Chapter 6

Texturing Extensions

This chapter explains how to use the different OpenGL texturing extensions. You learn about:

The Texture Extension

The texture extension, EXT_texture, allows you to select an internal format for textures. When you create a texture using OpenGL 1.0, you can only define it as having 1, 2, 3, or 4 components. The texture extension provides several additional predefined formats. You can indicate the following on a per-texture basis:

- the number of components to store in texture memory
- the internal image format (GL_RGB4_EXT and so on, see Table 6-1)

The extension also defines a robust method for applications to determine what combinations of texture dimensions and resolutions are supported. It also introduces a new texture environment: GL_REPLACE_EXT.

This extension is the foundation for other extensions discussed in this chapter.

Note: Some of the functionality of this extension was available in IRIS GL on RealityEngine.

Why Use the Texture Extension?

Using the texture extension, you can improve your program as follows:

- **Save space**. Use the formats best suited for the texture-memory size of a specific system. Smaller internal formats use less texture memory so more textures can be simultaneously resident in texture memory. This can be especially useful if you want to fit a texture into the 16-bit texture storage on current machines.
- **Increase speed**. The extension offers a fine-grained selection of sizes. At times, you may prefer using 16-bit textures to using 32-bit textures because the increase in rendering speed outweighs the slight decrease in image quality. Smaller textures are usually faster and may therefore be preferable, if sufficient.
- **Predict texture size**. Use the texture extension's proxy mechanism to determine whether a texture you intend to use fits into available texture memory.

Using the Texture Extension

To use the texture extension, call **glTexImage1D()** or **glTexImage2D()**, specifying one of the tokens defined by this extension as the *components* parameter.

Table 6-1 lists some of the formats defined by the extension; for a complete list, see the reference page for **glTexImage*()**. The table shows the token names with their corresponding base formats and their desired component resolutions. It also shows Red (R), Green (G), Blue (B), Luminance (L), Alpha (A), and Intensity (I) values for each format. You can query the actual resolution with **glGetTexLevelParameteriv()** using the appropriate token, such as GL_TEXTURE_RED_SIZE_EXT. All OpenGL implementations accept all tokens but may not always allocate a resolution exactly equal to that indicated in the table.

Table 6-1 Some Internal Formats Supported by the Texture Extension

Token Name	Base Format	R	G	B	A	L	I
GL_LUMINANCE8_EXT	GL_LUMINANCE					8	
GL_LUMINANCE4_ALPHA4_EXT	GL_LUMINANCE_ALPHA				4	4	
GL_LUMINANCE12_ALPHA4_EXT	GL_LUMINANCE_ALPHA				4	12	
GL_LUMINANCE16_ALPHA16_EXT	GL_LUMINANCE_ALPHA				16	16	
GL_INTENSITY8_EXT	GL_INTENSITY_EXT						8
GL_RGB4_EXT	GL_RGB	4	4	4			
GL_RGB8_EXT	GL_RGB	8	8	8			
GL_RGBA4_EXT	GL_RGBA	4	4	4	4		
GL_RGB5_A1_EXT	GL_RGBA	5	5	5	1		
GL_RGBA8_EXT	GL_RGBA	8	8	8	8		

The extension redefines the application of a texture to the color components of a fragment. In particular, a new texture environment, GL_REPLACE_EXT, and two new texture formats, GL_ALPHA and GL_INTENSITY_EXT, are available. Table 6-2 illustrates this, with the abbreviations having the following meanings:

Rv, Gv, Bv, Av	result of the texture environment function
Rt, Gt, Bt, At	texture color
Rf, Gf, Bf, Af	fragment color
Rc, Gc, Bc, Ac	texture environment color (see GL_TEXTURE_ENV_COLOR)

Table 6-2 Texture Functions

Base Texture Format	Replace	Modulate	Blend	Decal
GL_ LUMINANCE	Rv = Lt Gv = Lt Bv = Lt Av = Af	Rv = Rf * Lt Gv = Gf * Lt Bv = Bf * Lt Av = Af	Rv = Rf * (1-Lt) + Rc* Lt Gv = Gf * (1-Lt) + Gc * Lt Bv = Bf * (1-Lt) + Bc * Lt Av = Af	undefined
GL_ALPHA	Rv = Rf Gv = Gf Bv = Gf Av = At	Rv = Rf Gv = Gf Bv = Gf Av = Af * At	Rv = Rf Gv = Gf Bv = Bf Av = Af * At	undefined
GL_ INTENSITY_ EXT	Rv = It Gv = It Bv = It Av = It	Rv = Rf * It Gv = Gf * It Bv = Bf * It Av = Af * It	Rv = Rf * (1-It) + Rc * It Gv = Gf * (1-It) + Gc * It Bv = Bf * (1-It) + Bc * It Av = Af * (1-It) + AcIt	undefined
GL_ LUMINANCE _ALPHA	Rv = Lt Gv = Lt Bv = Lt Av = At	Rv = Rf * Lt Gv = Gf * Lt Bv = Bf * Lt Av = Af * At	Rv = Rf * (1-Lt) + Rc * Lt Gv = Gf * (1-Lt) + Gc * Lt Bv = Bf * (1-Lt) + Bc * Lt Av = Af * At	undefined
GL_RGB	Rv = Rt Gv = Gt Bv = Bt Av = Af	Rv = Rf * Rt Gv = Gf * Gt Bv = Bf * Bt Av = Af	Rv = Rf * (1-Rt) + Rc * Rt Gv = Gf * (1-Gt) + Gc * Gt Bv = Bf * (1-Bt) + Bc * Bt Av = Af	Rv = Rt Gv = Gt Bv = Bt Av = Af
GL_RBGA	Rv = Rt Gv = Gt Bv = Bt Av = At	Rv = Rf * Rt Gv = Gf * Gt Bv = Bf * Bt Av = Af * At	Rv = Rf * (1-Rt) + Rc * Rt Gv = Gf * (1-Gt) + Gc * Gt Bv = Bf * (1-Bt) + Bc * Bt Av = Af * At	Rv = Rf * (1-At) + Rt * At Gv = Gf * (1-At) + Gt * At Bv = Bf * (1-At) + Bt * At Av = Af

The Texture Extension Proxy Mechanism

In OpenGL 1.0, it's difficult to find out whether a texture fits into texture memory on a certain system. When you call **glGetIntegerv()** with the parameter GL_MAX_TEXTURE_SIZE, you can't be sure of an accurate result: Texel format, component resolution, and the shape of a texture determine whether the texture fits. To ensure that a texture of any format and resolution fits, the function returns a result based on the worst-case scenario.

A solution to this problem is the proxy mechanism offered by the texture extension. To test whether a certain texture fits, follow these steps:

1. Call **glTexImage2D()** or **glTexImage1D()** with target set to GL_PROXY_TEXTURE_2D_EXT or GL_PROXY_TEXTURE_1D_EXT (see Example 6-1).
2. Test the proxy texture state values to see if the texture would fit and how it would be stored in texture memory by calling **glGetTexLevelParameteriv()** or **glGetTexLevelParameterfv()**. Set *target* to one of the proxy textures: GL_PROXY_TEXTURE_2D_EXT or GL_PROXY_TEXTURE_1D_EXT. Set *pname* to the parameter you are interested in, for example, GL_TEXTURE_WIDTH. If the call returns non-zero, the texture will fit.

