# CSE 167 Final Project (Daryl Foo #A16535281)

## Introduction:

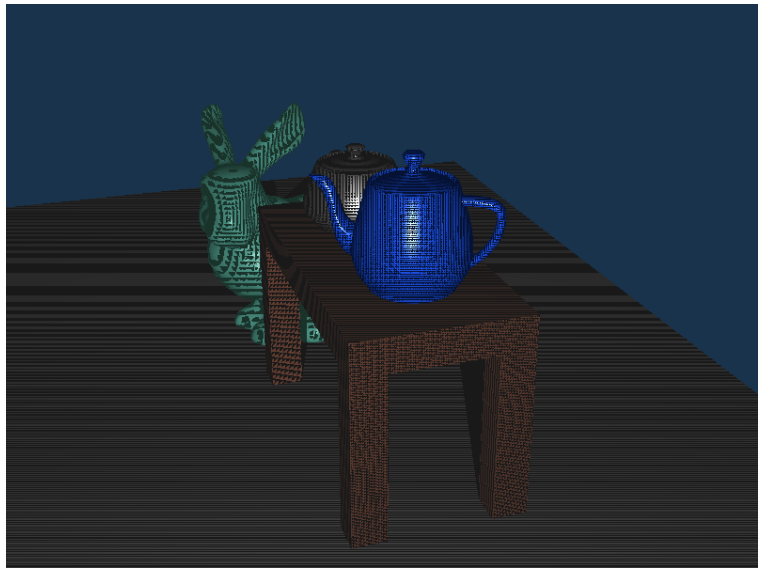
The topic I worked on was Shadow Mapping. I used HW 3 as the basis for this project. The problem with HW 3 was that even though it had proper shading, the objects did not cast shadows, causing the scene to lack realism. Hence, the goal for this project was to implement a basic shadow mapping to the scene based on a single light source. In addition, I also used the PCF method to reduce aliasing, so that the shadows looked less pixelated and hence more natural.

## Explanation:

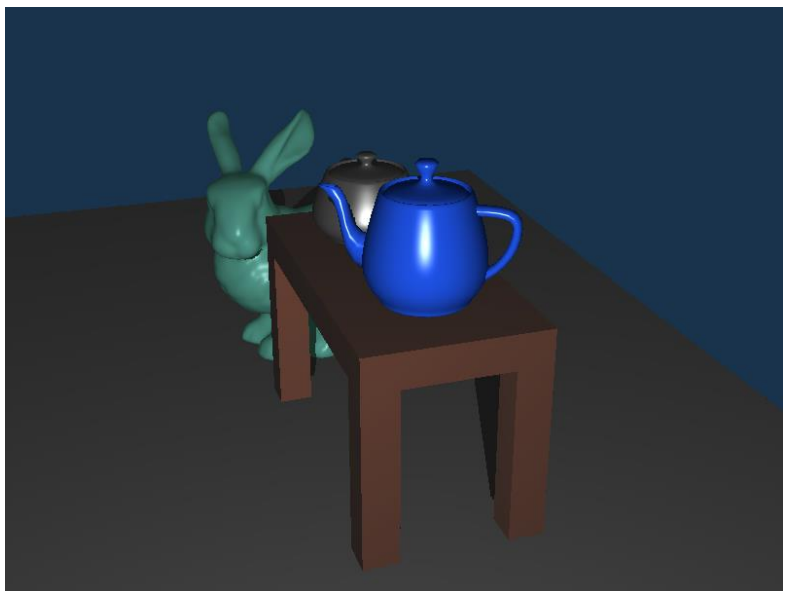
To produce shadows, the scene had to be rendered twice, once with the camera at the light source to cast shadows, and again with the camera at its actual position to form the image. I first tested the “depth.frag” to produce a scene where the color of each pixel represents its depth in the scene (black = close, white = far). With our camera at our default camera position (0.0f, 1.0f, 5.0f) and “depth.frag” properly implemented, the depth of the scene looked like below:

Next was to carry out our first pass and render the scene from the perspective of the light source (light space) at coordinates (5.25f, 3.5f, 1.75f) and store that into a texture, which produced the scene below:

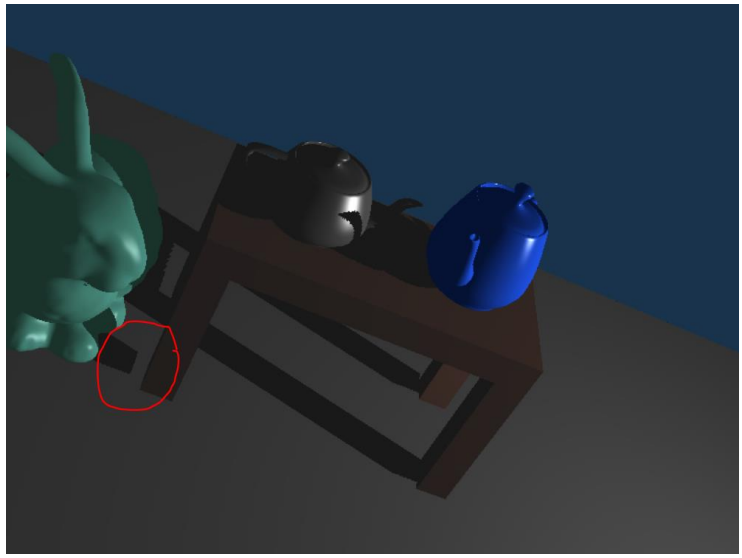
Using that texture, I just had to use “lighting.frag” to produce the scene with proper (Blinn-Phong) lighting. Even though the shadows looked right, the scene looked like below:



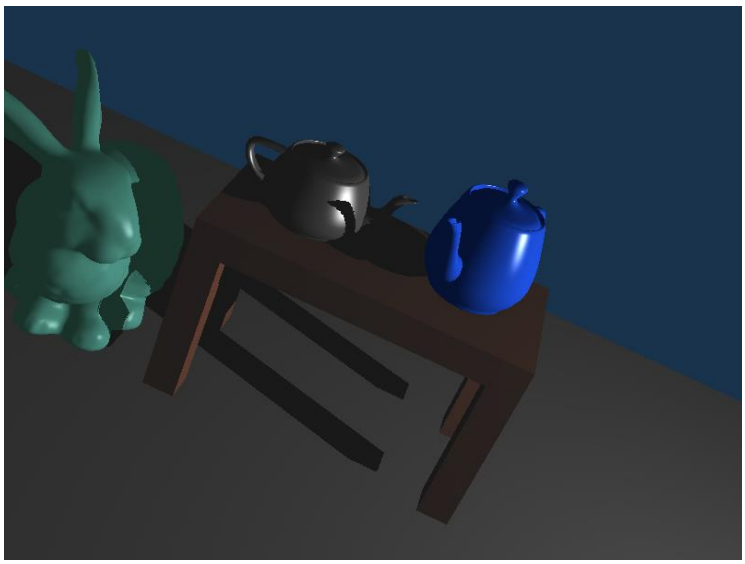
This was what is known as shadow acne, which occurs due to limited resolution of the shadow map, causing multiple fragments to sample the same value from the depth map when they are relatively far away from the light source. To fix this, I just introduced a bias to the sampled depth from the shadow map. After implementing the bias, the scene looked much better:



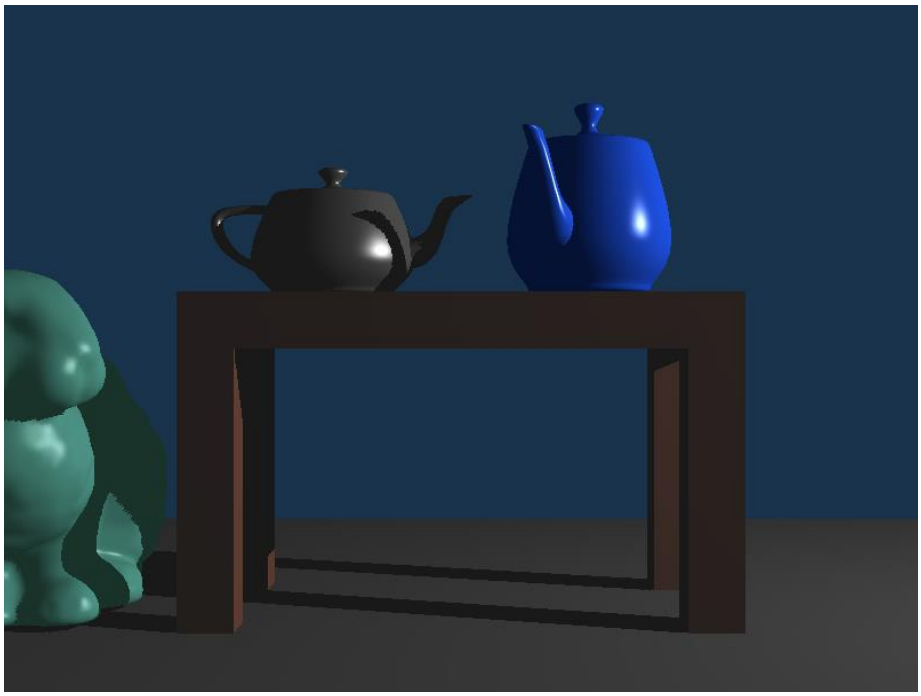
However, upon closer inspection, I noticed that the shadow was still slightly off. For example, the shadow in the red circle below was supposed to be the shadow of the table leg, but it seemed to not be attached to that leg.

