Homework 2

Simple Triangle

Functions

- `framebuffer_size_callback`: 在每次窗口大小被调整的时候被调用，用来同时调整`viewport`
- `processInput`: 检查用户是否按下 `ESC` 键，如果按下就通过`glfwSetwindowShouldClose` 把 `WindowShouldClose` 属性设置为true的方法关闭GLFW。

Algorithm

1. **创建顶点数据**，每个顶点由6个浮点数构成（前三个为位置属性，后三个为颜色属性）。注：此处三个点的颜色是一样的，可以直接在片段着色器直接硬编码颜色，但此处为了方便，直接修改了第二问的代码。

```
// Triangle vertices data
float vertices[] = {
    //   positions            colors
        //-0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,  // left
        // 0.0f,  0.5f, 0.0f,  1.0f, 0.0f, 0.0f,  // up
        // 0.5f, -0.5f, 0.0f,  0.0f, 0.0f, 1.0f   // right
        -0.5f, -0.5f, 0.0f,  1.0f, 0.5f, 0.2f,  // left
         0.0f,  0.5f, 0.0f,  1.0f, 0.5f, 0.2f,  // up
         0.5f, -0.5f, 0.0f,  1.0f, 0.5f, 0.2f  // right
};
```

2. **创建顶点数组对象和顶点缓冲对象**

```
unsigned int VAO, VBO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
```

3. **绑定VAO和VBO，并配置**

```
glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

4. **链接顶点属性**

```
// Postion attribute: location = 0
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// Color attribute: location = 1
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
```

其中：

`glVertexAttribPointer`：

1. 第一个参数指定要配置的顶点属性，步骤1中的顶点数据中有坐标值和颜色值，这里我们约定 `location=0` 为坐标属性，`location=1` 为颜色属性
2. 第二个参数指定顶点属性的大小。顶点属性是一个vec3，所以大小是3
3. 第三个参数指定数据的类型
4. 第四个参数定义我们是否希望数据被标准化，如果为true，所有数据都会被映射到0~1
5. 第五个参数是步长，它告诉我们在连续的顶点属性组之间的间隔，因为我们每6个浮点数为一个顶点数据，所以步长为6
6. 最后一个参数表示位置数据在缓冲中起始位置的偏移量

`glEnableVertexAttribArray`：以顶点属性位置值作为参数，启用顶点属性。

5. **编写、编译并链接着色器程序**
   - **着色器源码：**

```
const char* vertexShaderSource = "#version 330 core\n"
"layout (location = 0) in vec3 aPos;\n"
"layout (location = 1) in vec3 aColor;\n"
"out vec3 ourColor;\n"
"void main() {\n"
"    gl_Position = vec4(aPos, 1.0);\n"
"    ourColor = aColor;\n"
"}\n";

const char* fragmentShaderSource = "#version 330 core\n"
"in vec3 ourColor;\n"
"out vec4 FragColor;\n"
"void main() {\n"
"    FragColor = vec4(ourColor, 1.0f);\n"
"}\n";
```

   其中：

   1. 使用 `GLSL` 编写着色器，第一行需要标注一个版本声明，这是是GLSL 330以及核心模式
   2. 使用 `in` 关键字来声明着色器的输入。在顶点着色器中，有两个输入：坐标和颜色；在片段着色器中，有一个输入：颜色

3. 使用 `out` 关键字来声明着色器的输出。在顶点着色器中，输出想要的颜色，是一个 vec3；在片段着色器中，输出最终想要的颜色，是一个vec4

- **编译着色器：**

```
// build and compile shader program
// ------------------- vertex shader --------------------------
int vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
int success;
char infoLog[512];
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
if (!success) {
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
    std::cout << "ERROR: VERTEX SHADER COMPILATION FAILED!\n" << infoLog << std::endl;
}
// ------------------- fragment shader -------------------------
int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);
glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
if (!success) {
    glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
    std::cout << "ERROR: FRAGMENT SHADER COMPILATION FAILED!\n" << infoLog << std::endl;
}
```

其中：

1. `glCreateShader` 创建一个着色器，参数为着色器类型
2. `glCompileShader` 编译着色器，参数为刚刚创建的着色器

- **链接着色器：**

