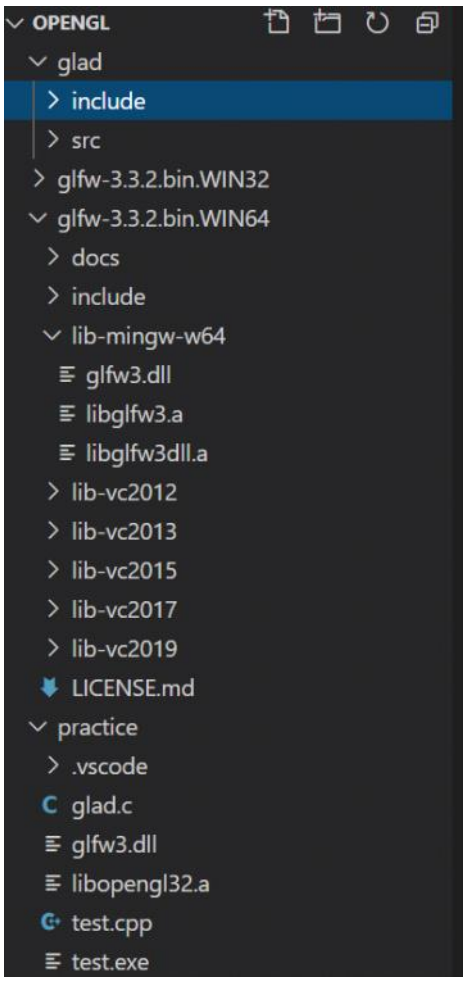
项目目录结构

Sunday, 01 November 2020 00:43



glfw相关：其中glfw.dll是在window运行时的动态库，libglfwdll.a是编译时用的动态库

编译参数

2020年11月1日 13:11

```
"args": [
    "-g",
    "${file}","glad.c",
    "-o",
    "${fileDirname}\\${fileBasenameNoExtension}.exe",
    //"-I","${workspaceFolder}/**",
    "-I","../glad/include",
    "-I","../glfw-3.3.2.bin.WIN64/include",
    "-L","../glfw-3.3.2.bin.WIN64/lib-mingw-w64",
    "-lglfw3dll",
    //"-lopengl32",
```

注意这里的参数"-lglfw3dll"，而根据官方文档中提到

> The link library for the GLFW DLL is named `glfw3dll`. When compiling an application that uses the DLL version of GLFW, you need to define the **GLFW_DLL** macro *before* any inclusion of the GLFW header. This can be done either with a compiler switch or by defining it in your source code.

