A Simple and Practical Method for Interactive Ray Tracing of Dynamic Scenes

Ingo Wald Carsten Benthin Philipp Slusallek
Saarland University
{wald,benthin,slusallek}@graphics.cs.uni-sb.de

Abstract

Recently developed interactive ray tracing systems combine the high-performance of todays CPUs with new algorithms and implementations to achieve a flexible and high-performance rendering system offering high-quality, interactive 3D graphics. However, due to its history in off-line rendering, interactive ray tracing has been limited to static scenes and simple walkthroughs. However, in order to become truly interactive ray tracing must support dynamic scenes efficiently.

In this paper, we present a simple and practical approach for ray tracing of dynamic scenes. It separates the scene into independent objects with common properties concerning dynamic updates — similar to OpenGL display lists and scene graph libraries. Three classes of objects are distinguished: Static objects are treated as before, objects undergoing affine transformations are handled by transforming rays, and objects with unstructured motion are rebuild whenever necessary.

Using this approach, an interactive ray tracing system is able to efficiently support a wide range of dynamic scenes, which is demonstrated with many examples.

1 Introduction

Methods for creating computer generated images can be broadly classified into two different approaches, both with different strengths and weaknesses. On one side, triangle rasterization is easy to build in hardware, is cheaply available on todays graphics cards, and clearly dominates todays interactive graphics market. On the other side, ray tracing is well-known for achieving superior image quality, but is also infamous for its high computational cost, and has therefore traditionally been used only for off-line rendering.

Recently, the speed of ray tracing has been improved to interactive rates [24, 18]. For a number of application, interactive ray tracing even starts to challenge the dominating role of triangle rasterization: Due to its logarithmic behavior in scene complexity, ray tracing becomes increasingly efficient for complex environments [26]. It offers a much more flexible image generation algorithm than rasterization, supporting features that are hard to achieve with rasterization hardware, including exact reflections, refractions, shadows, arbitrary procedural shaders, and recently even global illumination at interactive rates [25]. Several research projects have even started to investigate new hardware architectures for real-time ray tracing [19, 21].

1.1 Ray Tracing in Dynamic Environments

Interactive ray tracing is relatively new field of research. Most ray tracing research had been concentrated on accelerating the process of creating a *single* image, which could take from minutes to hours. Most of these approaches relied on doing extensive preprocessing by building up complex data structures to accelerate tracing a ray. This preprocessing was then amortized over the remainder of a frame.

This approach was very successful for off-line computations, where the cost for building the data structures was negligible compared to the cost for the actual rendering phase. However, when the time required for tracing rays was reduced by more than an order of magnitude [24] and when ray tracing was used in an interactive setting, this approach was not feasible any more.

Building the acceleration data structure became a bottleneck due to its super-linear behavior with respect to scene complexity. In dynamic scenes, where the acceleration structure would need to be rebuilt for every frame, this preprocessing alone would often exceed the total time available per frame.

Without methods for interactively modifying the scene, interactive ray tracing systems are limited to simple walkthroughs of static environments, and can therefore hardly be termed truly interactive, as real *interaction* between the user and the environment was impossible. In order to be truly interactive, ray tracing *must* be able to efficiently support dynamic environments.

1.2 Paper Outline

We start this paper with an overview over previous work on fast and interactive ray tracing (Section 2). We then present the general approach (Section 3) as well as some implementation details (Sections 4) and report results of applying the techniques to a number of test scenes (Section 5). This is followed by a detailed discussion of the strengths and weaknesses of our approach (Section 6) before offering conclusions and ideas for future work (Section 7 and 8).

2 Previous Work

Ray tracing has first been used by Appel [1], and has been adopted and extended by many other researchers [27, 3, 4]. Since then, speeding up ray tracing has attracted a lot of attention, and has lead to dozens of algorithms. Most of these algorithms rely on reducing the required ray/object intersection tests by building an *acceleration or index structure* over the scene's geometry.

Many different data structures have been proposed, including regular and hierarchical grids, bounding volume hierarchies (BVHs [7]), octrees, BSP trees [22] and kd-trees, and even directional techniques such as ray classification. Dozens of variations of these basic algorithms exist. For a survey of these techniques, see e.g. [4, 8].

Quite recently, ray-tracing has been improved to the point where interactive frame rates could be achieved at least for moderate screen resolutions. By exploiting the inherent parallelism of ray-tracing Muuss [14, 13] and Parker et al. [18, 16, 17] achieved interactive ray tracing performance on shared memory supercomputer systems by massive parallelization and low-level optimizations.

Last year, Wald et al. [24] showed that interactive ray-tracing performance can also be obtained on inexpensive, off-the-shelf PCs. Their implementation is designed for good cache performance using optimized intersection and traversal algorithms as well as a careful layout and alignment of core data structures. Together these techniques increased the performance of ray-tracing by more than an order of magnitude compared to other software ray-tracers [24]. In a related publication it was shown that ray-tracing also scales well in a distributed memory environment using commodity PCs and networks [26].

Unfortunately, all the available acceleration structures have been designed for static environments, thus limiting interactive ray tracing systems to simple walkthroughs of static environments.

Some methods have been proposed for the cases with predefined animation paths known in advance (e.g. [5, 6]). Little research is available for truly interactive systems. In their system Parker et al. [18] keep moving primitives out of the acceleration structure and check them individually for every ray. Of course, this is only feasible for a small number of moving primitives.

Another approach concentrates on efficiently updating an acceleration structures when objects move. Because objects may occupy a large number of cells in an acceleration structure this may require costly updates to large part of the acceleration structure for each moving primitive. To overcome this problem, Reinhard et al. [20] proposed a dynamic acceleration structure based on a hierarchical grid. In order to quickly insert and delete objects independently of their size, larger objects are being kept in coarser levels of the hierarchy. As a result, objects always cover approximately a constant number of cells, thus allowing to update the acceleration structure in constant time. However, their data structure had to be rebuilt once in a while in order to clean up after the updates.

