

GUI and Draw simple graphics

王培钰 16340220 电子政务

基础知识

学习来源

- learnOpenGL

重要概念

顶点数组对象(VAO)

顶点缓冲对象(VBO)

索引缓冲对象(EBO)

图形渲染管线

3D坐标转为2D像素的处理过程是由OpenGL的**图形渲染管线**管理的，它分为两部分工作。

- 第一部分把3D坐标转为适应屏幕的2D坐标。
- 第二部分把2D坐标转变为实际有颜色的像素。

顶点输入

标准化设备坐标(NDC)，将输入的顶点定义在-1.0~1.0之间，接着通过**视口变换**将标准化设备坐标变为**屏幕空间坐标**。

顶点着色器会在GPU上创建内存存储这些顶点数据并配置OpenGL如何解释这些内存并指定其如何发送给显卡。通过**顶点缓冲对象(VBO)**来管理这个内存，它会在显存中存储大量顶点。使用VBO的好处是能一次性的发送大批数据到显卡上。

着色器(Shader)

着色器是使用一种为GLSL的类c语言写成的，GLSL是为图形计算量身定制的。

- 顶点着色器：把一个单独的顶点作为输入。主要目的是把3D坐标转为另一种3D坐标，同时允许我们对顶点属性进行一些基本操作。
- 图元装配：将顶点着色器输出的所有顶点作为输入(GL_POINTS、GL_TRIANGLES、GL_LINE_STRIP等来指定数据表示的渲染类型，这称为图元)，并将所有的点装配成指定图元的形状。

- 几何着色器：将图元形式的一些列顶点的集合作为输入，它通过产生新顶点构造出新的（或是其他的）图元来生成其他形状。
- 光栅化阶段：把图元映射为最终屏幕上相应的像素，生成供片段着色器使用的片段。同时在片段着色器运行前会执行裁切。裁切会丢弃超出视图外的所有像素来提升执行效率。
- 片段着色器：计算一个像素的最终颜色，通常包含3D场景的数据（比如光照，阴影，光的颜色等等），这些数据可以被用来计算最终像素的颜色。
- Alpha测试和混合阶段：检测片段对应的深度(和模板)值，用他们来判断这个像素是其他物体的前面还是后面，决定是否应该丢弃。同时检测alpha值(这定义了一个物体的透明度)，并对物体进行混合。

链接顶点属性

设置顶点属性指针将其绑定到顶点数组对象(**VAO**)上，VAO上会存储以下内容：

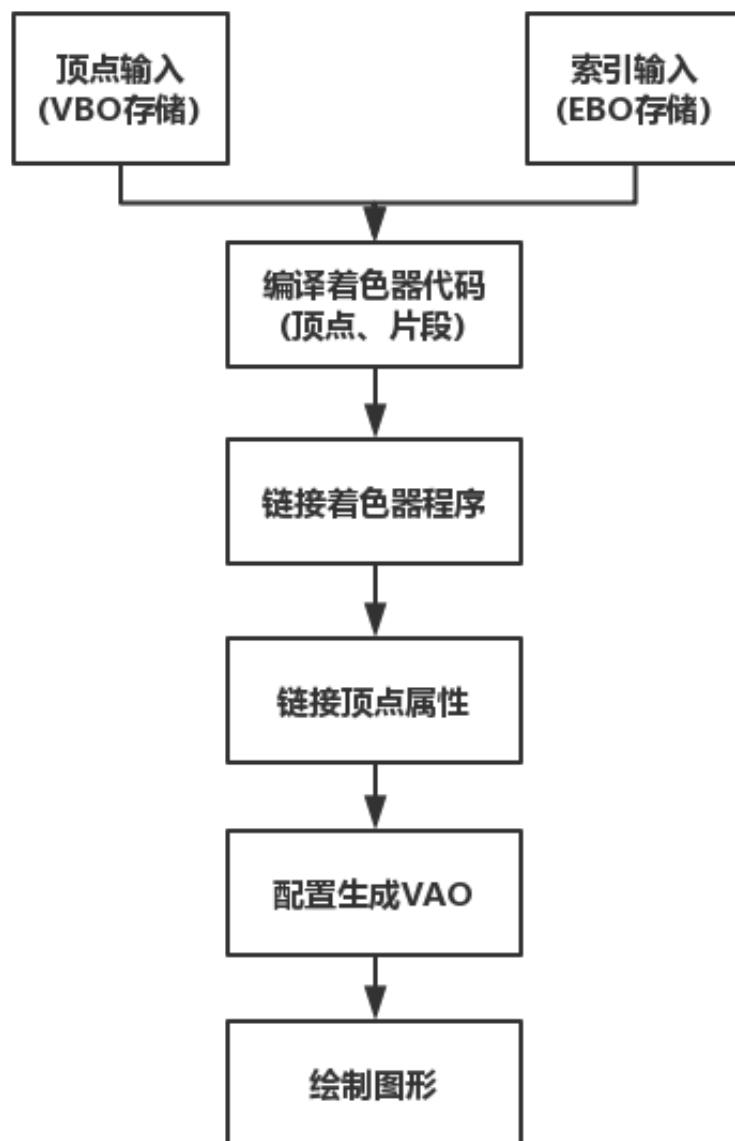
- glEnableVertexAttribArray和glDisableVertexAttribArray的调用。
- 通过glVertexAttribPointer设置的顶点属性配置。
- 通过glVertexAttribPointer调用与顶点属性关联的顶点缓冲对象。

