第三章 虚拟世界的几何模型

2.2 节介绍了虚拟世界生成器（VWG），它维护着虚拟世界的几何和物理规律。本章涵盖几何部分，关于制作模型与按需求改换。这些模型可能包括建筑物的墙壁，家具，天空中的云彩，用户的虚拟形象等等。第 3.1 节介绍了关于如何定义一致且有用的模型的基础知识。第 3.2 节解释了如何应用能够在虚拟世界移动它们的数学变换。这涉及两个组成部分：转换（更改位置）和旋转（改变方向）。第 3.3 节介绍了最佳的表达并操纵 3D 旋转的方式，这是移动模型中最复杂的部分。第 3.4 节将介绍如果我们试图从特定的角度“看”虚拟世界，它将如何呈现。这是视觉渲染中的几何部分，在第 7 章中将会介绍。最后，第 3.5 节将所有的变换结合在一起，以便您可以看到如何从定义模型转到让它出现在显示屏的正确位置。

如果你使用高级引擎来构建 VR 体验，那么大多情况下本章中的概念似乎不是必需的。你可能只需要从菜单中选择选项并编写简单的脚本。然而，理解基本的转换，例如如何表达 3D 旋转或移动相机视角，对于使用软件做你想做的事情还是至关重要的。此外，如果你想从头开始构建虚拟世界，或者至少想要了解在软件引擎的背后发生了什么，那么本章至关重要。

3.1 几何模型

我们首先需要一个虚拟世界来包含几何模型。为此，一个具有笛卡尔坐标的 3D 欧几里德空间就足够了。因此，让 $\mathbb{R}^3$ 表示虚拟世界，其中每个点都表示为一个三元组实值坐标：（x，y，z）。我们的虚拟世界的坐标轴如图 3.1 所示。我们在此书中将始终使用右手坐标系，因为它们代表了整个物理学和工程学的主要选择；然而，左手系统也在一些地方出现，最值得注意的是微软的 DirectX 图形渲染库。在这些情况下，与右手系统中的方向相比，三个轴中的一个指向相反的方向。这种不一致可能导致在编写软件时几个小时的疯狂；因此，要了解它们之间的差异和要求如果您混合使用两种软件或模型。如果可能，避免混合使用右手和左手系统。

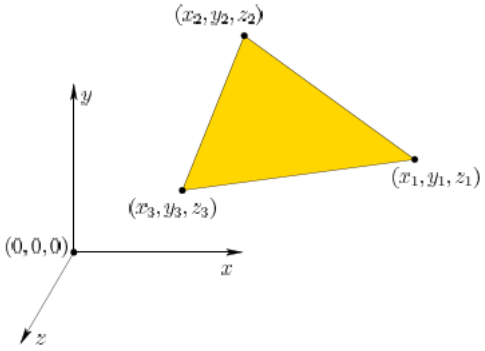

图 3.1：虚拟世界中的点在 y 轴向上的右手坐标系中给出坐标。原点（0，0，0）位于轴相交的点上。还显示了一个由其三个顶点而决定的 3D 三角形，每个顶点都是 $\mathbb{R}^3$ 中的一个点。

几何模型由 $\mathbb{R}^3$ 中的曲面或实心区域组成，并包含无限多点。由于计算机中的表示是有限的，模型是根据基元来定义的，其中每个基元表示一组无限点。最简单和最有用的基元是一个 3D 三角形，如图 3.1。对应于“内部”所有点和三角形边界上的平面由三角形顶点的坐标完全指定：

$$((x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3)). \tag{3.1}$$

为了模拟虚拟世界中的复杂物体，我们可以将大量的三角形排列成一个网格，如图 3.2

所示。这引发了很多重要的问题：

1. 我们如何确定，VR 用户查看每个三角形时，它们是什么样子的？
2. 我们如何让对象“移动”？
3. 如果物体表面是尖锐弯曲的，那么我们应该使用弯曲的基元而不是试图用微小的三角形拟合弯曲的物体吗？
4. 物体的内部是模型的一部分吗，还是只有它的表面？
5. 是否有有效的算法来确定哪些三角形与沿面指定的三角形相邻？
6. 我们应该避免重复许多相邻三角形共有的顶点坐标吗？

我们以相反的顺序来解决这些问题。

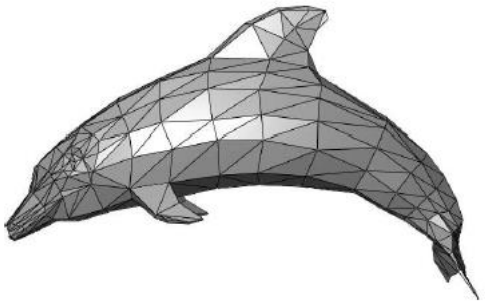

图 3.2：一只海豚的几何模型，由 3D 三角形的网格组成。（来自维基百科用户 Chrschn）

数据结构

考虑列出文件或存储数组中的所有三角形。如果三角形形成一个网格，那么在多个三角形中，大部分或全部顶点将会被共享。这显然是浪费空间。另一个问题是我们会经常想要在模型上执行操作。例如，当移动一个物体后，我们能确定它是否会与另一个对象发生碰撞吗（在 8.3 节会提及）？一个典型的低级任务可能是确定哪些三角形与给定的三角形共享一个共同的顶点或边。这可能需要在三角形列表中线性搜索以确定它们是共享一个顶点还是两个。如果有数以百万计的三角形，这并不罕见，那么它会因为重复执行此操作而花费许多。

由于以上及更多原因，几何模型通常利用精炼的数据结构编码。数据结构的选择应该取决于将在模型上执行的操作。其中最有用和最常见的是双连接边列表，也称为半边数据结构[2,10]。如图3.3。在这个和与其类似的数据结构中，有三种数据元素，面、边和定点。它们分别在模型中代表着二维、一维和零维部分。在我们的例子中，每个面元素代表一个三角形。每条边代表无重复的一个或两个三角形的边界。每个顶点都是一个或多个三角形之间共享的，同样没有重复。数据结构包含相邻面，边，和顶点之间的指针，以便算法可以按照与它们如何连接在一起的方式快速遍历模型组件。

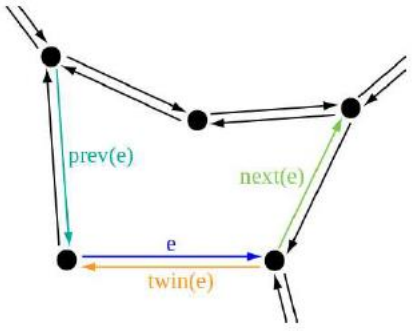

图 3.3：这里显示了一个五条边界表面的双连接边列表的一部分。每个半边结构 e 存储着沿着表面边界指向前一条和下一条边的指针。它还存储一个指向它自己的孪生半边结构的指针，这是相邻面边界的一部

分。（图来自维基百科用户 Accountalive）。

内部 vs 外部

现在考虑物体的内部是否属于模型的一部分（回忆图 3.2）。假设网格中三角形完美匹配，因此每条边都恰好与两个三角形相交，并且除非三角形沿表面相邻，否则它们不会相交。在这种情况下，模型会形成一个完整的在物体内部与外部之间的屏障。如果我们假设用气体填充内部，并且它们不会泄漏到外部。这是一个连贯模型的例子。如果内部或外部的概念对 VWG 至关重要，则需要使用此类模型。例如，一分钱可能在海豚的内部，但是它不与任何边界三角形相交。它始终需要被检测吗？如果我们移除一个三角形，那么假想的气体就会泄漏。物体的内外将不再有明显的区别，它使得这个关于硬币和海豚的问题难以回答。在极端情况下，在空间中可以有一个三角形。它显然没有天然的内部或外部。在极端情况下，模型可能会像*多边形汤*一样糟糕，它们是一堆不能很好匹配的，并且甚至有内部相交的三角形。总之，在建模时要多加小心，以便你稍后要执行的操作在逻辑上保持清晰。如果你要使用高级设计工具，比如 Blender 或者 Maya 来建模，那么连贯模型将会被自动建立。

为什么是三角形

通过上述问题，继续向前思考。使用三角形是因为它们对于算法来说是最方便处理的，特别在硬件实施方面。GPU 实现倾向于偏向于较小的表示，以便可以并行地将多个指令的紧凑列表应用于多个模型部分。当然可以使用更复杂的原型，如四边形，样条线和半代数曲面[3,4,9]。这可能导致更小的模型尺寸，但通常会因处理这样的原型而带来更大的计算开销。例如，两个样条曲面很难确定是否碰撞，相比于两个 3D 三角形来说。

