Graphics Programming HW 4

Due: 5/2 11:59 PM

In this homework, you will implement the algorithms about lighting. Because there is several new concepts that you may be unfamiliar with, you are **strongly recommended to read** LearnOpenGL's lighting, model loading, normal mapping and shadow part. You can find answers to most problems. You are allowed to copy and paste LearnOpenGL's code, but it would be a challenging task to integrate each part into the single application.

Prob 0. Loading Wavefront .obj Files (10 pt)

So far, we've extensively relied on quad or cube geometry. However, in practical graphics applications there are a lot of complicated objects. Our goal here is to make use of existing fancy models. Although there are dozens of different file formats, we will start from the simplest case, wavefront obj file. Wavefront obj file contains model data with minor material information like model colors and diffuse/specular maps. Instead of implementing obj loading from scratch, we will use a popular model importing library Assimp which stands for *Open Asset Import Library*. Assimp can handle various file formats besides obj. It loads all the model's data, such as position, normal and uv, into its own data structure. We can retrieve all the data we need from Assimp's data structures. We do not discuss the details of obj file format or Assimp's data structure. Please refer to Wikipedia or LearnOpenGL.

We provided complete code for obj file loading system to reduce your burden. Basically the given code is the same as LearnOpenGL's model loading code. A model is composed of polygonal meshes and several textures. A mesh contains vertex/face data such as position, normal, uv or face indices. Model can also have different textures (role of these textures will be explained further in next questions) besides diffuse map that we have used so far. The code is slightly modified from the original LearnOpenGL code in:

- General obj files can contain multiple, hierarchical models. However, we will consider only the case with a single model. The code is modified to only load the first model of obj file.
- For the original LearnOpenGL code, obj loading also includes texture loading. However, for simplicity and scalability we've detached it.
- Using this data structure, we can represent scene objects in more organized way. For each model, there can be many copies of it. We call a copy *entity*. Model/entity relationship is similar with C++'s class/instance relationship. Check the skeleton code for the details.

You have to check whether the given obj loading code works well and load your own model. We provided several models (brickcube, barrel, boulder, and fire extinguisher), but please find additional models and load them. We recommend Free3D or Turbosquid. You can find a bunch of free models. Find what you want, and import it into the application. Please make sure that the model's format is obj. There are no limit on the number of additional models, but please add at least one new model.

Prob 1. Phong Illumination (50 pt)

In this problem, you will implement the Phong illumination model using shaders. In the Phong model, there are three components of lights (ambient light, diffuse light, and specular light) and the illumination on a point on the surface is the sum of them. The ambient light is the illumination inherent to the surface regardless of the incoming light. Diffuse light, also referred as Lambertian reflectance model, assumes uniform surface reflectance in any direction. Specular light is mirror-like specularity, which is stronger when the viewing direction is close to the direction of reflection of the light (R).

Following equations represent the Phong reflection model for a single light source:

$$\begin{aligned} I_{\text{ambient}} &= k_a I_a \\ I_{\text{diffuse}} &= k_d I_d (N \cdot L) \\ I_{\text{specular}} &= k_s I_s (R \cdot V)^n \\ I &= I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}}. \end{aligned} \tag{1}$$

k_a, k_d, and k_s are reflectance coefficients, I_a, I_d, and I_s are the incident light intensity components, and n is the shininess of the surface. N, L, R, and V are 3-dimensional unit vectors, and they are represented in Figure 1.

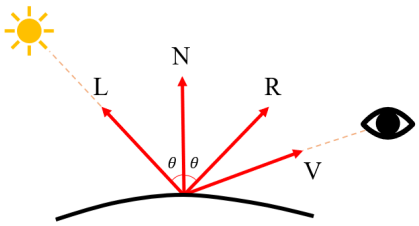
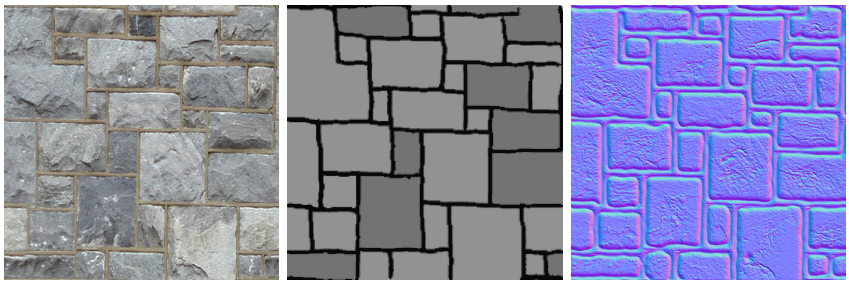


Figure 1: N is the normal vector of the surface, L is the vector pointing to the light source, R is the reflected vector of L by N, and V is the vector pointing to the camera (eye). All of them are unit vectors.

