Homework2 - GUI and Draw simple graphics

黄树凯

16340085

作业要求

1. 使用OpenGL(3.3及以上)+GLFW或freeglut画一个简单的三角形。
2. 对三角形的三个顶点分别改为红绿蓝，像下面这样。并解释为什么会出现这样的结果。
3. 给上述工作添加一个GUI，里面有一个菜单栏，使得可以选择并改变三角形的颜色。

Bonus:

1. 绘制其他的图元，除了三角形，还有点、线等。
2. 使用EBO(Element Buffer Object)绘制多个三角形。

实现及结果

准备工作

- 初始化GLFW
  - 调用glfwInit函数来初始化GLFW
  - 用 glfwWindowHint 函数来配置GLFW
  - 使用OpenGL 3.3 版本
  - 使用 GLFW 核心模式

```
glfwInit();
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);  // 主版本号 3
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);  // 次版本号 3
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);  // 使用核心模式
```

- 创建一个窗口对象
  - 前两个参数为窗口的长宽
  - 第三个参数为窗口的标题

```
/* 创建一个窗口对象 */
GLFWwindow* window = glfwCreateWindow(960, 650, "LearnOpenGL", NULL, NULL);
if (window == NULL) {
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
```

- 将窗口上下文设置为主上下文

```
glfwMakeContextCurrent(window);
glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
glfwSwapInterval(1);
```

- 初始化 GLAD

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

- 用着色器绘制

  着色器是使用一种叫GLSL的类C语言写成的。GLSL是为图形计算量身定制的，它包含一些针对向量和矩阵操作的有用特性。着色器的开头总是要声明版本，接着是输入和输出变量、uniform和main函数。每个着色器的入口点都是main函数，在这个函数中处理所有的输入变量，并将结果输出到输出变量中。

  - 顶点着色器

    ```
    #version 330 core
    layout (location = 0) in vec3 aPos;
    layout (location = 1) in vec3 aColor;

    out vec3 ourColor;

    void main() {
        gl_Position = vec4(aPos, 1.0);
        ourColor = aColor;
    }
    ```

  - 片段着色器

    ```
    #version 330 core
    out vec4 FragColor;
    in vec3 ourColor;

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

  - 编译着色器

    用 GLSL 语言编写并在运行时进行动态编译使 OpenGL 能够使用着色器。

```
unsigned int vertexShader;
vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);

unsigned int fragmentShader;
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);
```

- 链接顶点属性

  创建一个程序对象

```
unsigned int shaderProgram;
shaderProgram = glCreateProgram();
```

  glCreateProgram函数创建一个程序，并返回新创建程序对象的ID引用。把之前编译的着色器附加到程序对象上，然后用 glLinkProgram 链接它们

```
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
```

- 着色器类

  对上面的着色器的实现可以封装成类以便于使用且方便移植。类的基本结构如下：

```
#ifndef SHADER_H
#define SHADER_H

#include <glad/glad.h>; // 包含glad来获取所有的必须OpenGL头文件

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>

class Shader {
public:
    // 程序ID
    unsigned int ID;

    // 构造器读取并构建着色器
    Shader(const GLchar* vertexPath, const GLchar* fragmentPath);
    // 使用/激活程序
    void use();
    // uniform工具函数
    void setBool(const std::string &name, bool value) const;
    void setInt(const std::string &name, int value) const;
    void setFloat(const std::string &name, float value) const;
};
```

```
#endif
```

使用着色器类只需要将前面的顶点着色器和片段着色器的 GLSL 代码文件引入构造着色器对象，调用 use() 函数即可激活使用程序。

- 渲染

  把所有的渲染(Rendering)操作放到渲染循环中，让这些渲染指令在每次渲染循环迭代的时候都能被执行

```
// 渲染循环
while(!glfwWindowShouldClose(window)) {
    // 输入
    processInput(window);
    // 渲染指令
    ...
    // 检查并调用事件，交换缓冲
    glfwPollEvents();
    glfwSwapBuffers(window);
}
```

绘制三角形

- 定义三角形三个顶点的坐标

  渲染一个三角形，一共要指定三个顶点，每个顶点都有一个3D位置。将它们以标准化设备坐标的形式定义为一个`float`数组。

```
  float vertices[] = {
      //x     y     z
      0.3f, 0.3f, 0.0f, // 顶部
      0.3f, -0.3f, 0.0f, // 左
      -0.3f, -0.3f, 0.0f  // 右
  };
```

由于OpenGL是在3D空间中工作的，而渲染的是一个2D三角形，将它顶点的z坐标设置为0.0。这样三角形每一点的深度都是一样的，从而使它看上去像是2D的。

- 三角形的颜色由片段着色器进行设置

  在计算机图形中颜色被表示为有4个元素的数组：红色、绿色、蓝色和透明度分量，通常缩写为RGBA。当在OpenGL或GLSL中定义一个颜色的时候，把颜色每个分量的强度设置在0.0到1.0之间。