固定模型 vs 可移动模型

以后再虚拟世界中将会有两种模式，它们被嵌入在 R^3 中：

•固定模型，它们永远保持相同的坐标。典型的例子包括街道，地板和建筑物。

•可移动模型，它们可以转换为多种位置和方向。例子包括车辆，虚拟人像和小家具。运动可以通过多种方式引起。使用跟踪系统（第 9 章），模型可能会移动以匹配用户的动作。或者，用户可能操作控制器来移动虚拟世界中的对象，包括自己的形象。最后，物体可以根据虚拟世界中的物理规律自行移动。第 3.2 节将涵盖将模型移动到他们想要的位置的数学运算，第 8 章将描述速度，加速度以及其他关于运动的物理概念。

选择坐标轴

一个经常被忽略的问题就是根据位置和规模，为模型选择坐标轴。如果这些在一开始就被很好地定义了，那么可以避免许多繁琐。如果虚拟世界与真实环境中熟悉的环境相对应，那么轴的缩放应该与常见单位相匹配。例如，（1,0,0）应该对应（0，0，0）右边一米。放置原点（0，0，0）在一个方便的位置也是明智的。通常，y = 0 对应建筑物的底层或者海平面。 x = 0 和 z = 0 的位置可能是虚拟世界的中心，以便根据符号能很好地划分象限。另一个常见的选择是当从上面观看世界时，将其放在左上角，使得所有 x 和 z 坐标都是非负的。对于可移动模型，原点的位置和轴的方向非常重要，因为它们会影响模型的旋转。这部分在 3.2 和 3.3 节介绍旋转后应该会变得更清楚。

查看模型

当然，VR 最重要的方面之一是当在显示器上观看时，模型看上去如何。 这个问题分为两部分。 第一部分涉及确定虚拟世界中的点应该出现在显示器上的什么位置。 这是通过查看第 3.4 以及第 3.5 节的其他转换产生的最终结果完成的。第二部分涉及考虑虚拟世界中定义的光源和表面介质之后模型中每个部分应该如何呈现。这是渲染问题，这第 7 章中会有介绍。

3.2 改变位置和方向

假设可移动模型已被定义为三角形网格。为了移动它，我们对每个三角形的每个顶点

应用单一变换。本节首先考虑变换的简单情况，其次是相当复杂的旋转情况。通过结合变换和旋转，模型可以放置在任何地方，以及虚拟世界 $\mathbb{R}^3$ 中的任何方向。

变换

考虑以下 3D 三角形，

$$((x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3)), \tag{3.2}$$

顶点坐标表示为通用常量。

设 xt，yt 和 zt 是分别沿 x，y 和 z 轴的我们想要改变的三角形位置量。 改变位置的操作称为变换，由下式给出

$$\begin{aligned}(x_1, y_1, z_1) &\rightarrow (x_1 + x_t, y_1 + y_t, z_1 + z_t) \\ (x_2, y_2, z_2) &\rightarrow (x_2 + x_t, y_2 + y_t, z_2 + z_t) \\ (x_3, y_3, z_3) &\rightarrow (x_3 + x_t, y_3 + y_t, z_3 + z_t)\end{aligned} \tag{3.3}$$

其中 a→b 表示在变换之后 a 被 b 替换。将（3.3）应用于模型中的每个三角形将会将其全部转换至所需的位置。如果三角形排列成网格，那将变换单独应用于顶点就足够了。 所有的三角形将保留它们原有的大小和形状。

相关性

在变换变得太复杂之前，我们想要提醒你要能正确解释它们。图 3.4（a）和 3.4（b）展示了一个例子：一个三角形以 $x_t = -8$ 和 $y_t = -7$ 变换。该顶点坐标与在图 3.4（b）和 3.4（c）中是相同的。 图 3.4（b）显示了我们到目前为止打算覆盖的情况：三角形被解释为在虚拟世界中发生移动。但是，图 3.4（c）显示了另一种可能：虚拟世界的坐标已被重新分配，三角形更加接近原点。这相当于让整个世界移动了，而三角形是唯一不移动的部分。在这种情况下，变换应用于坐标轴，但它们是负的。当我们应用更多一般变换时，它扩展了，因此变换坐标轴会导致相应地移动模型的变换的逆。否定就是在变换情况下的逆。

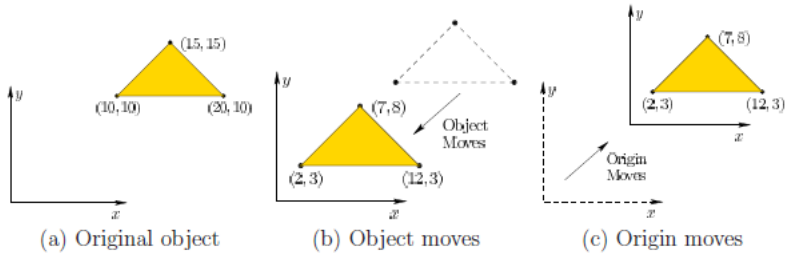

图 3.4：尽管数学上是一样的，每个转换还是有两种可能的解释。这是一个 2D 的例子，（a）中定义了一个三角形。我们可以通过 $x_t = -8$ 和 $y_t = -7$ 来变换这个三角形以获得（b）中的结果。如果我们想要固定三角形，但将原点在 x 方向上往前移动 8，在 y 方向上往前移动 7，那么三角形顶点的坐标改变方式完全相同，如（c）所示。

因此，我们有一种“相对性”：物体是移动的还是它周围的世界在移动？这个想法在 3.4 节会变得很重要当我们想改变视角。如果我们站在原点看三角形，那么结果在任何一种情况下看起来都是相同的;但是，如果原点移动，那么我们也会随之移动。一个深刻的感知问题在此出现。如果我们认为自己已经移动了，那么 VR 的缺陷可能会增加即使它是移动的对象。换句话说，我们的大脑会最佳地猜测哪种类型的动作发生，但有时还是会出错。

为旋转作准备

我们如何让汽车上的车轮滚动？或者把桌子翻一个面？为了完成这些，我们需要改变模型在虚拟世界中的定位。这个改变定位的操作改称为旋转。不幸的是，三维旋转比变换复杂许多，这给工程师和开发人员带来无数挫折。为了让 3D 旋转概念更为清晰，我们首先从一个更简单的问题开始：二维线性变换。

考虑一个 2D 虚拟世界，其中的点坐标（x，y）。 你可以把它想象成我们最初的 3D 虚拟世界中的一个垂直平面。 现在考虑一个通用的 2x2 矩阵

$$M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \tag{3.4}$$

其中四项的每一个可以是任何实数。我们看看当这个矩阵乘以点（x，y）后会发生什么，当它被写为列向量时。

执行乘法，我们有

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}, \tag{3.5}$$

其中（x’，y’）是变换点。使用简单的代数，矩阵乘法产生

$$\begin{aligned} x' &= m_{11}x + m_{12}y \\ y' &= m_{21}x + m_{22}y. \end{aligned} \tag{3.6}$$

使用（3.3）中的符号，M 是（x，y）→（x’，y’）的变换。

将二维矩阵应用于点

假设我们放置两个点（1,0）和（0,1）在平面上。它们分别位于 x 和 y 轴上，距离原点的距离（0，0）一个单位。使用向量空间，这两点就是标准单位矢量（有时写为 i 和 j）。如果我们将它们替换成（3.5），看看会发生什么：

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} m_{11} \\ m_{21} \end{bmatrix} \tag{3.7}$$

和

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{12} \\ m_{22} \end{bmatrix}. \tag{3.8}$$

