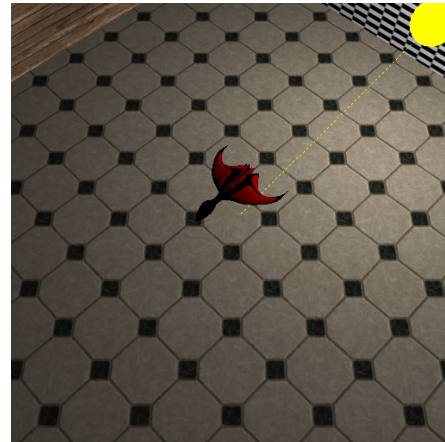


# Lab6 : Shadow Maps

In this lab, we'll implement a technique called *Shadow Mapping*. Shadows greatly increase the realism of computer generated images, and shadow mapping is one of the most commonly used techniques to produce shadows in real time applications.

To start with this lab, run **Lab 6**. You should see something similar to the image shown on the right. There's a light source rotating around the space ship in the middle - but the shadows are missing! Let's fix that.

**Important** Before proceeding with this lab, make sure that you've read the section on shadow mapping in the course book (9.1.4). Then look through the code and make sure you understand what it does. Especially note the view and projection matrices for the light source, since we will need those when rendering the shadow map. This is similar to the TV camera in the previous tutorial; if necessary, revisit that.



## Creating the shadow map

A shadow map is simply a texture containing a depth buffer rendered from the light's point of view. We'll need to do a render-to-texture pass to get this, much like the previous lab. Before we get there, we'll render the shadow map to the default frame buffer, so we get to see what it looks like.

Look at the `drawShadowMap()` method. It's rather similar to `drawScene()`, except that it uses a different shader. That shader is much simpler - for instance, it doesn't perform any lighting.

For now, comment out the call to `drawScene()` in `display()`:

```
//drawScene(viewMatrix, projMatrix, lightViewMatrix, lightProjMatrix);
```

Add instead a call to `drawShadowMap()` at the beginning of `display()`, just after we've computed the light's view and projection matrices:

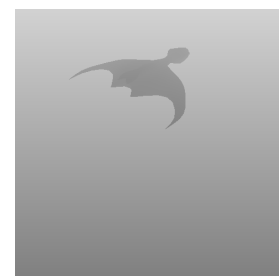
```
float4x4 lightProjMatrix = perspectiveMatrix(45.0f, 1.0, 5.0f, 100.0f);
drawShadowMap(lightViewMatrix, lightProjMatrix);
```

The current fragment shader (*simple.frag*) outputs white everywhere, so change that:

```
//fragmentColor = vec4(1.0);
fragmentColor = vec4(gl_FragCoord.z);
```

So, we're now writing the fragment's depth to all the color channels (as well as the depth buffer). Run the program now and make sure that you're seeing something similar to the picture on the right.

Here, a color closer to white (1.0, 1.0, 1.0) indicates a larger depth value. A larger depth value indicates that something is further away.



If everything is as it should be, undo the changes you've made to the fragment shader. Also, uncomment the call to `drawScene()` again. Run the program, and make sure it again looks like the very first picture.

Now we're ready to create the actual shadow map. For that we need a texture and a framebuffer object to render to. Create these at the end of `initGL()`, like so:

```
// Generate and bind our shadow map texture
glGenTextures( 1, &shadowMapTexture );
glBindTexture( GL_TEXTURE_2D, shadowMapTexture );

// Specify the shadow map texture's format: GL_DEPTH_COMPONENT[32] is
// for depth buffers/textures.
glTexImage2D( GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32,
             shadowMapResolution, shadowMapResolution, 0,
             GL_DEPTH_COMPONENT, GL_FLOAT, 0
           );

// We need to setup these; otherwise the texture is illegal as a
// render target.
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_EDGE );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );

// Cleanup: unbind the texture again - we're finished with it for now
glBindTexture( GL_TEXTURE_2D, 0 );

// Generate and bind our shadow map frame buffer
glGenFramebuffers( 1, &shadowMapFBO );
glBindFramebuffer( GL_FRAMEBUFFER, shadowMapFBO );

// Bind the depth texture we just created to the FBO's depth attachment
glFramebufferTexture2D( GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                      GL_TEXTURE_2D, shadowMapTexture, 0 );

// We're rendering depth only, so make sure we're not trying to access
// the color buffer by setting glDrawBuffer() and glReadBuffer() to GL_NONE
glDrawBuffer( GL_NONE );
glReadBuffer( GL_NONE );

// Cleanup: activate the default frame buffer again
glBindFramebuffer( GL_FRAMEBUFFER, 0 );
```

Don't forget to declare the necessary (global) variables:

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

Finally, we need to tell `drawShadowMap()` to render to our newly created shadow map. In the beginning of `drawShadowMap()`, add:

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

In the end of `drawShadowMap()`, add:

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

Congratulations! You now render a shadow map.

