All About OpenGL Extensions, including specifications for some significant OpenGL extensions

Mark J. Kilgard [*]
NVIDIA Corporation

OpenGL is an *extensible* low-level graphics API. Extensible is the key word. OpenGL implementations are free to extend OpenGL's basic rendering functionality with new rendering operations. Scores of OpenGL extensions have been specified and implemented. These extensions provide OpenGL application developers with new rendering features above and beyond the features specified in the official OpenGL standard. OpenGL extensions keep the OpenGL API current with the latest innovations in graphics hardware and rendering algorithms. Better yet, extensions provide OpenGL with that fresh minty taste that developers love.

The document reviews the OpenGL extension mechanism and describes a set of new significant OpenGL extensions likely to be interesting to PC game developers. By reading this document, you will learn not just what extensions are but how to use them portably in your programs and how to read OpenGL extension specifications. The appendixes contains selected significant OpenGL extension specifications of interest to the PC 3D game developer. Note that in some cases as noted, several extension specifications are preliminary versions.

1. How OpenGL Extensions are Documented

An OpenGL extension is defined by its specification. These specifications are typically written as standard ASCII text files. OpenGL extension specifications are written by and for OpenGL implementers. A well-written OpenGL specification is documented to the level of detail needed for a hardware designer and/or OpenGL library engineer to unambiguously implement the extension. This means that OpenGL application programmers should not expect an extension's specification to fully justify why the functionality exists or how an OpenGL application would go about using the functionality. An OpenGL extension is not a tutorial on how to use the particular extension. Still, being able to read and understand an OpenGL specification helps you, the application programmer, fully understand an OpenGL extension's functionality.

2. Where to Find OpenGL Extension Specifications

The latest public OpenGL specifications can be found on the www.opengl.org web site. Note that extension specifications are updated from time to time based on reviews and implementation feedback.

3. How to Read an OpenGL Extension Specification

When reading an OpenGL extension specification, it helps to be familiar with the original OpenGL specification. The operation of an OpenGL extension is described as additions and changes to the core OpenGL specification. Having a copy of the core OpenGL specification handy is a good idea when reviewing an OpenGL specification.

---

[*] Mark graduated with B.A. in Computer Science from Rice University and is a System Software Engineer at NVIDIA. Mark is the author of *OpenGL Programming for the X Window System* (Addison-Wesley, ISBN 0-201-48359-9) and can be reached by electronic mail addressed to mjk@nvidia.com

OpenGL extension specifications consist of multiple sections. There is common form established by convention used by nearly all OpenGL extension specifications. Often within a specification, the `gl` and `GL` prefixes on routine names and tokens are assumed. The following describes the purpose of the most common sections in the order that they normally appear in extension specifications:

Name
Lists the official name of the extension. This name uses underscores instead of spaces between words. The name also begins with a prefix that indicates who developed the extension. This prefix helps to avoid naming conflicts if two independent groups implement a similar extension. It also helps identity who is promoting use of the extension. For example: `SGIS_point_parameters` was an extension proposed by Silicon Graphics. The `SGIS` prefix belongs to Silicon Graphics. SGI uses the `SGIS` prefix to indicate the extension is specialized and may not be available on all SGI hardware. Other prefixes in use are:

- `ARB` – Extensions officially approved by the OpenGL Architectural Review Board
- `EXT` – Extensions agreed upon by multiple OpenGL vendors
- `HP` – Hewlett-Packard
- `IBM` – International Business Machines
- `INTEL` – Intel
- `NVIDIA` – NVIDIA Corporation, coolest 3D company on the planet
- `MESA` – Brian Paul's freeware portable OpenGL implementation
- `SGIX` – Silicon Graphics (experimental)
- `SUN` – Sun Microsystems
- `WIN` – Microsoft

Note that the `SGIS_point_parameters` extension has since been standardized by other OpenGL vendors such as NVIDIA. So now there is also an `EXT_point_parameters` extension with the same basic functionality as the `SGIS` version. The `EXT` prefix indicates that multiple vendors have agreed to support the extension. Successful OpenGL extensions are often promoted to `EXT` or `ARB` extensions or made an official part of OpenGL in a future revision to the core OpenGL specification. Almost all of the new functionality in OpenGL 1.1 and 1.2 showed up first as OpenGL extensions.

Name Strings
The name string or strings is used to indicate that the extension is supported by a given OpenGL implementation. Applications can query the `GL_EXTENSIONS` string with OpenGL's `glGetString` to determine what extensions are available. OpenGL also supports the idea of window system dependent extensions. Core OpenGL extension name strings are generally prefixed with `GL` while window system dependent extensions are prefixed with `GLX` for the X Window System or `WGL` for Win32 based on what window system the extension applies to. Note that there may be multiple strings if the extension provides both core OpenGL rendering functionality and window system dependent functionality.

Version
A source code control revision string to keep track of what version of the specification the given text file represents. It is important to make sure that you have the latest version of the extension specification in case there are any important changes. Normally the version string has the date the extension was last updated.

Number

Each OpenGL extension is assigned a unique number. Silicon Graphics allocates these numbers to ensure that OpenGL extensions do not overlap in their usage of enumerants or protocol tokens. This number is only important to extension implementers.

Dependencies
Often an extension specification builds on the functionality of pre-existing extensions. This section documents what extensions the specified extension depends on. Dependencies indicate that another extension "is required" to support the specified extension or that the specified extension "affects" the specification of another extension. When an extension affects the specification of another extension, the affecting extension is responsible for fully documenting the interactions between the two extensions.

The dependencies section often also indicates which version of the OpenGL core standard that the extension specification is based on. Later sections specify the extension based on updates to the relevant section of the particular OpenGL specification that the extension is based on.

You can often tell how important a given extension is to the evolution of OpenGL based on how many other extensions are listed that depend on or are affected by the given extension. The multitexture extension to be discussed later affects gobs of other extensions!

Overview
The section provides a description, often terse and without justification, for the extension's specified functionality. If you are trying to figure out what the extension does, this is the most useful section of an OpenGL extension specification. Do not expect a tutorial though.

Issues
Often there are issues that need to be resolved in the specification of an extension. This section documents open issues and states the resolution to resolved issues. These issues are often things of interest to the extension implementer, but can also help a programmer understand how the extension really works.

New Procedures and Functions
This section lists the function prototypes for any new procedures and functions that the extension adds. Keep in mind that specifications often leave out the `gl` prefix when discussing routines. Also note that the extension's new functions will be suffixed with the same letters used as the prefix for the extension name.

New Tokens
This section lists the tokens (also called enumerants) that the extension adds. The routines that accept each set of new enumerants are documented. The integer value of the enumerants is documented here. These values should be added to `<GL/gl.h>`. Keep in mind that specifications often leave out the `GL` prefix when discussing enumerants. Also note that the extension's new enumerants will be suffixed with the same letters used as the prefix for the extension name.

Additions to Chapter *XX* of the 1.*X* Specification (*XXX*)
These sections document how the core OpenGL specification should be amended to add the extension's functionality to the core OpenGL functionality. Notice that the exact version of the core OpenGL specification (such as 1.0, 1.1, or 1.2) is documented. The chapters typically amended by an extension specification are:

- Chapter 2 – OpenGL Operations
- Chapter 3 – Rasterization

- Chapter 4 – Fragments and the Framebuffer
- Chapter 5 – Special Functions
- Chapter 6 – State and State Requests

These sections are quite legalistic. They indicate precisely how the OpenGL specification wording should be amended or changed. Often tables within the specification are amended as well.

Additions to the GLX Specification
If an extension has any window system dependent functionality affecting the GLX interface to the X Window System, these issues would be documented here.

GLX Protocol
When implementing the extension for the X Window System, if any special X11 extension protocol for the GLX extension is required to support the extension, the protocol would be documented in this section.

Dependencies on *XXX*
These sections describe how the extension depends on some other extension that was listed in the Dependencies section. Usually the wording says that if the other extension is not supported, simply ignore the portion of this extension dealing with the dependent extension's state and functionality.

Errors
If the extension introduces any new error conditions particular to the extension, they are documented here.

New State
Extensions typically add new state variables to OpenGL's state machine. These new variables are documented in this section. The variable's get enumerant, type, get command, initial value, description, section of the specification describing the state variable's function, and the attribute group that the state belongs to are all documented in tables in this section.

New Implementation Dependent State
Extensions may add implementation dependent state. These are typically maximum and minimum supported ranges for the extension functionality. For example, what is the widest line size supported by the extension. These values can be queried through OpenGL's `glGet` family of routines.

Backward Compatibility
If the extension supercedes an older extension, issues surrounding backward compatibility with the older extension are documented in this section.

Note that these sections are merely established by convention. While the conventions for OpenGL extension specifications are normally followed, extensions vary in how closely they stick to the conventions. Generally, the more preliminary an extension is, the more loosely specified it is. Hopefully after sufficient review and even implementation, the specification language and format is improved to provide an unambiguous final specification.

4. Portably Using OpenGL Extensions

The advantage of using OpenGL extensions is getting access to cutting edge rendering functionality so you application can achieve higher performance and higher quality rendering. OpenGL extensions give you access to the latest features of the hottest new graphics hardware.

The problem with OpenGL extensions is that lots of OpenGL implementations, particularly older implementations, will not support the extensions that you would like to use. When you write an OpenGL application that uses extensions, you should make sure that your application still works when the extension is not supported. At the very least your program should report that it requires whatever extension is missing and exit without crashing.

The first step to using OpenGL extensions is to locate the copy of the <GL/gl.h> header file that advertises the API interfaces for the extensions that you plan to use. Typically you can get this from your OpenGL implementation vendor or OpenGL driver vendor. You could also get the API interface prototypes and macros directly from the extension specifications, but getting the right <GL/gl.h> from your OpenGL vendor is definitely the preferred way.

You will notice that <GL/gl.h> sets C preprocessor macros to indicate whether the header advertises the interface of a particular extension or not. For example, the basic <GL/gl.h> supplied with Visual C++ 4.2 has a section reading:

```
/* Extensions */
#define GL_EXT_vertex_array             1
#define GL_WIN_swap_hint                1
#define GL_EXT_bgra                     1
#define GL_EXT_paletted_texture         1
#define GL_EXT_clip_disable             1
```

These macros indicate that the header file advertises the above five extensions. The EXT_bgra extension lets you read and draw pixels in the Blue, Green, Red, Alpha component order as opposed to OpenGL's standard RGBA color component ordering. If you wanted to write a program to use the EXT_bgra extension, you could test that the extension is supported at compile time like this:

```
#ifdef GL_EXT_bgra
   glDrawPixels(width, height, GL_BGRA_EXT, GL_UNSIGNED_BYTE, pixels);
#endif
```

When GL_EXT_bgra is defined, you can expect to find the GL_BGRA_EXT enumerant defined. Note that if the EXT_bgra extension were not supported, you would expect the glDrawPixels line above to generate a compiler error because the base OpenGL standard does not define the GL_BGRA_EXT enumerant.

So based on the extension name #defines in <GL/gl.h>, you can write your code so that it can compile in the extension functionality if your compiler environment supports the extension's interfaces. The next problem is that even though your compiler environment may support the extension's interface at compile-time, at run-time, the target system where you run your application may not support the extension. In the Win32 environment, different OpenGL ICD drivers can support different OpenGL extensions depending on what the hardware and the vendor's ICD driver writers implement in the ICD driver.

Assuming that your application thread is made current to an OpenGL rendering context, the following routine can be used to determine at run-time if the OpenGL implementation really supports a particular extension:

```
#include <GL/gl.h>
#include <strength>

int
isExtensionSupported(const char *extension)
```

```
{
  const GLubyte *extensions = NULL;
  const GLubyte *start;
  GLubyte *where, *terminator;

  /* Extension names should not have spaces. */
  where = (GLubyte *) strchr(extension, ' ');
  if (where || *extension == '\0')
    return 0;

  extensions = glGetString(GL_EXTENSIONS);

  /* It takes a bit of care to be fool-proof about parsing the
     OpenGL extensions string.  Don't be fooled by sub-strings,
     etc. */
  start = extensions;
  for (;;) {
    where = (GLubyte *) strstr((const char *) start, extension);
    if (!where)
      break;
    terminator = where + strlen(extension);
    if (where == start || *(where - 1) == ' ')
      if (*terminator == ' ' || *terminator == '\0')
        return 1;
    start = terminator;
  }
  return 0;
}
```

With the `isExtensionSupported` routine, you can check if the current OpenGL rendering context supports a given OpenGL extension. To make sure that the EXT_bgra extension is supported before using it, you can do the following:

```
  /* At context initialization. */
  int hasBGRA = isExtensionSupported("GL_EXT_bgra");

  /* When trying to use EXT_bgra extension. */
#ifdef GL_EXT_bgra
  if (hasBGRA) {
    glDrawPixels(width, height, GL_BGRA_EXT, GL_UNSIGNED_BYTE, pixels);
  } else
#endif
  {
    /* No EXT_bgra so bail (or implement software workaround). */
    fprintf(stderr, "Needs EXT_bgra extension!\n");
    exit(1);
  }
```

Notice that if the `EXT_bgra` extension is lacking at either run-time or compile-time, the code above will detect the lack of `EXT_bgra` support. Sure the code is a bit messy, but the code above works. You can skip the compile-time check if you know what development environment you are using and you do not expect to ever compile with a <GL/gl.h> that does not support the extensions that your application uses. But the run-time check really should be performed since who knows on what system your program ends up getting run on.

5. Win32's Scheme for Getting Extension Function Pointers

The example above for safely detecting and using the EXT_bgra extension at run-time and compile-time is straightforward because the EXT_bgra simply adds two new enumerants (GL_BGRA_EXT and GL_BGR_EXT) and does not require any new function pointers.

Using an extension that includes new function call entry-points is harder in Win32 because you must first request the function pointer from the OpenGL ICD driver before you can call the OpenGL function.

The EXT_point_parameters extension provides eye-distance attenuation of OpenGL's point primitive. This extension is used by Id Software in Quake 2 when the extension is present for rendering particle systems. With the extension, firing weapon and explosions are rendered as huge clusters of OpenGL point primitives with OpenGL automatically adjusting the point size based on the distance of the particles from the viewer. Closer particles appear bigger; particles in the distance appear smaller. A particle whose size would be smaller than a pixel is automatically faded based on its sub-pixel size. Anyone that wants to see the improvement this extension brings to a 3D game should play Quake 2 on a PC with NVIDIA's RIVA 128 graphics processor. Start a gun battle and check out the particles!

The EXT_point_parameters extension adds two new OpenGL entry points called glPointParameterfEXT and glPointParameterfvEXT. These routines allow the application to specify the attenuation equation parameters and fade threshold. The problem is that because of the way Microsoft chose to support OpenGL extension functions, an OpenGL application cannot simply link with these functions. The application must first use the wglGetProcAddress[1] routine to query the function address and then call through the returned address to call the extension function.

First, declare function prototype typedefs that match the extension's entry points. For example:

```
#ifdef _WIN32
typedef void (APIENTRY * PFNGLPOINTPARAMETERFEXTPROC)
               (GLenum pname, GLfloat param);
typedef void (APIENTRY * PFNGLPOINTPARAMETERFVEXTPROC)
               (GLenum pname, const GLfloat *params);
#endif
```

Your <GL/gl.h> header file may already have these typedefs declared if your <GL/gl.h> defines the GL_EXT_point_parameters macro. Now declare global variables of the type of these function prototype typedefs like this:

```
#ifdef _WIN32
PFNGLPOINTPARAMETERFEXTPROC glPointParameterfEXT;
PFNGLPOINTPARAMETERFVEXTPROC glPointParameterfvEXT;
#endif
```

The names above exactly match the extension's function names. Once we use wglGetProcAddress to assign these function variables the address of the OpenGL driver's extension functions, we can call glPointParameterfEXT and glPointParameterfvEXT as if they were normal functions. You pass wglGetProcAddress the name of the routine as an ASCII string. Verify that the extension is supported and, if so, initialize the function variables like this:

[1] Note that wglGetProcAddress was introduced to Windows 95 in OEM Service Release 2.

```
  int hasPointParams = isExtensionSupported("GL_EXT_point_parameters");
#ifdef _WIN32
  if (hasPointParams) {
    glPointParameterfEXT = (PFNGLPOINTPARAMETERFEXTPROC)
      wglGetProcAddress("glPointParameterfEXT");
    glPointParameterfvEXT = (PFNGLPOINTPARAMETERFVEXTPROC)
      wglGetProcAddress("glPointParameterfvEXT");
  }
#endif
```

Note that before the code above is called, you should have a current OpenGL rendering context.