Excellent research on ray tracing in dynamic environments has also been performed by Lext et al. [12]. They provide an excellent analysis and classification of the problems arising due to dynamic environments. They proposed a representative set of test scenes designed to stress the different aspects of ray tracing in dynamic scenes. The BART benchmark provides an excellent tool for evaluating a dynamic ray tracing engine, and will be used extensively in our experiments.

In their research the behavior of dynamic scenes was classified into two inherently different classes [12]: One form is *hierarchical motion*, where a whole group of primitives is subject to the same transformation, which may also be organized hierarchically. The other class is *unstructured motion*, where a set of triangles moves in an unstructured way without relation to each other. For a closer explanation of the different kinds of motion, also see the BART paper [12]. Our method builds on these ideas as proposed independently by Lext et al. [11].

3 Our Approach

Our approach is motivated by the same observations as Lext et al. of how applications typical use a scene graph. Large parts of a scene often remain static over long periods of time. Other parts of a scene undergo well-structured transformations such as rigid motion or affine transformations. Yet other parts receive completely unstructured transformations. This common structure within scenes can be exploited maintaining geometry in separate *objects* according to their dynamic properties.

Each object has its own acceleration structures and can be updates independently of the rest of the scene. An additional top-level acceleration structure must be maintained that accelerates ray traversal between the objects in a scene. For *static objects* ray traversal simply proceeds within an object if its bounding box is hit during traversal of the top-level structure.

For *hierarchical motion*, all triangles that are subject to the same set of transformations (e.g. all the triangles forming the head of the animated robot in Figure 1) must be grouped by the application into the same object. We simply store a transformation that will be applied to the ray before traversal continues.

Transforming such an object simply requires simply updating its transformation matrix. This way, whole groups of triangles can be transformed without modifying their acceleration structure at all. Only the top-level acceleration structure must be updated to reflect the new position of the object.

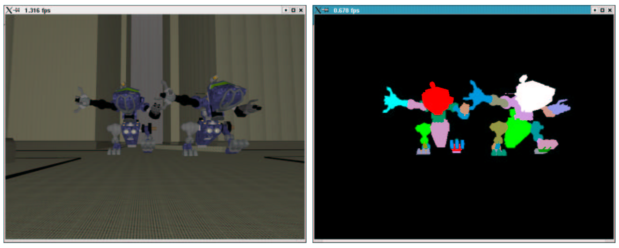

Figure 1. Two robots (left), with color-coded objects (right). All triangles of the same color belong to the same object.

For *unstructured motion*, we rebuilt the acceleration structure for every frame with motion. As such, objects organize triangles subject to similar unstructured motion (e.g. during the same time spans). The local acceleration structures of these objects are discarded and rebuilt from the transformed triangles whenever necessary. Even though this process is costly, it is only required for objects with unstructured motion and does not affect any of the other objects.

Because the implementation the three types of objects are identical, the application is free handle object as required for the current frame — keeping it, updating its transformation, or redefining it.

The top-level acceleration structure is highly-likely to be rebuilt for very frame — whenever any object has been changed. For this case we use a BSP tree that offers highly optimized algorithms for building and traversing (see below).

4 Implementation

Before discussing the actual algorithms for quickly building and traversing acceleration structures, we first give an overview of how objects may be specified by an application. This description is based on the proposed OpenRT API for interactive ray tracing [23]. Similar to OpenGL, OpenRT operates on a very low level that allows it to be used from almost any application or scene graph library. For this paper we concentrate on only a small aspect of OpenRT relevant for dynamic scenes.

4.1 OpenRT API

Our method is similar to the way optimized application make use of OpenGL [15, 2]. Primitives are grouped into display lists depending on their (dynamic update) properties. All triangles inside display lists can then be transformed efficiently adjusting the current transformation calling the display list. Triangles with unstructured motion would simply be rendered in immediate mode.

With OpenRT application operate in a similar way using *objects* instead of *display lists*. The main difference between them is that OpenRT objects do not allow for side effects. In both cases, it is the application that need to organize its geometry accordingly. This organization would be similar for OpenGL and OpenRT, but might require adjustment due to the semantic differences between objects and display lists. Instead of using OpenGL's immediate mode for primitives with unstructured motion, special objects are used that may be redefined when necessary.

In OpenRT, objects are defined by calls to *rtNewObject*, *rtBeginObject*, and *rtEndObject*, which closely correspond to the OpenGL functions for specifying display lists. This similarity simplifies porting of OpenGL based applications. We also use similar calls for specifying geometry within an object, using functions like *rtBegin/End(RT_TRIANGLE)*, *rtVertex3f()*, with all the functionality for transformations (e.g. *rtRotatef()*) and matrix stack handling (e.g. *rtPushMatrix()*) being supported.

After an object has been defined, it can be instantiated any time with a call to *rtInstantiateObject()* using the transformation currently on the transformation stack. Each *instance* of an object consists of a reference to the original object and the applied transformation. Affine transformation of objects can then be implemented by simply reinstantiating the object with a different transformation matrix.