You will implement the Phong illumination with diffuse map and specular map specifically designed for calculating the illumination, instead of setting k_d and k_s to a constant value. You can find more details in LearnOpenGL lighting map part. We provided the texture files and you should use the diffuse map and the specular map as reflectance coefficients for calculating the each component of the illumination. You can use `Model` class to manage these textures in an organized way. First, load the obj file with `Model("example.obj")`. Then attach diffuse map and specular map to the model. You can find an example code in the skeleton code. Figure 2 (a) and (b) show the provided maps. Assume that $k_a = k_d$, $I_d = I_s$ and $I_a = 0.3 I_d$. Also only consider the red channel of the specular map as a reflectance coefficient. Note that I_{ambient}, I_{diffuse}, and I_{specular} should not be negative for all channels, and you should consider the specular light only when the diffuse light exists.

Throughout this assignment(prob 1,2,3 and 4), we only consider a directional light (sun). Make it possible to change the direction of the light. Assume that the sun is on the celestial sphere. Its position can be represented with azimuth ($0 \sim 360°$) and elevation ($0 \sim 90°$). **Increase or decrease azimuth and elevation of the sun, when we press up/down/left/right arrow key.** Limit elevation range to be within the range from 15° to 80°, because shadow in Problem 3 may not work well with extreme elevations.



(a) (b) (c)

Figure 2: (a): Diffuse map (b): Specular map (c): Normal map

Prob 2. Normal Mapping (35+5 pt)

This problem is an extension of Problem 1, and you will add the normal mapping in your rendering. Here, we will just briefly explain normal mapping. You can find more details in LearnOpenGL normal mapping part. Normal mapping is a widely used technique to show a detailed appearance without adding more polygons. The main idea is to change the surface normal without changing the geometry. By using a normal map, we can easily replace the normal vectors with new vectors. Normal map is a texture with RGB channels. The new normal vectors can be calculated as a linear combination of tangent, bitangent and existing normal vectors, and the coefficients are made from the pixel values of the normal map. Note that the channel values of the sampled pixel in a normal map are in the range of $[0, 1]$ but you should make them in the range of $[-1, 1]$ to use them as the coefficients. Figure 2-(c) shows the normal map we provided (Can you guess why it is bluish?).

Tangent, bitangent, and existing normal vectors are orthonormal. Thus, tangent and bitangent vectors can be any two orthogonal vectors lying on the surface. To specify the two vectors, you should find the object space representation of the uv axes of the normal map. The tangent vector is defined as the vector corresponding to the u-directional unit vector, and the bitangent vector is defined as the vector corresponding to the v-directional unit vector. Figure 3 shows the tangent, bitangent, and normal vectors.

Assimp automatically calculates tangent and bitangent vector if you enable `aiProcess_CalcTangentSpace` configuration bit. You can just use it. **(Optional)** Or calculate tangent and bitangent from scratch, without using Assimp's functionality. In this case, you can get additional 5 points.

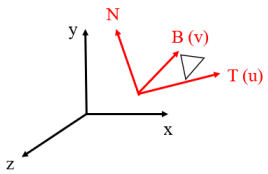


Figure 3: T: tangent vector, B: bitangent vector, N: normal vector

Prob 3. Shadow Mapping (30 pt)

This problem is independent from Problem 2. You will implement the shadow mapping for scene objects. Again we will only briefly explain what shadow mapping is. Please read LearnOpenGL shadow mapping part for the details.

Shadow mapping is a widely used technique for generating the shadow effect. The core idea is that a point is in the shadow when the ray toward the point already intersected with other objects. Shadow map is a texture with one channel and the stored value in each pixel is normalized depth from the light source. It can be created by applying the same rendering pipeline we used, but there are two main differences; the first is that the position of the virtual camera is at the position of the light source, and the second is that we render the scene on a different framebuffer (For details of frame buffer, please read LearnOpenGL framebuffers part.)

