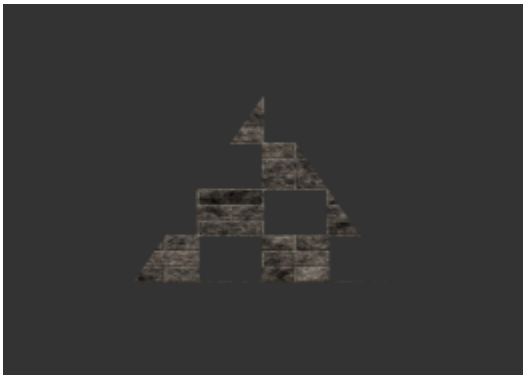
Tutorial 5: Scissors and Stencils



Summary

There may be occasions when we want to only render to parts of the screen, masking off the rest of the current colour buffer from any changes. In OpenGL there's two ways of doing so, *Scissor Regions*, and the *Stencil Buffer*. This tutorial will explain how to use both of them in your OpenGL applications.

New Concepts

Scissor Regions, Scissor Testing, Stencil Buffer, Stencil Testing

Scissor Regions

Of the two methods used to mask of sections of the back buffer when rendering, the simpler of the two is the scissor region. A scissor region is a rectangular section of the buffer, within which rendering takes place normally - any attempt to render outside of a scissor region is ignored - *discarded*. A scissor region locks *all* of the buffers against writing - so it is impossible to write to the colour and depth buffers outside of an active scissor region.

Like other OpenGL states, testing against a scissor region when rendering is enabled with **glEnable**, using the **GL_SCISSOR_TEST** symbolic constant. There can be only one scissor region active at a time, and the area this scissor region covers is set by the **glScissor** OpenGL function. This takes 4 parameters - the x and y axis start positions of the scissor region, and the x and y axis size of the region. Like **glViewport**, these parameters are in *screen space*, and so are measured in pixels. It's worth noting that scissor testing also influences **glClear** - it'll only clear what's inside the scissor region, if scissor testing is enabled.

Stencil Buffer

A more advanced method of masking off sections of the screen is to use the stencil buffer. This is a screen-sized buffer, just like the depth, front, and back buffers. Unlike the scissor regions, which limits rendering to a single rectangular section of screen, the stencil buffer can be rendered into just as you would a colour buffer; and then tested against when drawing geometry, just like the depth buffer you were introduced to last tutorial. For example, you could draw a circle to the stencil buffer, so only fragments within that circle are rendered - handy for a sniper's scope! Or perhaps draw your HUD before drawing your game world, then mask it off to save rendering fragments that will never be seen.

How many bits are set aside for each pixel in a stencil buffer is variable, although you can usually guarantee 8 bits per pixel on modern graphics hardware. The value written to a stencil buffer during rendering is programmable - writes can increment, decrement, overwrite, or perform a boolean operator on an existing stencil value. As with the depth buffer, we can decide to either allow or discard a fragment drawn into a pixel containing a stencil value. You could draw multiple objects into a stencil buffer, each of which increments the existing value in the stencil buffer - and then only allow further drawing into the colour buffer on pixels that have a stencil buffer value greater than 8, or exactly 4, if you really wanted to!

Testing and writing to the stencil buffer can be enabled using the **GL_STENCIL_TEST** symbolic constant, and controlled using two OpenGL functions - **glStencilFunc** and **glStencilOp**. **glStencilFunc** controls how the stencil buffer is tested against, and has three parameters, *func*, *ref*, and *mask*. Stencil testing works the same as the depth testing introduced earlier in the tutorial series - you can check if the existing value is greater than, less than, or equal to a value, among others, set with the *func* parameter. The value used by depth testing was the eye-space z coordinate, but stencil testing uses the value provided by the *ref* parameter. Both the reference parameter, and the existing parameter, are **AND**ed with the value of the *mask* parameter to make the comparison. This allows us to use a single stencil buffer for many concurrent tests. Here are some examples of using the stencil func:

glStencilFunc(GL_ALWAYS, 1, 0) - If we enable the stencil buffer, and use this stencil function to determine what goes into the stencil buffer, the stencil test will always pass (due to the GL_ALWAYS operation).

glStencilFunc(GL_GREATER, 1, ~0) - In this case, both the ref value (1 in this case) and the existing value in the stencil buffer will be ANDed together with the mask value 0, which performs a bitwise NOT operation. This will invert the value 0 to be all 1s - meaning the mask will leave the values unaffected (ANDing something with all 1s results in the original value). As the stencil function is GL_GREATER, the stencil will only allow values greater than the existing value to pass - so in this case, if the stencil buffer had 0 in it, this stencil func would pass (1 is greater than 0!), otherwise the stencil func will fail, and no drawing will take place.

