**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 论述 1](#_Toc60305373)

[2实验（OpenGL） 2](#_Toc60305374)

[2.1 实验内容 2](#_Toc60305375)

[2.2 实验方法和过程 2](#_Toc60305376)

[2.3 实验结果 6](#_Toc60305377)

[2.4 心得体会 8](#_Toc60305378)

[2.5 源代码 9](#_Toc60305379)

# 1 论述

（1）我选修计算机图形学想学到关于图形学的相关知识，还有图形学应用，特别是游戏引擎方面的应用。现在课程结束我觉得我还是收获到了很多，特别是最后的结课实验，让我又好好温习了一遍图形学的知识以及提高自己的动手能力。另外，需要进一步补足，特别是复习相关知识的话，我还是会去网上进一步学习，这里附上一个opengl的教程链接https://learnopengl-cn.github.io

在这里面我学到了很多，对于最后的结课实验，我又通过这个教程重新学习了一遍。这里面还有实验中没有的进阶教程例如高级光照之类的，我觉得是非常值得拿出来分享的一个教程。

（2）对于图形的变换我比较感兴趣，利用矩阵相乘实现缩放，位移和旋转展现了线性代数的强大和神奇，缩放和位移都相对比较简单，缩放只要乘以一个缩放的因子，位移利用了w分量，也就是齐次分量，通过将3维坐标增加一个齐次分量升为到4维，变可以通过一个矩阵实现位移，我觉得是非常巧妙的，最后旋转是真的复杂，甚至还可能产生万向节死锁问题（稍微了解了一下，不过还是不太懂）。总的来说，把要变换的向量升到4维以后，可以通过与变换矩阵相乘实现基本的变换，不过要注意乘的顺序，因为矩阵乘法不满足交换律，正确的顺序应该是从右往左看。一般先缩放，后旋转，最后位移。

（3）我觉得这是一门理论与实践想结合的课，目前来说，这门课最需要的是增加上机实验课，现在就只有一个结课实验实在是有点少。

# 2实验（OpenGL）

**2.1 实验内容**

利用OpenGL框架，设计一个日地月运动模型动画。

（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。

**2.2 实验方法和过程**

**2.2.1 绘制球体**

第一步先要绘制球体，需要球体的顶点坐标，同时为了接下来贴纹理，所以还需要纹理坐标，因为球体的法向量和顶点坐标是一致的，因此我就没有另外保存法向量。

根据球体的参数方程：

其中xyz作为顶点坐标，uv作为纹理坐标。

一共使用20\*20=400个顶点数据生成球体，生成绑定球体的VAO和VBO后将这些顶点坐标和纹理坐标数据绑定到缓冲中，因为每次需要6个顶点绘制一个矩形，其中2个顶点是重复的，所以为了不重复保存数据，使用EBO生成球体。最后设置顶点属性指针，保存了三维的顶点坐标和二维的纹理坐标，所以步长都是5\*sizeof(float)，顶点坐标的起始偏移量是0，纹理坐标的起始偏移量是3\*sizeof(float)。将这些数据传给顶点着色器完成球体的绘制。

**2.2.2 纹理贴图**

上网找到了地球，太阳和月亮的bmp格式的纹理文件，bmp格式的文件通道只有RGB三个。所以使用glTexImage2D的时候注意参数是GL\_RGB而不是GL\_RGBA。使用3个纹理句柄分别绑定日地月的纹理贴图，在绘制的时候使用相应的句柄绑定对应纹理贴图即可。

**2.2.3 运动变换**

太阳的变换最为简单，只需要绕着y轴自转即可。使用glfwGetTime()获取程序运行的时间currentTime，设置glm::rotate旋转的参数，绕y轴旋转currentTime\*每秒旋转的角度即可。

地球自转的同时还绕着太阳公转，并且还要考虑黄道夹角。为了方便计算，这里将黄道夹角简化为45°，这样旋转轴也就可以简化为将（1.0f,1.0f,0.0f）归一化后的结果，自转的方法同太阳，只要改下旋转轴即可，另外设置好速度。最后地球绕着太阳公转，可以使用位移矩阵解决，先看椭圆参数方程：

