2023-2024学年第2学期

《三维图形程序设计》

期末作业实验报告

组长：

学号： 姓名：

成员：

学号： 姓名：

学号： 姓名：

学号： 姓名：

# 实验题目：雪中即景

|  |
| --- |
| 1. 定义并构建一个与作业主题相关的基本场景，要求至少包括3种以上的对象，如雪人、房屋、树木等； 2. 模拟真实的光照效果，可以呈现村落的白天和黑夜。 3. 自定义一种及以上的动画效果，如下雪到雪停、白天到黑夜、树枝晃动等。 4. 选作：构造一个立方体的树桩，使用它进行以下交互操作。    1. 使用鼠标左键对其进行平移操作；    2. 使用ctrl键对其进行旋转操作；    3. 使用上、下、左、右键对其进行拉伸；    4. 使用鼠标右键使其恢复初始状态。 |

# 实验要求

|  |
| --- |
| 1. 以小组为单位提交源码、实验报告以及程序结果展示视频文件，由组长提交。 2. 实验报告按照标准格式填写；程序结果展示视频文件要求包括实验结果展示以及程序解说。 3. 允许团队合作，每个小组可以由1~5人组成，小组自行推选组长。多人小组需要在实验报告中写明任务分工。 |

# 任务分工

肖枨钧100%

# 关键代码说明

## 总体思路

总体上，我是先实现了摄像机类（class Camera），用于模拟人眼。然后我实现了天空盒类（class SkyBox），用于绘制天空盒，以及实现一部分昼夜交替的效果。之后就是实现网格类和模型类（class Mesh和class Model），用于将模型导入到天空盒中。在实现了这些基本功能后，我又实现了一个玩家类（class Player），即绘制一个能与玩家交互的树桩。

在实现完这些类后就是将这些类集成在mian函数中，绘制自己想要的场景。此外我对模型光照的渲染部分也是在main.cpp的一个函数中实现。

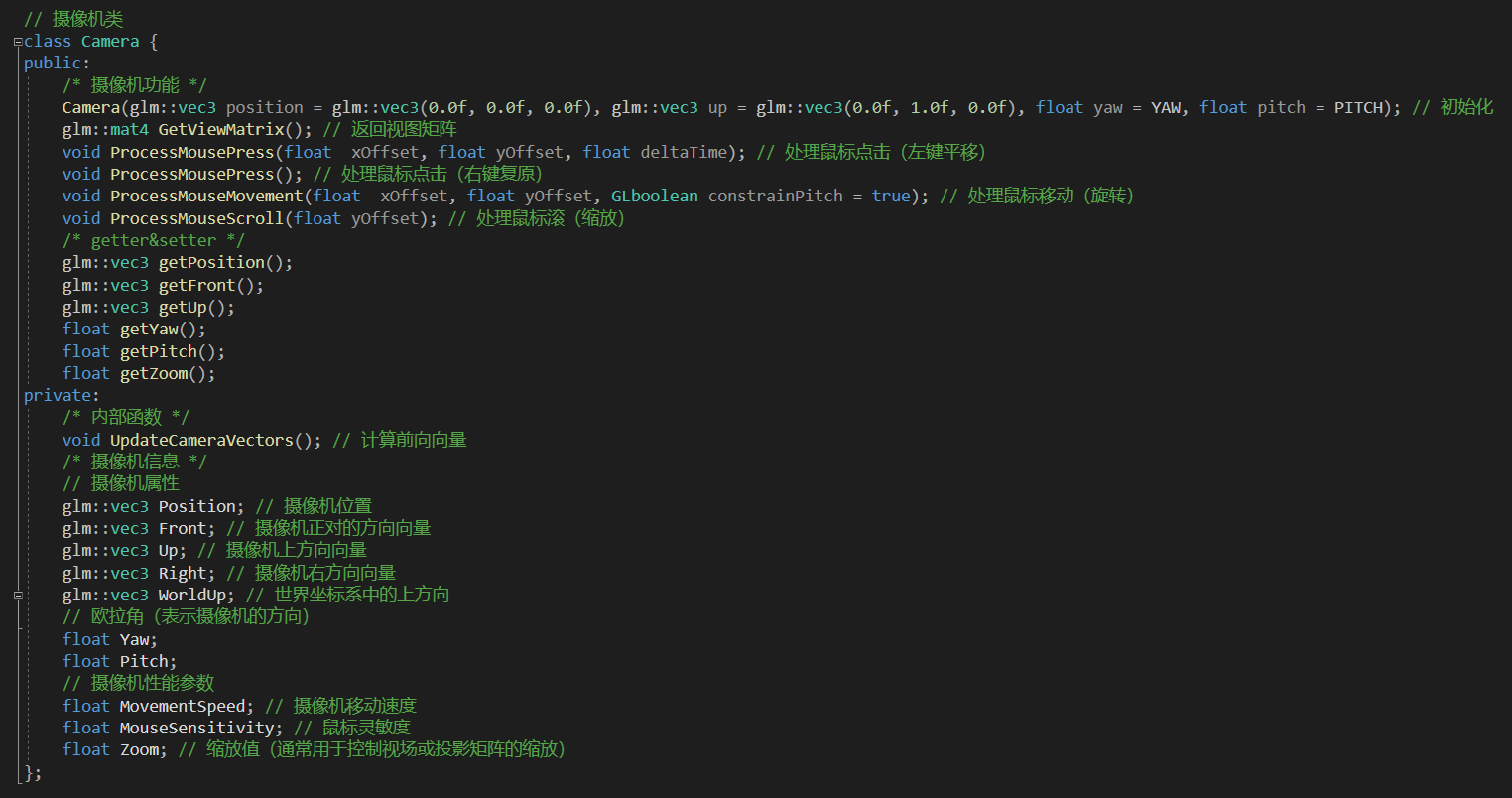
### 摄像机部分（class Camera）

为了模拟人眼效果，我首先实现了一个摄像机类，可以参考下图。

摄像机类中的成员主要可以分为两大部分。

第一部分是摄像机的成员变量，用于储存摄像机当前的状态信息。其中也能大致分为三部分，分别是摄像机位置、摄像机朝向，以及摄像机参数。前两个比较好理解，至于第三个参数，则是代表摄像机的灵敏度和焦距，通过改变这几个参数，来得到更好的使用体验。

