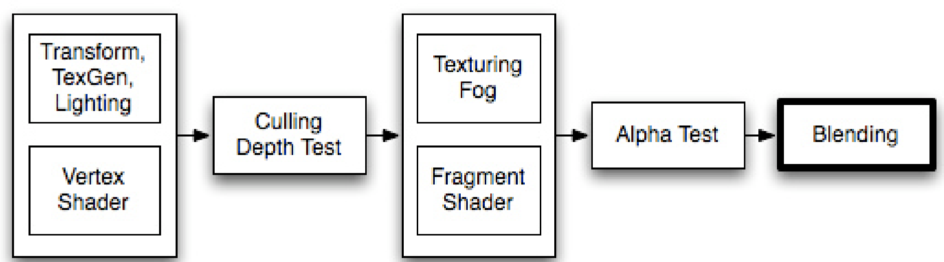
Note that we have some setup inside the SubShader, rather than in the individual passes. Any state set in the SubShader is inherited as defaults in passes inside it.

Page last updated: 2008-04-27

---

## SL-Blend

Blending is used to make transparent objects.




When graphics are rendered, after all shaders have executed and all textures have been applied, the pixels are written to the screen. How they are combined with what is already there is controlled by the Blend command.

### Syntax

**Blend Off**
: Turn off blending

**Blend *SrcFactor DstFactor***
: Configure & enable blending. The generated color is multiplied by the **SrcFactor**. The color already on screen is multiplied by **DstFactor** and the two are added together.

**Blend *SrcFactor DstFactor*, *SrcFactorA DstFactorA***
: Same as above, but use different factors for blending the alpha channel.

**BlendOp *Min | Max | Sub | RevSub***
: Instead of adding blended colors together, do a different operation on them.

### Properties

All following properties are valid for both SrcFactor & DstFactor. **Source** refers to the calculated color, **Destination** is the color already on the screen.

| | |
|---|---|
| **One** | The value of one - use this to let either the source or the destination color come through fully. |
| **Zero** | The value zero - use this to remove either the source or the destination values. |
| **SrcColor** | The value of this stage is multiplied by the source color value. |
| **SrcAlpha** | The value of this stage is multiplied by the source alpha value. |
| **DstColor** | The value of this stage is multiplied by frame buffer source color value. |
| **DstAlpha** | The value of this stage is multiplied by frame buffer source alpha value. |
| **OneMinusSrcColor** | The value of this stage is multiplied by (1 - source color). |
| **OneMinusSrcAlpha** | The value of this stage is multiplied by (1 - source alpha). |
| **OneMinusDstColor** | The value of this stage is multiplied by (1 - destination color). |
| **OneMinusDstAlpha** | The value of this stage is multiplied by (1 - destination alpha). |

### Details

Below are the most common blend types:

```
Blend SrcAlpha OneMinusSrcAlpha     // Alpha blending
Blend One One                       // Additive
Blend OneMinusDstColor One          // Soft Additive
Blend DstColor Zero                 // Multiplicative
Blend DstColor SrcColor             // 2x Multiplicative
```

### Example

Here is a small example shader that adds a texture to whatever is on the screen already:

```
Shader "Simple Additive" {
   Properties {
      _MainTex ("Texture to blend", 2D) = "black" {}
   }
   SubShader {
      Tags { "Queue" = "Transparent" }
      Pass {
         Blend One One
         SetTexture [_MainTex] { combine texture }
      }
   }
}
```

And a more complex one, Glass. This is a two-pass shader:

1. The first pass renders a lit, alpha-blended texture on to the screen. The alpha channel decides the transparency.
2. The second pass renders a reflection cubemap on top of the alpha-blended window, using additive transparency.

```
Shader "Glass" {
   Properties {
      _Color ("Main Color", Color) = (1,1,1,1)
      _MainTex ("Base (RGB) Transparency (A)", 2D) = "white" {}
      _Reflections ("Base (RGB) Gloss (A)", Cube) = "skybox" { TexGen CubeReflect }
   }
   SubShader {
      Tags { "Queue" = "Transparent" }
      Pass {
         Blend SrcAlpha OneMinusSrcAlpha
         Material {
            Diffuse [_Color]
         }
         Lighting On
         SetTexture [_MainTex] {
```

```
            combine texture * primary double, texture * primary
         }
      }
      Pass {
         Blend One One
         Material {
            Diffuse [_Color]
         }
         Lighting On
         SetTexture [_Reflections] {
            combine texture
            Matrix [_Reflection]
         }
      }
   }
}
```

Page last updated: 2012-05-31

---

# SL-PassTags

Passes use tags to tell how and when they expect to be rendered to the rendering engine.

## Syntax

**Tags { "*TagName1*" = "*Value1*" "*TagName2*" = "*Value2*" }**

Specifies **TagName1** to have **Value1**, **TagName2** to have **Value2**. You can have as many tags as you like.

## Details

Tags are basically key-value pairs. Inside a Pass tags are used to control which role this pass has in the lighting pipeline (ambient, vertex lit, pixel lit etc.) and some other options. Note that the following tags recognized by Unity *'must* be inside Pass section and not inside SubShader!

#### LightMode tag

**LightMode** tag defines Pass' role in the lighting pipeline. See render pipeline for details. These tags are rarely used manually; most often shaders that need to interact with lighting are written as Surface Shaders and then all those details are taken care of.

Possible values for LightMode tag are:

- **Always**: Always rendered; no lighting is applied.
- **ForwardBase**: Used in Forward rendering, ambient, main directional light and vertex/SH lights are applied.
- **ForwardAdd**: Used in Forward rendering; additive per-pixel lights are applied, one pass per light.
- **PrepassBase**: Used in Deferred Lighting, renders normals & specular exponent.
- **PrepassFinal**: Used in Deferred Lighting, renders final color by combining textures, lighting & emission.
- **Vertex**: Used in Vertex Lit rendering when object is not lightmapped; all vertex lights are applied.
- **VertexLMRGBM**: Used in Vertex Lit rendering when object is lightmapped; on platforms where lightmap is RGBM encoded.
- **VertexLM**: Used in Vertex Lit rendering when object is lightmapped; on platforms where lightmap is double-LDR encoded (generally mobile platforms & old dekstop GPUs).
- **ShadowCaster**: Renders object as shadow caster.
- **ShadowCollector**: Gathers object's shadows into screen-space buffer for Forward rendering path.

#### RequireOptions tag

A pass can indicate that it should only be rendered when some external conditions are met. This is done by using **RequireOptions** tag, whose value is a string of space separated options. Currently the options supported by Unity are:

- **SoftVegetation**: Render this pass only if Soft Vegetation is on in Quality Settings.