```
// -------------------- shader program --------------------------
int shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
if (!success) {
    glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
    std::cout << "ERROR: SHADER PROGRAM LINKING FAILED!" << infoLog << std::endl;
}
// delete shaders after linking
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

其中：

1. `glCreateProgram` 创建一个着色器程序对象
2. `glAttachShader` 将着色器附加到着色器程序上
3. `glLinkProgram` 链接着色器
4. 链接完成后，使用 `glDeleteShader` 将不再需要的着色器删除以节约空间

6. **渲染**

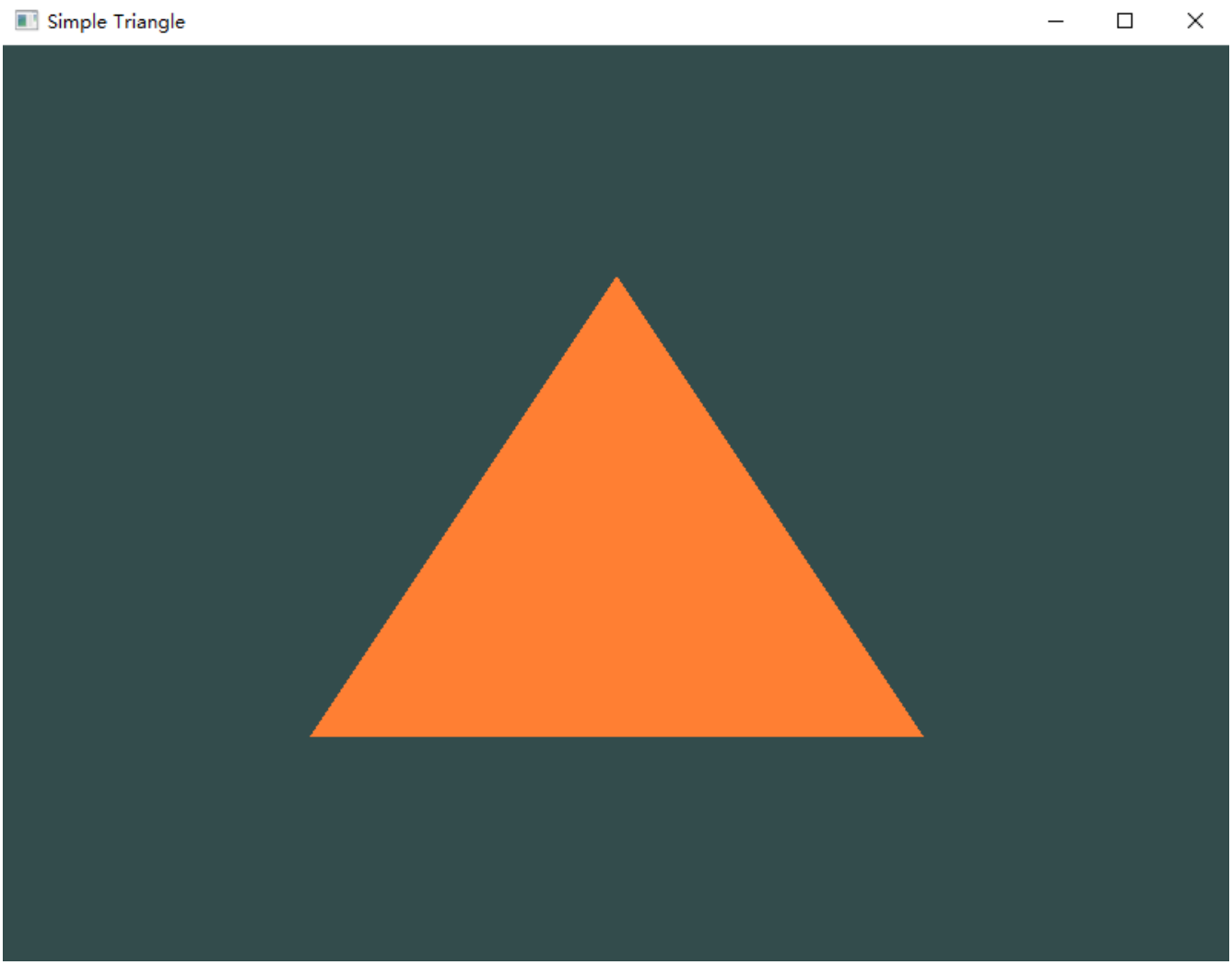
```
// render loop
// -----------
while (!glfwWindowShouldClose(window)) { ... }
```