第二部分则是摄像机的成员函数，用于实现摄像机的功能，如旋转视角、平移、复原等，具体的说明在“主要函数说明”中。



### 角色部分（class Player）

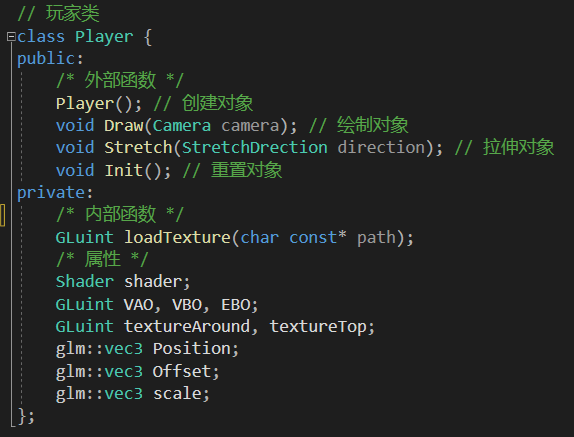
实现摄像机类后，可以简单理解为完成了第一视角的实现。但为了满足作业的要求，还需要一个可交互的树桩，效果上有点像从第一视角变为第三视角。

我将与树桩相关的操作全部封装在了玩家类中，因为我认为这个树桩其实就是游戏中我们所操作的角色，不过在这简化为了树桩。类的定义可参考下图。

玩家类中主要包含了玩家对象的属性和能够进行的交互。

在玩家属性方面，一部分是保存用于渲染的信息，还有一部分是保存玩家的位置、与摄像机的距离，以及缩放矩阵。后一部分信息主要是为实现用户与角色交互提供服务。

在交互功能方面，主要是包括绘制、拉伸和重置，具体的说明在“主要函数说明”中。

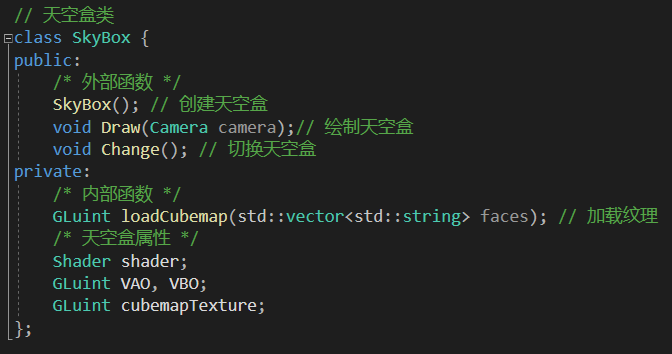


### 天空盒部分（class SkyBox）

为了使环境尽可能真实，我实现了一个天空盒，这里用了雪景的图片，创建出一个雪地的场景。

与玩家类一样，我也将天空盒部分的信息和操作都封装到了天空盒类中。并且天空盒类的组成结构与玩家类相似，但更为简单，可参考下图。

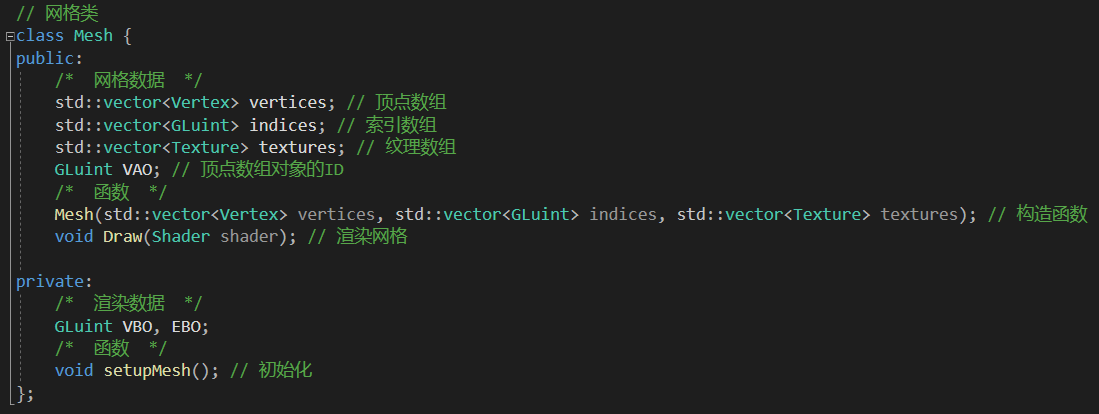
天空盒属性方面，其中包含着天空盒渲染所需的信息。此外就是天空盒的功能部分，具体的说明在“主要函数说明”中。



### 模型部分（class Mesh和class Model）

这一部分主要是为assimp库服务，用于导入.obj格式的模型。

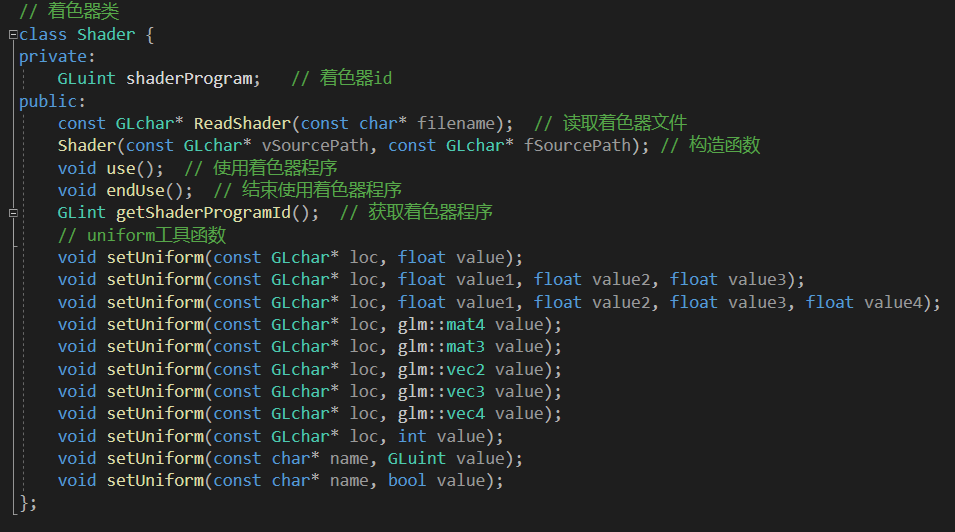
一个模型可能由许多个网格组成，因此class Mesh的对象其实是class Model对象中的一个属性，其内包含着每部分渲染所需的信息。关于这两个类的定义可参考下图，在实现这两个类的时候我参考了LearnOpenGL上的资料，感觉这两个类的实现已经可以算套路化和模板化了，因此在此就不过多赘述。



### 着色器部分（class Shader）

关于着色器类的实现，我大体上和老师提供的差不多，没有什么值得说明的。不过之前在实现的同时，顺便根据自己的编程习惯在一些命名上略作修改，可参考下图。

此外，在这次的作业中，我为天空盒对象、文件对象，以及模型对象各实现了一个着色器文件，即这三个类所使用的是不同的着色器。



## 主要函数说明

### 摄像机部分（class Camera）

摄像机类中主要的功能实现是包括以下几个函数。

* 首先是创建摄像机对象，我直接将这部分代码放入构造函数中实现，在构造函数中初始化摄像机各个参数值。

Camera(glm::vec3 position = glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f), float yaw = YAW, float pitch = PITCH); // 初始化

