浙江大学

《计算机图形学》
实验报告

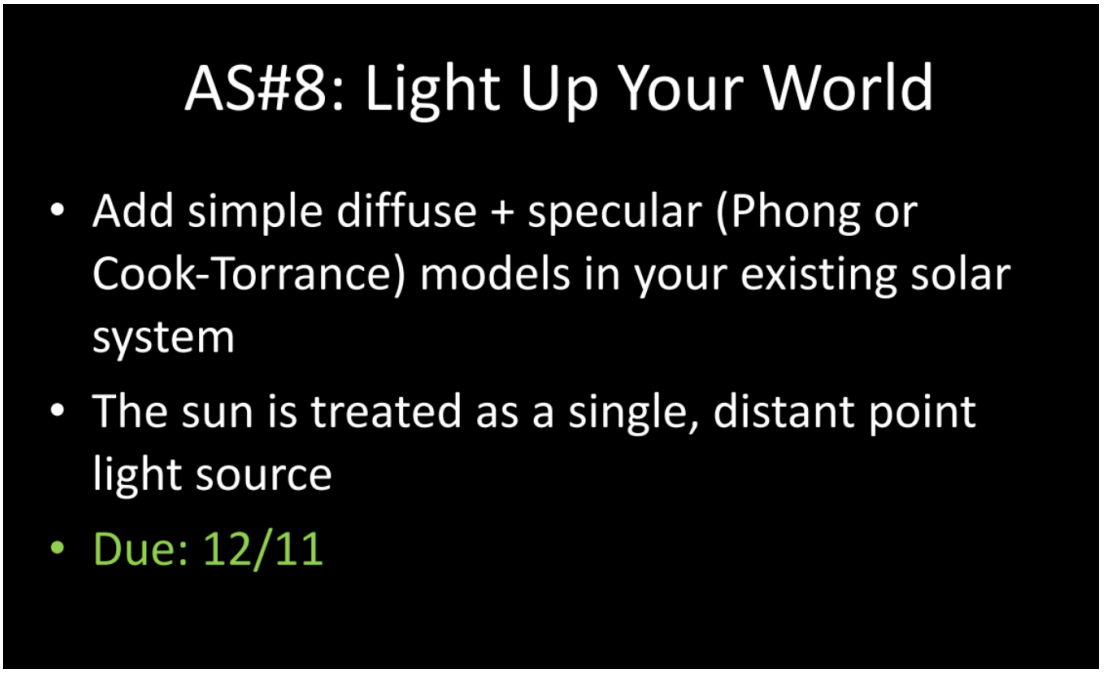
实验名称	：	LightUp_World
姓　　名	：	王晓宇
学　　号	：	3220104364
电子邮箱	：	3220104364@zju.edu.cn
联系电话	：	19550222634
授课教师	：	吴鸿智
助　　教	：	丁华铿

2024 年 12 月 8 日

LightUp_World

LightUp_World

1 实验内容及简要原理介绍



1.1 实验内容

本次实验旨在通过OpenGL和GLAD库实现一个简单的太阳系模拟程序。程序将展示一个太阳和三个行星，以及它们的卫星，模拟它们围绕太阳的旋转和轨道运动，此外项目实现了漫反射与镜面反射效果。

- 在现有的太阳系中添加简单的漫反射+镜面反射（Phong或Cook-Torrance）模型。
- 太阳被视为一个单一的、遥远的点光源。

1.2 简要原理介绍

OpenGL(Open Graphics Library)是一个跨语言、跨平台的图形API，用于渲染2D和3D矢量图形。本实验中，我们利用OpenGL的3D图形渲染能力和GLEW的窗口管理功能，创建了一个太阳系的动态模拟。

Phong光照模型是一种用于计算机图形学中模拟光照效果的经典模型，它由Bui Tuong Phong在1975年提出。Phong模型结合了漫反射（Diffuse Reflection）和镜面反射（Specular Reflection）两种光照效果，能够较为真实地模拟物体表面的光照效果。

1.2.1 漫反射（Diffuse Reflection）

漫反射是指光线照射到物体表面后，均匀地向各个方向反射的现象。漫反射的光强与光源的方向和物体表面的法线方向有关，但与观察者的位置无关。

公式：

$$I_d = k_d \cdot I_l \cdot \max(0, \vec{N} \cdot \vec{L})$$

- I_d：漫反射光强度。
- k_d：物体表面的漫反射系数（颜色）。
- I_l：光源的强度。
- $\vec{N}$：物体表面的法线向量。
- $\vec{L}$：从物体表面到光源的向量。
- $\max(0, \vec{N} \cdot \vec{L})$：确保法线与光源向量的点积为非负值，避免负值导致光照错误。

1.2.2 镜面反射（Specular Reflection）

镜面反射是指光线照射到物体表面后，沿着反射方向集中反射的现象。镜面反射的光强与光源的方向、物体表面的法线方向以及观察者的位置有关。

公式：

$$I_s = k_s \cdot I_l \cdot \max(0, \vec{V} \cdot \vec{R})^n$$

- I_s：镜面反射光强度。
- k_s：物体表面的镜面反射系数（颜色）。
- I_l：光源的强度。
- $\vec{V}$：从物体表面到观察者的向量。
- $\vec{R}$：光线反射方向的向量。

- n：镜面反射的指数，控制镜面高光的锐利程度。n越大，高光越集中。
- $\max(0, \vec{V} \cdot \vec{R})$：确保观察者向量与反射向量的点积为非负值。

1.2.3 总光照强度

Phong模型将漫反射和镜面反射结合起来，得到物体表面的总光照强度：

$$I = I_a + I_d + I_s$$

- I_a：环境光强度，用于模拟全局光照效果，通常是一个常数。
- I_d：漫反射光强度。
- I_s：镜面反射光强度。
- **漫反射**模拟了光线均匀反射的效果，与观察者位置无关，主要影响物体表面的整体亮度。
- **镜面反射**模拟了光线集中反射的效果，与观察者位置有关，主要影响物体表面的高光部分。

2 实验框架思路与代码实现

2.1 实验框架思路

这个主程序的框架思路是使用 OpenGL 和 GLFW 创建一个窗口，并在其中进行 3D 图形渲染。以下是对主程序框架的详细介绍：

2.1.1 主程序main.cpp

- 首先，程序包含了一些必要的头文件，这些头文件提供了 OpenGL、GLFW 以及其他工具和库的功能。
- 接下来，程序定义了一些回调函数和实用函数，用于处理窗口大小变化、鼠标输入和加载纹理等操作。

```
void framebuffer_size_callback(GLFWwindow *window, int width,
int height);
void mouse_callback(GLFWwindow *window, double xpos, double
ypos);
void processInput(GLFWwindow *window);
unsigned int loadTexture(char const *path);
unsigned int loadCubemap(vector<std::string> faces);
```

- 程序定义了一些全局变量，用于存储屏幕尺寸、时间增量、鼠标位置和摄像机对象等信息。
- main是程序的入口点，负责初始化 GLFW、创建窗口、加载 OpenGL 函数指针、设置回调函数、加载资源并进入渲染循环。
    1. 初始化 GLFW 并创建窗口。
    2. 加载 OpenGL 函数指针。
    3. 设置回调函数和 OpenGL 状态。
    4. 加载资源（如着色器、纹理、模型等）。
    5. 进入渲染循环，处理输入并绘制场景。
    6. 释放资源并终止 GLFW。

2.1.2 Phong实现

实现Phong光照模型主要是依靠着色器（Shader）来完成的。着色器是OpenGL中用于在图形渲染管线中执行特定任务的程序，主要包括顶点着色器（Vertex Shader）和片段着色器（Fragment Shader）。