在 `Simple Triangle` 中，我们只需要在这个循环里处理输入事件以及绑定VAO画出三角形即可。

```
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

```
// glfw: swap buffers and poll IO events (keys pressed/released, mouse moved etc.)
// -------------------------------------------------------------------------------
glfwSwapBuffers(window);
glfwPollEvents();
```

效果



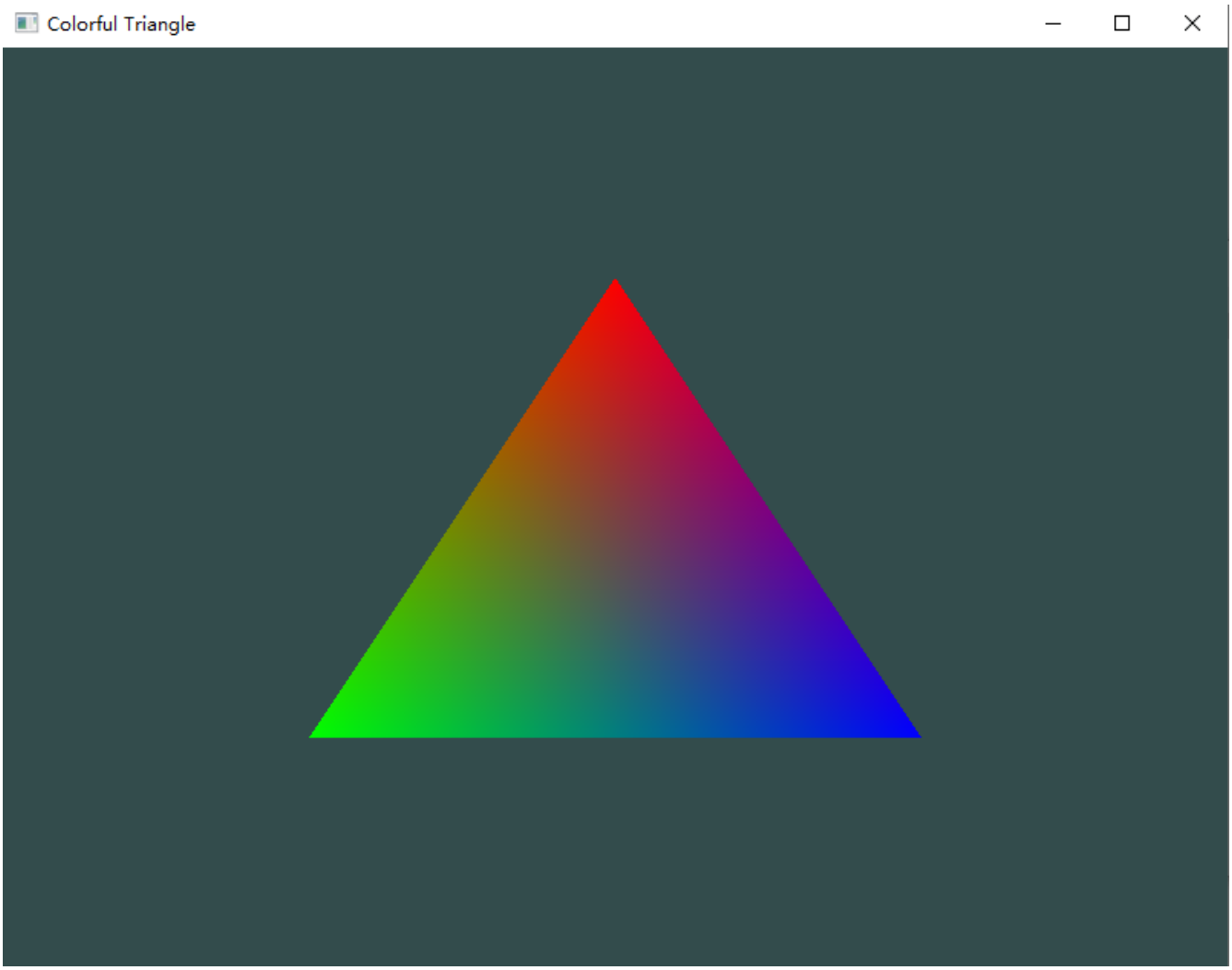
Colorful Triangle

Algorithm

在上一个程序的基础上，只需要修改一下三个顶点的颜色值：

