UNIVERSITÄT BERN

Introduction to Computer Graphics
Assignment 7 – Shadows and Cube Mapping

Handout date: 8.11.2019

Submission deadline: 15.11.2019 12:00

Late submissions are not accepted

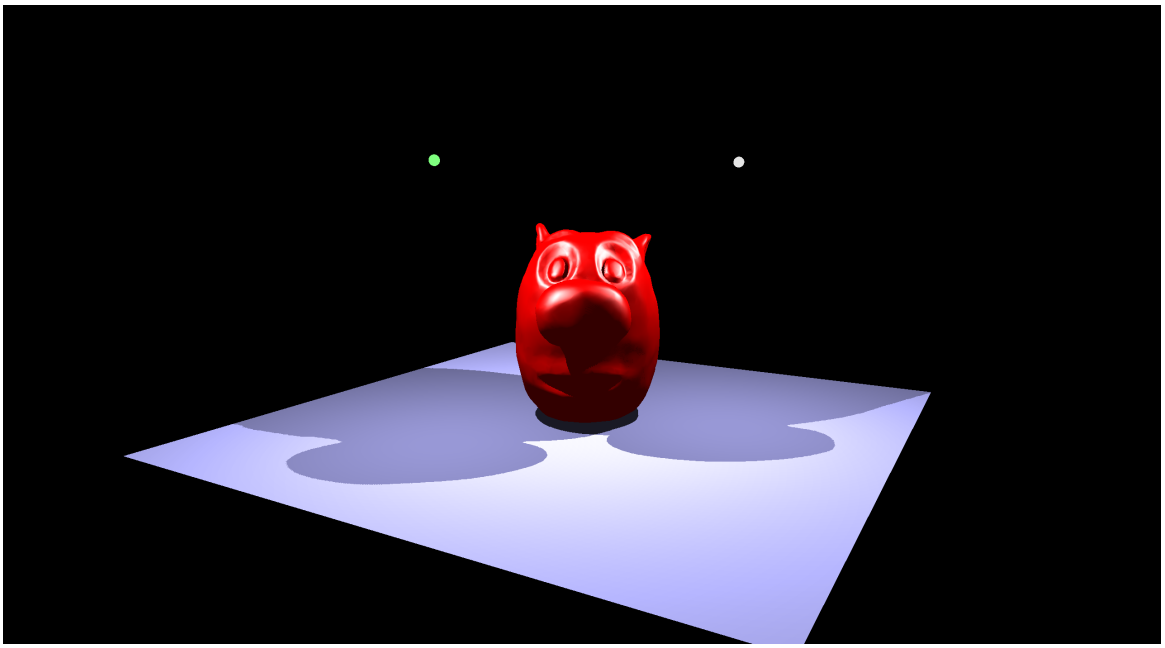


Figure 1: The mesh `neutral.off` rendered with two lights.

In this assignment, you will implement a real-time shadowing technique for multiple omni-directional point lights. We will be using a new framework code, but it should be familiar to you: it's built on the same libraries as the solar system codebase.

Before getting started, you'll want to copy your per-vertex normal computation from Assignment 3 into `Mesh.cpp` and your normal transformation matrix construction from Assignment 6 into `ShadowViewer.cpp`.

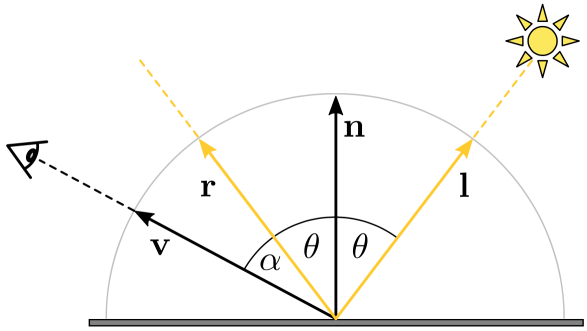
Overview

This exercise consists of two main parts:

1. Constructing view and projection matrices for rendering each light's shadow map.
2. Writing the GLSL shaders to draw the shadow maps and accumulate the diffuse and specular light contributions to the rendered scene.

