

# 计算机图形学课程报告

计算机科学与技术学院

班 级： CS1701

学 号： U201714501

姓 名： 熊逸钦

指导教师： 何云峰

完成日期： 2020 年 12 月 27 日

## 1 论述

(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：

① 我希望通过选修计算机图形学课程，学习计算机如何对图形进行处理、生成和显示的，了解在计算机系统中是怎么通过 CPU 和 GPU 处理得到图形并最终展现在显示器上的。

② 我对于图形从几何阶段到光栅化阶段再最终展现在显示器上的过程有了一定的了解，同时学习了 OpenGL 的编程技术来绘制自己的图形，对所学的知识较为满意。

③ 这门课作为我在计算机图形学领域的入门启蒙课，使我学到了很多图形学相关的底层和应用层知识，今后如果需要继续涉及图形学项目的话，应该会通过诸如 LearnOpenGL-CN 这样的网站进行高级图形处理技术的学习。

(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：我对图形的变换这部分内容最感兴趣。图形的平移、旋转、缩放、错切等操作要对图形的所有顶点计算它们变换后的新坐标，如果逐个通过模拟的方法去计算显然是不实际的，而且多个变换的叠加也非常复杂。在计算机图形学的课程中巧妙地使用线性代数中的矩阵来做变换，平移、旋转、缩放、错切这些基本操作都可以抽象成一个

矩阵，通过矩阵的乘法来叠加多个变换，最后只需要计算矩阵乘法就可以将多个变换转换为一个变换矩阵。我还学习到矩阵乘法的运算顺序和变换作用到图形的顺序是相反的，在做图形变换的时候需要考虑到变换施加的顺序，以及变换前后参数的取值范围的变化。在实际的 OpenGL 程序中，引用一些数学库来方便矩阵运算（课程实验用的是 `vmath.h`），虽然把一些基本变换做了抽象，但还是需要线性代数的一些底层知识和对矩阵的认识才能正确使用。

（3）你对计算机图形学课程的内容，教学方法有什么看法和建议。

答：希望除了最后的大作业之外还可以增加一些去机房的实验课，在课上提出问题和解决问题的效率或许会高一些，因为很多人（比如我）在截止日期之前都不太有动力做实验。

## 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。

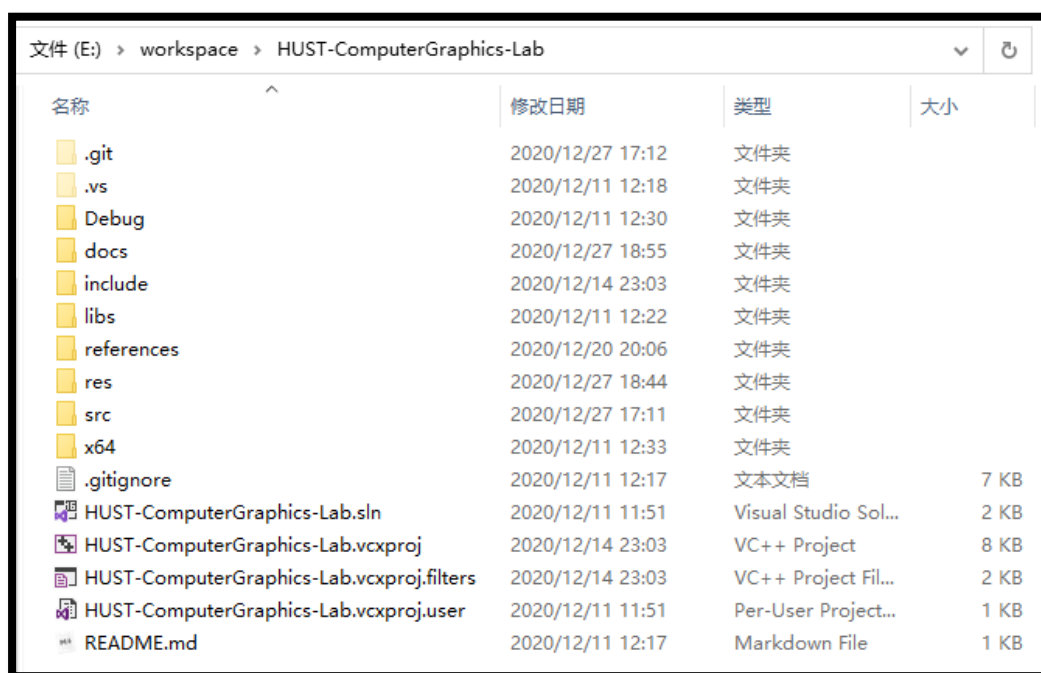
## 2.2 实验方法和过程

### (1) 实验平台搭建

实验中引用与 OpenGL 相关的库有 glfw、glad、vmath 和 stb\_image，其中 glfw 库需要从官网下载源代码后使用 CMake 工具编译，glad 库可在官网进行在线配置后下载得到，vmath 库和 stb\_image 库使用的是老师提供的版本。

搭建过程按照老师提供的详细手册逐步进行，不同的是使用了 Visual Studio 2019 版本，需要在项目设置中将 x86 和 x64 生成方式的引用项目统一按照手册中内容进行修改，否则会报“无法解析的外部符号”错误。

为了上传到 github 方便，我把所需要的库和头文件放在工程根目录下的 libs 和 include 文件夹中，源代码在 src 文件夹中，目录结构如下图所示：



