華中科技大學

计算机图形学课程报告

题目:

院　　系	**计算机学院**
专业班级	**大数据 2102 班**
指导教师	**何云峰**
组　　名	OmegaXYZ
姓　　名	**张钧玮**
学　　号	U202115520

2023 年 12 月 10 日

目　　录

1 简答

1.1 问题 1

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。

我选修图形学是想要了解图形学的基本知识，关于图形如何被 CPU 和 GPU 处理，是什么样格式的数据，有哪些处理办法。但是感觉课时太短，内容又太多，显得知识十分杂乱，没有明确的体系和脉络，掌握的不是很好，只能多去 bilibili 大学上网课了。

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。

我对图形的变换最感兴趣。现在我了解了很多旋转和线性代数之间的关系，比如二维的逆时针旋转 90°就相当于矩阵乘以一个矩阵{(0,1),(-1,0)}，三维的旋转也是同理。通过一系列矩阵变换竟然就可以实现复杂的变换，这让我感到十分惊奇。

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。

加课时加学分，不要放到早八。

2 论述

选择 A 实验报告

3 课后作业

选择 A 日地月模型

3.1 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。

3.2 实验方法和过程

继承并且使用第二次作业的代码。实现了球体的绘制（事实上也实现了运动关系）。通过以下步骤逐步完成实验要求。

3.2.1 添加运动关系

运动关系的思路在于正确地描述旋转矩阵。假设太阳位于中心，地球绕太阳旋转，这是矩阵 1；月球绕地球转，这是矩阵 2；月球的位置就在于矩阵 1 跟矩阵 2。值得注意，结果还需要乘以一个旋转矩阵来模拟月球的偏转。下面给出旋转矩阵的关键代码：

```
//地球的旋转矩阵
vmath::mat4 trans2 = vmath::perspective(60, aspact, 1.0f, 500.0f)
*vmath::translate(0.0f, 0.0f, -5.0f)
*vmath::rotate(xRot, vmath::vec3(0.0, 1.0, 0.0))
*vmath::translate(3.0f, 0.0f, 0.0f)
*vmath::scale(0.3f);
//月球的旋转矩阵
vmath::mat4 trans3 = vmath::perspective(60, aspact, 1.0f, 500.0f)
*vmath::translate(0.0f, 0.0f, -5.0f)
*vmath::rotate(xRot, vmath::vec3(0.0, 1.0, 0.0))
*vmath::translate(3.0f, 0.0f, 0.0f)
*vmath::scale(0.3f)
*vmath::rotate(yRot, vmath::vec3(0.0, 0.0, 1.0))
*vmath::translate(3.0f, 0.0f, 0.0f)
*vmath::scale(0.3f);
```

通过以上旋转矩阵就可以轻松确认几个球体对象的位置，着色器附着上去就可以实现正确的运动关系了。

3.2.2 添加纹理

添加纹理的思路在于正确地导入图片，然后绑定纹理对象。这里我使用了 stb_image.h 库文件，修改了着色器的设置，以下代码是关键代码：

```
    //导入处理图片的库文件
    #define STB_IMAGE_IMPLEMENTATION
    #include <stb_image.h>
    ........

    //声明所需的纹理对象的句柄
    GLuint texture_buffer_object_sun; // 太阳纹理对象句柄
```

```
    GLuint texture_buffer_object_earth; // 地球纹理对象句柄
    GLuint texture_buffer_object_moon;  // 月球纹理对象句柄
    int shader_program;       // 着色器程序句柄
    .........
    //对于球,生成顶点的时候需要增加生成纹理顶点
    sphereVertices.push_back(xSegment);
    sphereVertices.push_back(ySegment);

    .........
    // 创建纹理对象并加载纹理
      void loadTexture(GLuint& texture_buffer_object, const char*
filename) {
      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);

      int width, height, nrchannels;
      stbi_set_flip_vertically_on_load(true);
        unsigned char* data = stbi_load(filename, &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);
      }
      else {
      std::cout << "Failed to load texture: " << filename << std::endl;
      }

      glBindTexture(GL_TEXTURE_2D, 0);
      stbi_image_free(data);
    }

    // 创建纹理对象并加载所有纹理
    void loadAllTextures() {
      loadTexture(texture_buffer_object_sun, "sun.jpg");
      loadTexture(texture_buffer_object_earth, "earth.jpg");
      loadTexture(texture_buffer_object_moon, "moon.jpg");
    }

    .........
    //绑定太阳纹理
    glUniform1i(glGetUniformLocation(shader_program, "tex"), 0);
```

```
54
55      glActiveTexture(GL_TEXTURE0);
56      glBindTexture(GL_TEXTURE_2D, texture_buffer_object_sun);
```

3.2.3 添加光源

修改 shader 增加光源相关属性然后增加法线信息输入从而在片段着色器里计算光源信息，以下是修改关键代码：

```
1  // shader
2    // 顶点着色器和片段着色器源码
3    const char* vertex_shader_source =
4      "#version 330 core\n"
5      "layout (location = 0) in vec3 vPos;\n"      // 位置变量的属性位置值为 0
6      "layout (location = 1) in vec2 vTexture;\n"    // 纹理变量的属性位置值为 1
7      "out vec4 vColor;\n"                // 输出 4 维颜色向量
8      "out vec2 myTexture;\n"               // 输出 2 维纹理向量
9      "out vec3 FragPos;\n"
10     "out vec3 Normal;\n"                // 光照
11     "uniform mat4 transform;\n"
12     "uniform vec4 color;\n"
13     "uniform mat4 projection;\n"
14     "void main()\n"
15     "{\n"
16     "    gl_Position = transform * vec4(vPos, 1.0);\n"
17     "    vColor = color;\n"
18     "    myTexture = vTexture;\n"
19     "}\n\0";
20
21   const char* fragment_shader_source =
22     "#version 330 core\n"
23     "in vec4 vColor;\n"      // 输入的颜色向量
24     "in vec3 Normal;\n"      // 输入的法向量
25     "in vec3 FragPos;\n"       // 输入的片段位置向量
26     "in vec2 myTexture;\n"     // 输入的纹理向量
27     "out vec4 FragColor;\n"    // 输出的颜色向量
28     "uniform vec3 lightPos;\n"
29     "uniform vec3 lightColor;\n"
30     "uniform sampler2D tex;\n"
31     "void main()\n"
32     "{\n"
33     "    FragColor = texture(tex, myTexture) * vColor;\n" // 顶点颜色和纹理混合
34     "}\n\0";
35
36 ------------------------------------------------
37
```

```
38    //初始化光源
39    vmath::vec3 lightPos(0.0f, 3.0f, 0.0f);
      glUniform3fv(glGetUniformLocation(shader_program, "lightPos"), 1,
40  &lightPos[0]);
41
42  ------------------------------------------------
43    //计算光照
      glUniform3fv(glGetUniformLocation(shaderProgram, "lightPos"), 1,
44  glm::value_ptr(lightPos));
      glUniform3fv(glGetUniformLocation(shaderProgram, "lightColor"), 1,
45  glm::value_ptr(lightColor));
```

3.3 实验结果

图 1: 日地月模型

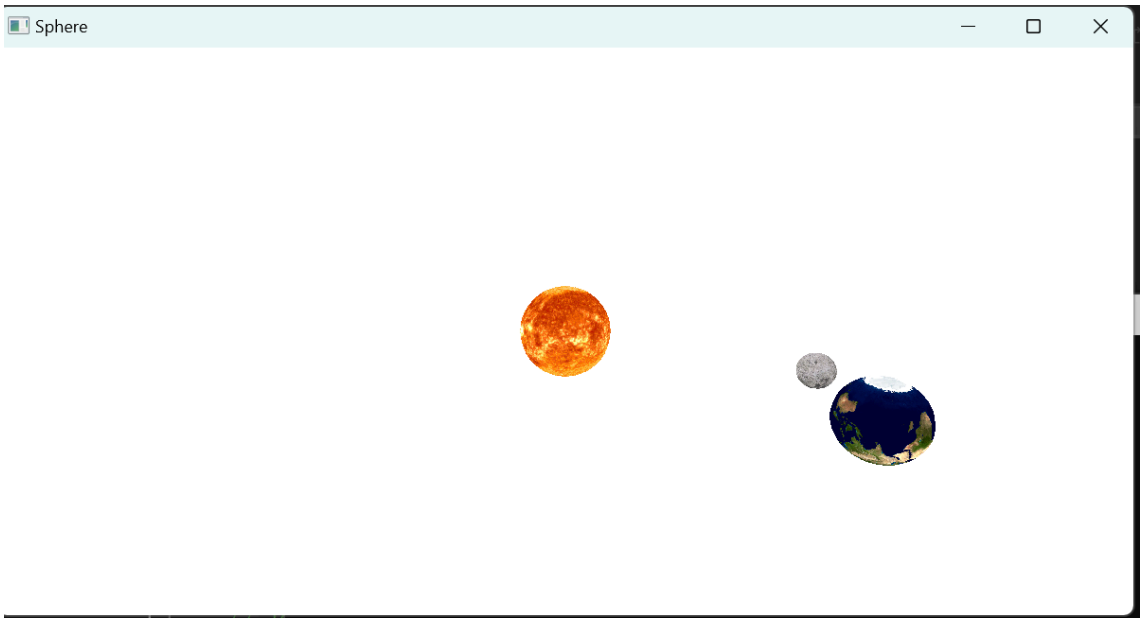

图 2: 日地月模型

运动关系和纹理都实现了，但是光照效果不是很好，具体原因不是很清楚，因为光源确实是存在效果的。

3.4 心得体会

课时太短然后因此实验和平时作业内容显得十分跳跃，很多重要的知识点和概念都没有来得及讲透学透。早八太困。

3.5 源代码

```cpp
////////////////////////////////////////////////////////////////////////////////
//
//  Sphere.cpp
//  1. 球体的绘制（求出球面上所有的点）
//  2. 三角面片的构造
//  3. 利用统一变量进行数据传递
////////////////////////////////////////////////////////////////////////////////


#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>

// 窗口尺寸参数
```

```
21 const unsigned int SCR_WIDTH = 1200;
22 const unsigned int SCR_HEIGHT = 600;
23
24 // 旋转角度
25 static GLfloat aspect = 0.0;
26
27 // 旋转参数
28 const float fovy = 60;
29 float aspact = (float)SCR_WIDTH / (float)SCR_HEIGHT;
30 const float znear = 1;
31 const float zfar = 800;
32
33 // 句柄参数
34 GLuint vertex_array_object;   // VAO 句柄
35 GLuint vertex_buffer_object;  // VBO 句柄
36 GLuint element_buffer_object; // EBO 句柄
37 GLuint texture_buffer_object_sun; // 太阳纹理对象句柄
38 GLuint texture_buffer_object_earth; // 地球纹理对象句柄
39 GLuint texture_buffer_object_moon;  // 月球纹理对象句柄
40 int shader_program;        // 着色器程序句柄
41
42 // 球面顶点数据
43 std::vector<float> sphereVertices;
44 std::vector<int> sphereIndices;
45 const int Y_SEGMENTS = 20;
46 const int X_SEGMENTS = 20;
47 const float Radio = 2.0;
48 const GLfloat  PI = 3.14159265358979323846f;
49
50 // 生成球的顶点和纹理顶点
51 void generateBallVerticles(std::vector<float>& sphereVertices) {
52   for (int y = 0; y <= Y_SEGMENTS; y++)
53   {
54     for (int x = 0; x <= X_SEGMENTS; x++)
55     {
56       float xSegment = (float)x / (float)X_SEGMENTS;
57       float ySegment = (float)y / (float)Y_SEGMENTS;
58      float xPos = std::cos(xSegment * Radio * PI) * std::sin(ySegment
   * PI);
59       float yPos = std::cos(ySegment * PI);
60      float zPos = std::sin(xSegment * Radio * PI) * std::sin(ySegment
   * PI);
61       // 球的顶点
62       sphereVertices.push_back(xPos);
63       sphereVertices.push_back(yPos);
64       sphereVertices.push_back(zPos);
65       sphereVertices.push_back(xSegment);
66       sphereVertices.push_back(ySegment);
67     }
68   }
69 }
70
```

```
71  // 生成球的顶点索引
72  void generateBallIndices(std::vector<int>& sphereIndices) {
73    for (int i = 0; i < Y_SEGMENTS; i++)
74    {
75      for (int j = 0; j < X_SEGMENTS; j++)
76      {
77        sphereIndices.push_back(i * (X_SEGMENTS + 1) + j);
78        sphereIndices.push_back((i + 1) * (X_SEGMENTS + 1) + j);
79        sphereIndices.push_back((i + 1) * (X_SEGMENTS + 1) + j + 1);
80
81        sphereIndices.push_back(i * (X_SEGMENTS + 1) + j);
82        sphereIndices.push_back((i + 1) * (X_SEGMENTS + 1) + j + 1);
83        sphereIndices.push_back(i * (X_SEGMENTS + 1) + j + 1);
84      }
85    }
86  }
87
88  // 创建纹理对象并加载纹理
89  void   loadTexture(GLuint&   texture_buffer_object,   const   char*
    filename) {
90    glGenTextures(1, &texture_buffer_object);
91    glBindTexture(GL_TEXTURE_2D, texture_buffer_object);
92
93    // 指定纹理的参数
94   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
95   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
96    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
97    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
98
99    int width, height, nrchannels;
100   stbi_set_flip_vertically_on_load(true);
101     unsigned  char*  data  =  stbi_load(filename,  &width,  &height,
    &nrchannels, 0);
102
103   if (data) {
