

2. HELLO TRIANGLE

图形渲染管线 (GRAPHICS PIPELINE)

OpenGL中，任何事物都在3D空间中，
而屏幕和窗口却是2D像素数组，
这导致OpenGL的大部分工作都是关于把3D坐标转变为适应你屏幕的2D像素
3D坐标转为2D坐标的处理过程是由OpenGL的图形渲染管线管理的

指的是一堆原始图形数据途经一个输送管道，期间经过各种变化处理最终出现在屏幕的过程

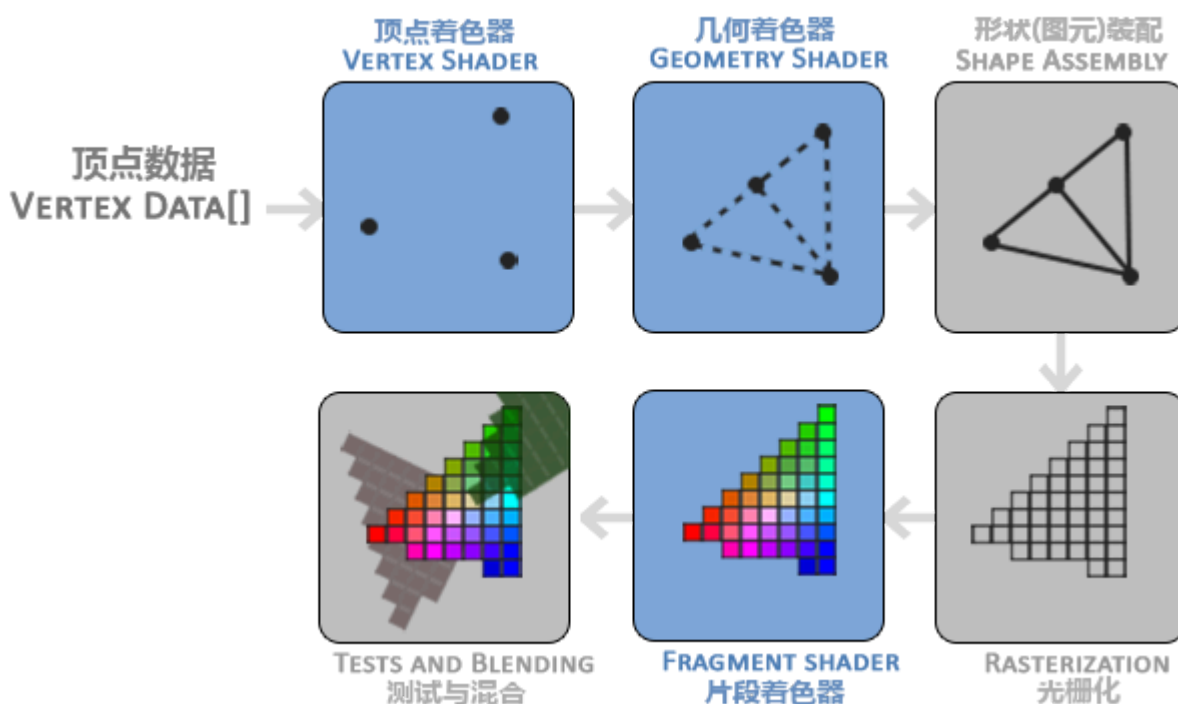
图形渲染管线可以被划分为两个主要部分：

- 第一部分把3D坐标转换为2D坐标，
- 第二部分是把2D坐标转变为实际的有颜色的像素。
- 输入：一组3D坐标
- 输出：屏幕上的有色2D像素输出

! 2 D 坐标 和 2 D 像素

2D坐标和像素也是不同的，

- 2D坐标精确表示一个点在2D空间中的位置，
- 2D像素是这个点的近似值，
2D像素受到你的屏幕/窗口分辨率的限制。



0. 顶点输入

OpenGL仅当3D坐标在3个轴（x、y和z）上-1.0到1.0的范围内时才处理它

- 称为**标准化设备坐标(Normalized Device Coordinates),NDC**
 - 此范围内的坐标最终显示在屏幕上
 - 在这个范围以外的坐标则不会显示

