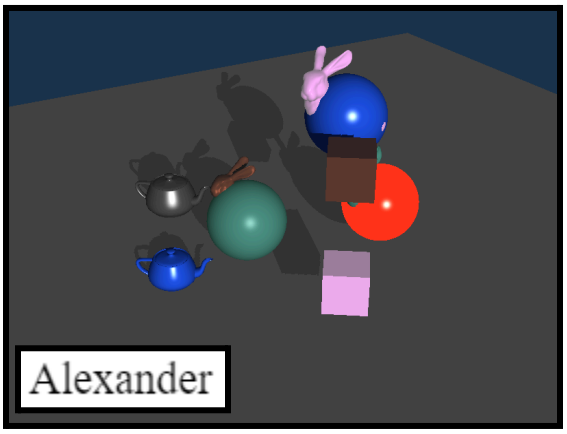
Alexander G. Arias - A16525320 Alex Simonyan - A16483204

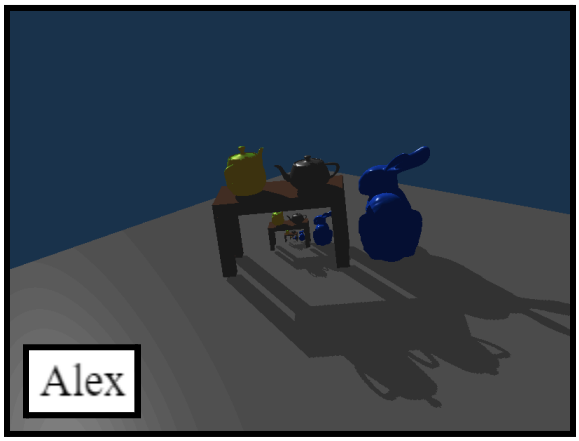
Shadow Mapping

Project Preview:

Alexander's Scene HW3



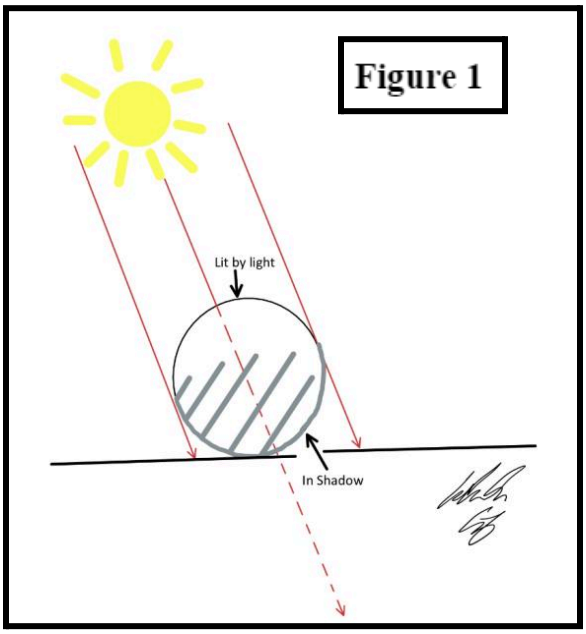
Alex's Scene HW3



Project Introduction:

In this Final Project we attempted to address shadow mapping, which took a while to understand, but eventually was pretty minimal in terms of work and understanding. We both agree that the most difficult part of this project was learning OpenGL and how to actually render to texture. However, with each other's ideas and general hard work ethic, we are happy to introduce you to Shadow mapping using C++ and OpenGL! (We assume you have CSE 167's HW3 completed and working correctly for the duration of this writeup).

So what is Shadow Mapping? What are Shadows? A beginning graphics programmer might ask, as they reconsider their general interest in graphics design, as graphics can be very difficult at times. Shadows result from the absence of light due to the obstruction by an object or other objects which prevent light rays from hitting an obstructed object or objects. As a result shadows have the capacity to add more depth to our scenes and provide spatial awareness in terms of where objects actually are within our scenes…i.e. refer to the project preview images from above.



Shadow Mapping is mainly rendering the scene in the perspective of the light. Everything that can be seen in the light's perspective must be lit, and everything that is not must be in a shadow. (**Refer to Figure 1**) It is natural and obvious that a ray of light hits the face of the illustrated circle, but such rays can not hit the circles back. In this particular case, we get a point on the ray where it first hits the object and we can use this point to compare the closest points to the other points within the

ray. By running a basic test, of continuous comparisons, of any test points ray position regarding if it is further down the ray than the closest points mentioned earlier, then such test points must be in a shadow. Overall, a depth buffer is utilized to accomplish this task, such that all depth values are stored within a texture. This is the main reason depth values can be sampled and can be tested in regards to the light's perspective. All these depth values that are stored within the texture are known as a depth map, or Shadow Map.

