1 Introduction

In the previous chapter the basics of using a shader to render simple primitives were introduced. This chapter introduces basic usage of the fragment shader. In particular, this chapter shows a few things that can be done in fragment shaders using one the most basic built-in inputs and functions. Later chapters will build on these examples.

1.1 Drawing Circles Using \textit{gl\textunderscore FragCoord}

It would seem that a fragment shader such as Figure 2 would have no inputs when paired with a vertex shader such as Figure 1. However, every fragment shader has at least one built-in input: the current fragment position. The variable \textit{gl\textunderscore FragCoord} contains the window position of the current fragment in its X and Y components. The Z component contains fragment’s depth value on the range [0, 1].

The fragment shader in Figure 3 uses \textit{gl\textunderscore FragCoord} to render a simple pattern. The shader partitions the window into 50-by-50 pixel squares. Line 3 in the shader computes the position of the current fragment relative to the center of the 50-by-50 pixel region that contains it. Line 4 computes the squared distance from the center. Lines 6 through 8 use that distance to write either gray or blue to \textit{gl\textunderscore FragColor}. The leftmost image of Figure 4 is a sample image rendered by this shader. The result is a series of 20 pixel diameter gray circles.
The shader in Figure 3 can be rewritten to utilize two useful functions that are built into GLSL. The `step` function returns 0.0 for each component of `value` that is less than `edge`.

```
void step(genType edge, genType value);
```

Like many functions in GLSL, `step` has parameters and return values with the type `genType`. This is shorthand for `float`, `vec2`, `vec3`, and `vec4`. The full set of overloaded versions of `step` is:

```
void step(float edge, float value);
vec2 step(vec2 edge, vec2 value);
vec3 step(vec3 edge, vec3 value);
vec4 step(vec4 edge, vec4 value);
vec2 step(float edge, vec2 value);
vec3 step(float edge, vec3 value);
vec4 step(float edge, vec4 value);
```

The `mix` function calculates the component-wise linear interpolation of `v0` and `v1` using `t` as the blend factor. More precisely, `mix` computes $v_0 \times (1.0-t) + v_1 \times t$. Note that `t` need not be limited to the range [0, 1].

```
void mix(genType v0, genType v1, genType t);
```

Using `step` and `mix` the shader in Figure 3 can be rewritten as the shader in Figure 5. Both shaders should produce identical output.

```
1 int main(void)
2 {
3   vec2 pos = mod(gl_FragCoord.xy, vec2(50.0)) - vec2(25.0);
4   float dist_squared = dot(pos, pos);
5     gl_FragColor = mix(vec4(.90, .90, .90, 1.0), vec4(.20, .20, .40, 1.0),
6                        step(400.0, dist_squared));
7 }
```

A close examination of the leftmost Figure 4 reveals an unpleasant sharp boundary at the edge of the circle. In this implementation, each pixel is either gray or blue. The exact edge of a 20 pixel radius circle actually passes through a different portion of each pixel along the edge. As a result, pixels along the edge should be drawn with some combination of blue and gray. This can be accomplished by adjusting the third parameter to the `mix` function.

If the distance from the center is below some threshold, say 19.5, the pixel should be completely gray. If the distance from the center is above some other threshold, say 20.5, the pixel should be completely blue. If the distance between these two thresholds, the pixel should be some combination of blue and gray. This new blend factor could be implemented as in Figure 6. Using `linearstep` in place of `step` in Figure 5 results in the middle image in Figure 4.
Notice that the edges have gone from too sharp to too blurry. Instead of a simple linear interpolation, an interpolator that changes slowly near the edges and quickly through the middle will produce better results. The Hermite interpolator, \( f(t) = 3t^2 - 2t^3 \), does this exactly. There is a function built into GLSL, `smoothstep`, which computes this value over a given range. `linearstep` can be directly replaced with `smoothstep` to produce the rightmost image in Figure 4. The final shader appears in Figure 7.

```cpp
void main(void)
{
vec2 pos = mod(rotation * gl_FragCoord.xy, vec2(50.0)) - vec2(25.0);
float dist_squared = dot(pos, pos);

gl_FragColor = mix(vec4(.90, .90, .90, 1.0), vec4(.20, .20, .40, 1.0),
smoothstep(380.25, 420.25, dist_squared));
}
```

There is one big disadvantage of the shader in Figure 8. If the application wants to change the rotation angle, it has to change the shader source. A later chapter will show a method for passing parameters into shaders that are constant across a group of primitives.
1.3 Cutting Holes

Occasionally it is useful to have "holes" in the rendered image. A special fragment shader keyword `discard` can do this. In lines 6 and 7 of Figure 10 compare the squared radius to the range [100, 575]. Any fragments outside that range are discarded. In Figure 11 the black pixels are the background color. This shows through where fragments are discarded.

```cpp
1 void main(void)  
2 {  
3   vec2 pos = mod(gl_FragCoord.xy, vec2(50.0)) - vec2(25.0);  
4   float dist_squared = dot(pos, pos);  
5   if ((dist_squared > 575.0) || (dist_squared < 100.0))  
6     discard;  
7   gl_FragColor = mix(vec4(.90, .90, .90, 1.0), vec4(.20, .20, .40, 1.0),  
8     smoothstep(380.25, 420.25, dist_squared));  
9 }  
```

It is tempting the think of `discard` as analogous to calling `exit` in a C program. Consider the code in Figure 12. Lines 2 and 3 discard the fragment if the variable `x` is less than some small value. Lines 5 through 7 loop using `x` as a loop control. If `x` is very small or zero, the number of loop iterations is very large or infinite.

Lines 2 and 3 are designed to terminate the shader to prevent this from occuring. Hardware typically runs fragment shaders on small blocks of fragments, usually 2x2, together. If one fragment in a group does not execute the `discard`, all of the fragments in the may continue executing. The shader may enter an infinite loop even though the programmer has attempted to prevent it. The solution is to surround the remainder of the shader in the `else` portion of the `if`-statement.

---

1 Earlier shading languages called this "killing" a fragment. Other texts may refer to discarding fragments as killing.
Figure 12 Shader with a possible infinite loop

```c
/* If x is less than 1/32nd, exit. */
if (x < 0.03125)
    discard;

for (int i = 0; i < int(1.0 / x); i++) {
    /* ... do some work ... */
}
```