Shadow mapping

Practical Assignment

Introduction

Principle

This practical aims at understanding how shadow mapping works using modern OpenGL.

The key idea behind shadow mapping is to render our scene from the point of view of a light source in order to accurately estimate where the light reaches. Points visible from the location of the light are considered lighted and those that are not visible are considered in shadow. Therefore, in this practical you will be required to produce code to create a shadow map by rendering the scene to a texture, and then use this texture to perform the shadow test.

Exercise 1 : Multiple viewpoints

The first step towards implementing shadow mapping is having the ability to store and switch between different viewpoints for rendering the shadow map and the main camera image. The code included with this practical contains the useful `Camera` structure for this, which is used for the main scene camera. Your first task is to add a second camera to the program. You will need to modify the keyboard handler function to select which camera is active with the 1 or 2 keyboard buttons. This camera should be used for rendering and should be updated with the `updateCamera` function. Additionally, use the inactive camera as the light source for rendering by sending its position to the shader program as a uniform.

Exercise 2 : Render to texture

Now that you are comfortable using different cameras and viewpoints, it ?s time to create the shadow map. To do this, it is necessary to render the scene directly to a texture instead of the window. This requires you to create 2 OpenGL objects : a *framebuffer*, which is an object used to render, and a texture that will store the result of the render call.

A framebuffer is created using the following OpenGL function, which designates an id to refer to it :

```
GLuint framebuffer;
glGenFramebuffers(1, &framebuffer);
```

And should be released before program termination :

```
glDeleteFramebuffers(1, &framebuffer);
```

The framebuffer stores OpenGL internal state structures used for rendering, but it does not store the final rendering result. For that, it is necessary to create a texture :

```
GLuint texShadow;
const int SHADOWTEX_WIDTH  = 1024;
const int SHADOWTEX_HEIGHT = 1024;

glGenTextures(1, &texShadow);
glBindTexture(GL_TEXTURE_2D, texShadow);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32F, SHADOWTEX_WIDTH, SHADOWTEX_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, nullptr);
```

The preceding code creates a 2D texture of size 1024x1024 that is specified to store depth values. This kind of texture is used by OpenGL as depth buffers (also called z-buffers), and a shadow map is essentially the z-buffer of a rendering of the scene from the point of view of the light.

Once a framebuffer and a depth texture have been created, we need to tell OpenGL that the texture will be used with this framebuffer. We can do this via the following OpenGL call :

```
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, texShadow, 0);
```

Framebuffers are used by binding them before rendering. If we use the id 0 instead of the framebuffer we have created, the default framebuffer is bound, which will be the one used for rendering to the main window. Therefore, the typical structure of a program that performs shadow mapping looks like :

```
while (!glfwWindowShouldClose(window)) {

    // ... Process input / Update camera and other variables ...

    glUseProgram(shadowProgram); // Program used for rendering the shadow map

    glBindFramebuffer(GL_FRAMEBUFFER, framebuffer); // Bind your framebuffer

    glClearDepth(1.0f);
    glClear(GL_DEPTH_BUFFER_BIT); // Clear the depth buffer

    glViewport(0, 0, SHADOWTEX_WIDTH, SHADOWTEX_HEIGHT); // Set viewport size to size of the texure in the framebuffer

    // ... Set neccesary uniforms ...

    // ... Execute draw command ...

    glUseProgram(mainProgram);

    glBindFramebuffer(GL_FRAMEBUFFER, 0); // Bind default framebuffer

    glClearDepth(1.0f);
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear the color and depth buffer

    glViewport(0, 0, WIDTH, HEIGHT); // Set viewport size to size of the window

    // ... Set neccesary uniforms ...

    // ... Execute draw command ...
}
```

The texture that is attached to your framebuffer can be used as an input texture (a sampler2D uniform) in your main rendering program. The following

code does this assuming your sampler2D is called texShadow in your shader code :

```
// Bind the texture to slot 0
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texShadow);
glUniform1i(glGetUniformLocation(mainProgram, "texShadow"), 0);
```

