COMP3069 Computer Graphics
Coursework 3 - Final Report

Name	Zhangli WANG
Student ID Number	20028336
Email	scyzw1@nottingham.edu.cn

INTRODUCTION

This project should be launched in the x64 configuration in Visual Studio IDE.

Scene Introduction

In the final version of this project, the scene of an Aztec temple in a raining jungle has been built. The light mode can be set either as daylight mode or night mode.

Figure 1 Overall Scene

The above figure demonstrates the overall scene in the daylight mode. An Aztec temple is built in the center with trees and rocks surrounded. Eagles are hovering over the temple. The scene is covered by falling raindrops.

Figure 2 Eagle Perspective

By keyboard input, several viewpoints can be fixed, for example, the camera can be fixed at the hovering eagle or fixed looking at the temple. And the effects can be switched, for example, flashlight on/off, raining on/off.

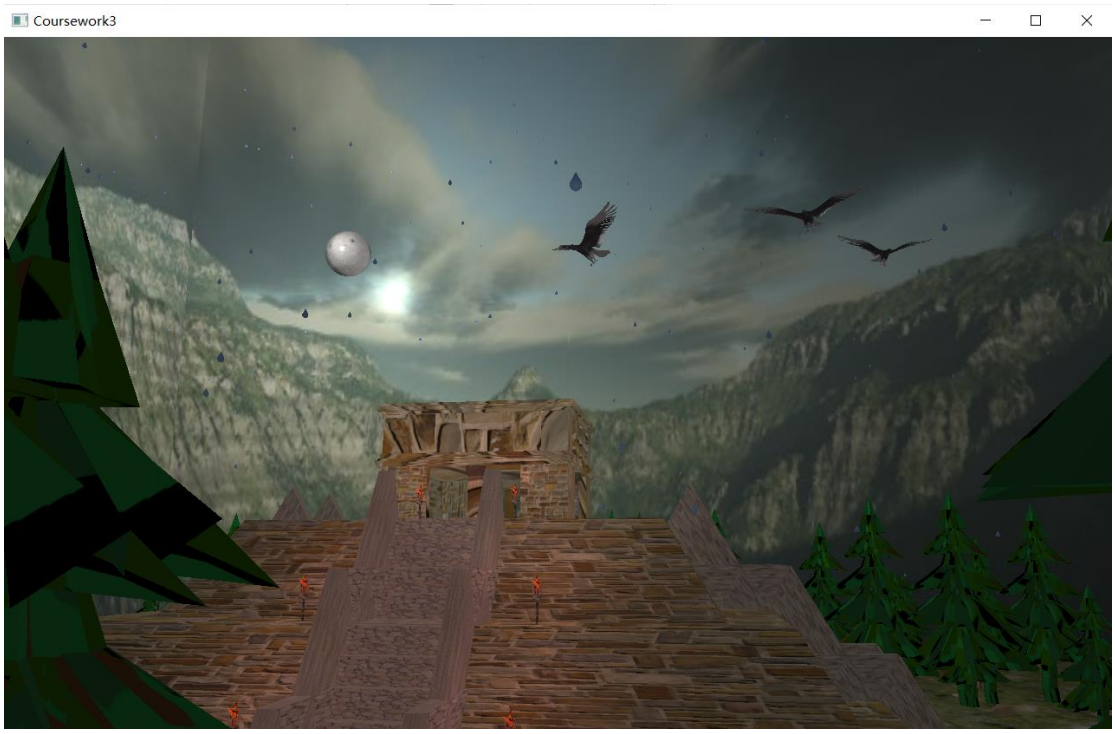


Figure 3 Night Mode

In the night mode, the moon is rotating around the horizon and giving out moonlight. Torches on the temple are twinkling with firelights. The camera gives out flashlight.

The screenshots are static, so to view the complete visual effect, please check out the video clip.

Program Instructions

After running the program, instructions can be called. Full instructions are in the table.