### See Also

SubShaders can be given Tags as well, see SubShader Tags.

Page last updated: 2012-01-26

---

# SL-Name

### Syntax

**Name "*PassName*"**

Gives the *PassName* name to the current pass.

### Details

A pass can be given a name so that a UsePass command can reference it.

Page last updated: 2008-06-16

---

# SL-BindChannels

**BindChannels** command allows you to specify how vertex data maps to the graphics hardware.

*BindChannels has no effect when programmable vertex shaders are used, as in that case bindings are controlled by vertex shader inputs.*

By default, Unity figures out the bindings for you, but in some cases you want custom ones to be used.

For example you could map the primary UV set to be used in the first texture stage and the secondary UV set to be used in the second texture stage; or tell the hardware that vertex colors should be taken into account.

### Syntax

**BindChannels { Bind "*source*", *target* }**

Specifies that vertex data *source* maps to hardware *target*.

**Source** can be one of:

- **Vertex**: vertex position
- **Normal**: vertex normal
- **Tangent**: vertex tangent
- **Texcoord**: primary UV coordinate
- **Texcoord1**: secondary UV coordinate
- **Color**: per-vertex color

**Target** can be one of:

- **Vertex**: vertex position
- **Normal**: vertex normal
- **Tangent**: vertex tangent
- **Texcoord0**, **Texcoord1**, ...: texture coordinates for corresponding texture stage
- **Texcoord**: texture coordinates for all texture stages
- **Color**: vertex color

### Details

Unity places some restrictions on which sources can be mapped to which targets. Source and target must match for **Vertex**, **Normal**, **Tangent** and **Color**. Texture coordinates from the mesh (**Texcoord** and **Texcoord1**) can be mapped into texture coordinate targets (**Texcoord** for all texture stages, or **TexcoordN** for a specific stage).

There are two typical use cases for BindChannels:

- Shaders that take vertex colors into account.
- Shaders that use two UV sets.

### Examples

```
// Maps the first UV set to the first texture stage
// and the second UV set to the second texture stage
BindChannels {
   Bind "Vertex", vertex
   Bind "texcoord", texcoord0
   Bind "texcoord1", texcoord1
}
```

```
// Maps the first UV set to all texture stages
// and uses vertex colors
BindChannels {
   Bind "Vertex", vertex
   Bind "texcoord", texcoord
   Bind "Color", color
}
```

Page last updated: 2008-04-27

---

# SL-UsePass

The UsePass command uses named passes from another shader.

### Syntax

**UsePass** "*Shader/Name*"

Inserts all passes with a given name from a given shader. *Shader/Name* contains the name of the shader and the name of the pass, separated by a slash character. Note that only first supported subshader is taken into account.

### Details

Some of the shaders could reuse existing passes from other shaders, reducing code duplication. For example, in most pixel lit shaders the ambient or vertex lighting passes are the same as in the corresponding VertexLit shaders. The UsePass command does just that - it includes a given pass from another shader. As an example the following command uses the pass with the name *"BASE"* from the builtin *Specular* shader:

```
UsePass "Specular/BASE"
```

In order for UsePass to work, a name must be given to the pass one wishes to use. The Name command inside the pass gives it a name:

```
Name "MyPassName"
```

Note that internally all pass names are uppercased, so UsePass must refer to the name **in uppercase**.

Page last updated: 2012-04-25

---

# SL-GrabPass

GrabPass is a special passtype - it grabs the contents of the screen where the object is about to be drawn into a texture. This

texture can be used in subsequent passes to do advanced image based effects.

### Syntax

The GrabPass belongs inside a subshader. It can take two forms:

- Just `GrabPass { }` will grab current screen contents into a texture. The texture can be accessed in further passes by `_GrabTexture` name. Note: this form of grab pass will do the expensive screen grabbing operation for each object that uses it!
- `GrabPass { "`*`TextureName`*`" }` will grab screen contents into a texture, but will only do that once per frame for the first object that uses the given texture name. The texture can be accessed in further passes by the given texture name. This is a more performant way when you have multiple objects using grab pass in the scene.

Additionally, GrabPass can use Name and Tags commands.

### Example

Here is an expensive way to invert the colors of what was rendered before:

```
Shader "GrabPassInvert" {
   SubShader {
      // Draw ourselves after all opaque geometry
      Tags { "Queue" = "Transparent" }

      // Grab the screen behind the object into _GrabTexture
      GrabPass { }

      // Render the object with the texture generated above, and invert it's colors
      Pass {
         SetTexture [_GrabTexture] { combine one-texture }
      }
   }
}
```

This shader has two passes: First pass grabs whatever is behind the object at the time of rendering, then applies that in the second pass. Now of course, the same effect could be achieved much cheaper using an invert blend mode.

### See Also

- Regular Pass command

Page last updated: 2012-07-10

---

## SL-SubshaderTags

Subshaders use tags to tell how and when they expect to be rendered to the rendering engine.

### Syntax

**Tags { "*TagName1*" = "*Value1*" "*TagName2*" = "*Value2*" }**

Specifies **TagName1** to have **Value1**, **TagName2** to have **Value2**. You can have as many tags as you like.

### Details

Tags are basically key-value pairs. Inside a SubShader tags are used to determine rendering order and other parameters of a subshader. Note that the following tags recognized by Unity **must** be inside SubShader section and not inside Pass!

#### Rendering Order - Queue tag

You can determine in which order your objects are drawn using the *Queue* tag. A Shader decides which render queue its objects belong to, this way any Transparent shaders make sure they are drawn after all opaque objects and so on.

There are four pre-defined render queues, but there can be more queues in between the predefined ones. The predefined queues are:

- **Background** - this render queue is rendered before any others. It is used for skyboxes and the like.
- **Geometry** *(default)* - this is used for most objects. Opaque geometry uses this queue.
- **AlphaTest** - alpha tested geometry uses this queue. It's a separate queue from **Geometry** one since it's more efficient to render alpha-tested objects after all solid ones are drawn.
- **Transparent** - this render queue is rendered after *Geometry* and **AlphaTest**, in back-to-front order. Anything alpha-blended (i.e. shaders that don't write to depth buffer) should go here (glass, particle effects).
- **Overlay** - this render queue is meant for overlay effects. Anything rendered last should go here (e.g. lens flares).

```
Shader "Transparent Queue Example" {
    SubShader {
      Tags {"Queue" = "Transparent" }
      Pass {
         // rest of the shader body...
      }
   }
}
```

*An example illustrating how to render something in the transparent queue*

**Geometry** render queue optimizes the drawing order of the objects for best performance. All other render queues sort objects by distance, starting rendering from the furthest ones and ending with the closest ones.