## Using the shadow map

Now that we have a shadow map, we probably want to use it. Basically, each time a fragment is drawn, we need to compare that fragment's depth to a corresponding value in the shadow map. If the current fragment is further from the light than the value in the shadow map, it's in shadow. Otherwise, it's lit.

To find the corresponding value in the shadow map, we need to transform a coordinate from the camera's view-space to the shadow map's texture space. This involves a couple of matrices:

```
glUseProgram( shaderProgram );

float4x4 lightMatrix = lightProjectionMatrix * lightViewMatrix
    * inverse(viewMatrix);
setUniformSlow( shaderProgram, "lightMatrix", lightMatrix );
```

Study the above matrix carefully. First, `inverse(viewMatrix)` transforms from (camera) view space to world space; next `lightViewMatrix` transforms from world space to light view space; and finally, `lightProjectionMatrix` transforms from light view space to light clip space. Thus, `lightMatrix` transforms a coordinate from *camera view space* to *light clip space*.

We'll use the `lightMatrix` in the *vertex* shader (*shading.vert*), where we perform the transformation and send the result to the fragment shader:

```
viewSpacePosition = (modelViewMatrix * vec4(position, 1.0)).xyz;
shadowMapCoord = lightMatrix * vec4(viewSpacePosition, 1.0);
```

Don't forget to declare the uniforms and outputs:

```
uniform mat4 modelViewProjectionMatrix;

uniform mat4 lightMatrix;
out vec4 shadowMapCoord;
```

All good? No, not quite. Now, `shadowMapCoord` is in light clip space. Light clip space has coordinates in the range  $(-1, -1, -1)$  to  $(1, 1, 1)$ . However, texture coordinates run from  $(0, 0)$  to  $(1, 1)$ . The depth value we've stored in our shadow map has the same range, i.e. between 0 and 1. So, it's convenient to rescale our coordinates here:

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

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

This scales the coordinates by 0.5 and adds 0.5; the `shadowMapCoord.w` will cancel out later when we divide by it.

Time to implement the actual shadow map lookup. This we do in the *fragment* shader (*shading.frag*). First, make sure that you have declared the necessary inputs and uniforms:

```
out vec4 fragmentColor;

in vec4 shadowMapCoord;
uniform sampler2D shadowMapTex;
```

Next, we need to sample the depth value from the shadow map at the correct location. Then, we compare the depth value from the shadow map with the fragment's depth. If the depth from the

shadow map is larger than the fragment's depth (i.e., the fragment is closer), then the fragment is lit. Otherwise, it's shadowed.

To do this, replace the assignment to `fragmentColor` with the following code:

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

float depth = texture( shadowMapTex, shadowMapCoord.xy / shadowMapCoord.w ).x;
float visibility = (depth >= (shadowMapCoord.z/shadowMapCoord.w)) ? 1.0 : 0.0;

