Assignment 1 : Creating your own OpenGL program

NAME: ZHANG ZHANRUI
STUDENT NUMBER: 2019533227
EMAIL: ZHANGZHR2@SHANGHAITECH.EDU.CN

1 INTRODUCTION

In this Assignment, the following tasks are finished. First, the mesh object in resources folder are loaded. Then they are properly drawn on the screen. After that, multiple lighting was added to the object, based on the Phong shading model. At last, keyboard and mouse input are handled so that we can move around in the scene and watch those objects all directions.

2 IMPLEMENTATION DETAILS

2.1 Loading Meshes From File

First we need to load the mesh file into our program. To make it clear, this part is implement in the '.scr/loadmesh.h' file. Given the file name, then the object is load into a 'Mesh' class, which contains 4 members.

- 'vertexAttrib' is a float vector. It contains all the vertex attribute. In our cases, it is three number representing vertex position and then three number representing normal direction.
- 'indices' is a integer vector. It contains the indices of vertices of each triangle. The index starts from 0.
- 'v_num' and 'f_num' are two integers that record the number of vertices and faces of the obj.

In 'main.cpp', I created two 'Mesh' objects called Bunny and Pumpkin, now we can send these data into the rendering pipeline.

2.2 Draw Meshes on the screen

Once loading the meshes and store them in vectors, we can draw them on the screen.

2.2.1 Bound Vertex Attribute. We need to create a VBO object and bind the vector containing vertex attribute onto it. We need to create EBO and do similar things for index vector. At last we bind VBO and EBO on VAO. Two VAO are created to handle two objects.

2.2.2 Shader Program. After binding those vertex attribute, two more shader programs are needed in the rendering pipeline – vertex shader and fragment shader. We are provided with a shader class. We only need to provide the path to our shader program, and it will create, compile the shader for us.

2.2.3 Vertex Shader. The vertex shader receives data from VBO object. First, 'layout' is used to tell the shader where each vertex attribute is. In out cases, we have two vertex attributes – vertex position and normal.

For vertex position, The input position from VBO is local space, while the output should be the coordinate in clip space, which is a cube. so we need to apply three matrix.

The model matrix transform the coordinate into the world space. This matrix is manually implemented. For bunny, I keep it in origin with model matrix being identical matrix. For pumpkin, I moved it a little bit.

The view matrix transform the coordinate into view space. For bunny and pumpkin, this transformation is the same. The view matrix is kept in 'camera' class in 'main.cpp'.

The projection matrix project the object in the frustum into a cube. This matrix is also the same for bunny and pumpkin. This matrix can be created by using the 'perspective()' function provided by glm.

All three matrix above can be created in 'main.cpp' and pass to the shader by using 'uniform'. A variable with keyword 'uniform' is required in the shader program. In 'main.cpp', we just need to call 'setMat4( )' in 'shader.h' provided by TA.

At last, these matrices are applied to vertex position and the result is assigned to 'gl_Position'.

2.3 Lighting

So far, the object we have drawn just have shape but do not have shadow, so we need to add lighting. Fragment shader controls the behavior of each fragment. The output from the vertex shader will be interpolated automatically.

2.3.1 Multiple Light Sources. As is required, I placed 2 light sources in my scene. To define a point light source, we need its position in the world space and its color. I set coordinate of light in world space in main function and pass it into the shader by setting two uniform variable in vertex shader. The lighting calculation will be performed in view space, so view transformation is needed. We just apply view matrix on light position and pass the results into the fragment shader.

2.3.2 Ambient Light. Ambient light comes from the reflection of environment. Here I assume that the environment is white, which means the reflection color is just the same as the color of light source. Then I just need to define a 'ambt_factor' to represent the strength of ambient light.

2.3.3 Diffuse Light. Diffuse light depends on the relationship between light source and object.

We already have the light source position in view space in fragment shader.

To know the position and normal of object in view space, we just need to apply corresponding transformation matrix on their position to get the coordinate in view space and output them to fragment shader.

© 2021

Now, we have everything in the view space, just calculate dot product of light direction and normal vector to figure out how strong the diffuse color will be.

2.3.4 Specular Light. When calculating diffuse light, we already have the light direction and normal in view space. We can use reflect function to calculate the reflection and compare it to the 'view vector' which is easy to calculate since the camera is on origin and reflection point is known.