With the function variables properly initialized to the extension entry-points, you can use the extension like this:

```
  if (hasPointParams) {
    static GLfloat quadratic[3] = { 0.25, 0.0, 1/60.0 };
    glPointParameterfvEXT(GL_DISTANCE_ATTENUATION_EXT, quadratic);
    glPointParameterfEXT(GL_POINT_FADE_THRESHOLD_SIZE_EXT, 1.0);
  }
```

Be careful because the function returned by `wglGetProcAddress` is only be guaranteed to work for the OpenGL rendering context that was current when `wglGetProcAddress` was called. If you have multiple contexts that return different extension function addresses, keeping the function addresses in global variables as shown above may create problems. You may need to maintain distinct function addresses on a per-context basis. Specifically, the Microsoft documentation for `wglGetProcAddress` warns:

> The [Microsoft] OpenGL library supports multiple implementations of its functions. Extension functions supported in one rendering context are not necessarily available in a separate rendering context. Thus, for a given rendering context in an application, use the function addresses returned by the `wglGetProcAddress` function only.
>
> The spelling and the case of the extension function pointed to by string must be identical to that of a function supported and implemented by OpenGL. Because extension functions are not exported by OpenGL, you *must* use `wglGetProcAddress` to get the addresses of vendor-specific extension functions.
>
> The extension function addresses are unique for each pixel format. All rendering contexts of a given pixel format share the same extension function addresses.

Win32's requirement that you use `wglGetProcAddress` is a real drag, but if you do everything right, using OpenGL extensions works and gives you access to amazing new OpenGL features.

Amazing New OpenGL Features

So what OpenGL extensions are in the works to help OpenGL programmers write better high-performance, high-quality games and other 3D applications?

Here we review seven OpenGL extensions that are certain to be useful for PC game and 3D application programmers. The functionality of each extension will be briefly described in this section, but the appendixes below provide the complete extension specifications for the seven extensions. This provides you the opportunity to learn how to read and understand OpenGL extension specifications for yourself.

The first three extensions represent already finalized, implemented, and available extensions. The next four extensions are nearing finalization with at least one of the four (the multitexture extension) being partially implemented today. Hardware support for all four of these preliminary extensions will be available by the end of 1998.

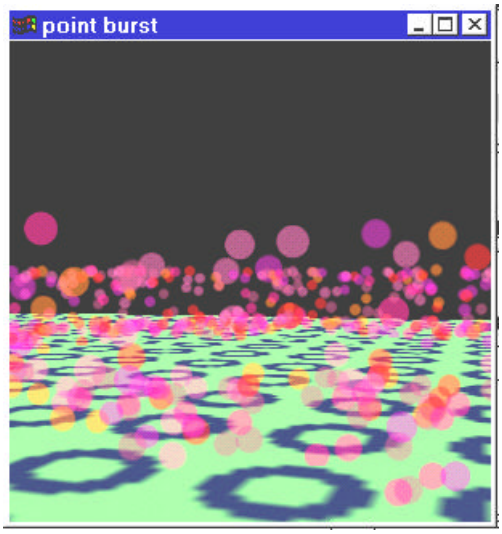


Figure 1 The pointburst demo uses the point parameters extension running on RIVA 128. The points have exploded outward in a circle from the center of the ground plane. The points closer to the viewer are large, while the far away points on the other side of the explosion source are small.

The Point Parameters Extension

The above discussion has already explained the basic functionality provided by the `EXT_point_parameters` extension. The extension was originally proposed by Silicon Graphics to address the needs of flight simulators for rendering point light sources such as landing lights that attenuate their brightness based on the distance from the viewer. Id Software also found the extension useful for rendering particle effects in Quake 2. In unextended OpenGL, the point primitive's size is controlled with the `glPointSize` routine. The point size is specified as a constant number of pixels. Because `glPointSize` can not be called within `glBegin` and `glEnd`, it is difficult to render efficiently a batch of points of various sizes to simulate a particle system such as water drops or exploding shrapnel.

What the extension provides is a means to attenuate the point size based on distance from the viewer. The further away the point primitive is from the viewer the smaller it should be rendered. If the point size becomes smaller than a pixel, the point's alpha component is attenuated based on the sub-pixel size to dim the point.

In addition to SGI's implementation of the extension for InfiniteReality, the extension has also been implemented in NVIDIA's OpenGL ICD driver for RIVA 128 and Brian Paul has implemented the extension in Mesa, the freeware implementation of the OpenGL programming interface. More implementations of the point parameters extension are expected. Figure 1 shows a snap shot of an OpenGL demo that uses the `EXT_point_parameters` extension.

Appendix A is the `EXT_point_parameters` specification.

The Paletted Texture Extension

Textures are typically 2D arrays of RGB or RGBA color values. For colorful textures with 8 bits of precision per color component (3 or 4 bytes per texel), large textures can eat up quite a bit of texture memory. The `EXT_paletted_texture` enables a texture to be specified as a 2D array of indices into a texture palette. Generally the indices are 8 bits per texel, but the texture palette itself contains full 24-bit or 32-bit color values. For textures that use 256 or fewer unique colors, a paletted texture can take up a lot less texture memory. Of course, the texture palette takes up some space too though.

Microsoft proposed the paletted texture extension to reduce the amount of texture memory needed by games and other 3D applications. Paletted textures also have the advantage that colors in the palette can be edited to change effectively the colors within the texture.

Appendix B is the `EXT_paletted_texture` specification.

The Shared Texture Palette Extension

The paletted texture extension provides a unique palette per texture, but this generality can make the management of texture palettes in hardware difficult. Since there may be just a single texture palette within the hardware rendering engine, the hardware may be continually loading the hardware texture palette on texture binds because each paletted texture maintains its own palette. By enabling the shared texture palette with `glEnable`, all the paletted textures of the rendering texture will share a single palette.

Appendix C is the `EXT_shared_texture_palette` specification.

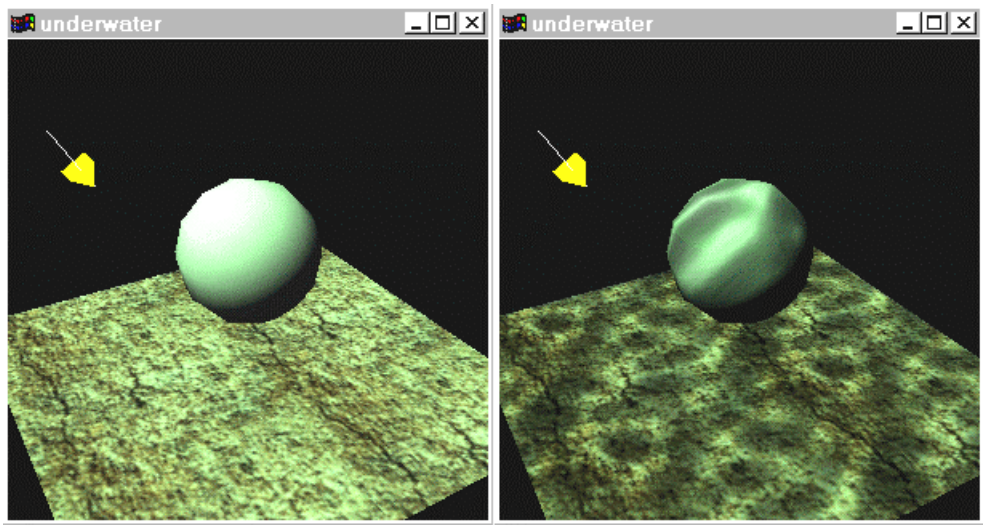


Figure 2 The right image uses an second rendering pass to combine a second texture with the textured floor. The second textured rendering pass cycles through a set of shifting caustic patterns to simulate the effect of underwater lighting. The left image shows the scene without the underwater effect from the second texture pass. With unextended OpenGL, the underwater effect requires two textured rendering passes, but with the OpenGL multitexture extension on hardware such as NVIDIA's multitexture-capable RIVA TNT graphics processor, the scene can be rendered with a single rendering pass.

The Multitexture Extension

The `SGIS_multitexture` extension provides the capability to specify multiple sets of texture coordinates that look up into multiple textures. Multitexture support will redefine the performance

and quality levels seen in tomorrow's 3D games. Extra rendering passes to blend in lightmaps, as done by Quake 2, can be performed in a single rendering pass with the multitexture extension. Multitexture is useful for all manner of cool effects, not just lightmaps. Figure 2 shows how multitexturing can be used to simulate a dynamic underwater caustic effect.

The SGIS_multitexture extension is very likely to be renamed the ARB_multitexture extension when it is finalized. The ARB prefix would indicate not just that multiple OpenGL vendors intend to implement the extension (that is what EXT means) but that the OpenGL Architectural Review Board, OpenGL's governing body, considers the multitexture functionality to be an important enough OpenGL capability to approve it as an ARB endorsed standard.

Appendix D is the *preliminary* SGIS_multitexture specification.

NVIDIA's Multitexture Combiners Extension

One problem with the SGIS_multitexture extension as specified is a straightforward, but simplistic, means of combining each texture with the results from the previous texture stage (see the ASCII diagram of this in the multitexture specification in Appendix D). Most of the vendors involved in the discussions of multitexturing for OpenGL agree that this strict pipeline model is too limiting for many interesting applications for multitexture. The difficulty is coming up with a more general approach to combining the results from multiple textures that all the vendors can agree on. Rather than argue, the vendors agreed to specify the initial simplistic pipeline model found in the current multitexture specification and hopefully find common ground in a future extension once multitexture hardware design was better understood.

NVIDIA has proposed its NVIDIA_multitexture_combiners extension in expectation of the flexible texture combining hardware found in NVIDIA's RIVA TNT graphics processor. Clever OpenGL programmers can use NVIDIA's combiner extension to implement sophisticated texture-based lighting models including bump mapping.

Appendix E is the preliminary NVIDIA_multitexture_combiners specification.

The Secondary Color Extension

When texturing and lighting are both enabled, OpenGL performs per-vertex lighting calculations that are then combined with the filtered texture result based on the texture environment. Before OpenGL 1.2 introduced the ideal of a separate specular color, OpenGL, as originally specified, computed the post-lighting per-vertex color by simply adding in the specular contribution as part of the per-vertex lighting equation. The unfortunate result with this approach is that the specular lighting contribution is typically modulated with the texture color. This means a bright specular highlight can wind up blended into a dark surface texture. A bright specular highlight on a surface appears "on top of" the surface texture. This is not very realistic. A more plausible lighting equation would add the specular contribution after the texture environment. Think about a specular highlight on an eight ball on a pool table. Even though the ball's surface texture is black, the highlight should still appear white.

OpenGL 1.2 provides for a primary color and a secondary color. When both lighting and OpenGL 1.2's new GL_SEPARATE_SPECULAR_COLOR state are enabled, the primary color is the result of the OpenGL's lighting equation *excluding* the specular contribution while the secondary color *is* the equation's specular contribution. Otherwise the secondary color is zero. The primary color is merged in the texture environment, and then the secondary color is added to the texture environment result. By adding the specular contribution after the texture environment, specular highlights appear "on top of" the surface texture.

OpenGL 1.2 added support for a secondary color, but the secondary color is only updated through OpenGL's lighting equations. The application programmer cannot directly assign the

specular color. Programmers who implement their own per-vertex lighting calculations (a common requirement in game engines) have no easy way in the OpenGL 1.2 specification to supply their own pre-computed per-vertex specular color. The secondary color extension adds the capability to directly specify the secondary color on a per-vertex basis.

Appendix F is the `EXT_secondary_color` specification.

The Fog Coordinate Extension

OpenGL specifies that fogging should be computed based on eye distance but also allows implementations to use the fragment's depth as an approximation of eye distance. The fog coordinate extension allows OpenGL applications to substitute OpenGL's eye distance (or depth based approximation of eye distance) with an explicitly specified *fog coordinate*. The fog coordinate is a single coordinate that can be specified per-vertex. Game programmers often like to specify the fog coordinate explicitly, either because they want better control of the fog equation or they intend to use the hardware's fog stage for some other devious purpose.

Support for an application settable per-vertex secondary color and fog coordinate are responses from feedback from game developers. Because Direct3D Immediate Mode supported explicit control of these rasterization parameters and therefore hardware designed for Direct3D already had the fundamental support for specifying these parameters on a per-vertex basis, it made sense to expose explicit control of these rasterization parameters through OpenGL extensions.

Both the fog coordinate and the secondary color can be passed both through immediate mode routines (`glSecondaryColor3fEXT` and `glFogCoordfEXT`) as well as through vertex arrays.

Conclusions

OpenGL continues to evolve its support for 3D game and applications programmers. OpenGL's extension mechanism provides a way to keep the simplicity of OpenGL's basic programming model while integrating innovative hardware capabilities into the API.

Compile-time and run-time checking for OpenGL extension support is necessary for robust OpenGL programs that use OpenGL extensions. Win32 makes accessing OpenGL extension functions more difficult because of the requirement to retrieve function pointers with wglGetProcAddress, but once the function entry-point addresses are retrieved, OpenGL extensions are easy to use with Win32.

The seven OpenGL extensions described above give OpenGL programmers new capabilities to control point size on a dynamic basis, to conserve texture memory usage, to utilize cutting-edge multitexture hardware, and gain explicit control over per-vertex parameters such as the specular color and fog coordinate.

The key to exploiting OpenGL extensions is reading and understanding the OpenGL extension specifications. An extension's specification is the definitive word on how a given extension should work. Review the seven extension specifications in the appendixes that follow and look on the Web for the specifications to scores of other available OpenGL extensions.

A. EXT_point_parameters Specification

Name
EXT_point_parameters

Name Strings
GL_EXT_point_parameters

Version
$Date: 1997/08/21 21:26:36 $ $Revision: 1.6 $

Number
54

Dependencies
SGIS_multisample affects the definition of this extension.

Overview
This extension supports additional geometric characteristics of points. It can be used to render particles or tiny light sources, commonly referred as "Light points".

The raster brightness of a point is a function of the point area, point color, point transparency, and the response of the display's electron gun and phosphor. The point area and the point transparency are derived from the point size, currently provided with the <size> parameter of glPointSize.

The primary motivation is to allow the size of a point to be affected by distance attenuation. When distance attenuation has an effect, the final point size decreases as the distance of the point from the eye increases.

The secondary motivation is a mean to control the mapping from the point size to the raster point area and point transparency. This is done in order to increase the dynamic range of the raster brightness of points. In other words, the alpha component of a point may be decreased (and its transparency increased) as its area shrinks below a defined threshold.

This extension defines a derived point size to be closely related to point brightness. The brightness of a point is given by:

$$dist\_atten(d) = \frac{1}{a + b * d + c * d^2}$$

$$brightness(Pe) = Brightness * dist\_atten(|Pe|)$$

where 'Pe' is the point in eye coordinates, and 'Brightness' is some initial value proportional to the square of the size provided with glPointSize. Here we simplify the raster brightness to be a function of the rasterized point area and point transparency.

$$area(Pe) = \begin{cases} brightness(Pe) & brightness(Pe) >= Threshold\_Area \\ Threshold\_Area & Otherwise \end{cases}$$

$$factor(Pe) = brightness(Pe)/Threshold\_Area$$

$$alpha(Pe) = Alpha * factor(Pe)$$

where 'Alpha' comes with the point color (possibly modified by lighting).

'Threshold_Area' above is in area units. Thus, it is proportional to the square of the threshold provided by the programmer through this extension.

The new point size derivation method applies to all points, while the threshold applies to multisample points only.

Issues
* Does point alpha modification affect the current color ?

No.

* Do we need a special function glGetPointParameterfvEXT, or get by with glGetFloat ?
No.

* If alpha is 0, then we could toss the point before it reaches the fragment stage.

No. This can be achieved with enabling the alpha test with reference of 0 and function of LEQUAL.

* Do we need a disable for applying the threshold ? The default threshold value is 1.0. It is applied even if the point size is constant.

If the default threshold is not overridden, the area of multisample points with provided constant size of less than 1.0, is mapped to 1.0, while the alpha component is modulated accordingly, to compensate for the larger area. For multisample points this is not a problem, as there are no relevant applications yet. As mentioned above, the threshold does not apply to alias or antialias points.

The alternative is to have a disable of threshold application, and state that threshold (if not disabled) applies to non antialias points only (that is, alias and multisample points).

