Additionally, multisampling using sample shading can add a lot more work in computing the color of a pixel. If your system has four samples per pixels, you've quadrupled the work per pixel in rasterizing primitives, which can potentially hinder your application's performance. **glMinSampleShading()** controls how many samples per pixel receive individually shaded values (i.e., each executing its own version of the bound fragment shader at the sample location). Reducing the minimum-sample-shading ratio can help improve performance in applications bound by the speed at which it can shade fragments.

We'll visit multisampling again in "Testing and Operating on Fragments" on Page 156, because a fragment's alpha value can be modified by the results of shading at sample locations.

Testing and Operating on Fragments

When you draw geometry on the screen, OpenGL starts processing it by executing the currently bound vertex shader; then the tessellation, and geometry shaders, if they're bound; and then assembles the final geometry into primitives that get sent to the rasterizer, which figures out which pixels in the window are affected. After OpenGL determines that an individual fragment should be generated, its fragment shader is executed, and then several processing stages, which control how and whether the fragment is drawn as a pixel into the framebuffer, remain. For example, if the fragment is outside a rectangular region or if it's farther from the *viewpoint* than the pixel that's already in the framebuffer, its processing is stopped, and it's not 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. Most of these tests and operations are enabled and disabled using **glEnable()** and **glDisable()**, respectively. The tests and operations occur in the following order—if a fragment is eliminated in an enabled earlier test, none of the later enabled tests or operations are executed:

1. Scissor test
2. Multisample fragment operations
3. Stencil test
4. Depth test

5. Blending
6. Dithering
7. Logical operations

All of these tests and operations are described in detail in the following subsections.

Note: As we'll see in "Framebuffer Objects" on Page 180, we can render into multiple buffers at the same time. For many of the fragment tests and operations, they can be controlled on a per-buffer basis, as well as for all of the buffers collectively. In many cases, we describe both the OpenGL function that will set the operation for all buffers, as well as the routine for affecting a single buffer. In most cases, the single buffer version of a function will have an **'i'** appended to the function's name.

Scissor Test

The first additional test you can enable to control fragment visibility is the scissor test. The *scissor box* is a rectangular portion of your window and restricts all drawing to its region. You specify the scissor box using the **glScissor()** command, and enable the test by specifying GL_SCISSOR_TEST with **glEnable()**. 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.

All rendering—including clearing the window—is restricted to the scissor box if the test is enabled (as compared to the viewport, which doesn't limit screen clears). 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()**.

Multisample Fragment Operations

By default, multisampling calculates fragment coverage values that are independent of alpha. However, if you **glEnable()** one of the following special modes, then a fragment's alpha value is taken into consideration when calculating the coverage, assuming that multisampling itself is enabled and that there is a multisample buffer associated with the framebuffer. The special modes are as follows:

- GL_SAMPLE_ALPHA_TO_COVERAGE uses the alpha value of the fragment in an implementation-dependent manner to compute the final coverage value.
- GL_SAMPLE_ALPHA_TO_ONE sets the fragment's alpha value the maximum alpha value, and then uses that value in the coverage calculation.
- GL_SAMPLE_COVERAGE uses the value set with the **glSampleCoverage()** routine, which is combined (ANDed) with the calculated coverage value. Additionally, the generated sample mask can be inverted by setting the invert flag with the **glSampleCoverage()** routine.

void **glSampleCoverage**(GLfloat *value*, GLboolean *invert*);

Sets parameters to be used to interpret alpha values while computing multisampling coverage. *value* is a temporary coverage value that is used if GL_SAMPLE_COVERAGE or GL_SAMPLE_ALPHA_TO_COVERAGE has been enabled. *invert* is a Boolean that indicates whether the temporary coverage value ought to be bitwise inverted before it is used (ANDed) with the fragment coverage.

- GL_SAMPLE_MASK specifies an exact bit-representation for the coverage mask (as compared to it being generated by the OpenGL implementation). This mask is once again ANDed with the sample coverage for the fragment. The sample mask is specified using the **glSampleMaski()** function.

void **glSampleMaski**(GLuint *index*, GLbitfield *mask*);

Sets one 32-bit word of the sample mask, *mask*. The word to set is specified by *index* and the new value of that word is specified by *mask*. As samples are written to the framebuffer, only those whose corresponding bits in the current sample mask will be updated and the rest will be discarded.

The sample mask can also be specified in a fragment shader by writing to the `gl_SampleMask` variable. Details of using `gl_SampleMask` are covered in "Built-in GLSL Variables and Functions".

Stencil Test

The stencil test takes place only if there is a stencil buffer, which you need to request when your window is created. (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 can be 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*);
void **glStencilFuncSeparate**(GLenum *face*, GLenum *func*,
GLint *ref*, GLuint *mask*);

Sets the comparison function (*func*), the reference value (*ref*), and a mask (*mask*) for use with the stencil test. The reference value is compared with the value in the stencil buffer using the comparison function, but the comparison applies only to those bits for which 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-order *s* 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 zero, *mask* is all ones, and stenciling is disabled.

glStencilFuncSeparate() allows separate stencil function parameters to be specified for front- and back-facing polygons (as set with **glCullFace()**).

---

void **glStencilOp**(GLenum *fail*, GLenum *zfail*, GLenum *zpass*);
void **glStencilOpSeparate**(GLenum *face*, 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_INCR_WRAP, GL_DECR, GL_DECR_WRAP, or GL_INVERT. They correspond to keeping the current value, replacing it with zero, replacing it with the reference value, incrementing it with saturation, incrementing it without saturation, decrementing it with saturation, decrementing it without saturation, 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 holds s 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* is applied if the depth test passes, or if no depth test is performed. By default, all three stencil operations are GL_KEEP.

glStencilOpSeparate() allows separate stencil tests to be specified for front- and back-facing polygons (as set with **glCullFace()**).

"With saturation" means that the stencil value will clamp to extreme values. If you try to decrement zero with saturation, the stencil value remains zero. "Without saturation" means that going outside the indicated range wraps around. If you try to decrement zero without saturation, the stencil value becomes the maximum unsigned integer value (quite large!).

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 4.2.

You can also determine whether the stencil test is enabled by passing GL_STENCIL_TEST to **glIsEnabled()**.

Table 4.2 Query Values for the Stencil Test

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

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. To do this, fill the stencil mask with zeros, and then draw the desired shape in the stencil buffer with ones. 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). 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 set 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 4.5 Using the Stencil Test: stencil.c

```
void
init(void)
{
   ...// Set up our vertex arrays and such

   // Set the stencil's clear value
   glClearStencil(0x0);

   glEnable(GL_DEPTH_TEST);
   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 sphere where the stencil is 1
   glStencilFunc(GL_EQUAL, 0x1, 0x1);
   glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
   drawSphere();

   // draw the tori where the stencil is not 1
   glStencilFunc(GL_NOTEQUAL, 0x1, 0x1);
   drawTori();
}

// Whenever the window is reshaped, redefine the
// coordinate system and redraw the stencil area.

void
reshape(int width, int height)
{
   glViewport(0, 0, width, height);

   // create a diamond shaped stencil area
   glClear(GL_STENCIL_BUFFER_BIT);
   glStencilFunc(GL_ALWAYS, 0x1, 0x1);
   glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
   drawMask();
}
```

Example 4.5 demonstrates 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.

The following examples illustrate other uses of the stencil test.

1. 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 see 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 always to 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.

2. Stippling—Suppose you want to draw an image with a *stipple* pattern. 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 is 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 value 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" on Page 146.) You can also choose a different comparison function for the depth test with **glDepthFunc()**.

void **glDepthFunc**(GLenum *func*);

Sets the comparison fun 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 that the corresponding objects are closer to the viewpoint.

More context is provided in "OpenGL Transformations" in Chapter 5 for setting a depth range.

Polygon Offset

If you want to highlight the edges of a solid object, you might draw the object with polygon mode set to GL_FILL, and then draw it again, but in a different color and with the polygon mode set to GL_LINE. However, because lines and filled polygons are not rasterized in exactly the same way, the depth values generated for the line and polygon edge are usually not the same, 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.

This undesirable effect can be eliminated by using polygon offset, which adds an appropriate offset to force coincident *z*-values apart, separating a polygon edge from its highlighting line. (The stencil buffer, 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 by 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, and 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 modified by adding a calculated offset value before the depth test is performed. The offset value is calculated by

$$offset = m \cdot factor + r \cdot units$$

where m is the maximum depth slope of the polygon (computed during rasterization), and r is the smallest value guaranteed to produce a resolvable difference in depth values and is an implementation-specific constant. Both *factor* and *units* may be negative.

