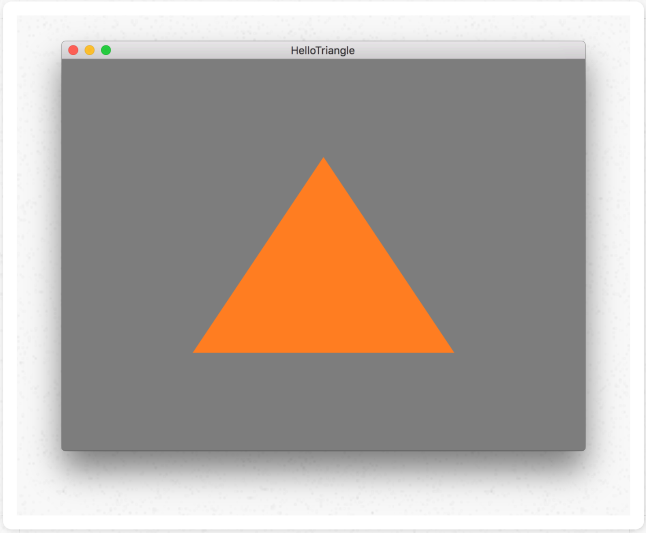
14331047-陈主润-HW2

2.2 Hello, Triangle!

相关工具：OpenGL、GLFW、GLEW
IDE：Xcode、VS（与代码无关，所以在Xcode上写完后又在VS上测试）

1. Draw the Triangle

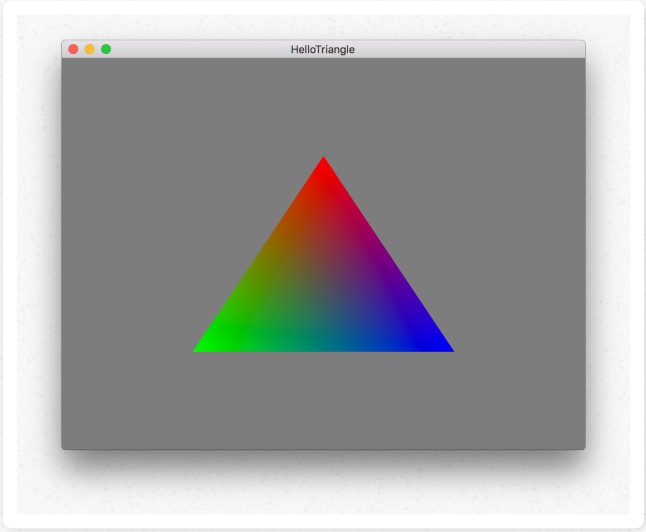
结果如下：



---

2. Change the color of 3 vertices

结果如下：



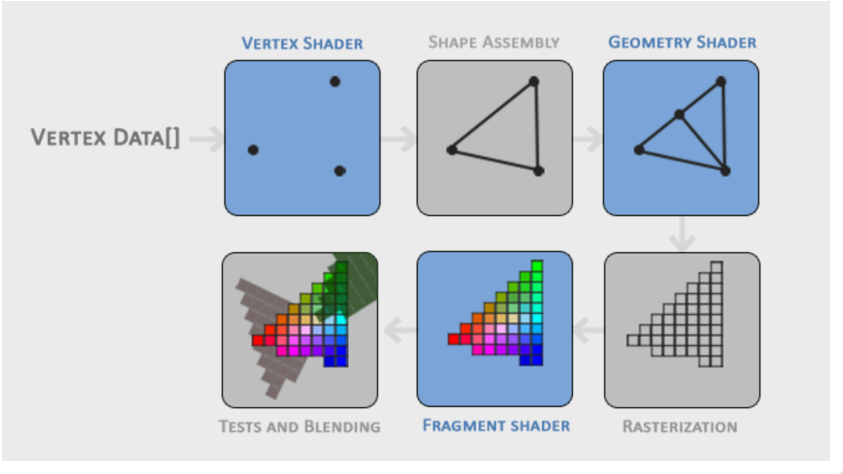
原因：会产生这样的结果，是受**光栅化过程（离散和插值）**的影响。在渲染过程中，经过光栅化时，首先将三角形内的点离散化，分成尽可能小的块，并给每个小块计算出到3个顶点的距离，根据这个距离以及三

