浙江大学

《计算机图形学》
实验报告

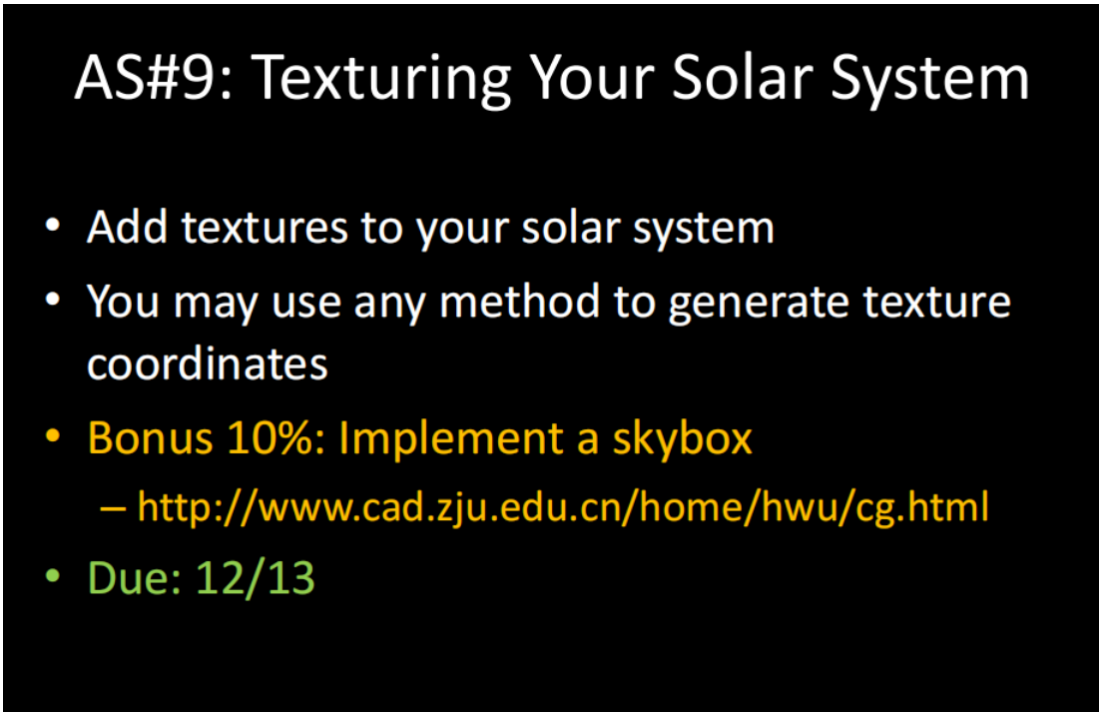
实验名称	：	Textured_Solar_System
姓　　名	：	王晓宇
学　　号	：	3220104364
电子邮箱	：	3220104364@zju.edu.cn
联系电话	：	19550222634
授课教师	：	吴鸿智
助　　教	：	丁华铿

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。

- 为太阳系添加纹理
- 添加天空盒

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。

- 在OpenGL中实现纹理映射（Texture Mapping）是一种常见的技术，用于将图像应用到3D模型的表面，从而增强视觉效果。纹理映射通常涉及以下几个步骤：
    - 加载一张图像作为纹理。可以使用第三方库（如`stb_image.h`）来加载图像数据
    - 顶点着色器需要将纹理坐标传递给片段着色器。
    - 片段着色器需要使用纹理坐标来采样纹理图像，并将其应用到像素颜色上。
    - 在渲染循环中，需要绑定和激活纹理，并将其传递给着色器。
- 天空盒（Skybox）是一种用于模拟远处背景的技术，通常用于创建逼真的3D环境。天空盒是一个立方体，其六个面都贴有纹理，模拟天空、山脉、云层等远景。在OpenGL中实现天空盒通常涉及以下几个步骤：
    - 天空盒的纹理通常由六张图像组成，分别对应立方体的六个面。可以使用`stb_image.h`库来加载这些图像。
    - 天空盒是一个立方体，因此需要定义其顶点数据。通常，天空盒的顶点数据是固定的，不需要法线或纹理坐标。
    - 使用顶点数据创建VAO（顶点数组对象）和VBO（顶点缓冲对象）。
    - 天空盒的着色器相对简单，只需要处理顶点位置和纹理采样。
    - 在渲染循环中，首先渲染其他对象，然后渲染天空盒。为了确保天空盒始终在其他对象的后面，可以关闭深度写入（`glDepthMask(GL_FALSE)`）。