渲染一个三角形，我们一共要指定三个顶点，每个顶点都有一个3D位置。

我们会将它们以**标准化设备坐标的形式**（OpenGL的可见区域）定义为一个 `float` 数组

```
float vertices[] = {  
    -0.5f, -0.5f, 0.0f,  
    0.5f, -0.5f, 0.0f,  
    0.0f, 0.5f, 0.0f  
};
```

定义这样的顶点数据以后，我们会把它作为输入发送给图形渲染管线的第一个处理阶段：**顶点着色器**。

它会在GPU上创建内存用于储存我们的顶点数据，还要配置OpenGL如何解释这些内存，并且指定其如何发送给显卡。

顶点缓冲对象 VBO

我们通过**顶点缓冲对象(Vertex Buffer Objects, VBO)**管理这个内存，会在**GPU内存（通常被称为显存）**中储存大量顶点。

使用这些缓冲对象的好处是

- 可以一次性的发送一大批数据到显卡上，
- 而不是每个顶点发送一次。
- 从CPU把数据发送到显卡相对较慢，所以只要可能我们都要尝试尽量一次性发送尽可能多的数据。
- 当数据发送至显卡的内存中后，**顶点着色器几乎能立即访问顶点**，这是个非常快的过程。

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

`glGenBuffers` 生成一个OpenGL对象，带有缓冲ID

缓冲

OpenGL有很多缓冲对象类型，

顶点缓冲对象的缓冲类型是 `GL_ARRAY_BUFFER`

OpenGL允许我们同时绑定多个缓冲，只要它们是不同的缓冲类型

我们可以使用 `glBindBuffer` 函数把新创建的缓冲绑定到 `GL_ARRAY_BUFFER` 目标上

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

从这一刻起，我们使用的任何（在 `GL_ARRAY_BUFFER` 目标上的）缓冲调用都会用来配置当前绑定的缓冲(VBO)。

我们可以调用 `glBufferData` 函数，它会把之前定义的顶点数据复制到缓冲的内存中

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

- `glBufferData` 是一个专门用来把用户定义的数据复制到当前绑定缓冲的函数。它的第一个参数是**目标缓冲的类型**：顶点缓冲对象当前绑定到 `GL_ARRAY_BUFFER` 目标上。
- 第二个参数指定**传输数据的大小(以字节为单位)**；用一个简单的 `sizeof` 计算出顶点数据大小就行。
- 第三个参数是**我们希望发送的实际数据**。
- 第四个参数指定了**我们希望显卡如何管理给定的数据**。它有三种形式：
 - `GL_STATIC_DRAW`：数据不会或几乎不会改变。
 - `GL_DYNAMIC_DRAW`：数据会被改变很多。
 - `GL_STREAM_DRAW`：数据每次绘制时都会改变。

三角形的位置数据不会改变，每次渲染调用时都保持原样，所以它的使用类型最好是 `GL_STATIC_DRAW`。

如果，比如说一个缓冲中的数据**将频繁被改变**，那么使用的类型就是 `GL_DYNAMIC_DRAW` 或 `GL_STREAM_DRAW`，这样就能确保显卡把数据放在能够**高速写入的内存部分**。

