作业 2

Basic:

1. 使用 OpenGL(3.3 及以上)+GLFW 或 freeglut 画一个简单的三角形。

关键代码解释：

- **用着色器语言 GLSL(OpenGL Shading Language)编写片段着色器**

  片段着色器(Fragment Shader)是第二个也是最后一个我们打算创建的用于渲染三角形的着色器。片段着色器所做的是计算像素最后的颜色输出。

```
//片段着色器源代码
const char *fragmentShaderSource = "#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"    FragColor = vec4(0.41f, 0.84f, 0.94f, 1.0f);\n"
"}\n\0";
```

- **编译顶点着色器并检测是否成功**

  为了能够让 OpenGL 使用编写好的顶点着色器，我们必须在运行时动态编译它的源码

```
//编译着色器
unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
//检测是否编译成功
int success;
char infoLog[512];
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
if (!success)
{
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
}
```

- **链接着色器并检测是否成功**

  着色器程序对象(Shader Program Object)是多个着色器合并之后并最终链接完成的版本。如果要使用刚才编译的着色器我们必须把它们链接(Link)为一个着色器程序对象，然后在渲染对象的时候激活这个着色器程序。已激活着色器程序的着色器将在我们发送渲染调用的时候被使用。

```
//链接着色器
unsigned int shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
//检测链接是否成功
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
if (!success) {
    glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
}
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

- **链接顶点属性**

顶点着色器允许我们指定任何以顶点属性为形式的输入。这使其具有很强的灵活性的同时，它还的确意味着我们必须手动指定输入数据的哪一个部分对应顶点着色器的哪一个顶点属性。所以，我们必须在渲染前指定 OpenGL 该如何解释顶点数据。

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

glBindVertexArray(VAO);
//把新创建的缓冲绑定到GL_ARRAY_BUFFER
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

//链接顶点属性
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindVertexArray(0);
```

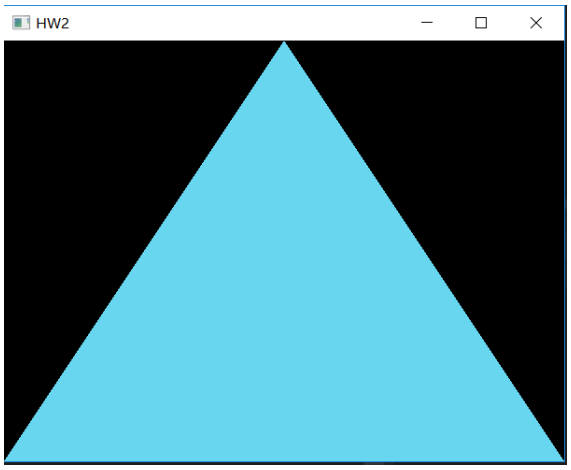
- **画三角形的实现**

OpenGL 给我们提供了 glDrawArrays 函数，它使用当前激活的着色器，之前定义的顶点属性配置，和 VBO 的顶点数据（通过 VAO 间接绑定）来绘制图元。glDrawArrays 函数第一个参数是我们打算绘制的 OpenGL 图元的类型。由于我们在一开始时说过，我们希望绘制的是一个三角形，这里传递 GL_TRIANGLES 给它。

```
//画三角形
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

具体代码见 basic-1.cpp

三角形如下，颜色数据为 **vec4(0.41f, 0.84f, 0.94f, 1.0f)**



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。

顶点改为红绿蓝后，出现结果如下图。原因如下：虽然我们只在提供了三个顶点的颜色属性，这是由于在片段着色器中进行片段插值的结果。当渲染一个三角形时，光栅化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，它会插值所有片段着色器的输入变量。而在这里有 3 个顶点，和相应的 3 个颜色，片段着色器为三角形的这些像素进行插值颜色。就会出现如下图所见的情况，例如在每一条边颜色渐变的。

关键代码：

- **顶点数据**

  把颜色数据加进顶点数据中。我们将把颜色数据添加为 3 个 float 值至 vertices 数组。我们将把三角形的三个角分别指定为红色、绿色和蓝色

```
//顶点输入
float vertices[] = {
    // 位置              // 颜色
    1.0f, -1.0f, 0.0f,   0.0f, 0.0f, 1.0f,    // 右下 蓝
    -1.0f, -1.0f, 0.0f,   0.0f, 1.0f, 0.0f,    // 左下 绿
    0.0f,  1.0f, 0.0f,   1.0f, 0.0f, 0.0f     // 顶部 红
};
```

- **顶点着色器**

  调整一下顶点着色器，使它能够接收颜色值作为一个顶点属性输入。需要注意的是我们用 layout 标识符来把 aColor 属性的位置值设置为 1