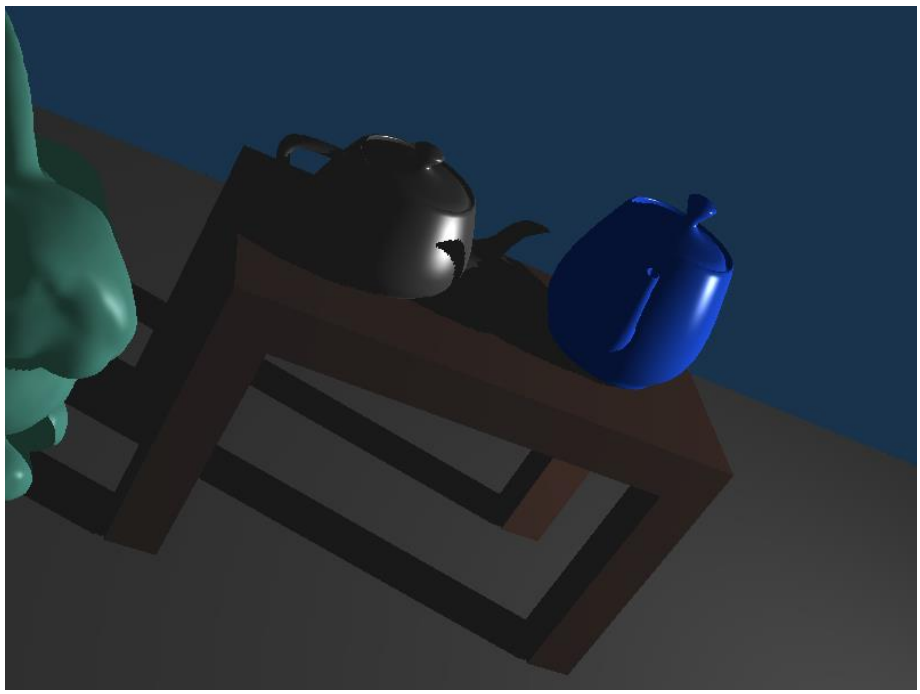


This was known as peter panning, an unintended effect produced when objects look detached from their shadow, making it look like the object is floating, hence its name. This was due to the use of the bias, which is actually an offset applied to the actual depth of objects. In some cases, this bias is so large such that the offset is visible. An exaggerated example of peter panning produced a scene like so:



To fix it, all that had to be done was to make the bias value smaller.With this, peter panning was fixed, creating a scene with an accurate (enough) shadow:





Even though the scene looked good enough from afar, when the scene was zoomed in, it could be seen that the shadow looked quite pixelated:

One solution to this is known as PCF, or percentage-closer filtering, which uses different filtering function to produce softer and less blocky shadows (more details below). The final result is a shadow that is much less pixelated and more smooth:

## Documentation:

The code was largely based from HW 3, hence a majority of the scene was already implemented. However, some changes still had to be made.

i) I added a "floor" object to *scene.inl*, so that the shadows could be projected onto a ground plane.

```
node["floor"]->models.push_back(model["floor"]);
node["floor"]->modeltransforms.push_back(scale(vec3(10.0f, 0.1f, 5.0f)) * translate(vec3(0.0f, -0.5f, 0.0f)));
```

ii) I created the *DepthShader*, which inherits from *Shader*, and is very similar to the already existing *SurfaceShader*, but with less variables.

```
glm::mat4 modelview = glm::mat4(1.0f);  GLuint modelview_loc;
glm::mat4 projection = glm::mat4(1.0f); GLuint projection_loc;
```