在Phong光照模型中，顶点着色器通常会传递以下信息到片段着色器：

- 顶点的位置（经过模型视图变换和投影变换后的位置）。
- 顶点的法线（经过模型视图变换后的法线）。
- 顶点的纹理坐标（如果需要纹理映射）。

在Phong光照模型中，片段着色器会根据顶点着色器传递过来的信息，计算漫反射和镜面反射的光照效果，并结合环境光，最终输出像素的颜色。

实现细节按照理论原理实现即可，我们在下方的代码实现中有更为详细的解释。

2.2 代码实现

2.2.1 main.cpp

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <iostream>
#include <cmath>

#define GLM_ENABLE_EXPERIMENTAL
```

```cpp
#include <geometry/BoxGeometry.h>
#include <geometry/PlaneGeometry.h>
#include <geometry/SphereGeometry.h>

#define STB_IMAGE_IMPLEMENTATION
#include <tool/stb_image.h>

#include <tool/gui.h>
#include <tool/mesh.h>
#include <tool/model.h>
#include <tool/Torus.hpp>
#include <tool/shader.h>
#include <tool/camera.h>

void framebuffer_size_callback(GLFWwindow *window, int width, int height);
void mouse_callback(GLFWwindow *window, double xpos, double ypos);
void processInput(GLFWwindow *window);
unsigned int loadTexture(char const *path);
unsigned int loadCubemap(vector<std::string> faces);

std::string Shader::dirName;

int SCREEN_WIDTH = 800;
int SCREEN_HEIGHT = 600;

// delta time
float deltaTime = 0.0f;
float lastTime = 0.0f;

float lastX = SCREEN_WIDTH / 2.0f; // 鼠标上一帧的位置
float lastY = SCREEN_HEIGHT / 2.0f;

Camera camera(glm::vec3(0.0, 0.0, 5.0), glm::vec3(0.0, 1.0, 0.0));

using namespace std;
```

```
42
43  int main(int argc, char *argv[])
44  {
45      Shader::dirName = argv[1];
46      glfwInit();
47      // 设置主要和次要版本
48      const char *glsl_version = "#version 330";
49
50      // 片段着色器将作用域每一个采样点（采用4倍抗锯齿，则每个像素有4个片段（四个采样点））
51      // glfwWindowHint(GLFW_SAMPLES, 4);
52      glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
53      glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
54      glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
55
56      // 窗口对象
57      GLFWwindow *window = glfwCreateWindow(SCREEN_WIDTH, SCREEN_HEIGHT, "LearnOpenGL", NULL, NULL);
58      if (window == NULL)
59      {
60          std::cout << "Failed to create GLFW window" << std::endl;
61          glfwTerminate();
62          return -1;
63      }
64      glfwMakeContextCurrent(window);
65
66      if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
67      {
68          std::cout << "Failed to initialize GLAD" << std::endl;
69          return -1;
70      }
71
72      // ----------------------
73      // 创建imgui上下文
74      ImGui::CreateContext();
75      ImGuiIO &io = ImGui::GetIO();
```

```
76      (void)io;
77      // 设置样式
78      ImGui::StyleColorsDark();
79      // 设置平台和渲染器
80      ImGui_ImplGlfw_InitForOpenGL(window, true);
81      ImGui_ImplOpenGL3_Init(glsl_version);
82
83      // ------------------------
84
85      // 设置视口
86      glViewport(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT);
87      glEnable(GL_PROGRAM_POINT_SIZE);
88      glEnable(GL_BLEND);
89      glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
90
91      glEnable(GL_DEPTH_TEST);
92      // glDepthFunc(GL_LESS);
93
94      // 鼠标键盘事件
95      // 1.注册窗口变化监听
96      glfwSetFramebufferSizeCallback(window,
framebuffer_size_callback);
97      // 2.鼠标事件
98      glfwSetCursorPosCallback(window, mouse_callback);
99
100     Shader Simple_Shader("./shader/vertex.glsl",
"./shader/fragment.glsl");
101     Shader Light_Shader("./shader/light_vert.glsl",
"./shader/light_frag.glsl");
102     Shader Skybox_Shader("./shader/skybox_vert.glsl",
"./shader/skybox_frag.glsl");
103
104     SphereGeometry Sun_1(0.2, 50, 50);
105     SphereGeometry Sun_2(0.2, 50, 50);
106     SphereGeometry Planet_1(0.1f, 50, 50);
107     SphereGeometry Satellite_of_p1(0.03f, 50, 50);
```

```
    SphereGeometry Planet_2(0.12f, 50, 50);
    SphereGeometry Satellite_of_p2(0.03f, 50, 50);
    SphereGeometry Planet_3(0.28f, 50, 50);
    Torus Sun_torus(0.003, 0.7, 100, 50);
    Torus Planet1_torus(0.003, 1.5, 100, 50);
    Torus Satellite_of_p1_torus(0.003, 0.5, 100, 50);
    Torus Planet2_torus(0.003, 2.5, 100, 50);
    Torus Satellite_of_p2_torus(0.003, 0.5, 100, 50);
    Torus Planet3_torus(0.003, 3.5, 100, 50);

    float skyboxVertices[] = {
        // positions
        -1.0f, 1.0f, -1.0f,
        -1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,
        1.0f, 1.0f, -1.0f,
        -1.0f, 1.0f, -1.0f,

        -1.0f, -1.0f, 1.0f,
        -1.0f, -1.0f, -1.0f,
        -1.0f, 1.0f, -1.0f,
        -1.0f, 1.0f, -1.0f,
        -1.0f, 1.0f, 1.0f,
        -1.0f, -1.0f, 1.0f,

        1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, 1.0f,
        1.0f, 1.0f, 1.0f,
        1.0f, 1.0f, 1.0f,
        1.0f, 1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,

        -1.0f, -1.0f, 1.0f,
        -1.0f, 1.0f, 1.0f,
        1.0f, 1.0f, 1.0f,
```

```
        1.0f, 1.0f, 1.0f,
        1.0f, -1.0f, 1.0f,
        -1.0f, -1.0f, 1.0f,

        -1.0f, 1.0f, -1.0f,
        1.0f, 1.0f, -1.0f,
        1.0f, 1.0f, 1.0f,
        1.0f, 1.0f, 1.0f,
        -1.0f, 1.0f, 1.0f,
        -1.0f, 1.0f, -1.0f,

        -1.0f, -1.0f, -1.0f,
        -1.0f, -1.0f, 1.0f,
        1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,
        -1.0f, -1.0f, 1.0f,
        1.0f, -1.0f, 1.0f};
    // 加载天空盒
    vector<std::string> faces{
        "./textures/skybox/right.jpg",
        "./textures/skybox/left.jpg",
        "./textures/skybox/top.jpg",
        "./textures/skybox/bottom.jpg",
        "./textures/skybox/front.jpg",
        "./textures/skybox/back.jpg"};

    unsigned int cubemapTexture = loadCubemap(faces);

    unsigned int diffuseMap_Sun =
loadTexture("./textures/sun.jpg");
    unsigned int specularMap_Sun =
loadTexture("./textures/sun.jpg");
    unsigned int diffuseMap_Earth =
loadTexture("./textures/earth.jpg");
    unsigned int specularMap_Earth =
loadTexture("./textures/earth_specular.jpg");
```