名称	修改日期	类型	大小
.git	2020/12/27 17:12	文件夹	
.vs	2020/12/11 12:18	文件夹	
Debug	2020/12/11 12:30	文件夹	
docs	2020/12/27 18:55	文件夹	
include	2020/12/14 23:03	文件夹	
libs	2020/12/11 12:22	文件夹	
references	2020/12/20 20:06	文件夹	
res	2020/12/27 18:44	文件夹	
src	2020/12/27 17:11	文件夹	
x64	2020/12/11 12:33	文件夹	
.gitignore	2020/12/11 12:17	文本文档	7 KB
HUST-ComputerGraphics-Lab.sln	2020/12/11 11:51	Visual Studio Sol...	2 KB
HUST-ComputerGraphics-Lab.vcxproj	2020/12/14 23:03	VC++ Project	8 KB
HUST-ComputerGraphics-Lab.vcxproj.filters	2020/12/14 23:03	VC++ Project Fil...	2 KB
HUST-ComputerGraphics-Lab.vcxproj.user	2020/12/11 11:51	Per-User Project...	1 KB
README.md	2020/12/11 12:17	Markdown File	1 KB

图 1 工程目录结构

## （2）总体思路

要实现日地月系统，首先要绘制球体。因为太阳、地球和月亮都是球体，所以可以复用一套顶点，之后只需要使用缩放变换来调整大小即可。

然后可以使用旋转变换矩阵、平移变换矩阵的叠加变换来实现自转、公转。

轨道交角的调节可以修改旋转变换矩阵的参数来修改旋转轴的方向。

椭圆轨道可以通过数学公式计算出随角度变化的顶点位置信息，然后直接使用平移变换将球体平移到相应位置即可。

纹理可以将球体的经纬度映射到 2D 纹理顶点，然后创建纹理对象并使用 `stb_image` 库加载图片纹理文件，再修改着色器代码进行渲染即可。

## （3）代码框架

整个框架在之前的课程作业基础上修改而来。先是搭建了一个显示黑框的框架，然后在此基础上绘制了一个三个顶点分别为三原色的三角形，且可以读取键盘按键进行控制。

最后的实验保持 `main` 函数流程基本不变，主要修改 `initial` 函数和 `Draw` 函数来完成实验。

## （4）绘制球体

这部分参考老师给的参考代码中的 `Sphere.cpp`，先将球分为 `x`、`y` 坐标上的 `X_SEGMENTS`、`Y_SEGMENTS` 段，然后通过纬度角和经

度角来计算得到球面上每一点的 x, y, z 三维坐标, 将坐标点数值压入 vector 容器中。

我把这部分代码封装为 generateBallVertices 函数如下:

---

```
void generateBallVertices(std::vector<float>& sphereVertices) {
    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);
            // 球的顶点
            sphereVertices.push_back(xPos);
            sphereVertices.push_back(yPos);
            sphereVertices.push_back(zPos);
        }
    }
}
```

---

得到球的顶点坐标后, 再生成球的顶点索引, 用来将数据绑定至缓冲。我把这部分代码封装为 generateBallIndices 函数如下:

---

```
void generateBallIndices(std::vector<int>& sphereIndices) {
    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);
        }
    }
}
```

---

调用 glGenVertexArrays、glGenBuffers、glBindVertexArray、glBindBuffer 等函数生成并绑定球体的 VAO、VBO、EBO, 对于 VBO 和 EBO 再调用 glBufferData 函数将顶点数据分别绑定至 GL\_ARRAY\_BUFFER 和 GL\_ELEMENT\_ARRAY\_BUFFER 缓冲中。

然后使用 `glVertexAttribPointer` 和 `glEnableVertexAttribArray` 函数设置顶点属性指针。这里需要注意 `glVertexAttribPointer` 函数的 6 个参数，第一个是属性位置值 `location`（这里会在着色器代码中用到），第二个是每个顶点需要读取的数值个数，第三个默认填 `GL_FLOAT`，第四个默认填 `GL_FALSE`，第五个是两个顶点数据之间的偏移量 `offset`，第六个是从数组中读取顶点数据的开始位置。最后 `glEnableVertexAttribArray` 的参数注意和上面的属性位置值 `location` 保持一致。

着色器程序暂时复用示例代码，其他框架代码保持不变。

### （5）处理变换

这里通过变换来绘制太阳、地球和月亮，实现他们的自转和公转。

#### ① 创建观察矩阵、投影矩阵

观察矩阵可以使用 `vmath` 库中的 `vmath::lookat` 函数来实现：

---

```
view = vmath::lookat(vmath::vec3(0.0, 3.0, 0.0), vmath::vec3(0.0, 0.0, -10.0),  
vmath::vec3(0.0, 1.0, 0.0));
```

---

有三个参数，其中的第一个参数是观察者（镜头）的坐标位置；第二个参数是被观察者（日地月系统）的坐标位置；第三个参数是观察者头顶的朝向方向。我这里设置为观察者头顶向上，在(0,0,3)的位置俯视位于(0,0,-10)的日地月系统。

投影矩阵可以使用 `vmath` 库中的 `vmath::lookat` 函数来实现：

---

```
projection = vmath::perspective(fovy, aspect, znear, zfar);
```

---

有四个参数，其中的第一个参数 `fovy` 是眼睛上下睁开的角度值，值越小视野范围越狭小，值越大视野范围越宽阔；第二个参数 `aspect` 表示裁剪面的宽高比，这个影响到视野的截面有多大（这里设置成和

显示区域的宽高比一致即可，比如 800\*600，则设置成 4/3)；第三个参数 `znear` 和第四个参数 `zfar` 分别表示近裁剪面和远裁剪面到眼睛的距离。注意 `znear` 和 `zfar` 不能设置为负值，因为无法看到眼睛后面的东西。我这里设置 `fovy` 为 60，`aspect` 由窗口宽高比决定，`znear` 为 1，`zfar` 为 800。

