计算机图形学课程报告

计算机科学与技术学院

班　　级: CS2107

学　　号: U202115538

姓　　名: 陈侠锟

指导教师: 何云峰

完成日期: 2023/12/6

1 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。

答：计算机图形学在许多领域如计算机游戏、视觉特效、数据可视化等都有着广泛的应用，我最开始选修这么课程是因为对于计算机游戏界面渲染方面的兴趣。在玩大型游戏时，很容易就能遇到各种材质布料的仿真、水面云层之类的模拟、光线追踪效果的实现等等。玩的久了，自己也想去探索一下如何实现这些效果。对于课程我是很满意的，老师的讲解比较清晰，速度也始终，PPT 除了基础知识点之外也有一些关于代码的解读，对像我这样的图形学小白来说还是比较友好的，两次作业以及最后的大作业也比较基础（虽然对我来说还是难度不小）。不过在初步学习过后，越发觉得这门课程体系的庞大，今后我也会根据自己的兴趣去看一些相关领域的文献，进一步加深了解。

（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。

答：经过本学期课程的初步学习之后，我对光照这部分印象最深，也最感兴趣。在真正开始学习之前，我一度以为光照效果的实现不会很复杂，但当我真正上手之后才发现，平常我们觉得简单的事物（就比如光照），用代码实现的难度可能是非常非常大的。就算是 OpenGL 对光照进行了简化，也还是需要考虑环境光照、漫反射光照、镜面光照、不同材质对光照的影响等等。经本学期的学习之后，我首先对一些定义有了更深刻的认识，如环境光照：即使在黑暗的情况下，世界上通常也仍然有一些光亮（月亮、远处的光），所以物体几乎永远不会是完全黑暗的。为了模拟这个，我们会使用一个环境光照常量，它永远会给物体一些颜色；漫反射光照：模拟光源对物体的方向性影响(Directional Impact)。它是冯氏光照模型中视觉上最显著的分量。物体的某一部分越是正对着光源，它就会越亮；镜面光照：模拟有光泽物体上面出现的亮点。镜面光照的颜色相比于物体的颜色会更倾向于光的颜色。此外，我还学习到了如点光源、聚光灯等多种光源模型，一系列的内容使我对如何产生逼真的三维场景有了更加系统的认识。将来我会更加专注对光照系统的研究，掌握更多在图像合成方面的光照技巧，寻求更加丰富的视觉效果。

（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。

总体而言，我觉得老师讲课讲的很好，但美中不足的是（我认为）对代码的讲解过少，实操的机会也较少（因为大部分情况下，只有实验或者作业才能有实操的机会）。因为即使理解原理再透彻，也还是不会写代码，只有在充分的练习过后才能知道许多函数的使用方法，或者是仿真时经常会遇到的一些坑。就拿这

学期前两次的课后作业来说，修改的代码量非常少，基本不超过 10 行就可以实现（也就是说可能根本不需要看懂整个代码文件而只需要看懂一小段代码就行了）。而大作业要求我们在第二次作业的基础上加入纹理和光照，改动代码量较大，并且尽管有一些实例程序，但是并不太符合同学们的需求，这就导致了大作业的难度变得非常大。也就是说，我比较希望在今后可以将这门课程改为理论+实验的方式进行。

2 实验报告

2.1 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。

2.2 实验方法和过程

（1）环境搭建

使用老师提供的 vmath.h，并从网上下载 camera.h、shader.h 以及 glm 的相关文件，放置在 opengl 文件夹中，使用 vs2019 进行编程，包含目录及库目录修改为对应路径，如图 2-1 所示。

包含目录	**D:\ComputerGraphics\opengl\include;$(IncludePath)**
引用目录	$(VC_ReferencesPath_x86);
库目录	**D:\ComputerGraphics\opengl\libs;$(LibraryPath)**

图 2-1 包含目录和库目录路径

代码引用的头文件如图 2-2 所示。

```
#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 <map>
#include "shader.h"
#include "camera.h"
#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>
```

图 2-2 代码引用的头文件

注意在引用 stb_image.h 头文件前，定义 STB_IMAGE_IMPLEMENTATION，否则会出现如图 2-3 所示的错误。

代	说明	项目	文件	行
LNK112(	3 个无法解析的外部命令	transformation	transformation.exe	1
LNK201(	无法解析的外部符号 _stbi_image_free，该符号在函数 "void __cdecl initial (void)" (?initial@@YAXXZ) 中被引用	transformation	transformation.obj	1
LNK201(	无法解析的外部符号 _stbi_load，该符号在函数 "void __cdecl initial(void)" (?	transformation	transformation.obj	1

图 2-3 未定义 STB_IMAGE_IMPLEMENTATION 导致的错误

（2）日地月模型初步实现

在第二次作业中，我们已经初步实现了日月地模型，只需在其基础上稍加修改，以达到“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”的要求，同时，给太阳、地球、月球添加自转效果。首先是太阳，模型定义如图 2-4 所示。

```
glm::mat4 scale_sun = glm::scale(glm::mat4(1.0f), glm::vec3(2.4, 2.4, 2.4));
glm::mat4 rot_self_sun = glm::rotate(glm::mat4(1.0f), currentTime * glm::radians(10.0f), glm::vec3(0.0f, 1.0f, 0.0f));
glm::mat4 model_sun = rot_self_sun * scale_sun;

lightingShader.setMat4("model", model_sun);
```