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

OpenGL calculates the depth slope, as illustrated in Figure 4.2, which is the z (depth) value divided by the change in either the x- or y-coordinates as you traverse the polygon. The depth values are clamped to the range $[0, 1]$, and the x- and y-coordinates are in window coordinates. To estimate the maximum depth slope of a polygon (m in the offset equation above), use the formula

$$m = \sqrt{\left(\frac{\partial z}{\partial x}\right)^2 + \left(\frac{\partial z}{\partial y}\right)^2}$$

or an implementation may use the approximation

$$m = \max\left(\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}\right)$$

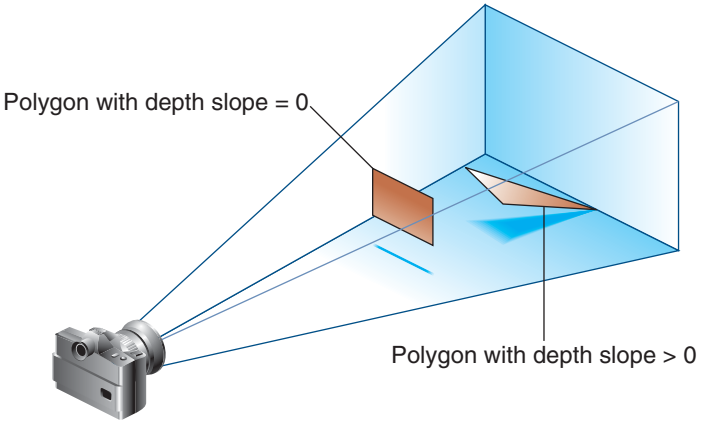

Figure 4.2 Polygons and their depth slopes

For polygons that are parallel to the near and far clipping planes, the depth slope is zero. Those polygons can use a small constant offset, which you can specify by setting *factor* = 0.0 and *units* = 1.0 in your call 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. A small, nonzero value for *factor*, such as 0.75 or 1.0, is probably enough to generate distinct depth values and eliminate the unpleasant visual artifacts.

In some situations, the simplest values for *factor* and *units* (1.0 and 1.0) aren't the answer. For instance, if the widths of the lines that are highlighting the edges are greater than 1, then increasing the value of *factor* may be necessary. Also, since depth values while using a perspective projection are unevenly transformed into window coordinates, less offset is needed for polygons that are closer to the near clipping plane, and more offset is needed for polygons that are farther away. You may need to experiment with the values you pass to **glPolygonOffset()** to get the result you're looking for.

Blending

Once an incoming fragment has passed all of the enabled fragment tests, 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, which admittedly isn't much of a combination. Alternatively, you might want to combine the color present in the framebuffer with the incoming fragment color—a process called blending. Most often, blending is associated with the fragment's *alpha value* (or commonly just alpha), but that's not a strict requirement. We've mentioned alpha several times but haven't given it a proper description. Alpha is the fourth color component, and all colors in OpenGL have an alpha value (even if you don't explicitly set one). However, you don't see alpha, but rather you see alpha's effect: it's a measure of translucency, and is what's used when you want to simulate translucent objects, like colored glass for example.

However, unless you enable blending by calling **glEnable()** with GL_BLEND, or employ advanced techniques like order-independent transparency (discussed in "Order-Independent Transparency" in Chapter 11), alpha is pretty much ignored by the OpenGL pipeline. You see, just like the real world, where color of a translucent object is a combination of that object's color with the colors of all the objects you see behind it. For OpenGL to do something useful with alpha, the pipeline needs more information than the current primitive's color (which is the color output from the fragment shader); it needs to know what color is already present for that pixel in the framebuffer.

Blending Factors

In basic blending mode, the incoming fragment's color is linearly combined with the current pixel's color. As with any linear combination, *coefficients* control the contributions of each term. For blending in OpenGL, those coefficients are called the *source-* and *destination-blending factors*. The source-blending factor is associated with the color output from the fragment shader, and similarly, the destination-blending factor is associated with the color in the framebuffer.

If we let (S_r, S_g, S_b, S_a) represent the source-blending factors, and likewise let (D_r, D_g, D_b, D_a) represent the destination factors, and use (R_s, G_s, B_s, A_s), and (R_d, G_d, B_d, A_d) represent the colors of the source fragment and destination pixel respectively, the blending equation yields a final color of

$$(S_r R_s + D_r R_d,\ S_g G_s + D_g G_d,\ S_b B_s + D_b B_d,\ S_a A_s + D_a A_d)$$

The default blending operation is addition, but we'll see in "The Blending Equation" on Page 170 that we can also control the blending operator.

Controlling Blending Factors

You have two different ways to choose the source and destination blending factors. You may call **glBlendFunc()** and choose two blending factors: the first factor for the source RGBA and the second for the destination RGBA. Or, you may use **glBlendFuncSeparate()** and choose four blending factors, which allows you to use one blending operation for RGB and a different one for its corresponding alpha.

Note: We also list the functions **glBlendFunci()** and **glBlendFuncSeparatei()**, which are used when you're drawing to multiple buffers simultaneously. This is an advanced topic that we describe in "Framebuffer Objects" on Page 180, but since the functions are virtually identical actions to **glBlendFunc()** and **glBlendFuncSeparate()**, we include them here.

---

void **glBlendFunc**(GLenum *srcfactor*, GLenum *destfactor*);
void **glBlendFunci**(GLuint *buffer*, GLenum *srcfactor*,
GLenum *destfactor*);

Controls how color values in the fragment being processed (the source) are combined with those already stored in the framebuffer (the destination). The possible values for these arguments are explained in

Table 4.3. The argument *srcfactor* indicates how to compute a source blending factor; *destfactor* indicates how to compute a destination blending factor.

glBlendFunc() specifies the blending factors for all drawable buffers, while **glBlendFunci()** specifies the blending factors only for buffer *buffer*.

The blending factors are clamped to either the range $[0, 1]$ or $[-1, 1]$ for unsigned-normalized or signed-normalized framebuffer formats respectively. If the framebuffer format is floating point, then no clamping of factors occurs.

---

void **glBlendFuncSeparate**(GLenum *srcRGB*, GLenum *destRGB*,
GLenum *srcAlpha*,
GLenum *destAlpha*);
void **glBlendFuncSeparatei**(GLuint *buffer*, GLenum *srcRGB*,
GLenum *destRGB*, GLenum *srcAlpha*,
GLenum *destAlpha*);

Similar to **glBlendFunc()**, **glBlendFuncSeparate()** also controls how source color values (fragment) are combined with destination values (in the framebuffer). **glBlendFuncSeparate()** also accepts the same arguments (shown in Table 4.3) as **glBlendFunc()**. The argument *srcRGB* indicates the source-blending factor for color values; *destRGB* is the destination-blending factor for color values. The argument *srcAlpha* indicates the source-blending factor for alpha values; *destAlpha* is the destination-blending factor for alpha values.

glBlendFuncSeparatei() specifies the blending factors for all drawable buffers, while **glBlendFuncSeparatei()** specifies the blending factors only for buffer *buffer*.

Note: In Table 4.3, the values with the subscript $_{s1}$ are for dual-source blending factors, which are described in "Dual-Source Blending" on Page 198.

If you use one of the GL_CONSTANT blending functions, you need to use **glBlendColor()** to specify the constant color.

Table 4.3 Source and Destination Blending Factors

Constant	RGB Blend Factor	Alpha Blend Factor
GL_ZERO	$(0, 0, 0)$	0
GL_ONE	$(1, 1, 1)$	1
GL_SRC_COLOR	(R_s, G_s, B_s)	A_s
GL_ONE_MINUS_SRC_COLOR	$(1, 1, 1) - (R_s, G_s, B_s)$	$1 - A_s$
GL_DST_COLOR	(R_d, G_d, B_d)	A_d
GL_ONE_MINUS_DST_COLOR	$(1, 1, 1) - (R_d, G_d, B_d)$	$1 - A_d$
GL_SRC_ALPHA	(A_s, A_s, A_s)	A_s
GL_ONE_MINUS_SRC_ALPHA	$(1, 1, 1) - (A_s, A_s, A_s)$	$1 - A_s$
GL_DST_ALPHA	(A_d, A_d, A_d)	A_d
GL_ONE_MINUS_DST_ALPHA	$(1, 1, 1) - (A_d, A_d, A_d)$	$1 - A_d$
GL_CONSTANT_COLOR	(R_c, G_c, B_c)	A_c
GL_ONE_MINUS_CONSTANT_COLOR	$(1, 1, 1) - (R_c, G_c, B_c)$	$1 - A_c$
GL_CONSTANT_ALPHA	(A_c, A_c, A_c)	A_c
GL_ONE_MINUS_CONSTANT_ALPHA	$(1, 1, 1) - (A_c, A_c, A_c)$	$1 - A_c$
GL_SRC_ALPHA_SATURATE	$(f, f, f), f = \min(A_s, 1 - A_d)$	1
GL_SRC1_COLOR	(R_{s1}, G_{s1}, B_{s1})	A_{s1}
GL_ONE_MINUS_SRC1_COLOR	$(1, 1, 1) - (R_{s1}, G_{s1}, B_{s1})$	$1 - A_{s1}$
GL_SRC1_ALPHA	(A_{s1}, A_{s1}, A_{s1})	A_{s1}
GL_ONE_MINUS_SRC1_ALPHA	$(1, 1, 1) - (A_{s1}, A_{s1}, A_{s1})$	$1 - A_{s1}$

void **glBlendColor**(GLclampf *red*, GLclampf *green*, GLclampf *blue*, GLclampf *alpha*);

Sets the current *red*, *blue*, *green*, and *alpha* values for use as the constant color (R_c, G_c, B_c, A_c) in blending operations.

Similarly, use **glDisable()** with GL_BLEND to disable blending. Note that using the constants GL_ONE (as the source factor) and GL_ZERO (for the destination factor) gives the same results as when blending is disabled; these values are the default.

