附录——源代码

头文件　MyHeader.h

```
#ifndef MYHEADER
#define MYHEADER

// 宏定义
#define STB_IMAGE_IMPLEMENTATION
#define MAX_BONE_INFLUENCE 4

// 附加头文件
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <iostream>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <tuple>
#include <vector>
#include <algorithm>
#include "assimp/scene.h"
#include "assimp/Importer.hpp"
#include "assimp/postprocess.h"

// opengl窗口的高和宽
constexpr int WINDOW_WIDTH = 800;
constexpr int WIDOW_HEIGHT = 600;
// 摄像机的三个坐标轴
extern glm::vec3 cameraPos;
extern glm::vec3 cameraFront;
extern glm::vec3 cameraUp;
extern bool keys[1024];
extern GLfloat deltaTime;
extern GLfloat lastFrame;
extern GLfloat yaw;
extern GLfloat pitch;
extern bool firstMouse;
extern GLfloat lastX;
extern GLfloat lastY;
extern GLfloat aspect;
extern glm::vec3 lightPos;
```

```
// 顶点类
struct Vertex
{
    // 顶点坐标
    glm::vec3 Position;
    // 法线坐标
    glm::vec3 Normal;
    // 纹理坐标
    glm::vec2 TexCoords;
    // 两个切线空间坐标轴
    glm::vec3 Tangent;
    glm::vec3 Bitangent;
    int m_BoneIDs[MAX_BONE_INFLUENCE];
    float m_Weights[MAX_BONE_INFLUENCE];
};

// 纹理类
struct Texture
{
    // 纹理标识符
    GLuint id;
    // 纹理种类
    std::string type;
    // 路径
    aiString path;
};

// 网格类,
class Mesh
{
public:
    // 顶点数组
    std::vector<Vertex> vertices;
    // 索引数组
    std::vector<GLuint> indices;
    // 纹理数组
    std::vector<Texture> textures;
    Mesh(std::vector<Vertex> vertices, std::vector<GLuint> indices,
std::vector<Texture> textures);
    void Draw(const GLuint shaderprogram);
private:
    // 顶点数组对象、顶点缓冲对象、索引缓冲对象
    GLuint VAO, VBO, EBO;
    void setupMesh(void);
};

GLint TextureFromFile(const char* path, std::string directory, aiTextureType
type);

// 模型类
class Model
{
public:
    Model(const char* path)
    {
        this->loadModel(path);
    }
    void Draw(GLuint shaderProgram)
    {
        for (GLuint i = 0; i < this->meshes.size(); i++)
        {
            this->meshes[i].Draw(shaderProgram);
        }
    }
private:
    // 网格数组
    std::vector<Mesh> meshes;
    // 纹理目录
    std::string directory;
    // 已加载过的纹理
    std::vector<Texture> textures_loaded;

    void loadModel(std::string path);
    void processNode(aiNode* node, const aiScene* scene);
    Mesh processMesh(aiMesh* mesh, const aiScene* scene);
    std::vector<Texture> loadMaterialTextures(aiMaterial* mat, aiTextureType
type, std::string typeName);
};



// 光源模型的顶点数组
const float vertices[] = {
    // positions          // normals           // texture coords
    -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f, 0.0f,
     0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f, 0.0f,
     0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f, 1.0f,
     0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f, 1.0f,
    -0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f, 1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f, 0.0f,

    -0.5f, -0.5f,  0.5f,  0.0f,  0.0f, 1.0f,   0.0f, 0.0f,
     0.5f, -0.5f,  0.5f,  0.0f,  0.0f, 1.0f,   1.0f, 0.0f,
     0.5f,  0.5f,  0.5f,  0.0f,  0.0f, 1.0f,   1.0f, 1.0f,
     0.5f,  0.5f,  0.5f,  0.0f,  0.0f, 1.0f,   1.0f, 1.0f,
    -0.5f,  0.5f,  0.5f,  0.0f,  0.0f, 1.0f,   0.0f, 1.0f,
    -0.5f, -0.5f,  0.5f,  0.0f,  0.0f, 1.0f,   0.0f, 0.0f,

    -0.5f,  0.5f,  0.5f, -1.0f,  0.0f,  0.0f,  1.0f, 0.0f,
    -0.5f,  0.5f, -0.5f, -1.0f,  0.0f,  0.0f,  1.0f, 1.0f,
    -0.5f, -0.5f, -0.5f, -1.0f,  0.0f,  0.0f,  0.0f, 1.0f,
    -0.5f, -0.5f, -0.5f, -1.0f,  0.0f,  0.0f,  0.0f, 1.0f,
    -0.5f, -0.5f,  0.5f, -1.0f,  0.0f,  0.0f,  0.0f, 0.0f,
    -0.5f,  0.5f,  0.5f, -1.0f,  0.0f,  0.0f,  1.0f, 0.0f,

     0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  0.0f,  1.0f, 0.0f,
     0.5f,  0.5f, -0.5f,  1.0f,  0.0f,  0.0f,  1.0f, 1.0f,
     0.5f, -0.5f, -0.5f,  1.0f,  0.0f,  0.0f,  0.0f, 1.0f,
     0.5f, -0.5f, -0.5f,  1.0f,  0.0f,  0.0f,  0.0f, 1.0f,
     0.5f, -0.5f,  0.5f,  1.0f,  0.0f,  0.0f,  0.0f, 0.0f,
     0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  0.0f,  1.0f, 0.0f,

    -0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f,  0.0f, 1.0f,
     0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f,  1.0f, 1.0f,
     0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f,  1.0f, 0.0f,
     0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f,  1.0f, 0.0f,
    -0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f,  0.0f, 0.0f,
    -0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f,  0.0f, 1.0f,

    -0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f,  0.0f, 1.0f,
     0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f,  1.0f, 1.0f,
     0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,  1.0f, 0.0f,
     0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,  1.0f, 0.0f,
    -0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,  0.0f, 0.0f,
    -0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f,  0.0f, 1.0f
};
```

```
// 光源模型的索引坐标
const GLuint indices[] = {
    0, 1, 3,
    1, 2, 3
};

// 按键回调函数
void key_callback(GLFWwindow* window, int key, int scancode, int action, int
mode);

// 创建特定类型的shader并编译
GLuint MyCreateShader(const GLenum ShaderType, const GLchar* &ShaderSource);

// 创建着色器程序
GLuint MyCreateShaderProgram(const GLchar* &VertexShaderSource, const GLchar*
&FragmentShaderSource);

// 初始化glfw库、glew库，创建窗口和视口
GLFWwindow* MyInitOpenGL(void);

// 清空删除VBO,VAO,EBO
void MyDeleteVertexBuffer(GLuint& VBO, GLuint& VAO, GLuint& EBO) noexcept;

// 创建摄像机坐标系统
const glm::mat4 MyCamera(void);

// 完成上下左右键的检测回调
void do_movement(void);

// 鼠标回调函数
void mouse_callback(GLFWwindow* window, double xpos, double ypos);

// 鼠标滚轮回调函数
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset);
#endif
```

源文件　　MyFunction.cpp

```
#include "MyHeader.h"
#include "stb_image.h"
```

```
#include <string>

extern glm::vec3 cameraPos = glm::vec3(0.0f, -0.8f, 2.0f);
extern glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
extern glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);
extern bool keys[1024]{true};
extern GLfloat deltaTime = 0.0f;
extern GLfloat lastFrame = 0.0f;
extern GLfloat yaw = -90.0f;
extern GLfloat pitch = 0.0f;
extern bool firstMouse = true;
extern GLfloat lastX = 400;
extern GLfloat lastY = 300;
extern GLfloat aspect = 20.0f;
extern glm::vec3 lightPos(1.2f, 0.5f, 3.0f);

// 按键回调函数
void key_callback(GLFWwindow * window, int key, int scancode, int action, int mode)
{
        if (key == GLFW_KEY_ESCAPE and action == GLFW_PRESS)
        {
            glfwSetWindowShouldClose(window, GL_TRUE);
        }

        // 动作判定
        if (action == GLFW_PRESS)
            keys[key] = true;
        else if (action == GLFW_RELEASE)
            keys[key] = false;
}

// 创建特定类型的shader并编译
GLuint MyCreateShader(const GLenum ShaderType, const GLchar* &ShaderSource)
{
    // create shader
    GLuint shader = glCreateShader(ShaderType);
    // attribute shader with program
    glShaderSource(shader, 1, &ShaderSource, nullptr);
    // compile shader
    glCompileShader(shader);
    return shader;
```

```
}

// 创建着色器程序
GLuint MyCreateShaderProgram(const GLchar* &VertexShaderSource, const GLchar*
&FragmentShaderSource)
{
    // create a vertex shader
    GLuint VertexShader = MyCreateShader(GL_VERTEX_SHADER,
VertexShaderSource);

    // test vertex shader if vaild
    GLint ifsuccess = GL_TRUE;
    GLchar info[512];
    glGetShaderiv(VertexShader, GL_COMPILE_STATUS, &ifsuccess);
    if (not ifsuccess)
    {
        glGetShaderInfoLog(VertexShader, 512, nullptr, info);
        std::cout << "ERROR: vertex shader build or compile failed!" <<
std::endl;
        std::abort();
    }

    // create a fragment shader
    GLuint FragmentShader = MyCreateShader(GL_FRAGMENT_SHADER,
FragmentShaderSource);

    // test fragment shader if vaild
    glGetShaderiv(FragmentShader, GL_COMPILE_STATUS, &ifsuccess);
    if (not ifsuccess)
    {
        glGetShaderInfoLog(FragmentShader, 512, nullptr, info);
        std::cout << "ERROR: fragment shader build or compile failed!" <<
std::endl;
        std::abort();
    }

    // create a shader program
    GLuint shaderProgram = glCreateProgram();
    // attach vertex shader and fragment shader to content
    glAttachShader(shaderProgram, VertexShader);
    glAttachShader(shaderProgram, FragmentShader);
    // link Program with content
    glLinkProgram(shaderProgram);

    // test shader program if vaild
    glGetProgramiv(shaderProgram, GL_COMPILE_STATUS, &ifsuccess);
    if (not ifsuccess)
    {
        glGetProgramInfoLog(shaderProgram, 512, nullptr, info);
        std::cout << "ERROR:shader program wrong!" << std::endl;
        std::abort();
    }

    // delete shader
    glDeleteShader(VertexShader);
    glDeleteShader(FragmentShader);

    // return shader program
    return shaderProgram;
}


// 初始化glfw库、glew库，创建窗口和视口
GLFWwindow* MyInitOpenGL(void)
{
    // glfw init
    if (GLFW_TRUE != glfwInit())
    {
        std::cout << "ERROR: init glfw failed!" << std::endl;
        std::abort();
    }

    // set some parameters
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);

    // build a window
    GLFWwindow* window = glfwCreateWindow(WINDOW_WIDTH, WIDOW_HEIGHT,
"LearnOpenGL", nullptr, nullptr);
    if (nullptr == window)
    {
        std::cout << "ERROR: create a window failed!" << std::endl;
        glfwTerminate();
        std::abort();
    }

    // make content
    glfwMakeContextCurrent(window);

    // init glew
    glewExperimental = GL_TRUE;
    if (GLEW_OK != glewInit())
    {
        std::cout << "ERROR: init glew failed!" << std::endl;
        std::abort();
    }

    // make viewport
    glViewport(0, 0, WINDOW_WIDTH, WIDOW_HEIGHT);

    return window;
}

// 清空删除VBO,VAO,EBO
void MyDeleteVertexBuffer(GLuint & VBO, GLuint & VAO, GLuint & EBO) noexcept
{
    glDeleteBuffers(1, &VBO);
    glDeleteBuffers(1, &EBO);
    glDeleteVertexArrays(1, &VAO);
}

// 创建摄像机坐标系统
const glm::mat4 MyCamera(void)
{
    glm::mat4 view(1.0f);
    view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
    return view;
}

// 完成上下左右键的检测回调
void do_movement(void)
{
    GLfloat currentFrame = glfwGetTime();
    deltaTime = currentFrame - lastFrame;
    lastFrame = currentFrame;
    GLfloat cameraSpeed = 15.0f * deltaTime;
    if (keys[GLFW_KEY_UP])
        cameraPos += cameraSpeed * cameraFront;
    if (keys[GLFW_KEY_DOWN])
        cameraPos -= cameraSpeed * cameraFront;
    if(keys[GLFW_KEY_LEFT])
        cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) *
cameraSpeed;
    if (keys[GLFW_KEY_RIGHT])
        cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) *
cameraSpeed;

}

// 鼠标回调函数
void mouse_callback(GLFWwindow * window, double xpos, double ypos)
{
        if (firstMouse)
        {
            lastX = xpos;
            lastY = ypos;
            firstMouse = false;
        }
        GLfloat xoffset = xpos - lastX;
        GLfloat yoffset = lastY - ypos;
        lastX = xpos;
        lastY = ypos;

        GLfloat sensitivity = 0.05f;
        xoffset *= sensitivity;
        yoffset *= sensitivity;

        yaw += xoffset;
        pitch += yoffset;

        if (pitch > 89.0f)
            pitch = 89.0f;
        if (pitch < -89.0f)
            pitch = -89.0f;

        glm::vec3 front;
        front.x = cos(glm::radians(pitch)) * cos(glm::radians(yaw));
        front.y = sin(glm::radians(pitch));
        front.z = cos(glm::radians(pitch)) * sin(glm::radians(yaw));
        cameraFront = glm::normalize(front);

}

// 鼠标滚轮回调函数
void scroll_callback(GLFWwindow * window, double xoffset, double yoffset)
{
        aspect -= yoffset;
    if (aspect < 1.0f)
        aspect = 1.0f;
    if (aspect > 45.0f)
        aspect = 45.0f;
}

// Mesh 构造函数
Mesh::Mesh(std::vector<Vertex> vertices, std::vector<GLuint> indices,
std::vector<Texture> textures)
{
    this->vertices = vertices;
    this->indices = indices;
    this->textures = textures;
    this->setupMesh();
}

// 完成 Mesh 的 VBO,VAO,EBO 绑定
void Mesh::setupMesh(void)
{
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);

    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);

    glBufferData(GL_ARRAY_BUFFER, this->vertices.size() * sizeof(Vertex),
&this->vertices[0], GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, this->indices.size() *
sizeof(GLuint), &this->indices[0], GL_STATIC_DRAW);

    // 设置顶点坐标
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(GLvoid*)0);
    glEnableVertexAttribArray(0);

    // 设置法线坐标
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(GLvoid*)offsetof(Vertex, Normal));
    glEnableVertexAttribArray(1);

    // 设置纹理坐标
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(GLvoid*)offsetof(Vertex, TexCoords));
    glEnableVertexAttribArray(2);

    // vertex tangent
    glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, Tangent));
    glEnableVertexAttribArray(3);

    // vertex bitangent
    glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, Bitangent));
    glEnableVertexAttribArray(4);
    // ids
    glVertexAttribIPointer(5, 4, GL_INT, sizeof(Vertex),
(void*)offsetof(Vertex, m_BoneIDs));
    glEnableVertexAttribArray(5);

    // weights
    glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, m_Weights));
    glEnableVertexAttribArray(6);
    glBindVertexArray(0);

}

// 网格纹理处理
void Mesh::Draw(const GLuint shaderprogram)
{
```

```
// 绑定合适的纹理
unsigned int diffuseNr = 1;
unsigned int specularNr = 1;
unsigned int normalNr = 1;
unsigned int heightNr = 1;
unsigned int ambientNr = 1;
unsigned int displacementNr = 1;
unsigned int emissiveNr = 1;
unsigned int lightmapNr = 1;
unsigned int reflectionNr = 1;
unsigned int shininessNr = 1;
unsigned int opacityNr = 1;
unsigned int noneNr = 1;
unsigned int unknownNr = 1;
for (unsigned int i = 0; i < textures.size(); i++)
{
    // 激活纹理单元
    glActiveTexture(GL_TEXTURE0 + i);
    std::string number;
    std::string name = textures[i].type;
    if (name == "texture_diffuse")
        number = std::to_string(diffuseNr++);
    else if (name == "texture_specular")
        number = std::to_string(specularNr++);
    else if (name == "texture_normal")
        number = std::to_string(normalNr++);
    else if (name == "texture_height")
        number = std::to_string(heightNr++);
    else if (name == "texture_ambient")
        number = std::to_string(ambientNr++);
    else if (name == "texture_displacement")
        number = std::to_string(displacementNr++);
    else if (name == "texture_emissive")
        number = std::to_string(emissiveNr++);
    else if (name == "texture_lightmap")
        number = std::to_string(lightmapNr++);
    else if (name == "texture_reflection")
        number = std::to_string(reflectionNr++);
    else if (name == "texture_shininess")
        number = std::to_string(shininessNr++);
    else if (name == "texture_opacity")
        number = std::to_string(opacityNr++);
        else if (name == "texture_none")
            number = std::to_string(noneNr++);
        else if (name == "texture_unknown")
            number = std::to_string(unknownNr++);

        // 给着色器的 uniform 赋值
        glUniform1i(glGetUniformLocation(shaderprogram, ("material." + name +
number).c_str()), i);
        // 绑定纹理
        glBindTexture(GL_TEXTURE_2D, textures[i].id);
    }

    // draw mesh
    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, static_cast<unsigned int>(indices.size()),
GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);

    // always good practice to set everything back to defaults once configured.
    glActiveTexture(GL_TEXTURE0);
}

// 预处理加载模型
void Model::loadModel(std::string path)
{
    Assimp::Importer import;
    const aiScene* scene = import.ReadFile(path, aiProcess_Triangulate |
aiProcess_GenSmoothNormals | aiProcess_FlipUVs | aiProcess_CalcTangentSpace);
    if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE
|| !scene->mRootNode)
    {
        std::cout << "ERROR::ASSIMP::" << import.GetErrorString() << std::endl;
        return;
    }
    this->directory = path.substr(0, path.find_last_of('/'));
    this->processNode(scene->mRootNode, scene);
}

// 处理每个节点
void Model::processNode(aiNode* node, const aiScene* scene)
{
    // 添加当前节点中的所有 Mesh
    for (GLuint i = 0; i < node->mNumMeshes; i++)
    {
        aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
        this->meshes.push_back(this->processMesh(mesh, scene));
    }
    // 递归处理该节点的子孙节点
    for (GLuint i = 0; i < node->mNumChildren; i++)
    {
        this->processNode(node->mChildren[i], scene);
    }
}

// 处理网格
Mesh Model::processMesh(aiMesh* mesh, const aiScene* scene)
{
    std::vector<Vertex> vertices;
    std::vector<GLuint> indices;
    std::vector<Texture> textures;

    for (GLuint i = 0; i < mesh->mNumVertices; i++)
    {
        Vertex vertex;
        glm::vec3 vector(1.0f);
        // 获取位置坐标
        vector.x = mesh->mVertices[i].x;
        vector.y = mesh->mVertices[i].y;
        vector.z = mesh->mVertices[i].z;
        vertex.Position = vector;
        // 获取法线坐标
        if (mesh->HasNormals())
        {
            vector.x = mesh->mNormals[i].x;
            vector.y = mesh->mNormals[i].y;
            vector.z = mesh->mNormals[i].z;
            vertex.Normal = vector;
        }
        // 获取纹理坐标
        if (mesh->mTextureCoords[0])
        {
            glm::vec2 vec(1.0f);
            vec.x = mesh->mTextureCoords[0][i].x;
            vec.y = mesh->mTextureCoords[0][i].y;
            vertex.TexCoords = vec;
            // tangent
            vector.x = mesh->mTangents[i].x;
            vector.y = mesh->mTangents[i].y;
            vector.z = mesh->mTangents[i].z;
            vertex.Tangent = vector;
            // bitangent
            vector.x = mesh->mBitangents[i].x;
            vector.y = mesh->mBitangents[i].y;
            vector.z = mesh->mBitangents[i].z;
            vertex.Bitangent = vector;
        }
        else
            vertex.TexCoords = glm::vec2(0.0f, 0.0f);
        vertices.push_back(vertex);
    }

    // 处理顶点索引
    for (GLuint i = 0; i < mesh->mNumFaces; i++)
    {
        aiFace face = mesh->mFaces[i];
        for (GLuint j = 0; j < face.mNumIndices; j++)
            indices.push_back(face.mIndices[j]);
    }
    //std::cout << mesh->mMaterialIndex << std::endl;

    if (mesh->mMaterialIndex >= 0)
    {
        // 处理材质
        aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];
        // 1. Diffuse maps  get
        std::vector<Texture> diffuseMaps =
this->loadMaterialTextures(material, aiTextureType_DIFFUSE,
"texture_diffuse");
        textures.insert(textures.end(), diffuseMaps.begin(),
diffuseMaps.end());
        // 2. Specular maps
        std::vector<Texture> specularMaps =
this->loadMaterialTextures(material, aiTextureType_SPECULAR,
"texture_specular");
        textures.insert(textures.end(), specularMaps.begin(),
specularMaps.end());
```

```
        // 3. normal maps   get
        std::vector<Texture> normalMaps = loadMaterialTextures(material,
aiTextureType_NORMALS, "texture_normal");
        textures.insert(textures.end(), normalMaps.begin(),
normalMaps.end());
        //// 4. height maps
        //std::vector<Texture> heightMaps = loadMaterialTextures(material,
aiTextureType_HEIGHT, "texture_height");
        //textures.insert(textures.end(), heightMaps.begin(),
heightMaps.end());
        //// 5. ambient maps
        //std::vector<Texture> ambientsMaps = loadMaterialTextures(material,
aiTextureType_AMBIENT, "texture_ambient");
        //textures.insert(textures.end(), ambientsMaps.begin(),
ambientsMaps.end());
        //// 6. displacement maps
        //std::vector<Texture> diplacementMaps =
loadMaterialTextures(material, aiTextureType_DISPLACEMENT,
"texture_displacement");
        //textures.insert(textures.end(), diplacementMaps.begin(),
diplacementMaps.end());
        //// 7. emissive maps
        //std::vector<Texture> emissiveMaps = loadMaterialTextures(material,
aiTextureType_EMISSIVE, "texture_emissive");
        //textures.insert(textures.end(), emissiveMaps.begin(),
emissiveMaps.end());
        // 8. lightmap     get
        std::vector<Texture> lightmapMaps = loadMaterialTextures(material,
aiTextureType_LIGHTMAP, "texture_lightmap");
        textures.insert(textures.end(), lightmapMaps.begin(),
lightmapMaps.end());
        //// 9.
        //std::vector<Texture> reflectionMaps = loadMaterialTextures(material,
aiTextureType_REFLECTION, "texture_reflection");
        //textures.insert(textures.end(), reflectionMaps.begin(),
reflectionMaps.end());
        //// 10.
        //std::vector<Texture> shininessMaps = loadMaterialTextures(material,
aiTextureType_SHININESS, "texture_shininess");
        //textures.insert(textures.end(), shininessMaps.begin(),
shininessMaps.end());
        //// 11.
        //std::vector<Texture> opacityMaps = loadMaterialTextures(material,
aiTextureType_OPACITY, "texture_opacity");
        //textures.insert(textures.end(), opacityMaps.begin(),
opacityMaps.end());
        //// 12.
        //std::vector<Texture> noneMaps = loadMaterialTextures(material,
aiTextureType_NONE, "texture_none");
        //textures.insert(textures.end(), noneMaps.begin(), noneMaps.end());
        //// 13.
        //std::vector<Texture> unknownMaps = loadMaterialTextures(material,
aiTextureType_UNKNOWN, "texture_unknown");
        //textures.insert(textures.end(), unknownMaps.begin(),
unknownMaps.end());
    }

    return Mesh(vertices, indices, textures);
}

std::vector<Texture> Model::loadMaterialTextures(aiMaterial* mat,
aiTextureType type, std::string typeName)
{
    std::vector<Texture> textures;
    for (GLuint i = 0; i < mat->GetTextureCount(type); i++)
    {

        aiString str;
        mat->GetTexture(type, i, &str);
        GLboolean skip = false;
        for (GLuint j = 0; j < textures_loaded.size(); j++)
        {
            if (textures_loaded[j].path == str)
            {
                textures.push_back(textures_loaded[j]);
                skip = true;
                break;
            }
        }
        if (!skip)
        {   // 如果纹理没有被加载过，加载之
            Texture texture;
            texture.id = TextureFromFile(str.C_Str(), this->directory, type);
            texture.type = typeName;
            texture.path = str.C_Str();
            textures.push_back(texture);
            this->textures_loaded.push_back(texture);  // 添加到纹理列表
textures
        }
    }
    return textures;
}

// 加载纹理贴图
GLint TextureFromFile(const char* path, std::string directory, aiTextureType
type)
{
    std::string filename = std::string(path);
    filename = directory + ’/’ + filename;

    //std::cout << filename << std::endl;

    unsigned int textureID;
    glGenTextures(1, &textureID);

    int width, height, nrComponents;
    unsigned char *data = stbi_load(filename.c_str(), &width, &height,
&nrComponents, 0);
    if (data)
    {
        GLenum format;
        if (nrComponents == 1)
            format = GL_RED;
        else if (nrComponents == 3)
            format = GL_RGB;
        else if (nrComponents == 4)
            format = GL_RGBA;

        glBindTexture(GL_TEXTURE_2D, textureID);
        if (type == aiTextureType_DIFFUSE)
        {
            glTexImage2D(GL_TEXTURE_2D, 0, GL_SRGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, data);

        }
        else
        {
            glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format,
GL_UNSIGNED_BYTE, data);
        }
        glGenerateMipmap(GL_TEXTURE_2D);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        stbi_image_free(data);
    }
    else
    {
        std::cout << "Texture failed to load at path: " << path << std::endl;
        stbi_image_free(data);
    }

    return textureID;
}
```

源文件　pra1_main.cpp

```
#include "MyHeader.h"

void Draw3DModelwithLight(void)
{
    // 模型顶点着色器
    const GLchar* ModelVertexShader =
        "#version 330 core\n"
        "layout (location = 0) in vec3 position;\n"
        "layout (location = 1) in vec3 normal;\n"
        "layout (location = 2) in vec2 texCoords;\n"
        "layout (location = 3) in vec3 tangent;\n"
        "layout (location = 4) in vec3 bitangent;\n"

        "out vec2 TexCoords;\n"
        "out vec3 fragPosition;\n"
        "out vec3 Normal;\n"
        "out vec3 TangentLightPos;\n"
        ″out vec3 TangentViewPos;\n″
        ″out vec3 TangentFragPos;\n″