The behavior without an enable/disable looks fine.

* Future extensions (to the extension)

1. GL_POINT_FADE_ALPHA_CLAMP_EXT

When the derived point size is larger than the threshold size defined by the GL_POINT_FADE_THRESHOLD_SIZE_EXT parameter, it might be desired to clamp the computed alpha to a minimum value, in order to keep the point visible. In this case the formula below change:

factor = (derived_size/threshold)^2

```
                factor                  clamp <= factor
clamped_value =
                clamp                   factor < clamp

         1.0                            derived_size >= threshold
alpha *=
         clamped_value                  Otherwise
```

where clamp is defined by the GL_POINT_FADE_ALPHA_CLAMP_EXT new parameter.

New Procedures and Functions

```
void glPointParameterfEXT ( GLenum pname, GLfloat param );
void glPointParameterfvEXT ( GLenum pname, GLfloat *params );
```

New Tokens

Accepted by the <pname> parameter of glPointParameterfEXT, and the <pname> of glGet:

```
GL_POINT_SIZE_MIN_EXT
GL_POINT_SIZE_MAX_EXT
GL_POINT_FADE_THRESHOLD_SIZE_EXT
```

Accepted by the <pname> parameter of glPointParameterfvEXT, and the <pname> of glGet:

```
GL_POINT_SIZE_MIN_EXT              0x8126
GL_POINT_SIZE_MAX_EXT              0x8127
GL_POINT_FADE_THRESHOLD_SIZE_EXT   0x8128
GL_DISTANCE_ATTENUATION_EXT        0x8129
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

All parameters of the glPointParameterfEXT and glPointParameterfvEXT functions set various values applied to point rendering. The derived point size is defined to be the <size> provided with glPointSize modulated with a distance attenuation factor.

The parameters GL_POINT_SIZE_MIN_EXT and GL_POINT_SIZE_MAX_EXT simply define an upper and lower bounds respectively on the derived point size.

The above parameters affect non-multisample points as well as multisample points, while the GL_POINT_FADE_THRESHOLD_SIZE_EXT parameter, has no effect on non multisample points. If the derived point size is larger than the threshold size defined by the GL_POINT_FADE_THRESHOLD_SIZE_EXT parameter, the derived point size is used as the diameter of the rasterized point, and the alpha component is intact. Otherwise, the threshold size is set to be the diameter of the rasterized point, while the alpha component is modulated accordingly, to compensate for the larger area.

The distance attenuation function coefficients, namely a, b, and c in:

$$dist\_atten(d) = \frac{1}{a + b * d + c * d^2}$$

are defined by the <pname> parameter GL_DISTANCE_ATTENUATION_EXT of the function glPointParameterfvEXT. By default a = 1, b = 0, and c = 0.

Let 'size' be the point size provided with glPointSize, let 'dist' be the distance of the point from the eye, and let 'threshold' be the threshold size defined by the GL_POINT_FADE_THRESHOLD_SIZE parameter of glPointParameterfEXT. The derived point size is given by:

```
derived_size = size * sqrt(dist_atten(dist))
```

Note that when default values are used, the above formula reduces to:

```
derived_size = size
```

the diameter of the rasterized point is given by:

$$diameter = \begin{cases} derived\_size & derived\_size >= threshold \\ threshold & Otherwise \end{cases}$$

The alpha of a point is calculated to allow the fading of points instead of shrinking them past a defined threshold size. The alpha component of the rasterized point is given by:

$$alpha \mathrel{*}= \begin{cases} 1 & derived\_size >= threshold \\ (derived\_size/threshold)^2 & Otherwise \end{cases}$$

The threshold defined by GL_POINT_FADE_THRESHOLD_SIZE_EXT is not clamped to the minimum and maximum point sizes.

Points do not affect the current color.

This extension doesn't change the feedback or selection behavior of points.

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on SGIS_multisample

If SGIS_multisample is not implemented, then the references to multisample points are invalid, and should be ignored.

Errors

INVALID_ENUM is generated if PointParameterfEXT parameter <pname> is not GL_POINT_SIZE_MIN_EXT, GL_POINT_SIZE_MAX_EXT, or GL_POINT_FADE_THRESHOLD_SIZE_EXT.

INVALID_ENUM is generated if PointParameterfvEXT parameter <pname> is not GL_POINT_SIZE_MIN_EXT, GL_POINT_SIZE_MAX_EXT, GL_POINT_FADE_THRESHOLD_SIZE_EXT, or GL_DISTANCE_ATTENUATION_EXT

INVALID_VALUE is generated when values are out of range according to:

<pname>	valid range
GL_POINT_SIZE_MIN_EXT	>= 0
GL_POINT_SIZE_MAX_EXT	>= 0
GL_POINT_FADE_THRESHOLD_SIZE_EXT	>= 0

Issues

- should we generate INVALID_VALUE or just clamp?

New State

Get Value	Get Command	Type	Initial Value	Attribute
GL_POINT_SIZE_MIN_EXT	GetFloatv	R	0	point
GL_POINT_SIZE_MAX_EXT	GetFloatv	R	M	point
GL_POINT_FADE_THRESHOLD_SIZE_EXT	GetFloatv	R	1	point
GL_DISTANCE_ATTENUATION_EXT	GetFloatv	3xR	(1,0,0)	point

M is the largest available point size.

New Implementation Dependent State

None

Backward Compatibility

This extension replaces SGIS_point_parameters. The procedures, tokens, and name strings now refer to EXT instead of SGIS. Enumerant values are unchanged. SGI implementations which previously provided this functionality should support both forms of the extension.

B. Paletted Texture Specification

Name

EXT_paletted_texture

Name Strings

GL_EXT_paletted_texture

Version

$Date: 1997/06/12 01:07:42 $ $Revision: 1.2 $

Number

78

Dependencies

GL_EXT_paletted_texture shares routines and enumerants with GL_SGI_color_table with the minor modification that EXT replaces SGI. In all other ways these calls should function in the same manner and the enumerant values should be identical. The portions of GL_SGI_color_table that are used are:

ColorTableSGI, GetColorTableSGI, GetColorTableParameterivSGI, GetColorTableParameterfvSGI.
COLOR_TABLE_FORMAT_SGI, COLOR_TABLE_WIDTH_SGI, COLOR_TABLE_RED_SIZE_SGI, COLOR_TABLE_GREEN_SIZE_SGI, COLOR_TABLE_BLUE_SIZE_SGI, COLOR_TABLE_ALPHA_SIZE_SGI, COLOR_TABLE_LUMINANCE_SIZE_SGI, COLOR_TABLE_INTENSITY_SIZE_SGI.

Portions of GL_SGI_color_table which are not used in GL_EXT_paletted_texture are:

CopyColorTableSGI, ColorTableParameterivSGI, ColorTableParameterfvSGI.
COLOR_TABLE_SGI, POST_CONVOLUTION_COLOR_TABLE_SGI, POST_COLOR_MATRIX_COLOR_TABLE_SGI, PROXY_COLOR_TABLE_SGI, PROXY_POST_CONVOLUTION_COLOR_TABLE_SGI, PROXY_POST_COLOR_MATRIX_COLOR_TABLE_SGI, COLOR_TABLE_SCALE_SGI, COLOR_TABLE_BIAS_SGI.

EXT_paletted_texture can be used in conjunction with EXT_texture3D. EXT_paletted_texture modifies TexImage3DEXT to accept paletted image data and allows TEXTURE_3D_EXT and PROXY_TEXTURE_3D_EXT to be used a targets in the color table routines. If EXT_texture3D is unsupported then references to 3D texture support in this spec are invalid and should be ignored.

Overview

EXT_paletted_texture defines new texture formats and new calls to support the use of paletted textures in OpenGL. A paletted texture is defined by giving both a palette of colors and a set of image data which is composed of indices into the palette. The paletted texture cannot function properly without both pieces of information so it increases the work required to define a texture. This is offset by the fact that the overall amount of texture data can be reduced dramatically by factoring redundant information out of the logical view of the texture and placing it in the palette.

Paletted textures provide several advantages over full-color textures:

* As mentioned above, the amount of data required to define a texture can be greatly reduced over what would be needed for full-color specification. For example, consider a source texture that has only 256 distinct colors in a 256 by 256 pixel grid. Full-color representation requires three bytes per pixel, taking 192K of texture data. By putting the distinct colors in a palette only eight bits are required per pixel, reducing the 192K to 64K plus 768 bytes for the palette. Now add an alpha channel to the texture. The full-color representation increases by 64K while the paletted version would only increase by 256 bytes. This reduction in space required is particularly important for hardware accelerators where texture space is limited.

```
    * Paletted textures allow easy reuse of texture data for images
    which require many similar but slightly different colored objects.
    Consider a driving simulation with heavy traffic on the road.  Many of
    the cars will be similar but with different color schemes.  If
    full-color textures are used a separate texture would be needed for each
    color scheme, while paletted textures allow the same basic index data to
    be reused for each car, with a different palette to change the final
    colors.

    * Paletted textures also allow use of all the palette tricks
    developed for paletted displays.  Simple animation can be done, along
    with strobing, glowing and other palette-cycling effects.  All of these
    techniques can enhance the visual richness of a scene with very little
    data.
```

New Procedures and Functions

```
    void ColorTableEXT(
        enum target,
        enum internalFormat,
        sizei width,
        enum format,
        enum type,
        const void *data);

    void ColorSubTableEXT(
        enum target,
        sizei start,
        sizei count,
        enum format,
        enum type,
        const void *data);

    void GetColorTableEXT(
        enum target,
        enum format,
        enum type,
        void *data);

    void GetColorTableParameterivEXT(
        enum target,
        enum pname,
        int *params);

    void GetColorTableParameterfvEXT(
        enum target,
        enum pname,
        float *params);
```

New Tokens

```
    Accepted by the internalformat parameter of TexImage1D, TexImage2D and
    TexImage3DEXT:
        COLOR_INDEX1_EXT                0x80E2
        COLOR_INDEX2_EXT                0x80E3
        COLOR_INDEX4_EXT                0x80E4
        COLOR_INDEX8_EXT                0x80E5
        COLOR_INDEX12_EXT               0x80E6
        COLOR_INDEX16_EXT               0x80E7

    Accepted by the pname parameter of GetColorTableParameterivEXT and
    GetColorTableParameterfvEXT:
        COLOR_TABLE_FORMAT_EXT          0x80D8
        COLOR_TABLE_WIDTH_EXT           0x80D9
        COLOR_TABLE_RED_SIZE_EXT        0x80DA
        COLOR_TABLE_GREEN_SIZE_EXT      0x80DB
        COLOR_TABLE_BLUE_SIZE_EXT       0x80DC
        COLOR_TABLE_ALPHA_SIZE_EXT      0x80DD
        COLOR_TABLE_LUMINANCE_SIZE_EXT  0x80DE
        COLOR_TABLE_INTENSITY_SIZE_EXT  0x80DF

    Accepted by the value parameter of GetTexLevelParameter{if}v:
```

```
        TEXTURE_INDEX_SIZE_EXT          0x80ED
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)
None

Additions to Chapter 3 of the GL Specification (Rasterization)

Section 3.6.4, 'Pixel Transfer Operations,' subsection 'Color Index Lookup,'

Point two is modified from 'The groups will be loaded as an image into texture memory' to 'The groups will be loaded as an image into texture memory and the internalformat parameter is not one of the color index formats from table 3.8.'

Section 3.8, 'Texturing,' subsection 'Texture Image Specification' is modified as follows:

The portion of the first paragraph discussing interpretation of format, type and data is split from the portion discussing target, width and height. The target, width and height section now ends with the sentence 'Arguments width and height specify the image's width and height.'

The format, type and data section is moved under a subheader 'Direct Color Texture Formats' and begins with 'If internalformat is not one of the color index formats from table 3.8,' and continues with the existing text through the internalformat discussion.

After that section, a new section 'Paletted Texture Formats' has the text:

If format is given as COLOR_INDEX then the image data is composed of integer values representing indices into a table of colors rather than colors themselves. If internalformat is given as one of the color index formats from table 3.8 then the texture will be stored internally as indices rather than undergoing index-to-RGBA mapping as would previously have occurred. In this case the only valid values for type are BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT and UNSIGNED_INT.

The image data is unpacked from memory exactly as for a DrawPixels command with format of COLOR_INDEX for a context in color index mode. The data is then stored in an internal format derived from internalformat. In this case the only legal values of internalformat are COLOR_INDEX1_EXT, COLOR_INDEX2_EXT, COLOR_INDEX4_EXT, COLOR_INDEX8_EXT, COLOR_INDEX12_EXT and COLOR_INDEX16_EXT and the internal component resolution is picked according to the index resolution specified by internalformat. Any excess precision in the data is silently truncated to fit in the internal component precision.

An application can determine whether a particular implementation supports a particular paletted format (or any paletted formats at all) by attempting to use the paletted format with a proxy target. TEXTURE_INDEX_SIZE_EXT will be zero if the implementation cannot support the texture as given.

An application can determine an implementation's desired format for a particular paletted texture by making a TexImage call with COLOR_INDEX as the internalformat, in which case target must be a proxy target. After the call the application can query TEXTURE_INTERNAL_FORMAT to determine what internal format the implementation suggests for the texture image parameters. TEXTURE_INDEX_SIZE_EXT can be queried after such a call to determine the suggested index resolution numerically. The index resolution suggested by the implementation does not have to be as large as the input data precision. The resolution may also be zero if the implementation is unable to support any paletted format for the given texture image.

Table 3.8 should be augmented with a column titled 'Index bits.' All existing formats have zero index bits. The following formats are added with zeroes in all existing columns:

Name	Index bits

```
            COLOR_INDEX1_EXT              1
            COLOR_INDEX2_EXT              2
            COLOR_INDEX4_EXT              4
            COLOR_INDEX8_EXT              8
            COLOR_INDEX12_EXT             12
            COLOR_INDEX16_EXT             16

At the end of the discussion of level the following text should be
added:

  All mipmapping levels share the same palette.  If levels
  are created with different precision indices then their internal formats
  will not match and the texture will be inconsistent, as discussed above.

In the discussion of internalformat for CopyTexImage{12}D, at end of the
sentence specifying that 1, 2, 3 and 4 are illegal there should also be
a mention that paletted internalformat values are illegal.

At the end of the width, height, format, type and data section under
TexSubImage there should be an additional sentence:

  If the target texture has an color index internal format
  then format may only be COLOR_INDEX.

At the end of the first paragraph describing TexSubImage and
CopyTexSubImage the following sentence should be added:

  If the target of a CopyTexSubImage is a paletted texture
  image then INVALID_OPERATION is returned.

After the Alternate Image Specification Commands section, a new 'Palette
Specification Commands' section should be added.

  Paletted textures require palette information to
  translate indices into full colors.  The command
    void ColorTableEXT(enum target, enum internalformat, sizei width,
             enum format, enum type, const void *data);
  is used to specify the format and size of the palette
  for paletted textures.  target specifies which texture is to have its
  palette changed and may be one of TEXTURE_1D, TEXTURE_2D,
  PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, TEXTURE_3D_EXT or
  PROXY_TEXTURE_3D_EXT.  internalformat specifies the desired format and
  resolution of the palette when in its internal form.  internalformat can
  be any of the non-index values legal for TexImage internalformat
  although implementations are not required to support palettes of all
  possible formats.  width controls the size of the palette and must be a
  power of two greater than or equal to one.  format and type specify the
  number of components and type of the data given by data.  format can be
  any of the formats legal for DrawPixels although implementations are not
  required to support all possible formats.  type can be any of the types
  legal for DrawPixels except GL_BITMAP.

  Data is taken from memory and converted just as if each
  palette entry were a single pixel of a 1D texture.  Pixel unpacking and
  transfer modes apply just as with texture data.  After unpacking and
  conversion the data is translated into a internal format that matches
  the given format as closely as possible.  An implementation does not,
  however, have a responsibility to support more than one precision for
  the base formats.

  If the palette's width is greater than the range of
  the color indices in the texture data then some of the palettes entries
  will be unused.  If the palette's width is less than the range of the
  color indices in the texture data then the most-significant bits of the
  texture data are ignored and only the appropriate number of bits of the
  index are used when accessing the palette.

  Specifying a proxy target causes the proxy texture's
  palette to be resized and its parameters set but no data is transferred
  or accessed.  If an implementation cannot handle the palette data given
  in the call then the color table width and component resolutions are set
```

to zero.