glStencilFunc(GL_EQUAL, 255, 8) - If the value in the stencil buffer was *1*, this test would *fail* as we're testing for equality - the reference value of 255 **AND**ed with 8 (only the 4th *bit* of the mask is enabled) is 8, while the buffer value of 1 **AND**ed with 8 is 0. 0 and 8 are obviously not equal, so the currently processed fragment is discarded. If however, the existing stencil buffer value *did* have the 4th bit set, then the stencil test would *pass*.

What happens when a stencil test either passes or fails is determined by the **glStencilOp** function. This function takes 3 parameters, which control what happens when a stencil test fails, when a *stencil* test passes but then the fragment fails the *depth* test, and when both the stencil and depth test pass (or the stencil test passes and depth testing is disabled), respectively. We can then either keep, reset, increment, or replace the current stencil buffer value. This allows you to test against the stencil buffer without actually updating its contents, if you so wish. More examples!

glStencilOp(GL_ZERO, GL_KEEP, GL_KEEP) - If the stencil test fails, this will set the stencil buffer at that test location to zero. If the stencil test passes, even if the depth test fails, then the stencil buffer is left alone.

glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE) - In this example, if the stencil test fails, or passes when the depth test fails, nothing happens. If however both the stencil and depth test *pass*, the value in the stencil buffer is replaced by the current *ref* value of the stencil func.

glStencilOp(GL_KEEP, GL_REPLACE, GL_KEEP) - Here, if the stencil test passes, but the depth test fails, the value in the stencil buffer will be replaced. This can be used to determine where one object intersects with another, as the parts of object A that intersect object B will fail the depth test.

Example program

The example program for this tutorial will do two things - use scissor testing to limit the rendering of a triangle to a rectangular region in the middle of the screen, and use stencil test to limit the rendering to a chessboard pattern across the screen, via an alpha-mapped texture, to show how geometry drawn into the stencil buffer can affect subsequent rendering. We'll be reusing a vertex shader from earlier in the tutorial series, but writing a new fragment shader - so add a new text file called *StencilFragment.glsl* to your *Shaders* folder, and add a *Renderer* class, and a *Tutorial5.cpp* file to the *Tutorial5* project.

Renderer header file

In our *Renderer* class header file, we have two new public functions - *ToggleScissor* and *ToggleStencil*, which control a couple of protected member variables, *usingScissor* and *usingStencil*. We also declare two *Meshes*.

```
1  #pragma once
2
3  #include "./nclgl/OGLRenderer.h"
4
5  class Renderer : public OGLRenderer {
6  public:
7     Renderer(Window &parent);
8     virtual ~Renderer(void);
9
10    virtual void RenderScene();
11
12    void  ToggleScissor();
13    void  ToggleStencil();
14
15 protected:
16    Mesh* triangle;
17    Mesh* quad;
18
19    bool usingScissor;
20    bool usingStencil;
21 };
```

renderer.h

Renderer Class file

We start our *Renderer* class with our **constructor**, as usual. We use it to initialise our two meshes, load in our shader, and load in the two textures we'll be using for this tutorial. Note that the shader uses the *TexturedVertex.glsl* vertex shader we wrote back in Tutorial 3, and the *StencilFragment.glsl* file we'll be writing shortly. Everything we create we must of course **delete**, so our **destructor deletes** our two meshes, which will in turn **delete** our textures.

```
1  #include "Renderer.h"
2
3  Renderer::Renderer(Window &parent) : OGLRenderer(parent) {
4     triangle = Mesh::GenerateTriangle();
5     quad     = Mesh::GenerateQuad();
6
```

```
7     currentShader = new Shader("../Shaders/TexturedVertex.glsl",
8                                "../Shaders/StencilFragment.glsl");
9
10    if(!currentShader->LinkProgram()) {
11       return;
12    }
13
14    triangle->SetTexture(SOIL_load_OGL_texture("../Textures/brick.tga",
15             SOIL_LOAD_AUTO, SOIL_CREATE_NEW_ID, 0));
16    quad->SetTexture(SOIL_load_OGL_texture("../Textures/chessboard.tga",
17             SOIL_LOAD_AUTO, SOIL_CREATE_NEW_ID, 0));
18
19    if(!triangle->GetTexture() || !quad->GetTexture()) {
20       return;
21    }
22
23    usingScissor = false;
24    usingStencil = false;
25    init         = true;
26 }
27
28 Renderer::~Renderer(void)  {
29    delete triangle;
30    delete quad;
31 }
```

