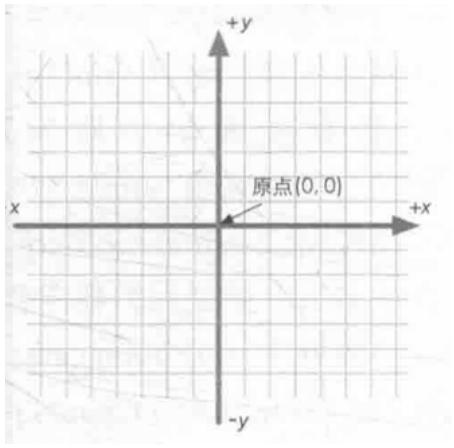
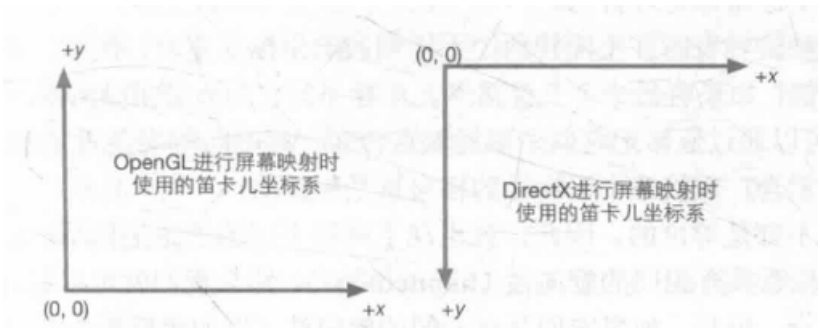
Lecture4 Shader 数学基础

1. 笛卡尔坐标系 Cartesian Coordinate System

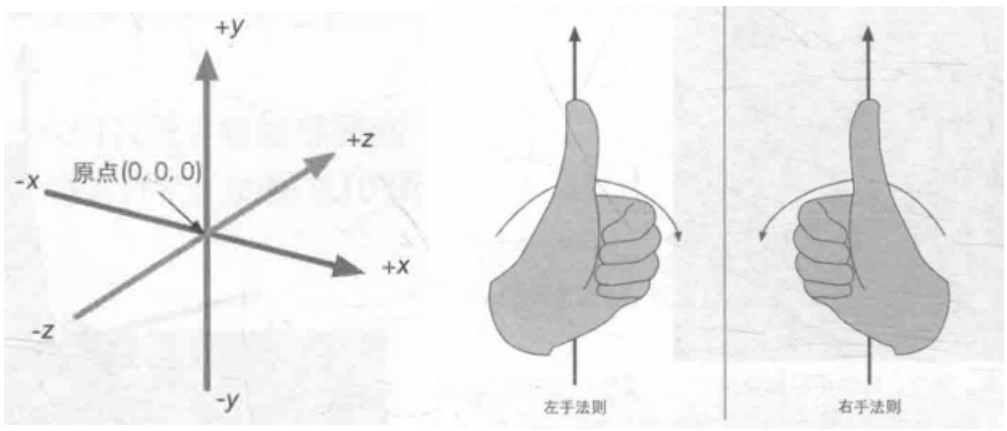
二维笛卡尔坐标系



OpenGL 和 DirectX 使用不同的二维笛卡尔坐标系



三维笛卡尔坐标系

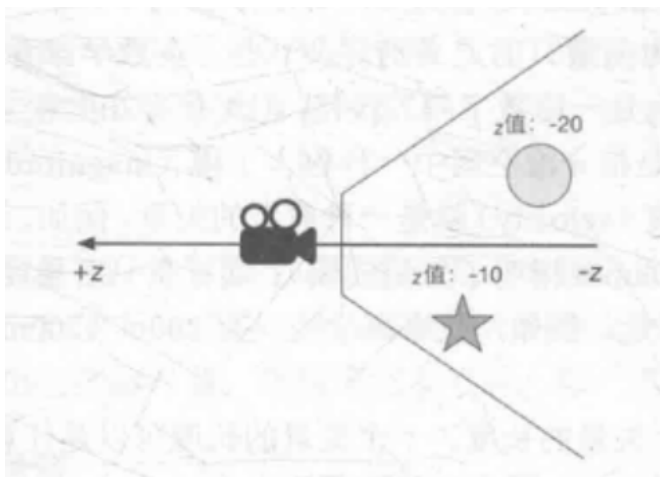


三维笛卡尔坐标系分成两种

- 左手坐标系
- 右手坐标系

左右手坐标系在 z 轴上的移动以及旋转方向是不同的

Unity 使用的坐标系



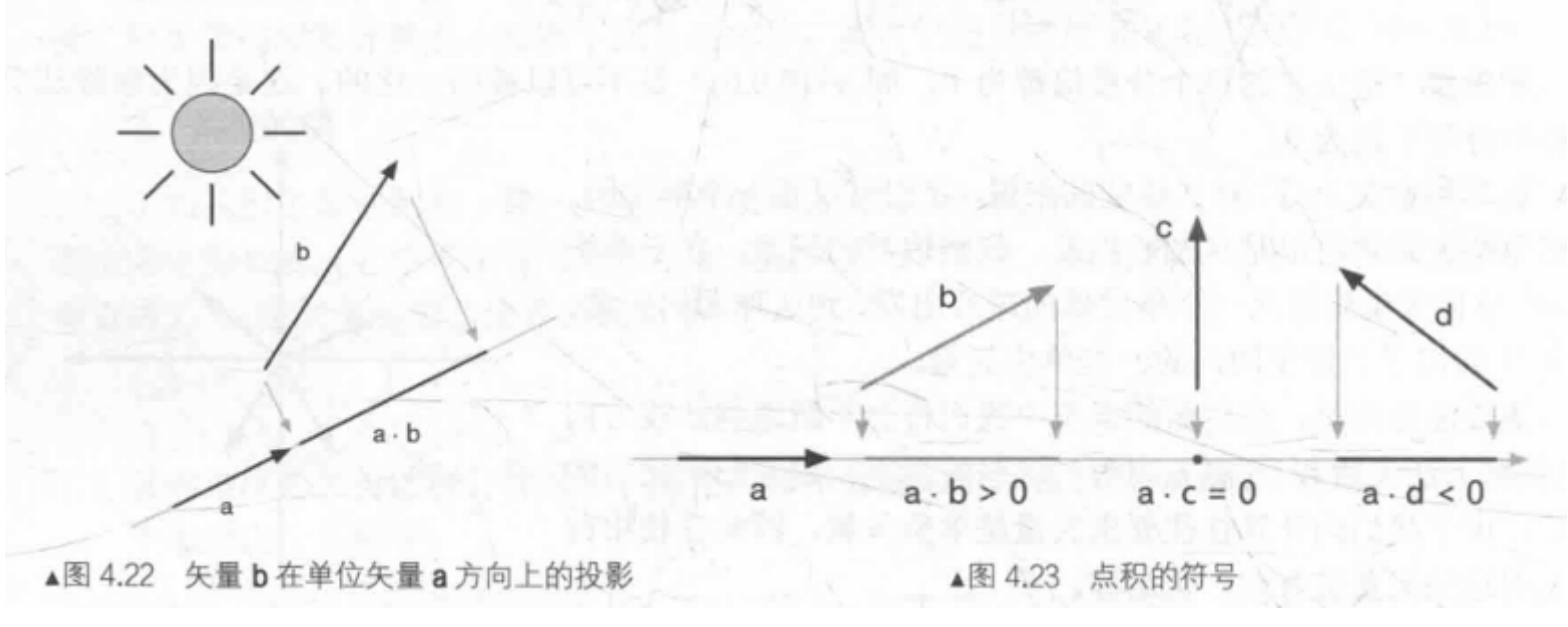
- **模型空间和世界空间**：Unity 使用的的是**左手坐标系**，物体的右侧 right、上侧 up 和前侧 forward 分别对应了 x 轴、y 轴和 z 轴的正方向
- **观察空间**：Unity 使用的是**右手坐标系**，即以摄像机为原点的坐标系，摄像机的前向是 z 轴的负方向

2. 向量

向量的点积

点积的几何意义很重要，因为点积几乎应用到了图形学的各个方面。

其中一个几何意义就是**投影 projection**。



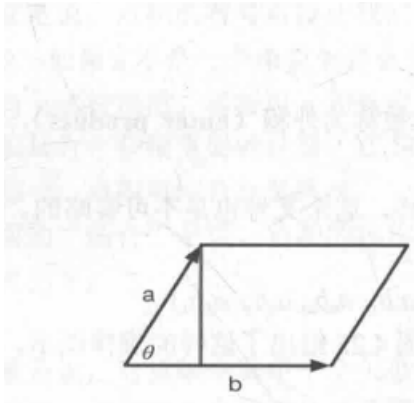
▲图 4.22　矢量 b 在单位矢量 a 方向上的投影

▲图 4.23　点积的符号

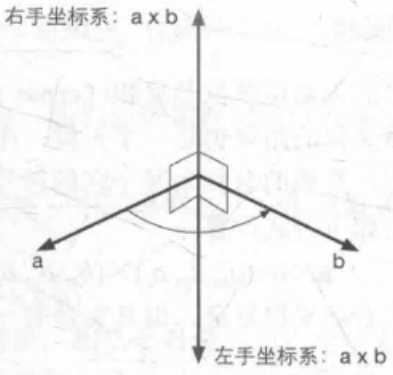
$$\mathbf{a}\cdot\mathbf{b}=(a_x,a_y,a_z)\cdot(b_x,b_y,b_z)=a_xb_x+a_yb_y+a_zb_z=|\mathbf{a}|\times|\mathbf{b}|\times cos\theta$$

- 满足交换律 $\mathbf{a}\cdot\mathbf{b}=\mathbf{b}\cdot\mathbf{a}$
- 投影结果的正负号与 $\mathbf{a},\mathbf{b}$ 的方向有关
  - <0：夹角大于 90°
  - =0：相互垂直
  - >0：夹角小于 90°

向量的叉积



▲图 4.26　使用矢量 a 和矢量 b 构建一个平行四边形



▲图 4.27　分别使用左手坐标系和右手坐标系得到的叉积结果

叉积最常见的应用是计算垂直于一个平面、三角形的矢量。另外，还可以用于判断三角面片的朝向。

3. 矩阵

行矩阵还是列矩阵

在 Unity 中，常规做法是把向量放在矩阵的**右侧**，即把向量转换成**列矩阵**来进行运算。

这意味着我们的矩阵乘法通常都是右乘，例如

$$\boldsymbol{CBAv}=(\boldsymbol{C}(\boldsymbol{B}(\boldsymbol{Av})))$$

使用列向量的结果是，我们的阅读顺序是从右到左，即先对 $\mathbf{v}$ 使用 $\mathbf{A}$ 进行变化，再使用 $\mathbf{B}$ 进行变换，最后使用 $\mathbf{C}$ 进行变换

上面的计算结果等于下面的行矩阵运算

$$\boldsymbol{v}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}^{\mathrm{T}}\boldsymbol{C}^{\mathrm{T}}=\left(\left(\left(\boldsymbol{v}\boldsymbol{A}^{\mathrm{T}}\right)\boldsymbol{B}^{\mathrm{T}}\right)\boldsymbol{C}^{\mathrm{T}}\right)$$

4. 坐标空间

我们在第 2 章讲的渲染流水线中接触了坐标空间的变换。顶点着色器最基本的功能就是把模型的顶点坐标从**模型空间**转换到**齐次裁剪坐标空间**中。

为什么要使用这么多不同的坐标空间

在编写 Shader 的过程中，很多看起来很难理解的复杂的数学运算都是为了在不同坐标空间之间转换点和向量。

我们需要在不同的情况下使用不同的坐标空间，因为一些概念只有在特定的坐标空间下才有意义，才更容易理解。

坐标空间的变换

要想定义一个坐标空间，必须指明其**原点位置和 3 个坐标轴的方向**

这些数值其实是相对于另一个坐标空间的，坐标空间会形成一个层次结构

- 每个坐标空间都是另一个坐标空间的子空间
- 每个坐标空间都有一个父坐标空间

对坐标空间的变换实际上就是在父空间和子空间之间对点和矢量进行变换

> 假设，现在父坐标空间 $\mathbf{P}$ 和子坐标空间 $\mathbf{C}$。我们知道在父坐标空间中子坐标空间的原点位置以及 3 个单位坐标轴
>
> 我们一般会有两种需求
>
> - 一种需求是把子坐标空间下表示的点或矢量 $\mathbf{A}_c$，转换到父坐标空间下的表示$\mathbf{A}_p$
> - 另一个需求是反过来，即把父坐标空间下表示的点或矢量 $\mathbf{B}_p$，转换到子坐标空间下的表示 $\mathbf{B}_c$
>
> 我们可以使用下面的公式来表示这两种需求：
>
> $$\boldsymbol{A}_p = \boldsymbol{M}_{c-p}\boldsymbol{A}_c$$
> $$\boldsymbol{B}_c = \boldsymbol{M}_{p-c}\boldsymbol{B}_p$$
>
> - $\boldsymbol{M}_{c-p}$：从子坐标空间变换到父坐标空间的变换矩阵
> - $\boldsymbol{M}_{p-c}$：从父坐标空间变换到子坐标空间的变换矩阵（逆矩阵）

已知子坐标空间 $\mathbf{C}$ 的 3 个坐标轴在父坐标空间 $\mathbf{P}$ 下的表示 $\mathbf{x}_c, \mathbf{y}_c, \mathbf{z}_c$ 以及其原点位置 $\mathbf{O}_c$，当给定一个子坐标空间中的点 $\mathbf{A}_c = (a, b, c)$，我们可以按照如下步骤来确定其在父坐标空间下的点 $\mathbf{A}_p$

$$\mathbf{A}_p = \mathbf{O}_c + a\mathbf{x}_c + b\mathbf{y}_c + c\mathbf{z}_c$$

我们把它以矩阵的形式表述为：

$$
\begin{aligned}
&= \begin{bmatrix} x_{O_c} \\ y_{O_c} \\ z_{O_c} \end{bmatrix} + a \begin{bmatrix} x_{x_c} \\ y_{x_c} \\ z_{x_c} \end{bmatrix} + b \begin{bmatrix} x_{y_c} \\ y_{y_c} \\ z_{y_c} \end{bmatrix} + c \begin{bmatrix} x_{z_c} \\ y_{z_c} \\ z_{z_c} \end{bmatrix} \\
&= \begin{bmatrix} x_{O_c} \\ y_{O_c} \\ z_{O_c} \end{bmatrix} + \begin{bmatrix} x_{x_c} & x_{y_c} & x_{z_c} \\ y_{x_c} & y_{y_c} & y_{z_c} \\ z_{x_c} & z_{y_c} & z_{z_c} \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} \\
&= \begin{bmatrix} x_{O_c} \\ y_{O_c} \\ z_{O_c} \end{bmatrix} + \begin{bmatrix} | & | & | \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c \\ | & | & | \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix}
\end{aligned}
$$

由于它还是一个仿射变换，我们已经知道 3x3 的矩阵无法表示平移变换，所以为了得到更同意的结果，我们拓展上述式子到齐次空间坐标系中

$$
\begin{bmatrix} x_{O_c} \\ y_{O_c} \\ z_{O_c} \\ 1 \end{bmatrix} + \begin{bmatrix} | & | & | & 0 \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c & 0 \\ | & | & | & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix}
$$

$$
= \begin{bmatrix} 1 & 0 & 0 & x_{O_c} \\ 0 & 1 & 0 & y_{O_c} \\ 0 & 0 & 1 & z_{O_c} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} | & | & | & 0 \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c & 0 \\ | & | & | & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix}
$$

$$
= \begin{bmatrix} | & | & | & x_{O_c} \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c & y_{O_c} \\ | & | & | & z_{O_c} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix}
$$

$$
= \begin{bmatrix} | & | & | & | \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c & O_c \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix}
$$

所以

$$
\boldsymbol{M}_{c \to p} = \begin{bmatrix} | & | & | & | \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c & \boldsymbol{O}_c \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

一旦求出 $\boldsymbol{M}_{c \to p}$，那么 $\boldsymbol{M}_{p \to c}$ 就可以通过**逆矩阵**的方式求出来

子坐标空间的原点和坐标轴反推

当我们知道一个变换矩阵后，可以使用它来反推获取子坐标空间的原点和坐标轴方向

我们提取变换矩阵的前三列再归一化，就得到了子空间下的 x 轴，y 轴和 z 轴

向量的坐标空间变换

向量没有位置，因此坐标空间变换的原点变换可以忽略

向量的坐标空间变换可以使用 3 × 3 的矩阵表示

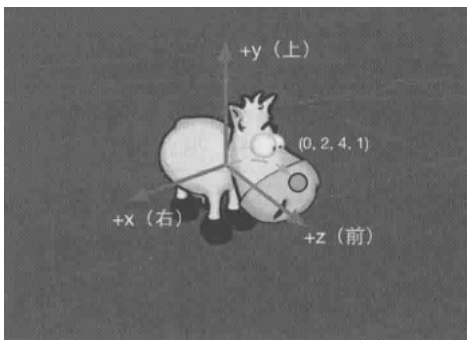
$$
\boldsymbol{M}_{c \to p} = \begin{bmatrix} | & | & | \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c \\ | & | & | \end{bmatrix}
$$

5. 顶点空间坐标变换流程

在渲染流水线中，一个顶点要经过多个坐标空间的变换才能最终被画到屏幕上

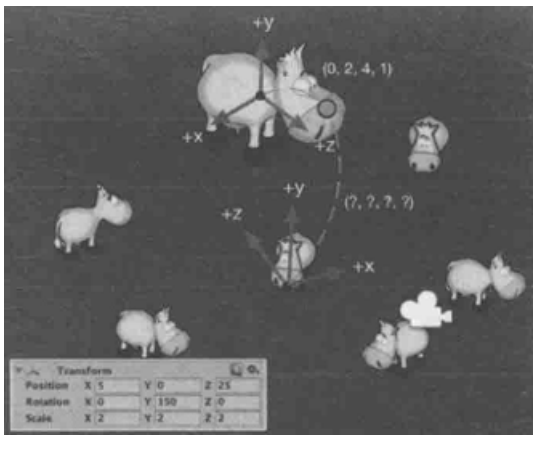
一个顶点最开始是在**模型空间**中定义的，最终它将会变换到**屏幕空间**中

模型空间 Model Space



- 模型空间也被称为**对象空间 Object Space** 或**局部空间 Local Space**，每个模型都有自己独立的坐标空间，**当它旋转或移动的时候，模型空间也会跟着移动旋转**，当模型转动时，模型本身的前后左右方向也会跟着改变
- Unity 使用的是**左手坐标系**，在模型空间中，+x 轴、+y 轴和 +z 轴分别对应模型的**右、上和前向。**我们在 Hierarchy 视图中单击任意对象就可以看到它们对应模型空间的 3 个坐标轴
- 模型空间的原点通常是**美术**人员在**建模软件**确认好的，通常是模型的中心，当导入 Unity 后，可以通过顶点着色器访问到模型的定点信息，包括坐标（相对于模型空间中的原点）
- 由于顶点变换经常包括平移变换，因此需要把其**扩展到齐次坐标系下**，得到的顶点坐标是 (0, 2, 4, 1)，如上图所示

世界空间 World Space



- 通常，我们会把**世界空间中的原点放在游戏空间的中心**
- Unity 中的世界空间使用**左手坐标系。**
- 可以通过改变物体 Transform 组件中的 Position 属性来改变模型的位置
  - 如果有父节点，则位置是相对于这个 Transform 父节点的模型坐标空间定义的
  - 如果没有父节点，那么这个位置就是**世界坐标系**中的位置

模型变换 Model Transform

- 将顶点坐标从模型空间变换到世界空间中
- 在上图中，对小牛模型进行了 (2, 2, 2) 的缩放，(0, 150, 0) 的旋转和 (5, 0, 25) 的平移
  - 注意顺序是进行**缩放**、再进行**旋转**、最后再**平移**

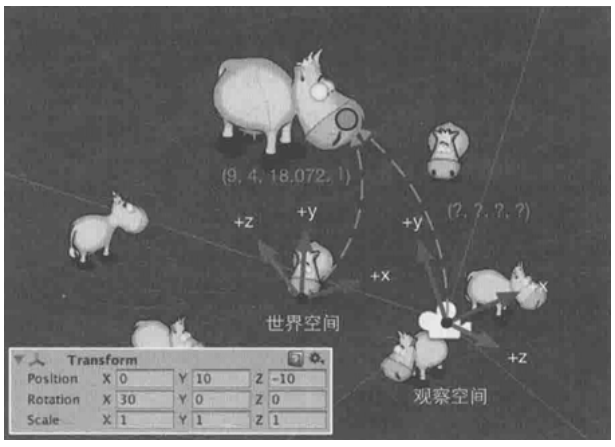
据此我们可以构建模型的变换矩阵

$$
\begin{aligned}
\boldsymbol{M}_{\text{modsl}} &= \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 25 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -0.866 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \\ -0.5 & 0 & -0.866 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} -1.732 & 0 & 1 & 5 \\ 0 & 2 & 0 & 0 \\ -1 & 0 & -1.732 & 25 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

对顶点坐标 (0, 2, 4, 1) 进行变换

$$
\begin{aligned}
\boldsymbol{P}_{\text{world}} &= \boldsymbol{M}_{\text{mods}} \, \boldsymbol{P}_{\text{modsl}} \\
&= \begin{bmatrix} -1.732 & 0 & 1 & 5 \\ 0 & 2 & 0 & 0 \\ -1 & 0 & -1.732 & 25 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 9 \\ 4 \\ 18.072 \\ 1 \end{bmatrix}
\end{aligned}
$$

观察空间 View Space



- 观察空间也被称为**摄像机空间 Camera Space**
- 摄像机决定了我们渲染游戏所使用的视角，在 Unity 中，观察空间使用**右手坐标系**，这这是符合 OpenGL的传统，此时摄像机的前方是指 **-z 轴方向**

观察变换 View Transformation

- 将顶点坐标从世界空间变换到观察空间中
- 为了得到顶点在观察空间中的位置，我们有两种方法
  - 第一种是**计算观察空间的 3 个坐标轴和原点在世界空间下的表示**，构建出从观察空间变换到世界空间的变换矩阵
  - 另一种方法是**平移整个观察空间，使得摄像机的原点位于世界坐标的原点**，坐标轴与世界空间中的坐标轴重合即可

我们这里使用第二种方法

- 由上图摄像机的 Transform 组件可以知道，摄像机在世界空间中先按照 (30, 0, 0) 进行旋转，然后按照 (0, 10, -10) 进行平移
- 为了把摄像机重新移回初始状态，我们要进行逆向变换，即先按照 (0, -10, 10) 平移，再按照 (-30, 0, 0) 进行旋转

$$
\begin{aligned}
\boldsymbol{M}_{\text{visw}} &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \text{cso}\,\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & 0 \\ 0 & -0.5 & 0.866 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -10 \\ 0 & 0 & 1 & 10 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & -3.66 \\ 0 & -0.5 & 0.866 & 13.66 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

- 但是，由于观察空间使用的是右手坐标系，因此要对 z 分类进行取反操作，我们可以通过乘另一个特殊的矩阵来得到最终的观察变换矩阵

$$
\begin{aligned}
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & -3.66 \\ 0 & -0.5 & 0.866 & 13.66 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & -3.66 \\ 0 & 0.5 & -0.866 & -13.66 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

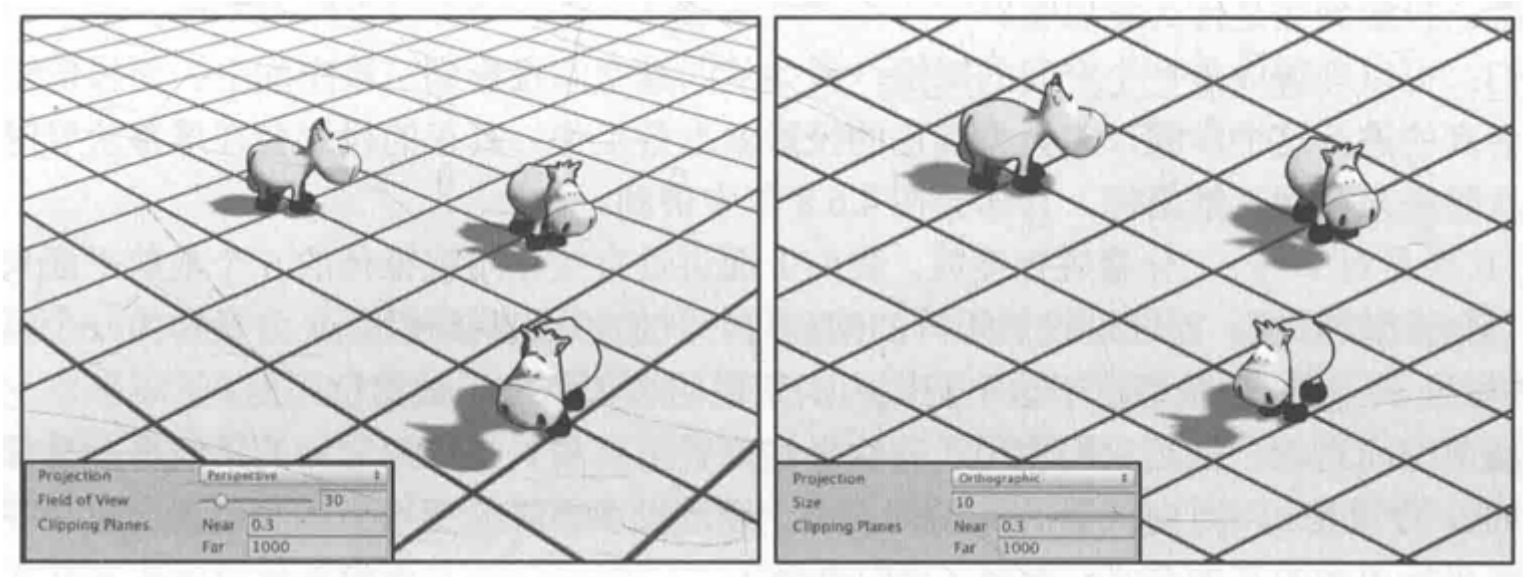
- 现在，我们进行顶点变换

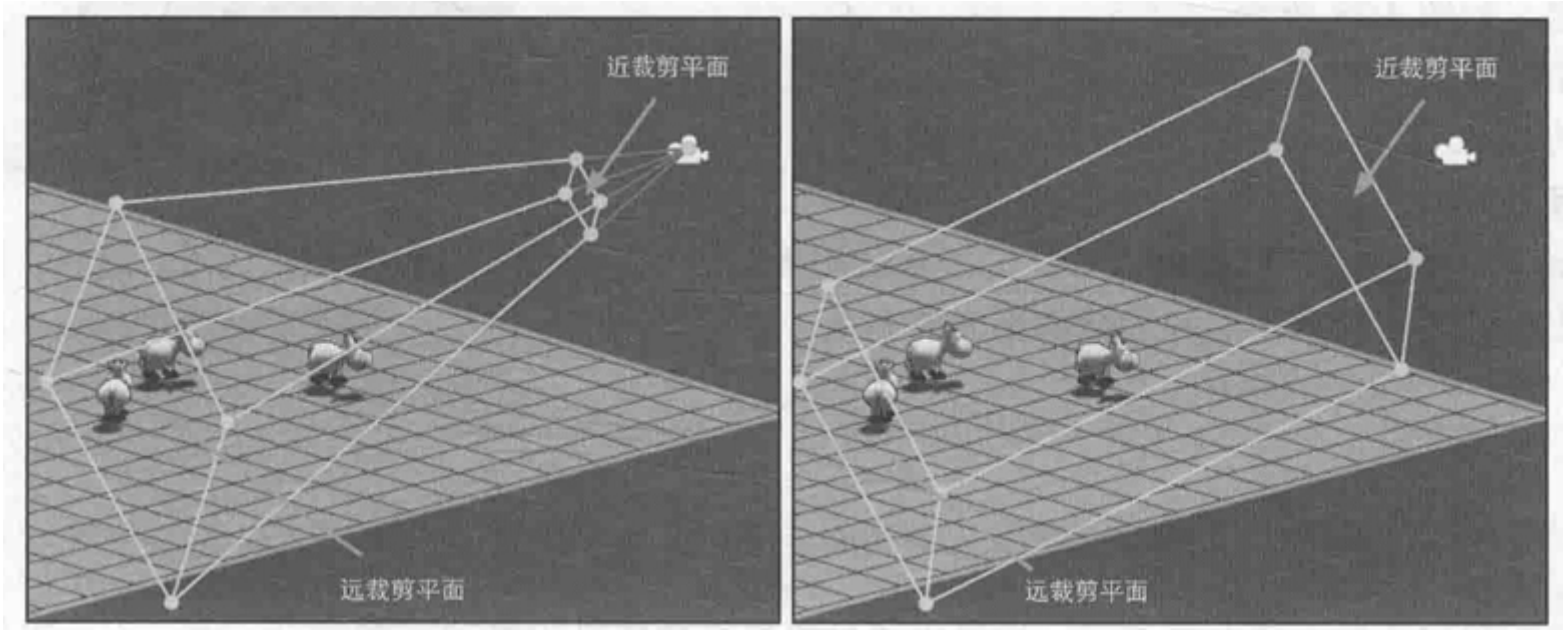
$$
\begin{aligned}
\boldsymbol{P}_{\text{wisw}} &= \boldsymbol{M}_{\text{visw}}\,\boldsymbol{P}_{\text{world}} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & -3.66 \\ 0 & 0.5 & -0.866 & -13.66 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 9 \\ 4 \\ 18.072 \\ 1 \end{bmatrix} = \begin{bmatrix} 9 \\ 8.84 \\ -27.31 \\ 1 \end{bmatrix}
\end{aligned}
$$

裁剪空间 Clip Space

- 裁剪空间的目标是方便对渲染图元进行裁剪：完全位于这块空间内部的图元将会被保留，完全位于这块空间外部的图元将会被剔除，与这块空间边界相交的图元会被裁剪
- 这块空间由**视锥体 view transform** 决定
- 用于将观察空间坐标变换到裁剪空间坐标的矩阵交**裁剪矩阵 clip matrix**，也成为**投影矩阵 projection matrix**

投影矩阵 Projection Matrix



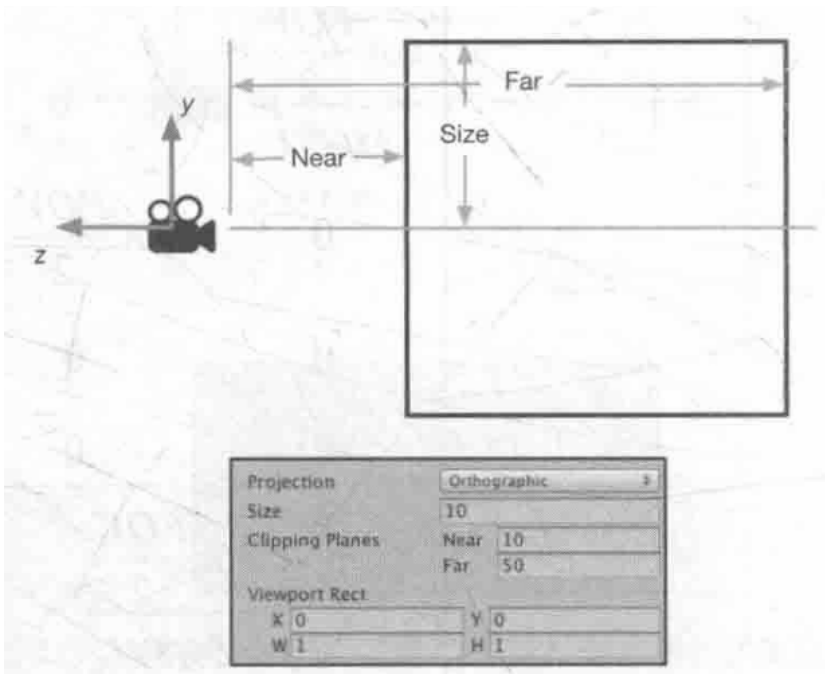


- 在视锥体的 6 块裁剪平面中，有两块裁剪平面比较特殊，它们被称为**近裁剪平面 near clip plane** 和**远裁剪平面 far clip plane**，决定了摄像机可以看到的深度的范围
- 投影矩阵有两个目的
  - **为投影做准备**，投影矩阵并没有进行真正的投影工作，真正的工作发生在后面的齐次除法 homogeneous division 过程中
  - **对 x, y, z 分量进行缩放**，经过缩放后，可以直接使用 w 分量作为一个范围值，如果 x, y, z 分量都位于这个范围内，则表示该定点位于裁剪空间中

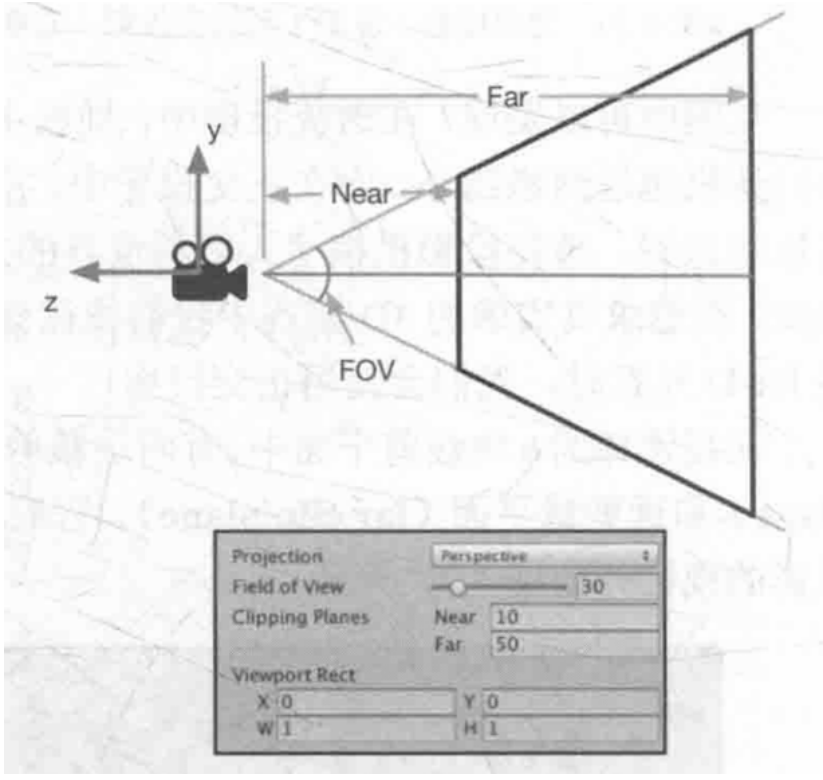
我们使用齐次坐标系来表示点和向量

- 点的 w 分量为 1
- 向量的 w 分量为 0

正交投影 Orthographic Projection



透视投影 Perspective Projection



- 视锥体定义了场景中的一块三维空间，所有位于这块空间的物体都会被渲染，否则会被剔除或裁剪
- 在 Unity 中，它们由 Camera 组件中的参数和 Game 视图的纵横比共同决定
  - FOV：Camera 组件的 Fireld of View 决定
  - Near/Far：Camera 组件中的 Clipping Planes 中的 Near 和 Far 参数来控制

我们可以计算视锥体的近裁剪平面和远裁剪平面的高度：

$$\begin{aligned}\text{nearClipPlaneHeight} &= 2\cdot \text{ Near } \cdot \tan\frac{FOV}{2}\\ \text{farClipPlaneHeight} &= 2\cdot \text{ Far } \cdot \tan\frac{FOV}{2}\end{aligned}$$

现在我们还缺乏横向星系，在 Unity 中，一个摄像机的纵横比由 Game 视图的纵横比和 Viewport Rect 中的 W 和 H 属性共同决定，假设当前摄像机的纵横比为 Aspect，我们定义

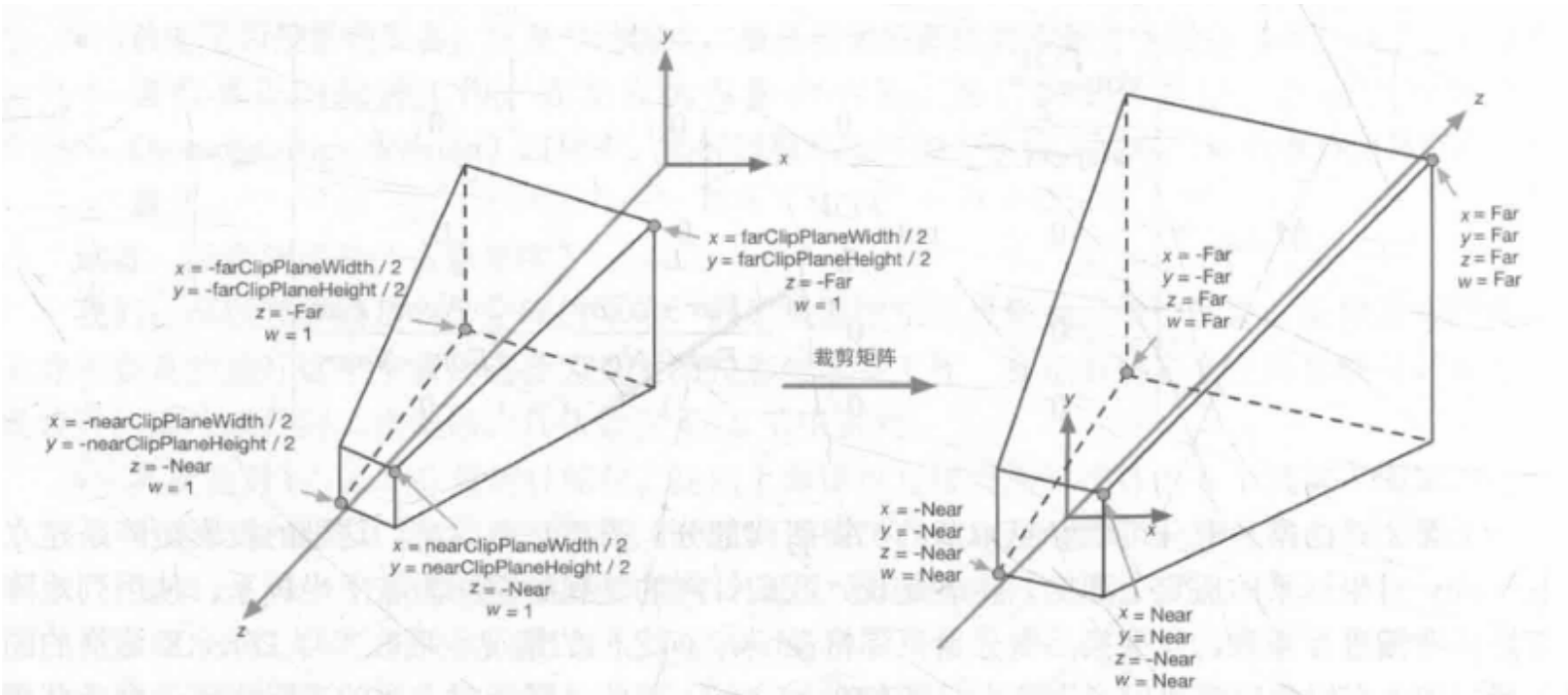
$$\begin{aligned}\text{Aspect} &= \frac{\text{nearClipPlaneWidth}}{\text{nearClipPlaneHeight}}\\ \text{Aspect} &= \frac{\text{farClipPlaneWidth}}{\text{farClipPlaneHeight}}\end{aligned}$$

我们可以根据已知的 Near、Far、FOV 和 Aspect 的值来确定投影矩阵

$$\boldsymbol{M}_{\text{frustum}} = \begin{bmatrix}\frac{\cot\frac{FOV}{2}}{\text{Aspect}} & 0 & 0 & 0\\ 0 & \cot\frac{FOV}{2} & 0 & 0\\ 0 & 0 & -\frac{Far+Near}{Far-Near} & -\frac{2\cdot \text{Near}\cdot \text{Far}}{Far-Near}\\ 0 & 0 & -1 & 0\end{bmatrix}$$

一个顶点与上述的投影矩阵相乘后，可以由观察空间变换到裁剪空间中，结果如下

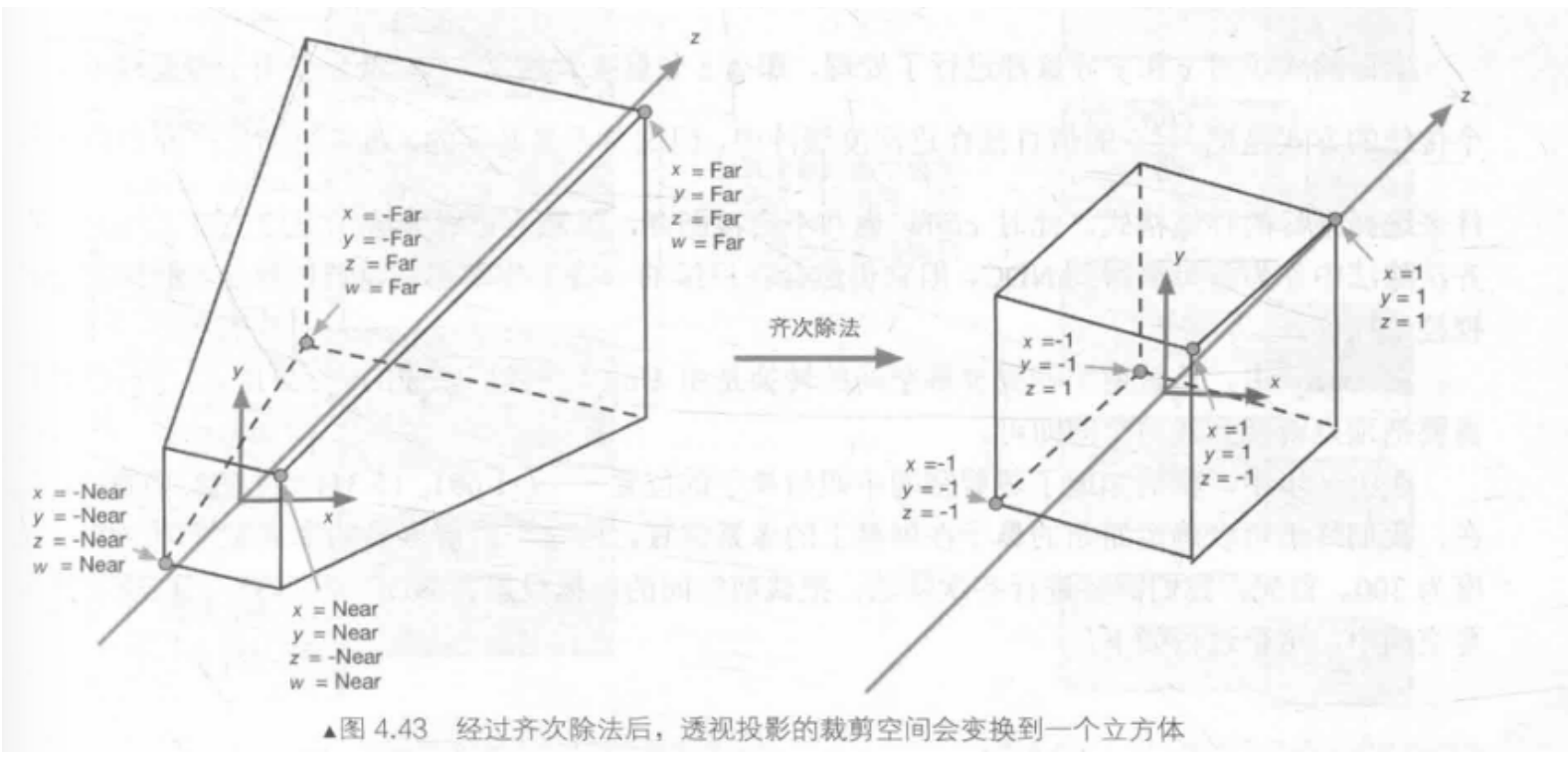
$$\begin{aligned}&= \begin{bmatrix}\frac{\cot\frac{FOV}{2}}{\text{Aspect}} & 0 & 0 & 0\\ 0 & \cot\frac{FOV}{2} & 0 & 0\\ 0 & 0 & \frac{Far+Near}{\text{Far}-\text{Near}} & -\frac{2\cdot \text{Near}\cdot \text{Far}}{\text{Far}-\text{Near}}\\ 0 & 0 & -1 & 0\end{bmatrix}\begin{bmatrix}x\\y\\z\\1\end{bmatrix}\\ &= \begin{bmatrix}x\frac{\cot\frac{FOV}{2}}{\text{Aspect}}\\ y\cot\frac{FOV}{2}\\ -z\frac{Far+Near}{Far-Near}-\frac{2\cdot Near\cdot Far}{Far-Near}\\ -z\end{bmatrix}\end{aligned}$$



▲图 4.39　在透视投影中，投影矩阵对顶点进行了缩放。图 3.38 中标注了 4 个关键点经过投影矩阵变换后的结果。从这些结果可以看出 *x*、*y*、*z* 和 *w* 分量的范围发生的变化

- 注意，此时顶点的 w 分量不再是 1，而是原先 z 分量取反的结果，我们可以用如下不等式**判断变换后的顶点是否位于视锥体内**
  - $-w \leq x \leq w$
  - $-w \leq y \leq w$
  - $-w \leq z \leq w$
- 不满足上述条件的图元将会被剔除或者裁剪
- 裁剪矩阵使得空间从右手坐标系变成了**左手坐标系**，这意味着，离摄像机越远，z 值越大

屏幕空间 Screen Space



▲图 4.43　经过齐次除法后，透视投影的裁剪空间会变换到一个立方体

- 真正的投影，在这一步之后，我们获得真正像素的位置
- 进行**标准齐次除法 homogeneous division**，也被称为**透视除法 perspective division**，用齐次坐标系的 w 分量去除以 x, y, z 分量
  - 在 OpenGL 中被称为**归一化的设备坐标 Normalized Device Coordinates, NDC**
- 齐次除法后变换到一个立方体内，该立方体的 x, y, z 分量的范围均为 [-1. 1]
  - 在 DirectX 中，z 分量的范围是 [0, 1]，Unity 使用 OpenGL

经过齐次除法后，透视投影和正交投影的视锥体都变换到一个相同的立方体内。现在，我们可以根据变换后的 x 和 y 坐标来映射输出窗口的对应像素坐标

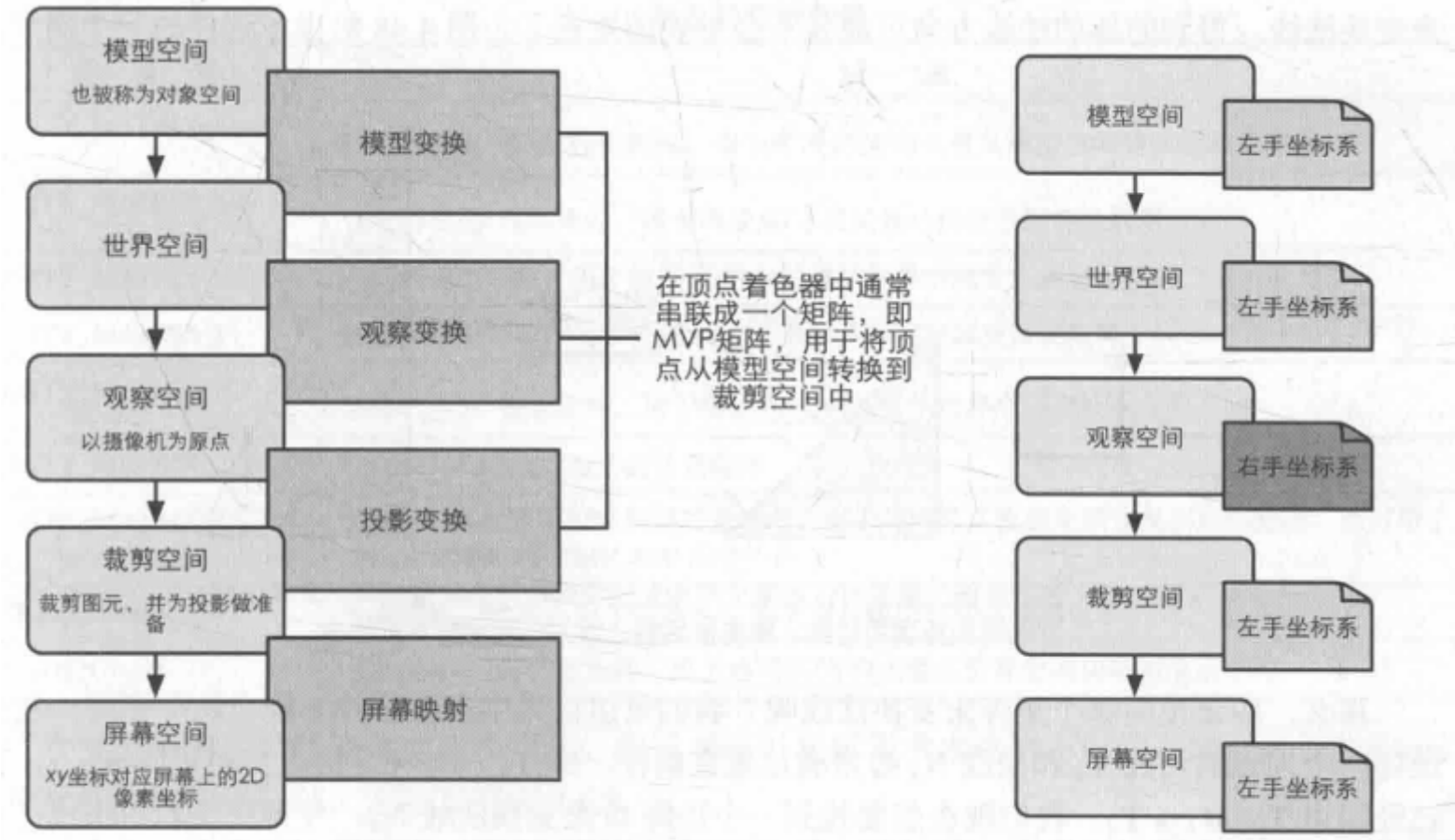
- Unity 中，**屏幕空间左下角的坐标是 (0, 0)**，右上角的像素坐标是 (pixelWidth, pixelHeight)

$$\text{screen}_x = \frac{\text{clip}_x \cdot \text{pixelWidth}}{2 \cdot \text{clip}_w} + \frac{\text{pixelWidth}}{2}$$
$$\text{screen}_y = \frac{\text{clip}_y \cdot \text{pixelHeight}}{2 \cdot \text{clip}_w} + \frac{\text{pixelHeight}}{2}$$

- 在 Unity 中，从裁剪空间到屏幕空间的转换是由 Unity 帮忙完成的

小结

- 顶点着色器最基本的任务就是顶点坐标从模型空间中换到裁剪空间中
- 片元着色器可以得到该片元在屏幕空间的像素位置
- 在 Unity 中，坐标系的旋向性会随着变换而发生改变



▲图 4.45　渲染流水线中顶点的空间变换过程

▲图 4.46　Unity 中各个坐标空间的旋向性

6. 法线变换

7. Unity Shader 内置数学变量

Unity Shaer 提供很多内置参数，使得我们不需要自己再手动计算一些值

变换矩阵