104     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
    GL_UNSIGNED_BYTE, data);
105     glGenerateMipmap(GL_TEXTURE_2D);
106   }
107   else {
108     std::cout << "Failed to load texture: " << filename << std::endl;
109   }
110
111   glBindTexture(GL_TEXTURE_2D, 0);
112   stbi_image_free(data);
113 }
114
115 // 创建纹理对象并加载所有纹理
116 void loadAllTextures() {
117   loadTexture(texture_buffer_object_sun, "sun.jpg");
118   loadTexture(texture_buffer_object_earth, "earth.jpg");
119   loadTexture(texture_buffer_object_moon, "moon.jpg");
```

```
}


// 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
    "out vec3 FragPos;\n"
    "out vec3 Normal;\n"               // 光照
    "uniform mat4 transform;\n"
    "uniform vec4 color;\n"
    "uniform mat4 projection;\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 vec3 Normal;\n"     // 输入的法向量
    "in vec3 FragPos;\n"      // 输入的片段位置向量
    "in vec2 myTexture;\n"    // 输入的纹理向量
    "out vec4 FragColor;\n"   // 输出的颜色向量
    "uniform vec3 lightPos;\n"
    "uniform vec3 lightColor;\n"
    "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);
```

```
168   if (!success)
169   {
170     glGetShaderInfoLog(vertex_shader, 512, NULL, info_log);
        std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" <<
171 info_log << std::endl;
172   }
173   // 片段着色器
174   int fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
175  glShaderSource(fragment_shader, 1, &fragment_shader_source, NULL);
176   glCompileShader(fragment_shader);
177   // 检查着色器是否成功编译，如果编译失败，打印错误信息
178   glGetShaderiv(fragment_shader, GL_COMPILE_STATUS, &success);
179   if (!success)
180   {
181     glGetShaderInfoLog(fragment_shader, 512, NULL, info_log);
      std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" <<
182 info_log << std::endl;
183   }
184   // 链接顶点和片段着色器至一个着色器程序
185   shader_program = glCreateProgram();
186   glAttachShader(shader_program, vertex_shader);
187   glAttachShader(shader_program, fragment_shader);
188   glLinkProgram(shader_program);
189   // 检查着色器是否成功链接，如果链接失败，打印错误信息
190   glGetProgramiv(shader_program, GL_LINK_STATUS, &success);
191   if (!success) {
192     glGetProgramInfoLog(shader_program, 512, NULL, info_log);
        std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" <<
193 info_log << std::endl;
194   }
195
196   // 删除着色器
197   glDeleteShader(vertex_shader);
198   glDeleteShader(fragment_shader);
199
200   // 使用着色器程序
201   glUseProgram(shader_program);
202 }
203
204 void initial(void)
205 {
206
207   // 生成球的顶点
208   generateBallVerticles(sphereVertices);
209
210   // 生成球的顶点索引
211   generateBallIndices(sphereIndices);
212
213   //生成太阳光源
214   vmath::vec3 lightPos(0.0f, 3.0f, 0.0f);
```

```
    glUniform3fv(glGetUniformLocation(shader_program, "lightPos"), 1,
215 &lightPos[0]);
216
217   // 生成并绑定球体的 VAO 和 VBO
218   glGenVertexArrays(1, &vertex_array_object);
219   glGenBuffers(1, &vertex_buffer_object);
220   glBindVertexArray(vertex_array_object);
221   glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer_object);
222
223   // 将顶点数据绑定至当前默认的缓冲中
          glBufferData(GL_ARRAY_BUFFER,    sphereVertices.size()    *
224 sizeof(float), &sphereVertices[0], GL_STATIC_DRAW);
225
226   // 生成并绑定 EBO
227   glGenBuffers(1, &element_buffer_object);
228   glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, element_buffer_object);
229
230   // 将数据绑定至缓冲
