**计算机图形学课程报告**

计算机科学与技术学院

**班 级：**

**学 号：**

**姓 名：**

**指导教师：**

**完成日期：**

**目录**

[1 简答 3](#_Toc153281111)

[2 论述 5](#_Toc153281112)

[2.1 实验内容 5](#_Toc153281113)

[2.2 实验方法和过程 5](#_Toc153281114)

[2.3 实验结果 9](#_Toc153281115)

[2.4 心得体会 10](#_Toc153281116)

[2.5 源代码 11](#_Toc153281117)

# 简答

（1）你选修计算机图形学课程，想得到的是什么知识？现在课程结束，对于所得的知识是否满意？如果不满意，你准备如何寻找自己需要的知识。

我希望得到一些能在游戏开发中使用的知识，使自己能渲染出自己想要效果的画面（比如更加写实，或者偏卡通画风等）。然后也希望了解图形渲染的流程，希望学完课后能独立完成游戏引擎的图形渲染部分。同时也希望通过学习该课程了解到一些游戏引擎方面的前沿知识。课程结束，我对所得到知识较为满意，我的期待大部分都实现了。我打算继续阅读图形学方面的论文，同时和在相关方面有研究的同学进行讨论来寻求自己所需要的知识。

（2）你对计算机图形学课程中的哪一个部分的内容最感兴趣，请叙述一下，并谈谈你现在的认识。

我对图形学中的光照渲染方式较为感兴趣。经过学习后我了解到光照渲染大致有简单光照模型、整体光照模型、光线追踪算法、辐射度算法以及BRDF模型这几种算法。

其中简单光照模型不考虑环境镜面反射光以及环境规则透射光，实现后真实度有限。而整体光照模型则会考虑这些。

光线追踪算法本质就是模拟人的视线的投射，并在投射到的点上进一步进行反射、折射、阴影等计算，然后结合局部光照模型得到最后该点的颜色。其中需要大量的递归操作，由于Shader中不支持递归的写法，所以需要手动写栈。整体看来，该方法计算量大但真实感强，而且其实可以用包围盒技术来加速其运算。

辐射度算法是基于物理学的一种算法，其认为光是一种辐射能，在封闭环境中经过多次反射和投射后，各表面上的该能量将会达到平衡。而表面上该能量的多少即反映了亮度的强弱。该算法主要的步骤为表面分割、形状因子计算、方程求解与绘制。表面分割通常有均匀网格化与自适应网格化两种方法，形状因子的计算通常采用数值方法而非解析方法求解，而数值方法主要有半立方体与光线采样两种方法。

BRDF模型对于不同材质反射光的性质有了更具体的刻画，这主要体现在他提供了一个基于光线波长、入射角、反射角、表面空间位置的反射率函数。

（3）你对计算机图形学课程的教学内容和教学方法有什么看法和建议。

我认为图形学的教学内容覆盖范围较广，其图形学基础的部分基本都有所涉及，像图形渲染管线、坐标变换、光照模型等重要内容的讲解也很细致。

但我觉得其实图形学可以再多涉及一点相关领域的前沿内容以及与人工智能等热门学科的交叉内容，以顺应时代并激发大家兴趣。

而图形学这门课的教学方法主要以理论授课与课后实践作业为主。我觉得这种形式虽然传统但有效。

但是我希望我们的作业的对我们所要求的修改部分能更多一点，而不是类似于完型填空的形式。这样我们能有更大的发挥空间，能锻炼我们的代码能力并使我们对课上所学知识的掌握更为深刻。同时我觉得一些作业可以结合例如unity等游戏引擎，告诉我们在unity中如何应用图形学的相关知识。

# 论述

* 1. 实验内容

利用OpenGL框架，设计一个日地月运动模型动画，要求如下：

（1）运动关系正确，相对速度合理，且地球绕太阳，月亮绕地球的轨道不能在一个平面内。

（2）地球绕太阳，月亮绕地球可以使用简单圆或者椭圆轨道。

（3）对球体纹理的处理，至少地球应该有纹理贴图。

（4）增加光照处理，光源设在太阳上面。

（5）为了提高太阳的显示效果 ，可以在侧后增加一个专门照射太阳的灯。

* 1. 实验方法和过程
     1. 构造球面顶点

太阳、地球、月亮都是球体，所以他们可以共用同一球体顶点数据数组，数组需要包含顶点的xyz坐标以及uv坐标（纹理坐标）。生成球体对应的顶点数据数组较为简单，设置x方向和y方向的顶点数，然后进行两层for循环，遍历每一个顶点，顶点的经度可由x计算，顶点的纬度可由y计算。然后根据经纬度计算得出所需数据。

把顶点的经纬度由极坐标变换公式映射到xyz空间即可求得顶点的坐标，具体公式如公式1所示，其中θ为纬度，Φ为经度。

公式 1 极坐标变换公式

顶点的uv坐标可以直接通过把θ与Φ映射到（0，1）的值域求得。

接着要构造球体顶点的下标数组，这里我们以三角形为基本面元进行构建。每次在一个矩形中构建，遍历所有矩形的左下角顶点，每次遍历把矩形分为一个左上三角和一个右下三角，分别把这两个三角对应的三个顶点的下标放到当前下标数组的队尾。

* + 1. 配置球体与背景的VAO

生成球体和背景的VBO、VAO以及EBO句柄，把球体与背景的顶点数据数组和顶点下标数据都进行相应的绑定，同时设置顶点属性指针。

顶点属性指针要指出location=0处的三个float数据为顶点坐标数据，location=1处的两个float数据为顶点纹理坐标数据。

其中背景的顶点数据我们设置为左上角为（-1，1）同时右下角为（1，-1）的一个矩形。这样，之后进行坐标变换时，就不需要对背景的顶点再进行MVP变换。

* + 1. 编写Shader工具类

为了提高代码的封装性与可复用性，这里我们把Shader程序的读取、编译、链接、报错信息输出、使用以及公共变量的设置等功能全都封装为一个Shader工具类。

* + 1. 编写顶点着色器

顶点着色器要接收顶点的局部坐标以及纹理坐标这两个数据。

首先考虑生成顶点的法向量。由于要应用光照的物体皆为球体，所以可以直接根据球体的局部坐标生成其法向量，再将法向量左乘Model矩阵的逆的装置，即可得到变换到世界坐标系后的法向量并进行输出。