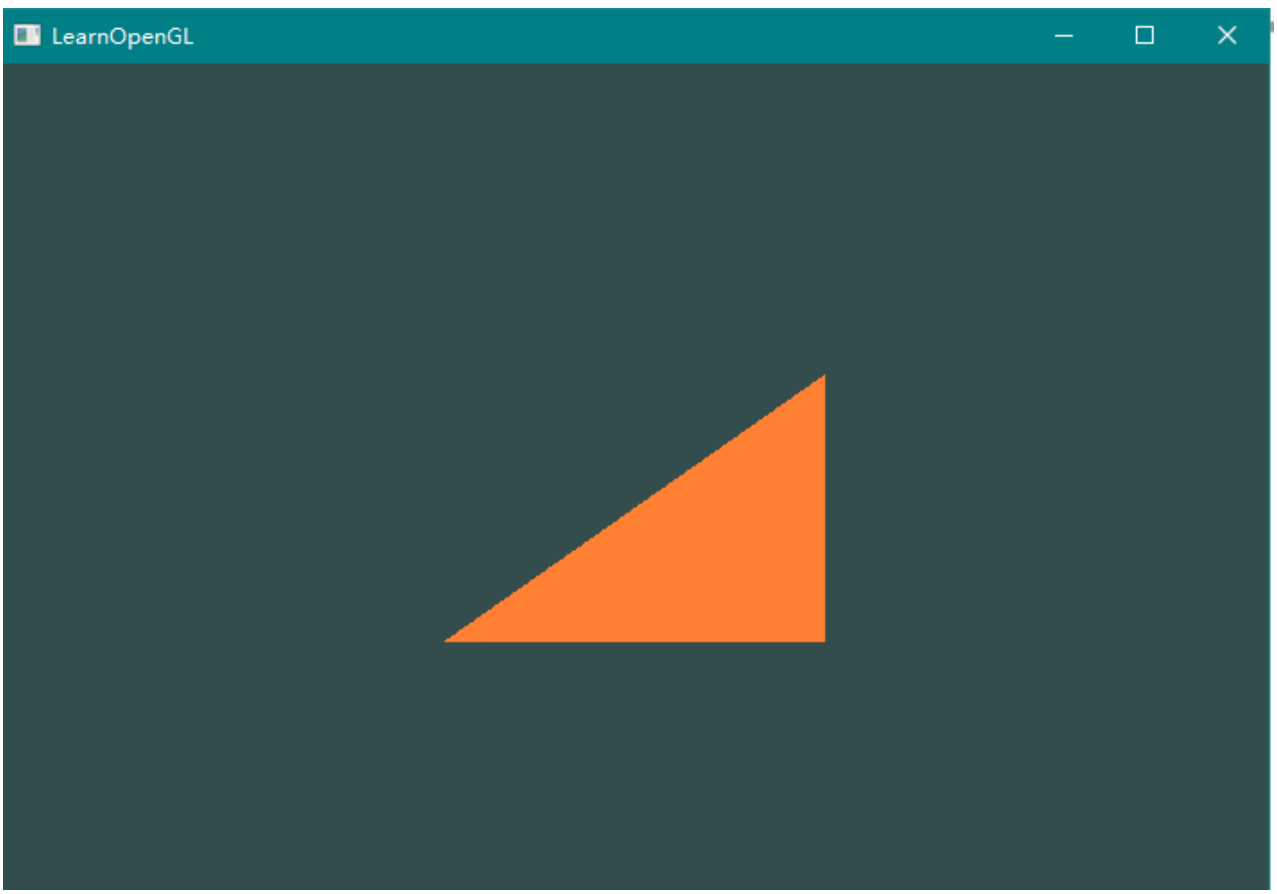
```
#version 330 core
out vec4 FragColor;

void main() {
    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}
```

片段着色器只需要一个输出变量，这个变量是一个4分量向量，它表示的是最终的输出颜色

```
//生成VAO、VBO、EBO对象
```

```
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glGenBuffers(1, &EBO);
// 1. 绑定顶点数组对象
glBindVertexArray(VAO);
// 2. 把顶点数组复制到一个顶点缓冲中，供OpenGL使用
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. 复制索引数组到一个索引缓冲中，供OpenGL使用
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
// 4. 设定顶点属性指针
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// 5. 设定顶点颜色属性指针
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 *
sizeof(float)));
// 6.创建一个程序对象
// 绘制出三角形
glEnableVertexAttribArray(1);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);
```