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 纹理加载实现

> 我们介绍一些外部库的使用函数

- `stbi_set_flip_vertically_on_load(true)`：这个函数设置图像在加载时是否需要垂直翻转。由于OpenGL的纹理坐标系和大多数图像文件的坐标系不同，通常需要将图像垂直翻转以正确显示。
- `stbi_load(path, &width, &height, &nrComponents, 0)`：这个函数从指定路径加载图像数据，并返回图像的宽度、高度和颜色通道数（`nrComponents`）。`data`是指向图像数据的指针。
- `glGenTextures(1, &textureID)`：这个函数生成一个或多个纹理对象，并将它们的ID存储在`textureID`中。
- `glBindTexture(GL_TEXTURE_2D, textureID)`：这个函数将生成的纹理对象绑定到当前的纹理目标（`GL_TEXTURE_2D`），以便后续的纹理操作都作用于这个纹理对象。
- `glTexParameteri`：这个函数用于设置纹理参数。

    - `GL_TEXTURE_WRAP_S` 和 `GL_TEXTURE_WRAP_T`：设置纹理在S（水平）和T（垂直）方向上的环绕模式。`GL_REPEAT`表示纹理重复。
    - `GL_TEXTURE_MIN_FILTER`：设置纹理在缩小（**minification**）时的过滤模式。`GL_LINEAR_MIPMAP_LINEAR`表示使用三线性过滤。
    - `GL_TEXTURE_MAG_FILTER`：设置纹理在放大（**magnification**）时的过滤模式。`GL_LINEAR`表示使用线性过滤。
- `glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data)`：这个函数用于生成2D纹理图像。
    - `GL_TEXTURE_2D`：指定纹理目标。
    - `0`：指定**mipmap**级别。
    - `format`：指定纹理的内部格式（如`GL_RGB`或`GL_RGBA`）。
    - `width` 和 `height`：指定纹理的宽度和高度。
    - `0`：指定边界宽度（通常为**0**）。
    - `format`：指定图像数据的格式（如`GL_RGB`或`GL_RGBA`）。
    - `GL_UNSIGNED_BYTE`：指定图像数据的数据类型。
    - `data`：指向图像数据的指针。
- `glGenerateMipmap(GL_TEXTURE_2D)`：这个函数为当前绑定的纹理对象生成**mipmap**。**Mipmap**是一种用于提高纹理渲染质量的技术，通过生成不同分辨率的纹理图像，可以在不同的距离上使用不同的纹理分辨率。

依靠上述函数，我们便可以实现纹理图像的加载，之后只需要传入纹理对应的缓冲对象进行绘制即可。

2.2 代码实现

2.2.1 main.cpp

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

#define GLM_ENABLE_EXPERIMENTAL
#include <geometry/BoxGeometry.h>
#include <geometry/PlaneGeometry.h>
```

```cpp
#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;

int main(int argc, char *argv[])
```

```
{
    Shader::dirName = argv[1];
    glfwInit();
    // 设置主要和次要版本
    const char *glsl_version = "#version 330";

    // 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））
    // glfwWindowHint(GLFW_SAMPLES, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    // 窗口对象
    GLFWwindow *window = glfwCreateWindow(SCREEN_WIDTH, SCREEN_HEIGHT, "LearnOpenGL", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);