* 其次是返回视图矩阵，用于其他对象的创建

glm::mat4 GetViewMatrix();// 返回视图矩阵

而用户能直观感受到摄像机的则是以下几个与交互有关的功能。

* 当用户点击鼠标左键时，代表用户想要移动摄像机位置，此时会触发此函数。

void ProcessMousePress(float xOffset, float yOffset, float deltaTime); // 左键平移

* 当用户点击鼠标右键时，代表用户想要重置摄像机状态，此时会触发此函数。

void ProcessMousePress(); // 右键复原

* 当用户按住ctrl键并移动鼠标时，代表用户想要转动摄像机朝向，此时会触发此函数。

void ProcessMouseMovement(float xOffset, float yOffset, GLboolean constrainPitch = true); // 旋转

* 当用户旋转鼠标滚轮时，代表用户想要调整摄像机的焦距，此时会触发此函数，修改摄像机的Zoom属性。

void ProcessMouseScroll(float yOffset); // 缩放

### 角色部分（class Player）

玩家类的功能主要由以下几个函数实现。

* 首先是角色的创建部分，包括初始化和贴图等，我都在构造函数中进行实现

Player(); // 创建对象

* 每一帧都会重新渲染对象，保证角色对象在摄像机对象的前方

void Draw(Camera camera); // 绘制对象

* 当用户按下键盘上的方向键时，说明用户想拉伸角色对象，此时会触发此函数

void Stretch(StretchDrection direction); // 拉伸对象

* 当用户点击鼠标右键时，代表用户想要重置对象状态，此时会触发此函数。

void Init(); // 重置对象

### 天空盒部分（class SkyBox）

天空盒类的功能由以下几个函数实现。

* 首先是创建天空盒，与前面一样，我还是直接在构造函数中完成了初始化

SkyBox(); // 创建天空盒

* 其次是渲染天空盒，值得一提的是，在这个函数中我同时还实现了天空盒昼夜交替的效果，使天空盒有明暗变化

void Draw(Camera camera);// 绘制天空盒

* 最后这个函数我虽然在类中定义了出来，但由于时间关系没来得及实现，按原先的设想，是进行昼夜交替的同时切换天空盒，即将白天的天空盒与夜晚的天空盒相互切换

void Change(); // 切换天空盒

### 主函数部分（main）

集成的部分主要在main函数中进行，此外在main.cpp中也实现了几个重要的函数

* 第一部分是回调函数，用于实时检测用户的输入

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

void mouse\_callback(GLFWwindow\* window, Player& player);

void scroll\_callback(GLFWwindow\* window, double xOffset, double yOffset);

void processInput(GLFWwindow\* window, Player& player);

* 第二部分是模型的渲染函数，用于控制模型的位置与大小，最重要的是实现了太阳光照，模拟出太阳在一天不同时刻中的光照情况。

void rend(Shader shader, Model myModel, glm::vec3 translate, glm::vec3 scale)

# 实验结果

这里仅简单展示白天与黑夜的静态图，动态效果参考演示视频。

**白天**



**黑夜**



# 程序代码

### 摄像机部分（class Camera）

1. **camera.h**

#pragma once

#include <glad/glad.h>

#include <glm/glm.hpp>

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

#include <vector>

// 默认摄像机数值

const float YAW = -90.0f;

const float PITCH = 0.0f;

const float SPEED = 2.5f;

const float SENSITIVITY = 0.1f;

const float ZOOM = 45.0f;

// 摄像机类

class Camera {

public:

/\* 摄像机功能 \*/

Camera(glm::vec3 position = glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f), float yaw = YAW, float pitch = PITCH); // 初始化

glm::mat4 GetViewMatrix(); // 返回视图矩阵

void ProcessMousePress(float xOffset, float yOffset, float deltaTime); // 处理鼠标点击（左键平移）

void ProcessMousePress(); // 处理鼠标点击（右键复原）

void ProcessMouseMovement(float xOffset, float yOffset, GLboolean constrainPitch = true); // 处理鼠标移动（旋转）

void ProcessMouseScroll(float yOffset); // 处理鼠标滚（缩放）

/\* getter&setter \*/

glm::vec3 getPosition();

glm::vec3 getFront();

glm::vec3 getUp();

float getYaw();

float getPitch();

float getZoom();

private:

/\* 内部函数 \*/

void UpdateCameraVectors(); // 计算前向向量

/\* 摄像机信息 \*/

// 摄像机属性

glm::vec3 Position; // 摄像机位置

glm::vec3 Front; // 摄像机正对的方向向量

glm::vec3 Up; // 摄像机上方向向量

glm::vec3 Right; // 摄像机右方向向量

glm::vec3 WorldUp; // 世界坐标系中的上方向

// 欧拉角（表示摄像机的方向）

float Yaw;

float Pitch;

// 摄像机性能参数

float MovementSpeed; // 摄像机移动速度

float MouseSensitivity; // 鼠标灵敏度

float Zoom; // 缩放值（通常用于控制视场或投影矩阵的缩放）

};

1. **camera.cpp**

#include "camera.h"

/\* 摄像机功能 \*/

// 初始化

Camera::Camera(glm::vec3 position, glm::vec3 up, float yaw, float pitch) {

// 摄像机属性

this->Position = position;

this->Front = glm::vec3(0.0f, 0.0f, -1.0f);

this->WorldUp = up;

// 欧拉角

this->Yaw = yaw;

this->Pitch = pitch;

// 相机选项

this->MovementSpeed = SPEED;

this->MouseSensitivity = SENSITIVITY;

this->Zoom = ZOOM;

this->UpdateCameraVectors();

}

// 返回视图矩阵

glm::mat4 Camera::GetViewMatrix() {

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

}

// 处理鼠标点击（左键平移）

void Camera::ProcessMousePress(float xOffset, float yOffset, float deltaTime) {

float velocity = this->MovementSpeed \* deltaTime;

// 计算水平方向的位移

glm::vec3 horizontalMovement = this->Right \* (xOffset / 10.0f);

// 只保留水平方向的分量，将垂直方向分量设为0

horizontalMovement.y = 0.0f;

this->Position += horizontalMovement;

// 计算前进方向的位移

glm::vec3 forwardMovement = this->Front \* (yOffset / 10.0f);

// 只保留前进方向的分量，将垂直方向分量设为0

forwardMovement.y = 0.0f;

this->Position += forwardMovement;

}

// 处理鼠标点击（右键复原）

void Camera::ProcessMousePress() {

this->Position = glm::vec3(0.0f, 0.0f, 3.0f);

this->Front = glm::vec3(0.0f, 0.0f, -1.0f);

this->WorldUp = glm::vec3(0.0f, 1.0f, 0.0f);

this->Yaw = YAW;

this->Pitch = PITCH;

this->MovementSpeed = SPEED;

this->MouseSensitivity = SENSITIVITY;

this->Zoom = ZOOM;

this->UpdateCameraVectors();