   If the texture is too large, all of the proxy state variables are set to zero. If the texture can be accommodated, these values are set to the requested parameters. The proxy 1-D texture behaves like the proxy 2-D texture; however, its state does not include GL_TEXTURE_HEIGHT.

   You still need to call **glTexImage2D()** with *target* GL_TEXTURE_2D to define the actual texture.
3. To determine the maximum array size for a mipmap texture, specify and query the proxy texture at the highest level that accurately reflects the aspect ratio of the desired level zero array.

Example 6-1 provides a code fragment that determines the largest texture that can be allocated for a given internal image format and aspect ratio. The program incrementally selects larger textures to find the maximum that will fit.

Example 6-1 Using the Proxy Mechanism

```
/*
** Demonstrates use of the proxy texture target to probe texture space
** to determine the largest texture which can be allocated for a given
** internal image format and aspect ratio.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <GL/gl.h>
#include <GL/glu.h>

/* (int)floor(log2(x)) */
static int
intFloorLog2(unsigned int x)
{
    int a = 0;
    while (x >>= 1) ++a;
    return a;
}

/* true if x is a power of two */
static GLboolean
isPow2(unsigned int x)
{
    return ((x > 0) && (x & (x - 1) == 0));
}

/* find largest texture with specified internal format and aspect */
static void
findMaxTexture(GLenum internalFormat, int dim,
    int xAspect, int yAspect, int border, int maxMipmapLevel,
    int *widthOut, int *heightOut)
{
    int level = 0, levelOffset = 0;
    int width = 1, height = 1;
    int maxLevel = 0, maxWidth = 0, maxHeight = 0;

    if (xAspect > yAspect) {
        width = xAspect / yAspect;
        levelOffset = intFloorLog2(width);
    } else {
        height = yAspect / xAspect;
        levelOffset = intFloorLog2(height);
    }
```

```
while (1) {
    GLint proxyComponents;

    switch (dim) {
      case 1:
        glTexImage1D(GL_PROXY_TEXTURE_1D_EXT,
                     level, internalFormat,
                     width+2*border, border,
                     GL_RGBA, GL_UNSIGNED_BYTE, NULL);

        glGetTexLevelParameteriv(GL_PROXY_TEXTURE_1D_EXT, level,
                GL_TEXTURE_COMPONENTS, &proxyComponents);
        break;
      case 2:
        glTexImage2D(GL_PROXY_TEXTURE_2D_EXT,
                 level, internalFormat,
                 width+2*border, height+2*border, border,
                 GL_RGBA, GL_UNSIGNED_BYTE, NULL);

        glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D_EXT,level
                 GL_TEXTURE_COMPONENTS,&proxyComponents);
        break;
      default:
        *widthout = 0;
        *heightout = 0;
        return;
    }

    if (proxyComponents != internalFormat) {
       /* proxy allocation failed -- we're done */
        break;
    } else {
       /* proxy allocation succeeded -- see how we did */
           if (level>maxLevel || width>maxWidth ||
                                                 height>maxHeight)
        {
            maxLevel = level;
            maxWidth = width << level;
            maxHeight = height << level;
        }
    }

    if (maxMipmapLevel == 0) {
   /* try the next larger image size at level zero */
```

```
            width <<= 1;
            height <<= 1;
        } else {
       /* try same image size at next higher mipmap level */
            ++level;
            if (level+levelOffset > maxMipmapLevel) {
       /* can't query levels which don't exist--we're done */
                break;
            }
        }
    }

    if (maxWidth>0 && maxHeight>0) {
        maxWidth += 2*border;
        maxHeight += 2*border;
    }
    *widthOut = maxWidth;
    *heightOut = maxHeight;
}
static void
displayTextureSizeInfo(void)
{
    GLint maxTextureSize, maxTextureLevel;
    int maxWidth, maxHeight;

    glGetIntegerv(GL_MAX_TEXTURE_SIZE, &maxTextureSize);
    printf("GL_MAX_TEXTURE_SIZE: %d\n", maxTextureSize);

    maxTextureLevel = intFloorLog2(maxTextureSize);

    findMaxTexture(GL_RGB, 2, 1, 1, 0, 0, &maxWidth, &maxHeight);
    printf("RGB 1:1 (%d x %d)\n", maxWidth, maxHeight);

    findMaxTexture(GL_RGBA, 2, 2, 1, 0, 0, &maxWidth, &maxHeight);
    printf("RGBA 2:1 (%d x %d)\n", maxWidth, maxHeight);

    findMaxTexture(GL_RGB, 2, 1, 1, 0, maxTextureLevel, &maxWidth,
                                                         &maxHeight);
    printf("RGB 1:1 (%d x %d) mipmap\n", maxWidth, maxHeight);

    findMaxTexture(GL_LUMINANCE_ALPHA, 2, 1, 1, 1, 0, &maxWidth,
                                                       &maxHeight);
    printf("LUMINANCE_ALPHA 1:1 (%d x %d) w/border\n", maxWidth,
                                                        maxHeight);
}
```

Extended Functions

See the reference pages of the following functions for related information:

glTexImage1D(), glTexImage2D(), glGetTexLevelParameteriv(), glTexEnvf(), glTexEnvi(), glTexEnvfv(), glTexEnviv(), glGetTexLevelParameterfv().

The Texture Object Extension

The texture object extension, EXT_texture_object, can improve the performance of programs that use multiple textures. It allows you to create as many textures as you need, and provides an efficient way to switch from one texture to another. This is similar to the texdef/texbind model of IRIS GL.

The extension also adds a new texture parameter, the priority, which you can use to tell the system which textures should have priority when it allocates hardware resources.

How Texture Objects Work

In OpenGL 1.0, only one texture can be associated with each of the texture targets GL_TEXTURE_1D and GL_TEXTURE_2D. If you want to render a scene that contains many 2D textures, you have to redefine the 2D texture using **glTexImage2D()** for each texture in the scene. If you want each texture to have different parameters, such as wrap modes and filters, you have to redefine those as well.

The texture object extension lets you create as many textures as you need. You associate a name (a positive number) with a texture object when you create it, and then you define the images and parameters of the texture. As you render your scene, bind the name of each desired texture object to the appropriate texture target. Since binding a texture takes less time than defining one, this is a more efficient way to switch from one texture to another.

Note: Texture objects and display lists are in different name spaces. However, if two contexts are sharing display lists, they also share texture objects.

Using Texture Objects

To create a texture object, bind an unused name to a texture target. The name is considered to be in use until you delete the texture. The following code fragment creates, defines, and binds texture objects with the names 1 and 2:

```
/* the first bind of 1 will create the texture object */
glBindTextureEXT( GL_TEXTURE_2D, 1 );
/* define the texture image and parameterss */
glTexImage2D( GL_TEXTURE_2D, 0, 4, 32, 32, 0, GL_RGBA, GL_BYTE, img1 );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );

/* the first bind of 2 will create the texture object */
glBindTextureEXT( GL_TEXTURE_2D, 2 );
/* define the texture image and parameters */
glTexImage2D( GL_TEXTURE_2D, 0, 4, 64, 64, 0, GL_RGBA, GL_BYTE, img2 );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );

/* bind each texture object as scene is rendered */
glBindTextureEXT( GL_TEXTURE_2D, 1 );
<draw some primitive>
glBindTextureEXT( GL_TEXTURE_2D, 2 );
<draw some primitive>
```

There is also an example program in the source tree that demonstrates how to use texture objects.