The only necessary uniform variables for our *DepthShader* was our *modelview* and *projection* matrices, which are needed to simulate the perspective from our light source.

iii) Two new shaders were added, the fragment shader *depth.frag* and vertex shader *depth.vert*. For *depth.frag*, I had to produce a pixel color to represent the depth (distance from light source) of each pixel. To do so, I just had to assign *fragColor* based on its z-coordinate with some rescaling like below:

```
fragColor = vec4(vec3(gl_FragCoord.z / gl_FragCoord.w), 1.0f) * 0.1f;
```

For *depth.vert*, it was just a simplified version of *perspective.vert*,

```
gl_Position = projection * modelview * vec4(vertex_position, 1.0f);
```

because *gl_Position* is calculated the same way as done in our usual scene.

iv) *Scene.cpp*'s draw function was modified to take a boolean parameter *isLight*. If *isLight* was true, we would draw the scene using the *SurfaceShader*, if not, we would draw the scene using the *DepthShader*.

```
// If (isLight != 0), draw using SurfaceShader, else if (isLight == 0) draw using DepthShader
void Scene::draw(bool isLight){
```

v) Light.h was changed to a class, to store the shadow map texture and buffer object. Also we set the uniforms (*modelview* and *projection*) of *depth.vert* appropriately.

```
Light() {

    shader = new DepthShader();

    glm::mat4 modelview = glm::lookAt(glm::vec3(position), glm::vec3(0.0f, 1.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f)) * glm::mat4(1.0f);
    glm::mat4 projection = glm::ortho(-4.0f, 4.0f, -4.0f, 4.0f, 0.5f, 12.0f);

    shader->read_source("shaders/depth.vert", "shaders/depth.frag");
    shader->compile();
    shader->initUniforms();
    shader->projection = projection;
    shader->modelview = modelview;

    glGenFramebuffers(1, &depthMapFBO);

    glGenTextures(1, &depthMap);
    glBindTexture(GL_TEXTURE_2D, depthMap);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
    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_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMap, 0);
    glDrawBuffer(GL_NONE);
    glReadBuffer(GL_NONE);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

vi) In *main.cpp*'s display function is where the two-passes are implemented. First is the depth pass, then the color pass, both done by calling the previously mentioned *draw()* function.

```
// Depth pass
Light* light = scene.light["sun"];
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
glBindFramebuffer(GL_FRAMEBUFFER, light->depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
glViewport(0, 0, light->shadowWidth, light->shadowHeight);
scene.draw(0);
glBindFramebuffer(GL_FRAMEBUFFER, 0);

// Color pass
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glViewport(0, 0, width, height);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, light->depthMap);
scene.draw(1);

glutSwapBuffers();
glFlush();
```

vii) Finally, in *lighting.frag*, is where lighting and shadow is calculated.

```
vec3 projCoords = (positionLS.xyz / positionLS.w) * 0.5f + 0.5f;
float currentDepth = projCoords.z;
float closestDepth = texture(shadowMap, projCoords.xy).r;
float shadow = currentDepth > closestDepth ? 1.0f : 0.0f;
```

First, *projCoords* are scaled appropriately, and a naive *shadow* is calculated. This simplified *shadow* calculation results in shadow acne (explained earlier). To fix this, we implement a bias and recalculate the *shadow* like below:

```
float bias = max(0.05f * (1.0f - dot(n, dir_LS)), 0.005f);
float shadow = currentDepth - bias > closestDepth ? 1.0f : 0.0f;
```

However, this bias calculation results in minor peter panning (also explained earlier), hence we change the value from 0.05f to 0.005f to resolve this issue. The final *shadow* calculation looks like this.

