计算机图形学与虚拟现实

实验报告

2025秋

课程类型 限选□ 选修□

实验题目 **金刚石的绘制**

姓　名 **邴海诺**

学　号 **2024112353**

学　院 **计算学部**

2025 年　月　日

一、实验目的

了解坐标变换，了解使用纹理的方法，编写着色器实现金刚石的绘制。

二、实验要求及实验环境

```
OpenGL version: 4.6 (Compatibility Profile) Mesa 25.2.7-cachyos1.2
GLEW version: 2.2.0
```

三、设计思想

Camera 类：抽象的摄像机类，包含位置向量、方向向量、欧拉角、移动速度、鼠标灵敏度等属性，顶点数据数组： 包含每个顶点的位置（aPos）和纹理坐标（aTexCoords）。

四、实验结果与分析（根据章节分节）

1. 摄像机类的使用和实现

首先是实现和使用摄像机。它负责模拟一个在三维虚拟世界中移动和旋转的观察者，主要作用是管理所有与视角相关的数据，并计算出将物体从世界空间转换到观察空间的观察矩阵（View Matrix）。为此参考代码指导构造摄像机类，其中包含了如下成员：

数据名称	类型	描述
位置 (Position)	glm::vec3	摄像机在世界坐标系中的坐标点。
前向量 (Front)	glm::vec3	摄像机当前指向的方向，用于前进/后退。
上向量 (Up)	glm::vec3	定义摄像机头顶的方向（通常固定为世界 Y 轴），用于生成观察矩阵。
右向量 (Right)	glm::vec3	定义摄像机右侧的方向，用于侧移（扫射）。

数据名称	类型	描述
世界法向 (World Up)	glm::vec3	世界坐标系中固定的向上向量，通常为(0.0f, 1.0f, 0.0f)。
偏航角 (Yaw)	float	欧拉角之一，用于控制摄像机水平（左右）旋转的角度。
俯仰角 (Pitch)	float	欧拉角之一，用于控制摄像机垂直（上下）旋转的角度。

摄像机类有三个核心功能：计算观察矩阵、处理键盘输入和处理鼠标输入。

1. 它根据摄像机的位置和方向，生成一个 4x4 的观察矩阵（View Matrix）。使用 lookat 函数 这个矩阵将世界坐标系中的所有物体转换到摄像机所处的观察空间。它本质上是将整个世界沿相反方向移动和旋转，使得摄像机（即观察者）位于原点且朝向负 Z 轴。

2. 通过鼠标移动来控制视角方向。原理是接收鼠标在 X 轴和 Y 轴上的偏移量。将偏移量乘以鼠标灵敏度，并加到当前的偏航角（Yaw）和俯仰角（Pitch）上。更新欧拉角后，调用 UpdateCameraVectors() 方法重新计算前向量(Front)、右向量 (Right) 和上向量 (Up)。

方向向量的更新是基于更新后的偏航角和俯仰角，通过三角函数完成的：

front.x=cos(yaw)*cos(pitch)

fornt.y=sin(pitch)

front.z=sin(yaw)*con(pitch)

计算出 Front 向量后，通过叉乘计算 Right 和 Up 向量，和 front 一起构成相机坐标系。

right=front x worldup

up=right x front

3. 通过键盘按键控制摄像机在空间中的移动 根据按下的方向键（如 WASD），确定要移动的方向：前进/后退沿着 Front 向量方向；向左/向右沿着 Right 向量方向；向上/向下 沿着 Up 或 WorldUp 向量方向移动距离是 distance = MovementSpeed * deltaTime。更新位置是 Position += Front * distance。

2. 导入纹理与使用纹理

使用 stb_image.h 库读取图像数据，并通过 glGenTextures、glBindTexture 和 glTexImage2D 等 API 创建并绑定纹理对象，将图像数据上传到 GPU 。