现在我们已经把**顶点数据储存在显卡的内存中**，用VBO这个顶点缓冲对象管理。

1. 顶点着色器

它会在GPU上创建内存用于储存我们的顶点数据，还要配置OpenGL如何解释这些内存，并且指定其如何发送给显卡。

用着色器语言 **GLSL** (OpenGL Shading Language) 编写顶点着色器，然后编译这个着色器，这样我们就可以在程序中使用它了

```
#version 330 core
layout (location = 0) in vec3 aPos;

void main() {
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

- 每个着色器都**起始于一个版本声明**。

- OpenGL 3.3以及和更高版本中，GLSL版本号和OpenGL的版本是匹配的（比如说GLSL 420版本对应于OpenGL 4.2）。
- 明确表示我们会使用**核心模式**。
- 使用 `in` 关键字，在**顶点着色器中声明所有的输入顶点属性(Input Vertex Attribute)**。
 - 现在我们只关心位置(Position)数据，所以我们只需要一个顶点属性。
 - GLSL有一个**向量数据类型**，它包含**1到4个 float 分量**，包含的数量可以从它的后缀数字看出来，vec3即为 3 个float分量的向量
 - 由于每个顶点都有一个3D坐标，我们就创建一个 `vec3` 输入变量aPos。
 - 我们同样也通过 `layout (location = 0)` 设定了**输入变量的位置值(Location)**。

为了设置顶点**着色器的输出**，我们必须把位置数据赋值给**预定义的gl_Position变量**

- 它在幕后是 `vec4` 类型的
- 在main函数的最后，我们将gl_Position设置的值会成为该顶点着色器的输出。
- 由于我们的输入是一个3分量的向量，我们必须把它转换为4分量的。
 - 我们可以把 `vec3` 的数据作为 `vec4` 构造器的参数，同时把 `w` 分量设置为 `1.0f` 来完成这一任务。

编译着色器

我们首先要做的是**创建一个着色器对象**，注意还是**用ID来引用的**。

所以我们储存这个顶点着色器为 `unsigned int (id)`，然后用 `glCreateShader` 创建这个着色器：

```
unsigned int vertexShader;
vertexShader = glCreateShader(GL_VERTEX_SHADER);
```

我们把需要创建的着色器类型以**参数形式提供给glCreateShader**。

由于我们正在创建一个**顶点着色器**，传递的参数是 `GL_VERTEX_SHADER`。

下一步我们把这个**着色器源码附加到着色器对象上**，然后编译它：

```
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
```

`glShaderSource` 函数参数

- 编译的着色器对象作为第一个参数。
- 第二参数指定了传递的源码字符串数量，这里只有一个。
- 第三个参数是顶点着色器真正的源码，
- 第四个参数我们先设置为 `NULL`。

2. 片段着色器

片段着色器所做的是计算像素最后的颜色输出。

```
#version 330 core
out vec4 FragColor;
void main() {
    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}
```

- 片段着色器只需要一个输出变量，
 - 这个变量是一个4分量向量，
 - 它表示的是最终的输出颜色，
 - 我们应该自己将其计算出来。
- 声明输出变量可以使用 `out` 关键字，这里我们命名为FragColor。
- 1.0f, 0.5f, 0.2f, 1.0f分别表示RGBA的完全不透明橘黄色的 `vec4` 赋值给颜色输出。

创建和编译片段着色器的过程和顶点着色器完全类似，仅仅修改创建的着色器类型

```
unsigned int fragmentShader;
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);
```

3. 连接着色器（生成着色器程序）

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。

当链接着色器至一个程序的时候，它会把每个着色器的输出链接到下个着色器的输入。当输出和输入不匹配的时候，你会得到一个连接错误。

创建程序

```
unsigned int shaderProgram;
shaderProgram = glCreateProgram();
```

`glCreateProgram` 函数创建一个程序，并返回新创建程序对象的ID引用。

附加着色器到程序上，并连接

把之前编译的着色器用 `glAttachShader` 附加到程序对象上，然后用 `glLinkProgram` 链接它们

```
glAttachShader(shaderProgram, vertexShader); glAttachShader(shaderProgram,
fragmentShader); glLinkProgram(shaderProgram);
```

得到的结果就是一个程序对象。

激活程序

我们调用 `glUseProgram` 函数，用刚创建的程序对象作为它的参数，以激活这个程序对象：

```
glUseProgram(shaderProgram);
```

在 `glUseProgram` 函数调用之后，每个着色器调用和渲染调用都会使用这个程序对象（也就是之前写的着色器）了。

连接后删除着色器