Portions of the current palette can be replaced with
  void ColorSubTableEXT(enum target, sizei start, sizei count,
           enum format, enum type, const void *data);
target can be any of the non-proxy values legal for
ColorTableEXT. start and count control which entries of the palette are changed out of the range allowed by the internal format used for the palette indices. count is silently clamped so that all modified entries all within the legal range. format and type can be any of the values legal for ColorTableEXT. The data is treated as a 1D texture just as in ColorTableEXT.

In the 'Texture State and Proxy State' section the sentence fragment beginning 'six integer values describing the resolutions...' should be changed to refer to seven integer values, with the seventh being the index resolution.

Palette data should be added in as a third category of texture state.

After the discussion of properties, the following should be added:

Next there is the texture palette. All textures have a palette, even if their internal format is not color index. A texture's palette is initially one RGBA element with all four components set to 1.0.

The sentence mentioning that proxies do not have image data or properties should be extended with 'or palettes.'

The sentence beginning 'If the texture array is too large' describing the effects of proxy failure should change to read:

If the implementation is unable to handle the texture image data the proxy width, height, border width and component resolutions are set to zero. This situation can occur when the texture array is too large or an unsupported paletted format was requested.

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

In the section on GetTexImage, the sentence saying 'The components are assigned among R, G, B and A according to' should be changed to be

If the internal format of the texture is not a color index format then the components are assigned among R, G, B, and A according to Table 6.1. Specifying COLOR_INDEX for format in this case will generate the error INVALID_ENUM. If the internal format of the texture is color index then the components are handled in one of two ways depending on the value of format. If format is not COLOR_INDEX, the texture's indices are passed through the texture's palette and the resulting components are assigned among R, G, B, and A according to Table 6.1. If format is COLOR_INDEX then the data is treated as single components and the palette indices are returned. Components are taken starting...

Following the GetTexImage section there should be a new section:

GetColorTableEXT is used to get the current texture palette.
  void GetColorTableEXT(enum target, enum format, enum type, void *data);

GetColorTableEXT retrieves the texture palette of the texture given by target. target can be any of the non-proxy targets

```
      valid for ColorTableEXT.  format and type are interpreted just as for
      ColorTableEXT.  All textures have a palette by default so
      GetColorTableEXT will always be able to return data even if the internal
      format of the texture is not a color index format.

      Palette parameters can be retrieved using
        void GetColorTableParameterivEXT(enum target, enum pname, int *params);
        void GetColorTableParameterfvEXT(enum target, enum pname, float *params);
      target specifies the texture being queried and pname
      controls which parameter value is returned.  Data is returned in the
      memory pointed to by params.

      Querying COLOR_TABLE_FORMAT_EXT returns the internal
      format requested by the most recent ColorTableEXT call or the default.
      COLOR_TABLE_WIDTH_EXT returns the width of the current palette.
      COLOR_TABLE_RED_SIZE_EXT, COLOR_TABLE_GREEN_SIZE_EXT,
      COLOR_TABLE_BLUE_SIZE_EXT and COLOR_TABLE_ALPHA_SIZE_EXT return the
      actual size of the components used to store the palette data internally,
      not the size requested when the palette was defined.

    Table 6.11, "Texture Objects" should have a line appended for
    TEXTURE_INDEX_SIZE_EXT:

TEXTURE_INDEX_SIZE_EXT  n x Z+  GetTexLevelParameter 0  xD texture image i's index resolution 3.8 -


Revision History

Original draft, revision 0.5, December 20, 1995 (drewb) Created

Minor revisions and clarifications, revision 0.6, January 2, 1996 (drewb)
    Replaced all request-for-comment blocks with final text
    based on implementation.

Minor revisions and clarifications, revision 0.7, February 5, 1996 (drewb)
    Specified the state of the palette color information
    when existing data is replaced by new data.

    Clarified behavior of TexPalette on inconsistent textures.

Major changes due to ARB review, revision 0.8, March 1, 1996 (drewb)
    Switched from using TexPaletteEXT and GetTexPaletteEXT
    to using SGI's ColorTableEXT routines.  Added ColorSubTableEXT so
    equivalent functionality is available.

    Allowed proxies in all targets.

    Changed PALETTE?_EXT values to COLOR_INDEX?_EXT.  Added
    support for one and two bit palettes.  Removed PALETTE_INDEX_EXT in
    favor of COLOR_INDEX.

    Decoupled palette size from texture data type.  Palette
    size is controlled only by ColorTableEXT.

Changes due to ARB review, revision 1.0, May 23, 1997 (drewb)
    Mentioned texture3D.

    Defined TEXTURE_INDEX_SIZE_EXT.

    Allowed implementations to return an index size of zero to indicate
    no support for a particular format.

    Allowed usage of GL_COLOR_INDEX as a generic format in
    proxy queries for determining an optimal index size for a particular
    texture.

    Disallowed CopyTexImage and CopyTexSubImage to paletted formats.

    Deleted mention of index transfer operations during GetTexImage with
    paletted formats.
```

C. Shared Texture Palette

Name
EXT_shared_texture_palette

Name Strings
GL_EXT_shared_texture_palette

Version
$Date: 1997/09/10 23:23:04 $ $Revision: 1.2 $

Number
141

Dependencies
EXT_paletted_texture is required.

Overview
EXT_shared_texture_palette defines a shared texture palette which may be used in place of the texture object palettes provided by EXT_paletted_texture. This is useful for rapidly changing a palette common to many textures, rather than having to reload the new palette for each texture. The extension acts as a switch, causing all lookups that would normally be done on the texture's palette to instead use the shared palette.

Issues

* Do we want to use a new <target> to ColorTable to specify the shared palette, or can we just infer the new target from the corresponding Enable?

* A future extension of larger scope might define a "texture palette object" and bind these objects to texture objects dynamically, rather than making palettes part of the texture object state as the current EXT_paletted_texture spec does.

* Should there be separate shared palettes for 1D, 2D, and 3D textures?

  Probably not; palette lookups have nothing to do with the dimensionality of the texture. If multiple shared palettes are needed, we should define palette objects.

* There's no proxy mechanism for checking if a shared palette can be defined with the requested parameters. Will it suffice to assume that if a texture palette can be defined, so can a shared palette with the same parameters?

* The changes to the spec are based on changes already made for EXT_paletted_texture, which means that all three documents must be referred to. This is quite difficult to read.

* The changes to section 3.8.6, defining how shared palettes are enabled and disabled, might be better placed in section 3.8.1. However, the underlying EXT_paletted_texture does not appear to modify these sections to define exactly how palette lookups are done, and it's not clear where to put the changes.

New Procedures and Functions
None

New Tokens
Accepted by the <pname> parameters of GetBooleanv, GetIntegerv, GetFloatv, GetDoublev, IsEnabled, Enable, Disable, ColorTableEXT, ColorSubTableEXT, GetColorTableEXT, GetColorTableParameterivEXT, and GetColorTableParameterfd EXT:

```
SHARED_TEXTURE_PALETTE_EXT              0x81FB
```

Additions to Chapter 2 of the 1.1 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.1 Specification (Rasterization)

Section 3.8, 'Texturing,' subsection 'Texture Image Specification' is modified as follows:

In the Palette Specification Commands section, the sentence beginning 'target specifies which texture is to' should be changed to:

> target specifies the texture palette or shared palette to be changed, and may be one of TEXTURE_1D, TEXTURE_2D, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, TEXTURE_3D_EXT, PROXY_TEXTURE_3D_EXT, or SHARED_TEXTURE_PALETTE_EXT.

In the 'Texture State and Proxy State' section, the sentence beginning 'A texture's palette is initially...' should be changed to:

> There is also a shared palette not associated with any texture, which may override a texture palette. All palettes are initially...

Section 3.8.6, 'Texture Application' is modified by appending the following:

Use of the shared texture palette is enabled or disabled using the generic Enable or Disable commands, respectively, with the symbolic constant SHARED_TEXTURE_PALETTE_EXT.

The required state is one bit indicating whether the shared palette is enabled or disabled. In the initial state, the shared palettes is disabled.

Additions to Chapter 4 of the 1.1 Specification (Per-Fragment Operations and the Frame buffer)

Additions to Chapter 5 of the 1.1 Specification (Special Functions)

Additions to Chapter 6 of the 1.1 Specification (State and State Requests)

In the section on GetTexImage, the sentence beginning 'If format is not COLOR_INDEX...' should be changed to:

> If format is not COLOR_INDEX, the texture's indices are passed through the texture's palette, or the shared palette if one is enabled, and the resulting components are assigned among R, G, B, and A according to Table 6.1.

In the GetColorTable section, the first sentence of the second paragraph should be changed to read:

> GetColorTableEXT retrieves the texture palette or shared palette given by target.

The first sentence of the third paragraph should be changed to read:

> Palette parameters can be retrieved using
> void GetColorTableParameterivEXT(enum target, enum pname, int *params);
> void GetColorTableParameterfvEXT(enum target, enum pname, float *params);
> target specifies the texture palette or shared palette being queried and pname controls which parameter value is returned.

Additions to the GLX Specification

None

New State

Get Value	Get Command	Type	Initial Value	Attribute
SHARED_TEXTURE_PALETTE_EXT	IsEnabled	B	False	texture/enable

New Implementation Dependent State
None

D. Multitexture Specification

XXX – Preliminary

Name
SGIS_multitexture

Name Strings
GL_SGIS_multitexture

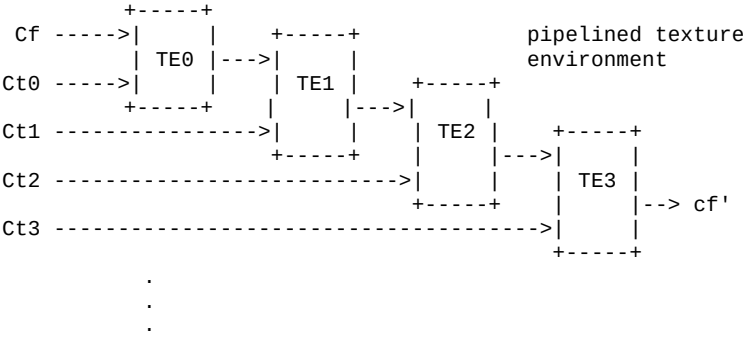
Version
$Date: 1998/04/10 06:42:49 $ $Revision: 1.15 $

Number
116

Dependencies
OpenGL 1.1 is required
EXT_texture3D affects the definition of this extension.
SGIS_texture4D affects the definition of this extension.
SGIS_texture_border_clamp affects the definition of this extension.
SGI_texture_color_table affects the definition of this extension.
SGIS_texture_edge_clamp affects the definition of this extension.
SGIX_texture_add_env affects the definition of this extension.
SGIS_texture_filter4 affects the definition of this extension.
SGIS_texture_lod affects the definition of this extension.
SGIX_texture_lod_bias affects the definition of this extension.
SGIX_texture_scale_bias affects the definition of this extension.
SGIS_texture_select affects the definition of this extension.
SGIS_detail_texture affects the definition of this extension.
SGIS_sharpen_texture affects the definition of this extension.
SGIX_shadow affects the definition of this extension.
SGIX_shadow_ambient affects the definition of this extension.
SGIX_clipmap affects the definition of this extension.
SGIS_point_line_texgen affects the definition of this extension.

Overview
This extension adds support for multiple active textures. The texture capabilities are symmetric for all active textures. Any texture capability extension supported for one texture must be supported for all active textures. Each active texture has its own state vector which includes texture image and filtering parameters and texture environment application.

The texture environments are applied in a pipelined fashion whereby the output of one texture environment is used as the input fragment color for the texture environment for the next active texture. Changes to texture state other than texture coordinates are routed through a selector which controls which instance of texture state is affected.



Ct<i> = texture color from texture lookup <i>
Cf = fragment color
TE = texture environment

Texture coordinate set, texture coordinate evaluator state, texture

```
    generation function, and texture matrix are replicated independently of
    the texture rasterization state and may differ in number from the
    number of textures which can be active during rasterization.
    Post-transform texture coordinates sets are associated with a texture
    rasterization unit by binding them to a texture environment and they may
    be multicast to several texture rasterization units.

    The specification is written using four active textures and four
    sets of texture coordinates though the actual number supported is
    implementation dependent and can be larger or smaller than four.

Issues
    *  MultiTexCoord is an annoying name

    *  alternatives for supplying fine grain texcoord

       1. Tex<k>Coord<n><T>[v|f](<T> data);
          a. efficient, no error checking required
          d. adds *a lot* of new commands

       2. MultiTexCoord<n><T>[v|f](enum target, <T> data);
          a. only a small number of commands added
          a. can be fairly efficient (may need hw tweak)
          d. needs range checking for <target>

       3. reuse TexCoord command and add SelectTextureCoordSetSGIS(enum target)
          to control routing
          a. only add one new commands
          d. adds a lot of function call overhead when using multiple
             textures
          d. need to range check <target>

    *  seems a little hacky to have SelectTextureSGIS control texture matrix
       since that is part of transform state and to have it control evaluator
       state yet SELECTED_TEXTURE itself is part of texture state.

    *  SelectTextureSGIS probably should not affect client state such as
       the vertex array state.

       it doesn't any more

    *  mechanism to replicate input texcoords across multiple texture paths
       could be done with a pre-transform multicast or post-transform
       multicast.

       done using TEXTURE_ENV_COORD_SET_SGIS texture parameter
       which is a post-transform mechanism.

       RESOLVED: leave the coord source binding separate from
       the texture object state => needs a new command to set it.

    *  need proxy/macro object to handle resource constraints
       save for another extension?

    *  still need a way to route textures to lighting block :(

       defined in light_texture.spec

    *  should there be a post-filter colortable per texture?

    *  should the number of textures and the number of texture
       coordinate paths be decoupled?
       RESOLVED: yes
       There are some issues with this.  We choose to break
       texture state into 3 pieces:
           1. client state deal with issuing texture coordinates
              from the application
           2. transform state which includes texgen, texture
              matrix, evaluation maps, and texture coordinate
              retrieval from Gets and Feedback.
           3. rasterization state which includes texture
```

```
            images, filter parameters and environment.
        2 & 3 are both server state.  there is an implication
        that 1 and 2 are a little more tightly coupled and
        equal in number but we need to keep the client
        state separate.
        There is some clumsiness with referring to the 2nd
        group of state as transform state.  There is a problem
        that the texgen state is part of the texture state
        used in PushAttrib and PopAttrib so some finessing
        is required.

    *  special treatment of name 0?
       RESOLVED: no

    *  more texture environment functions, SUBTRACT, ...?
       leading candidates are SUBTRACT and REVERSE_SUBTRACT
       could also make a new version of environment which is
       similar to blending.
       RESOLVED: new environment, see texture_env.spec

    * more general combination of texture results?
       RESOLVED: do them in add-on specs

    * allow texture environment computation to do something
      even when texture is disabled. This contradicts the current
      specification of texturing (the difference would show
      up in the REPLACE environment), so we redefine this
      behavior in a new environment (see texture_env.spec)

    * support for interleaved arrays

      add a command which acts as a multiplier on the current
      interleaved array token causing the texture coordinate
      array to have <n> contiguous texture coords of the same
      type and format.

    * some clarifications:

      SelectTextureCoordSetSGIS affects client state only and
      affects the commands TexCoord<n>{T}[v], TexCoordPointer,
      EnableClientState, and DisableClientState.  Display lists
      contain texture coordinates for which the binding is fully
      resolved to one of TEXTURE0_SGIS ..  TEXTURE<n>_SGIS.

      I chose to remove MultiTexCoordPointerSGIS as it was difficult
      to also include tokens which would make it possible to call
      Enable/DisableClientState with a token corresponding to the
      appropriate texture coordinate set, so SelectTextureCoordSetSGIS
      is required to manipulate the array state.  To maintain symmetry,
      I made all commands use SelectTextureCoordSetSGIS and the
      MultiTexCoord<n>{T}[v]SGIS commands are added to help with
      performance.  An alternative would be to have both
      MultiTexCoordPointerSGIS and add new tokens
      TEXTURE_COORD_ARRAY0_SGIS .. TEXTURE_COORD_ARRAY<n>_SGIS and
      not give TEXTURE_COORD_ARRAY0_SGIS the same value as
      TEXTURE_COORD_ARRAY, so that we can have the relationship
      TEXTURE_COORD_ARRAY<i>_SGIS = TEXTURE_COORD_ARRAY0+SGIS+i.
      This still might cause some confusion/asymmetry if the <target>
      parameter of MultiTexCoordPointerSGIS/MultiTexCoord<n>{T}[v]SGIS
      is TEXTURE0_SGIS .. TEXTURE<n>_SGIS but EnableClientState/
      DisableClientState use TEXTURE_COORD_ARRAY0_SGIS ..
      TEXTURE_COORD_ARRAY<n>_SGIS
```

