计算机图形学实验报告

第一部分：实现思路和代码分析

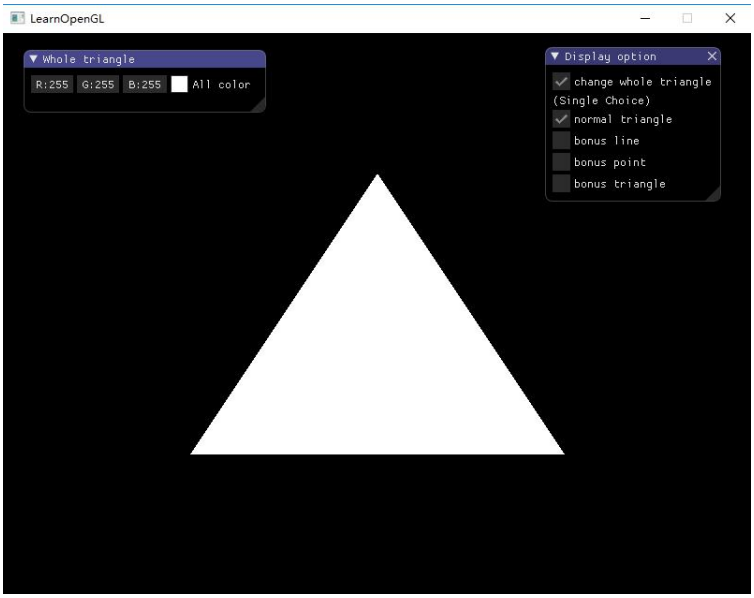
代码截图	代码说明
`// 创建顶点着色器` `const char* shader = "#version 330 core\n\` `layout(location = 0) in vec3 pos;\n\` `layout(location = 1) in vec3 color;\n\` `out vec3 outColor;\n\` `void main() {\n\` `gl_Position = vec4(pos, 1.0);\n\` `outColor = color;\n\` `}\0";` `// 创建片段着色器` `const char* fragment = "#version 330 core\n\` `in vec3 outColor;\n\` `out vec4 color;\n\` `void main() {\n\` `color = vec4(outColor, 1.0f);\n\` `}\0";`	用着色器语言 GLSL 编写顶点着色器和片段着色器。其中顶点着色器的输入为一个三维数组代表位置（pos）和一个三维数组代表颜色（color），输出为颜色。片段着色器输入一个三维的 RGB 数组，输出一个四维的 RGBA 数组。
`// 实例化GLFW窗口` `glfwInit();` `glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);` `glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);` `glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);` `glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);` `// 创建一个窗口对象` `GLFWwindow* window = glfwCreateWindow(WINDOW_WIDTH,` `WINDOW_HEIGHT,` `"LearnOpenGL",` `nullptr,` `nullptr);` `if (window == nullptr) {` `std::cout << "Failed to create GLFW window" << std::endl;` `glfwTerminate();` `return -1;` `}` `glfwMakeContextCurrent(window);`	实例化一个 GLFW 窗口，创建一个 GLFW 的窗口对象，用于显示渲染图形。
`// 开始游戏循环之前注册函数至合适的回调(Esc退出)` `glfwSetKeyCallback(window, key_callback);`	设置窗口和键盘输入之间的回调函数，这里设置了 ESC 键为退出窗口的功能。
`// 初始化GLEW/GLAD` `if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {` `std::cout << "Failed to initialize GLAD" << std::endl;` `return -1;` `}`	初始化 GLAD 组件。这里与教程稍有不同，我使用了更新的 GLAD 库替换原来的 GLEW 库。
`// 编译顶点着色器` `unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);` `glShaderSource(vertexShader, 1, &shader, NULL);` `glCompileShader(vertexShader);` `// 检测是否成功编译` `int success;` `char infoLog[512];` `glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);` `if (!success) {` `glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);` `std::cout << "ERROR: " << infoLog << std::endl;` `}`	这部分代码用于编译顶点着色器以便后续调用。使用了先前设置好的 GLSL 语言编写的顶点着色器代码作为输入。
`// 编译片段着色器` `unsigned int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);` `glShaderSource(fragmentShader, 1, &fragment, NULL);` `glCompileShader(fragmentShader);` `// 检测是否成功编译` `glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);` `if (!success) {` `glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);` `std::cout << "ERROR: " << infoLog << std::endl;` `}`	这部分代码用于编译片段着色器以便后续调用。使用了先前设置好的 GLSL 语言编写的片段着色器代码作为输入。
`// 创建着色器程序` `unsigned 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: " << infoLog << std::endl;` `}`	这部分代码用于创建着色器程序。将前面的生成的顶点着色器和片段着色器绑定到当前的着色器程序上。
`while (!glfwWindowShouldClose(window)) {`	开始进行游戏循环
`// 检查事件` `glfwPollEvents();`	GLFW 需要定期与窗口通信，以便接收事件。这里使用轮询模式。