Advanced

OpenGL has the ability to render into multiple buffers simultaneously (see "Writing to Multiple Renderbuffers Simultaneously" on Page 193 for details). All buffers can have blending enabled and disabled simultaneously (using **glEnable()** and **glDisable()**). Blending settings can be managed on a per-buffer basis using **glEnablei()** and **glDisablei()**.

The Blending Equation

With standard blending, colors in the framebuffer are combined (using addition) with incoming fragment colors to produce the new framebuffer color. Either **glBlendEquation()** or **glBlendEquationSeparate()** may be used to select other mathematical operations to compute the difference, minimum, or maximum between color fragments and framebuffer pixels.

---

void **glBlendEquation**(GLenum *mode*);
void **glBlendEquationi**(GLuint *buffer*, GLenum *mode*);

Specifies how framebuffer and source colors are blended together. The allowable values for *mode* are GL_FUNC_ADD (the default), GL_FUNC_SUBTRACT, GL_FUNC_REVERSE_SUBTRACT, GL_MIN, and GL_MAX. The possible modes are described in Table 4.4.

glBlendEquation() specifies the blending mode for all buffers, while **glBlendEquationi()** sets the mode for the buffer specified by the *buffer* argument, which is the integer index of the buffer.

---

void **glBlendEquationSeparate**(GLenum *modeRGB*,
GLenum *modeAlpha*);
void **glBlendEquationSeparatei**(GLuint *buffer*,
GLenum *modeRGB*,
GLenum *modeAlpha*);

Specifies how framebuffer and source colors are blended together, but allows for different blending modes for the rgb and alpha color components. The allowable values for *modeRGB* and *modeAlpha* are identical for the modes accepted by **glBlendEquation()**.

Again, **glBlendEquationSeparate()** sets the blending modes for all buffers, while **glBlendEquationSeparatei()** sets the modes for the buffer whose index is specified in *buffer*.

In Table 4.4, C_s and C_d represent the source and destination colors. The S and D parameters in the table represent the source- and destination-blending factors as specified with **glBlendFunc()** or **glBlendFuncSeparate()**.

Table 4.4 Blending Equation Mathematical Operations

Blending Mode Parameter	Mathematical Operation
GL_FUNC_ADD	$C_sS + C_dD$
GL_FUNC_SUBTRACT	$C_sS - C_dD$
GL_FUNC_REVERSE_SUBTRACT	$C_dD - C_sS$
GL_MIN	$\min(C_sS, C_dD)$
GL_MAX	$\max(C_sS, C_dD)$

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 half-toning 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 appearance 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.

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 examplese of using a logical operation, XOR can be used to draw on an image in a revertible way; simply XOR the same drawing again, and the original image is restored.

You enable and disable logical operations by passing GL_COLOR_LOGIC_OP to **glEnable()** and **glDisable()**. You also must choose among the 16 logical operations with **glLogicOp()**, or you'll just get the effect of the default value, GL_COPY.

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 4.5 shows the possible values for opcode and their meaning (s represents source and d destination). The default value is GL_COPY.

Table 4.5 Sixteen Logical Operations

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 *XOR* d
GL_INVERT	$\neg d$	GL_EQUIV	$\neg(s$ *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	$\neg s \vee d$

For floating-point buffers, or those in sRGB format, logical operations are ignored.

Occlusion Query

Advanced

The depth buffer determines visibility on a per-pixel basis. For performance reasons, it would be nice to be able to determine if a *geometric object* is visible before sending all of its (perhaps complex) geometry for rendering. *Occlusion query*s enable you to determine if a representative set of geometry will be visible after depth testing.

This is particularly useful for complex geometric objects with many polygons. Instead of rendering all of the geometry for a complex object, you might render its bounding box or another simplified representation that require less rendering resources. If OpenGL returns that no fragments or samples would have been modified by rendering that piece of geometry, you know that none of your complex object will be visible for that frame, and you can skip rendering that object for the frame.

The following steps are required to utilize occlusion queries:

1. Generate a query id for each occlusion query that you need.
2. Specify the start of an occlusion query by calling **glBeginQuery()**.
3. Render the geometry for the occlusion test.
4. Specify that you've completed the occlusion query by calling **glEndQuery()**.
5. Retrieve the number of, or if any, samples passed the depth tests.

In order to make the occlusion query process as efficient as possible, you'll want to disable all rendering modes that will increase the rendering time but won't change the visibility of a pixel.

Generating Query Objects

In order to use queries, you'll first need to request identifiers for your query tests. **glGenQueries()** will generate the requested number of unused query ids for your subsequent use.

void **glGenQueries**(GLsizei *n*, GLuint **ids*);

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

The names returned are marked as used for the purposes of allocating additional query objects, but only acquire valid state once they have been specified in a call to **glBeginQuery()**.

Zero is a reserved occlusion query object name and is never returned as a valid value by **glGenQueries()**.

You can also determine if an identifier is currently being used as an occlusion query by calling **glIsQuery()**.

GLboolean **glIsQuery**(GLuint *id*);

Returns GL_TRUE if *id* is the name of an occlusion query object. Returns GL_FALSE if *id* is zero or if *id* is a nonzero value that is not the name of a buffer object.

Initiating an Occlusion Query Test

To specify geometry that's to be used in an occlusion query, merely bracket the rendering operations between calls to **glBeginQuery()** and **glEndQuery()**, as demonstrated in Example 4.6

Example 4.6 Rendering Geometry with Occlusion Query: occquery.c

```
glBeginQuery(GL_SAMPLES_PASSED, Query);
glDrawArrays(GL_TRIANGLES, 0, 3);
glEndQuery(GL_SAMPLES_PASSED);
```

All OpenGL operations are available while an occlusion query is active, with the exception of **glGenQueries()** and **glDeleteQueries()**, which will raise a GL_INVALID_OPERATION error.

void **glBeginQuery**(GLenum *target*, GLuint *id*);

Specifies the start of an occlusion query operation. *target* must be GL_SAMPLES_PASSED, GL_ANY_SAMPLES_PASSED, or GL_ANY_SAMPLES_PASSED_CONSERVATIVE. *id* is an unsigned integer identifier for this occlusion query operation.

void **glEndQuery**(GLenum *target*);

Ends an occlusion query. *target* must be GL_SAMPLES_PASSED, or GL_ANY_SAMPLES_PASSED.

Determining the Results of an Occlusion Query

Once you've completed rendering the geometry for the occlusion query, you need to retrieve the results. This is done with a call to **glGetQueryObjectiv()** or **glGetQueryObjectuiv()**, as shown in Example 4.7, which will return the number of fragments, or samples, if you're using multisampling.

void **glGetQueryObjectiv**(GLenum *id*, GLenum *pname*, GLint **params*);
void **glGetQueryObjectuiv**(GLenum *id*, GLenum *pname*, GLuint **params*);

Queries the state of an occlusion query object. *id* is the name of a query object. If *pname* is GL_QUERY_RESULT, then *params* will contain the number of fragments or samples (if multisampling is enabled) that passed the depth test, with a value of zero representing the object being entirely occluded.

There may be a delay in completing the occlusion query operation. If *pname* is GL_QUERY_RESULT_AVAILABLE, *params* will contain GL_TRUE if the results for query *id* are available, or GL_FALSE otherwise.

Example 4.7 Retrieving the Results of an Occlusion Query

```
count = 1000; /* counter to avoid a possible infinite loop */

while (!queryReady && count-) {
    glGetQueryObjectiv(Query, GL_QUERY_RESULT_AVAILABLE, &queryReady);
}

if (queryReady) {
    glGetQueryObjectiv(Query, GL_QUERY_RESULT, &samples);
    cerr << "Samples rendered: " << samples << endl;
}
else {
    cerr << " Result not ready ... rendering anyways" << endl;
    samples = 1; /* make sure we render */
}
```

```
if (samples > 0) {
    glDrawArrays(GL_TRIANGLE_FAN}, 0, NumVertices);
}
```

Cleaning Up Occlusion Query Objects

After you've completed your occlusion query tests, you can release the resources related to those queries by calling **glDeleteQueries()**.

void **glDeleteQueries**(GLsizei *n*, const GLuint ***ids*);

Deletes *n* occlusion query objects, named by elements in the array *ids*. The freed query objects may now be reused (for example, by **glGenQueries()**).

Conditional Rendering

Advanced

One of the issues with occlusion queries is that they require OpenGL to pause processing geometry and fragments, count the number of affected samples in the depth buffer, and return the value to your application. Stopping modern graphics hardware in this manner usually catastrophically affects performance in performance-sensitive applications. To eliminate the need to pause OpenGL's operation, *conditional rendering* allows the graphics server (hardware) to decide if an occlusion query yielded any fragments, and to render the intervening commands. Conditional rendering is enabled by surrounding the rendering operations you would have conditionally executed using the results of **glGetQuery***().

void **glBeginConditionalRender**(GLuint *id*, GLenum *mode*);
void **glEndConditionalRender**(void);

Delineates a sequence of OpenGL rendering commands that may be discarded based on the results of the occlusion query object *id*. *mode* specifies how the OpenGL implementation uses the results of the occlusion query, and must be one of: GL_QUERY_WAIT, GL_QUERY_NO_WAIT, GL_QUERY_BY_REGION_WAIT, or GL_QUERY_BY_REGION_NO_WAIT.