Operation	Effect
Mouse	
Sliding	The direction of the camera is changed accordingly.
Scrolling up	The camera is zoomed in.
Scrolling down	The camera is zoomed out.
Keyboard	
Pressing ‘W’	The camera is moving forward.
Pressing ‘A	The camera is moving leftward.
Pressing ‘S’	The camera is moving backward.
Pressing ‘D’	The camera is moving rightward.
Pressing ‘L’	The illumination mode is set to “night mode”.
Pressing ‘F’	Turn on the flashlight in the night mode.
Pressing ‘G’	Turn off the flashlight in the night mode.
Pressing ‘R’	Disable raining effect.
Pressing ‘T’	Enable raining effect.
Pressing ‘O’	The illumination mode is set to “daylight mode”.
Pressing ‘P’	The camera is locked at the downside of the eagle.
Pressing ‘1’	The camera is locked at the front of the temple.
Pressing ‘2’	The camera is locked at the back of the temple.
Pressing ‘3’	The camera is locked at the left of the temple.
Pressing ‘4’	The camera is locked at the right of the temple.
Pressing ‘ESC’	The program is terminated.

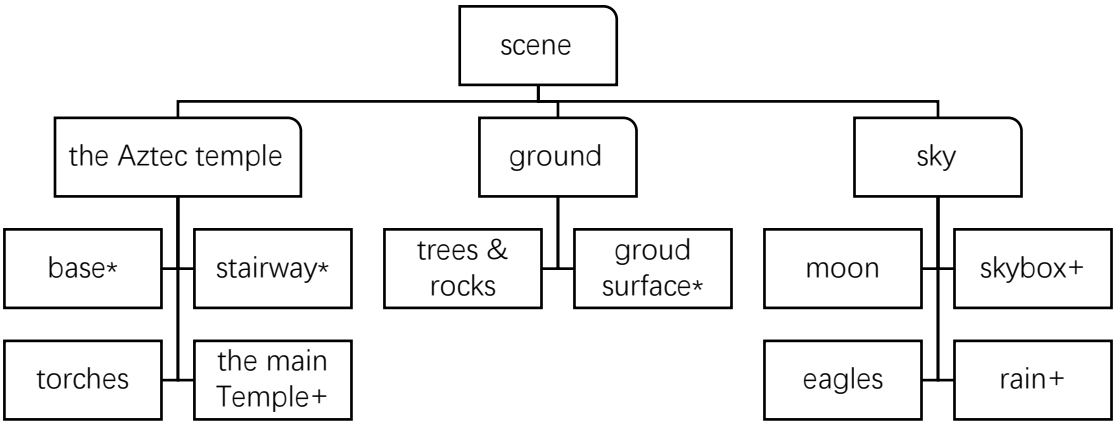
Requirements Meeting & Creative Ideas

The source codes requirements have been completed. In this program, several imported models, blender-built models, and self-built models are drawn with textures bound. Animations of the moon, raindrops, and eagles are demonstrated. Model transformations are mainly reflected by the stairway building, trees & rocks building, and the animations. Perspective changing and several viewpoints are also implemented. Lighting is adopted by the daylight, twinkling torchlights, dynamic moonlight, and the flashlight. The program is also refactored, encapsulated classes are created for different usages.

In my view, there are some creative ideas. The first one is the dynamic moonlight and the twinkling torchlights in the night mode. The changing lighting effects can make the scene looks different at different time. The second one is the raining effect. It makes

the scene looks more vivid. The third one is the functionality of locking the camera at the eagle. It creates a unique viewpoint to the scene.

IMPLEMENTATION



From the structure point of view, above tree diagram demonstrates the general classes and types of objects, where the models marked with an asterisk sign are manually built with vertices, the models marked with a plus sign are built with the modeling software, blender, whereas other models are imported.

In this program, several types of files or classes are defined for different usages.
Classes:
`manual_model` is used for loading and drawing the manually built models.
`imported_model` is used for loading and drawing the imported models.
`lighting` is used for setting the attributes of the lights for the shader.
Data File:
`temple_data.h` is used for storing the vertex data of the manually drawn models, with coordinates, texture coordinates, and normals.
Shader files:
`*.vert` and `*.frag` store the vertex shader and fragment shader codes.

In the following sections, implementation of concrete models or functionalities will be introduced.

