# 1.1 向量

计算机绘图、碰撞检测和物理模拟是现代视频游戏的基本组成部分，向量（vector）在这些领域中具有至关重要的作用。本书所采用的教学方式与普通教科书不同，我们所提供的所有的知识都是基于实践的；我们在这里为读者推荐一本专门讲解 3D 游戏与绘图的数学教程[Verth04]。读者在本书的每个演示程序中都会看到一些注释，我们通过这些注释来强调向量的重要性。

**学习目标**

1．学习使用几何与数值方式表示向量。

2．了解向量运算及向量的几何应用。

3．熟悉 XNA 数学中的向量函数和向量类。

向量（vector）是一种同时具有大小和方向的物理量（quantity）。同时具有大小和方向的物理量称为向量值物理量（vector-valued quantity）。常见的向量值物理量有：力（在某个特定方向上施加一定的作用力——量值），位移（在某个净方向上移动一段距离），速度（速率和方向）。因此，向量可以用来表示力、位移和速度。有时我们也用向量来表示一个单个方向，比如玩家在 3D 游戏中的观察方向、多边形面对的方向、光线的传播方向以及从一个物体表面折回的反射光方向。

首先，我们从几何学角度描述向量的算术特征：我们通过一个有向线段来表示向量（参见图 1.1），其中长度表示向量的大小，箭头表示向量的方向。我们注意到，向量所描绘的位置并不重要，改变向量的位置并不会影响向量的大小和方向（这是向量具有的两个属性）。也就是，当且仅当两个向量具有相同的长度和方向时，我们说这两个向量相等。所以，图 1.1a 中的两个向量 **u** 和 **v** 实际上是相等的，因为它们具有相同的长度和方向。其实，位置对于向量来说无关紧要，我们可以随便平移一个向量，但是不会改变该向量所表示的含义（因为平移即不会改变向量的长度，也不会改变向量的方向）。注意，我们可以平移 **u**，使它与 **v** 重叠（反之亦然），由此使它们彼此难以区分——因为它们是相等的。举一个物理上的例子，图 1.1b 中的向量 **u** 和 **v** 表示位于不同位置上的两只蚂蚁 A 和 B 从当前位置开始向北移动 10 米。也就是这里的 **u=v**。由于向量本身与位置无关；所以它们只是告诉蚂蚁该从当前位置向哪个方向移动，以及移动多远的距离。在本例中，它们告诉蚂蚁向北（方向）移动 10 米（长度）。

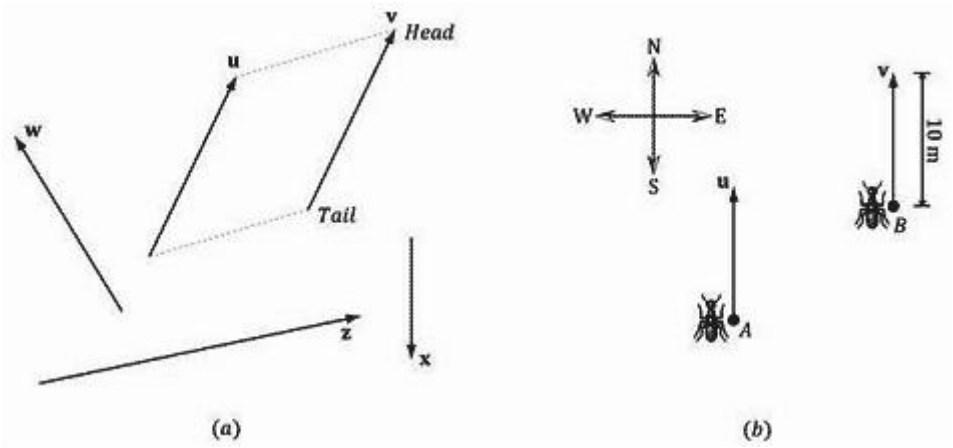



**图 1.1 （a）绘制在 2D 平面上的向量。（b）向量告诉蚂蚁向北移动 10 米。**

## 1.1.1 向量和坐标系

我们现在学习如何定义用于几何运算的向量，随后我们要用向量来解决各种向量值问题。不过，我们无法在计算机中以几何方式表示向量，我们必须寻求一种替代方案，以数字方式表示向量。所以我们在空间中引入了 3D 坐标系的概念，并将所有的向量尾部平移到坐标系原点（图 1.2）。然后通过指定向量首部的坐标来表示一个向量，记作 **v**= (*x*,*y*,*z*)，如图 1.3 所示。现在，我们可以在计算机程序中用 3 个浮点数来表示一个向量了。

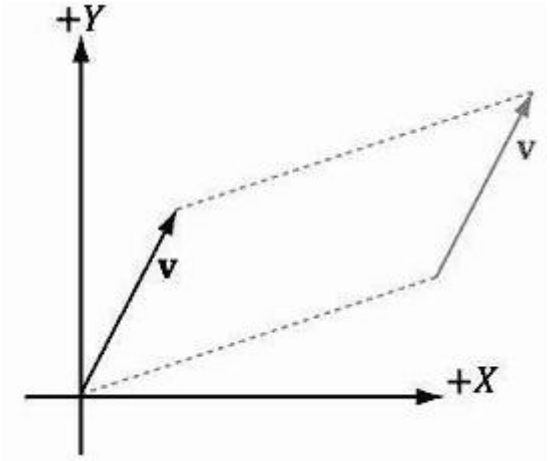



**图 1.2 平移向量 v，使它的尾部与坐标系的原点对齐。当一个向量的尾部与原点对齐时，我们说该向量位于标准位置。**

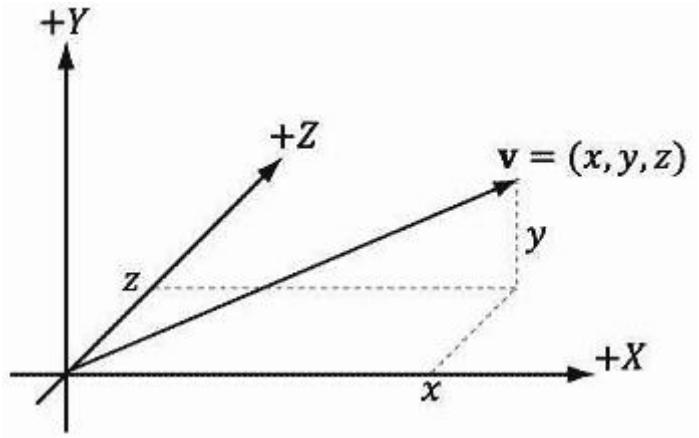



**图 1.3 以坐标表示的相对于坐标系的向量。**

**注意**：若只处理 2D，那么只需要使用一个 2D 坐标系统，向量只有两个坐标：**v**= (*x*,*y*)，在计算机程序中可以用 2 个浮点数来表示一个向量。

考虑图 1.4，它展示了一个向量 **v** 以及空间中的两个参照系。注意，我们在本书中使用术语 frame、参照系（frame of reference）、空间（space）、坐标系（coordinate system）来表示相同含义。我们平移 **v**，使它位于两个参照系的标准位置上。可以看到，向量 **v** 相对于参照系 A 的坐标不同于向量 **v** 相对于参照系 B 的坐标。换句话说，对于不同的参照系，同一个向量会有不同的坐标。

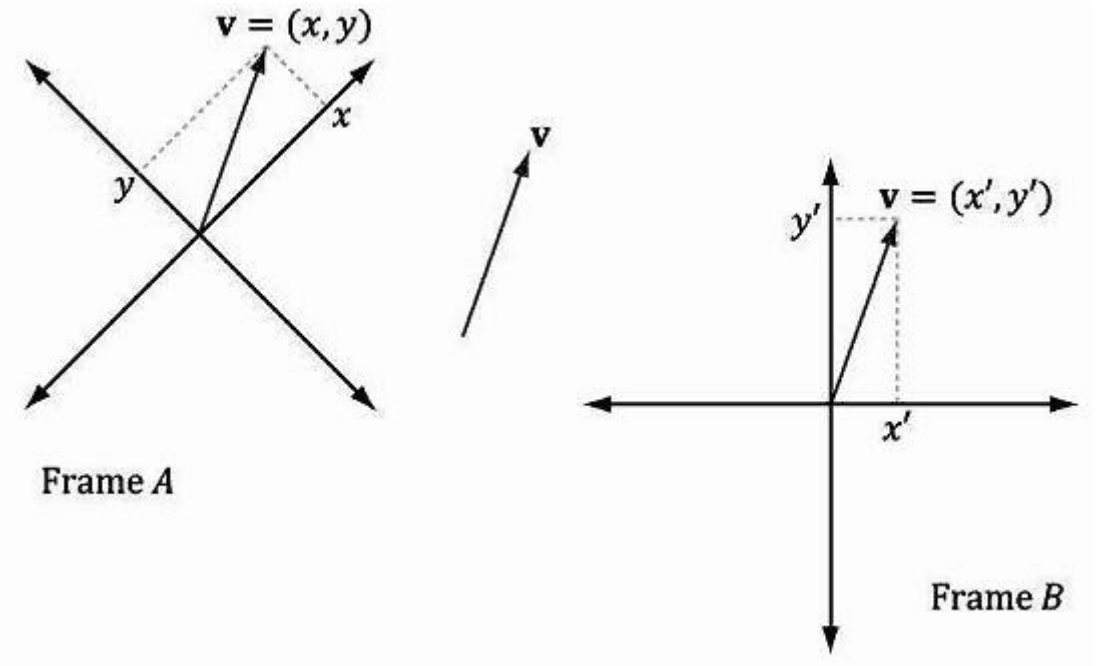



**图 1.4 当以不同参照系描述时，同一个向量 v 会有不同的坐标。**

这一概念说起来与物理学中的温度颇为相似。水的沸点是摄氏 100º 或华氏 212º 。使水沸腾的物理温度是相同的，与温标无关（即，我们不可能通过改变温标来提高或降低水的沸点），但是我们必须根据当前选用的温标来指定不同的标量值。同样，对于一个向量来说，它的方向和大小被牢牢地固定在有向线段中，不会改变；只有在以不同的参照系描述向量时，它的坐标才会改变。这一点很重要，因为我们无论何时通过坐标来表示一个向量，这些坐标都是相对于某一参照系的。通常在 3D 计算机绘图中我们使用的参照系不只一个，所以必须保证向量坐标与对应的参考系一致；另外，我们需要知道如何完成从一个参照系到另一个参照系的向量坐标转换。

**注意**：我们看到，在一个参照系中向量和点都可以使用坐标(*x*,*y*,*z*)来描述。但是，它们的意义完全不同；点表示的是三维空间中的位置，而向量表示的是大小和方向。点会在随后的 1.5 节中讨论。

## 1.1.2 左手坐标系和右手坐标系

Direct3D 使用所谓的左手坐标系（left-handed coordinate system）。假如你把左手手指指向 *x* 轴正方向，然后朝 *y* 轴正方向弯曲四指，大拇指就会指向 *z* 轴正方向。图 1.5 展示了左手坐标系和右手坐标系之间的区别。

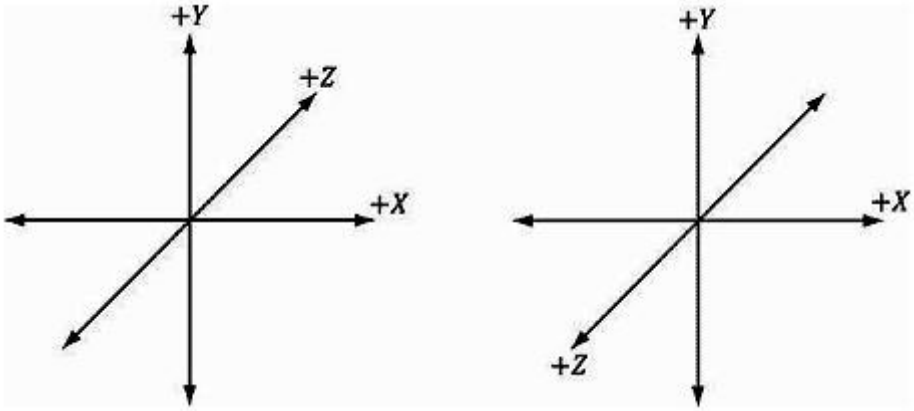



**图 1.5 左边是左手坐标系，*z* 轴正方向向页面内部延伸。右边是右手坐标系，*z* 轴正方向向页面外部延伸。**

观察右手坐标系。假如你把右手手指指向 *x* 轴正方向，然后朝 *y* 轴正方向弯曲四指，大拇指就会指向 *z* 轴正方向。

## 1.1.3 基本向量运算

现在，我们使用坐标来描述向量的判等运算、加法运算、标量乘法运算和减法运算。我们设 $\mathbf{u}=(u_x,u_y,u_z)$和 $\mathbf{v}=(v_x,v_y,v_z)$。

1. 当且仅当两个向量的各自分量相等时，这两个向量相等。也就是说，仅当 $u_x=v_x$、$u_y=v_y$、$u_z=v_z$时，**u**=**v**。

2．两个向量的对应分量可以相加；仅当两个向量的维度相同时，向量分量加法才有意义。$\mathbf{u}+\mathbf{v}=(u_x+v_x,u_y+v_y,u_z+v_z)$。

3. 标量（即，实数）可以与向量相乘，所得结果仍为向量。设 *k* 为标量，则 $k\mathbf{u}=(ku_x,ku_y,ku_z)$。这种向量运算称为标量乘法。

4．减法可以通过向量加法和标量乘法来表示。也就是，$\mathbf{u}-\mathbf{v}=\mathbf{u}+(-1\cdot\mathbf{v})=\mathbf{u}+(-\mathbf{v})=(u_x-v_x,u_y-v_y,u_z-v_z)$。

# 例 1.1

设 **u**= (1,2,3)、**v**= (1,2,3)、**w**=(3, 0, −2)、*k*=2。则，

1．**u**+**w**= (1, 2,3) + (3, 0, −2) = (4, 2,1)；

2．**u**=**v**；

3．**u**−**v**=**u**+ (−**v**) = (1,2,3) + (−1, −2, −3) = (0, 0,0) = 0；

4．k**w**=2(3, 0, −2) = (6, 0, −4)。

第 3 行例举了一个特殊的向量，称为零向量。零向量的所有分量均为零，它可以由 0 来表示。

# 例 1.2

我们将用个例子来解释如何使用 2D 向量来简化绘图操作。3D 向量的原理基本相同，

只是比 2D 向量多了一个分量而已。

**1**．设 **v**= (2,1)。在几何学中 **v** 和−**v**/2 是如何进行比较的呢？把 **v** 和−**v**/2 同时绘制出来（图 1.6a），我们注意到−**v**/2= (−1, −1/2)，它们的方向恰好相反，而−**v**/2 的长度是 **v** 的一半。所以，在几何学中对向量取负值可以“反转”向量的方向，而标量乘法可以缩放向量的长度。

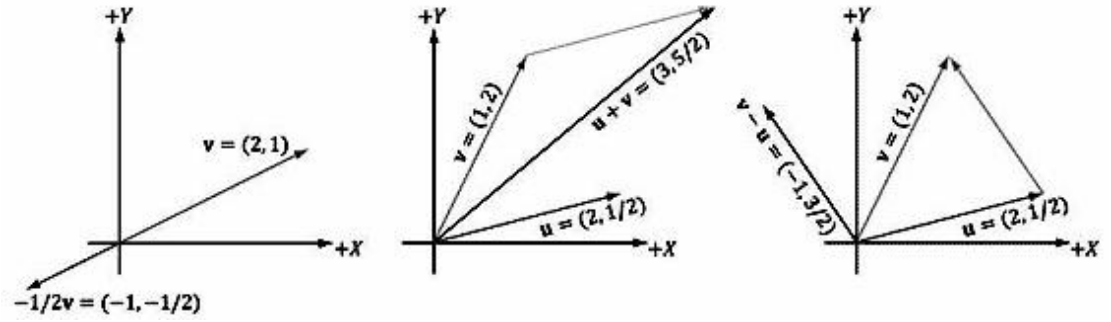



**图 1.6 (a)标量乘法的几何表示。(b)向量加法的几何表示。(c)向量减法的几何表示。**

**2**．设 **u**= (2,1/2)、**v**= (1,2)，则 **u**+**v**= (3,5/2)。图 1.6b 描绘了向量加法在几何学中的含义：对 **u** 进行平移，使其尾部与 **v** 的首部对齐。那么，二者之和等于从 **v** 的尾部开始、到 **u** 的首部结束的向量。（如果我们保持 **u** 的位置不变，对 **v** 进行平移，使 **v** 的尾部和 **u** 的首部对齐，那么得到的结果完全相同。此时，**u**+**v** 所得的向量从 **u** 的尾部开始、到平移后的 **v** 的首部结束。）另外，我们还可以看到向量加法的运算规则与我们根据自然规律从生活经验中判断出的结果一致，当我们把多个作用力加在一起时， 可以得到一个最终的净作用力：如果将同一个方向上的两个作用力（向量） 相加，那么就可以得到该方向上的一个更大的作用力（更长的向量）。如果将两个方向相反的作用力（向量）相加，那么就会得到一个较小的净作用力（较短的向量）。图 1.7 说明了这一概念。

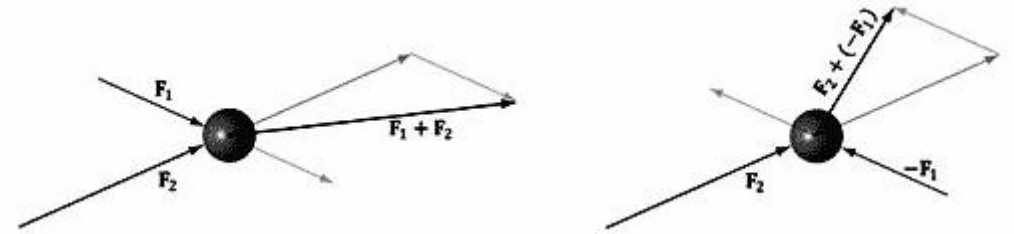



**图 1.7 施加在一个球上的作用力。向量加法将些作用力合为一体，得到一个净作用力。**

3．设 **u**=(2,1/2)、**v**= (1,2)，则 **v**-**u**= (−1,3/2)。图 1.6c 说明了向量减法的几何含义。本质上，**v**−**u** 得到的差值等于从 **u** 的首部开始、到 v 的首部结束的一个向量。如果我们把 **u** 和 **v** 视为点而不是向量，那么 **v**−**u** 得到的差值就等于从点 **u** 开始、到点 **v** 结束的一个向量；这种解释方式非常重要，因为我们经常需要从一点指向另一点的向量。同时我们还可以看到，当把 **u** 和 **v** 视为点时，**v**−**u** 的长度就是从 **u** 到 **v** 的距离。

# 1.2 长度和单位向量

在几何学中，向量的大小等于有向线段的长度。我们用双竖线来表示向量的大小（例如，‖**u**‖表示 **u** 的大小）。现在，给出一个向量 **u**=(*x*,*y*,*z*)，我们希望以代数方式计算它的大小。通过运用两次毕达哥拉斯定理可以计算出 3D 向量的大小（译者注：毕达哥拉斯定理和勾股定理的概念相同，换言之，毕达哥拉斯定理就是勾股定理）；参见图 1.8。

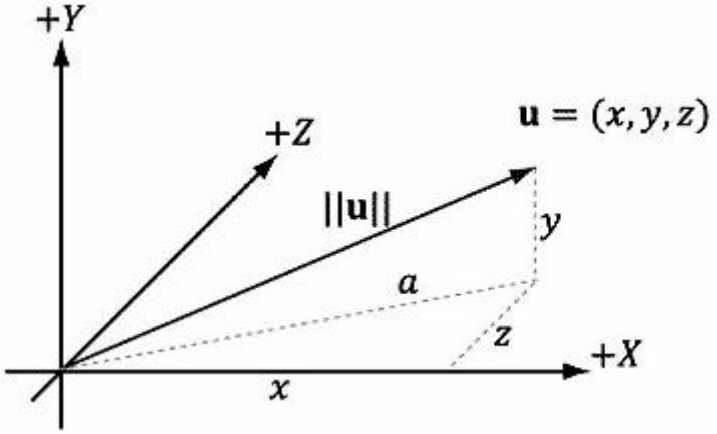



**图 1.8 通过运用两次毕达哥拉斯定理计算 3D 向量的长度。**

首先，我们来看 *xz* 平面上的三角形边 *x*、*z* 及斜边 *a*。由毕达哥拉斯定理可知 $a=\sqrt{x^2+z^2}$ 。现在来看三角形边 a、*y* 及斜边‖**u**‖。通过再次使用毕达哥拉斯定理，可以得到如下求模公式：

$$\|\mathbf{u}\|=\sqrt{y^2+a^2}=\sqrt{y^2+(\sqrt{x^2+z^2})^2}=\sqrt{x^2+y^2+z^2} \quad (1.1)$$

在某些应用中，我们不关心向量的长度， 只希望用向量来表示一个单纯的方向。对于这种只表示方向、不表示大小的向量，我们希望将其长度精确地设定为 1。当我们想要让一个向量具有单位长度时，我们说要对该向量进行规范化。我们将向量的每个分量除以该向量的模，得到规范化向量：

$$\hat{\mathbf{u}}=\frac{\mathbf{u}}{\|\mathbf{u}\|}=\left(\frac{x}{\|\mathbf{u}\|},\frac{y}{\|\mathbf{u}\|},\frac{z}{\|\mathbf{u}\|}\right) \quad (1.2)$$

要验证这个公式的正确性，只需计算 $\hat{\mathbf{u}}$ 的长度即可：

$$\|\hat{\mathbf{u}}\|=\sqrt{\left(\frac{x}{\|\mathbf{u}\|}\right)^2+\left(\frac{y}{\|\mathbf{u}\|}\right)^2+\left(\frac{z}{\|\mathbf{u}\|}\right)^2}=\frac{\sqrt{x^2+y^2+z^2}}{\sqrt{\|\mathbf{u}\|^2}}=\frac{\mathbf{u}}{\|\mathbf{u}\|}=1$$

因此，$\hat{\mathbf{u}}$ 确实是一个单位向量。

## 例 1.3

对向量 **v**= (−1,3,4) 进行规范化。我们计算 $\|\mathbf{v}\|=\sqrt{(-1)^2+3^2+4^2}=\sqrt{26}$ 。则，

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|} = \left( -\frac{1}{\sqrt{26}}, \frac{3}{\sqrt{26}} \frac{4}{\sqrt{26}} \right)$$

要验证 $\hat{\mathbf{v}}$ 是否为单位向量，只需计算它的长度：

$$\|\hat{\mathbf{v}}\| = \sqrt{\left( -\frac{1}{\sqrt{26}} \right)^2 + \left( \frac{3}{\sqrt{26}} \right)^2 + \left( \frac{4}{\sqrt{26}} \right)^2} = \sqrt{\frac{1}{26} + \frac{9}{26} + \frac{16}{26}} = \sqrt{1} = 1$$

# 1.3 点积

点积（dot product）是向量乘法的一种形式，它的计算结果是一个标量值；由于这一原因，有时也将点积称为标量积（scalar product）。设 **u**=（$u_x$,$u_y$,$u_z$），**v**=（$v_x$,$v_y$,$v_z$），则点积定义如下：

$$\mathbf{u}\cdot\mathbf{v}=u_xv_x+u_yv_y+u_zv_z \quad (1.3)$$

简言之，点积等于两个向量对应分量的乘积之和。

点积的定义不存在任何明显的几何含义。但是，使用余弦定理可以发现存在如下关系：

$$\mathbf{u}\cdot\mathbf{v}=\|\mathbf{u}\|\|\mathbf{v}\|\cos\theta \quad (1.4)$$

其中，$\theta$表示向量 **u** 和 **v** 之间的夹角，且 $0\leqslant\theta\leqslant\pi$（参见图 1.9）。公式 1.4 说明这两个向量的点积等于向量夹角的余弦值和向量模之间的乘积。在特殊情况下，如果 **u** 和 **v** 都是单位向量，那么 **u**·**v** 就等于它们之间夹角的余弦值（即，**u**·**v**=cos$\theta$）。

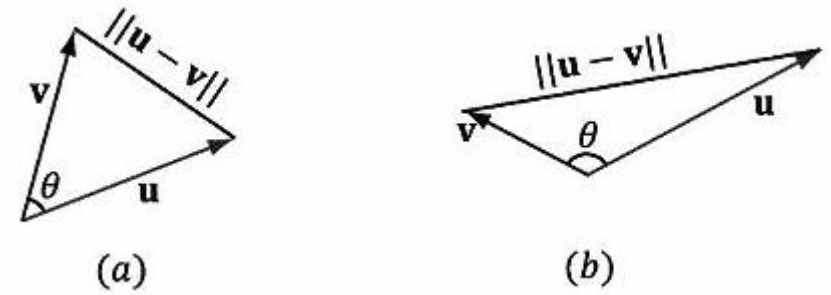



**图 1.9 在左图中，u、v 之间的夹角$\theta$为锐角。在右图中，u、v 之间的夹角$\theta$为钝角。记住，当我们提及两个向量之间的夹角时，通常指的是最小的角，也就是角度$\theta$，且 $0\leqslant\theta\leqslant\pi$。**

公式 1.4 提供了一些有用的点积的几何性质：

1．如果 **u**·**v**=0，则 **u**⊥**v**（即，向量相互垂直）。

2．如果 **u**·**v**＞0，则两个向量之间的夹角$\theta$小于 90º（即，向量形成一个锐角）。

3．如果 **u**·**v**＜0，则两个向量之间的夹角$\theta$大于 90º（即，向量形成一个钝角）。

**注意**：“相互垂直”也可称为“互成直角”。

**【例 1.4】**

设 **u**=（1, 2,3）、**v**=（−4, 0, −1）。求 **u** 和 **v** 之间的夹角。首先，我们要做如下计算：

$$\mathbf{u}\cdot\mathbf{v}=(1,2,3)\cdot(-4,0,-1)=-4-3=-7$$
$$\|\mathbf{u}\|=\sqrt{1^2+2^2+3^3}=\sqrt{14}$$
$$\|\mathbf{v}\|=\sqrt{(-4)^2+0^2+(-1)^2}=\sqrt{17}$$

现在，由公式 1.4 解得$\theta$为：

$$\cos\theta=\frac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|}=\frac{-7}{\sqrt{14}\sqrt{17}}$$
$$\theta=\cos^{-1}\frac{-7}{\sqrt{14}\sqrt{17}}\approx 117°$$

**【例 1.5】**

考虑图 1.10。给出 **v** 和单位向量 **n**，推导出一个使用点积求解向量 **p** 的公式。

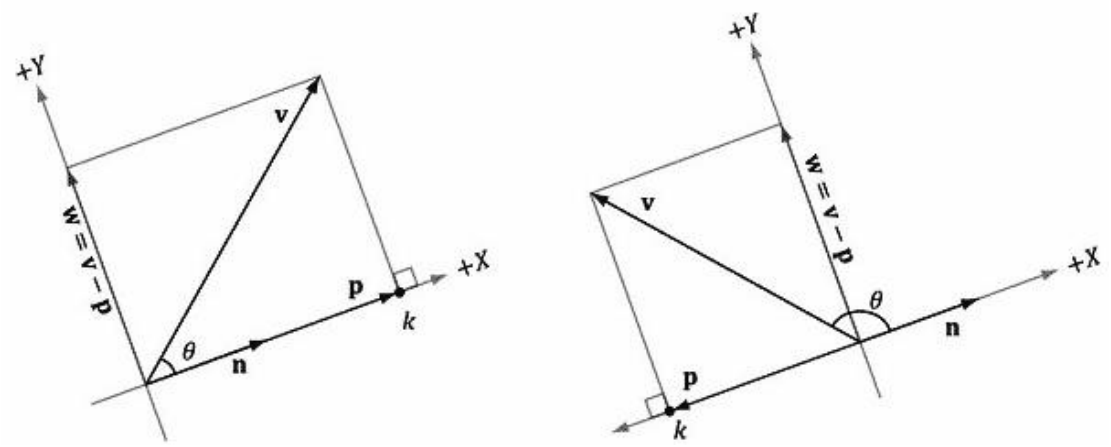



**图 1.10 v 在 n 上的正交投影。**

首先，从该图中可以看到标量 $k$ 可以使 $\mathbf{p}=k\mathbf{n}$；而且，由于我们已知 $\|\mathbf{n}\|=1$，所以有 $\|\mathbf{p}\|=\|k\mathbf{n}\|=|k|\|\mathbf{n}\|=|k|$。（注意，当且仅当 **p** 与 **n** 的方向相反时，$k$ 为负数。）我们使用三角函数，可以得出 $k=\|\mathbf{v}\|\cos\theta$；由此，$\mathbf{p}=k\mathbf{n}=(\|\mathbf{v}\|\cos\theta)\mathbf{n}$。不过，因为 **n** 是一个单位向量，所以我们可以用另一种方式进行表达：

$$\mathbf{p}=(\|\mathbf{v}\|\cos\theta)\mathbf{n}=(\|\mathbf{v}\|\cdot 1\cos\theta)\mathbf{n}=(\|\mathbf{v}\|\cdot\|\mathbf{n}\|\cos\theta)\mathbf{n}=(\mathbf{v}\cdot\mathbf{n})\mathbf{n}$$

请注意，这里的 $k=\mathbf{v}\cdot\mathbf{n}$，它说明了当 **n** 为单位向量时 $\mathbf{v}\cdot\mathbf{n}$ 的几何含义。我们将 **p** 称为 **v** 在 **n** 上的正交投影（orthogonal projection），并记为

$$\mathbf{p}=proj_{\mathbf{n}}(\mathbf{v})$$

如果我们把 **v** 理解为一个作用力，那么 **p** 可以被认为是 **v** 在方向 **n** 上的分力。同理，向量 $\mathbf{w}=perp_{\mathbf{n}}(\mathbf{v})=\mathbf{v}-\mathbf{p}$ 是与 **n** 垂直方向上的分力。可以看到 $\mathbf{v}=\mathbf{p}+\mathbf{w}$，这说明我们已经将向量 **v** 分解成了两个相互垂直的向量 **p** 和 **w**。

如果 n 不是一个单位向量，那我们可以对它进行规范化，使其保持单位长度。通过用单位向量 $\frac{\mathbf{n}}{\|\mathbf{n}\|}$ 来代替 **n**，可以得到一个更通用的投影公式：

$$\mathbf{p}=proj_{\mathbf{n}}(\mathbf{v})=(\mathbf{v}\cdot\frac{\mathbf{n}}{\|\mathbf{n}\|})\frac{\mathbf{n}}{\|\mathbf{n}\|}=\frac{(\mathbf{v}\cdot\mathbf{n})}{\|\mathbf{n}\|^2}\mathbf{n}$$

## 1.3.1 正交化

若一个向量集合{$\mathbf{v}_0$，……，$\mathbf{v}_{n-1}$}中的向量相互正交（即集合中的每一个向量与其他向量正交）并具有单位长度，我们将这个集合称之为规范化正交集。有时一组向量几乎是正交的，但又不完全是，一个常见的任务就是使其正交。在三维计算机图形中，开始时通常是一个规范化正交的向量集合，但由于计算精度问题，这个集合就会逐渐成为非规范化的了。我们主要关心的是 2D 和 3D 的情况下任何处理这个问题（即，含有两个和三个向量的情况）。

首先讨论简单的 2D 情况。假设有一组向量为{$\mathbf{v}_0$,$\mathbf{v}_1$}，我们要将它们正交到一个规范正交集{$\mathbf{w}_0$,$\mathbf{w}_1$}中，如图 1.11 所示。首先令 $\mathbf{w}_0=\mathbf{v}_0$，然后修改 $\mathbf{v}_1$ 使它与 $\mathbf{w}_0$ 垂直；这需要减去 $\mathbf{v}_1$ 向量在 $\mathbf{w}_0$ 上的投影：

$\mathbf{w}_1 = \mathbf{v}_1 - \text{proj}_{\mathbf{w}0}(\mathbf{v}_1)$

现在就有了一组互相垂直的向量$\{\mathbf{w}_0,\mathbf{w}_1\}$；最后需要规范化 $\mathbf{w}_0$ 和 $\mathbf{w}_1$ 才能构建规范化的正交集。

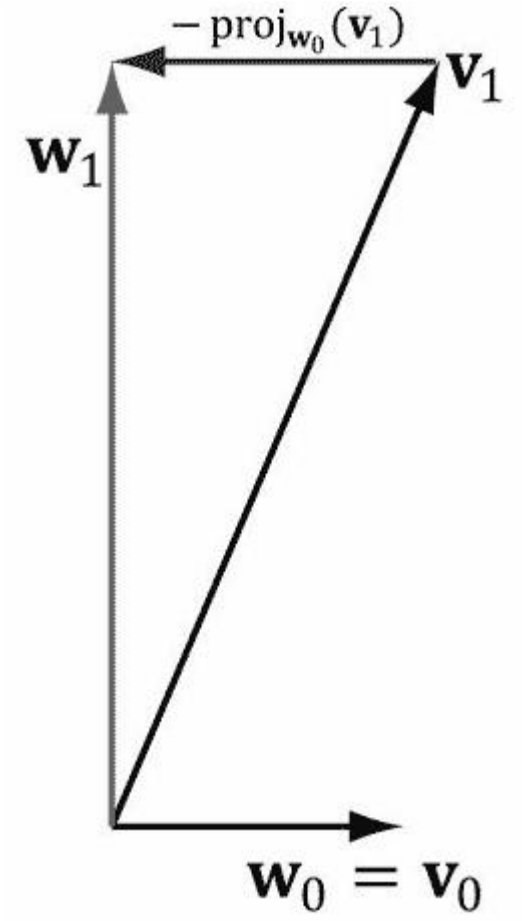



**图 1.11 2D 正交化**

处理 3D 情况的原理与 2D 相同，但需要更多的步骤。假设有一组向量$\{\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2\}$需要正交规范化到$\{\mathbf{w}_0,\mathbf{w}_1,\mathbf{w}_2\}$，如图 1.12 所示。首先令 $\mathbf{w}_0=\mathbf{v}_0$，然后修改 $\mathbf{v}_1$ 使之垂直于 $\mathbf{w}_0$；这需要从 $\mathbf{v}_1$ 中减去 $\mathbf{v}_1$ 在 $\mathbf{w}_0$ 方向上的投影：

$\mathbf{w}_1 = \mathbf{v}_1 - \text{proj}_{\mathbf{w}0}(\mathbf{v}_1)$

下一步需要令 $\mathbf{v}_2$ 同时垂直于 $\mathbf{w}_0$ 和 $\mathbf{w}_1$，这需要从 $\mathbf{v}_2$ 中减去 $\mathbf{v}_2$ 在 $\mathbf{w}_0$ 上的投影再减去 $\mathbf{v}_2$ 在 $\mathbf{w}_1$ 上的投影：

$\mathbf{w}_2 = \mathbf{v}_2 - \text{proj}_{\mathbf{w}0}(\mathbf{v}_2) - \text{proj}_{\mathbf{w}1}(\mathbf{v}_2)$

现在就有了一组互相垂直的向量$\{\mathbf{w}_0,\mathbf{w}_1,\mathbf{w}_2\}$；最后需要规范化 $\mathbf{w}_0$、$\mathbf{w}_1$ 和 $\mathbf{w}_2$ 才能构建规范化的正交集。

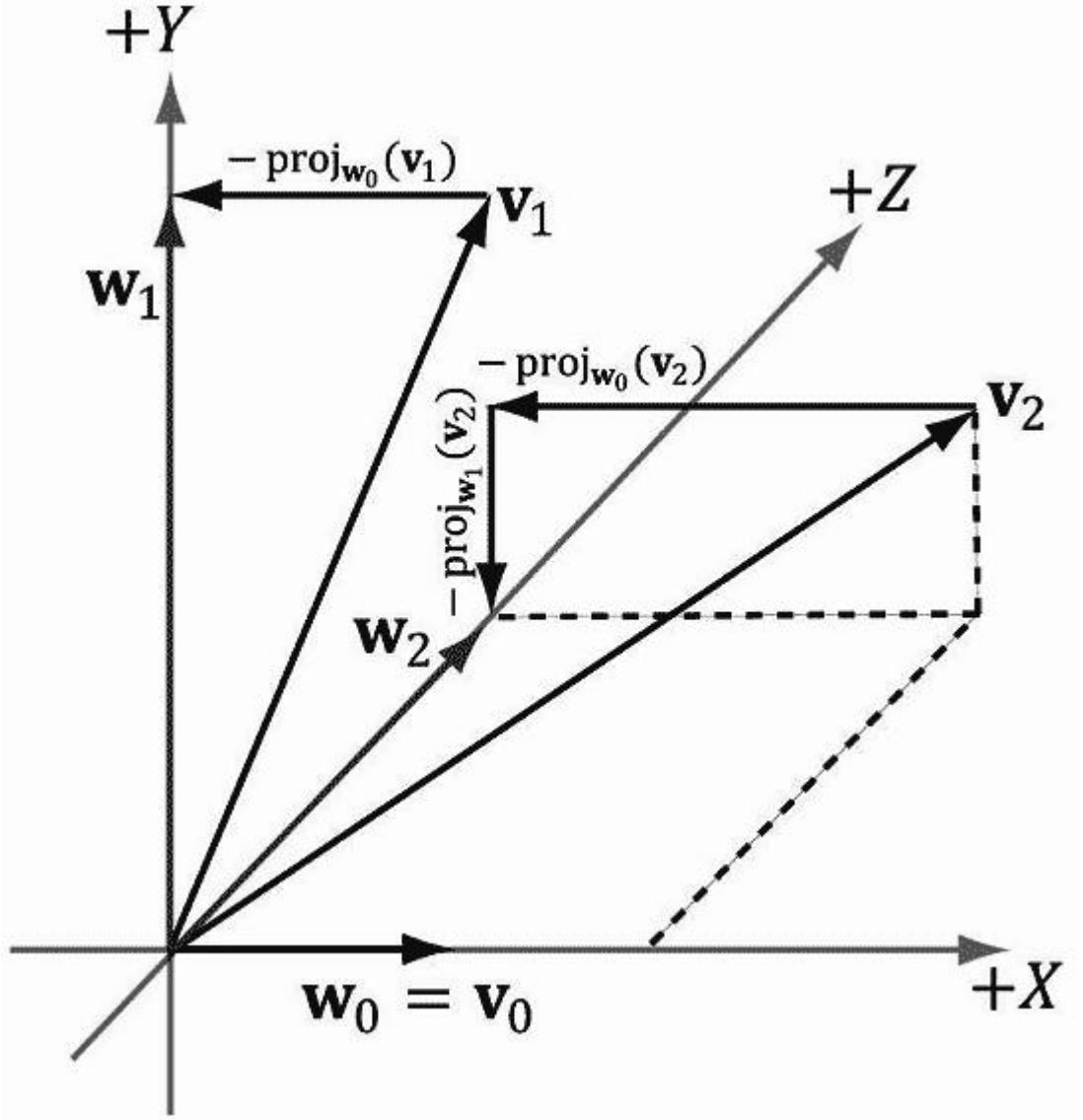



**图 1.12 3D 正交化**

要规范正交化任意数量的向量集$\{\mathbf{v}_0,\ldots,\mathbf{v}_{n-1}\}$，我们需要按照通常叫做 Gram-Schmidt 正交化的处理过程进行：

基本步骤：令 $\mathbf{w}_0=\mathbf{v}_0$

for $1 \le i \le n\text{-}1$，令 $\mathbf{w}_i = \mathbf{v}_i - \sum_{j=0}^{i-1} proj_{\mathbf{w}_j}(\mathbf{v}_i)$

规范化步骤：令 $\mathbf{w}_i = \frac{\mathbf{w}_i}{\|\mathbf{w}_i\|}$

原理与上面是类似的，当选取一个向量 $\mathbf{v}_i$ 将它添加到规范化的正交集时，我们需要减去这个向量在正交集中其他向量（$\mathbf{w}_0,\mathbf{w}_1,\ldots,\mathbf{w}_{i-1}$）上的投影，这样可以确保这个新添的向量与正交集中的其他向量垂直。

# 1.4 叉积

叉积（cross product）是向量数学定义的第二种乘法形式。它与点积不同，点积的计算结果是一个标量，而叉积的计算结果是一个向量；另外，叉积只能用于 3D 向量（2D 向量没有叉积）。通过对两个 3D 向量 **u** 和 **v** 计算叉积，可以得到第 3 个向量 **w**，该向量同时垂直于 **u** 和 **v**。也就是说，**w** 即垂直于 **u**，**w** 也垂直于 **v**（参见图 1.13）。设 **u**=（$u_x$,$u_y$,$u_z$），**v**=（$v_x$,$v_y$,$v_z$），则叉积为：

$$\mathbf{w}=\mathbf{u}\times\mathbf{v}=(u_yv_z-u_zv_y,u_zv_x-u_xv_z,u_xv_y-u_yv_x) \qquad (1.5)$$

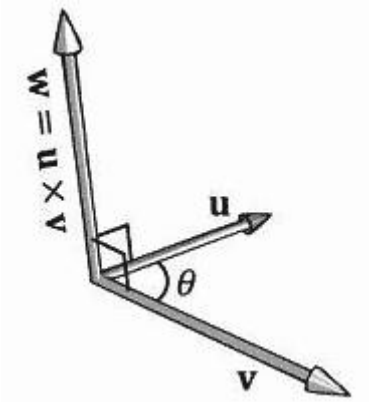



**图 1.13 通过为两个 3D 向量 u 和 v 计算叉积，可以得到第 3 个向量 w，该向量同时垂直于 u 和 v。如果读者抬起左手，将拇指之外的其他 4 个手指指向第一个向量 u 的方向，然后朝着 v 的方向沿角度 0≤$\theta$≤$\pi$弯曲手指，此时拇指所指的方向即为 w=u×v 的方向；这叫做左手拇指定则（left-hand-thumb rule）。**

注意：如果你处理的是一个右手坐标系，则需要使用右手拇指定则（right-hand-thumb rule）：如果抬起右手，将拇指之外的其他 4 个手指指向第一个向量 **u** 的方向，然后朝着 **v** 的方向沿角度 0≤$\theta$≤$\pi$弯曲手指，此时拇指所指的方向即为 **w=u×v** 的方向。

【例 1.6】

设 u= (2, 1,3)、v= (2, 0,0)。计算 **w=u×v** 和 **z=v×u**，并验证 **w** 既垂直于 **u**，也垂直于 **v**。由公式 1.5 可得，

**w=u×v**

= (2, 1,3)×(2, 0,0)

= (1 · 0 − 3 · 0,3 · 2 − 2 · 0,2 · 0 − 1 · 2)

= (0, 6, −2)

和

**z=v×u**

= (2, 0,0) × (2, 1,3)

= (0 · 3 − 0 · 1,0 · 2 − 2 · 3,2 · 1 − 0 · 2)

= (0, −6,2)

该结果说明 **u×v≠v×u**。也就是，叉积不支持交换律。实际上，它可以表达为 **u×v= −v×u**。读者可以通过左手拇指定则来判断由这个叉积得出的向量。如果你从第 1 个向量朝着第 2 个向量的方向卷曲手指时（通常选择角度最小的路径），你的拇指会指向最终得到的向量的方向，如图 1.13 所示。

为了说明 **w** 既垂直于 **u**，也垂直于 **v**，我们回顾 1.3 节的内容：如果 **u·v=0**，则 **u⊥v**（即，

向量相互垂直）。因为

$\mathbf{w}\cdot\mathbf{u}= (0, 6, -2) \cdot (2, 1,3) =0 \cdot 2 +6 \cdot 1+ (-2) \cdot 3 =0$

和

$\mathbf{w}\cdot\mathbf{v}= (0, 6, -2) \cdot (2, 0,0) =0 \cdot 2 +6 \cdot 0+ (-2) \cdot 0 =0$

我们得出结论：**w** 既垂直于 **u**，也垂直于 **v**。

## 1.4.1 2D 伪叉积

叉积可以计算垂直于给定两个 3D 向量的向量。在 2D 的情况中，并不存在这种情况，但我们常常要求出垂直于给定 2D 向量 $\mathbf{u} = (u_x,u_y)$的向量 **v**。图 1.14 展示了这种操作的几何图景，从图中可以看出 $\mathbf{v} = (-u_y, u_x)$。数学证明很简单：

$\mathbf{u}\cdot\mathbf{v}= (u_x,u_y)\cdot(-u_y,u_x)= -u_xu_y+ u_yu_x= 0$

所以 **u**⊥**v**。而 $\mathbf{u}\cdot -\mathbf{v}=u_xu_y+ u_y(-u_x) =0$，也为零，所以还能得出结论：**u**⊥**-v**。

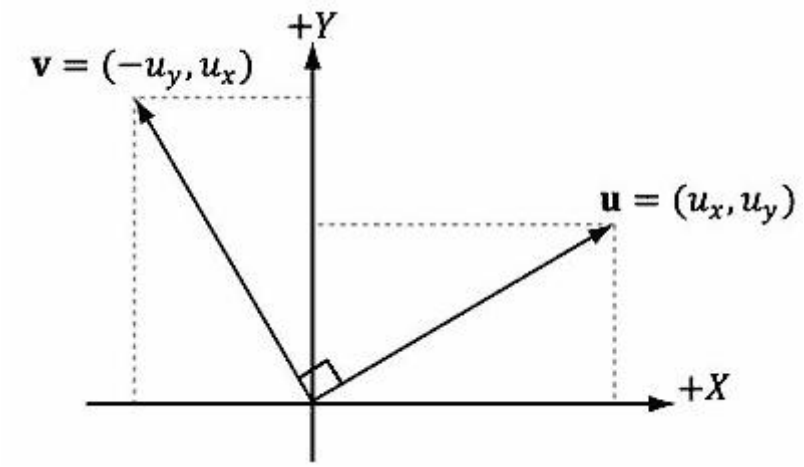



**图 1.14 u 向量的 2D 伪叉积为一个垂直于它的向量 v**

## 1.4.2 使用叉积进行正交规范化

在 1.3.1 节中，我们介绍了一种正交化一组向量的处理方法。对 3D 的情况来说，还可以使用叉积对一组向量（这些向量近似正交，但由于数值累积精度的误差，会变得不再正交）进行正交规范化操作。可参见图 1.15 理解这个过程的几何图景：

1．令 $\mathbf{w}_0 = \dfrac{\mathbf{v}_0}{\|\mathbf{v}_0\|}$

2．令 $\mathbf{w}_2 = \dfrac{\mathbf{w}_0 \times \mathbf{v}_1}{\|\mathbf{w}_0 \times \mathbf{v}_1\|}$

3．令 $\mathbf{w}_1 = \mathbf{w}_2 \times \mathbf{w}_0$。由后面的练习 14 可知，因为 $\mathbf{w}_2 \perp \mathbf{w}_0$ 且$\|\mathbf{w}_2\|=\|\mathbf{w}_0\|=1$，所以$\|\mathbf{w}_2 \times \mathbf{w}_0\|=1$，这样我们就无需进行规范化操作了。

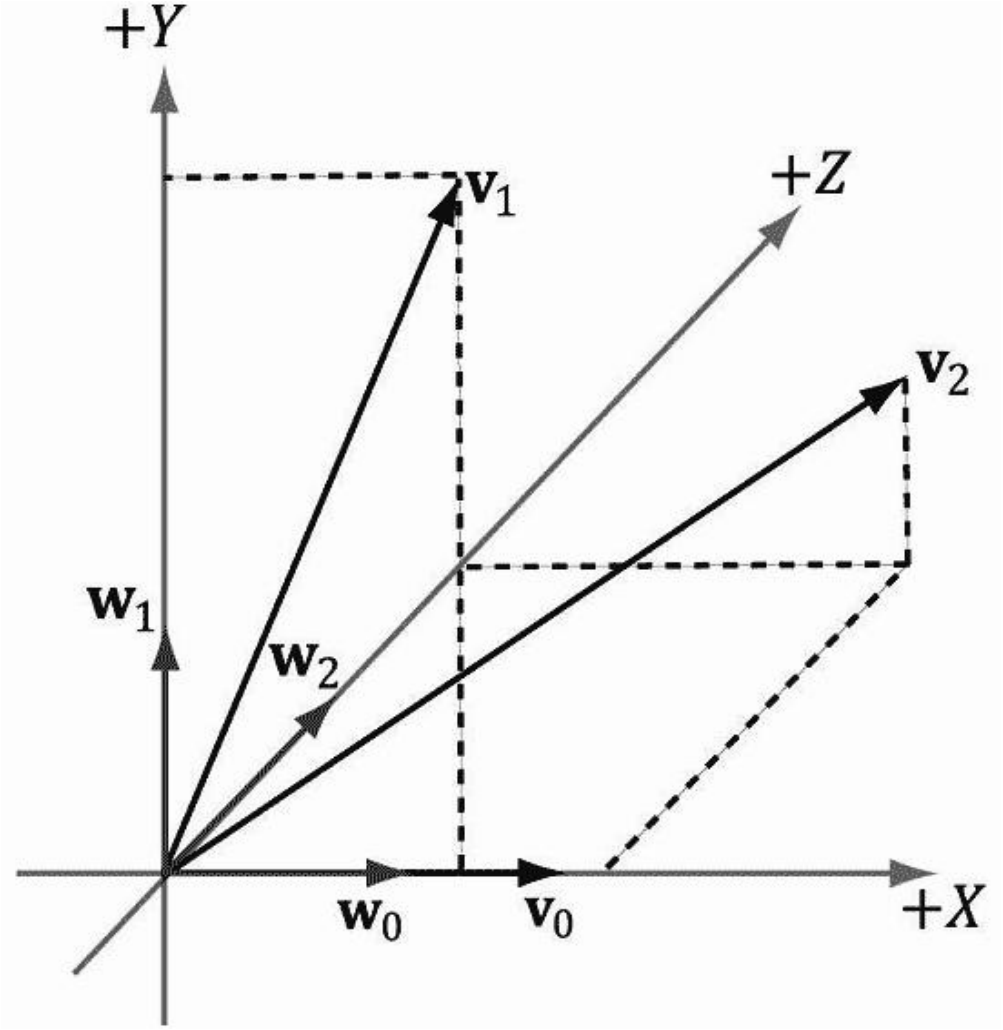



**图 1.15 使用叉积进行 3D 正交化**

至此，完成了向量集{$\mathbf{w}_0$,$\mathbf{w}_1$,$\mathbf{w}_2$}的正交规范化处理。

**注意**：在前面的示例中，我们首先令 $\mathbf{w}_0 = \frac{\mathbf{v}_0}{\|\mathbf{v}_0\|}$，表示从 $\mathbf{v}_0$ 变化到 $\mathbf{w}_0$ 并没有改变向量的方向，只是改变了大小。但是，$\mathbf{w}_1$ 和 $\mathbf{w}_2$ 的方向与 $\mathbf{v}_1$ 和 $\mathbf{v}_2$ 的方向并不相同。根据应用程序的需要，选择哪个向量不改变方向可能会很重要。例如，本书的后面我们将会使用三个正交向量{$\mathbf{v}_0$,$\mathbf{v}_1$,$\mathbf{v}_2$}代表相机的朝向，其中第三个向量 $\mathbf{v}_2$ 表示相机的观察方向。当正交规范化这三个向量时，我们常常不想改变观察的方向，因此，我们会首先使用上面的算法处理 $\mathbf{v}_2$，然后修改 $\mathbf{v}_0$ 和 $\mathbf{v}_1$ 生成正交向量。

# 1.5 点

到目前为止，我们讨论的是与位置无关的向量。而我们在 3D 程序中需要描述坐标位置；比如，3D 几何体的位置和 3D 虚拟摄像机的位置。相对于一个坐标系，我们可以使用在标准位置上的向量（参见图 1.16）来表示空间中的 3D 位置；我们将它称为位置向量（position vector）。在这里，向量末端的位置是唯一需要关注的特性，而方向和大小都无关紧要。我们会交替使用术语“位置向量”和“点”，因为位置向量表示的就是一个点。

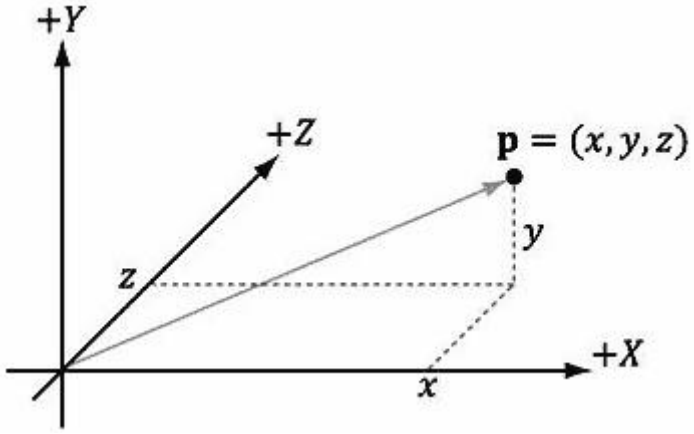



**图 1.16 从原点延伸到点的位置向量，它足以描述相对于坐标系的点的位置。**

使用向量来表示点的一个好处是可以在代码中使用向量运算，虽然向量运算对点来说没有实际意义；比如，在几何学中，两点相加有什么意义？不过有一些运算确实可以被扩展为点运算。比如，两点之差 **q**−**p** 可以表示从 **p** 到 **q** 的向量。点 **p** 与向量 **v** 相加得到点 **q**，可以认为 **q** 是 **v** 对 **p** 进行的平移。由于使用向量可以很方便地表示相对于坐标系的点，所以我们不必为单独设计一套针对于点的运算，只需要借助于前面讨论过的向量代数框架就可以处理它们（参见图 1.17）。

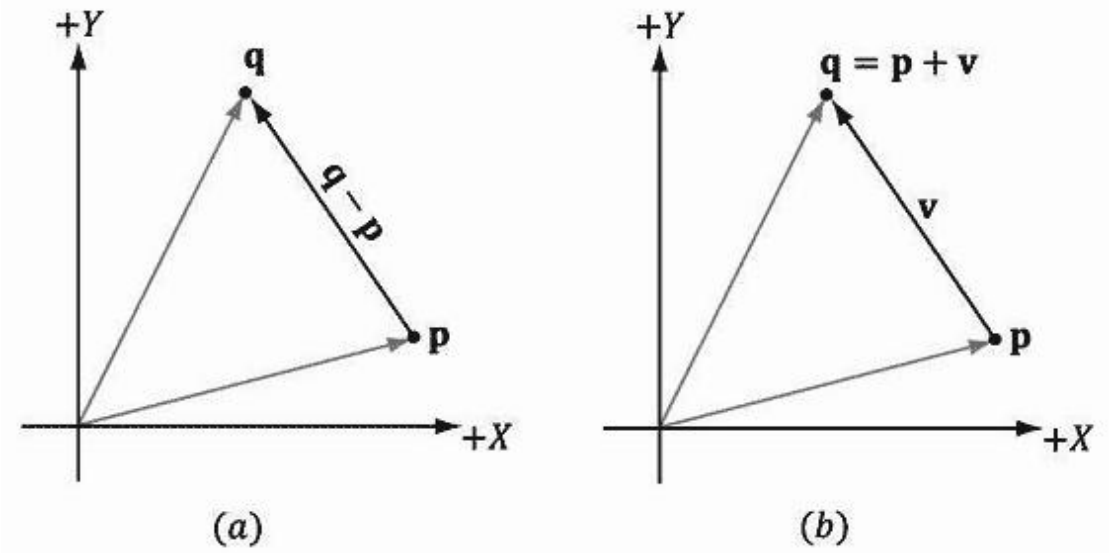



**图 1.17 （a）两点之差 q−p 可以表示从 p 到 q 的向量。（b）点 p 与向量 v 相加得到点 q，可以认为 q 是 v 对 p 进行的平移。**

**注意**：其实在几何学中有一种非常重要的方法叫做仿射组合（affine combination），它用于对点进行特殊的求和运算，就像是对点求加权平均值一样。

# 2.1 定义

在 3D 计算机绘图中，我们用矩阵（matrix）来紧凑地描述几何变换，比如缩放、旋转和平移，并将点或向量的坐标从一种坐标系转换到另一种坐标系。本章探讨了矩阵代数。

**学习目标：**

1．了解矩阵及矩阵运算。

2．了解如何将向量-矩阵乘法视为一个线性组合。

3．学习单位矩阵、转置矩阵、行列式和逆矩阵。

4．熟悉 XNA 库中的用于矩阵代数的类和函数。

一个 $m\times$n 矩阵 **M** 是一个 $m$ 行、$n$ 列的矩形实数数组。行和列的数量指定了矩阵的维数。矩阵中的数值称为元素。我们使用行和列组成的双下标 $M_{ij}$ 来标识矩阵元素，其中，第 1 个下标指定了元素的所在的行，第 2 个下标指定了元素所在的列。

## 例 2.1

考虑如下矩阵：

$$A=\begin{bmatrix}3.5 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0.5 & 0\\ 2 & -5 & \sqrt{2} & 1\end{bmatrix}\quad \mathbf{B}=\begin{bmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\\ B_{31} & B_{32}\end{bmatrix}\quad \mathbf{u}=\begin{bmatrix}u_1,u_2,u_3\end{bmatrix}\quad \mathbf{v}=\begin{bmatrix}1\\ 2\\ \sqrt{3}\\ \pi\end{bmatrix}$$

1．矩阵 **A** 是一个 4×4 矩阵；矩阵 **B** 是一个 3×2 矩阵；矩阵 **u** 是一个 1×3 矩阵；矩阵 **v** 是一个 4×1 矩阵。

2．$A_{42}$=-5 表示矩阵 **A** 的第 4 行、第 2 列的元素。$B_{21}$ 表示矩阵 **B** 的第 2 行、第 1 列的元素。

3．矩阵 **u** 和 **v** 是特殊矩阵，因为它们只包含一行或一列。我们有时将这种矩阵称为行向量或列向量，因为它们可以用矩阵的形式表示一个向量（例如，可以随意地将$(x,y,z)$和$[x,y,z]$互换使用，这两种记法都可用于表示向量）。注意，对于行向量和列向量，不必使用双下标表示矩阵元素——只用一个下标即可。

有时，我们希望将矩阵的每一行视为一个向量。例如，我们可以这样写：

$$\begin{bmatrix}A_{11} & A_{12} & A_{13}\\ A_{21} & A_{22} & A_{23}\\ A_{31} & A_{32} & A_{33}\end{bmatrix}=\begin{bmatrix}\leftarrow \mathbf{A}_{1,*} \rightarrow\\ \leftarrow \mathbf{A}_{2,*} \rightarrow\\ \leftarrow \mathbf{A}_{3,*} \rightarrow\end{bmatrix}$$

其中，$\mathbf{A}_{1,*}=[A_{11},A_{12},A_{13}]$、$\mathbf{A}_{2,*}=[A_{21},A_{22},A_{23}]$、$\mathbf{A}_{3,*}=[A_{31},A_{32},A_{33}]$。在这种记法中，第 1 个索引指定行标，第 2 个索引以星号（*）表示我们引用的是整个行向量。同样，我们也可以用这种方法来表示矩阵的列：

$$\begin{bmatrix}A_{11} & A_{12} & A_{13}\\ A_{21} & A_{22} & A_{23}\\ A_{31} & A_{32} & A_{33}\end{bmatrix}=\begin{bmatrix}\uparrow & \uparrow & \uparrow\\ A_{*,1} & A_{*,2} & A_{*,3}\\ \downarrow & \downarrow & \downarrow\end{bmatrix}$$

其中

$$\mathbf{A}_{*,1}=\begin{bmatrix}A_{11}\\A_{21}\\A_{31}\end{bmatrix},\mathbf{A}_{*,2}=\begin{bmatrix}A_{12}\\A_{22}\\A_{32}\end{bmatrix},\mathbf{A}_{*,3}=\begin{bmatrix}A_{13}\\A_{23}\\A_{33}\end{bmatrix}$$

在这种记法中，第 2 个索引指定列标，第 1 个索引以星号（*）表示我们引用的是整个列向量。

我们现在对矩阵上的判等运算、加法运算、标量乘法运算和减法运算做以定义：

1．当且仅当两个矩阵的对应元素相等时，这两个矩阵相等；这两个矩阵必须具有相同的行数和列数，才能进行比较。

2．矩阵加法是对两个矩阵的对应元素相加；只有在两个矩阵的行数和列数相同的情况下，矩阵加法才有意义。

3．矩阵的标量乘法是将一个标量和矩阵中的每个元素相乘。

4．矩阵减法可以由矩阵加法和标量乘法表示。也就是，$\mathbf{A}-\mathbf{B}=\mathbf{A}+(-1\cdot\mathbf{B})=\mathbf{A}+(-\mathbf{B})$。

# 例 2.2

设

$$\mathbf{A}=\begin{bmatrix}1&5\\-2&3\end{bmatrix},\mathbf{B}=\begin{bmatrix}6&2\\5&-8\end{bmatrix},\mathbf{C}=\begin{bmatrix}1&5\\-2&3\end{bmatrix},\mathbf{D}=\begin{bmatrix}2&1&-3\\-6&3&0\end{bmatrix}$$

则，

（ i ） $\mathbf{A}+\mathbf{B}=\begin{bmatrix}1&5\\-2&3\end{bmatrix}+\begin{bmatrix}6&2\\5&-8\end{bmatrix}=\begin{bmatrix}1+6&5+2\\-2+5&3+(-8)\end{bmatrix}=\begin{bmatrix}7&7\\3&-5\end{bmatrix}$

（ ii ） $\mathbf{A}=\mathbf{C}$

（iii） $3\mathbf{D}=3\begin{bmatrix}2&1&-3\\-6&3&0\end{bmatrix}=\begin{bmatrix}3(2)&3(1)&3(-3)\\3(-6)&3(3)&3(0)\end{bmatrix}=\begin{bmatrix}6&3&-9\\-18&9&0\end{bmatrix}$

（iv） $\mathbf{A}-\mathbf{B}=\begin{bmatrix}1&5\\-2&3\end{bmatrix}-\begin{bmatrix}6&2\\5&-8\end{bmatrix}=\begin{bmatrix}1-6&5-2\\-2-5&3-(-8)\end{bmatrix}=\begin{bmatrix}-5&3\\-7&11\end{bmatrix}$

因为矩阵加法和标量乘法是逐元素进行的，所以矩阵也继承了实数的加法和标量乘法的性质：

1．$\mathbf{A}+\mathbf{B}=\mathbf{B}+\mathbf{A}$

2．$(\mathbf{A}+\mathbf{B})+\mathbf{C}=\mathbf{A}+(\mathbf{B}+\mathbf{C})$

3．$r(\mathbf{A}+\mathbf{B})=r\mathbf{A}+r\mathbf{B}$

4．$(r+s)\mathbf{A}=r\mathbf{A}+s\mathbf{A}$

# 2.2 矩阵乘法

本节讲解矩阵乘法。我们将在第 3 章中看到，矩阵乘法用于实现点和向量的变换，并通过矩阵乘法将一系列的变换组合在一起。

## 2.2.1 定义

假设 **A** 是一个 $m \times n$ 矩阵，**B** 是一个 $n \times p$ 矩阵，乘积 **AB** 由 **C** 表示，则 **C** 是一个 $m \times p$ 矩阵，其中结果 **C** 的第 $ij$ 个元素的值等于 **A** 的第 $i$ 个行向量和 **B** 的第 $j$ 个列向量的点积，也就是，

$$\mathbf{C}_{ij}=\mathbf{A}_{i,*} \cdot \mathbf{B}_{*,j} \qquad (2.1)$$

注意，矩阵 **A** 的列数必须与矩阵 **B** 的行数相同，只有这样才能计算矩阵乘积 **C**，也就是说，**A** 中的行向量的维数必须与 **B** 中的列向量的维数相同。如果维数不同，那么公式 2.1 中的点积就没有意义。

## 例 2.3

设

$$\mathbf{A}=\begin{bmatrix} 1 & 5 \\ -2 & 3 \end{bmatrix}, \mathbf{B}=\begin{bmatrix} 2 & -6 \\ 1 & 3 \\ -3 & 0 \end{bmatrix}$$

则，无法计算乘积 **AB**，因为 **A** 中的行向量的维数是 2，而 **B** 中的列向量的维数是 3。我们无法计算 **A** 的第 1 个行向量与 **B** 的第 1 个列向量的点积，因为一个 2D 向量是无法与一个 3D 向量计算点积的。

## 例 2.4

设

$$\mathbf{A}=\begin{bmatrix} -1 & 5 & -4 \\ 3 & 2 & 1 \end{bmatrix}, \mathbf{B}=\begin{bmatrix} 2 & 1 & 0 \\ 0 & -2 & 1 \\ -1 & 2 & 3 \end{bmatrix}$$

首先我们要指出的是可以计算乘积 AB（是一个 2×3 矩阵），因为 A 的列数与 B 的行数相等。运用公式 2.1 可得：

$$\mathbf{AB}=\begin{bmatrix}-1 & 5 & -4\\ 3 & 2 & 1\end{bmatrix}\begin{bmatrix}2 & 1 & 0\\ 0 & -2 & 1\\ -1 & 2 & 3\end{bmatrix}$$
$$=\begin{bmatrix}(-1,5,-4)\cdot(2,0,-1) & (-1,5,-4)\cdot(1,-2,2) & (-1,5,-4)\cdot(0,1,3)\\ (3,2,1)\cdot(2,0,-1) & (3,2,1)\cdot(1,-2,2) & (3,2,1)\cdot(0,1,3)\end{bmatrix}$$
$$=\begin{bmatrix}2 & -19 & -7\\ 5 & 1 & 5\end{bmatrix}$$

注意，在个例子中我们无法计算 **BA** 的乘积，因为 **B** 的列数与 **A** 的行数不相等。这说明矩阵乘法通常不满足交换律；也就是 **AB**≠**BA**。

## 2.2.2 向量-矩阵乘法

考虑下面的向量-矩阵乘法：

$$\mathbf{uA}=\begin{bmatrix}x,y,z\end{bmatrix}\begin{bmatrix}A_{11} & A_{12} & A_{13}\\ A_{21} & A_{21} & A_{23}\\ A_{31} & A_{32} & A_{33}\end{bmatrix}=\begin{bmatrix}x,y,z\end{bmatrix}\begin{bmatrix}\uparrow & \uparrow & \uparrow\\ \mathbf{A}_{*,1} & \mathbf{A}_{*,2} & \mathbf{A}_{*,3}\\ \downarrow & \downarrow & \downarrow\end{bmatrix}$$

注意，这里 **uA** 的计算结果是一个 1×3 行向量。运用公式 2.1 可得：

$$\begin{aligned}\mathbf{uA}&=\begin{bmatrix}\mathbf{u}\cdot\mathbf{A}_{*,1} & \mathbf{u}\cdot\mathbf{A}_{*,2} & \mathbf{u}\cdot\mathbf{A}_{*,3}\end{bmatrix}\\ &=\begin{bmatrix}xA_{11}+yA_{21}+zA_{31}, xA_{12}+yA_{22}+zA_{32}, xA_{13}+yA_{23}+zA_{33}\end{bmatrix}\\ &=\begin{bmatrix}xA_{11}, xA_{12}, xA_{13}\end{bmatrix}+\begin{bmatrix}yA_{21}, yA_{22}, yA_{23}\end{bmatrix}+\begin{bmatrix}zA_{31}, zA_{32}, zA_{33}\end{bmatrix}\\ &=x\begin{bmatrix}A_{11}, A_{12}, A_{13}\end{bmatrix}+y\begin{bmatrix}A_{21}, A_{22}, A_{23}\end{bmatrix}+z\begin{bmatrix}A_{31}, A_{32}, A_{33}\end{bmatrix}\\ &=x\mathbf{A}_{1,*}+y\mathbf{A}_{2,*}+z\mathbf{A}_{3,*}\end{aligned}$$

因此，

$$\mathbf{uA}=x\mathbf{A}_{1,*}+y\mathbf{A}_{2,*}+z\mathbf{A}_{3,*}\quad(2.2)$$

公式 2.2 是一个常用的线性组合，它说明向量-矩阵的乘积 **uA** 等于向量 **u** 给出的标量系数 $x$、$y$、$z$ 与矩阵 **A** 的每个行向量的线性组合。注意，虽然我们这里给出的例子是一个 1×3 行向量和一个 3×3 矩阵，但是这个计算方法是通用的。也就是，对于一个 1×$n$ 行向量 **u** 和一个 $n$×$m$ 矩阵 **A**，乘积 **uA** 等于 **u** 给出的标量系数与 **A** 中的每个行向量的线性组合：

$$\begin{bmatrix}u_1, & ,u_n\end{bmatrix}\begin{bmatrix}A_{11} & & A_{1m}\\ & & \\ A_{n1} & & A_{nm}\end{bmatrix}=u_1A_{1,*}+\quad+u_nA_{n,*}\quad(2.3)$$

## 2.2.3 结合律

矩阵乘法具有一些有用的代数特性。例如，可以将矩阵乘法分配给每个加法分量：**A(B+C)=AB+AC**、**(A+B)C=AC+BC**。有时我们会使用矩阵乘法的结合律来改变相乘矩阵的

计算顺序：

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

# 2.3 转置矩阵

对一个矩阵的行和列进行互换，即可得到该矩阵的转置（transpose）矩阵。一个 $m\times n$ 矩阵的转置矩阵是一个 $n\times m$ 矩阵。我们将矩阵 **M** 的转置矩阵记作 $\mathbf{M}^T$。

## 例 2.5

求以下 3 个矩阵的转置矩阵：

$$\mathbf{A}=\begin{bmatrix}2 & -1 & 8\\ 3 & 6 & -4\end{bmatrix},\mathbf{B}=\begin{bmatrix}a & b & c\\ d & e & f\\ g & h & i\end{bmatrix},\mathbf{C}=\begin{bmatrix}1\\ 2\\ 3\\ 4\end{bmatrix}$$

对行和列进行互换，即可得到转置矩阵，因此

$$\mathbf{A}^T=\begin{bmatrix}2 & 3\\ -1 & 6\\ 8 & -4\end{bmatrix},\mathbf{B}^T=\begin{bmatrix}a & d & g\\ b & e & h\\ c & f & i\end{bmatrix},\mathbf{C}^T=\begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix}$$

矩阵转置有以下有用的特点：

1．$(\mathbf{A}+\mathbf{B})^T=\mathbf{A}^T+B^T$

2．$(c\mathbf{A})^T=c\mathbf{A}^T$

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

4．$(\mathbf{A}^T)^T=\mathbf{A}$

5．$(\mathbf{A}^{-1})^T=(\mathbf{A}^T)^{-1}$

# 2.4 单位矩阵

有一种特殊的矩阵称为单位矩阵（identity matrix）。单位矩阵是一个正方形矩阵，它除了对角线上的元素为 1 外，其他元素均为 0。

例如，下面是 2×2、3×3 和 4×4 单位矩阵。

$$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix},\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix},\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

单位矩阵的作用相当于一个乘法单位；也就是，如果 **A** 是一个 $m\times n$ 矩阵，**B** 是一个 $n\times p$ 矩阵，**I** 是 $n\times n$ 单位矩阵，那么

**AI=A** 且 **IB=B**

换句话说，将一个矩阵与单位矩阵相乘，得到结果不会发生改变。单位矩阵可以被看成是矩阵中的数字 1。如果 **M** 是一个正方形矩阵，那么 **M** 与单位矩阵之间的相乘次序可以交换：

**MI=IM=M**

## 例 2.6

设 $\mathbf{M}=\begin{bmatrix}1 & 2\\ 0 & 4\end{bmatrix}$，$\mathbf{I}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$。证明 **MI=IM=M**。

运用公式 2.1 可得：

$$\mathbf{MI}=\begin{bmatrix}1 & 2\\ 0 & 4\end{bmatrix}\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}=\begin{bmatrix}(1,2)\cdot(1,0) & (1,2)\cdot(0,1)\\ (0,4)\cdot(1,0) & (0,4)\cdot(0,1)\end{bmatrix}=\begin{bmatrix}1 & 2\\ 0 & 4\end{bmatrix}$$

和

$$\mathbf{IM}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}1 & 2\\ 0 & 4\end{bmatrix}=\begin{bmatrix}(1,0)\cdot(1,0) & (1,0)\cdot(2,4)\\ (0,1)\cdot(1,0) & (0,1)\cdot(2,4)\end{bmatrix}=\begin{bmatrix}1 & 2\\ 0 & 4\end{bmatrix}$$

因此，**MI=IM=M** 为真。

## 例 2.7

设 **u**=[−1,2]，$\mathbf{I}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$。证明 **uI=u**。

运用公式 2.1 可得：

$$\mathbf{uI}=\begin{bmatrix}-1 & 2\end{bmatrix}\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}=\begin{bmatrix}(-1,2)\cdot(1,0) & (-1,2)\cdot(0,1)\end{bmatrix}=\begin{bmatrix}-1 & 2\end{bmatrix}$$

注意，我们无法计算 **Iu** 的乘积，因为它在矩阵乘法中没有意义。

# 2.5 矩阵行列式

矩阵行列式是一个特殊的函数，它可以将一个正方矩阵映射为一个实数，正方矩阵 **A** 的行列式通常用符号 det**A** 表示。行列式描述的是一个线性变换对“体积”所造成的影响。此外，当线性方程组对应的行列式不为零时，由克莱姆法则，可以直接以行列式的形式写出方程组的解。但是，我们使用行列式的主要目的是为了用它得到逆矩阵（2.7 节的主题）。此外，还可以证明：当且仅当正方矩阵 **A** 的行列式 det**A**≠0 时，它才是可逆的。这个结论非常有用，因为它提供了一个判断矩阵是否可逆的计算工具。在对行列式下定义之前，我们首先介绍余子式的概念。

## 2.5.1 余子式

给定一个 $n\times n$ 矩阵 **A**，余子式 $\overline{\mathbf{A}}_{ij}$ 是指删除了第 $i$ 行和第 $j$ 列后的$(n-1)\times(n-1)$矩阵。

## 例 2.8

找到下列矩阵的余子式 $\overline{\mathbf{A}}_{11}$、$\overline{\mathbf{A}}_{22}$ 和 $\overline{\mathbf{A}}_{13}$：

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

删除第 1 行和第 1 列可得：

$$\overline{\mathbf{A}}_{11}=\begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix}$$

删除第 2 行和第 2 列可得：

$$\overline{\mathbf{A}}_{22}=\begin{bmatrix} A_{11} & A_{13} \\ A_{31} & A_{33} \end{bmatrix}$$

删除第 1 行和第 3 列可得：

$$\overline{\mathbf{A}}_{13}=\begin{bmatrix} A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix}$$

## 2.5.2 定义

行列式是递归定义的；例如，4×4 矩阵的行列式是以 3×3 矩阵的形式定义的，3×3 矩阵的定义式是以 2×2 矩阵的形式定义的，2×2 矩阵的定义式是以 1×1 矩阵的形式定义的（1×1 矩阵 **A**=[$A_{11}$]可简单地表示为 det[A11] = A11）。

若 **A** 为一个 $n\times n$ 矩阵，在 $n>1$ 时我们可以定义：

$$\det \mathbf{A}=\sum_{j=1}^{n} A_{1j}(-1)^{1+j}\det \overline{\mathbf{A}}_{1j} \qquad \text{（公式 2.4）}$$

回忆一下 2×2 矩阵的余子式 $\overline{\mathbf{A}}_{ij}$ 的定义，可以得到以下式子：

$$\det\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}=A_{11}\det\begin{bmatrix}A_{22}\end{bmatrix}-A_{12}\det\begin{bmatrix}A_{21}\end{bmatrix}=A_{11}A_{22}-A_{12}A_{21}$$

若是 3×3 矩阵，则公式如下：

$$\det\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}=A_{11}\det\begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix}-A_{12}\det\begin{bmatrix} A_{21} & A_{23} \\ A_{31} & A_{33} \end{bmatrix}+A_{13}\det\begin{bmatrix} A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix}$$

换成 4×4 矩阵，公式变为：

$$\det\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}$$

$$=A_{11}\det\begin{bmatrix} A_{22} & A_{23} & A_{24} \\ A_{31} & A_{33} & A_{34} \\ A_{42} & A_{43} & A_{44} \end{bmatrix}-A_{12}\det\begin{bmatrix} A_{21} & A_{23} & A_{24} \\ A_{31} & A_{33} & A_{34} \\ A_{41} & A_{43} & A_{44} \end{bmatrix}$$

$$+A_{13}\det\begin{bmatrix} A_{21} & A_{22} & A_{24} \\ A_{31} & A_{32} & A_{34} \\ A_{41} & A_{42} & A_{44} \end{bmatrix}-A_{14}\det\begin{bmatrix} A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \\ A_{41} & A_{42} & A_{43} \end{bmatrix}$$

在 3D 图形中，我们主要使用 4×4 矩阵，所以就不再讨论 $n>4$ 时的公式了。

# 例 2.9

求下面矩阵的行列式：

$$\mathbf{A}=\begin{bmatrix} 2 & -5 & 3 \\ 1 & 3 & 4 \\ -2 & 3 & 7 \end{bmatrix}$$

我们可以得到：

$$\det \mathbf{A} = A_{11} \det \begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix} - A_{12} \det \begin{bmatrix} A_{21} & A_{23} \\ A_{31} & A_{33} \end{bmatrix} + A_{13} \det \begin{bmatrix} A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix}$$

$$\det \mathbf{A} = 2 \det \begin{bmatrix} 3 & 4 \\ 3 & 7 \end{bmatrix} - (-5) \det \begin{bmatrix} 1 & 4 \\ -2 & 7 \end{bmatrix} + 3 \det \begin{bmatrix} 1 & 3 \\ -2 & 3 \end{bmatrix}$$

$$= 2(3 \cdot 7 - 4 \cdot 3) + 5(1 \cdot 7 - 4 \cdot (-2)) + 3(1 \cdot 3 - 3 \cdot (-2))$$

$$= 18 + 75 + 27$$

$$= 120$$

# 2.6 伴随矩阵

设 **A** 为一个 $n \times n$ 矩阵，则 $C_{ij} = (-1)^{i+j} \det \overline{\mathbf{A}}_{ij}$ 称为元素 $A_{ij}$ 的代数余子式。如果我们计算 $C_{ij}$ 并用它替换 **A** 中的第 $ij$ 位置的每个元素，我们就可以获得 **A** 的余子矩阵 $\mathbf{C}_A$：

$$\begin{bmatrix} C_{11} & C_{12} & & C_{1n} \\ C_{21} & C_{22} & & C_{2n} \\ & & & \\ C_{n1} & C_{n2} & & C_{nn} \end{bmatrix}$$

如果我们对 $\mathbf{C}_A$ 进行转置，得到的矩阵称为 **A** 的伴随矩阵，可由下面的公式表示：

$$\mathbf{A}^* = \mathbf{C}_A^T \quad \text{（公式 2.5）}$$

在下一节中，我们会学习如何用伴随矩阵帮我们找到计算逆矩阵的明确公式。

# 2.7 逆矩阵

矩阵代数没定义除法运算，但是它定义了一种乘法的逆（inverse）运算。下面的列表总结了有关逆运算的要点：

1．只有正方形矩阵能做逆运算；所以，当我们说求逆矩阵时是假设我们正在处理的是一个正方形矩阵。

2．一个 $n\times n$ 矩阵 $\mathbf{M}$ 的逆矩阵仍然是一个 $n\times n$ 矩阵，记作 $\mathbf{M}^{-1}$。

3．不是所有的正方形矩阵都有逆矩阵。有逆矩阵的正方形矩阵称为可逆（invertible）矩阵，没有逆矩阵的称为单调（singular）矩阵。

4．如果存在逆矩阵，则该逆矩阵是唯一的。

5．将一个矩阵与它的逆矩阵相乘，其结果必定为单位矩阵：$\mathbf{MM}^{-1}=\mathbf{M}^{-1}\mathbf{M}=\mathbf{I}$。注意，矩阵与它的逆矩阵的相乘次序可以互换，这是矩阵乘法中的一个特例。

逆矩阵在求解矩阵方程时非常有用。例如，我们给出矩阵方程 $\mathbf{p}'=\mathbf{pM}$，已知 $\mathbf{p}'$和 $\mathbf{M}$ 的值，求解 $\mathbf{p}$。假设 $\mathbf{M}$ 是可逆矩阵（即，$\mathbf{M}^{-1}$ 存在），那么我们可以按照如下步骤求解：

| | |
|---|---|
| $\mathbf{p}'=\mathbf{pM}$ | |
| $\mathbf{p}'\mathbf{M}^{-1}=\mathbf{pMM}^{-1}$ | 等式两边同时乘以 $\mathbf{M}^{-1}$。 |
| $\mathbf{p}'\mathbf{M}^{-1}=\mathbf{pI}$ | 由逆矩阵的定义可知 $\mathbf{M}^{-1}\mathbf{M}=\mathbf{I}$。 |
| $\mathbf{p}'\mathbf{M}^{-1}=\mathbf{p}$ | 由单位矩阵的定义可知 $\mathbf{pI}=\mathbf{p}$。 |

下面的这个方程可以用来求逆矩阵，本书不会给出证明过程，但是读者可以在任何一本大学线性代数的书籍中找到证明过程，这个方程是用伴随矩阵和行列式的形式给出的：

$$\mathbf{A}^{-1}=\frac{\mathbf{A}^{*}}{\det\mathbf{A}}\quad\text{（公式 2.6）}$$

## 例 2.10

找到一个求 $2\times2$ 矩阵 $\mathbf{A}=\begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix}$ 的逆矩阵的通用公式，并使用这个公式求出 $\mathbf{M}=\begin{bmatrix}3 & 0\\ -1 & 2\end{bmatrix}$ 的逆矩阵。

我们已经知道：

$$\det\mathbf{A}=A_{11}A_{22}-A_{12}A_{21}$$

$$\mathbf{C}_A=\begin{bmatrix}(-1)^{1+1}\det\overline{\mathbf{A}}_{11} & (-1)^{1+2}\det\overline{\mathbf{A}}_{12}\\ (-1)^{2+1}\det\overline{\mathbf{A}}_{21} & (-1)^{2+2}\det\overline{\mathbf{A}}_{22}\end{bmatrix}=\begin{bmatrix}A_{22} & -A_{21}\\ -A_{12} & A_{11}\end{bmatrix}$$

所以，

$$\mathbf{A}^{-1}=\frac{\mathbf{A}^{*}}{\det\mathbf{A}}=\frac{\mathbf{C}_A^{T}}{\det\mathbf{A}}=\frac{1}{A_{11}A_{22}-A_{12}A_{21}}\begin{bmatrix}A_{22} & -A_{12}\\ -A_{21} & A_{11}\end{bmatrix}$$

现在用这个公式求 $\mathbf{M}=\begin{bmatrix}3 & 0\\ -1 & 2\end{bmatrix}$ 的逆矩阵：

$$\mathbf{M}^{-1}=\frac{1}{3\cdot 2-0\cdot(-1)}\begin{bmatrix}2 & 0\\ 1 & 3\end{bmatrix}=\begin{bmatrix}1/3 & 0\\ 1/6 & 1/2\end{bmatrix}$$

我们只需检验 $\mathbf{MM}^{-1}=\mathbf{M}^{-1}\mathbf{M}=\mathbf{I}$ 就可以证明结果是否正确：

$$\begin{bmatrix}3 & 0\\ -1 & 2\end{bmatrix}\begin{bmatrix}1/3 & 0\\ 1/6 & 1/2\end{bmatrix}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}=\begin{bmatrix}1/3 & 0\\ 1/6 & 1/2\end{bmatrix}\begin{bmatrix}3 & 0\\ -1 & 2\end{bmatrix}$$

**注意**：对于小矩阵（4×4 矩阵或更小）来说，使用伴随矩阵的方法更有效率。对于更大的矩阵来说，我们可以使用诸如高斯消元法之类的其他方法求逆矩阵。但是，在 3D 计算机图形中，我们要处理的矩阵具有特定的形式，因此可以事先确定求逆矩阵的方程，这样我们就无需浪费 CPU 资源去求一般矩阵的逆矩阵了。这样，在代码中我们往往很少用到公式 2.6。

在本节结束之前，我们要介绍一个与逆矩阵相乘时非常有用的代数特性：

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

这个特性假设 **A** 和 **B** 都是可逆的，它们都是维数相同的正方形矩阵。要证明 $\mathbf{B}^{-1}\mathbf{A}^{-1}$ 是 **AB** 的逆矩阵，我们只需要证明$(\mathbf{AB})^{-1}(\mathbf{B}^{-1}\mathbf{A}^{-1})=\mathbf{I}$ 和$(\mathbf{B}^{-1}\mathbf{A}^{-1})(\mathbf{AB})=\mathbf{I}$。推导过程如下：

$$(\mathbf{AB})^{-1}(\mathbf{B}^{-1}\mathbf{A}^{-1})=\mathbf{A}(\mathbf{BB}^{-1})\mathbf{A}^{-1}=\mathbf{AIA}^{-1}=\mathbf{AA}^{-1}=\mathbf{I}$$

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

# 2.9 小结

1．一个 $m\times n$ 矩阵 **M** 是 $m$ 行、$n$ 列的矩形实数数组。当且仅当维数相同的两个矩阵的对应元素相等时，这两个矩阵相等。将维数相同的两个矩阵相加，即是将矩阵中的对应元素相加。将一个标量与矩阵相乘，即是将标量与矩阵中的每个元素相乘。

2．如果 **A** 是一个 $m\times n$ 矩阵，**B** 是一个 $n\times p$ 矩阵，**C** 表示 **AB** 的乘积，那么 **C** 是一个 $m\times p$ 矩阵，其中结果 **C** 的第 $ij$ 个元素等于 **A** 中的第 $i$ 个行向量与 **B** 中的第 $j$ 个列向量的点积；也就是，$\mathbf{C}_{ij}=\mathbf{A}_{i,*}\cdot\mathbf{B}_{*,j}$。

3．矩阵乘法不满足交换律（即，多数情况下 $\mathbf{AB}\neq\mathbf{BA}$）。矩阵乘法满足结合律：$\mathbf{(AB)C=A(BC)}$。

4．对矩阵的行和列进行互换，即可得到矩阵的转置矩阵。因此，一个 $m\times n$ 矩阵的转置矩阵是一个 $n\times m$ 矩阵。我们使用 $\mathbf{M}^{\mathrm{T}}$ 表示矩阵 **M** 的转置矩阵。

5．单位矩阵是一种正方形矩阵，它除了对角线上的元素值为 1 外，其他元素均为 0。

6．矩阵行列式 det**A** 是一个特殊的函数，它可以将一个正方矩阵转换为一个实数。只有在 $\det\mathbf{A}\neq 0$ 的情况下，正方矩阵才是是可逆的。我们可以使用行列式计算逆矩阵。

7．将一个矩阵与它的逆矩阵相乘，结果为单位矩阵：$\mathbf{MM}^{-1}=\mathbf{M}^{-1}\mathbf{M}=\mathbf{I}$。如果一个矩阵存在逆矩阵，则该逆矩阵是唯一的。只有正方形矩阵会有逆矩阵，但不是所有的正方形矩阵都可逆。逆矩阵可以使用公式 $\mathbf{A}^{-1}=\dfrac{\mathbf{A}^{*}}{\det\mathbf{A}}$ 得到，其中 $\mathbf{A}^{*}$是伴随矩阵（**A** 的余子矩阵的转置）。

# 3.1 线性变换

我们以几何方式描述3D场景中的物体；也就是用一组三角形近似地模拟物体的外表面。如果我们创建的物体都静止不动，那么场景就会显得索然无趣。所以，我们必须学习对几何体进行变换的方法；常见的几何变换包括平移、旋转和缩放。本章会给出许多矩阵公式，读者可以使用些公式对3D空间中的点和向量进行变换。

**学习目标**

1．了解如何使用矩阵表示线性变换和仿射变换。

2．学习用于缩放、旋转和平移几何体的坐标变换。

3．了解如何通过矩阵-矩阵乘法将多个变换矩阵组合为一个净变换矩阵。

4．了解如何将坐标从一个坐标系转换到另一个坐标系，以及如何通过一个矩阵来描述坐标变换。

5．熟悉用于创建变换矩阵的函数，这些函数是XNA数学库的一个子集。

## 3.1.1 定义

考虑一个数学函数$\tau(\mathbf{v})= \tau(x,y,z) = (x',y',z')$。这个函数的输入和输出都是一个3D向量。当且仅当$\tau$满足以下性质时我们才认为它是一个线性变换：

$$\begin{aligned} &1.\tau(\mathbf{u}+\mathbf{v}) = \tau(\mathbf{u}) + \tau(\mathbf{v}) \\ &2.\tau(k\mathbf{u}) = k\tau(\mathbf{u}) \end{aligned} \quad \text{（公式 3.1）}$$

其中 $\mathbf{u}=(u_x,u_y,u_z)$ 和 $\mathbf{v} = (v_x,v_y,v_z)$为任意3D向量，$k$ 是一个标量。

**注意**：线性变换的输入和输出不一定是3D向量，但在3D图形学的书中我们无需使用其他更普遍的形式。

## 例 3.1

定义一个函数$\tau(x,y,z) = (x^2,y^2,z^2)$；例如，$\tau(1,2,3) =(1,4,9)$。这个函数不是线性的，这是因为若 $k$=2、$\mathbf{u}$=(1,2,3)，我们可以得到：

$$\tau(k\mathbf{u}) = \tau(2, 4, 6) = (4, 16, 36)$$

但

$$k\tau(\mathbf{u}) = 2\ (1, 4, 9) = (2, 8, 18)$$

所以不满足公式 3.1。

如果$\tau$是线性的，它应该满足下面的式子：

$$\begin{aligned} &\tau(a\mathbf{u} + b\mathbf{c} + c\mathbf{w}) = \tau(a\mathbf{u} + (b\mathbf{v} + c\mathbf{w})) \\ &= a\tau(\mathbf{u}) + \tau(b\mathbf{v} + c\mathbf{w}) \\ &= a\tau(\mathbf{u}) + b\tau(\mathbf{v}) + c\tau(\mathbf{w}) \end{aligned} \quad \text{（公式 3.2）}$$

我们会在下一节中使用这个结果。

## 3.1.2 矩阵描述

令 **u**=(x,y,z)。我们总可以写成下面的形式：

$$\mathbf{u}=(x,y,z)=x\,\mathbf{i}+y\,\mathbf{j}+z\,\mathbf{k}=x\,(1,0,0)+y\,(0,1,0)+z\,(0,0,1)$$

向量 **i** = (1, 0, 0)， **j** = (0, 1, 0)和 **k** = (0, 0, 1)都是沿着坐标轴的单位向量，我们把它们称为$\mathbb{R}^3$的标准基向量（*standard basis vectors*）（$\mathbb{R}^3$表示所有 3D 坐标向量$(x, y, z)$的集合）。令$\tau$为线性变换，则根据线性函数的特点（即公式 3.2），我们可以得到：

$$\tau(\mathbf{u})=\tau(x\mathbf{i}+y\mathbf{j}+z\mathbf{k})=x\tau(\mathbf{i})+y\tau(\mathbf{j})+z\tau(\mathbf{k})\quad\text{（公式 3.3）}$$