A GL_INVALID_VALUE is set if *id* is not an existing occlusion query. A GL_INVALID_OPERATION is generated if **glBeginConditionalRender()** is called while a conditional-rendering sequence is in operation;

if **glEndConditionalRender()** is called when no conditional render is underway; if *id* is the name of an occlusion query object with a target different than GL_SAMPLES_PASSED; or if *id* is the name of an occlusion query in progress.

The code shown in Example 4.8 completely replaces the sequence of code in Example 4.7. Not only is it the code more compact, it is far more efficient as it completely removes the results query to the OpenGL server, which is a major performance inhibitor.

Example 4.8 Rendering Using Conditional Rendering

```
glBeginConditionalRender(Query, GL_QUERY_WAIT);
glDrawArrays(GL_TRIANGLE_FAN, 0, NumVertices);
glEndConditionalRender();
```

You may have noticed that there is a *mode* parameter to **glBeginConditionalRender()**, which may be one of GL_QUERY_WAIT, GL_QUERY_NO_WAIT, GL_QUERY_BY_REGION_WAIT, or GL_QUERY_BY_REGION_NO_WAIT. These modes control whether the GPU will wait for the results of a query to be ready before continuing to render, and whether it will consider global results or results only pertaining to the region of the screen that contributed to the original occlusion query result.

- If *mode* is GL_QUERY_WAIT then the GPU will wait for the result of the occlusion query to be ready before determining whether it will continue with rendering.
- If *mode* is GL_QUERY_NO_WAIT then the GPU may not wait for the result of the occlusion query to be ready before continuing to render. If the result is not ready, then it may choose to render the part of the scene contained in the conditional rendering section anyway.
- If *mode* is GL_QUERY_BY_REGION_WAIT then the GPU will wait for anything that contributes to the region covered by the controled rendering to be completed. It may still wait for the complete occlusion query result to be ready.
- If *mode* is GL_QUERY_BY_REGION_NO_WAIT, then the GPU will discard any rendering in regions of the framebuffer that contributed no samples to the occlusion query, but may choose to render into other regions if the result was not available in time.

By using these modes wisely, you can improve performance of the system. For example, waiting for the results of an occlusion query may actually

take more time than just rendering the conditional part of the scene. In particular, if it is expected that most results will mean that some rendering should take place, then on aggregate, it may be faster to always use one of the NO_WAIT modes even if it means more rendering will take place overall.

Per-Primitive Antialiasing

You might have noticed in some of your OpenGL images that lines, especially nearly horizontal and 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 one antialiasing technique for reducing it. Figure 4.3 shows two intersecting lines, both aliased and antialiased. The pictures have been magnified to show the effect.

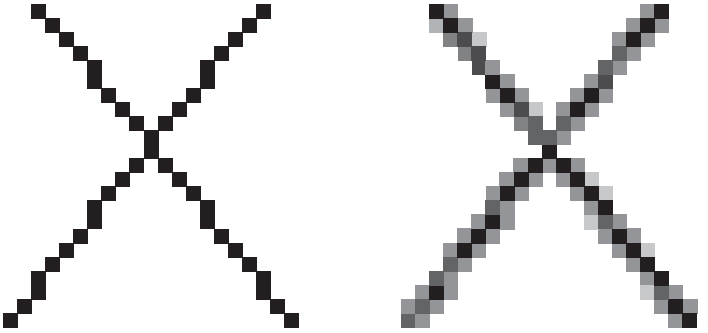

Figure 4.3 Aliased and antialiased lines

Figure 4.3 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. 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.

The details of calculating coverage values are complex, and difficult to specify in general. In fact, computations 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 4.6. 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 OpenGL implementation can ignore them entirely.

Table 4.6 Values for Use with **glHint()**

Parameter	Specifies
GL_LINE_SMOOTH_HINT	Line antialiasing quality
GL_POLYGON_SMOOTH_HINT	Polygon edge antialiasing quality
GL_TEXTURE_COMPRESSION_HINT	Quality and performance of texture-image *compression* (See Chapter 6, "Textures" for more detail)
GL_FRAGMENT_SHADER_DERIVATIVE_HINT	Derivative accuracy for fragment processing built-in functions `dFdx`, `dFdy`, and `fwidth` (See Appendix C for more details)

We've discussed multisampling before as a technique for antialiasing; however, it's not usually the best solution for lines. Another way to antialias lines, and polygons if the multisample results are quite what you want, is to turn on antialiasing with **glEnable()**, and passing in GL_LINE_SMOOTH or GL_POLYGON_SMOOTH, as appropriate. You might also want to provide a quality hint with **glHint()**. We'll describe the steps for each type of primitive that can be antialiased in the next sections.

Antialiasing Lines

First, 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. However, in most cases, the ordering can be ignored without significant adverse effects.

Example 4.9 shows the initialization for line antialiasing.

Example 4.9 Setting Up Blending for Antialiasing Lines: antilines.cpp

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

Antialiasing Polygons

Antialiasing the edges of filled polygons is similar to antialiasing lines. When different polygons have overlapping edges, you need to blend the color values appropriately.

To antialias polygons, 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.

In order to have edges blend appropriately, 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.

Note: Antialiasing can be adversely affected when using the depth buffer, in that pixels may be discarded when they should have been blended. To ensure proper blending and antialiasing, you'll need to disable the depth buffer.

Framebuffer Objects

Advanced

Up to this point, all of our discussion regarding buffers has focused on the buffers provided by the windowing system, as you requested when you

called **glutCreateWindow()** (and configured by your call to **glutInitDisplayMode()**). Although you can quite successfully use any technique with just those buffers, quite often various operations require moving data between buffers superfluously. This is where framebuffer objects enter the picture. Using framebuffer objects, you can create our own framebuffers and use their attached renderbuffers to minimize data copies and optimize performance.

Framebuffer objects are quite useful for performing off-screen-rendering, updating texture maps, and engaging in *buffer ping-ponging* (a data-transfer techniques used in *GPGPU*).

The framebuffer that is provided by the windowing system is the only framebuffer that is available to the display system of your graphics server—that is, it is the only one you can see on your screen. It also places restrictions on the use of the buffers that were created when your window opened. By comparison, the framebuffers that your application creates cannot be displayed on your *monitor*; they support only *off-screen rendering*.

Another difference between window-system-provided framebuffers and framebuffers you create is that those managed by the window system allocate their buffers—color, depth, and stencil—when your window is created. When you create an application-managed framebuffer object, you need to create additional renderbuffers that you associate with the framebuffer objects you created. The buffers with the window-system-provided buffers can never be associated with an application-created framebuffer object, and vice versa.

To allocate an application-generated framebuffer object name, you need to call **glGenFramebuffers()**, which will allocate an unused identifier for the framebuffer object.

void **glGenFramebuffers**(GLsizei *n*, GLuint **ids*);

Allocate *n* unused framebuffer object names, and return those names in *ids*.

A GL_INVALID_VALUE error will be generated if *n* is negative.

Allocating a framebuffer object name doesn't actually create the framebuffer object or allocate any storage for it. Those tasks are handled through a call to **glBindFramebuffer()**. **glBindFramebuffer()** operates in a similar manner to many of the other **glBind*()** routines you've seen in OpenGL. The first time it is called for a particular framebuffer, it causes

storage for the object to be allocated and initialized. Any subsequent calls will bind the provided framebuffer object name as the active one.

void **glBindFramebuffer**(GLenum *target*, GLuint *framebuffer*);

Specifies a framebuffer for either reading or writing. When target is GL_DRAW_FRAMEBUFFER, *framebuffer* specifies the destination framebuffer for rendering. Similarly, when *target* is set to GL_READ_FRAMEBUFFER, *framebuffer* specifies the source of read operations. Passing GL_FRAMEBUFFER for target sets both the read and write framebuffer bindings to *framebuffer*.

framebuffer must either be zero, which binds *target* to the default window-system-provided framebuffer, or a framebuffer object generated by a call to **glGenFramebuffers()**.

A GL_INVALID_OPERATION error is generated if *framebuffer* is neither zero nor a valid framebuffer object previously generated by calling **glGenFramebuffers()** but not deleted by calling **glDeleteFramebuffers()**.

As with all of the other objects you have encountered in OpenGL, you can release an application-allocated framebuffer by calling **glDeleteFramebuffers()**. That function will mark the framebuffer object's name as unallocated and release any resources associated with the framebuffer object.

void **glDeleteFramebuffers**(GLsizei *n*, const GLuint **ids*);

Deallocates the *n* framebuffer objects associated with the names provided in *ids*. If a framebuffer object is currently bound (i.e., its name was passed to the most recent call to **glBindFramebuffer()**) and is deleted, the framebuffer target is immediately bound to *id* zero (the window-system provided framebuffer), and the framebuffer object is released.

A GL_INVALID_VALUE error is generated by **glDeleteFramebuffers()** if *n* is negative. Passing unused names or zero does not generate any errors; they are simply ignored.

