Programming Assignment 5: Shadow Mapping

MIT 6.837 Computer Graphics

Fall 2020

Due November 18 at 8pm, Boston Time

1 Getting Started

The goal of this assignment is to implement shadow mapping[1], a popular algorithm for rendering shadows in real time. The assignment is implemented in OpenGL. This handout contains step-by-step instructions for how to draw the meshes, how to apply textures in OpenGL, and how to cast shadows. Most of the assignment will be implemented in C++, with a few parts written in GLSL.

Please start as early as possible! OpenGL programming is error-prone and debugging these programs can be very time-consuming for newcomers.

Please also note that your **final project proposal** is due on **November 11 at 8pm**.

The sample solution is included in the starter code distribution. Look at the `sample_solution` directory for Linux, Mac, and Windows binaries. You may need to change the file mask on the Linux/Mac binaries, e.g., by executing the following command in your terminal:

```
chmod a+x sample_solution/linux/assignment5
```

To run the sample solution, execute

```
sample_solution/linux/assignment5
```

You will see a Stanford dragon sitting in the "sponza" scene with textures and shadows.

The starter code for this assignment is in the `assignment_code/assignment5` directory. You will also need to **read through, understand, and modify** files in `gloo/gl_wrapper`, `gloo/shaders`, and `gloo/Renderer.*`.

To build the starter code, follow the same steps as in previous assignments. For instance, on MacOS or Linux, execute the following:

[1] https://en.wikipedia.org/wiki/Shadow_mapping

```
mkdir build
cd build
cmake ..
make
```

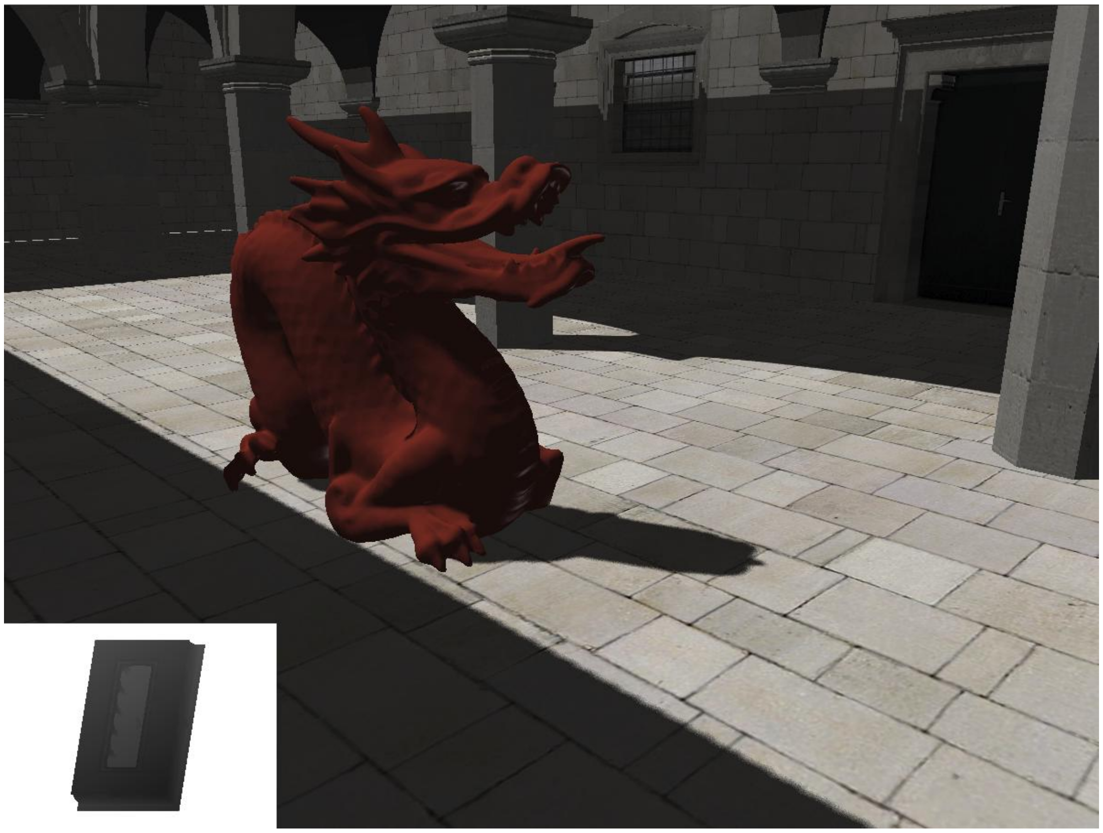

Figure 1: Sample solution result. On the lower left is the shadow map (depth) visualized.

