

绘制三角形

华中科技大学软件学院 万琳





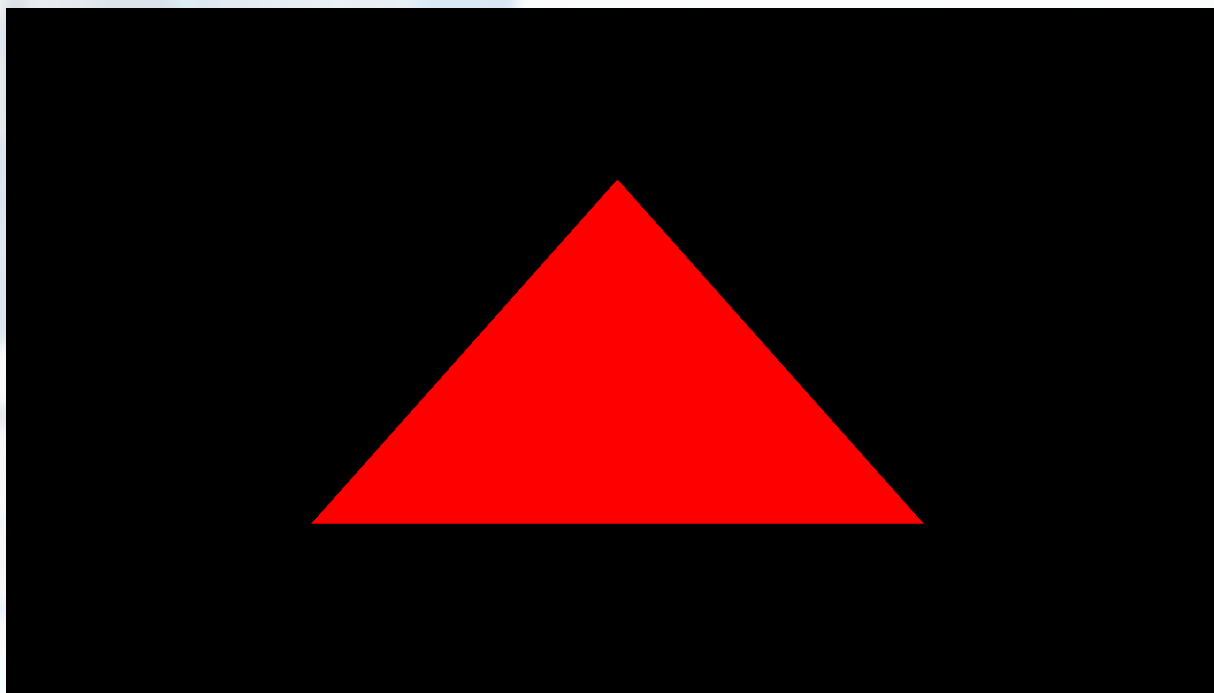
提纲

- ① 实验要求
- ② 程序流程
- ③ 要点解析
- ④ 程序演示

1

实验要求

在窗口中绘制一个三角形。



2

程序流程

在窗口中绘制一个三角形。



3

要点解析

➤初始化：初始化GLFW和GLAD



3

要点解析

➤初始化：初始化GLFW和GLAD



初始化
GLFW

```
glfwInit(); // 初始化GLFW
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
// OpenGL版本为3.3，主版本号均设为3
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
// OpenGL版本为3.3，次版本号均设为3
glfwWindowHint(GLFW_OPENGL_PROFILE,
GLFW_OPENGL_CORE_PROFILE); // 使用核心模式（无需向后兼容性）
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
// 如果使用的是Mac OS X系统，需加上这行
glfwWindowHint(GLFW_RESIZABLE, FALSE); // 不可改变窗口大小
```

3

要点解析

➤初始化：初始化GLFW和GLAD



创建窗口

// 窗口宽, 高

```
int screen_width = 1280;
```

```
int screen_height = 720;
```

// 创建窗口(宽、高、窗口名称)

```
auto window = glfwCreateWindow(screen_width, screen_height,  
    "Computer Graphics", nullptr, nullptr);
```