New Procedures and Functions

```
    void MultiTexCoord1dSGIS(enum target, double s);
    void MultiTexCoord1dvSGIS(enum target, const double *v);
    void MultiTexCoord1fSGIS(enum target, float s);
    void MultiTexCoord1fvSGIS(enum target, const float *v);
    void MultiTexCoord1iSGIS(enum target, int s);
    void MultiTexCoord1ivSGIS(enum target, const int *v);
    void MultiTexCoord1sSGIS(enum target, short s);
```

```
    void MultiTexCoord1svSGIS(enum target, const short *v);
    void MultiTexCoord2dSGIS(enum target, double s, double t);
    void MultiTexCoord2dvSGIS(enum target, const double *v);
    void MultiTexCoord2fSGIS(enum target, float s, float t);
    void MultiTexCoord2fvSGIS(enum target, const float *v);
    void MultiTexCoord2iSGIS(enum target, int s, int t);
    void MultiTexCoord2ivSGIS(enum target, const int *v);
    void MultiTexCoord2sSGIS(enum target, short s, short t);
    void MultiTexCoord2svSGIS(enum target, const short *v);
    void MultiTexCoord3dSGIS(enum target, double s, double t, double r);
    void MultiTexCoord3dvSGIS(enum target, const double *v);
    void MultiTexCoord3fSGIS(enum target, float s, float t, float r);
    void MultiTexCoord3fvSGIS(enum target, const float *v);
    void MultiTexCoord3iSGIS(enum target, int s, int t, int r);
    void MultiTexCoord3ivSGIS(enum target, const int *v);
    void MultiTexCoord3sSGIS(enum target, short s, short t, short r);
    void MultiTexCoord3svSGIS(enum target, const short *v);
    void MultiTexCoord4dSGIS(enum target, double s, double t, double r, double q);
    void MultiTexCoord4dvSGIS(enum target, const double *v);
    void MultiTexCoord4fSGIS(enum target, float s, float t, float r, float q);
    void MultiTexCoord4fvSGIS(enum target, const float *v);
    void MultiTexCoord4iSGIS(enum target, int s, int t, int r, int q);
    void MultiTexCoord4ivSGIS(enum target, const int *v);
    void MultiTexCoord4sSGIS(enum target, short s, short t, short r, short q);
    void MultiTexCoord4svSGIS(enum target, const short *v);
    void InterleavedTextureCoordSetsSGIS(int factor);
    void SelectTextureSGIS(enum target);
    void SelectTextureCoordSetSGIS(enum target);
    void SelectTextureTransformSGIS(enum target);
```

New Tokens

Accepted by the <pname> parameters of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
SELECTED_TEXTURE_SGIS            0x83C0
SELECTED_TEXTURE_COORD_SET_SGIS  0x83C1
SELECTED_TEXTURE_TRANSFORM_SGIS  0x83C2
MAX_TEXTURES_SGIS                0x83C3
MAX_TEXTURE_COORD_SETS_SGIS              0x83C4
```

Accepted by the <pname> parameter of TexEnvi, TexEnvf, TexEnviv, TexEnvfv, GetTexEnviv, and GetTexEnvfv:

```
TEXTURE_ENV_COORD_SET_SGIS               0x83C5
```

Accepted by the <target> parameter of SelectTextureSGIS, SelectTextureTransformSGIS, SelectTextureCoordSetSGIS, MultiTexCoord<n>{T}[v]SGIS, and the <param> of TexParameteri and TexParameterf, and the <params> parameter of TexParameteriv, and TexParameterfv:

```
TEXTURE0_SGIS                    0x83C6
TEXTURE1_SGIS                    0x83C7
TEXTURE2_SGIS                    0x83C8
TEXTURE3_SGIS                    0x83C9
<reserve enums for 32>
```

Additions to Chapter 2 of the 1.1 Specification (OpenGL Operation)

Section 2.6 Begin/End Paradigm

<amend paragraph 2 & 3>

Each vertex is specified with two, three, or four coordinates. In addition, a current normal, current texture coordinate set, and current color may be used in processing each vertex. Normals are used by the GL in lighting calculations; the current normal is a three-dimensional vector that may be set by sending three coordinates that specify it. Texture coordinates determine how a texture image is mapped onto a primitive. Multiple sets of texture coordinates may be used to specify how multiple texture images are mapped onto a primitive. The number of texture coordinate sets supported is implementation dependent but must be at least one.

A color is associated with each vertex as it is specified. This associated color is either the current color or a color produced by lighting depending on whether or not lighting is enabled. Texture coordinates are similarly associated with each vertex. Multiple sets of texture coordinates may be associated with a vertex. Figure 2.2 summarizes the association of auxiliary data with a transformed vertex to produce a processed vertex.

<amend figure 2.2 to include multiple texcoord processing blocks
(current texcoords, texgen, texture matrix)>

<amend paragraph 6>
Before a color has been assigned to a vertex, the state required by a vertex is the vertex's coordinates, the current normal, and the current texture coordinate sets. Once color has been assigned, however, the current normal is no longer needed. Because color assignment is done vertex-by-vertex, a processed vertex comprises the vertex's coordinates, its assigned color, and its texture coordinate sets.

Section 2.7 Vertex Specifications <texture coordinates>
<amend paragraph 2>

Current values are used in associating auxiliary data with a vertex as described in section 2.6. A current value may be changed at any time by issuing an appropriate command. The commands

```
void TexCoord{1234}{sifd}SGIS(T coords);
void TexCoord{1234}{sifd}vSGIS(T coords);
```

specify the current homogeneous texture coordinates, named s,t,r, and q. The TexCoord1 family of commands set the s coordinate to the provided single argument while setting t and r to 0 and q to 1. Similarly, TexCoord2 sets s and t to the specified values, r to 0, and q to 1; TexCoord3 sets s, t, and r, with q set to 1, and TexCoord4 sets all four texture coordinates.

Implementations may support more than 1 set of texture coordinates. The MultiTexCoord family of commands takes the coordinate set to be modified as the <target> parameter. The <target> parameter is one of TEXTURE0_SGIS through TEXTURE3_SGIS. If a <target> parameter greater than the number of supported coordinate sets is specified, the command has no effect. The command

```
void SelectTextureCoordSetSGIS(enum target);
```

is used to change the texture coordinate set modified by the TexCoord* family of commands. <target> is one of TEXTURE0_SGIS through TEXTURE3_SGIS corresponding to the texture coordinate set to be modified by the TexCoord commands. The current coordinate set selection is part of client state rather than server state.

Section 2.8 Vertex Arrays
<amend paragraph 1>
The vertex specification commands in section 2.7 accept data in almost any format, but their use requires many command executions to specify even simple geometry. Vertex data may also be placed in arrays that are stored in the client's address space. Blocks of data in these arrays may be used to specify multiple geometric primitives through the execution of a single GL command. The client may specify 6 or more arrays at once: one each to store vertex coordinates, edge flags, colors, color indices, normals and one or more texture coordinate sets. The commands

```
void EdgeFlagPointer(sizei stride, void *pointer);
void VertexPointer(int size, enum type, sizei stride, void *pointer);
void ColorPointer(int size, enum type, sizei stride, void *pointer);
void IndexPointer(enum type, sizei stride, void *pointer);
void NormalPointer(enum type, sizei stride, void *pointer);
void TexCoordPointer(int size, enum type, sizei stride, void *pointer);
```

...

```
<insert this paragraph> between paragraph 2 & 3>

In implementations which support more than one set of texture coordinates,
the command SelectTextureCoordSetSGIS is used to select the vertex array
parameters to be modified by the TexCoordPointer command and the array
affected by client state enable and disable commands with the
TEXTURE_COORD_ARRAY parameter.

<modify the section on interleaved arrays as follows>

   The commands

   void InterleavedArrays(enum format, sizei stride,
       void *pointer) ;

   void InterleavedTextureCoordSetsSGIS(int factor) ;

efficiently initializes the six arrays and their enables to one of 14
configurations. <format> must be one 14 symbolic constants: V2F, V3F,
C4UB_V2F, C4UB_V3F, C3F_V3F, N3F_V3F, C4F_N3F_V3F, T2F_V3F, T4F_V4F,
T2F_C4UB_V3F, T2F_C3F_V3F, T2F_N3F_V3F, T2F_C4F_N3F_V3F, T4F_C4F_N3F_V4F.
<factor> is an integer between 1 and SELECTED_TEXTURE_COORD_SET_SGIS
and specifies how many texture coordinate sets are enabled as part
of the InterleavedArrays command.

   The effect of

       InterleavedArrays(format, stride, pointer);
       InterleavedTextureCoordSetsSGIS(factor);

is the same as the effect of the command sequence

      <copy command sequence from 1.1 spec, but change the part dealing
       with texture coords to>

       GetIntegerv(SELECTED_TEXTURE_COORD_SET_SGIS, &x);
       if (<et>) {
           for(i = 0; i < factor; i++) {
               SelectTextureCoordSetSGIS(TEXTURE0_SGIS+i);
               EnableClientState(TEXTURE_COORD_ARRAY);
               TexCoordPointer(st, FLOAT, str, <pointer>+i*pc);
           }
           for(i = factor; i < MAX_TEXTURE_COORD_SETS_SGIS; i++) {
               SelectTextureCoordSetSGIS(TEXTURE0_SGIS+i);
               DisableClientState(TEXTURE_COORD_ARRAY);
           }
       } else {
           for(i = 0; i < MAX_TEXTURE_COORD_SETS_SGIS; i++) {
               SelectTextureCoordSetSGIS(TEXTURE0_SGIS+i);
               DisableClientState(TEXTURE_COORD_ARRAY);
           }
       }
       SelectTextureCoordSetSGIS(x);
       pc *= factor;


If the number of supported is texture coordinate sets,
MAX_TEXTURE_COORD_SETS_SGIS, is <k>, then the client state require to
implement vertex arrays consists of five plus <k> boolean values, five
plus <k> integer stride values, four plus <k> constants representing array
types, and three plus <k> integers representing values per element.  In the
initial state, the boolean values are each disabled, the memory pointers
are each null, the strides are each zero, the array types are each FLOAT,
and the integers representing values per element are each four.

Section 2.10.2 Matrices
<amend paragraph 8 texture matrix>
There is another 4x4 matrix that is applied to texture coordinates.
This matrix is applied as

           | m1 m5 m9  m13 | |s|
```

```
          | m2 m6 m10 m14 | |t| ,
          | m3 m7 m11 m15 | |q|
          | m4 m8 m12 m16 | |r|
```

where the left matrix is the current texture matrix. The Matrix is applied to the coordinates resulting from texture coordinate generation which (which may simply be the current texture coordinates), and the resulting transformed coordinates become the texture coordinates associated with a vertex. Setting the matrix mode to TEXTURE causes the already described matrix operations to apply to the texture matrix stack.

For implementations which support more than one set of texture coordinates, there is a corresponding texture matrix for each coordinate set. Each stack has the same depth. The texture matrix which is affected by the matrix operations is set using the SelectTextureTransformSGIS command.

There is a stack of matrices for each of the matrix modes. For MODELVIEW mode, the stack depth is at least 32 (that is, there is a stack of at least 32 model-view matrices). For other modes, the depth is at least 2. Texture matrix stacks for all texture coordinate sets have the same depth.

```
   void PushMatrix( void );
```

pushes the stack down by one, duplicating the current matrix in both the top of the stack and the entry below it.

```
   void PopMatrix( void );
```

pops the top entry off of the stack, replacing the current matrix with the matrix that was the second entry in the stack. The pushing or popping takes place on the stack corresponding to the current matrix mode. Popping a matrix off a stack with only one entry generates the error STACK_UNDERFLOW; pushing a matrix onto a full stack generates STACK_OVERFLOW.

When the current matrix mode is TEXTURE, the texture matrix stack corresponding to the currently selected textured is pushed or popped.

The state required to implement transformations consists of a three-value integer indicating the current matrix mode, a stack of at least two 4x4 matrices for PROJECTION and one stack of at least two 4x4 matrices for each set of texture coordinates, TEXTURE, as well as associated stack pointers, and a stack of at least 32 4x4 matrices with an associated stack pointer for MODELVIEW. Initially, there is only one matrix on each stack and all matrices are set to the identity. The initial matrix mode is MODELVIEW.

Section 2.10.4 Generating texture coordinates

<amend paragraph 4>

The state required for texture coordinate generation for each set of texture coordinates supported by the implementation comprises a three-valued integer for each coordinate indicating coordinate generation mode, and a bit for each coordinate to indicate whether texture coordinate generation is enabled or disabled. In addition, four coefficients are required for the four coordinates for each of EYE_LINEAR and OBJECT_LINEAR. The initial state has the texture generation function disabled for all texture coordinates. The initial values of p_i for s except p_1 which is one; for t all the p_i are zero except p_2, which is 1. The values of p_i for r and q are all zero. These values of p_i apply for both the EYE_LINEAR and OBJECT_LINEAR versions. Initially all texture generation modes are EYE_LINEAR.

Section 2.12 Current Raster Position

<amend paragraph 2>

The current raster position consists of three window coordinates x_w, y_w, and z_w, a clip coordinate w_c value, and eye coordinate distance, a valid bit, and associated data consisting op a color and texture coordinate sets. It is set using one of the RasterPos commands:
...

<amend paragraph 5>

The current raster position requires five single-precision floating-point values for its x_w, y_w, and z_w window coordinates, its w_c clip coordinate, and its eye coordinate distance, a single valid bit, a color (RGBA and color index), and texture coordinates for each set of texture coordinates supported by the implementation. In the initial state, the coordinates and texture coordinates and both (0,0,0,1), the eye coordinate distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1) and the associated color index is 1. In RGBA mode, the associated color index always has its initial value; in color index mode, the RGBA color always maintains its initial value.

Additions to Chapter 3 of the 1.1 Specification (Rasterization)

Section 3.8 Texturing

<amend paragraphs 1 & 2>

Texturing maps a portion of one or more specified images onto each primitive for which texturing is enabled. This mapping is accomplished by using the color of an image at the location indicated by a fragment's (s,t,r) coordinates to modify the fragment's RGBA color (r is currently ignored). An implementation may support texturing using more than one image at a time. In this case the fragment carries multiple sets of texture coordinates (s,t,r) which are used to index separate images to produce color values which are collectively used to modify the fragment's RGBA color. Texturing is specified only for RGBA mode; its use in color index mode is undefined. The following subsections (up to and including Section 3.8.5) specify the GL operation with a single texture and Section 3.8.6 specifies the details of how multiple textures interact.

The GL provides a means to specify the details of how texturing of a primitive is effected. These details include specifications of the image to be texture mapped, the means by which the image is filtered when applied to the primitive, and the function that determines what RGBA value is produced given a fragment color and an image value.

Section 3.8.4 Texture Objects

<add this paragraph to the end of the section>

The texture object name space is shared amongst all textures in multiple texture implementations. A texture object may be bound to more than one texture target simultaneously, though they must all be of the same type (e.g., TEXTURE_1D, TEXTURE_2D). After a texture object is bound, any GL operations on that target also affect any other target to which the same texture object is bound.

Section 3.8.5 Texture Environments and Texture Functions

<amend the second half of paragraph 1>

The possible environment parameters are TEXTURE_ENV_MODE, TEXTURE_ENV_COLOR, and TEXTURE_ENV_COORD_SET_SGIS. TEXTURE_ENV_MODE may be set to one of REPLACE, MODULATE, DECAL, or BLEND; TEXTURE_ENV_COLOR is set to an RGBA color by providing four single-precision floating-point values in the range [0,1](values outside this range are clamped to it). If integers are provided for TEXTURE_ENV_COLOR, then they are converted to floating-point as specified in Table 2.6 for signed integers. TEXTURE_ENV_COORD_SET_SGIS may be set to one of TEXTURE0_SGIS .. TEXTURE<n>_SGIS where <n> is the one less than the number of supported texture coordinate sets.

<insert before paragraph 3>

The value of TEXTURE_ENV_COORD_SET_SGIS specifies which set of fragment texture coordinates are used to determine the texture value used in the texture function. The same set of texture coordinates may be simultaneously used by multiple textures.

<replace paragraph 3>

The state required for the current texture environment consists of the four-valued integer indicating the texture function, four floating-point TEXTURE_ENV_COLOR values, and one MAX_TEXTURE_COORD_SETS_SGIS-valued integer indicating the texture coordinate set binding. In the initial state, the texture function is given by MODULATE, TEXTURE_ENV_COLOR is

(0,0,0,0), and texture coordinate set is TEXTURE0_SGIS.

Section 3.8.6 Texture Application <replace with this>

Texturing is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constant TEXTURE_1D or TEXTURE_2D to enable the one-dimensional or two-dimensional texture, respectively. If both the one- and two-dimensional textures are enabled, the two-dimensional texture is used. If all texturing is disabled, a rasterized fragment is passed unaltered to the next stage of the GL (although its texture coordinates may be discarded). Otherwise, a texture value is found according to the parameter values of the currently bound texture image of the appropriate dimensionality using the rules given in sections 3.8.1 and 3.8.2. This texture value is used along with the incoming fragment in computing the texture function indicated by the currently bound texture environment. The result of this function replaces the incoming fragment's R, G, B, and A values. These are the color values passed to subsequent operations. Other data associated with the incoming fragment remain unchanged, except that the texture coordinates may be discarded.

When multiple textures are supported, additional textures are each paired with an environment function. The second texture function is computed using the texture value from the second texture, the fragment resulting from the first texture function computation and the environment function currently bound to the second texture. If there is a third texture, the fragment resulting from the second texture function is combined with the third texture value using the environment function bound to the third texture and so on. Texturing is enabled and disabled individually for each texture. If texturing is disabled for one of the textures, then the fragment result from the previous stage is passed unaltered to the next stage.

Additions to Chapter 4 of the 1.1 Specification (Per-Fragment Operations and the Framebuffer)

Additions to Chapter 5 of the 1.1 Specification (Special Functions)

Section 5.1 Evaluators

<amend second part of paragraph 2 to indicate that the evaluator map modified is affected by SELECTED_TEXTURE_TRANSFORM_SGIS when the the type parameter is one of the texture coordinates.>

<amend paragraph 7>

The evaluation of a defined map is enabled or disabled with Enable and Disable using the constant corresponding to the map as described above. In implementations which support multiple texture coordinates the affected texture evaluator map is further qualified by the value of SELECTED_TEXTURE_TRANSFORM_SGIS. The error INVALID_VALUE results if either ustride or vstride is less than k, or if u1 is equal to u2, or if v1 is equal to v2.

Section 5.3 Feedback

<amend bottom of paragraph 2>

The texture coordinates and colors returned are these resulting from the clipping operations described in (section 2.13.8). Only one set of texture coordinates is returned even for implementations which support multiple texture coordinates. The texture coordinate set returned is the one corresponding to the value of SELECTED_TEXTURE_TRANSFORM_SGIS.

Additions to Chapter 6 of the 1.1 Specification (State and State Requests)

<add this paragraph after paragraph 14 regarding multi-valued state variables>

When multiple textures are supported, most texture state variables are further qualified by the value of SELECTED_TEXTURE_TRANSFORM_SGIS or SELECTED_TEXTURE_SGIS to determine which server texture state vector is queried. Client texture state variables such as texture coordinate array pointers are qualified with SELECTED_TEXTURE_COORD_SET_SGIS. Tables 6.5 through 6.22 indicate those state variables which are