Review on Phong Lighting and Shadows

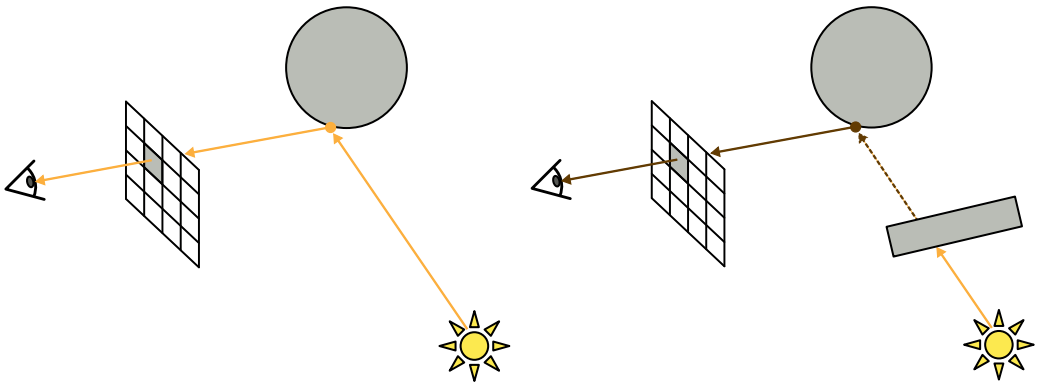
First, recall the Phong illumination model you've used in the previous assignments:



$$I = I_a m_a + \sum_l I_l m_d (\mathbf{n} \cdot \mathbf{l}) + I_l m_s (\mathbf{r} \cdot \mathbf{v})^s, \tag{1}$$

where the final fragment's intensity, I, is computed adding specular and diffuse contributions from each light, l, atop the ambient contribution. Here I_a is the ambient light intensity, I_l is diffuse/specular intensity of light source l, $m_{[a|d|s]}$ is the ambient/diffuse/specular component of the material, s is the shininess, and $\mathbf{n}, \mathbf{l}, \mathbf{r}, \mathbf{v}$ are the normal, light, reflected light, and view vectors respectively.

Remember that, in the second raytracing assignment, you computed shadows by neglecting the contributions from lights that are obscured by other scene geometry. To do this, you cast a *shadow ray* from the point being lit, "$\mathbf{p}$," to the light to check if there was an intersection between the two.



A slightly different formulation of this test, which we will see is more compatible with the OpenGL rasterization pipeline, is to instead cast a ray from the light toward the point and check if there is any intersection *closer* to the light than $\mathbf{p}$ is. If so, then $\mathbf{p}$ is in a shadow, and no diffuse or specular components from this light should be added.

Real-time Shadows with Shadow Mapping

The nice thing about this new shadow test formulation is that determining the first intersection of a whole frustum of rays with a scene is exactly the problem OpenGL rasterization solves: we can simply render the scene from the perspective of the light to determine the distance of the closest intersection along every light ray. The resulting grayscale image of distance values is called a shadow map:

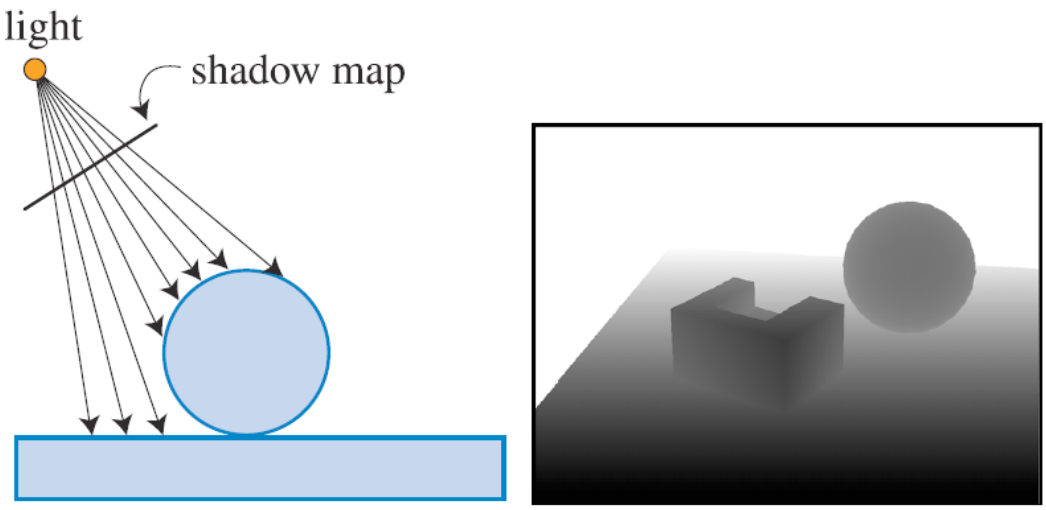


Figure 2: from Akenine-Möller, "Real-Time Rendering"

Then, when computing the lighting for point $\mathbf{p}$, we can compare its distance from the light against the corresponding ray's intersection distance value as read from this shadow map. This test involves only an efficient, constant-time texture look-up operation.

