CSCI 4110 Laboratory Five

Only the Shadow Knows

Introduction

In this laboratory we will implement the shadow maps that we discussed in class. The first step in the laboratory is to make our test scene more interesting, since with a single object we won't be able to see any shadows. The second step is to create the shadow map and the third step is rendering our model with shadows. We will start with the code from laboratory four, since it draws our sphere with the full Phong lighting model.

Adding to the Model

We will make our model slightly more complex by adding a plane below the sphere, so the sphere can cast a shadow on it. The plane will be a separate object and we can modify the init procedure for the cube to do this. Start by adding the following two global variables to the program:

```
GLuint planeVAO;
GLuint planeBuffer;
```

You should also make the ibuffer variable in the init procedure for the sphere global at the same time. Now change the name of the procedure to initPlane, so we don't have two init procedures in our program. We only need one face of the cube, so you only need to keep the first four entries of the vertices and normals arrays. Change the entries in the vertices array so they all lie in the z = -2.0 plane and they form a square in the place. Change the normals array entries so they are all (0.0, 0.0, 1.0). Keep the first 6 entries of the indexes array and keep the same values for them. Make sure that you use planeVAO and planeBuffer at the appropriate places in this procedure.

Now we need to make a few changes to our displayFunc procedure. The last few lines of this procedure should be:

```
glBindVertexArray(objVAO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibuffer);
glDrawElements(GL_TRIANGLES, 3*triangles, GL_UNSIGNED_INT, NULL);

glBindVertexArray(planeVAO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, planeBuffer);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, NULL);
```

Finally in the main procedure we change the way we are loading the shaders to the following:

```
vs = buildShader(GL_VERTEX_SHADER, "lab4a.vs");
fs = buildShader(GL_FRAGMENT_SHADER, "lab4c.fs");
program = buildProgram(vs, fs, 0);
```

You might also want to change the values of eyey and r from 5.0 to 7.0 to get a better view of the model. Once you have done this compile the program and run it. You should get something

similar to figure 1. Notice that the plane has a bright specular reflection where the shadow from the sphere should be.

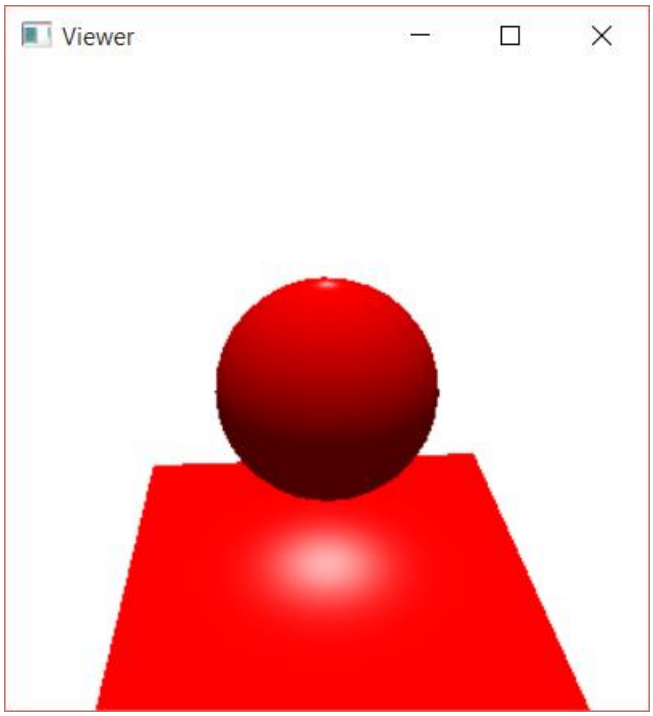

Figure 1 Model without Shadows

Building the Shadow Map

The next step is to create the shadow map. For this we will need a frame buffer object that the shadow map will be written into. In theory the frame buffer object only needs a depth map, since that is what we are interested in. In practice we seem to need to add a colour buffer otherwise OpenGL thinks the frame buffer object isn't complete and won't use it. Since the plane doesn't have anything to cast shadows onto we will only process the sphere.