对三角形的三个顶点分别改为红绿蓝

把颜色数据添加为3个float值至vertices数组。将三角形的三个角分别指定为红色、绿色和蓝色

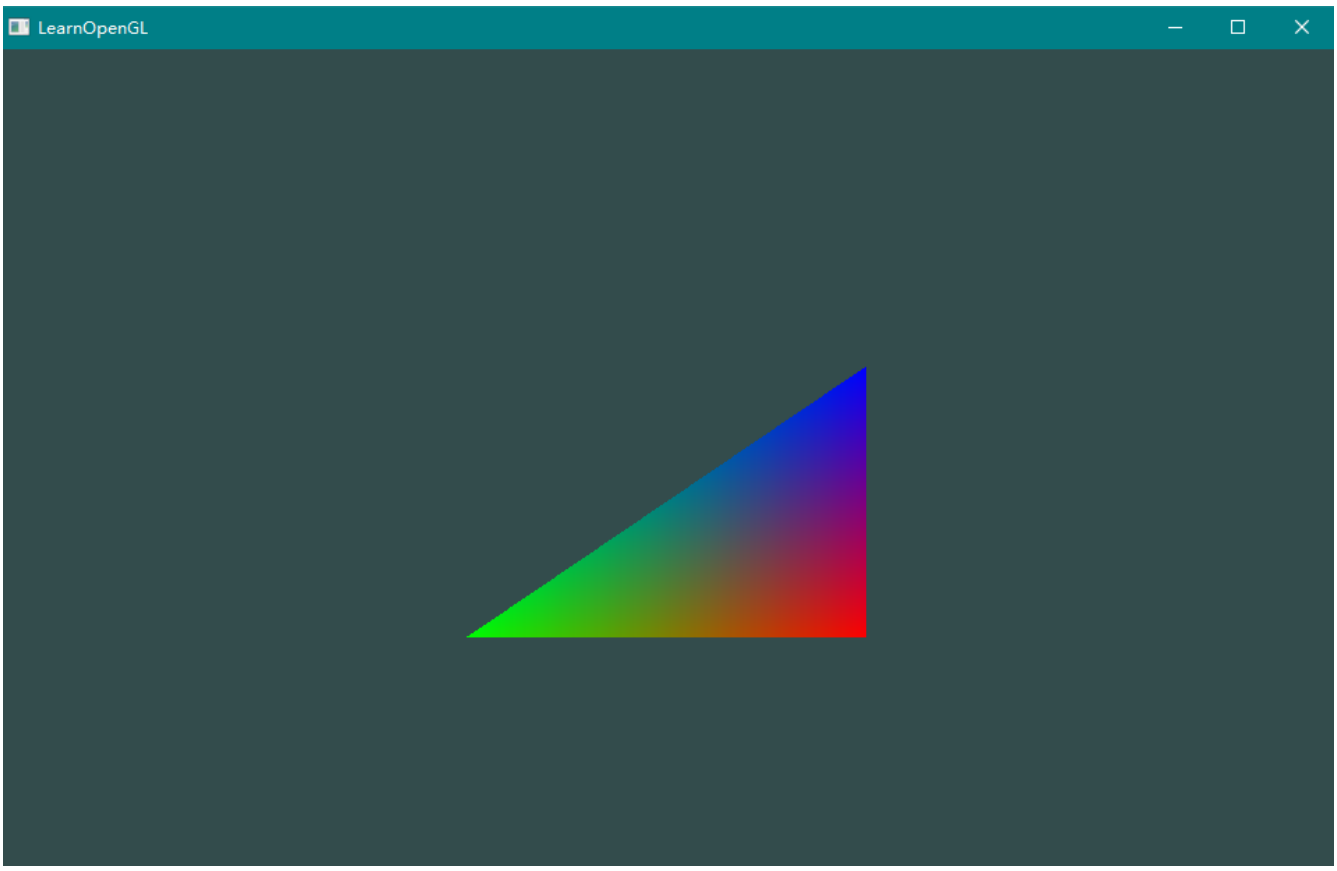
```
float vertices[] = {
    // 位置              // 颜色
     0.3f, 0.3f, 0.0f,  0.0f, 0.0f, 1.0f    // 顶部
     -0.3f, -0.3f, 0.0f,  0.0f, 1.0f, 0.0f,   // 左
     0.3f, -0.3f, 0.0f,  1.0f, 0.0f, 0.0f,   // 右
};
```

修改顶点着色器，使它能够接收颜色值作为一个顶点属性输入

修改片段着色器，并更新VBO的内存，重新配置顶点属性指针

```
// 位置属性
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// 颜色属性
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3*
sizeof(float)));
glEnableVertexAttribArray(1);
```

为获得数据队列中下一个属性值，必须向右移动6个float，其中3个是位置值，另外3个是颜色值。这使步长值为6乘以float的字节数（=24字节）。 同样，这次必须指定一个偏移量。对于每个顶点来说，位置顶点属性在前，所以它的偏移量是0。颜色属性紧随位置数据之后，所以偏移量就是`3 * sizeof(float)`，用字节来计算就是12字节。