    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" << std::endl;
        return -1;
    }

    // ------------------------
    // 创建imgui上下文
    ImGui::CreateContext();
    ImGuiIO &io = ImGui::GetIO();
    (void)io;
    // 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);
108     SphereGeometry Planet_2(0.12f, 50, 50);
109     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");
    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),
207         glm::vec3(1.0f, 0.0f, 0.0f),
208         glm::vec3(0.0f, 1.0f, 0.0f),
```

```
209         glm::vec3(0.7f, 0.5f, 0.5f)};
210     // 轨道颜色
211     glm::vec3 _orbit_color = glm::vec3(0.7f, 0.7f, 0.7f);
212 
213     while (!glfwWindowShouldClose(window))
214     {
215         processInput(window);
216 
217         float currentFrame = glfwGetTime();
218         deltaTime = currentFrame - lastTime;
219         lastTime = currentFrame;
220 
221         glfwSetWindowTitle(window, "LightUp_world");
222 
223         // 渲染指令
224         // ...
225         glClearColor(clear_color.x, clear_color.y, clear_color.z,
clear_color.w);
226         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
227 
228         Simple_Shader.use();
229         glm::mat4 view = camera.GetViewMatrix();
230         glm::mat4 projection = glm::mat4(1.0f);
231         projection = glm::perspective(glm::radians(fov),
(float)SCREEN_WIDTH / (float)SCREEN_HEIGHT, 0.1f, 100.0f);
232 
233         Simple_Shader.setMat4("view", view);
234         Simple_Shader.setMat4("projection", projection);
235         Simple_Shader.setVec3("viewPos", camera.Position);
236 
237         // 设置点光源属性
238         for (unsigned int i = 0; i < 2; i++)
239         {
240             Simple_Shader.setVec3("pointLights[" +
std::to_string(i) + "].position", _planet_postions[i]);
```

```
241             Simple_Shader.setVec3("pointLights[" +
std::to_string(i) + "].ambient", 0.01f, 0.01f, 0.01f);
242             // Simple_Shader.setVec3("pointLights[" +
std::to_string(i) + "].diffuse", _planet_colors[i]);
243             Simple_Shader.setVec3("pointLights[" +
std::to_string(i) + "].diffuse", glm::vec3(1.0f, 1.0f, 1.0f));
244             Simple_Shader.setVec3("pointLights[" +
std::to_string(i) + "].specular", 1.0f, 1.0f, 1.0f);
245 
246             // // 设置衰减
247             Simple_Shader.setFloat("pointLights[" +
std::to_string(i) + "].constant", 1.0f);
248             Simple_Shader.setFloat("pointLights[" +
std::to_string(i) + "].linear", 0.09f);
249             Simple_Shader.setFloat("pointLights[" +
std::to_string(i) + "].quadratic", 0.032f);
250         }
251         glm::mat4 model = glm::mat4(1.0f);
252         // 绘制轨道
253         {
254             // Sun
255             glActiveTexture(GL_TEXTURE0);
256             glBindTexture(GL_TEXTURE_2D, 0);
257             glActiveTexture(GL_TEXTURE1);
258             glBindTexture(GL_TEXTURE_2D, 0);
259 
260             model = glm::mat4(1.0f);
261             Simple_Shader.setMat4("model", model);
262             Simple_Shader.setVec3("lightColor", _orbit_color);
263             Sun_torus.draw();
264             // Planet_1
265             model = glm::mat4(1.0f);
266             model = glm::rotate(model, glm::radians(135.0f),
glm::vec3(1.0f, 0.0f, 0.0f));
267             Simple_Shader.setMat4("model", model);
268             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);

            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);
385             // Light_Shader.setVec3("lightColor",
_planet_colors[0]);
```

