《计算机图形学》实验课程大作业报告

班级：软件 2101 学号：2206213297 姓名：杨豪

班级：软件 001 学号：2201212259 姓名：毕梓博

（以上为封面内容）

一、实验目的

1、了解和掌握 OpenGL 的基本命令。

2、掌握纹理映射以及利用鼠标与系统进行交互。

二、实验内容与要求

地球仪绘制：OpenGL 绘制球体，图片作为纹理映射到整个球面上，双点触控缩放球体，拖动旋转球体。需要有

三、代码

```
// camera.h
#pragma once
#ifndef _CAMERA_H_
#define _CAMERA_H_

#include "glew/glew.h"
#include "glfw/glfw3.h"
#include "glm/glm.hpp"
#include "glm/gtc/matrix_transform.hpp"
#include "glm/gtc/type_ptr.hpp"
#include <vector>
#include <iostream>

using namespace std;
enum Camera_Movement
{
    FORWARD,
    BACKWARD,
    LEFT,
    RIGHT
};

const GLfloat PITCH = 0.0f;
const GLfloat YAW = -90.0f;
const GLfloat ROLL = 0.0f;
const GLfloat SPEED = 36.0f;
const GLfloat ZOOM = 45.0f;
const GLfloat SENSITIVITY = 0.25f;        //缩放的快慢程度
```

```
//一个抽象的camera类，用于处理输入并计算OpenGL中使用的相应Euler角（欧拉角）、向量和矩阵
class Camera
{
public:
    //相机属性
    glm::vec3 Position;
    glm::vec3 Front;
    glm::vec3 Up;
    glm::vec3 Right;
    glm::vec3 WorldUp;
    //欧拉角
    GLfloat Pitch;//俯仰
    GLfloat Yaw;    //偏航
    GLfloat Roll; //翻滚
    //相机选项
    GLfloat MovementSpeed;
    GLfloat MouseSensitivity;//鼠标偏移量系数
    GLfloat Zoom;

    //使用向量构造
    Camera(glm::vec3 position = glm::vec3(0.0f, 0.0f, 3.0f), glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f), GLfloat pitch = PITCH, GLfloat yaw = YAW) : Front(glm::vec3(0.0f, 0.0f, -2.0f)), MovementSpeed(SPEED), MouseSensitivity(SENSITIVITY), Zoom(ZOOM)
    {
        this->Position = position;
        this->WorldUp = up;
        this->Pitch = pitch;
        this->Yaw = yaw;
        this->UpdateCameraVectors();
    }
    //使用标量构造
    Camera(GLfloat posX, GLfloat posY, GLfloat posZ, GLfloat upX, GLfloat upY, GLfloat upZ, GLfloat pitch, GLfloat yaw) : Front(glm::vec3(0.0f, 0.0f, -2.0f)), MovementSpeed(SPEED), MouseSensitivity(SENSITIVITY), Zoom(ZOOM)
    {
        this->Position = glm::vec3(posX, posY, posZ);
        this->WorldUp = glm::vec3(upX, upY, upZ);
        this->Pitch = pitch;
        this->Yaw = yaw;
        this->UpdateCameraVectors();
    }
    //返回使用Eular角和LookAt矩阵计算的视图矩阵
    glm::mat4 GetViewMatrix()
```

```
    {
        return glm::lookAt(this->Position, this->Position + this->Front, this->Up);
    }
    //处理从任何类似键盘的输入系统接收的输入。接受摄像机定义枚举形式的输入参数（从窗口系统中提取）
    void ProcessKeyboard(Camera_Movement direction, GLfloat deltaTime)
    {
        GLfloat velocity = this->MovementSpeed * deltaTime * 0.1;
        if (direction == FORWARD)
        {
            this->Position += this->Front * velocity;
        }
        if (direction == BACKWARD)
        {
            this->Position -= this->Front * velocity;
        }
        if (direction == LEFT)
        {
            this->Position -= this->Right * velocity;
            //this->Position -= glm::cross(this->Front, this->Up) * velocity;
        }
        if (direction == RIGHT)
        {
            this->Position += this->Right * velocity;
            //this->Position += glm::cross(this->Front, this->Up) * velocity;
        }
    }
    //处理从鼠标输入系统接收的输入。在x和y方向都需要偏移值。
    void ProcessMouseMovement(GLfloat xOffset, GLfloat yOffset, GLboolean constrainPitch = true)
    {
        xOffset *= this->MouseSensitivity;
        yOffset *= this->MouseSensitivity;

        this->Yaw += xOffset;   //偏航，左右是X轴
        this->Pitch += yOffset;//俯仰，上下是Y轴

        //确保当俯仰角超出+-90度，屏幕不会翻转
        if (constrainPitch)
        {
            if (this->Pitch > 89.0f)
            {
                this->Pitch = 89.0f;
            }
```

```
            if (this->Pitch < -89.0f)
            {
                this->Pitch = -89.0f;
            }
        }

        //使用更新的Eular角度更新前、右和上向量
        this->UpdateCameraVectors();
    }

    //处理从鼠标滚轮事件接收的输入。只需要输入垂直车轮轴
    void ProcessMouseScroll(GLfloat zOffset)
    {
        if (this->Zoom >= 1.0f && this->Zoom <= 45.0f)
        {
            this->Zoom -= zOffset;
        }
        if (this->Zoom <= 1.0f)
        {
            this->Zoom = 1.0f;
        }
        if (this->Zoom >=45.0f)
        {
            this->Zoom = 45.0f;
        }
    }
protected:
private:
    //根据相机（更新的）Eular角度计算前向量
    void UpdateCameraVectors()
    {
        //计算新的前向量
        glm::vec3 front;
        front.x = cos(glm::radians(this->Yaw)) * cos(glm::radians(this->Pitch));
        front.y = sin(glm::radians(this->Pitch));
        front.z = sin(glm::radians(this->Yaw)) * cos(glm::radians(this->Pitch));
        this->Front = glm::normalize(front);//attention
        //同时重新计算右上向量
        //规范化向量，因为向量的长度越接近0，向上或向下查找的次数越多，移动速度
就越慢。
        this->Right = glm::normalize(glm::cross(this->Front, this->WorldUp));
        this->Up = glm::normalize(glm::cross(this->Right, this->Front));   //cross如果参数交
换，方向相反
    }
```

```
};
#endif
```

