Chapter 10

The Framebuffer

Chapter 10

The Framebuffer

Chapter Objectives

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

Understand what buffers make up the framebuffer and how they're used

Clear selected buffers and enable them for writing

Control the parameters of the scissoring, alpha, stencil, and depth−buffer tests that are applied to pixels

Perform dithering and logical operations

Use the accumulation buffer for such purposes as scene antialiasing

An important goal of almost every graphics program is to draw pictures on the screen. The screen is composed of a rectangular array of pixels, each capable of displaying a tiny square of color at that point in the image. After the rasterization stage (including texturing and fog), the data are not yet pixels, but are fragments. Each fragment has coordinate data which corresponds to a pixel, as well as color and depth values. Then each fragment undergoes a series of tests and operations, some of which have been previously described (See "Blending" in Chapter 6) and others that are discussed in this chapter.

If the tests and operations are survived, the fragment values are ready to become pixels. To draw these pixels, you need to know what color they are, which is the information that's stored in the color buffer. Whenever data is stored uniformly for each pixel, such storage for all the pixels is called a *buffer*. Different buffers might contain different amounts of data per pixel, but within a given buffer, each pixel is assigned the same amount of data. A buffer that stores a single bit of information about pixels is called a **bitplane**.

As shown in Figure 10−1, the lower−left pixel in an OpenGL window is pixel (0, 0), corresponding to the window coordinates of the lower−left corner of the 1×1 region occupied by this pixel. In general, pixel (*x, y*) fills the region bounded by *x* on the left, *x*+1 on the right, *y* on the bottom, and *y*+1 on the top.

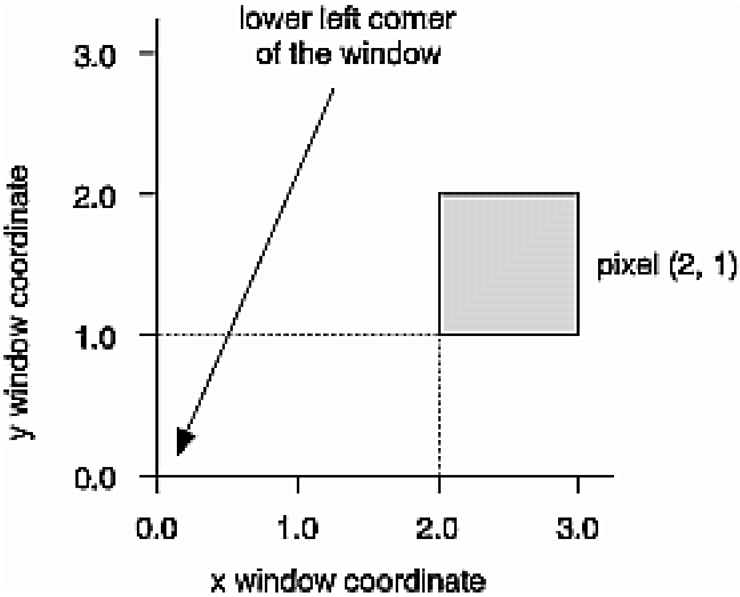

Figure 10−1 Region Occupied by a Pixel

As an example of a buffer, let's look more closely at the color buffer, which holds the color information that's to be displayed on the screen. Assume that the screen is 1280 pixels wide and 1024 pixels high and that it's a full 24−bit color screen—in other words, there are 2^{24} (or 16,777,216) different colors that can be displayed. Since 24 bits translates to 3 bytes (8 bits/byte), the color buffer in this example has to store at least 3 bytes of data for each of the 1,310,720 (1280*1024) pixels on the screen. A particular hardware system might have more or fewer pixels on the physical screen as well as more or less color data per pixel. Any particular color buffer, however, has the same amount of data saved for each pixel on the screen.

The color buffer is only one of several buffers that hold information about a pixel. For example, in "A Hidden−Surface Removal Survival Kit" in Chapter 5, you learned that the depth buffer holds depth information for each pixel. The color buffer itself can consist of several subbuffers. The **framebuffer** on a system comprises all of these buffers. With the exception of the color buffer(s), you don't view these other buffers directly; instead, you use them to perform such tasks as hidden−surface elimination, antialiasing of an entire scene, stenciling, drawing smooth motion, and other operations.

This chapter describes all the buffers that can exist in an OpenGL implementation and how they're used. It also discusses the series of tests and pixel operations that are performed before any data is written to the viewable color buffer. Finally, it explains how to use the accumulation buffer, which is used to accumulate images that are drawn into the color buffer. This chapter has the following major sections.

"Buffers and Their Uses" describes the possible buffers, what they're for, and how to clear them and enable them for writing.

"Testing and Operating on Fragments" explains the scissoring, alpha, stencil, and depth−buffer tests that occur after a pixel's position and color have been calculated but before this information is drawn on the screen. Several operations—blending, dithering, and logical operations—can also be performed before a fragment updates the screen.

"The Accumulation Buffer" describes how to perform several advanced techniques using the accumulation buffer. These techniques include antialiasing an entire scene, using motion blur, and simulating photographic depth of field.

Buffers and Their Uses

An OpenGL system can manipulate the following buffers:

Color buffers: front−left, front−right, back−left, back−right, and any number of auxiliary color buffers

Depth buffer

Stencil buffer

Accumulation buffer

Your particular OpenGL implementation determines which buffers are available and how many bits per pixel each holds. Additionally, you can have multiple visuals, or window types, that have different buffers available. Table 10−1lists the parameters to use with *glGetIntegerv()* to query your OpenGL system about per−pixel buffer storage for a particular visual.

Note: If you're using the X Window System, you're guaranteed, at a minimum, to have a visual with one color buffer for use in RGBA mode with associated stencil, depth, and accumulation buffers that have color components of nonzero size. Also, if your X Window System implementation supports a Pseudo−Color visual, you are also guaranteed to have one OpenGL visual that has a color buffer for use in color−index mode with associated depth and stencil buffers. You'll probably want to use *glXGetConfig()* to query your visuals; see Appendix C and the *OpenGL Reference Manual* for more information about this routine.

Parameter	Meaning
GL_RED_BITS, GL_GREEN_BITS, GL_BLUE_BITS, GL_ALPHA_BITS	Number of bits per R, G, B, or A component in the color buffers
GL_INDEX_BITS	Number of bits per index in the color buffers
GL_DEPTH_BITS	Number of bits per pixel in the depth buffer
GL_STENCIL_BITS	Number of bits per pixel in the stencil buffer
GL_ACCUM_RED_BITS, GL_ACCUM_GREEN_BITS, GL_ACCUM_BLUE_BITS, GL_ACCUM_ALPHA_BITS	Number of bits per R, G, B, or A component in the accumulation buffer

Table 10−1 Query Parameters for Per−Pixel Buffer Storage

Color Buffers

The color buffers are the ones to which you usually draw. They contain either color−index or RGB color data and may also contain alpha values. An OpenGL implementation that supports stereoscopic viewing has left and right color buffers for the left and right stereo images. If stereo isn't supported, only the left buffers are used. Similarly, double−buffered systems have front and back buffers, and a single−buffered system has the front buffers only. Every OpenGL implementation must provide a front−left color buffer.

Optional, nondisplayable auxiliary color buffers may also be supported. OpenGL doesn't specify any particular uses for these buffers, so you can define and use them however you please. For example, you might use them for saving an image that you use repeatedly. Then rather than redrawing the image, you can just copy it from an auxiliary buffer into the usual color buffers. (See the description of *glCopyPixels()* in "Reading, Writing, and Copying Pixel Data" in Chapter 8 for more information about how to do this.)

