*Chapter 9*

# Texture Mapping

**Chapter Objectives**

After reading this chapter, you'll be able to do the following:

- Understand what texture mapping can add to your scene
- Specify texture images in compressed and uncompressed formats
- Control how a texture image is filtered as it is applied to a fragment
- Create and manage texture images in texture objects, and control a high-performance working set of those texture objects
- Specify how the texture and fragment colors are combined
- Supply texture coordinates describing how the texture image should be mapped onto objects in your scene
- Generate texture coordinates automatically to produce effects such as contour maps and environment maps
- Perform complex texture operations in a single pass with multitexturing (sequential texture units)
- Use texture combiner functions to mathematically operate on texture, fragment, and constant color values
- After texturing, process fragments with secondary colors
- Specify textures to be used for processing point sprites
- Transform texture coordinates using the texture matrix
- Render shadowed objects, using depth textures

So far, every geometric primitive has been drawn as either a solid color or smoothly shaded between the colors at its vertices—that is, they've been drawn without texture mapping. If you want to draw a large brick wall without texture mapping, for example, each brick must be drawn as a separate polygon. Without texturing, a large flat wall—which is really a single rectangle—might require thousands of individual bricks, and even then the bricks may appear too smooth and regular to be realistic.

Texture mapping allows you to glue an image of a brick wall (obtained, perhaps, by scanning in a photograph of a real wall) to a polygon and to draw the entire wall as a single polygon. Texture mapping ensures that all the right things happen as the polygon is transformed and rendered. For example, when the wall is viewed in perspective, the bricks may appear smaller as the wall gets farther from the viewpoint. Other uses for texture mapping include depicting vegetation on large polygons representing the ground in flight simulation; wallpaper patterns; and textures that make polygons look like natural substances such as marble, wood, and cloth. The possibilities are endless. Although it's most natural to think of applying textures to polygons, textures can be applied to all primitives—points, lines, polygons, bitmaps, and images. Plates 6, 8, 18–21, and 24–32 all demonstrate the use of textures.

Because there are so many possibilities, texture mapping is a fairly large, complex subject, and you must make several programming choices when using it. For starters, most people intuitively understand a two-dimensional texture, but a texture may be one-dimensional or even three-dimensional. You can map textures to surfaces made of a set of polygons or to curved surfaces, and you can repeat a texture in one, two, or three directions (depending on how many dimensions the texture is described in) to cover the surface. In addition, you can automatically map a texture onto an object in such a way that the texture indicates contours or other properties of the item being viewed. Shiny objects can be textured so that they appear to be in the center of a room or other environment, reflecting the surroundings from their surfaces. Finally, a texture can be applied to a surface in different ways. It can be painted on directly (like a decal placed on a surface), used to modulate the color the surface would have been painted otherwise, or used to blend a texture color with the surface color. If this is your first exposure to texture mapping, you might find that the discussion in this chapter moves fairly quickly. As an additional reference, you might look at the chapter on texture mapping in *3D Computer Graphics* by Alan Watt (Addison-Wesley, 1999).

Textures are simply rectangular arrays of data—for example, color data, luminance data, or color and alpha data. The individual values in a texture

array are often called *texels*. What makes texture mapping tricky is that a rectangular texture can be mapped to nonrectangular regions, and this must be done in a reasonable way.

Figure 9-1 illustrates the texture-mapping process. The left side of the figure represents the entire texture, and the black outline represents a quadrilateral shape whose corners are mapped to those spots on the texture. When the quadrilateral is displayed on the screen, it might be distorted by applying various transformations—rotations, translations, scaling, and projections. The right side of the figure shows how the texture-mapped quadrilateral might appear on your screen after these transformations. (Note that this quadrilateral is concave and might not be rendered correctly by OpenGL without prior tessellation. See Chapter 11 for more information about tessellating polygons.)

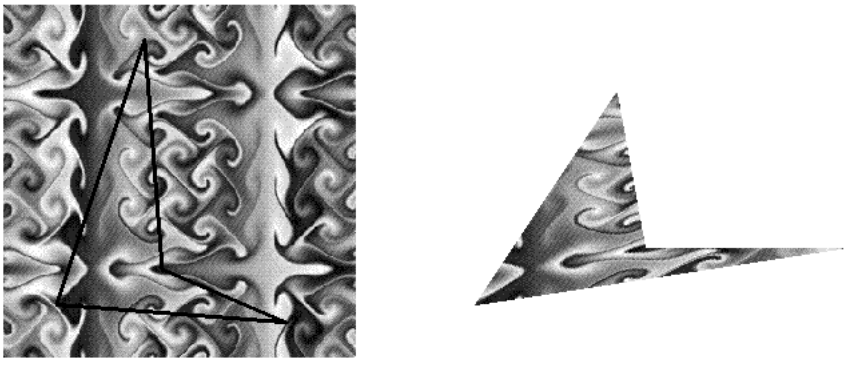


**Figure 9-1** Texture-Mapping Process

Notice how the texture is distorted to match the distortion of the quadrilateral. In this case, it's stretched in the *x*-direction and compressed in the *y*-direction; there's a bit of rotation and shearing going on as well. Depending on the texture size, the quadrilateral's distortion, and the size of the screen image, some of the texels might be mapped to more than one fragment, and some fragments might be covered by multiple texels. Since the texture is made up of discrete texels (in this case, $256 \times 256$ of them), filtering operations must be performed to map texels to fragments. For example, if many texels correspond to a fragment, they're averaged down to fit; if texel boundaries fall across fragment boundaries, a weighted average of the applicable texels is performed. Because of these calculations, texturing is computationally expensive, which is why many specialized graphics systems include hardware support for texture mapping.

An application may establish texture objects, with each texture object representing a single texture (and possible associated mipmaps). Some implementations of OpenGL can support a special *working set* of texture objects that have better performance than texture objects outside the working set. These high-performance texture objects are said to be *resident* and may have special hardware and/or software acceleration available. You may use OpenGL to create and delete texture objects and to determine which textures constitute your working set.

This chapter covers the OpenGL's texture-mapping facility in the following major sections.

- "An Overview and an Example" gives a brief, broad look at the steps required to perform texture mapping. It also presents a relatively simple example of texture mapping.
- "Specifying the Texture" explains how to specify one-, two-, or three-dimensional textures. It also discusses how to use a texture's borders, how to supply a series of related textures of different sizes, and how to control the filtering methods used to determine how an applied texture is mapped to screen coordinates.
- "Filtering" details how textures are either magnified or minified as they are applied to the pixels of polygons. Minification using special mipmap textures is also explained.
- "Texture Objects" describes how to put texture images into objects so that you can control several textures at one time. With texture objects, you may be able to create a working set of high-performance textures, which are said to be resident. You may also prioritize texture objects to increase or decrease the likelihood that a texture object is resident.
- "Texture Functions" discusses the methods used for painting a texture onto a surface. You can choose to have the texture color values replace those that would be used if texturing were not in effect, or you can have the final color be a combination of the two.
- "Assigning Texture Coordinates" describes how to compute and assign appropriate texture coordinates to the vertices of an object. It also explains how to control the behavior of coordinates that lie outside the default range—that is, how to repeat or clamp textures across a surface.
- "Automatic Texture-Coordinate Generation" shows how to have OpenGL automatically generate texture coordinates so that you can achieve such effects as contour and environment maps.
- "Multitexturing" details how textures may be applied in a serial pipeline of successive texturing operations.

- "Texture Combiner Functions" explains how you can control mathematical operations (multiplication, addition, subtraction, interpolation, and even dot products) on the RGB and alpha values of textures, constant colors, and incoming fragments. Combiner functions expose flexible, programmable fragment processing.
- "Applying Secondary Color after Texturing" shows how secondary colors are applied to fragments after texturing.
- "Point Sprites" discusses how textures can be applied to large points to improve their visual quality.
- "The Texture Matrix Stack" explains how to manipulate the texture matrix stack and use the *q* texture coordinate.
- "Depth Textures" describes the process for using the values stored in the depth buffer as a texture for use in determining shadowing for a scene.

Version 1.1 of OpenGL introduced several texture-mapping operations:

- Additional internal texture image formats
- Texture proxy, to query whether there are enough resources to accommodate a given texture image
- Texture subimage, to replace all or part of an existing texture image, rather than completely delete and create a texture to achieve the same effect
- Specifying texture data from framebuffer memory (as well as from system memory)
- Texture objects, including resident textures and prioritizing

Version 1.2 added:

- 3D texture images
- A new texture-coordinate wrapping mode, GL_CLAMP_TO_EDGE, which derives texels from the edge of a texture image, not its border
- Greater control over mipmapped textures to represent different levels of detail (LOD)
- Calculating specular highlights (from lighting) after texturing operations

Version 1.3 granted more texture-mapping operations:

- Compressed textures
- Cube map textures

- Multitexturing, which is applying several textures to render a single primitive
- Texture-wrapping mode, GL_CLAMP_TO_BORDER
- Texture environment modes: GL_ADD and GL_COMBINE (including the dot product combination function)

Version 1.4 supplied these texture capabilities:

- Texture-wrapping mode, GL_MIRRORED_REPEAT
- Automatic mipmap generation with GL_GENERATE_MIPMAP
- Texture parameter GL_TEXTURE_LOD_BIAS, which alters selection of the mipmap level of detail
- Application of a secondary color (specified by **glSecondaryColor*()**) after texturing
- During the texture combine environment mode, the ability to use texture color from different texture units as sources for the texture combine function
- Use of depth (*r* coordinate) as an internal texture format and texturing modes that compare depth texels to decide upon texture application

Version 1.5 added support for:

- Additional texture-comparison modes for use of textures for shadow mapping

Version 2.0 modified texture capabilities by:

- Removing the power-of-two restriction on texture maps
- Iterated texture coordinates across point sprites

Version 2.1 added support for:

- Specifying textures in sRGB format, which accepts gamma-corrected red, green, and blue texture components
- Specifying and retrieving pixel rectangle data in server-side buffer objects. See "Using Buffer Objects with Pixel Rectangle Data" in Chapter 8 for details on using pixel buffer objects.

If you try to use one of these texture-mapping operations and can't find it, check the version number of your implementation of OpenGL to see if it actually supports it. (See "Which Version Am I Using?" in Chapter 14.) In

some implementations, a particular feature may be available only as an extension.

For example, in OpenGL Version 1.2, multitexturing was approved by the OpenGL Architecture Review Board (ARB), the governing body for OpenGL, as an optional extension. An implementation of OpenGL 1.2 supporting multitexturing would have function and constant names suffixed with **ARB**, such as **glActiveTextureARB(GL_TEXTURE1_ARB)**. In OpenGL 1.3, multitexturing became mandatory, and the ARB suffix was removed.

## An Overview and an Example

This section gives an overview of the steps necessary to perform texture mapping. It also presents a relatively simple texture-mapping program. Of course, you know that texture mapping can be a very involved process.

### Steps in Texture Mapping

To use texture mapping, you perform the following steps:

1. Create a texture object and specify a texture for that object.
2. Indicate how the texture is to be applied to each pixel.
3. Enable texture mapping.
4. Draw the scene, supplying both texture and geometric coordinates.

Keep in mind that texture mapping works only in RGBA mode. Texture mapping results in color-index mode are undefined.

#### Create a Texture Object and Specify a Texture for That Object

A texture is usually thought of as being two-dimensional, like most images, but it can also be one-dimensional or three-dimensional. The data describing a texture may consist of one, two, three, or four elements per texel and may represent an (R, G, B, A) quadruple, a modulation constant, or a depth component.

In Example 9-1, which is very simple, a single texture object is created to maintain a single uncompressed, two-dimensional texture. This example does not find out how much memory is available. Since only one texture is

created, there is no attempt to prioritize or otherwise manage a working set of texture objects. Other advanced techniques, such as texture borders, mipmaps, or cube maps, are not used in this simple example.

### Indicate How the Texture Is to Be Applied to Each Pixel

You can choose any of four possible functions for computing the final RGBA value from the fragment color and the texture image data. One possibility is simply to use the texture color as the final color; this is the *replace* mode, in which the texture is painted on top of the fragment, just as a decal would be applied. (Example 9-1 uses *replace* mode.) Another method is to use the texture to *modulate*, or scale, the fragment's color; this technique is useful for combining the effects of lighting with texturing. Finally, a constant color can be blended with that of the fragment, based on the texture value.

### Enable Texture Mapping

You need to enable texturing before drawing your scene. Texturing is enabled or disabled using **glEnable()** or **glDisable()**, with the symbolic constant GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, or GL_TEXTURE_CUBE_MAP for one-, two-, three-dimensional, or cube map texturing, respectively. (If two or all three of the dimensional texturing modes are enabled, the largest dimension enabled is used. If cube map textures are enabled, it trumps all the others. For the sake of clean programs, you should enable only the one you want to use.)

### Draw the Scene, Supplying Both Texture and Geometric Coordinates

You need to indicate how the texture should be aligned relative to the fragments to which it's to be applied before it's "glued on." That is, you need to specify both texture coordinates and geometric coordinates as you specify the objects in your scene. For a two-dimensional texture map, for example, the texture coordinates range from 0.0 to 1.0 in both directions, but the coordinates of the items being textured can be anything. To apply the brick texture to a wall, for example, assuming the wall is square and meant to represent one copy of the texture, the code would probably assign texture coordinates (0, 0), (1, 0), (1, 1), and (0, 1) to the four corners of the wall. If the wall is large, you might want to paint several copies of the texture map on it. If you do so, the texture map must be designed so that the bricks at the left edge match up nicely with the bricks at the right edge, and similarly for the bricks at the top and bottom.

You must also indicate how texture coordinates outside the range [0.0, 1.0] should be treated. Do the textures repeat to cover the object, or are they clamped to a boundary value?

## A Sample Program

One of the problems with showing sample programs to illustrate texture mapping is that interesting textures are large. Typically, textures are read from an image file, since specifying a texture programmatically could take hundreds of lines of code. In Example 9-1, the texture—which consists of alternating white and black squares, like a checkerboard—is generated by the program. The program applies this texture to two squares, which are then rendered in perspective, one of them facing the viewer squarely and the other tilting back at 45 degrees, as shown in Figure 9-2. In object coordinates, both squares are the same size.

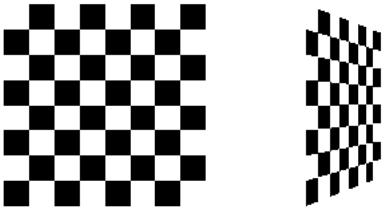


**Figure 9-2** Texture-Mapped Squares

**Example 9-1** Texture-Mapped Checkerboard: checker.c

```
/*  Create checkerboard texture  */
#define checkImageWidth 64
#define checkImageHeight 64
static GLubyte checkImage[checkImageHeight][checkImageWidth][4];

static GLuint texName;

void makeCheckImage(void)
{
   int i, j, c;

   for (i = 0; i < checkImageHeight; i++) {
```

```
      for (j = 0; j < checkImageWidth; j++) {
         c = ((((i&0x8)==0)^((j&0x8))==0))*255;
         checkImage[i][j][0] = (GLubyte) c;
         checkImage[i][j][1] = (GLubyte) c;
         checkImage[i][j][2] = (GLubyte) c;
         checkImage[i][j][3] = (GLubyte) 255;
      }
   }
}

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
   glEnable(GL_DEPTH_TEST);
   makeCheckImage();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

   glGenTextures(1, &texName);
   glBindTexture(GL_TEXTURE_2D, texName);

   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                   GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                   GL_NEAREST);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,
                checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                checkImage);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glEnable(GL_TEXTURE_2D);
   glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
   glBindTexture(GL_TEXTURE_2D, texName);
   glBegin(GL_QUADS);
   glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
   glTexCoord2f(0.0, 1.0); glVertex3f(-2.0, 1.0, 0.0);
   glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 1.0, 0.0);
   glTexCoord2f(1.0, 0.0); glVertex3f(0.0, -1.0, 0.0);
```

```
   glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
   glTexCoord2f(0.0, 1.0); glVertex3f(1.0, 1.0, 0.0);
   glTexCoord2f(1.0, 1.0); glVertex3f(2.41421, 1.0, -1.41421);
   glTexCoord2f(1.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
   glEnd();
   glFlush();
   glDisable(GL_TEXTURE_2D);
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 30.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef(0.0, 0.0, -3.6);
}
/*  keyboard() and main() deleted to reduce printing  */
```

The checkerboard texture is generated in the routine **makeCheckImage()**, and all the texture-mapping initialization occurs in the routine **init()**. **glGenTextures()** and **glBindTexture()** name and create a texture object for a texture image. (See "Texture Objects" on page 414.) The single, full-resolution texture map is specified by **glTexImage2D()**, whose parameters indicate the size, type, location, and other properties of the texture image. (See "Specifying the Texture" below for more information about **glTexImage2D()**.)

The four calls to **glTexParameter*()** specify how the texture is to be wrapped and how the colors are to be filtered if there isn't an exact match between texels in the texture and pixels on the screen. (See "Filtering" on page 411 and "Repeating and Clamping Textures" on page 428.)

In **display()**, **glEnable()** turns on texturing. **glTexEnv*()** sets the drawing mode to GL_REPLACE so that the textured polygons are drawn using the colors from the texture map (rather than taking into account the color in which the polygons would have been drawn without the texture).

Then, two polygons are drawn. Note that texture coordinates are specified along with vertex coordinates. The **glTexCoord*()** command behaves similarly to the **glNormal()** command. **glTexCoord*()** sets the current texture coordinates; any subsequent vertex command has those texture coordinates associated with it until **glTexCoord*()** is called again.

**Note:** The checkerboard image on the tilted polygon might look wrong when you compile and run it on your machine—for example, it might look like two triangles with different projections of the checkerboard image on them. If so, try setting the parameter GL_PERSPECTIVE_CORRECTION_HINT to GL_NICEST and running the example again. To do this, use **glHint()**.

## Specifying the Texture

The command **glTexImage2D()** defines a two-dimensional texture. It takes several arguments, which are described briefly here and in more detail in the subsections that follow. The related commands for one- and three-dimensional textures, **glTexImage1D()** and **glTexImage3D()**, are described in "One-Dimensional Textures" and "Three-Dimensional Textures," respectively.

void **glTexImage2D**(GLenum *target*, GLint *level*, GLint *internalFormat*, GLsizei *width*, GLsizei *height*, GLint *border*, GLenum *format*, GLenum *type*, const GLvoid **texels*);

Defines a two-dimensional texture. The *target* parameter is set to one of the constants: GL_TEXTURE_2D, GL_PROXY_TEXTURE_2D, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, or GL_PROXY_TEXTURE_CUBE_MAP. (See "Cube Map Textures" for information about use of the GL_*CUBE_MAP* constants with **glTexImage2D** and related functions.) You use the *level* parameter if you're supplying multiple resolutions of the texture map; with only one resolution, *level* should be 0. (See "Mipmaps: Multiple Levels of Detail" for more information about using multiple resolutions.)

The next parameter, *internalFormat*, indicates which components (RGBA, depth, luminance, or intensity) are selected for the texels of an image. The value of *internalFormat* is an integer from 1 to 4, or one of the following symbolic constants: GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_COMPRESSED_ALPHA, GL_COMPRESSED_LUMINANCE, GL_COMPRESSED_LUMINANCE_ALPHA, GL_COMPRESSED_INTENSITY, GL_COMPRESSED_RGB,

GL_COMPRESSED_RGBA, GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16, GL_DEPTH_COMPONENT24, GL_DEPTH_COMPONENT32, GL_INTENSITY, GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16, GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_RGB, GL_R3_G3_B2, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12, GL_RGBA16, GL_SRGB, GL_SRGB8, GL_SRGB_ALPHA, GL_SRGB8_ALPHA8, GL_SLUMINANCE_ALPHA, GL_SLUMINANCE8_ALPHA8, GL_SLUMINANCE, GL_SLUMINANCE8, GL_COMPRESSED_SRGB, GL_COMPRESSED_SRGB_ALPHA, GL_COMPRESSED_SLUMINANCE, or GL_COMPRESSED_SLUMINANCE_ALPHA. (See "Texture Functions" for a discussion of how these selected components are applied, and see "Compressed Texture Images" for a discussion of how compressed textures are handled.)

The *internalFormat* may request a specific resolution of components. For example, if *internalFormat* is GL_R3_G3_B2, you are asking that texels be 3 bits of red, 3 bits of green, and 2 bits of blue. But OpenGL is not guaranteed to deliver this; OpenGL is only obligated to choose an internal representation that closely approximates what is requested, but not necessarily an exact match. By definition, GL_INTENSITY, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_DEPTH_COMPONENT, GL_RGB, GL_RGBA, GL_SRGB, GL_SRGB_ALPHA, GL_SLUMINANCE, GL_SLUMINANCE_ALPHA, and the compressed forms of the above tokens are lenient, because they do not ask for a specific resolution. (For compatibility with the OpenGL release 1.0, the numeric values 1, 2, 3, and 4 for *internalFormat* are equivalent to the symbolic constants GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB, and GL_RGBA, respectively.)

The *width* and *height* parameters give the dimensions of the texture image; *border* indicates the width of the border, which is either 0 (no border) or 1. For OpenGL implementations that do not support Version 2.0, both *width* and *height* must have the form $2^m + 2b$, where $m$ is a non-negative integer (which can have a different value for *width* than for *height*) and $b$ is the value of *border*. The maximum size of a texture map depends on the

implementation of OpenGL, but it must be at least 64 × 64 (or 66 × 66 with borders). For OpenGL implementations supporting Version 2.0 and greater, textures may be of any size. The *format* and *type* parameters describe the format and data type of the texture image data. They have the same meaning as they do for **glDrawPixels()**. (See "Imaging Pipeline" in Chapter 8.) In fact, texture data is in the same format as the data used by **glDrawPixels()**, so the settings of **glPixelStore*()** and **glPixelTransfer*()** are applied. (In Example 9-1, the call

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

is made because the data in the example isn't padded at the end of each texel row.) The *format* parameter can be GL_COLOR_INDEX, GL_DEPTH_COMPONENT, GL_RGB, GL_RGBA, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_LUMINANCE, or GL_LUMINANCE_ALPHA—that is, the same formats available for **glDrawPixels()** with the exception of GL_STENCIL_INDEX.