For special uses in-between queues can be used. Internally each queue is represented by integer index; **Background** is 1000, **Geometry** is 2000, **AlphaTest** is 2450, **Transparent** is 3000 and **Overlay** is 4000. If a shader uses a queue like this:

```
Tags { "Queue" = "Geometry+1" }
```

This will make the object be rendered after all opaque objects, but before transparent objects, as render queue index will be 2001 (geometry plus one). This is useful in situations where you want some objects be always drawn between other sets of objects. For example, in most cases transparent water should be drawn after opaque objects but before transparent objects.

#### RenderType tag

**RenderType** tag categorizes shaders into several predefined groups, e.g. is is an opaque shader, or an alpha-tested shader etc. This is used by Shader Replacement and in some cases used to produce camera's depth texture.

#### IgnoreProjector tag

If **IgnoreProjector** tag is given and has a value of "True", then an object that uses this shader will not be affected by Projectors. This is mostly useful on semitransparent objects, because there is no good way for Projectors to affect them.

### See Also

Passes can be given Tags as well, see Pass Tags.

Page last updated: 2012-06-21

---

# SL-Fallback

After all Subshaders a Fallback can be defined. It basically says "if none of subshaders can run on this hardware, try using the ones from another shader".

### Syntax

**Fallback "name"**
: Fallback to shader with a given *name*.

**Fallback Off**
: Explicitly state that there is no fallback and no warning should be printed, even if no subshaders can run on this hardware.

### Details

A fallback statement has the same effect as if all subshaders from the other shader would be inserted into its place.

### Example

```
Shader "example" {
    // properties and subshaders here...
    Fallback "otherexample"
}
```

Page last updated: 2008-04-27

---

## SL-Other

### Category

**Category** is a logical grouping of any commands below it. This is mostly used to "inherit" rendering state. For example, your shader might have multiple subshaders, and each of them requires fog to be off, blending set to additive, etc. You can use Category for that:

```
Shader "example" {
Category {
    Fog { Mode Off }
    Blend One One
    SubShader {
        // ...
    }
    SubShader {
        // ...
    }
    // ...
}
}
```

Category block only affects shader parsing, it's exactly the same as "pasting" any state set inside Category into all blocks below it. It does not affect shader execution speed at all.

Page last updated: 2009-07-24

---

## SL-AdvancedTopics

Read those to improve your ShaderLab-fu!

- Unity's Rendering Pipeline
- Performance Tips when Writing Shaders
- Rendering with Replaced Shaders
- Using Depth Textures
- Camera's Depth Texture
- Platform Specific Rendering Differences
- Shader Level of Detail

Page last updated: 2008-06-23

---

# SL-RenderPipeline

Shaders define both how an object looks by itself (its material properties) and how it reacts to the light. Because lighting calculations must be built into the shader, and there are many possible light & shadow types, writing quality shaders that "just work" would be an involved task. To make it easier, Unity 3 introduces Surface Shaders, where all the lighting, shadowing, lightmapping, forward vs. deferred lighting things are taken care of automatically.

This document describes the pecularities of Unity's lighting & rendering pipeline and what happens behind the scenes of Surface Shaders.

### Rendering Paths

How lighting is applied and which Passes of the shader are used depends on which Rendering Path is used. Each pass in a shader communicates its lighting type via Pass Tags.

- In Deferred Lighting, `PrepassBase` and `PrepassFinal` passes are used.
- In Forward Rendering, `ForwardBase` and `ForwardAdd` passes are used.
- In Vertex Lit, `Vertex`, `VertexLMRGBM` and `VertexLM` passes are used.
- In any of the above, to render Shadows, `ShadowCaster` and `ShadowCollector` passes are used.

### Deferred Lighting path

`PrepassBase` pass renders normals & specular exponent; `PrepassFinal` pass renders final color by combining textures, lighting & emissive material properties. All regular in-scene lighting is done separately in screen-space. See Deferred Lighting for details.

### Forward Rendering path

`ForwardBase` pass renders ambient, lightmaps, main directional light and not important (vertex/SH) lights at once. `ForwardAdd` pass is used for any additive per-pixel lights; one invocation per object illuminated by such light is done. See Forward Rendering for details.

If forward rendering is used, but a shader does not have forward-suitable passes (i.e. neither `ForwardBase` nor `ForwardAdd` pass types are present), then that object is rendered just like it would in Vertex Lit path, see below.

### Vertex Lit Rendering path

Since vertex lighting is most often used on platforms that do not support programmable shaders, Unity can't create multiple shader permutations internally to handle lightmapped vs. non-lightmapped cases. So to handle lightmapped and non-lightmapped objects, multiple passes have to be written explicitly.

- `Vertex` pass is used for non-lightmapped objects. All lights are rendered at once, using a fixed function OpenGL/Direct3D lighting model (Blinn-Phong)
- `VertexLMRGBM` pass is used for lightmapped objects, when lightmaps are RGBM encoded (this happens on most desktops and consoles). No realtime lighting is applied; pass is expected to combine textures with a lightmap.
- `VertexLMM` pass is used for lightmapped objects, when lightmaps are double-LDR encoded (this happens on mobiles and old desktops). No realtime lighting is applied; pass is expected to combine textures with a lightmap.

Page last updated: 2012-07-31

---

# SL-ShaderPerformance

### Use Common sense ;)

Compute only things that you need; anything that is not actually needed can be eliminated. For example, supporting per-material color is nice to make a shader more flexible, but if you always leave that color set to white then it's useless computations performed for each vertex or pixel rendered on screen.

Another thing to keep in mind is frequency of computations. Usually there are many more pixels rendered (hence their pixel shaders executed) than there are vertices (vertex shader executions); and more vertices than objects being rendered. So generally if you can, move computations out of pixel shader into the vertex shader; or out of shaders completely and set the values once from a script.

### Less Generic Surface Shaders

Surface Shaders are great for writing shaders that interact with lighting. However, their default options are tuned for "general case". In many cases, you can tweak them to make shaders run faster or at least be smaller:

- `approxview` directive for shaders that use view direction (i.e. Specular) will make view direction be normalized per-vertex instead of per-pixel. This is approximate, but often good enough.
- `halfasview` for Specular shader types is even faster. Half-vector (halfway between lighting direction and view vector) will be computed and normalized per vertex, and lighting function will already receive half-vector as a parameter instead of view vector.
- `noforwardadd` will make a shader fully support only one directional light in Forward rendering. The rest of the lights can still have an effect as per-vertex lights or spherical harmonics. This is great to make shader smaller and make sure it always renders in one pass, even with multiple lights present.
- `noambient` will disable ambient lighting and spherical harmonics lights on a shader. This can be slightly faster.

