James Chung
A15852697

Final Project Writeup (Shadowmapping)

Introduction

Previous assignments in this course have given us experience with rendering scenes and adding Blinn-Phong shading to color scenes. For realism though, just shading objects based on lighting is not enough; shadows are an important aspect of capturing photorealism in graphics. One technique to implement shadows in a scene is the process of shadow mapping. In simple terms, the shadow mapping technique involves generating a depth map (often in the form of a gradient of black and white) of the scene and comparing that mapping to our scene to determine what pixels are under a shadow.

General Process

To go more in depth, shadow mapping requires two passes of rendering the scene. In the first pass, we view the scene in the coordinate system of some light source in the scene. For the sake of simplicity in this project, we only handle a single light source. Acting as if the camera is at the light source pointing at the scene, we then use the depth value of each fragment from the light source to generate a color for each fragment. This color should be some encoding of the depth value, usually seen as a gradient between black and white. The resulting scene is then the depth map.

After this first pass of rendering, the depth map is saved as a texture so the second pass can refer to it later. Moving on to the second pass then, we want to see if there is a shadow or not. The way to check this is to compare the sampled depth from the depth map with the depth value of a given fragment from our actual camera; if the sampled depth is (noticeably) shorter than the current depth value, that means the pixel is blocked from the view of the light source, so there is a shadow. If there is a shadow, then do not have the light source to contribute to the shading except the ambient component.
(I have omitted heavy technical details due to time constraints)