Similarly, the *type* parameter can be GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_BITMAP, or one of the packed pixel data types.

Finally, *texels* contains the texture image data. This data describes the texture image itself as well as its border.

The internal format of a texture image may affect the performance of texture operations. For example, some implementations perform texturing faster with GL_RGBA than with GL_RGB, because the color components align to processor memory better. Since this varies, you should check specific information about your implementation of OpenGL.

The internal format of a texture image also may control how much memory a texture image consumes. For example, a texture of internal format GL_RGBA8 uses 32 bits per texel, while a texture of internal format GL_R3_G3_B2 uses only 8 bits per texel. Of course, there is a corresponding trade-off between memory consumption and color resolution.

A GL_DEPTH_COMPONENT texture stores depth values, as compared to colors, and is most often used for rendering shadows as described in "Depth Textures" on page 459.

Textures specified with an internal format of GL_SRGB, GL_SRGB8, GL_SRGB_ALPHA, GL_SRGB8_ALPHA8, GL_SLUMINANCE_ALPHA,

GL_SLUMINANCE8_ALPHA8, GL_SLUMINANCE, GL_SLUMINANCE8, GL_COMPRESSED_SRGB, GL_COMPRESSED_SRGB_ALPHA, GL_COMPRESSED_SLUMINANCE, or GL_COMPRESSED_SLUMINANCE_ALPHA) are expected to have their red, green, and blue color components specified in the sRGB color space (officially known as the International Electrotechnical Commission IEC standard 61966-2-1). The sRGB color space is approximately the same as the 2.2 gamma-corrected linear RGB color space. For sRGB textures, the alpha values in the texture should not be gamma corrected.

Although texture mapping results in color-index mode are undefined, you can still specify a texture with a GL_COLOR_INDEX image. In that case, pixel-transfer operations are applied to convert the indices to RGBA values by table lookup before they're used to form the texture image.

If your OpenGL implementation supports the *Imaging Subset* and any of its features are enabled, the texture image will be affected by those features. For example, if the two-dimensional convolution filter is enabled, then the convolution will be performed on the texture image. (The convolution may change the image's width and/or height.)

The number of texels for both the width and height of a texture image, not including the optional border, must be a power of 2. If your original image does not have dimensions that fit that limitation, you can use the OpenGL Utility Library routine **gluScaleImage()** to alter the sizes of your textures.

int **gluScaleImage**(GLenum *format*, GLint *widthin*, GLint *heightin*, GLenum *typein*, const void **datain*, GLint *widthout*, GLint *heightout*, GLenum *typeout*, void **dataout*);

Scales an image using the appropriate pixel-storage modes to unpack the data from *datain*. The *format*, *typein*, and *typeout* parameters can refer to any of the formats or data types supported by **glDrawPixels()**. The image is scaled using linear interpolation and box filtering (from the size indicated by *widthin* and *heightin* to *widthout* and *heightout*), and the resulting image is written to *dataout*, using the pixel GL_PACK* storage modes. The caller of **gluScaleImage()** must allocate sufficient space for the output buffer. A value of 0 is returned on success, and a GLU error code is returned on failure.

**Note:** In GLU 1.3, **gluScaleImage()** supports packed pixel formats (and their related data types).

The framebuffer itself can also be used as a source for texture data. **glCopyTexImage2D()** reads a rectangle of pixels from the framebuffer and uses that rectangle as texels for a new texture.

void **glCopyTexImage2D**(GLenum *target*, GLint *level*,
GLint *internalFormat*,
GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*,
GLint *border*);

Creates a two-dimensional texture, using framebuffer data to define the texels. The pixels are read from the current GL_READ_BUFFER and are processed exactly as if **glCopyPixels()** had been called, but instead of going to the framebuffer, the pixels are placed into texture memory. The settings of **glPixelTransfer*()** and other pixel-transfer operations are applied.

The *target* parameter must be one of the constants GL_TEXTURE_2D, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, or GL_TEXTURE_CUBE_MAP_NEGATIVE_Z. (See "Cube Map Textures" on page 441 for information about use of the *CUBE_MAP* constants.) The *level*, *internalFormat*, and *border* parameters have the same effects that they have for **glTexImage2D()**. The texture array is taken from a screen-aligned pixel rectangle with the lower left corner at coordinates specified by the (*x*, *y*) parameters. The *width* and *height* parameters specify the size of this pixel rectangle. For OpenGL implementations that do not support Version 2.0, both *width* and *height* must have the form $2^m+2b$, where $m$ is a non-negative integer (which can have a different value for *width* than for *height*) and $b$ is the value of *border*. For implementations supporting OpenGL Version 2.0 and greater, textures may be of any size.

The next sections give more detail about texturing, including the use of the *target*, *border*, and *level* parameters. The *target* parameter can be used to query accurately the size of a texture (by creating a texture proxy with **glTexImage*D()**) and whether a texture possibly can be used within the texture resources of an OpenGL implementation. Redefining a portion of a texture is described in "Replacing All or Part of a Texture Image" on page 387. One- and three-dimensional textures are discussed in "One-Dimensional Textures" on page 390 and "Three-Dimensional Textures" on page 392, respectively. The texture border, which has its size controlled by the *border* parameter, is detailed in "Compressed Texture Images" on page 397. The *level* parameter is used to specify textures of different

resolutions and is incorporated into the special technique of *mipmapping*, which is explained in "Mipmaps: Multiple Levels of Detail" on page 400. Mipmapping requires understanding how to filter textures as they're applied; filtering is covered on page 411.

## Texture Proxy

To an OpenGL programmer who uses textures, size is important. Texture resources are typically limited, and texture format restrictions vary among OpenGL implementations. There is a special texture proxy target to evaluate whether your OpenGL implementation is capable of supporting a particular texture format at a particular texture size.

**glGetIntegerv**(GL_MAX_TEXTURE_SIZE,...) tells you a lower bound on the largest width or height (without borders) of a texture image; typically, the size of the largest square texture supported. For 3D textures, GL_MAX_3D_TEXTURE_SIZE may be used to query the largest allowable dimension (width, height, or depth, without borders) of a 3D texture image. For cube map textures, GL_MAX_CUBE_MAP_TEXTURE_SIZE is similarly used.

However, use of any of the GL_MAX*TEXTURE_SIZE queries does not consider the effect of the internal format or other factors. A texture image that stores texels using the GL_RGBA16 internal format may be using 64 bits per texel, so its image may have to be 16 times smaller than an image with the GL_LUMINANCE4 internal format. Textures requiring borders or mipmaps further reduce the amount of available memory.

A special placeholder, or *proxy*, for a texture image allows the program to query more accurately whether OpenGL can accommodate a texture of a desired internal format.

For instance, to find out whether there are enough resources available for a standard 2D texture, call **glTexImage2D()** with a *target* parameter of GL_PROXY_TEXTURE_2D and the given *level, internalFormat, width, height, border, format,* and *type*. For a proxy, you should pass NULL as the pointer for the *texels* array. (For a cube map, use **glTexImage2D()** with the target GL_PROXY_TEXTURE_CUBE_MAP. For one- or three-dimensional textures, use corresponding 1D or 3D routines and symbolic constants.)

After the texture proxy has been created, query the texture state variables with **glGetTexLevelParameter*()**. If there aren't enough resources to accommodate the texture proxy, the texture state variables for width, height, border width, and component resolutions are set to 0.

void **glGetTexLevelParameter**{if}**v**(GLenum *target*, GLint *level*, GLenum *pname*, *TYPE* **params*);

Returns in *params* texture parameter values for a specific level of detail, specified as *level*. *target* defines the target texture and is GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, GL_PROXY_TEXTURE_1D, GL_PROXY_TEXTURE_2D, GL_PROXY_TEXTURE_3D, or GL_PROXY_TEXTURE_CUBE_MAP. (GL_TEXTURE_CUBE_MAP is not valid, because it does not specify a particular face of a cube map.) Accepted values for *pname* are GL_TEXTURE_WIDTH, GL_TEXTURE_HEIGHT, GL_TEXTURE_DEPTH, GL_TEXTURE_BORDER, GL_TEXTURE_INTERNAL_FORMAT, GL_TEXTURE_RED_SIZE, GL_TEXTURE_GREEN_SIZE, GL_TEXTURE_BLUE_SIZE, GL_TEXTURE_ALPHA_SIZE, GL_TEXTURE_LUMINANCE_SIZE, and GL_TEXTURE_INTENSITY_SIZE.

Example 9-2 demonstrates how to use the texture proxy to find out if there are enough resources to create a $64 \times 64$ texel texture with RGBA components with 8 bits of resolution. If this succeeds, then **glGetTexLevelParameteriv()** stores the internal format (in this case, GL_RGBA8) into the variable *format*.

**Example 9-2** Querying Texture Resources with a Texture Proxy

```
GLint width;

glTexImage2D(GL_PROXY_TEXTURE_2D, 0, GL_RGBA8,
             64, 64, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D, 0,
                         GL_TEXTURE_WIDTH, &width);
```

**Note:** There is one major limitation with texture proxies: the texture proxy answers the question of whether a texture is capable of being loaded into texture memory. The texture proxy provides the same answer, regardless of how texture resources are currently being used. If other textures are using resources, then the texture proxy query may respond affirmatively, but there may not be enough resources to make your texture resident (that is, part of a possibly high-performance working set of textures). The texture proxy does not answer the question of whether there is sufficient capacity to handle the requested texture. (See "Texture Objects" for more information about managing resident textures.)

## Replacing All or Part of a Texture Image

Creating a texture may be more computationally expensive than modifying an existing one. Often it is better to replace all or part of a texture image with new information, rather than create a new one. This can be helpful for certain applications, such as using real-time, captured video images as texture images. For that application, it makes sense to create a single texture and use **glTexSubImage2D()** to replace repeatedly the texture data with new video images. Also, there are no size restrictions for **glTexSubImage2D()** that force the height or width to be a power of 2. (This is helpful for processing video images, which generally do not have sizes that are powers of 2. However, you must load the video images into an initial, larger image that must have $2^n$ texels for each dimension, and adjust texture coordinates for the subimages.)

void **glTexSubImage2D**(GLenum *target*, GLint *level*, GLint *xoffset*, GLint *yoffset*, GLsizei *width*, GLsizei *height*, GLenum *format*, GLenum *type*, const GLvoid **texels*);

Defines a two-dimensional texture image that replaces all or part of a contiguous subregion (in 2D, it's simply a rectangle) of the current, existing two-dimensional texture image. The *target* parameter must be set to one of the same options that are available for **glCopyTexImage2D**.

The *level*, *format,* and *type* parameters are similar to the ones used for **glTexImage2D()**. *level* is the mipmap level-of-detail number. It is not an error to specify a width or height of 0, but the subimage will have no effect. *format* and *type* describe the format and data type of the texture image data. The subimage is also affected by modes set by **glPixelStore*()** and **glPixelTransfer*()** and other pixel-transfer operations.

*texels* contains the texture data for the subimage. *width* and *height* are the dimensions of the subregion that is replacing all or part of the current texture image. *xoffset* and *yoffset* specify the texel offset in the *x*- and *y*-directions—with (0, 0) at the lower left corner of the texture—and specify where in the existing texture array the subimage should be placed. This region may not include any texels outside the range of the originally defined texture array.

In Example 9-3, some of the code from Example 9-1 has been modified so that pressing the 's' key drops a smaller checkered subimage into the existing image. (The resulting texture is shown in Figure 9-3.) Pressing the 'r' key restores the original image. Example 9-3 shows the two routines,

**makeCheckImages()** and **keyboard()**, that have been substantially changed. (See "Texture Objects" for more information about **glBindTexture()**.)

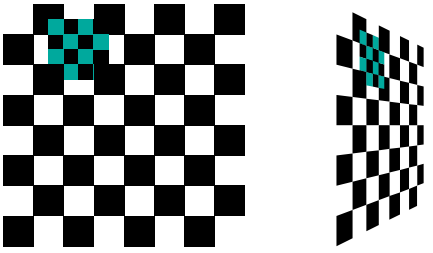


**Figure 9-3** Texture with Subimage Added

**Example 9-3** Replacing a Texture Subimage: texsub.c

```
/*  Create checkerboard textures  */
#define checkImageWidth 64
#define checkImageHeight 64
#define subImageWidth 16
#define subImageHeight 16
static GLubyte checkImage[checkImageHeight][checkImageWidth][4];
static GLubyte subImage[subImageHeight][subImageWidth][4];

void makeCheckImages(void)
{
   int i, j, c;

   for (i = 0; i < checkImageHeight; i++) {
      for (j = 0; j < checkImageWidth; j++) {
         c = (((i&0x8)==0) ^ ((j&0x8)==0))*255;
         checkImage[i][j][0] = (GLubyte) c;
         checkImage[i][j][1] = (GLubyte) c;
         checkImage[i][j][2] = (GLubyte) c;
         checkImage[i][j][3] = (GLubyte) 255;
      }
   }
   for (i = 0; i < subImageHeight; i++) {
      for (j = 0; j < subImageWidth; j++) {
         c = (((i&0x4)==0) ^ ((j&0x4)==0))*255;
         subImage[i][j][0] = (GLubyte) c;
         subImage[i][j][1] = (GLubyte) 0;
         subImage[i][j][2] = (GLubyte) 0;
         subImage[i][j][3] = (GLubyte) 255;
      }
   }
```

```
}

void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 's':
      case 'S':
         glBindTexture(GL_TEXTURE_2D, texName);
         glTexSubImage2D(GL_TEXTURE_2D, 0, 12, 44,
                         subImageWidth, subImageHeight, GL_RGBA,
                         GL_UNSIGNED_BYTE, subImage);
         glutPostRedisplay();
         break;
      case 'r':
      case 'R':
         glBindTexture(GL_TEXTURE_2D, texName);
         glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA,
                      checkImageWidth, checkImageHeight, 0,
                      GL_RGBA, GL_UNSIGNED_BYTE, checkImage);
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
      default:
         break;
   }
}
```

Once again, the framebuffer itself can be used as a source for texture data—this time, a texture subimage. **glCopyTexSubImage2D()** reads a rectangle of pixels from the framebuffer and replaces a portion of an existing texture array. (**glCopyTexSubImage2D()** is something of a cross between **glCopyTexImage2D()** and **glTexSubImage2D()**.)

void **glCopyTexSubImage2D**(GLenum *target*, GLint *level*, GLint *xoffset*, GLint *yoffset*, GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*);

Uses image data from the framebuffer to replace all or part of a contiguous subregion of the current, existing two-dimensional texture image. The pixels are read from the current GL_READ_BUFFER and are processed exactly as if **glCopyPixels()** had been called, but instead of going to the framebuffer, the pixels are placed into texture memory. The settings of **glPixelTransfer*()** and other pixel-transfer operations are applied.

The *target* parameter must be set to one of the same options that are available for **glCopyTexImage2D**. *level* is the mipmap level-of-detail number. *xoffset* and *yoffset* specify the texel offset in the *x*- and *y*-directions—with (0, 0) at the lower left corner of the texture—and specify where in the existing texture array the subimage should be placed. The subimage texture array is taken from a screen-aligned pixel rectangle with the lower left corner at coordinates specified by the (*x*, *y*) parameters. The *width* and *height* parameters specify the size of this subimage rectangle.

## One-Dimensional Textures

Sometimes a one-dimensional texture is sufficient—for example, if you're drawing textured bands where all the variation is in one direction. A one-dimensional texture behaves as a two-dimensional one with *height* = 1, and without borders along the top and bottom. All the two-dimensional texture and subtexture definition routines have corresponding one-dimensional routines. To create a simple one-dimensional texture, use **glTexImage1D()**.

void **glTexImage1D**(GLenum *target*, GLint *level*, GLint *internalFormat*, GLsizei *width*, GLint *border*, GLenum *format*, GLenum *type*, const GLvoid **texels*);

Defines a one-dimensional texture. All the parameters have the same meanings as for **glTexImage2D()**, except that *texels* is now a one-dimensional array. As before, for OpenGL implementations that do not support OpenGL Version 2.0 or greater, the value of *width* is $2^m$ (or $2^m + 2$, if there's a border), where *m* is a non-negative integer. You can supply mipmaps and proxies (set *target* to GL_PROXY_TEXTURE_1D), and the same filtering options are available as well.

For a sample program that uses a one-dimensional texture map, see Example 9-8.

If your OpenGL implementation supports the *Imaging Subset* and if the one-dimensional convolution filter is enabled (GL_CONVOLUTION_1D), then the convolution is performed on the texture image. (The convolution may change the width of the texture image.) Other pixel operations may also be applied.

To replace all or some of the texels of a one-dimensional texture, use **glTexSubImage1D()**.

void **glTexSubImage1D**(GLenum *target*, GLint *level*, GLint *xoffset*,
GLsizei *width*, GLenum *format*,
GLenum *type*, const GLvoid ***texels*);

Defines a one-dimensional texture array that replaces all or part of a contiguous subregion (in 1D, a row) of the current, existing one-dimensional texture image. The *target* parameter must be set to GL_TEXTURE_1D.

The *level*, *format,* and *type* parameters are similar to the ones used for **glTexImage1D()**. *level* is the mipmap level-of-detail number. *format* and *type* describe the format and data type of the texture image data. The subimage is also affected by modes set by **glPixelStore*()**, **glPixelTransfer*()**, or other pixel-transfer operations.

*texels* contains the texture data for the subimage. *width* is the number of texels that replace part or all of the current texture image. *xoffset* specifies the texel offset in the existing texture array where the subimage should be placed.

To use the framebuffer as the source of a new one-dimensional texture or a replacement for an old one-dimensional texture, use either **glCopyTexImage1D()** or **glCopyTexSubImage1D()**.

void **glCopyTexImage1D**(GLenum *target*, GLint *level*,
GLint *internalFormat*, GLivnt *x*, GLint *y*,
GLsizei *width*, GLint *border*);

Creates a one-dimensional texture using framebuffer data to define the texels. The pixels are read from the current GL_READ_BUFFER and are processed exactly as if **glCopyPixels()** had been called, but instead of going to the framebuffer, the pixels are placed into texture memory. The settings of **glPixelStore*()** and **glPixelTransfer*()** are applied.

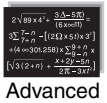
The *target* parameter must be set to the constant GL_TEXTURE_1D. The *level*, *internalFormat*, and *border* parameters have the same effects that they have for **glCopyTexImage2D()**. The texture array is taken from a row of pixels with the lower left corner at coordinates specified by the (*x*, *y*) parameters. The *width* parameter specifies the number of pixels in this row. For OpenGL implementations that do not support Version 2.0, the value of *width* is $2^m$ (or $2^m + 2$ if there's a border), where *m* is a non-negative integer.

void **glCopyTexSubImage1D**(GLenum *target*, GLint *level*, GLint *xoffset*, GLint *x*, GLint *y*, GLsizei *width*);

Uses image data from the framebuffer to replace all or part of a contiguous subregion of the current, existing one-dimensional texture image. The pixels are read from the current GL_READ_BUFFER and are processed exactly as if **glCopyPixels()** had been called, but instead of going to the framebuffer, the pixels are placed into texture memory. The settings of **glPixelTransfer*()** and other pixel-transfer operations are applied.

The *target* parameter must be set to GL_TEXTURE_1D. *level* is the mipmap level-of-detail number. *xoffset* specifies the texel offset and where to put the subimage within the existing texture array. The subimage texture array is taken from a row of pixels with the lower left corner at coordinates specified by the (*x*, *y*) parameters. The *width* parameter specifies the number of pixels in this row.

## Three-Dimensional Textures


Advanced

**Advanced**

Three-dimensional textures are most often used for rendering in medical and geoscience applications. In a medical application, a three-dimensional texture may represent a series of layered computed tomography (CT) or magnetic resonance imaging (MRI) images. To an oil and gas researcher, a three-dimensional texture may model rock strata. (Three-dimensional texturing is part of an overall category of applications, called *volume rendering*. Some advanced volume rendering applications deal with *voxels*, which represent data as volume-based entities.)

Due to their size, three-dimensional textures may consume a lot of texture resources. Even a relatively coarse three-dimensional texture may use 16 or 32 times the amount of texture memory that a single two-dimensional texture uses. (Most of the two-dimensional texture and subtexture definition routines have corresponding three-dimensional routines.)

A three-dimensional texture image can be thought of as layers of two-dimensional subimage rectangles. In memory, the rectangles are arranged in a sequence. To create a simple three-dimensional texture, use **glTexImage3D()**.

**Note:** There are no three-dimensional convolutions in the *Imaging Subset*. However, 2D convolution filters may be used to affect three-dimensional texture images.

void **glTexImage3D**(GLenum *target*, GLint *level*, GLint *internalFormat*, GLsizei *width*, GLsizei *height*, GLsizei *depth*, GLint *border*, GLenum *format*, GLenum *type*, const GLvoid ***texels*);

Defines a three-dimensional texture. All the parameters have the same meanings as for **glTexImage2D()**, except that *texels* is now a three-dimensional array, and the parameter *depth* has been added. Likewise, if the OpenGL implementation does not support Version 2.0, the value of *depth* is $2^m$ (or $2^m + 2$, if there's a border), where *m* is a non-negative integer. For OpenGL 2.0 implementations, the power-of-two dimension requirement has been eliminated. You can supply mipmaps and proxies (set *target* to GL_PROXY_TEXTURE_3D), and the same filtering options are available as well.

For a portion of a program that uses a three-dimensional texture map, see Example 9-4.