// 如果窗口创建失败, 输出Failed to Create OpenGL Context

```
if (window == nullptr)
```

```
    {std::cout << "Failed to Create OpenGL Context" << std::endl;
```

```
    glfwTerminate();
```

```
    return -1;}
```

// 将窗口的上下文设置为当前线程的主上下文

```
glfwMakeContextCurrent(window);
```

3

要点解析

➤初始化：初始化GLFW和GLAD



初始化 GLAD

// 初始化GLAD，加载OpenGL函数指针地址的函数

```
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    std::cout << "Failed to initialize GLAD" << std::endl;
    return -1;
}
```


3

要点解析

➤初始化：初始化GLFW和GLAD



创建视口

// 指定当前视口尺寸(前两个参数为左下角位置，后两个参数是渲染窗口宽、高)

```
glViewport(0, 0, screen_width, screen_height);
```

3

要点解析

➤ 数据处理：生成和绑定VBO和VAO，设置属性指针



3

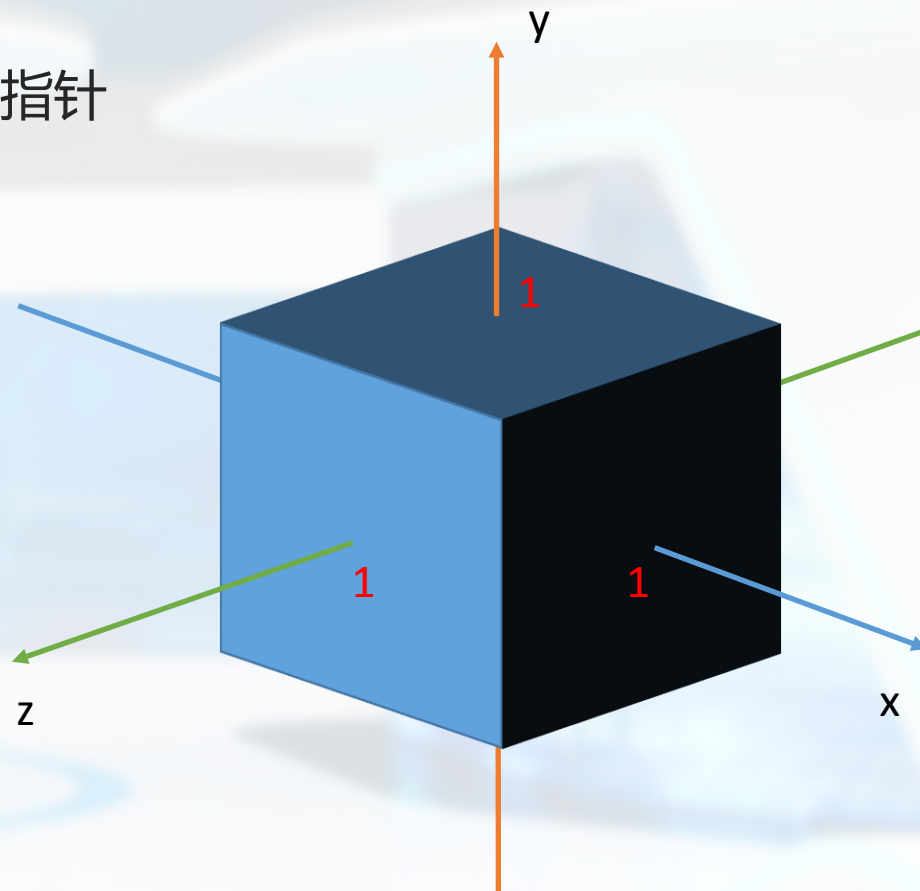
要点解析

➤ 数据处理：生成和绑定VBO和VAO，设置属性指针



顶点数据定义

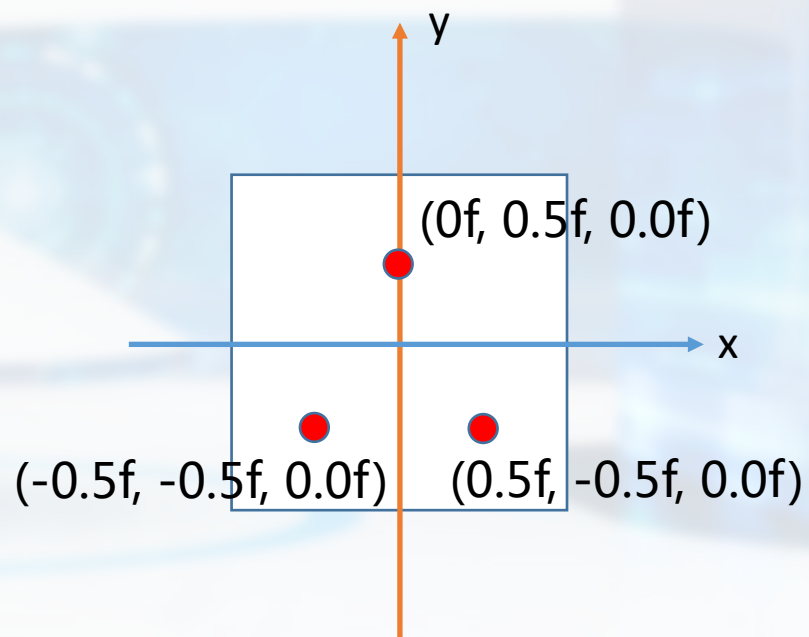
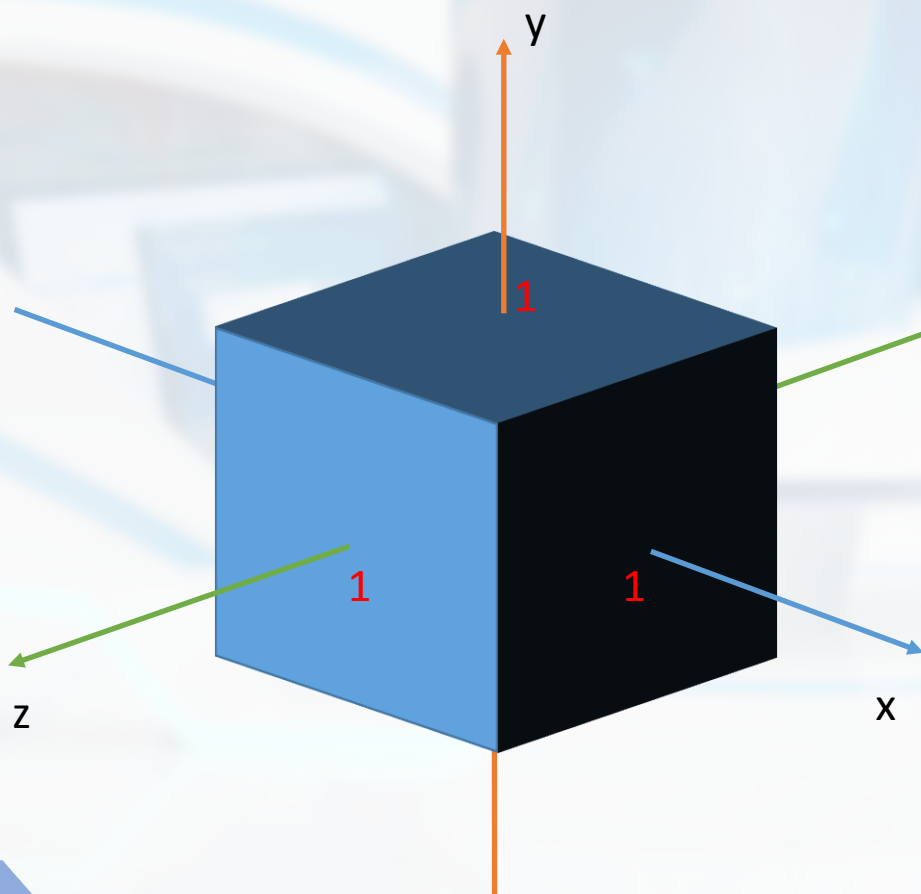
```
// 三角形的顶点数据
const float triangle[] = {
// ---- 位置 ----
-0.5f, -0.5f, 0.0f, // 左下
0.5f, -0.5f, 0.0f, // 右下
0.0f, 0.5f, 0.0f // 正上
};
```