## ② 绘制太阳

太阳是日地月系统的中心，因此只需要考虑自转然后再将其平移到观察点(0,0,-10)即可。

部分关键代码如下：

---

```
GLfloat angle_sun_self = day * (360.0f / 25.05f); // 自转角

trans *= vmath::translate(0.0f, 0.0f, -10.0f);
vmath::mat4 trans_sun = trans * vmath::rotate(angle_sun_self, vmath::vec3(0.0f, 1.0f, 0.0f)); // 自转
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, trans_sun);
glUniform4fv(colorLoc, 1, vColor[0]);
glDrawElements(GL_TRIANGLES, X_SEGMENTS * Y_SEGMENTS * 6, GL_UNSIGNED_INT, 0); // 绘制三角形
```

---

`vmath` 库提供了 `vmath::translate` 函数来实现平移变换，只需要提供三维方向的平移分量即可。提供了 `vmath::rotate` 函数来实现旋转变换，只需要提供旋转角度和旋转轴方向的单位向量即可（查询资料得知这里遵循的是右手螺旋法则）。

太阳的自转周期是 25.05 天，所以自转角的计算如上述代码。

变换施加的顺序是旋转变换实现自转、平移变换移动到观察位置。由于矩阵相乘的顺序和变换施加顺序相反，因此矩阵先乘以 `vmath::translate` 再乘以 `vmath::rotate`。将变换矩阵和颜色应用于 `Uniform` 之后，最后调用 `glDrawElements` 函数绘制三角面片，画出图



形。

### ③ 绘制地球

由于地球要围绕太阳公转（这里使用椭圆轨道），同时还需要自转，而且大小比太阳要小。所以先进行缩放变换，然后用旋转变换实现自转，再计算从公转角映射到椭圆轨道  $x$ 、 $y$  坐标的函数，并使用平移变换实现绕太阳的公转。

部分关键代码如下：

---

```
float a_earth = 9.0f;    // 椭圆长轴
float b_earth = 3.0f;    // 椭圆短轴
GLfloat angle_earth = day * (360.0f / 365.00f); // 公转角
float x_earth = a_earth * cosf(angle_earth * (float)PI / 180.0f);
float y_earth = b_earth * sinf(angle_earth * (float)PI / 180.0f);

GLfloat angle_earth_self = day * (360.0f / 1.00f); // 自转角

trans *= vmath::translate(-x_earth, 0.0f, y_earth); // 3.公转椭圆轨道
vmath::mat4 trans_earth = trans * vmath::rotate(angle_earth_self, vmath::vec3(0.0f, 1.0f, 0.0f)); // 2.自转
trans_earth *= vmath::scale(0.6f); // 1.缩放
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, trans_earth);
glUniform4fv(colorLoc, 1, vColor[1]);
glDrawElements(GL_TRIANGLES, X_SEGMENTS * Y_SEGMENTS * 6, GL_UNSIGNED_INT, 0);
```

---

地球的自转周期为 1.00 天，公转周期设为 365.00 天，由此可计算出自转角和公转角。

椭圆轨道的  $x$ 、 $y$  坐标可以根据公式  $x = a \cdot \cos \theta$ ， $y = b \cdot \sin \theta$  计算得到。

其余流程和太阳类似，不再赘述。

### ④ 绘制月球

月球围绕地球公转（这里和地球做区别，使用正圆轨道，且轨道和地球轨道有交角），自己也有自转，而且大小比地球还要小。所以

先进行缩放变换，然后用旋转变换实现自转，再使用平移变换确定公转轨道半径，最后使用旋转变换实现绕地球的公转。

部分关键代码如下：

---

```
GLfloat angle_moon = day * (360.0f / (365.00f / 12.00f)); // 公转角
GLfloat angle_moon_self = day * (360.0f / 27.32f); // 自转角

trans *= vmath::rotate(angle_moon, vmath::vec3(sqrtf(2.0) / 2.0f, sqrtf(2.0) / 2.0f, 0.0f));
// 4.倾斜 45 度角公转
trans *= vmath::translate(0.0f, 0.0f, 1.5f); // 3.设置公转半径
vmath::mat4 trans_moon = trans * vmath::rotate(angle_moon_self, vmath::vec3(0.0f, 1.0f, 0.0f)); // 2.自转
trans_moon *= vmath::scale(0.6f * 0.5f); // 1.缩放
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, trans_moon);
glUniform4fv(colorLoc, 1, vColor[2]);
glDrawElements(GL_TRIANGLES, X_SEGMENTS * Y_SEGMENTS * 6, GL_UNSIGNED_INT, 0);
```

---

月球的自转周期为 27.32 天，公转周期设为(365.00/12.00)天，由此可计算出自转角和公转角。

倾斜 45 度角的公转只需要将旋转轴的单位向量设置为( $\sqrt{2.0}/2.0f, \sqrt{2.0}/2.0f, 0.0f$ )，就定义了一个在 xy 平面上与 x 轴成 45 度角的单位方向向量。

其余流程和地球类似，不再赘述。

## （6）添加纹理

这里仿照给四边形添加纹理的方法，给球体添加图片纹理。

### ① 生成纹理顶点

按照经纬度分割法，将球面的点映射到纹理图片上，因为纹理的 x、y 取值范围都是从 0 到 1，和生成球顶点时使用的 xSegment、ySegment 变量取值范围一致，所以可以直接将他们作为纹理顶点加入顶点数组。

---

```
// 纹理顶点，映射到经纬
sphereVertices.push_back(xSegment);
```

---

---

```
sphereVertices.push_back(ySegment);
```

---

## ② 设置顶点属性指针

这里和之前绘制球体不同，因为顶点数组中加入了纹理顶点，所以需要添加一个属性位置来存放纹理顶点的属性。

修改后此处代码如下：

---

```
// 设置顶点属性指针 <ID>, <num>, GL_FLOAT, GL_FALSE, <offset>, <begin>  
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);
```