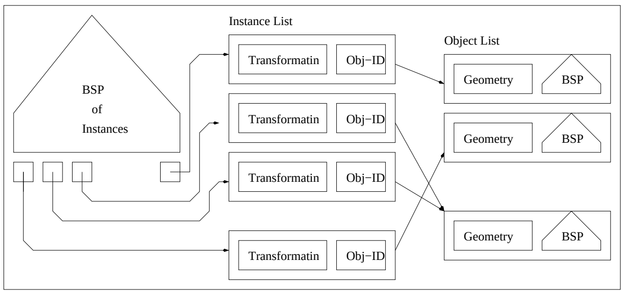

Figure 2. Two-level hierarchy as used in our approach: A top-level BSP contains references to instances, which contain a transformation and a reference to an object. Object in turn consist of geometry and a local BSP tree.

A typical scene then consists on a set of objects with associated acceleration structure and a set of instances (Figure 2). During rendering, each ray is traversed through this data structure. In order to support efficient traversal, we use a two-level approach with specialized data structures for each object and an additional acceleration structure for the list of instances.

4.2 Object Construction and Traversal

During traversal of the data structure, rays have to be intersected with objects. As described above, each object consists of a set of geometric primitives as well as its own acceleration structure for fast traversal within that object.

Within each object the approach is identical to traditional ray tracing in static environments. Consequently, we use exactly the same algorithms for building and traversing that data structure. In our case, we use the BSP tree and data structures described in [24].

For static objects or those with hierarchical motion, the local BSP tree must only be built once directly after the object definition. The time for building these objects is not an issue, allowing us to use sophisticated and slow algorithms for building the acceleration structures[1] already used in [24]. An extensive study of such algorithms can for example be found in [8].

[1]Though the objects (and thus its BSP) remains static in its local coordinate system, its instance in the world coordinate system can still be transformed with rigid-body transformations.

4.3 Fast Handling of Unstructured Motion

The mentioned algorithms for creating highly optimized BSP trees may require several seconds even for moderately complex objects. Thus they are not applicable for unstructured motion, where object may have to be rebuilt for every frame. For these cases sacrifice traversal speed for construction speed. This is accomplished by using less expensive heuristics for BSP plane placement, and by using different quality parameters for BSP construction.

A particularly important cost factor for BSP tree construction is the subdivision depth of the BSP. The subdivision criteria typically consists of a maximum tree depth and a target number of triangles per leaf cell. Subdivision continues on cells with more than the target number of triangles up to the maximum depth. Typical criteria specify 2 or 3 triangles per cell and usually results in fast traversal times, but also in deeper BSPs, which are more costly to create. Particularly costly are degenerate cases, in which subdivision can not reduce the number of triangles per cell, for example if too many primitives occupy the same point in space, e.g. at vertices with a valence higher than the maximum numbers of triangles.

In order to avoid such excessive subdivision in degenerate regions, we modified the subdivision criterion: The deeper the subdivision, the more triangles will be tolerated per cell. We currently increase the tolerance threshold by a constant factor for level of subdivision. Thus we generally obtain significantly lower BSP trees and larger leaf cells than for static objects.

With these compromises on BSP tree construction, unstructured motion for moderate-sized objects can be supported by rebuilding the respective object BSP at every frame. However, these compromises also lead to more triangles per cell, and therefore to a somewhat slower traversal speed.

4.4 Efficient Traversal

Having a separate acceleration structure for every object allows for efficiently intersecting each ray with its geometry. However, a scene may contain many object instances requiring an additional data structure for efficiently traversing the list of instances.

We also use a BSP tree for the top-level acceleration structure (Figure 2), which allows us to use the same efficient, stable, and reliable algorithms for traversing the top-level BSP as for objects. The only difference is that each leaf cell of the top-level BSP tree contains a list of instance ids instead of triangle ids. Only minor changes where required to implement this modified traversal code.

As with the original implementation, a ray is first clipped to the scene bounding box and is then traversed iteratively through the top-level BSP tree. As soon as it encounter a leaf cell, it sequentially intersects the ray with all instances in this cell: For each instance, the ray is first transformed to the local coordinate system of the object, and the transformed ray is then being intersected with the object using the original algorithms. Traversal stops as soon as a valid hit point is found inside the current top-level cell. The cost of intersecting a ray with the same object multiple times can efficiently be avoided with mail-boxing [10].

4.5 Fast Top-Level BSP Construction

While traversal of the top-level BSP required only minor changes to the original implementation, this is not the case for the construction algorithm. As described above, a scene can easily contain several thousand instances. A straightforward BSP construction approach would be too costly for interactive use. On the other hand, the top-level BSP is traversed by *every* ray, and thus has to be sufficiently efficient for ray traversal.

Fortunately, the task of building the top-level BSP is different than for object BSPs thus simplifying the problem: Object BSPs require costly triangle-in-cell computations, careful placement of the splitting plane, and handling of degenerate cases. In contrast, the top-level BSP contains only instances represented by an axis-aligned bounding box (AABB) of its transformed object. Considering only the AABBs, optimized placement of the splitting plane becomes much easier, and problematic cases can be avoided.

For splitting a cell, we incorporate several observations:

1. It is usually beneficial to subdivide a cell in the dimension of its maximum extent, as this usually yields the most well-formed cells [8].

2. Placement of the BSP plane only makes sense at the boundary of objects contained within the current cell. This is due to the fact that the cost-function can be maximal only at such boundaries, see [8].

3. It can been shown that the optimal position for the splitting plane lies between the cells geometric center and the object median [8]

Using these three observations, the BSP tree can be built in a way that it is both suited for fast traversal by optimized plane placement, and can still be built quickly and efficiently: For each subdivision step, we try to find a splitting plane in the dimension of maximum extent (observation 1). As potential splitting planes, only the AABB borders will be considered (observation 2). In order to find a good splitting plane, we first split the cell in the middle, and decide which side contains more objects, i.e. which contain the object median. From this side, we choose the object boundary that is closest to the center of the cell. As such, the splitting plane lies between cell center and object median, which is generally a good choice (observation 3).