公式 3.3 其实就是一个线性组合，我们在上一章就已经讨论过了，这个线性组合可以根据公式 2.2 写成矢量与矩阵的乘法，因此我们可以将公式 3.3 重写成如下形式：

$$\begin{aligned}&\tau(\mathbf{u})=x\tau(\mathbf{i})+y\tau(\mathbf{j})+z\tau(\mathbf{k})\\&=\mathbf{u}\mathbf{A}=[x,y,z]\begin{bmatrix}\leftarrow & \tau(\mathbf{i}) & \rightarrow\\ \leftarrow & \tau(\mathbf{j}) & \rightarrow\\ \leftarrow & \tau(\mathbf{k}) & \rightarrow\end{bmatrix}=[x,y,z]\begin{bmatrix}A_{11} & A_{12} & A_{13}\\ A_{21} & A_{21} & A_{23}\\ A_{31} & A_{32} & A_{33}\end{bmatrix}\end{aligned}\quad\text{（公式 3.4）}$$

其中$\tau$ (**i**) = ($A_{11}$, $A_{12}$, $A_{13}$)， $\tau$ (**j**) = ($A_{21}$, $A_{22}$, $A_{23}$)， $\tau$ (**k**) = ($A_{31}$, $A_{32}$, $A_{33}$)。我们把矩阵 **A** 称为线性变换$\tau$的矩阵描述。

## 3.1.3 缩放

缩放是指改变一个物体的大小，如图 3.1 所示。

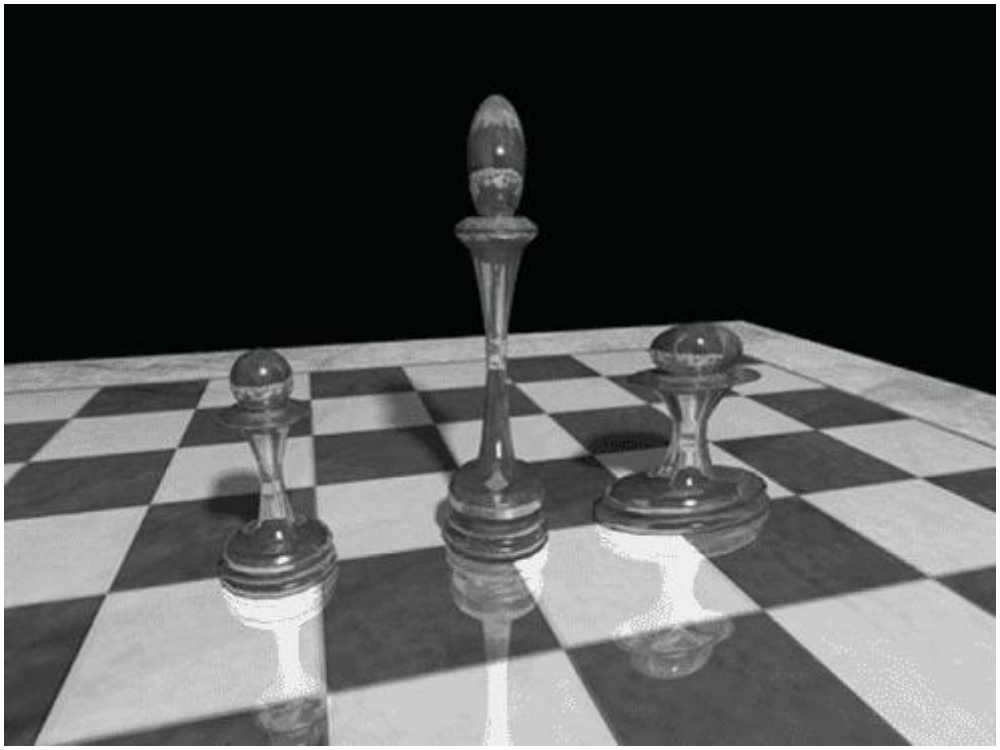


**图 3.1 左边的兵（pawn，国际象棋中的兵）是原始物体。中间的兵是沿 y 轴放大两倍后的结果。右边的兵是沿 x 轴放大两倍后的结果。**

我们将缩放变换定义为：

$$S(x,y,z)=(s_x x, s_y y, s_z z)$$

上述变换将向量沿 $x$ 轴方向缩放 $s_x$ 单位，$y$ 轴方向缩放 $s_y$ 个单位，$z$ 轴方向缩放 $s_z$ 个单位（相对于目前的坐标系原点）。下面我们证明 $S$ 是一个线性变换：

$$\begin{aligned}
S(\mathbf{u}+\mathbf{v}) &= (s_x(u_x+v_x), s_y(u_y+v_y), s_z(u_z+v_z)) \\
&= (s_x u_x + s_x v_x, s_y u_y + s_y v_y, s_z u_z + s_z v_z) \\
&= (s_x u_x, s_y u_y, s_z u_z) + (s_x v_x, s_y v_y, s_z v_z) \\
&= S(\mathbf{u}) + S(\mathbf{v})
\end{aligned}$$

$$\begin{aligned}
S(k\mathbf{u}) &= (s_x k u_x, s_y k u_y, s_z k u_z) \\
&= k(s_x u_x, s_y u_y, s_z u_z) \\
&= kS(\mathbf{u})
\end{aligned}$$

满足公式 3.1 的两个性质，所以 $S$ 是线性的，应该存在一个矩阵描述。要找到这个矩阵描述，我们只需将 $S$ 代入公式 3.3 中的每个标准基向量中即可，然后将得出的结果向量替换矩阵的行：

$$\begin{aligned}
S(\mathbf{i}) &= (s_x \cdot 1, s_y \cdot 0, s_z \cdot 0) = (s_x, 0, 0) \\
S(\mathbf{j}) &= (s_x \cdot 0, s_y \cdot 1, s_z \cdot 0) = (0, s_y, 0) \\
S(\mathbf{k}) &= (s_x \cdot 0, s_y \cdot 0, s_z \cdot 1) = (0, 0, s_z)
\end{aligned}$$

$S$ 的矩阵表示为：

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

我们把这个矩阵叫做缩放矩阵。

缩放矩阵的逆矩阵为：

$$\mathbf{S} = \begin{bmatrix} 1/s_x & 0 & 0 \\ 0 & 1/s_y & 0 \\ 0 & 0 & 1/s_z \end{bmatrix}$$

# 例 3.2

假设我们通过一个最小点(−4, −4,0)和一个最大点(4, 4,0)来定义一个正方形，我们希望将正方形沿 $x$ 轴缩小 0.5 倍，沿 $y$ 轴放大 2.0 倍，$z$ 轴保持不变。则对应的缩放矩阵为：

$$\mathbf{S} = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

现在，对正方形进行缩放（变换），将正方形的两个点与该矩阵相乘：

$$\begin{bmatrix}-4 & -4 & 0\end{bmatrix}\begin{bmatrix}0.5 & 0 & 0\\ 0 & 2 & 0\\ 0 & 0 & 1\end{bmatrix}=\begin{bmatrix}-2 & -8 & 0\end{bmatrix}$$

$$\begin{bmatrix}4 & 4 & 0\end{bmatrix}\begin{bmatrix}0.5 & 0 & 0\\ 0 & 2 & 0\\ 0 & 0 & 1\end{bmatrix}=\begin{bmatrix}2 & 8 & 0\end{bmatrix}$$

结果如图 3.2 所示。

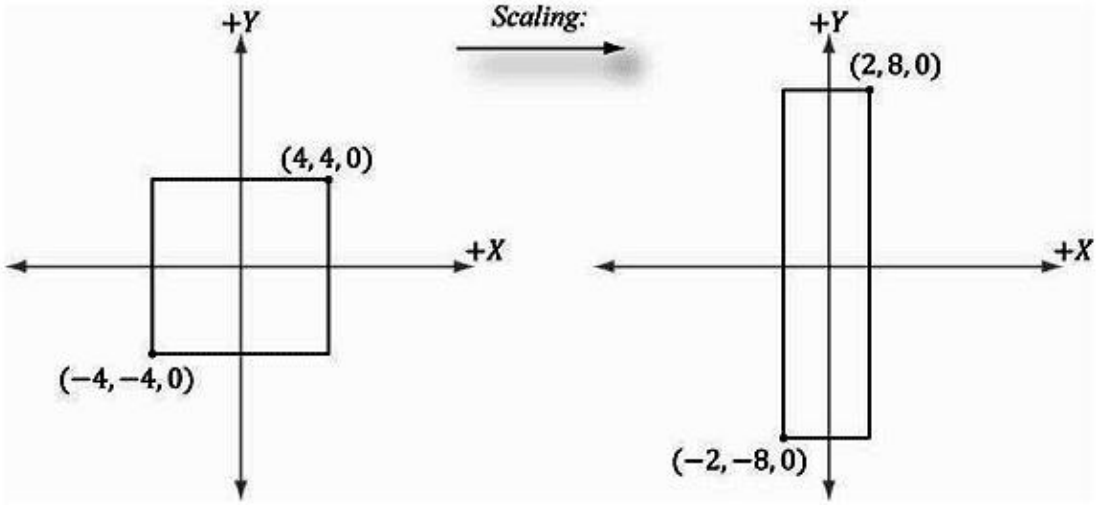



**图 3.2 沿 x 轴缩小 0.5 倍，沿 *y* 轴扩大 2 倍。注意，当沿 *z* 轴负方向俯视时，由于 *z* 值为 0，几何体看上去是一个 2D 平面图形。**

## 3.1.4 旋转

本节我们将介绍如何将向量 **v** 绕一根轴 **n** 旋转$\theta$角度；如图 3.3 所示。注意，在左手坐标系中，当沿着旋转轴的正轴方向俯视时，顺时针方向为正角；而且，我们假设$\|\mathbf{n}\|=1$。

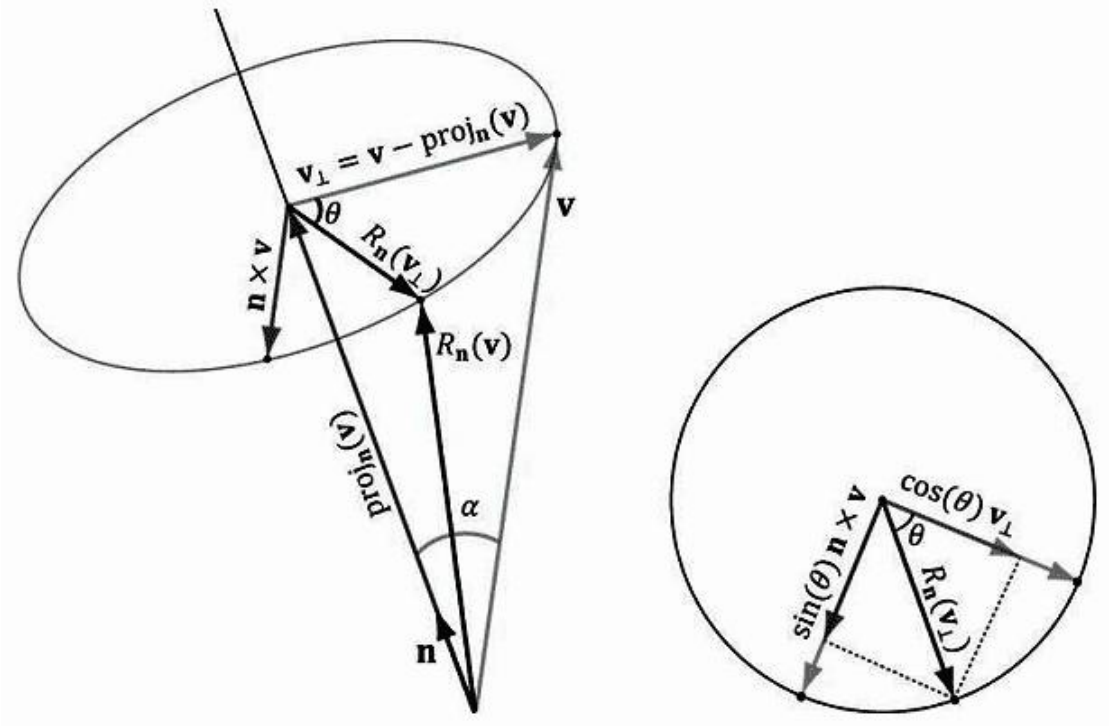



**图 3.3 绕任意轴 n 旋转的几何表示。**

首先，将 **v** 分解为两个分量，其中一个分量平行于 **n**，另一个垂直于 **n**。平行分量即 $\text{proj}_\mathbf{n}(\mathbf{v})$（回忆一下例 1.5）；垂直分量可以通过 $\mathbf{v}_\perp = \text{perp}_\mathbf{n}(\mathbf{v}) = \mathbf{v} - \text{proj}_\mathbf{n}(\mathbf{v})$得到（还是回忆一下例 1.5，因为 **n** 是单位向量，**所以** $\text{proj}_\mathbf{n}(\mathbf{v}) = (\mathbf{n}\cdot\mathbf{v})\mathbf{n}$。）。平行于 **n** 的 $\text{proj}_\mathbf{n}(\mathbf{v})$分量在旋转过程中是不变的，所以我们只需计算垂直分量的旋转。从图 3.3 中我们可以看出，旋转后的向量 $R_\mathbf{n}(\mathbf{v}) = \text{proj}_\mathbf{n}(\mathbf{v}) + R_\mathbf{n}(\mathbf{v}_\perp)$。

要找到 $R_n(\mathbf{v}_\perp)$，我们需要建立一个位于旋转平面的 2D 坐标系。将 $\mathbf{v}_\perp$作为一个基准向量，第二个基准向量需要同时垂直于 $\mathbf{v}_\perp$和 **n**，我们取为 $\mathbf{n} \times \mathbf{v}$（左手拇指定则）。根据图 3.3 中的几何关系和第一章的练习 14，我们可以得出：

$$\|\mathbf{n}\times\mathbf{v}\| = \|\mathbf{n}\|\|\mathbf{v}\|\sin\alpha = \|\mathbf{v}\|\sin\alpha = \|\mathbf{v}_\perp\|$$

其中$\alpha$为 **n** 和 **v** 之间的夹角。这样这两个基准向量都有相同的长度并且都在旋转平面上。创建了两个基准向量后，我们就可以根据三角学的知识得出：

$$R_n(\mathbf{v}_\perp) = \cos\theta\mathbf{v}_\perp + \sin\theta(\mathbf{n}\times\mathbf{v})$$

并由此得到下面的旋转方程：

$$\begin{aligned} R_n(\mathbf{v}) &= proj_n(\mathbf{v}) + R_n(\mathbf{v}_\perp) \\ &= (\mathbf{n}\cdot\mathbf{v})\mathbf{n} + \cos\theta\mathbf{v}_\perp + \sin\theta(\mathbf{n}\times\mathbf{v}) \\ &= (\mathbf{n}\cdot\mathbf{v})\mathbf{n} + \cos\theta(\mathbf{v} - (\mathbf{n}\cdot\mathbf{v})\mathbf{n}) + \sin\theta(\mathbf{n}\times\mathbf{v}) \\ &= \cos\theta\mathbf{v} + (1-\cos\theta)(\mathbf{n}\cdot\mathbf{v})\mathbf{n} + \sin\theta(\mathbf{n}\times\mathbf{v}) \end{aligned} \qquad \text{（公式 3.5）}$$

我们把证明公式 3.5 为一个线性变换放在了后面的练习中，这里不予讨论。要找到对应的矩阵描述，我们只需将 $R_\mathbf{n}$ 代入公式 3.3 中的每个标准基向量中即可，然后将得出的结果向量替换矩阵的行（即在公式 3.4 中）。最终结果为：

$$\mathbf{R}_n = \begin{bmatrix} c+(1-c)x^2 & (1-c)xy+sz & (1-c)xz-sy \\ (1-c)xy-sz & c+(1-c)y^2 & (1-c)yz-sx \\ (1-c)xz-sy & (1-c)yz+sx & c+(1-c)z^2 \end{bmatrix}$$

其中 $c=\cos\theta$，$s=\sin\theta$。

旋转矩阵有一个有趣的特性。读者可以验证一下：旋转矩阵的每个行向量都是单位向量，而且相互垂直。也就是说，它的每个行向量都是标准正交的（即，相互垂直且为单位长度）。我们将这种矩阵称为正交矩阵（orthogonal matrix）。正交矩阵有一个非常有用的特性，它的逆矩阵与它的转置矩阵相等。也就是，$\mathbf{R}_n$ 的逆矩阵为：

$$\mathbf{R}_n^{-1} = \mathbf{R}_n^T = \begin{bmatrix} c+(1-c)x^2 & (1-c)xy-sz & (1-c)xz+sy \\ (1-c)xy+sz & c+(1-c)y^2 & (1-c)yz-sx \\ (1-c)xz-sy & (1-c)yz+sx & c+(1-c)z^2 \end{bmatrix}$$

通常，正交矩阵是最容易使用的矩阵，因为它们计算逆矩阵的过程非常简单，也非常高效。

当我们以 $x$、$y$、$z$ 轴（即，$\mathbf{n}=(1,0,0)$、$\mathbf{n}=(0,1,0)$、$\mathbf{n}=(0,0,1)$）为旋转轴时，对应的旋转矩阵如下：

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix}, \mathbf{R}_y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}, \mathbf{R}_z = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

# 例 3.3

假设我们通过一个最小点(−1,0,−1)和一个最大点(1,0,1)来定义一个正方形。让正方形绕着 $y$ 轴的顺时针方向旋转-30º（即，逆时针方向旋转 30º）。在这种情况中，$\mathbf{n}=(0,1,0)$，$\mathbf{R}_n$ 大为简化；对应的 $y$ 轴旋转矩阵为：

$$\mathbf{R}_y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} = \begin{bmatrix} \cos(-30°) & 0 & -\sin(-30°) \\ 0 & 1 & 0 \\ \sin(-30°) & 0 & \cos(-30°) \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2} \end{bmatrix} \approx \begin{bmatrix} 0.36, 0, 1.36 \end{bmatrix}$$

现在，对正方形进行旋转（变换），将正方形的两个点与该矩阵相乘：

$$[-1,0,-1]\begin{bmatrix} \frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2} \end{bmatrix} \approx [-0.36,0,-1.36]$$

$$[1,0,1]\begin{bmatrix} \frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2} \end{bmatrix} \approx [0.36,0,1.36]$$

结果如图 3.4 所示。

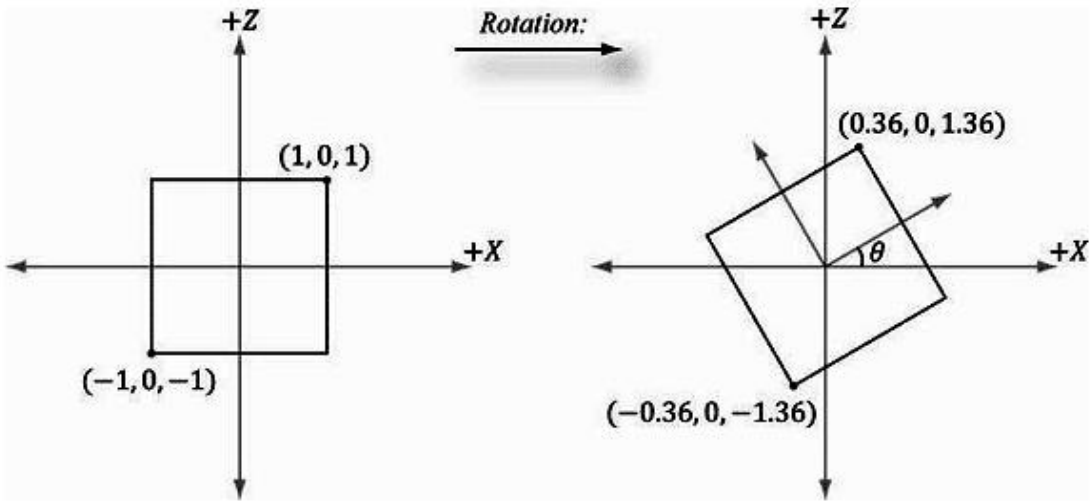



**图 3.4：绕 *y* 轴顺时针方向旋转−30°。注意，当沿 *y* 轴正方向俯视时，由于 *y* 值为 0，几何体看上去是一个 2D 平面图形。**

# 3.2 仿射变换

## 3.2.1 齐次坐标

下一节我们就会知道仿射变换是一个组合了平移的线性变换。但是，因为向量只表示方向和长度，与位置无关，所以平移一个向量是无意义的，换句话说，平移后的向量是不变的。平移只能作用在点上（即，位置向量）。齐次坐标提供了一个便捷的表示方法用来统一处理点和向量。在齐次坐标中，我们使用 4 个元素，我们通过它的第 4 个坐标分量 $w$ 来决定所描述的是一个点还是一个向量。确切地说，我们写为：

1．$(x,y,z,0)$用于向量

2．$(x,y,z,1)$用于点

我们将会看到，把 $w$ 设为 1 是为了让点的平移操作得到正确执行，把 $w$ 设为 0 是为了防止向量在变换过程中发生平移。（我们不希望平移向量的坐标，因为向量可以改变的只有方向和大小——平移对向量来说没有意义。）

**注意**：齐次坐标的记法与图 1.17 所示的概念一致。也就是，两点相减 $\mathbf{q}-\mathbf{p}=(q_x,q_y,q_z,1)-(p_x,p_y,p_z,1)=(q_x-p_x,q_y-p_y,q_z-p_z,0)$ 的结果是一个向量，而一个点与一个向量相加 $\mathbf{p}+\mathbf{v}=(p_x,p_y,p_z,1)+(v_x,v_y,v_z,0)=(p_x+v_x,p_y+v_y,p_z+v_z,1)$ 的结果是个点。

## 3.2.2 定义和矩阵表示

一个线性变换无法表示所有我们需要的变换；所以，我们需要添加一组叫做仿射变换的函数。仿射变换是一个线性变换加上一个平移向量 **b**；也就是：

$$\alpha(\mathbf{u})=\tau(\mathbf{u})+b$$

或者用矩阵表示为：

$$\alpha(\mathbf{u})=\mathbf{uA}+\mathbf{b}=[x,y,z]\begin{bmatrix}A_{11}&A_{12}&A_{13}\\A_{21}&A_{22}&A_{23}\\A_{31}&A_{32}&A_{33}\end{bmatrix}+[b_x,b_y,b_z]=[x',y',z']$$

式中的 A 是线性变换的矩阵表示。

如果使用 $w$=1 的齐次坐标，则可表示为：

$$[x,y,z,1]\begin{bmatrix}A_{11}&A_{12}&A_{13}&0\\A_{21}&A_{22}&A_{23}&0\\A_{31}&A_{32}&A_{33}&0\\b_x&b_y&b_z&1\end{bmatrix}=[x',y',z',1]\quad\text{（公式 3.6）}$$

公式 3.6 中的 4×4 矩阵称为仿射矩阵的矩阵表示。

额外添加的 **b** 就是指平移（即，位置的改变）。因为向量没有位置的概念，所以我们并不想将 **b** 作用在向量上。但是，我们还是想将仿射矩阵的线性变换部分作用在向量上。如果我们把向量的第四个分量 $w$ 设置为 0，那么 **b** 对应的平移部分就不会作用到向量上。

**注意**：因为行向量与 4×4 仿射矩阵第 4 列的点乘是 $[x,y,z,w]\cdot[0,0,0,1]=w$，所以

这个矩阵不会改变输入向量的 *w* 坐标。

### 3.2.3 平移

单位变换（identity transformation）是一个线性变换，返回值就是输入的向量；即，$I(\mathbf{u})=\mathbf{u}$。这说明线性变换的矩阵表示就是一个单位矩阵。

现在，我们将一个平移变换定义为一个仿射变换，这个仿射变换的线性变换部分是一个单位变换；即：

$$\tau(\mathbf{u}) = \mathbf{u}\mathbf{I} + \mathbf{b} = \mathbf{u} + \mathbf{b}$$

如你所见，这可以简化将点 **u** 移动 **b** 的操作。图 3.5 说明了如何平移物体——要对点 **u** 进行平移，只需要将一个移位向量 **b** 和该点相加，即可得到新的点 **u**+**b**。注意，要平移一个完整的物体，我们就要通过相同的向量 **b** 来平移物体上的每个点。

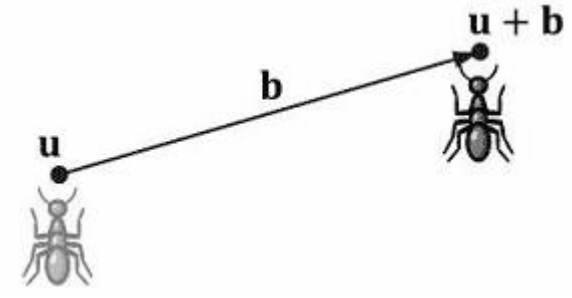



**图 3.5 通过位移向量 b 对蚂蚁的位置进行平移。**

根据公式 3.6，τ的矩阵表示如下：

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ b_x & b_y & b_z & 1 \end{bmatrix}$$

这个矩阵称之为平移矩阵。

平移矩阵的逆矩阵如下：

$$\mathbf{T}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -b_x & -b_y & -b_z & 1 \end{bmatrix}$$

## 例 3.4

假设我们通过一个最小点(−8, 2,0)和一个最大点(−2, 8,0) 来定义一个正方形。让正方形沿 *x* 轴平移 12，沿 *y* 轴平移-10，*z* 轴保持不变。则对应的平移矩阵如下：

$$\mathbf{T}=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 12 & -10 & 0 & 1\end{bmatrix}$$

现在，对正方形进行平移（变换），将正方形的两个点与该矩阵相乘：

$$\begin{bmatrix}-8 & 2 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 12 & -10 & 0 & 1\end{bmatrix}=\begin{bmatrix}4 & -8 & 0 & 1\end{bmatrix}$$

$$\begin{bmatrix}-2 & 8 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 12 & -10 & 0 & 1\end{bmatrix}=\begin{bmatrix}10 & -2 & 0 & 1\end{bmatrix}$$

结果如图 3.6 所示。

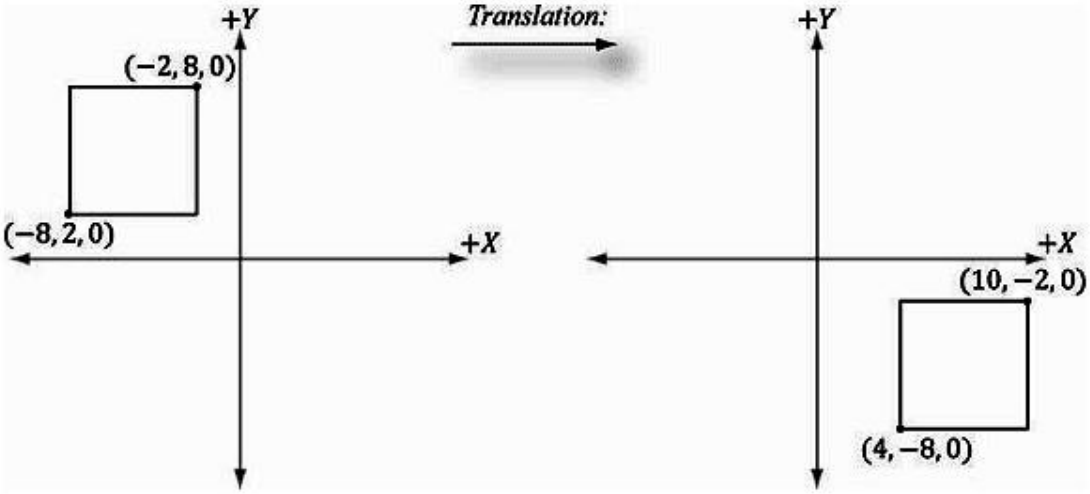



**图 3.6 沿 x 轴平移 12，沿 *y* 轴平移−10。注意，当沿 *z* 轴负方向俯视时，由于 *z* 值为 0，几何体看上去是一个 2D 平面图形。**

**注意**：令 **T** 为一个变换矩阵，通过计算 **vT= v′**就可以对点/向量进行变换。如果先使用 **T** 对点/向量进行变换，然后再使用逆矩阵 $\mathbf{T}^{-1}$ 进行变换，我们就会得到初始的向量：$\mathbf{vTT}^{-1}=\mathbf{vI}=\mathbf{v}$。换句话说，逆变换可以撤销变换。例如，如果我们将一个点沿 x 轴移动 5，然后沿 x 轴移动-5，会又回到出发点。类似地有，将一个点沿 y 轴旋转 30°，然后反向旋转 30°，该点又回到了原来的位置。总而言之，逆变换矩阵的变换效果与变换相反，两者的组合会导致不产生变换效果。

## 3.2.4 缩放和旋转的仿射矩阵

若 **b**=0，仿射变换就退化为一个线性变换。我们可以将任何一个线性变换表示成 **b**=0 的仿射变换。换句话说，我们可以用一个 4×4 仿射矩阵表示任意一个线性变换。例如，缩放和旋转矩阵可以用 4×4 矩阵写成如下形式：

$$\mathbf{S}=\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_n=\begin{bmatrix} c+(1-c)x^2 & (1-c)xy+sz & (1-c)xz-sy & 0 \\ (1-c)xy-sz & c+(1-c)y^2 & (1-c)yz+sx & 0 \\ (1-c)xz+sy & (1-c)yz-sx & c+(1-c)z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

通过这种方式，我们使用 4×4 矩阵表示所有变换，使用 1×4 齐次行向量矩阵表示点和向量，形式得到了统一。

## 3.2.5 仿射变换矩阵的几何解释

在本节中，我们会对一个仿射变换矩阵中的数字表示的几何意义有个直观的认识。首先，考虑一个刚体变换，它是一个形状保持不变的变换。现实世界中的例子是将一本书从桌上拿起放到书架上，在这个过程中，你将书从书桌平移到了书架，还改变了书的朝向（旋转）。令$\tau$为旋转变换，**b** 为位移矢量，则这个刚体变换可以用以下仿射变换表示：

$$\alpha(x,y,z)=\tau(x,y,z)+\mathbf{b}=x\tau(\mathbf{i})+y\tau(\mathbf{j})+z\tau(\mathbf{k})+\mathbf{b}$$

在矩阵表示中，使用其次坐标（$w$=1 表示位置，$w$=0 表示向量，这样平移就不会作用在向量上），上式可以写成：

$$[x,y,z,w]\begin{bmatrix} \leftarrow \tau(\mathbf{i}) \rightarrow \\ \leftarrow \tau(\mathbf{j}) \rightarrow \\ \leftarrow \tau(\mathbf{k}) \rightarrow \\ \leftarrow \mathbf{b} \rightarrow \end{bmatrix}=[x',y',z',w] \quad \text{（公式 3.7）}$$

现在看一下公式 3.7 的几何意义，我们只需画出矩阵中的行向量（参见图 3.7）。因为τ是一个旋转变换，所以它保存了长度和角度信息；而且τ只是将标准基向量 **i**，**j** 和 **k** 旋转到一个新的朝向τ(**i**)，τ(**j**)和τ(**k**)。向量 **b** 只是一个位置向量，它表示的是离开原点的位移。图 3.7 展示了如何通过计算$\alpha(x,y,z)=x\tau(\mathbf{i})+y\tau(\mathbf{j})+z\tau(\mathbf{k})+\mathbf{b}$获得变换后的点。

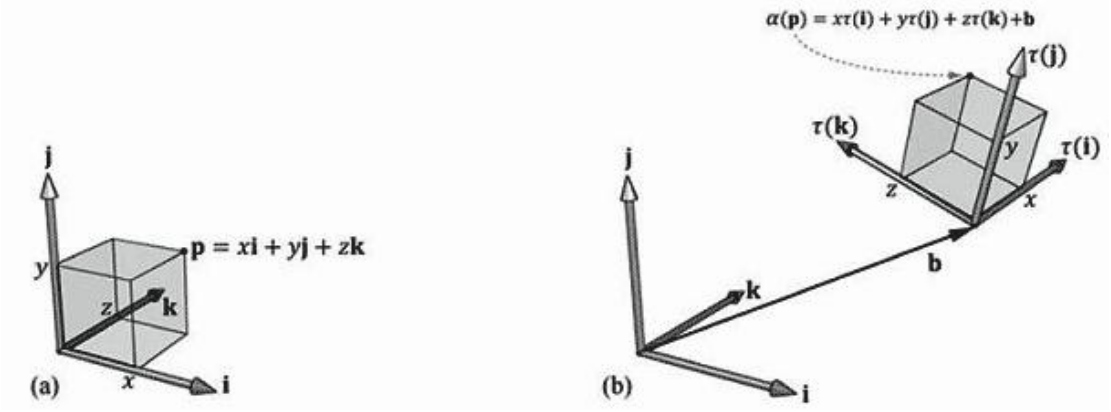



**图 3.7 仿射变换矩阵中行向量的几何意义。变换后的点α(p)为基向量τ(i)，τ(j)，τ(k)的线性**

**组合和偏移量 b 的和。**

缩放和扭曲变换的原理相同。考虑线性变换τ，如图 3.8 所示，这个变换将正方形扭曲成一个平行四边形。扭曲后的点就是扭曲后的基向量的线性组合。

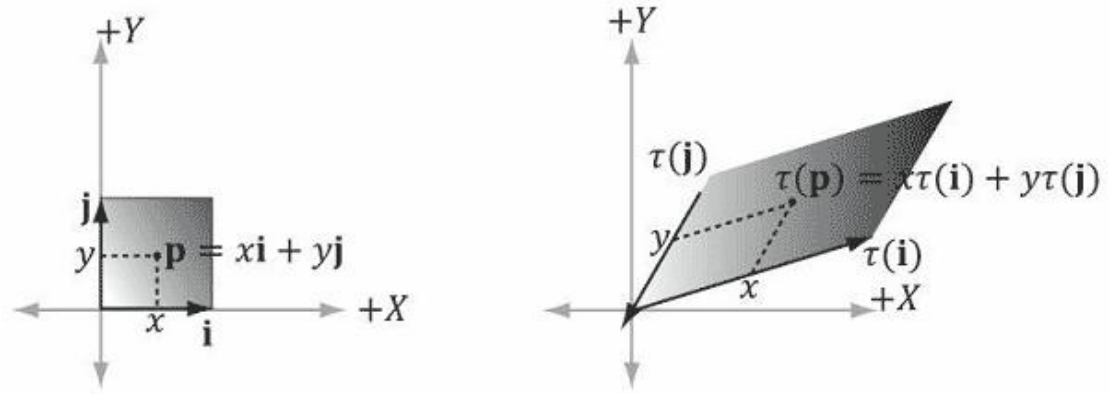



**图 3.8 对于将正方形扭曲为一个平行四边形的线性变换来说，变换后的点τ(p)=(*x*, *y*)是变换后的基向量τ(i)，τ(j)的线性组合。**

# 3.3 组合变换

假设 **S** 是一个缩放矩阵，**R** 是一个旋转矩阵，**T** 是一个平移矩阵；另外，我们有一个由 8 个顶点 $\mathbf{v}_i$（i=0，1，…，7）构成的立方体，我们希望将这 3 个变换连续应用于立方体的每个顶点。一种最容易想到的方法是将这些矩阵逐一应用于每个顶点：

$$((\mathbf{v}_i\mathbf{S})\mathbf{R})\mathbf{T} = (\mathbf{v}_i'\mathbf{R})\mathbf{T} = \mathbf{v}_i''\mathbf{T} = \mathbf{v}_i''' \quad \text{其中 } i=0，1，\cdots，7$$

但是，由于矩阵乘法支持结合律，所以我们可以将上面的方程改为：

$$\mathbf{v}_i(\mathbf{SRT}) = \mathbf{v}_i''' \quad \text{其中 } i=0，1，\cdots，7$$

我们可以将 **SRT** 看成一个矩阵 **C**，将所有的 3 个变换封装为一个净仿射变换矩阵。换句话说，矩阵-矩阵乘法可以让我们把多个变换连接在一起。

这种方法有助于提升性能。比如，我们将 3 个连续的几何变换应用于一个由 20,000 个点构成的 3D 物体。使用逐一相乘的方法，我们需要执行 20,000×3 次向量-矩阵乘法。而改用组合矩阵方式，我们只需要执行 20,000 次向量-矩阵乘法和两次矩阵-矩阵乘法。很明显，两次额外的矩阵-矩阵乘法所产生的资源消耗微乎其微，而它们却能省去大量的向量-矩阵乘法。

**注意**：我们再次强调，矩阵乘法不支持交换律。这完全符合于几何学中的定义。例如，在旋转之后进行一次平移，可以由矩阵乘积 **RT** 表示，但是它的结果与 **TR** 完全不同，也就是，在相同的平移之后进行一次相同的旋转，得到的变换结果完全不同。图 3.9 说明了这一点。

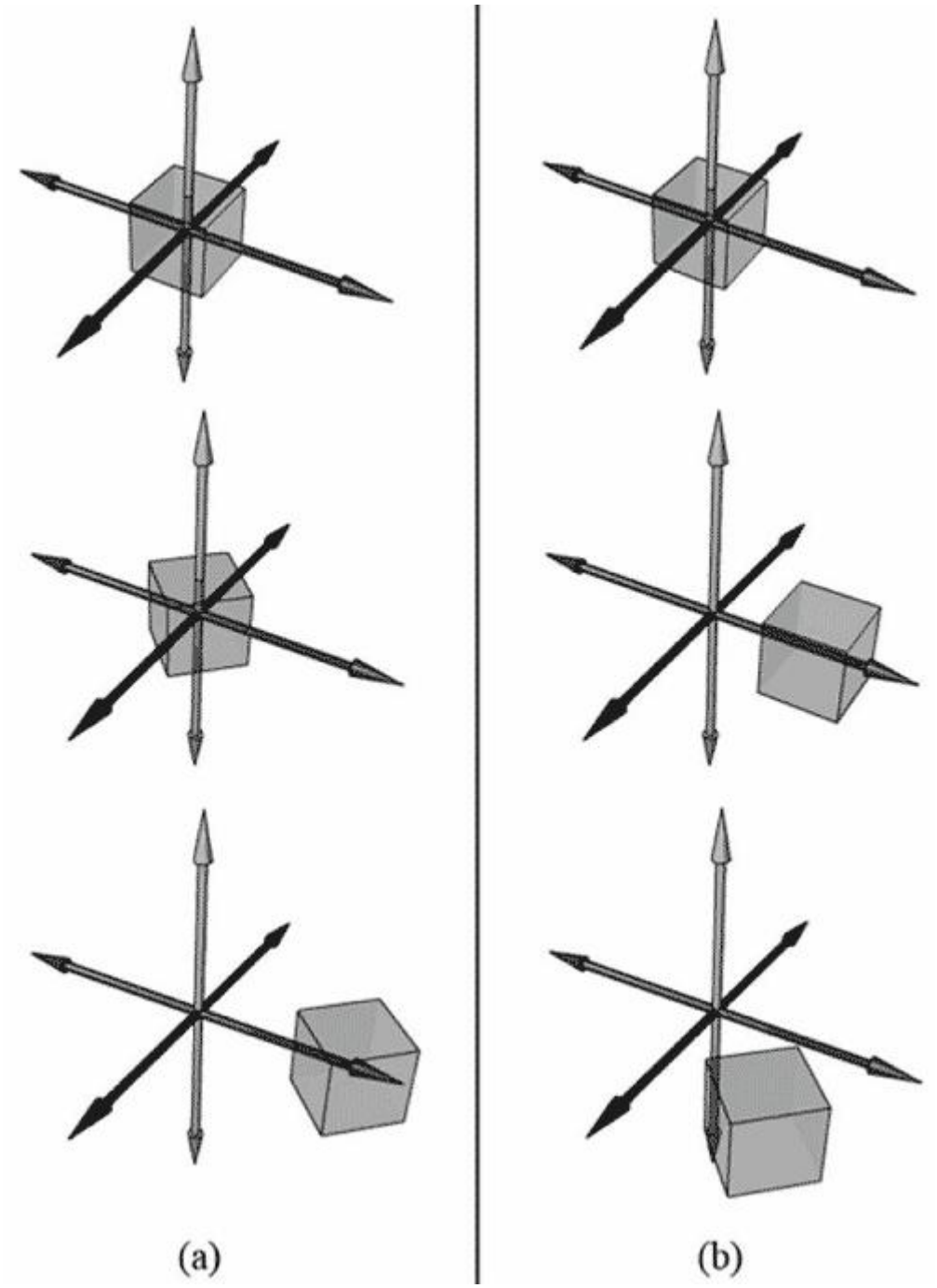



**图 3.9 (a)先旋转再平移。(b)先平移再旋转。**

# 3.4 坐标转换变换

标量 100℃是相对于摄氏温标表示的水的沸点温度。那么我们该如何以华氏温标来描述的水的沸点温度呢？换句话说就是在华氏温标中表示水的沸点温度的标量是多少？要实现这一转换（或参考系变换），我们需要知道摄氏与华氏之间的比例关系。它们的关系如下：$T_F=\frac{9}{5}T_C+32°$。因此，$T_F=\frac{9}{5}$（100）°＋32°=212℉；也就是，水的沸点温度为华氏 212℉。

这个例子说明，只要我们知道参考系 *A* 和参考系 *B* 的关系，就可以将一个相对于参考系 *A* 标量 *k* 转换为相对于参考系 *B* 描述的等价标量 *k*′。在下面的小节中，我们会看到一些类似的问题，但不是标量而是坐标，我们会将一个点或向量的坐标从一个参考系转换到另一个不同的参考系（参见图 3.10）。我们将这种把坐标从一个参考系转换到另一个参考系的变换称为坐标转换变换（change of coordinate transformation）。

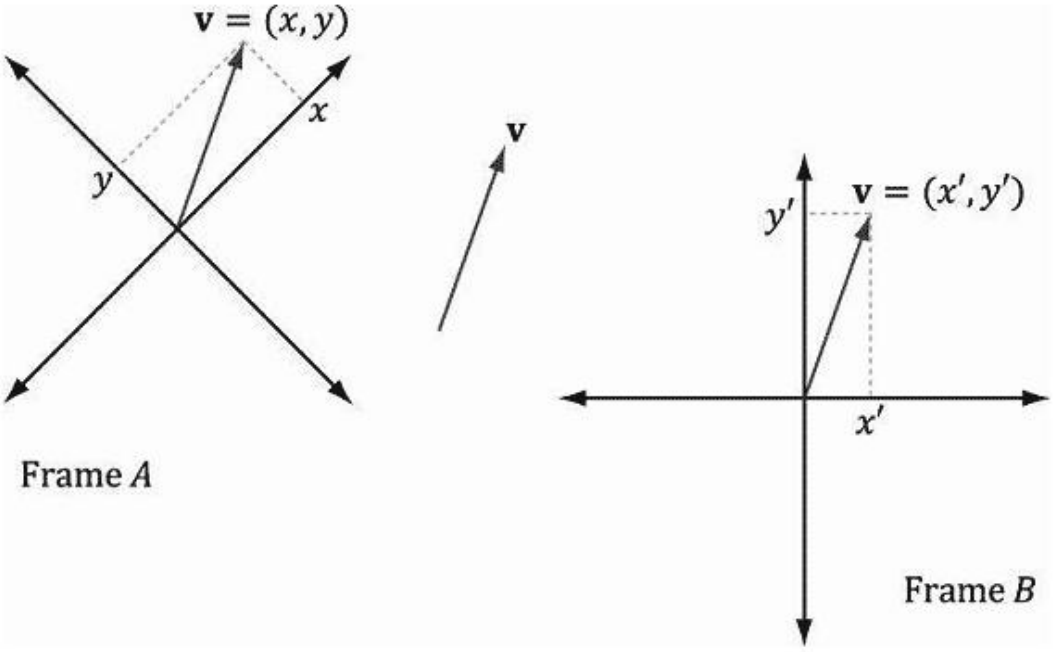



**图 3.10 当相对于不同的参考系来描述同一个向量 v 时，该向量会有不同的坐标。**

值得强调的是，在坐标转换变换中，我们并不认为几何体发生了什么改变；而是认为我们对参考系进行了转换，改变了几何体坐标的表达方式。相比之下，我们通常认为旋转、平移和缩放会对几何体产生实质性的移动或变形。

在 3D 计算机绘图中，由于我们会用到很多种不同坐标系，所以我们需要知道如何从一种坐标系转换到另一种坐标系。由于位置是点的属性，而不是向量的属性，所以点和向量在实现坐标转换变换时要区别对待，使用不同的处理方式。

## 3.4.1 向量

考虑图3.11中的两个参考系*A*、参考系*B*及向量**p**。假设**p**在参考系A中的坐标为$\mathbf{p}_A=(x,y)$，

我们想要求出 **p** 相对于参考系 *B* 的坐标 $\mathbf{p}_B=(x',y')$。换句话说，在一个参考系中通过一个坐标指定一个向量，我们将个向量保持不变，只更换一个不同的参考系，那我们该如何求出这个向量的新坐标呢？

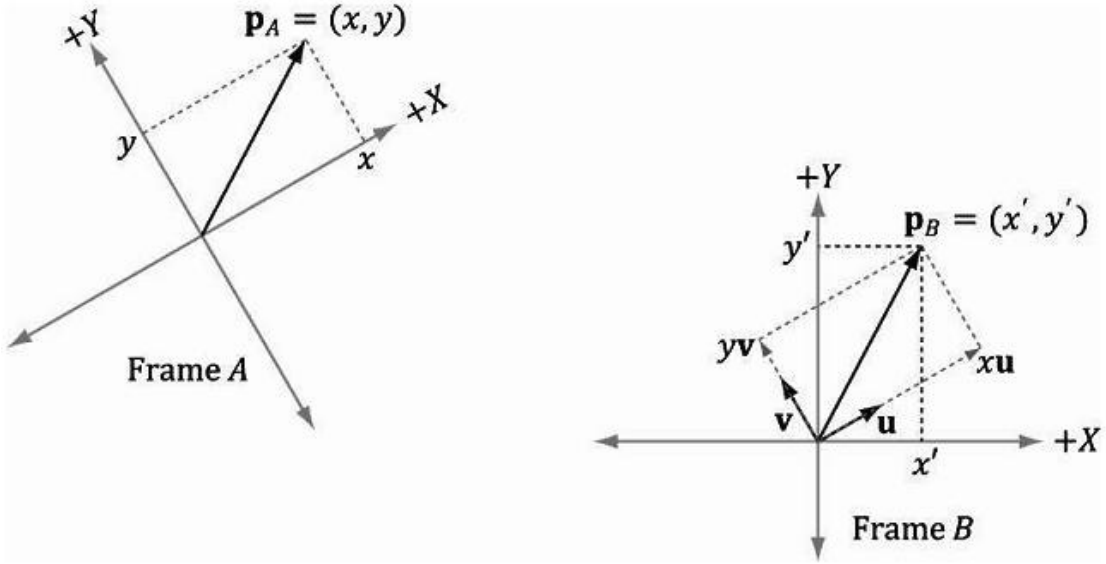



**图 3.11 求 p 相对于参考系 *B* 的几何坐标。**

从图 3.11 可知

$$\mathbf{p}=x\mathbf{u}+y\mathbf{v}$$

其中 **u** 和 **v** 是单位向量，它们所指向的方向分别与参考系 *A* 的 *x* 轴和 *y* 轴方向相同。通过上述方程，我们可以得到每个向量在参考系 B 中的坐标：

$$\mathbf{p}_B=x\mathbf{u}_B+y\mathbf{v}_B$$

这样，只要我们知道向量 **u**、**v** 相对于参考系 *B* 的坐标，即 $\mathbf{u}_B=(u_x,u_y)$和 $\mathbf{v}_B=(v_x,v_y)$，那么对于给出的任意 $\mathbf{p}_A=(x,y)$，都可以计算出 $\mathbf{p}_B=(x',y')$。

推导为 3D 向量，如果 $\mathbf{p}_A=(x,y,z)$，则

$$\mathbf{p}_B=x\mathbf{u}_B+y\mathbf{v}_B+z\mathbf{w}_B$$

其中 **u**、**v**、**w** 是单位向量，它们所指向的方向分别与参考系 *A* 的 *x* 轴、*y* 轴、*z* 轴方向相同。

## 3.4.2 点

当进行坐标转换变换时，点与向量之间存在一些微小差异；由于位置是点的一个重要属性，所以我们不能按照图 3.11 中的平移向量的方式来平移点。

图 3.12 说明了这一情况。我们看到，点 **p** 可以由一个方程来表示：

$$\mathbf{p}=x\mathbf{u}+y\mathbf{v}+\mathbf{Q}$$

其中 **u** 和 **v** 是单位向量，它们所指向的方向分别与参考系 *A* 的 *x* 轴和 *y* 轴方向相同，**Q** 是参考系 *A* 的原点。通过上述方程，我们可以得到向量或点在参考系 *B* 中的坐标：

$$\mathbf{p}_B=x\mathbf{u}_B+y\mathbf{v}_B+\mathbf{Q}_B$$

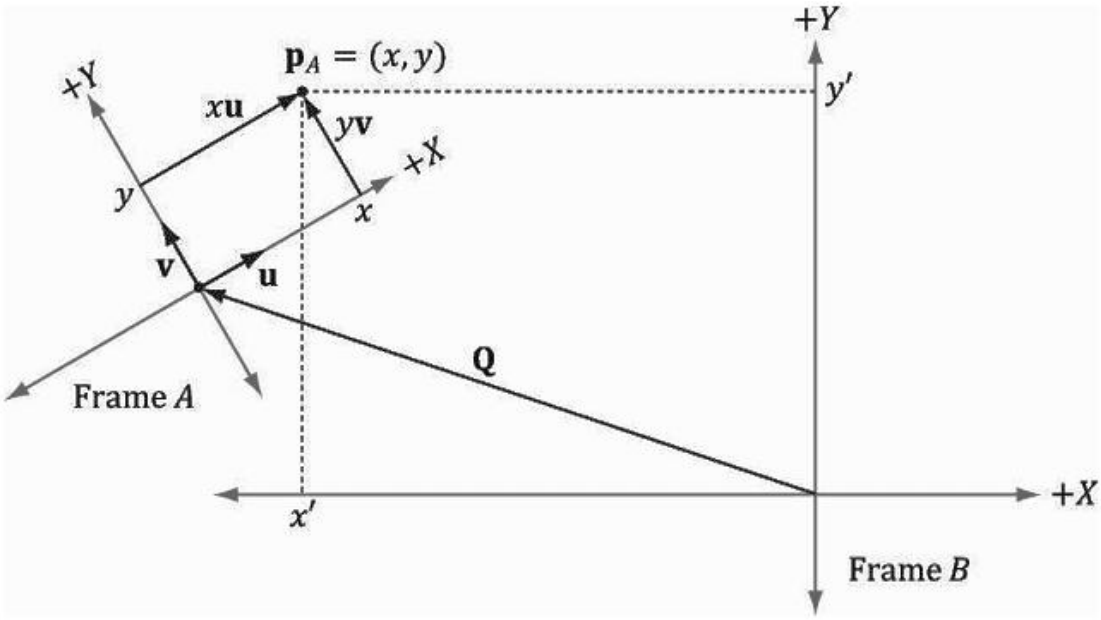



**图 3.12 求 p 相对于参考系 *B* 的几何坐标。**

这样，只要我们知道向量 **u**、**v** 的坐标以及相对于参考系 *B* 的原点，即 $\mathbf{u}_B=(u_x,u_y)$、$\mathbf{v}_B=(v_x,v_y)$、$\mathbf{Q}_B=(Q_x,Q_y)$，那么对于给出的任意坐标 $\mathbf{p}_A=(x,y)$，都可以求出 $\mathbf{p}_B=(x',y')$。

推导为 3D 向量，如果 $\mathbf{p}_A=(x,y,z)$，则

$$\mathbf{p}_B=x\mathbf{u}_B+y\mathbf{v}_B+z\mathbf{w}_B+\mathbf{Q}_B$$

其中 **u**、**v**、**w** 是单位向量，它们所指的方向分别与参考系 *A* 的 *x* 轴、*y* 轴、*z* 轴方向相同，**Q** 是参考系 *A* 的原点。

## 3.4.3 矩阵表示

回顾之前讲过的内容，向量和点的坐标转换变换分别为：

$(x',y',z')= x\mathbf{u}_B+y\mathbf{v}_B+z\mathbf{w}_B$ 用于向量

$(x',y',z')= x\mathbf{u}_B+y\mathbf{v}_B+z\mathbf{w}_B+\mathbf{Q}_B$ 用于点

如果我们使用齐次坐标，那么就可以通过一个方程同时处理向量和点：

$$(x', y', z', w) = x\mathbf{u}_B + y\mathbf{v}_B + z\mathbf{w}_B + w\mathbf{Q}_B \quad \text{（公式 3.8）}$$

当 *w* 设为 0 时，该方程可用于处理向量的坐标转换变换；当 *w* 设为 1 时，该方程可用于处理点的坐标转换变换。方程 3.8 的优点在于只要我们给出正确的 *w* 坐标，就可以同时完成向量和点的处理；不再需要定义两个方程（一个用于向量，另一个用于点）。方程 2.3 说明，我们可以用矩阵的形式来表示方程 3.8：

$$[x',y',z',w]=[x,y,z,w]\begin{bmatrix}\leftarrow \mathbf{u}_B \rightarrow\\ \leftarrow \mathbf{v}_B \rightarrow\\ \leftarrow \mathbf{w}_B \rightarrow\\ \leftarrow \mathbf{Q}_B \rightarrow\end{bmatrix}$$

$$=[x,y,z,w]\begin{bmatrix}u_x & u_y & u_z & 0\\ v_x & v_y & v_z & 0\\ w_x & w_y & w_z & 0\\ Q_x & Q_y & Q_z & 1\end{bmatrix}$$

（公式 3.9）

其中 $\mathbf{Q}_B=(Q_x,Q_y,Q_z,1)$、$\mathbf{u}_B=(u_x,u_y,u_z,0)$、$\mathbf{v}_B=(v_x,v_y,v_z,0)$、$\mathbf{w}_B=(w_x,w_y,w_z,0)$均为齐次坐标，它们描述了参考系 A 相对于参考系 B 的原点位置和坐标轴方向。我们将方程 3.9 中的 4×4 矩阵称为坐标转换矩阵或参考系转换矩阵，它可以将参考系 *A* 的坐标转换（或映射）为参考系 *B* 的坐标。

## 3.4.4 结合律与坐标转换矩阵

假设现在我们有 3 个参考系 *F*、*G*、*H*。而且，设 **A** 为从 *F* 到 *G* 的参考系转换矩阵，设 **B** 为从 *G* 到 *H* 的参考系转换矩阵。假设 $\mathbf{p}_F$ 为参考系 *F* 中的一个向量坐标，我们想要求解这个向量相对于参考系 *H* 的坐标，也就是求解 $\mathbf{p}_H$。一种方法是将每个转换矩阵逐一相乘：

$$(\mathbf{p}_F\mathbf{A})\mathbf{B}=\mathbf{p}_H$$
$$(\mathbf{p}_G)\mathbf{B}=\mathbf{p}_H$$

不过，由于矩阵乘法支持结合律，所以我们可以将$(\mathbf{p}_F\mathbf{A})\mathbf{B}=\mathbf{p}_H$改写为：

$$\mathbf{p}_F(\mathbf{AB})=\mathbf{p}_H$$

从这一意义上说，矩阵乘积 **C=AB** 可以被看成是从 *F* 直接向 *H* 的参考系转换矩阵；它将 **A** 和 **B** 产生的结果组合成了一个净矩阵。（这就像是把多个函数组合起来一样。）

这种方法有助于提升性能。比如，我们将两个连续的坐标转换变换应用于一个由 20,000 个点构成的 3D 物体。使用逐一相乘的方式，我们需要执行 20,000×2 次向量-矩阵乘法。而改用组合矩阵方式， 我们只需要执行 20,000 次向量-矩阵乘法和一次矩阵-矩阵乘法。很明显，一次额外的矩阵-矩阵乘法所产生的资源消耗微乎其微，而它们却能省去大量的向量-矩阵乘法。

**注意**：我们再次强调，矩阵乘法不支持交换律，**AB** 和 **BA** 表示不同的组合变换。矩阵相乘的顺序就是应用变换的顺序，这通常是一个不可交换的过程。

## 3.4.5 逆矩阵与坐标转换矩阵

假设 $\mathbf{p}_B$ 为向量 **p** 在参考系 *B* 中的坐标，**M** 为参考系 *A* 到参考系 *B* 的坐标转换矩阵；也就是 $\mathbf{p}_B=\mathbf{p}_A\mathbf{M}$。现在我们想求解 $\mathbf{p}_A$。换句话说，就是使用坐标转换矩阵把从 *A* 到 *B* 的映射反转为从 *B* 到 *A* 的映射。如果 **M** 是一个可逆矩阵（即，$\mathbf{M}^{-1}$ 存在），那么就可以求出结果。我们可以按照如下步骤求解 $\mathbf{p}_A$：

| $\mathbf{p}_B=\mathbf{p}_A\mathbf{M}$ | |
|---|---|

| | |
|---|---|
| $\mathbf{p}_B\mathbf{M}^{-1}=\mathbf{p}_A\mathbf{M}\mathbf{M}^{-1}$ | 等式两边同时乘以 $\mathbf{M}^{-1}$。 |
| $\mathbf{p}_B\mathbf{M}^{-1}=\mathbf{p}_A\mathbf{I}$ | 由逆矩阵的定义可知 $\mathbf{M}\mathbf{M}^{-1}=\mathbf{I}$。 |
| $\mathbf{p}_B\mathbf{M}^{-1}=\mathbf{p}_A$ | 由单位矩阵的定义可知 $\mathbf{p}_A\mathbf{I}=\mathbf{p}_A$。 |

这样，矩阵 $\mathbf{M}^{-1}$ 即为从 *B* 到 *A* 的坐标转换矩阵。

图 3.13 说明了坐标转换矩阵与它的逆矩阵之间的关系。另外请读者注意，在本书中使用的所有坐标系转换映射都是可逆的，所以我们不必担心其逆矩阵是否存在。

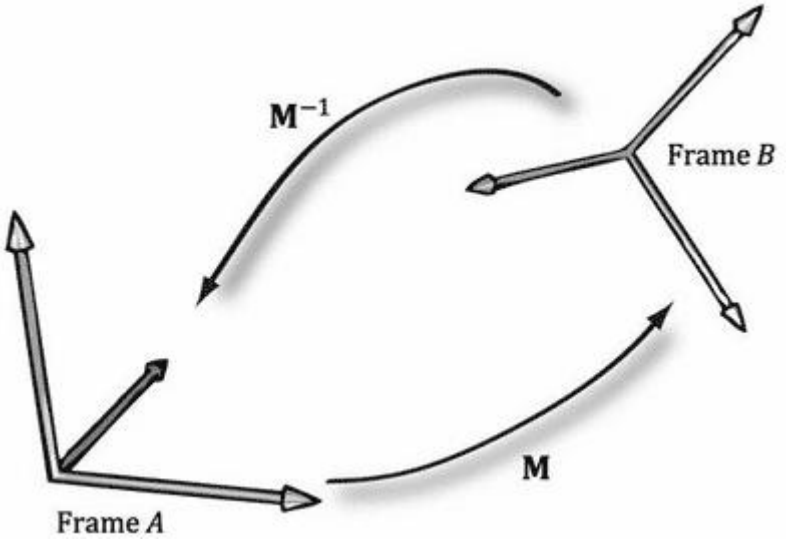



**图 3.13 M 为从 A 到 B 的映射，$\mathbf{M}^{-1}$ 为从 B 到 A 的映射。**

图 3.14 说明了逆矩阵的特性$(\mathbf{AB})^{-1}=\mathbf{B}^{-1}\mathbf{A}^{-1}$ 可以被用于任何坐标转换矩阵。

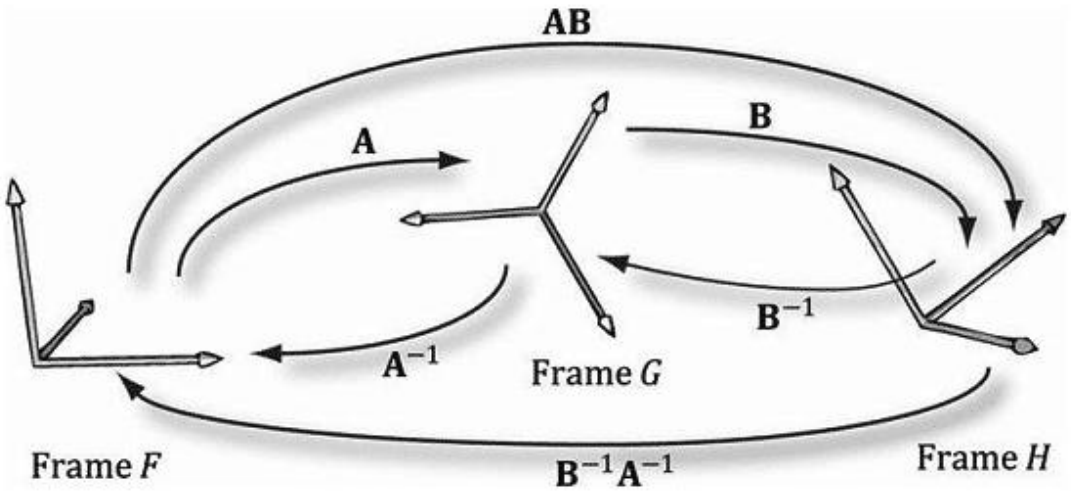



**图 3.14 A 为从 *F* 到 *G* 的映射，B 为从 *G* 到 *H* 的映射，AB 为从 *F* 直接到 *H* 的映射，$\mathbf{B}^{-1}$ 为从 *H* 到 *G* 的映射，$\mathbf{A}^{-1}$ 为从 *G* 到 *F* 的映射，$\mathbf{B}^{-1}\mathbf{A}^{-1}$ 为从 *H* 直接到 *F* 的映射。**

# 3.5 转换矩阵与坐标转换矩阵的对比

在前面的几节中，我们已经区分了 active 变换（缩放、旋转和平移）和坐标转换变换。本节我们将会证明两者在数学上是等价的，一个 active 变换可以解释为一个坐标转换变换，反之亦然。

图 3.15 说明了公式 3.7 中的行向量（仿射矩阵变换实现的平移加旋转）与公式 3.9 中的行向量（坐标转换矩阵）的相似之处。

这是说得通的。在坐标转换变换的情况中，参考系的位置和朝向都是不同的。所以，将一个参考系变换到另一个参考系的数学方程需要旋转和平移坐标，最终我们会得到相同的数学形式。无论哪种情况，我们都会得到相同的数字；区别只是在于我们解释变换的方式。在某些情况下，使用多个坐标系统更符合思维习惯，我们可以让物体自身保持不变，只是从一个参考系转换到另一个参考系，由于参考系发生了改变，因此物体的坐标也会随之改变（这种情况对应图 3.15b）。在另一些情况中，我们不想改变参考系，而只想在同一个参考系中对物体进行变换（这种情况对于图 3.15a）。

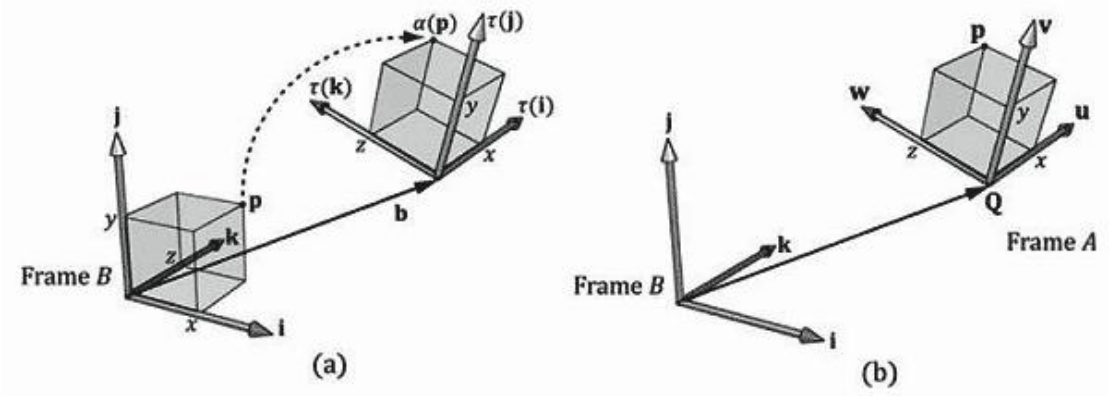



**图 3.15 从图中我们看到 b = Q，τ (i) = u，τ (j) = v，τ (k) = w。（a）中只使用一个坐标系统 *B*，我们在立方体上施加了一个仿射变换：α (*x*, *y*, *z*, w) = *x*τ (i) + *y*τ (j) + *z*τ (k) + *w*b，改变了它相对于坐标系 B 的位置和朝向；(b)中使用了两个坐标系 A 和 B。通过公式 $p_B = xu_B + y\,v_B + z\,w_B + w\,Q_B$（其中 $p_A = (x, y, z, w)$），立方体的各点的坐标可以从参考系 *A* 转换到参考系 *B*。从以上两种情况中我们可以得出α(p) = (*x*′, *y*′, *z*′, w) = $p_B$（$p_B$ 为相对于参考系 B 的坐标）。**

**注意**：上述讨论表明，我们可以将一个 active 变换组合（缩放，旋转，平移）解释为坐标系转换。这一点很重要，因为我们常常会将世界空间（第 5 章）的坐标变换矩阵定义为一个缩放、旋转、平移变换的组合。

# 3.6 XNA 数学库中的转换函数

下面我们总结一下 XNA 数学库中有关变换的函数：

```
// 创建一个缩放矩阵：
XMMATRIX XMMatrixScaling(
    FLOAT ScaleX,
    ScaleY,
    FLOAT ScaleZ);    // 缩放因子

// 从向量中的分量中创建一个缩放矩阵：
XMMATR IX XMMatrixScalingFromVector(
    FXMVECTOR Scale); // 缩放因子(sx,sy,sz)

// 创建一个绕 x 轴旋转的矩阵 ： Rx
XMMATRIX XMMatrixRotationX(
    FLOAT Angle); // 顺时针的旋转角度θ

// 创建一个绕 y 轴旋转的矩阵：Ry
XMMATRIX XMMatrixRotationY(
    FLOAT Angle); // 顺时针的旋转角度θ

// 创建一个绕 z 轴旋转的矩阵：Rz
XMMATRIX XMMatrixRotationZ(
    FLOAT Angle); // 顺时针的旋转角度θ

// 创建一个绕任意轴旋转的矩阵 ： Rn
XMMATRIX XMMatrixRotationAxis(
    FXMVECTOR Axis,     // 旋转轴
    FLOAT Angle);       // 顺时针的旋转角度θ

// 创建一个平移矩阵：
XMMATRIX XMMatrixTranslation(
    FLOAT OffsetX,
    FLOAT OffsetY,
    FLOAT OffsetZ); // 平移因子

// 从向量的分量中创建一个平移矩阵：
XMMATRIX XMMatrixTranslationFromVector(
    FXMVECTOR Offset); // 平移因子(tx , ty ,tz)

// 计算向量-矩阵乘积 vM：
XMVECTOR XMVector3Transform(
    FXMVECTOR V,    // Input v
```

```
    CXMMATRIX M);   // Input M

// 计算向量-矩阵乘积 vM，其中 vw = 1，用于变换点的坐标：
XMVECTOR XMVector3TransformCoord(
    FXMVECTOR V,   // Input v
    CXMMATRIX M);  // Input M

// 计算向量-矩阵乘积 vM，其中 vw = 0，用于变换向量：
XMVECTOR XMVector3TransformNormal(
    FXMVECTOR V,     // Input v
    CXMMATRIX M);    // Input M
```

对于最后两个函数 **XMVector3TransformCoord** 和 **XMVector3TransformNormal**，你无需显式地指定 $w$ 坐标。**XMVector3TransformCoord** 总是令使用 $v_w = 1$，而 **XMVector3TransformNormal** 总是令 $v_w = 0$。

# 3.7 小结

**1**．基本变换矩阵——缩放、旋转和平移——如下：

$$\mathbf{S}=\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{T}=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ b_x & b_y & b_z & 1 \end{bmatrix}$$

$$\mathbf{R}_n=\begin{bmatrix} c+(1-c)x^2 & (1-c)xy+sz & (1-c)xz-sy & 0 \\ (1-c)xy-sz & c+(1-c)y^2 & (1-c)yz+sx & 0 \\ (1-c)xz+sy & (1-c)yz-sx & c+(1-c)z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**2**．我们用 4×4 矩阵来描述变换，用 1×4 齐次坐标来描述点和向量，其中第 4 个分量 *w* 设为 1 时表示点，*w* 设为 0 时表示向量。通过这一方式，平移只会应用于点，而不会应用于向量。

**3**．如果一个矩阵的所有行向量都是单位向量且相互垂直，则该矩阵为正交矩阵。正交矩阵有一个特殊的性质，它的逆矩阵与转置矩阵相等，因此我们可以很容易地计算出它的逆矩阵。所有的旋转矩阵都是正交矩阵。

**4**．由矩阵乘法的结合律可知，我们可以将多个变换矩阵组合为一个净变换矩阵，最终得到的变换结果与执行多次单个矩阵乘法的变换结果相同。

**5**．设 $\mathbf{Q}_B$、$\mathbf{u}_B$、$\mathbf{v}_B$、$\mathbf{w}_B$ 分别表示参考系 *A* 相对于参考系 *B* 的原点位置及 *x*、*y*、*z* 坐标轴方向。如果向量或点 **p** 相对于参考系 *A* 的坐标为 $\mathbf{p}_A=(x,y,z)$，那该向量相对于参考系 *B* 的坐标为：

$\mathbf{p}_B=(x',y',z')=x\mathbf{u}_B+y\mathbf{v}_B+z\mathbf{w}_B$　　用于向量（方向和大小）

$\mathbf{p}_B=(x',y',z')=\mathbf{Q}_B+x\mathbf{u}_B+y\mathbf{v}_B+z\mathbf{w}_B$　用于位置向量（点）

使用齐次坐标可以将些坐标转换变换改写为矩阵的形式。

**6**．假设我们现在有 3 个参考系 *F*、*G*、*H*，并且，设 **A** 为从 *F* 到 *G* 的参考系转换矩阵，设 **B** 为从 *G* 到 *H* 的参考系转换矩阵。使用矩阵-矩阵乘法，矩阵 **C**=**AB** 可以被视为从 *F* 直接到 *H* 的参考系变换矩阵；也就是，矩阵-矩阵乘法将 **A** 和 **B** 所产生的变换结果组合成了一个净矩阵，记作：$\mathbf{p}_F(\mathbf{AB})=\mathbf{p}_H$。

**7**．如果矩阵 **M** 将参考系 *A* 的坐标映射为参考系 *B* 的坐标，那么矩阵 $\mathbf{M}^{-1}$ 可以将参考系 *B* 的坐标映射为参考系 *A* 的坐标。

**8**．一个 active 变换也可以理解为一个坐标系转换变换，反之亦然。在某些情况下，使用多个坐标系统更符合思维习惯，我们可以让物体自身保持不变，只是从一个参考系转换到另一个参考系，由于参考系发生了改变，因此物体的坐标也会随之改变。在另一些情况中，我们不想改变参考系，而只想在同一个参考系中对物体进行变换。

# 4.1 准备工作

Direct3D 的初始化过程要求我们熟悉一些基本的 Direct3D 类型和基本绘图概念；本章第一节会向读者介绍些必要的基础知识。然后我们会详细讲解 Direct3D 初始化过程中的每一个必要步骤，并顺便介绍一下实时绘图应用程序必须使用的精确计时和时间测量。最后，我们将讨论一下示例框架代码，它是本书所有演示程序使用的统一编程接口。

## 学习目标：

- 对 Direct3D 在规划调度 3D 硬件方面所起的作用有一个基本了解。
- 理解 COM 在 Direct3D 运行时起到的作用。
- 学习基本的绘图概念，比如 2D 图像的存储方式、页面翻转、深度缓存和多重采样。
- 学习如何使用性能计数器函数来获取高精度的计时器读数。
- 阐述 Direct3D 的初始化过程。
- 熟悉本书所有演示程序采用的通用应用程序框架结构。

Direct3D 的初始化过程要求我们熟悉一些基本的绘图概念和 Direct3D 类型。我们会在本节讲解这些概念和类型，以使读者可以顺利地阅读之后的章节。

### 4.1.1 Direct3D 概述

Direct3D 是一种底层绘图 **API（application programming interface，应用程序接口）**，它可以让我们可以通过 3D 硬件加速绘制 3D 世界。从本质上讲，Direct3D 提供的是一组软件接口，我们可以通过这组接口来控制绘图硬件。例如，要命令绘图设备清空渲染目标（例如屏幕），我们可以调用 Direct3D 的 **ID3D11DeviceContext::ClearRenderTargetView** 方法来完成这一工作。Direct3D 层位于应用程序和绘图硬件之间，这样我们就不必担心 3D 硬件的实现细节，只要设备支持 Direct3D 11，我们就可以通过 Direct3D 11 API 来控制 3D 硬件了。

支持 Direct3D 11 的设备必须支持 Direct3D 11 规定的整个功能集合以及少数的额外附加功能（有一些功能，比如多重采样数量，仍然需要以查询方式实现，这是因为不同的 Direct3D 硬件这个值可能并不一样）。在 Direct3D 9 中，设备可以只支持 Direct3D 9 的部分功能；所以，当一个 Direct3D 9 应用程序要使用某一特性时，应用程序就必须先检查硬件是否支持该特性。如果要调用的是一个不为硬件支持 Direct3D 函数，那应用程序就会出错。而在 Direct3D 11 中，不需要再做这种设备功能检查，因为 Direct3D 11 强制要求设备实现 Direct3D 11 规定的所有功能特性。

### 4.1.2 COM

组件对象模型（COM）技术使 DirectX 独立于任何编程语言，并具有版本向后兼容的特性。我们经常把 COM 对象称为接口，并把它当成一个普通的 C++类来使用。当使用 C++编写 DirectX 程序时，许多 COM 的底层细节都不必考虑。唯一需要知道的一件事情是，我们必须通过特定的函数或其他的 COM 接口方法来获取指向 COM 接口的指针，而不能用 C++

的 **new** 关键字来创建 COM 接口。另外，当我们不再使用某个接口时，必须调用它的 **Release** 方法来释放它（所有的 COM 接口都继承于 **IUnknown** 接口，而 **Release** 方法是 **IUnknown** 接口的成员），而不能用 **delete** 语句——COM 对象在其自身内部实现所有的内存管理工作。

当然，有关 COM 的细节还有很多，但是在实际工作中只需知道上述内容就足以有效地使用 DirectX 了。

**注意**：COM 接口都以大写字母“I”为前缀。例如，表示 2D 纹理的接口为 **ID3D11Texture2D**。

### 4.1.3 纹理和数据资源格式

2D 纹理（texture）是一种数据元素矩阵。2D 纹理的用途之一是存储 2D 图像数据，在纹理的每个元素中存储一个像素颜色。但这不是纹理的唯一用途；例如， 有一种称为法线贴图映射（normal mapping）的高级技术在纹理元素中存储的不是颜色，而是 3D 向量。因此，从通常意义上讲，纹理用来存储图像数据，但是在实际应用中纹理可以有更广泛的用途。1D 纹理类似于一个 1D 数据元素数组，3D 纹理类似于一个 3D 数据元素数组。但是在随后的章节中我们会讲到，纹理不仅仅是一个数据数组；纹理可以带有多级渐近纹理层（mipmap level），GPU 可以在纹理上执行特殊运算，比如使用过滤器（filter）和多重采样（multisampling）。此外，不是任何类型的数据都能存储到纹理中的；纹理只支持特定格式的数据存储，这些格式由 **DXGI_FORMAT** 枚举类型描述。一些常用的格式如下：

- **DXGI_FORMAT_R32G32B32_FLOAT**：每个元素包含 3 个 32 位浮点分量。
- **DXGI_FORMAT_R16G16B16A16_UNORM**：每个元素包含 4 个 16 位分量，分量的取值范围在[0,1]区间内。
- **DXGI_FORMAT_R32G32_UINT**：每个元素包含两个 32 位无符号整数分量。
- **DXGI_FORMAT_R8G8B8A8_UNORM**：每个元素包含 4 个 8 位无符号分量，分量的取值范围在[0,1]区间内。
- **DXGI_FORMAT_R8G8B8A8_SNORM**：每个元素包含 4 个 8 位有符号分量，分量的取值范围在[−1,1] 区间内。
- **DXGI_FORMAT_R8G8B8A8_SINT**：每个元素包含 4 个 8 位有符号整数分量，分量的取值范围在[−128, 127] 区间内。
- **DXGI_FORMAT_R8G8B8A8_UINT**：每个元素包含 4 个 8 位无符号整数分量，分量的取值范围在[0, 255]区间内。

注意，字母 R、G、B、A 分别表示 red（红）、green（绿）、blue（蓝）和 alpha（透明度）。每种颜色都是由红、绿、蓝三种基本颜色组成的（例如，黄色是由红色和绿色组成的）。alpha 通道（或 alpha 分量）用于控制透明度。不过，正如我们之前所述，纹理存储的不一定是颜色信息；例如，格式 **DXGI_FORMAT_R32G32B32_FLOAT** 包含 3 个浮点分量，可以存储一个使用浮点坐标的 3D 向量。另外，还有一种**弱类型（typeless）**格式，可以预先分配内存空间，然后在纹理绑定到管线时再指定如何重新解释数据内容（这一过程与 C++ 中的数据类型转换颇为相似）；例如，下面的弱类型格式为每个元素预留 4 个 8 位分量，且不指定数据类型（例如：整数、浮点数、无符号整数）：

**DXGI_FORMAT_R8G8B8A8_TYPELESS**

### 4.1.4 交换链和页面翻转

为了避免在动画中出现闪烁，最好的做法是在一个离屏（off-screen）纹理中执行所有的动画帧绘制工作，这个离屏纹理称为**后台缓冲区（back buffer）**。当我们在后台缓冲区中完成给定帧的绘制工作后，便可以将后台缓冲区作为一个完整的帧显示在屏幕上；使用这种方法，用户不会察觉到帧的绘制过程，只会看到完整的帧。从理论上讲，将一帧显示到屏幕上所消耗的时间小于屏幕的垂直刷新时间。硬件会自动维护两个内置的纹理缓冲区来实现这一功能，这两个缓冲区分别称为**前台缓冲区（front buffer）**和后台缓冲区。前台缓冲区存储了当前显示在屏幕上的图像数据，而动画的下一帧会在后台缓冲区中执行绘制。当后台缓冲区的绘图工作完成之后，前后两个缓冲区的作用会发生翻转：后台缓冲区会变为前台缓冲区，而前台缓冲区会变为后台缓冲区，为下一帧的绘制工作提前做准备。我们将前后缓冲区功能互换的行为称做**呈现（presenting）**。提交是一个运行速度很快的操作，因为它只是将前台缓冲区的指针和后台缓冲区的指针做了一个简单的交换。图 4.1 说明了这一过程。

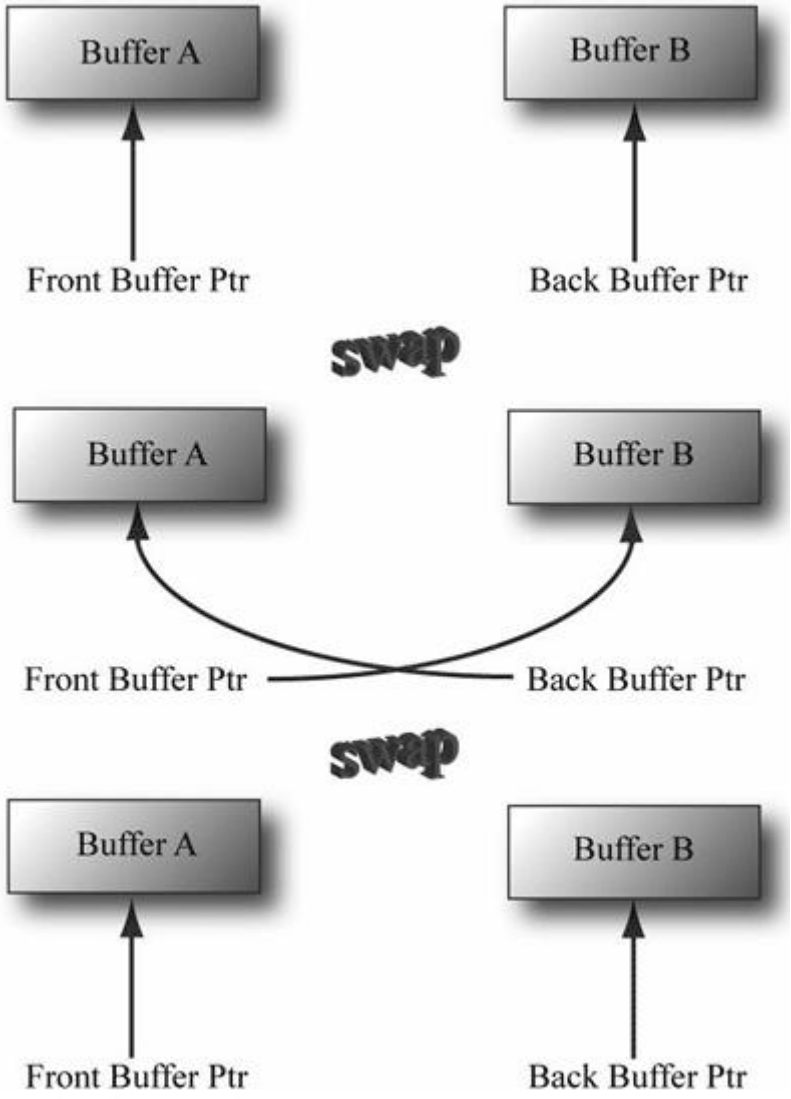



**图 4.1：我们首先渲染缓冲区 B，它是当前的后台缓冲区。一旦帧渲染完成，前后缓冲区的指针会相互交换，缓冲区 B 会变为前台缓冲区，而缓冲区 A 会变为新的后台缓冲区。之后，我们将在缓冲区 A 中进行下一帧的渲染。一旦帧渲染完成，前后缓冲区的指针会再**

**次进行交换，缓冲区 A 会变为前台缓冲区，而缓冲区 B 会再次变为后台缓冲区。**

前后缓冲区形成了一个**交换链（swap chain）**。在 Direct3D 中，交换链由 **IDXGISwapChain** 接口表示。该接口保存了前后缓冲区纹理，并提供了用于调整缓冲区尺寸的方法（**IDXGISwapChain::ResizeBuffers**）和呈现方法（**IDXGISwapChain::Present**）。我们会在 4.4 节中详细讨论些方法。

使用（前后）两个缓冲区称为**双缓冲（double buffering）**。缓冲区的数量可多于两个；比如，当使用三个缓冲区时称为**三缓冲（triple buffering**）。不过，两个缓冲区已经足够用了。

**注意**：虽然后台缓冲区是一个纹理（纹理元素称为 texel），但是我们更习惯于将纹理元素称为像素（pixel），因为后台缓冲区存储的是颜色信息。有时，即使纹理中存储的不是颜色信息，人们还是会将纹理元素称为像素（例如，“法线贴图像素”）。

## 4.1.5 深度缓冲区

**深度缓冲区（depth buffer）**是一个不包含图像数据的纹理对象。在一定程度上，深度信息可以被认为是一种特殊的像素。常见的深度值范围在 0.0 到 1.0 之间，其中 0.0 表示离观察者最近的物体，1.0 表示离观察者最远的物体。深度缓冲区中的每个元素与后台缓冲区中的每个像素一一对应（即，后台缓冲区的第 ij 个元素对应于深度缓冲区的第 ij 个元素）。所以，当后台缓冲区的分辨率为 1280×1024 时，在深度缓冲区中有 1280×1024 个深度元素。

**图 4.2 彼此遮挡的一组物体**

图 4.2 是一个简单的场景，其中一些物体挡住了它后面的一些物体的一部分区域。为了判定物体的哪些像素位于其他物体之前，Direct3D 使用了一种称为**深度缓存（depth buffering）**或 **z 缓存（z-buffering）**的技术。我们所要强调的是在使用深度缓存时，我们不必关心所绘物体的先后顺序。

**注意**：要处理深度的问题，有人可能会建议按照从远至近的顺序绘制场景中的物体。使用这种方法，离得近的物体会覆盖在离得远的物体之上，这样就会产生正确的绘制结果，这也是画家作画时用到的方法。但是，这种方法会导致另一个问题——如何将大量的物体和相

交的几何体按从远到近的方式进行排序？此外，图形硬件本身就提供了深度缓存供我们使用，因此我们不会采用画家算法。

为了说明深度缓存的工作方式，让我们来看一个例子。如图 4.3 所示，它展示的是观察者看到的立体空间（左图）以及该立体空间的 2D 侧视图（右图）。从这个图中我们可以发现，3 个不同的像素会被渲染到视图窗口的同一个像素点 P 上。（当然，我们知道只有最近的像素会被渲染到 P 上，因为它挡住了后面的其他像素，可是计算机不知道这些事情。）首先，在渲染之前，我们必须把后台缓冲区清空为一个默认颜色（比如黑色或白色），把深度缓冲区清空为默认值——通常设为 1.0（像素所具有的最远深度值）。

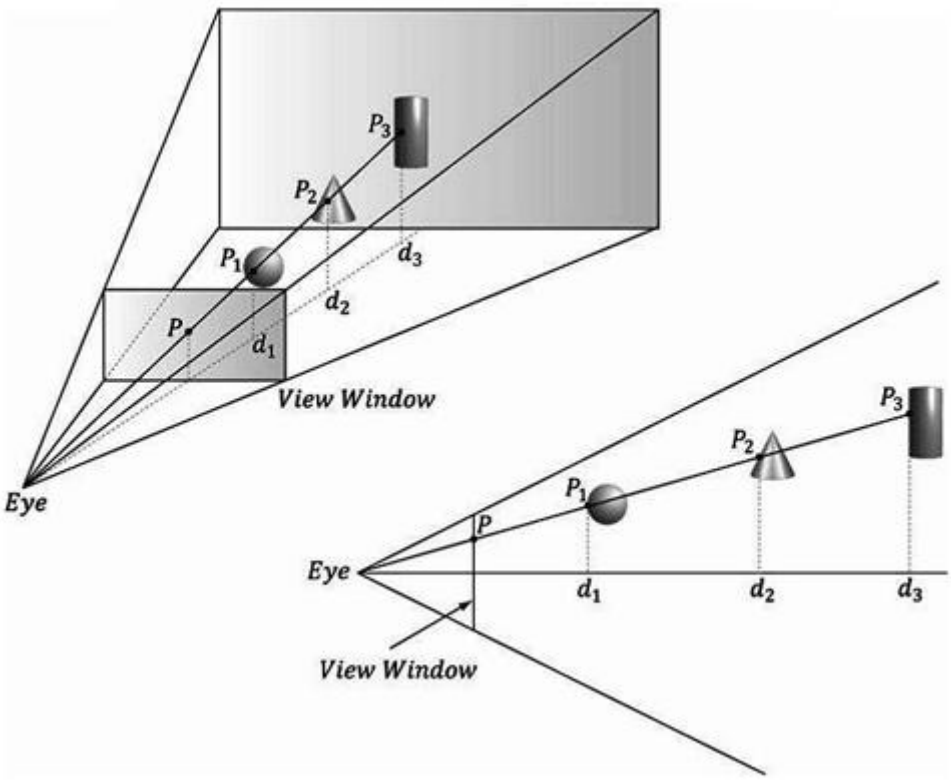



**图 4.3：视图窗口相当于从 3D 场景生成的 2D 图像（后台缓冲区）。我们看到，有 3 个不同的像素可以被投影到像素 P 上。直觉告诉我们，$P_1$ 是 P 的最终颜色，因为它离观察者最近，而且遮挡了其他两个像素。深度缓冲区算法提供了一种可以在计算机上实现的判定过程。注意，我们所说的深度值是相对于观察坐标系而言的。实际上，当深度值存入深度缓冲区时，它会被规范到[0.0,1.0]区间内。**

现在，假设物体的渲染顺序依次为：圆柱体、球体和圆锥体。下面的表格汇总了在绘制些物体时像素 P 及相关深度值的变化过程；其他像素的处理过程与之类似。

**表 4.1**

| 操作 | P | d | 说明 |
|---|---|---|---|
| **清空** | 黑色 | 1.0 | 初始化像素以及相应的深度元素。 |
| **绘制圆柱体** | $P_3$ | $d_3$ | 因为 $d_3 \leqslant d = 1.0$，所以深度测试通过，更新缓冲区，设置 $P = P_3$、$d = d_3$。 |
| **绘制球体** | $P_1$ | $d_1$ | 因为 $d_1 \leqslant d = d_3$，所以深度测试通过，更新缓冲区，设置 $P = P_1$、$d = d_1$。 |
| **绘制圆锥体** | $P_1$ | $d_1$ | 因为 $d_2 > d = d_1$，所以深度测试未通过，不更新缓冲区。 |

从上表可以看到，当我们发现某个像素具有更小的深度值时，就更新该像素以及它在深

度缓冲区中的相应深度值。通过一方式，在最终得到的渲染结果中只会包含那些离观察者最近的像素。（如果读者对此仍有疑虑，那么可以试着交换本例的绘图顺序，看看得到的计算结果是否相同。）

综上所述，深度缓冲区用于为每个像素计算深度值和实现深度测试。深度测试通过比较像素深度来决定是否将该像素写入后台缓冲区的特定像素位置。只有离观察者最近的像素才会胜出，成为写入后台缓冲区的最终像素。这很容易理解，因为离观察者最近的像素会遮挡它后面的其他像素。

深度缓冲区是一个纹理，所以在创建它时必须指定一种数据格式。用于深度缓存的格式如下：

- **DXGI_FORMAT_D32_FLOAT_S8X24_UINT**：32 位浮点深度缓冲区。为模板缓冲区预留 8 位（无符号整数），每个模板值的取值范围为[0,255]。其余 24 位闲置。
- **DXGI_FORMAT_D32_FLOAT**：32 位浮点深度缓冲区。
- **DXGI_FORMAT_D24_UNORM_S8_UINT**：无符号 24 位深度缓冲区，每个深度值的取值范围为[0,1]。为模板缓冲区预留 8 位（无符号整数），每个模板值的取值范围为[0,255]。
- **DXGI_FORMAT_D16_UNORM**：无符号 16 位深度缓冲区，每个深度值的取值范围为[0,1]。

**注意**：模板缓冲区对应用程序来说不是必须的，但是如果用到了模板缓冲区，那么模板缓冲区必定是与深度缓冲区存储在一起的。例如，32 位格式 **DXGI_FORMAT_D24_UNORM_S8_UINT** 使用 24 位用于深度缓冲区，8 位用于模板缓冲区。 所以，将深度缓冲区称为“深度/模板缓冲区”更为合适。模板缓冲区是一个比较高级的主题，我们会在第 10 章讲解模板缓冲区的用法。

### 4.1.6 纹理资源视图

纹理可以被绑定到**渲染管线（rendering pipeline）**的不同**阶段（stage）**；例如，比较常见的情况是将纹理作为渲染目标（即，Direct3D 渲染到纹理）或着色器资源（即，在着色器中对纹理进行采样）。当创建用于这两种目的的纹理资源时，应使用绑定标志值：

