Chapter 6

Blending, Antialiasing, Fog, and Polygon Offset

Blending
- The Source and Destination Factors
- Sample Uses of Blending
- A Blending Example
- Three−Dimensional Blending with the Depth Buffer

Antialiasing
- Antialiasing Points or Lines
- Antialiasing Polygons

Fog
- Using Fog
- Fog Equations

Polygon Offset

Chapter 6
Blending, Antialiasing, Fog, and Polygon Offset

Chapter Objectives

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

- Blend colors to achieve such effects as making objects appear translucent
- Smooth jagged edges of lines and polygons with antialiasing
- Create scenes with realistic atmospheric effects
- Draw geometry at or near the same depth, but avoid unaesthetic artifacts from intersecting geometry

The preceding chapters have given you the basic information you need to create a computer−graphics scene; you've learned how to do the following:

- Draw geometric shapes
- Transform those geometric shapes so that they can be viewed from whatever perspective you wish
- Specify how the geometric shapes in your scene should be colored and shaded
- Add lights and indicate how they should affect the shapes in your scene

Now you're ready to get a little fancier. This chapter discusses four techniques that can add extra detail and polish to your scene. None of these techniques is hard to use—in fact, it's probably harder to explain them than to use them. Each of these techniques is described in its own major section:

- "Blending" tells you how to specify a blending function that combines color values from a source and a destination. The final effect is that parts of your scene appear translucent.
- "Antialiasing" explains this relatively subtle technique that alters colors so that the edges of points, lines, and polygons appear smooth rather than angular and jagged.
- "Fog" describes how to create the illusion of depth by computing the color values of an object based on its distance from the viewpoint. Thus, objects that are far away appear to fade into the background, just as they do in real life.
- If you've tried to draw a wireframe outline atop a shaded object and used the same vertices, you've probably noticed some ugly visual artifacts. "Polygon Offset" shows you how to tweak (offset) depth values to make an outlined, shaded object look beautiful.

Blending

You've already seen alpha values (alpha is the A in RGBA), but they've been ignored until now. Alpha values are specified with *glColor*()*, when using *glClearColor()* to specify a clearing color and when specifying certain lighting parameters such as a material property or light−source intensity. As you learned in Chapter 4, the pixels on a monitor screen emit red, green, and blue light, which is controlled by the red, green, and blue color values. So how does an alpha value affect what gets drawn in a window on the screen?

When blending is enabled, the alpha value is often used to combine the color value of the fragment being processed with that of the pixel already stored in the framebuffer. Blending occurs after your scene has been rasterized and converted to fragments, but just before the final pixels are drawn in the framebuffer. Alpha values can also be used in the alpha test to accept or reject a fragment based on its alpha value. (See Chapter 10 for more information about this process.)

Without blending, each new fragment overwrites any existing color values in the framebuffer, as though the fragment were opaque. With blending, you can control how (and how much of) the existing color value should be combined with the new fragment's value. Thus you can use alpha blending to create a translucent fragment that lets some of the previously stored color value "show through." Color blending lies at the heart of techniques such as transparency, digital compositing, and painting.

Note: Alpha values aren't specified in color−index mode, so blending operations aren't performed in color−index mode.

The most natural way to think of blending operations is to think of the RGB components of a fragment as representing its color and the alpha component as representing opacity. Transparent or translucent surfaces have lower opacity than opaque ones and, therefore, lower alpha values. For example, if you're viewing an object through green glass, the color you see is partly green from the glass and partly the color of the object. The percentage varies depending on the transmission properties of the glass: If the glass transmits 80 percent of the light that strikes it (that is, has an opacity of 20 percent), the color you see is a combination of 20 percent glass color and 80 percent of the color of the object behind it. You can easily imagine situations with multiple translucent surfaces. If you look at an automobile, for instance, its interior has one piece of glass between it and your viewpoint; some objects behind the automobile are visible through two pieces of glass.

The Source and Destination Factors

During blending, color values of the incoming fragment (the *source*) are combined with the color values of the corresponding currently stored pixel (the *destination*) in a two−stage process. First you specify how to compute source and destination factors. These factors are RGBA quadruplets that are multiplied by each component of the R, G, B, and A values in the source and destination, respectively. Then the corresponding components in the two sets of RGBA quadruplets are added. To show this mathematically, let the source and destination blending factors be (S_r, S_g, S_b, S_a) and (D_r, D_g, D_b, D_a), respectively, and the RGBA values of the source and destination be indicated with a subscript of s or d. Then the final, blended RGBA values are given by

$$(R_sS_r+R_dD_r, G_sS_g+G_dD_g, B_sS_b+B_dD_b, A_sS_a+A_dD_a)$$

Each component of this quadruplet is eventually clamped to [0,1].

Now consider how the source and destination blending factors are generated. You use *glBlendFunc()* to supply two constants: one that specifies how the source factor should be computed and one that indicates how the destination factor should be computed. To have blending take effect, you also need to enable it:

```
glEnable(GL_BLEND);
```

Use *glDisable()* with GL_BLEND to disable blending. Also note that using the constants GL_ONE (source) and GL_ZERO (destination) gives the same results as when blending is disabled; these values are the default.

void glBlendFunc(GLenum sfactor, GLenum dfactor);

Controls how color values in the fragment being processed (the source) are combined with those already stored in the framebuffer (the destination). The argument sfactor indicates how to compute a source blending factor; dfactor indicates how to compute a destination blending factor. The possible values for these arguments are explained in Table 6−1 The blend factors are assumed to lie in the range [0,1]; after the color values in the source and destination are combined, they're clamped to the range [0,1].

Note: In Table 6−1, the RGBA values of the source and destination are indicated with the subscripts s and d, respectively. Subtraction of quadruplets means subtracting them componentwise. The Relevant Factor column indicates whether the corresponding constant can be used to specify the source or destination blend factor.

Constant	Relevant Factor	Computed Blend Factor
GL_ZERO	source or destination	(0, 0, 0, 0)
GL_ONE	source or destination	(1, 1, 1, 1)
GL_DST_COLOR	source	(R_d, G_d, B_d, A_d)
GL_SRC_COLOR	destination	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_DST_COLOR	source	$(1, 1, 1, 1)-(R_d, G_d, B_d, A_d)$
GL_ONE_MINUS_SRC_COLOR	destination	$(1, 1, 1, 1)-(R_s, G_s, B_s, A_s)$
GL_SRC_ALPHA	source or destination	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	source or destination	$(1, 1, 1, 1)-(A_s, A_s, A_s, A_s)$
GL_DST_ALPHA	source or destination	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	source or destination	$(1, 1, 1, 1)-(A_d, A_d, A_d, A_d)$
GL_SRC_ALPHA_SATURATE	source	(f, f, f, 1); f=min(As, 1−Ad)

Table 6−1 Source and Destination Blending Factors

Sample Uses of Blending

Not all combinations of source and destination factors make sense. Most applications use a small number of combinations. The following paragraphs describe typical uses for particular combinations of source and destination factors. Some of these examples use only the incoming alpha value, so they work even when alpha values aren't stored in the framebuffer. Also note that often there's more than one way to achieve some of these effects.