设定a=6.0f，b=3.0f，角度是时间乘以角速度，公转的速度是自转速度的1/365，这样就完成了一个椭圆轨道的公转。另外还需要把地球缩放为0.4倍，当然这是最先做的变换，最后的模型矩阵是这个样子的：

model = trans\_revolution\_earth \*rot\_self\* rot\_tilt \* scale\_earth;

scale\_earth是缩放矩阵，缩小为0.4倍，rot\_tilt将地球倾斜45度，rot\_self控制地球绕着斜45度的轴随时间自转，trans\_revolution\_earth控制地球绕着太阳在一个椭圆轨道上随着时间公转。

月球绕着地球在圆轨道公转，同时又在绕太阳的椭圆轨道公转，使用圆的坐标方程：

设定r=1.0f，月球绕地球的公转速度约为地球自转的1/28。然后让这个圆轨道倾斜45°，最后再乘上地球绕太阳的公转轨道位移矩阵就可以实现。另外月球缩放为0.1f。

月球的模型矩阵：

model = trans\_revolution\_earth \* rot\_tilt \* trans\_revolution\_moon \* rot\_self \* scale\_moon;

scale\_moon缩放月球为0.1倍，rot\_self控制月球自旋，trans\_revolution\_moon是随着时间绕着地球旋转的一个圆轨道，rot\_tilt使得圆轨道绕z轴倾斜45度，使得圆轨道面与地轴垂直。trans\_revolution\_earth即为地球随时间变化绕太阳的椭圆轨道矩阵。

**2.2.4 光照处理**

光照分为环境光照，漫反射，镜面反射和点光源。

环境光照只需要设置常量环境因子为0.5f，乘以物体的纹理颜色rgb即可。

计算漫反射光照需要法向量和光源与片段位置向量差的方向向量。其中法向量可以直接使用球体的顶点坐标aPos：

Normal = mat3(transpose(inverse(model))) \* aPos;

在顶点着色器中使用inverse和transpose函数生成法线矩阵并传递给片段着色器。

光源的坐标为(0.0f,0.0f,0.0f)，片段位置向量由顶点着色器计算传递：

FragPos = vec3(model \* vec4(aPos, 1.0));

将法向量和方向向量点乘得到漫反射因子，计算漫反射分量。

镜面光照还需要观察者坐标，使用摄像机的坐标作为观察者坐标，利用观察者坐标和片段坐标计算视线方向向量：

vec3 viewDir = normalize(viewPos - FragPos);

利用灯光方向向量和法向量计算反射向量：

vec3 reflectDir = reflect(-lightDir, norm);

将上面两个向量点乘计算镜面分量：

float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);

其中,shininess是反光度，反光度越高，反射能力越强，散射越少，光高点越小。这里设置为2.0f。为了让镜面光照对最终结果不产生过大的影响，设置镜面强度为0.05f。

点光源的公式如下：

常数项设置为1.0f，一次项系数设置为0.045f，二次项系数设置为0.0075f，根据片段距离光源的距离计算衰减系数。

**2.3 实验结果**

实验结果如图 1，地球绕太阳的公转轨道和月球绕地球的公转轨道不在一个平面，有45°的夹角，由图 2可以看出地球的自转以及光照的处理，由图 3可以看到正确的距离关系，开启了深度测试，有遮挡效果。