You can use GL_STEREO or GL_DOUBLEBUFFER with *glGetBooleanv()* to find out if your system supports stereo (that is, has left and right buffers) or double−buffering (has front and back buffers). To find out how many, if any, auxiliary buffers are present, use *glGetIntegerv()* with GL_AUX_BUFFERS.

Depth Buffer

The depth buffer stores a depth value for each pixel. As described in "A Hidden−Surface Removal Survival Kit" in Chapter 5, depth is usually measured in terms of distance to the eye, so pixels with larger depth−buffer values are overwritten by pixels with smaller values. This is just a useful convention, however, and the depth buffer's behavior can be modified as described in "Depth Test." The depth buffer is sometimes called the *z buffer* (the z comes from the fact that *x* and *y* values measure horizontal and vertical displacement on the screen, and the *z* value measures distance perpendicular to the screen).

Stencil Buffer

One use for the stencil buffer is to restrict drawing to certain portions of the screen, just as a cardboard stencil can be used with a can of spray paint to make fairly precise painted images. For example, if you want to draw an image as it would appear through an odd−shaped windshield, you can store an image of the windshield's shape in the stencil buffer, and then draw the entire scene. The stencil buffer prevents anything that wouldn't be visible through the windshield from being drawn. Thus, if your application is a driving simulation, you can draw all the instruments and other items inside the automobile once, and as the car moves, only the outside scene need be updated.

Accumulation Buffer

The accumulation buffer holds RGBA color data just like the color buffers do in RGBA mode. (The results of using the accumulation buffer in color−index mode are undefined.) It's typically used for accumulating a series of images into a final, composite image. With this method, you can perform operations like scene antialiasing by supersampling an image and then averaging the samples to produce the values that are finally painted into the pixels of the color buffers. You don't draw directly into the accumulation buffer; accumulation operations are always performed in rectangular blocks, which are usually transfers of data to or from a color buffer.

Clearing Buffers

In graphics programs, clearing the screen (or any of the buffers) is typically one of the most expensive operations you can perform—on a 1280×1024 monitor, it requires touching well over a million pixels. For simple graphics applications, the clear operation can take more time than the rest of the drawing. If you need to clear not only the color buffer but also the depth and stencil buffers, the clear operation can be three times as expensive.

To address this problem, some machines have hardware that can clear more than one buffer at once. The OpenGL clearing commands are structured to take advantage of architectures like this. First, you specify the values to be written into each buffer to be cleared. Then you issue a single command to perform the clear operation, passing in a list of all the buffers to be cleared. If the hardware is capable of simultaneous clears, they all occur at once; otherwise, each buffer is cleared sequentially.

The following commands set the clearing values for each buffer.

void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
void glClearIndex(GLfloat index);
void glClearDepth(GLclampd depth);
void glClearStencil(GLint s);
void glClearAccum(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);

> *Specifies the current clearing values for the color buffer (in RGBA mode), the color buffer (in color−index mode), the depth buffer, the stencil buffer, and the accumulation buffer. The GLclampf and GLclampd types (clamped GLfloat and clamped GLdouble) are clamped to be between 0.0 and 1.0. The default depth−clearing value is 1.0; all the other default clearing values are 0. The values set with the clear commands remain in effect until they're changed by another call to the same command.*

After you've selected your clearing values and you're ready to clear the buffers, use *glClear()*.

void glClear(GLbitfield mask);

> *Clears the specified buffers. The value of mask is the bitwise logical OR of some combination of GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, GL_STENCIL_BUFFER_BIT, and GL_ACCUM_BUFFER_BIT to identify which buffers are to be cleared. GL_COLOR_BUFFER_BIT clears either the RGBA color or the color−index buffer, depending on the mode of the system at the time. When you clear the color or color−index buffer, all the color buffers that are enabled for writing (see the next section) are cleared. The pixel ownership test, scissor test, and dithering, if enabled, are applied to the clearing operation. Masking operations, such as glColorMask() and glIndexMask(), are also effective. The alpha test, stencil test, and depth test do not affect the operation of glClear().*

Selecting Color Buffers for Writing and Reading

The results of a drawing or reading operation can go into or come from any of the color buffers: front, back, front−left, back−left, front−right, back−right, or any of the auxiliary buffers. You can choose an individual buffer to be the drawing or reading target. For drawing, you can also set the target to draw into more than one buffer at the same time. You use *glDrawBuffer()* to select the buffers to be written and *glReadBuffer()* to select the buffer as the source for *glReadPixels()*, *glCopyPixels()*, *glCopyTexImage*()*, and *glCopyTexSubImage*()*.

If you are using double−buffering, you usually want to draw only in the back buffer (and swap the buffers when you're finished drawing). In some situations, you might want to treat a double−buffered window as though it were single−buffered by calling *glDrawBuffer()* to enable you to draw to both front and back buffers at the same time.

glDrawBuffer() is also used to select buffers to render stereo images (GL*LEFT and GL*RIGHT) and to render into auxiliary buffers (GL_AUX*i*).

void glDrawBuffer(GLenum mode);

> *Selects the color buffers enabled for writing or clearing. Disables buffers enabled by previous calls to glDrawBuffer(). More than one buffer may be enabled at one time. The value of mode*

can be one of the following:

GL_FRONT	GL_FRONT_LEFT	GL_AUX*i*
GL_BACK	GL_FRONT_RIGHT	GL_FRONT_AND_BACK
GL_LEFT	GL_BACK_LEFT	GL_NONE
GL_RIGHT	GL_BACK_RIGHT	

Arguments that omit LEFT or RIGHT refer to both the left and right buffers; similarly, arguments that omit FRONT or BACK refer to both. The i in GL_AUXi is a digit identifying a particular auxiliary buffer.

By default, mode is GL_FRONT for single−buffered contexts and GL_BACK for double−buffered contexts.

Note: You can enable drawing to nonexistent buffers as long as you enable drawing to at least one buffer that does exist. If none of the specified buffers exist, an error results.

void glReadBuffer(GLenum mode);

Selects the color buffer enabled as the source for reading pixels for subsequent calls to glReadPixels(), glCopyPixels(), glCopyTexImage(), and glCopyTexSubImage*(). Disables buffers enabled by previous calls to glReadBuffer(). The value of mode can be one of the following:*

GL_FRONT	GL_FRONT_LEFT	GL_AUX*i*
GL_BACK	GL_FRONT_RIGHT	
GL_LEFT	GL_BACK_LEFT	
GL_RIGHT	GL_BACK_RIGHT	

By default, mode is GL_FRONT for single−buffered contexts and GL_BACK for double−buffered contexts.

Note: You must enable reading from a buffer that does exist or an error results.

Masking Buffers

Before OpenGL writes data into the enabled color, depth, or stencil buffers, a masking operation is applied to the data, as specified with one of the following commands. A bitwise logical AND is performed with each mask and the corresponding data to be written.

void glIndexMask(GLuint mask);
void glColorMask(GLboolean red, GLboolean green, GLboolean blue,
GLboolean alpha);
void glDepthMask(GLboolean flag);
void glStencilMask(GLuint mask);

Sets the masks used to control writing into the indicated buffers. The mask set by glIndexMask() applies only in color−index mode. If a 1 appears inmask, the corresponding bit in the color−index buffer is written; where a 0 appears, the bit isn't written. Similarly,glColorMask() affects drawing in RGBA mode only. The red, green, blue, and alpha values control whether the corresponding component is written. (GL_TRUE means it is written.) If flag is GL_TRUE for glDepthMask(), the depth buffer is enabled for writing; otherwise, it's disabled. The mask for glStencilMask() is used for stencil data in the same way as the mask is used for color−index data in glIndexMask(). The default values of all the GLboolean masks are GL_TRUE, and the default values for the two GLuint masks are all 1's.