在定义金刚石的顶点数据时，除了位置（aPos），必须添加纹理坐标（Texture Coordinates / UV 坐标），(aPos.x, aPos.y, aPos.z, aTexCoords.u, aTexCoords.v)它定义了模型上的一个点对应到纹理

图像上的哪个位置，

顶点着色器负责接收输入的纹理坐标，并将其传递给下一阶段：接收来自 VBO 的纹理坐标属性 `layout (location = 1) in vec2 aTexCoords;`将纹理坐标作为 **`out`** 变量传递给片段着色器，光栅化阶段会自动对其进行插值。

片段着色器使用插值后的纹理坐标进行采样。接收来自顶点着色器插值后的纹理坐标 `in vec2 TexCoords;`声明一个特殊的 Uniform 变量 `sampler2D`，指向 GPU 内存中的纹理对象。使用内置的 `texture()` 函数，传入采样器和插值纹理坐标，获取颜色值。

3. 坐标变换

金刚石模型在三维世界中被正确放置（Model Matrix），并且根据摄像机的视角（View Matrix）和透视设置（Projection Matrix）正确投影到屏幕上。Model Matrix 通过平移、旋转和缩放操作，将金刚石的原始模型空间坐标转换到世界空间中期望的位置和大小。View Matrix 如前所述，由摄像机类计算，将世界空间坐标转换到摄像机观察空间，决定了“看”的位置和方向。Projection Matrix：实现了透视投影，将三维空间中的物体转换为二维视锥体内的裁剪空间坐标。最终，在顶点着色器中将这三个矩阵相乘应用于顶点。

五、结论

出现金刚石图像，可以旋转视角进行查看

六、参考文献

七、附录：源代码

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

#include <GLFW/glfw3.h>


#include <glm/fwd.hpp>

#include <iostream>


#include "Camera.hpp"

#include "Shader.hpp"
```

```
#define STB_IMAGE_IMPLEMENTATION

#include <filesystem>

#include "stb_image.h"

namespace fs = std::filesystem;

constexpr unsigned int SCR_WIDTH = 800;

constexpr unsigned int SCR_HEIGHT = 600;

Camera camera(glm::vec3(0.0f, 1.0f, 0.0f));

static float lastX = SCR_WIDTH / 2.0f;

static float lastY = SCR_HEIGHT / 2.0f;

static bool firstMouse = true;

static float deltaTime = 0.0f;

static float lastFrame = 0.0f;

void processInput(GLFWwindow* window) {

if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS) {

glfwSetWindowShouldClose(window, true);

}

if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) {

camera.processKeyboard(CameraMovement::FORWARD, deltaTime);
```

```
}

if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS) {
camera.processKeyboard(CameraMovement::BACKWARD, deltaTime);
}

if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS) {
camera.processKeyboard(CameraMovement::LEFT, deltaTime);
}

if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS) {
camera.processKeyboard(CameraMovement::RIGHT, deltaTime);
}

if (glfwGetKey(window, GLFW_KEY_LEFT_SHIFT) == GLFW_PRESS) {
camera.processKeyboard(CameraMovement::DOWN, deltaTime);
}

if (glfwGetKey(window, GLFW_KEY_SPACE) == GLFW_PRESS) {
camera.processKeyboard(CameraMovement::UP, deltaTime);
}
}

void mouseCallback(GLFWwindow* window, double xpos, double ypos)
{
```

```
if (firstMouse) {
lastX = xpos;
lastY = ypos;
firstMouse = false;
}

float xoffset = xpos - lastX;
float yoffset =
lastY - ypos; // 注意这里是相反的，因为y坐标是从底部往顶部依次增大的

lastX = xpos;
lastY = ypos;

camera.processMouseMovement(xoffset, yoffset);
}

void scrollCallback(GLFWwindow* window, double xoffset, double
yoffset) {
camera.processMouseScroll(static_cast<float>(yoffset));
}

unsigned int loadTexture(fs::path path) {
unsigned int textureID;
glGenTextures(1, &textureID);
```