So far, we've rendered objects on the default framebuffer, which is composed of RGB buffer and depth buffer (z-buffer). OpenGL gives us the flexibility to define our own framebuffers and thus define our own color, depth buffer. We will render the shadow map on the additional virtual screen (framebuffer) which does not show up on the final screen. For the shadow map, we do not need RGB data, but only depth information, so we only attach a depth buffer to the framebuffer. When creating an attachment we have two options to take: textures or renderbuffer objects. If we use a texture, we can directly use the output of the framebuffer as a texture. If we use a renderbuffer object, we cannot directly read value of the output, but it is more performance friendly. We have to directly read values of shadow map, so we use texture attachment for the depth buffer. Note that the stored values in the depth buffer are in the range of $[0, 1]$ as in the case of the normal map, so when you compare the two depth values, don't forget to synchronize the range of them. Also you have to render new shadow map every frame so that it can handle the change in the light direction. Figure 4 is a visualization of a shadow map in our initial setting.

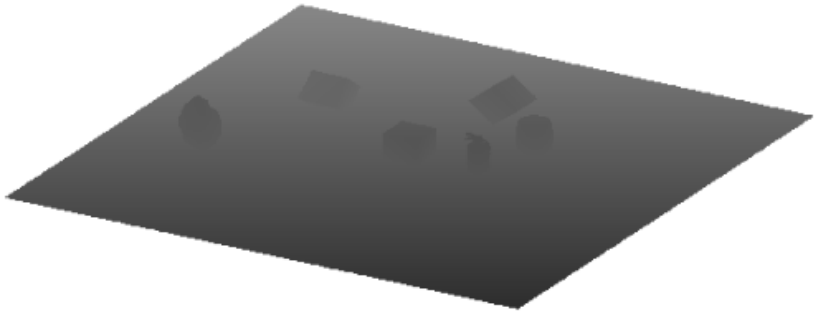


Figure 4: Shadow map. We considered the plane for visualizing this depth map, but you can ignore the plane when you implement the shadow map.

Followings are details for creating and applying a shadow map. First, create a framebuffer to render shadow map. (Actually, this is already in the skeleton code, so you have nothing to do.) Then, you should create a shadow map by placing a virtual camera to the point of the light source and rendering the scene from the point of view of the light. For the virtual camera, you should generate the view matrix and projection matrix appropriately to cover the entire scene (This part is also implemented in the skeleton code). Note that you should use orthographic projection to create a shadow map because we assume that the light source

is a directional light. After creating the shadow map, we can create the shadow effect by comparing the normalized depth of a point which is transformed to the clip space of the light source and the corresponding value of shadow map. Although you implement the algorithm correctly, there can be moiré patterns. To solve this problem, you can increase the resolution of the shadow map or make the shadow condition stricter with bias (implementation details can be found in LearnOpenGL site). Please remove the moiré patterns before you submit the homework.

Prob 4. Advanced Shadow Mapping (20 pt)

Unfortunately, you can easily notice that there is an annoying aliasing on the border of the shadow. The easiest way to improve it is increasing the resolution of the shadow map, but it is not efficient. In this problem, you will implement two advanced techniques related to anti-aliasing and soft shadowing. Implementation details are not specified and we will not provide any skeleton code for this problem. Therefore, to help us understand what you have implemented, you have to also submit a short (0.5 to 1 page) report that describes your implementation. Just a single txt file is OK.

(1) Percentage-Closer Filtering (10 pt)

First method to achieve anti-aliasing is using percentage-closer filtering, or PCF which refers to many filtering functions that produce softer shadows. One simple way to implement PCF is sampling surrounding texels of the depth map and average the results (bilinear filtering). This approach reduces blocky edges of the shadow, but the shadow map's texels are still visible. (Fig.5 left) We can resolve this problem by irregular sampling, or jittering. You can implement this by whatever you want, but we recommend to use Stratified Poisson Sampling, one of the most widely used methods. You can find more details on Stratified Poisson Sampling here.