You can do plenty of tricks with color masking in color−index mode. For example, you can use each bit in the index as a different layer and set up interactions between arbitrary layers with appropriate settings of the color map. You can create overlays and underlays, and do so−called color−map animations. (See Chapter 14 for examples of using color masking.) Masking in RGBA mode is useful less often, but you can use it for loading separate image files into the red, green, and blue bitplanes, for example.

You've seen one use for disabling the depth buffer in "Three−Dimensional Blending with the Depth Buffer" in Chapter 6. Disabling the depth buffer for writing can also be useful if a common background is desired for a series of frames, and you want to add some features that may be obscured by parts of the background. For example, suppose your background is a forest, and you would like to draw repeated frames with the same trees, but with objects moving among them. After the trees are drawn with their depths recorded in the depth buffer, then the image of the trees is saved, and the new items are drawn with the depth buffer disabled for writing. As long as the new items don't overlap each other, the picture is correct. To draw the next frame, restore the image of the trees and continue. You don't need to restore the values in the depth buffer. This trick is most useful if the background is extremely complex—so complex that it's much faster just to recopy the image into the color buffer than to recompute it from the geometry.

Masking the stencil buffer can allow you to use a multiple−bit stencil buffer to hold multiple stencils (one per bit). You might use this technique to perform capping as explained in "Stencil Test" or to implement the Game of Life as described in "Life in the Stencil Buffer" in Chapter 14.

Note: The mask specified by *glStencilMask()* controls which stencil bitplanes are written. This mask isn't related to the mask that's specified as the third parameter of *glStencilFunc()*, which specifies which bitplanes are considered by the stencil function.

Testing and Operating on Fragments

When you draw geometry, text, or images on the screen, OpenGL performs several calculations to rotate, translate, scale, determine the lighting, project the object(s) into perspective, figure out which pixels in the window are affected, and determine what colors those pixels should be drawn. Many of the earlier chapters in this book give some information about how to control these operations. After OpenGL determines that an individual fragment should be generated and what its color should be, several processing stages remain that control how and whether the fragment is drawn as a pixel into the framebuffer. For example, if it's outside a rectangular region or if it's farther from the viewpoint than the pixel that's already in the framebuffer, it isn't drawn. In another stage, the fragment's color is blended with the color of the pixel already in the framebuffer.

This section describes both the complete set of tests that a fragment must pass before it goes into the framebuffer and the possible final operations that can be performed on the fragment as it's written. The tests and operations occur in the following order; if a fragment is eliminated in an early test, none of the later tests or operations take place.

1. Scissor test
2. Alpha test
3. Stencil test
4. Depth test
5. Blending
6. Dithering

7. Logical operation

Each of these tests and operations is described in detail in the following sections.

Scissor Test

You can define a rectangular portion of your window and restrict drawing to take place within it by using the *glScissor()* command. If a fragment lies inside the rectangle, it passes the scissor test.

void glScissor(GLint x, GLint y, GLsizei width, GLsizei height);

Sets the location and size of the scissor rectangle (also known as the scissor box). The parameters define the lower−left corner (x, y), and the width and height of the rectangle. Pixels that lie inside the rectangle pass the scissor test. Scissoring is enabled and disabled by passing GL_SCISSOR_TEST to glEnable() and glDisable(). By default, the rectangle matches the size of the window and scissoring is disabled.

The scissor test is just a version of a stencil test using a rectangular region of the screen. It's fairly easy to create a blindingly fast hardware implementation of scissoring, while a given system might be much slower at stenciling—perhaps because the stenciling is performed in software.

Advanced

An advanced use of scissoring is performing nonlinear projection. First divide the window into a regular grid of subregions, specifying viewport and scissor parameters that limit rendering to one region at a time. Then project the entire scene to each region using a different projection matrix.

To determine whether scissoring is enabled and to obtain the values that define the scissor rectangle, you can use GL_SCISSOR_TEST with *glIsEnabled()* and GL_SCISSOR_BOX with *glGetIntegerv()*.

Alpha Test

In RGBA mode, the alpha test allows you to accept or reject a fragment based on its alpha value. The alpha test is enabled and disabled by passing GL_ALPHA_TEST to *glEnable()* and *glDisable()*. To determine whether the alpha test is enabled, use GL_ALPHA_TEST with *glIsEnabled()*.

If enabled, the test compares the incoming alpha value with a reference value. The fragment is accepted or rejected depending on the result of the comparison. Both the reference value and the comparison function are set with *glAlphaFunc()*. By default, the reference value is zero, the comparison function is GL_ALWAYS, and the alpha test is disabled. To obtain the alpha comparison function or reference value, use GL_ALPHA_TEST_FUNC or GL_ALPHA_TEST_REF with *glGetIntegerv()*.

void glAlphaFunc(GLenum func, GLclampf ref);

Sets the reference value and comparison function for the alpha test. The reference value ref is clamped to be between zero and one. The possible values for func and their meaning are listed in Table 10−2

Parameter	Meaning
GL_NEVER	Never accept the fragment
GL_ALWAYS	Always accept the fragment
GL_LESS	Accept fragment if fragment alpha < reference alpha
GL_LEQUAL	Accept fragment if fragment alpha ≤ reference alpha
GL_EQUAL	Accept fragment if fragment alpha = reference alpha

GL_GEQUAL	Accept fragment if fragment alpha ≥ reference alpha
GL_GREATER	Accept fragment if fragment alpha > reference alpha
GL_NOTEQUAL	Accept fragment if fragment alpha ≠ reference alpha

Table 10−2 glAlphaFunc() Parameter Values (continued)

One application for the alpha test is to implement a transparency algorithm. Render your entire scene twice, the first time accepting only fragments with alpha values of one, and the second time accepting fragments with alpha values that aren't equal to one. Turn the depth buffer on during both passes, but disable depth buffer writing during the second pass.

Another use might be to make decals with texture maps where you can see through certain parts of the decals. Set the alphas in the decals to 0.0 where you want to see through, set them to 1.0 otherwise, set the reference value to 0.5 (or anything between 0.0 and 1.0), and set the comparison function to GL_GREATER. The decal has see−through parts, and the values in the depth buffer aren't affected. This technique, called billboarding, is described in "Sample Uses of Blending" in Chapter 6.