把顶点的局部坐标进行MVP矩阵变换，即可得到顶点在NDC空间中的坐标，将其传递给gl\_Position进行进一步处理。

为了计算光照，顶点着色器的顶点也需要把其对应的世界坐标系中的坐标进行传递。

同时为了应用纹理，顶点着色器也需要把纹理坐标原封不动地进行输出。

* + 1. 编写片段着色器

这里需要分为普通物体和背景两种情况进行着色。

当当前渲染物体不为背景时，按照Phong光照模型的公式，进行对于光照的计算，得到当前要输出的颜色，具体公式如公式2所示。其中顶点的世界坐标、法向量都从之前的阶段传递，顶点的颜色为共享的Texture中的顶点纹理坐标所对应的颜色，而光源坐标、摄像机坐标都由C++程序进行设置并共享给该阶段。

公式 2 Phong光照模型

当当前渲染物体为背景时，不需要光照模型，直接获取Texture中该顶点纹理坐标对应的颜色并输出即可。

* + 1. 生成View矩阵

View矩阵可以直接使用glm库的lookAt函数进行生成，通过设置Uniform变量的方式共享给GPU中的Shader程序。

* + 1. 生成Projection矩阵

这里采用透视成像，可以直接使用glm自带的perspective函数来生成矩阵，并同样通过Uniform的方式共享给Shader程序。

* + 1. 生成太阳的Model矩阵

太阳一直位于世界坐标的原点处，所以其Model矩阵设置为单位矩阵即可。这样生成的太阳会是一个处于原点，半径为1的球体。

* + 1. 生成地球的Model矩阵

地球围绕太阳进行公转，同时其自身也在自传，其自转并非竖直自转而是绕着有一定角度倾斜的地轴进行自转。先通过坐标变换求出地球应该公转到的世界坐标，再由该世界坐标生成地球的平移矩阵。应用平移矩阵前，把地球进行一定程度的缩放，绕y轴旋转到当前的自转角度，同时根据地轴的倾向程度将地球绕z轴旋转一定角度，全部完成后再应用平移矩阵。具体公式如公式3所示。

公式 3 地球Model矩阵计算公式

其中 分别为地球坐标平移矩阵、地轴旋转矩阵、缩放矩阵与自转旋转矩阵。

* + 1. 生成月球的Model矩阵

先将月球进行缩放，再向x轴平移一个其绕地球公转轨道的半径的距离，再将其绕z轴旋转23.5°，使其和地球公转的轨道不在一个平面上，同时将其公转轴也进行同样的旋转，再绕其旋转轴旋转其当前公转应转过的角度，最后将其平移到地球当前的世界坐标处。具体公式如公式4所示，其中

公式 4 月球Model矩阵计算公式

其中 分别为地球坐标平移矩阵、公转旋转矩阵、公转平面旋转矩阵、公转半径平移矩阵与缩放矩阵。

* + 1. 绑定光照相关的全局变量

和光照相关的全局变量主要有：光源的位置，此处和太阳的位置一致，都一直保持在原点；光源的颜色，这里直接设置为（1，1，1）；摄像机的位置。

把这些变量都通过设置Uniform变量的方式共享给Shader程序进行光照渲染的计算。

* + 1. 生成背景的变换矩阵

背景不需要做任何变换，所以其Model、View、Projection矩阵都为单位矩阵。

* + 1. 读取图片并绑定纹理

使用stb\_image.h进行图片数据的读取，再给每个物体都生成其对应的texture句柄并绑定其图片数据。当渲染某一物体前，需将0号texture缓存位绑定为该物体对应的texture句柄。

* + 1. 编写镜头控制逻辑

由于要通过鼠标控制镜头的朝向，所以要把镜头改变函数作为鼠标移动后的回调函数进行绑定。镜头改变函数主要检测当前鼠标的偏移，根据该偏移改变yaw与pitch，并根据yaw和pitch计算出当前摄像机的正方向、上方向与右方向，由于我们规定上方向向量一定在竖直平面内，所以计算右方向向量时可直接用正方向向量叉乘竖直向量。具体计算公式如公式5所示

公式 5 摄像机正方向、上方向、右方向计算公式

* + 1. 编写移动控制逻辑

由于要用键盘控制摄像机的移动，所以每次循环时都要判断WASD按键是否被按下，被按下则对摄像机的位置按照其正、上、右方向进行相应的更改。

* 1. 实验结果

具体程序可见github仓库：<https://github.com/FlashLizard/SolarSysModel.git>

由图1与图2可见首先光照模型运行正常，可以看见清晰的环境光、漫反射光与较为清晰的镜面反射光（由于是星球，所以镜面反射不宜设置得过于明显）。 其次可见运动模型正常，地球地轴与竖直方向有一定偏移，地球公转平面与月球公转平面不在同一平面，地球有自转。