One way to draw a picture composed half of one image and half of another, equally blended, is to set the source factor to GL_ONE and the destination factor to GL_ZERO, and draw the first image. Then set the source factor to GL_SRC_ALPHA and destination factor to GL_ONE_MINUS_SRC_ALPHA, and draw the second image with alpha equal to 0.5. This pair of factors probably represents the most commonly used blending operation. If the picture is supposed to be blended with 0.75 of the first image and 0.25 of the second, draw the first image as before, and draw the second with an alpha of 0.25.

To blend three different images equally, set the destination factor to GL_ONE and the source factor to GL_SRC_ALPHA. Draw each of the images with an alpha equal to 0.3333333. With this technique, each image is only one−third of its original brightness, which is noticeable where the images don't overlap.

Suppose you're writing a paint program, and you want to have a brush that gradually adds color so that each brush stroke blends in a little more color with whatever is currently in the image (say 10 percent color with 90 percent image on each pass). To do this, draw the image of the brush with alpha of 10 percent and use GL_SRC_ALPHA (source) and GL_ONE_MINUS_SRC_ALPHA (destination). Note that you can vary the alphas across the brush to make the brush add more of its color in the middle and less on the edges, for an antialiased brush shape. (See "Antialiasing.") Similarly, erasers can be implemented by setting the eraser color to the background color.

The blending functions that use the source or destination colors—GL_DST_COLOR or GL_ONE_MINUS_DST_COLOR for the source factor and GL_SRC_COLOR or GL_ONE_MINUS_SRC_COLOR for the destination factor—effectively allow you to modulate each color component individually. This operation is equivalent to applying a simple filter—for example, multiplying the red component by 80 percent, the green component by 40 percent, and the blue component by 72 percent would simulate viewing the scene through a photographic filter that blocks 20 percent of red light, 60 percent of green, and 28 percent of blue.

Suppose you want to draw a picture composed of three translucent surfaces, some obscuring others, and all over a solid background. Assume the farthest surface transmits 80 percent of the color behind it, the next transmits 40 percent, and the closest transmits 90 percent. To compose this picture, draw the background first with the default source and destination factors, and then change the blending factors to GL_SRC_ALPHA (source) and GL_ONE_MINUS_SRC_ALPHA (destination). Next, draw the farthest surface with an alpha of 0.2, then the middle surface with an alpha of 0.6, and finally the closest surface with an alpha of 0.1.

If your system has alpha planes, you can render objects one at a time (including their alpha values), read them back, and then perform interesting matting or compositing operations with the fully rendered objects. (See "Compositing 3D Rendered Images" by Tom Duff, SIGGRAPH 1985 Proceedings, p. 41-44, for examples of this technique.) Note that objects used for picture composition can come from any source—they can be rendered using OpenGL commands, rendered using techniques such as ray−tracing or radiosity that are implemented in another graphics library, or obtained by scanning in existing images.

You can create the effect of a nonrectangular raster image by assigning different alpha values to individual fragments in the image. In most cases, you would assign an alpha of 0 to each "invisible" fragment and an alpha of 1.0 to each opaque fragment. For example, you can draw a polygon in the shape of a tree and apply a texture map of foliage; the viewer can see through parts of the rectangular texture that aren't part of the tree if you've assigned them alpha values of 0. This method, sometimes called *billboarding*, is much faster than creating the tree out of three−dimensional polygons. An example of this technique is shown inFigure 6−1: The tree is a single rectangular polygon that can be rotated about the center of the trunk, as shown by the outlines, so that it's always facing the viewer. (See "Texture Functions" in Chapter 9 for more information about blending textures.)

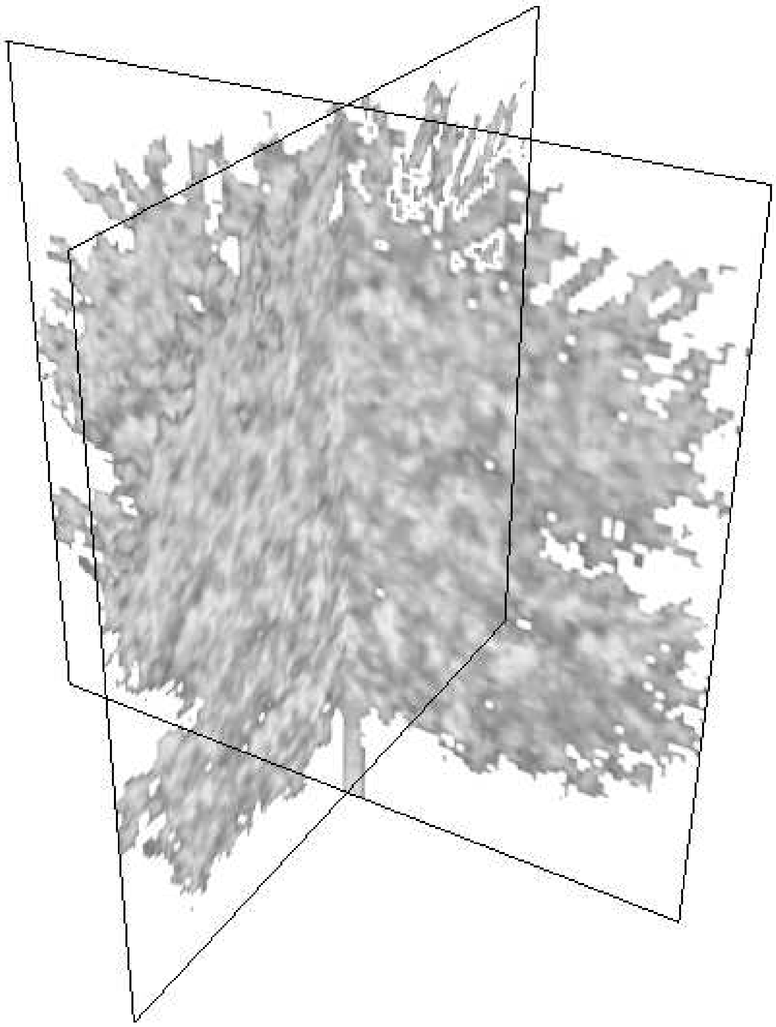

Figure 6−1 Creating a Nonrectangular Raster Image

Blending is also used for **antialiasing**, which is a rendering technique to reduce the jagged appearance of primitives drawn on a raster screen. (See "Antialiasing" for more information.)

A Blending Example

Example 6−1draws two overlapping colored triangles, each with an alpha of 0.75. Blending is enabled and the source and destination blending factors are set to GL_SRC_ALPHA and GL_ONE_MINUS_SRC_ALPHA, respectively.

When the program starts up, a yellow triangle is drawn on the left and then a cyan triangle is drawn on the right so that in the center of the window, where the triangles overlap, cyan is blended with the original yellow. You can change which triangle is drawn first by typing ‘t’ in the window.

