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	Opengl环境搭建
姓　名	邴海诺
学　号	2024112353
学　院	计算学部

2025 年　11 月 26 日

一、实验目的

搭建 OpenGL 实验环境

了解 OpenGL 的基本概念,包括绑定,缓冲区,着色器等

用 OpenGL 绘制一个简单的三角形

二、实验要求及实验环境

ubuntu22 .04

```
OpenGL version: 4.6 (Compatibility Profile) Mesa 23.2.1-1ubuntu3.1~22.04.3
GLEW version: 2.2.0
```

三、设计思想

无

四、实验结果与分析（根据章节分节）

4.1 opengl 环境配置：

sudo apt install libglfw3-dev libglm-dev

4.2 窗口创建：

OpenGL 自身是一个巨大的状态机(State Machine):一系列的变量描述 OpenGL 此刻应当如何运行。OpenGL 的状态通常被称为 OpenGL 上下文(Context)。

“窗口（window）”是由 GLFW 创建并由操作系统管理的可视化界面；它关联了一个 OpenGL 上下文（context）和帧缓冲（framebuffer）。程序通过这个窗

口显示渲染结果、接收输入事件（鼠标、键盘）并管理交换帧缓冲（双缓冲）。下面的代码展示了如何创建窗口对象并设置上下文

```
if (!glfwInit()) {
    std::cerr << "Failed to initialize GLFW\n";
    return -1;
}

// 创建窗口
GLFWwindow* window = glfwCreateWindow(width: 800, height: 600, title: "OpenGL Demo", monitor: nullptr, share: nullptr);
if (!window) {
    std::cerr << "Failed to create window\n";
    glfwTerminate();
    return -1;
}

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

只有当某个 OpenGL context 被绑定到当前线程后，才能调 OpenGL 的 API

glfwMakeContext Current(window) 把 window 对象对应的 OpenGL context 绑定到当前线程上。

4.3 三角形绘制

顶点数据在 CPU 上定义（顶点坐标数组），然后上传到 GPU 的缓冲（VBO）。VAO 记录顶点属性布局（每个属性的大小、类型、stride、offset）和它们对应的 VBO/EBO。着色器（顶点着色器和片段着色器）在 GPU 上执行，决定如何处理每个顶点和像素。最后通过 draw call（glDrawElements）告诉 GPU 使用存储的缓冲数据绘制。渲染结果写入窗口默认帧缓冲，经双缓冲显示（交换前后缓冲）。

定义三角形的位置：

```
41      float vertices[] = {
42          -0.5f, -0.5f, 0.0f, // 左下角
43           0.5f, -0.5f, 0.0f, // 右下角
44           0.0f,  0.5f, 0.0f  // 顶部
45      };
```

float 数组中每 3 个 float 表示一个顶点 (x, y, z)。值默认是 NDC（Normalized Device Coordinates）范围是[-1, 1]。

VAO 用于记录顶点元素的配置和关联 buffer。

```
47      // 1. 创建并绑定 VAO（必须最先做，这样后续的 VBO 配置都会被记录到这个 VAO 中）
48      unsigned int VAO;
49      glGenVertexArrays(1, &VAO);
50      glBindVertexArray(VAO);
```

创建并填充 VBO，数据传到 GPU 上

```
52      // 2. 创建并绑定 VBO，传输数据
53      unsigned int VBO;
54      glGenBuffers(1, &VBO);
55      glBindBuffer(GL_ARRAY_BUFFER, VBO);
56      glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

把 VBO 的数据记录到 VAO 中

```
59      // 3. 设置顶点属性指针（告诉 OpenGL 如何解析数据）（这一步会把指向 VBO 的信息记录到 VAO 里）
60      // 第一个参数 0：属性索引（location），要和顶点着色器里的 layout(location = 0) 对应。
61      // 3：每个顶点有 3 个分量（x,y,z）。
62      // GL_FLOAT：数据类型。
63      // GL_FALSE：是否归一化。
64      // 3 * sizeof(float)：stride（每个顶点占用的字节数）。
65      // (void*)0：属性在数组中的偏移（这里顶点位置从第 0 个 float 开始）。 启用属性后，VAO 会记录这项配置
66      glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
67      glEnableVertexAttribArray(0);
```