Project Content:

The general rundown when dealing with shadow mapping involves two passes in a program's display function or display procedure. Specifically, the first pass will render a depth map using a depth shader procedure that is in charge of drawing the scene in the perspective of the light and storing the information onto a frame buffer object. This depth map will not be colored, and the depth.frag file in charge of coloring the depth map would simply write out depth values onto this depth map. The second pass will render the scene by comparing depth values for every pixel in a standardized shader procedure, that is normally used to render the scene as usual, and determine whether an object should be included in a shadow or not. At the end of the procedure, the scene is drawn in the perspective of the real camera and would include the corresponding shadowed scene.

We can start this process by first making a depth map. In order to do this, we will need the help of our family friendly frame buffers, so that we may render the depth map. We can accomplish this by making a global variable called **GLuint depthMapFBO** to have global access to the depth map frame buffer and by making the GL function call: **glGenFramebuffers(1, &depthMapFBO)** in order to create a new frame buffer object, within the initialize function of the main.cpp file.

We then need to create a 2D texture to use as the framebuffer's depth buffer. This can be accomplished by simply making a global called **GLuint depthMap** and some constant globals initialized as follows: **const unsigned int SHADOW_WIDTH = 1024** and **SHADOW_HEIGHT = 1024.** From here, we would have to bind the depth map to the frame buffer and conduct a few more OpenGL procedures highlighted here. Once that is done and added to our initialize function in main.cpp right before **scene.init(),** we can move on to define a new camera, call it the light camera, to get the perspective from our lightsource onto the scene. This needs to be done before our first pass can happen. Some basic additions to accomplish this task would be to declare a compute light camera matrices function in our Camera.h file, declaring a new camera instance in Scene.h, a new draw function for the light camera in Scene.h, and a new depth shader function header that comprised solely of view, modelview, and projection matrices. This might sound like a lot, and it is, but it is necessary to continue forward with our implementation. (See the following illustration, Figure 2, as a checkpoint. Regarding getting the

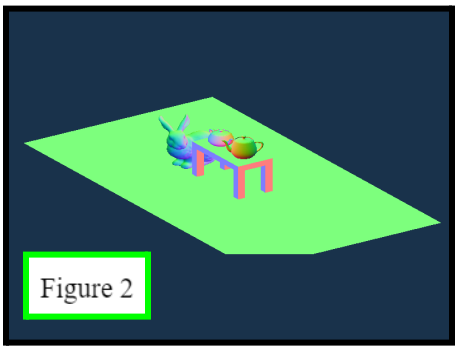


Figure 2

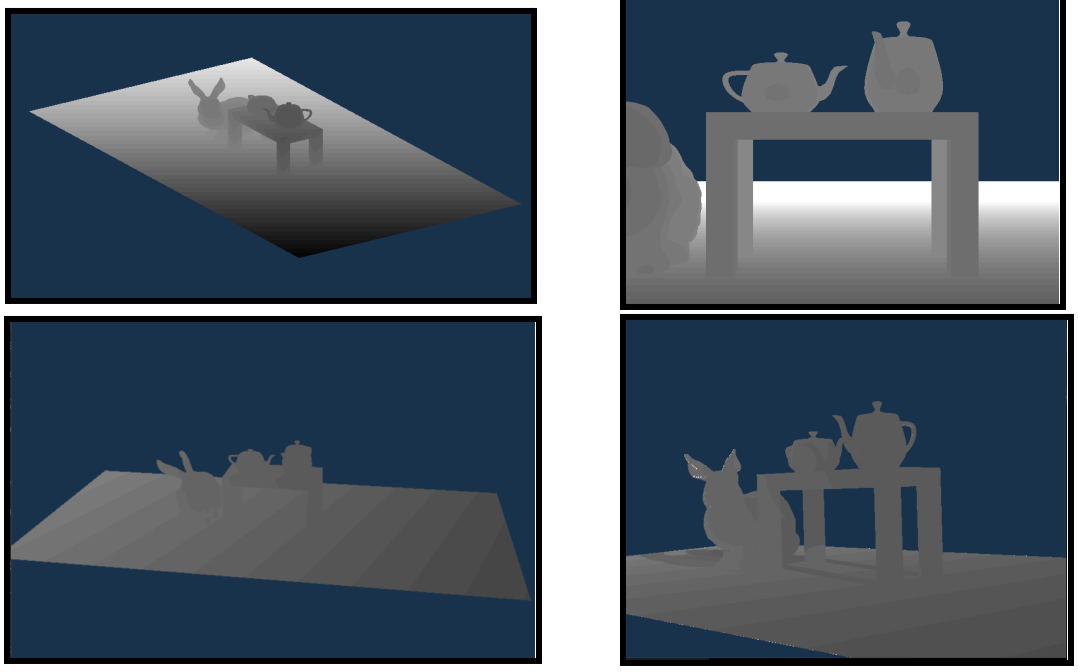
lights perspective of the scene). Here is some source code that might be able to help you implement the light camera and its perspective using OpenGL lookAt() and ortho() procedures.