```
//顶点着色器源代码
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"
"{\n"
"    gl_Position = vec4(aPos, 1.0);\n"
"    ourColor = aColor;\n"
"}\0";
```

- **链接顶点属性**

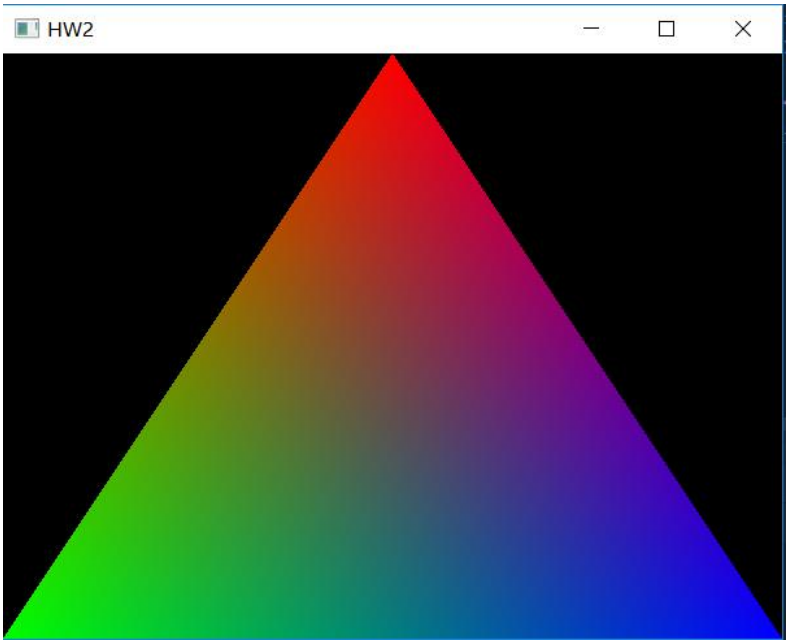
由于我们现在有了两个顶点属性，我们不得不重新计算步长值。为获得数据队列中下一个属性值（比如位置向量的下个 x 分量）我们必须向右移动 6 个 float，其中 3 个是位置值，另外 3 个是颜色值。这使我们的步长值为 6 乘以 float 的字节数（=24 字节）。

同样，这次我们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。

```
//链接顶点属性
// 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);