}

// 处理鼠标移动（旋转）

void Camera::ProcessMouseMovement(float xOffset, float yOffset, GLboolean constrainPitch) {

xOffset \*= this->MouseSensitivity;

yOffset \*= this->MouseSensitivity;

this->Yaw += xOffset;

this->Pitch += yOffset;

// 确保在俯仰角超出边界时，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;

}

}

// 更新

this->UpdateCameraVectors();

}

// 处理鼠标滚轮（缩放）

void Camera::ProcessMouseScroll(float yOffset) {

this->Zoom -= (float)yOffset;

if (Zoom < 1.0f) {

Zoom = 1.0f;

}

if (Zoom > 45.0f) {

Zoom = 45.0f;

}

}

// getter

glm::vec3 Camera::getPosition() {

return this->Position;

}

glm::vec3 Camera::getFront() {

return this->Front;

}

glm::vec3 Camera::getUp() {

return this->Up;

}

float Camera::getYaw() {

return this->Yaw;

}

float Camera::getPitch() {

return this->Pitch;

}

float Camera::getZoom() {

return this->Zoom;

}

/\* 内部函数 \*/

// 计算前向向量

void Camera::UpdateCameraVectors() {

// 计算新的Front向量

glm::vec3 front = glm::vec3(0.0f, 0.0f, 0.0f);

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);

// 重新计算Right和Up向量

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

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

}

### 角色部分（class Player）

1. **player.h**

#pragma once

#include "shader.h"

#include "camera.h"

#include "stb\_image.h"

// 方向

enum StretchDrection {

UP,

DOWN,

LEFT,

RIGHT

};

// 玩家类

class Player {

public:

/\* 外部函数 \*/

Player(); // 创建对象

void Draw(Camera camera); // 绘制对象

void Stretch(StretchDrection direction); // 拉伸对象

void Init(); // 重置对象

private:

/\* 内部函数 \*/

GLuint loadTexture(char const\* path);

/\* 属性 \*/

Shader shader;

GLuint VAO, VBO, EBO;

GLuint textureAround, textureTop;

glm::vec3 Position;

glm::vec3 Offset;

glm::vec3 scale;

};

1. **player.cpp**

#include "player.h"

// 创建对象

Player::Player() : shader("player.vs", "player.fs") {

GLfloat vertices[] = {

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

};

unsigned int indices[] = {

0, 1, 2,

3, 4, 5,

6, 7, 8,

9, 10, 11,

12, 13, 14,

15, 16, 17,

18, 19, 20,

21, 22, 23,

24, 25, 26,

27, 28, 29,

30, 31, 32,

33, 34, 35

};

glGenVertexArrays(1, &this->VAO);

glBindVertexArray(this->VAO);

glGenBuffers(1, &this->VBO);

glBindBuffer(GL\_ARRAY\_BUFFER, this->VBO);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 8 \* sizeof(float), (void\*)0);

glVertexAttribPointer(1, 2, GL\_FLOAT, GL\_FALSE, 8 \* sizeof(float), (void\*)(3 \* sizeof(float)));

glVertexAttribPointer(2, 3, GL\_FLOAT, GL\_FALSE, 8 \* sizeof(float), (void\*)(5 \* sizeof(float)));

glEnableVertexAttribArray(0);

glEnableVertexAttribArray(1);

glEnableVertexAttribArray(2);

glGenBuffers(1, &this->EBO);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, this->EBO);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(indices), indices, GL\_STATIC\_DRAW);

glBindVertexArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);

this->textureAround = loadTexture("resource/player/around.jpg");

this->textureTop = loadTexture("resource/player/top.jpg");

this->Offset = glm::vec3(0.0f, -1.75f, -5.0f);

this->scale = glm::vec3(1.0f, 1.0f, 1.0f);

}

// 绘制对象

void Player::Draw(Camera camera) {

this->shader.use();

glActiveTexture(GL\_TEXTURE0);

glBindTexture(GL\_TEXTURE\_2D, this->textureAround);

shader.setUniform("firtsTexture", 0);

glActiveTexture(GL\_TEXTURE1);

glBindTexture(GL\_TEXTURE\_2D, this->textureTop);

this->shader.setUniform("secondTexture", 1);

glm::mat4 model = glm::mat4(1.0f);

model = glm::translate(model, this->Offset + camera.getPosition());

model = glm::scale(model, this->scale);

shader.setUniform("model", model);

glm::mat4 projection = glm::perspective(glm::radians(camera.getZoom()), (float)800 / (float)600, 0.1f, 100.0f);

shader.setUniform("projection", projection);

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

shader.setUniform("view", view);

shader.setUniform("viewPos", camera.getPosition());

// 平行光

shader.setUniform("myMaterial.diffuse", 0.5f);

shader.setUniform("myMaterial.specular", 0.5f);

shader.setUniform("myMaterial.shiness", 32.0f);

float t = fmod(glfwGetTime(), 20.0f) / 20.0f;

float transitionT = 0.0f;

if (t < 0.25f) {

transitionT = t / 0.25f;

}

else if (t < 0.5f) {

transitionT = 1.0f;

}

else if (t < 0.75f) {

transitionT = 1.0f - (t - 0.5f) / 0.25f;

}

else {

transitionT = 0.0f;

}

float sunAngle = transitionT \* 2.0f \* glm::pi<float>();

glm::vec3 sunDirection = glm::vec3(cos(sunAngle), sin(sunAngle), -0.5f);

shader.setUniform("myDirectionalLight.direction", sunDirection);

float ambientIntensity = glm::mix(0.2f, 1.0f, transitionT);

float diffuseIntensity = glm::mix(0.4f, 1.0f, transitionT);

float specularIntensity = glm::mix(0.1f, 0.3f, transitionT);

shader.setUniform("myDirectionalLight.ambient", ambientIntensity, ambientIntensity, ambientIntensity);

shader.setUniform("myDirectionalLight.diffuse", diffuseIntensity, diffuseIntensity, diffuseIntensity);

shader.setUniform("myDirectionalLight.specular", specularIntensity, specularIntensity, specularIntensity);

glBindVertexArray(this->VAO);

glDrawElements(GL\_TRIANGLES, 36, GL\_UNSIGNED\_INT, 0);

glBindVertexArray(0);

}

// 拉伸对象

void Player::Stretch(StretchDrection direction) {

if (direction == UP) {

this->scale.y += 0.05f;

}

else if (direction == DOWN) {

this->scale.y -= 0.05f;

}

else if (direction == LEFT) {

this->scale.x += 0.05f;

}

else {

this->scale.x -= 0.05f;

}

}

// 重置对象

void Player::Init() {

this->scale = glm::vec3(1.0f, 1.0f, 1.0f);

