UOW AUSTRALIA

SCIT

School of Computing & Information Technology

CSCI336 – Interactive Computer Graphics

Face Culling, the Depth Test and Render-to-Texture

Face Culling

Open the Tutorial8a project. This example demonstrates face culling.

1. First, compile and run the program. Using the user interface, check the Wireframe option to render the scene in wireframe. Note that all polygons are currently rendered.
2. Now, check the CullFace option. You will see that approximately half the polygons will not be rendered. These are the back facing polygons, i.e. back-face culling.
3. Rotate the object. You will see that the front facing polygons will be render, whereas the back facing polygons will not be.
4. Uncheck the Wireframe option. Then repeatedly check and uncheck the CullFace option, you will see that there is no difference in the image with and without back-face culling, even though there is a difference in terms of the amount of computation required for rendering.
5. By default the backfaces of polygons will be culled, since this is what is most likely required. Check the FrontFace option, this sets the face to cull as front facing polygons rather than back facing polygons. Notice that now the back facing polygons are displayed because the front facing polygons are no longer rendered.
6. OpenGL uses a right-handed coordinate system. Hence, by default polygons with clockwise winding are defined as back facing polygons, whereas polygons with counter-clockwise winding are defined as front facing polygons. Check the CW Winding option. This sets the back facing polygons as counter-clockwise winding.

The functions to set face culling are simple. Have a look at the code in the rendering loop.

- Culling is disabled by default and can be enabled using `glEnable(GL_CULL_FACE)` or disabled using `glDisable(GL_CULL_FACE)`
- The face to cull is set using `glCullFace` (by default it is set to GL_BACK).
- Front face winding is set using `glFrontFace` (by default this is set to GL_CCW).

There is also a built-in system Boolean variable in the fragment shader that indicates whether a fragment is front facing: `gl_FrontFacing`

In the fragment shader, uncomment the following code:

```
if(!gl_FrontFacing)
{
```

```
            fColor = vec3(1.0f, 1.0f, 1.0f);
            return;
    }
```

This will set the colour of back facing fragments to white and will skip lighting computations.

To see this, compile and run the program, also make sure you select options which allow you to see the back facing polygons (set the rendering to wireframe mode, or enable culling and set the cull face to cull front facing polygons).

The Depth Buffer Test

In most of the previous tutorial programs, we have enabled the depth buffer test. The depth buffer test is enabled using (disabled by default):

```
glEnable(GL_DEPTH_TEST);
```

It can be disabled using:

```
glDisable(GL_DEPTH_TEST);
```

Before rendering the image, the depth buffer is initialised or "cleared" using:

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

This example shows what the depth buffer test does. Open the Tutorial8b Visual Studio project. Compile and run the program.

There are 3 objects in the scene. The user interface allows you to translate one of the cubes. Move this cube around; move into/behind the other cube; move it into/beneath the floor (you can set rendering to wireframe mode to see that the object is still there). You will see that the scene is rendered with visibility determined correctly. This is because the depth buffer test is enabled.

Now, uncheck the depth buffer test option. The scene now looks weird. This is because visibility is not determined correctly. With the depth buffer test disabled. Render resorts to the "painter's algorithm", where polygons that are rendered last will overwrite the previous contents in the colour buffer.

If you look at the code in render_scene(), the floor is rendered last. Hence, it always overwrites anything that was rendered before it. The same applies to the polygons of the two cubes. Polygons that are rendered later will always overwrite anything that was previously rendered. The cube that you can translate is rendered later than the other cube. Therefore, even if you change the translate z value to be behind the other cube, it will still be rendered on top of the other cube.

Render-to-Texture

In this example, we will look at a technique called render to texture. As the name indicates, instead of rendering to image to the colour buffer for display, we will render the image into a texture instead. This is useful for a variety of rendering effects.

To see the results, we will use the texture and display on a screen sized quad (i.e. a quad that covers the screen and is textured with the texture that we rendered the image to).