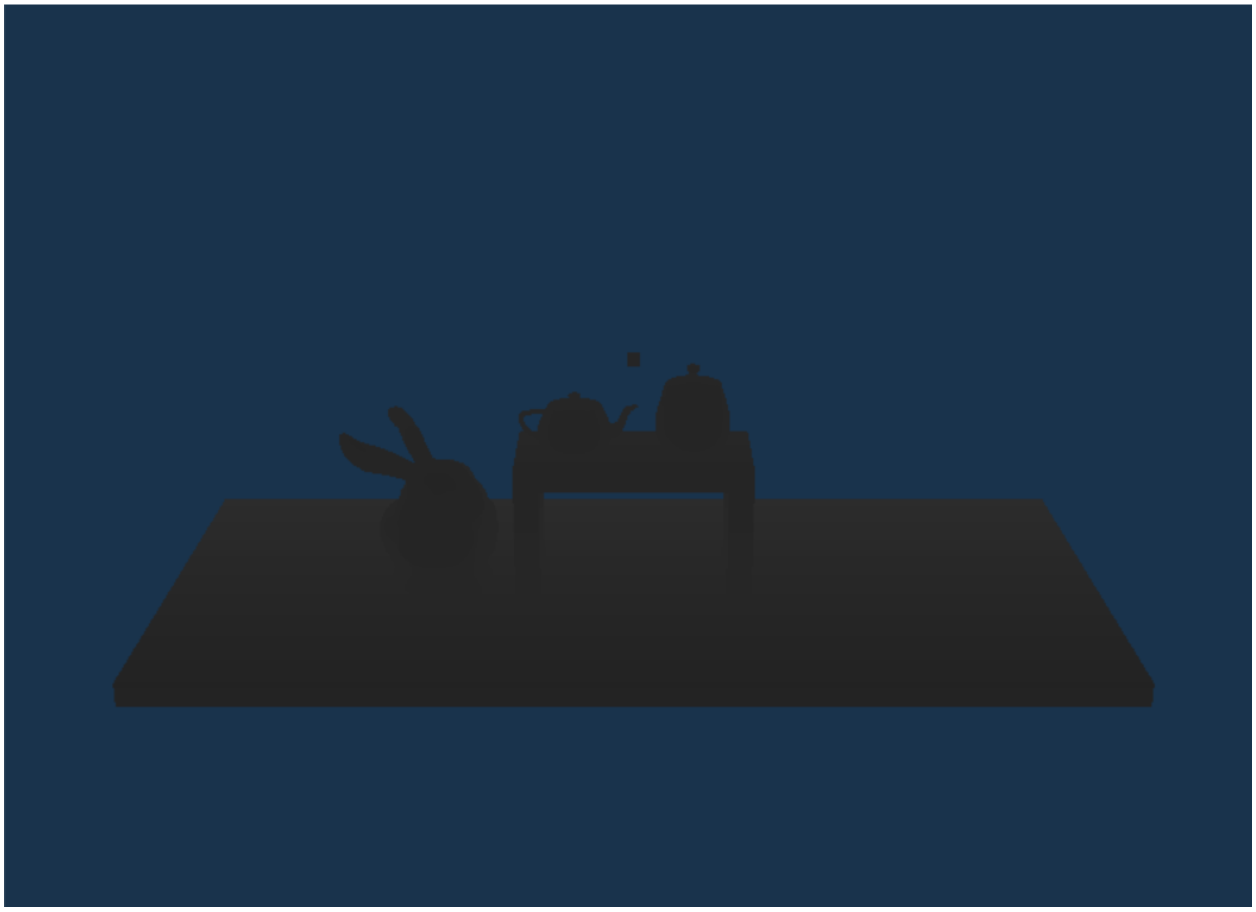
Steps for Implementation

(Generally, I followed the steps in the Piazza megathread for shadow mapping (@423))
*Note: Screenshots were taken after improving shadow quality with PCF and Poisson sampling, so the shadows may not reflect exactly what they looked like during the development process.
Task 1:

1) I created depth.frag such that it sets the fragColor to some function of the depth (z-value). I tried to set it directly to z at first, but I found the depth map was easier to visualize by using the function using near/far to assign fragColor.
2) I created depth.vert, which is really just a direct copy of the original projective.vert. Later, I found it useful to remove modelview and instead split it into model and view (in both depth.vert and projective.vert).