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

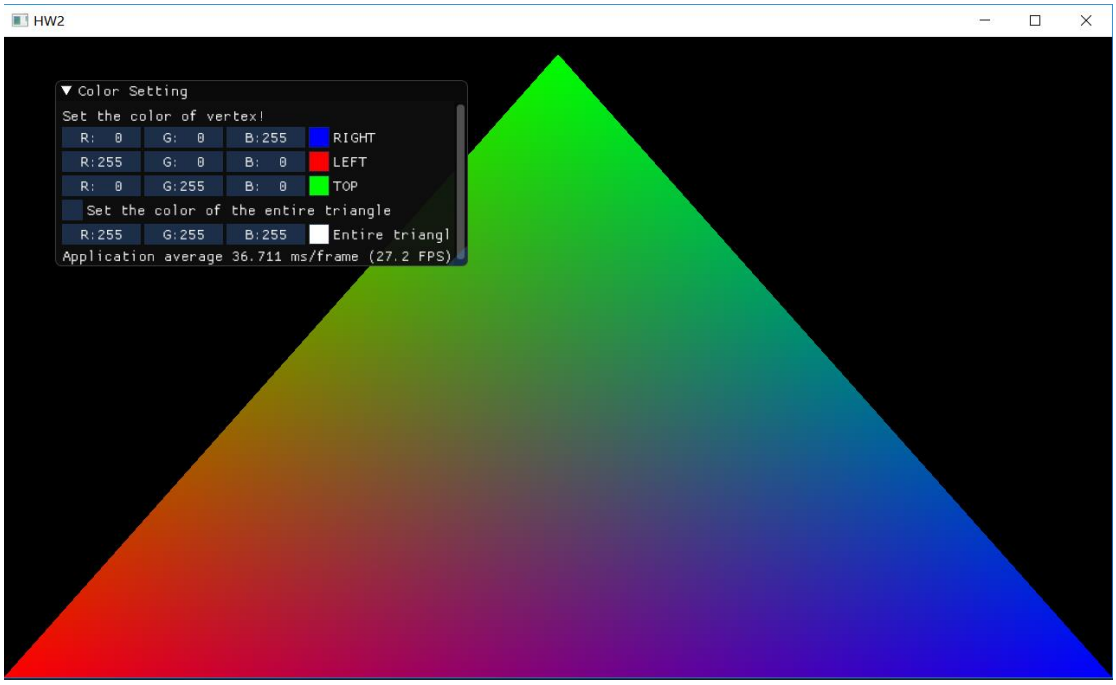
具体代码见 basic-2.cpp



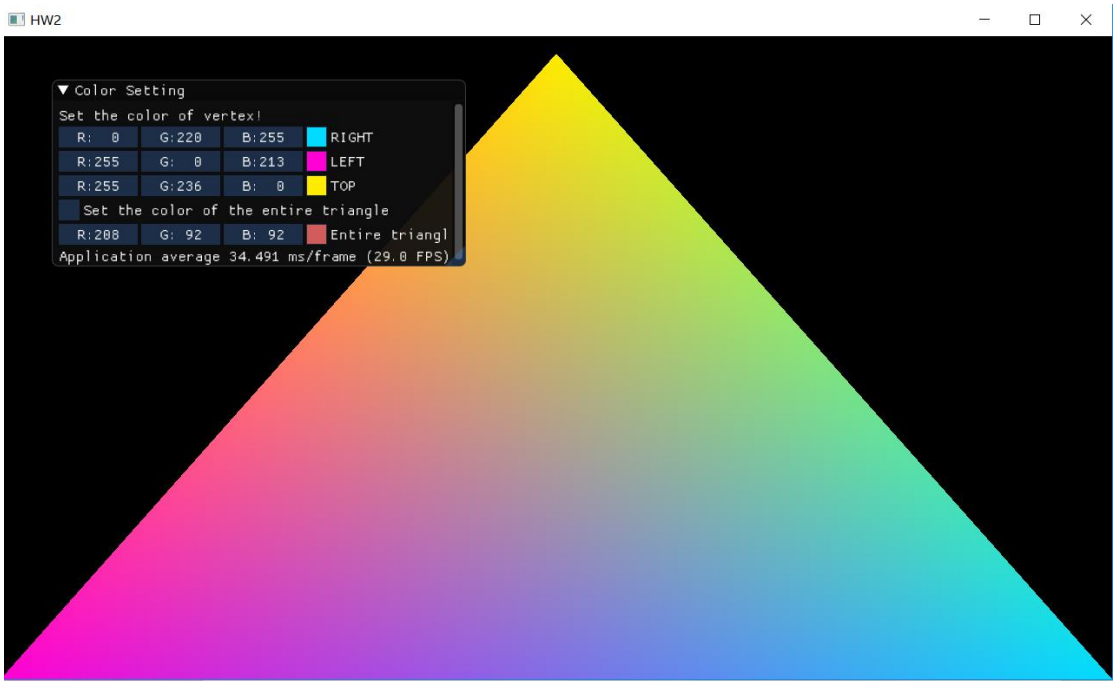
3. 给上述工作添加一个 GUI，里面有一个菜单栏，使得可以选择并改变三角形的颜色

由下图所示，添加的 GUI 为 IMGUI。设计成为可以单独调整各顶点颜色和设置整个三角形颜色的两种模式，可以通过选择 **Set the color of the entire triangle** 来改变

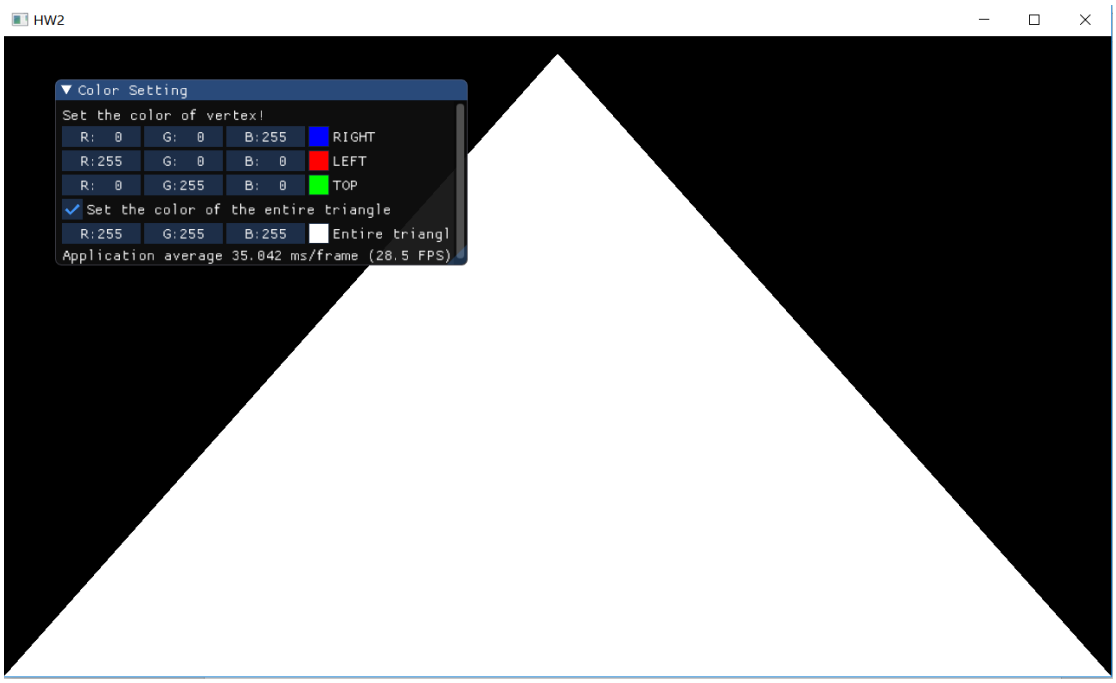
初始状态：

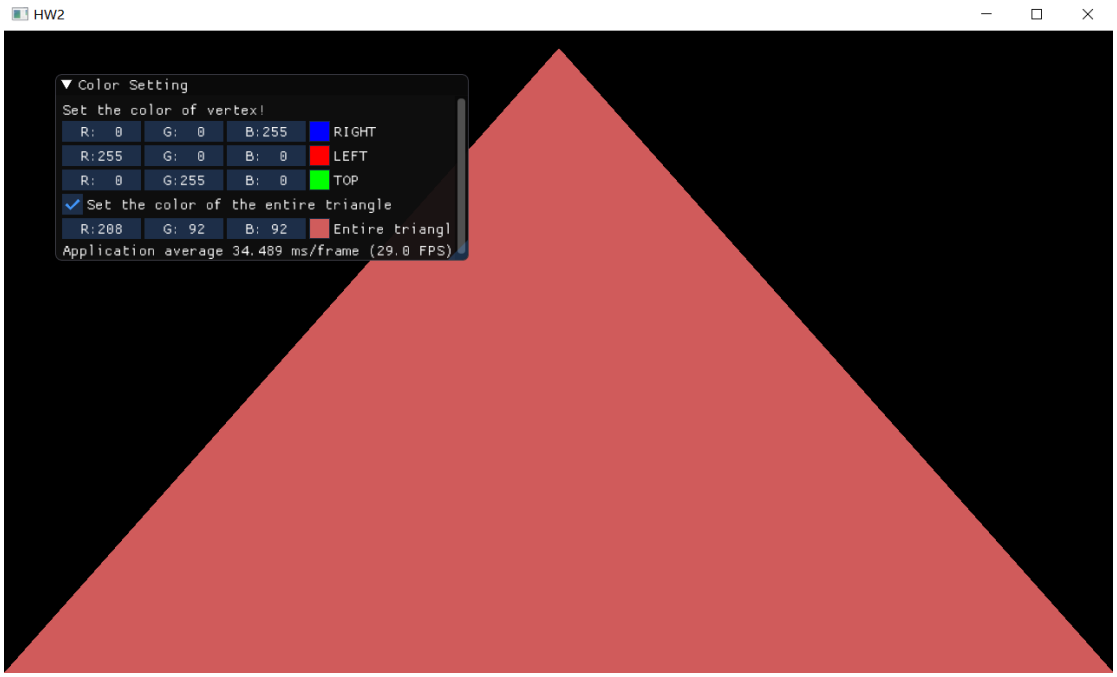


改变顶点颜色：



选择 Set the color of the entire triangle 后的三角形颜色先变为白色，再随便通过 GUI 修改为另一个颜色





关键代码：

- **imgui_impl_opengl3.h 及头文件**

在这里我们是 opengl3+GLAD，所以我们要修改 **imgui_impl_opengl3.h** 中的一处内容。将原本的 #define IMGUI_IMPL_OPENGL_LOADER_GL3W 修改为 #define IMGUI_IMPL_OPENGL_LOADER_GLAD。并需要添加如第二个图所示的三个头文件

```
// Set default OpenGL3 loader to be gl3w
#if !defined(IMGUI_IMPL_OPENGL_LOADER_GL3W)     \
 && !defined(IMGUI_IMPL_OPENGL_LOADER_GLEW)     \
 && !defined(IMGUI_IMPL_OPENGL_LOADER_GLAD)     \
 && !defined(IMGUI_IMPL_OPENGL_LOADER_CUSTOM)
#define IMGUI_IMPL_OPENGL_LOADER_GLAD
#endif
```