```
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

连接完成之后不再需要着色器，故删除

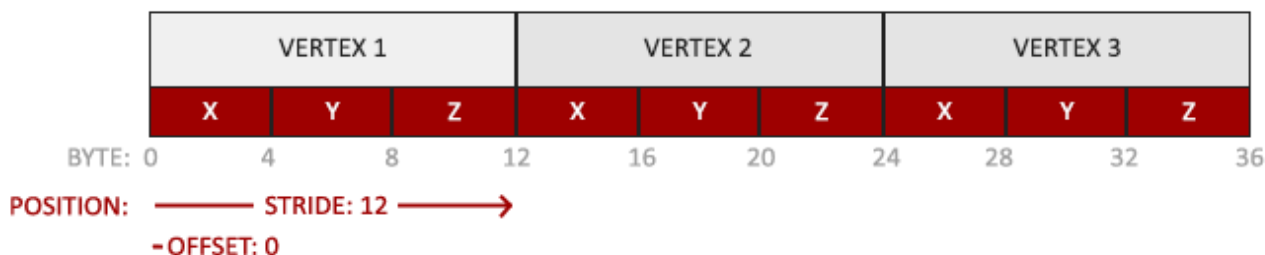
4. 链接顶点属性

当前进度：

- ☒ 把输入顶点数据发送给了GPU
- ☒ 指示了GPU如何在顶点和片段着色器中处理它。
- ☐ OpenGL还不知道它该如何解释内存中的顶点数据，
- ☐ 该如何将顶点数据链接到顶点着色器的属性上。

解析顶点数据 VBO

VBO展示如下



- 位置数据被储存为**32位**（4字节）浮点值。
- 每个位置包含**3个这样的值**。
- 在这**3个值之间没有空隙**（或其他值）。这几个值在数组中紧密排列(Tightly Packed)。

- 数据中**第一个值在缓冲开始的位置**。

使用 `glVertexAttribPointer` 函数告诉OpenGL该如何解析顶点数据（应用到逐个顶点属性上）

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

`glVertexAttribPointer` 函数的参数非常多

- 第一个参数指定我们要**配置的顶点属性**。
 - 我们在顶点着色器中使用 `layout(location = 0)` 定义了position顶点属性的位置值 (Location)，它可以把顶点属性的位置值设置为 0。
 - 因为我们希望把数据传递到这一个顶点属性中，所以这里我们传入 0。
 - 类似于利用ID在着色器中对应
- 第二个参数指定**顶点属性的大小**。
 - 顶点属性是一个 `vec3`，它由**3个值组成**，所以大小是3。
- 第三个参数**指定数据的类型**
 - 这里是 `GL_FLOAT` (GLSL中 `vec*` 都是由浮点数值组成的)。
- 下个参数定义**我们是否希望数据被标准化(Normalize)**。
 - 如果我们设置为 `GL_TRUE`，所有数据都会被映射到0（对于有符号型signed数据是-1）到1之间。
 - 我们把它设置为 `GL_FALSE`。
- 第五个参数叫做**步长(Stride)**
 - 从这个**属性第二次出现的地方到整个数组0位置之间有多少字节**
 - 由于下个组位置数据在3个 `float` 之后，我们把步长设置为 `3 * sizeof(float)`。
 - 由于我们知道这个数组是紧密排列的（在两个顶点属性之间没有空隙）
 - 我们也可以设置为0来让OpenGL决定具体步长是多少（**只有当数值是紧密排列时才可用**）。
 - 一旦我们有更多的顶点属性，我们就必须更小心地定义每个顶点属性之间的间隔
- 最后一个参数的**类型是 `void*`**，所以需要我们进行这个奇怪的强制类型转换。它表示**位置数据在缓冲中起始位置的偏移量**。
 - 由于位置数据在数组的开头，所以这里是0。