```
float bias = max(0.005f * (1.0f - dot(n, dir_LS)), 0.005f);
vec3 projCoords = (positionLS.xyz / positionLS.w) * 0.5f + 0.5f;
float currentDepth = projCoords.z;
float closestDepth = texture(shadowMap, projCoords.xy).r;
float shadow = currentDepth - bias > closestDepth ? 1.0f : 0.0f;
```

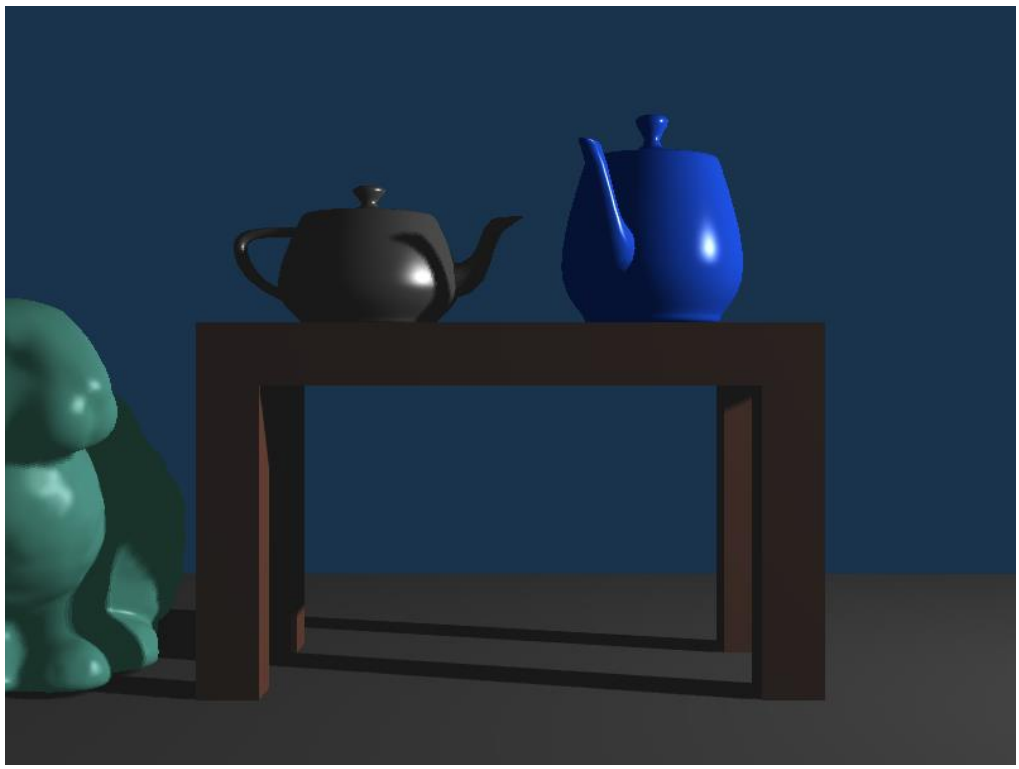