3) I created DepthShader.h, which is a stripped down version of SurfaceShader, but also passes in near and far values as uniforms (for depth.frag's fragColor computation). The only other variables I needed in DepthShader was model, view, projection, and enablelighting.
4) Finally, in Scene.inl I added the necessary code to initialize the DepthShader (basically identical to initializing SurfaceShader, but with depth.vert, depth.frag instead)

For a preliminary test, here is the depth map produced when I temporarily changed SurfaceShader to use depth.frag (zoomed out/adjusted to see gradient better):



(I found near =1.0, far =8.0 best suited the depth map shading overall for this task and future tasks)

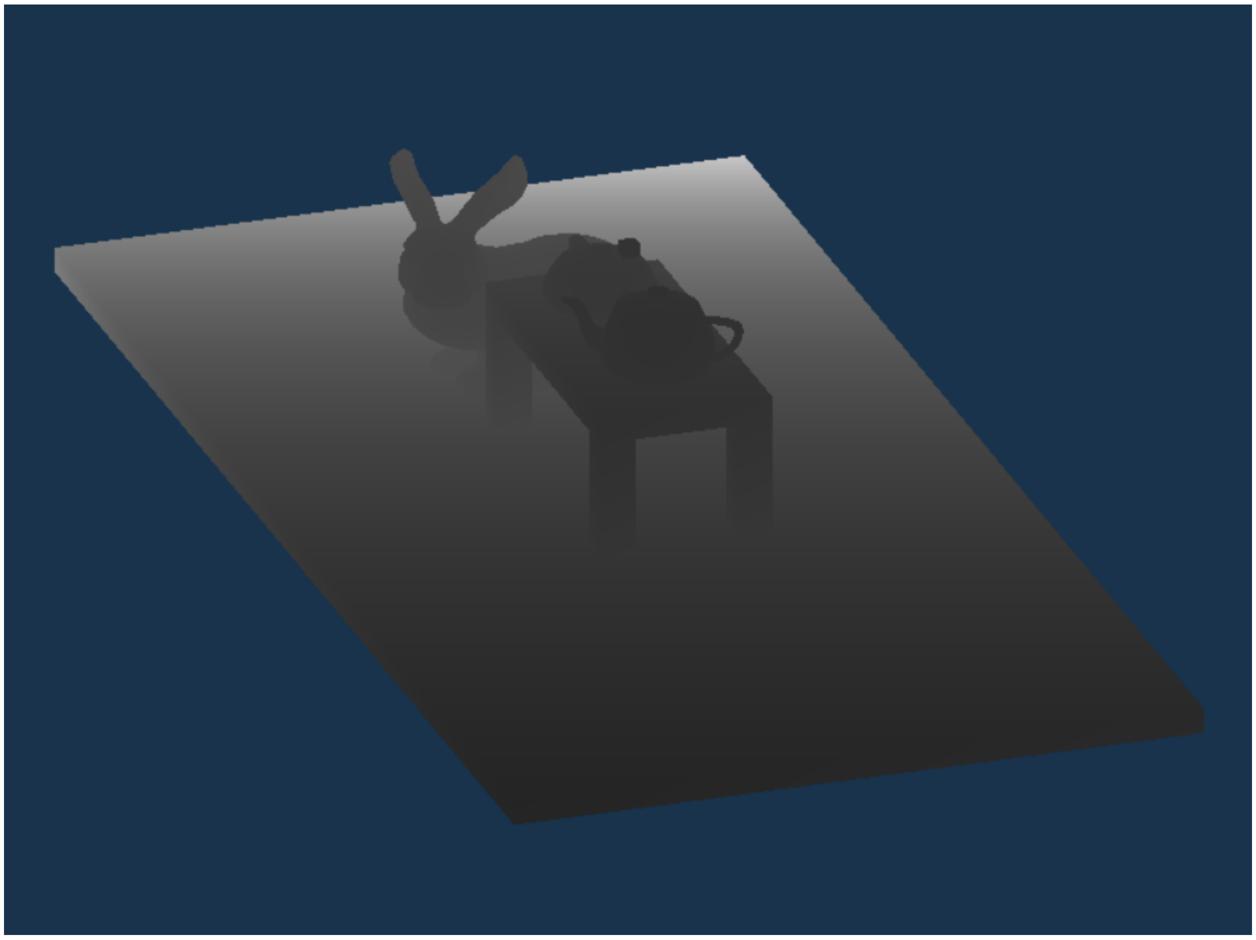
Task 2:

5) In scene.cpp, I copied the original draw() and modified it to only work with model matrices and not the modelview matrix. The pre-draw sequence takes care of setting up the camera space view/model, so all that is left is setting up light space view/model.
6) In light.h, to set up light space view/model, I added a view and model matrix. I also added a computeMatrices() function (similar to camera's) to dynamically compute the view and model matrices after the light's position was known. This function was called by scene.draw() before setting shader's light space view/model. The view model was

created using glm::lookat with the light's position (technically directional), the target which is the center of the screen/origin, and the general up vector (0, 1, 0). The projection matrix is just an orthographic projection with the first 4 values being adjusted to maximally fit the depth map, and the near/far values being adjusted to reduce sharp cutoffs at the front/back of the scene.

7) For now, I hotwired the scene.draw() call in main.cpp::display() to the depth shader.

Here is the resulting depth map from the light's perspective:

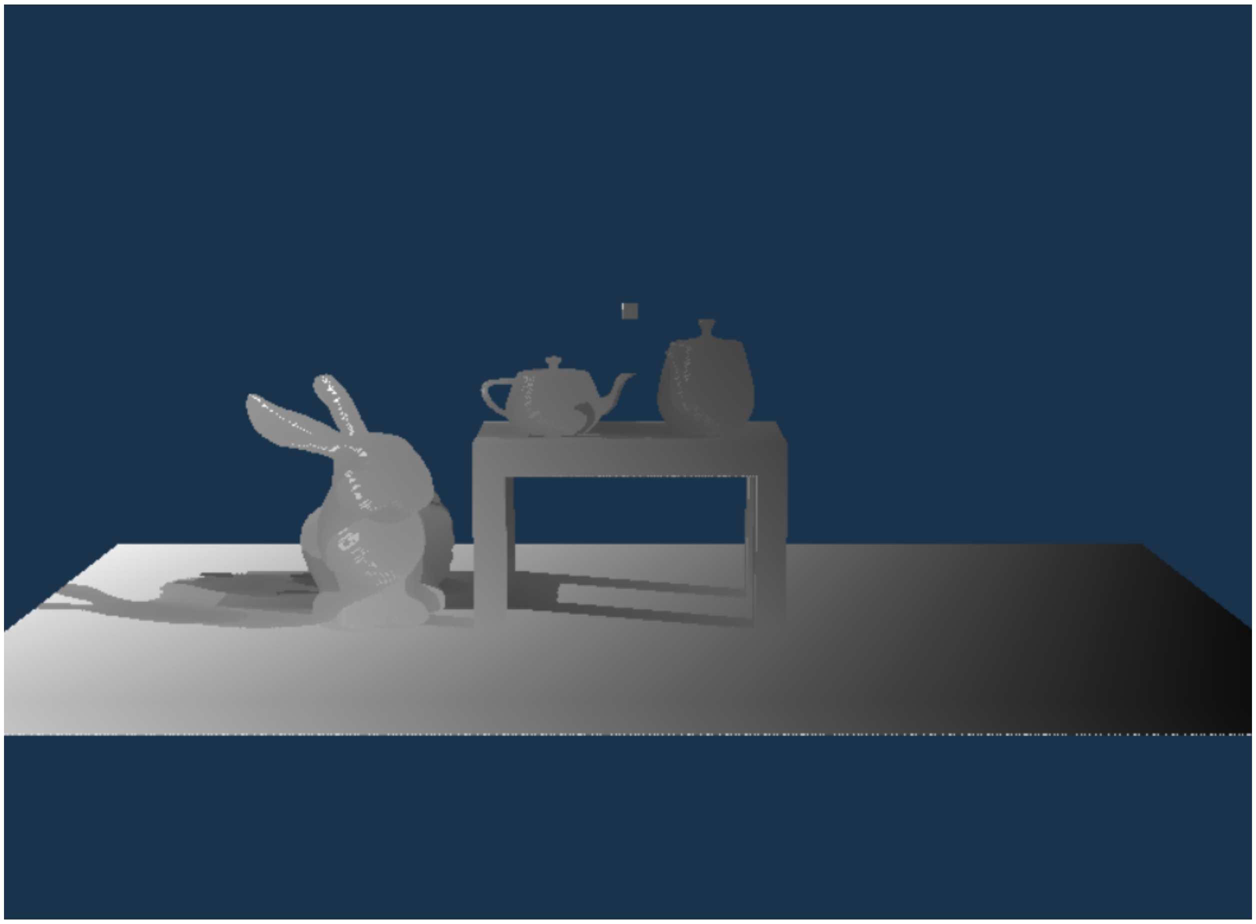


Task 3:

8) In Light.h, I added the GLuints depthMap and depthMapFBO, initializing them in the Light() constructor by following the code provided in the project writeup on Page 3.
9) In main.cpp::display(), I modified the rendering to do the two-pass method, following pseudocode from the piazza megathread. Specifically, the depth pass involves binding the depthMapFBO and drawing the depth map using the depth shader. The second pass binds the actual depth map to a texture and draws the final scene. At this point though, the shadows have not been computed yet.

Task 4:

10) In lighting.frag, I added the sampler2D shadowMap. To check if Task 3 was correct, I added a debug option to assign the frag color to:
fragColor = vec4(vec3(depthSampled), 1.f);
Using texture() to get the sampled depth.
The resulting image looks correct:

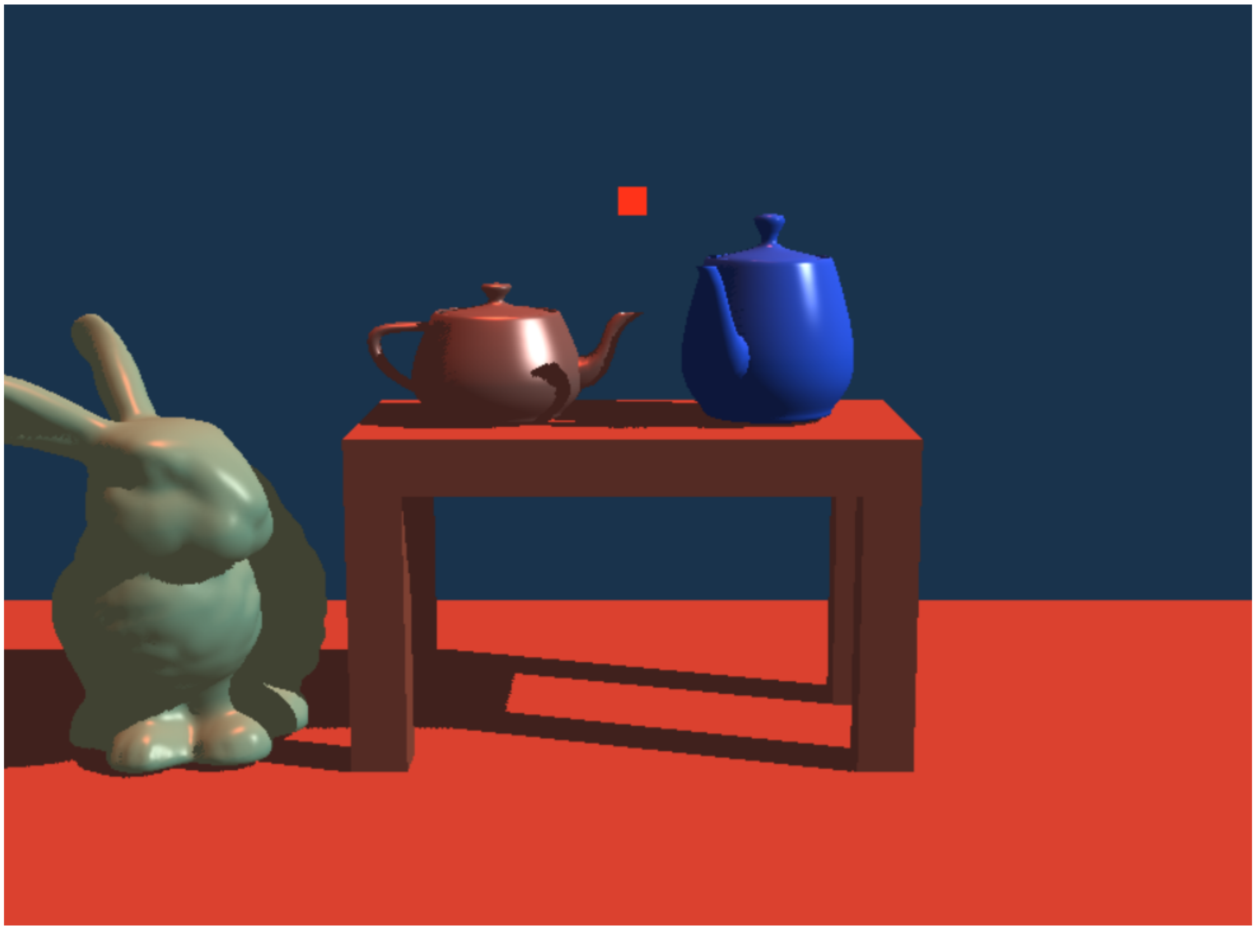


11) In projective.vert, I added a new out variable position_light, which transforms a position into the lights coordinate system. This variable is then used as an in to lighting.frag.
12) Back in lighting.frag, I added a function computeShadow() which takes position_light, transforms into NDC, de-homogenizes, and compares position_light.z (new depth) to the shadow map's depth at that position. The function returns 1.0 if there is a shadow (newDepth > sampledDepth), 0.0 if there isn't.
13) computeShadow() in the Blinn-Phong shading to keep or remove a light's contribution to a fragment's shading, except the ambient shading. This is done by multiplying the diffuse and specular by (1.0 - computeShadow()) such that the contribution is not added if there is a shadow.

Shadow Quality improvements:

14) Shadow acne: I introduced a bias such that now the shadow comparison is (newDepth - bias > sampledDepth). The bias is somewhat based on the normal and position in light-space.