        ″uniform vec3 lightPosition;\n″
        ″uniform vec3 viewPosition;\n″
        ″uniform mat4 model;\n″
        ″uniform mat4 view;\n″
        ″uniform mat4 projection;\n″
        ″void main()\n″
        ″{\n″
        ″fragPosition = vec3(model * vec4(position, 1.0f));\n″
        ″mat3 normalMatrix = transpose(inverse(mat3(model)));\n″
        ″vec3 T = normalize(normalMatrix * tangent);\n″
        ″vec3 N = normalize(normalMatrix * normal);\n″
        ″T = normalize(T - dot(T, N) * N);″
        ″vec3 B = cross(N, T);″
        ″mat3 TBN = transpose(mat3(T, B, N));\n″
        ″TangentLightPos = TBN * lightPosition;\n″
        ″TangentViewPos  = TBN * viewPosition;\n″
        ″TangentFragPos  = TBN * fragPosition;\n″

        ″gl_Position = projection * view * model * vec4(position, 1.0f);\n″
        ″TexCoords = texCoords;\n″
        ″Normal = mat3(transpose(inverse(model))) * normal;\n″
        ″}\0″;
    // 模型片段着色器
    const GLchar* ModelFragShader_Blinn_mdfy =
        ″#version 330 core\n″
        ″out vec4 color;\n″
        ″in vec3 Normal;\n″
        ″in vec2 TexCoords;\n″
        ″in vec3 fragPosition;\n″
        ″uniform vec3 lightPos;\n″
        ″uniform vec3 viewPos;\n″

        ″struct Material\n″
        ″{\n″
        ″sampler2D texture_diffuse1;\n″
        ″sampler2D texture_lightmap1;\n″
        ″vec3 specular;\n″
        ″float shininess;\n″
        ″};\n″
```

```
        "uniform Material material;\n"

        "struct Light\n"
        "{\n"
        "vec3 position;\n"
        "vec3 ambient;\n"
        "vec3 diffuse;\n"
        "vec3 specular;\n"
        "float constant;\n"
        "float linear;\n"
        "float quadratic;\n"
        "};\n"
        "uniform Light light;\n"

        "void main()\n"
        "{\n"
        // 计算衰减
        "float distance = length(light.position - fragPosition);\n"
        "float attenuation = 1.0f / (light.constant + light.linear*distance +
light.quadratic*(distance*distance));\n"
        // 环境光照
        "vec3 ambient =  light.ambient * vec3(texture(material.texture_diffuse1,
TexCoords)) * vec3(texture(material.texture_lightmap1, TexCoords));\n"
        // 漫反射光照
        "vec3 norm = normalize(Normal);\n"
        "vec3 lightDir = normalize(lightPos - fragPosition);\n"
        "float diff = max(dot(norm, lightDir), 0.0f);\n"
        "vec3 diffuse = diff * light.diffuse *
vec3(texture(material.texture_diffuse1, TexCoords));\n"
        // 镜面高光
        "vec3 viewDir = normalize(viewPos - fragPosition);\n"
        "vec3 halfwayDir = normalize(lightDir + viewDir);\n"   // 半程向量
        "float spec = pow(max(dot(norm, halfwayDir), 0.0f),
material.shininess);\n"
        "vec3 specular = (material.specular * spec)* light.specular;\n"
        "ambient *= attenuation;\n"
        "diffuse *= attenuation;\n"
        "specular *= attenuation;\n"
        // 光照结果
        "vec3 result = ambient + diffuse + specular;\n"
        "float gamma = 2.2;\n"
        "result = pow(result, vec3(1.0f/gamma));\n"
        ″color = vec4(result, 1.0f);\n″
        ″}\0″;



    // 光照的片段着色器
    const GLchar* LightFragShader =
        ″#version 330 core\n″
        ″out vec4 color;\n″

        ″void main()\n″
        ″{\n″
        ″color = vec4(1.0f);\n″
        ″}\0″;


    // 初始化opengl
    GLFWwindow* window = MyInitOpenGL();

    // 键盘、鼠标、滚轮回调
    glfwSetKeyCallback(window, key_callback);
    glfwSetCursorPosCallback(window, mouse_callback);
    glfwSetScrollCallback(window, scroll_callback);

    // 隐藏光标
    glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

    // 深度测试
    glEnable(GL_DEPTH_TEST);

    // 创建VBO，VAO，EBO
    GLuint lightVBO, lightVAO, lightEBO;
    glGenBuffers(1, &lightVBO);
    glBindBuffer(GL_ARRAY_BUFFER, lightVBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
    glGenBuffers(1, &lightEBO);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, lightEBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);
    glGenVertexArrays(1, &lightVAO);
    glBindVertexArray(lightVAO);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat),
(GLvoid*)0);