Texture Object Names

You can create a texture object with any name, as long as it isn't already the name of another texture object. To find out whether a particular name is in use, call **glIsTextureEXT()**. As a convenience, **glGenTexturesEXT()** returns a set of texture names that are known to be unused.

To delete one or more texture objects, call **glDeleteTexturesEXT()**. After a texture object is deleted, its name is freed; it has no contents, and its resources are returned to the system.

Editing and Querying Texture Objects

Once a texture object has been created, you can change its images and parameters at any time. You edit a texture object the same way you define it: you bind the texture object to a target and then call **glTexImage*()** or **glTexParameter*()**. These commands take a texture target as an argument, and they affect the texture object that is currently bound to that target. Similarly, you can query the parameters and images of a texture by calling **glGetTexParameter*()**, **glGetTexLevelParameter*()**, and **glGetTexImage()**, which also operate on the currently bound texture.

To find out which texture object is currently bound to each texture target, call **glGetIntegerv()** with GL_TEXTURE_1D_BINDING_EXT, GL_TEXTURE_2D_BINDING_EXT, or GL_TEXTURE_3D_BINDING_EXT (if your system supports 3D textures).

Texture Priorities and Residency

On some systems, textures must reside in hardware resources such as texture memory before they can be used. When you bind a texture, the system may need to swap out some other textures to make room, if the application's textures do not all fit in texture memory.

An application can guide the system in determining which textures should remain resident by specifying a priority for each texture using **glPrioritizeTexturesEXT()**. Each priority value is clamped to the range 0 to 1, with 0 indicating the lowest priority and hence the least likelihood of being resident, and 1 indicating the highest priority. You can also set the priority of a single texture by calling **glTexParameter*()** with GL_TEXTURE_PRIORITY_EXT. Note that the performance effects, if any, of setting texture priorities are entirely system dependent.

You can query whether a set of texture objects is resident by calling **glAreTexturesResidentEXT()**, and you can query whether a single texture object is resident by calling **glGetTexParameter*()** with GL_TEXTURE_RESIDENT_EXT.

Default Textures

You can still access the OpenGL 1.0 textures, now called the “default textures,” by binding 0 to a texture target. This is in fact the initial binding when you create a rendering context. When you create a texture object, its images and parameters are the same as the initial settings of the default textures.

You can set the priority of a default texture by calling **glTexParameter*()**; you cannot use **glPrioritizeTexturesEXT()** with a default texture. Similarly, you can query the residence of a default texture with **glGetTexParameter*()**, but you cannot do so with **glAreTexturesResidentEXT()**.

New and Extended Functions

New: glGenTexturesEXT(), glDeleteTexturesEXT(), glIsTextureEXT(), glBindTextureEXT() glPrioritizeTexturesEXT(), glAreTexturesResidentEXT().

Extended: glGetParameterTex*().

The Subtexture Extension

The subtexture extension, EXT_subtexture, lets you replace a subrectangle of an existing texture image without affecting the remaining portions of the image.

Why Use the Subtexture Extension?

Using a subtexture is the fastest way to update only the image of a texture. When you need to redefine only the image of the texture and keep all other parameters, using subtexture is much faster than using the **glTexImage*()** call.

The subtexture extension is especially useful under the following circumstances:

- When you’re dealing with an image that doesn’t fit into texture memory, you can use the extension to load only parts of the texture and replace them as appropriate.
- When you’re using video frames as textures, none of the common video formats (NTSC, PAL, SECAM, HD-TV, and so on) have height and width that’s a power of 2. This is problematic because height and width of those dimensions is required for textures. You therefore create a NULL texture whose dimensions are larger than the video frame and then load the video frame as a subtexture.
- When you’re working with an animation, loading a large texture may decrease the frame rate. Instead, load several subtextures over multiple frames to maintain a high frame rate.

Using Subtextures

To load a subrectangle of a texture image, call **glTexSubImage1DEXT()**, **glTexSubImage2DEXT()**—prototype below—or **glTexSubImage3DEXT()**.

```
void glTexSubImage2DEXT(enum target, int level, int xoffset,int yoffset,
    sizei width, sizei height, enum format, enum type, const void* pixels)
```

- *target* specifies the existing texture image for which the subimage is being defined.
- *level*, *depth*, *format*, *type*, and *pixels* correspond to the arguments of **glTexImage2D()**; *width* and *height* are those of the subimage.
- *xoffset* and *yoffset* specify texel offsets in the x and y directions within the texture image being modified.

Using any uninitialized portion of a texture while drawing yields undefined results.

Using Null Images With Subtextures

It is sometimes useful to define the parameters of a texture image without actually initializing the contents of that image. For example, if you want to load a frame of an NTSC video as a texture, you need to first define a region larger than the video frame that meets the height and width requirements of textures (power of 2). Null images are also useful when you want to first define the parameters of a texture and later initialize the texture image using **glTexSubImage*()** calls.

To create a null image, call one of the texture creation functions, for example, **glTexImage2D()** with *pixels* set to the null pointer. The specified texture is created, but no pixels are processed. Using an uninitialized part of a texture yields undefined results.

The following code fragment creates a null image texture, then loads a subimage:

```
{
/* need to initialize "image" to something interesting */
    static unsigned char image[32][32][4];
/* create a 256 x 256 null texture */
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB5_EXT, 256, 256, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, NULL);
/* load a 32 x 32 subimage starting at 100,110 */
    glTexSubImage2DEXT(GL_TEXTURE_2D, 0, 100, 110, 32, 32,
                       GL_RGBA, GL_UNSIGNED_BYTE, image);
}
```

New Functions

glTexSubImage1DEXT(), glTexSubImage2DEXT(), glTexSubImage2DEXT().

The Copy Texture Extension

The copy texture extension, EXT_copy_texture, lets you load texture images directly from the framebuffer. You can replace either part of the texture image or the whole image. If your system supports the 3D texture extension, you can also load 2D slices of a 3D texture from the framebuffer. You can't load an entire 3D texture because the framebuffer is 2D.

Why Use the Copy Texture Extension?

The copy texture extension lets you do multi-pass rendering, using the full set of OpenGL rendering capabilities to generate a texture you can use for further drawing. For example, you can render a scene, copy the resulting image to texture memory, and then apply the texture to a triangle mesh. By changing the mesh, you can warp the scene, break it into pieces, wrap it around complex objects, and so on.

Copying textures is also the only means of directly using video frames as textures, in conjunction with the video source extension. See "The Video Source Extension" on page 183.

Copying Texture Images

The functions **glCopyTexImage1DEXT()** and **glCopyTexImage2DEXT()** copy image data from the color buffer specified by the current read buffer rather than accepting image data from host memory.

To copy an image from the framebuffer, call **glCopyTexImage2DEXT()**, **glCopyTexImage1DEXT()**, **glCopyTexSubImage1DEXT()**, or **glCopyTexSubImage2DEXT()**. The prototypes for the four functions are very similar; here's **glCopyTexImage2DEXT()** as an example:

```
void glCopyTexImage2DEXT( GLenum target,GLint level,GLenum internalformat,
        GLint x, GLint y, GLsizei width, GLsizei height, GLint border)
```

- *target* and *level* are equivalent to their **glTexImage2D()** counterparts, except that *target* does not accept GL_PROXY_TEXTURE_2D_EXT.
- *internalformat* is identical to the *components* parameter of **glTexImage2D()**, except that **glCopyTexImage2DEXT()** does not accept the values 1, 2, 3, and 4 but instead the internal formats defined by the texture extension, some of which are listed in Table 6-1 on page 87. For complete information, see the reference page.
- Each function reads image data from a color buffer region that is (*width+2*border_pixel)* wide and *(height+2*border_pixel*) high, and whose lower-left pixel has window coordinates x,y.