Self-built Models

Bases, ground surface, and stairways are manually drawn models. Before the render loop, the vertex data of some basic geometries are loaded to the VAOs from the data

file `temple_data.h`, and the texture data is loaded to the texture buffer from image files, as introduced in the interim report.

```
// transform: left enclosure for the first base
glm::mat4 enclosure1Model;
enclosure1Model = glm::translate(benchmark1, glm::vec3(2.0f, 0.0f, 17.0f));
enclosure1Model = glm::rotate(enclosure1Model, glm::radians(240.0f), glm::vec3(1.0f, 0.0f, 0.0f));
enclosure1Model = glm::translate(enclosure1Model, glm::vec3(0.0f, -root3 / 2, 0.0f));
enclosure1Model = glm::scale(enclosure1Model, glm::vec3(0.1f, 1.0f, 1.0f));
lightShader.setMat4("model", enclosure1Model);
// bind textures
glActiveTexture(GL_TEXTURE14);
glBindTexture(GL_TEXTURE_2D, enclosure_texBuffer);
glUniform1i(glGetUniformLocation(lightShader.ID, "theTexture"), 14);
// draw the enclosure
// transform: left enclosure for the first base
glBindVertexArray(enclosure_vao);
glDrawArrays(GL_TRIANGLES, 0, 36);
```

In the render loop, the model is drawn, for example, the model of the left enclosure for the first base is drawn as follows:

1. Transform the model.
   i. Translate the model to the edge of the base.
   ii. Rotate the model to align its local horizontal line to the edge.
   iii. Translate the model to the proper position to the edge.
   iv. Scale the model as the shape of an enclosure.
2. Bind corresponding texture to the draw.
3. Bind corresponding VAO to the draw.
4. Draw the model.

To draw the models for the stairway requires accurate calculation to draw the stairway attached to the bases.

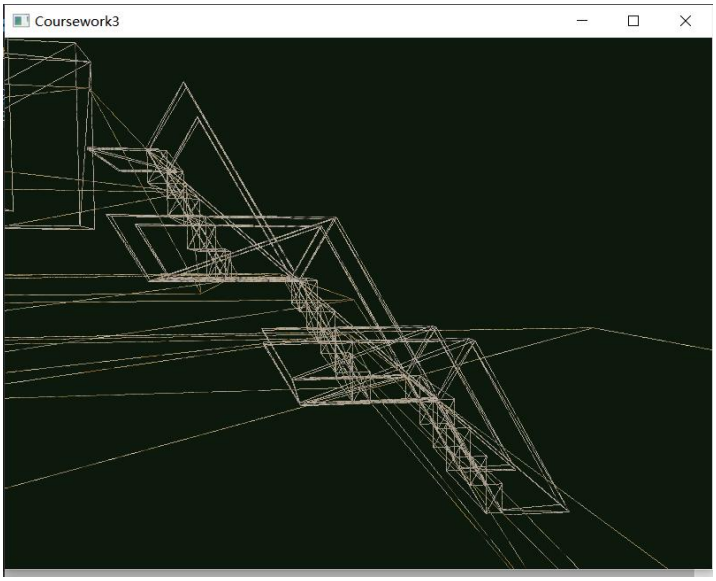


Figure 4 Self-built Stairway Structure

The above figure demonstrates the overall structure of my self-built stairway.

Imported Models

For imported models, before the render loop, vertex data and texture data are loaded from .obj files to the meshes with the assimp importer. In the render loop, meshes of the models are drawn. The models are drawn as follows.

Torches

```
for (int i = -1; i <= 1; i += 2)
{
    glm::mat4 torchModelMatrix = glm::translate(manual_models.get_base1ModelMatrix(), glm::vec3(4.0f * i, root3, 15.0f));
    modelShader.setMat4("model", torchModelMatrix);
    // draw the torch
    torchModel.Draw(modelShader);
}
```

Torches are imported models; they are built according to the positions of bases.

Trees & Rocks

```
    // model: rocks
    // transform: rocks
    for (int i = 0; i < NUM_ROCKS; i++)
    {
        GLfloat rand_rock_pos = rand() / GLfloat(RAND_MAX);
        GLfloat rand_rock_size = 1.0f - rand() / GLfloat(RAND_MAX) / 2;
        GLfloat rand_rock_posture = rand() / GLfloat(RAND_MAX);
        glm::mat4 rockModelMatrix = glm::rotate(model, glm::radians(360.0f * rand_rock_pos), glm::vec3(0.0f, 1.0f, 0.0f));
        rockModelMatrix = glm::translate(rockModelMatrix, glm::vec3(0.0f, 0.0f, 30.0f + 1.0f * random(10)));
        rockModelMatrix = glm::scale(rockModelMatrix, glm::vec3(1.0f * rand_rock_size, 1.0f * rand_rock_size, 1.0f * rand_rock_size));
        rockModelMatrix = glm::rotate(rockModelMatrix, glm::radians(360.0f * rand_rock_posture), glm::vec3(0.0f, 1.0f, 0.0f));
        modelShader.setMat4("model", rockModelMatrix);
        // draw the rock
        rockModel.Draw(modelShader);
    }
```

Trees and rocks are imported models. They are drawn distributed around the temple with random sizes and random positions.