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

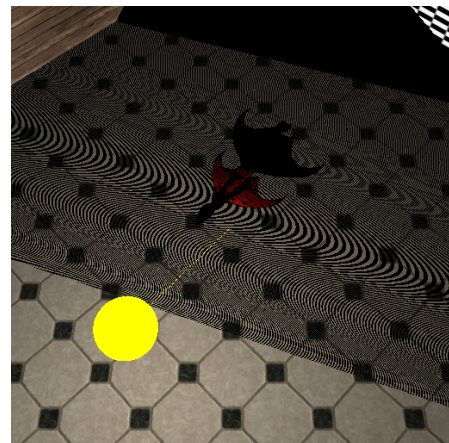
Before we can see the results of our (hard) work, we need to bind the resources our shaders use in `drawScene()`:

```
setUniformSlow(shaderProgram, "viewSpaceLightPosition", viewSpaceLightPos);

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

Now, run the program. It should look like something in the image to the right. There are some shadows, but we can also see some things that aren't quite right.

For one, there are some rather ugly patterns on the floor. The effect you're seeing is commonly referred to as "surface acne", and it's one of the problems that shadow maps commonly have. The following happens: A fragment on the floor that is not in shadow, looks up the corresponding depth in the shadow map. That depth is *almost* the same as the fragment's depth (ideally, it would be the same!). So sometimes our comparison will return that the fragment is closer and sometimes that it's further than the depth from the shadow map. The problem is discussed in the course book; especially look for Figure 9.17 there.



Secondly, you should see a completely black region opposite to the light source. Our shadow map is limited in size, and we haven't really tried to handle the case where we try to sample outside of the shadow map.

## Fixing the problems

There are a couple of ways to 'fix' the first problem, surface acne. A simple (but far from perfect) way to do this is to simply "push" the polygons a tiny bit away from the light source when drawing the shadow map. In fact, OpenGL has some built-in functionality to help us with this. In the beginning (before anything is drawn) of `drawShadowMap()`, add:

```
glClear( GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT );

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

After drawing everything, disable the polygon offset again:

```
glUseProgram( currentProgram );

glDisable( GL_POLYGON_OFFSET_FILL );
```

We start attacking the second problem by changing the clamping mode of the shadow map texture. Earlier, we set it to `GL_CLAMP_TO_EDGE`. So, when we sample outside of the shadow map, the shadow map lookup returns the value of the texel closest to the requested point. In this case, that's not a very useful value.

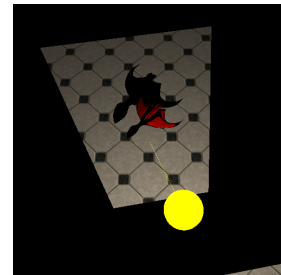
First, let's change the clamping mode for `shadowMapTexture` in `initGL()`:

```
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_EDGE );
//glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );

float4 zeros = {0.0f, 0.0f, 0.0f, 0.0f};
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER );
glTexParameterfv( GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, &zeros.x );
```

Now, whenever we attempt to sample outside of the shadow map, our lookup returns 0.0. This causes any fragments that ask for such values to be in shadow.

Run the program and make sure it looks similar to the picture on the right. There's still a unshadowed stripe where there shouldn't be one, and all in all, the hard rectangular shadow border doesn't look too good.



Our fundamental problem is that we are rendering a single, rather directional shadow map. But, currently, our light is omni-directional. For this lab, we'll fix this by turning our light into a directional spot light. (The other option would be to extend the shadow map to be omni-directional, by e.g., turning it into a shadow *cube* map.)

A spot light only casts light inside a limited cone. That cone is defined by a direction (the light's direction) and an angle. To see whether a fragment is lit by a spot light, we need to compute the angle between the vector connecting the light to the fragment and the light's direction. This angle we then compare to the spot light's cone's opening angle.

In the fragment shader (*shading.frag*), add the following code:

```
float visibility = (...) ? 1.0 : 0.0;

float angle = dot(posToLight, -viewSpaceLightDir);
float spotAttenuation = (angle > spotOpeningAngle) ? 1.0 : 0.0;

//fragmentColor = vec4(diffuseColor * diffuseReflectance * visibility, 1.0);

float attenuation = diffuseReflectance * visibility * spotAttenuation;
fragmentColor = vec4(diffuseColor * attenuation, 1.0);
```

Declare the necessary uniforms:

```
uniform sampler2D shadowMapTex;

uniform vec3 viewSpaceLightDir;
uniform float spotOpeningAngle;
```

