Extra buffers

Up until now there is only one type of output buffer you've made use of, the color buffer. This chapter will discuss two additional types, the depth buffer and the stencil buffer. For each of these a problem will be presented and subsequently solved with that specific buffer.

Preparations

To best demonstrate the use of these buffers, let's draw a cube instead of a flat shape. The vertex shader needs to be modified to accept a third coordinate:

```
in vec3 position;
...
gl_Position = proj * view * model * vec4(position, 1.0);
```

We're also going to need to alter the color again later in this chapter, so make sure the fragment shader multiplies the texture color by the color attribute:

```
vec4 texColor = mix(texture(texKitten, Texcoord),
                    texture(texPuppy, Texcoord), 0.5);
outColor = vec4(Color, 1.0) * texColor;
```

Vertices are now 8 floats in size, so you'll have to update the vertex attribute offsets and strides as well. Finally, add the extra coordinate to the vertex array:

```
float vertices[] = {
  // X      Y     Z     R     G     B     U     V
    -0.5f,  0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
     0.5f,  0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f,
     0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f,
    -0.5f, -0.5f, 0.0f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f
};
```

Confirm that you've made all the required changes by running your program and checking if it still draws a flat spinning image of a kitten blended with a puppy. A single cube consists of 36 vertices (6 sides * 2 triangles * 3 vertices), so I will ease your life by providing the array here.

```
glDrawArrays(GL_TRIANGLES, 0, 36);
```

We will not make use of element buffers for drawing this cube, so you can use `glDrawArrays` to draw it. If you were confused by this explanation, you can compare your program to this reference code.

Introduction
Context creation
Drawing polygons
Textures
Transformations
Depth and stencils
Framebuffers
Geometry shaders
Transform Feedback

Links

OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

It immediately becomes clear that the cube is not rendered as expected when seeing the output. The sides of the cube are being drawn, but they overlap each other in strange ways! The problem here is that when OpenGL draws your cube triangle-by-triangle, it will simply write over pixels even though something else may have been drawn there before. In this case OpenGL will happily draw triangles in the back over triangles at the front.

Luckily OpenGL offers ways of telling it when to draw over a pixel and when not to. I'll go over the two most important ways of doing that, depth testing and stencilling, in this chapter.

Depth buffer

Z-buffering is a way of keeping track of the depth of every pixel on the screen. The depth is proportional to the distance between the screen plane and a fragment that has been drawn. That means that the fragments on the sides of the cube further away from the viewer have a higher depth value, whereas fragments closer have a lower depth value.

If this depth is stored along with the color when a fragment is written, fragments drawn later can compare their depth to the existing depth to determine if the new fragment is closer to the viewer than the old fragment. If that is the case, it should be drawn over and otherwise it can simply be discarded. This is known as depth testing.

OpenGL offers a way to store these depth values in an extra buffer, called the depth buffer, and perform the required check for fragments automatically. The fragment shader will not run for fragments that are invisible, which can have a significant impact on performance. This functionality can be enabled by calling `glEnable`.

```
glEnable(GL_DEPTH_TEST);
```

If you enable this functionality now and run your application, you'll notice that you get a black screen. That happens because the depth buffer is filled with 0 depth for each pixel by default. Since no fragments will ever be closer than that they are all discarded.

The depth buffer can be cleared along with the color buffer by extending the `glClear` call:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

The default clear value for the depth is `1.0f`, which is equal to the depth of your far clipping plane and thus the furthest depth that can be represented. All fragments will be closer than that, so they will no longer be discarded.

With the depth test capability enabled, the cube is now rendered correctly. Just like the color buffer, the depth buffer has a certain amount of bits of precision which can be specified by you. Less bits of precision reduce the extra memory use, but can introduce rendering errors in more complex scenes.