之后再打算绘制一个物体的时候，我们只要在绘制物体前简单地把VAO绑定到希望使用的设定上就行了。

索引缓存对象(**EBO**)

索引绘制：通过设置顶点，以索引的方式来绘制图形，这就避免了重复存储顶点。

总结以上流程理解并绘制如下：



HomeWork

环境配置

操作系统：

- Mac OS X

IDE：

- Xcode

引用库：

- glfw3+glad+ImGui

Basic:

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

顶点输入:

```
float vertices[] = {
    -1.0f, -1.0f, 0.0f, // 左
    1.0f, -1.0f, 0.0f, // 右
    0.0f, 1.0f, 0.0f // 上
};
```

编写着色器代码

```
// 顶点着色器
const char *vertexShaderSource = "#version 330 core\n"
"layout (location = 0) in vec3 aPos;\n"
"out vec4 vertexColor;\n"
"void main()\n"
"{\n"
"    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
"    vertexColor = vec4(1.0f, 1.0f, 0.8f, 0.8f);\n"
"}\0";

// 片段着色器
const char *fragmentShaderSource = "#version 330 core\n"
"out vec4 FragColor;\n"
"in vec4 vertexColor;\n"
"void main()\n"
"{\n"
"    FragColor = vertexColor;\n"
"}\n\0";
```

编译着色器代码

```
// 编译顶点着色器
int vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
// 错误检查
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;
}

// 编译片段着色器
int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);
// 错误检查
glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
if (!success)
{
    glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" <<
infoLog << std::endl;
}
```

链接着色器

```
// 创建一个程序对象
int shaderProgram = glCreateProgram();
// 将编译好的着色器附着到程序上
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
// 链接错误检查
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
if (!success) {
    glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog
<< std::endl;
}
// 链接完成，删除着色器对象
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

配置生成VAO

```
unsigned int VBO, VAO;
 glGenVertexArrays(1, &VAO);
 glGenBuffers(1, &VBO);
 // bind the Vertex Array Object first, then bind and set vertex
 buffer(s), and then configure vertex attributes(s).
 glBindVertexArray(VAO);

 // 复制顶点数组到缓冲中供openGL使用
 glBindBuffer(GL_ARRAY_BUFFER, VBO);
 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
 GL_STATIC_DRAW);

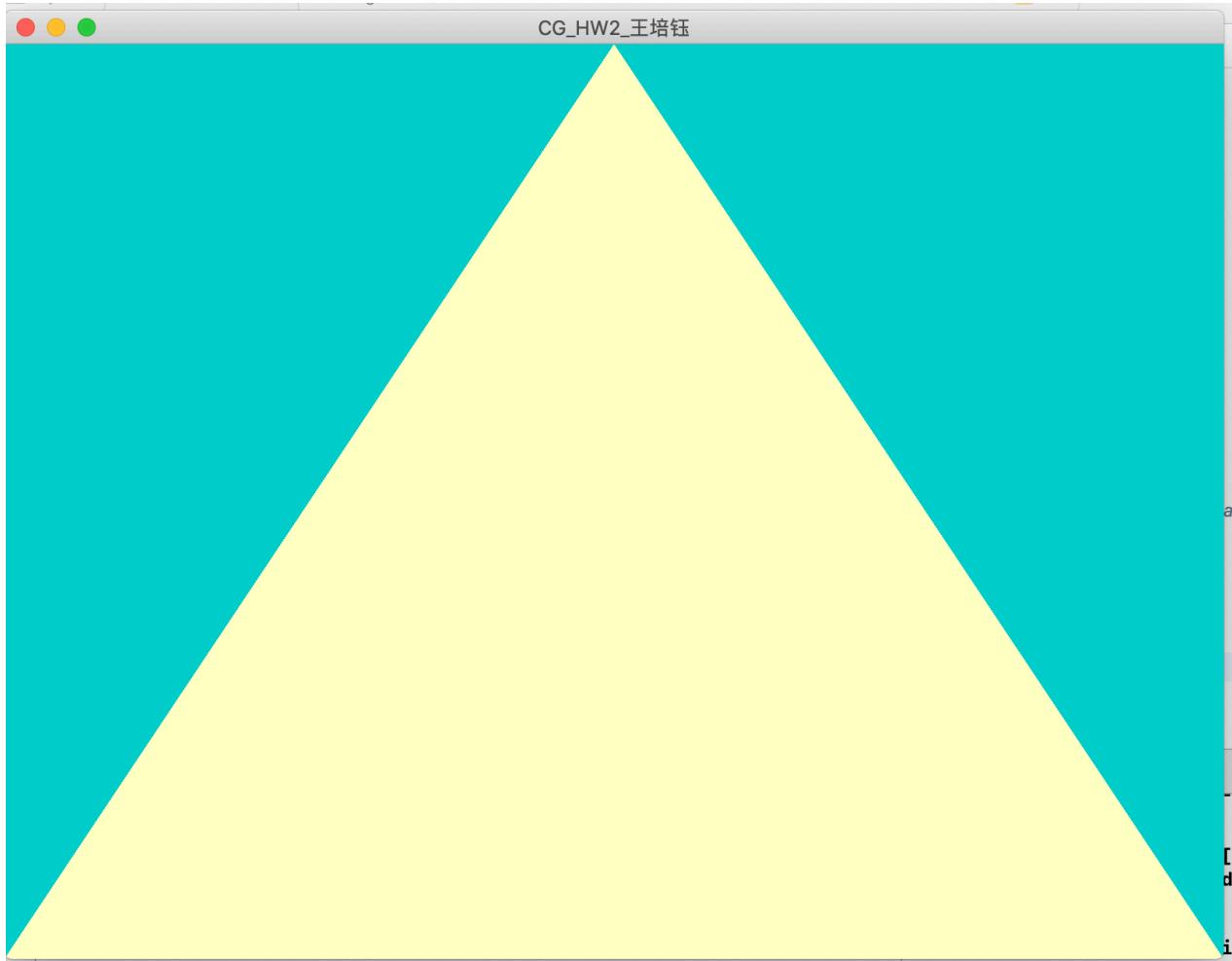
 // 设置顶点属性指针，告知openGL如何解析这些顶点数据
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
 (void*)0);
 glEnableVertexAttribArray(0);
```

绘制三角形：