3

要点解析

➤ 数据处理：生成和绑定VBO和VAO，设置属性指针



3

要点解析

➤ 数据处理：生成和绑定VBO和VAO，设置属性指针



生成与绑定
VAO、VBO

// 生成并绑定立方体的VAO和VBO

GLuint vertex_array_object; // VAO

glGenVertexArrays(1, &vertex_array_object);

glBindVertexArray(vertex_array_object);

GLuint vertex_buffer_object; // VBO

glGenBuffers(1, &vertex_buffer_object);

glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer_object);

// 将顶点数据绑定至当前默认的缓冲中

**glBufferData(GL_ARRAY_BUFFER, sizeof(triangle), triangle,
GL_STATIC_DRAW);**

3

要点解析

➤ 数据处理：生成和绑定VBO和VAO，设置属性指针



属性设置

// 设置顶点属性指针

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

```
glEnableVertexAttribArray(0);
```

3

要点解析

➤ 数据处理：生成和绑定VBO和VAO，设置属性指针



解绑

// 解绑VAO和VBO

```
glBindVertexArray(0);
```

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

问题：为什么要解绑呢？

3

要点解析

➤着色器：顶点和片段着色器



3

要点解析

➤着色器：顶点和片段着色器



着色器源码

// 顶点着色器和片段着色器源码

```
const char *vertex_shader_source =
```

```
"#version 330 core\n"
```

```
"layout (location = 0) in vec3 aPos;\n " // 位置变量的属性位置值为0
```

```
"void main()\n"
```

```
"{\n"
```

```
"    gl_Position = vec4(aPos, 1.0);\n"
```

```
"}\n\n0";
```

```
const char *fragment_shader_source =
```

```
"#version 330 core\n"
```

```
"out vec4 FragColor;\n " // 输出的颜色向量
```

```
"void main()\n"
```

```
"{\n"
```

```
"    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
```

```
"}\n\n0";
```

3

要点解析

➤着色器：顶点和片段着色器



生成并编译
着色器

```
// 生成并编译着色器
// 顶点着色器
int vertex_shader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex_shader, 1, &vertex_shader_source, NULL);
glCompileShader(vertex_shader);
int success;
char info_log[512];
// 检查着色器是否成功编译，如果编译失败，打印错误信息
glGetShaderiv(vertex_shader, GL_COMPILE_STATUS, &success);
if (!success)
{
    glGetShaderInfoLog(vertex_shader, 512, NULL, info_log);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n"
<< info_log << std::endl;
}
```

3

要点解析

➤着色器：顶点和片段着色器



生成并编译
着色器

```
// 生成并编译着色器
```

```
// 片段着色器
```

```
int fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);  
glShaderSource(fragment_shader, 1, &fragment_shader_source,  
NULL);
```

```
glCompileShader(fragment_shader);
```

```
// 检查着色器是否成功编译，如果编译失败，打印错误信息
```

```
glGetShaderiv(fragment_shader, GL_COMPILE_STATUS, &success);
```

```
if (!success)
```

```
{
```

```
    glGetShaderInfoLog(fragment_shader, 512, NULL, info_log);
```

```
    std::cout <<
```

```
"ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << info_log
```

```
<< std::endl;
```

```
}
```