图 2-4 太阳模型定义

其中，glm::scale()函数的作用是将物体进行缩放，glm::rotate()函数的作用是将物体进行旋转。currentTime 是用 glfwGetTime()函数获取的时间，乘上一个弧度制的度数，使得太阳能够随时间变化而自动旋转，glm::vec3(0.0f, 1.0f, 0.0f)表示沿 y 轴旋转。

对于地球和月亮，自转和公转都要考虑，且为了贴合实际情况，自转轴和公转轴不应与太阳的自转轴完全相同，地球的模型定义如图 2-5 所示。

```
glm::mat4 scale_earth = glm::scale(glm::mat4(1.0f), glm::vec3(0.6, 0.6, 0.6));
glm::mat4 rot_tilt_earth = glm::rotate(glm::mat4(1.0f), glm::radians(135.0f), glm::vec3(0.0f, 0.0f, 1.0f)); //倾斜角度
glm::mat4 trans_revolution_earth = glm::translate(glm::mat4(1.0f), glm::vec3(-a * std::cos(PI / 3 * currentTime), 0.0f, b * std::sin(PI / 3 * currentTime)));   //公转
glm::mat4 rot_self_earth = glm::rotate(glm::mat4(1.0f), -10 * currentTime * glm::radians(45.0f), glm::normalize(glm::vec3(1.0f, 1.0f, 0.0f)));  //自转
glm::mat4 model_earth = trans_revolution_earth * rot_self_earth * rot_tilt_earth * scale_earth;

lightingShader.setMat4("model", model_earth);       //定义模型变换矩阵
```

图 2-5 地球模型定义

为了便于观察，将地球设置为了太阳的四分之一倍大小（尽管不是很贴合实际）。图中 rot_tilt_earth 这一项用于调整地球的自转轴角度，trans_revolution_earth 这一项代表地球的公转数据，所用到的 glm::translate()函数用于将物体进行位移，参数中同样使用了 currentTime 以保证地球能够自动进行公转。

月球模型定义如图 2-6 所示，不再过多赘述，只需修改相关参数以使其公转平面与地球的公转平面不同即可。

```
glm::mat4 scale_moon = glm::scale(glm::mat4(1.0f), glm::vec3(0.2, 0.2, 0.2));
glm::mat4 trans_revolution_moon = glm::translate(glm::mat4(1.0f), glm::vec3(r * std::cos(PI * 7 * currentTime), 0.0f, r * std::sin(PI * 7 * currentTime)));
glm::mat4 rot_tilt_moon = glm::rotate(glm::mat4(1.0f), glm::radians(-45.0f), glm::vec3(0.0f, 0.0f, 1.0f));
glm::mat4 rot_self_moon = glm::rotate(glm::mat4(1.0f), -365 * currentTime * glm::radians(45.0f), glm::normalize(glm::vec3(1.0f, 1.0f, 0.0f)));
glm::mat4 model_moon = trans_revolution_earth * rot_tilt_moon * trans_revolution_moon * rot_self_moon * scale_moon;

lightingShader.setMat4("model", model_moon);
```

图 2-6 月球模型定义

（3）添加纹理

首先来看生成球顶点部分的代码，如图 2-7 所示。

```
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 = std::cos(xSegment * Radio * PI) * std::sin(ySegment * PI);
        float yPos = std::cos(ySegment * PI);
        float zPos = std::sin(xSegment * Radio * PI) * std::sin(ySegment * PI);
        glm::vec3 normal = glm::normalize(glm::vec3(xPos, yPos, zPos));
        //位置
        sphereVertices.push_back(xPos);
        sphereVertices.push_back(yPos);
        sphereVertices.push_back(zPos);
        //法向量
        sphereVertices.push_back(normal.x);
        sphereVertices.push_back(normal.y);
        sphereVertices.push_back(normal.z);
        //添加纹理坐标
        sphereVertices.push_back(xSegment);
        sphereVertices.push_back(ySegment);
    }
}
```

图 2-7 生成球顶点部分的代码

可以看到在第二次作业的基础上，我们新保存了两组数据，第一组是法向量，这将在后续添加光照中发挥作用，第二组是纹理坐标，为了能够把纹理映射到图形上，我们需要指定图形的每个顶点各自对应纹理的哪个部分，这样每个顶点就会关联一个纹理坐标，用来标明该顶点从纹理图像的哪个部分采样。纹理坐标在 x 和 y 轴上，范围一般在 0~1 之间，这里直接取 xSegment 和 ySegment 作为纹理坐标。随后需要设置纹理坐标指针，如图 2-8 所示。

```
// 设置顶点属性指针
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// 纹理坐标属性
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 * sizeof(float)));
glEnableVertexAttribArray(1);
```

