# 1．引言

支持多个光源的光照效果，使用着色器渲染，支持多种视点浏览方式，支持纹理-使用图片进行纹理映射，支持obj三维模型文件格式，添加了bezier曲面,场景中添加光照，bezier曲面上添加纹理贴图以达到个性化效果。具体为添加bezier曲面，至少包含5\*5个控制点；为bezier曲面添加纹理贴图，图片自行选择；在场景中添加ambient light, diffuse light 和 specular light进行组合来产生不同的光照效果；场景中的图形能分别显示顶点模式，面片模式和线框模式。键盘的上下左右实现视角的缩小、放大和左右移动；键盘WSAD实现对模型的上下左右控制；TGFH实现对曲面的上下左右控制。QE可以切换显示模式。

# 2. 主体介绍

## 2.1 程序简介

程序主要分为6个文件，其中包含5个头文件——camera.h（相机实现）、mesh.h（模型成像到画面中）、model.h（模型读取实现）、shader.h（着色器）、stb\_image.h以及实现具体功能的main.cpp。

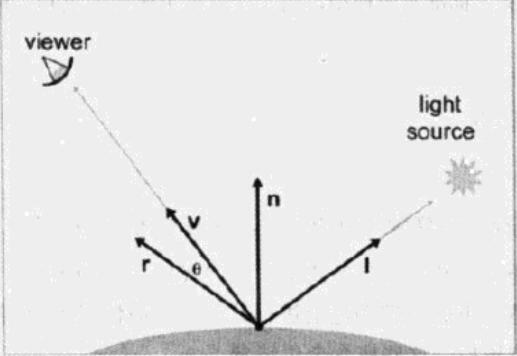
## 2.2 算法

# 算法主要分为两部分，光照模型算法和贝塞尔曲面算法

其中光照模型算法如下：

**镜面反射（Specular）**

镜面反射又称高光，是指由光源直接经物体表面反射入眼睛的光线。镜面反射使物体看上去有光泽，粗糙表面因反射率不高，所以缺乏此效果。镜面反射的强度取决于物体，光源和观察者。

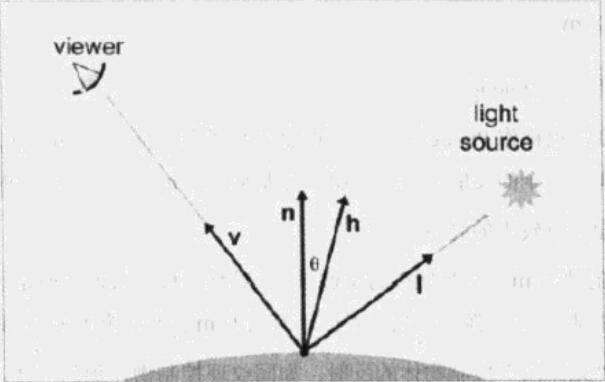
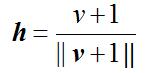


n为表面法向量  
 v 指向观察者  
 l 指向光源  
 r 为"镜像"向量，即l对n镜像的结果。  
 θ 为r和v的夹角，由r·v给出。描述镜像的方位性。  
 以上所有向量均为单位向量。  
  
  
下列等式给出给出镜面反射的Phong模型。

IMG_257

mgls为材料的光泽度，也称为Phong指数。它控制"亮斑"的范围，小的mgls带来大而平滑的光滑的光斑，大的值带来小而亮的光斑。  
mspec为材料的反射颜色，控制光斑的强度。强反射面有，有较大的mspec值，粗糙些的表面则有较小的mspec值。  
Sspec是光源的镜面反射颜色，控制光本身的色彩和强度。常与光的漫反射颜色Sdiff

由n是变化的，导致r需要频繁计算，在观察者离物体的距离远大于物体的尺寸的情况下，可以尽量避免计算r值。  
下面介绍一下Blinn模型

  
Blinn模型通过计算一个稍微不同的角度来避免计算，使用的是h，表示v,l的中间向量，由标准化的v,l的平均值求得，计算如下:  
  
  
Blinn模型和Phong模型类似，注意θ表示是n与h的夹角。  
IMG_260

# **漫反射光照（Diffuse）**

漫反射是物体粗糙表面散开的随机方向上的反射。  
漫反射光照服从Lambert法则:反射光强正比于法向量与光线夹角的余弦，用点乘来计算余弦，公式如下:  
IMG_256  
  
 n为表面法向量，l为指向光源的单位向量，mdiff为材料的散射色，即多数人认同的物体颜色，Sdiff为光源散射色，一般和光源镜面色Sspec。

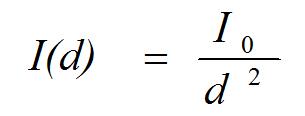
# ****环境光照(Ambient)****

镜面反射和漫反射都是刻画光源经物体反射后直接进入眼中的光线。对于光线经常经过多次反射进入眼睛的反射，称为环境光。例如，黑暗的厨房打开冰箱，整个房间会亮些，尽管箱门(或身体)挡住了大部分直线光。环境光取决于材质和全局环境光，下面的公式计算环境光:  
  
IMG_256

****mamb****为材质的环境光分量。它总是等于漫反射分量。****gamb****为整个场景的环境光照值。

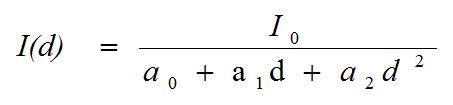
# ****光照衰弱(Attenuation)****

从物理学上来说，光照强度的衰弱为光源的光照强度除以当前距离的平方。公式如下：



其中是I0距离光源d=1时的光照强度。

但对于3D图形编程来说，一般不使用这公式，选择下面的公式较多，而且便于控制效果。



其中a0、a1和a2衰减参数。通过调整这个三个参数的大小可以实现不同的光强衰减的效果。

结合光照模型，会有如下公式：IMG_260

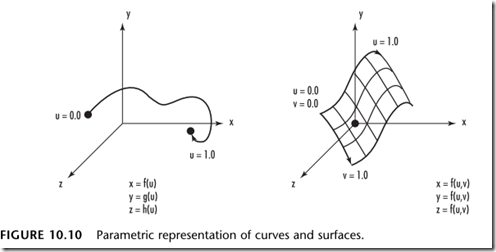
其中D为漫反射的光强，S为高光的光强，k为相应的参数的系数

**贝塞尔曲面：**

OpenGl定义一条曲线时，也把它定义为一个曲线方程。我们把这条曲线的参数成为u,它的值域就是曲线的定义域。曲面则需要u和v两个参数来描述。注意，u和v参数只表示了描述曲线的参数方程的范围，它们并没有反映实际的坐标值。其坐标可以表示为:

x = f(u); y = g(u); z = h(u);

如下图：

[](http://static.oschina.net/uploads/img/201312/11205612_O72t.png)

创建一个贝塞尔曲面与创建一个贝塞尔曲线类似。除了给出u的定义域之外，还要给出v的定义域。

在这里我们用glMap2f替换了之前的glMap1f, 这个函数指定了u和v两个域上的点。除了指定u的上界和下界之外，还要指定v的上界和下界。v定义域内点的距离是9，因为这里使用了3维数组，包含了3个u值，每个u值又包含了3个点，3x3=9。然后指定v方向上的阶，即每个u分支上v方向有多少个点。最后一个参数是指向控制点的指针。

然后我们设置求值器.

//启用求值器

glEnable(GL\_MAP2\_VERTEX\_3);  
//从0到10映射一个包含10个点的网格

glMapGrid2f(10,0.0f,10.0f,10,0.0f,10.0f);

计算网格网格表面，用线的方式表示。

// 计算网格  
  glEvalMesh2(GL\_LINE,0,10,0,10);

定义控制点：

float ctrl\_points[] = {

-2., 2., 1.2,

-1., 2., 1.,

0., 2., 0.5,

1., 2., 1.,

2., 2., 1.2,

-2., 1., 0.,

-1., 1., 0.,

0., 1., 0.,

1., 1., 0.,

2., 1., 0.,

-2., 0., 0.,

-1., 0., 0.,

0., 0., -0.5,

1., 0., 0.,

2., 0., 0.,

-2., -1., 0.,

-1., -1., 0.,

0., -1., 0.,

1., -1., 0.,

2., -1., 0.,

-2., -2., 1.2,

-1., -2., 1.,

0., -2., 0.5,

1., -2., 1.,

2., -2., 1.2

};

## 2.3 实现细节

**实现细节将分文件进行描述，首先是实现相机的camera.h头文件。**

#include <vector>

#define GLEW\_STATIC

#include <GL/glew.h>

#include <glm/glm.hpp>

#include <glm/gtc/matrix\_transform.hpp>

// Defines several possible options for camera movement. Used as abstraction to stay away from window-system specific input methods

enum Camera\_Movement {

FORWARD,

BACKWARD,

LEFT,

RIGHT

};

// Default camera values

const GLfloat YAW = -90.0f;

const GLfloat PITCH = 0.0f;

const GLfloat SPEED = 1.2f;

const GLfloat SENSITIVTY = 0.04f;

const GLfloat ZOOM = 45.0f;

// An abstract camera class that processes input and calculates the corresponding Eular Angles, Vectors and Matrices for use in OpenGL

class Camera

{

public:

// Camera Attributes

glm::vec3 Position;

glm::vec3 Front;

glm::vec3 Up;

glm::vec3 Right;

glm::vec3 WorldUp;

// Eular Angles

GLfloat Yaw;

GLfloat Pitch;

// Camera options

GLfloat MovementSpeed;

GLfloat MouseSensitivity;

GLfloat Zoom;

// Constructor with vectors

Camera(glm::vec3 position = glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f), GLfloat yaw = YAW, GLfloat pitch = PITCH) : Front(glm::vec3(0.0f, 0.0f, -1.0f)), MovementSpeed(SPEED), MouseSensitivity(SENSITIVTY), Zoom(ZOOM)

{

this->Position = position;

this->WorldUp = up;

this->Yaw = yaw;

this->Pitch = pitch;

this->updateCameraVectors();

}

// Constructor with scalar values

Camera(GLfloat posX, GLfloat posY, GLfloat posZ, GLfloat upX, GLfloat upY, GLfloat upZ, GLfloat yaw, GLfloat pitch) : Front(glm::vec3(0.0f, 0.0f, -1.0f)), MovementSpeed(SPEED), MouseSensitivity(SENSITIVTY), Zoom(ZOOM)

{

this->Position = glm::vec3(posX, posY, posZ);

this->WorldUp = glm::vec3(upX, upY, upZ);

this->Yaw = yaw;

this->Pitch = pitch;

this->updateCameraVectors();

}

// Returns the view matrix calculated using Eular Angles and the LookAt Matrix

glm::mat4 GetViewMatrix()

{

return glm::lookAt(this->Position, this->Position + this->Front, this->Up);

}

// Processes input received from any keyboard-like input system. Accepts input parameter in the form of camera defined ENUM (to abstract it from windowing systems)

void ProcessKeyBoard(Camera\_Movement direction, GLfloat deltaTime)

{

GLfloat velocity = this->MovementSpeed \* deltaTime;

if (direction == FORWARD)

this->Position += this->Front \* velocity;

if (direction == BACKWARD)

this->Position -= this->Front \* velocity;

if (direction == LEFT)

this->Position -= this->Right \* velocity;

if (direction == RIGHT)

this->Position += this->Right \* velocity;

}

// Processes input received from a mouse input system. Expects the offset value in both the x and y direction.

void ProcessMouseMovement(GLfloat xoffset, GLfloat yoffset, GLboolean constrainPitch = true)

{

xoffset \*= this->MouseSensitivity;

yoffset \*= this->MouseSensitivity;

this->Yaw += xoffset;

this->Pitch += yoffset;

// Make sure that when pitch is out of bounds, screen doesn't get flipped

if (constrainPitch)

{

if (this->Pitch > 89.0f)

this->Pitch = 89.0f;

if (this->Pitch < -89.0f)

this->Pitch = -89.0f;

}

// Update Front, Right and Up Vectors using the updated Eular angles

this->updateCameraVectors();

}

// Processes input received from a mouse scroll-wheel event. Only requires input on the vertical wheel-axis

void ProcessMouseScroll(GLfloat yoffset)

{

if (this->Zoom >= 1.0f && this->Zoom <= 45.0f)

this->Zoom -= yoffset;

if (this->Zoom <= 1.0f)

this->Zoom = 1.0f;

if (this->Zoom >= 45.0f)

this->Zoom = 45.0f;

}

private:

// Calculates the front vector from the Camera's (updated) Eular Angles

void updateCameraVectors()

{

// Calculate the new Front vector

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);

// Also re-calculate the Right and Up vector

this->Right = glm::normalize(glm::cross(this->Front, this->WorldUp)); // Normalize the vectors, because their length gets closer to 0 the more you look up or down which results in slower movement.

this->Up = glm::normalize(glm::cross(this->Right, this->Front));

}

};

**接着是实现模型读取的model.h头文件。**

#define GLEW\_STATIC

#include <GL/glew.h>

#include <glm/glm.hpp>

#include <glm/gtc/matrix\_transform.hpp>

#define STB\_IMAGE\_IMPLEMENTATION

#include "stb\_image.h"

#include <assimp/Importer.hpp>

#include <assimp/scene.h>

#include <assimp/postprocess.h>

#include "mesh.h"

#include "shader.h"

#include <string>

#include <fstream>

#include <sstream>

#include <iostream>

#include <map>

#include <vector>

unsigned int TextureFromFile(const char\* path, const string& directory, bool gamma = false);

class Model {

public:

//模型数据

vector<Texture> textures\_loaded;

vector<Mesh> meshes;

string directory;

bool gammaCorrection;

//构造函数

Model(string const& path, bool gamma = false) : gammaCorrection(gamma) {

loadModel(path);

}

//遍历网格，调用其构造函数

void Draw(Shader& shader, bool vertexMode) {

for (unsigned int i = 0; i < meshes.size(); i++)

meshes[i].Draw(shader, vertexMode);

}

private:

void loadModel(string const& path) {

//通过ASSIMP读入模型

Assimp::Importer importer;

const aiScene\* scene = importer.ReadFile(path, aiProcess\_Triangulate | aiProcess\_GenSmoothNormals | aiProcess\_FlipUVs | aiProcess\_CalcTangentSpace);

//检查错误

if (!scene || scene->mFlags & AI\_SCENE\_FLAGS\_INCOMPLETE || !scene->mRootNode) {

cout << "ERROR::ASSIMP:: " << importer.GetErrorString() << endl;

return;

}

//获取文件路径的目录路径

directory = path.substr(0, path.find\_last\_of('/'));

//从根节点开始处理

processNode(scene->mRootNode, scene);

}

void processNode(aiNode\* node, const aiScene\* scene) {

//处理当前节点中的所有网格

for (unsigned int i = 0; i < node->mNumMeshes; i++) {

aiMesh\* mesh = scene->mMeshes[node->mMeshes[i]];

meshes.push\_back(processMesh(mesh, scene));

}

//接着处理当前节点的子节点

for (unsigned int i = 0; i < node->mNumChildren; i++) {

processNode(node->mChildren[i], scene);

}

}

Mesh processMesh(aiMesh\* mesh, const aiScene\* scene) {

//需要填充的数据

vector<Vertex> vertices;

vector<unsigned int> indices;

vector<Texture> textures;

Material material\_st;

//遍历当前网格的所有节点

for (unsigned int i = 0; i < mesh->mNumVertices; i++) {

Vertex vertex;

glm::vec3 vector;

//位置

vector.x = mesh->mVertices[i].x;

vector.y = mesh->mVertices[i].y;

vector.z = mesh->mVertices[i].z;

vertex.Position = vector;

//法向量

if (mesh->HasNormals()) {

vector.x = mesh->mNormals[i].x;

vector.y = mesh->mNormals[i].y;

vector.z = mesh->mNormals[i].z;

vertex.Normal = vector;

}

//纹理坐标

//检查网格是否包含纹理坐标

if (mesh->mTextureCoords[0]) {

glm::vec2 vec;

//Assimp允许一个模型在一个顶点上有最多8个不同的纹理坐标

//暂时仅设置第一组纹理坐标

vec.x = mesh->mTextureCoords[0][i].x;

vec.y = mesh->mTextureCoords[0][i].y;

vertex.TexCoords = vec;

vector.x = mesh->mTangents[i].x;

vector.y = mesh->mTangents[i].y;

vector.z = mesh->mTangents[i].z;

vertex.Tangent = vector;

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);

}

//遍历所有的面并将其索引储存到indices这个vector中

for (unsigned int i = 0; i < mesh->mNumFaces; i++) {

aiFace face = mesh->mFaces[i];

for (unsigned int j = 0; j < face.mNumIndices; j++)

indices.push\_back(face.mIndices[j]);

}

//处理材质

if (mesh->mMaterialIndex >= 0) {

aiMaterial\* material = scene->mMaterials[mesh->mMaterialIndex];

//漫反射贴图

if (material->GetTextureCount(aiTextureType\_DIFFUSE) > 0) {

vector<Texture> diffuseMaps = loadMaterialTextures(material, aiTextureType\_DIFFUSE, "texture\_diffuse");

textures.insert(textures.end(), diffuseMaps.begin(), diffuseMaps.end());

}

//2.镜面贴图

if (material->GetTextureCount(aiTextureType\_SPECULAR) > 0) {

vector<Texture> specularMaps = loadMaterialTextures(material, aiTextureType\_SPECULAR, "texture\_specular");

textures.insert(textures.end(), specularMaps.begin(), specularMaps.end());

}

//3.法线贴图

std::vector<Texture> normalMaps = loadMaterialTextures(material, aiTextureType\_HEIGHT, "texture\_normal");

textures.insert(textures.end(), normalMaps.begin(), normalMaps.end());

aiColor3D dif(1.f, 1.f, 1.f);

aiColor3D amb(1.f, 1.f, 1.f);

aiColor3D spec(0.6f, 0.6f, 0.6f);

float shininess = 0.0;

material->Get(AI\_MATKEY\_COLOR\_AMBIENT, amb);

material->Get(AI\_MATKEY\_COLOR\_DIFFUSE, dif);

material->Get(AI\_MATKEY\_COLOR\_SPECULAR, spec);

material->Get(AI\_MATKEY\_SHININESS, shininess);

material\_st.ambient = glm::vec3(amb.r, amb.g, amb.b);

material\_st.diffuse = glm::vec3(dif.r, dif.g, dif.b);

material\_st.specular = glm::vec3(spec.r, spec.g, spec.b);

material\_st.shininess = shininess;

}

return Mesh(vertices, indices, textures, material\_st);

}

//检查所有给定材质纹理，并将没有加载的纹理加载进来

vector<Texture> loadMaterialTextures(aiMaterial\* mat, aiTextureType type, string typeName) {

vector<Texture> textures;

for (unsigned int i = 0; i < mat->GetTextureCount(type); i++) {

aiString str;

mat->GetTexture(type, i, &str);

//检查纹理是否已加载，若已加载，则将flag skip设置为真跳过该次循环

bool skip = false;

for (unsigned int j = 0; j < textures\_loaded.size(); j++) {

if (std::strcmp(textures\_loaded[j].path.data(), str.C\_Str()) == 0) {

textures.push\_back(textures\_loaded[j]);

skip = true;

break;

}

}

//如果纹理并未被加载，进行加载

if (!skip) {

Texture texture;

texture.id = TextureFromFile(str.C\_Str(), this->directory);

texture.type = typeName;

texture.path = str.C\_Str();

textures.push\_back(texture);

textures\_loaded.push\_back(texture);

}

}

return textures;

}

};

unsigned int TextureFromFile(const char\* path, const string& directory, bool gamma) {

string filename = string(path);

filename = filename;

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);

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;

}

**接着是实现着色器的shader.h头文件。**

#ifndef SHADER\_H

#define SHADER\_H

#define GLEW\_STATIC

#include <GL/glew.h>

#include <glm/glm.hpp>

#include <string>

#include <fstream>

#include <sstream>

#include <iostream>

class Shader {

public:

unsigned int ID;

Shader() {

}

//构造器读取并构建shader

void setupShader(const GLchar\* vertexPath, const GLchar\* fragmentPath, const GLchar\* geometryPath = nullptr) {

//从文件路径中获取顶点/片段着色器

std::string vertexCode;

std::string fragmentCode;

std::string geometryCode;

std::ifstream vShaderFile;

std::ifstream fShaderFile;

std::ifstream gShaderFile;

vShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);

fShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);

gShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);

try {

vShaderFile.open(vertexPath);

fShaderFile.open(fragmentPath);

std::stringstream vShaderStream, fShaderStream;

vShaderStream << vShaderFile.rdbuf();

fShaderStream << fShaderFile.rdbuf();

vShaderFile.close();

fShaderFile.close();

vertexCode = vShaderStream.str();

fragmentCode = fShaderStream.str();

// 如果存在几何着色器的路径，同样加载它

if (geometryPath != nullptr)

{

gShaderFile.open(geometryPath);

std::stringstream gShaderStream;

gShaderStream << gShaderFile.rdbuf();

gShaderFile.close();

geometryCode = gShaderStream.str();

}

}

catch (std::ifstream::failure e) {

std::cout << "ERROR::SHADER::FILE\_NOT\_SUCCESFULLY\_READ" << std::endl;

}

const char\* vShaderCode = vertexCode.c\_str();

const char\* fShaderCode = fragmentCode.c\_str();

//编译着色器

unsigned int vertex, fragment;

int success;

char infoLog[512];

//顶点着色器

vertex = glCreateShader(GL\_VERTEX\_SHADER);

glShaderSource(vertex, 1, &vShaderCode, NULL);

glCompileShader(vertex);

checkCompileErrors(vertex, "VERTEX");

//片段着色器

fragment = glCreateShader(GL\_FRAGMENT\_SHADER);

glShaderSource(fragment, 1, &fShaderCode, NULL);

glCompileShader(fragment);

checkCompileErrors(fragment, "FRAGMENT");

//如果给定一个几何着色器，对其进行编译

unsigned int geometry;

if (geometryPath != nullptr)

{

const char\* gShaderCode = geometryCode.c\_str();

geometry = glCreateShader(GL\_GEOMETRY\_SHADER);

glShaderSource(geometry, 1, &gShaderCode, NULL);

glCompileShader(geometry);

checkCompileErrors(geometry, "GEOMETRY");

}

//着色器程序

ID = glCreateProgram();

glAttachShader(ID, vertex);

glAttachShader(ID, fragment);

if (geometryPath != nullptr) {

glAttachShader(ID, geometry);

}

glLinkProgram(ID);

checkCompileErrors(ID, "PROGRAM");

//删除着色器

glDeleteShader(vertex);

glDeleteShader(fragment);

if (geometryPath != nullptr) {

glDeleteShader(geometry);

}

}

void setupShader(const GLchar\* vertexPath, const GLchar\* controlPath, const GLchar\* evaluationPath, const GLchar\* fragmentPath) {

//从文件路径中获取着色器

std::string vertexCode;

std::string controlCode;

std::string evaluationCode;

std::string fragmentCode;

std::ifstream vShaderFile;

std::ifstream tcShaderFile;

std::ifstream teShaderFile;

std::ifstream fShaderFile;

vShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);

tcShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);

teShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);

fShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);

try {

vShaderFile.open(vertexPath);

tcShaderFile.open(controlPath);

teShaderFile.open(evaluationPath);

fShaderFile.open(fragmentPath);

std::stringstream vShaderStream, tcShaderStream, teShaderStream, fShaderStream;

vShaderStream << vShaderFile.rdbuf();

tcShaderStream << tcShaderFile.rdbuf();

teShaderStream << teShaderFile.rdbuf();

fShaderStream << fShaderFile.rdbuf();

vShaderFile.close();

tcShaderFile.close();

teShaderFile.close();

fShaderFile.close();

vertexCode = vShaderStream.str();

controlCode = tcShaderStream.str();

evaluationCode = teShaderStream.str();

fragmentCode = fShaderStream.str();

}

catch (std::ifstream::failure e) {

std::cout << "ERROR::SHADER::FILE\_NOT\_SUCCESFULLY\_READ" << std::endl;

}

const char\* vShaderCode = vertexCode.c\_str();

const char\* tcShaderCode = controlCode.c\_str();

const char\* teShaderCode = evaluationCode.c\_str();

const char\* fShaderCode = fragmentCode.c\_str();

//编译着色器

unsigned int vertex, control, evaluation, fragment;

int success;

char infoLog[512];

vertex = glCreateShader(GL\_VERTEX\_SHADER);

glShaderSource(vertex, 1, &vShaderCode, NULL);

glCompileShader(vertex);

checkCompileErrors(vertex, "VERTEX");

control = glCreateShader(GL\_TESS\_CONTROL\_SHADER);

glShaderSource(control, 1, &tcShaderCode, NULL);

glCompileShader(control);

checkCompileErrors(control, "TESS\_CONTROL");

evaluation = glCreateShader(GL\_TESS\_EVALUATION\_SHADER);

glShaderSource(evaluation, 1, &teShaderCode, NULL);

glCompileShader(evaluation);

checkCompileErrors(evaluation, "TESS\_EVALUATION");

fragment = glCreateShader(GL\_FRAGMENT\_SHADER);

glShaderSource(fragment, 1, &fShaderCode, NULL);

glCompileShader(fragment);

checkCompileErrors(fragment, "FRAGMENT");

//着色器程序

ID = glCreateProgram();

glAttachShader(ID, vertex);

glAttachShader(ID, control);

glAttachShader(ID, evaluation);

glAttachShader(ID, fragment);

glLinkProgram(ID);

checkCompileErrors(ID, "PROGRAM");

//删除着色器

glDeleteShader(vertex);

glDeleteShader(control);

glDeleteShader(evaluation);

glDeleteShader(fragment);

}

//激活程序

void use() {

glUseProgram(ID);

}

//uniform工具函数

void setBool(const std::string& name, bool value) const {

glUniform1i(glGetUniformLocation(ID, name.c\_str()), (int)value);

}

void setInt(const std::string& name, int value) const {

glUniform1i(glGetUniformLocation(ID, name.c\_str()), value);

}

void setFloat(const std::string& name, float value) const {

glUniform1f(glGetUniformLocation(ID, name.c\_str()), value);

}

void setVec2(const std::string& name, const glm::vec2& value) const {

glUniform2fv(glGetUniformLocation(ID, name.c\_str()), 1, &value[0]);

}

void setVec2(const std::string& name, float x, float y) const {

glUniform2f(glGetUniformLocation(ID, name.c\_str()), x, y);

}

void setVec3(const std::string& name, const glm::vec3& value) const {

glUniform3fv(glGetUniformLocation(ID, name.c\_str()), 1, &value[0]);

}

void setVec3(const std::string& name, float x, float y, float z) const {

glUniform3f(glGetUniformLocation(ID, name.c\_str()), x, y, z);

}

void setVec4(const std::string& name, const glm::vec4& value) const {

glUniform4fv(glGetUniformLocation(ID, name.c\_str()), 1, &value[0]);

}

void setVec4(const std::string& name, float x, float y, float z, float w) {

glUniform4f(glGetUniformLocation(ID, name.c\_str()), x, y, z, w);

}

void setMat2(const std::string& name, const glm::mat2& mat) const {

glUniformMatrix2fv(glGetUniformLocation(ID, name.c\_str()), 1, GL\_FALSE, &mat[0][0]);

}

void setMat3(const std::string& name, const glm::mat3& mat) const {

glUniformMatrix3fv(glGetUniformLocation(ID, name.c\_str()), 1, GL\_FALSE, &mat[0][0]);

}

void setMat4(const std::string& name, const glm::mat4& mat) const {

glUniformMatrix4fv(glGetUniformLocation(ID, name.c\_str()), 1, GL\_FALSE, &mat[0][0]);

}

private:

void checkCompileErrors(GLuint shader, std::string type) {

GLint success;

GLchar infoLog[1024];

if (type != "PROGRAM")

{

glGetShaderiv(shader, GL\_COMPILE\_STATUS, &success);

if (!success)

{

glGetShaderInfoLog(shader, 1024, NULL, infoLog);

std::cout << "ERROR::SHADER\_COMPILATION\_ERROR of type: " << type << "\n" << infoLog << "\n -- --------------------------------------------------- -- " << std::endl;

}

}

else

{

glGetProgramiv(shader, GL\_LINK\_STATUS, &success);

if (!success)

{

glGetProgramInfoLog(shader, 1024, NULL, infoLog);

std::cout << "ERROR::PROGRAM\_LINKING\_ERROR of type: " << type << "\n" << infoLog << "\n -- --------------------------------------------------- -- " << std::endl;

}

}

}

};

#endif

**接着是处理顶点数据的mesh.h头文件。**

#ifndef MESH\_H

#define MESH\_H

#include <string>

#include <vector>

#define GLEW\_STATIC

#include <GL/glew.h>

#include <glm/glm.hpp>

#include <glm/gtc/matrix\_transform.hpp>

#include "shader.h"

using namespace std;

struct Vertex {

glm::vec3 Position;

glm::vec3 Normal;

glm::vec2 TexCoords;

glm::vec3 Tangent;

glm::vec3 Bitangent;

};

enum ETextureType {

ETextureTypeDiffuse = 1, //漫反射

ETextureTypeSpecular = 2 //镜面反射

};

struct Texture {

unsigned int id;

string type;

string path;

};

struct Material {

glm::vec3 ambient;

glm::vec3 diffuse;

glm::vec3 specular;

float shininess;

};

class Mesh {

public:

//网格数据

vector<Vertex> vertices;

vector<unsigned int> indices;

vector<Texture> textures;

Material material;

//构造器

Mesh(vector<Vertex> vertices, vector<unsigned int> indices, vector<Texture> textures, Material material) {

this->vertices = vertices;

this->indices = indices;

this->textures = textures;

this->material = material;

setupMesh();

}

void Draw(Shader &shader, bool vertexMode) {

unsigned int diffuseNr = 1;

unsigned int specularNr = 1;

unsigned int normalNr = 1;

unsigned int heightNr = 1;

for (unsigned int i = 0; i < textures.size(); i++) {

// 在绑定之前激活相应的纹理单元

glActiveTexture(GL\_TEXTURE0 + i);

// 获取纹理序号

string number;

string name = textures[i].type;

if (name == "texture\_diffuse") {

number = to\_string(diffuseNr++);

}

else if (name == "texture\_specular") {

number = to\_string(specularNr++);

}

else if (name == "texture\_normal") {

number = std::to\_string(normalNr++);

}

else if (name == "texture\_height") {

number = std::to\_string(heightNr++);

}

glUniform1i(glGetUniformLocation(shader.ID, (name + number).c\_str()), i);

glBindTexture(GL\_TEXTURE\_2D, textures[i].id);

}

shader.setVec3("material.ambient", this->material.ambient);

shader.setVec3("material.diffuse", this->material.diffuse);

shader.setVec3("material.specular", this->material.specular);

shader.setFloat("material.shininess", this->material.shininess);

// 绘制网格

glBindVertexArray(VAO);

if (!vertexMode) {

glDrawElements(GL\_TRIANGLES, indices.size(), GL\_UNSIGNED\_INT, 0);

}

else {

glPointSize(8.0f);

glDrawArrays(GL\_POINTS, 0, this->vertices.size());

}

glBindVertexArray(0);

}

private:

unsigned int VAO, VBO, EBO;

void setupMesh() {

glGenVertexArrays(1, &VAO);

glGenBuffers(1, &VBO);

glGenBuffers(1, &EBO);

glBindVertexArray(VAO);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glBufferData(GL\_ARRAY\_BUFFER, vertices.size() \* sizeof(Vertex), &vertices[0], GL\_STATIC\_DRAW);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, indices.size() \* sizeof(unsigned int), &indices[0], GL\_STATIC\_DRAW);

// 顶点位置

glEnableVertexAttribArray(0);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (void\*)0);

// 顶点法线

glEnableVertexAttribArray(1);

glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (void\*)offsetof(Vertex, Normal));

// 顶点纹理坐标

glEnableVertexAttribArray(2);

glVertexAttribPointer(2, 2, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (void\*)offsetof(Vertex, TexCoords));

// 顶点切线

glEnableVertexAttribArray(3);

glVertexAttribPointer(3, 3, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (void\*)offsetof(Vertex, Tangent));

// 顶点余切线

glEnableVertexAttribArray(4);

glVertexAttribPointer(4, 3, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (void\*)offsetof(Vertex, Bitangent));

glBindVertexArray(0);

}

};

#endif

**接着是主程序main.cpp头文件。**

**#include <string>**

**#define GLEW\_STATIC**

**#include <GL/glew.h>**

**#include <GLFW/glfw3.h>**

**#include <glm/glm.hpp>**

**#include <glm/gtc/matrix\_transform.hpp>**

**#include <glm/gtc/type\_ptr.hpp>**

**#include "shader.h"**

**#include "Camera.h"**

**#include "model.h"**

**#include <iostream>**

**void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height);**

**void mouse\_callback(GLFWwindow\* window, double xpos, double ypos);**

**void mouse\_button\_callback(GLFWwindow\* window, int button, int action, int mods);**

**void scroll\_callback(GLFWwindow\* window, double xoffset, double yoffset);**

**void processInput(GLFWwindow\* window);**

**/\*窗口初始设置\*/**

**const unsigned int SCR\_WIDTH = 800;**

**const unsigned int SCR\_HEIGHT = 600;**

**/\*相机\*/**

**Camera camera(glm::vec3(-3.0f, 0.0f, 20.0f));**

**float lastX = SCR\_WIDTH / 2.0f;**

**float lastY = SCR\_HEIGHT / 2.0f;**

**/\*着色器\*/**

**Shader ctrl\_shader;**

**Shader surface\_shader;**

**Shader model\_shader;**

**Shader vertex\_shader;**

**/\*计时（鼠标输入相关）\*/**

**float deltaTime = 0.0f;**

**float lastFrame = 0.0f;**

**//鼠标**

**bool mouse\_left\_button = false;**

**bool mouse\_right\_button = false;**

**/\*变换相关\*/**

**glm::mat4 model\_surface = glm::scale(glm::mat4(1.0f), glm::vec3(1.2f, 1.2f, 1.2f));**

**glm::mat4 model\_model = glm::scale(glm::translate(glm::mat4(1.0f), glm::vec3(7.0f, 3.0f, 0.0f)), glm::vec3(0.01f, 0.01f, 0.01f));**

**float movement\_speed\_model = 4.0f;**

**float mouse\_sensitivity\_model = 0.2;**

**glm::mat4 rot = glm::mat4(1.0);**

**/\*控制模型/曲面\*/**

**bool control\_model = true;**

**bool control\_surface = false;**

**/\*线框/顶点模式\*/**

**bool wireMode = false;**

**bool vertexMode = false;**

**/\*曲面细分等级\*/**

**float level = 200.0f;**

**int main() {**

**/\*-------------------初始化-------------------\*/**

**glfwInit();**

**glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 4);**

**glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 1);**

**glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);**

**GLFWwindow\* window = glfwCreateWindow(SCR\_WIDTH, SCR\_HEIGHT, "Project2-3", NULL, NULL);**

**if (window == NULL) {**

**std::cout << "Failed to create GLFW window" << std::endl;**

**glfwTerminate();**

**return -1;**

**}**

**glfwMakeContextCurrent(window);**

**glfwSetFramebufferSizeCallback(window, framebuffer\_size\_callback);**

**glfwSetCursorPosCallback(window, mouse\_callback);**

**glfwSetMouseButtonCallback(window, mouse\_button\_callback);**

**glfwSetScrollCallback(window, scroll\_callback);**

**glewExperimental = GL\_TRUE;**

**glewInit();**

**Model model\_to\_load("E:\\vs source\\repos\\Assignment2\\Assignment2\\model\\111.obj");**

**ctrl\_shader.setupShader("E:\\vs source\\repos\\Assignment2\\Assignment2\\shader\\surface.vs.glsl", "E:\\vs source\\repos\\Assignment2\\Assignment2\\shader\\vert.fs.glsl");**

**surface\_shader.setupShader("E:\\vs source\\repos\\Assignment2\\Assignment2\\shader\\surface.vs.glsl", "E:\\vs source\\repos\\Assignment2\\Assignment2\\shader\\surface.tcs.glsl",**

**"E:\\vs source\\repos\\Assignment2\\Assignment2\\shader\\surface.tes.glsl", "E:\\vs source\\repos\\Assignment2\\Assignment2\\shader\\texture.fs.glsl");**

**model\_shader.setupShader("E:\\vs source\\repos\\Assignment2\\Assignment2\\shader\\model\_loading.vs.glsl", "E:\\vs source\\repos\\Assignment2\\Assignment2\\shader\\model\_loading.fs.glsl");**

**vertex\_shader.setupShader("E:\\vs source\\repos\\Assignment2\\Assignment2\\shader\\model\_loading.vs.glsl", "E:\\vs source\\repos\\Assignment2\\Assignment2\\shader\\vert.fs.glsl");**

**float ctrl\_points[] = {**

**-2., 2., 1.2,**

**-1., 2., 1.,**

**0., 2., 0.5,**

**1., 2., 1.,**

**2., 2., 1.2,**

**-2., 1., 0.,**

**-1., 1., 0.,**

**0., 1., 0.,**

**1., 1., 0.,**

**2., 1., 0.,**

**-2., 0., 0.,**

**-1., 0., 0.,**

**0., 0., -0.5,**

**1., 0., 0.,**

**2., 0., 0.,**

**-2., -1., 0.,**

**-1., -1., 0.,**

**0., -1., 0.,**

**1., -1., 0.,**

**2., -1., 0.,**

**-2., -2., 1.2,**

**-1., -2., 1.,**

**0., -2., 0.5,**

**1., -2., 1.,**

**2., -2., 1.2**

**};**

**unsigned int VBO, VAO;**

**glGenVertexArrays(1, &VAO);**

**glGenBuffers(1, &VBO);**

**glBindVertexArray(VAO);**

**glBindBuffer(GL\_ARRAY\_BUFFER, VBO);**

**glBufferData(GL\_ARRAY\_BUFFER, sizeof(ctrl\_points), ctrl\_points, GL\_STATIC\_DRAW);**

**glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3 \* sizeof(float), (GLvoid\*)0);**

**glEnableVertexAttribArray(0);**

**unsigned int texture;**

**glGenTextures(1, &texture);**

**glEnable(GL\_TEXTURE\_2D);**

**glBindTexture(GL\_TEXTURE\_2D, texture);**

**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);**

**glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);**

**int width, height, num\_channels;**

**unsigned char\* data = stbi\_load("E:\\vs source\\repos\\Assignment2\\Assignment2\\texture\\111.jpg", &width, &height, &num\_channels, 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." << std::endl;**

**}**

**surface\_shader.use();**

**surface\_shader.setInt("Texture", 0);**

**stbi\_image\_free(data);**

**glEnable(GL\_DEPTH\_TEST);**

**while (!glfwWindowShouldClose(window)) {**

**float currentFrame = glfwGetTime();**

**deltaTime = currentFrame - lastFrame;**

**lastFrame = currentFrame;**

**glClearColor(0.059f, 0.816f, 0.78f, 1.0f);**

**glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);**

**glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)SCR\_WIDTH / (float)SCR\_HEIGHT, 0.1f, 100.0f);**

**glm::mat4 view = camera.GetViewMatrix();**

**surface\_shader.setVec3("lightColor", glm::vec3(1.0f, 1.0f, 1.0f));**

**surface\_shader.setMat4("inverse", glm::inverse(projection));**

**surface\_shader.setMat4("frag\_view", view);**

**surface\_shader.setMat4("frag\_model", model\_surface);**

**surface\_shader.setVec3("lightPos", glm::vec3(1.2f, 1.0f, 2.0f));**

**surface\_shader.setVec3("viewPos", camera.Position);**

**surface\_shader.setVec3("material.ambient", glm::vec3(1.0f, 1.0f, 1.0f));**

**surface\_shader.setVec3("material.diffuse", glm::vec3(1.0f, 1.0f, 1.0f));**

**surface\_shader.setVec3("material.specular", glm::vec3(0.6f, 0.6f, 0.6f));**

**surface\_shader.setFloat("material.shininess", 128.0f);**

**if (!vertexMode) {**

**glBindVertexArray(VAO);**

**surface\_shader.use();**

**surface\_shader.setFloat("outer\_x", level);**

**surface\_shader.setFloat("outer\_y", level);**

**surface\_shader.setFloat("inner\_vert", level);**

**surface\_shader.setFloat("inner\_hori", level);**

**// ±‰ªªæÿ’Û**

**surface\_shader.setMat4("projection", projection);**

**surface\_shader.setMat4("view", view);**

**surface\_shader.setMat4("model", model\_surface);**

**// Œ∆¿Ì**

**surface\_shader.setInt("Texture", 0);**

**glPolygonMode(GL\_FRONT\_AND\_BACK, wireMode ? GL\_LINE : GL\_FILL);**

**glPatchParameteri(GL\_PATCH\_VERTICES, 25);**

**glDrawArrays(GL\_PATCHES, 0, 25);**

**glBindVertexArray(0);**

**}**

**else {**

**// ªÊ÷∆øÿ÷∆µ„**

**glBindVertexArray(VAO);**

**ctrl\_shader.use();**

**ctrl\_shader.setMat4("projection", projection);**

**ctrl\_shader.setMat4("view", view);**

**ctrl\_shader.setMat4("model", model\_surface);**

**glPointSize(10.0f);**

**glDrawArrays(GL\_POINTS, 0, 25);**

**glBindVertexArray(0);**

**}**

**if (!vertexMode) {**

**// ªÊ÷∆ƒ£–Õ**

**model\_shader.use();**

**model\_shader.setMat4("projection", projection);**

**model\_shader.setMat4("view", view);**

**model\_shader.setMat4("model", model\_model);**

**// ƒ£–Õπ‚’’**

**model\_shader.setVec3("lightColor", glm::vec3(1.0f, 1.0f, 1.0f));**

**model\_shader.setMat4("inverse", glm::inverse(projection));**

**model\_shader.setMat4("fragView", view);**

**model\_shader.setVec3("lightPos", glm::vec3(1.2f, 1.0f, 2.0f));**

**model\_shader.setVec3("viewPos", camera.Position);**

**model\_to\_load.Draw(model\_shader, vertexMode);**

**}**

**else {**

**// ªÊ÷∆ƒ£–Õ∂•µ„**

**vertex\_shader.use();**

**vertex\_shader.setMat4("projection", projection);**

**vertex\_shader.setMat4("view", view);**

**vertex\_shader.setMat4("model", model\_model);**

**model\_to\_load.Draw(vertex\_shader, vertexMode);**

**}**

**processInput(window);**

**// Ωªªªª∫≥Â°¢ºÏ≤È ¬º˛¥•∑¢◊¥Ã¨º∞∏¸–¬¥∞ø⁄◊¥Ã¨**

**glfwPollEvents();**

**glfwSwapBuffers(window);**

**}**

**//  Õ∑≈◊ ‘¥**

**glDeleteVertexArrays(1, &VAO);**

**glDeleteBuffers(1, &VBO);**

**glfwTerminate();**

**glfwTerminate();**

**return 0;**

**}**

**void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height) {**

**glViewport(0, 0, width, height);**

**}**

**void mouse\_callback(GLFWwindow\* window, double xpos, double ypos) {**

**if (mouse\_right\_button) {**

**float xoffset\_c = xpos - lastX;**

**float yoffset\_c = lastY - ypos;**

**lastX = xpos;**

**lastY = ypos;**

**camera.ProcessMouseMovement(xoffset\_c, yoffset\_c);**

**}**

**if (mouse\_left\_button && control\_model) {**

**float xoffset\_m = xpos - lastX;**

**float yoffset\_m = lastY - ypos;;**

**lastX = xpos;**

**lastY = ypos;**

**float radians\_x = xoffset\_m \* mouse\_sensitivity\_model;**

**float radians\_y = -yoffset\_m \* mouse\_sensitivity\_model;**

**if (radians\_y > 89.0f) {**

**radians\_y = 89.0f;**

**}**

**if (radians\_y < -89.0f) {**

**radians\_y = -89.0f;**

**}**

**rot = glm::mat4(1.0f);**

**rot = glm::rotate(rot, glm::radians(radians\_x), glm::vec3(0.0f, 1.0f, 0.0f));**

**rot = glm::rotate(rot, glm::radians(radians\_y), glm::vec3(1.0f, 0.0f, 0.0f));**

**model\_model = model\_model \* rot;**

**}**

**if (mouse\_left\_button && control\_surface) {**

**float xoffset\_m = xpos - lastX;**

**float yoffset\_m = lastY - ypos;;**

**lastX = xpos;**

**lastY = ypos;**

**float radians\_x = xoffset\_m \* mouse\_sensitivity\_model;**

**float radians\_y = -yoffset\_m \* mouse\_sensitivity\_model;**

**if (radians\_y > 89.0f) {**

**radians\_y = 89.0f;**

**}**

**if (radians\_y < -89.0f) {**

**radians\_y = -89.0f;**

**}**

**glm::mat4 rot\_surface = glm::mat4(1.0f);**

**rot\_surface = glm::rotate(rot\_surface, glm::radians(radians\_x), glm::vec3(0.0f, 1.0f, 0.0f));**

**rot\_surface = glm::rotate(rot\_surface, glm::radians(radians\_y), glm::vec3(1.0f, 0.0f, 0.0f));**

**model\_surface = model\_surface \* rot\_surface;**

**}**

**lastX = xpos;**

**lastY = ypos;**

**}**

**void mouse\_button\_callback(GLFWwindow\* window, int button, int action, int mods) {**

**if (action == GLFW\_PRESS) {**

**switch (button) {**

**case GLFW\_MOUSE\_BUTTON\_LEFT:**

**mouse\_left\_button = true;**

**break;**

**case