顶点着色器（Vertex Shader）接受顶点输入（位置、法线、颜色、纹理坐标等），负责将顶点从模型坐标或世界坐标变换到裁剪空间（NDC），即输出 gl_Position。

输入：layout(location = 0) in vec3 aPos; 用于位置。

exprl.vs

```
#version 330 core
layout(location = 0) in vec3 aPos;
void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

片段着色器（Fragment Shader）对每个片元（将成为屏幕上像素的候选）计算最终颜色。

输出：out vec4 FragColor;(rgb 颜色)

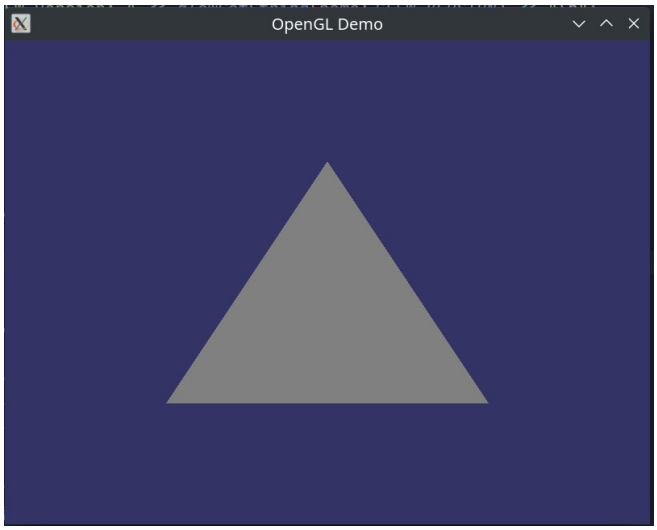
着色器的编写参考实验指导

exprl.fs

```
#version 330 core

out vec4 FragColor;
void main()
{
    FragColor = vec4(0.5f, 0.5f, 0.5f, 1.0f);
}
```

五、结论



六、参考文献

实验指导

七、附录：源代码

#include <GL/glew.h>

```
#include <GLFW/glfw3.h>
#include <iostream>
#include "shader.h"

void processInput(GLFWwindow* window) {
      if  (glfwGetKey(window,  GLFW_KEY_ESCAPE)  ==
GLFW_PRESS)
      glfwSetWindowShouldClose(window, true);
}

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

   // 创建窗口
    GLFWwindow* window = glfwCreateWindow(800,
600, "OpenGL Demo", 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（必须在上下文创建后）
GLenum err = glewInit();
if (err != GLEW_OK) {
    std::cerr << "Failed to initialize GLEW: "
              << glewGetErrorString(err) << "\n";
    return -1;
}

        std::cout << "OpenGL version: " << glGetString(GL_VERSION) << "\n";
        std::cout << "GLEW version: " << glewGetString(GLEW_VERSION) << "\n";

float vertices[] = {
```

```
    -0.5f, -0.5f, 0.0f, // 左下角
     0.5f, -0.5f, 0.0f, // 右下角
     0.0f,  0.5f, 0.0f  // 顶部
};

// 1. 创建并绑定 VAO (必须最先做，这样后续的 VBO 配置都会被记录到这个 VAO 中)
unsigned int VAO;
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

// 2. 创建并绑定 VBO，传输数据
unsigned int VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);


 // 3. 设置顶点属性指针 (告诉 OpenGL 如何解析数据)（这一步会把指向 VBO 的信息记录到 VAO 里）
// 第一个参数 0：属性索引（location），要和顶点着色
```

器里的 layout(location = 0) 对应。

// 3：每个顶点有 3 个分量 (x,y,z)。

// GL_FLOAT：数据类型。

// GL_FALSE：是否归一化。

// 3 * sizeof(float)：stride（每个顶点占用的字节数）。

// (void*)0：属性在数组中的偏移（这里顶点位置从第 0 个 float 开始）。 启用属性后，VAO 会记录这项配置

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

glEnableVertexAttribArray(0);

// 4. 解绑 VBO (可选，但 VAO 此时已记录了 VBO，所以解绑 VBO 不影响 VAO)

// glBindBuffer(GL_ARRAY_BUFFER, 0);

// 5. 解绑 VAO (防止意外修改)

glBindVertexArray(0);

Shader shader_triangle("../exprl.vs", "../exprl.fs");

shader_triangle.use();

// 主循环

