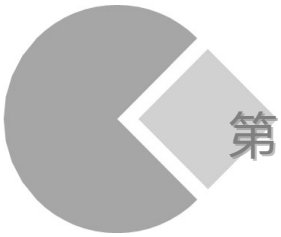
Unity Shader 入门精要

冯乐乐 著

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS



第 4 章　学习 Shader 所需的数学基础

不懂数学者不得入内。
——古希腊柏拉图学院门口的碑文

计算机图形学之所以深奥难懂，很大原因是在于它是建立在虚拟世界上的数学模型。数学渗透到图形学的方方面面，当然也包括 Shader。在学习 Shader 的过程中，我们最常使用的就是矢量和矩阵（即数学的分支之一——线性代数）。

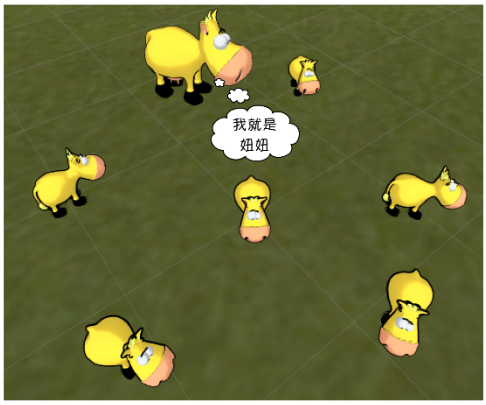
很多读者认为图形学中的数学复杂难懂。的确，一些数学模型在初学者看来晦涩难懂。但很多情况下，我们需要打交道的只是一些基础的数学运算，而只要掌握了这些内容，就会发现很多事情可以迎刃而解。我们在研究和学习他人编写的 Shader 代码时，也不再会疑问：“他为什么要这么写”，而是“哦，这里就是使用矩阵进行了一个变换而已。”

为了让读者能够参与到计算中来，而不是填压式地阅读，在一些小节的最后我们会给出一些练习题。练习题的答案会在本章最后给出（不要偷看答案！）。需要注意的是，这些练习题并不是可有可无的，我们并非想利用题海战术来让读者掌握这些数学运算，而是想利用这些练习题来阐述一些容易出错或实践中常见的问题。通过这些练习题，读者可以对本节内容有更加深刻的理解。

那么，拿起笔来，让我们一起走进数学的世界吧！

4.1 背景：农场游戏

为了让读者更加理解数学计算的几何意义，我们先来假定一个场景。现在，假设我们正在开发一款卡通风格的农场游戏。在这个游戏里，玩家可以在农场里养很多可爱的奶牛。与普通农场游戏不同的是，我们的主角不是玩家，而是一头牛——妞妞，如图 4.1 所示。妞妞不仅长得壮，它对很多事情都充满了好奇心。



▲图 4.1 我们的农场游戏。我们的主角妞妞是一头长得最壮、好奇心很强的奶牛

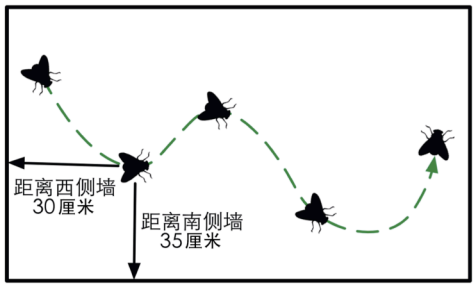
读者：为什么游戏主角不是玩家呢？ 我们：因为我们的策划就是这么任性。

在故事的一开始，农场世界是没有数学概念的。通过下面的学习，我们会见证数学给这个世界带来了怎样翻天覆地的变化。

4.2 笛卡尔坐标系

在游戏制作中，我们使用数学绝大部分都是为了计算位置、距离和角度等变量。而这些计算大部分都是在**笛卡尔坐标系（Cartesian Coordinate System）**下进行的。这个名字来源于法国伟大的哲学家、物理学家、心理学家、数学家笛卡尔（René Descartes ）。

那么，我们为什么需要笛卡尔坐标系呢？有这样一个传说，讲述了笛卡尔提出笛卡尔坐标系的由来。笛卡尔从小体弱多病，所以他所在的寄宿学校的老师允许他可以一直留在床上直到中午。在笛卡尔的一生中，他每天的上午时光几乎都是在床上度过的。笛卡尔并没有把这段时间用在睡懒觉上，而是思考了很多关于数学和哲学上的问题。有一天，笛卡尔发现一只苍蝇在天花板上爬来爬去，他观察了很长一段时间。笛卡尔想：我要如何来描述这只苍蝇的运动轨迹呢？最后，笛卡尔意识到，他可以使用这只苍蝇距离房间内不同墙面的位置来描述，如图 4.2 所示。他从床上起身，写下了他的发现。然后，他试图描述一些点的位置，正如他要描述苍蝇的位置一样。最后，笛卡尔就发明了这个坐标平面。而这个坐标平面后来逐渐发展，就形成了坐标系系统。人们为了纪念笛卡尔的工作，就用他的名字来给这种坐标系进行命名。



▲图 4.2 传说，笛卡尔坐标系来源于笛卡尔对天花板上一只苍蝇的运动轨迹的观察。笛卡尔发现，可以使用苍蝇距不同墙面的距离来描述它的当前位置

当然，上面传说的可靠性无从验证。一些较真儿的读者就不用急着向本书勘误邮箱中发邮件说：“嘿，你简直是胡说！”不过，读者可以从这个传说中发现，笛卡尔坐标系和我们的生活是密切相关的。

4.2.1 二维笛卡尔坐标系

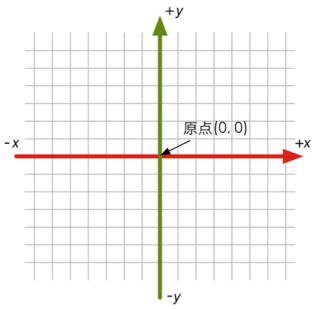
事实上，读者很可能一直在用二维笛卡尔坐标系，尽管你可能并没有听过笛卡尔这个名词。你还记得在《哈利波特与魔法石》电影中，哈利和罗恩大战奇洛教授的魔法棋盘吗？这里的国际

象棋棋盘也可以理解成是一个二维的笛卡尔坐标系。

图 4.3 显示了一个二维笛卡尔坐标系。它是不是很像一个棋盘呢？

一个二维的笛卡尔坐标系包含了两个部分的信息：

- 一个特殊的位置，即原点，它是整个坐标系的中心；
- 两条过原点的互相垂直的矢量，即 x 轴和 y 轴。这些坐标轴也被称为是该坐标系的基矢量。

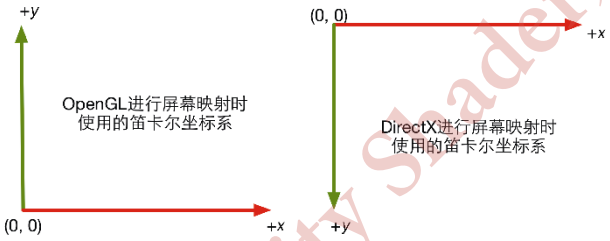


▲图 4.3 一个二维笛卡尔坐标系

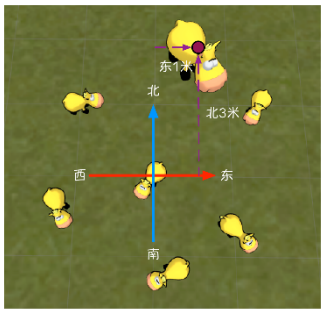
虽然在图 4.3 中 x 轴和 y 轴分别是水平和垂直方向的，但这并不是必须的。想象把上面的坐标系整体向左旋转 30°。而且，虽然图中的 x 轴指向右、y 轴指向上，但这也并不是必须的。例如，在 2.3.4 节屏幕映射中，OpenGL 和 DirectX 使用了不同的二维笛卡尔坐标系，如图 4.4 所示。

而有了这个坐标系我们就可以精确地定位一个点的位置。例如，如果说：“在（1, 2）的位置上画一个点。”那么相信读者肯定知道这个位置在哪里。

我们来看一下笛卡尔坐标系给奶牛农场带来了什么变化。在没有笛卡尔坐标系的时候，奶牛们根本没有明确的位置概念。如果一头奶牛问：“妞妞，你现在在哪里啊？”妞妞只能回答说“我在这里”或者“我在那里”这些模糊的词语。但那头奶牛永远不会知道妞妞的确切位置。而把笛卡尔坐标系引入到奶牛农场后，所有的一切都变得清晰起来。我们把奶牛农场的中心定义成坐标原点，而把地理方向中的东、北定义成坐标轴方向。现在，如果奶牛再问：“妞妞，你现在在哪里啊？”妞妞就可以回答说：“我在东 1 米、北 3 米的地方。”如图 4.5 所示。



▲图 4.4 在屏幕映射时，OpenGL 和 DirectX 使用了不同方向的二维笛卡尔坐标系



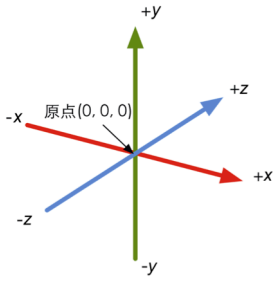
▲图 4.5 笛卡尔坐标系可以让妞妞精确表述自己的位置

4.2.2 三维笛卡尔坐标系

在上面一节中，我们已经了解了二维笛卡尔坐标系。可以看出，二维笛卡尔坐标系实际上是比较简单的。那么，三维比二维只多了一个维度，是不是也就难了 50%而已呢？

不幸的是，答案是否定的。三维笛卡尔坐标系相较于二维来说要复杂许多，但这并不意味着很难学会它。对人类来说，我们生活的世界就是三维的，因此对于理解更低维度的空间（一维和二维）是比较容易的。而对于同等维度的一些概念；理解起来难度就大一些；对于更高维度的空间（如四维空间），理解难度就更大了。

在三维笛卡尔坐标系中，我们需要定义 3 个坐标轴和一个原点。图 4.6 显示了一个三维笛卡尔坐标系。



▲图 4.6 一个三维笛卡尔坐标系

这 3 个坐标轴也被称为是该坐标系的**基矢量（basis vector）**。通常情况下，这 3 个坐标轴之间是互相垂直的，且长度为 1，这样的基矢量被称为**标准正交基（orthonormal basis）**，但这并不是必须的。例如，在一些坐标系中坐标轴之间互相垂直但长度不为 1，这样的基矢量被称为**正交基（orthogonal basis）**。如非特殊说明，本书默认情况下使用的坐标轴指的都是标准正交基。

读者：正交这个词是什么意思呢？

我们：正交可以理解成互相垂直的意思。在下面矩阵的内容中，我们还会看到正交矩阵的概念。

和二维笛卡尔坐标系类似，三维笛卡尔坐标系中的坐标轴方向也不是固定的，即不一定是像图 4.6 中那样的指向。但这种不同导致了两种不同种类的坐标系：**左手坐标系（left-handed coordinate space）**和**右手坐标系（right-handed coordinate space）**。

4.2.3 左手坐标系和右手坐标系

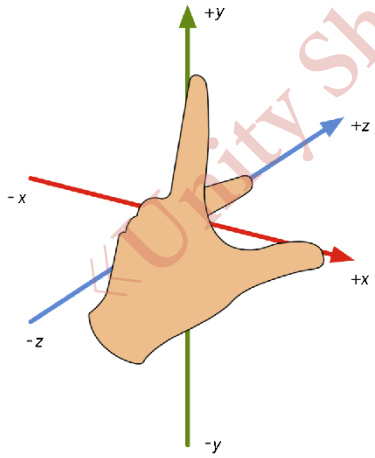
为什么在三维笛卡尔坐标系中要区分左手坐标系和右手坐标系，而二维中就没有这些烦人的事情呢？这是因为，在二维笛卡尔坐标系中，x 轴和 y 轴的指向虽然可能不同，就如我们在图 4.4 中看到的一样。但我们总可以通过一些旋转操作来使它们的坐标轴指向相同。以图 4.4 中 OpenGL 和 DirectX 使用的坐标系为例，为了把右侧的坐标轴指向转换到左侧那样的指向，我们可以首先对右侧的坐标系顺时针旋转 180°，此时它的 y 轴指向上，而 x 轴指向左。然后，我们再把整个纸面水平翻转一下，就可以把 x 轴翻转到指向右了，此时左右两侧的坐标轴指向就完全相同了。从这种意义上来说，所有的二维笛卡尔坐标系都是等价的。

但对于三维笛卡尔坐标系，靠这种旋转有时并不能使两个不同朝向的坐标系重合。例如，在图 4.6 中，$+z$ 轴的方向指向纸面的内部，如果有另一个三维笛卡尔坐标系，它的$+z$ 轴是指向纸面外部，x 轴和 y 轴保持不变，那么我们可以通过旋转把这两个坐标轴重合在一起吗？答案是否定的。我们总可以让其中两个坐标轴的指向重合，但第三个坐标轴的指向总是相反的。

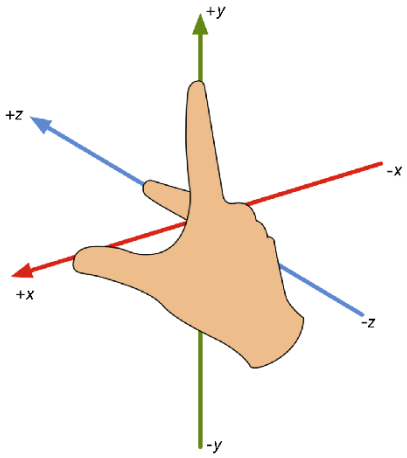
也就是说，三维笛卡尔坐标系并不都是等价的。因此，就出现了两种不同的三维坐标系：左手坐标系和右手坐标系。如果两个坐标系具有相同的**旋向性（handedness）**，那么我们就可以通过旋转的方法来让它们的坐标轴指向重合。但是，如果它们具有不同的旋向性（例如坐标系 A 属于左手坐标系，而坐标系 B 属于右手坐标系），那么就无法达到重合的目的。

那么，为什么叫左手坐标系和右手坐标系呢？和手有什么关系？这是因为，我们可以利用我们的双手来判断一个坐标系的旋向性。请读者举起你的左手，用食指和大拇指摆出一个“L”的手势，并且让你的食指指向上，大拇指指向右。现在，伸出你的中指，不出意外的话它应该指向你的前方（如果你一定要展示自己骨骼惊奇的话我也没有办法）。恭喜你，你已经得到了一个左手坐标系了！你的大拇指、食指和中指分别对应了$+x$、$+y$ 和$+z$ 轴的方向，如图 4.7 所示。

同样，读者可以通过右手来得到一个右手坐标系。举起你的右手，这次食指仍然指向上，中指指向前方，不同的是，大拇指将指向左侧，如图 4.8 所示。



▲图 4.7 左手坐标系



▲图 4.8 右手坐标系

正如我们之前所说，左手坐标系和右手坐标系之间无法通过旋转来同时使它们的 3 个坐标轴指向重合，如果你不信，你现在可以拿自己的双手来试验一下。

另外一个确定是左手还是右手坐标系的方法是，判断**前向（forward）**的方向。请读者坐直，

向右伸直你的右手，此时右手方向就是 x 轴的正向，而你的头顶向上的方向就是 y 轴的正向。这时，如果你的正前方的方向是 z 轴的正向，那么你本身所在的坐标系就是一个左手坐标系；如果你的正前方的方向对应的是 z 轴的负向，那么这就是一个右手坐标系。

除了坐标轴朝向不同之外，左手坐标系和右手坐标系对于正向旋转的定义也不同，即在初高中物理中学到的**左手法则（left-hand rule）**和**右手法则（right-hand rule）**。假设现在空间中有一条直线，还有一个点，我们希望把这个点以该直线为旋转轴旋转某个角度，比如旋转 30°。读者可以拿一支笔当成这个旋转轴，再拿自己的手当成这个需要旋转的点，可以发现，我们有两个旋转方向可以选择。那么，我们应该往哪个方向旋转呢？这意味着，我们需要在坐标系中定义一个旋转的正方向。在左手坐标系中，这个旋转正方向是由左手法则定义的，而在右手坐标系中则是由右手法则定义的。

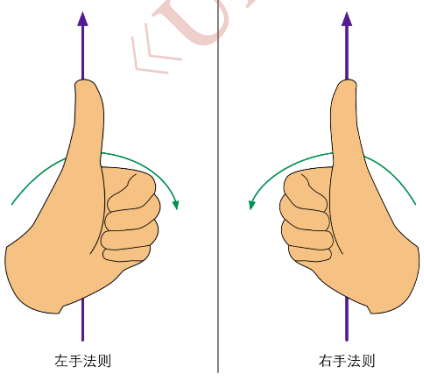
在左手坐标系中，我们可以这样来应用左手法则：还是举起你的左手，握拳，伸出大拇指让它指向旋转轴的正方向，那么旋转的正方向就是剩下 4 个手指的弯曲方向。在右手坐标系中，使用右手法则对旋转正方向的判断类似。如图 4.9 所示。

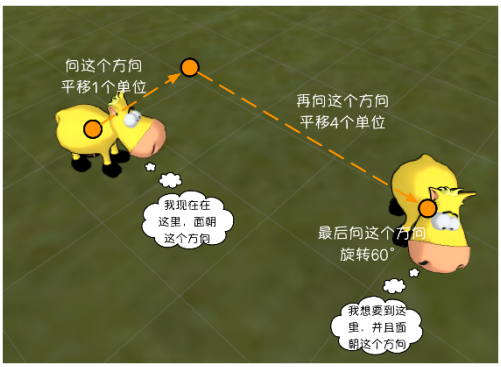
从图 3.9 中可以看出，在左手坐标系中，旋转正方向是顺时针的，而在右手坐标系中，旋转正方向是逆时针的。

左右手坐标系之间是可以进行互相转换的。最简单的方法就是把其中一个轴反转，并保持其他两个轴不变。

对于开发者来说，使用左手坐标系还是右手坐标系都是可以的，它们之间并没有优劣之分。无论使用哪种坐标系，绝大多数情况下并不会影响底层的数学运算，而只是在映射到视觉上时会有差别（见练习题 2）。这是因为，一个点或者旋转在空间内来说是绝对的。一些较真儿读者可能会看不惯“绝对”这个词：“你怎么能忽略相对论呢？这世上一切都是相对的！”这些读者请容我解释。这里所说的绝对是说，在我们所关心的最广阔的空间中，这些值是绝对的。例如我说，把你的书从桌子的左边移到右边，你不会对这个过程产生什么疑问，此时我们关心的整个空间就是桌子这个空间，而在这个空间中，书的运动是绝对的。但是，在数学的世界中，我们需要使用一种数学模型来精确地描述它们，这个模型就是坐标系。一旦有了坐标系，每个点的位置就不再是绝对的，而是相对于这个坐标系来说的。这种相对关系导致，即便从数学表示上来说两种表示方式完全一样，但从视觉上来说是不一样的。

我们可以在奶牛农场的例子中体会左手坐标系和右手坐标系的分别。我们假设，妞妞想要到一个新的地方，因为那里的草很美味。妞妞知道到达这个目标点的“绝对路径”是怎样的，如图 4.10 所示。

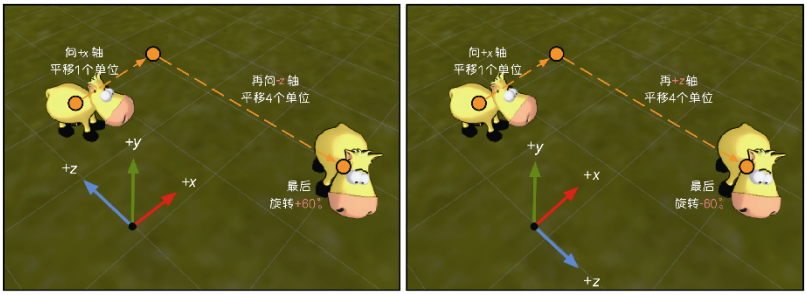




▲图 4.9 用左手法则和右手法则来判断旋转正方向

▲图 4.10 为了移动到新的位置，妞妞需要首先向某个方向平移 1 个单位，再向另一个方向平移 4 个单位，最后再向一个方向旋转 60°。

我们可以分别在一个左手坐标系和右手坐标系中描述这样一次运动，即使用数学表达式来描述它。我们会发现，在不同的坐标系中描述这样同一次运动是不一样的，如图 4.11 所示。



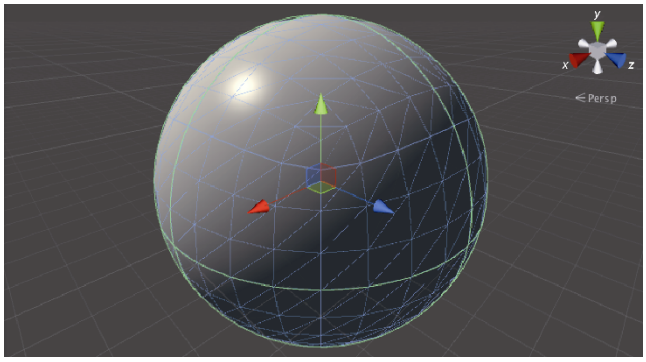
▲图 4.11 左图和右图分别表示了在左手坐标系和右手坐标系中描述妞妞这次运动的结果，得到的数学描述是不同的

在左手坐标系中，3 个坐标轴的朝向如图 4.11 左图所示。妞妞首先向 x 轴正方向平移 1 个单位，然后再向 z 轴负方向移动 4 个单位，最后朝旋转的正方向旋转 60°。而在右手坐标系中，$+z$ 轴的方向和左手坐标系中刚好相反，因此妞妞首先向 x 轴正方向平移 1 个单位（与左手坐标系中的移动一致），然后再向 z 轴正方向移动 4 个单位（与左手坐标系中的移动相反），最后朝旋转的负方向旋转 60°（与左手坐标系中的旋转相反）。

可以看出，为了达到同样的视觉效果（这里指把妞妞移动到视觉上的同一个位置），左右手坐标系在 z 轴上的移动以及旋转方向是不同的。如果使用相同的数学运算（指均向 z 轴某方向移动或均朝旋转正方向旋转等），那么得到的视觉效果就是不一样的。因此，如果我们需要从左手坐标系迁移到右手坐标系，并且保持视觉上的不变，就需要进行一些转换。读者可以参见本章最后的扩展阅读部分。

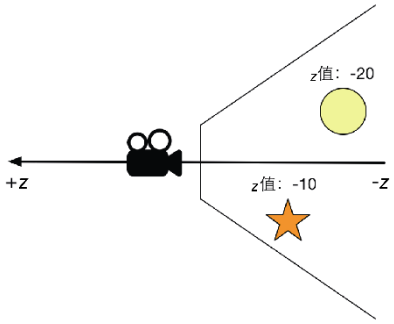
4.2.4 Unity 使用的坐标系

对于一个需要可视化虚拟的三维世界的应用（如 Unity）来说，它的设计者就要进行一个选择。对于模型空间和世界空间（在 4.6 节中会具体讲解这两个空间是什么），Unity 使用的是左手坐标系。这可以从 Scene 视图的坐标轴显示看出来，如图 4.12 所示。这意味着，在模型空间中，一个物体的右侧（right）、上侧（up）和前侧（forward）分别对应了 x 轴、y 轴和 z 轴的正方向。



▲图 4.12 在模型空间和世界空间中，Unity 使用的是左手坐标系。图中，球的坐标轴显示了它在模型空间中的 3 个坐标轴（红色为 *x* 轴，绿色是 *y* 轴，蓝色是 *z* 轴）

但对于观察空间来说，Unity 使用的是右手坐标系。观察空间，通俗来讲就是以摄像机为原点的坐标系。在这个坐标系中，摄像机的前向是 z 轴的负方向，这与在模型空间和世界空间中的定义相反。也就是说，*z* 轴坐标的减少意味着场景深度的增加，如图 4.13 所示。



▲图 4.13 在 Unity 中，观察空间使用的是右手坐标系，摄像机的前向是 *z* 轴的负方向，*z* 轴越小，物体的深度越大，离摄像机越远

关于 Unity 中使用的坐标系的旋向性，我们会在 4.5.9 节中详细地讲解。

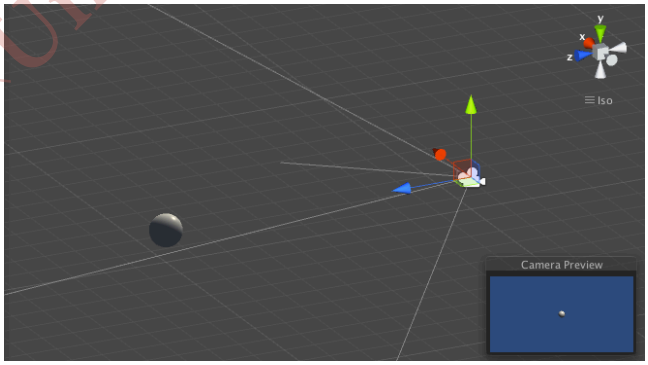
4.2.5 练习题

这是本书中第一次出现练习题的地方，希望你可以快速解决它们！

1．在非常流行的建模软件 3ds Max 中，默认的坐标轴方向是：*x* 轴正方向指向右方，*y* 轴正方向指向前方，*z* 轴正方向指向上方。那么它是左手坐标系还是右手坐标系？