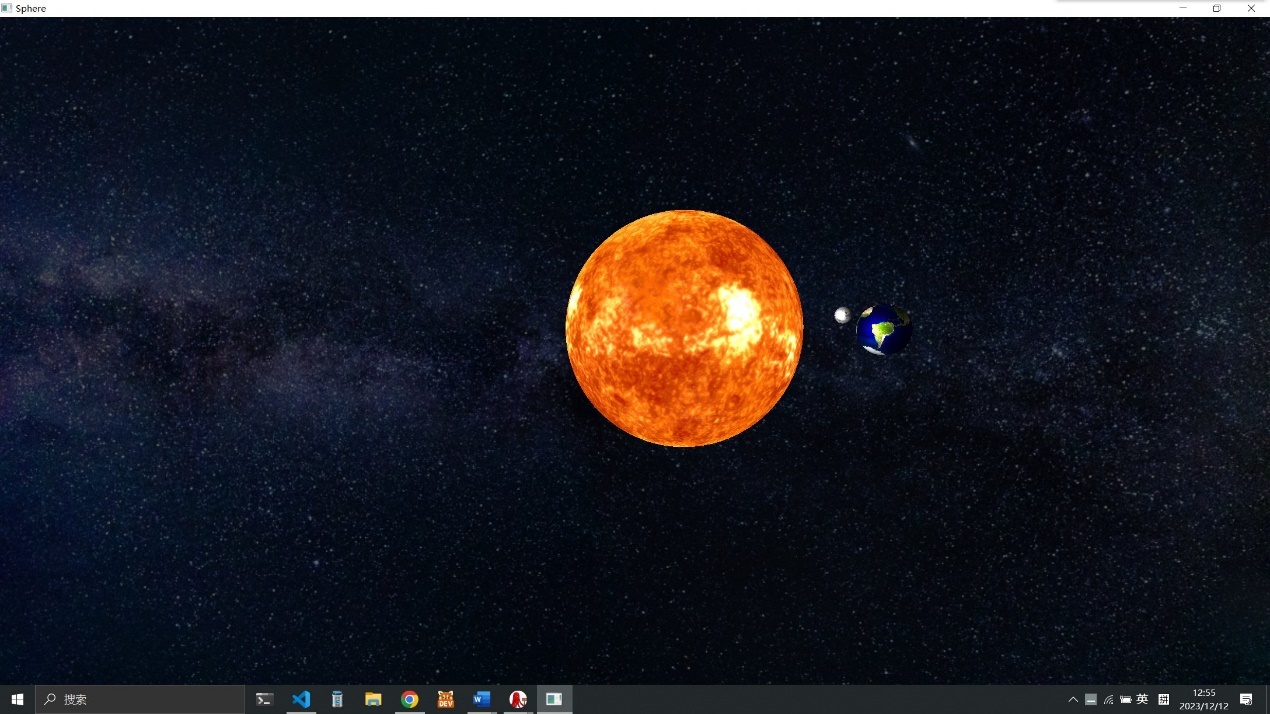


图 1 运行结果1

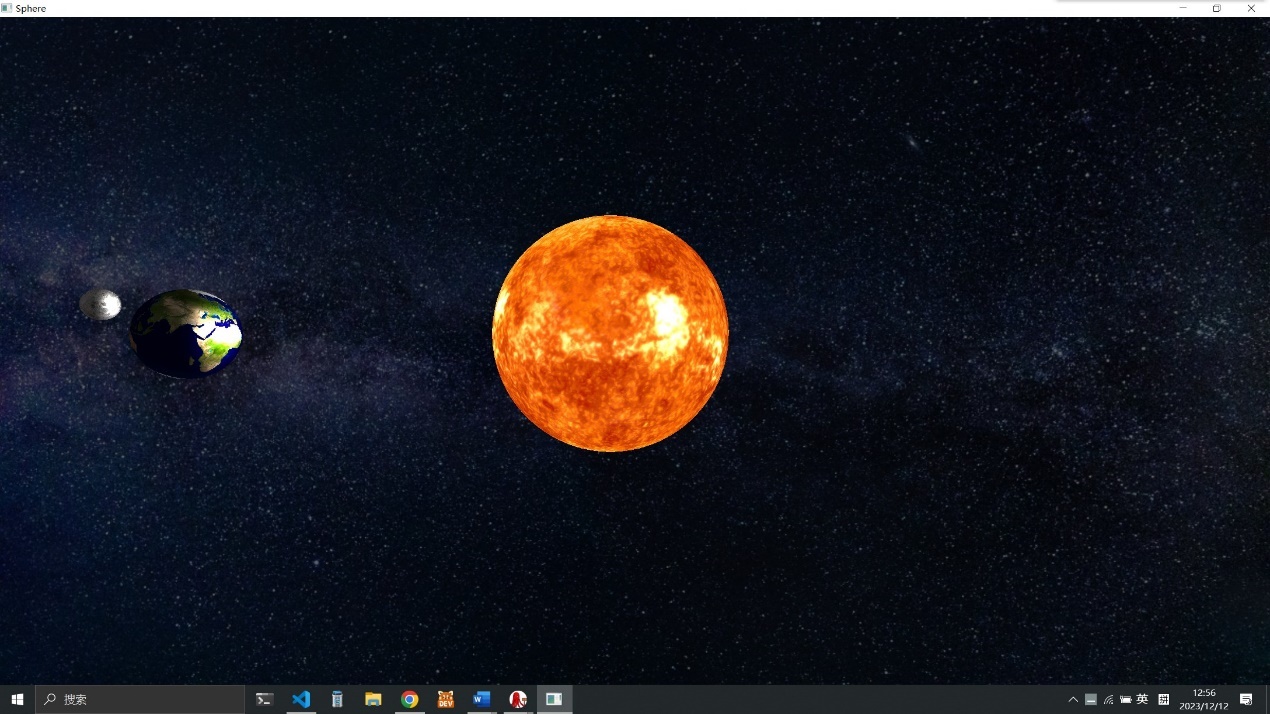


图 2 运行结果2

由图3与图4可见镜头控制与移动控制正常，这两张图中均改变了视角与位置来观察日地月模型。

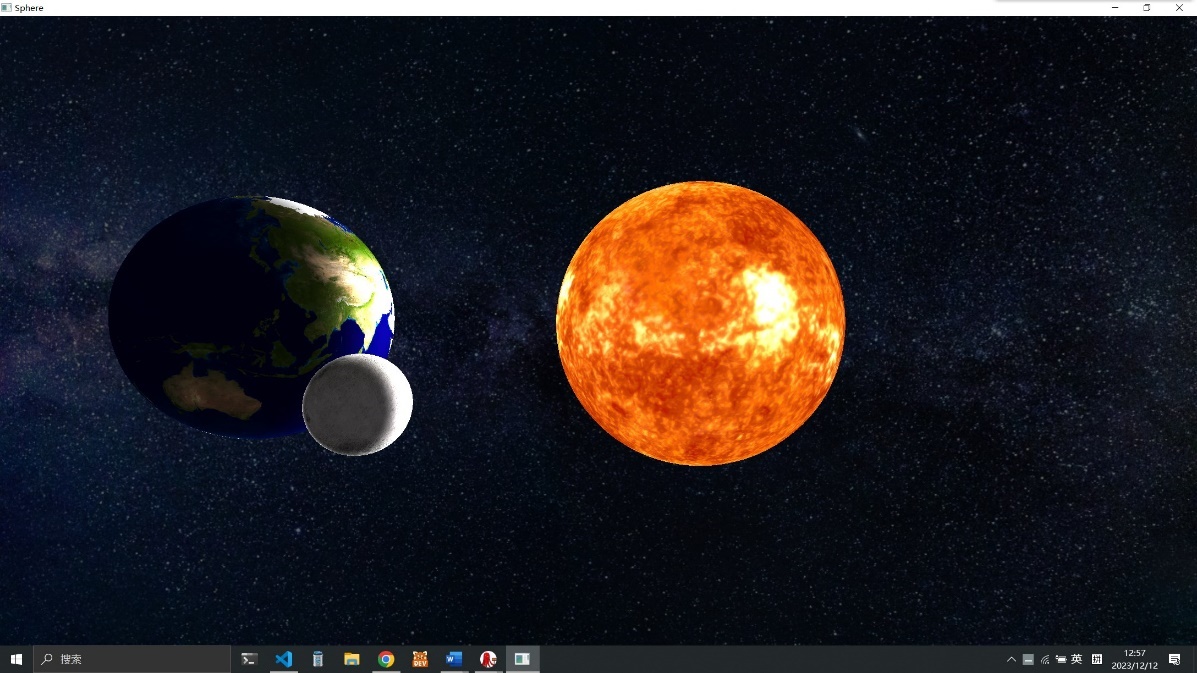


图 3 运行结果3

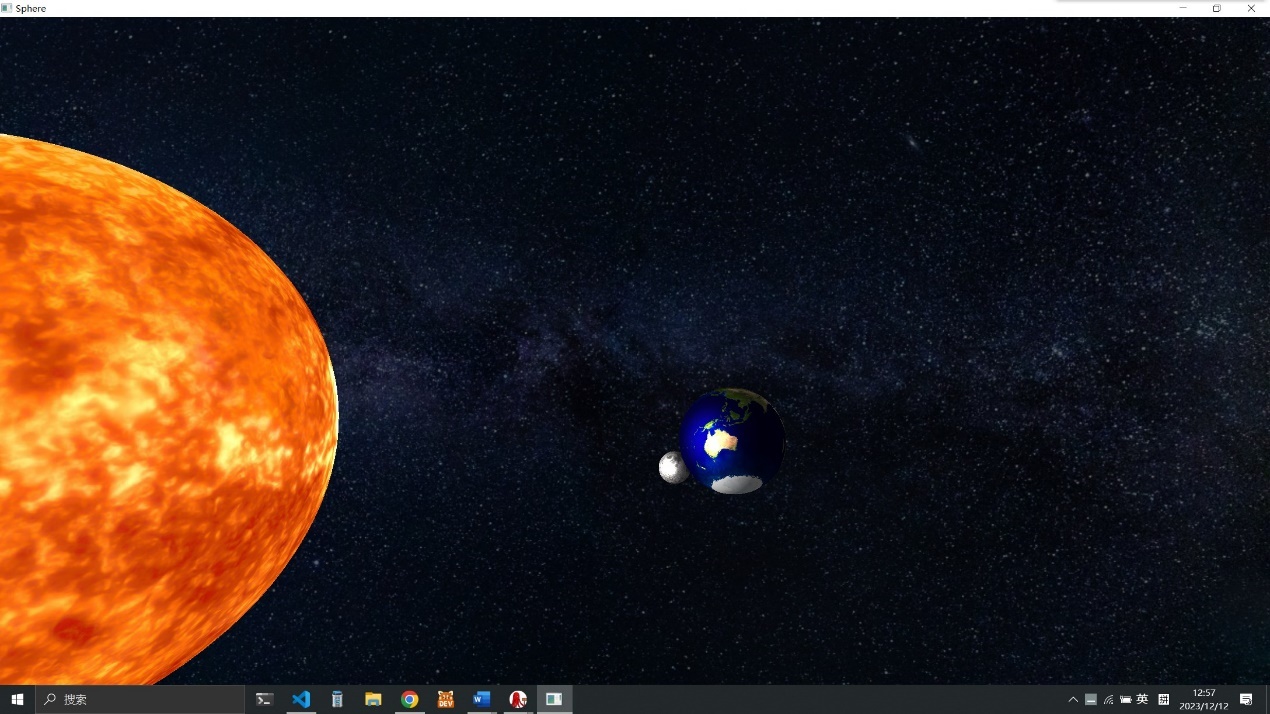


图 4 运行结果4

* 1. 心得体会

本次实验让我收获颇多。

首先，在把三个球体与背景渲染到窗口的过程中，我对于如何使用opengl控制图形渲染管线有了更深刻的理解，对顶点数据进行绑定的流程更加熟练，对于VBO、VAO以及EBO之间的关系与区别也了解得更加深入。

其次，在进行纹理贴图映射的过程中，我了解了opengl窗口坐标系与一般坐标系（比如jpg图片）的异同，学会了如何在球体上进行纹理映射，同时更加深刻地意识到mipmap对于反走样的重要性。

然后，在进行光照渲染时，我进一步掌握了GLSL语言的语法，了解了该如何正确编写顶点着色器文件与片段着色器文件、如何将CPU的数据共享到GPU中进行使用，同时也深入理解了Phong光照模型的理论原理与实现方式。