```
176     unsigned int diffuseMap_Moon =
    loadTexture("./textures/moon.jpg");
177     unsigned int specularMap_Moon =
    loadTexture("./textures/moon.jpg");
178     unsigned int diffuseMap_Mars =
    loadTexture("./textures/mars.jpg");
179     unsigned int specularMap_Mars =
    loadTexture("./textures/mars.jpg");
180 
181     Simple_Shader.use();
182     Simple_Shader.setInt("material.diffuse", 0);
183     Simple_Shader.setInt("material.specular", 1);
184     // 传递材质属性
185     Simple_Shader.setVec3("material.ambient", 1.0f, 0.5f, 0.31f);
186     Simple_Shader.setFloat("material.shininess", 32.0f);
187 
188     float fov = 45.0f; // 视锥体的角度
189     glm::vec3 view_translate = glm::vec3(0.0, 0.0, -5.0);
190     ImVec4 clear_color = ImVec4(25.0 / 255.0, 25.0 / 255.0, 25.0 /
    255.0, 1.0); // 25, 25, 25
191 
192     // 行星位置
193     glm::vec3 _planet_postions[] = {
194         glm::vec3(0.7f, 0.0f, 0.0f),
195         glm::vec3(-0.7f, 0.0f, 0.0f),
196         glm::vec3(1.5f, 0.0f, 0.0f),
197         glm::vec3(-0.5f, 0.0f, 0.0f),
198         glm::vec3(-2.5f, 0.0f, 0.0f),
199         glm::vec3(-0.5f, 0.0f, 0.0f),
200         glm::vec3(3.5f, 0.0f, 0.0f)};
201     // 行星颜色
202     glm::vec3 _planet_colors[] = {
203         glm::vec3(1.0f, 0.55f, 0.0f),
204         glm::vec3(1.0f, 0.55f, 0.0f),
205         glm::vec3(0.0f, 0.0f, 1.0f),
206         glm::vec3(0.0f, 1.0f, 0.0f),
```

```
        glm::vec3(1.0f, 0.0f, 0.0f),
        glm::vec3(0.0f, 1.0f, 0.0f),
        glm::vec3(0.7f, 0.5f, 0.5f)};
    // 轨道颜色
    glm::vec3 _orbit_color = glm::vec3(0.7f, 0.7f, 0.7f);

    while (!glfwWindowShouldClose(window))
    {
        processInput(window);

        float currentFrame = glfwGetTime();
        deltaTime = currentFrame - lastTime;
        lastTime = currentFrame;

        glfwSetWindowTitle(window, "LightUp_world");

        // 渲染指令
        // ...
        glClearColor(clear_color.x, clear_color.y, clear_color.z,
clear_color.w);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        Simple_Shader.use();
        glm::mat4 view = camera.GetViewMatrix();
        glm::mat4 projection = glm::mat4(1.0f);
        projection = glm::perspective(glm::radians(fov),
(float)SCREEN_WIDTH / (float)SCREEN_HEIGHT, 0.1f, 100.0f);

        Simple_Shader.setMat4("view", view);
        Simple_Shader.setMat4("projection", projection);
        Simple_Shader.setVec3("viewPos", camera.Position);

        // 设置点光源属性
        for (unsigned int i = 0; i < 2; i++)
        {
```

```
            Simple_Shader.setVec3("pointLights[" +
std::to_string(i) + "].position", _planet_postions[i]);
            Simple_Shader.setVec3("pointLights[" +
std::to_string(i) + "].ambient", 0.01f, 0.01f, 0.01f);
            // Simple_Shader.setVec3("pointLights[" +
std::to_string(i) + "].diffuse", _planet_colors[i]);
            Simple_Shader.setVec3("pointLights[" +
std::to_string(i) + "].diffuse", glm::vec3(1.0f, 1.0f, 1.0f));
            Simple_Shader.setVec3("pointLights[" +
std::to_string(i) + "].specular", 1.0f, 1.0f, 1.0f);

            // // 设置衰减
            Simple_Shader.setFloat("pointLights[" +
std::to_string(i) + "].constant", 1.0f);
            Simple_Shader.setFloat("pointLights[" +
std::to_string(i) + "].linear", 0.09f);
            Simple_Shader.setFloat("pointLights[" +
std::to_string(i) + "].quadratic", 0.032f);
        }
        glm::mat4 model = glm::mat4(1.0f);
        // 绘制轨道
        {
            // Sun
            glActiveTexture(GL_TEXTURE0);
            glBindTexture(GL_TEXTURE_2D, 0);
            glActiveTexture(GL_TEXTURE1);
            glBindTexture(GL_TEXTURE_2D, 0);

            model = glm::mat4(1.0f);
            Simple_Shader.setMat4("model", model);
            Simple_Shader.setVec3("lightColor", _orbit_color);
            Sun_torus.draw();
            // Planet_1
            model = glm::mat4(1.0f);
            model = glm::rotate(model, glm::radians(135.0f),
glm::vec3(1.0f, 0.0f, 0.0f));
```

```
            Simple_Shader.setMat4("model", model);
            Simple_Shader.setVec3("lightColor", _orbit_color);
            Planet1_torus.draw();
            // Satellite_of_p1
            model = glm::mat4(1.0f);
            model = glm::rotate(model, glm::radians(1.0f * 150.0f * (float)glfwGetTime()), glm::vec3(0.0f, 1.0f, 1.0f));
            model = glm::translate(model, (glm::vec3(1.5f, 0.0f, 0.0f)));
            Simple_Shader.setMat4("model", model);
            Simple_Shader.setVec3("lightColor", _orbit_color);
            Satellite_of_p1_torus.draw();
            // Planet_2
            model = glm::mat4(1.0f);
            model = glm::rotate(model, glm::radians(45.0f), glm::vec3(1.0f, 0.0f, 0.0f));
            Simple_Shader.setMat4("model", model);
            Simple_Shader.setVec3("lightColor", _orbit_color);
            Planet2_torus.draw();
            // Satellite_of_p2
            model = glm::mat4(1.0f);
            model = glm::rotate(model, glm::radians(0.5f * 150.0f * (float)glfwGetTime()), glm::vec3(0.0f, -1.0f, 1.0f));
            model = glm::translate(model, (glm::vec3(-2.5f, 0.0f, 0.0f)));
            Simple_Shader.setMat4("model", model);
            Simple_Shader.setVec3("lightColor", _orbit_color);
            Satellite_of_p2_torus.draw();
            // Planet_3
            model = glm::mat4(1.0f);
            Simple_Shader.setMat4("model", model);
            Simple_Shader.setVec3("lightColor", _orbit_color);
            Planet3_torus.draw();
        }
        // 绘制行星
        {
```

```
            // Planet_1
            model = glm::mat4(1.0f);
            model = glm::rotate(model, glm::radians(1.0f * 150.0f * (float)glfwGetTime()), glm::vec3(0.0f, 1.0f, 1.0f));
            model = glm::translate(model, _planet_postions[2]);
            Simple_Shader.setMat4("model", model);
            Simple_Shader.setVec3("lightColor", _planet_colors[2]);

            glActiveTexture(GL_TEXTURE0);
            glBindTexture(GL_TEXTURE_2D, diffuseMap_Earth);
            glActiveTexture(GL_TEXTURE1);
            glBindTexture(GL_TEXTURE_2D, specularMap_Earth);

            glBindVertexArray(Planet_1.VAO);
            glDrawElements(GL_TRIANGLES, Planet_1.indices.size(), GL_UNSIGNED_INT, 0);
            // Satellite_of_p1
            model = glm::mat4(1.0f);
            model = glm::rotate(model, glm::radians(1.0f * 150.0f * (float)glfwGetTime()), glm::vec3(0.0f, 1.0f, 1.0f));
            model = glm::translate(model, (glm::vec3(1.5f, 0.0f, 0.0f)));
            model = glm::rotate(model, glm::radians(0.2f * 100.0f * (float)glfwGetTime()), glm::vec3(0.0f, 0.0f, 1.0f));
            model = glm::translate(model, _planet_postions[3]);
            Simple_Shader.setMat4("model", model);
            Simple_Shader.setVec3("lightColor", _planet_colors[3]);

            glActiveTexture(GL_TEXTURE0);
            glBindTexture(GL_TEXTURE_2D, diffuseMap_Moon);
            glActiveTexture(GL_TEXTURE1);
            glBindTexture(GL_TEXTURE_2D, specularMap_Moon);