2．在左手坐标系中，有一点的坐标是（0, 0, 1），如果把该点绕 *y* 轴正方向旋转+90°，旋转后的坐标是什么？如果是在右手坐标系中，同样有一点坐标为(0, 0, 1)，把它绕 *y* 轴正方向旋转+90°，旋转后的坐标是什么？

3．在 Unity 中，新建的场景中主摄像机的位置位于世界空间中的（0, 1, −10）位置。在不改变摄像机的任何设置（如保持 Rotation 为(0, 0, 0)，Scale 为（1, 1, 1））的情况下，在世界空间中的(0, 1, 0)位置新建一个球体，如图 4.14 所示。



▲图 4.14 摄像机的位置是（0, 1, -10），球体的位置是（0, 1, 0）

在摄像机的观察空间下，该球体的 *z* 值是多少？在摄像机的模型空间下，该球体的 *z* 值又是

多少？

4.3 点和矢量

点（point）是 n 维空间（游戏中主要使用二维和三维空间）中的一个位置，它没有大小、宽度这类概念。在笛卡尔坐标系中，我们可以使用 2 个或 3 个实数来表示一个点的坐标，如：$P = (P_x, P_y)$，表示二维空间的点，$P = (P_x, P_y, P_z)$，来表示三维空间中的点。

矢量（vector，也被称为向量）的定义则复杂一些。在数学家看来，矢量就是一串数字。你可能要问了，点的表达式不也是一串数字吗？没错，但矢量存在的意义更多是为了和**标量（scalar）**区分开来。通常来讲，矢量是指 n 维空间中一种包含了**模（magnitude）**和**方向（direction）的有向线段**，我们通常讲到的速度（velocity）就是一种典型的矢量。例如，这辆车的速度是向南 80km/h（向南指明了矢量的方向，80km/h 指明了矢量的模）。而标量只有模没有方向，生活中常常说到的距离（distance）就是一种标量。例如，我家离学校只有 200 米（200 米就是一个标量）。

具体来讲。

- 矢量的模指的是这个矢量的长度。一个矢量的长度可以是任意的非负数。
- 矢量的方向则描述了这个矢量在空间中的指向。

矢量的表示方法和点类似。我们可以使用 $\mathbf{v} = (x, y)$来表示二维矢量，用 $\mathbf{v} = (x, y, z)$来表示三维矢量，用 $\mathbf{v} = (x, y, z, w)$来表示四维矢量。

为了方便阐述，我们对不同类型的变量在书写和印刷上使用不同的样式：

- 对于标量，我们使用小写字母来表示，如 a，b，x，y，z，θ，α等；
- 对于矢量，我们使用小写的粗体字母来表示，如 **a**，**b**，**u**，**v** 等；
- 对于后面要学习的矩阵，我们使用大写的粗体字母来表示，如 **A**，**B**，**S**，**M**，**R** 等。

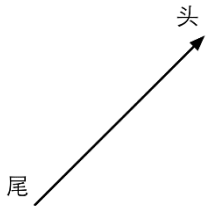
在图 4.15 中，一个矢量通常由一个箭头来表示。我们有时会讲到一个矢量的**头（head）**和**尾（tail）**。矢量的头指的是它的箭头所在的端点处，而尾指的是另一个端点处，如图 4.15 所示。

那么一个矢量要放在哪里呢？从矢量的定义来看，它只有模和方向两个属性，并没有位置信息。这听起来很难理解，但实际上在生活中我们总是会和这样的矢量打交道。例如，当我们讲到一个物体的速度时，可能会这样说“那个小偷正在以 100km/h 的速度向南逃窜”（快抓住他！），这里的“以 100km/h 的速度向南”就可以使用一个矢量来表示。通常，矢量被用于表示相对于某个点的**偏移（displacement）**，也就是说它是一个相对量。只要矢量的模和方向保持不变，无论放在哪里，都是同一个矢量。

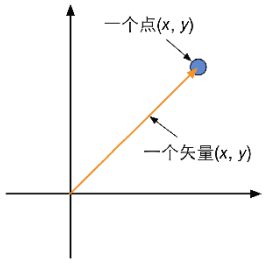
4.3.1 点和矢量的区别

回顾一下，点是一个没有大小之分的空间中的位置，而矢量是一个有模和方向但没有位置的量。从这里看，点和矢量具有不同的意义。但是，从表示方式上两者非常相似。

在上一节中我们提到，矢量通常用于描述偏移量，因此，它们可以用于描述相对位置，即相对于另一个点的位置，此时矢量的尾是一个位置，那么矢量的头就可以表示另一个位置了。而一个点可以用于指定空间中的一个位置（即相对于原点的位置）。如果我们把矢量的尾固定在坐标系原点，那么这个矢量的表示就和点的表示重合了。图 4.16 表示了两者之间的关系。



▲图 4.15 一个二维向量以及它的头和尾



▲图 4.16 点和矢量之间的关系

尽管上面的内容看起来显而易见，但区分点和矢量之间的不同是非常重要的，尽管它们在数学表达式上是一样的，都是一串数字而已。如果一定要给它们之间建立一个联系的话，我们可以认为，任何一个点都可以表示成一个从原点出发的矢量。为了明确点和矢量的区别，在本书后面的内容中，我们将用于表示方向的矢量称为方向矢量。

4.3.2 矢量运算

在下面的内容里，我们将给出一些最常见的矢量运算。幸运的是，这些运算大的很好理解。对于每种运算，我们会先给出数学上的描述，然后再给出几何意义上的解释。同样，为了让读者加深印象，我们会在最后给出一些练习题。相信读完本节后，你一定可以快速地解决它们！

1．矢量和标量乘法/除法

还记得吗？标量是只有模没有方向的量，虽然我们不能把矢量和标量进行相加/相减的运算（想象一下，你会把速度和距离相加吗），但可以对它们进行乘法运算，结果会得到一个不同长度且可能方向相反的新的矢量。

公式非常简单，我们只需要把矢量的每个分量和标量相乘即可：

$$k\mathbf{v} = (kv_x, kv_y, kv_z)$$

类似的，一个矢量也可以被一个非零的标量除。这等同于和这个标量的倒数相乘：

$$\frac{\mathbf{v}}{k} = \frac{(x,y,z)}{k} = \frac{1}{k}(x,y,z) = \left(\frac{x}{k}, \frac{y}{k}, \frac{z}{k}\right), k \neq 0$$

下面给出一些例子：

$$2(1, 2, 3) = (2, 4, 6)$$
$$-3.5(2, 0) = (-7, 0)$$
$$\frac{(1,2,3)}{2} = (0.5,1,1.5)$$

注意，对于乘法来说，矢量和标量的位置可以互换。但对于除法，只能是矢量被标量除，而不能是标量被矢量除，这是没有意义的。

从几何意义上看，把一个矢量 $\mathbf{v}$ 和一个标量 k 相乘，意味着对矢量 $\mathbf{v}$ 进行一个大小为$|k|$的缩放。例如，如果想要把一个矢量放大两倍，就可以乘以 2。当 $k < 0$ 时，矢量的方向也会取反。图 4.17 显示了这样的一些例子。

2．矢量的加法和减法

我们可以对两个矢量进行相加或相减，其结果是一个相同维度的新矢量。

我们只需要把两个矢量的对应分量进行相加或相减即可。公式如下：

$$\mathbf{a} + \mathbf{b} = (a_x + b_x, a_y + b_y, a_z + b_z)$$

$$\mathbf{a} - \mathbf{b} = (a_x - b_x, a_y - b_y, a_z - b_z)$$

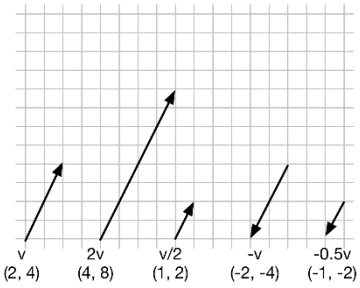
下面是一些例子：

$$(1, 2, 3) + (4, 5, 6) = (5, 7, 9)$$

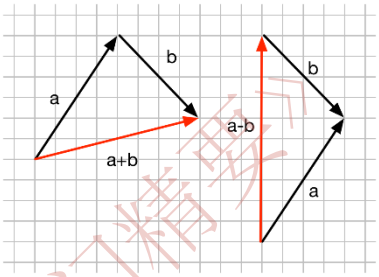
$$(5, 2, 7) - (3, 8, 4) = (2, -6, 3)$$

需要注意的是，一个矢量不可以和一个标量相加或相减，或者是和不同维度的矢量进行运算。

从几何意义上来看，对于加法，我们可以把矢量 **a** 的头连接到矢量 **b** 的尾，然后画一条从 **a** 的尾到 **b** 的头的矢量，来得到 **a** 和 **b** 相加后的矢量。也就是说，如果我们从一个起点开始进行了一个位置偏移 **a**，然后又进行一个位置偏移 **b**，那么就等同于进行了一个 **a+b** 的位置偏移。这被称为矢量加法的**三角形定则（triangle rule）**。矢量的减法是类似的。如图 4.18 所示。



▲图 4.17　二维矢量和一些标量的乘法和除法



▲图 4.18　二维矢量的加法和减法

读者需要时刻谨记，在图形学中矢量通常用于描述位置偏移（简称位移）。因此，我们可以利用矢量的加法和减法来计算一点相对于另一点的位移。

假设，空间内有两点 *a* 和 *b*。还记得吗，我们可以用矢量 **a** 和 **b** 来表示它们相对于原点的位移。如果我们想要计算点 *b* 相对于点 *a* 的位移，就可以通过把 **b** 和 **a** 相减得到，如图 4.19 所示。

3．矢量的模

正如我们之前讲到的一样，矢量是有模和方向的。矢量的模是一个标量，可以理解为是矢量在空间中的长度。它的表示符号通常是在矢量两旁分别加上一条垂直线（有的文献中会使用两条垂直线）。三维矢量的模的计算公式如下：

$$|\mathbf{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

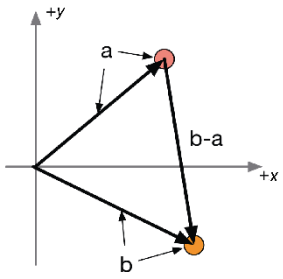
其他维度的矢量的模计算类似，都是对每个分量的平方相加后再开根号得到。

下面给出一些例子：

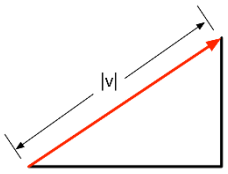
$$|(1,2,3)| = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{1+4+9} = \sqrt{14} \approx 3.742$$

$$|(3,4)| = \sqrt{3^2 + 4^2} = \sqrt{9+16} = \sqrt{25} = 5$$

我们可以从几何意义来理解上述公式。对于二维矢量来说，我们可以对任意矢量构建一个三角形，如图 4.20 所示。



▲图 4.19 使用矢量减法来计算从点 a 到点 b 的位移



▲图 4.20 矢量的模

从 4.20 图可以看出，对于二维矢量，其实就是使用了勾股定理，矢量的两个分量的绝对值对应了三角形两个直角边的长度，而斜边的长度就是矢量的模。

4．单位矢量

在很多情况下，我们只关心矢量的方向而不是模。例如，在计算光照模型时，我们往往需要得到顶点的法线方向和光源方向，此时我们不关心这些矢量有多长。在这些情况下，我们就需要计算**单位矢量（unit vector）**。

单位矢量指的是那些模为 1 的矢量。单位矢量也被称为**被归一化的矢量（normalized vector）**。对任何给定的非零矢量，把它转换成单位矢量的过程就被称为**归一化（normalization）**。

给定任意非零矢量 **v**，我们可以计算和 **v** 方向相同的单位矢量。在本书中，我们通过在一个矢量的头上添加一个戴帽符号来表示单位矢量，例如 $\hat{\mathbf{v}}$。为了对矢量进行归一化，我们可以用矢量的模除以该矢量来得到。公式如下：

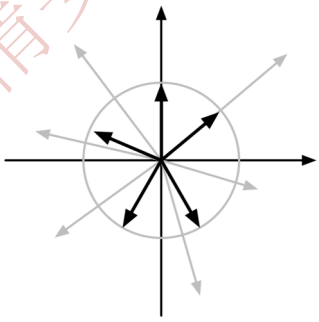
$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{|\mathbf{v}|}, \mathbf{v} \text{ 是任意非零矢量}$$

下面给出一些例子：

$$\frac{(3,-4)}{|(3,-4)|} = \frac{(3,-4)}{\sqrt{3^2+(-4)^2}} = \frac{(3,-4)}{\sqrt{25}} = \frac{(3,-4)}{5} = \left(\frac{3}{5},\frac{-4}{5}\right) = (0.6,-0.8)$$

零矢量（即矢量的每个分量值都为 0，如 **v** = (0, 0, 0)）是不可以被归一化的。这是因为做除法运算时分母不能为 0。

从几何意义上看，对二维空间来说，我们可以画一个单位圆，那么单位矢量就可以是从圆心出发、到圆边界的矢量。在三维空间中，单位矢量就是从一个单位球的球心出发、到达球面的矢量。图 4.21 给出了二维空间内的一些单位矢量。



▲图 4.21 二维空间的单位矢量都会落在单位圆上

需要注意的是，在后面的章节中我们将会不断遇到法线方向（也被称为法矢量）、光源方向等，这些矢量不一定是归一化后的矢量。由于我们的计算往往要求矢量是单位矢量，因此在使用前应先对这些矢量进行归一化运算。

5．矢量的点积

矢量之间也可以进行乘法，但是和标量之间的乘法有很大不同。矢量的乘法有两种最常用的种类：**点积（dot product，也被称为内积，inner product）**和**叉积（cross product，也被称为外积，outer product）**。在本节中，我们将讨论第一种类型：点积。

读者可能认为上面几节的内容都很简单，“这些都显而易见嘛”。那么从这一节开始，我们就会遇到一些真正需要花费力气（真的只要一点点）去记忆的公式。幸运的是，绝大多数公式是有几何意义的，也就是说，我们可以通过画图的方式来理解和帮助记忆。

比仅仅记住这些公式更加重要的是，我们要真正理解它们是做什么的。只有这样，我们才能在需要时想起来，“噢，这个需求我可以用这个公式来实现！”在我们编写 Shader 的过程中，通常程序接口都会提供这些公式的实现，因此我们往往不需要手工输入这些公式。例如，在 Unity Shader 中，我们可以直接使用形如 dot(a, b)的代码来对两个矢量值进行点积的运算。

点积的名称来源于这个运算的符号：**a**·**b**。中间的这个圆点符号是不可以省略的。点积的公式有两种形式，我们先来看第一种。两个三维矢量的点积是把两个矢量对应分量相乘后再取和，最后的结果是一个标量。

公式一：

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

下面是一些例子：

$$(1, 2, 3)\cdot(0.5, 4, 2.5) = 0.5 + 8 + 7.5 = 16$$
$$(-3, 4, 0)\cdot(5, -1, 7) = -15 + -4 + 0 = -19$$

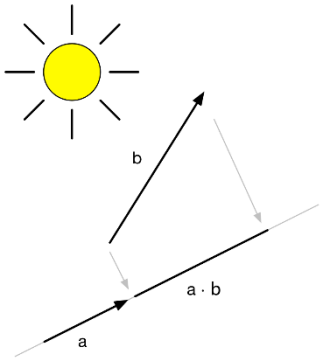
矢量的点积满足交换律，即 $\mathbf{a}\cdot\mathbf{b} = \mathbf{b}\cdot\mathbf{a}$

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）**。

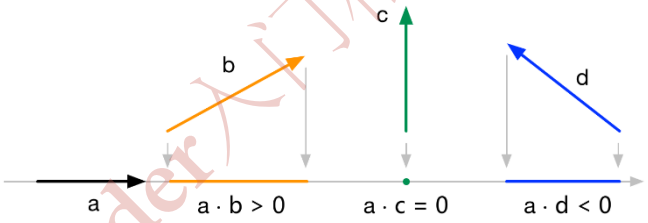
假设，有一个单位矢量 $\hat{\mathbf{a}}$ 和另一个长度不限的矢量 **b**。现在，我们希望得到 **b** 在平行于 $\hat{\mathbf{a}}$ 的一条直线上的投影。那么，我们就可以使用点积 $\hat{\mathbf{a}}\cdot\mathbf{b}$ 来得到 **b** 在 $\hat{\mathbf{a}}$ 方向上的有符号的投影。

那么，投影到底是什么意思呢？这里给出一个通俗的解释。我们可以认为，现在有一个光源，它发出的光线是垂直于 $\hat{\mathbf{a}}$ 方向的，那么 **b** 在 $\hat{\mathbf{a}}$ 方向上的投影就是 **b** 在 $\hat{\mathbf{a}}$ 方向上的影子，如图 4.22 所示。

需要注意的是，投影的值可能是负数。投影结果的正负号与 $\hat{\mathbf{a}}$ 和 **b** 的方向有关：当它们的方向相反（夹角大于 90°）时，结果小于 0；当它们的方向互相垂直（夹角为 90°）时，结果等于 0；当它们的方向相同（夹角小于 90°）时，结果大于 0。图 4.23 给出了这 3 种情况的图示。



▲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。

也就是说，点积的符号可以让我们知道两个矢量的方向关系。

那么，如果 $\hat{\mathbf{a}}$ 不是一个单位矢量会如何呢？这很容易想到，任何两个矢量的点积 $\mathbf{a}\cdot\mathbf{b}$ 等同于 **b** 在 **a** 方向上的投影值，再乘以 **a** 的长度。

点积具有一些很重要的性质，在 Shader 的计算中，我们会经常利用这些性质来帮助计算。

性质一：点积可结合标量乘法。

上面的“结合”是说，点积的操作数之一可以是另一个运算的结果，即矢量和标量相乘的结果。公式如下：

$$(k\mathbf{a})\cdot\mathbf{b} = \mathbf{a}\cdot(k\mathbf{b}) = k(\mathbf{a}\cdot\mathbf{b})$$

也就是说，对点积中其中一个矢量进行缩放的结果，相当于对最后的点积结果进行缩放。

性质二：点积可结合矢量加法和减法，和性质一类似。

这里的“结合”指的是，点积的操作数可以是矢量相加或相减后的结果。用公式表达就是：

$$\mathbf{a}\cdot(\mathbf{b} + \mathbf{c}) = \mathbf{a}\cdot\mathbf{b} + \mathbf{a}\cdot\mathbf{c}$$

把上面的 **c** 换成−**c** 就可以得到减法的版本。

性质三：一个矢量和本身进行点积的结果，是该矢量的模的平方。

这点可以很容易从公式验证得到：

$$\mathbf{v}\cdot\mathbf{v}=v_xv_x+v_yv_y+v_zv_z=|\mathbf{v}|^2$$

这意味着，我们可以直接利用点积来求矢量的模，而不需要使用模的计算公式。当然，我们需要对点积结果进行开平方的操作来得到真正的模。但很多情况下，我们只是想要比较两个矢量的长度大小，因此可以直接使用点积的结果。毕竟，开平方的运算需要消耗一定性能。

现在是时候来看点积的另一种表示方法了。这种方法是从三角代数的角度出发的，这种表示方法更加具有几何意义，因为它可以明确地强调出两个矢量之间的角度。

我们先直接给出第二个公式。

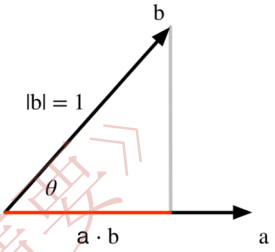
公式二：

$$\mathbf{a}\cdot\mathbf{b}=|\mathbf{a}||\mathbf{b}|\cos\theta$$

初看之下，似乎和公式一没有什么联系，怎么会相等呢？我们先来看最简单的情况。假设，我们对两个单位矢量进行点积，即 $\hat{\mathbf{a}}\cdot\hat{\mathbf{b}}$，如图 4.24 所示。

到了产生魔法的时间了！我们知道 $\hat{\mathbf{b}}$ 的模为 1，且读者应该记得 $\cos\theta=\frac{\text{直角边}}{\text{斜边}}$。我们可以发现，图中 $\hat{\mathbf{a}}\cdot\hat{\mathbf{b}}$ 的结果刚好就是 $\cos\theta$对应的直角边。因此，由图 4.24 可以得到：

$$\hat{\mathbf{a}}\cdot\hat{\mathbf{b}}=\frac{\text{直角边}}{\text{斜边}}=\cos\theta$$



▲图 4.24 两个单位矢量进行点积

这也就是说，两个单位矢量的点积等于它们之间夹角的余弦值。再应用性质一就可以得到公式二了：

$$\mathbf{a}\cdot\mathbf{b}=(|\mathbf{a}|\hat{\mathbf{a}})\cdot(|\mathbf{b}|\hat{\mathbf{b}})=|\mathbf{a}||\mathbf{b}|(\hat{\mathbf{a}}\cdot\hat{\mathbf{b}})=|\mathbf{a}||\mathbf{b}|\cos\theta$$

也就是说，两个矢量的点积可以表示为两个矢量的模相乘，再乘以它们之间夹角的余弦值。从这个公式也可以看出，为什么计算投影时两个矢量的方向不同会得到不同符号的投影值：当夹角小于 90°时，$\cos\theta>0$；当夹角等于 90°时，$\cos\theta=0$；当夹角大于 90°时，$\cos\theta<0$。

利用这个公式我们还可以求得两个向量之间的夹角（在 0°～180°）：

$\theta=\arccos(\hat{\mathbf{a}}\cdot\hat{\mathbf{b}})$，假设 $\hat{a}$ 和 $\hat{b}$ 是单位矢量。

其中，arcos 是反余弦操作。

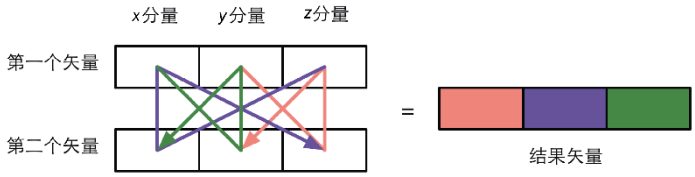
6．矢量的叉积

另一个重要的矢量运算就是**叉积（cross product）**，也被称为**外积（outer product）**。与点积不同的是，矢量叉积的结果仍是一个矢量，而非标量。

和点积类似，叉积的名称来源于它的符号：$\mathbf{a}\times\mathbf{b}$。同样，这个叉号也是不可省略的。两个矢量的叉积可以用如下公式计算：

$$\mathbf{a}\times\mathbf{b}=(a_x,a_y,a_z)\times(b_x,b_y,b_z)=(a_yb_z-a_zb_y,a_zb_x-a_xb_z,a_xb_y-a_yb_x)$$

上面的公式看起来很复杂，但其实是有一定规律的。图 4.25 给出了这样的规律图示。



▲图 4.25 三维矢量叉积的计算规律。不同颜色的线表示了计算结果矢量中对应颜色的分量的计算路径。以红色为例，即结果矢量的第一个分量，它是从第一个矢量的 y 分量出发乘以第二个矢量的 z 分量，再减去第一个矢量的 z 分量和第二矢量的 y 分量的乘积

例如：

$$(1, 2, 3)\times(-2, -1, 4) = ((2)(4) - (3)(-1), (3)(-2) - (1)(4), (1)(-1) - (2)(-2))$$
$$= (8 - (-3), (-6) - 4, (-1) - (-4)) = (11, -10, 3)$$

需要注意的是，叉积不满足交换律，即 $\mathbf{a}\times\mathbf{b} \neq \mathbf{b}\times\mathbf{a}$。实际上，叉积是满足反交换律的，即 $\mathbf{a}\times\mathbf{b} = -(\mathbf{b}\times\mathbf{a})$。而且叉积也不满足结合律，即 $(\mathbf{a}\times\mathbf{b})\times\mathbf{c} \neq \mathbf{a}\times(\mathbf{b}\times\mathbf{c})$。

从叉积的几何意义出发，我们可以更加深入地理解它的用处。对两个矢量进行叉积的结果会得到一个同时垂直于这两个矢量的新矢量。我们已经知道，矢量是由一个模和方向来定义的，那么这个新的矢量的模和方向是什么呢？

我们先来看它的模。$\mathbf{a}\times\mathbf{b}$ 的长度等于 $\mathbf{a}$ 和 $\mathbf{b}$ 的模的乘积再乘以它们之间夹角的正弦值。公式如下：

$$|\mathbf{a}\times\mathbf{b}| = |\mathbf{a}||\mathbf{b}|\sin\theta$$

读者可能已经发现，上述公式和点积的计算公式很类似，不同的是，这里使用的是正弦值。如果读者对中学数学还有记忆的话，可能还会发现，这和平行四边形的面积计算公式是一样的。如果你忘记了，没关系，我们在这里回忆一下。

如图 4.26 所示，我们使用 $\mathbf{a}$ 和 $\mathbf{b}$ 构建一个平行四边形。

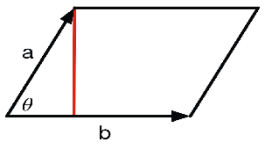
我们知道，平行四边形的面积可以使用 $|\mathbf{b}|h$ 来得到，即底乘以高。而 h 又可以使用 $|\mathbf{a}|$ 和夹角 θ 来得到，即

$$A = |\mathbf{b}|h = |\mathbf{b}|(|\mathbf{a}|\sin\theta) = |\mathbf{a}||\mathbf{b}|\sin\theta = |\mathbf{a}\times\mathbf{b}|$$