```
//渲染
glClearColor(0.0f, 0.8f, 0.8f, 1.0f); //设置清空屏幕所用的颜色
glClear(GL_COLOR_BUFFER_BIT); //使用当前状态来获取应该清除为的颜色

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

生成效果：



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。

- 比起第一问主要有以下更改：

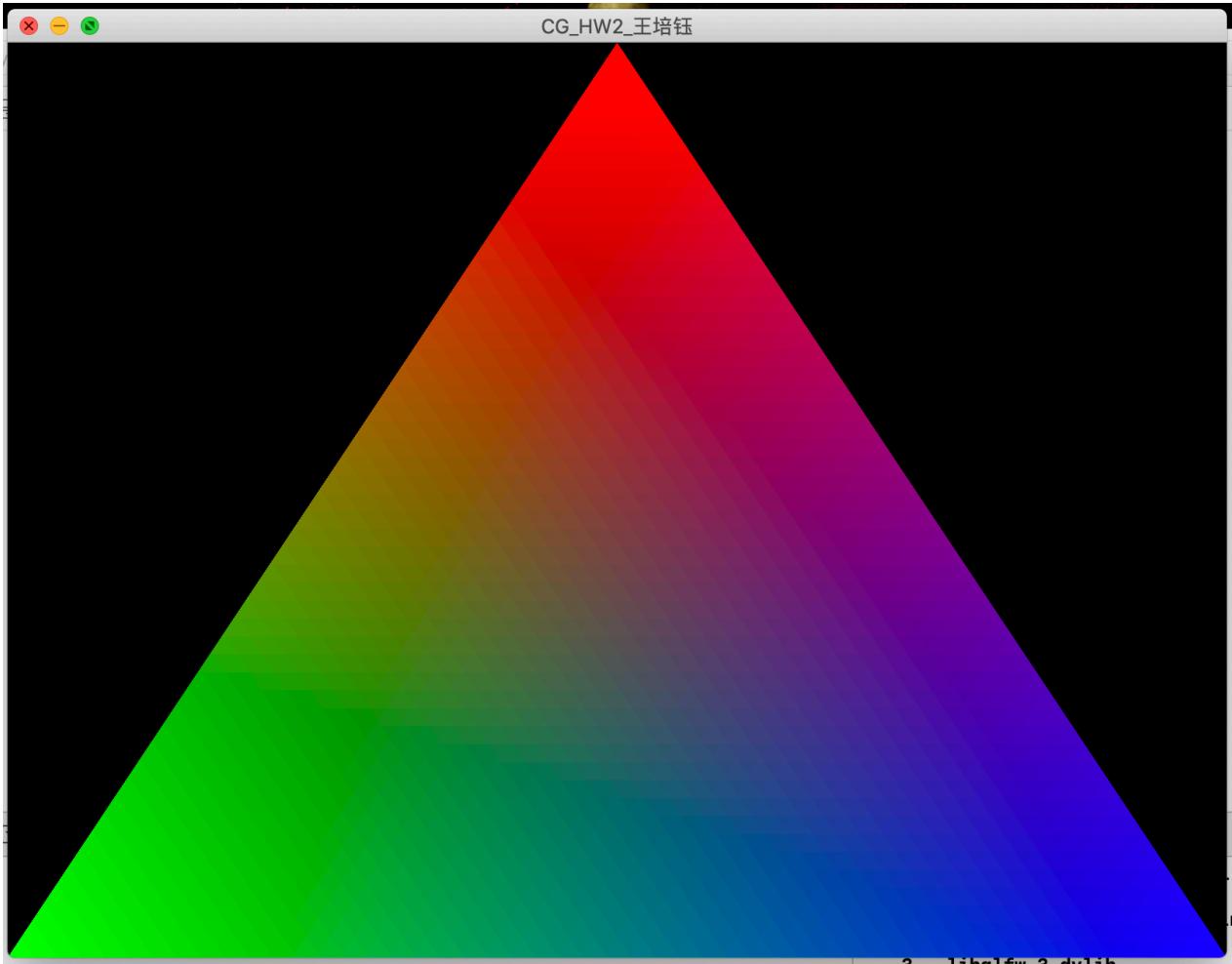
在输入顶点中添加颜色属性：

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

添加配置顶点颜色属性

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

效果：



现象解释：

这是在片段着色器中进行的片段插值(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。

光栅化会把顶点数据映射为像素点，生成供片段着色器使用的片段，同时会裁切出超出视图外的所有像素。也就是说虽然我们只输入三个顶点但是光栅化会补充其他构成三角形的片段像素点。然后片段着色器会根据我们提供的三个点的颜色值对其他的像素点进行等比例插值。

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。

比起前两步的工作：

- 将着色器封装成一个类文件，将着色器代码写到单独的文件中去
- 调用ImGui做出一个图形界面

窗口初始化设置：

```

// imgui窗口设置
IMGUI_CHECKVERSION();
ImGui::CreateContext();
ImGuiIO &io = ImGui::GetIO(); (void)io;
// 设置颜色主题
ImGui::StyleColorsLight();
// Setup Platform/Renderer bindings
ImGui_ImplGlfw_InitForOpenGL(window, true);
ImGui_ImplOpenGL3_Init("#version 330");

```

启动imGUI的框架并处理相应的IO事件

```

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

{
    //static float f = 0.0f;
    ImGui::Begin("HW2");
    ImGui::Text("change color");
    //ImGui::SliderFloat("float", &f, 0.0f, 1.0f);
    // 将更改的颜色存入color中
    ImGui::ColorEdit3("choose one color", (float *)&color, 1);
    // 勾选框，选中则使三角形保持三色渐变色，取消勾选则课进行全局颜色设置
    ImGui::Checkbox("default color", &default_color);
    if (default_color)
    {
        vertices[3] = 0.0f; vertices[4] = 1.0f; vertices[5] = 0.0f;
        vertices[9] = 0.0f; vertices[10] = 0.0f; vertices[11] =
1.0f;
        vertices[15] = 1.0f; vertices[16] = 0.0f; vertices[17] =
0.0f;
    }
    else
    {
        for (int i = 0; i < 3; i++) {
            vertices[i * 6 + 3] = color.x;
            vertices[i * 6 + 4] = color.y;
            vertices[i * 6 + 5] = color.z;
        }
    }
    ImGui::Text("Application average %.3f ms/frame (%.1f FPS)",
1000.0f / ImGui::GetIO().Framerate, ImGui::GetIO().Framerate);
    ImGui::End();
}

```

同时因为在每次IO操作都会改变三角形的属性，所以将VBO，VAO配置加入到窗口循环中，并用ImGUI进行渲染

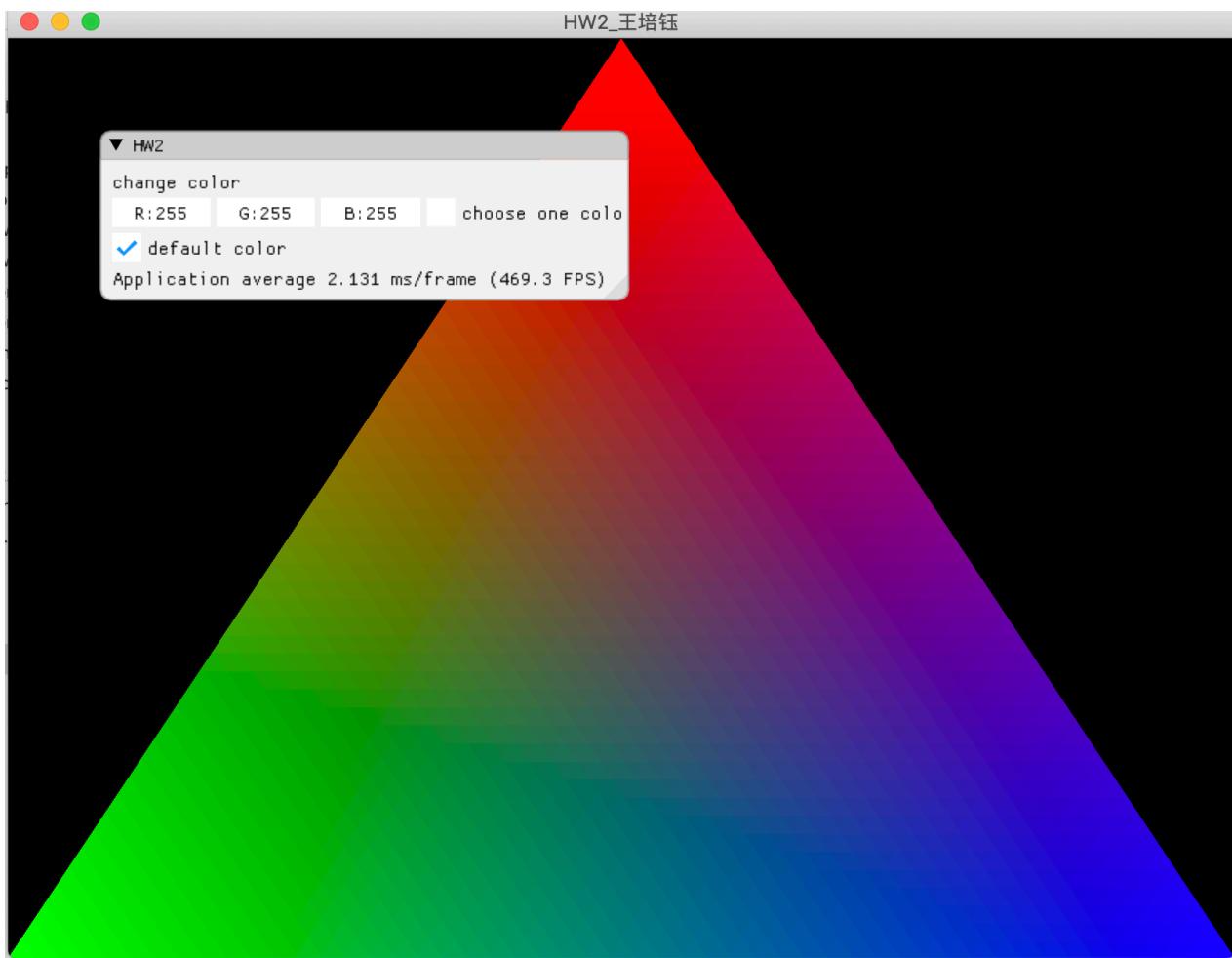
```
// 渲染  
ImGui::Render();  
ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());
```