Stencil Test

The stencil test takes place only if there is a stencil buffer. (If there is no stencil buffer, the stencil test always passes.) Stenciling applies a test that compares a reference value with the value stored at a pixel in the stencil buffer. Depending on the result of the test, the value in the stencil buffer is modified. You can choose the particular comparison function used, the reference value, and the modification performed with the *glStencilFunc()* and *glStencilOp()* commands.

void glStencilFunc(GLenum func, GLint ref, GLuint mask);

Sets the comparison function (func), reference value (ref), and a mask (mask) for use with the stencil test. The reference value is compared to the value in the stencil buffer using the comparison function, but the comparison applies only to those bits where the corresponding bits of the mask are 1. The function can be GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_EQUAL, GL_GEQUAL, GL_GREATER, or GL_NOTEQUAL. If it's GL_LESS, for example, then the fragment passes if ref is less than the value in the stencil buffer. If the stencil buffer contains s bitplanes, the low−orders bits of mask are bitwise ANDed with the value in the stencil buffer and with the reference value before the comparison is performed. The masked values are all interpreted as nonnegative values. The stencil test is enabled and disabled by passing GL_STENCIL_TEST to glEnable() and glDisable(). By default, func is GL_ALWAYS, ref is 0, mask is all 1's, and stenciling is disabled.

void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);

Specifies how the data in the stencil buffer is modified when a fragment passes or fails the stencil test. The three functions fail, zfail, and zpass can be GL_KEEP, GL_ZERO, GL_REPLACE, GL_INCR, GL_DECR, or GL_INVERT. They correspond to keeping the current value, replacing it with zero, replacing it with the reference value, incrementing it, decrementing it, and bitwise−inverting it. The result of the increment and decrement functions is clamped to lie between zero and the maximum unsigned integer value (2^s-1 if the stencil buffer holdss bits). The fail function is applied if the fragment fails the stencil test; if it passes, then zfail is applied if the depth test fails and zpass if the depth test passes, or if no depth test is performed. (See "Depth Test.") By default, all three stencil operations are GL_KEEP.

Stencil Queries

You can obtain the values for all six stencil−related parameters by using the query function *glGetIntegerv()* and one of the values shown in Table 10−3. You can also determine whether the stencil test is enabled by passing GL_STENCIL_TEST to *glIsEnabled()*.

Query Value	Meaning
GL_STENCIL_FUNC	Stencil function
GL_STENCIL_REF	Stencil reference value
GL_STENCIL_VALUE_MASK	Stencil mask
GL_STENCIL_FAIL	Stencil fail action
GL_STENCIL_PASS_DEPTH_FAIL	Stencil pass and depth buffer fail action
GL_STENCIL_PASS_DEPTH_PASS	Stencil pass and depth buffer pass action

Table 10−3 Query Values for the Stencil Test (continued)

Stencil Examples

Probably the most typical use of the stencil test is to mask out an irregularly shaped region of the screen to prevent drawing from occurring within it (as in the windshield example in "Buffers and Their Uses"). To do this, fill the stencil mask with zeros, and then draw the desired shape in the stencil buffer with 1's. You can't draw geometry directly into the stencil buffer, but you can achieve the same result by drawing into the color buffer and choosing a suitable value for the *zpass* function (such as GL_REPLACE). (You can use *glDrawPixels()* to draw pixel data directly into the stencil buffer.) Whenever drawing occurs, a value is also written into the stencil buffer (in this case, the reference value). To prevent the stencil−buffer drawing from affecting the contents of the color buffer, set the color mask to zero (or GL_FALSE). You might also want to disable writing into the depth buffer.

After you've defined the stencil area, set the reference value to one, and the comparison function such that the fragment passes if the reference value is equal to the stencil−plane value. During drawing, don't modify the contents of the stencil planes.

Example 10−1demonstrates how to use the stencil test in this way. Two tori are drawn, with a diamond−shaped cutout in the center of the scene. Within the diamond−shaped stencil mask, a sphere is drawn. In this example, drawing into the stencil buffer takes place only when the window is redrawn, so the color buffer is cleared after the stencil mask has been created.

Example 10−1 Using the Stencil Test: stencil.c

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

#define YELLOWMAT   1
#define BLUEMAT 2

void init (void)
{
   GLfloat yellow_diffuse[] = { 0.7, 0.7, 0.0, 1.0 };
   GLfloat yellow_specular[] = { 1.0, 1.0, 1.0, 1.0 };
```

```
   GLfloat blue_diffuse[] = { 0.1, 0.1, 0.7, 1.0 };
   GLfloat blue_specular[] = { 0.1, 1.0, 1.0, 1.0 };

   GLfloat position_one[] = { 1.0, 1.0, 1.0, 0.0 };

   glNewList(YELLOWMAT, GL_COMPILE);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, yellow_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, yellow_specular);
   glMaterialf(GL_FRONT, GL_SHININESS, 64.0);
   glEndList();

   glNewList(BLUEMAT, GL_COMPILE);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, blue_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, blue_specular);
   glMaterialf(GL_FRONT, GL_SHININESS, 45.0);
   glEndList();

   glLightfv(GL_LIGHT0, GL_POSITION, position_one);

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

   glClearStencil(0x0);
   glEnable(GL_STENCIL_TEST);
}

/* Draw a sphere in a diamond-shaped section in the
 * middle of a window with 2 tori.
 */
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

/* draw blue sphere where the stencil is 1 */
   glStencilFunc (GL_EQUAL, 0x1, 0x1);
   glStencilOp (GL_KEEP, GL_KEEP, GL_KEEP);
   glCallList (BLUEMAT);
   glutSolidSphere (0.5, 15, 15);

/* draw the tori where the stencil is not 1 */
   glStencilFunc (GL_NOTEQUAL, 0x1, 0x1);
   glPushMatrix();
      glRotatef (45.0, 0.0, 0.0, 1.0);
      glRotatef (45.0, 0.0, 1.0, 0.0);
      glCallList (YELLOWMAT);
      glutSolidTorus (0.275, 0.85, 15, 15);
```

```
      glPushMatrix();
         glRotatef (90.0, 1.0, 0.0, 0.0);
         glutSolidTorus (0.275, 0.85, 15, 15);
      glPopMatrix();
   glPopMatrix();
}

/*  Whenever the window is reshaped, redefine the
 *  coordinate system and redraw the stencil area.
 */
void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);

/* create a diamond shaped stencil area */
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      gluOrtho2D(-3.0, 3.0, -3.0*(GLfloat)h/(GLfloat)w,
                 3.0*(GLfloat)h/(GLfloat)w);
   else
      gluOrtho2D(-3.0*(GLfloat)w/(GLfloat)h,
                 3.0*(GLfloat)w/(GLfloat)h, -3.0, 3.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();

   glClear(GL_STENCIL_BUFFER_BIT);
   glStencilFunc (GL_ALWAYS, 0x1, 0x1);
   glStencilOp (GL_REPLACE, GL_REPLACE, GL_REPLACE);
   glBegin(GL_QUADS);
      glVertex2f (-1.0, 0.0);
      glVertex2f (0.0, 1.0);
      glVertex2f (1.0, 0.0);
      glVertex2f (0.0, -1.0);
   glEnd();

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(45.0, (GLfloat) w/(GLfloat) h, 3.0, 7.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef(0.0, 0.0, -5.0);
}

/* Main Loop
 * Be certain to request stencil bits.
```

```
 */
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB
                        | GLUT_DEPTH | GLUT_STENCIL);
   glutInitWindowSize (400, 400);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutReshapeFunc(reshape);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

The following examples illustrate other uses of the stencil test. (See Chapter 14 for additional ideas.)

Capping—Suppose you're drawing a closed convex object (or several of them, as long as they don't intersect or enclose each other) made up of several polygons, and you have a clipping plane that may or may not slice off a piece of it. Suppose that if the plane does intersect the object, you want to cap the object with some constant−colored surface, rather than seeing the inside of it. To do this, clear the stencil buffer to zeros, and begin drawing with stenciling enabled and the stencil comparison function set to always accept fragments. Invert the value in the stencil planes each time a fragment is accepted. After all the objects are drawn, regions of the screen where no capping is required have zeros in the stencil planes, and regions requiring capping are nonzero. Reset the stencil function so that it draws only where the stencil value is nonzero, and draw a large polygon of the capping color across the entire screen.