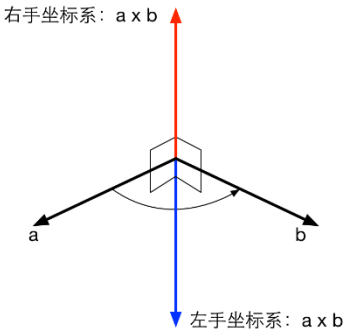
你可能会问，如果 $\mathbf{a}$ 和 $\mathbf{b}$ 平行（可以是方向完全相同，也可以是完全相反）怎么办，不就不能构建平行四边形了吗？我们可以认为构建出来的平行四边形面积为 0，那么 $\mathbf{a}\times\mathbf{b} = \mathbf{0}$。注意，这里得到的是零向量，而不是标量 0。

下面，我们来看结果矢量的方向。你可能会说：“方向？不是已经说了方向了嘛，就是和两个矢量都垂直就可以了啊。”但是，如果你仔细想一下就会发现，实际上我们有两个方向可以选择，这两个方向都和这两个矢量垂直。那么，我们要选择哪个方向呢？

这里就要和之前提到的左手坐标系和右手坐标系联系起来了，如图 4.27 所示。



▲图 4.26 使用矢量 a 和矢量 b 构建一个平行四边形



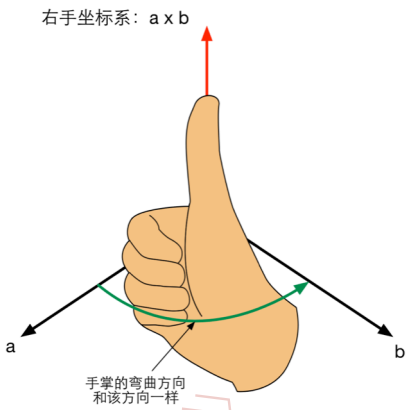
▲图 4.27 分别使用左手坐标系和右手坐标系得到的叉积结果

这个结果是怎么得到的呢？来，举起你的双手！哦，不……先举起你的右手。在右手坐标系

中，**a**×**b** 的方向将使用右手法则来判断。我们先想象把手心放在了 **a** 和 **b** 的尾部交点处，然后张开你的手掌让手掌方向和 **a** 的方向重合，再弯曲你的四指让它们向 **b** 的方向靠拢，最后伸出你的大拇指！大拇指指向的方向就是右手坐标系中 **a**×**b** 的方向了。如果你实在不明白怎么摆放和扭动你的手，那么就看图 4.28 好了。

同理，我们可以使用左手法则来判断左手坐标系中 **a**×**b** 的方向。赶紧举起你的左手试试吧（你可能会发现这个姿势比较扭曲）！

需要注意的是，虽然看起来左右手坐标系的选择会影响叉积的结果，但这仅仅是“看起来”而已。从叉积的数学表达式可以发现，使用左手坐标系还是右手坐标系不会对计算结果产生任何影响，它影响的只是数字在三维空间中的视觉化表现而已。当从右手坐标系转换为左手坐标系时，所有点和矢量的表达和计算方式都会保持不变，只是当呈现到屏幕上时，我们可能会发现，“咦，怎么图像反过来了！”。当我们想要两个坐标系达到同样的视觉效果时，可能就需要改变一些数学运算公式，这不在本书的范畴内。有兴趣的读者可以参考本章的扩展阅读部分。



▲图 4.28　使用右手法则判断右手坐标系中 **a**×**b** 的方向

那么，叉积到底有什么用呢？最常见的一个应用就是计算垂直于一个平面、三角形的矢量。还可以用于判断三角面片的朝向。读者可以在本节的练习题中找到这些应用。

4.3.3　练习题

又到了做练习的时候了，大家是不是都很激动！那么，赶紧拿起笔、拿起纸开始吧！

1．是非题

（1）一个矢量的大小不重要，我们只需要在正确的位置把它画出来就可以了。

（2）点可以认为是位置矢量，这是通过把矢量的尾固定在原点得到的。

（3）选择左手坐标系还是右手坐标系很重要，因为这会影响叉积的计算。

2．计算下面的矢量运算：

（1）|(2, 7, 3)|

（2）2.5(5, 4, 10)

（3）$\frac{(3,4)}{2}$

（4）对(5, 12)进行归一化

（5）(1, 1, 1)进行归一化

（6）(7, 4) + (3, 5)

（7）(9, 4, 13) − (15, 3, 11)

3．假设，场景中有一个光源，位置在(10, 13, 11)处，还有一个点(2, 1, 1)，那么光源距离该点的距离是的少？

4．计算下面的矢量运算：

（1）(4, 7) · (3, 9)

（2）(2, 5, 6) · (3, 1, 2) − 10

（3）$0.5(-3, 4) \cdot (-2, 5)$

（4）$(3, -1, 2) \times (-5, 4, 1)$

（5）$(-5, 4, 1) \times (3, -1, 2)$

5．已知矢量 **a** 和矢量 **b**，**a** 的模为 4，**b** 的模为 6，它们之间的夹角为 60°。计算：

（1）$\mathbf{a} \cdot \mathbf{b}$

（2）$|\mathbf{a} \times \mathbf{b}|$ 提示：$\sin 60° = \frac{\sqrt{3}}{2} \approx 0.866$，$\cos 60° = \frac{1}{2} = 0.5$。

6．假设，场景中有一个 NPC，它位于点 $\boldsymbol{p}$ 处，它的前方（forward）可以使用矢量 $\mathbf{v}$ 来表示。

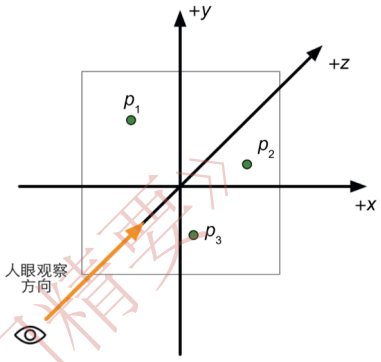
（1）如果现在玩家运动到了点 $\mathbf{x}$ 处，那么如何判断玩家是在 NPC 的前方还是后方？请使用数学公式来描述你的答案。提示：使用点积。

（2）使用你在（1）中提到的方法，代入 $\mathbf{p} = (4, 2)$, $\mathbf{v} = (-3, 4)$, $\mathrm{x} = (10, 6)$来验证你的答案。

（3）现在，游戏有了新的需求：NPC 只能观察到有限的视角范围，这个视角的角度是ϕ，也就是说 NPC 最多只能看到它前方左侧或右侧 $\frac{\phi}{2}$ 角度内的物体。那么，我们如何通过点积来判断 NPC 是否可以看到点 $\mathbf{x}$ 呢？

（4）在（3）的条件基础上，策划又有了新的需求：NPC 的观察距离也有了限制，它只能看到固定距离内的对象，现在又如何判断呢？

7．在渲染中我们时常会需要判断一个三角面片是正面还是背面，这可以通过判断三角形的 3 个顶点在当前空间中是顺时针还是逆时针排列来得到。给定三角形的三个顶点 $\mathbf{p}_1$、$\mathbf{p}_2$ 和 $\mathbf{p}_3$，如何利用叉积来判断这三个点的顺序是顺时针还是逆时针？假设我们使用的是左手坐标系，且 $\mathbf{p}_1$、$\mathbf{p}_2$ 和 $\mathbf{p}_3$ 都位于 xy 平面（即它们的 z 分量均为 0），人眼位于 z 轴的负方向上，向 z 轴正方向观察，如图 4.29 所示。



▲图 4.29　三角形的三个顶点位于 xy 平面上，人眼位于 z 轴负方向，向 z 轴正方向观察

4.4 矩阵

不幸的是，没有人能告诉你母体（matrix）究竟是什么。你需要自己去发现它。

——电影《黑客帝国》（英文名：The Matrix）

矩阵，英文名是 matrix。如果你用翻译软件去查 matrix 这个单词的翻译，就会发现它还有一个意思就是母体。事实上，很多人都不知道，那部具有跨时代意义的电影《黑客帝国》的英文名就是《The Matrix》。在电影《黑客帝国》中，母体是一个庞大的虚拟系统，它看似虚无缥渺，但又连接万物。这一点和矩阵有异曲同工之妙。

没有人敢否认矩阵在三维数学中的重要性，事实上矩阵在整个线性代数的世界中都扮演了举足轻重的角色。在三维数学中，我们通常会使用矩阵来进行变换。一个矩阵可以把一个矢量从一个坐标空间转换到另一个坐标空间。在第 1 章渲染流水线中，我们就看到了很多坐标变换，例如在顶点着色器中我们需要把顶点坐标从模型空间变换到齐次裁剪坐标系中。而在这一章中，我们先来认识一下矩阵这个概念。

那么，现在我们就来看一下，这些放在一个小括号里的数字怎么就这么重要呢？为什么数学家们都喜欢用这个小东西来搞出这么多名堂呢？

4.4.1 矩阵的定义

相信很多读者都见过矩阵的真容，例如像下面这个样子：

$$\begin{bmatrix} 1 & 0.5 & 3 & 2 \\ 2.3 & 5 & \sqrt{3} & 10 \\ 4 & 8 & 11 & 5 \end{bmatrix}$$

从它的外观上来看，就是一个长方形的网格，每个格子里放了一个数字。的确，矩阵就是这么简单：它是由 $m \times n$ 个标量组成的长方形数组。在上面的式子中，我们是用方括号来围住矩阵中的数字，而一些其他的资料可能会使用圆括号或花括号来表示，这都是等价的。

既然是网格结构，就意味着矩阵有**行**（row）**列**（column）之分。例如上面的例子就是一个 3×4 的矩阵，它有三行四列。据此，我们可以给出矩阵的一般表达式。以 3×3 的矩阵为例，它可以写成：

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

m_{ij} 表明了这个元素在矩阵 M 的第 i 行、第 j 列。

这样看起来矩阵也没什么神秘的嘛。但是，越简单的东西往往越厉害，这也是数学的魅力所在。

4.4.2 和矢量联系起来

前面说到，矢量其实就是一个数组，而矩阵也是一个数组。既然都是数组，那就是一家人了！我们很容易想到，我们可以用矩阵来表示矢量。实际上，矢量可以看成是 $n \times 1$ 的**列矩阵（column matrix）**或 $1 \times n$ 的**行矩阵（row matrix）**，其中 n 对应了矢量的维度。例如，矢量 $\mathbf{v} = (3, 8, 6)$可以写成行矩阵

$$[3 \quad 8 \quad 6]$$

或列矩阵

$$\begin{bmatrix} 3 \\ 8 \\ 6 \end{bmatrix}$$

为什么我们要把矢量和矩阵联系在一起呢？这是为了可以让矢量像一个矩阵一样一起参与矩阵运算。这在空间变换中将非常有用。

到现在，使用行矩阵还是列矩阵来表示矢量看起来是没什么分别的。的确，我们可以根据自己的喜好来选择表示方法，但是，如果要和矩阵一起参与乘法运算时，这种选择会影响我们的书写顺序和结果。这正是我们下面要讲到的。

4.4.3 矩阵运算

矩阵这个家伙看起来比矢量要庞大很多，那么它的运算是不是很复杂呢？答案是肯定的。但是，幸运的是在写 Shader 的过程中，我们只需要和很简单的一部分运算打交道。

1．矩阵和标量的乘法

和矢量类似，矩阵也可以和标量相乘，它的结果仍然是一个相同维度的矩阵。它们之间的乘法非常简单，就是矩阵的每个元素和该标量相乘。以 3×3 的矩阵为例，其公式如下：

$$k\mathbf{M} = \mathbf{M}k = k\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} = \begin{bmatrix} km_{11} & km_{12} & km_{13} \\ km_{21} & km_{22} & km_{23} \\ km_{31} & km_{32} & km_{33} \end{bmatrix}$$

2．矩阵和矩阵的乘法

两个矩阵的乘法也很简单，它们的结果会是一个新的矩阵，并且这个矩阵的维度和两个原矩阵的维度都有关系。

一个 $r\times n$ 的矩阵 **A** 和一个 $n\times c$ 的矩阵 **B** 相乘，它们的结果 **AB** 将会是一个 $r\times c$ 大小的矩阵。请读者注意它们的行列关系，第一个矩阵的列数必须和第二个矩阵的行数相同，它们相乘得到的矩阵的行数是第一个矩阵的行数，而列数是第二个矩阵的列数。例如，如果矩阵 **A** 的维度是 4×3，矩阵 **B** 的维度是 3×6，那么 **AB** 的维度就是 4×6。

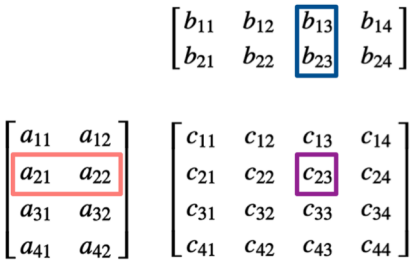
如果两个矩阵的行列不满足上面的规定怎么办？那么很抱歉，这两个矩阵就不能相乘，因为它们之间的乘法是没有被定义的（当然，读者完全可以自己定义一种新的乘法，但是数学家们会不会买账就不一定了）。那么为什么会有上面的规定呢？等我们理解了矩阵乘法的操作过程自然就会明白。

我们先给出看起来很复杂难懂（当给出直观的表式后读者会发现其实它没那么难懂）的数学表达式：设有 $r\times n$ 的矩阵 **A** 和一个 $n\times c$ 的矩阵 **B**，它们相乘会得到一个 $r\times c$ 的矩阵 $\mathbf{C}=\mathbf{AB}$。那么，**C** 中的每一个元素 c_{ij} 等于 **A** 的第 i 行所对应的矢量和 **B** 的第 j 列所对应的矢量进行矢量点乘的结果，即

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

看起来很复杂对吗？但是，我们可以用一个更简单的方式来解释：对于每个元素 c_{ij}，我们找到 **A** 中的第 i 行和 **B** 中的第 j 列，然后把它们的对应元素相乘后再加起来，这个和就是 c_{ij}。

一种更直观的方式如图 4.30 所示。假设 **A** 的大小是 4×2，**B** 的大小是 2×4，那么如果要计算 **C** 的元素 c_{23} 的话，先找到对应的行矩阵和列矩阵，即 **A** 中的第 2 行和 **B** 中的第 3 列，把它们进行矢量点积后就可以得到结果值。因此，$c_{23} = a_{21}b_{13} + a_{22}b_{23}$。



▲图 4.30 计算 c_{23} 的过程

在 Shader 的计算中，我们更多的是使用 4×4 矩阵来运算的。

矩阵乘法满足一些性质。

性质一：矩阵乘法并不满足交换律。

也就是说，通常情况下：

$$\mathbf{AB} \neq \mathbf{BA}$$

性质二：矩阵乘法满足结合律。

也就是说，

$$(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$$

矩阵乘法的结合律可以扩展到更多矩阵的相乘。例如，

$$\mathbf{ABCDE} = ((\mathbf{A}(\mathbf{BC}))\mathbf{D})\mathbf{E} = (\mathbf{AB})(\mathbf{CD})\mathbf{E}$$

读者可根据矩阵乘法的定义很轻松地验证上述结论。

4.4.4 特殊的矩阵

有一些特殊的矩阵类型在 Shader 中经常见到。这些特殊的矩阵往往具有一些重要的性质。

1. 方块矩阵

方块矩阵（square matrix），简称方阵，是指那些行和列数目相等的矩阵。在三维渲染里，最常使用的就是 3×3 和 4×4 的方阵。

方阵之所以值得单独拿出来讲，是因为矩阵的一些运算和性质是只有方阵才具有的。例如，**对角元素（diagonal elements）**。方阵的对角元素指的是行号和列号相等的元素，例如 m_{11}、m_{22}、m_{33} 等。如果把方阵看成一个正方形的话，这些元素排列在正方形的对角线上，这也是它们名字的由来。如果一个矩阵除了对角元素外的所有元素都为 0，那么这个矩阵就叫做**对角矩阵（diagonal matrix）**。例如，下面就是一个 4×4 的对角矩阵：

$$\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 7 \end{bmatrix}$$

2. 单位矩阵

一个特殊的对角矩阵是**单位矩阵（identity matrix）**，用 $\mathbf{I}_n$ 来表示。一个 3×3 的单位矩阵如下：

$$\mathbf{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

为什么要为这种矩阵单独起一个名字呢？这是因为，任何矩阵和它相乘的结果都还是原来的矩阵。也就是说，

$$\mathbf{MI} = \mathbf{IM} = \mathbf{M}$$

这就跟标量中的数字 1 一样！

3. 转置矩阵

转置矩阵（transposed matrix）实际是对原矩阵的一种运算，即转置运算。给定一个 $r \times c$ 的矩阵 $\mathbf{M}$，它的转置可以表示成 $\mathbf{M}^T$，这是一个 $c \times r$ 的矩阵。转置矩阵的计算非常简单，我们只需要把原矩阵翻转一下即可。也就是说，原矩阵的第 i 行变成了第 i 列，而第 j 列变成了第 j 行。数学公式是：

$$\mathbf{M}_{ij}^T = \mathbf{M}_{ji}$$

例如，

$$\begin{bmatrix} 6 & 2 & 10 & 3 \\ 7 & 5 & 4 & 9 \end{bmatrix}^T = \begin{bmatrix} 6 & 7 \\ 2 & 5 \\ 10 & 4 \\ 3 & 9 \end{bmatrix}$$

对于行矩阵和列矩阵来说，我们可以使用转置操作来转换行列矩阵：

$$\begin{bmatrix} x & y & z \end{bmatrix}^T = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}^T = \begin{bmatrix} x & y & z \end{bmatrix}$$

转置矩阵也有一些常用的性质。

性质一：矩阵转置的转置等于原矩阵。

很容易理解，我们把一个矩阵翻转一下后再翻转一下，等于没有对矩阵做任何操作。即

$$(\mathbf{M}^T)^T = \mathbf{M}$$

性质二：矩阵串接的转置，等于反向串接各个矩阵的转置。

用公式表示就是：

$$(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$$

该性质同样可以扩展到更多矩阵相乘的情况。

4．逆矩阵

逆矩阵（inverse matrix）大概是本书讲到的关于矩阵最复杂的一种操作了。不是所有的矩阵都有逆矩阵，第一个前提就是，该矩阵必须是一个方阵。

给定一个方阵 **M**，它的逆矩阵用 $\mathbf{M}^{-1}$ 来表示。逆矩阵最重要的性质就是，如果我们把 **M** 和 $\mathbf{M}^{-1}$ 相乘，那么它们的结果将会是一个单位矩阵。也就是说，

$$\mathbf{M}\mathbf{M}^{-1} = \mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$$

前面说了，并非所有的方阵都有对应的逆矩阵。一个明显的例子就是一个所有元素都为 0 的矩阵，很显然，任何矩阵和它相乘都会得到一个零矩阵，即所有的元素仍然都是 0。如果一个矩阵有对应的逆矩阵，我们就说这个矩阵是**可逆的（invertible）**或者说是**非奇异的（nonsingular）**；相反的，如果一个矩阵没有对应的逆矩阵，我们就说它是**不可逆的（noninvertible）**或者说是**奇异的（singular）**。

那么如何判断一个矩阵是否是可逆的呢？简单来说，如果一个矩阵的**行列式（determinant）**不为 0，那么它就是可逆的。关于矩阵的行列式是什么以及如何求解一个矩阵的逆矩阵，可以参见本章的扩展阅读部分。由于这部分内容涉及较多计算和其他定义，本书不再赘述。在写 Shader 的过程中，这些矩阵通常可以通过调用第三方库（如 C++数学库 Eigen）来直接求得，不需要开发者手动计算。在 Unity 中，重要变换矩阵的逆矩阵 Unity 也提供了相应的变量供我们使用。关于这些 Unity 内置的矩阵，读者可以在本章的 4.8 节找到更详细的解释。

逆矩阵有很多非常重要的性质。

性质一：逆矩阵的逆矩阵是原矩阵本身。

假设矩阵 **M** 是可逆的，那么

$$(\mathbf{M}^{-1})^{-1} = \mathbf{M}$$

性质二：单位矩阵的逆矩阵是它本身。

即

$$\mathbf{I}^{-1} = \mathbf{I}$$

性质三：转置矩阵的逆矩阵是逆矩阵的转置。

即

$$(\mathbf{M}^T)^{-1} = (\mathbf{M}^{-1})^T$$

性质四：矩阵串接相乘后的逆矩阵等于反向串接各个矩阵的逆矩阵。

即

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$$

这个性质也可以扩展到更多矩阵的连乘，如：

$$(\mathbf{ABCD})^{-1} = \mathbf{D}^{-1}\mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1}$$

逆矩阵是具有几何意义的。我们知道一个矩阵可以表示一个变换（详见 4.5 节），而逆矩阵允许我们还原这个变换，或者说是计算这个变换的反向变换。因此，如果我们使用变换矩阵 **M** 对矢量 **v** 进行了一次变换，然后再使用它的逆矩阵 $\mathbf{M}^{-1}$ 进行另一次变换，那么我们会得到原来的矢量。这个性质可以使用矩阵乘法的结合律很容易地进行证明：

$$\mathbf{M}^{-1}(\mathbf{Mv}) = (\mathbf{M}^{-1}\mathbf{M})\mathbf{v} = \mathbf{Iv} = \mathbf{v}$$

5．正交矩阵

另一个特殊的方阵是**正交矩阵（orthogonal matrix）**。正交是矩阵的一种属性。如果一个方阵 **M** 和它的转置矩阵的乘积是单位矩阵的话，我们就说这个矩阵是**正交的（orthogonal）**。反过来也是成立的。也就是说，矩阵 **M** 是正交的等价于：

$$\mathbf{MM}^T = \mathbf{M}^T\mathbf{M} = \mathbf{I}$$

读者可能已经看出来，上式和我们在上一节讲到的逆矩阵时遇到的公式很像。把这两个公式结合起来，我们就可以得到一个重要的性质，即如果一个矩阵是正交的，那么它的转置矩阵和逆矩阵是一样的。也就是说，矩阵 **M** 是正交的等价于：

$$\mathbf{M}^T = \mathbf{M}^{-1}$$

这个式子非常有用，因为在三维变换中我们经常会需要使用逆矩阵来求解反向的变换。而逆矩阵的求解往往计算量很大，但转置矩阵却非常容易求解：我们只需要把矩阵翻转一下就可以了。那么，我们如何提前判断一个矩阵是否是正交矩阵呢？读者可能会说，判断 $\mathbf{MM}^T = \mathbf{I}$ 是否成立就可以了嘛！但是，求解这样一个表达式无疑是需要一定计算量的，这些计算量可能和直接求解逆矩阵无异。而且，如果我们判断出来这不是一个正交矩阵，那么这些花在验证是否是正交矩阵上的计算就浪费了。因此，我们更想不需要计算，而仅仅根据一个矩阵的构造过程来判断这个矩阵是否是正交矩阵。为此，我们需要来了解正交矩阵的几何意义。

我们来看一下对于 3×3 的正交矩阵有什么特点。根据正交矩阵的定义，我们有：

$$\begin{aligned}\mathbf{M}^T\mathbf{M} &= \begin{bmatrix} - & \mathbf{c}_1 & - \\ - & \mathbf{c}_2 & - \\ - & \mathbf{c}_3 & - \end{bmatrix}\begin{bmatrix} | & | & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \mathbf{c}_3 \\ | & | & | \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{c}_1\cdot\mathbf{c}_1 & \mathbf{c}_1\cdot\mathbf{c}_2 & \mathbf{c}_1\cdot\mathbf{c}_3 \\ \mathbf{c}_2\cdot\mathbf{c}_1 & \mathbf{c}_2\cdot\mathbf{c}_2 & \mathbf{c}_2\cdot\mathbf{c}_3 \\ \mathbf{c}_3\cdot\mathbf{c}_1 & \mathbf{c}_3\cdot\mathbf{c}_2 & \mathbf{c}_3\cdot\mathbf{c}_3 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \mathbf{I}\end{aligned}$$

这样，我们就有了 9 个等式：

$$\mathbf{c}_1\cdot\mathbf{c}_1=1，\ \mathbf{c}_1\cdot\mathbf{c}_2=0，\ \mathbf{c}_1\cdot\mathbf{c}_3=0$$
$$\mathbf{c}_2\cdot\mathbf{c}_1=0，\ \mathbf{c}_2\cdot\mathbf{c}_2=1，\ \mathbf{c}_2\cdot\mathbf{c}_3=0$$
$$\mathbf{c}_3\cdot\mathbf{c}_1=0，\ \mathbf{c}_3\cdot\mathbf{c}_2=0，\ \mathbf{c}_3\cdot\mathbf{c}_3=1$$

我们可以得到以下结论：

- 矩阵的每一行，即 $\mathbf{c}_1$、$\mathbf{c}_2$ 和 $\mathbf{c}_3$ 是单位矢量，因为只有这样它们与自己的点积才能是 1。
- 矩阵的每一行，即 $\mathbf{c}_1$、$\mathbf{c}_2$ 和 $\mathbf{c}_3$ 之间互相垂直，因为只有这样它们之间的点积才能是 0。
- 上述两条结论对矩阵的每一列同样适用，因为如果 **M** 是正交矩阵的话，$\mathbf{M}^T$ 也会是正交矩阵。

也就是说，如果一个矩阵满足上面的条件，那么它就是一个正交矩阵。读者可以注意到，一组标准正交基（定义详见 4.2.2 节）可以精确地满足上述条件。在 4.6.2 节中，我们会使用坐标空间的基矢量来构建用于空间变换的矩阵。因此，如果这些基矢量是一组标准正交基的话（例如只存在旋转变换），那么我们就可以直接使用转置矩阵来求得该变换的逆变换。

读者：我被标准正交、正交这些概念搞混了，可以再说明一下是什么意思吗？

我们：读者应该已经知道，一个坐标空间需要指定一组基矢量，也就是我们理解的坐标轴。如果这些基矢量之间是互相垂直的，那么我们就把它们称为是一组**正交基（orthogonal basis）**。但是，它们的长度并不要求一定是 1。如果它们的长度的确是 1 的话，我们就说它们是一组**标准正交基（orthonormal basis）**。因此，一个正交矩阵的行和列之间分别构成了一组标准正交基。但是，如果我们使用一组正交基来构建一个矩阵的话，这个矩阵可能就不是一个正交矩阵，因为这些基矢量的长度可能不为 1，也就是说它们不是标准正交基。

4.4.5 行矩阵还是列矩阵

我们已经了解了足够多的数学概念，但在学习矩阵的几何意义之前，我们有必要说明一下行矩阵和列矩阵的问题。

在前面的章节中我们讲到，可以把一个矢量转换成一个行矩阵或是列矩阵。它们本身是没有区别的，但是，当我们需要把它和另一个矩阵相乘时，就会出现一些差异。

假设有一个矢量 $\mathbf{v}=(x,y,z)$，我们可以把它转换成行矩阵 $\mathbf{v}=[x\quad y\quad z]$或列矩阵 $\mathbf{v}=[x\ \ y\ \ z]^T$（这里使用了转置符号来避免列矩阵在我们的这一行中显得太高）。现在，有另一个矩阵 **M**：

$$\mathbf{M}=\begin{bmatrix}m_{11} & m_{12} & m_{13}\\ m_{21} & m_{22} & m_{23}\\ m_{31} & m_{32} & m_{33}\end{bmatrix}$$

那么 **M** 分别和行矩阵以及列矩阵相乘后会是什么结果呢？我们先来看 **M** 和行矩阵的相乘。由矩阵乘法的定义可知，我们需要把行矩阵放在 **M** 的左边（还记得吗，矩阵乘法要求两个矩阵的行列数满足一定条件），即

$$\mathbf{vM}=[xm_{11}+ym_{21}+zm_{31}\quad xm_{12}+ym_{22}+zm_{32}\quad xm_{13}+ym_{23}+zm_{33}]$$

而如果和列矩阵相乘的话，结果是：

$$\mathbf{Mv}=\begin{bmatrix}xm_{11}+ym_{12}+zm_{13}\\ xm_{21}+ym_{22}+zm_{23}\\ xm_{31}+ym_{32}+zm_{33}\end{bmatrix}$$

读者认真对比就会发现，结果矩阵除了行列矩阵的区别外，里面的元素也是不一样的。这就意味着，在和矩阵相乘时选择行矩阵还是列矩阵来表示矢量是非常重要的，因为这决定了矩阵乘法的书写次序和结果值。

在 Unity 中，常规做法是把矢量放在矩阵的右侧，即把矢量转换成列矩阵来进行运算。因此，在本书后面的内容中，如无特殊情况，我们都将使用列矩阵。这意味着，我们的矩阵乘法通常都是右乘，例如：

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

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

上面的计算等价于下面的行矩阵运算：

$$\mathbf{vA}^T\mathbf{B}^T\mathbf{C}^T = (((\mathbf{vA}^T)\mathbf{B}^T)\mathbf{C}^T)$$

如果你还是不能明白上面的含义，可以参见练习题 3。

4.4.6 练习题

1．判断下面矩阵的乘法是否存在。如果存在，计算它们的乘积。

（1）$\begin{bmatrix}1 & 3\\ 2 & 4\end{bmatrix}\begin{bmatrix}-1 & 5\\ 0 & 2\end{bmatrix}$

（2）$\begin{bmatrix}2 & 4 & 3\\ 2 & 1 & 4\end{bmatrix}\begin{bmatrix}-1 & 5\\ 0 & 2\\ 3 & 10\\ 4 & 5\end{bmatrix}$

（3）$\begin{bmatrix}1 & -2 & 3\\ 5 & 1 & 4\\ 6 & 0 & 3\end{bmatrix}\begin{bmatrix}-5\\ 4\\ 8\end{bmatrix}$

2．判断下面的矩阵是否是正交矩阵。

（1）$\begin{bmatrix}1 & 0 & 0\\ 1 & 0 & 0\\ 1 & 0 & 0\end{bmatrix}$

（2）$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$

（3）$\begin{bmatrix}\cos\theta & -\sin\theta & 0\\ \sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{bmatrix}$

3．给定一个矢量(3, 2, 6)，分别把它当成行矩阵和列矩阵与下面的矩阵进行乘法。考虑两种情况下得到的矢量结果是否一样。如果不一样，考虑如何得到相同的结果。

（1）$\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$

（2）$\begin{bmatrix}1 & 0 & 2\\ 0 & 1 & -3\\ 0 & 0 & 3\end{bmatrix}$

（3）$\begin{bmatrix}2 & -1 & 3\\ -1 & 5 & -3\\ 3 & -3 & 4\end{bmatrix}$

4.5 矩阵的几何意义：变换

关于矩阵，很多困扰初学者的问题都是类似的：

- 点和矢量都可以在图像中画出来，那么矩阵可以吗？
- 我听说矩阵和线性变换、仿射变换有关，这些变换到底是什么意思呢？
- 我总是听到齐次坐标这个名词，它是什么意思呢？
- 变换和矩阵的关系又是什么呢？或者说，给定一个变换，我如何得到它对应的矩阵呢？

在学习完本节后，希望读者们能够回答出这些问题。

对于第一个问题，在三维渲染中矩阵可以可视化吗？幸运的是，答案是肯定的，这个可视化的结果就是变换。因此，如果读者在后面的内容中看到了一个矩阵，那么你可以认为自己看到的就是一个变换（当然，在线性代数中矩阵的用处不仅是用于变换，但本书的讨论范围仅在于此）。

在游戏的世界中，这些变换一般包含了旋转、缩放和平移。游戏开发人员希望给定一个点或矢量，再给定一个变换（例如把点平移到另一个位置，把矢量的方向旋转 30°等），就可以通过某个数学运算来求得新的点和矢量。聪明的先人们发现，可以使用矩阵来完美地解决这个问题。那么问题就变成了，我们如何使用矩阵来表示这些变换？

4.5.1 什么是变换

变换（transform），指的是我们把一些数据，如点、方向矢量甚至是颜色等，通过某种方式进行转换的过程。在计算机图形学领域，变换非常重要。尽管通过变换我们能够进行的操作是有限的，但这些操作已经足够奠定变换在图形学领域举足轻重的地位了。

我们先来看一个非常常见的变换类型——**线性变换（linear transform）**。线性变换指的是那些可以保留矢量加和标量乘的变换。用数学公式来表示这两个条件就是：

$$\mathbf{f}(\mathbf{x}) + \mathbf{f}(\mathbf{y}) = \mathrm{f}(\mathbf{x} + \mathbf{y})$$

$$k\mathbf{f}(\mathbf{x}) = \mathbf{f}(k\mathbf{x})$$

上面的式子看起来很抽象。**缩放（scale）**就是一种线性变换。例如，$\mathbf{f}(\mathbf{x}) = 2\mathbf{x}$，可以表示一个大小为 2 的统一缩放，即经过变换后矢量 $\mathbf{x}$ 的模将被放大两倍。可以发现，$\mathbf{f}(\mathbf{x}) = 2\mathbf{x}$ 是满足上面的两个条件的。同样，**旋转（rotation）**也是一种线性变换。对于线性变换来说，如果我们要对一个三维的矢量进行变换，那么仅仅使用 3×3 的矩阵就可以表示所有的线性变换。

线性变换除了包括旋转和缩放外，还包括**错切（shear）**、**镜像（mirroring，也被称为 reflection）**、**正交投影（orthographic projection）**等，但本书着重讲述旋转和缩放变换。

但是，仅有线性变换是不够的。我们来考虑平移变换，例如 **f**(x) = x + (1, 2, 3)。这个变换就不是一个线性变换，它既不满足标量乘法，也不满足矢量加法。如果我们令 **x** = (1, 1, 1)，那么：

$$\mathbf{f}(x) + \mathbf{f}(x) = (4, 6, 8)$$

$$\mathbf{f}(x + x) = (3, 4, 5)$$

可见，两个运算得到的结果是不一样的。因此，我们不能用一个 3×3 的矩阵来表示一个平移变换。这是我们不希望看到的，毕竟平移变换是非常常见的一种变换。

这样，就有了**仿射变换（affine transform）**。仿射变换就是合并线性变换和平移变换的变换类型。仿射变换可以使用一个 4×4 的矩阵来表示，为此，我们需要把矢量扩展到四维空间下，这就是**齐次坐标空间（homogeneous space）**。

表 4.1 给出了图形学中常见变换矩阵的名称和它们的特性。

表 4.1　常见的变换种类和它们的特性（N 表示不满足该特性，Y 表示满足该特性）

变 换 名 称	是线性变换吗	是仿射变换吗	是可逆矩阵吗	是正交矩阵吗
平移矩阵	N	Y	Y	N
绕坐标轴旋转的旋转矩阵	Y	Y	Y	Y
绕任意轴旋转的旋转矩阵	Y	Y	Y	Y
按坐标轴缩放的缩放矩阵	Y	Y	Y	N
错切矩阵	Y	Y	Y	N
镜像矩阵	Y	Y	Y	Y
正交投影矩阵	Y	Y	N	N
透视投影矩阵	N	N	N	N

在下面的内容中，我们将学习其中一些基本的变换类型：旋转，缩放和平移。对于正交投影和透视投影，我们将在 4.6.7 节中给出它们的表示方法。而对于其他变换类型，本书不再具体讨论，读者可以在本章的扩展阅读中找到更多内容。

4.5.2 齐次坐标

我们知道，由于 3×3 矩阵不能表示平移操作，我们就把其扩展到了 4×4 的矩阵（是的，只要多一个维度就可以实现对平移的表示）。为此，我们还需要把原来的三维矢量转换成四维矢量，也就是我们所说的**齐次坐标（homogeneous coordinate）**（事实上齐次坐标的维度可以超过四维，但本书中所说的齐次坐标将泛指四维齐次坐标）。我们可以发现，齐次坐标并没有神秘的地方，它只是为了方便计算而使用的一种表示方式而已。

如上所说，齐次坐标是一个四维矢量。那么，我们如何把三维矢量转换成齐次坐标呢？对于一个点，从三维坐标转换成齐次坐标是把其 w 分量设为 1，而对于方向矢量来说，需要把其 w 分量设为 0。这样的设置会导致，当用一个 4×4 矩阵对一个点进行变换时，平移、旋转、缩放都会施加于该点。但是如果是用于变换一个方向矢量，平移的效果就会被忽略。我们可以从下面的内容中理解这些差异的原因。

4.5.3 分解基础变换矩阵

我们已经知道，可以使用一个 4×4 的矩阵来表示平移、旋转和缩放。我们把表示纯平移、纯

旋转和纯缩放的变换矩阵叫做基础变换矩阵。这些矩阵具有一些共同点，我们可以把一个基础变换矩阵分解成 4 个组成部分：

$$\begin{bmatrix} \mathbf{M}_{3\times3} & \mathbf{t}_{3\times1} \\ \mathbf{0}_{1\times3} & 1 \end{bmatrix}$$

其中，左上角的矩阵 $\mathbf{M}_{3\times3}$ 用于表示旋转和缩放，$\mathbf{t}_{3\times1}$ 用于表示平移，$\mathbf{0}_{1\times3}$ 是零矩阵，即 $\mathbf{0}_{1\times3}$ = [0　0　0]，右下角的元素就是标量 1。

接下来，我们来具体学习如何用这样一个 4×4 的矩阵来表示平移、旋转和缩放。

4.5.4 平移矩阵

我们可以使用矩阵乘法来表示对一个点进行平移变换：

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x+t_x \\ y+t_y \\ z+t_z \\ 1 \end{bmatrix}$$

从结果来看我们可以很容易看出为什么这个矩阵有平移的效果：点的 x、y、z 分量分别增加了一个位置偏移。在 3D 中的可视化效果是，把点(x, y, z)在空间中平移了(t_x, t_y, t_z)个单位。

有趣的是，如果我们对一个方向矢量进行平移变换，结果如下：

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

可以发现，平移变换不会对方向矢量产生任何影响。这点很容易理解，我们在学习矢量的时候就说过了，矢量没有位置属性，也就是说它可以位于空间中的任意一点，因此对位置的改变（即平移）不应该对方向矢量产生影响。

现在，读者应该明白当给定一个平移操作时如何构建一个平移矩阵：基础变换矩阵中的 $\mathbf{t}_{3\times1}$ 矢量对应了平移矢量，左上角的矩阵 $\mathbf{M}_{3\times3}$ 为单位矩阵 $\mathbf{I}_3$。

平移矩阵的逆矩阵就是反向平移得到的矩阵，即

$$\begin{bmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

可以看出，平移矩阵并不是一个正交矩阵。

4.5.5 缩放矩阵

我们可以对一个模型沿空间的 x 轴、y 轴和 z 轴进行缩放。同样，我们可以使用矩阵乘法来表示一个缩放变换：

$$\begin{bmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} k_x x \\ k_y y \\ k_z z \\ 1 \end{bmatrix}$$

对方向矢量可以使用同样的矩阵进行缩放：

$$\begin{bmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} k_x x \\ k_y y \\ k_z z \\ 0 \end{bmatrix}$$

如果缩放系数 $k_x = k_y = k_z$，我们把这样的缩放称为**统一缩放（uniform scale）**，否则称为**非统一缩放（nonuniform scale）**。从外观上看，统一缩放是扩大整个模型，而非统一缩放会拉伸或挤压模型。更重要的是，统一缩放不会改变角度和比例信息，而非统一缩放会改变与模型相关的角度和比例。例如在对法线进行变换时，如果存在非统一缩放，直接使用用于变换顶点的变换矩阵的话，就会得到错误的结果。正确的变换方法可参见 4.7 节。

缩放矩阵的逆矩阵是使用原缩放系数的倒数来对点或方向矢量进行缩放，即

$$\begin{bmatrix} \frac{1}{k_x} & 0 & 0 & 0 \\ 0 & \frac{1}{k_y} & 0 & 0 \\ 0 & 0 & \frac{1}{k_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

缩放矩阵一般不是正交矩阵。

上面的矩阵只适用于沿坐标轴方向进行缩放。如果我们希望在任意方向上进行缩放，就需要使用一个复合变换。其中一种方法的主要思想就是，先将缩放轴变换成标准坐标轴，然后进行沿坐标轴的缩放，再使用逆变换得到原来的缩放轴朝向。

4.5.6 旋转矩阵

旋转是三种常见的变换矩阵中最复杂的一种。我们知道，旋转操作需要指定一个旋转轴，这个旋转轴不一定是空间中的坐标轴，但本节所讲的旋转就是指绕着空间中的 x 轴、y 轴或 z 轴进行旋转。

如果我们需要把点绕着 x 轴旋转 θ 度，可以使用下面的矩阵：

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

绕 y 轴的旋转也是类似的：

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

最后，是绕 z 轴的旋转：

$$R_z(\theta)=\begin{bmatrix}\cos\theta & -\sin\theta & 0 & 0\\ \sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

旋转矩阵的逆矩阵是旋转相反角度得到的变换矩阵。旋转矩阵是正交矩阵，而且多个旋转矩阵之间的串联同样是正交的。

4.5.7 复合变换

我们可以把平移、旋转和缩放组合起来，来形成一个复杂的变换过程。例如，可以对一个模型先进行大小为(2, 2, 2)的缩放，再绕 y 轴旋转 30°，最后向 z 轴平移 4 个单位。复合变换可以通过矩阵的串联来实现。上面的变换过程可以使用下面的公式来计算：

$$\mathbf{p}_{new}=\mathbf{M}_{translation}\mathbf{M}_{rotation}\mathbf{M}_{scale}\mathbf{p}_{old}$$

由于上面我们使用的是列矩阵，因此阅读顺序是从右到左，即先进行缩放变换，再进行旋转变换，最后进行平移变换。需要注意的是，变换的结果是依赖于变换顺序的，由于矩阵乘法不满足交换律，因此矩阵的乘法顺序很重要。也就是说，不同的变换顺序得到的结果可能是不一样的。想象一下，如果让读者向前一步然后左转，记住此时的位置。然后回到原位，这次先左转再向前走一步，得到的位置和上一次是不一样的。究其本质，是因为矩阵的乘法不满足交换律，因此不同的乘法顺序得到的结果是不一样的。

在绝大多数情况下，我们约定变换的顺序就是先缩放，再旋转，最后平移。

读者：为什么要约定这样的顺序，而不是其他顺序呢？

我们：因为这样的变换顺序是我们需要的。想象我们对奶牛妞妞进行一个复合变换。如果我们按先平移、再缩放的顺序进行变换，假设初始情况下妞妞位于原点，我们先按(0, 0, 5)平移它，现在它距离原点 5 个单位。然后再将它放大 2 倍，这样所有的坐标都变成了原来的 2 倍，而这意味着妞妞现在的位置是(0, 0, 10)，这不是我们希望的。正确的做法是，先缩放再平移。也就是说，我们先在原点对妞妞进行 2 倍的缩放，再进行平移，这样妞妞的大小正确了，位置也正确了。

为了从数学公式上理解变换顺序的本质，我们可以对比不同变换顺序产生的变换矩阵的表达式。如果我们只考虑对 y 轴的旋转的话，按先缩放、再旋转、最后平移这样的顺序组合三种变换得到的变换矩阵是：

$$\mathbf{M}_{translation}\mathbf{M}_{rotation}\mathbf{M}_{scale}=\begin{bmatrix}1 & 0 & 0 & t_x\\ 0 & 1 & 0 & t_y\\ 0 & 0 & 1 & t_z\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}\cos\theta & 0 & \sin\theta & 0\\ 0 & 1 & 0 & 0\\ -\sin\theta & 0 & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}k_x & 0 & 0 & 0\\ 0 & k_y & 0 & 0\\ 0 & 0 & k_z & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$=\begin{bmatrix}k_x\cos\theta & 0 & k_z\sin\theta & t_x\\ 0 & k_y & 0 & t_y\\ -k_x\sin\theta & 0 & k_z\cos\theta & t_z\\ 0 & 0 & 0 & 1\end{bmatrix}$$

而如果我们使用了其他变换顺序，例如先平移，再缩放，最后旋转，那么得到的变换矩阵是：

$$\mathbf{M}_{rotation}\mathbf{M}_{scale}\mathbf{M}_{translation}=\begin{bmatrix}\cos\theta & 0 & \sin\theta & 0\\ 0 & 1 & 0 & 0\\ -\sin\theta & 0 & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}k_x & 0 & 0 & 0\\ 0 & k_y & 0 & 0\\ 0 & 0 & k_z & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & t_x\\ 0 & 1 & 0 & t_y\\ 0 & 0 & 1 & t_z\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$=\begin{bmatrix}k_x\cos\theta & 0 & k_z\sin\theta & t_xk_x\cos\theta+t_zk_z\sin\theta\\ 0 & k_y & 0 & t_yk_y\\ -k_x\sin\theta & 0 & k_z\cos\theta & -t_xk_x\sin\theta+t_zk_z\cos\theta\\ 0 & 0 & 0 & 1\end{bmatrix}$$

从两个结果可以看出，得到的变换矩阵是不一样的。

除了需要注意不同类型的变换顺序外，我们有时还需要小心旋转的变换顺序。在 4.5.6 节中，我们给出了分别绕 x 轴、y 轴和 z 轴旋转的变换矩阵。一个问题是，如果我们需要同时绕着三个轴进行旋转，是先绕 x 轴、再绕 y 轴最后绕 z 轴旋转还是按其他的旋转顺序呢？

当我们直接给出$(\theta_x, \theta_y, \theta_z)$这样的旋转角度时，需要定义一个旋转顺序。在 Unity 中，这个旋转顺序是 *zxy*，这在旋转相关的 API 文档中都有说明。这意味着，当给定$(\theta_x, \theta_y, \theta_z)$这样的旋转角度时，得到的组合旋转变换矩阵是：

$$\mathbf{M}_{rotate_Z}\mathbf{M}_{rotate_X}\mathbf{M}_{rotate_Y}=\begin{bmatrix}\cos\theta_z & -\sin\theta_z & 0 & 0\\ \sin\theta_z & \cos\theta_z & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\theta_x & -\sin\theta_x & 0\\ 0 & \sin\theta_x & \cos\theta_x & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}\cos\theta_y & 0 & \sin\theta_y & 0\\ 0 & 1 & 0 & 0\\ -\sin\theta_y & 0 & \cos\theta_y & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

一些读者会有疑问：上面的公式书写顺序是不是反了？不是说列矩阵要从右往左读吗？这样一来顺序不就颠倒了吗？实际上，有一个非常重要的东西我们没有说明白，那就是旋转时使用的坐标系。给定一个旋转顺序（例如这里的 *zxy*），以及它们对应的旋转角度$(\theta_x, \theta_y, \theta_z)$，有两种坐标系可以选择。

- 绕坐标系 E 下的 z 轴旋转θ_z，绕坐标系 E 下的 y 轴旋转θ_y，绕坐标系 E 下的 x 轴旋转θ_x，即进行一次旋转时不一起旋转当前坐标系。
- 绕坐标系 E 下的 z 轴旋转θ_z，在坐标系 E 下绕 z 轴旋转θ_z 后的新坐标系 E'下的 y 轴旋转θ_y，在坐标系 E'下绕 y 轴旋转θ_y 后的新坐标系 E"下的 x 轴旋转θ_x，即在旋转时，把坐标系一起转动。

很容易知道，这两种选择的结果是不一样的。但如果把它们的旋转顺序颠倒一下，它们得到的结果就会是一样的！说得明白点，在第一种情况下按 *zxy* 顺序旋转和在第二种情况下按 *yxz* 顺序旋转是一样的。而 Unity 文档中说明的旋转顺序指的是在第一种情况下的顺序。

和上面不同类型的变换顺序导致的问题类似，不同的旋转顺序得到的结果也可能是不一样的。我们同样可以通过对比不同旋转顺序得到的变换矩阵来理解为什么会出现这样的不同。而这个验证过程留给读者作为练习。

4.6 坐标空间

我们已经学会了如何使用矩阵来表示基本的变换，如平移、旋转和缩放。而在本节中，我们将关注如何使用这些变换来对坐标空间进行变换。

我们在第 2 章渲染流水线中就接触了坐标空间的变换。例如，在学习顶点着色器流水线阶段时，我们说过，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。

渲染游戏的过程可以理解成是把一个个顶点经过层层处理最终转化到屏幕上的过程，那么本节我们就将学习这个转换的过程是如何实现的。更具体来说，顶点是经过了哪些坐标空间后，最后被画在了我们的屏幕上。

4.6.1 为什么要使用这么多不同的坐标空间

我们先要回答读者的一个疑问。在编写 Shader 的过程中，很多看起来很难理解和复杂的数学运算都是为了在不同坐标空间之间转换点和矢量。看起来，这么多的坐标空间就是“万恶之源”啊！很多人都有这样的疑问：“为什么我们不能只使用一个坐标空间来做所有的事情呢？这样一来我们不就不用学习这些烦人的数学公式了吗？这样世界将变得多美好啊！”

事情看起来虽然是这样——在只有一个坐标空间的世界里，Shader 的开发者会生活得更加美好。但事实是，一旦你真的这么做了，就会发现理想和现实之间的差距：我们不可以也不愿意抛弃这些不同的坐标空间。

事实上，在我们的生活中，我们也总是使用不同的坐标空间来交流。现在正在读这本书的你，很可能正坐在办公室或书房中。如果问你：“办公室的饮水机在哪里？”你大概会回答：“在办公室门的左方 3 米处。”这里，你很自然地使用了以门为原点的坐标空间。现在，公司的前台小姐走进门来，你非常惊讶地看到她脸上还残留有中午吃饭的米粒！我们假设正在读这本书的你是一个好心而且不喜欢看别人笑话的人，这时你可能会提醒她：“嘿，你左脸上面有些东西没有擦掉！”此时，你又使用了以前台小姐的嘴巴为原点的坐标空间。如果只有一个坐标系会怎么样呢？你可以尝试一下使用以你的办公室的门为原点的坐标空间来描述前台小姐脸上的一粒饭粒。