renderer.cpp

Now we'll get the boring toggle functions out of the way. Like with previous tutorials, they use the **NOT** boolean operator to flip our **bools** around, but this time we don't directly enable or disable the relevant OpenGL state - *why* will be explained shortly!

```
32 void  Renderer::ToggleScissor()   {
33    usingScissor = !usingScissor;
34 }
35 void  Renderer::ToggleStencil()   {
36    usingStencil = !usingStencil;
37 }
```

renderer.cpp

The RenderScene function is a little bit more complicated this time around. As usual, we clear the buffers - this time, with the **GL_STENCIL_BUFFER_BIT** symbolic constant **OR**'d in, which will clear the stencil buffer to a value of all 1s. Then, if scissor testing has been selected, we **glEnable** the scissor test, and use **glScissor** to restrict further scene rendering to a rectangle roughly in the middle of the screen - remember, **glScissor** works directly in *screen coordinates*, so we've used the screen width and height *OGLRenderer* member variables to define the region. Also, we wait until *after* **glClear** is called - remember, **glScissor** affects **glClear**! *That's* why we don't directly enable or disable the scissor test in our toggle function.

```
38 void Renderer::RenderScene()  {
39    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT
40             | GL_STENCIL_BUFFER_BIT);
41
42    if(usingScissor) {
43       glEnable(GL_SCISSOR_TEST);
44       glScissor((float)width / 2.5f, (float)height / 2.5f,
45                 (float)width / 5.0f, (float)height / 5.0f);
46    }
```

renderer.cpp

Then, we bind our shader, and update its matrices, as usual. As in the texturing tutorial, we also bind our shader's *diffuseTex* texture sampler to texture unit 0.

```
47    glUseProgram(currentShader->GetProgram());
48    UpdateShaderMatrices();
49
50    glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
51                "diffuseTex"), 0);
```

renderer.cpp

Here's the interesting bit! If we have stencil testing enabled, we're going to draw a screen-sized quad over the screen, that has a chessboard texture applied to it. The black chessboard tiles will write the value *2* to the stencil buffer, which we will then set up to only allow subsequent draws to pass if the stencil buffer at that fragment is equal to. But, we don't actually want the quad to render to the screen, only to the stencil buffer, so we use a function you may not have come across before: **glColorMask**. This function lets you switch off colour writing per channel - there's 4 parameters, 1 each for red, green, blue, and alpha, and we set them all to **false**. Even though this will disable all anything being written to the active buffer, all geometry will still pass through the rendering pipeline, so it'll still effect the stencil buffer.

So, we enable the stencil buffer, turn off colour writes, use the stencil functions to *always* write a value of *2* into the stencil buffer, and then draw our chessboard textured quad over the screen. We then turn colour writes back on, and set the stencil buffer to only allow fragments to pass on sections of the stencil buffer with a value of *2*; we also don't want anything else changing the stencil buffer, so we use **glStencilOp** to *keep* the stencil buffer as it is no matter what it contains.

```
52    if(usingStencil)  {
53       glEnable(GL_STENCIL_TEST);
54
55       glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
56       glStencilFunc(GL_ALWAYS, 2, ~0);
57       glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
58
59       quad->Draw();
60
61       glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
62       glStencilFunc(GL_EQUAL, 2, ~0);
63       glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
64    }
```

renderer.cpp

Finally, draw the triangle. We also **glDisable** scissor and stencil testing, ready for the next frame's rendering.

```
65    triangle->Draw();
66
67    glUseProgram(0);
68    glDisable(GL_SCISSOR_TEST);
69    glDisable(GL_STENCIL_TEST);
70
71    SwapBuffers();
72 }
```

renderer.cpp

Main file

Nice and simple, all we change from our very first main file is a couple of keyboard checks to toggle scissor and stencil testing.

```
#include "./nclgl/window.h"
#include "Renderer.h"
#pragma comment(lib, "nclgl.lib")

int main() {
	Window w("Index Buffers!", 800,600,false);
	if(!w.HasInitialised()) {
		return -1;
	}

	Renderer renderer(w);
	if(!renderer.HasInitialised()) {
		return -1;
	}

	while(w.UpdateWindow() &&
	!Window::GetKeyboard()->KeyDown(KEYBOARD_ESCAPE)){
		if(Window::GetKeyboard()->KeyTriggered(KEYBOARD_1)) {
			renderer.ToggleScissor();
		}
		if(Window::GetKeyboard()->KeyTriggered(KEYBOARD_2)) {
			renderer.ToggleStencil();
		}

		renderer.RenderScene();
	}
	return 0;
}
```