Note that in this assignment you will need to work closely with the OpenGL APIs. In the past assignment, we have been indirectly calling those APIs via GLOO, but this time you will have to complete new C++ classes that directly call OpenGL APIs. It is not necessary for this assignment, but if you are curious how the GLOO interfaces OpenGL in the previous assignments, checkout `gloo/gl_wrapper/*`.

Before you start, here are a few tips on working with OpenGL:

1. Note that OpenGL API calls may fail silently if you do not explicitly check the return values. Therefore, it is highly recommended to wrap all OpenGL calls with no return value in the `GL_CHECK(...)` macro (defined in `gloo/utils.hpp`) for OpenGL calls to fail loudly (i.e. it will print the error to the console).
2. You may find this OpenGL API reference card useful:

   https://www.khronos.org/files/opengl43-quick-reference-card.pdf
3. Another good source of OpenGL is the OpenGL official wiki: https://www.khronos.org/opengl/wiki/. For an overview of OpenGL APIs, this page is worth checking out: https://www.khronos.org/opengl/wiki/Category:Core_API_Reference.

4. Quick explanations to OpenGL APIs can be found on

   https://www.khronos.org/registry/OpenGL-Refpages/gl4/index.php

5. Later in this assignment you will need to write OpenGL shaders in GLSL and load them in C++. Hopefully you have had some practice with shaders the in assignment 0. A good tutorial on that is https://learnopengl.com/Getting-started/Shaders, where both GLSL and relevant OpenGL APIs are covered. In this assignment, GLOO has wrapped some of the OpenGL API calls in object-oriented classes that are more friendly.

2 Summary of Requirements

In this assignment you will need to implement the following items with increasing difficulty:

2.1 Setting up the scene (10% of grade)

Your first task is to create a scene tree (as always) and render the meshes without textures.

2.2 Texture mapping (30% of grade)

So far, your objects only have uniform materials (e.g. appearance that does not vary over the surface of the object). Spatially-varying materials are more interesting, so in this part of the assignment, you will

- Create OpenGL texture objects.
- Specify which texture to use when drawing.
- Modify the shader code to sample from the texture using texture (UV) coordinates.

2.3 Shadow mapping (60% of grade)

Until now, objects in the scene do not cast shadows. One of the most prevalent techniques to render shadows in real-time is *shadow mapping*. You will implement shadow mapping in this step. Along the way, you will

- Learn how to render to a texture in an off-screen framebuffer object (FBO), rather than straight to the screen.
- Render a depth map to an off-screen FBO, from the camera's point of view.
- Let OpenGL read from this depth map in the next draw call, so that you know which pixels are lit and which are not.
- Implement the shadow mapping algorithm as part of the fragment shader.

3 Setting up the scene (10% of grade)

The first step is to load the geometry and get a textureless rendering of the scene. You will need to fill in the `ShadowViewerApp::SetupScene` function. Note that a slightly different set of GLOO API will be used in this assignment compared to previous ones. At a high level, you will need to make use of `MeshData` which contains a group of meshes. Specifically,

1. We have created a `PhongShader` instance for you to shade the surface colors. This is the same shader used in assignment 0-3. Later on you will need to modify this shader to allow shadowing. For now, just create a shared pointer holding a `PhongShader` without modification.
2. A shared `VertexObject` instance needs to be created to hold the geometry data for all the meshes. You can later pass that to the `SceneNode`s.
3. Most importantly, this time you will make use of the `MeshData` object `mesh_data` returned by the call to `MeshLoader::Import`. Note that it has multiple `MeshGroup`s, each group with its own vertices, indices, and materials. That means you will need to create a `SceneNode` for each element (a `MeshGroup`) of the `mesh_data.groups` array. More details on `MeshData` can be found in `gloo/MeshData.hpp`.