After all this, the shadows are still blocky and pixelated, hence we use a method known as PCF (explained earlier) to soften the shadows. This is done by sampling the surrounding texels of the depth map and averaging the results. In our case, we choose to sample 9 values around each coordinate, test for shadow occlusion, and average by the number of samples.

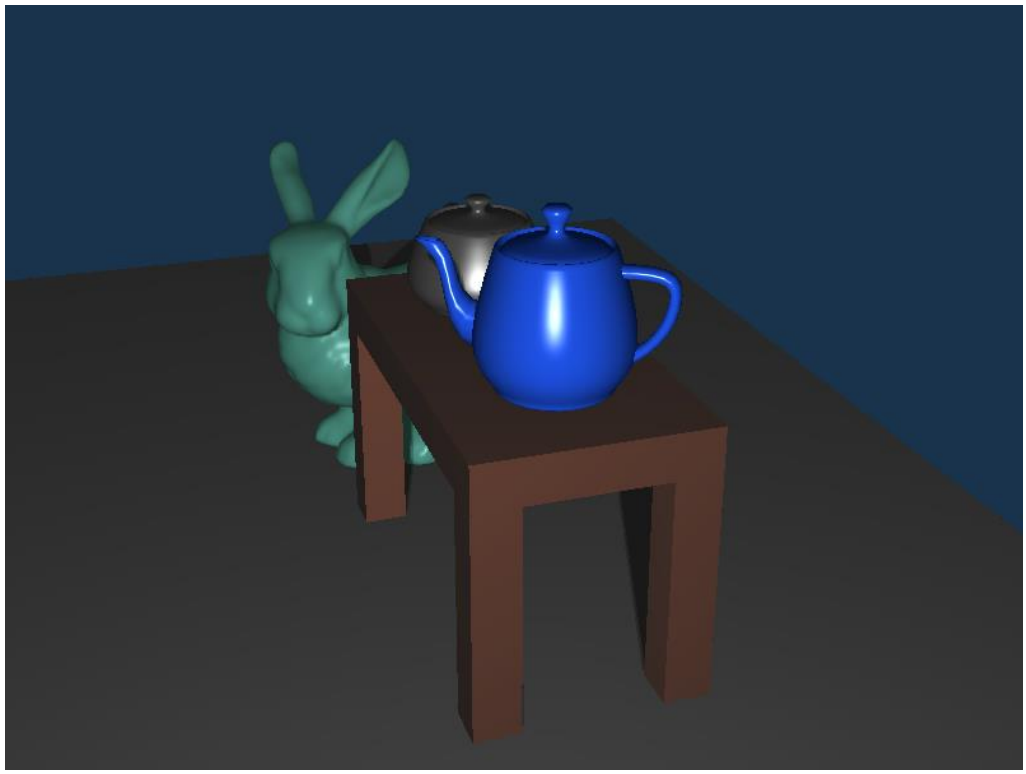
```
vec2 texelSize = 1.0f / textureSize(shadowMap, 0);
