Assignment 1:
Exploring OpenGL and Phong Lighting

NAME: HIDDEN
STUDENT NUMBER: HIDDEN
EMAIL: HIDDEN@SHANGHAITECH.EDU.CN

1 INTRODUCTION

In this assignment, the following tasks are finished.

- First, the mesh object in the `assets` folder are loaded into main memory. They are then sent to and processed by the GPU.
- Second, keyboard and mouse (cursor moving and scrolling) input are handled to allow us moving around in the scene and watch those objects from any direction we like.
- After that, the phong lighting model is implemented to make the object look more realistic.
- Finally, multiple lighting of different kinds are implemented (bonus1), and normal vectors are shown with the help of geometry shader (bonus2).

2 IMPLEMENTATION DETAILS

2.1 Loading Meshes From File

First we need to read `obj` files into our program. We will implement this inside the `Mesh` class, located in `src/mesh.cpp` and header file `include/mesh.h`. The `Mesh` class contains mainly the following members.

- `std::vector<Vertex> vertices`, which holds the vertices' position.
- `std::vector<GLuint> indices`, which holds the indices of vertex position and normal of faces.
- `uint VAO, VBO, EBO`. These are VAO (Vertex Array Object), VBO (Vertex Buffer Object) and EBO (Element Buffer Object) ids, or references number.

The constructor reads in a mesh object by calling member function `void Mesh::loadDataFromFile(const std::string &path)` to complete that. It reads in a file line by line, checking the first character (which should be `'#'`, `'v'`, `'n'` or `'f'`), and then reads in the float or int values and store in vertices and normals.

2.2 Draw meshes on the screen

So far, we have read the file to our CPU memory. Then we need to copy these data to GPU memory for further handling. We achieve that with the help of VAO, VBO and EBO. These are implemented in member function `void init_objects()`. In detail, the following things are done.

- Bound Vertex Attribute. We need to create a VBO object and bind the vector containing vertex attribute to it. We need to create EBO (index buffer object) and do the similar things for the indices. At last we will bind VBO and EBO to VAO.
- Write the shader program. In vertex shader, we need to transform the vertices from model space to clip space. In the fragment shader, we need to calculate the color for each pixel. Note that, for simplicity, we will keep the color to be pure white and implement the lighting model in the next step.
- Create and compile shader programs. After the shaders are implemented, we need to compile and link them to a shader program. This is implemented in `void initWithCode(...)` member function in `Shader` class. Error handling is also implemented which will output an error message if the compilation fails.

2.3 Handling mouse and keyboard input events

Now the objects are correctly rendered, but we cannot move them. We listen for input events by defining a callback function and set them using openGL. However, before that, we need to understand perspective projection first.

The input position of vertices stored in VBO is in world space and we need to transform them into clip space. The transformation is done by multiplying the vertex position with 3 matrices: the model matrix, projection matrix and view matrix.

- The model matrix can change the object coordinate in the world space. In this project, we won't change the positions, so this matrix will be omitted (or filled with all 1s).
- The projection matrix is calculated by `glm::perspective (fov, aspect, near, far)` where `fov` is the field of view, `aspect` is the aspect ratio of the screem, `near` is the nearest plane and `far` is the furthest plane.
- The view matrix is calculated by `glm::lookAt(camera_pos), camera_pos + camera_gaze, camera_up`, where `camera_pos` is the position of the camera, `camera_gaze` is the direction the camera is facing, and `camera_up` is the up direction of the camera.

All the three matrices above are calculated in `main.cpp` and then passed into shader programs using uniform variables. The vertex shader multiplies the matrices and transform the vertex position into clip space.

Now back to input handling. There are three callback functions for keyboard, mouse cursor movements and mouse scrollin. To make the code structure easier for management, we implemented a `Camera` class with camera information and camera movement member functions inside it.

There are three callback functions for keyboard, mouse cursor move, and mouse scrolling.

- Keyboard Input.
  When any of W,A,S,D,R,F is pressed, we move the camera position. This is done in `void process_input(GLFWwindow *win)`.
  - we can move the camera up and down by pressing W or S
  - we can move the camera left and right by pressing A or D
  - we can move the camera forward and backward along the gaze direction by pressing R and F
- Mouse cursor input.
  When mouse is moved, we update `pitch`, `yaw`. This is done in `void my_mouse_callback(GLFWwindow *win, double x, double y)`. Note that since we need to get the mouse movement instead of mouse position at each time, we have to set a `bool firstMouse` flag, which is initialized to be true. In the first mouse callback, we store the current mouse position. Otherwise, we check for mouse movement scale and update the current position.
  Then we bind this function by `glfwSetCursorPosCallback` before entering the main loop.
- Mouse scrolling movement.
  We will update `fov` (field of view) when mouse scrolling is detected. We bind this by `glfwSetScrollCallback` before entering the main loop.
- Disable cursor.
  Finally we disable the cursor display (so that we will not see the cursor while moving our mouse) by `glfwSetInputMode`.

2.4 Phong shading model implementation

So far, we have successfully drawn the object shape, but it is in pure orange color. We need to add lighting to it. Fragment shader controls the behavior of each fragment, so we will mainly implement the phong lighting in `fShader.glsl` file.

According to phong lighting model, the color of a fragment is the sum of ambient light, diffuse light and specular light.

- Ambient light. This comes from the reflection of the environment. It is independent of the surface normal. We will use a constant factor of `0.2`.
- Diffuse light. The diffuse light depends on the relations between the light source and the object. We calculate the dot product of light direction and normal vector (of a fragment) to get the strength of diffuse color.
- Specular light. We use the `reflect` function to calculate the reflection of light direction and normal vector, and then adding the view matrix.

2.5 Geometry shader

The geometry shader is an optional shader stage that sits between the vertex and fragment shaders. It can convert the original primitives to completely different primitices. In this assignment, we will show the normal vectors of the object using this shader.

We will implemented another combination of shaders (with geometry shader) and compile these two shaders. Then inside the main loop, we will use the original shader first, and then apply this geometry shader.

3 RESULTS

3.1 Basic Phong lighting model and multiple light sources

Here are the front and back side of the bunny. There's a bright and yellow light source in the front, and a darker pink light source in the back.

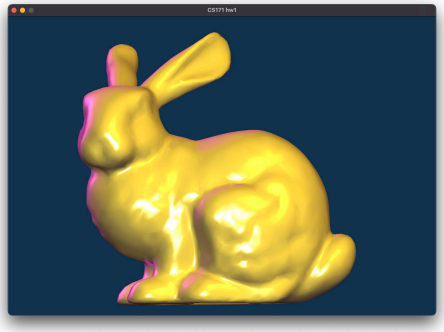

Fig. 1. Front side of the bunny

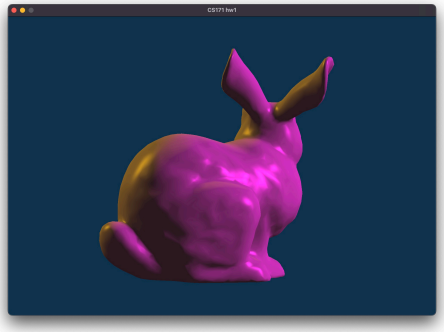

Fig. 2. Back side of the bunny

3.2 Spotlight

There's also a spotlight shining from the camera. When we move the bunny slowly away from the spotlight, we can see its surface darkening.

(a) under spotlight

(b) halfway outside spotlight

(c) outside spotlight

Fig. 3. Spotlight

3.3 Geometry shader and Normal vector

By implementing a geometry shader, we can show the bunny's normal vectors.

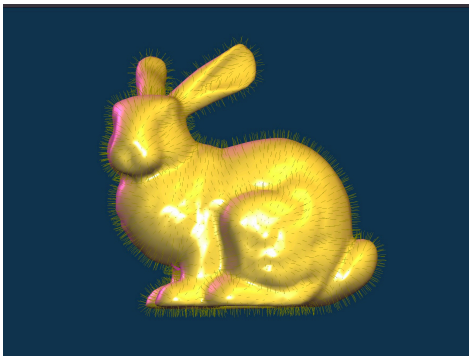

Fig. 4. Bunny with its normal vectors as fur

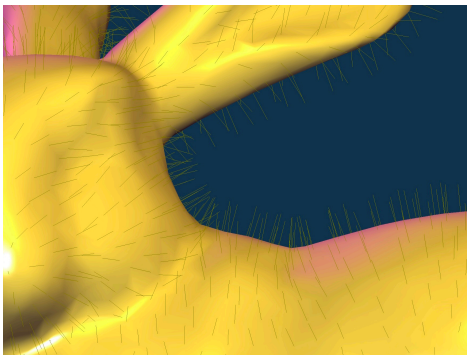

Fig. 5. Details of the above