Note that for each `MeshGroup`'s scene node, you will need to specify its `ShadingComponent` (using the shared `PhongShader`), `RenderingComponent` (using the shared `VertexObject`), and `MaterialComponent`.

The `RenderComponent` is a *segment* of the `VertexObject` of the scene, which means the `VertexObject` is **split into a few segments of indices**, each belonging to a `MeshGroup`. You can pass the shared `VertexObject` to the `RenderingComponent` of every `SceneNode`, and specify the **starting face index** and **number of indices** (see `gloo/MeshData.hpp`) using a call to `RenderingComponent::SetDrawRange`. These two quantities can be obtained from `start_face_index` and `num_indices` fields of a `MeshGroup`. Recall that the number of indices is three times the number of faces in a `MeshGroup`.

After finishing `ShadowViewerApp::SetupScene`, you should see a texture-less mesh with mostly Lambertian reflectance as shown in Figure 2.

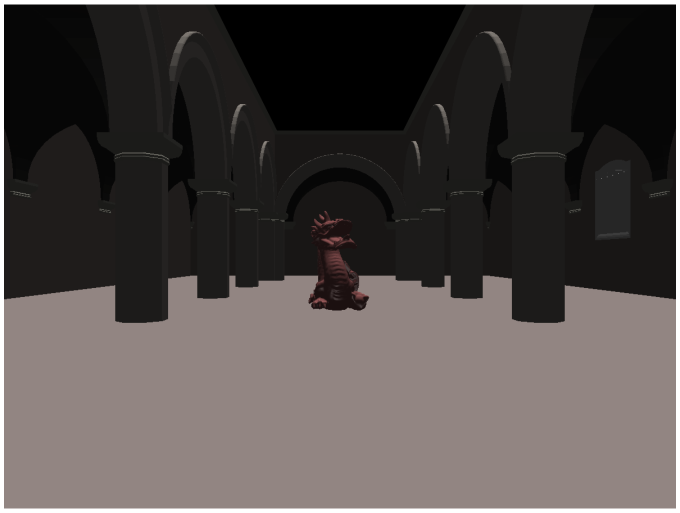

Figure 2: Texture-less rendering.

4 Adding Textures (30% of grade)

Now we are ready to apply textures to the meshes. You will need to create OpenGL texture objects and read the diffuse material color from the texture, rather than per-object uniform colors as we did in previous assignments. Note that GLOO will handle per-vertex texture coordinates (i.e. UV coordinates) for you.

4.1 The `GLOO::Texture` class

We have prepared a mostly-ready-to-use wrapper class `GLOO::Textures` of OpenGL textures for you, in files `gloo/gl_wrapper/Texture.*`. Please read through the implementation of the class to have a basic understanding of existing code, since you need to fill in a few lines of code there.

Just a bit of background: creating a texture in OpenGL requires several steps (click on the OpenGL APIs to jump to the documentation):

1. Generate a new *texture handle* using `glGenTextures(1, &handle)`.
2. *Bind* the new handle to make it active `glBindTexture(GL_TEXTURE_2D, handle)`.
3. Allocate storage for the texture currently bound as `GL_TEXTURE_2D` and upload pixel data using `glTexImage2D(GL_TEXTURE_2D, 0, <data_format>, <width>, <height>, <border>, <format>, <type>, <data>)`. The OpenGL online reference explains all these parameters in detail. The `<data>` parameter must be a pointer to an array of pixel values (see `Texture::UpdateImage`), or it can be a `nullptr` if we want to reserve space and later write to this texture (see, e.g., `Texture::Reserve`).
4. Configure the texture minification filter using `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)`. This enables basic bilinear filtering.

We have implemented most of these steps for you with the exception of `Texture::BindToUnit(int id)`. **You will need to implement that on your own by calling two OpenGL APIs**. Specifically, you will have to first activate the texture unit `id` with `glActiveTexture`. Note that the texture unit id in OpenGL API starts with `GL_TEXTURE0`, a constant that is not zero, and you will see further hints on this in the starter code. After the texture unit is activated, you can bind a (2D) texture handle to it, using `glBindTexture`. Please figure out the arguments to this API call (you will need to check out what is happening in `Texture::Initialize`).