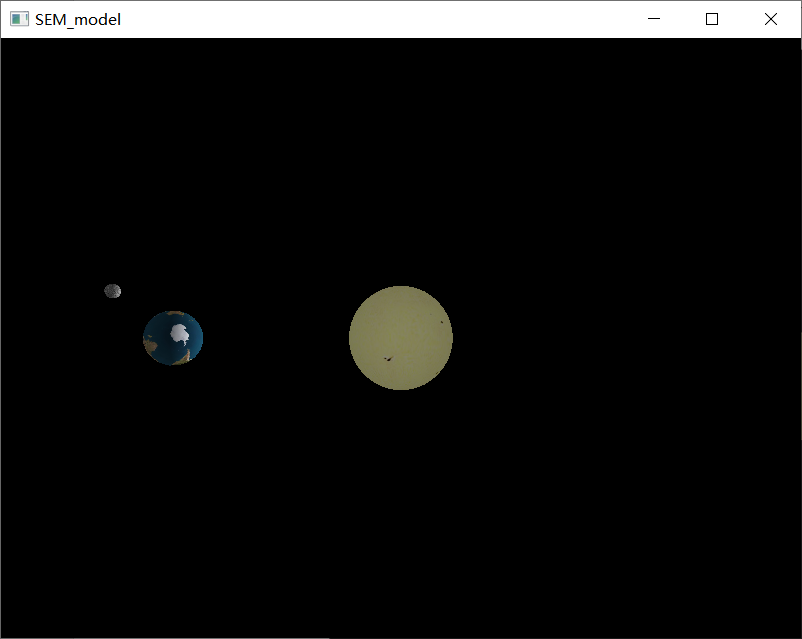


图 1日地月模型轨迹和纹理贴图

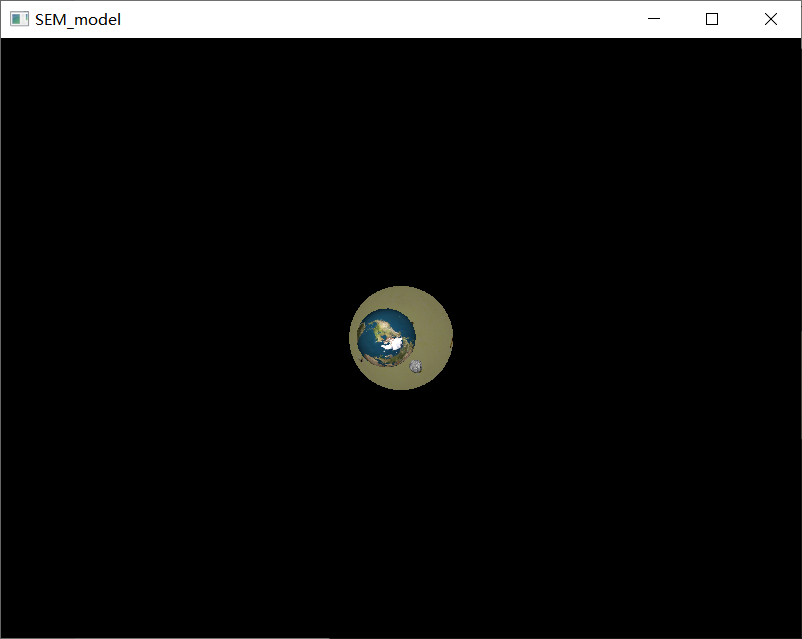


图 2日地月模型光照处理

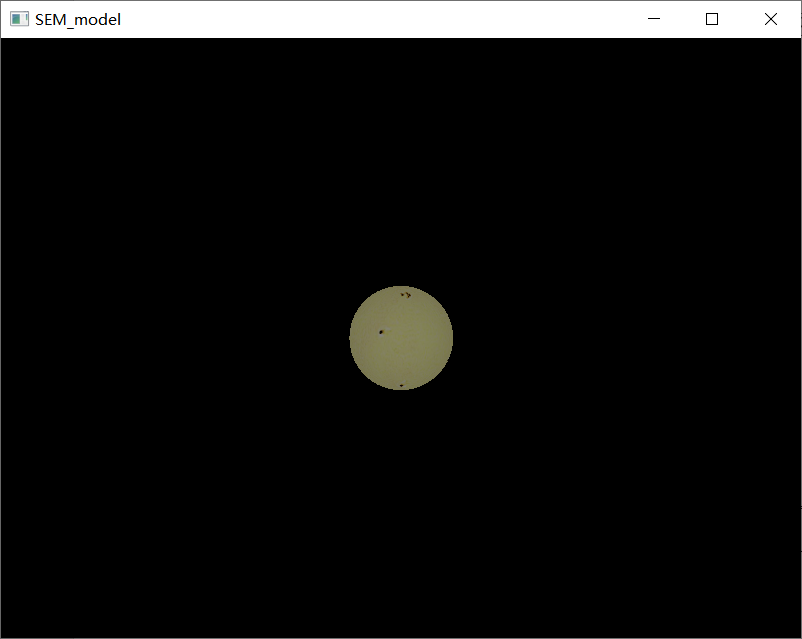


图 3日地月模型深度测试

**2.4 心得体会**

正如我论述里面所说，图形学是一门理论与实践相结合的课，只有通过亲自的动手实验才能感受图形学的魅力（和困难）。图形学的理论听起来就跟数学课一样，本身图形学就是与几何学还有线性代数密切相关的，所以听理论的时候难免会觉得枯燥，这个时候实际动手往往能更快掌握和理解，不过opengl的入门还是有些劝退的，配置环境，接触各种glfw/glad库的函数，学习并使用它们真的是一个漫长的过程。所以最后我对自己还是比较满意的，我感觉通过这个实验，强迫自己去学习使用glfw/glad，真的学到了很多东西，从最开始难以理解VAO，VBO和EBO到最后完成整个实验，收获颇丰。最后感谢何老师上课认真的讲解，虽然我在上课的时候吸收的效率真的很低，但是做到的时候，偶然突然想起来“这里我好像上课听过”就算没有白去上课。还有感谢老师在群里的每个阶段的代码，再结合网上的教程，对我有很大的参考价值。