---

## ③ 创建纹理对象并加载纹理

这里使用 `glGenTextures`、`glBindTexture` 函数来创建纹理对象，再使用 `glTexParameteri` 函数来指定纹理的参数，这部分参考示例代码 `Texture_Image.cpp` 中的内容。

然后调用 `stb_image` 库加载纹理图片，这里使用的是网络上搜索到的一张岩石的图片，如下图 2 所示：

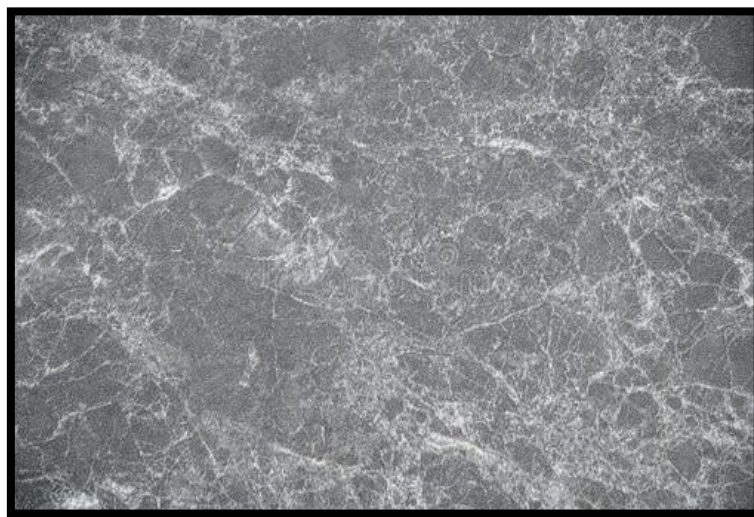


图 2 从网络下载的岩石图片

## ④ 修改并编译着色器程序

编译着色器程序的过程直接参照示例代码即可，这里主要是修改

着色器程序。

修改后的顶点着色器和片段着色器源码如下：

---

```
const char* vertex_shader_source =
"#version 330 core\n"
"layout (location = 0) in vec3 vPos;\n"           // 位置变量的属性位置值为 0
"layout (location = 1) in vec2 vTexture;\n"       // 纹理变量的属性位置值为 1
"out vec4 vColor;\n"                             // 输出 4 维颜色向量
"out vec2 myTexture;\n"                          // 输出 2 维纹理向量
"uniform mat4 transform;\n"
"uniform vec4 color;\n"
"void main()\n"
"{\n"
"    gl_Position = transform * vec4(vPos, 1.0);\n"
"    vColor = color;\n"
"    myTexture = vTexture;\n"
"}\n\0";

const char* fragment_shader_source =
"#version 330 core\n"
"in vec4 vColor;\n"           // 输入的颜色向量
"in vec2 myTexture;\n"       // 输入的纹理向量
"out vec4 FragColor;\n"      // 输出的颜色向量
"uniform sampler2D tex;\n"
"void main()\n"
"{\n"
"    FragColor = texture(tex, myTexture) * vColor;\n" // 顶点颜色和纹理混合
"}\n\0";
```

---

如上述代码，在顶点着色器中的纹理变量的属性位置值和之前设置顶点属性指针时传入 `glVertexAttribPointer` 函数的 `location` 参数值保持一致，添加了一个 2 维纹理向量的输出 `myTexture`。

在片段着色器中，`myTexture` 作为输入，最终输出的颜色向量混合了纹理和顶点颜色。

生成和编译着色器程序的过程详见源代码，此处不再赘述。

## 2.3 实验结果

（1）实现效果说明

- ① 实现了一个日地月运动模型动画；
- ② 地球绕太阳、月球绕地球公转；

- ③ 太阳、地球和月亮均有自转；
- ④ 运动速度基本遵循实际情况，查询了相关资料；
- ⑤ 地球绕太阳，月亮绕地球的轨道有 45 度的交角；
- ⑥ 地球绕太阳使用椭圆轨道，月亮绕地球使用正圆轨道；
- ⑦ 太阳、地球和月球均有纹理贴图，且与顶点颜色混合；

## (2) 实现效果展示

下面的展示图中，红色为太阳，蓝色为地球，灰色为月球。

① 下图 3 展示了太阳、地球和月球的遮挡关系，此时月球离观察者最近，然后是地球，最远的是太阳。

可见运动和遮挡关系正确。

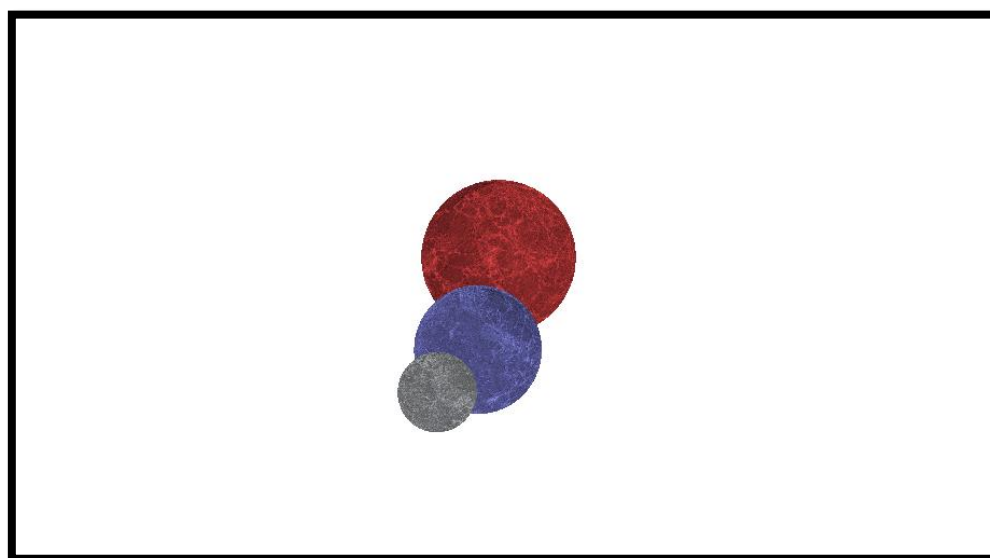


图 3 显示效果展示 1

② 下图 4 展示了地月系统运动到太阳背面后太阳、地球和月球的遮挡关系。此时太阳距离观察者最近，月球刚刚移动到太阳和地球之间，地球距离观察者最远，因此可以看见图中地球的一小部分被月球遮挡，大部分被太阳遮挡。