4.2 Upgrade `PhongShader` class

The current `PhongShader` implementation shades each mesh with a uniform color. Now it is the time to upgrade your `PhongShader::SetTargetNode` function so that the `GLOO::Textures` are correctly bound to the OpenGL texture units and used in the fragment shaders. As you can see in the `gloo/parsers/ObjParser.cpp` file, we have already handled texture content loading for you.

The relevant source files are `gloo/shaders/glsl/phong.frag`, `gloo/shaders/glsl/phong.vert`, and `gloo/shaders/PhongShader.cpp`. You will need to modify the `gloo/shaders/glsl/phong.frag` file in the beginning of this part, and the vertex shader when you later implement shadow mapping.

In the fragment shader, you need to create one `sampler2D` for each of ambient, diffuse, specular texture. **In the assignment only the diffuse texture is used**, but it doesn't harm to create a slightly more general version. Note that you will need to work with multiple textures (diffuse texture + shadow map texture) later in this assignment anyway.

In the original fragment shader `phong.frag`, the `Get[Ambient/Diffuse/Specular]Color` functions directly load the corresponding uniform color from the material; that is, `material.diffuse` for `GetDiffuseColor`. In order to support texturing, you will need to call the `texture` function to sample a pixel from the right texture unit. Please implement that for all `Ambient/Diffuse/Specular` components. It is also recommended, but not required, to add boolean texture/plain color flags to corresponding shading component, such as `diffuse_use_texture`, so that you can turn on/off texture shading from the C++ side in `PhongShader`.

After you are done with the fragment shader, it's time to work on the `PhongShader::SetTargetNode` function, where you will make use of the `Texture::BindToUnit` function you just implemented to bind the textures to OpenGL texture units. Note that you also need to set the `sampler2D` uniform variables in your shader program using `SetUniform`: if you are binding a `Texture` to OpenGL texture unit i, then the value of the `sample2D` variable should also be i (as an integer).

After you are done with this part, you will see a textured scene as shown in Fig. 3

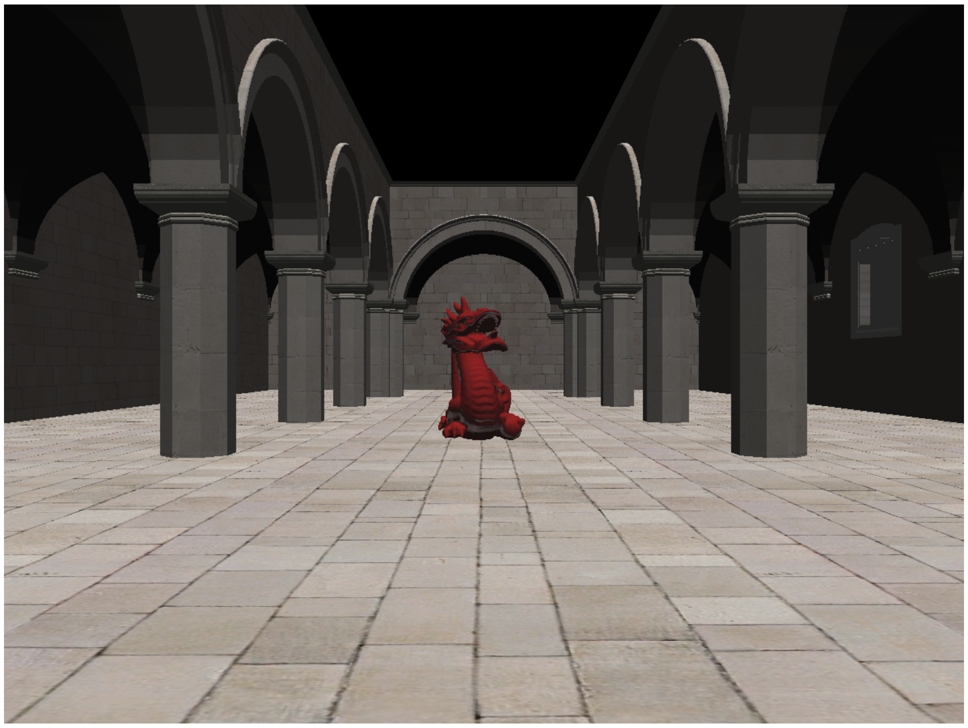
Figure 3: Textured scene.