图 2-8 设置纹理坐标指针

其中，glVertexAttribPointer()函数用于设置指针，glEnableVertexAttribArray()函数激活顶点属性数组，使得上面设置的指针生效。加下来是加载纹理，代码如图 2-9 所示。

```
for (int i = 0; i < 3; i++) {
    //加载纹理数据
    glGenTextures(1, &texture_buffer_object1[i]);
    glBindTexture(GL_TEXTURE_2D, texture_buffer_object1[i]);
    //指定纹理的参数
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    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
    unsigned char* data = stbi_load(mapp[i].c_str(), &width, &height, &nrchannels, 0);
    //std::cout << mapp[i].c_str() << ' ' << width  << ' '<< height << ' ' << nrchannels << std::endl;
    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);

        glGenTextures(1, &texture_buffer_object2[i]);
        glBindTexture(GL_TEXTURE_2D, texture_buffer_object2[i]);
        //指定纹理的参数
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        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);
```

图 2-9 加载纹理

首先使用 glGenTextures()函数和 glBindTexture()函数进行纹理绑定，使用 glTexParameteri()函数指定纹理的参数，其中 GL_NEAREST 代表邻近过滤。定义纹理长宽、通道数，使用 stbi_load 函数载入纹理图片，之后使用管理 TexImage2D()函数来生成纹理，注意第三个参数只能是 GL_RGB 而不能是 GL_RGBA，否则会报错（在后文中会详细解释），最后使用 glGenerateMipmap()函数生成 Mipmap 纹理即可。后面将上述步骤重复了一遍是为了为另一个纹理对象加载图像像素数据（因为之后将纹理绑定到球体时，要分别绑定漫反射贴图与镜面反射贴图）。加载完数据后，使用 stbi_image_free()函数释放加载图像的内存数据。

在绘制球体时，需要将加载的纹理与球体绑定，代码如图 2-10 所示。

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture_buffer_object1[0]);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, texture_buffer_object2[0]);
```

图 2-10 球体绑定纹理

glActiveTexture()函数的作用是激活纹理单元，激活后再进行一次绑定，最后还需要使用 glDrawElements 发起绘制调用，用之前设置的纹理、模型矩阵进行渲染。

（4）添加光照

光照系统较为繁琐，首先注意到在第二次作业的代码的 main 函数中进行了着色器的指定，如图 2-11 所示。

```
Shader lightingShader("shader/colors.vs", "shader/colors.fs");  //指定顶点着色器
```

图 2-11 指定着色器

colors.vs（顶点着色器）和 colors.fs（片段着色器）文件对光照的设置做了一系列的约束，这里着重讲一下 colors.fs 文件的设置，重点部分如图 2-12 所示。

```
// 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);
```

图 2-12 colors.fs 文件（部分）

环境光照：使用一个很小的常量（光照）颜色，添加到物体片段的最终颜色中，这样即使场景中没有直接的光源也能看起来存在有一些发散的光；

漫反射光照：漫反射光照使物体上与光线方向越接近的片段嫩从光源处获得更多的亮度。为了测量光线和片段的角度，需要使用法向量（计算请见图 2-7，之后参照图 2-8 的方式设置法向量属性指针）。在顶点着色器中把顶点位置属性乘以模型矩阵来把它变换到世界空间坐标，并在片段着色器中添加相应的输入变量 FragPos。在片段着色器中，计算光源和片段位置之间的方向向量（结果需要标准化），再对 norm 和 lightDir 进行点成，计算光源对当前片段实际的漫反射影响。结果再乘以光的颜色，得到漫反射分量。两个向量之间的角度越大，漫反射分量就会越小；

镜面光照：镜面光照取决于光的方向向量、物体的法向量还有观察方向，因此，观察向量是我们计算镜面光照时需要的一个额外变量，我们可以使用观察者的世界空间位置（直接使用摄像机的位置向量即可）和片段的位置来计算它。之后我们计算出镜面光照强度，用它乘以光源的颜色即可；

光的衰减：光会随着距离进行强度衰减，因此要对上面得到的三种光照进行修正；

完成上述计算后，将三样光照相加，获得片段最后的输出颜色。

总结来说，colors.vs 实现了顶点数据的转换，为片段着色器传递了世界空间

位置、法线、纹理坐标等重要信息，同时利用矩阵变换计算出片段的裁剪空间位置；而 color.fs 实现了基本的光照模型，可用于创建基础 3D 光照效果。

在生成球体前，我们需要进行一些参数配置，如图 2-13 所示。

```
// be sure to activate shader when setting uniforms/drawing objects
// 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);
lightingShader.setVec3("lightPos", lightPos);

//光源属性
/*
一个光源对它的ambient、diffuse和specular光照分量有着不同的强度。
环境光照通常被设置为一个比较低的强度，因为我们不希望环境光颜色太过主导。
光源的漫反射分量通常被设置为我们希望光所具有的那个颜色，通常是一个比较明亮的白色。
镜面光分量通常会保持为vec3(1.0)，以最大强度发光。
*/
lightingShader.setVec3("light.ambient", 1.2f, 1.2f, 1.2f);//环境光强度向量（三维为RGB强度，下同）
lightingShader.setVec3("light.diffuse", 1.0f, 1.0f, 1.0f);  //漫反射强度向量
lightingShader.setVec3("light.specular", 1.0f, 1.0f, 1.0f); //高光（镜面光）强度向量
lightingShader.setFloat("light.constant", 1.0f);
lightingShader.setFloat("light.linear", 0.045f);
lightingShader.setFloat("light.quadratic", 0.0075f);