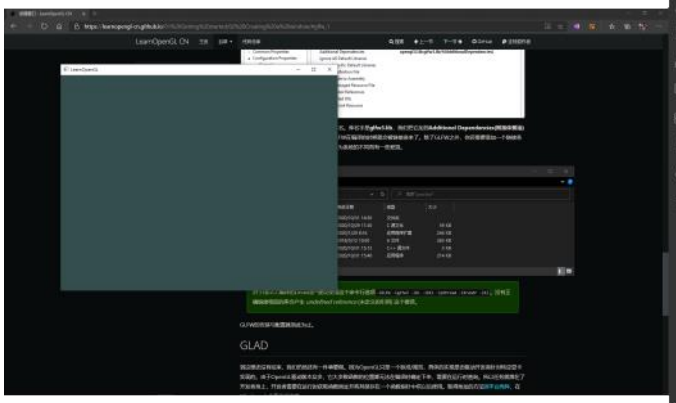
于是在test.cpp最前面加上宏定义

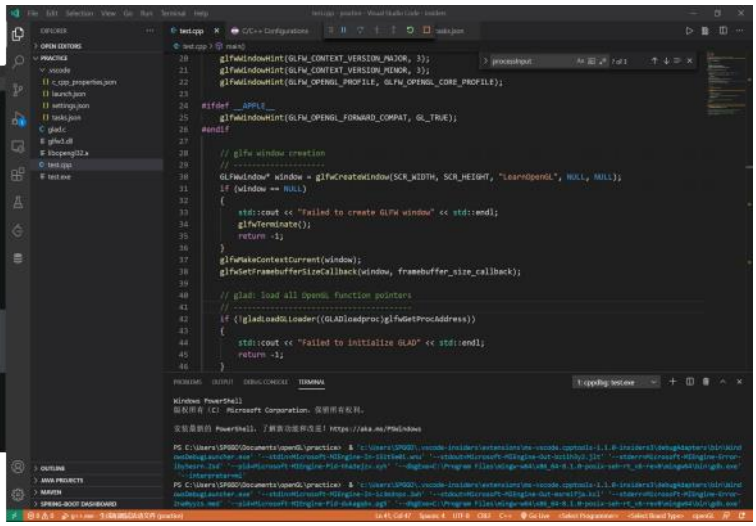
```
#define  GLFW_DLL
```

编译通过

2020年11月1日 13:12

使用https://learnopengl.com/（下文称为官网）的代码，编译通过，左侧成功创建窗口

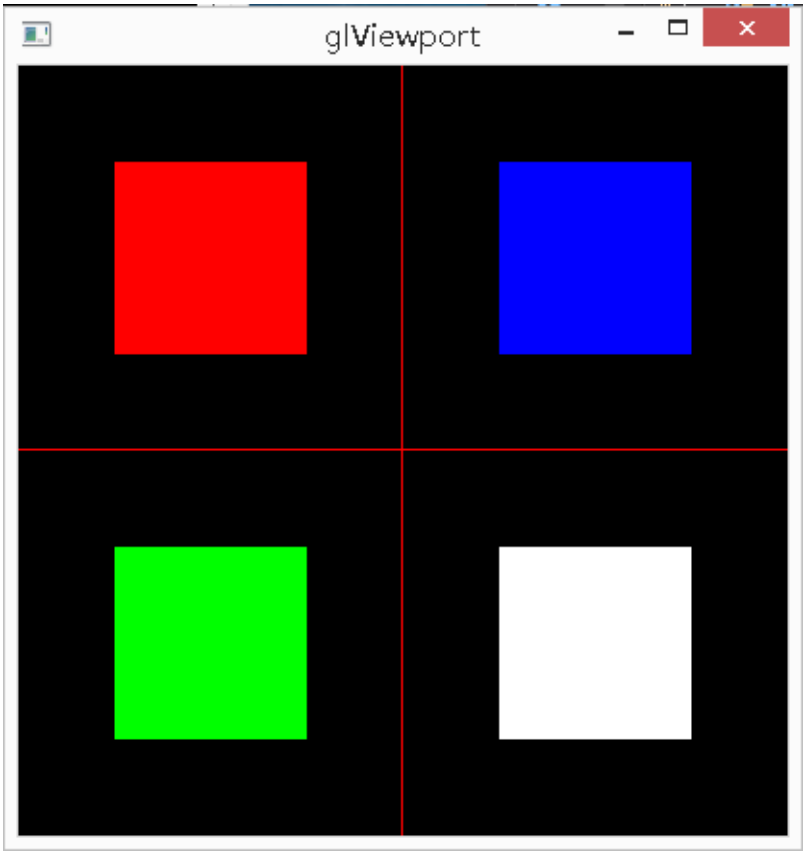




Viewport

Sunday, 01 November 2020 00:44

```
//定义左下角区域
glColor3f(0.0, 1.0, 0.0);
glViewport(0, 0, 200, 200);
glBegin(GL_POLYGON);
glVertex2f(-0.5, -0.5);
glVertex2f(-0.5, 0.5);
glVertex2f(0.5, 0.5);
glVertex2f(0.5, -0.5);
glEnd();
//定义右上角区域
glColor3f(0.0, 0.0, 1.0);
glViewport(200, 200,200, 200);
glBegin(GL_POLYGON);
glVertex2f(-0.5, -0.5);
glVertex2f(-0.5, 0.5);
glVertex2f(0.5, 0.5);
glVertex2f(0.5, -0.5);
glEnd();
//定义在左上角的区域
glColor3f(1.0, 0.0, 0.0);
glViewport(0, 200, 200, 200);
glBegin(GL_POLYGON);
glVertex2f(-0.5, -0.5);
glVertex2f(-0.5, 0.5);
glVertex2f(0.5, 0.5);
glVertex2f(0.5, -0.5);
glEnd();
//定义在右下角的区域
glColor3f(1.0, 1.0, 1.0);
glViewport(200, 0, 200, 200);
glBegin(GL_POLYGON);
glVertex2f(-0.5, -0.5);
glVertex2f(-0.5, 0.5);
glVertex2f(0.5, 0.5);
glVertex2f(0.5, -0.5);
glEnd();
```



如上，一个窗口（window）里面可包含多个视口（viewport），以红框框区域为例，它的四个参数（0，200，200，200），前两个表示红框框区域左下角，后两个表示长宽（窗口大小为400*400）。而在一个viewport中，坐标的范围是（-1，1），即红框框区域的中心点是（0，0），那红方块四个顶点（vertex）的坐标就是0.5的那些东西。

Register a Callback Function

2020年11月1日 13:14

在glfw中，可以注册许多回调函数，比如

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
}
```

We do have to tell GLFW we want to call this function on every window resize by registering it:

```
glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
```

第一个函数是在窗口大小发生变化时的回调函数，第二个则是向glfw注册第一个回调函数。

除此以外，还可以注册由键盘或者鼠标触发的回调函数，而这些函数往往被如下调用

```
while(!glfwWindowShouldClose(window))
{
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

其中glfwPollEvents()函数就是检查是否有注册的函数对应的事件发生（类似中断查询）。这个整个while循环用于一直渲染窗口，使窗口保持open。