VBO 绑定对应

每个**顶点属性**从一个**VBO管理的内存中获得它的数据**，而具体是从哪个VBO（程序中可以有多个VBO）获取则是通过在调用 `glVertexAttribPointer` 时绑定到 `GL_ARRAY_BUFFER` 的VBO决定的。

由于在调用 `glVertexAttribPointer` 之前绑定的是先前定义的VBO对象，顶点属性 0 现在会链接到它的顶点数据。

当前进度：

- ☒ 把输入顶点数据发送给了GPU
- ☒ 指示了GPU如何在顶点和片段着色器中处理它。
- ☒ OpenGL还不知道它该如何解释内存中的顶点数据，~~←【HERE】~~
- ☐ 该如何将顶点数据链接到顶点着色器的属性上。

配置顶点属性 VAO

现在我们已经定义了OpenGL该如何解释顶点数据，我们现在应该使用

`glEnableVertexAttribArray`，以顶点属性位置值作为参数，启用顶点属性；

但告诉OpenGL如何把顶点数据链接到顶点着色器的顶点属性上时，**每当我们绘制一个物体的时候都必须重复【配置顶点属性】这个过程**，所以必须寻找快速绑定和恢复的方式

顶点数组对象(Vertex Array Object, VAO)可以**像顶点缓冲对象那样被绑定**，任何随后的顶点属性调用都会储存在这个VAO中。

这样的好处就是当**配置顶点属性指针**时，将那些调用执行一次，之后再绘制物体的时候只需要**绑定相应的VAO**就行了。

这使在不同顶点数据和属性配置之间切换变得非常简单，只需要绑定不同的VAO就行了。

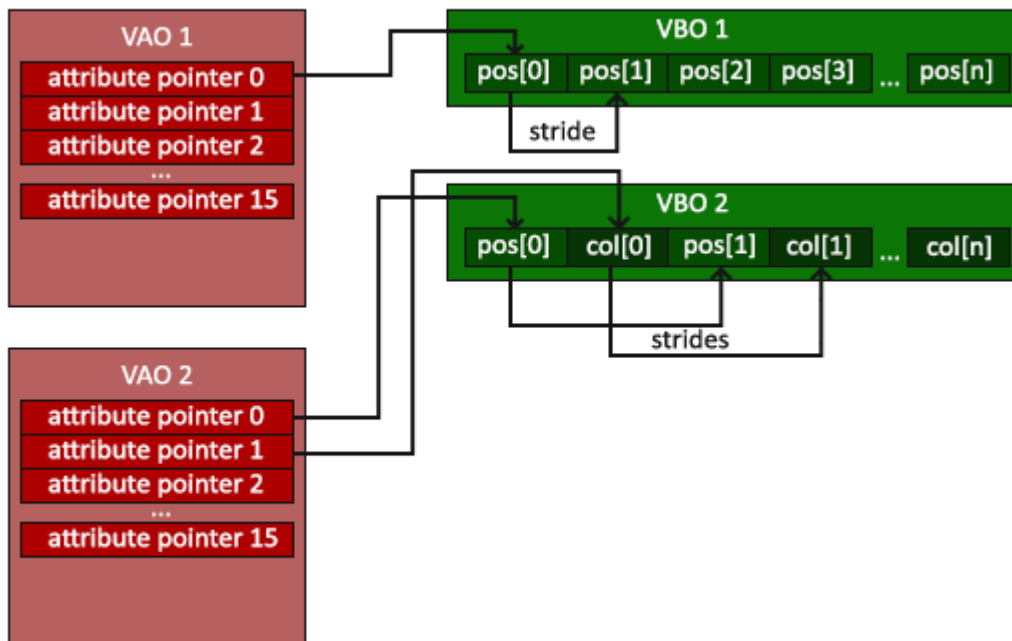
! OpenGL的核心模式**要求**我们使用VAO，所以它知道该如何处理我们的顶点输入。如果我们绑定VAO失败，OpenGL会拒绝绘制任何东西。