再比如，我们每个人所生活的城市可以看成是一个世界坐标系（三维渲染里的世界坐标系将在 4.6.5 节中讲到），这个坐标系的坐标轴可以认为是由东南西北这些定义的方向轴。如果一个陌生人向你问路，你很有可能会说：“向东走 800 米上桥，然后再向南走 50 米就到了”。但是我们知道，现实生活中有很多人是分不清东南西北的（在作者小时候，经常使用“上北下南左西右东”来傻傻地判断东南西北，因此总是得到错误方位）。如果现在有一个饥肠辘辘又分不清东南西北的路人来问你最近的餐厅怎么走，你可能会说：“你先往前走 50 米，到了路口向左拐 100 米就有一家非常好吃的烤鸭店。”此时，你使用的是以这个路人为原点的坐标空间。想象一下，如果在这个世界上我们只能使用东南西北来描述所有东西的话，该会有多少人会被饿死。

由此可见，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。这也是为什么在渲染中我们要使用这么多坐标空间。

在开始介绍一些不同的坐标空间之前，读者需要注意，所有的坐标空间在理论上都是平等的，没有谁优谁劣之分，不会因为我们从一个坐标空间转换到另一个坐标空间计算就出错了。但是，在特定的情况下，一些坐标空间的确比另一些坐标空间更加吸引人。

现在，就让我们来看一下在游戏渲染流水线中，一个顶点到底经过了怎样的空间变换。

4.6.2 坐标空间的变换

我们先要为后面的内容做些数学铺垫。在渲染流水线中，我们往往需要把一个点或方向矢量从一个坐标空间转换到另一个坐标空间。这个过程到底是怎么实现的呢？

我们把问题一般化。我们知道，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。而这些数值实际上是相对于另一个坐标空间的（读者需要记住，所有的都是相对的）。也就是说，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，反过来

说，每个空间都有一个父（parent）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，现在有父坐标空间 **P** 以及一个子坐标空间 **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：

$$\mathbf{A}_p = \mathbf{M}_{c\to p}\,\mathbf{A}_c$$
$$\mathbf{B}_c = \mathbf{M}_{p\to c}\,\mathbf{B}_p$$

其中，$\mathbf{M}_{c\to p}$ 表示的是从子坐标空间变换到父坐标空间的变换矩阵，而 $\mathbf{M}_{p\to c}$ 是其逆矩阵（即反向变换）。那么，现在的问题就是，如何求解这些变换矩阵？事实上，我们只需要解出两者之一即可，另一个矩阵可以通过求逆矩阵的方式来得到。

下面，我们就来讲解如何求出从子坐标空间到父坐标空间的变换矩阵 $\mathbf{M}_{c\to p}$。

首先，我们来回顾一个看似很简单的问题：当给定一个坐标空间以及其中一点(a, b, c)时，我们是如何知道该点的位置的呢？我们可以通过 4 个步骤来确定它的位置：

（1）从坐标空间的原点开始；

（2）向 *x* 轴方向移动 *a* 个单位；

（3）向 *y* 轴方向移动 *b* 个单位；

（4）向 *z* 轴方向移动 *c* 个单位。

需要说明的是，上面的步骤只是我们的想象，这个点实际上并没有发生移动。上面的步骤看起来再简单不过了，坐标空间的变换就蕴含在上面的 4 个步骤中。现在，我们已知子坐标空间 **C** 的 3 个坐标轴在父坐标空间 **P** 下的表示 $\mathbf{x}_c$、$\mathbf{y}_c$、$\mathbf{z}_c$，以及其原点位置 $\boldsymbol{O}_c$。当给定一个子坐标空间中的一点 $\mathbf{A}_c = (a,b,c)$，我们同样可以依照上面 4 个步骤来确定其在父坐标空间下的位置 $\mathbf{A}_p$：

1．从坐标空间的原点开始

这很简单，我们已经知道了子坐标空间的原点位置 $\mathbf{O}_c$。

2．向 *x* 轴方向移动 *a* 个单位

仍然很简单，因为我们已经知道了 *x* 轴的矢量表示，因此可以得到

$$\mathbf{O}_c + a\mathbf{x}_c$$

3．向 *y* 轴方向移动 *b* 个单位

同样的道理，这一步就是：

$$\mathbf{O}_c + a\mathbf{x}_c + b\mathbf{y}_c$$

4．向 *z* 轴方向移动 *c* 个单位

最后，就可以得到

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

现在，我们已经求出了 $\mathbf{M}_{c\to p}$！什么？你没看出来吗？我们再来看一下最后得到的式子：

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

读者可能会问，这个式子里根本没有矩阵啊！其实我们只要稍稍使用一点“魔法”，矩阵就会出现在上面的式子中：

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

其中“|”符号表示是按列展开的。上面的式子实际上就是使用了我们之前所学的公式而已。但这个最后的表达式还不是很漂亮，因为还存在加法表达式，即平移变换。我们已经知道 3×3 的矩阵无法表示平移变换，因此为了得到一个更漂亮的结果，我们把上面的式子扩展到齐次坐标空间中，得

$$
\begin{aligned}
\mathbf{A}_p &= (x_{O_c}, y_{O_c}, z_{O_c}, 1) + \begin{bmatrix} | & | & | & 0 \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c & 0 \\ | & | & | & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & x_{O_c} \\ 0 & 1 & 0 & y_{O_c} \\ 0 & 0 & 1 & z_{O_c} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} | & | & | & 0 \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c & 0 \\ | & | & | & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} | & | & | & x_{O_c} \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c & y_{O_c} \\ | & | & | & z_{O_c} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} | & | & | & | \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c & O_c \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix}
\end{aligned}
$$

那么现在，你看到 $\mathbf{M}_{c\to p}$ 在哪里了吧？没错，

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

读者：这个看起来太神奇了！怎么就变着变着就出现了矩阵呢？

我们：上面只是运用了一些基础的矢量和矩阵运算，一旦当你真正理解了这些运算就会发现上面的过程只是简单地推导了一下而已。

一旦求出来 $\mathbf{M}_{c\to p}$，$\mathbf{M}_{p\to c}$ 就可以通过求逆矩阵的方式求出来，因为从坐标空间 **C** 变换到坐标

空间 **P** 与从坐标空间 **P** 变换到坐标空间 **C** 是互逆的两个过程。

可以看出来，变换矩阵 $\mathbf{M}_{c\to p}$ 实际上可以通过坐标空间 **C** 在坐标空间 **P** 中的原点和坐标轴的矢量表示来构建出来：把 3 个坐标轴依次放入矩阵的前 3 列，把原点矢量放到最后一列，再用 0 和 1 填充最后一行即可。

需要注意的是，这里我们并没有要求 3 个坐标轴 $\mathbf{x}_c$、$\mathbf{y}_c$ 和 $\mathbf{z}_c$ 是单位矢量，事实上，如果存在缩放的话，这三个矢量值很可能不是单位矢量。

更加令人振奋的是，我们可以利用反向思维，从这个变换矩阵反推来获取子坐标空间的原点和坐标轴方向！例如，当我们已知从模型空间到世界空间的一个 4×4 的变换矩阵，可以提取它的第一列再进行归一化后（为了消除缩放的影响）来得到模型空间的 x 轴在世界空间下的单位矢量表示。同样的方法可以提取 *y* 轴和 *z* 轴。我们可以从另一个角度来理解这个提取过程。因为矩阵 $\mathbf{M}_{c\to p}$ 可以把一个方向矢量从坐标空间 **C** 变换到坐标空间 **P** 中，那么，我们只需要用它来变换坐标空间 **C** 中的 x 轴(1, 0, 0, 0)，即使用矩阵乘法 $\mathbf{M}_{c\to p}[1 \quad 0 \quad 0 \quad 0]^T$，得到的结果正是 $\mathbf{M}_{c\to p}$ 的第一列。

另一个有趣的情况是，对方向矢量的坐标空间变换。我们知道，矢量是没有位置的，因此坐标空间的原点变换是可以忽略的。也就是说，我们仅仅平移坐标系的原点是不会对矢量造成任何影响的。那么，对矢量的坐标空间变换就可以使用 3×3 的矩阵来表示，因为我们不需要表示平移变换。那么变换矩阵就是：

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

在 Shader 中，我们常常会看到截取变换矩阵的前 3 行前 3 列来对法线方向、光照方向来进行空间变换，这正是原因所在。

现在，我们再来关注 $\mathbf{M}_{p\to c}$。我们前面讲到，可以通过求 $\mathbf{M}_{c\to p}$ 的逆矩阵的方式求解出来反向变换 $\mathbf{M}_{p\to c}$。但有一种情况我们不需要求解逆矩阵就可以得到 $\mathbf{M}_{p\to c}$，这种情况就是 $\mathbf{M}_{c\to p}$ 是一个正交矩阵。如果它是一个正交矩阵的话，$\mathbf{M}_{c\to p}$ 的逆矩阵就等于它的转置矩阵。这意味着我们不需要进行复杂的求逆操作就可以得到反向变换。也就是说，

$$\mathbf{M}_{p\to c}=\begin{bmatrix} | & | & | \\ \mathbf{x}_p & \mathbf{y}_p & \mathbf{z}_p \\ | & | & | \end{bmatrix}=\mathbf{M}_{c\to p}^{-1}=\mathbf{M}_{c\to p}^{T}$$
$$=\begin{bmatrix} - & \mathbf{x}_c & - \\ - & \mathbf{y}_c & - \\ - & \mathbf{z}_c & - \end{bmatrix}$$

而现在，我们不仅可以根据变换矩阵 $\mathbf{M}_{c\to p}$ 反推出子坐标空间的坐标轴方向在父坐标空间中的表示 $\mathbf{x}_c$、$\mathbf{y}_c$ 和 $\mathbf{z}_c$，还可以反推出父坐标空间的坐标轴方向在子坐标空间中的表示 $\mathbf{x}_p$、$\mathbf{y}_p$ 和 $\mathbf{z}_p$，这些坐标轴对应的就是 $\mathbf{M}_{c\to p}$ 的每一行！也就是说，如果我们知道坐标空间变换矩阵 $\mathbf{M}_{A\to B}$ 是一个正交矩阵，那么我们可以提取它的第一列来得到坐标空间 **A** 的 x 轴在坐标空间 **B** 下的表示，还可以提取它的第一行来得到坐标空间 **B** 的 x 轴在坐标空间 **A** 下的表示。反过来，如果我们知道坐标空间 **B** 的 x 轴、y 轴和 z 轴（必须是单位矢量，否则构建出来的就不是正交矩阵了）在坐标空间 **A** 下的表示，就可以把它们依次放在矩阵的每一行就可以得到从 **A** 到 **B** 的变换矩阵了。

读者：天呐，我的脑子已经完全乱掉了，一会儿从 P 到 C，一会儿又从 C 到 P，一会儿是行，一会儿又是列，我自己写的时候一定会搞不清楚！

我们：我们知道这个过程很容易造成思维的混乱，因此才要花费大量的篇幅来解释背后的数学原理。只有知道了这些原理，遇到疑问时你才知道怎样去验证结果的正确性。例如像下面这样。

当你不知道把坐标轴的表示是按行放还是按列放的时候，不妨先选择一种摆放方式来得到变换矩阵。例如，现在我们想把一个矢量从坐标空间 **A** 变换到坐标空间 **B**，而且我们已经知道坐标空间 **B** 的 x 轴、y 轴、z 轴在空间 **A** 下的表示，即 $\mathbf{x_B}$、$\mathbf{y_B}$ 和 $\mathbf{z_B}$。那么想要得到从 **A** 到 **B** 的变换矩阵 $\mathbf{M}_{A\to B}$，我们是把它们按列放呢还是按行放呢？如果读者实在想不起来正确答案，我们不妨先随便选择一种方式，例如按列摆放。那么，

$$\mathbf{M}_{A\to B}=\begin{bmatrix} | & | & | \\ \mathbf{x}_B & \mathbf{y}_B & \mathbf{z}_B \\ | & | & | \end{bmatrix}$$，注意，这个矩阵是不对的

现在，我们可以非常快速地来验证它是否是正确的。方法就是，用 $\mathbf{M}_{A\to B}$ 来变换 $\mathbf{x_B}$。在计算前我们先想一下这个结果，如果我们用变换矩阵来变换 **B** 的 x 轴的话，那么结果应该是(1,0,0)才对。因为当变换到空间 B 中时，x 轴的指向就是(1, 0, 0)。好了，我们可以来进行真正的计算来验证它了：

$$\mathbf{M}_{A\to B}\mathbf{x_B}=\begin{bmatrix} | & | & | \\ \mathbf{x}_B & \mathbf{y}_B & \mathbf{z}_B \\ | & | & | \end{bmatrix}\mathbf{x_B}$$

读者看到这里会有疑问，“我不知道这个结果是什么啊”。没错，这不是你的计算有问题，而是上式的计算结果的确不可知。这种时候你就会发现我们的摆放方式选择错了。现在，我们使用正确的摆放方式，即按行来摆放，那么就有：

$$\mathbf{M}_{A\to B}\mathbf{x_B}=\begin{bmatrix} - & \mathbf{x}_B & - \\ - & \mathbf{y}_B & - \\ - & \mathbf{z}_B & - \end{bmatrix}\mathbf{x_B}=\begin{bmatrix} \mathbf{x}_B\cdot\mathbf{x}_B \\ \mathbf{y}_B\cdot\mathbf{x}_B \\ \mathbf{z}_B\cdot\mathbf{x}_B \end{bmatrix}=\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

这次结果就和我们预期的一样了。

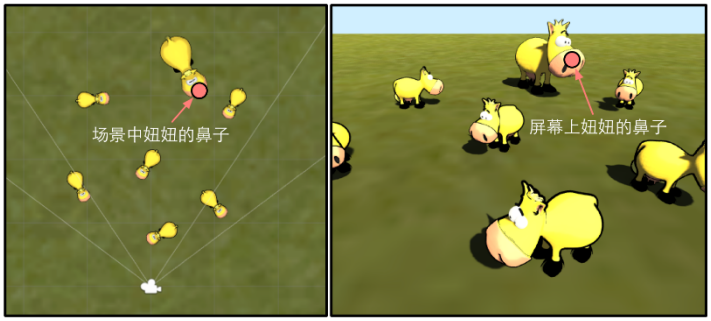
理解上面的原理和过程非常重要。我们在本书的后面也会经常遇到坐标空间的变换。

4.6.3 顶点的坐标空间变换过程

我们知道，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在屏幕上。一个顶点最开始是在模型空间（见 4.6.4 节）中定义的，最后它将会变换到屏幕空间（见 4.6.8 节）中，得到真正的屏幕像素坐标。因此，接下来的内容我们将解释顶点要进行的各种空间变换的过程。

为了帮助读者理解这个过程，我们将建立在农场游戏的实例背景下，每讲到一种空间变换，我们会解释如何应用到这个案例中。

在我们的农场游戏中，妞妞很好奇自己是如何被渲染到屏幕上的。它只知道自己和一群小伙伴在农场里快乐地吃草，而前面有一个摄像机一直在观察它们，如图 4.31 所示。妞妞特别喜欢自己的鼻子，它想知道鼻子是怎么被画到屏幕上的？



▲图 4.31 场景中的妞妞（左图）和屏幕上的妞妞（右图）。妞妞想知道，自己的鼻子是如何被画到屏幕上的

在下面的内容中，我们将了解妞妞的鼻子是如何一步步画到屏幕上的。

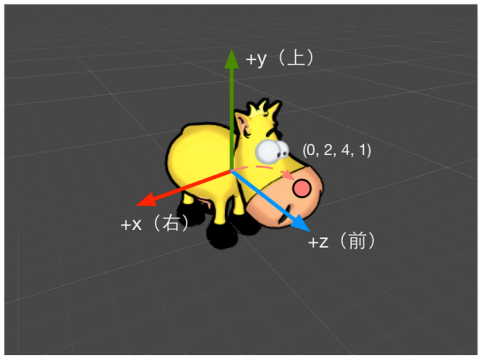
4.6.4 模型空间

模型空间（model space），如它的名字所暗示的那样，是和某个模型或者说是对象有关的。有时模型空间也被称为**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，当它移动或旋转的时候，模型空间也会跟着它移动和旋转。把我们自己当成游戏中的模型的话，当我们在办公室里移动时，我们的模型空间也在跟着移动，当我们转身时，我们本身的前后左右方向也在跟着改变。

在模型空间中，我们经常使用一些方向概念，例如“前（forward）”、“后（back）”、“左（left）”、“右（right）”、“上（up）”、“下（down）”。在本书中，我们把这些方向称为自然方向。模型空间中的坐标轴通常会使用这些自然方向。在 4.2.4 节中我们讲过，Unity 在模型空间中使用的是左手坐标系，因此在模型空间中，$+x$ 轴、$+y$ 轴、$+z$ 轴分别对应的是模型的右、上和前向。需要注意的是，模型坐标空间中的 x 轴、y 轴、z 轴和自然方向的对应不一定是上述这种关系，但由于 Unity 使用的是这样的约定，因此本书将使用这种方式。我们可以在 Hierarchy 视图中单击任意对象就可以看见它们对应的模型空间的 3 个坐标轴。

模型空间的原点和坐标轴通常是由美术人员在建模软件里确定好的。当导入到 Unity 中后，我们可以在顶点着色器中访问到模型的顶点信息，其中包含了每个顶点的坐标。这些坐标都是相对于模型空间中的原点（通常位于模型的重心）定义的。

当我们把妞妞放到场景中时，就会有一个模型坐标空间时刻跟随着它。妞妞鼻子的位置可以通过访问顶点属性来得到。假设这个位置是(0, 2, 4)，由于顶点变换中往往包含了平移变换，因此需要把其扩展到齐次坐标系下，得到顶点坐标是(0, 2, 4, 1)，如图 4.32 所示。



▲图 4.32 在我们的农场游戏中，每个奶牛都有自己的模型坐标系。在模型坐标系中妞妞鼻子的位置是(0, 2, 4, 1)

4.6.5 世界空间

世界空间（world space）是一个特殊的坐标系，因为它建立了我们所关心的最大的空间。一些读者可能会指出，空间可以是无限大的，怎么会有“最大”这一说呢？这里说的最大指的是一

个宏观的概念，也就是说它是我们所关心的最外层的坐标空间。以我们的农场游戏为例，在这个游戏里世界空间指的就是农场，我们不关心这个农场是在什么地方，在这个虚拟的游戏世界里，农场就是最大的空间概念。

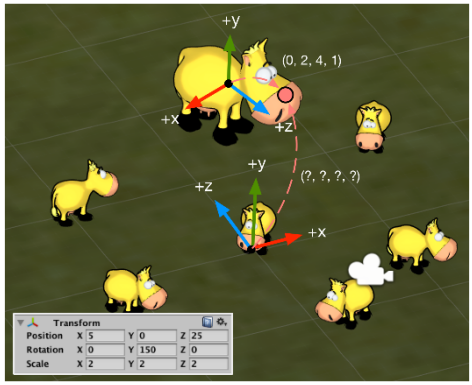
世界空间可以被用于描述绝对位置（较真的读者可能会再一次提醒我，没有绝对的位置。没错，但我相信读者可以明白这里绝对的意思）。在本书中，绝对位置指的就是在世界坐标系中的位置。通常，我们会把世界空间的原点放置在游戏空间的中心。

在 Unity 中，世界空间同样使用了左手坐标系。但它的 x 轴、y 轴、z 轴是固定不变的。在 Unity 中，我们可以通过调整 Transform 组件中的 Position 属性来改变模型的位置，这里的位置指的是相对于这个 Transform 的**父节点（parent）**的模型坐标空间中的原点定义的。如果一个 Transform 没有任何父节点，那么这个位置就是在世界坐标系中的位置，如图 4.33 所示。我们可以想象成还有一个虚拟的根模型，这个根模型的模型空间就是世界空间，所有的游戏对象都附属于这个根模型。同样，Transform 中的 Rotation 和 Scale 也是同样的道理。

顶点变换的第一步，就是将顶点坐标从模型空间变换到世界空间中。这个变换通常叫做**模型变换（model transform）**。

现在，我们来对妞妞的鼻子进行模型变换。为此，我们首先需要知道妞妞在世界坐标系中进行了哪些变换，这可以通过面板中的 Transform 组件来得到相关的变换信息，如图 4.34 所示。

▲图 4.33 Unity 的 Transform 组件可以调节模型的位置。如果 Transform 有父节点，如图中的"Mesh"，那么 Position 将是在其父节点（这里是"Cow"）的模型空间中的位置；如果没有父节点，Position 就是在世界空间中的位置



▲图 4.34 农场游戏中的世界空间。世界空间的原点被放置在农场的中心。左下角显示了妞妞在世界空间中所做的变换。我们想要把妞妞的鼻子从模型空间变换到世界空间中

根据 Transform 组件上的信息，我们知道在世界空间中，妞妞进行了(2, 2, 2)的缩放，又进行了(0, 150, 0)的旋转以及(5, 0, 25)的平移。注意这里的变换顺序是不能互换的，即先进行缩放，再进行旋转，最后是平移。据此我们可以构建出模型变换的变换矩阵：

$$\mathbf{M}_{model} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 25 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -0.866 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \\ -0.5 & 0 & -0.866 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} -1.732 & 0 & 1 & 5 \\ 0 & 2 & 0 & 0 \\ -1 & 0 & -1.732 & 25 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

现在我们可以用它来对妞妞的鼻子进行模型变换了：

$$\mathbf{P}_{world} = \mathbf{M}_{model}\mathbf{P}_{model}$$

$$= \begin{bmatrix} -1.732 & 0 & 1 & 5 \\ 0 & 2 & 0 & 0 \\ -1 & 0 & -1.732 & 25 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 9 \\ 4 \\ 18.072 \\ 1 \end{bmatrix}$$

也就是说，在世界空间下，妞妞鼻子的位置是(9, 4, 18.072)。注意，这里的浮点数都是近似值，这里近似到小数点后 3 位。实际数值和 Unity 采用的浮点值精度有关。

4.6.6 观察空间

观察空间（view space）也被称为**摄像机空间（camera space）**。观察空间可以认为是模型空间的一个特例——在所有的模型中有一个非常特殊的模型，即摄像机（虽然通常来说摄像机本身是不可见的），它的模型空间值得我们单独拿出来讨论，也就是观察空间。

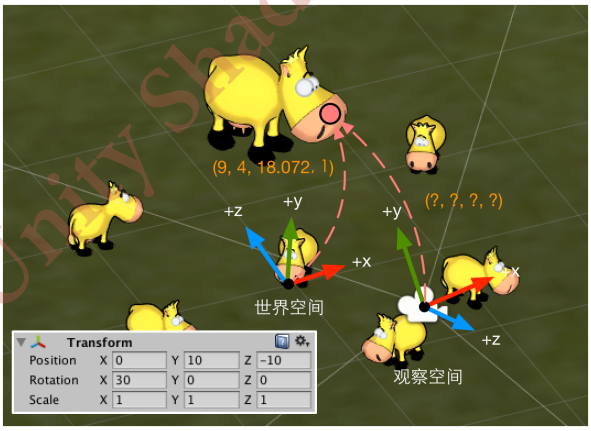
摄像机决定了我们渲染游戏所使用的视角。在观察空间中，摄像机位于原点，同样，其坐标轴的选择可以是任意的，但由于本书讨论的是以 Unity 为主，而 Unity 中观察空间的坐标轴选择是：+*x* 轴指向右方，+*y* 轴指向上方，而+*z* 轴指向的是摄像机的后方。读者在这里可能觉得很奇怪，我们之前讨论的模型空间和世界空间中+*z* 轴指的都是物体的前方，为什么这里不一样了呢？这是因为，Unity 在模型空间和世界空间中选用的都是左手坐标系，而在观察空间中使用的是右手坐标系。这是符合 OpenGL 传统的，在这样的观察空间中，摄像机的正前方指向的是-*z* 轴方向。

这种左右手坐标系之间的改变很少会对我们在 Unity 中的编程产生影响，因为 Unity 为我们做了很多渲染的底层工作，包括很多坐标空间的转换。但是，如果读者需要调用类似 Camera.cameraToWorldMatrix、Camera.worldToCameraMatrix 等接口自行计算某模型在观察空间中的位置，就要小心这样的差异。

最后要提醒读者的一点是，观察空间和屏幕空间（详见 4.6.8 节）是不同的。观察空间是一个三维空间，而屏幕空间是一个二维空间。从观察空间到屏幕空间的转换需要经过一个操作，那就是**投影（projection）**。我们后面就会讲到。

顶点变换的第二步，就是将顶点坐标从世界空间变换到观察空间中。这个变换通常叫做**观察变换（view transform）**。

回到我们的农场游戏。现在我们需要把妞妞的鼻子从世界空间变换到观察空间中。为此，我们需要知道世界坐标系下摄像机的变换信息。这同样可以通过摄像机面板中的 Transform 组件得到，如图 4.35 所示。



▲图 4.35 农场游戏中摄像机的观察空间。观察空间的原点位于摄像机处。注意在观察空间中，摄像机的前向是 *z* 轴的负方向（图中只画出了 *z* 轴正方向），这是因为 Unity 在观察空间中使用了右手坐标系。左下角显示了摄像机在世界空间中所做的变换。我们想要把妞妞的鼻子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，我们可以有两种方法。一种方法是计算观察空间的三个坐标轴在世界空间下的表示，然后根据 4.6.2 节中讲到的方法，构建出从观察空间变换到世界空间的变换矩阵，再对该矩阵求逆来得到从世界空间变换到观察空间的变换矩阵。我们还可以使用另