Overlapping translucent polygons—Suppose you have a translucent surface that's made up of polygons that overlap slightly. If you simply use alpha blending, portions of the underlying objects are covered by more than one transparent surface, which doesn't look right. Use the stencil planes to make sure that each fragment is covered by at most one portion of the transparent surface. Do this by clearing the stencil planes to zeros, drawing only when the stencil plane is zero, and incrementing the value in the stencil plane when you draw.

Stippling—Suppose you want to draw an image with a stipple pattern. (See "Displaying Points, Lines, and Polygons" in Chapter 2 for more information about stippling.) You can do this by writing the stipple pattern into the stencil buffer, and then drawing conditionally on the contents of the stencil buffer. After the original stipple pattern is drawn, the stencil buffer isn't altered while drawing the image, so the object gets stippled by the pattern in the stencil planes.

Depth Test

For each pixel on the screen, the depth buffer keeps track of the distance between the viewpoint and the object occupying that pixel. Then if the specified depth test passes, the incoming depth value replaces the one already in the depth buffer.

The depth buffer is generally used for hidden−surface elimination. If a new candidate color for that pixel appears, it's drawn only if the corresponding object is closer than the previous object. In this way, after the entire scene has been rendered, only objects that aren't obscured by other items remain. Initially, the clearing value for the depth buffer is a value that's as far from the viewpoint as possible, so the depth of any object is nearer than that value. If this is how you want to use the depth buffer, you simply have to enable it by passing GL_DEPTH_TEST to *glEnable()* and remember to clear the depth buffer before you redraw each frame. (See "Clearing Buffers.") You can also choose a different comparison function for the depth test with *glDepthFunc()*.

void glDepthFunc(GLenum func);

Sets the comparison function for the depth test. The value for func must be GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_EQUAL, GL_GEQUAL, GL_GREATER, or GL_NOTEQUAL. An incoming fragment passes the depth test if its z value has the specified relation to the value already stored in the depth buffer. The default is GL_LESS, which means that an incoming fragment passes the test if its z value is less than that already stored in the depth buffer. In this case, the z value represents the distance from the object to the viewpoint, and smaller values mean the corresponding objects are closer to the viewpoint.

Blending, Dithering, and Logical Operations

Once an incoming fragment has passed all the tests described in the previous section, it can be combined with the current contents of the color buffer in one of several ways. The simplest way, which is also the default, is to overwrite the existing values. Alternatively, if you're using RGBA mode and you want the fragment to be translucent or antialiased, you might average its value with the value already in the buffer (blending). On systems with a small number of available colors, you might want to dither color values to increase the number of colors available at the cost of a loss in resolution. In the final stage, you can use arbitrary bitwise logical operations to combine the incoming fragment and the pixel that's already written.

Blending

Blending combines the incoming fragment's R, G, B, and alpha values with those of the pixel already stored at the location. Different blending operations can be applied, and the blending that occurs depends on the values of the incoming alpha value and the alpha value (if any) stored at the pixel. (See "Blending" in Chapter 6 for an extensive discussion of this topic.)

Dithering

On systems with a small number of color bitplanes, you can improve the color resolution at the expense of spatial resolution by dithering the color in the image. Dithering is like halftoning in newspapers. Although The New York Times has only two colors—black and white—it can show photographs by representing the shades of gray with combinations of black and white dots. Comparing a newspaper image of a photo (having no shades of gray) with the original photo (with grayscale) makes the loss of spatial resolution obvious. Similarly, systems with a small number of color bitplanes may dither values of red, green, and blue on neighboring pixels for the perception of a wider range of colors.

The dithering operation that takes place is hardware−dependent; all OpenGL allows you to do is to turn it on and off. In fact, on some machines, enabling dithering might do nothing at all, which makes sense if the machine already has high color resolution. To enable and disable dithering, pass

GL_DITHER to *glEnable()* and *glDisable()*. Dithering is enabled by default.

Dithering applies in both RGBA and color−index mode. The colors or color indices alternate in some hardware−dependent way between the two nearest possibilities. For example, in color−index mode, if dithering is enabled and the color index to be painted is 4.4, then 60% of the pixels may be painted with index 4 and 40% of the pixels with index 5. (Many dithering algorithms are possible, but a dithered value produced by any algorithm must depend upon only the incoming value and the fragment's x and y coordinates.) In RGBA mode, dithering is performed separately for each component (including alpha). To use dithering in color−index mode, you generally need to arrange the colors in the color map appropriately in ramps, otherwise, bizarre images might result.

Logical Operations

The final operation on a fragment is the **logical operation**, such as an OR, XOR, or INVERT, which is applied to the incoming fragment values (source) and/or those currently in the color buffer (destination). Such fragment operations are especially useful on bit−blt−type machines, on which the primary graphics operation is copying a rectangle of data from one place in the window to another, from the window to processor memory, or from memory to the window. Typically, the copy doesn't write the data directly into memory but instead allows you to perform an arbitrary logical operation on the incoming data and the data already present; then it replaces the existing data with the results of the operation.

Since this process can be implemented fairly cheaply in hardware, many such machines are available. As an example of using a logical operation, XOR can be used to draw on an image in an undoable way; simply XOR the same drawing again, and the original image is restored. As another example, when using color−index mode, the color indices can be interpreted as bit patterns. Then you can compose an image as combinations of drawings on different layers, use writemasks to limit drawing to different sets of bitplanes, and perform logical operations to modify different layers.

You enable and disable logical operations by passing GL_INDEX_LOGIC_OP or GL_COLOR_LOGIC_OP to *glEnable()* and *glDisable()* for color−index mode or RGBA mode, respectively. You also must choose among the sixteen logical operations with *glLogicOp()*, or you'll just get the effect of the default value, GL_COPY. (For backward compatibility with OpenGL Version 1.0, *glEnable*(GL_LOGIC_OP) also enables logical operation in color−index mode.)

void glLogicOp(GLenum opcode);

Selects the logical operation to be performed, given an incoming (source) fragment and the pixel currently stored in the color buffer (destination). Table 10−4shows the possible values for opcode and their meaning (s represents source and d destination). The default value is GL_COPY.

Parameter	Operation	Parameter	Operation
GL_CLEAR	0	GL_AND	$s \wedge d$
GL_COPY	s	GL_OR	$s \vee d$
GL_NOOP	d	GL_NAND	$\neg(s \wedge d)$
GL_SET	1	GL_NOR	$\neg(s \vee d)$
GL_COPY_INVERTED	$\neg s$	GL_XOR	$s \text{ XOR } d$
GL_INVERT	$\neg d$	GL_EQUIV	$\neg(s \text{ XOR } d)$
GL_AND_REVERSE	$s \wedge \neg d$	GL_AND_INVERTED	$\neg s \wedge d$
GL_OR_REVERSE	$s \vee \neg d$	GL_OR_INVERTED	$\neg s \vee d$

Table 10-4 Sixteen Logical Operations

The Accumulation Buffer

Advanced

The accumulation buffer can be used for such things as scene antialiasing, motion blur, simulating photographic depth of field, and calculating the soft shadows that result from multiple light sources. Other techniques are possible, especially in combination with some of the other buffers. (See *The Accumulation Buffer: Hardware Support for High−Quality Rendering* by Paul Haeberli and Kurt Akeley (SIGGRAPH 1990 Proceedings, p. 309-318) for more information on the uses for the accumulation buffer.)