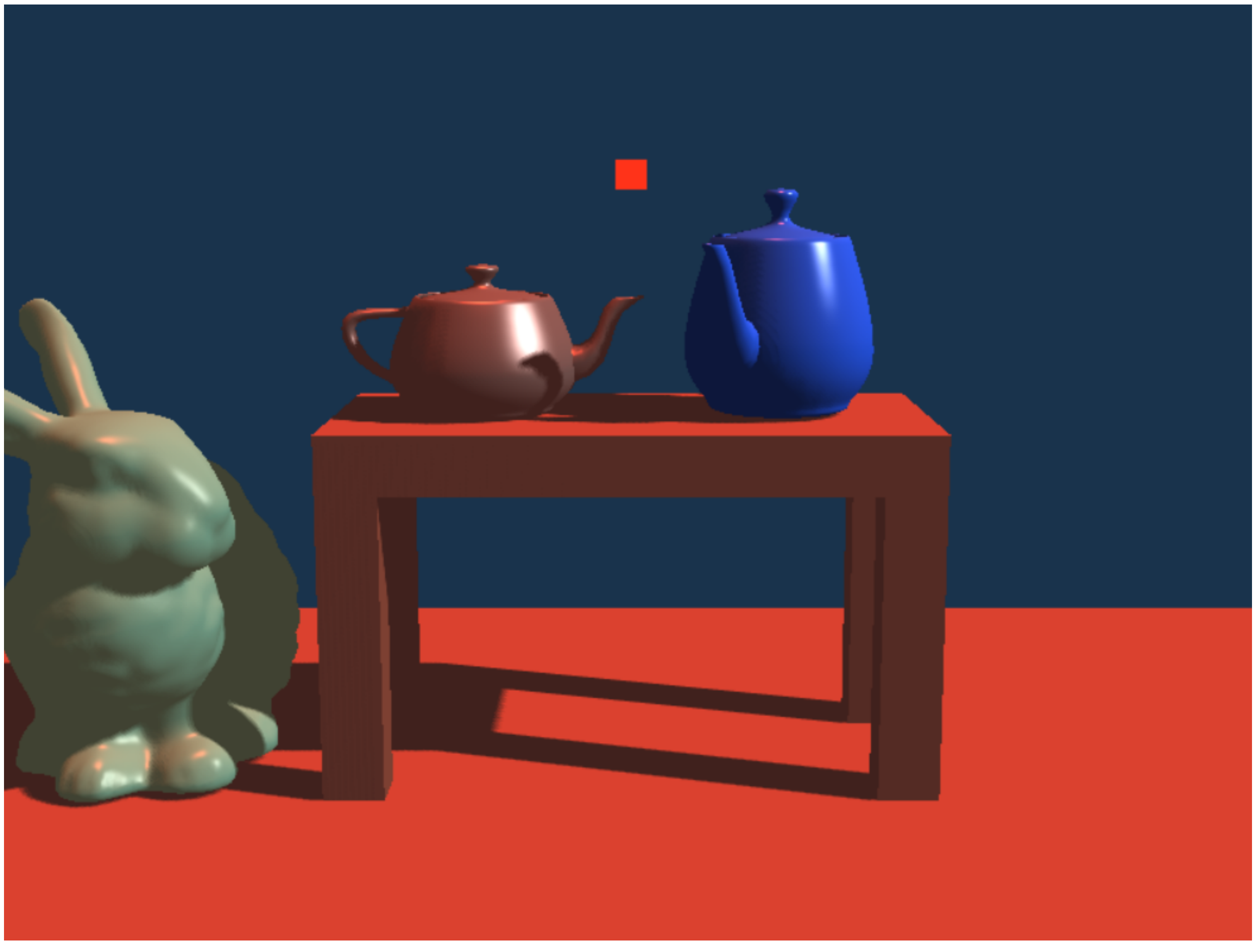
Resulting image:



15) PCF: Following the example of PCF from learnOpenGL, I implemented PCF to soften the shadows' jagged edges. This is in the function computePCF(), which replaces computeShadow() in the Blinn-Phong shading. PCF essentially samples depths from neighboring texels as well as the original position and averages the shadow result to get a value from the range [0.0, 1.0].
16) Poisson Sampling: Similar to PCF sampling neighboring texels, I added Poisson sampling to offset sampled depths using Poisson discs. This was simply added into computePCF()

Note: I tried to implement the augmented PCSS on top of PCF, but due to time constraints, I ended up not finishing that implementation. Looking at examples of PCSS though did give me the idea to use Poisson sampling in PCF because those examples also use Poisson sampling in PCSS.