    glEnableVertexAttribArray(0);
    glBindVertexArray(GL_FALSE);

    // 创建着色器程序
    GLuint ShaderProgram = MyCreateShaderProgram(ModelVertexShader,
ModelFragShader_Blinn_mdfy);
    GLuint LightShaderProgram = MyCreateShaderProgram(ModelVertexShader,
LightFragShader);

    // 创建Model实体
    Model ourModel("D:/zhuo_mian/学习
/opengl_pra/opengl_p1/opengl_project_pra_1/model5/scene.gltf");

    // 渲染循环
    while (not glfwWindowShouldClose(window))
    {
        // Set frame time
        GLfloat currentFrame = glfwGetTime();
        deltaTime = currentFrame - lastFrame;
        lastFrame = currentFrame;

        // 检查事件
        glfwPollEvents();
        do_movement();

        // 清空缓冲
        glClearColor(0.05f, 0.05f, 0.05f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // 启动着色器程序
        glUseProgram(ShaderProgram);

        GLint lightPosLoc = glGetUniformLocation(ShaderProgram, "lightPos");
        GLint viewPosLoc = glGetUniformLocation(ShaderProgram, "viewPos");
        glUniform3f(lightPosLoc, lightPos.x, lightPos.y, lightPos.z);
        glUniform3f(viewPosLoc, cameraPos.x, cameraPos.y, cameraPos.z);

        glUniform3f(glGetUniformLocation(ShaderProgram, "material.specular"),
0.5f, 0.5f, 0.5f);
        glUniform1f(glGetUniformLocation(ShaderProgram, "material.shininess"),
128.0f);
```

```
        glUniform3f(glGetUniformLocation(ShaderProgram, "light.ambient"), 0.08f,
0.08f, 0.08f);
        glUniform3f(glGetUniformLocation(ShaderProgram, "light.diffuse"), 0.85f,
0.85f, 0.85f);
        glUniform3f(glGetUniformLocation(ShaderProgram, "light.specular"), 1.0f,
1.0f, 1.0f);
        glUniform1f(glGetUniformLocation(ShaderProgram, "light.constant"),
1.0f);
        glUniform1f(glGetUniformLocation(ShaderProgram, "light.linear"), 0.09);
        glUniform1f(glGetUniformLocation(ShaderProgram, "light.quadratic"),
0.032);



        // model 坐标系统转换
        glm::mat4 projection = glm::perspective(glm::radians(aspect), 800.0f /
600.0f, 0.1f, 100.0f);
        glm::mat4 view = MyCamera();
        glUniformMatrix4fv(glGetUniformLocation(ShaderProgram,"projection"), 1,
GL_FALSE, glm::value_ptr(projection));
        glUniformMatrix4fv(glGetUniformLocation(ShaderProgram, "view"), 1,
GL_FALSE, glm::value_ptr(view));
        glm::mat4 model(1.0f);
        model = glm::translate(model, glm::vec3(0.0f, -0.8f, 0.0f));
        model = glm::scale(model, glm::vec3(0.2f, 0.2f, 0.2f));
        model = glm::rotate(model, glm::radians(180.0f), glm::vec3(0.0f, 1.0f,
0.0f));
        glUniformMatrix4fv(glGetUniformLocation(ShaderProgram, "model"), 1,
GL_FALSE, glm::value_ptr(model));

        // 启动model构建
        ourModel.Draw(ShaderProgram);

        // light
        glUseProgram(LightShaderProgram);
        GLuint modelLoc = glGetUniformLocation(LightShaderProgram, "model");
        GLuint viewLoc = glGetUniformLocation(LightShaderProgram, "view");
        GLuint projLoc = glGetUniformLocation(LightShaderProgram, "projection");
        // 光源模型 坐标系统转换
        glm::mat4 lightmodelmat = glm::mat4(1.0f);
        glm::mat4 lightviewmat = MyCamera();
        glm::mat4 lightprjmat = glm::mat4(1.0f);
        lightprjmat = glm::perspective(glm::radians(aspect), 800.0f / 600.0f,
0.1f, 100.0f);
        glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(lightviewmat));
        glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(lightprjmat));
        // human_skull
        lightPos.x = 0.25f + sin(glfwGetTime()) * 2.2f;
        lightPos.y = -0.1f + sin(glfwGetTime() / 2.0f) * 3.6f;
        lightmodelmat = glm::translate(lightmodelmat, lightPos);
        lightmodelmat = glm::scale(lightmodelmat, glm::vec3(0.2f));
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(lightmodelmat));

        // 画光源模型
        glBindVertexArray(lightVAO);
        glDrawArrays(GL_TRIANGLES, 0, 36);
        glBindVertexArray(0);



        // 交换缓冲
        glfwSwapBuffers(window);
    }
    MyDeleteVertexBuffer(lightVBO, lightVAO, lightEBO);
    glfwTerminate();
}

int main()
{
    Draw3DModelwithLight();
    return 0;
}
```