1. Transform the model.
    i. Rotate and translate the model randomly to place it at a random position in a circle.
    ii. Scale the model randomly to randomize the sizes of the models.
    iii. Rotate the model randomly again to form different postures of the models.
2. Draw the model.

Eagles & The Moon (Animation)

```
    // model: eagles
    // define rotation parameters
    GLfloat rotationRate = currentFrame * 36;
    glm::mat4 eagleModelMatrix;
    // transform: eagle
    for (int i = 0; i < 3; i++)
    {
        eagleModelMatrix = glm::translate(model, glm::vec3(0.0f, 20.0f, 0.0f));
        eagleModelMatrix = glm::rotate(eagleModelMatrix, glm::radians(-rotationRate + i * 36.0f), glm::vec3(0.0f, 1.0f, 0.0f));
        eagleModelMatrix = glm::translate(eagleModelMatrix, glm::vec3(0.0f, 0.0f + i * 2.0f, 10.0f + i * 5.0f));
        modelShader.setMat4("model", eagleModelMatrix);
        // draw the eagle
        eagleModel.Draw(modelShader);
    }
    this->eagleMatrix = eagleModelMatrix;
```

Eagle models are imported with the animation of hovering over the temple periodically.

1. Specify the period of rotation.
2. Transform the model.
   i. Translate the model to elevate it to the sky. Different eagles are specified with different heights.
   ii. Rotate and translate the model according to the frame time to set the animation of hovering. Different eagles are specified with different radiuses.
3. Draw the model.

The animation of moon model is drawn using the same technique.