Traditionally, shadow mapping implementations stored the depth buffer (or z-buffer) value in the shadow map. However, for omnidirectional lighting, it will be simpler to instead compute the Euclidean distance from the light to the intersection point in a Fragment shader.

With this approach, we need the following multi-pass rendering process:

for all rasterized fragments "$\mathbf{p}$" **do**
  $I(\mathbf{p}) \leftarrow$ `ambient_contribution`
end for
for lights l in the scene **do**
  Draw the shadow map for l by computing distances to each fragment seen by l
  for all rasterized fragments "$\mathbf{p}$" **do**
    if `length`$(\mathbf{p} - l.\mathtt{pos}) <$ `shadowmap_depth` **then**
      $I(\mathbf{p}) \leftarrow I(\mathbf{p) +}$ `diffuse_specular_contribution`
    end if
  end for
end for

In our implementation, the loops over rasterized fragments will be executed by the fragment shaders `shaders/solid_color.frag` and `shaders/phong_shadow.frag`. The shadow map distance values are written by `shaders/shadowmap_gen.frag`.

Cube Mapping

Because we are working with *omnidirectional* point lights, our situation is a bit more complicated. It's not possible to set up a single view frustum to render the light rays emanating in all directions. Instead, we will set up 6 different view frustums—one for each face of an imaginary cube surrounding our light—and take advantage of the GPU's cube-mapping functionality.

One such frustum together with its corresponding shadow map is depicted in Figure 3. Please note that, when the eye is placed inside the light cube to render the shadow map, we will refer to it as the **light camera** to distinguish it from the viewpoint used to render the image on screen.

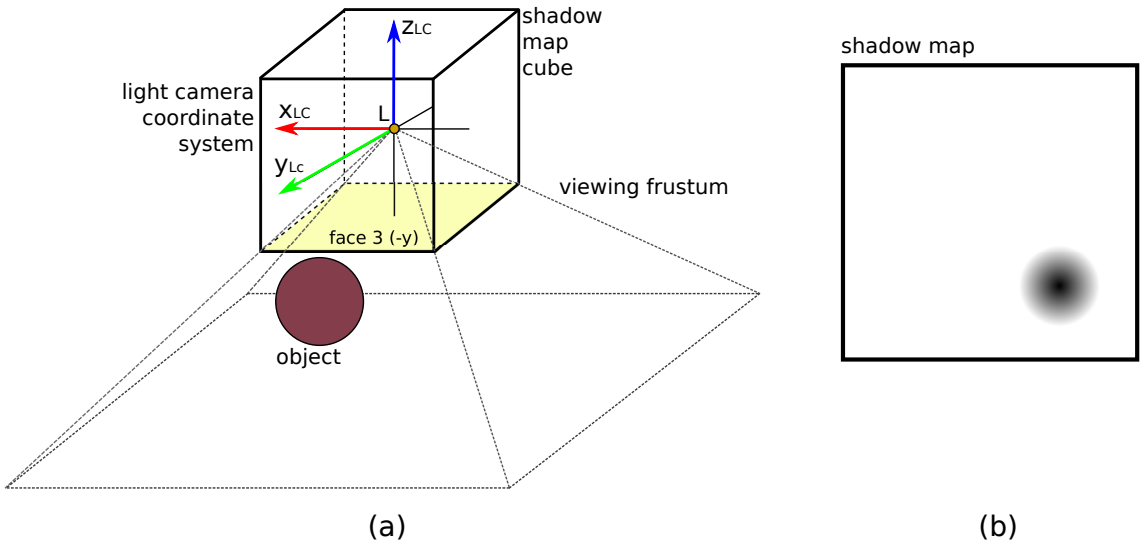
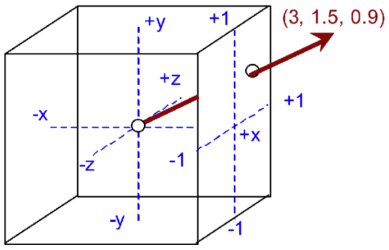


Figure 3: (a) The shadow cube map for the omnidirectional point light source $\mathbf{L}$. It is necessary to render a separate shadow map texture for each of the six cube faces. Here, the setup for rendering `face` 3's map is visualized. The coordinate axes indicate the orientation of the light camera used to render the shadow map; the camera looks along the -z axis. (b) The resulting shadow map texture (darker is closer).

