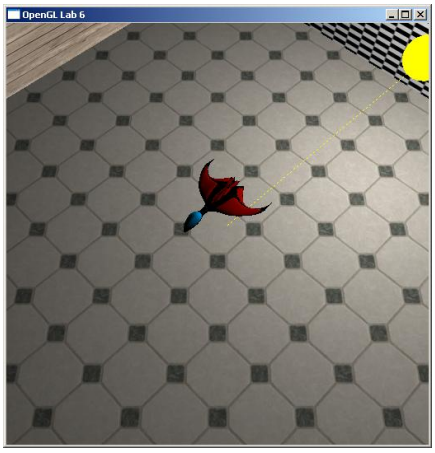
Lab 6 – Shadow Maps



In this lab, we will implement a technique called Shadow Mapping. Shadows greatly increase the realism in computer genereated images and this technique is the one most commonly used in games and realtime applications, these days. Make *OpenGL_Lab_6* the startup project and run the program. It should look like pictured, a light source rotates around the space ship in the scene, but were are the shadows? Lets fix this!

Before proceeding with this lab, make sure you read the section in the course book on shadow mapping (chapter 9.1.4). Then look through the code and make sure you understand how it works. Note that we set up a whole new *view* and *projection* matrix for the light source, as we will need to render the scene from the lights viewpoint to obtain our shadow map (much like we did for the tv camera in the previous tutorial).

Creating the shadowMap

A shadow map is simply a texture containing the depth buffer rendered from the point of view of the light. So, we will need to do one render-to-texture pass, as in the previous lab. First though, let's render the shadow map to the default frame buffer so we can see what it looks like.

Look at the `drawShadowMap()` method. It works just like `drawScene()`, only it uses a different, much simpler, shader that doesn't perform any lighting. For now, in `display()` comment out the calls to `drawScene()` and add a call to `drawShadowMap()` instead:

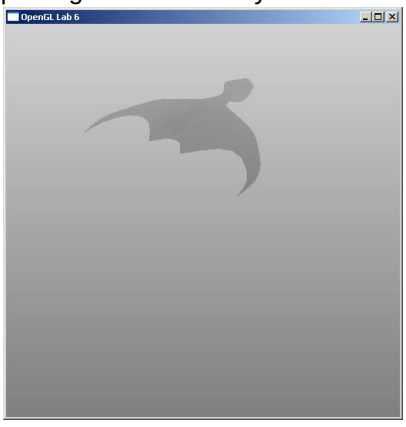
```
// drawScene(viewMatrix, projectionMatrix,
//    lightViewMatrix, lightProjectionMatrix);
```

The call to `drawShadowMap()`, should be added just after the setup of the light view and projection matrices at the beginning of display.

```
float4x4 lightProjectionMatrix = …;
drawShadowMap(lightViewMatrix, lightProjectionMatrix);
```

This will not look like much though since our fragment shader only writes (1,1,1,1) to the color buffer. To temporarily see what gets written into the depth-buffer, change the simple.frag shader so it says:

```
fragmentColor = vec4(gl_FragCoord.z);
```



This means that the fragments depth will be written to all the color channels as well as the depth buffer. Now run the program and see what the shadowMap will look like. A whiter color means a higher depth. It should look like the picture to the ritght.

When you understand what you are looking at, change the fragment shader back again. Also, uncomment the calls to `drawScene` again.

Now create the shadow map texture and the framebuffer used for drawing to that texture. Add at the end of initGL():

```
//****************************************
// Create shadowMap texture and framebuffer
//****************************************
glGenTextures( 1, &shadowMapTexture );
glBindTexture( GL_TEXTURE_2D, shadowMapTexture );
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32,
       shadowMapResolution, shadowMapResolution, 0,
       GL_DEPTH_COMPONENT, GL_FLOAT, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                  GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                  GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
                  GL_COMPARE_REF_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
float4 ones = {1.0, 1.0, 1.0, 1.0};
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, &ones.x);

glGenFramebuffers(1, &shadowMapFBO);
glBindFramebuffer(GL_FRAMEBUFFER, shadowMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D, shadowMapTexture, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
// Activate the default framebuffer again
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Note that we set the border color to 1.0, which corresponds to the maximum depth, and use the wrap mode of `GL_CLAMP_TO_BORDER`. This ensures that tests that fall outside the view of the light return the far depth, and will thus not be shadowed.

And in the beginning somewhere in the global scope, add these global variables:

```
GLuint shadowMapTexture;
GLuint shadowMapFBO;
const int shadowMapResolution = 1024;
```

In drawShadowMap() add:

```
glBindFramebuffer(GL_FRAMEBUFFER, shadowMapFBO);
glViewport(0,0,shadowMapResolution, shadowMapResolution);
```

on the first line and:

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

on the last.

Congratulations! Your program now creates a shadow map.

Using the shadowMap

Now that we have a shadow map, we want to change our shaders so that for every fragment drawn, we look into the shadow map to see if there is anything closer to the light. If there is, the fragment is in shadow. To do this, we will need to be able to transform a coordinate from the cameras view-space into the shadow maps *texture space*. Add the following code to `drawScene()`:

```
/*************************
 * Draw the floor
 ************************/
float4x4 modelViewMatrix = viewMatrix * floorModelMatrix;
float4x4 lightMatrix = lightProjectionMatrix * lightViewMatrix
                        * inverse(viewMatrix);
setUniformSlow(shaderProgram, "lightMatrix", lightMatrix);
```

Study the above transformation matrix carefully, reading from right to left: `inverse(viewMatrix)` transforms from (camera) view space to world space, `lightViewMatrix` transforms from world space to light view space and finally `lightProjectionMatrix` transforms from light view to light clip space. Thus, if we transform a coordinate in *camera view space* using this matrix, it will end up in *light clip space*.

Then in the vertex shader, add this uniform and a `shadowMapCoord` that can be sent to the fragment shader:

```
uniform mat4 lightMatrix;
out vec4 shadowMapCoord;
```

Then do the transformation on the last line of the vertex shader:

```
shadowMapCoord = lightMatrix * vec4(viewSpacePosition, 1.0);
```

Now `shadowMapCoord` is in the light clip space. However, clip space is in range (-1, -1, -1) to (1,1,1), whereas texture space is defined from (0,0,0) to (1,1,1). To be able to use our shadowMapCoord to look up in the shadow map texture we will need to transform it further. Add these two lines:

```
shadowMapCoord.xyz *= vec3(0.5f, 0.5f, 0.5f);
shadowMapCoord.xyz += shadowMapCoord.w * vec3(0.5f, 0.5f, 0.5f);
```

This will scale 0.5 the coordinates by half and then add 0.5, the use of `shadowMapCoord.w` in the second part is because the homogenous clip space coordinate are divided by w later, which will cancel out.

Time to do the actual lookup. In the fragment shader, first add the shadowMapCoord input:

```
in vec4 shadowMapCoord;
```

Then we need to add a sampler for the shadowMap:

```
uniform sampler2DShadow shadowMapTex;
```

Then, in drawScene() we need to bind the shadowMap to the sampler called '`shadowMapTex`':

```
setUniformSlow(shaderProgram, "shadowMapTex", 1 );
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, shadowMapTexture);
```

Finally in the fragment shader we can do a shadow-lookup, replace:

```
fragmentColor = vec4(diffuseColor * diffuseReflectance, 1.0);
```

With:

```
float visibility = textureProj(shadowMapTex, shadowMapCoord);
fragmentColor = vec4(diffuseColor * diffuseReflectance
                        * visibility, 1.0);
```

Assignment: Read in the GLSL 1.3 specification about `textureProj` and `texture`, how do they differ and why are we using `textureProj` here?

_____

_____

Now run the program. The shadow will be there, but there will be some very annoying patterns on the floor. This is called "surface acne" and is one of the problems with shadow-maps. Consider what happens when a fragment from the floor which is *not* in shadow looks into the shadow map. The closest depth in the shadowMap will be from *almost* the same part of the floor, but not exactly. So sometimes the lookup will return that the current fragment is closer and sometimes that it is further from the light than what is reported in the shadow map. A good figure that describes the problem is in the course book, **Figure 9.17**.

A simple, but not perfect, solution to this is to simply "push" the polygons a tiny bit away from the light when drawing the shadow map. Add the folowing to the begining of `drawShadowMap()`:

```
glPolygonOffset(2.5, 10);
glEnable(GL_POLYGON_OFFSET_FILL);
```

And then at the end of the same function, turn it off again:

```
glDisable(GL_POLYGON_OFFSET_FILL);
```

Now run the program and enjoy the shadows.

Assignments

- Make the shadow map smaller now. Make the resolution 256x256 instead. Now, if you zoom in on the shadow you will clearly see aliasing artifacts.

- These graphics cards support simple *Percentage Closer Filtering* in hardware. Change the lines:

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

  so they use `GL_LINEAR` filtering instead. Then run and zoom in on the shadows again. The problem will be slightly reduced, but far from completely removed.

- We adjust the shadow map coordinates in the vertex shader, by scaling by 0.5 and also adding 0.5. These transformations could easily be made part of the `lightMatrix`, recall that a 4x4 transformation matrix can represent scaling and translation. Implement this in `lab6_main.cpp`, where the `lightMatrix` is constructed. Remember to comment out the corresponding lines in the vertex shader!

  Hint: Make use of the matrix functions `make_scale<float4x4>` and `make_translation`.

When done, show your result to one of the assistants. Have the finished program running and be prepared to answer some questions about what you have done.