            glBindVertexArray(Satellite_of_p1.VAO);
```

```
            glDrawElements(GL_TRIANGLES, Satellite_of_p1.indices.size(), GL_UNSIGNED_INT, 0);
            // Planet_2
            model = glm::mat4(1.0f);
            model = glm::rotate(model, glm::radians(0.5f * 150.0f * (float)glfwGetTime()), glm::vec3(0.0f, -1.0f, 1.0f));
            model = glm::translate(model, _planet_postions[4]);
            Simple_Shader.setMat4("model", model);
            Simple_Shader.setVec3("lightColor", _planet_colors[4]);

            glActiveTexture(GL_TEXTURE0);
            glBindTexture(GL_TEXTURE_2D, diffuseMap_Mars);
            glActiveTexture(GL_TEXTURE1);
            glBindTexture(GL_TEXTURE_2D, specularMap_Mars);

            glBindVertexArray(Planet_2.VAO);
            glDrawElements(GL_TRIANGLES, Planet_2.indices.size(), GL_UNSIGNED_INT, 0);
            // Satellite_of_p2
            model = glm::mat4(1.0f);
            model = glm::rotate(model, glm::radians(0.5f * 150.0f * (float)glfwGetTime()), glm::vec3(0.0f, -1.0f, 1.0f));
            model = glm::translate(model, (glm::vec3(-2.5f, 0.0f, 0.0f)));
            model = glm::rotate(model, glm::radians(0.1f * 100.0f * (float)glfwGetTime()), glm::vec3(0.0f, 0.0f, 1.0f));
            model = glm::translate(model, _planet_postions[5]);
            Simple_Shader.setMat4("model", model);
            Simple_Shader.setVec3("lightColor", _planet_colors[5]);

            glActiveTexture(GL_TEXTURE0);
            glBindTexture(GL_TEXTURE_2D, diffuseMap_Moon);
            glActiveTexture(GL_TEXTURE1);
            glBindTexture(GL_TEXTURE_2D, specularMap_Moon);
```

```
355
356             glBindVertexArray(Satellite_of_p2.VAO);
357             glDrawElements(GL_TRIANGLES,
Satellite_of_p2.indices.size(), GL_UNSIGNED_INT, 0);
358             // Planet_3
359             model = glm::mat4(1.0f);
360             model = glm::rotate(model, glm::radians(0.05f * 150.0f
* (float)glfwGetTime()), glm::vec3(0.0f, 0.0f, 1.0f));
361             model = glm::translate(model, _planet_postions[6]);
362             Simple_Shader.setMat4("model", model);
363             Simple_Shader.setVec3("lightColor",
_planet_colors[6]);
364             // Simple_Shader.setVec3("lightColor", glm::vec3(1.0f,
1.0f, 1.0f));
365
366             glActiveTexture(GL_TEXTURE0);
367             glBindTexture(GL_TEXTURE_2D, diffuseMap_Earth);
368             glActiveTexture(GL_TEXTURE1);
369             glBindTexture(GL_TEXTURE_2D, specularMap_Earth);
370
371             glBindVertexArray(Planet_3.VAO);
372             glDrawElements(GL_TRIANGLES, Planet_3.indices.size(),
GL_UNSIGNED_INT, 0);
373         }
374
375         // 绘制恒星
376         {
377             Light_Shader.use();
378             Light_Shader.setMat4("view", view);
379             Light_Shader.setMat4("projection", projection);
380             // //Sun_1
381             model = glm::mat4(1.0f);
382             model = glm::rotate(model, glm::radians(0.1f * 150.0f
* (float)glfwGetTime()), glm::vec3(0.0f, 0.0f, 1.0f));
383             model = glm::translate(model, _planet_postions[0]);
384             Light_Shader.setMat4("model", model);
```

```
            // Light_Shader.setVec3("lightColor", _planet_colors[0]);
            Light_Shader.setInt("diffuse", 0);
            Light_Shader.setInt("specular", 1);

            glActiveTexture(GL_TEXTURE0);
            glBindTexture(GL_TEXTURE_2D, diffuseMap_Sun);
            glActiveTexture(GL_TEXTURE1);
            glBindTexture(GL_TEXTURE_2D, specularMap_Sun);

            glBindVertexArray(Sun_1.VAO);
            glDrawElements(GL_TRIANGLES, Sun_1.indices.size(), GL_UNSIGNED_INT, 0);
            // Sun_2
            model = glm::mat4(1.0f);
            model = glm::rotate(model, glm::radians(0.1f * 150.0f * (float)glfwGetTime()), glm::vec3(0.0f, 0.0f, 1.0f));
            model = glm::translate(model, _planet_postions[1]);
            Light_Shader.setMat4("model", model);
            Light_Shader.setVec3("lightColor", _planet_colors[1]);
            glBindVertexArray(Sun_2.VAO);
            glDrawElements(GL_TRIANGLES, Sun_2.indices.size(), GL_UNSIGNED_INT, 0);
        }

        // 绘制天空盒
        {
            unsigned int skyboxVAO, skyboxVBO;
            glGenVertexArrays(1, &skyboxVAO);
            glGenBuffers(1, &skyboxVBO);
            glBindVertexArray(skyboxVAO);
            glBindBuffer(GL_ARRAY_BUFFER, skyboxVBO);
            glBufferData(GL_ARRAY_BUFFER, sizeof(skyboxVertices), &skyboxVertices, GL_STATIC_DRAW);
            glEnableVertexAttribArray(0);
```

```
            glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 *
sizeof(float), (void *)0);
            glBindVertexArray(0);
            glDepthFunc(GL_LEQUAL);
            Skybox_Shader.use();
            // view = camera.GetViewMatrix();
            //  重点代码：取4x4矩阵左上角的3x3矩阵来移除变换矩阵的位移
部分，再变回4x4矩阵。///
            //  防止摄像机移动，天空盒会受到视图矩阵的影响而改变位置，
即摄像机向z后退，天空盒和cube向z前进
            view = glm::mat4(glm::mat3(camera.GetViewMatrix()));
            Skybox_Shader.setMat4("view", view);
            Skybox_Shader.setMat4("projection", projection);
            glBindVertexArray(skyboxVAO);
            glActiveTexture(GL_TEXTURE0);
            glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture); //
第一个参数从GL_TEXTURE_2D 变为GL_TEXTURE_CUBE_MAP
            glDrawArrays(GL_TRIANGLES, 0, 36);
            glBindVertexArray(0);
            glDepthFunc(GL_LESS);
        }

        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    Sun_1.dispose();
    Sun_2.dispose();
    Planet_1.dispose();
    Satellite_of_p1.dispose();
    Planet_2.dispose();
    Satellite_of_p2.dispose();
    Planet_3.dispose();
    glfwTerminate();
```

```
447     return 0;
448 }
449 
450 // 窗口变动监听
451 void framebuffer_size_callback(GLFWwindow *window, int width, int
    height)
452 {
453     glViewport(0, 0, width, height);
454 }
455 
456 // 键盘输入监听
457 void processInput(GLFWwindow *window)
458 {
459     if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
460     {
461         glfwSetWindowShouldClose(window, true);
462     }
463 
464     // 相机按键控制
465     // 相机移动
466     if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
467     {
468         camera.ProcessKeyboard(FORWARD, deltaTime);
469     }
470     if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
471     {
472         camera.ProcessKeyboard(BACKWARD, deltaTime);
473     }
474     if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
475     {
476         camera.ProcessKeyboard(LEFT, deltaTime);
477     }
478     if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
479     {
480         camera.ProcessKeyboard(RIGHT, deltaTime);
481     }
```

```
}

// 鼠标移动监听
void mouse_callback(GLFWwindow *window, double xpos, double ypos)
{

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos;

    lastX = xpos;
    lastY = ypos;

    camera.ProcessMouseMovement(xoffset, yoffset);
}

// 加载纹理贴图
unsigned int loadTexture(char const *path)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);