After knowing how strong ambient, diffuse and specular light is, we just add them together, then times the color of light source, we can get the color of the fragment. Since there are two light source in the scene, we need to add two color together to get the final output color.

2.4 Camera control

I defined an FPS style camera in this assignment. The goal is to control the camera in the following manner:

- move the camera up and down along the up direction of camera by pressing 'W' and 'S'.
- move the camera left or right along the right direction of camera by pressing 'A' and 'D'.
- move the camera forward and backward along the gaze direction of camera by pressing 'R' and 'F'.
- pitch and yaw the camera by mouse.

A class 'camera' is defined in the file 'camera.h'. It contains several members:

- a 4×4 matrix: this is the view matrix, defining how we transform the world space to the view space.
- Three vectors: 'camPos', 'camGaze', 'camUp', defining the camera's position, directions it looking at, and up direction in the world space.
- Two doubles: 'yaw' and 'pitch' to record the pitch and yaw angle of the camera.

2.4.1 Control with keyboard. The code skeleton has already defined a 'processInput' function to handle the ESC input for exiting the program, so similarly I add 6 more conditional branches to handle the input from 'A', 'S', 'D', 'W', 'R', 'F'. Once a key is pressed, we can call corresponding move function to change corresponding vector and them update the view matrix.

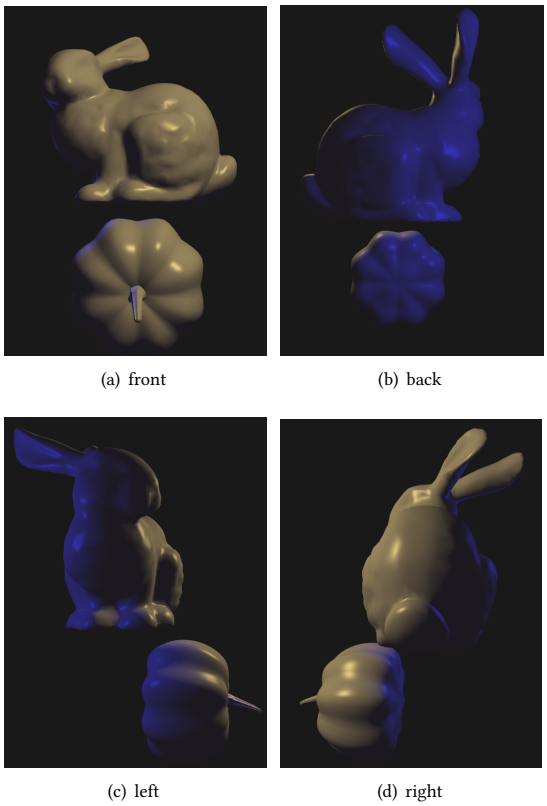
2.4.2 Control with mouse. To control with mouse, a call back function is needed. And we just calculate the distance the mouse moved on X and Y direction and corresponding shift on variable 'yaw' and 'pitch', and then update the view matrix.

2.5 Enable MSAA

GLFW has provided us with corresponding multi-sample anti-aliasing functionality, we need to call 'glfWindowHint( )' and tell glfw the number of sub-sampling point we need. Then we need to call 'glEnable(GL_MULTISAMPLE)' to enable MSAA.

3 RESULTS

The results can be seen in the picture. The bunny is placed on the origin, and the pumpkin is moved and scaled down a little bit to avoid intersection. Both object's base color are set to 60% grey. Two point lights are placed. One brighter, yellow light in the front, and one darker, blue light in the back.

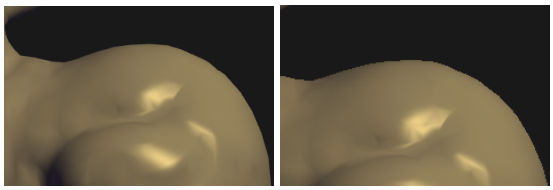


(a) front (b) back

(c) left (d) right

Fig. 1. different angle

Comparing the result before and after enabling MSAA, it is clear that the edge of the object become smooth after enabling MSAA.



(a) MSAA enabled (b) MSAA disabled

Fig. 2. Anti-aliasing