Double Buffer

2020年11月1日 13:14

When an application draws in a single buffer the resulting image may display flickering issues. This is because the resulting output image is not drawn in an instant, but drawn pixel by pixel and usually from left to right and top to bottom. Because this image is not displayed at an instant to the user while still being rendered to, the result may contain artifacts. To circumvent these issues, windowing applications apply a double buffer for rendering.
The front buffer contains the final output image that is shown at the screen, while all the rendering commands draw to the back buffer. As soon as all the rendering commands are finished we swap the back buffer to the front buffer so the image can be displayed without still being rendered to, removing all the aforementioned artifacts

Double buffer其实就是通过swap buffer（）函数实现的

Close Window

2020年11月1日 13:14

在点击“X”关闭窗口之后，实际上窗口并不会直接关闭，而是经由
glfwWindowShouldClose(window)
函数判断出，本窗口已经被点击了关闭，“should close”，然后退出render loop 的while循环，再调用
glfwTerminate()
释放资源，最后关闭窗口

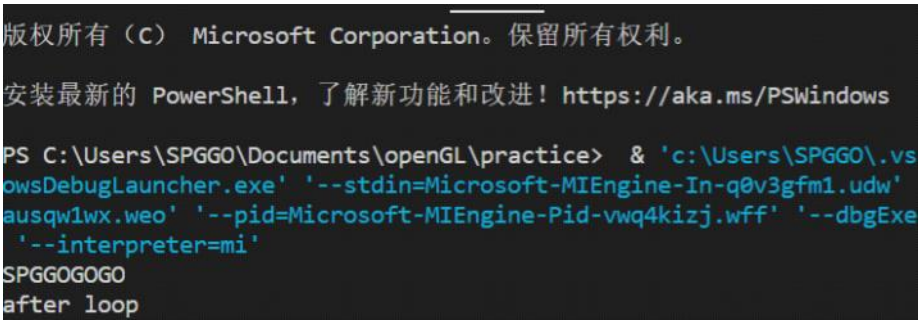
测试如下：

```
if(glfwWindowShouldClose(window))
{
    std::cout << "before loop" << std::endl;
}
// render loop
// -----------
while (!glfwWindowShouldClose(window))
{
    // input
    // -----
    processInput(window);

    // render
    // ------
    glClearColor(0.f, 0.f, 0.f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // glfw: swap buffers and poll IO events (keys pressed/released, mouse moved etc.)
    // -------------------------------------------------------------------------------
    glfwSwapBuffers(window);
    glfwPollEvents();
}
if(glfwWindowShouldClose(window))
{
    std::cout << "after loop" << std::endl;
}
```