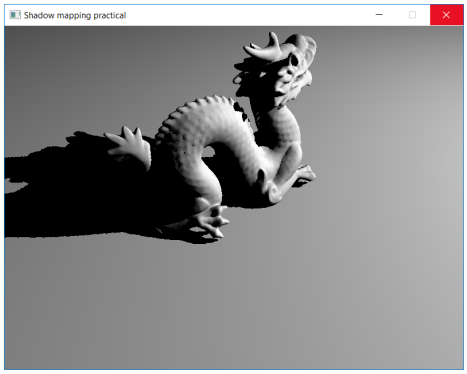
Finally, in order to sample the corresponding texel in your shadow map, you need to map the point you are shading in your fragment shader to the shadow map coordinates. To do this, your program needs the same transorm matrix (mvp) that was used to render the shadow map. If we assume you pass this to your main program as uniform lightMVP, then the following code retrieves the depth value in the shadow map :

```
vec4 fragLightCoord = lightMVP * vec4(fragPos, 1.0);

// Divide by w because fragLightCoord are homogenous coordinates
fragLightCoord.xyz /= fragLightCoord.w;

// The resulting value is in NDC space (-1 to 1),
//  we transform them to texture space (0 to 1)
fragLightCoord.xyz = fragLightCoord.xyz * 0.5 + 0.5;

// Depth of the fragment with respect to the light
float fragLightDepth = fragLightCoord.z;

// Shadow map coordinate corresponding to this fragment
vec2 shadowMapCoord  = fragLightCoord.xy;

// Shadow map value from the corresponding shadow map position
float shadowMapDepth = texture(texShadow, shadowMapCoord).x;
```

As you can see in the code above, the first step is transforming the fragment world space position to the light space position by using the `lightMVP` matrix. As seen in the course, this matrix transforms the point to normalize device coordinates, which go from -1 to 1. Since the texture coordinates and the depth need to be expressed from 0 to 1, we perform a scale by 0.5 and a shift by 0.5 to map them to the correct range. Finally, the `fragLightCoord` variable contains the required shadow map coordinates in the xy fields and the depth in the z field, which you will need to test against the depth that was stored in the shadow map in order to find out if the fragment is in light or in shadow.

The files included with this practical include shader files (`shadow.frag` and textttshadow.vert) that you can use to create a shader program to render the shadow map. If you look at the fragment shader, you will see that there is no color output at all! This is because the shadow map is simply the depth buffer (or z-buffer) used for rendering, and those values are saved automatically in the texture set as depth buffer in your framebuffer.

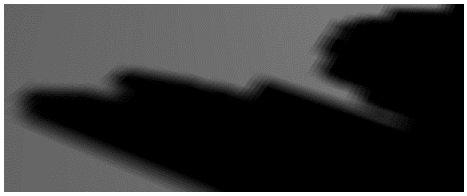
At this point you have all the information to perform the shadow test. Remember to add a small bias value to avoid self-shadowing. If you set all values that do not pass the shadow test to 0, then you should obtain a result close to the following :



Exercise 3 : Softer shadows

If you look closely at the shadows created in the previous exercise you should notice that they can be somewhat blocky, especially if the light position creates very large shadows :

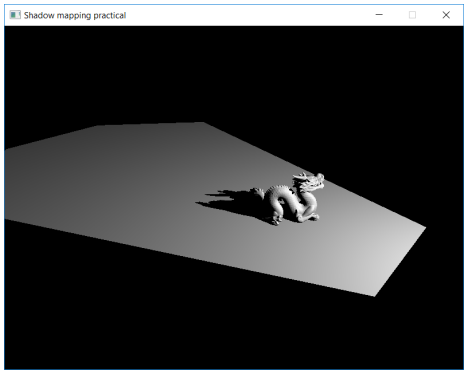
One way to make this better is to perform the shadow map not only in the corresponding shadow map position, but to perform the test for several samples around that point and average the results. This procedure is called Percentage-Closer Filtering (PCF) and results in a smoother and more natural shadow look :