    // 图像y轴翻转
    stbi_set_flip_vertically_on_load(true);
    int width, height, nrComponents;
    unsigned char *data = stbi_load(path, &width, &height,
&nrComponents, 0);
    if (data)
    {
        GLenum format;
        if (nrComponents == 1)
            format = GL_RED;
        else if (nrComponents == 3)
            format = GL_RGB;
        else if (nrComponents == 4)
            format = GL_RGBA;

```

```
517         glBindTexture(GL_TEXTURE_2D, textureID);
518         glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0,
format, GL_UNSIGNED_BYTE, data);
519         glGenerateMipmap(GL_TEXTURE_2D);
520 
521         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_REPEAT);
522         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_REPEAT);
523         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
524         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
525 
526         stbi_image_free(data);
527     }
528     else
529     {
530         std::cout << "Texture failed to load at path: " << path <<
std::endl;
531         stbi_image_free(data);
532     }
533 
534     return textureID;
535 }
536 
537 unsigned int loadCubemap(vector<std::string> faces)
538 {
539     unsigned int textureID;
540     glGenTextures(1, &textureID);
541     glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
542 
543     int width, height, nrChannels;
544     for (unsigned int i = 0; i < faces.size(); i++)
545     {
```

```
546         unsigned char *data = stbi_load(faces[i].c_str(), &width,
&height, &nrChannels, 0);
547         if (data)
548         {
549             glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0,
GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
550             stbi_image_free(data);
551         }
552         else
553         {
554             std::cout << "Cubemap texture failed to load at path:"
<< faces[i] << std::endl;
555             stbi_image_free(data);
556         }
557     }
558     glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
559     ;
560     glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
561     ;
562     glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
563     ;
564     glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
565     ;
566     glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R,
GL_CLAMP_TO_EDGE);
567     ;
568
569     return textureID;
570 }
571
```

大体框架如2.1.1介绍，这里不再赘述

2.2.2 vertex.glsl

```glsl
#version 330 core
layout(location = 0) in vec3 Position;
layout(location = 1) in vec3 Normal;
layout(location = 2) in vec2 TexCoords;

out vec2 outTexCoord;
out vec3 outNormal;
out vec3 outFragPos;

uniform float factor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {

  gl_Position = projection * view * model * vec4(Position, 1.0f);

  outFragPos = vec3(model * vec4(Position, 1.0));

  outTexCoord = TexCoords;
  // 解决不等比缩放，对法向量产生的影响
  outNormal = mat3(transpose(inverse(model))) * Normal;
}
```

这段 GLSL 顶点着色器代码的主要功能是将顶点从模型空间转换到裁剪空间，并计算出片段着色器所需的各种属性，如法线、纹理坐标和片段位置。

1. 输入和输出变量

```
layout(location = 0) in vec3 Position;
layout(location = 1) in vec3 Normal;
layout(location = 2) in vec2 TexCoords;

out vec2 outTexCoord;
out vec3 outNormal;
out vec3 outFragPos;
```

- `Position`：顶点的位置，来自顶点缓冲对象。
- `Normal`：顶点的法线，来自顶点缓冲对象。
- `TexCoords`：顶点的纹理坐标，来自顶点缓冲对象。
- `outTexCoord`：传递给片段着色器的纹理坐标。
- `outNormal`：传递给片段着色器的法线。
- `outFragPos`：传递给片段着色器的片段位置。

2. Uniform 变量

接下来，定义了一些 **uniform** 变量，这些变量在渲染过程中由应用程序传递给着色器：

```
uniform float factor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
```

- `factor`：一个浮点数，用于控制某些效果（在这段代码中未使用）。
- `model`：模型矩阵，用于将顶点从模型空间转换到世界空间。
- `view`：视图矩阵，用于将顶点从世界空间转换到视图空间。
- `projection`：投影矩阵，用于将顶点从视图空间转换到裁剪空间。

3. main

主函数 `main` 负责执行顶点着色器的主要逻辑：

```
void main() {
  // 将顶点从模型空间转换到裁剪空间
  gl_Position = projection * view * model * vec4(Position, 1.0f);

  // 计算片段位置，将顶点从模型空间转换到世界空间
  outFragPos = vec3(model * vec4(Position, 1.0));

  // 传递纹理坐标
  outTexCoord = TexCoords;

  // 解决不等比缩放对法向量的影响，计算法线矩阵并应用于法线
  outNormal = mat3(transpose(inverse(model))) * Normal;
}
```

- **顶点位置转换**:

  ```
  gl_Position = projection * view * model * vec4(Position, 1.0f);
  ```

  这行代码将顶点位置从模型空间转换到裁剪空间。首先将顶点位置乘以模型矩阵，将其转换到世界空间；然后乘以视图矩阵，将其转换到视图空间；最后乘以投影矩阵，将其转换到裁剪空间。

- **片段位置计算**:

  ```
  outFragPos = vec3(model * vec4(Position, 1.0));
  ```

  这行代码计算片段位置，将顶点位置从模型空间转换到世界空间，并传递给片段着色器。

- **传递纹理坐标**:

  ```
  outTexCoord = TexCoords;
  ```

  这行代码将纹理坐标直接传递给片段着色器。

- **法线转换**:

  ```
  outNormal = mat3(transpose(inverse(model))) * Normal;
  ```

  这行代码解决了不等比缩放对法向量的影响。法线需要使用法线矩阵进行变换，法线矩阵是模型矩阵的逆转置矩阵。通过将法线乘以法线矩阵，可以确保法线在进行不等比缩放时保持正确的方向。

2.2.3 fragment.glsl

```
#version 330 core
out vec4 FragColor;

// 点光源
struct PointLight {
  vec3 position;

  float constant;
  float linear;
  float quadratic;

  vec3 ambient;
  vec3 diffuse;
  vec3 specular;
};

// 材质
struct Material {
  sampler2D diffuse; // 漫反射贴图
  sampler2D specular; // 镜面光贴图
  float shininess; // 高光指数
};

#define NR_POINT_LIGHTS 2

uniform Material material;
uniform PointLight pointLights[NR_POINT_LIGHTS];

in vec2 outTexCoord;
in vec3 outNormal;
in vec3 outFragPos;

uniform vec3 viewPos;

```

```
35 vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos,
   vec3 viewDir);
36
37 void main() {
38   vec4 objectColor = vec4(1.0f, 1.0f, 1.0f, 1.0f);
39   vec3 viewDir = normalize(viewPos - outFragPos);
40   vec3 normal = normalize(outNormal);
41   vec3 result = vec3(0.0);
42   // 点光源
43   for(int i = 0; i < NR_POINT_LIGHTS; i++) {
44     result += CalcPointLight(pointLights[i], normal, outFragPos,
   viewDir);
45   }
46
47   FragColor = vec4(result, 1.0);
48 }
49
50
51 // 计算点光源
52 vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos,
   vec3 viewDir) {
53   vec3 lightDir = normalize(light.position - fragPos);
54     // 漫反射着色
55   float diff = max(dot(normal, lightDir), 0.0);
56     // 镜面光着色
57   vec3 reflectDir = reflect(-lightDir, normal);
58   float spec = pow(max(dot(viewDir, reflectDir), 0.0),
   material.shininess);
59     // 衰减
60   float distance = length(light.position - fragPos);
61   float attenuation = 1.0 / (light.constant + light.linear *
   distance +
62     light.quadratic * (distance * distance));
63     // 合并结果
64   vec3 ambient = light.ambient * vec3(texture(material.diffuse,
   outTexCoord));
```

```
  vec3 diffuse = light.diffuse * diff *
vec3(texture(material.diffuse, outTexCoord));
  vec3 specular = light.specular * spec *
vec3(texture(material.specular, outTexCoord));
  ambient *= attenuation;
  diffuse *= attenuation;
  specular *= attenuation;
  return (ambient + diffuse + specular);
}

```

1. 定义结构体

首先，我们定义了两个结构体：`PointLight` 和 `Material`。