`` // 顶点坐标 GLfloat vertices[] = { // 位置(x,y,z) // 颜色(r,g,b) -0.5f, -0.5f, 0.0f, left.x, left.y, left.z, // 左下 0.5f, -0.5f, 0.0f, right.x, right.y, right.z, // 右下 0.0f, 0.5f, 0.0f, top.x, top.y, top.z, // 上 -0.25f, -0.9f, 0.0f, leftColor1.x, leftColor1.y, leftColor1.z, // 左下 0.25f, -0.9f, 0.0f, rightColor1.x, rightColor1.y, rightColor1.z, // 右下 0.0f, -0.5f, 0.0f, topColor1.x, topColor1.y, topColor1.z // 上 }; unsigned int indices[] = { // 注意索引从0开始 0, 1, 2, // 第一个三角形 3, 4, 5 // 第二个三角形 }; ```	设置待渲染的三角形的各顶点参数。每个三角形占三行空间，每一行前三个数值代表该顶点的标准化设备坐标，后三个代表其 RGB 值。
`` /*总流程*/ // 使用glGenBuffers函数和一个缓冲ID生成一个VBO对象 glGenBuffers(1, &VBO); // 使用glGenBuffers函数和一个缓冲ID生成一个EBO对象 glGenBuffers(1, &EBO); // 使用glGenBuffers函数和一个缓冲ID生成一个VAO对象 glGenVertexArrays(1, &VAO); // 绑定VAO glBindVertexArray(VAO); // 缓冲类型:顶点缓冲 glBindBuffer(GL_ARRAY_BUFFER, VBO); // 顶点数据复制到缓冲的内存中 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW); // 缓冲类型:索引缓冲 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO); // 索引数据复制到缓冲的内存中 glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW); // 顶点属性指针 // 位置 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0); glEnableVertexAttribArray(0); // 颜色 glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float))); glEnableVertexAttribArray(1); // 解除绑定 glBindBuffer(GL_ARRAY_BUFFER, 0); ```	链接顶点属性过程。使用 OpenGL 函数为渲染对象分配内存和缓冲空间，同时将三角形的顶点对象添加到顶点缓冲对象（VBO），同时将对应的 VBO 指针添加到顶点数组对象 VAO 对象上，以供后续调用。
`` // ImGUI生成 ImGui_ImplOpenGL3_NewFrame(); ImGui_ImplGlfw_NewFrame(); ImGui::NewFrame(); // 设置GUI窗口内容 bool showWindow = true; if (showWindow) { if (!change_all_color) { ImGui::Begin("Color Choose"); ImGui::BeginChild("triangle 1", ImVec2(380, 90)); ImGui::Text("Normal triangle"); ImGui::ColorEdit3("top color", (float*)&topColor0); ImGui::ColorEdit3("bottom left color", (float*)&leftColor0); ImGui::ColorEdit3("bottom right color", (float*)&rightColor0); ImGui::EndChild(); ImGui::BeginChild("triangle 2", ImVec2(380, 90)); ImGui::Text("Bonus triangle"); ImGui::ColorEdit3("top color", (float*)&topColor1); ImGui::ColorEdit3("bottom left color", (float*)&leftColor1); ImGui::ColorEdit3("bottom right color", (float*)&rightColor1); ImGui::EndChild(); ImGui::End(); } ImGui::Begin("Display option", &showWindow); ImGui::Checkbox("change whole triangle", &change_all_color); ImGui::Text("(Single Choice)"); ImGui::Checkbox("normal triangle", &show_normal_triangle); ImGui::Checkbox("bonus line", &show_bonus_line); ImGui::Checkbox("bonus point", &show_bonus_point); ImGui::Checkbox("bonus triangle", &show_bonus_triangle); ImGui::End(); if (change_all_color) { ImGui::Begin("Whole triangle"); ImGui::ColorEdit3("All color", (float*)&sameColor); ImGui::End(); } } ```	创建并初始化 ImGUI 界面。与多个显示选项绑定，用于实现不同图像的显示。
`` // 渲染指令 ImGui::Render(); int display_w, display_h; glfwMakeContextCurrent(window); glfwGetFramebufferSize(window, &display_w, &display_h); glViewport(0, 0, display_w, display_h); // 清除窗口的移动轨迹 glClearColor(clear_color.x, clear_color.y, clear_color.z, clear_color.w); glClear(GL_COLOR_BUFFER_BIT); ```	渲染 ImGUI 界面的代码部分。
`` glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, (void*)0); glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, (void*)0); // bonus line为两个三角形的底边 glDrawArrays(GL_LINE_STRIP, 0, 2); glDrawArrays(GL_LINE_STRIP, 3, 2); // bonus point为两个三角形的顶点 glDrawArrays(GL_POINTS, 0, 6); ```	渲染简单的三角形以及 Bonus 的内容。