可见运动和遮挡关系正确。

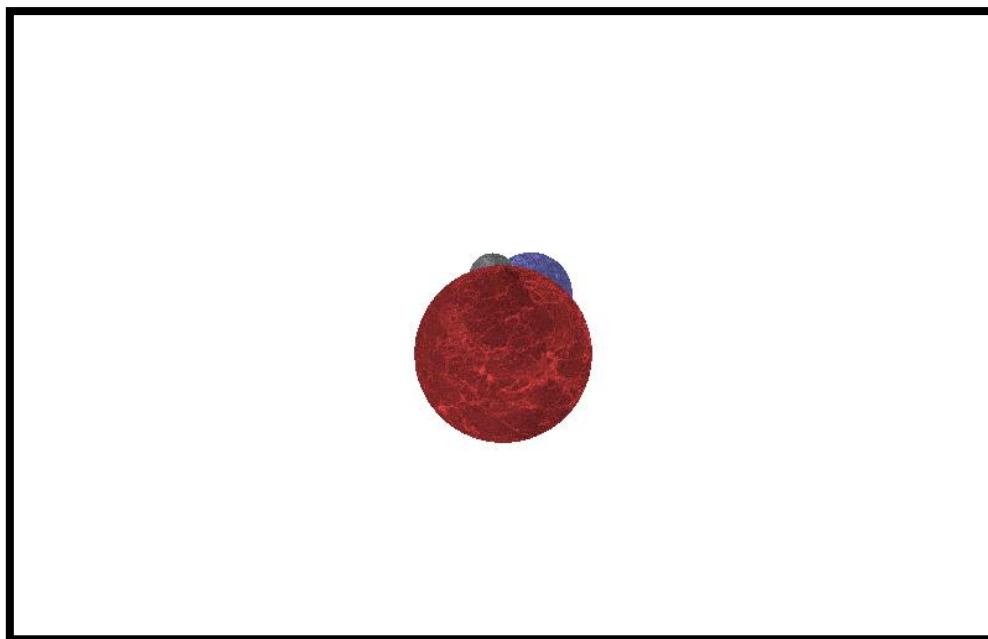


图 4 显示效果展示 2

③ 下图 5 展示了地球运动到远日点附近的状态，可见月球的公转轨道和地球的公转轨道不在同一平面。

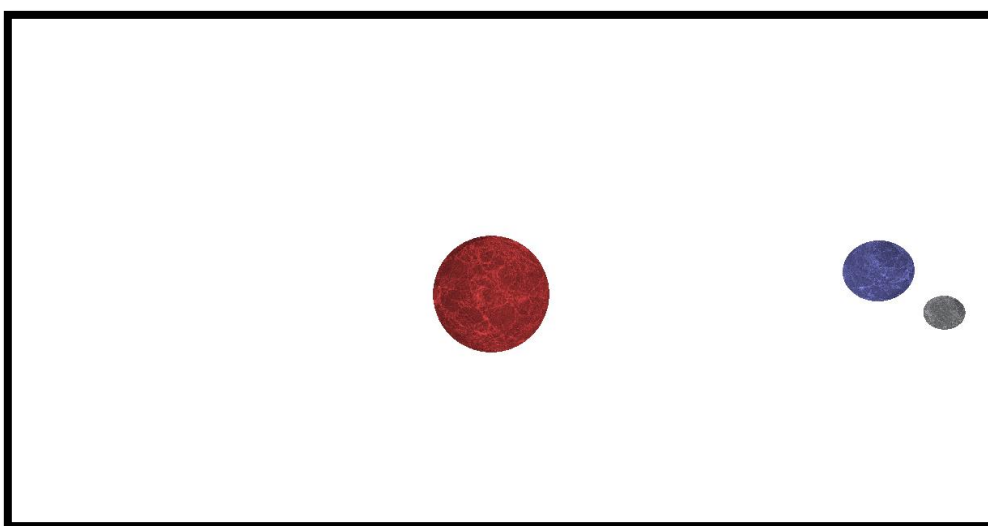


图 5 显示效果展示 3

④ 下图 6 展示了地球运动到近日点附近的状态，可以观察到遮挡关系和运动关系均正确，且能证明地球运行在椭圆轨道上。

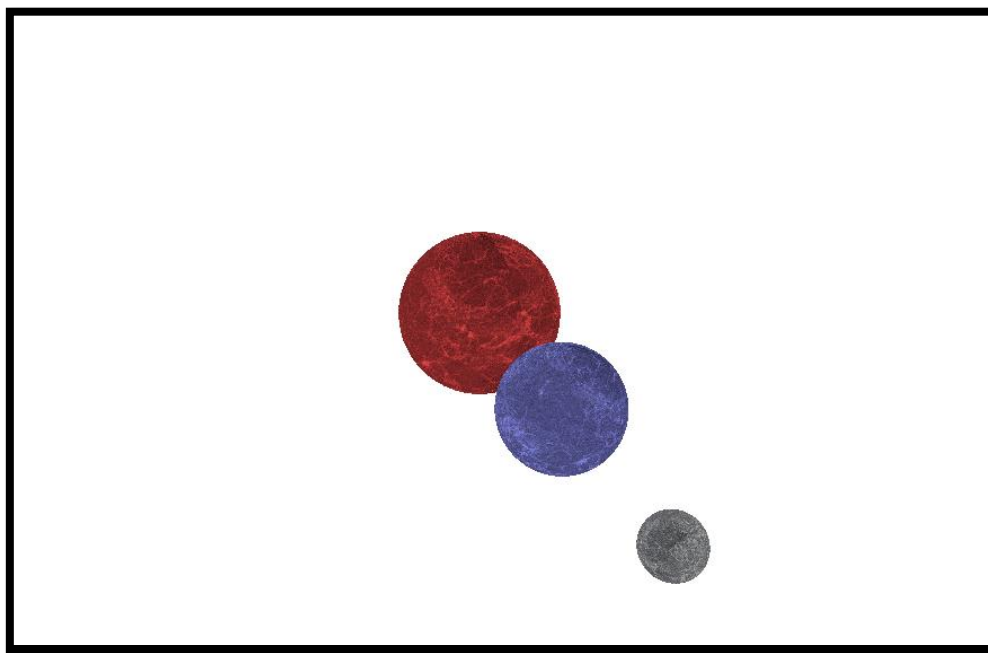


图 6 显示效果展示 4

⑤ 最后展示一个完整过程，录制的 GIF 动图如下图 7 所示：

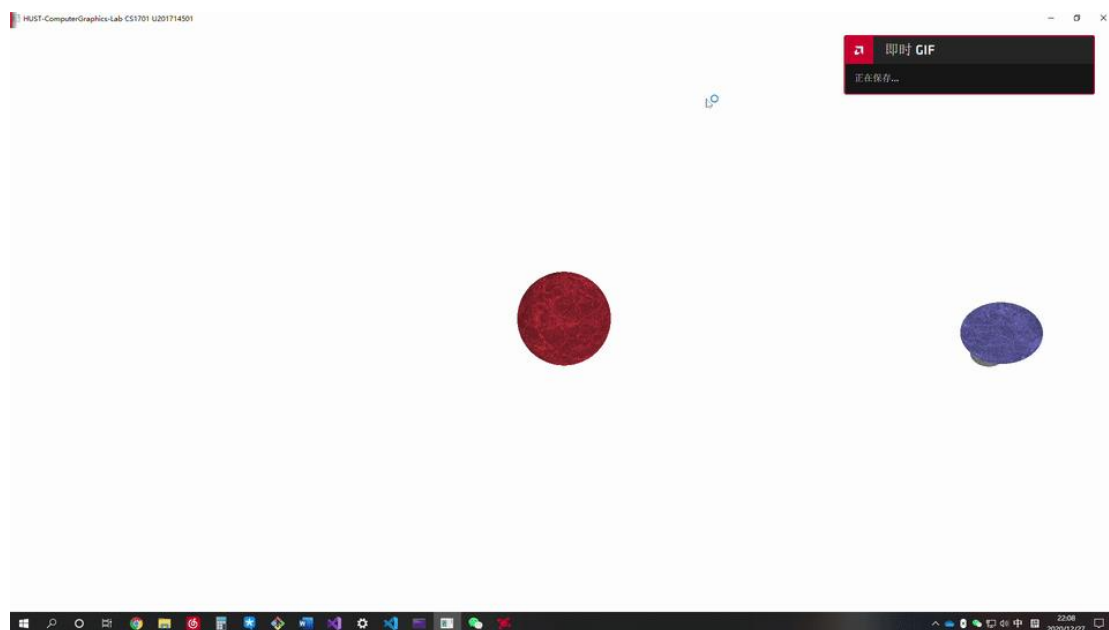


图 7 GIF 动图展示

## 2.4 心得体会

通过课程大作业，我对使用 OpenGL 框架绘制计算机图形有了一定的了解，不仅了解了图形渲染的管线流程，还掌握了图形复杂变换、

纹理贴图等实用技巧。虽然最后由于时间紧迫没能在程序中实现光照，但相关的理论知识也对我十分受用。

学习这门课我最大的感触是数学与计算机科学的结合，引入矩阵的概念之后，能够神奇地将复杂变换糅合成一个变换矩阵，极大简化了思考和计算的复杂度，实现得非常“优雅”。

我觉得计算机图形学这门课更像是一门导论课，虽然以后不一定会从事相关的领域，但计算机图形学是许多计算机学科分支的基础，开阔了我的眼界，非常感谢老师细致的教导。

## 2.5 源代码

(1) 整个工程文件夹放在 GitHub 上：

<https://github.com/YiqinXiong/HUST-ComputerGraphics-Lab>。

(2) main.cpp 源代码如下：

---

### 程序 1: main.cpp

---

```
////////////////////////////////////
//
//  main.cpp
//  HUST-ComputerGraphics-Lab CS1701 U201714501
////////////////////////////////////

#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <iostream>
#include <vmath.h>
#include <vector>
#include <Windows.h>

#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>

// 日地月系统的日期
static GLfloat day = 0.0;

// 窗口尺寸参数
const unsigned int SCR_WIDTH = 1440;
const unsigned int SCR_HEIGHT = 480;

// 投影矩阵参数
```