In the init procedure we add the following code to create the frame buffer object:

```
	glGenFramebuffers(1, &shadowBuffer);
	glBindFramebuffer(GL_FRAMEBUFFER, shadowBuffer);
	glGenTextures(1, &shadowTex);
	glBindTexture(GL_TEXTURE_2D, shadowTex);
	glTexStorage2D(GL_TEXTURE_2D, 1, GL_DEPTH_COMPONENT32, 512, 512);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
GL_COMPARE_REF_TO_TEXTURE);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);

	glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, shadowTex, 0);

	glGenTextures(1, &colourTex);
	glBindTexture(GL_TEXTURE_2D, colourTex);
	glTexStorage2D(GL_TEXTURE_2D, 1, GL_R32F, 512, 512);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, colourTex, 0);

glBindTexture(GL_TEXTURE_2D, 0);
```

There is nothing particularly new here, the last line ensures that the texture we created isn't accidently used by another part of the rendering process.

We have a very simple set of shaders for the shadow map. In the main procedure we have the following code:

```
vs = buildShader(GL_VERTEX_SHADER, "shadow.vs");
fs = buildShader(GL_FRAGMENT_SHADER, "shadow.fs");
shadowProgram = buildProgram(vs, fs, 0);
```

At the end of the init procedure we have the following code so the shader program can access the vertex information for the sphere:

```
glUseProgram(shadowProgram);
vPosition = glGetAttribLocation(program, "vPosition");
glVertexAttribPointer(vPosition, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(vPosition);
```

Now we need to add some code to the display function to draw the shadow map. This code is:

```
glBindFramebuffer(GL_FRAMEBUFFER, shadowBuffer);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(shadowProgram);

view = glm::lookAt(glm::vec3(0.0, 0.0, 3.0),
       glm::vec3(0.0f, 0.0f, 0.0f),
       glm::vec3(0.0f, 1.0f, 0.0f));

scale = glm::mat4(glm::vec4(0.5, 0.0, 0.0, 0.0),
       glm::vec4(0.0, 0.5, 0.0, 0.0),
       glm::vec4(0.0, 0.0, 0.5, 0.0),
       glm::vec4(0.5, 0.5, 0.5, 1.0));

proj = glm::frustum(-1.0, 1.0, -1.0, 1.0, 1.0, 100.0);

shadowMatrix = scale*proj*view;

viewLoc = glGetUniformLocation(shadowProgram, "modelView");
glUniformMatrix4fv(viewLoc, 1, 0, glm::value_ptr(view));
projLoc = glGetUniformLocation(shadowProgram, "projection");
glUniformMatrix4fv(projLoc, 1, 0, glm::value_ptr(proj));

glViewport(0, 0, 511, 511);
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(20.0f, 6.0f);
glDrawBuffers(1, buffs);

status = glCheckFramebufferStatus(GL_DRAW_FRAMEBUFFER);
if (status != GL_FRAMEBUFFER_COMPLETE) {
       printf("bad framebuffer object\n");
```

```
    }

    glBindVertexArray(objVAO);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibuffer);
    glDrawElements(GL_TRIANGLES, 3 * triangles, GL_UNSIGNED_INT, NULL);

    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glDisable(GL_POLYGON_OFFSET_FILL);
```

This code needs a bit of explanation. We start by creating the projection and viewing matrices that are used for drawing the shadow map. We also create a scale and shadowMatrix which will be used later when we render our model. We have set the resolution of our frame buffer object to 512 x 512, so we need to change the viewport to that size. We then have two lines of code that effect how the depth buffer is updated. We are drawing the sphere into the depth buffer, and we will be drawing the same sphere when we render the model. We need to be careful that the sphere doesn't shadow itself, after all the depth of the sphere will be the same as the values stored in the depth buffer. To avoid this problem we use a polygon offset, OpenGL generates an offset, on a per pixel basis, that is added to the depth value. The parameters to glPolygonOffset are parameters to this calculation and quite often need to be set by trial and error (the second parameter is implementation depend so isn't even described in the standard).

We next need to enable the colour buffers, which is done by the glDrawBuffers call. The buffs variable is declared in the following way:

```
    GLenum buffs[] = { GL_COLOR_ATTACHMENT0 };
```

We check to see if the frame buffer object has been constructed correctly and then draw the sphere the way that we normally do.