3

要点解析

➤着色器：顶点和片段着色器



链接着色器
到着色器程序

```
// 链接顶点和片段着色器至一个着色器程序
int shader_program = glCreateProgram();
glAttachShader(shader_program, vertex_shader);
glAttachShader(shader_program, fragment_shader);
glLinkProgram(shader_program);
// 检查着色器是否成功链接，如果链接失败，打印错误信息
glGetProgramiv(shader_program, GL_LINK_STATUS, &success);
if (!success) {
    glGetProgramInfoLog(shader_program, 512, NULL, info_log);
    std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" <<
    info_log << std::endl;
}
```


3

要点解析

➤着色器：顶点和片段着色器



删除着色器

// 删除着色器

```
glDeleteShader(vertex_shader);  
glDeleteShader(fragment_shader);
```

3

要点解析

➤渲染

```
while (!glfwWindowShouldClose(window)) {
```

```
    // 清空颜色缓冲
```

```
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
```

```
    glClear(GL_COLOR_BUFFER_BIT);
```

```
    // 使用着色器程序
```

```
    glUseProgram(shader_program);
```

```
    // 绘制三角形
```

```
    glBindVertexArray(vertex_array_object); // 绑定VAO
```

```
    glDrawArrays(GL_TRIANGLES, 0, 3); // 绘制三角形
```

```
    glBindVertexArray(0); // 解除绑定
```

```
    // 交换缓冲并且检查是否有触发事件(比如键盘输入、鼠标移动等 )
```

```
    glfwSwapBuffers(window);
```

```
    glfwPollEvents();
```

```
}
```

3

要点解析

➤ 善后工作

```
// 删除VAO和VBO  
glDeleteVertexArrays(1, &vertex_array_object);  
glDeleteBuffers(1, &vertex_buffer_object);
```

```
// 清理所有的资源并正确退出程序  
glfwTerminate();  
return 0;
```