If no splitting plane can be found in the dimension of maximum extent, the other two dimensions will be checked in turn. If neither dimension yields a valid splitting plane, no subdivisions are necessary any more, and a leaf cell will be created (see Algorithm 1).

Algorithm 1 Algorithm for building the top-level BSP tree.

```
BuildTree(instances,cell)
  for d = x,y,z in order of maximum extent
    P = $\{i.min_d, i.max_d | i \in instances\}$
    c = center of cell
    if (more instances on left side of c than on right)
      p = max($\{p \in P || p < c\}$)
    else
      p = min($\{p \in P || p >= c\}$)
    if ($p$ inside cell)
      {leftvox,leftinst,rightvox,rightinst}
        = SplitCell(instances,cell,p)
      BuildTree(leftinst,leftvox);
      BuildTree(rightinst,rightvox);
      return;
  # no valid splitting plane found
  cell = Leaf(instances)
```

This way, the construction algorithm yields an optimized BSP subdivision that be traversed quickly while using a fast and efficient algorithm for BSP tree creation. As each subdivision step removes at least one potential splitting plane, termination of the subdivision can be guaranteed, and no special measures have to be taken to avoid excessive subdivision. Choosing the splitting plane in the described way also yields relatively small and well-balanced BSP trees. Finally, degenerate cases as for triangles cannot happen for axis aligned bounding boxes as only boundaries are considered, and not the overlapping space itself.

5 Results

For testing our system, we have chosen to use the BART benchmark scenes, which represent a wide variety of stress factors for ray tracing of dynamic scenes [12]. Additionally, we use several of the scenes that we encountered in practical applications [25], and a few custom-made scenes for stress-testing. Snapshots of test scenes can be found in Figure 6.

All of the following experiments have been performed on a cluster of dual AMD AthlonMP 1800+ machines with a FastEthernet network connection. The network is fully switched with a gigabit ethernet connection to a dual AthlonMP 1700+ server. The application is running on the server and is unaware of the distributed rendering happening in the rendering engine. It manages the geometry in a

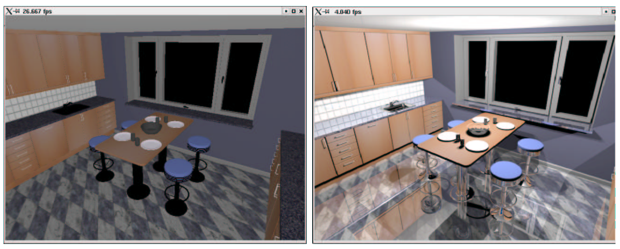

Figure 3. Two snapshots from the BART kitchen. Left: OpenGL-like shading running at >26 fps on 32 CPUs. Right: fully-featured ray tracing with shadows and reflections (reflection depth 3) running at >4 fps on 32 CPUs.

scene graph, and transparently controls rendering via calls to the OpenRT API. All examples are rendered at a resolution of 640 × 480.

5.1 BART Kitchen

The Kitchen scene contains hierarchical animations of 110.000 triangles divided across 5 objects. This results in negligible network bandwidth and BSP construction overhead. Overlap of bounding boxes may results in a certain overhead, which is hard to measure exactly but is definitely not a major cost factor.

The main cost of this scene is due to the need to trace many rays due to shadows from 6 point lights and a high degree of reflectivity on many objects. Due to fast camera motion and highly curved objects (see Figure 3), these rays are rather incoherent. However, these aspects are completely independent of the dynamic nature of the scene and are handled efficiently by our system.

We achieve interactive frame rates even for the large amount of rays to be shot. A reflection depth of 3 results in a total of 3.867.661 rays/frame. At a measured rate of 526.000 rays traced per second and CPU in this scene, this translating to a frame rate of 4.3 fps on 32 CPUs. Scalability is almost linear (see Table 1) – using twice as many CPUs results in roughly twice the frame rate.

CPUs	2	4	8	16	32
OpenGL-like	1.7	3.4	6.8	13.6	>26
Ray Tracing	0.25	0.5	1.05	2	4

Table 1. Scalability in the Kitchen scene in frames/sec.

5.2 BART Robots

The Robots scene was mainly designed for stressing hierarchical animation. 16 Robots move through a complex city with hierarchical animation of their body parts organized into 161 different objects. All dynamic motion is hierarchical with no unstructured motion. Therefore, the BSP trees for all objects have to be built only once, and only the top-level BSP must be rebuilt for every frame.

Scene	num. objs	num. triangles	reconstruction time (in msec)
Robots	161	100K	1
Office	9	34K	<1
Terrain	661	8M	4

Table 2. Reconstruction time of the top-level BSP: Using our optimized algorithm, reconstruction time remains in the order of a few milliseconds. Other scenes are even less expensive.

Using the algorithms described above, rebuilding the top-level BSP is very efficient taking less than one millisecond (see Table 2). Furthermore, updating the transformation matrices requires only a well tolerable network bandwidth of 20 kB/frame for each client.

CPUs	2	4	8	16	32
OpenGL-like	1.25	2.49	5.1	10	20
Ray Tracing	0.24	0.48	1.01	2.01	4

Table 3. Scalability in the Robot scene in frames/sec.

With such a small transmission and reconstruction overhead, we again achieve almost-linear scalability (see Table 3) and high rendering performance. Using 32 CPUs, we achieve a frame rate of 4 frames per second. Again, the high cost of this scene is due to the large number of reflection and shadow rays. Using a simple OpenGL-like shader (see Figure 4) results in frame rates of more than 20 frames per second.

5.3 BART Museum

The museum has been designed mainly for testing unstructured motion and is the only BART scene featuring non-hierarchical motion. In the center of the museum, several triangles are animated on predefined animation paths to form differently shaped objects. The number of triangles undergoing unstructured motion can be configured to 64, 256, 1k, 4k, 16k, or 64k.