这些特殊点只选择 M 上的列向量。这意味着什么？如果 M 应用于模型转换，则 M 的每一列确切地表示每个坐标轴是如何改变的。

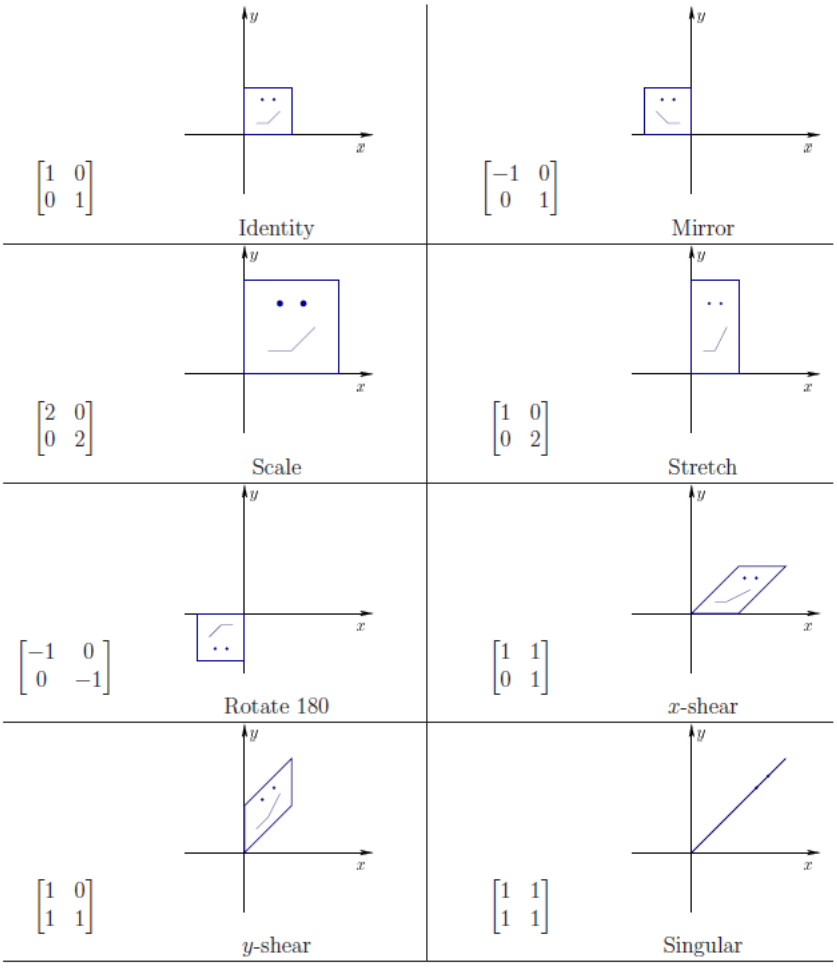

图 3.5：八个应用于变换方形面的不同矩阵。这些例子很好地从定性的角度涵盖了所有可能的情况。

图 3.5 说明了将各种矩阵 M 应用于模型的效果。从右上角开始，单位矩阵不会导致坐标改变：（x，y）→（x，y）。第二个例子会导致翻转，就像镜子被放置在 y 轴上一样。在这种情况下，（x，y）→（-x，y）。第二行是缩放的例子。左边的矩阵产生（x，y）→（2x，2y），其中尺寸加倍了。右边的矩阵只在 y 方向延伸模型，导致纵横比失真。在第三行中，似乎左边的矩阵产生了关于 x 和 y 的镜像轴。这是事实，除了镜像的镜像恢复了原始图像。因此，这对应于 180 度（π 弧度）旋转的情况，而不是镜像。右边的矩阵产生沿 x 方向的剪切：（x，y）→（x + y，y）。位移量与 y 成比例。

在最下面一行中，左边的矩阵显示在 y 方向的偏斜。最终的矩阵起初可能会引起更多的偏差，但它是退化的。当应用 M 时，二维形状会折叠成单一的维度：（x，y）→（x + y，x + y）。 这对应于奇异矩阵的情况，意味着它的列不是线性独立的（它们实际上是相同的）。一个矩阵当且仅当它的行列式为零时才是奇异的。

只有一些矩阵产生旋转

图 3.5 中的例子涵盖了各种 2×2 矩阵 M 之间的主要定性差异。其中两个是旋转矩阵：0 度旋转，和 180 度旋转的单位矩阵。在所有可能的 M 集合中，哪些可能是有效的旋转？我们必须确保模型不会被扭曲。这是通过确保 M 满足以下规则来实现的：

1. 没有伸展的轴。

2. 没有剪切。

3. 没有镜像。

如果没有违反这些规则，那么结果就是旋转。

为了满足第一条规则，M 的列必须有单位长度：

$$m_{11}^2 + m_{21}^2 = 1 \text{ and } m_{12}^2 + m_{22}^2 = 1. \tag{3.9}$$

图 3.5 中的缩放和剪切变换违反了这一点。

为了满足第二条规则，坐标轴必须保持垂直。否则，会发生剪切。由于 M 的列表示轴是如何变换的，那么此规则意味着它们的内部（点）积为零：

$$m_{11}m_{12} + m_{21}m_{22} = 0. \tag{3.10}$$

图 3.5 中的剪切变换违反了这个规则，这明显导致模型中的直角被破坏。

满足第三条规则要求 M 的行列式是正数。当满足前两条规则后，唯一可能的行列式是 1（正常情况下）和-1（镜像情况）。 因此，该规则意味着：

$$\det \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} = m_{11}m_{22} - m_{12}m_{21} = 1. \tag{3.11}$$

图 3.5 中的镜像示例导致 det M = -1。

第一个约束（3.9）表示每列必须都被选择以使它的组件位于一个以原点为中心的单位圆上。在标准的平面坐标，我们通常将这个圆的方程写作 $x^2 + \quad = 1$

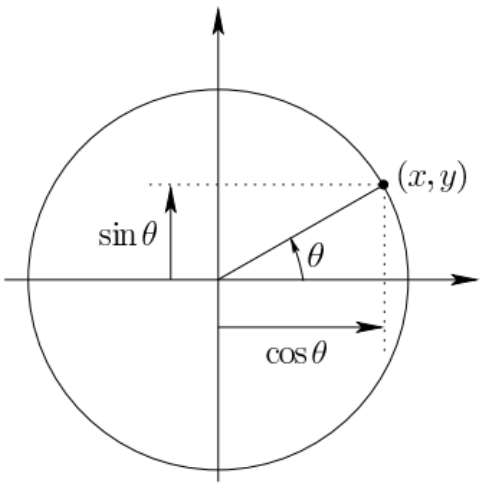

图 3.6 在单位圆中，可以用参数θ（0 ~ 2π）来表示圆上点的坐标

第一个约束条件（3.9）表示每一列的元素都要落在单位圆上，标准形式写为 $x^2+y^2=1$。我们也可以用角度θ（弧度 0 ~ 2π）来表示坐标（x，y），如图 3.6 所示：

$$x = \cos\theta \quad y = \sin\theta \tag{3.12}$$

我们用角度θ代替点的坐标（x，y），令 $m_{11}=\cos\theta$ $m_{21}= \sin\theta$，式（3.4）可以写为

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{3.13}$$

m_{12}，m_{22} 由式（3.10）和（3.11）唯一确定。通过θ的变化（0~2π范围内），可以表示所有 2D 旋转。

接下来我们讨论旋转的自由度（DOF）。一开始的时候，矩阵 M 的四个元素是可以任意取值的，也就是说有 4 个自由度。（3.9）中的约束条件减少两个自由度，（3.10）的约束条件也减少一个自由度，（3.11）并没有减少自由度，它只去掉了一半可能的变换，因为这些变换

是另一半的镜像翻转。最后，我们得出结论，2D 旋转只有一个自由度，由参数θ决定；而且，这些旋转都可以用单位圆的点来表示。

3D 旋转

现在我们尝试将 2D 旋转的模式扩展到三维空间中，式（3.4）矩阵的三维形式如下所示，包含 9 个元素：

$$\mathrm{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \tag{3.14}$$

因此，一开始 3D 旋转有着 9 个自由度，同样的，我们也要知道一个合法的 3D 旋转需要哪些限制条件。首先，每一列必须保证是单位长度。例如，$m_{11}^2 + m_{21}^2 + m_{31}^2 = 1$。这就是说每一列中的元素必须落在单位球面上。因此，单位长度的限制将 DOF 减少到 6 个。然后，根据正交轴定理，取矩阵的任意两列，它们的内积必须为 0。例如，我们选择前两列

$$m_{11}m_{12} + m_{21}m_{22} + m_{31}m_{32} = 0 \tag{3.15}$$

同理，对于后两列、一三列同样有如上的约束。每一种情况减少一个自由度，我们还剩下 3 个自由度。为了避免镜像出现，矩阵 M 的行列式要为 1，但这不减少自由度。

最后，我们得到了一系列满足代数约束的矩阵，然而，和 2D 旋转不同，它并不能用单位球面上的点来表示。我们只知道它有 3 个旋转自由度，意味着它应该可以由 3 个独立的参数来表示，（3.14）中的 9 个元素均可由这 3 个参数计算出。