Example 6−1 Blending Example: alpha.c

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

```
#include <stdlib.h>

static int leftFirst = GL_TRUE;

/*  Initialize alpha blending function.  */
static void init(void)
{
   glEnable (GL_BLEND);
   glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
   glShadeModel (GL_FLAT);
   glClearColor (0.0, 0.0, 0.0, 0.0);
}

static void drawLeftTriangle(void)
{
/* draw yellow triangle on LHS of screen */
   glBegin (GL_TRIANGLES);
      glColor4f(1.0, 1.0, 0.0, 0.75);
      glVertex3f(0.1, 0.9, 0.0);
      glVertex3f(0.1, 0.1, 0.0);
      glVertex3f(0.7, 0.5, 0.0);
   glEnd();
}


static void drawRightTriangle(void)
{
/* draw cyan triangle on RHS of screen */
   glBegin (GL_TRIANGLES);
      glColor4f(0.0, 1.0, 1.0, 0.75);
      glVertex3f(0.9, 0.9, 0.0);
      glVertex3f(0.3, 0.5, 0.0);
      glVertex3f(0.9, 0.1, 0.0);
   glEnd();
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);

   if (leftFirst) {
      drawLeftTriangle();
      drawRightTriangle();
   }
   else {
      drawRightTriangle();
      drawLeftTriangle();
```

```
   }
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      gluOrtho2D (0.0, 1.0, 0.0, 1.0*(GLfloat)h/(GLfloat)w);
   else
      gluOrtho2D (0.0, 1.0*(GLfloat)w/(GLfloat)h, 0.0, 1.0);
}

void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 't':
      case 'T':
         leftFirst = !leftFirst;
         glutPostRedisplay();
         break;
      case 27:  /*  Escape key  */
         exit(0);
         break;
      default:
         break;
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize (200, 200);
   glutCreateWindow (argv[0]);
   init();
   glutReshapeFunc (reshape);
   glutKeyboardFunc (keyboard);
   glutDisplayFunc (display);
   glutMainLoop();
   return 0;
}
```

The order in which the triangles are drawn affects the color of the overlapping region. When the left triangle is drawn first, cyan fragments (the source) are blended with yellow fragments, which are already in the framebuffer (the destination). When the right triangle is drawn first, yellow is blended with cyan. Because the alpha values are all 0.75, the actual blending factors become 0.75 for the source and 1.0 − 0.75 = 0.25 for the destination. In other words, the source fragments are somewhat translucent, but they have more effect on the final color than the destination fragments.

Three−Dimensional Blending with the Depth Buffer

As you saw in the previous example, the order in which polygons are drawn greatly affects the blended result. When drawing three−dimensional translucent objects, you can get different appearances depending on whether you draw the polygons from back to front or from front to back. You also need to consider the effect of the depth buffer when determining the correct order. (See "A Hidden−Surface Removal Survival Kit" in Chapter 5for an introduction to the depth buffer. Also see "Depth Test" in Chapter 10 for more information.) The depth buffer keeps track of the distance between the viewpoint and the portion of the object occupying a given pixel in a window on the screen; when another candidate color arrives for that pixel, it's drawn only if its object is closer to the viewpoint, in which case its depth value is stored in the depth buffer. With this method, obscured (or hidden) portions of surfaces aren't necessarily drawn and therefore aren't used for blending.

If you want to render both opaque and translucent objects in the same scene, then you want to use the depth buffer to perform hidden−surface removal for any objects that lie behind the opaque objects. If an opaque object hides either a translucent object or another opaque object, you want the depth buffer to eliminate the more distant object. If the translucent object is closer, however, you want to blend it with the opaque object. You can generally figure out the correct order to draw the polygons if everything in the scene is stationary, but the problem can quickly become too hard if either the viewpoint or the object is moving.

The solution is to enable depth buffering but make the depth buffer read−only while drawing the translucent objects. First you draw all the opaque objects, with the depth buffer in normal operation. Then you preserve these depth values by making the depth buffer read−only. When the translucent objects are drawn, their depth values are still compared to the values established by the opaque objects, so they aren't drawn if they're behind the opaque ones. If they're closer to the viewpoint, however, they don't eliminate the opaque objects, since the depth−buffer values can't change. Instead, they're blended with the opaque objects. To control whether the depth buffer is writable, use *glDepthMask()*; if you pass GL_FALSE as the argument, the buffer becomes read−only, whereas GL_TRUE restores the normal, writable operation.

Example 6−2demonstrates how to use this method to draw opaque and translucent three−dimensional objects. In the program, typing 'a' triggers an animation sequence in which a translucent cube moves through an opaque sphere. Pressing the 'r' key resets the objects in the scene to their initial positions. To get the best results when transparent objects overlap, draw the objects from back to front.

Example 6−2 Three−Dimensional Blending: alpha3D.c

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

#define MAXZ 8.0
```

```
#define MINZ -8.0
#define ZINC 0.4
static float solidZ = MAXZ;
static float transparentZ = MINZ;
static GLuint sphereList, cubeList;

static void init(void)
{
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 0.15 };
   GLfloat mat_shininess[] = { 100.0 };
   GLfloat position[] = { 0.5, 0.5, 1.0, 0.0 };

   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
   glLightfv(GL_LIGHT0, GL_POSITION, position);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);

   sphereList = glGenLists(1);
   glNewList(sphereList, GL_COMPILE);
      glutSolidSphere (0.4, 16, 16);
   glEndList();

   cubeList = glGenLists(1);
   glNewList(cubeList, GL_COMPILE);
      glutSolidCube (0.6);
   glEndList();
}

void display(void)
{
   GLfloat mat_solid[] = { 0.75, 0.75, 0.0, 1.0 };
   GLfloat mat_zero[] = { 0.0, 0.0, 0.0, 1.0 };
   GLfloat mat_transparent[] = { 0.0, 0.8, 0.8, 0.6 };
   GLfloat mat_emission[] = { 0.0, 0.3, 0.3, 0.6 };

   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glPushMatrix ();
      glTranslatef (-0.15, -0.15, solidZ);
      glMaterialfv(GL_FRONT, GL_EMISSION, mat_zero);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_solid);
      glCallList (sphereList);
   glPopMatrix ();
```

```
   glPushMatrix ();
      glTranslatef (0.15, 0.15, transparentZ);
      glRotatef (15.0, 1.0, 1.0, 0.0);
      glRotatef (30.0, 0.0, 1.0, 0.0);
      glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_transparent);
      glEnable (GL_BLEND);
      glDepthMask (GL_FALSE);
      glBlendFunc (GL_SRC_ALPHA, GL_ONE);
      glCallList (cubeList);
      glDepthMask (GL_TRUE);
      glDisable (GL_BLEND);
   glPopMatrix ();

   glutSwapBuffers();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLint) w, (GLint) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
               1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
   else
      glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
               1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

void animate(void)
{
   if (solidZ <= MINZ || transparentZ >= MAXZ)
      glutIdleFunc(NULL);
   else {
      solidZ -= ZINC;
      transparentZ += ZINC;
      glutPostRedisplay();
   }
}

void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
```

```
      case 'a':
      case 'A':
         solidZ = MAXZ;
         transparentZ = MINZ;
         glutIdleFunc(animate);
         break;
      case 'r':
      case 'R':
         solidZ = MAXZ;
         transparentZ = MINZ;
         glutPostRedisplay();
         break;
      case 27:
        exit(0);
    }
}

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

Antialiasing

You might have noticed in some of your OpenGL pictures that lines, especially nearly horizontal or nearly vertical ones, appear jagged. These jaggies appear because the ideal line is approximated by a series of pixels that must lie on the pixel grid. The jaggedness is called **aliasing**, and this section describes antialiasing techniques to reduce it. Figure 6−2shows two intersecting lines, both aliased and antialiased. The pictures have been magnified to show the effect.