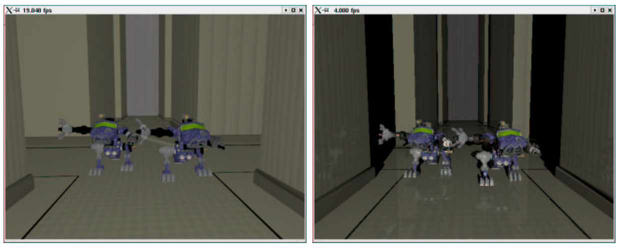

Figure 4. The Bart Robots: 16 robots consisting of 161 objects rendered interactively. Left image: OpenGL-like shading at >20 fps on 32 CPUs. Center image: standard ray tracing (reflection depth of 3) at >4 fps at 32 CPUs. Right image: a color-coded version showing the different objects.

Even though the complete animation paths are specified in the BART scene graph, we do not make use of this information. User controlled movement of the triangles, i.e. without knowledge of future positions, would create the same results.

As can be expected, unstructured motion becomes costly for many triangles. Building the BSP tree for the complex version of 64k triangles already requires more than one second (see Table 4). Note, however, that our current algorithms for building object BSP trees still leaves plenty of room for further optimizations.

Furthermore, the reconstruction time is strongly affected by the distribution of triangles in space: In the beginning of the animation, all triangles are equally and almost-randomly distributed. This is the worst case for BSPs, which are best at handling highly uneven distributions, and construction is consequently costly. During the animation, the triangles organize themselves to form a single surface. This results in much faster reconstruction time. The numbers given in Table 4 are taken at the beginning of the animation, and are thus worst-case results.

Apart from raw reconstruction cost, significant network bandwidth is required for sending all triangles to every client for every frame. Since we use reliable unicast for network transport, 4096 triangles and 16 clients (32 CPUs), resulting roughly 6.5 MB have to be transferred (Table 4). Though this does not yet saturate the network, the performance of the server is already affected. Consequently, we do not scale completely linearly any more (see Table 6).

This scene also requires the computation of shadows from two point lights as well as large amounts of reflection rays. All of the moving triangles are reflective and incoherently sample the whole environment (see Figure 5). As the dynamic behavior of a scene is completely transparent to the shaders, integrating all these effects does not require any additional effort except for the cost for tracing the rays.

	num. triangles	reconstruction time (in msec)	data sent/client (in bytes)
museum3	64	1	6,4k
museum4	256	2	25,6k
museum5	1k	8	102k
museum6	4k	34	409k
museum7	16k	101	1,6M
museum8	64k	>1s	6,5M

Table 4. Unstructured motion in different configurations of the museum scene. The number of triangles only specifies the number of triangles undergoing unstructured motion.

Even with all these effects – unstructured motion, shadows, and highly incoherent reflections on moving objects – the museum can be rendered interactively: Using 8 clients, we achieve 4.8 fps for 1024 triangles, and still 4.2 fps for 4096 triangles in video resolution. Again, the frame rate is dominated by the cost for shadows and reflections. Using an OpenGL-like shader without these effects allows to render the scene at 19 frames per second on 8 clients.

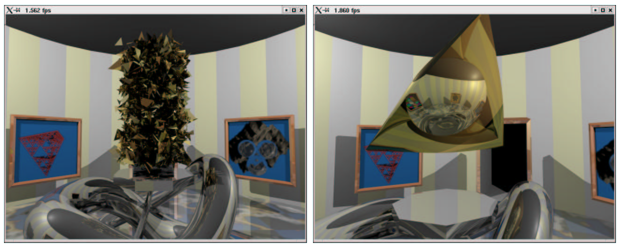

Figure 5. Unstructured motion in the BART museum: Up to 64.000 triangles are moving incoherently through the room. Note especially how the entire environment reflects in these moving triangles (right).

5.4 Outdoor Terrain

The Terrain scene has been specifically designed to test the scalability with a large number of instances and triangles. It contains up to 661 instances of 2 different trees, which corresponds to roughly 10 million triangles after instantiation. A point light source creates highly detailed shadows from the leaves (see Figure 6). All trees can be moved around interactively, both in groups or individually.

The large number of instances results in construction times for the top-level BSP of up to 4 msec per frame. This cost — together with the transmission cost for updating all 661 instance matrices on all clients — limits the scalability for a large number of instances and clients (see Table 6).

6 Discussion

The above scenes stress our dynamic ray tracing system in different areas. Together with the terrain experiment, our test scenes contain a strong variation of parameters, ranging from 5 to 661 instances, from a few thousand to several million triangles, from simple shading to lots of shadows and reflections, and from hierarchical animation to unstructured motion of thousands of triangles (for an overview, see Figure 6). Taken together, these experiments allow to analyze and evaluate our method with respect to many different aspects.

Transformation Cost The core idea of our method was to avoid rebuilding the complete data structure, but rather transform the rays to the coordinate system of each object whenever possible. This implies that every ray intersecting a object has to be transformed to that objects local coordinate system via matrix-vector multiplications for both ray origin and direction, resulting in several matrix operations per ray. As our system shoots approximately half a million rays per second on an AthlonMP 1800+ CPU, this can amount to hundreds of thousands of matrix-vector multiplications per frame (see Table 5). Furthermore, more transformations are often required during shading, e.g. by transforming the shading normal or for calculating procedural noise in the local coordinate system.

	Office	Terrain	Robots
objects	9	661	161
matrix ops	480K	1600K	1000K

Table 5. Number of the matrix-vector multiplies for the scenes in our benchmark scenes (resolution 640x480). Note, a matrix operation can be performed in only 23 cycles even in plain C code, which is negligible compared to traversal cost.