之后，在进行坐标变换的实践时，我更加清楚地认识了Model矩阵、View矩阵、Projection矩阵与MVP矩阵之间的关系与使用场景，理解了局部坐标系、世界坐标系、摄像机坐标系、NDC空间、屏幕坐标系之间地联系与转换方式。

最后，通过鼠标与键盘控制方面的应用，我理解到了回调函数的好处，对于回调函数的使用也更加得心应手。

总而言之，本次日地月模型的图形学大作业让我受益匪浅，锻炼了我的代码能力、学习能力、逻辑能力、线性代数能力与空间想象能力，相信在以后的学习与生活中这些都将成为我进一步发展的动力。

* 1. 源代码

|  |
| --- |
| **程序1：main.cpp** |
| **#include <glad/glad.h>**  **#include <GLFW/glfw3.h>**  **#include "shader.h"**  **#include <stb/stb\_image.h>**  **#include <iostream>**  **#include <vector>**  **#include <chrono>**  **#include <glm/glm.hpp>**  **#include <glm/gtc/matrix\_transform.hpp>**  **#include <glm/gtc/type\_ptr.hpp>**  **#include <cmath>**  **struct DotLight**  **{**  **glm::vec3 pos;**  **glm::vec3 color;**  **};**  **DotLight sunLight = {**  **glm::vec3(0, 0, 0),**  **glm::vec3(1, 1, 1)};**  **class Image**  **{**  **public:**  **unsigned char \*data;**  **int width, height, nrChannels;**  **Image(std::string path)**  **{**  **data = stbi\_load(path.c\_str(), &width, &height, &nrChannels, 0);**  **}**  **void Print()**  **{**  **printf("Tex %d,%d,%d\ndata:", width, height, nrChannels);**  **for (int i = 0; i < std::min(width \* height, 50); i++)**  **{**  **printf("%d,", data[i]);**  **}**  **if (width \* height >= 50)**  **{**  **printf("...\n");**  **}**  **else**  **{**  **printf("\n");**  **}**  **}**  **~Image()**  **{**  **stbi\_image\_free(data);**  **}**  **};**  **// 窗口大小参数**  **const unsigned int SCR\_WIDTH = 800;**  **const unsigned int SCR\_HEIGHT = 600;**  **int fullWidth;**  **int fullHeight;**  **int fullWin = 0;**  **float aspect = (float)4.0 / (float)3.0;**  **// 旋转参数**  **static GLfloat earthRot = 20.0f;**  **static GLfloat moonRot = 20.0f;**  **static GLfloat selfRot = 0.0f;**  **// 句柄参数**  **GLuint ballVAO; // == VAO句柄**  **GLuint ballVBO; // == VBO句柄**  **GLuint ballEBO; //==EBO句柄**  **int ballSize;**  **// background**  **GLuint backVAO;**  **GLuint backVBO;**  **GLuint backEBO;**  **int backSize;**  **// 球的数据参数**  **const int X\_SEGMENTS = 50;**  **const int Y\_SEGMENTS = 50;**  **const GLfloat PI = 3.14159265358979323846f;**  **// 贴图**  **Image \*earthImg;**  **Image \*moonImg;**  **Image \*sunImg;**  **Image \*backImg;**  **unsigned int earthTex;**  **unsigned int sunTex;**  **unsigned int moonTex;**  **unsigned int backTex;**  **// 相机控制相关**  **bool firstMouse = true;**  **float lastX, lastY;**  **float deltaTime = 0;**  **float speed = 0.3;**  **float yaw = 90, pitch = 0;**  **glm::vec3 viewPos(0.0f, 0.0f, -5.0f);**  **glm::vec3 cameraFront(0, 0, 1.0);**  **glm::vec3 cameraRight(-1, 0, 0);**  **glm::vec3 cameraUp(0, 1, 0);**  **int pause = 0;**  **void genSphere(float radius, int xSegment, int ySegment, bool uv, std::vector<float> &sphereVertices, std::vector<int> &sphereIndices)**  **{**  **// 进行球体顶点和三角面片的计算**  **// 生成球的顶点**  **for (int y = 0; y < ySegment; y++)**  **{**  **for (int x = 0; x < xSegment; x++)**  **{**  **float xi = (float)x / (float)(xSegment - 1);**  **float yi = (float)y / (float)(ySegment - 1);**  **float theta = yi \* PI;**  **float phi = xi \* 2 \* PI;**  **float xPos = radius \* std::sin(theta) \* std::cos(phi);**  **float yPos = radius \* std::cos(theta);**  **float zPos = radius \* std::sin(theta) \* std::sin(phi);**  **sphereVertices.push\_back(xPos);**  **sphereVertices.push\_back(yPos);**  **sphereVertices.push\_back(zPos);**  **if (uv)**  **{**  **float u = xi;**  **float v = 1.0 - yi;**  **sphereVertices.push\_back(u);**  **sphereVertices.push\_back(v);**  **}**  **}**  **}**  **// 生成球的顶点**  **for (int i = 0; i < ySegment; i++)**  **{**  **for (int j = 0; j < xSegment; j++)**  **{**  **sphereIndices.push\_back(i \* (xSegment) + j);**  **sphereIndices.push\_back((i + 1) \* (xSegment) + j);**  **sphereIndices.push\_back((i + 1) \* (xSegment) + j + 1);**  **sphereIndices.push\_back(i \* (xSegment) + j);**  **sphereIndices.push\_back((i + 1) \* (xSegment) + j + 1);**  **sphereIndices.push\_back(i \* (xSegment) + j + 1);**  **}**  **}**  **}**  **void genTex(unsigned int \*id, Image \*img)**  **{**  **glGenTextures(1, id);**  **glBindTexture(GL\_TEXTURE\_2D, \*id);**  **// 为当前绑定的纹理对象设置环绕、过滤方式**  **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);**  **glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);**  **glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, img->width, img->height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE,**  **img->data);**  **glGenerateMipmap(GL\_TEXTURE\_2D);**  **}**  **Shader initial(void)**  **{**  **stbi\_set\_flip\_vertically\_on\_load(true);**  **// tell stb\_image.h to flip loaded texture's on the y-axis.