```
    qualified by SELECTED_TEXTURE_TRANSFORM_SGIS, SELECTED_TEXTURE_SGIS or
    SELECTED_TEXTURE_COORD_SET_SGIS during state queries.

    <add this paragraph after paragraph 16 regarding the TEXTURE_BIT>

    When multiple textures are supported, operations on groups containing
    replicated texture state push or pop all versions of texture state
    within that group.  When server state for a group is pushed all state
    in the group corresponding to TEXTURE0_SGIS is pushed first, followed by
    state corresponding to TEXTURE1_SGIS, and so on up to and including the
    state corresponding to TEXTURE<n>_SGIS where <n> is the value of
    max{MAX_TEXTURES_SGIS, MAX_TEXTURE_COORD_SETS_SGIS}.  If state does
    not exist for an attribute (this can occur when MAX_TEXTURES_SGIS is
    not equal to MAX_TEXTURE_COORD_SETS_SGIS) then it is ignored.
    When server state for a group is popped the replicated texture state is
    restored in the opposite order that it was pushed, starting with state
    corresponding to TEXTURE<n>_SGIS and ending with TEXTURE0_SGIS.
    Identical rules are observed for client texture state push and pop
    operations.

    <rename vertex_array attribute group to vertex>

Additions to the GLX Specification
    None

GLX Protocol
    TBD

Dependencies on EXT_texture3D
    If EXT_texture3D is not supported than the functionality and state
    associated with EXT_texture3D does not exist and is therefore not extended.

Dependencies on SGIS_texture4D
    If SGIS_texture4D is not supported than the functionality and state
    associated with SGIS_texture4D does not exist and is therefore not extended.

Dependencies on SGIS_texture_border_clamp
    If SGIS_texture_border_clamp is not supported than the functionality and state
    associated with SGIS_texture_border_clamp does not exist and is therefore
    not extended.

Dependencies on SGI_texture_color_table
    If SGI_texture_color_table is not supported than the functionality and state
    associated with SGI_texture_color_table does not exist and is therefore not extended.

Dependencies on SGIS_texture_edge_clamp
    If SGIS_texture_edge_clamp is not supported than the functionality and state
    associated with SGIS_texture_edge_clamp does not exist and is therefore not extended.

Dependencies on SGIX_texture_add_env
    If SGIX_texture_add_env is not supported than the functionality and state
    associated with SGIX_texture_add_env does not exist and is therefore not extended.

Dependencies on SGIS_texture_filter4
    If SGIS_texture_filter4 is not supported than the functionality and state
    associated with SGIS_texture_filter4 does not exist and is therefore not extended.

Dependencies on SGIS_texture_lod
    If SGIS_texture_lod is not supported than the functionality and state
    associated with SGIS_texture_lod does not exist and is therefore not extended.

Dependencies on SGIX_texture_lod_bias
    If SGIX_texture_lod_bias is not supported than the functionality and state
    associated with SGIX_texture_lod_bias does not exist and is therefore not extended.

Dependencies on SGIX_texture_scale_bias
    If SGIX_texture_scale_bias is not supported than the functionality and state
    associated with SGIX_texture_scale_bias does not exist and is therefore not extended.

Dependencies on SGIS_texture_select
    If SGIS_texture_select is not supported than the functionality and state
```

associated with SGIS_texture_select does not exist and is therefore not extended.

Dependencies on SGIS_detail_texture

If SGIS_detail_texture is not supported than the functionality and state associated with SGIS_detail_texture does not exist and is therefore not extended.

Dependencies on SGIS_sharpen_texture

If SGIS_sharpen_texture is not supported than the functionality and state associated with SGIS_sharpen_texture does not exist and is therefore not extended.

Dependencies on SGIX_shadow

If SGIX_shadow is not supported than the functionality and state associated with SGIX_shadow does not exist and is therefore not extended.

Dependencies on SGIX_shadow_ambient

If SGIX_shadow_ambient is not supported than the functionality and state associated with SGIX_shadow_ambient does not exist and is therefore not extended.

Dependencies on SGIX_clipmap

If SGIX_clipmap is not supported than the functionality and state associated with SGIX_clipmap does not exist and is therefore not extended.

Dependencies on SGIS_point_line_texgen

If SGIS_point_line_texgen is not supported than the functionality and state associated with SGIS_point_line_texgen does not exist and is therefore not extended.

Errors

INVALID_ENUM is generated if SelectTextureSGIS, SelectTextureTransformSGIS, SelectTextureCoordSetSGIS, MultiTexCoord<n>{T}[v], or MultiTexCoordPointer parameter <target> is not TEXTURE0_SGIS .. TEXTURE3_SGIS.

INVALID_OPERATION is generated if SelectTextureCoordSetSGIS or SelectTextureTransformSGIS parameter <target> is one of TEXTURE0_SGIS .. TEXTURE3_SGIS and <target> is greater or equal than the number of available textures coordinate sets.

INVALID_VALUE is generated if InterleavedTextureCoordSetsSGIS parameter <factor> is not between 1 and MAX_TEXTURE_COORD_SETS_SGIS.

INVALID_OPERATION is generated if SelectTextureSGIS parameter <target> is one of TEXTURE0_SGIS .. TEXTURE3_SGIS and <target> is greater or equal than the number of available textures.

INVALID_ENUM is generated if TexEnv{T}[v] parameter <pname> is TEXTURE_ENV_COORD_SET_SGIS and the <param> parameter is not one of TEXTURE0_SGIS .. TEXTURE3_SGIS.

INVALID_OPERATION is generated if TexEnv{T}[v] parameter <pname> is TEXTURE_ENV_COORD_SET_SGIS and the <param> parameter is greater or equal than the number of available textures coordinate sets.

INVALID_OPERATION is generated if SelectTextureSGIS or SelectTextureTransformSGIS is executed between execution of Begin and the corresponding execution of End.

INVALID_OPERATION is generated if SelectTextureCoordSetSGIS or MultiTexCoordPointerSGIS is executed between execution of Begin and the corresponding execution of End, but some implementations may not generate the error. In such cases the result of executing these commands is undefined.

New State

Get Value	Get Command	Type	Initial Value	Attribute
SELECTED_TEXTURE_SGIS	GetIntegerv	Z4	TEXTURE0_SGIS	texture
SELECTED_TEXTURE_TRANSFORM_SGIS	GetIntegerv	Z4	TEXTURE0_SGIS	texture
SELECTED_TEXTURE_COORD_SET_SGIS	GetIntegerv	Z4	TEXTURE0_SGIS	vertex
TEXTURE_COORD_SET_INTERLEAVE_FACTOR_SGIS	GetIntegerv	Z4	1	vertex

Replicated State

Get Value	Get Command	Type	Initial Value	Attribute
x CURRENT_TEXTURE_COORDS	GetFloatv	1* x T	(0,0,0,1)	current
x CURRENT_RASTER_TEXTURE_COORDS	GetFloatv	1* x T	(0,0,0,1)	current
c TEXTURE_COORD_ARRAY	IsEnabled	1* x B	False	vertex-array
c TEXTURE_COORD_ARRAY_SIZE	GetIntegerv	1* x Z+	0	vertex-array
c TEXTURE_COORD_ARRAY_TYPE	GetIntegerv	1* x Z4	FLOAT	vertex-array
c TEXTURE_COORD_ARRAY_STRIDE	GetIntegerv	1* x Z+	0	vertex-array
c TEXTURE_COORD_ARRAY_POINTER	GetPointerv	1* x Y	0	vertex-array
x TEXTURE_MATRIX	GetFloatv	1* x 2* x M4	Identity	-
x TEXTURE_STACK_DEPTH	GetIntegerv	1* x Z+	1	-
TEXTURE_1D	IsEnabled	1* x B	False	texture/enable
TEXTURE_2D	IsEnabled	1* x B	False	texture/enable
TEXTURE_3D_EXT	IsEnabled	1* x B	False	texture/enable
TEXTURE_4D_SGIS	IsEnabled	1* x B	False	texture/enable
TEXTURE_BINDING_1D	GetIntegerv	1* x Z+	0	texture
TEXTURE_BINDING_2D	GetIntegerv	1* x Z+	0	texture
TEXTURE_BINDING_3D_EXT	GetIntegerv	1* x Z+	0	texture
TEXTURE_BINDING_4D_SGIS	GetIntegerv	1* x Z+	0	texture
TEXTURE	GetTexImage	1* x n x I	see sec 3.8	-
TEXTURE_WIDTH	GetTexLevelParameter	1* x n x Z+	0	-
TEXTURE_HEIGHT	GetTexLevelParameter	1* x n x Z+	0	-
+TEXTURE_DEPTH_EXT	GetTexLevelParameter	1* x n x Z+	0	-
TEXTURE_BORDER	GetTexLevelParameter	1* x n x Z+	0	-
TEXTURE_INTERNAL_FORMAT (TEXTURE_COMPONENTS)	GetTexLevelParameter	1* x n x Z+	0	-
TEXTURE_RED_SIZE	GetTexLevelParameter	1* x n x Z+	0	-
TEXTURE_GREEN_SIZE	GetTexLevelParameter	1* x n x Z+	0	-
TEXTURE_BLUE_SIZE	GetTexLevelParameter	1* x n x Z+	0	-
TEXTURE_ALPHA_SIZE	GetTexLevelParameter	1* x n x Z+	0	-
TEXTURE_LUMINANCE_SIZE	GetTexLevelParameter	1* x n x Z+	0	-
TEXTURE_INTENISTY_SIZE	GetTexLevelParameter	1* x n x Z+	0	-
TEXTURE_BORDER_COLOR	GetTexParameter	1* x 2+ x C	(0,0,0,0)	texture
TEXTURE_MIN_FILTER	GetTexParameter	1* x 2+ x Z6	sec 3.8	texture
TEXTURE_MAG_FILTER	GetTexParameter	1* x 2+ x Z2	sec 3.8	texture
TEXTURE_WRAP_S	GetTexParameter	1* x 2+ x Z2	REPEAT	texture
TEXTURE_WRAP_T	GetTexParameter	1* x 2+ x Z2	REPEAT	texture
+TEXTURE_WRAP_R_EXT	GetTexParameter	1* x 2+ x Z2	REPEAT	texture
+TEXTURE_WRAP_Q_SGIS	GetTexParameter	1* x 2+ x Z2	REPEAT	texture
TEXTURE_PRIORITY	GetTexParameterfv	1* x 2+ x R	[0,1] 1	texture
TEXTURE_RESIDENT	GetTexParameterfv	1* x 2+ x B	False	texture
+TEXTURE_MIN_LOD_SGIS	GetTexParameterfv	1* x n x R	-1000	texture
+TEXTURE_MAX_LOD_SGIS	GetTexParameterfv	1* x n x R	1000	texture
+TEXTURE_BASE_LEVEL_SGIS	GetTexParameteriv	1* x n x R	0	texture
+TEXTURE_MAX_LEVEL_SGIS	GetTexParameteriv	1* x n x R	1000	texture
+TEXTURE_LOD_BIAS_S_SGIX	GetTexParameterfv	1* x n x R	0	texture
+TEXTURE_LOD_BIAS_T_SGIX	GetTexParameterfv	1* x n x R	0	texture
+TEXTURE_LOD_BIAS_R_SGIX	GetTexParameterfv	1* x n x R	0	texture
+TEXTURE_FILTER4_FUNC_SGIS	GetTexFilterFuncSGIS	1* x 2 x Size x R	see text	texture
+DETAIL_TEXTURE_2D_BINDING_SGIS	GetIntegerv	1* x Z+	0	texture
+DETAIL_TEXTURE_LEVEL_SGIS	GetTexParameteriv	1* x n x Z-	-4	texture
+DETAIL_TEXTURE_MODE_SGIS	GetTexParameteriv	1* x n x Z2	ADD	texture
+DETAIL_TEXTURE_FUNC_POINTS_SGIS	GetTexParameteriv	1* x n x Z+	2	texture
+<DETAIL_TEXTURE_FUNC>	GetDetailTexFuncSGIS	1* x n x m x R	{0, 0}, {-4, 1}	texture
+SHARPEN_TEXTURE_FUNC_POINTS_SGIS	GetTexParameteriv	1* x n x Z+	2	texture
+<SHARPEN_TEXTURE_FUNC>	GetSharpenTexFuncSGIS	1* x n x m x R	{0, 0},{-4, 1}	texture
+TEXTURE_COMPARE_SGIX	GetTexParameter[if]v	1* x B	False	texture
+TEXTURE_COMPARE_OPERATOR_SGIX	GetTexParameter[if]v	1* x Z_2	TEXTURE_LEQUAL_R_SGIX	texture
+SHADOW_AMBIENT_SGIX	GetTexParameter[if]v	1* x R[0,1]	0.0	texture
+TEXTURE_CLIPMAP_FRAME_SGIX	GetTexParameterf	1* x Z+	0	texture
+TEXTURE_CLIPMAP_CENTER_SGIX	GetTexParameterfv	1* x 2 x Z+	0,0	texture
+TEXTURE_CLIPMAP_OFFSET_SGIX	GetTexParameterfv	1* x 2 x Z+	0,0	texture
+TEXTURE_CLIPMAP_VIRTUAL_DEPTH_SGIX	GetTexParameterfv	1* x 3 x Z+	0,0,0	texture
+DUAL_TEXTURE_SELECT_SGIS	GetTexParameter	1* x n x 3 x Z2	0	texture
+QUAD_TEXTURE_SELECT_SGIS	GetTexParameter	1* x n x 3 x Z4	0	texture
+POST_TEXTURE_FILTER_BIAS_SGIX	GetTexParameterfv	1* x n x 4 x R	(0,0,0,0)	texture
+POST_TEXTURE_FILTER_SCALE_SGIX	GetTexParameterfv	1* x n x 4 x R	(1,1,1,1)	texture
TEXTURE_COLOR_TABLE_SGI	IsEnabled	B	False	texture/enable
+COLOR_TABLE	GetColorTableSGI	4 x I	empty	-
+COLOR_TABLE_FORMAT_SGI	GetColorTableParameterivSGI	2 x 4 x Z38	RGBA	-

+COLOR_TABLE_WIDTH_SGI	GetColorTableParameterivSGI	2 x 4 x Z+	0	-
+COLOR_TABLE_RED_SIZE_SGI	GetColorTableParameterivSGI	2 x 4 x Z+	0	-
+COLOR_TABLE_GREEN_SIZE_SGI	GetColorTableParameterivSGI	2 x 4 x Z+	0	-
+COLOR_TABLE_BLUE_SIZE_SGI	GetColorTableParameterivSGI	2 x 4 x Z+	0	-
+COLOR_TABLE_ALPHA_SIZE_SGI	GetColorTableParameterivSGI	2 x 4 x Z+	0	-
+COLOR_TABLE_LUMINANCE_SIZE_SGI	GetColorTableParameterivSGI	2 x 4 x Z+	0	-
+COLOR_TABLE_INTENSITY_SIZE_SGI	GetColorTableParameterivSGI	2 x 4 x Z+	0	-
+COLOR_TABLE_SCALE_SGI	GetColorTableParameterfvSGI	4 x R4	(1,1,1,1)	pixel
+COLOR_TABLE_BIAS_SGI	GetColorTableParameterfvSGI	4 x R4	(0,0,0,0)	pixel
TEXTURE_ENV_MODE	GetTexEnviv	1* x Z4	MODULATE	texture
TEXTURE_ENV_COLOR	GetTexEnviv	1* x C	(0,0,0,0)	texture
TEXTURE_ENV_COORD_SET_SGIS	GetTexEnviv	1* x Z4	see sec 3.8	texture
x TEXTURE_GEN_x	IsEnabled	1* x 4 x B	False	texture/enable
x EYE_PLANE	GetTexGenfv	1* x 4 x R4	see sec 2.10.4	texture
x OBJECT_PLANE	GetTexGenfv	1* x 4 x R4	see sec 2.10.4	texture
x TEXTURE_GEN_MODE	GetTexGeniv	1* x 4 x Z3	EYE_LINEAR	texture
+TEXTURE_ENV_BIAS_SGIX	GetFloatv	1* x C	(0,0,0,0)	texture
x+EYE_POINT_SGIS	GetTexGeniv	1* x 4 x R	(0,0,0,1)	texture
x+OBJECT_POINT_SGIS	GetTexGeniv	1* x 4 x R	(0,0,0,1)	texture
x+EYE_LINE_SGIS	GetTexGeniv	1* x 7 x R	(0,0,0,1,0,0,1)	texture
x+OBJECT_LINE_SGIS	GetTexGeniv	1* x 7 x R	(0,0,0,1,0,0,1)	texture
x ORDER	GetMapiv	(k+9) x Z8*	1	-
x ORDER	GetMapiv	(k+9) x 2 x Z8*	1,1	-
x COEFF	GetMapfv	(k+9) x 8* x Rn	see sec 5.1	-
x COEFF	GetMapfv	(k+9) x 8* x 8* x Rn	see sec 5.1	-
x DOMAIN	GetMapfv	(k+9) x 2 x R	see sec 5.1	-
x DOMAIN	GetMapfv	(k+9) x 4 x R	see sec 5.1	-
x MAP1_x	IsEnabled	(k+9) x B	False	-
x MAP2_x	IsEnabled	(k+9) x B	False	-

+ = state defined in another extension.
x = state qualified by SELECTED_TEXTURE_TRANSFORM_SGIS
c = state qualified by SELECTED_TEXTURE_COORD_SET_SGIS

New Implementation Dependent State

Get Value	Get Command	Type	Minimum Value
MAX_TEXTURES_SGIS	GetIntegerv	Z+	1
MAX_TEXTURE_COORD_SETS_SGIS	GetIntegerv	Z+	1

E. NVIDIA's Multitexture Combiners Specification

XXX – Preliminary

Name

GL_NVIDIA_multitexture_combiners

Version

Date: 1998/04/16 9:00 am
$Id: //sw/docs/OpenGL/specs/GL_NVIDIA_multitexture_combiners.txt#3$

Number

???

Dependencies

GL_SGIS_multitexture

Overview

This extension provides a much more flexible mechanism for specifying how multiple textures are combined with previous fragments. It allows for multiple combiners of the form:

(Arg0 * Arg1) Op (Arg2 * Arg3)

where Arg<n> can come from a variety of inputs, such as the diffuse color, or any of the currently active textures. The result of the previous combiner can be used as an input to the next combiner in the chain, thus allowing for very flexible specification of how textures, diffuse color, blend factors, and other inputs can be factored into the resultant fragment color.

The alpha channel can receive input from separate sources, and can use a separate <Op>, thereby giving even more flexibility to the application.

Issues

None

New Procedures and Functions