```
// Triangle vertices data
float vertices[] = {
//   positions            colors
    -0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,  // left  : g
     0.0f,  0.5f, 0.0f,  1.0f, 0.0f, 0.0f,  // up    : r
     0.5f, -0.5f, 0.0f,  0.0f, 0.0f, 1.0f   // right : b
};
```

效果



解释

片段着色器会进行片段插值的操作，简单来说，例如有一条线段，一端是红色，一端是蓝色，在靠近红色一端的30%处，该点的颜色值为70%红+30%蓝。这也就解释了在上图中，左边的这条边上，中间部分的颜色呈现的是一种紫色。

ImGui Colorful Triangle

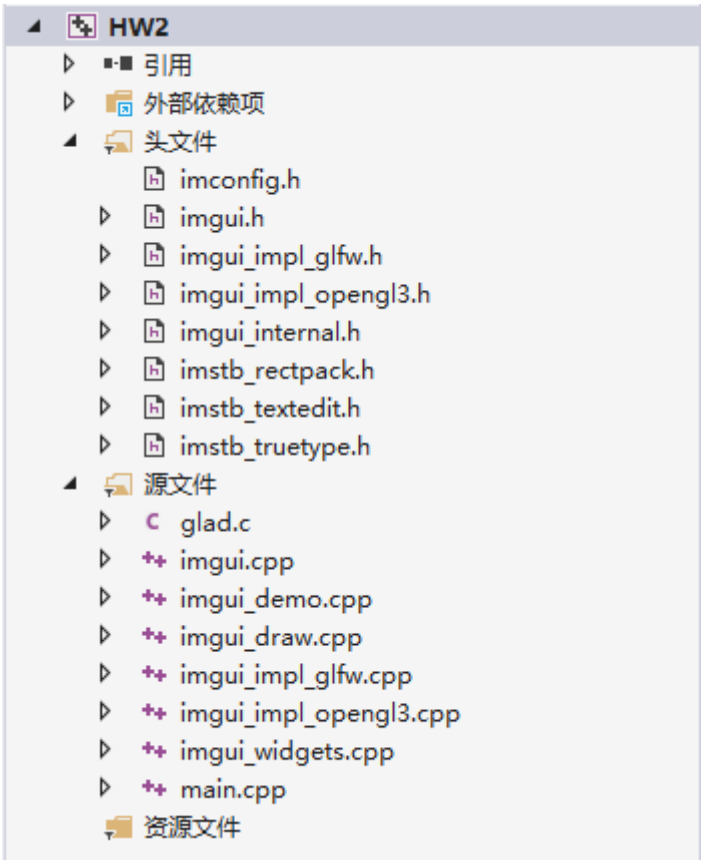
前提工作

> 环境：VS2017 OpenGL3.3 + GLFW

1. 下载 ImGui 源码 并复制到项目文件夹中。注意：目前只能通过复制到项目文件夹后再引入项目才能不会出现找不到 `ImGui` 的错误。
2. 修改 `imgui_impl_opengl3.cpp` 中引入头文件部分语句：

```
91  #if defined(IMGUI_IMPL_OPENGL_LOADER_GL3W)
92  #include <glad/glad.h>
93  #include <GLFW/glfw3.h>
94  //#include <GL/gl3w.h>
```

   将 `#include <GL/gl3w.h>` 注释掉，并用上述两条语句替换。
3. 最终项目结构为：



Algorithm

1. **引入三个头文件**