**D3D11_BIND_RENDER_TARGET | D3D10_BIND_SHADER_RESOURCE**

指定纹理所要绑定的两个管线阶段。其实，资源不能被直接绑定到一个管线阶段；我们只能把与资源关联的资源视图绑定到不同的管线阶段。无论以哪种方式使用纹理，Direct3D 始终要求我们在初始化时为纹理创建相关的**资源视图（resource view）**。这样有助于提高运行效率，正如 SDK 文档指出的那样：“运行时环境与驱动程序可以在视图创建执行相应的验证和映射，减少绑定时的类型检查”。所以，当把纹理作为一个渲染目标和着色器资源时，我们要为它创建两种视图：渲染目标视图（**ID3D11RenderTargetView**）和着色器资源视图（**ID3D11ShaderResourceView**）。资源视图主要有两个功能：（1）告诉 Direct3D 如何使用资源（即，指定资源所要绑定的管线阶段）；（2）如果在创建资源时指定的是弱类型（typeless）格式，那么在为它创建资源视图时就必须指定明确的资源类型。对于弱类型格式，纹理元素可能会在一个管线阶段中视为浮点数，而在另一个管线阶段中视为整数。

为了给资源创建一个特定视图，我们必须在创建资源时使用特定的绑定标志值。例如，如果在创建资源没有使用 **D3D11_BIND_DEPTH_STENCIL** 绑定标志值（该标志值表示纹理将作为一个深度/模板缓冲区绑定到管线上），那我们就无法为该资源创建 **ID3D11DepthStencilView 视图**。只要你试一下就会发现 Direct3D 会给出如下调试错误：

**ERROR: ID3D11Device::CreateDepthStencilView: A DepthStencilView cannot be**

**created of a Resource that did not specify D3D10_BIND_DEPTH_STENCIL.**

我们会在本章的 4.2 节中看到用来创建渲染目标视图和深度/模板视图的代码。在第 8 章中看到用于创建着色器资源视图的代码。本书随后的许多例子都有会把纹理用作渲染目标和着色器资源。

**注意**：2009 年 8 月的 SDK 文档指出：“当创建资源时，为资源指定强类型（fully-typed）格式，把资源的用途限制在格式规定的范围内，有利于提高运行时环境对资源的访问速度……”。所以，你只应该在真正需要弱类型资源时（使用弱类型的优点是可以使用不同的视图将数据用于不同的用途），才创建弱类型资源；否则，应尽量创建强类型资源。

## 4.1.7 多重采样

因为计算机显示器上的像素分辨率有限，所以当我们绘制一条任意直线时，该直线很难精确地显示在屏幕上。图 4.4 中的第一条直线说明了“阶梯”（**aliasing，锯齿**）效应，当使用像素矩阵近似地表示一条直线时就会出现这种现象，类似的锯齿也会发生在三角形的边缘上。

**图 4.4：我们可以看到，第一条直线带有明显的锯齿（当使用像素矩阵近似地表示一条直线时就会出现阶梯效应）。而第二条直线使用了抗锯齿技术，通过对一个像素周围的邻接像素进行采样得到该像素的最终颜色；这样可以形成一条较为平滑的直线，使阶梯效果得到缓解。**

通过提高显示器的分辨率，缩小像素的尺寸，也可以有效地缓解一问题，使阶梯效应明显降低。

当无法提高显示器分辨率或分辨率不够高时，我们可以使用**抗锯齿（antialiasing）**技术。其中的一种技术叫做**超级采样（supersampling）**，它把后台缓冲和深度缓冲的大小提高到屏幕分辨率的 4 倍。3D 场景会以这个更大的分辨率渲染到后台缓存中，当在屏幕上呈现后台缓冲时，后台缓冲会将 4 个像素的颜色取平均值后得到一个像素的最终颜色。从效果上来说，超级采样的工作原理就是以软件的方式提升分辨率。

超级采样代价昂贵，因为它处理的像素数量和所需的内存数量增加为原来的 4 倍。Direct3D 支持另一种称为**多重采样（multisampling）**的抗锯齿技术，它通过对一个像素的子像素进行采样计算出该像素的最终颜色，比超级采样节省资源。假如我们使用的是 4X 多重采样（每个像素采样 4 个邻接像素），多重采样仍然会使用屏幕分辨率 4 倍大小的后台缓冲和深度缓冲，但是，不像超级采样那样计算每个子像素的颜色，而是只计算像素中心颜色一次，然后基于子像素的可见性（基于子像素的深度/模板测试）和范围（子像素中心在多边形之外还是之内）共享颜色信息。图 4.5 展示了这样的一个例子。

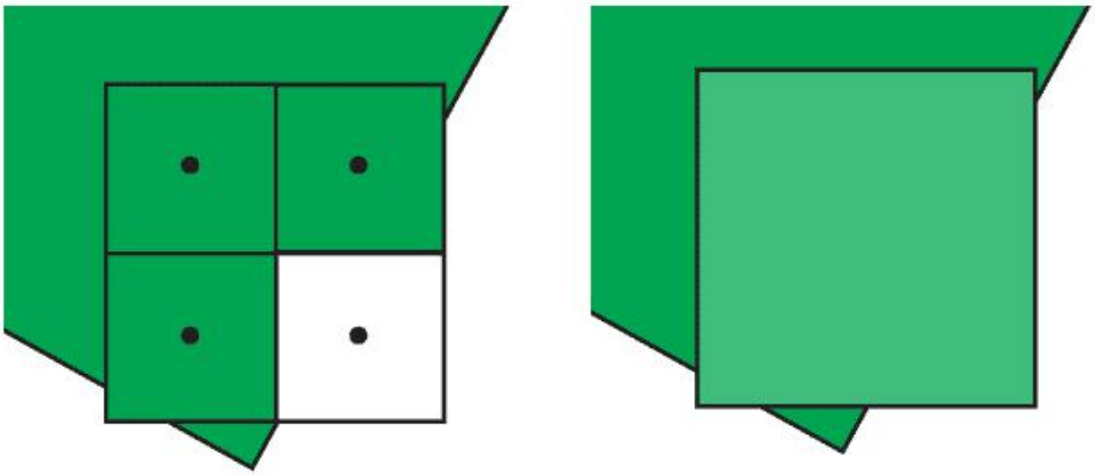


图 4.5：如图（a）所示，一个像素与多边形的边缘相交，像素中心的绿颜色存储在可见的三个子像素中，而第 4 个子像素没有被多边形覆盖，因此不会被更新为绿色，它仍保持为原来绘制的几何体颜色或 **Clear** 操作后的颜色。如图（b）所示，要获得最后的像素颜色，我们需要对 4 个子像素（3 个绿色和一个白色）取平均值，获得淡绿色，通过这个操作，可以减弱多边形边缘的阶梯效果，实现更平滑的图像。

**注意**：supersampling 与 multisampling 的关键区别在于：使用 supersampling 时，图像的颜色需要通过每个子像素的颜色计算得来，而每个子像素颜色可能不同；使用 multisampling（图 4.5）时，每个像素的颜色只计算一次，这个颜色会填充到所有可见的、被多边形覆盖的子像素中，即这个颜色是共享的。因为计算图像的颜色是图形管线中最昂贵的操作之一，因此 multisampling 相比 supersampling 而言节省的资源是相当可观的。但是，supersampling 更为精确，这是 multisampling 做不到的。

**注意**：在图 4.5 中，我们用标准的网格图形表示一个像素的 4 个子像素，但由于硬件的不同，实际的子像素放置图形也是不同的，Direct3D 并不定义子像素的放置方式，在特定情况下，某些放置方式会优于其他的放置方式。

## 4.1.8 Direct3D 中的多重采样

在下一节中， 我们要填充一个 **DXGI_SAMPLE_DESC** 结构体。该结构体包含两个成员，其定义如下：

```
typedef struct DXGI_SAMPLE_DESC {
    UINT Count;
    UINT Quality;
} DXGI_SAMPLE_DESC, *LPDXGI_SAMPLE_DESC;
```

**Count** 成员用于指定每个像素的采样数量，**Quality** 成员用于指定希望得到的质量级别（不同硬件的质量级别表示的含义不一定相同）。质量级别越高，占用的系统资源就越多，所以我们必须在质量和速度之间权衡利弊。质量级别的取值范围由纹理格式和单个像素的采样数量决定。我们可以使用如下方法，通过指定纹理格式和采样数量来查询相应的质量级别：

```
HRESULT ID3D11Device::CheckMultisampleQualityLevels(
DXGI_FORMAT Format, UINT SampleCount, UINT *pNumQualityLevels);
```

如果纹理格式和采样数量的组合不被设备支持，则该方法返回 0。反之，通过

pNumQualityLevels 参数返回符合给定的质量等级数值。有效的质量级别范围为 0 到 pNumQualityLevels−1。

采样的最大数量可以由以下语句定义：

```
#define D3D11_MAX_MULTISAMPLE_SAMPLE_COUNT(32)
```

采样数量通常使用 4 或 8，可以兼顾性能和内存消耗。如果你不使用多重采样，可以将采样数量设为 1，将质量级别设为 0。所有符合 Direct3D 11 功能特性的设备都支持用于所有渲染目标格式的 4X 多重采样。

**注意**：我们需要为交换链缓冲区和深度缓冲区各填充一个 **DXGI_SAMPLE_DESC** 结构体。当创建后台缓冲区和深度缓冲区时，必须使用相同的多重采样设置；具体的代码会在下一节给出。

## 4.1.9 特征等级

Direct3D 11 提出了特征等级（feature levels，在代码中由枚举类型 **D3D_FEATURE_LEVEL** 表示）的概念，对应了定义了 d3d11 中定义了如下几个等级以代表不同的 d3d 版本：

```
typedef enum D3D_FEATURE_LEVEL {
  D3D_FEATURE_LEVEL_9_1 = 0x9100,
  D3D_FEATURE_LEVEL_9_2 = 0x9200,
  D3D_FEATURE_LEVEL_9_3 = 0x9300,
  D3D_FEATURE_LEVEL_10_0 = 0xa000,
  D3D_FEATURE_LEVEL_10_1 = 0xa100,
  D3D_FEATURE_LEVEL_11_0 = 0xb000
} D3D_FEATURE_LEVEL;
```

特征等级定义了一系列支持不同 d3d 功能的相应的等级(每个特征等级支持的功能可参见 SDK 文档),用意即如果一个用户的硬件不支持某一特征等级,程序可以选择较低的等级。例如，为了支持更多的用户，应用程序可能需要支持 Direct3D 11，10.1，9.3 硬件。程序会从最新的硬件一直检查到最旧的,即首先检查是否支持 Direct3D 11,第二检查 Direct3D 10.1，然后是 Direct3D 10，最后是 Direct3D 9。要设置测试的顺序，可以使用下面的特征等级数组（数组内元素的顺序即特征等级测试的顺序）：

```
D3D_FEATURE_LEVEL featureLevels [4] =
{
  D3D_FEATURE_LEVEL_11_0,   //  First check D3D 11 support
  D3D_FEATURE_LEVEL_10_1,   //  Second check D3D 10.1 support
  D3D_FEATURE_LEVEL_10_0,   //  Next,check D3D 10 support
  D3D_FEATURE_LEVEL_9_3    //  Finally,check D3D 9.3 support
} ;
```

这个数组可以放置在 Direct3D 初始化方法（4.2.1 节）中，方法会输出数组中第一个可被支持的特征等级。例如，如果 Direct3D 报告数组中第一个可被支持的特征等级是 **D3D_FEATURE_LEVEL_10_0**，程序就会禁用 Direct3D 11 和 Direct3D 10.1 的特征，而使用 Direct3D 10 的绘制路径。本书中我们要求必须能支持 **D3D_FEATURE_LEVEL_11_0**。

# 4.2 对 Direct3D 进行初始化

下面的各小节将讲解如何初始化 Direct3D。我们将 Direct3D 的初始化过程分为如下几个步骤：

1．使用 **D3D11CreateDevice** 方法创建 **ID3D11Device** 和 **ID3D11DeviceContext**。

2．使用 **ID3D11Device::CheckMultisampleQualityLevels** 方法检测设备支持的 4X 多重采样质量等级。

3．填充一个 **IDXGI_SWAP_CHAIN_DESC** 结构体，该结构体描述了所要创建的交换链的特性。

4．查询 **IDXGIFactory** 实例，这个实例用于创建设备和一个 **IDXGISwapChain** 实例。

5．为交换链的后台缓冲区创建一个渲染目标视图。

6．创建深度/模板缓冲区以及相关的深度/模板视图。

7．将渲染目标视图和深度/模板视图绑定到渲染管线的输出合并阶段，使它们可以被 Direct3D 使用。

8．设置视口。

## 4.2.1 创建设备（Device）和上下文（Context）

要初始化 Direct3D，首先需要创建 Direct3D 11 设备（**ID3D11Device**）和上下文（**ID3D11DeviceContext**）。它们是是最重要的 Direct3D 接口，可以被看成是物理图形设备硬件的软控制器；也就是说，我们可以通过该接口与硬件进行交互，命令硬件完成一些工作（比如：在显存中分配资源、清空后台缓冲区、将资源绑定到各种管线阶段、绘制几何体）。具体而言：

1．**ID3D11Device** 接口用于检测显示适配器功能和分配资源。

2．**ID3D11DeviceContext** 接口用于设置管线状态、将资源绑定到图形管线和生成渲染命令。

设备和上下文可用如下函数创建：

```
HRESULT  D3D11CreateDevice (
    IDXGIAdapter  *pAdapter,
    D3D_DRIVER_TYPE  DriverType,
    HMODULE  Software ,
    UINT  Flags ,
    CONST  D3D_FEATURE_LEVEL  *pFeatureLevels ,
    UINT  FeatureLevels ,
    UINT  SDKVersion,
    ID3D11Device  **ppDevice ,
    D3D_FEATURE_LEVE L  *pFeatureLevel,
    ID3D11DeviceContext  **ppImmediateContext
);
```

1．**pAdapter**：指定要为哪个物理显卡创建设备对象。当该参数设为空值时，表示使用主显卡。在本书的示例程序中，我们只使用主显卡。

2．**DriverType**：一般来讲，该参数总是指定为 **D3D_DRIVER_TYPE_HARDWARE**，

表示使用 3D 硬件来加快渲染速度。但是，也可以有两个其他选择：

**D3D_DRIVER_TYPE_REFERENCE**：创建所谓的引用设备（reference device）。引用设备是 Direct3D 的软件实现，它具有 Direct3D 的所有功能（只是运行速度非常慢，因为所有的功能都是用软件来实现的）。引用设备随 DirectX SDK 一起安装，只用于程序员，而不应该用于程序发布。使用引用设备有两个原因：

- 测试硬件不支持的代码；例如，在一块不支持 Direct3D 11 的显卡上测试一段 Direct3D 11 的代码。
- 测试驱动程序缺陷。当代码能在引用设备上正常运行，而不能在硬件上正常工作时，说明硬件的驱动程序可能存在缺陷。

**D3D_DRIVER_TYPE_SOFTWARE**：创建一个用于模拟 3D 硬件的软件驱动器。要使用软件驱动器，你必须自己创建一个，或使用第三方的软件驱动器。与下面要说的 WARP 驱动器不同，Direct3D 不提供软件驱动器。

**D3D_DRIVER_TYPE_WARP**：创建一个高性能的 Direct3D 10.1 软件驱动器。WARP 代表 Windows Advanced Rasterizati on Platform。因为 WARP 不支持 Direct3D 11，因此我们对它不感兴趣。

3．**Software**：用于支持软件光栅化设备（software rasterizer）。我们总是将该参数设为空值，因为我们使用硬件进行渲染。如果读者想要使用这一功能，那么就必须先安装一个软件光栅化设备。

4．**Flags**：可选的设备创建标志值。当以 release 模式生成程序时，该参数通常设为 0（无附加标志值）；当以 debug 模式生成程序时，该参数应设为：

**D3D11_CREATE_DEVICE_DEBUG**：用以激活调试层。当指定调试标志值后，Direct3D 会向 VC++的输出窗口发送调试信息；图 4.6 展示了输出错误信息的一个例子。

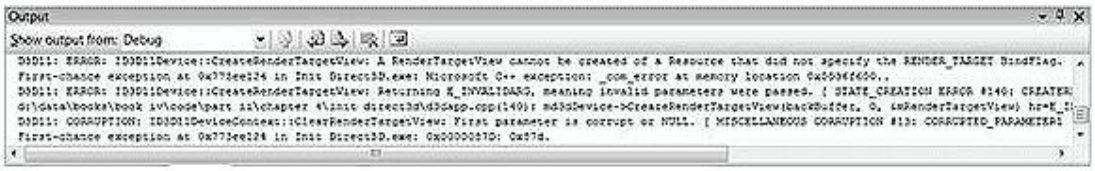



**图 4.6　Direct3D 11 调试输出的一个例子。**

5．**pFeatureLevels**：**D3D_FEATURE_LEVEL** 数组，元素的顺序表示要特征等级（见§4.1.9）的测试顺序。将这个参数设置为 null 表示选择可支持的最高等级。

6．**FeatureLevels**：**pFeatureLevels** 数组中的元素 **D3D_FEATURE_LEVELs** 的数量，若 **pFeatureLevels** 设置为 null，则这个值为 0。

7．**SDKVersion**：始终设为 **D3D11_SDK_VERSION**。

8．**ppDevice**：返回创建后的设备对象。

9．**pFeatureLevel**：返回 **pFeatureLevels** 数组中第一个支持的特征等级（如果 **pFeatureLevels** 为 null，则返回可支持的最高等级）。

10．**ppImmediateContext**：返回创建后的设备上下文。

下面是调用该函数的一个示例：

```
UINT createDeviceFlags = 0;

#if  defined(DEBUG)||defined(_DEBUG)
   createDeviceFlags  |= D3D11_CREATE_DEVICE_DEBUG;
```

```
#endif

D3D_FEATURE_LEVEL featureLevel;
ID3D11Device *  md3dDevice;
ID3D11Device Context*  md3dImmediate Context;
HRESULT  hr = D3D11CreateDevice(
    0,                  //  默认显示适配器
    D3D_DRIVER_TYPE_HARDWARE ,
    0,                  //  不使用软件设备
    createDeviceFlags ,
    0, 0,              //  默认的特征等级数组
    D3D11_SDK_VERSION,
    &  md3dDevice ,
    & featureLevel,
    &  md3dImmediateContext);
if(FAILED(hr) )
{
    MessageBox(0, L"D3D11CreateDevice Failed.", 0, 0);
    return  false ;
}
if(featureLevel != D3D_FEATURE_LEVEL_11_0)
{
    MessageBox(0, L"Direct3D FeatureLevel 11 unsupported.", 0, 0);
    return  false;
}
```

从上面的代码可以看到我们使用的是立即执行上下文（immediate context）：

**ID3D11DeviceContext* md3dImmediateContext;**

还有一种上下文叫做延迟执行上下文（**ID3D11Device::CreateDeferredContext**）。该上下文主要用于 Direct3D 11 支持的多线程程序。多线程编程是一个高级话题，本书并不会介绍，但下面介绍一点基本概念：

1．在主线程中使用立即执行上下文。

2．在工作线程总使用延迟执行上下文。

（a）每个工作线程可以将图形指令记录在一个命令列表（**ID3D11CommandList**）中。

（b）随后，每个工作线程中的命令列表可以在主渲染线程中加以执行。

在多核系统中，可并行处理命令列表中的指令，这样可以缩短编译复杂图形所需的时间。

## 4.2.2 检测 4X 多重采样质量支持

创建了设备后，我们就可以检查 4X 多重采样质量等级了。所有支持 Direct3D 11 的设备都支持所有渲染目标格式的 4X MSAA（支持的质量等级可能并不相同）。

```
UINT  m4xMsaaQuality;
HR(md3dDevice ->CheckMultisampleQualityLevels(
    DXGI_FORMAT_R8G8B8A8_UNORM, 4, &  m4xMsaaQuality));
```

```
assert(m4xMsaaQuality > 0);
```

因为 4X MSAA 总是被支持的，所以返回的质量等级总是大于 0。

### 4.2.3 描述交换链

下一步是创建交换链，首先需要填充一个 **DXGI_SWAP_CHAIN_DESC** 结构体来描述我们将要创建的交换链的特性。该结构体的定义如下：

```
typedef struct DXGI_SWAP_CHAIN_DESC {
  DXGI_MODE_DESC BufferDesc;
  DXGI_SAMPLE_DESC SampleDesc;
  DXGI_USAGE BufferUsage;
  UINT BufferCount;
  HWND OutputWindow;
  BOOL Windowed;
  DXGI_SWAP_EFFECT SwapEffect;
  UINT Flags;
} DXGI_SWAP_CHAIN_DESC;
```

**DXGI_MODE_DESC** 类型是另一个结构体，其定义如下：

```
typedef struct DXGI_MODE_DESC
{
  UINT Width;                    // 后台缓冲区宽度
  UINT Height;                   // 后台缓冲区高度
  DXGI_RATIONAL RefreshRate;     // 显示刷新率
  DXGI_FORMAT Format;            // 后台缓冲区像素格式
  DXGI_MODE_SCANLINE_ORDER ScanlineOrdering;// display scanline mode
  DXGI_MODE_SCALING Scaling;     // display scaling mode
} DXGI_MODE_DESC;
```

注意：在下面的数据成员描述中，我们只涵盖了一些常用的标志值和选项，它们对于初学者来说非常重要。对于其他标志值和选项的描述，请参阅 SDK 文档。

1．**BufferDesc**：该结构体描述了我们所要创建的后台缓冲区的属性。我们主要关注的属性有：宽度、高度和像素格式；其他属性的详情请参阅 SDK 文档。

2．**SampleDesc**：多重采样数量和质量级别（参阅 4.1.8 节）。

3．**BufferUsage**：设为 **DXGI_USAGE_RENDER_TARGET_OUTPUT**，因为我们要将场景渲染到后台缓冲区（即，将它用作渲染目标）。

4．**BufferCount**：交换链中的后台缓冲区数量；我们一般只用一个后台缓冲区来实现双缓存。当然，你也可以使用两个后台缓冲区来实现三缓存。

5．**OutputWindow**：我们将要渲染到的窗口的句柄。

6．**Windowed**：当设为 **true** 时，程序以窗口模式运行；当设为 **false** 时，程序以全屏（full-screen）模式运行。

7．**SwapEffect**：设为 **DXGI_SWAP_EFFECT_DISCARD**，让显卡驱动程序选择最高效的显示模式。

8．**Flags**：可选的标志值。如果设为 **DXGI_SWAP_CHAIN_FLAG_ALLOW_MODE_SWITCH**，那么当应用程序切换到全屏模

式时，Direct3D 会自动选择与当前的后台缓冲区设置最匹配的显示模式。如果未指定该标志值，那么当应用程序切换到全屏模式时，Direct3D 会使用当前的桌面显示模式。我们在示例框架中没有使用该标志值，因为对于我们的演示程序来说，在全屏模式下使用当前的桌面显示模式可以得到很好的效果。

下面是在我们的示例框架中填充 **DXGI_SWAP_CHAIN_DESC** 结构体的代码：

```
DXGI_SWAP_CHAIN_DESC sd;
sd.BufferDesc.Width    = mClientWidth;    // 使用窗口客户区宽度
sd.BufferDesc.Height = mClientHeight;
sd.BufferDesc.RefreshRate.Numerator = 60;
sd.BufferDesc.RefreshRate.Denominator = 1;
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
sd.BufferDesc.ScanlineOrdering                                        =
DXGI_MODE_SCANLINE_ORDER_UNSPECIFIED;
sd.BufferDesc.Scaling = DXGI_MODE_SCALING_UNSPECIFIED;
// 是否使用 4X MSAA?
if(mEnable4xMsaa)
{
  sd.SampleDesc.Count = 4;
  // m4xMsaaQuality 是通过 CheckMultisampleQualityLevels()方法获得的
  sd.SampleDesc.Quality = m4xMsaaQuality-1;
}
// NoMSAA
else
{
  sd.SampleDesc.Count = 1;
  sd.SampleDesc.Quality = 0;
}
sd.BufferUsage    = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.BufferCount    = 1;
sd.OutputWindow = mhMainWnd;
sd.Windowed     = true;
sd.SwapEffect    = DXGI_SWAP_EFFECT_DISCARD;
sd.Flags         = 0;
```

**注意**：如果你想在运行时改变多重采样的设置，那么必须销毁然后重新创建交换链。

**注意**：因为大多数显示器不支持超过 24 位以上的颜色，再多的颜色也是浪费，所以我们将后台缓冲区的像素格式设置为 **DXGI_FORMAT_R8G8B8A8_UNORM**（红、绿、蓝、alpha 各 8 位）。额外的 8 位 alpha 并不会输出在显示器上，但在后台缓冲区中可以用于特定的用途。

## 4.2.4 创建交换链

交换链（**IDXGISwapChain**）是通过 **IDXGIFactory** 实例的 **IDXGIFactory::CreateSwapChain** 方法创建的：

```
HRESULT  IDXGIFactory::CreateSwapChain(
    IUnknown *pDevice , // 指向 ID3D11Device 的指针
    DXGI_SWAP_CHAIN_DESC *pDesc, // 指向一个交换链描述的指针
    IDXGISwapChain **ppSwapChain); // 返回创建后的交换链
```

我们可以通过 **CreateDXGIFactory**（需要链接 dxgi.lib）获取指向一个 **IDXGIFactory** 实例的指针。但是使用这种方法获取 **IDXGIFactory** 实例，并调用 **IDXGIFactory::CreateSwapChain** 方法后，会出现如下的错误信息：

```
DXGI  Warning: IDXGIFactory::CreateSwapChain: This  function  is
being  called  with a device  from  a  different IDXGIFactory.
```

要修复这个错误，我们需要使用创建设备的那个 **IDXGIFactory** 实例，要获得这个实例，必须使用下面的 COM 查询（具体解释可参见 **IDXGIFactory** 的文档）：

```
IDXGIDevice *  dxgiDevice  =  0;
HR(md3dDevice ->QueryInterface(__uuidof(IDXGIDevice),
    (void**)&dxgiDevice ));
IDXGIAdapter* dxgiAdapter  =  0;
HR(dxgiDevice ->GetParent(__uuidof(IDXGIAdapter),
    (void**))&dxgiAdapte r ));
// 获得 IDXGIFactory 接口
IDXGIFactory*  dxgiFactory  =  0;
HR(dxgiAdapter->GetParent(__uuid of(IDXGIFactory),
    (void**))&dxgiFactor y));
// 现在，创建交换链
IDXGISwapChain*  mSwapChain;
HR(dxgiFactory->CreateSwapChain(md3dDevice, &sd , &mSw ap Chain);
// 释放 COM 接口
ReleaseCOM (dxgiDevice ;
ReleaseCOM (dxgiAdapter);
ReleaseCOM (dxgiFactory);
```

DXGI（DirectX Graphics Inf rastructure）是独立于 Direct3D 的 API，用于处理与图形关联的东西，例如交换链等。DXGI 与 Direct3D 分离的目的在于其他图形 API（例如 Direct2D）也需要交换链、图形硬件枚举、在窗口和全屏模式之间切换，通过这种设计，多个图形 API 都能使用 DXGI API。

**补充**：你也可以使用 **D3D11CreateDeviceAndSwapChain** 方法同时创建设备、设备上下文和交换链，详情请见 Direct3D 11 教程 1：Direct3D 11 基础。

## 4.2.5 创建渲染目标视图

如 4.1.6 节所述，资源不能被直接绑定到一个管线阶段；我们必须为资源创建资源视图，然后把资源视图绑定到不同的管线阶段。尤其是在把后台缓冲区绑定到管线的输出合并器阶段时（使 Direct3D 可以在后台缓冲区上执行渲染工作），我们必须为后台缓冲区创建一个**渲染目标视图（render target view）**。下面的代码说明了一实现过程：

```
ID3D11RenderTargetView* mRenderTargetView;
ID3D11Texture2D* backBuffer;
```

```
mSwapChain->GetBuffer(0,__uuidof(ID3D11Texture2D),
reinterpret_cast<void**>(&backBuffer));
md3dDevice->CreateRenderTargetView(backBuffer,                    0,
&mRenderTargetView);
ReleaseCOM(backBuffer);
```

1．**IDXGISwapChain::GetBuffer** 方法用于获取一个交换链的后台缓冲区指针。该方法的第一个参数表示所要获取的后台缓冲区的索引值（由于后台缓冲区的数量可以大于 1，所以这里必须指定索引值）。在我们的演示程序中，我们只使用一个后台缓冲区，所以该索引值设为 0。第二个参数是缓冲区的接口类型，它通常是一个 2D 纹理（**ID3D11Texture2D**）。第三个参数返回指向后台缓冲区的指针。

2．我们使用 **ID3D11Device::CreateRenderTargetView** 方法创建渲染目标视图。第一个参数指定了将要作为渲染目标的资源，在上面的例子中，渲染目标是后台缓冲区（即，我们为后台缓冲区创建了一个渲染目标视图）。第二个参数是一个指向 **D3D11_RENDER_TARGET_VIEW_DESC** 结构体的指针，该结构体描述了资源中的元素的数据类型。如果在创建资源时使用的是某种强类型格式（即，非弱类型格式），则该参数可以为空，表示以资源的第一个 mipmap 层次（后台缓冲区也只有一个 mipmap 层次）作为视图格式。第三个参数通过指针返回了创建后的渲染目标视图对象。

3．每调用一次 **IDXGISwapChain::GetBuffer** 方法，后台缓冲区的 COM 引用计数就会向上递增一次，这便是我们在代码片段的结尾处释放它（**ReleaseCOM**）的原因。

## 4.2.6 创建深度/模板缓冲区及其视图

我们现在需要创建深度/模板缓冲区。如 4.1.5 节所述，深度缓冲区只是一个存储深度信息的 2D 纹理（如果使用模板，则模板信息也在该缓冲区中）。要创建纹理，我们必须填充一个 **D3D11_TEXTURE2D_DESC** 结构体来描述所要创建的纹理，然后再调用 **ID3D11Device::CreateTexture2D** 方法。该结构体的定义如下：

```
typedef struct D3D11_TEXTURE2D_DESC {
  UINT Width;
  UINT Height;
  UINT MipLevels;
  UINT ArraySize;
  DXGI_FORMAT Format;
  DXGI_SAMPLE_DESC SampleDesc;
  D3D10_USAGE Usage;
  UINT BindFlags;
  UINT CPUAccessFlags;
  UINT MiscFlags;
} D3D11_TEXTURE2D_DESC;
```

1．**Width**：纹理的宽度，单位为纹理元素（texel）。

2．**Height**：纹理的高度，单位为纹理元素（texel）。

3．**MipLevels**：多级渐近纹理层（mipmap level）的数量。多级渐近纹理将在后面的章节“纹理”中进行讲解。对于深度/模板缓冲区来说，我们的纹理只需要一个多级渐近纹理层。

4．**ArraySize**：在纹理数组中的纹理数量。对于深度/模板缓冲区来说，我们只需要一个纹理。

5．**Format**：一个 **DXGI_FORMAT** 枚举类型成员，它指定了纹理元素的格式。对于深度/模板缓冲区来说，它必须是 4.1.5 节列出的格式之一。

6．**SampleDesc**：多重采样数量和质量级别；请参阅 4.1.7 和 4.1.8 节。

7．**Usage**：表示纹理用途的 **D3D11_USAGE** 枚举类型成员。有 4 个可选值：

- **D3D11_USAGE_DEFAULT**：表示 GPU（graphics processing unit，图形处理器）会对资源执行读写操作。CPU 不能读写这种资源。对于深度/模板缓冲区，我们使用 **D3D11_USAGE_DEFAULT** 标志值，因为 GPU 会执行所有读写深度/模板缓冲区的操作。
- **D3D10_USAGE_IMMUTABLE**：表示在创建资源后，资源中的内容不会改变。这样可以获得一些内部优化，因为 GPU 会以只读方式访问这种资源。除了在创建资源时 CPU 会写入初始化数据外，其他任何时候 CPU 都不会对这种资源执行任何读写操作。
- **D3D10_USAGE_DYNAMIC**：表示应用程序（CPU）会频繁更新资源中的数据内容（例如，每帧更新一次）。GPU 可以从这种资源中读取数据，而 CPU 可以向这种资源中写入数据。
- **D3D10_USAGE_STAGING**：表示应用程序（CPU）会读取该资源的一个副本（即，该资源支持从显存到系统内存的数据复制操作）。

8．**BindFlags**：指定该资源将会绑定到管线的哪个阶段。对于深度/模板缓冲区，该参数应设为 **D3D11_BIND_DEPTH_STENCIL**。其他可用于纹理的绑定标志值还有：

- **D3D11_BIND_RENDER_TARGET**：将纹理作为一个渲染目标绑定到管线上。
- **D3D11_BIND_SHADER_RESOURCE**：将纹理作为一个着色器资源绑定到管线上。

9．**CPUAccessFlags**：指定 CPU 对资源的访问权限。如果 CPU 需要向资源写入数据，则应指定 **D3D11_CPU_ACCESS_WRITE**。具有写访问权限的资源的 Usage 参数应设为 **D3D11_USAGE_DYNAMIC** 或 **D3D11_USAGE_STAGING**。如果 CPU 需要从资源读取数据，则应指定 **D3D11_CPU_ACCESS_READ**。具有读访问权限的资源的 Usage 参数应设为 **D3D11_USAGE_STAGING**。对于深度/模板缓冲区来说，只有 GPU 会执行读写操作；所以，我们将该参数设为 0，因为 CPU 不会在深度/模板缓冲区上执行读写操作。

10．**MiscFlags**：可选的标志值，与深度/模板缓冲区无关，所以设为 0。

**注意**：推荐避免使用 **D3D11_USAGE_DYNAMIC** 和 **D3D11_USAGE_STAGING**，因为有性能损失。要获得最佳性能，我们应创建所有的资源并将它们上传到 GPU 并保留其上，只有 GPU 在读取或写入这些资源。但是，在某些程序中必须有 CPU 的参与，因此这些标志无法避免，但你应该将这些标志的使用减到最小。

在本书中，我们会看到以各种不同选项来创建资源的例子；例如，使用不同的 Usage 标志值、绑定标志值和 CPU 访问权限标志值。但就目前来说，我们只需要关心那些与创建深度/模板缓冲区有关的标志值即可，其他选项可以以后再说。

另外，在使用深度/模板缓冲区之前，我们必须为它创建一个绑定到管线上的深度/模板视图。过程与创建渲染目标视图的过程相似。下面的代码示范了如何创建深度/模板纹理以及与它对应的深度/模板视图：

```
D3D11_TEXTURE2D_DESC depthStencilDesc;
depthStencilDesc.Width              = mClientWidth;
depthStencilDesc.Height            = mClientHeight;
depthStencilDesc.MipLevels          = 1;
depthStencilDesc.ArraySize          = 1;
depthStencilDesc.Format                                    =
```

```
DXGI_FORMAT_D24_UNORM_S8_UINT;
// 使用 4X MSAA?——必须与交换链的 MSAA 的值匹配
if( mEnable4xMsaa)
{
    depthStencilDesc.SampleDesc.Count  = 4;
    depthStencilDesc.SampleDesc.Quality  = m4xMsaaQuality-1;
}
//  不使用 MSAA
else
{
    depthStencilDesc.SampleDesc.Count  =  1;
    depthStencilDesc.SampleDesc.Quality  = 0;
}
depthStencilDesc.Usage              = D3D10_USAGE_DEFAULT;
depthStencilDesc.BindFlags          = D3D10_BIND_DEPTH_STENCIL;
depthStencilDesc.CPUAccessFlags     = 0;
depthStencilDesc.MiscFlags          = 0;
ID3D10Texture2D* mDepthStencilBuffer;
ID3D10DepthStencilView* mDepthStencilView;

HR(md3dDevice->CreateTexture2D(
    &depthStencilDesc, 0, &mDepthStencilBuffer));

HR(md3dDevice->CreateDepthStencilView(
    mDepthStencilBuffer, 0, &mDepthStencilView));
```

**CreateTexture2D** 的第二个参数是一个指向初始化数据的指针，这些初始化数据用来填充纹理。不过，由于个纹理被用作深度/模板缓冲区，所以我们不需要为它填充任何初始化数据。当执行深度缓存和模板操作时，Direct3D 会自动向深度/模板缓冲区写入数据。所以，我们在这里将第二个参数指定为空值。

**CreateDepthStencilView** 的第二个参数是一个指向 **D3D11_DEPTH_STENCIL_VIEW_DESC** 的指针。这个结构体描述了资源中这个元素数据类型（格式）。如果资源是一个有类型的格式（非 typeless），这个参数可以为空值，表示创建一个资源的第一个 mipmap 等级的视图（深度/模板缓冲也只能使用一个 mipmap 等级）。因为我们指定了深度/模板缓冲的格式，所以将这个参数设置为空值。

## 4.2.7 将视图绑定到输出合并器阶段

现在我们已经为后台缓冲区和深度缓冲区创建了视图，就可以将些视图绑定到管线的输出**合并器阶段（output merger stage）**，使些资源成为管线的渲染目标和深度/模板缓冲区：

```
md3dImmediateContext->OMSetRenderTargets(
    1,&mRenderTargetView,mDepthStencilView);
```

第一个参数是我们将要绑定的渲染目标的数量；我们在这里仅绑定了一个渲染目标，不过该参数可以为着色器同时绑定多个渲染目标（是一项高级技术）。第二个参数是我们将要

绑定的渲染目标视图数组中的第一个元素的指针。第三个参数是将要绑定到管线的深度/模板视图。

**注意**：我们可以设置一组渲染目标视图，但是只能设置一个深度/模板视图。使用多个渲染目标是一项高级技术，会在本书的第三部分加以介绍。

## 4.2.8 设置视口

通常我们会把 3D 场景渲染到整个后台缓冲区上。不过，有时我们只希望把 3D 场景渲染到后台缓冲区的一个子矩形区域中，如图 4.7 所示。

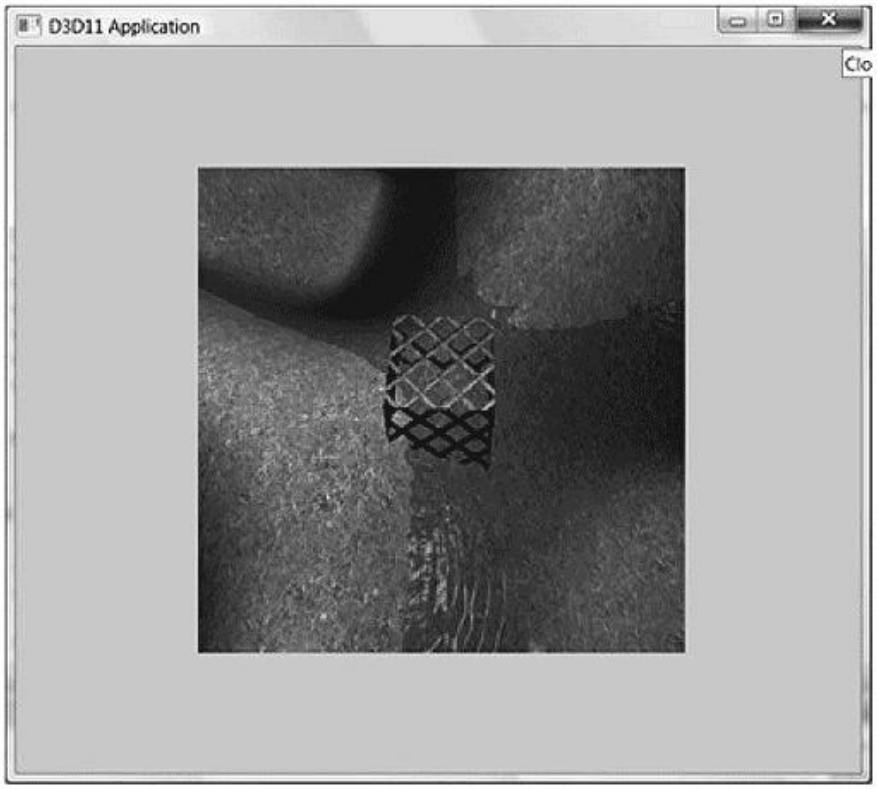



图 4.7：通过修改视口，我们可以把 3D 场景渲染到后台缓冲区的一个子矩形区域中。随后，后台缓冲区中的渲染结果会被呈现到窗口客户区上。

我们将后台缓冲区的子矩形区域称为**视口（viewport）**，它由如下结构体描述：

```
typedef struct D3D11_VIEWPORT {
  FLOAT TopLeftX;
  FLOAT TopLeftY;
  FLOAT Width;
  FLOAT Height;
  FLOAT MinDepth;
  FLOAT MaxDepth;
} D3D11_VIEWPORT;
```

前 4 个数据成员定义了相对于窗口客户区的视口矩形范围。MinDepth 成员表示深度缓冲区的最小值，**MaxDepth** 表示深度缓冲区的最大值。Direct3D 使用的深度缓冲区取值范围

是 0 到 1，除非你想要得到一些特殊效果，否则应将 **MinDepth** 和 **MaxDepth** 分别设为 0 和 1。

在填充了 **D3D11_VIEWPORT** 结构体之后，我们可以使用 **ID3D11Device::RSSetViewports** 方法设置 Direct3D 的视口。下面的例子创建和设置了一个视口，该视口与整个后台缓冲区的大小相同：

```
D3D11_VIEWPORT vp;
vp.TopLeftX = 0;
vp.TopLeftY = 0;
vp.Width     = static_cast<float>(mClientWidth);
vp.Height    = static_cast<float>(mClientHeight);
vp.MinDepth = 0.0f;
vp.MaxDepth = 1.0f;

md3dImmediateContext->RSSetViewports(1, &vp);
```

第一个参数是绑定的视图的数量（可以使用超过 1 的数量用于高级的效果），第二个参数指向一个 viewports 的数组。

例如，你可以使用视口来实现双人游戏模式中的分屏效果。创建两个视口，各占屏幕的一半，一个居左，另一个居右。然后在左视口中以第一个玩家的视角渲染 3D 场景，在右视口中以第二个玩家的视角渲染 3D 场景。你也可以使用视口只绘制到屏幕的一个子矩形中，而在其他区域保留诸如按钮、列表框之类的 UI 控件。

# 4.3 计时和动画

要正确实现动画效果，我们就必须记录时间，尤其是要精确测量动画帧之间的时间间隔。当帧速率高时，帧之间的时间间隔就会很短；所以，我们需要一个高精确度计时器。

## 4.3.1 性能计时器

我们使用性能计时器（或性能计数器）来实现精确的时间测量。为了使用用于查询性能计时器的 Win32 函数，我们必须在代码中添加包含语句“#include<windows.h>”。

性能计时器采用的时间单位称为计数（count）。我们使用 **QueryPerformanceCounter** 函数来获取以计数测量的当前时间值：

```
__int64 currTime;
QueryPerformanceCounter((LARGE_INTEGER*)&currTime);
```

注意，该函数通过它的参数返回当前时间值，该参数是一个 64 位整数。

我们使用 **QueryPerformanceFrequency** 函数来获取性能计时器的频率（每秒的计数次数）：

```
__int64 countsPerSec;
QueryPerformanceFrequency((LARGE_INTEGER*)&countsPerSec);
```

而每次计数的时间长度等于频率的倒数（这个值很小，它只是百分之几秒或者千分之几秒）：

```
mSecondsPerCount = 1.0 / (double)countsPerSec;
```

这样，要把一个时间读数 **valueInCounts** 转换为秒，我们只需要将它乘以转换因子 **mSecondsPerCount**：

```
valueInSecs = valueInCounts * mSecondsPerCount;
```

由 **QueryPerformanceCounter** 函数返回的值本身不是非常有用。我们使用 **QueryPerformanceCounter** 函数的主要目的是为了获取两次调用之间的时间差——在执行一段代码之前记下当前时间，在该段代码结束之后再获取一次当前时间，然后计算两者之间的差值。也就是，我们总是查看两个时间戳之间的相对差，而不是由性能计数器返回的实际值。下面的代码更好地说明了这一概念：

```
__int64 A = 0;
QueryPerformanceCounter((LARGE_INTEGER*)&A);
/* Do work */
__int64 B = 0;
QueryPerformanceCounter((LARGE_INTEGER*)&B);
```

这样我们就可以知道执行这段代码所要花费的计数时间为（B−A），或者以秒表示的时间为（B−A）*mSecondsPerCount。

注意：MSDN 指出当使用 **QueryPerformanceCounter** 函数时，有以下注意事项：“在多处理器计算机中，任何一个处理器单独调用该函数都不会出现问题。但是，由于基础输入/输出系统（BIOS）或硬件抽象层（HAL）存在技术瓶颈，所以你在不同的处理器上调用该函数会得到不同的结果”。你可以使用 **SetThreadAffinityMask** 函数让主应用程序线程只运行在一个处理器上，不在处理器之间进行切换。

### 4.3.2 游戏计时器类

在下面的两节中，我们将讨论 **GameTimer** 类的实现。

```
class GameTimer
{
public:
    GameTimer();

    float TotalTime()const;  // 单位为秒
    float DeltaTime()const; // 单位为秒

    void Reset(); // 消息循环前调用
    void Start(); // 取消暂停时调用
    void Stop();  // 暂停时调用
    void Tick();  // 每帧调用

private:
    double mSecondsPerCount;
    double mDeltaTime;

    __int64 mBaseTime;
    __int64 mPausedTime;
    __int64 mStopTime;
    __int64 mPrevTime;
    __int64 mCurrTime;

    bool mStopped;
};
```

需要特别注意的是，构造函数查询了性能计数器的频率。其他成员函数将在随后的两节中讨论。

```
GameTimer::GameTimer()
: mSecondsPerCount(0.0), mDeltaTime(-1.0), mBaseTime(0),
  mPausedTime(0), mPrevTime(0), mCurrTime(0), mStopped(false)
{
    __int64 countsPerSec;
    QueryPerformanceFrequency((LARGE_INTEGER*)&countsPerSec);
    mSecondsPerCount = 1.0 / (double)countsPerSec;
}
```

**注意：GameTimer** 类的定义和实现部分都保存在了 GameTimer.h 和 GameTimer.cpp 文件中，你可以在示例代码的 Common 目录中找到它们。

### 4.3.3 帧之间的时间间隔

当渲染动画帧时，我们必须知道帧之间的时间间隔，以使我们根据逝去的时间长度来更新游戏中的物体。我们可以采用以下步骤来计算帧之间的时间间隔：设 $t_i$ 为第 $i$ 帧时性能计数器返回的时间值，设 $t_{i-1}$ 为前一帧时性能计数器返回的时间值，那么两帧之间的时间差为 $\Delta t = t_i - t_{i-1}$。对于实时渲染来说，我们至少要达到每秒 30 帧的频率才能得到比较平滑的动画效果（我们一般可以达到更高的频率）；所以，$\Delta t = t_i - t_{i-1}$ 通常是一个非常小的值。

下面的代码示范了 $\Delta t$ 的计算过程：

```
void GameTimer::Tick()
{
    if( mStopped )
    {
        mDeltaTime = 0.0;
        return;
    }

    __int64 currTime;
    QueryPerformanceCounter((LARGE_INTEGER*)&currTime);
    mCurrTime = currTime;

    // 当前帧和上一帧之间的时间差
    mDeltaTime = (mCurrTime - mPrevTime)*mSecondsPerCount;

    // 为计算下一帧做准备
    mPrevTime = mCurrTime;

    // 确保不为负值。DXSDK 中的 CDXUTTimer 提到：如果处理器进入了节电模式
    // 或切换到另一个处理器，mDeltaTime 会变为负值。
    if(mDeltaTime < 0.0)
    {
        mDeltaTime = 0.0;
    }
}

float GameTimer::getDeltaTime() const
{
    return (float)mDeltaTime;
}
```

函数 **Tick** 在应用程序消息循环中的调用如下：

```
int D3DApp::Run()
{
    MSG msg = {0};

```

```
    mTimer.Reset();

    while(msg.message != WM_QUIT)
    {
        // 如果接收到 Window 消息，则处理这些消息
        if(PeekMessage( &msg, 0, 0, 0, PM_REMOVE ))
        {
            TranslateMessage( &msg );
            DispatchMessage( &msg );
        }
        // 否则，则运行动画/游戏
        else
        {
            mTimer.Tick();

            if( !mAppPaused )
            {
                CalculateFrameStats();
                UpdateScene(mTimer.DeltaTime());
                DrawScene();
            }
            else
            {
                Sleep(100);
            }
        }
    }

    return (int)msg.wParam;
}
```

通过这一方式，每帧都会计算出一个$\Delta t$并将它传送给 **UpdateScene** 方法，根据当前帧与前一帧之间的时间间隔来更新场景。下面是 **Reset** 方法的实现代码：

```
void GameTimer::Reset()
{
    __int64 currTime;
    QueryPerformanceCounter((LARGE_INTEGER*)&currTime);

    mBaseTime = currTime;
    mPrevTime = currTime;
    mStopTime = 0;
    mStopped  = false;
}
```

这里包含一些还未讨论过的变量（请参见 4.3.3 节）。不过，我们可以看到，当调用 **Reset** 方法时，**mPrevTime** 被初始化为当前时间。这一点非常重要，因为对于动画的第一帧来说，

没有前面的那一帧，也就是说没有前面的时间戳。所以个值必须在消息循环开始之前初始化。

## 4.3.4 游戏时间

另一个需要测量的时间是从应用程序开始运行时起经过的时间总量，其中不包括暂停时间；我们将这一时间称为游戏时间（game time）。下面的情景说明了游戏时间的用途。假设玩家有 300 秒的时间来完成一个关卡。当关卡开始时，我们会获取时间 $t_{start}$，它是从应用程序开始运行时起经过的时间总量。当关卡开始后，我们不断地将 $t_{start}$ 与总时间 $t$ 进行比较。如果 $t - t_{start} > 300$（如图 4.8 所示），就说明玩家在关卡中的用时超过了 300 秒，输掉了这一关。很明显，在一情景中我们不希望计算游戏的暂停时间。

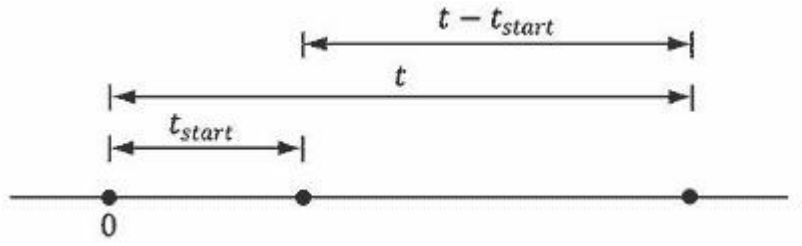



**图 4.8：计算从关卡开始时起的时间。注意，我们将应用程序的开始时间作为原点（0），测量相对于这个时间原点的时间值。**

游戏时间的另一个用途是通过时间函数来驱动动画运行。例如，我们希望一个灯光在时间函数的驱动下环绕着场景中的一个圆形轨道运动。灯光位置可由以下参数方程描述：

$x = 10\cos t$

$y = 20$

$z = 10\sin t$

这里 $t$ 表示时间，随着 $t$（时间）的增加，灯光的位置会发生改变，使灯光在平面 y = 20 上围绕着半径为 10 的圆形轨道运动。对于这种类型的动画，我们也不希望计算游戏的暂停时间；参见图 4.9。

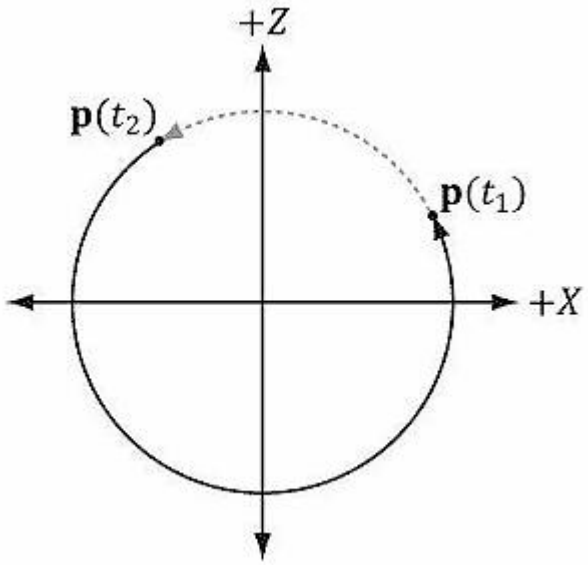



**图 4.9　如果我们在 $t_1$ 时暂停，在 $t_2$ 时取消暂停，并计算暂停时间，那么当我们取消暂停时，灯光的位置会从 $\mathbf{p}(t_1)$ 突然跳到 $\mathbf{p}(t_2)$。**

我们使用以下变量来实现游戏计时：

```
__int64 mBaseTime;
```

```
__int64 mPausedTime;
__int64 mStopTime;
```

如 4.3.3 节所述，当调用 **Reset** 方法时，**mBaseTime** 会被初始化为当前时间。我们可以把它视为从应用程序开始运行时起经过的时间总量。在多数情况下，你只会在消息循环开始之前调用一次 **Reset**，之后不会再调用个方法，因为 **mBaseTime** 在应用程序的整个运行周期中保持不变。变量 **mPausedTime** 用于累计游戏的暂停时间。我们必须累计这一时间，以使我们从总的运行时间中减去暂停时间。当计时器停止时（或者说，当暂停时），**mStopTime** 会帮我们记录暂停时间。

**GameTimer** 类包含两个重要的方法 **Stop** 和 **Start**，它们分别在应用程序暂停和取消暂停时调用，让 GameTimer 记录暂停时间。代码中的注释解释了这两个方法的实现思路。

```
void GameTimer::Stop()
{
    // 如果正处在暂停状态，则略过下面的操作
    if( !mStopped )
    {
        __int64 currTime;
        QueryPerformanceCounter((LARGE_INTEGER*)&currTime);

        // 记录暂停的时间，并设置表示暂停状态的标志
        mStopTime = currTime;
        mStopped  = true;
    }
}

void GameTimer::Start()
{
    __int64 startTime;
    QueryPerformanceCounter((LARGE_INTEGER*)&startTime);


    // 累加暂停与开始之间流逝的时间
    //
    //                     |<-------d------->|
    // ----*---------------*-----------------*------------> time
    //  mBaseTime       mStopTime        startTime

    // 如果仍处在暂停状态
    if( mStopped )
    {
        // 则累加暂停时间
        mPausedTime += (startTime - mStopTime);
        // 因为我们重新开始计时，因此 mPrevTime 的值就不正确了，
        // 要将它重置为当前时间
        mPrevTime = startTime;
```

```
        // 取消暂停状态
        mStopTime = 0;
        mStopped  = false;
    }
}
```

最后，成员函数 **TotalTime** 返回了自调用 **Reset** 之后经过的时间总量，其中不包括暂停时间。它的代码实现如下：

```
// 返回自调用 Reset()方法之后的总时间，不包含暂停时间
float GameTimer::TotalTime()const
{
    // 如果处在暂停状态，则无需包含自暂停开始之后的时间。
    // 此外,如果我们之前已经有过暂停,则 mStopTime - mBaseTime 会包含暂停时间,我们不想包含这个暂停时间，
    // 因此还要减去暂停时间：
    //
    //                    |<--paused time-->|
    //
----*---------------*-----------------*------------*------------*
------> time
    //  mBaseTime        mStopTime          startTime      mStopTime     mCurrTime

    if( mStopped )
    {
        return                    (float)(((mStopTime                    -mPausedTime)-mBaseTime)*mSecondsPerCount);
    }

    // mCurrTime - mBaseTime 包含暂停时间,而我们不想包含暂停时间，
    // 因此我们从 mCurrTime 需要减去 mPausedTime：
    //
    //  (mCurrTime - mPausedTime) - mBaseTime
    //
    //                    |<--paused time-->|
    //  ----*---------------*-----------------*------------*------>
time
    //  mBaseTime       mStopTime        startTime     mCurrTime

    else
    {
        return
(float)(((mCurrTime-mPausedTime)-mBaseTime)*mSecondsPerCount);
    }
}
```

**注意**：我们的演示框架创建了一个 **GameTimer** 实例用于计算应用程序开始后的总时间和两帧之间的时间；你也可以创建额外的实例作为通用的秒表使用。例如，当点着一个炸弹时，你可以启动一个新的 **GameTimer**，当 **TotalTime** 达到 5 秒时，你可以引发一个事件让炸弹爆炸。

# 4.4 演示程序框架

本书中的演示程序均使用 d3dUtil.h、d3dApp.h、d3dApp.cpp 文件中的代码，这些文件可以从本书网站下载。由于本书的第 II 部分和第III部分的所有演示程序都会用到些常用文件，所以我们把些文件保存在了 Common 目录下，使些文件被所有的工程共享，避免多次复制文件。d3dUtil.h 文件包含了一些有用的工具代码，d3dApp.h 和 d3dApp.cpp 文件包含了 Direct3D 应用程序类的核心代码。我们希望读者在阅读本章之后，仔细研究一下些文件，因为我们不会涵盖些文件中的每一行代码（例如，我们不会讲解如何创建一个 Windows 窗口，因为基本的 Win32 编程是阅读本书的先决条件）。该框架的目标是隐藏窗口的创建代码和 Direct3D 的初始化代码；通过隐藏些代码，我们可以在设计演示程序时减少注意力的分散，把注意力集中在示例程序所要表达的特定细节上。

## 4.4.1 D3DApp

**D3DApp** 是所有 Direct3D 应用程序类的基类，它提供了用于创建主应用程序窗口、运行应用程序消息循环、处理窗口消息和初始化 Direct3D 的函数。另外，这个类还定义了一些框架函数。所有的 Direct3D 应用程序类都继承于 **D3DApp** 类，重载它的 virtual 框架函数，并创建一个 **D3DApp** 派生类的单例对象。**D3DApp** 类的定义如下：

```
#ifndef D3DAPP_H
#define D3DAPP_H

#include "d3dUtil.h"
#include "GameTimer.h"
#include <string>

class D3DApp
{
public:
    D3DApp(HINSTANCE hInstance);
    virtual ~D3DApp();

    HINSTANCE AppInst()const;
    HWND      MainWnd()const;
    float     AspectRatio()const;

    int Run();

    // 框架方法。派生类需要重载这些方法实现所需的功能。

    virtual bool Init();
    virtual void OnResize();
    virtual void UpdateScene(float dt)=0;
```

```
    virtual void DrawScene()=0;
    virtual LRESULT MsgProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM
lParam);

    // 处理鼠标输入事件的便捷重载函数
    virtual void OnMouseDown(WPARAM btnState, int x, int y){ }
    virtual void OnMouseUp(WPARAM btnState, int x, int y)  { }
    virtual void OnMouseMove(WPARAM btnState, int x, int y){ }

protected:
    bool InitMainWindow();
    bool InitDirect3D();

    void CalculateFrameStats();

protected:

    HINSTANCE mhAppInst;     // 应用程序实例句柄
    HWND      mhMainWnd;     // 主窗口句柄
    bool      mAppPaused;    // 程序是否处在暂停状态
    bool      mMinimized;    // 程序是否最小化
    bool      mMaximized;    // 程序是否最大化
    bool      mResizing;     // 程序是否处在改变大小的状态
    UINT      m4xMsaaQuality;// 4X MSAA 质量等级

    // 用于记录"delta-time"和游戏时间(§4.3)
    GameTimer mTimer;

    //  D3D11 设备(§4.2.1)，交换链(§4.2.4)，用于深度/模板缓存的 2D 纹理(§
4.2.6)，
    //  渲染目标(§4.2.5)和深度/模板视图(§4.2.6)，和视口(§4.2.8)。
    ID3D11Device* md3dDevice;
    ID3D11DeviceContext* md3dImmediateContext;
    IDXGISwapChain* mSwapChain;
    ID3D11Texture2D* mDepthStencilBuffer;
    ID3D11RenderTargetView* mRenderTargetView;
    ID3D11DepthStencilView* mDepthStencilView;
    D3D11_VIEWPORT mScreenViewport;

    //  下面的变量是在 D3DApp 构造函数中设置的。但是，你可以在派生类中重写这些
值。

    //  窗口标题。D3DApp 的默认标题是"D3D11 Application"。
    std::wstring mMainWndCaption;
```

```
    //   Hardware device 还是 reference device？D3DApp 默认使用
D3D_DRIVER_TYPE_HARDWARE。
    D3D_DRIVER_TYPE md3dDriverType;
    // 窗口的初始大小。D3DApp 默认为 800x600。注意，当窗口大小在运行阶段改变时，
这些值也会随之改变。
    int mClientWidth;
    int mClientHeight;
    //  设置为 true 则使用 4XMSAA（§4.1.8），默认为 false。
    bool mEnable4xMsaa;
};
#endif // D3DAPP_H
```

在上面的代码中，我们使用注释描述了一些数据成员的含义；这些方法将在随后的几节中讨论。

## 4.4.2 非框架方法

**1．D3DApp**：构造函数，将数据成员简单地初始化为默认值。

**2．~D3DApp**：析构函数，释放 **D3DApp** 获取的 COM 接口。

**3．AppInst**：简单的取值函数，返回应用程序实例句柄的一个副本。

**4．MainWnd**：简单的取值函数，返回主窗口句柄的一个副本。

**5．AspectRatio**：后台缓存区的长宽比，这个比值会在下一章中用到，可以通过下面的代码获得：

```
float D3DApp::AspectRatio() const
{
    return static_cast<float>(mClientWidth)/mClientHeight;
}
```

**6．Run**：该方法封装了应用程序消息循环。它使用 Win32 **PeekMessage** 函数，当没有消息时，它让应用程序处理我们的游戏逻辑。该函数的实现代码请参见 4.3.3 节。

**7．InitMainWindow**：初始化主应用程序窗口；我们假定读者已经具备了基本的 Win32 编程知识，知道如何初始化一个 Windows 窗口。

**8．InitDirect3D**：通过 4.2 节描述的各个步骤初始化 Direct3D。

**9．CalculateFrameStats**：计算每秒的平均帧数和每帧的平均时间（单位为毫秒），这个方法的实现在 4.4.4 节中介绍。

## 4.4.3 框架方法

在本书的每个演示程序中，我们都会重载 **D3DApp** 的 5 个 virtual 函数。这 5 个函数用于实现特定示例中的代码细节。**D3DApp** 类实现的这种结构可以将所有的初始化代码、消息处理代码和其他代码安排得井井有条，使派生类专注于实现演示程序的特定代码。下面是对这些框架方法的描述：

**1．Init**：该方法包含应用程序的初始化代码，比如分配资源、初始化对象和设置灯光。该方法在 **D3DApp** 的实现中包含 **InitMainWindow** 和 **InitDirect3D** 方法的调用语句；所以，

当在派生类中重载该方法时，应首先调用该方法的 D3DApp 版本，就像下面这样：

```
void TestApp::Init()
{
    if(!D3DApp::Init())
        return false;
    /* 剩下的初始化代码从这里开始 */
}
```

为你的后续初始化代码提供一个可用的 **ID3D11Device** 设备对象。（通常在获取 Direct3D 资源时都要传递一个有效的 **ID3D11Device** 设备对象。）

**2．OnResize**：该方法在 **D3DApp::MsgProc** 收到 **WM_SIZE** 消息时调用。当窗口的尺寸改变时，一些与客户区大小相关的 Direct3D 属性也需要改变。尤其是需要重新创建后台缓冲区和深度/模板缓冲区，使它们与窗口客户区的大小一致。后台缓冲区的大小可以通过调用 **IDXGISwapChain::ResizeBuffers** 方法来进行调整。而深度/模板缓冲区必须被销毁，然后根据新的大小重新创建。另外，渲染目标视图和深度/模板视图也必须重新创建。**OnResize** 方法在 **D3DApp** 的实现中包含了调整后台缓冲区和深度/模板缓冲区的代码；详情请直接参见源代码。除缓冲区外，依赖于客户区大小的其他属性（例如，投影矩阵）也必须重新创建。我们把该方法作为框架的一部分是因为当窗口大小改变时，客户代码可能需要执行一些它自己的逻辑。

**3．UpdateScene**：该抽象方法每帧都会调用，用于随着时间更新 3D 应用程序（例如，实现动画和碰撞检测、检查用户输入、计算每秒帧数等等）。

**4．DrawScene**：该抽象方法每帧都会调用，用于将 3D 场景的当前帧绘制到后台缓冲区。当绘制当前帧时，我们调用了 **IDXGISwapChain::Present** 方法将后台缓冲区的内容呈现在屏幕上。

**5．MsgProc**：该方法是主应用程序窗口的消息处理函数。通常，当你只需重载该方法，就可以处理未由 **D3DApp::MsgProc** 处理（或者没按照你所希望的方式处理）的消息。该方法的 **D3DApp** 实现版本会在 4.4.5 节中讲解。如果你重载了这个方法，那么那些你没有处理的消息都会送到 **D3DApp::MsgProc** 中进行处理。

**注意**：除了上述的五个框架方法之外，为了使用起来更方便，我们还提供了三个虚函数，用于处理鼠标点击、释放和移动的事件。

```
virtual void OnMouseDown(WPARAM btnState, int x, int y){ }
virtual void OnMouseUp(WPARAM btnState, int x, int y)  { }
virtual void OnMouseMove(WPARAM btnState, int x, int y){ }
```

你可以重载这些方法处理鼠标事件，而用不着重载 **MsgProc** 方法。这些方法的第一个参数 **WPARAM** 都是相同的，保存了鼠标按键的状态（例如，哪个鼠标按键被按下），第二、三个参数是光标在客户区域的（x，y）坐标。

## 4.4.4 帧的统计数值

通常游戏和绘图应用程序都要测量每秒的渲染帧数（FPS）。要实现这一工作，我们只需计算在某一特定时间段 $t$ 中处理的总帧数（并存储在中变量 $n$ 中）。然后得到时间段 $t$ 中的平均 FPS 为 $fps_{avg}=n/t$。如果我们将 $t$ 设为 1，那么 $fps_{avg}=n/1=n$。在我们的代码中，我们将 $t$ 设为 1，这样可以减少一次除法操作，而且，以 1 秒为限可以得到一个最恰当的平均值——个时间间隔既不长也不短。计算 FPS 的代码由 **D3Dapp::CalculateFrameStats** 方法实

现：

```
void D3DApp::CalculateFrameStats()
{
    // 计算每秒平均帧数的代码，还计算了绘制一帧的平均时间。
    // 这些统计信息会显示在窗口标题栏中。
    static int frameCnt = 0;
    static float timeElapsed = 0.0f;

    frameCnt++;

    // 计算一秒时间内的平均值
    if( (mTimer.TotalTime() - timeElapsed) >= 1.0f )
    {
        float fps = (float)frameCnt; // fps = frameCnt / 1
        float mspf = 1000.0f / fps;

        std::wostringstream outs;
        outs.precision(6);
        outs << mMainWndCaption << L"    "
            << L"FPS: " << fps << L"    "
            << L"Frame Time: " << mspf << L" (ms)";
        SetWindowText(mhMainWnd, outs.str().c_str());

        // 为了计算下一个平均值重置一些值。
        frameCnt = 0;
        timeElapsed += 1.0f;
    }
}
```

为了统计帧数，我们在每帧中都会调用该方法。

除了计算 FPS 外，上面的代码还计算了处理一帧所花费的平均时间，单位为毫秒：

```
float mspf = 1000.0f / fps;
```

**注意**：帧时间与 FPS 是倒数关系，通过乘以 1000ms/ 1s 可以将秒转换为毫秒（1 秒等于 1000 毫秒）。

这条语句的含义是：以毫秒为单位计算渲染一帧所花费的时间；是一个与 FPS 不同的值（虽然个值源于 FPS）。实际上，计算帧时间比计算 FPS 更有用，因为它可以更直观地反映出由于修改场景而产生的渲染时间变化（增加或减少）。另一方面，FPS 无法反映出这一变化。而且，[Dunlop03]在他的文章《FPS versus Frame Time》中指出：由于 FPS 曲线是非线性的，所以使用 FPS 可能会得到误导性的结果。例如，考虑情景一：假设我们的应用程序以 1000FPS 的速率运行，每 1ms（毫秒）渲染一帧。当帧速率下降到 250FPS 时，每 4ms 渲染一帧。现在，再考虑情景二：假设我们的应用程序以的 100FPS 的速率运行，每 10ms 渲染一帧。当帧速率下落到大约 76.9 FPS 时，大约为每 13ms 渲染一帧。在两个情景中，帧时间都是增加了 3 毫秒，增加的渲染时间完全相同。但是 FPS 的读数不够直观。从表面看上，似乎从 1000FPS 下降到 250FPS，要比从 100FPS 下降到 76.9FPS 更严重一些。然而，正如我们之前所说，它们实际表示的渲染时间的增长量是相同的。

### 4.4.5 消息处理函数

我们在消息处理函数中实现的代码与整个应用程序框架相比微不足道。通常，我们不会用到许多 Win32 消息。其实，我们的应用程序的核心代码会在处理器空闲执行（即，当没有窗口消息执行）。不过，有一些重要的消息我们必须处理。因为考虑到篇幅问题，我们不可能在这里列出所有的代码； 我们只能对本例使用的几个消息做以讲解。我们希望读者下载源代码文件，花一些时间熟悉应用程序框架代码，因为它是本书每个示例的基础。

我们处理的第 1 个消息是 **WM_ACTIVATE**。当应用程序获得焦点或失去焦点时，该消息被发送。我们这样来处理它：

```
// 当窗口被激活或非激活时会发送 WM_ACTIVATE 消息。
// 当非激活时我们会暂停游戏，当激活时则重新开启游戏。
case WM_ACTIVATE:
    if( LOWORD(wParam) == WA_INACTIVE )
    {
        mAppPaused = true;
        mTimer.Stop();
    }
    else
    {
        mAppPaused = false;
        mTimer.Start();
    }
    return 0;
```

可以看到，当应用程序失去焦点时，我们将数据成员 **mAppPaused** 设为 **true**，当应用程序获得焦点时，我们将数据成员 **mAppPaused** 设为 **false**。另外，当应用程序暂停时，计时器停止运行，当应用程序再次激活时，计时器恢复运行。如果回顾 4.3.3 节中 **D3DApp::Run** 方法，我们会发现当应用程序暂停时，我们并没有执行应用程序中的更新 3D 场景的代码，而是将空闲的 CPU 周期返回给了操作系统；通过这一方式，应用程序不会在处于非活动状态时独占 CPU 周期。

我们处理的第二个消息是 **WM_SIZE**。该消息在改变窗口大小时发生。我们处理该消息的主要原因是希望后台缓冲区和深度/模板缓冲区的大小与窗口客户区的大小相同（为了不出现图像拉伸）。所以，每次改变窗口大小时，我们希望同改变缓冲区的大小。这一任务由 **D3DApp::OnResize** 方法实现。如前所述，后台缓冲区的大小可以通过调用 **IDXGISwapChain::ResizeBuffers** 方法来进行调整。而深度/模板缓冲区必须被销毁，然后根据新的大小重新创建。另外，渲染目标视图和深度/模板视图也必须重新创建。当用户拖动窗口边框时，我们必须格外小心，因为此时会有接连不断的 WM_SIZE 消息发出，我们不希望连续地调整缓冲区大小。所以，当用户拖动窗口边框时，我们（除了暂停应用程序外）不应该执行任何代码，等到用户的拖动操作结束之后我们再调整缓冲区的大小。我们通过处理 **WM_EXITSIZEMOVE** 消息来完成一工作。该消息在用户释放窗口边框时发送。

```
// 当用户拖动窗口边框时会发送 WM_EXITSIZEMOVE 消息。
case WM_ENTERSIZEMOVE:
    mAppPaused = true;
    mResizing  = true;
```

```
    mTimer.Stop();
    return 0;

// 当用户是否窗口边框时会发送 WM_EXITSIZEMOVE 消息。
// 然后我们会基于新的窗口大小重置所有图形变量
case WM_EXITSIZEMOVE:
    mAppPaused = false;
    mResizing  = false;
    mTimer.Start();
    OnResize();
    return 0;
```

最后处理的 3 个消息的实现过程非常简单，所以我们直接来看代码：

```
// 窗口被销毁时发送 WM_DESTROY 消息
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;

// 如果使用者按下 Alt 和一个与菜单项不匹配的字符时，或者在显示弹出式菜单而
// 使用者按下一个与弹出式菜单里的项目不匹配的字符键时。
case WM_MENUCHAR:
    // 按下 alt-enter 切换全屏时不发出声响
    return MAKELRESULT(0, MNC_CLOSE);

// 防止窗口变得过小。
case WM_GETMINMAXINFO:
    ((MINMAXINFO*)lParam)->ptMinTrackSize.x = 200;
    ((MINMAXINFO*)lParam)->ptMinTrackSize.y = 200;
    return 0;
```

## 4.4.6 全屏模式

我们创建的 **IDXGISwapChain** 接口可以自动捕获 Alt+Enter 组合键消息，将应用程序切换到全屏模式（full-screen mode）。在全屏模式下，再次按下 Alt+Enter 组合键，可以返回到窗口模式。在这两种模式的切换中，应用程序的窗口大小会发生变化，会有一个 **WM_SIZE** 消息发送到应用程序的消息队列中；应用程序可以在此时调整后台缓冲区和深度/模板缓冲区的大小，使缓冲区与新的窗口大小匹配。另外，当切换到全屏模式时，窗口样式也会发生改变（即，窗口边框和标题栏会消失）。读者可以使用 Visual Studio 的 Spy++工具查看一下在按下 Alt+Enter 组合键时由演示程序产生的 Windows 消息。

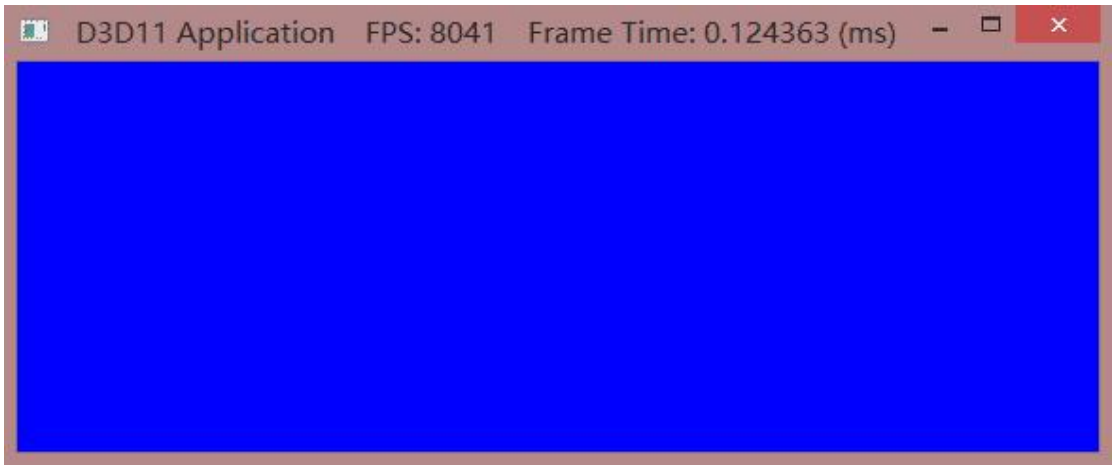



**图 4.10 第 4 章示例程序的屏幕截图。**

注意：读者可以回顾一下 4.2.3 节描述的 **DXGI_SWAP_CHAIN_DESC::Flags** 标志值。

## 4.4.7 初始化 Direct3D 演示程序

现在，我们已经讨论了应用程序框架的所有内容，下面让我们来使用该框架生成一个小程序。基本上，我们用不着做任何实际工作就可以实现这个程序，因为基类 **D3DApp** 已经实现了它所需要的大部分功能。读者在这里应该关注是如何编写 D3DApp 的派生类以及实现框架方法，我们将要在这些框架方法中编写特定的示例代码。本书中的所有程序都使用这一模板。

```
#include "d3dApp.h"

class InitDirect3DApp : public D3DApp
{
public:
    InitDirect3DApp(HINSTANCE hInstance);
    ~InitDirect3DApp();

    bool Init();
    void OnResize();
    void UpdateScene(float dt);
    void DrawScene();
};

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE prevInstance,
                   PSTR cmdLine, int showCmd)
{
    // Enable run-time memory check for debug builds.
#if defined(DEBUG) | defined(_DEBUG)
    _CrtSetDbgFlag( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
#endif
```

```
    InitDirect3DApp theApp(hInstance);

    if( !theApp.Init() )
        return 0;

    return theApp.Run();
}

InitDirect3DApp::InitDirect3DApp(HINSTANCE hInstance)
: D3DApp(hInstance)
{
}

InitDirect3DApp::~InitDirect3DApp()
{
}

bool InitDirect3DApp::Init()
{
    if(!D3DApp::Init())
        return false;

    return true;
}

void InitDirect3DApp::OnResize()
{
    D3DApp::OnResize();
}

void InitDirect3DApp::UpdateScene(float dt)
{

}

void InitDirect3DApp::DrawScene()
{
    assert(md3dImmediateContext);
    assert(mSwapChain);

    md3dImmediateContext->ClearRenderTargetView(mRenderTargetView,
reinterpret_cast<const float*>(&Colors::Blue));
    md3dImmediateContext->ClearDepthStencilView(mDepthStencilView,
D3D11_CLEAR_DEPTH|D3D11_CLEAR_STENCIL, 1.0f, 0);
```

```
    HR(mSwapChain->Present(0, 0));
}
```

# 4.5 调试 Direct3D 应用程序

为了简化代码并突出重点，我们在本书的示例中省略了很多错误处理语句。不过，我们实现了一个宏，用它来检查许多 Direct3D 函数返回的 **HRESULT** 值。这个宏定义在 d3dUtil.h 文件中：

```
#if defined(DEBUG) | defined(_DEBUG)
    #ifndef HR
    #define HR(x)                                              \
    {                                                          \
        HRESULT hr = (x);                                      \
        if(FAILED(hr))                                         \
        {                                                      \
            DXTrace(__FILE__, (DWORD)__LINE__, hr, L#x, true); \
        }                                                      \
    }
    #endif

#else
    #ifndef HR
    #define HR(x) (x)
    #endif
#endif
```

当函数的返回值表明调用失败时，我们把返回值传递给 **DXTrace** 函数。请注意，当使用该函数时，我们必须在代码中添加包含语句“#include<dxerr.h>”，并链接 dxerr.lib 库文件，只有这样程序才能通过编译。

```
HRESULT WINAPI DXTraceW(const char* strFile, DWORD dwLine,
    HRESULT hr, const WCHAR* strMsg, BOOL bPopMsgBox);
```

该函数可以弹出一个消息框，显示出现错误的文件、行号、有关错误的描述信息以及导致错误的函数名。图 4.11 是它的一个例子。注意，当 **DXTrace** 函数的最后一个参数设为 **false** 时，该函数不会显示消息框，而是把调试信息输出到 Visual C++的输出窗口。当我们不使用调试模式时，**HR** 宏不执行任何代码。另外，**HR** 必须是一个宏而不能是一个函数；否则 **__FILE__**和**__LINE__**将无法引用调用 **HR** 宏的函数所在的文件和行。

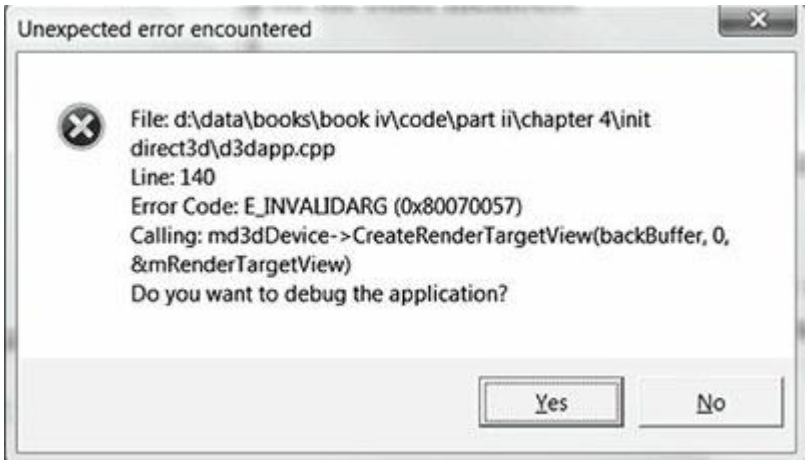



**图 4.11：当 Direct3D 函数返回一个错误时，通过 DXTrace 函数显示消息框。**

现在我们使用 **HR** 宏来包围返回 **HRESULT** 值的 Direct3D 函数，下面是一个例子：

```
HR(D3DX11CreateShaderResourceViewFromFile(md3dDevice,
    L"grass.dds", 0, 0, &mGrassTexRV, 0 ));
```

当我们调试演示程序时，这个宏可以很好地运作，但是对于一个实际的应用程序来说，我们应该使用更完善的错误处理机制。

注意：L#x 将 **HR** 宏的参数转换成一个 Unicode 字符串。通过这一方式，我们可以把导致错误的函数调用语句输出到消息框上面。

# 4.6 小结

**1.** Direct3D 可以被视为程序员和图形硬件之间的一个中介。例如，程序员调用 Direct3D 函数将资源视图绑定到硬件渲染管线、设定渲染管线的输出并绘制 3D 几何体。

**2.** 在 Direct3D 11 中，一个支持 Direct3D 11 的图形设备必须支持 Direct3D 11 规定的整个功能集合以及少量的额外功能。

**3.** 组件对象模型（COM）技术使 DirectX 独立于任何编程语言，并具有版本向后兼容的特性。Direct3D 程序员不必知道 COM 的实现细节及工作方式；只需要知道如何获取和释放 COM 接口即可。

**4.** 1D 纹理如同一维数据元素数组，2D 纹理如同二维数据元素数组，3D 纹理如同三维数据元素数组。纹理元素的格式由 **DXGI_FORMAT** 枚举类型成员描述。纹理通常用于存储图像数据，但是也可以用于存储其他数据，比如深度信息（例如，深度缓冲区）。GPU 可以在纹理上执行特殊运算，比如过滤器和多重采样。

**5.** 在 Direct3D 中，资源不能被直接绑定到一个管线阶段；我们只能把与资源关联的资源视图绑定到不同的管线阶段。我们可以为一个资源创建多个不同的视图。通过这一方式，一个资源可以被绑定到多个不同的渲染管线阶段。如果在创建资源时使用的是弱类型格式，那么在为该资源创建视图时必须指定明确的类型。

**6. ID3D11Device** 和 **ID3D11DeviceContext** 接口可以被视为物理图形设备硬件的软控制器；也就是，我们可以通过这些接口与硬件进行交互。**ID3D11Device** 接口负责检查硬件支持的功能、分配资源。**ID3D11DeviceContext** 接口负责设置渲染状态，将资源绑定到图形管线，发送渲染指令。

**7.** 为了避免动画出现闪烁，最好是将整个帧绘制到一个叫做后台缓冲区的离屏纹理中。当整个屏幕绘制到后台缓冲区之后，它就会以一个完整帧的形式呈现在屏幕上，通过这种方式，观察者就不会觉察到图像的绘制过程了。当帧绘制到后台缓冲区之后，后台缓冲和前台缓冲就会发生互换：后台变前台，前台变后台。交换两者的过程叫做呈现（presenting）。前台缓冲和后台缓冲构成一个交换链，由 IDXGISwapChain 接口表示，使用两个缓冲被称为双缓冲。

8. 对于屏幕上不透明的物体来说，离相机近的点会遮挡后面的点。深度缓冲就是一种判断哪个点离相机近的技术。通过这一方式，我们就无需关心对象的绘制顺序。

9. 性能计数器是一种高精度计时器，它为测量微小的时间差提供了准确无误的计时测量方法，比如帧之间的时间间隔。性能计时器采用的时间单位叫做计数。**QueryPerformanceFrequency** 函数用于输出性能计时器每秒的计数值，这个值的单位可以从计数转换为秒。我们可以通过 **QueryPerformanceCounter** 函数获取性能计时器的当前时间。

10. 我们通过累计某一时间段$\Delta t$ 内的帧数来计算 FPS（frames per second，每秒帧数）。设 $n$ 为时间段$\Delta t$ 中的帧数；在一段时间中的平均每秒帧数为 $fps_{avg}=n/\Delta t$。帧速率会对性能评定产生误导；帧时间是更有效的信息。以秒为单位的帧时间等于帧速率的倒数，即 $1/fps_{avg}$。

11. 示例框架用于为本书的所有演示程序提供统一的编程接口。这些代码保存在 d3dUtil.h、d3dApp.h 和 d3dApp.cpp 文件中，它们封装了每个应用程序必须实现的标准初始化代码。通过封装些代码，可以使示例程序更专注于所要演示的技术。

12. 当以调试模式生成程序时，我们可以使用 D3D11_CREATE_DEVICE_DEBUG 标志值创建 Direct3D 设备来启用调试层。在指定了调试标志值后，Direct3D 会把调试信息发送到 VC++的输出窗口。另外，当以调试模式生成程序时，我们应使用 D3DX 库的调试版本（即，

d3dx11d.lib）。

# 5.1 三维视觉

本章的主要讲解渲染管线（rendering pipeline）。渲染管线是指：在给定一个 3D 场景的几何描述及一架已确定位置和方向的虚拟摄像机（virtual camera）时，根据虚拟摄像机的视角生成 2D 图像的一系列步骤（如图 5.1 所示）。本章的内容大部分是理论性的——下一章才会把理论用于实践。在我们开始学习渲染管线之前，读者应该先了解两个基本概念：一是构成 3D 视觉的基本要素（也就是，研究如何通过 2D 屏幕来呈现 3D 场景）；二是讲解如何在 Direct3D 中以数学方式表示和使用颜色。

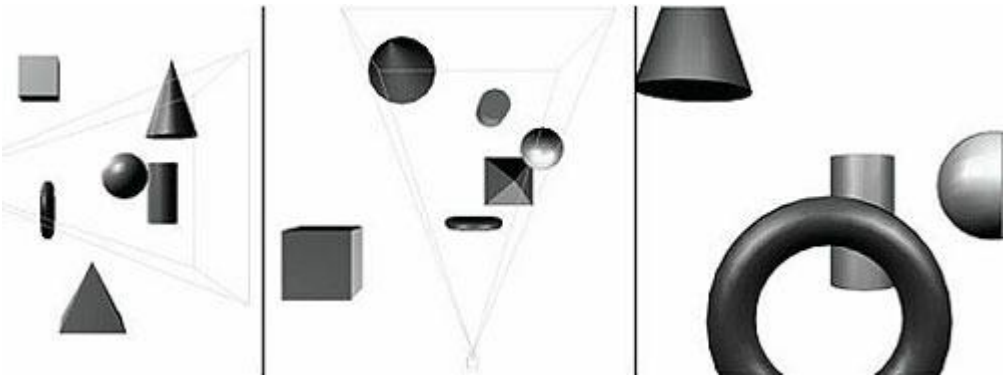


**图 5.1：左图是放置在 3D 场景中的一些物体以及一架已确定位置和方向的摄像机；中图表示的是同一个场景，但是是从上方向下看的视角。“棱锥体”表示观察者所能看到的视域空间；视域空间之外的物体（或者物体的一部分）不会被看到。右图是根据摄像机的“视角”生成的 2D 图像。**

## 学习目标

1. 了解如何在 2D 图像中表现物体的体积感和纵深感。
2. 了解如何在 Direct3D 中描述 3D 物体。
3. 学习如何模拟虚拟摄像机。
4. 理解渲染管线——以几何方式描述 3D 场景并生成 2D 图像的过程。

在踏上 3D 计算机绘图的旅程之前，我们必须先弄明白一个问题：如何在 2D 屏幕上表现 3D 场景的纵深感和体积感？幸运的是，这个问题已经得到了很好的解决，因为几个世纪以来艺术家们一直是在 2D 画布上绘制 3D 场景。本节我们将讲述几种表现图像立体感的关键技术，虽然它实际上是画在平面上的。

假设有一条笔直的铁路，它一直向远处延伸。两条铁轨之间彼此平行，互不相交。当你站在铁路上向远处望时，你会发现随着距离的增加，两条铁轨之间的间隔会越来越近，直至在一个无限远的地方相交为一点。这是通过观察总结出来的我们人类视觉系统的一个特性：平行线会汇集为一个零点（vanishing point），如图 5.2 所示。

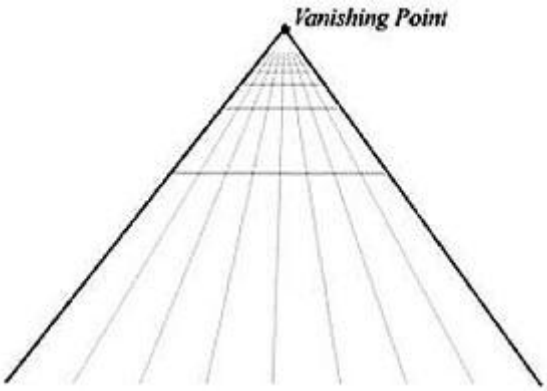



**图 5.2：平行线汇集为一个零点。艺术家们有时将它称为线性透视（linear perspective）。**

从观察中总结出来的人类视觉的另一个特性是物体的大小会随着深度的增加而减小；也就是，近处的物体比远处的物体大。例如，在远处山坡上的一栋房子看上去很小，而近处的一棵树看上去很大。图 5.3 展示了一个简单的场景，其中放置了几排柱子。这些柱子的大小实际上是一样的，但是从观察者的角度上看，随着深度的增加，柱子会变得越来越小。而且我们还可以看到，这些柱子在地平线上会相交为一个零点。

**图 5.3：这里，所有的柱子大小相同，但是由于景深现象（depth phenomenon），观察者会发现柱子越来越小。**

物体重叠（object overlap）是我们能感受到的另一种现象。物体重叠是指一个不透明的物体会挡住它后面的其他物体的一部分（或全部）。这一点非常重要，因为它告诉我们物体在场景中的远近关系。我们已经（在第 4 章中）讨论了如何使用 Direct3D 的深度缓冲区来判定哪些像素会被遮挡，不应该被绘制。出于完整性的考虑，我们在图 5.4 中再次展示了这一情形。

**图 5.4：彼此遮挡的一组物体。**

考虑图 5.5。左边是一个没有光线照射的球体，右边是一个有光线照射的球体。可以看到，左边的球体看上去相当单调——完全不像球体，只能算是一个 2D 圆！因此，在表现 3D 物体的立体感和体积感时光照（lighting）和阴影（shading）具有非常重要的作用。

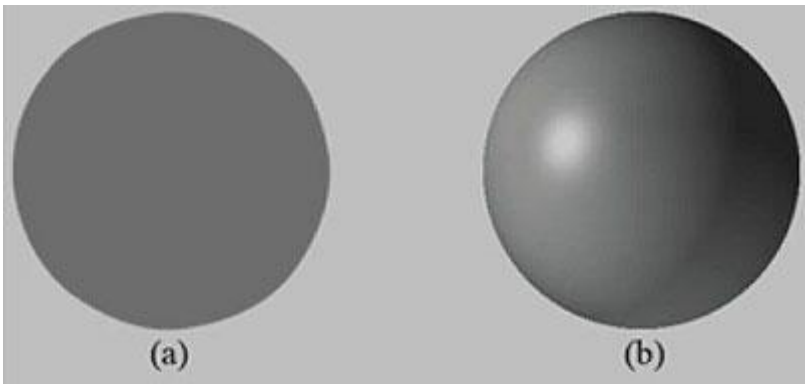



**图 5.5：（a）没有光线照射的球体看上去就像一个 2D 圆。（b）而有光线照射的球体看上去很立体。**

最后，图 5.6 展示了一艘飞船和它的阴影。阴影具有两个关键作用：一是告诉我们场景中的光源位置，二是告诉我们飞船距离地面的高度。

**图 5.6：一艘飞船和它的阴影。阴影间接地说明了光源在场景中的位置以及飞船距离地面的高度。**

刚才讨论的现象都很简单，都是我们在日常生活中能够观察到的现象。但是，更进一步地了解这些现象有助于我们学习和使用 3D 计算机绘图。

# 5.2 模型的表现形式

3D 物体可以通过三角形网格近似地模拟表示，三角形是构成物体模型的基本单位。图 5.7 说明，我们可以通过三角形网格来模拟真实世界中的任何 3D 物体。一般来说，网格的三角形密度越大，模拟出来的效果就越好。当然，我们使用的三角形越多，所要求的硬件性能也就越高，所以必须根据应用程序目标用户的硬件性能来决定模型的精度。除三角形外，有时还需要绘制点和直线。例如，通过绘制一系列 1 像素宽的短线段可以模拟出一条平滑曲线。

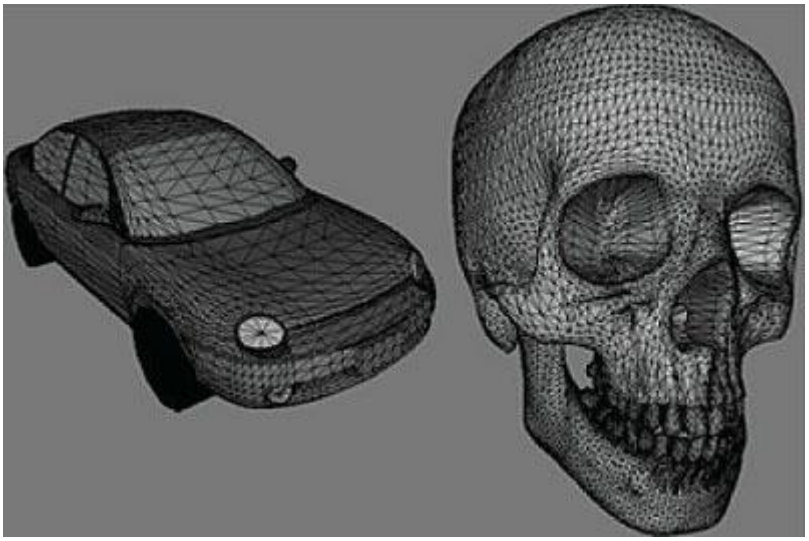


**图 5.7：（左）由低密度三角形网格模拟的轿车。（右）由高密度三角形网格模拟的头骨。**

图 5.7 中的大规则三角形网格说明了一件事情：要以手工方式编写一个 3D 模型的三角形列表是一件极其困难的事情。除了最简单的模型外，所有的模型都是用专门的 3D 建模软件生成的。这些建模软件提供了可视化的交互环境以及非常丰富的建模工具，用户可以使用些软件来创作复杂而逼真的网格模型，整个建模过程非常简单，很容易就能学会。现在在游戏开发领域中较为流行的建模软件有：3ds Max（http://usa.autodesk.com/3ds-max/）、LightWave 3D（http://www.newtek.com/lightwave/）、Maya（http://www.autodesk.com/maya）和 Softimage XSI（www.softimage.com）和 Blender（http://www.blender.org/）。不过，在本书的第一部分中，我们仍会通过手工方式或数学公式生成一些非常简单的 3D 模型（例如，使用参量公式可以很容易地生成圆柱体和球体的三角形列表）。在本书的第三部分中，我们会讲解如何载入和显示由 3D 建模软件导出的 3D 模型。

# 5.3 基本计算机颜色

计算机显示器通过每个像素发射红、绿、蓝光的混合光线。当混合光线进入人的眼睛时会触碰到视网膜的某些区域，对锥感细胞产生刺激，神经触突会通过视觉神经传送到大脑。大脑会解释些信号并生成颜色。随着混合光线的变化，细胞会受到不同的刺激，从而在我们的思想意识中产生不同的颜色。图 5.8 说明了红、绿、蓝三色的混合方式以及不同强度的红色。通过为每个颜色分量指定不同的强度并对其进行混合，可以描述我们所要显示的真实图像中的所有颜色。

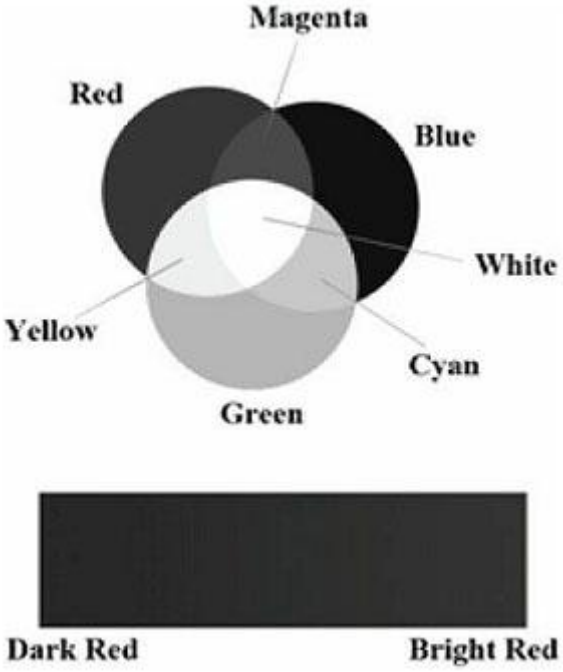



**图 5.8 （上）对纯红、纯绿、纯蓝三种颜色进行混合，得到新的颜色。（下）通过控制红光的强度得到不同明暗程度的红色。**

读者可以使用绘图软件（比如 Adobe Photoshop）或 Win32 颜色对话框（图 5.9）进一步了解如何使用 RGB（red、green、blue）值来描述颜色。尝试使用不同的 RGB 组合，看一看它们产生的颜色。

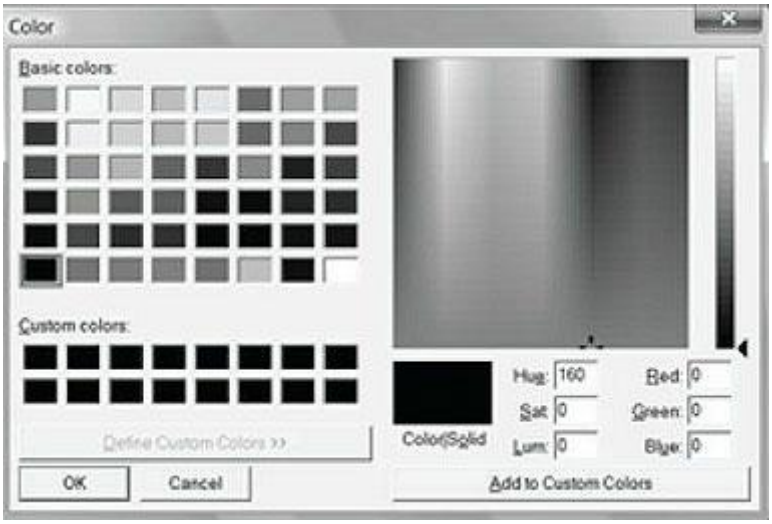


**图 5.9 颜色对话框。**

显示器所能发射的红、绿、蓝光的强度有最大限制。我们使用从 0 到 1 的规范化区间来描述光线强度。0 表示没有强度，1 表示最高强度。中间值表示中等强度。例如，值（0.25,0.67,1.0）表示混合光由强度为 25%的红光、强度为 67%的绿和强度为 100%的蓝光组成。如本例所示，我们可以通过 3D 向量（r,g,b）来表示颜色，其中 0≤r，g，b≤1，三个颜色分量分别描述红、绿、蓝光的强度。

## 5.3.1 颜色运算

某些向量运算也适用于颜色向量。例如，我们可以把颜色向量加在一起得到一个新的颜色：

(0.0, 0.5, 0.0) + (0.0, 0.0, 0.25) = (0.0, 0.5, 0.25)

通过混合一个中等强度的绿色和一个低强度的蓝色，得到一个深绿色。

也可以通过颜色相减来得到一个新的颜色：

(1, 1,1) − (1, 1,0) = (0, 0,1)

也就是，我们从白色中减去它的红色和绿色部分，得到最终的蓝色。

标量乘法也有意义。例如：

0.5(1, 1,1) = (0.5, 0.5, 0.5)

也就是，将白色乘以 0.5，得到一个中等强度的灰色。而运算 2(0.25, 0.0, 0.0) = (0.5, 0.0, 0.0)，可使红色分量的强度增大一倍。

颜色向量的点积和叉积没有意义。不过，颜色向量有一种特殊的乘法运算，叫做分量乘法（componentwise multiplication）。其定义如下：

$(c_r,c_g,c_b)\otimes(k_r,k_g,k_b) = (c_rk_r,c_gk_g,c_bk_b)$

这一运算主要用于光照方程。例如，一个颜色为(r,g,b)的入射光，照射在一个平面上。该平面反射 50%的红光、75%的绿和 25%的蓝光，其余线均被平面吸收。则折回的反射光颜色为：

$(r,g,b)\otimes(0.5,0.75,0.25) = (0.5r,0.75g,0.25b)$

我们可以看到，由于平面吸收了一些线，所以当光线照射在平面上时会丢失一些颜色。

当进行颜色运算时，某些颜色分量可能会超出[0,1]区间；例如，方程(1,0.1,0.6) + (0.0,0.3,0.5) = (1,0.4,1.1)。由于 1.0 表示颜色分量的最大强度，任何分量都不能大于该值。所以，我们要把 1.1 截取为 1.0。同样，显示器不能发射负光，所以任何负的颜色分量（负值是由减法运算取得的结果）都必须截取为 0.0。

## 5.3.2 128 位颜色

通常，在颜色中会包含一个附加的颜色分量，叫做 alpha 分量。alpha 分量用于表示颜色的不透明度，我们会在第 9 章“混合”中使用 alpha 分量。（由于我们目前还用不到混合，所以现在暂且将 alpha 分量设置为 1。）

包含 alpha 分量意味着我们要使用 4D 向量(r,g,b,a)来表示颜色，其中 0≤r,g,b,a≤1。要表示一个 128 位颜色，可以为每个分量指定一个浮点值。因为从数学上来说，颜色就是一个 4D 向量，所以我们可以在代码中使用 **XMVECTOR** 类型表示一个颜色，而且还可以利用 XNA 数学矢量函数所用的 SIMD 操作带来的优势进行颜色运算（例如颜色相加、相减、标量乘法）。对于分量乘法，XNA 数学库提供了以下方法：

```
XMVECTOR XMColorModulate(// Returns (cr, cg, cb, ca) ⊗ (kr,kg,kb,ka)
    FXMVECTOR C1, // (cr, cg, cb, ca)
    FXMVECTOR C2 // (kr, kg, kb, ka) );
```

### 5.3.3 32 位颜色

当使用 32 位表示一个颜色时，每个字节会对应于一个颜色分量。由于每个颜色分量占用一个 8 位字节，所以每个颜色分量可以表示 256 种不同的明暗强度——0 表示没有强度，255 表示最高强度，中间值表示中等强度。从表面上看，为每个颜色分量分配一个字节似乎很小，但是通过计算组合值（256×256×256 = 16,777,216）可以发现，这种方式可以表示上千万种不同的颜色。XNA 数学库提供了以下结构用于存储 32 位颜色：

```
// ARGB Color; 8-8-8-8 bit unsigned normalized integer components
packed into
// a 32 bit integer.  The normalized color is packed into 32 bits using
8 bit
// unsigned, normalized integers for the alpha, red, green, and blue
components.
// The alpha component is stored in the most significant bits and the
blue
// component in the least significant bits (A8R8G8B8):
// [32] aaaaaaaa rrrrrrrr gggggggg bbbbbbbb [0]
typedef struct _XMCOLOR
{
    union
    {
        struct
        {
            UINT b    : 8;  // Blue:    0/255 to 255/255
            UINT g    : 8;  // Green:   0/255 to 255/255
            UINT r    : 8;  // Red:     0/255 to 255/255
            UINT a    : 8;  // Alpha:   0/255 to 255/255
        };
        UINT c;
    };

#ifdef __cplusplus

    _XMCOLOR() {};
    _XMCOLOR(UINT Color) : c(Color) {};
    _XMCOLOR(FLOAT _r, FLOAT _g, FLOAT _b, FLOAT _a);
    _XMCOLOR(CONST FLOAT *pArray);

    operator UINT () { return c; }
```

```
    _XMCOLOR& operator= (CONST _XMCOLOR& Color);
    _XMCOLOR& operator= (CONST UINT Color);

#endif // __cplusplus

} XMCOLOR;
```

通过将整数区间[0,255]映射到实数区间[0,1]，可以将一个 32 位颜色转换为一个 128 位颜色。这一映射工作是通过将每个分量除以 255 来实现。也就是，当 n 为 0 到 255 之间的一个整数时，对应于规范化区间[0,1]的分量值为 $0 \leqslant \frac{n}{255} \leqslant 1$。例如，32 位颜色(80,140,200,255)变为：

$$(80,140,200,255) \rightarrow \left(\frac{80}{255},\frac{140}{255},\frac{200}{255},\frac{255}{255}\right) \approx (0.31,0.55,0.78,1.0)$$

另一方面，通过将每个颜色分量乘以 255 并进行四舍五入，可以将一个 128 位颜色转换为一个 32 位颜色。例如：

(0.3,0.6,0.9,1.0) → (0.3*255,0.6*255,0.9*255,1.0*255) = (77,153,230,255)

当把一个 32 位颜色转换为一个 128 位颜色或者进行反向转换时，通常要执行额外的位运算，因为 8 位颜色分量通常会被封装在一个 32 位整数中（例如，无符号整数），即在 **XMCOLOR** 中。XNA 数学库使用以下函数处理一个 **XMCOLOR** 并以 **XMVECTOR** 的形式返回：

```
XMVECTOR XMLoadColor(CONST XMCOLOR* pSource);
```

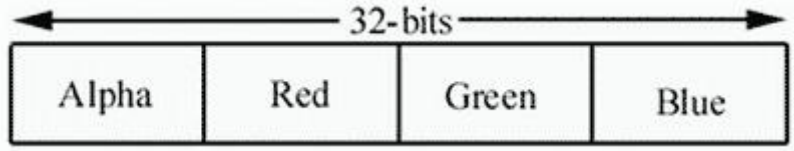



**图 5.10：一个 32 位颜色，它为每个颜色分量分配一个字节。**

图 5.10 说明了如何将 4 个 8 位颜色分量封装为一个无符号整数。注意，这只是用于封装颜色分量的方式之一。除使用 ARGB 外，还可以使用 ABGR 或 RGBA。不过，**XMCOLOR** 类使用 ARGB 格式。XNA 数学库提供了一个函数可以将一 **XMVECTOR** 颜色转化为一个 **XMCOLOR**：

```
VOID XMStoreColor(XMCOLOR*  pDestination,FXMVECTOR V);
```

通常，许多颜色运算（例如，在像素着色器中）使用的都是 128 位颜色值；通过这一方式，我们可以有足够多的二进制位来保证计算的精确度，减少算术错误的累积。不过，最终的像素颜色通常是存储在后台缓冲区的 32 位颜色值中；目前的物理显示设备还不能充分利用更高的分辨率颜色。

# 5.4 渲染管线概述

渲染管线（rendering pipeline）是指：在给定一个 3D 场景的几何描述及一架已确定位置和方向的虚拟摄像机时，根据虚拟摄像机的视角生成 2D 图像的一系列步骤（译者注：渲染管线由许多步骤组成，每个步骤称为一个阶段）。图 5.11 所示为构成渲染管线的各个阶段，以及与各个阶段相关的内存资源。从内存指向阶段的箭头表示该阶段可以从内存读取数据；例如，像素着色器阶段（pixel shader stage）可以从内存中的纹理资源中读取数据。从阶段指向内存的箭头表示该阶段可以向内存写入数据；例如，输出合并器阶段（output merger stage）可以将数据写入后台缓冲区和深度/模板缓冲区。我们还可以看到输出合并器阶段的箭头是双向的（可以读取和写入 GPU 资源）。大多数阶段并不会写入 GPU 资源，它们只是将输出传递到下一阶段；例如，顶点着色器阶段（Vertex Shader Stage）读取输入装配阶段的数据，然后进行处理，接着将结果输出到几何着色器阶段（Geometry Shader Stage）。在随后的几节中，我们将分别讲解渲染管线的各个阶段。

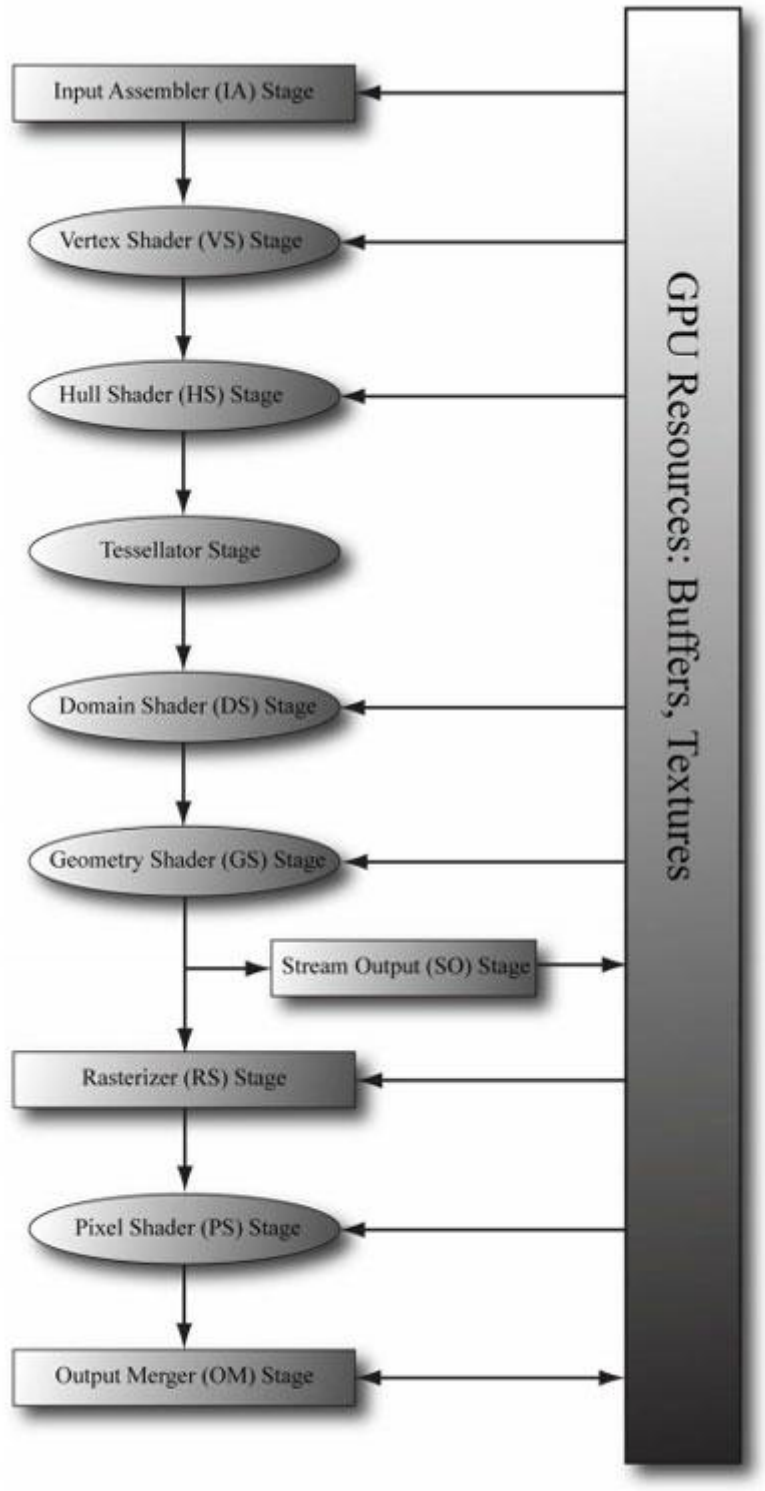



**图 5.11：渲染管线的各个阶段**

# 5.5 输入装配阶段

输入装配（Input Assembler，简称 IA）阶段从内存读取几何数据（顶点和索引）并将这些数据组合为几何图元（例如，三角形、直线）。（索引将在随后的小节中讲解。简单地说，索引规定了顶点的组织形式，解释了该以何种方式组成图元。）

## 5.5.1 顶点

从数学上讲，三角形的顶点位于两条边相交的位置上；而直线的顶点是端点。对于一个单个点来说，点本身就是顶点。图 5.12 说明了顶点的几何图形。

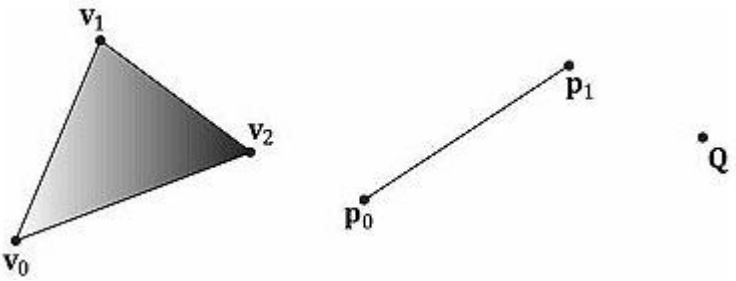



**图 5.12 由三个顶点 $V_1$、$V_2$、$V_3$ 组成的三角形；由两个顶点 $p_0$、$p_1$ 表示的直线；由顶点表示的点。**

图 5.12 说明顶点只是几何图元中的一个特殊的点。不过，在 Direct3D 中，顶点具有更多含义。本质上，Direct3D 中的顶点由空间位置和各种附加属性组成；例如，在第 7 章中，我们会在顶点中添加法线向量实现光照，在第 8 章中，在顶点中添加纹理坐标实现纹理。Direct3D 可以让我们灵活地建立属于我们自己的顶点格式(例如，允许我们定义顶点的分量)。在本书中，我们会基于要绘制的效果定义一些不同的顶点格式。

## 5.5.2 图元拓扑

顶点是以一个叫做顶点缓冲区的 Direct3D 数据结构的形式绑定到图形管线的。顶点缓冲区只是在连续的内存中存储了一个顶点列表。它并没有说明以何种方式组织顶点，形成几何图元。例如，是应该把顶点缓冲区中的每两个顶点解释为一条直线，还是应该把顶点缓冲区中的每三个顶点解释为一个三角形？我们通过指定图元拓扑来告诉 Direct3D 以何种方式组成几何图元：

```
void ID3D11Device::IASetPrimitiveTopology(
    D3D11_PRIMITIVE_TOPOLOGY Topology);

typedef enum D3D11_PRIMITIVE_TOPOLOGY
{
    D3D11_PRIMITIVE_TOPOLOGY_UNDEFINED = 0,
    D3D11_PRIMITIVE_TOPOLOGY_POINTLIST = 1,
    D3D11_PRIMITIVE_TOPOLOGY_LINELIST = 2,
    D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP = 3,
```

```
    D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST = 4,
    D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP = 5,
    D3D11_PRIMITIVE_TOPOLOGY_LINELIST_ADJ = 10,
    D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP_ADJ = 11,
    D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ = 12,
    D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP_ADJ= 13,
    D3D11_PRIMITIVE_TOPOLOGY_1_CONTROL_POINT_PATCHLIST = 33,
    D3D11_PRIMITIVE_TOPOLOGY_2_CONTROL_POINT_PATCHLIST = 34,
    .
    .
    .
    D3D11_PRIMITIVE_TOPOLOGY_32_CONTROL_POINT_PATCHLIST = 64,
} D3D11_PRIMITIVE_TOPOLOGY;
```

所有的绘图操作以当前设置的图元拓扑方式为准。在没有改变拓扑方式之前，当前设置的拓扑方式会一直有效。下面的代码说明了一点：

```
md3dDevice->IASetPrimitiveTopology(
    D3D11_PRIMITIVE_TOPOLOGY_LINELIST);
/* ...draw objects using line list... */
md3dDevice->IASetPrimitiveTopology(
    D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
/* ...draw objects using triangle list... */
md3dDevice->IASetPrimitiveTopology(
    D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP);
/* ...draw objects using triangle strip... */
```

下面的各小节会详细描述各种不同的图元拓扑方式。在本书中，我们主要使用三角形列表。

### 5.5.2.1 点列表

点列表（point list）由 **D3D11_PRIMITIVE_TOPOLOGY_POINTLIST** 标志值表示。当使用点列表时，每个顶点都会被绘制为一个独立的点，如图 5.13a 所示。

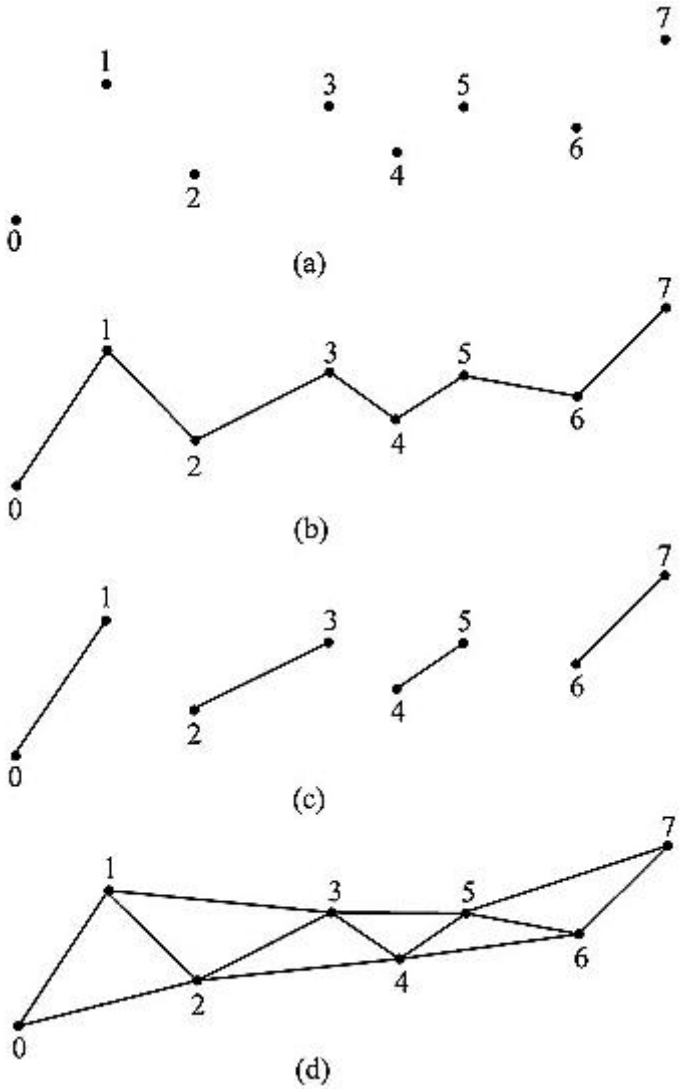



**图 5.13 (a)点列表。(b)线带。(c)线列表。(d)三角形带。**

## 5.5.2.2 线带

线带（line strip）由 **D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP** 标志值表示。当使用线带时，前后相邻的两个顶点会形成一条直线（参见图 5.13b）；这样，n+1 个顶点可以形成 n 条直线。

## 5.5.2.3 线列表

线列表（line list）由 **D3D11_PRIMITIVE_TOPOLOGY_LINELIST** 标志值表示。当使用线列表时，每两个顶点会形成一条独立的直线（参见图 5.13c）；这样，2n 个顶点可以形成 n 条直线。线列表和线带之间的区别是线列表中的直线可以断开，而线带中的直线会自动连在一起；因为内部的每个顶点由两条直线共享，所以线带使用的顶点数量更少。

## 5.5.2.4 三角形带

三角形带（triangle strip）由 **D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP** 标志值表示。当使用三角形带时，顶点会按照图 5.13d 所示的带状方式形成连续的三角形。我们可以看到顶点由相邻的三角形共享，n 个顶点可以形成 n-2 个三角形。

**注意**：偶数三角形和奇数三角形的顶点环绕顺序不同，由此会产生背面消隐问题（参见 5.10.2 节）。为了解决一问题，GPU 会在内部交换偶数三角形的前两个顶点的顺序，以使它们的环绕顺序与奇数三角形相同。

## 5.5.2.5 三角形列表

三角形列表（triangle list）由 **D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST** 标志值表示。当使用三角形列表时，每三个顶点会形成一个独立的三角形（参见图 5.14a）；这样，3n 个顶点可以形成 n 个三角形。三角形列表与三角形带之间的区别是：三角形列表中的三角形可以断开，而三角形带中的三角形会自动连在一起。

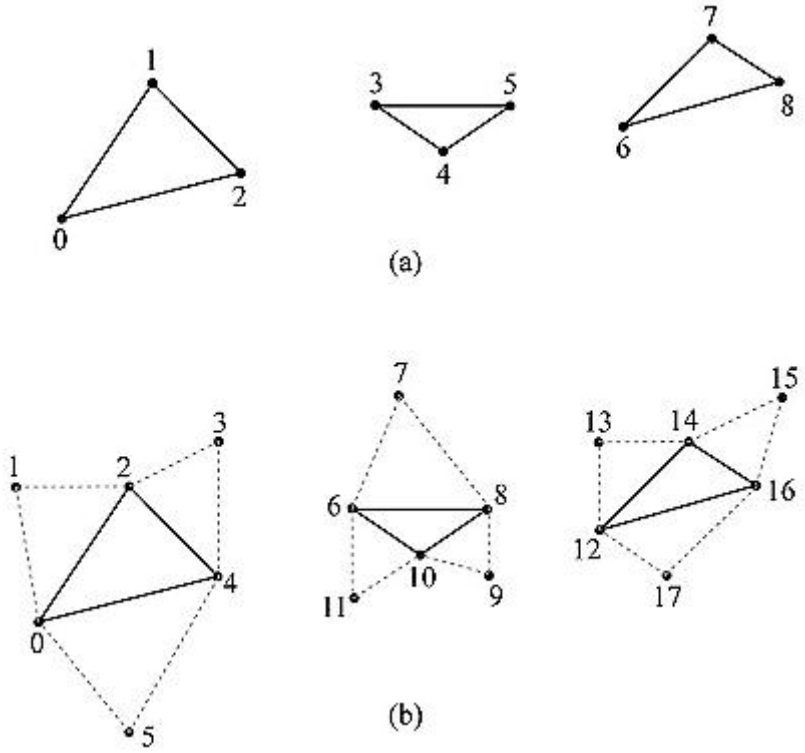



**图 5.14 (a)三角形列表。(b)邻接三角形列表——可以看到每个三角形需要 6 个顶点来描述它和它的邻接三角形。所以，6n 个顶点可以形成 n 个带有邻接信息的三角形。**

## 5.5.2.6 带有邻接信息的图元

在包含邻接信息的三角形列表中，每个三角形都有与之相邻的 3 个邻接三角形；图 5.14b 说明了这些三角形的定义方式。它们主要用于几何着色器，因为某些几何着色算法需要访问邻接三角形。为了实现这些算法，邻接三角形必须与原三角形一起通过顶点/索引缓冲区提

交给管线。通过指定 **D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ** 拓扑标志值可以使管线知道如何从顶点缓冲区中构建三角形以及它的邻接三角形。注意，邻接图元顶点只能作为几何着色器的输入数据——它们不会被绘制出来。如果没有几何着色器，那么邻接图元也不会被绘制出来。

线列表、线带和三角形带也可以包含邻接图元；详情请参见 SDK 文档。

#### 5.5.2.7 控制点面片列表

**D3D11_PRIMITIVE_TOPOLOGY_N_CONTROL_POINT_PATCHLIST** 拓扑标志表示将顶点数据作为 N 控制点的面片列表，这些点用于（可选）图形管线的曲面细分阶段（tessellation stage），我们要到第 13 章才会讨论到它。

### 5.5.3 索引

如前所述，三角形是构成 3D 物体的基本单位。下面的代码示范了使用三角形列表来构建四边形和八边形的顶点数组（即，每三个顶点构成一个三角形）。

```
Vertex quad[6] ={
    v0, v1, v2, // Triangle0
    v0, v2, v3, // Triangle1
};

Vertex octagon[24] ={
    v0, v1, v2, // Triangle0
    v0, v2, v3, // Triangle1
    v0, v3, v4, // Triangle2
    v0, v4, v5, // Triangle3
    v0, v5, v6, // Triangle4
    v0, v6, v7, // Triangle5
    v0, v7, v8, // Triangle6
    v0, v8, v1    // Triangle 7
};
```

**注意**：三角形的顶点顺序非常重要，我们将该顺序称为环绕顺序（winding order）；详情请参见 5.10.2 节。

如图 5.15 所示，构成 3D 物体的三角形会共享许多相同的顶点。更确切地说，在图 5.15a 中，四边形的每个三角形都会共享顶点 $\mathbf{v}_0$ 和 $\mathbf{v}_2$。当复制两个顶点时问题并不明显，但是在八边形的例子中问题就比较明显了（图 5.15b），八边形的每个三角形都会复制中间的顶点 $\mathbf{v}_0$，而且边缘上的每个顶点都由相邻的两个三角形共享。通常，复制顶点的数量会随着模型细节和复杂性的提高而骤然上升。

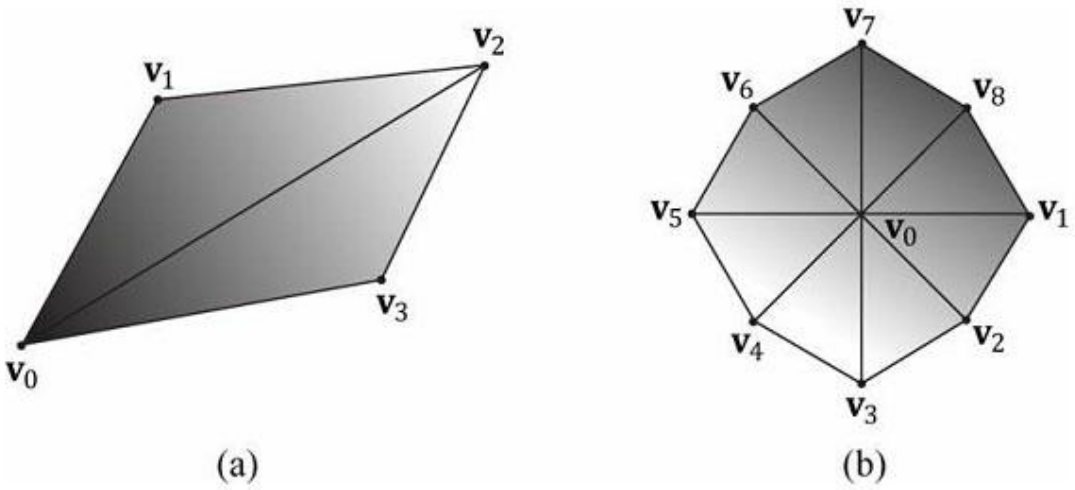



**图 5.15 (a)由 2 个三角形构成的四边形。(b)由 8 个三角形构成的八边形。**

我们不希望对顶点进行复制，主要有两个原因：

**1.** 增加内存需求量。（为什么要多次存储相同的顶点数据？）

**2.** 增加图形硬件的处理负担。（为什么要多次处理相同的顶点数据？）

三角形带在一定程度上可以解决复制顶点问题，但是几何体必须按照带状方式组织，实现起来难度较大。相比之下，三角形列表具有更好的灵活性（三角形不必彼此相连），如果能找到一种方法，即移除复制顶点，又保留三角形列表的灵活性，那么会是一件非常有价值的事情。索引（index）可以解决一问题。它的工作原理是：我们创建一个顶点列表和一个索引列表。顶点列表包含所有唯一的顶点，而索引列表包含指向顶点列表的索引值，这些索引定义了顶点以何种方式组成三角形。回顾图 5.15 中的图形，四边形的顶点列表可以这样创建：

```
Vertex v[4] = {v0, v1, v2, v3};
```

而索引列表需要定义如何将顶点列表中的顶点放在一起，构成两个三角形。

```
UINT indexList[6] = {0, 1, 2,    // Triangle0
    0, 2, 3}; // Triangle 1
```

在索引列表中，每 3 个元素表示一个三角形。所以上面的索引列表的含义为：“使用顶点 **v[0]**、**v[1]**、**v[2]**构成三角形 0，使用顶点 **v[0]**、**v[2]**、**v[3]**构成三角形 1”。

与之类似，八边形的顶点列表可以这样创建：

```
Vertex v[9] = {v0, v1, v2, v3, v4, v5, v6, v7, v8};
```

索引列表为：

```
UINT indexList[24] = {
    0, 1, 2,    // Triangle 0
    0, 2, 3,    // Triangle 1
    0, 3, 4,    // Triangle 2
    0, 4, 5,    // Triangle 3
    0, 5, 6,    // Triangle 4
    0, 6, 7,    // Triangle 5
    0, 7, 8,    // Triangle 6
    0, 8, 1    // Triangle7
};
```

当顶点列表中的唯一顶点得到处理之后，显卡可以使用索引列表把顶点放在一起构成三

角形。我们将“复制问题”转嫁给了索引列表，但是这种复制是可以让人接受的。因为：

1. 索引是简单的整数，不像顶点结构体那样占用很多内存（顶点结构体包含的分量越多，占用的内存就越多）。
2. 通过适当的顶点缓存排序，图形硬件不必重复处理顶点（在绝大多数的情况下）。

# 5.6 顶点着色器阶段

在完成图元装配后，顶点将被送往顶点着色器（vertex shader）阶段。顶点着色器可以被看成是一个以顶点作为输入输出数据的函数。每个将要绘制的顶点都会通过顶点着色器推送至硬件；实际上，我们可以概念性地认为在硬件上执行了如下代码：

```
for(UINT i = 0; i < numVertices; ++i)
    outputVertex[i] = VertexShader(inputVertex[i]);
```

顶点着色器函数由我们自己编写，但是它会在 GPU 上运行，所以执行速度非常快。

许多效果，比如变换（transformation）、光照（lighting）和置换贴图映射（displacement mapping）都是由顶点着色器来实现的。记住，在顶点着色器中，我们不仅可以访问输入的顶点数据，也可以访问在内存中的纹理和其他数据，比如变换矩阵和场景灯光。

我们将会在本书中看到许多不同的顶点着色器示例；当读完本书时，读者会对顶点着色器的功能有一个全面的认识。不过，我们的第一个示例会比较简单，只是用顶点着色器实现顶点变换。在随后的小节中，我们将讲解各种常用的变换算法。

## 5.6.1 局部空间和世界空间

现在让我们来假设一个情景：你正在参与一部影片的拍摄工作，剧组要为拍摄某些特殊效果而搭建一个微缩场景。你的具体任务是搭建一座小桥。现在，你不能在场景中搭建小桥，你必须另选地点，在远离场景的地方建立工作台，制作小桥，以避免弄乱场景中的其他微缩物品。当小桥建成后，你要按照正确的位置和角度把小桥放到场景中。

3D 美术师在创建 3D 场景时也采用同样的工作方式。他们不在全局场景坐标系（world space，世界空间）中建立物体，而是在局部坐标系（local space，局部空间）中建立物体；局部坐标系是最常用的实用坐标系，它的原点接近于物体中心，坐标轴的方向与物体的方向对齐。在完成 3D 模型的制作之后，美术师会将模型放到全局场景中；通过计算局部坐标系相对于世界坐标系的原点和轴向，实现相应的坐标转换变换（参见图 5.16，并回顾 3.4 节的内容）。将坐标从局部坐标系转换到世界坐标系的过程称为世界变换（world transform），相应的变换矩阵称为世界矩阵（world matrix）。当所有的物体都从局部空间变换到世界空间后，这些物体就会位于同一个坐标系（世界空间）中。如果你希望直接在世界空间中定义物体，那么可以使用单位世界矩阵（identity world matrix）。

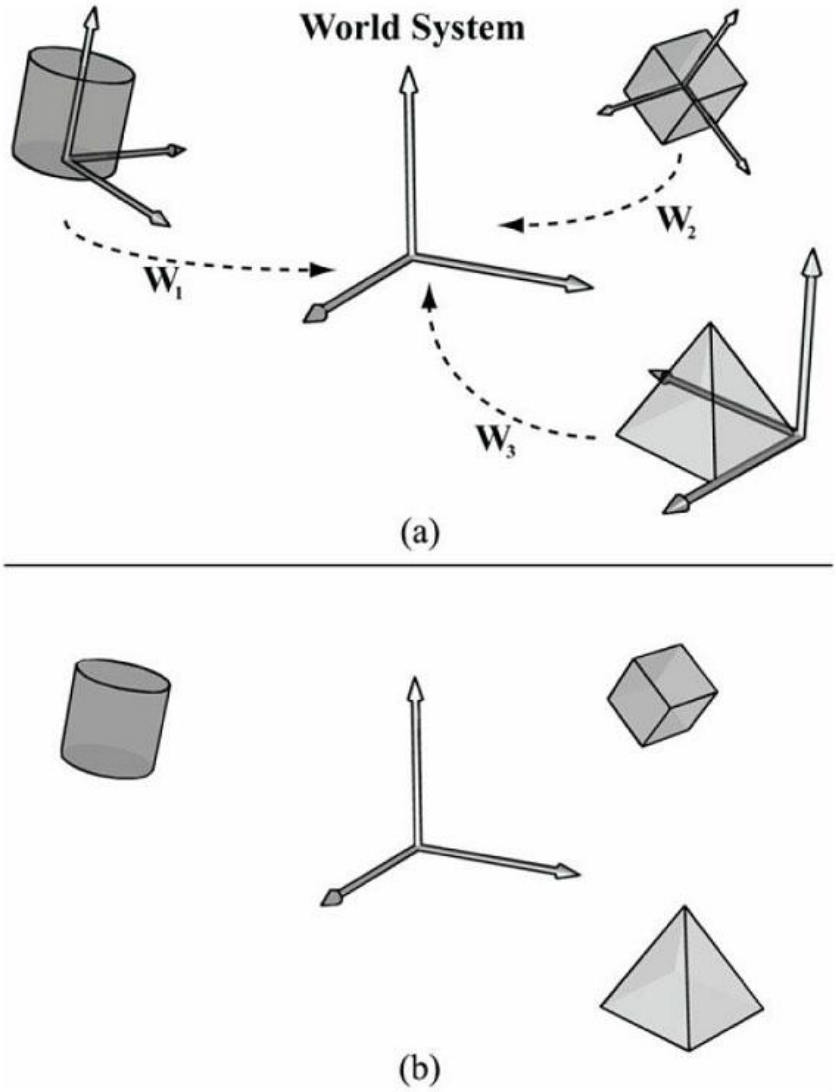



**图 5.16 (a)物体的每个顶点都是相对于它们自己的局部坐标系来定义的。我们根据物体在场景中的位置和方向来定义每个局部坐标系相对于世界坐标系的位置和方向。然后我们执行坐标转换变换，将所有坐标转换到世界坐标系中。(b)在世界变换后，所有物体的顶点都会位于同一个世界坐标系中。**

根据模型自身的局部坐标系定义模型，有以下几点好处：

**1．**简单易用。比如，在局部坐标系中，坐标系的原点通常会与物体的中心对齐，而某个主轴可能正是物体的对称轴。又如，当我们使用局部坐标系时，由于坐标系的原点与立方体的中心对齐，坐标轴垂直于立方体表面，所以可以更容易地描述立方体的顶点（参见图5.17）。

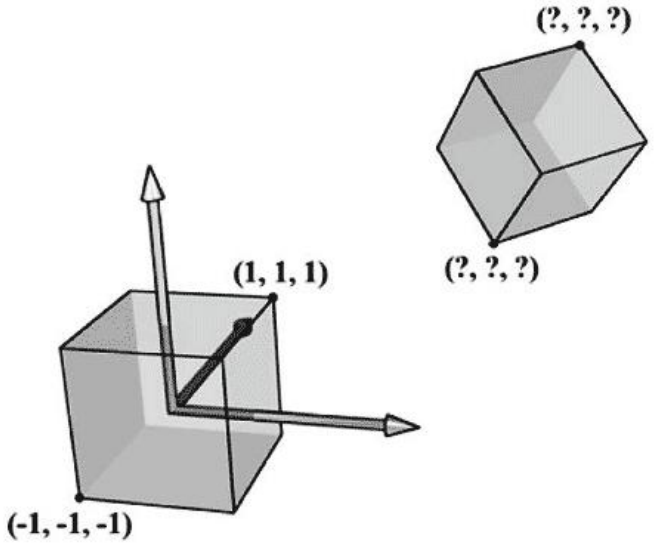



**图 5.17 当立方体的中心位于坐标系原点且与轴对齐（axis-aligned）时，可以很容易地描述立方体的顶点。当立方体位于坐标系的任意一个位置和方向上时，就很难描述这些坐标了。所以，当我们创建模型时，总是选择一种与物体位置接近且与物体方向对齐的实用坐标系。**

**2**．物体可以在多个场景中重复使用，对物体坐标进行相对于特定场景的硬编码是毫无意义的事情。较好的做法是：在局部坐标系中存储物体坐标，通过坐标转换矩阵将物体从局部坐标系变换到世界坐标系，建立物体与场景之间的联系。

**3**．最后，有时我们会多次绘制相同的物体，只是物体的位置、方向和大小有所不同（比如，将一棵树重绘多次形成一片森林）。在这种情况下，我们只需要一个相对于局部坐标系的单个副本，而不是多次复制物体数据，为每个实例创建一个副本。当绘制物体时，我们为每个物体指定不同的世界矩阵，改变它们在世界空间中的位置、方向和大小。这种方法叫做**instancing**。

如 3.4.3 节所述，世界矩阵描述的是一个物体的局部空间相对于世界空间的原点位置和坐标轴方向，这些坐标可以存放在一个行矩阵中。设 $\mathbf{Q}_w = (Q_x, Q_y, Q_z, 1)$、$\mathbf{u}_w = (u_x, u_y, u_z, 0)$、$\mathbf{v}_w = (v_x, v_y, v_z, 0)$、$\mathbf{w}_w = (w_x, w_y, w_z, 0)$分别表示局部空间相对于世界空间的原点、$x$ 轴、$y$ 轴、$z$ 轴的齐次坐标，由 3.4.3 节可知，从局部空间到世界空间的坐标转换矩阵为：

$$\mathbf{W} = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ Q_x & Q_y & u_z & 1 \end{pmatrix}$$

# 示例

假设一个正方形的顶点局部坐标在(−0.5,0,−0.5)和(0.5,0,0.5)之间，将它的边长变为 2，顺时针旋转 45°，并放置在世界空间的（10,0,10）坐标上，那如何求它在世界空间中的坐标呢？我们需要构建 **S**，**R**，**T** 矩阵，世界矩阵 **W** 如下所示：

$$\mathbf{S}=\begin{pmatrix}2&0&0&0\\0&1&0&0\\0&0&2&0\\0&0&0&1\end{pmatrix}\quad \mathbf{R}=\begin{pmatrix}\sqrt{2}/2&0&-\sqrt{2}/2&0\\0&1&0&0\\\sqrt{2}/2&0&\sqrt{2}/2&0\\0&0&0&1\end{pmatrix}\quad \mathbf{T}=\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\10&0&10&1\end{pmatrix}$$

$$\mathbf{W}=\mathbf{SRT}=\begin{pmatrix}\sqrt{2}&0&-\sqrt{2}&0\\0&1&0&0\\\sqrt{2}&0&\sqrt{2}&0\\10&0&10&1\end{pmatrix}$$

根据 3.5 节的讲解，**W** 中的行表示相对于世界空间的局部坐标系；即 $\mathbf{u}_w=(\sqrt{2}\ ,0,-\sqrt{2}\ ,0)$，$\mathbf{v}_w=(0,1,0\ ,0)$，$\mathbf{w}_w=(\sqrt{2}\ ,0,\sqrt{2}\ ,0)$，$\mathbf{Q}_w=(10,0,10,1)$。当我们使用 W 将局部坐标系转换到世界坐标系时，正方形就会处在期望的位置上（见图 5.18）。

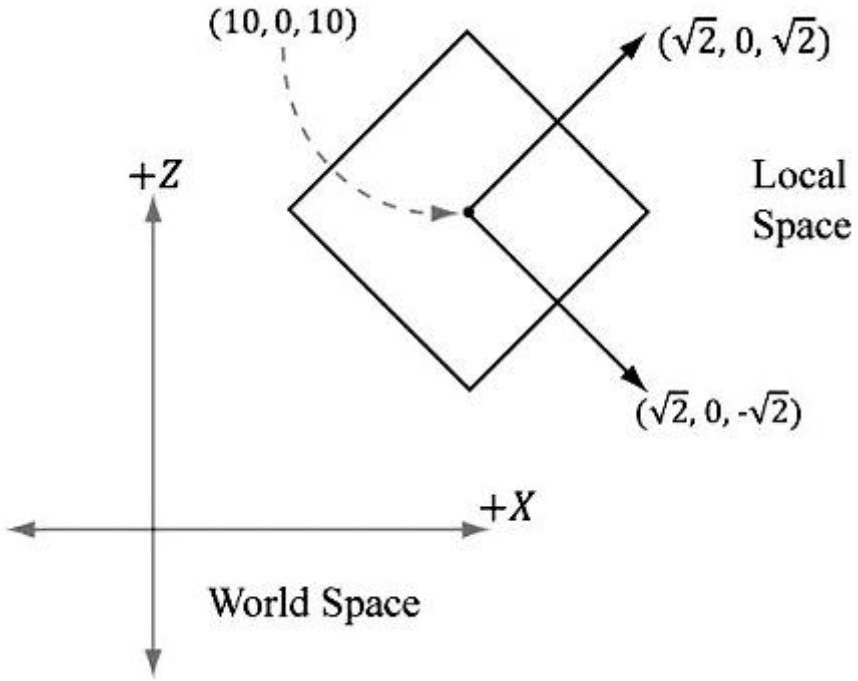



**图 5.18 世界矩阵的行向量表示相对于世界空间的局部坐标系。**

这个例子的要点是无需计算 $\mathbf{Q}_w$，$\mathbf{u}_w$，$\mathbf{v}_w$，$\mathbf{w}_w$ 直接获得世界矩阵，而是通过组合一系列简单的变换矩阵获得世界矩阵，这通常比直接求解 $\mathbf{Q}_w$，$\mathbf{u}_w$，$\mathbf{v}_w$，$\mathbf{w}_w$ 简单。我们只需确定：物体在世界空间中的尺寸多大，在世界空间中的朝向如何，我们要将该物体放置在世界空间中的何处。

还有一种考虑世界变换的方式是：只考虑局部坐标并把它作为世界坐标对待（这相当于使用一个单位矩阵作为世界变换矩阵）。这样，如果物体建模时就位于局部坐标的原点，那么它也在世界空间的坐标原点。通常，世界空间的坐标原点并不是我们想放置物体的位置，所以，对每个物体，我们只要施加一组变换用于缩放、选择、平移，将物体放置在世界空间中的确定位置。从数学上来说，这与将矩阵从局部空间转换到世界空间的变换效果是相同的。

## 5.6.2 观察空间

为了生成场景的 2D 图像，我们必须在场景中放置一架虚拟摄像机。虚拟摄像机指定了

观察者可以看到的场景范围，或者说是我们所要生成的 2D 图像所显示的场景范围。我们把一个局部坐标系（称为观察空间、视觉空间或摄像机空间）附加在摄像机上，如图 5.19 所示；该坐标系以摄像机的位置为原点，以摄像机的观察方向为 $z$ 轴正方向，以摄像机的右侧为 $x$ 轴，以摄像机的上方为 $y$ 轴。在渲染管线的随后阶段中，使用观察空间来描述顶点比使用世界空间来描述顶点要方便的多。从世界空间到观察空间的坐标转换称为观察变换（view transform），相应的矩阵称为观察矩阵（view matrix）。

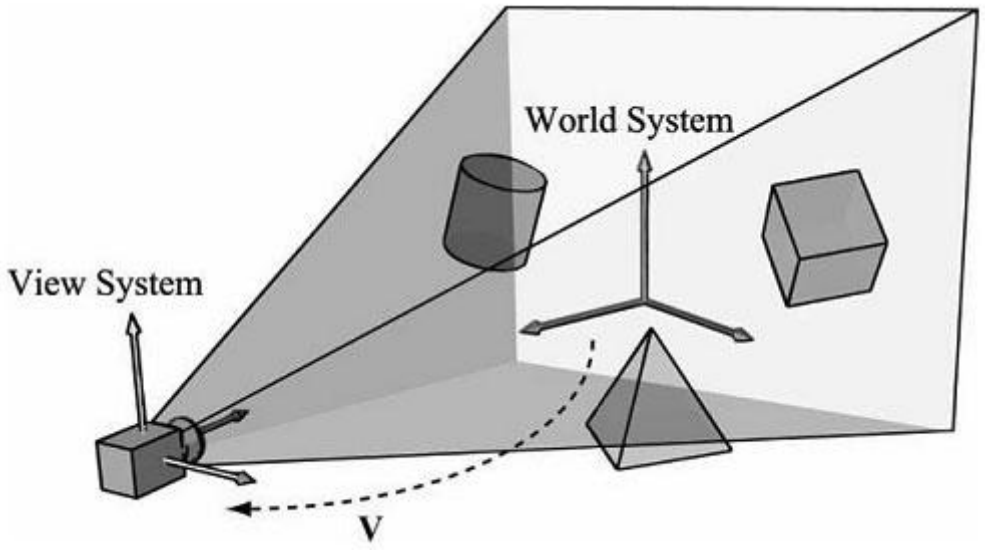



**图 5.19 将相对于世界空间的顶点坐标转换为相对于摄像机空间的顶点坐标。**

设 $\mathbf{Q}_w=(Q_x, Q_y, Q_z, 1)$、$\mathbf{u}_w=(u_x, u_y, u_z, 0)$、$\mathbf{v}_w=(v_x, v_y, v_z, 0)$、$\mathbf{w}_w=(w_x, w_y, w_z, 0)$分别表示观察空间相对于世界空间的原点、$x$ 轴、$y$ 轴、$z$ 轴的齐次坐标，我们由 3.4.3 节可知，从观察空间到世界空间的坐标转换矩阵为：

$$\mathbf{W}=\begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ Q_x & Q_y & u_z & 1 \end{bmatrix}$$

不过，这不是我们想要的结果。我们希望得到的是从世界空间到观察空间的反向变换。回顾 3.4.5 节可知，反向变换可由逆运算取得。也就是，$\mathbf{W}^{-1}$ 为世界空间到观察空间的变换矩阵。

世界坐标系和观察坐标系通常具有不同的位置和方向，所以凭直觉就可以知道 $\mathbf{W}=\mathbf{RT}$ 的含义（即，世界矩阵可以被分解为一个旋转矩阵和一个平移矩阵）。这种方式可以使逆矩阵的计算过程更简单一些：

$$\mathbf{V}=\mathbf{W}^{-1}=(\mathbf{RT})^{-1}=\mathbf{T}^{-1}\mathbf{R}^{-1}=\mathbf{T}^{-1}\mathbf{R}^{\mathrm{T}}$$

$$=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -Q_x & -Q_y & -Q_z & 1 \end{bmatrix}\begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}=\begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ -\mathbf{Q}\cdot\mathbf{u} & -\mathbf{Q}\cdot\mathbf{v} & -\mathbf{Q}\cdot\mathbf{w} & 1 \end{bmatrix}$$

所以，观察矩阵为：

$$\mathbf{V}=\begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ -\mathbf{Q}\cdot\mathbf{u} & -\mathbf{Q}\cdot\mathbf{v} & -\mathbf{Q}\cdot\mathbf{w} & 1 \end{bmatrix}$$

我们现在介绍一种更直观的方法来创建构成观察矩阵的向量。设 **Q** 为摄像机的位置，**T** 为摄像机瞄准的目标点，**j** 为描述世界空间“向上”方向的单位向量。参考图 5.20，摄像机的观察方向为：

$$\mathbf{w}=\frac{\mathbf{T}-\mathbf{Q}}{\|\mathbf{T}-\mathbf{Q}\|}$$

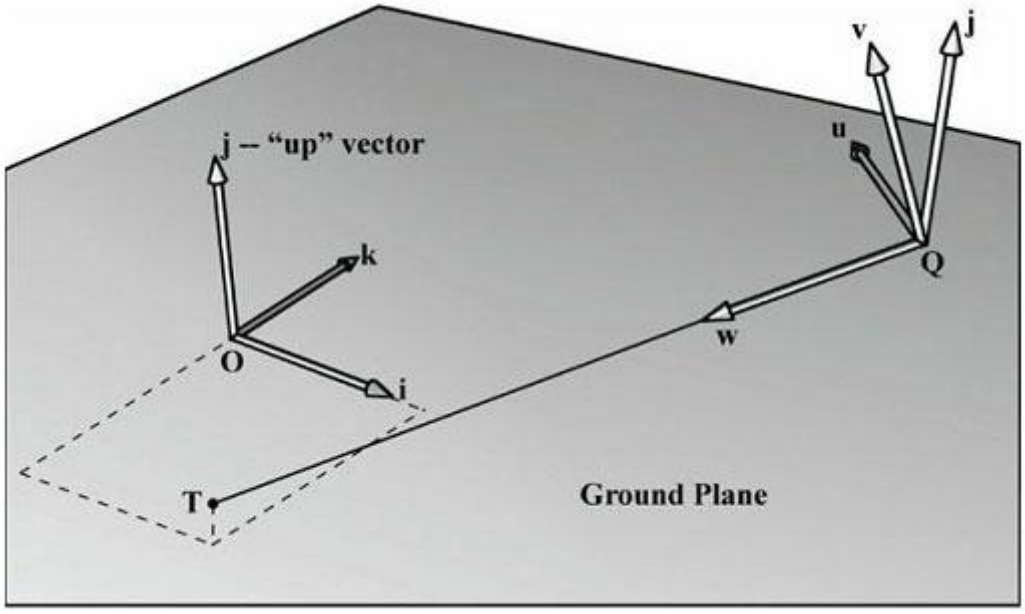



**图 5.20 通过指定摄像机的位置、目标点和世界“向上”向量来创建摄像机坐标系。**

向量 **w** 描述的是摄像机坐标系的 z 轴。指向 **w**“右边”的单位向量为：

$$\mathbf{u}=\frac{\mathbf{j}\times\mathbf{w}}{\|\mathbf{j}\times\mathbf{w}\|}$$

向量 **u** 描述的是摄像机坐标系的 *x* 轴。最后，描述摄像机坐标系 *y* 轴的向量为：

$$\mathbf{v}=\mathbf{w}\times\mathbf{u}$$

由于 **w** 和 **u** 是相互垂直的单位向量，所以 **w**×**u** 必定为单位向量，不需要对它做规范化处理。

这样，给出摄像机的位置、目标点和世界“向上”向量，我们就能够得到摄像机的局部坐标系，该坐标系可以用于创建观察矩阵。

XNA 库提供了如下函数，根据刚才描述的过程计算观察矩阵：

```
XMMATRIX XMMatrixLookAtLH( // Outputs resulting view matrix V
    FXMVECTOR EyePosition,   // Input camera position Q
    FXMVECTOR FocusPosition,   // Input target point T
    FXMVECTOR UpDirection);  // Input world up vector j
```

通常，世界坐标系的 *y* 轴就是“向上”方向，所以“向上”向量 **j** 通常设为(0, 1,0)。举

一个例子，假设摄像机相对于世界空间的位置为(5, 3, −10)，目标点为世界原点(0, 0,0)。我们可以使用如下代码创建观察矩阵：

```
XMVECTOR pos = XMVectorSet(5,3,-10,1.0f);
XMVECTOR target = XMVectorZero();
XMVECTOR up = XMVectorSet(0.0f,1.0f,0.0f,0.0f);
XMMATRIXV = XMMatrixLookAtLH(pos,target,up);
```

## 5.6.3 投影与齐次裁剪空间

到目前为止，我们已经知道了如何在场景中描述摄像机的位置和方向，下面我们来讲解如何描述摄像机所能看到的空间范围。该范围通过一个平截头体（frustum）来描述（图 5.21），它是一个在近平面处削去尖部的棱锥体。

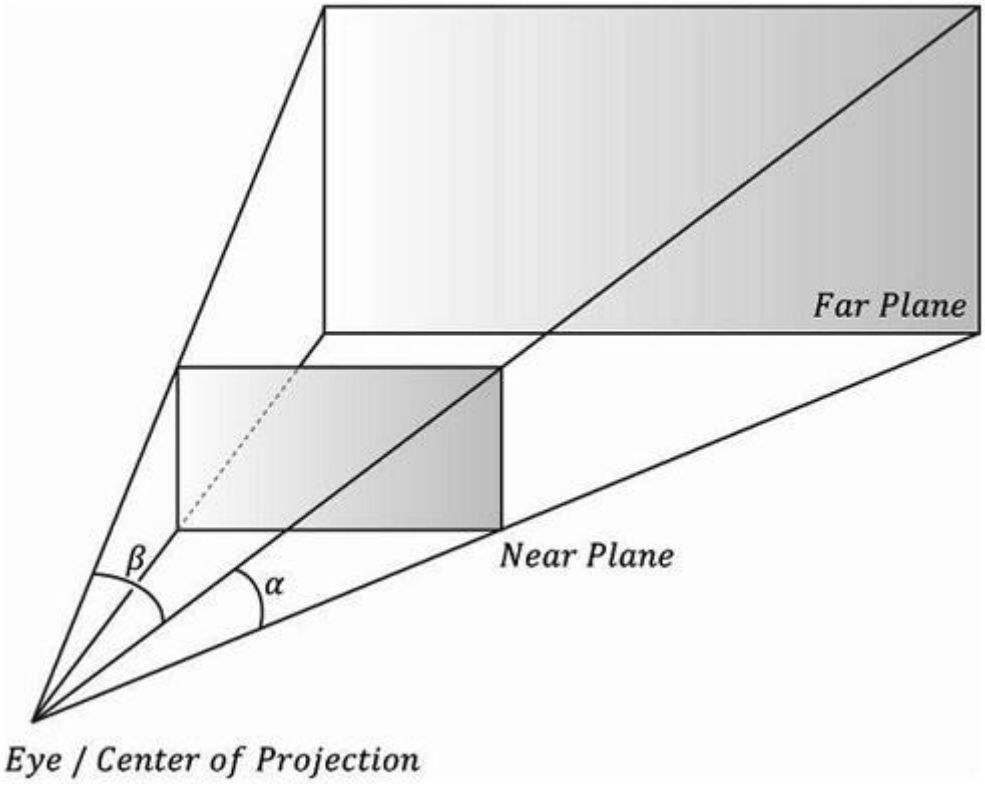



**图 5.21 平截头体描述了摄像机可以“看到”的空间范围。**

我们的下一个任务是把平截头体内的 3D 物体投影到 2D 投影窗口上。投影（projection）必须按照平行线汇集为零点的方式来实现，随着一个物体的 3D 深度增加，它的投影尺寸会越来越小；图 5.22 说明了透视投影的实现过程。我们将“从顶点连向观察点的直线”称为顶点的投影线。然后我们可以定义透视投影变换，将 3D 顶点 **v** 变换到它的投影线与 2D 投影平面相交的点 **v**′上；我们将 **v**′称为 **v** 的投影。对一个 3D 物体的投影就是对组成该物体的所有顶点的投影。

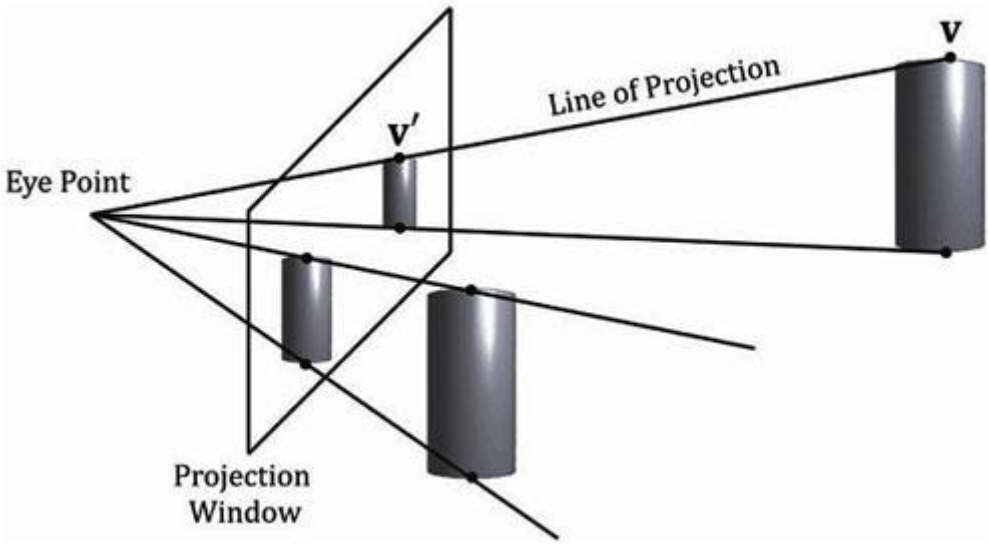



**图 5.22 在 3D 空间中，大小相同、深度不同的两个圆柱体。与观察点距离较近的圆柱体生成的投影较大。在平截头体内的物体可以被映射到投影窗口上；在平截头体外的物体可以映射到投影平面上，但是不会映射到投影窗口上。**

## 5.6.3.1 定义平截头体

我们可以在观察空间中使用如下 4 个参数来定义以原点为投影中心、以 $z$ 轴正方向为观察方向的平截头体：近平面 $n$、远平面 $f$、垂直视域角 $\alpha$ 和横纵比 $r$。注意，在观察空间中，近平面和远平面都平行于 $xy$ 平面；所以，我们只需要简单地指定它们沿 $z$ 轴方向到原点之间的距离即可表示这两个平面。横纵比由 $r = w/h$ 定义，其中 $w$ 表示投影窗口的宽度，$h$ 表示投影窗口的高度（单位由观察空间决定）。投影窗口本质上是指场景在观察空间中的 2D 图像。该图像最终会被映射到后台缓冲区中；所以，我们希望投影窗口的尺寸比例与后台缓冲区的尺寸比例保持相同。在大多数情况下，横纵比就是指后台缓冲区的尺寸比例（它是一个比例值，所以没有单位）。例如，当后台缓冲区的尺寸为 800×600 时，横纵比 $r = 800/600 \approx 1.333$。如果投影窗口的横纵比与后台缓冲区的横纵比不同，那么当投影窗口映射到后台缓冲区时，必然会出现比例失衡，导致图像变形（例如，投影窗口中的一个正圆会被拉伸为后台缓冲区中的一个椭圆）。

将水平视域角设为 $\beta$，它是由垂直视域角 $\alpha$ 和横纵比 $r$ 决定的。考虑图 5.23，分析一下如何通过 $\alpha$、$r$ 来求解 $\beta$。注意，投影窗口的实际尺寸并不重要，重要的只是横纵比。所以，我们将高度设定为 2，则对应的宽度为：

$$r=\frac{w}{h}=\frac{w}{2} \Longrightarrow w=2r$$

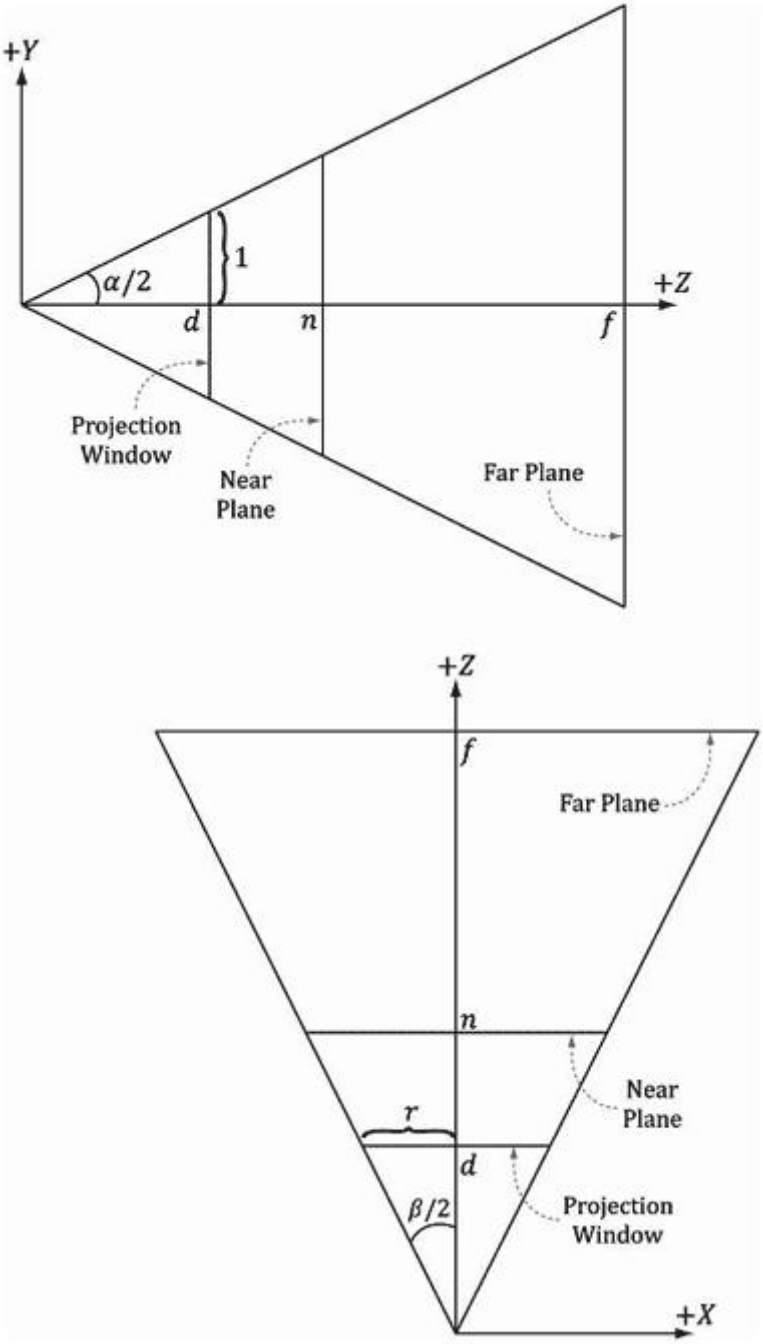



**图 5.23 给出垂直视域角$\alpha$和横纵比 $r$，求解水平视域角$\beta$。**

为了获得指定的垂直视域角$\alpha$，投影窗口必须放在与原点距离为 $d$ 的位置上：

$$\tan\frac{\alpha}{2} = \frac{1}{d} \Longrightarrow d = \cot\frac{\alpha}{2}$$

观察图 5.23 中的 xz 平面，可知：

$$\tan\frac{\beta}{2} = \frac{r}{d} = \frac{r}{\cot\frac{\alpha}{2}} = r \cdot \tan\frac{\alpha}{2}$$

所以，只要给出垂直视域角$\alpha$和横纵比 $r$，我们就能求出水平视域角$\beta$。

$$\beta = 2\tan^{-1}\left(r \cdot \tan\frac{\alpha}{2}\right)$$

## 5.6.3.2 对顶点进行投影

参见图 5.24。给出一个点 $(x, y, z)$，求它在投影平面 $z = d$ 上的投影点 $(x', y', d)$。通过分析 $x$、$y$ 坐标以及使用相似三角形，我们可以求出：

$$\frac{x'}{d} = \frac{x}{z} \Longrightarrow x' = \frac{xd}{z} = \frac{x \cot\frac{\alpha}{2}}{z} = \frac{x}{z\tan\frac{\alpha}{2}}$$

和

$$\frac{y'}{d} = \frac{y}{z} \Longrightarrow y' = \frac{yd}{z} = \frac{y \cot\frac{\alpha}{2}}{z} = \frac{y}{z\tan\frac{\alpha}{2}}$$

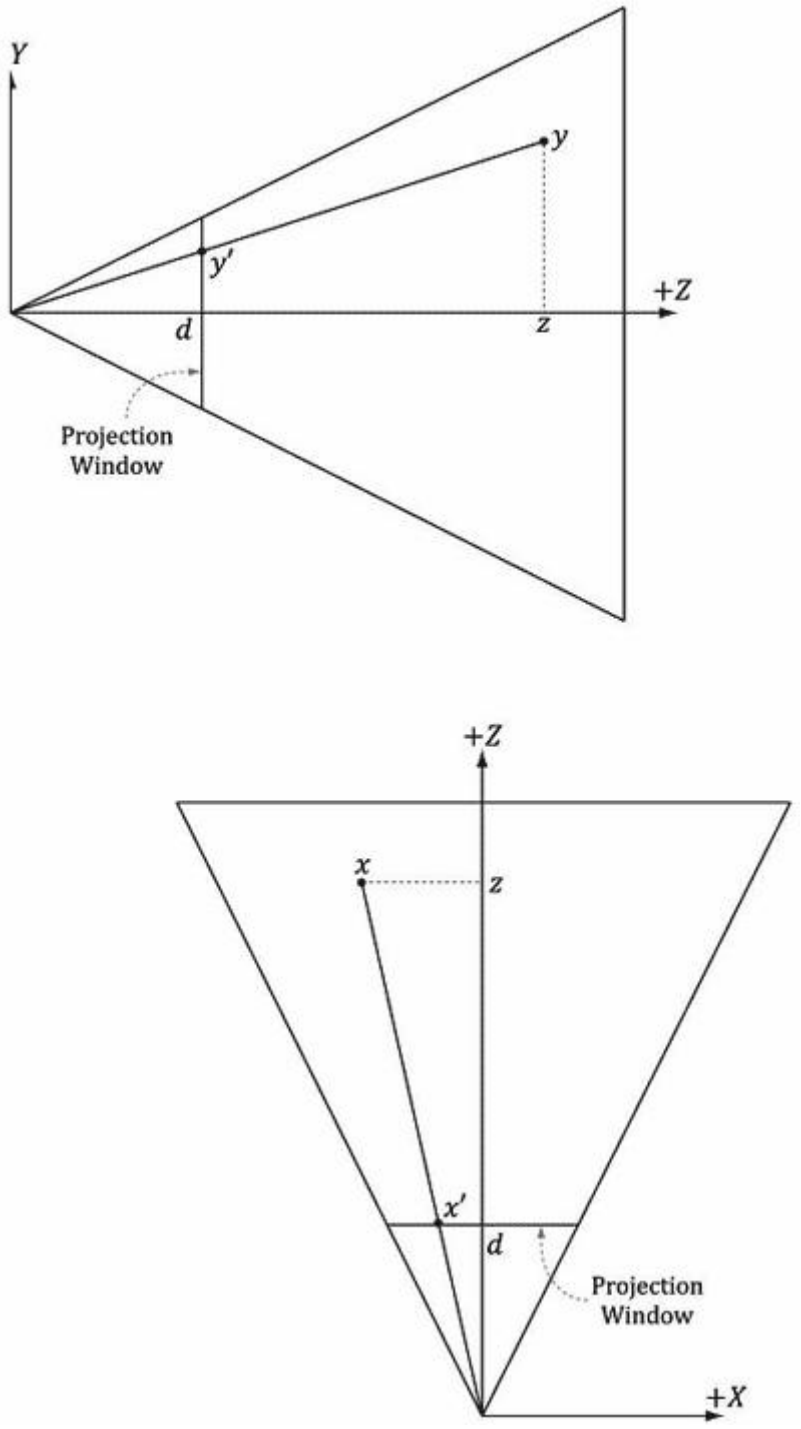



**图 5.24 相似三角形。**

当且仅当以下条件成立时，点(x, y , z)在平截头体内。

$-r \leqslant x' \leqslant r$

$-1 \leqslant y' \leqslant 1$

$n \leqslant z \leqslant f$

## 5.6.3.3 规范化设备坐标（NDC）

上一节我们讲解了如何在观察空间中计算点的投影坐标。在观察空间中，投影窗口的高

度为 2，宽度为 2*r*，其中 *r* 表示横纵比。这里存在的一个问题是：尺寸依赖于横纵比。这意味着我们必须为硬件指定横纵比，否则硬件将无法执行那些与投影窗口尺寸相关的运算（比如，将投影窗口映射到后台缓冲区）。如果我们能去除对横纵比的依赖性，那么会使相关的运算变得更加简单。为了解决一问题，我们将的投影 *x* 坐标从[−*r* , *r*] 区间缩放到[−1,1]区间：

$-r \leqslant x' \leqslant r$

$-1 \leqslant x'/r \leqslant 1$

在映射之后，*x*、*y* 坐标称为规范化设备坐标（normalized device coordinates，简称 NDC）（*z* 坐标还没有被规范化）。当且仅当以下条件成立时，点(*x* ,*y*, *z*)在平截头体内。

$-1 \leqslant x'/r \leqslant 1$

$-1 \leqslant y' \leqslant 1$

$n \leqslant z \leqslant f$

从观察空间到 NDC 空间的变换可以看成是一个单位转换。我们有这样一个关系式：在 *x* 轴上的一个 NDC 单位等于观察空间中的 *r* 个单位（即，1 ndc = *r* vs）。所以给出 *x* 观察空间单位，我们可以使用这个关系式来转换单位：

$$x \text{ vs} \frac{1 \text{ ndc}}{r \text{ vs}} = \frac{x}{r} \text{ ndc}$$

我们可以修改之前的投影公式，直接使用 NDC 空间中的 x、y 投影坐标：

$$x' = \frac{x}{rz\tan\frac{\alpha}{2}}$$

$$y' = \frac{y}{z\tan\frac{\alpha}{2}} \qquad \text{（方程 5.1）}$$

注意：在 NDC 空间中，投影窗口的高度和宽度都为 2。也就是说，现在的尺寸是固定的，硬件不需要知道横纵比，但是我们必须自己来完成投影坐标从观察空间到 NDC 空间的转换工作（图形硬件假定我们会完成一工作） 。

## 5.6.3.4 用矩阵来描述投影方程

为了保持一致，我们将用一个矩阵来描述投影变换。不过，方程 5.1 是非线性的，无法用矩阵描述。所以我们要使用一种“技巧”将它分为两部分来实现：一个线性部分和一个非线性部分。非线性部分要除以 *z*。我们会在下一节讨论“如何规范化 *z* 坐标”时讲解这一问题；现在读者只需要知道，我们会因为个除法操作而失去原始的 z 坐标。所以，我们必须在变换之前保存输入的 *z* 坐标；我们可以利用齐次坐标来解决一问题，将输入的 z 坐标复制给输出的 w 坐标。在矩阵乘法中，我们要将元素[2][3]设为 1、元素[3][3]设为 0（从 0 开始的索引）。我们的投影矩阵大致如下：

$$\mathbf{P}=\begin{bmatrix} \dfrac{1}{r\tan\dfrac{\alpha}{2}} & 0 & 0 & 0 \\ 0 & \dfrac{1}{\tan\dfrac{\alpha}{2}} & 0 & 0 \\ 0 & 0 & A & 1 \\ 0 & 0 & B & 0 \end{bmatrix}$$

注意矩阵中的常量 A 和 B（它们将在下一节讨论）；这些常量用于把输入的 z 坐标变换到规范化区间。将一个任意点(x , y, z,1)与该矩阵相乘，可以得到：

$$[x,y,z,1]\begin{bmatrix} \dfrac{1}{r\tan\dfrac{\alpha}{2}} & 0 & 0 & 0 \\ 0 & \dfrac{1}{\tan\dfrac{\alpha}{2}} & 0 & 0 \\ 0 & 0 & A & 1 \\ 0 & 0 & B & 0 \end{bmatrix}=\begin{bmatrix} \dfrac{x}{r\tan\dfrac{\alpha}{2}}, \dfrac{y}{\tan\dfrac{\alpha}{2}}, Az+B, z \end{bmatrix} \quad \text{（公式 5.2）}$$

在与投影矩阵（线性部分）相乘之后，我们要将每个坐标除以 w = z（非线性部分），得到最终的变换结果：

$$\begin{bmatrix} \dfrac{x}{r\tan\dfrac{\alpha}{2}}, \dfrac{y}{\tan\dfrac{\alpha}{2}}, Az+B, z \end{bmatrix} \xrightarrow{\text{除以w}} \begin{bmatrix} \dfrac{x}{rz\tan\dfrac{\alpha}{2}}, \dfrac{y}{z\tan\dfrac{\alpha}{2}}, A+\dfrac{B}{z}, 1 \end{bmatrix} \quad \text{（公式 5.3）}$$

顺便提一句，你可能会问：“如何处理除数为 0 的情况”；对于一问题我们不必担心，因为近平面总是大于 0 的，其他的点都会被裁剪掉（参见 5.9 节）。有时，与 w 相除的过程也称为透视除法（perspective divide）或齐次除法（homogeneous divide）。我们可以看到 $x$、$y$ 的投影坐标与方程 5.1 相同。

## 5.6.3.5 规范化深度值

你可能认为在投影之后可以丢弃原始的 3D z 坐标，因为所有的投影点已经摆放在 2D 投影窗口上，形成了我们最终看到的 2D 图像，不会再使用 3D z 坐标了。其实不然，我们仍然需要为深度缓存算法提供 3D 深度信息。就如同 Direct3D 希望我们把 x、y 投影坐标映射到一个规范化区间一样，Direct3D 也希望我们将深度坐标映射到一个规范化区间[0,1]中。所以，我们需要创建一个保序函数（order preserving function）g(x)把[*n*,*f*]区间映射到[0,1]区间。由于该函数是保序的，所以当 $z_1,z_2\in[n,f]$且 $z_1<z_2$ 时，必有 $g(z_1)<g(z_2)$。这样，即使深度值已经被变换过了，相对的深度关系还是会被完好无损地保留下来，我们依然可以在规范化区间中得到正确的深度测试结果，这就是我们要为深度缓存算法做的全部工作。

通过缩放和平移可以实现从[*n* ,*f*]到[0,1]的映射。但是，这种方式无法与我们当前的投影方程整合。我们可以从方程 5.3 中看到经过变换的 z 坐标为：

$$g(z) = A + \frac{B}{z}$$

我们现在需要让 A 和 B 满足以下条件：

- 条件 1：g(n) = A + B/n = 0（近平面映射为 0）
- 条件 2：g(f) = A +B/f = 1（远平面映射为 1）

由条件 1 得到 B 的结果为：B= −An。把它代入条件 2，得到 A 的结果为：

$$A + \frac{-An}{f} = 1$$

$$\frac{Af - An}{f} = 1$$

$$Af - An = f$$

$$A = \frac{f}{f - n}$$

所以，

$$g(x) = \frac{f}{f - n} - \frac{nf}{(f - n)z}$$

从 g(z)的曲线图（图 5.25）中可以看出，它会限制增长的幅度（保序）而且是非线性的。从图中我们还可以看到，区间中的大部分取值落在近平面附近。因此，大多数深度值被映射到了一个很窄的取值范围内。这会导致深度缓冲区出现精度问题（由于所能表示的数值范围有限，计算机将无法识别变换后的深度值之间的微小差异）。通常的建议是让近平面和远平面尽可能接近，把深度的精度性问题减小到最低程度。

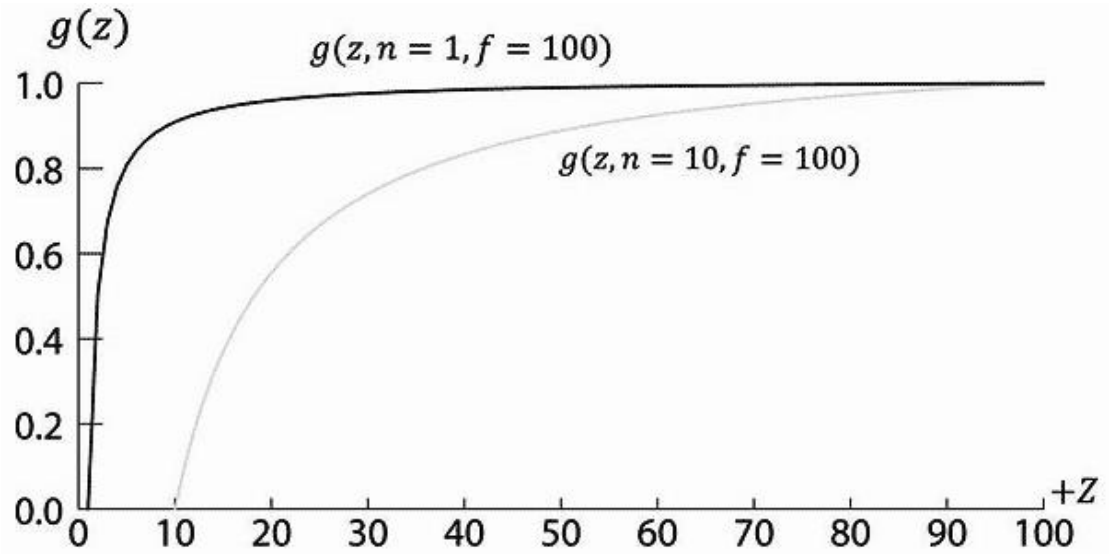



**图 5.25 相对于不同近平面的 g(z)曲线图。**

现在我们已经解出了 A 和 B，我们可以确定出完整的透视投影矩阵：

$$\mathbf{P}=\begin{bmatrix} \frac{1}{r\tan\frac{\alpha}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\frac{\alpha}{2}} & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & 1 \\ 0 & 0 & \frac{-nf}{f-n} & 0 \end{bmatrix}$$

在与投影矩阵相乘之后，进行透视除法之前，几何体所处的空间称为齐次裁剪空间（homogeneous clip space）或投影空间（projection space）。在透视除法之后，几何体所处的空间称为规范化设备空间（normalized device coordinates，简称 NDC）。

## 5.6.3.6 XMMatrixPerspectiveFovLH

透视投影矩阵可由如下 XNA 函数生成：

```
XMMATRIX XMMatrixPerspective FovLH(// returns projection matrix
    FLOAT FovAngleY, // vertical field of view angle in radians
    FLOAT AspectRatio, // aspect ratio =  width / height
    FLOAT NearZ, // distance to near plane
    FLOAT FarZ);  // distance to far plane
```

下面的代码片段示范了 XMMatrixPerspectiveFovLH 函数的使用方法。这里，我们将垂直视域角设为 45°，近平面 z 设为 1，远平面 z 设为 1000（这些长度是在观察空间中的）。

```
XMMATRIX P = XMMatrixPerspectiveFovLH(0.25f*MathX::Pi,
    AspectRatio(),1.0f,1000.0f);
```

横纵比要匹配窗口的横纵比：

```
float D3Dapp::AspectRatio() const
{
    return static_cast<float>(mClientWidth)/mClientHeight;
}
```

# 5.7 曲面细分阶段

曲面细分（Tessellation）是指通过添加三角形的方式对一个网格的三角形进行细分，这些新添加的三角形可以偏移到一个新的位置，让网格的细节更加丰富。（见图 5.26）。

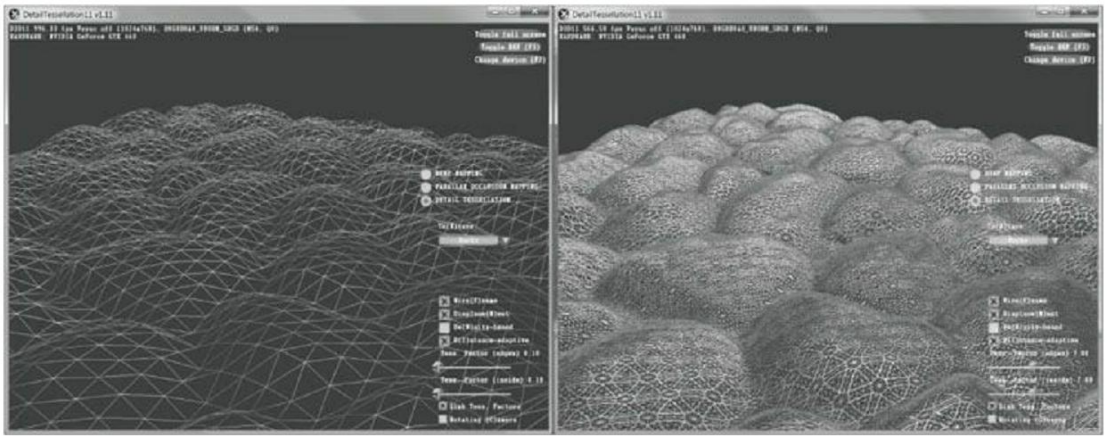


**图 5.26 左图是原始网格，右图是经过曲面细分处理后的网格**

下面是曲面细分的一些优点：

1．我们可以通过曲面细分实现细节层次（level -of-detai l，LOD），使靠近相机的三角形通过细分产生更多细节，而那些远离相机的三角形则保持不变。通过这种方式，我们只需在需要细节的地方使用更多的三角形就可以了。

2．我们可以在内存中保存一个低细节（低细节意味着三角形数量少）的网格，但可以实时地添加额外的三角形，这样可以节省内存。

3．我们可以在一个低细节的网格上处理动画和物理效果，而只在渲染时才使用细分过的高细节网格。

曲面细分阶段是 Direct3D 11 中新添加的，这样我们就可以在 GPU 上对几何体进行细分了。而在 Direct3D 11 之前，如果你想要实现曲面细分，则必须在 CPU 上完成，经过细分的几何体还要发送到 GPU 用于渲染。然而，将新的几何体从 CPU 内存发送到显存是很慢的，而且还会增加 CPU 的负担。因此，在 Direct3D 11 出现之前，曲面细分的方法在实时图形中并不流行。Direct3D 11 提供了一个可以完全在硬件上实现的曲面细分 API。这样曲面细分就成为了一个非常有吸引力的技术了。曲面细分阶段是可选的（即在需要的时候才使用它）。我们要在第 13 章才会详细介绍曲面细分。

# 5.8 几何着色器阶段

几何着色器阶段（geometry shader stage）是可选的，我们在第 11 章之前不会用到它，所以这里只做一个简短的概述。几何着色器以完整的图元作为输入数据。例如，当我们绘制三角形列表时，输入到几何着色器的数据是构成三角形的三个点。（注意，这三个点是从顶点着色器传递过来的。）几何着色器的主要优势是它可以创建或销毁几何体。例如，输入图元可以被扩展为一个或多个其他图元，或者几何着色器可以根据某些条件拒绝输出某些图元。这一点与顶点着色器有明显的不同：顶点着色器无法创建顶点，只要输入一个顶点，那么就必须输出一个顶点。几何着色器通常用于将一个点扩展为一个四边形，或者将一条线扩展为一个四边形。

我们可以在图 5.11 中看到一个“流输出（stream output）”箭头。也就是，几何着色器可以将顶点数据流输出到内存中的一个顶点缓冲区内，这些顶点可以在管线的随后阶段中渲染出来。这是一项高级技术，我们会在后面的章节中对它进行讨论。