       glBufferData(GL_ELEMENT_ARRAY_BUFFER,  sphereIndices.size()  *
231 sizeof(int), &sphereIndices[0], GL_STATIC_DRAW);
232
233  // 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),
234 (void*)0);
235   glEnableVertexAttribArray(0);
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(float),
236 (void*)(3 * sizeof(float)));
237   glEnableVertexAttribArray(1);
238
239   // 创建纹理对象并加载纹理
240   loadAllTextures();
241
242   // 编写并编译着色器程序
243   editAndCompileShaderProgram();
244
245   // 设定点线面的属性
246   glPointSize(3); // 设置点的大小
247   glLineWidth(1); // 设置线宽
248
249   // opengl 属性
250   glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);  // 指定多边形模式为填充
251   glEnable(GL_DEPTH_TEST);  // 启用深度测试
252
253 }
254
255 void key_callback(GLFWwindow* window, int key, int scancode, int
action, int mods)
256 {
257   switch (key)
258   {
```

```
259     case GLFW_KEY_ESCAPE:
260       glfwSetWindowShouldClose(window, GL_TRUE);  // 关闭窗口
261       break;
262     case GLFW_KEY_1:
263       glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);  // 线框模式
264       break;
265     case GLFW_KEY_2:
266       glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);  // 填充模式
267       break;
268     case GLFW_KEY_3:
269       glEnable(GL_CULL_FACE);            // 打开背面剔除
270       glCullFace(GL_BACK);             // 剔除多边形的背面
271       break;
272     case GLFW_KEY_4:
273       glDisable(GL_CULL_FACE);           // 关闭背面剔除
274       break;
275     default:
276       break;
277     }
278 }
279 
280 void Draw(void)
281 {
282   // 清空颜色缓冲
283   glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
284   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
285 
286     unsigned int transformLoc = glGetUniformLocation(shader_program,
    "transform");
287       unsigned  int  colorLoc  =  glGetUniformLocation(shader_program,
    "color");
288   // 设置纹理单元的值
289 
290 
291 
292 
293   GLfloat vColor[3][4] = {
294     { 1.0f, 1.0f, 1.0f, 1.0f },
295     { 1.0f, 1.0f, 1.0f, 1.0f },
296     { 1.0f, 1.0f, 1.0f, 1.0f } };
297 
298 
299   // 绑定 VAO
300   glBindVertexArray(vertex_array_object);
301 
302   vmath::mat4 view, projection, trans;
303 
304 
305   {
306     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));
307     projection = vmath::perspective(fovy, aspact, znear, zfar);
```

```
308     trans = projection * view;
309   }
310
311
312   // 画太阳
313   {
314
315     glUniform1i(glGetUniformLocation(shader_program, "tex"), 0);
316
317     glActiveTexture(GL_TEXTURE0);
318     glBindTexture(GL_TEXTURE_2D, texture_buffer_object_sun);
319
320     trans *= vmath::translate(0.0f, 0.0f, -10.0f);
          vmath::mat4  trans_sun  =  trans  *  vmath::rotate(0.0f,
321 vmath::vec3(0.0f, 1.0f, 0.0f));
322     glUniformMatrix4fv(transformLoc, 1, GL_FALSE, trans_sun);
323     glUniform4fv(colorLoc, 1, vColor[0]);
        glDrawElements(GL_TRIANGLES,  X_SEGMENTS  *  Y_SEGMENTS  *  6,
324 GL_UNSIGNED_INT, 0);  // 绘制三角形
325   }
326
327   // 画地球
328   {
329     glUniform1i(glGetUniformLocation(shader_program, "tex"), 1);  //
    地球纹理单元为 1
330     glActiveTexture(GL_TEXTURE1);
331     glBindTexture(GL_TEXTURE_2D, texture_buffer_object_earth);
332     float a_earth = 6.0f;
333     float b_earth = 6.0f;
334     float x_earth = a_earth * cosf(aspect * (float)PI / 180.0f);
335     float y_earth = b_earth * sinf(aspect * (float)PI / 180.0f);
336
337
338
339     trans *= vmath::translate(-x_earth, 0.0f, y_earth);