**Example 9-4** Three-Dimensional Texturing: texture3d.c

```
#define iWidth 16
#define iHeight 16
#define iDepth 16

static GLubyte image [iDepth][iHeight][iWidth][3];
static GLuint texName;

/*  Create a 16x16x16x3 array with different color values in
 *  each array element [r, g, b]. Values range from 0 to 255.
 */
void makeImage(void)
{
   int s, t, r;

   for (s = 0 ; s < 16 ; s++)
      for (t = 0 ; t < 16 ; t++)
         for (r = 0 ; r < 16 ; r++) {
            image[r][t][s][0] = s * 17;
            image[r][t][s][1] = t * 17;
            image[r][t][s][2] = r * 17;
         }
}

/*  Initialize state: the 3D texture object and its image
 */
void init(void)
{
```

```
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_FLAT);
    glEnable(GL_DEPTH_TEST);

    makeImage();
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    glGenTextures(1, &texName);
    glBindTexture(GL_TEXTURE_3D, texName);

    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_CLAMP);

    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST);
    glTexImage3D(GL_TEXTURE_3D, 0, GL_RGB, iWidth, iHeight,
                 iDepth, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
}
```

To replace all or some of the texels of a three-dimensional texture, use **glTexSubImage3D()**.

void **glTexSubImage3D**(GLenum *target*, GLint *level*, GLint *xoffset*, GLint *yoffset*, GLint *zoffset*, GLsizei *width*, GLsizei *height*, GLsizei *depth*, GLenum *format*, GLenum *type*, const GLvoid **texels*);

Defines a three-dimensional texture array that replaces all or part of a contiguous subregion of the current, existing three-dimensional texture image. The *target* parameter must be set to GL_TEXTURE_3D.

The *level*, *format,* and *type* parameters are similar to the ones used for **glTexImage3D()**. *level* is the mipmap level-of-detail number. *format* and *type* describe the format and data type of the texture image data. The subimage is also affected by modes set by **glPixelStore*()**, **glPixelTransfer*()**, and other pixel-transfer operations.

*texels* contains the texture data for the subimage. *width, height,* and *depth* specify the size of the subimage in texels. *xoffset, yoffset,* and *zoffset* specify the texel offset indicating where to put the subimage within the existing texture array.

To use the framebuffer as the source of replacement for a portion of an existing three-dimensional texture, use **glCopyTexSubImage3D()**.

void **glCopyTexSubImage3D**(GLenum *target*, GLint *level*, GLint *xoffset*, GLint *yoffset*, GLint *zoffset*, GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*);

Uses image data from the framebuffer to replace part of a contiguous subregion of the current, existing three-dimensional texture image. The pixels are read from the current GL_READ_BUFFER and are processed exactly as if **glCopyPixels()** had been called, but instead of going to the framebuffer, the pixels are placed into texture memory. The settings of **glPixelTransfer*()** and other pixel-transfer operations are applied.

The *target* parameter must be set to GL_TEXTURE_3D. *level* is the mipmap level-of-detail number. The subimage texture array is taken from a screen-aligned pixel rectangle with the lower left corner at coordinates specified by the (*x*, *y*) parameters. The *width* and *height* parameters specify the size of this subimage rectangle. *xoffset, yoffset,* and *zoffset* specify the texel offset indicating where to put the subimage within the existing texture array. Since the subimage is a two-dimensional rectangle, only a single slice of the three-dimensional texture (the slice at *zoffset*) is replaced.

### Pixel-Storage Modes for Three-Dimensional Textures

Pixel-storage values control the row-to-row spacing of each layer (in other words, of one 2D rectangle). **glPixelStore*()** sets pixel-storage modes, with parameters such as *ROW_LENGTH, *ALIGNMENT, *SKIP_PIXELS, and *SKIP_ROWS (where * is either GL_UNPACK_ or GL_PACK_), which control referencing of a subrectangle of an entire rectangle of pixel or texel data. (These modes were previously described in "Controlling Pixel-Storage Modes" on page 325.)

The aforementioned pixel-storage modes remain useful for describing two of the three dimensions, but additional pixel-storage modes are needed to support referencing of subvolumes of three-dimensional texture image data. New parameters, *IMAGE_HEIGHT and *SKIP_IMAGES, allow the routines **glTexImage3D()**, **glTexSubImage3D()**, and **glGetTexImage()** to delimit and access any desired subvolume.

If the three-dimensional texture in memory is larger than the subvolume that is defined, you need to specify the height of a single subimage with the *IMAGE_HEIGHT parameter. Also, if the subvolume does not start with the very first layer, the *SKIP_IMAGES parameter needs to be set.

*IMAGE_HEIGHT is a pixel-storage parameter that defines the height (number of rows) of a single layer of a three-dimensional texture image. If the *IMAGE_HEIGHT value is zero (a negative number is invalid), then the number of rows in each two-dimensional rectangle is the value of *height*, which is the parameter passed to **glTexImage3D()** or **glTexSubImage3D()**. (This is commonplace because *IMAGE_HEIGHT is zero, by default.) Otherwise, the height of a single layer is the *IMAGE_HEIGHT value.

Figure 9-4 shows how *IMAGE_HEIGHT determines the height of an image (when the parameter *height* determines only the height of the subimage.) This figure shows a three-dimensional texture with only two layers.

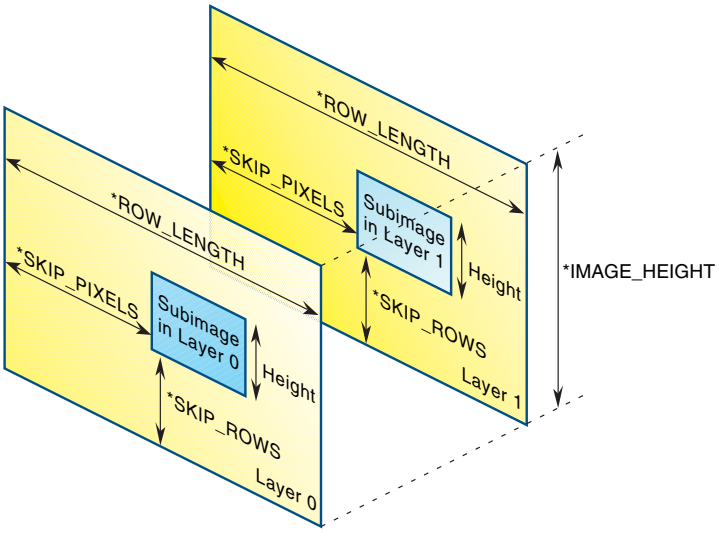



**Figure 9-4** *IMAGE_HEIGHT Pixel-Storage Mode

*SKIP_IMAGES defines how many layers to bypass before accessing the first data of the subvolume. If the *SKIP_IMAGES value is a positive integer (call the value *n*), then the pointer in the texture image data is advanced that many layers (*n* * the size of one layer of texels). The resulting subvolume starts at layer *n* and is several layers deep—how many layers deep is determined by the *depth* parameter passed to **glTexImage3D()** or **glTexSubImage3D()**. If the *SKIP_IMAGES value is zero (the default), then accessing the texel data begins with the very first layer described in the texel array.

Figure 9-5 shows how the *SKIP_IMAGES parameter can bypass several layers to get to where the subvolume is actually located. In this example, *SKIP_IMAGES == 3, and the subvolume begins at layer 3.

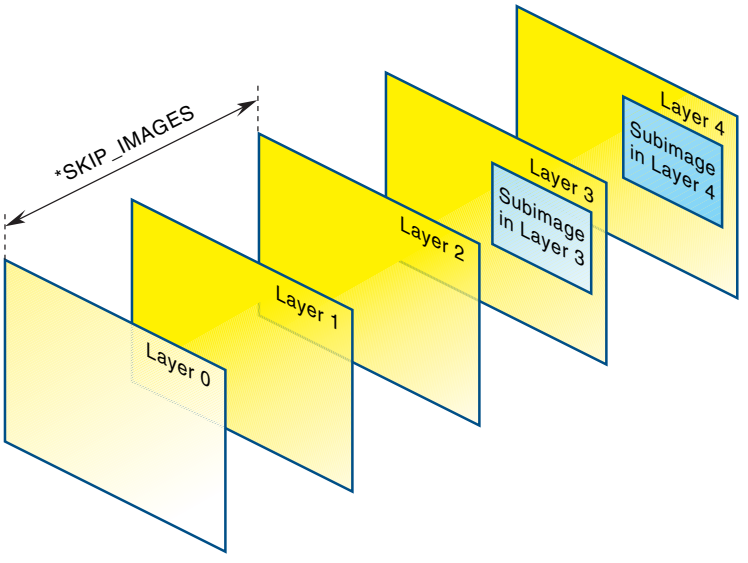



**Figure 9-5** *SKIP_IMAGES Pixel-Storage Mode

## Compressed Texture Images

Texture maps can be stored internally in a compressed format to possibly reduce the amount of texture memory used. A texture image can either be compressed as it is being loaded or loaded directly in its compressed form.

### Compressing a Texture Image While Loading

To have OpenGL compress a texture image while it's being downloaded, specify one of the GL_COMPRESSED_* enumerants for the *internalformat* parameter. The image will automatically be compressed after the texels have been processed by any active pixel-store (See "Controlling Pixel-Storage Modes") or pixel-transfer modes (See "Pixel-Transfer Operations").

Once the image has been loaded, you can determine if it was compressed, and into which format, using the following:

```
GLboolean compressed;
GLenum    textureFormat;
GLsizei   imageSize;
```

```
glGetTexLevelParameteriv(GL_TEXTURE_2D, GL_TEXTURE_COMPRESSED,
    &compressed);
if (compressed == GL_TRUE) {
    glGetTexLevelParameteriv(GL_TEXTURE_2D,
        GL_TEXTURE_INTERNAL_FORMAT, &textureFormat);
    glGetTexLevelParameteriv(GL_TEXTURE_2D,
        GL_TEXTURE_COMPRESSED_IMAGE_SIZE, &imageSize);
}
```

### Loading a Compressed Texture Images

OpenGL doesn't specify the internal format that should be used for compressed textures; each OpenGL implementation is allowed to specify a set of OpenGL extensions that implement a particular texture compression format. For compressed textures that are to be loaded directly, it's important to know their storage format and to verify that the texture's format is available in your OpenGL implementation.

To load a texture stored in a compressed format, use the **glCompressedTexImage*D()** calls.

void **glCompressedTexImage1D**(GLenum *target*, GLint *level*,
GLenum *internalformat*, GLsizei *width*,
GLint *border*, GLsizei *imageSize*,
const GLvoid **texels*);
void **glCompressedTexImage2D**(GLenum *target*, GLint *level*,
GLenum *internalformat*, GLsizei *width*,
GLsizei *height*, GLint *border*,
GLsizei *imageSize*, const GLvoid **texels*);
void **glCompressedTexImage3D**(GLenum *target*, GLint *level*,
GLenum *internalformat*, GLsizei *width*,
GLsizei *height*, GLsizei *depth*,
GLint *border*, GLsizei *imageSize*,
const GLvoid **texels*);

Defines a one-, two-, or three-dimensional texture from a previously compressed texture image.

Use the *level* parameter if you're supplying multiple resolutions of the texture map; with only one resolution, *level* should be 0. (See "Mipmaps: Multiple Levels of Detail" for more information about using multiple resolutions.)

*internalformat* specifies the format of the compressed texture image. It must be a supported compression format of the implementation loading the texture, otherwise a GL_INVALID_ENUM error is specified. To determine supported compressed texture formats, see Appendix B for details.

*width*, *height*, and *depth* represent the dimensions of the texture image for one-, two-, and three-dimensional texture images, respectively. As with uncompressed textures, *border* indicates the width of the border, which is either 0 (no border) or 1. Each value must have the form $2^m + 2b$, where $m$ is a non-negative integer and $b$ is the value of *border*. For OpenGL 2.0 implementations, the power-of-two dimension requirement has been eliminated.

Additionally, compressed textures can be used, just like uncompressed texture images, to replace all or part of an already loaded texture. Use the **glCompressedTexSubImage*D()** calls.

void **glCompressedTexSubImage1D**(GLenum *target*, GLint *level*,
GLint *xoffset*, GLsizei *width*,
GLenum *format*, GLsizei *imageSize*,
const GLvoid **texels*);
void **glCompressedTexSubImage2D**(GLenum *target*, GLint *level*,
GLint *xoffset*, GLint *yoffet*,
GLsizei *width*, GLsizei *height*,
GLsizei *imageSize*,
const GLvoid **texels*);
void **glCompressedTexSubImage3D**(GLenum *target*, GLint *level*,
GLint *xoffset* GLint *yoffset*,
GLint *zoffset*, GLsizei *width*,
GLsizei *height*, GLsizei *depth*,
GLsizei *imageSize*,
const GLvoid **texels*);

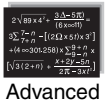
Defines a one-, two-, or three-dimensional texture from a previously compressed texture image.

The *xoffset*, *yoffset*, and *zoffset* parameters specify the pixel offsets for the respective texture dimension where to place the new image inside of the texture array.

*width*, *height*, and *depth* specify the size of the one-, two-, or three-dimensional texture image to be used to update the texture image.

*imageSize* specifies the number of bytes stored in the *texels* array.

## Using a Texture's Borders


Advanced

**Advanced**

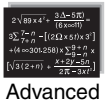
If you need to apply a larger texture map than your implementation of OpenGL allows, you can, with a little care, effectively make larger textures by tiling with several different textures. For example, if you need a texture twice as large as the maximum allowed size mapped to a square, draw the square as four subsquares, and load a different texture before drawing each piece.

Since only a single texture map is available at one time, this approach might lead to problems at the edges of the textures, especially if some form of linear filtering is enabled. The texture value to be used for pixels at the edges must be averaged with something beyond the edge, which, ideally, should come from the adjacent texture map. If you define a border for each texture whose texel values are equal to the values of the texels at the edge of the adjacent texture map, then the correct behavior results when linear filtering takes place.

To do this correctly, notice that each map can have eight neighbors—one adjacent to each edge and one touching each corner. The values of the texels in the corner of the border need to correspond with the texels in the texture maps that touch the corners. If your texture is an edge or corner of the whole tiling, you need to decide what values would be reasonable to put in the borders. The easiest reasonable thing to do is to copy the value of the adjacent texel in the texture map with **glTexSubImage2D()**.

A texture's border color is also used if the texture is applied in such a way that it only partially covers a primitive. (See "Repeating and Clamping Textures" on page 428 for more information about this situation.)

## Mipmaps: Multiple Levels of Detail


Advanced

**Advanced**

Textured objects can be viewed, like any other objects in a scene, at different distances from the viewpoint. In a dynamic scene, as a textured object moves farther from the viewpoint, the texture map must decrease in size along with the size of the projected image. To accomplish this, OpenGL has to filter the texture map down to an appropriate size for mapping onto the object, without introducing visually disturbing artifacts, such as shimmering, flashing, and scintillation. For example, to render a brick wall, you may

use a large texture image (say 128 × 128 texels) when the wall is close to the viewer. But if the wall is moved farther away from the viewer until it appears on the screen as a single pixel, then the filtered textures may appear to change abruptly at certain transition points.

To avoid such artifacts, you can specify a series of prefiltered texture maps of decreasing resolutions, called *mipmaps*, as shown in Figure 9-6. The term *mipmap* was coined by Lance Williams, when he introduced the idea in his paper "Pyramidal Parametrics" (*SIGGRAPH 1983 Proceedings*). *Mip* stands for the Latin *multum in parvo*, meaning "many things in a small place." Mipmapping uses some clever methods to pack image data into memory.

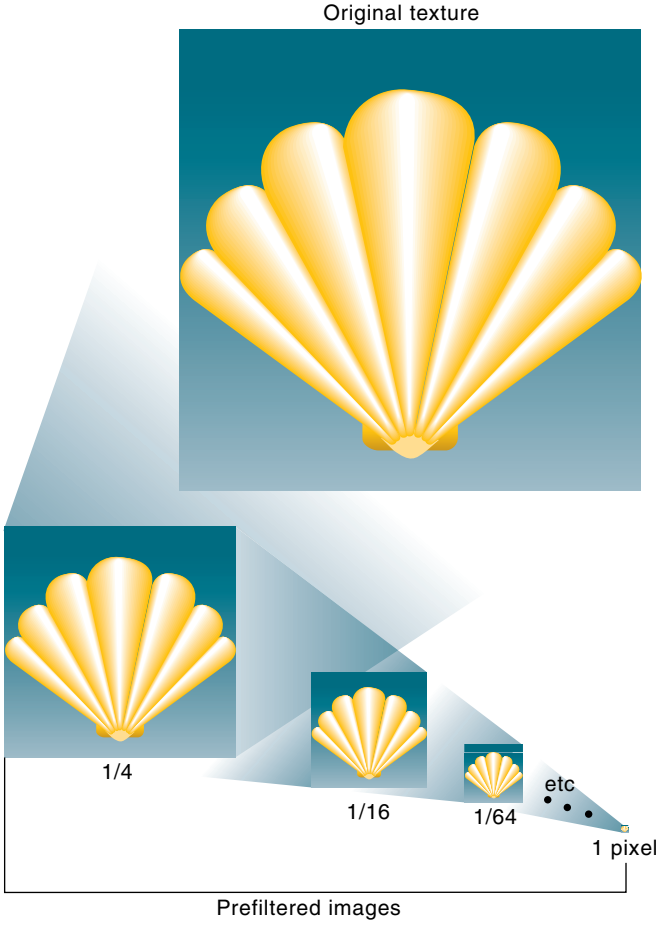



**Figure 9-6** Mipmaps

**Note:** To acquire a full understanding of mipmaps, you need to understand minification filters, which are described in "Filtering" on page 411.

When using mipmapping, OpenGL automatically determines which texture map to use based on the size (in pixels) of the object being mapped. With this approach, the level of detail in the texture map is appropriate for the image that's drawn on the screen—as the image of the object gets smaller, the size of the texture map decreases. Mipmapping requires some extra computation and texture storage area; however, when it's not used, textures that are mapped onto smaller objects might shimmer and flash as the objects move.

To use mipmapping, you must provide all sizes of your texture in powers of 2 between the largest size and a $1 \times 1$ map. For example, if your highest-resolution map is $64 \times 16$, you must also provide maps of size $32 \times 8$, $16 \times 4$, $8 \times 2$, $4 \times 1$, $2 \times 1$, and $1 \times 1$. The smaller maps are typically filtered and averaged-down versions of the largest map in which each texel in a smaller texture is an average of the corresponding 4 texels in the higher-resolution texture. (Since OpenGL doesn't require any particular method for calculating the lower-resolution maps, the differently sized textures could be totally unrelated. In practice, unrelated textures would make the transitions between mipmaps extremely noticeable, as in Plate 20.)

To specify these textures, call **glTexImage2D()** once for each resolution of the texture map, with different values for the *level*, *width*, *height*, and *image* parameters. Starting with zero, *level* identifies which texture in the series is specified; with the previous example, the highest-resolution texture of size $64 \times 16$ would be declared with *level* = 0, the $32 \times 8$ texture with *level* = 1, and so on. In addition, for the mipmapped textures to take effect, you need to choose one of the appropriate filtering methods described in "Filtering" on page 411.

**Note:** This description of OpenGL mipmapping avoids detailed discussion of the scale factor (known as $\lambda$) between texel size and polygon size. This description also assumes default values for parameters related to mipmapping. To see an explanation of $\lambda$ and the effects of mipmapping parameters, see "Calculating the Mipmap Level" on page 405 and "Mipmap Level of Detail Control" on page 406.

Example 9-5 illustrates the use of a series of six texture maps decreasing in size from $32 \times 32$ to $1 \times 1$. This program draws a rectangle that extends from the foreground far back in the distance, eventually disappearing at a point, as shown in Plate 20. Note that the texture coordinates range from 0.0 to 8.0, so 64 copies of the texture map are required to tile the rectangle—eight

in each direction. To illustrate how one texture map succeeds another, each map has a different color.

**Example 9-5** Mipmap Textures: mipmap.c

```
GLubyte mipmapImage32[32][32][4];
GLubyte mipmapImage16[16][16][4];
GLubyte mipmapImage8[8][8][4];
GLubyte mipmapImage4[4][4][4];
GLubyte mipmapImage2[2][2][4];
GLubyte mipmapImage1[1][1][4];

static GLuint texName;

void makeImages(void)
{
   int i, j;

   for (i = 0; i < 32; i++) {
      for (j = 0; j < 32; j++) {
         mipmapImage32[i][j][0] = 255;
         mipmapImage32[i][j][1] = 255;
         mipmapImage32[i][j][2] = 0;
         mipmapImage32[i][j][3] = 255;
      }
   }
   for (i = 0; i < 16; i++) {
      for (j = 0; j < 16; j++) {
         mipmapImage16[i][j][0] = 255;
         mipmapImage16[i][j][1] = 0;
         mipmapImage16[i][j][2] = 255;
         mipmapImage16[i][j][3] = 255;
      }
   }
   for (i = 0; i < 8; i++) {
      for (j = 0; j < 8; j++) {
         mipmapImage8[i][j][0] = 255;
         mipmapImage8[i][j][1] = 0;
         mipmapImage8[i][j][2] = 0;
         mipmapImage8[i][j][3] = 255;
      }
   }
   for (i = 0; i < 4; i++) {
      for (j = 0; j < 4; j++) {
         mipmapImage4[i][j][0] = 0;
         mipmapImage4[i][j][1] = 255;
```

```
            mipmapImage4[i][j][2] = 0;
            mipmapImage4[i][j][3] = 255;
         }
      }
      for (i = 0; i < 2; i++) {
         for (j = 0; j < 2; j++) {
            mipmapImage2[i][j][0] = 0;
            mipmapImage2[i][j][1] = 0;
            mipmapImage2[i][j][2] = 255;
            mipmapImage2[i][j][3] = 255;
         }
      }
      mipmapImage1[0][0][0] = 255;
      mipmapImage1[0][0][1] = 255;
      mipmapImage1[0][0][2] = 255;
      mipmapImage1[0][0][3] = 255;
   }

   void init(void)
   {
      glEnable(GL_DEPTH_TEST);
      glShadeModel(GL_FLAT);

      glTranslatef(0.0, 0.0, -3.6);
      makeImages();
      glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

      glGenTextures(1, &texName);
      glBindTexture(GL_TEXTURE_2D, texName);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                      GL_NEAREST);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                      GL_NEAREST_MIPMAP_NEAREST);
      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 32, 32, 0,
                   GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage32);
      glTexImage2D(GL_TEXTURE_2D, 1, GL_RGBA, 16, 16, 0,
                   GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage16);
      glTexImage2D(GL_TEXTURE_2D, 2, GL_RGBA, 8, 8, 0,
                   GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage8);
      glTexImage2D(GL_TEXTURE_2D, 3, GL_RGBA, 4, 4, 0,
                   GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage4);
      glTexImage2D(GL_TEXTURE_2D, 4, GL_RGBA, 2, 2, 0,
                   GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage2);
      glTexImage2D(GL_TEXTURE_2D, 5, GL_RGBA, 1, 1, 0,
                   GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage1);
```

```
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
    glEnable(GL_TEXTURE_2D);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindTexture(GL_TEXTURE_2D, texName);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
    glTexCoord2f(0.0, 8.0); glVertex3f(-2.0, 1.0, 0.0);
    glTexCoord2f(8.0, 8.0); glVertex3f(2000.0, 1.0, -6000.0);
    glTexCoord2f(8.0, 0.0); glVertex3f(2000.0, -1.0, -6000.0);
    glEnd();
    glFlush();
}
```

Example 9-5 illustrates mipmapping by making each mipmap a different color so that it's obvious when one map is replaced by another. In a real situation, you define mipmaps such that the transition is as smooth as possible. Thus, the maps of lower resolution are usually filtered versions of an original, high-resolution texture map.