//反光度
lightingShader.setFloat("shininess", 1.25f);
```

图 2-13 点光源激活并配置参数

然后还需要定义投影变换矩阵和视图变换矩阵（要在定义模型变换矩阵之前进行），如图 2-14、图 2-15 所示。

```
glm::mat4 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);   //定义投影变换矩阵
lightingShader.setMat4("view", view);   //定义视图变换矩阵
```

图 2-14、图 2-15 定义投影变换矩阵和视图变换矩阵

由于投影变换矩阵和视图变换矩阵和我们的 camera 参数直接相关，因此无需多次修改，在生成三个球体时，只需对 model 进行修改即可，它决定了球体的一切运动参数。

（5）实验中遇到的问题

1、加载纹理图片时发生访问冲突，如图 2-16 所示。

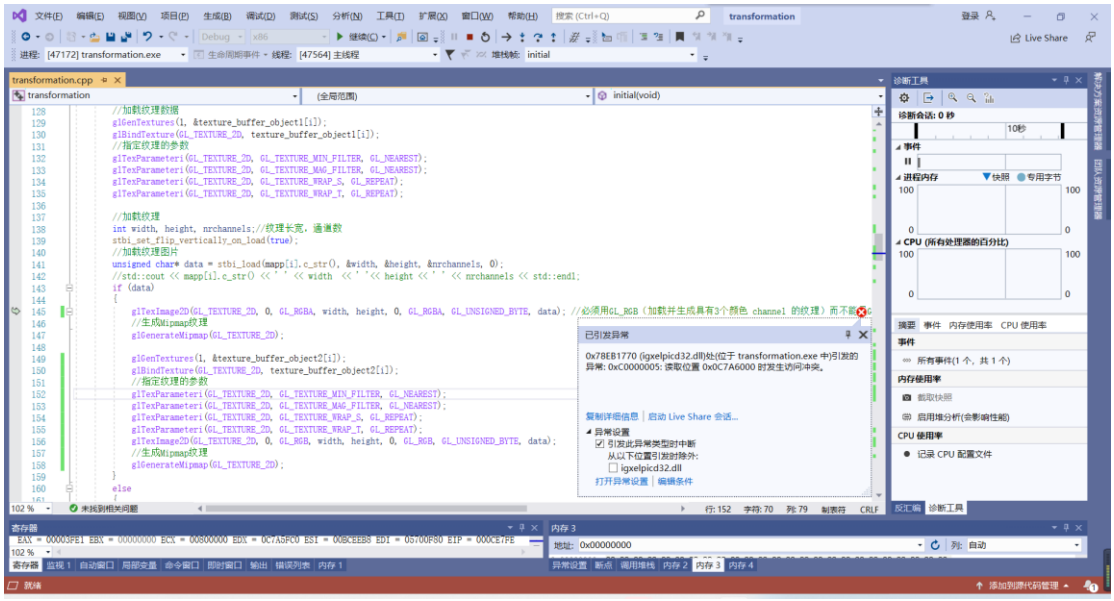


图 2-16 加载纹理图片时发生访问冲突

其原因是因为 glTexImage2D()函数中的参数使用了 GL_RGBA，其指示加载并生成具有 4 个颜色 channel 的纹理，但是通过前面对 nrchannels 参数的输出，可以看出我们的纹理图片只有 3 个颜色 channel。

2、球体的纹理贴图仅有一半，如图 2-17 所示。

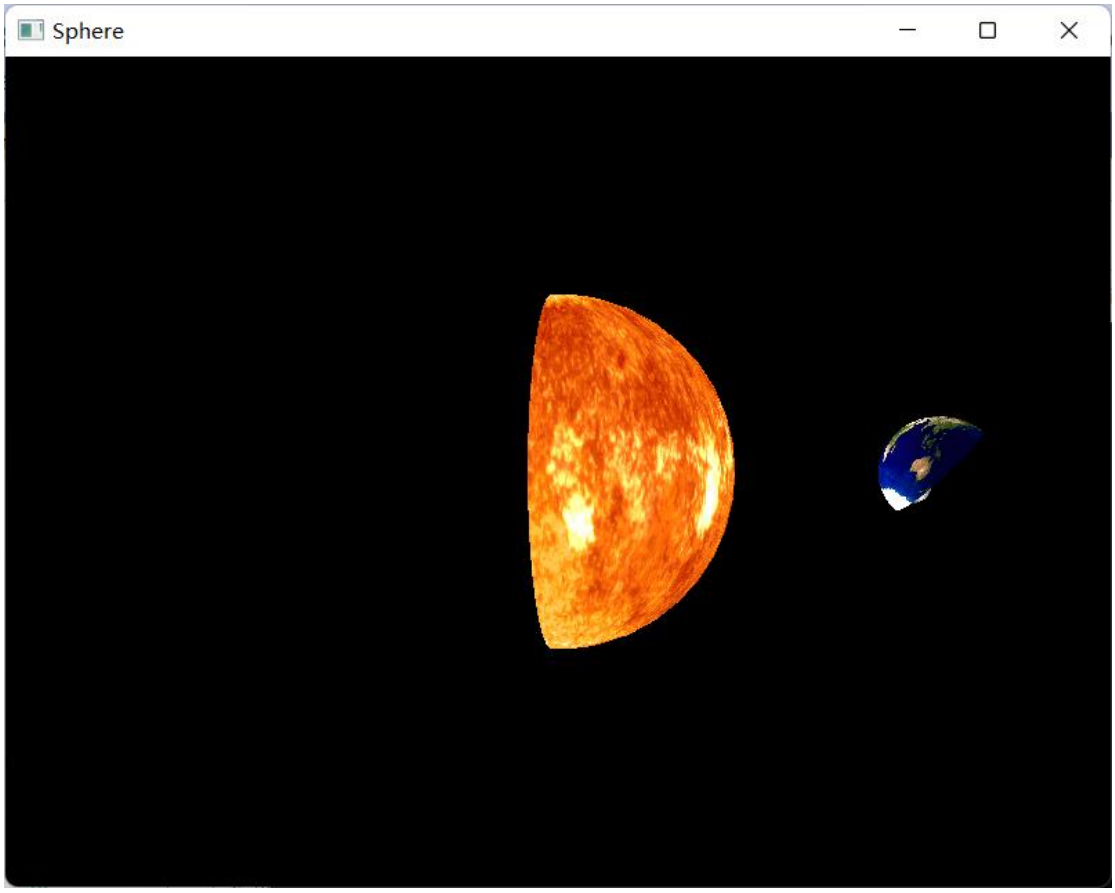


图 2-17 球体贴图异常

原因是在生成球的顶点时，受 Radio 参数的影响，导致球面取点不均匀，而纹理坐标是均匀的，最终导致了这样的结果。（虽然同是由于 Radio 参数造成的贴图异常，但是异常现象与老师发在群里的不同，尚不清楚原因）

3、加入光照后，阴影面不正确，如图 2-18 所示。

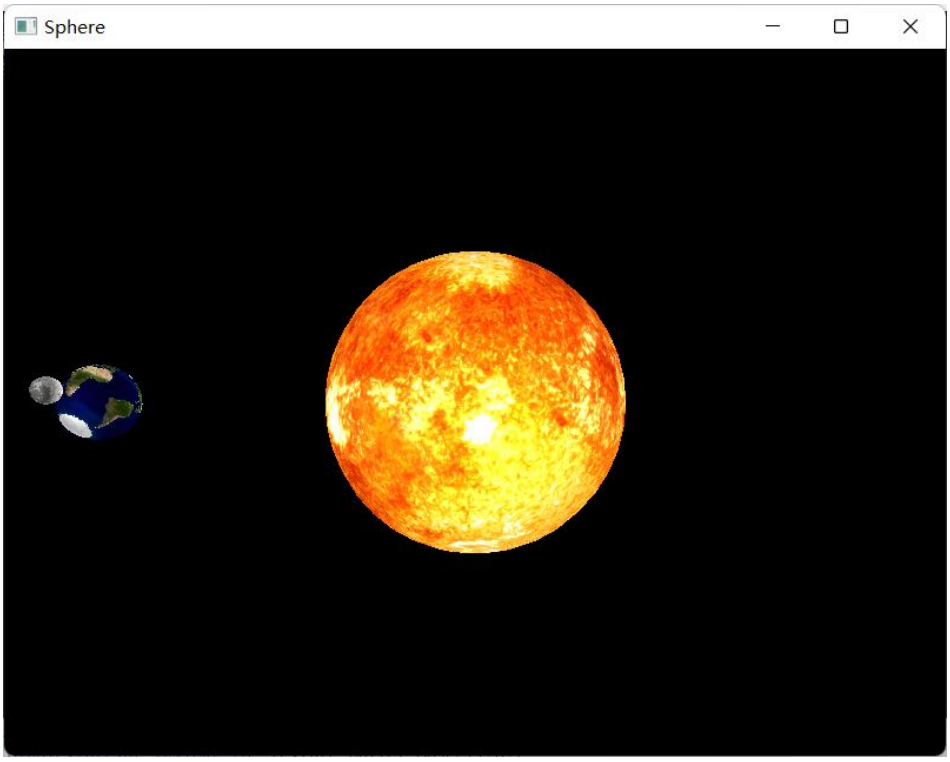
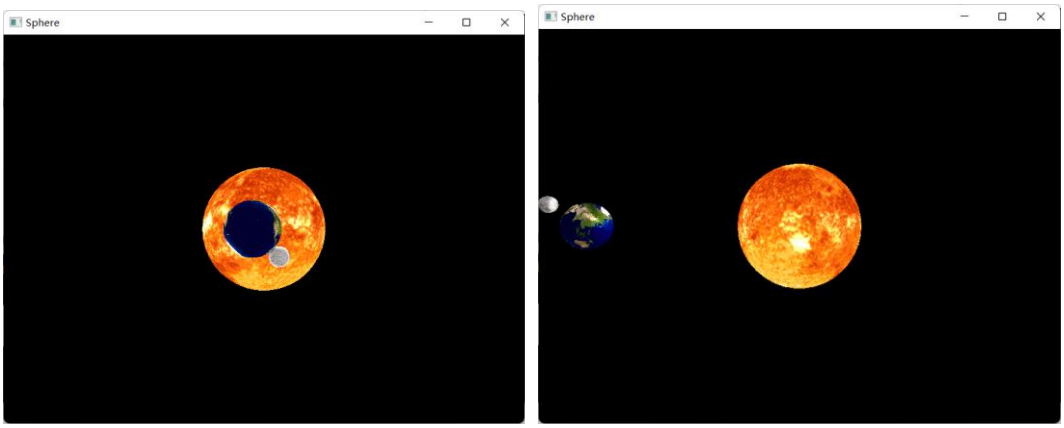


图 2-18 光照阴影面异常

正常情况下，应该是朝向太阳的一面亮，另一面暗，而上图显然不是这样，最后发现是因为没有关闭背面剔除，关闭后即可解决此异常

2.3 实验结果

实验结果如图 2-19 至图 2-22 所示。



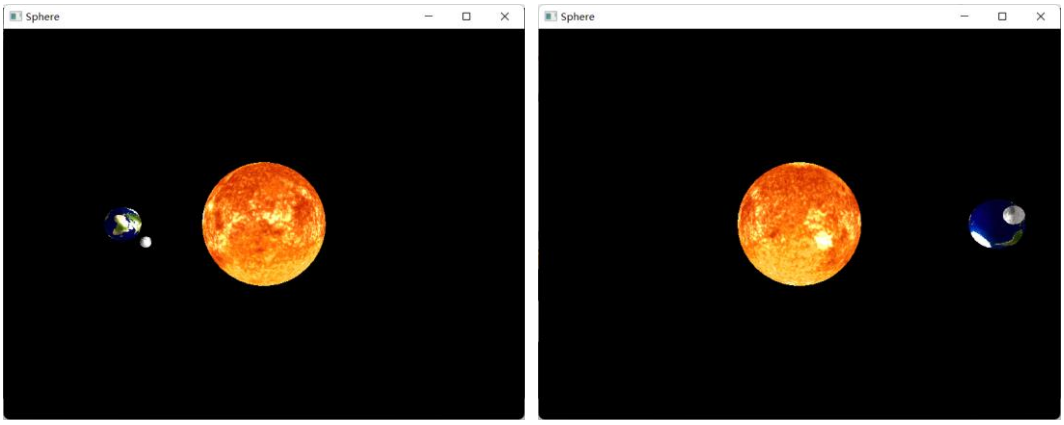


图 2-19 至图 2-22 实验结果

总的来说，实现了以下效果：

1、以太阳为点光源的照明系统，在地球和月球上实现了镜面反射和漫反射，向阳面偏亮，背阳面偏暗，符合实际情况；

2、日地月系统的各球体的公转、自转以及正确的相对速度，月球公转平面与地球公转平面不相同；

3、三个球体纹理正确配布。

2.4 心得体会

这是我第一次接触图形渲染的相关知识，在学习过程中有很多困难（主要是因为不好找资料 or 官方文档导致的），但是一点点坚持下来之后，才能深深地体会到图形学的魅力，也越发觉得图形学这门课程体系之庞大，精通之困难。

我原以为，在已经完成了第二次作业的基础上，加上纹理和光照并不是什么难事，却没想到实操时完全向在学一门新的课程，好不容易找到了一份靠谱的文档，比着上面的纹理和光照教程补全代码后，运行虽然没有报错，但是却出现了各种各样的问题，比如贴图贴了一半、球体平面不光滑（甚至成爆炸状或像陨石一样）等等。非常感谢身边能和我一起讨论的同学们，讨论的过程帮我解决了至少 80%的问题，也是在讨论的过程中，我才逐渐理解了 opengl 的框架，了解了许多函数的使用目的和使用方法。同时也非常感谢何老师在群里分析的代码包还有亲自帮助同学们 debug 的做法，真真正正帮助到了许多同学。