输出如下

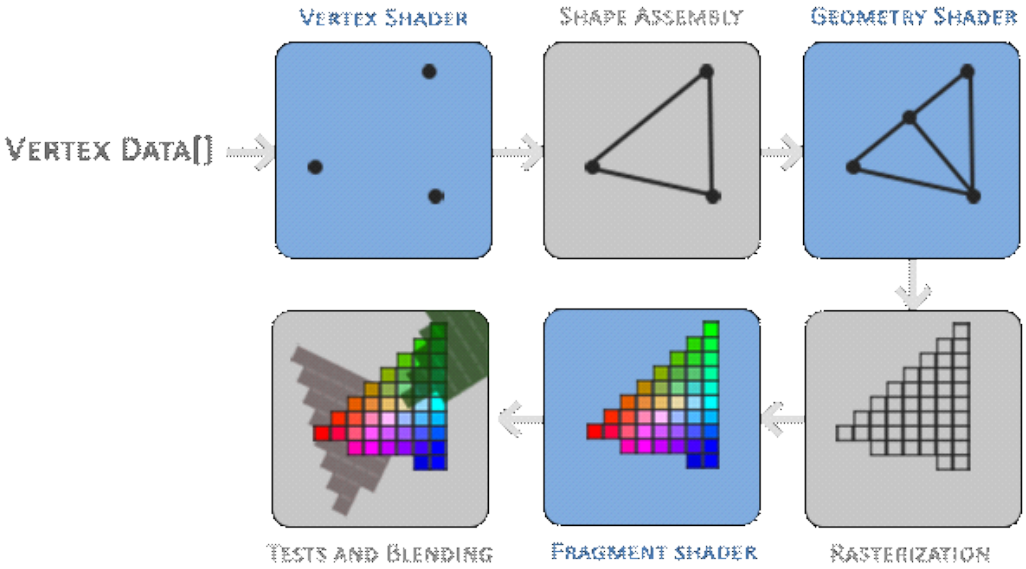


由上可知，在没有点击“X”的情况下，glfwWindowShouldClose(window) 返回false，不会打印“before loop”；而在点击了“X”之后，glfwWindowShouldClose(window)返回true，于是退出while循环，并且打印“after loop”

Pipeline and Shader

2020年11月1日 16:01

一个pipeline内有许多shader，一个shader负责一个具体的渲染步骤，经过所有的shader着色之后，一个pipeline就完成了对一个图像的渲染（加工）。下图中的整个过程就是在pipeline内完成的，或者说这就是一个pipeline，而每一个步骤就是一个shader。其中蓝色的就代表可以我们自定义的shader，其中最常见的是vertex shder 和 fragment shader。
(there are no default vertex/fragment shaders on the GPU



- Vertex shader 将顶点的三维坐标转化
- Assemble shader 根据指定的primitive shape组装顶点
- Geometry shader 将图形复杂化
- Rasterization shader 使图形光栅化，并且修剪多余的像素
- Fragment shader 给单个图形每一个像素确定颜色（根据颜色、不透明度、光照、阴影等）
- alpha test and blending shader 将多个图形堆叠在一起后最终每个像素的颜色