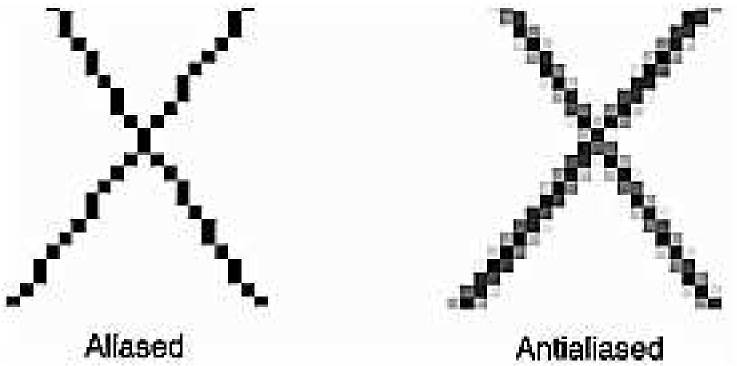

Figure 6−2 Aliased and Antialiased Lines

Figure 6−3 shows how a diagonal line 1 pixel wide covers more of some pixel squares than others. In fact, when performing antialiasing, OpenGL calculates a *coverage* value for each fragment based on the fraction of the pixel square on the screen that it would cover. The figure shows these coverage values for the line. In RGBA mode, OpenGL multiplies the fragment's alpha value by its coverage. You can then use the resulting alpha value to blend the fragment with the corresponding pixel already in the framebuffer. In color−index mode, OpenGL sets the least significant 4 bits of the color index based on the fragment's coverage (0000 for no coverage and 1111 for complete coverage). It's up to you to load your color map and apply it appropriately to take advantage of this coverage information.

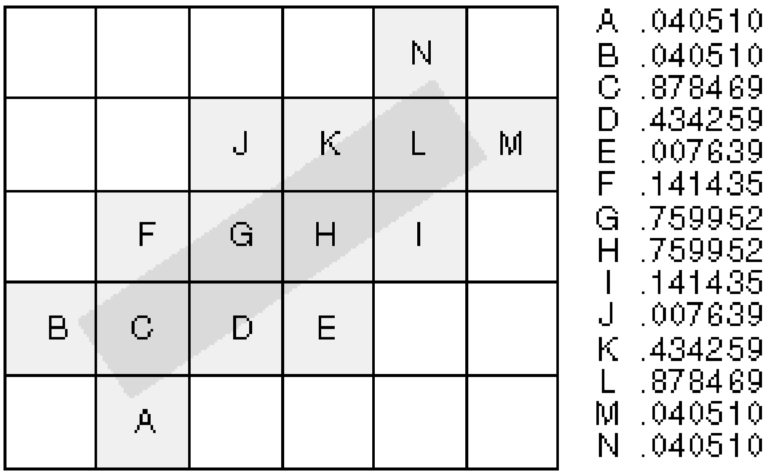

Figure 6−3 Determining Coverage Values

The details of calculating coverage values are complex, difficult to specify in general, and in fact may vary slightly depending on your particular implementation of OpenGL. You can use the *glHint()* command to exercise some control over the trade−off between image quality and speed, but not all implementations will take the hint.

void glHint(GLenum target, GLenum hint);

Controls certain aspects of OpenGL behavior. The target parameter indicates which behavior is to be controlled; its possible values are shown in Table 6−2 The hint parameter can be GL_FASTEST to indicate that the most efficient option should be chosen, GL_NICEST to indicate the highest−quality option, or GL_DONT_CARE to indicate no preference. The interpretation of hints is implementation−dependent; an implementation can ignore them entirely. (For more information about the relevant topics, see "Antialiasing" for the details on sampling

and "Fog" for details on fog.)
The GL_PERSPECTIVE_CORRECTION_HINT target parameter refers to how color values and texture coordinates are interpolated across a primitive: either linearly in screen space (a relatively simple calculation) or in a perspective−correct manner (which requires more computation). Often, systems perform linear color interpolation because the results, while not technically correct, are visually acceptable; however, in most cases textures require perspective−correct interpolation to be visually acceptable. Thus, an implementation can choose to use this parameter to control the method used for interpolation. (See Chapter 3 for a discussion of perspective projection, Chapter 4 for a discussion of color, and Chapter 9 for a discussion of texture mapping.)

Parameter	Meaning
GL_POINT_SMOOTH_HINT, GL_LINE_SMOOTH_HINT, GL_POLYGON_SMOOTH_HINT	Specify the desired sampling quality of points, lines, or polygons during antialiasing operations
GL_FOG_HINT	Specifies whether fog calculations are done per pixel (GL_NICEST) or per vertex (GL_FASTEST)
GL_PERSPECTIVE_CORRECTION_HINT	Specifies the desired quality of color and texture−coordinate interpolation

Table 6−2 Values for Use with glHint()

Antialiasing Points or Lines

To antialias points or lines, you need to turn on antialiasing with *glEnable()*, passing in GL_POINT_SMOOTH or GL_LINE_SMOOTH, as appropriate. You might also want to provide a quality hint with *glHint()*. (Remember that you can set the size of a point or the width of a line. You can also stipple a line. See "Line Details" in Chapter 2.) Next follow the procedures described in one of the following sections, depending on whether you're in RGBA or color−index mode.

Antialiasing in RGBA Mode

In RGBA mode, you need to enable blending. The blending factors you most likely want to use are GL_SRC_ALPHA (source) and GL_ONE_MINUS_SRC_ALPHA (destination). Alternatively, you can use GL_ONE for the destination factor to make lines a little brighter where they intersect. Now you're ready to draw whatever points or lines you want antialiased. The antialiased effect is most noticeable if you use a fairly high alpha value. Remember that since you're performing blending, you might need to consider the rendering order as described in "Three−Dimensional Blending with the Depth Buffer." However, in most cases, the ordering can be ignored without significant adverse effects. Example 6−3initializes the necessary modes for antialiasing and then draws two intersecting diagonal lines. When you run this program, press the 'r' key to rotate the lines so that you can see the effect of antialiasing on lines of different slopes. Note that the depth buffer isn't enabled in this example.

Example 6−3 Antialiased lines: aargb.c

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

```
#include <stdio.h>

static float rotAngle = 0.;

/*  Initialize antialiasing for RGBA mode, including alpha
 *  blending, hint, and line width.  Print out implementation
 *  specific info on line width granularity and width.
 */
void init(void)
{
   GLfloat values[2];
   glGetFloatv (GL_LINE_WIDTH_GRANULARITY, values);
   printf ("GL_LINE_WIDTH_GRANULARITY value is %3.1f\n",
      values[0]);
   glGetFloatv (GL_LINE_WIDTH_RANGE, values);
   printf ("GL_LINE_WIDTH_RANGE values are %3.1f %3.1f\n",
      values[0], values[1]);

   glEnable (GL_LINE_SMOOTH);
   glEnable (GL_BLEND);
   glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
   glHint (GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
   glLineWidth (1.5);

   glClearColor(0.0, 0.0, 0.0, 0.0);
}

/* Draw 2 diagonal lines to form an X */
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);

   glColor3f (0.0, 1.0, 0.0);
   glPushMatrix();
   glRotatef(-rotAngle, 0.0, 0.0, 0.1);
   glBegin (GL_LINES);
      glVertex2f (-0.5, 0.5);
      glVertex2f (0.5, -0.5);
   glEnd ();
   glPopMatrix();

   glColor3f (0.0, 0.0, 1.0);
   glPushMatrix();
   glRotatef(rotAngle, 0.0, 0.0, 0.1);
   glBegin (GL_LINES);
      glVertex2f (0.5, 0.5);
      glVertex2f (-0.5, -0.5);
```

```
   glEnd ();
   glPopMatrix();

   glFlush();
}
void reshape(int w, int h)
{
   glViewport(0, 0, (GLint) w, (GLint) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      gluOrtho2D (-1.0, 1.0,
         -1.0*(GLfloat)h/(GLfloat)w, 1.0*(GLfloat)h/(GLfloat)w);
   else
      gluOrtho2D (-1.0*(GLfloat)w/(GLfloat)h,
         1.0*(GLfloat)w/(GLfloat)h, -1.0, 1.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 'r':
      case 'R':
         rotAngle += 20.;
         if (rotAngle >= 360.) rotAngle = 0.;
         glutPostRedisplay();
         break;
      case 27:  /*  Escape Key  */
         exit(0);
         break;
      default:
         break;
    }
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize (200, 200);
   glutCreateWindow (argv[0]);
   init();
   glutReshapeFunc (reshape);
   glutKeyboardFunc (keyboard);
```

```
   glutDisplayFunc (display);
   glutMainLoop();
   return 0;
}
```