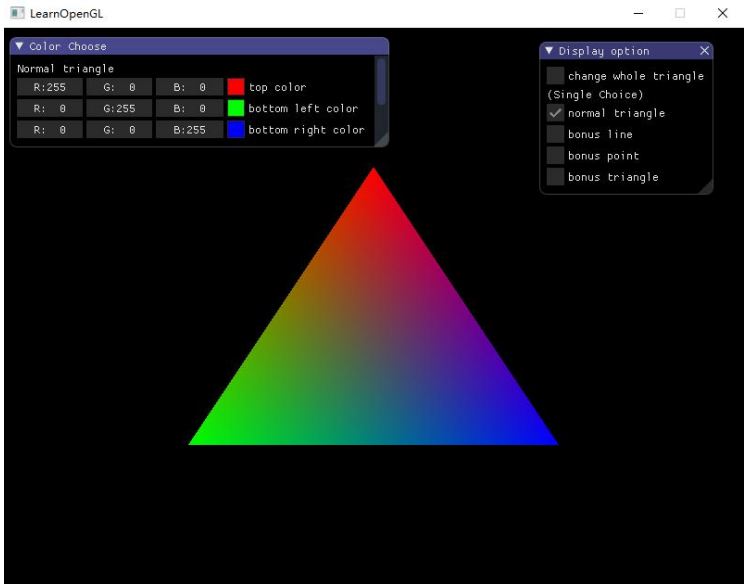
`// 交换缓冲` `glfwSwapBuffers(window);` `// 释放VBO, VAO, EBO` `glDeleteVertexArrays(1, &VAO);` `glDeleteBuffers(1, &VBO);` `glDeleteBuffers(1, &EBO);`	交换缓冲并释放申请的 VAO,VBO,EBO 对象的内存。这部分非常重要，如果申请的内存不能及时被释放，很可能造成内存占用过大的情况。
`// Cleanup` `ImGui_ImplOpenGL3_Shutdown();` `ImGui_ImplGlfw_Shutdown();` `ImGui::DestroyContext();` `//=============================================` `// 释放GLFW分配的内存` `glfwDestroyWindow(window);` `glfwTerminate();`	关闭窗口后释放 ImGUI 界面和 GLFW 所占用的系统内存。