The pixel values are processed exactly as if **glCopyPixels()** had been called, but the process stops just before final conversion, that is, before clamping and conversion to fragments. At this point all pixel component values are clamped to [0,1], and then treated exactly as if the corresponding **glTexImage*()** or **glTexSubImage*()** had been called. Pixel ordering is such that lower x screen coordinates correspond to lower i (horizontal) texture coordinates, and lower y screen coordinates correspond to lower j (vertical) texture coordinates.

If the subtexture extension and the 3D texture extension are both supported, you can also use **glCopyTexSubImage3DEXT()**. See “The 3D Texture Extension” on page 100 for more information about 3D textures.

New Functions

glCopyTexImage1DEXT(), glCopyTexImage2DEXT(), glCopyTexSubImage1DEXT(), glCopyTexSubImage2DEXT(), glCopyTexSubImage3DEXT().

The 3D Texture Extension

The 3D texture extension, EXT_texture3D, defines 3-dimensional texture mapping and in-memory formats for 3D images, and adds pixel storage modes to support them.

3D textures can be thought of as an array of 2D textures, as illustrated in Figure 6-1.

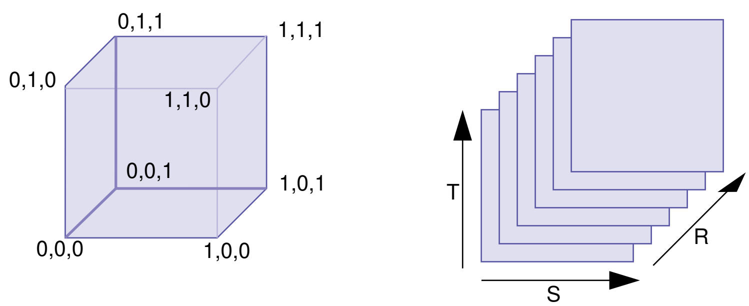

Figure 6-1 3D Texture

A 3D texture is mapped into (s,t,r) coordinates such that its lower left back corner is (0,0,0) and its upper right front corner is (1,1,1).

Why Use the 3D Texture Extension?

3D textures are useful for

- volume rendering and examining a 3D volume one slice at a time
- animating textured geometry, for example, people that move
- solid texturing, for example, wood, marble and so on
- eliminating distortion effects that occur when you try to map a 2D image onto 3D geometry

Texel values defined in a 3D coordinate system form a texture volume. You can extract textures from this volume by intersecting it with a 3D plane, as shown in Figure 6-2.

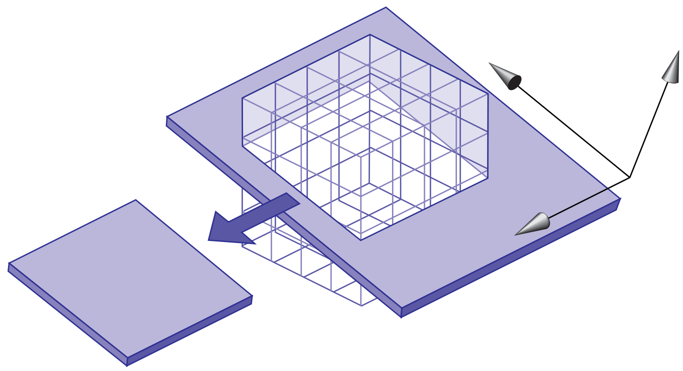

Figure 6-2 Extracting a Planar Texture From a 3D Texture Volume

The resulting texture, applied to a polygon, is the intersection of the volume and the plane. The orientation of the plane is determined from the texture coordinates of the vertices of the polygon.

Using 3D Textures

To create a 3D texture, use **glTexImage3DEXT()**. The function is defined like **glTexImage2DEXT()** but has a *depth* argument that specifies how many "slices" the texture consists of. Internal formats are supported.

The extension also provides the following additional features:

- **Pixel storage modes.** The extension extends the pixel storage modes by adding four new state variables:
  - GL_(UN)PACK_IMAGE_HEIGHT_EXT defines the height of the image the texture is read from, analogous to the GL_(UN)PACK_LENGTH variable for image width.
  - GL_(UN)PACK_SKIP_IMAGES_EXT determines an initial skip analogous to GL_(UN)PACK_SKIP_PIXELS and GL_(UN)PACK_SKIP_LINES.

  All four default to zero.
- **Texture wrap modes.** The functions **glTexParameter*()**, accept the additional token value GL_TEXTURE_WRAP_R_EXT.

It affects the R coordinate in the same way that GL_TEXTURE_WRAP_S affects the S coordinate and GL_TEXTURE_WRAP_T affects the T coordinate. The default value is GL_REPEAT.

- **Mipmapping.** Mipmapping for two-dimensional textures is discussed in the section “Multiple Levels of Detail,” in Chapter 9, “Texture Mapping,” of the *OpenGL Programming Guide*. Mipmapping for 3D textures works the same way: A 3D mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one. The filtering options GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, and GL_LINEAR_MIPMAP_LINEAR apply to subvolumes instead of texels.
- **Proxy textures.** Use the proxy texture GL_PROXY_TEXTURE_3D_EXT to query an implementation’s maximum configuration. For more information on proxy textures, see “The Texture Extension Proxy Mechanism” on page 89.
- **Querying.** Use the following call to query the 3D texture:

  ```
  glGetTexImage(GL_TEXTURE_3D_EXT)
  ```
- Subvolumes of the 3D texture can be replaced using **glTexSubImage3DEXT()** and **glCopyTexSubImage3DEXT()** (see “The Subtexture Extension” on page 96).