Antialiasing in Color−Index Mode

The tricky part about antialiasing in color−index mode is loading and using the color map. Since the last 4 bits of the color index indicate the coverage value, you need to load sixteen contiguous indices with a color ramp from the background color to the object's color. (The ramp has to start with an index value that's a multiple of 16.) Then you clear the color buffer to the first of the sixteen colors in the ramp and draw your points or lines using colors in the ramp. Example 6−4demonstrates how to construct the color ramp to draw antialiased lines in color−index mode. In this example, two color ramps are created: one contains shades of green and the other shades of blue.

Example 6−4 Antialiasing in Color−Index Mode: aaindex.c

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

#define RAMPSIZE 16
#define RAMP1START 32
#define RAMP2START 48

static float rotAngle = 0.;

/*  Initialize antialiasing for color-index mode,
 *  including loading a green color ramp starting
 *  at RAMP1START, and a blue color ramp starting
 *  at RAMP2START. The ramps must be a multiple of 16.
 */
void init(void)
{
   int i;

   for (i = 0; i < RAMPSIZE; i++) {
      GLfloat shade;
      shade = (GLfloat) i/(GLfloat) RAMPSIZE;
      glutSetColor(RAMP1START+(GLint)i, 0., shade, 0.);
      glutSetColor(RAMP2START+(GLint)i, 0., 0., shade);
   }
   glEnable (GL_LINE_SMOOTH);
   glHint (GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
   glLineWidth (1.5);

   glClearIndex ((GLfloat) RAMP1START);
```

```
}
/*  Draw 2 diagonal lines to form an X */
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);

   glIndexi(RAMP1START);
   glPushMatrix();
   glRotatef(-rotAngle, 0.0, 0.0, 0.1);
   glBegin (GL_LINES);
      glVertex2f (-0.5, 0.5);
      glVertex2f (0.5, -0.5);
   glEnd ();
   glPopMatrix();

   glIndexi(RAMP2START);
   glPushMatrix();
   glRotatef(rotAngle, 0.0, 0.0, 0.1);
   glBegin (GL_LINES);
      glVertex2f (0.5, 0.5);
      glVertex2f (-0.5, -0.5);
   glEnd ();
   glPopMatrix();

   glFlush();
}


void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      gluOrtho2D (-1.0, 1.0,
         -1.0*(GLfloat)h/(GLfloat)w, 1.0*(GLfloat)h/(GLfloat)w);
   else
      gluOrtho2D (-1.0*(GLfloat)w/(GLfloat)h,
         1.0*(GLfloat)w/(GLfloat)h, -1.0, 1.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 'r':
      case 'R':
```