### Precision of computations

When writing shaders in Cg/HLSL, there are three basic number types: `float`, `half` and `fixed` (as well as vector & matrix variants of them, e.g. half3 and float4x4):

- `float`: high precision floating point. Generally 32 bits, just like float type in regular programming languages.
- `half`: medium precision floating point. Generally 16 bits, with a range of -60000 to +60000 and 3.3 decimal digits of precision.
- `fixed`: low precision fixed point. Generally 11 bits, with a range of -2.0 to +2.0 and 1/256th precision.

Use lowest precision that is possible; this is especially important on mobile platforms like iOS and Android. Good rules of thumb are:

- For colors and unit length vectors, use `fixed`.
- For others, use `half` if range and precision is fine; otherwise use `float`.

On mobile platforms, the key is to ensure as much as possible stays in low precision in the fragment shader. On most mobile GPUs, applying swizzles to low precision (fixed/lowp) types is costly; converting between fixed/lowp and higher precision types is quite costly as well.

### Alpha Testing

Fixed function AlphaTest or it's programmable equivalent, `clip()`, has different performance characteristics on different platforms:

- Generally it's a small advantage to use it to cull out totally transparent pixels on most platforms.
- However, on PowerVR GPUs found in iOS and some Android devices, alpha testing is expensive. Do not try to use it as "performance optimization" there, it will be slower.

### Color Mask

On some platforms (mostly mobile GPUs found in iOS and Android devices), using ColorMask to leave out some channels (e.g. `ColorMask RGB`) can be expensive, so only use it if really necessary.

Page last updated: 2011-01-13

---

# SL-ShaderReplacement

Some rendering effects require rendering a scene with a different set of shaders. For example, good edge detection would need a texture with scene normals, so it could detect edges where surface orientations differ. Other effects might need a texture with scene depth, and so on. To achieve this, it is possible to render the scene with replaced shaders of all objects.

Shader replacement is done from scripting using Camera.RenderWithShader or Camera.SetReplacementShader functions. Both functions take a **shader** and a **replacementTag**.

It works like this: the camera renders the scene as it normally would. the objects still use their materials, but the actual shader that ends up being used is changed:

- If **replacementTag** is empty, then all objects in the scene are rendered with the given replacement shader.
- If **replacementTag** is not empty, then for each object that would be rendered:
  - The real object's shader is queried for the tag value.
  - If it does not have that tag, object is **not rendered**.
  - A subshader is found in the replacement shader that has a given tag with the found value. If no such subshader is found, object is **not rendered**.
  - Now that subshader is used to render the object.

So if all shaders would have, for example, a "RenderType" tag with values like "Opaque", "Transparent", "Background", "Overlay", you could write a replacement shader that only renders solid objects by using one subshader with RenderType=Solid tag. The other tag types would not be found in the replacement shader, so the objects would be not rendered. Or you could write several subshaders for different "RenderType" tag values. Incidentally, all built-in Unity shaders have a "RenderType" tag set.

### Shader replacement tags in built-in Unity shaders

All built-in Unity shaders have a "**RenderType**" tag set that can be used when rendering with replaced shaders. Tag values are the following:

- **Opaque**: most of the shaders (Normal, Self Illuminated, Reflective, terrain shaders).
- **Transparent**: most semitransparent shaders (Transparent, Particle, Font, terrain additive pass shaders).
- **TransparentCutout**: masked transparency shaders (Transparent Cutout, two pass vegetation shaders).
- **Background**: Skybox shaders.
- **Overlay**: GUITexture, Halo, Flare shaders.
- **TreeOpaque**: terrain engine tree bark.
- **TreeTransparentCutout**: terrain engine tree leaves.
- **TreeBillboard**: terrain engine billboarded trees.
- **Grass**: terrain engine grass.
- **GrassBillboard**: terrain engine billboarded grass.

### Built-in scene depth/normals texture

A Camera has a built-in capability to render depth or depth+normals texture, if you need that in some of your effects. See Camera Depth Texture page. Note that in some cases (depending on the hardware), the depth and depth+normals textures can internally be rendered using shader replacement. So it is important to have the correct "**RenderType**" tag in your shaders.

Page last updated: 2012-06-21

---

# SL-DepthTextures

It is possible to create Render Textures where each pixel contains a high precision "depth" value (see RenderTextureFormat.Depth). This is mostly used when some effects need scene's depth to be available (for example, soft particles, screen space ambient occlusion, translucency would all need scene's depth).

Pixel values in the depth texture range from 0 to 1 with a nonlinear distribution. Precision is usually 24 or 16 bits, depending on depth buffer used. When reading from depth texture, a high precision value in 0..1 range is returned. If you need to get distance from the camera, or otherwise linear value, you should compute that manually.

Depth textures in Unity are implemented differently on different platforms.

- On Direct3D 9 (Windows), depth texture is either a native depth buffer, or a single channel 32 bit floating point texture ("R32F" Direct3D format).
  - Graphics card must support either native depth buffer (INTZ format) or floating point render textures in order for them to work.
  - When rendering into the depth texture, fragment program must output the value needed.
  - When reading from depth texture, red component of the color contains the high precision value.
- On OpenGL (Mac OS X), depth texture is the native OpenGL depth buffer (see ARB_depth_texture).
  - Graphics card must support OpenGL 1.4 or ARB_depth_texture extension.
  - Depth texture corresponds to Z buffer contents that are rendered, it **does not** use the result from the fragment program.

- OpenGL ES 2.0 (iOS/Android) is very much like OpenGL above.
  - GPU must support GL_OES_depth_texture extension.
- Direct3D 11 (Windows) has native depth texture capability just like OpenGL.
- Flash (Stage3D) uses a color-encoded depth texture to emulate the high precision required for it.

### Using depth texture helper macros

Most of the time depth textures are used to render depth from the camera. `UnityCG.cginc` include file contains some macros to deal with the above complexity in this case:

- **UNITY_TRANSFER_DEPTH(o)**: computes eye space depth of the vertex and outputs it in **o** (which must be a float2). Use it in a vertex program when rendering into a depth texture. On platforms with native depth textures this macro does nothing at all, because Z buffer value is rendered implicitly.
- **UNITY_OUTPUT_DEPTH(i)**: returns eye space depth from **i** (which must be a float2). Use it in a fragment program when rendering into a depth texture. On platforms with native depth textures this macro always returns zero, because Z buffer value is rendered implicitly.
- **COMPUTE_EYEDEPTH(i)**: computes eye space depth of the vertex and outputs it in **o**. Use it in a vertex program when **not** rendering into a depth texture.
- **DECODE_EYEDEPTH(i)**: given high precision value from depth texture **i**, returns corresponding eye space depth. This macro just returns **i*FarPlane** on Direct3D. On platforms with native depth textures it linearizes and expands the value to match camera's range.