**2.5 源代码**

|  |
| --- |
| 程序1：window.cpp |
| #include <glad/glad.h>  #include <GLFW/glfw3.h>  #include <iostream>  #include <vmath.h>  #include <vector>  #include <glm/glm.hpp>  #include <glm/gtc/matrix\_transform.hpp>  #include <glm/gtc/type\_ptr.hpp>  #include "shader.h"  #include "camera.h"  #define STB\_IMAGE\_IMPLEMENTATION  #include "stb/stb\_image.h"  using namespace std;  //窗口大小参数  const unsigned int SCR\_WIDTH = 800;  const unsigned int SCR\_HEIGHT = 600;  float aspact = (float)4.0 / (float)3.0;  //指定摄像机  Camera camera(glm::vec3(0.0f, 0.0f, 10.0f));  float lastX = SCR\_WIDTH / 2.0f;  float lastY = SCR\_HEIGHT / 2.0f;  bool firstMouse = true;  // 指定时间参数  float deltaTime = 0.0f;  float lastFrame = 0.0f;  //句柄参数  GLuint vertex\_array\_object; // == VAO句柄  GLuint vertex\_buffer\_object; // == VBO句柄  GLuint element\_buffer\_object;//==EBO句柄  GLuint texture\_buffer\_object1; // 纹理对象句柄1  GLuint texture\_buffer\_object2; // 纹理对象句柄2  GLuint texture\_buffer\_object3; // 纹理对象句柄3  //球的数据参数  std::vector<int> sphereIndices;  const int Y\_SEGMENTS = 20;  const int X\_SEGMENTS = 20;  const float Radio = 1.0;  const GLfloat PI = 3.14159265358979323846f;  vector<float> Vertices;  void initial(void)  {  //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 <= Y\_SEGMENTS; y++)  {  for (int x = 0; x <= X\_SEGMENTS; x++)  {  float xSegment = (float)x / (float)X\_SEGMENTS;  float ySegment = (float)y / (float)Y\_SEGMENTS;  float xPos = Radio \* std::cos(2 \* xSegment \* PI) \* std::sin(ySegment \* PI);  float yPos = std::cos(ySegment \* PI);  float zPos = Radio \* std::sin(2 \* xSegment \* PI) \* std::sin(ySegment \* PI);  Vertices.push\_back(xPos);  Vertices.push\_back(yPos);  Vertices.push\_back(zPos);  Vertices.push\_back(xSegment);  Vertices.push\_back(ySegment);  }  }  // 生成球的顶点  for (int i = 0; i < Y\_SEGMENTS; i++)  {  for (int j = 0; j < X\_SEGMENTS; j++)  {  sphereIndices.push\_back(i \* (X\_SEGMENTS + 1) + j);  sphereIndices.push\_back((i + 1) \* (X\_SEGMENTS + 1) + j);  sphereIndices.push\_back((i + 1) \* (X\_SEGMENTS + 1) + j + 1);  sphereIndices.push\_back(i \* (X\_SEGMENTS + 1) + j);  sphereIndices.push\_back((i + 1) \* (X\_SEGMENTS + 1) + j + 1);  sphereIndices.push\_back(i \* (X\_SEGMENTS + 1) + j + 1);  }  }  // 球  glGenVertexArrays(1, &vertex\_array\_object);  glGenBuffers(1, &vertex\_buffer\_object);  //生成并绑定球体的VAO和VBO  glBindVertexArray(vertex\_array\_object);  glBindBuffer(GL\_ARRAY\_BUFFER, vertex\_buffer\_object);  // 将顶点数据绑定至当前默认的缓冲中  glBufferData(GL\_ARRAY\_BUFFER, Vertices.size() \* sizeof(float) \* 5, &Vertices[0], GL\_STATIC\_DRAW);  glGenBuffers(1, &element\_buffer\_object);  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, element\_buffer\_object);  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);  glEnableVertexAttribArray(0);  glVertexAttribPointer(1, 2, GL\_FLOAT, GL\_FALSE, 5 \* sizeof(float), (void\*)(3 \* sizeof(float)));  glEnableVertexAttribArray(1);    //加载纹理数据  //地球  glGenTextures(1, &texture\_buffer\_object1);  glBindTexture(GL\_TEXTURE\_2D, texture\_buffer\_object1);  //指定纹理的参数  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);  //加载纹理  int width, height, nrchannels;//纹理长宽，通道数  //stbi\_set\_flip\_vertically\_on\_load(true);  //加载纹理图片wall.jpg或者htmdb.png  unsigned char\* data = stbi\_load("res/earth.bmp", &width, &height, &nrchannels, 0);  if (data)  {  glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, data);  //生成Mipmap纹理  glGenerateMipmap(GL\_TEXTURE\_2D);  }  else  {  std::cout << "Failed to load texture" << std::endl;  }  stbi\_image\_free(data);//释放资源  //太阳  glGenTextures(1, &texture\_buffer\_object2);  glBindTexture(GL\_TEXTURE\_2D, texture\_buffer\_object2);  //指定纹理的参数  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);  //加载纹理  //stbi\_set\_flip\_vertically\_on\_load(true);  //加载纹理图片wall.jpg或者htmdb.png  data = stbi\_load("res/sun.bmp", &width, &height, &nrchannels, 0);  if (data)  {  glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, data);  //生成Mipmap纹理  glGenerateMipmap(GL\_TEXTURE\_2D);  }  else  {  std::cout << "Failed to load texture" << std::endl;  }  stbi\_image\_free(data);//释放资源  //月亮  glGenTextures(1, &texture\_buffer\_object3);  glBindTexture(GL\_TEXTURE\_2D, texture\_buffer\_object3);  //指定纹理的参数  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);  //加载纹理  //stbi\_set\_flip\_vertically\_on\_load(true);  //加载纹理图片wall.jpg或者htmdb.png  data = stbi\_load("res/moon.bmp", &width, &height, &nrchannels, 0);  if (data)  {  glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, data);  //生成Mipmap纹理  glGenerateMipmap(GL\_TEXTURE\_2D);  }  else  {  std::cout << "Failed to load texture" << std::endl;  }  stbi\_image\_free(data);//释放资源  //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);  }  void key\_callback(GLFWwindow\* window, int key, int scancode, int action, int mods)  {  switch (key)  {  case GLFW\_KEY\_ESCAPE:  glfwSetWindowShouldClose(window, GL\_TRUE);  break;  case GLFW\_KEY\_L:  glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);  break;  case GLFW\_KEY\_F:  glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_FILL);  break;  case GLFW\_KEY\_B:  glEnable(GL\_CULL\_FACE); //打开背面剔除  glCullFace(GL\_BACK); //剔除多边形的背面  break;  case GLFW\_KEY\_C:  glDisable(GL\_CULL\_FACE); //关闭背面剔除  break;  default:  break;  }  }  void processInput(GLFWwindow\* window)  {  if (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS)  glfwSetWindowShouldClose(window, true);  if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS)  camera.ProcessKeyboard(FORWARD, deltaTime);  if (glfwGetKey(window, GLFW\_KEY\_S) == GLFW\_PRESS)  camera.ProcessKeyboard(BACKWARD, deltaTime);  if (glfwGetKey(window, GLFW\_KEY\_A) == GLFW\_PRESS)  camera.ProcessKeyboard(LEFT, deltaTime);  if (glfwGetKey(window, GLFW\_KEY\_D) == GLFW\_PRESS)  camera.ProcessKeyboard(RIGHT, deltaTime);  }  //鼠标滚轮回调函数  void scroll\_callback(GLFWwindow\* window, double xoffset, double yoffset)  {  //对摄像机进行操作  camera.ProcessMouseScroll((float)yoffset);  }  // 鼠标移动回调函数  void mouse\_callback(GLFWwindow\* window, double xpos, double ypos)  {  if (firstMouse)  {  lastX = (float)xpos;  lastY = (float)ypos;  firstMouse = false;  }  float xoffset = (float)xpos - lastX;  float yoffset = lastY - (float)ypos; // y坐标系进行反转  lastX = (float)xpos;  lastY = (float)ypos;  camera.ProcessMouseMovement(xoffset, yoffset);  }  void reshaper(GLFWwindow\* window, int width, int height)  {  glViewport(0, 0, width, height);  if (height == 0)  {  aspact = (float)width;  }  else  {  aspact = (float)width / (float)height;  }  }  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, 3);  glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);  // 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, "SEM\_model", 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;  }  initial();//初始化  //窗口大小改变时调用reshaper函数  glfwSetFramebufferSizeCallback(window, reshaper);  //窗口中有键盘操作时调用key\_callback函数  glfwSetKeyCallback(window, key\_callback);  //鼠标回调函数  glfwSetCursorPosCallback(window, mouse\_callback);  //当鼠标滚轮滚动时调用函数scroll\_callback  glfwSetScrollCallback(window, scroll\_callback);  Shader lightingShader("sphere.vs", "sphere.fs");  lightingShader.use();  lightingShader.setInt("texture1", 0);  float a = 6.0f, b = 3.0f;  float r = 1.0f;  while (!glfwWindowShouldClose(window))  {  //处理时间  float currentFrame = (float)glfwGetTime();  deltaTime = currentFrame - lastFrame;  lastFrame = currentFrame;  // 输入处理  processInput(window);  // 清空颜色缓冲和深度缓冲区  