The construction of a series of such mipmaps is a software process, and thus isn't part of OpenGL, which is simply a rendering library. Since mipmap construction is such an important operation, the OpenGL Utility Library contains routines that aid in the manipulation of images to be used as mipmapped textures, as described in "Automated Mipmap Generation."

### Calculating the Mipmap Level

Computing which level of mipmap to texture a particular polygon depends on the scale factor between the texture image and the size of the polygon to be textured (in pixels). Let's call this scale factor $\rho$ and also define a second value, $\lambda$, where $\lambda = \log_2 \rho + \text{lod}_{\text{bias}}$. (Since texture images can be multidimensional, it is important to clarify that $\rho$ is the maximum scale factor of all dimensions.)

$\text{lod}_{\text{bias}}$ is the level-of-detail bias, a constant value set by **glTexEnv*()** to adjust $\lambda$. (For information about how to use **glTexEnv*()** to set level-of-detail bias, see "Texture Functions" on page 421.) By default, $\text{lod}_{\text{bias}} = 0.0$, which has no effect. It's best to start with this default value and adjust in small amounts, if needed.

If $\lambda \leq 0.0$, then the texture is smaller than the polygon, so a magnification filter is used. If $\lambda > 0.0$, then a minification filter is used. If the minification

filter selected uses mipmapping, then λ indicates the mipmap level. (The minification-to-magnification switchover point is usually at λ = 0.0, but not always. The choice of mipmapping filter may shift the switchover point.)

For example, if the texture image is 64 × 64 texels and the polygon size is 32 × 32 pixels, then ρ = 2.0 (not 4.0), and therefore λ = 1.0. If the texture image is 64 × 32 texels and the polygon size is 8 × 16 pixels, then ρ = 8.0 (*x* scales by 8.0, *y* by 2.0; use the maximum value) and therefore λ = 3.0.

### Mipmap Level of Detail Control

By default, you must provide a mipmap for every level of resolution, down to 1 texel in every dimension. For some techniques, you want to avoid representing your data with very small mipmaps. For instance, you might use a technique called *mosaicing*, where several smaller images are combined on a single texture. One example of mosaicing is shown in Figure 9-7, where many characters are on a single texture, which may be more efficient than creating a texture image for each character. To map only a single letter from the texture, you make smart use of texture coordinates to isolate the letter you want.

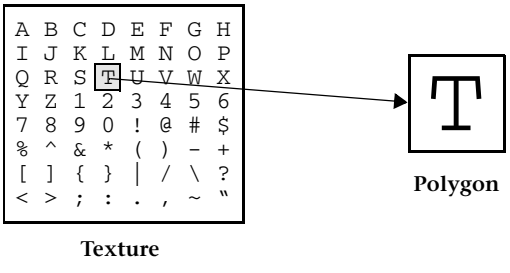



**Figure 9-7** Using a Mosaic Texture

If you have to supply very small mipmaps, the lower-resolution mipmaps of the mosaic crush together detail from many different letters. Therefore, you may want to set restrictions on how low your resolution can go. Generally, you want the capability to add or remove levels of mipmaps as needed.

Another visible mipmapping problem is *popping*—the sudden transition from using one mipmap to using a radically higher- or lower-resolution mipmap, as a mipmapped polygon becomes larger or smaller.

**Note:** Many mipmapping features were introduced in later versions of OpenGL. Check the version of your implementation to see if a particular feature is supported. In some versions, a particular feature may be available as an extension.

To control mipmapping levels, the constants GL_TEXTURE_BASE_LEVEL, GL_TEXTURE_MAX_LEVEL, GL_TEXTURE_MIN_LOD, and GL_TEXTURE_MAX_LOD are passed to **glTexParameter*()**. The first two constants (for brevity, shortened to BASE_LEVEL and MAX_LEVEL in the remainder of this section) control which mipmap levels are used and therefore which levels need to be specified. The other two constants (shortened to MIN_LOD and MAX_LOD) control the active range of the aforementioned scale factor $\lambda$.

These texture parameters address several of the previously described problems. Effective use of BASE_LEVEL and MAX_LEVEL may reduce the number of mipmaps that need to be specified and thereby streamline texture resource usage. Selective use of MAX_LOD may preserve the legibility of a mosaic texture, and MIN_LOD may reduce the popping effect with higher-resolution textures.

BASE_LEVEL and MAX_LEVEL are used to set the boundaries for which mipmap levels are used. BASE_LEVEL is the level of the highest-resolution (largest texture) mipmap level that is used. The default value for BASE_LEVEL is 0. However, you may later change the value for BASE_LEVEL, so that you add additional higher-resolution textures "on the fly." Similarly, MAX_LEVEL limits the lowest-resolution mipmap to be used. The default value for MAX_LEVEL is 1000, which almost always means that the smallest-resolution texture is 1 texel.

To set the base and maximum mipmap levels, use **glTexParameter*()** with the first argument set to GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, or GL_TEXTURE_CUBE_MAP, depending on your textures. The second argument is one of the parameters described in Table 9-1. The third argument denotes the value for the parameter.

| Parameter | Description | Values |
|---|---|---|
| GL_TEXTURE_BASE_LEVEL | level for highest-resolution texture (lowest numbered mipmap level) in use | any non-negative integer |
| GL_TEXTURE_MAX_LEVEL | level for smallest-resolution texture (highest numbered mipmap level) in use | any non-negative integer |

**Table 9-1** Mipmapping Level Parameter Controls

The code in Example 9-6 sets the base and maximum mipmap levels to 2 and 5, respectively. Since the image at the base level (level 2) has a $64 \times 32$ texel resolution, the mipmaps at levels 3, 4, and 5 must have the appropriate lower resolution.

**Example 9-6** Setting Base and Maximum Mipmap Levels

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 2);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 5);
glTexImage2D(GL_TEXTURE_2D, 2, GL_RGBA, 64, 32, 0, GL_RGBA,
             GL_UNSIGNED_BYTE, image1);
glTexImage2D(GL_TEXTURE_2D, 3, GL_RGBA, 32, 16, 0, GL_RGBA,
             GL_UNSIGNED_BYTE, image2);
glTexImage2D(GL_TEXTURE_2D, 4, GL_RGBA, 16, 8, 0, GL_RGBA,
             GL_UNSIGNED_BYTE, image3);
glTexImage2D(GL_TEXTURE_2D, 5, GL_RGBA, 8, 4, 0, GL_RGBA,
             GL_UNSIGNED_BYTE, image4);
```

Later on, you may decide to add additional higher-or lower-resolution mipmaps. For example, you may add a $128 \times 64$ texel texture to this set of mipmaps at level 1, but you must remember to reset BASE_LEVEL.

**Note:** For mipmapping to work, all mipmaps between BASE_LEVEL and the *largest possible level*, inclusive, must be loaded. The largest possible level is the smaller of either the value for MAX_LEVEL or the level at which the size of the mipmap is only 1 texel (either 1, $1 \times 1$, or $1 \times 1 \times 1$). If you fail to load a necessary mipmap level, then texturing may be mysteriously disabled. If you are mipmapping and texturing does not appear, ensure that each required mipmap level has been loaded with a legal texture.

As with BASE_LEVEL and MAX_LEVEL, **glTexParameter*()** sets MIN_LOD and MAX_LOD. Table 9-2 lists possible values.

| Parameter | Description | Values |
|---|---|---|
| GL_TEXTURE_MIN_LOD | minimum value for $\lambda$ (scale factor of texture image versus polygon size) | any value |
| GL_TEXTURE_MAX_LOD | maximum value for $\lambda$ | any value |

**Table 9-2** Mipmapping Level-of-Detail Parameter Controls

The following code is an example of using **glTexParameter*()** to specify the level-of-detail parameters:

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_LOD, 2.5);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_LOD, 4.5);
```

MIN_LOD and MAX_LOD provide minimum and maximum values for $\lambda$ (the scale factor from texture image to polygon) for mipmapped minification, which indirectly specifies which mipmap levels are used.

If you have a $64 \times 64$ pixel polygon and MIN_LOD is the default value of 0.0, then a level 0 $64 \times 64$ texel texture map may be used for minification (provided BASE_LEVEL = 0; as a rule, BASE_LEVEL $\leq$ MIN_LOD). However, if MIN_LOD is set to 2.0, then the largest texture map that may be used for minification is $16 \times 16$ texels, which corresponds to $\lambda = 2.0$.

MAX_LOD has influence only if it is less than the maximum $\lambda$ (which is either MAX_LEVEL or where the mipmap is reduced to 1 texel). In the case of a $64 \times 64$ texel texture map, $\lambda = 6.0$ corresponds to a $1 \times 1$ texel mipmap. In the same case, if MAX_LOD is 4.0, then no mipmap smaller than $4 \times 4$ texels will be used for minification.

You may find that a MIN_LOD that is fractionally greater than BASE_LEVEL or a MAX_LOD that is fractionally less than MAX_LEVEL is best for reducing visual effects (such as popping) related to transitions between mipmaps.

#### Automated Mipmap Generation

Assuming you have constructed the level 0, or highest-resolution, map, the routines **gluBuild1DMipmaps()**, **gluBuild2DMipmaps()**, or **gluBuild3DMipmaps()** construct and define the pyramid of mipmaps down to a resolution of $1 \times 1$ (or 1, for one-dimensional, or $1 \times 1 \times 1$, for three-dimensional). If your original image has dimensions that are not exact powers of 2, **gluBuild*DMipmaps()** helpfully scales the image to the nearest power of 2. Also, if your texture is too large, **gluBuild*DMipmaps()** reduces the size of the image until it fits (as measured by the GL_PROXY_TEXTURE mechanism).

```
int gluBuild1DMipmaps(GLenum target,
                      GLint internalFormat, GLint width,
                      GLenum format, GLenum type,
                      const void *texels);
int gluBuild2DMipmaps(GLenum target, GLint internalFormat,
                      GLint width, GLint height, GLenum format,
                      GLenum type, const void *texels);
int gluBuild3DMipmaps(GLenum target, GLint internalFormat,
                      GLint width, GLint height, GLint depth,
                      GLenum format, GLenum type, const void *texels);
```

Constructs a series of mipmaps and calls **glTexImage*D()** to load the images. The parameters for *target*, *internalFormat*, *width*, *height*, *depth*, *format*, *type*, and *texels* are exactly the same as those for **glTexImage1D()**, **glTexImage2D()**, and **glTexImage3D()**. A value of 0 is returned if all the mipmaps are constructed successfully; otherwise, a GLU error code is returned.

With increased control over level of detail (using BASE_LEVEL, MAX_LEVEL, MIN_LOD, and MAX_LOD), you may need to create only a subset of the mipmaps defined by **gluBuild*DMipmaps()**. For example, you may want to stop at a $4 \times 4$ texel image, rather than go all the way to the smallest $1 \times 1$ texel image. To calculate and load a subset of mipmap levels, you may call **gluBuild*DMipmapLevels()**.

```
int gluBuild1DMipmapLevels(GLenum target, GLint internalFormat,
                           GLint width, GLenum format,
                           GLenum type, GLint level, GLint base,
                           GLint max, const void *texels);
int gluBuild2DMipmapLevels(GLenum target, GLint internalFormat,
                           GLint width, GLint height, GLenum format,
                           GLenum type, GLint level, GLint base,
                           GLint max, const void *texels);
int gluBuild3DMipmapLevels(GLenum target, GLint internalFormat,
                           GLint width, GLint height, GLint depth,
                           GLenum format, GLenum type,
                           GLint level, GLint base, GLint max,
                           const void *texels);
```

Constructs a series of mipmaps and calls **glTexImage*D()** to load the images. *level* indicates the mipmap level of the *texels* image. *base* and *max* determine which mipmap levels will be derived from *texels*. Otherwise, the parameters for *target*, *internalFormat*, *width*, *height*, *depth*, *format*, *type*, and *texels* are exactly the same as those for **glTexImage1D()**, **glTexImage2D()**, and **glTexImage3D()**. A value of 0 is returned if all the mipmaps are constructed successfully; otherwise, a GLU error code is returned.

If you expect that any texels in a mipmapped texture image will change, you will have to replace the complete set of related mipmaps. If you use **glTexParameter*()** to set GL_GENERATE_MIPMAP to GL_TRUE, then any change to the texels (interior or border) of a BASE_LEVEL mipmap will automatically cause all the textures at all mipmap levels from BASE_LEVEL+1 to MAX_LEVEL to be recomputed and replaced. Textures at all other mipmap levels, including at BASE_LEVEL, remain unchanged.

## Filtering

Texture maps are square or rectangular, but after being mapped to a polygon or surface and transformed into screen coordinates, the individual texels of a texture rarely correspond to individual pixels of the final screen image. Depending on the transformations used and the texture mapping applied, a single pixel on the screen can correspond to anything from a tiny portion of a texel (magnification) to a large collection of texels (minification), as shown in Figure 9-8. In either case, it's unclear exactly which texel values should be used and how they should be averaged or interpolated. Consequently, OpenGL allows you to specify any of several filtering options to determine these calculations. The options provide different trade-offs between speed and image quality. Also, you can specify independently the filtering methods for magnification and minification.

In some cases, it isn't obvious whether magnification or minification is called for. If the texture map needs to be stretched (or shrunk) in both the *x*- and *y*- directions, then magnification (or minification) is needed. If the texture map needs to be stretched in one direction and shrunk in the other, OpenGL makes a choice between magnification and minification that in most cases gives the best result possible. It's best to try to avoid these situations by using texture coordinates that map without such distortion. (See "Computing Appropriate Texture Coordinates" on page 427.)

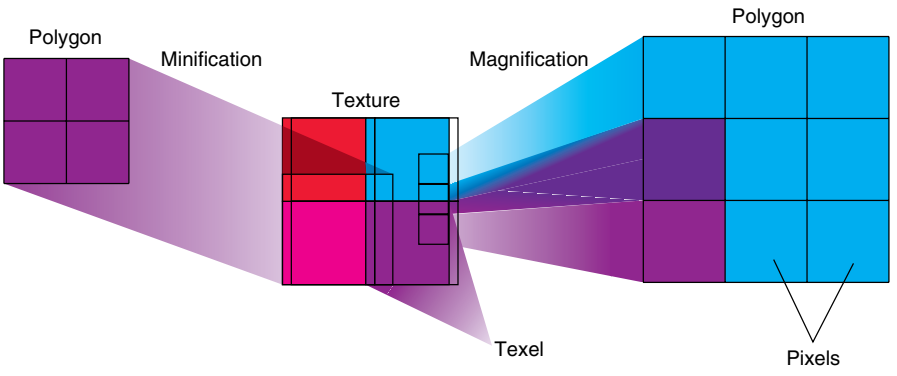



**Figure 9-8** Texture Magnification and Minification

The following lines are examples of how to use **glTexParameter*()** to specify the magnification and minification filtering methods:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_NEAREST);
```

The first argument to **glTexParameter*()** is GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, or GL_TEXTURE_CUBE_MAP, whichever is appropriate. For the purposes of this discussion, the second argument is either GL_TEXTURE_MAG_FILTER, or GL_TEXTURE_MIN_FILTER, to indicate whether you're specifying the filtering method for magnification or minification. The third argument specifies the filtering method; Table 9-3 lists the possible values.

| Parameter | Values |
|---|---|
| GL_TEXTURE_MAG_FILTER | GL_NEAREST or GL_LINEAR |
| GL_TEXTURE_MIN_FILTER | GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, or GL_LINEAR_MIPMAP_LINEAR |

**Table 9-3** Filtering Methods for Magnification and Minification

If you choose GL_NEAREST, the texel with coordinates nearest the center of the pixel is used for both magnification and minification. This can result in aliasing artifacts (sometimes severe). If you choose GL_LINEAR, a weighted

linear average of the $2 \times 2$ array of texels that lie nearest to the center of the pixel is used, again for both magnification and minification. (For three-dimensional textures, it's a $2 \times 2 \times 2$ array; for one-dimensional, it's an average of 2 texels.) When the texture coordinates are near the edge of the texture map, the nearest $2 \times 2$ array of texels might include some that are outside the texture map. In these cases, the texel values used depend on which wrapping mode is in effect and whether you've assigned a border for the texture. (See "Repeating and Clamping Textures" on page 428.) GL_NEAREST requires less computation than GL_LINEAR and therefore might execute more quickly, but GL_LINEAR provides smoother results.

With magnification, even if you've supplied mipmaps, only the base level texture map is used. With minification, you can choose a filtering method that uses the most appropriate one or two mipmaps, as described in the next paragraph. (If GL_NEAREST or GL_LINEAR is specified with minification, only the base level texture map is used.)

As shown in Table 9-3, four additional filtering options are available when minifying with mipmaps. Within an individual mipmap, you can choose the nearest texel value with GL_NEAREST_MIPMAP_NEAREST, or you can interpolate linearly by specifying GL_LINEAR_MIPMAP_NEAREST. Using the nearest texels is faster but yields less desirable results. The particular mipmap chosen is a function of the amount of minification required, and there's a cutoff point from the use of one particular mipmap to the next. To avoid a sudden transition, use GL_NEAREST_MIPMAP_LINEAR or GL_LINEAR_MIPMAP_LINEAR for linear interpolation of texel values from the two nearest best choices of mipmaps. GL_NEAREST_MIPMAP_LINEAR selects the nearest texel in each of the two maps and then interpolates linearly between these two values. GL_LINEAR_MIPMAP_LINEAR uses linear interpolation to compute the value in each of two maps and then interpolates linearly between these two values. As you might expect, GL_LINEAR_MIPMAP_LINEAR generally produces the highest-quality results, but it requires the most computation and therefore might be the slowest.

**Caution:** If you request a mipmapped texture filter, but you have not supplied a full and consistent set of mipmaps (all correct-sized texture images between GL_TEXTURE_BASE_LEVEL and GL_TEXTURE_MAX_LEVEL), OpenGL will, without any error, implicitly disable texturing. If you are trying to use mipmaps and no texturing appears at all, check the texture images at all your mipmap levels.

Some of these texture filters are known by more popular names. GL_NEAREST is often called *point sampling*. GL_LINEAR is known as *bilinear sampling*, because for two-dimensional textures, a $2 \times 2$ array of texels is sampled.

GL_LINEAR_MIPMAP_LINEAR is sometimes known as *trilinear sampling*, because it is a linear average between two bilinearly sampled mipmaps.

**Note:** The minification-to-magnification switchover point is usually at $\lambda = 0.0$, but is affected by the type of minification filter you choose. If the current magnification filter is GL_LINEAR and the minification filter is GL_NEAREST_MIPMAP_NEAREST or GL_NEAREST_MIPMAP_LINEAR, then the switch between filters occurs at $\lambda = 0.5$. This prevents the minified texture from looking sharper than its magnified counterpart.

#### Nate Robins' Texture Tutorial

If you have downloaded Nate Robins' suite of tutorial programs, now run the **texture** tutorial. (For information on how and where to download these programs, see "Nate Robins' OpenGL Tutors" on page xl.) With this tutorial, you can experiment with the texture-mapping filtering method, switching between GL_NEAREST and GL_LINEAR.

## Texture Objects

A texture object stores texture data and makes it readily available. You may control many textures and go back to textures that have been previously loaded into your texture resources. Using texture objects is usually the fastest way to apply textures, resulting in big performance gains, because it is almost always much faster to bind (reuse) an existing texture object than it is to reload a texture image using **glTexImage*D()**.

Also, some implementations support a limited *working set* of high-performance textures. You can use texture objects to load your most often used textures into this limited area.

To use texture objects for your texture data, take these steps:

1. Generate texture names.
2. Initially bind (create) texture objects to texture data, including the image arrays and texture properties.
3. If your implementation supports a working set of high-performance textures, see if you have enough space for all your texture objects. If there isn't enough space, you may wish to establish priorities for each texture object so that more often used textures stay in the working set.
4. Bind and rebind texture objects, making their data currently available for rendering textured models.

## Naming a Texture Object

Any nonzero unsigned integer may be used as a texture name. To avoid accidentally reusing names, consistently use **glGenTextures()** to provide unused texture names.

void **glGenTextures**(GLsizei *n*, GLuint **textureNames*);

Returns *n* currently unused names for texture objects in the array *textureNames*. The names returned in *textureNames* do not have to be a contiguous set of integers.

The names in *textureNames* are marked as used, but they acquire texture state and dimensionality (1D, 2D, or 3D) only when they are first bound.

Zero is a reserved texture name and is never returned as a texture name by **glGenTextures()**.

**glIsTexture()** determines if a texture name is actually in use. If a texture name was returned by **glGenTextures()** but has not yet been bound (calling **glBindTexture()** with the name at least once), then **glIsTexture()** returns GL_FALSE.

GLboolean **glIsTexture**(GLuint *textureName*);

Returns GL_TRUE if *textureName* is the name of a texture that has been bound and has not been subsequently deleted, and returns GL_FALSE if *textureName* is zero or *textureName* is a nonzero value that is not the name of an existing texture.

## Creating and Using Texture Objects

The same routine, **glBindTexture()**, both creates and uses texture objects. When a texture name is initially bound (used with **glBindTexture()**), a new texture object is created with default values for the texture image and texture properties. Subsequent calls to **glTexImage*()**, **glTexSubImage*()**, **glCopyTexImage*()**, **glCopyTexSubImage*()**, **glTexParameter*()**, and **glPrioritizeTextures()** store data in the texture object. The texture object may contain a texture image and associated mipmap images (if any), including associated data such as width, height, border width, internal format, resolution of components, and texture properties. Saved texture

properties include minification and magnification filters, wrapping modes, border color, and texture priority.

When a texture object is subsequently bound once again, its data becomes the current texture state. (The state of the previously bound texture is replaced.)

void **glBindTexture**(GLenum *target*, GLuint *textureName*);

**glBindTexture()** does three things. When using the *textureName* of an unsigned integer other than zero for the first time, a new texture object is created and assigned that name. When binding to a previously created texture object, that texture object becomes active. When binding to a *textureName* value of zero, OpenGL stops using texture objects and returns to the unnamed default texture.

When a texture object is initially bound (that is, created), it assumes the dimensionality of *target*, which is GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, or GL_TEXTURE_CUBE_MAP. Immediately on its initial binding, the state of the texture object is equivalent to the state of the default *target* dimensionality at the initialization of OpenGL. In this initial state, texture properties such as minification and magnification filters, wrapping modes, border color, and texture priority are set to their default values.

In Example 9-7, two texture objects are created in **init()**. In **display()**, each texture object is used to render a different four-sided polygon.

**Example 9-7** Binding Texture Objects: texbind.c