}

// 加载纹理

GLuint Player::loadTexture(char const\* path)

{

stbi\_set\_flip\_vertically\_on\_load(true);

unsigned int textureID;

glGenTextures(1, &textureID);

glBindTexture(GL\_TEXTURE\_2D, textureID);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

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

int width, height, nrComponents;

unsigned char\* data = stbi\_load(path, &width, &height, &nrComponents, 0);

if (data) {

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, data);

}

else {

std::cout << "Failed to load texture" << std::endl;

}

stbi\_image\_free(data);

return textureID;

}

### 天空盒部分（class SkyBox）

1. **skybox.h**

#pragma once

#include "stb\_image.h"

#include "shader.h"

#include "camera.h"

#include "vector"

// 天空盒类

class SkyBox {

public:

/\* 外部函数 \*/

SkyBox(); // 创建天空盒

void Draw(Camera camera);// 绘制天空盒

void Change(); // 切换天空盒

private:

/\* 内部函数 \*/

GLuint loadCubemap(std::vector<std::string> faces); // 加载纹理

/\* 天空盒属性 \*/

Shader shader;

GLuint VAO, VBO;

GLuint cubemapTexture;

};

1. **skybox.cpp**

#include "skybox.h"

// 创建天空盒

SkyBox::SkyBox() : shader("skybox.vs", "skybox.fs") {

float vertices[] = {

-1.0f, 1.0f, -1.0f,

-1.0f, -1.0f, -1.0f,

1.0f, -1.0f, -1.0f,

1.0f, -1.0f, -1.0f,

1.0f, 1.0f, -1.0f,

-1.0f, 1.0f, -1.0f,

-1.0f, -1.0f, 1.0f,

-1.0f, -1.0f, -1.0f,

-1.0f, 1.0f, -1.0f,

-1.0f, 1.0f, -1.0f,

-1.0f, 1.0f, 1.0f,

-1.0f, -1.0f, 1.0f,

1.0f, -1.0f, -1.0f,

1.0f, -1.0f, 1.0f,

1.0f, 1.0f, 1.0f,

1.0f, 1.0f, 1.0f,

1.0f, 1.0f, -1.0f,

1.0f, -1.0f, -1.0f,

-1.0f, -1.0f, 1.0f,

-1.0f, 1.0f, 1.0f,

1.0f, 1.0f, 1.0f,

1.0f, 1.0f, 1.0f,

1.0f, -1.0f, 1.0f,

-1.0f, -1.0f, 1.0f,

-1.0f, 1.0f, -1.0f,

1.0f, 1.0f, -1.0f,

1.0f, 1.0f, 1.0f,

1.0f, 1.0f, 1.0f,

-1.0f, 1.0f, 1.0f,

-1.0f, 1.0f, -1.0f,

-1.0f, -1.0f, -1.0f,

-1.0f, -1.0f, 1.0f,

1.0f, -1.0f, -1.0f,

1.0f, -1.0f, -1.0f,

-1.0f, -1.0f, 1.0f,

1.0f, -1.0f, 1.0f

};

glGenVertexArrays(1, &this->VAO);

glBindVertexArray(this->VAO);

glGenBuffers(1, &this->VBO);

glBindBuffer(GL\_ARRAY\_BUFFER, this->VBO);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), &vertices, GL\_STATIC\_DRAW);

glEnableVertexAttribArray(0);

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

std::vector<std::string> faces {

"resource/skybox/day/right.png",

"resource/skybox/day/left.png",

"resource/skybox/day/top.png",

"resource/skybox/day/bottom.png",

"resource/skybox/day/front.png",

"resource/skybox/day/back.png",

};

this->cubemapTexture = loadCubemap(faces);

}

// 绘制天空盒子

void SkyBox::Draw(Camera camera) {

glDepthFunc(GL\_LEQUAL);

this->shader.use();

this->shader.setUniform("skybox", 0);

glm::mat4 view = glm::mat4(glm::mat3(camera.GetViewMatrix()));

glm::mat4 projection = glm::perspective(45.0f, (float)800 / (float)600, 0.1f, 100.0f);

shader.setUniform("view", view);

shader.setUniform("projection", projection);

// 昼夜交替

float t = fmod(glfwGetTime(), 20.0f) / 20.0f;

glm::vec3 color1 = glm::vec3(1.0f, 1.0f, 1.0f);

glm::vec3 color2 = glm::vec3(0.2f, 0.2f, 0.2f);

glm::vec3 ambientLight;

if (t < 0.25f) {

// 前五秒，逐渐变亮

float transitionT = glm::smoothstep(0.0f, 1.0f, t / 0.25f);

ambientLight = glm::mix(color2, color1, transitionT);

}

else if (t < 0.5f) {

// 中间五秒，保持一个亮度

ambientLight = color1;

}

else if (t < 0.75f) {

// 后五秒，逐渐变暗

float transitionT = glm::smoothstep(0.0f, 1.0f, (t - 0.5f) / 0.25f);

ambientLight = glm::mix(color1, color2, transitionT);

}

else {

// 最后五秒，保持一个亮度

ambientLight = color2;

}

this->shader.setUniform("ambientLight", ambientLight);

glBindVertexArray(this->VAO);

glActiveTexture(GL\_TEXTURE0);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, this->cubemapTexture);

glDrawArrays(GL\_TRIANGLES, 0, 36);

glBindVertexArray(0);

glDepthFunc(GL\_LESS);

}

// 切换天空盒

void SkyBox::Change() {

}

// 加载纹理

GLuint SkyBox::loadCubemap(std::vector<std::string> faces) {

GLuint textureID;

glGenTextures(1, &textureID);

int width, height, nrChannels;

unsigned char\* image;

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, textureID);

for (GLuint i = 0; i < faces.size(); i++) {

image = stbi\_load(faces[i].c\_str(), &width, &height, &nrChannels, 0);

if (image) {

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X + i, 0, GL\_RGB, width, height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, image);

}

else {

std::cout << "Cubemap texture failed to load at path: " << faces[i] << std::endl;

}

stbi\_image\_free(image);

}

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_R, GL\_CLAMP\_TO\_EDGE);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, 0);

return textureID;

}

### 模型部分（class Mesh和class Model）

1. **mesh.h**

#pragma once

#include <glad/glad.h>

#include <glm/glm.hpp>

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

#include "shader.h"

#include <string>

#include <vector>

#define MAX\_BONE\_INFLUENCE 4

// 顶点结构体

struct Vertex

{

glm::vec3 Position; // 位置

glm::vec3 Normal; // 法线

glm::vec2 TexCoords; // 纹理坐标

glm::vec3 Tangent; // 切线

glm::vec3 Bitangent; // 副法线

int m\_BoneIDs[MAX\_BONE\_INFLUENCE]; // 骨骼影响该顶点的骨骼索引数组

float m\_Weights[MAX\_BONE\_INFLUENCE]; // 每个骨骼对该顶点的权重

};