最终clean up

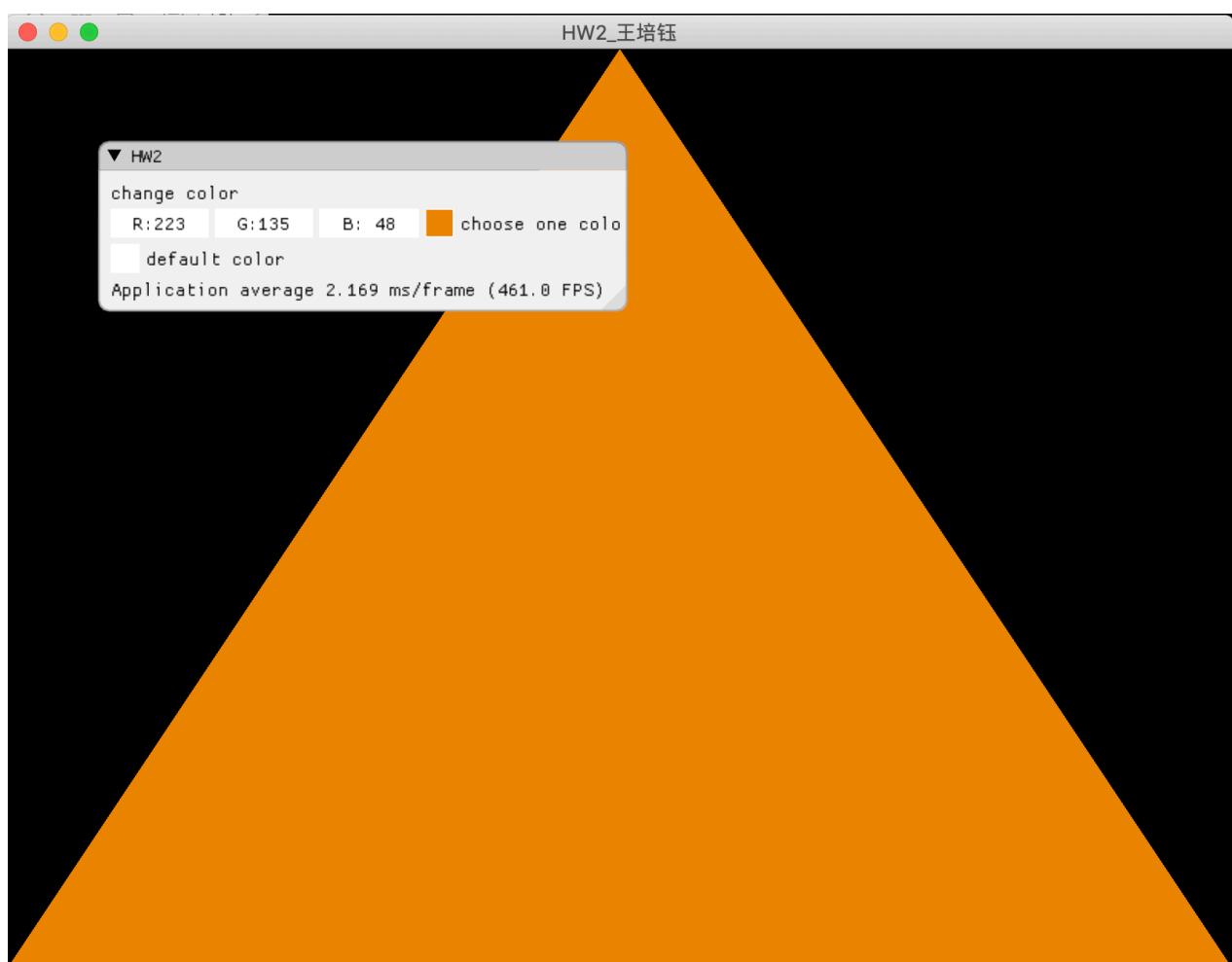
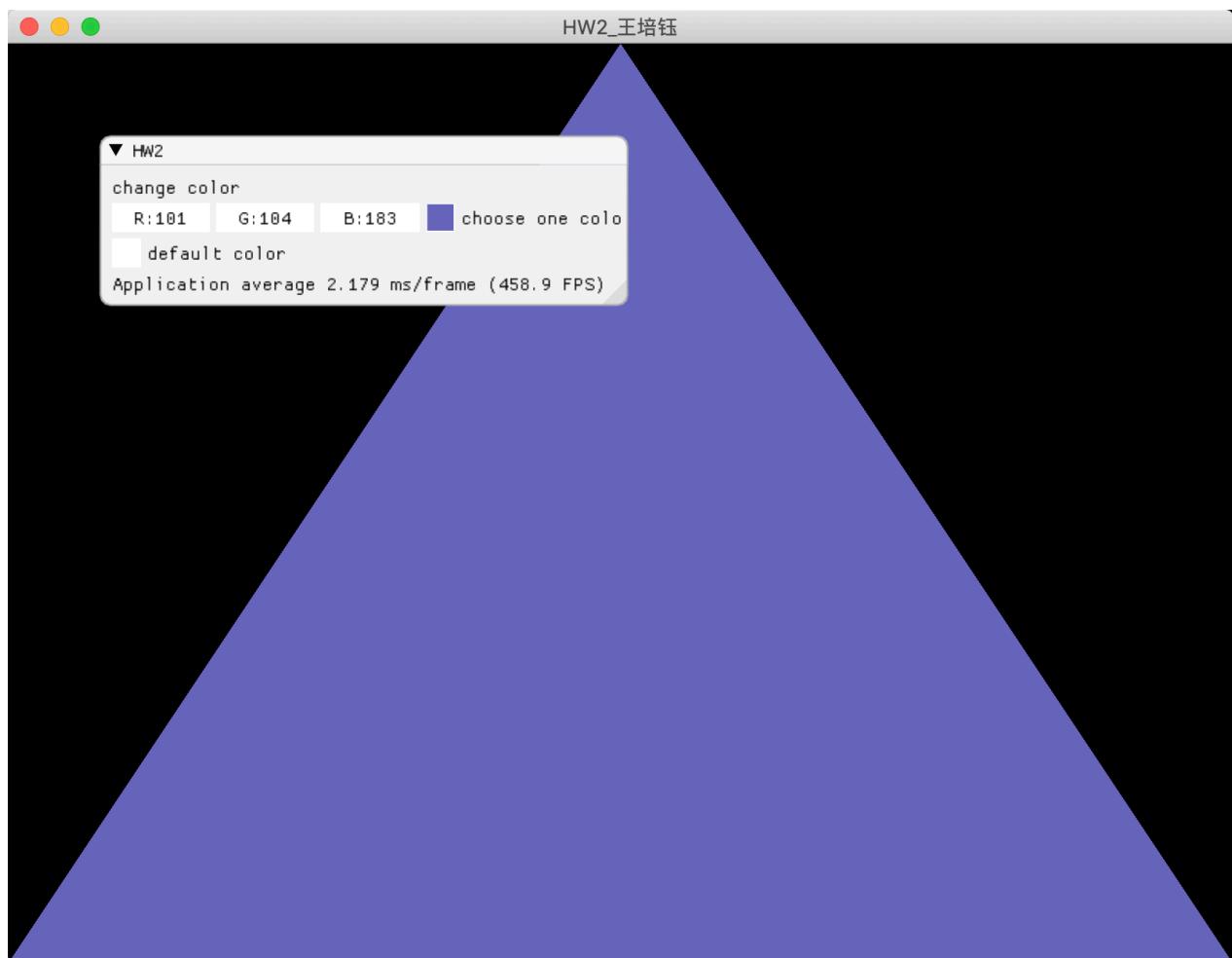
```
// 清除GUI资源  
ImGui_ImplOpenGL3_Shutdown();  
ImGui_ImplGlfw_Shutdown();  
ImGui::DestroyContext();
```