```
#define checkImageWidth 64
#define checkImageHeight 64
static GLubyte checkImage[checkImageHeight][checkImageWidth][4];
static GLubyte otherImage[checkImageHeight][checkImageWidth][4];

static GLuint texName[2];

void makeCheckImages(void)
{
   int i, j, c;

   for (i = 0; i < checkImageHeight; i++) {
      for (j = 0; j < checkImageWidth; j++) {
         c = (((i&0x8)==0)^((j&0x8)==0))*255;
```

```
            checkImage[i][j][0] = (GLubyte) c;
            checkImage[i][j][1] = (GLubyte) c;
            checkImage[i][j][2] = (GLubyte) c;
            checkImage[i][j][3] = (GLubyte) 255;
            c = (((i&0x10)==0)^((j&0x10)==0))*255;
            otherImage[i][j][0] = (GLubyte) c;
            otherImage[i][j][1] = (GLubyte) 0;
            otherImage[i][j][2] = (GLubyte) 0;
            otherImage[i][j][3] = (GLubyte) 255;
        }
    }
}

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
   glEnable(GL_DEPTH_TEST);

   makeCheckImages();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

   glGenTextures(2, texName);
   glBindTexture(GL_TEXTURE_2D, texName[0]);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                   GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                   GL_NEAREST);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,
                checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                checkImage);

   glBindTexture(GL_TEXTURE_2D, texName[1]);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                   GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                   GL_NEAREST);
   glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,
                checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                otherImage);
   glEnable(GL_TEXTURE_2D);
}
```

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glBindTexture(GL_TEXTURE_2D, texName[0]);
   glBegin(GL_QUADS);
   glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
   glTexCoord2f(0.0, 1.0); glVertex3f(-2.0, 1.0, 0.0);
   glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 1.0, 0.0);
   glTexCoord2f(1.0, 0.0); glVertex3f(0.0, -1.0, 0.0);
   glEnd();
   glBindTexture(GL_TEXTURE_2D, texName[1]);
   glBegin(GL_QUADS);
   glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
   glTexCoord2f(0.0, 1.0); glVertex3f(1.0, 1.0, 0.0);
   glTexCoord2f(1.0, 1.0); glVertex3f(2.41421, 1.0, -1.41421);
   glTexCoord2f(1.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
   glEnd();
   glFlush();
}
```

Whenever a texture object is bound once again, you may edit the contents of the bound texture object. Any commands you call that change the texture image or other properties change the contents of the currently bound texture object as well as the current texture state.

In Example 9-7, after completion of **display()**, you are still bound to the texture named by the contents of *texName[1]*. Be careful that you don't call a spurious texture routine that changes the data in that texture object.

When mipmaps are used, all related mipmaps of a single texture image must be put into a single texture object. In Example 9-5, levels 0–5 of a mipmapped texture image are put into a single texture object named *texName*.

## Cleaning Up Texture Objects

As you bind and unbind texture objects, their data still sits around somewhere among your texture resources. If texture resources are limited, deleting textures may be one way to free up resources.

void **glDeleteTextures**(GLsizei *n*, const GLuint ***textureNames*);

Deletes *n* texture objects, named by elements in the array *textureNames*. The freed texture names may now be reused (for example, by **glGenTextures()**).

If a texture that is currently bound is deleted, the binding reverts to the default texture, as if **glBindTexture()** were called with zero for the value of *textureName*. Attempts to delete nonexistent texture names or the texture name of zero are ignored without generating an error.

## A Working Set of Resident Textures

Some OpenGL implementations support a working set of high-performance textures, which are said to be resident. Typically, these implementations have specialized hardware to perform texture operations and a limited hardware cache to store texture images. In this case, using texture objects is recommended, because you are able to load many textures into the working set and then control them.

If all the textures required by the application exceed the size of the cache, some textures cannot be resident. If you want to find out if a single texture is currently resident, bind its object, and then call **glGetTexParameter*v()** to determine the value associated with the GL_TEXTURE_RESIDENT state. If you want to know about the texture residence status of many textures, use **glAreTexturesResident()**.

GLboolean **glAreTexturesResident**(GLsizei *n*,
const GLuint **textureNames*,
GLboolean **residences*);

Queries the texture residence status of the *n* texture objects, named in the array *textureNames*. *residences* is an array in which texture residence status is returned for the corresponding texture objects in the array *textureNames*. If all the named textures in *textureNames* are resident, the **glAreTexturesResident()** function returns GL_TRUE, and the contents of the array *residences* are undisturbed. If any texture in *textureNames* is not resident, then **glAreTexturesResident()** returns GL_FALSE, and the elements in *residences*, which correspond to nonresident texture objects in *textureNames*, are also set to GL_FALSE.

Note that **glAreTexturesResident()** returns the current residence status. Texture resources are very dynamic, and texture residence status may change at any time. Some implementations cache textures when they are first used. It may be necessary to draw with the texture before checking residency.

If your OpenGL implementation does not establish a working set of high-performance textures, then the texture objects are always considered resident. In that case, **glAreTexturesResident()** always returns GL_TRUE and basically provides no information.

### Texture Residence Strategies

If you can create a working set of textures and want to get the best texture performance possible, you really have to know the specifics of your implementation and application. For example, with a visual simulation or video game, you have to maintain performance in all situations. In that case, you should never access a nonresident texture. For these applications, you want to load up all your textures on initialization and make them all resident. If you don't have enough texture memory available, you may need to reduce the size, resolution, and levels of mipmaps for your texture images, or you may use **glTexSubImage*()** to repeatedly reuse the same texture memory.

**Note:** If you have several short-lived textures of the same size, you can use **glTexSubImage*()** to reload existing texture objects with different images. This technique may be more efficient than deleting textures and reestablishing new textures from scratch.

For applications that create textures "on the fly," nonresident textures may be unavoidable. If some textures are used more frequently than others, you may assign a higher priority to those texture objects to increase their likelihood of being resident. Deleting texture objects also frees up space. Short of that, assigning a lower priority to a texture object may make it first in line for being moved out of the working set, as resources dwindle. **glPrioritizeTextures()** is used to assign priorities to texture objects.

void **glPrioritizeTextures**(GLsizei *n*, const GLuint **textureNames*, const GLclampf **priorities*);

Assigns the *n* texture objects, named in the array *textureNames*, the texture residence priorities in the corresponding elements of the array *priorities*. The priority values in the array *priorities* are clamped to the range [0.0, 1.0] before being assigned. Zero indicates the lowest priority (textures least likely to be resident), and 1 indicates the highest priority.

**glPrioritizeTextures()** does not require that any of the textures in *textureNames* be bound. However, the priority might not have any effect on a texture object until it is initially bound.

**glTexParameter*()** also may be used to set a single texture's priority, but only if the texture is currently bound. In fact, use of **glTexParameter*()** is the only way to set the priority of a default texture.

If texture objects have equal priority, typical implementations of OpenGL apply a least recently used (LRU) strategy to decide which texture objects to move out of the working set. If you know that your OpenGL implementation uses this algorithm, then having equal priorities for all texture objects creates a reasonable LRU system for reallocating texture resources.

If your implementation of OpenGL doesn't use an LRU strategy for texture objects of equal priority (or if you don't know how it decides), you can implement your own LRU strategy by carefully maintaining the texture object priorities. When a texture is used (bound), you can maximize its priority, which reflects its recent use. Then, at regular (time) intervals, you can degrade the priorities of all texture objects.

**Note:** Fragmentation of texture memory can be a problem, especially if you're deleting and creating numerous new textures. Although it may be possible to load all the texture objects into a working set by binding them in one sequence, binding them in a different sequence may leave some textures nonresident.

## Texture Functions

In each of the examples presented so far in this chapter, the values in the texture map have been used directly as colors to be painted on the surface being rendered. You can also use the values in the texture map to modulate the color in which the surface would be rendered without texturing or to combine the color in the texture map with the original color of the surface. You choose texturing functions by supplying the appropriate arguments to **glTexEnv*()**.

void **glTexEnv**{if}(GLenum *target*, GLenum *pname*, *TYPE param*);
void **glTexEnv**{if}**v**(GLenum *target*, GLenum *pname*, const *TYPE* ***param*);

Sets the current texturing function. *target* must be either GL_TEXTURE_FILTER_CONTROL or GL_TEXTURE_ENV.

If *target* is GL_TEXTURE_FILTER_CONTROL, then *pname* must be GL_TEXTURE_LOD_BIAS, and *param* is a single, floating-point value used to bias the mipmapping level-of-detail parameter.

If *target* is GL_TEXTURE_ENV and if *pname* is GL_TEXTURE_ENV_MODE, then *param* is one of GL_DECAL, GL_REPLACE, GL_MODULATE, GL_BLEND, GL_ADD, or GL_COMBINE, which specifies how texture values are combined with the color values of the fragment being processed. If *pname* is GL_TEXTURE_ENV_COLOR, then *param* is an array of 4 floating-point numbers (R, G, B, A) which denotes a color to be used for GL_BLEND operations.

If *target* is GL_POINT_SPRITE and if *pname* is GL_COORD_REPLACE, then setting *param* to GL_TRUE will enable the iteration of texture coordinates across a point sprite. Texture coordinates will remain constant across the primitive if *param* is set to GL_FALSE.

**Note:** This is only a partial list of acceptable values for **glTexEnv*()**, excluding texture combiner functions. For complete details about GL_COMBINE and a complete list of options for *pname* and *param* for **glTexEnv*()**, see "Texture Combiner Functions" on page 449 and Table 9-8.

The combination of the texturing function and the base internal format determines how the textures are applied for each component of the texture. The texturing function operates on selected components of the texture and the color values that would be used with no texturing. (Note that the selection is performed after the pixel-transfer function has been applied.) Recall that when you specify your texture map with **glTexImage*D()**, the third argument is the internal format to be selected for each texel.

There are six base internal formats: GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_INTENSITY, GL_RGB, and GL_RGBA. Other internal formats (such as GL_LUMINANCE6_ALPHA2 or GL_R3_G3_B2) specify desired resolutions of the texture components and can be matched to one of these six base internal formats.

Texturing calculations are ultimately in RGBA, but some internal formats are not in RGB. Table 9-4 shows how the RGBA color values are derived from different texture formats, including the less obvious derivations.

| Base Internal Format | Derived Source Color (R, G, B, A) |
|---|---|
| GL_ALPHA | (0, 0, 0, A) |
| GL_LUMINANCE | (L, L, L, 1) |
| GL_LUMINANCE_ALPHA | (L, L, L, A) |

**Table 9-4** Deriving Color Values from Different Texture Formats

| Base Internal Format | Derived Source Color (R, G, B, A) |
|---|---|
| GL_INTENSITY | (I, I, I, I) |
| GL_RGB | (R, G, B, 1) |
| GL_RGBA | (R, G, B, A) |

**Table 9-4 (continued)** Deriving Color Values from Different Texture Formats

Table 9-5 and Table 9-6 show how a texturing function (except for GL_COMBINE) and base internal format determine the texturing application formula used for each component of the texture.

In Table 9-5 and Table 9-6, note the following use of subscripts:

- *s* indicates a texture source color, as determined in Table 9-4
- *f* indicates an incoming fragment value
- *c* indicates values assigned with GL_TEXTURE_ENV_COLOR
- no subscript indicates a final, computed value

In these tables, multiplication of a color triple by a scalar means multiplying each of the R, G, and B components by the scalar; multiplying (or adding) two color triples means multiplying (or adding) each component of the second by (or to) the corresponding component of the first.

| Base Internal Format | GL_REPLACE Function | GL_MODULATE Function | GL_DECAL Function |
|---|---|---|---|
| GL_ALPHA | $C = C_f$<br>$A = A_s$ | $C = C_f$<br>$A = A_f A_s$ | undefined |
| GL_LUMINANCE | $C = C_s$<br>$A = A_f$ | $C = C_f C_s$<br>$A = A_f$ | undefined |
| GL_LUMINANCE_ALPHA | $C = C_s$<br>$A = A_s$ | $C = C_f C_s$<br>$A = A_f A_s$ | undefined |
| GL_INTENSITY | $C = C_s$<br>$A = C_s$ | $C = C_f C_s$<br>$A = A_f C_s$ | undefined |
| GL_RGB | $C = C_s$<br>$A = A_f$ | $C = C_f C_s$<br>$A = A_f$ | $C = C_s$<br>$A = A_f$ |
| GL_RGBA | $C = C_s$<br>$A = A_s$ | $C = C_f C_s$<br>$A = A_f A_s$ | $C = C_f (1 - A_s) + C_s A_s$<br>$A = A_f$ |

**Table 9-5** Replace, Modulate, and Decal Texture Functions

| Base Internal Format | GL_BLEND Function | GL_ADD Function |
|---|---|---|
| GL_ALPHA | $C = C_f$<br>$A = A_f A_s$ | $C = C_f$<br>$A = A_f A_s$ |
| GL_LUMINANCE | $C = C_f(1 - C_s) + C_c C_s$<br>$A = A_f$ | $C = C_f + C_s$<br>$A = A_f$ |
| GL_LUMINANCE_ALPHA | $C = C_f(1 - C_s) + C_c C_s$<br>$A = A_f A_s$ | $C = C_f + C_s$<br>$A = A_f A_s$ |
| GL_INTENSITY | $C = C_f(1 - C_s) + C_c C_s$<br>$A = A_f(1 - A_s) + A_c A_s$ | $C = C_f + C_s$<br>$A = A_f + A_s$ |
| GL_RGB | $C = C_f(1 - C_s) + C_c C_s$<br>$A = A_f$ | $C = C_f + C_s$<br>$A = A_f$ |
| GL_RGBA | $C = C_f(1 - C_s) + C_c C_s$<br>$A = A_f A_s$ | $C = C_f + C_s$<br>$A = A_f A_s$ |

**Table 9-6** Blend and Add Texture Functions

The replacement texture function simply takes the color that would have been painted in the absence of any texture mapping (the fragment's color), tosses it away, and replaces it with the texture color. You use the replacement texture function in situations where you want to apply an opaque texture to an object—such as, for example, if you were drawing a soup can with an opaque label.

The decal texture function is similar to replacement, except that it works for only the RGB and RGBA internal formats and it processes alpha differently. With the RGBA internal format, the fragment's color is blended with the texture color in a ratio determined by the texture alpha, and the fragment's alpha is unchanged. The decal texture function may be used to apply an alpha blended texture, such as an insignia on an airplane wing.

For modulation, the fragment's color is modulated by the contents of the texture map. If the base internal format is GL_LUMINANCE, GL_LUMINANCE_ALPHA, or GL_INTENSITY, the color values are multiplied by the same value, so the texture map modulates between the fragment's color (if the luminance or intensity is 1) to black (if it's 0). For the GL_RGB and GL_RGBA internal formats, each of the incoming color components is multiplied by a corresponding (possibly different) value in the texture. If there's an alpha value, it's multiplied by the fragment's alpha. Modulation is a good texture function for use with lighting, since the lit polygon color can be used to attenuate the texture color. Most of the texture-mapping examples in the color plates use modulation for this reason. White, specular

polygons are often used to render lit, textured objects, and the texture image provides the diffuse color.

The additive texture function simply adds the texture color to the fragment color. If there's an alpha value, it's multiplied by the fragment alpha, except for the GL_INTENSITY format, where the texture's intensity is added to the fragment alpha. Unless the texture and fragment colors are carefully chosen, the additive texture function easily results in oversaturated or clamped colors.

The blending texture function is the only function that uses the color specified by GL_TEXTURE_ENV_COLOR. The luminance, intensity, or color value is used somewhat like an alpha value to blend the fragment's color with the GL_TEXTURE_ENV_COLOR. (See "Sample Uses of Blending" in Chapter 6 for the billboarding example, which uses a blended texture.)

#### Nate Robins' Texture Tutorial

If you have downloaded Nate Robins' suite of tutorial programs, run the **texture** tutorial. Change the texture-mapping environment attribute and see the effects of several texture functions. If you use GL_MODULATE, note the effect of the color specified by **glColor4f()**. If you choose GL_BLEND, see what happens if you change the color specified by the *env_color* array.

## Assigning Texture Coordinates

As you draw your texture-mapped scene, you must provide both object coordinates and texture coordinates for each vertex. After transformation, the object's coordinates determine where on the screen that particular vertex is rendered. The texture coordinates determine which texel in the texture map is assigned to that vertex. In exactly the same way that colors are interpolated between two vertices of shaded polygons and lines, texture coordinates are interpolated between vertices. (Remember that textures are rectangular arrays of data.)

Texture coordinates can comprise one, two, three, or four coordinates. They're usually referred to as the *s-, t-, r-,* and *q*-coordinates to distinguish them from object coordinates (*x, y, z,* and *w*) and from evaluator coordinates (*u* and *v*; see Chapter 12). For one-dimensional textures, you use the *s*-coordinate; for two-dimensional textures, you use *s* and *t*; and for three-dimensional textures, you use *s*, *t*, and *r*. The *q*-coordinate, like *w*, is typically given the value 1 and can be used to create homogeneous coordinates; it's

described as an advanced feature in "The *q*-Coordinate." The command to specify texture coordinates, **glTexCoord*()**, is similar to **glVertex*()**, **glColor*()**, and **glNormal*()**—it comes in similar variations and is used the same way between **glBegin()** and **glEnd()** pairs. Usually, texture-coordinate values range from 0 to 1; values can be assigned outside this range, however, with the results described in "Repeating and Clamping Textures."

void **glTexCoord**{1234}{sifd}(*TYPE coords*);
void **glTexCoord**{1234}{sifd}**v**(const *TYPE* **coords*);

Sets the current texture coordinates (*s, t, r, q*). Subsequent calls to **glVertex*()** result in those vertices being assigned the current texture coordinates. With **glTexCoord1*()**, the *s*-coordinate is set to the specified value, *t* and *r* are set to 0, and *q* is set to 1. Using **glTexCoord2*()** allows you to specify *s* and *t*; *r* and *q* are set to 0 and 1, respectively. With **glTexCoord3*()**, *q* is set to 1 and the other coordinates are set as specified. You can specify all coordinates with **glTexCoord4*()**. Use the appropriate suffix (s, i, f, or d) and the corresponding value for *TYPE* (GLshort, GLint, GLfloat, or GLdouble) to specify the coordinates' data type. You can supply the coordinates individually, or you can use the vector version of the command to supply them in a single array. Texture coordinates are multiplied by the $4 \times 4$ texture matrix before any texture mapping occurs. (See "The Texture Matrix Stack" on page 457.) Note that integer texture coordinates are interpreted directly, rather than being mapped to the range [–1, 1] as normal coordinates are.

The next subsection discusses how to calculate appropriate texture coordinates. Instead of explicitly assigning them yourself, you can choose to have texture coordinates calculated automatically by OpenGL as a function of the vertex coordinates. (See "Automatic Texture-Coordinate Generation" on page 434.)

### Nate Robins' Texture Tutorial

If you have Nate Robins' **texture** tutorial, run it, and experiment with the parameters of **glTexCoord2f()** for the four different vertices. See how you can map from a portion of the entire texture. (What happens if you make a texture coordinate less than 0 or greater than 1?)

## Computing Appropriate Texture Coordinates

Two-dimensional textures are square or rectangular images that are typically mapped to the polygons that make up a polygonal model. In the simplest case, you're mapping a rectangular texture onto a model that's also rectangular—for example, your texture is a scanned image of a brick wall, and your rectangle represents a brick wall of a building. Suppose the brick wall is square and the texture is square, and you want to map the whole texture to the whole wall. The texture coordinates of the texture square are (0, 0), (1, 0), (1, 1), and (0, 1) in counterclockwise order. When you're drawing the wall, just give those four coordinate sets as the texture coordinates as you specify the wall's vertices in counterclockwise order.

Now suppose that the wall is two-thirds as high as it is wide, and that the texture is again square. To avoid distorting the texture, you need to map the wall to a portion of the texture map so that the aspect ratio of the texture is preserved. Suppose that you decide to use the lower two-thirds of the texture map to texture the wall. In this case, use texture coordinates of (0, 0), (1, 0), (1, 2/3), and (0, 2/3) for the texture coordinates, as the wall vertices are traversed in a counterclockwise order.