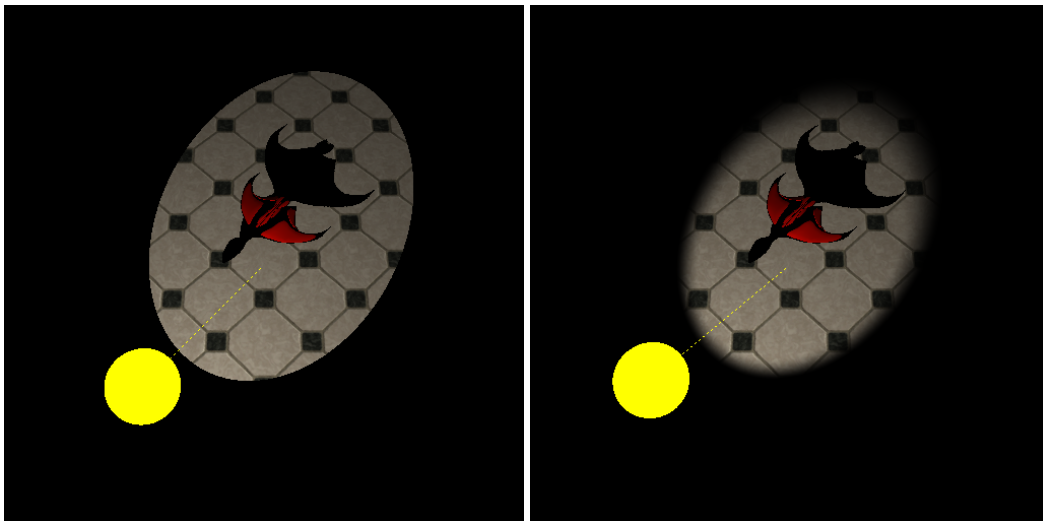
We need to set a value for `spotOpeningAngle` in `drawScene()`. What's an appropriate value then? Recall that the dot-product of two normalized vectors returns the cosine of the angle between them. So, for an opening angle of  $20^\circ$ , we want to set `spotOpeningAngle` to  $\cos 20^\circ$ . Do this in `drawScene()` (additionally, we compute and set `viewSpaceLightDir`):

```
glBindTexture( GL_TEXTURE_2D, shadowMapTexture );

float3 viewSpaceLightDir = transformDirection(
    viewMatrix, -normalize(lightPosition) );
setUniformSlow( shaderProgram, "viewSpaceLightDir", viewSpaceLightDir );

setUniformSlow( shaderProgram, "spotOpeningAngle",
    std::cos(20.0f*3.1415f/180.0f) );
```

Run the program. You should see something like the picture shown below, to the left. The hard border isn't that nice - we'd prefer a soft transition like in the image to the right.



This is achieved by defining two angles, the *inner* and the *outer* angle. Everything outside of the outer angle is never lit by the spot light; everything inside the inner angle is potentially lit fully. For samples between those two angles, we'd like to smoothly interpolate, which gives us the nice soft and fuzzy transition shown above.

For the interpolation, we'll use the GLSL built-in function `smoothstep()`. The function takes three arguments, a lower bound, an upper bound, and a value. If the value is outside of the lower bound, `smoothstep()` returns zero. If the value is outside of the upper bound, `smoothstep()` returns one. Otherwise, it returns a smoothly varying value between zero and one. Check the GLSL documentation for a more exact definition!

Anyway, in the fragment shader, change the following:

```
float angle = dot(posToLight, -viewSpaceLightDir);

//float spotAttenuation = (angle > spotOpeningAngle) ? 1.0 : 0.0;

float spotAttenuation = smoothstep( spotOuterAngle, spotInnerAngle, angle );
```

You'll need to declare `spotOuterAngle` and `spotInnerAngle` as uniforms, and set their values in `drawScene()`. At this point, you should know how to do this, so you get to do this by yourself. Find appropriate values for the inner and outer angle – try to maximize the spot light's "size", but avoid making the hard borders from the shadow map visible.



Also, `spotOpeningAngle` is no longer used, so remove any references to it.

## Hardware Support and other improvements

Now that we've fixed the most glaring issues with our shadow map, let's instead focus on some improvements.

**Re-scaling** Recall the vertex shader. We adjust the shadow map coordinates by scaling with 0.5 and translating with 0.5; we need to do this to transform the coordinates from their original  $(-1, -1, -1) \rightarrow (1, 1, 1)$  range to the  $(0, 0, 0) \rightarrow (1, 1, 1)$  range that we need for the computations in the fragment shader.

This transformation could easily be included in the `lightMatrix`. After all, a  $4 \times 4$  matrix can represent scaling and translation.

Implement this in `lab6_main.cpp`, where you construct the `lightMatrix`. Then remove the old code from the vertex shader; your goal here is to reduce the vertex shader work to compute `shadowMapCoord` to a single matrix-vector multiplication.