A cube map texture is really a collection of 6 textures that are conceptually attached to the faces of a cube. Instead of sampling this cube map with 2D texture coordinates (s, t), you sample it with a 3D vector (s, t, r); the GPU then returns the color of the point on the cube where this 3D vector pokes through:



This sampling mechanism is a perfect fit for our shadow mapping problem: if we sample the cube map with the shadow ray vector pointing from the light to $\mathbf{p}$, it will pull out the intersection distance value for this ray (as long as we correctly rendered each face's shadow map).

The 6 texture images making up the cube map are oriented as follows:

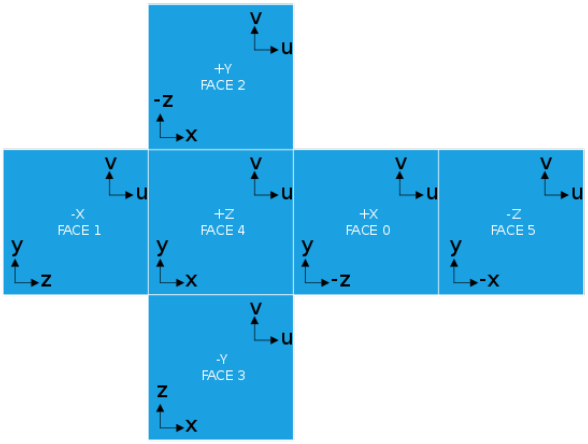


Figure 4: Source: Wikipedia cube mapping article

They are wrapped around the cube as shown in Figure 5. An example of how these shadow maps will be drawn is shown in Figure 3.

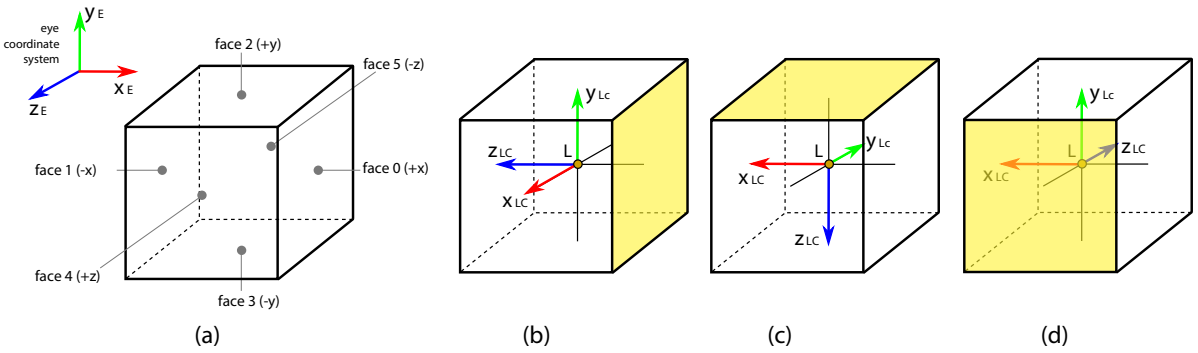


Figure 5: (a) Names of the shadow map cube faces as used in this assignment. The example of the light camera coordinate system oriented so as to render the face 0 (b), face 2 (c) and face 4 (d).

Building View and Projection Matrices for a Cube Face

The most challenging part of this assignment is to construct the proper view and projection matrices for rendering the shadow maps on each face of the cube.

Start by implementing the light projection (`light_proj`) matrix in the function `ShadowViewer::m_constructLightProjectionMatrix()`. Please note that the shadow map cube model assumes the light lies at the center of the cube. Given such a geometry, think of what aspect ratio and field of view properly define the light camera's view frustum as visualized in Figure 3. The near/far parameters control the minimum and maximum distances at which you can compute shadow ray intersections. For this scene, you can use 0.1 and 6 respectively.

Next, implement the `light_view` matrix construction for a given light and cube face by editing the function `ShadowViewer::m_constructLightViewMatrix`. This matrix specifies the transformation from the world coordinate system to the coordinate system for the light camera, which coincides with the specified light

and looks through the given cube face.

Please note that the orientation of the shadow map cube itself (i.e., the shadow map cube coordinate system) must always be aligned with current eye coordinate system (see Figure 6 (a)). This will allow you sample the cube map texture directly with the light vectors calculated in your Phong shader (since your lighting calculation is done in eye space).