---



---

```

const float fovy = 60;
float aspect = (float)SCR_WIDTH / (float)SCR_HEIGHT;
const float znear = 1;
const float zfar = 800;

// 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_object;       // 纹理对象句柄
int shader_program;                 // 着色器程序句柄

// 球面顶点数据
std::vector<float> sphereVertices;
std::vector<int> sphereIndices;
const int Y_SEGMENTS = 50;
const int X_SEGMENTS = 50;
const float Radio = 2.0;
const GLfloat PI = 3.14159265358979323846f;

// 生成球的顶点和纹理顶点
void generateBallVertices(std::vector<float>& sphereVertices) {
    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);
            // 球的顶点
            sphereVertices.push_back(xPos);
            sphereVertices.push_back(yPos);
            sphereVertices.push_back(zPos);
            // 纹理顶点, 映射到经纬
            sphereVertices.push_back(xSegment);
            sphereVertices.push_back(ySegment);
        }
    }
}

// 生成球的顶点索引
void generateBallIndices(std::vector<int>& sphereIndices) {
    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);
        }
    }
}

```

---

---

```

    }
}

// 创建纹理对象并加载纹理
void loadTexture() {
    //创建纹理对象
    glGenTextures(1, &texture_buffer_object);
    glBindTexture(GL_TEXTURE_2D, texture_buffer_object);
    //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("res/rock.jpg", &width, &height, &nrchannels, 0);

    if (data)
    {
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);//生成 Mipmap 纹理
    }
    else
        std::cout << "Failed to load texture" << std::endl;
    stbi_image_free(data);//释放资源
}

// 编写并编译着色器程序
void editAndCompileShaderProgram() {
    // 顶点着色器和片段着色器源码
    const char* vertex_shader_source =
        "#version 330 core\n"
        "layout (location = 0) in vec3 vPos;\n" // 位置变量的属性位置值为
0
        "layout (location = 1) in vec2 vTexture;\n" // 纹理变量的属性位置值为
1
        "out vec4 vColor;\n" // 输出 4 维颜色向量
        "out vec2 myTexture;\n" // 输出 2 维纹理向量

        "uniform mat4 transform;\n"
        "uniform vec4 color;\n"
        "void main()\n"
        "{\n"
        "    gl_Position = transform * vec4(vPos, 1.0);\n"
        "    vColor = color;\n"
        "    myTexture = vTexture;\n"
        "}\n\n0";
    const char* fragment_shader_source =
        "#version 330 core\n"
        "in vec4 vColor;\n" // 输入的颜色向量

```

---

---

```

        "in vec2 myTexture;\n"           // 输入的纹理向量
        "out vec4 FragColor;\n"         // 输出的颜色向量
        "uniform sampler2D tex;\n"
        "void main()\n"
        "{\n"
        "    FragColor = texture(tex, myTexture) * vColor;\n" // 顶点颜色和纹理混合
        "}\n\0";

// 生成并编译着色器
// 顶点着色器
int success;
char info_log[512];
int vertex_shader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex_shader, 1, &vertex_shader_source, NULL);
glCompileShader(vertex_shader);
// 检查着色器是否成功编译, 如果编译失败, 打印错误信息
glGetShaderiv(vertex_shader, GL_COMPILE_STATUS, &success);
if (!success)
{
    glGetShaderInfoLog(vertex_shader, 512, NULL, info_log);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" <<
info_log << std::endl;
}
// 片段着色器
int fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragment_shader, 1, &fragment_shader_source, NULL);
glCompileShader(fragment_shader);
// 检查着色器是否成功编译, 如果编译失败, 打印错误信息
glGetShaderiv(fragment_shader, GL_COMPILE_STATUS, &success);
if (!success)
{
    glGetShaderInfoLog(fragment_shader, 512, NULL, info_log);
    std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n"
<< info_log << std::endl;
}
// 链接顶点和片段着色器至一个着色器程序
shader_program = glCreateProgram();
glAttachShader(shader_program, vertex_shader);
glAttachShader(shader_program, fragment_shader);
glLinkProgram(shader_program);
// 检查着色器是否成功链接, 如果链接失败, 打印错误信息
glGetProgramiv(shader_program, GL_LINK_STATUS, &success);
if (!success) {
    glGetProgramInfoLog(shader_program, 512, NULL, info_log);
    std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" <<
info_log << std::endl;
}

// 删除着色器
glDeleteShader(vertex_shader);
glDeleteShader(fragment_shader);

// 使用着色器程序
glUseProgram(shader_program);
}

```

---

---

```
void initial(void)
{
    // 生成球的顶点
    generateBallVertices(sphereVertices);

    // 生成球的顶点索引
    generateBallIndices(sphereIndices);

    // 生成并绑定球体的 VAO 和 VBO
    glGenVertexArrays(1, &vertex_array_object);
    glGenBuffers(1, &vertex_buffer_object);
    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);

    // 生成并绑定 EBO
    glGenBuffers(1, &element_buffer_object);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, element_buffer_object);

    // 将数据绑定至缓冲
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sphereIndices.size() * sizeof(int),
    &sphereIndices[0], GL_STATIC_DRAW);

    // 设置顶点属性指针 <ID>, <num>, GL_FLOAT, GL_FALSE, <offset>, <begin>
    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);

    // 创建纹理对象并加载纹理
    loadTexture();

    // 编写并编译着色器程序
    editAndCompileShaderProgram();

    // 设定点线面的属性
    glPointSize(3); // 设置点的大小
    glLineWidth(1); // 设置线宽

    // opengl 属性
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL); // 指定多边形模式为填充
    glEnable(GL_DEPTH_TEST); // 启用深度测试
}

void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    switch (key)
    {
        case GLFW_KEY_ESCAPE:
            glfwSetWindowShouldClose(window, GL_TRUE); // 关闭窗口
```