The vertex shader for the shadow map is:

```
#version 330 core

uniform mat4 modelView;
uniform mat4 projection;
in vec4 vPosition;

void main() {

    gl_Position = projection * modelView * vPosition;

}
```

The fragment shader for the shadow map is:

```
#version 330 core

void main() {
```

```
    gl_FragColor =vec4(1.0);
}
```

We can now compile our code and run it. Since we are not using the shadow map for rendering the model the output should look the same. I saved the shadow map to a file and figure 2 shows what it looks like.

Figure 2 Shadow map

Using the Shadow Map

Now that we have the shadow map it's time to use it in rendering our model. We need to compare the distance of the current point that we are rendering to the value in the shadow map. There are several ways in which this can be done, but one of the easiest is to transform each vertex with the same transformation used to construct the shadow map and pass this value to the fragment shader. The fragment shader can then compare this value to the one in the shadow map to determine if the pixel is in a shadow. There is one complication here. The coordinates used in texture lookup go from 0 to 1, while the coordinates that come from the projection and viewing transformations go from -1.0 to 1.0. The scale matrix that we constructed earlier handles this problem.

The code in the displayFunc procedure for rendering the model is:

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

view = glm::lookAt(glm::vec3(eyex, eyey, eyez),
       glm::vec3(0.0f, 0.0f, 0.0f),
       glm::vec3(0.0f, 0.0f, 1.0f));

viewLoc = glGetUniformLocation(program, "modelView");
glUniformMatrix4fv(viewLoc, 1, 0, glm::value_ptr(view));
projLoc = glGetUniformLocation(program, "projection");
glUniformMatrix4fv(projLoc, 1, 0, glm::value_ptr(projection));
shadowMatrixLoc = glGetUniformLocation(program, "shadowMatrix");
glUniformMatrix4fv(shadowMatrixLoc, 1, 0, glm::value_ptr(shadowMatrix));
```

```
colourLoc = glGetUniformLocation(program, "colour");
glUniform4f(colourLoc, 1.0, 0.0, 0.0, 1.0);
lightLoc = glGetUniformLocation(program, "light");
glUniform3f(lightLoc, 0.0, 0.0, 3.0);
materialLoc = glGetUniformLocation(program, "material");
glUniform4f(materialLoc, 0.3, 0.7, 0.7, 150.0);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, shadowTex);
glViewport(0, 0, width, height);

glBindVertexArray(objVAO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibuffer);
glDrawElements(GL_TRIANGLES, 3*triangles, GL_UNSIGNED_INT, NULL);

glBindVertexArray(planeVAO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, planeBuffer);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, NULL);

glutSwapBuffers();
```

This is pretty much the same as what we had before. We have one extra transformation matrix, the shadow matrix that we need to send to the vertex shader. We also need to bind the shadow map texture, but otherwise everything else is basically the same.

The code for the vertex shader is:

```
#version 330 core

uniform mat4 modelView;
uniform mat4 projection;
uniform mat4 shadowMatrix;
in vec4 vPosition;
in vec3 vNormal;
out vec3 normal;
out vec3 position;
out vec4 sPosition;

void main() {

    gl_Position = projection * modelView * vPosition;
    position = vPosition.xyz;
    normal = vNormal;
    sPosition = shadowMatrix * vPosition;

}
```

This is basically the same as what we had in laboratory four, except we now have an extra matrix as input and produce another output, sPosition, the vertex position in the light coordinate system.

The code for the fragment shader is:

```
#version 420 core

in vec3 normal;
in vec3 position;
in vec4 sPosition;
uniform vec4 colour;
uniform vec3 light;
uniform vec4 material;
uniform sampler2DShadow tex;

void main() {
	vec4 white = vec4(1.0, 1.0, 1.0, 1.0);
	float diffuse;
	vec3 L = normalize(light);
	vec3 N;
	vec3 R = normalize(reflect(-(light-position),normal));
	float specular;
	float shadow;