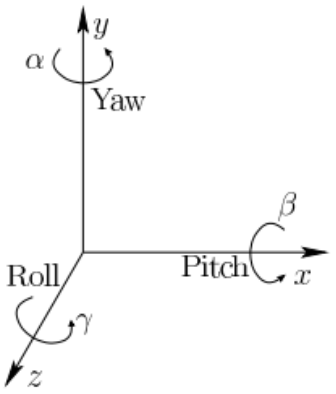

图 3.7 用偏航角（yaw）、俯仰角（pitch）、翻滚角（roll）可以描述任意三维旋转

偏航角、俯仰角、翻滚角

如何确定这 3 个参数呢？我们很容易想到，是不是可以构建几个 2D 旋转变换来描述 3D 旋转，如图 3.7 所示。先考虑绕 z 轴的旋转。我们用*翻滚角（roll）*来表示绕 z 轴逆时针方向的旋转γ。旋转矩阵如下

$$R_z(\gamma) = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.16}$$

可以看出，矩阵的左上部分就是 2D 旋转矩阵，仅仅是θ变成了γ。这是因为它实际上就是 xy 平面上的 2D 旋转。矩阵的剩余项看上去是一个单位阵，这使得经过翻滚角旋转后 z 方向保持不变。

相似地，我们用*俯仰角（pitch）*来表示绕 x 轴的逆时针旋转β。在这种情况下，x 轴保持不变而在 yz 平面进行 2D 旋转。

$$R_x(\beta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\beta & -\sin\beta \\ 0 & \sin\beta & \cos\beta \end{bmatrix} \quad (3.17)$$

最后，我们用*偏航角（yaw）*来表示绕 y 轴的逆时针旋转α，类似地，在 xz 平面进行 2D 旋转而 y 方向不变。

$$R_y(\alpha) = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix} \quad (3.18)$$

旋转组合

（3.16）（3.17）（3.18）分别代表旋转中的一个自由度。偏航角、俯仰角、翻滚角经过组合可以表示所有可能的 3D 旋转：

$$R(\alpha,\beta,\gamma) = R_y(\alpha)R_x(\beta)R_z(\gamma) \quad (3.19)$$

在式（3.19）中，α 和 γ 的取值范围都是 0~2π，然而β 的取值范围是-π/2 ~ π/2，尽管如此，它已经可以表示所有可能的 3D 旋转。

务必注意式（3.19）中的顺序不可颠倒，因为它不满足交换律。例如，先旋转π/2 的偏航角再旋转π/2 的俯仰角与先旋转俯仰角再旋转偏航角的结果是不一样的。你可以将π/2 代入到（3.17）（3.18）中，看看经过顺序不同的矩阵乘法的结果是怎样的。2D 旋转满足交换律是因为旋转轴总是相同的，所以旋转角可以加性结合。编写 VR 软件时把旋转矩阵的次序弄错了是令人崩溃的。

矩阵乘法是“后向的”

当我们对矩阵进行乘法运算时我们要注意什么？考虑对点 p=（x, y, z）进行旋转，我们有两个旋转矩阵 R 和 Q。如果我们用 R 进行旋转，可以得到$p' = Rp$，然后继续用 Q 进行旋转，可以得到$p'' = Qp'$。现在，我们希望将两个旋转组合起来从而直接从 p 得到 p''。程序员通常将它们组合成 RQ 因为我们是从左到右读的，也是这样理解的。然而，在线性代数中，它是后向的。因为 Rp 是先进行的运算，因此，它应该是从右向左读的。因此，我们必须将其组合成 QR，从而得到$p'' = QRp$。在本章的后面内容中，我们会将几个旋转矩阵链式结合在一起，要记得从右向左读来理解它们做了什么。

用单一矩阵完成平移和旋转

如果可以通过一次操作同时完成平移和旋转，那我们使用起来将会方便很多。我们用一个旋转矩阵 R 进行旋转，紧接着平移（x_t，y_t，z_t），代数形式如下所示：

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = R\begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} x_t \\ y_t \\ z_t \end{bmatrix} \quad (3.20)$$

尽管我们没有办法通过一个 3x3 的矩阵完成如上操作，但是我们可以增加矩阵的维度，用一个 4x4 的齐次变换矩阵来表示。

$$T_{rb} = \begin{bmatrix} & & & x_t \\ & R & & y_t \\ & & & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (3.21)$$

符号T_{rb}指代一个做刚体变换的矩阵，也就是说，不包含扭曲变形。一个齐次变换矩阵可能包含其他种类的变换，我们会在 3.5 节介绍。

式（3.20）表示的变换等价于原始点乘上（3.21）的变换，如下所示：

$$\begin{bmatrix} R & \begin{matrix} x_t \\ y_t \\ z_t \end{matrix} \\ \begin{matrix} 0 & 0 & 0 \end{matrix} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}. \tag{3.22}$$

因为存在额外的维度，我们对点（x，y，z）扩展一个维度得到（x，y，z，1）。要十分注意，因为平移和旋转是不能交换的，所以变换矩阵一定要满足（3.21）的格式。

反变换

我们经常会想要进行反变换，对于平移变换（x_t，y_t，z_t），只需简单地对其取相反数（$-x_t$，$-y_t$，$-z_t$）即可；对于一般的矩阵变换 M，我们取它的逆 M^{-1}（如果存在）。这计算起来通常很复杂，但幸运的是，对于我们所研究的情况，它们的逆计算起来要简单很多。对于旋转矩阵 R，取逆的过程相当于取它的转置 $R^{-1}=R^T$。对于齐次变换矩阵（3.21），它的反变换写为：

$$\begin{bmatrix} R^T & \begin{matrix} -x_t \\ -y_t \\ -z_t \end{matrix} \\ \begin{matrix} 0 & 0 & 0 \end{matrix} & 1 \end{bmatrix}. \tag{3.23}$$

这会将平移和旋转复原，然而，这个顺序是错的。要记住，我们必须要正确处理操作的顺序，因为它们是不可交换的，如图 3.8。对应一般的矩阵（非交换群理论中的一部分），对矩阵的乘积取逆时要将顺序颠倒过来：

$$(\mathrm{ABC})^{-1} = C^{-1}B^{-1}A^{-1} \tag{3.24}$$

我们可以将逆接在乘积的后面，即

$$ABCC^{-1}B^{-1}A^{-1} \tag{3.25}$$

首先，C 被自己的逆抵消，接着是 B，最后是 A。如果顺序有错的话，就不能产生抵消的效果。

矩阵T_{rb}（式 3.21）先旋转再平移，而式（3.23）同样是先旋转再平移，未将顺序反过来。因此，T_{rb}的逆应该是

$$\begin{bmatrix} R^T & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \\ \begin{matrix} 0 & 0 & 0 \end{matrix} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_t \\ 0 & 1 & 0 & -y_t \\ 0 & 0 & 1 & -z_t \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.26}$$

右面的矩阵先抵消平移操作，然后左面的矩阵抵消旋转。

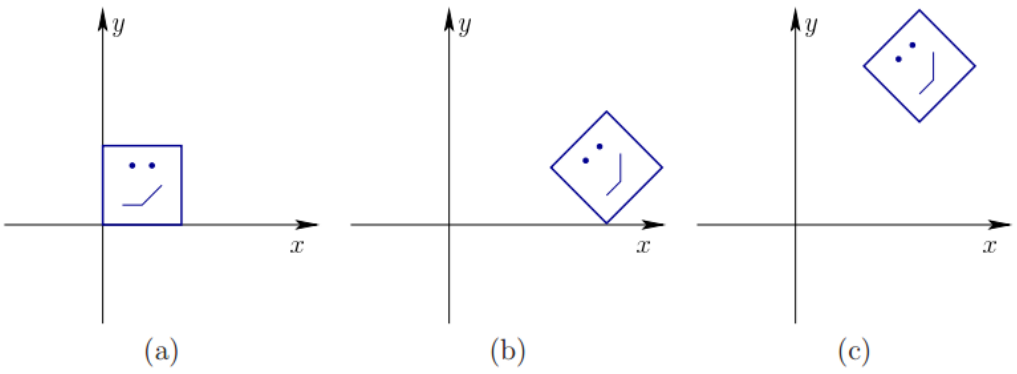

图 3.8 (a)一个 1x1 的刚体正方形 (b)将图形平移 x_t=2 再旋转π/4（45°）(c)颠倒顺序：先旋转π/4 再平移 x_t=2

3.3 旋转的轴-角表示

如 3.2 节所示，3D 旋转是很复杂的，原因有三：（1）参数化的过程不简单，仅由 3 个独立的参数来代替旋转矩阵中 9 个元素（2）每次的旋转轴是不同的。（3）运算是不可交换的，所以顺序非常重要。这些问题在 2D 旋转中都不存在。

运动奇异点

当我们使用偏航角、俯仰角、翻滚角（以及相关的欧拉角变量）时，有一个严重的问题。即使它们直觉上容易理解，但是一些具象化特征会有所退化，使得运动奇异点很难被发现。举一个小例子，考虑我们是如何表示地球上点的位置，我们用经纬度坐标来表示这些三维的点，就像是对于偏航角和俯仰角的限制，经度的范围是 0~2π，而纬度的范围只有-π/2~π/2（通常我们用东西经 0~180° 来描述经度，这是等价的）。当我们从一个地方到另一个地方时，经纬度坐标看上去就像是 xy 坐标；然而，这是位置点离极点较远的情况。当我们靠近北极点，虽然纬度还能保持正常，但是经度会发生变化，我们移动一点点距离，经度就会发生很大的变化。想想世界地图中极点周围是怎样的，巨大的格陵兰岛以及围绕整个底部的南极洲（假定在投影中经度仍然保持直线）。这些极点就是运动奇异点：在这些特殊的点中，你可以改变经度，但是在地球中的位置并不会改变。在这些点上两个自由度中似乎有一个不起作用了。

同样的问题也出现在 3D 旋转中，但由于额外的自由度，它变得更难理解。如果俯仰角保持$\beta = \pi/2$，就像是“北极点”一样，虽然α 和 γ的变换之间没有关系，但是只会产生一个自由度（就像是经纬度，虽然有一个参数在变化，但是它并没有实际效果）。我们看下偏航角、俯仰角、翻滚角是怎么组合的：

$$\begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos(\alpha-\gamma) & \sin(\alpha-\gamma) & 0 \\ 0 & 0 & -1 \\ -\sin(\alpha-\gamma) & \cos(\alpha-\gamma) & 0 \end{bmatrix} \quad (3.27)$$

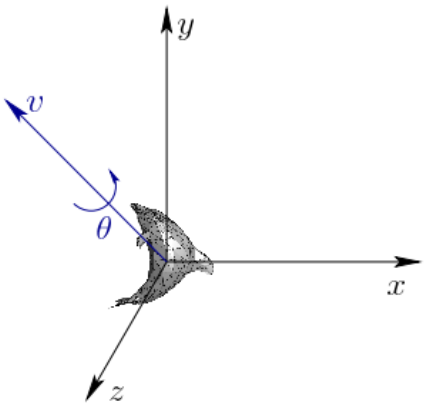

图 3.9：欧拉旋转理论指出，给定单位方向矢量 v=（v_1，v_2，v_3），每一个 3D 旋转都可以视为一个经过原点的角度 θ 的旋转

上式第二个矩阵对应着俯仰角β = π/2（3.17）。经过矩阵乘法和减法三角恒等式，我们得到右面的结果。可以发现，矩阵中的元素是一个关于α 和 γ的函数，但是仅有α 和 γ的差对结果有影响，所以它只有一个自由度。在视频游戏行业，大家对于这个问题的重视程度也不尽相同。在第一人称射击游戏（FPS）中，玩家是不会将头部的俯仰角一直维持±π/2，正好避免了这个问题。但在虚拟现实中，用户是可以让他的头部垂直向上/下的，所以这个问题一直存在，运动奇异点经常会使得可视区域的旋转不可控。这种现象也出现在用机械系统传感和控制航天器的方向时，称为*万向锁*问题。

这个问题用*轴-角*表示方法可以轻易的解决。它比起欧拉角来说难于学习，但是因为它可以避免这些问题，所以也是值得研究的。此外，很多现有的软件库和游戏引擎都直接应用了这种表示方法。因此，要有效率的使用这些开源库，你需要理解它的内容。

解决运动奇异点问题最重要的就是欧拉旋转定理，如图 3.9 所示。即使多次旋转组合之后旋转轴可能会变，但欧拉发现任何 3D 旋转都可以被表示为一个过原点的旋转θ。这与旋转的 3 个自由度是匹配的：它用两个参数来指示方向，一个参数来指示θ。唯一的缺点就是在旋转矩阵和轴-角表示之间来回变换有些不方便。这推动了一个新的数学概念的产生，它和轴-角表示类似，同样可以模拟 3D 旋转的代数形式，甚至可以直接应用到旋转模型中。叫做*四元数*。

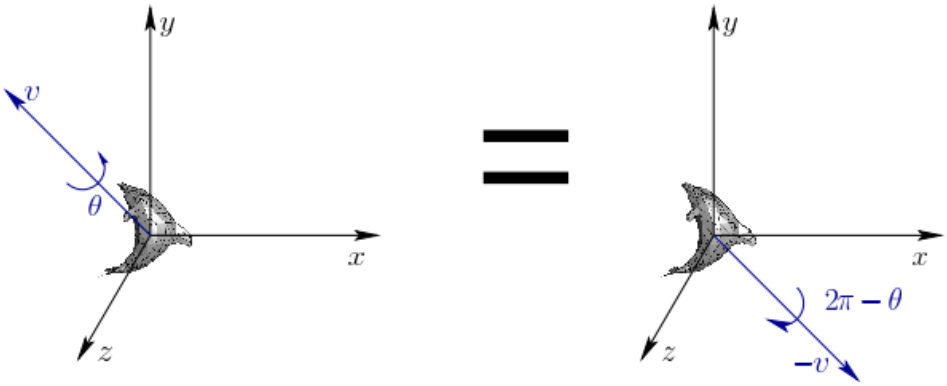

图 3.10：按照轴-角表示，同样的旋转可以用两种方式来表示，对应着方向 v 和-v

2 对 1 问题

在介绍四元数之前，我们要指出欧拉旋转定理中的一个问题。如 3.10 所示，轴-角表示不是唯一的。事实上，除了恒等变换，所有 3D 旋转都存在着两种表示方法。这是因为轴的

方向是分正负的。我们可以规定轴总是指向一个方向，如+y，但是这不能完全解决问题，因为存在着边界条件（水平轴）。除了这个无法避免的问题之外，四元数可以很好的解决的 3D 旋转的所有问题。

四元数由 William Rowan Hamilton 于 1843 年提出。第一次看到的时候，很多人无法理解它独特的代数形式。因此，我们先搞清楚四元数与其准确对应的 3D 旋转之间的关系。然后，我们介绍有限四元数代数；这在 VR 系统开发中用处不大，除非你想要自己实现 3D 旋转的工具库。然而，四元数和 3D 旋转的对应关系是非常重要的。

四元数 h 是一个 4D 的向量：

$$\mathrm{q} = (\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}) \qquad (3.28)$$

a,b,c,d 均取实数。因此，q 可以认为是 4 维空间的一个点。经过证明，我们仅能用*单位四元数*来表示 3D 旋转，也就是说要保证

$$a^2 + b^2 + c^2 + d^2 = 1 \qquad (3.29)$$

可以看出，这是一个在单位球面上的方程，只是维度要更高一些。通常的球面是一个 2D 平面，而单位四元数是在一个 3D 的“超平面”上，通常也叫做*流形*【1，5】。我们将用单位四元数空间来表示所有的 3D 旋转。它们都具有 3 个自由度，看上去似乎是合理的。

Quaternion	Axis-Angle	Description
$(1,0,0,0)$	$(\text{undefined}, 0)$	Identity rotation
$(0,1,0,0)$	$((1,0,0),\pi)$	Pitch by π
$(0,0,1,0)$	$((0,1,0),\pi)$	Yaw by π
$(0,0,0,1)$	$((0,0,1),\pi)$	Roll by π
$(\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}},0,0)$	$((1,0,0),\pi/2)$	Pitch by $\pi/2$
$(\frac{1}{\sqrt{2}},0,\frac{1}{\sqrt{2}},0)$	$((0,1,0),\pi/2)$	Yaw by $\pi/2$
$(\frac{1}{\sqrt{2}},0,0,\frac{1}{\sqrt{2}})$	$((0,0,1),\pi/2)$	Roll by $\pi/2$

图 3.11：这些情况下，可以很清楚地看出 3D 旋转四元数表示和对应的轴-角表示

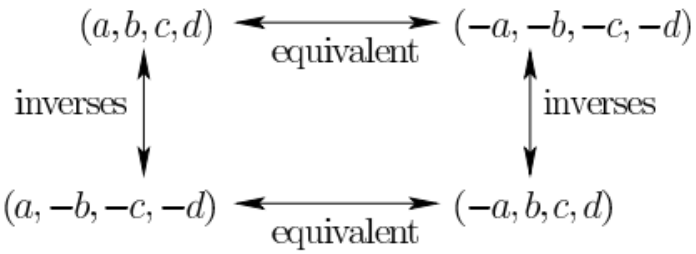

图 3.12：相等的四元数及它们的相反数示意图

令（v，θ）为 3D 旋转的轴-角并表示，如图 3.9 所示。它可以转化为下面的四元数形式：

$$\mathrm{q} = (\cos\frac{\theta}{2}, v_1 \sin\frac{\theta}{2}, v_2 \sin\frac{\theta}{2}, v_3 \sin\frac{\theta}{2}) \qquad (3.30)$$