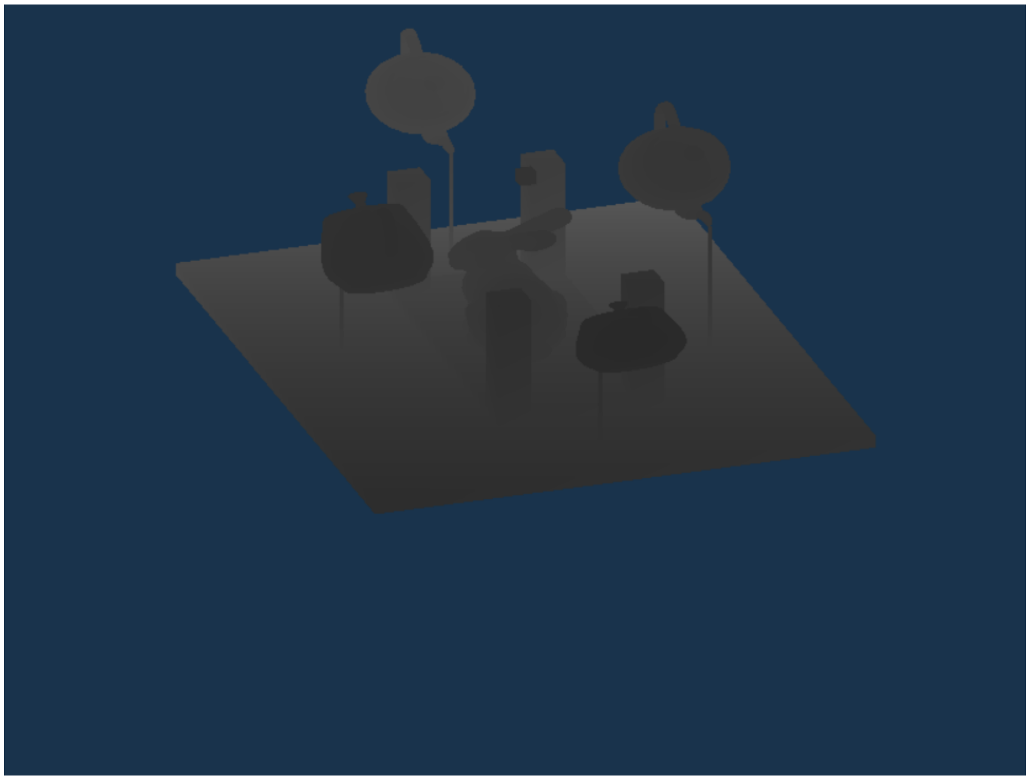
The final image with shadow acne removal, PCF, Poisson sampling:

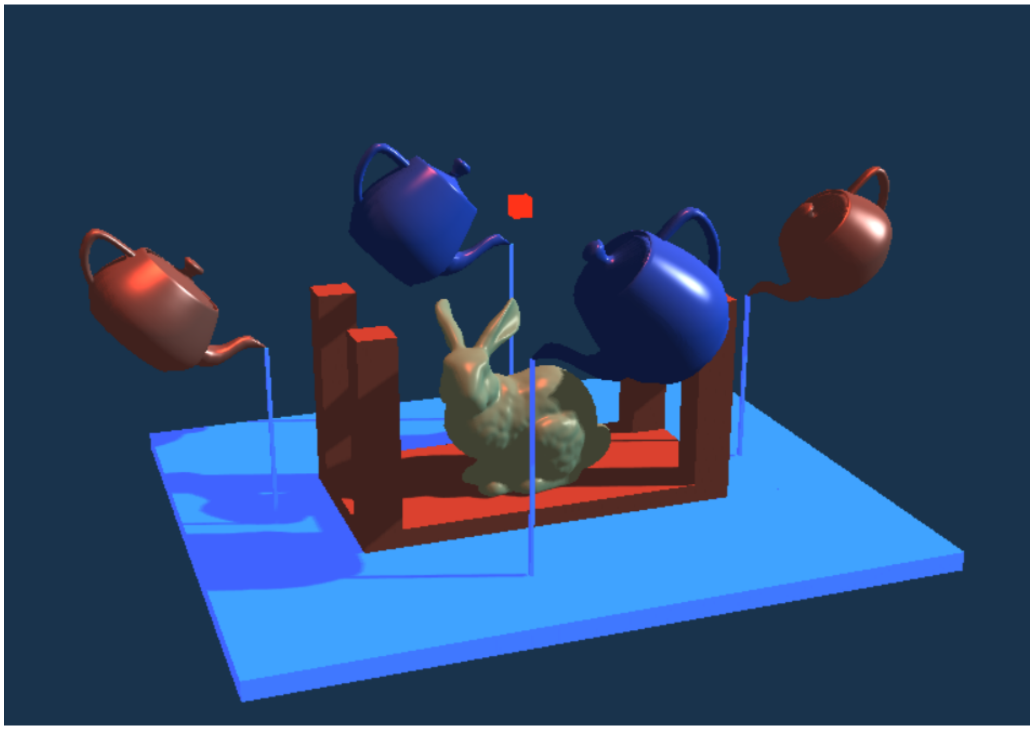


As you can see, the shadows are a bit softer and thus it is harder to see artifacting/aliasing.

Bonus

Here's some bonus renderings of my submitted scene for the competition back in HW3.





And a short clip showing the soft shadows (a bit low quality):
https://drive.google.com/file/d/1QS6WnDtr2E0dGtfwkXkTEyZXb7xRnbOA/view?usp=sharing

References

1) http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/#poisson-sampling
2) https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping
3) https://andrew-pham.blog/2019/08/03/percentage-closer-soft-shadows/