3D Texture Example Program

The code fragment presented in this section illustrates the use of the extension. The complete program is included in the example source tree.

```
/*
 * tex3d - simple 3D texturing program
 *
 * Shows a 3D texture by drawing slices through it.
 */
/* compile: cc -o tex3d tex3d.c -lGL -lX11 */

#include <GL/glx.h>
#include <GL/glu.h>
#include <X11/keysym.h>
#include <stdlib.h>
#include <stdio.h>

static int attributeList[] = { GLX_RGBA, None };
```

```
unsigned int tex[64][64][64];

/* generate a simple 3D texture */
static void
make_texture(void) {
    int i, j, k;
    unsigned int *p = &tex[0][0][0];

    for (i=0; i<64; i++) {
        for (j=0; j<64; j++) {
            for (k=0; k<64; k++) {
                if (i < 10 || i > 48 ||
                    j < 10 || j > 48 ||
                    k < 10 || k > 48) {
                    if (i < 2 || i > 62 ||
                        j < 2 || j > 62 ||
                        k < 2 || k > 62) {
                        *p++ = 0x00000000;
                    } else {
                        *p++ = 0xff80ffff;
                    }
                } else {
                    *p++ = 0x000000ff;
                }
            }
        }
    }
}

static void
init(void) {
    make_texture();
    glEnable(GL_TEXTURE_3D_EXT);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);
    glClearColor(0.2,0.2,0.5,1.0);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    glMatrixMode(GL_PROJECTION);
    gluPerspective(60.0, 1.0, 1.0, 100.0 );
    glMatrixMode(GL_MODELVIEW);
    glTranslatef(0.,0.,-3.0);
    glMatrixMode(GL_TEXTURE);
```

```
    /* Similar to defining a 2D texture, but note the setting of the */
    /* wrap parameter for the R coordinate.  Also, for 3D textures   */
    /* you probably won't need mipmaps, hence the linear min filter. */
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_MIN_FILTER,
                                                       GL_LINEAR);
    glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_R_EXT,
                                                        GL_CLAMP);
    glTexImage3DEXT(GL_TEXTURE_3D_EXT, 0, 4, 64, 64, 64, 0,
                    GL_RGBA, GL_UNSIGNED_BYTE, tex);
}

#define NUMSLICES 256

static void
draw_scene(void) {
    int i;
    float r, dr, z, dz;

    glColor4f(1, 1, 1, 1.4/NUMSLICES);
    glClear(GL_COLOR_BUFFER_BIT);

    /* Display the entire 3D texture by drawing a series of quads */
    /* that  slice through the texture coordinate space.  Note that */
    /* the transformations below are applied to the texture matrix, */
    /* not the modelview matrix. */

    glLoadIdentity();
    /* center the texture coords around the [0,1] cube */
    glTranslatef(.5,.5,.5);
    /* a rotation just to make the picture more interesting */
    glRotatef(45.,1.,1.,.5);

    /* to make sure that the texture coords, after arbitrary */
    /* rotations, still fully contain the [0,1] cube, make them span */
    /* a range sqrt(3)=1.74 wide */
    r = -0.87; dr = 1.74/NUMSLICES;
    z = -1.00; dz = 2.00/NUMSLICES;
    for (i=0; i < NUMSLICES; i++) {
        glBegin(GL_TRIANGLE_STRIP);
        glTexCoord3f(-.87,-.87,r); glVertex3f(-1,-1,z);
        glTexCoord3f(-.87, .87,r); glVertex3f(-1, 1,z);
        glTexCoord3f( .87,-.87,r); glVertex3f( 1,-1,z);
```

```
        glTexCoord3f( .87, .87,r); glVertex3f( 1, 1,z);
        glEnd();
        r += dr;
        z += dz;
    }
}

/* process input and error functions and main(), which handles window
 * setup, go here.
 */
```

New Functions

glTexImage3DEXT().

The Texture Color Table Extension

The texture color table extension, SGI_texture_color_table, adds a color lookup table to the texture mechanism. The table is applied to the filtered result of a texture lookup before that result is used in the texture environment equations.

Why Use a Texture Color Table?

Here are two example situations in which the texture color table extension is useful:

- **Volume rendering.** You can store something other than color in the texture (for example, a physical attribute like bone density) and use the table to map that density to an RGB color. This is useful if you want to display just that physical attribute (for example, bone density) and also if you want to distinguish between that attribute and another (for example, muscle density). You can selectively replace the table to display different features. Note that updating the table can be faster than updating the texture. (This technique also called "false color imaging" or "segmentation").
- **Representing shades (gamut compression).** If you need to display a high color-resolution image using a texture with low color-component resolution, the result is often unsatisfactory. A 16-bit texel doesn't offer a lot of shades for each color, because each color component has to be evenly spaced between black and the strongest shade of the color. If an image contains several shades of light blue but no

dark blue, for example, the on-screen image can't represent that easily because only a limited number of shades of blue, many of them dark, are available. When using a color table, you can "stretch" the colors.

Using Texture Color Tables

To use a texture color table, define a color table, as described in "The Color Table Extension" on page 136. Use GL_TEXTURE_COLOR_TABLE_SGI as the value for the *target* parameter of the various commands, keeping in mind the following points:

- The table size, specified by the *width* parameter of **glColorTableSGI(),** is limited to powers of two.
- The maximum table size is 256 entries.
- Use GL_PROXY_TEXTURE_COLOR_TABLE_SGI to find out whether there is enough room for the texture color table in exactly the manner described in "The Texture Extension Proxy Mechanism" on page 89.

The following code fragment loads a table that inverts a texture. It uses a GL_LUMINANCE external format table to make identical R, G, and B mappings.

```
loadinversetable()
{
    static unsigned char table[256];
    int i;

    for (i = 0; i < 256; i++) {
        table[i] = 255-i;
    }

    glColorTableSGI(GL_TEXTURE_COLOR_TABLE_SGI, GL_RGBA8_EXT,
                    256, GL_LUMINANCE, GL_UNSIGNED_BYTE, table);
    glEnable(GL_TEXTURE_COLOR_TABLE_SGI);
}
```

The Sharpen Texture Extension

This section discusses the sharpen texture extension, SGIS_sharpen_texture. This extension and the detail texture extension (see "The Detail Texture Extension" on page 112) are useful in situations where you want to maintain good image quality when a texture must be magnified for close-up views.

When a textured surface is viewed close up, the magnification of the texture can cause blurring. One way to reduce blurring is to use a higher-resolution texture for the close-up view, at the cost of extra storage. The sharpen texture extension offers a way to keep the image crisp without increasing texture storage requirements.

Sharpen texture works best when the high-frequency information is used to represent edge information, for example:

- In a stop sign, the edges of the letters have distinct outlines, and bilinear magnification normally causes the letters to blur. Sharpen texture keeps the edges crisp.
- In a tree texture, the alpha values are high inside the outline of the tree and low outside the outline (where the background shows through). Bilinear magnification normally causes the outline of the tree to blur. Sharpen texture, applied to the alpha component, keeps the outline crisp.

Sharpen texture works by extrapolating from mipmap levels 1 and 0 to create a magnified image that has sharper features than either mipmap.

Using the Sharpen Texture Extension

This section first explains how to use the sharpen texture extension to sharpen the component of your choice. It then gives some background information about how the extension works and explains how you can customize the LOD extrapolation function.

How to Use the Sharpen Texture Extension

You can use the extension to sharpen the alpha component, the color components, or both, depending on the magnification filter. To specify sharpening, use one of the magnification filters in Table 6-3.

Table 6-3 Magnification Filters for Sharpen Texture

GL_TEXTURE_MAG_FILTER	Alpha	Red, Green, Blue
GL_LINEAR_SHARPEN_SGIS	sharpen	sharpen
GL_LINEAR_SHARPEN_COLOR_SGIS	bilinear	sharpen
GL_LINEAR_SHARPEN_ALPHA_SGIS	sharpen	bilinear

For example, suppose that a texture contains a picture of a tree in the color components, and the opacity in the alpha component. To sharpen the outline of the tree, use

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                               GL_LINEAR_SHARPEN_ALPHA_SGIS);
```

How Sharpen Texture Works

When OpenGL applies a texture to a pixel, it computes a level of detail (LOD) factor that represents the amount by which the base texture (that is, level 0) must be scaled. LOD n represents a scaling of 2^{-n}. For example, if OpenGL needs to magnify the base texture by a factor of 4 in both S and T, the LOD is -2. Note that magnification corresponds to negative values of LOD.

To produce a sharpened texel at level-of-detail n, OpenGL adds the weighted difference between the texel at LOD 0 and LOD 1 to LOD 0; that is:

```
LODn = LOD0 + weight(n) * (LOD0 - LOD1)
```

The variables are defined as follows:

n	level-of-detail
weight(n)	LOD extrapolation function
LOD0	base texture
LOD1	texture at mipmap level 1

By default, OpenGL uses a linear extrapolation function, where weight(n) = $-n/4$. You can customize the LOD extrapolation function by specifying its control points, as discussed in the next section.

Customizing the LOD Extrapolation Function

With the default linear LOD extrapolation function, the weight may be too large at high levels of magnification, that is, as n becomes more negative. This can result in so much extrapolation that noticeable bands appear around edge features, an artifact known as "ringing." In this case, it's useful to create a nonlinear LOD extrapolation function.

Figure 6-3 shows LOD extrapolation curves as a function of magnification factors. The curve on the left is the default linear extrapolation, where weight(n) = $-n/4$. The curve on the right is a nonlinear extrapolation, where the LOD extrapolation function is modified to control the amount of sharpening so that less sharpening is applied as the magnification factor increases. The function is defined for n less than or equal to 0.

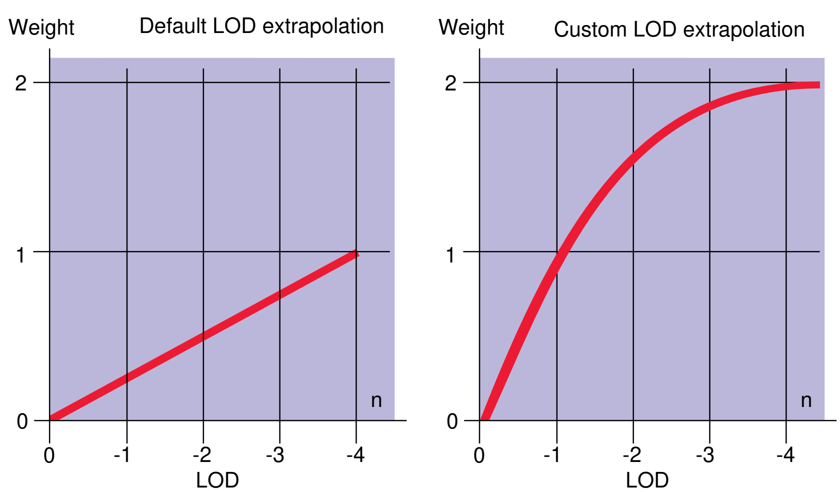

Figure 6-3 LOD Extrapolation Curves

Use **glSharpenTexFuncSGIS()** to specify control points for shaping the LOD extrapolation function. Each control point contains a pair of values; the first value specifies the LOD, and the second value specifies a weight multiplier for that magnification level. (Remember that the LOD values are negative.)

For example, to gradually ease the sharpening effect, use a nonlinear LOD extrapolation curve—as shown on the right in Figure 6-3—with these control points:

```
GLfloat points[] = {
     0., 0.,
-    1., 1.,
-    2., 1.7,
-    4., 2.
};
glSharpenTexFuncSGIS(GL_TEXTURE_2D, 4, points);
```

Note that how these control points determine the function is system dependent. For example, your system may choose to create a piecewise linear function, a piecewise quadratic function, or a cubic function. However, regardless of which kind of function is chosen, the function will pass through the control points.

Using Sharpen Texture and Texture Object

If you are using the texture object extension, each texture object contains its own LOD extrapolation function and magnification filter. Setting the function or the filter therefore affects only the texture object that is currently bound to the texture target.

Sharpen Texture Example Program

Example 6-2 illustrates the use of the sharpen texture example program. Because of space limitations, the sections dealing with X Window System setup and some of the keyboard input are omitted. The complete example is included in the source tree as *sharpen.c*.

Example 6-2 Sharpen Texture Example

```
/* tree texture: high alpha in foreground, zero alpha in background */
#define B 0x00000000
#define F 0xA0A0A0ff
unsigned int tex[] = {
    B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,F,F,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,F,F,B,B,B,B,B,B,B,
    B,B,B,B,B,B,F,F,F,F,B,B,B,B,B,B,
    B,B,B,B,B,B,F,F,F,F,B,B,B,B,B,B,
    B,B,B,B,B,F,F,F,F,F,F,B,B,B,B,B,
    B,B,B,B,B,F,F,F,F,F,F,B,B,B,B,B,
    B,B,B,B,F,F,F,F,F,F,F,F,B,B,B,B,
    B,B,B,B,F,F,F,F,F,F,F,F,B,B,B,B,
    B,B,B,F,F,F,F,F,F,F,F,F,F,B,B,B,
    B,B,B,F,F,F,F,F,F,F,F,F,F,B,B,B,
    B,B,F,F,F,F,F,F,F,F,F,F,F,F,B,B,
    B,B,F,F,F,F,F,F,F,F,F,F,F,F,B,B,
    B,B,B,B,B,B,F,F,F,F,B,B,B,B,B,B,
    B,B,B,B,B,B,F,F,F,F,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,
};

static void
init(void) {
    glEnable(GL_TEXTURE_2D);
    glMatrixMode(GL_PROJECTION);
    gluPerspective(60.0, 1.0, 1.0, 10.0 );
    glMatrixMode(GL_MODELVIEW);
    glTranslatef(0.,0.,-2.5);

    glColor4f(0,0,0,1);
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    /* sharpening just alpha keeps the tree outline crisp */
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR_SHARPEN_ALPHA_SGIS);
    /* generate mipmaps; levels 0 and 1 are needed for sharpening */
    gluBuild2DMipmaps(GL_TEXTURE_2D, 4, 16, 16, GL_RGBA,
                      GL_UNSIGNED_BYTE, tex);
}
```