```
            rotAngle += 20.;
            if (rotAngle >= 360.) rotAngle = 0.;
            glutPostRedisplay();
            break;
         case 27:  /*  Escape Key */
            exit(0);
            break;
         default:
            break;
      }
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_INDEX);
   glutInitWindowSize (200, 200);
   glutCreateWindow (argv[0]);
   init();
   glutReshapeFunc (reshape);
   glutKeyboardFunc (keyboard);
   glutDisplayFunc (display);
   glutMainLoop();
   return 0;
}
```

Since the color ramp goes from the background color to the object's color, the antialiased lines look correct only in the areas where they are drawn on top of the background. When the blue line is drawn, it erases part of the green line at the point where the lines intersect. To fix this, you would need to redraw the area where the lines intersect using a ramp that goes from green (the color of the line in the framebuffer) to blue (the color of the line being drawn). However, this requires additional calculations and it is usually not worth the effort since the intersection area is small. Note that this is not a problem in RGBA mode, since the colors of object being drawn are blended with the color already in the framebuffer.

You may also want to enable the depth test when drawing antialiased points and lines in color−index mode. In this example, the depth test is disabled since both of the lines lie in the same *z*−plane. However, if you want to draw a three−dimensional scene, you should enable the depth buffer so that the resulting pixel colors correspond to the "nearest" objects.

The trick described in "Three−Dimensional Blending with the Depth Buffer"can also be used to mix antialiased points and lines with aliased, depth−buffered polygons. To do this, draw the polygons first, then make the depth buffer read−only and draw the points and lines. The points and lines intersect nicely with each other but will be obscured by nearer polygons.

Try This

Take a previous program, such as the robot arm or solar system examples described in "Examples of Composing Several Transformations" in Chapter 3, and draw wireframe objects with antialiasing. Try it in either RGBA or color−index mode. Also try different line widths or point sizes to see their effects.

Antialiasing Polygons

Antialiasing the edges of filled polygons is similar to antialiasing points and lines. When different polygons have overlapping edges, you need to blend the color values appropriately. You can either use the method described in this section, or you can use the accumulation buffer to perform antialiasing for your entire scene. Using the accumulation buffer, which is described in Chapter 10, is easier from your point of view, but it's much more computation−intensive and therefore slower. However, as you'll see, the method described here is rather cumbersome.

Note: If you draw your polygons as points at the vertices or as outlines—that is, by passing GL_POINT or GL_LINE to *glPolygonMode()*—point or line antialiasing is applied, if enabled as described earlier. The rest of this section addresses polygon antialiasing when you're using GL_FILL as the polygon mode.

In theory, you can antialias polygons in either RGBA or color−index mode. However, object intersections affect polygon antialiasing more than they affect point or line antialiasing, so rendering order and blending accuracy become more critical. In fact, they're so critical that if you're antialiasing more than one polygon, you need to order the polygons from front to back and then use *glBlendFunc()* with GL_SRC_ALPHA_SATURATE for the source factor and GL_ONE for the destination factor. Thus, antialiasing polygons in color−index mode normally isn't practical.

To antialias polygons in RGBA mode, you use the alpha value to represent coverage values of polygon edges. You need to enable polygon antialiasing by passing GL_POLYGON_SMOOTH to *glEnable()*. This causes pixels on the edges of the polygon to be assigned fractional alpha values based on their coverage, as though they were lines being antialiased. Also, if you desire, you can supply a value for GL_POLYGON_SMOOTH_HINT.

Now you need to blend overlapping edges appropriately. First, turn off the depth buffer so that you have control over how overlapping pixels are drawn. Then set the blending factors to GL_SRC_ALPHA_SATURATE (source) and GL_ONE (destination). With this specialized blending function, the final color is the sum of the destination color and the scaled source color; the scale factor is the smaller of either the incoming source alpha value or one minus the destination alpha value. This means that for a pixel with a large alpha value, successive incoming pixels have little effect on the final color because one minus the destination alpha is almost zero. With this method, a pixel on the edge of a polygon might be blended eventually with the colors from another polygon that's drawn later. Finally, you need to sort all the polygons in your scene so that they're ordered from front to back before drawing them.

Example 6−5shows how to antialias filled polygons; clicking the left mouse button toggles the antialiasing on and off. Note that backward−facing polygons are culled and that the alpha values in the color buffer are cleared to zero before any drawing. Pressing the 't' key toggles the antialiasing on and off.

Note: Your color buffer must store alpha values for this technique to work correctly. Make sure you request GLUT_ALPHA and receive a legitimate window.

Example 6−5 Antialiasing Filled Polygons: aapoly.c

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

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

GLboolean polySmooth = GL_TRUE;
static void init(void)
{
   glCullFace (GL_BACK);
   glEnable (GL_CULL_FACE);
   glBlendFunc (GL_SRC_ALPHA_SATURATE, GL_ONE);
   glClearColor (0.0, 0.0, 0.0, 0.0);
}

#define NFACE 6
#define NVERT 8
void drawCube(GLdouble x0, GLdouble x1, GLdouble y0,
              GLdouble y1, GLdouble z0, GLdouble z1)
{
   static GLfloat v[8][3];
   static GLfloat c[8][4] = {
      {0.0, 0.0, 0.0, 1.0}, {1.0, 0.0, 0.0, 1.0},
      {0.0, 1.0, 0.0, 1.0}, {1.0, 1.0, 0.0, 1.0},
      {0.0, 0.0, 1.0, 1.0}, {1.0, 0.0, 1.0, 1.0},
      {0.0, 1.0, 1.0, 1.0}, {1.0, 1.0, 1.0, 1.0}
   };

/*  indices of front, top, left, bottom, right, back faces  */
   static GLubyte indices[NFACE][4] = {
      {4, 5, 6, 7}, {2, 3, 7, 6}, {0, 4, 7, 3},
      {0, 1, 5, 4}, {1, 5, 6, 2}, {0, 3, 2, 1}
   };

   v[0][0] = v[3][0] = v[4][0] = v[7][0] = x0;
   v[1][0] = v[2][0] = v[5][0] = v[6][0] = x1;
   v[0][1] = v[1][1] = v[4][1] = v[5][1] = y0;
   v[2][1] = v[3][1] = v[6][1] = v[7][1] = y1;
   v[0][2] = v[1][2] = v[2][2] = v[3][2] = z0;
   v[4][2] = v[5][2] = v[6][2] = v[7][2] = z1;

#ifdef GL_VERSION_1_1
   glEnableClientState (GL_VERTEX_ARRAY);
   glEnableClientState (GL_COLOR_ARRAY);
   glVertexPointer (3, GL_FLOAT, 0, v);
   glColorPointer (4, GL_FLOAT, 0, c);
```

```
   glDrawElements(GL_QUADS, NFACE*4, GL_UNSIGNED_BYTE, indices);
   glDisableClientState (GL_VERTEX_ARRAY);
   glDisableClientState (GL_COLOR_ARRAY);
#else
   printf ("If this is GL Version 1.0, ");
   printf ("vertex arrays are not supported.\n");
   exit(1);
#endif
}
/*  Note:  polygons must be drawn from front to back
 *  for proper blending.
 */
void display(void)
{
   if (polySmooth) {
      glClear (GL_COLOR_BUFFER_BIT);
      glEnable (GL_BLEND);
      glEnable (GL_POLYGON_SMOOTH);
      glDisable (GL_DEPTH_TEST);
   }
   else {
      glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glDisable (GL_BLEND);
      glDisable (GL_POLYGON_SMOOTH);
      glEnable (GL_DEPTH_TEST);
   }

   glPushMatrix ();
      glTranslatef (0.0, 0.0, -8.0);
      glRotatef (30.0, 1.0, 0.0, 0.0);
      glRotatef (60.0, 0.0, 1.0, 0.0);
      drawCube(-0.5, 0.5, -0.5, 0.5, -0.5, 0.5);
   glPopMatrix ();

   glFlush ();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(30.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
void keyboard(unsigned char key, int x, int y)
```

```
{
   switch (key) {
      case 't':
      case 'T':
         polySmooth = !polySmooth;
         glutPostRedisplay();
         break;
      case 27:
         exit(0);  /*  Escape key  */
         break;
      default:
         break;
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB
                        | GLUT_ALPHA | GLUT_DEPTH);
   glutInitWindowSize(200, 200);
   glutCreateWindow(argv[0]);
   init ();
   glutReshapeFunc (reshape);
   glutKeyboardFunc (keyboard);
   glutDisplayFunc (display);
   glutMainLoop();
   return 0;
}
```

Fog

Computer images sometimes seem unrealistically sharp and well defined. Antialiasing makes an object appear more realistic by smoothing its edges. Additionally, you can make an entire image appear more natural by adding fog, which makes objects fade into the distance. *Fog* is a general term that describes similar forms of atmospheric effects; it can be used to simulate haze, mist, smoke, or pollution. (See Plate 9.) Fog is essential in visual−simulation applications, where limited visibility needs to be approximated. It's often incorporated into flight−simulator displays.

When fog is enabled, objects that are farther from the viewpoint begin to fade into the fog color. You can control the density of the fog, which determines the rate at which objects fade as the distance increases, as well as the fog's color. Fog is available in both RGBA and color−index modes, although the calculations are slightly different in the two modes. Since fog is applied after matrix transformations, lighting, and texturing are performed, it affects transformed, lit, and textured objects. Note that with large simulation programs, fog can improve performance, since you can choose not to draw objects that would be too fogged to be visible.

All types of geometric primitives can be fogged, including points and lines. Using the fog effect on points and lines is also called **depth−cuing** (as shown in Plate 2) and is popular in molecular modeling and other applications.

Using Fog

Using fog is easy. You enable it by passing GL_FOG to *glEnable()*, and you choose the color and the equation that controls the density with *glFog*()*. If you want, you can supply a value for GL_FOG_HINT with *glHint()*, as described on Table 6−2 Example 6−6draws five red spheres, each at a different distance from the viewpoint. Pressing the 'f' key selects among the three different fog equations, which are described in the next section.