However the cost for these transformation is rather low in practice. Even for a straight-forward C-code implementation, a matrix-vector operation costs only 23 cycles on an AMD Athlon MP CPU, which is almost negligible compared to the cost for tracing a ray, which is in the order of several hundred to a thousand cycles. The cost for matrix operations could be further reduced by replacing the matrix-vector multiplications by SSE code [9].

Unstructured Motion As could be expected, the Museum scene has shown that unstructured motion remains costly for ray tracing. A moderate number of a few thousand independently moving triangles can be supported, but larger numbers would lead to intolerable reconstruction times for the respective objects (see Table 4). As such, our method is still not suitable for scenes with strong unstructured motion.

To support such scenes, algorithms for faster reconstruction of dynamic objects have to be developed. Note that our method could also be combined with Reinhards approach [20] by using his method only for the unstructured objects. Even then, lots of unstructured motion would still create a performance problem due to the need to send all triangle updates to the clients. This is not a limitation of our specific method, but would be similar for any kind of algorithm in a distributed environment.

Bounding Volume Overlap One of the stress cases defined in [12] was Bounding Volume Overlap. In fact, this results in some form of overhead, as it limits early ray termination. All instances must be tested sequentially in the overlap area as a valid intersection computed in the first object might not be visible due to being occluded by another instance.

Though it would be easy to construct scenarios where this would lead to excessive overhead, this is rarely significant in practice. Bounding volume overlap does happens in *all* our test cases, but has not proven a major performance problem. In fact, overlapping objects are identical to using bounding volume hierarchies (BVHs) [7] as an acceleration structure, which have proven to work well in practice.

Over-Estimation of Object Bounds Building the top-level BSP requires an estimate of the bounding box of each instance in world coordinates. As transforming each individual vertex would be too costly, we conservatively estimate this bounds based on the transformed bounding box of the original object.

This sometimes over-estimates the correct bounds and results in some overhead: During top-level BSP traversal, a ray may be intersected with an object that it would not have intersected otherwise. However, this overhead is restricted to only transforming and clipping the ray: After transformation to the local coordinate system, such a ray is first clipped against the correct bounding box, and can thus be immediately discarded without further traversal.

Teapot-in-a-Stadium Problem The teapot-in-a-stadium problem is handled very well by out method: BSP trees adapt automatically to varying object density in a scene [8], which solves the problem for both objects and top-level

BSP. In fact, our method even allows to increase performance for these cases: If the 'teapot' is contained in a separate object, the shape of the 'stadium' BSP is usually much better: The teapot object is already enclosed in tightly fitting bounds, without using several additional BSP levels to tightly enclose the teapot.

Scalability with the number of Instances Apart from unstructured motion, the main cost of our method results from the need to recompute the top-level BSP tree. As such, a large number of instances becomes expensive, as can be seen in the Terrain scene. Still, even the thousand complex instances can be rendered interactively, and using only a few dozen instances has negligible impact.

As such, the number of instances should be minimized in order to achieve optimal performance. It is generally much faster to use a few, large objects instead of many small ones. All static triangles in a scene should best be stored in a single object, instead of using multiple objects. This is completely different to OpenGLs approach of using many, small display lists, and still requires some amount of manual porting and optimization when porting applications from OpenGL to OpenRT.

On the other hand, supporting instantiation (i.e. using exactly the same object multiple times in the same frame) is a valuable feature of our method, as this allows to render complex environments very efficiently: With instantiation, memory is required for storing only the two original trees and the top-level BSP, allowing to render even that many triangles with a small memory footprint. For OpenGL rendering, all triangles would still be handled individually by the graphics hardware even when using display lists.

Scalability in Distributed Environments As could be seen by the experiments in Section 5, we achieve rather good scalability even for many clients except for scenes that require to update a lot of information on all clients, i.e. for a high degree of unstructured motion (where every moving triangle must be transmitted), and for a large number of instances.

In the terrain scene, using 16 clients would require to send 676 KB[2] per frame simply for updating the 661 transformation matrices on the clients. Though this data can be sent in a compressed form, load balancing and client/server communication further adds to the network bandwidth. Without broadcast/multicast functionality on the network, the server bandwidth increases linearly with the number of clients. For many clients and lots of updated data, this creates a bandwidth bottleneck on the server, and severely limits the scalability (see Table 6).

[2]661 instances$\times$16 clients$\times(4 \times 4)$ floats

In principle, the same is true for unstructured motion, where sending several thousand triangles to each client also creates a network bottleneck. On the other hand, both problems are not specific to our method, but apply for any kind of distributed rendering.

OpenGL-like	1	2	4	8	16
Robots	1.25	2.49	5.1	10	20
Kitchen	1.7	3.4	6.8	13.6	26(-)
Terrain	0.68	1.34	2.55	4.76	8.33
Museum/1k	2.7	5.4	10.2	19.5	26(-)
Museum/4k	2.5	4.5	7.5	4.5	2.5
Museum/16k	1.6	2.4	1.7	1	0.5

Ray Tracing	1	2	4	8	16
Robots	0.24	0.48	1.01	2.01	4
Terrain	0.3	0.6	1.19	2.29	4.26
Kitchen	0.25	0.5	1.05	2	4
Museum/1k	0.6	1.2	2.4	4.8	9.3
Museum/4k	0.55	1.1	2.2	4.2	2.5
Museum/16k	0.45	0.9	1.65	0.98	0.53

Table 6. Scalability of our method in the different test scenes. '-' means that the servers network connection is completely saturated, and thus no higher performance can be achieved. The numbers in the upper table correspond to pure OpenGL like shading, the lower one is for full ray tracing including shadows and reflections.