Therefore, when setting up the `light_view` matrix, you will need to use the current eye's view matrix (i.e. the transformation from the world coordinates system to the eye coordinate system) `scene_view_matrix`, which is provided for you. You will also find the function `mat4::look_at` helpful to point the camera through the specified cube face. See Figures 5 (b-d) for example light camera orientations.

After you've set up the view and projection matrices, and before you move on to the fragment shaders, we recommend that you test your matrices using the debugging visualizations described at the end of this document. By repeatedly pressing the 'F' key, you should cycle through the view seen by light camera for each cube face. If you press 'C', the correct face of the visualization texture should fill the screen (though it will be flipped left-right since we're viewing it from inside).

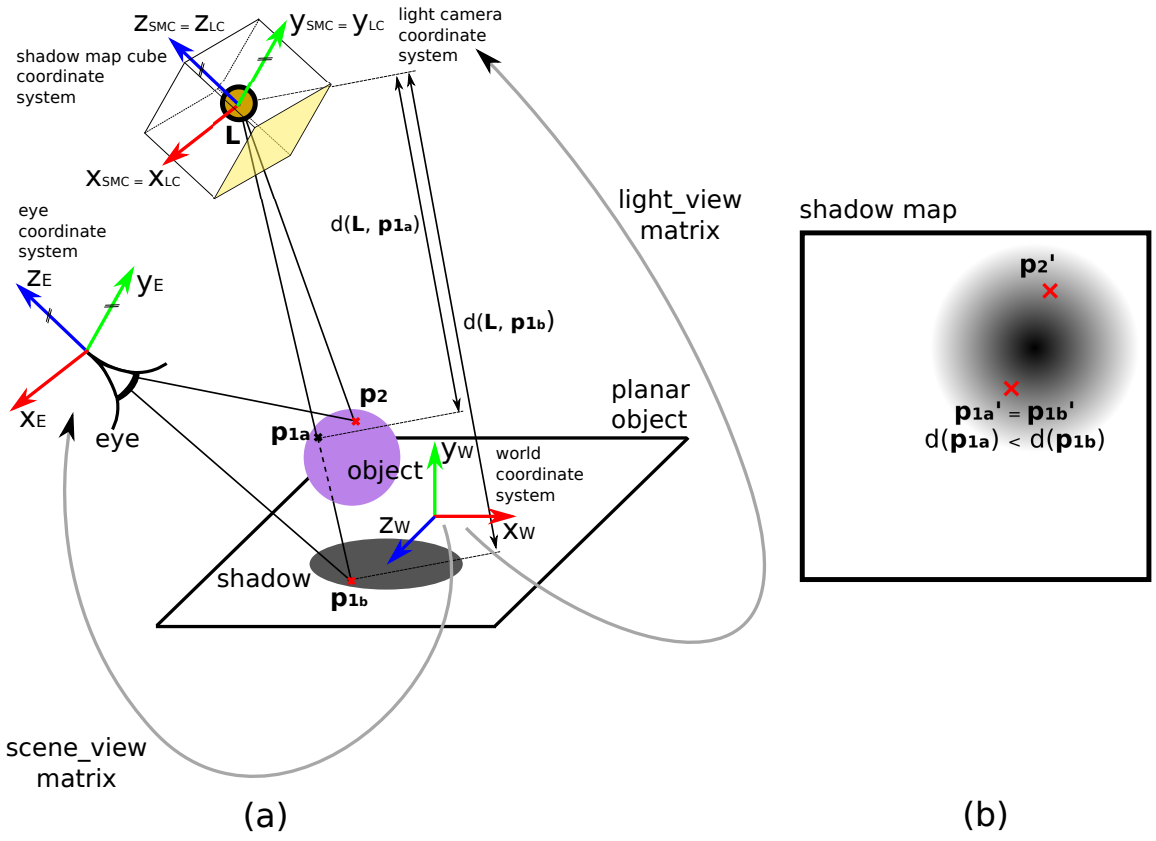


Figure 6: (a) A scene with one sphere, one plane, one light source $\mathbf{L}$, and the eye. Here, we want to render two fragments corresponding to the points $\mathbf{p_{1_b}}$ and $\mathbf{p_2}$. Note that $\mathbf{p_{1_b}}$ is not illuminated by $\mathbf{L}$: it lies in the shadow cast by the spherical object. Note that the orientation of the shadow map cube is aligned with the eye coordinate system. Also note that this figure depicts the scenario where we render/query the shadow map for the cube `face` 5; the light camera coordinate system is properly aligned with the shadow map cube to render this shadow map. (b) A distance map corresponding to shadow map cube face 5. Note that both points $\mathbf{b_{1_a}}$ and $\mathbf{b_{1_b}}$ project to the same 2D point $\mathbf{p_{1_a}}' = \mathbf{p_{1_b}}'$ in the shadow map but their distances from the light source $\mathbf{L}$ differ, i.e. $\texttt{length}(\mathbf{p_{1_a}} - \mathbf{L}) < \texttt{length}(\mathbf{p_{1_b}} - \mathbf{L})$. In this case, the shadow map should store only the smaller distance.