```
386             Light_Shader.setInt("diffuse", 0);
387             Light_Shader.setInt("specular", 1);
388 
389             glActiveTexture(GL_TEXTURE0);
390             glBindTexture(GL_TEXTURE_2D, diffuseMap_Sun);
391             glActiveTexture(GL_TEXTURE1);
392             glBindTexture(GL_TEXTURE_2D, specularMap_Sun);
393 
394             glBindVertexArray(Sun_1.VAO);
395             glDrawElements(GL_TRIANGLES, Sun_1.indices.size(),
    GL_UNSIGNED_INT, 0);
396             // Sun_2
397             model = glm::mat4(1.0f);
398             model = glm::rotate(model, glm::radians(0.1f * 150.0f
    * (float)glfwGetTime()), glm::vec3(0.0f, 0.0f, 1.0f));
399             model = glm::translate(model, _planet_postions[1]);
400             Light_Shader.setMat4("model", model);
401             Light_Shader.setVec3("lightColor", _planet_colors[1]);
402             glBindVertexArray(Sun_2.VAO);
403             glDrawElements(GL_TRIANGLES, Sun_2.indices.size(),
    GL_UNSIGNED_INT, 0);
404         }
405 
406         // 绘制天空盒
407         {
408             unsigned int skyboxVAO, skyboxVBO;
409             glGenVertexArrays(1, &skyboxVAO);
410             glGenBuffers(1, &skyboxVBO);
411             glBindVertexArray(skyboxVAO);
412             glBindBuffer(GL_ARRAY_BUFFER, skyboxVBO);
413             glBufferData(GL_ARRAY_BUFFER, sizeof(skyboxVertices),
    &skyboxVertices, GL_STATIC_DRAW);
414             glEnableVertexAttribArray(0);
415             glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 *
    sizeof(float), (void *)0);
416             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();


    return 0;
}

```

```
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     }
482 }
483
484 // 鼠标移动监听
```

```
485 void mouse_callback(GLFWwindow *window, double xpos, double ypos)
486 {
487
488     float xoffset = xpos - lastX;
489     float yoffset = lastY - ypos;
490
491     lastX = xpos;
492     lastY = ypos;
493
494     camera.ProcessMouseMovement(xoffset, yoffset);
495 }
496
497 // 加载纹理贴图
498 unsigned int loadTexture(char const *path)
499 {
500     unsigned int textureID;
501     glGenTextures(1, &textureID);
502
503     // 图像y轴翻转
504     stbi_set_flip_vertically_on_load(true);
505     int width, height, nrComponents;
506     unsigned char *data = stbi_load(path, &width, &height,
&nrComponents, 0);
507     if (data)
508     {
509         GLenum format;
510         if (nrComponents == 1)
511             format = GL_RED;
512         else if (nrComponents == 3)
513             format = GL_RGB;
514         else if (nrComponents == 4)
515             format = GL_RGBA;
516
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 loadTexture函数说明

> 此函数负责纹理图像的加载，以便在渲染时绑定纹理使用

```
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;

        glBindTexture(GL_TEXTURE_2D, textureID);
        glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

```
28
29          stbi_image_free(data);
30      }
31      else
32      {
33          std::cout << "Texture failed to load at path: " << path <<
    std::endl;
34          stbi_image_free(data);
35      }
36
37      return textureID;
38  }
39
```

函数 `loadTexture` 的主要功能是加载图像文件并将其转换为 OpenGL 纹理对象。

1. **生成纹理对象**：

   ```
   1  unsigned int textureID;
   2  glGenTextures(1, &textureID);
   ```

   使用 `glGenTextures`

函数生成一个纹理对象，并将其 `ID` 存储在 `textureID` 变量中。

2. **图像 Y 轴翻转**：

   ```
   1  stbi_set_flip_vertically_on_load(true);
   ```

   使用 `stbi_set_flip_vertically_on_load`函数将图像在加载时进行 Y 轴翻转。这是因为图像的 Y 轴方向通常与 OpenGL 的 Y 轴方向相反。

3. **加载图像数据**：

   使用 `stbi_load`函数加载图像数据，并获取图像的宽度、高度和颜色通道数。

4. **检查图像数据是否加载成功**：