OpenGL graphics operations don't write directly into the accumulation buffer. Typically, a series of images is generated in one of the standard color buffers, and these are accumulated, one at a time, into the accumulation buffer. When the accumulation is finished, the result is copied back into a color buffer for viewing. To reduce rounding errors, the accumulation buffer may have higher precision (more bits per color) than the standard color buffers. Rendering a scene several times obviously takes longer than rendering it once, but the result is higher quality. You can decide what trade−off between quality and rendering time is appropriate for your application.

You can use the accumulation buffer the same way a photographer can use film for multiple exposures. A photographer typically creates a multiple exposure by taking several pictures of the same scene without advancing the film. If anything in the scene moves, that object appears blurred. Not surprisingly, a computer can do more with an image than a photographer can do with a camera. For example, a computer has exquisite control over the viewpoint, but a photographer can't shake a camera a predictable and controlled amount. (See "Clearing Buffers" for information about how to clear the accumulation buffer; use *glAccum()* to control it.)

void glAccum(GLenum op, GLfloat value);

Controls the accumulation buffer. The op parameter selects the operation, and value is a number to be used in that operation. The possible operations are GL_ACCUM, GL_LOAD, GL_RETURN, GL_ADD, and GL_MULT.

GL_ACCUM reads each pixel from the buffer currently selected for reading with *glReadBuffer()*, multiplies the R, G, B, and alpha values by *value*, and adds the result to the accumulation buffer.

GL_LOAD does the same thing, except that the values replace those in the accumulation buffer rather than being added to them.

GL_RETURN takes values from the accumulation buffer, multiplies them by *value*, and places the result in the color buffer(s) enabled for writing.

GL_ADD and GL_MULT simply add or multiply the value of each pixel in the accumulation buffer by *value* and then return it to the accumulation buffer. For GL_MULT, *value* is clamped to be in the range [−1.0,1.0]. For GL_ADD, no clamping occurs.

Scene Antialiasing

To perform scene antialiasing, first clear the accumulation buffer and enable the front buffer for reading and writing. Then loop several times (say, *n*) through code that jitters and draws the image (**jittering** is moving the image to a slightly different position), accumulating the data with

```
glAccum(GL_ACCUM, 1.0/n);
```

and finally calling

```
glAccum(GL_RETURN, 1.0);
```

Note that this method is a bit faster if, on the first pass through the loop, GL_LOAD is used and clearing the accumulation buffer is omitted. See Table 10−5for possible jittering values. With this code, the image is drawn *n* times before the final image is drawn. If you want to avoid showing the user the intermediate images, draw into a color buffer that's not displayed, accumulate from that, and use the GL_RETURN call to draw into a displayed buffer (or into a back buffer that you subsequently swap to the front).

You could instead present a user interface that shows the viewed image improving as each additional piece is accumulated and that allows the user to halt the process when the image is good enough. To accomplish this, in the loop that draws successive images, call *glAccum()* with GL_RETURN after each accumulation, using 16.0/1.0, 16.0/2.0, 16.0/3.0, ... as the second argument. With this technique, after one pass, 1/16 of the final image is shown, after two passes, 2/16 is shown, and so on. After the GL_RETURN, the code should check to see if the user wants to interrupt the process. This interface is slightly slower, since the resultant image must be copied in after each pass.

To decide what *n* should be, you need to trade off speed (the more times you draw the scene, the longer it takes to obtain the final image) and quality (the more times you draw the scene, the smoother it gets, until you make maximum use of the accumulation buffer's resolution). "Plate 22" and "Plate 23" show improvements made using scene antialiasing.

Example 10−2defines two routines for jittering that you might find useful: *accPerspective()* and *accFrustum()*. The routine *accPerspective()* is used in place of *gluPerspective()*, and the first four parameters of both routines are the same. To jitter the viewing frustum for scene antialiasing, pass the *x* and *y* jitter values (of less than one pixel) to the fifth and sixth parameters of *accPerspective()*. Also pass 0.0 for the seventh and eighth parameters to *accPerspective()* and a nonzero value for the ninth parameter (to prevent division by zero inside *accPerspective()*). These last three parameters are used for depth−of−field effects, which are described later in this chapter.

Example 10−2 Routines for Jittering the Viewing Volume: accpersp.c

```
#define PI_ 3.14159265358979323846

void accFrustum(GLdouble left, GLdouble right, GLdouble bottom,
    GLdouble top, GLdouble near, GLdouble far, GLdouble pixdx,
    GLdouble pixdy, GLdouble eyedx, GLdouble eyedy,
    GLdouble focus)
{
    GLdouble xwsize, ywsize;
    GLdouble dx, dy;
    GLint viewport[4];

    glGetIntegerv (GL_VIEWPORT, viewport);
```

```
    xwsize = right - left;
    ywsize = top - bottom;
    dx = -(pixdx*xwsize/(GLdouble) viewport[2] +
            eyedx*near/focus);
    dy = -(pixdy*ywsize/(GLdouble) viewport[3] +
            eyedy*near/focus);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum (left + dx, right + dx, bottom + dy, top + dy,
        near, far);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef (-eyedx, -eyedy, 0.0);
}

void accPerspective(GLdouble fovy, GLdouble aspect,
    GLdouble near, GLdouble far, GLdouble pixdx, GLdouble pixdy,
    GLdouble eyedx, GLdouble eyedy, GLdouble focus)
{
    GLdouble fov2,left,right,bottom,top;
    fov2 = ((fovy*PI_) / 180.0) / 2.0;

    top = near / (fcos(fov2) / fsin(fov2));
    bottom = -top;
    right = top * aspect;
    left = -right;

    accFrustum (left, right, bottom, top, near, far,
        pixdx, pixdy, eyedx, eyedy, focus);
}
```

Example 10–3uses these two routines to perform scene antialiasing.

Example 10–3 Scene Antialiasing: accpersp.c

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

void init(void)
{
   GLfloat mat_ambient[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat light_position[] = { 0.0, 0.0, 10.0, 1.0 };
```

```
   GLfloat lm_ambient[] = { 0.2, 0.2, 0.2, 1.0 };

   glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialf(GL_FRONT, GL_SHININESS, 50.0);
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
   glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lm_ambient);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
   glShadeModel (GL_FLAT);

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

void displayObjects(void)
{
   GLfloat torus_diffuse[] = { 0.7, 0.7, 0.0, 1.0 };
   GLfloat cube_diffuse[] = { 0.0, 0.7, 0.7, 1.0 };
   GLfloat sphere_diffuse[] = { 0.7, 0.0, 0.7, 1.0 };
   GLfloat octa_diffuse[] = { 0.7, 0.4, 0.4, 1.0 };

   glPushMatrix ();
   glTranslatef (0.0, 0.0, -5.0);
   glRotatef (30.0, 1.0, 0.0, 0.0);

   glPushMatrix ();
   glTranslatef (-0.80, 0.35, 0.0);
   glRotatef (100.0, 1.0, 0.0, 0.0);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, torus_diffuse);
   glutSolidTorus (0.275, 0.85, 16, 16);
   glPopMatrix ();

   glPushMatrix ();
   glTranslatef (-0.75, -0.50, 0.0);
   glRotatef (45.0, 0.0, 0.0, 1.0);
   glRotatef (45.0, 1.0, 0.0, 0.0);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, cube_diffuse);
   glutSolidCube (1.5);
   glPopMatrix ();

   glPushMatrix ();
   glTranslatef (0.75, 0.60, 0.0);
   glRotatef (30.0, 1.0, 0.0, 0.0);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, sphere_diffuse);
```