As a slightly more complicated example, suppose you'd like to display a tin can with a label wrapped around it on the screen. To obtain the texture, you purchase a can, remove the label, and scan it in. Suppose the label is 4 units tall and 12 units around, which yields an aspect ratio of 3 to 1. Since textures must have aspect ratios of $2^n$ to 1, you can either simply not use the top third of the texture, or you can cut and paste the texture until it has the necessary aspect ratio. Suppose you decide not to use the top third. Now suppose the tin can is a cylinder approximated by 30 polygons of length 4 units (the height of the can) and width 12/30 (1/30 of the circumference of the can). You can use the following texture coordinates for each of the 30 approximating rectangles:

1: (0, 0), (1/30, 0), (1/30, 2/3), (0, 2/3)

2: (1/30, 0), (2/30, 0), (2/30, 2/3), (1/30, 2/3)

3: (2/30, 0), (3/30, 0), (3/30, 2/3), (2/30, 2/3)

. . .

30: (29/30, 0), (1, 0), (1, 2/3), (29/30, 2/3)

Only a few curved surfaces such as cones and cylinders can be mapped to a flat surface without geodesic distortion. Any other shape requires some

distortion. In general, the higher the curvature of the surface, the more distortion of the texture is required.

If you don't care about texture distortion, it's often quite easy to find a reasonable mapping. For example, consider a sphere whose surface coordinates are given by ($\cos\theta \cos\phi$, $\cos\theta \sin\phi$, $\sin\theta$), where $0 \le \theta \le 2\pi$ and $0 \le \phi \le \pi$. The $\theta$-$\phi$ rectangle can be mapped directly to a rectangular texture map, but the closer you get to the poles, the more distorted the texture is. The entire top edge of the texture map is mapped to the north pole, and the entire bottom edge to the south pole. For other surfaces, such as that of a torus (doughnut) with a large hole, the natural surface coordinates map to the texture coordinates in a way that produces only a little distortion, so it might be suitable for many applications. Figure 9-9 shows two toruses, one with a small hole (and therefore a lot of distortion near the center) and one with a large hole (and only a little distortion).

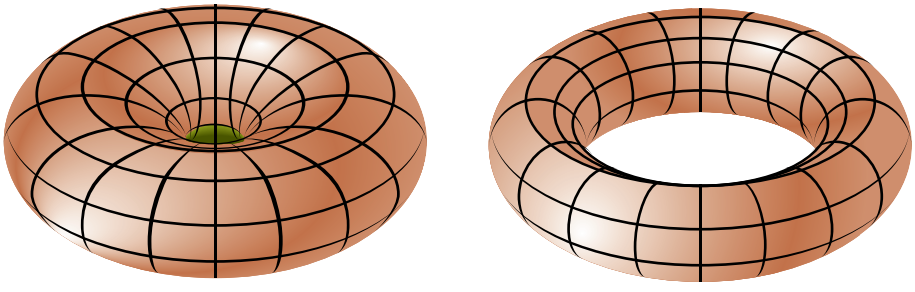


**Figure 9-9** Texture-Map Distortion

If you're texturing spline surfaces generated with evaluators (see Chapter 12), the *u* and *v* parameters for the surface can sometimes be used as texture coordinates. In general, however, there's a large artistic component to successful mapping of textures to polygonal approximations of curved surfaces.

## Repeating and Clamping Textures

You can assign texture coordinates outside the range [0, 1] and have them either clamp or repeat in the texture map. With repeating textures, if you have a large plane with texture coordinates running from 0.0 to 10.0 in both directions, for example, you'll get 100 copies of the texture tiled together on the screen. During repeating, the integer parts of texture coordinates are ignored, and copies of the texture map tile the surface. For most

applications in which the texture is to be repeated, the texels at the top of the texture should match those at the bottom, and similarly for the left and right edges.

A "mirrored" repeat is available, where the surface tiles "flip-flop." For instance, within texture coordinate range [0, 1], a texture may appear oriented from left-to-right (or top-to-bottom or near-to-far), but the "mirrored" repeat wrapping reorients the texture from right-to-left for texture coordinate range [1, 2], then back again to left-to-right for coordinates [2, 3], and so on.

Another possibility is to clamp the texture coordinates: any values greater than 1.0 are set to 1.0, and any values less than 0.0 are set to 0.0. Clamping is useful for applications in which you want a single copy of the texture to appear on a large surface. If the texture coordinates of the surface range from 0.0 to 10.0 in both directions, one copy of the texture appears in the lower left corner of the surface.

If you are using textures with borders or have specified a texture border color, both the wrapping mode and the filtering method (see "Filtering" on page 411) influence whether and how the border information is used.

If you're using the filtering method GL_NEAREST, the closest texel in the texture is used. For most wrapping modes, the border (or border color) is ignored. However, if the texture coordinate is outside the range [0, 1] and the wrapping mode is GL_CLAMP_TO_BORDER, then the nearest border texel is chosen. (If no border is present, the constant border color is used.)

If you've chosen GL_LINEAR as the filtering method, a weighted combination in a $2 \times 2$ array (for two-dimensional textures) of color data is used for texture application. If there is a border or border color, the texture and border colors are used together, as follows:

- For the wrapping mode GL_REPEAT, the border is always ignored. The $2 \times 2$ array of weighted texels wraps to the opposite edge of the texture. Thus, texels at the right edge are averaged with those at the left edge, and top and bottom texels are also averaged.
- For the wrapping mode GL_CLAMP, the texel from the border (or GL_TEXTURE_BORDER_COLOR) is used in the $2 \times 2$ array of weighted texels.
- For the wrapping mode GL_CLAMP_TO_EDGE, the border is always ignored. Texels at or near the edge of the texture are used for texturing calculations, but not the border.
- For the wrapping mode GL_CLAMP_TO_BORDER, if the texture coordinate is outside the range [0, 1], then only border texels (or if no

border is present, the constant border color) are used for texture application. Near the edge of texture coordinates, texels from both the border and the interior texture may be sampled in a $2 \times 2$ array.

If you are using clamping, you can avoid having the rest of the surface affected by the texture. To do this, use alpha values of 0 for the edges (or borders, if they are specified) of the texture. The decal texture function directly uses the texture's alpha value in its calculations. If you are using one of the other texture functions, you may also need to enable blending with good source and destination factors. (See "Blending" in Chapter 6.)

To see the effects of wrapping, you must have texture coordinates that venture beyond [0.0, 1.0]. Start with Example 9-1, and modify the texture coordinates for the squares by mapping the texture coordinates from 0.0 to 4.0, as follows:

```
glBegin(GL_QUADS);
  glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
  glTexCoord2f(0.0, 4.0); glVertex3f(-2.0, 1.0, 0.0);
  glTexCoord2f(4.0, 4.0); glVertex3f(0.0, 1.0, 0.0);
  glTexCoord2f(4.0, 0.0); glVertex3f(0.0, -1.0, 0.0);

  glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
  glTexCoord2f(0.0, 4.0); glVertex3f(1.0, 1.0, 0.0);
  glTexCoord2f(4.0, 4.0); glVertex3f(2.41421, 1.0, -1.41421);
  glTexCoord2f(4.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
glEnd();
```

With GL_REPEAT wrapping, the result is as shown in Figure 9-10.

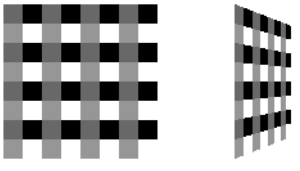


**Figure 9-10** Repeating a Texture

In this case, the texture is repeated in both the *s*- and *t*- directions, since the following calls are made to **glTexParameter*()**:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

Some OpenGL implementations support GL_MIRRORED_REPEAT wrapping, which reverses orientation at every integer texture coordinate boundary. Figure 9-11 shows the contrast between ordinary repeat wrapping (left) and the mirrored repeat (right).

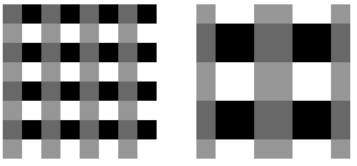


**Figure 9-11** Comparing GL_REPEAT to GL_MIRRORED_REPEAT

In Figure 9-12, GL_CLAMP is used for each direction. Where the texture coordinate s or t is greater than one, the texel used is from where each texture coordinate is exactly one.

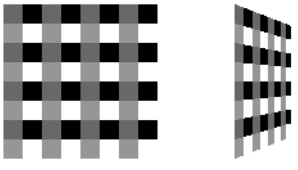


**Figure 9-12** Clamping a Texture

Wrapping modes are independent for each direction. You can also clamp in one direction and repeat in the other, as shown in Figure 9-13.

**Figure 9-13** Repeating and Clamping a Texture

You've now seen several arguments for **glTexParameter*()**, which is summarized as follows.

void **glTexParameter**{if}(GLenum *target*, GLenum *pname*, *TYPE param*);
void **glTexParameter**{if}**v**(GLenum *target*, GLenum *pname*,
const *TYPE* ***param*);

Sets various parameters that control how a texture is treated as it's applied to a fragment or stored in a texture object. The *target* parameter is GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, or GL_TEXTURE_CUBE_MAP to match the intended texture. The possible values for *pname* and *param* are shown in Table 9-7. You can use the vector version of the command to supply an array of values for GL_TEXTURE_BORDER_COLOR, or you can supply individual values for other parameters using the nonvector version. If these values are supplied as integers, they're converted to floating-point numbers according to Table 4-1; they're also clamped to the range [0, 1].

| Parameter | Values |
|---|---|
| GL_TEXTURE_WRAP_S | GL_CLAMP, GL_CLAMP_TO_EDGE, GL_CLAMP_TO_BORDER, GL_REPEAT, GL_MIRRORED_REPEAT |
| GL_TEXTURE_WRAP_T | GL_CLAMP, GL_CLAMP_TO_EDGE, GL_CLAMP_TO_BORDER, GL_REPEAT, GL_MIRRORED_REPEAT |
| GL_TEXTURE_WRAP_R | GL_CLAMP, GL_CLAMP_TO_EDGE, GL_CLAMP_TO_BORDER, GL_REPEAT, GL_MIRRORED_REPEAT |
| GL_TEXTURE_MAG_FILTER | GL_NEAREST, GL_LINEAR |
| GL_TEXTURE_MIN_FILTER | GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR |
| GL_TEXTURE_BORDER_COLOR | any four values in [0.0, 1.0] |
| GL_TEXTURE_PRIORITY | [0.0, 1.0] for the current texture object |
| GL_TEXTURE_MIN_LOD | any floating-point value |
| GL_TEXTURE_MAX_LOD | any floating-point value |

**Table 9-7** glTexParameter*() Parameters

| Parameter | Values |
|---|---|
| GL_TEXTURE_BASE_LEVEL | any non-negative integer |
| GL_TEXTURE_MAX_LEVEL | any non-negative integer |
| GL_TEXTURE_LOD_BIAS | any floating-point value |
| GL_DEPTH_TEXTURE_MODE | GL_LUMINANCE, GL_INTENSITY, GL_ALPHA |
| GL_TEXTURE_COMPARE_MODE | GL_NONE, GL_COMPARE_R_TO_TEXTURE |
| GL_TEXTURE_COMPARE_FUNC | GL_LEQUAL, GL_GEQUAL, GL_LESS, GL_GREATER, GL_EQUAL, GL_NOTEQUAL, GL_ALWAYS, GL_NEVER |
| GL_GENERATE_MIPMAP | GL_TRUE, GL_FALSE |

**Table 9-7 (continued)** glTexParameter*() Parameters

#### Try This


Figures 9-12 and 9-13 are drawn using GL_NEAREST for the minification and magnification filters. What happens if you change the filter values to GL_LINEAR? The resulting image should look more blurred.

Border information may be used while calculating texturing. For the simplest demonstration of this, set GL_TEXTURE_BORDER_COLOR to a noticeable color. With the filters set to GL_NEAREST and the wrapping mode set to GL_CLAMP_TO_BORDER, the border color affects the textured object (for texture coordinates beyond the range [0, 1]). The border also affects the texturing with the filters set to GL_LINEAR and the wrapping mode set to GL_CLAMP.

What happens if you switch the wrapping mode to GL_CLAMP_TO_EDGE or GL_REPEAT? In both cases, the border color is ignored.

### Nate Robins' Texture Tutorial

Run the Nate Robins' **texture** tutorial and see the effects of the wrapping parameters GL_REPEAT and GL_CLAMP. You will need to make the texture coordinates at the vertices (parameters to **glTexCoord2f()**) less than 0 and/or greater than 1 to see any repeating or clamping effect.

## Automatic Texture-Coordinate Generation

You can use texture mapping to make contours on your models or to simulate the reflections from an arbitrary environment on a shiny model. To achieve these effects, let OpenGL automatically generate the texture coordinates for you, rather than explicitly assign them with **glTexCoord*()**. To generate texture coordinates automatically, use the command **glTexGen()**.

void **glTexGen**{ifd}(GLenum *coord*, GLenum *pname*, *TYPE param*);
void **glTexGen**{ifd}**v**(GLenum *coord*, GLenum *pname*, const *TYPE* ***param*);

Specifies the functions for automatically generating texture coordinates. The first parameter, *coord*, must be GL_S, GL_T, GL_R, or GL_Q to indicate whether texture coordinate *s, t, r*, or *q* is to be generated. The *pname* parameter is GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE, or GL_EYE_PLANE. If it's GL_TEXTURE_GEN_MODE, *param* is an integer (or, in the vector version of the command, points to an integer) that is one of GL_OBJECT_LINEAR, GL_EYE_LINEAR, GL_SPHERE_MAP, GL_REFLECTION_MAP, or GL_NORMAL_MAP. These symbolic constants determine which function is used to generate the texture coordinate. With either of the other possible values for *pname*, *param* is a pointer to an array of values (for the vector version) specifying parameters for the texture-generation function.

The different methods of texture-coordinate generation have different uses. Specifying the reference plane in object coordinates is best when a texture image remains fixed to a moving object. Thus, GL_OBJECT_LINEAR would be used for putting a wood grain on a tabletop. Specifying the reference plane in eye coordinates (GL_EYE_LINEAR) is best for producing dynamic contour lines on moving objects. GL_EYE_LINEAR may be used by specialists in the geosciences who are drilling for oil or gas. As the drill goes deeper into the ground, the drill may be rendered with different colors to represent the layers of rock at increasing depths. GL_SPHERE_MAP and GL_REFLECTION_MAP are used mainly for spherical environment mapping, and GL_NORMAL_MAP is used for cube maps. (See "Sphere Map" on page 439 and "Cube Map Textures" on page 441.)

## Creating Contours

When GL_TEXTURE_GEN_MODE and GL_OBJECT_LINEAR are specified, the generation function is a linear combination of the object coordinates of the vertex ($x_o$, $y_o$, $z_o$, $w_o$):

generated coordinate = $p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o$

The $p_1$, ..., $p_4$ values are supplied as the *param* argument to **glTexGen*v()**, with *pname* set to GL_OBJECT_PLANE. With $p_1$, ..., $p_4$ correctly normalized, this function gives the distance from the vertex to a plane. For example, if $p_2 = p_3 = p_4 = 0$ and $p_1 = 1$, the function gives the distance between the vertex and the plane $x = 0$. The distance is positive on one side of the plane, negative on the other, and zero if the vertex lies on the plane.

Initially, in Example 9-8, equally spaced contour lines are drawn on a teapot; the lines indicate the distance from the plane $x = 0$. The coefficients for the plane $x = 0$ are in this array:

```
static GLfloat xequalzero[] = {1.0, 0.0, 0.0, 0.0};
```

Since only one property is being shown (the distance from the plane), a one-dimensional texture map suffices. The texture map is a constant green color, except that at equally spaced intervals it includes a red mark. Since the teapot is sitting on the *xy*-plane, the contours are all perpendicular to its base. Plate 18 shows the picture drawn by the program.

In the same example, pressing the 's' key changes the parameters of the reference plane to

```
static GLfloat slanted[] = {1.0, 1.0, 1.0, 0.0};
```

The contour stripes are parallel to the plane $x + y + z = 0$, slicing across the teapot at an angle, as shown in Plate 18. To restore the reference plane to its initial value, $x = 0$, press the 'x' key.

**Example 9-8** Automatic Texture-Coordinate Generation: texgen.c

```
#define stripeImageWidth 32
GLubyte stripeImage[4*stripeImageWidth];

static GLuint texName;

void makeStripeImage(void)
{
   int j;
```

```
   for (j = 0; j < stripeImageWidth; j++) {
      stripeImage[4*j]   = (GLubyte) ((j<=4) ? 255 : 0);
      stripeImage[4*j+1] = (GLubyte) ((j>4) ? 255 : 0);
      stripeImage[4*j+2] = (GLubyte) 0;
      stripeImage[4*j+3] = (GLubyte) 255;
   }
}

/*  planes for texture-coordinate generation  */
static GLfloat xequalzero[] = {1.0, 0.0, 0.0, 0.0};
static GLfloat slanted[] = {1.0, 1.0, 1.0, 0.0};
static GLfloat *currentCoeff;
static GLenum currentPlane;
static GLint currentGenMode;

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_SMOOTH);

   makeStripeImage();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

   glGenTextures(1, &texName);
   glBindTexture(GL_TEXTURE_1D, texName);
   glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
                   GL_LINEAR);
   glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
                   GL_LINEAR);
   glTexImage1D(GL_TEXTURE_1D, 0, GL_RGBA, stripeImageWidth, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, stripeImage);

   glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
   currentCoeff = xequalzero;
   currentGenMode = GL_OBJECT_LINEAR;
   currentPlane = GL_OBJECT_PLANE;
   glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
   glTexGenfv(GL_S, currentPlane, currentCoeff);

   glEnable(GL_TEXTURE_GEN_S);
   glEnable(GL_TEXTURE_1D);
   glEnable(GL_CULL_FACE);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
```

```
   glEnable(GL_AUTO_NORMAL);
   glEnable(GL_NORMALIZE);
   glFrontFace(GL_CW);
   glCullFace(GL_BACK);
   glMaterialf(GL_FRONT, GL_SHININESS, 64.0);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glPushMatrix();
   glRotatef(45.0, 0.0, 0.0, 1.0);
   glBindTexture(GL_TEXTURE_1D, texName);
   glutSolidTeapot(2.0);
   glPopMatrix();
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho(-3.5, 3.5, -3.5*(GLfloat)h/(GLfloat)w,
               3.5*(GLfloat)h/(GLfloat)w, -3.5, 3.5);
   else
      glOrtho(-3.5*(GLfloat)w/(GLfloat)h,
               3.5*(GLfloat)w/(GLfloat)h, -3.5, 3.5, -3.5, 3.5);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 'e':
      case 'E':
         currentGenMode = GL_EYE_LINEAR;
         currentPlane = GL_EYE_PLANE;
         glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
         glTexGenfv(GL_S, currentPlane, currentCoeff);
         glutPostRedisplay();
         break;
      case 'o':
      case 'O':
```

```
            currentGenMode = GL_OBJECT_LINEAR;
            currentPlane = GL_OBJECT_PLANE;
            glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
            glTexGenfv(GL_S, currentPlane, currentCoeff);
            glutPostRedisplay();
            break;
         case 's':
         case 'S':
            currentCoeff = slanted;
            glTexGenfv(GL_S, currentPlane, currentCoeff);
            glutPostRedisplay();
            break;
         case 'x':
         case 'X':
            currentCoeff = xequalzero;
            glTexGenfv(GL_S, currentPlane, currentCoeff);
            glutPostRedisplay();
            break;
         case 27:
            exit(0);
            break;
         default:
            break;
      }
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(256, 256);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

You enable texture-coordinate generation for the *s*-coordinate by passing GL_TEXTURE_GEN_S to **glEnable()**. To generate other coordinates, enable them with GL_TEXTURE_GEN_T, GL_TEXTURE_GEN_R, or GL_TEXTURE_GEN_Q. Use **glDisable()** with the appropriate constant to disable coordinate generation. Also note the use of GL_REPEAT to cause the contour lines to be repeated across the teapot.

The GL_OBJECT_LINEAR function calculates the texture coordinates in the model's coordinate system. Initially, in Example 9-8, the GL_OBJECT_LINEAR function is used, so the contour lines remain perpendicular to the base of the teapot, no matter how the teapot is rotated or viewed. However, if you press the 'e' key, the texture-generation mode is changed from GL_OBJECT_LINEAR to GL_EYE_LINEAR, and the contour lines are calculated relative to the eye coordinate system. (Pressing the 'o' key restores GL_OBJECT_LINEAR as the texture-generation mode.) If the reference plane is $x = 0$, the result is a teapot with red stripes parallel to the *yz*-plane from the eye's point of view, as shown in Plate 18. Mathematically, you are multiplying the vector $(p_1\ p_2\ p_3\ p_4)$ by the inverse of the modelview matrix to obtain the values used to calculate the distance to the plane. The texture coordinate is generated with the following function:

generated coordinate = $p_1'x_e + p_2'y_e + p_3'z_e + p_4'w_e$

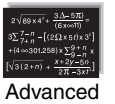
where $(p_1'\ p_2'\ p_3'\ p_4') = (p_1\ p_2\ p_3\ p_4)\mathbf{M}^{-1}$

In this case, $(x_e, y_e, z_e, w_e)$ are the eye coordinates of the vertex, and $p_1$, ..., $p_4$ are supplied as the *param* argument to **glTexGen***(), with *pname* set to GL_EYE_PLANE. The primed values are calculated only at the time they're specified, so this operation isn't as computationally expensive as it looks.

In all these examples, a single texture coordinate is used to generate contours. *s*, *t*, and (if needed) *r* texture coordinates can be generated independently, however, to indicate the distances to two or three different planes. With a properly constructed two- or three-dimensional texture map, the resulting two or three sets of contours can be viewed simultaneously. For an added level of complexity, you can mix generation functions. For example, you can calculate the *s*-coordinate using GL_OBJECT_LINEAR, and the *t*-coordinate using GL_EYE_LINEAR.

## Sphere Map

#### Advanced


Advanced

The goal of environment mapping is to render an object as if it were perfectly reflective, so that the colors on its surface are those reflected to the eye from its surroundings. In other words, if you look at a perfectly polished, perfectly reflective silver object in a room, you see the reflections of the walls, floor, and other items in the room from the object. (A classic example of using environment mapping is the evil, morphing cyborg in the film *Terminator 2*.) The objects whose reflections you see depend on the position of your eye and on the position and surface angles of the silver

object. To perform environment mapping, all you have to do is create an appropriate texture map and then have OpenGL generate the texture coordinates for you.

Environment mapping is an approximation based on the assumption that the items in the environment are far away in comparison with the surfaces of the shiny object—that is, it's a small object in a large room. With this assumption, to find the color of a point on the surface, take the ray from the eye to the surface, and reflect the ray off the surface. The direction of the reflected ray completely determines the color to be painted there. Encoding a color for each direction on a flat texture map is equivalent to putting a polished perfect sphere in the middle of the environment and taking a picture of it with a camera that has a lens with a very long focal length placed far away. Mathematically, the lens has an infinite focal length and the camera is infinitely far away. The encoding therefore covers a circular region of the texture map, tangent to the top, bottom, left, and right edges of the map. The texture values outside the circle make no difference, because they are never accessed in environment mapping.

To make a perfectly correct environment texture map, you need to obtain a large silvered sphere, take a photograph of it in some environment with a camera located an infinite distance away and with a lens that has an infinite focal length, and scan in the photograph. To approximate this result, you can use a scanned-in photograph of an environment taken with an extremely wide-angle (or fish-eye) lens. Plate 21 shows a photograph taken with such a lens and the results when that image is used as an environment map.

Once you've created a texture designed for environment mapping, you need to invoke OpenGL's environment-mapping algorithm. This algorithm finds the point on the surface of the sphere with the same tangent surface as that of the point on the object being rendered, and it paints the object's point with the color visible on the sphere at the corresponding point.

To generate automatically the texture coordinates to support environment mapping, use this code in your program:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

The GL_SPHERE_MAP constant creates the proper texture coordinates for the environment mapping. As shown, you need to specify it for both the *s*- and *t*-directions. However, you don't have to specify any parameters for the texture-coordinate generation function.

The GL_SPHERE_MAP texture function generates texture coordinates using the following mathematical steps:

1. **u** is the unit vector pointing from the origin to the vertex (in eye coordinates).
2. **n'** is the current normal vector, after transformation to eye coordinates.
3. **r** is the reflection vector, $(r_x \;\; r_y \;\; r_z)^T$, which is calculated by $\mathbf{u} - 2\mathbf{n}'\mathbf{n}'^T\mathbf{u}$.
4. An interim value, *m*, is calculated by

   $$m = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$$

5. Finally, the *s* and *t* texture coordinates are calculated by

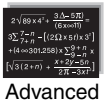
   $$s = r_x/m + \frac{1}{2}$$

   and

   $$t = r_y/m + \frac{1}{2}$$

## Cube Map Textures

#### Advanced


Advanced

Cube map textures are a special technique that uses a set of six two-dimensional texture images to form a texture cube centered at the origin. For each fragment, the texture coordinates (*s, t, r*) are treated as a direction vector, with each texel representing what on the texture cube is "seen" from the origin. Cube maps are ideal for environment, reflection, and lighting effects. Cube maps can also wrap a spherical object with textures, distributing texels relatively evenly on all its sides.

The cube map textures are supplied by calling **glTexImage2D()** six times, with the *target* argument indicating the face of the cube (+X, –X, +Y, –Y, +Z, or –Z). As the name implies, each cube map texture must have the same

dimensions so that a cube is formed with the same number of texels on each side, as shown in this code, where *imageSize* has been set to a power of 2:

```
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGBA,
 imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image1);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0, GL_RGBA,
 imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image4);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, 0, GL_RGBA,
 imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image2);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, GL_RGBA,
 imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image5);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, 0, GL_RGBA,
 imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image3);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, GL_RGBA,
 imageSize, imageSize, 0, GL_RGBA, GL_UNSIGNED_BYTE, image6);
```

Useful cube map texture images may be generated by setting up a (real or synthetic) camera at the origin of a scene and taking six "snapshots" with 90-degree field-of-view, oriented along the positive and negative axes. The "snapshots" break up the entire 3D space into six frustums, which intersect at the origin.

Cube map functionality is orthogonal to many other texturing operations, so cube maps work with standard texturing features, such as texture borders, mipmaps, copying images, subimages, and multitexturing. There is a special proxy texture target for cube maps (GL_PROXY_TEXTURE_CUBE_MAP) because a cube map generally uses six times as much memory as an ordinary 2D texture. Texture parameters and texture objects should be established for the entire cube map as a whole, not for the six individual cube faces. The following code is an example of setting wrapping and filtering methods with the *target* GL_TEXTURE_CUBE_MAP:

```
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
                GL_REPEAT);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
                GL_REPEAT);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R,
                GL_REPEAT);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,
                GL_NEAREST);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,
                GL_NEAREST);
```

To determine which texture (and texels) to use for a given fragment, the current texture coordinates (*s, t, r*) first select one of the six textures, based upon which of *s, t,* and *r* has the largest absolute value (major axis) and its sign (orientation). The remaining two coordinates are divided by the

coordinate with the largest value to determine a new ($s'$, $t'$), which is used to look up the corresponding texel(s) in the selected texture of the cube map.

Although you can calculate and specify the texture coordinates explicitly, this is generally laborious and unnecessary. Almost always, you'll want to use **glTexGen*()** to automatically generate cube map texture coordinates, using one of the two special texture coordinate generation modes: GL_REFLECTION_MAP or GL_NORMAL_MAP.

GL_REFLECTION_MAP uses the same calculations (until step 3 of the sphere-mapping coordinate computation described in "Sphere Map" on page 439) as the GL_SPHERE_MAP texture coordinate generation to determine ($r_x$ $r_y$ $r_z$) for use as ($s$, $t$, $r$). The reflection map mode is well-suited for environment mapping as an alternative to sphere mapping.

GL_NORMAL_MAP is particularly useful for rendering scenes with infinite (or distant local) light sources and diffuse reflection. GL_NORMAL_MAP uses the model-view matrix to transform the vertex's normal into eye coordinates. The resulting ($n_x$ $n_y$ $n_z$) becomes texture coordinates ($s$, $t$, $r$). In Example 9-9, the normal map mode is used for texture generation and cube map texturing is also enabled.

**Example 9-9** Generating Cube Map Texture Coordinates: cubemap.c

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
/*  turn on cube map texturing  */
glEnable(GL_TEXTURE_CUBE_MAP);
```

## Multitexturing

During standard texturing, a single texture image is applied once to a polygon. Multitexturing allows several textures to be applied, one by one in a pipeline of texture operations, to the same polygon. There is a series of *texture units*, where each texture unit performs a single texturing operation and successively passes its result onto the next texture unit, until all defined units are completed. Figure 9-14 shows how a fragment might undergo four texturing operations—one for each of four texture units.

Multitexturing enables advanced rendering techniques, such as lighting effects, decals, compositing, and detail textures.

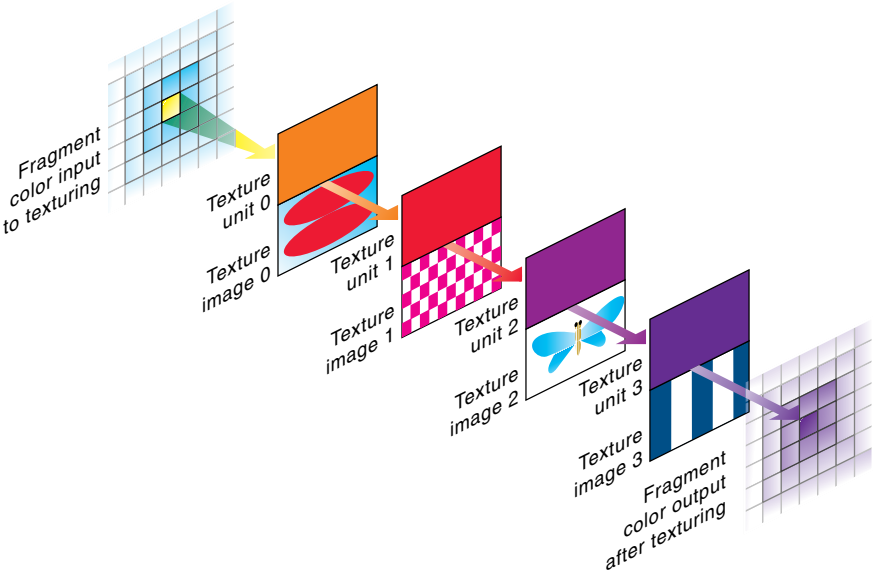



**Figure 9-14** Multitexture Processing Pipeline

### Steps in Multitexturing

To write code that uses multitexturing, perform the following steps:

**Note:** In feedback mode, multitexturing is undefined beyond the first texture unit.

1. For each texturing unit, establish the texturing state, including texture image, filter, environment, coordinate generation, and matrix. Use **glActiveTexture()** to change the current texture unit. This is discussed further in the next subsection, "Establishing Texture Units." You may also call **glGetIntegerv(GL_MAX_TEXTURE_UNITS,...)** to see how many texturing units are available on your implementation. In a worst-case scenario, there are at least two texture units.
2. During vertex specification, use **glMultiTexCoord*()** to specify more than one texture coordinate per vertex. A different texture coordinate may be used for each texturing unit. Each texture coordinate will be used during a different texturing pass. Automatic texture-coordinate generation and specification of texture coordinates in vertex arrays are special cases of this situation. The special cases are described in "Other Methods of Texture-Coordinate Specification" on page 448.

### Establishing Texture Units

Multitexturing introduces multiple texture units, which are additional texture application passes. Each texture unit has identical capabilities and houses its own texturing state, including the following:

- Texture image
- Filtering parameters
- Environment application
- Texture matrix stack
- Automatic texture-coordinate generation
- Vertex-array specification (if needed)

Each texture unit combines the previous fragment color with its texture image, according to its texture state. The resulting fragment color is passed onto the next texture unit, if it is active.

To assign texture information to each texture unit, the routine **glActiveTexture()** selects the current texture unit to be modified. After that, calls to **glTexImage*()**, **glTexParameter*()**, **glTexEnv*()**, **glTexGen*()**, and **glBindTexture()** affect only the current texture unit. Queries of these texture states also apply to the current texture unit, as well as queries of the current texture coordinates and current raster texture coordinates.

void **glActiveTexture**(GLenum *texUnit*);

Selects the texture unit that is currently modified by texturing routines. *texUnit* is a symbolic constant of the form GL_TEXTURE*i*, where *i* is in the range from 0 to $k - 1$, and $k$ is the maximum number of texture units.

If you use texture objects, you can bind a texture to the current texture unit. The current texture unit has the values of the texture state contained within the texture object (including the texture image).

The following code fragment, Example 9-10, has two distinct parts. In the first part, two ordinary texture objects are created (assume the arrays *texels0* and *texels1* define texture images). In the second part, the two texture objects are used to set up two texture units.

**Example 9-10** Initializing Texture Units for Multitexturing: multitex.c

```
/*  Two ordinary texture objects are created  */
GLuint texNames[2];
glGenTextures(2, texNames);
glBindTexture(GL_TEXTURE_2D, texNames[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 32, 32, 0, GL_RGBA,
             GL_UNSIGNED_BYTE, texels0);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glBindTexture(GL_TEXTURE_2D, texNames[1]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 16, 16, 0, GL_RGBA,
            GL_UNSIGNED_BYTE, texels1);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                GL_CLAMP_TO_EDGE);
/*  Use the two texture objects to define two texture units
 *  for use in multitexturing.  */
glActiveTexture(GL_TEXTURE0);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texNames[0]);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
glMatrixMode(GL_TEXTURE);
   glLoadIdentity();
   glTranslatef(0.5f, 0.5f, 0.0f);
   glRotatef(45.0f, 0.0f, 0.0f, 1.0f);
   glTranslatef(-0.5f, -0.5f, 0.0f);
glMatrixMode(GL_MODELVIEW);
glActiveTexture(GL_TEXTURE1);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texNames[1]);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

When a textured polygon is now rendered, it is rendered with two texturing units. In the first unit, the *texels0* texture image is applied with nearest texel filtering, repeat wrapping, replacement texture environment, and a texture matrix that rotates the texture image. After the first unit is completed, the newly textured polygon is sent onto the second texture unit (GL_TEXTURE1), where it is processed with the *texels1* texture image with linear filtering, edge clamping, modulation texture environment, and the default identity texture matrix.

**Note:** Operations to a texture attribute group (using **glPushAttrib()**, **glPushClientAttrib()**, **glPopAttrib()**, or **glPopClientAttrib()**) save or restore the texture state of *all* texture units (except for the texture matrix stack).

### Specifying Vertices and Their Texture Coordinates

With multitexturing, it isn't enough to have one set of texture coordinates per vertex. You need to have one set for each texture unit for each vertex. Instead of using **glTexCoord*()**, you must use **glMultiTexCoord*()**, which specifies the texture unit, as well as the texture coordinates.

void **glMultiTexCoord**{**1234**}{**sifd**}(GLenum *texUnit*, *TYPE coords*);
void **glMultiTexCoord**{**1234**}{**sifd**}**v**(GLenum *texUnit*, const *TYPE* **coords*);

Sets the texture-coordinate data (*s, t, r, q*) in *coords* for use with the texture unit *texUnit*. The enumerated values for *texUnit* are the same as for **glActiveTexture()**.

In Example 9-11, a triangle is given the two sets of texture coordinates necessary for multitexturing with two active texture units.

**Example 9-11** Specifying Vertices for Multitexturing

```
glBegin(GL_TRIANGLES);
glMultiTexCoord2f(GL_TEXTURE0, 0.0, 0.0);
glMultiTexCoord2f(GL_TEXTURE1, 1.0, 0.0);
glVertex2f(0.0, 0.0);
glMultiTexCoord2f(GL_TEXTURE0, 0.5, 1.0);
glMultiTexCoord2f(GL_TEXTURE1, 0.5, 0.0);
glVertex2f(50.0, 100.0);
glMultiTexCoord2f(GL_TEXTURE0, 1.0, 0.0);
glMultiTexCoord2f(GL_TEXTURE1, 1.0, 1.0);
glVertex2f(100.0, 0.0);
glEnd();
```

**Note:** If you are multitexturing and you use **glTexCoord*()**, you are setting the texture coordinates for the first texture unit. In other words, using **glTexCoord*()** is equivalent to using **glMultiTexCoord*(GL_TEXTURE0,...)**.

In the rare case that you are multitexturing a bitmap or image rectangle, you need to associate several texture coordinates with each raster position. Therefore, you must call **glMultiTexCoord*()** several times, once for each active texture unit, for each **glRasterPos*()** or **glWindowPos*()** call. (Since

there is only one current raster position for the entire bitmap or image rectangle, there is only one corresponding texture coordinate per unit, so the aesthetic possibilities are extremely limited.)

### Other Methods of Texture-Coordinate Specification

Explicitly calling **glMultiTexCoord*()** is only one of three ways to specify texture coordinates when multitexturing. The other two ways are to use automatic texture-coordinate generation (with **glTexGen*()**) or vertex arrays (with **glTexCoordPointer()**).

If you are multitexturing and using automatic texture-coordinate generation, then **glActiveTexture()** directs which texture unit is affected by the following automatic texture-coordinate generation routines:

- **glTexGen*(...)**
- **glEnable(GL_TEXTURE_GEN_*)**
- **glDisable(GL_TEXTURE_GEN_*)**

If you are multitexturing and specifying texture coordinates in vertex arrays, then **glClientActiveTexture()** directs the texture unit for which **glTexCoordPointer()** specifies its texture-coordinate data.

void **glClientActiveTexture**(GLenum *texUnit*);

Selects the current texture unit for specifying texture-coordinate data with vertex arrays. *texUnit* is a symbolic constant of the form GL_TEXTURE*i*, with the same values that are used for **glActiveTexture()**.

### Reverting to a Single Texture Unit

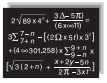
If you are using multitexturing and want to return to a single texture unit, then you need to disable texturing for all units, except for texture unit 0, with code as shown in Example 9-12.

**Example 9-12** Reverting to Texture Unit 0

```
/*  disable texturing for other texture units  */
glActiveTexture (GL_TEXTURE1);
glDisable (GL_TEXTURE_2D);
glActiveTexture (GL_TEXTURE2);
glDisable (GL_TEXTURE_2D);
/*  make texture unit 0 current  */
glActiveTexture (GL_TEXTURE0);
```

## Texture Combiner Functions

**Advanced**


Advanced

OpenGL has evolved from its early focus on vertex processing (transformation, clipping) toward more concern with rasterization and fragment operations. Texturing functionality is increasingly exposed to the programmer to improve fragment processing.

In addition to multipass texture techniques, flexible texture combiner functions provide the programmer with finer control over mixing fragments with texture or other color values. Texture combiner functions support high-quality texture effects, such as bump mapping, more realistic specular lighting, and texture fade effects (such as interpolating between two textures). A combiner function takes color and alpha data from up to three sources and processes them, generating RGBA values as output for subsequent operations.

**glTexEnv*()** is used extensively to configure combiner functions. In "Texture Functions," you encountered an abbreviated description of **glTexEnv*()**, and now here's the complete description:

---

void **glTexEnv**{if}(GLenum *target*, GLenum *pname*, *TYPE param*);
void **glTexEnv**{if}**v**(GLenum *target*, GLenum *pname*, const *TYPE* **param*);

---

Sets the current texturing function. *target* must be either GL_TEXTURE_FILTER_CONTROL or GL_TEXTURE_ENV.

If *target* is GL_TEXTURE_FILTER_CONTROL, then *pname* must be GL_TEXTURE_LOD_BIAS, and *param* is a single, floating-point value used to bias the mipmapping level-of-detail parameter.

If *target* is GL_TEXTURE_ENV, acceptable values for the second and third arguments (*pname* and *param)* are listed in Table 9-8. If *pname* is GL_TEXTURE_ENV_MODE, *param* specifies how texture values are combined with the color values of the fragment being processed. Several environment modes (GL_BLEND, GL_COMBINE, GL_COMBINE_RGB, and GL_COMBINE_ALPHA) determine whether other environment modes are useful.

If the texture environment mode is GL_BLEND, then the GL_TEXTURE_ENV_COLOR setting is used.

If the texture environment mode is GL_COMBINE, then the GL_COMBINE_RGB, GL_COMBINE_ALPHA, GL_RGB_SCALE, or GL_ALPHA_SCALE parameters are also used. For the GL_COMBINE_RGB function, the GL_SOURCE*i*_RGB and GL_OPERAND*i*_RGB parameters (where *i* is 0, 1, or 2) also may be specified. Similarly for the GL_COMBINE_ALPHA function, GL_SOURCE*i*_ALPHA and GL_OPERAND*i*_ALPHA may be specified.

| glTexEnv pname | glTexEnv param |
|---|---|
| GL_TEXTURE_ENV_MODE | GL_DECAL, GL_REPLACE, GL_MODULATE, GL_BLEND, GL_ADD, or GL_COMBINE |
| GL_TEXTURE_ENV_COLOR | array of 4 floating-point numbers: (R, G, B, A) |
| GL_COMBINE_RGB | GL_REPLACE, GL_MODULATE, GL_ADD, GL_ADD_SIGNED, GL_INTERPOLATE, GL_SUBTRACT, GL_DOT3_RGB, or GL_DOT3_RGBA |
| GL_COMBINE_ALPHA | GL_REPLACE, GL_MODULATE, GL_ADD, GL_ADD_SIGNED, GL_INTERPOLATE, or GL_SUBTRACT |
| GL_SRC*i*_RGB or GL_SRC*i*_ALPHA (where *i* is 0, 1, or 2) | GL_TEXTURE, GL_TEXTURE*n* (where *n* denotes the *n*th texture unit and multitexturing is enabled), GL_CONSTANT, GL_PRIMARY_COLOR, or GL_PREVIOUS |
| GL_OPERAND*i*_RGB (where *i* is 0, 1, or 2) | GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR, GL_SRC_ALPHA, or GL_ONE_MINUS_SRC_ALPHA |
| GL_OPERAND*i*_ALPHA (where *i* is 0, 1, or 2) | GL_SRC_ALPHA, or GL_ONE_MINUS_SRC_ALPHA |
| GL_RGB_SCALE | floating-point color scaling factor |
| GL_ALPHA_SCALE | floating-point alpha scaling factor |

**Table 9-8** Texture Environment Parameters If *target* Is GL_TEXTURE_ENV

Here are the steps for using combiner functions. If you are multitexturing, you may use a different combiner function for every texture unit and thus repeat these steps for each unit.

- To use any combiner function, you must call

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);
```

- You should specify how you want RGB or alpha values to be combined (see Table 9-9). For instance, Example 9-13 directs the current texture unit to subtract RGB and alpha values of one source from another source.

**Example 9-13** Setting the Programmable Combiner Functions

```
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_SUBTRACT);
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_ALPHA, GL_SUBTRACT);
```

| glTexEnv param | Combiner Function |
|---|---|
| GL_REPLACE | *Arg0* |
| GL_MODULATE (default) | *Arg0 * Arg1* |
| GL_ADD | *Arg0 + Arg1* |
| GL_ADD_SIGNED | *Arg0 + Arg1 – 0.5* |
| GL_INTERPOLATE | *Arg0 * Arg2 + Arg1 * (1 – Arg2)* |
| GL_SUBTRACT | *Arg0 – Arg1* |
| GL_DOT3_RGB<br>GL_DOT3_RGBA | $4 * ((Arg0_r - 0.5) * (Arg1_r - 0.5) +$<br>$(Arg0_g - 0.5) * (Arg1_g - 0.5) +$<br>$(Arg0_b - 0.5) * (Arg1_b - 0.5))$ |

**Table 9-9** GL_COMBINE_RGB and GL_COMBINE_ALPHA Functions

**Note:** GL_DOT3_RGB and GL_DOT3_RGBA are used only for GL_COMBINE_RGB and not used for GL_COMBINE_ALPHA.

The GL_DOT3_RGB and GL_DOT3_RGBA modes differ subtly. With GL_DOT3_RGB, the same dot product is placed into all three (R, G, B) values. For GL_DOT3_RGBA, the result is placed into all four (R, G, B, A).