```
if (data)
{
    GLenum format;
    if (nrComponents == 1)
        format = GL_RED;
    else if (nrComponents == 3)
        format = GL_RGB;
    else if (nrComponents == 4)
        format = GL_RGBA;
```

根据图像的颜色通道数，确定图像的格式（单通道、RGB 或 RGBA）。

5. 绑定纹理对象并设置纹理参数：
    - 使用 `glBindTexture` 函数绑定纹理对象。
    - 使用 `glTexImage2D`函数将图像数据传递给 OpenGL。
    - 使用 `glGenerateMipmap` 函数生成多级渐远纹理。
    - 使用 `glTexParameteri` 函数设置纹理参数，如纹理环绕方式和纹理过滤方式。
6. 使用 `stbi_image_free` 函数释放图像数据。
7. 返回纹理对象 ID：

2.2.3 loadCubemap函数介绍

```
unsigned int loadCubemap(vector<std::string> faces)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    int width, height, nrChannels;
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        unsigned char *data = stbi_load(faces[i].c_str(), &width,
&height, &nrChannels, 0);
        if (data)
        {
```

```
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
            stbi_image_free(data);
        }
        else
        {
            std::cout << "Cubemap texture failed to load at path:" << faces[i] << std::endl;
            stbi_image_free(data);
        }
    }
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    ;
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    ;
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    ;
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    ;
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
    ;

    return textureID;
}

```

`loadCubemap` 的函数用于加载立方体贴图纹理。立方体贴图是一种特殊的纹理类型，通常用于环境映射和天空盒效果。函数的参数是一个包含六个面纹理路径的字符串向量 `faces`。

- 首先，函数声明了一个无符号整数 `textureID`，用于存储生成的纹理对象的ID。通过调用 `glGenTextures` 函数生成一个纹理对象，并将其绑定到 `GL_TEXTURE_CUBE_MAP` 目标上：

```
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
```

- 函数声明了三个整数变量 `width`、`height` 和 `nrChannels`，用于存储加载图像的宽度、高度和通道数。在一个循环中，函数遍历 `faces` 向量中的每个纹理路径，并使用 `stbi_load` 函数加载图像数据：
- 如果图像数据成功加载，函数将图像数据传递给 OpenGL，并使用 `glTexImage2D` 函数将其存储在立方体贴图的相应面上。然后，释放图像数据。如果图像加载失败，函数会输出错误信息并释放图像数据。
- 在加载所有六个面之后，函数设置立方体贴图的纹理参数：这些参数设置了纹理的缩小和放大过滤方式为线性过滤，并将纹理的环绕方式设置为 `GL_CLAMP_TO_EDGE`，以确保纹理坐标在边界处被正确处理。
- 最后，函数返回生成的纹理对象的ID：

2.2.4 light_vertex.glsl

```
#version 330 core
layout (location = 0) in vec3 aPos;