将 q 考虑成一个表示 3D 旋转的数据结构，同样的，也可以将 q 转换为（v，θ）

$$\theta = 2cos^{-1}a \ \ and \ \ v = \frac{1}{\sqrt{1-a^2}}(b,c,d) \qquad (3.31)$$

如果 a=1，那么式（3.31）不成立，这也正对应了恒等旋转。

你现在可以在（v，θ）和 q 之间相互转换。图 3.11 展示了几个简单的，经常会出现的例子。此外，图 3.12 展示了四元数及其对应旋转之间的关系。水平箭头的 q 和-q 表示相同

的旋转，对应着图 3.10 中的两种表示方式；从式（3.30）中也可以看出它们之间的相等关系。竖直箭头对应着反向旋转，是因为轴的方向反向了（旋转θ变成了旋转 2π-θ）。

我们怎么将四元数 h=（a,b,c,d）应用到旋转模式？一种方式是如下面所示，将四元数变换为旋转矩阵：

$$\mathrm{R(h)} = \begin{bmatrix} 2(a^2+b^2)-1 & 2(bc-ad) & 2(bd+ac) \\ 2(bc+ad) & 2(a^2+c^2)-1 & 2(cd-ab) \\ 2(bd-ac) & 2(cd+ab) & 2(a^2+d^2)-1 \end{bmatrix} \qquad (3.32)$$

也可以不用这么麻烦，我们定义四元数乘法，对于任意两个四元数 q_1 和 q_2，乘积用 q_1*q_2 表示，定义如下：

$$\begin{aligned} a_3 &= a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2 \\ b_3 &= a_1b_2 + a_2b_1 + c_1d_2 - c_2d_1 \\ c_3 &= a_1c_2 + a_2c_1 + b_2d_1 - b_1d_2 \\ d_3 &= a_1d_2 + a_2d_1 + b_1c_2 - b_2c_1 \end{aligned} \qquad (3.33)$$

换句话说，$q_3 = q_1*q_2$ 由式（3.33）定义。

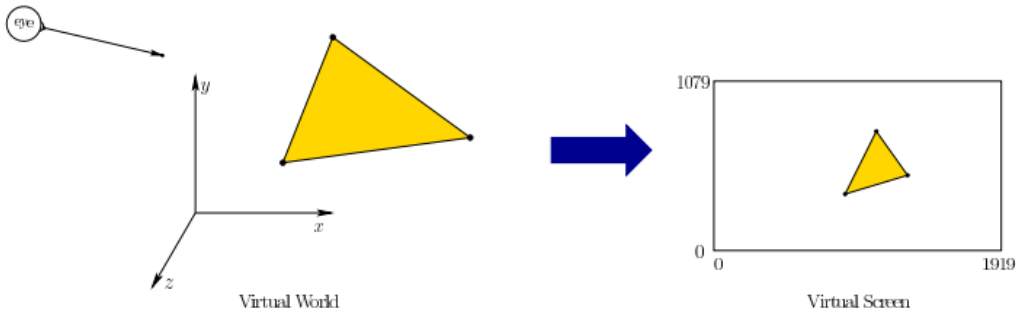

图 3.13：如果我们将虚拟眼睛或相机放入虚拟世界，它会看到什么？ 3.4 节提供了基于虚拟眼睛的特定视点，能将物体从虚拟世界放置到虚拟屏幕上。 由于工程和历史原因，虚拟屏幕选择矩形的形状，尽管它与视网膜的形状不匹配。

我们使用关于 h 的旋转表达式来对（x，y，z）点进行旋转。令 p=（x，y，z，1），这是为了给点 p 与四元数相同的维数。通过应用四元数乘法来旋转这个点，如下式所示：

$$p' = q*p*q^{-1}, \qquad (3.34)$$

其中 q^{-1}，=（a，-b，-c，-d）（从图 3.12 中可得到）。 旋转后的点是（x'，y'，z'），它取自结果 p'=（x'，y'，z'，1）。

我们举一个简单的例子：点（1，0，0）。假设 p=（1,0,0,1），并考虑执行偏航旋转 π/2。根据图 3.11，相应的四元数是 q=（0,0,1,0）。 q^{-1},=（0,0，-1,0）。应用式（3.33）计算式（3.34）后，结果为p'=（0,1,0,1）。因此，在偏航角为 π/2 时，旋转得到的点是（0,1,0）。

3.4 视角变换

本节介绍如何转换虚拟世界中的模型，以便它们出现在虚拟屏幕上。主要目的是为图形渲染奠定基础，从而能增加光照，材料特性和量化效果，最终可以实现在物理显示屏上显示。这些转换的另一方面也解释了相机如何形成图像，至少是一个理想的成像过程。将这些转换部分看作描述在虚拟世界中的虚拟相机，那么用该相机拍摄的虚拟照片应该是什么样子？为了使 VR 正常工作，“相机”实际上应该是置于虚拟世界中的两个虚拟眼中的一个。因此，根据虚拟世界中的位置和方向，虚拟眼睛应该看到什么？这里并不是指在视网膜上会出现什么，这部分将会在 4.4 节之后讲解，这里我们只是计算模型顶点在虚拟世界中的平面矩形屏

幕上的出现位置。见图 3.13。

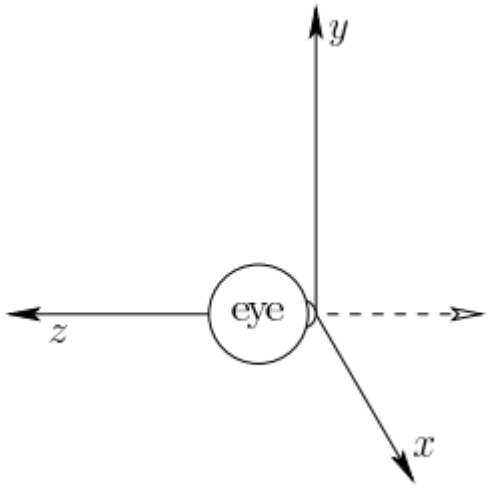

图 3.14：考虑一个在负方向上向下看 z 轴的眼睛。模型的起源是光线进入眼睛的点。

人眼观看

图 3.14 显示了一个俯视负 Z 轴的虚拟眼睛。它的放置位置从眼睛的角度来看，x 向右增加，y 向上，这对应于一般的笛卡尔坐标。替代方案可以是：1）在正 z 方向上朝向眼睛，这使得 xy 坐标平面向后推移，或者 2）颠倒 z 轴，这将导致左手坐标系。 因此，我们做出选择可以来避免了更糟的情形。

假设眼睛是我们想要放置在三维虚拟世界中的物体模型，我们可以在某个位置 e=（e1，e2，e3）和矩阵给出的方向，如下式所示：

$$R_{eye} = \begin{bmatrix} \hat{x}_1 & \hat{y}_1 & \hat{z}_1 \\ \hat{x}_2 & \hat{y}_2 & \hat{z}_2 \\ \hat{x}_3 & \hat{y}_3 & \hat{z}_3 \end{bmatrix} \tag{3.35}$$

如果图 3.14 中的眼球是由三角形组成的，那么 R_{eye} 的旋转和 e 的平移将应用于所有的三维空间的顶点中。

然而，这并不能解决虚拟世界如何出现在眼前的问题。我们需要将虚拟世界中的所有模型移动到眼睛的参照系，而不是在虚拟世界中移动眼睛。这意味着我们需要应用逆变换。 旋转的逆是R_{eye}^{T}，R_{eye} 的转置。e 的倒数是-e。应用（3.26）做适当的变换：

$$T_{eye} = \begin{bmatrix} \hat{x}_1 & \hat{x}_2 & \hat{x}_3 & 0 \\ \hat{y}_1 & \hat{y}_2 & \hat{y}_3 & 0 \\ \hat{z}_1 & \hat{z}_2 & \hat{z}_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_1 \\ 0 & 1 & 0 & -e_2 \\ 0 & 0 & 1 & -e_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.36}$$