Vertex input（VBO）

Vertex shader需要的数据输入就是点坐标的集合，每一个点都是三维的坐标，且范围在（-1，1），称为normalized device coordinates。
我们通过管理一块名为 vertex buffer objects (VBO) 的内存来存放点坐标数据，而Vertex shader也从这块内存中读入数据。下图就是创建一个VBO到存入数据的全部过程。通过 glBindBuffer(GL_ARRAY_BUFFER, VBO) 绑定VBO，且之后所有对于类型为GL_ARRAY_BUFFER的buffer的操作都是对这个VBO的操作

```
unsigned int VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

linking vertex attribute

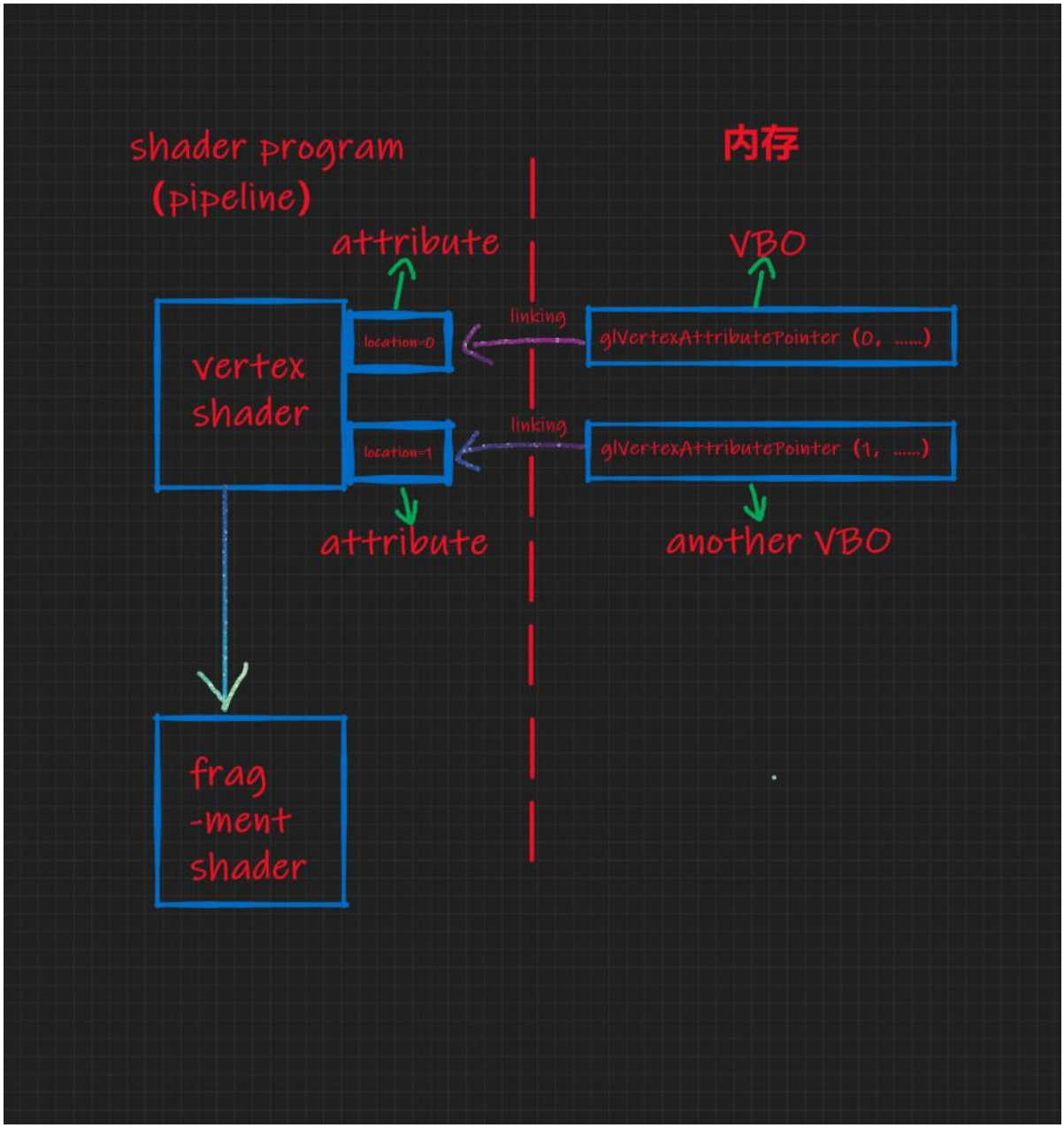
2020年11月3日 16:07