第二部分：运行结果截图和相关问题解答

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

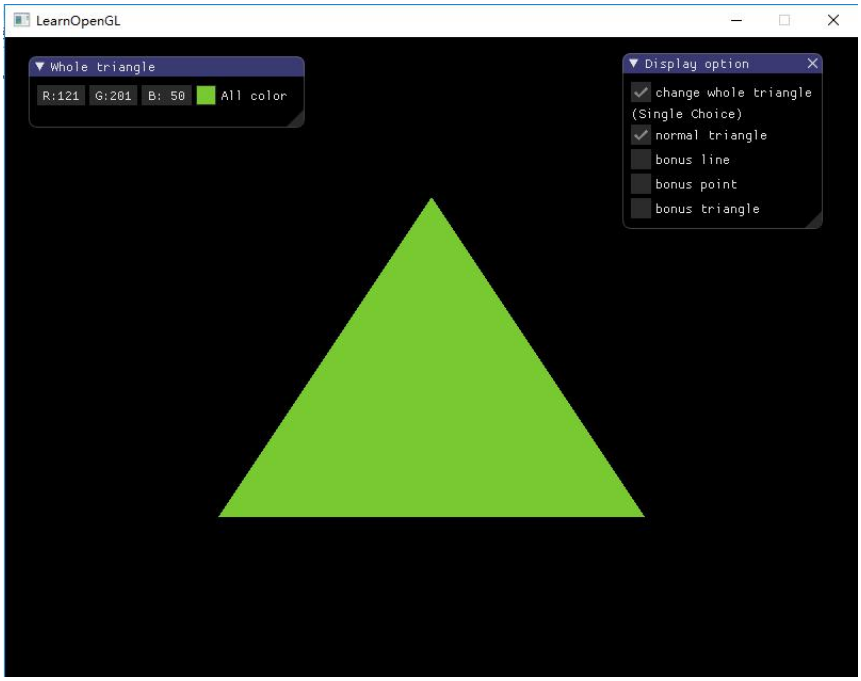


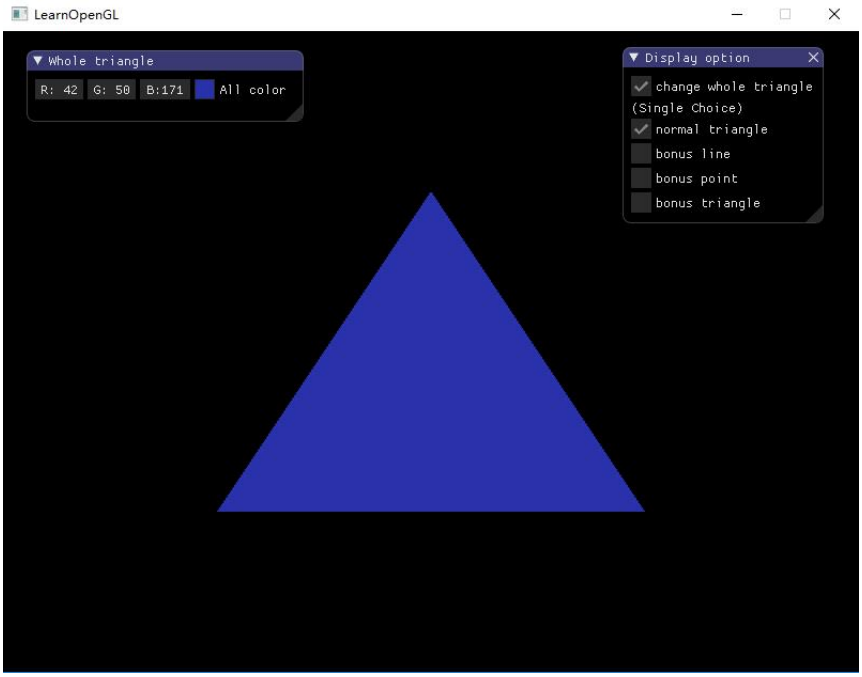
出现这种结果的原因：

图形渲染管线在光栅化阶段中会使用片段着色器对渲染图像上的各个像素计算他们最终呈现出来的颜色。在光栅化阶段结束后，所有对应的颜色值都被正确确定，然后图像渲染管线会把图像对象传到 