个顶点的颜色，按比例给这一小块计算出一个颜色值，最后由fragment着色器进行着色。这样，经过离散和插值两个过程后，就渲染出了这样一个三角形。

---

3. Algorithms

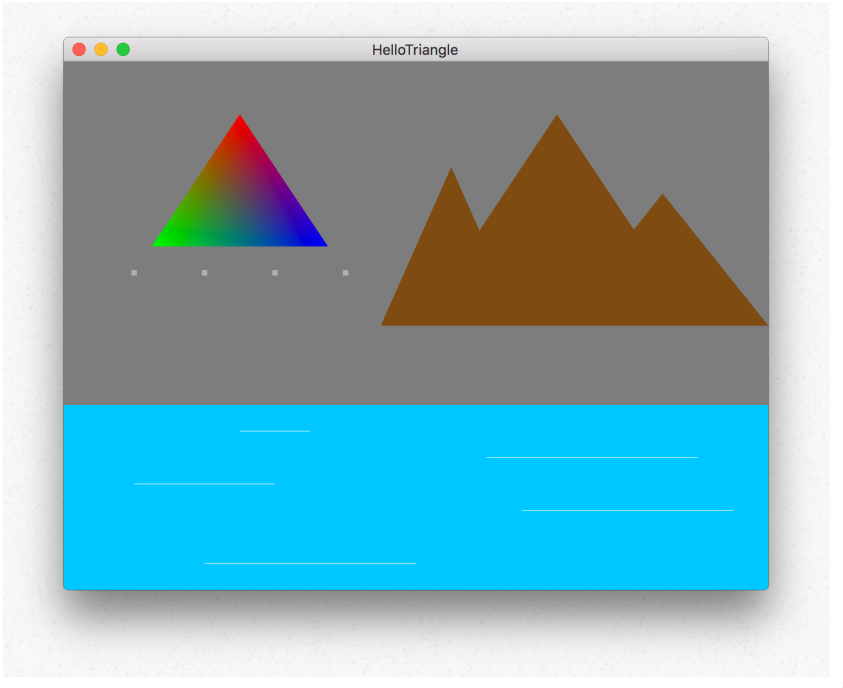
通过上课和课后网上学习了解了**3D图形渲染的流水**，这个过程主要如下图：



流程：输入顶点数据后：

- 首先由顶点着色器进行处理，该过程主要包括顶点坐标转换（Object Space->World Space->Camera Space->Screen Space）以及对顶点属性的一些基本处理
- 到达光栅化阶段后，就是上面提到的离散化以及插值的过程
- 而片段着色器，就是对图形每个片段进行着色的过程。 经过这几步，就已经基本可以得到我们这次作业要的图形了。

实现：
为了更深入的学习OpenGL以及理解图形渲染流水的过程，就用了核心模式来完成。通过课程的理解和在LearnOpenGL以及Anton's OpenGL Tourial的学习，实现了VertexShader和FragmentShader，最终绘制出一幅“有山有水”的图，如下：



其中由上方有座由三角形组成的“山”（彩色三角形及下面四个灰点是作业需要），下面的“河流”是由矩形和5条线组成，其中矩形是由两个全等三角形拼接而成的。

在我的实现中，创建了一个Shader类，是一个链接了顶点着色器和片段着色器的一个“程序”。这里的顶点着色器主要是存储顶点颜色和位置两个属性，而片段着色器则负责着色：

```
//build shader program
Shader myShader(vertexShaderSource, fragmentShaderSource);
```