For this, we need to create a framebuffer object. The framebuffer is where the rendered information is stored in. We create a framebuffer object and associate it with a texture, so that the rendered information will be stored in the texture instead of the usual colour buffer. A framebuffer object identifier is declared:

```
GLuint g_FBO = 0;
```

In this example, we will use four texture objects; Two for normal texturing (this example also shows how to texture different objects with different textures). One to store the rendered colours and another to store the contents of the depth buffer.

```
GLuint g_textureID[4];
```

The first two textures are initialised the same way as in the previous example. The third and fourth textures are created slightly differently:

```
glBindTexture(GL_TEXTURE_2D, g_textureID[2]);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, g_windowWidth, g_windowHeight, 0, GL_RGB,
       GL_UNSIGNED_BYTE, 0);
```

Notice that 0 is passed as the last argument and the window's width and height are passed as the width and height of the texture.

For the depth buffer texture:

```
glBindTexture(GL_TEXTURE_2D, g_textureID[3]);

glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT16, g_windowWidth, g_windowHeight, 0,
       GL_DEPTH_COMPONENT, GL_FLOAT, 0);
```

Notice that the for the format and type, we use a 16-bit depth format as floats. The last argument is also 0.

Next, we create and initialise the framebuffer object. We generate an identifier for the framebuffer object and bind this to the framebuffer:

```
glGenFramebuffers(1, &g_FBO);
glBindFramebuffer(GL_FRAMEBUFFER, g_FBO);
```

In this example, we want to see the contents of the depth buffer. Therefore, we create and initialise a render buffer for the depth buffer (i.e. a buffer to render depth values to instead of the usual depth buffer) and attach it to the framebuffer for rendering:

```
GLuint depthRenderBufferID;
glGenRenderbuffers(1, &depthRenderBufferID);
glBindRenderbuffer(GL_RENDERBUFFER, depthRenderBufferID);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, g_windowWidth, g_windowHeight);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER,
       depthRenderBufferID);
```

Then, the textures that we previously created are then associated with the respective framebuffer attachments (i.e. a colour buffer and the depth buffer):

```
glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, g_textureID[2], 0);
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, g_textureID[3], 0);
```

This is followed by the following to set the list of buffers to draw to:

```
GLenum drawBuffers[1] = { GL_COLOR_ATTACHMENT0 };
glDrawBuffers(1, drawBuffers);
```

The following checks whether it was successful:

```
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
	exit(EXIT_FAILURE);
```

To render to the framebuffer object that we created, before rendering we have to bind the framebuffer. The following is at the start of the render_scene() function:

```
glBindFramebuffer(GL_FRAMEBUFFER, g_FBO);
```

The scene is drawn using the normal commands. However, the image will now be drawn into the framebuffer object that we created.

To render objects using different textures, we simply have to bind it to the texture unit before drawing the object. The first object uses the first texture:

```
	glActiveTexture(GL_TEXTURE0);
	glBindTexture(GL_TEXTURE_2D, g_textureID[0]);

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

The second object uses the second texture:

```
	glBindTexture(GL_TEXTURE_2D, g_textureID[1]);

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

The third object goes back to using the first texture (this is merely an example, for efficiency reasons it makes more sense to group objects that use the same properties):

```
	glBindTexture(GL_TEXTURE_2D, g_textureID[0]);

	glDrawArrays(GL_TRIANGLES, 0, 6);
```

To see the contents to textures, we use a screen space quad, draw this to the screen and texture it with the contents of the textures that we created. The ScreenQuadVS.vert and ScreenQuadFS.frag shaders are used for this purpose.

A screen space quad is created with -1 and +1 in the x and y dimensions:

```
GLfloat g_screenspaceQuad[] = {
	-1.0f, 1.0f, 0.0f,
	-1.0f, -1.0f, 0.0f,
	1.0f, 1.0f, 0.0f,
	1.0f, 1.0f, 0.0f,
	-1.0f, -1.0f, 0.0f,
	1.0f, -1.0f, 0.0f,
```

```
};
```

The ScreenQuadVS.vert simply sets positions of the vertices, calculates the texture coordinates and passes these to the fragment shader. The ScreenQuadFS.frag simply maps the fragment colour with the appropriate colour from either the colour buffer texture or the depth buffer texture, depending on which we passed to the shader.