For example, this shader would render depth of its objects:

```
Shader "Render Depth" {
SubShader {
    Tags { "RenderType"="Opaque" }
    Pass {
        Fog { Mode Off }
CGPROGRAM
#pragma vertex vert
#pragma fragment frag
#include "UnityCG.cginc"

struct v2f {
    float4 pos : SV_POSITION;
    float2 depth : TEXCOORD0;
};

v2f vert (appdata_base v) {
    v2f o;
    o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
    UNITY_TRANSFER_DEPTH(o.depth);
    return o;
}

half4 frag(v2f i) : COLOR {
    UNITY_OUTPUT_DEPTH(i.depth);
}
ENDCG
    }
}
}
```

Page last updated: 2012-09-04

---

# SL-CameraDepthTexture

In Unity a Camera can generate a depth or depth+normals texture. This is a minimalistic G-buffer texture that can be used for

post-processing effects or to implement custom lighting models (e.g. light pre-pass). Camera actually builds the depth texture using Shader Replacement feature, so it's entirely possible to do that yourself, in case you need a different G-buffer setup.

Camera's depth texture can be turned on using Camera.depthTextureMode variable from script.

There are two possible depth texture modes:

- **DepthTextureMode.Depth**: a depth texture.
- **DepthTextureMode.DepthNormals**: depth and view space normals packed into one texture.

### DepthTextureMode.Depth texture

This builds a screen-sized depth texture.

### DepthTextureMode.DepthNormals texture

This builds a screen-sized 32 bit (8 bit/channel) texture, where view space normals are encoded into R&G channels, and depth is encoded in B&A channels. Normals are encoded using Stereographic projection, and depth is 16 bit value packed into two 8 bit channels.

`UnityCG.cginc` include file has a helper function `DecodeDepthNormal` to decode depth and normal from the encoded pixel value. Returned depth is in 0..1 range.

For examples on how to use the depth and normals texture, please refer to the EdgeDetection image effect in the Shader Replacement example project or SSAO Image Effect.

### Tips & Tricks

When implementing complex shaders or Image Effects, keep Rendering Differences Between Platforms in mind. In particular, using depth texture in an Image Effect often needs special handling on Direct3D + Anti-Aliasing.

In some cases, the depth texture might come directly from the native Z buffer. If you see artifacts in your depth texture, make sure that the shaders that use it **do not** write into the Z buffer (use ZWrite Off).

### Under the hood

Depth texture can come directly from the actual depth buffer, or be rendered in a separate pass, depending on the rendering path used and the hardware. When the depth texture is rendered in a separate pass, this is done through Shader Replacement. Hence it is important to have correct "**RenderType**" tag in your shaders.

Page last updated: 2012-09-04

---

## SL-PlatformDifferences

Unity runs on various platforms, and in some cases there are differences in how things behave. Most of the time Unity hides the differences from you, but sometimes you can still bump into them.

### Render Texture Coordinates

Vertical texture coordinate conventions differ between Direct3D, OpenGL and OpenGL ES:

- In Direct3D, the coordinate is zero at the top, and increases downwards.
- In OpenGL and OpenGL ES, the coordiante is zero at the bottom, and increases upwards.

Most of the time this does not really matter, except when rendering into a Render Texture. In that case, Unity internally flips rendering upside down when rendering into a texture on Direct3D, so that the conventions match between the platforms.

One case where this does not happen, is when Image Effects and Anti-Aliasing is used. In this case, Unity renders to screen to get anti-aliasing, and then "resolves" rendering into a RenderTexture for further processing with an Image Effect. The resulting source texture for an image effect is *not flipped upside down* on Direct3D (unlike all other Render Textures).

If your Image Effect is a simple one (processes one texture at a time), this does not really matter, because Graphics.Blit takes care of that.

However, **if you're processing more than one RenderTexture together** in your Image Effect, most likely they will come out at different vertical orientations (only in Direct3D-like platforms, and only when anti-aliasing is used). You need to manually "flip" the screen texture upside down in your vertex shader, like this:

```
// On D3D when AA is used, the main texture & scene depth texture
// will come out in different vertical orientations.
// So flip sampling of the texture when that is the case (main texture
// texel size will have negative Y).
#if UNITY_UV_STARTS_AT_TOP
if (_MainTex_TexelSize.y < 0)
      uv.y = 1-uv.y;
#endif
```

Check out Edge Detection scene in Shader Replacement sample project for an example of this. Edge detection there uses both screen texture and Camera's Depth+Normals texture.

### AlphaTest and programmable shaders

Some platforms, most notably mobile (OpenGL ES 2.0) and Direct3D 11, do not have fixed function alpha testing functionality. When you are using programmable shaders, it's advised to use Cg/HLSL `clip()` function in the pixel shader instead.

### Direct3D 11 shader compiler is more picky about syntax

Direct3D 9 and OpenGL use NVIDIA's Cg to compile shaders, but Direct3D 11 (and Xbox 360) use Microsoft's HLSL shader compiler. HLSL compiler is more picky about various subtle shader errors. For example, it won't accept function output values that aren't initialized properly.

Most common places where you'd run into this:

- Surface shader vertex modifier that has an "out" parameter. Make sure to initialize the output like this:

```
void vert (inout appdata_full v, out Input o)
{
  UNITY_INITIALIZE_OUTPUT(Input,o);
  // ...
}
```

- Partially initialized values, e.g. a function returning float4, but the code only sets .xyz values of it. Make sure to set all values, or change to float3 if you only need those.

### Using OpenGL Shading Language (GLSL) shaders with OpenGL ES 2.0

OpenGL ES 2.0 provides only limited native support for OpenGL Shading Language (GLSL), for instance OpenGL ES 2.0 layer provides no built-in parameters to the shader.