两个参数分别为vertexShader和fragmentShader的GLSL待执行代码串，也是它们的功能代码：

```
//source codes
const GLchar* vertexShaderSource =
    "#version 330 core\n"
    "layout (location = 0) in vec3 position;\n"
    "layout (location = 1) in vec3 color;\n"
    "out vec3 myColor;\n"
    "void main() {\n"
        "gl_Position = vec4(position, 1.0);\n"
        "myColor = color;\n"
    "}";

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

创建Shader后，就要获取顶点数据。顶点数据是存在内存中的，必须通过CPU读取后传到GPU中来才能处理。而从CPU将数据传到GPU中时，速度相对于GPU内部传输是很慢的。因而，数据的处理也会变得非常慢。

VBO

为了解决这个问题，GPU中就创建了一个专门用来存储大量顶点数据的缓存VBO（Vertex Buffer Objects）。这样，在CPU传输数据给GPU时，每次传输就能传输尽可能多的数据，并把数据临时存储在VBO中。

VAO

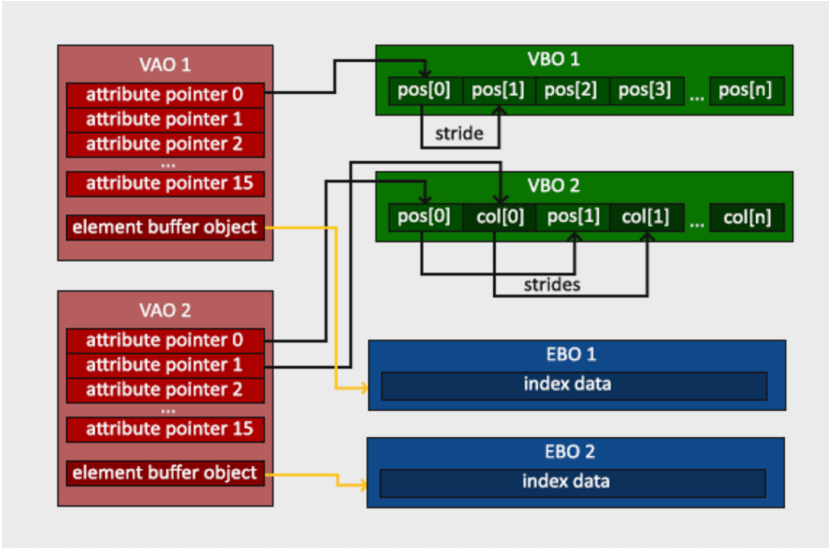
如果仅仅是有VBO的话，我们只是有了缓存来存顶点数据；但是如果顶点数据很多，其中就包括如颜色、位置的属性。那么要如何区分这些数据哪些是表示位置的、哪些是表示颜色？为此，Core OpenGL中还有VAO（Vertex Array Object）来解决这一问题。简单的说，VAO中存储的是状态，这些状态就是VBO中顶点数据存储的状态（即哪些是表示颜色、哪些表示位置）。这样，Shader程序在执行时才能区分这些属性。

EBO

EBO（Elements Buffer Objects）专门存储索引，这些索引是顶点数据的索引。例如，如果要绘制一个矩形，我们可以用两个全等三角形来实现。而实际上只需要4个顶点来绘制这2个三角形。EBO就存储了绘制这两个三角形的索引数据，OpenGL绘制时，就会调用这些顶点的索引来绘制。如下：

```
// draw rectangle with two triangle(as river)
GLfloat rectangle_points[] = {
    //points(3)           //colors(3)
    -1.0f, -0.3f, 0.0f,   0.0f, 0.8f, 1.0f,
    1.0f,  -0.3f, 0.0f,   0.0f, 0.8f, 1.0f,
    1.0f,  -1.0f, 0.0f,   0.0f, 0.8f, 1.0f,
    -1.0f, -1.0f, 0.0f,   0.0f, 0.8f, 1.0f
};
GLuint river_indices[] = {
    0, 1, 2,
    0, 2, 3
};
```

VBO、VAO、EBO三者的关系如下图所示：



理解后，就开始实现。以下是作业实现中的"河流"部分的具体实现（用2个三角形来绘制矩形）：

```
//draw a rectangle as rivers
//rectangle
GLuint rectangle_VAO, rectangle_VBO, river_EBO;
glGenVertexArrays(1, &rectangle_VAO);
glGenBuffers(1, &rectangle_VBO);
glGenBuffers(1, &river_EBO);
glBindVertexArray(rectangle_VAO);
glBindBuffer(GL_ARRAY_BUFFER, rectangle_VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(rectangle_points), rectangle_points, GL_STATIC_DRAW);
//bind EBO
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, river_EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(river_indices), river_indices, GL_STATIC_DRAW);
//
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
//enable array buffer
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
//unable array VBO and VAO
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

`glGenBuffers`、`glBindBuffer`等部分代码都是VBO、VAB、EBO的创建、绑定的实现。
主要是`glVertexAttribPointer`这里，第一次用是要说明每个顶点（包括所有属性）的偏移是多少；第二次调用是要说明一个顶点中每个属性（这里是位置、颜色）的偏移是多少。这样才能最后渲染出来。

渲染部分代码：

```
//draw river
glBindVertexArray(rectangle_VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
glBindVertexArray(0);
```

绘制前先绑定VAO，然后用`glDrawElements`来绘制（由于使用了EBO）。

---

4. Bonus

提交的最终可执行程序即为Bonus部分，也包括了前面的普通三角形和上色三角形的绘制：
绘制的图形包括点、三角形、线、矩形：