实现效果：

- 功能一：
 - 勾选default color三角形保持默认三原色渐变



- 功能二：
 - 不勾选default color，选择一个颜色，三角形变为选择的全局色



Bonus:

1. 绘制其他的图元，除了三角形，还有点、线等。

加入两条直线，6个点

```
// 直线1  
-0.5f, 0.0f, 0.0f, 0.9f, 0.9f, 0.666f, // 左下  
-1.0f, 1.0f, 0.0f, 0.9f, 0.9f, 0.666f, // 左上  
// 直线2  
0.5f, 0.0f, 0.0f, 0.8f, 0.36f, 0.36f, // 右下  
1.0f, 1.0f, 0.0f, 0.8f, 0.36f, 0.36f, // 右上  
// 点1  
0.75f, 0.05f, 0.0f, 0.0f, 0.0f, 0.0f, // 上  
// 点2  
0.75f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // 中  
// 点3  
0.75f, -0.05f, 0.0f, 0.0f, 0.0f, 0.0f, // 下  
// 点1  
0.8f, 0.05f, 0.0f, 0.0f, 0.0f, 0.0f, // 上  
// 点2  
0.8f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // 中  
// 点3  
0.8f, -0.05f, 0.0f, 0.0f, 0.0f, 0.0f // 下
```

加入勾选框，选择则出现对应的图形

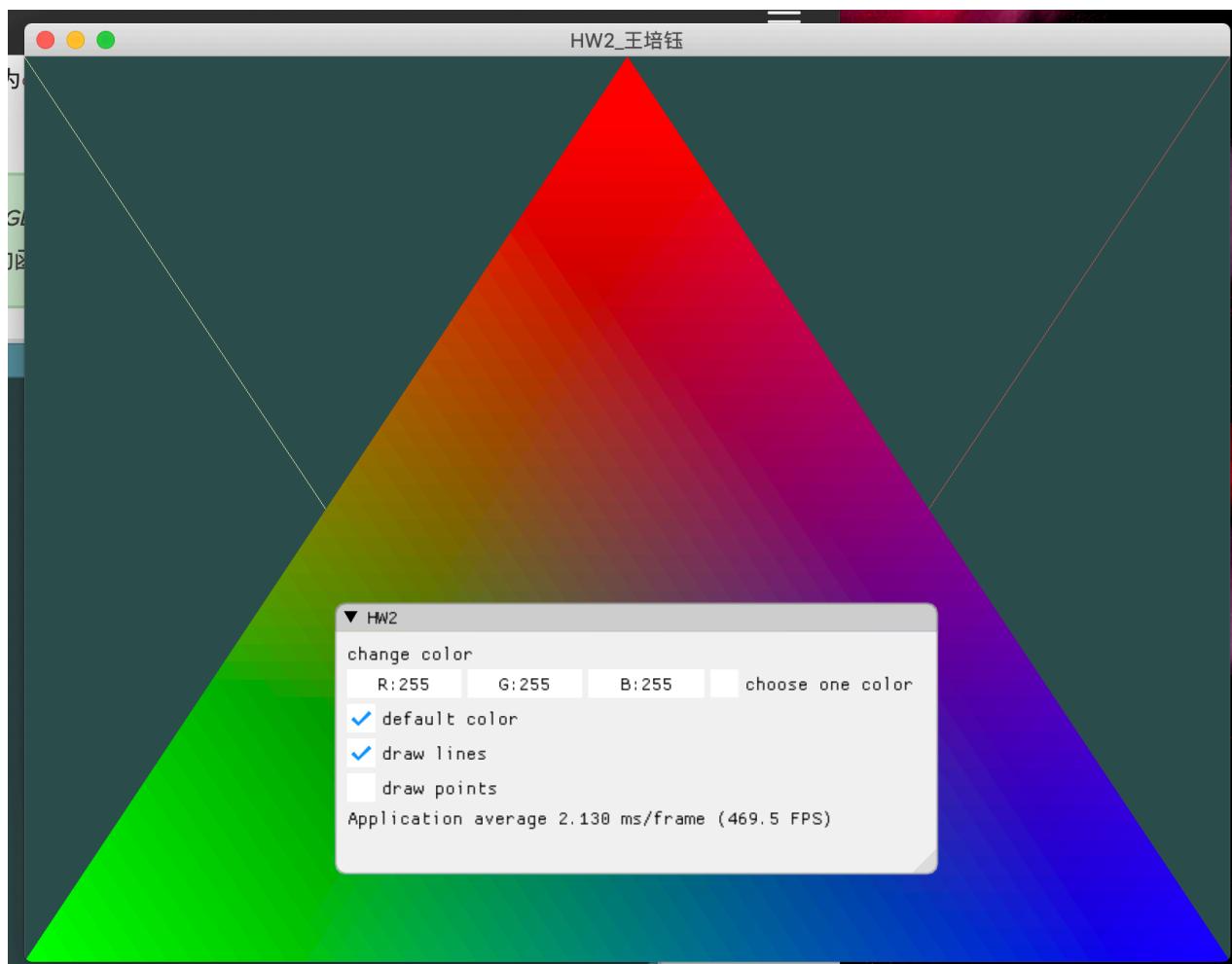
```
ImGui::Checkbox("default color", &default_color);  
ImGui::Checkbox("draw lines", &lines);  
ImGui::Checkbox("draw points", &points);
```