一种方法，即想象平移整个观察空间，让摄像机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。这两种方法得到的变换矩阵都是一样的，不同的只是我们思考的方式。

这里我们使用第二种方法。由 Transform 组件可以知道，摄像机在世界空间中的变换是先按(30, 0, 0)进行旋转，然后按(0, 10, −10)进行了平移。那么，为了把摄像机重新移回到初始状态（这里指摄像机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），我们需要进行逆向变换，即先按(0, −10, 10)平移，以便将摄像机移回到原点，再按(−30, 0, 0)进行旋转，以便让坐标轴重合。因此，变换矩阵就是：

$$\mathbf{M}_{view}=\begin{bmatrix}1&0&0&0\\0&\cos\theta&-\sin\theta&0\\0&\sin\theta&\cos\theta&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&t_x\\0&1&0&t_y\\0&0&1&t_z\\0&0&0&1\end{bmatrix}$$

$$=\begin{bmatrix}1&0&0&0\\0&0.866&0.5&0\\0&-0.5&0.866&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&1&0&-10\\0&0&1&10\\0&0&0&1\end{bmatrix}$$

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

但是，由于观察空间使用的是右手坐标系，因此需要对 z 分量进行取反操作。我们可以通过乘以另一个特殊的矩阵来得到最终的观察变换矩阵：

$$\mathbf{M}_{view}=\mathbf{M}_{negate\,z}\,\mathbf{M}_{view}$$

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

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

现在我们可以用它来对妞妞的鼻子进行顶点变换了：

$$\mathbf{P}_{view}=\mathbf{M}_{view}\,\mathbf{P}_{world}$$

$$=\begin{bmatrix}1&0&0&0\\0&0.866&0.5&-3.66\\0&0.5&-0.866&-13.66\\0&0&0&1\end{bmatrix}\begin{bmatrix}9\\4\\18.072\\1\end{bmatrix}=\begin{bmatrix}9\\8.84\\-27.31\\1\end{bmatrix}$$

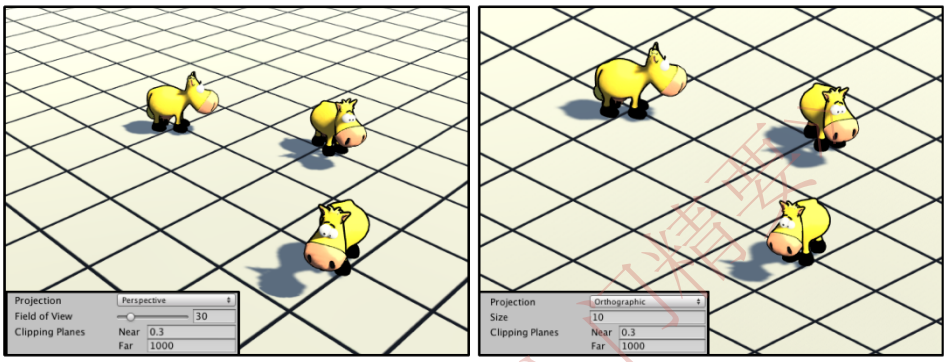
这样，我们就得到了观察空间中妞妞鼻子的位置——(9, 8.84, −27.31)。

4.6.7 裁剪空间

顶点接下来要从观察空间转换到**裁剪空间（clip space，也被称为齐次裁剪空间）**中，这个用于变换的矩阵叫做**裁剪矩阵（clip matrix）**，也被称为投影矩阵**（projection matrix）**。

裁剪空间的目标是能够方便地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，而与这块空间边界相交的图元就会被裁剪。那么，这块空间是如何决定的呢？答案是由**视锥体（view frustum）**来决定。

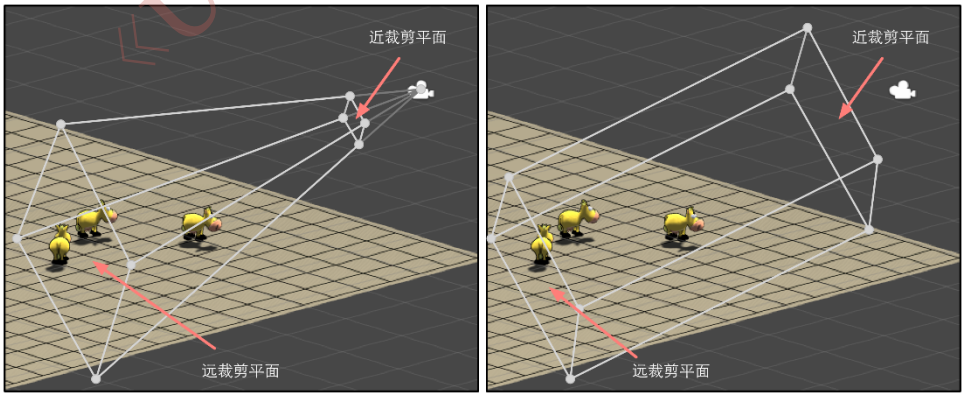
视锥体指的是空间中的一块区域，这块区域决定了摄像机可以看到的空间。视锥体由六个平面包围而成，这些平面也被称为**裁剪平面（clip planes）**。视锥体有两种类型，这涉及两种投影类型：一种是**正交投影（orthographic projection）**，一种是**透视投影（perspective projection）**。图 4.36 显示了从同一位置、同一角度渲染同一个场景的两种摄像机的渲染结果。



▲图 4.36　透视投影（左图）和正交投影（右图）。左下角分别显示了当前摄像机的投影模式和相关属性

从图中可以发现，在透视投影中，地板上的平行线并不会保持平行，离摄像机越近网格越大，离摄像机越远网格越小。而在正交投影中，所有的网格大小都一样，而且平行线会一直保持平行。可以注意到，透视投影模拟了人眼看世界的方式，而正交投影则完全保留了物体的距离和角度。因此，在追求真实感的 3D 游戏中我们往往会使用透视投影，而在一些 2D 游戏或渲染小地图等其他 HUD 元素时，我们会使用正交投影。

在视锥体的 6 块裁剪平面中，有两块裁剪平面比较特殊，它们分别被称为**近剪裁平面（near clip plane）**和**远剪裁平面（far clip plane）**。它们决定了摄像机可以看到的深度范围。正交投影和透视投影的视锥体如图 4.37 所示。



▲图 4.37　视锥体和裁剪平面。左图显示了透视投影的视锥体，右图显示了正交投影的视锥体

由图可以看出，透视投影的视锥体是一个金字塔形，侧面的四个裁剪平面将会在摄像机处相交。它更符合视锥体这个词语。正交投影的视锥体是一个长方体。前面讲到，我们希望根据视锥体围成的区域对图元进行裁剪，但是，如果直接使用视锥体定义的空间来进行裁剪，那么不同的视锥体就需要不同的处理过程，而且对于透视投影的视锥体来说，想要判断一个顶点是否处于一个金字塔内部是比较麻烦的。因此，我们想用一种更加通用、方便和整洁的方式来进行裁剪的工作，这种方式就是通过一个投影矩阵把顶点转换到一个裁剪空间中。

投影矩阵有两个目的：

- 首先是为投影做准备。这是个迷惑点，虽然投影矩阵的名称包含了投影二字，但是它并没有进行真正的投影工作，而是在为投影做准备。真正的投影发生在后面的**齐次除法（homogeneous division）**过程中。而经过投影矩阵的变换后，顶点的 w 分量将会具有特殊的意义。

读者：投影到底是什么意思呢？

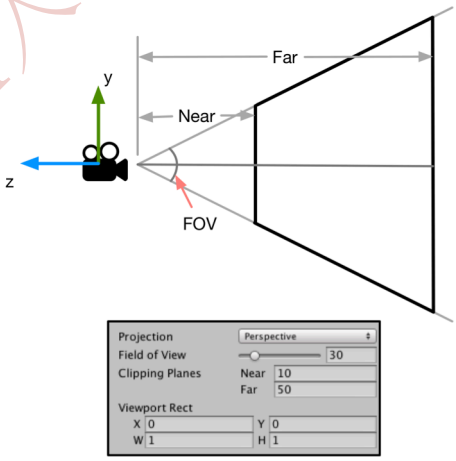
我们：可以理解成是一个空间的降维，例如从四维空间投影到三维空间中。而投影矩阵实际上并不会真的进行这个步骤，它会为真正的投影做准备工作。真正的投影会在屏幕映射时发生，通过齐次除法来得到二维坐标。具体会在 4.6.8 节中讲到。

- 其次是对 x、y、z 分量进行缩放。我们上面讲过直接使用视锥体的 6 个裁剪平面来进行裁剪会比较麻烦。而经过投影矩阵的缩放后，我们可以直接使用 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，就说明该顶点位于裁剪空间内。

在裁剪空间之前，虽然我们使用了齐次坐标来表示点和矢量，但它们的第四个分量都是固定的：点的 w 分量是 1，方向矢量的 w 分量是 0。经过投影矩阵的变换后，我们就会赋予齐次坐标的第 4 个坐标更加丰富的含义。下面，我们来看一下两种投影类型使用的投影矩阵具体是什么。

1．透视投影

视锥体的意义在于定义了场景中的一块三维空间。所有位于这块空间内的物体将会被渲染，否则就会被剔除或裁剪。我们已经知道，这块区域由 6 个裁剪平面定义，那么这 6 个裁剪平面又是怎么决定的呢？在 Unity 中，它们由 Camera 组件中的参数和 Game 视图的横纵比共同决定，如图 4.38 所示。



▲图 4.38　透视摄像机的参数对透视投影视锥体的影响

由图 4.38 可以看出，我们可以通过 Camera 组件的 Field of View（简称 FOV）属性来改变视锥体竖直方向的张开角度，而 Clipping Planes 中的 Near 和 Far 参数可以控制视锥体的近裁剪平面和远裁剪平面距离摄像机的远近。这样，我们可以求出视锥体近裁剪平面和远裁剪平面的高度，也就是：

$$nearClipPlaneHeight = 2 \cdot Near \cdot \tan\frac{FOV}{2}$$

$$farClipPlaneHeight = 2 \cdot Far \cdot \tan\frac{FOV}{2}$$

现在我们还缺乏横向的信息。这可以通过摄像机的横纵比得到。在 Unity 中，一个摄像机的

横纵比由 Game 视图的横纵比和 Viewport Rect 中的 W 和 H 属性共同决定（实际上，Unity 允许我们在脚本里通过Camera.aspect进行更改，但这里不做讨论）。假设，当前摄像机的横纵比为 Aspect，我们定义：

$$Aspect = \frac{nearClipPlaneWidth}{nearClipPlaneHeight}$$

$$Aspect = \frac{farClipPlaneWidth}{farClipPlaneHeight}$$

现在，我们可以根据已知的 Near、Far、FOV 和 Aspect 的值来确定透视投影的投影矩阵。如下：

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

上面公式的推导部分可以参见本章的扩展阅读部分。需要注意的是，这里的投影矩阵是建立在 Unity 对坐标系的假定上面的，也就是说，我们针对的是观察空间为右手坐标系，使用列矩阵在矩阵右侧进行相乘，且变换后 z 分量范围将在[-w, w]之间的情况。而在类似 DirectX 这样的图形接口中，它们希望变换后 z 分量范围将在[0, w]之间，因此就需要对上面的透视矩阵进行一些更改。这不在本书的讨论范围内。

而一个顶点和上述投影矩阵相乘后，可以由观察空间变换到裁剪空间中，结果如下：

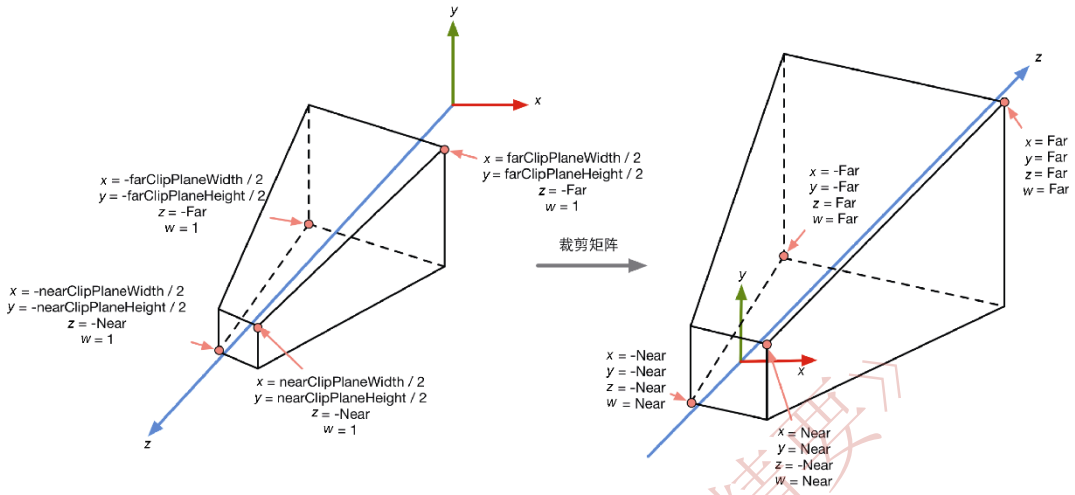
$$\begin{aligned} \mathbf{P}_{clip} &= \mathbf{M}_{frustum}\mathbf{P}_{view} \\ &= \begin{bmatrix} \frac{\cot\frac{FOV}{2}}{Aspect} & 0 & 0 & 0 \\ 0 & \cot\frac{FOV}{2} & 0 & 0 \\ 0 & 0 & -\frac{Far+Near}{Far-Near} & -\frac{2\cdot Near\cdot Far}{Far-Near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} x\frac{\cot\frac{FOV}{2}}{Aspect} \\ y\cot\frac{FOV}{2} \\ -z\frac{Far+Near}{Far-Near} - \frac{2\cdot Near\cdot Far}{Far-Near} \\ -z \end{bmatrix} \end{aligned}$$

从结果可以看出，这个投影矩阵本质就是对 x、y 和 z 分量进行了不同程度的缩放（当然，z 分量还做了一个平移），缩放的目的是为了方便裁剪。我们可以注意到，此时顶点的 w 分量不再是 1，而是原先 z 分量的取反结果。现在,我们就可以按如下不等式来判断一个变换后的顶点是否位于视锥体内。如果一个顶点在视锥体内，那么它变换后的坐标必须满足：

$$-w \leqslant x \leqslant w$$

$$-w \leqslant y \leqslant w$$
$$-w \leqslant z \leqslant w$$

任何不满足上述条件的图元都需要被剔除或者裁剪。图 4.39 显示了经过上述投影矩阵后，视锥体的变化。

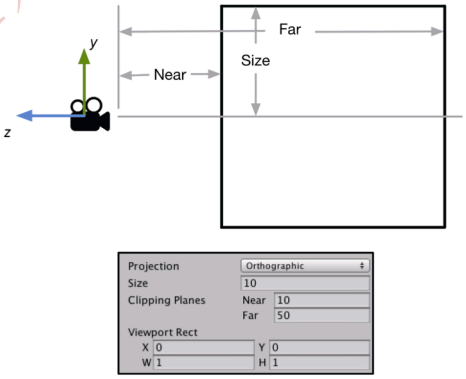


▲图 4.39　在透视投影中，投影矩阵对顶点进行了缩放。图中标注了 4 个关键点经过投影矩阵变换后的结果。从这些结果可以看出 *x*、*y*、*z* 和 *w* 分量的范围发生的变化

从图 4.39 还可以注意到，裁剪矩阵会改变空间的旋向性：空间从右手坐标系变换到了左手坐标系。这意味着，离摄像机越远，*z* 值将越大。

2．正交投影

首先，我们还是看一下正交投影中的 6 个裁剪平面是如何定义的。和透视投影类似，在 Unity 中，它们也是由 Camera 组件中的参数和 Game 视图的横纵比共同决定，如图 4.40 所示。



▲图 4.40　正交摄像机的参数对正交投影视锥体的影响

正交投影的视锥体是一个长方体，因此计算上相比透视投影来说更加简单。由图可以看出，我们可以通过 Camera 组件的 Size 属性来改变视锥体竖直方向上高度的一半，而 Clipping Planes 中的 Near 和 Far 参数可以控制视锥体的近裁剪平面和远裁剪平面距离摄像机的远近。这样，我们可以求出视锥体近裁剪平面和远裁剪平面的高度，也就是：

$$nearClipPlaneHeight = 2 \cdot Size$$
$$farClipPlaneHeight = nearClipPlaneHeight$$

现在我们还缺乏横向的信息。同样，我们可以通过摄像机的横纵比得到。假设，当前摄像机的横纵比为 Aspect，那么：

$$nearClipPlaneWidth = Aspect \cdot nearClipPlaneHeight$$
$$farClipPlaneWidth = nearClipPlaneWidth$$

现在，我们可以根据已知的 Near、Far、Size 和 Aspect 的值来确定正交投影的裁剪矩阵。如下：

$$\mathbf{M}_{ortho}=\begin{bmatrix}\frac{1}{Aspect\cdot Size} & 0 & 0 & 0\\ 0 & \frac{1}{Size} & 0 & 0\\ 0 & 0 & -\frac{2}{Far-Near} & -\frac{Far+Near}{Far-Near}\\ 0 & 0 & 0 & 1\end{bmatrix}$$

上面公式的推导部分可以参见本章的扩展阅读部分。同样，这里的投影矩阵是建立在 Unity 对坐标系的假定上面的。

一个顶点和上述投影矩阵相乘后的结果如下：

$$\mathbf{P}_{clip}=\mathbf{M}_{ortho}\,\mathbf{P}_{view}$$

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

$$=\begin{bmatrix}\frac{x}{Aspect\cdot Size}\\ \frac{y}{Size}\\ -\frac{2z}{Far-Near}-\frac{Far+Near}{Far-Near}\\ 1\end{bmatrix}$$

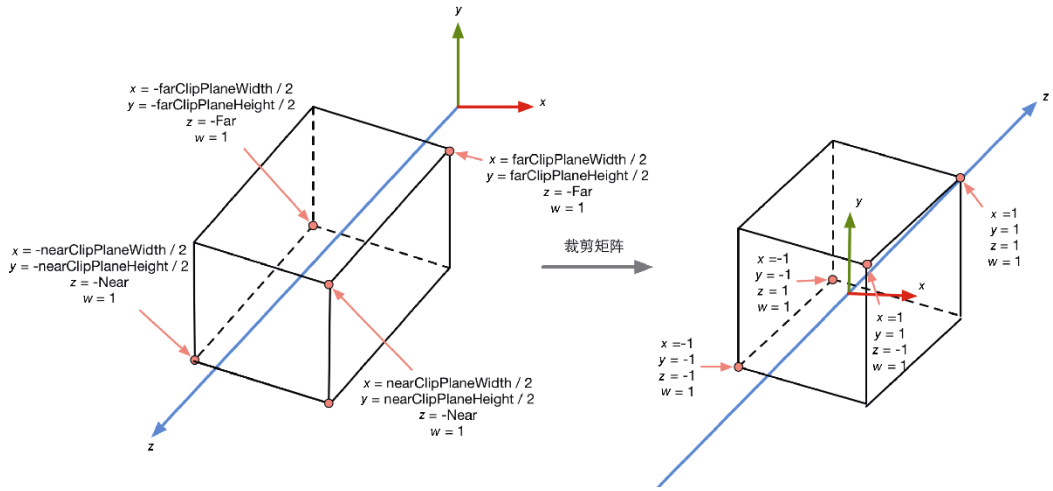
注意到，和透视投影不同的是，使用正交投影的投影矩阵对顶点进行变换后，其 w 分量仍然为 1。本质是因为投影矩阵最后一行的不同，透视投影的投影矩阵的最后一行是[0　0　−1　0]，而正交投影的投影矩阵的最后一行是[0　0　0　1]。这样的选择是有原因的，是为了为齐次除法做准备。具体会在下一节中讲到。

判断一个变换后的顶点是否位于视锥体内使用的不等式和透视投影中的一样，这种通用性也是为什么要使用投影矩阵的原因之一。图 4.41 显示了经过上述投影矩阵后，正交投影的视锥体的变化。

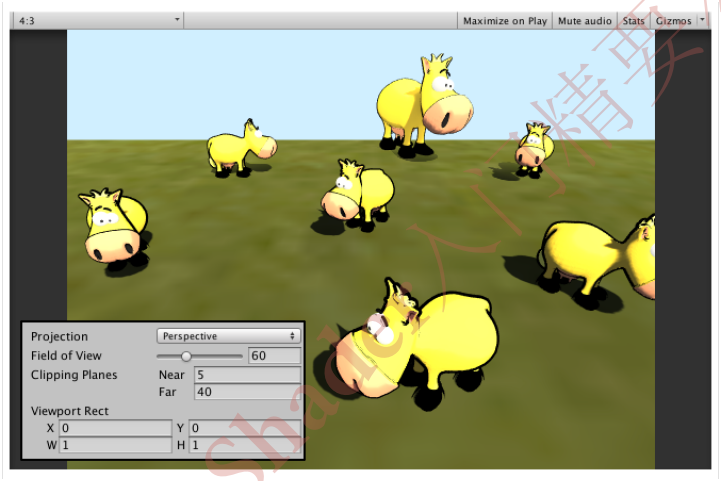
同样，裁剪矩阵改变了空间的旋向性。可以注意到，经过正交投影变换后的顶点实际已经位于一个立方体内了。

希望看到这里读者的脑袋还没有爆炸。现在，我们继续来看我们的农场游戏。在 4.6.6 节的最后，我们已经帮助妞妞确定了它的鼻子在观察空间中的位置——(9, 8.84, -27.31)。现在，我们要计算它在裁剪空间中的位置。

首先，我们需要知道农场游戏中使用的摄像机类型。由于农场游戏是一个 3D 游戏，因此这里我们使用了透视摄像机。摄像机参数和 Game 视图的横纵比如图 4.42 所示。



▲图 4.41　在正交投影中，投影矩阵对顶点进行了缩放。图中标注了 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* 分量范围发生的变化。



▲图 4.42　农场游戏使用的摄像机参数和游戏画面的横纵比

据此，我们可以知道透视投影的参数：FOV 为 60°，Near 为 5，Far 为 40，Aspect 为 4/3 = 1.333。那么，对应的投影矩阵就是：

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

$$=\begin{bmatrix}1.299 & 0 & 0 & 0\\ 0 & 1.732 & 0 & 0\\ 0 & 0 & -1.286 & -11.429\\ 0 & 0 & -1 & 0\end{bmatrix}$$

然后，我们用这个投影矩阵来把妞妞的鼻子从观察空间转换到裁剪空间中。如下：

$$\mathbf{P}_{clip}=\mathbf{M}_{frustum}\mathbf{P}_{view}$$

$$=\begin{bmatrix}1.299 & 0 & 0 & 0\\ 0 & 1.732 & 0 & 0\\ 0 & 0 & -1.286 & -11.429\\ 0 & 0 & -1 & 0\end{bmatrix}\begin{bmatrix}9\\ 8.84\\ -27.31\\ 1\end{bmatrix}=\begin{bmatrix}11.691\\ 15.311\\ 23.692\\ 27.31\end{bmatrix}$$

这样，我们就求出了妞妞的鼻子在裁剪空间中的位置——(11.691, 15.311, 23.692, 27.31)。接下来，Unity 会判断妞妞的鼻子是否需要裁剪。通过比较得到，妞妞的鼻子满足下面的不等式：

$$-w \leqslant x \leqslant w \rightarrow -27.31 \leqslant 11.691 \leqslant 27.31$$
$$-w \leqslant x \leqslant y \rightarrow -27.31 \leqslant 15.311 \leqslant 27.31$$
$$-w \leqslant z \leqslant w \rightarrow -27.31 \leqslant 23.692 \leqslant 27.31$$

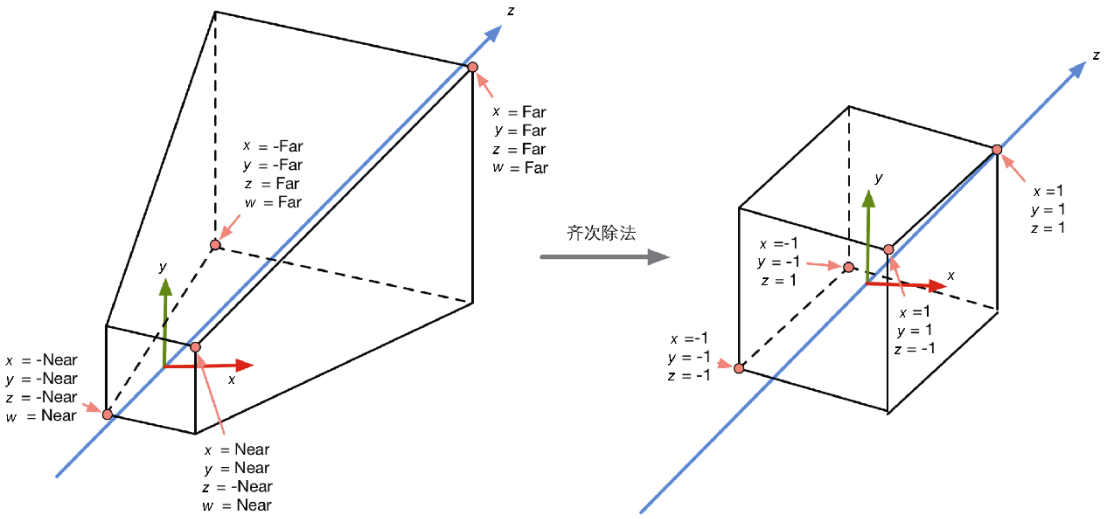
由此，我们可以判断，妞妞的鼻子位于视锥体内，不需要被裁剪。

4.6.8 屏幕空间

经过投影矩阵的变换后，我们可以进行裁剪操作。当完成了所有的裁剪工作后，就需要进行真正的投影了，也就是说，我们需要把视锥体投影到**屏幕空间（screen space）**中。经过这一步变换，我们会得到真正的像素位置，而不是虚拟的三维坐标。