For completeness, you can determine whether a particular unsigned integer is an application-allocated framebuffer object by calling **glIsFramebuffer()**:

GLboolean **glIsFramebuffer**(GLuint *framebuffer*);

Returns GL_TRUE if *framebuffer* is the name of a framebuffer returned from **glGenFramebuffers()**. Returns GL_FALSE if *framebuffer* is zero (the window-system default framebuffer) or a value that's either unallocated or been deleted by a call to **glDeleteFramebuffers()**.

void **glFramebufferParameteri**(GLenum *target*, GLenum *pname*, GLint *param*);

Sets parameters of a framebuffer object, when the framebuffer object has no attachments, otherwise the values for these parameters are specified by the framebuffer attachments.

target must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER. *pname* specifies the parameter of the framebuffer object bound to target to set, and must be one of GL_FRAMEBUFFER_DEFAULT_WIDTH, GL_FRAMEBUFFER_DEFAULT_HEIGHT, GL_FRAMEBUFFER_DEFAULT_LAYERS, GL_FRAMEBUFFER_DEFAULT_SAMPLES, or GL_FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS.

Once a framebuffer object is created, you still can't do much with it, generally speaking. You need to provide a place for drawing to go and reading to come from; those places are called *framebuffer attachment*s. We'll discuss those in more detail after we examine renderbuffers, which are one type of buffer you can attach to a framebuffer object.

Renderbuffers

Renderbuffers are effectively memory managed by OpenGL that contains formatted image data. The data that a renderbuffer holds takes meaning once it is attached to a framebuffer object, assuming that the format of the image buffer matches what OpenGL is expecting to render into (e.g., you can't render colors into the depth buffer).

As with many other buffers in OpenGL, the process of allocating and deleting buffers is similar to what you've seen before. To create a new renderbuffer, you would call **glGenRenderbuffers()**.

void **glGenRenderbuffers**(GLsizei *n*, GLuint **ids*);

Allocate *n* unused renderbuffer object names, and return those names in *ids*. Names are unused until bound with a call to **glBindRenderbuffer()**.

Likewise, a call to **glDeleteRenderbuffers()** will release the storage associated with a renderbuffer.

void **glDeleteRenderbuffers**(GLsizei *n*, const GLuint **ids*);

Deallocates the *n* renderbuffer objects associated with the names provided in *ids*. If one of the renderbuffers is currently bound and passed to **glDeleteRenderbuffers()**, a binding of zero replaces the binding at the current framebuffer attachment point, in addition to the renderbuffer being released.

No errors are generated by **glDeleteRenderbuffers()**. Unused names or zero are simply ignored.

Likewise, you can determine whether a name represents a valid renderbuffer by calling **glIsRenderbuffer()**.

void **glIsRenderbuffer**(GLuint *renderbuffer*);

Returns GL_TRUE if *renderbuffer* is the name of a renderbuffer returned from **glGenRenderbuffers()**. Returns GL_FALSE if *framebuffer* is zero (the window-system default framebuffer) or a value that's either unallocated or deleted by a call to **glDeleteRenderbuffers()**.

Similar to the process of binding a framebuffer object so that you can modify its state, you call **glBindRenderbuffer()** to affect a renderbuffer's creation and to modify the state associated with it, which includes the format of the image data that it contains.

void **glBindRenderbuffer**(GLenum *target*, GLuint *renderbuffer*);

Creates a renderbuffer and associates it with the name *renderbuffer*. *target* must be GL_RENDERBUFFER. *renderbuffer* must either be zero, which removes any renderbuffer binding, or a name that was generated by a call to **glGenRenderbuffers()**; otherwise, a GL_INVALID_OPERATION error will be generated.

Creating Renderbuffer Storage

When you first call **glBindRenderbuffer()** with an unused renderbuffer name, the OpenGL server creates a renderbuffer with all its state information set to the default values. In this configuration, no storage has been allocated to store image data. Before you can attach a renderbuffer to a framebuffer and render into it, you need to allocate storage and specify its image format. This is done by calling either **glRenderbufferStorage()** or **glRenderbufferStorageMultisample()**.

---

void **glRenderbufferStorage**(GLenum *target*,
GLenum *internalformat*,
GLsizei *width*, GLsizei *height*);

void **glRenderbufferStorageMultisample**(GLenum *target*,
GLsizei *samples*,
GLenum *internalformat*,
GLsizei *width*,
GLsizei *height*);

Allocates storage for image data for the bound renderbuffer. *target* must be GL_RENDERBUFFER. For a color-renderable buffer, *internalformat* must be one of:

GL_RED	GL_R8	GL_R16
GL_RG	GL_RG8	GL_RG16
GL_RGB	GL_R3_G3_B2	GL_RGB4
GL_RGB5	GL_RGB8	GL_RGB10
GL_RGB12	GL_RGB16	GL_RGBA
GL_RGBA2	GL_RGBA4	GL_RGB5_A1
GL_RGBA8	GL_RGB10_A2	GL_RGBA12
GL_RGBA16	GL_SRGB	GL_SRGB8
GL_SRGB_ALPHA	GL_SRGB8_ALPHA8	GL_R16F

GL_R32F	GL_RG16F	GL_RG32F
GL_RGB16F	GL_RGB32F	GL_RGBA16F
GL_RGBA32F	GL_R11F_G11F_B10F	GL_RGB9_E5
GL_R8I	GL_R8UI	GL_R16I
GL_R16UI	GL_R32I	GL_R32UI
GL_RG8I	GL_RG8UI	GL_RG16I
GL_RG16UI	GL_RG32I	GL_RG32UI
GL_RGB8I	GL_RGB8UI	GL_RGB16I
GL_RGB16UI	GL_RGB32I	GL_RGB32UI
GL_RGBA8I	GL_RGBA8UI	GL_RGBA16I
GL_RGBA16UI	GL_RGBA32I	GL_R8_SNORM
GL_R16_SNORM	GL_RG8_SNORM	GL_RG16_SNORM
GL_RGB8_SNORM	GL_RGB16_SNORM	GL_RGBA8_SNORM
GL_RGBA16_SNORM		

To use a renderbuffer as a depth buffer, it must be depth-renderable, which is specified by setting internalformat to either GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16, GL_DEPTH_COMPONENT32, GL_DEPTH_COMPONENT32, or GL_DEPTH_COMPONENT32F.

For use exclusively as a stencil buffer, internalformat should be specified as either GL_STENCIL_INDEX, GL_STENCIL_INDEX1, GL_STENCIL_INDEX4, GL_STENCIL_INDEX8, or GL_STENCIL_INDEX16.

For packed depth-stencil storage, internalformat must be GL_DEPTH_STENCIL, which allows the renderbuffer to be attached as the depth buffer, stencil buffer, or at the combined depth-stencil attachment point.

width and *height* specify the size of the renderbuffer in pixels, and samples specifies the number of multisample samples per pixel. Setting samples to zero in a call to **glRenderbufferStorageMultisample()** is identical to calling **glRenderbufferStorage()**.

A GL_INVALID_VALUE is generated if width or height is greater than the value returned when querying GL_MAX_RENDERBUFFER_SIZE, or if *samples* is greater than the value returned when querying GL_MAX_SAMPLES. A GL_INVALID_OPERATION is generated if internalformat is a signed- or unsigned-integer format (e.g., a format containing a "I", or "UI" in its token), and *samples* is not zero, and the implementation doesn't support multisampled integer buffers. Finally, if the renderbuffer size and format combined exceed the available memory able to be allocated, then a GL_OUT_OF_MEMORY error is generated.

Example 4.10 Creating a 256 × 256 RGBA Color Renderbuffer

```
glGenRenderbuffers(1, &color);
glBindRenderbuffer(GL_RENDERBUFFER, color);
glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA, 256, 256);
```

Once you have created storage for your renderbuffer as shown in Example 4.10, you need to attach it to a framebuffer object before you can render into it.

Framebuffer Attachments

When you render, you can send the results of that rendering to a number of places:

- The color buffer to create an image, or even multiple color buffers if you're using multiple render targets (see "Writing to Multiple Renderbuffers Simultaneously" on Page 193).
- The depth buffer to store occlusion information.
- The stencil buffer for storing per-pixel masks to control rendering. Each of those buffers represents a framebuffer attachment, to which you can attach suitable image buffers that you later render into, or read from. The possible framebuffer attachment points are listed in Table 4.7.

Table 4.7 Framebuffer Attachments

Attachment Name	Description
GL_COLOR_ATTACHMENT*i*	The i^{th} color buffer. *i* can range from zero (the default color buffer) to GL_MAX_COLOR_ATTACHMENTS - 1
GL_DEPTH_ATTACHMENT	The depth buffer
GL_STENCIL_ATTACHMENT	The stencil buffer
GL_DEPTH_STENCIL_ATTACHMENT	A special attachment for packed depth-stencil buffers (which require the renderbuffer to have been allocated as a GL_DEPTH_STENCIL pixel format)

Currently, there are two types of rendering surfaces you can associate with one of those attachments: renderbuffers and a level of a texture image.

We'll first discuss attaching a renderbuffer to a framebuffer object, which is done by calling **glFramebufferRenderbuffer()**.

void **glFramebufferRenderbuffer**(GLenum *target*,
GLenum *attachment*,
GLenum *renderbuffertarget*,
GLuint *renderbuffer*);

Attaches *renderbuffer* to attachment of the currently bound framebuffer object. *target* must either be GL_READ_FRAMEBUFFER, GL_DRAW_FRAMEBUFFER, or GL_FRAMEBUFFER (which is equivalent to GL_DRAW_FRAMEBUFFER).

attachment is one of GL_COLOR_ATTACHMENT*i*, GL_DEPTH_ATTACHMENT, GL_STENCIL_ATTACHMENT, or GL_DEPTH_STENCIL_ATTACHMENT.

renderbuffertarget must be GL_RENDERBUFFER, and *renderbuffer* must either be zero, which removes any renderbuffer attachment at *attachment*, or a renderbuffer name returned from **glGenRenderbuffers**(), or a GL_INVALID_OPERATION error is generated.