// 纹理坐标是3维的
out vec3 TexCoords;
// 不用model转换到世界矩阵
uniform mat4 projection;
uniform mat4 view;
void main()
{
    // 纹理坐标等于位置坐标/
    TexCoords = aPos;
    vec4 pos = projection * view * vec4(aPos, 1.0);
    // z为w，透视除法除后z=(z=w/w)=1，深度为最远///
    gl_Position = pos.xyww;
}

```

> 天空盒的顶点位置通常不需要进行模型变换，因为天空盒是一个巨大的立方体，通常位于场景的远处。顶点着色器只需要将顶点位置转换到裁剪空间，并确保天空盒始终在其他对象的后面。

这个顶点着色器用于渲染天空盒。天空盒是一种用于模拟远处环境的技术，通常用于创建逼真的背景。

- `layout (location = 0) in vec3 aPos;`：顶点位置，使用位置 `0`。
- `out vec3 TexCoords;`：输出的纹理坐标，将传递给片段着色器。
- `uniform mat4 projection;`：投影矩阵，用于将顶点从视图空间变换到裁剪空间。
- `uniform mat4 view;`：视图矩阵，用于将顶点从世界空间变换到视图空间。

1. 在 `main` 函数中，首先将输入的顶点位置 `aPos` 直接赋值给输出变量 `TexCoords`，因为天空盒的纹理坐标与顶点位置相同：

   ```
   TexCoords = aPos;
   ```

2. 然后，计算顶点的最终位置，并将其赋值给 `gl_Position`：

   ```
   vec4 pos = projection * view * vec4(aPos, 1.0);
   gl_Position = pos.xyww;
   ```

3. 这里，我们将顶点位置从模型空间依次变换到视图空间和裁剪空间。特别注意的是，`gl_Position` 的 `z` 分量被设置为 `w` 分量，这样在透视除法后，`z` 分量将始终为 `1.0`，从而确保天空盒的深度值为最远。这种处理方式可以避免天空盒被场景中的其他物体遮挡。

顶点着色器将顶点位置和纹理坐标传递给片段着色器，以便在后续的渲染过程中使用。这个顶点着色器的设计确保了天空盒始终在背景中，并且不会受到场景中其他物体的影响。

2.2.5 light_fragment.glsl

```glsl
#version 330 core
out vec4 FragColor;

// 纹理坐标是3维的
in vec3 TexCoords;// 纹理坐标

// 天空盒纹理采样
uniform samplerCube skybox;