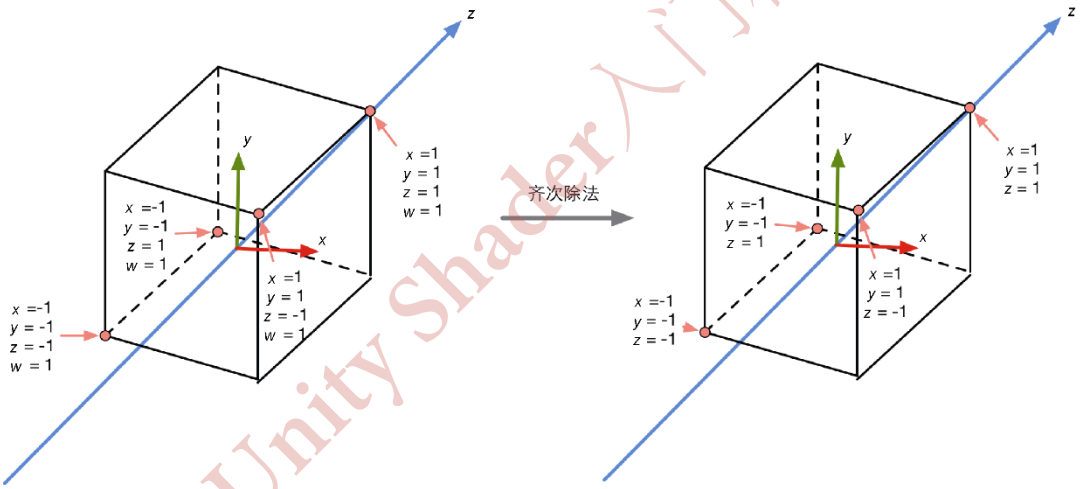
屏幕空间是一个二维空间，因此，我们必须把顶点从裁剪空间投影到屏幕空间中，来生成对应的 2D 坐标。这个过程可以理解成有两个步骤。

首先，我们需要进行标准**齐次除法（homogeneous division）**，也被称为**透视除法（perspective division）**。虽然这个步骤听起来很陌生，但是它实际上非常简单，就是用齐次坐标系的 w 分量去除以 x、y、z 分量。在 OpenGL 中，我们把这一步得到的坐标叫做**归一化的设备坐标（Normalized Device Coordinates，NDC）**。经过这一步，我们可以把坐标从齐次裁剪坐标空间转换到 NDC 中。经过透视投影变换后的裁剪空间，经过齐次除法后会变换到一个立方体内。按照 OpenGL 的传统，这个立方体的 x、y、z 分量的范围都是[−1, 1]。但在 DirectX 这样的 API 中，z 分量的范围会是[0, 1]。而 Unity 选择了 OpenGL 这样的齐次裁剪空间。如图 4.43 所示。



▲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

而对于正交投影来说，它的裁剪空间实际已经是一个立方体了，而且由于经过正交投影矩阵变换后的顶点的 w 分量是 1，因此齐次除法并不会对顶点的 x、y、z 坐标产生影响。如图 4.44 所示。



▲图 4.44　经过齐次除法后，正交投影的裁剪空间会变换到一个立方体。

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。

在 Unity 中，屏幕空间左下角的像素坐标是(0, 0)，右上角的像素坐标是(pixelWidth, pixelHeight)。由于当前 x 和 y 坐标都是[−1, 1]，因此这个映射的过程就是一个缩放的过程。

齐次除法和屏幕映射的过程可以使用下面的公式来总结：

$$screen_x = \frac{clip_x \cdot pixelWidth}{2 \cdot clip_w} + \frac{pixelWidth}{2}$$
$$screen_y = \frac{clip_y \cdot pixelHeight}{2 \cdot clip_w} + \frac{pixelHeight}{2}$$

上面的式子对 x 和 y 分量都进行了处理，那么 z 分量呢？通常，z 分量会被用于深度缓冲。一个传统的方式是把 $\frac{clip_z}{clip_w}$ 的值直接存进深度缓冲中，但这并不是必须的。通常驱动生产商会根据硬件来选择最好的存储格式。此时 $clip_w$ 也并不会被抛弃，虽然它已经完成了它的主要工作——在齐次除法中作为分母来得到 NDC，但它仍然会在后续的一些工作中起到重要的作用，例如进行透视校正插值。

在 Unity 中，从裁剪空间到屏幕空间的转换是由底层帮我们完成的。我们的顶点着色器只需要把顶点转换到裁剪空间即可。

在上一步中，我们知道了裁剪空间中妞妞鼻子的位置——(11.691, 15.311, 23.692, 27.31)。现在，我们终于可以确定妞妞的鼻子在屏幕上的像素位置。假设，当前屏幕的像素宽度为 400，高度为 300。首先，我们需要进行齐次除法，把裁剪空间的坐标投影到 NDC 中。然后，再映射到屏幕空间中。这个过程如下：

$$\begin{aligned} screen_x &= \frac{clip_x \cdot pixelWidth}{2 \cdot clip_w} + \frac{pixelWidth}{2} \\ &= \frac{11.691 \cdot 400}{2 \cdot 27.31} + \frac{400}{2} \\ &= 285.617 \end{aligned}$$
$$\begin{aligned} screen_y &= \frac{clip_y \cdot pixelHeight}{2 \cdot clip_w} + \frac{pixelHeight}{2} \\ &= \frac{15.311 \cdot 300}{2 \cdot 27.31} + \frac{300}{2} \\ &= 234.096 \end{aligned}$$

由此，我们知道了妞妞鼻子在屏幕上的位置——(285.617, 234.096)。

4.6.9 总结

以上就是一个顶点如何从模型空间变换到屏幕坐标的过程。图 4.45 总结了这些空间和用于变换的矩阵。

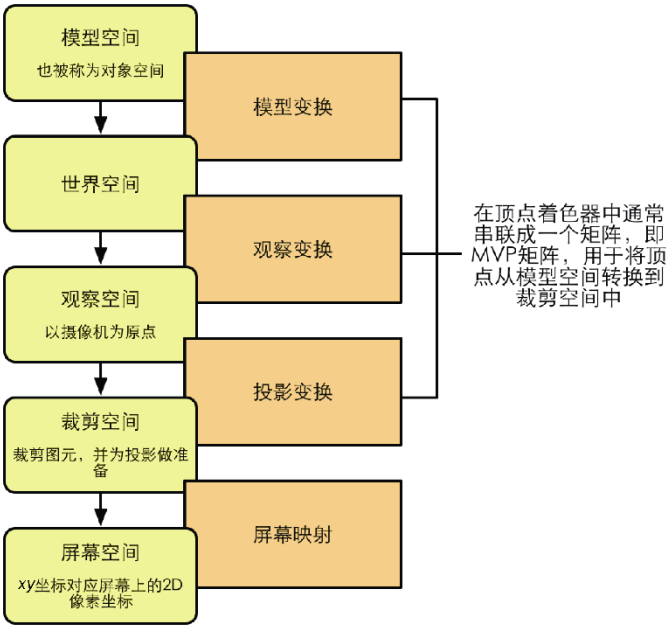
顶点着色器的最基本的任务就是把顶点坐标从模型空间转换到裁剪空间中。这对应了图 4.45 中的前三个顶点变换过程。而在片元着色器中，我们通常也可以得到该片元在屏幕空间的像素位置。我们会在 4.9.3 节中看到如何得到这些像素位置。

在 Unity 中，坐标系的旋向性也随着变换发生了改变。图 4.46 总结了 Unity 中各个空间使用的坐标系旋向性。

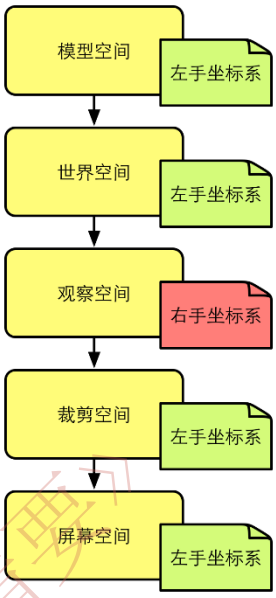
从图 4.46 中可以发现，只有在观察空间中 Unity 使用了右手坐标系。

需要注意的是，这里仅仅给出的是一些最重要的坐标空间。还有一些空间在实际开发中也会遇到，例如**切线空间（tangent space）**。切线空间通常用于法线映射，在后面的 4.7 节中我们

会讲到。



▲图 4.45 渲染流水线中顶点的空间变换过程



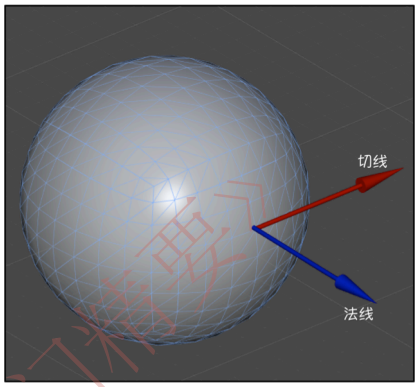
▲图 4.46 Unity 中各个坐标空间的旋向性

4.7 法线变换

在本章的最后，我们来看一种特殊的变换：法线变换。

法线（normal），也被称为**法矢量（normal vector）**。在上面我们已经看到如何使用变换矩阵来变换一个顶点或一个方向矢量，但法线是需要我们特殊处理的一种方向矢量。在游戏中，模型的一个顶点往往会携带额外的信息，而顶点法线就是其中一种信息。当我们变换一个模型的时候，不仅需要变换它的顶点，还需要变换顶点法线，以便在后续处理（如片元着色器）中计算光照等。

一般来说，点和绝大部分方向矢量都可以使用同一个 4×4 或 3×3 的变换矩阵 $\mathbf{M}_{A\to B}$ 把其从坐标空间 **A** 变换到坐标空间 **B** 中。但在变换法线的时候，如果使用同一个变换矩阵，可能就无法确保维持法线的垂直性。下面就来了解一下为什么会出现这样的问题。



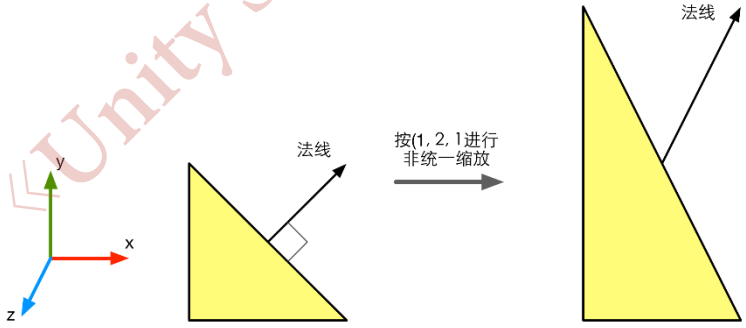
▲图 4.47　顶点的切线和法线。切线和法线互相垂直

我们先来了解一下另一种方向矢量——**切线（tangent）**，也被称为**切矢量（tangent vector）**。与法线类似，切线往往也是模型顶点携带的一种信息。它通常与纹理空间对齐，而且与法线方向垂直，如图 4.47 所示。

由于切线是由两个顶点之间的差值计算得到的，因此我们可以直接使用用于变换顶点的变换矩阵来变换切线。假设，我们使用 3×3 的变换矩阵 $\mathbf{M}_{A\to B}$ 来变换顶点（注意，这里涉及的变换矩阵都是 3×3 的矩阵，不考虑平移变换。这是因为切线和法线都是方向矢量，不会受平移的影响），可以由下面的式子直接得到变换后的切线：

$$\mathbf{T}_B = \mathbf{M}_{A\to B}\mathbf{T}_A$$

其中 $\mathbf{T}_A$ 和 $\mathbf{T}_B$ 分别表示在坐标空间 A 下和坐标空间 B 下的切线方向。但如果直接使用 $\mathbf{M}_{A\to B}$ 来变换法线，得到的新的法线方向可能就不会与表面垂直了。图 4.48 给出了这样的一个例子。



▲图 4.48　进行非统一缩放时，如果使用和变换顶点相同的变换矩阵来变换法线，就会得到错误的结果，即变换后的法线方向与平面不再垂直

那么，应该使用哪个矩阵来变换法线呢？我们可以由数学约束条件来推出这个矩阵。我们知道同一个顶点的切线 $\mathbf{T}_A$ 和法线 $\mathbf{N}_A$ 必须满足垂直条件，即 $\mathbf{T}_A \cdot \mathbf{N}_A = 0$。给定变换矩阵 $\mathbf{M}_{A\to B}$，我们已经知道 $\mathbf{T}_B = \mathbf{M}_{A\to B}\ \mathbf{T}_A$。我们现在想要找到一个矩阵 **G** 来变换法线 $\mathbf{N}_A$，使得变换后的法线仍然

与切线垂直。即

$$\mathbf{T}_B \cdot \mathbf{N}_B = (\mathbf{M}_{A\to B}\,\mathbf{T}_A) \cdot (\mathbf{G}\mathbf{N}_A) = 0$$

对上式进行一些推导后可得

$$(\mathbf{M}_{A\to B}\mathbf{T}_A)\cdot(\mathbf{G}\mathbf{N}_A) = (\mathbf{M}_{A\to B}\mathbf{T}_A)^T(\mathbf{G}\mathbf{N}_A) = \mathbf{T}_A^T\mathbf{M}_{A\to B}^T\mathbf{G}\mathbf{N}_A = \mathbf{T}_A^T(\mathbf{M}_{A\to B}^T\mathbf{G})\mathbf{N}_A = 0$$

由于 $\mathbf{T}_A \cdot \mathbf{N}_A = 0$，因此如果 $\mathbf{M}_{A\to B}^T\mathbf{G} = \mathbf{I}$，那么上式即可成立。也就是说，如果 $\mathbf{G} = (\mathbf{M}_{A\to B}^T)^{-1} = (\mathbf{M}_{A\to B}^{-1})^T$，即使用原变换矩阵的逆转置矩阵来变换法线就可以得到正确的结果。

值得注意的是，如果变换矩阵 $\mathbf{M}_{A\to B}$ 是正交矩阵，那么 $\mathbf{M}_{A\to B}^{-1} = \mathbf{M}_{A\to B}^T$，因此 $(\mathbf{M}_{A\to B}^T)^{-1} = \mathbf{M}_{A\to B}$，也就是说我们可以使用用于变换顶点的变换矩阵来直接变换法线。如果变换只包括旋转变换，那么这个变换矩阵就是正交矩阵。而如果变换只包含旋转和统一缩放，而不包含非统一缩放，我们利用统一缩放系数 k 来得到变换矩阵 $\mathbf{M}_{A\to B}$ 的逆转置矩阵 $(\mathbf{M}_{A\to B}^T)^{-1} = \frac{1}{k}\mathbf{M}_{A\to B}$。这样就可以避免计算逆矩阵的过程。如果变换中包含了非统一变换，那么我们就必须要求解逆矩阵来得到变换法线的矩阵。

4.8 Unity Shader 的内置变量（数学篇）

使用 Unity 写 Shader 的一个好处在于，它提供了很多内置的参数，这使得我们不再需要自己手动计算一些值。本节将给出 Unity 内置的用于空间变换和摄像机以及屏幕参数的内置变量。这些内置变量可以在 UnityShaderVariables.cginc 文件中找到定义和说明。

4.8.1 变换矩阵

首先是用于坐标空间变换的矩阵。表 4.2 给出了 Unity 5.2 版本提供的所有内置变换矩阵。下面所有的矩阵都是 float4x4 类型的。

读者：为什么在我的 Unity 中，有些变量不存在呢？

我们：可能是由于你使用的 Unity 版本和本书使用的版本不同。在写本书时，我们使用的 Unity 版本是最新的 5.2。而在 4.x 版本中，一些内置变量可能会与之不同。

表 4.2 Unity 内置的变换矩阵

变 量 名	描 述
UNITY_MATRIX_MVP	当前的模型·观察·投影矩阵，用于将顶点/方向矢量从模型空间变换到裁剪空间
UNITY_MATRIX_MV	当前的模型·观察矩阵，用于将顶点/方向矢量从模型空间变换到观察空间
UNITY_MATRIX_V	当前的观察矩阵，用于将顶点/方向矢量从世界空间变换到观察空间
UNITY_MATRIX_P	当前的投影矩阵，用于将顶点/方向矢量从观察空间变换到裁剪空间
UNITY_MATRIX_VP	当前的观察·投影矩阵，用于将顶点/方向矢量从世界空间变换到裁剪空间
UNITY_MATRIX_T_MV	UNITY_MATRIX_MV 的转置矩阵
UNITY_MATRIX_IT_MV	UNITY_MATRIX_MV 的逆转置矩阵，用于将法线从模型空间变换到观察空间，也可用于得到 UNITY_MATRIX_MV 的逆矩阵
_Object2World	当前的模型矩阵，用于将顶点/方向矢量从模型空间变换到世界空间
_World2Object	_Object2World 的逆矩阵，用于将顶点/方向矢量从世界空间变换到模型空间

表 4.2 给出了这些矩阵的常用用法。但读者可以根据需求来达到不同的目的，例如我们可以提取坐标空间的坐标轴，方法可回顾 4.6.2 节。

其中有一个矩阵比较特殊，即 UNITY_MATRIX_T_MV 矩阵。很多对数学不了解的读者不理

解这个矩阵有什么用处。如果读者认真看过矩阵一节的知识，应该还会记得一种非常吸引人的矩阵类型——正交矩阵。对于正交矩阵来说，它的逆矩阵就是转置矩阵。因此，如果 UNITY_MATRIX_MV 是一个正交矩阵的话，那么 UNITY_MATRIX_T_MV 就是它的逆矩阵，也就是说，我们可以使用 UNITY_MATRIX_T_MV 把顶点和方向矢量从观察空间变换到模型空间。那么问题是，UNITY_MATRIX_MV 什么时候是一个正交矩阵呢？读者可以从 4.5 节找到答案。总结一下，如果我们只考虑旋转、平移和缩放这三种变换的话，如果一个模型的变换只包括旋转，那么 UNITY_MATRIX_MV 就是一个正交矩阵。这个条件似乎有些苛刻，我们可以把条件再放宽一些，如果只包括旋转和统一缩放（假设缩放系数是 k），那么 UNITY_MATRIX_MV 就几乎是一个正交矩阵了。为什么是几乎呢？因为统一缩放可能会导致每一行（或每一列）的矢量长度不为 1，而是 k，这不符合正交矩阵的特性，但我们可以通过除以这个统一缩放系数，来把它变成正交矩阵。在这种情况下，UNITY_MATRIX_MV 的逆矩阵就是 $\frac{1}{k}$UNITY_MATRIX_T_MV。而且，如果我们只是对方向矢量进行变换的话，条件可以放得更宽，即不用考虑有没有平移变换，因为平移对方向矢量没有影响。因此，我们可以截取 UNITY_MATRIX_T_MV 的前三行前三列来把方向矢量从观察空间变换到模型空间（前提是只存在旋转变换和统一缩放）。对于方向矢量，我们可以在使用前对它们进行归一化处理，来消除统一缩放的影响。

还有一个矩阵需要说明一下，那就是 UNITY_MATRIX_IT_MV 矩阵。我们在 4.7 节已经知道，法线的变换需要使用原变换矩阵的逆转置矩阵。因此 UNITY_MATRIX_IT_MV 可以把法线从模型空间变换到观察空间。但只要我们做一点手脚，它也可以用于直接得到 UNITY_MATRIX_MV 的逆矩阵——我们只需要对它进行转置就可以了。因此，为了把顶点或方向矢量从观察空间变换到模型空间，我们可以使用类似下面的代码：

```
// 方法一：使用 transpose 函数对 UNITY_MATRIX_IT_MV 进行转置，
// 得到 UNITY_MATRIX_MV 的逆矩阵，然后进行列矩阵乘法，
// 把观察空间中的点或方向矢量变换到模型空间中
float4 modelPos = mul(transpose(UNITY_MATRIX_IT_MV), viewPos);

// 方法二：不直接使用转置函数 transpose，而是交换 mul 参数的位置，使用行矩阵乘法
// 本质和方法一是完全一样的
float4 modelPos = mul(viewPos, UNITY_MATRIX_IT_MV);
```

关于 mul 函数参数位置导致的不同，在 4.9.2 节中我们会继续讲到。

4.8.2 摄像机和屏幕参数

Unity 提供了一些内置变量来让我们访问当前正在渲染的摄像机的参数信息。这些参数对应了摄像机上的 Camera 组件中的属性值。表 4.3 给出了 Unity 5.2 版本提供的这些变量。

表 4.3 Unity 内置的摄像机和屏幕参数

变 量 名	类型	描 述
_WorldSpaceCameraPos	float3	该摄像机在世界空间中的位置
_ProjectionParams	float4	x = 1.0（或−1.0，如果正在使用一个翻转的投影矩阵进行渲染），y = Near，z = Far，w = 1.0 + 1.0/Far，其中 Near 和 Far 分别是近裁剪平面和远裁剪平面和摄像机的距离
_ScreenParams	float4	x = width，y = height，z = 1.0 + 1.0/width，w = 1.0 + 1.0/height，其中 width 和 height 分别是该摄像机的渲染目标（render target）的像素宽度和高度
_ZBufferParams	float4	x = 1− Far/Near，y = Far/Near，z = x/Far，w = y/Far，该变量用于线性化 Z 缓存中的深度值（可参考 13.1 节）

续表

变 量 名	类型	描 述
unity_OrthoParams	float4	x = width，y = heigth，z 没有定义，w = 1.0（该摄像机是正交摄像机）或 w = 0.0（该摄像机是透视摄像机），其中 width 和 height 是正交投影摄像机的宽度和高度
unity_CameraProjection	float4x4	该摄像机的投影矩阵
unity_CameraInvProjection	float4x4	该摄像机的投影矩阵的逆矩阵
unity_CameraWorldClipPlanes[6]	float4	该摄像机的 6 个裁剪平面在世界空间下的等式，按如下顺序：左、右、下、上、近、远裁剪平面。

4.9 答疑解惑

恭喜你已经几乎完成了本书所有的数学训练！我们希望你能从上面的内容中得到很多收获和启发。但是，我们也相信在读完上面的内容后你可能对某些概念仍然感到迷惑。不要担心，答疑解惑一节就可以帮你跨过一些障碍。

4.9.1 使用 3×3 还是 4×4 的变换矩阵

对于线性变换（例如旋转和缩放）来说，仅使用 3×3 的矩阵就足够表示所有的变换了。但如果存在平移变换，我们就需要使用 4×4 的矩阵。因此，在对顶点的变换中，我们通常使用 4×4 的变换矩阵。当然，在变换前我们需要把点坐标转换成齐次坐标的表示，即把顶点的 w 分量设为 1。而在对方向矢量的变换中，我们通常使用 3×3 的矩阵就足够了，这是因为平移变换对方向矢量是没有影响的。

4.9.2 Cg 中的矢量和矩阵类型

我们通常在 Unity Shader 中使用 Cg 作为着色器编程语言。在 Cg 中变量类型有很多种，但在本节我们是想解释如何使用这些类型进行数学运算。因此，我们只以 float 家族的变量来做说明。

在 Cg 中，矩阵类型是由 float3x3、float4x4 等关键词进行声明和定义的。而对于 float3、float4 等类型的变量，我们既可以把它当成一个矢量，也可以把它当成是一个 1 x n 的行矩阵或者一个 n x 1 的列矩阵。这取决于运算的种类和它们在运算中的位置。例如，当我们进行点积操作时，两个操作数就被当成矢量类型，如下：

```
float4 a = float4(1.0, 2.0, 3.0, 4.0);
float4 b = float4(1.0, 2.0, 3.0, 4.0);
// 对两个矢量进行点积操作
float result = dot(a, b);
```

但在进行矩阵乘法时，参数的位置将决定是按列矩阵还是行矩阵进行乘法。在 Cg 中，矩阵乘法是通过 mul 函数实现的。例如：

```
float4 v = float4(1.0, 2.0, 3.0, 4.0);
float4x4 M = float4x4(1.0, 0.0, 0.0, 0.0,
                      0.0, 2.0, 0.0, 0.0,
                      0.0, 0.0, 3.0, 0.0,
                      0.0, 0.0, 0.0, 4.0);
// 把 v 当成列矩阵和矩阵 M 进行右乘
float4 column_mul_result = mul(M, v);
// 把 v 当成行矩阵和矩阵 M 进行左乘
float4 row_mul_result = mul(v, M);
// 注意：column_mul_result 不等于 row_mul_result，而是：
// mul(M,v) == mul(v, tranpose(M))
// mul(v,M) == mul(tranpose(M), v)
```