绘制图形

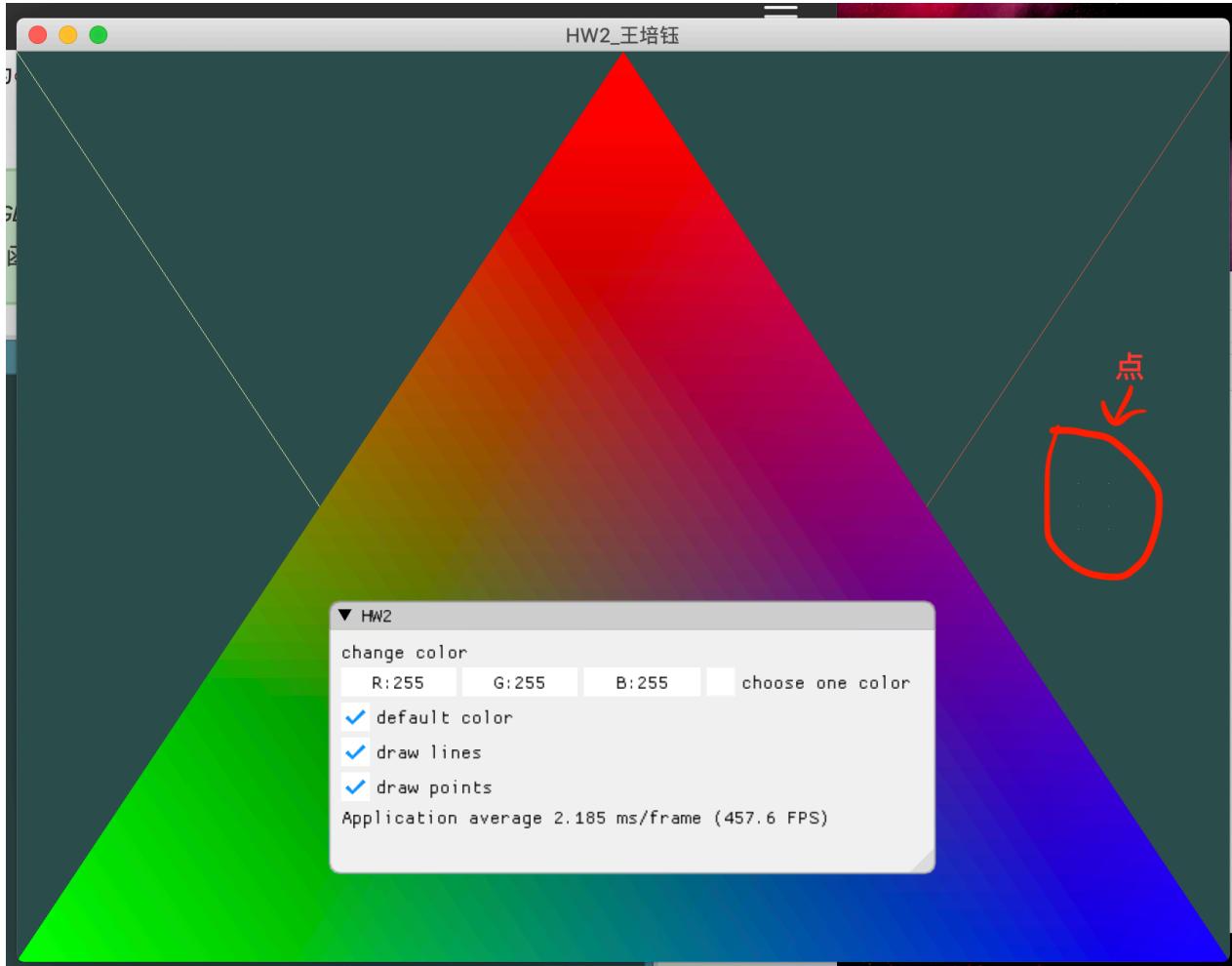
```
glDrawArrays(GL_LINES, 3, 4);  
glDrawArrays(GL_POINTS, 7, 6);
```

效果

- 直线：



- 点：



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

五个顶点索引2个三角形

```
// 三角形
-1.0f, -1.0f, 0.0f, 0.0f, 1.0f, 0.0f, // 左
1.0f, -1.0f, 0.0f, 0.0f, 0.0f, 1.0f, // 右
0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, // 上
0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, // 第二个三角形
0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, //
```

绘制索引

```
// 索引绘制
unsigned int indices[] = {
    0, 1, 2, // 第一个三角形
    2, 3, 4 // 第二个三角形
};
```

生成EBO对象

```
unsigned int VBO, VAO, EBO;  
// 生成对象  
glGenVertexArrays(1, &VAO);  
glGenBuffers(1, &VBO);  
glGenBuffers(1, &EBO);
```

将索引数组复制到索引缓冲中供openGL使用

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

绘制

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