Stencil buffer

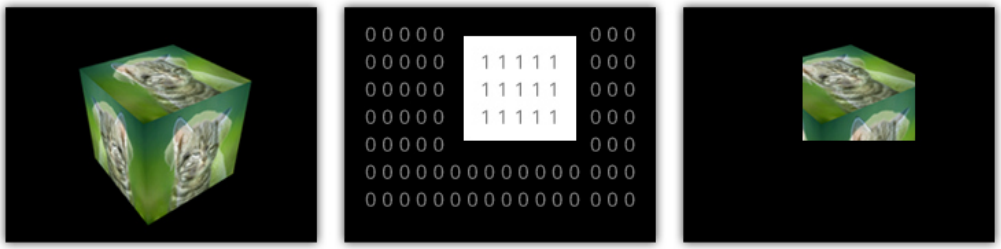
Introduction
Context creation
Drawing polygons
Textures
Transformations
Depth and stencils
Framebuffers
Geometry shaders
Transform Feedback

Links

OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

The stencil buffer is an optional extension of the depth buffer that gives you more control over the question of which fragments should be drawn and which shouldn't. Like the depth buffer, a value is stored for every pixel, but this time you get to control when and how this value changes and when a fragment should be drawn depending on this value. Note that if the depth test fails, the stencil test no longer determines whether a fragment is drawn or not, but these fragments can still affect values in the stencil buffer!

To get a bit more acquainted with the stencil buffer before using it, let's start by analyzing a simple example.



Color buffer without stencil test | Stencil buffer | Color buffer with stencil test

In this case the stencil buffer was first cleared with zeroes and then a rectangle of ones was drawn to it. The drawing operation of the cube uses the values from the stencil buffer to only draw fragments with a stencil value of 1.

Now that you have an understanding of what the stencil buffer does, we'll look at the relevant OpenGL calls.

```
glEnable(GL_STENCIL_TEST);
```

Stencil testing is enabled with a call to `glEnable`, just like depth testing. You don't have to add this call to your code just yet. I'll first go over the API details in the next two sections and then we'll make a cool demo.

Setting values

Regular drawing operations are used to determine which values in the stencil buffer are affected by any stencil operation. If you want to affect a rectangle of values like in the sample above, simply draw a 2D quad in that area. What happens to those values can be controlled by you using the `glStencilFunc`, `glStencilOp` and `glStencilMask` functions.

The `glStencilFunc` call is used to specify the conditions under which a fragment passes the stencil test. Its parameters are discussed below.

- `func`: The test function, can be `GL_NEVER`, `GL_LESS`, `GL_LEQUAL`, `GL_GREATER`, `GL_GEQUAL`, `GL_EQUAL`, `GL_NOTEQUAL`, and `GL_ALWAYS`.
- `ref`: A value to compare the stencil value to using the test function.
- `mask`: A bitwise AND operation is performed on the stencil value and reference value with this mask value before comparing them.

If you don't want stencils with a value lower than 2 to be affected, you would use:

```
glStencilFunc(GL_GEQUAL, 2, 0xFF);
```

The mask value is set to all ones (in case of an 8 bit stencil buffer), so it will not affect the test.

The `glStencilOp` call specifies what should happen to stencil values depending on the outcome of the stencil and depth tests. The parameters are:

- `sfail`: Action to take if the stencil test fails.
- `dpfail`: Action to take if the stencil test is successful, but the depth test failed.
- `dppass`: Action to take if both the stencil test and depth tests pass.

Stencil values can be modified in the following ways:

- `GL_KEEP`: The current value is kept.
- `GL_ZERO`: The stencil value is set to 0.
- `GL_REPLACE`: The stencil value is set to the reference value in the `glStencilFunc` call.
- `GL_INCR`: The stencil value is increased by 1 if it is lower than the maximum value.
- `GL_INCR_WRAP`: Same as `GL_INCR`, with the exception that the value is set to 0 if the maximum value is exceeded.
- `GL_DECR`: The stencil value is decreased by 1 if it is higher than 0.
- `GL_DECR_WRAP`: Same as `GL_DECR`, with the exception that the value is set to the maximum value if the current value is 0 (the stencil buffer stores unsigned integers).
- `GL_INVERT`: A bitwise invert is applied to the value.

Finally, `glStencilMask` can be used to control the bits that are written to the stencil buffer when an operation is run. The default value is all ones, which means that the outcome of any operation is unaffected.