         vmath::mat4  trans_earth  =  trans  *  vmath::rotate(aspect,
340 vmath::vec3(0.0f, 1.0f, 0.0f));
341     trans_earth *= vmath::scale(0.6f);  // 缩放
342     glUniformMatrix4fv(transformLoc, 1, GL_FALSE, trans_earth);
343     glUniform4fv(colorLoc, 1, vColor[1]);
        glDrawElements(GL_TRIANGLES,  X_SEGMENTS  *  Y_SEGMENTS  *  6,
344 GL_UNSIGNED_INT, 0);
345   }
346
347   // 画月球
348   {
349     glUniform1i(glGetUniformLocation(shader_program, "tex"), 2);  //
    月球纹理单元为 2
350     glActiveTexture(GL_TEXTURE2);
351     glBindTexture(GL_TEXTURE_2D, texture_buffer_object_moon);
```

```
352
353     trans *= vmath::rotate(aspect*12, vmath::vec3(sqrtf(2.0) / 2.0f,
    sqrtf(2.0) / 2.0f, 0.0f));
354     trans *= vmath::translate(0.0f, 0.0f, 1.5f);
355          vmath::mat4 trans_moon = trans * vmath::rotate(0.0f,
    vmath::vec3(0.0f, 1.0f, 0.0f));
356     trans_moon *= vmath::scale(0.6f * 0.5f);  // 缩放
357     glUniformMatrix4fv(transformLoc, 1, GL_FALSE, trans_moon);
358     glUniform4fv(colorLoc, 1, vColor[2]);
359       glDrawElements(GL_TRIANGLES, X_SEGMENTS * Y_SEGMENTS * 6,
    GL_UNSIGNED_INT, 0);
360   }
361
362   // 解除绑定
363   glBindVertexArray(0);
364
365 }
366
367 void reshaper(GLFWwindow* window, int width, int height)
368 {
369   glViewport(0, 0, width, height);
370   if (height == 0)
371   {
372     aspact = (float)width;
373   }
374   else
375   {
376     aspact = (float)width / (float)height;
377   }
378
379 }
380
381 int main()
382 {
383
384   glfwInit(); // 初始化 GLFW
385
386   // OpenGL 版本为 3.3, 主次版本号均设为 3
387   glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
388   glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
389
390   // 使用核心模式(无需向后兼容性)
391   glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
392
393   // 创建窗口(宽、高、窗口名称)
394     GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT,
    "Sphere", NULL, NULL);
395
396   if (window == NULL)
397   {
398     std::cout << "Failed to Create OpenGL Context" << std::endl;
```

```
399       glfwTerminate();
400       return -1;
401     }
402
403     // 将窗口的上下文设置为当前线程的主上下文
404     glfwMakeContextCurrent(window);
405
406     // 初始化 GLAD，加载 OpenGL 函数指针地址的函数
407     if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
408     {
409       std::cout << "Failed to initialize GLAD" << std::endl;
410       return -1;
411     }
412
413     initial();
414
415     //窗口大小改变时调用 reshaper 函数
416     glfwSetFramebufferSizeCallback(window, reshaper);
417
418     //窗口中有键盘操作时调用 key_callback 函数
419     glfwSetKeyCallback(window, key_callback);
420
421     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;
422     std::cout << "数字键 3 打开剔除模式并且剔除多边形的背面。" << std::endl;
423     std::cout << "数字键 4 关闭剔除模式。" << std::endl;
424
425     while (!glfwWindowShouldClose(window))
426     {
427       aspect += 1.2;
428       if (aspect >= 365)
429         aspect = 0;
430       Draw();
431       Sleep(33.3);
432       glfwSwapBuffers(window);
433       glfwPollEvents();
434     }
435
436     // 解绑和删除 VAO 和 VBO
437     glBindVertexArray(0);
438     glBindBuffer(GL_ARRAY_BUFFER, 0);
439     glDeleteVertexArrays(1, &vertex_array_object);
440     glDeleteBuffers(1, &vertex_buffer_object);
441
442     //解绑并删除纹理
443     glBindTexture(GL_TEXTURE_2D, 0);
444     glDeleteTextures(1, &texture_buffer_object_sun);
445
446     glfwDestroyWindow(window);
447
448     glfwTerminate();
449     return 0;
450   }
```