VAO展示如下：

一个**顶点数组对象**会储存以下内容：

- `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。



创建VAO与创建VBO非常类似

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

使用VAO仅需要

- 用 `glBindVertexArray` 绑定VAO
- 配置对应的VBO和属性指针

```
// ...: 初始化代码（只运行一次（除非你的物体频繁改变）） :: ...  
  
// 1. 绑定VAO  
glBindVertexArray(VAO);  
  
// 2. 把顶点数组复制到缓冲中供OpenGL使用  
glBindBuffer(GL_ARRAY_BUFFER, VBO);  
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);  
  
// 3. 设置顶点属性指针  
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),  
(void*)0);  
glEnableVertexAttribArray(0);  
  
// ...: 绘制代码（渲染循环中） :: ...  
  
// 4. 绘制物体
```

```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
someOpenGLFunctionThatDrawsOurTriangle();
```

一般当你打算绘制多个物体时，你首先要生成/配置所有的VAO（和必须的VBO及属性指针），然后储存它们供后面使用。

当我们打算绘制物体的时候就拿出相应的VAO，绑定它，绘制完物体后，再解绑VAO。

5. 绘制三角形

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```

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。
- 第二个参数指定了顶点数组的起始索引，我们这里填0。
- 最后一个参数指定我们打算绘制多少个顶点，这里是3（我们只从我们的数据中渲染一个三角形，它只有3个顶点长）。

（6.）元素缓冲对象EBO

假设我们需要绘制一个矩形，用两个三角形方式绘制

```
float vertices[] = {
    // 第一个三角形
    0.5f, 0.5f, 0.0f, // 右上角
    0.5f, -0.5f, 0.0f, // 右下角
    -0.5f, 0.5f, 0.0f, // 左上角
    // 第二个三角形
    0.5f, -0.5f, 0.0f, // 右下角
    -0.5f, -0.5f, 0.0f, // 左下角
    -0.5f, 0.5f, 0.0f // 左上角
};
```

- 有几个顶点叠加了。我们指定了右下角和左上角两次
- 一个矩形只有4个而不是6个顶点，这样就产生50%的额外开销

更好的解决方案是只储存不同的顶点，并设定绘制这些顶点的顺序。

这样子我们只要储存4个顶点就能绘制矩形了，之后只要指定绘制的顺序就行了。

EBO是一个缓冲区，就像一个顶点缓冲区对象一样，它存储 OpenGL 用来决定要绘制哪些顶点的【索引】。

首先，我们先要定义**不重复的顶点**，和绘制出矩形所需的**索引**

...

```
float vertices[] =
{
    0.5f, 0.5f, 0.0f, // 右上角
    0.5f, -0.5f, 0.0f, // 右下角
    -0.5f, -0.5f, 0.0f, // 左下角
    -0.5f, 0.5f, 0.0f // 左上角
};

unsigned int indices[] = {
    // 注意索引从0开始!
    // 此例的索引(0,1,2,3)就是顶点数组vertices的下标,
    // 这样可以由下标代表顶点组合成矩形
    0, 1, 3, // 第一个三角形
    1, 2, 3 // 第二个三角形
};
...
```