Unity implements built-in parameters for you exactly in the same way as OpenGL does, however following built-in parameters are missing:

- **gl_ClipVertex**
- **gl_SecondaryColor**
- **gl_DepthRange**
- **halfVector** property of the **gl_LightSourceParameters** structure
- **gl_FrontFacing**
- **gl_FrontLightModelProduct**
- **gl_BackLightModelProduct**
- **gl_BackMaterial**
- **gl_Point**
- **gl_PointSize**
- **gl_ClipPlane**
- **gl_EyePlaneR**, **gl_EyePlaneS**, **gl_EyePlaneT**, **gl_EyePlaneQ**
- **gl_ObjectPlaneR**, **gl_ObjectPlaneS**, **gl_ObjectPlaneT**, **gl_ObjectPlaneQ**
- **gl_Fog**

### iPad2 and MSAA and alpha-blended geometry

There is a bug in apple driver resulting in artifacts when MSAA is enabled and alpha-blended geometry is drawn with non RGBA colorMask. To prevent artifacts we force RGBA colorMask when this configuration is encountered, though it will render built-in Glow FX unusable (as it needs DST_ALPHA for intensity value). Also, please update your shaders if you wrote them yourself (see "Render Setup -> ColorMask" in Pass Docs).

Page last updated: 2012-11-16

---

# SL-ShaderLOD

Shader Level of Detail (LOD) works by only using shaders or subshaders that have their LOD value less than a given number.

By default, allowed LOD level is infinite, that is, all shaders that are supported by the user's hardware can be used. However, in some cases you might want to drop shader details, even if the hardware can support them. For example, some cheap graphics cards might support all the features, but are too slow to use them. So you may want to not use parallax normal mapping on them.

Shader LOD can be either set per individual shader (using Shader.maximumLOD), or globally for all shaders (using Shader.globalMaximumLOD).

In your custom shaders, use **LOD** command to set up LOD value for any subshader.

Built-in shaders in Unity have their LODs set up this way:

- VertexLit kind of shaders = 100
- Decal, Reflective VertexLit = 150
- Diffuse = 200
- Diffuse Detail, Reflective Bumped Unlit, Reflective Bumped VertexLit = 250
- Bumped, Specular = 300
- Bumped Specular = 400
- Parallax = 500
- Parallax Specular = 600

Page last updated: 2010-09-25

---

# SL-BuiltinValues

Unity provides a handful of builtin values for your shaders: things like current object's transformation matrices, time etc.

You just use them in ShaderLab like you'd use any other property, the only difference is that you don't have to declare it somewhere - they are "built in".

Using them in programmable shaders requires including **UnityCG.cginc** file.

## Transformations

**float4x4 UNITY_MATRIX_MVP**
    Current model*view*projection matrix
**float4x4 UNITY_MATRIX_MV**
    Current model*view matrix
**float4x4 UNITY_MATRIX_P**
    Current projection matrix
**float4x4 UNITY_MATRIX_T_MV**
    Transpose of model*view matrix
**float4x4 UNITY_MATRIX_IT_MV**
    Inverse transpose of model*view matrix
**float4x4 UNITY_MATRIX_TEXTURE0 to UNITY_MATRIX_TEXTURE3**

Texture transformation matrices

**float4x4 _Object2World**
: Current model matrix

**float4x4 _World2Object**
: Inverse of current world matrix

**float3 _WorldSpaceCameraPos**
: World space position of the camera

**float4 unity_Scale**
: `xyz` components unused; `.w` contains scale for uniformly scaled objects.

**Lighting**

In plain ShaderLab, you access the following properties by appending zero at the end: e.g. the light's model*light color is `_ModelLightColor0`. In Cg shaders, they are exposed as arrays with a single element, so the same in Cg is `_ModelLightColor[0]`.

| Name | Type | Value |
|---|---|---|
| _ModelLightColor | float4 | Material's Main * Light color |
| _SpecularLightColor | float4 | Material's Specular * Light color |
| _ObjectSpaceLightPos | float4 | Light's position in object space. *w* component is 0 for directional lights, 1 for other lights |
| _Light2World | float4x4 | Light to World space matrix |
| _World2Light | float4x4 | World to Light space matrix |
| _Object2Light | float4x4 | Object to Light space matrix |

### Various

- **float4 _Time** : Time (t/20, t, t*2, t*3), use to animate things inside the shaders
- **float4 _SinTime** : Sine of time: (t/8, t/4, t/2, t)
- **float4 _CosTime** : Cosine of time: (t/8, t/4, t/2, t)
- **float4 _ProjectionParams** :
  `x` is 1.0 or -1.0, negative if currently rendering with a flipped projection matrix
  `y` is camera's near plane
  `z` is camera's far plane
  `w` is 1/FarPlane.
- **float4 _ScreenParams** :
  `x` is current render target width in pixels
  `y` is current render target height in pixels
  `z` is 1.0 + 1.0/width
  `w` is 1.0 + 1.0/height

Page last updated: 2012-09-04

---

# Scripting Concepts

- Layers
  - Layer-Based Collision Detection.
- What is a Tag?
- Rigidbody Sleeping

Page last updated: 2007-11-16

---

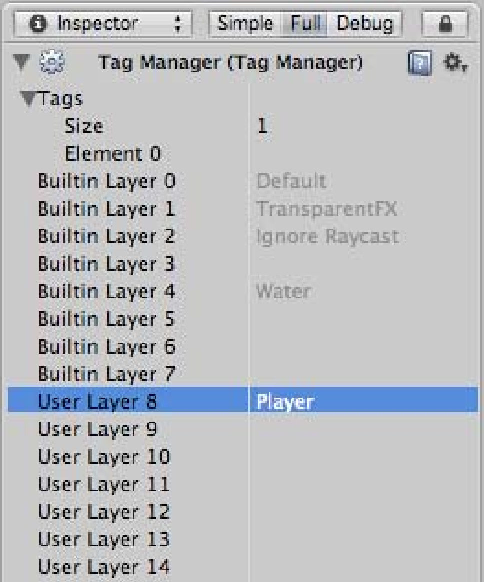
# Layers

**Layers** are most commonly used by **Cameras** to render only a part of the scene, and by **Lights** to illuminate only parts of the scene. But they can also used by raycasting to selectively ignore colliders or to create collisions.

### Creating Layers

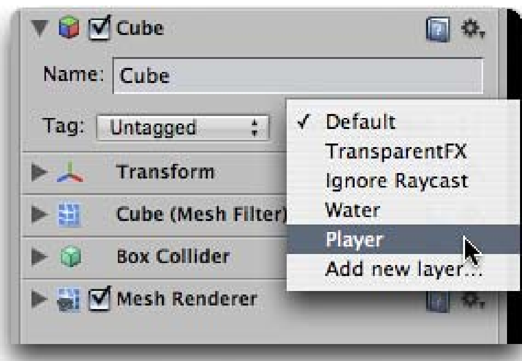
The first step is to create a new layer, which we can then assign to a **GameObject**. To create a new layer, open the Edit menu and select **Project Settings->Tags**.