请注意，如（3.35）中所示的 R_{eye} 已被转置并放置在上面的左侧矩阵中。另外，转换顺序和旋转顺序已经交换过，如 3.2 节所述，逆顺序是必需的。

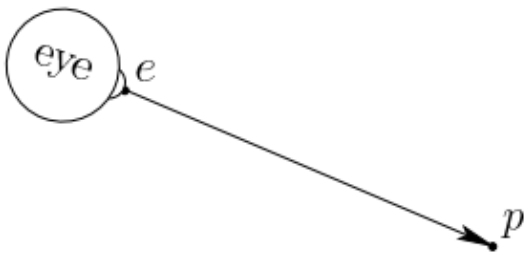

图 3.15：从眼睛位置 e 到它正在查看的点 p 的矢量被归一化为（3.37）中的 c。

在图 3.4 后，式（3.36）有两种可能的解释。如前所述，这可能对应于移动所有虚拟世界模型（对应于图 3.4（b））。但在当前的设置中，更合适的解释是虚拟世界的坐标系正在移动，以便它与图 3.14 中的眼睛帧相匹配。这对应于图 3.4（c）的情况，不是 3.2 节中的适当解释。

从 VR 的外观来看，虚拟世界中眼睛的位置和方向是由跟踪系统和可能的控制器作为输入。相比之下，在计算机图形学中，通常首先描述眼睛的位置以及眼睛所看的方向。这被称为“look-at”，具有以下部分：

1.眼睛的位置：e

2.眼睛的中心方向：$\hat{c}$

3.向上的方向：$\hat{u}$。

$\hat{c}$和$\hat{u}$都是单位向量。 第一个方向$\hat{c}$对应于视角的中心。无论$\hat{c}$指向什么都应该最终在显示屏的中心。如果我们希望这是三维空间中的特定点 p（见图 3.15），则 c 可以计算为

$$\hat{c} = \frac{p - e}{\|p - e\|}, \tag{3.37}$$

其中‖ ‖表示矢量的长度。结果是从 e 到 p 的矢量并做归一化。

第二个方向$\hat{u}$指示哪个方向为向上的方向。想象一下拿着相机，就好像你要拍一张照片然后进行旋转一样。你可以使图像中的水平地面看起来倾斜或甚至颠倒。因此，$\hat{u}$表示虚拟相机或眼睛的向上方向。

我们现在从式（3.36）构造 T_{eye} 变换。转换部分已经由 e 来确定，这是在“look-at”中给出的。我们只需要确定旋转 R_{eye}，如（3.35）所示。回想一下在 3.2 节中矩阵列指出了矩阵如何转换坐标轴（参考（3.7）和（3.8））。这简化了确定 R_{eye} 的问题。每个列向量计算为

$$\begin{aligned} \hat{z} &= -\hat{c} \\ \hat{x} &= \hat{u} \times \hat{z} \\ \hat{y} &= \hat{z} \times \hat{x}. \end{aligned} \tag{3.38}$$

计算$\hat{z}$会出现负号，因为眼睛正在向下看 z 轴。$\hat{x}$方向使用标准叉积$\hat{z}$来计算。对于第三个方程，我们可以使用$\hat{y} = \hat{u}$; 然而，$\hat{z} \times \hat{x}$会巧妙地纠正$\hat{u}$通常指向上方但不垂直于$\hat{c}$的情况。式（3.38）中的单位矢量代入（3.35）可得到 R_{eye}。 因此，我们拥有构建 T_{eye} 所需的全部信息。

正交投影

当 T_{eye} 已知，设（x，y，z）表示任意点的坐标。如果我们强制把每个 z 坐标取值为 0，然后将所有点直接投影到垂直 xy 平面中，将会发生什么？换句话说，也就是（x，y，z）→（x，y，0），这就是所谓的正交投影。如果我们将 xy 平面想象为模型的虚拟展示，那么会出现以下几个问题：

1.混杂的物体将被叠加，而不是物体之间有相互遮挡的关系。

2.显示画面将在所有方向无限延伸（z 除外）。如果显示器是 xy 平面上的一个小矩形，则超出范围的模型部分可以被消除掉。

3.距离较近的物体应大于距离较远的物体，这存在于现实世界中。我们可以回想一下第 1.3 节（图 1.22（c））正确处理视角的绘画。

前两个问题是重要的图形操作，这部分的介绍将在第 7 章讲述。我们要解决第三个问题。

透视投影

我们定义一个透视投影，而不用上述提到的正交投影。 对于每个点（x，y，z）它位于通过原点的一条线上。 下式表示所有点的集合：

$$（\lambda x，\ \lambda y，\ \lambda z），\qquad (3.39)$$

其中 λ 可以是任何实数。 换句话说， λ 是一个能够到达线上所有点（包含（x，y，z）和（0，0，0））的参数，见图 3.16。

现在我们可以在虚拟世界的任何地方放置一个平面的“电影屏幕”，并查看所有线条穿透它的位置。 为了简单化这个数学表达式，我们选择 z = -1 平面将我们的虚拟屏幕直接放在眼睛的前方，见图 3.17。使用（3.39）的第三个分量，我们有 λ z= -1，这意味着 λ = -1 / z。使用（3.39）的前两个分量，屏幕上点的坐标计算为 x'= -x / z，y'= -y / z。 请注意，由于 x 和 y 对于每个轴的缩放量 z 相同，因此它们的宽高比将保留在屏幕上。

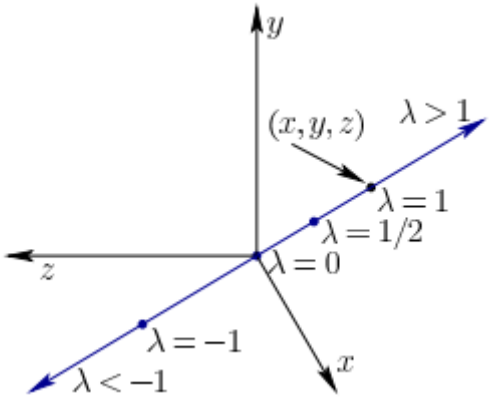

图 3.16：从任意点（x，y，z）开始，通过原点的线可以使用参数 λ 形成。 它是任何实数 λ 的形式（ λ x， λ y， λ z）的所有点的集合。 例如， λ = 1/2 对应于（x，y，z）和（0,0,0）沿线的中点。

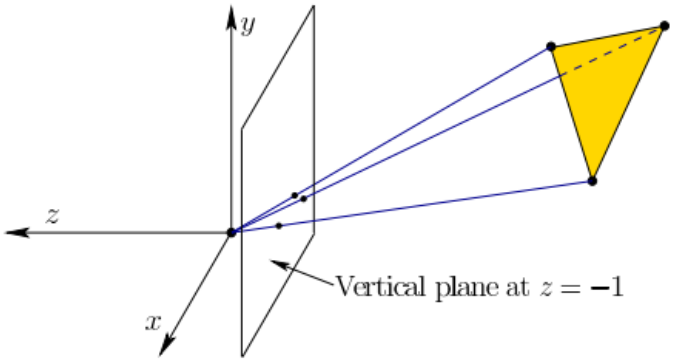

图 3.17：透视投影的例证。 模型顶点通过绘制直线和原点（0,0,0）投影到虚拟屏幕上。 虚拟屏幕上的点的“图像”对应于线条与屏幕的交点。

更一般地说，假设垂直屏幕放置在沿着 z 轴的某个位置 d 处。 在这种情况下，我们可以获得更多的通用表达式来显示屏幕上某个点的位置：

$$\begin{aligned} x' &= dx/z \\ y' &= dy/z. \end{aligned} \qquad (3.40)$$

这是通过对 λ 求解 d = λ z 并将其代入（3.39）得到的。

这就是我们将点投影到虚拟屏幕上，同时考虑到各种距离下物体的缩放属性时所需要做

的所有事情。在虚拟现实中正确使用这一点有助于理解深度和规模，这在第 6.1 节中介绍。在 3.5 节中，我们将使用变换矩阵来调整（3.40）。 此外，只有位于眼前区域的点才会投射到虚拟屏幕上。太靠近，太远或超出正常视野范围的点将不会在虚拟屏幕上呈现; 这在第 3.5 节和第 7 章中讨论。

3.5 转换级联

本节将本章的所有转换级联在一起，同时稍微调整其形式以适应 VR 和计算机图形工业目前使用的形式。 本节中出现的一些矩阵可能看起来有些复杂。 原因是因为表达式是由算法和硬件问题二者同时驱动的，而不是仅是数学表达式那么简单。 特别是，即使在不是线性的透视投影（式（3.40））的情况下，也存在偏向于将每个变换放入 4×4 齐次变换矩阵。以这种方式，可以在矩阵链上迭代有效的矩阵乘法算法以产生结果。