// **sphere.h**

```
#pragma once

#include "glm/glm.hpp"
#include <cmath>
#include <vector>

using namespace std;

class Sphere
{
private:
    int _numVertices;   //顶点总数
    int _numIndices;    //顶点索引总数

    vector<int> _indices;
    vector<glm::vec3> _vertices;  //顶点向量数
    vector<glm::vec2> _texCoords;
    vector<glm::vec3> _normals;
    vector<glm::vec3> _tangents;  //切片数量

    void init(int prec);
    float toRadians(float degree);

public:
    Sphere();
    Sphere(int prec);

    int getNumVertices();
    int getNumIndices();

    vector<int> getIndices();
    vector<glm::vec3> getVertices();
    vector<glm::vec2> getTexCoords();
    vector<glm::vec3> getNormals();
    vector<glm::vec3> getTangents();

};
```

// **utils.h**

```
#include <glew\glew.h>
#include <GLFW\glfw3.h>
#include <SOIL2\soil2.h>
#include <string>
#include <iostream>
#include <fstream>
#include <cmath>
#include <vector>
#include <glm\glm.hpp>
#include <glm\gtc\type_ptr.hpp>
#include <glm\gtc\matrix_transform.hpp>

class Utils
{
private:
    static std::string readShaderFile(const char *filePath);
    static void printShaderLog(GLuint shader);
    static void printProgramLog(int prog);
    static GLuint prepareShader(int shaderTYPE, const char *shaderPath);
    static int finalizeShaderProgram(GLuint sprogram);

public:
    Utils();
    static bool checkOpenGLError();
    static GLuint createShaderProgram(const char *vp, const char *fp);
    static GLuint createShaderProgram(const char *vp, const char *gp, const char *fp);
    static GLuint createShaderProgram(const char *vp, const char *tCS, const char* tES,
const char *fp);
    static GLuint createShaderProgram(const char *vp, const char *tCS, const char* tES,
char *gp, const char *fp);
    static GLuint loadTexture(const char *texImagePath);
    static GLuint loadCubeMap(const char *mapDir);

    static float* goldAmbient();
    static float* goldDiffuse();
    static float* goldSpecular();
    static float goldShininess();

    static float* silverAmbient();
    static float* silverDiffuse();
    static float* silverSpecular();
    static float silverShininess();

    static float* bronzeAmbient();
```

```
    static float* bronzeDiffuse();
    static float* bronzeSpecular();
    static float bronzeShininess();
};
```