Blender Models

The Main Temple

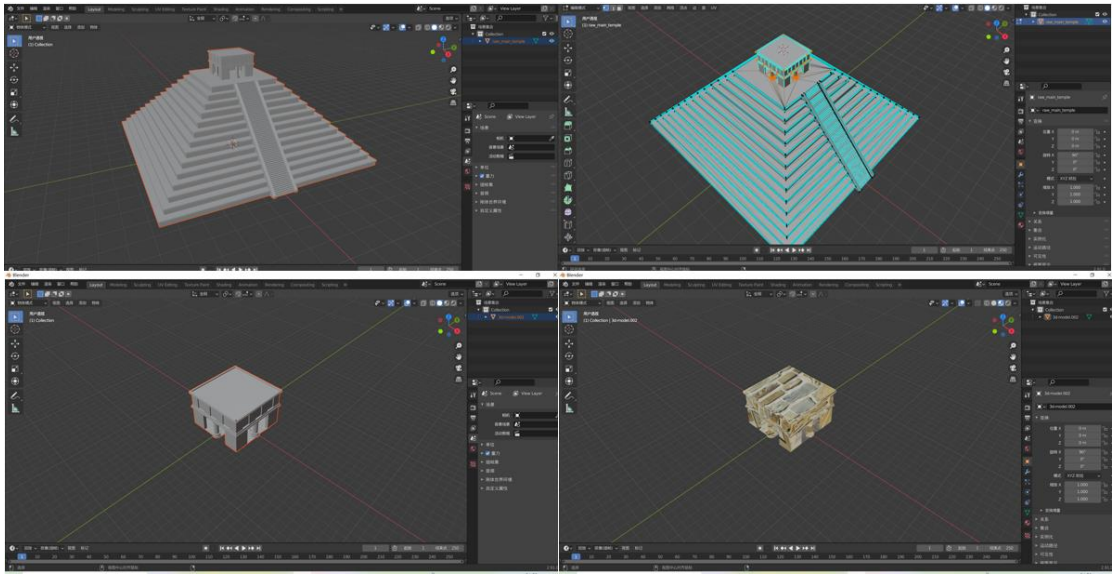


Figure 5 Main Temple Extraction & Texture Binding

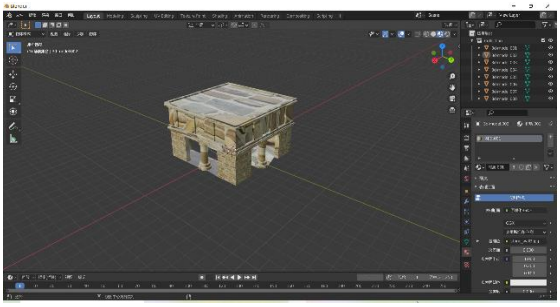


Figure 6 Main Temple Texture Re-mapping

The raw main temple is initially downloaded from the online sources, it should be modified to remove the base and bind with texture to be loaded in the program. The procedure is as follows:

1. Select all the vertices of the main temple.
2. Reversely select all the vertices of the base.

3. Delete the vertices of the base.
4. Scale the model to fit the size of the manually drawn bases.
5. Bind texture to the main temple.
6. Adjust texture mapping to fix texture binding errors.

After the model is built, it is loaded to the program and drawn at the top of the third base.

Rain (Animation)

The raining effect is basically achieved by two steps: creating raindrop model, specify raining animation.

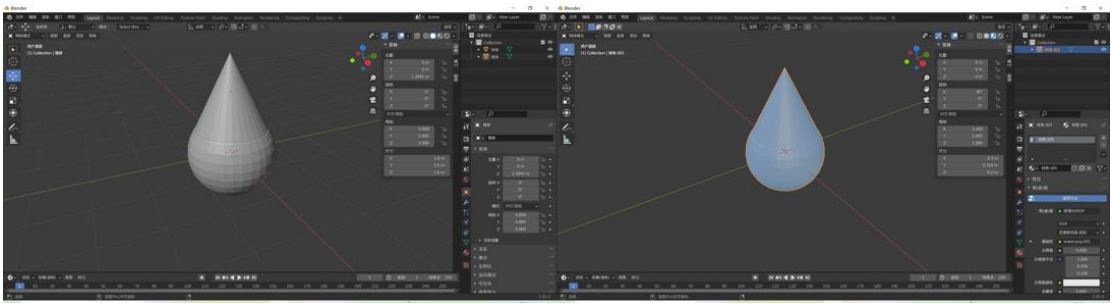


Figure 7 Raindrop Model

In blender, the model of a raindrop is built as follows:

1. Create a sphere model and a cone model.
2. Assemble the two models to an integrated one.
3. Scale down the size of the model.
4. Bind water texture to the model.

```
// model: raindrops
srand(1);
glm::mat4 raindropModelMatrix = glm::translate(model, glm::vec3(-50.0f, 80.0f, -50.0f)); // specify initial position
float drop_distance = currentFrame * 50; // specify speed
for (int i = 0; i < NUM_RAINDROPS; i++)
{
    GLfloat rand_pos_x = rand() / GLfloat(RAND_MAX) * 100.0f; // specify random position in x
    GLfloat rand_pos_y = rand() / GLfloat(RAND_MAX) * 80.0f;  // specify random position in y
    GLfloat rand_pos_z = rand() / GLfloat(RAND_MAX) * 100.0f; // specify random position in z
    glm::mat4 raindropModelMatrix_temp = glm::translate(raindropModelMatrix,
                                                        glm::vec3(rand_pos_x,
                                                                  rand_pos_y - ((int)drop_distance % (80 + (int)rand_pos_y)),
                                                                  rand_pos_z));
    modelShader.setMat4("model", raindropModelMatrix_temp);
    // draw the raindrop
    raindropModel.Draw(modelShader);
}
```

After the model is built, the animation of raindrops should be specified. The implementation is as follows:

1. Initialize the position of the model at a top vertex of the skybox.
2. Randomly reposition the model in a 3D cubic space above the skybox.
3. Move the model downward the y-axis according to the speed, respawn the model at its original height when it approaches the ground.

The respawning functionality is achieved by utilized the mod function to the total drop distance and the specified original height.