解释为什么会出现图中的效果

这是在片段着色器中进行的所谓片段插值(Fragment Interpolation)的结果。当渲染一个三角形时，光栅化(Rasterization)阶段通常会造成比原指定顶点更多的片段。光栅会根据每个片段在三角形形状上所处相对位置决定这些片段的位置。 基于这些位置，它会插值(Interpolate)所有片段着色器的输入变量。比如说，我们有一个线段，上面的端点是绿色的，下面的端点是蓝色的。如果一个片段着色器在线段的70%的位置运行，它的颜色输入属性就会是一个绿色和蓝色的线性结合；更精确地说就是30%蓝 + 70%绿。

添加ImGui菜单

- 创建并绑定GUI

```
//创建并绑定ImGui
ImGui::CreateContext();
ImGuiIO& io = ImGui::GetIO(); (void)io;
ImGui_ImplOpenGL3_Init();
ImGui_ImplGlfw_InitForOpenGL(window, true);
ImGui::StyleColorsClassic();
```

- 在渲染循环中创建 ImGui

```
ImGui_ImplOpenGL3_NewFrame();
ImGui_ImplGlfw_NewFrame();
ImGui::NewFrame();
ImGui::Begin("Edit color", &ImGui, ImGuiWindowFlags_MenuBar); // 指定菜单栏标题
```

创建ColorEdit3生成一个调色板UI

```
ImGui::ColorEdit3("total color", (float*)&total_color);
// 指定该调色板名称，参数二为绑定的数据
```

创建Checkbox生成一个选择框UI，绑定布尔型的数据，并在每次循环渲染中判断该变量的值从而实现不同的功能控制

```
ImGui::Checkbox("Wireframe Mode", &wireframe_mode);
// 指定该UI的名称，参数二为绑定的数据
```

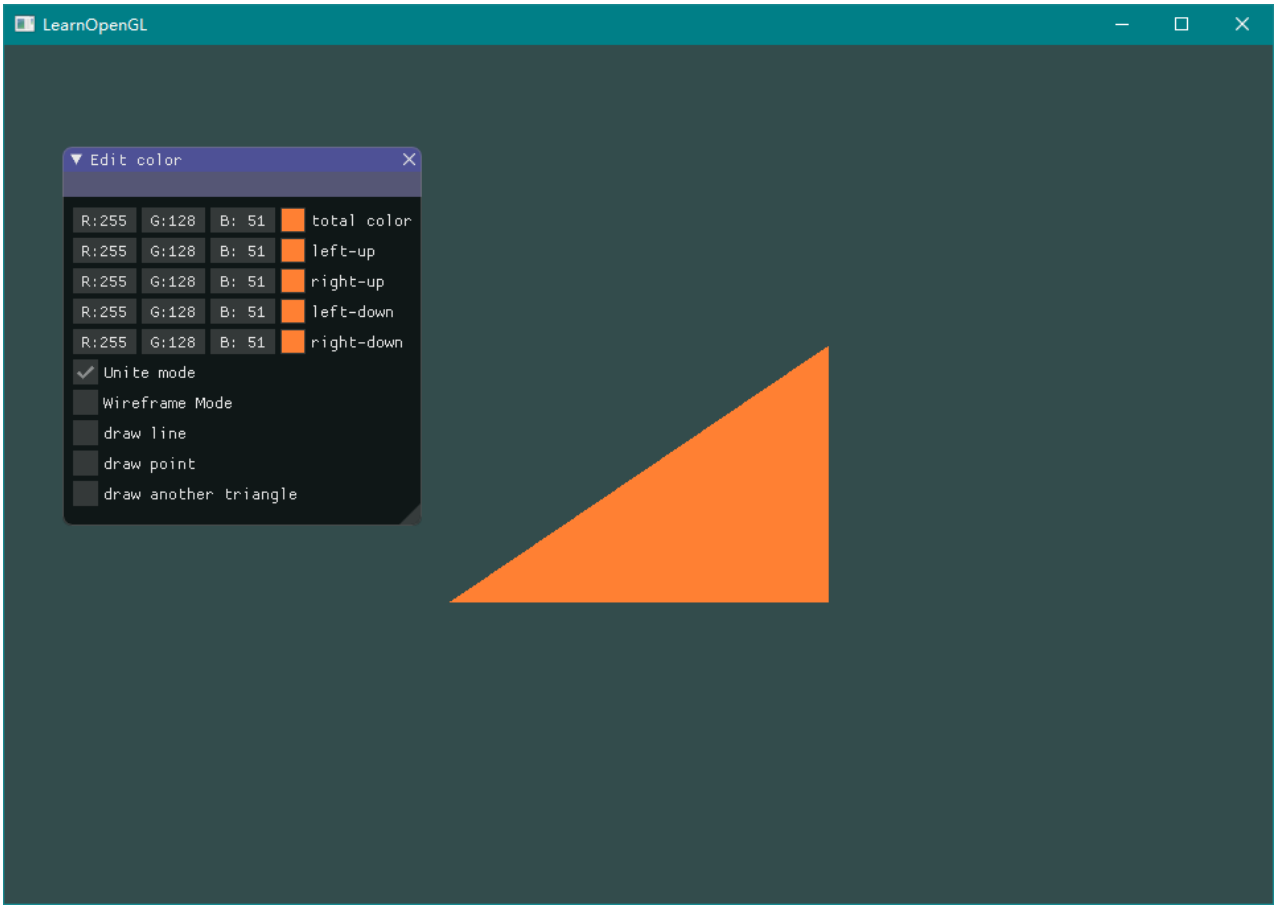
渲染imGui 菜单

```
ImGui::Render();
ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());
```

