Computer Graphics 4052

Lab 3

Binh-Son Hua
Trinity College Dublin

In this lab, we will learn how to light up our scene in OpenGL. There are two objectives: implementing lighting and shadows, respectively.

1. Lighting

Let us start with the given template code. At this point you will find the template code very straightforward to understand since it simply prepares a box geometry with some colors, setup the camera, and render the scene.

In this lab, for simplicity we will keep using boxes, but you are recommended to learn how to load scene data from 3D asset files (e.g., OBJ format) if you are interested in making complex scenes.

The box data used in this lab is derived from the Cornell Box, a classic example in computer graphics. Refer to Wikipedia and the original Cornell Box page for a description of the scene. Lab 3 used the data provided in the original Cornell Box page in an OpenGL implementation. By convention, coordinates along the X and Z axis are negated while coordinates along the Y axis are kept as is, so that the camera looks toward negative Z axis.

The default scene in Lab 3 only includes the floor, ceiling, left wall (red), right wall (green), and back wall. We assume that the surfaces are Lambertian, which means that the surfaces reflect light equally to all outgoing directions.

Our goal is to illuminate the box surfaces by adding a point light at the center of the ceiling wall. We represent the light source by a lightPosition and a lightIntensity parameter that are passed to the shaders for lighting computation.

You can control the lightPosition variable by using the S/W key to move the light back and forth (along Z axis), and the mouse cursor to move the light along X/Y axis. If you need to control the light intensity you can add a similar implementation.

You will need to implement the following steps.

Step 1: Prepare normal vectors at each vertex.

Follow the above steps by first defining normal vectors for existing surfaces of the box. Extend the code and shaders to include the normal vector in the shaders.

Step 2: Implement the lighting equation in the shader.

Implement the lighting equation that computes the output color at a screen fragment. Assume that the surface is Lambertian, and the point light radiates equally in all directions.

Step 3: Implement simple post-processing steps including a tone mapping operation and a gamma correction step to map the lighting output to displayable RGB values.

You might find that your rendered image can be too dark or too bright. It is expected that the output RGB is in range [0, 1] (which is subsequently map to [0, 255] for an 8-bit color channel for display). However, sometimes our output range can vary.

To scale the output color range suitable for display, we need to implement tone mapping. A simple tone mapping equation (known as the Reinhard tone mapper) is as follows:

$$c = \frac{v}{1 + v}$$

where v is the input RGB vector and c is the output.

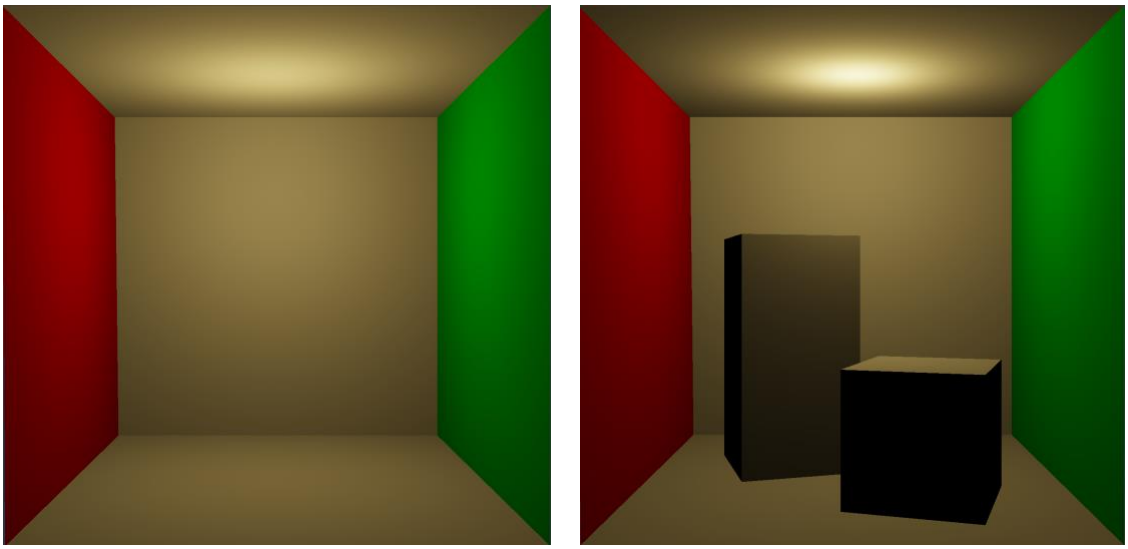
Additionally, note that so far, we have computed the color in the linear RGB space, i.e., the output color is linear to the light source intensity. In practice, it is common that our display uses a non-linear color space, e.g., the sRGB space, for an optimal distribution of color responses to human eyes. To convert our color to sRGB, we can apply an (approximate) gamma correction as follows:

$$c' = c^{\gamma}$$

where c is input and gamma = 2.2 to approximate sRGB.

To adjust the global brightness, you can also use an exposure value to control the final output.