5 Shadow Mapping (60% of grade)

Finally, we have all we need for implementing shadow mapping. Shadow mapping is an example of a multi-pass algorithm in computer graphics: In the first pass, we render to a depth texture with the light source position being the camera location. Imagine that the light source is now the "camera" and for each pixel we are computing the distance ("depth") from the camera to the object. We are using directional light in this assignment. In the second pass, we use the depth value stored in this texture to determine the light source visibility of scene points.

By default, OpenGL renders directly to the screen. With a bit of additional configuration, we can make OpenGL render to a texture instead. To be precise, we will create one OpenGL framebuffer object (FBO[2]) that contains a texture for the shadow map depth.

5.1 The `GLOO::Framebuffer` class

For shadow mapping, you will first need to render into a framebuffer object instead of directly to the screen. We have created the class `GLOO::Framebuffer` as a object-oriented wrapping of the original C-style OpenGL frame buffer APIs. Please take a look at the **Framebuffer Objects** section of the OpenGL API reference card.

Please go through the implementation of the class, since you will use it later. **A warm-up task** for you is to fill in `Framebuffer::AssociateTexture` with an OpenGL API call `glFramebufferTexture2D`. You will need to figure out the arguments on your own.

5.2 The `ShadowShader` class

In this part you will make use of a pair of shaders (vertex + fragment) to render the shadow map to an off-screen framebuffer that you should create on your own. The shadow should be rendered from the light (sun) position, so a world-to-light NDC[3] matrix needs to be passed to the vertex shader. The shadow fragment shader may simply return a constant color – the depth will be recorded automatically.

The shaders for the shadow map is easy, so we have created them for you (`gloo/shaders/glsl/shadow.*`). Please take a look at these files to understand their logic. Most importantly, think about how the `vertex_position` vector is transformed by a series of matrices. In the shadow map fragment shader, we do not really care about the color output. This is because we only care about the depth output of this shading pass. Your job is to create the `ShadowShader` class that links the vertex and fragment shaders into a shader program. Checkout `SimpleShader` if you need some hint here (it should be simpler than `SimpleShader`).

After you are done with `ShadowShader`, take a look at the `Renderer` class, which controls how shader programs are used in rendering. You will find a few direct OpenGL API calls - try to figure their meaning out from the OpenGL references.

In the `Renderer` class, you need to initialize the `shadow_depth_tex_` object with resolution 4096×4096, as indicated by the constants `kShadowWidth` and `kShadowHeight`. Make sure you reserve its storage using the correct arguments:

[2]https://www.khronos.org/opengl/wiki/Framebuffer_Object

[3]Normalized device coordinates: https://learnopengl.com/Getting-started/Coordinate-Systems

```
shadow_depth_tex_->Reserve(GL_DEPTH_COMPONENT, kShadowWidth, kShadowHeight, GL_DEPTH_COMPONENT, GL_FLOAT);
```

Next create a `Framebuffer` instance. Using `Framebuffer::Bind` and `Framebuffer::AssociateTexture`, you can direct OpenGL to output to the texture associated with the current framebuffer, instead of to the default framebuffer (a.k.a. "screen"). Since we only care about the depth channel, use `GL_DEPTH_ATTACHMENT` as the `attachment` argument for `Framebuffer::AssociateTexture`.

It's recommended to encapsulate the shadow map rendering logic into a standalone function called `Renderer::RenderShadow`. There you will need to figure out matrices (uniform variables) used in the shadow map shaders (`model_matrix` and `world_to_light_ndc_matrix`), and correctly assign values to them using `ShaderProgram::SetUniform`. You will find the following OpenGL commands useful to set up the shadow map rendering context:

```
GL_CHECK(glViewport(0, 0, kShadowWidth, kShadowHeight));
GL_CHECK(glDepthMask(GL_TRUE));
GL_CHECK(glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE));
GL_CHECK(glClear(GL_DEPTH_BUFFER_BIT));
```

Again, before you execute the shader, remember to bind the shadow framebuffer, so that you actually render to the shadow map instead of your screen. Don't forget to unbind the framebuffer. You can make use of `BindGuard` that uses RAII to ensure framebuffer unbinding is called on exit of the current scope. Please also reset your viewport (`glViewport`) to the dimensions of your window (`application_.GetWindowSize()`) after rendering the shadow map.