```
struct PointLight {
  vec3 position;
  float constant;
  float linear;
  float quadratic;
  vec3 ambient;
  vec3 diffuse;
  vec3 specular;
};

struct Material {
  sampler2D diffuse; // 漫反射贴图
  sampler2D specular; // 镜面光贴图
  float shininess; // 高光指数
};
```

- `PointLight` 结构体包含了点光源的位置、衰减系数（常数、线性和二次项）、环境光、漫反射光和镜面反射光的颜色。
- `Material` 结构体包含了材质的漫反射贴图、镜面光贴图和高光指数。

2. 定义 Uniform 变量

接下来，我们定义了一些 **uniform** 变量，用于在着色器中传递光源和材质的属性。

```
#define NR_POINT_LIGHTS 2

uniform Material material;
uniform PointLight pointLights[NR_POINT_LIGHTS];
uniform vec3 viewPos;
```

- `material`：材质属性。
- `pointLights`：点光源数组，数量由 `NR_POINT_LIGHTS` 定义。
- `viewPos`：观察者的位置。

3. 计算点光源的光照

我们定义了一个辅助函数 `CalcPointLight`，用于计算点光源的光照效果。

```
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir) {
  vec3 lightDir = normalize(light.position - fragPos);
  // 漫反射着色
  float diff = max(dot(normal, lightDir), 0.0);
  // 镜面光着色
  vec3 reflectDir = reflect(-lightDir, normal);
  float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
  // 衰减
  float distance = length(light.position - fragPos);
  float attenuation = 1.0 / (light.constant + light.linear * distance +
    light.quadratic * (distance * distance));
  // 合并结果
  vec3 ambient = light.ambient * vec3(texture(material.diffuse, outTexCoord));
  vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, outTexCoord));
  vec3 specular = light.specular * spec * vec3(texture(material.specular, outTexCoord));
  ambient *= attenuation;
  diffuse *= attenuation;
```

```
  specular *= attenuation;
  return (ambient + diffuse + specular);
}
```

- **漫反射计算**：`float diff = max(dot(normal, lightDir), 0.0);` 计算光线方向和法线方向的点积，用于漫反射分量的计算。
- **镜面反射计算**：`vec3 reflectDir = reflect(-lightDir, normal);` 和 `float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);` 计算反射向量和视线方向的点积，并通过 `pow` 函数计算镜面反射分量。
- **衰减计算**：`float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic * (distance * distance));` 计算光照的衰减。
- **合并结果**：将环境光、漫反射光和镜面反射光分量组合在一起，并考虑衰减。

4. main函数

在 `main` 函数中，我们计算每个点光源的光照效果，并将结果累加到 `result` 变量中。

```
void main() {
  vec4 objectColor = vec4(1.0f, 1.0f, 1.0f, 1.0f);
  vec3 viewDir = normalize(viewPos - outFragPos);
  vec3 normal = normalize(outNormal);
  vec3 result = vec3(0.0);
  // 点光源
  for(int i = 0; i < NR_POINT_LIGHTS; i++) {
    result += CalcPointLight(pointLights[i], normal, outFragPos, viewDir);
  }

  FragColor = vec4(result, 1.0);
}
```

- **视线方向**：`vec3 viewDir = normalize(viewPos - outFragPos);` 计算视线方向。
- **法线向量**：`vec3 normal = normalize(outNormal);` 计算法线向量。

- **累加光照效果**：通过循环遍历所有点光源，调用 `CalcPointLight` 函数计算每个点光源的光照效果，并将结果累加到 `result` 变量中。
- **输出颜色**：`FragColor = vec4(result, 1.0);` 将计算结果赋值给 `FragColor`，用于输出片段颜色。

2.2.4 Torus.hpp

> 我们使用圆环体现轨道，当内半径足够小时，圆环即表现为一个圆

```
#include <vector>
#include <cmath>
#include <glad/glad.h>
#include <GLFW/glfw3.h>

struct Torus_Vertex {
    float Position[3];
    float Normal[3];
    float TexCoords[2];
};

std::vector<Torus_Vertex> GenerateTorusVertices(float innerRadius, float outerRadius, int sides, int rings) {
    std::vector<Torus_Vertex> vertices;
    float ringFactor = (2.0f * PI) / rings;
    float sideFactor = (2.0f * PI) / sides;

    for (int ring = 0; ring <= rings; ++ring) {
        float u = ring * ringFactor;
        float cu = cos(u);
        float su = sin(u);

        for (int side = 0; side <= sides; ++side) {
            float v = side * sideFactor;
            float cv = cos(v);
            float sv = sin(v);

            float x = (outerRadius + innerRadius * cv) * cu;
```

```
            float y = (outerRadius + innerRadius * cv) * su;
            float z = innerRadius * sv;

            Torus_Vertex Torus_vertex;
            Torus_vertex.Position[0] = x;
            Torus_vertex.Position[1] = y;
            Torus_vertex.Position[2] = z;

            Torus_vertex.Normal[0] = cv * cu;
            Torus_vertex.Normal[1] = cv * su;
            Torus_vertex.Normal[2] = sv;

            Torus_vertex.TexCoords[0] = u / (2.0f * PI);
            Torus_vertex.TexCoords[1] = v / (2.0f * PI);

            vertices.push_back(Torus_vertex);
        }
    }

    return vertices;
}

std::vector<unsigned int> GenerateTorusIndices(int sides, int
rings) {
    std::vector<unsigned int> indices;

    for (int ring = 0; ring < rings; ++ring) {
        for (int side = 0; side < sides; ++side) {
            int first = (ring * (sides + 1)) + side;
            int second = first + sides + 1;

            indices.push_back(first);
            indices.push_back(second);
            indices.push_back(first + 1);

            indices.push_back(second);
```

```
            indices.push_back(second + 1);
            indices.push_back(first + 1);
        }
    }

    return indices;
}

class Torus {
private:
    unsigned int VAO,VBO,EBO;
    std::vector<Torus_Vertex> mVertices;
    std::vector<unsigned int> mIndices;
public:
    Torus(float innerRadius, float outerRadius, int sides, int
rings) {
        mVertices = GenerateTorusVertices(innerRadius,
outerRadius, sides, rings);
        mIndices = GenerateTorusIndices(sides, rings);

        glGenVertexArrays(1, &VAO);
        glGenBuffers(1, &VBO);
        glGenBuffers(1, &EBO);

        glBindVertexArray(VAO);

        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER, mVertices.size() *
sizeof(Torus_Vertex), &mVertices[0], GL_STATIC_DRAW);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, mIndices.size() *
sizeof(unsigned int), &mIndices[0], GL_STATIC_DRAW);

        // 设置顶点属性指针
```

```
94          glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
    sizeof(Torus_Vertex), (void*)0);
95          glEnableVertexAttribArray(0);
96
97          glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
    sizeof(Torus_Vertex), (void*)offsetof(Torus_Vertex, Normal));
98          glEnableVertexAttribArray(1);
99
100         glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
    sizeof(Torus_Vertex), (void*)offsetof(Torus_Vertex, TexCoords));
101         glEnableVertexAttribArray(2);
102
103         glBindVertexArray(0);
104
105     }
106     void draw() {
107         // draw the torus
108         glBindVertexArray(VAO);
109         glDrawElements(GL_TRIANGLES, mIndices.size(),
    GL_UNSIGNED_INT, 0);
110         glBindVertexArray(0);
111     }
112 };
```

Torus类的实现原理是生成一个圆环（Torus）的顶点和索引数据，并使用 OpenGL 将这些数据传递到 GPU 进行渲染。

1. 定义一个结构体Torus_Vertex