在真正实现了实验要求的那一刻，心中的成就感溢于言表。总的来说，这次实验让我更清晰地了解了 OpenGL 编程的过程，对 OpenGL 框架有了一些基本的认知，尽管难度很大，但也确实激发了我在图形学方面进一步探索的热情，让我获益匪浅。

2.5 源代码

以下是 main.cpp，colors.fs 和 colors.vs 的源代码，头文件不再在这里展示

程序 1：main.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 <map>
#include "shader.h"
#include "camera.h"
#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>

//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;

//旋转参数
static GLfloat xRot = 20.0f;
static GLfloat yRot = 20.0f;

//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[4]; //  纹理对象句柄 1
GLuint texture_buffer_object2[4]; //  纹理对象句柄 2
//int shader_program;//着色器程序句柄

//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;
float a = 6.0f, b = 3.0f;
float r = 1.0f;

//球的数据参数
std::vector<float> sphereVertices;
std::vector<int> sphereIndices;
const int Y_SEGMENTS = 20;
const int X_SEGMENTS = 20;
const float Radio = 2.0;//调整此参数将影响贴图质量，经测试取值 2 最佳
const GLfloat    PI = 3.14159265358979323846f;

std::map<int, std::string> mapp;
//指定光照
glm::vec3 lightPos(0.0f, 0.0f, 0.0f);
```

```
void initial(void)
{
    mapp[0] = "image/earth.jpg";
    mapp[1] = "image/sun.jpg";
    mapp[2] = "image/moon.jpg";
    //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 = std::cos(xSegment * Radio * PI) * std::sin(ySegment * PI);
            float yPos = std::cos(ySegment * PI);
            float zPos = std::sin(xSegment * Radio * PI) * std::sin(ySegment * PI);
            /*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);*/
            glm::vec3 normal = glm::normalize(glm::vec3(xPos, yPos, zPos));

            //位置