void main(){
    FragColor = texture(skybox, TexCoords);
}
```

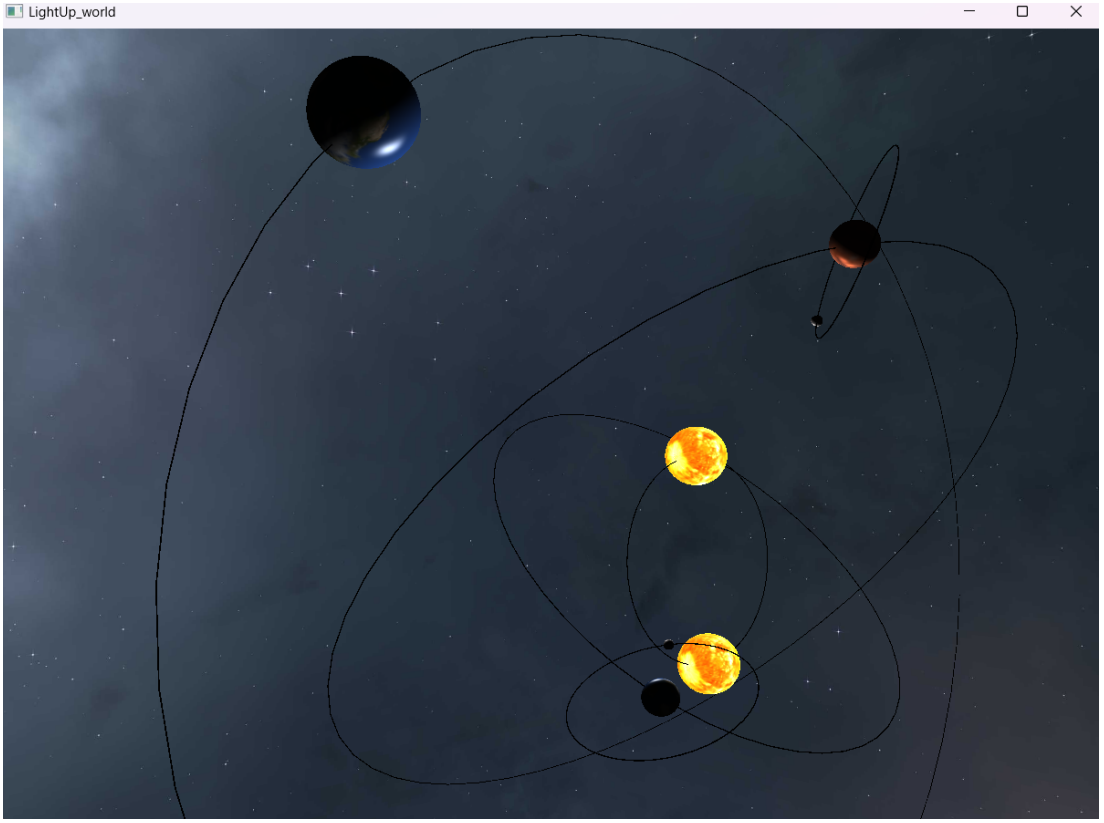
> 普通的片段着色器通常会处理光照计算（如漫反射、镜面反射等），并使用纹理坐标从2D纹理中采样颜色。最终输出像素的颜色。
>
> 天空盒的片段着色器相对简单，主要任务是从立方体贴图（Cubemap）中采样颜色，并输出像素的颜色。由于天空盒不需要进行光照计算，片段着色器的主要工作是从传入的纹理坐标（即顶点位置）中采样颜色。

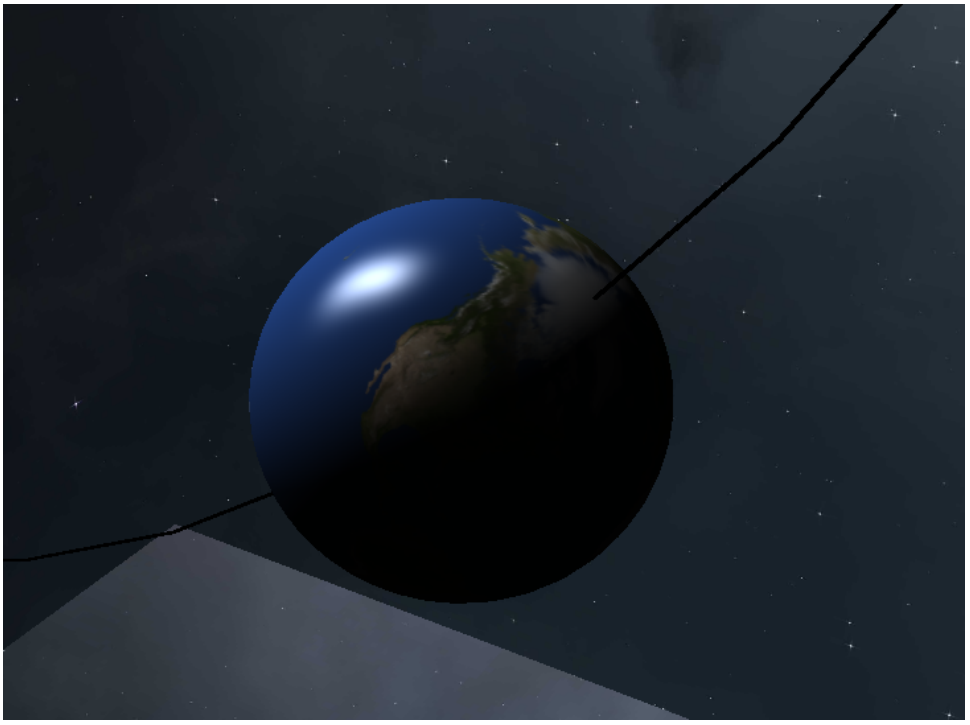
- 定义输出变量 `FragColor`，用于存储最终的片段颜色
- 定义输入变量 `TexCoords`，用于接收从顶点着色器传递过来的三维纹理坐标