在每个vertice初始化时将其颜色数据赋值为特定的几个变量，再将几个变量用调色UI进行绑定，若操作调色板，在下一轮重复渲染时就会达到颜色改变的效果

```
float vertices[] = {
    0.3f, 0.3f, 0.0f, right_up_color.x, right_up_color.y, right_up_color.z,
    0.3f, -0.3f, 0.0f, right_down_color.x, right_down_color.y,right_down_color.z,
    -0.3f, -0.3f, 0.0f, left_down_color.x, left_down_color.y, left_down_color.z
}
```

实现效果



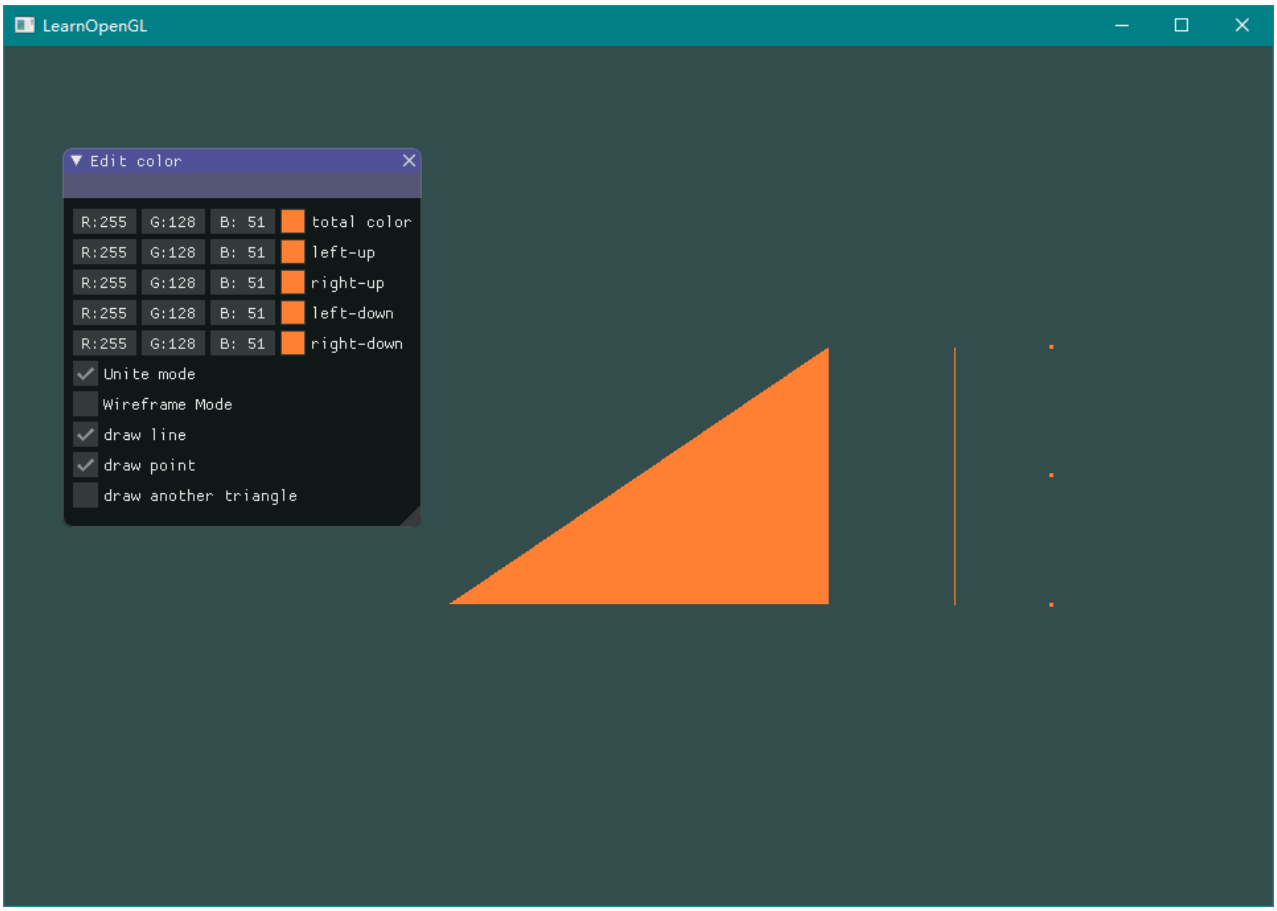
绘制其他图元

- 在vertices数组中加上指定直线或点的坐标

```
// 坐标点集
float vertices[] = {
// 三角形
0.3f, 0.3f, 0.0f, right_up_color.x, right_up_color.y, right_up_color.z,
0.3f, -0.3f, 0.0f, right_down_color.x, right_down_color.y, right_down_color.z,
-0.3f, -0.3f, 0.0f, left_down_color.x, left_down_color.y, left_down_color.z,
-0.3f, 0.3f, 0.0f, left_up_color.x, left_up_color.y, left_up_color.z,
// 线段
0.5f, -0.3f, 0.0f, right_down_color.x, right_down_color.y, right_down_color.z,
0.5f, 0.3f, 0.0f, right_up_color.x, right_up_color.y, right_up_color.z,
// 点
0.65, 0.3f, 0.0f, right_down_color.x, right_down_color.y, right_down_color.z,
0.65, -0.3f, 0.0f, right_up_color.x, right_up_color.y, right_up_color.z,
0.65, 0.0f, 0.0f, left_down_color.x, left_down_color.y, left_down_color.z
};
```

- 定义布尔型变量绑定到ImGui的Checkbox，在每次渲染循环中判断布尔值是否为真即是否被选中，来决定是否画出直线或点
- 绘制直线或点只需要改变指定的数组起始指针位置和元素个数，以及默认的直线（点）指定参数

```
// 渲染线段
if (draw_line) {
    glEnableVertexAttribArray(1);
    glBindVertexArray(VAO);
    glDrawArrays(GL_LINE_STRIP, 4, 2);
}
// 渲染点
if (draw_point) {
    glEnableVertexAttribArray(1);
    glBindVertexArray(VAO);
    glPointSize(3.0f);
    glDrawArrays(GL_POINTS, 6, 1);
    glDrawArrays(GL_POINTS, 7, 1);
    glDrawArrays(GL_POINTS, 8, 1);
}
```



使用EBO绘制

- 定义一个索引数组，用来指定两个三角形的三个顶点

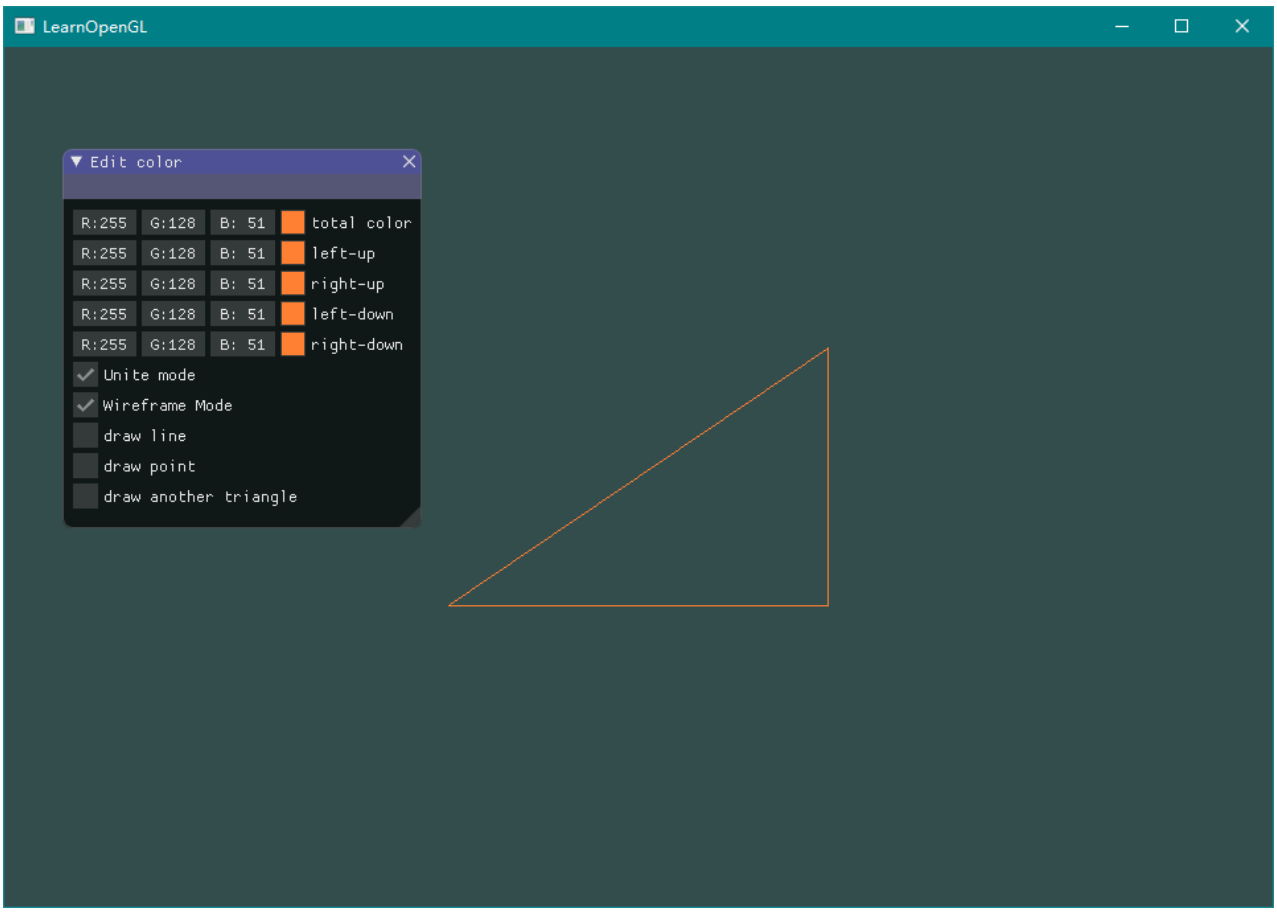
```
unsigned int indices[] = { // 注意索引从0开始
    0, 1, 2, // 第一个三角形
    0, 2, 3  // 第二个三角形
};
```