```
static void
draw_scene(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_TRIANGLE_STRIP);
        glTexCoord2f( 0, 1); glVertex2f(-1,-1);
        glTexCoord2f( 0, 0); glVertex2f(-1, 1);
        glTexCoord2f( 1, 1); glVertex2f( 1,-1);
        glTexCoord2f( 1, 0); glVertex2f( 1, 1);
    glEnd();
    glFlush();
}
```

New Functions

glSharpenTexFuncSGIS(), glGetSharpenTexFuncSGIS().

The Detail Texture Extension

This section discusses the detail texture extension, SGIS_detail_texture, which like the sharpen texture extension (see "The Sharpen Texture Extension" on page 107) is useful in situations where you want to maintain good image quality when a texture must be magnified for close-up views.

Ideally, programs should always use textures that have high enough resolution to allow magnification without blurring. High-resolution textures maintain realistic image quality for both close-up and distant views. For example, in a high-resolution road texture, the large features—such as potholes, oil stains, and lane markers that are visible from a distance—as well as the asphalt of the road surface, look realistic no matter where the viewpoint is.

Unfortunately, a high-resolution road texture with that much detail may be as large as 2K x 2K, which may exceed the texture storage capacity of the system. Making the image close to or equal to the maximum allowable size still leaves little or no memory for the other textures in the scene.

The detail texture extension provides a solution for representing a 2K x 2K road texture with smaller textures. Detail texture works best for a texture with high-frequency information that is not strongly correlated to its low-frequency information. This occurs in images that have a uniform color and texture variation throughout, such as a field of grass or a wood panel with a uniform grain. If high-frequency information in your texture is used to represent edge information (for example, a stop sign or the outline of a tree) consider the sharpen texture extension (see “The Sharpen Texture Extension” on page 107).

Using the Detail Texture Extension

Because the high-frequency detail in a texture (for example, a road) is often virtually the same across the entire texture, the detail from an arbitrary portion of the texture image can be used as the detail across the entire image.

When you use the detail texture extension, the high-resolution texture image is represented by the combination of a low-resolution texture image and a small high-frequency detail texture image (the detail texture). OpenGL combines these two images during rasterization to create an approximation of the high-resolution image.

This section first explains how to create the detail texture and the low resolution texture that are used by the extension, then briefly looks at how detail texture works and how to customize the LOD interpolation function, which controls how OpenGL combines the two textures.

Creating a Detail Texture and a Low-Resolution Texture

This section explains how to convert a high-resolution texture image into a detail texture and a low-resolution texture image. For example, for a 2K x 2K road texture, you may want to use a 512 x 512 low-resolution base texture and a 256 x 256 detail texture. Follow these steps to create the textures:

1. Make the low-resolution image using *izoom* or another resampling program to make the low-resolution image by shrinking the high-resolution image by 2^n.

   In this example, n is 2, so the resolution of the low-resolution image is 512 x 512. This band-limited image has the two highest-frequency bands of the original image removed from it.

2. Create the subimage for the detail texture using *subimage* or another tool to select a 256 x 256 region of the original high-resolution image, whose n highest-frequency bands are characteristic of the image as a whole. (For example, rather than choosing a subimage from the lane markings or a road, choose an area in the middle of a lane.)

3. Optionally, make this image self-repeating along its edges to eliminate seams.

4. Create a blurry version of the 256×256 subimage as follows:
   - First shrink the 256×256 subimage by 2^n, to 64×64.
   - Then scale the resulting image back up to 256×256.

   The image is blurry because it is missing the two highest-frequency bands present in the two highest levels of detail.
5. Subtract the blurry subimage from the original subimage. This difference image—the detail texture—has only the two highest frequency bands.
6. Define the low-resolution texture (the base texture created in Step 1) with the GL_TEXTURE_2D target and the detail texture (created in Step 5) with the GL_DETAIL_TEXTURE_2D_SGIS target. In the road example, you would use

   ```
   GLvoid *detailtex, *basetex;
   glTexImage2D(GL_DETAIL_TEXTURE_2D_SGIS, 0, 4, 256, 256, 0, GL_RGBA,
               GL_UNSIGNED_BYTE, detailtex);
   glTexImage2D(GL_TEXTURE_2D, 0, 4, 512, 512, 0, GL_RGBA,
                GL_UNSIGNED_BYTE, basetex);
   ```

   The internal format of the detail texture and the base texture must match exactly.
7. Set the GL_DETAIL_TEXTURE_LEVEL_SGIS parameter to specify the level at which the detail texture resides. In the road example, the detail texture is level -2 (since the original 2048 x 2048 texture is two levels below the 512 x 512 base texture):

   ```
   glTexParameteri(GL_TEXTURE_2D, GL_DETAIL_TEXTURE_LEVEL_SGIS, -2);
   ```

   Since the actual detail texture supplied to OpenGL is 256 x 256, OpenGL replicates the detail texture as necessary to fill a 2048 x 2048 texture. In this case, the detail texture repeats eight times in S and in T.

   Note that the detail texture level is set on the GL_TEXTURE_2D target, not on GL_DETAIL_TEXTURE_2D_SGIS.
8. Set the magnification filter to specify whether the detail texture is applied to the alpha or color component, or both. Use one of the filters in Table 6-3. For example, to apply the detail texture to both alpha and color components, use

   ```
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                   GL_LINEAR_DETAIL_SGIS);
   ```

Note that the magnification filter is set on the GL_TEXTURE_2D target, not on GL_DETAIL_TEXTURE_2D_SGIS.

Table 6-4 Magnification Filters for Detail Texture

GL_TEXTURE_MAG_FILTER	Alpha	Red, Green, Blue
GL_LINEAR_DETAIL_SGIS	detail	detail
GL_LINEAR_DETAIL_COLOR_SGIS	bilinear	detail
GL_LINEAR_DETAIL_ALPHA_SGIS	detail	bilinear

How Detail Texture Is Computed

For each pixel that OpenGL textures, it computes an LOD-based factor that represents the amount by which the base texture (that is, level 0) is scaled. LOD n represents a scaling of 2^{-n}. Negative values of LOD correspond to magnification of the base texture.

To produce a detailed textured pixel at level-of-detail n, OpenGL uses one of the two formulas shown in Table 6-5, depending on the detail texture mode.

Table 6-5 How Detail Texture Is Computed

GL_DETAIL_TEXTURE_MODE_SGIS	Formula
GL_ADD	LODn = LOD0 + weight(n) ∗ DET
GL_MODULATE	LODn = LOD0 + weight(n) * DET * LOD0

The variables in the formulas are defined as follows:

n	level of detail
weight(n)	detail function
LOD0	base texture
DET	detail texture

For example, to specify GL_ADD as the detail mode, use

```
glTexParameteri(GL_TEXTURE_2D, GL_DETAIL_TEXTURE_MODE_SGIS, GL_ADD);
```

Customizing the Detail Function

In the road example, the 512 x 512 base texture is LOD 0. The detail texture combined with the base texture represents LOD -2, which is called the maximum-detail texture.

By default, OpenGL performs linear interpolation between LOD 0 and LOD -2 when a pixel's LOD is between 0 and -2. Linear interpolation between more than one LOD can result in aliasing. To minimize aliasing between the known LODs, OpenGL lets you specify a nonlinear LOD interpolation function.

Figure 6-4 shows the default linear interpolation curve and a nonlinear interpolation curve that minimizes aliasing when interpolating between two LODs.

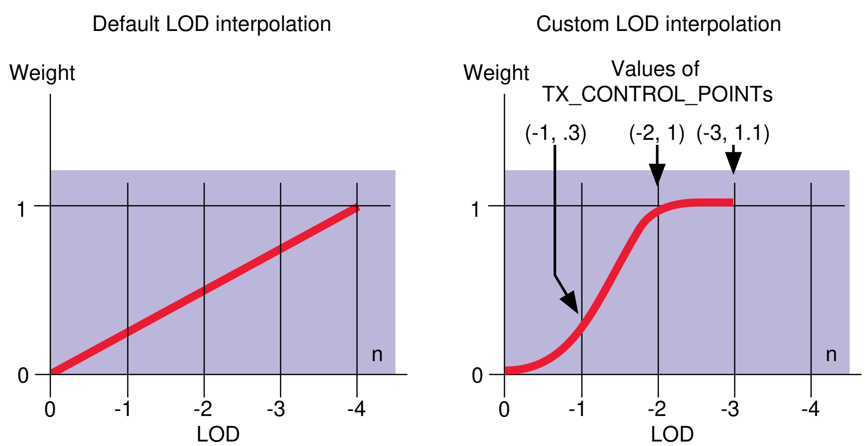

Figure 6-4 LOD Interpolation Curves

The basic strategy is to use very little of the detail texture until the LOD is within one LOD of the maximum-detail texture. More of the information from the detail texture can be used as the LOD approaches LOD -2. At LOD -2, the full amount of detail is used, and the resultant texture exactly matches the high-resolution texture.

Use **glDetailTexFuncSGIS()** to specify control points for shaping the LOD interpolation function. Each control point contains a pair of values; the first value specifies the LOD, and the second value specifies the weight for that magnification level.

The following control points can be used to create a nonlinear interpolation function (as shown above in Figure 6-4):