- 定义一个 **uniform** 变量 `skybox`，它是一个立方体贴图采样器，用于采样天空盒纹理
- 在 `main` 函数中，我们使用 `texture` 函数从立方体贴图中采样颜色，并将结果赋值给 `FragColor`，`texture` 函数根据传入的三维纹理坐标 `TexCoords` 从 `skybox` 立方体贴图中获取相应的颜色值。这个颜色值就是最终的片段颜色 `FragColor`。

通过这些操作，片段着色器将根据传入的纹理坐标从天空盒纹理中采样颜色，并将其作为片段的最终颜色输出。这个片段着色器的设计确保了天空盒的每个片段都能正确地显示对应的环境颜色，从而实现逼真的背景效果。

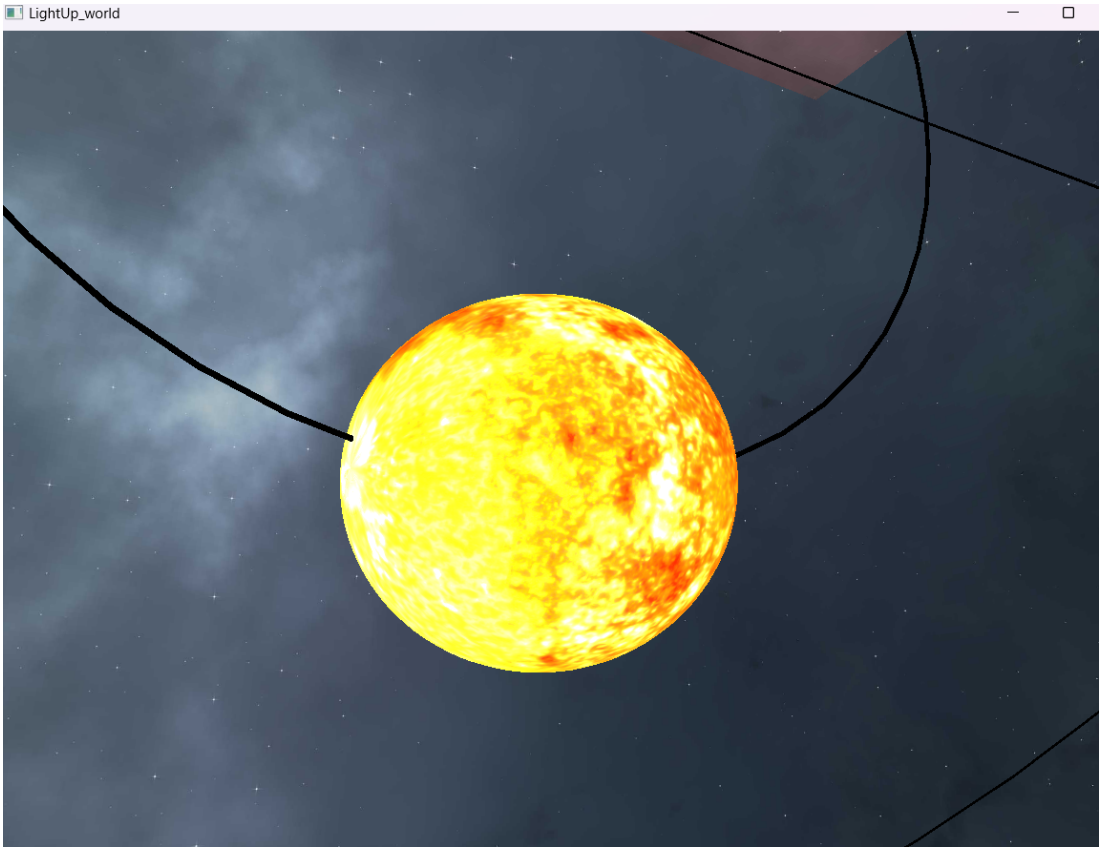
3　实验结果与分析

我们顺利完成了漫反射与镜面反射的实现，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