因此，参数的位置会直接影响结果值。通常在变换顶点时，我们都是使用右乘的方式来按列矩阵进行乘法。这是因为，Unity 提供的内置矩阵（如 UNITY_MATRIX_MVP 等）都是按列存储的。但有时，我们也会使用左乘的方式，这是因为可以省去对矩阵转置的操作。

需要注意的一点是，Cg 对矩阵类型中元素的初始化和访问顺序。在 Cg 中，对 float4x4 等类型的变量是按行优先的方式进行填充的。什么意思呢？我们知道，想要填充一个矩阵需要给定一串数字，例如，如果需要声明一个 3×4 的矩阵，我们需要提供 12 个数字。那么，这串数字是一行一行地填充矩阵还是一列一列地填充矩阵呢？这两种方式得到的矩阵是不同的。例如，我们使用(1, 2, 3, 4, 5, 6, 7, 8, 9)去填充一个 3 x 3 的矩阵，如果是按照行优先的方式，得到的矩阵是：

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

如果是按照列优先的方式，得到的矩阵是：

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Cg 使用的是行优先的方法，即是一行一行地填充矩阵的。因此，如果读者需要自己定义一个矩阵时（例如，自己构建用于空间变换的矩阵），就要注意这里的初始化方式。

类似地，当我们在 Cg 中访问一个矩阵中的元素时，也是按行来索引的。例如：

```
// 按行优先的方式初始化矩阵 M
float3x3 M = float3x3(1.0, 2.0, 3.0,
                      4.0, 5.0, 6.0,
                      7.0, 8.0, 9.0);
// 得到 M 的第一行，即(1.0, 2.0, 3.0)
float3 row = M[0];

// 得到 M 的第 2 行第 1 列的元素，即 4.0
float ele = M[1][0];
```

之所以 Unity Shader 中的矩阵类型满足上述规则，是因为使用的是 Cg 语言。换句话说，上面的特性都是 Cg 的规定。

如果读者熟悉 Unity 的 API，可能知道 Unity 在脚本中提供了一种矩阵类型——Matrix4x4。脚本中的这个矩阵类型则是采用列优先的方式。这与 Unity Shader 中的规定不一样，希望读者在遇到时不会感到困惑。

4.9.3 Unity 中的屏幕坐标：ComputeScreenPos/VPOS/WPOS

我们在 4.6.8 节中讲了屏幕空间的转换细节。在写 Shader 的过程中，我们有时候希望能够获得片元在屏幕上的像素位置。

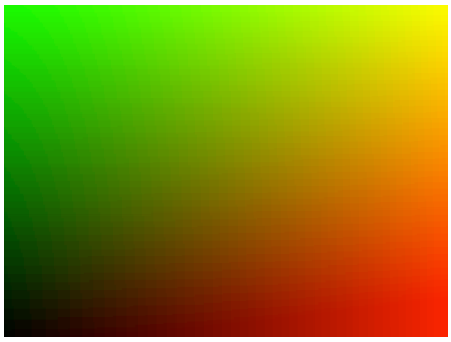
在顶点/片元着色器中，有两种方式来获得片元的屏幕坐标。

一种是在片元着色器的输入中声明 **VPOS** 或 **WPOS** 语义（关于什么是语义，可参见 5.4 节）。VPOS 是 HLSL 中对屏幕坐标的语义，而 WPOS 是 Cg 中对屏幕坐标的语义。两者在 Unity Shader 中是等价的。我们可以在 HLSL/Cg 中通过语义的方式来定义顶点/片元着色器的默认输入，而不需要自己定义输入输出的数据结构。这里的内容有一些超前，因为我们还没有具体讲解顶点/片元着色器的写法，读者在这里可以只关注 VPOS 和 WPOS 的语义。使用这种方法，可以在片元着色

器中这样写：

```
fixed4 frag(float4 sp : VPOS) : SV_Target {
    // 用屏幕坐标除以屏幕分辨率_ScreenParams.xy，得到视口空间中的坐标
    return fixed4(sp.xy/_ScreenParams.xy,0.0,1.0);
}
```

得到的效果如图 4.49 所示。



▲图 4.49 由片元的像素位置得到的图像

VPOS/WPOS 语义定义的输入是一个 float4 类型的变量。我们已经知道它的 *xy* 值代表了在屏幕空间中的像素坐标。如果屏幕分辨率为 400 x 300，那么 *x* 的范围就是[0.5, 400.5]，*y* 的范围是[0.5, 300.5]。注意，这里的像素坐标并不是整数值，这是因为 OpenGL 和 DirectX 10 以后的版本认为像素中心对应的是浮点值中的 0.5。那么，它的 *zw* 分量是什么呢？在 Unity 中，VPOS/WPOS 的 z 分量范围是[0,1]，在摄像机的近裁剪平面处，*z* 值为 0，在远裁剪平面处，*z* 值为 1。对于 *w* 分量，我们需要考虑摄像机的投影类型。如果使用的是透视投影，那么 *w* 分量的范围是 $\left[\frac{1}{Near},\frac{1}{Far}\right]$，Near 和 Far 对应了在 Camera 组件中设置的近裁剪平面和远裁剪平面距离摄像机的远近；如果使用的是正交投影，那么 *w* 分量的值恒为 1。这些值是通过对经过投影矩阵变换后的 *w* 分量取倒数后得到的。在代码的最后，我们把屏幕空间除以屏幕分辨率来得到**视口空间（viewport space）**中的坐标。视口坐标很简单，就是把屏幕坐标归一化，这样屏幕左下角就是(0, 0)，右上角就是(1, 1)。如果已知屏幕坐标的话，我们只需要把 *xy* 值除以屏幕分辨率即可。

另一种方式是通过 Unity 提供的 **ComputeScreenPos** 函数。这个函数在 UnityCG.cginc 里被定义。通常的用法需要两个步骤，首先在顶点着色器中将 ComputeScreenPos 的结果保存在输出结构体中，然后在片元着色器中进行一个齐次除法运算后得到视口空间下的坐标。例如：

```
struct vertOut {
    float4 pos : SV_POSITION;
    float4 scrPos : TEXCOORD0;
};

vertOut vert(appdata_base v){
    vertOut o;
    o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
    // 第一步：把 ComputeScreenPos 的结果保存到 scrPos 中
    o.scrPos = ComputeScreenPos(o.pos);
    return o;
}

fixed4 frag(vertOut i) : SV_Target {
    // 第二步：用 scrPos.xy 除以 scrPos.w 得到视口空间中的坐标
    float2 wcoord = (i.scrPos.xy/i.scrPos.w);
    return fixed4(wcoord,0.0,1.0);
}
```

上面代码的实现效果和图 4.49 中的一样。我们现在来看一下这种方式的实现细节。这种方法实际上是手动实现了屏幕映射的过程，而且它得到的坐标直接就是视口空间中的坐标。我们在 3.6.8 节中已经看到了如何将裁剪坐标空间中的点映射到屏幕坐标中。据此，我们可以得到视口空间中的坐标，公式如下：

$$viewport_x = \frac{clip_x}{2 \cdot clip_w} + \frac{1}{2}$$

$$viewport_y = \frac{clip_y}{2 \cdot clip_w} + \frac{1}{2}$$

上面公式的思想就是，首先对裁剪空间下的坐标进行齐次除法，得到范围在[-1, 1]的 NDC，然后再将其映射到范围在[0, 1]的视口空间下的坐标。那么 ComputeScreenPos 究竟是如何做到的呢？我们可以在 UnityCG.cginc 文件中找到 ComputeScreenPos 函数的定义。如下：

```
inline float4 ComputeScreenPos (float4 pos) {
    float4 o = pos * 0.5f;
    #if defined(UNITY_HALF_TEXEL_OFFSET)
    o.xy = float2(o.x, o.y*_ProjectionParams.x) + o.w * _ScreenParams.zw;
    #else
    o.xy = float2(o.x, o.y*_ProjectionParams.x) + o.w;
    #endif

    o.zw = pos.zw;
    return o;
}
```

ComputeScreenPos 的输入参数 pos 是经过 MVP 矩阵变换后在裁剪空间中的顶点坐标。UNITY_HALF_TEXEL_OFFSET 是 Unity 在某些 DirectX 平台上使用的宏，在这里我们可以忽略它。这样，我们可以只关注#else 的部分。_ProjectionParams.x 在默认情况下是 1（如果我们使用了一个翻转的投影矩阵的话就是−1，但这种情况很少见）。那么上述代码的过程实际是输出了：

$$Output_x = \frac{clip_x}{2} + \frac{clip_w}{2}$$

$$Output_y = \frac{clip_y}{2} + \frac{clip_w}{2}$$

$$Output_z = clip_z$$

$$Output_w = clip_w$$

可以看出，这里的 *xy* 并不是真正的视口空间下的坐标。因此，我们在片元着色器中再进行一步处理，即除以裁剪坐标的 *w* 分量。至此，完成整个映射的过程。因此，虽然 ComputeScreenPos 的函数名字似乎意味着会直接得到屏幕空间中的位置，但并不是这样的，我们仍需在片元着色器中除以它的 *w* 分量来得到真正的视口空间中的位置。那么，为什么 Unity 不直接在 ComputeScreenPos 中为我们进行除以 *w* 分量的这个步骤呢？为什么还需要我们来进行这个除法？这是因为，如果 Unity 在顶点着色器中这么做的话，就会破坏插值的结果。我们知道，从顶点着色器到片元着色器的过程实际会有一个插值的过程（如果你忘了的话，可以回顾 2.3.6 小节）。如果不在顶点着色器中进行这个除法，保留 *x*、*y* 和 *w* 分量，那么它们在插值后再进行这个除法，得到的 $\frac{x}{w}$ 和 $\frac{y}{w}$ 就是正确的（我们可以认为是除法抵消了插值的影响）。但如果我们直接在顶点着色器中进行这个除法，那么就需要对 $\frac{x}{w}$ 和 $\frac{y}{w}$ 直接进行插值，这样得到的插值结果就会不准确。原因是，我们不可以在投影空间中进行插值，因为这并不是一个线性空间，而插值往往是线性的。

经过除法操作后，我们就可以得到该片元在视口空间中的坐标了，也就是一个 *xy* 范围都在[0, 1]之间的值。那么它的 *zw* 值是什么呢？可以看出，我们在顶点着色器中直接把裁剪空间的 *zw* 值存进了输出结构体中，因此片元着色器输入的就是这些插值后的裁剪空间中的 *zw* 值。这意味着，

如果使用的是透视投影，那么 z 值的范围是[-Near, Far]，w 值的范围是[Near, Far]；如果使用的是正交投影，那么 z 值范围是[−1, 1]，而 w 值恒为 1。

4.10 扩展阅读

计算机图形学使用的数学还有很多， 本书仅涵盖了其中非常小的一部分。如果读者想要深入学习这些知识的话，书籍[1][2]是非常好的图形学数学学习资料，读者可以在那里找到更多类型的变换及其数学表示。关于如何从左手坐标系转换到右手坐标系同时又保持视觉效果一样，可以参考资料[3]。关于如何得到线性的深度值可以参考资料[4]。

[1] Fletcher Dunn, Ian Parberry. 3D Math Primer for Graphics and Game Development (2nd Edition). November 2, 2011 by A K Peters/CRC Press

[2] Eric Lengyel. Mathematics for 3D game programming and computer graphics (3rd Edition). 2011 by Charles River Media.

[3] David Eberly. Conversion of Left-Handed Coordinates to Right-Handed Coordinates

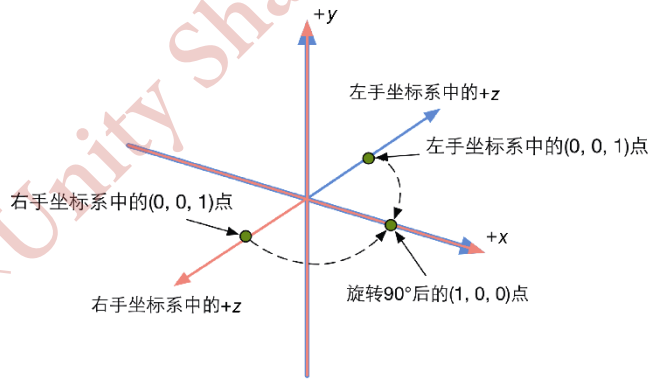
[4] http://www.humus.name/temp/Linearize%20depth.txt

4.11 练习题答案

4.2.5 节

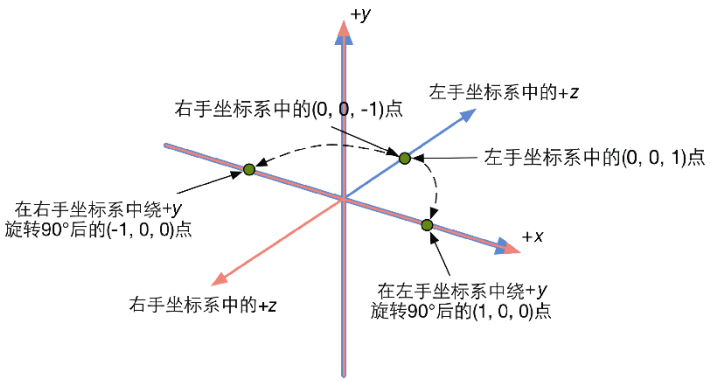
1．右手坐标系。

2．(1, 0, 0)。(1, 0, 0)。从坐标表示来看，结果是完全一样的。左手坐标系和右手坐标系在绝大多数情况下不会对底层的数学运算造成影响，但是会在视觉表现上有所差异。以本题为例，虽然旋转之前点的坐标是一样的，但如果把它们统一在同一个空间中显示出来，其绝对位置是不同的。如图 4.50 所示。



▲图 4.50　图中两个坐标系的 x 轴和 y 轴是重合的，区别仅在于 z 轴的方向。左手坐标系的（0, 0, 1）点和右手坐标系中的（0, 0, 1）点是不同的，但它们旋转后的点却对应到了同一点

因此，如果我们想要在左手和右手坐标系中表示同一个点，就需要把其中一个坐标系中的表示方法中的某个轴反向，一般是把 z 值取反。在本例中，左手坐标系的(0, 0, 1)点和右手坐标系中的(0, 0, −1)点是同一点。但是，如果此时对该点再次分别在左手和右手坐标系中绕 y 轴正方向旋转 90°，结果就不是同一个点了，如图 4.51 所示。



▲图 4.51　绝对空间中的同一点，在左手和右手坐标系中进行同样角度的旋转，其旋转方向是不一样的。在左手坐标系中将按顺时针方向旋转，在右手坐标系中将按逆时针方向旋转。

3．−10。10。这是因为，在 Unity 中，模型空间使用的是左手坐标系。球体所在的位置位于摄像机模型空间中的 z 轴正方向，因此在模型空间下其 z 值为 10。而观察空间使用的右手坐标系，摄像机的正前方是 z 轴的负方向，因此在观察空间下其 z 值为−10。

4.3.3 节

1．

（1）错误，完全说反了。对于矢量来说它有两个属性：模（即大小）和方向，矢量是没有位置属性的，也就是说，我们可以随意把它放在空间的任何位置。

（2）正确。

（3）错误。坐标系的选择不会对底层的数学计算产生影响，对于叉积来说，我们总可以使用公式

$$\begin{aligned}\mathbf{a}\times\mathbf{b} &= (a_x, a_y, a_z)\times(b_x, b_y, b_z)\\ &= (a_yb_z - a_zb_y, a_zb_x - a_xb_z, a_xb_y - a_yb_x)\end{aligned}$$

来计算。但是，不同的坐标系会影响最后的显示结果，即视觉上的表现。数学是一门非常严谨的学科，但人类往往需要可视化一些东西，例如在屏幕上显示虚拟的三维空间，在把数字转换成视觉表现的时候，选择不同的坐标系可能会得到不同的结果。

2．

（1）$\sqrt{62} \approx 7.874$

（2）(12.5, 10, 25)

（3）(1.5, 2)

（4）$\left(\frac{5}{13}, \frac{12}{13}\right) \approx (0.385, 0.923)$

（5）$\left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right) \approx (0.577, 0.577, 0.577)$

（6）(10, 9)

（7）(−6, 1, 2)

3．$\sqrt{308} \approx 17.55$

4.

（1）75

（2）13

（3）13

（4）(−9, −13, 7)

（5）(9, 13, −7)，注意，结果和答案（4）是相反的。这是因为，叉积满足反交换律。

5.

（1）12

（2）$12\sqrt{3} \approx 20.785$

6.

（1）我们可以通过判断 **x** - **p** 和 **v** 点积的符号来判断 **x** 是否在 NPC 的前方。这是因为：

$$(\mathbf{x}-\mathbf{p})\cdot\mathbf{v}=|\mathbf{v}||\mathbf{x}-\mathbf{p}|\cos\theta$$

其中θ是 **x** - **p** 和 **v** 之间的夹角。如果它们点积的结果大于 0，那么说明$\theta < 90°$，即点 **x** 在 NPC 的前方；如果点积结果小于 0，那么说明$\theta > 90°$，即点 **x** 在 NPC 的后方；如果点积结果等于 0，那么说明$\theta = 90°$，即点 **x** 在 NPC 的正左侧或正右侧。

（2）代入得

$$(\mathbf{x}-\mathbf{p})\cdot\mathbf{v}=((10,6)-(4,2))\cdot(-3,4)=(6,4)\cdot(-3,4)=-18+16=-2<0$$

因此，点 **x** 在 NPC 的后方。

（3）我们现在需要判断 $\cos\theta$和 $\cos\frac{\phi}{2}$的大小。如果 $\cos\theta > \cos\frac{\phi}{2}$，那么说明 $\theta < \frac{\phi}{2}$，即 NPC 可以看到该点；如果 $\cos\theta < \cos\frac{\phi}{2}$，那么说明 $\theta > \frac{\phi}{2}$，即 NPC 无法看到该点。$\cos\theta$可以由 $\cos\theta=\frac{(\mathbf{x}-\mathbf{p})\cdot\mathbf{v}}{|\mathbf{x}-\mathbf{p}||\mathbf{v}|}$来得到，而 $\cos\frac{\phi}{2}$可直接计算得到。

（4）如果有距离限制，我们只需要判断该点到 **p** 的距离是否小于该限制值即可。

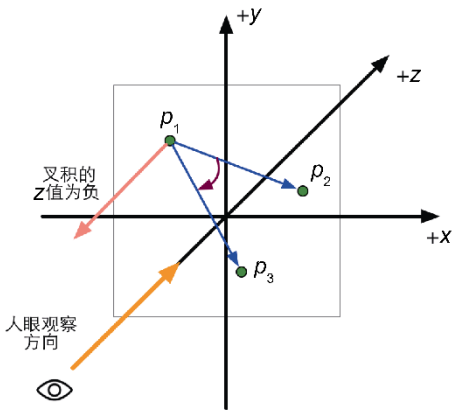
7．令 $\mathbf{u}=\mathbf{p}_2-\mathbf{p}_1$，$\mathbf{v}=\mathbf{p}_3-\mathbf{p}_1$。由于三点都位于 xy 平面，那么有：

$$\mathbf{u}=(u_x, u_y, 0), \mathbf{v}=(v_x, v_y, 0)$$

它们的叉积为：

$$\mathbf{u}\times\mathbf{v}=(0, 0, u_xv_y-u_yv_x)$$

我们可以通过判断 $u_xv_y-u_yv_x$的符号来判断三角形的朝向。如果该值为负，则由左手法则判断可得到 3 个顶点的顺序是顺时针方向；如果为正，则为逆时针方向。如图 4.52 所示。



▲图 4.52　在左手坐标系中，如果叉积结果为负，那么 3 点的顺序是顺时针方向

4.4.6　小节

1.

（1）$\begin{bmatrix}1 & 3\\ 2 & 4\end{bmatrix}\begin{bmatrix}-1 & 5\\ 0 & 2\end{bmatrix}=\begin{bmatrix}(1)(-1)+(3)(0) & (1)(5)+(3)(2)\\ (2)(-1)+(4)(0) & (2)(5)+(4)(2)\end{bmatrix}=\begin{bmatrix}-1 & 11\\ -2 & 18\end{bmatrix}$

（2）无法进行矩阵乘法，两个矩阵相乘要求第一个矩阵的列数等于第二个的行数，因此我们无法对 2×3 和 4×2 的矩阵进行乘法。

（3）$\begin{bmatrix}1 & -2 & 3\\ 5 & 1 & 4\\ 6 & 0 & 3\end{bmatrix}\begin{bmatrix}-5\\ 4\\ 8\end{bmatrix}=\begin{bmatrix}(1)(-5)+(-2)(4)+(3)(8)\\ (5)(-5)+(1)(4)+(4)(8)\\ (6)(-5)+(0)(4)+(3)(8)\end{bmatrix}=\begin{bmatrix}11\\ 11\\ -6\end{bmatrix}$

2.

（1）不是正交矩阵。它的转置矩阵和本身相乘的结果不是单位矩阵。也可以通过验证矩阵的行是否构成一组标准正交基来判断。

（2）是正交矩阵。

（3）是正交矩阵。这实际上是一个绕 z 轴旋转 $\theta°$ 的旋转矩阵。

3.

（1）$[3\quad 2\quad 6]\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}=\begin{bmatrix}(3)(1)+(2)(0)+(6)(0)\\ (3)(0)+(2)(1)+(6)(0)\\ (3)(0)+(2)(0)+(6)(1)\end{bmatrix}=[3\quad 2\quad 6]$

$\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}3\\ 2\\ 6\end{bmatrix}=\begin{bmatrix}(1)(3)+(0)(2)+(0)(6)\\ (0)(3)+(1)(2)+(0)(6)\\ (3)(0)(3)+(0)(2)+(1)(6)\end{bmatrix}=\begin{bmatrix}3\\ 2\\ 6\end{bmatrix}$

得到的结果转换成矢量都是(3, 2, 6)，是一样的。这是因为，该矩阵是一个单位矩阵，单位矩阵和任何矩阵相乘都是原矩阵本身。

（2）

$$[3\quad 2\quad 6]\begin{bmatrix}1 & 0 & 2\\ 0 & 1 & -3\\ 0 & 0 & 3\end{bmatrix}=\begin{bmatrix}(3)(1)+(2)(0)+(6)(0)\\ (3)(0)+(2)(1)+(6)(0)\\ (3)(2)+(2)(-3)+(6)(3)\end{bmatrix}=[3\quad 2\quad 18]$$

$$\begin{bmatrix}1 & 0 & 2\\ 0 & 1 & -3\\ 0 & 0 & 3\end{bmatrix}\begin{bmatrix}3\\ 2\\ 6\end{bmatrix}=\begin{bmatrix}(1)(3)+(0)(2)+(2)(6)\\ (0)(3)+(1)(2)+(-3)(6)\\ (0)(3)+(0)(2)+(3)(6)\end{bmatrix}=\begin{bmatrix}15\\ -16\\ 18\end{bmatrix}$$

得到的结果不一致。为了得到一致的结果，我们可以对矩阵进行转置。例如，为了得到和列矩阵相同的结果，在进行行矩阵乘法时，对矩阵进行转置，得

$$[3\quad 2\quad 6]\begin{bmatrix}1 & 0 & 2\\ 0 & 1 & -3\\ 0 & 0 & 3\end{bmatrix}^{T}=[3\quad 2\quad 6]\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 2 & -3 & 3\end{bmatrix}$$

$$=\begin{bmatrix}(3)(1)+(2)(0)+(6)(2)\\ (3)(0)+(2)(1)+(6)(-3)\\ (3)(0)+(2)(0)+(6)(3)\end{bmatrix}=\begin{bmatrix}15\quad -16\quad 18\end{bmatrix}$$

（3）

$$[3\quad 2\quad 6]\begin{bmatrix}2 & -1 & 3\\ -1 & 5 & -3\\ 3 & -3 & 4\end{bmatrix}=\begin{bmatrix}(3)(2)+(2)(-1)+(6)(3)\\ (3)(-1)+(2)(5)+(6)(-3)\\ (3)(3)+(2)(-3)+(6)(4)\end{bmatrix}=[22\quad -11\quad 27]$$

$$\begin{bmatrix}2 & -1 & 3\\ -1 & 5 & -3\\ 3 & -3 & 4\end{bmatrix}\begin{bmatrix}3\\ 2\\ 6\end{bmatrix}=\begin{bmatrix}(2)(3)+(-1)(2)+(3)(6)\\ (-1)(3)+(5)(2)+(-3)(6)\\ (3)(3)+(-3)(2)+(4)(6)\end{bmatrix}=\begin{bmatrix}22\\ -11\\ 27\end{bmatrix}$$

得到的结果转换成矢量都是(22, −11, 27)，是一样的。这是因为，该矩阵是一个**对称矩阵（symmetric matrix）**。对称矩阵的转置是其本身，因此行矩阵和列矩阵不会对结果产生影响。