Reproduced from:
OpenGL Programming Guide, 2nd Edition
OpenGL Architecture Review Board
Addison-Wesley, 1998

## Multiple Levels of Detail


**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. For example, to render a brick wall, you may use a large (say 128×128 texel) texture image when it 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 9-4. 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 *multim im parvo*, meaning "many things in a small place." Mipmapping uses some clever methods to pack image data into memory.

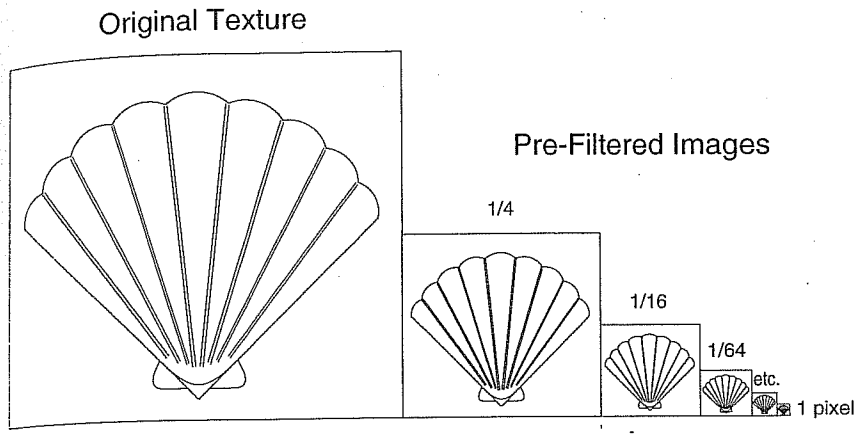


**Figure 9-4** Mipmaps

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 1×1 map. For example, if your highest-resolution map is 64×16, you must also provide maps of size 32×8, 16×4, 8×2, 4×1, 2×1, and 1×1. 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 four texels in the larger texture. (Since OpenGL doesn't require any particular method for calculating the smaller maps, the differently sized textures could be totally unrelated. In practice, unrelated textures would make the transitions between mipmaps extremely noticeable.)

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 largest texture of size 64×16 would be declared with *level* = 0, the 32×8 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 the next section.

Example 9-4 illustrates the use of a series of six texture maps decreasing in size from 32×32 to 1×1. 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 9-4** Mipmap Textures: mipmap.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>

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_DECAL);
   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();
}

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, 30000.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

void keyboard (unsigned char key, int x, int y)
{
   switch (key) {
      case 27:
         exit(0);
         break;
      default:
         break;
   }
```

```
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutInitWindowPosition(50, 50);
   glutCreateWindow(argv[0]);
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

Example 9-4 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 so that the transition is as smooth as possible. Thus, the maps of lower resolution are usually filtered versions of an original, high-resolution 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. However, since mipmap construction is such an important operation, however, the OpenGL Utility Library contains two routines that aid in the manipulation of images to be used as mipmapped textures.

Assuming you have constructed the level 0, or highest-resolution map, the routines **gluBuild1DMipmaps()** and **gluBuild2DMipmaps()** construct and define the pyramid of mipmaps down to a resolution of $1 \times 1$ (or 1, for one-dimensional texture maps). 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.

int **gluBuild1DMipmaps**(GLenum *target*, GLint *components*, GLint *width*, GLenum *format*, GLenum *type*, void **data*);
int **gluBuild2DMipmaps**(GLenum *target*, GLint *components*, GLint *width*, GLint *height*, GLenum *format*, GLenum *type*, void **data*);

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

## 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 9-5. 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.

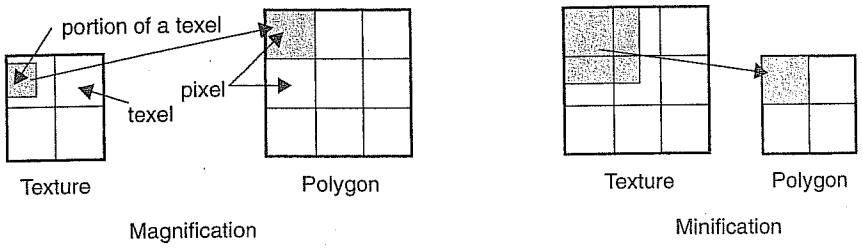


**Figure 9-5** Texture Magnification and Minification

In some cases, it isn't obvious whether magnification or minification is called for. If the mipmap needs to be stretched (or shrunk) in both the *x* and *y* directions, then magnification (or minification) is needed. If the mipmap 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 358.)

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 either GL_TEXTURE_2D or GL_TEXTURE_1D, depending on whether you're working with two- or one-dimensional textures. 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 9-1 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 9-1** 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 2×2 array of texels that lie nearest to the center of the pixel is used, again for both magnification and minification. When the texture coordinates are near the edge of the texture map, the nearest 2×2 array of texels might include some that are outside the texture map. In these cases, the texel values used depend on whether GL_REPEAT or GL_CLAMP is in effect and whether you've assigned a border for the texture. (See "Using a Texture's Borders" on page 337.) 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, the largest texture map (*level* = 0) is always 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, the largest texture map is used.)