**注意**：顶点位置在离开几何着色器之前，必须被变换到齐次裁剪空间。

# 5.9 裁剪阶段

我们必须完全丢弃在平截头体之外的几何体，裁剪与平截头体边界相交的几何体，只留下平截头体内的部分；图 5.27 以 2D 形式说明了一概念。

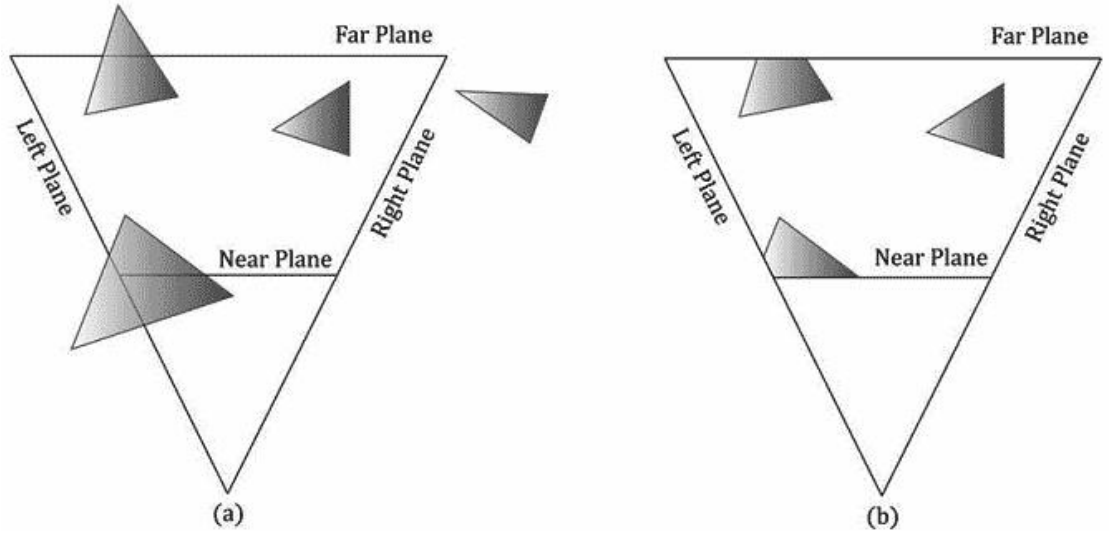



**图 5.27 (a)裁剪之前。(b)裁剪之后。**

我们可以将平截头体视为由 6 个平面界定的空间范围：顶、底、左、右、近、远平面。要裁剪与平截头体方向相反的多边形，其实就是逐个裁剪与每个平截头体平面方向相反的多边形，当裁剪一个与平面方向相反的多边形时（参见图 5.28），我们将保留平面正半空间中的部分，而丢弃平面负半空间中的部分。对一个与平面方向相反的凸多边形进行裁剪，得到的结果仍然会是一个凸多边形。由于硬件会自动完成所有的裁剪工作，所以我们不在这里讲解具体的实现细节；有兴趣的读者可以参阅[Sutherland74]，了解一下目前流行的 Sutherland-Hodgeman 裁剪算法。它基本思路是：求出平面与多边形边之间的交点，然后对顶点进行排序，形成新的裁剪后的多边形。

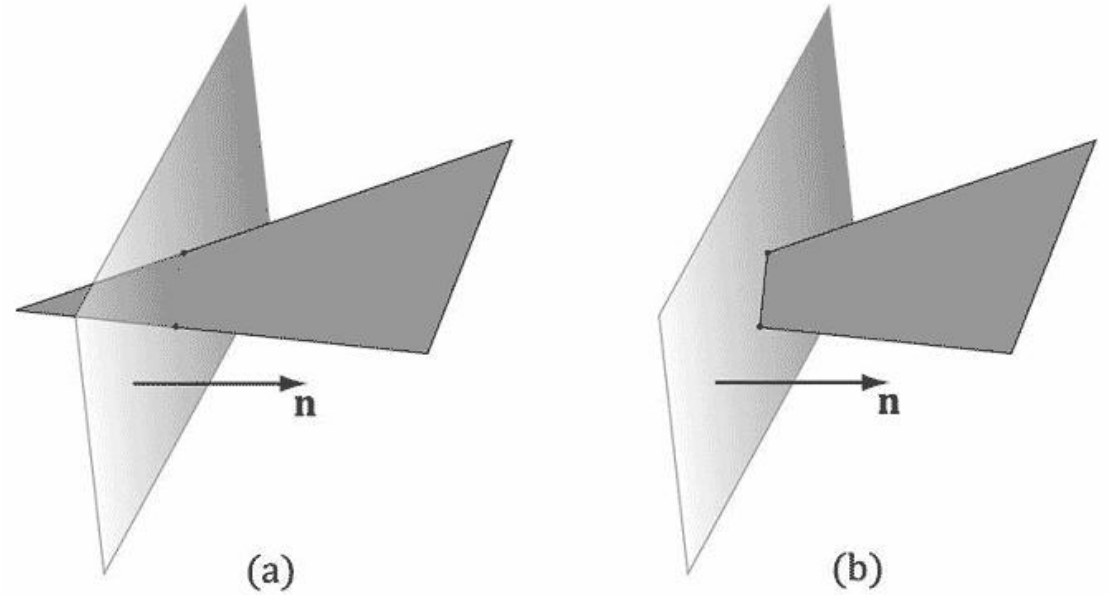



**图 5.28 (a)裁剪一个与平面方向相反的三角形。(b)裁剪后的三角形。注意，裁剪后的三角形已经不再是一个三角形了，它是一个四边形。所以，硬件必须将个四边形重新划分为三角**

**形，对于凸多边形来说这是一个非常简单的处理过程。**

[Blinn78]描述了如何在 4D 齐次空间中实现裁剪算法（图 5.29）。在透视除法之后，平截头体内的点$(x/w, y/w, z/w, 1)$将位于规范化设备空间，它的边界如下：

$-1 \leqslant x/w \leqslant 1$

$-1 \leqslant y/w \leqslant 1$

$0 \leqslant z/w \leqslant 1$

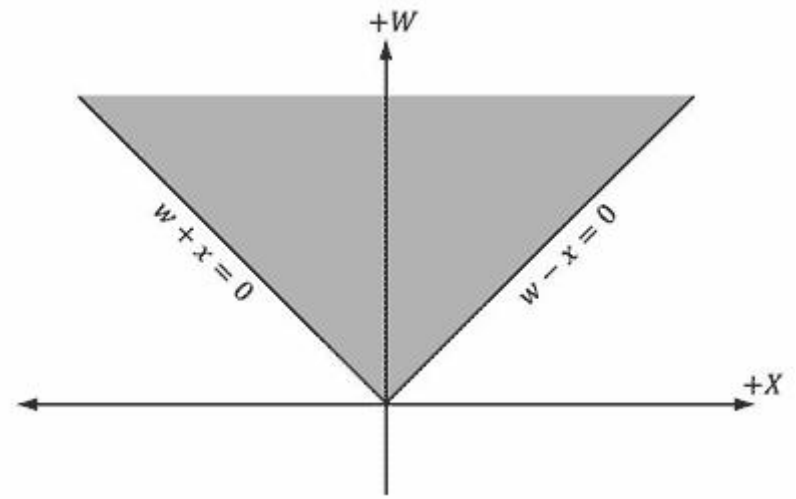



**图 5.29 齐次裁剪空间中 xw 平面上的截头体边界**

那么在透视除法之前，平截头体内的 4D 点$(x, y, z, w)$在齐次裁剪空间中的边界为：

$-w \leqslant x \leqslant w$

$-w \leqslant y \leqslant w$

$0 \leqslant z \leqslant w$

也就是，顶点被限定在以下 4D 平面构成的空间范围内：

左：$w = -x$

右：$w = x$

底：$w = -y$

顶：$w = y$

近：$z = 0$

远：$z = w$

只要我们知道齐次剪裁空间中的平截头体平面方程，我们就能使用任何一种裁剪算法（比如 Sutherland-Hodgeman）。注意，由线段/平面相交测试的数学推论可知，这个测试在 $\mathbb{R}^4$ 也能使用，所以我们可以在齐次裁剪空间中进行 4D 点和 4D 平面的相交测试。

# 5.10 光栅化阶段

光栅化（rasterization）阶段的主要任务是为投影后的 3D 三角形计算像素颜色。

## 5.10.1 视口变换

在裁剪之后，硬件会自动执行透视除法，将顶点从齐次裁剪空间变换到规范化设备空间（NDC）。一旦顶点进入 NDC 空间，构成 2D 图像的 2D *x*、*y* 坐标就会被变换到后台缓冲区中的一个称为视口的矩形区域内（回顾 4.2.8 节）。在该变换之后，*x*、*y* 坐标将以像素为单位。通常，视口变换不修改 *z* 坐标，因为 *z* 坐标还要由深度缓存使用，但是我们可以通过 **D3D11_VIEWPORT** 结构体的 **MinDepth** 和 **MaxDepth** 值修改 z 坐标的取值范围。**MinDepth** 和 **MaxDepth** 的值必须在 0 和 1 之间。

## 5.10.2 背面消隐

一个三角形有两个面。我们使用如下约定来区分这两个面。假设三角形的顶点按照 $\mathbf{v}_0$、$\mathbf{v}_1$、$\mathbf{v}_2$ 的顺序排列，我们这样来计算三角形的法线 **n**：

$\mathbf{e}_0 = \mathbf{v}_1 - \mathbf{v}_0$

$\mathbf{e}_1 = \mathbf{v}_2 - \mathbf{v}_1$

$$\mathbf{n} = \frac{\mathbf{e}_0 \times \mathbf{e}_1}{\|\mathbf{e}_0 \times \mathbf{e}_1\|}$$

带有法线向量的面为正面，而另一个面为背面。图 5.30 说明了这一概念。

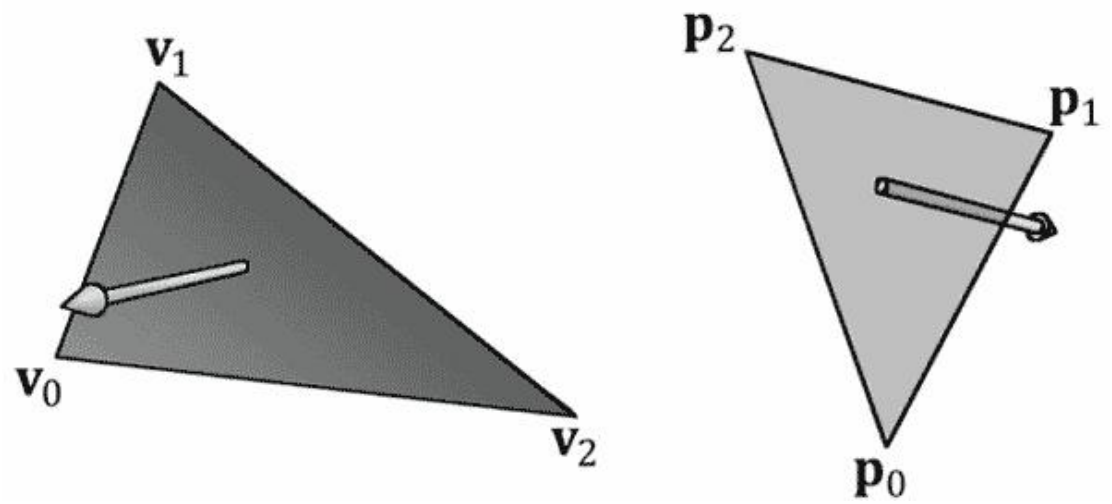



**图 5.30 左边的三角形正对我们的观察点，而右边的三角形背对我们的观察点。**

当观察者看到三角形的正面时，我们说三角形是朝前的；当观察者看到三角形的背面时，我们说三角形是朝后的。如图 5.30 所示，左边的三角形是朝前的，而右边的三角形是朝后的。而且，按照我们的观察角度，左边的三角形会按顺时针方向环绕，而右边的三角形会按逆时针方向环绕。这不是巧合：因为按照我们选择的约定（即，我们计算三角形法线的方式），按顺时针方向环绕的三角形（相对于观察者）是朝前的，而按逆时针方向环绕的三角形（相

对于观察者）是朝后的。

现在，3D 空间中的大部分物体都是封闭实心物体。当我们按照这一方式将每个三角形的法线指向物体外侧时，摄像机就不会看到实心物体朝后的三角形，因为朝前的三角形挡住了朝后的三角形；图 5.31 和图 5.32 分别以 2D 和 3D 形式说明了一概念。由于朝前的三角形挡住了朝后的三角形，所以绘制它们是毫无意义的。背面消隐（backface culling）是指让管线放弃对朝后的三角形的处理。这可以将所要处理的三角形的数量降低到原数量的一半。

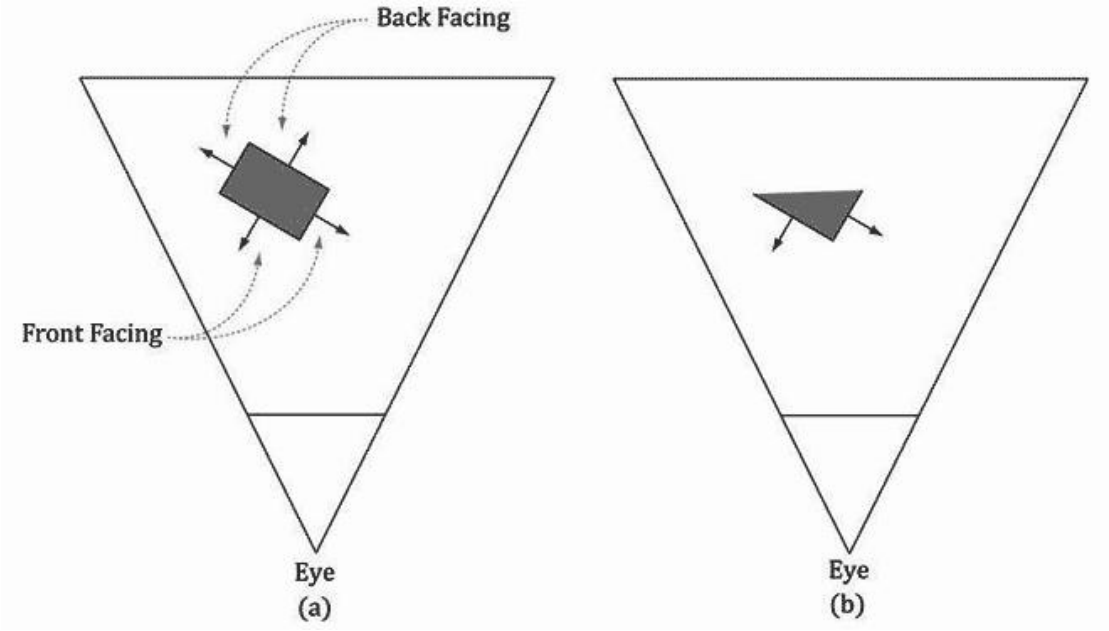



**图 5.31 (a)一个带有朝前和朝后三角形的实心物体。(b)在剔除了朝后的三角形之后的场景。注意，背面消隐不会影响最终的图像，因为朝后的三角形会被朝前的三角形阻挡。**

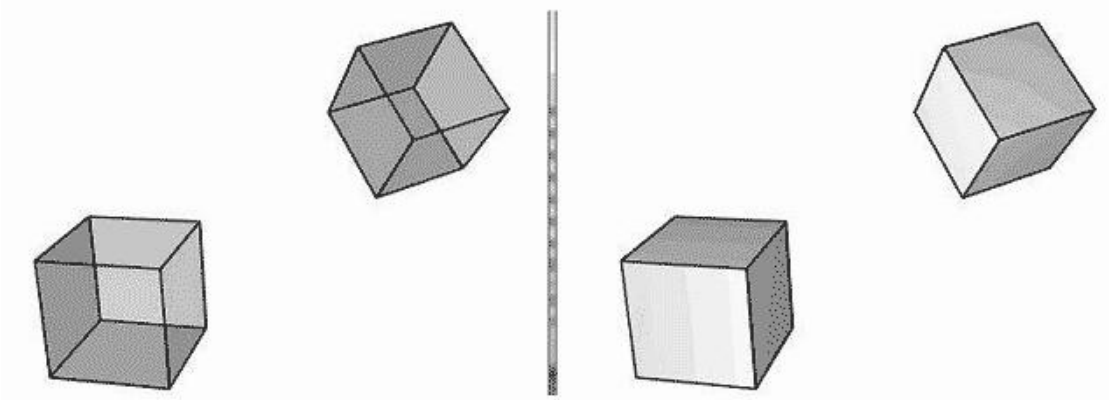


**图 5.32 （左图）当以透明方式绘制立方体时，我们可以看到所有的 6 个面。（右图）当以实心方式绘制立方体时，我们无法看到朝后的 3 个面，因为朝前的 3 个面挡住了它们——所以朝后的三角形可以被直接丢弃，不再接受后续处理，没人能看到些朝后的三角形。**

默认情况下，Direct3D 将（相对于观察者）顺时针方向环绕的三角形视为朝前的三角形，将（相对于观察者）逆时针方向环绕的三角形视为朝后的三角形。不过，这一约定可以通过修改 Direct3D 渲染状态颠倒过来。

## 5.10.3 顶点属性插值

如前所述，我们通过指定三角形的 3 个顶点来定义一个三角形。除位置外，顶点还可以

包含其他属性，比如颜色、法线向量和纹理坐标。在视口变换之后，这些属性必须为三角形表面上的每个像素进行插值。顶点深度值也必须进行插值，以使每个像素都有一个可用于深度缓存算法的深度值。对屏幕空间中的顶点属性进行插值，其实就是对 3D 空间中的三角形表面进行线性插值（如图 5.33 所示）；这一工作需要借助所谓的透视矫正插值（perspective correct interpolation）来实现。本质上，三角形表面内部的像素颜色都是通过顶点插值得到的。

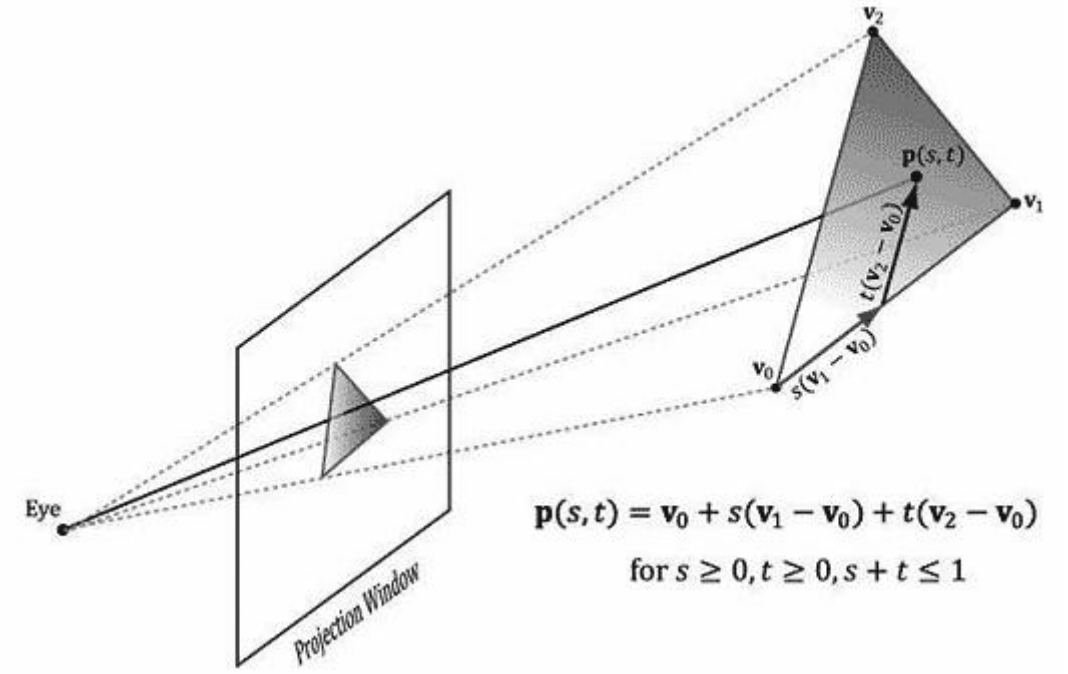



**图 5.33：通过对三角形顶点之间的属性值进行线性插值，可以得到三角形表面上的任一属性值 p(s,t)。**

我们不必关心透视精确插值的数学细节，因为硬件会自动完成这一工作；不过，有兴趣的读者可以在[Eberly01]中查阅相关的数学推导过程。图 5.34 介绍了一点基本思路：

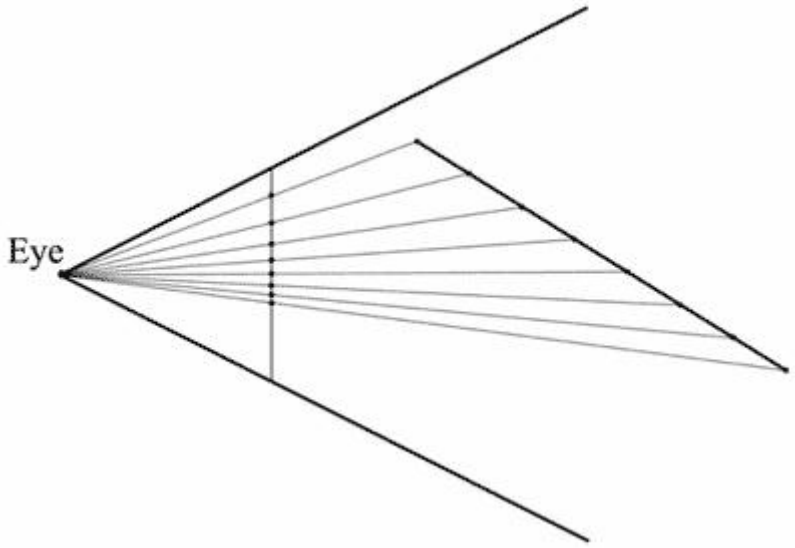



**图 5.34 一条 3D 线被投影到投影窗口上（在屏幕空间中投影是一条 2D 线）。我们看到，在 3D 线上取等距离的点，在 2D 屏幕空间上的投影点却不是等距离的。所以，我们在 3D 空间中执行线性插值，在屏幕空间需要执行非线性插值。**

# 5.11 像素着色器阶段

像素着色器（Pixel shader）是由我们编写的在 GPU 上执行的程序。像素着色器会处理每个像素片段（pixel fragment），它的输入是插值后的顶点属性，由此计算出一个颜色。像素着色器可以非常简单地输出一个颜色，也可以很复杂，例如实现逐像素光照、反射和阴影等效果。

# 5.12 输出合并阶段

当像素片段由像素着色器生成之后，它们会被传送到渲染管线的输出合并（output merger，简称 OM）阶段。在该阶段中，某些像素片段会被丢弃（例如，未能通过深度测试或模板测试）。未丢弃的像素片段会被写入后台缓冲区。混合（blending）工作是在该阶段中完成的，一个像素可以与后台缓冲区中的当前像素进行混合，并以混合后的值作为该像素的最终颜色。某些特殊效果，比如透明度，就是通过混合来实现的；我们会在第 9 章专门讲解混合。

# 5.13 小结

**1.** 我们可以根据人眼的视觉特性，在 2D 图像上模拟 3D 场景。我们发现平行线会汇集为一个零点（或称消失点），物体的尺寸会随着深度的增加而减小，一个物体会挡住它后面的其他物体，灯光和阴影可以表现物体的立体感和体积感，阴影可以体现光源的位置并烘托物体之间的层次关系。

**2.** 我们使用三角形网络来模拟物体。我们可以通过指定三角形的 3 个顶点来定义一个三角形。在许多网格中，顶点是由多个三角形共享的；索引列表可以用于避免顶点重复。

**3.** 颜色可以通过红、绿、蓝的强度来描述。通过将这三种颜色以不同的强度进行混合，可以得到上千万种不同的颜色。我们通常使用规范化区间[0,1]描述红、绿、蓝的强度。0 表示没有强度，1 表示最高强度，中间值表示中等强度。通常在颜色中还会包含一个称为 alpha 分量的附加分量，它用于表示颜色的不透明度。当使用混合时，alpha 分量非常有用。我们可以使用 4D 向量(r ,g ,b ,a )来表示带有 alpha 分量的颜色，其中 $0 \le r, g, b, a \le 1$。在 Direct3D 中，颜色由 **XMVECTOR** 类来表示，使用 XNA 数学库进行颜色操作可以发挥 SIMD 的优势。如果用 32 位表示颜色，则每个分量占一个字节，XNA 数学库提供了 **XMCOLOR** 结构用于存储 32 位颜色。颜色向量可以像普通向量那样进行加法、减法和标量乘法运算，只是每个分量的取值范围必须限定在[0, 1]区间内（或 32 位颜色的[0,255]区间内）。其他的向量运算（比如点积和叉积）对颜色向量来说没有意义。符号⊗表示分量乘法，它的含义为：$(c_1, c_2, c_3, c_4) \otimes (k_1, k_2, k_3, k_4) = (c_1k_1, c_2k_2, c_3k_3, c_4k_4)$。

**4.** 渲染管线是指：在给定一个 3D 场景的几何描述及一架已确定位置和方向的虚拟摄像机时，根据虚拟摄像机的视角生成 2D 图像的一系列步骤。

**5.** 渲染管线可以被分解为以下主要阶段：输入装配（IA）阶段；顶点着色器（VS）阶段；曲面细分阶段；几何着色器（GS）阶段；裁剪阶段、光栅化（RS）阶段、像素着色器（PS）阶段和输出合并器（OM）阶段。

# 6.1 顶点和顶点布局

5.5.1 节已经讲过，在 Direct3D 中，顶点由空间位置和各种附加属性组成，Direct3D 可以让我们灵活地建立属于我们自己的顶点格式；换句话说，它允许我们定义顶点的分量。要创建一个自定义的顶点格式，我们必须先创建一个包含顶点数据的结构体。例如，下面是两种不同类型的顶点格式；一个由位置和颜色组成，另一个由位置、法线和纹理坐标组成。

```
struct Vertex1
{
    XMFLOAT3 Pos;
    XMFLOAT4 Color;
};
struct Vertex2
{
    XMFLOAT3 Pos;
    XMFLOAT3 Normal;
    XMFLOAT2 Tex0;
    XMFLOAT2 Tex1;
};
```

在定义了顶点结构体之后，我们必须设法描述该顶点结构体的分量结构，使 Direct3D 知道该如何使用每个分量。这一描述信息是以输入布局（**ID3D11InputLayout**）的形式提供给 Direct3D 的。输入布局是一个 **D3D11_INPUT_ELEMENT_DESC** 数组。**D3D11_INPUT_ELEMENT_DESC** 数组中的每个元素描述了顶点结构体的一个分量。比如，当顶点结构体包含两个分量时，对应的 **D3D11_INPUT_ELEMENT_DESC** 数组会包含两个元素。我们将 **D3D11_INPUT_ELEMENT_DESC** 称为输入布局描述（input layout description）。**D3D11_INPUT_ELEMENT_DESC** 结构体定义如下：

```
typedef struct D3D11_INPUT_ELEMENT_DESC {
    LPCSTR SemanticName;
    UINT SemanticIndex;
    DXGI_FORMAT Format;
    UINT InputSlot;
    UINT AlignedByteOffset
    D3D11_INPUT_CLASSIFICATION InputSlotClass;
    UINT InstanceDataStepRate;
} D3D11_INPUT_ELEMENT_DESC;
```

**1．SemanticName**：一个与元素相关的字符串。它可以是任何有效的语义名。语义（semantic）用于将顶点结构体中的元素映射为顶点着色器参数（参见图 6.1）。

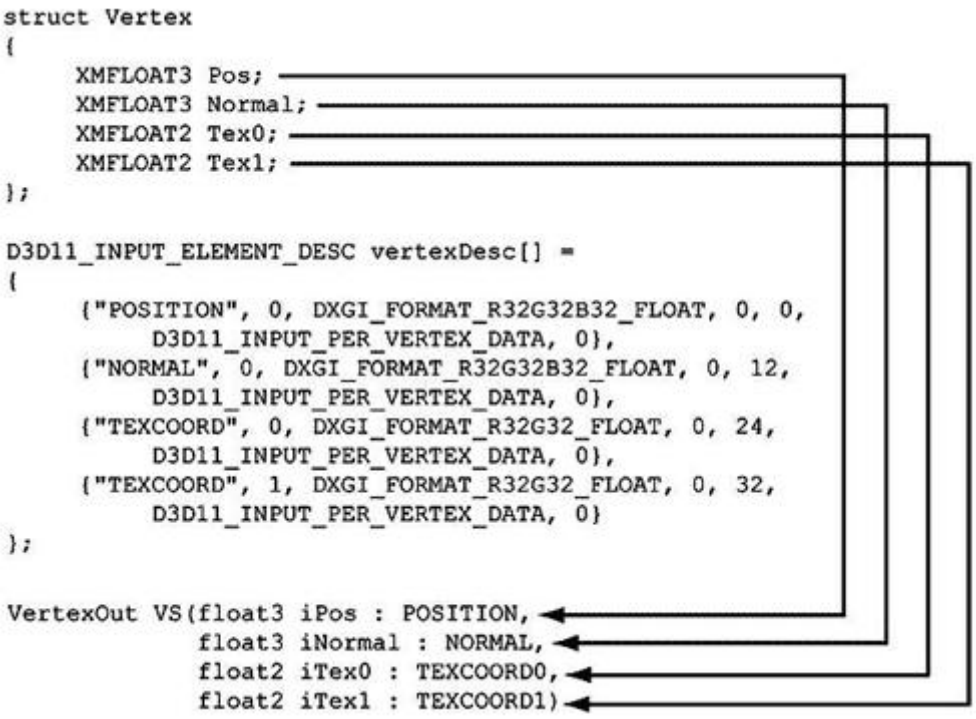



**图 6.1 顶点结构体中的每个元素分别由 D3D11_INPUT_ELEMENT_DESC 数组中的对应元素描述。语义名和语义索引提供了一种将顶点元素映射为顶点着色器参数的方法。**

**2．SemanticIndex**：附加在语义上的索引值。图 6.1 说明了使用该索引的原因；举例来说，当顶点结构体包含多组纹理坐标时，我们不是添加一个新的语义名，而是在语义名的后面加上一个索引值。在着色器代码中没有指定索引的语义默认索引为 0，例如，在图 6.1 中的 **POSITION** 相当于 **POSITION0**。

**3．Format**：一个用于指定元素格式的 DXGI_FORMAT 枚举类型成员；下面是一些常用的格式：

- DXGI_FORMAT_R32_FLOAT // 1D 32-bit float scalar
- DXGI_FORMAT_R32G32_FLOAT // 2D 32-bit float vector
- DXGI_FORMAT_R32G32B32_FLOAT // 3D 32-bit float vector
- DXGI_FORMAT_R32G32B32A32_FLOAT // 4D 32-bit float vector
- DXGI_FORMAT_R8_UINT // 1D 8-bit unsigned integer scalar
- DXGI_FORMAT_R16G16_SINT // 2D 16-bit signed integer vector
- DXGI_FORMAT_R32G32B32_UINT // 3D 32-bit unsigned integer vector
- DXGI_FORMAT_R8G8B8A8_SINT // 4D 8-bit signed integer vector
- DXGI_FORMAT_R8G8B8A8_UINT // 4D 8-bit unsigned integer vector

**4．InputSlot**：指定当前元素来自于哪个输入槽（input slot）。Direct3D 支持 16 个输入槽（索引依次为 0 到 15），通过这些输入槽我们可以向着色器传入顶点数据。例如，当一个顶点由位置元素和颜色元素组成时，我们既可以使用一个输入槽传送两种元素，也可以将两种元素分开，使用第一个输入槽传送顶点元素，使用第二个输入槽传送颜色元素。Direct3D 可以将来自于不同输入槽的元素重新组合为顶点。在本书中，我们只使用一个输入槽，但是在本章结尾的练习 2 中我们会引导读者做一个使用两个输入槽的练习。

**5．AlignedByteOffset**：对于单个输入槽来说，该参数表示从顶点结构体的起始位置到顶点元素的起始位置之间的字节偏移量。例如在下面的顶点结构体中，元素 **Pos** 的字节偏移量为 0，因为它的起始位置与顶点结构体的起始位置相同；元素 **Normal** 的字节偏移量为 12，

因为必须跳过由 **Pos** 占用的字节才能到达 **Normal** 的起始位置；元素 **Tex0** 的字节偏移量为 24，因为必须跳过由 **Pos** 和 **Normal** 占用的字节才能到达 **Tex0** 的起始位置；元素 **Tex1** 的字节偏移量为 32，因为必须跳过由 **Pos**，**Normal** 和 **Tex0** 占用的字节才能到达 **Tex1** 的起始位置。

```
struct Vertex2
{
    XMFLOAT3 Pos;       // 0-byte offset
    XMFLOAT3 Normal;    // 12-byte offset
    XMFLOAT2 Tex0;      // 24-byte offset
    XMFLOAT2 Tex1;      // 32-byte offset
};
```

**6．InputSlotClass**：目前指定为 **D3D11_INPUT_PER_VERTEX_DATA**；其他选项用于高级实例技术。

**7．InstanceDataStepRate**：目前指定为 0；其他值只用于高级实例技术。

对于前面的两个示例顶点结构体 Vertex1 和 Vertex2 来说，对应的输入布局描述为：

```
D3D11_INPUT_ELEMENT_DESC desc1[]=
{
    {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D11_INPUT_PER_VERTEX_DATA, 0},
    {"COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
        D3D11_INPUT_PER_VERTEX_DATA, 0}
};

D3D11_INPUT_ELEMENT_DESC desc2[]=
{
    {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D11_INPUT_PER_VERTEX_DATA, 0},
    {"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT,0, 12,
        D3D11_INPUT_PER_VERTEX_DATA, 0},
    {"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
        D3D11_INPUT_PER_VERTEX_DATA, 0}
    {"TEXCOORD", 1, DXGI_FORMAT_R32G32_FLOAT, 0, 32,
        D3D11_INPUT_PER_VERTEX_DATA, 0}
};
```

指定了输入布局描述之后，我们就可以使用 **ID3D11Device::CreateInputLayout** 方法获取一个表示输入布局的 **ID3D11InputLayout** 接口的指针：

```
HRESULT ID3D11Device::CreateInputLayout(
    const D3D11_INPUT_ELEMENT_DESC *pInputElementDescs,
    UINT NumElements,
    const void *pShaderBytecodeWithInputSignature,
    SIZE_T BytecodeLength,
    ID3D11InputLayout **ppInputLayout);
```

**1．pInputElementDescs**：一个用于描述顶点结构体的 D3D11_INPUT_ELEMENT_DESC 数组。

**2．NumElements：D3D11_INPUT_ELEMENT_DESC** 数组的元素数量。

**3．pShaderBytecodeWithInputSignature：**指向顶点着色器参数的字节码的指针。

**4．BytecodeLength：**顶点着色器参数的字节码长度，单位为字节。

**5．ppInputLayout：**返回创建后的 **ID3D11InputLayout** 指针。

我们需要进一步解释一下第 3 个参数的含义。本质上，顶点着色器以一组顶点元素作为它的输入参数——也就是所谓的输入签名（input signature）。自定义顶点结构体中的元素必须被映射为与它们对应的顶点着色器参数，图 6.1 解释了这一问题。通过在创建输入布局时传入顶点着色器签名，Direct3D 在创建时就可以验证输入布局是否与输入签名匹配，并建立从顶点结构体到着色器参数之间的映射关系。一个 **ID3D11InputLayout** 对象可以在多个参数完全相同的着色器中重复使用。

假设有下列输入参数和顶点结构：

```
VertexOut VS(float3 Pos:POSITION, float4 Color:COLOR,
    float3 Normal: NORMAL){ }
struct Vertex
{
    XMFLOAT3  Pos ;
    XMFLOAT4  Color;
};
```

这样会产生错误，VC++的调试输出窗口会显示以下信息：

```
D3D11:ERROR:ID3D11Device::CreateInputLayout:The provided input
signature expects to read an element with SemanticName/Index:
'NORMAL'/0, but the declaration doesn't provide a matching name.
```

假如顶点结构和输入参数与输入元素匹配，但类型不同：

```
VertexOut VS(int3 Pos:POSITION, float4 Color:COLOR) { }
struct Vertex
{
    XMFLOAT3  Pos;
    XMFLOAT4  Color;
} ;
```

这样做是可行的，因为 Direct3D 允许输入寄存器中的字节被重新解释。但是，VC ++调试输出窗口会显示以下信息：

```
D3D11:WARNING:ID3D11Device::CreateInputLayout:The provided input
signature expects to read an element with SemanticName/Index:
'POSITION'/0 and component(s) of the type 'int32'. However,the
matching  entry in the InputLayout declaration, element[0],
specifies mismatched format:'R32G32B32_FLOAT'.This is not an error,
since behavior is well defined :The element format determines what
data conversion algorithm gets applied before it shows up in a
shader register.Independently,the shader input signature defines
how the shader will interpret the data that has been placed in its
input registers,with no change in the bits stored.It is valid for
the application to reinterpret data as a different type once it is
in the vertex shader,so this warning is issued just in case reinte
```

```
rpretation  was not intended by the author.
```

下面的代码说明了该如何调用 **ID3D11Device::CreateInputLayout** 方法。注意，这些代码涉及了一些我们还未讨论的内容（比如 **ID3D11Effect**）。本质上，一个 effect 可以封装一个或多个 pass，而每个 pass 都会与一个顶点着色器相连。所以，我们可以从 effect 中得到有关 pass 的描述信息（**D3D11_PASS_DESC**），然后再从中得到顶点着色器的输入签名。

```
ID3D11Effect* mFX;
ID3D11EffectTechnique* mTech;
ID3D11InputLayout* mVertexLayout;
/* ...create the effect... */
mTech = mFX->GetTechniqueByName("Tech");
D3D11_PASS_DESC PassDesc;
mTech->GetPassByIndex(0)->GetDesc(&PassDesc);
HR(md3dDevice->CreateInputLayout(vertexDesc, 4,
    PassDesc.pIAInputSignature, PassDesc.IAInputSignatureSize,
    &mVertexLayout));
```

创建了输入布局对象之后，它不会自动绑定到设备上。我们必须调用下面的语句来实现绑定：

```
ID3D11InputLayout* mVertexLayout;
/* ...create the input layout... */
md3dImmediateContext->IASetInputLayout(mVertexLayout);
```

如果你打算用一个输入布局来绘制一些物体，然后再使用另一个的布局来绘制另一些物体，那你必须按照下面的形式来组织代码：

```
md3dImmediateContext->IASetInputLayout(mVertexLayout1);
/* ...draw objects using input layout 1... */
md3dImmediateContext->IASetInputLayout(mVertexLayout2);
/* ...draw objects using input layout 2... */
```

换句话说，当一个 **ID3D11InputLayout** 对象被绑定到设备上时，如果不去改变它，那么它会始终驻留在那里。

# 6.2 顶点缓冲

为了让 GPU 访问顶点数组，我们必须把它放置在一个称为缓冲（buffer）的特殊资源容器中，该容器由 **ID3D11Buffer** 接口表示。

用于存储顶点的缓冲区称为顶点缓冲（vertex buffer）。Direct3D 缓冲不仅可以存储数据，而且还说明了如何访问数据以及数据被绑定到图形管线的那个阶段。要创建一个顶点缓冲，我们必须执行以下步骤：

1．填写一个 **D3D11_BUFFER_DESC** 结构体，描述我们所要创建的缓冲区。

2．填写一个 **D3D11_SUBRESOURCE_DATA** 结构体，为缓冲区指定初始化数据。

3．调用 **ID3D11Device::CreateBuffer** 方法来创建缓冲区。

**D3D11_BUFFER_DESC** 结构体的定义如下：

```
typedef struct D3D11_BUFFER_DESC{
    UINT ByteWidth;
    D3D11_USAGE Usage;
    UINT BindFlags;
    UINT CPUAccessFlags;
    UINT MiscFlags;
    UINT StructureByteStride;
} D3D11_BUFFER_DESC;
```

**1．ByteWidth**：我们将要创建的顶点缓冲区的大小，单位为字节。

**2．Usage**：一个用于指定缓冲区用途的 **D3D11_USAGE** 枚举类型成员。有 4 个可选值：

（a）**D3D10_USAGE_DEFAULT**：表示 GPU 会对资源执行读写操作。在使用映射 API（例如 **ID3D11DeviceContext::Map**）时，CPU 在使用映射 API 时不能读写这种资源，但它能使用 **ID3D11DeviceContext::UpdateSubresource**。**ID3D11DeviceContext::Map** 方法会在 6.14 节中介绍。

（b）**D3D11_USAGE_IMMUTABLE**：表示在创建资源后，资源中的内容不会改变。这样可以获得一些内部优化，因为 GPU 会以只读方式访问这种资源。除了在创建资源时 CPU 会写入初始化数据外，其他任何时候 CPU 都不会对这种资源执行任何读写操作，我们也无法映射或更新一个 immutable 资源。

（c）**D3D11_USAGE_DYNAMIC**：表示应用程序（CPU）会频繁更新资源中的数据内容（例如，每帧更新一次）。GPU 可以从这种资源中读取数据，使用映射 API（**ID3D11DeviceContext::Map**）时，CPU 可以向这种资源中写入数据。因为新的数据要从 CPU 内存（即系统 RAM）传送到 GPU 内存（即显存），所以从 CPU 动态地更新 GPU 资源会有性能损失；若非必须，请勿使用 **D3D11_USAGE_DYNAMIC**。

（d）**D3D11_USAGE_STAGING**：表示应用程序（CPU）会读取该资源的一个副本（即，该资源支持从显存到系统内存的数据复制操作）。显存到系统内存的复制是一个缓慢的操作，应尽量避免。使用 **ID3D11DeviceContext::CopyResource** 和 **ID3D11DeviceContext::CopySubresourceRegion** 方法可以复制资源，在 12.3.5 节会介绍一个复制资源的例子。

**3．BindFlags**：对于顶点缓冲区，该参数应设为 **D3D11_BIND_VERTEX_BUFFER**。

**4．CPUAccessFlags**：指定 CPU 对资源的访问权限。设置为 0 则表示 CPU 无需读写缓冲。如果 CPU 需要向资源写入数据，则应指定 **D3D11_CPU_ACCESS_WRITE**。具有写访问权限的资源的 Usage 参数应设为 **D3D11_USAGE_DYNAMIC** 或

**D3D11_USAGE_STAGING**。如果 CPU 需要从资源读取数据，则应指定 **D3D11_CPU_ACCESS_READ**。具有读访问权限的资源的 Usage 参数应设为 **D3D11_USAGE_STAGING**。当指定这些标志值时，应按需而定。通常，CPU 从 Direct3D 资源读取数据的速度较慢。CPU 向资源写入数据的速度虽然较快，但是把内存副本传回显存的过程仍很耗时。所以，最好的做法是（如果可能的话）不指定任何标志值，让资源驻留在显存中，只用 GPU 来读写数据。

**5．MiscFlags**：我们不需要为顶点缓冲区指定任何杂项（miscellaneous）标志值，所以该参数设为 0。有关 **D3D11_RESOURCE_MISC_FLAG** 枚举类型的详情请参阅 SDK 文档。

**6．StructureByteStride**：存储在结构化缓冲中的一个元素的大小，以字节为单位。这个属性只用于结构化缓冲，其他缓冲可以设置为 0。所谓结构化缓冲，是指存储其中的元素大小都相等的缓冲。

**D3D11_SUBRESOURCE_DATA** 结构体的定义如下：

```
typedef struct D3D11_SUBRESOURCE_DATA {
    const void *pSysMem;
    UINT SysMemPitch;
    UINT SysMemSlicePitch;
} D3D11_SUBRESOURCE_DATA;
```

**1．pSysMem**：包含初始化数据的系统内存数组的指针。当缓冲区可以存储 n 个顶点时，对应的初始化数组也应至少包含 n 个顶点，从而使整个缓冲区得到初始化。

**2．SysMemPitch**：顶点缓冲区不使用该参数。

**3．SysMemSlicePitch**：顶点缓冲区不使用该参数。

下面的代码创建了一个只读的顶点缓冲区，并以中心在原点上的立方体的 8 顶点来初始化该缓冲区。之所以说该缓冲区是只读的，是因为当立方体创建后相关的几何数据从不改变——始终保持为一个立方体。另外，我们为每个顶点指定了不同的颜色；这些颜色将用于立方体着色，我们会在本章随后的小节中对此进行讲解。

```
//  定义在 d3dUtil.h 中的 Colors 命名空间
//
// #define XMGLOBALCONST extern CONST __declspec(selectany)
//   1. extern so there is only one copy of the variable, and not a separate
//      private copy in each .obj.
//   2. __declspec(selectany) so that the compiler does not complain about
//      multiple definitions in a .cpp file (it can pick anyone and discard
//      the rest because they are constant--all the same).

namespace Colors
{
    XMGLOBALCONST XMVECTORF32 White     = {1.0f, 1.0f, 1.0f, 1.0f};
    XMGLOBALCONST XMVECTORF32 Black     = {0.0f, 0.0f, 0.0f, 1.0f};
    XMGLOBALCONST XMVECTORF32 Red       = {1.0f, 0.0f, 0.0f, 1.0f};
    XMGLOBALCONST XMVECTORF32 Green     = {0.0f, 1.0f, 0.0f, 1.0f};
    XMGLOBALCONST XMVECTORF32 Blue      = {0.0f, 0.0f, 1.0f, 1.0f};
    XMGLOBALCONST XMVECTORF32 Yellow    = {1.0f, 1.0f, 0.0f, 1.0f};
    XMGLOBALCONST XMVECTORF32 Cyan      = {0.0f, 1.0f, 1.0f, 1.0f};
    XMGLOBALCONST XMVECTORF32 Magenta   = {1.0f, 0.0f, 1.0f, 1.0f};
```

```
    XMGLOBALCONST XMVECTORF32 Silver    = {0.75f, 0.75f, 0.75f, 1.0f};
    XMGLOBALCONST XMVECTORF32 LightSteelBlue = {0.69f, 0.77f, 0.87f, 1.0f};
}

// 创建顶点缓冲
Vertex vertices[] =
{
    {        XMFLOAT3(-1.0f,        -1.0f,        -1.0f),        (const
float*)&Colors::White   },
    {        XMFLOAT3(-1.0f,        +1.0f,        -1.0f),        (const
float*)&Colors::Black   },
    { XMFLOAT3(+1.0f, +1.0f, -1.0f), (const float*)&Colors::Red     },
    {        XMFLOAT3(+1.0f,        -1.0f,        -1.0f),        (const
float*)&Colors::Green   },
    { XMFLOAT3(-1.0f, -1.0f, +1.0f), (const float*)&Colors::Blue    },
    {        XMFLOAT3(-1.0f,        +1.0f,        +1.0f),        (const
float*)&Colors::Yellow  },
    { XMFLOAT3(+1.0f, +1.0f, +1.0f), (const float*)&Colors::Cyan    },
    { XMFLOAT3(+1.0f, -1.0f, +1.0f), (const float*)&Colors::Magenta }
};

D3D11_BUFFER_DESC vbd;
vbd.Usage = D3D11_USAGE_IMMUTABLE;
vbd.ByteWidth = sizeof(Vertex) * 8;
vbd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
vbd.CPUAccessFlags = 0;
vbd.MiscFlags = 0;
vbd.StructureByteStride = 0;
D3D11_SUBRESOURCE_DATA vinitData;
vinitData.pSysMem = vertices;

ID3D11Buffer * mVB;
HR(md3dDevice->CreateBuffer(&vbd, &vinitData, &mVB));
```

Vertex 类型和颜色由以下结构定义：

```
struct Vertex
{
    XMFLOAT3 Pos;
    XMFLOAT4 Color;
};
```

在创建顶点缓冲区后， 我们必须把它绑定到设备的输入槽上，只有这样才能将顶点送入管线。这一工作使用如下方法完成：

```
void ID3D11DeviceContext::IASetVertexBuffers(
    UINT StartSlot,
```

```
    UINT NumBuffers,
    ID3D10Buffer *const *ppVertexBuffers,
    const UINT *pStrides,
    const UINT *pOffsets);
```

**1．StartSlot**：顶点缓冲区所要绑定的起始输入槽。一共有 16 个输入槽，索引依次为 0 到 15。

**2．NumBuffers**：顶点缓冲区所要绑定的输入槽的数量，如果起始输入槽为索引 k，我们绑定了 n 个缓冲，那么缓冲将绑定在索引为 $I_k$，$I_{k+1}$……$I_{k+n-1}$ 的输入槽上。

**3．ppVertexBuffers**：指向顶点缓冲区数组的第一个元素的指针。

**4．pStrides**：指向步长数组的第一个元素的指针（该数组的每个元素对应一个顶点缓冲区，也就是，第 i 个步长对应于第 i 个顶点缓冲区）。这个步长是指顶点缓冲区中的元素的字节长度。

**5．pOffsets**：指向偏移数组的第一个元素的指针（该数组的每个元素对应一个顶点缓冲区，也就是，第 i 个偏移量对应于第 i 个顶点缓冲区）。这个偏移量是指从顶点缓冲区的起始位置开始，到输入装配阶段将要开始读取数据的位置之间的字节长度。当希望跳过顶点缓冲区前面的一部分数据时，可以使用该参数。

因为 **IASetVertexBuffers** 方法支持将一个顶点缓冲数组设置到不同的输入槽中，因此这个方法看起来有点复杂。但是，大多数情况下我们只使用一个输入槽。本章最后的练习部分你会遇到使用两个输入插槽的情况。

顶点缓冲区会一直绑定在输入槽上时。如果不改变输入槽的绑定对象，那么当前的顶点缓冲区会一直驻留在那里。所以，当使用多个顶点缓冲区时，你可以按照下面的形式组织代码：

```
ID3D11Buffer* mVB1; // stores vertices of type Vertex1
ID3D11Buffer* mVB2; // stores vertices of type Vertex2
/*...Create the vertex buffers...*/
UINT stride = sizeof(Vertex1);
UINT offset = 0;
md3dImmediateContext->IASetVertexBuffers(0,  1,  &mVB1,  &stride,
&offset);
/* ...draw objects using vertex buffer 1... */
stride = sizeof(Vertex2);
offset = 0;
md3dImmediateContext->IASetVertexBuffers(0,  1,  &mVB2,  &stride,
&offset);
/* ...draw objects using vertex buffer 2... */
```

把顶点缓冲区指定给输入槽并不能实现顶点的绘制；它只是绘制前的准备工作（准备把顶点传送到管线）。顶点的实际绘制工作由 **ID3D11DeviceContext::Draw** 方法完成：

```
void     ID3D11DeviceContext::Draw(UINT     VertexCount,     UINT
StartVertexLocation);
```

这两个参数定义了在顶点缓冲区中所要绘制的顶点的范围，如图 6.2 所示。

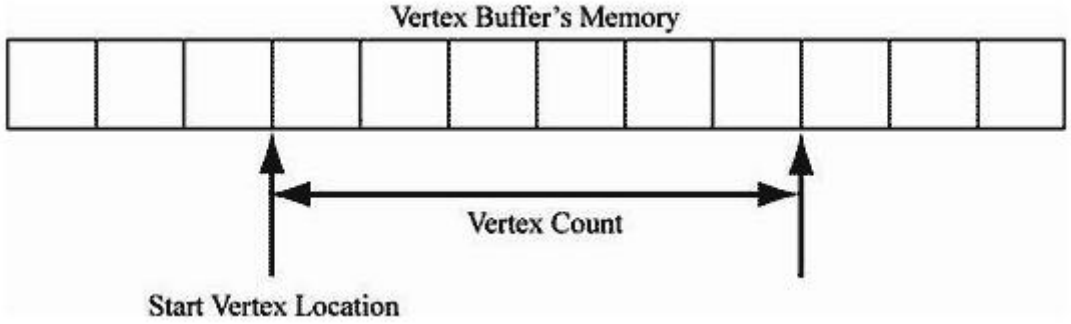



**图 6.2 StartVertexLocation 指定了在顶点缓冲区中所要绘制的第一个顶点的索引（从 0 开始）。VertexCount 指定了所要绘制的顶点的数量。**

# 6.3 索引和索引缓冲

由于索引要由 GPU 访问，所以它们必须放在一个特定的资源容器中，该容器称为索引缓冲（index buffer）。创建索引缓冲的过程与创建顶点缓冲的过程非常相似，只不过索引缓冲存储的是索引而非顶点。所以，这里不再赘述之前讨论过的内容，我们直接给出一个创建索引缓冲区的示例：

```
UINT indices [24] = {
    0, 1, 2,   //  Triangle  0
    0, 2, 3,   //  Triangle  1
    0, 3, 4,   //  Triangle  2
    0, 4, 5,   //  Triangle  3
    0, 5, 6,   //  Triangle  4
    0, 6, 7,   //  Triangle  5
    0, 7, 8,   //  Triangle  6
    0, 8, 1    //  Triangle  7
} ;

// 要创建的索引的描述
D3D11_BUFFER_DESC ibd;
ibd.Usage = D3D11_USAGE_IMMUTABLE;
ibd.ByteWidth = sizeof(UINT) * 24;
ibd.BindFlags = D3D11_BIND_INDEX_BUFFER;
ibd.CPUAccessFlags = 0;
ibd.MiscFlags = 0;
ibd.StructureByteStride = 0;

// 设定用于初始化索引缓冲的数据
D3D11_SUBRESOURCE_DATA iinitData;
iinitData.pSysMem = indices;

// 创建索引缓冲
ID3D11Buffer* mIB;
HR(md3dDevice->CreateBuffer(&ibd, &iinitData, &mIB));
```

与顶点缓冲区相同，所有的 Direct3D 资源在使用之前都必须先绑定到管线上。我们使用 **ID3D11DeviceContext::IASetIndexBuffer** 方法将一个索引缓冲区绑定到输入装配阶段。下面是一个例子：

```
md3dImmediateContext->IASetIndexBuffer(mIB,  DXGI_FORMAT_R32_UINT,
0);
```

第 2 个参数表示索引格式。在本例中，我们使用的是 32 位无符号整数（DWORD）；所以，该参数设为 DXGI_FORMAT_R32_UINT。如果你希望节约一些内存，不需要这大的取值范围，那么可以改用 16 位无符号整数。还要注意的是，在 **IASetIndexBuffer** 方法中指定的格式必须与 **D3D11_BUFFER_DESC::ByteWidth** 数据成员指定的字节长度一致，否则会

出现问题。索引缓冲区只支持 **DXGI_FORMAT_R16_UINT** 和 **DXGI_FORMAT_R32_UINT** 两种格式。第 3 个参数是一个偏移值，它表示从索引缓冲区的起始位置开始、到输入装配时实际读取数据的位置之间的字节长度。如果希望跳过索引缓冲区前面的一部分数据，那么可以使用该参数。

最后，当使用索引时，我们必须用 **DrawIndexed** 方法代替 **Draw** 方法：

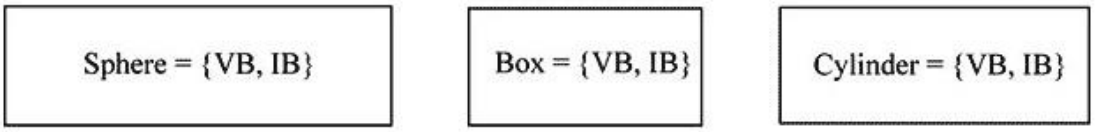
```
void ID3D11DeviceContext::DrawIndexed(
    UINT IndexCount,
    UINT StartIndexLocation,
    INT BaseVertexLocation);
```

**1．IndexCount**：在当前绘图操作中使用的索引的数量。在一次绘图操作中不一定使用索引缓冲区中的全部索引；也就是说，我们可以绘制索引的一个连续子集。

**2．StartIndexLocation**：指定从索引缓冲区的哪个位置开始读取索引数据。

**3．BaseVertexLocation**：在绘图调用中与索引相加的一个整数。

我们通过分析如下情景来解释些参数。假设有三个物体：一个球体、一个立方体和一个圆柱体。每个物体都有它自己的顶点缓冲和索引缓冲。在每个独立的索引缓冲中的索引都与各自独立的顶点缓冲对应。现在，我们把这三个物体的顶点和索引数据合并到一个全局的顶点缓冲和一个全局的索引缓冲中，如图 6.3 所示（假如有许多小顶点缓冲和索引缓冲，并且它们可以很容易的合并，那么合并它们可以带来性能提升）。在合并之后，索引就不再正确了（因为这些索引是与各自独立的顶点缓冲区对应的，它们与全局顶点缓冲区没有对应关系）；所以，这些索引必须被重新计算，使它们与全局顶点缓冲区建立正确的对应关系。



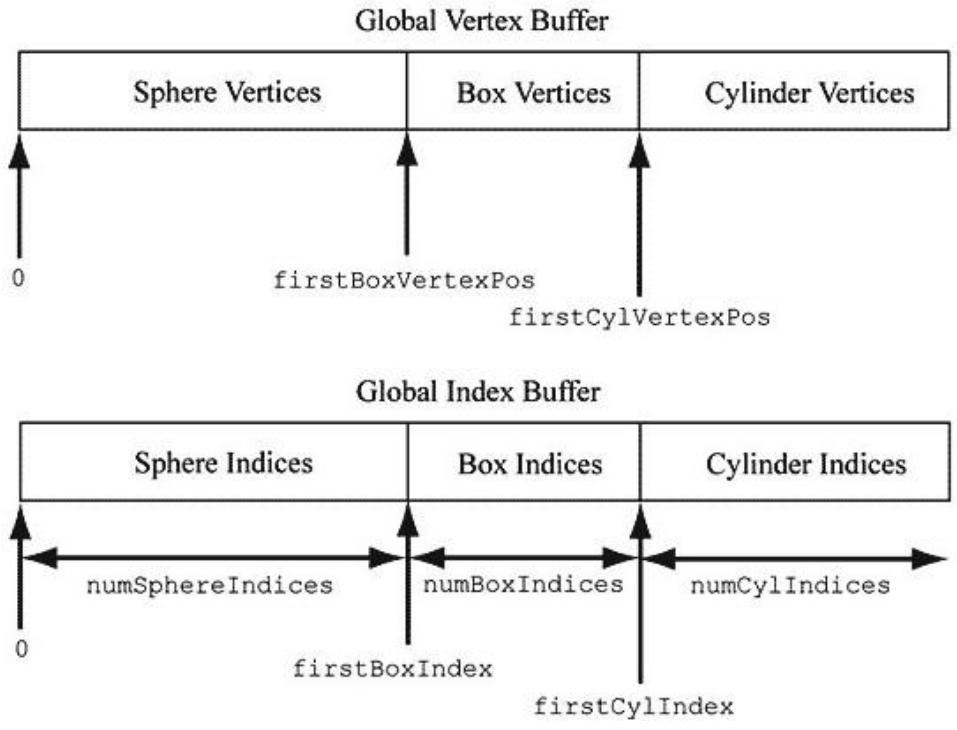



**图 6.3 将多个顶点缓冲合并为一个大的顶点缓冲，将多个索引缓冲合并为一个大的索引缓冲。**

假设原先的立方体索引为 0、1、...、numBoxVertices-1，通过个索引可以遍历立方体的所有顶点。在合并之后，索引应该变为 firstBoxVertexPos、firstBoxVertexPos+1、...、firstBoxVertexPos+ numBoxVertices -1。所以，要更新索引，我们必须将 firstBoxVertexPos 与立方体的每个索引相加。而且，我们必须将 firstCylVertexPos 与圆柱体的每个索引相加。注意，球体的索引不需要修改（因为球体的顶点位置为 0）。通常，只要将一个物体在全局顶点缓冲区中的第一个顶点位置与原索引相加就可以得到该物体的新索引值。所以，只要给出物体在全局顶点缓冲中的第一个顶点的位置（例如，BaseVertexLocation），Direct3D 就可以在绘图调用中为我们重新计算索引。我们可以使用以下 3 条语句依次绘制球体、立方体和圆柱体：

```
md3dImmediateContext->DrawIndexed(numSphereIndices, 0, 0);
md3dImmediateContex->DrawIndexed(numBoxIndices,     firstBoxIndex,
firstBoxVertexPos);
md3dImmediateContex->DrawIndexed(numCylIndices,     firstCylIndex,
firstCylVertexPos);
```

后面的“Shape”示例程序就用到了这个技术。

# 6.4 顶点着色器示例

下面是一个顶点着色器的示例，它的代码非常简单：

```
cbuffer cbPerObject
{
    float4x4 gWVP;
};

void VS(float3 iPosL : POSITION,
    float4 iColor : COLOR,
    out float4 oPosH : SV_POSITION,
    out float4 oColor : COLOR)
{
    // 转换到齐次裁剪空间
    oPosH = mul(float4(iPosL, 1.0f), gWVP);
    // 把顶点颜色直接传到像素着色器
    oColor = iColor;
}
```

着色器使用一种称为高级着色语言（High-Level Shading Language，简称 HLSL）的脚本语言来编写，它的语法与 C++相似，很容易就能学会。附录 B 提供了一些有关 HLSL 的简要概述。在本书中，我们将采用一种基于示例的方式讲解 HLSL 及着色器编程。也就是，根据贯穿本书的每个演示程序所涉及的技术讲解相关的 HLSL 概念。着色器通常保存在一种称为 effect 文件（.fx）的纯文本文件中。我们会在本章随后的小节中讨论 effect 文件，而现在我们主要讨论顶点着色器。

这里，顶点着色器是一个称为 VS 的函数。注意，你可以为顶点着色器指定任何有效的函数名。该顶点着色器包含 4 个参数；前两个是输入参数，后两个是输出参数（由 **out** 关键字表示）。HLSL 没有类似于 C++的引用和指针，所以当一个函数要返回多个值时，我们必须使用结构体或输出参数。

前两个输入参数对应于我们在顶点结构体中定义的数据成员。参数语义“**:POSITION**”和“**:COLOR**”用于将顶点结构体的数据成员映射为顶点着色器的输入参数，如图 6.4 所示。

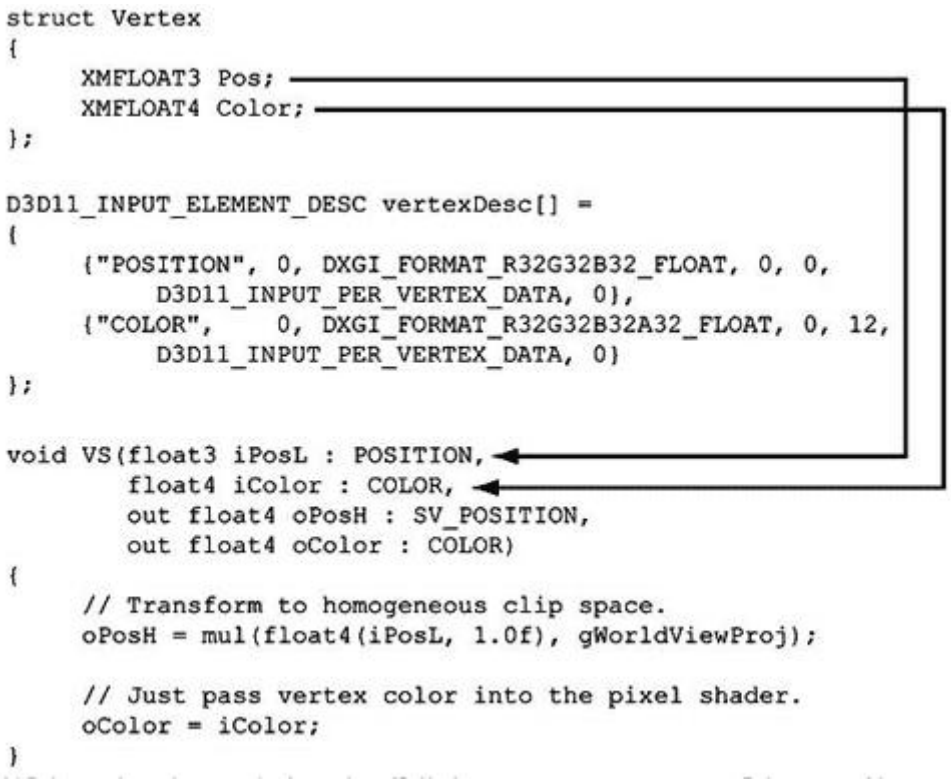



**图 6.4 D3D11_INPUT_ELEMENT_DESC 数组为每个顶点元素指定了一个相关的语义，而顶点着色器的每个参数也都带有一个附加语义。语义用于建立顶点元素和顶点着色器参数之间的对应关系。**

输出参数也带有附加语义（“**:SV_POSITION**”和“**:COLOR**”）。这些语义用于将顶点着色器的输出数据映射为下一阶段（几何着色器或像素着色器）的输入数据。注意，**SV_POSITION** 是一个特殊的语义（SV 表示系统值，即 system value 的缩写）。它用于告诉顶点着色器该元素存储的是顶点位置。顶点位置的处理方式与其他顶点属性不同，因为它涉及到一些其他属性所没有的特殊运算，比如裁剪。若不是系统值，那么输出参数的语义名称可以是任何有效的语义名称。

该顶点着色器的代码非常简单。第一行通过与一个 4×4 矩阵 **gWorldViewProj** 相乘，将顶点位置从局部空间变换到齐次裁剪空间，矩阵 **gWorldViewProj** 是世界矩阵、观察矩阵和投影矩阵的组合矩阵：

```
// 转换到齐次裁剪空间
oPosH = mul(float4(iPosL, 1.0f), gWorldViewProj);
```

构造函数语法“float4(iPosL, 1.0f)”用于创建 4D 向量，它相当于“float4(iPosL.x, iPosL.y, iPosL.z, 1.0f)”；我们知道，顶点位置是一个点而不是一个向量，所以第 4 个分量应设为 1（即 w=1）。**float2** 和 **float3** 分别表示 2D 和 3D 向量。矩阵变量 **gWorldViewProj** 定义在一个常量缓冲区中，我们会在下一节讨论对它进行讨论。内置函数 **mul** 用于实现向量-矩阵乘法，它为不同维数的矩阵乘法定义了多个重载版本；例如，该函数可以实现 4×4 矩阵乘法、3×3 矩阵乘法、或者 1×3 向量与 3×3 矩阵的向量-矩阵乘法。最后一行是将输入的颜色赋值给输出参数，把颜色传递给管线的下一阶段：

```
oColor = iColor;
```

我们可以使用结构体来重写上面的顶点着色器，实现相同的功能：

```
cbuffer cbPerObject
{
```

```
    float4x4 gWVP;
};
struct VS_IN
{
    float3 posL    : POSITION;
    float4 color : COLOR;
};
struct VS_OUT
{
    float4 posH : SV_POSITION;
    float4 color : COLOR;
};
VS_OUT VS(VS_IN input)
{
    VS_OUT output;
    output.posH = mul(float4(input.posL, 1.0f), gWVP);
    output.color = input.color;
    return output;
}
```

**注意**：当没有几何着色器时，顶点着色器至少要实现投影变换，因为当顶点离开顶点着色器时（在没有几何着色器的情况下），硬件假定顶点位于投影空间。当包含一个几何着色器时，投影工作可以转嫁到几何着色器中完成。

**注意**：顶点着色器（或几何着色器）不执行透视除法；它只完成投影矩阵部分。透视除法会随后由硬件完成。

# 6.5 常量缓冲

在上一节的顶点着色器示例中包含如下代码：

```
cbuffer cbPerObject
{
    float4x4 gWorldViewProj;
};
```

这段代码定义了一个称为 cbPerObject 的 **cbuffer** 对象（constant buffer，常量缓冲）。常量缓冲只是一个用于存储各种变量的数据块，这些变量可以由着色器来访问。在本例中，常量缓冲区只存储了一个称为 **gWorldViewProj** 的 4×4 矩阵，它是世界矩阵、观察矩阵和投影矩阵的组合矩阵，用于将顶点从局部空间变换到齐次裁剪空间。在 HLSL 中，4×4 矩阵由内置的 **float4x4** 类型表示；与之类似，要定义一个 3×4 矩阵和一个 2×2 矩阵，可以分别使用 **float3x4** 和 **float2x2** 类型。顶点着色器不能修改常量缓冲中的数据，但是通过 effect 框架（6.9 节），C++应用程序代码可以在运行时修改常量缓冲中的内容。它为 C++应用程序代码和 effect 代码提供了一种有效的通信方式。例如，因为每个物体的世界矩阵各不相同，所以每个物体的“WVP”组合矩阵也各不相同；所以，当使用上述顶点着色器绘制多个物体时，我们必须在绘制每个物体前修改 **gWorldViewProj** 变量。

通常的建议是根据变量修改的频繁程度创建不同的常量缓冲。比如，你可以创建下面的常量缓冲：

```
cbuffer cbPerObject
{
    float4x4 gWVP;
};
cbuffer cbPerFrame
{
    float3 gLightDirection;
    float3 gLightPosition;
    float4 gLightColor;
};
cbuffer cbRarely
{
    float4 gFogColor;
    float gFogStart;
    float gFogEnd;
};
```

在本例中，我们使用了 3 个常量缓冲区。第 1 个常量缓冲区存储“WVP”组合矩阵。该变量随物体而定，所以它必须在物体级别上更新。也就是，当我们每帧渲染 100 个物体时，每帧都要对这个变量更新 100 次。第 2 个常量缓冲存储了场景中的灯光变量。这里，我们假设要生成灯光动画，所以些变量必须在每帧中更新一次。最后一个常量缓冲存储了用于控制雾效的变量。这里，我们假设场景的雾效变化频率很低（例如，在游戏的一个特定时段中变化一次）。

对常量缓冲进行分组是为了提高运行效率。当一个常量缓冲区被更新时，它里面的所有

变量都会同时更新；所以，根据它们的更新频率进行分组，可以减少不必要的更新操作，提高运行效率。

# 6.6 像素着色器示例

5.10.3 节说过，由顶点着色器（或几何着色器）输出的顶点属性都已经过了插值处理。这些插值随后会作为像素着色器（pixel shader）的输入数据传入像素着色器。假设这里没有几何着色器，图 6.5 说明了目前顶点数据的流动过程。

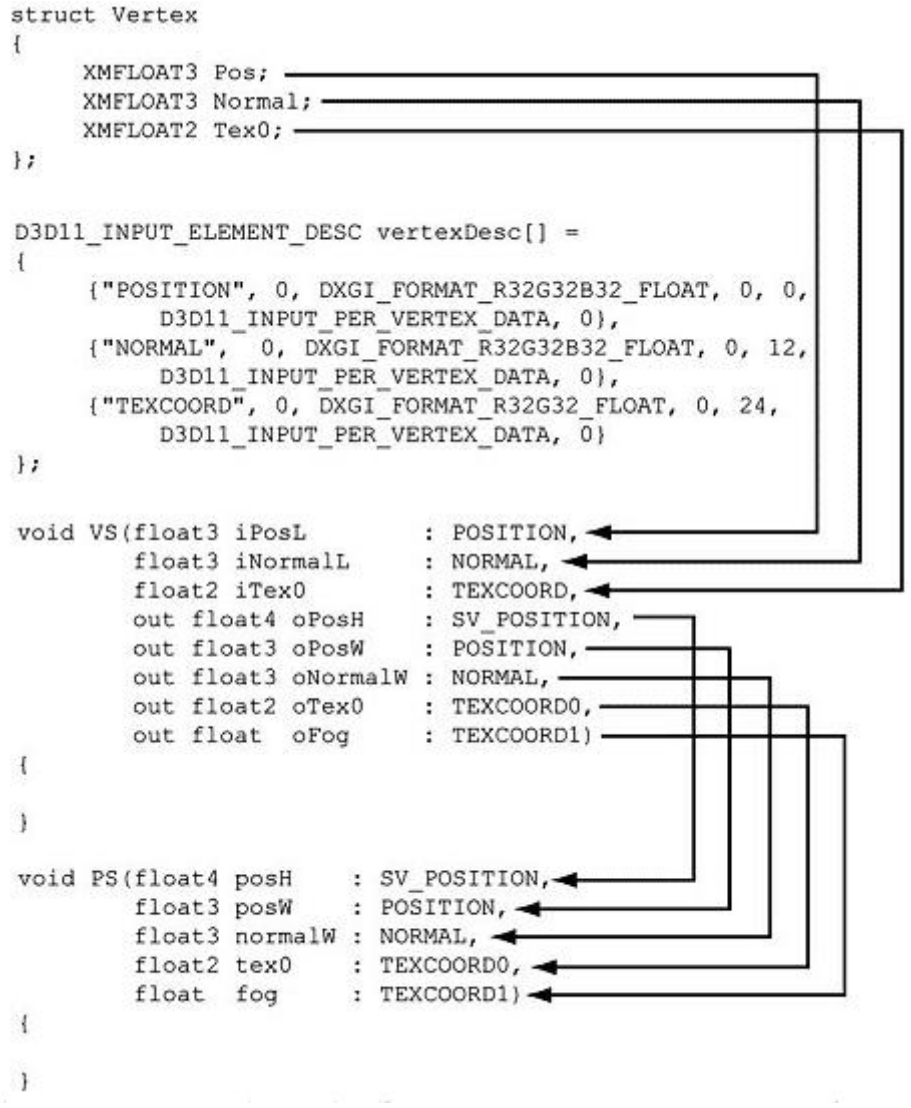



**图 6.5 D3D11_INPUT_ELEMENT_DESC 数组为每个顶点元素指定了一个关联语义，而顶点着色器的每个参数都有一个附加语义。这些语义描述了顶点元素和顶点着色器参数之间的对应关系。同样，顶点着色器的每个输出参数和像素着色器的每个输入参数也都有一个附加语义。这些语义用于将顶点着色器的输出参数映射为像素着色器的输入参数。**

与顶点着色器相似，像素着色器也是一个函数，只不过它处理的数据是像素片段（pixel fragment）。像素着色器的任务是为每个像素片段计算一个颜色值。请注意，像素和像素片段的含义不同，像素片段可能不会被存入后台缓冲区；例如，像素着色器可以对像素片段进行裁剪（在 HLSL 中的一个 **clip** 函数，它可以终止对一个像素片段的处理工作），或者当一个像素片段没有通过深度测试或模板测试时，它会被丢弃。所以，后台缓冲区中的一个像素可能会对应多个候选像素片段；这就是我们要区分“像素片段”和“像素”这两个术语的原因，虽然有时这两个术语会交替使用，但是读者应该明白它们在特定环境下的特定含义。

下面是一个简单的像素着色器，它与 6.4 节给出的顶点着色器对应。出于完整性的考虑，我们将该顶点着色器再次列了出来。

```
cbuffer cbPerObject
{
    float4x4 gWorldViewProj;
};
void VS(float3 iPosL : POSITION, float4 iColor : COLOR,
    out float4 oPosH    : SV_POSITION,
    out float4 oColor : COLOR)
{
    // 转换到齐次剪裁空间
    oPosH = mul(float4(iPosL, 1.0f), gWVP);
    // 将顶点颜色直接传递到像素着色器
    oColor = iColor;
}

float4 PS(float4 posH: SV_POSITION,float4 color : COLOR) : SV_TARGET
{
    return color;
}
```

在本例中，像素着色器只是简单地返回插值颜色。注意，像素着色器的输入参数和顶点着色器的输出参数是对应的；这是一项规定。该像素着色器返回了一个 4D 颜色值。函数参数列表中的 **SV_TARGET** 语义表示返回值与渲染目标视图的格式一致。

我们也可以使用输入/输出结构体的形式重写上面的顶点着色器和像素着色器代码。我们将语义附加到输入/输出结构的成员上，使用 return 语句用于输出代替输出参数。

```
cbuffer cbPerObject
{
    float4x4 gWorldViewProj;
};

struct VertexIn
{
    float3 PosL  : POSITION;
    float4 Color : COLOR;
};

struct VertexOut
{
    float4 PosH  : SV_POSITION;
    float4 Color : COLOR;
};

VertexOut VS(VertexIn vin)
{
```

```
    VertexOut vout;

    // 转换到齐次剪裁空间
    vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj);

    // 将顶点颜色直接传递到像素着色器
    vout.Color = vin.Color;

    return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
    return pin.Color;
}
```

# 6.7 渲染状态

从本质上讲，Direct3D 是一个状态机（state machine）。在我们改变它的状态之前，驻留在状态机内的当前状态是不会改变的。例如，我们在 6.1 节、6.2 节和 6.3 节中看到，当顶点缓冲和索引缓冲绑定到管线的输入装配阶段时，如果我们不绑定其他缓冲，那么它们就会一直驻留在那里；同样，在没有改变图元拓扑之前，当前的图元拓扑设置会一直有效。另外，Direct3D 将配置信息封装在状态组中，我们可以使用如下 3 种状态组配置 Direct3D：

**1．ID3D11RasterizerState**：该接口表示用于配置管线光栅化阶段的状态组。

**2．ID3D11BlendState**：该接口表示用于配置混合操作的状态组。我们将在有关混合的章节讨论这些状态；默认情况下，混合处于禁用状态，所以我们可以先不考虑这方面的问题。

**3．ID3D11DepthStencilState**：该接口表示用于配置深度测试和模板测试的状态组。我们将在有关模板缓冲的章节讨论这些状态；默认情况下，模板是禁用的，所以我们可以先不考虑这方面的问题。而默认的深度测试是我们在 4.1.5 节描述的标准深度测试。

目前，我们唯一需要关心的状态块接口是 **ID3D11RasterizerState**。我们可以通过填充一个 **D3D11_RASTERIZER_DESC** 结构体并调用如下方法来创建 **ID3D11RasterizerState** 对象：

```
HRESULT ID3D11Device::CreateRasterizerState(
    const D3D11_RASTERIZER_DESC *pRasterizerDesc,
    ID3D11RasterizerState **ppRasterizerState);
```

第 1 个参数是一个指向 **D3D11_RASTERIZER_DESC** 结构体的指针，该结构体用于描述所要创建的光栅化状态块；第二个参数用于返回创建后的 **ID3D11RasterizerState** 对象。

**D3D11_RASTERIZER_DESC** 结构体的定义如下：

```
typedef struct D3D11_RASTERIZER_DESC{
    D3D11_FILL_MODE FillMode;     // Default:D3D11_FILL_SOLID
    D3D11_CULL_MODE CullMode;     // Default:D3D11_CULL_BACK
    BOOL FrontCounterClockwise;  // Default:false
    INT DepthBias;   // Default:0
    FLOAT DepthBiasClamp;   // Default:0.0f
    FLOAT SlopeScaledDepthBias;   // Default:0.0f
    BOOL DepthClipEnable;   // Default:true
    BOOL ScissorEnable;   // Default:false
    BOOL MultisampleEnable;  // Default:false
    BOOL AntialiasedLineEnable;  // Default:false
} D3D11_RASTERIZER_DESC;
```

这里面的大部分成员是高级选项或者不常用的选项；因此，我们在这里只讲解前 3 个成员的含义，其他成员的详情请参见 SDK 文档。

**1．FillMode**：当指定为 **D3D11_FILL_WIREFRAME** 时，表示以线框模式渲染几何体；当指定为 **D3D11_FILL_SOLID** 时，表示以实心模式渲染几何体，这是默认值。

**2．CullMode**：当指定为 **D3D11_CULL_NONE** 时，表示禁用背面消隐功能；当指定为 D3D11_CULL_FRONT 时，表示消隐朝前的三角形；当指定为 **D3D11_CULL_BACK** 时，表示消隐朝后的三角形，这是默认值。

**3．FrontCounterClockwise**：当设为 **false** 时，表示按顺时针方向环绕的三角形（相对

于观察者）是朝前的，而按逆时针方向环绕的三角形（相对于观察者）是朝后的，这是默认值。当设为 **true** 时，表示按逆时针方向环绕的三角形（相对于观察者）是朝前的，而按顺时针方向环绕的三角形（相对于观察者）是朝后的。

在创建 **ID3D11RasterizerState** 对象之后，我们可以使用个新的状态块来更新设备：

```
void         ID3D11DeviceContext::RSSetState(ID3D11RasterizerState
*pRasterizerState);
```

下面的代码示范了如何通过创建一个光栅化状态块来禁用背面消隐：

```
D3D11_RASTERIZER_DESC rsDesc;
ZeroMemory(&rsDesc, sizeof(D3D11_RASTERIZER_DESC));
rsDesc.FillMode = D3D11_FILL_SOLID;
rsDesc.CullMode = D3D11_CULL_NONE;
rsDesc.FrontCounterClockwise = false;
rsDesc.DepthClipEnable = true;

HR(md3dDevice->CreateRasterizerState(&rsDesc,&mNoCullRS));
```

**注意**：因为没有设置的属性的默认值是 0 或 false，所以使用 ZeroMemory 可以正常初始化这些属性。但是，若有些属性默认值不是 0 或是 true，那么你就必须显式地设置这些值。

注意，对于一个应用程序来说，你可能会用到多个不同的 **ID3D11RasterizerState** 对象。所以，你应该在初始化时把它们都创建出来，然后在应用程序的更新/绘图代码中切换些状态。例如，场景中有两个物体，你希望先以线框模式绘制第一个物体，然后再以实心模式绘制第二个物体。那么，你就应该创建两个 **ID3D11RasterizerState** 对象，当绘制物体时，切换这两种不同的状态：

```
// Create render state objects at initialization time.
ID3D11RasterizerState* mWireframeRS;
ID3D11RasterizerState* mSolidRS;
...
// Switch between the render state objects inthe draw function.
md3dDeviceContext->RSSetState(mSolidRS);
DrawObject();
md3dDeviceContext->RSSetState(mWireframeRS);
DrawObject();
```

注意，Direct3D 不会从一种状态自动恢复到先前状态。所以，当绘制物体时，你应该根据需要手工指定状态对象。错误地假设设备的当前状态必然会导致错误的渲染结果。

每个状态块都有一个默认状态。我们可以通过在调用 **RSSetState** 方法时指定空值来恢复默认状态：

```
md3dDeviceContext->RSSetState( 0 );
```

**注意**：应用程序无需在运行时创建额外的渲染状态组。所以，应该在初始化时就定义并创建所有需要用到的状态组。而且，因为无需在运行时修改状态组，你可以在渲染代码中对这些状态组提供全局只读访问。例如，你可以将所有状态组对象放置在一个静态类中，通过这个方法，你就无需创建重复的状态组，渲染代码的不同部分都能共享这个渲染状态组对象。

# 6.8 Effects

effect 框架是一组用于管理着色器程序和渲染状态的工具代码。例如，你可能会使用不同的 effect 绘制水、云、金属物体和动画角色。每个 effect 至少要由一个顶点着色器、一个像素着色器和渲染状态组成。

在 Direct3D 11 中，effects 框架已从 D3DX 库中移除，你必须包含一个单独的头文件（**d3dx11Effect.h**），链接一个单独的库文件（**D3DX11Effects.lib** 用于 release 生成，而 **D3DX11EffectsD.lib** 用于 debug 生成）。

而且，在 Direct3D 11 中提供了 effect 库的完整源代码（DirectX SDK\Samples\C++\Effects11）。因此，你可以根据需要修改 effect 框架。本书中，我们只是使用、并不会修改 effect 框架。要使用这个库，首先需要生成 Effects11 项目的 Release 和 Debug 模式，用于获得 **D3DX11Effects.lib** 和 **D3DX11EffectsD.lib** 文件，除非 effect 框架进行了更新（例如，新版本的 DirectX SDK 可能会更新这些文件，这时就需要重新生成.lib 文件），这个步骤只需进行一次。**d3dx11Effect.h** 头文件可在 DirectX SDK\Samples\C++\Effects11\Inc 文件夹中找到。在示例代码中，我们将 **d3dx11Effect.h**，**D3DX11EffectsD.lib** 和 **D3DX11Effects.lib** 文件都放在 Common 文件夹中，这样所有的项目文件都能共享这些文件。

## 6.8.1 Effect 文件

我们已经讨论了顶点着色器、像素着色器，并对几何着色器、曲面细分着色器进行了简要概述。我们还讨论了常量缓冲，它可以用于存储由着色器访问的“全局”变量。这些代码通常保存在一个 effect 文件(.fx)中，它是一个纯文本文件中(就像是 C++代码保存在.h 和.cpp 文件中一样)。除了着色器和常量缓冲之外，每个 effect 文件至少还要包含一个 technique，而每个 technique 至少要包含一个 pass。

**1．technique11**：一个 technique 由一个或多个 pass 组成，用于创建一个渲染技术。每个 pass 实现一种不同的几何体渲染方式，按照某些方式将多个 pass 的渲染结果混合在一起就可以得到我们最终想要的渲染结果。例如，在地形渲染中我们将使用多通道纹理映射技术（multi-pass texturing technique）。注意，多通道技术通常会占用大量的系统资源，因为每个 pass 都要对几何体进行一次渲染；不过，要实现某些渲染效果，我们必须使用多通道技术。

**2．pass**：一个 pass 由一个顶点着色器、一个可选的几何着色器、一个像素着色器和一些渲染状态组成。这些部分定义了 pass 的几何体渲染方式。像素着色器也是可选的（很罕见）。例如，若我们只想绘制深度缓冲，不想绘制后台缓冲，在这种情况下我们就不需要像素着色器计算像素的颜色。

**注意**：techniques 也可以组合在一起成为 effect 组。如果你没有显式地定义一个 effect 组，那么编译器会创建一个匿名 effect 组，把所有 technique 包含在 effect 文件中。本书中，我们不显式地定义 effect 组。

下面是本章演示程序使用的 effect 文件：

```
cbuffer cbPerObject
{
    float4x4 gWorldViewProj;
};
```

```
struct VertexIn
{
    float3 PosL  : POSITION;
    float4 Color : COLOR;
};

struct VertexOut
{
    float4 PosH  : SV_POSITION;
    float4 Color : COLOR;
};

VertexOut VS(VertexIn vin)
{
    VertexOut vout;

    // 转换到齐次剪裁空间
    vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj);

    // 将顶点颜色直接传递到像素着色器
    vout.Color = vin.Color;

    return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
    return pin.Color;
}

technique11 ColorTech
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_5_0, VS() ) );
        SetPixelShader( CompileShader( ps_5_0, PS() ) );
    }
}
```

**注意**：点和向量可以在许多不同的空间中描述（例如，局部空间、世界空间、观察空间、齐次裁剪空间）。当阅读代码时，有时很难看出点和向量的坐标系是相对于哪个坐标系的。所以，我们经常使用下面的后缀来表示空间：L（局部空间）、W（世界空间）、V（观察空间）、H（齐次裁剪空间）。下面是一些例子：

```
float3 iPosL;     // local space
float3 gEyePosW;    // world space
```

```
float3 normalV;    // view space
float4 posH;       // homogeneous clip space
```

前面提到，pass 可以包含渲染状态。也就是，状态块可以直接在 effect 文件中创建和指定。当 effect 需要特定的渲染状态时，这种方式非常实用；但是，当某些 effect 需要在运行过程中改变渲染状态时，我们更倾向于在应用程序层执行状态设定，因为这样进行状态切换更方便一些。下面的代码示范了如何在一个 effect 文件中创建和指定光栅化状态块。

```
RasterizerState Wireframe
{
    FillMode = Wireframe;
    CullMode = Back;
    FrontCounterClockwise = false;
    // 我们没有设置的属性使用默认值
};

technique11 ColorTech
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_5_0, VS() ) );
        SetPixelShader( CompileShader( ps_5_0, PS() ) );
        SetRasterizerState(Wireframe);
    }
}
```

可以看到，在光栅化状态对象中定义的常量与 C++中的枚举成员基本相同，只是省去了前缀而已（例如，**D3D11_FILL_**和 **D3D11_CULL_**）。

注意：由于 effect 通常保存在扩展名为.fx 的文件中，所以在修改 effect 代码之后，不必重新编译 C++源代码。

## 6.8.2 编译着色器

创建一个 effect 的第一步是编译定义在.fx 文件中的着色器程序，可以由下面的 D3DX 方法完成：

```
HRESULT D3DX11CompileFromFile (
    LPCTSTR pSrcFile ,
    CONST D3D10_SHADE R_MACRO *pDefines,
    LPD3D10INCLUDE  pInclude ,
    LPCSTR  pFunctionName ,
    LPCSTR pProfile,
    UINT Flags 1,
    UINT Flags 2,
    ID3DX11ThreadPump *pPump ,
    ID3D10Blob  **ppShader,
    ID3D10Blob  **ppErrorMsgs,
```

```
    HRESULT *pHResult);
```

**1．pSrcFile**：.fx 文件名，该文件包含了我们所要编译的效果源代码。

**2．pDefines**：高级选项，我们不使用；请参阅 SDK 文档。

**3．pInclude**：高级选项，我们不使用；请参阅 SDK 文档。

**4．pFunctionName**：着色器入口函数的名字。只用于单独编译着色器程序的情况。当使用 effect 框架时设置为 null，这是因为在 effect 文件中已经定义了入口点。

**5．pProfile**：用于指定着色器版本的字符串。对于 Direct3D 11 来说，我们使用的着色器版本为 5.0（“fx_5_0”）。

**6．Flags1**：用于指定着色器代码编译方式的标志值。SDK 文档列出了很多标志值，但本书只使用其中的 2 个：

- **D3D10_SHADER_DEBUG**：以调试模式编译着色器。
- **D3D10_SHADER_SKIP_OPTIMIZATION**：告诉编译器不做优化处理（用于进行调试）。

**7．Flags2**：高级选项，我们不使用；请参阅 SDK 文档。

**8．pPump**：指向线程泵的指针，多线程编程时使用，是高级选项，我们不使用；请参阅 SDK 文档。本书中这个值都设为 null。

**9．ppShader**：返回一个指向 **ID3D10Blob** 数据对象的指针，这个数据对象保存了经过编译的代码。

**10．ppErrorMsgs**：返回一个指向 **ID3D10Blob** 数据对象的指针，这个数据对象存储了一个包含错误信息的字符串。

**11．pHResult**：在使用异步编译时，用于获得返回的错误代码。仅当使用 pPump 时才使用该参数；我们在本书中将该参数设为空值。

**注意**：1．除了可以编译在.fx 文件内的着色器代码，这个方法也可以编译单独的着色器代码。有些程序不使用 effect 框架，它们会单独的定义和编译自己的着色器代码。

2. 方法中的指向“D3D10”的引用并不是打印错误。因为 D3D11 编译器是建立在 D3D10 的编译器之上的，所以 Direct3D 11 开发组就没有修改某些标识的名称。

3．**ID3D10Blob** 只是一个通用内存块，它有两个方法：

（a）LPVOID GetBufferPointer：返回指向数据的一个 void*，所以在使用时应该对它执行相应的类型转换（具有请参见下面的示例）。

（b）SIZE_T GetBufferSize：返回缓冲的大小，以字节为单位。

编译完成后，我们就可以使用下面的方法创建一个 effect（用 **ID3DXEffect11** 接口表示）：

```
HRESULT D3DX11CreateEffectFromMemory(
    void *pData,
    SIZE_T DataLength,
    UINT FXFlags ,
    ID3D11Device *pDevice,
    ID3DX11Effect **ppEffect);
```

**1．pData**：指向编译好的 effect 数据的指针。

**2．DataLength**：effect 数据的长度，以字节为单位。

**3．FXFlags**：Effect 标识必须与定义在 **D3DX11CompileFromFile** 方法中的 **Flags2** 匹配。

**4．pDevice**：指向 Direct3D 11 设备的指针。

**5．ppEffect**：指向创建好的 effect 的指针。

下面的代码演示了如何编译并创建一个 effect：

```
DWORD shaderFlags = 0;
#ifdefined (DEBUG)||defined(_DEBUG)
    shaderFlags |= D3D10_SHADER_DEBUG;
    shaderFlags |= D3D10_SHADER_SKIP_OPTIMIZATION ;
#endif
ID3D10Blob * compiledShader = 0;
ID3D10Blob * compilationMsgs = 0;
HRESULT hr = D3DX11CompileFromFile(L"color.fx", 0,
    0, 0, "fx_5_0", shaderFlags,
    0, 0, &compiledShader, &compilationMsgs, 0);

//  compilationMsgs 包含错误或警告的信息
if(compilationMsgs ! =  0)
{
    MessageBoxA(0,  (char*)compilationMsgs->GetBufferPointer(),  0,
0);
    ReleaseCOM(compilationMsgs);
}

// 就算没有 compilationMsgs，也需要确保没有其他错误
if(FAILED(hr))
{
    DXTrace(__FILE__,(DWORD)__LINE__,hr,L"D3DX11Compile
FromFile",true);
}

ID3DX11Effect*  mFX;
HR(D3DX11CreateEffectFromMemory(
    compiledShader->Ge tBufferPointer(),
    compiledShader->Ge tBufferSize(),
    0, md3dDevice, &mFX));

// 编译完成释放资源
ReleaseCOM(compiledShader);
```