// utils.cpp

```
#include <glew\glew.h>
#include <GLFW\glfw3.h>
#include <SOIL2\soil2.h>
#include <string>
#include <iostream>
#include <fstream>
#include <cmath>
#include <glm\glm.hpp>
#include <glm\gtc\type_ptr.hpp> // glm::value_ptr
#include <glm\gtc\matrix_transform.hpp> // glm::translate, glm::rotate, glm::scale,
glm::perspective
#include "Utils.h"
using namespace std;

Utils::Utils() {}

string Utils::readShaderFile(const char *filePath) {
    string content;
    ifstream fileStream(filePath, ios::in);
    string line = "";
    while (!fileStream.eof()) {
        getline(fileStream, line);
        content.append(line + "\n");
    }
    fileStream.close();
    return content;
}

bool Utils::checkOpenGLError() {
    bool foundError = false;
    int glErr = glGetError();
    while (glErr != GL_NO_ERROR) {
        cout << "glError: " << glErr << endl;
        foundError = true;
        glErr = glGetError();
    }
    return foundError;
```

```
}

void Utils::printShaderLog(GLuint shader) {
    int len = 0;
    int chWrittn = 0;
    char *log;
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &len);
    if (len > 0) {
        log = (char *)malloc(len);
        glGetShaderInfoLog(shader, len, &chWrittn, log);
        cout << "Shader Info Log: " << log << endl;
        free(log);
    }
}

GLuint Utils::prepareShader(int shaderTYPE, const char *shaderPath)
{   GLint shaderCompiled;
    string shaderStr = readShaderFile(shaderPath);
    const char *shaderSrc = shaderStr.c_str();
    GLuint shaderRef = glCreateShader(shaderTYPE);
    glShaderSource(shaderRef, 1, &shaderSrc, NULL);
    glCompileShader(shaderRef);
    checkOpenGLError();
    glGetShaderiv(shaderRef, GL_COMPILE_STATUS, &shaderCompiled);
    if (shaderCompiled != 1)
    {   if (shaderTYPE == 35633) cout << "Vertex ";
        if (shaderTYPE == 36488) cout << "Tess Control ";
        if (shaderTYPE == 36487) cout << "Tess Eval ";
        if (shaderTYPE == 36313) cout << "Geometry ";
        if (shaderTYPE == 35632) cout << "Fragment ";
        cout << "shader compilation error." << endl;
        printShaderLog(shaderRef);
    }
    return shaderRef;
}

void Utils::printProgramLog(int prog) {
    int len = 0;
    int chWrittn = 0;
    char *log;
    glGetProgramiv(prog, GL_INFO_LOG_LENGTH, &len);
    if (len > 0) {
        log = (char *)malloc(len);
        glGetProgramInfoLog(prog, len, &chWrittn, log);
```

```
        cout << "Program Info Log: " << log << endl;
        free(log);
    }
}

int Utils::finalizeShaderProgram(GLuint sprogram)
{   GLint linked;
    glLinkProgram(sprogram);
    checkOpenGLError();
    glGetProgramiv(sprogram, GL_LINK_STATUS, &linked);
    if (linked != 1)
    {   cout << "linking failed" << endl;
        printProgramLog(sprogram);
    }
    return sprogram;
}

GLuint Utils::createShaderProgram(const char *vp, const char *fp) {
    GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
    GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
    GLuint vfprogram = glCreateProgram();
    glAttachShader(vfprogram, vShader);
    glAttachShader(vfprogram, fShader);
    finalizeShaderProgram(vfprogram);
    return vfprogram;
}

GLuint Utils::createShaderProgram(const char *vp, const char *gp, const char *fp) {
    GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
    GLuint gShader = prepareShader(GL_GEOMETRY_SHADER, gp);
    GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
    GLuint vgfprogram = glCreateProgram();
    glAttachShader(vgfprogram, vShader);
    glAttachShader(vgfprogram, gShader);
    glAttachShader(vgfprogram, fShader);
    finalizeShaderProgram(vgfprogram);