```
1 struct Torus_Vertex {
2     float Position[3];
3     float Normal[3];
4     float TexCoords[2];
5 };
```

- `Position`：顶点的位置。
- `Normal`：顶点的法线，用于光照计算。

- `TexCoords`：顶点的纹理坐标。

2. 生成圆环顶点数据

> GenerateTorusVertices

- **innerRadius**：圆环的内半径。
- **outerRadius**：圆环的外半径。
- **sides**：圆环的侧面数。
- **rings**：圆环的环数。

这个函数通过双重循环生成圆环的顶点数据。外层循环遍历环，内层循环遍历每个环上的侧面。每个顶点的位置、法线和纹理坐标都被计算并存储在 **Torus_Vertex** 结构体中，然后添加到 **vertices** 向量中。

3. 生成圆环索引数据

> GenerateTorusIndices

```
std::vector<unsigned int> GenerateTorusIndices(int sides, int rings) {
    std::vector<unsigned int> indices;

    for (int ring = 0; ring < rings; ++ring) {
        for (int side = 0; side < sides; ++side) {
            int first = (ring * (sides + 1)) + side;
            int second = first + sides + 1;

            indices.push_back(first);
            indices.push_back(second);
            indices.push_back(first + 1);

            indices.push_back(second);
            indices.push_back(second + 1);
            indices.push_back(first + 1);
        }
    }

```

```
    return indices;
}
```

这个函数通过双重循环生成圆环的索引数据。每个环和侧面都生成两个三角形的索引，用于绘制圆环的表面。

4. `Torus`类的定义

`Torus` 类负责管理圆环的顶点和索引数据，并将这些数据传递到 GPU。

- **构造函数**：生成顶点和索引数据，并将这些数据传递到 GPU。
  - `glGenVertexArrays`、`glGenBuffers`：生成 VAO、VBO 和 EBO。
  - `glBindVertexArray`、`glBindBuffer`、`glBufferData`：绑定 VAO 和 VBO，并将顶点和索引数据传递到 GPU。
  - `glVertexAttribPointer`、`glEnableVertexAttribArray`：设置顶点属性指针，用于顶点位置、法线和纹理坐标。
- `draw` 方法：绑定 VAO 并调用 `glDrawElements`绘制圆环。

2.2.5 light_vertex.glsl

```
#version 330 core
layout(location = 0) in vec3 Position;
layout(location = 1) in vec3 Normal;
layout(location = 2) in vec2 TexCoords;
out vec2 outTexCoord;
out vec3 outNormal;
out vec3 outFragPos;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
  gl_Position = projection * view * model * vec4(Position, 1.0f);
  outTexCoord = TexCoords;
  outFragPos = vec3(model * vec4(Position, 1.0));
    // 解决不等比缩放，对法向量产生的影响
  outNormal = mat3(transpose(inverse(model))) * Normal;
```

```
}
```

> 由于点光源物体本身并不会受到光源计算影响，因此我们需要重新给定一个着色器以绘制这个点光源

顶点着色器用于处理顶点的变换和法向量的计算。顶点着色器是图形渲染管线中的第一个阶段，它接收顶点数据并进行处理，最终输出给片段着色器。

- `layout(location = 0) in vec3 Position;`：顶点位置，使用位置 0。
- `layout(location = 1) in vec3 Normal;`：顶点法向量，使用位置 1。
- `layout(location = 2) in vec2 TexCoords;`：顶点纹理坐标，使用位置 2。
- `out vec2 outTexCoord;`：输出的纹理坐标，将传递给片段着色器。
- `out vec3 outNormal;`：输出的法向量，将传递给片段着色器。
- `out vec3 outFragPos;`：输出的片段位置，将传递给片段着色器。
- `uniform mat4 model;`：模型矩阵，用于将顶点从模型空间变换到世界空间。
- `uniform mat4 view;`：视图矩阵，用于将顶点从世界空间变换到视图空间。
- `uniform mat4 projection;`：投影矩阵，用于将顶点从视图空间变换到裁剪空间。

在 `main` 函数中，首先计算顶点的最终位置，并将其赋值给 `gl_Position`：

```
gl_Position = projection * view * model * vec4(Position, 1.0f);
```

这一步将顶点位置从模型空间依次变换到世界空间、视图空间和裁剪空间。

接下来，将输入的纹理坐标直接传递给输出变量 `outTexCoord`：

```
outTexCoord = TexCoords;
```

然后，计算片段在世界空间中的位置，并将其赋值给输出变量 `outFragPos`：

```
outFragPos = vec3(model * vec4(Position, 1.0));
```

最后，为了解决不等比缩放对法向量产生的影响，我们使用模型矩阵的逆转置矩阵来变换法向量，并将结果赋值给输出变量 `outNormal`：

```
outNormal = mat3(transpose(inverse(model))) * Normal;
```

这段代码确保了法向量在经过不等比缩放后仍然保持正确的方向。

通过这些操作，顶点着色器将顶点位置、纹理坐标和法向量传递给片段着色器，以便在后续的渲染过程中使用。

2.2.6 light_fragment.glsl

```glsl
#version 330 core
out vec4 FragColor;
in vec2 outTexCoord;

uniform vec3 lightColor;
uniform sampler2D diffuse; // 漫反射贴图
uniform sampler2D specular; // 镜面光贴图
void main() {

  vec3 temp = vec3(texture(diffuse, outTexCoord));
  temp += vec3(texture(specular, outTexCoord));
  // FragColor = vec4(temp, 1.0);
  FragColor = vec4(vec3(1.0f,1.0f,1.0f) * temp, 1.0);
}
```

> 由于点光源物体本身并不会受到光源计算影响，因此我们需要重新给定一个着色器以绘制这个点光源

1. **输入和输出变量**：

```glsl
out vec4 FragColor;
in vec2 outTexCoord;
```

   - `FragColor`：输出的片段颜色。
   - `outTexCoord`：从顶点着色器传递过来的纹理坐标。

2. **uniform 变量**：

```glsl
uniform vec3 lightColor;
uniform sampler2D diffuse; // 漫反射贴图
uniform sampler2D specular; // 镜面光贴图
```

   - `lightColor`：光源的颜色。
   - `diffuse`：漫反射贴图的采样器。

- `specular`：镜面光贴图的采样器。

3. 在 `main` 函数中进行纹理采样和颜色计算：

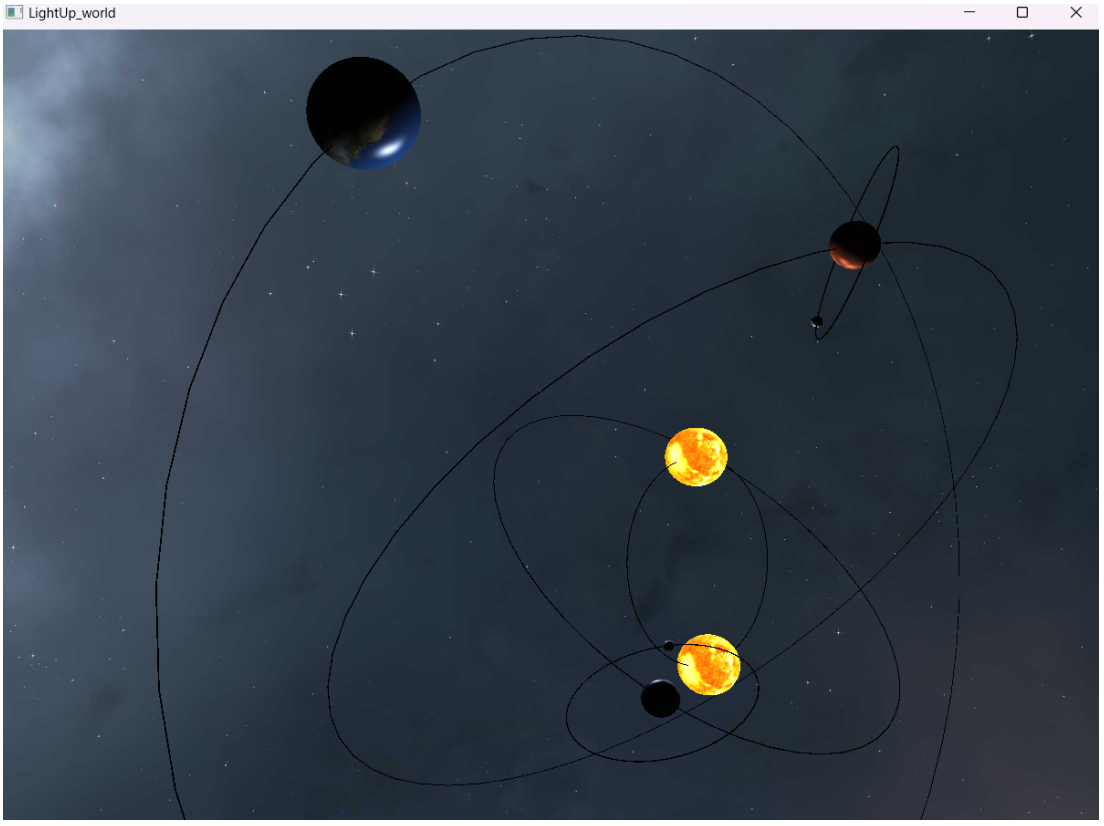
```
void main() {
    // 从漫反射贴图中采样颜色
    vec3 diffuseColor = vec3(texture(diffuse, outTexCoord));