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

- **IMGUI 初始化**
  初始化 IMGUI

```
const char* glsl_version = "#version 330 core";
// Setup Dear ImGui context
IMGUI_CHECKVERSION();
ImGui::CreateContext();
ImGuiIO& io = ImGui::GetIO(); (void)io;
// Setup Dear ImGui style
ImGui::StyleColorsDark();

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

- **设置 IMGUI 界面**
  设置 UI 的标题和功能。

```
ImGui::Begin("Color Setting");
ImGui::SetWindowFontScale((float)1.4);
static float f = 0.0f;
static int counter = 0;
ImGui::Text("Set the color of vertex!");

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

ImGui::Checkbox("Set the color of the entire triangle", &isWhole);
ImGui::ColorEdit3("Entire triangle", (float*)&entireColor, 1);
```

- **通过 IMGUI 修改三角形颜色**
  根据用户的选择来修改顶点数组中的值改变顶点或者整个三角形的颜色，

```
//修改三角形颜色
if (isWhole) {
    for (int i = 0; i < 3; i++) {
        vertices[i * 6 + 3] = entireColor.x;
        vertices[i * 6 + 4] = entireColor.y;
        vertices[i * 6 + 5] = entireColor.z;
    }
}
else {
    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;
    }
}
```

Bonus：

以下两题代码在 bonus.cpp 中。

1. 绘制其他的图元，除了三角形，还有点、线等。

绘制点和线的结果如下图一，可以看出三角形左侧有五个点，右侧有一条线。实现方法同三角形类似，在数组中加入各个点，并使用 glDrawArrays 绘制。如下图二、三。

```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);
glDrawArrays(GL_LINES, 3, 2);
glDrawArrays(GL_POINTS, 5, 5);
```

```
//顶点输入
float vertices[] = {
    // 位置              // 颜色
    1.0f, -1.0f, 0.0f,   0.0f, 0.0f, 1.0f,    // 右下 蓝
    -1.0f, -1.0f, 0.0f,   0.0f, 1.0f, 0.0f,    // 左下 绿
    0.0f,  1.0f, 0.0f,   1.0f, 0.0f, 0.0f ,    // 顶部 红

    //线
    0.5f, 1.0f, 0.0f, 1.0f, 1.0f, 1.0f,
    0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f,

    //点
    -0.5f, 0.9f, 0.0f, 1.0f, 1.0f, 1.0f,
    -0.5f, 0.7f, 0.0f, 1.0f, 1.0f, 1.0f,
    -0.5f, 0.5f, 0.0f, 1.0f, 1.0f, 1.0f,
    -0.5f, 0.3f, 0.0f, 1.0f, 1.0f, 1.0f,
    -0.5f, 0.1f, 0.0f, 1.0f, 1.0f, 1.0f,
};
```

2. 使用 EBO(Element Buffer Object)绘制多个三角形。

使用 EBO 绘制两个三角形:

a) 在顶点中加入两个点。

```
// 位置              // 颜色
1.0f, -1.0f, 0.0f,   0.0f, 0.0f, 1.0f,
-1.0f, -1.0f, 0.0f,   0.0f, 1.0f, 0.0f,
0.0f,  1.0f, 0.0f,   1.0f, 0.0f, 0.0f ,

1.0f, 1.0f, 0.0f,   0.0f, 1.0f, 0.0f,
0.5f,  1.0f, 0.0f,   1.0f, 0.0f, 0.0f ,
```

b) 定义三角形由哪几个顶点组成并储存

```
unsigned int indices[] = { // 注意索引从0开始!
    0, 1, 2, // 第一个三角形
    0, 3, 4  // 第二个三角形
};
```

c) 与 VBO 类似，我们先绑定 EBO 然后用 glBufferData 把索引复制到缓冲里。

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

d) 用 glDrawElements 来替换 glDrawArrays 函数

```
//画三角形
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
glDrawArrays(GL_LINES, 5, 2);
glDrawArrays(GL_POINTS, 7, 5);
```