We create a new layer in one of the empty User Layers. We choose layer 8.




### Assigning Layers

Now that you have created a new layer, you have to assign the layer to one of the game objects.

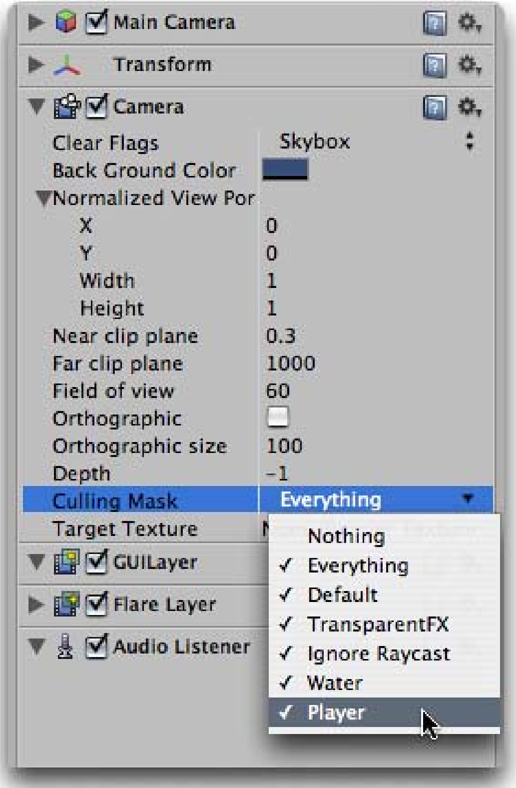



In the tag manager we assigned the Player layer to be in layer 8.

### Drawing only a part of the scene with the camera's culling mask

Using the camera's culling mask, you can selectively render objects which are in one particular layer. To do this, select the camera that should selectively render objects.

Modify the culling mask by checking or unchecking layers in the culling mask property.




### Casting Rays Selectively

Using layers you can cast rays and ignore colliders in specific layers. For example you might want to cast a ray only against the player layer and ignore all other colliders.

The Physics.Raycast function takes a bitmask, where each bit determines if a layer will be ignored or not. If all bits in the layerMask are on, we will collide against all colliders. If the layerMask = 0, we will never find any collisions with the ray.

```
// JavaScript example.

// bit shift the index of the layer to get a bit mask
var layerMask = 1 << 8;
// Does the ray intersect any objects which are in the player layer.
if (Physics.Raycast (transform.position, Vector3.forward, Mathf.Infinity, layerMask))
    print ("The ray hit the player");


// C# example.

int layerMask = 1 << 8;

// Does the ray intersect any objects which are in the player layer.
if (Physics.Raycast(transform.position, Vector3.forward, Mathf.Infinity, layerMask))
    Debug.Log("The ray hit the player");
```

In the real world you want to do the inverse of that however. We want to cast a ray against all colliders except those in the Player layer.

```
// JavaScript example.
function Update () {
  // Bit shift the index of the layer (8) to get a bit mask
  var layerMask = 1 << 8;
  // This would cast rays only against colliders in layer 8.
```

```
  // But instead we want to collide against everything except layer 8. The ~ operator does this, it inverts a bitmask.
  layerMask = ~layerMask;

  var hit : RaycastHit;
  // Does the ray intersect any objects excluding the player layer
  if (Physics.Raycast (transform.position, transform.TransformDirection (Vector3.forward), hit, Mathf.Infinity, layerMask)) {
    Debug.DrawRay (transform.position, transform.TransformDirection (Vector3.forward) * hit.distance, Color.yellow);
    print ("Did Hit");
  } else {
    Debug.DrawRay (transform.position, transform.TransformDirection (Vector3.forward) *1000, Color.white);
    print ("Did not Hit");
  }
}


// C# example.
void Update () {
    // Bit shift the index of the layer (8) to get a bit mask
    int layerMask = 1 << 8;

    // This would cast rays only against colliders in layer 8.
    // But instead we want to collide against everything except layer 8. The ~ operator does this, it inverts a bitmask.
    layerMask = ~layerMask;

    RaycastHit hit;
    // Does the ray intersect any objects excluding the player layer
    if (Physics.Raycast(transform.position, transform.TransformDirection (Vector3.forward), out hit, Mathf.Infinity, layerMask))
        Debug.DrawRay(transform.position, transform.TransformDirection (Vector3.forward) * hit.distance, Color.yellow);
        Debug.Log("Did Hit");
    } else {
        Debug.DrawRay(transform.position, transform.TransformDirection (Vector3.forward) *1000, Color.white);
        Debug.Log("Did not Hit");
    }
}
```

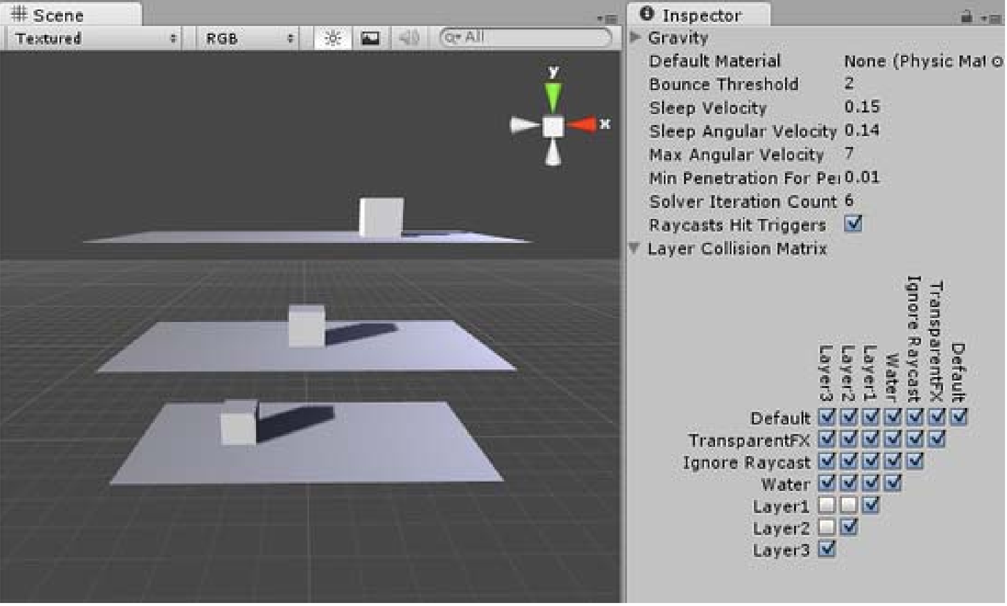
When you don't pass a layerMask to the Raycast function, it will only ignore colliders that use the IgnoreRaycast layer. This is the easiest way to ignore some colliders when casting a ray.