    // 从镜面光贴图中采样颜色
    vec3 specularColor = vec3(texture(specular, outTexCoord));

    // 将漫反射和镜面反射的颜色值相加
    vec3 temp = diffuseColor + specularColor;

    // 将光源颜色与纹理颜色相乘，得到最终的片段颜色
    FragColor = vec4(lightColor * temp, 1.0);
}
```

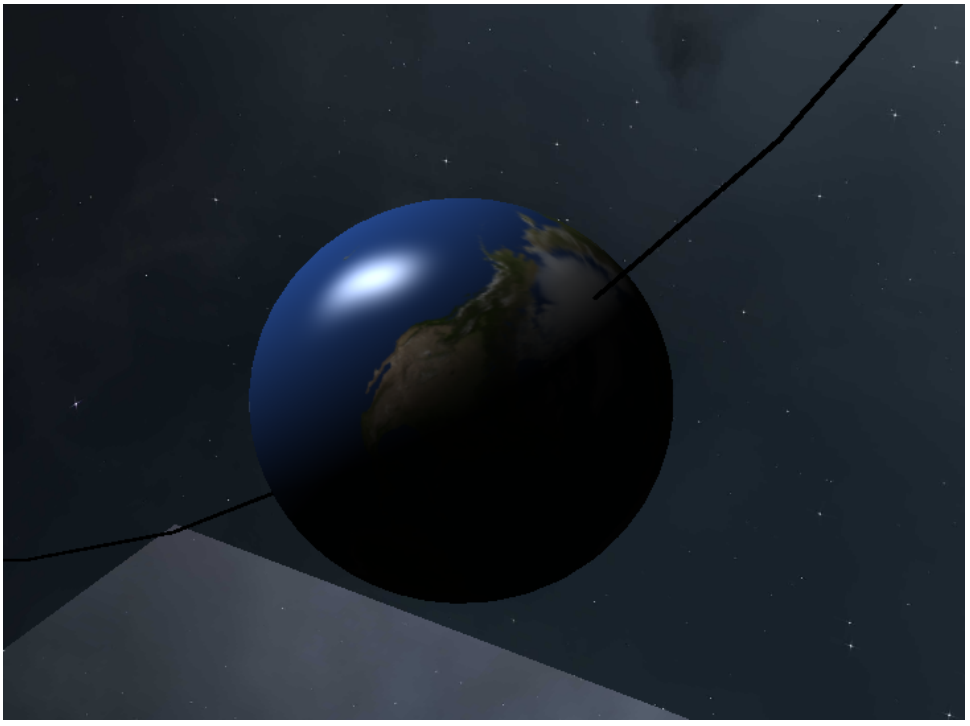
- `diffuseColor`：从漫反射贴图中采样的颜色。
- `specularColor`：从镜面光贴图中采样的颜色。
- `temp`：将漫反射和镜面反射的颜色值相加。
- `FragColor`：将光源颜色与纹理颜色相乘，得到最终的片段颜色。

通过这种方式，片段着色器可以实现点光源本身物体的纹理贴图效果。

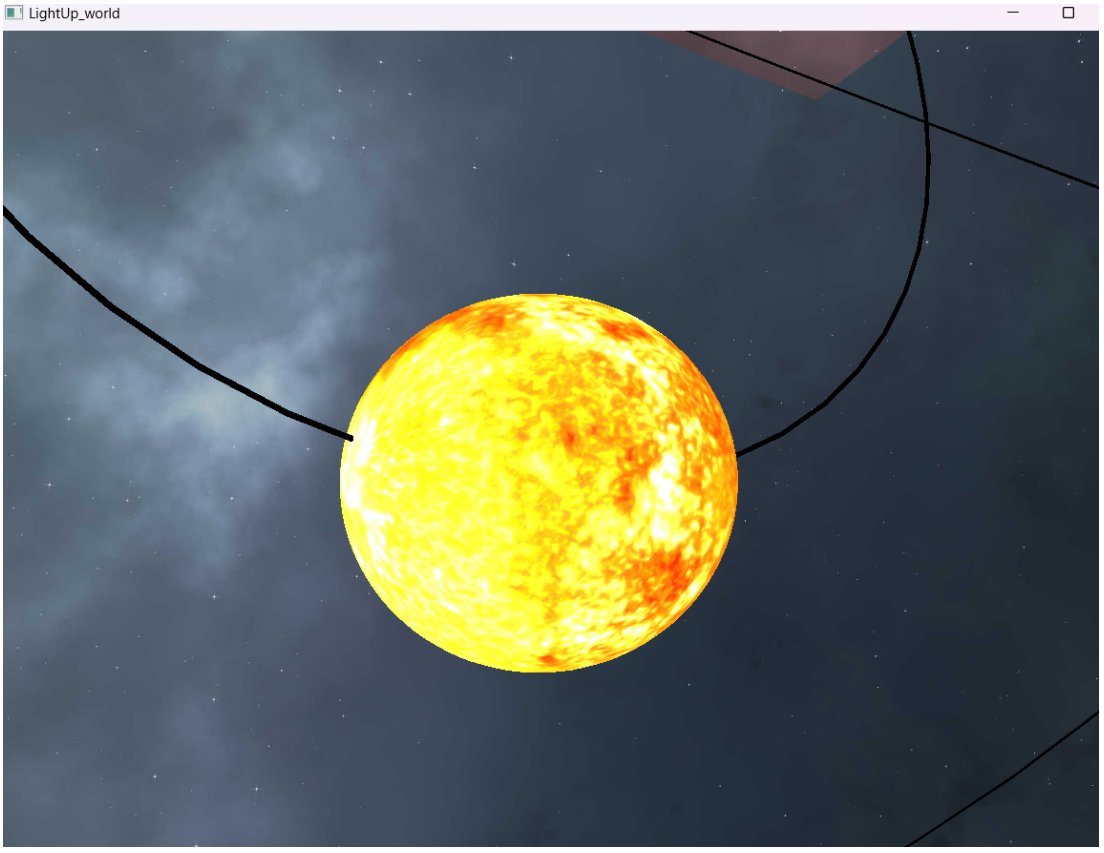
3　实验结果与分析

我们顺利完成了漫反射与镜面反射的实现，并且提前为行星贴好了图，具体的贴图分析我们放至下一次作业报告，以下是实验具体演示：





> 可以看出受点光源影响，我们的实际物体存在亮暗面区别



> 但是我们的点光源物体不受影响，其发光是向各个方向的，不受位置影响