```
GLvoid
glMTexCombinerColorArgNVIDIA(GLenum combiner,
                             GLenum arg,
                             GLenum input,
                             GLboolean complementInput,
                             GLboolean useAlphaComponent);
```

glMTexCombinerColorArg() allows any one of the 4 args (Arg0 ... Arg3) to be set to an input to the combiner. Valid inputs are enumerated below. The input may be pulled from the alpha channel by setting useAlphaComponent to GL_TRUE, such as in the case of a 1-chanel texture used as an intensity texture; otherwise the normal red, green, or blue component is used. The mathematical complement of the input arg may be used by setting complementInput to GL_TRUE.

```
GLvoid
glMTexCombinerAlphaArgNVIDIA(Glenum combiner,
                             GLenum arg,
                             GLenum input,
                             GLboolean complementInput);
```

glMTexCombinerAlphaArg() allows any one of the 4 args (Arg0 ... Arg3) to be set to an input to the combiner. Valid inputs are enumerated below. The mathematical complement of the input arg may be used by setting complementInput to GL_TRUE.

```
GLvoid
glMTexCombinerColorOpNVIDIA(Glenum combiner,
                            GLenum operation);
```

glMTexCombinerColorOp() can be used to set the operation used to combine (Arg0 * Arg1) with (Arg2 * Arg3) for the red, green, and

```
    blue components of the fragment in the specified combiner.  It can be any
    of the GL_MTEX_COMBINER_ALPHA_OP_* enumerants below.

    GLvoid
    glMTexCombinerAlphaOpNVIDIA(GLenum combiner,
                                GLenum operation);

    glMTexCombinerAlphaOp() can be used to set the operation used to
    combine ( Arg0 * Arg1 ) with ( Arg2 * Arg3 ) for the alpha component
    of the fragment in the specified combiner.  It can be any of the
    GL_MTEX_COMBINER_ALPHA_OP_* enumerants below.

    GLvoid
    glMTexCombinerBlendFactorNVIDIA(GLenum combiner,
                                    GLfloat red,
                                    GLfloat green,
                                    GLfloat blue,
                                    GLfloat alpha);

    glMTexCombinerBlendFactor() allows the application to specify a
    color that can be optionally used as one of the inputs to the
    specified combiner.  This allows for texture applications such as
    the GL_BLEND TexEnv to be implemented.

New Tokens
    Accepted by glEnable() and glDisable():
        GL_NVIDIA_COMBINERS_ENABLE          ??? 0x900F

    Accepted by the <combiner> parameter of all of the above-specified
    functions:
        GL_MTEX_COMBINER_0                  ??? 0x9010
        GL_MTEX_COMBINER_1                  ??? 0x9011
        GL_MTEX_COMBINER_2                  ??? 0x9012
        GL_MTEX_COMBINER_3                  ??? 0x9013
        <reserve enums for 32 combiners>

    Accepted by the <arg> parameter of glMTexCombinerColorArg() and
    glMTexCombinerAlphaArg():
        GL_MTEX_COMBINER_ARG0               ??? 0x9030
        GL_MTEX_COMBINER_ARG1               ??? 0x9031
        GL_MTEX_COMBINER_ARG2               ??? 0x9032
        GL_MTEX_COMBINER_ARG3               ??? 0x9033

    Accepted by the <input> parameter of glMTexCombinerColorArg() and
    glMTexCombinerAlphaArg():
        GL_MTEX_COMBINER_INPUT_ZERO         ??? 0x9040
        GL_MTEX_COMBINER_INPUT_FACTOR       ??? 0x9041  /* Blend factor */
        GL_MTEX_COMBINER_INPUT_DIFFUSE      ??? 0x9042  /* Diffuse color */
        GL_MTEX_COMBINER_INPUT_PREVCOMBINER ??? 0x9043  /* Output of
                                                           previous combiner */
        GL_MTEX_COMBINER_INPUT_TEXTURE0     ??? 0x9044  /* Multitexture Tex0 */
        GL_MTEX_COMBINER_INPUT_TEXTURE1     ??? 0x9045  /* Multitexture Tex1 */
            <reserve enums for 32 textures>
        GL_MTEX_COMBINER_INPUT_TEXTURELOD   ??? 0x9064  /* Fractional component
                                                           of TextureLOD
                                                           computation, allowing
                                                           for very flexible
                                                           mipmapping effects */

    Accepted by the <operation> parameter of glMTexCombinerColorOp() and
    glMTexCombinerAlphaOp():

GL_MTEX_COMBINER_OP_ADD             ??? 0x9070  /*    (A0*A1) +  (A2*A3) */
GL_MTEX_COMBINER_OP_ADDTIMES2       ??? 0x9071  /*  ( (A0*A1) +  (A2*A3) ) << 1 */
GL_MTEX_COMBINER_OP_ADDTIMES4       ??? 0x9072  /*  ( (A0*A1) +  (A2*A3) ) << 2 */
GL_MTEX_COMBNIER_OP_ADDSIGNED       ??? 0x9073  /*    (A0*A1) +  (A2*A3) - 128 */
GL_MTEX_COMBINER_OP_MUX             ??? 0x9074  /*    (A0*A1) or (A2*A3) */
GL_MTEX_COMBINER_OP_ADDCOMPLEMENT   ??? 0x9075  /* ~( (A0*A1) +  (A2*A3) ) */
GL_MTEX_COMBINER_OP_ADDSIGNEDTIMES2 ??? 0x9076  /*  ( (A0*A1) +  (A2*A3) - 128 ) << 1 */
```

F. Secondary Color Specification

XXX – Preliminary

Name
secondary_color

Name Strings
GL_EXT_secondary_color

Version
$Date: 1998/04/23 04:51:41 $ $Revision: 1.5 $

Number
145

Dependencies
Either EXT_separate_specular_color or OpenGL 1.2 is required, to specify the "Color Sum" stage and other handling of the secondary color. This is written against the 1.2 specification (available from www.opengl.org).

Overview
This extension allows specifying the RGB components of the secondary color used in the Color Sum stage, instead of using the default (0,0,0,0) color. It applies only in RGBA mode and when LIGHTING is disabled.

Issues

* Can we use the secondary alpha as an explicit fog weighting factor?

  ISVs prefer a separate interface (see GL_EXT_fog_coord). The current interface specifies only the RGB elements, leaving the option of a separate extension for SecondaryColor4() entry points open.

  There is an unpleasant asymmetry with Color3() - one assumes A = 1.0, the other assumes A = 0.0 - but this appears unavoidable given the 1.2 color sum specification language. Alternatively, the color sum language could be rewritten to not sum secondary A.

* What about multiple "color iterators" for use with aggrandized multitexture implementations?

  We may need this eventually, but the secondary color is well defined and a more generic interface doesn't seem justified now.

* Interleaved array formats?

  No. The multiplicative explosion of formats is too great.

* Do we want to be able to query the secondary color value? How does it interact with lighting?

  The secondary color is not part of the GL state in the separate_specular_color extension. It can't be queried or obtained via feedback.

  Since the secondary_color extension is just to serve during transition to fragment lighting, let's not go overboard - it needs to be in the GL state, but not much beyond that.

New Procedures and Functions

```
    void SecondaryColor3[bsifd ubusui]EXT(T components)
    void SecondaryColor3[bsifd ubusui]vEXT(T components)
    void SecondaryColorPointerEXT(int size, enum type, sizei stride,
                            void *pointer)
```

New Tokens

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
   COLOR_SUM_EXT                 ???
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
   CURRENT_SECONDARY_COLOR_EXT          ???
   SECONDARY_COLOR_ARRAY_SIZE_EXT       ???
   SECONDARY_COLOR_ARRAY_TYPE_EXT       ???
   SECONDARY_COLOR_ARRAY_STRIDE_EXT    ???
```

Accepted by the <pname> parameter of GetPointerv:

```
   SECONDARY_COLOR_ARRAY_POINTER_EXT   ???
```

Accepted by the <array> parameter of EnableClientState and DisableClientState:

```
   SECONDARY_COLOR_ARRAY_EXT      ???
```

Additions to Chapter 2 of the 1.2 Draft Specification (OpenGL Operation)

These changes describe a new current state type, the secondary color, and the commands to specify it:

- (2.6, p. 12) Second paragraph changed to:

  "Each vertex is specified with two, three, or four coordinates. In addition, a current normal, current texture coordinates, current color, and current secondary color may be used in processing each vertex."

  Third paragraph, second sentence changed to:

  "These associated colors are either based on the current color and current secondary color, or produced by lighting, depending on whether or not lighting is enabled."

- 2.6.3, p. 19) First paragraph changed to

  "The only GL commands that are allowed within any Begin/End pairs are the commands for specifying vertex coordinates, vertex colors, normal coordinates, and texture coordinates (Vertex, Color, SecondaryColorEXT, Index, Normal, TexCoord)..."

- (2.7, p. 20) Starting with the fourth paragraph, change to:

  "Finally, there are several ways to set the current color and secondary color. The GL stores a current single-valued color index as well as a current four-valued RGBA color and secondary color. Either the index or the color and secondary color are significant depending as the GL is in color index mode or RGBA mode. The mode selection is made when the GL is initialized.

The commands to set RGBA colors and secondary colors are:

```
    void Color[34][bsifd ubusui](T components)
    void Color[34][bsifd ubusui]v(T components)
    void SecondaryColor3[bsifd ubusui]EXT(T components)
    void SecondaryColor3[bsifd ubusui]vEXT(T components)
```

The color command has two major variants: Color3 and Color4. The four value versions set all four values. The three value versions set R, G, and B to the provided values; A is set to 1.0. (The conversion of integer color components (R, G, B, and A) to floating-point values is discussed in section 2.13.)

The secondary color command has only the three value versions. Secondary A is always set to 0.0.

Versions of the Color and SecondaryColorEXT commands that take floating-point values accept values nominally between 0.0 and 1.0...."

The last paragraph is changed to read:

"The state required to support vertex specification consists of four floating-point numbers to store the current texture coordinates s, t, r, and q, four floating-point values to store the current RGBA color, four floating-point values to store the current RGBA secondary color, and one floating-point value to store the current color index. There is no notion of a current vertex, so no state is devoted to vertex coordinates. The initial values of s, t, and r of the current texture coordinates are zero; the initial value of q is one. The initial current normal has coordinates (0,0,1). The initial RGBA color is (R,G,B,A) = (1,1,1,1). The initial RGBA secondary color is (R,G,B,A) = (0,0,0,0). The initial color index is 1."