Alpha 测试和混合阶段，这个阶段会检查图像的 alpha 值，即一个物体的透明度，并对物体进行混合。这个混合过程其实是当前要绘制的物体的颜色和颜色缓冲区中已经绘制的颜色进行混合，最终决定了当前物体的颜色。因此即使在片段着色器中计算出来了一个像素输出的颜色，在渲染的时候最后的像素颜色也可能完全不同。混合操作的结果就是三角形中部出现的渐变效果。

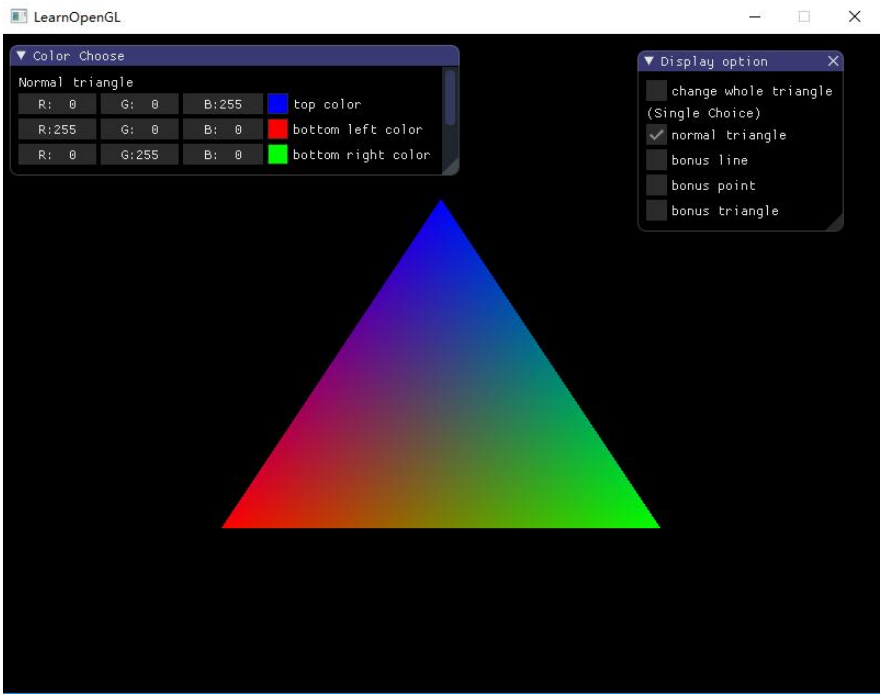
3. 给上述工作添加一个 GUI，里面有一个菜单栏，使得可以选择并改变三角形的颜色