Total Overhead In order to estimate the total overhead of our method, we have compared several scenes in both a static and dynamic configuration. As there are no static equivalents for the BART benchmarks, we have taken several of our static test scenes, and have modified them in a way that they can be rendered in both a static configuration with all triangles in a single, static BSP tree, and in a dynamic configuration, where triangles are grouped into different objects that can then be moved dynamically.

Note, however, that the total performance is affected by several factors. Even though transformation and reconstruction cost lead to overhead, using a hierarchy can also have positive side effects. For example, having small and complicated objects (e.g. teapots in a stadium) contained in separate objects can lead to BSP trees that are actually better situated than those for a single static BSP tree, and can even result in faster traversal.

For the scenes that are available in both static and dynamic configurations, we find that our method creates an overhead of only 10 to 20 percent for typical scenes. We consider this overhead tolerable for the added flexibility gained through supporting dynamic scenes.

7 Conclusions

We have presented a simple and practical method for interactive ray tracing of dynamic scenes. It supports a large variety of dynamic scenes, including all the BART benchmark scenes (see Figure 6). It imposes no limitations on the kind of rays to be shot, and as such allows for all the usual ray-tracing features like shadows and reflections.

For unstructured motion, our method still incurs a high reconstruction cost per frame, that makes it infeasible for a large number of incoherently moving triangles. For a moderate amount of unstructured motion in the order of a few thousand moving triangles, however, it is well applicable, and results in frame rates of several frames per second at video resolution.

For mostly hierarchical animation — as often applied in scene graphs — our method is highly efficient, and allows to interactively render even highly complex models with hundreds of instances, and millions of triangles per object [23].

With our proposed method, we have been successfully able to interactively ray trace all the dynamic scenes we have encountered so far. To our knowledge, this is the first time that the BART benchmark suite has been interactively ray traced at all. Using only an OpenGL like shading model and a small cluster of commodity PCs, more than 15 to 20 frames per second can be achieved in most scenes.

With the unique advantages of ray tracing — now combined with the flexibility to handle dynamic environments — we believe that interactive ray tracing is a significant step closer to be a viable alternative to triangle rasterization for future interactive 3D graphics.

8 Future Work

Even though rebuilding the top-level BSP has not proven to be a major problem, we would be able to organization objects into a hierarchy instead of a flat list. This would further limit the number of objects affected by a local change. As most of the updated data is the same for every client, the support of network broadcast/multicast would be a very simple solution to the bandwidth problem, as the transmission time would not be affected by the number of clients.

In order to avoid the scalability bottleneck due to transmission cost on the network and BSP construction cost on all clients, future work will investigate algorithms for lazy loading of the geometry and for lazy construction of the BSP trees. Unstructured motion could be improved by designing specialized algorithms for cases where motion is spatially limited in some form, such as for skinning.

We are also investigating how existing applications can be mapped to our method, e.g. by evaluating how a scene graph library such as OpenInventor or VRML can be efficiently implemented on top of our system.

References

[1] A. Appel. Some Techniques for Shading Machine Renderings of Solids. *SJCC*, pages 27–45, 1968.

[2] OpenGL Architecture Review Board. *OpenGL Reference Manual: The Official Reference Document for OpenGL, Release 1*, 1993.

[3] Robert Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 137–144, 1984.

[4] Andrew Glassner. *An Introduction to Raytracing*. Academic Press, 1989.

[5] Andrew S. Glassner. Spacetime ray tracing for animation. *IEEE Computer Graphics and Applications*, 8(2):60–70, 1988.

[6] Eduard Gröller and Werner Purgathofer. Using temporal and spatial coherence for accelerating the calculation of animation sequences. In *Proceedings of EUROGRAPHICSt'91*, pages 103–113. Elsevier Science Publishers, 1991.

[7] Eric Haines. Efficiency improvements for hierarchy traversal in ray tracing. In James Arvo, editor, *Graphics Gems II*, pages 267–272. Academic Press, 1991.

[8] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, 2001.

[9] Intel Corp. *Intel Pentium III Streaming SIMD Extensions*. http://developer.intel.com/vtune/cbts/simd.htm.

[10] David Kirk and James Arvo. Improved ray tragging for voxel-based ray tracing. *Graphics Gems II*, 1991.

[11] Jonas Lext and Tomas Akenine-Möller. Towards rapid reconstruction for animated ray tracing. In *Eurographics 2001 – Short Presentations*, pages pp. 311–318, 2001.

[12] Jonas Lext, Ulf Assarsson, and Tomas Moeller. BART: A benchmark for animated ray tracing. Technical report, Department of Computer Engineering, Chalmers University of Technology, Goeteborg, Sweden, May 2000. Available at http://www.ce.chalmers.se/BART/.

[13] Michael J. Muuss. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium '95*, June 1995.

[14] Michael J. Muuss and Maximo Lorenzo. High-resolution interactive multispectral missile sensor simulation for atr and dis. In *Proceedings of BRL-CAD Symposium '95*, June 1995.

[15] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading MA, 1993.

[16] Steven Parker, Michael Parker, Yaren Livnat, Peter Pike Sloan, Chuck Hansen, and Peter Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Computer Graphics and Visualization*, 5(3):238–250, July-September 1999.

[17] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive ray tracing for isosurface rendering. In *IEEE Visualization '98*, pages 233–238, October 1998.

[18] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive Ray Tracing. In *Symposium on Interactive 3D Graphics*, pages 119–126. ACM SIGGRAPH, 1999.

[19] Timothy John Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *Proceedings of SIGGRAPH*, 2002. (to appear).