Once the new light camera is created in Scene.h, and defined in Scene.cpp, and its perspective can be verified, we would like to render the scene from the light camera's perspective onto the depth map. We accomplish this task by simply making a vertex lightSpace.vert shader that only transforms the vertices of the scene to the light space and a fragment depth.frag shader that processes every depth value of the current scene as seen by the light source. Our first pass may now be implemented by the following process within our main.cpp's display function, in order to render the depth/shadow map:

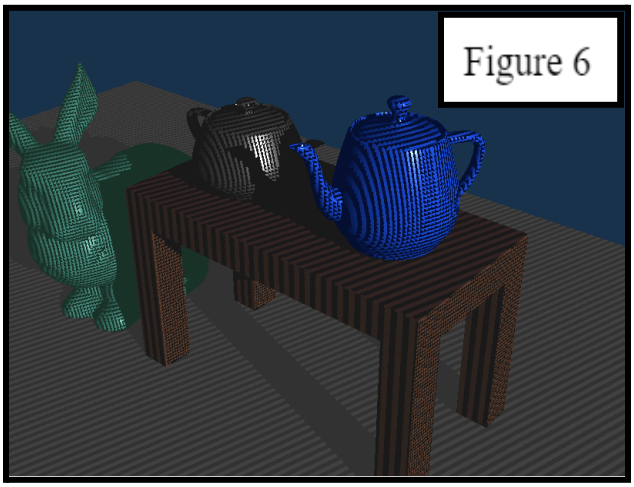
```
// 1. first render to depth map
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
glUseProgram(scene.depthShader->program);
scene.drawLightCameraView();
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

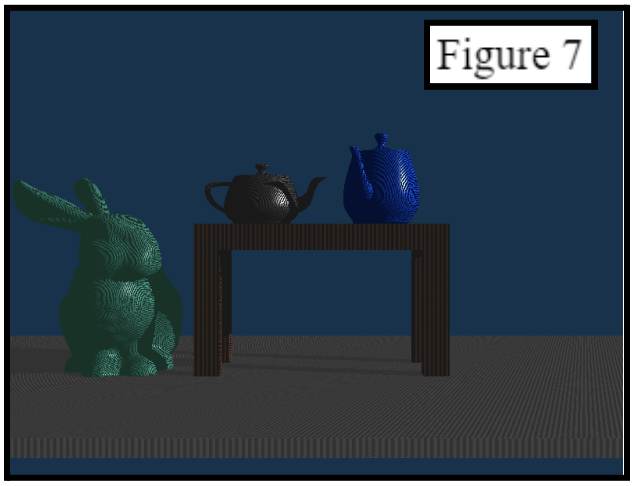
Scene.drawLightCameraView() is a function that is mostly in charge of populating the corresponding model, view, modelview, and projection matrices for the light camera. Overall this procedure should result in a nicely filled depth buffer holding the closest depths of each visible fragment from the light's perspective. In order to ensure the work you are doing remains accurate and purposeful, here are some example 2D textures of the depth map, as outputted by the program. Figure 3 is what the light sees using the orthogonal projection matrix, and Figure 4 is what the camera sees using the perspective projection matrix, with the color being the depth from the light and camera respectively. Figure 5a is what the camera sees when the color is the current depth from the light, and Figure 5b is when the color is the closest depth from the light.

Figure 3 - 5



Now that we have a properly generated depth map we can proceed to generating our actual shadows. For every world-space vertex position, we transform it to light space in our lighting.frag fragment shader file. The tutorial from OpenGL does this process in a different shader, but for consistency, we decided to simply move this process into our lighting shader. This main shader uses the Blinn-Phong lighting model, and it is nice to observe that the process remains consistent as HW3 was based on implementing the Blinn-Phong lighting model in this lighting.frag shader. Overall, we implemented a **ShadowCalculation()** function that calculates a shadow value that is either 1.0f when a fragment is in a shadow, or 0.0f when not in a shadow. Since shadows are not completely dark due to light scattering, ambience was not part of the multiplication process. The process can be found here. Once the shader is activated while binding the respective textures, the second pass will render the projection and view matrices and result in a scene that is similar to the following Figure 6 and 7 below:



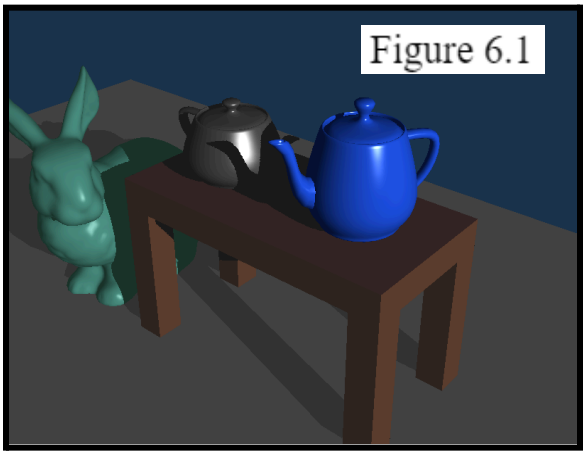


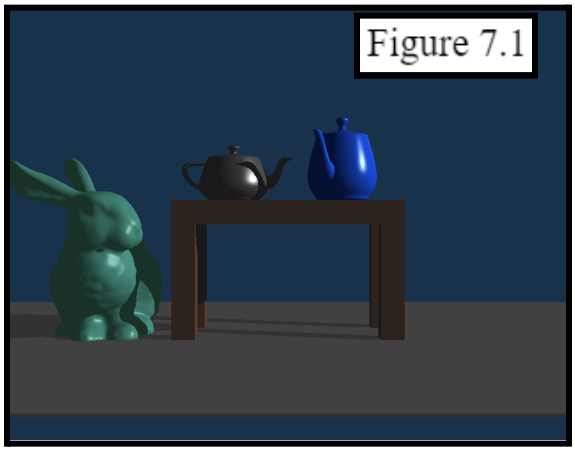
If you have made it this far, then CONGRATULATIONS!, we did things right! And you should be as proud as we were when we saw this shadow acne! If not, then you can refer to the demo application that we based our project on here, and try to follow the tutorial more closely in terms of logic introduced.

Now, it is obvious that something wrong is happening within our application due to the unwanted line artifacts throughout the scenes illustrated in the above figures. What is shadow acne? Shadow Acne is a circumstance that is caused by multiple fragments sampling the same values from the depth map when the light source is relatively far away. Although this is fine, it starts to become a problem when angles are introduced since multiple fragments start accessing the same angled depth values, causing some parts to be above or below the floor/surface causing this unwanted pattern. You can also refer to the explanation and diagrams given here. In order to fix this shadow acne, all we need to do is introduce a shadow bias which offsets the depth of the surface(s) or shadow map in our **ShadowCalculation()** function. The following code snippet should suffice, and should replace our old shadow calculation.