---

---

```

        break;
    case GLFW_KEY_1:
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);    // 线框模式
        break;
    case GLFW_KEY_2:
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);    // 填充模式
        break;
    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;
}
}

void Draw(void)
{
    // 清空颜色缓冲
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    unsigned int transformLoc = glGetUniformLocation(shader_program, "transform");
    unsigned int colorLoc = glGetUniformLocation(shader_program, "color");

    // 处理图形的颜色
    GLfloat vColor[3][4] = {
        { 1.0f, 0.3f, 0.3f, 1.0f },
        { 0.6f, 0.6f, 1.0f, 1.0f },
        { 0.8f, 0.8f, 0.8f, 1.0f } };

    // 绑定 VAO
    glBindVertexArray(vertex_array_object);

    vmath::mat4 view, projection, trans;

    // 创建观察矩阵、投影矩阵
    {
        view = vmath::lookat(vmath::vec3(0.0, 3.0, 0.0), vmath::vec3(0.0, 0.0, -10.0),
vmath::vec3(0.0, 1.0, 0.0));
        /*
        * 投影矩阵参数:
        * perspective(Fovy, Aspact, ZNear, ZFar): 投影改变
        * Fovy 是眼睛上下睁开的幅度, 角度值, 值越小, 视野范围越狭小; 值越大, 视
        野范围越宽阔;
        * Aspact 表示裁剪面的宽 w 高 h 比, 这个影响到视野的截面有多大 (这里设置
        成和显示区域的宽高比一致即可, 比如 800*600, 则设置成 4/3);
        * ZNear 表示近裁剪面到眼睛的距离, ZFar 表示远裁剪面到眼睛的距离。注意
        zNear 和 zFar 不能设置为负值 (你怎么看到眼睛后面的东西)。
        */
        projection = vmath::perspective(fovy, aspact, znear, zfar);
        trans = projection * view;
    }
}

```

---

---

```

    }

    // 画太阳
    {
        GLfloat angle_sun_self = day * (360.0f / 25.05f);    // 自转角

        /*
        * translate(x, y, z): 图形平移（相对于显示区域而言）
        * 此处 x, y=0 表示在屏幕中间, z=-10 表示图形在屏幕里面（离摄像机）10 个
        单位距离
        */
        trans *= vmath::translate(0.0f, 0.0f, -10.0f);
        vmath::mat4 trans_sun = trans * vmath::rotate(angle_sun_self, vmath::vec3(0.0f,
1.0f, 0.0f));    // 自转
        glUniformMatrix4fv(transformLoc, 1, GL_FALSE, trans_sun);
        glUniform4fv(colorLoc, 1, vColor[0]);
        glDrawElements(GL_TRIANGLES, X_SEGMENTS * Y_SEGMENTS * 6,
GL_UNSIGNED_INT, 0);    // 绘制三角形
    }

    // 画地球
    {
        float a_earth = 9.0f;    // 椭圆长轴
        float b_earth = 3.0f;    // 椭圆短轴
        GLfloat angle_earth = day * (360.0f / 365.00f);    // 公转角
        float x_earth = a_earth * cosf(angle_earth * (float)PI / 180.0f);
        float y_earth = b_earth * sinf(angle_earth * (float)PI / 180.0f);

        GLfloat angle_earth_self = day * (360.0f / 1.00f);    // 自转角

        trans *= vmath::translate(-x_earth, 0.0f, y_earth);    // 3.公转椭圆轨道
        vmath::mat4 trans_earth = trans * vmath::rotate(angle_earth_self,
vmath::vec3(0.0f, 1.0f, 0.0f));    // 2.自转
        trans_earth *= vmath::scale(0.6f);    // 1.缩放
        glUniformMatrix4fv(transformLoc, 1, GL_FALSE, trans_earth);
        glUniform4fv(colorLoc, 1, vColor[1]);
        glDrawElements(GL_TRIANGLES, X_SEGMENTS * Y_SEGMENTS * 6,
GL_UNSIGNED_INT, 0);
    }

    // 画月球
    {
        GLfloat angle_moon = day * (360.0f / (365.00f / 12.00f));    // 公转角
        GLfloat angle_moon_self = day * (360.0f / 27.32f);    // 自转角

        trans *= vmath::rotate(angle_moon, vmath::vec3(sqrtf(2.0) / 2.0f, sqrtf(2.0) / 2.0f,
0.0f));    // 4.倾斜 45 度角公转
        trans *= vmath::translate(0.0f, 0.0f, 1.5f);    // 3.设置公转半径
        vmath::mat4 trans_moon = trans * vmath::rotate(angle_moon_self,
vmath::vec3(0.0f, 1.0f, 0.0f));    // 2.自转
        trans_moon *= vmath::scale(0.6f * 0.5f);    // 1.缩放
        glUniformMatrix4fv(transformLoc, 1, GL_FALSE, trans_moon);
        glUniform4fv(colorLoc, 1, vColor[2]);
        glDrawElements(GL_TRIANGLES, X_SEGMENTS * Y_SEGMENTS * 6,

```

---

---

```
GL_UNSIGNED_INT, 0);
}

// 解除绑定
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;
    }
}

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,
"HUST-ComputerGraphics-Lab CS1701 U201714501", 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();
}
```

---

---

```
//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);

std::cout << "数字键 1, 2 设置多边形模式为线模式和填充模式。" << std::endl;
std::cout << "数字键 3 打开剔除模式并且剔除多边形的背面。" << std::endl;
std::cout << "数字键 4 关闭剔除模式。" << std::endl;

while (!glfwWindowShouldClose(window))
{
    day+=1.2;
    if (day >= 365)
        day = 0;
    Draw();
    Sleep(33.3);
    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);

//解绑并删除纹理
glBindTexture(GL_TEXTURE_2D, 0);
glDeleteTextures(1, &texture_buffer_object);

glfwDestroyWindow(window);

glfwTerminate();
return 0;
}
```

---