```
#include "imgui.h"
#include "imgui_impl_glfw.h"
#include "imgui_impl_opengl3.h"
```

2. **设置IMGUI上下文及样式**

```
// Setup ImGui Context
IMGUI_CHECKVERSION();
ImGui::CreateContext();
ImGuiIO& io = ImGui::GetIO();
(void)io;

// Setup ImGui style
ImGui::StyleColorsDark();
```

3. **设置平台和渲染器绑定**

```
// Setup Platform/Renderer bindings
ImGui_ImplGlfw_InitForOpenGL(window, true);
ImGui_ImplOpenGL3_Init(glsl_version);
```

4. **为3值调色板创建初始颜色**

```
std::vector<ImVec4> colors(3);
colors[0] = ImVec4(0.0f, 1.0f, 0.0f, 1.0f);
colors[1] = ImVec4(1.0f, 0.0f, 0.0f, 1.0f);
colors[2] = ImVec4(0.0f, 0.0f, 1.0f, 1.0f);
```

5. **在渲染循环中渲染 ImGui 窗口**

```
ImGui_ImplOpenGL3_NewFrame();
ImGui_ImplGlfw_NewFrame();
ImGui::NewFrame();

{
    ImGui::Begin("COLOR CHANGER");
    ImGui::SetWindowFontScale(1.4);
    static float f = 0.f;
    static int counter = 0;
    ImGui::Text("Change the colors!");

    ImGui::ColorEdit3("LEFT", (float*)&colors[0]);
    ImGui::ColorEdit3("TOP", (float*)&colors[1]);
    ImGui::ColorEdit3("RIGHT", (float*)&colors[2]);

    for (int i = 0; i < 3; ++i) {
        vertices[i * 6 + 3] = colors[i].x;
        vertices[i * 6 + 4] = colors[i].y;
        vertices[i * 6 + 5] = colors[i].z;
    }

    ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f / ImGui::GetIO().Framerate, ImGui::GetIO().Framerate);
    ImGui::End();
}
```

其中：

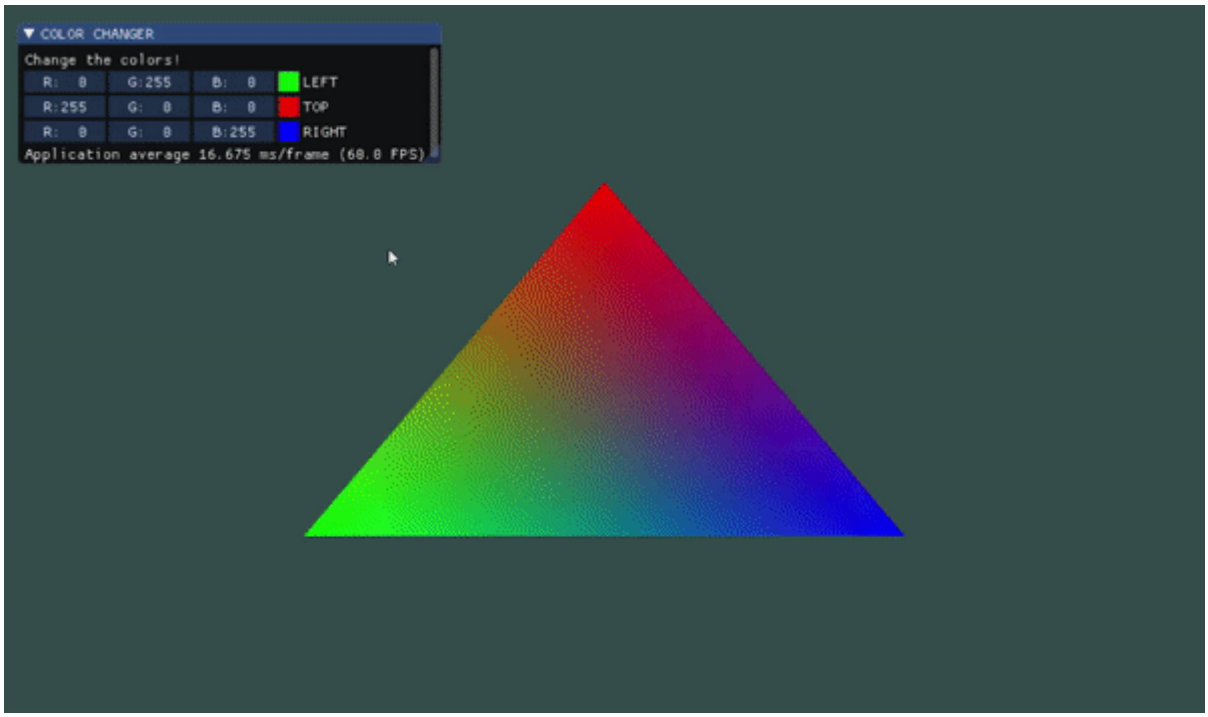
- 将 `colors` 这个数组绑定到 `ColorEdit3` 这个组件上，这样我们在调色时也会改变数组对应的值；
- 替换顶点数组中各个顶点的颜色值