```
const char *vertexShaderSource = "#version 330 core\n"
    "layout (location = 0) in vec3 aPos;\n"
    "void main()\n"
    "{\n"
    "   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
    "}\0";
```

- 在shader source中，用in定义shader的输入变量，用out定义输出变量
- Vertex shader中，gl_Position为默认输出量，所以不用定义输出。
- 对于shader（program）这端来说，layout（location）=0指定了aPos（x，y，z）是第0组属性，用于存储点的位置（Position）。你也可以来一个layout（location=1）用来获取一个vec4颜色属性（RGBs），这样vec4就是第1组了；对于VBO这端来说，下图的第一个参数0指定了现在绑定的VBO（之前与GL_ARRAY_BUFFER绑定过的VBO）里面的vertex attribute是location为0的组，也就是说这里获取到的数据都会分配给上面的aPos。所以这一个步骤叫做 Linking Vertex Attributes

  ```
  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
  ```

- Linking vertex attribute 过程如下图



在告诉了opengl要将哪组参数作为输入后，还要激活一下VAO（参数是要激活的属性的location），因为VAO的默认状态是disabled，这样即使没有绑定vertex buffer，也不会报错。

```
glEnableVertexAttribArray(0);
```

VAO

2020年11月3日　　16:09

```
vao ---> | attrib[0] |--- vbo ---> vec3 array { {x,y,z}, ... }; //3d-坐标
         |-----------|
         | attrib[1] |--- vbo ---> vec2 array { {x,y}, ... };//whatever
         |-----------|
         |    ...    |
         |-----------|
         | attrib[k] |--- vbo ---> vec4 array { {x,y,z,w}, ... };
         ~~~~~~~~~~~
```

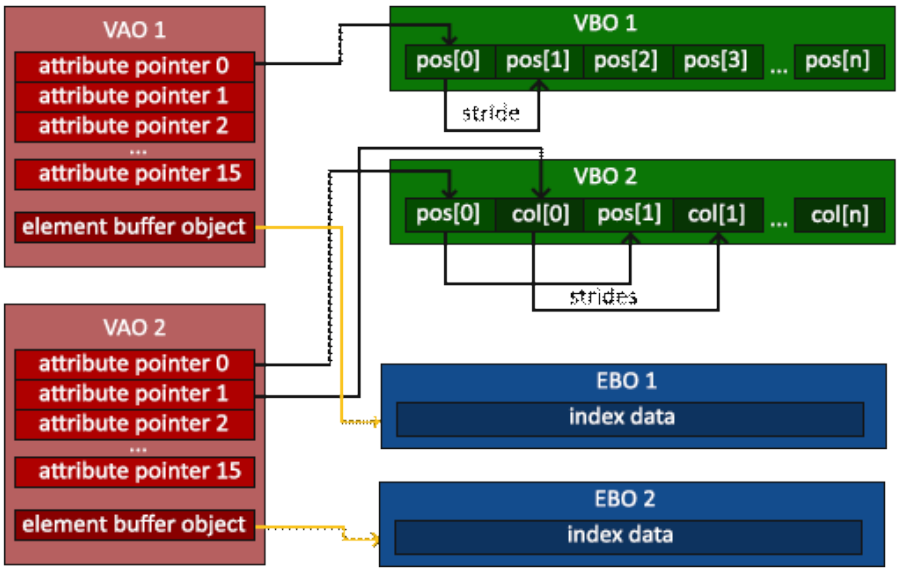
VAO中的每一个对象都指向了VBO中vertex属性的一种组合（like Position，RGBS，etc），VAO的下标就是之前提到的location。VAO还记录了VBO具体是哪一个（通过glBindBuffer（））渲染之前绑定VAO就行了，shader渲染的时候就会而不需要像之前那样，每次渲染点的Position属性还要用glVertexAttribPointer（）指定怎么用VBO中的数据。

```
glBindVertexArray(VAO);
```

这行代码有两个作用：

1、当在绑定VBO以及配置attribute pointer之前，它就等于启动VAO，让它记录接下来发生的事，即顶点数据在哪（bindVBO），顶点数据怎么组织和怎么传递给shader（glVertexAttribPointer（））还有顺序（bindEBO）

2、当在main函数中，glDrawElements调用之前，它就是用来激活指定的VAO。



★ VAO说白了就是有人想偷懒搞出来的东西。比方说你刚刚绘制了一百个点，然后VBO等数据池也没了（被覆盖了），现在又要绘制同样的一百个点，如果没有VAO，只有

VBO，那你就只能再写一遍从glBindBuffer到glVertexAttributePointer的一系列函数来告诉GPU，要从哪个VBO里面怎么拿数据。但是有了VAO，它里面就保存了你刚刚画那一百个点的操作，直接告诉GPU用这个VAO，就省去了告诉GPU从哪拿怎么拿的过程。所以VAO说到底就是配置文件一样的东西，配置之前先用glBindVertexArray开启VAO，VAO自动记录配置；画图之前要用哪个配置就用glBindVertexArray激活哪个VAO

Uniform And Attribute

2020年11月4日 14:14

GLSL是在GPU上运行的代码，于是就涉及到和CPU数据交换的问题。有两种和方法可以将数据从CPU运到GPU

1. 就是之前提到的vertex attribute，用layout（location=0）标识的顶点属性，存储在VBO中，GPU每次执行glDrawArrays（）时都会从VBO中读取顶点数据。所以想要在这种方式中动态改变顶点的属性，就要在更新了顶点属性的数值之后，再用glBufferData（）将数据复制到VBO中。
2. 第二种就是uniform。通过在GLSL中用关键字uniform定义的全局变量，在一个shader program的所有shader中都可以访问且唯一。这也是为什么，每一个uniform变量虽然也有一个location，但是它不用像vertex attribute那样，显式定义一个location=0之类的，而是可以直接在c语言（opengl端）用shader program+uniform变量名的组合访问到，如下：

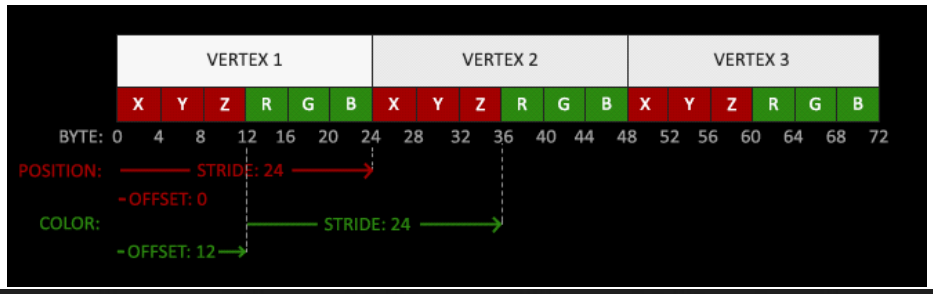
```
int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

- 由上可知，vertex attribute（in and out）和uniform这两种变量，都有location，只是前者需要显式地用数字0、1等定义，而后者可以直接通过shader program+uniform访问到。
- uniform变量是外部程序传递给（vertex和fragment）shader的变量。因此它是application通过函数glUniform**（）函数赋值的。在（vertex和fragment）shader程序内部，uniform变量就像是C语言里面的常量（const），它不能被shader程序修改。（shader只能用，不能改）
- attribute变量是只能在vertex shader中使用的变量。（它不能在fragment shader中声明attribute变量，也不能被fragment shader中使用）

- uniform变量一般用来表示：变换矩阵，材质，光照参数和颜色等信息。
- 一般用attribute变量来表示一些顶点的数据，如：顶点坐标，法线，纹理坐标，顶点颜色等。

More attributes

2020年11月5日 14:02




```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
```