[20] Erik Reinhard, Brian Smits, and Chuck Hansen. Dynamic acceleration structures for interactive ray tracing. In *Proceedings Eurographics Workshop on Rendering*, pages 299–306, Brno, Czech Republic, June 2000.

[21] Joerg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR – A Hardware Architecture for Realtime Ray-Tracing. submitted for publication, also available as a technical report, http://graphics.cs.uni-sb.de/Publications, 2002.

[22] K. Sung and P. Shirley. Ray Tracing with the BSP-tree. In D. Kirk, editor, *Graphics Gems III*, pages 271–274. Academic Press, 1992.

[23] Ingo Wald, Carsten Benthin, and Philipp Slusallek. OpenRT - a Flexible and Scalable Rendering Engine for Interactive 3D Graphics. submitted for publication, also available as a technical report, http://graphics.cs.uni-sb.de/Publications, 2002.

[24] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3), 2001.

[25] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive Global Illumination. submitted for publication, also available as a technical report, http://graphics.cs.uni-sb.de/Publications, 2002.

[26] Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive distributed ray tracing of highly complex models. In *Proceedings of the 12th EUROGRPAHICS Workshop on Rendering*, June 2001. London.

[27] T. Whitted. An improved illumination model for shaded display. *CACM*, 23(6):343–349, June 1980.

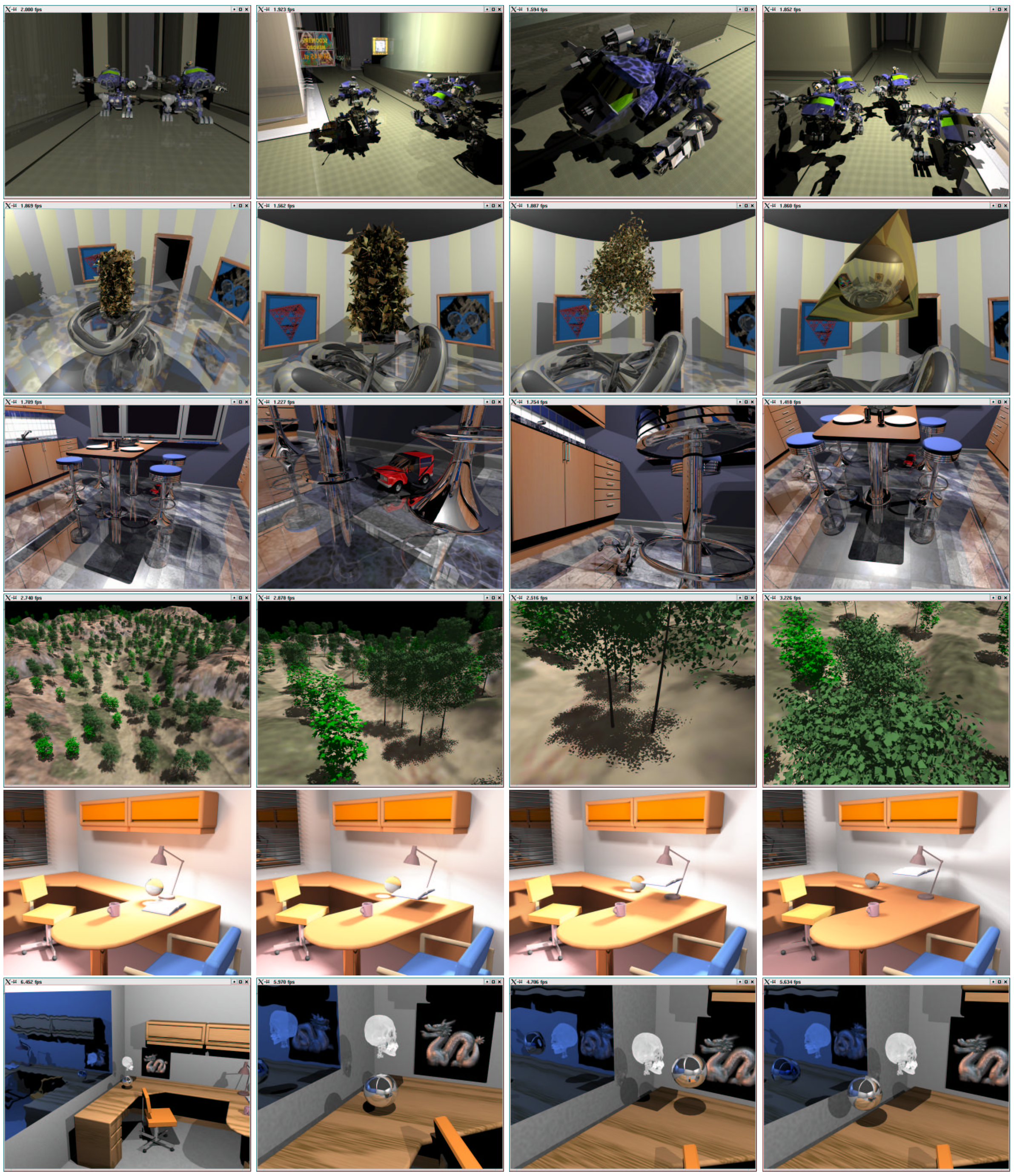

Figure 6. Some example frames from several dynamic scenes. From top to bottom: The BART robots scene contains roughly 100.000 triangles in 161 moving objects. Below that, is the BART kitchen scene. The museum scene contains unstructured motion of several thousand triangles. Note how the entire museum reflects in these triangles. The terrain viewer application uses up to 661 instances of 2 trees, would contain several million triangles without instantiation, and even calculates shadows. The office scene is a practical application from interactive lighting simulation, and demonstrates that the method works fully automatically and completely transparently to the shader.