- (2.8, p. 21) Added secondary color command for vertex arrays:

Change first paragraph to read:

"The vertex specification commands described in section 2.7 accept data in almost any format, but their use requires many command executions to specify even simple geometry. Vertex data may also be placed into arrays that are stored in the client's address space. Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single GL command. The client may specify up to seven arrays: one each to store edge flags, texture coordinates, colors, secondary colors, color indices, normals, and vertices. The commands"

Add to functions listed following first paragraph:

```
    void SecondaryColorPointerEXT(int size, enum type, sizei stride,
                                  void *pointer)
```

Add to table 2.4 (p. 22):

Command	Sizes	Types
SecondaryColorPointerEXT	3,4	byte,ubyte,short,ushort,int,uint, float,double

Starting with the second paragraph on p. 23, change to add SECONDARY_COLOR_ARRAY_EXT:

"An individual array is enabled or disabled by calling one of

```
    void EnableClientState(enum array)
    void DisableClientState(enum array)
```

with array set to EDGE_FLAG_ARRAY, TEXTURE_COORD_ARRAY, COLOR_ARRAY, SECONDARY_COLOR_ARRAY_EXT, INDEX_ARRAY, NORMAL_ARRAY, or VERTEX_ARRAY, for the edge flag, texture coordinate, color, secondary color, color index, normal, or vertex array, respectively.

The ith element of every enabled array is transferred to the GL by calling

```
    void ArrayElement(int i)
```

For each enabled array, it is as though the corresponding command from section 2.7 or section 2.6.2 were called with a pointer to element i. For the vertex array, the corresponding command is Vertex<size><type>v, where <size> is one of [2,3,4], and <type> is one of [s,i,f,d], corresponding to array types short, int, float, and double respectively. The corresponding commands for the edge flag, texture coordinate, color, secondary color, color index, and normal arrays are EdgeFlagv, TexCoord<size><type>v, Color<size><type>v, SecondaryColor3<type>vEXT, Index<type>v, and Normal<type>v, respectively..."

Change pseudocode on p. 27 to disable secondary color array for canned interleaved array formats. After the lines

```
    DisableClientState(EDGE_FLAG_ARRAY);
    DisableClientState(INDEX_ARRAY);
```

insert the line

```
    DisableClientState(SECONDARY_COLOR_ARRAY_EXT);
```

Substitute "seven" for every occurrence of "six" in the final paragraph on p. 27.

- (2.12, p. 41) Add secondary color to the current rasterpos state.

  Change the last paragraph to read

  "The current raster position requires five single-precision floating-point values for its x_w, y_w, and z_w window coordinates, its w_c clip coordinate, and its eye coordinate distance, a single valid bit, a color (RGBA color, RGBA secondary color, and color index), and texture coordinates for associated data. In the initial state, the coordinates and texture coordinates are both $(0,0,0,1)$, the eye coordinate distance is 0, the valid bit is set, the associated RGBA color is $(1,1,1,1)$, the associated RGBA secondary color is $(0,0,0,0)$, and the associated color index color is 1. In RGBA mode, the associated color index always has its initial value; in color index mode, the RGBA color and secondary color always maintain their initial values."

- (2.13, p. 43) Change second paragraph to acknowledge two colors when lighting is disabled:

  "Next, lighting, if enabled, produces either a color index or primary and secondary colors. If lighting is disabled, the current color index or current color (primary color) and current secondary

color are used in further processing. After lighting, RGBA colors are clamped..."

- (Figure 2.8, p. 42) Change to show primary and secondary RGBA colors in both lit and unlit paths.

- (2.13.1, p. 44) Change so that the second paragraph starts:

"Lighting may be in one of two states:

1. Lighting Off. In this state, the current color and current secondary color are assigned to the vertex primary color and vertex secondary color, respectively.

2. ..."

- (2.13.1, p. 48) Change the sentence following equation 2.5 (for spot_i) so that color sum is implicitly enabled when SEPARATE_SPECULAR_COLOR is set:

"All computations are carried out in eye coordinates. When c_es = SEPARATE_SPECULAR_COLOR, it is as if color sum (see section 3.9) were enabled, regardless of the value of COLOR_SUM_EXT."

- (3.9, p. 136) Change the first paragraph to read

"After texturing, a fragment has two RGBA colors: a primary color c_pri (which texturing, if enabled, may have modified) and a secondary color c_sec.

If color sum is enabled, the components of these two colors are summed to produce a single post-texturing RGBA color c (the A component of the secondary color is always 0). The components of c are then clamped to the range [0,1]. If color sum is disabled, then c_pri is assigned to the post texturing color. Color sum is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constant COLOR_SUM_EXT.

The state required is a single bit indicating whether color sum is enabled or disabled. In the initial state, color sum is disabled."

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None (we don't expect this extension to be used on GLX implementations; if it is, new GLX enumerants for the secondary color will be introduced).

GLX Protocol

None (as for the GLX Specification)

Errors

INVALID_VALUE is generated if SecondaryColorPointerEXT parameter <size> is not 3.

INVALID_ENUM is generated if SecondaryColorPointerEXT parameter <type> is not BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT, FLOAT, or DOUBLE.

INVALID_VALUE is generated if SecondaryColorPointerEXT parameter <stride> is negative.

New State

(table 6.5, p. 195)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_SECONDARY_COLOR_EXT	C	GetIntegerv, GetFloatv	(0,0,0,0)	Current secondary color	2.7	current

(table 6.6, p. 197)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
SECONDARY_COLOR_ARRAY_EXT	B	IsEnabled	False	Sec. color array enable	2.8	vertex-array
SECONDARY_COLOR_ARRAY_SIZE_EXT	Z+	GetIntegerv	3	Sec. colors per vertex	2.8	vertex-array
SECONDARY_COLOR_ARRAY_TYPE_EXT	Z8	GetIntegerv	FLOAT	Type of sec. color components	2.8	vertex-array
SECONDARY_COLOR_ARRAY_STRIDE_EXT	Z+	GetIntegerv	0	Stride between sec. colors	2.8	vertex-array
SECONDARY_COLOR_ARRAY_POINTER_EXT	Y	GetPointerv	0	Pointer to the sec. color array	2.8	vertex-array

(table 6.8, p. 198)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
COLOR_SUM_EXT	B	IsEnabled	False	True if color sum enabled	3.9	fog/enable

G. Fog Coordinate Specification

XXX - Not complete yet!!!

Name
fog_coord

Name Strings
GL_EXT_fog_coord

Version
$Date: 1998/04/09 22:09:14 $ $Revision: 1.2 $

Number
149

Dependencies
OpenGL 1.1 is required. fog_coord is written against the OpenGL 1.2 specification (available from www.opengl.org) to make the spec forward-looking, although no 1.2 features are required.

Overview
This extension allows specifying an explicit per-vertex fog coord to be used in fog computations, rather than using a fragment depth-based fog equation.

Issues

* Should the specified value be used directly as the fog weighting factor, or in place of the z input to the fog equations?

  As the z input, adding more flexibility at potential performance cost.

* Do we want vertex array entry points? Interleaved array formats?

  Yes for entry points, no for interleaved formats, following the argument for secondary_color.

* Which scalar types should FogCoord accept? The full range, or just the unsigned and float versions? At the moment it follows Index(), which takes unsigned byte, signed short, signed int, float, and double.

  Since we're now specifying a number which behaves like an eye-space distance, rather than a [0,1] quantity, integer types are less useful. However, restricting the commands to floating-point forms only introduces some nonorthogonality.

  Restrict to only float and double, for now.

* Interpolation of the fog coordinate may be perspective-correct or not. Should this be affected by PERSPECTIVE_CORRECTION_HINT, FOG_HINT, or another to-be-defined hint?

  PERSPECTIVE_CORRECTION_HINT; this is already defined to affect all interpolated parameters. Admittedly this is a loss of orthogonality.

* Should the current fog coordinate be queryable?

  Yes; GetFloatv(FOG_COORDINATE).

* Control the fog coordinate source via an Enable instead of a fog parameter?

  No. We might want to add more sources later.

* Should the fog coordinate be restricted to non-negative values?

  Perhaps. Eye-coordinate distance of fragments will be non-negative due to clipping. Specifying explicit negative coordinates may result

in very large computed f values, although they are defined to be clipped after computation.

* Use existing DEPTH enum instead of FRAGMENT_DEPTH? Change name of FRAGMENT_DEPTH to FOG_FRAGMENT_DEPTH?

Undecided.

New Procedures and Functions

```
void FogCoord[fd]EXT(T coord)
void FogCoord[fd]vEXT(T coord)
void FogCoordPointerEXT(enum type, sizei stride, void *pointer)
```

New Tokens

Accepted by the <param> parameter of Fogi and Fogf:

FOG_COORDINATE_SOURCE	???
FOG_COORDINATE	???
FRAGMENT_DEPTH	???

Accepted by the <array> parameter of EnableClientState and DisableClientState:

FOG_FACTOR_ARRAY_EXT	???

Additions to Chapter 2 of the 1.2 Draft Specification (OpenGL Operation)

These changes describe a new current state type, the fog coordinate, and the commands to specify it:

- (2.6, p. 12) Second paragraph changed to:

"Each vertex is specified with two, three, or four coordinates. In addition, a current normal, current texture coordinates, current color, and current fog coordinate may be used in processing each vertex."

- 2.6.3, p. 19) First paragraph changed to

"The only GL commands that are allowed within any Begin/End pairs are the commands for specifying vertex coordinates, vertex colors, normal coordinates, texture coordinates, and fog coordinates (Vertex, Color, Index, Normal, TexCoord, FogCoord)..."

- (2.7, p. 20) Insert the following paragraph following the third paragraph describing current normals:

" The current fog factor is set using
   void FogCoord[fd]EXT(T coord)
   void FogCoord[fd]vEXT(T coord)."

The last paragraph is changed to read:

"The state required to support vertex specification consists of four floating-point numbers to store the current texture coordinates s, t, r, and q, one floating-point value to store the current fog coordinate, four floating-point values to store the current RGBA color, and one floating-point value to store the current color index. There is no notion of a current vertex, so no state is devoted to vertex coordinates. The initial values of s, t, and r of the current texture coordinates are zero; the initial value of q is one. The initial fog coordinate is zero. The initial current normal has coordinates (0,0,1). The initial RGBA color is (R,G,B,A) = (1,1,1,1). The initial color index is 1."

- (2.8, p. 21) Added fog coordinate command for vertex arrays:

Change first paragraph to read:

"The vertex specification commands described in section 2.7 accept data in almost any format, but their use requires many command executions to specify even simple geometry. Vertex data may also be

```
    placed into arrays that are stored in the client's address space.
    Blocks of data in these arrays may then be used to specify multiple
    geometric primitives through the execution of a single GL command.
    The client may specify up to seven arrays: one each to store edge
    flags, texture coordinates, fog coordinates, colors, color indices,
    normals, and vertices. The commands"

Add to functions listed following first paragraph:

    void FogCoordPointerEXT(enum type, sizei stride, void *pointer)

Add to table 2.4 (p. 22):

    Command                  Sizes   Types
    -------                  -----   -----
    FogCoordPointerEXT       1       float,double

Starting with the second paragraph on p. 23, change to add
FOG_FACTOR_ARRAY_EXT:

    "An individual array is enabled or disabled by calling one of

        void EnableClientState(enum array)
        void DisableClientState(enum array)

    with array set to EDGE_FLAG_ARRAY, TEXTURE_COORD_ARRAY,
    FOG_FACTOR_ARRAY_EXT, COLOR_ARRAY, INDEX_ARRAY, NORMAL_ARRAY, or
    VERTEX_ARRAY, for the edge flag, texture coordinate, fog coordinate,
    color, color index, normal, or vertex array, respectively.

    The ith element of every enabled array is transferred to the GL by
    calling

        void ArrayElement(int i)

    For each enabled array, it is as though the corresponding command
    from section 2.7 or section 2.6.2 were called with a pointer to
    element i. For the vertex array, the corresponding command is
    Vertex<size><type>v, where <size> is one of [2,3,4], and <type> is
    one of [s,i,f,d], corresponding to array types short, int, float,
    and double respectively. The corresponding commands for the edge
    flag, texture coordinate, fog coordinate, color, secondary color,
    color index, and normal arrays are EdgeFlagv, TexCoord<size><type>v,
    FogCoord<type>v, Color<size><type>v, Index<type>v, and
    Normal<type>v, respectively..."

Change pseudocode on p. 27 to disable fog coordinate array for canned
interleaved array formats. After the lines

        DisableClientState(EDGE_FLAG_ARRAY);
        DisableClientState(INDEX_ARRAY);

    insert the line

        DisableClientState(FOG_FACTOR_ARRAY_EXT);

Substitute "seven" for every occurrence of "six" in the final paragraph
on p. 27.

- (2.12, p. 41) Add fog coordinate to the current rasterpos state.

Change the first sentence of the first paragraph to read

    "The state required for the current raster position consists of
    three window coordinates x_w, y_w, and z_w, a clip coordinate w_c
    value, an eye coordinate distance, a fog coordinate, a valid bit,
    and associated data consisting of a color and texture coordinates."

Change the last paragraph to read

    "The current raster position requires six single-precision
```

```
        floating-point values for its x_w, y_w, and z_w window coordinates,
        its w_c clip coordinate, its eye coordinate distance, and its fog
        coordinate, a single valid bit, a color (RGBA color and color
        index), and texture coordinates for associated data. In the initial
        state, the coordinates and texture coordinates are both (0,0,0,1),
        the fog coordinate is 0, the eye coordinate distance is 0, the valid
        bit is set, the associated RGBA color is (1,1,1,1), and the
        associated color index color is 1. In RGBA mode, the associated
        color index always has its initial value; in color index mode, the
        RGBA color always maintains its initial value."

  - (3.10, p. 139) Change the second and third paragraphs to read

        "This factor f may be computed according to one of three equations:"

            f = exp(-d*c)     (3.24)
            f = exp(-(d*c)^2) (3.25)
            f = (e-c)/(e-s)   (3.26)

        If the fog source (as defined below) is FRAGMENT_DEPTH, then c is
        the eye-coordinate distance from the eye, (0 0 0 1) in eye
        coordinates, to the fragment center. If the fog source if
        FOG_COORDINATE, then c is the interpolated value of the fog
        coordinate for this fragment. The equation and the fog source, along
        with either d or e and s, is specified with

            void Fog{if}(enum pname, T param);
            void Fog{if}v(enum pname, T params);

        If <pname> is FOG_MODE, then <param> must be, or <param> must point
        to an integer that is one of the symbolic constants EXP, EXP2, or
        LINEAR, in which case equation 3.24, 3.25, or 3.26,, respectively,
        is selected for the fog calculation (if, when 3.26 is selected, e =
        s, results are undefined). If <pname> is FOG_COORDINATE_SOURCE, then
        <param> is or <params> points to an integer that is one of the
        symbolic constants FRAGMENT_DEPTH or FOG_COORDINATE. If <pname> is
        FOG_DENSITY, FOG_START, or FOG_END, then <param> is or <params>
        points to a value that is d, s, or e, respectively. If d is
        specified less than zero, the error INVALID_VALUE results."

  - (3.10, p. 140) Change the last paragraph preceding section 3.11
    to read

        "The state required for fog consists of a three valued integer to
        select the fog equation, three floating-point values d, e, and s, an
        RGBA fog color and a fog color index, a two-valued integer to select
        the fog coordinate source, and a single bit to indicate whether or
        not fog is enabled. In the initial state, fog is disabled,
        FOG_COORDINATE_SOURCE is FRAGMENT_DEPTH, FOG_MODE is EXP, d = 1.0, e
        = 1.0, and s = 0.0; C_f = (0,0,0,0) and i_f=0."

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)
  None

Additions to the GLX Specification
  None (we don't expect this extension to be used on GLX implementations; if
    it is, new GLX enumerants for the fog coordinate will be introduced).

GLX Protocol
  None (as for the GLX Specification)

Errors
    INVALID_ENUM is generated if FogCoordPointerEXT parameter <type> is not
    FLOAT or DOUBLE.

    INVALID_VALUE is generated if FogCoordPointerEXT parameter <stride> is
    negative.

New State

Get Value             Type    Get Command    Initial Value  Description    Sec Attribute
```

```
---------             ----    -----------    -------------  -----------    --- ---------
(table 6.6, p. 196)
    For each of the 4 INDEX_ARRAY* table entries, add a corresponding
    FOG_FACTOR_ARRAY*_EXT entry with the same type, get command, initial
    value, section, and attribute, and with "fog coordinate" replacing "index"
    and "indice" in its description.

(table 6.8, p. 198)
    FOG_COORDINATE     R       GetIntegerv,   0              Current        3.10 fog
                                  GetFloatv                                fog coordinate
    FOG_COORDINATE_SOURCE Z2   GetIntegerv,   FRAGMENT_DEPTH Source of fog  3.10 fog
                                  GetFloatv                                coord for
                                                                           fog calculation
```