Introduction
Context creation
Drawing polygons
Textures
Transformations
Depth and stencils
Framebuffers
Geometry shaders
Transform Feedback

Links
OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

If, like in the example, you want to set all stencil values in a rectangular area to 1, you would use the following calls:

```
glStencilFunc(GL_ALWAYS, 1, 0xFF);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
glStencilMask(0xFF);
```

In this case the rectangle shouldn't actually be drawn to the color buffer, since it is only used to determine which stencil values should be affected.

```
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);
```

The `glColorMask` function allows you to specify which data is written to the color buffer during a drawing operation. In this case you would want to disable all color channels (red, green, blue, alpha). Writing to the depth buffer needs to be disabled separately as well with `glDepthMask`, so that cube drawing operation won't be affected by leftover depth values of the rectangle. This is cleaner than simply clearing the depth buffer again later.

Using values in drawing operations

With the knowledge about setting values, using them for testing fragments in drawing operations becomes very simple. All you need to do now is re-enable color and depth writing if you had disabled those earlier and setting the test function to determine which fragments are drawn based on the values in the stencil buffer.

```
glStencilFunc(GL_EQUAL, 1, 0xFF);
```

If you use this call to set the test function, the stencil test will only pass for pixels with a stencil value equal to 1. A fragment will only be drawn if it passes both the stencil and depth test, so setting the `glStencilOp` is not necessary. In the case of the example above only the stencil values in the rectangular area were set to 1, so only the cube fragments in that area will be drawn.

```
glStencilMask(0x00);
```

One small detail that is easy to overlook is that the cube draw call could still affect values in the stencil buffer. This problem can be solved by setting the stencil bit mask to all zeroes, which effectively disables stencil writing.

Planar reflections

Let's spice up the demo we have right now a bit by adding a floor with a reflection under the cube. I'll add the vertices for the floor to the same vertex buffer the cube is currently using to keep things simple:

```
float vertices[] = {
    ...

    -1.0f, -1.0f, -0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
     1.0f, -1.0f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f,
     1.0f,  1.0f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f,
     1.0f,  1.0f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f,
    -1.0f,  1.0f, -0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f,
    -1.0f, -1.0f, -0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f
}
```

Now add the extra draw call to your main loop:

```
glDrawArrays(GL_TRIANGLES, 36, 6);
```

To create the reflection of the cube itself, it is sufficient to draw it again but inverted on the Z-axis:

```
model = glm::scale(
    glm::translate(model, glm::vec3(0, 0, -1)),
    glm::vec3(1, 1, -1)
);
glUniformMatrix4fv(uniModel, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);
```

I've set the color of the floor vertices to black so that the floor does not display the texture image, so you'll want to change the clear color to white to be able to see it. I've also changed the camera parameters a bit to get a good view of the scene.

Introduction
Context creation
Drawing polygons
Textures
Transformations
Depth and stencils
Framebuffers
Geometry shaders
Transform Feedback

Links

OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

Two issues are noticeable in the rendered image:

- The floor occludes the reflection because of depth testing.
- The reflection is visible outside of the floor.

The first problem is easy to solve by temporarily disabling writing to the depth buffer when drawing the floor:

```
glDepthMask(GL_FALSE);
glDrawArrays(GL_TRIANGLES, 36, 6);
glDepthMask(GL_TRUE);
```

To fix the second problem, it is necessary to discard fragments that fall outside of the floor. Sounds like it's time to see what stencil testing is really worth!

It can be greatly beneficial at times like these to make a little list of the rendering stages of the scene to get a proper idea of what is going on.

- Draw regular cube.
- Enable stencil testing and set test function and operations to write ones to all selected stencils.
- Draw floor.
- Set stencil function to pass if stencil value equals 1.
- Draw inverted cube.
- Disable stencil testing.

The new drawing code looks like this:

```
glEnable(GL_STENCIL_TEST);

    // Draw floor
    glStencilFunc(GL_ALWAYS, 1, 0xFF); // Set any stencil to 1
    glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
    glStencilMask(0xFF); // Write to stencil buffer
    glDepthMask(GL_FALSE); // Don't write to depth buffer
    glClear(GL_STENCIL_BUFFER_BIT); // Clear stencil buffer (0 by default)

    glDrawArrays(GL_TRIANGLES, 36, 6);

    // Draw cube reflection
    glStencilFunc(GL_EQUAL, 1, 0xFF); // Pass test if stencil value is 1
    glStencilMask(0x00); // Don't write anything to stencil buffer
    glDepthMask(GL_TRUE); // Write to depth buffer

    model = glm::scale(
        glm::translate(model, glm::vec3(0, 0, -1)),
        glm::vec3(1, 1, -1)
    );
    glUniformMatrix4fv(uniModel, 1, GL_FALSE, glm::value_ptr(model));
    glDrawArrays(GL_TRIANGLES, 0, 36);

glDisable(GL_STENCIL_TEST);
```

I've annotated the code above with comments, but the steps should be mostly clear from the stencil buffer section.