*Hint:* Use the matrix construction functions `make_scale<float4x4>()` and `make_translation()`.

**Hardware shadow map lookup** Shadow mapping is common enough to warrant special support from OpenGL, and potentially from our GPU hardware. Right now, we fetch the depth from the shadow map and then perform the comparison manually. However, there are built-in functions that can help you with this in OpenGL and GLSL. Let's use these!

First, we need to change the way we set up our shadow map. In `initGL()` add the following lines:

```
glTexParameterfv( GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, &zeros.x );

glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
    GL_COMPARE_REF_TO_TEXTURE );
```

In the fragment shader, we need to do a few small adjustments too. First, we need to change the type of texture sampler we're going to use:



```
//uniform sampler2D shadowMapTex;
uniform sampler2DShadow shadowMapTex;
```

Next, instead of fetching a value with `texture()`, we're going to use (an overload of) `textureProj()`:

```
//float depth= texture( shadowMapTex, shadowMapCoord.xy/shadowMapCoord.w ).r;
//float visibility= (depth>=(shadowMapCoord.z/shadowMapCoord.w)) ? 1.0 : 0.0;

float visibility = textureProj( shadowMapTex, shadowMapCoord );
```

Visit the following two links for documentation on the changes that we've introduced:

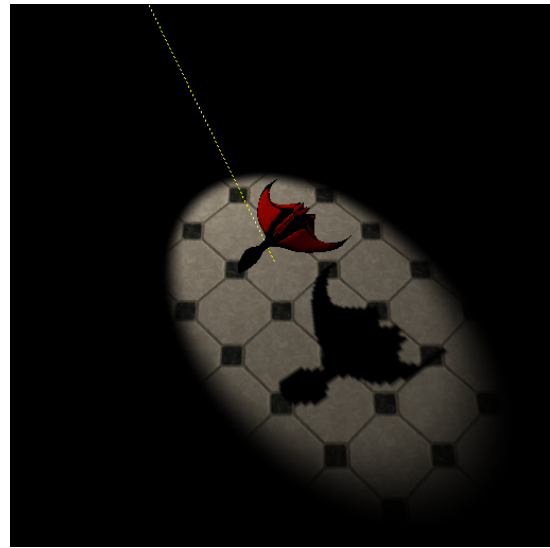
-  <http://www.opengl.org/sdk/docs/man/xhtml/glTexParameter.xml>
-  <http://www.opengl.org/sdk/docs/manglsl/xhtml/textureProj.xml>

First, identify the relevant sections in the documentation (e.g., which overload of `textureProj` did we use? Hint: look at the types of the function arguments.). Next, make sure you understand what each of the above changes is doing - you might be asked to explain this when you present your results to the assistants.

**Percentage Closer Filtering** First, make the shadow map smaller. For instance,  $128 \times 128$  is a good choice for this. Run the program. You shadows should now look blocky and bad. Since this ain't minecraft, we'd like to avoid such things and instead aim for a smooth look.

Percentage Closer Filter (*PCF*) will reduce this problem somewhat with very few changes. PCF is also supported by most hardware these days, so enabling it is quite cheap.

Change the texture filtering mode for the shadow map from `GL_NEAREST` to `GL_LINEAR`, and run the program again. Make sure it looks similar to the picture on the right. If necessary, zoom in and look at the shadows close up to see the PCF in action.



Normally, interpolating depth values doesn't make much sense. So, something magic happens when you enable `GL_LINEAR` on a texture in combination with a `GL_TEXTURE_COMPARE_MODE` set to `GL_COMPARE_REF_TO_TEXTURE`. Read Section 8.23.1 in the OpenGL 4.4 Specification:

 <http://www.opengl.org/registry/doc/glspec44.core.pdf>

Can you make sense of that? PCF is further described in the course book, in Section 9.1.4 (page 361++), and at e.g.

 [http://http.developer.nvidia.com/GPUGems/gpugems\\_ch11.html](http://http.developer.nvidia.com/GPUGems/gpugems_ch11.html)

Read the section on PCF in the book and/or the link provided, to get an idea on what's going on.

**When you're done with all elements in this lab, show your results to one of the assistants. Have the finished program running and the source code ready. Your friendly neighbourhood assistant may want to ask you some questions and/or ask you to show some things in your code, so be prepared to do so.**