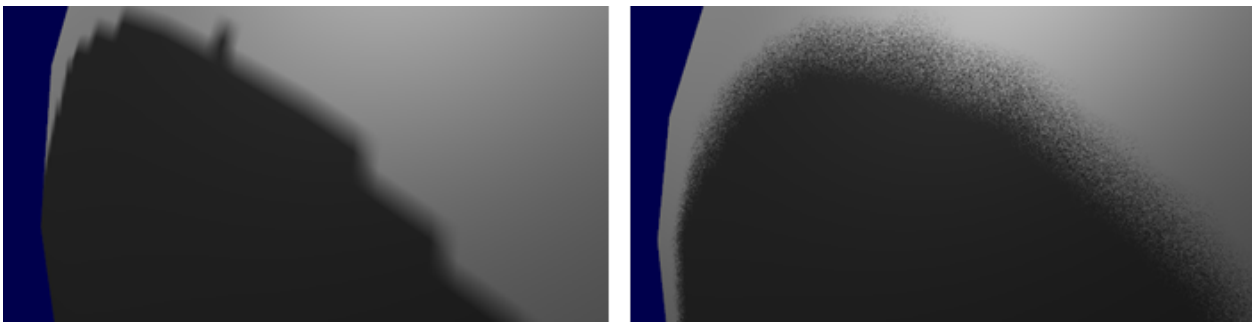
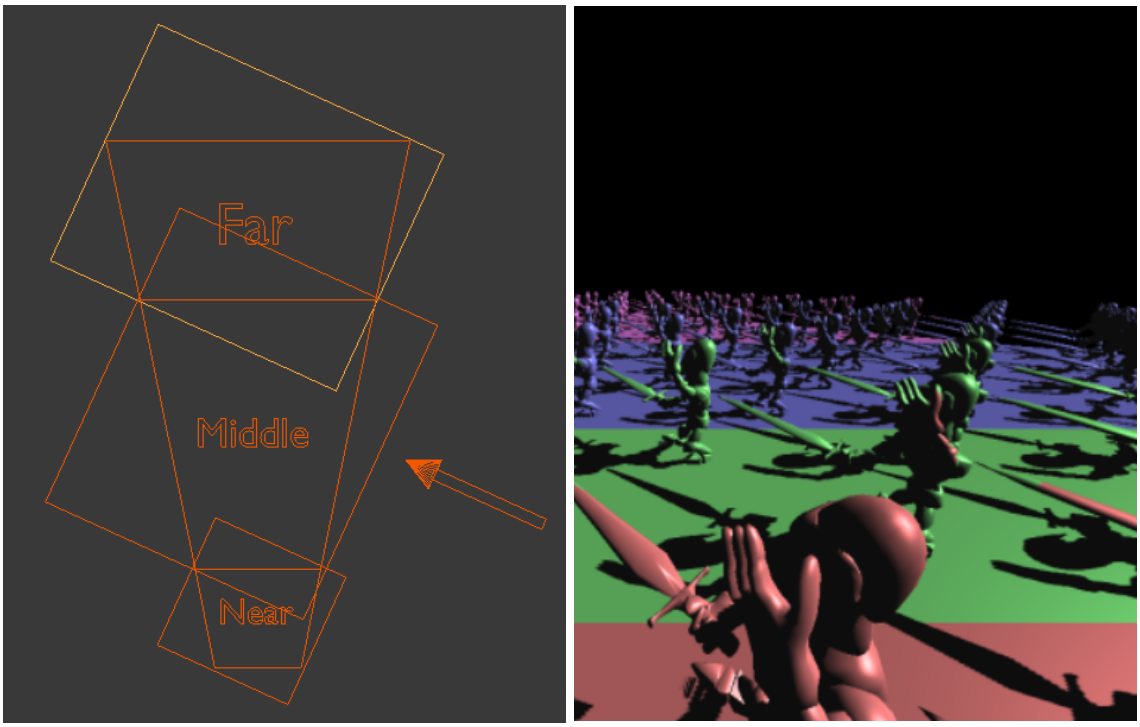


Figure 5: PCF with bilinear filter (left) vs PCF with stratified Poisson sampling (right) (source)

(2) Cascaded Shadow Mapping (10 pt)

Second method to achieve anti-aliasing is trying to fit the frustum as closely to the scene as possible, so that increase an effective shadow map resolution. Cascaded Shadow Mapping (CSM) is one of this technique that uses multiple shadow maps providing higher resolution depth map near the camera and lower resolution far away. CSM is usually used for casting shadows on a large terrain. In this case, using PCF alone is not sufficient since shadow rendering distance becomes large and a result of PCF becomes so blurry. You can find

more details of CSM here and LearnOpenGL. Please implement your own CSM. There are no restrictions, but please use at least 3 shadow maps.

