Programming Assignment #5

CS 541, Fall 2018

Due Thursday, December 6

In this assignment, you will complete your local illumination model by including shadows. Shadows should be implemented using the shadow map technique discussed in class.

The requirements of the previous assignments are still in place: at least one rotating object, a user–movable camera, a movable light source, local illumination (ambient light, diffuse reflection, and specular reflection), textured and non–textured objects, and a sky dome.

You are to add shadows from a single light source by implementing the two–pass shadow map algorithm. The light source may be either a spot light (with a direction and beam size) or a point light source. For a spot light, you may use a rectangular or circular beam. For a point light source, you can cheat by illuminating all objects that are outside of the shadow map — although you will want to use a larger field–of–view angle for the shadow map.

What to submit. Zip up your source code into a single zip–file (you can name the file as you please), and upload the file to Moodle. Do not include the files that I give you (`Affine.h`, `Camera.h`, et cetera).

Implementation details

Here are some programming details that you will need to implement the shadow map.

Storing depth values as a texture

For the shadow map, you will need to use a frame buffer object to create a texture that stores depth information. The demo program 'cs541fbo-demo' discussed in class illustrates how to create and use a frame buffer object. The demo creates both a color buffer and a depth buffer, which were then bound to the frame buffer object. However for this assignment, we only need the depth buffer values — no color buffer. This is actually somewhat simpler usage of a frame buffer object.

We need to create a texture buffer and a frame buffer object that writes the depth values to a texture buffer.

```
GLuint frame_buffer,
       depth_texture_buffer;
```

The frame buffer object is created and bound as usual:

```
glGenFramebuffers(1,&frame_buffer);
glBindFramebuffer(GL_FRAMEBUFFER,frame_buffer);
```

However, we need to create a texture buffer that will store depth values instead of color values:

```
glGenTextures(1,&depth_texture_buffer);
glBindTexture(GL_TEXTURE_2D,depth_texture_buffer);
glTexImage2D(GL_TEXTURE_2D,0,GL_DEPTH_COMPONENT,1024,1024,
             0,GL_DEPTH_COMPONENT,GL_FLOAT,0);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
```

This will create a texture of size 1024×1024 that stores depth values. We then simply attach this texture buffer to the frame buffer object:

```
glFramebufferTexture2D(GL_FRAMEBUFFER,GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D,depth_texture_buffer,0);
```

Vertex and fragment shaders

To render depth buffer to the texture, we use shaders tailored to the task. Since we do not need any lighting, the vertex shader only needs the to use the mesh vertices as input:

```
attribute vec4 position;
```

and the transformations needed to render the scene from the point of view of the light source:

```
uniform mat4 persp_matrix;
uniform mat4 view_matrix;
uniform mat4 model_matrix;
```

Here `view_matrix` maps from world space to camera space, using a camera with center of projection at the light location. For a spot light, the camera look–at vector is equal to the direction that the light is pointing. For a point light source, you will need to choose a look–at vector that is reasonable for your scene. In either case, you will also need to choose a (relative) up vector for the camera. You will also need to choose reasonable values for the field–of–view angle, aspect ratio, and the near and far distances to use for `persp_matrix`. The vertex shader has no explicit output, and the program only needs to compute the position of a mesh point in NDC clip coordinates and store the value in `gl_Position`.

The fragment shader is even simpler: it does nothing; i.e., has a trivial `main` function! Note that even though `main` is trivial, the depth value will be computed from the value of `gl_Position` in the vertex shader and stored in the depth buffer.

First pass: drawing the shadow map

As usual with the frame buffer object, you will first need to select the frame buffer object, and set the viewport to the depth buffer texture size:

```
  glBindFramebuffer(GL_FRAMEBUFFER,frame_buffer);
  glViewport(0,0,1024,1024);
```

(or whatever size you chose as the depth buffer texture). Then select the shadow shader program, clear the depth buffer, and draw the meshes as usual. Of course, you need to use the camera that represents the point of view of the light. Note that you only need to draw those meshes that will cast a shadow. The sky box/dome does not need to be drawn.

Once you are done rendering the shadow map, do not forget to select the default frame buffer (the window you are rendering to)

```
  glBindFramebuffer(GL_FRAMEBUFFER,0);
```

and reset the viewport to the window size.

Second pass: using the shadow map

You will need to add some additional code to your vertex and fragment shaders from the previous assignment.