**注意**：创建 Direct3D 资源代价昂贵，尽量在初始化阶段完成，即创建输入布局、缓冲、渲染状态对象和 effect 应该总在初始化阶段完成。

## 6.8.3 在 C++应用程序中与 Effect 进行交互

C++应用程序代码通常要与 effect 进行交互；尤其是 C++应用程序经常要更新常量缓冲中的变量。例如，在一个 effect 文件中，我们有如下常量缓冲定义：

```
cbuffer cbPerObject
{
```

```
    float4x4 gWVP;
    float4 gColor;
    float gSize;
    int gIndex;
    bool gOptionOn;
};
```

通过 **ID3D11Effect** 接口，我们可以获得指向常量缓冲变量的指针：

```
ID3D11EffectMatrixVariable* fxWVPVar;
ID3D11EffectVectorVariable* fxColorVar;
ID3D11EffectScalarVariable* fxSizeVar;
ID3D11EffectScalarVariable* fxIndexVar;
ID3D11EffectScalarVariable* fxOptionOnVar;
fxWVPVar     = mFX->GetVariableByName("gWVP")->AsMatrix();
fxColorVar    = mFX->GetVariableByName("gColor")->AsVector();
fxSizeVar     = mFX->GetVariableByName("gSize")->AsScalar();
fxIndexVar    = mFX->GetVariableByName("gIndex")->AsScalar();
fxOptionOnVar = mFX->GetVariableByName("gOptionOn")->AsScalar();
```

**ID3D11Effect::GetVariableByName** 方法返回一个 **ID3D11EffectVariable** 指针。它是一种通用 effect 变量类型；要获得指向特定类型变量的指针（例如，矩阵、向量、标量），你必须使用相应的 As-方法（例如，**AsMatrix**、**AsVector**、**AsScalar**）。

一旦我们获得变量指针，我们就可以通过 C++接口来更新它们了。下面是一些例子：

```
fxWVPVar->SetMatrix( (float*)&M ); // assume M is of type XMMATRIX
fxColorVar->SetFloatVector( (float*)&v ); // assume v is of type
XMVECTOR
fxSizeVar->>SetFloat( 5.0f );
fxIndexVar->SetInt( 77 );
fxOptionOnVar->SetBool( true );
```

注意，这些语句修改的只是 effect 对象在系统内存中的一个副本，它并没有传送到 GPU 内存中。所以在执行绘图操作时，我们必须使用 **Apply** 方法更新 GPU 内存（参见 6.8.4 节）。这样做的原因是为了提高效率，避免频繁地更新 GPU 内存。如果每修改一个变量就要更新一次 GPU 内存，那么效率会很低。

注意：effect 变量不一定要被类型化。例如，可以有如下代码：

```
ID3D11EffectVariable* mfxEyePosVar;
mfxEyePosVar = mFX->GetVariableByName("gEyePosW");
...
mfxEyePosVar->SetRawValue(&mEyePos, 0, sizeof(XMFLOAT3));
```

这种方式可以用来设置任意大小的变量（例如，普通结构体）。注意，**ID3D11EffectVectorVariable** 接口使用 4D 向量。如果你希望使用 3D 向量的话，那应该像上面那样使用 **ID3D11EffectVariable** 接口。

除了常量缓冲变量之外，我们还需要获得指向 technique 对象的指针。实现方法如下：

```
ID3D11EffectTechnique* mTech;
mTech = mFX->GetTechniqueByName("ColorTech");
```

该方法只包含一个用于指定 technique 名称的字符串参数。

## 6.8.4 使用 effect 绘图

要使用 technique 来绘制几何体，我们只需要确保对常量缓冲中的变量进行实时更新。然后，使用循环语句来遍历 technique 中的每个 pass，使用 pass 来绘制几何体：

```
// 设置常量缓冲
XMMATRIX world  = XMLoadFloat4x4(&mWorld);
XMMATRIX view = XMLoadFloat4x4(&mView);
XMMATRIX proj = XMLoadFloat4x4(&mProj);
XMMATRIX worldViewProj = world*view*proj;

mfxWorldViewProj->SetMatrix(reinterpret_cast<float*>(&worldViewPr
oj));

D3DX11_TECHNIQUE_DESC techDesc;
mTech->GetDesc(&techDesc);
for(UINT p = 0;p < techDesc.Passes;++p )
{
    mTech->GetPassByIndex(p)->Apply(0,md3dImmediateContext);
    //  绘制几何体
    md3dImmediateContext->DrawIndexed(36, 0, 0);
}
```

当使用 pass 来绘制几何体时，Direct3D 会启用在 pass 中指定的着色器和渲染状态。**ID3D11EffectTechnique::GetPassByIndex** 方法返回一个指定索引的 pass 对象的 **ID3D11EffectPass** 接口指针。**Apply** 方法更新存储在 GPU 内存中的常量缓冲、将着色器程序绑定到管线、并启用在 pass 中指定的各种渲染状态。在当前版本的 Direct3D 11 中，**ID3D11EffectPass::Apply** 方法的第一个参数还未使用，应设置为 0；第二个参数指向 pass 使用的设备上下文的指针。

如果你需要在绘图调用之间改变常量缓冲中的变量值，那你必须在绘制几何体之前调用 **Apply** 方法：

```
for(UINT i = 0; i < techDesc.Passes; ++i)
{
    ID3D11EffectPass* pass = mTech->GetPassByIndex(i);

    //设置地面几何体的 WVP 组合矩阵
    worldViewProj = mLandWorld*mView*mProj;
    mfxWorldViewProj->SetMatrix(reinterpret_cast<float*>(&
worldViewProj);
    pass->Apply(0, md3dImmediateContext);
    mLand.draw();

    // 设置水波几何体的 WVP 组合矩阵
    worldViewProj = mWavesWorld*mView*mProj;
    mfxWorldViewProj->SetMatrix(reinterpret_cast<float*>(&
```

```
worldViewProj);
    pass->Apply(0 ,md3dImmediateContext);
    mWaves.draw();
}
```

## 6.8.5 在生成期间编译 effect

我们已经介绍了如何在运行时通过 **D3DX11CompileFromFile** 方法编译一个 effect。但这样做会带来一个小小的不便：如果你的 effect 文件有一个编译错误，直到程序运行时你才会发现这个错误。我们还可以使用 DirectX SDK 自带的 *fxc* 工具（位于 DirectX SDK\Utilities\bin\x86）离线编译你的 effect。而且，你还可以修改你的 VC++项目，将调用 fxc 编译 effect 的过程作为生成过程的一部分。步骤如下：

**1**．确保路径 DirectX SDK\Utilities\bin\x86 位于你的项目的 VC++目录的“可执行文件目录（Executable Directories）”之下。

**2**．在项目中添加 effect 文件。

**3**．在解决方案资源管理器中右击每个 effect 文件选择**属性**，添加自定义生成工具（见图 6.6）：

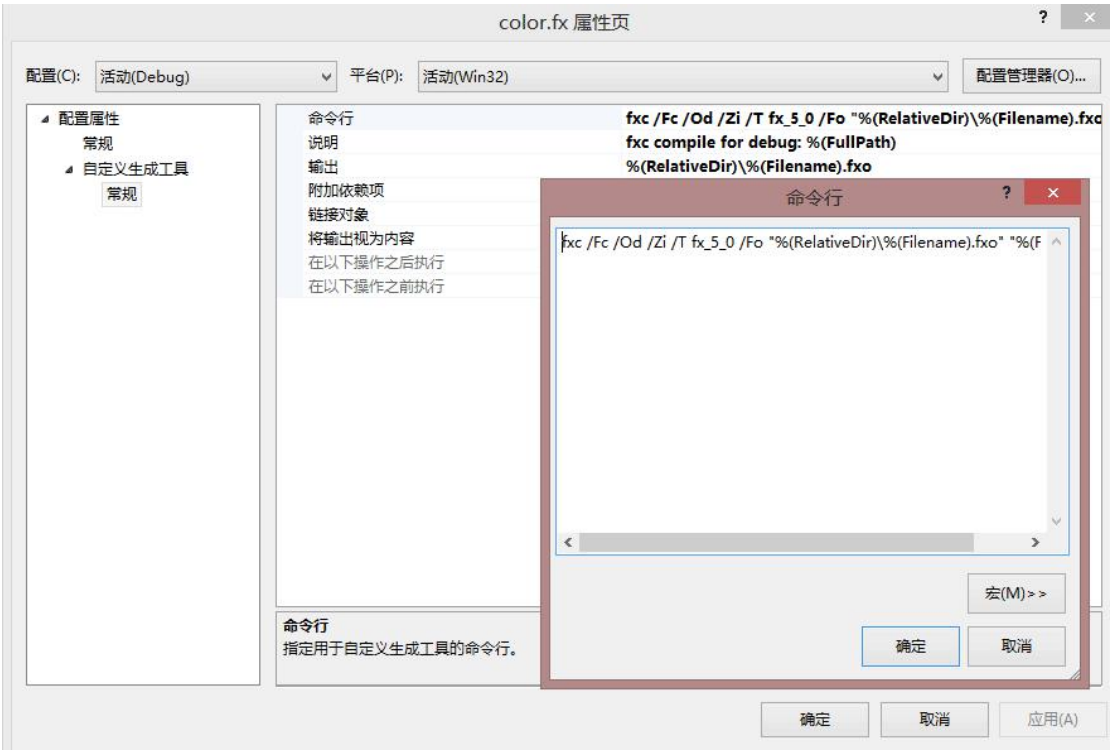



**图 6.6 在项目中添加自定义生成工具**

调试模式：

**fxc /Fc /Od /Zi /T fx_5_0 /Fo " % (RelativeDir)\% (Filename).fxo " " % (FuIIPath) "**

发布模式：

**fxc /T fx_5_0 /Fo " o/o (RelativeDir)\% (Filename).fxo” " % (FuIIPath) "**

你可以在 SDK 文档中找到 fxc 完整的编译参数说明。在调试模式中我们使用了以下三个参数，“/Fc /Od /Zi”分别对应输出汇编指令，禁用优化，开启调试信息。

现在当生成项目时，就会在每个 effect 上调用 *fxc* 并生成它的编译版本，以后缀为.fxo

的文件的形式保存。而且，如果有来自于 *fxc* 的编译警告或错误，会在调试输出窗口显示相关信息。例如，如果在 color.fx 文件中打错了一个变量的名称：

```
// 应该是 gWorldViewProj 而不是 worldViewProj!
vout.PosH = mul(float4(vin.Pos, l.Of), worldViewProj);
```

在调试输出窗口就会显示从这个错误引发的一系列错误的信息（第一条错误信息是修正的关键）：

```
error X3004: undeclared identifier  'worldViewProj'
error X3013: 'mul': intrinsic function does not take 2 parameters
error X3013: Possible intrinsic functions are:
error X3013: mul(float, float)…
```

在编译阶段获取错误信息要比运行时获取方便得多。现在我们在生成过程中编译 effect 文件（.fxo），再也不需要在运行时进行这个操作了（即，我们无须再调用 **D3DX11CompileFromFile** 方法了）。但是，我们仍需要从.fxo 文件中加载编译过的 shader，并将它们传递给 **D3DX11CreateEffectFromMemory** 方法。这个工作可以通过使用 C++的文件输入功能实现：

```
std::ifstream fin("fx/color.fxo",std::ios::binary);

fin.seekg(0, std::ios_base::end);
int size = (int)fin.tellg();
fin.seekg(0, std::ios_base::beg);
std::vector<char> compiledShader(size);

fin.read(&compiledShader[0],size);
fin.close();

HR(D3DX11CreateEffectFromMemory(&compiledShader[0],size,
    0, md3dDevice, &mFX));
```

除了在颜色立方体演示程序中我们在运行时编译了 shader 之外，本书的其他示例都是在生成过程中编译了所有 shader。

## 6.8.6 将 effect 框架作为“着色器生成器”

在本节的一开始我们提到过一个 effect 可以包含多个 technique。那为什么我们要使用多个 technique 呢？下面我们用阴影绘制为例子解释一下这个问题，但不会讨论实现阴影的细节内容。显然，阴影质量越高，要求的资源就越多。为了支持用户不同等级的显卡，我们可能会提供低、中、高不同质量的阴影技术。因此，即使只有一个阴影效果，我们也会使用多个 technique 去实现它。我们的阴影 effect 文件如下所示：

```
// 省略了常量缓冲，顶点结构等代码...
VertexOut VS(Vertexln vin) {/* Omit implementation details */}

float4 LowQualityPS(VertexOut pin) : SV_Target
{
    /* Do work common to all quality levels */
```

```
    /* Do low quality specific stuff */
    /* Do more work common to all quality levels */
}

float4 MediumQualityPS(VertexOut pin) : SV_Target
{
    /* Do work common to all quality levels */
    /* Do medium quality specific stuff */
    /* Do more work common to all quality levels */
}

float4 HighQualityPS(VertexOut pin) : SV_Target
{
    /* Do work common to all quality levels */
    /* Do high quality specific stuff */
    /* Do more work common to all quality levels */
}

technique11 ShadowsLow
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
        SetPixeIShader(CompileShader(ps_5_0, LowQualityPS()));
    }
}

technique11 ShadowsMedium
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
        SetPixeIShader(CompileShader(ps_5_0, MediumQualityPS()));
    }
}

technique11 ShadowsHigh
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
        SetPixeIShader(CompileShader(ps_5_0, HighQualityPS ()));
    }
}
```

C++应用程序会侦测玩家的显卡等级，选择最合适的 technique 进行渲染。

**注意**：前面的代码假设三个不同的阴影 technique 只在像素着色器中有所区别，所有的 technique 共享相同的顶点着色器。但是，每个 technique 都有不同的顶点着色器也是有可能的。

前面的实现中还有一个问题：即使像素着色器的代码是不同的，但是还是有一些通用的代码是重复的。建议使用条件分支语句解决这个问题。在 shader 中使用动态分支语句代价不菲，所以只在必要时才使用它们。其实我们真正想要的是一个条件编译，它可以生成不同的 shader 代码，但又不使用分支指令。幸运的是，effect 框架提供了一个方法可以解决这个问题。下面是具体实现：

```
// 省略常量缓冲，顶点结构等...
VertexOut VS(VertexIn vin) {/* 省略代码细节 */}
#define LowQuality 0
#define MediumQuality 1
#define HighQuality 2

float4 PS(VertexOut pin, uniform int gQuality) : SV_Target
{
    /* Do work common to all quality levels */
    if(gQuality == LowQuality)
    {
       /* Do low quality specific stuff */
    }
    elseif(gQuality == MediumQuality)
    {
        /* Do medium quality specific stuff */
    }
    else
    {
        /* Do high quality specific stuff */
    }
    /* Do more work common to all quality levels */
}

technique11 ShadowsLow
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
        SetPixeIShader(CompileShader(ps_5_0, PS(LowQuality)));
    }
}

technique11 ShadowsMedium
{
   pass P0
```

```
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
        SetPixeIShader(CompileShader(ps_5_0, PS(MediumQuality)));
    }
}

techniquell ShadowsHigh
{
   pass P0
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
        SetPixeIShader(CompileShader(ps_5_0, PS(HighQuality)));
    }
}
```

我们在像素着色器中添加了一个额外的 uniform 参数，用来表示阴影质量等级。这个参数值是不同的，但对每个像素来说却是不变的，but is instead uniform/constant。Moreover，we do not change it at runtime either，like we change constant buffer variables。我们是在编译时设置这些参数的，而且这些值在编译时就是已知的，所以 effect 框架会基于这个值生成不同的 shader 变量。这样，我们不用复制代码（effect 框架帮我们在编译时复制了这些代码）就可以生成低、中、高三种不同阴影质量的 shader 代码，而且没有用到条件分支语句。

下面的两个例子是使用 shader 生成器的常见情景：

**1．是否需要纹理**？有个应用程序需要在一些物体上施加纹理，而另一些物体不使用纹理。一个解决方法是创建两个像素着色器，一个提供纹理而另一个不提供。或者我们也可以使用 shader 生成技巧创建两个像素着色器，然后在 C++程序中选择期望的 technique。

```
float4 PS(VertexOut pin, uniform bool gApplyTexture) : SV_Target
{
    /* Do common work */
    if(gApplyTexture)
    {
        /* Apply texture */
    }
    /* Do more common work */
}

technique11 BasicTech
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
        SetPixelShader(CompileShader(ps_5_0, PS(false)));
    }
}

technique11 TextureTech
```

```
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
        SetPixelShader(CompileShader(ps_5_0,PS(true)));
    }
}
```

**2．使用多少个光源？**一个游戏关卡可能会支持 1 至 4 个光源。光源越多，光照计算就越慢。我们可以基于光源数量设计不同的顶点着色器，或者也可以使用 shader 生成技巧创建四个顶点着色器，然后在 C++程序中根据当前激活的光源数量选择期望的 technique：

```
VertexOut VS(VertexOut pin, uniform int gLightCount)
{
    /* Do common work */
    for(int i = 0; i< gLightCount; ++i)
    {
        /* do lighting work */
    }
    /* Do more common work */
}

technique11 Light1
{
    P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS(1)));
        SetPixeIShader(CompileShader(ps_5_0, PS()));
    }
}

technique11 Light2
{
    P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS(2)));
        SetPixeIShader(CompileShader(ps_5_0, PS()));
    }
}
technique11 Light3
{
    P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS(3)));
        SetPixeIShader(CompileShader(ps_5_0, PS()));
    }
```

```
}
technique11 Light4
{
    P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS(4)));
        SetPixeIShader(CompileShader(ps_5_0, PS()));
    }
}
```

参数也可以不止一个。要将阴影质量，纹理和多个光源组合在一起，我们可以使用以下的顶点和像素着色器：

```
VertexOut VS(VertexOut pin, uniform int gLightCount)
{}
float4 PS(VertexOut pin,uniform int gQuality,uniform bool
gApplyTexture) : SV_Target
{}
```

要创建一个使用低质量阴影，两个光源，不使用纹理的 technique，我们可以这样写代码：

```
technique11 LowShadowsTwoLightsNoTextures
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS(2)));
        SetPixeIShader(CompileShader(ps_5_0,PS(LowQuality,false)));
    }
}
```

## 6.8.7 对应的汇编代码

In case you do not bother to ever look at the assembly output of one of your effect files, we show what one looks like in this section.

我们不会解释汇编代码，但是，如果你以前学过汇编，你可能会认得代码中的 mov 指令，and

maybe could guess thatdp4 does a 4D dot product. Even without understanding the assembly, the listing gives some useful

information. It clearly identifies the input and output signatures, and gives the approximate mstruction count, which is one useful metric

to understand how expensive/complex your shader is. Moreover, we see that multiple versions of our shaders really were generated based on the compile time parameters with no branching instructions. The effect file we use is the same as the one shown in ~6.8.1,except that we add a simple uniform bool parameter to generate two techniques:

float4 PS(VertexOut pin, uniform bool gUseColor) : SV_Target

```
            {
 if(gUs e Color)
{
   return pin.Color;
else
{
   return float4(0, 0, 0, 1);          techniquell ColorTech
            {
pass PO
{
      SetVe rtexShade r(Compile Shade r(vs_5_0,    VSO》;
   SetPixeIShader(CompileShader(ps_5_0, PS(true》);
         techniquell NoColorTech
            {
pass PO
{
      SetVertexShader(CompileShade r(vs_5_0,    VS(》);
   SetPixeIShader(CompileShader(ps_5_0, PS(false》);
      //
            // FX Version: fx_5_0
      //
             11 1 local buffer(s)
      //
          cbuffer cbPerObject
            {
float4x4 gWorldViewProj; // Offset: O, size: 64
      //
             11 1 groups(s)
      //
fxgroup
            {
//
11 2 technique(s)
//
techniquell ColorTech
{
    pass PO
     {
            VertexShader = asm {
            //
                        // Generated by Microsoft (R) HLSL Shader Compiler 9.29.952.3111
            //
            //
                         // Buffer Definitions:
```

```
            //
                        // cbuffer cbPerObject
                       // {
            //
                         // float4x4 gWorldViewProj; // Offset: O Size: 64
            //
                        // }
            //
            //
                         // Resource Bindings:
            //
                         // Name Type Format Dim Slot Elements
                        // -----.
                        // cbPerObject cbuffer NA NA 0 1
            //     //
    //
              // Input signature:
    //
          // Name Index Mask Register SysValue Format Used
          // -----
           // POSlrflON 0 xyz O NONE float xyz
           // COLOR O xyzw l NONE float xyzw
    //
    //
          // Output signature:
    //
          // Name Index Mask Register SysValue Format Used
          // -----
          // SV_POSITION O xyzw O POS float xyzw
           // COLOR O xyzw l NONE float xyzw
    //
vs_5_0
        dcl_globaIFlags refactoringAllowed
                        dcl_constantbuffe r    cb0 [4],    imme diatelnde xe d
          dcljnput vO.xyz
          dcl_input vl.xyzw
          dcl_output_siv oO.xyzw, position
          dcl_output ol.xyzw
        dcl_temps 1
          mov rO.xyz, vO.xyzx
             mov rO.w, l(l.000000)
          dp4 00.x, rO.xyzw, cbO[O].xyzw
          dp4 00.y, rO.xyzw, cbO[l].xyzw
          dp4 00.z, rO.xyzw, cb0[2].xyzw
```

```
        dp4 00.w, rO.xyzw, cb0[3].xyzw
        mov ol.xyzw, vl.xyzw
   ret
          // Approximately 8 instruction slots used
};
PixeIShader = asm {
    //
        // Generated by Microsoft (R) HLSL Shader Compiler 9.29.952.3111
    //
    //
    //
            // Input signature:
    //
        // Name Index Mask Register SysValue Format Used
        // -----
        // SV_POSIUON O xyzw o POS float
         // COLOR O xyzw l NONE float xyzw
    //
    //
        // Output signature:
    //
        // Name Index Mask Register SysValue Format Used
        // -----
        // SV_Target O xyzw O TARGET float xyzw
    //
ps_5_0
       dcl_globaIFlags refactoringAllowed
        dcl_input_ps linear vl.xyzw
        dcl_output oO.xyzw
        mov oO.xyzw, vl.xyzw
   ret                              // Approximately 2 instruction slots used
         };
techniquell NoColorTech
{
    pass PO
     {
            VertexShader = asm {
            //
                    // Generated by Microsoft (R) HLSL Shader Compiler 9.29.952.3111
            //
            //
                     // Buffer Definitions:
            //
                    // cbuffer cbPerObject
```

```
                    // {
//
                      // float4x4 gWorldViewProj; // Offset: O Size: 64
//
                    // }
//
//
                      // Resource Bindings:
//
                      // Name Type Format Dim Slot Elements
                    // -----.
                    // cbPerObject cbuffer NA NA 0 1
//
//
//
                            // Input signature:
//
                      // Name Index Mask Register SysValue Format Used
                    // -----.
                        // POSlrflON O xyz O NONE float xyz
                          // COLOR O xyzw l NONE float xyzw
//
//
                    // Output signature:
//
                      // Name Index Mask Register SysValue Format Used
                    // -----.
                      // SV_POSITION O xyzw O POS float xyzw
                          // COLOR 0 xyzw l NONE float xyzw
//
vs_5_0
          dcl_globaIFlags refactoringAllowed
                                        dcl_constantbuffe r cb0 [4], imme
diatelnde xe d
            dcljnput vO.xyz
            dcl_input vl.xyzw
              dcl_output_siv oO.xyzw, position
            dcl_output ol.xyzw
           dcl_temps 1
              mov rO.xyz, vO.xyzx
                  mov rO.w, 1(1.000000)
            dp4 00.x, rO.xyzw, cbO[O].xyzw
            dp4 00.y, rO.xyzw, cbO[l].xyzw
              dp4 00.z, rO.xyzw, cb0[2].xyzw
```

```
                        dp4 00.w, rO.xyzw, cb0[3].xyzw
                         mov ol.xyzw, vl.xyzw}
}
}
   ret
            // Approximately 8 instruction slots used
};
PixeIShader = asm {
Generated by Microsoft (R) HLSL Shader Compiler 9.29.952.3111
Input signature:
Name Index Mask Register SysValue Format Used
SV_POSITION O xyzw O POS float
COLOR O xyzw l NONE float
Output signature:
Name Index Mask Register SysValue Format Used
SV_Target O xyzw O TARGET float xyzw
ps_5_0
       dcl_globaIFlags refactoringAllowed
        dcl_output oO.xyzw
          mov oO.xyzw, 1(0,0,0,1.000000)
   ret
            // Approximately 2 instruction slots used
```

6.9 BOX DEMO

At last, we have covered enough material to present a simple demo, which renders a colored box. This example essentially puts

everythmg we have discussed in this chapter up to now into a single program. The reader should study the code and refer back to the

previous sections of this chapter until every line is understood. Note that the program uses the "color.fx" effect, as written in ~6.8.1.

，/击击出士击击出士击出去击击出去击击出士击击出士击击击击击出去击击出去击击出士击击出士击出去击击出去击击古士击击出士击击出去击出去击击出去击

```
// BoxDemo.cpp by Frank Luna (C) 2011 All Rights Reserved.
//
／／Demonstrates rendering a colored box.
||
/ / Controls：
／／Hold the left mouse button down and move the mouse to rotate.
／／Hold the right mouse button down to zoom in and out.
||
```

，/击出去击击出去击击古去士击古去士出去击击出去击击出去士击古去士击古去击出去击击出去击击古去士击古去士击击击击出去击击出去士击古去士击古去击

```
#include "d3dApp.h"
#include "d3dxllEffect.h"
#include "MathHelper.h"
```

```
struct Vertex
        XMFLOArl3 Pos;
        XMFLOAT4 Color;
// // // // // // // // // // // // // // // // // // // // //
```
暂未整理。

# 6.9 颜色立方体演示程序

我们已经讲解了足够多的内容，现在我们可以开始编写一个简单的颜色立方体演示程序了。这个例子基本上包含了我们前面讲到的所有内容。读者应该对照前面的几节，仔细研究这些代码，直到把每一行代码都弄懂为止。注意，程序使用了 6.8.1 节编写的“color.fx”effect。

```
//***************************************************************************
************************
// BoxDemo.cpp by Frank Luna (C) 2011 All Rights Reserved.
//
// Demonstrates rendering a colored box.
//
// Controls:
//     Hold the left mouse button down and move the mouse to rotate.
//      Hold the right mouse button down to zoom in and out.
//
//***************************************************************************
************************

#include "d3dApp.h"
#include "d3dx11Effect.h"
#include "MathHelper.h"

struct Vertex
{
    XMFLOAT3 Pos;
    XMFLOAT4 Color;
};

class BoxApp : public D3DApp
{
public:
    BoxApp(HINSTANCE hInstance);
    ~BoxApp();

    bool Init();
    void OnResize();
    void UpdateScene(float dt);
    void DrawScene();

    void OnMouseDown(WPARAM btnState, int x, int y);
    void OnMouseUp(WPARAM btnState, int x, int y);
    void OnMouseMove(WPARAM btnState, int x, int y);
```

```
private:
    void BuildGeometryBuffers();
    void BuildFX();
    void BuildVertexLayout();

private:
    ID3D11Buffer* mBoxVB;
    ID3D11Buffer* mBoxIB;

    ID3DX11Effect* mFX;
    ID3DX11EffectTechnique* mTech;
    ID3DX11EffectMatrixVariable* mfxWorldViewProj;

    ID3D11InputLayout* mInputLayout;

    XMFLOAT4X4 mWorld;
    XMFLOAT4X4 mView;
    XMFLOAT4X4 mProj;

    float mTheta;
    float mPhi;
    float mRadius;

    POINT mLastMousePos;
};

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE prevInstance,
                   PSTR cmdLine, int showCmd)
{
    // Enable run-time memory check for debug builds.
#if defined(DEBUG) | defined(_DEBUG)
    _CrtSetDbgFlag( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
#endif

    BoxApp theApp(hInstance);

    if( !theApp.Init() )
        return 0;

    return theApp.Run();
}


BoxApp::BoxApp(HINSTANCE hInstance)
```

```
: D3DApp(hInstance), mBoxVB(0), mBoxIB(0), mFX(0), mTech(0),
  mfxWorldViewProj(0), mInputLayout(0),
  mTheta(1.5f*MathHelper::Pi),          mPhi(0.25f*MathHelper::Pi),
mRadius(5.0f)
{
    mMainWndCaption = L"Box Demo";

    mLastMousePos.x = 0;
    mLastMousePos.y = 0;

    XMMATRIX I = XMMatrixIdentity();
    XMStoreFloat4x4(&mWorld, I);
    XMStoreFloat4x4(&mView, I);
    XMStoreFloat4x4(&mProj, I);
}

BoxApp::~BoxApp()
{
    ReleaseCOM(mBoxVB);
    ReleaseCOM(mBoxIB);
    ReleaseCOM(mFX);
    ReleaseCOM(mInputLayout);
}

bool BoxApp::Init()
{
    if(!D3DApp::Init())
        return false;

    BuildGeometryBuffers();
    BuildFX();
    BuildVertexLayout();

    return true;
}

void BoxApp::OnResize()
{
    D3DApp::OnResize();

    // 当窗口大小改变时，需要更新横纵比，并重新计算投影矩阵
    XMMATRIX   P   =   XMMatrixPerspectiveFovLH(0.25f*MathHelper::Pi,
AspectRatio(), 1.0f, 1000.0f);
    XMStoreFloat4x4(&mProj, P);
```

```
}

void BoxApp::UpdateScene(float dt)
{
    // Convert Spherical to Cartesian coordinates.
    float x = mRadius*sinf(mPhi)*cosf(mTheta);
    float z = mRadius*sinf(mPhi)*sinf(mTheta);
    float y = mRadius*cosf(mPhi);

    // 创建视矩阵
    XMVECTOR pos    = XMVectorSet(x, y, z, 1.0f);
    XMVECTOR target = XMVectorZero();
    XMVECTOR up     = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);

    XMMATRIX V = XMMatrixLookAtLH(pos, target, up);
    XMStoreFloat4x4(&mView, V);
}

void BoxApp::DrawScene()
{
    md3dImmediateContext->ClearRenderTargetView(mRenderTargetView,
reinterpret_cast<const float*>(&Colors::LightSteelBlue));
    md3dImmediateContext->ClearDepthStencilView(mDepthStencilView,
D3D11_CLEAR_DEPTH|D3D11_CLEAR_STENCIL, 1.0f, 0);

    md3dImmediateContext->IASetInputLayout(mInputLayout);

md3dImmediateContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPO
LOGY_TRIANGLELIST);

    UINT stride = sizeof(Vertex);
    UINT offset = 0;
    md3dImmediateContext->IASetVertexBuffers(0, 1, &mBoxVB, &stride,
&offset);
    md3dImmediateContext->IASetIndexBuffer(mBoxIB,
DXGI_FORMAT_R32_UINT, 0);

    // Set constants
    XMMATRIX world = XMLoadFloat4x4(&mWorld);
    XMMATRIX view  = XMLoadFloat4x4(&mView);
    XMMATRIX proj  = XMLoadFloat4x4(&mProj);
    XMMATRIX worldViewProj = world*view*proj;

    mfxWorldViewProj->SetMatrix(reinterpret_cast<float*>(&worldVie
```

```
wProj));

    D3DX11_TECHNIQUE_DESC techDesc;
    mTech->GetDesc( &techDesc );
    for(UINT p = 0; p < techDesc.Passes; ++p)
    {
        mTech->GetPassByIndex(p)->Apply(0, md3dImmediateContext);

        // 立方体有 36 个索引
        md3dImmediateContext->DrawIndexed(36, 0, 0);
    }

    HR(mSwapChain->Present(0, 0));
}

void BoxApp::OnMouseDown(WPARAM btnState, int x, int y)
{
    mLastMousePos.x = x;
    mLastMousePos.y = y;

    SetCapture(mhMainWnd);
}

void BoxApp::OnMouseUp(WPARAM btnState, int x, int y)
{
    ReleaseCapture();
}

void BoxApp::OnMouseMove(WPARAM btnState, int x, int y)
{
    if( (btnState & MK_LBUTTON) != 0 )
    {
        // Make each pixel correspond to a quarter of a degree.
        float dx = XMConvertToRadians(0.25f*static_cast<float>(x -
mLastMousePos.x));
        float dy = XMConvertToRadians(0.25f*static_cast<float>(y -
mLastMousePos.y));

        // Update angles based on input to orbit camera around box.
        mTheta += dx;
        mPhi   += dy;

        // Restrict the angle mPhi.
        mPhi = MathHelper::Clamp(mPhi, 0.1f, MathHelper::Pi-0.1f);
```

```
    }
    else if( (btnState & MK_RBUTTON) != 0 )
    {
        // Make each pixel correspond to 0.005 unit in the scene.
        float dx = 0.005f*static_cast<float>(x - mLastMousePos.x);
        float dy = 0.005f*static_cast<float>(y - mLastMousePos.y);

        // Update the camera radius based on input.
        mRadius += dx - dy;

        // Restrict the radius.
        mRadius = MathHelper::Clamp(mRadius, 3.0f, 15.0f);
    }

    mLastMousePos.x = x;
    mLastMousePos.y = y;
}

void BoxApp::BuildGeometryBuffers()
{
    // 创建顶点缓冲
    Vertex vertices[] =
    {
        { XMFLOAT3(-1.0f, -1.0f, -1.0f), (const float*)&Colors::White   },
        { XMFLOAT3(-1.0f, +1.0f, -1.0f), (const float*)&Colors::Black   },
        { XMFLOAT3(+1.0f, +1.0f, -1.0f), (const float*)&Colors::Red     },
        { XMFLOAT3(+1.0f, -1.0f, -1.0f), (const float*)&Colors::Green   },
        { XMFLOAT3(-1.0f, -1.0f, +1.0f), (const float*)&Colors::Blue    },
        { XMFLOAT3(-1.0f, +1.0f, +1.0f), (const float*)&Colors::Yellow  },
        { XMFLOAT3(+1.0f, +1.0f, +1.0f), (const float*)&Colors::Cyan    },
        { XMFLOAT3(+1.0f, -1.0f, +1.0f), (const float*)&Colors::Magenta }
    };

    D3D11_BUFFER_DESC vbd;
    vbd.Usage = D3D11_USAGE_IMMUTABLE;
    vbd.ByteWidth = sizeof(Vertex) * 8;
```

```
    vbd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
    vbd.CPUAccessFlags = 0;
    vbd.MiscFlags = 0;
    vbd.StructureByteStride = 0;
    D3D11_SUBRESOURCE_DATA vinitData;
    vinitData.pSysMem = vertices;
    HR(md3dDevice->CreateBuffer(&vbd, &vinitData, &mBoxVB));


    // 创建索引缓冲

    UINT indices[] = {
        // 前表面
        0, 1, 2,
        0, 2, 3,

        // 后表面
        4, 6, 5,
        4, 7, 6,

        // 左表面
        4, 5, 1,
        4, 1, 0,

        // 右表面
        3, 2, 6,
        3, 6, 7,

        // 上表面
        1, 5, 6,
        1, 6, 2,

        // 下表面
        4, 0, 3,
        4, 3, 7
    };

    D3D11_BUFFER_DESC ibd;
    ibd.Usage = D3D11_USAGE_IMMUTABLE;
    ibd.ByteWidth = sizeof(UINT) * 36;
    ibd.BindFlags = D3D11_BIND_INDEX_BUFFER;
    ibd.CPUAccessFlags = 0;
    ibd.MiscFlags = 0;
    ibd.StructureByteStride = 0;
```

```
    D3D11_SUBRESOURCE_DATA iinitData;
    iinitData.pSysMem = indices;
    HR(md3dDevice->CreateBuffer(&ibd, &iinitData, &mBoxIB));
}

void BoxApp::BuildFX()
{
    DWORD shaderFlags = 0;
    #if defined( DEBUG ) || defined( _DEBUG )
        shaderFlags |= D3D10_SHADER_DEBUG;
        shaderFlags |= D3D10_SHADER_SKIP_OPTIMIZATION;
    #endif

    ID3D10Blob* compiledShader = 0;
    ID3D10Blob* compilationMsgs = 0;
    HRESULT hr = D3DX11CompileFromFile(L"FX/color.fx", 0, 0, 0,
"fx_5_0", shaderFlags,
        0, 0, &compiledShader, &compilationMsgs, 0);

    // compilationMsgs 中包含错误或警告信息
    if( compilationMsgs != 0 )
    {
        MessageBoxA(0, (char*)compilationMsgs->GetBufferPointer(), 0,
0);
        ReleaseCOM(compilationMsgs);
    }

    // 就算没有 compilationMsgs，也需要确保没有其他错误
    if(FAILED(hr))
    {
        DXTrace(__FILE__,              (DWORD)__LINE__,              hr,
L"D3DX11CompileFromFile", true);
    }

    HR(D3DX11CreateEffectFromMemory(compiledShader->GetBufferPoint
er(), compiledShader->GetBufferSize(),
        0, md3dDevice, &mFX));

    // 编译完成释放资源
    ReleaseCOM(compiledShader);

    mTech    = mFX->GetTechniqueByName("ColorTech");
    mfxWorldViewProj                                              =
mFX->GetVariableByName("gWorldViewProj")->AsMatrix();
```

```
}

void BoxApp::BuildVertexLayout()
{
    // 顶点输入布局描述
    D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
    {
        {"POSITION",    0,    DXGI_FORMAT_R32G32B32_FLOAT,    0,    0,
D3D11_INPUT_PER_VERTEX_DATA, 0},
        {"COLOR",       0,  DXGI_FORMAT_R32G32B32A32_FLOAT,  0,  12,
D3D11_INPUT_PER_VERTEX_DATA, 0}
    };

    // 创建顶点输入布局
    D3DX11_PASS_DESC passDesc;
    mTech->GetPassByIndex(0)->GetDesc(&passDesc);
    HR(md3dDevice->CreateInputLayout(vertexDesc,                    2,
passDesc.pIAInputSignature,
        passDesc.IAInputSignatureSize, &mInputLayout));
}
```

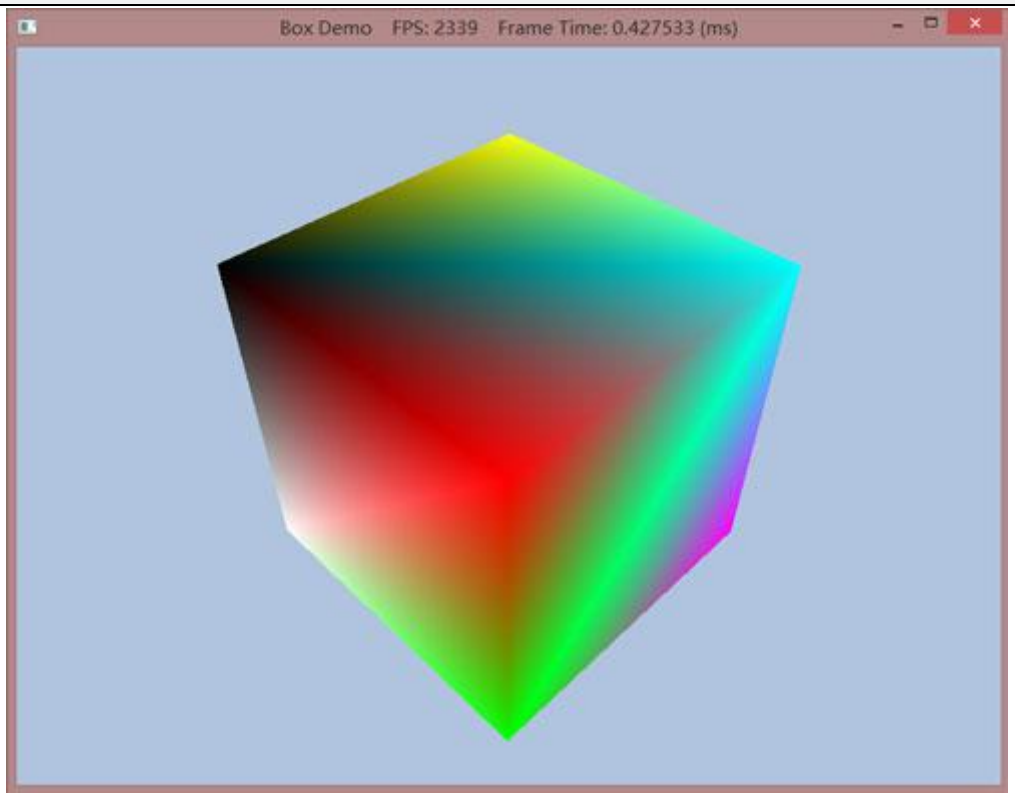



**图 6.7 立方体演示程序的屏幕截图**

# 6.10 山峰与河谷演示程序

本章还包含了一个“山峰与河谷”的例子。它使用了与颜色立方体演示程序相同的Direct3D 方法，只是它绘制的几何体更复杂一些。它主要讲解的是如何使用代码来生成三角形网格；这种几何体在实现地形渲染和水体渲染时非常有用。

实数函数 $y = f(x, z)$可以生成一个“漂亮的”曲面。我们可以通过构造一个 $xz$ 平面上的网格来模拟该曲面，其中每个四边形都由两个三角形构成。然后，我们将每个网格点代入该函数；参见图 6.8。

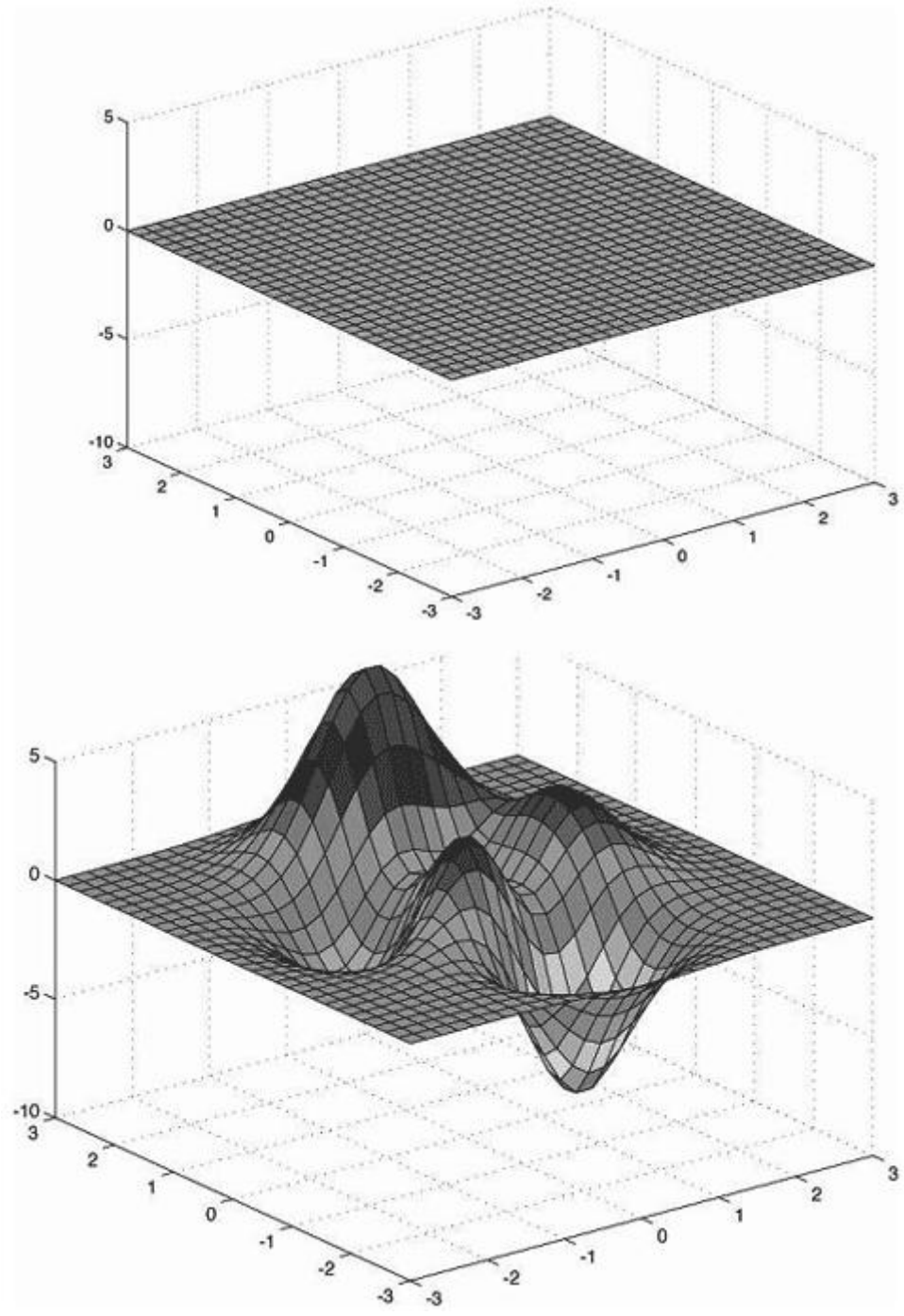



**图 6.8 （上）建立 $xz$ 平面上的网格。（下）将每个网格点代入函数 $f(x, z)$，得到 $y$ 坐标。通**

**过将大量的点$(x, f(x, z), z)$连接起来，即可形成上述曲面。**

## 6.10.1 生成网格顶点

下面的主要任务是创建 $xz$ 平面上的网格。一个包含 $m \times n$ 个顶点的网格可以生成$(m-1)\times(n-1)$个多边形（或单元格），如图 6.9 所示。每个多边形由两个三角形组成，一共 $2\times(m-1)\times(n-1)$个三角形。如果网格的宽度为 $w$、深度为 $d$，则 $x$ 轴方向上的单元格间距为 $dx = w/(n-1)$、 轴方向上的单元格间距为 $dz=d/(m-1)$。我们从左上角开始生成顶点，逐行计算每个顶点的坐标。在 $xz$ 平面上，第 $ij$ 个网格顶点的坐标为：

$$\mathbf{v}_{ij} = (-0.5w + j \cdot dx,\ 0.0,\ 0.5d - i \cdot dz)$$

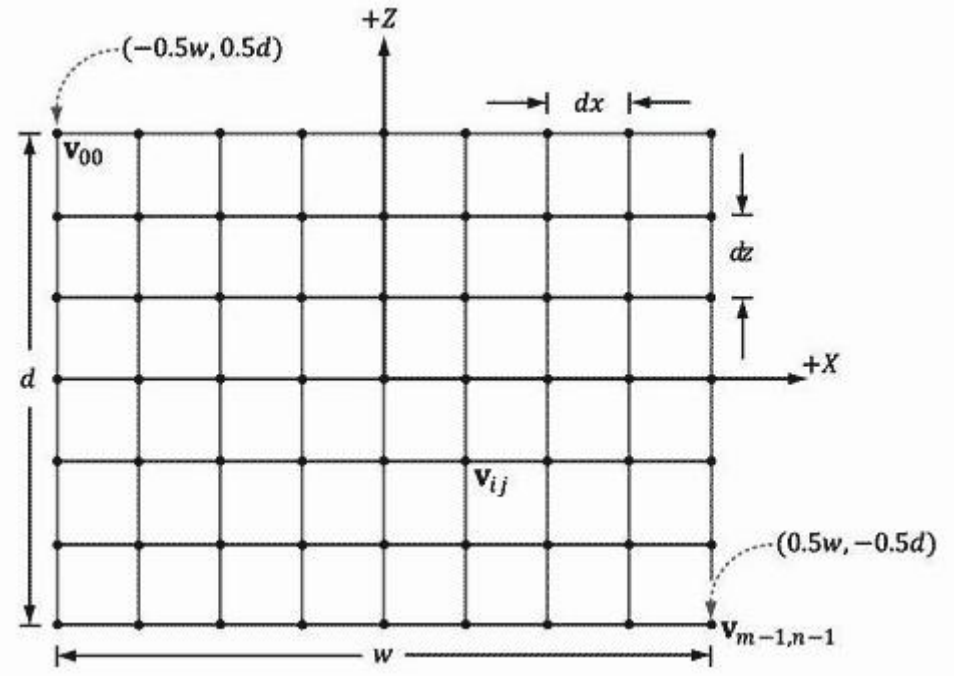



**图 6.9 网格结构。**

下面的代码实现了这一工作。

```
void GeometryGenerator::CreateGrid(float width, float depth, UINT m,
UINT n, MeshData& meshData)
{
    UINT vertexCount = m*n;
    UINT faceCount   = (m-1)*(n-1)*2;

    //
    // 创建顶点
    //

    float halfWidth = 0.5f*width;
    float halfDepth = 0.5f*depth;

    float dx = width / (n-1);
    float dz = depth / (m-1);
```

```
    float du = 1.0f / (n-1);
    float dv = 1.0f / (m-1);

    meshData.Vertices.resize(vertexCount);
    for(UINT i = 0; i < m; ++i)
    {
        float z = halfDepth - i*dz;
        for(UINT j = 0; j < n; ++j)
        {
            float x = -halfWidth + j*dx;

            meshData.Vertices[i*n+j].Position = XMFLOAT3(x, 0.0f, z);
            meshData.Vertices[i*n+j].Normal   = XMFLOAT3(0.0f, 1.0f,
0.0f);
            meshData.Vertices[i*n+j].TangentU = XMFLOAT3(1.0f, 0.0f,
0.0f);

            // Stretch texture over grid.
            meshData.Vertices[i*n+j].TexC.x = j*du;
            meshData.Vertices[i*n+j].TexC.y = i*dv;
        }
    }
}
```

**GeometryGenerator** 是一个工具类，用于生成诸如网格、球、圆柱体、盒子之类的几何形状，在本书的其他示例中都会用到这些形状。这个类在系统内存中生成数据，我们必须将这些数据复制到顶点和索引缓冲中。**GeometryGenerator** 创建的某些顶点数据在后面的章节中才会用到，这个演示程序不会用到，所以也无需将这些数据复制到顶点缓冲中。**MeshData** 结构体用于存储顶点和索引的集合列表。

```
class GeometryGenerator
{
public:
    struct Vertex
    {
        Vertex(){}
        Vertex(const XMFLOAT3& p, const XMFLOAT3& n, const XMFLOAT3&
t, const XMFLOAT2& uv)
            : Position(p), Normal(n), TangentU(t), TexC(uv){}
        Vertex(
            float px, float py, float pz,
            float nx, float ny, float nz,
            float tx, float ty, float tz,
            float u, float v)
            : Position(px,py,pz), Normal(nx,ny,nz),
```

```
            TangentU(tx, ty, tz), TexC(u,v){}

        XMFLOAT3 Position;
        XMFLOAT3 Normal;
        XMFLOAT3 TangentU;
        XMFLOAT2 TexC;
    };

    struct MeshData
    {
        std::vector<Vertex> Vertices;
        std::vector<UINT> Indices;
    };
…
};
```

## 6.10.2 生成网格索引

在完成顶点的计算之后，我们必须通过索引来定义网格三角形。我们再次从左上角开始逐行遍历每个四边形，通过计算索引来定义构成四边形的两个三角形。如图 6.10 所示，对于一个由 $m \times n$ 个顶点构成的网格来说，两个三角形的线性数组索引为：

$$\triangle ABC = (i \cdot n + j,\quad i \cdot n + j + 1,\quad (i+1) \cdot n + j)$$

$$\triangle CBD = ((i+1) \cdot n + j,\ i \cdot n + j + 1 \quad \cdot \quad (i+1) \cdot n + j + 1)$$

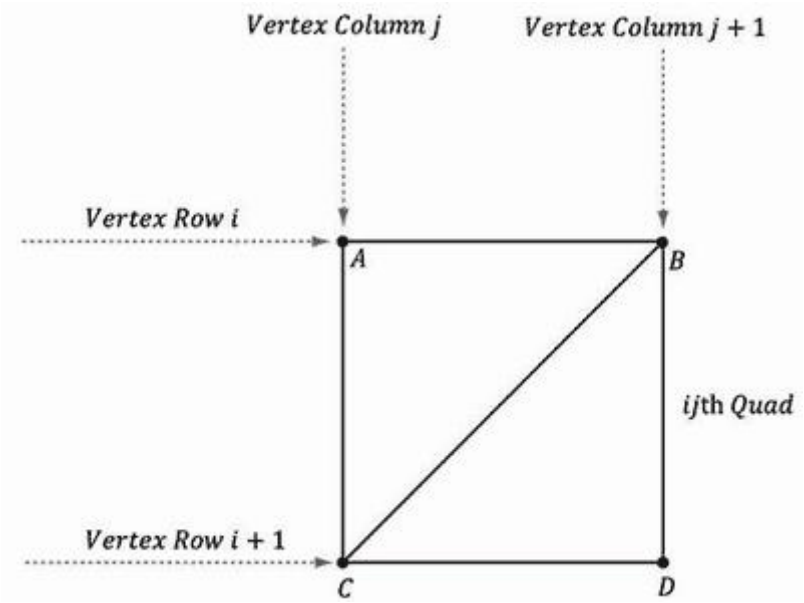



**图 6.10 第 *ij* 个四边形的顶点的索引。**

下面是对应的代码：

```
meshData.Indices.resize(faceCount*3); // 3 indices per face

// 遍历所有四边形并计算索引
```

```
UINT k = 0;
for(UINT i = 0; i < m-1; ++i)
{
    for(UINT j = 0; j < n-1; ++j)
    {
        meshData.Indices[k]   = i*n+j;
        meshData.Indices[k+1] = i*n+j+1;
        meshData.Indices[k+2] = (i+1)*n+j;

        meshData.Indices[k+3] = (i+1)*n+j;
        meshData.Indices[k+4] = i*n+j+1;
        meshData.Indices[k+5] = (i+1)*n+j+1;

        k += 6; // next quad
    }
}
```

## 6.10.3 代入高度函数

创建了网格之后，我们要从 MeshData 中提取顶点元素，将平面网格转换为代表山丘的曲折表面，并基于顶点的高度（$y$ 坐标）设置它们的颜色。

```
struct Vertex
{
    XMFLOAT3 Pos;
    XMFLOAT4 Color;
};

void HillsApp::BuildGeometryBuffers()
{
    GeometryGenerator::MeshData grid;

    GeometryGenerator geoGen;

    geoGen.CreateGrid(160.0f, 160.0f, 50, 50, grid);

    mGridIndexCount = grid.Indices.size();

    //
    // 在每个顶点上附加高度函数。此外，还根据顶点的高度设置它们的颜色：
    // 沙滩为沙的颜色，小山为绿色，山顶为白色的雪。
    //

    std::vector<Vertex> vertices(grid.Vertices.size());
```

```
    for(size_t i = 0; i < grid.Vertices.size(); ++i)
    {
        XMFLOAT3 p = grid.Vertices[i].Position;

        p.y = GetHeight(p.x, p.z);

        vertices[i].Pos   = p;

        // 根据顶点高度设置颜色
        if( p.y < -10.0f )
        {
            // 沙滩色
            vertices[i].Color = XMFLOAT4(1.0f, 0.96f, 0.62f, 1.0f);
        }
        else if( p.y < 5.0f )
        {
            // 淡绿色
            vertices[i].Color = XMFLOAT4(0.48f, 0.77f, 0.46f, 1.0f);
        }
        else if( p.y < 12.0f )
        {
            // 深绿色
            vertices[i].Color = XMFLOAT4(0.1f, 0.48f, 0.19f, 1.0f);
        }
        else if( p.y < 20.0f )
        {
            // 棕色
            vertices[i].Color = XMFLOAT4(0.45f, 0.39f, 0.34f, 1.0f);
        }
        else
        {
            // 白色
            vertices[i].Color = XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f);
        }
    }

    D3D11_BUFFER_DESC vbd;
    vbd.Usage = D3D11_USAGE_IMMUTABLE;
    vbd.ByteWidth = sizeof(Vertex) * grid.Vertices.size();
    vbd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
    vbd.CPUAccessFlags = 0;
    vbd.MiscFlags = 0;
    D3D11_SUBRESOURCE_DATA vinitData;
    vinitData.pSysMem = &vertices[0];
```

```
    HR(md3dDevice->CreateBuffer(&vbd, &vinitData, &mVB));

    //
    // 将所有网格的索引放入一个索引缓冲中
    //

    D3D11_BUFFER_DESC ibd;
    ibd.Usage = D3D11_USAGE_IMMUTABLE;
    ibd.ByteWidth = sizeof(UINT) * mGridIndexCount;
    ibd.BindFlags = D3D11_BIND_INDEX_BUFFER;
    ibd.CPUAccessFlags = 0;
    ibd.MiscFlags = 0;
    D3D11_SUBRESOURCE_DATA iinitData;
    iinitData.pSysMem = &grid.Indices[0];
    HR(md3dDevice->CreateBuffer(&ibd, &iinitData, &mIB));
}
```

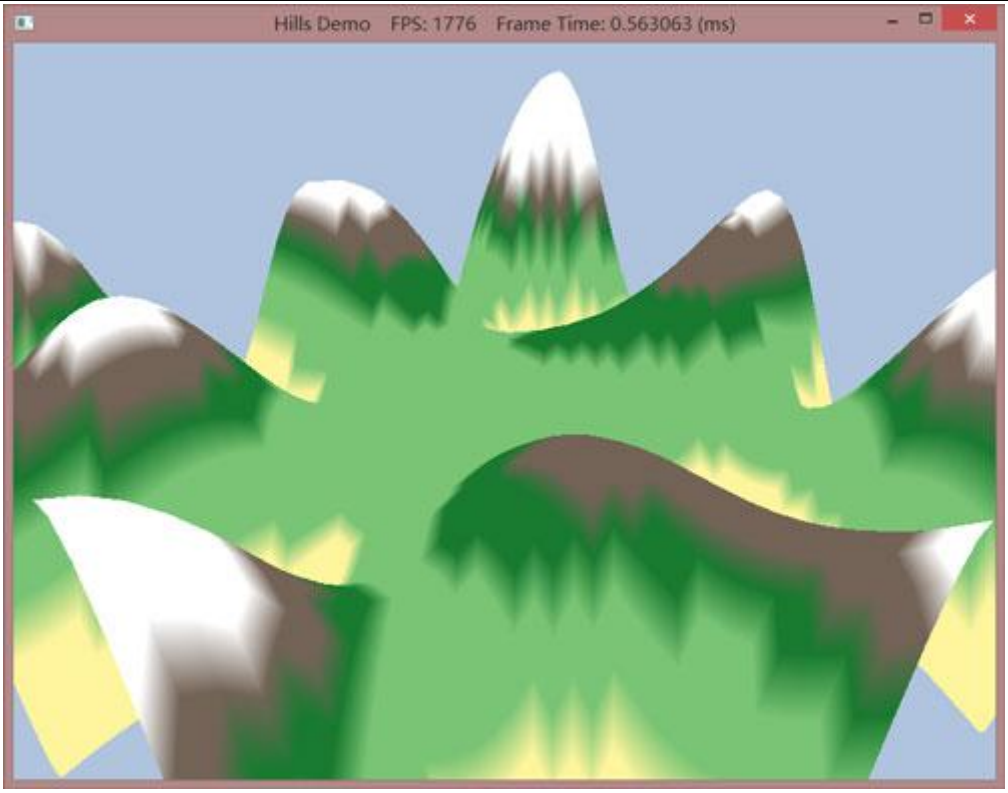



**图 6.11 山峰演示程序的截图。**

这个程序所用的函数 $f(x,z)$由以下代码给出：

```
float HillsApp::GetHeight(float x, float z)const
{
    return 0.3f*( z*sinf(0.1f*x) + x*cosf(0.1f*z) );
}
```

这个函数生成的图形看起了就像是山峰和山谷（见图 6.11），程序的其他部分与上一节颜色正方体的示例类似。

# 6.12 从文件加载几何体

虽然对于本书的某些示例来说，盒子、网格、球和圆柱形就足够了，但是还有些示例要绘制更加复杂的几何体。本书的后面我们会介绍如何从一个流行的 3D 模型格式加载 3D 网格。同时，我们已经将一个骷髅网格的几何体（见图 6.18）导出为一个顶点（只包含位置和法线向量）和索引的简单列表，可以使用标准的 C++文件 I/O 从文件中读取顶点和索引，并将它们复制到顶点和索引缓冲。文件的格式是一个非常简单的文本文件：

```
VertexCount:31076
TriangleCount:60339
VertexList(pos, normal)
{
    0. 592978  1. 92413  -2. 62486  0. 572276  0. 816877  0. 0721907
    0. 571224  1. 94331  -2. 66948  0. 572276  0. 816877  0. 0721907
    0. 609047  1. 90942  -2. 58578  0. 572276  0. 816877  0. 0721907
    …
}
TriangleList
{
    0  1  2
    3  4  5
    6  7  8
    …
}
```

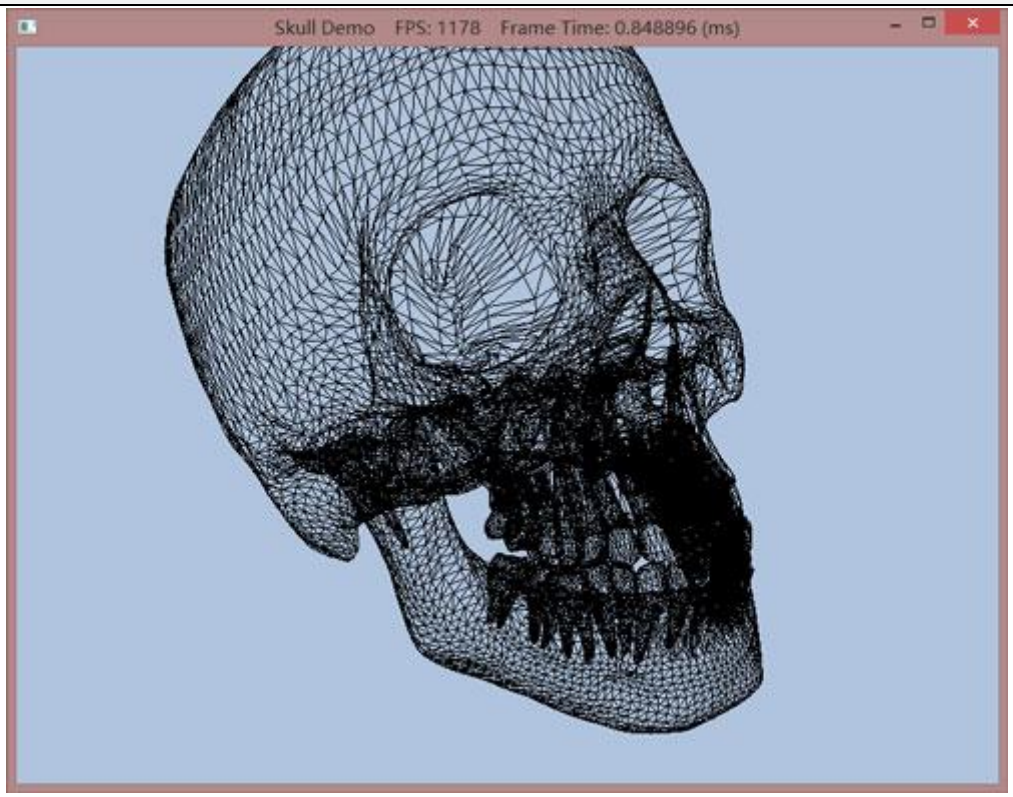



**图 6.18 “骷髅”演示示例的屏幕截图**

# 6.13 动态顶点缓冲

到目前为止，我们一直使用的是静态缓冲（static buffer），它的内容是在初始化时固定下来的。相比之下，动态缓冲（dynamic buffer）的内容可以在每一帧中进行修改。当实现一些动画效果时，我们通常使用动态缓冲区。例如，我们要模拟一个水波效果，并通过函数 $f(x, z, t)$来描述水波方程，计算当时间为 $t$ 时，$xz$ 平面上的每个点的高度。在这一情景中，我们必须使用“山峰与河谷”中的那种三角形网格，将每个网格点代入 $f(x, z, t)$函数得到相应的水波高度。由于该函数依赖于时间 $t$（即，水面会随着时间而变化），我们必须在很短的时间内（比如 1/30 秒）重新计算这些网格点，以得到较为平滑的动画。所以，我们必须使用动态顶点缓冲区来实时更新三角形网格顶点的高度。

前面提到，为了获得一个动态缓冲区，我们必须在创建缓冲区时将 Usage 标志值指定为 **D3D11_USAGE_DYNAMIC**；同时，由于我们要向缓冲区写入数据，所以必须将 CPU 访问标志值指定为 **D3D11_CPU_ACCESS_WRITE**。

```
D3D11_BUFFER_DESC vbd;
vbd.Usage = D3D11_USAGE_DYNAMIC;
vbd.ByteWidth = sizeof(Vertex) * mWaves.VertexCount();
vbd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
vbd.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
vbd.MiscFlags = 0;
HR(md3dDevice->CreateBuffer(&vbd, 0, &mWavesVB));
```

**然后，使用 ID3D11Buffer::Map** 函数获取缓冲区内存的起始地址指针，并向它写入数据：

```
HRESULT ID3D11DeviceContext::Map(
    ID3D11Resource *pResource ,
    UINT Subresource,
    D3D11_MAP MapType,
    UINT MapFlags,
    D3D11_MAPPED_SUBRESOURCE *pMappedResource);
```

**1．pResource**：指向要访问的用于读/写的资源的指针。缓冲是一种 Direct3D 11 资源，其他类型的资源，例如纹理资源，也可以使用这个方法进行访问。

**2．Subresource**：包含在资源中的子资源的索引。后面我们会看到如何使用这个索引，而缓冲不包含子资源，所以设置为 0。

**3．MapType**：常用的标志有以下几个：

- **D3D11_MAP_WRITE_DISCARD**：让硬件抛弃旧缓冲，返回一个指向新分配缓冲的指针，通过指定这个标志，可以让我们写入新分配的缓冲的同时，让硬件绘制已抛弃的缓冲中的内容，可以防止绘制停顿。
- **D3D11_MAP_WRITE_NO_OVERWRITE**：我们只会写入缓冲中未初始化的部分；通过指定这个标志，可以让我们写入未初始化的缓冲的同时，让硬件绘制前面已经写入的内容，可以防止绘制停顿。
- **D3D11_MAP_READ**：表示应用程序（CPU）会读取 GPU 缓冲的的一个副本到系统内存中。

**4．MapFlags**：可选标志，这里不使用，所以设置为 0；具体细节可参见 SDK 文档。

**5．pMappedResource**：返回一个指向 **D3D11_MAPPED_SUBRESOURCE** 的指针，这样我们就可以访问用于读/写的资源数据。

**D3D11_MAPPED_SUBRESOURCE** 结构定义如下：

```
typedef struct D3D11_MAPPED_SUBRESOURCE{
    void  *pData;
    UINT Row Pitch;
    UINT DepthPitch;
}D3D11_MAPPED_SUBRESOURCE
```

**1．pData**：指向用于读/写的资源内存的指针，你必须将它转换为资源中存储的数据的格式。

**2．RowPitch**：资源中一行数据的字节大小。例如，对于一个 2D 纹理来说，这个大小为一行的字节大小。

**3．DepthPitch**：资源中一页数据的大小。例如，对于一个 3D 纹理来说，这个大小为 3D 纹理中一个 2D 图像的字节大小。

**RowPitch** 和 **DepthPitch** 的区别是针对 2D 和 3D 资源（类似于 2D 和 3D 数组）而言的。顶点/索引缓冲本质上是 1D 数组，**RowPitch** 和 **DepthPitch** 的值是相同的，都等于顶点/索引缓冲的字节大小。

下面的代码展示如何在水波演示程序中更新顶点缓冲：

```
D3D11_MAPPED_SUBRESOURCE mappedData;
HR(md3dImmediateContext->Map(mWavesVB, 0, D3D11_MAP_WRITE_DISCARD,
0, &mappedData));

Vertex* v = reinterpret_cast<Vertex*>(mappedData.pData);
for(UINT i = 0; i < mWaves.VertexCount(); ++i)
{
    v[i].Pos = mWaves[i];
    v[i].Color = XMFLOAT4(0.0f, 0.0f, 0.0f, 1.0f);
}

md3dImmediateContext->Unmap(mWavesVB, 0);
```

当你完成缓冲区的更新操作之后，必须调用 **ID3D11Buffer::Unmap** 函数。

当使用动态缓冲区时，必然会有一些额外开销，因为这里存在一个从 CPU 内存向 GPU 内存回传数据的过程。 所以，在实际工作中应尽可能多使用静态缓冲区，少使用动态缓冲区。在 Direct3D 的最新版本中已经引入了一些新特性用于减少对动态缓冲区的需求。例如：

1．可以在顶点着色器中实现简单动画。

2．通过渲染到纹理（render to texture）和顶点纹理推送（vertex texture fetch）功能，可以实现完全运行在 GPU 上的水波模拟动画。

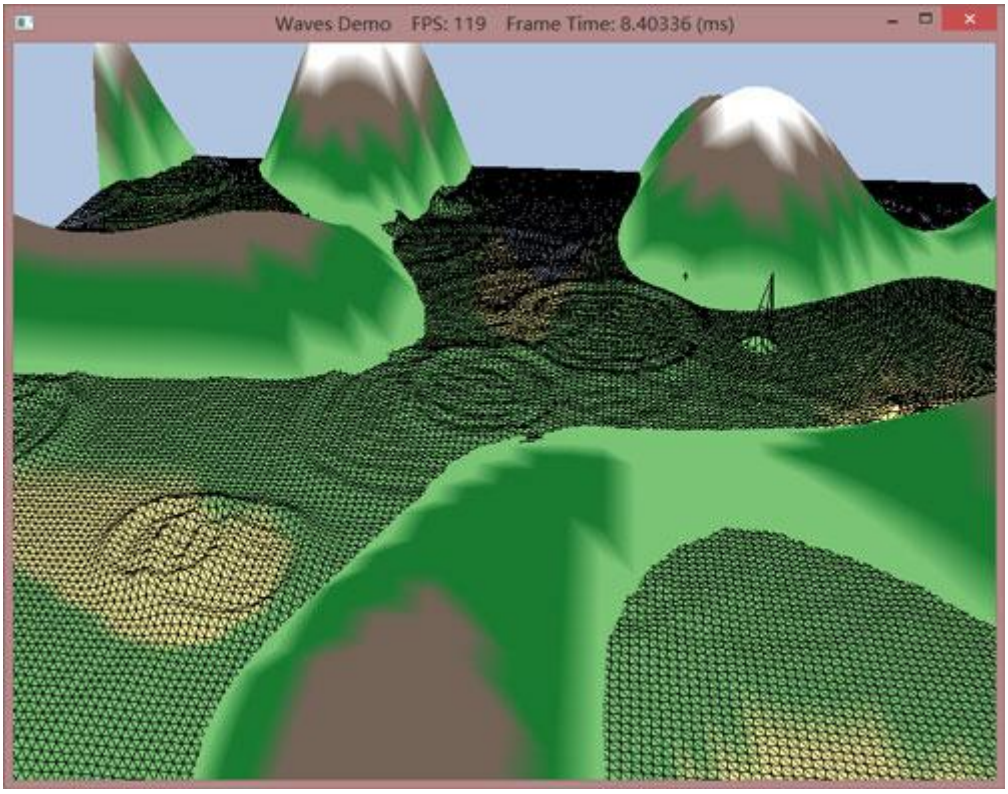
3．几何着色器为 GPU 提供了创建和销毁图元的能力，在以前没有几何着色器时，这些工作都是由 CPU 来完成的。

索引缓冲区可以是动态的。不过，在水波演示程序中，三角形的拓扑结构始终不变，只有顶点高度会发生变化；所以，这里只需要让顶点缓冲区变为动态缓冲区。

本章的水波演示程序使用了一个动态缓冲区来实现简单的水波效果。本书不会将重点放在水波模拟算法的实现细节上（有兴趣的读者可以参见[Lengyel02]），我们只是用它来说明动态缓冲区的用法：在 CPU 上更新模拟数据，然后调用 **Map/Unmap** 方法更新顶点缓冲区。

**注意**：在水波演示程序中，我们以线框模式渲染水波；是因为我们现在还没有讲到灯光的用法，在实体填充模式下，很难看出水波的运动效果。

**注意**：我们再次强调，这个示例应该在 GPU 上使用更高级的方式实现，比如渲染到纹理和顶点纹理推送。但是由于我们还没有讲到些技术，所以现在只能在 CPU 上实现，暂时使用动态顶点缓冲区来更新顶点。




**6.19 水波演示程序截图**

# 7.1 光照与材质的相互作用

图 7.1 展示了光照和阴影在表现物体的立体感和体积感时起到的重要作用。左边的球体没有灯照射，看上去就像是一个扁平的 2D 圆；而右边的球体有灯照射，看上去很立体。实际上，我们的视觉能力完全取决于光照及光照与材质之间的相互作用，所以，许多超写实场景的渲染工作都是通过精确的物理光照模型（lighting model）来实现的。（译者注：本章经常会提及“光照模型”个词，请读者不要将它与“3D 网格模型”的含义混淆。“光照模型”即不是说“用灯照射一个 3D 网格模型”，也不是指“支持光照运算的 3D 网格模型”。它的正确含义是“光照算法的数学模型（或者说数学公式）”。请读者一定要将“光照模型”与“3D 网格模型”的含义区分开。）

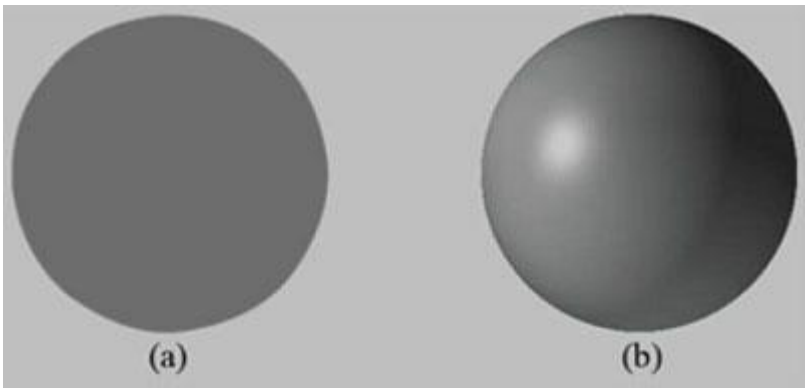



**图 7.1 (a)没有灯照明的球体看上去就像是一个 2D 圆。(b)有灯照明的球体看上去很立体。**

当然，一般来讲，越精确的光照模型，花费的计算时间就越长；我们必须在真实感和速度之间寻求平衡。例如，电影中的 3D 特效场景可以做得非常复杂，可以使用更为写实的光照模型，因为电影中的帧是预渲染的（pre-rendered），电影制作者可以花费数小时甚至数日的时间来渲染一帧。而游戏是实时渲染应用程序，它至少要以每秒 30 帧的速度渲染场景。

注意，本书关于光照模型的解释和实现方法大部分来自于[Möller02]的描述。

## 学习目标

1．了解光照与材质之间的相互作用。

2．了解局部照明和全局照明之间的区别。

3．了解如何以数学方式描述平面上的点所“面对”的方向，以使我们确定线与平面之间的入射角。

4．学习如何正确变换法线向量。

5．了解环境光、漫反射和镜面光之间的区别。

6．学习如何实现方向光、点光和聚光灯。

7．了解如何通过深度来控制衰减参数，改变光照的强度。

当使用光照时，我们不再直接指定顶点颜色；而是指定材质和灯光，然后使用光照方程，根据灯光与材质的相互作用计算顶点颜色。这样可以产生非常逼真的物体颜色（再次比较图 7.1a 和 7.1b 中的球体）。

材质可以被认为是决定光照如何与物体表面相互作用的属性。例如，表面反射的灯光颜

色、吸收的灯光颜色、反射率、透明度和光泽度都是构成表面材质的参数。不过，在本章中，我们主要讲解的是表面反射的灯光颜色、吸收的灯光颜色和光泽度。

在我们的光照模型中，光源可以发射各种强度的红、绿、蓝光；通过一方式，我们可以模拟很多灯光颜色。当光线从光源发出照射到一个物体上时，一部分光线会被物体吸收，另一部分线会被反射回来（对于透明物体，比如玻璃，还会有一部分光线会从物体中间穿过，不过在这里我们先不用考虑透明度的问题）。反射会沿着它的新路径传播，可能会照射在其他物体上，其中一部分线会被物体吸收，另一部分线会再次反射。在光线的能量完全耗尽之前，它会照射到许多物体。很可能会有一部分线最终传入人的眼睛（参见图 7.2），触碰到视网膜上的视感细胞（称为视锥细胞和视杆细胞）。

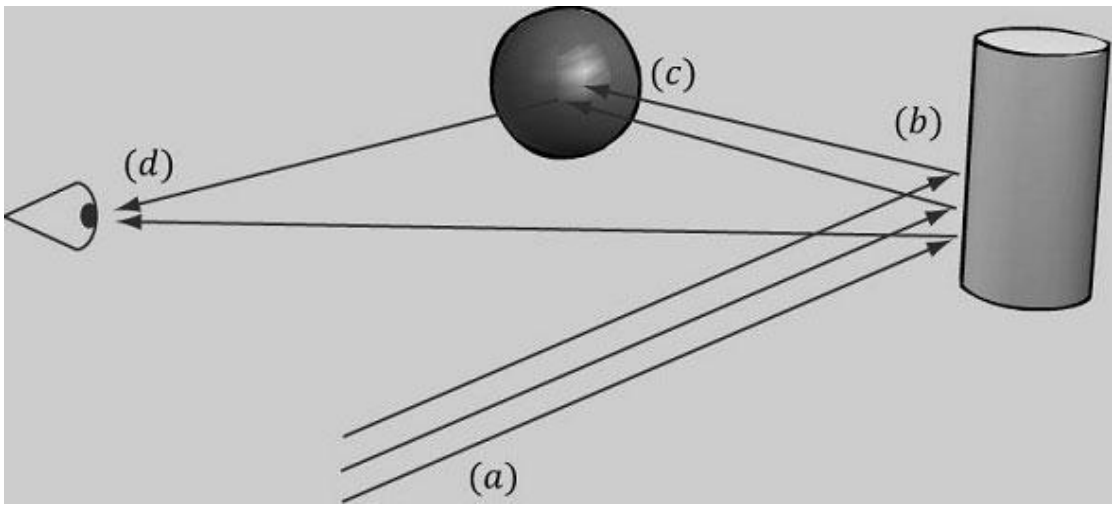



**图 7.2 (a)连续射入的白色光线。(b)当光线照射到圆柱体上时，一部分光线会被圆柱体吸收，另一部分光线分散传向眼睛和球体。(c)当圆柱体的反射光照射到球体上时，一部分光线会被球体吸收，另一部分光线会再次反射，传入眼睛。(d)眼睛收到入射线，看到物体。**

根据三原色理论（参见[Santrock03]），视网膜包含三种类型的有色感受器，分别对红、绿、蓝光（以及某些重叠部分）敏感。根据光的波长改变射入的 RGB 光线强度，刺激相应的感受器。这样，感受器就会受到刺激（或者不受刺激），神经触突会通过视觉神经传送到大脑，大脑根据感受器产生的信号形成头脑中的最终图像。（当然，如果你闭上或盖上眼睛，感受器细胞就不会受到刺激，大脑就会认为是黑色。）

例如，再次考虑图 7.2。假设圆柱体的材质反射 75%的红和 75%的绿光，其余线均被圆柱体吸收；球体反射 25%的红光，其余线均被球体吸收。同时，假设光源发射的线为纯白色光线。当光线照射到圆柱体上时，所有的蓝会被吸收，只有 75%的红和 75%的绿光被反射回来（即，中高强度的黄色线）。这些光线会产生散射，其中一部分光线会传入眼睛，另一部分线会传向球体。传进眼睛的那一部分线主要刺激的是红色和蓝色圆锥细胞；因此，观察者看到的圆柱体为亮黄色。现在，另一部分线会传向球体，并照射在球体表面上。球体反射 25%的红光，其余线均被球体吸收；那些曾被稀释过的红色（中高强度的红色线）会被再次稀释，所有射入的绿光均被吸收，只有一部分红光反射回来。然后些剩下的红光传入眼睛，对红色视锥细胞产生刺激。因此，观察者看到的球体为暗红色。

本书（和许多实时渲染应用程序）采用的光照模型都是局部光照模型（local illumination model）。在局部光照模型中，每个物体的光照都是相互独立的，所有的光线都是从光源直接照射到物体上（即，本来应该被场景中其他物体阻隔的线，仍然会照射到后面的物体上）。图 7.3 说明了一光照模型的特点。

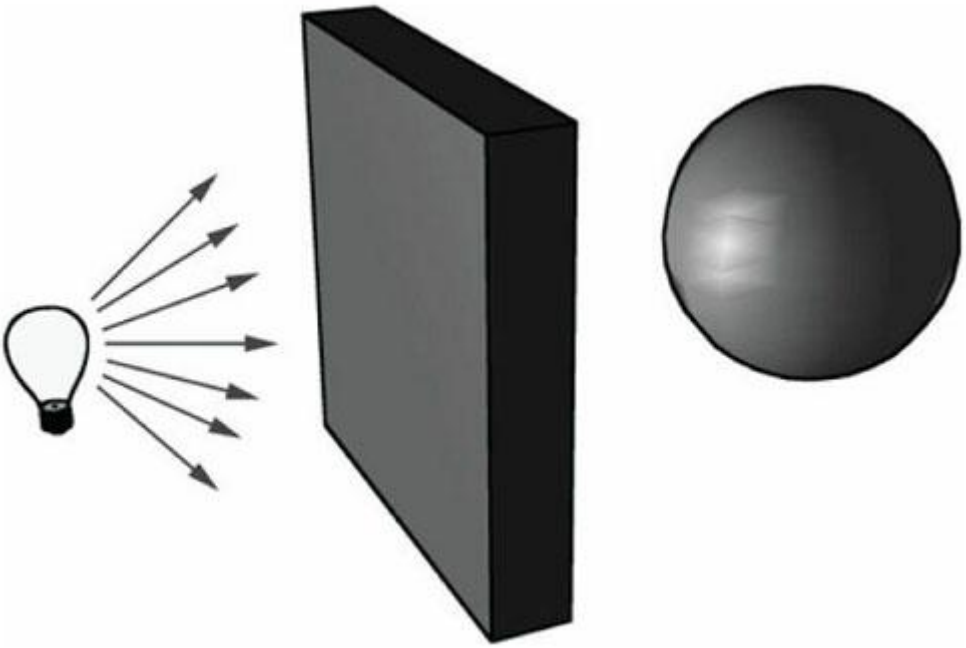


**图 7.3 在现实环境中，如果墙壁挡住了从灯泡射出的光线，那么球体就应该位于墙壁的阴影之中。然而，在局部光照模型中，无论墙壁是否存在，球体都会被照亮。**

与之相反，全局照明模型（global illumination model）不仅会考虑光源对物体的直接照明，也会考虑场景中的其他物体反射造成的间接照明。它之所以被称为全局照明模型，就是因为它在照亮一个物体的同时，也会考虑到整个场景中的其他因素。对于实时游戏来讲，全局照明模型占用的系统资源量过大（虽然它可以渲染出接近于照片的超写实场景）。目前，实时全局照明（real-time global illumination）的实现方法还处于探索阶段。

# 7.2 法线向量

平面法线（face normal）是描述多边形所朝方向的单位向量（即，它与多边形上的所有点相互垂直），如图 7.4a 所示。表面法线（surface normal）是与物体表面上的点的正切平面（tangent plane）相互垂直的单位向量，如图 7.4b 所示。表面法线确定了表面上的点“面对”的方向。

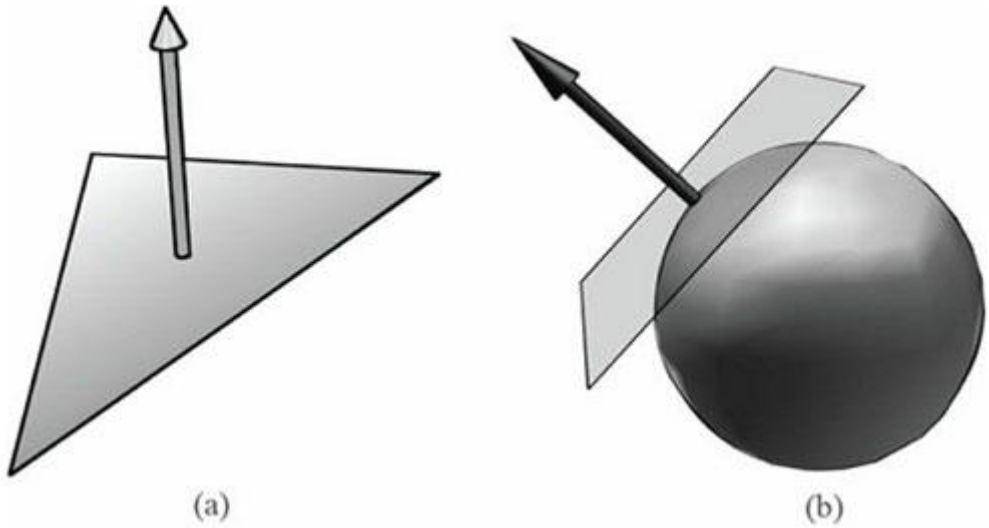



**图 7.4 (a)平面法线与平面上的所有点相互垂直。(b)表面法线与物体表面上的点的正切平面相互垂直。**

当进行光照计算时，我们必须为三角形网格表面上的每个点求解表面法线，以确定光线与网格表面在该点位置上的入射角度。为了获得表面法线，我们必须为每个顶点指定表面法线（这些法线称为顶点法线）。然后在光栅化阶段，这些顶点法线会在三角形表面上进行线性插值，使三角形表面上的每个点都获得一个表面法线（回顾 5.10.3 节并参见图 7.5）。

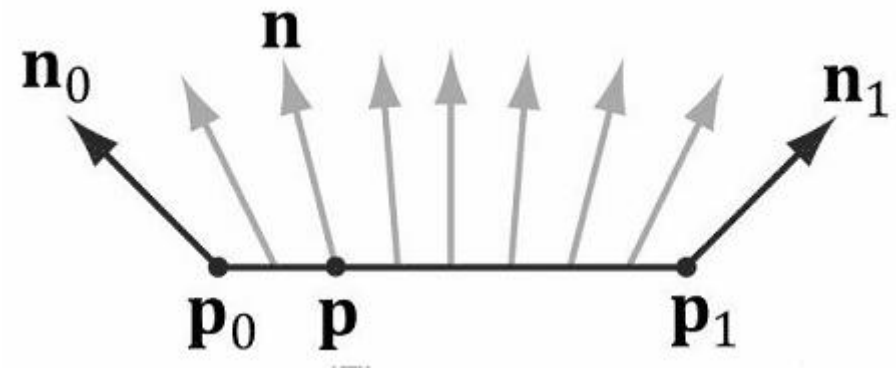



图 7.5 $\mathbf{p}_0$ 和 $\mathbf{p}_1$ 是线段的两个顶点，$\mathbf{n}_0$ 和 $\mathbf{n}_1$ 是对应的顶点法线。点 $\mathbf{p}$ 是通过线性插值（加权平均值）得到的线段上的一点，$\mathbf{n}$ 是点 $\mathbf{p}$ 的法线向量，它介于两个顶点法线之间。也就是，当存在一个位置使 $\mathbf{p} = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0)$ 时，$\mathbf{n} = \mathbf{n}_0 + t(\mathbf{n}_1 - \mathbf{n}_0)$。为了简单起见，我们只解释了线段的法线插值，但是一概念可以被直接扩展为 3D 三角形的法线插值。

**注意：**对每个像素的法线进行插值，并进行光照计算称为逐像素光照或 phong 光照。还有一种负担较轻，但不够精确的方法是对每个顶点进行光照运算，称为逐顶点光照，计算结果是从顶点着色器中输出的，在像素着色器中进行插值。将计算从像素着色器转移到顶点着色器是一种常见的性能优化措施，而且在很多情况下的视觉表现与逐像素光照差别不大。

## 7.2.1 计算法线向量

为了求解一个三角形 $\Delta \mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$ 的平面法线，我们必须先计算该三角形边上的两个向量：

$\mathbf{u} = \mathbf{p}_1 - \mathbf{p}_0$

$\mathbf{v} = \mathbf{p}_2 - \mathbf{p}_0$

然后求得平面法线为：

$$\mathbf{n} = \frac{\mathbf{u} \times \mathbf{v}}{\|\mathbf{u} \times \mathbf{v}\|}$$

下面的函数可以根据三角形的 3 个顶点来计算三角形正面（参见 5.10.2 节）的平面法线。

```
void ComputeNormal(const XMVector3& p0,
    const XMVector3& p1,
    const XMVector3& p2,
    XMVector3& out)
{
    XMVector3 u = p1 - p0;
    XMVector3 v = p2 - p0;
    XMVector3Cross(&out, &u, &v);
    XMVector3Normalize (&out, &out);
}
```

对于一个微分曲面来说，我们可以使用微积分计算曲面上的点的法线。但遗憾的是，三角形网格不是微分曲面。我们通常使用一种称为顶点法线平均值（vertex normal averaging）的技术求解三角形网格上的顶点法线。对于网格上的任意顶点 **v** 来说，**v** 的顶点法线 **n** 等于以 **v** 为共享顶点的每个多边形的平面法线的平均值。例如在图 7.6 中，网格上的四个多边形共享顶点 **v**；所以，**v** 的顶点法线为：

$$\mathbf{n}_{avg} = \frac{\mathbf{n}_0 + \mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3}{\|\mathbf{n}_0 + \mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3\|}$$

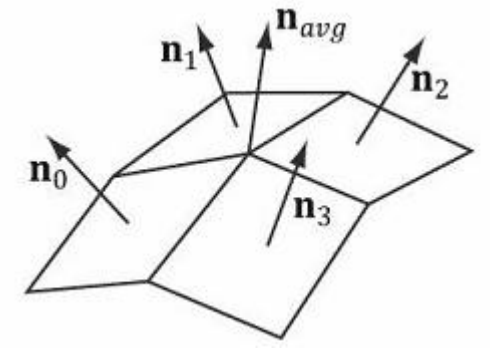



**图 7.6 中间的顶点由相邻的 4 个多边形共享，我们通过计算这 4 个多边形平面法线的平均值就可以估算出该顶点的法线。**

在上面的例子中，我们不需要除以 4，因为我们想要的是一个普通平均值，我们可以对结果进行规范化。注意，我们还可以构造更巧妙的平均值计算公式；例如，以每个多边形的面积作为权值，计算加权平均值（这样，面积较大的多边形会占有较大的权重，而面积较小的多边形会占有较小的权重）。

下面的伪代码说明了在给出一个三角形网格的顶点列表和索引列表时，如何计算该平均值：

```
// 输入：
// 1.一个顶点数组(mVertices)，每个顶点都有一个位置分量(pos)和
// 一个法线分量(normal)。
// 2.一个索引数组(mIndices)。
// 处理网格中的每个三角形：
for(DWORD i = 0; i < mNumTriangles; ++i)
{
    // 第 i 个三角形的索引
    UNIT i0 = mIndices[i*3+0];
    UNIT i1 = mIndices[i*3+1];
    UNIT i2 = mIndices[i*3+2];
    // 第 i 个三角形的顶点
    Vertex v0 = mVertices[i0];
    Vertex v1 = mVertices[i1];
    Vertex v2 = mVertices[i2];
    // 计算面法线
    Vector3 e0 = v1.pos - v0.pos;
    Vector3 e1 = v2.pos - v0.pos;
    Vector3 faceNormal Cross( &e0, &e1);
    // 这个三角形共享了以下三个顶点，
    // 所以要将面法线加入到这些顶点法线的平均值中。
    mVertices[i0].normal += faceNormal;
    mVertices[i1].normal += faceNormal;
    mVertices[i2].normal += faceNormal;
}
// 对每个顶点 v，我们已经将共享 v 的所有三角形的面法线相加了，
// 所以现在只需归一化即可。
for(UNIT i = 0; i < mNumVertices; ++i)
    mVertices[i].normal =  Normalize(&mVertices[i].normal));
```