```
int width, height, nrComponents;
unsigned char* data =
stbi_load(path.string().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); // 设置纹理环绕方式
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;
}

int main() {
// 初始化 GLFW
if (!glfwInit()) {
std::cerr << "Failed to initialize GLFW\n";
return -1;
}

// 创建窗口
GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT,
"Learn OpenGL",
nullptr, nullptr);
if (!window) {
```

```
std::cerr << "Failed to create window\n";
glfwTerminate();
return -1;
}

// 设置当前上下文
glfwMakeContextCurrent(window);

// 初始化 GLEW
GLenum err = glewInit();
if (err != GLEW_OK) {
std::cerr << "Failed to initialize GLEW: " <<
glewGetErrorString(err)
<< "\n";
return -1;
}

glfwSetCursorPosCallback(window, mouseCallback);
glfwSetScrollCallback(window, scrollCallback);
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);

std::cout << "OpenGL version: " << glGetString(GL_VERSION) <<
```

```
"\n";
std::cout << "GLEW version: " << glewGetString(GLEW_VERSION) <<
"\n";

float diamondVertices[] = {
// 前三个顶点位置 // 后两个纹理位置，即diamond.png的UV坐标
0.5f, 0.0f, 0.0f, 0.0f, 0.0f, // 左
0.0f, 0.5f, 0.0f, 1.0f, 0.0f, // 上
0.0f, 0.0f, 0.5f, 1.0f, 1.0f, // 前
-0.5f, 0.0f, 0.0f, 0.0f, 0.0f, // 右
0.0f, -0.5f, 0.0f, 1.0f, 0.0f, // 下
0.0f, 0.0f, -0.5f, 1.0f, 1.0f, // 后
};

unsigned int diamondIndices[] = {
2, 1, 0, // 前上左
2, 0, 4, // 前左下
2, 4, 3, // 前下右
2, 1, 3, // 前上右
4, 0, 5, // 后左下
0, 1, 5, // 后上左
1, 3, 5, // 后右上
4, 3, 5, // 后右下这里指导书里面给的不对
```

```
};

glEnable(GL_DEPTH_TEST);

unsigned int VBO, VAO, EBO;
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);
glGenBuffers(1, &VBO);
glGenBuffers(1, &EBO);
glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER,             sizeof(diamondVertices),
diamondVertices,
GL_STATIC_DRAW);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(diamondIndices),
diamondIndices, GL_STATIC_DRAW);

glVertexAttribPointer(0,    3,    GL_FLOAT,    GL_FALSE,    5    *
sizeof(float),
(void*)0);
glEnableVertexAttribArray(0);

glVertexAttribPointer(1,    2,    GL_FLOAT,    GL_FALSE,    5    *
```

```
sizeof(float),
(void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);

Shader shader_diamond("../../lab2/res/expr2.vert",
"../../lab2/res/expr2.frag");
unsigned int texture = loadTexture("../../lab2/res/diamond.png");

shader_diamond.use();
shader_diamond.setInt("texture1", 0);

while (!glfwWindowShouldClose(window)) {
glClearColor(0.6f, 0.6f, 0.6f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

float currentFrame = static_cast<float>(glfwGetTime());
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;

processInput(window);

shader_diamond.use();
//Model 矩阵:
shader_diamond.setMat4("model", glm::mat4(1.0f));
//View 矩阵:
```

```
shader_diamond.setMat4("view", camera.getViewMatrix());
//Projection 矩阵:
//最后在着色器里面相乘，实现透视投影
shader_diamond.setMat4(
"projection", glm::perspective(glm::radians(camera.zoom),
(float)SCR_WIDTH / (float)SCR_HEIGHT,
0.1f, 100.0f));
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 24, GL_UNSIGNED_INT, 0);

glfwSwapBuffers(window);
glfwPollEvents();
}

glfwDestroyWindow(window);
glfwTerminate();
return 0;
}
```