After implementing and calling your `Renderer::RenderShadow` properly in `Renderer::RenderScene` (see the TODOs there in the comments), the shadow map should be rendered into the shadow texture.

The implementation of `Renderer::RenderScene` can be general enough to render more than one shadow maps, one per (directional) light. However, for this assignment, there is only one directional light that needs shadow mapping. For the ambient light source, no treatment is needed for shadowing.

To help you debug the shadow map, the starter code comes with a functionality to plot the depth texture at the lower-left corner of the window. To enable that after your shadow maps passes are implemented, call `Renderer::DebugShadowMap` after your scene is rendered. Note that the method may crash if you do not initialize related class members properly. **Please enable the lower-left shadow mapping window when submitting your program, since TAs will inspect that for grading.**

5.3 Casting shadows in the Phong shader

Now we are at the final step: use the shadow map (shadow texture) we just now rendered to cast shadows. The high-level idea is simple: just pass the shadow map as a texture to the Phong shader, and then for each pixel sample the correct location in the shadow map. Think about how to evaluate the corresponding shadow position given the world coordinates of the pixels. Based on the shadow map depth sample, determine if the pixel is lit or not. Be careful with numerical errors, and use a reasonable bias value to get rid of artifacts such as "shadow acne". You will need to modify `CalcDirectionalLight` in the fragment shader to darken the pixel if it is in shadow.

You also need to modify the `Renderer` class to set the right parameters for the new Phong shader. Notice there is a new virtual method SetShadowMapping in the base class ShaderProgram, and your upgraded PhongShader will need to implement it (overriding the base class implementation) before using it.

You should be all set if everything is implemented correctly (Fig. 4). It is fine if your shadows have slight aliasing. To get rid of that, you can optionally implement percentage closer filtering (PCF) as extra credit as done by the sample solution.

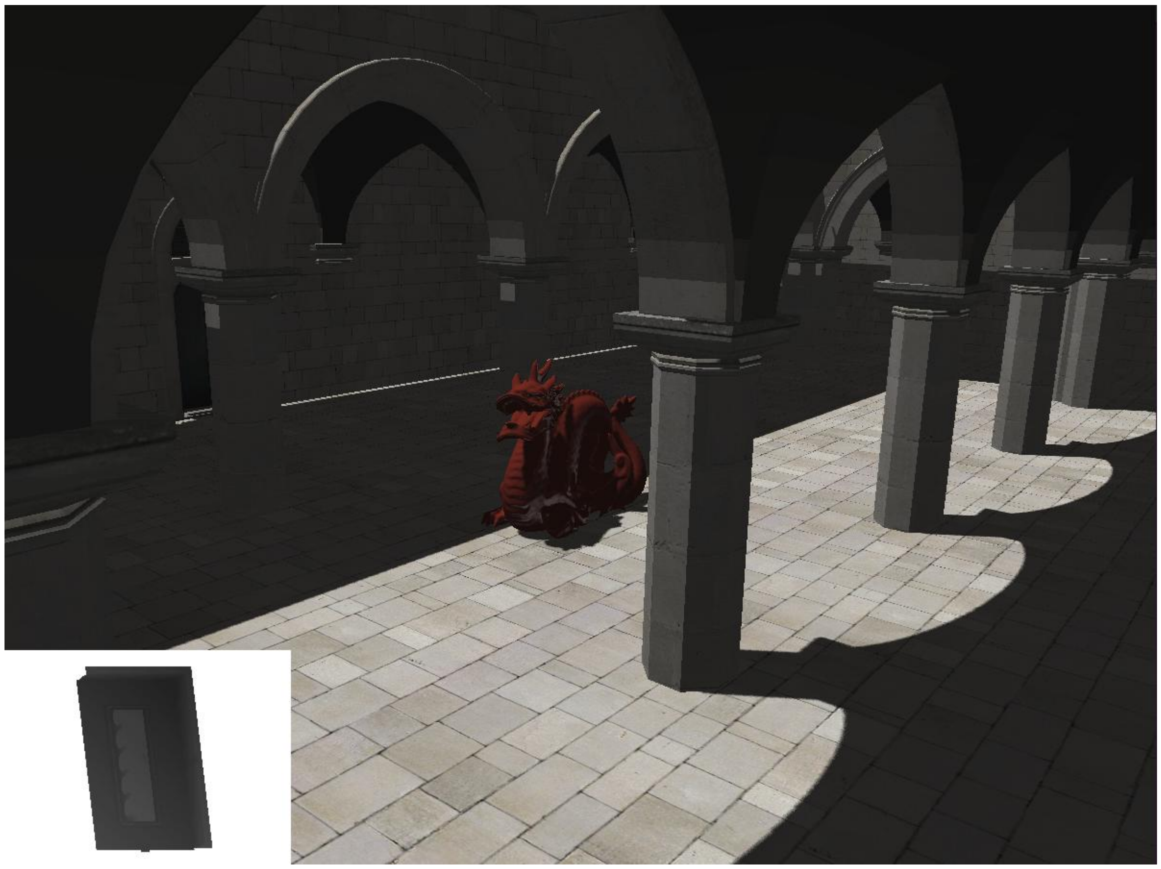
Figure 4: The final result of this assignment (screenshot includes PCF extra credit).