    return vgfprogram;
}

GLuint Utils::createShaderProgram(const char *vp, const char *tCS, const char* tES,
const char *fp) {
    GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
    GLuint tcShader = prepareShader(GL_TESS_CONTROL_SHADER, tCS);
    GLuint teShader = prepareShader(GL_TESS_EVALUATION_SHADER, tES);
```

```
    GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
    GLuint vtfprogram = glCreateProgram();
    glAttachShader(vtfprogram, vShader);
    glAttachShader(vtfprogram, tcShader);
    glAttachShader(vtfprogram, teShader);
    glAttachShader(vtfprogram, fShader);
    finalizeShaderProgram(vtfprogram);
    return vtfprogram;
}

GLuint Utils::createShaderProgram(const char *vp, const char *tCS, const char* tES,
char *gp, const char *fp) {
    GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
    GLuint tcShader = prepareShader(GL_TESS_CONTROL_SHADER, tCS);
    GLuint teShader = prepareShader(GL_TESS_EVALUATION_SHADER, tES);
    GLuint gShader = prepareShader(GL_GEOMETRY_SHADER, gp);
    GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
    GLuint vtgfprogram = glCreateProgram();
    glAttachShader(vtgfprogram, vShader);
    glAttachShader(vtgfprogram, tcShader);
    glAttachShader(vtgfprogram, teShader);
    glAttachShader(vtgfprogram, gShader);
    glAttachShader(vtgfprogram, fShader);
    finalizeShaderProgram(vtgfprogram);
    return vtgfprogram;
}

GLuint Utils::loadCubeMap(const char *mapDir) {
    GLuint textureRef;
    string xp = mapDir; xp = xp + "/xp.jpg";
    string xn = mapDir; xn = xn + "/xn.jpg";
    string yp = mapDir; yp = yp + "/yp.jpg";
    string yn = mapDir; yn = yn + "/yn.jpg";
    string zp = mapDir; zp = zp + "/zp.jpg";
    string zn = mapDir; zn = zn + "/zn.jpg";
    textureRef = SOIL_load_OGL_cubemap(xp.c_str(), xn.c_str(), yp.c_str(), yn.c_str(),
zp.c_str(), zn.c_str(),
        SOIL_LOAD_AUTO, SOIL_CREATE_NEW_ID, SOIL_FLAG_MIPMAPS);
    if (textureRef == 0) cout << "didnt find cube map image file" << endl;
    return textureRef;
}

GLuint Utils::loadTexture(const char *texImagePath)
{   GLuint textureRef;
```

```
    textureRef = SOIL_load_OGL_texture(texImagePath, SOIL_LOAD_AUTO,
SOIL_CREATE_NEW_ID, SOIL_FLAG_INVERT_Y);
    if (textureRef == 0) cout << "didnt find texture file " << texImagePath << endl;
    // ----- mipmap/anisotropic section
    glBindTexture(GL_TEXTURE_2D, textureRef);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    glGenerateMipmap(GL_TEXTURE_2D);
    if (glewIsSupported("GL_EXT_texture_filter_anisotropic")) {
        GLfloat anisoset = 0.0f;
        glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &anisoset);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, anisoset);
    }
    // ----- end of mipmap/anisotropic section
    return textureRef;
}

// GOLD material - ambient, diffuse, specular, and shininess
float* Utils::goldAmbient() { static float a[4] = { 0.2473f, 0.1995f, 0.0745f, 1 };
return (float*)a; }
float* Utils::goldDiffuse() { static float a[4] = { 0.7516f, 0.6065f, 0.2265f, 1 };
return (float*)a; }
float* Utils::goldSpecular() { static float a[4] = { 0.6283f, 0.5559f, 0.3661f, 1 };
return (float*)a; }
float Utils::goldShininess() { return 51.2f; }

// SILVER material - ambient, diffuse, specular, and shininess
float* Utils::silverAmbient() { static float a[4] = { 0.1923f, 0.1923f, 0.1923f, 1 };
return (float*)a; }
float* Utils::silverDiffuse() { static float a[4] = { 0.5075f, 0.5075f, 0.5075f, 1 };
return (float*)a; }
float* Utils::silverSpecular() { static float a[4] = { 0.5083f, 0.5083f, 0.5083f, 1 };
return (float*)a; }
float Utils::silverShininess() { return 51.2f; }

// BRONZE material - ambient, diffuse, specular, and shininess
float* Utils::bronzeAmbient() { static float a[4] = { 0.2125f, 0.1275f, 0.0540f, 1 };
return (float*)a; }
float* Utils::bronzeDiffuse() { static float a[4] = { 0.7140f, 0.4284f, 0.1814f, 1 };
return (float*)a; }
float* Utils::bronzeSpecular() { static float a[4] = { 0.3936f, 0.2719f, 0.1667f, 1 };
return (float*)a; }
float Utils::bronzeShininess() { return 25.6f; }
```

```
// sphere.cpp
#include "Sphere.h"

static const float pai = 3.1415926f;

Sphere::Sphere()
{
    init(48);
}

Sphere::Sphere(int prec)
{
    init(prec);
}

void Sphere::init(int prec)  //prec:表示精度，即一个球体被切成prec片
{
    _numVertices = (prec + 1) * (prec + 1);   //顶点总数
    _numIndices = prec * prec * 6;  //每个顶点周围被6个三角形包围，索引总数为：prec * prec * 6

    for (int i=0; i<_numVertices; i++)
    {
        _vertices.push_back(glm::vec3());
        _texCoords.push_back(glm::vec2());
        _normals.push_back(glm::vec3());
        _tangents.push_back(glm::vec3());
    }

    for (int i=0; i<_numIndices; i++)
    {
        _indices.push_back(0);
    }

    // calculate triangle vertices
    for (int i=0; i<=prec; i++)
    {
        for (int j=0; j<=prec; j++)
        {
            float y = (float)(glm::cos(toRadians(180.f - i * 180.f / prec)));
            float x = (float)(glm::cos(toRadians(j * 360.f / prec)) * ((float)(glm::abs(glm::cos(glm::asin(y))))));
            float z = (float)(glm::sin(toRadians(j * 360.f / (float)(prec))) *
```

```
(float)(glm::abs(glm::cos(glm::asin(y)))));
            _vertices[i * (prec + 1) + j] = glm::vec3(x, y, z);
            _texCoords[i * (prec + 1) + j] = glm::vec2(((float)(j) / prec),
((float)(i) / prec));
            _normals[i * (prec + 1) + j] = glm::vec3(x, y, z);

            // calculate tangent vector
            if (((0 == x) && (1 == y) && (0 == z)) || ((0 == x) && (-1 == y) && (0 ==
z) ))
            {
                _tangents[i * (prec + 1) + j] = glm::vec3(0.f, 0.f, -1.f);
            }
            else
            {
                _tangents[i * (prec + 1) + j] = glm::cross(glm::vec3(0.f, 1.f, 0.f),
glm::vec3(x, y, z));
            }
        }
    }

    // calculate triangle indices
    for (int i=0; i<prec; i++)
    {
        for (int j=0; j<prec; j++)
        {
            _indices[6 * (i * prec + j) + 0] = i * (prec + 1) + j;
            _indices[6 * (i * prec + j) + 1] = i * (prec + 1) + j + 1;
            _indices[6 * (i * prec + j) + 2] = (i + 1) * (prec + 1) + j;
            _indices[6 * (i * prec + j) + 3] = i * (prec + 1) + j + 1;
            _indices[6 * (i * prec + j) + 4] = (i + 1) * (prec + 1) + j + 1;
            _indices[6 * (i * prec + j) + 5] = (i + 1) * (prec + 1)+ j;
        }
    }
}

float Sphere::toRadians(float degree)
{
    return (degree * 2.f * pai) / 360.f;
}



int Sphere::getNumVertices()
{
```

```
    return _numVertices;
}

int Sphere::getNumIndices()
{
    return _numIndices;
}

std::vector<int> Sphere::getIndices()
{
    return _indices;
}

std::vector<glm::vec3> Sphere::getVertices()
{
    return _vertices;
}

std::vector<glm::vec2> Sphere::getTexCoords()
{
    return _texCoords;
}

std::vector<glm::vec3> Sphere::getNormals()
{
    return _normals;
}

std::vector<glm::vec3> Sphere::getTangents()
{
    return _tangents;
}
```

```
// main.cpp
#include "glew/glew.h"
#include "glfw/glfw3.h"
#include "glm/glm.hpp"
#include "glm/gtc/matrix_transform.hpp"
#include "glm/gtc/type_ptr.hpp"
#include "Sphere.h"
```

```
#include "Utils.h"
#include "camera.h"
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

static const int screen_width = 1920;
static const int screen_height = 1080;

int width = 0, height = 0;
float aspect = 0.f;
float cameraX = 0.f, cameraY = 0.f, cameraZ = 0.f;
float sphereLocX = 0.f, spherelocY = 0.f, sphereLocZ = 0.f;

GLuint renderingProgram = 0;
static const int numberVAOs = 1;
static const int numberVBOs = 3;
GLuint vao[numberVAOs] = { 0 };
GLuint vbo[numberVBOs] = { 0 };

glm::mat4 mMat(1.f), vMat(1.f), mvMat(1.f), pMat(1.f);
int mvLoc = 0;
int projLoc = 0;
float rotAmt = 0.f;

GLuint earthTextureId = 0;

Sphere earth = Sphere(48);

bool firstMouse = GL_TRUE;

float lastX = 0.f;
float lastY = 0.f;

GLboolean keys[1024];

Camera camera(glm::vec3(0.f, 0.f, 6.f));

GLfloat lastFrame = 0.0f;
GLfloat deltaTime = 0.0f;

void setupVertices()
```

```
{
    vector<int> ind = earth.getIndices();
    vector<glm::vec3> vert = earth.getVertices();
    vector<glm::vec2> tex = earth.getTexCoords();
    vector<glm::vec3> norm = earth.getNormals();
    vector<glm::vec3> tang = earth.getTangents();

    vector<float> pValues;
    vector<float> tValues;
    vector<float> nValues;

    int numIndices = earth.getNumIndices();
    for (int i=0; i<numIndices; i++)
    {
        pValues.push_back((vert[ind[i]]).x);
        pValues.push_back((vert[ind[i]]).y);
        pValues.push_back((vert[ind[i]]).z);

        tValues.push_back((tex[ind[i]]).s);
        tValues.push_back((tex[ind[i]]).t);

        nValues.push_back((norm[ind[i]]).x);
        nValues.push_back((norm[ind[i]]).y);
        nValues.push_back((norm[ind[i]]).z);
    }

    glGenVertexArrays(numberVAOs, vao);
    glBindVertexArray(vao[0]);

    glGenBuffers(numberVBOs, vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glBufferData(GL_ARRAY_BUFFER, pValues.size() * 4, &pValues[0], GL_STATIC_DRAW);

    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    glBufferData(GL_ARRAY_BUFFER, tValues.size() * 4, &tValues[0], GL_STATIC_DRAW);

    glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
    glBufferData(GL_ARRAY_BUFFER, nValues.size() * 4, &nValues[0], GL_STATIC_DRAW);
}

void press_key_callback(GLFWwindow* window, int key, int scancode, int action, int
mode)
{
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
```

```
    {
        glfwSetWindowShouldClose(window, GL_TRUE);
    }
    if (action == GLFW_PRESS)
    {
        keys[key] = GL_TRUE;
    }
    else if (action == GLFW_RELEASE)
    {
        keys[key] = GL_FALSE;
    }

}

void key_movement()
{
    if (keys[GLFW_KEY_W])
    {
        camera.ProcessKeyboard(FORWARD, deltaTime);
    }
    if (keys[GLFW_KEY_S])
    {
        camera.ProcessKeyboard(BACKWARD, deltaTime);
    }
    if (keys[GLFW_KEY_A])
    {
        camera.ProcessKeyboard(LEFT, deltaTime);
    }
    if (keys[GLFW_KEY_D])
    {
        camera.ProcessKeyboard(RIGHT, deltaTime);
    }

}

void mouse_move_callback(GLFWwindow* window, double xPos, double yPos)
{
    if (firstMouse)
    {
        lastX = xPos;
        lastY = yPos;
        firstMouse = GL_FALSE;
    }
```

```
    GLfloat xOffset = xPos - lastX;
    GLfloat yOffset = lastY - yPos;

    lastX = xPos;
    lastY = yPos;

    camera.ProcessMouseMovement(xOffset, yOffset);
}

void scroll_callback(GLFWwindow* window, double xPos, double yPos)
{
    camera.ProcessMouseScroll(yPos);
}

void init(GLFWwindow* window)
{
    renderingProgram = Utils::createShaderProgram("vertShader.glsl",
"fragShader.glsl");
    cameraX = 0.f, cameraY = 0.f, cameraZ = 4.f;
    sphereLocX = 0.f, spherelocY = 0.f, sphereLocZ = 0.f;
    glfwGetFramebufferSize(window, &width, &height);
    aspect = (float)width / (float)height;
    pMat = glm::perspective(glm::radians(45.f), aspect, 0.01f, 1000.f);

    setupVertices();

    earthTextureId = Utils::loadTexture("resource/earth.jpg");
}

void display(GLFWwindow* window, float currentTime)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glClearColor(0.1f, 0.2f, 1.f, 1.f);

    //必不可少！！！否则窗口是黑的，不会渲染任何东西
    glUseProgram(renderingProgram);

    GLfloat currentFrame = glfwGetTime();
    deltaTime = currentFrame - lastFrame;
    lastFrame = currentFrame;

    mvLoc = glGetUniformLocation(renderingProgram, "mv_matrix");
    projLoc = glGetUniformLocation(renderingProgram, "proj_matrix");
```

```
    //移动相机矩阵:这里必须是-cameraZ，否则相机看不到球体
    //vMat = glm::translate(glm::mat4(1.f), glm::vec3(cameraX, cameraY, -cameraZ));
    vMat = camera.GetViewMatrix();
    mMat = glm::translate(glm::mat4(1.f), glm::vec3(sphereLocX, spherelocY,
sphereLocZ));
    mMat = glm::rotate(glm::mat4(1.f), currentTime * 0.5f, glm::vec3(0.f, 1.f, 0.f));

    pMat = glm::perspective(camera.Zoom, (GLfloat)screen_width /
(GLfloat)screen_height, 0.001f, 1000.f);



    //mvMat = mMat * vMat;
    mvMat = vMat * mMat;
    glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvMat));
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(pMat));

    //绑定到球体顶点缓冲区
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    //指定了渲染时索引值为 index 的顶点属性数组的数据格式和位置

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    //启用或禁用通用顶点属性数组,参数0索引和着色器中的layout(location = 0）中的0相对应,
顶点位置
    glEnableVertexAttribArray(0);

    //绑定到纹理坐标
    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(1);

    //激活纹理坐标
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, earthTextureId);

    //背面剔除，默认情况下，背面剔除是关闭的
    glEnable(GL_CULL_FACE);
    glFrontFace(GL_CCW);

    glDrawArrays(GL_TRIANGLES, 0, earth.getNumIndices());
    //glDrawArrays(GL_TRIANGLES, 0, earth.getNumVertices());
}

void window_size_callback(GLFWwindow* window, int newWidth, int newHeight)
{
```

```
    glViewport(0, 0, newWidth, newHeight);
    aspect = (float)newWidth / (float)newHeight;
    pMat = glm::perspective(glm::radians(45.f), aspect, 0.01f, 1000.f);
}

int main(int argc, char** argv)
{
    int glfwState = glfwInit();
    if (GLFW_FALSE == glfwState)
    {
        cout << "GLFW initialize failed,invoke glfwInit()......Error file:" <<
__FILE__ << "......Error line:" << __LINE__ << endl;
        glfwTerminate();
        exit(EXIT_FAILURE);
    }

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);
    glfwWindowHint(GLFW_OPENGL_CORE_PROFILE, GLFW_OPENGL_PROFILE);
    glfwWindowHint(GLFW_RESIZABLE, GL_TRUE);

    GLFWwindow* window = glfwCreateWindow(screen_width, screen_height, "Sphere Draw",
nullptr, nullptr);
    if (!window)
    {
        cout << "GLFW create window failed,invoke glfwCreateWindow()......Error file:"
<< __FILE__ << "......Error line:" << __LINE__ << endl;
        glfwTerminate();
        exit(EXIT_FAILURE);
    }

    glfwMakeContextCurrent(window);
    glfwSetCursorPosCallback(window, mouse_move_callback);
    glfwSetKeyCallback(window, press_key_callback);
    glfwSetScrollCallback(window, scroll_callback);

    glfwSetWindowSizeCallback(window, window_size_callback);

    glfwSwapInterval(1);

    int glewState = glewInit();
    if (GLEW_OK != glewState)
    {
        cout << "GLEW initialize failed,invoke glewInit()......Error file:" <<
```

```
__FILE__ << ".....Error line:" << __LINE__ << endl;
        glfwTerminate();
        exit(EXIT_FAILURE);
    }

    init(window);

    while (!glfwWindowShouldClose(window))
    {
        display(window, glfwGetTime() * 0.3);
        key_movement();
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwDestroyWindow(window);
    glfwTerminate();
    exit(EXIT_SUCCESS);

    return 0;
}
```