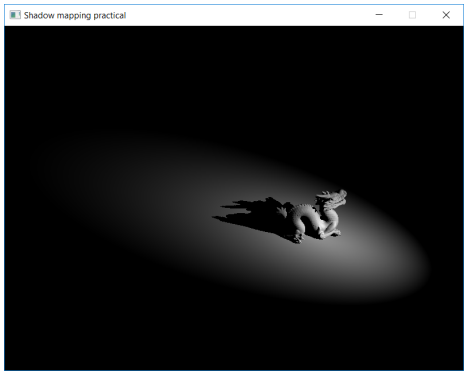
Try different amount of samples and larger sampling areas to get a better feeling of the possible results and the performance limits of this approach.

Exercise 4 : Spotlights

Up to now, we have not considered what happens when the fragment maps to a texture coordinate in the shadow map that is invalid (outside the range of 0 to 1). If we simply retain the original light values in this case, there will be no shadows outside the light frustum. On the other hand, simply setting lighting to 0 in that case would result in hard shadow boundaries outside the light frustum :



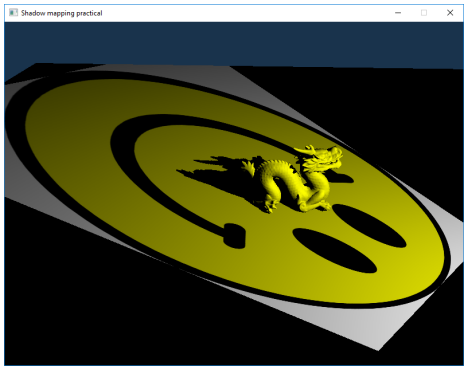
Instead, we can create a more natural spotlight lighting by gradually dimming the light as the sampled shadow map coordinate gets closer to the shadow map border. One way of doing this is measuring the distance from the coordinate to the center (0.5, 0.5) and setting a linear light multiplier that is 1 at the center coordinates and 0 when the distance is 0.5 or above. That would result in the following look :



Additionally raising this dimming value to a small power (e.g. 0.5) will create more defined boundaries. Experiment with different dimming functions and change the field of view of the light camera to see how it affects the final image.

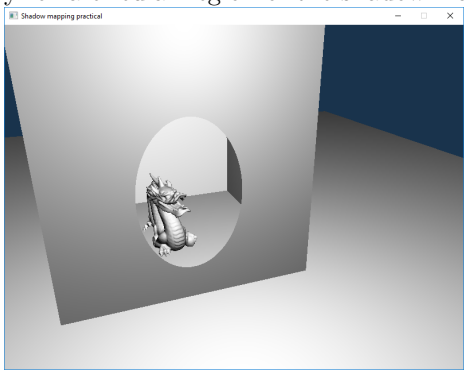
Exercise 5 : Colored lights

One way of creating colored lights is to define a texture that we will use to retrieve a color, the same way we do with the shadow map. To help you do this, a texture is created in the code included with this exercise (called `texLight`). You can set it as an extra sampler in your shader program and set the uniform values as is done with the shadow map texture. You can then sample the colors with the same texture coordinates, or maybe you want to modify the coordinates to create an interesting effect. The following is an example of a simple mapping of the light color texture :



Exercise 6 : Peeling with shadow maps

A shadow map can be used for other purposes besides shadows. In this exercise your task is to use the shadow test to peel away the first layer of objects that the light touches. To do this, you can run the normal shadow test, and discard all fragments that are visible by the light. Discarding a fragment is easy : in the fragment shader you can call the `discard` command and that will prevent the current fragment from being written to the final image. We have also included a second scene file (sceneWithBox.obj) in order for you to try this. The following is an example of the result when we discard the visible fragments from a light, but only for a circular region of the shadow map :



You can also use the light color texture from the previous exercise to further limit where the peel takes place, and create interesting effects.