6 Extra Credit

Easy (3 % for each)

- Implement Percentage-Closer Filtering[4]. You can do so with a few lines of code in the fragment shader.
- Dynamically compute a tightly fitting orthographic projection matrix for the shadow camera
- We left all texture interpolation settings to their default values. As an extension, add bilinear texture filtering with mip-mapping. Can you get OpenGL to do anisotropic filtering for you?
- Add some form of Anti-Aliasing, e.g. SSAA[5] or MSAA[6].
- Add a spot light, also with shadow.

Medium (6 % for each)

- Implement Cascaded Shadow Maps[7]
- Render the scene with *screen-space* ambient occlusion[8].

Hard (9 % for each)

- Implement stencil shadow volumes.
- Use a modified version of the ray tracer from the previous assignment to render out light maps (textures that store global illumination for static illumination and diffuse objects).

7 Submission Instructions

You are to include a `README.txt` file or PDF report that answers the following questions:

- How do you compile and run your code? Specify which OS you have tested on.
- Did you collaborate with anyone in the class? If so, let us know who you talked to and what sort of help you gave or received.
- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list. In particular, mention if you borrowed the model(s) used as your artifact from somewhere.

[4] `http://graphics.pixar.com/library/ShadowMaps/paper.pdf`
[5] `https://en.wikipedia.org/wiki/Supersampling`
[6] `https://en.wikipedia.org/wiki/Multisample_anti-aliasing`
[7] `http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf`
[8] `https://en.wikipedia.org/wiki/Screen_space_ambient_occlusion`

- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. This is very important, as we're much more likely to assign partial credit if you help us understand what's going on.
- Did you do any of the extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe how you did it.
- Do you have any comments about this assignment that you'd like to share? We know this was a tough one, but did you learn a lot from it? Or was it overwhelming?

You should submit your entire project folder including your **source code** and **executable (at the root directory of your project)**, but **excluding `external/` and `build/`** since they take too much space. **Make sure your code can be built successfully, since we will compile it from scratch for grading.**

If you are on Windows, make sure the paths in your `#include "..."` pre-processing commands in `cpp/hpp` files use "/" instead of "\" as file separators. The later will not work on other operating systems.

To sum up, your submission should be your entire project folder, plus:

- The `README.txt` file or PDF report answering the questions above. Leave this file at the root directory of the project folder.
- **A screenshot of your program named `shadowmap.png` at the root directory, with the scene with clear shadows, and the shadow map visualization at the lower-left corner.**
- **As always, TAs will compile and run your code on their computers.**
- If you implemented extra credits, please attach the images and corresponding commands to generate the images using your code.
- The executable file at the root directory of the project folder.

Please compress your project folder into a `.zip` file and submit using Canvas.

We will follow the late day policy explained in Lecture 1.