In Example 4.11, we create and attach two renderbuffers: one for color, and the other for depth. We then proceed to render, and finally copy the results back to the window-system-provided framebuffer to display the results. You might use this technique to generate frames for a movie rendering off-screen, where you don't have to worry about the visible framebuffer being corrupted by overlapping windows or someone resizing the window and interrupting rendering.

One important point to remember is that you might need to reset the viewport for each framebuffer before rendering, particularly if the size of your application-defined framebuffers differs from the window-system provided framebuffer.

Example 4.11 Attaching a Renderbuffer for Rendering

```
enum { Color, Depth, NumRenderbuffers };

GLuint framebuffer, renderbuffer[NumRenderbuffers]

void
init()
{
     glGenRenderbuffers(NumRenderbuffers, renderbuffer);
     glBindRenderbuffer(GL_RENDERBUFFER, renderbuffer[Color]);
     glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA, 256, 256);

     glBindRenderbuffer(GL_RENDERBUFFER, renderbuffer[Depth]);
```

```
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24, 256, 256);

    glGenFramebuffers(1, &framebuffer);
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, framebuffer);

    glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                              GL_RENDERBUFFER, renderbuffer[Color]);

    glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                              GL_RENDERBUFFER, renderbuffer[Depth]);

    glEnable(GL_DEPTH_TEST);
}

void
display()
{
    // Prepare to render into the renderbuffer

    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, framebuffer);

    glViewport(0, 0, 256, 256);

    // Render into renderbuffer

    glClearColor(1.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    ...

    // Set up to read from the renderbuffer and draw to
    //   window-system framebuffer

    glBindFramebuffer(GL_READ_FRAMEBUFFER, framebuffer);
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);

    glViewport(0, 0, windowWidth, windowHeight);
    glClearColor(0.0, 0.0, 1.0, 1.0);

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    /* Do the copy */

    glBlitFramebuffer(0, 0, 255, 255, 0, 0, 255, 255,
                      GL_COLOR_BUFFER_BIT, GL_NEAREST);
    glutSwapBuffers();
}
```

Framebuffer Completeness

Given the myriad of combinations between texture and buffer formats, and between framebuffer attachments, various situations can arise that prevent the completion of rendering when you are using application-defined framebuffer objects. After modifying the attachments to a framebuffer object, it's best to check the framebuffer's status by calling **glCheckFramebufferStatus()**.

GLenum **glCheckFramebufferStatus**(GLenum *target*);

Returns one of the framebuffer completeness status enums listed in Table 4.8. *target* must be one of GL_READ_FRAMEBUFFER, GL_DRAW_FRAMEBUFFER, or GL_FRAMEBUFFER (which is equivalent to GL_DRAW_FRAMEBUFFER).

If **glCheckFramebufferStatus()** generates an error, zero is returned.

The errors representing the various violations of framebuffer configurations are listed in Table 4.8.

Of the listed errors, GL_FRAMEBUFFER_UNSUPPORTED is very implementation dependent, and may be the most complicated to debug.

Advanced

glClear(GL_COLOR_BUFFER_BIT) will clear all of the bound color buffers (we have see in "Framebuffer Objects" on Page 180 how to configure multiple color buffers). You can use the **glClearBuffer*()** commands to clear individual buffers.

If you're using multiple draw buffers—particularly those that have floating-point or nonnormalized integer pixel formats—you can clear each individually bound buffer using **glClearBuffer*()** functions. Unlike functions such as **glClearColor()** and **glClearDepth()**, which set a clear value within OpenGL that's used when **glClear()** is called, **glClearBuffer*()** uses the values passed to it to immediately clear the bound drawing buffers. Additionally, to reduce the number of function calls associated with using multiple draw buffers, you can call **glClearBufferfi()** to simultaneously clear the depth and stencil buffers (which is effectively equivalent to calling **glClearBuffer*()** twice—once for the depth buffer and once for the stencil buffer).

Table 4.8 Errors Returned by **glCheckFramebufferStatus()**

Framebuffer Completeness Status Enum	Description
GL_FRAMEBUFFER_COMPLETE	The framebuffer and its attachments match the rendering or reading state required.
GL_FRAMEBUFFER_UNDEFINED	The bound framebuffer is specified to be the default framebuffer (i.e., **glBindFramebuffer()** with zero specified as the framebuffer), and the default framebuffer doesn't exist.
GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT	A necessary attachment to the bound framebuffer is uninitialized
GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT	There are no images (e.g., texture layers or renderbuffers) attached to the framebuffer.
GL_FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER	Every drawing buffer (e.g., GL_DRAW_BUFFER*i* as specified by **glDrawBuffers()**) has an attachment.
GL_FRAMEBUFFER_INCOMPLETE_READ_BUFFER	An attachment exists for the buffer specified for the buffer specified by **glReadBuffer()**.
GL_FRAMEBUFFER_UNSUPPORTED	The combination of images attached to the framebuffer object is incompatible with the requirements of the OpenGL implementation.
GL_FRAMEBUFFER_INCOMPLETE_MULTISAMPLE	The number of samples for all images across the framebuffer's attachments do not match.

void **glClearBuffer**{fi ui}v(GLenum *buffer*, GLint *drawbuffer*,
const TYPE **value*);
void **glClearBufferfi**(GLenum *buffer*, GLint *drawbuffer*,
GLfloat *depth*, GLint *stencil*);

Clears the buffer indexed by *drawbuffer* associated with *buffer* to *value*. *buffer* must be one of GL_COLOR, GL_DEPTH, or GL_STENCIL.

If *buffer* is GL_COLOR, *drawbuffer* specifies an index to a particular draw buffer, and *value* is a four-element array containing the clear color. If the buffer indexed by *drawbuffer* has multiple draw buffers (as specified by a call the **glDrawBuffers()**), all draw buffers are cleared to *value*.

If *buffer* is GL_DEPTH or GL_STENCIL, *drawbuffer* must be zero, and *value* is a single-element array containing an appropriate clear value (subject to clamping and type conversion for depth values, and masking and type conversion for stencil values). Use only **glClearBufferfv()** for clearing the depth buffer, and **glClearBufferiv()** for clearing the stencil buffer.

glClearBufferfi() can be used to clear both the depth and stencil buffers simultaneously. *buffer* in this case must be GL_DEPTH_STENCIL.

GL_INVALID_ENUM is generated by **glClearbuffer{if ui}v** if *buffer* is not one of the accepted values listed above. GL_INVALID_ENUM is generated by **glClearBufferfi()** if *buffer* is not GL_DEPTH_STENCIL. GL_INVALID_VALUE is generated if *buffer* is GL_COLOR, and *drawbuffer* is less than zero, or greater than or equal to GL_MAX_DRAW_BUFFERS; or if *buffer* is GL_DEPTH, GL_STENCIL, or GL_DEPTH_STENCIL and *drawbuffer* is not zero.

Invalidating Framebuffers

Implementations of OpenGL (including OpenGL ES on mobile or embedded devices, most often) may work in limited memory environments. Framebuffers have the potential of taking up considerable memory resources (particularly for multiple, multisampled color attachments and textures). OpenGL provides a mechanism to state that a region or all of a framebuffer is no longer needed and can be released. This operation is done with either **glInvalidateSubFramebuffer()** or **glInvalidateFramebuffer()**.

void **glInvalidateFramebuffer**(GLenum *target*,
GLsizei *numAttachments*,
const GLenum **attachments*);
void **glInvalidateSubFramebuffer**(GLenum *target*,
GLsizei *numAttachmens*,
const GLenum **attachments*,
GLint *x*, GLint *y*,
GLsizei *width*, GLsizei *height*);

Specifies that a portion, or the entirety, of the bound framebuffer object are not necessary to preserve. For either function, *target* must be either GL_DRAW_FRAMEBUFFER, GL_READ_FRAMEBUFFER, or GL_FRAMEBUFFER specifying both the draw and read targets at the same time. *attachments* provides a list of attachment tokens: GL_COLOR_ATTACHMENT*i*, GL_DEPTH_ATTACHMENT, or GL_STENCIL_ATTACHMENT; and *numAttachments* specifies how many entries are in the *attachments* list.

For **glInvalidateSubFramebuffer()**, the region specified by lower-left corner (x, y) with width *width*, and height *height* (measured from (x, y)), is marked as invalid for all attachments in *attachments*.

Various errors are returned from the calls: A GL_INVALID_ENUM is generated if any tokens are not from those listed above; A GL_INVALID_OPERATION is generated if an index of an attachment (e.g., *i* from GL_COLOR_ATTACHMENT*i*) is greater than or equal to the maximum number of color attachments; A GL_INVALID_VALUE is generated if any of *numAttachments*, *width*, or *height* are negative.