Below is an example of the Cornell Box lit by a point light:

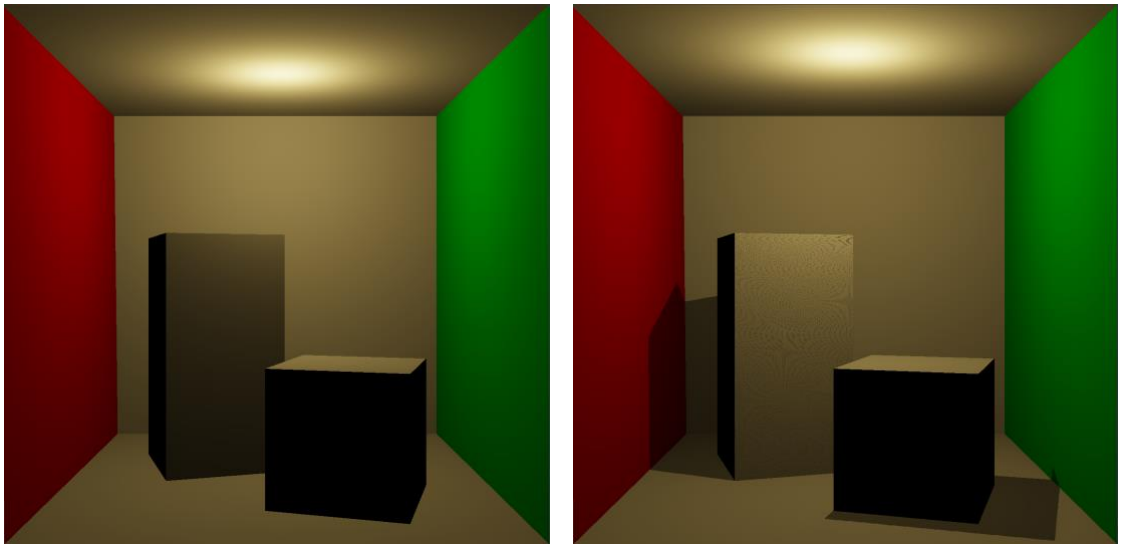


Finally, complete the Cornell Box by adding the short and tall boxes (refer to the original Cornell Box data page). You will need to compute (or pre-compute the normal vectors) of the additional boxes so that they can receive proper lighting. See the attached screenshot above.

2. Shadows

In this section we will improve the photorealism resulting from our lighting implementation. Did you notice that for the current lighting, the scene has no shadows?

Our goal is to implement shadow mapping, a popular algorithm for rendering real-time shadows in OpenGL. An example screenshot of the scene without/with shadows is as follows.



To implement shadow mapping, follow these steps for a **two-pass rendering** algorithm.

Step 0: Learn how to use a framebuffer object (FBO) in OpenGL. This is a type of screen buffer that stores a set of buffers including the color buffer, the depth buffer, the stencil buffer, etc. You can refer to examples generated by ChatGPT, or from online resources. Focus on use cases that involves attaching a depth texture to the FBO.

We will use FBO to implement render-to-texture so that we can store the scene's depth buffer into a texture for later use with shadow mapping.

Prepare a frame buffer object (FBO). Prepare a new texture and and attach it to depth buffer of the FBO.

Step 1: Activate the FBO frame buffer and render the scene as viewed from the light source. For simplicity, we only take care of the shadow of the light when the light is shining downwards. Therefore, only a single texture is required. (You do not need to implement cubemaps for shadows of a point light.)

Set the view frustum such that the view looks downwards from the light location on the ceiling. Take note of the view and projection matrix in this step. Let's call it the light-space view and projection matrix.

As the FBO is activated, it will record the depth buffer (also known as the z-buffer) into our previously created texture.

Step 2: Render the scene from our camera view as usual. Bind the depth texture to the fragment shader. Also bind the light-space view and project matrix. In the fragment shader, shadow can be determined as follows. First, we project the current fragment position (in world space) to the

light-space view, and retrieve its projected pixel coordinates (denoted as uv) and the depth value in the light space (denoted as depth). Use uv to query the depth texture to obtain the existing depth value (existingDepth). Now shadow can be determined by comparing d with existingDepth using the following formula:

```
shadow = (depth >= existingDepth + 1e-3) ? 0.2 : 1.0;
```

where 1e-3 is a small depth bias value to support the comparison. When the current fragment position is in shadow, its depth in the light space falls behind the existing depth value, and hence the comparison returns true. In this case, the final color can be multiplied with a scalar factor (e.g., 0.2) to simulate the darkened area. Otherwise the fragment is not in shadow and therefore the final color is kept as is (multiply by 1.0).

The implementation of shadow mapping might need some efforts for debugging. The following supporting features are provided:

- Save depth texture. Use this function to save the depth buffer into a PNG image. Make sure that you set the view and projection in Step 1 properly and there is some content in the depth texture. This can be observed in the PNG image (make sure to darken/brighten the image during inspection to see some saturated image features if any).
- Note that the depth buffer values in texture is in [0, 1]. Be aware of the coordinate space when the fragment is projected into the light-space view.
- Be aware of depth precision. When depth is in NDC space [-1, 1], it is mapped to the entire range between the near and far plane. If depths are represented by only a small range in [near, far], depth resolution is very low, and the shadow can be incorrect. Therefore, a place to check is to adjust near planes, far planes to make sure the depth buffers are correct.

Submission:
Package the completed lab3_cornellbox.cpp and an mp4 video that captures the rendering of your Cornell Box in different viewpoints or lighting conditions, into lab3_results.zip.

Your submission is counted as complete when it has the following features:

1. Complete Cornell Box data (with the short and tall boxes inside).
2. Lighting implemented
3. Shadow implemented
4. The implementation must follow the structure in the existing template code.

Implementations with significant differences in coding styles and structures (e.g., from GPT-generated code or some online shadow mapping code) will be marked as incomplete.

On Blackboard, go to Submissions -> Lab 3 and upload your zip file.
Please only pack the .cpp and .mp4. Do not upload the entire source code unless you have other dependencies.

Deadline: **Wednesday, 13 November 2024, at 12pm (noon).**

Marking: You will get a complete/incomplete score based on your results.