3

程序流程

在窗口中绘制一个三角形。

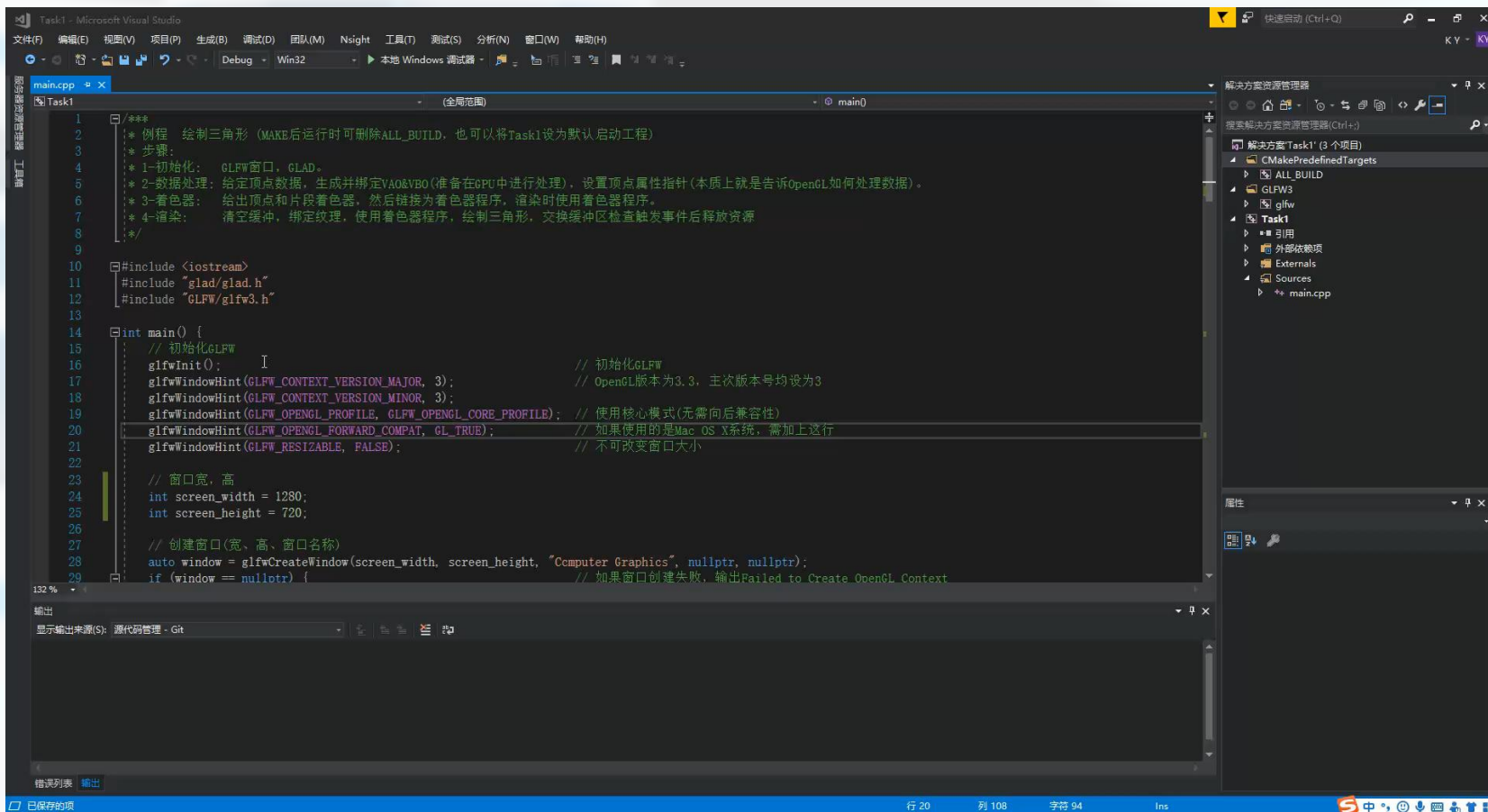
思考：在绘制三角形的基础上，如何绘制一个四边形呢？绘制四边形是否可以用到我们之前介绍过的EBO的知识呢？



4

程序演示

【运行】➡【更改窗口尺寸】➡【更改三角形颜色】➡【更改背景色】➡【开启线框模式】



```
1  /* 例程 绘制三角形 (MAKE后运行时可删除ALL_BUILD, 也可以将Task1设为默认启动工程)
2  * 步骤:
3  * 1-初始化:  GLFW窗口, GLAD.
4  * 2-数据处理:  给定顶点数据, 生成并绑定VAO&VBO (准备在GPU中进行处理), 设置顶点属性指针 (本质上就是告诉OpenGL如何处理数据)。
5  * 3-着色器:   给出顶点和片段着色器, 然后链接为着色器程序, 渲染时使用着色器程序。
6  * 4-渲染:    清空缓冲, 绑定纹理, 使用着色器程序, 绘制三角形, 交换缓冲区检查触发事件后释放资源
7  */
8
9
10 #include <iostream>
11 #include "glad/glad.h"
12 #include "GLFW/glfw3.h"
13
14 int main() {
15     // 初始化GLFW
16     glfwInit();
17     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3); // 初始化GLFW
18     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3); // OpenGL版本为3.3, 主次版本号均设为3
19     glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE); // 使用核心模式(无需向后兼容性)
20     glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); // 如果使用的是Mac OS X系统, 需加上这行
21     glfwWindowHint(GLFW_RESIZABLE, FALSE); // 不可改变窗口大小
22
23     // 窗口宽, 高
24     int screen_width = 1280;
25     int screen_height = 720;
26
27     // 创建窗口(宽、高、窗口名称)
28     auto window = glfwCreateWindow(screen_width, screen_height, "Computer Graphics", nullptr, nullptr);
29     if (window == nullptr) {
30         // 如果窗口创建失败, 输出Failed to Create OpenGL Context
31     }
32 }
```



谢谢

软件学院 万琳