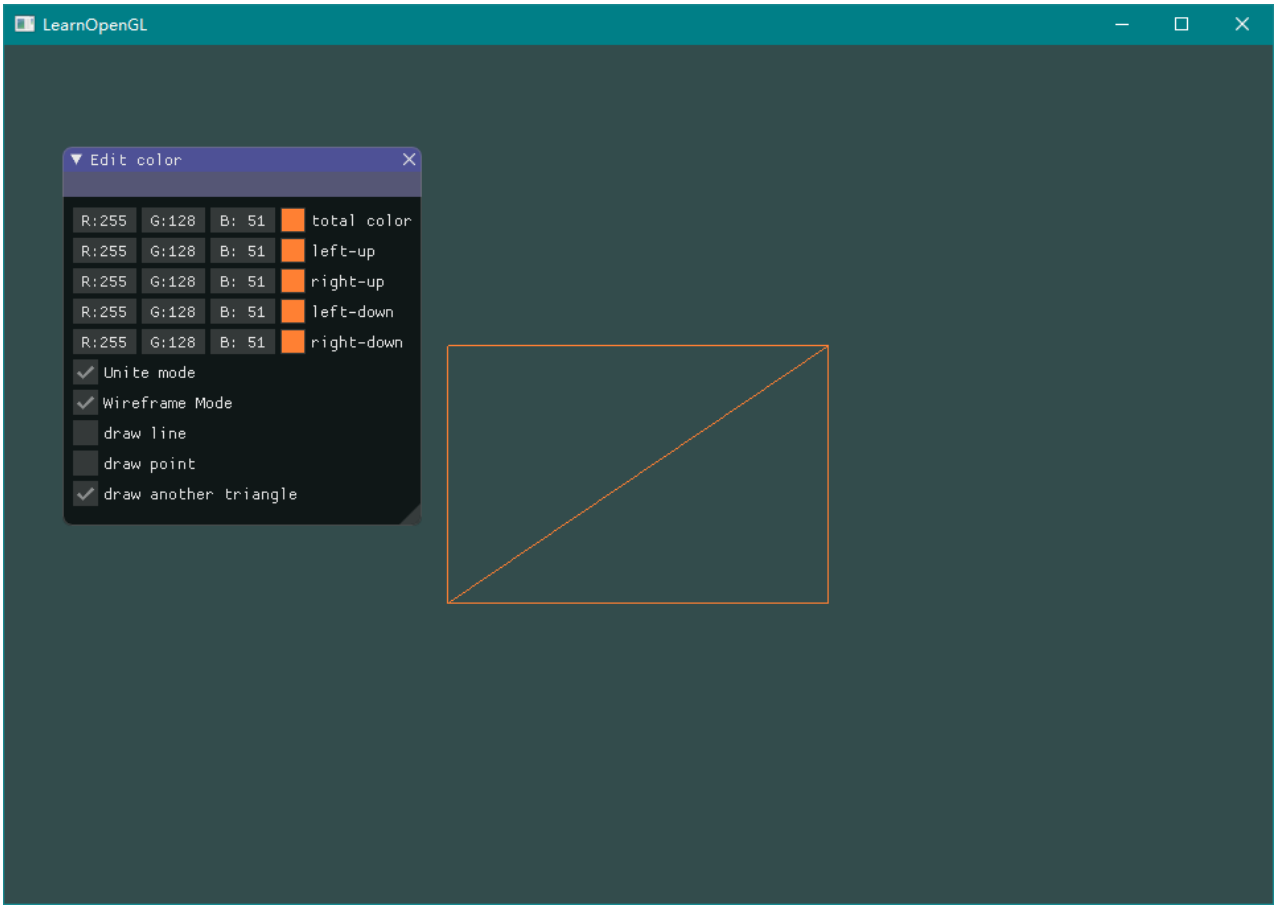
- 顶点数组已经复制到顶点缓冲中，将索引数组复制索引缓冲中即EBO，即可绘制出索引数组中的点