```
// render
glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);

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

其中：

- 需要使用 `glClear` 来清空颜色缓冲，不然在拖动 `ImGui` 窗口时会留下痕迹
- 需要重新给 `VBO` 传送顶点数据，因为我们改变了 `vertices` 的数据

效果



使用EBO

Algorithm

1. **创建4个顶点**

```
// Triangle vertices data
float vertices[] = {
//   positions           colors
    -0.75f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,  // g
    -0.25f,  0.5f, 0.0f,  1.0f, 0.0f, 0.0f,  // r
     0.25f, -0.5f, 0.0f,  0.0f, 0.0f, 1.0f,  // b
     0.75f,  0.5f, 0.0f,  0.0f, 1.0f, 0.0f   // g
};
```

2. **创建索引数组**

```
unsigned int indices[] = {
    0, 1, 2, // first triangle
    1, 2, 3  // second triangle
};
```

3. **创建VAO、VBO和EBO**

```
unsigned int VAO, VBO, EBO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glGenBuffers(1, &EBO);
```

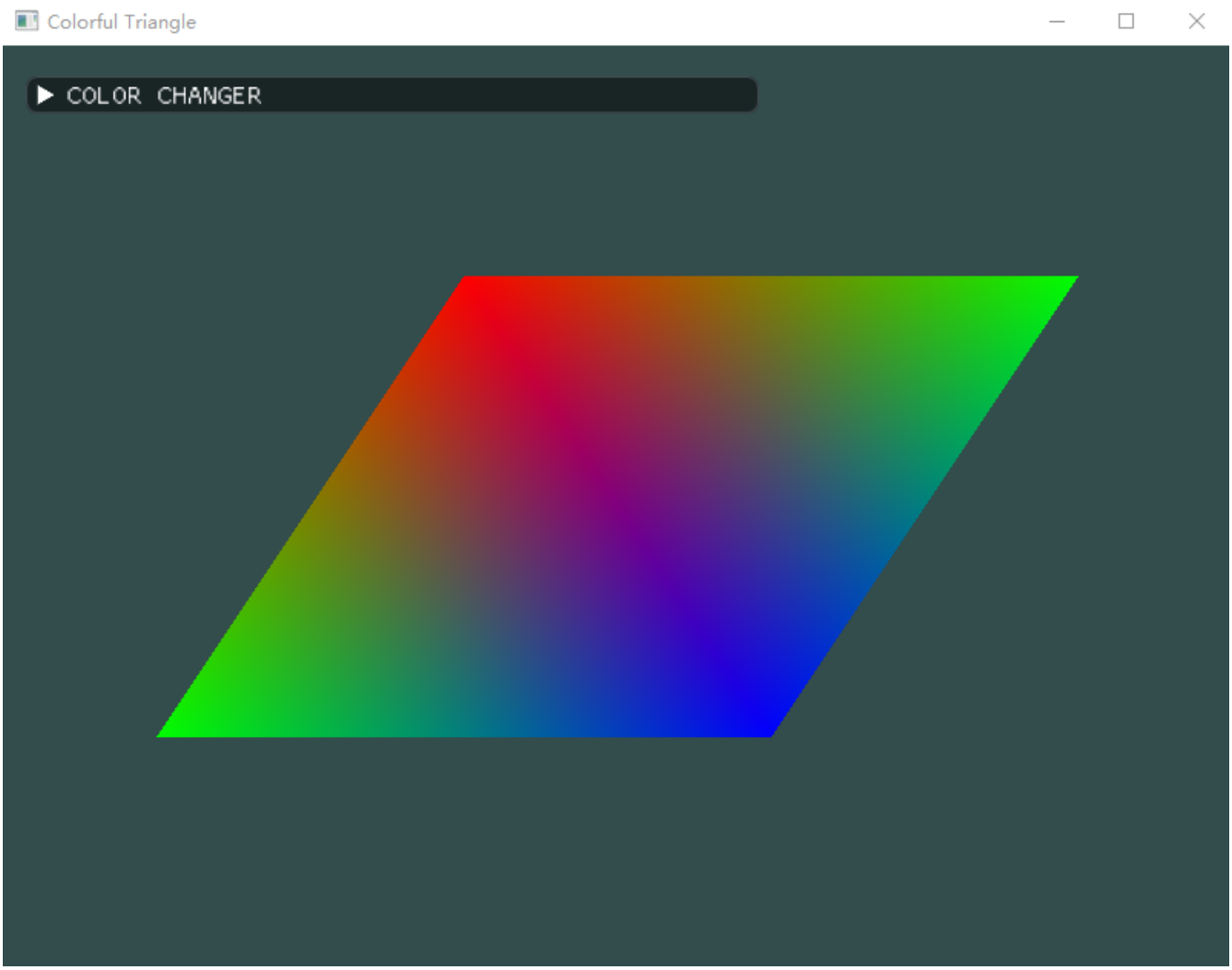
4. **配置EBO**

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

5. **使用 glDrawElements 绘制6个顶点即2个三角形**

```
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

效果



绘制多种图形

Algorithm

1. **创建多个VAO、VBO、EBO**