Skybox

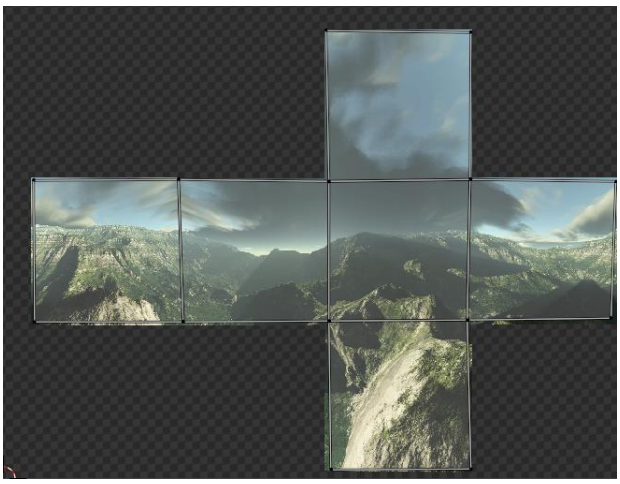


Figure 8 Skybox Texture Binding

The implementation of the skybox is similar. To build the model, the sky texture is bound to a cube in the blender. Then the model is loaded to the program and drawn.

Lighting

```
// specify attributes of point lights
void lighting::specify_point_lights(Shader shader, float currentFrame, bool on_off)
{
    // mutative brightness of point light 1-8
    srand((unsigned) time(0));
    GLfloat rand_offset = rand() / GLfloat(RAND_MAX) * 3.1415926f;
    GLfloat brightness_torchlight = sin((currentFrame + rand_offset) * 50) * 0.05f + 1.0f;

    // dynamic position of point light 9
    glm::mat4 moonModelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, -50.0f));
    moonModelMatrix = glm::rotate(moonModelMatrix, glm::radians(-currentFrame * 36), glm::vec3(0.0f, 0.0f, 1.0f));
    moonModelMatrix = glm::translate(moonModelMatrix, glm::vec3(0.0f, 50.0f, 0.0f));

    // turn on point lights
    if (!on_off) { ... }
    // turn off point lights
    else
    {
        for (int i = 0; i < 9; i++)
            specify_point_light(shader, i, glm::vec3(0.0f), glm::vec3(0.0f));
    }
}
```

Point lights in this program are given out by torches and the moon, where the torchlights are flickering, and the moonlight is dynamic. The implementation is to set the brightness of torchlights as a periodic function and calculate the position of the moonlight according to the current frame. Then set the attributes accordingly. Besides, point lights are turned off in daylight mode, so they are extinguished according to the Boolean light mode.

```
void specify_directional_light(Shader shader, bool on_off);
void specify_spotlight(Shader shader, Camera camera, bool on_off);
```

Spotlight is the flashlight in this program, it is set pointing to the front of the camera and is extinguished in daylight mode. Directional light is the daylight in the program, it is extinguished in night mode.

Camera

`Camera` class specifies the attributes of the camera and keyboard/mouse callback functions. In each render loop, first, the camera attributes are updated by keyboard/mouse input, then the view matrix is specified according to the camera attributes.

```
void SetViewPoint(glm::vec3 position, glm::vec3 front, glm::vec3 up, float yaw, float pitch)
{
    Position = position;
    Up = up;
    Front = front;
    Yaw = yaw;
    Pitch = pitch;
}
```

To realize the viewpoint setting functionality, a function is defined in the `Camera` class to set the viewpoint according to passed attributes.

```
// process camera input
processInput_Camera(window, imported_models.get_eagleMatrix());
```

```
// set view point to the eagle
void processInput_Eagle(GLFWwindow* window, glm::mat4 eagleModelMatrix)
{
    if (glfwGetKey(window, GLFW_KEY_P) == GLFW_PRESS)
    {
        glm::vec3 new_position = glm::vec3(eagleModelMatrix[3][0], eagleModelMatrix[3][1] + 25.0f, eagleModelMatrix[3][2]);
        glm::vec3 new_front = glm::vec3(0.0f, -1.0f, 0.0f);
        glm::vec3 new_up = glm::vec3(0.0f, 0.0f, -1.0f);
        float new_yaw = -90.0f;
        float new_pitch = -90.0f;
        camera.SetViewPoint(new_position, new_front, new_up, new_yaw, new_pitch);
    }
}
```

For the functionality of fixing the camera at the eagle, the position of the eagle is passed from the `imported_models` object to the input processing function. Position and direction of the camera is specified accordingly while the input is called.