只需定义4个顶点

创建EBO:

```
unsigned int EBO;
glGenBuffers(1, &EBO);
```

与VBO类似，我们需要

- 绑定EBO
- 用glBufferData把索引复制到缓冲里。
- 但和之前不同的是，缓冲的类型定义为GL_ELEMENT_ARRAY_BUFFER。

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

最后一件要做的事是用glDrawElements来替换glDrawArrays函数，表示我们要从索引缓冲区渲染三角形。

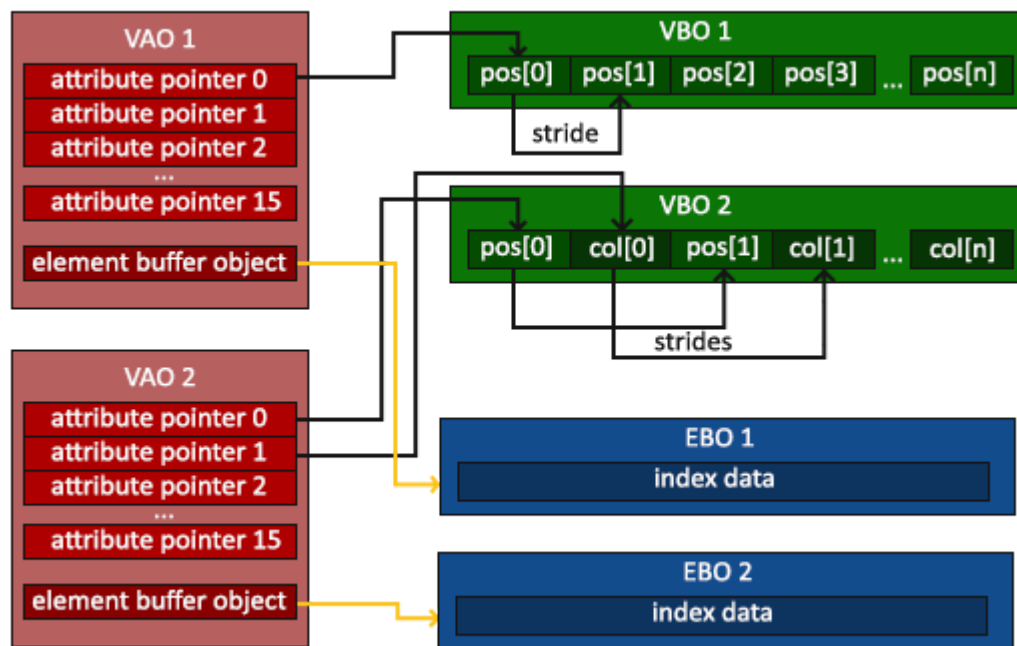
使用glDrawElements时，会使用当前绑定的索引缓冲对象中的索引进行绘制：

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO); glDrawElements(GL_TRIANGLES, 6,
GL_UNSIGNED_INT, 0);
```

`glDrawElements` 参数：

- 第一个参数指定了**我们绘制的模式（图元）**，这个和glDrawArrays的一样。
- 第二个参数是**打算绘制顶点的个数**，这里填6。
- 第三个参数是**索引的类型**，这里是GL_UNSIGNED_INT。
- 最后一个参数里我们可以指定**EBO中的偏移量**（或者传递一个索引数组，但是这是当你不在使用索引缓冲对象的时候），但是我们会在这里填写0。

glDrawElements函数从当前绑定到GL_ELEMENT_ARRAY_BUFFER目标的EBO中获取其索引。这意味着**我们每次想要使用索引渲染对象时都必须绑定相应的EBO**，这又有点麻烦。碰巧**顶点数组对象也可以进行元素缓冲区对象绑定**，VAO的最后一个绑定对象为元素缓冲区对象



最后的初始化和绘制代码如下：

```
// ...: 初始化代码 :: ...

// 1. 绑定顶点数组对象
glBindVertexArray(VAO);
// 2. 把我们的顶点数组复制到一个顶点缓冲中，供OpenGL使用
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. 复制我们的索引数组到一个索引缓冲中，供OpenGL使用
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);
// 4. 设定顶点属性指针 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 *
sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

[...]

// ...: 绘制代码（渲染循环中） :: ...
```

```
glUseProgram(shaderProgram); glBindVertexArray(VAO);  
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);  
glBindVertexArray(0);
```

绘制模式

通过 `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)` 函数配置OpenGL如何绘制图元。

- 第一个参数表示我们打算将其应用到所有的三角形的正面和背面
 - 第二个参数告诉我们用线来绘制。
- 之后的绘制调用会一直以线框模式绘制三角形，直到我们用 `glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)` 将其设置回**默认模式**。