该级联的一般表达式如下：

$$T = T_{vp}T_{can}T_{eye}T_{rb}. \tag{3.41}$$

当 T 应用于点（x，y，z，1）时，会生成屏幕上点的位置。请记住，这些矩阵乘法不可交换，并且操作顺序是从右向左。第一个矩阵 T_{rb} 是应用于可移动模型上的点的刚体变换（3.21）。对于模型中的每个刚体，T_{rb} 保持不变，然而，不同的物体通常会放置在不同的位置和方向。例如，“虚拟汽车的轮子移动”将会不同于“阿凡达的头部”。根据（3.36），在应用 T_{rb} 之后，T_{eye} 将虚拟世界转换为眼睛的坐标系。在一个固定的瞬间，虚拟世界中所有点的 T_{eye} 和所有剩余的变换矩阵是相同的。在这里，我们假定眼睛位于两个虚拟眼之间的中点，从而形成一个环视点。在本节的后面，我们将把它扩展到左右眼的情况，以便构建立体视点。

规范化视图变换

下一个变换，T_{can} 执行 3.4 节中所述的透视投影; 然而，我们必须解释它是如何“强迫”进入 4x4 矩阵的。 再次受到工业需求的驱动，我们希望变换的结果是一种看起来没有单位的规范形式，因此，T_{can} 被称为规范视图变换。图 3.18 显示了一个基于矩形虚拟屏幕的四个角的视锥。z = n 和 z = f 分别为近平面和远平面。请注意，这些情况下 z <0，因为 z 轴指向相反的方向。虚拟屏幕包含在近平面中。透视投影应该将截锥内的所有点放置在以近平面为中心的虚拟屏幕上。运用到式（3.40）我们发现 d = n。

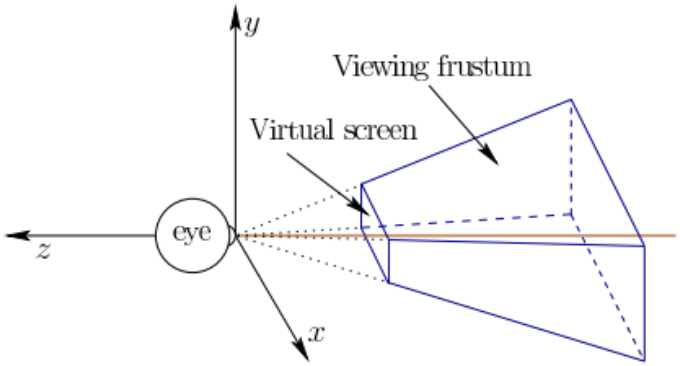

图 3.18 截掉的椎体视角

我们现在想用矩阵来重新推导出式（3.40）。 应用以下矩阵乘法的结果为：

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ nz \\ z \end{bmatrix} \tag{3.42}$$

在前两个坐标中，我们得到（3.40）的分子。 （3.40）的非线性部分是 1 / z 因子。 为了处理这个问题，第四个坐标用来表示 z，而不是 1，以此来处理 T_{rb}。 从这一点开始，所得到的 4D 矢量被解释为一个 3D 矢量，该矢量通过划分其第四个分量来缩放。 例如，（v1，v2，v3，v4）被解释为

$$(v_1/v_4, v_2/v_4, v_3/v_4) \tag{3.43}$$

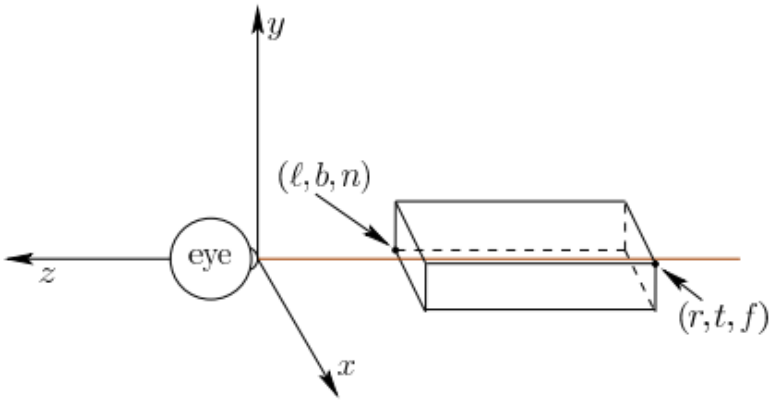

图 3.19：由 Tp 转换后的视锥角形成的矩形区域。 所选对角的坐标在 Tst 中使用的六个参数，l，r，b，t，n 和 f。

因此，（3.42）的结果被解释为：

$$(nx/z, ny/z, n), \tag{3.44}$$

其中前两个坐标与 d = n 匹配（3.42），第三个坐标是沿着 z 轴的虚拟屏幕的位置。

跟踪深度

以下矩阵常用于计算机图形学，并将在我们的级联中用于此处：

$$T_p = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.45}$$

除了转换 z 坐标的方式外，它与（3.42）中的矩阵相同。 在虚拟屏幕上放置点，这种做法是不必要的，因为我们已经知道它们全部放置在 z = n 处。 因此，z 坐标被用于另一个目的：跟踪每个点与眼睛的距离，以便图形算法可以确定哪些对象位于其他对象的前面。矩阵 Tp 计算第三个坐标为：

$$(n+f)z - fn \tag{3.46}$$

当将式（3.46）除以 z，我们不保留精确的距离，但图形方法（其中一些在第 7 章中介绍）仅需要保留距离排序。 换句话说，如果点 p 比点 q 离眼睛更远，那么即使距离失真，点 p 仍然会比 q 远。 但是，它确实保留了两种特殊情况下的距离：z = n 和 z = f。 这

可以通过将这些代入（3.46）并除以 z 来看出。

额外的转换和缩放

应用 Tp 后，平截头体的 8 个角被转换成矩形框的角，如图 3.19 所示。 以下内容对 z 轴进行了一个简单的转换，并进行了一些重新调整，使其在原点处居中，其角点坐标为（±1，±1，±1）：

$$T_{st} = \begin{bmatrix} \frac{2}{r-\ell} & 0 & 0 & -\frac{r+\ell}{r-\ell} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.47}$$

如果平截头体在 xy 平面上完全居中，则最后一列的前两个分量变为 0。最后，我们将（3.41）中的规范视图变换 T_{can} 定义为：

$$T_{can} = T_{st}T_p \tag{3.48}$$

视口变换

要在式（3.41）中应用的最后一个转换是视口转换 T_{vp}。 在应用了 T_{can} 之后，x 和 y 坐标的每个范围都从-1 到 1。需要最后一个步骤才能将投影点带到用于对物理显示中的像素进行索引的坐标。令 m 为水平像素的数量，n 为垂直像素的数量。 例如，对于 1080p 显示器，n＝1080，m＝1920。 假设显示索引的行数从 0 到 n-1，列数从 0 到 m-1。此外，（0，0）位于左下角。在这种情况下，视口变换是：

$$T_{vp} = \begin{bmatrix} \frac{m}{2} & 0 & 0 & \frac{m-1}{2} \\ 0 & \frac{n}{2} & 0 & \frac{n-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.49}$$

左右眼

我们现在讨论如何改变转换式（3.41）进行立体观看。设 t 表示左右眼之间的距离。其在现实世界中的值因人而异，其平均值在 t＝0.064 米左右。为了处理左眼视图，我们需要简单地将环眼（中心）眼睛水平向左移动。回想一下第 3.4 节，实际上反过来也是适用的。模型需要向右移动。 所以，

$$T_{left} = \begin{bmatrix} 1 & 0 & 0 & \frac{t}{2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.50}$$

当从眼睛的角度看时，这对应于模型的右移。这个转换放在 T_{eye} 之后再来调整它的输出。 对（3.41）做适当的修改：

$$T = T_{vp}T_{can}T_{left}T_{eye}T_{rb}. \tag{3.51}$$

通过对称性，通过用（3.51）替换 T_{left} 类似地处理右眼：

$$T_{right} = \begin{bmatrix} 1 & 0 & 0 & -\frac{t}{2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.52}$$

这就解释了在虚拟世界中放置和移动模型的整个转换公式，然后让它们出现在显示器的正确位置。在阅读第4 章之后，很明显，在整个公式应用之后可能需要进行最后的转换。这是为了补偿由于VR头盔中广角镜头引入的非线性光学畸变。

进一步阅读

大多数矩阵变换出现在标准的计算机图形文本中。 本章中的介绍严格遵循[202]。 有关四元数及其相关代数性质的更多详细信息，请参见[157]。 机器人文本通常涵盖了刚体和链体的三维变换，并且还考虑了单元的奇异性; 见[163，303]。