```
   glutSolidSphere (1.0, 16, 16);
   glPopMatrix ();

   glPushMatrix ();
   glTranslatef (0.70, -0.90, 0.25);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, octa_diffuse);
   glutSolidOctahedron ();
   glPopMatrix ();

   glPopMatrix ();
}

#define ACSIZE  8

void display(void)
{
   GLint viewport[4];
   int jitter;

   glGetIntegerv (GL_VIEWPORT, viewport);

   glClear(GL_ACCUM_BUFFER_BIT);
   for (jitter = 0; jitter < ACSIZE; jitter++) {
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      accPerspective (50.0,
         (GLdouble) viewport[2]/(GLdouble) viewport[3],
         1.0, 15.0, j8[jitter].x, j8[jitter].y, 0.0, 0.0, 1.0);
      displayObjects ();
      glAccum(GL_ACCUM, 1.0/ACSIZE);
   }
   glAccum (GL_RETURN, 1.0);
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
}

/*  Main Loop
 *  Be certain you request an accumulation buffer.
 */
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB
```

```
                         | GLUT_ACCUM | GLUT_DEPTH);
   glutInitWindowSize (250, 250);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init();
   glutReshapeFunc(reshape);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

You don't have to use a perspective projection to perform scene antialiasing. You can antialias a scene with orthographic projection simply by using *glTranslate*()* to jitter the scene. Keep in mind that *glTranslate*()* operates in world coordinates, but you want the apparent motion of the scene to be less than one pixel, measured in screen coordinates. Thus, you must reverse the world−coordinate mapping by calculating the jittering translation values, using its width or height in world coordinates divided by its viewport size. Then multiply that world−coordinate value by the amount of jitter to determine how much the scene should be moved in world coordinates to get a predictable jitter of less than one pixel. Example 10−4shows how the *display()* and *reshape()* routines might look with a world−coordinate width and height of 4.5.

Example 10−4 Jittering with an Orthographic Projection: accanti.c

```
#define ACSIZE  8

void display(void)
{
   GLint viewport[4];
   int jitter;

   glGetIntegerv (GL_VIEWPORT, viewport);

   glClear(GL_ACCUM_BUFFER_BIT);
   for (jitter = 0; jitter < ACSIZE; jitter++) {
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glPushMatrix ();
/*      Note that 4.5 is the distance in world space between
 *      left and right and bottom and top.
 *      This formula converts fractional pixel movement to
 *      world coordinates.
 */
      glTranslatef (j8[jitter].x*4.5/viewport[2],
                    j8[jitter].y*4.5/viewport[3], 0.0);
      displayObjects ();
      glPopMatrix ();
      glAccum(GL_ACCUM, 1.0/ACSIZE);
   }
   glAccum (GL_RETURN, 1.0);
```

```
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho (-2.25, 2.25, -2.25*h/w, 2.25*h/w, -10.0, 10.0);
   else
      glOrtho (-2.25*w/h, 2.25*w/h, -2.25, 2.25, -10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
```

Motion Blur

Similar methods can be used to simulate motion blur, as shown in "Plate 7" in Appendix I and Figure 10−2 Suppose your scene has some stationary and some moving objects in it, and you want to make a motion−blurred image extending over a small interval of time. Set up the accumulation buffer in the same way, but instead of spatially jittering the images, jitter them temporally. The entire scene can be made successively dimmer by calling

```
glAccum (GL_MULT, decayFactor);
```

as the scene is drawn into the accumulation buffer, where *decayFactor* is a number from 0.0 to 1.0. Smaller numbers for *decayFactor* cause the object to appear to be moving faster. You can transfer the completed scene with the object's current position and "vapor trail" of previous positions from the accumulation buffer to the standard color buffer with

```
glAccum (GL_RETURN, 1.0);
```

The image looks correct even if the items move at different speeds, or if some of them are accelerated. As before, the more jitter points (temporal, in this case) you use, the better the final image, at least up to the point where you begin to lose resolution due to finite precision in the accumulation buffer. You can combine motion blur with antialiasing by jittering in both the spatial and temporal domains, but you pay for higher quality with longer rendering times.

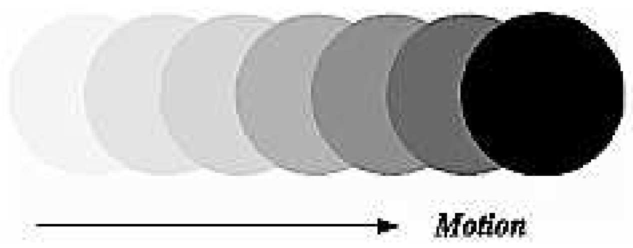

Figure 10−2 Motion−Blurred Object

Depth of Field

A photograph made with a camera is in perfect focus only for items lying on a single plane a certain distance from the film. The farther an item is from this plane, the more out of focus it is. The depth of field for a camera is a region about the plane of perfect focus where items are out of focus by a small enough amount.

Under normal conditions, everything you draw with OpenGL is in focus (unless your monitor's bad, in which case everything is out of focus). The accumulation buffer can be used to approximate what you would see in a photograph where items are more and more blurred as their distance from a plane of perfect focus increases. It isn't an exact simulation of the effects produced in a camera, but the result looks similar to what a camera would produce.

To achieve this result, draw the scene repeatedly using calls with different argument values to *glFrustum()*. Choose the arguments so that the position of the viewpoint varies slightly around its true position and so that each frustum shares a common rectangle that lies in the plane of perfect focus, as shown in Figure 10−3 The results of all the renderings should be averaged in the usual way using the accumulation buffer.

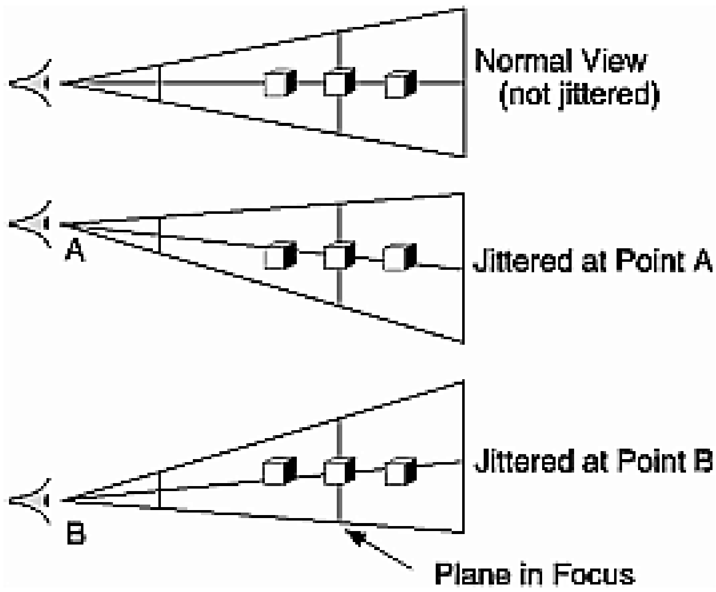

Figure 10−3 Jittered Viewing Volume for Depth−of−Field Effects

"Plate 10" in Appendix I shows an image of five teapots drawn using the depth−of−field effect. The gold teapot (second from the left) is in focus, and the other teapots get progressively blurrier, depending upon their distance from the focal plane (gold teapot). The code to draw this image is shown in Example 10−5(which assumes *accPerspective()* and *accFrustum()* are defined as described in Example 10−2). The scene is drawn eight times, each with a slightly jittered viewing volume, by calling *accPerspective()*. As you recall, with scene antialiasing, the fifth and sixth parameters jitter the viewing volumes in the *x* and *y* directions. For the depth−of−field effect, however, you want to jitter the volume while holding it stationary at the focal plane. The focal plane is the depth value defined by the ninth (last) parameter to *accPerspective()*, which is $z = 5.0$ in this example. The amount of blur is determined by multiplying the *x* and *y* jitter values (seventh and eighth parameters of *accPerspective()*) by a constant. Determining the constant is not a science; experiment with values until the depth of field is as pronounced as you want. (Note that in Example 10−5, the fifth and sixth parameters to *accPerspective()* are set to 0.0, so scene antialiasing is turned off.)