// 纹理结构体

struct Texture

{

GLuint id; // 纹理ID

std::string type; // 纹理类型

std::string path; // 纹理文件路径

};

// 网格类

class Mesh {

public:

/\* 网格数据 \*/

std::vector<Vertex> vertices; // 顶点数组

std::vector<GLuint> indices; // 索引数组

std::vector<Texture> textures; // 纹理数组

GLuint VAO; // 顶点数组对象的ID

/\* 函数 \*/

Mesh(std::vector<Vertex> vertices, std::vector<GLuint> indices, std::vector<Texture> textures); // 构造函数

void Draw(Shader shader); // 渲染网格

private:

/\* 渲染数据 \*/

GLuint VBO, EBO;

/\* 函数 \*/

void setupMesh(); // 初始化

};

1. **mesh.cpp**

#include "mesh.h"

// 初始化缓冲区和顶点数组对象

void Mesh::setupMesh()

{

// 创建顶点数组对象和缓冲区

glGenVertexArrays(1, &this->VAO);

glGenBuffers(1, &this->VBO);

glGenBuffers(1, &this->EBO);

glBindVertexArray(this->VAO);

// 将数据加载到顶点缓冲区

glBindBuffer(GL\_ARRAY\_BUFFER, this->VBO);

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

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, this->EBO);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, this->indices.size() \* sizeof(GLuint), &this->indices[0], GL\_STATIC\_DRAW);

// 设置顶点坐标指针

glEnableVertexAttribArray(0);

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

// 设置法线指针

glEnableVertexAttribArray(1);

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

// 设置顶点的纹理坐标

glEnableVertexAttribArray(2);

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

glBindVertexArray(0);

}

// 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();

}

// 渲染网格

void Mesh::Draw(Shader shader) {

// 绑定纹理

GLuint diffuseNr = 1;

GLuint specularNr = 1;

GLuint normalNr = 1;

GLuint heightNr = 1;

for (GLuint i = 0; i < textures.size(); i++)

{

glActiveTexture(GL\_TEXTURE0 + i); // 在绑定纹理前需要激活相应的纹理单元

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++);

}

shader.setUniform((name + number).c\_str(), i);

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

}

// 绘制网格

glBindVertexArray(VAO);

glDrawElements(GL\_TRIANGLES, static\_cast<unsigned int>(indices.size()), GL\_UNSIGNED\_INT, 0);

glBindVertexArray(0);

// 恢复默认设置

glActiveTexture(GL\_TEXTURE0);

}

1. **model.h**

#pragma once

#include <glad/glad.h>

#include <glm/glm.hpp>

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

#include <assimp/Importer.hpp>

#include <assimp/scene.h>

#include <assimp/postprocess.h>

#include "stb\_image.h"

#include "mesh.h"

#include "shader.h"

#include <string>

#include <fstream>

#include <sstream>

#include <iostream>

#include <map>

#include <vector>

// 模型类

class Model

{

public:

/\* 外部函数 \*/

Model(const GLchar\* path); // 加载模型

void Draw(Shader shader); // 绘制模型

private:

/\* 模型数据 \*/

std::vector<Mesh> meshes; // 存储模型的所有网格

std::string directory; // 存储模型文件所在的目录路径

std::vector<Texture> textures\_loaded;

/\* 内部函数 \*/

void loadModel(std::string path); // 加载模型

void processNode(aiNode\* node, const aiScene\* scene); // 处理模型节点

Mesh processMesh(aiMesh\* mesh, const aiScene\* scene); // 处理单个网格

std::vector<Texture> loadMaterialTextures(aiMaterial\* mat, aiTextureType type, std::string typeName); // 加载材质纹理

GLint TextureFromFile(const char\* path, std::string directory);// 从文件路径加载纹理

};

1. **model.cpp**

#include "model.h"

/\* 外部函数 \*/

// 加载模型

Model::Model(const GLchar\* path) {

this->loadModel(path);

}

// 绘制模型

void Model::Draw(Shader shader) {

for (GLuint i = 0; i < this->meshes.size(); i++) {

this->meshes[i].Draw(shader);

}

}

/\* 内部函数 \*/

// 加载模型

void Model::loadModel(std::string path) {

// 使用ASSIMP读取文件

Assimp::Importer importer;

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

// 检查错误

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

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

return;

}

// 获取文件路径的目录

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

// 递归处理ASSIMP的根节点

this->processNode(scene->mRootNode, scene);

}

// 处理模型节点

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

// 逐个处理当前节点上的每个网格

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

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

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

}

// 处理子节点

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

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

}

}

// 处理单个网格

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

// 用于存储网格数据

std::vector<Vertex> vertices;

std::vector<GLuint> indices;

std::vector<Texture> textures;

// 处理每个网格的顶点

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

Vertex vertex;

glm::vec3 vector;

// 顶点位置

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

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

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

vertex.Position = vector;

// 法线

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

if (mesh->mTextureCoords[0]) {

glm::vec2 vec;

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

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

vertex.TexCoords = vec;

}

else {

vertex.TexCoords = glm::vec2(0.0f, 0.0f);

}

vertices.push\_back(vertex);

}

// 处理面的索引

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

aiFace face = mesh->mFaces[i];

if (face.mNumIndices < 3) {

continue;

}

for (GLuint 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

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

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

// 镜面

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

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

}

// 返回由提取的网格数据创建的Mesh对象

return Mesh(vertices, indices, textures);

}

// 加载材质纹理

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);

// 检查纹理是否已加载，如果是，则继续下一次迭代：跳过加载新纹理的步骤

GLboolean skip = false;

for (GLuint 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;

}

}

if (!skip) {

// 如果纹理尚未加载，则加载它

Texture texture;

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

texture.type = typeName;

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

textures.push\_back(texture);

this->textures\_loaded.push\_back(texture);

}

}

return textures;

}

// 从文件路径加载纹理

GLint Model::TextureFromFile(const char\* path, std::string directory)

{

// 生成纹理ID并加载纹理数据

std::string filename = path;

filename = directory + '/' + filename;

GLuint textureID;

glGenTextures(1, &textureID);

int width, height, nrChannels;

unsigned char\* image = stbi\_load(filename.c\_str(), &width, &height, &nrChannels, 0);

if (!image) {

std::cerr << "Failed to load texture: " << filename << std::endl;

return -1;

}

// 将纹理分配给ID

glBindTexture(GL\_TEXTURE\_2D, textureID);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, image);

glGenerateMipmap(GL\_TEXTURE\_2D);

// 设置纹理参数

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);

glBindTexture(GL\_TEXTURE\_2D, 0);

stbi\_image\_free(image);

return textureID;