**  **// 图片**  **earthImg = new Image("res/earth.jpg");**  **sunImg = new Image("res/sun.jpg");**  **moonImg = new Image("res/moon.jpg");**  **backImg = new Image("res/background.jpg");**  **// backImg->Print();**  **// 图片**  **genTex(&earthTex, earthImg);**  **genTex(&sunTex, sunImg);**  **genTex(&moonTex, moonImg);**  **genTex(&backTex, backImg);**  **// 球vao设置**  **std::vector<float> sphereVertices;**  **std::vector<int> sphereIndices;**  **genSphere(1.0, X\_SEGMENTS, Y\_SEGMENTS, true, sphereVertices, sphereIndices);**  **ballSize = sphereIndices.size();**  **glGenVertexArrays(1, &ballVAO);**  **glGenBuffers(1, &ballVBO);**  **// 生成并绑定球体的VAO和VBO**  **glBindVertexArray(ballVAO);**  **glBindBuffer(GL\_ARRAY\_BUFFER, ballVBO);**  **// 将顶点数据绑定至当前默认的缓冲中**  **glBufferData(GL\_ARRAY\_BUFFER, sphereVertices.size() \* sizeof(float), &sphereVertices[0], GL\_STATIC\_DRAW);**  **glGenBuffers(1, &ballEBO);**  **glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, ballEBO);**  **glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sphereIndices.size() \* sizeof(int), &sphereIndices[0], GL\_STATIC\_DRAW);**  **// 设置顶点属性指针**  **glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 5 \* sizeof(float), (void \*)0);**  **glVertexAttribPointer(1, 2, GL\_FLOAT, GL\_FALSE, 5 \* sizeof(float), (void \*)(3 \* sizeof(float)));**  **glEnableVertexAttribArray(0);**  **glEnableVertexAttribArray(1);**  **// 解绑VAO和VBO**  **glBindBuffer(GL\_ARRAY\_BUFFER, 0);**  **glBindVertexArray(0);**  **// 背景vao设置**  **float bDeep = 0.99;**  **std::vector<float> backVertices = {**  **-1, 1, bDeep, 0, 1,**  **1, 1, bDeep, 1, 1,**  **-1, -1, bDeep, 0, 0,**  **1, -1, bDeep, 1, 0};**  **std::vector<int> backIndices = {**  **0, 2, 1,**  **3, 1, 2};**  **backSize = backIndices.size();**  **glGenVertexArrays(1, &backVAO);**  **glGenBuffers(1, &backVBO);**  **glBindVertexArray(backVAO);**  **glBindBuffer(GL\_ARRAY\_BUFFER, backVBO);**  **// 将顶点数据绑定至当前默认的缓冲中**  **glBufferData(GL\_ARRAY\_BUFFER, backVertices.size() \* sizeof(float), &backVertices[0], GL\_STATIC\_DRAW);**  **glGenBuffers(1, &backEBO);**  **glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, backEBO);**  **glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, backIndices.size() \* sizeof(int), &backIndices[0], GL\_STATIC\_DRAW);**  **// 设置顶点属性指针**  **glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 5 \* sizeof(float), (void \*)0);**  **glVertexAttribPointer(1, 2, GL\_FLOAT, GL\_FALSE, 5 \* sizeof(float), (void \*)(3 \* sizeof(float)));**  **glEnableVertexAttribArray(0);**  **glEnableVertexAttribArray(1);**  **// 解绑VAO和VBO**  **glBindBuffer(GL\_ARRAY\_BUFFER, 0);**  **glBindVertexArray(0);**  **// 顶点着色器和片段着色器源码**  **const char \*vertexShader = "shader/shader.vs";**  **const char \*fragmentShader = "shader/shader.fs";**  **// 生成并编译着色器**  **Shader shaderProgram(vertexShader, fragmentShader);**  **shaderProgram.use();**  **// 设定点线面的属性**  **glPointSize(15); // 设置点的大小**  **glLineWidth(5); // 设置线宽**  **// 启动剔除操作**  **// glEnable(GL\_CULL\_FACE);**  **// glCullFace(GL\_BACK);**  **// 开启深度测试**  **glEnable(GL\_DEPTH\_TEST);**  **return shaderProgram;**  **}**  **void Draw(Shader shaderProgram)**  **{**  **// 改变球体角度**  **if (!(pause & 2))**  **{**  **earthRot += (float)0.004f;**  **selfRot += (float)0.3f;**  **moonRot += (float)0.1f;**  **}**  **// 清空颜色缓冲和深度缓冲区**  **glClearColor(1.0f, 1.0f, 1.0f, 1.0f);**  **glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);**  **glm::mat4 view(1.0f);**  **glm::mat4 one(1.0f);**  **view = glm::lookAt(viewPos, viewPos + cameraFront, cameraUp);**  **glm::mat4 projection = glm::perspective(glm::radians(60.0f), aspect, 0.1f, 500.0f);**  **shaderProgram.setMatrix4fv("view", glm::value\_ptr(view));**  **shaderProgram.setMatrix4fv("projection", glm::value\_ptr(projection));**  **shaderProgram.setInt("ourTexture", 0);**  **shaderProgram.setVec3("viewPos", viewPos);**  **// 太阳光源**  **shaderProgram.setVec3("lightPos", sunLight.pos);**  **shaderProgram.setVec3("lightColor", sunLight.color);**  **// 贴图**  **glBindTexture(GL\_TEXTURE\_2D, sunTex);**  **// 当前不是背景**  **shaderProgram.setInt("background", 0);**  **// 当前是太阳**  **shaderProgram.setBool("sun", true);**  **// 绘制sun**  **shaderProgram.setMatrix4fv("model", glm::value\_ptr(one));**  **glBindVertexArray(ballVAO); // 绑定VAO**  **glDrawElements(GL\_TRIANGLES, ballSize, GL\_UNSIGNED\_INT, 0);**  **shaderProgram.setBool("sun", false);**  **// 绘制earth**  **glm::mat4 earthTrans;**  **earthTrans = glm::rotate(one, glm::radians(earthRot), glm::vec3(0.0, 1.0, 0.0)) \* glm::translate(one, glm::vec3(3.0f, 0.0f, 0.0f));**  **glm::vec4 oriPos(0.0f, 0.0f, 0.0f, 1.0f);**  **glm::vec4 earthPos = earthTrans \* oriPos;**  **glm::mat4 eclipticRot = glm::rotate(glm::mat4(1.0f), glm::radians(23.5f), glm::vec3(0.0f, 0.0f, 1.0f));**  **glm::vec4 earthAix4 = eclipticRot \* glm::vec4(0.0f, 1.0f, 0.0f, 0.0f);**  **glm::vec3 earthAix3 = glm::vec3(earthAix4);**  **glm::mat4 earthPosTrans = glm::translate(glm::mat4(1.0f), glm::vec3(earthPos));**  **earthTrans = earthPosTrans \* eclipticRot \* glm::scale(one, glm::vec3(0.3f, 0.3f, 0.3f)) \* glm::rotate(one, glm::radians(selfRot), glm::vec3(0.0, 1.0, 0.0));**  **;**  **shaderProgram.setMatrix4fv("model", glm::value\_ptr(earthTrans));**  **// 贴图**  **glBindTexture(GL\_TEXTURE\_2D, earthTex);**  **glDrawElements(GL\_TRIANGLES, ballSize, GL\_UNSIGNED\_INT, 0);**  **// 绘制moon**  **glm::mat4 moonTrans = earthPosTrans \* glm::rotate(glm::mat4(1.0f), glm::radians(moonRot), earthAix3) \* eclipticRot \* glm::translate(glm::mat4(1.0f), glm::vec3(0.5f, 0.0f, 0.0f)) \* glm::scale(glm::mat4(1.0f), glm::vec3(0.1f, 0.1f, 0.1f));**  **shaderProgram.setMatrix4fv("model", glm::value\_ptr(moonTrans));**  **// 贴图**  **glBindTexture(GL\_TEXTURE\_2D, moonTex);**  **glDrawElements(GL\_TRIANGLES, ballSize, GL\_UNSIGNED\_INT, 0);**  **// 绘制背景**  **glBindVertexArray(backVAO); // 绑定VAO**  **shaderProgram.setInt("background", 1);**  **shaderProgram.setMatrix4fv("model", glm::value\_ptr(one));**  **shaderProgram.setMatrix4fv("view", glm::value\_ptr(one));**  **shaderProgram.setMatrix4fv("projection", glm::value\_ptr(one));**  **// 贴图**  **glBindTexture(GL\_TEXTURE\_2D, backTex);**  **glDrawElements(GL\_TRIANGLES, backSize, GL\_UNSIGNED\_INT, 0);**  **glBindVertexArray(0);**  **}**  **void reshaper(GLFWwindow \*window, int width, int height)**  **{**  **glViewport(0, 0, width, height);**  **if (height == 0)**  **{**  **aspect = (float)width;**  **}**  **else**  **{**  **aspect = (float)width / (float)height;**  **}**  **}**  **void mouse\_callback(GLFWwindow \*window, double xpos, double ypos)**  **{**  **if (firstMouse)**  **{**  **glfwSetCursorPos(window, 400, 300);**  **xpos = 400;**  **ypos = 300;**  **lastX = 400;**  **lastY = 300;**  **firstMouse = false;**  **}**  **float xoffset = xpos - lastX;**  **float yoffset = lastY - ypos;**  **float sensitivity = 0.01;**  **xoffset \*= sensitivity;**  **yoffset \*= sensitivity;**  **yaw += xoffset;**  **pitch += yoffset;**  **if (pitch > 89.0f)**  **pitch = 89.0f;**  **if (pitch < -89.0f)**  **pitch = -89.0f;**  **glm::vec3 front;**  **front.x = cos(glm::radians(yaw)) \* cos(glm::radians(pitch));**  **front.y = sin(glm::radians(pitch));**  **front.z = sin(glm::radians(yaw)) \* cos(glm::radians(pitch));**  **cameraFront = glm::normalize(front);**  **cameraRight = glm::normalize(glm::cross(cameraFront, glm::vec3(0, 1, 0))); // normalize the vectors, because their length gets closer to 0 the more you look up or down which results in slower movement.**  **cameraUp = glm::normalize(glm::cross(cameraRight, cameraFront));**  **// 重设鼠标位置**  **glfwSetCursorPos(window, 400, 300);**  **}**  **void processInput(GLFWwindow \*window)**  **{**  **if (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS)**  **glfwSetWindowShouldClose(window, true);**  **if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS)**  **viewPos += speed \* cameraFront \* deltaTime;**  **if (glfwGetKey(window, GLFW\_KEY\_S) == GLFW\_PRESS)**  **viewPos -= speed \* cameraFront \* deltaTime;**  **if (glfwGetKey(window, GLFW\_KEY\_A) == GLFW\_PRESS)**  **viewPos -= speed \* cameraRight \* deltaTime;**  **if (glfwGetKey(window, GLFW\_KEY\_D) == GLFW\_PRESS)**  **viewPos += speed \* cameraRight \* deltaTime;**  **// 重置镜头**  **if (glfwGetKey(window, GLFW\_KEY\_R) == GLFW\_PRESS)**  **{**  **yaw = 90;**  **pitch = 0;**  **viewPos = glm::vec3(0.0f, 0.0f, -5.0f);**  **cameraFront = glm::vec3(0, 0, 1.0);**  **cameraRight = glm::vec3(1, 0, 0);**  **cameraUp = glm::vec3(0, 1, 0);**  **}**  **if (glfwGetKey(window, GLFW\_KEY\_Q) == GLFW\_PRESS)**  **{**  **if (!(fullWin & 1))**  **{**  **fullWin++;**  **if (fullWin&2)**  **{**  **glfwSetWindowSize(window, SCR\_WIDTH, SCR\_WIDTH);**  **}**  **else**  **{**  **glfwSetWindowSize(window, fullWidth, fullHeight);**  **}**  **}**  **} else**  **{**  **if (fullWin & 1)**  **{**  **fullWin++;**  **}**  **}**  **if (glfwGetKey(window, GLFW\_KEY\_P) == GLFW\_PRESS)**  **{**  **if (!(pause & 1))**  **{**  **pause++;**  **}**  **} else**  **{**  **if (pause & 1)**  **{**  **pause++;**  **}**  **}**  **}**  **void run(GLFWwindow \*window, float fps)**  **{**  **Shader shaderProgram = initial(); // 初始化**  **float interval = 1000 / fps;**  **auto startTime = std::chrono::high\_resolution\_clock::now();**  **float lastFrame = static\_cast<float>(glfwGetTime());**  **glfwSetInputMode(window, GLFW\_CURSOR, GLFW\_CURSOR\_DISABLED);**  **glfwSetCursorPosCallback(window, mouse\_callback);**  **// 窗口大小改变时调用reshaper函数**  **glfwSetFramebufferSizeCallback(window, reshaper);**  **// 获取主监视器的分辨率**  **GLFWmonitor \*primaryMonitor = glfwGetPrimaryMonitor();**  **const GLFWvidmode \*mode = glfwGetVideoMode(primaryMonitor);**  **fullHeight = mode->height;**  **fullWidth = mode->width;**  **while (!glfwWindowShouldClose(window))**  **{**  **auto endTime = std::chrono::high\_resolution\_clock::now();**  **auto accTime = std::chrono::duration\_cast<std::chrono::milliseconds>(endTime - startTime);**  **// if (accTime.count() < interval)**  **// continue;**  **processInput(window);**  **float currentFrame = static\_cast<float>(glfwGetTime());**  **deltaTime = currentFrame - lastFrame;**  **lastFrame = currentFrame;**  **Draw(shaderProgram);**  **glfwSwapBuffers(window);**  **glfwPollEvents();**  **}**  **// 解绑和删除VAO和VBO**  **glBindVertexArray(0);**  **glBindBuffer(GL\_ARRAY\_BUFFER, 0);**  **glDeleteVertexArrays(1, &ballVAO);**  **glDeleteBuffers(1, &ballVBO);**  **}**  **int main()**  **{**  **glfwInit(); // 初始化GLFW**  **// OpenGL版本为3.3，主次版本号均设为3**  **glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 4);**  **glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 4);**  **// 使用核心模式(无需向后兼容性)**  **glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);**  **// 创建窗口(宽、高、窗口名称)**  **GLFWwindow \*window = glfwCreateWindow(SCR\_WIDTH, SCR\_HEIGHT, "Sphere", NULL, NULL);**  **if (window == NULL)**  **{**  **std::cout << "Failed to Create OpenGL Context" << std::endl;**  **glfwTerminate();**  **return -1;**  **}**  **// 将窗口的上下文设置为当前线程的主上下文**  **glfwMakeContextCurrent(window);**  **// 初始化GLAD，加载OpenGL函数指针地址的函数**  **if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))**  **{**  **std::cout << "Failed to initialize GLAD" << std::endl;**  **return -1;**  **}**  **run(window, 30);**  **glfwDestroyWindow(window);**  **glfwTerminate();**  **delete earthImg;**  **delete sunImg;**  **delete moonImg;**  **delete backImg;**  **return 0;**  **}** |