```
GLfloat points[] = {
     0.0, 0.0,
    -1.0, 0.3,
    -2.0, 1.0,
    -3.0, 1.1
};
glDetailTexFuncSGIS(GL_TEXTURE_2D, 4, points);
```

Note that how these control points determine a function is system dependent. For example, your system may choose to create a piecewise linear function, a piecewise quadratic function, or a cubic function. However, regardless of which kind of function is chosen, the function passes through the control points.

Using Detail Texture and Texture Object

If you are using the texture object extension, the base texture and the detail texture are separate texture objects. You can bind any base texture object to GL_TEXTURE_2D and any detail texture object to GL_DETAIL_TEXTURE_2D_SGIS. Each base texture object contains its own detail mode, magnification filter, and LOD interpolation function. Setting these parameters therefore affects only the texture object that is currently bound to GL_TEXTURE_2D. (If you set these parameters on the detail texture object, they are ignored.)

Detail Texture Example Program

Example 6-3 is a code fragment taken from a simple detail texture example program. The complete example is included in the source tree as *detail.c.*

Example 6-3 Detail Texture Example

```
unsigned int tex[128][128];
unsigned int detailtex[256][256];

static void
make_textures(void) {
    int i, j;
    unsigned int *p;

    /* base texture is solid gray */
```

```
    p = &tex[0][0];
    for (i=0; i<128*128; i++) *p++ = 0x808080ff;

    /* detail texture is a yellow grid over a gray background     */
    /* this artificial detail texture is just a simple example    */
    /* you should derive a real detail texture from the original  */
    /* image as explained in the text.                            */
    p = &detailtex[0][0];
    for (i=0; i<256; i++) {
        for (j=0; j<256; j++) {
            if (i%8 == 0 || j%8 == 0) {
                *p++ = 0xffff00ff;
            } else {
                *p++ = 0x808080ff;
            }
        }
    }
}

static void
init(void) {
    make_textures();

    glEnable(GL_TEXTURE_2D);
    glMatrixMode(GL_PROJECTION);
    gluPerspective(90.0, 1.0, 0.3, 10.0 );
    glMatrixMode(GL_MODELVIEW);
    glTranslatef(0.,0.,-1.5);

    glClearColor(0.0, 0.0, 0.0, 1.0);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

    /* NOTE: parameters are applied to base texture, not the detail */
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR_DETAIL_SGIS);
    glTexParameteri(GL_TEXTURE_2D, GL_DETAIL_TEXTURE_LEVEL_SGIS, -1);
    glTexImage2D(GL_TEXTURE_2D,
                 0, 4, 128, 128, 0, GL_RGBA, GL_UNSIGNED_BYTE, tex);
    glTexImage2D(GL_DETAIL_TEXTURE_2D_SGIS,
                 0, 4, 256, 256, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                 detailtex);
}
```

```
static void
draw_scene(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_TRIANGLE_STRIP);
        glTexCoord2f( 0, 0); glVertex3f(-1,-0.4, 1);
        glTexCoord2f( 0, 1); glVertex3f(-1,-0.4,-1);
        glTexCoord2f( 1, 0); glVertex3f( 1,-0.4, 1);
        glTexCoord2f( 1, 1); glVertex3f( 1,-0.4,-1);
    glEnd();
    glFlush();
}
```

New Functions

glDetailTexFuncSGIS(), glGetDetailTexFuncSGIS().

Texture Clamp Extensions

This section first provides some background information on texture clamping. It then looks at reasons for using the texture clamping extensions and explains how to use them. The two extensions are

- the texture edge clamp extension, SGIS_texture_edge_clamp
- the texture border clamp extension, SGIS_texture_border_clamp

Texture clamping is especially useful for nonrepeating textures.

Texture Clamping Background Information

OpenGL 1.0 provides clamping of texture coordinates: Any values greater than 1.0 are set to 1.0, any values less than 0.0 are set to 0.0. Clamping is useful for applications where you want a single copy of the texture to be mapped onto a large surface. Clamping is discussed in detail in the section “Repeating and Clamping Textures” in Chapter 9, “Texture Mapping,” of the *OpenGL Programming Guide.*

Why Use the Texture Clamp Extensions?

When a texture coordinate is clamped using the default OpenGL 1.0 algorithm, and a GL_LINEAR filter or one of the LINEAR mipmap filters is used, the texture sampling filter straddles the edge of the texture image, taking half its sample values from within the texture image and the other half from the texture border.

It is sometimes desirable to alter the default behavior of OpenGL texture clamping operations as follows:

- Clamp a texture without requiring a border or a constant border color. This is possible with the texture clamping algorithm provided by the texture edge-clamp extension. GL_CLAMP_TO_EDGE_SGIS clamps texture coordinates at all mipmap levels such that the texture filter never samples a border texel.

  When used with a GL_NEAREST or a GL_LINEAR filter, the color returned when clamping is derived only from texels at the edge of the texture image.
- Clamp a texture to the border color, rather than to an average of the border and edge colors. This is possible with the texture border-clamp extension. GL_CLAMP_TO_BORDER_SGIS clamps texture coordinates at all mipmap levels:

  GL_NEAREST and GL_LINEAR filters return the color of the border texels.

  This mode is well-suited for using projective textures such as spotlights.

Both clamping extensions are supported for one-, two-, and three-dimensional textures.

Using the Texture Clamp Extensions

To specify texture clamping, call **glTexParameteri()**:

- Set *target* to GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D_EXT.
- Set *pname* to GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, or GL_TEXTURE_WRAP_R_EXT.
- Set *param* to
  - GL_CLAMP_TO_EDGE_SGIS for edge clamping
  - GL_CLAMP_TO_BORDER_SGIS for border clamping

The Texture LOD Extension

The texture LOD extension, SGIS_texture_lod, imposes constraints on the texture LOD parameter. Together these constraints allow a large texture to be loaded and used initially at low resolution, and to have its resolution raised gradually as more resolution is desired or available. By providing separate, continuous clamping of the LOD parameter, the extension makes it possible to avoid "popping" artifacts when higher-resolution images are provided.

To achieve this, the extension imposes the following constraints:

- It clamps LOD to a specific floating point range.
- It limits the selection of mipmap image arrays to a subset of the arrays that would otherwise be considered.

To understand the issues discussed in this section, you should be familiar with the issues discussed in the sections "Multiple Levels of Detail" and "Controlling Filtering" in Chapter 9, "Texture Mapping," of the *OpenGL Programming Guide.*

Specifying a Minimum or Maximum Level of Detail

To specify a minimum or maximum level of detail for a specific texture, call **glTexParameter*()** and set

- *target* to GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D_EXT
- *pname* to GL_TEXTURE_MIN_LOD_SGIS or GL_TEXTURE_MAX_LOD_SGIS
- *param* to (or *params* pointing to) the new value

LOD is clamped to the specified range before its use in the texturing process. Whether the minification or magnification filter is used depends on the clamped LOD.

Specifying Image Array Availability

The *OpenGL Specification* describes a "complete" set of mipmap image arrays at levels 0 (zero) through p, where p is a well-defined function of the dimensions of the level 0 image.

This extension lets you redefine any image level as the base level (or maximum level). This is useful, for example, if your application runs under certain time constraints, and you want to make it possible for the application to load as many levels of detail as possible but stop loading and continue processing, choosing from the available ones after a certain period of time has elapsed. Availability in that case does not depend on what's explicitly specified in the program but on what could be loaded in a specified time.

To set a new base (or maximum) level, call **glTexParameteri()**, **glTexParemeterf()**, **glTexParameteriv()**, or **glTexParameterfv()** and set

- *target* to GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D_EXT
- *pname* to
  - GL_TEXTURE_BASE_LEVEL_SGIS to specify a base level
  - GL_TEXTURE_MAX_LEVEL_SGIS to specify a maximum level
- *param* to (or *params* pointing to) the desired value