```
// check whether current frag pos is in shadow
float bias = 0.005;
float shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;
```

Once the shadow bias is implemented, a similar looking scene should be rendered to the one below:





If for some reason, you are still able to see some shadow acne, replacing the sampled value by an increment throughout program runs can potentially allow you to choose the correct bias value for your scene to completely illuminate the shadow acne. Overall, it is just a process of continuous incrementation until you can observe no more shadow acne.

Project Conclusion:

If you have made it this far, then it means you must have been able to fully implement the shadow mapping according to the tutorial and this write-up's specifications. Overall, the process is a bit confusing and a bit complex for such a short problem that needs to be solved. Aside from this, we will like to end with a brief overview of our personal experience(s) within this project.

Alexander G. Arias:

I was stuck on how to get started the first couple of days, until I managed to talk to Professor Chern about potential hints or tips regarding shadow mapping. This was when he informed me of the OpenGL shadow mapping tutorial mentioned throughout this writeup. I eventually understood the basics regarding setting up a new camera that holds the perspective of our light source onto the scene, however I hit a major roadblock when it came to rendering to texture. With the help of Professor Chern and Teaching Assistant Peter, I was finally able to grasp that rendering to texture simply requires making a frame buffer object and a depth map to store the values that we would acquire from rendering the scene in the first pass. With more help from Alex Simonyan, my group partner, the OpenGL procedures became easier to understand, and a great majority of the confusion was lifted. Fortunately, with just a few more observations and tweaks, we were finally able to acquire a finished product in terms of the shadow mapping. As a result, we have more deliverables waiting outside this writeup for more visual illustration of the process.

Alex Simonyan:

I was also stuck for multiple days because I was rusty on OpenGL and it was hard to understand what we were doing incorrectly when referencing the tutorial. After many hours of

changing numbers and observing differences in the scene that weren't what we wanted, I started getting worried about finishing the project. After having wasted a lot of time and not making any progress, I decided to read the other introductory tutorials relevant to things used in the project (shaders, textures, etc.), OpenGL finally started making sense to me. After that, me and Alexander ended up realizing what things were missing in our project, and a few tiny fixes made the shadows finally work, which was an extreme relief for both of us.

Resources and Works Cited

- https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping
- https://cseweb.ucsd.edu/~alchern/teaching/cse167_fa21/project-shadowmap.pdf