Writing to Multiple Renderbuffers Simultaneously

Advanced

One feature of using framebuffer objects with multiple renderbuffer (or textures, as described in Chapter 6, "Textures") is the ability to write to multiple buffers from a fragment shader simultaneously, often called *MRT* (for *multiple-render target*) rendering. This is mostly a performance optimization, saving processing the same list of vertices multiple times and rasterizing the same primitives multiple times.

While this technique is used often in GPGPU, it can also be used when generating geometry and other information (like textures or normal map) which is written to different buffers during the same rendering pass. Enabling this technique requires setting up a framebuffer object with multiple color (and potentially depth and stencil) attachments, and modification of the fragment shader. Having just discussed setting up multiple attachments, we'll focus on the fragment shader here.

As we've discussed, fragment shaders output values through their `out` variables. In order to specify the correspondence between `out` variables and framebuffer attachments, we simply need to use the `layout` qualifier to direct values to the right places. For instance, Example 4.12 demonstrates associating two variables with color attachment locations zero and one.

Example 4.12 Specifying `layout` Qualifiers for MRT Rendering

```
layout (location = 0) out vec4 color;
layout (location = 1) out vec4 normal;
```

If the attachments of the currently bound framebuffer don't match those of the currently bound fragment shader, misdirected data (i.e., fragment shader data written to an attachment with nothing attached) accumulates in dark corners of the universe, but is otherwise ignored.

Additionally, if you're using dual-source blending (see "Dual-Source Blending" on Page 198), with MRT rendering, you merely specify both the `location` and `index` options to the `layout` directive.

Using the `layout` qualifier within a shader is the preferred way to associate fragment shader outputs with framebuffer attachments, but if they are not specified, then OpenGL will do the assignments during shader linking. You can direct the linker to make the appropriate associations by using the **glBindFragDataLocation()**, or **glBindFragDataLocationIndexed()** if you need to also specify the fragment `index`. Fragment shader bindings specified in the shader source will be used if specified, regardless of whether a location was specified using one of these functions.

void **glBindFragDataLocation**(GLuint *program*,
GLuint *colorNumber*,
const GLchar **name*);
void **glBindFragDataLocationIndexed**(GLuint *program*,
GLuint *colorNumber*,
GLuint *index*,
const GLchar **name*);

Uses the value in *color* for fragment shader variable *name* to specify the output location associated with shader *program*. For the indexed case, *index* specifies the output index as well as the location.

A GL_INVALID_VALUE is generated if *program* is not a shader program, or if either *index* is greater than one, or if *colorNumber* is greater than or equal to the maximum number of color attachments.

After a program is linked, you can retrieve a fragment shader variable's output location, and source index, if applicable, by calling either **glGetFragDataLocation()**, or **glGetFragDataIndex()**.

GLint **glGetFragDataLocation**(GLuint *program*,
const GLchar **name*);
GLint **glGetFragDataIndex**(GLuint *program*,
const GLchar **name*);

Returns either the location or index of a fragment shader variable *name* associated with the linked shader program *program*.

A −1 is returned if: *name* is not the name of applicable variable for *program*; if *program* successfully linked, but doesn't have an associated fragment shader; or if *program* has not yet been, or failed, linking. In the last case, a GL_INVALID_OPERATION error is also generated.

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, or back-right for the default framebuffer, or
- front, or any renderbuffer attachment for a user-defined framebuffer object.

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()**, or **glDrawBuffers()** to select the buffers to be written and **glReadBuffer()** to select the buffer as the source for **glReadPixels()**, **glCopyTexImage*()**, and **glCopyTexSubImage*()**.

---

void **glDrawBuffer**(GLenum *mode*);
void **glDrawBuffers**(GLsizei *n*, const GLenum **buffers*);

Selects the color buffers enabled for writing or clearing and disables buffers enabled by previous calls to **glDrawBuffer()** or **glDrawBuffers()**. 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_NONE
GL_BACK	GL_FRONT_RIGHT	GL_FRONT_AND_BACK
GL_LEFT	GL_BACK_LEFT	GL_COLOR_ATTACHMENT*i*
GL_RIGHT	GL_BACK_RIGHT	

If *mode*, or the entries in *buffers* is not one of the above, a GL_INVALID_ENUM error is generated. Additionally, if a framebuffer object is bound that is not the default framebuffer, then only GL_NONE and GL_COLOR_ATTACHMENT*i* are accepted, otherwise a GL_INVALID_ENUM error is generated.

Arguments that omit LEFT or RIGHT refer to both the left and right stereo buffers; similarly, arguments that omit FRONT or BACK refer to both.

By default, mode is GL_BACK for double-buffered contexts.

The **glDrawBuffers()** routine specifies multiple color buffers capable of receiving color values. *buffers* is an array of buffer enumerates. Only GL_NONE, GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, and GL_BACK_RIGHT are accepted.

When 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(GL_FRONT_AND_BACK)** to enable you to draw to both front and back buffers at the same time.

For selecting the read buffer, use **glReadBuffer()**.

void **glReadBuffer**(GLenum *mode*);

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

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

If *mode* is not one of the above tokens, a GL_INVALID_ENUM is generated.

As we've seen, when a framebuffer object has multiple attachments, you can control various aspects of what happens with the renderbuffer at an attachment, like controlling the scissors box, or blending. You use the commands **glEnablei()** and **glDisablei()** to control capabilities on a per-attachment granularity.

void **glEnablei**(GLenum *capability*, GLuint *index*);
void **glDisablei**(GLenum *capability*, GLuint *index*);

Enables or disables *capability* for buffer *index*.

A GL_INVALID_VALUE is generated if *index* is greater than or equal to GL_MAX_DRAW_BUFFERS.

GLboolean **glIsEnabledi**(GLenum *capability*, GLuint *index*);

Specifies whether *target* is enabled for buffer *index*.

A GL_INVALID_VALUE is generated if *index* is outside of the range supported for *target*.

Dual-Source Blending

Advanced

Two of the blend factors already described in this chapters are the *second source* blending factors and are special in that they are driven by a second output in the fragment shader. These factors, GL_SRC1_COLOR and GL_SRC1_ALPHA, are produced in the fragment shader by writing to an output whose index is 1 (rather than the default 0). To create such an output we use the `index` *layout qualifier* when declaring it in the fragment shader. Example 4.13 shows an example of such a declaration.

Example 4.13 Layout Qualifiers Specifying the Index of Fragment Shader Outputs

```
layout (location = 0, index = 0) out vec4 first_output;
layout (location = 0, index = 1) out vec4 second_output;
```

When calling **glBlendFunc()**, **glBlendFunci()**, **glBlendFuncSeparate()**, or **glBlendFuncSeparatei()**, the GL_SRC_COLOR, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_COLOR, or GL_ONE_MINUS_SRC_ALPHA factors will cause the blending equation's input to be taken from `first_input`. However, passing GL_SRC1_COLOR, GL_SRC1_ALPHA GL_ONE_MINUS_SRC1_COLOR, or GL_ONE_MINUS_SRC1_ALPHA to these functions will cause the input to be taken from `second_output`. This allows some interesting blending equations to be built up by using combinations of the first and second sources in each of the source and destination blend factors.

For example, setting the source factor to GL_SRC1_COLOR and the destination factor to GL_ONE_MINUS_SRC1_COLOR using one of the blending functions essentially allows a *per-channel alpha* to be created in the fragment shader. This type of functionality is especially useful when implementing subpixel accurate antialiasing techniques in the fragment shader. By taking the location of the red, green, and blue color elements in the pixels on the screen into account, coverage for each element can be generated in the fragment shader and be used to selectively light each color by a function of its coverage. Figure 4.4 shows a close-up picture of the red, green and blue picture elements in a liquid crystal computer monitor. The subpixels are clearly visible, although when viewed at normal distance, the display appears white. By lighting each of the red, green, and blue elements separately, very high-quality antialiasing can be implemented.

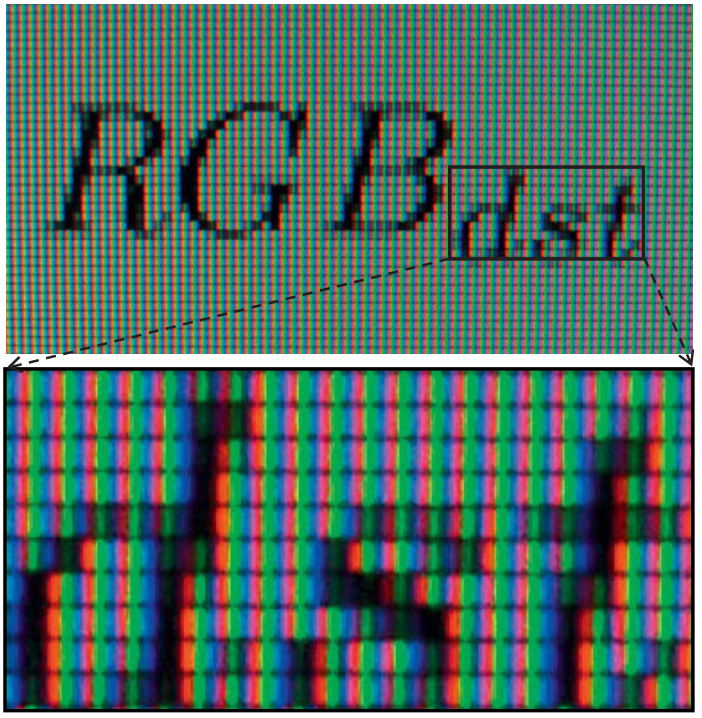

Figure 4.4 Close-up of RGB color elements in an LCD panel

Another possible use is to set the source and destination factors in the blending equation to GL_ONE and GL_SRC1_COLOR. In this configuration, the first color output is added to the framebuffer's content, while the second color output is used to attenuate the framebuffer's content. The equation becomes:

$$RGB_{dst} = RGB_{src0} + RGB_{src1} * RGB_{dst}$$

This is a classic multiply-add operation and can be used for many purposes. For example, if you want to render a translucent object with a colored specular highlight, write the color of the object to `second_output` and the highlight color to `first_output`.