For the vertex shader, you will need to compute the position of a point on the mesh *from the point of view of the light source,* and output this value to the fragment shader. Remember that the shadow map stores the depth values *in a texture*. Depth values are stored in the range $[0, 1]$ (just like RGBA values), and the depth map values are accessed using texture coordinates, also in the range $[0, 1]$. On the other hand, the perspective matrix Π maps to the *standard cube* $[-1, 1] \times [-1, 1] \times [-1, 1]$. So we need to map the standard cube to the *unit cube* $[0, 1] \times [0, 1] \times [0, 1]$. That is, the vertex shader needs to apply the transformation

$$B \circ \Pi \circ M_c^{-1} \circ M_o \tag{1}$$

to a point on the mesh. Here M_o is the modeling (object–to–world) transformation, M_c^{-1} is the viewing (world–to–camera) transform, Π is the perspective (camera–to–NDC) transform, and B maps the standard cube to the unit cube:

$$B = T_{\langle 1/2,1/2,1/2\rangle} \circ H_{1/2}$$

Of course, the viewing transformation uses the camera that represents the point of view of the light source.

Remark: the matrix B is only needed for the second pass. In the first pass (shadow map construction), we use the composition $\Pi \circ M_c^{-1} \circ M_o$ to render the scene from the point of view of the light. OpenGL automatically converts the depth values to the range $[0, 1]$ when it stores the depth values in the shadow map texture.

Practically speaking, for the vertex shader you will need to add a uniform variable, or variables, for the transformation in equation (1). And you will need to add an output variable for the result of applying (1) to a mesh point. E.g.,

```
  out vec4 vshadow_position;
```

Remark: you can define the matrix B as a constant in the shader, but if you do so, you will have to store the matrix in *column–major order* (since this is what is used by GLSL).

For the fragment shader, you will need to add an input variable for the computed shadow map key value

```
in vec4 vshadow_position;
```

You will also need to add a sampler for the shadow map texture (this is in addition to, and different from, the sampler needed for texture mapping of objects). E.g.,

```
uniform sampler2D shadow_sampler;
```

Remark: to use the shadow map with a textured object, we need to use two textures: one for the texture used by the object, and one for the shadow map texture. We need distinguish which sampler uses which texture. This is done by assigning a *texture unit* number to the sampler. We also need to tell OpenGL which texture is uses which texture unit. For example, if we want the shadow map texture to use texture unit number 1, then we would write

```
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D,depth_texture_buffer);
GLint location = glGetUniformLocation(program,"shadow_sampler");
glUniform1i(location,1);
```

where `program` is the name of the shader program for rendering textured objects. You will also need to do this for the texture used by the object, although with a different texture unit number.

In the fragment shader program itself, we need to use the input shadow key value `vshadow_position` to look up the shadow depth value for at the fragment location, and compare it to the fragment location depth value. The texture coordinate for the shadow map is

```
vec2 uv = vshadow_position.xy/vshadow_position.w;
```

We need the perspective divide here since the transformation (1) is a perspective transformation. We use this value to look up the shadow depth value for this location:

```
float shadow_depth = texture(shadow_sampler,uv).z;
```

(the '.z' here is arbitrary, using any other component, say '.x' or '.r', will return the same value). And the fragment location's depth value is

```
float point_depth = vshadow_position.z/vshadow_position.w;
```

A fragment is in shadow if its depth value is greater than the shadow depth value; i.e. if

```
shadow_depth < point_depth
```

However, we do need to be careful. The fragment can be behind the light, or outside of the beam of light. The fragment is inside the light beam if

```
vshadow_position.w > 0
```

and the values of `uv` are in the range $[0, 1]$. Only if the fragment is inside the light beam can it be in shadow.

Shadow acne

Due to finite numerical precision, if the fragment depth close to the shadow depth, the comparison

```
shadow_depth > point_depth
```

can result in a value of true, even if the fragment depth is slightly less than the shadow depth. For points on a mesh where this false positive occurs, there will be a noticeable dark dot where where there should not be one.

There are several strategies to remove this "shadow acne" (yes, this is the common term for this effect). One solution is to perform front face culling for meshes in the first (shadow map construction) pass. However, this will only work for meshes that represent solids with nonzero volume. This technique will not work for the `Plane` mesh unless you render a second copy of the plane that is offset from the original by a small amount and has the opposite orientation. Another technique is to "push" the fragment location slightly in the direction of the light source. That is, we decrease the fragment depth value by a small value:

```
point_depth = point_depth - epsilon;
```

I'll leave it to you to find a reasonable value of `epsilon` if you use this technique.