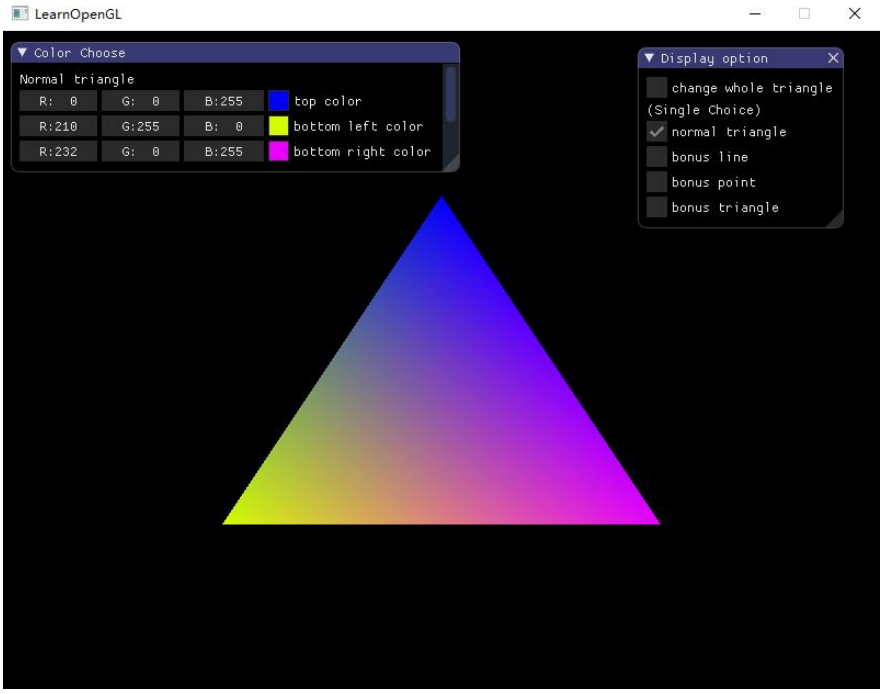
（1）修改整个三角形





（2）单独修改每个定点的颜色

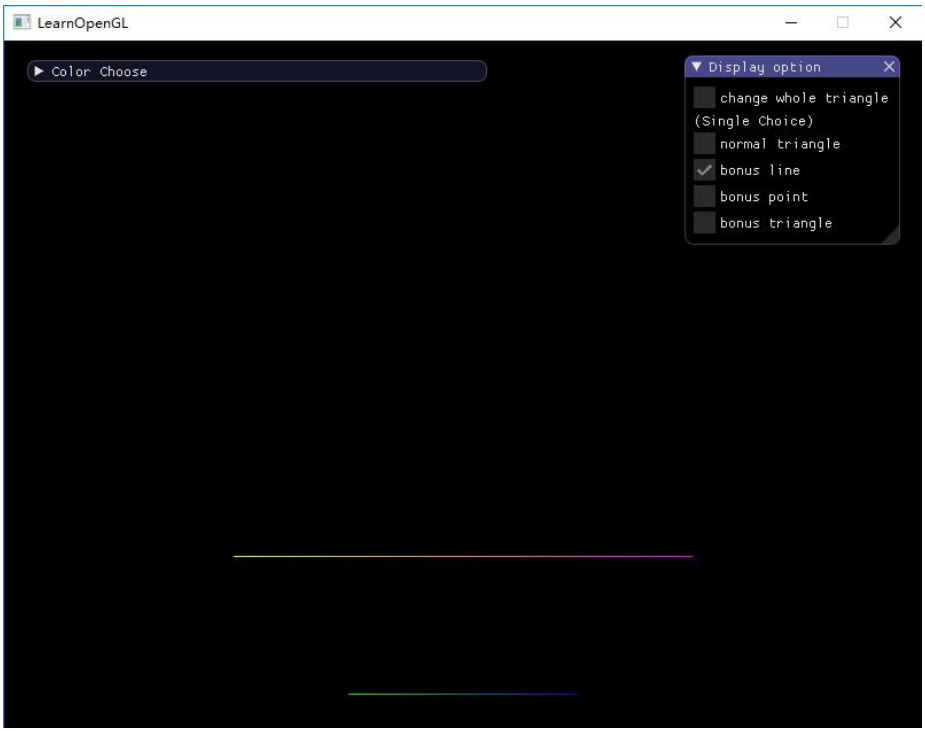




4. Bonus

（1）绘制其他图元（点，线）

（图中白点）



（2）使用 EBO 绘制多个三角形