As shown in Table 9-1, four additional filtering choices 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 to linearly interpolate 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 smoothest results, but it requires the most computation and therefore might be the slowest.

## Texture Objects

Texture objects are an important new feature in release 1.1 of OpenGL. A texture object stores texture data and makes it readily available. You can now 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.

## Texture Functions

In all the examples 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 that the surface would be rendered without texturing, or to blend the color in the texture map with the original color of the surface. You choose one of four 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*, *TYPE *param*);

---

Sets the current texturing function. *target* must be GL_TEXTURE_ENV. If *pname* is GL_TEXTURE_ENV_MODE, *param* can be GL_DECAL, GL_REPLACE, GL_MODULATE, or GL_BLEND, to specify how texture values are to be combined with the color values of the fragment being processed. If *pname* is GL_TEXTURE_ENV_COLOR, *param* is an array of four floating-point values representing R, G, B, and A components. These values are used only if the GL_BLEND texture function has been specified as well.

The combination of the texturing function and the base internal format determine 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.

Table 9-2 and Table 9-3 show how the texturing function and base internal format determine the texturing application formula used for each component of the texture. There are six base internal formats (the letters in parentheses represent their values in the tables): GL_ALPHA (A), GL_LUMINANCE (L), GL_LUMINANCE_ALPHA (L and A), GL_INTENSITY (I), GL_RGB (C), and GL_RGBA (C and A). Other internal formats specify

desired resolutions of the texture components and can be matched to one of these six base internal formats.

| Base Internal Format | Replace Texture Function | Modulate Texture Function |
|---|---|---|
| GL_ALPHA | $C = C_f$,<br>$A = A_t$ | $C = C_f$,<br>$A = A_f A_t$ |
| GL_LUMINANCE | $C = L_t$,<br>$A = A_f$ | $C = C_f L_t$,<br>$A = A_f$ |
| GL_LUMINANCE_ALPHA | $C = L_t$,<br>$A = A_t$ | $C = C_f L_t$,<br>$A = A_f A_t$ |
| GL_INTENSITY | $C = I_t$,<br>$A = I_t$ | $C = C_f I_t$,<br>$A = A_f I_t$ |
| GL_RGB | $C = C_t$,<br>$A = A_f$ | $C = C_f C_t$,<br>$A = A_f$ |
| GL_RGBA | $C = C_t$,<br>$A = A_t$ | $C = C_f C_t$,<br>$A = A_f A_t$ |

**Table 9-2** Replace and Modulate Texture Functions

| Base Internal Format | Decal Texture Function | Blend Texture Function |
|---|---|---|
| GL_ALPHA | undefined | $C = C_f$,<br>$A = A_f A_t$ |
| GL_LUMINANCE | undefined | $C = C_f(1-L_t) + C_c L_t$,<br>$A = A_f$ |
| GL_LUMINANCE_ALPHA | undefined | $C = C_f(1-L_t) + C_c L_t$,<br>$A = A_f A_t$ |
| GL_INTENSITY | undefined | $C = C_f(1-I_t) + C_c I_t$,<br>$A = A_f(1-I_t) + A_c I_t$ |
| GL_RGB | $C = C_t$,<br>$A = A_f$ | $C = C_f(1-C_t) + C_c C_t$,<br>$A = A_f$ |
| GL_RGBA | $C = C_f(1-A_t) + C_t A_t$,<br>$A = A_f$ | $C = C_f(1-C_t) + C_c C_t$,<br>$A = A_f A_t$ |

**Table 9-3** Decal and Blend Texture Functions

**Note:** In Table 9-2 and Table 9-3, a subscript of t indicates a texture value, f indicates the incoming fragment value, c indicates the values assigned with GL_TEXTURE_ENV_COLOR, and no subscript indicates the final, computed value. Also in the 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 the corresponding component of the first.

The decal texture function makes sense only for the RGB and RGBA internal formats (remember that texture mapping doesn't work in color-index mode).-With the RGB internal format, the color that would have been painted in the absence of any texture mapping (the fragment's color) is replaced by the texture color, and its alpha is unchanged. 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. You use the decal texture function in situations where you want to apply an opaque texture to an object—if you were drawing a soup can with an opaque label, for example. The decal texture function also can be used to apply an alpha blended texture, such as an insignia onto an airplane wing.

The replacement texture function is similar to decal; in fact, for the RGB internal format, they are exactly the same. With all the internal formats, the component values are either replaced or left alone.

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 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" on page 217 for the billboarding example, which uses a blended texture.)

## 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 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 also 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*. In Release 1.1, the *r* coordinate is ignored. (Some implementations have 3D texture mapping as an extension, and that extension uses the *r* coordinate.) 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" on page 372. 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" on page 360.

void **glTexCoord**{1234}{sifd}(*TYPE coords*);
void **glTexCoord**{1234}{sifd}v(*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 4×4 texture matrix before any texture mapping occurs. (See "The Texture Matrix Stack" on page 371.) Note that integer texture coordinates are interpreted directly rather than being mapped to the range [–1,1] as normal coordinates are.

The next section 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 364.)