glClearColor(0.0f, 0.0f, 0.0f, 1.0f);  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  // 激活光照  lightingShader.use();  lightingShader.setVec3("light.position", glm::vec3(0.0f,0.0f,0.0f));  lightingShader.setVec3("light.direction", camera.Front);  lightingShader.setVec3("viewPos", camera.Position);  //光源属性  lightingShader.setVec3("light.ambient", 0.5f, 0.5f, 0.5f);  lightingShader.setVec3("light.diffuse", 1.0f, 1.0f, 1.0f);  lightingShader.setVec3("light.specular", 0.05f, 0.05f, 0.05f);  lightingShader.setFloat("light.constant", 1.0f);  lightingShader.setFloat("light.linear", 0.045f);  lightingShader.setFloat("light.quadratic", 0.0075f);  //反光度  lightingShader.setFloat("shininess", 2.0f);  //地球  glActiveTexture(GL\_TEXTURE0);  glBindTexture(GL\_TEXTURE\_2D,texture\_buffer\_object1);  glm::mat4 projection = glm::mat4(1.0f);  projection = glm::perspective(glm::radians(60.0f), (float)SCR\_WIDTH / (float)SCR\_HEIGHT, 0.1f, 100.0f);  glm::mat4 view = camera.GetViewMatrix();    lightingShader.setMat4("projection", projection);  lightingShader.setMat4("view", view);  float currentTime = (float)glfwGetTime();  glBindVertexArray(vertex\_array\_object);  glm::mat4 model = glm::mat4(1.0f);  glm::mat4 scale\_earth = glm::mat4(1.0f);  glm::mat4 trans\_revolution\_earth = glm::mat4(1.0f);  glm::mat4 rot\_tilt = glm::mat4(1.0f);  glm::mat4 rot\_self = glm::mat4(1.0f);  scale\_earth = glm::scale(scale\_earth, glm::vec3(0.4, 0.4, 0.4));  rot\_tilt = glm::rotate(rot\_tilt, glm::radians(45.0f), glm::vec3(0.0f, 0.0f, 1.0f));  trans\_revolution\_earth = glm::translate(trans\_revolution\_earth, glm::vec3(-a \* std::cos(PI/4\*currentTime), 0.0f, b \* std::sin(PI/4 \* currentTime)));    rot\_self = glm::rotate(rot\_self, -365 \* currentTime\* glm::radians(45.0f), glm::normalize(glm::vec3(1.0f, 1.0f, 0.0f)));  model = trans\_revolution\_earth \*rot\_self\* rot\_tilt \* scale\_earth;    lightingShader.setMat4("model", model);  glDrawElements(GL\_TRIANGLES, X\_SEGMENTS \* Y\_SEGMENTS \* 6, GL\_UNSIGNED\_INT, 0);    //太阳  glActiveTexture(GL\_TEXTURE0);  glBindTexture(GL\_TEXTURE\_2D, texture\_buffer\_object2);  lightingShader.setMat4("projection", projection);  lightingShader.setMat4("view", view);  model = glm::mat4(1.0f);  model = glm::rotate(model, currentTime \* glm::radians(10.0f), glm::vec3(0.0f, 1.0f, 0.0f));  lightingShader.setMat4("model", model);  glDrawElements(GL\_TRIANGLES, X\_SEGMENTS \* Y\_SEGMENTS \* 6, GL\_UNSIGNED\_INT, 0);    //月亮  glActiveTexture(GL\_TEXTURE0);  glBindTexture(GL\_TEXTURE\_2D, texture\_buffer\_object3);    lightingShader.setMat4("projection", projection);  lightingShader.setMat4("view", view);  glBindVertexArray(vertex\_array\_object);  glm::mat4 scale\_moon = glm::mat4(1.0f);  glm::mat4 trans3 = glm::mat4(1.0f);  glm::mat4 trans\_revolution\_moon = glm::mat4(1.0f);  scale\_moon = glm::scale(scale\_moon, glm::vec3(0.1, 0.1, 0.1));  trans3 = glm::translate(trans3, glm::vec3(a, 0.0f, 0.0f));  trans\_revolution\_moon = glm::translate(trans\_revolution\_moon, glm::vec3(r \* std::cos(PI \*7 \* currentTime), 0.0f, r \* std::sin(PI \* 7 \* currentTime)));  rot\_tilt = glm::mat4(1.0f);  rot\_tilt = glm::rotate(rot\_tilt, glm::radians(-45.0f), glm::vec3(0.0f, 0.0f, 1.0f));  model = trans\_revolution\_earth \* rot\_tilt \* trans\_revolution\_moon \* rot\_self \* scale\_moon;  lightingShader.setMat4("model", model);  glDrawElements(GL\_TRIANGLES, X\_SEGMENTS \* Y\_SEGMENTS \* 6, GL\_UNSIGNED\_INT, 0);  glfwSwapBuffers(window);  glfwPollEvents();  }  // 解绑和删除VAO和VBO  glBindVertexArray(0);  glBindBuffer(GL\_ARRAY\_BUFFER, 0);  glDeleteVertexArrays(1, &vertex\_array\_object);  glDeleteBuffers(1, &vertex\_buffer\_object);  glfwDestroyWindow(window);  glfwTerminate();  return 0;  } |