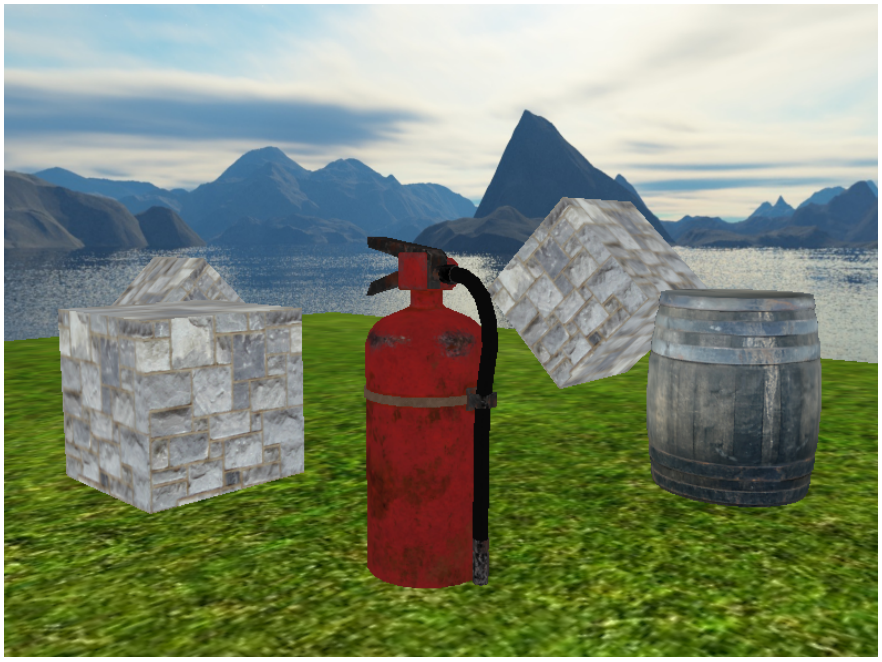


(a) (b)

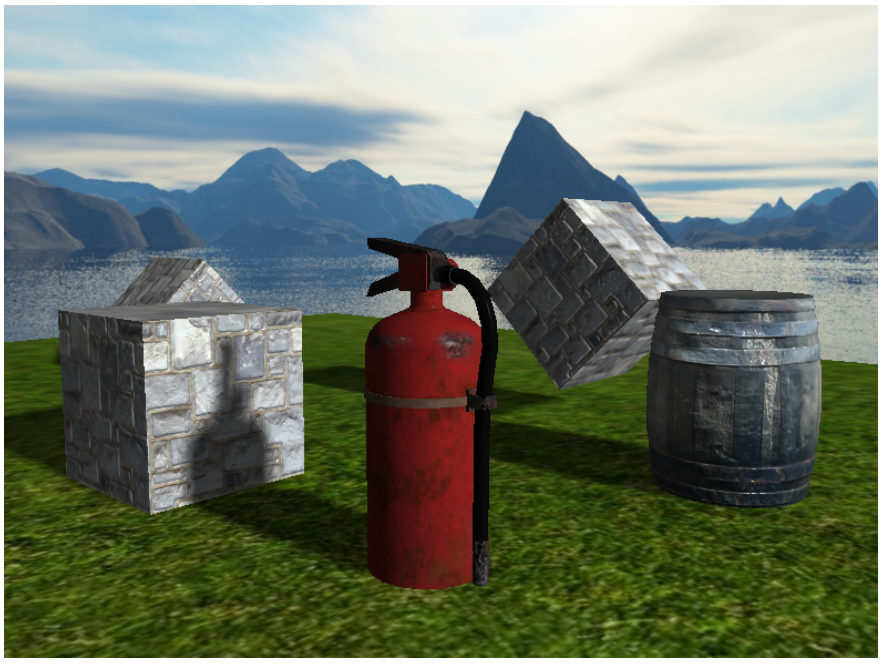
Figure 6: (a) CSM with three shadow maps (source) (b) Displays the shadow map used for each part of the scene. (source)

(IMPORTANT) Here are things to know for all of the problems above.

- Some models do not have specular/normal map. For example, ground plane only has diffuse map. Then we should not bind its normal/specular map and the lighting calculation within the shader should also be different. Instead of creating a new shader for this special case, use a uniform variable to enable/disable whether to use normal map or specular map.
- Make the lighting algorithm on and off by pressing the following keys. Key 1 (normal mapping), Key 2 (shadow mapping), Key 3 (whole lighting). Also, as mentioned in Problem 1, assign the arrow keys for changing the sun azimuth and elevation.
- If you implemented PCF or CSM, show that you implemented the algorithms correctly by enabling its functionality. You can choose which functionality to on/off and corresponding key whatever you want, but please let us know by writing it on the report.



(a)



(b)

Figure 7: (a): Initial state (b): Result after applying the all algorithms except CSM