float shadow = 0.0f;
for (int x = -1; x <= 1; x++) {
    for (int y = -1; y <= 1; y++) {
        float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize).r;
        shadow += currentDepth - bias > pcfDepth ? 1.0f : 0.0f;
    }
}
shadow /= 9.0f;
```

With all that done, our final results looks like what is shown in the next section.

## Demonstration (Compilation of the final scene from different angles):

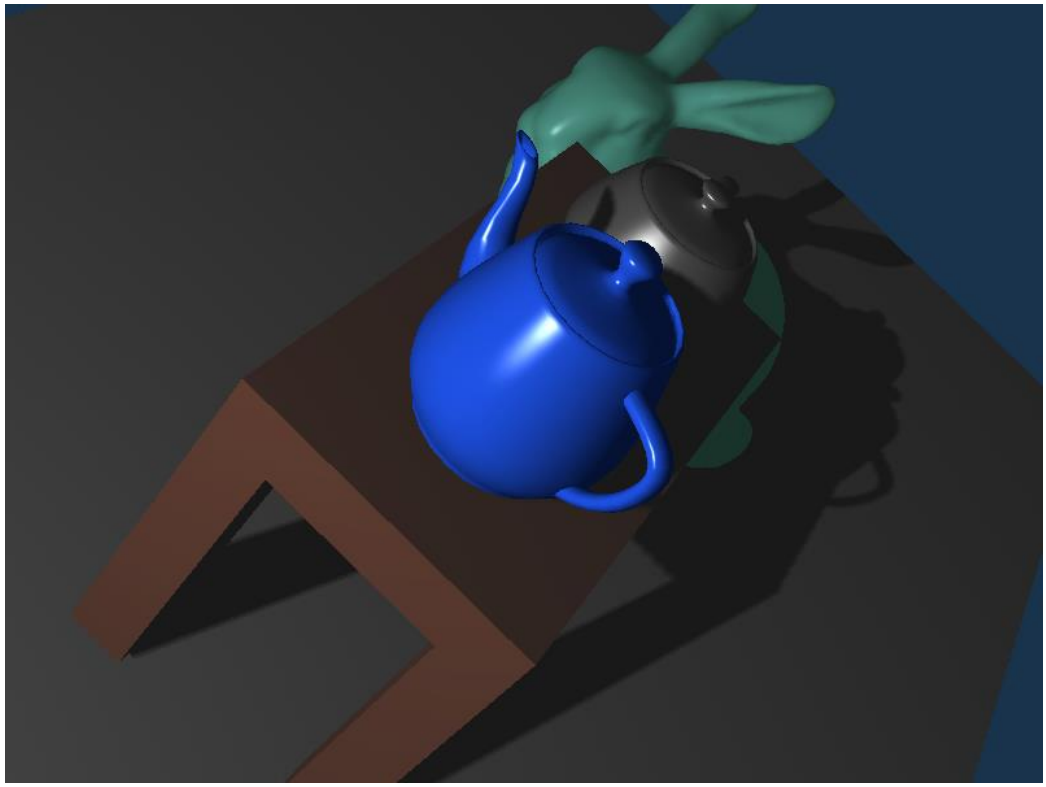
Scene from default camera angle and light source's angle respectively:

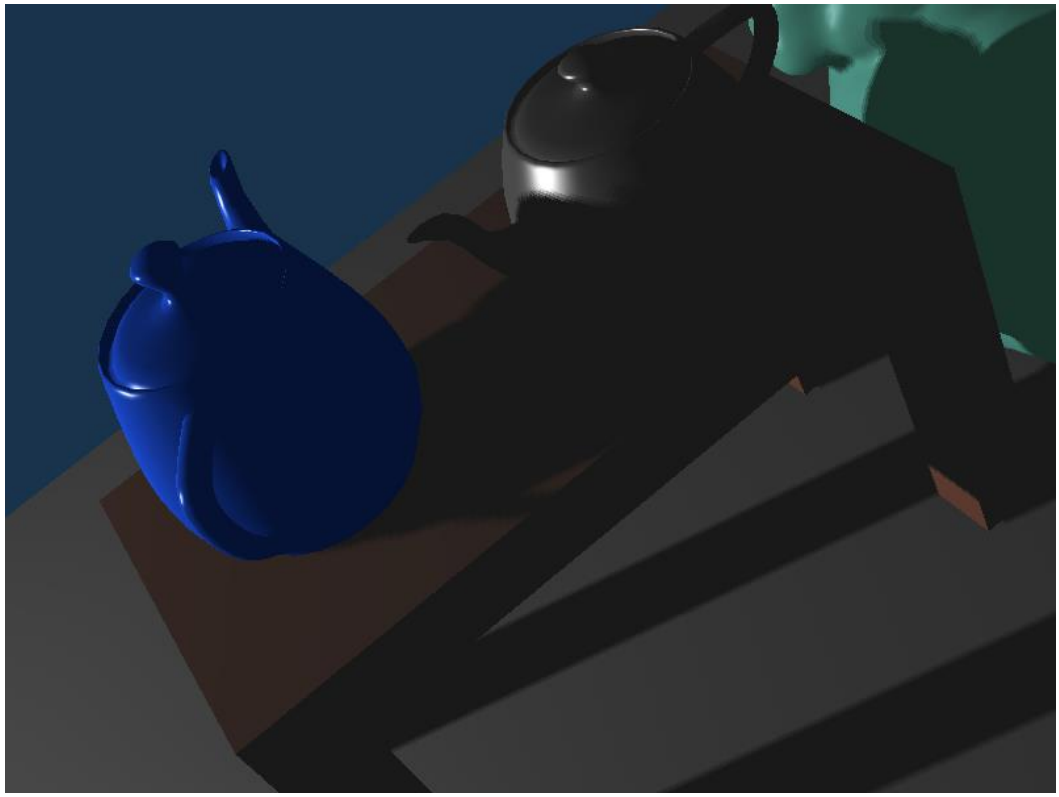




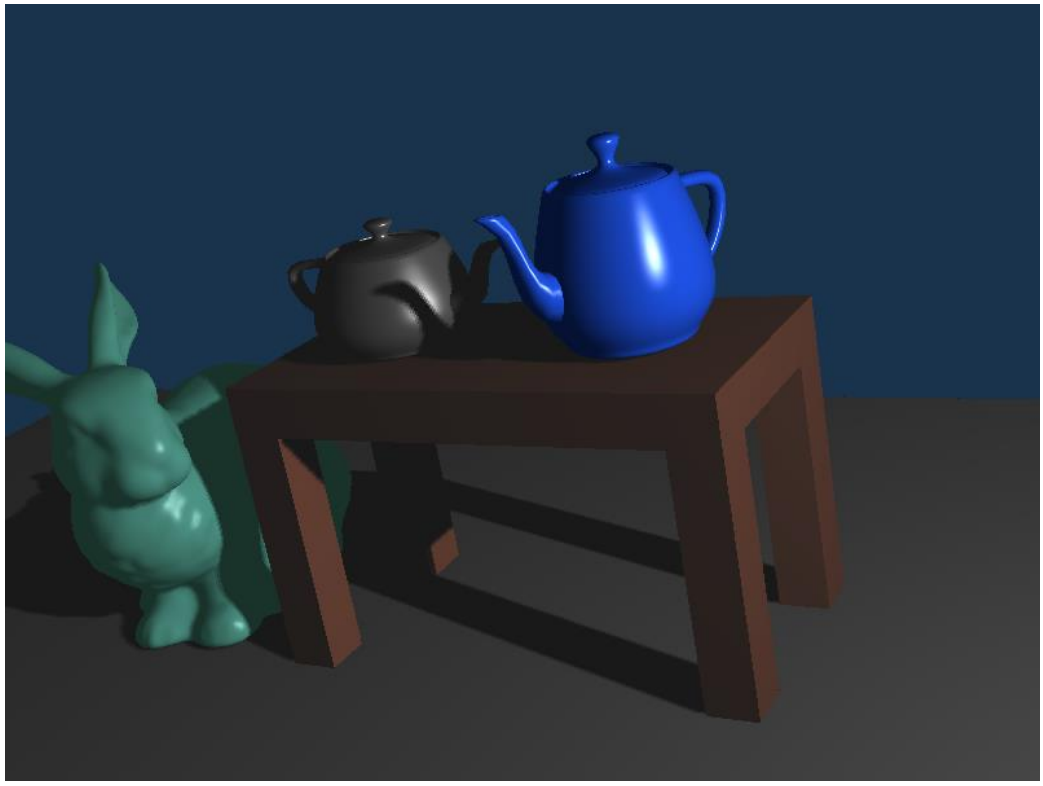
Scene from top view:

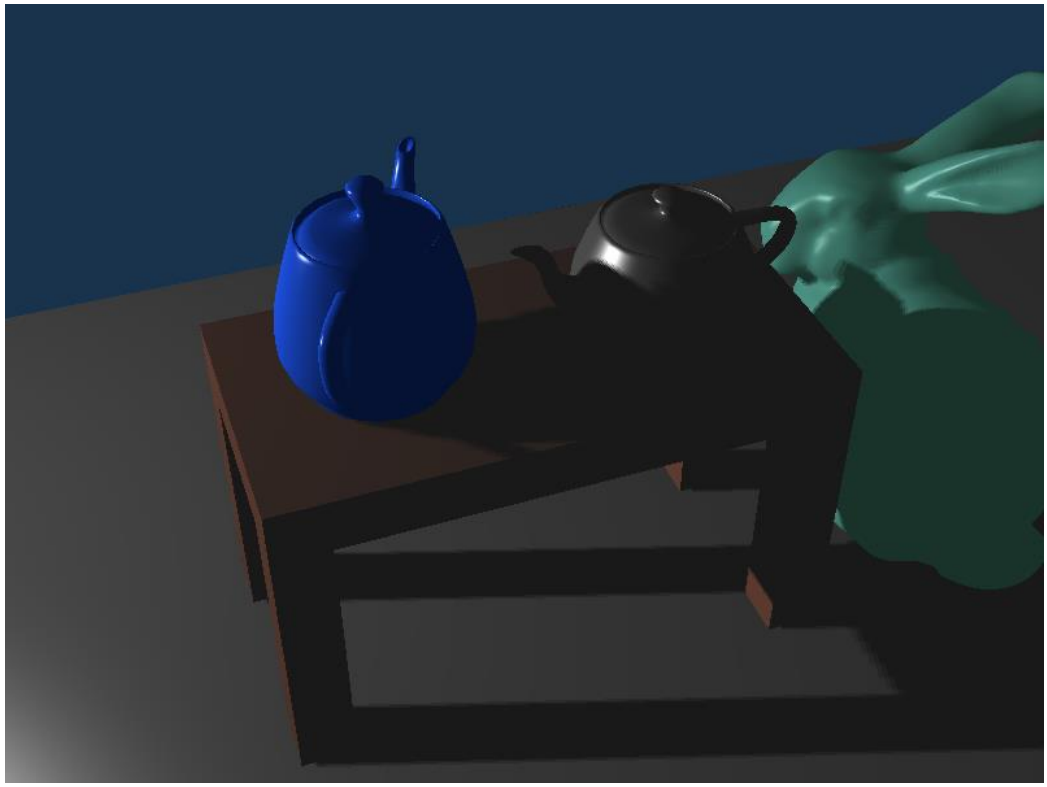
Scene from side angle (1):





Scene from side angle (2):





Scene from further away: