6.2 Fragment Operations

A number of fragment operations are applied to rasterization fragments before they are allowed to update pixels in the framebuffer. Fragment operations can be separated into two categories, operations that test fragments, and operations that modify them. To maximize efficiency, the fragment operations are ordered so that the fragment tests are applied first. The most interesting tests for advanced rendering are: alpha test, stencil test, and depth buffer test. These tests can either pass, allowing the fragment to continue, or fail, discarding the fragment so it can't pass on to later fragment operations or update the framebuffer. The stencil test is a special case, since it can produce useful side effects even when fragments fail the comparison.

All of the fragment tests use the same set of comparison operators: Never, Always, Less, Less than or Equal, Equal, Greater than or Equal, Greater, and Not Equal. In each test, a fragment value is compared against a reference value saved in the current OpenGL state (including the depth and stencil buffers), and if the comparison succeeds, the test passes. The details of the fragment tests are listed in Table 6.1.

The list of comparison operators is very complete. In fact, it may seem that some of the comparison operations, such as `GL_NEVER` and `GL_ALWAYS` are redundant, since their functionality can be duplicated by enabling or disabling a given fragment test. There is a use for them, however. The OpenGL invariance rules require that invariance is maintained if a comparison is changed, but not if a test is enabled or disabled. So if invariance must be maintained (because the test is used in a multipass algorithm, for example), the application should enable and disable tests using the comparison operators, rather than enabling or disabling the tests themselves.

Table 6.1 Fragment Test

Constant	Comparison
GL_ALWAYS	always pass
GL_NEVER	never pass
GL_LESS	pass if $incoming < ref$
GL_LEQUAL	pass if $incoming \leq ref$
GL_GEQUAL	pass if $incoming \geq ref$
GL_GREATER	pass if $incoming > ref$
GL_EQUAL	pass if $incoming = ref$
GL_NOTEQUAL	pass if $incoming \neq ref$

6.2.1 Multisample Operations

Multisample operations provide limited ways to affect the fragment coverage and alpha values. In particular, an application can reduce the coverage of a fragment, or convert the fragment's alpha value to another coverage value that is combined with the fragment's value to further reduce it. These operations are sometimes useful as an alternative to alpha blending, since they can be more efficient.

6.2.2 Alpha Test

The alpha test reads the alpha component value of each fragment's color, and compares it against the current alpha test value. The test value is set by the application, and can range from zero to one. The comparison operators are the standard set listed in Table 6.1. The alpha test can be used to remove parts of a primitive on a pixel by pixel basis. A common technique is to apply a texture containing alpha values to a polygon. The alpha test is used to trim a simple polygon to a complex outline stored in the alpha values of the surface texture. A detailed description of this technique is available in Section 14.5.

6.2.3 Stencil Test

The stencil test performs two tasks. The first task is to conditionally eliminate incoming fragments based on a comparison between a reference value and stencil value from the stencil buffer at the fragment's destination. The second purpose of the stencil test is to update the stencil values in the framebuffer. How the stencil buffer is modified depends on the outcome of the stencil and depth buffer tests. There are three possible outcomes of the two tests: the stencil buffer test fails, the stencil buffer test passes but the depth buffer fails, or both tests fail. OpenGL makes it possible to specify how the stencil buffer is updated for each of these possible outcomes.

The conditional elimination task is controlled with `glStencilFunc`. It sets the stencil test comparison operator. The comparison operator can be selected from the list of operators in Table 6.1.

Setting the stencil update requires setting three parameters, each one corresponding to one of the stencil/depth buffer test outcomes. The `glStencilOp` command takes three operands, one for each of the comparison outcomes (see Figure 6.4). Each operand value specifies how the stencil pixel corresponding to the fragment being tested should be modified. Table 6.2 shows the possible values and how they change the stencil pixels.

The stencil buffer is often used to create and use per-pixel masks. The desired stencil mask is created by drawing geometry (often textured with an alpha pattern to produce a complex shape). Before rendering this template geometry, the stencil test is configured to update the stencil buffer as the mask is rendered. Often the pipeline is configured so that the color and depth buffers are not actually updated when this geometry is rendered; this can be done with the `glColorMask` and `glDepthMask` commands, or by setting the depth test to always fail.

Table 6.2 Stencil Update Values

Constant	Description
GL_KEEP	*stencil pixel* ← *old value*
GL_ZERO	*stencil pixel* ← 0
GL_REPLACE	*stencil pixel* ← *reference value*
GL_INCR	*stencil pixel* ← *old value* + 1
GL_DECR	*stencil pixel* ← *old value* − 1
GL_INVERT	*stencil pixel* ← *old value*

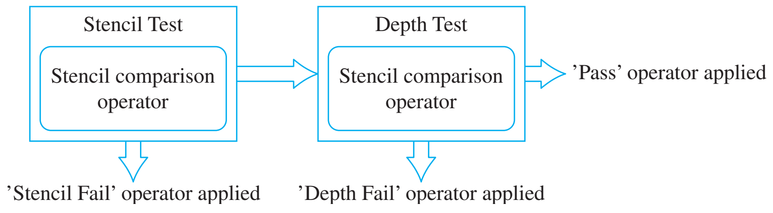
Stencil Test
Stencil comparison operator
Depth Test
Stencil comparison operator
'Pass' operator applied
'Stencil Fail' operator applied
'Depth Fail' operator applied

Figure 6.4 Stencil/depth test functionality.

Once the stencil mask is in place, the geometry to be masked is rendered. This time, the stencil test is pre-configured to draw or discard fragments based on the current value of the stencil mask. More elaborate techniques may create the mask using a combination of template geometry and carefully chosen depth and stencil comparisons to create a mask whose shape is influenced by the geometry that was previously rendered. There are are also some extensions for enhancing stencil functionality. One allows separate stencil operations, reference value, compare mask, and write mask to be selected depending on whether the polygon is front- or back-facing.[1] A second allows the stencil arithmetic operations to wrap rather than clamp, allowing a stencil value to temporarily go out of range, while still producing the correct answer if the final answer lies within the representable range.[2]

6.2.4 Blending

One of the most useful fragment modifier operations supported by OpenGL is blending, also called *alpha blending*. Without blending, a fragment that passes all the filtering

1. EXT_stencil_two_side
2. EXT_stencil_wrap

Table 6.3 Blend Factors

Constant	Used In	Action
GL_ZERO	src, dst	scale each color element by zero
GL_ONE	src, dst	scale each element by one
GL_SRC_COLOR	dst	scale color with source color
GL_DST_COLOR	src	scale color with destination color
GL_ONE_MINUS_SRC_COLOR	dst	scale color with one minus source color
GL_ONE_MINUS_DST_COLOR	dst	scale color with one minus destination color
GL_SRC_ALPHA	src, dst	scale color with source alpha
GL_ONE_MINUS_SRC_ALPHA	src, dst	scale color with source alpha
GL_DST_ALPHA	src, dst	scale color with destination alpha
GL_ONE_MINUS_DST_ALPHA	src, dst	scale color with one minus destination alpha
GL_SRC_ALPHA_SATURATE	src	scale color by minimum of source alpha and destination alpha
GL_CONSTANT_COLOR	src, dst	scale color with application-specified color
GL_ONE_MINUS_CONSTANT_COLOR	src, dst	scale color with one minus application-specified color
GL_CONSTANT_ALPHA	src, dst	scale color with alpha of application-specified color
GL_ONE_MINUS_CONSTANT_ALPHA	src, dst	scale color with one minus alpha of application-specified color

and modification steps simply replaces the appropriate color pixel in the framebuffer. If blending is enabled, the incoming fragment, the corresponding target pixel, or an application-defined constant color[3] are combined using a linear equation instead. The color components (including alpha) of both the fragment and the pixel are first scaled by a specified *blend factor*, then either added or subtracted.[4] The resulting value is used to update the framebuffer.

There are two fixed sets of blend factors (also called *weighting factors*) for the blend operation; one set is for the source argument, one for the destination. The entire set is listed in Table 6.3; the second column indicates whether the factor can be used

3. In implementations supporting OpenGL 1.2 or newer.

4. In OpenGL 1.4 subtraction and min and max blend equations were moved from the ARB_imaging extension to the core.

Table 6.4 Blend Equations

Operand	Result
GL_ADD	*soruce* + *destination*
GL_SUBTRACT	*source* − *destination*
GL_REVERSE_SUBTRACT	*destination* − *source*
GL_MIN	min(*source*, *dest*)
GL_MAX	max(*source*, *dest*)

with a source, a destination, or both. These factors take a color from one of the three inputs, the incoming fragment, the framebuffer pixel, or the application-specified color, modify it, and insert it into the blending equation. The source and destination arguments are used by the blend equation, one of GL_FUNC_ADD, GL_FUNC_SUBTRACT, GL_FUNC_REVERSE_SUBTRACT, GL_MIN, or GL_MAX. Table 6.4 lists the operations. Note that the result of the subtract equation depends on the order of the arguments, so both subtract and reverse subtract are provided. In either case, negative results are clamped to zero.

Some blend factors are used more frequently than others: GL_ONE is commonly used when an unmodified source or destination color is needed in the equation. Using GL_ONE for both factors, for example, simply adds (or subtracts) the source pixel and destination pixel value. The GL_ZERO factor is used to eliminate one of the two colors. The GL_SRC_ALPHA/GL_ONE_MINUS_ALPHA combination is used for a common transparency technique, where the alpha value of the fragment determines the opacity of the fragment. Another transparency technique uses GL_SRC_ALPHA_SATURATE instead; it is particularly useful for wireframe line drawings, since it minimizes line brightening where multiple transparent lines overlap.

6.2.5 Logic Op

As of OpenGL 1.1, a new fragment processing stage, *logical operation*,[5] can be used instead of blending (if both stages are enabled by the application, logic op takes precedence, and blending is implicitly disabled). Logical operations are defined for both index and RGBA colors; only the latter is discussed here. As with blending, logic op takes the incoming fragment and corresponding pixel color, and performs an operation on it. This *bitwise* operation is chosen from a fixed set by the application using the glLogicOp command. The possible logic operations are shown in Table 6.5 (C-style logical operands are used for clarity).

5. In OpenGL 1.0, the logic op stage operated in color index mode only.