            sphereVertices.push_back(xPos);
            sphereVertices.push_back(yPos);
            sphereVertices.push_back(zPos);
            //法向量
            sphereVertices.push_back(normal.x);
            sphereVertices.push_back(normal.y);
            sphereVertices.push_back(normal.z);
            //添加纹理坐标
            sphereVertices.push_back(xSegment);
            sphereVertices.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, sphereVertices.size() * sizeof(float), &sphereVertices[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, 8 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
    // 法线属性
    glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3 * sizeof(float)));//最后一个参数：从 sphereVertices 的第四项（序号 3）开始为法线属性
    glEnableVertexAttribArray(2);
    // 纹理坐标属性
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 * sizeof(float)));//最后一个参数：从 sphereVertices 的第七项（序号 6）开始为纹理属性
    glEnableVertexAttribArray(1);

    for (int i = 0; i < 3; i++) {
        //加载纹理数据
        glGenTextures(1, &texture_buffer_object1[i]);
        glBindTexture(GL_TEXTURE_2D, texture_buffer_object1[i]);
        //指定纹理的参数
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        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
        unsigned char* data = stbi_load(mapp[i].c_str(), &width, &height, &nrchannels, 0);
        //std::cout << mapp[i].c_str() << ' ' << width  << ' '<< height << ' ' << nrchannels << std::endl;
        if (data)
        {
            glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data); //必须用 GL_RGB（加载并生成具有 3 个颜色 channel 的纹理）而不能是 GL_RGBA（加载并生成具有 4 个颜色 channel 的纹理）