Introduction
Context creation
Drawing polygons
Textures
Transformations
Depth and stencils
Framebuffers
Geometry shaders
Transform Feedback

Links

OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

Now just one final touch is required, to darken the reflected cube a little to make the floor look a little less like a perfect mirror. I've chosen to create a uniform for this called `overrideColor` in the vertex shader:

```
uniform vec3 overrideColor;
...
Color = overrideColor * color;
```

And in the drawing code for the reflected cube

```
glUniform3f(uniColor, 0.3f, 0.3f, 0.3f);
    glDrawArrays(GL_TRIANGLES, 0, 36);
glUniform3f(uniColor, 1.0f, 1.0f, 1.0f);
```

where `uniColor` is the return value of a `glGetUniformLocation` call.

Awesome! I hope that, especially in chapters like these, you get the idea that working with an API as low-level as OpenGL can be a lot of fun and pose interesting challenges! As usual, the final code is available here.

Exercises

There are no real exercises for this chapter, but there are a lot more interesting effects you can create with the stencil buffer. I'll leave researching the implementation of other effects, such as stencil shadows and object outlining as an exercise to you.

Introduction
Context creation
Drawing polygons
Textures
Transformations
Depth and stencils
Framebuffers
Geometry shaders
Transform Feedback

Fork me on GitHub

Links

OpenGL boilerplate code
Easy-to-build code
Matrix math tutorials
OpenGL reference

57 Comments Open.GL Login

Recommend 17 Share Sort by Best

Join the discussion…

AnonymousPerson • 4 years ago
Awesome work here! Thank you.
9 • Reply • Share ›

silverhammermba • 4 years ago
The only part I found confusing here was when you started on the "Stencil buffer" section. Just following the example in my mind was very confusing and I spent a while trying (and failing) to figure out what all of the different function calls were doing. It was much clearer once I got down to the "Planar reflections" section and you laid out the list showing the order in which everything needs to be rendered. I think you should move that list much earlier in the guide, since the rendering order for the simple rectangle stencil example is very similar.
10 • Reply • Share ›

kt • 3 years ago
had some problems but stencil worked with this:

SDL_Init(SDL_INIT_VIDEO | SDL_INIT_TIMER);

SDL_GL_SetAttribute(SDL_GL_STENCIL_SIZE, 8);

window = SDL_CreateWindow("OpenGL", 100, 100, 800, 600, SDL_WINDOW_OPENGL);
context = SDL_GL_CreateContext(window);
5 • Reply • Share ›

Jacob Peterson • 3 years ago
Will you be continuing these tutorials?
5 • Reply • Share ›

Alexander Overvoorde Mod ➚ Jacob Peterson • 3 years ago
I currently have no plans to write new chapters, because I have to focus on other projects.
• Reply • Share ›

1Ra • 3 years ago
Just a warning: if you are using SDL and you encounter an error with the stencil buffer, you may need to call

SDL_GL_SetAttribute(SDL_GL_STENCILSIZE,8);

in order to create an 8-bit stencil buffer.
3 • Reply • Share ›

Mateusz Łukomski ➚ 1Ra • a year ago
thank you very much, that was my problem exactly!
• Reply • Share ›

pj ➚ 1Ra • 3 years ago
This was my problem, thanks!
• Reply • Share ›

Matthew Hoggan • 3 years ago
I am noticing in the tutorials that the following lines of code:

```
// Set up projection
glm::mat4 view = glm::lookAt(
glm::vec3( 2.5f, 2.5f, 2.0f ),
glm::vec3( 0.0f, 0.0f, 0.0f ),
glm::vec3( 0.0f, 0.0f, 1.0f )
);
GLint uniView = glGetUniformLocation( shaderProgram, "view" );
glUniformMatrix4fv( uniView, 1, GL_FALSE, glm::value_ptr( view ) );
```

When changed to:

```
// Set up projection
glm::mat4 view = glm::lookAt(
glm::vec3( 0.0f, 0.0f, 2.0f )
```