Writing the Fragment Shaders

Follow the `todo` comments in the `shadowmap_gen.frag` and `phong_shadow.frag` shaders to fill in the missing parts needed to write the ray intersection distances to the shadow map texture and compute the diffuse and specular contribution for the unshaded fragments. Note that `shadowmap_gen.frag` should write the *Euclidean distance*, not the depth coordinate, to the fragment.

Setting the Blend Function

We need each iteration of our loop over the lights to *add* to the rendering, not overwrite it. This can be accomplished by enabling blending (like in the last assignment) and appropriately configuring the blend function. Study the reference page for `glBlendFunc` to understand how to do this.

Debugging, Keyboard Interface, and Screen Shots

To help you debug your view/projection matrices and distance calculations, we provide a visualization of the cubemap. By pressing the '`C`' key, you can cycle between showing the cube corresponding to one of the lights, with either a demo cube map texture or the shadow map you compute. By pressing the '`F`' key, you can cycle between rendering the scene from the 6 cube face perspectives to test the `light_view` and `light_proj` you computed. By combining these two features, you can directly inspect from inside the shadow map cube the distance textures you draw.

We also have prepared some light and camera configurations for you. Pressing keys 1-3 will switch between these pre-set scenes.

Finally, the plus and minus keys add and remove lights, the "WASD" keys position the currently selected light, the tab key cycles the selection through the lights, the arrow keys rotate the view, and 8/9 zoom in/out.

Pressing the '`P`' key writes the current view to '`screenshot_#.png`'.

If your implementation is correct, you should see the result shown in Figure 7 when you load `meshes/neutral.off`, press the 1 key, and press the C key twice.

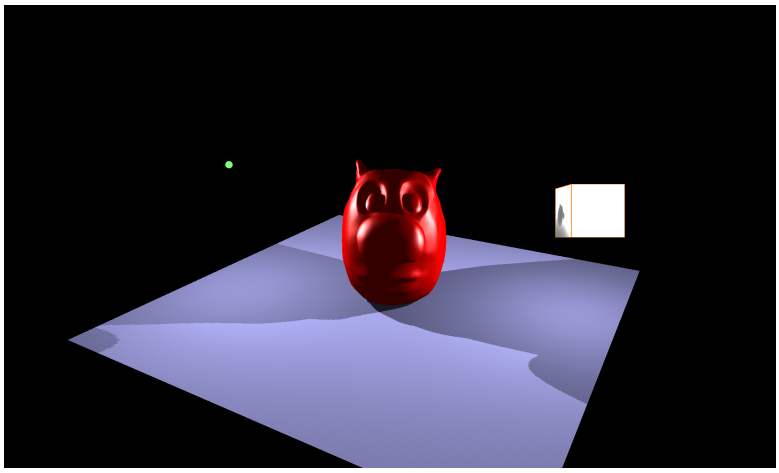


Figure 7: Expected result for `neutral.off` with scene preset 1 and showing the shadow cubemap visualization.

Grading

The scores for this assignment are broken down as follows:

- 20%: Building the projection matrix for rendering the shadow maps
- 30%: Building the view matrix for each face of the shadow cube map
- 10%: Setting the blending mode so that each diffuse/specular lighting pass *adds* to the output color.
- 10%: Completing `shadowmap_gen.frag` to draw the distance map for a light
- 30%: Completing `phong_shadow.frag` to render the diffuse and specular contributions to unshaded fragments only.

what to hand in

A compressed .zip file with the following contents:

- the files you changed (in this case, `mesh.cpp`, `shadowviewer.cpp`, `shadowmap_gen.frag`, and `phong_shadow.frag`). If you change any other files (should not be necessary), include them as well!
- screenshots for the model `confused.off` from view/light configuration preset 1, 2, and 3 while displaying your shadow map on the visualization cube.
- a `readme.txt` file containing:
  - the full names of every group member
  - a brief description on how you solved each exercise (use the same numbers and titles)
  - which, if any problems you encountered