

Jaxin's Cookbook

Jaxin, TONG

2018 年 12 月 4 日

Chapter 1 Rendering Pipeline

1.1 Talk about vertex processing matrices

在计算机架构中，管线通常意味着数据处理各个元部件间元数据的处理与衔接；对于计算机图形学来说，其渲染显示过程需要这样一套流水线，其以由大量多边形构成的 3D 图形作为输入端，对众多多边形进行处理将其转换为 2D 图形并计算每个多边形像素的颜色值，这样一来，渲染管线（*rendering pipeline*）这一名词诞生了。渲染管线可以将现实世界中的物体经由 GPU 的元数据处理零件进行流水作业将其变换到 OpenGL 的世界中去，在这一过程中，我们需要暂时进入数学领域，从矩阵的角度描述一个物体到底是怎样经由顶点、光栅化、片元化处理成为最终在计算机光栅扫描式图形显示器上所显示的图像的。

1.1.1 顶点处理：4 spaces +3 matrixes

计算机图形开发的过程与摄影很相似，目的就是把真实的场景及场景的动态变化在一帧影像上展示出来。模型变换和视图变换的过程就像是我们摆设相机的位置，选择好要照的物体，摆好物体的造型；投影变换就像相机把真实的三维场景显示在具体的一帧影像上。我们拿 OpenGL 的茶壶为例，在这个茶壶从创建到显示在屏幕中形成一帧图像之前的一个生命周期中，一共游历了 4 个空间：即对象空间、世界空间、摄像机空间和裁切面空间。

对象空间和世界空间 首先是茶壶创建时所采用的坐标系，称之为对象坐标系，可以假设茶壶的中心点坐标为 $(0, 0, 0)$ ，那么由这个中心点所创建的坐标系就是对象坐标系，它构成了对象空间（*Object Space* 或 *Model Space*）。接下来，我们需要将茶壶放桌子上，茶壶的中心点在放到桌子上时所经历的必要的平移、旋转和缩放变换，都是在世界坐标系中进行的，世界坐标系代表了世界空间（*World Space*）。

摄像机空间 到目前为止，整个变换过程还是处于一个内部变化，我们暂时还无法知道茶壶长什么样，它究竟放在桌子上的哪个位置，为了观察这个茶壶，需要在世界空间中放置一台摄像机，用它来观察这个茶壶，这样由摄像机的角度出发，由摄像机坐标系形成了摄像机空间（*Camera Space*）。摄像机的坐标系就与前面的两个坐标系有明显的区别了，它的坐标通常由摄像机中心点（EYE），摄像机倾向（UP）和摄像机观察方向（AT）这样三个矢量通过简单的数学变换得到，*JungHyun Han* 的书《*3D Graphics for Game Programming*》中详细描述了这个数学变换过程。

裁切面空间 最后就是图形渲染管线光栅化（*Rasterization*）和片元处理（*Fragment Processing*）之前的一个尤为重要的坐标系，裁切面坐标系以及由此形成的裁切面空间

光栅扫描式图形显示器

光栅扫描式图形显示器可看作是一个点阵单元发生器，其工作过程是电子束受偏转部件的控制不断从左到右、从上到下将图像逐行逐点扫描到显示屏并通过控制电子束强弱来产生灰度或彩色图像；显示屏显示面扫描线为 2^n 行，每行可以分为 2^m 个小点，每个小点被称为像素。

显示驱动卡

在图形显示卡上都有一个由视频存储器组成的显示缓冲区，它接受并暂存计算机送来的图形图像数字信息，经数模转换器转换为模拟信号后，再送到显示器去显示；显卡主要由：显示芯片 GPU、显示存储器（VRAM, *Video Random Access Memory*）、显卡基本输入输出系统 BIOS、视频（显存通道随机）数模转换器 RAMDAC 等构成，RAMDAC 决定了显卡刷新频率的高低，如图 1.1 所示。



Fig. 1.1 显卡的硬件结构

(*Clip Space*), 它可以通过投影矩阵 (*Projection Matrix*) 所定义的视锥体 (*View Frustum*) 中所蕴含的空间转换关系经过数学计算得到。

在这四个空间中对应形成的 **3 个**转换矩阵分别记为模型矩阵 (*Model Matrix*)、视图矩阵 (*View Matrix*) 和投影矩阵 (*Projection Matrix*), 如图 1.2 所示。

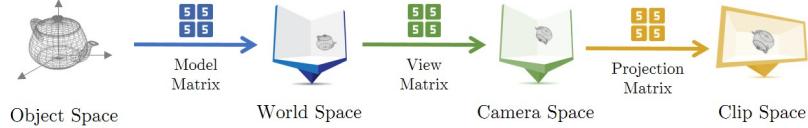


Fig. 1.2 渲染管线中的矩阵变换

当然除了这 3 个顶点变换矩阵之外, 还有一个视口变换, 被称为 *Viewport Mapping*, 这一变换的目的是为了将规范化的窗口坐标转换为实际显示在屏幕上的视口坐标, 这种转换会对结合视口大小对视景体进行调整, 图像的大小会根据视口产生相应的变形, 这一部分列入第 1.1.4 节中。

1.1.2 模型观测变换: ModelView Matrix

OpenGL 渲染管线分为两大部分, 模型观测变换 (*ModelView Transformation*) 和投影变换 (*Projection Transformation*), 本节首先介绍模型观测变换。

模型观测变换的主要执行动态是将物体由对象空间变换到世界空间, 再由世界空间变换到摄像机空间, 这一过程的数学描述分别为模型矩阵和视图矩阵, 如公式 1.1 二者相乘即得到模型观测变换矩阵。

齐次坐标

齐次坐标就是将一个原本是 n 维的向量用一个 $n+1$ 维向量来表示, 它使得我们可以用同一个公式对点和方向作运算。目前为止, 我们仍然把三维顶点视为三元组 (x, y, z) , 引入一个新的分量 w , 得到向量 (x, y, z, w) 。若 $w=1$, 则向量 $(x, y, z, 1)$ 为空间中的点; 若 $w=0$, 则向量 $(x, y, z, 0)$ 为方向。

$$(x \ y \ z \ w)_{camera}^T = M_{modelview} \cdot (x \ y \ z \ w)_{object}^T = M_{view} \cdot M_{model} \cdot (x \ y \ z \ w)_{object}^T \quad (1.1)$$

注意上面公式中对象空间和摄像机空间中的坐标都是齐次坐标的形式; 在平面视觉中, 齐次坐标的引入可以解决平行线交点的问题, 通过非齐次坐标 (x, y) 的齐次表示 (x_1, x_2, x_3) (其中 $x = x_1/x_3$, $y = x_2/x_3$), 通过这种表示, 在两平行直线的交点求解过程中忽略交点坐标齐次表示中的标量, 将交点的 x_3 坐标置 0, 则可以将交点描述为无穷远处的一个点, 这一部分可以参考 *Richard Hartley* 的书《*Multiple View Geometry in Computer Vision (Second Edition)*》2.2 节第 27 页。

(1) Model Matrix

模型矩阵主要包括三个部分, 即平移矩阵、缩放矩阵和旋转矩阵。其中平移矩阵最为简单, 其形式如 1.2 所示:

$$M_T = \begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1.2)$$

式中, dx, dy, dz 分别为点沿 x, y, z 轴的位移量。

缩放矩阵的形式也并不复杂, 令 s_x, s_y, s_z 分别为向量沿 x, y, z 轴的缩放量, 则缩放矩

阵的形式如 1.3 所示:

$$M_S = \begin{pmatrix} s_x & 0 & 0 & 1 \\ 0 & s_y & 0 & 1 \\ 0 & 0 & s_z & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1.3)$$

旋转的描述稍复杂些, 通常来说其可以有欧拉角、旋转矩阵和四元数等几种方式进行描述: 欧拉角在 3D 系统中对应两种形式, 即 “heading-pitch-bank” (朝向 -俯仰 -倾斜) 序列和 “roll-pitch-row” (横滚 -俯仰 -偏航) 序列, 二者描述的是同一种旋转系统; 旋转矩阵即为向量分别沿着 x, y, z 轴的旋转参量的乘积; 四元数则是给定一个单位长度的旋转轴 (x, y, z) 和一个角度 θ , 对应生成四元数 $\mathbf{q} = ((x, y, z) \cdot \sin(\theta/2), \cos(\theta/2))^T$, 这种旋转描述方式可以解决万向节锁问题, 但比较抽象不好理解; 关于旋转的理解可以参考 *Timothy D. Barfoot* 的书《State Estimation for Robotics》6.2 节第 176 页。一般我们还是用旋转矩阵来描述旋转, 如式 1.4 所示:

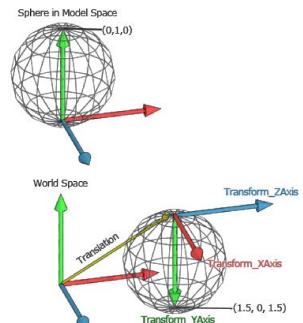
$$\begin{aligned} R &= R_x \cdot R_y \cdot R_z \\ &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta_1 & -\sin \theta_1 \\ 0 & \sin \theta_1 & \cos \theta_1 \end{pmatrix} \cdot \begin{pmatrix} \cos \theta_2 & 0 & \sin \theta_2 \\ 0 & 1 & 0 \\ -\sin \theta_2 & 0 & \cos \theta_2 \end{pmatrix} \cdot \begin{pmatrix} \cos \theta_3 & \sin \theta_3 & 0 \\ -\sin \theta_3 & \cos \theta_3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{aligned} \quad (1.4)$$

将式 1.4 中的矩阵 R 写成齐次坐标 M_R 的形式则有:

$$M_R = \begin{pmatrix} R & 0 \\ 0 & 1 \end{pmatrix} \quad (1.5)$$

将 M_T, M_S, M_R 三个矩阵进行组合即可将某一指定向量由对象空间变换到世界空间。如假设对象空间 (又称模型空间, *Model Space*) 中一三维点 $(0, 1, 0)$ 上的一个球绕 Y 轴旋转 90°, 然后绕 X 轴旋转 180°, 最后平移到点 $(1.5, 1, 1.5)$, 则在这一过程中:

$$\begin{aligned} M_{model} &= M_T \cdot M_{R_x(180^\circ)} \cdot M_{R_y(90^\circ)} \\ &= \begin{pmatrix} 1 & 0 & 0 & 1.5 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 0 & 1 & 1.5 \\ 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & 1.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned}$$



最终球所在位置由 $(0, 1, 0)$ 变换到 $(1.5, 0, 1.5)$, 如图 1.3 所示:

$$M_{model} \cdot (0 \ 1 \ 0 \ 1)^T = (1.5 \ 0 \ 1.5 \ 1)^T$$

Fig. 1.3 从对象空间变换到世界空间中的球 (引自: *World, View and Projection Transformation Matrices*)

(2) View Matrix

视觉矩阵用于直接将世界坐标系下的坐标转换到摄像机坐标系下, 已知摄像机的坐标系和相机在世界空间下的坐标, 就可以求出视觉矩阵。令 uvn 为摄像机坐标系下的三个基,

对于一个摄像机来说，它在开始的时候和世界坐标系是重合的，用户控制摄像机在世界空间中移动之后，摄像机的状态可以用两个属性来描述，即朝向 (AT) 和位置 (EYE)。也就是说，有了这两个属性，一个摄像机模型在世界中的状态就确定了，而这两个属性对应的数学概念就是旋转和平移。可以想象，对于世界中的任何一个摄像机状态，我们都可以把它看成是：摄像机先围绕自身基原点旋转一定的角度，然后平移到世界空间的某个地方。图 1.4 展示了这个过程：

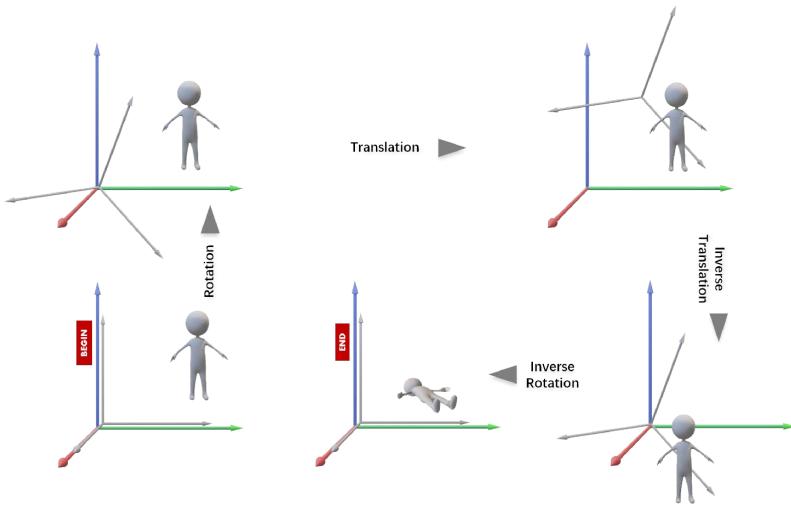


Fig. 1.4 摄像机的变换与逆变换 (引自：详解 MVP 矩阵之 ViewMatrix)

图 1.4 中，红色坐标轴是摄像机坐标系的基，而黑色是世界坐标系的基，也就是参考系。小人是世界中的一个物体。相机在移动之前，两个基是重合的。当摄像机在屏幕中定位时，它首先会进行朝向的确定——旋转，然后进行位置的确定——平移。图中的 Rotation 和 Translation 两步就是摄像机定位时所发生的变换，可以看到在这两步过程中摄像机相对于小人的运动。

而当进行摄像机变换的时候，小人应该从世界基变换到相机的基里面。因而需要进行摄像机定位的逆定位，即先逆平移小人和摄像机，然后再逆旋转小人和摄像机，最后摄像机归位，小人随摄像机变到了摄像机空间。这是由 Inverse Translation 和 Inverse Rotation 两个步骤完成的，这两个步骤合起来就是摄像机变换。

JungHyun Han 的书中 37 页对上面的过程进行了精简，以 LookAt 摄像机系统作为视图矩阵的推导载体，直接考虑将世界坐标系变换到摄像机坐标系中，这样世界坐标系首先根据 EYE 点反向平移，再利用由世界坐标系 $\{\mathbf{O}, e_1, e_2, e_3\}$ 到摄像机坐标系 $\{\mathbf{EYE}, u, v, n\}$ 的基变换进行旋转，从而得到我们所需要的视觉矩阵。

在这种情况下，视图矩阵可以描述为：

$$\begin{aligned}
 M_{view} &= R \cdot T \\
 &= \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -\mathbf{EYE}_x \\ 0 & 1 & 0 & -\mathbf{EYE}_y \\ 0 & 0 & 1 & -\mathbf{EYE}_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1.6)
 \end{aligned}$$

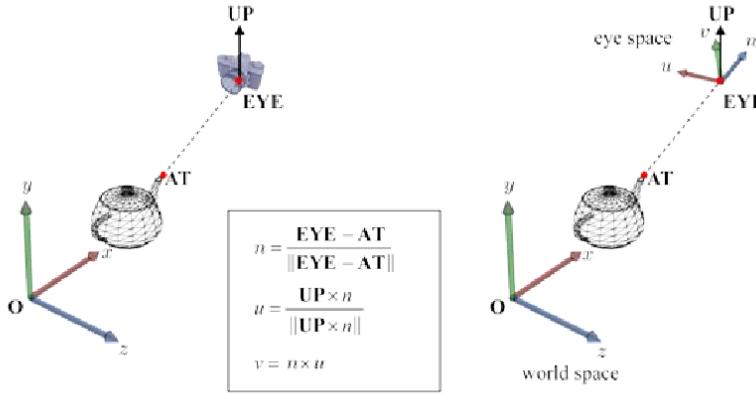


Fig. 1.5 摄像机空间的计算 (引自: *Vertex Shader*)

摄像机空间中的基 (u, v, n) 可由图 1.5 中的公式来计算, 而旋转矩阵 R 则对应了从 (e_1, e_2, e_3) 到摄像机空间基的基变换矩阵, 其中:

$$e_1 = (1, 0, 0), e_2 = (0, 1, 0), e_3 = (0, 0, 1)$$

在 LookAt 摄像机坐标系为右手坐标系, 其有三个坐标描述参量: EYE, AT 和 UP。这三个参量通过图 1.5 中的公式即可建立摄像机空间 $\{\text{EYE}, u, v, n\}$, 也可以称之为 View Space 或是 Eye Space。

- + **EYE** 为世界坐标系中摄像机的位置;
- + **AT** 为世界坐标系中摄像机的指向的场景中的某个位置, 一般来说为场景的中心点;
- + **UP** 通常代表了摄像机顶部朝向, 一般情况下为世界坐标系中 y 轴的指向。

(3) Transform of Surface Normals

OpenGL 从对象坐标系 (*Object Coordinates*) 变换到视觉坐标 (*Camera Coordinates*) 还有一个关于标准向量 (*Normal vectors*) 的变换, 这一变换主要用来计算光照视觉坐标 (*lighting calculation*) 的。标准向量的变换和顶点的不同, 它的视觉矩阵是 Model View Matrix (对应 OpenGL 中的 `GL_MODELVIEW`) 逆矩阵的转置矩阵, 如式 1.7 所示。

$$(n_x \ n_y \ n_z \ n_w)_{camera}^T = (M_{modelview}^{-1})^T \cdot (n_x \ n_y \ n_z \ n_w)_{object}^T \quad (1.7)$$

关于这一部分平面法向量, 即标准向量从对象空间变换到摄像机空间的公式推导, 可以参考 Song Ho Ahn 的博客 [OpenGL Normal Vector Transformation](#)。

1.1.3 投影变换: Projection Matrix

投影方式决定以何种方式成像, 投影方式有很多种, OpenGL 中主要使用两种方式, 即透视投影 (*Perspective Projection*) 和正交投影 (*Orthographic Projection*), 如图 1.7 所示 (引自: *Modern OpenGL*)。

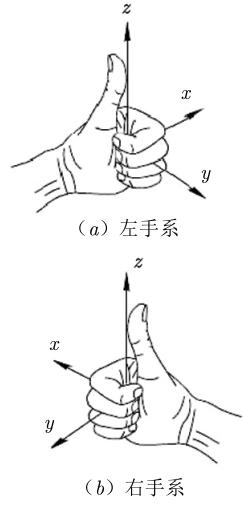
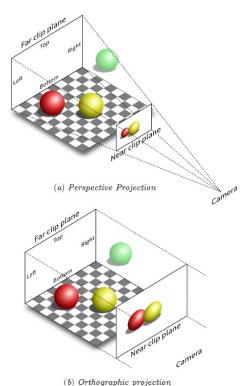


Fig. 1.6 左手系和右手系的判别方式



- + **Orthographic Projection** 正交投影是平行投影的一种特殊情形，其投影的投影线垂直于观察平面，平行投影的投影线相互平行，投影的结果与原物体的大小相等，因此广泛地应用于工程制图等方面；
- + **Perspective Projection** 透视投影的投影线相交于一点，因此其投影的结果与原物体的实际大小并不一致，而是会近大远小，因此透视投影是一种更接近于真实世界的投影方式。

(1) Perspective Projection Matrix

透视投影矩阵需要将摄像机坐标系中的点映射到一个标准立方体（即规范化设备坐标系， *Normalized Device Coordinates*, NDC）中。在建立透视投影转换关系时一共分为两步：第一步投影变换确定进入 Perspective Frustum (透视视锥体) 中的顶点坐标，第二步执行透视除法将 x, y, z 除以 w 得到其 $[-1, 1]$ 区间的规范化设备坐标，如图 1.8 所示。因此，投影矩阵应满足如下两个条件：

$$(x \ y \ z \ w)_{clip}^T = M_{projection} \cdot (x \ y \ z \ w)_{camera}^T \quad (1.8)$$

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}_{NDC} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \end{pmatrix} \quad (1.9)$$

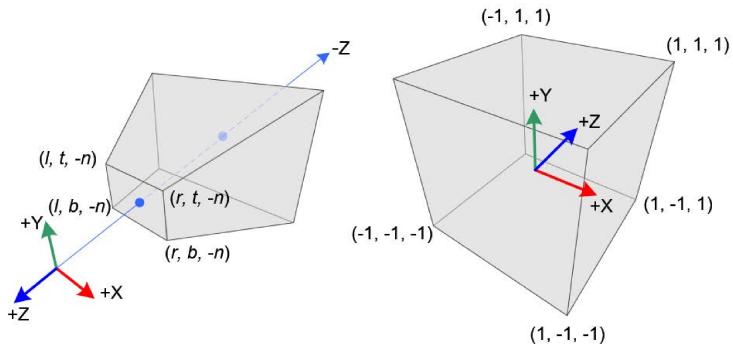


Fig. 1.8 透视投影视锥体及规范化设备坐标 (引自: OpenGL Projection Matrix)

透视投影视锥体有两种定义方式：第一种定义 `glFrustum` 有 6 个参数，即视锥体的左边界 x 坐标 `left`、右边界 x 坐标 `right`、底面 y 坐标 `bottom`、顶面 y 坐标 `top`、视锥体近裁剪面 z 坐标 `near` 和视锥体远裁剪面 z 坐标 `far`；第二种定义 `gluPerspective` 有四个参数，即视场角 `fovy`、近裁剪面宽高比 `aspect`，近裁剪面 z 坐标 `zNear` 和远裁剪面 z 坐标 `zFar`。

```
+ void glFrustum(GLdouble left, GLdouble Right, GLdouble bottom,
GLdouble top, GLdouble near, GLdouble far);
+ gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear,
GLdouble zFar)
```

通过已知的投影位置 (x_p, y_p) 与摄像机坐标系中点 $(x, y)_{camera}$ 的联系，以及投影后像机坐标系所有的 z 分量对于投影坐标来说都是 $-zNear$ 这两个条件，可以推导出基于第一种

定义条件所指定的透视投影矩阵形式如式 1.10 所示, 详见 [OpenGL Projection Matrix](#):

$$M_{projection} = \begin{pmatrix} 2n/(r-l) & 0 & (r+l)/(r-l) & 0 \\ 0 & 2n/(t-b) & (t+b)/(t-b) & 0 \\ 0 & 0 & -(f+n)/(f-n) & -2fn/(f-n) \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad (1.10)$$

式 1.10 的通用性更强, 当然, 我们一般用到的透视视锥体都是对称的, 在这种情况下, 上面的矩阵可以简化为如下形式:

$$M_{projection} = \begin{pmatrix} 2/r & 0 & 0 & 0 \\ 0 & n/t & 0 & 0 \\ 0 & 0 & -(f+n)/(f-n) & -2fn/(f-n) \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad (1.11)$$

当我们利用第二种方式进行透视投影视锥体的定义时, 矩阵的形式就变化为:

$$M_{projection} = \begin{pmatrix} \cot(\frac{\theta}{2})/aspect & 0 & 0 & 0 \\ 0 & \cot(\frac{\theta}{2}) & 0 & 0 \\ 0 & 0 & -(f+n)/(f-n) & -2fn/(f-n) \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad (1.12)$$

(2) Orthographic Projection Matrix

相对于透视投影变换, 正交投影的变换形式要简单些, 通用的正交投影矩阵如式 1.13 所示:

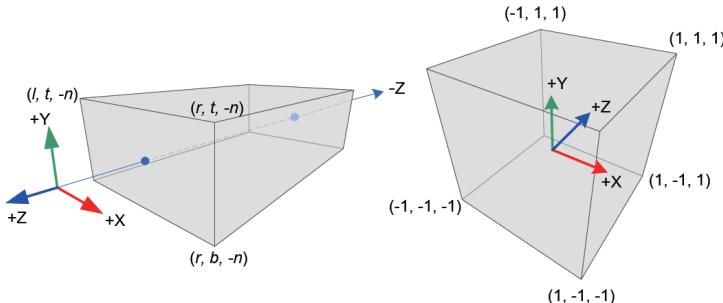


Fig. 1.9 正交投影视见体及规范化设备坐标 (引自: [OpenGL Projection Matrix](#))

$$M_{projection} = \begin{pmatrix} 2/(r-l) & 0 & 0 & -(r+l)/(r-l) \\ 0 & 2/(t-b) & 0 & -(t+b)/(t-b) \\ 0 & 0 & -2/(f-n) & -(f+n)/(f-n) \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1.13)$$

同样的, 当视见体对称时, 其形式可以简化为如下形式:

$$M_{projection} = \begin{pmatrix} 1/r & 0 & 0 & 0 \\ 0 & 1/t & 0 & 0 \\ 0 & 0 & -2/(f-n) & -(f+n)/(f-n) \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1.14)$$

1.1.4 视口变换: Viewport Mapping

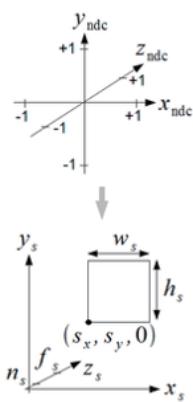


Fig. 1.10 视口坐标变换

结合前面所说的, 在 OpenGL 中, 视景体创建函数 `glOrtho()` 只是负责使用什么样的视景体来截取图像, 并不负责使用某种规则把图像呈现在屏幕上。视口函数 `glViewport()` 则负责把视景体截取的图像按照怎样的高和宽显示到屏幕上, 图像会根据视口的大小拉伸变形。视口变化通过:

```
+ glViewport(GLint  $s_x$ , GLint  $s_y$ , GLsizei  $w_s$ , GLsizei  $h_s$ );  
+ glDepthRange(GLclampf  $n_s$ , GLclampf  $f_s$ );
```

两个函数来指定, 其中 (s_x, s_y) 表示窗口的左下角, w_s 和 h_s 表示窗口尺寸, n_s 和 f_s 指定远近剪裁平面到屏幕坐标的映射关系。

视口变换的映射关系如下 (引自: [OpenGL 学习脚印: 投影矩阵和视口变换矩阵](#)):

$$\mathbf{X} : (-1, 1) \mapsto (s_x, s_x + w_s)$$

$$\mathbf{Y} : (-1, 1) \mapsto (s_y, s_y + h_s)$$

$$\mathbf{Z} : (-1, 1) \mapsto (n_s, f_s)$$

求出其映射函数为:

$$\begin{aligned} x_s &= \frac{w_s}{2} \cdot (x_n + 1) + s_x = \frac{w_s}{2} \cdot x_n + s_x + \frac{w_s}{2} \\ y_s &= \frac{h_s}{2} \cdot (y_n + 1) + s_y = \frac{h_s}{2} \cdot y_n + s_y + \frac{h_s}{2} \\ z_s &= \frac{f_s - n_s}{2} \cdot (z_n + 1) + n_s = \frac{f_s - n_s}{2} \cdot z_n + \frac{f_s + n_s}{2} \end{aligned}$$

由此可以得到视口变换的矩阵变换公式为:

$$\begin{pmatrix} x_s \\ y_s \\ z_s \\ 1 \end{pmatrix} = \begin{pmatrix} w_s/2 & 0 & 0 & s_x + w_s/2 \\ 0 & h_s/2 & 0 & s_y + h_s/2 \\ 0 & 0 & 0 & (n_s + f_s)/2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_n \\ y_n \\ z_n \\ 1 \end{pmatrix} = M_{viewport} \cdot \begin{pmatrix} x_n \\ y_n \\ z_n \\ 1 \end{pmatrix} \quad (1.15)$$

举个例子来说明, 若程序创建的窗口的宽为 `width`, 高为 `height`, 使用函数 `glViewport(x_s, y_s, width, height)` 所创建的视口即为图 1.11 所示, 图示情况下, 函数使用的具体参数为 $(width/2, 0, width/2, height/2)$ 。

1.1.5 Summary

顶点处理的几个变换, 除了模型变换容易借助简单的图示进行构建外, 其他的几个变换都不是很直观; 前文已经对 View Matrix 进行了一个粗略的描述, 我们会知道这个矩阵是一个什么样的矩阵, 在应用程序使用时该如何创建它, 至于公式推导, 在前面章节中提到的书里面有相应的方法, 简单来说, View Matrix 所对应的变换就是一个只涉及平移旋转的坐标变换。

而 Projection Matrix 则相对复杂一些, 它是直接对模型进行操作的一种变换, 经过投影变换和透视除法后, 原有的模型已经成为另一种形态, 而且它们的坐标也被限定在一个

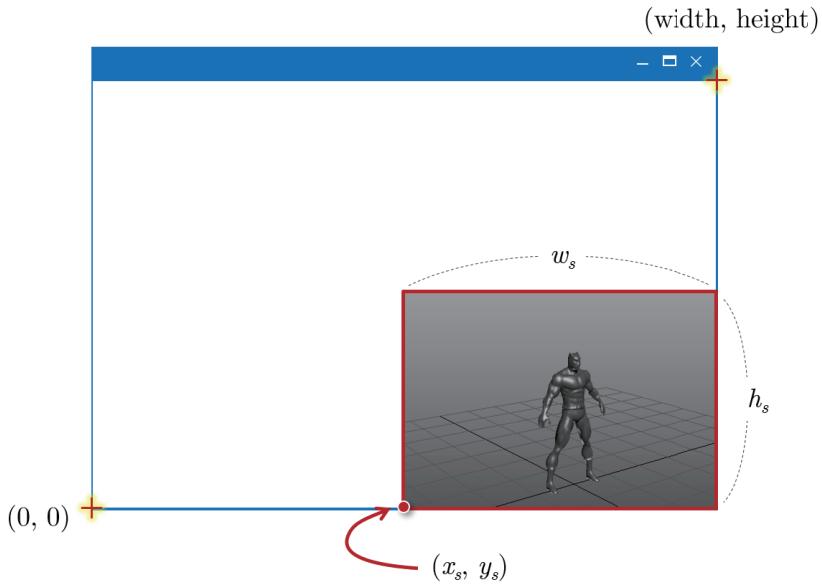


Fig. 1.11 $(width/2, 0, width/2, height/2)$ 创建的视口 (引自: OpenGL 坐标与矩阵转换)

以 $(0,0,0)$ 为中心, 边长为 2 的立方体内, opengl-tutorial 网站中的 [Tutorial 3 : Matrices](#) 对这一抽象过程作了一个阐释, 相对于其他参考材料而言更为直观易懂, 原文摘录如下:

Here's another diagram so that you understand better what happens with this Projection stuff. Before projection, we've got our blue objects, in Camera Space, and the red shape represents the frustum of the camera : the part of the scene that the camera is actually able to see.

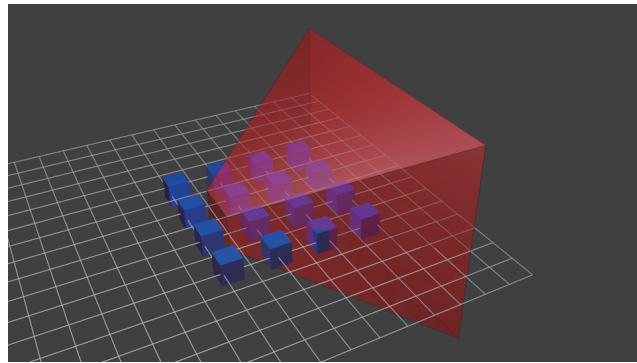


Fig. 1.12 Cube before projection

Multiplying everything by the Projection Matrix has the following effect in figure 1.13 .

In this image, the frustum is now a perfect cube (between -1 and 1 on all axes, it's a little bit hard to see it), and all blue objects have been deformed in the same way. Thus, the objects that are near the camera (= near the face of the cube that we can't see) are big, the others are smaller. Seems like real life!

Let's see what it looks like from the "behind" the frustum in figure 1.14 .

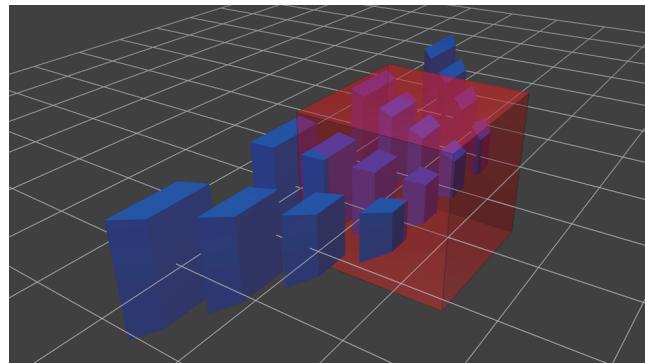


Fig. 1.13 Cube after projection

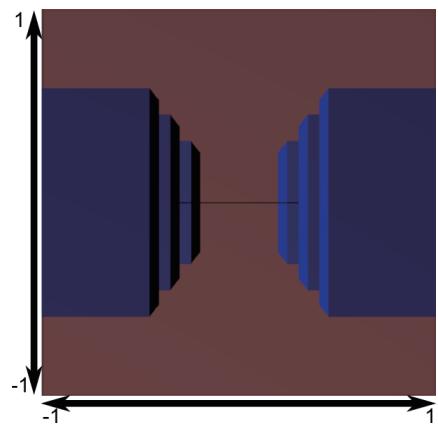


Fig. 1.14 Behind the NDC frustum

Here you get your image! It's just a little bit too square, so another mathematical transformation is applied (this one is automatic, you don't have to do it yourself in the shader) to fit this to the actual window size:

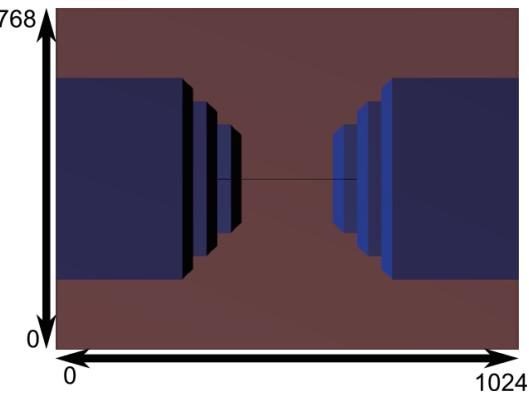


Fig. 1.15 Fit NDC to actual window size

And this is the image that is actually rendered !

1.2 Graphics rendering

进入图形渲染领域有三把紧密相连的钥匙，即通过 Vertex 所指定的顶点、直线和多边形的几何图元 **Geometric Primitives**，根据几何图元创建的 Object 所生成的模型 **Model** 以及根据模型创建显示在屏幕上的显示的图像的渲染 **Rendering**。

这里需要探讨的就是计算机是如何将三维应用程序中的一系列顶点经由图形处理器来制定几何图元，管线这一概念诉说了显示芯片处理图形信号时各个相互独立的并行处理单元间的流水线关系。

无论是二维图形还是三维图形，最终显示在显示屏上的都是众多的像素点，在内存中，一系列和像素有关的像素颜色等信息会被组织成位平面的形式。位平面是一块内存区域，保存了屏幕上每个像素的一个位的信息，例如它可以指定一个特定像素的颜色中红色成分的强度。位平面又可以组织成帧缓冲区（*Frame Buffer*）的形式，后者保存了图形硬件为了控制屏幕上所有像素的颜色和强度所需要的全部信息。

渲染管线的作用就是在 OpenGL 的管道当中通过各个处理单元的功能特性，经由数据在管道中的传递和操作最终形成我们想要的最终的一幅像素图像。

1.2.1 渲染管线概览：Pipeline

OpenGL 的渲染管线，即 Pipeline，就是指 OpenGL 的渲染过程，即从输入数据到最终产生渲染结果数据所经过的通路及所经受的处理，其主要内容概览如图 1.17 所示：

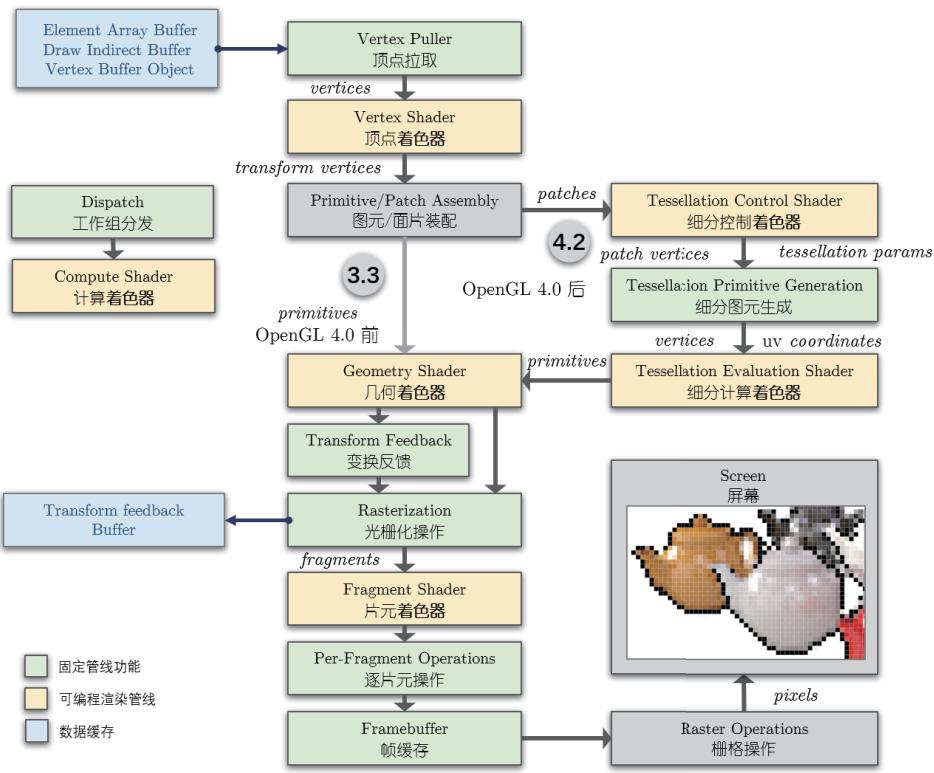


Fig. 1.17 OpenGL 渲染管线 (参考 [亮亮的园子](#) 重制)

首先解释一下这个图，图中的蓝色框表示数据缓存（data buffers），绿色框为固定管线

位平面

位平面是图像在内存中存储时所提出的一个与像素灰度值息息相关的一个概念，一幅灰度图像每个像素的灰度值可以用 0~255 这 256 个数表示：

$$256 = 2^8 \sim 8\text{bit}$$

即对应了 1Byte、8bit 的数据存储，想象这个 8 位存储空间是竖直方向的 8 个格子，那么会得到一个长宽与图像对应，高度为 8 的图像立方体，将其中某一高度的二值平面提出来就是我们所说的位平面。



Fig. 1.16 图像立方体

相应的，一幅彩色图像的一个像素可以由 RGBA 四个字节的数据来表示，对应的也就是一个高度位为 32 的存储空间。

功能 (fixed function stages), **黄色框**为可编程部分 (programmable stages) 即着色器。图中的箭头表示数据流的方向, 这个方向标识读写操作。一般来说, 并不把 Compute Shader (计算着色器) 当作管线的一部分, 它执行通用计算, 计算的结果可以用于渲染, Compute Shader 是可选的。

从 Vertex Puller (顶点拉取) 到 Framebuffer (帧缓存) 的管线中, 除了 Vertex Shader (顶点着色器) 是必须提供之外, 其他 shader 均是可选的, 如果没有提供相应 shader, 数据将不经过加工直接通过。注意 Tessellation Control Shader (TCS, 细分控制着色器) 到 Tessellation Evaluation Shader (TES, 细分求值着色器) 这整个 Tessellation (曲面细分) 过程是可选的, 在 Tessellation 过程中 TCS 是可选的, 即 TCS 只能伴随 TES 存在 (否则会报运行时错误)。Vertex Shader 不可选是因为管线至少要包含一个着色器, 而其他着色器存在的前提是存在 Vertex Shader (文献[OpenGL 4.5 Core Profile 管线 \(GLSL 与应用程序接口详解\)](#)), 从语义上来说, Vertex Shader 定义了整个管线的输入数据。

图 1.18 是一个图形渲染管线的每个阶段的抽象表达, 蓝色部分代表的是可编程部分, 也就是可以自定义的着色器。图中以数组的形式传递 3 个 3D 坐标作为图形渲染管线的输入, 在此次渲染中这个数组构成的顶点数据表示了一个三角形。一个顶点是一个 3D 坐标 (x, y, z) 的集合, 其数据是用顶点属性 (Vertex Attributes) 表示的, 它可以包含任何可编程阶段所希望用到的数据, 一个简单的顶点属性可能只包含顶点位置和 RGBA 四个颜色值。

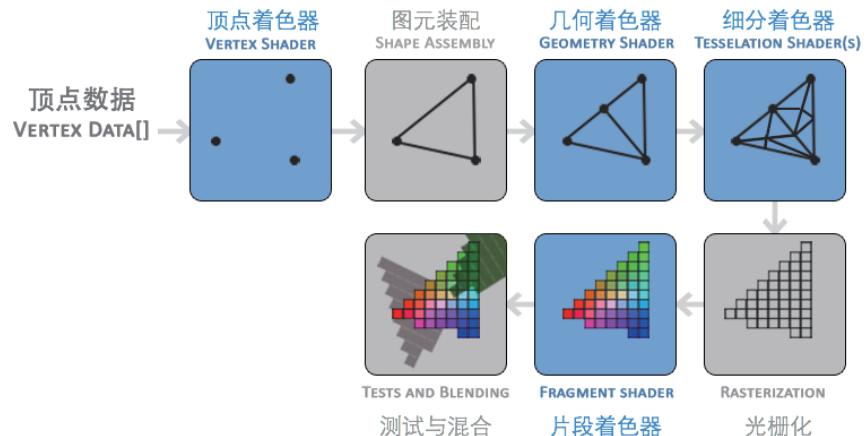


Fig. 1.18 顶点数据的操作 (引自: [入门 -04. 你好三角形](#))

管线中的每个阶段其输入为上一阶段的输出, 其输出作为下一阶段的输入, Vertex Shader 的输入和应用程序的顶点属性数据接口, Fragment Shader 的输出和帧缓存的颜色缓存接口, 互相对接的接口其内容和格式要一致。固定管线功能阶段需要的一些特定输入输出由着色器的内置输出输入变量定义。

接下来要介绍 OpenGL 渲染管线中各个处理单元的主要功能, 下面的内容主要参考自亮亮的园子里的[OpenGL 4.5 Core Profile 管线 \(GLSL 与应用程序接口详解\)](#)。

1. *Vertex Puller* (顶点拉取) : 这一操作是整个渲染管线的操作引导, 也是顶点处理器的输入端。顶点数据中的每个顶点有可任意定义的若干属性, 如位置、颜色、纹理坐标等, 这些数据存于 Vertex Buffer 中, 每个图元 (primitive) 的顶点索引存于 Element Array Buffer 中, 应用程序通过绘制命令启动管线, 如 `glDrawElements()`;

2. *Vertex Shader* (VS, 顶点着色器)：对每个顶点，OpenGL 启动一个 VS，VS 的输入即每个顶点的全部属性，还包括该顶点的索引 (index, 由内置变量 `gl_VertexID` 指示) 等，VS 无法访问其他顶点数据 (非本 VS 的输入)，VS 必须且只能输出一个顶点。VS 对顶点进行处理，典型的例子是顶点坐标变换 (将顶点齐次坐标乘以模型视图矩阵和投影矩阵) 和逐顶点光照；
3. *Tessellation* (曲面细分，可选阶段)：又分为 Tessellation Control Shader (TCS，细分控制着色器)、Tessellation Primitive Generation (细分图元生成)、Tessellation Evaluation Shader (TES，细分求值着色器)，其中 TCS 是可选的。当 Tessellation 阶段存在时，只能给管线提供 `GL_PATCHES` 类型的图元，不存在时，不能提供 `GL_PATCHES` 类型的图元，即 Tessellation 和 `GL_PATCHES` 需同时出现。
 - (1) 对每个 PATCH 由 TCSs 处理后输出的 PATCH，OpenGL 对其每个输出顶点启动一个 TCS (该输出顶点在输出 PATCH 中的索引由 `gl_InvocationID` 指示)，该 TCS 可以访问其所在的输入 PATCH 的全部顶点数据及 PATCH 的属性 (在 TCS/TES 内部由 patch 关键字指定)，即一个输入 PATCH 被一组 TCSs 处理得到一个输出 PATCH，组内 TCSs 的个数等于输出 PATCH 的顶点数，TCS 必须且只能输出一个顶点，并且可以修改其所在输出 PATCH 的属性。TCS 一般对顶点进行操作并计算 PATCH 的细分水平值 (tessellation level, `gl_TessLevelOuter[4]/Inner[2]`，是 PATCH 的属性)，若 TCS 不存在，该细分水平值均为默认 (可通过 `glPatchParameteri(GL_PATCH_VERTICES,)` 设置)。
 - (2) Tessellation Primitive Generation 根据此细分水平值对每个 PATCH 在抽象图元上 (abstract patch, "triangles" 或 "quads") 进行细分 (此阶段仅在 TES 存在时进行) 并给出生成顶点相对抽象图元顶点的相对坐标 (`gl_TessCoord`，包含抽象图元的顶点，其相对坐标某个元素为 1 其余为 0)。
 - (3) 对 Tessellation Primitive Generation 生成的每个相对坐标，OpenGL 启动一个 TES，TES 可以访问其所在输入 PATCH 的全部顶点数据及 PATCH 的属性 (即 TCS 的输出)，即 TCS 输出的 PATCH 外加相对坐标被一组 TESs 处理得到一些列输出顶点，组内 TESs 的个数等于输出顶点数，TES 必须且只能输出一个顶点。TES 一般用相对坐标从输入 PATCH 的所有顶点计算输出顶点的齐次坐标 (如根据贝塞尔曲面方程计算)。
 - (4) Tessellation 输出的顶点 (由 TES 产生) 随后被组装为图元 (非 `GL_PATCHES` 类型图元，后续阶段可以处理)；
4. *Geometry Shader* (GS, 几何着色器，可选阶段)：前面阶段中顶点数据被组装为图元 (primitive assembly)，其类型由绘制命令的参数或 TES 的输出决定。对每个图元，OpenGL 启动一个 GS，可以定义对每个图元启动多个 GS (GS 中 "layout(invocations=?)" in;)，GS 可以输出零个至多个新的图元 (用 `EmmitVertex()`、`EndPrimitive()` 等)，GS 不能访问除输入图元之外的图元，当图元类型为 `GL_*ADJACENCY` 时，每个图元带有邻接信息，该邻接信息在 GS 不存在时将被忽略。可以指定将 GS 的输出定向到特定 Stream 中，只有 Stream 0 中的

数据会继续进入 Transform Feedback 之后的图元裁剪、光栅化阶段，可以同时定向 CS 的输出至多个 Stream（此时图元类型受限制），若 GS 不存在，图元数据都将进入 Stream 0。GS 一般用作几何计算，如 Shadow Volumes。从 VS 到 GS 的阶段合起来称为顶点处理（vertex processing），顶点处理阶段应该对内置变量 `gl_Position` 进行写入（可在 VS 到 GS 的任何阶段），否则其值是未定义的（undefined），该裁剪空间的齐次坐标（clip coordinates）是后续阶段（图元裁剪、光栅化等）需要的；

5. *Vertex Post-Processing*（顶点后处理）：顶点处理的结果即输出的图元，其顶点的位置（`gl_Position`）是裁剪空间的齐次坐标（clip coords），图元在此空间内进行裁剪（clipping），随后进行透视线除法（perspective divide）转换为规范化坐标（normalized coords），然后进行视口变换（viewport transform）转换为窗口坐标（window coords），该坐标将在光栅化阶段使用，多边形图元将根据其顶点在二维窗口中的环绕方向（顺时针或逆时针，可通过二维向量叉乘结果的符号确定）被指定为正面或背面（front/back-facing）。顶点的其他属性不进行这些变换操作，但在图元被裁剪产生新顶点时，这些属性值被插值；
6. *Rasterization*（光栅化）：顶点后处理的图元其顶点为窗口坐标，光栅化阶段将确定哪些像素属于图元，对于属于图元的像素，从图元的顶点属性插值得到相应像素属性，这些具有和顶点一样属性格式的像素即“片断”（fragment）。插值的方式一般在裁剪空间进行（窗口空间的权值要除以 w_c ），也可以指定在窗口空间进行插值（FS 中“`layout(noperspective) in;`”或在 VS 中“`layout(noperspective) out;`”）或不进行插值（FS 中“`layout(flat) in;`”或在 VS 中“`layout(flat) out;`”），但对于纹理坐标等属性在窗口空间插值将得到错误结果；
7. *Transform Feedback*（变换反馈，可选阶段）：在 GS 之后，可以开启一个固定管线功能阶段，若开启（用 `glBeginTransformFeedback()` 开启，`glEndTransformFeedback()` 关闭，即在它们之间的绘制结果被记录），顶点处理的结果即输出的图元可以被写入 Transform Feedback Buffer（这一操作通过 OpenGL 的函数 `glBindBuffer(GL_TRANSFORM_FEEDBACK_BUFFER,)` 执行），Transform Feedback 过程仅记录数据不对图元进行处理，即图元直接通过（bypass）。记录在 buffer 中的数据在随后又可以被用来进行绘制（用 `glDrawTransformFeedback()` 等）；
8. *Fragment Shader*（FS，片断着色器，可选阶段）：对于光栅化产生的每个片，OpenGL 启动一个 FS，FS 的输入数据即片断的坐标（`gl_FragCoord`）、图元 ID（`gl_PrimitiveID`）等内置变量及其他定义的属性，FS 无法访问其他片断（尤其是同一图元的片断）。FS 可以丢弃（用 `discard` 关键字）但至多输出一个片断，FS 的输出和应用程序的帧缓存接口，内置变量 `gl_FragDepth` 对应深度缓存，其他属性统称为“颜色”（colors），和颜色缓存对应；VS、TCS、TES、GS、FS 均可以读写 Image/Atomic Counter/Shader Storage Buffer，可以读 Texture/Uniform Block Buffer；
9. *Per-Fragment Operations*（逐片断操作）：对 FS 或光栅化的输出片断，在将其

写入帧缓存之前，要进行一些列操作，依次是：Stencil Test、Depth Buffer Test、Occlusion Query、Blending、Logicop 等，其中前三个操作可以提前到 FS 之前（FS 中“`layout(early_fragment_tests) in;`”），这样将减少 FS 的调用次数。通过这些操作并且未被丢弃的片断将被写入帧缓存；

10. *Framebuffer*（帧缓存）：帧缓存最多由深度缓存（depth buffer）、模板缓存（stencil buffer）、若干颜色缓存（color buffers），帧缓存本身是个容器，将具体缓存加入帧缓存的操作称为 attach（这些具体缓存称为 attachment），缓存的数据统称为像素数据。可以通过缓存或纹理直接写入或读取像素数据，即 Pixel Unpack/Pack Buffer；

11. *Compute Shader*（CS，计算着色器，独立阶段）：CS 不能和其他着色器一同链接到同一个 Program 对象（参考：OpenGL Core Profile 官方手册 [p91]），和上面说的图形/渲染管线相对，可以认为 CS 独自构成计算管线。和 CUDA 类似，CS 分为两层线程模型被调用，内层为多个 Invocation 构成 Work Group（Group 内线程 ID 由 `gl_LocalInvocationID` 指示），多个 Work Group 再构成 Dispatch（Dispatch 内 Group ID 由 `gl_WorkGroupID` 指示），可以在内层声明 Work Group 内共享的变量（用 `shared` 关键字），Work Group 内还支持高效的同步机制，Dispatch 整体共享 Image/Atomic Counter/Shader Storage/Texture/Uniform Block Buffer，这也是 CS 和 VS、TCS、TES、GS、FS 通信的方式。

1.2.2 管线编程：How to program a pipeline

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。图形渲染管线可以被划分为两个主要部分：第一个部分把的 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（引自 IceMJ：入门 -04. 你好三角形）。

接下的内容将以绘制三角面片为例打通 OpenGL 的管线编程一脉，看看在 OpenGL 的实际代码中是如何使用管线的。

前文中提到了顶点属性衔接了很多着色器中所希望看到的数据，比如输入的数据应该构成什么样的基本图元，也就是 Primitives：OpenGL 的基本图元有点、线、三角形、四边形和多边形这五种，对应 Point、Line、Polygon、Triangle & Quadrangle。

(1) Vertex Input

以最简单的三角形面片的渲染为例，以 `GLfloat` 的方式定义三个 3D 顶点坐标，为了使其显示在 OpenGL 的可见区域中，将其定义为标准化设备坐标 $[0, 1]$ 区间内的点，设置其 z 坐标为 0。

```

1 static const GLfloat vertices[] = {
2     -0.5f, -0.5f,  0.0f,
3     0.5f, -0.5f,  0.0f,
4     0.0f,  0.5f,  0.0f
5 }
```

OpenGL 基本图元

点: <code>GL_POINTS</code>
线段: <code>GL_LINES</code>
折线: <code>GL_LINE_STRIP</code>
闭合线: <code>GL_LINE_LOOP</code>
三角形: <code>GL_TRIANGLES</code>
连续三角形串: <code>GL_TRIANGLE_STRIP</code>
扇形连续三角: <code>GL_TRIANGLE_FAN</code>

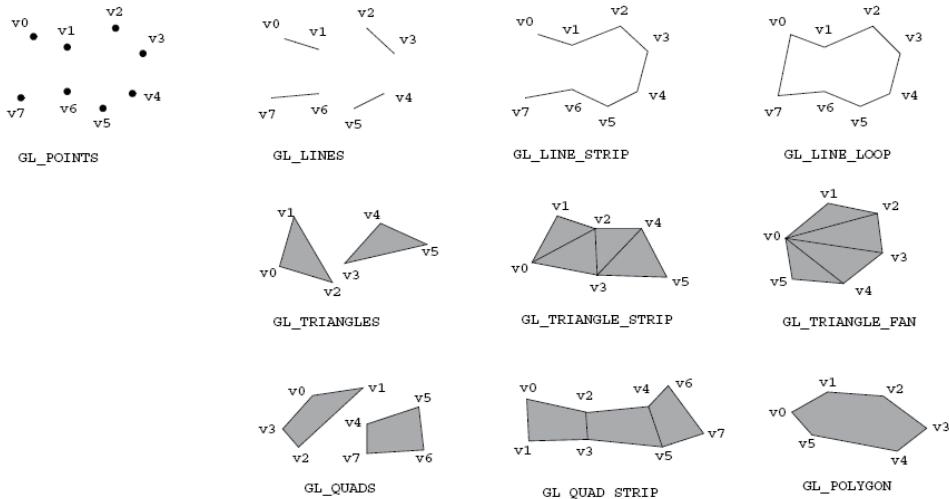


Fig. 1.19 OpenGL 的基本图元

构建好顶点数组之后，还需要在 GPU 上创建存储空间，并对相应内存进行配置，对 OpenGL 着色器如何处理这些顶点进行解释；这块内存的管理通过 Vertex Buffer Objects（简称 VBO）来管理，它会在 GPU 显存中存储大批量的顶点，从而减少一个个传输顶点的开销。

创建顶点时所产生的缓存对象需要用 `glGenBuffers()` 函数生成一个缓冲 ID，并用 `glBindBuffer()` 函数将缓存对象绑定到对应的缓存类型上，绑定的机制是为了满足 OpenGL 对同时绑定多个缓存类型的需求。

glBufferData()

该函数的 `usage` 枚举有如下 9 个枚举值：

`GL_STATIC_DRAW`
`GL_STATIC_READ`
`GL_STATIC_COPY`
`GL_DYNAMIC_DRAW`
`GL_DYNAMIC_READ`
`GL_DYNAMIC_COPY`
`GL_STREAM_DRAW`
`GL_STREAM_READ`
`GL_STREAM_COPY`

```

1  GLuint VBO;
2  /* 函数: glGenBuffer(GLsizei n, GLunit* buffers)
3      n: 需要创建的缓存的数量
4      buffers: 存储单一 ID 或多个 ID 的 GLunit 变量或数组地址 */
5  glGenBuffers(1, &VBO);
6  /* 函数: glBindBuffer(GLenum target, GLunit buffer)
7      target: 指定缓存对象的类型, GL_ARRAY_BUFFER/GL_ELEMENT_ARRAY
8      buffer: 缓存对象的 ID 或地址 */
9  glBindBuffer(GL_ARRAY_BUFFER, VBO);
10 /* 函数: glBufferData(GLenum target,
11                     GLsizeiptr size,
12                     const GLvoid* data,
13                     GLenum usage)
14      target: 可以为 GL_ARRAY_BUFFER/GL_ELEMENT_ARRAY_BUFFER
15      size: 待传递的数据的字节数量
16      data: 源数据的数组指针, 为空则预留给定数据大小的内存空间
17      usage: 标志 VBO 缓存对象如何使用的枚举 */
18  glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

```

绑定之后在对应的缓存类型上所进行的一系列缓存操作都会用来解释当前绑定的缓存

对象 VBO，这时，我们就可以通过 `glBufferData()` 函数将之前定义的顶点数据复制到缓冲的显存中，该函数的第四个参数的枚举可分为三种形式：STATIC 用于数据几乎不会改变时，DYNAMIC 用于数据改变很多时，STREAM 用于数据每次绘制时都会改变时。

OpenGL 的工作模式是 Client-Server 模式，绘图的整个过程不过是把数据从系统的内存中复制到图形卡中，然后绘制出图形。一般而言 OpenGL 系统的客户端负责发送 OpenGL 命令，服务端负责接收 OpenGL 命令并执行相应的操作。比如用户编写的程序可以作为客户端，而计算机图形硬件制造商提供的 OpenGL 的实现就是服务器。对于个人计算机来说，可以将 CPU、内存等硬件，以及用户编写的 OpenGL 程序看做客户端，而将 OpenGL 驱动程序、显示设备等看做服务端（可参考[OpenGL 的客户端和服务器模式](#)）。

用 VBO 每绘制一个几何体都需要重复同样的工作（首先绑定缓冲区、然后设置顶点属性），因此当需要绘制的物体很多时，这个过程就会略有些耗时。为了简化这一过程，Vertex Array Object 一脚迈进了 OpenGL 的世界，它将所有顶点绘制过程中的设置和绑定过程等配置集中存储在一起，当我们需要时，只需要使用相应的 VAO 即可。

这里需要着重介绍 VBO、VAO、EBO 三个对象，即：Vertex Buffer Object，Vertex Array Object & Element Buffer Object（可参考[OpenGL 图形渲染管线、VBO、VAO、EBO 概念及用例](#)）。

+ **VBO**：顶点缓冲对象 VBO 是在显卡存储空间中开辟出的一块内存缓存区，用于存储顶点的各类属性信息，如顶点坐标，顶点法向量，顶点颜色数据等。在渲染时，可以直接从 VBO 中取出顶点的各类属性数据，由于 VBO 在显存而不是在内存中，不需要从 CPU 传输数据，处理效率更高。所以可以理解为 VBO 就是显存中的一个存储区域，可以保持大量的顶点属性信息。并且可以开辟很多个 VBO，每个 VBO 在 OpenGL 中有它的唯一标识 ID，这个 ID 对应着具体的 VBO 的显存地址，通过这个 ID 可以对特定的 VBO 内的数据进行存取操作。

+ **VAO**：VBO 保存了一个模型的顶点属性信息，每次绘制模型之前需要绑定顶点的所有信息，当数据量很大时，重复这样的动作变得非常麻烦。顶点数组对象 VAO 可以把这些所有的配置都存储在一个对象中，每次绘制模型时，只需要绑定这个 VAO 对象就可以了。VAO 是一个保存了所有顶点数据属性的状态结合，它存储了顶点数据的格式以及顶点数据所需的 VBO 对象的引用。VAO 本身并没有存储顶点的相关属性数据，这些信息是存储在 VBO 中的，VAO 相当于是对很多个 VBO 的引用，把一些 VBO 组合在一起作为一个对象统一管理。VAO 对象中存储的内容包括：

- a. VAO 开启或者关闭的状态；
- b. 使用 `glVertexAttribPointer` 对顶点属性进行的设置；
- c. 存储顶点数据的 VBO 对象的引用。

+ **EBO**：索引缓冲对象 EBO 是为了解决同一个顶点多次重复调用的问题而出现的，EBO 可以减少内存空间浪费，提高执行效率。这样，当需要使用重复的顶点时就能够通过顶点的位置索引来调用顶点，而非对需要重复使用的顶点进行重复记录。当然，EBO 也是在显存中的一块内存缓冲器，只不过它存储顶点位置的索引 Indices。

VAO 是 OpenGL CoreProfile 引入的一个特性，主要用来 OpenGL 处理顶点数据的一个缓冲区对象，不能单独使用，需要结合 VBO 来一起使用，在 CoreProfile 中做顶点数据传入时，必须使用 VAO 方式。使用 VAO 和使用 VBO 很相似，也需要先在显存上开辟存储空间并记录唯一地址标识 ID，使用它的唯一操作就是调用 `glBindVertexArray` 将 VBO 绑定到希望用到的配置。通常情况下配置好 VAO 后需要解绑 OpenGL 对象，避免在某些地方错误地配置它们。

```

1 GLuint VAO;
2 glGenVertexArrays(1, &VAO);
3 glBindVertexArray(VAO);
4 //generate and bind VBO
5 //other OpenGL configer settings
6 glBindVertexArray(0);

```

使用 EBO 也需要几个类似操作：创建缓存对象、绑定缓存类型、将索引缓存数据复制到显存，只不过它用 `glBufferData` 函数复制的数据是顶点的索引值。

```

1 /* Vertices of rectangle.*/
2 static const GLfloat vertices[] = {
3     -0.5f, -0.5f,  0.0f,
4     -0.5f,  0.5f,  0.0f,
5     0.5f,  0.5f,  0.0f,
6     0.5f, -0.5f,  0.0f
7 };
8 /* Elements indices of the rectangle.*/
9 static const GLuint indices[] = {
10     0, 1, 2, //first triangle
11     0, 2, 3 //second triangle
12 };
13 /* Generate and bind VBO data.*/
14 //...
15 /* Generate and bind EBO data.*/
16 GLuint EBO;
17 glGenBuffers(1, &EBO);
18 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
19 glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
20             GL_STATIC_DRAW);

```

(2) Use Vertex and Fragment Shader

准备好顶点数据后，接下来就可以凭借 OpenGL Shader Language（简称 GLSL）这种语言接引可编程管线并对自定义其渲染操作；这里使用两个必备的着色器：顶点着色器和片元着色器。

Step 01 Prepare the shaders

首先需要用 `const char*` 类型声明并定义好两个用 GLSL 书写的着色器 Vertex shader 和 Fragment Shader, 着色器中的 `layout (location = 0)` 标识符标识了脚本中输入变量 `vPosition` 对应的 ID, 它的作用是在接下来用 `glVertexAttribPointer` 函数阐明顶点数据时链接顶点着色器变量和 OpenGL 中对应创建的变量。着色器脚本代码如下:

```

1  /* Write a vertex shader with GLSL.*/
2  const char* vertexShaderSource = {
3      "#version 450 core\n"
4      "layout (location = 0) in vec3 vPosition;\n"
5      "void main()\n"
6          "    gl_Position = vec4(vPosition.xyz, 1.0);\n"
7      }\n"
8  };
9  /* Write a fragment shader with GLSL.*/
10 const char* fragmentShaderSource = {
11      "#version 450 core\n"
12      "layout (location = 0) out vec4 fColor;\n"
13      "void main()\n"
14          "    fColor = vec4(1.0, 0.5, 0.2, 1.0);\n"
15      }\n"
16 };

```

Step 02 Generate and compile the shader

上面已经定义好了一个顶点着色器脚本和一个片元着色器脚本, 为了让 OpenGL 能够使用它, 需要创建一个着色器对象, 然后引用它的 ID 把这个着色器源码用 `glShaderSource` 函数附加到着色器对象并用 `glCompileShader` 编译它::

```

1  /* Create vertex shader.*/
2  GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
3  glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
4  glCompileShader(vertexShader);
5
6  /* Create fragment shader.*/
7  GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
8  glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
9  glCompileShader(fragmentShader);

```

Step 03 Use shader program object link the shaders

着色器程序对象 (Shader Program Object) 是多个着色器最后链接的版本。为了使用刚才编译的着色器必须在 OpenGL 中把它们链接为一个着色器程序对象, 然后在渲染物体

的时候激活这个着色器程序。激活了的着色器程序的着色器，在调用渲染函数时才可用。把着色器链接为一个程序就等于把每个着色器的输出链接到下一个着色器的输入。如果输出和输入不匹配那么就会得到一个链接错误。

使用着色器程序对象时首先要用 `glCreateProgram` 函数创建一个程序，返回新创建的程序对象的 ID 引用，接下来把前面编译好的着色器用 `glAttachShader` 附加到程序对象上，随后用 `glLinkProgram` 链接它们：

```

1  /* Create a shader program object.*/
2  GLuint shaderProgram = glCreateProgram();
3  /* Attach the compiled shader.*/
4  glAttachShader(shaderProgram, vertexShader);
5  glAttachShader(shaderProgram, fragmentShader);
6  /* Link shades to the program.*/
7  glLinkProgram(shaderProgram);

```

在链接好着色器程序对象后，就可以使用 `glDeleteShader` 函数删掉前面生成的两个着色器对象了，在绘图时需要使用 `glUseProgram` 函数来调用着色器对象。

(3) Link and interpret the vertex data.

顶点着色器具有很强的灵活性，它允许我们以任何想要的形式作为 Vertex Attribute 的输入端，所以必须手动指定程序输入数据的哪一个部分对应顶点着色器的哪一个顶点属性，即在渲染前指定 OpenGL 如何解释顶点数据；这一操作可以通过 `glVertexAttribPointer` 函数实现。

```

1  /* 函数: void glVertexAttribPointer(GLuint index,
2  //                                GLint size,
3  //                                GLenum type,
4  //                                GLboolean normalized,
5  //                                GLsizei stride,
6  //                                const void *ptr)
7  //      index: 指定配置着色器的 layout 所声明的那个变量
8  //      size: 此类型数据的个数
9  //      type: 指定数据使用哪种类型
10 //      normalized: 是否进行归一化处理, 映射到 [0,1] 或 [-1,1] (signed)
11 //      stride: 步长, 连续顶点属性间隔多少字节
12 //      ptr: 属性数据在当前 VBO 中起始位置的偏移量 */
13 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
14 glEnableVertexAttribArray(0); //open the vertex attribute

```

每个 Vertex Attribute 从 VBO 管理的内存中获得它的数据，它所获取数据的那个 VBO，当前绑定到 `GL_ARRAY_BUFFER` 的那个 VBO。如上面代码中的顶点属性 0（当前仅含位置属性且其起始位置偏移量为 0）现在链接到了它的顶点数据，图 1.20 描述了 `glVertexAttribPointer` 与 VAO、VBO 及 EBO 之间的关系。

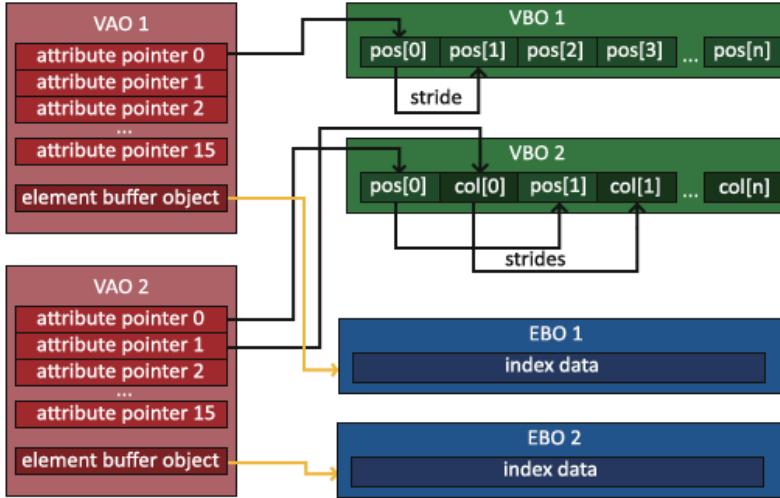


Fig. 1.20 用顶点属性链接函数链接的 VBO、VAO 和 EBO (引自: Hello Triangle)

(4) Draw the object.

到目前为止, 本节基本已经介绍完了管线装配跟顶点操作的所有内容, 下一步就是在屏幕上显示它, 我们传入的三个顶点需要按照三角形的形式绘制出来, 使用 `glDrawArrays` 函数, 传递基本类型为三角形, 数组索引从 0 开始, 共绘制 3 个点, 其代码如下:

```

1 glUseProgram(shaderProgram);
2 glBindVertexArray(VAO);
3 glDrawArrays(GL_TRIANGLES, 0, 3);
4 glBindVertexArray(0);

```

绘制结果如图 1.21 所示。

假设我们不再绘制一个三角形而是矩形, 可以通过绘制两个三角形来组成一个矩形 (OpenGL 主要就是绘制三角形), 绘制需要 6 个顶点, 但其实矩形只有 4 个顶点, 右下角和左上角的顶点被指定了两次, 从而产生了 50% 的额外开销, 最好的解决方案就是每个顶点只储存一次, 通过 EBO 来使用顶点的索引。使用 EBO 对象绘制矩形的操作与上面的内容类似, 不过绘制对象时使用的函数为 `glDrawElements`, 基本类型与上面一样, 第二个参数为绘制 6 个顶点, 第三个参数为索引类型, 第四个参数为指定 EBO 中的初始偏移量, 绘制代码如下:

```

1 glPolygonMode(GL_FRONT_AND_BACK, GL_LINE); //draw lines
2 glUseProgram(shaderProgram);
3 glBindVertexArray(VAO);
4 glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0)
5 glBindVertexArray(0);
6 glPolygonMode(GL_FRONT_AND_BACK, GL_FILL); //draw polygon

```

绘制结果如图 1.22 所示。

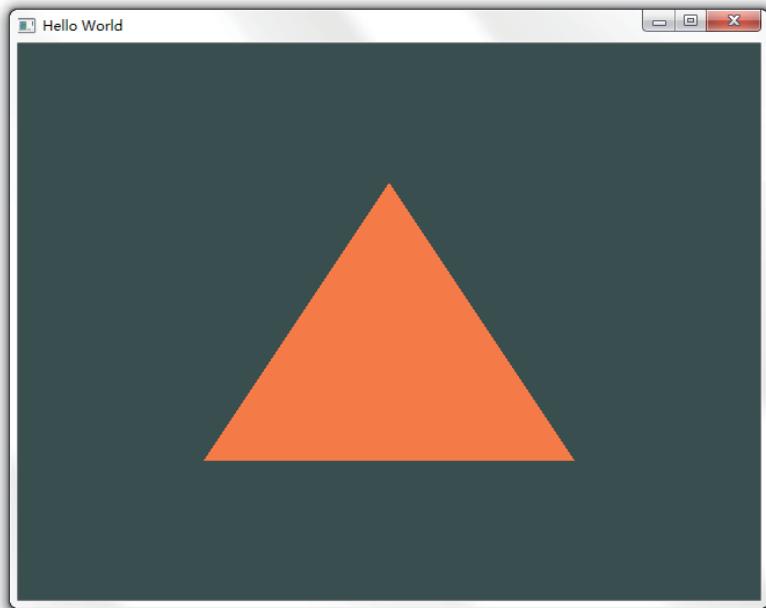


Fig. 1.21 使用 VAO 和 VBO 绘制的三角形

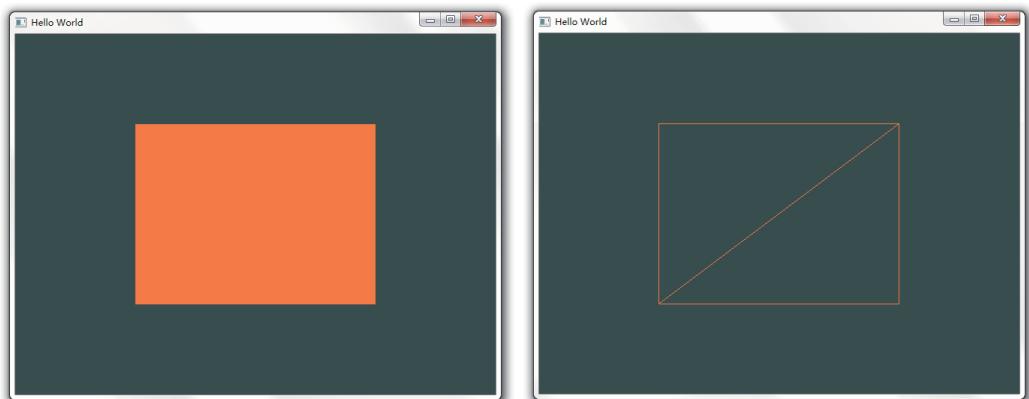


Fig. 1.22 使用 VAO、VBO 和 EBO 绘制的矩形（右边为线框模式）

至此，关于管线编程的基本内容已经完成，本节内容主要参考自英文文献 [Hello Triangle] 及 IceMJ 对英文原文的翻译改动 [入门 -04. 你好三角形]。本节绘制三角形和矩形的源码可点击[此处](#)链接下载。