GLFW\_MOUSE\_BUTTON\_RIGHT:**

**mouse\_right\_button = true;**

**break;**

**default:**

**return;**

**}**

**}**

**else if (action == GLFW\_RELEASE) {**

**switch (button) {**

**case GLFW\_MOUSE\_BUTTON\_LEFT:**

**mouse\_left\_button = false;**

**break;**

**case GLFW\_MOUSE\_BUTTON\_RIGHT:**

**mouse\_right\_button = false;**

**break;**

**default:**

**return;**

**}**

**}**

**}**

**//模型缩放**

**void scroll\_callback(GLFWwindow\* window, double xoffset, double yoffset) {**

**camera.ProcessMouseScroll(yoffset);**

**}**

**void processInput(GLFWwindow\* window) {**

**// πÿ±’¥∞ø⁄**

**if (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS) {**

**glfwSetWindowShouldClose(window, true);**

**}**

**// œ‡ª˙Œª÷√**

**if (glfwGetKey(window, GLFW\_KEY\_UP) == GLFW\_PRESS) {**

**camera.ProcessKeyBoard(Camera\_Movement::FORWARD, deltaTime);**

**}**

**if (glfwGetKey(window, GLFW\_KEY\_DOWN) == GLFW\_PRESS) {**

**camera.ProcessKeyBoard(Camera\_Movement::BACKWARD, deltaTime);**

**}**

**if (glfwGetKey(window, GLFW\_KEY\_LEFT) == GLFW\_PRESS) {**

**camera.ProcessKeyBoard(Camera\_Movement::LEFT, deltaTime);**

**}**

**if (glfwGetKey(window, GLFW\_KEY\_RIGHT) == GLFW\_PRESS) {**

**camera.ProcessKeyBoard(Camera\_Movement::RIGHT, deltaTime);**

**}**

**// ƒ£–Õ“∆∂Ø**

**if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS) {**

**model\_model = glm::translate(model\_model, glm::vec3(0.0f, movement\_speed\_model \* deltaTime \* 2.5f, 0.0f));**

**}**

**if (glfwGetKey(window, GLFW\_KEY\_S) == GLFW\_PRESS) {**

**model\_model = glm::translate(model\_model, glm::vec3(0.0f, -(movement\_speed\_model \* deltaTime \* 2.5f), 0.0f));**

**}**

**if (glfwGetKey(window, GLFW\_KEY\_A) == GLFW\_PRESS) {**

**model\_model = glm::translate(model\_model, glm::vec3(-(movement\_speed\_model \* deltaTime \* 2.5f), 0.0f, 0.0f));**

**}**

**if (glfwGetKey(window, GLFW\_KEY\_D) == GLFW\_PRESS) {**

**model\_model = glm::translate(model\_model, glm::vec3(movement\_speed\_model \* deltaTime \* 2.5f, 0.0f, 0.0f));**

**}**

**// «˙√Ê“∆∂Ø**

**if (glfwGetKey(window, GLFW\_KEY\_T) == GLFW\_PRESS) {**

**model\_surface = glm::translate(model\_surface, glm::vec3(0.0f, movement\_speed\_model \* deltaTime, 0.0f));**

**}**

**if (glfwGetKey(window, GLFW\_KEY\_G) == GLFW\_PRESS) {**

**model\_surface = glm::translate(model\_surface, glm::vec3(0.0f, -(movement\_speed\_model \* deltaTime), 0.0f));**

**}**

**if (glfwGetKey(window, GLFW\_KEY\_F) == GLFW\_PRESS) {**

**model\_surface = glm::translate(model\_surface, glm::vec3(-(movement\_speed\_model \* deltaTime), 0.0f, 0.0f));**

**}**

**if (glfwGetKey(window, GLFW\_KEY\_H) == GLFW\_PRESS) {**

**model\_surface = glm::translate(model\_surface, glm::vec3(movement\_speed\_model \* deltaTime, 0.0f, 0.0f));**

**}**

**// œﬂøÚƒ£ Ω**

**if (glfwGetKey(window, GLFW\_KEY\_Q) == GLFW\_PRESS) {**

**wireMode = true;**

**}**

**else {**

**wireMode = false;**

**}**

**// ∂•µ„ƒ£ Ω**

**if (glfwGetKey(window, GLFW\_KEY\_E) == GLFW\_PRESS) {**

**vertexMode = true;**

**}**

**else {**

**vertexMode = false;**

**}**

**// øÿ÷∆∂‘œÛ**

**if (glfwGetKey(window, GLFW\_KEY\_LEFT\_CONTROL) == GLFW\_PRESS) {**

**control\_model = true;**

**control\_surface = false;**

**}**

**if (glfwGetKey(window, GLFW\_KEY\_RIGHT\_CONTROL) == GLFW\_PRESS) {**

**control\_model = false;**

**control\_surface = true;**

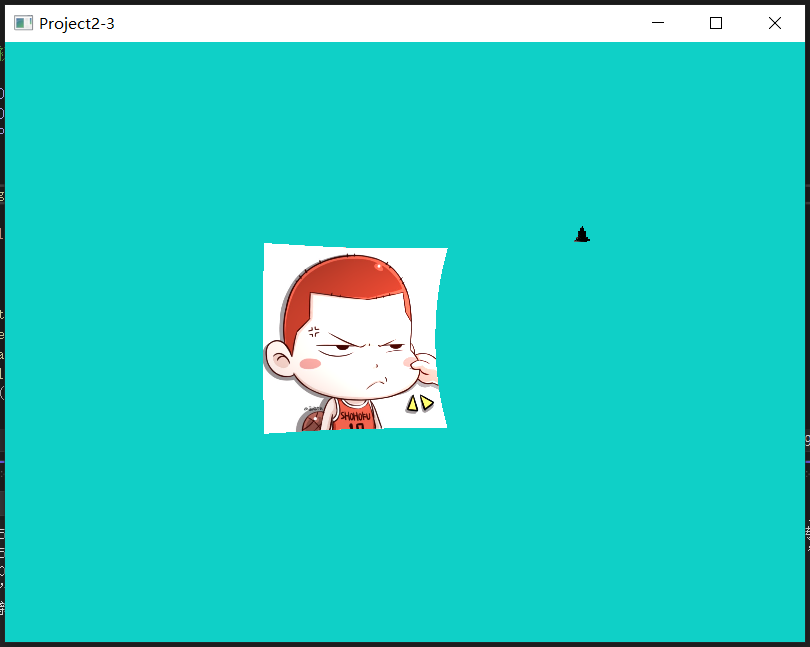
**}**

**}**

# 3. 结果与讨论

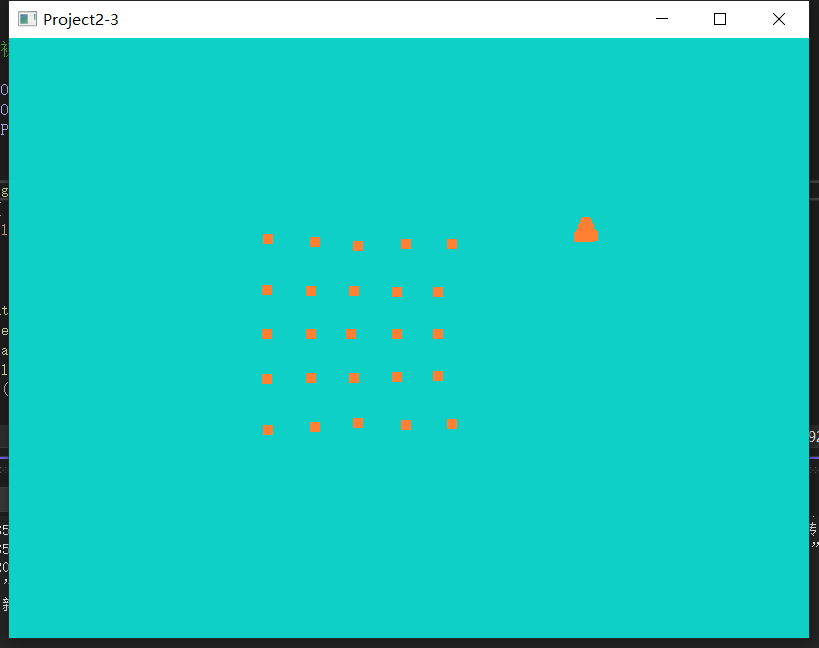
## 3.1 结果

贝塞尔曲线效果：

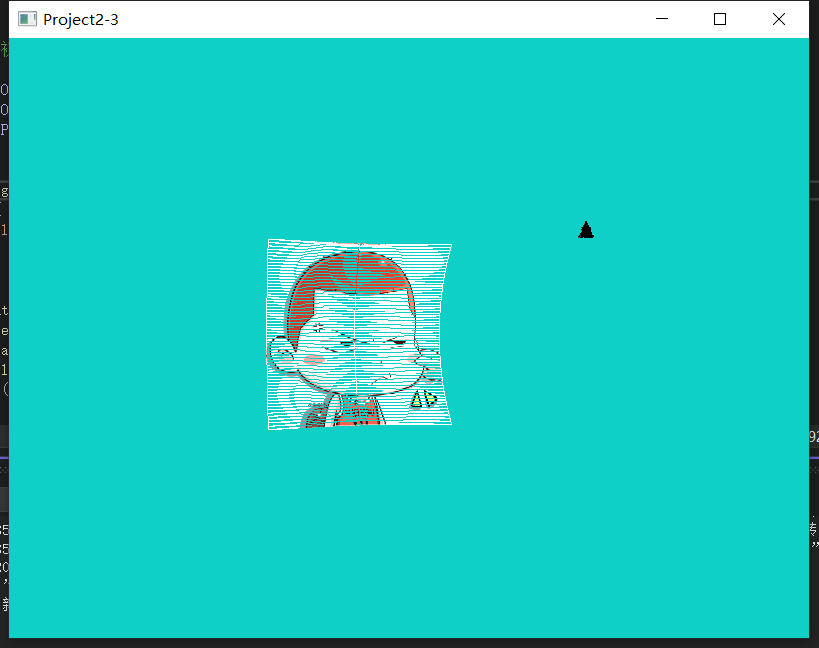


模型放在曲面旁可能太小，在曲面的右上角。

顶点模式：



线框模式:



## 3.2 讨论

在整个作业开发过程当中，首先可以分为三个阶段，第一个阶段是绘制5\*5的bezier曲面，这个在网上没有搜到很好的学习教程，所以自己还经历的不少次的尝试，虽然最终的效果可能稍微差一些，但还是勉强可以接受的。并且也在学习使用的过程中，更加深入的了解了贝塞尔曲线，用“[贝塞尔](https://baike.baidu.com/item/%E8%B4%9D%E5%A1%9E%E5%B0%94" \t "https://baike.baidu.com/item/%E8%B4%9D%E5%A1%9E%E5%B0%94%E6%9B%B2%E7%BA%BF/_blank)”工具无论是画直线或是曲线，都非常简单，随手可得。其操作特点是通过用鼠标在面板上放置各个[锚点](https://baike.baidu.com/item/%E9%94%9A%E7%82%B9" \t "https://baike.baidu.com/item/%E8%B4%9D%E5%A1%9E%E5%B0%94%E6%9B%B2%E7%BA%BF/_blank)，根据锚点的路径和描绘的先后顺序，产生直线或者是曲线的效果。我们都知道路径由一个或多个直线段或曲线段组成。锚点标记路径段的[端点](https://baike.baidu.com/item/%E7%AB%AF%E7%82%B9" \t "https://baike.baidu.com/item/%E8%B4%9D%E5%A1%9E%E5%B0%94%E6%9B%B2%E7%BA%BF/_blank)。在曲线段上，每个选中的锚点显示一条或两条[方向线](https://baike.baidu.com/item/%E6%96%B9%E5%90%91%E7%BA%BF" \t "https://baike.baidu.com/item/%E8%B4%9D%E5%A1%9E%E5%B0%94%E6%9B%B2%E7%BA%BF/_blank)，方向线以方向点结束。方向线和方向点的位置确定曲线段的大小和形状。移动这些元素将改变路径中曲线的形状，可以看右图。路径可以是[闭合](https://baike.baidu.com/item/%E9%97%AD%E5%90%88" \t "https://baike.baidu.com/item/%E8%B4%9D%E5%A1%9E%E5%B0%94%E6%9B%B2%E7%BA%BF/_blank)的，没有起点或终点（如[圆圈](https://baike.baidu.com/item/%E5%9C%86%E5%9C%88" \t "https://baike.baidu.com/item/%E8%B4%9D%E5%A1%9E%E5%B0%94%E6%9B%B2%E7%BA%BF/_blank)），也可以是开放的，有明显的端点（如波浪线）。

第二个阶段，主要是就是将两个作业相结合起来，这看起来是一件挺简单的事情，但实际操作的时候，就遇到了许多的bug，给自己作业的推进带来了不小的麻烦，而且有些问题并不知道是哪里出错了，查阅了一些资料也没有很好的结果，可能是由于一些底层的知识自己学习的还不到位，还需要更深入的学习opengl的相关知识。

第三个阶段，主要就是实现一些所要求的光照模型，包括ambient light光照模型, diffuse light光照模型 specular light光照模型。Ambient Lighting环境光照 描述的就是 在开放的空间中，物体会受到周围的 各种光源 的影响。物体可能会变暗，但是绝对不会变成纯黑。而对于diffuse light漫反射光，我们计算漫射光的时候，根据光源的色素，设为L = (0.8, 0.8, 0.8)，然后根据物体的材料特征，反射的颜色为M = (0.5, 0.5, 1.0)，然后利用 Lambert‘s Consine Law计算反射光的强度f(a) = max(cosa, 0) = max(L\*n, 0), 其中n代表表面的法线单位向量。最终得到公式计算最终颜色C = max(L\*n, 0)\*L\*M;我们平时看到的桌面或者墙壁都是平滑的，那么也就是说n法线都应该是垂直向上才对，但我们从不同角度看这个桌面，它的漫射光都是一样的，这主要由于在微观条件下表面都是粗糙的。就是如果我们使用显微镜观察一个很平滑的桌面，也会发现是凹凸不平的。从所有不同方向和角度都能观察到其粗糙程度是一样的。微观法线并不是垂直向上的，而是在不同角度，方向观察的时候都是和眼睛成一定角度的，也可以认为微观法线是固定不变的。