Example 6−6 Five Fogged Spheres in RGBA Mode: fog.c

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

static GLint fogMode;

static void init(void)
{
   GLfloat position[] = { 0.5, 0.5, 3.0, 0.0 };

   glEnable(GL_DEPTH_TEST);

   glLightfv(GL_LIGHT0, GL_POSITION, position);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   {
      GLfloat mat[3] = {0.1745, 0.01175, 0.01175};
      glMaterialfv (GL_FRONT, GL_AMBIENT, mat);
      mat[0] = 0.61424; mat[1] = 0.04136; mat[2] = 0.04136;
      glMaterialfv (GL_FRONT, GL_DIFFUSE, mat);
      mat[0] = 0.727811; mat[1] = 0.626959; mat[2] = 0.626959;
      glMaterialfv (GL_FRONT, GL_SPECULAR, mat);
      glMaterialf (GL_FRONT, GL_SHININESS, 0.6*128.0);
   }

   glEnable(GL_FOG);
   {
      GLfloat fogColor[4] = {0.5, 0.5, 0.5, 1.0};

      fogMode = GL_EXP;
      glFogi (GL_FOG_MODE, fogMode);
      glFogfv (GL_FOG_COLOR, fogColor);
```

```
      glFogf (GL_FOG_DENSITY, 0.35);
      glHint (GL_FOG_HINT, GL_DONT_CARE);
      glFogf (GL_FOG_START, 1.0);
      glFogf (GL_FOG_END, 5.0);
   }
   glClearColor(0.5, 0.5, 0.5, 1.0);  /* fog color */
}

static void renderSphere (GLfloat x, GLfloat y, GLfloat z)
{
   glPushMatrix();
   glTranslatef (x, y, z);
   glutSolidSphere(0.4, 16, 16);
   glPopMatrix();
}

/* display() draws 5 spheres at different z positions.
 */
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   renderSphere (−2., −0.5, −1.0);
   renderSphere (−1., −0.5, −2.0);
   renderSphere (0., −0.5, −3.0);
   renderSphere (1., −0.5, −4.0);
   renderSphere (2., −0.5, −5.0);
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho (−2.5, 2.5, −2.5*(GLfloat)h/(GLfloat)w,
         2.5*(GLfloat)h/(GLfloat)w, −10.0, 10.0);
   else
      glOrtho (−2.5*(GLfloat)w/(GLfloat)h,
         2.5*(GLfloat)w/(GLfloat)h, −2.5, 2.5, −10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity ();
}