当加入了第二组attribute：Color后，需要将其的location设为1，然后在vertex shader中定义这组输入时也要用layout（location=1）

交换缓冲区

2020年11月6日 10:15

GLFW在默认情况下使用两个缓冲区。 这意味着每个窗口有两个渲染缓冲区——前缓冲区和后缓冲区。 前缓冲区是正在显示的缓冲区，后缓冲区是即将显示的缓冲区。 当整个帧已经被渲染时，缓冲器需要彼此交换，因此后缓冲器变为前缓冲器，反之亦然。

```
glfwSwapBuffers(window);
```

OpenGL上下文

2020年11月6日 10:28

OpenGL。中，一个窗体对应一个上下文（context）。上下文在创建窗体（window）的时候同时被创建。使用hint那几个函数设置上下文属性。

`glfwMakeContextCurrent ()` 就是用来指定当前线程使用哪个窗口，也就是哪个上下文的。

一个上下文一次只能是单个线程的当前上下文，而一个线程一次只能有一个当前上下文

Mipmap

2020年11月7日 16:38

当texture的resolution大于要渲染物体（object）在屏幕上占的resolution时，就会产生纹理过大的问题，从而产生摩尔纹。
产生这种现象的原因就是，显示器，或者说openGL要渲染一个fragment，于是就要从texture中选择合适的点去采样，但是由于texture分辨率太高，一个fragment对应的点（footprint）有很多个，那就不知道该选哪个点了。从信号的角度来说就是，采样频率过低无法还原信号原貌。
产生这种现象的情况有很多，最常见的就是，远处的object，在屏幕上所占的像素很少，使用texture自然容易出现纹理过大的问题。
而Mipmap就是解决这个问题的工具。试想一下，既然问题是由texture分辨率过大造成的，那降低texture的分辨率不就行了。但是直接降低texture的分辨率，那别的物体（比如说近处的）要用这个texture不就gg了。所以要在保留高resolution的texture的前提下创建小分辨率的材质纹理，并且还是一组像素数不同（距离远近）的texture，这样一组texture就叫Mipmap。

Texture

2020年11月8日 15:37

在渲染之前绑定Texture

```
glBindTexture(GL_TEXTURE_2D, texture);
```

然后在fragment shader中通过内建变量sampler读取绑定的Texture，后缀2D对应

```
uniform sampler2D texture1;
```

最后用texture函数，参数为sampler读入的纹理和vertex shader传入的纹理坐标。注意如果第一个参数是sampler3D类型的（读取GL_TEXTURE_3D类型的纹理），那第二个参数就要传入一个vec3类型的坐标，毕竟3D纹理肯定是用三维坐标的

```
FragColor = texture(texture1, TexCoord);
```

在frame shader中，texture函数返回一个四维向量，所以可以和vec4相乘?

```
FragColor = texture(texture1, TexCoord) * vec4(ourColor,1.0);
```

我们可以同时拥有多个texture，要使用其中某一个texture时，只需要在bindTexture之前使用glActiveTexture激活某一个texture就行了，如下：

```
glActiveTexture(GL_TEXTURE0); // activate the texture unit first before binding texture
glBindTexture(GL_TEXTURE_2D, texture);
```

其中，常量GL_TEXTURE0~GL_TEXTURE15都是一系列连续的整型常量，所以可以通过GL_TEXTURE0+N来访问激活任意一个texture

texture filtering就是指，在选取texture中的点时，如果遇到这个坐标对应几个像素点的时候，该如何决定究竟用哪个像素点。

Fragment shader

2020年11月8日 18:41

Fragment shader 默认的输出FragColor只是一个Vec4向量，只能代表一个像素的RGBA值，所以GPU中会使用多个核来并行执行多个frame shader程序，一次性渲染出多个像素点。假设有64个像素点要渲染，一个gpu同时能跑32个核，那就要跑两遍才能把所有的像素点渲染完。

Fragment shader接收的in变量直接来自于vertex shader的同名out变量，跳过了中间的其他shader