- Specify the source for the *i*th argument of the combiner function with the constant GL_SOURCE*i*_RGB. The number of arguments (up to three) depends upon the type of function chosen. As shown in Table 9-9, GL_SUBTRACT requires two arguments, which may be set with the following code:

**Example 9-14** Setting the Combiner Function Sources

```
glTexEnvf(GL_TEXTURE_ENV, GL_SRC0_RGB, GL_TEXTURE);
glTexEnvf(GL_TEXTURE_ENV, GL_SRC1_RGB, GL_PREVIOUS);
```

When *pname* is GL_SOURCE*i*_RGB, these are your options for *param* along with how the source is determined:

- GL_TEXTURE—the source for the *i*th argument is the texture of the current texture unit
- GL_TEXTURE*n*—the texture associated with texture unit *n*. (If you use this source, texture unit *n* must be enabled and valid, or the result will be undefined.)
- GL_CONSTANT—the constant color set with GL_TEXTURE_ENV_COLOR
- GL_PRIMARY_COLOR—the incoming fragment to texture unit 0, which is the fragment color, prior to texturing
- GL_PREVIOUS—the incoming fragment from the previous texture unit (for texture unit 0, this is the same as GL_PRIMARY_COLOR)

If you suppose that the GL_SUBTRACT combiner code in Example 9-14 is set for texture unit 2, then the output from texture unit 1 (GL_PREVIOUS, *Arg1*) is subtracted from texture unit 2 (GL_TEXTURE, *Arg0*).

- Specify which values (RGB or alpha) of the sources are used and how they are used:
  - GL_OPERAND*i*_RGB matches the corresponding GL_SOURCE*i*_RGB and determines the color values for the current GL_COMBINE_RGB function. If GL_OPERAND*i*_RGB is *pname*, then *param* must be one of GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR, GL_SRC_ALPHA, or GL_ONE_MINUS_SRC_ALPHA.
  - Similarly, GL_OPERAND*i*_ALPHA matches the corresponding GL_SOURCE*i*_ALPHA and determines the alpha values for the current GL_COMBINE_ALPHA function. However, *param* is limited to either GL_SRC_ALPHA or GL_ONE_MINUS_SRC_ALPHA.

When GL_SRC_ALPHA is used for the GL_COMBINE_RGB function, the alpha values for the combiner source are interpreted as R, G, B values. In Example 9-15, the three R, G, B components for *Arg2* are (0.4, 0.4, 0.4).

**Example 9-15** Using an Alpha Value for RGB Combiner Operations

```
static GLfloat constColor[4] = {0.1, 0.2, 0.3, 0.4};
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, constColor);
glTexEnvf(GL_TEXTURE_ENV, GL_SRC2_RGB, GL_CONSTANT);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND2_RGB, GL_SRC_ALPHA);
```

In Example 9-15, if the operand had instead been GL_SRC_COLOR, the RGB components would be (0.1, 0.2, 0.3). For GL_ONE_MINUS* modes, a value's complement (either 1–color or 1–alpha) is used for combiner calculations. In Example 9-15, if the operand is GL_ONE_MINUS_SRC_COLOR, the RGB components are (0.9, 0.8, 0.7). For GL_ONE_MINUS_SRC_ALPHA, the result is (0.6, 0.6, 0.6).

- Optionally choose RGB or alpha scaling factors. The defaults are

```
glTexEnvf(GL_TEXTURE_ENV, GL_RGB_SCALE, 1.0);
glTexEnvf(GL_TEXTURE_ENV, GL_ALPHA_SCALE, 1.0);
```

- Finally draw the geometry, ensuring vertices have associated texture coordinates.

## The Interpolation Combiner Function

The interpolation function helps illustrate texture combiners, because it uses the maximum number of arguments and several source and operand modes. Example 9-16 is a portion of the sample program combiner.c.

**Example 9-16** Interpolation Combiner Function: combiner.c

```
/* for use as constant texture color */
static GLfloat constColor[4] = {0.0, 0.0, 0.0, 0.0};

constColor[3] = 0.2;
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, constColor);
glBindTexture(GL_TEXTURE_2D, texName[0]);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_INTERPOLATE);
glTexEnvf(GL_TEXTURE_ENV, GL_SRC0_RGB, GL_TEXTURE);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND0_RGB, GL_SRC_COLOR);
glTexEnvf(GL_TEXTURE_ENV, GL_SRC1_RGB, GL_PREVIOUS);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND1_RGB, GL_SRC_COLOR);
glTexEnvf(GL_TEXTURE_ENV, GL_SRC2_RGB, GL_CONSTANT);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND2_RGB, GL_SRC_ALPHA);

/* geometry is now rendered */
```

In Example 9-16, there is only one active texture unit. Since GL_INTERPOLATE is the combiner function, there are three arguments, and they are combined with the following formula: *(Arg0 * Arg2) + (Arg1 * (1 – Arg2))*. The three arguments are as follows:

- *Arg0*, GL_TEXTURE, the texture image associated with the currently bound texture object (*texName[0]*)
- *Arg1*, GL_PREVIOUS, the result of the previous texture unit, but since this is texture unit 0, GL_PREVIOUS is the fragment prior to texturing
- *Arg2*, GL_CONSTANT, a constant color; currently (0.0, 0.0, 0.0, 0.2)

The interpolated result you get is a weighted blending of the texture image and the untextured fragment. Because GL_SRC_ALPHA is specified for GL_OPERAND2_RGB, the alpha value of the constant color (*Arg2*) serves as the weighting.

If you run the sample program combiner.c, you'll see 20 percent of the texture blended with 80 percent of a smooth-shaded polygon. combiner.c also varies the alpha value of the constant color, so you'll see the results of different weightings.

Examining the interpolation function explains why several of the OpenGL default values were chosen. The third argument for interpolation is intended as weight for the two other sources. Since interpolation is the only combiner function to use three arguments, it's safe to make GL_CONSTANT the default for *Arg2*. At first glance, it may seem odd that the default value for GL_OPERAND2_RGB is GL_SRC_ALPHA. But the interpolation weight is usually the same for all three color components, so using a single value makes sense, and taking it from the alpha value of the constant is convenient.

| glTexEnv pname | Initial Value for param |
|---|---|
| GL_SRC0_RGB | GL_TEXTURE |
| GL_SRC1_RGB | GL_PREVIOUS |
| GL_SRC2_RGB | GL_CONSTANT |
| GL_OPERAND0_RGB | GL_SRC_COLOR |
| GL_OPERAND1_RGB | GL_SRC_COLOR |
| GL_OPERAND2_RGB | GL_SRC_ALPHA |

**Table 9-10** Default Values for Some Texture Environment Modes

## Applying Secondary Color after Texturing

While applying a texture to a typical fragment, only a primary color is combined with the texel colors. The primary color may be the result of lighting calculations or **glColor*()**.

After texturing, but before fog calculations, sometimes a secondary color is also applied to a fragment. Application of a secondary color may result in a more realistic highlight on a textured object.

### Secondary Color When Lighting Is Disabled

If lighting is not enabled and the color sum mode is enabled (by **glEnable(GL_COLOR_SUM)**), then the current secondary color (set by **glSecondaryColor*()**) is added to the post-texturing fragment color.

void **glSecondaryColor3**{b s i f d ub us ui}(*TYPE r*, *TYPE g*, *TYPE b*);
void **glSecondaryColor3**{b s i f d ub us ui}**v**(const *TYPE* ***values*);

Sets the red, green, and blue values for the current secondary color. The first suffix indicates the data type for parameters: byte, short, integer, float, double, unsigned byte, unsigned short, or unsigned integer. If there is a second suffix, **v**, then *values* is a pointer to an array of values of the given data type.

**glSecondaryColor*()** accepts the same data types and interprets values the same way that **glColor*()** does. (See Table 4-1 on page 178.) Secondary colors may also be specified in vertex arrays.

### Secondary Specular Color When Lighting Is Enabled

Texturing operations are applied after lighting, but blending specular highlights with a texture's colors usually lessens the effect of lighting. As discussed earlier (in "Selecting a Lighting Model" on page 207), you can calculate two colors per vertex: a primary color, which consists of all nonspecular contributions, and a secondary color, which is a sum of all specular contributions. If specular color is separated, the secondary (specular) color is added to the fragment after the texturing calculation.

**Note:** If lighting is enabled, the secondary specular color is applied, regardless of the GL_COLOR_SUM mode, and any secondary color set by **glSecondaryColor*()** is ignored.

## Point Sprites

While OpenGL supports antialiasing of points with a point size greater than one (as set with **glPointSize()**), the visual results may not be precisely what your application requires. Point *sprites* allow better control over the shading of large points. By default, when point sprites are enabled, every fragment in the sprite is assigned the same state as the vertex that initiated the point's rendering. Point sprites modify how fragment data is generated by iterating texture coordinates across the fragments of the expanded point.

To enable point sprites, call **glEnable()** with a parameter of GL_POINT_SPRITE. This will cause OpenGL to ignore the current settings for point antialiasing. Each fragment in the point will be assigned the associated vertex data, and those values will be used in shading.

To enable the iteration of texture coordinates across the point sprite, you need to call **glTexEnv***(GL_POINT_SPRITE, GL_COORD_REPLACE, GL_TRUE). This needs to be done for each texture unit with a texture map that you want applied to the sprite. Figure 9-15 illustrates the differences between antialiased points and texture-mapped point sprites. On the left is an image of a 10-pixel antialiased point. The right image is a 10-pixel texture-mapped point sprite.

**Figure 9-15** Comparison of Antialiased Points and Textured Point Sprites

The texture coordinates for point sprites are automatically assigned by OpenGL during rasterization. Figure 9-16 illustrates how texture coordinates are assigned to the point sprite. The *s* texture coordinate increases from zero to one from left to right across the fragments of the sprite. However, for *t* texture coordinates, values are controlled by where the sprite's texture-coordinate origin is specified. Control of the origin is specified by calling

**glPointParameter()** with GL_POINT_SPRITE_COORD_ORIGIN and setting the value to either GL_LOWER_LEFT or GL_UPPER_LEFT.

When the sprite origin is specified to be the GL_LOWER_LEFT, the *t*-coordinate increases from zero to one going from bottom to top of the sprite. Conversely, when the value is specified to be GL_UPPER_LEFT, the *t*-coordinate increases from zero to one going from top to bottom.

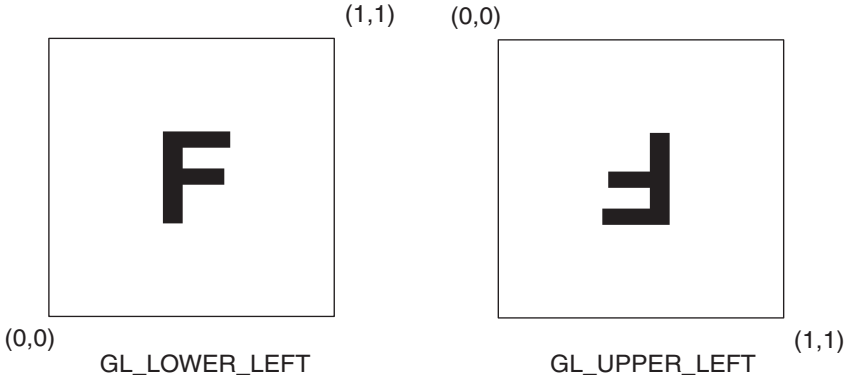



**Figure 9-16** Assignment of Texture Coordinates Based on the Setting of GL_POINT_SPRITE_COORD_ORIGIN

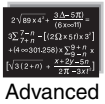
Example 9-17 demonstrates a 10-pixel-wide point sprite with an applied texture map.

**Example 9-17** Configuring a Point Sprite for Texture Mapping: sprite.c

```
glPointSize(10.0);
glTexEnvi(GL_POINT_SPRITE, GL_COORD_REPLACE, GL_TRUE);
glPointParameteri(GL_POINT_SPRITE_COORD_ORIGIN, GL_LOWER_LEFT);
glEnable(GL_POINT_SPRITE);
```

## The Texture Matrix Stack

**Advanced**


Advanced

Just as your model coordinates are transformed by a matrix before being rendered, texture coordinates are multiplied by a $4 \times 4$ matrix before any texture mapping occurs. By default, the texture matrix is the identity, so the texture coordinates you explicitly assign or those that are automatically generated remain unchanged. By modifying the texture matrix while

redrawing an object, however, you can make the texture slide over the surface, rotate around it, stretch and shrink, or any combination of the three. In fact, since the texture matrix is a completely general $4 \times 4$ matrix, effects such as perspective can be achieved.

The texture matrix is actually the top matrix on a stack, which must have a stack depth of at least two matrices. All the standard matrix-manipulation commands such as **glPushMatrix()**, **glPopMatrix()**, **glMultMatrix()**, and **glRotate*()** can be applied to the texture matrix. To modify the current texture matrix, you need to set the matrix mode to GL_TEXTURE, as follows:

```
glMatrixMode(GL_TEXTURE); /* enter texture matrix mode */
glRotated(...);
/* ... other matrix manipulations ... */
glMatrixMode(GL_MODELVIEW); /* back to modelview mode */
```

### The *q*-Coordinate

The mathematics of the fourth texture coordinate, $q$, are similar to the $w$-coordinate of the $(x, y, z, w)$ object coordinates. When the four texture coordinates $(s, t, r, q)$ are multiplied by the texture matrix, the resulting vector $(s', t', r', q')$ is interpreted as homogeneous texture coordinates. In other words, the texture map is indexed by $s'/q'$, $t'/q'$, and $r'/q'$.

You can make use of $q$ in cases where more than one projection or perspective transformation is needed. For example, suppose you want to model a spotlight that has some nonuniform pattern—brighter in the center, perhaps, or noncircular, because of flaps or lenses that modify the shape of the beam. You can emulate shining such a light onto a flat surface by making a texture map that corresponds to the shape and intensity of a light, and then projecting it onto the surface in question using projection transformations. Projecting the cone of light onto surfaces in the scene requires a perspective transformation ($q \neq 1$), since the lights might shine on surfaces that aren't perpendicular to them. A second perspective transformation occurs because the viewer sees the scene from a different (but perspective) point of view. (See Plate 28 for an example; and see "Fast Shadows and Lighting Effects Using Texture Mapping" by Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli, SIGGRAPH 1992 Proceedings [*Computer Graphics*, 26:2, July 1992, pp. 249–252] for more details.)

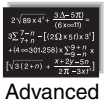
Another example might arise if the texture map to be applied comes from a photograph that itself was taken in perspective. As with spotlights, the final view depends on the combination of two perspective transformations.

### Nate Robins' Texture Tutorial

In Nate Robins' **texture** tutorial, you can use the popup menu to view the $4 \times 4$ texture matrix, make changes in matrix values, and then see their effects.

## Depth Textures

**Advanced**


Advanced

After lighting a surface (see Chapter 5), you'll soon notice that OpenGL light sources don't cast shadows. The color at each vertex is calculated without regard to any other objects in the scene. To have shadows, you need to determine and record which surfaces (or portions of the surfaces) are occluded from a direct path to a light source.

A multipass technique using depth textures provides a solution to rendering shadows. If you temporarily move the viewpoint to the light source position, you notice that everything you see is lit—there are no shadows from that perspective. A depth texture provides the mechanism to save the depth values for all "unshadowed" fragments in a *shadow map*. As you render your scene, if you compare each incoming fragment to the corresponding depth value in the shadow map, you can choose what to render, depending upon whether it is or isn't shadowed. The idea is similar to the depth test, except that it's done from the point of view of the light source.

The condensed description is as follows:

1. Render the scene from the point of view of the light source. It doesn't matter how the scene looks; you only want the depth values. Create a shadow map by capturing the depth buffer values and storing them in a texture map (shadow map).
2. Generate texture coordinates with (*s*, *t*) coordinates referencing locations within the shadow map, with the third texture coordinate (*r*), as the distance from the light source. Then draw the scene a second time, comparing the *r* value with the corresponding depth texture value to determine whether the fragment is lit or in shadow.

The following sections provide a more detailed discussion, along with sample code illustrating each of the steps.

## Creating a Shadow Map

The first step is to create a texture map of depth values. You create this by rendering the scene with the viewpoint positioned at the light source's position. Example 9-18 calls **glGetLightfv()** to obtain the current light source position, calculates an up-vector, and then uses it as the viewing transformation.

Example 9-18 begins by setting the viewport size to match that of the texture map. It then sets up the appropriate projection and viewing matrices. The objects for the scene are rendered, and the resulting depth image is copied into texture memory for use as a shadow map. Finally, the viewport is reset to it's original size and position.

Note a few more points:

- The projection matrix controls the shape of the light's "lampshade." The variables *lightFovy* and *lightAspect* in the **gluPerspective()** control the size of the lampshade. A small *lightFovy* value will be more like a spotlight, and a larger value will be more like a floodlight.
- The near and far clipping planes for the light (*lightNearPlane* and *lightFarPlane*) are used to control the precision of the depth values. Try to keep the separation between the near and far planes as small as possible to maximize the precision of the values.
- After the depth values have been established in the depth buffer, you want to capture them and put them into a GL_DEPTH_COMPONENT format texture map. Example 9-18 uses **glCopyTexImage2D()** to make a texture image from the depth buffer contents. As with any texture, ensure that the image width and height are powers of two.

**Example 9-18** Rendering Scene with Viewpoint at Light Source: shadowmap.c

```
GLint    viewport[4];
GLfloat  lightPos[4];

glGetLightfv(GL_LIGHT0, GL_POSITION, lightPos);
glGetIntegerv(GL_VIEWPORT, viewport);

glViewport(0, 0, SHADOW_MAP_WIDTH, SHADOW_MAP_HEIGHT);

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glMatrixMode(GL_PROJECTION);
glPushMatrix();
```

```
glLoadIdentity();
gluPerspective(lightFovy, lightAspect, lightNearPlane,
                 lightFarPlane);
glMatrixMode(GL_MODELVIEW);

glPushMatrix();
glLoadIdentity();
gluLookAt(lightPos[0], lightPos[1], lightPos[2],
    lookat[0], lookat[1], lookat[2],
    up[0], up[1], up[2]);
drawObjects();
glPopMatrix();

glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);

glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, 0, 0,
    SHADOW_MAP_WIDTH, SHADOW_MAP_HEIGHT, 0);

glViewport(viewport[0], viewport[1],
    viewport[2], viewport[3]);
```

## Generating Texture Coordinates and Rendering

Now use **glTexGen*()** to automatically generate texture coordinates that compute the eye-space distance from the light source position. The value of the *r* coordinate should correspond to the distance from the primitives to the light source. You can do this by using the same projection and viewing transformations that you used to create the shadow map. Example 9-19 uses the GL_MODELVIEW matrix stack to do all the matrix computations.

Note that the generated (*s, t, r, q*) texture coordinates and the depth values in the shadow map are not similarly scaled. The texture coordinates are generated in eye coordinates, so they fall in the range [–1, 1]. The depth values in the texels are within [0, 1]. Therefore, an initial translation and scaling maps the texture coordinates into the same range of values as the shadow map.

**Example 9-19** Calculating Texture Coordinates: shadowmap.c

```
GLfloat  tmpMatrix[16];

glMatrixMode(GL_MODELVIEW);
glPushMatrix();
```

```
glLoadIdentity();
glTranslatef(0.5, 0.5, 0.0);
glScalef(0.5, 0.5, 1.0);
gluPerspective(lightFovy, lightAspect,
    lightNearPlane, lightFarPlane);
gluLookAt(lightPos[0], lightPos[1], lightPos[2],
     lookat[0], lookat[1], lookat[2],
     up[0], up[1], up[2]);
glGetFloatv(GL_MODELVIEW_MATRIX, tmpMatrix);
glPopMatrix();

transposeMatrix(tmpMatrix);

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGeni(GL_Q, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);

glTexGenfv(GL_S, GL_OBJECT_PLANE, &tmpMatrix[0]);
glTexGenfv(GL_T, GL_OBJECT_PLANE, &tmpMatrix[4]);
glTexGenfv(GL_R, GL_OBJECT_PLANE, &tmpMatrix[8]);
glTexGenfv(GL_Q, GL_OBJECT_PLANE, &tmpMatrix[12]);

glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
glEnable(GL_TEXTURE_GEN_Q);
```

In Example 9-20, before the scene is rendered for the second and final time, the texture comparison mode GL_COMPARE_R_TO_TEXTURE instructs OpenGL to compare the fragment's *r*-coordiate with the texel value. If the *r* distance is less than or equal to (the comparison function GL_LEQUAL) the texel value, there is nothing between this fragment and the light source, and it is effectively treated as having a luminance value of one. If the comparison fails, then there is another primitive between this fragment and the light source, so this fragment is shadowed and has an effective luminance of zero.

**Example 9-20** Rendering Scene Comparing *r* Coordinate: shadowmap.c

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
    GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC,
    GL_LEQUAL);
glTexParameteri(GL_TEXTURE_2D, GL_DEPTH_TEXTURE_MODE,
    GL_LUMINANCE);
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
     GL_COMPARE_R_TO_TEXTURE);
glEnable(GL_TEXTURE_2D);
```

This technique can produce some unintended visual artifacts:

- Self-shadowing, whereby an object incorrectly casts a shadow upon itself, is a common problem.
- Aliasing of the projected texture, particularly in the regions farthest from the light sources, can occur. Using higher resolution shadow maps can help reduce the aliasing.
- GL_MODULATE mode, when used with depth texturing, may cause sharp transitions between shadowed and unshadowed regions.

Unfortunately, there are no steadfast rules for overcoming these issues. Some experimentation may be required to produce the best-looking image.