## 7.2.2 对法线向量进行变换

在图 7.7a 中，正切向量 $\mathbf{u} = \mathbf{v}_1 - \mathbf{v}_0$ 垂直于法线向量 $\mathbf{n}$。当我们对这两个向量应用一个不成比例的缩放变换 $\mathbf{A}$ 时，我们可以从图 7.7b 中看到，变换之后的切线向量 $\mathbf{uA} = \mathbf{v}_1\mathbf{A} - \mathbf{v}_0\mathbf{A}$ 不再垂直于变换之后的法线向量 $\mathbf{nA}$。

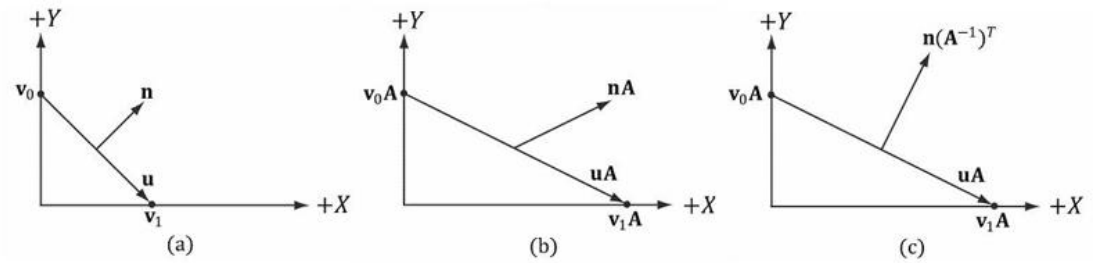



**图 7.7 (a)变换之前的表面法线。(b)当 *x* 轴上的单位长度增大两倍后，法线不再垂直于表面。(c)通过计算缩放变换的逆转置矩阵，我们可以得到正确的变换结果。**

所以我们的问题是：当给出一个用于变换点和（非法线）向量的变换矩阵 **A** 时，如何求出一个专门用来变换法线向量的变换矩阵 **B**，使变换之后的切线向量和法线向量依然保持垂直关系（即 **uA** • **uB** = 0）。要解决一问题，让我们先从一些已知条件开始：我们知道法线向量 **n** 直于切线向量 **u**。

| | |
|---|---|
| **u** • **v** = 0 | 切线向量垂直于法线向量 |
| $\mathbf{u}\mathbf{n}^T = 0$ | 用矩阵乘法来表示点积 |
| $\mathbf{u}(\mathbf{A}\mathbf{A}^{-1})\mathbf{n}^T = 0$ | 插入单位矩阵 $\mathbf{I} = \mathbf{A}\mathbf{A}^{-1}$ |
| $(\mathbf{u}\mathbf{A})(\mathbf{A}^{-1}\mathbf{n}^T) = 0$ | 矩阵乘法的结合性 |
| $(\mathbf{u}\mathbf{A})((\mathbf{A}^{-1}\mathbf{n}^T)^T)^T = 0$ | 转置特性$(\mathbf{A}^T)^T = \mathbf{A}$ |
| $(\mathbf{u}\mathbf{A})(\mathbf{n}(\mathbf{A}^{-1})^T)^T = 0$ | 转置特性$(\mathbf{A}\mathbf{B})^T = \mathbf{B}^T\mathbf{A}^T$ |
| $\mathbf{u}\mathbf{A} \cdot \mathbf{n}(\mathbf{A}^{-1})^T = 0$ | 用点积来表示矩阵乘法 |
| **uA** • **nB** = 0 | 变换之后的切线向量垂直于变换之后的法线向量 |

这样，使用 $\mathbf{B} = (\mathbf{A}^{-1})^T$（**A** 的逆转置矩阵）来变换法线向量，就可以使它与变换之后的切线向量依然保持垂直关系。

注意，当变换矩阵为正交矩阵（$\mathbf{A}^T = \mathbf{A}^{-1}$）时，$\mathbf{B} = (\mathbf{A}^{-1})^T = (\mathbf{A}^T)^T = \mathbf{A}$；也就是，我们不必计算逆转置矩阵，直接用 **A** 来代替 **B** 即可。总之，当以一个非等比变换矩阵对法线向量进行变换时，我们必须使用该矩阵的逆转置矩阵。

在 **MathHelper.h** 中有一个辅助方法用于计算逆转置矩阵：

```
static XMMATRIX InverseTranspose(CXMMATRIX M)
{
    // 逆转置矩阵仅用于法线。所以将矩阵中的平移行
    // 设置为 0，这样就不会影响逆转置矩阵的计算——因为我们
    // 不需要对平移进行逆转置运算。
    XMMATRIX A = M;
    A.r[3] = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);

    XMVECTOR det = XMMatrixDeterminant(A);
    return XMMatrixTranspose(XMMatrixInverse(&det, A));
}
```

因为逆转置只用于变换矢量，而平移是作用在点上的，因此需要从矩阵中排除平移因素。但是，3.2.1 节告诉我们，为了防止矢量被平移操作影响，它的 w 应设置为 0（使用齐次坐标）。因此，我们无须将矩阵中的平移行归零。但问题是，如果我们将逆转置矩阵和另一个不包含非等比例缩放的矩阵相乘，例如视矩阵$(\mathbf{A}^{-1})^T\mathbf{V}$，转置后位于$(\mathbf{A}^{-1})^T$ 第 4 列的平移项导致结果错误。所以，我们将平移项清零就是为了预防这个错误。正确的方法是使用$((\mathbf{AV})^{-1})^T$对法线进行变换。下面是一个缩放和平移矩阵的例子，第 4 列经过逆转置后并不是$[0,0,0,1]^T$：

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

$$(\mathbf{A}^{-1})^T = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 2 & 0 & -2 \\ 0 & 0 & 2 & -2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

**注意**：即使使用逆转置变换，法线向量也有可能会失去单位长度；所以，在变换之后必须重新规范化法线向量。

# 7.3 兰伯特余弦定理

垂直照向平面的线比从侧面照向平面的线更加强烈（参见图 7.8）。

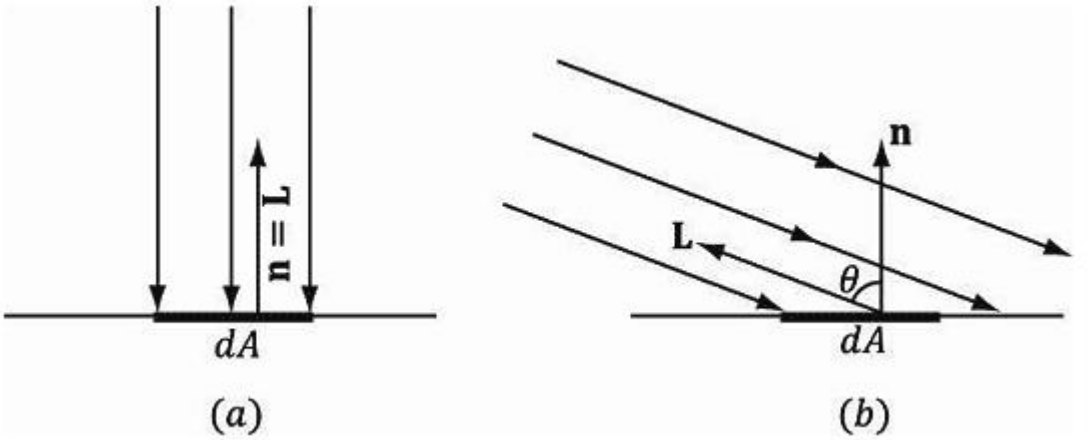



**图 7.8 假设有一块很小的区域 *dA*。当法线向量 n 与光照向量 L 平行时，区域 *dA* 受到的光线照射最多。随着 n 和 L 之间的夹角$\theta$逐渐增大，区域 *dA* 受到的光线照射量会越来越少（因为很多光线都无法照射到 *dA* 表面上了）。**

我们可以从这个概念中推导出一个函数，根据顶点法线和光照向量之间的夹角返回不同的光照强度。（注意，光照向量是从表面指向光源的向量；也就是，它与线的传播方向正好相反。）当顶点法线与照向量完全重叠时（即，它们的角度为 0º 时），该函数返回最大强度值；随着顶点法线与照向量之间的夹角逐渐增大，该函数返回的强度值会越来越小。当$\theta$>90 º 时，说明光线照射的是物体背面，此时我们应该将强度设置为 0。兰伯特（Lambert）余弦定理给出了上述函数的定义：

$$f(\theta) = \max(\cos\theta, 0) = \max(\mathbf{L} \cdot \mathbf{n}, 0)$$

其中，**L** 和 **n** 是单位向量。图 7.9 是 $f(\theta)$的曲线图。我们可以看到，随着$\theta$的变化，强度在 0.0 到 1.0（即，0%到 100%）之间变化。

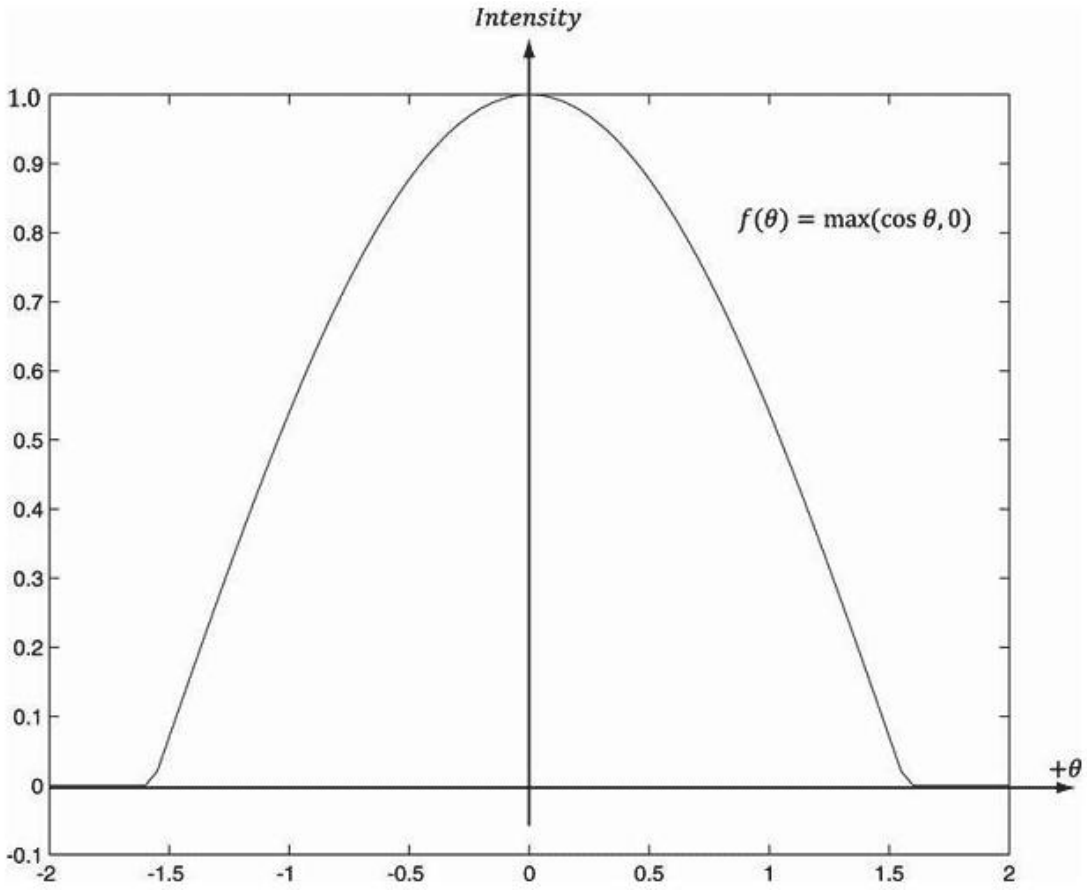



**图 7.9 当−2≤θ≤ 2 时，函数 $f(\theta) = \max(\cos\theta,0) = \max(L \cdot n,0)$的曲线图。注意，π/2≈1.57。**

# 1.2 长度和单位向量

在几何学中，向量的大小等于有向线段的长度。我们用双竖线来表示向量的大小（例如，‖**u**‖表示 **u** 的大小）。现在，给出一个向量 **u**=(*x*,*y*,*z*)，我们希望以代数方式计算它的大小。通过运用两次毕达哥拉斯定理可以计算出 3D 向量的大小（译者注：毕达哥拉斯定理和勾股定理的概念相同，换言之，毕达哥拉斯定理就是勾股定理）；参见图 1.8。

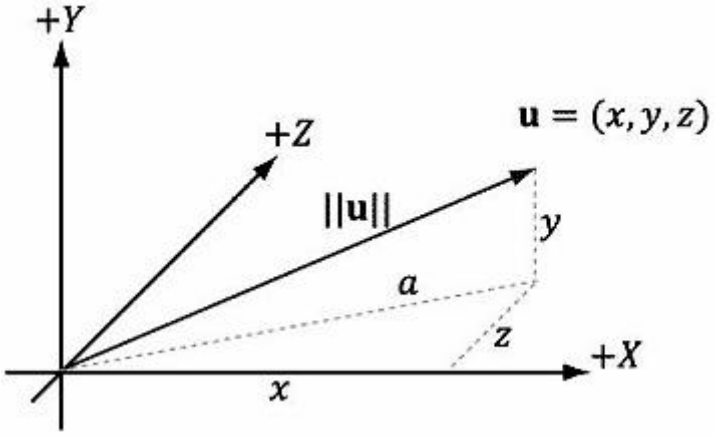



**图 1.8 通过运用两次毕达哥拉斯定理计算 3D 向量的长度。**

首先，我们来看 *xz* 平面上的三角形边 *x*、*z* 及斜边 *a*。由毕达哥拉斯定理可知 $a=\sqrt{x^2+z^2}$。现在来看三角形边 a、*y* 及斜边‖**u**‖。通过再次使用毕达哥拉斯定理，可以得到如下求模公式：

$$\|\mathbf{u}\|=\sqrt{y^2+a^2}=\sqrt{y^2+(\sqrt{x^2+z^2})^2}=\sqrt{x^2+y^2+z^2} \quad (1.1)$$

在某些应用中，我们不关心向量的长度， 只希望用向量来表示一个单纯的方向。对于这种只表示方向、不表示大小的向量，我们希望将其长度精确地设定为 1。当我们想要让一个向量具有单位长度时，我们说要对该向量进行规范化。我们将向量的每个分量除以该向量的模，得到规范化向量：

$$\hat{\mathbf{u}}=\frac{\mathbf{u}}{\|\mathbf{u}\|}=\left(\frac{x}{\|\mathbf{u}\|},\frac{y}{\|\mathbf{u}\|},\frac{z}{\|\mathbf{u}\|}\right) \quad (1.2)$$

要验证这个公式的正确性，只需计算 $\hat{\mathbf{u}}$ 的长度即可：

$$\|\hat{\mathbf{u}}\|=\sqrt{\left(\frac{x}{\|\mathbf{u}\|}\right)^2+\left(\frac{y}{\|\mathbf{u}\|}\right)^2+\left(\frac{z}{\|\mathbf{u}\|}\right)^2}=\frac{\sqrt{x^2+y^2+z^2}}{\sqrt{\|\mathbf{u}\|^2}}=\frac{\mathbf{u}}{\|\mathbf{u}\|}=1$$

因此，$\hat{\mathbf{u}}$ 确实是一个单位向量。

## 例 1.3

对向量 **v**= (−1,3,4) 进行规范化。我们计算 $\|\mathbf{v}\|=\sqrt{(-1)^2+3^2+4^2}=\sqrt{26}$ 。则，

$$\hat{\mathbf{v}}=\frac{\mathbf{v}}{\|\mathbf{v}\|}=\left(-\frac{1}{\sqrt{26}},\frac{3}{\sqrt{26}}\frac{4}{\sqrt{26}}\right)$$

要验证 $\hat{\mathbf{v}}$ 是否为单位向量，只需计算它的长度：

$$\|\hat{\mathbf{v}}\|=\sqrt{\left(-\frac{1}{\sqrt{26}}\right)^2+\left(\frac{3}{\sqrt{26}}\right)^2+\left(\frac{4}{\sqrt{26}}\right)^2}=\sqrt{\frac{1}{26}+\frac{9}{26}+\frac{16}{26}}=\sqrt{1}=1$$

# 1.3 点积

点积（dot product）是向量乘法的一种形式，它的计算结果是一个标量值；由于这一原因，有时也将点积称为标量积（scalar product）。设 **u**=（$u_x$,$u_y$,$u_z$），**v**=（$v_x$,$v_y$,$v_z$），则点积定义如下：

$$\mathbf{u}\cdot\mathbf{v}=u_xv_x+u_yv_y+u_zv_z \quad (1.3)$$

简言之，点积等于两个向量对应分量的乘积之和。

点积的定义不存在任何明显的几何含义。但是，使用余弦定理可以发现存在如下关系：

$$\mathbf{u}\cdot\mathbf{v}=\|\mathbf{u}\|\|\mathbf{v}\|\cos\theta \quad (1.4)$$

其中，$\theta$表示向量 **u** 和 **v** 之间的夹角，且 $0\leqslant\theta\leqslant\pi$（参见图 1.9）。公式 1.4 说明这两个向量的点积等于向量夹角的余弦值和向量模之间的乘积。在特殊情况下，如果 **u** 和 **v** 都是单位向量，那么 **u**·**v** 就等于它们之间夹角的余弦值（即，**u**·**v**=cos$\theta$）。

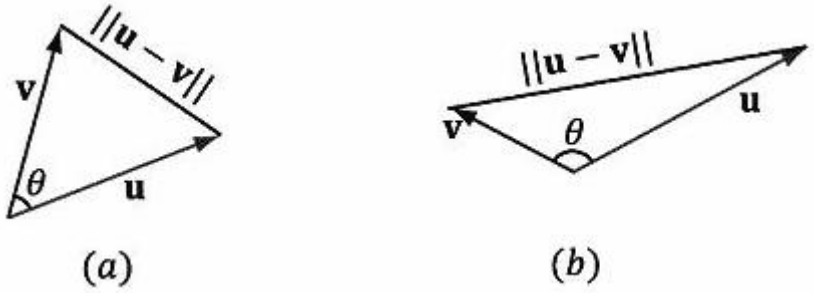



**图 1.9 在左图中，u、v 之间的夹角$\theta$为锐角。在右图中，u、v 之间的夹角$\theta$为钝角。记住，当我们提及两个向量之间的夹角时，通常指的是最小的角，也就是角度$\theta$，且 $0\leqslant\theta\leqslant\pi$。**

公式 1.4 提供了一些有用的点积的几何性质：

1．如果 **u**·**v**=0，则 **u**⊥**v**（即，向量相互垂直）。

2．如果 **u**·**v**＞0，则两个向量之间的夹角$\theta$小于 90º（即，向量形成一个锐角）。

3．如果 **u**·**v**＜0，则两个向量之间的夹角$\theta$大于 90º（即，向量形成一个钝角）。

**注意**：“相互垂直”也可称为“互成直角”。

**【例 1.4】**

设 **u**=（1, 2,3）、**v**=（−4, 0, −1）。求 **u** 和 **v** 之间的夹角。首先，我们要做如下计算：

$$\mathbf{u}\cdot\mathbf{v}=(1,2,3)\cdot(-4,0,-1)=-4-3=-7$$
$$\|\mathbf{u}\|=\sqrt{1^2+2^2+3^3}=\sqrt{14}$$
$$\|\mathbf{v}\|=\sqrt{(-4)^2+0^2+(-1)^2}=\sqrt{17}$$

现在，由公式 1.4 解得$\theta$为：

$$\cos\theta=\frac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|}=\frac{-7}{\sqrt{14}\sqrt{17}}$$
$$\theta=\cos^{-1}\frac{-7}{\sqrt{14}\sqrt{17}}\approx 117°$$

**【例 1.5】**

考虑图 1.10。给出 **v** 和单位向量 **n**，推导出一个使用点积求解向量 **p** 的公式。

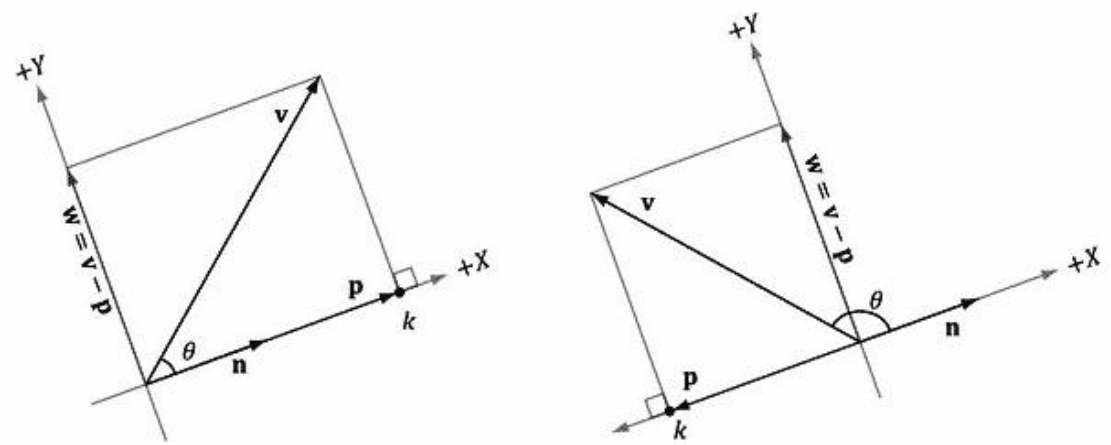



**图 1.10 v 在 n 上的正交投影。**

首先，从该图中可以看到标量 *k* 可以使 **p**=*k***n**；而且，由于我们已知 ‖ **n** ‖=1，所以有 ‖ **p** ‖ = ‖ *k***n** ‖ =| *k*| ‖ **n** ‖ =|*k*|。（注意，当且仅当 **p** 与 **n** 的方向相反时，*k* 为负数。）我们使用三角函数，可以得出 *k*= ‖ **v** ‖ cos$\theta$；由此，**p**=*k***n**=（ ‖ **v** ‖ cos$\theta$）**n**。不过，因为 **n** 是一个单位向量，所以我们可以用另一种方式进行表达：

$$\mathbf{p}=(\|\mathbf{v}\|\cos\theta)\mathbf{n}=(\|\mathbf{v}\|\cdot 1\cos\theta)\mathbf{n}=(\|\mathbf{v}\|\cdot\|\mathbf{n}\|\cos\theta)\mathbf{n}=(\mathbf{v}\cdot\mathbf{n})\mathbf{n}$$

请注意，这里的 *k*=**v**·**n**，它说明了当 **n** 为单位向量时 **v**·**n** 的几何含义。我们将 **p** 称为 **v** 在 **n** 上的正交投影（orthogonal projection），并记为

$$\mathbf{p}=proj_{\mathbf{n}}(\mathbf{v})$$

如果我们把 **v** 理解为一个作用力，那么 **p** 可以被认为是 **v** 在方向 **n** 上的分力。同理，向量 **w**=perp$_{\mathbf{n}}$(**v**)=**v**−**p** 是与 **n** 垂直方向上的分力。可以看到 **v**=**p**+**w**，这说明我们已经将向量 **v** 分解成了两个相互垂直的向量 **p** 和 **w**。

如果 n 不是一个单位向量，那我们可以对它进行规范化，使其保持单位长度。通过用单位向量 $\frac{\mathbf{n}}{\|\mathbf{n}\|}$ 来代替 **n**，可以得到一个更通用的投影公式：

$$\mathbf{p} = proj_{\mathbf{n}}(\mathbf{v}) = (\mathbf{v} \cdot \frac{\mathbf{n}}{\|\mathbf{n}\|}) \frac{\mathbf{n}}{\|\mathbf{n}\|} = \frac{(\mathbf{v} \cdot \mathbf{n})}{\|\mathbf{n}\|^2} \mathbf{n}$$

### 1.3.1 正交化

若一个向量集合{$\mathbf{v}_0$，……，$\mathbf{v}_{n-1}$}中的向量相互正交（即集合中的每一个向量与其他向量正交）并具有单位长度，我们将这个集合称之为规范化正交集。有时一组向量几乎是正交的，但又不完全是，一个常见的任务就是使其正交。在三维计算机图形中，开始时通常是一个规范化正交的向量集合，但由于计算精度问题，这个集合就会逐渐成为非规范化的了。我们主要关心的是 2D 和 3D 的情况下任何处理这个问题（即，含有两个和三个向量的情况）。

首先讨论简单的 2D 情况。假设有一组向量为{$\mathbf{v}_0$,$\mathbf{v}_1$}，我们要将它们正交到一个规范正交集{$\mathbf{w}_0$,$\mathbf{w}_1$}中，如图 1.11 所示。首先令 $\mathbf{w}_0$=$\mathbf{v}_0$，然后修改 $\mathbf{v}_1$ 使它与 $\mathbf{w}_0$ 垂直；这需要减去 $\mathbf{v}_1$ 向量在 $\mathbf{w}_0$ 上的投影：

$\mathbf{w}_1 = \mathbf{v}_1 - \text{proj}_{\mathbf{w}0}(\mathbf{v}_1)$

现在就有了一组互相垂直的向量{$\mathbf{w}_0$,$\mathbf{w}_1$}；最后需要规范化 $\mathbf{w}_0$ 和 $\mathbf{w}_1$ 才能构建规范化的正交集。

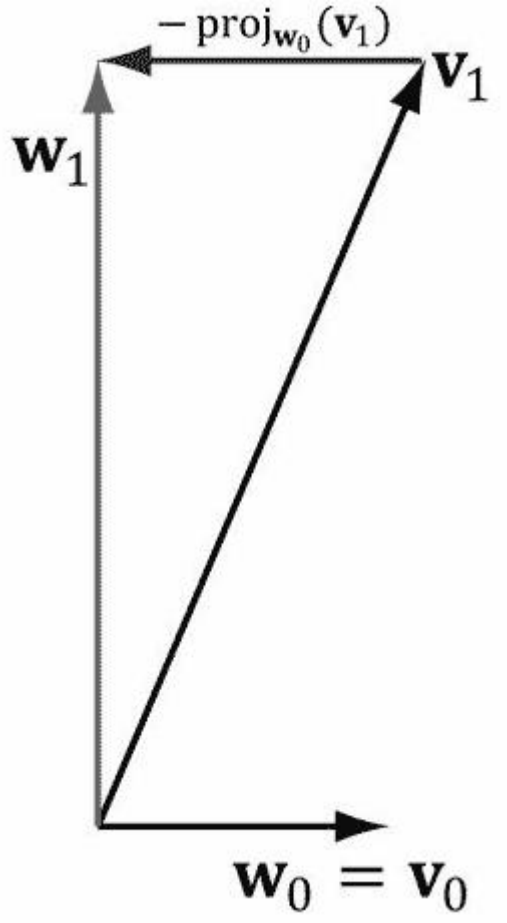



**图 1.11 2D 正交化**

处理 3D 情况的原理与 2D 相同，但需要更多的步骤。假设有一组向量{$\mathbf{v}_0$,$\mathbf{v}_1$,$\mathbf{v}_2$}需要正交规范化到{$\mathbf{w}_0$,$\mathbf{w}_1$,$\mathbf{w}_2$}，如图 1.12 所示。首先令 $\mathbf{w}_0$=$\mathbf{v}_0$，然后修改 $\mathbf{v}_1$ 使之垂直于 $\mathbf{w}_0$；这需要从 $\mathbf{v}_1$ 中减去 $\mathbf{v}_1$ 在 $\mathbf{w}_0$ 方向上的投影：

$\mathbf{w}_1 = \mathbf{v}_1 - \text{proj}_{\mathbf{w}0}(\mathbf{v}_1)$

下一步需要令 $\mathbf{v}_2$ 同时垂直于 $\mathbf{w}_0$ 和 $\mathbf{w}_1$，这需要从 $\mathbf{v}_2$ 中减去 $\mathbf{v}_2$ 在 $\mathbf{w}_0$ 上的投影再减去 $\mathbf{v}_2$ 在 $\mathbf{w}_1$ 上的投影：

$\mathbf{w}_2 = \mathbf{v}_2 - \text{proj}_{\mathbf{w}0}(\mathbf{v}_2) - \text{proj}_{\mathbf{w}1}(\mathbf{v}_2)$

现在就有了一组互相垂直的向量$\{\mathbf{w}_0,\mathbf{w}_1,\mathbf{w}_2\}$；最后需要规范化 $\mathbf{w}_0$、$\mathbf{w}_1$ 和 $\mathbf{w}_2$ 才能构建规范化的正交集。

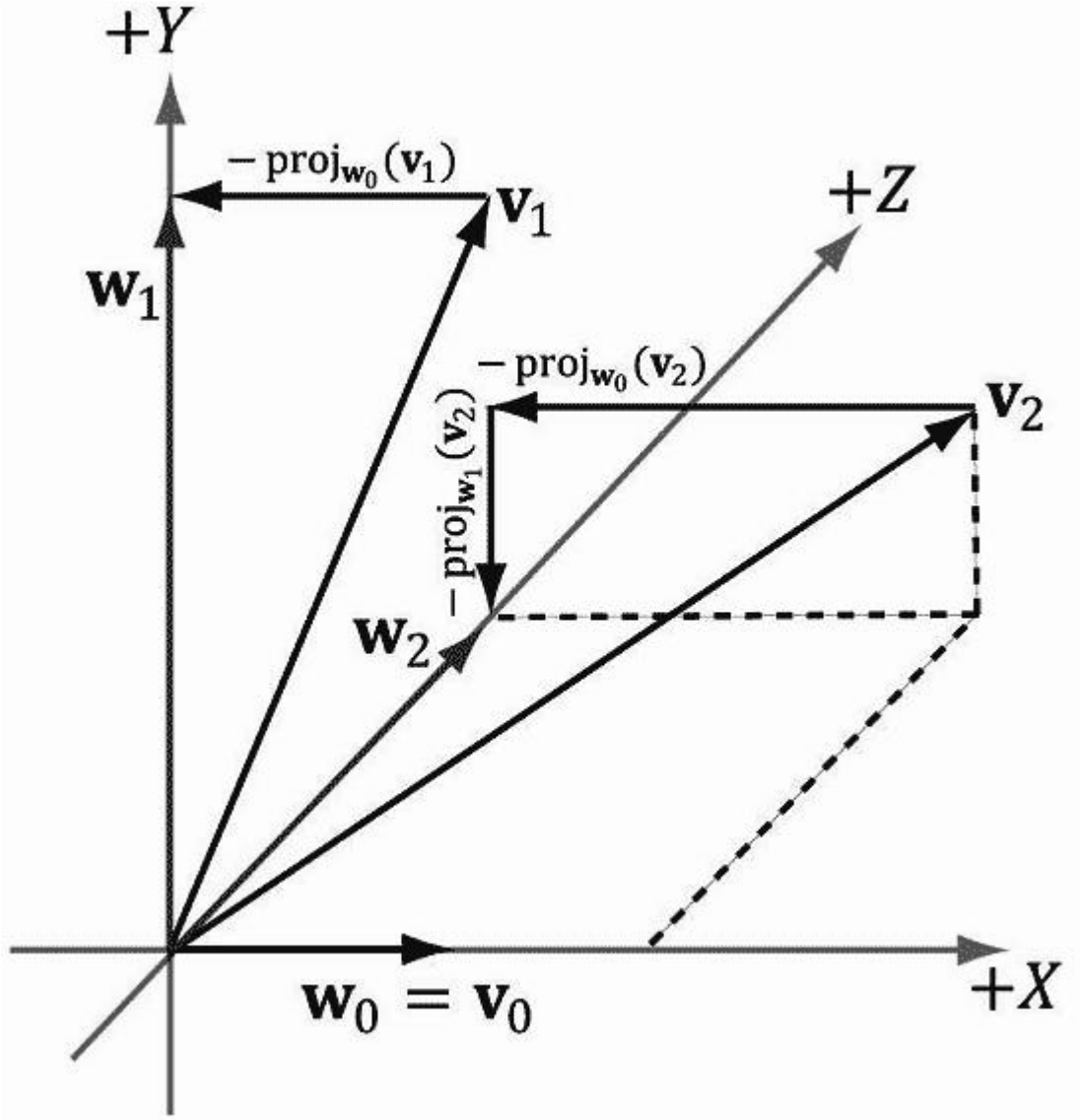



**图 1.12 3D 正交化**

要规范正交化任意数量的向量集$\{\mathbf{v}_0,\ldots,\mathbf{v}_{n-1}\}$，我们需要按照通常叫做 Gram-Schmidt 正交化的处理过程进行：

基本步骤：令 $\mathbf{w}_0=\mathbf{v}_0$

for $1 \le i \le n\text{-}1$，令 $\mathbf{w}_i = \mathbf{v}_i - \sum_{j=0}^{i-1} proj_{\mathbf{wj}}(\mathbf{v}_i)$

规范化步骤：令 $\mathbf{w}_i = \frac{\mathbf{w}_i}{\|\mathbf{w}_i\|}$

原理与上面是类似的，当选取一个向量 $\mathbf{v}_i$ 将它添加到规范化的正交集时，我们需要减去这个向量在正交集中其他向量（$\mathbf{w}_0,\mathbf{w}_1,\ldots,\mathbf{w}_{i-1}$）上的投影，这样可以确保这个新添的向量与正交集中的其他向量垂直。

# 1.4 叉积

叉积（cross product）是向量数学定义的第二种乘法形式。它与点积不同，点积的计算结果是一个标量，而叉积的计算结果是一个向量；另外，叉积只能用于 3D 向量（2D 向量没有叉积）。通过对两个 3D 向量 **u** 和 **v** 计算叉积，可以得到第 3 个向量 **w**，该向量同时垂直于 **u** 和 **v**。也就是说，**w** 即垂直于 **u**，**w** 也垂直于 **v**（参见图 1.13）。设 **u**=（$u_x,u_y,u_z$），**v**=（$v_x,v_y,v_z$），则叉积为：

$$\mathbf{w}=\mathbf{u}\times\mathbf{v}=(u_yv_z-u_zv_y,u_zv_x-u_xv_z,u_xv_y-u_yv_x) \qquad (1.5)$$

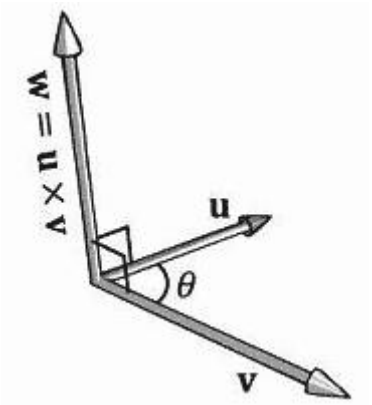



**图 1.13 通过为两个 3D 向量 u 和 v 计算叉积，可以得到第 3 个向量 w，该向量同时垂直于 u 和 v。如果读者抬起左手，将拇指之外的其他 4 个手指指向第一个向量 u 的方向，然后朝着 v 的方向沿角度 0≤$\theta$≤$\pi$弯曲手指，此时拇指所指的方向即为 w=u×v 的方向；这叫做左手拇指定则（left-hand-thumb rule）。**

注意：如果你处理的是一个右手坐标系，则需要使用右手拇指定则（right-hand-thumb rule）：如果抬起右手，将拇指之外的其他 4 个手指指向第一个向量 **u** 的方向，然后朝着 **v** 的方向沿角度 0≤$\theta$≤$\pi$弯曲手指，此时拇指所指的方向即为 **w=u×v** 的方向。

【例 1.6】

设 u= (2, 1,3)、v= (2, 0,0)。计算 **w=u×v** 和 **z=v×u**，并验证 **w** 既垂直于 **u**，也垂直于 **v**。由公式 1.5 可得，

**w=u×v**

= (2, 1,3)×(2, 0,0)

= (1 · 0 − 3 · 0,3 · 2 − 2 · 0,2 · 0 − 1 · 2)

= (0, 6, −2)

和

**z=v×u**

= (2, 0,0) × (2, 1,3)

= (0 · 3 − 0 · 1,0 · 2 − 2 · 3,2 · 1 − 0 · 2)

= (0, −6,2)

该结果说明 **u×v≠v×u**。也就是，叉积不支持交换律。实际上，它可以表达为 **u×v= −v**

×**u**。读者可以通过左手拇指定则来判断由这个叉积得出的向量。如果你从第 1 个向量朝着第 2 个向量的方向卷曲手指时（通常选择角度最小的路径），你的拇指会指向最终得到的向量的方向，如图 1.13 所示。

为了说明 **w** 既垂直于 **u**，也垂直于 **v**，我们回顾 1.3 节的内容：如果 **u**·**v**=0，则 **u**⊥**v**（即，向量相互垂直）。因为

**w**·**u**= (0, 6, −2) · (2, 1,3) =0 · 2 +6 · 1+ (−2) · 3 =0

和

**w**·**v**= (0, 6, −2) · (2, 0,0) =0 · 2 +6 · 0+ (−2) · 0 =0

我们得出结论：**w** 既垂直于 **u**，也垂直于 **v**。

## 1.4.1 2D 伪叉积

叉积可以计算垂直于给定两个 3D 向量的向量。在 2D 的情况中，并不存在这种情况，但我们常常要求出垂直于给定 2D 向量 **u** = ($u_x$,$u_y$)的向量 **v**。图 1.14 展示了这种操作的几何图景，从图中可以看出 **v** = (-$u_y$, $u_x$)。数学证明很简单：

**u**·**v**= ($u_x$,$u_y$)·(-$u_y$,$u_x$)= -$u_x u_y$+ $u_y u_x$= 0

所以 **u**⊥**v**。而 **u**・-**v**=$u_x u_y$+ $u_y$(-$u_x$) =0，也为零，所以还能得出结论：**u**⊥-**v**。

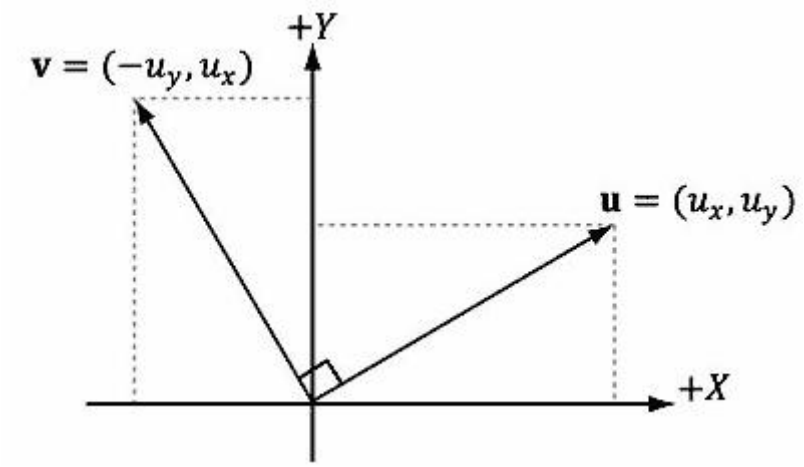



**图 1.14 u 向量的 2D 伪叉积为一个垂直于它的向量 v**

## 1.4.2 使用叉积进行正交规范化

在 1.3.1 节中，我们介绍了一种正交化一组向量的处理方法。对 3D 的情况来说，还可以使用叉积对一组向量（这些向量近似正交，但由于数值累积精度的误差，会变得不再正交）进行正交规范化操作。可参见图 1.15 理解这个过程的几何图景：

1．令 $\mathbf{w}_0 = \dfrac{\mathbf{v}_0}{\|\mathbf{v}_0\|}$

2．令 $\mathbf{w}_2 = \dfrac{\mathbf{w}_0 \times \mathbf{v}_1}{\|\mathbf{w}_0 \times \mathbf{v}_1\|}$

3．令 $\mathbf{w}_1 = \mathbf{w}_2 \times \mathbf{w}_0$。由后面的练习 14 可知，因为 $\mathbf{w}_2 \perp \mathbf{w}_0$ 且$\|\mathbf{w}_2\|=\|\mathbf{w}_0\|=1$，所以$\|\mathbf{w}_2 \times \mathbf{w}_0\|=1$，这样我们就无需进行规范化操作了。

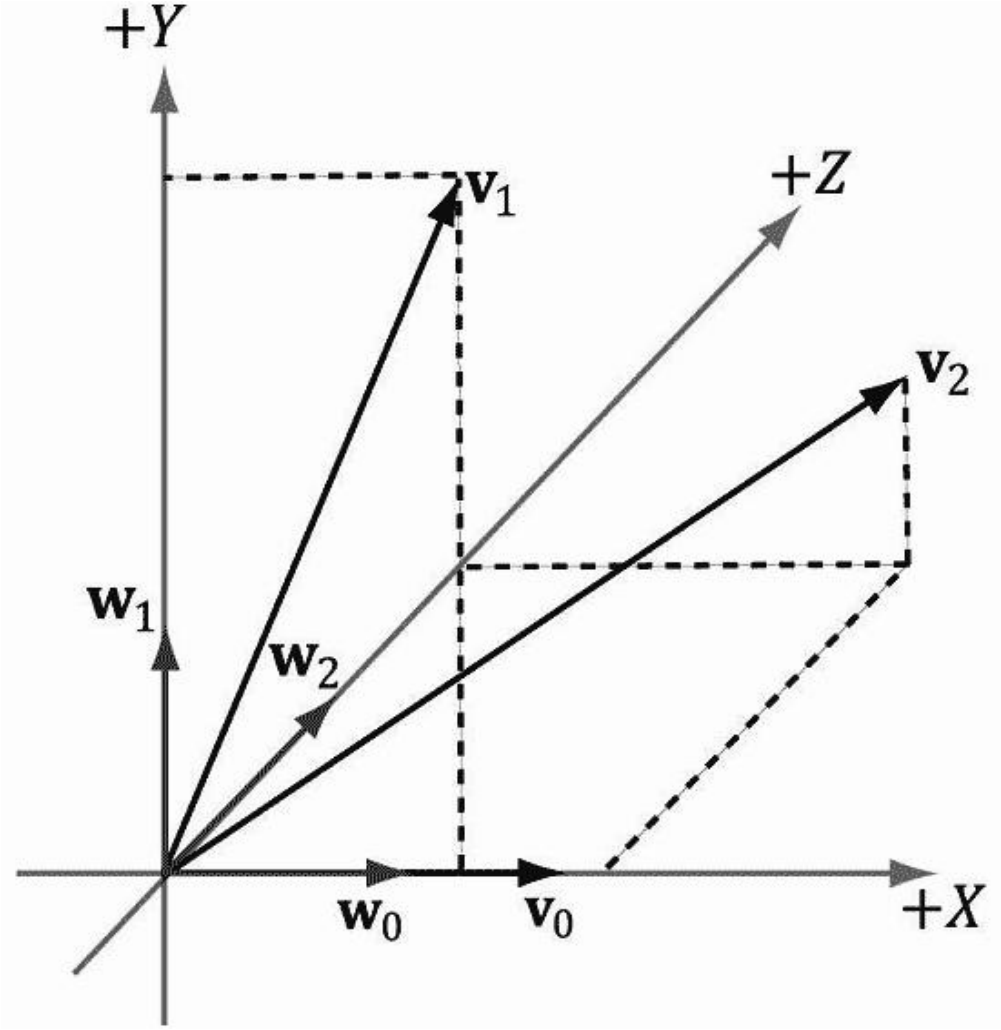



**图 1.15 使用叉积进行 3D 正交化**

至此，完成了向量集$\{\mathbf{w}_0,\mathbf{w}_1,\mathbf{w}_2\}$的正交规范化处理。

**注意**：在前面的示例中，我们首先令 $\mathbf{w}_0 = \dfrac{\mathbf{v}_0}{\|\mathbf{v}_0\|}$，表示从 $\mathbf{v}_0$ 变化到 $\mathbf{w}_0$ 并没有改变向量的方向，只是改变了大小。但是，$\mathbf{w}_1$ 和 $\mathbf{w}_2$ 的方向与 $\mathbf{v}_1$ 和 $\mathbf{v}_2$ 的方向并不相同。根据应用程序的需要，选择哪个向量不改变方向可能会很重要。例如，本书的后面我们将会使用三个正交向量$\{\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2\}$代表相机的朝向，其中第三个向量 $\mathbf{v}_2$ 表示相机的观察方向。当正交规范化这三个向量时，我们常常不想改变观察的方向，因此，我们会首先使用上面的算法处理 $\mathbf{v}_2$，然后修改 $\mathbf{v}_0$ 和 $\mathbf{v}_1$ 生成正交向量。

# 1.5 点

到目前为止，我们讨论的是与位置无关的向量。而我们在 3D 程序中需要描述坐标位置；比如，3D 几何体的位置和 3D 虚拟摄像机的位置。相对于一个坐标系，我们可以使用

在标准位置上的向量（参见图 1.16）来表示空间中的 3D 位置；我们将它称为位置向量（position vector）。在这里，向量末端的位置是唯一需要关注的特性，而方向和大小都无关紧要。我们会交替使用术语“位置向量”和“点”，因为位置向量表示的就是一个点。

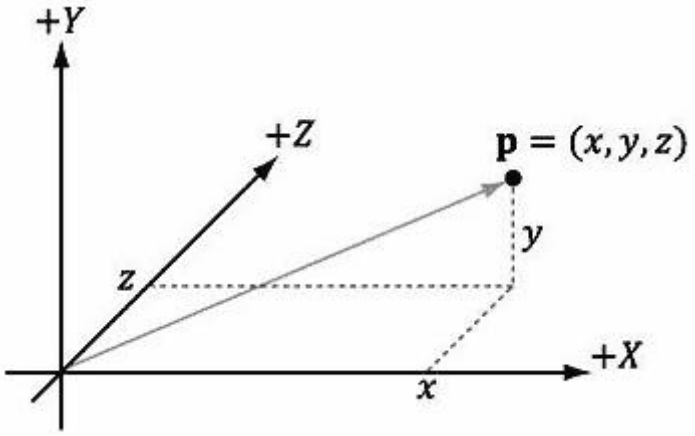



**图 1.16 从原点延伸到点的位置向量，它足以描述相对于坐标系的点的位置。**

使用向量来表示点的一个好处是可以在代码中使用向量运算，虽然向量运算对点来说没有实际意义；比如，在几何学中，两点相加有什么意义？不过有一些运算确实可以被扩展为点运算。比如，两点之差 **q−p** 可以表示从 **p** 到 **q** 的向量。点 **p** 与向量 **v** 相加得到点 **q**，可以认为 **q** 是 **v** 对 **p** 进行的平移。由于使用向量可以很方便地表示相对于坐标系的点，所以我们不必为单独设计一套针对于点的运算，只需要借助于前面讨论过的向量代数框架就可以处理它们（参见图 1.17）。

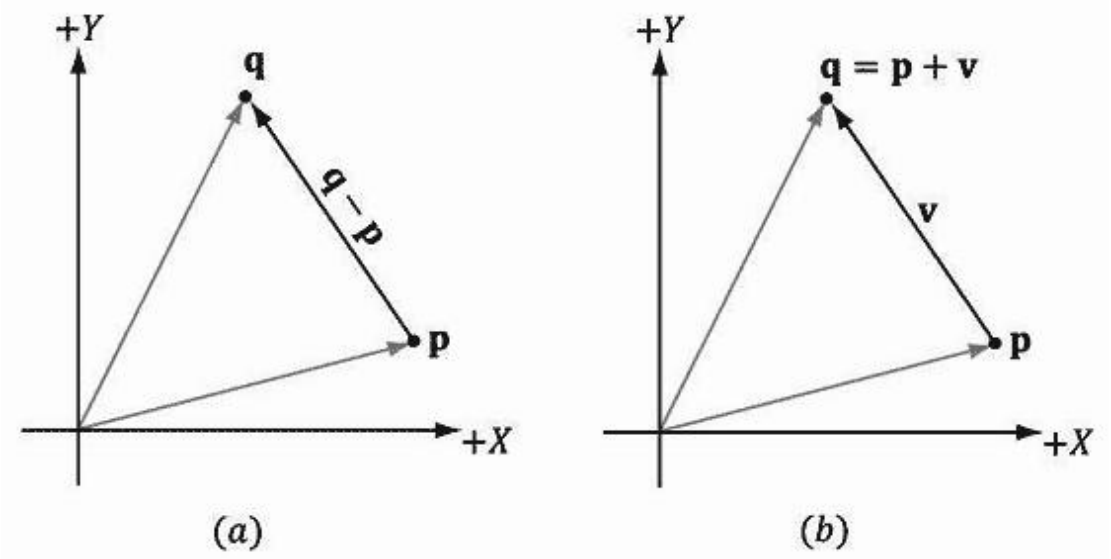



**图 1.17 （a）两点之差 q−p 可以表示从 p 到 q 的向量。（b）点 p 与向量 v 相加得到点 q，可以认为 q 是 v 对 p 进行的平移。**

**注意**：其实在几何学中有一种非常重要的方法叫做仿射组合（affine combination），它用于对点进行特殊的求和运算，就像是对点求加权平均值一样。

# 2.1 定义

在 3D 计算机绘图中，我们用矩阵（matrix）来紧凑地描述几何变换，比如缩放、旋转

和平移，并将点或向量的坐标从一种坐标系转换到另一种坐标系。本章探讨了矩阵代数。

**学习目标：**

1．了解矩阵及矩阵运算。

2．了解如何将向量-矩阵乘法视为一个线性组合。

3．学习单位矩阵、转置矩阵、行列式和逆矩阵。

4．熟悉 XNA 库中的用于矩阵代数的类和函数。

一个 $m\times$n 矩阵 **M** 是一个 $m$ 行、$n$ 列的矩形实数数组。行和列的数量指定了矩阵的维数。矩阵中的数值称为元素。我们使用行和列组成的双下标 $M_{ij}$ 来标识矩阵元素，其中，第 1 个下标指定了元素的所在的行，第 2 个下标指定了元素所在的列。

# 例 2.1

考虑如下矩阵：

$$A=\begin{bmatrix}3.5 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0.5 & 0\\ 2 & -5 & \sqrt{2} & 1\end{bmatrix}\quad \mathbf{B}=\begin{bmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\\ B_{31} & B_{32}\end{bmatrix}\quad \mathbf{u}=\begin{bmatrix}u_1,u_2,u_3\end{bmatrix}\quad \mathbf{v}=\begin{bmatrix}1\\ 2\\ \sqrt{3}\\ \pi\end{bmatrix}$$

1．矩阵 **A** 是一个 4×4 矩阵；矩阵 **B** 是一个 3×2 矩阵；矩阵 **u** 是一个 1×3 矩阵；矩阵 **v** 是一个 4×1 矩阵。

2．$A_{42}$=-5 表示矩阵 **A** 的第 4 行、第 2 列的元素。$B_{21}$ 表示矩阵 **B** 的第 2 行、第 1 列的元素。

3．矩阵 **u** 和 **v** 是特殊矩阵，因为它们只包含一行或一列。我们有时将这种矩阵称为行向量或列向量，因为它们可以用矩阵的形式表示一个向量（例如，可以随意地将$(x,y,z)$和$[x,y,z]$互换使用，这两种记法都可用于表示向量）。注意，对于行向量和列向量，不必使用双下标表示矩阵元素——只用一个下标即可。

有时，我们希望将矩阵的每一行视为一个向量。例如，我们可以这样写：

$$\begin{bmatrix}A_{11} & A_{12} & A_{13}\\ A_{21} & A_{22} & A_{23}\\ A_{31} & A_{32} & A_{33}\end{bmatrix}=\begin{bmatrix}\leftarrow \mathbf{A}_{1,*} \rightarrow\\ \leftarrow \mathbf{A}_{2,*} \rightarrow\\ \leftarrow \mathbf{A}_{3,*} \rightarrow\end{bmatrix}$$

其中，$\mathbf{A}_{1,*}=[A_{11},A_{12},A_{13}]$、$\mathbf{A}_{2,*}=[A_{21},A_{22},A_{23}]$、$\mathbf{A}_{3,*}=[A_{31},A_{32},A_{33}]$。在这种记法中，第 1 个索引指定行标，第 2 个索引以星号（*）表示我们引用的是整个行向量。同样，我们也可以用这种方法来表示矩阵的列：

$$\begin{bmatrix}A_{11} & A_{12} & A_{13}\\ A_{21} & A_{22} & A_{23}\\ A_{31} & A_{32} & A_{33}\end{bmatrix}=\begin{bmatrix}\uparrow & \uparrow & \uparrow\\ A_{*,1} & A_{*,2} & A_{*,3}\\ \downarrow & \downarrow & \downarrow\end{bmatrix}$$

其中

$$\mathbf{A}_{*,1}=\begin{bmatrix}A_{11}\\ A_{21}\\ A_{31}\end{bmatrix},\mathbf{A}_{*,2}=\begin{bmatrix}A_{12}\\ A_{22}\\ A_{32}\end{bmatrix},\mathbf{A}_{*,3}=\begin{bmatrix}A_{13}\\ A_{23}\\ A_{33}\end{bmatrix}$$

在这种记法中，第 2 个索引指定列标，第 1 个索引以星号（*）表示我们引用的是整个列向量。

我们现在对矩阵上的判等运算、加法运算、标量乘法运算和减法运算做以定义：

1．当且仅当两个矩阵的对应元素相等时，这两个矩阵相等；这两个矩阵必须具有相同的行数和列数，才能进行比较。

2．矩阵加法是对两个矩阵的对应元素相加；只有在两个矩阵的行数和列数相同的情况下，矩阵加法才有意义。

3．矩阵的标量乘法是将一个标量和矩阵中的每个元素相乘。

4．矩阵减法可以由矩阵加法和标量乘法表示。也就是，$\mathbf{A}-\mathbf{B}=\mathbf{A}+(-1\cdot\mathbf{B})=\mathbf{A}+(-\mathbf{B})$。

## 例 2.2

设

$$\mathbf{A}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix},\mathbf{B}=\begin{bmatrix}6 & 2\\ 5 & -8\end{bmatrix},\mathbf{C}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix},\mathbf{D}=\begin{bmatrix}2 & 1 & -3\\ -6 & 3 & 0\end{bmatrix}$$

则，

（i）$\mathbf{A}+\mathbf{B}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix}+\begin{bmatrix}6 & 2\\ 5 & -8\end{bmatrix}=\begin{bmatrix}1+6 & 5+2\\ -2+5 & 3+(-8)\end{bmatrix}=\begin{bmatrix}7 & 7\\ 3 & -5\end{bmatrix}$

（ii）$\mathbf{A}=\mathbf{C}$

（iii）$3\mathbf{D}=3\begin{bmatrix}2 & 1 & -3\\ -6 & 3 & 0\end{bmatrix}=\begin{bmatrix}3(2) & 3(1) & 3(-3)\\ 3(-6) & 3(3) & 3(0)\end{bmatrix}=\begin{bmatrix}6 & 3 & -9\\ -18 & 9 & 0\end{bmatrix}$

（iv）$\mathbf{A}-\mathbf{B}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix}-\begin{bmatrix}6 & 2\\ 5 & -8\end{bmatrix}=\begin{bmatrix}1-6 & 5-2\\ -2-5 & 3-(-8)\end{bmatrix}=\begin{bmatrix}-5 & 3\\ -7 & 11\end{bmatrix}$

因为矩阵加法和标量乘法是逐元素进行的，所以矩阵也继承了实数的加法和标量乘法的性质：

1．$\mathbf{A}+\mathbf{B}=\mathbf{B}+\mathbf{A}$

2．$(\mathbf{A}+\mathbf{B})+\mathbf{C}=\mathbf{A}+(\mathbf{B}+\mathbf{C})$

3．$r(\mathbf{A}+\mathbf{B})=r\mathbf{A}+r\mathbf{B}$

4．$(r+s)\mathbf{A}=r\mathbf{A}+s\mathbf{A}$

# 2.2 矩阵乘法

本节讲解矩阵乘法。我们将在第 3 章中看到，矩阵乘法用于实现点和向量的变换，并通过矩阵乘法将一系列的变换组合在一起。

## 2.2.1 定义

假设 **A** 是一个 $m \times n$ 矩阵，**B** 是一个 $n \times p$ 矩阵，乘积 **AB** 由 **C** 表示，则 **C** 是一个 $m \times p$ 矩阵，其中结果 **C** 的第 $ij$ 个元素的值等于 **A** 的第 $i$ 个行向量和 **B** 的第 $j$ 个列向量的点积，也就是，

$$\mathbf{C}_{ij}=\mathbf{A}_{i,*} \cdot \mathbf{B}_{*,j} \qquad (2.1)$$

注意，矩阵 **A** 的列数必须与矩阵 **B** 的行数相同，只有这样才能计算矩阵乘积 **C**，也就是说，**A** 中的行向量的维数必须与 **B** 中的列向量的维数相同。如果维数不同，那么公式 2.1 中的点积就没有意义。

## 例 2.3

设

$$\mathbf{A}=\begin{bmatrix} 1 & 5 \\ -2 & 3 \end{bmatrix}, \mathbf{B}=\begin{bmatrix} 2 & -6 \\ 1 & 3 \\ -3 & 0 \end{bmatrix}$$

则，无法计算乘积 **AB**，因为 **A** 中的行向量的维数是 2，而 **B** 中的列向量的维数是 3。我们无法计算 **A** 的第 1 个行向量与 **B** 的第 1 个列向量的点积，因为一个 2D 向量是无法与一个 3D 向量计算点积的。

## 例 2.4

设

$$\mathbf{A}=\begin{bmatrix} -1 & 5 & -4 \\ 3 & 2 & 1 \end{bmatrix}, \mathbf{B}=\begin{bmatrix} 2 & 1 & 0 \\ 0 & -2 & 1 \\ -1 & 2 & 3 \end{bmatrix}$$

首先我们要指出的是可以计算乘积 AB（是一个 2×3 矩阵），因为 A 的列数与 B 的行数相等。运用公式 2.1 可得：

$$\begin{aligned} \mathbf{AB} &= \begin{bmatrix} -1 & 5 & -4 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 0 & -2 & 1 \\ -1 & 2 & 3 \end{bmatrix} \\ &= \begin{bmatrix} (-1,5,-4)\cdot(2,0,-1) & (-1,5,-4)\cdot(1,-2,2) & (-1,5,-4)\cdot(0,1,3) \\ (3,2,1)\cdot(2,0,-1) & (3,2,1)\cdot(1,-2,2) & (3,2,1)\cdot(0,1,3) \end{bmatrix} \\ &= \begin{bmatrix} 2 & -19 & -7 \\ 5 & 1 & 5 \end{bmatrix} \end{aligned}$$

注意，在个例子中我们无法计算 **BA** 的乘积，因为 **B** 的列数与 **A** 的行数不相等。这说明矩阵乘法通常不满足交换律；也就是 $\mathbf{AB} \neq \mathbf{BA}$。

### 2.2.2 向量-矩阵乘法

考虑下面的向量-矩阵乘法：

$$\mathbf{uA}=\begin{bmatrix}x,y,z\end{bmatrix}\begin{bmatrix}A_{11} & A_{12} & A_{13}\\ A_{21} & A_{21} & A_{23}\\ A_{31} & A_{32} & A_{33}\end{bmatrix}=\begin{bmatrix}x,y,z\end{bmatrix}\begin{bmatrix}\uparrow & \uparrow & \uparrow\\ \mathbf{A}_{*,1} & \mathbf{A}_{*,2} & \mathbf{A}_{*,3}\\ \downarrow & \downarrow & \downarrow\end{bmatrix}$$

注意，这里 **uA** 的计算结果是一个 1×3 行向量。运用公式 2.1 可得：

$$\begin{aligned}\mathbf{uA}&=\begin{bmatrix}\mathbf{u}\cdot\mathbf{A}_{*,1} & \mathbf{u}\cdot\mathbf{A}_{*,2} & \mathbf{u}\cdot\mathbf{A}_{*,3}\end{bmatrix}\\&=\begin{bmatrix}xA_{11}+yA_{21}+zA_{31}, xA_{12}+yA_{22}+zA_{32}, xA_{13}+yA_{23}+zA_{33}\end{bmatrix}\\&=\begin{bmatrix}xA_{11}, xA_{12}, xA_{13}\end{bmatrix}+\begin{bmatrix}yA_{21}, yA_{22}, yA_{23}\end{bmatrix}+\begin{bmatrix}zA_{31}, zA_{32}, zA_{33}\end{bmatrix}\\&=x\begin{bmatrix}A_{11}, A_{12}, A_{13}\end{bmatrix}+y\begin{bmatrix}A_{21}, A_{22}, A_{23}\end{bmatrix}+z\begin{bmatrix}A_{31}, A_{32}, A_{33}\end{bmatrix}\\&=x\mathbf{A}_{1,*}+y\mathbf{A}_{2,*}+z\mathbf{A}_{3,*}\end{aligned}$$

因此，

$$\mathbf{uA}=x\mathbf{A}_{1,*}+y\mathbf{A}_{2,*}+z\mathbf{A}_{3,*}\quad(2.2)$$

公式 2.2 是一个常用的线性组合，它说明向量-矩阵的乘积 **uA** 等于向量 **u** 给出的标量系数 $x$、$y$、$z$ 与矩阵 **A** 的每个行向量的线性组合。注意，虽然我们这里给出的例子是一个 1×3 行向量和一个 3×3 矩阵，但是这个计算方法是通用的。也就是，对于一个 1×$n$ 行向量 **u** 和一个 $n$×$m$ 矩阵 **A**，乘积 **uA** 等于 **u** 给出的标量系数与 **A** 中的每个行向量的线性组合：

$$\begin{bmatrix}u_1, & ,u_n\end{bmatrix}\begin{bmatrix}A_{11} & & A_{1m}\\ & & \\ A_{n1} & & A_{nm}\end{bmatrix}=u_1A_{1,*}+\quad+u_nA_{n,*}\quad(2.3)$$

### 2.2.3 结合律

矩阵乘法具有一些有用的代数特性。例如，可以将矩阵乘法分配给每个加法分量：**A(B+C)=AB+AC**、**(A+B)C=AC+BC**。有时我们会使用矩阵乘法的结合律来改变相乘矩阵的计算顺序：

**(AB)C=A(BC)**

## 2.3 转置矩阵

对一个矩阵的行和列进行互换，即可得到该矩阵的转置（transpose）矩阵。一个 $m$×$n$ 矩阵的转置矩阵是一个 $n$×$m$ 矩阵。我们将矩阵 **M** 的转置矩阵记作 $\mathbf{M}^T$。

## 例 2.5

求以下 3 个矩阵的转置矩阵：

$$\mathbf{A}=\begin{bmatrix}2 & -1 & 8\\ 3 & 6 & -4\end{bmatrix},\mathbf{B}=\begin{bmatrix}a & b & c\\ d & e & f\\ g & h & i\end{bmatrix},\mathbf{C}=\begin{bmatrix}1\\ 2\\ 3\\ 4\end{bmatrix}$$

对行和列进行互换，即可得到转置矩阵，因此

$$\mathbf{A}^T=\begin{bmatrix}2 & 3\\ -1 & 6\\ 8 & -4\end{bmatrix},\mathbf{B}^T=\begin{bmatrix}a & d & g\\ b & e & h\\ c & f & i\end{bmatrix},\mathbf{C}^T=\begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix}$$

矩阵转置有以下有用的特点：

1．$(\mathbf{A}+\mathbf{B})^T=\mathbf{A}^T+B^T$

2．$(c\mathbf{A})^T=c\mathbf{A}^T$

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

4．$(\mathbf{A}^T)^T=\mathbf{A}$

5．$(\mathbf{A}^{-1})^T=(\mathbf{A}^T)^{-1}$

# 2.4 单位矩阵

有一种特殊的矩阵称为单位矩阵（identity matrix）。单位矩阵是一个正方形矩阵，它除了对角线上的元素为 1 外，其他元素均为 0。

例如，下面是 2×2、3×3 和 4×4 单位矩阵。

$$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix},\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix},\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

单位矩阵的作用相当于一个乘法单位；也就是，如果 **A** 是一个 $m\times n$ 矩阵，**B** 是一个 $n\times p$ 矩阵，**I** 是 $n\times n$ 单位矩阵，那么

$$\mathbf{AI}=\mathbf{A}\text{ 且 }\mathbf{IB}=\mathbf{B}$$

换句话说，将一个矩阵与单位矩阵相乘，得到结果不会发生改变。单位矩阵可以被看成是矩阵中的数字 1。如果 **M** 是一个正方形矩阵，那么 **M** 与单位矩阵之间的相乘次序可以交换：

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

## 例 2.6

设 $\mathbf{M}=\begin{bmatrix}1 & 2\\ 0 & 4\end{bmatrix}$，$\mathbf{I}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$。证明 **MI=IM=M**。

运用公式 2.1 可得：

$$\mathbf{MI}=\begin{bmatrix}1 & 2\\ 0 & 4\end{bmatrix}\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}=\begin{bmatrix}(1,2)\cdot(1,0) & (1,2)\cdot(0,1)\\ (0,4)\cdot(1,0) & (0,4)\cdot(0,1)\end{bmatrix}=\begin{bmatrix}1 & 2\\ 0 & 4\end{bmatrix}$$

和

$$\mathbf{IM}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}1 & 2\\ 0 & 4\end{bmatrix}=\begin{bmatrix}(1,0)\cdot(1,0) & (1,0)\cdot(2,4)\\ (0,1)\cdot(1,0) & (0,1)\cdot(2,4)\end{bmatrix}=\begin{bmatrix}1 & 2\\ 0 & 4\end{bmatrix}$$

因此，**MI=IM=M** 为真。

## 例 2.7

设 **u**=[−1,2]，$\mathbf{I}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$。证明 **uI=u**。

运用公式 2.1 可得：

$$\mathbf{uI}=\begin{bmatrix}-1 & 2\end{bmatrix}\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}=\begin{bmatrix}(-1,2)\cdot(1,0) & (-1,2)\cdot(0,1)\end{bmatrix}=\begin{bmatrix}-1 & 2\end{bmatrix}$$

注意，我们无法计算 **Iu** 的乘积，因为它在矩阵乘法中没有意义。

# 2.5 矩阵行列式

矩阵行列式是一个特殊的函数，它可以将一个正方矩阵映射为一个实数，正方矩阵 **A** 的行列式通常用符号 det**A** 表示。行列式描述的是一个线性变换对“体积”所造成的影响。此外，当线性方程组对应的行列式不为零时，由克莱姆法则，可以直接以行列式的形式写出方程组的解。但是，我们使用行列式的主要目的是为了用它得到逆矩阵（2.7 节的主题）。此外，还可以证明：当且仅当正方矩阵 **A** 的行列式 det**A**≠0 时，它才是可逆的。这个结论非常有用，因为它提供了一个判断矩阵是否可逆的计算工具。在对行列式下定义之前，我们首先介绍余子式的概念。

## 2.5.1 余子式

给定一个 $n\times n$ 矩阵 **A**，余子式 $\overline{\mathbf{A}}_{ij}$ 是指删除了第 $i$ 行和第 $j$ 列后的$(n-1)\times(n-1)$矩阵。

## 例 2.8

找到下列矩阵的余子式 $\overline{\mathbf{A}}_{11}$、$\overline{\mathbf{A}}_{22}$ 和 $\overline{\mathbf{A}}_{13}$：

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

删除第 1 行和第 1 列可得：

$$\overline{\mathbf{A}}_{11}=\begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix}$$

删除第 2 行和第 2 列可得：

$$\overline{\mathbf{A}}_{22}=\begin{bmatrix} A_{11} & A_{13} \\ A_{31} & A_{33} \end{bmatrix}$$

删除第 1 行和第 3 列可得：

$$\overline{\mathbf{A}}_{13}=\begin{bmatrix} A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix}$$

## 2.5.2 定义

行列式是递归定义的；例如，4×4 矩阵的行列式是以 3×3 矩阵的形式定义的，3×3 矩阵的定义式是以 2×2 矩阵的形式定义的，2×2 矩阵的定义式是以 1×1 矩阵的形式定义的（1×1 矩阵 **A**=[$A_{11}$]可简单地表示为 det[A11] = A11）。

若 **A** 为一个 $n\times n$ 矩阵，在 $n>1$ 时我们可以定义：

$$\det\mathbf{A}=\sum_{j=1}^{n}A_{1j}(-1)^{1+j}\det\overline{\mathbf{A}}_{1j} \qquad \text{（公式 2.4）}$$

回忆一下 2×2 矩阵的余子式 $\overline{\mathbf{A}}_{ij}$ 的定义，可以得到以下式子：

$$\det\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}=A_{11}\det\left[A_{22}\right]-A_{12}\det\left[A_{21}\right]=A_{11}A_{22}-A_{12}A_{21}$$

若是 3×3 矩阵，则公式如下：

$$\det\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}=A_{11}\det\begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix}-A_{12}\det\begin{bmatrix} A_{21} & A_{23} \\ A_{31} & A_{33} \end{bmatrix}+A_{13}\det\begin{bmatrix} A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix}$$

换成 4×4 矩阵，公式变为：

$$\det\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}$$

$$= A_{11}\det\begin{bmatrix} A_{22} & A_{23} & A_{24} \\ A_{31} & A_{33} & A_{34} \\ A_{42} & A_{43} & A_{44} \end{bmatrix} - A_{12}\det\begin{bmatrix} A_{21} & A_{23} & A_{24} \\ A_{31} & A_{33} & A_{34} \\ A_{41} & A_{43} & A_{44} \end{bmatrix}$$

$$+A_{13}\det\begin{bmatrix} A_{21} & A_{22} & A_{24} \\ A_{31} & A_{32} & A_{34} \\ A_{41} & A_{42} & A_{44} \end{bmatrix} - A_{14}\det\begin{bmatrix} A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \\ A_{41} & A_{42} & A_{43} \end{bmatrix}$$

在 3D 图形中，我们主要使用 4×4 矩阵，所以就不再讨论 $n>4$ 时的公式了。

## 例 2.9

求下面矩阵的行列式：

$$\mathbf{A} = \begin{bmatrix} 2 & -5 & 3 \\ 1 & 3 & 4 \\ -2 & 3 & 7 \end{bmatrix}$$

我们可以得到：

$$\det\mathbf{A} = A_{11}\det\begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix} - A_{12}\det\begin{bmatrix} A_{21} & A_{23} \\ A_{31} & A_{33} \end{bmatrix} + A_{13}\det\begin{bmatrix} A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix}$$

$$\det\mathbf{A} = 2\det\begin{bmatrix} 3 & 4 \\ 3 & 7 \end{bmatrix} - (-5)\det\begin{bmatrix} 1 & 4 \\ -2 & 7 \end{bmatrix} + 3\det\begin{bmatrix} 1 & 3 \\ -2 & 3 \end{bmatrix}$$

$$= 2(3\cdot 7 - 4\cdot 3) + 5(1\cdot 7 - 4\cdot(-2)) + 3(1\cdot 3 - 3\cdot(-2))$$

$$= 18 + 75 + 27$$

$$= 120$$

# 2.6 伴随矩阵

设 **A** 为一个 $n\times n$ 矩阵，则 $C_{ij} = (-1)^{i+j}\det\overline{\mathbf{A}}_{ij}$ 称为元素 $A_{ij}$ 的代数余子式。如果我们计算 $C_{ij}$ 并用它替换 **A** 中的第 $ij$ 位置的每个元素，我们就可以获得 **A** 的余子矩阵 $\mathbf{C}_A$：

$$\begin{bmatrix} C_{11} & C_{12} & & C_{1n} \\ C_{21} & C_{22} & & C_{2n} \\ & & & \\ C_{n1} & C_{n2} & & C_{nn} \end{bmatrix}$$

如果我们对 $\mathbf{C}_A$ 进行转置，得到的矩阵称为 **A** 的伴随矩阵，可由下面的公式表示：

$$\mathbf{A}^* = \mathbf{C}_A^T \quad \text{（公式 2.5）}$$

在下一节中，我们会学习如何用伴随矩阵帮我们找到计算逆矩阵的明确公式。

# 2.7 逆矩阵

矩阵代数没定义除法运算，但是它定义了一种乘法的逆（inverse）运算。下面的列表总结了有关逆运算的要点：

1．只有正方形矩阵能做逆运算；所以，当我们说求逆矩阵时是假设我们正在处理的是一个正方形矩阵。

2．一个 $n \times n$ 矩阵 **M** 的逆矩阵仍然是一个 $n \times n$ 矩阵，记作 $\mathbf{M}^{-1}$。

3．不是所有的正方形矩阵都有逆矩阵。有逆矩阵的正方形矩阵称为可逆（invertible）矩阵，没有逆矩阵的称为单调（singular）矩阵。

4．如果存在逆矩阵，则该逆矩阵是唯一的。

5．将一个矩阵与它的逆矩阵相乘，其结果必定为单位矩阵：$\mathbf{MM}^{-1}=\mathbf{M}^{-1}\mathbf{M}=\mathbf{I}$。注意，矩阵与它的逆矩阵的相乘次序可以互换，这是矩阵乘法中的一个特例。

逆矩阵在求解矩阵方程时非常有用。例如，我们给出矩阵方程 $\mathbf{p}'=\mathbf{pM}$，已知 $\mathbf{p}'$ 和 **M** 的值，求解 **p**。假设 **M** 是可逆矩阵（即，$\mathbf{M}^{-1}$ 存在），那么我们可以按照如下步骤求解：

| | |
|---|---|
| $\mathbf{p}'=\mathbf{pM}$ | |
| $\mathbf{p}'\mathbf{M}^{-1}=\mathbf{pMM}^{-1}$ | 等式两边同时乘以 $\mathbf{M}^{-1}$。 |
| $\mathbf{p}'\mathbf{M}^{-1}=\mathbf{pI}$ | 由逆矩阵的定义可知 $\mathbf{M}^{-1}\mathbf{M}=\mathbf{I}$。 |
| $\mathbf{p}'\mathbf{M}^{-1}=\mathbf{p}$ | 由单位矩阵的定义可知 $\mathbf{pI}=\mathbf{p}$。 |

下面的这个方程可以用来求逆矩阵，本书不会给出证明过程，但是读者可以在任何一本大学线性代数的书籍中找到证明过程，这个方程是用伴随矩阵和行列式的形式给出的：

$$\mathbf{A}^{-1} = \frac{\mathbf{A}^*}{\det \mathbf{A}} \quad \text{（公式 2.6）}$$

## 例 2.10

找到一个求 2×2 矩阵 $\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ 的逆矩阵的通用公式，并使用这个公式求出 $\mathbf{M} = \begin{bmatrix} 3 & 0 \\ -1 & 2 \end{bmatrix}$ 的逆矩阵。

我们已经知道：

$$\det \mathbf{A} = A_{11}A_{22} - A_{12}A_{21}$$

$$\mathbf{C}_A = \begin{bmatrix} (-1)^{1+1}\det\overline{\mathbf{A}}_{11} & (-1)^{1+2}\det\overline{\mathbf{A}}_{12} \\ (-1)^{2+1}\det\overline{\mathbf{A}}_{21} & (-1)^{2+2}\det\overline{\mathbf{A}}_{22} \end{bmatrix} = \begin{bmatrix} A_{22} & -A_{21} \\ -A_{12} & A_{11} \end{bmatrix}$$

所以，

$$\mathbf{A}^{-1} = \frac{\mathbf{A}^*}{\det\mathbf{A}} = \frac{\mathbf{C}_A^T}{\det\mathbf{A}} = \frac{1}{A_{11}A_{22} - A_{12}A_{21}} \begin{bmatrix} A_{22} & -A_{12} \\ -A_{21} & A_{11} \end{bmatrix}$$

现在用这个公式求 $\mathbf{M} = \begin{bmatrix} 3 & 0 \\ -1 & 2 \end{bmatrix}$ 的逆矩阵：

$$\mathbf{M}^{-1} = \frac{1}{3\cdot 2 - 0\cdot(-1)} \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix} = \begin{bmatrix} 1/3 & 0 \\ 1/6 & 1/2 \end{bmatrix}$$

我们只需检验 $\mathbf{MM}^{-1}=\mathbf{M}^{-1}\mathbf{M}=\mathbf{I}$ 就可以证明结果是否正确：

$$\begin{bmatrix} 3 & 0 \\ -1 & 2 \end{bmatrix}\begin{bmatrix} 1/3 & 0 \\ 1/6 & 1/2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1/3 & 0 \\ 1/6 & 1/2 \end{bmatrix}\begin{bmatrix} 3 & 0 \\ -1 & 2 \end{bmatrix}$$

**注意**：对于小矩阵（4×4 矩阵或更小）来说，使用伴随矩阵的方法更有效率。对于更大的矩阵来说，我们可以使用诸如高斯消元法之类的其他方法求逆矩阵。但是，在 3D 计算机图形中，我们要处理的矩阵具有特定的形式，因此可以事先确定求逆矩阵的方程，这样我们就无需浪费 CPU 资源去求一般矩阵的逆矩阵了。这样，在代码中我们往往很少用到公式 2.6。

在本节结束之前，我们要介绍一个与逆矩阵相乘时非常有用的代数特性：

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

这个特性假设 **A** 和 **B** 都是可逆的，它们都是维数相同的正方形矩阵。要证明 $\mathbf{B}^{-1}\mathbf{A}^{-1}$ 是 **AB** 的逆矩阵，我们只需要证明$(\mathbf{AB})^{-1}(\mathbf{B}^{-1}\mathbf{A}^{-1})=\mathbf{I}$ 和$(\mathbf{B}^{-1}\mathbf{A}^{-1})(\mathbf{AB})=\mathbf{I}$。推导过程如下：

$$(\mathbf{AB})^{-1}(\mathbf{B}^{-1}\mathbf{A}^{-1})=\mathbf{A}(\mathbf{BB}^{-1})\mathbf{A}^{-1}=\mathbf{AIA}^{-1}=\mathbf{AA}^{-1}=\mathbf{I}$$

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

# 2.9 小结

1．一个 $m\times n$ 矩阵 **M** 是 $m$ 行、$n$ 列的矩形实数数组。当且仅当维数相同的两个矩阵的对应元素相等时，这两个矩阵相等。将维数相同的两个矩阵相加，即是将矩阵中的对应元素相加。将一个标量与矩阵相乘，即是将标量与矩阵中的每个元素相乘。

2．如果 **A** 是一个 $m\times n$ 矩阵，**B** 是一个 $n\times p$ 矩阵，**C** 表示 **AB** 的乘积，那么 **C** 是一个 $m\times p$ 矩阵，其中结果 **C** 的第 $ij$ 个元素等于 **A** 中的第 $i$ 个行向量与 **B** 中的第 $j$ 个列向量的点积；也就是，$\mathbf{C}_{ij}=\mathbf{A}_{i,*}\cdot\mathbf{B}_{*,j}$。

3．矩阵乘法不满足交换律（即，多数情况下 $\mathbf{AB}\neq\mathbf{BA}$）。矩阵乘法满足结合律：$(\mathbf{AB})\mathbf{C}=\mathbf{A}(\mathbf{BC})$。

4．对矩阵的行和列进行互换，即可得到矩阵的转置矩阵。因此，一个 $m\times n$ 矩阵的转置矩阵是一个 $n\times m$ 矩阵。我们使用 $\mathbf{M}^{\mathrm{T}}$ 表示矩阵 **M** 的转置矩阵。

5．单位矩阵是一种正方形矩阵，它除了对角线上的元素值为 1 外，其他元素均为 0。

6．矩阵行列式 det**A** 是一个特殊的函数，它可以将一个正方矩阵转换为一个实数。只有

在 det**A**≠0 的情况下，正方矩阵才是是可逆的。我们可以使用行列式计算逆矩阵。

7．将一个矩阵与它的逆矩阵相乘，结果为单位矩阵：$\mathbf{M}\mathbf{M}^{-1}=\mathbf{M}^{-1}\mathbf{M}=\mathbf{I}$。如果一个矩阵存在逆矩阵，则该逆矩阵是唯一的。只有正方形矩阵会有逆矩阵，但不是所有的正方形矩阵都可逆。逆矩阵可以使用公式 $\mathbf{A}^{-1}=\frac{\mathbf{A}^{*}}{\det\mathbf{A}}$ 得到，其中 $\mathbf{A}^{*}$是伴随矩阵（**A** 的余子矩阵的转置）。

# 3.1 线性变换

我们以几何方式描述 3D 场景中的物体；也就是用一组三角形近似地模拟物体的外表面。如果我们创建的物体都静止不动，那么场景就会显得索然无趣。所以，我们必须学习对几何体进行变换的方法；常见的几何变换包括平移、旋转和缩放。本章会给出许多矩阵公式，读者可以使用些公式对 3D 空间中的点和向量进行变换。

**学习目标**

1．了解如何使用矩阵表示线性变换和仿射变换。

2．学习用于缩放、旋转和平移几何体的坐标变换。

3．了解如何通过矩阵-矩阵乘法将多个变换矩阵组合为一个净变换矩阵。

4．了解如何将坐标从一个坐标系转换到另一个坐标系，以及如何通过一个矩阵来描述坐标变换。

5．熟悉用于创建变换矩阵的函数，这些函数是 XNA 数学库的一个子集。

## 3.1.1 定义

考虑一个数学函数$\tau(\mathbf{v})=\tau(x,y,z)=(x',y',z')$。这个函数的输入和输出都是一个 3D 向量。当且仅当$\tau$满足以下性质时我们才认为它是一个线性变换：

$$\begin{aligned}&1.\tau(\mathbf{u}+\mathbf{v})=\tau(\mathbf{u})+\tau(\mathbf{v})\\&2.\tau(k\mathbf{u})=k\tau(\mathbf{u})\end{aligned}\qquad\text{（公式 3.1）}$$

其中 $\mathbf{u}=(u_x,u_y,u_z)$ 和 $\mathbf{v}=(v_x,v_y,v_z)$为任意 3D 向量，$k$ 是一个标量。

**注意**：线性变换的输入和输出不一定是 3D 向量，但在 3D 图形学的书中我们无需使用其他更普遍的形式。

## 例 3.1

定义一个函数$\tau(x,y,z)=(x^2,y^2,z^2)$；例如，$\tau(1,2,3)=(1,4,9)$。这个函数不是线性的，这是因为若 $k$=2、**u**=(1,2,3)，我们可以得到：

$$\tau(k\mathbf{u})=\tau(2,4,6)=(4,16,36)$$

但

$$k\tau(\mathbf{u})=2\,(1,4,9)=(2,8,18)$$

所以不满足公式 3.1。

如果τ是线性的，它应该满足下面的式子：

$$\begin{aligned}&\tau(a\mathbf{u}+b\mathbf{c}+c\mathbf{w})=\tau(a\mathbf{u}+(b\mathbf{v}+c\mathbf{w}))\\&=a\tau(\mathbf{u})+\tau(b\mathbf{v}+c\mathbf{w})\\&=a\tau(\mathbf{u})+b\tau(\mathbf{v})+c\tau(\mathbf{w})\end{aligned}$$ （公式 3.2）

我们会在下一节中使用这个结果。

## 3.1.2 矩阵描述

令 **u**=(x,y,z)。我们总可以写成下面的形式：

$$\mathbf{u}=(x,y,z)=x\,\mathbf{i}+y\,\mathbf{j}+z\,\mathbf{k}=x\,(1,0,0)+y\,(0,1,0)+z\,(0,0,1)$$

向量 **i** = (1, 0, 0)，**j** = (0, 1, 0)和 **k** = (0, 0, 1)都是沿着坐标轴的单位向量，我们把它们称为$\mathbb{R}^3$的标准基向量（*standard basis vectors*）（$\mathbb{R}^3$表示所有 3D 坐标向量$(x, y, z)$的集合）。令τ为线性变换，则根据线性函数的特点（即公式 3.2），我们可以得到：

$$\tau(\mathbf{u})=\tau(x\mathbf{i}+y\mathbf{j}+z\mathbf{k})=x\tau(\mathbf{i})+y\tau(\mathbf{j})+z\tau(\mathbf{k})$$ （公式 3.3）

公式 3.3 其实就是一个线性组合，我们在上一章就已经讨论过了，这个线性组合可以根据公式 2.2 写成矢量与矩阵的乘法，因此我们可以将公式 3.3 重写成如下形式：

$$\begin{aligned}&\tau(\mathbf{u})=x\tau(\mathbf{i})+y\tau(\mathbf{j})+z\tau(\mathbf{k})\\&=\mathbf{u}\mathbf{A}=\begin{bmatrix}x,y,z\end{bmatrix}\begin{bmatrix}\leftarrow & \tau(\mathbf{i}) & \rightarrow\\ \leftarrow & \tau(\mathbf{j}) & \rightarrow\\ \leftarrow & \tau(\mathbf{k}) & \rightarrow\end{bmatrix}=\begin{bmatrix}x,y,z\end{bmatrix}\begin{bmatrix}A_{11} & A_{12} & A_{13}\\ A_{21} & A_{21} & A_{23}\\ A_{31} & A_{32} & A_{33}\end{bmatrix}\end{aligned}$$ （公式 3.4）

其中$\tau(\mathbf{i})=(A_{11}, A_{12}, A_{13})$，$\tau(\mathbf{j})=(A_{21}, A_{22}, A_{23})$，$\tau(\mathbf{k})=(A_{31}, A_{32}, A_{33})$。我们把矩阵 **A** 称为线性变换τ的矩阵描述。

## 3.1.3 缩放

缩放是指改变一个物体的大小，如图 3.1 所示。

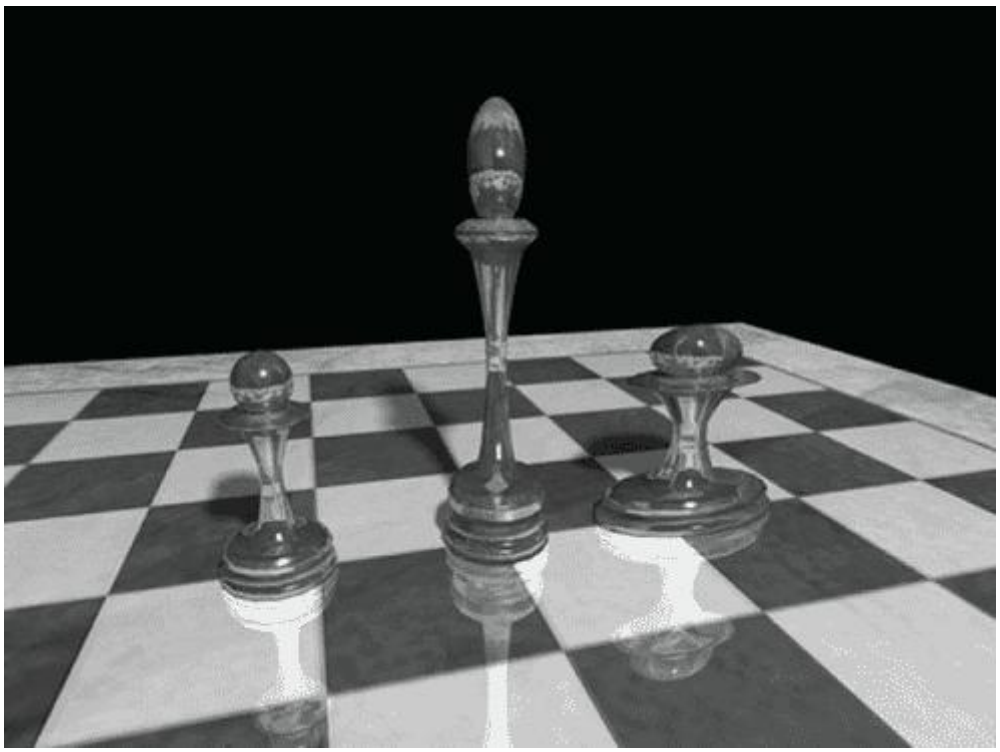


**图 3.1 左边的兵（pawn，国际象棋中的兵）是原始物体。中间的兵是沿 y 轴放大两倍后的结果。右边的兵是沿 x 轴放大两倍后的结果。**

我们将缩放变换定义为：

$$S(x,y,z)=(s_x x, s_y y, s_z z)$$

上述变换将向量沿 $x$ 轴方向缩放 $s_x$ 单位，$y$ 轴方向缩放 $s_y$ 个单位，$z$ 轴方向缩放 $s_z$ 个单位（相对于目前的坐标系原点）。下面我们证明 $S$ 是一个线性变换：

$$\begin{aligned}S(\mathbf{u}+\mathbf{v}) &= (s_x(u_x+v_x), s_y(u_y+v_y), s_z(u_z+v_z))\\ &= (s_x u_x + s_x v_x, s_y u_y + s_y v_y, s_z u_z + s_z v_z)\\ &= (s_x u_x, s_y u_y, s_z u_z) + (s_x v_x, s_y v_y, s_z v_z)\\ &= S(\mathbf{u}) + S(\mathbf{v})\end{aligned}$$

$$\begin{aligned}S(k\mathbf{u}) &= (s_x k u_x, s_y k u_y, s_z k u_z)\\ &= k(s_x u_x, s_y u_y, s_z u_z)\\ &= kS(\mathbf{u})\end{aligned}$$

满足公式 3.1 的两个性质，所以 $S$ 是线性的，应该存在一个矩阵描述。要找到这个矩阵描述，我们只需将 $S$ 代入公式 3.3 中的每个标准基向量中即可，然后将得出的结果向量替换矩阵的行：

$$\begin{aligned}S(\mathbf{i}) &= (s_x \cdot 1, s_y \cdot 0, s_z \cdot 0) = (s_x, 0, 0)\\ S(\mathbf{j}) &= (s_x \cdot 0, s_y \cdot 1, s_z \cdot 0) = (0, s_y, 0)\\ S(\mathbf{k}) &= (s_x \cdot 0, s_y \cdot 0, s_z \cdot 1) = (0, 0, s_z)\end{aligned}$$

$S$ 的矩阵表示为：

$$\mathbf{S}=\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

我们把这个矩阵叫做缩放矩阵。

缩放矩阵的逆矩阵为：

$$\mathbf{S}=\begin{bmatrix} 1/s_x & 0 & 0 \\ 0 & 1/s_y & 0 \\ 0 & 0 & 1/s_z \end{bmatrix}$$

## 例 3.2

假设我们通过一个最小点(−4, −4,0)和一个最大点(4, 4,0)来定义一个正方形，我们希望将正方形沿 *x* 轴缩小 0.5 倍，沿 *y* 轴放大 2.0 倍，*z* 轴保持不变。则对应的缩放矩阵为：

$$\mathbf{S}=\begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

现在，对正方形进行缩放（变换），将正方形的两个点与该矩阵相乘：

$$\begin{bmatrix} -4 & -4 & 0 \end{bmatrix}\begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}=\begin{bmatrix} -2 & -8 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 4 & 0 \end{bmatrix}\begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}=\begin{bmatrix} 2 & 8 & 0 \end{bmatrix}$$