}

### 着色器部分（class Shader）

1. **shader.h**

#pragma once

#include <iostream>

#include <string>

#include <glad/glad.h>

#include <GLFW/glfw3.h>

#include <glm/glm.hpp>

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

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

// 着色器类

class Shader {

private:

GLuint shaderProgram; // 着色器id

public:

const GLchar\* ReadShader(const char\* filename); // 读取着色器文件

Shader(const GLchar\* vSourcePath, const GLchar\* fSourcePath); // 构造函数

void use(); // 使用着色器程序

void endUse(); // 结束使用着色器程序

GLint getShaderProgramId(); // 获取着色器程序

// uniform工具函数

void setUniform(const GLchar\* loc, float value);

void setUniform(const GLchar\* loc, float value1, float value2, float value3);

void setUniform(const GLchar\* loc, float value1, float value2, float value3, float value4);

void setUniform(const GLchar\* loc, glm::mat4 value);

void setUniform(const GLchar\* loc, glm::mat3 value);

void setUniform(const GLchar\* loc, glm::vec2 value);

void setUniform(const GLchar\* loc, glm::vec3 value);

void setUniform(const GLchar\* loc, glm::vec4 value);

void setUniform(const GLchar\* loc, int value);

void setUniform(const char\* name, GLuint value);

void setUniform(const char\* name, bool value);

};

1. **shader.cpp**

#include "shader.h"

// 读取着色器文件

const GLchar\* Shader::ReadShader(const char\* filename)

{

FILE\* infile = NULL;

fopen\_s(&infile, filename, "rb");

if (!infile) {

std::cerr << "Unable to open file '" << filename << "'" << std::endl;

return NULL;

}

fseek(infile, 0, SEEK\_END);

size\_t len = ftell(infile);

fseek(infile, 0, SEEK\_SET);

GLchar\* source = new GLchar[len + 1];

fread(source, 1, len, infile);

fclose(infile);

source[len] = '\0';

return const\_cast<const GLchar\*>(source);

}

// 构造函数

Shader::Shader(const GLchar\* vSourcePath, const GLchar\* fSourcePath) {

// 从文件路径中获取顶点/片段着色器

const char\* vShaderSource = ReadShader(vSourcePath);

const char\* fShaderSource = ReadShader(fSourcePath);

// 创建顶点着色器

GLuint vertexShader = glCreateShader(GL\_VERTEX\_SHADER);

glShaderSource(vertexShader, 1, &vShaderSource, NULL);

glCompileShader(vertexShader);

//编译检查

GLint success;

char infoLog[512];

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

if (!success)

{

glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);

std::cerr << "VertexShader program failed to compile. " << infoLog << std::endl;

}

// 创建片元着色器

GLuint fragmentShader = glCreateShader(GL\_FRAGMENT\_SHADER);

glShaderSource(fragmentShader, 1, &fShaderSource, NULL);

glCompileShader(fragmentShader);

//编译检查

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

if (!success)

{

glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);

std::cerr << "FragmentShader program failed to compile. " << infoLog << std::endl;

}

// 创建着色器程序

this->shaderProgram = glCreateProgram();

// 链接着色器

glAttachShader(this->shaderProgram, vertexShader);

glAttachShader(this->shaderProgram, fragmentShader);

glLinkProgram(this->shaderProgram);

// 链接检查

glGetProgramiv(this->shaderProgram, GL\_LINK\_STATUS, &success);

if (!success) {

glGetProgramInfoLog(this->shaderProgram, 512, NULL, infoLog);

std::cout << "Shader program linking failed. " << infoLog << std::endl;

}

// 删除着色器

glDeleteShader(vertexShader);

glDeleteShader(fragmentShader);

}

// 使用着色器程序

void Shader::use() {

// 使用着色器

glUseProgram(this->shaderProgram);

}

// 结束使用着色器程序

void Shader::endUse()

{

glUseProgram(0);

}

// 获取着色器程序

GLint Shader::getShaderProgramId() {

return this->shaderProgram;

}

// uniform工具函数

void Shader::setUniform(const GLchar\* loc, float value) {

glUniform1f(glGetUniformLocation(shaderProgram, loc), value);

}

void Shader::setUniform(const GLchar\* loc, float value1, float value2, float value3) {

glUniform3f(glGetUniformLocation(shaderProgram, loc), value1, value2, value3);

}

void Shader::setUniform(const GLchar\* loc, float value1, float value2, float value3, float value4) {

glUniform4f(glGetUniformLocation(shaderProgram, loc), value1, value2, value3, value4);

}

void Shader::setUniform(const GLchar\* loc, glm::mat4 value) {

glUniformMatrix4fv(glGetUniformLocation(shaderProgram, loc), 1, GL\_FALSE, glm::value\_ptr(value));

}

void Shader::setUniform(const GLchar\* loc, glm::mat3 value) {

glUniformMatrix3fv(glGetUniformLocation(shaderProgram, loc), 1, GL\_FALSE, glm::value\_ptr(value));

}

void Shader::setUniform(const GLchar\* loc, glm::vec2 value) {

glUniform3fv(glGetUniformLocation(shaderProgram, loc), 1, glm::value\_ptr(value));

}

void Shader::setUniform(const GLchar\* loc, glm::vec3 value) {

glUniform3fv(glGetUniformLocation(shaderProgram, loc), 1, glm::value\_ptr(value));

}

void Shader::setUniform(const GLchar\* loc, glm::vec4 value) {

glUniform3fv(glGetUniformLocation(shaderProgram, loc), 1, glm::value\_ptr(value));

}

void Shader::setUniform(const GLchar\* loc, int value) {

glUniform1i(glGetUniformLocation(shaderProgram, loc), value);

}

void Shader::setUniform(const char\* loc, GLuint value) {

glUniform1ui(glGetUniformLocation(shaderProgram, loc), value);

}

void Shader::setUniform(const char\* loc, bool value) {

glUniform1i(glGetUniformLocation(shaderProgram, loc), value);

}

### 主函数部分（main）

#include <glad/glad.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 "skybox.h"

#include "player.h"

#include <iostream>

// 回调函数

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

void mouse\_callback(GLFWwindow\* window, Player& player);

void scroll\_callback(GLFWwindow\* window, double xOffset, double yOffset);

void processInput(GLFWwindow\* window, Player& player);

// 参数

const unsigned int SCR\_WIDTH = 800;

const unsigned int SCR\_HEIGHT = 600;

float deltaTime = 0.0f;

float lastFrame = 0.0f;

// 摄像机

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

float lastX = SCR\_WIDTH / 2.0f;

float lastY = SCR\_HEIGHT / 2.0f;

bool firstMouse = true;

// 渲染

void rend(Shader shader, Model myModel, glm::vec3 translate, glm::vec3 scale) {

shader.use();

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

shader.setUniform("projection", projection);