## 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 is to represent 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 thirty 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 thirty 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 9-6 shows two tori, 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).

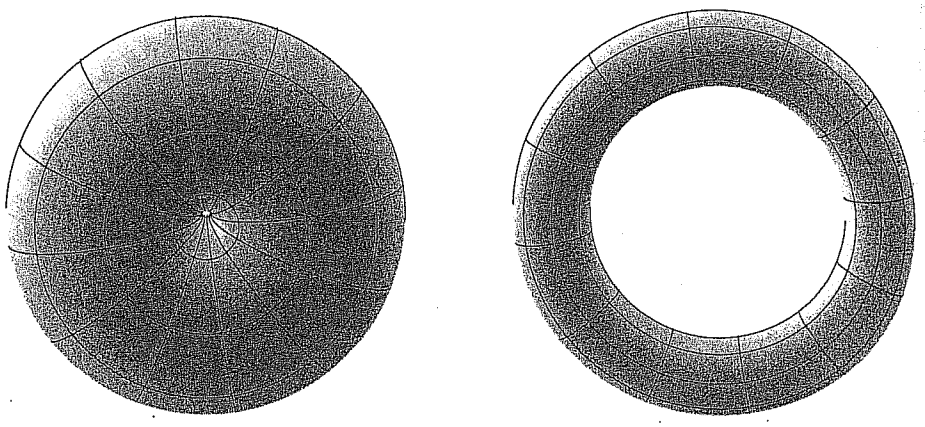

**Figure 9-6** Texture-Map Distortion

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 successfully mapping 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 part of texture coordinates is ignored, and copies of the texture map tile the surface. For most applications where 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.

The other 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 where you want a single copy of the texture to appear on a large surface. If the surface-texture coordinates range from 0.0 to 10.0 in both directions, one copy of the texture appears in the lower corner of the surface. If you've chosen GL_LINEAR as the filtering method

(see "Filtering" on page 344), an equally weighted combination of the border color and the texture color is used, as follows.

- When repeating, the 2×2 array wraps to the opposite edge of the texture. Thus, texels on the right edge are averaged with those on the left, and top and bottom texels are also averaged.
- If there is a border, then the texel from the border is used in the weighting. Otherwise, GL_TEXTURE_BORDER_COLOR is used. (If you've chosen GL_NEAREST as the filtering method, the border color is completely ignored.)

Note that 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" on page 214.)

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 3.0 as follows:

```
glBegin(GL_QUADS);
  glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
  glTexCoord2f(0.0, 3.0); glVertex3f(-2.0, 1.0, 0.0);
  glTexCoord2f(3.0, 3.0); glVertex3f(0.0, 1.0, 0.0);
  glTexCoord2f(3.0, 0.0); glVertex3f(0.0, -1.0, 0.0);

  glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
  glTexCoord2f(0.0, 3.0); glVertex3f(1.0, 1.0, 0.0);
  glTexCoord2f(3.0, 3.0); glVertex3f(2.41421, 1.0, -1.41421);
  glTexCoord2f(3.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
glEnd();
```

With GL_REPEAT wrapping, the result is as shown in Figure 9-7.

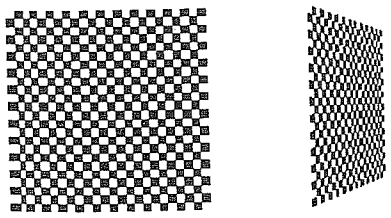

**Figure 9-7** Repeating a Texture

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);
```

If GL_CLAMP is used instead of GL_REPEAT for each direction, you see something similar to Figure 9-8.

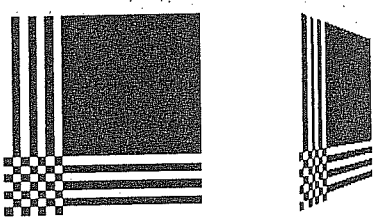

**Figure 9-8** Clamping a Texture

You can also clamp in one direction and repeat in the other, as shown in Figure 9-9.

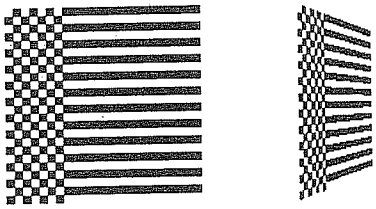

**Figure 9-9** Repeating and Clamping a Texture

You've now seen all the possible arguments for **glTexParameter***(), which is summarized here.

void **glTexParameter**{if}(GLenum *target*, GLenum *pname*, *TYPE param*);
void **glTexParameter**{if}v(GLenum *target*, GLenum *pname*,
*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 either GL_TEXTURE_2D or GL_TEXTURE_1D to indicate a two- or one-dimensional texture. The possible values for *pname* and *param* are shown in Table 9-4. 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 according to Table 4-1 on page 164; they're also clamped to the range [0,1].

| Parameter | Values |
|---|---|
| GL_TEXTURE_WRAP_S | GL_CLAMP, GL_REPEAT |
| GL_TEXTURE_WRAP_T | GL_CLAMP, GL_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 |

**Table 9-4** glTexParameter*() Parameters