            //生成 Mipmap 纹理
            glGenerateMipmap(GL_TEXTURE_2D);

            glGenTextures(1, &texture_buffer_object2[i]);
            glBindTexture(GL_TEXTURE_2D, texture_buffer_object2[i]);
            //指定纹理的参数
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
```

```
GL_NEAREST);
                glTexParameteri(GL_TEXTURE_2D,          GL_TEXTURE_MAG_FILTER,
GL_NEAREST);
                glTexParameteri(GL_TEXTURE_2D,                GL_TEXTURE_WRAP_S,
GL_REPEAT);
                glTexParameteri(GL_TEXTURE_2D,                GL_TEXTURE_WRAP_T,
GL_REPEAT);
                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
        }

        // 解绑 VAO 和 VBO
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glBindVertexArray(0);

        //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_3:
                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_4:
                glDisable(GL_CULL_FACE);        //关闭背面剔除
                break;
        default:
                break;
        }
}

//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);
}

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;
    }

}

void Draw(Shader& lightingShader)
{
    //处理时间
    float currentFrame = (float)glfwGetTime();
    deltaTime = currentFrame - lastFrame;
    lastFrame = currentFrame;

    // 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);

    // be sure to activate shader when setting uniforms/drawing objects
    // 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);
    lightingShader.setVec3("lightPos", lightPos);

    //光源属性
    /*
```