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

shader.setUniform("view", view);

shader.setUniform("viewPos", camera.getPosition());

// 平行光

float t = fmod(glfwGetTime(), 20.0f) / 20.0f;

float transitionT = 0.0f;

if (t < 0.25f) {

transitionT = t / 0.25f;

}

else if (t < 0.5f) {

transitionT = 1.0f;

}

else if (t < 0.75f) {

transitionT = 1.0f - (t - 0.5f) / 0.25f;

}

else {

transitionT = 0.0f;

}

float sunAngle = transitionT \* 2.0f \* glm::pi<float>();

glm::vec3 sunDirection = glm::vec3(cos(sunAngle), sin(sunAngle), -0.5f);

shader.setUniform("myDirectionalLight.direction", sunDirection);

float ambientIntensity = glm::mix(0.2f, 1.0f, transitionT);

float diffuseIntensity = glm::mix(0.4f, 1.0f, transitionT);

float specularIntensity = glm::mix(0.1f, 0.3f, transitionT);

shader.setUniform("myDirectionalLight.ambient", ambientIntensity, ambientIntensity, ambientIntensity);

shader.setUniform("myDirectionalLight.diffuse", diffuseIntensity, diffuseIntensity, diffuseIntensity);

shader.setUniform("myDirectionalLight.specular", specularIntensity, specularIntensity, specularIntensity);

glm::mat4 model = glm::mat4(1.0f);

model = glm::translate(model, translate);

model = glm::scale(model, scale);

shader.setUniform("model", model);

myModel.Draw(shader);

}

int main()

{

// 初始化和配置GLFW

glfwInit();

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

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

// 创建GLFW窗口

GLFWwindow\* window = glfwCreateWindow(SCR\_WIDTH, SCR\_HEIGHT, "RealSceneInTheSnow", NULL, NULL);

if (window == NULL) {

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

glfwTerminate();

return -1;

}

glfwMakeContextCurrent(window);

glfwSetFramebufferSizeCallback(window, framebuffer\_size\_callback);

glfwSetScrollCallback(window, scroll\_callback);

glfwSetInputMode(window, GLFW\_CURSOR, GLFW\_CURSOR\_DISABLED);

// 初始化GLAD

if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {

std::cout << "Failed to initialize GLAD" << std::endl;

return -1;

}

// 加载纹理时翻转图像y轴

stbi\_set\_flip\_vertically\_on\_load(false);

// 开启深度检测

glEnable(GL\_DEPTH\_TEST);

Shader shader("player.vs", "player.fs");

//创建天空盒与模型

SkyBox skybox;

Shader modelShader("model.vs", "model.fs");

Model sonwmanModel("resource/snowman/snowman\_finish.obj"); // 创建雪人

Model houseModel("resource/house/6H1.obj"); // 创建房屋

Model treeModel1("resource/tree/Gledista\_Triacanthos.obj"); // 创建树木1

Model treeModel2("resource/tree/Gledista\_Triacanthos\_2.obj"); // 创建树木2

Player player;

// 渲染循环

while (!glfwWindowShouldClose(window))

{

// 每帧时间逻辑

float currentFrame = static\_cast<float>(glfwGetTime());

deltaTime = currentFrame - lastFrame;

lastFrame = currentFrame;

// 处理输入

processInput(window, player);

mouse\_callback(window, player);

// 渲染

glClearColor(0.1f, 0.1f, 0.1f, 1.0f);

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

rend(modelShader, sonwmanModel, glm::vec3(5.0f, -1.75f, -5.0f), glm::vec3(1.0f, 1.0f, 1.0f));

rend(modelShader, houseModel, glm::vec3(0.0f, -1.75f, -15.0f), glm::vec3(0.1f, 0.1f, 0.1f));

rend(modelShader, treeModel1, glm::vec3(-10.0f, -1.75f, -25.0f), glm::vec3(0.3f, 0.3f, 0.3f));

rend(modelShader, treeModel2, glm::vec3(15.0f, -1.75f, -25.0f), glm::vec3(0.3f, 0.3f, 0.3f));

player.Draw(camera);

skybox.Draw(camera);

// 交换缓冲区和处理IO事件

glfwSwapBuffers(window);

glfwPollEvents();

}

glfwTerminate();

return 0;

}

// 调整窗口大小

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

{

glViewport(0, 0, width, height);

}

// 监控键盘输入（ctrl键旋转，上下左右键拉伸）

void processInput(GLFWwindow\* window, Player& player) {

// Esc键退出

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

glfwSetWindowShouldClose(window, true);

}

// ctrl键旋转

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

// 获取鼠标位置

double xPosIn, yPosIn;

glfwGetCursorPos(window, &xPosIn, &yPosIn);

// 坐标类型转换

float xPos = static\_cast<float>(xPosIn);

float yPos = static\_cast<float>(yPosIn);

// 首次移动处理

if (firstMouse) {

lastX = xPos;

lastY = yPos;

firstMouse = false;

}

// 计算偏移量

float xOffset = xPos - lastX;

float yOffset = lastY - yPos;

// 更新上一帧鼠标位置

lastX = xPos;

lastY = yPos;

// 调用摄像机的鼠标移动处理函数

camera.ProcessMouseMovement(xOffset, yOffset);

}

// 上下左右键拉伸

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

player.Stretch(UP);

}

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

player.Stretch(DOWN);

}

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

player.Stretch(LEFT);

}

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

player.Stretch(RIGHT);

}

}

// 监控鼠标点击（左键平移，右键复原）

void mouse\_callback(GLFWwindow\* window, Player& player) {

// 左键平移

if (glfwGetMouseButton(window, GLFW\_MOUSE\_BUTTON\_LEFT) == GLFW\_PRESS) {

// 获取鼠标位置

double xPosIn, yPosIn;

glfwGetCursorPos(window, &xPosIn, &yPosIn);

// 坐标类型转换

float xPos = static\_cast<float>(xPosIn);

float yPos = static\_cast<float>(yPosIn);

// 首次移动处理

if (firstMouse) {

lastX = xPos;

lastY = yPos;

firstMouse = false;

}

// 计算偏移量

float xOffset = xPos - lastX;

float yOffset = lastY - yPos;

// 更新上一帧鼠标位置

lastX = xPos;

lastY = yPos;

// 调用摄像机的鼠标点击处理函数

camera.ProcessMousePress(xOffset, yOffset, deltaTime);

}

// 右键复原

else if (glfwGetMouseButton(window, GLFW\_MOUSE\_BUTTON\_RIGHT) == GLFW\_PRESS) {

camera.ProcessMousePress();

player.Init();

}

}

// 监控鼠标滚轮（缩放）

void scroll\_callback(GLFWwindow\* window, double xOffset, double yOffset)

{

camera.ProcessMouseScroll(static\_cast<float>(yOffset));

}