void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
```

```
      case 'f':
      case 'F':
         if (fogMode == GL_EXP) {
            fogMode = GL_EXP2;
            printf ("Fog mode is GL_EXP2\n");
         }
         else if (fogMode == GL_EXP2) {
            fogMode = GL_LINEAR;
            printf ("Fog mode is GL_LINEAR\n");
         }
         else if (fogMode == GL_LINEAR) {
            fogMode = GL_EXP;
            printf ("Fog mode is GL_EXP\n");
         }
         glFogi (GL_FOG_MODE, fogMode);
         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(500, 500);
   glutCreateWindow(argv[0]);
   init();
   glutReshapeFunc (reshape);
   glutKeyboardFunc (keyboard);
   glutDisplayFunc (display);
   glutMainLoop();
   return 0;
}
```

Fog Equations

Fog blends a fog color with an incoming fragment's color using a fog blending factor. This factor, *f*, is computed with one of these three equations and then clamped to the range [0,1].

$$f = e^{-(density \cdot z)} \quad \text{(GL\_EXP)}$$

$$f = e^{-(density \cdot z)^2} \quad \text{(GL\_EXP2)}$$

$$f = \frac{end - z}{end - start} \quad \text{(GL\_LINEAR)}$$

In these three equations, z is the eye−coordinate distance between the viewpoint and the fragment center. The values for *density, start*, and *end* are all specified with *glFog*()*. The *f* factor is used differently, depending on whether you're in RGBA mode or color−index mode, as explained in the next subsections.

void glFog{if}(GLenum pname, TYPE param);
*void glFog{if}v(GLenum pname, TYPE *params);*

> *Sets the parameters and function for calculating fog. If pname is GL_FOG_MODE, then param is either GL_EXP (the default), GL_EXP2, or GL_LINEAR to select one of the three fog factors. If pname is GL_FOG_DENSITY, GL_FOG_START, or GL_FOG_END, then param is (or points to, with the vector version of the command) a value for density, start, or end in the equations. (The default values are 1, 0, and 1, respectively.) In RGBA mode, pname can be GL_FOG_COLOR, in which case params points to four values that specify the fog's RGBA color values. The corresponding value for pname in color−index mode is GL_FOG_INDEX, for which param is a single value specifying the fog's color index.*

Figure 6−4plots the fog−density equations for various values of the parameters.

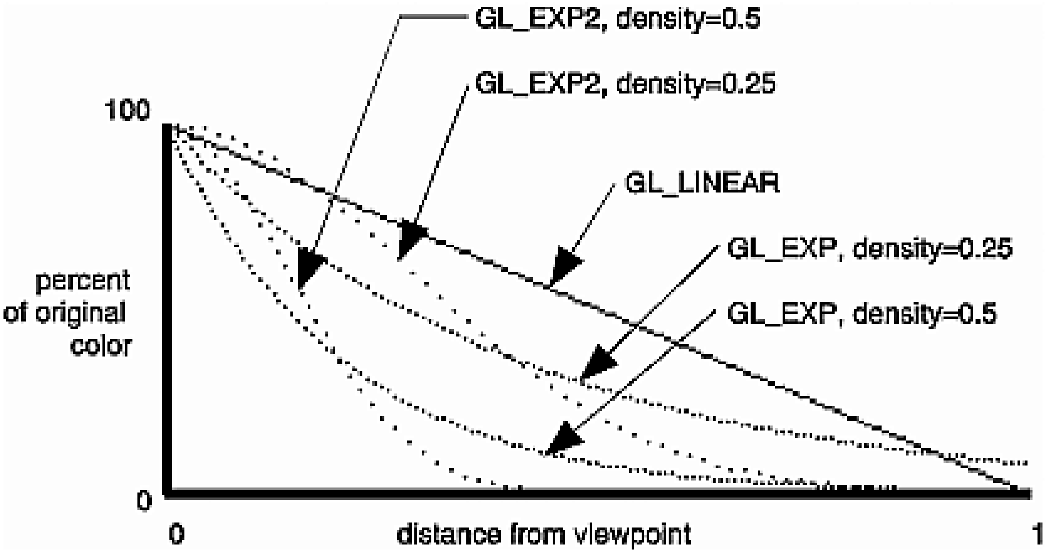

Figure 6−4 Fog−Density Equations

Fog in RGBA Mode

In RGBA mode, the fog factor *f* is used as follows to calculate the final fogged color:

$C = f\,C_i + (1 - f)\,C_f$

where C_i represents the incoming fragment's RGBA values and C_f the fog−color values assigned with GL_FOG_COLOR.

Fog in Color−Index Mode

In color−index mode, the final fogged color index is computed as follows:

$I = I_i + (1 - f)\, I_f$

where I_i is the incoming fragment's color index and I_f is the fog's color index as specified with GL_FOG_INDEX.

To use fog in color−index mode, you have to load appropriate values in a color ramp. The first color in the ramp is the color of the object without fog, and the last color in the ramp is the color of the completely fogged object. You probably want to use *glClearIndex()* to initialize the background color index so that it corresponds to the last color in the ramp; this way, totally fogged objects blend into the background. Similarly, before objects are drawn, you should call *glIndex*()* and pass in the index of the first color in the ramp (the unfogged color). Finally, to apply fog to different colored objects in the scene, you need to create several color ramps and call *glIndex*()* before each object is drawn to set the current color index to the start of each color ramp. Example 6−7illustrates how to initialize appropriate conditions and then apply fog in color−index mode.

Example 6−7 Fog in Color−Index Mode: fogindex.c

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

/*  Initialize color map and fog.  Set screen clear color
 *  to end of color ramp.
 */
#define NUMCOLORS 32
#define RAMPSTART 16

static void init(void)
{
   int i;

   glEnable(GL_DEPTH_TEST);

   for (i = 0; i < NUMCOLORS; i++) {
      GLfloat shade;
      shade = (GLfloat) (NUMCOLORS−i)/(GLfloat) NUMCOLORS;
      glutSetColor (RAMPSTART + i, shade, shade, shade);
   }
   glEnable(GL_FOG);

   glFogi (GL_FOG_MODE, GL_LINEAR);
   glFogi (GL_FOG_INDEX, NUMCOLORS);
```

```
   glFogf (GL_FOG_START, 1.0);
   glFogf (GL_FOG_END, 6.0);
   glHint (GL_FOG_HINT, GL_NICEST);
   glClearIndex((GLfloat) (NUMCOLORS+RAMPSTART−1));
}

static void renderSphere (GLfloat x, GLfloat y, GLfloat z)
{
   glPushMatrix();
   glTranslatef (x, y, z);
   glutWireSphere(0.4, 16, 16);
   glPopMatrix();
}

/*  display() draws 5 spheres at different z positions.
 */
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glIndexi (RAMPSTART);

   renderSphere (−2., −0.5, −1.0);
   renderSphere (−1., −0.5, −2.0);
   renderSphere (0., −0.5, −3.0);
   renderSphere (1., −0.5, −4.0);
   renderSphere (2., −0.5, −5.0);

   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho (−2.5, 2.5, −2.5*(GLfloat)h/(GLfloat)w,
         2.5*(GLfloat)h/(GLfloat)w, −10.0, 10.0);
   else
      glOrtho (−2.5*(GLfloat)w/(GLfloat)h,
         2.5*(GLfloat)w/(GLfloat)h, −2.5, 2.5, −10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity ();
}

void keyboard(unsigned char key, int x, int y)
{
```

```
    switch (key) {
        case 27:
            exit(0);
    }
}

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

Polygon Offset

If you want to highlight the edges of a solid object, you might try to draw the object with polygon mode GL_FILL and then draw it again, but in a different color with polygon mode GL_LINE. However, because lines and filled polygons are not rasterized in exactly the same way, the depth values generated for pixels on a line are usually not the same as the depth values for a polygon edge, even between the same two vertices. The highlighting lines may fade in and out of the coincident polygons, which is sometimes called "stitching" and is visually unpleasant.

The visual unpleasantness can be eliminated by using polygon offset, which adds an appropriate offset to force coincident *z* values apart to cleanly separate a polygon edge from its highlighting line. (The stencil buffer, described in "Stencil Test" in Chapter 10, can also be used to eliminate stitching. However, polygon offset is almost always faster than stenciling.) Polygon offset is also useful for applying decals to surfaces, rendering images with hidden−line removal. In addition to lines and filled polygons, this technique can also be used with points.

There are three different ways to turn on polygon offset, one for each type of polygon rasterization mode: GL_FILL, GL_LINE, or GL_POINT. You enable the polygon offset by passing the appropriate parameter to *glEnable()*, either GL_POLYGON_OFFSET_FILL, GL_POLYGON_OFFSET_LINE, or GL_POLYGON_OFFSET_POINT. You must also call *glPolygonMode()* to set the current polygon rasterization method.

void glPolygonOffset(GLfloat factor, GLfloat units);

> *When enabled, the depth value of each fragment is added to a calculated offset value. The offset is added before the depth test is performed and before the depth value is written into the depth buffer. The offset value o is calculated by:*
> *o = m * factor + r * units*

where m is the maximum depth slope of the polygon and r is the smallest value guaranteed to produce a resolvable difference in window coordinate depth values. The value r is an implementation−specific constant.

To achieve a nice rendering of the highlighted solid object without visual artifacts, you can either add a positive offset to the solid object (push it away from you) or a negative offset to the wireframe (pull it towards you). The big question is: "How much offset is enough?" Unfortunately, the offset required depends upon various factors, including the depth slope of each polygon and the width of the lines in the wireframe.

OpenGL calculates the depth slope (see Figure 6−5) of a polygon for you, but it's important that you understand what the depth slope is, so that you choose a reasonable value for *factor*. The depth slope is the change in *z* (depth) values divided by the change in either *x* or *y* coordinates, as you traverse a polygon. The depth values are in window coordinates, clamped to the range [0, 1]. To estimate the maximum depth slope of a polygon (*m* in the offset equation), use this formula:

$$m = max\left\{\left|\frac{\partial V}{\partial s}\right|, \left|\frac{\partial V}{\partial t}\right|\right\}$$

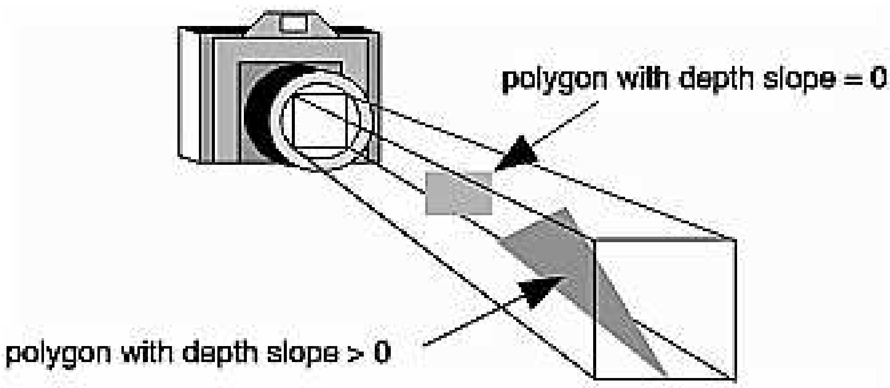

Figure 6−5 Polygons and Their Depth Slopes

For polygons that are parallel to the near and far clipping planes, the depth slope is zero. For the polygons in your scene with a depth slope near zero, only a small, constant offset is needed. To create a small, constant offset, you can pass *factor*=0.0 and *units*=1.0 to *glPolygonOffset()*.

For polygons that are at a great angle to the clipping planes, the depth slope can be significantly greater than zero, and a larger offset may be needed. Small, non−zero values for *factor,* such as 0.75 or 1.0, are probably enough to generate distinct depth values and eliminate the unpleasant visual artifacts.

Example 6−8 shows a portion of code, where a display list (which presumably draws a solid object) is first rendered with lighting, the default GL_FILL polygon mode, and polygon offset with *factor* of 1.0 and *units* of 1.0. These values ensure that the offset is enough for all polygons in your scene, regardless of depth slope. (These values may actually be a little more offset than the minimum needed, but too much offset is less noticeable than too little.) Then, to highlight the edges of the first object, the object is rendered as an unlit wireframe with the offset disabled.

Example 6−8 Polygon Offset to Eliminate Visual Artifacts: polyoff.c

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(1.0, 1.0);
glCallList (list);
glDisable(GL_POLYGON_OFFSET_FILL);

glDisable(GL_LIGHTING);
glDisable(GL_LIGHT0);
glColor3f (1.0, 1.0, 1.0);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glCallList (list);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

In a few situations, the simplest values for *factor* and *units* (1.0 and 1.0) aren't the answers. For instance, if the width of the lines that are highlighting the edges are greater than one, then increasing the value of *factor* may be necessary. Also, since depth values are unevenly transformed into window coordinates when using perspective projection (see "The Transformed Depth Coordinate" in Chapter 3), less offset is needed for polygons that are closer to the near clipping plane, and more offset is needed for polygons that are further away. Once again, experimenting with the value of *factor* may be warranted.