四、心得与体会

通过对本次地球仪绘制大作业的完成和 OpenGL12 讲代码复现，我对纹理映射，视角变换，光照明模型有了更加深入的理解，更进一步了解了 OpenGL 的用法，从计算机图形学课程中可能收获到的不仅是图形学的基础知识，还有 OpenGL 的编程技术。

计算机图形学是交互式图形开发的基本理论，同时也是一门实践性的学科，我通过复现 OpenGL 基础教程中的 12 讲代码，体会了 OpenGL 的基础思想。地球仪大作业的完成让我收获颇丰，和 OpenGL12 讲代码复现相比，地球仪的实现让我能更加灵活的调用核心、工具库,通过捕获键盘和鼠标的信息来实现用户交互功能。为实现该功能，我和我的队友花费了较多的时间，最终实现了较为满意的工程结果。感觉到 OpenGL 良好的底层封装，把原本很复杂的实现细节编程几行简单的代码调用就能完成的工作，因此我认为熟练使用 OpenGL 编程需要对它的 API 有足够的了解，否则在完成工程时调用函数不熟练会带来很多不便。

通过这学期的图形学学习，为以后有机会接触图形学相关知识打下了一定基础，同时也激发了自己学习图形学的兴趣，我想如果以后有机会我会更加深入学习有关计算机图形学的知识。