Example 10–5 Depth–of–Field Effect: dof.c

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

void init(void)
{
   GLfloat ambient[] = { 0.0, 0.0, 0.0, 1.0 };
   GLfloat diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat position[] = { 0.0, 3.0, 3.0, 0.0 };

   GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
   GLfloat local_view[] = { 0.0 };

   glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
   glLightfv(GL_LIGHT0, GL_POSITION, position);

   glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
   glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, local_view);

   glFrontFace (GL_CW);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_AUTO_NORMAL);
   glEnable(GL_NORMALIZE);
   glEnable(GL_DEPTH_TEST);

   glClearColor(0.0, 0.0, 0.0, 0.0);
   glClearAccum(0.0, 0.0, 0.0, 0.0);
/*  make teapot display list */
   teapotList = glGenLists(1);
   glNewList (teapotList, GL_COMPILE);
   glutSolidTeapot (0.5);
   glEndList ();
}

void renderTeapot (GLfloat x, GLfloat y, GLfloat z,
   GLfloat ambr, GLfloat ambg, GLfloat ambb,
   GLfloat difr, GLfloat difg, GLfloat difb,
   GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine)
{
```

```
   GLfloat mat[4];

   glPushMatrix();
   glTranslatef (x, y, z);
   mat[0] = ambr; mat[1] = ambg; mat[2] = ambb; mat[3] = 1.0;
   glMaterialfv (GL_FRONT, GL_AMBIENT, mat);
   mat[0] = difr; mat[1] = difg; mat[2] = difb;
   glMaterialfv (GL_FRONT, GL_DIFFUSE, mat);
   mat[0] = specr; mat[1] = specg; mat[2] = specb;
   glMaterialfv (GL_FRONT, GL_SPECULAR, mat);
   glMaterialf (GL_FRONT, GL_SHININESS, shine*128.0);
   glCallList(teapotList);
   glPopMatrix();
}

void display(void)
{
   int jitter;
   GLint viewport[4];

   glGetIntegerv (GL_VIEWPORT, viewport);
   glClear(GL_ACCUM_BUFFER_BIT);

   for (jitter = 0; jitter < 8; jitter++) {
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      accPerspective (45.0,
         (GLdouble) viewport[2]/(GLdouble) viewport[3],
         1.0, 15.0, 0.0, 0.0,
         0.33*j8[jitter].x, 0.33*j8[jitter].y, 5.0);

/*      ruby, gold, silver, emerald, and cyan teapots   */
      renderTeapot (-1.1, -0.5, -4.5, 0.1745, 0.01175,
                    0.01175, 0.61424, 0.04136, 0.04136,
                    0.727811, 0.626959, 0.626959, 0.6);
      renderTeapot (-0.5, -0.5, -5.0, 0.24725, 0.1995,
                    0.0745, 0.75164, 0.60648, 0.22648,
                    0.628281, 0.555802, 0.366065, 0.4);
      renderTeapot (0.2, -0.5, -5.5, 0.19225, 0.19225,
                    0.19225, 0.50754, 0.50754, 0.50754,
                    0.508273, 0.508273, 0.508273, 0.4);
      renderTeapot (1.0, -0.5, -6.0, 0.0215, 0.1745, 0.0215,
                    0.07568, 0.61424, 0.07568, 0.633,
                    0.727811, 0.633, 0.6);
      renderTeapot (1.8, -0.5, -6.5, 0.0, 0.1, 0.06, 0.0,
                    0.50980392, 0.50980392, 0.50196078,
                    0.50196078, 0.50196078, .25);
      glAccum (GL_ACCUM, 0.125);
```

```
   }
   glAccum (GL_RETURN, 1.0);
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
}

/*  Main Loop
 *  Be certain you request an accumulation buffer.
 */
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB
                        | GLUT_ACCUM | GLUT_DEPTH);
   glutInitWindowSize (400, 400);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init();
   glutReshapeFunc(reshape);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

Soft Shadows

To accumulate soft shadows due to multiple light sources, render the shadows with one light turned on at a time, and accumulate them together. This can be combined with spatial jittering to antialias the scene at the same time. (See "Shadows" in Chapter 14 for more information about drawing shadows.)

Jittering

If you need to take nine or sixteen samples to antialias an image, you might think that the best choice of points is an equally spaced grid across the pixel. Surprisingly, this is not necessarily true. In fact, sometimes it's a good idea to take points that lie in adjacent pixels. You might want a uniform distribution or a normalized distribution, clustering toward the center of the pixel. (The aforementioned SIGGRAPH paper discusses these issues.) In addition, Table 10−5shows a few sets of reasonable jittering values to be used for some selected sample counts. Most of the examples in the table are uniformly distributed in the pixel, and all lie within the pixel.

Count	Values
2	{0.25, 0.75}, {0.75, 0.25}
3	{0.5033922635, 0.8317967229}, {0.7806016275, 0.2504380877}, {0.2261828938, 0.4131553612}

4	{0.375, 0.25}, {0.125, 0.75}, {0.875, 0.25}, {0.625, 0.75}
5	{0.5, 0.5}, {0.3, 0.1}, {0.7, 0.9}, {0.9, 0.3}, {0.1, 0.7}
6	{0.4646464646, 0.4646464646}, {0.1313131313, 0.7979797979}, {0.5353535353, 0.8686868686}, {0.8686868686, 0.5353535353}, {0.7979797979, 0.1313131313}, {0.2020202020, 0.2020202020}
8	{0.5625, 0.4375}, {0.0625, 0.9375}, {0.3125, 0.6875}, {0.6875, 0.8125}, {0.8125, 0.1875}, {0.9375, 0.5625}, {0.4375, 0.0625}, {0.1875, 0.3125}
9	{0.5, 0.5}, {0.1666666666, 0.9444444444}, {0.5, 0.1666666666}, {0.5, 0.8333333333}, {0.1666666666, 0.2777777777}, {0.8333333333, 0.3888888888}, {0.1666666666, 0.6111111111}, {0.8333333333, 0.7222222222}, {0.8333333333, 0.0555555555}
12	{0.4166666666, 0.625}, {0.9166666666, 0.875}, {0.25, 0.375}, {0.4166666666, 0.125}, {0.75, 0.125}, {0.0833333333, 0.125}, {0.75, 0.625}, {0.25, 0.875}, {0.5833333333, 0.375}, {0.9166666666, 0.375}, {0.0833333333, 0.625}, {0.583333333, 0.875}
16	{0.375, 0.4375}, {0.625, 0.0625}, {0.875, 0.1875}, {0.125, 0.0625}, {0.375, 0.6875}, {0.875, 0.4375}, {0.625, 0.5625}, {0.375, 0.9375}, {0.625, 0.3125}, {0.125, 0.5625}, {0.125, 0.8125}, {0.375, 0.1875}, {0.875, 0.9375}, {0.875, 0.6875}, {0.125, 0.3125}, {0.625, 0.8125}

Table 10−5(continued) Sample Jittering Values