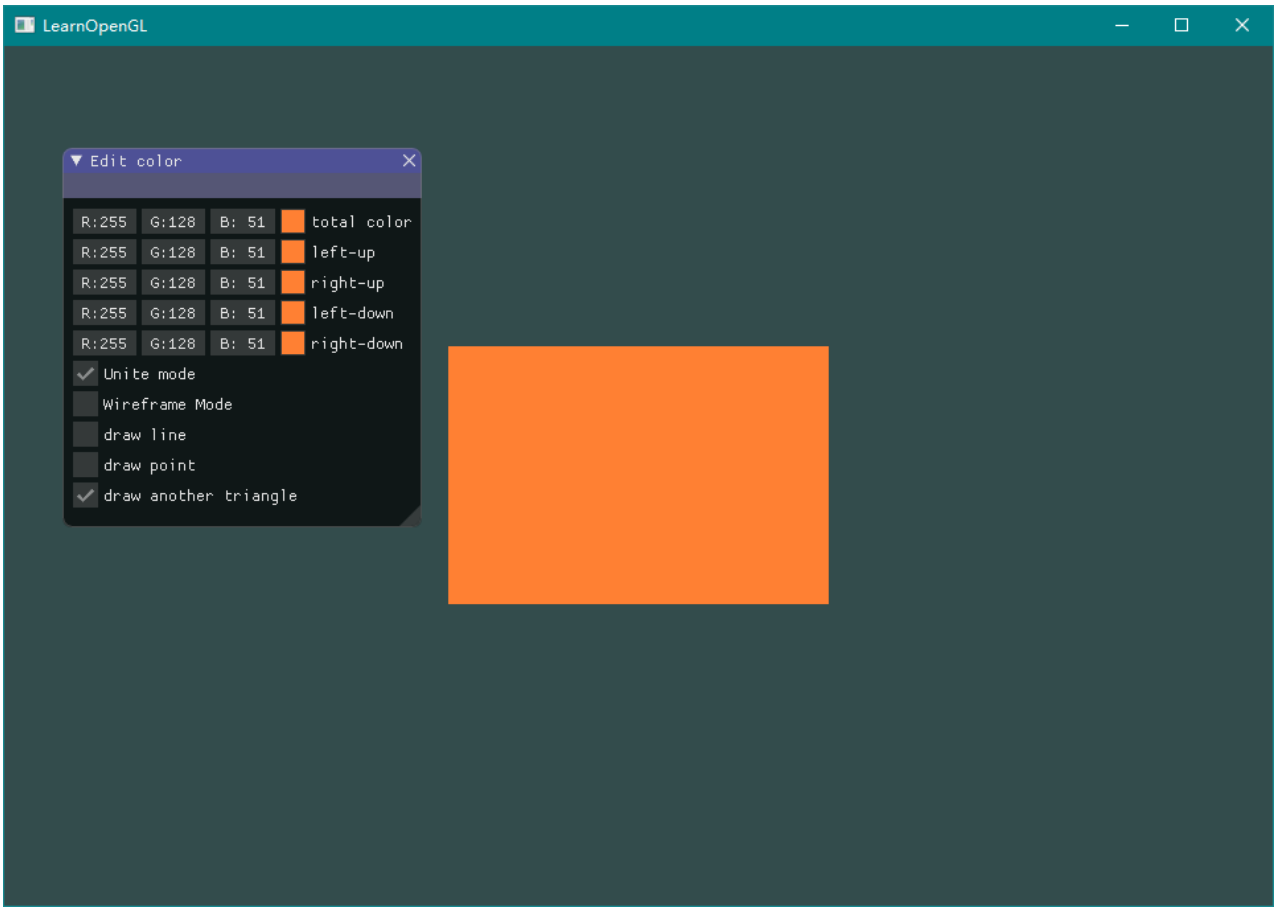
```
// 复制索引数组到一个索引缓冲中，供OpenGL使用
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

- 渲染时使用 glDrawElements

```
// 渲染第二个三角形
if (draw_triangle) {
    glEnableVertexAttribArray(1);
    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);
}
```




LearnOpenGL
Edit color
R:255 G:128 B: 51 total color
R:255 G:128 B: 51 left-up
R:255 G:128 B: 51 right-up
R:255 G:128 B: 51 left-down
R:255 G:128 B: 51 right-down
Unite mode
Wireframe Mode
draw line
draw point
draw another triangle


LearnOpenGL
Edit color
R:255 G:128 B: 51 total color
R:255 G:128 B: 51 left-up
R:255 G:128 B: 51 right-up
R:255 G:128 B: 51 left-down
R:255 G:128 B: 51 right-down
Unite mode
Wireframe Mode
draw line
draw point
draw another triangle