Page last updated: 2012-05-28

---

## Layer Based Collision detection

In Unity 3.x we introduce Layer-Based collision detection, which is a way to make **Game Objects** collide with another specific **Game Objects** that are tied up to specific layers.

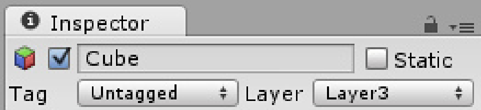



*Objects Colliding with their own layer.*

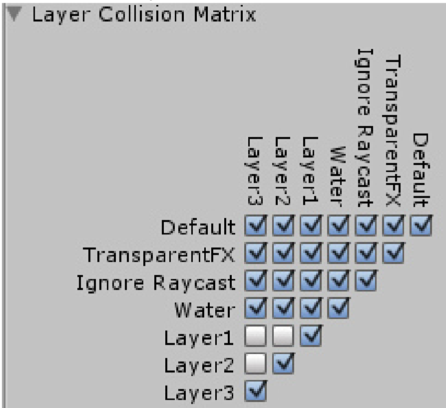
In the image above you can see 6 GameObjects, (3 planes, 3 cubes) and the "Collision Matrix" to the right that states which Objects can collide with which layer. In the example, we have set the Collision Matrix in a way that only GameObjects that belong to same layers can collide.

## Setting GameObjects to detect Collisions Based on Layers.

1. Select a layer your Game Objects will belong to




2. Repeat 1 for each Game Object until you have finished assigning your Game Objects to the layers.
3. Open the Physics Preference Panel by clicking on **Edit->Project Settings->Physics**.
4. Select which layers on the Collision Matrix will interact with the other layers by checking them.




Page last updated: 2010-09-22

---

## Tags

A **Tag** is a word which you link to one or more **GameObjects**. For instance, you might define �Player� and �Enemy� Tags for player-controlled characters and non-player characters respectively; a �Collectable� Tag could be defined for items the player can collect in the **Scene**; and so on. Clearly, Tags are intended to identify GameObjects for scripting purposes. We can use them to write script code to find a GameObject by looking for any object that contains our desired Tag. This is achieved using the **GameObject.FindWithTag()** function.

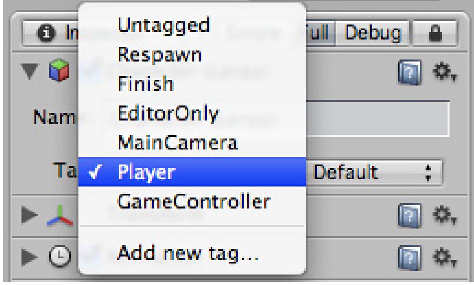
For example:

```
// Instantiates respawnPrefab at the location
// of the game object with tag "Respawn"

var respawnPrefab : GameObject;
var respawn = GameObject.FindWithTag ("Respawn");
Instantiate (respawnPrefab, respawn.position, respawn.rotation);
```

This saves us having to manually add our GameObjects to a script�s exposed properties using drag and drop -- a useful timesaver if the same script code is being used in a number of GameObjects. Another example is a Trigger Collider control script which needs to work out whether the player is interacting with an enemy, as opposed to, say, a random prop or collectable item. Tags make this kind of test easy.

### Applying a Tag

The **Inspector** will show the Tag and Layer drop-down menus just below any GameObject�s name. To apply a Tag to a GameObject, simply open the Tags drop-down and choose the Tag you require:




The GameObject will now be associated with this Tag.

### Creating new Tags

To create a new Tag, click the �Add new tag...� option at the end of the drop-down menu. This will open up the **Tag Manager** in the Inspector. The Tag Manager is described here.

Layers appear similar to Tags, but are used to define how Unity should render GameObjects in the Scene. See the Layers page for more information.

### Hints

- A GameObject can only have one Tag assigned to it.
- Unity includes some built-in Tags which do not appear in the Tag Manager:
  - "Untagged"
  - "Respawn"
  - "Finish"
  - "EditorOnly"
  - "MainCamera"
  - "Player"
  - and "GameController".
- You can use any word you like as a Tag. (You can even use short phrases, but you may need to widen the Inspector to see the tag's full name.)

Page last updated: 2008-02-08

---

# RigidbodySleeping

When Rigidbodies fall to rest - a box landing on the floor - they will start sleeping. Sleeping is an optimization which allows the Physics Engine to stop processing those rigidbodies. This way you can have huge amounts of rigidbodies in your scene as long as you make sure that they normally don't move.

Rigidbody sleeping happens completely automatically. Whenever a rigidbody is slower than the sleepAngularVelocity and sleepVelocity it will start falling asleep. After a few frames of resting it will then be set to sleep. When the body is sleeping, no collision detection or simulation will be performed anymore. This saves a lot of CPU cycles.

Rigidbodies automatically wake up when:

- another rigidbody collides with the sleeping rigidbody
- another rigidbody connected through a joint is moving.
- when modifying a property of the rigidbody
- when adding forces.

So if you want to make rigidbodies fall to rest, don't modify their properties or add forces when they are about to go into sleep mode.

There are two variables that you can tune to make sure your rigidbodies automatically fall to rest: Rigidbody.sleepVelocity and Rigidbody.sleepAngularVelocity. Those two variables are initialized to the sleepVelocity and sleepAngularVelocity variable defined in the Physics Manager (*Edit -> Project Settings -> Physics*).

Rigidbodies can also be forced to sleep using rigidbody.Sleep. This is useful to start rigidbodies in a rest state when loading a new level.

Kinematic rigidbodies wake up sleeping rigidbodies. Static Colliders do not. If you have a sleeping rigidbody and you move a static collider (A collider without a Rigidbody attached) into the rigidbody or pull it from underneath the rigidbody, the sleeping rigidbody will **not** awake. If you move a Kinematic Rigidbody out from underneath normal Rigidbodies that are at rest on top of it, the sleeping Rigidbodies will "wake up" and be correctly calculated again in the physics update. So if you have a lot of Static Colliders that you want to move around and have different objects fall on them correctly, use Kinematic Rigidbody Colliders.

Kinematic rigidbodies - they will not be calculated during the physics update since they are not going anywhere. If you move a Kinematic Rigidbody out from underneath normal Rigidbodies that are at rest on top of it, the sleeping Rigidbodies will "wake up" and be correctly calculated again in the physics update. So if you have a lot of Static Colliders that you want to move around and have different objects fall on them correctly, use Kinematic Rigidbody Colliders.

Page last updated: 2007-11-16