|  |
| --- |
| 程序2：sphere.vs |
| #version 330 core  layout (location = 0) in vec3 aPos;  layout (location = 1) in vec2 aTexCoord;  out vec4 vColor;  out vec2 TexCoords;  out vec3 Normal;  out vec3 FragPos;  uniform mat4 model;  uniform mat4 view;  uniform mat4 projection;  void main()  {  gl\_Position = projection \* view \* model \* vec4(aPos, 1.0f);  FragPos = vec3(model \* vec4(aPos, 1.0));  Normal = mat3(transpose(inverse(model))) \* aPos;  TexCoords = aTexCoord;  } |

|  |
| --- |
| 程序3：sphere.fs |
| #version 330 core  in vec4 vColor;  in vec2 TexCoords;  in vec3 Normal;  in vec3 FragPos;  out vec4 FragColor;  uniform mat4 transform;  uniform sampler2D texture1;  struct Light {  vec3 position;    vec3 ambient;  vec3 diffuse;  vec3 specular;    float constant;  float linear;  float quadratic;  };  uniform Light light;  uniform vec3 viewPos;  uniform float shininess;  void main()  {    // 环境光处理  vec3 ambient = light.ambient \* texture(texture1, TexCoords).rgb;    // 漫反射光处理  vec3 norm = normalize(Normal);  vec3 lightDir = normalize(light.position - FragPos);  float diff = max(dot(norm, lightDir), 0.0);  vec3 diffuse = light.diffuse \* diff \* texture(texture1, TexCoords).rgb;    // 镜面光处理  vec3 viewDir = normalize(viewPos - FragPos);  vec3 reflectDir = reflect(-lightDir, norm);  float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);  vec3 specular = light.specular \* spec \* texture(texture1, TexCoords).rgb;      //光的衰减  float distance = length(light.position - FragPos);  float attenuation = 1.0 / (light.constant + light.linear \* distance + light.quadratic \* (distance \* distance));  ambient \*= attenuation;  diffuse \*= attenuation;  specular \*= attenuation;    vec3 result = min(ambient + diffuse + specular,vec3(1.0));  FragColor = vec4(result, 1.0);    } |