	N = normalize(normal);
	diffuse = dot(N,L);
	if(diffuse < 0.0) {
		diffuse = 0.0;
		specular = 0.0;
	} else {
		specular = pow(max(0.0, dot(N,R)),material.w);
	}

	shadow = textureProj(tex, sPosition);
	gl_FragColor = material.x*colour;
	gl_FragColor += min((material.y*diffuse*colour + material.z*white*specular)*shadow,
vec4(1.0));
	gl_FragColor.a = colour.a;

}
```

Again this is pretty much the same as what we had before, except for the last few lines. We have the position of the fragment in light source space, sPosition, but it hasn't been projected yet. This is handled by textureProj that does both the projection and looks up the value in the shadow map. This function returns 0.0 if the fragment is in shadow and 1.0 otherwise. If we are performing any form of texture filtering the value could be between 0.0 and 1.0. The ambient

component of the light model isn't effected by the shadow, but the diffuse and specular components are so we divide the colour computation over two statements.

After putting this all together and running the program we get the result shown in figure 3.

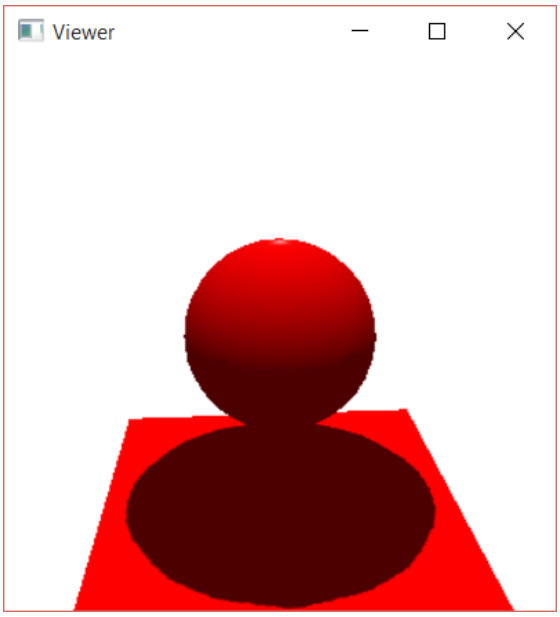

Figure 3 Rendering with shadow

Debugging OpenGL Programs

I had some problems getting the program for this laboratory to work. The problem is that OpenGL procedures fail silently. There are no error messages and you just get a blank screen or something that you didn't want. Early versions of OpenGL introduced the glGetError() procedure that tells you that an error occurred at some point in the past and gives you some idea of what the problem is. This is somewhat helpful, but you need to have some idea of where the problem could be otherwise you are calling glGetError after each OpenGL call.

More recent version of OpenGL provide a debug context which makes debugging easier. When the debug context is used your program can be informed of any problems through a callback procedure. This procedure is provided with a lot of information, but the most important one is a message that usually indicates what has gone wrong. One of the problems with this technique is the callback procedure is called quite often, even for completely correct programs. For example, it will be called each time your program does something that could be a performance problem. This is not strictly an error, but it is worth knowing. We can filter some of these messages out in our callback procedure. For this application I used the following callback procedure that only prints a message when an error occurs:

```
static void APIENTRY errorCallback(GLenum source, GLenum type, GLuint id, GLenum
severity,
	GLsizei length, const GLchar* message, void *userParam) {
	switch (type) {
	case GL_DEBUG_TYPE_ERROR:
		printf("error: %s\n", message);
```

```
            break;
        default:
            break;
        }
}
```

You can expand the switch statement to catch the other types of messages that are produced by the debug context. The debug content needs to be informed of the callback procedure that you want to use. This can be done in the following way:

```
glDebugMessageCallback(&errorCallback, NULL);
```

In addition you must add the following statement before glfwCreateWindow to create the debug context:

```
glfwWindowHint(GLFW_OPENGL_DEBUG_CONTEXT, GLFW_TRUE);
```

This is something that is easy to add to your program and can be quite helpful.

Laboratory Report

Try changing the value of the first parameter to glPolygonOffset and observe the impact that it has on the resulting image. Show your program to the TA and explain the results of your experiment with glPolygonOffset.