一个光源对它的 ambient、diffuse 和 specular 光照分量有着不同的强度。
环境光照通常被设置为一个比较低的强度，因为我们不希望环境光颜色太过主导。
光源的漫反射分量通常被设置为我们希望光所具有的那个颜色，通常是一个比较明亮的白色。
镜面光分量通常会保持为 vec3(1.0)，以最大强度发光。
*/
lightingShader.setVec3("light.ambient", 1.2f, 1.2f, 1.2f);//环境光强度向量（三维为 RGB 强度，下同）
lightingShader.setVec3("light.diffuse", 1.0f, 1.0f, 1.0f); //漫反射强度向量
lightingShader.setVec3("light.specular", 1.0f, 1.0f, 1.0f);//高光（镜面光）强度向量
lightingShader.setFloat("light.constant", 1.0f);
lightingShader.setFloat("light.linear", 0.045f);
lightingShader.setFloat("light.quadratic", 0.0075f);

//反光度
lightingShader.setFloat("shininess", 1.25f);

//glm::perspective 函数：创建一个透视投影矩阵，用于将 3D 空间中的坐标投影到 2D 平面上，参数一：视野角度（弧度制）；参数二：投影平面的宽高比；参数三四：投影平面到视点的最近和最远距离
glm::mat4 projection = glm::perspective(glm::radians(60.0f), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
glm::mat4 view = camera.GetViewMatrix();
//地球
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture_buffer_object1[0]);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, texture_buffer_object2[0]);

lightingShader.setMat4("projection", projection); //定义投影变换矩阵
lightingShader.setMat4("view", view);   //定义视图变换矩阵
float currentTime = (float)glfwGetTime();

glBindVertexArray(vertex_array_object);
//glm::rotate 函数：用于将物体进行旋转（不是指让物体动态旋转，而是将物体旋转到指定角度获取视图）
//glm::translate 函数：将指定矩阵位移至对应位置
glm::mat4 scale_earth = glm::scale(glm::mat4(1.0f), glm::vec3(0.6, 0.6, 0.6));
glm::mat4 rot_tilt_earth = glm::rotate(glm::mat4(1.0f), glm::radians(135.0f), glm::vec3(0.0f, 0.0f, 1.0f)); //倾斜角度
glm::mat4 trans_revolution_earth = glm::translate(glm::mat4(1.0f), glm::vec3(-a * std::cos(PI / 3 * currentTime), 0.0f, b * std::sin(PI / 3 * currentTime)));   //公转
glm::mat4 rot_self_earth = glm::rotate(glm::mat4(1.0f), -10 * currentTime * glm::radians(45.0f), glm::normalize(glm::vec3(1.0f, 1.0f, 0.0f))); //自转
glm::mat4 model_earth = trans_revolution_earth * rot_self_earth * rot_tilt_earth * scale_earth;

lightingShader.setMat4("model", model_earth);   //定义模型变换矩阵
glDrawElements(GL_TRIANGLES, X_SEGMENTS * Y_SEGMENTS * 6, GL_UNSIGNED_INT, 0);

//太阳
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture_buffer_object1[1]);

```
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, texture_buffer_object2[1]);

        lightingShader.setMat4("projection", projection);
        lightingShader.setMat4("view", view);
        glm::mat4 scale_sun = glm::scale(glm::mat4(1.0f), glm::vec3(1.8, 1.8, 1.8));
        glm::mat4 rot_self_sun = glm::rotate(glm::mat4(1.0f), currentTime * glm::radians(10.0f), glm::vec3(0.0f, 1.0f, 0.0f));
        glm::mat4 model_sun = rot_self_sun * scale_sun;

        lightingShader.setMat4("model", model_sun);
        glDrawElements(GL_TRIANGLES, X_SEGMENTS * Y_SEGMENTS * 6, GL_UNSIGNED_INT, 0);

        //月亮
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, texture_buffer_object1[2]);
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, texture_buffer_object2[2]);

        lightingShader.setMat4("projection", projection);
        lightingShader.setMat4("view", view);

        glBindVertexArray(vertex_array_object);
        glm::mat4 scale_moon = glm::scale(glm::mat4(1.0f), glm::vec3(0.2, 0.2, 0.2));
        glm::mat4 trans_revolution_moon = glm::translate(glm::mat4(1.0f), glm::vec3(r * std::cos(PI * 7 * currentTime), 0.0f, r * std::sin(PI * 7 * currentTime)));
        glm::mat4 rot_tilt_moon = glm::rotate(glm::mat4(1.0f), glm::radians(-45.0f), glm::vec3(0.0f, 0.0f, 1.0f));
        glm::mat4 rot_self_moon = glm::rotate(glm::mat4(1.0f), -365 * currentTime * glm::radians(45.0f), glm::normalize(glm::vec3(1.0f, 1.0f, 0.0f)));
        glm::mat4 model_moon = trans_revolution_earth * rot_tilt_moon * trans_revolution_moon * rot_self_moon * scale_moon;

        lightingShader.setMat4("model", model_moon);
        glDrawElements(GL_TRIANGLES, X_SEGMENTS * Y_SEGMENTS * 6, GL_UNSIGNED_INT, 0);

        glBindVertexArray(0);                                                                                    //解除绑定

}

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;
    }
    printf("按 3 开启背面剔除，按 4 关闭背面剔除（默认为关闭）\n");
    printf("背面剔除状态下，太阳会更亮\n");
    printf("代码 379 至 382 行可以启用摄像机\n\n");
    initial();//初始化
    glDisable(GL_CULL_FACE);        //关闭背面剔除

    //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("shader/colors.vs", "shader/colors.fs");   //指定顶点着色器
    lightingShader.use();
    //lightingShader.setInt("texture1", 0);
    //Shader lightingShader = Shader("shader/colors.vs", "shader/colors.fs");

    while (!glfwWindowShouldClose(window))
    {
        Draw(lightingShader);
        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：colors.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);

}
```

程序 3：colors.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;
}
```