The following is used to render to the screen space quad. First, the framebuffer is reset to the usual framebuffer and the buffers are cleared:

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Then, ScreenQuadVS.vert and ScreenQuadFS.frag shaders that were compiled as the second shader program are used:

```
glUseProgram(g_shaderProgramID[1]);
```

The vertex array object is bound and the texture unit0 activated.

```
glBindVertexArray(g_VAO[1]);
glUniform1i(g_texSampler_Index[1], 0);
glActiveTexture(GL_TEXTURE0);
```

The texture that we pass depends on what we select using the AntTweakBar interface. Either the colour buffer texture or the depth buffer texture:

```
if (g_bRenderDepth)
{
	glUniform1i(g_renderDepth_Index, 1);
	glBindTexture(GL_TEXTURE_2D, g_textureID[3]);
}
else
{
	glUniform1i(g_renderDepth_Index, 0);
	glBindTexture(GL_TEXTURE_2D, g_textureID[2]);
}
```

Finally, we render the screen space quad:

```
glDrawArrays(GL_TRIANGLES, 0, 6);
```

Compile and run the program. It looks no different from normal rendering. To see a difference, change the vertices in the definition of the screen space quad. At the moment, the vertices are defined between +/- 1. Change this to be between +/- 0.5 and you will see that the quad is only rendered to a smaller section of the display.

If you uncheck the DepthTest in the interface, you will see the scene rendered without the depth buffer test and visibility determination is not done correct.

To see the contents of the depth buffer, check the RenderDepth. This shows a black and white image that indicates the depth values of the pixels (distance from the screen). The contents of the depth buffer are between 0.0f (closest to the screen) and 1.0f (furthest from the screen). The actual contents of the depth buffer are really hard to see, as such this example has modified the colour so that you can see the contents easier:

```
fColor = (texture(uTextureSampler, vTexCoord).rrr * 10.0f) - 9.0f;
```

To see the actual contents of the depth buffer, you can change this to:

```
fColor = texture(uTextureSampler, vTexCoord).rrr;
```

If you change it, you may be able to see the contents if you look at the image really closely, but it is very difficult to see a difference in the values.

Screen Capture

The code in Tutorial8c also contains an example of how to save the contents of the display into a file. The bmpfuncs.h and bmpfuncs.cpp files now contain a function to write contents into a 24-bit RGB bitmap image file:

```
void writeBitmapRGBImage(const char *filename, char* imageData, int width, int height);
```

The function itself is very much the reverse of the readBitmapRGBImage() function.

In the key_callback() function in Tutorial8c.cpp, you will see some code added to take a screen capture when you press the '0' key. What the code does is it allocates enough memory for a buffer based on the window width and height:

```
int size = g_windowWidth * g_windowHeight * 3;

unsigned char* outBuffer = new unsigned char[size];
```

The following command is used to read the contents from the colour buffer into the buffer that was created:

```
glReadPixels(0, 0, g_windowWidth, g_windowHeight, GL_RGB, GL_UNSIGNED_BYTE,
       (GLvoid*)outBuffer);
```

The contents is written into a BMP image file:

```
writeBitmapRGBImage("images/screenshot.bmp", (char*)outBuffer, g_windowWidth,
       g_windowHeight);
```

Finally, the buffer memory is deallocated:

```
delete[] outBuffer;
```

References

Among others, much of the material in this tutorial was sourced from:

- Angel & Shreiner, “Interactive Computer Graphics: A Top-Down Approach with OpenGL”, Addison Wesley
- http://www.opengl-tutorial.org/
- http://ogldev.atspace.co.uk/index.html