Dual-Source Blending and Multiple Fragment Shader Outputs

Because the second output from the fragment shader that is required to implement dual source blending may take from the resources available to produce outputs for multiple framebuffer attachments (draw buffers), there are special counting rules for dual-source blending. When dual-source blending is enabled—that is, when any of the factors specified to one of the **glBlendFunc()** functions is one of the tokens that includes `SRC1`, the total number of outputs available in the fragment shader may be reduced. To determine how many outputs may be used (and consequently, how many framebuffer attachments may be active), query for the value of GL_MAX_DUAL_SOURCE_DRAW_BUFFERS. Note that the OpenGL specification only requires that GL_MAX_DUAL_SOURCE_DRAW_BUFFERS be at least one. If GL_MAX_DUAL_SOURCE_DRAW_BUFFERS is exactly one, this means that dual source blending and multiple draw buffers are mutually exclusive and cannot be used together.

Reading and Copying Pixel Data

Once your rendering is complete, you may want to retrieve the rendered image for posterity. In that case, you can use the **glReadPixels()** function to read pixels from the read framebuffer and return the pixels to your application. You can return the pixels into memory allocated by the application, or into a pixel pack buffer, if one's currently bound.

void **glReadPixels**(GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*, GLenum *format*, GLenum *type*, void **pixels*);

Reads pixel data from the read framebuffer rectangle whose lower-left corner is at (x, y) in window coordinates and whose dimensions are *width* and *height*, and then stores the data in the array pointed to by *pixels*. *format* indicates the kind of pixel data elements that are read (color, depth, or stencil value as listed in Table 4.9), and *type* indicates the data type of each element (see Table 4.10.)

glReadPixels() can generate a few OpenGL errors. A GL_INVALID_OPERATION error will be generated if format is set to GL_DEPTH and there is no depth buffer; or if format is GL_STENCIL and there is no stencil buffer; or if format is set to GL_DEPTH_STENCIL and there are not both a depth and a stencil buffer associated with the framebuffer, or if type is neither GL_UNSIGNED_INT_24_8 nor GL_FLOAT_32_UNSIGNED_INT_24_8_REV, then GL_INVALID_ENUM is set.

Table 4.9 **glReadPixels()** Data Formats

Format Value	**Pixel Format**
GL_RED or GL_RED_INTEGER	a single red color component
GL_GREEN or GL_GREEN_INTEGER	a single green color component
GL_BLUE or GL_BLUE_INTEGER	a single blue color component
GL_ALPHA or GL_ALPHA_INTEGER	a single alpha color component
GL_RG or GL_RG_INTEGER	a red color component, followed by a green component
GL_RGB or GL_RGB_INTEGER	a red color component, followed by green and blue components
GL_RGBA or GL_RGBA_INTEGER	a red color component, followed by green, blue, and alpha components
GL_BGR or GL_BGR_INTEGER	a blue color component, followed by green and red components
GL_BGRA or GL_BGRA_INTEGER	a blue color component, followed by green, red, and alpha components
GL_STENCIL_INDEX	a single stencil index
GL_DEPTH_COMPONENT	a single depth component
GL_DEPTH_STENCIL	combined depth and stencil components

You may need to specify which buffer you want to retrieve pixel values from. For example, in a double-buffered window, you could read the pixels from the front buffer or the back buffer. You can use the **glReadBuffer()** routine to specify which buffer to retrieve the pixels from.

Table 4.10 Data Types for **glReadPixels()**

Type Value	Data Type	Packed
GL_UNSIGNED_BYTE	GLubyte	No
GL_BYTE	GLbyte	No
GL_UNSIGNED_SHORT	GLushort	No
GL_SHORT	GLshort	No
GL_UNSIGNED_INT	GLuint	No
GL_INT	GLint	No
GL_HALF_FLOAT	GLhalf	
GL_FLOAT	GLfloat	No
GL_UNSIGNED_BYTE_3_3_2	GLubyte	Yes
GL_UNSIGNED_BYTE_2_3_3_REV	GLubyte	Yes
GL_UNSIGNED_SHORT_5_6_5	GLushort	Yes
GL_UNSIGNED_SHORT_5_6_5_REV	GLushort	Yes
GL_UNSIGNED_SHORT_4_4_4_4	GLushort	Yes
GL_UNSIGNED_SHORT_4_4_4_4_REV	GLushort	Yes
GL_UNSIGNED_SHORT_5_5_5_1	GLushort	Yes
GL_UNSIGNED_SHORT_1_5_5_5_REV	GLushort	Yes
GL_UNSIGNED_INT_8_8_8_8	GLuint	Yes
GL_UNSIGNED_INT_8_8_8_8_REV	GLuint	Yes
GL_UNSIGNED_INT_10_10_10_2	GLuint	Yes
GL_UNSIGNED_INT_2_10_10_10_REV	GLuint	Yes
GL_UNSIGNED_INT_24_8	GLuint	Yes
GL_UNSIGNED_INT_10F_11F_11F_REV	GLuint	Yes
GL_UNSIGNED_INT_5_9_9_9_REV	GLuint	Yes
GL_FLOAT_32_UNSIGNED_INT_24_8_REV	GLfloat	Yes

Clamping Returned Values

Various types of buffers within OpenGL—most notably floating-point buffers—can store values with ranges outside of the normal [0, 1] range of colors in OpenGL. When you read those values back using **glReadPixels()**,

you can control whether the values should be clamped to the normalized range or left at their full range using **glClampColor()**.

void **glClampColor**(GLenum *target*, GLenum *clamp*);

Controls the clamping of color values for floating- and fixed-point buffers, when *target* is GL_CLAMP_READ_COLOR. If *clamp* is set to GL_TRUE, color values read from buffers are clamped to the range [0, 1]; conversely, if *clamp* is GL_FALSE, no clamping is engaged. If your application uses a combination of fixed- and floating-point buffers, set *clamp* to GL_FIXED_ONLY to clamp only the fixed-point values; floating-point values are returned with their full range.

Copying Pixel Rectangles

To copy pixels between regions of a buffer, or even different framebuffers, use **glBlitFramebuffer()**. It uses greater pixel *filtering* during the copy operation, much in the same manner as texture mapping (in fact, the same filtering operations, GL_NEAREST and GL_LINEAR are used during the copy). Additionally, this routine is aware of multisampled buffers and supports copying between different framebuffers (as controlled by framebuffer objects).

void **glBlitFramebuffer**(GLint *srcX0*, GLint *srcY0*, GLint *srcX1*,
GLint *srcY1*, GLint *dstX0*, GLint *dstY0*,
GLint *dstX1*, GLint *dstY1*,
GLbitfield *buffers*, GLenum *filter*);

Copies a rectangle of pixel values from one region of the read framebuffer to another region of the draw framebuffer, potentially resizing, reversing, converting, and filtering the pixels in the process. *srcX0*, *srcY0*, *srcX1*, *srcY1* represent the source region where pixels are sourced from, and written to the rectangular region specified by *dstX0*, *dstY0*, *dstX1*, and *dstY1*. *buffers* is the bitwise-or of GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, and GL_STENCIL_BUFFER_BIT, which represent the buffers in which the copy should occur. Finally, *filter* specifies the method of interpolation done if the two rectangular regions are of different sizes, and must be one of GL_NEAREST or GL_LINEAR; no filtering is applied if the regions are of the same size.

If there are multiple-color draw buffers, each buffer receives a copy of the source region.

If *srcX1* < *srcX0*, or *dstX1* < *dstX0*, the image is reversed in the horizontal direction. Likewise, if *srcY1* < *srcY0* or *dstY1* < *dstY0*, the image is reversed in the vertical direction. However, If both the source and destination sizes are negative in the same direction, no reversal is done.

If the source and destination buffers are of different formats, conversion of the pixel values is done in most situations. However, if the read color buffer is a floating-point format, and any of the write color buffers are not, or vice versa; and if the read-color buffer is a signed (unsigned) integer format and not all of the draw buffers are signed (unsigned) integer values, the call will generate a GL_INVALID_OPERATION, and no pixels will be copied.

Multisampled buffers also have an effect on the copying of pixels. If the source buffer is multisampled, and the destination is not, the samples are resolved to a single pixel value for the destination buffer. Conversely, if the source buffer is not multisampled, and the destination is, the source pixel's data is replicated for each sample. Finally, if both buffers are multisampled and the number of samples for each buffer is the same, the samples are copied without modification. However, if the buffers have a different number of samples, no pixels are copied, and a GL_INVALID_OPERATION error is generated.

A GL_INVALID_VALUE error is generated if buffers have other bits set than those permitted, or if *filter* is other than GL_LINEAR or GL_NEAREST.