main.cpp

Fragment Shader

You may have been wondering how the chessboard texture will selectively write to the stencil buffer. Well, the texture's white tiles have an alpha value of 0 - something we can check for in the fragment shader! You may have already written something like this last tutorial, but if not, here's how to **discard** a fragment based on its alpha, so that neither the depth *or* the stencil buffer are updated. We do a simple **if** statement to check if the incoming alpha value is 0.0, and if so use the GLSL keyword **discard**. Note how we don't ever actually enable alpha blending in OpenGL - being able to sample an alpha value from a texture is entirely separate to the alpha blending process.

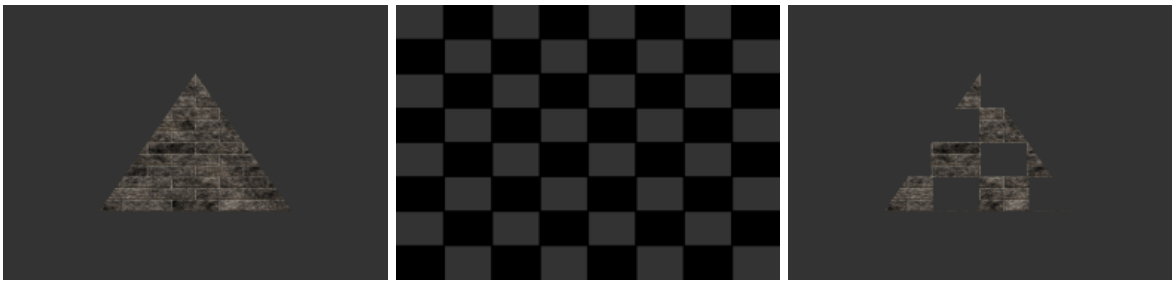
```
#version 150 core

uniform sampler2D diffuseTex;

in Vertex	{
	vec2 texCoord;
} IN;

out vec4 gl_FragColor;

void main(void)	{
```

```
12    vec4 value = texture(diffuseTex, IN.texCoord).rgba;
13
14    if(value.a == 0.0) {
15       discard;
16    }
17    gl_FragColor   = value;
18 }
```

StencilFragment.glsl

Running the Program

If everything works correctly, you'll see a textured triangle on screen when running this program - not very interesting! Pressing the *1* key will enable scissor testing, and restrict the drawing of the triangle to a small box in the middle of the screen. Pressing the *2* key will enable stencil testing, resulting in a a chessboard-like restriction of drawing. This is due to the chessboard texture we use on the quad that we draw over the screen if stencil testing is enabled - the 'white' board pieces have an alpha of 0.0, so only the 'black' board pieces write to the stencil buffer. As we disable colour writes, the colour buffer isn't written to by our chessboard quad, but the stencil buffer is updated - which we then test against when drawing the triangle. Stencil buffers and scissor tests aren't mutually exclusive - we can enable both if we want! If you still don't quite get how the stencil buffer is working, try commenting out line 55 and running it again, it should make it a bit clearer how the chessboard texture is selectively disabling colour writes to sections of the screen.



The left image is combined with the centre image stenciled regions, making the right hand image

Tutorial Summary

After completing this simple tutorial, you should have a pretty good idea of how to use both stencil buffers and scissor regions in your game rendering. There's not much more to say on scissor regions, but stencil buffers can be a bit more complicated - it's possible to add multiple stenciled regions together in a stencil buffer, using the *mask* variable of **glStencilFunc**; but for now you should have enough knowledge of stencil buffers to do some interesting effects in your games. In the next few tutorials, we're going to start extending the *Mesh* class we made back in Tutorial 1, to support a more efficient method of rendering called an *index buffer*. It'll also show you how to create terrain using a *heightmap* - finally, no more simple quads and triangles! We'll also look at how to organise your game objects using something called a *scene graph*.

Further Work

1) The program is currently set up to stencil the black chessboard pieces - what changes would have to be made to stencil off the white chessboard pieces instead?

2) Like the colour buffer and its **glClearColor** function, it is possible to set the clear value of the stencil buffer. Investigate the **glClearStencil** function.

3) It's possible to yet further control the writing to the stencil buffer, by turning off writes to individual bits. Investigate the **glStencilMask** OpenGL function.