```
unsigned int VAOs[2], VBOs[2], EBOs[2];
glGenVertexArrays(2, VAOs);
glGenBuffers(2, VBOs);
glGenBuffers(2, EBOs);
```

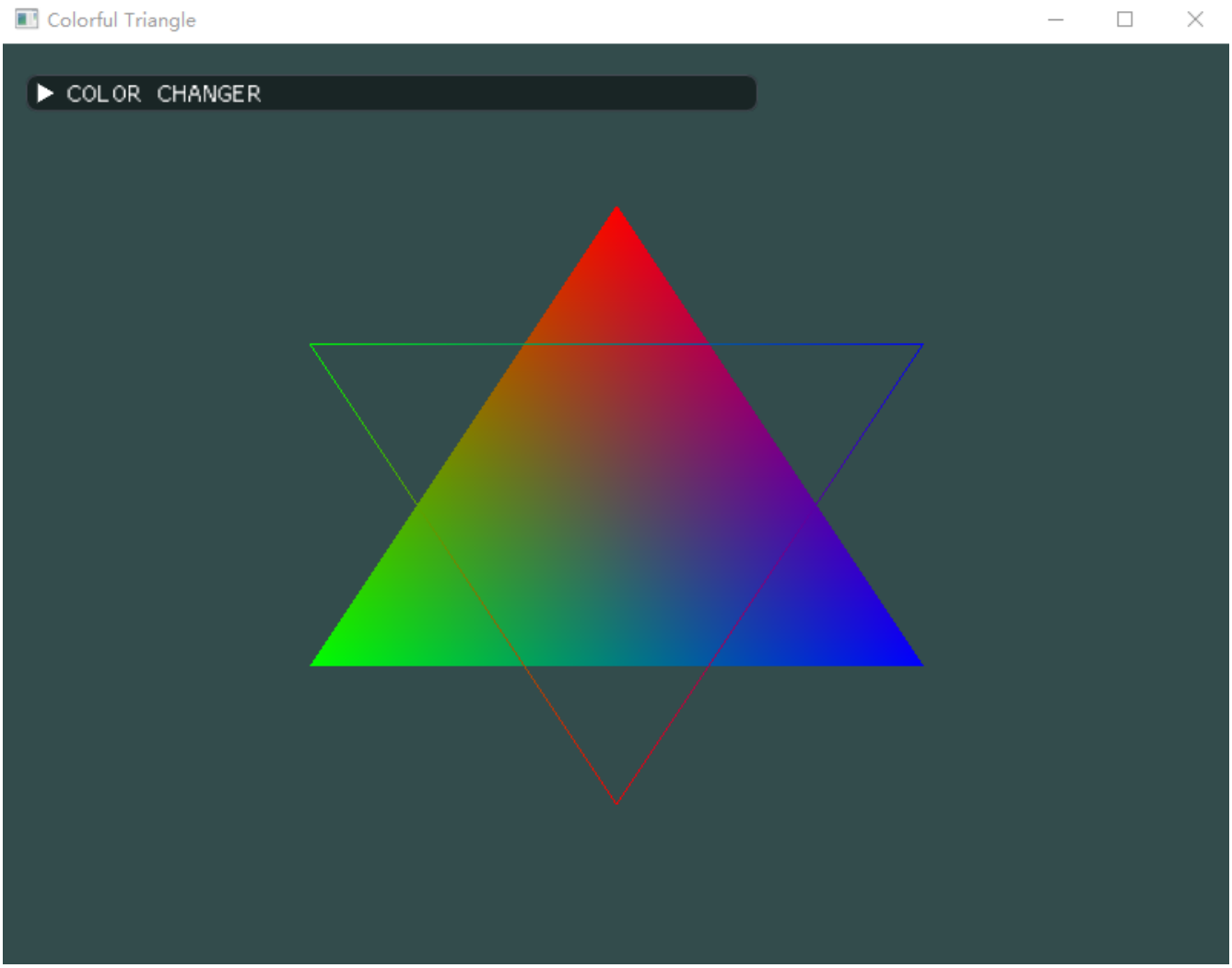
2. **创建顶点数据和索引数据**

```
// Triangle vertices data
float triangle_vertices[] = {
//    positions           colors
    -0.5f, -0.35f, 0.0f,  0.0f, 1.0f, 0.0f,  // g
    -0.0f,  0.65f, 0.0f,  1.0f, 0.0f, 0.0f,  // r
     0.5f, -0.35f, 0.0f,  0.0f, 0.0f, 1.0f   // b
};

// line vertices data
float line_vertices[] = {
    -0.5f,  0.35f, 0.0f,  0.0f, 1.0f, 0.0f,  // g
     0.0f, -0.65f, 0.0f,  1.0f, 0.0f, 0.0f,  // r
     0.5f,  0.35f, 0.0f,  0.0f, 0.0f, 1.0f   // b
};

unsigned int triangle_indices[] = {
    0, 1, 2 // first triangle
};

unsigned int line_indices[] = {
    0, 1,   // 1st line
    1, 2,   // 2nd line
    2, 0    // 3rd line
};
```

3. **第一个VAO存储三角形顶点数据，第二个VAO存储线段数据**
4. **渲染循环中，先后绑定两个VAO进行渲染**

```
glBindVertexArray(VAOs[0]);
glBindBuffer(GL_ARRAY_BUFFER, VBOs[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(triangle_vertices), triangle_vertices, GL_STATIC_DRAW);
//glDrawArrays(GL_TRIANGLES, 0, 3);
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, 0);

glBindVertexArray(VAOs[1]);
glBindBuffer(GL_ARRAY_BUFFER, VBOs[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(line_vertices), line_vertices, GL_STATIC_DRAW);
//glDrawArrays(GL_TRIANGLES, 0, 3);
glDrawElements(GL_LINES, 6, GL_UNSIGNED_INT, 0);
```

效果



注：这是三条线段围成的三角形，而不是设置了 `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);` 。