|  |
| --- |
| **程序2：shader.vs** |
| **#version 330 core**  **layout(location = 0) in vec3 aPos;**  **layout(location = 1) in vec2 aTexCoord;**  **out vec2 TexCoord;**  **out vec3 norm;**  **out vec3 FragPos;**  **uniform mat4 model;**  **uniform mat4 view;**  **uniform mat4 projection;**  **void main() {**  **//gl\_Position = transform \* vec4(vPos, 1.0);**  **norm = normalize(aPos);**  **norm = mat3(transpose(inverse(model))) \* norm;**  **FragPos = vec3(model \* vec4(aPos, 1.0));**  **gl\_Position = projection \* view \* model \* vec4(aPos, 1.0);**  **TexCoord = aTexCoord;**  **}** |

|  |
| --- |
| **程序3：shader.fs** |
| **#version 330 core**  **in vec2 TexCoord;**  **in vec3 norm;**  **in vec3 FragPos;**  **out vec4 FragColor;**  **uniform sampler2D ourTexture;**  **uniform vec3 lightPos;**  **uniform vec3 viewPos;**  **uniform vec3 lightColor;**  **uniform int background;**  **uniform bool sun;**  **void main() {**  **//FragColor = vColor;**  **if(background==1) {**  **FragColor = 0.4 \* texture2D(ourTexture, TexCoord);**  **} else {**  **vec3 objectColor = vec3(texture2D(ourTexture, TexCoord));**  **// ambient**  **float ambientStrength = sun ? 1.0 : 0.3;**  **vec3 ambient = ambientStrength \* lightColor;**  **// diffuse**  **vec3 norm = normalize(norm);**  **vec3 lightDir = normalize(lightPos - FragPos);**  **float diff = max(dot(norm, lightDir), 0.0);**  **vec3 diffuse = 1.0 \* diff \* lightColor;**  **// specular**  **float specularStrength = 3;**  **vec3 viewDir = normalize(viewPos - FragPos);**  **vec3 reflectDir = reflect(-lightDir, norm);**  **float spec = pow(max(dot(viewDir, reflectDir), 0.0), 2);**  **vec3 specular = specularStrength \* spec \* lightColor;**  **vec3 result = (ambient + diffuse + specular) \* objectColor;**  **FragColor = vec4(result, 1.0);**  **}**  **}** |