```
    while (!glfwWindowShouldClose(window)) {
        // 处理输入 esc 键
        processInput(window);
        // 设置清屏颜色（RGBA）
        glClearColor(0.2f, 0.2f, 0.4f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        shader_triangle.use();
        // 绘制三角形
        glBindVertexArray(VAO);
        glDrawArrays(GL_TRIANGLES, 0, 3);

        // 交换缓冲区
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    // 清理资源
    glfwDestroyWindow(window);
    glfwTerminate();
    return 0;
}
```

```
//
//shader.h
//
#ifndef SHADER_H
#define SHADER_H
#include <glm/glm.hpp>
#include <GL/glew.h>
#include <string>

class Shader {
public:
    unsigned int ID;
        Shader(const std::string& vertexPath, const std::string& fragmentPath);
    void use();

};

#endif

#include "shader.h"
```

```
#include <fstream>
#include <sstream>
#include <iostream>
#include <glm/gtc/type_ptr.hpp>

Shader::Shader(const std::string& vertexPath, const
std::string& fragmentPath) {
   std::string vertexCode;
   std::string fragmentCode;
   std::ifstream vShaderFile;
   std::ifstream fShaderFile;

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

    unsigned int vertex, fragment;
    int success;
    char infoLog[512];

    vertex = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex, 1, &vShaderCode, NULL);
```

```
    glCompileShader(vertex);
        glGetShaderiv(vertex, GL_COMPILE_STATUS,
&success);
    if (!success) {
        glGetShaderInfoLog(vertex, 512, NULL, infoLog);
                                        std::cout <<
"ERROR::SHADER::VERTEX::COMPILATION_FAILED
\n" << infoLog << std::endl;
    }

                                        fragment =
glCreateShader(GL_FRAGMENT_SHADER);
        glShaderSource(fragment, 1, &fShaderCode,
NULL);
    glCompileShader(fragment);
        glGetShaderiv(fragment, GL_COMPILE_STATUS,
&success);
    if (!success) {
            glGetShaderInfoLog(fragment, 512, NULL,
infoLog);
                                        std::cout <<
"ERROR::SHADER::FRAGMENT::COMPILATION_FAI
```

```
LED\n" << infoLog << std::endl;
    }

    ID = glCreateProgram();
    glAttachShader(ID, vertex);
    glAttachShader(ID, fragment);
    glLinkProgram(ID);
    glGetProgramiv(ID, GL_LINK_STATUS, &success);
    if (!success) {
        glGetProgramInfoLog(ID, 512, NULL, infoLog);
                                                std::cout       <<
"ERROR::SHADER::PROGRAM::LINKING_FAILED\n"
<< infoLog << std::endl;
    }

    ID = glCreateProgram();
    glAttachShader(ID, vertex);
    glAttachShader(ID, fragment);
    glLinkProgram(ID);
    glGetProgramiv(ID, GL_LINK_STATUS, &success);
    if (!success) {
        glGetProgramInfoLog(ID, 512, NULL, infoLog);
```

```
                                        std::cout       <<
"ERROR::SHADER::PROGRAM::LINKING_FAILED\n"
<< infoLog << std::endl;
    }

    glDeleteShader(vertex);
    glDeleteShader(fragment);
}

void Shader::use() {
    glUseProgram(ID);
}

#version 330 core

out vec4 FragColor;
void main()
{
    FragColor = vec4(0.5f, 0.5f, 0.5f, 1.0f);
}

#version 330 core
```

```
layout(location = 0) in vec3 aPos;
void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```