结果如图 3.2 所示。

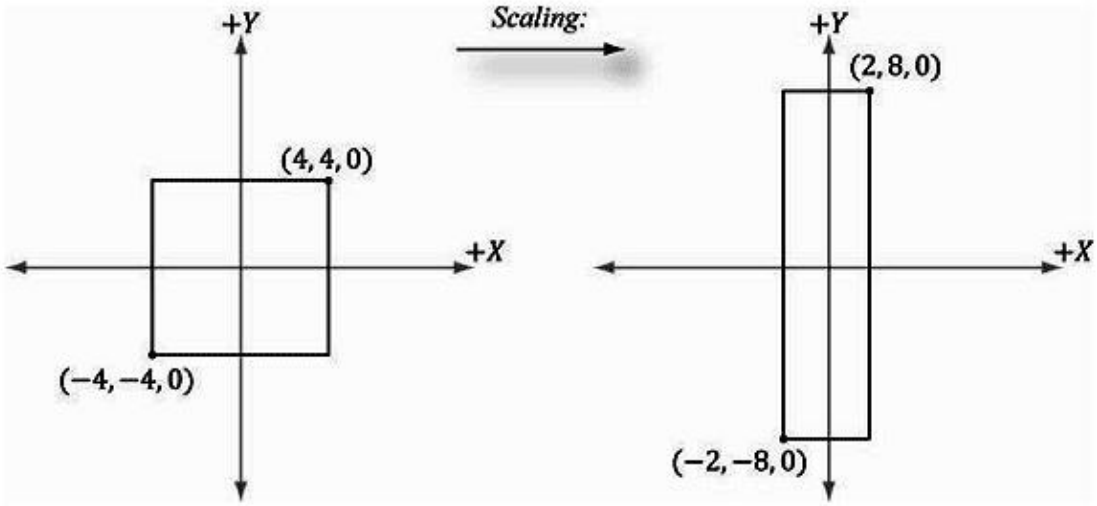



**图 3.2 沿 x 轴缩小 0.5 倍，沿 *y* 轴扩大 2 倍。注意，当沿 *z* 轴负方向俯视时，由于 *z* 值为 0，几何体看上去是一个 2D 平面图形。**

## 3.1.4 旋转

本节我们将介绍如何将向量 **v** 绕一根轴 **n** 旋转$\theta$角度；如图 3.3 所示。注意，在左手坐标系中，当沿着旋转轴的正轴方向俯视时，顺时针方向为正角；而且，我们假设||**n**||=1。

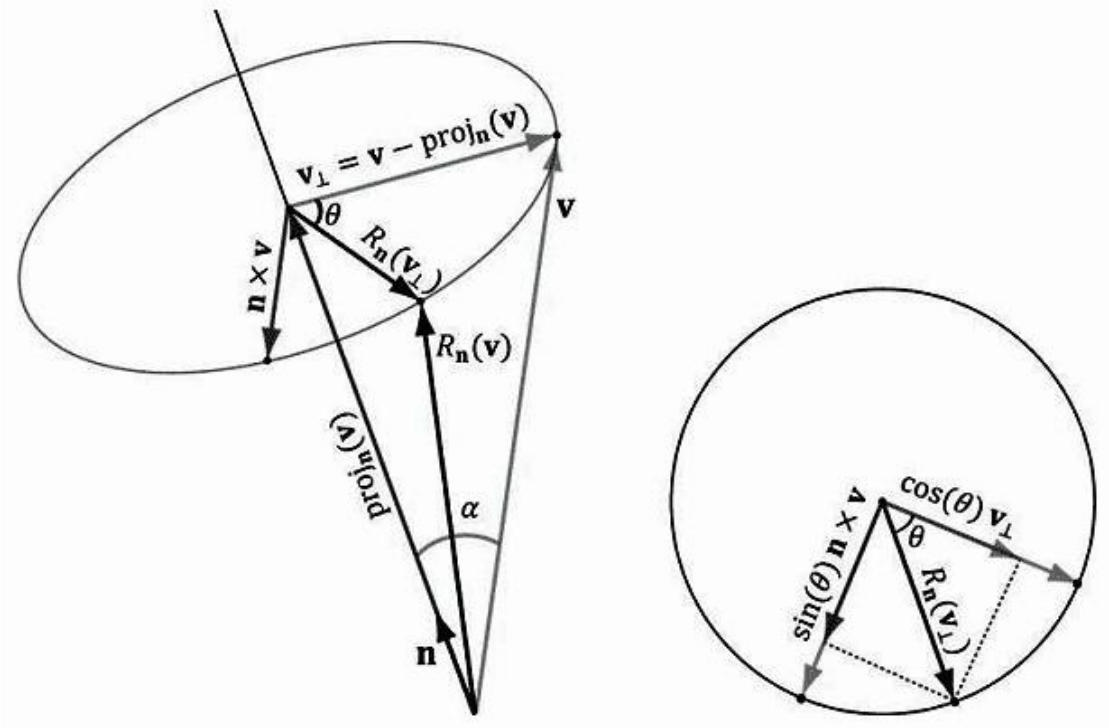



**图 3.3 绕任意轴 n 旋转的几何表示。**

首先，将 **v** 分解为两个分量，其中一个分量平行于 **n**，另一个垂直于 **n**。平行分量即 $\text{proj}_\mathbf{n}(\mathbf{v})$（回忆一下例 1.5）；垂直分量可以通过 $\mathbf{v}_\perp = \text{perp}_\mathbf{n}(\mathbf{v}) = \mathbf{v} - \text{proj}_\mathbf{n}(\mathbf{v})$得到（还是回忆一下例 1.5，因为 **n** 是单位向量，**所以** $\text{proj}_\mathbf{n}(\mathbf{v}) = (\mathbf{n}\cdot\mathbf{v})\mathbf{n}$。）。平行于 **n** 的 $\text{proj}_\mathbf{n}(\mathbf{v})$分量在旋转过程中是不变的，所以我们只需计算垂直分量的旋转。从图 3.3 中我们可以看出，旋转后的向量 $R_\mathbf{n}(\mathbf{v}) = \text{proj}_\mathbf{n}(\mathbf{v}) + R_\mathbf{n}(\mathbf{v}_\perp)$。

要找到 $R_n(\mathbf{v}_\perp)$，我们需要建立一个位于旋转平面的 2D 坐标系。将 $\mathbf{v}_\perp$作为一个基准向量，第二个基准向量需要同时垂直于 $\mathbf{v}_\perp$和 **n**，我们取为 $\mathbf{n} \times \mathbf{v}$（左手拇指定则）。根据图 3.3 中的几何关系和第一章的练习 14，我们可以得出：

$$\|\mathbf{n}\times\mathbf{v}\| = \|\mathbf{n}\|\|\mathbf{v}\|\sin\alpha = \|\mathbf{v}\|\sin\alpha = \|\mathbf{v}_\perp\|$$

其中$\alpha$为 **n** 和 **v** 之间的夹角。这样这两个基准向量都有相同的长度并且都在旋转平面上。创建了两个基准向量后，我们就可以根据三角学的知识得出：

$$R_n(\mathbf{v}_\perp) = \cos\theta\mathbf{v}_\perp + \sin\theta(\mathbf{n}\times\mathbf{v})$$

并由此得到下面的旋转方程：

$$\begin{aligned} R_n(\mathbf{v}) &= proj_n(\mathbf{v}) + R_n(\mathbf{v}_\perp) \\ &= (\mathbf{n}\cdot\mathbf{v})\mathbf{n} + \cos\theta\mathbf{v}_\perp + \sin\theta(\mathbf{n}\times\mathbf{v}) \\ &= (\mathbf{n}\cdot\mathbf{v})\mathbf{n} + \cos\theta(\mathbf{v} - (\mathbf{n}\cdot\mathbf{v})\mathbf{n}) + \sin\theta(\mathbf{n}\times\mathbf{v}) \\ &= \cos\theta\mathbf{v} + (1-\cos\theta)(\mathbf{n}\cdot\mathbf{v})\mathbf{n} + \sin\theta(\mathbf{n}\times\mathbf{v}) \end{aligned} \qquad \text{（公式 3.5）}$$

我们把证明公式 3.5 为一个线性变换放在了后面的练习中，这里不予讨论。要找到对应的矩阵描述，我们只需将 $R_n$ 代入公式 3.3 中的每个标准基向量中即可，然后将得出的结果向量替换矩阵的行（即在公式 3.4 中）。最终结果为：

$$\mathbf{R}_n = \begin{bmatrix} c+(1-c)x^2 & (1-c)xy+sz & (1-c)xz-sy \\ (1-c)xy-sz & c+(1-c)y^2 & (1-c)yz-sx \\ (1-c)xz-sy & (1-c)yz+sx & c+(1-c)z^2 \end{bmatrix}$$

其中 $c=\cos\theta$，$s=\sin\theta$。

旋转矩阵有一个有趣的特性。读者可以验证一下：旋转矩阵的每个行向量都是单位向量，而且相互垂直。也就是说，它的每个行向量都是标准正交的（即，相互垂直且为单位长度）。我们将这种矩阵称为正交矩阵（orthogonal matrix）。正交矩阵有一个非常有用的特性，它的逆矩阵与它的转置矩阵相等。也就是，$\mathbf{R}_n$ 的逆矩阵为：

$$\mathbf{R}_n^{-1} = \mathbf{R}_n^T = \begin{bmatrix} c+(1-c)x^2 & (1-c)xy-sz & (1-c)xz+sy \\ (1-c)xy+sz & c+(1-c)y^2 & (1-c)yz-sx \\ (1-c)xz-sy & (1-c)yz+sx & c+(1-c)z^2 \end{bmatrix}$$

通常，正交矩阵是最容易使用的矩阵，因为它们计算逆矩阵的过程非常简单，也非常高效。

当我们以 $x$、$y$、$z$ 轴（即，$\mathbf{n}=(1,0,0)$、$\mathbf{n}=(0,1,0)$、$\mathbf{n}=(0,0,1)$）为旋转轴时，对应的旋转矩阵如下：

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix}, \mathbf{R}_y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}, \mathbf{R}_z = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

# 例 3.3

假设我们通过一个最小点(−1,0,−1)和一个最大点(1,0,1)来定义一个正方形。让正方形绕着 $y$ 轴的顺时针方向旋转-30º（即，逆时针方向旋转 30º）。在这种情况中，$\mathbf{n}=(0,1,0)$，$\mathbf{R}_n$ 大为简化；对应的 $y$ 轴旋转矩阵为：

$$\mathbf{R}_y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} = \begin{bmatrix} \cos(-30°) & 0 & -\sin(-30°) \\ 0 & 1 & 0 \\ \sin(-30°) & 0 & \cos(-30°) \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2} \end{bmatrix} \approx \begin{bmatrix} 0.36, 0, 1.36 \end{bmatrix}$$

现在，对正方形进行旋转（变换），将正方形的两个点与该矩阵相乘：

$$[-1,0,-1]\begin{bmatrix}\frac{\sqrt{3}}{2} & 0 & \frac{1}{2}\\ 0 & 1 & 0\\ -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2}\end{bmatrix}\approx[-0.36,0,-1.36]$$

$$[1,0,1]\begin{bmatrix}\frac{\sqrt{3}}{2} & 0 & \frac{1}{2}\\ 0 & 1 & 0\\ -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2}\end{bmatrix}\approx[0.36,0,1.36]$$

结果如图 3.4 所示。

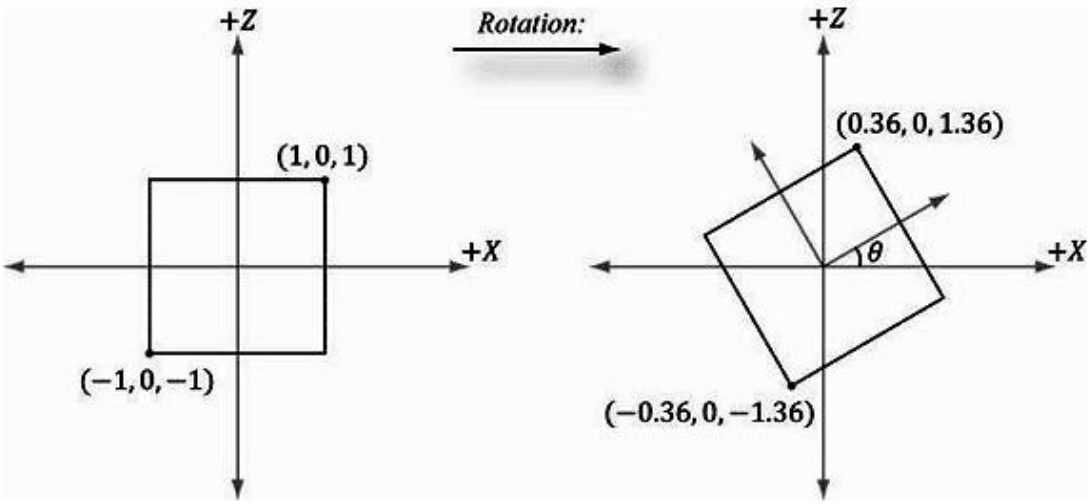



**图 3.4：绕 *y* 轴顺时针方向旋转−30°。注意，当沿 *y* 轴正方向俯视时，由于 *y* 值为 0，几何体看上去是一个 2D 平面图形。**

# 3.2 仿射变换

## 3.2.1 齐次坐标

下一节我们就会知道仿射变换是一个组合了平移的线性变换。但是，因为向量只表示方向和长度，与位置无关，所以平移一个向量是无意义的，换句话说，平移后的向量是不变的。平移只能作用在点上（即，位置向量）。齐次坐标提供了一个便捷的表示方法用来统一处理点和向量。在齐次坐标中，我们使用 4 个元素，我们通过它的第 4 个坐标分量 *w* 来决定所描述的是一个点还是一个向量。确切地说，我们写为：

1．(*x*,*y*,*z*,0)用于向量

2．(*x*,y,*z*,1)用于点

我们将会看到，把 *w* 设为 1 是为了让点的平移操作得到正确执行，把 *w* 设为 0 是为了防止向量在变换过程中发生平移。（我们不希望平移向量的坐标，因为向量可以改变的只有方向和大小——平移对向量来说没有意义。）

**注意**：齐次坐标的记法与图 1.17 所示的概念一致。也就是，两点相减 $\mathbf{q}-\mathbf{p}=(q_x,q_y,q_z,1)-(p_x,p_y,p_z,1)=(q_x-p_x,q_y-p_y,q_z-p_z,0)$的结果是一个向量，而一个点与一个向量相加 $\mathbf{p}+\mathbf{v}=(p_x,p_y,p_z,1)+(v_x,v_y,v_z,0)=(p_x+v_x,p_y+v_y,p_z+v_z,1)$的结果是个点。

## 3.2.2 定义和矩阵表示

一个线性变换无法表示所有我们需要的变换；所以，我们需要添加一组叫做仿射变换的函数。仿射变换是一个线性变换加上一个平移向量 **b**；也就是：

$$\alpha(\mathbf{u})=\tau(\mathbf{u})+b$$

或者用矩阵表示为：

$$\alpha(\mathbf{u})=\mathbf{uA}+\mathbf{b}=[x,y,z]\begin{bmatrix}A_{11}&A_{12}&A_{13}\\A_{21}&A_{22}&A_{23}\\A_{31}&A_{32}&A_{33}\end{bmatrix}+[b_x,b_y,b_z]=[x',y',z']$$

式中的 A 是线性变换的矩阵表示。

如果使用 $w$=1 的齐次坐标，则可表示为：

$$[x,y,z,1]\begin{bmatrix}A_{11}&A_{12}&A_{13}&0\\A_{21}&A_{22}&A_{23}&0\\A_{31}&A_{32}&A_{33}&0\\b_x&b_y&b_z&1\end{bmatrix}=[x',y',z',1]\quad\text{（公式 3.6）}$$

公式 3.6 中的 4×4 矩阵称为仿射矩阵的矩阵表示。

额外添加的 **b** 就是指平移（即，位置的改变）。因为向量没有位置的概念，所以我们并不想将 **b** 作用在向量上。但是，我们还是想将仿射矩阵的线性变换部分作用在向量上。如果我们把向量的第四个分量 $w$ 设置为 0，那么 **b** 对应的平移部分就不会作用到向量上。

**注意**：因为行向量与 4×4 仿射矩阵第 4 列的点乘是$[x, y, z, w]\cdot[0, 0, 0, 1]=w$，所以这个矩阵不会改变输入向量的 $w$ 坐标。

## 3.2.3 平移

单位变换（identity transformation）是一个线性变换，返回值就是输入的向量；即，$I(\mathbf{u})=\mathbf{u}$。这说明线性变换的矩阵表示就是一个单位矩阵。

现在，我们将一个平移变换定义为一个仿射变换，这个仿射变换的线性变换部分是一个单位变换；即：

$$\tau(\mathbf{u})=\mathbf{uI}+\mathbf{b}=\mathbf{u}+\mathbf{b}$$

如你所见，这可以简化将点 **u** 移动 **b** 的操作。图 3.5 说明了如何平移物体——要对点 **u** 进行平移，只需要将一个移位向量 **b** 和该点相加，即可得到新的点 **u**+**b**。注意，要平移一个完整的物体，我们就要通过相同的向量 **b** 来平移物体上的每个点。

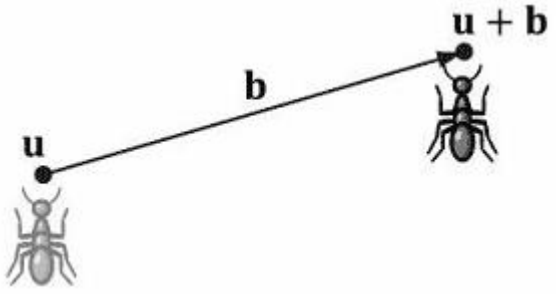



**图 3.5 通过位移向量 b 对蚂蚁的位置进行平移。**

根据公式 3.6，τ的矩阵表示如下：

$$\mathbf{T}=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ b_x & b_y & b_z & 1\end{bmatrix}$$

这个矩阵称之为平移矩阵。

平移矩阵的逆矩阵如下：

$$\mathbf{T}^{-1}=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ -b_x & -b_y & -b_z & 1\end{bmatrix}$$

## 例 3.4

假设我们通过一个最小点(−8, 2,0)和一个最大点(−2, 8,0) 来定义一个正方形。让正方形沿 *x* 轴平移 12，沿 *y* 轴平移-10，*z* 轴保持不变。则对应的平移矩阵如下：

$$\mathbf{T}=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 12 & -10 & 0 & 1\end{bmatrix}$$

现在，对正方形进行平移（变换），将正方形的两个点与该矩阵相乘：

$$\begin{bmatrix}-8 & 2 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 12 & -10 & 0 & 1\end{bmatrix}=\begin{bmatrix}4 & -8 & 0 & 1\end{bmatrix}$$

$$\begin{bmatrix}-2 & 8 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 12 & -10 & 0 & 1\end{bmatrix}=\begin{bmatrix}10 & -2 & 0 & 1\end{bmatrix}$$

结果如图 3.6 所示。

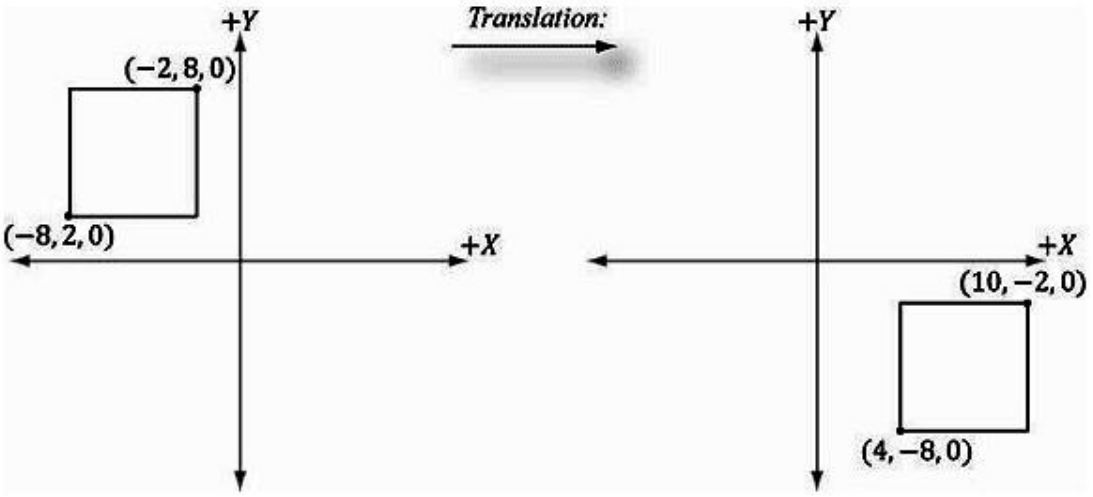



**图 3.6 沿 x 轴平移 12，沿 *y* 轴平移−10。注意，当沿 *z* 轴负方向俯视时，由于 *z* 值为 0，几何体看上去是一个 2D 平面图形。**

**注意**：令 **T** 为一个变换矩阵，通过计算 **vT**= **v**′就可以对点/向量进行变换。如果先使用 **T** 对点/向量进行变换，然后再使用逆矩阵 $\mathbf{T}^{-1}$ 进行变换，我们就会得到初始的向量：$\mathbf{vTT}^{-1} = \mathbf{vI} = \mathbf{v}$。换句话说，逆变换可以撤销变换。例如，如果我们将一个点沿 x 轴移动 5，然后沿 x 轴移动-5，会又回到出发点。类似地有，将一个点沿 y 轴旋转 30°，然后反向旋转 30°，该点又回到了原来的位置。总而言之，逆变换矩阵的变换效果与变换相反，两者的组合会导致不产生变换效果。

## 3.2.4 缩放和旋转的仿射矩阵

若 **b**=0，仿射变换就退化为一个线性变换。我们可以将任何一个线性变换表示成 **b**=0 的仿射变换。换句话说，我们可以用一个 4×4 仿射矩阵表示任意一个线性变换。例如，缩放和旋转矩阵可以用 4×4 矩阵写成如下形式：

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_n = \begin{bmatrix} c+(1-c)x^2 & (1-c)xy+sz & (1-c)xz-sy & 0 \\ (1-c)xy-sz & c+(1-c)y^2 & (1-c)yz+sx & 0 \\ (1-c)xz+sy & (1-c)yz-sx & c+(1-c)z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

通过这种方式，我们使用 4×4 矩阵表示所有变换，使用 1×4 齐次行向量矩阵表示点和向量，形式得到了统一。

## 3.2.5 仿射变换矩阵的几何解释

在本节中，我们会对一个仿射变换矩阵中的数字表示的几何意义有个直观的认识。首先，

考虑一个刚体变换，它是一个形状保持不变的变换。现实世界中的例子是将一本书从桌上拿起放到书架上，在这个过程中，你将书从书桌平移到了书架，还改变了书的朝向（旋转）。令$\tau$为旋转变换，**b** 为位移矢量，则这个刚体变换可以用以下仿射变换表示：

$$\alpha(x,y,z)=\tau(x,y,z)+\mathbf{b}=x\tau(\mathbf{i})+y\tau(\mathbf{j})+z\tau(\mathbf{k})+\mathbf{b}$$

在矩阵表示中，使用其次坐标（$w$=1 表示位置，$w$=0 表示向量，这样平移就不会作用在向量上），上式可以写成：

$$[x,y,z,w]\begin{bmatrix}\leftarrow\tau(\mathbf{i})\rightarrow\\ \leftarrow\tau(\mathbf{j})\rightarrow\\ \leftarrow\tau(\mathbf{k})\rightarrow\\ \leftarrow\mathbf{b}\rightarrow\end{bmatrix}=[x',y',z',w] \quad \text{（公式 3.7）}$$

现在看一下公式 3.7 的几何意义，我们只需画出矩阵中的行向量（参见图 3.7）。因为τ是一个旋转变换，所以它保存了长度和角度信息；而且τ只是将标准基向量 **i**，**j** 和 **k** 旋转到一个新的朝向**τ(i)**，**τ(j)**和**τ(k)**。向量 **b** 只是一个位置向量，它表示的是离开原点的位移。图 3.7 展示了如何通过计算$\alpha(x,y,z)=x\tau(\mathbf{i})+y\tau(\mathbf{j})+z\tau(\mathbf{k})+\mathbf{b}$获得变换后的点。

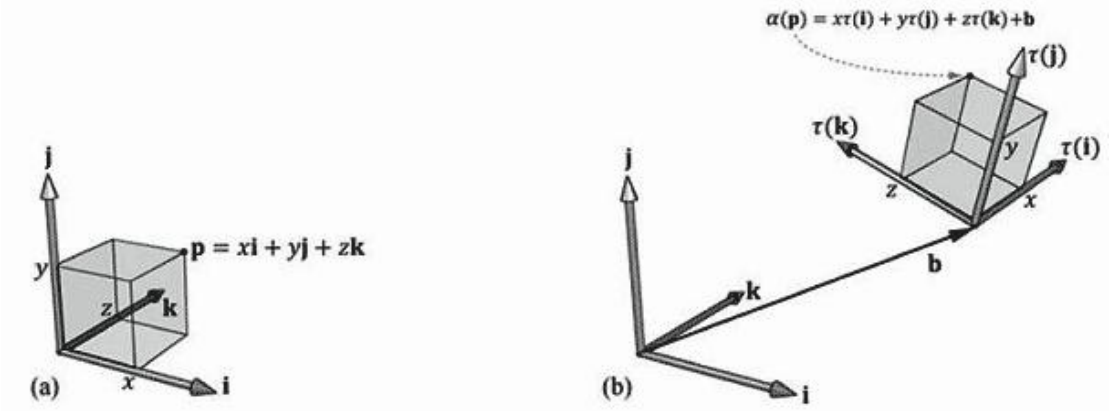



**图 3.7 仿射变换矩阵中行向量的几何意义。变换后的点α(p)为基向量τ(i)，τ(j)，τ(k)的线性组合和偏移量 b 的和。**

缩放和扭曲变换的原理相同。考虑线性变换τ，如图 3.8 所示，这个变换将正方形扭曲成一个平行四边形。扭曲后的点就是扭曲后的基向量的线性组合。

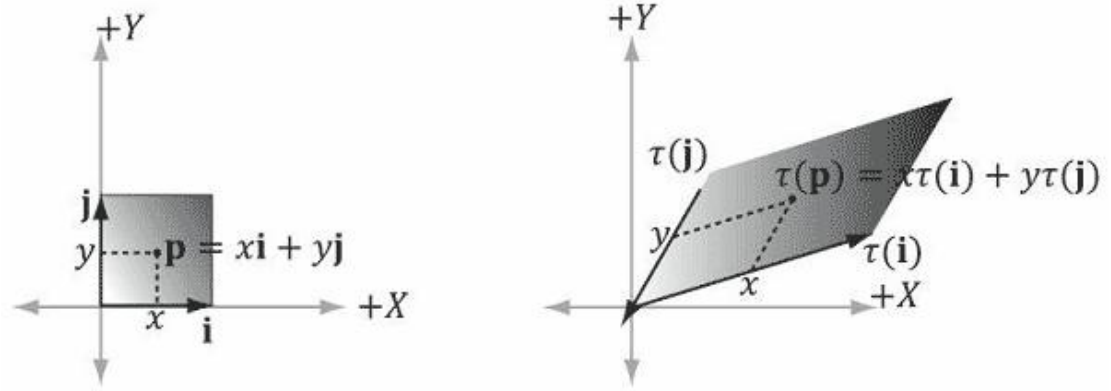



**图 3.8 对于将正方形扭曲为一个平行四边形的线性变换来说，变换后的点τ(p)=(*x*, *y*)是变换后的基向量τ(i)，τ(j)的线性组合。**

# 3.3 组合变换

假设 **S** 是一个缩放矩阵，**R** 是一个旋转矩阵，**T** 是一个平移矩阵；另外，我们有一个由 8 个顶点 $\mathbf{v}_i$（i=0，1，…，7）构成的立方体，我们希望将这 3 个变换连续应用于立方体的每个顶点。一种最容易想到的方法是将这些矩阵逐一应用于每个顶点：

$$((\mathbf{v}_i\mathbf{S})\mathbf{R})\mathbf{T} = (\mathbf{v}_i'\mathbf{R})\mathbf{T} = \mathbf{v}_i''\mathbf{T} = \mathbf{v}_i''' \quad \text{其中 } i=0，1，\cdots，7$$

但是，由于矩阵乘法支持结合律，所以我们可以将上面的方程改为：

$$\mathbf{v}_i(\mathbf{SRT}) = \mathbf{v}_i''' \quad \text{其中 } i=0，1，\cdots，7$$

我们可以将 **SRT** 看成一个矩阵 **C**，将所有的 3 个变换封装为一个净仿射变换矩阵。换句话说，矩阵-矩阵乘法可以让我们把多个变换连接在一起。

这种方法有助于提升性能。比如，我们将 3 个连续的几何变换应用于一个由 20,000 个点构成的 3D 物体。使用逐一相乘的方法，我们需要执行 20,000×3 次向量-矩阵乘法。而改用组合矩阵方式，我们只需要执行 20,000 次向量-矩阵乘法和两次矩阵-矩阵乘法。很明显，两次额外的矩阵-矩阵乘法所产生的资源消耗微乎其微，而它们却能省去大量的向量-矩阵乘法。

**注意**：我们再次强调，矩阵乘法不支持交换律。这完全符合于几何学中的定义。例如，在旋转之后进行一次平移，可以由矩阵乘积 **RT** 表示，但是它的结果与 **TR** 完全不同，也就是，在相同的平移之后进行一次相同的旋转，得到的变换结果完全不同。图 3.9 说明了这一点。

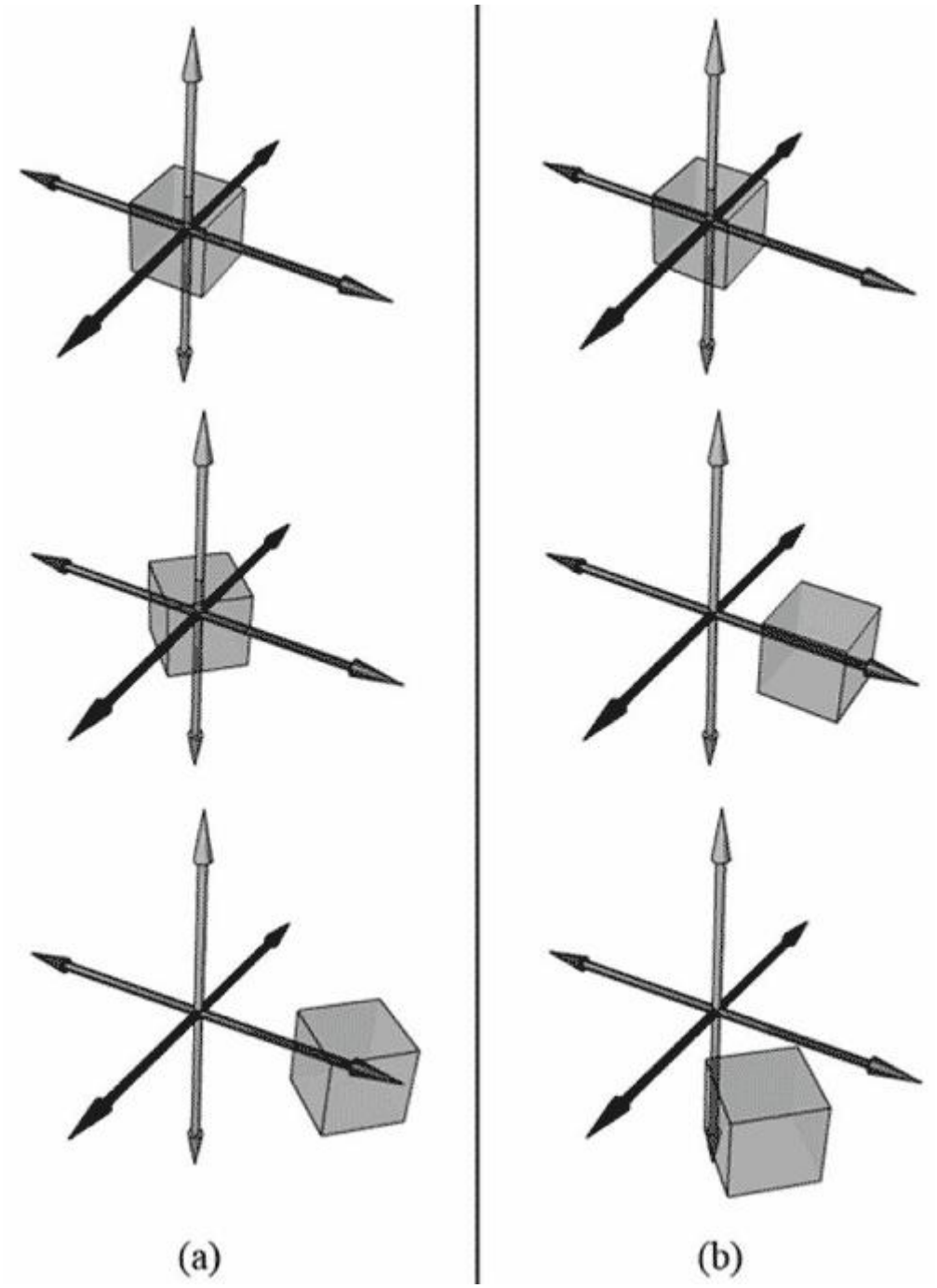



**图 3.9 (a)先旋转再平移。(b)先平移再旋转。**

# 3.4 坐标转换变换

标量 100℃是相对于摄氏温标表示的水的沸点温度。那么我们该如何以华氏温标来描述的水的沸点温度呢？换句话说就是在华氏温标中表示水的沸点温度的标量是多少？要实现这一转换（或参考系变换），我们需要知道摄氏与华氏之间的比例关系。它们的关系如下：$T_F=\frac{9}{5}T_C+32°$。因此，$T_F=\frac{9}{5}$（100）°＋32°=212℉；也就是，水的沸点温度为华氏 212

℉。

这个例子说明，只要我们知道参考系 *A* 和参考系 *B* 的关系，就可以将一个相对于参考系 *A* 标量 *k* 转换为相对于参考系 *B* 描述的等价标量 *k′*。在下面的小节中，我们会看到一些类似的问题，但不是标量而是坐标，我们会将一个点或向量的坐标从一个参考系转换到另一个不同的参考系（参见图 3.10）。我们将这种把坐标从一个参考系转换到另一个参考系的变换称为坐标转换变换（change of coordinate transformation）。

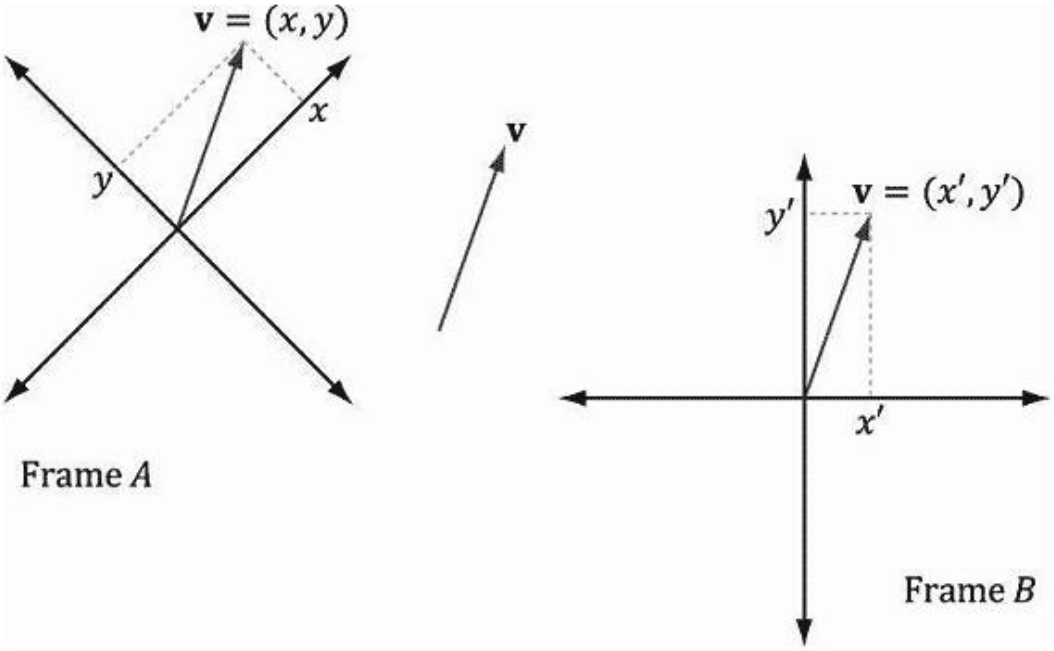



**图 3.10 当相对于不同的参考系来描述同一个向量 v 时，该向量会有不同的坐标。**

值得强调的是，在坐标转换变换中，我们并不认为几何体发生了什么改变；而是认为我们对参考系进行了转换，改变了几何体坐标的表达方式。相比之下，我们通常认为旋转、平移和缩放会对几何体产生实质性的移动或变形。

在 3D 计算机绘图中，由于我们会用到很多种不同坐标系，所以我们需要知道如何从一种坐标系转换到另一种坐标系。由于位置是点的属性，而不是向量的属性，所以点和向量在实现坐标转换变换时要区别对待，使用不同的处理方式。

## 3.4.1 向量

考虑图3.11中的两个参考系*A*、参考系*B*及向量**p**。假设**p**在参考系A中的坐标为$\mathbf{p}_A=(x,y)$，我们想要求出 **p** 相对于参考系 *B* *的*坐标 $\mathbf{p}_B=(x',y')$。换句话说，在一个参考系中通过一个坐标指定一个向量，我们将个向量保持不变，只更换一个不同的参考系，那我们该如何求出这个向量的新坐标呢？

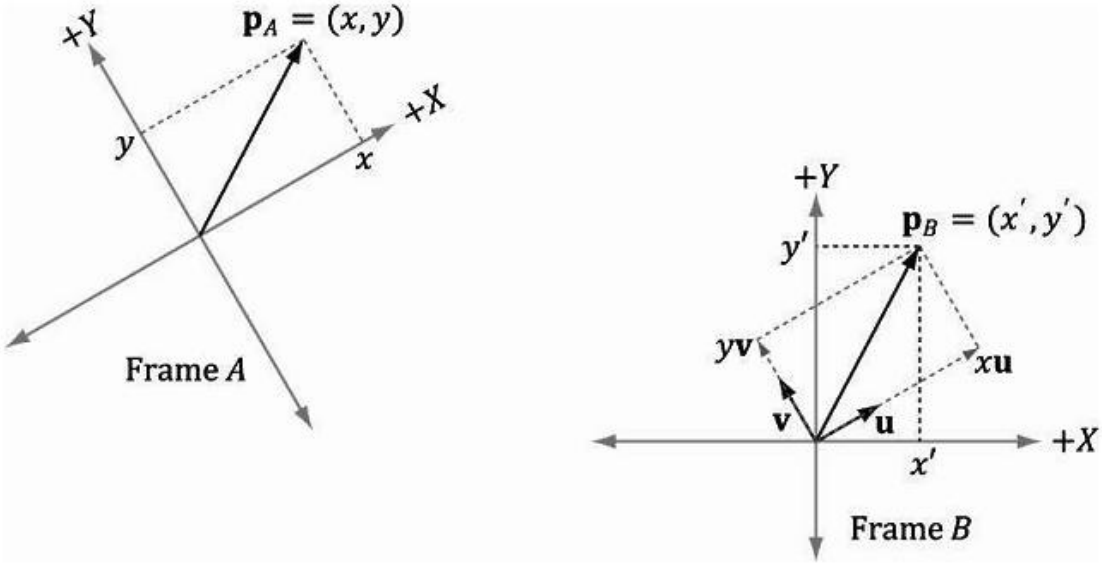



**图 3.11 求 p 相对于参考系 *B* 的几何坐标。**

从图 3.11 可知

$$\mathbf{p}=x\mathbf{u}+y\mathbf{v}$$

其中 **u** 和 **v** 是单位向量，它们所指向的方向分别与参考系 *A* 的 *x* 轴和 *y* 轴方向相同。通过上述方程，我们可以得到每个向量在参考系 B 中的坐标：

$$\mathbf{p}_B=x\mathbf{u}_B+y\mathbf{v}_B$$

这样，只要我们知道向量 **u**、**v** 相对于参考系 *B* 的坐标，即 $\mathbf{u}_B=(u_x,u_y)$和 $\mathbf{v}_B=(v_x,v_y)$，那么对于给出的任意 $\mathbf{p}_A=(x,y)$，都可以计算出 $\mathbf{p}_B=(x',y')$。

推导为 3D 向量，如果 $\mathbf{p}_A=(x,y,z)$，则

$$\mathbf{p}_B=x\mathbf{u}_B+y\mathbf{v}_B+z\mathbf{w}_B$$

其中 **u**、**v**、**w** 是单位向量，它们所指向的方向分别与参考系 *A* 的 *x* 轴、*y* 轴、*z* 轴方向相同。

## 3.4.2 点

当进行坐标转换变换时，点与向量之间存在一些微小差异；由于位置是点的一个重要属性，所以我们不能按照图 3.11 中的平移向量的方式来平移点。

图 3.12 说明了这一情况。我们看到，点 **p** 可以由一个方程来表示：

$$\mathbf{p}=x\mathbf{u}+y\mathbf{v}+\mathbf{Q}$$

其中 **u** 和 **v** 是单位向量，它们所指向的方向分别与参考系 *A* 的 *x* 轴和 *y* 轴方向相同，**Q** 是参考系 *A* 的原点。通过上述方程，我们可以得到向量或点在参考系 *B* 中的坐标：

$$\mathbf{p}_B=x\mathbf{u}_B+y\mathbf{v}_B+\mathbf{Q}_B$$

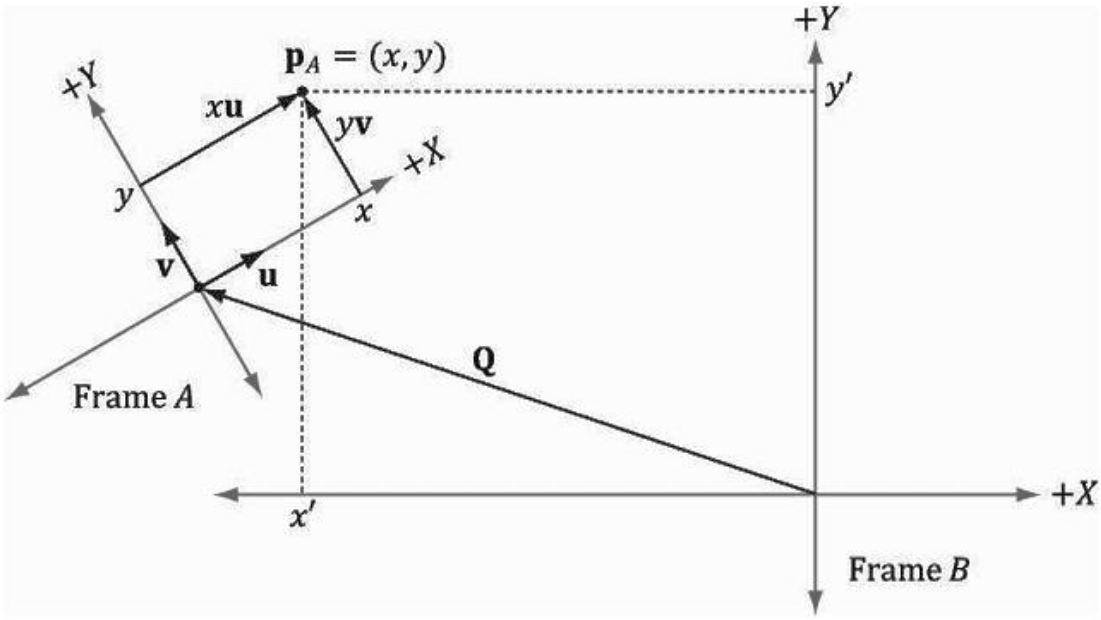



**图 3.12 求 p 相对于参考系 *B* 的几何坐标。**

这样，只要我们知道向量 **u**、**v** 的坐标以及相对于参考系 *B* 的原点，即 $\mathbf{u}_B=(u_x,u_y)$、$\mathbf{v}_B=(v_x,v_y)$、$\mathbf{Q}_B=(Q_x,Q_y)$，那么对于给出的任意坐标 $\mathbf{p}_A=(x,y)$，都可以求出 $\mathbf{p}_B=(x',y')$。

推导为 3D 向量，如果 $\mathbf{p}_A=(x,y,z)$，则

$$\mathbf{p}_B=x\mathbf{u}_B+y\mathbf{v}_B+z\mathbf{w}_B+\mathbf{Q}_B$$

其中 **u**、**v**、**w** 是单位向量，它们所指的方向分别与参考系 *A* 的 *x* 轴、*y* 轴、*z* 轴方向相同，**Q** 是参考系 *A* 的原点。

## 3.4.3 矩阵表示

回顾之前讲过的内容，向量和点的坐标转换变换分别为：

$(x',y',z')= x\mathbf{u}_B+y\mathbf{v}_B+z\mathbf{w}_B$ 用于向量

$(x',y',z')= x\mathbf{u}_B+y\mathbf{v}_B+z\mathbf{w}_B+\mathbf{Q}_B$ 用于点

如果我们使用齐次坐标，那么就可以通过一个方程同时处理向量和点：

$$(x', y', z', w) = x\mathbf{u}_B + y\mathbf{v}_B + z\mathbf{w}_B + w\mathbf{Q}_B \quad \text{（公式 3.8）}$$

当 *w* 设为 0 时，该方程可用于处理向量的坐标转换变换；当 *w* 设为 1 时，该方程可用于处理点的坐标转换变换。方程 3.8 的优点在于只要我们给出正确的 *w* 坐标，就可以同时完成向量和点的处理；不再需要定义两个方程（一个用于向量，另一个用于点）。方程 2.3 说明，我们可以用矩阵的形式来表示方程 3.8：

$$[x',y',z',w]=[x,y,z,w]\begin{bmatrix}\leftarrow \mathbf{u}_B \rightarrow\\ \leftarrow \mathbf{v}_B \rightarrow\\ \leftarrow \mathbf{w}_B \rightarrow\\ \leftarrow \mathbf{Q}_B \rightarrow\end{bmatrix}$$

$$=[x,y,z,w]\begin{bmatrix}u_x & u_y & u_z & 0\\ v_x & v_y & v_z & 0\\ w_x & w_y & w_z & 0\\ Q_x & Q_y & Q_z & 1\end{bmatrix}$$

（公式 3.9）

其中 $\mathbf{Q}_B=(Q_x,Q_y,Q_z,1)$、$\mathbf{u}_B=(u_x,u_y,u_z,0)$、$\mathbf{v}_B=(v_x,v_y,v_z,0)$、$\mathbf{w}_B=(w_x,w_y,w_z,0)$均为齐次坐标，它们描述了参考系 A 相对于参考系 B 的原点位置和坐标轴方向。我们将方程 3.9 中的 4×4 矩阵称为坐标转换矩阵或参考系转换矩阵，它可以将参考系 *A* 的坐标转换（或映射）为参考系 *B* 的坐标。

## 3.4.4 结合律与坐标转换矩阵

假设现在我们有 3 个参考系 *F*、*G*、*H*。而且，设 **A** 为从 *F* 到 *G* 的参考系转换矩阵，设 **B** 为从 *G* 到 *H* 的参考系转换矩阵。假设 $\mathbf{p}_F$ 为参考系 *F* 中的一个向量坐标，我们想要求解这个向量相对于参考系 *H* 的坐标，也就是求解 $\mathbf{p}_H$。一种方法是将每个转换矩阵逐一相乘：

$$(\mathbf{p}_F\mathbf{A})\mathbf{B}=\mathbf{p}_H$$
$$(\mathbf{p}_G)\mathbf{B}=\mathbf{p}_H$$

不过，由于矩阵乘法支持结合律，所以我们可以将$(\mathbf{p}_F\mathbf{A})\mathbf{B}=\mathbf{p}_H$改写为：

$$\mathbf{p}_F(\mathbf{AB})=\mathbf{p}_H$$

从这一意义上说，矩阵乘积 **C=AB** 可以被看成是从 *F* 直接向 *H* 的参考系转换矩阵；它将 **A** 和 **B** 产生的结果组合成了一个净矩阵。（这就像是把多个函数组合起来一样。）

这种方法有助于提升性能。比如，我们将两个连续的坐标转换变换应用于一个由 20,000 个点构成的 3D 物体。使用逐一相乘的方式，我们需要执行 20,000×2 次向量-矩阵乘法。而改用组合矩阵方式， 我们只需要执行 20,000 次向量-矩阵乘法和一次矩阵-矩阵乘法。很明显，一次额外的矩阵-矩阵乘法所产生的资源消耗微乎其微，而它们却能省去大量的向量-矩阵乘法。

**注意**：我们再次强调，矩阵乘法不支持交换律，**AB** 和 **BA** 表示不同的组合变换。矩阵相乘的顺序就是应用变换的顺序，这通常是一个不可交换的过程。

## 3.4.5 逆矩阵与坐标转换矩阵

假设 $\mathbf{p}_B$ 为向量 **p** 在参考系 *B* 中的坐标，**M** 为参考系 *A* 到参考系 *B* 的坐标转换矩阵；也就是 $\mathbf{p}_B=\mathbf{p}_A\mathbf{M}$。现在我们想求解 $\mathbf{p}_A$。换句话说，就是使用坐标转换矩阵把从 *A* 到 *B* 的映射反转为从 *B* 到 *A* 的映射。如果 **M** 是一个可逆矩阵（即，$\mathbf{M}^{-1}$ 存在），那么就可以求出结果。我们可以按照如下步骤求解 $\mathbf{p}_A$：

| $\mathbf{p}_B=\mathbf{p}_A\mathbf{M}$ | |
|---|---|

| | |
|---|---|
| $\mathbf{p}_B\mathbf{M}^{-1}=\mathbf{p}_A\mathbf{M}\mathbf{M}^{-1}$ | 等式两边同时乘以 $\mathbf{M}^{-1}$。 |
| $\mathbf{p}_B\mathbf{M}^{-1}=\mathbf{p}_A\mathbf{I}$ | 由逆矩阵的定义可知 $\mathbf{M}\mathbf{M}^{-1}=\mathbf{I}$。 |
| $\mathbf{p}_B\mathbf{M}^{-1}=\mathbf{p}_A$ | 由单位矩阵的定义可知 $\mathbf{p}_A\mathbf{I}=\mathbf{p}_A$。 |

这样，矩阵 $\mathbf{M}^{-1}$ 即为从 *B* 到 *A* 的坐标转换矩阵。

图 3.13 说明了坐标转换矩阵与它的逆矩阵之间的关系。另外请读者注意，在本书中使用的所有坐标系转换映射都是可逆的，所以我们不必担心其逆矩阵是否存在。

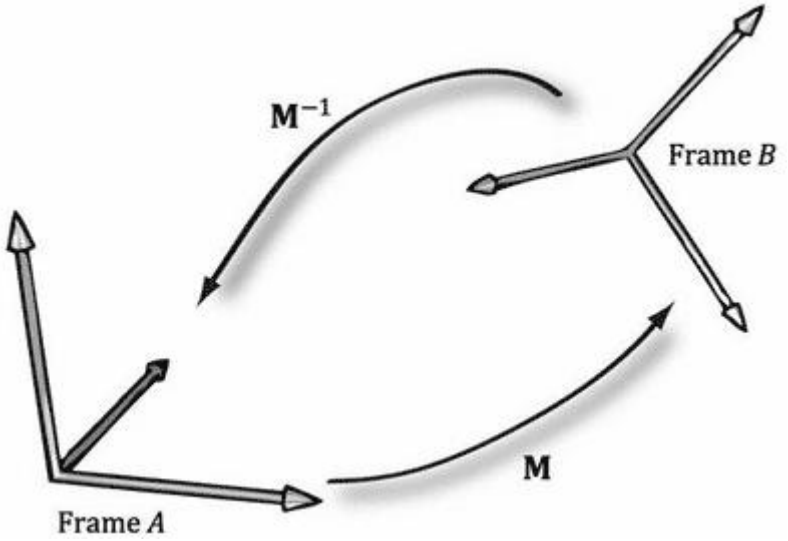



**图 3.13 M 为从 A 到 B 的映射，$\mathbf{M}^{-1}$ 为从 B 到 A 的映射。**

图 3.14 说明了逆矩阵的特性$(\mathbf{AB})^{-1}=\mathbf{B}^{-1}\mathbf{A}^{-1}$ 可以被用于任何坐标转换矩阵。

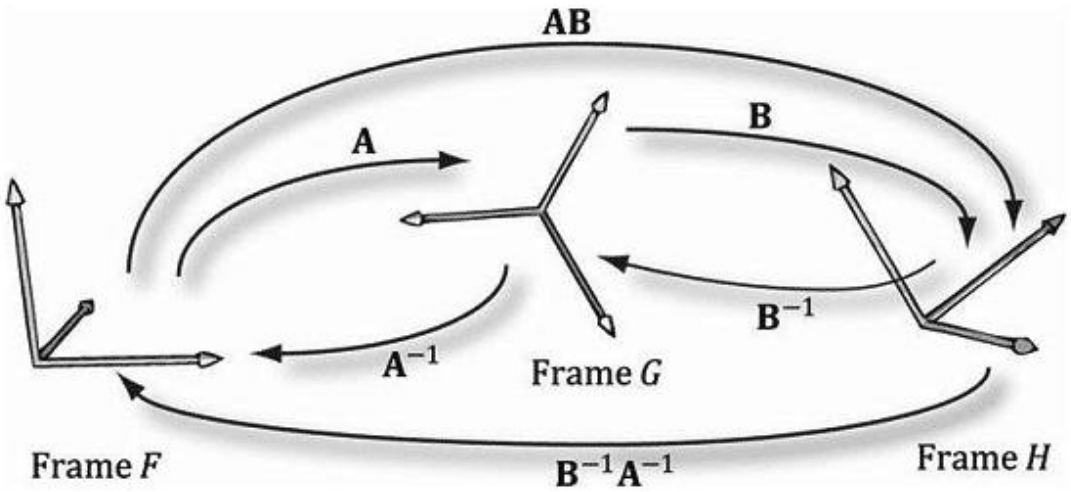



**图 3.14 A 为从 *F* 到 *G* 的映射，B 为从 *G* 到 *H* 的映射，AB 为从 *F* 直接到 *H* 的映射，$\mathbf{B}^{-1}$ 为从 *H* 到 *G* 的映射，$\mathbf{A}^{-1}$ 为从 *G* 到 *F* 的映射，$\mathbf{B}^{-1}\mathbf{A}^{-1}$ 为从 *H* 直接到 *F* 的映射。**

# 3.5 转换矩阵与坐标转换矩阵的对比

在前面的几节中，我们已经区分了 active 变换（缩放、旋转和平移）和坐标转换变换。本节我们将会证明两者在数学上是等价的，一个 active 变换可以解释为一个坐标转换变换，反之亦然。

图 3.15 说明了公式 3.7 中的行向量（仿射矩阵变换实现的平移加旋转）与公式 3.9 中的

行向量（坐标转换矩阵）的相似之处。

这是说得通的。在坐标转换变换的情况中，参考系的位置和朝向都是不同的。所以，将一个参考系变换到另一个参考系的数学方程需要旋转和平移坐标，最终我们会得到相同的数学形式。无论哪种情况，我们都会得到相同的数字；区别只是在于我们解释变换的方式。在某些情况下，使用多个坐标系统更符合思维习惯，我们可以让物体自身保持不变，只是从一个参考系转换到另一个参考系，由于参考系发生了改变，因此物体的坐标也会随之改变（这种情况对应图 3.15b）。在另一些情况中，我们不想改变参考系，而只想在同一个参考系中对物体进行变换（这种情况对于图 3.15a）。

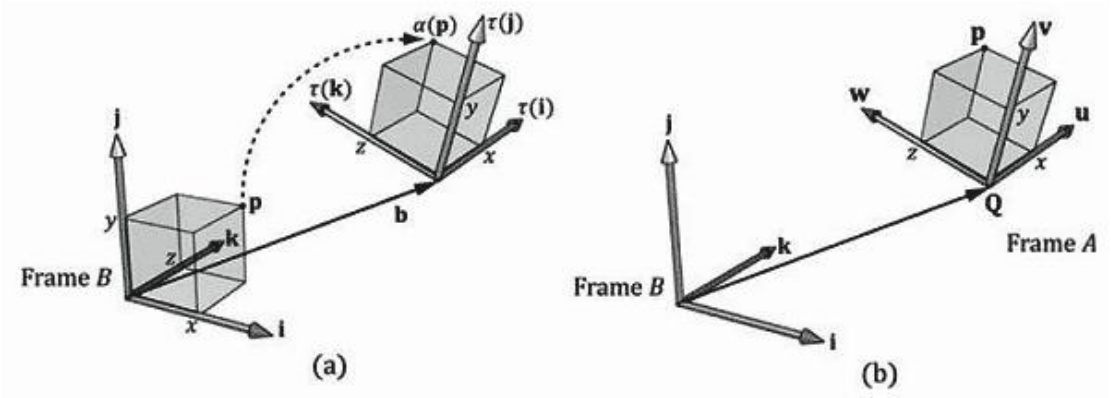



**图 3.15 从图中我们看到 b = Q，τ (i) = u，τ (j) = v，τ (k) = w。（a）中只使用一个坐标系统 *B*，我们在立方体上施加了一个仿射变换：α (*x*, *y*, *z*, w) = *x*τ (i) + *y*τ (j) + *z*τ (k) + *w*b，改变了它相对于坐标系 B 的位置和朝向；(b)中使用了两个坐标系 A 和 B。通过公式 $p_B = xu_B + y\,v_B + z\,w_B + w\,Q_B$（其中 $p_A = (x, y, z, w)$），立方体的各点的坐标可以从参考系 *A* 转换到参考系 *B*。从以上两种情况中我们可以得出α(p) = (*x*′, *y*′, *z*′, w) = $p_B$（$p_B$ 为相对于参考系 B 的坐标）。**

**注意**：上述讨论表明，我们可以将一个 active 变换组合（缩放，旋转，平移）解释为坐标系转换。这一点很重要，因为我们常常会将世界空间（第 5 章）的坐标变换矩阵定义为一个缩放、旋转、平移变换的组合。

# 3.6 XNA 数学库中的转换函数

下面我们总结一下 XNA 数学库中有关变换的函数：

```
// 创建一个缩放矩阵:
XMMATRIX XMMatrixScaling(
    FLOAT ScaleX,
    ScaleY,
    FLOAT ScaleZ);    // 缩放因子

// 从向量中的分量中创建一个缩放矩阵:
XMMATR IX XMMatrixScalingFromVector(
    FXMVECTOR Scale); // 缩放因子(sx,sy,sz)

// 创建一个绕 x 轴旋转的矩阵 : Rx
XMMATRIX XMMatrixRotationX(
    FLOAT Angle); // 顺时针的旋转角度θ
```

```
// 创建一个绕 y 轴旋转的矩阵：Ry
XMMATRIX XMMatrixRotationY(
    FLOAT Angle); // 顺时针的旋转角度θ

// 创建一个绕 z 轴旋转的矩阵：Rz
XMMATRIX XMMatrixRotationZ(
    FLOAT Angle); // 顺时针的旋转角度θ

// 创建一个绕任意轴旋转的矩阵 ：Rn
XMMATRIX XMMatrixRotationAxis(
    FXMVECTOR Axis,     // 旋转轴
    FLOAT Angle);       // 顺时针的旋转角度θ

// 创建一个平移矩阵：
XMMATRIX XMMatrixTranslation(
    FLOAT OffsetX,
    FLOAT OffsetY,
    FLOAT OffsetZ); // 平移因子

// 从向量的分量中创建一个平移矩阵：
XMMATRIX XMMatrixTranslationFromVector(
    FXMVECTOR Offset); // 平移因子(tx , ty ,tz)

// 计算向量-矩阵乘积 vM：
XMVECTOR XMVector3Transform(
    FXMVECTOR V,    // Input v
    CXMMATRIX M);   // Input M

// 计算向量-矩阵乘积 vM，其中 vw = 1，用于变换点的坐标：
XMVECTOR XMVector3TransformCoord(
    FXMVECTOR V,   // Input v
    CXMMATRIX M);  // Input M

// 计算向量-矩阵乘积 vM，其中 vw = 0，用于变换向量：
XMVECTOR XMVector3TransformNormal(
    FXMVECTOR V,     // Input v
    CXMMATRIX M);    // Input M
```

对于最后两个函数 **XMVector3TransformCoord** 和 **XMVector3TransformNormal**，你无需显式地指定 *w* 坐标。**XMVector3TransformCoord** 总是令使用 $v_w = 1$，而 **XMVector3TransformNormal** 总是令 $v_w = 0$。

# 3.7 小结

**1**．基本变换矩阵——缩放、旋转和平移——如下：

$$\mathbf{S}=\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{T}=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ b_x & b_y & b_z & 1 \end{bmatrix}$$

$$\mathbf{R}_n=\begin{bmatrix} c+(1-c)x^2 & (1-c)xy+sz & (1-c)xz-sy & 0 \\ (1-c)xy-sz & c+(1-c)y^2 & (1-c)yz+sx & 0 \\ (1-c)xz+sy & (1-c)yz-sx & c+(1-c)z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**2**．我们用 4×4 矩阵来描述变换，用 1×4 齐次坐标来描述点和向量，其中第 4 个分量 *w* 设为 1 时表示点，*w* 设为 0 时表示向量。通过这一方式，平移只会应用于点，而不会应用于向量。

**3**．如果一个矩阵的所有行向量都是单位向量且相互垂直，则该矩阵为正交矩阵。正交矩阵有一个特殊的性质，它的逆矩阵与转置矩阵相等，因此我们可以很容易地计算出它的逆矩阵。所有的旋转矩阵都是正交矩阵。

**4**．由矩阵乘法的结合律可知，我们可以将多个变换矩阵组合为一个净变换矩阵，最终得到的变换结果与执行多次单个矩阵乘法的变换结果相同。

**5**．设 $\mathbf{Q}_B$、$\mathbf{u}_B$、$\mathbf{v}_B$、$\mathbf{w}_B$ 分别表示参考系 *A* 相对于参考系 *B* 的原点位置及 *x*、*y*、*z* 坐标轴方向。如果向量或点 **p** 相对于参考系 *A* 的坐标为 $\mathbf{p}_A=(x,y,z)$，那该向量相对于参考系 *B* 的坐标为：

$\mathbf{p}_B=(x',y',z')=x\mathbf{u}_B+y\mathbf{v}_B+z\mathbf{w}_B$　　用于向量（方向和大小）

$\mathbf{p}_B=(x',y',z')=\mathbf{Q}_B+x\mathbf{u}_B+y\mathbf{v}_B+z\mathbf{w}_B$　用于位置向量（点）

使用齐次坐标可以将些坐标转换变换改写为矩阵的形式。

**6**．假设我们现在有 3 个参考系 *F*、*G*、*H*，并且，设 **A** 为从 *F* 到 *G* 的参考系转换矩阵，设 **B** 为从 *G* 到 *H* 的参考系转换矩阵。使用矩阵-矩阵乘法，矩阵 **C**=**AB** 可以被视为从 *F* 直接到 *H* 的参考系变换矩阵；也就是，矩阵-矩阵乘法将 **A** 和 **B** 所产生的变换结果组合成了一个净矩阵，记作：$\mathbf{p}_F(\mathbf{AB})=\mathbf{p}_H$。

**7**．如果矩阵 **M** 将参考系 *A* 的坐标映射为参考系 *B* 的坐标，那么矩阵 $\mathbf{M}^{-1}$ 可以将参考系 *B* 的坐标映射为参考系 *A* 的坐标。

**8**．一个 active 变换也可以理解为一个坐标系转换变换，反之亦然。在某些情况下，使用多个坐标系统更符合思维习惯，我们可以让物体自身保持不变，只是从一个参考系转换到另一个参考系，由于参考系发生了改变，因此物体的坐标也会随之改变。在另一些情况中，我们不想改变参考系，而只想在同一个参考系中对物体进行变换。

# 4.1 准备工作

Direct3D 的初始化过程要求我们熟悉一些基本的 Direct3D 类型和基本绘图概念；本章第一节会向读者介绍些必要的基础知识。然后我们会详细讲解 Direct3D 初始化过程中的每一个必要步骤，并顺便介绍一下实时绘图应用程序必须使用的精确计时和时间测量。最后，我们将讨论一下示例框架代码，它是本书所有演示程序使用的统一编程接口。

## 学习目标：

- 对 Direct3D 在规划调度 3D 硬件方面所起的作用有一个基本了解。
- 理解 COM 在 Direct3D 运行时起到的作用。
- 学习基本的绘图概念，比如 2D 图像的存储方式、页面翻转、深度缓存和多重采样。
- 学习如何使用性能计数器函数来获取高精度的计时器读数。
- 阐述 Direct3D 的初始化过程。
- 熟悉本书所有演示程序采用的通用应用程序框架结构。

Direct3D 的初始化过程要求我们熟悉一些基本的绘图概念和 Direct3D 类型。我们会在本节讲解这些概念和类型，以使读者可以顺利地阅读之后的章节。

### 4.1.1 Direct3D 概述

Direct3D 是一种底层绘图 **API（application programming interface，应用程序接口）**，它可以让我们可以通过 3D 硬件加速绘制 3D 世界。从本质上讲，Direct3D 提供的是一组软件接口，我们可以通过这组接口来控制绘图硬件。例如，要命令绘图设备清空渲染目标（例如屏幕），我们可以调用 Direct3D 的 **ID3D11DeviceContext::ClearRenderTargetView** 方法来完成这一工作。Direct3D 层位于应用程序和绘图硬件之间，这样我们就不必担心 3D 硬件的实现细节，只要设备支持 Direct3D 11，我们就可以通过 Direct3D 11 API 来控制 3D 硬件了。

支持 Direct3D 11 的设备必须支持 Direct3D 11 规定的整个功能集合以及少数的额外附加功能（有一些功能，比如多重采样数量，仍然需要以查询方式实现，这是因为不同的 Direct3D 硬件这个值可能并不一样）。在 Direct3D 9 中，设备可以只支持 Direct3D 9 的部分功能；所以，当一个 Direct3D 9 应用程序要使用某一特性时，应用程序就必须先检查硬件是否支持该特性。如果要调用的是一个不为硬件支持 Direct3D 函数，那应用程序就会出错。而在 Direct3D 11 中，不需要再做这种设备功能检查，因为 Direct3D 11 强制要求设备实现 Direct3D 11 规定的所有功能特性。

### 4.1.2 COM

组件对象模型（COM）技术使 DirectX 独立于任何编程语言，并具有版本向后兼容的特性。我们经常把 COM 对象称为接口，并把它当成一个普通的 C++类来使用。当使用 C++编写 DirectX 程序时，许多 COM 的底层细节都不必考虑。唯一需要知道的一件事情是，我们必须通过特定的函数或其他的 COM 接口方法来获取指向 COM 接口的指针，而不能用 C++

的 **new** 关键字来创建 COM 接口。另外，当我们不再使用某个接口时，必须调用它的 **Release** 方法来释放它（所有的 COM 接口都继承于 **IUnknown** 接口，而 **Release** 方法是 **IUnknown** 接口的成员），而不能用 **delete** 语句——COM 对象在其自身内部实现所有的内存管理工作。

当然，有关 COM 的细节还有很多，但是在实际工作中只需知道上述内容就足以有效地使用 DirectX 了。

**注意**：COM 接口都以大写字母“I”为前缀。例如，表示 2D 纹理的接口为 **ID3D11Texture2D**。

### 4.1.3 纹理和数据资源格式

2D 纹理（texture）是一种数据元素矩阵。2D 纹理的用途之一是存储 2D 图像数据，在纹理的每个元素中存储一个像素颜色。但这不是纹理的唯一用途；例如， 有一种称为法线贴图映射（normal mapping）的高级技术在纹理元素中存储的不是颜色，而是 3D 向量。因此，从通常意义上讲，纹理用来存储图像数据，但是在实际应用中纹理可以有更广泛的用途。1D 纹理类似于一个 1D 数据元素数组，3D 纹理类似于一个 3D 数据元素数组。但是在随后的章节中我们会讲到，纹理不仅仅是一个数据数组；纹理可以带有多级渐近纹理层（mipmap level），GPU 可以在纹理上执行特殊运算，比如使用过滤器（filter）和多重采样（multisampling）。此外，不是任何类型的数据都能存储到纹理中的；纹理只支持特定格式的数据存储，这些格式由 **DXGI_FORMAT** 枚举类型描述。一些常用的格式如下：

- **DXGI_FORMAT_R32G32B32_FLOAT**：每个元素包含 3 个 32 位浮点分量。
- **DXGI_FORMAT_R16G16B16A16_UNORM**：每个元素包含 4 个 16 位分量，分量的取值范围在[0,1]区间内。
- **DXGI_FORMAT_R32G32_UINT**：每个元素包含两个 32 位无符号整数分量。
- **DXGI_FORMAT_R8G8B8A8_UNORM**：每个元素包含 4 个 8 位无符号分量，分量的取值范围在[0,1]区间内。
- **DXGI_FORMAT_R8G8B8A8_SNORM**：每个元素包含 4 个 8 位有符号分量，分量的取值范围在[−1,1] 区间内。
- **DXGI_FORMAT_R8G8B8A8_SINT**：每个元素包含 4 个 8 位有符号整数分量，分量的取值范围在[−128, 127] 区间内。
- **DXGI_FORMAT_R8G8B8A8_UINT**：每个元素包含 4 个 8 位无符号整数分量，分量的取值范围在[0, 255]区间内。

注意，字母 R、G、B、A 分别表示 red（红）、green（绿）、blue（蓝）和 alpha（透明度）。每种颜色都是由红、绿、蓝三种基本颜色组成的（例如，黄色是由红色和绿色组成的）。alpha 通道（或 alpha 分量）用于控制透明度。不过，正如我们之前所述，纹理存储的不一定是颜色信息；例如，格式 **DXGI_FORMAT_R32G32B32_FLOAT** 包含 3 个浮点分量，可以存储一个使用浮点坐标的 3D 向量。另外，还有一种**弱类型（typeless）**格式，可以预先分配内存空间，然后在纹理绑定到管线时再指定如何重新解释数据内容（这一过程与 C++ 中的数据类型转换颇为相似）；例如，下面的弱类型格式为每个元素预留 4 个 8 位分量，且不指定数据类型（例如：整数、浮点数、无符号整数）：

**DXGI_FORMAT_R8G8B8A8_TYPELESS**

### 4.1.4 交换链和页面翻转

为了避免在动画中出现闪烁，最好的做法是在一个离屏（off-screen）纹理中执行所有的动画帧绘制工作，这个离屏纹理称为**后台缓冲区（back buffer）**。当我们在后台缓冲区中完成给定帧的绘制工作后，便可以将后台缓冲区作为一个完整的帧显示在屏幕上；使用这种方法，用户不会察觉到帧的绘制过程，只会看到完整的帧。从理论上讲，将一帧显示到屏幕上所消耗的时间小于屏幕的垂直刷新时间。硬件会自动维护两个内置的纹理缓冲区来实现这一功能，这两个缓冲区分别称为**前台缓冲区（front buffer）**和后台缓冲区。前台缓冲区存储了当前显示在屏幕上的图像数据，而动画的下一帧会在后台缓冲区中执行绘制。当后台缓冲区的绘图工作完成之后，前后两个缓冲区的作用会发生翻转：后台缓冲区会变为前台缓冲区，而前台缓冲区会变为后台缓冲区，为下一帧的绘制工作提前做准备。我们将前后缓冲区功能互换的行为称做**呈现（presenting）**。提交是一个运行速度很快的操作，因为它只是将前台缓冲区的指针和后台缓冲区的指针做了一个简单的交换。图 4.1 说明了这一过程。

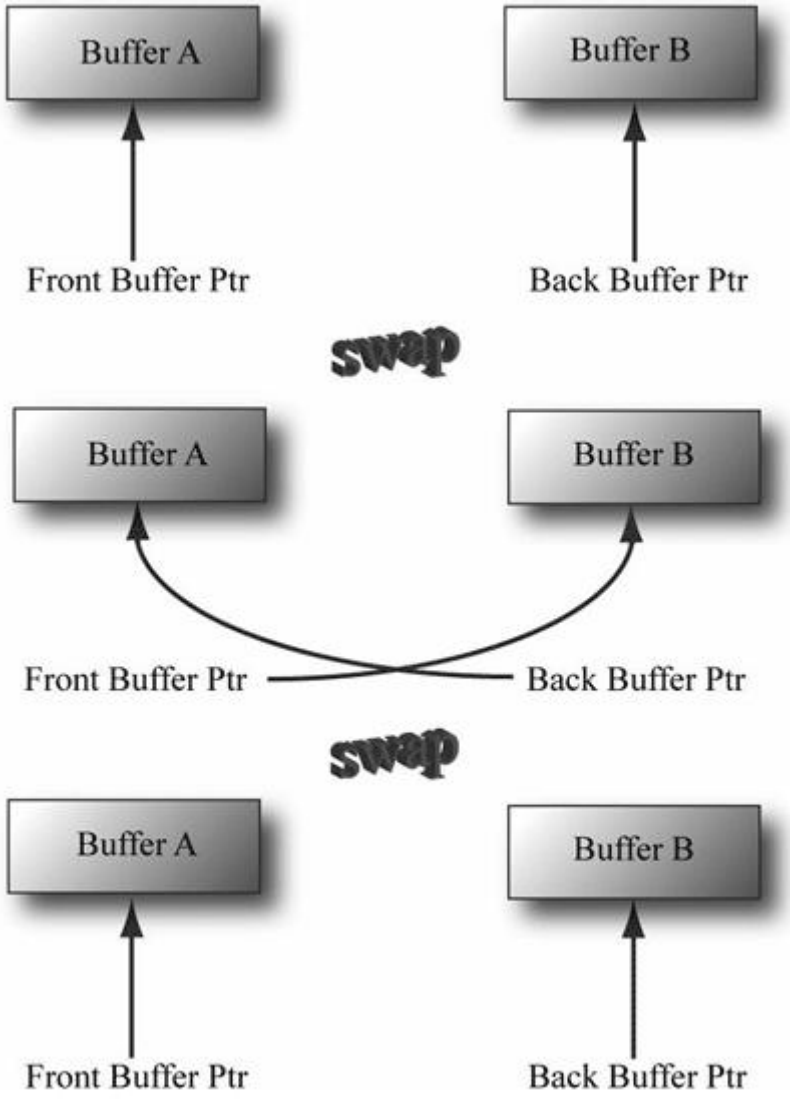



**图 4.1：我们首先渲染缓冲区 B，它是当前的后台缓冲区。一旦帧渲染完成，前后缓冲区的指针会相互交换，缓冲区 B 会变为前台缓冲区，而缓冲区 A 会变为新的后台缓冲区。之后，我们将在缓冲区 A 中进行下一帧的渲染。一旦帧渲染完成，前后缓冲区的指针会再**

**次进行交换，缓冲区 A 会变为前台缓冲区，而缓冲区 B 会再次变为后台缓冲区。**

前后缓冲区形成了一个**交换链（swap chain）**。在 Direct3D 中，交换链由 **IDXGISwapChain** 接口表示。该接口保存了前后缓冲区纹理，并提供了用于调整缓冲区尺寸的方法（**IDXGISwapChain::ResizeBuffers**）和呈现方法（**IDXGISwapChain::Present**）。我们会在 4.4 节中详细讨论些方法。

使用（前后）两个缓冲区称为**双缓冲（double buffering）**。缓冲区的数量可多于两个；比如，当使用三个缓冲区时称为**三缓冲（triple buffering**）。不过，两个缓冲区已经足够用了。

**注意**：虽然后台缓冲区是一个纹理（纹理元素称为 texel），但是我们更习惯于将纹理元素称为像素（pixel），因为后台缓冲区存储的是颜色信息。有时，即使纹理中存储的不是颜色信息，人们还是会将纹理元素称为像素（例如，“法线贴图像素”）。

## 4.1.5 深度缓冲区

**深度缓冲区（depth buffer）**是一个不包含图像数据的纹理对象。在一定程度上，深度信息可以被认为是一种特殊的像素。常见的深度值范围在 0.0 到 1.0 之间，其中 0.0 表示离观察者最近的物体，1.0 表示离观察者最远的物体。深度缓冲区中的每个元素与后台缓冲区中的每个像素一一对应（即，后台缓冲区的第 ij 个元素对应于深度缓冲区的第 ij 个元素）。所以，当后台缓冲区的分辨率为 1280×1024 时，在深度缓冲区中有 1280×1024 个深度元素。

**图 4.2 彼此遮挡的一组物体**

图 4.2 是一个简单的场景，其中一些物体挡住了它后面的一些物体的一部分区域。为了判定物体的哪些像素位于其他物体之前，Direct3D 使用了一种称为**深度缓存（depth buffering）**或 **z 缓存（z-buffering）**的技术。我们所要强调的是在使用深度缓存时，我们不必关心所绘物体的先后顺序。

**注意**：要处理深度的问题，有人可能会建议按照从远至近的顺序绘制场景中的物体。使用这种方法，离得近的物体会覆盖在离得远的物体之上，这样就会产生正确的绘制结果，这也是画家作画时用到的方法。但是，这种方法会导致另一个问题——如何将大量的物体和相

交的几何体按从远到近的方式进行排序？此外，图形硬件本身就提供了深度缓存供我们使用，因此我们不会采用画家算法。

为了说明深度缓存的工作方式，让我们来看一个例子。如图 4.3 所示，它展示的是观察者看到的立体空间（左图）以及该立体空间的 2D 侧视图（右图）。从这个图中我们可以发现，3 个不同的像素会被渲染到视图窗口的同一个像素点 P 上。（当然，我们知道只有最近的像素会被渲染到 P 上，因为它挡住了后面的其他像素，可是计算机不知道这些事情。）首先，在渲染之前，我们必须把后台缓冲区清空为一个默认颜色（比如黑色或白色），把深度缓冲区清空为默认值——通常设为 1.0（像素所具有的最远深度值）。

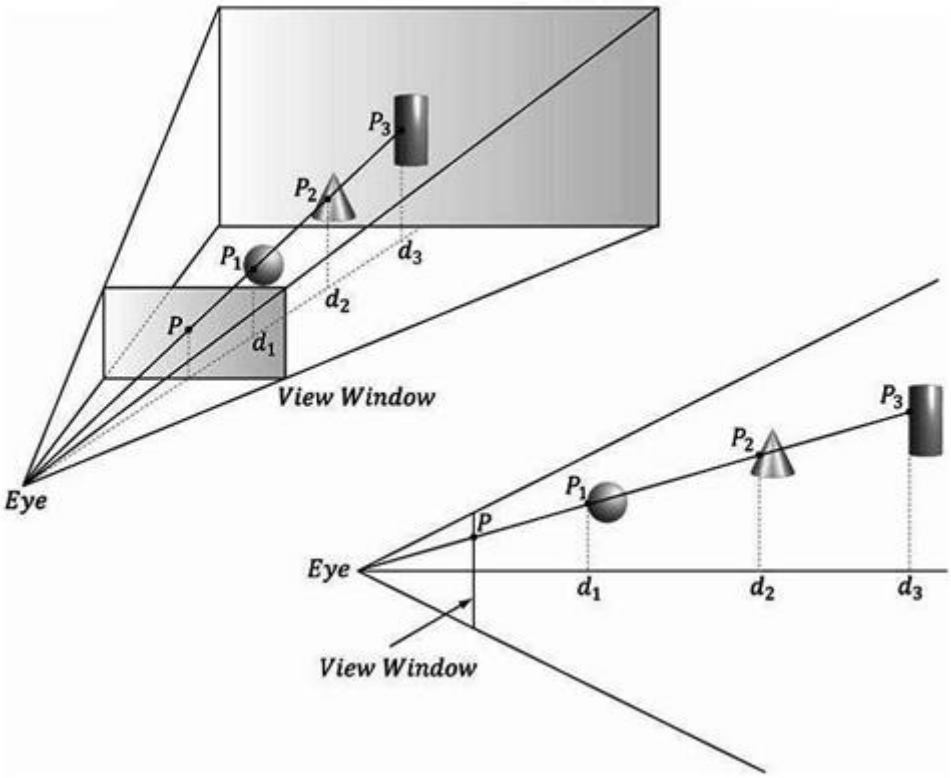



**图 4.3：视图窗口相当于从 3D 场景生成的 2D 图像（后台缓冲区）。我们看到，有 3 个不同的像素可以被投影到像素 P 上。直觉告诉我们，$P_1$ 是 P 的最终颜色，因为它离观察者最近，而且遮挡了其他两个像素。深度缓冲区算法提供了一种可以在计算机上实现的判定过程。注意，我们所说的深度值是相对于观察坐标系而言的。实际上，当深度值存入深度缓冲区时，它会被规范到[0.0,1.0]区间内。**

现在，假设物体的渲染顺序依次为：圆柱体、球体和圆锥体。下面的表格汇总了在绘制些物体时像素 P 及相关深度值的变化过程；其他像素的处理过程与之类似。

**表 4.1**

| 操作 | P | d | 说明 |
|---|---|---|---|
| **清空** | 黑色 | 1.0 | 初始化像素以及相应的深度元素。 |
| **绘制圆柱体** | $P_3$ | $d_3$ | 因为 $d_3 \leqslant d = 1.0$，所以深度测试通过，更新缓冲区，设置 $P = P_3$、$d = d_3$。 |
| **绘制球体** | $P_1$ | $d_1$ | 因为 $d_1 \leqslant d = d_3$，所以深度测试通过，更新缓冲区，设置 $P = P_1$、$d = d_1$。 |
| **绘制圆锥体** | $P_1$ | $d_1$ | 因为 $d_2 > d = d_1$，所以深度测试未通过，不更新缓冲区。 |

从上表可以看到，当我们发现某个像素具有更小的深度值时，就更新该像素以及它在深

度缓冲区中的相应深度值。通过一方式，在最终得到的渲染结果中只会包含那些离观察者最近的像素。（如果读者对此仍有疑虑，那么可以试着交换本例的绘图顺序，看看得到的计算结果是否相同。）

综上所述，深度缓冲区用于为每个像素计算深度值和实现深度测试。深度测试通过比较像素深度来决定是否将该像素写入后台缓冲区的特定像素位置。只有离观察者最近的像素才会胜出，成为写入后台缓冲区的最终像素。这很容易理解，因为离观察者最近的像素会遮挡它后面的其他像素。

深度缓冲区是一个纹理，所以在创建它时必须指定一种数据格式。用于深度缓存的格式如下：

- **DXGI_FORMAT_D32_FLOAT_S8X24_UINT**：32 位浮点深度缓冲区。为模板缓冲区预留 8 位（无符号整数），每个模板值的取值范围为[0,255]。其余 24 位闲置。
- **DXGI_FORMAT_D32_FLOAT**：32 位浮点深度缓冲区。
- **DXGI_FORMAT_D24_UNORM_S8_UINT**：无符号 24 位深度缓冲区，每个深度值的取值范围为[0,1]。为模板缓冲区预留 8 位（无符号整数），每个模板值的取值范围为[0,255]。
- **DXGI_FORMAT_D16_UNORM**：无符号 16 位深度缓冲区，每个深度值的取值范围为[0,1]。

**注意**：模板缓冲区对应用程序来说不是必须的，但是如果用到了模板缓冲区，那么模板缓冲区必定是与深度缓冲区存储在一起的。例如，32 位格式 **DXGI_FORMAT_D24_UNORM_S8_UINT** 使用 24 位用于深度缓冲区，8 位用于模板缓冲区。 所以，将深度缓冲区称为“深度/模板缓冲区”更为合适。模板缓冲区是一个比较高级的主题，我们会在第 10 章讲解模板缓冲区的用法。

### 4.1.6 纹理资源视图

纹理可以被绑定到**渲染管线（rendering pipeline）**的不同**阶段（stage）**；例如，比较常见的情况是将纹理作为渲染目标（即，Direct3D 渲染到纹理）或着色器资源（即，在着色器中对纹理进行采样）。当创建用于这两种目的的纹理资源时，应使用绑定标志值：

**D3D11_BIND_RENDER_TARGET | D3D10_BIND_SHADER_RESOURCE**

指定纹理所要绑定的两个管线阶段。其实，资源不能被直接绑定到一个管线阶段；我们只能把与资源关联的资源视图绑定到不同的管线阶段。无论以哪种方式使用纹理，Direct3D 始终要求我们在初始化时为纹理创建相关的**资源视图（resource view）**。这样有助于提高运行效率，正如 SDK 文档指出的那样：“运行时环境与驱动程序可以在视图创建执行相应的验证和映射，减少绑定时的类型检查”。所以，当把纹理作为一个渲染目标和着色器资源时，我们要为它创建两种视图：渲染目标视图（**ID3D11RenderTargetView**）和着色器资源视图（**ID3D11ShaderResourceView**）。资源视图主要有两个功能：（1）告诉 Direct3D 如何使用资源（即，指定资源所要绑定的管线阶段）；（2）如果在创建资源时指定的是弱类型（typeless）格式，那么在为它创建资源视图时就必须指定明确的资源类型。对于弱类型格式，纹理元素可能会在一个管线阶段中视为浮点数，而在另一个管线阶段中视为整数。

为了给资源创建一个特定视图，我们必须在创建资源时使用特定的绑定标志值。例如，如果在创建资源没有使用 **D3D11_BIND_DEPTH_STENCIL** 绑定标志值（该标志值表示纹理将作为一个深度/模板缓冲区绑定到管线上），那我们就无法为该资源创建 **ID3D11DepthStencilView 视图**。只要你试一下就会发现 Direct3D 会给出如下调试错误：

**ERROR: ID3D11Device::CreateDepthStencilView: A DepthStencilView cannot be**

**created of a Resource that did not specify D3D10_BIND_DEPTH_STENCIL.**

我们会在本章的 4.2 节中看到用来创建渲染目标视图和深度/模板视图的代码。在第 8 章中看到用于创建着色器资源视图的代码。本书随后的许多例子都有会把纹理用作渲染目标和着色器资源。

**注意**：2009 年 8 月的 SDK 文档指出：“当创建资源时，为资源指定强类型（fully-typed）格式，把资源的用途限制在格式规定的范围内，有利于提高运行时环境对资源的访问速度……”。所以，你只应该在真正需要弱类型资源时（使用弱类型的优点是可以使用不同的视图将数据用于不同的用途），才创建弱类型资源；否则，应尽量创建强类型资源。

## 4.1.7 多重采样

因为计算机显示器上的像素分辨率有限，所以当我们绘制一条任意直线时，该直线很难精确地显示在屏幕上。图 4.4 中的第一条直线说明了“阶梯”（**aliasing，锯齿**）效应，当使用像素矩阵近似地表示一条直线时就会出现这种现象，类似的锯齿也会发生在三角形的边缘上。

**图 4.4：我们可以看到，第一条直线带有明显的锯齿（当使用像素矩阵近似地表示一条直线时就会出现阶梯效应）。而第二条直线使用了抗锯齿技术，通过对一个像素周围的邻接像素进行采样得到该像素的最终颜色；这样可以形成一条较为平滑的直线，使阶梯效果得到缓解。**

通过提高显示器的分辨率，缩小像素的尺寸，也可以有效地缓解一问题，使阶梯效应明显降低。

当无法提高显示器分辨率或分辨率不够高时，我们可以使用**抗锯齿（antialiasing）**技术。其中的一种技术叫做**超级采样（supersampling）**，它把后台缓冲和深度缓冲的大小提高到屏幕分辨率的 4 倍。3D 场景会以这个更大的分辨率渲染到后台缓存中，当在屏幕上呈现后台缓冲时，后台缓冲会将 4 个像素的颜色取平均值后得到一个像素的最终颜色。从效果上来说，超级采样的工作原理就是以软件的方式提升分辨率。

超级采样代价昂贵，因为它处理的像素数量和所需的内存数量增加为原来的 4 倍。Direct3D 支持另一种称为**多重采样（multisampling）**的抗锯齿技术，它通过对一个像素的子像素进行采样计算出该像素的最终颜色，比超级采样节省资源。假如我们使用的是 4X 多重采样（每个像素采样 4 个邻接像素），多重采样仍然会使用屏幕分辨率 4 倍大小的后台缓冲和深度缓冲，但是，不像超级采样那样计算每个子像素的颜色，而是只计算像素中心颜色一次，然后基于子像素的可见性（基于子像素的深度/模板测试）和范围（子像素中心在多边形之外还是之内）共享颜色信息。图 4.5 展示了这样的一个例子。

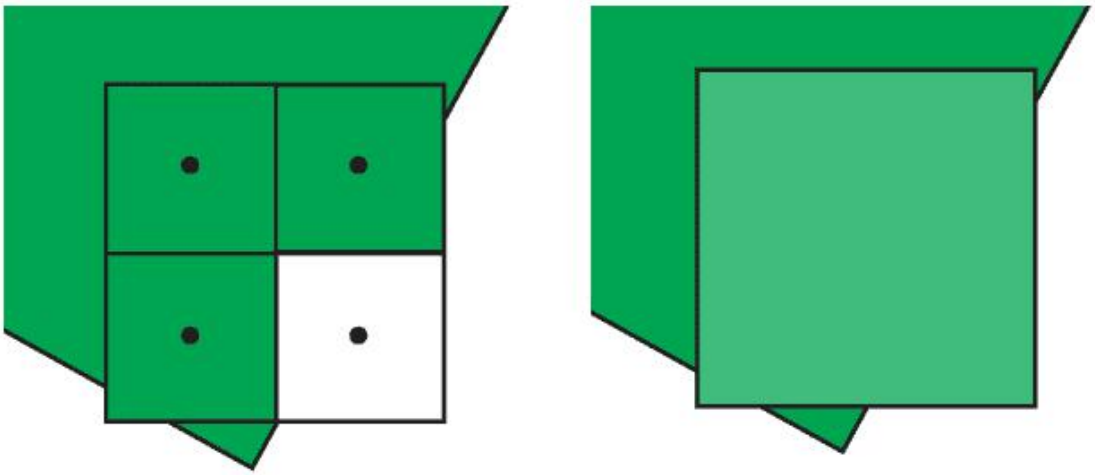


图 4.5：如图（a）所示，一个像素与多边形的边缘相交，像素中心的绿颜色存储在可见的三个子像素中，而第 4 个子像素没有被多边形覆盖，因此不会被更新为绿色，它仍保持为原来绘制的几何体颜色或 **Clear** 操作后的颜色。如图（b）所示，要获得最后的像素颜色，我们需要对 4 个子像素（3 个绿色和一个白色）取平均值，获得淡绿色，通过这个操作，可以减弱多边形边缘的阶梯效果，实现更平滑的图像。

**注意**：supersampling 与 multisampling 的关键区别在于：使用 supersampling 时，图像的颜色需要通过每个子像素的颜色计算得来，而每个子像素颜色可能不同；使用 multisampling（图 4.5）时，每个像素的颜色只计算一次，这个颜色会填充到所有可见的、被多边形覆盖的子像素中，即这个颜色是共享的。因为计算图像的颜色是图形管线中最昂贵的操作之一，因此 multisampling 相比 supersampling 而言节省的资源是相当可观的。但是，supersampling 更为精确，这是 multisampling 做不到的。

**注意**：在图 4.5 中，我们用标准的网格图形表示一个像素的 4 个子像素，但由于硬件的不同，实际的子像素放置图形也是不同的，Direct3D 并不定义子像素的放置方式，在特定情况下，某些放置方式会优于其他的放置方式。

## 4.1.8 Direct3D 中的多重采样

在下一节中， 我们要填充一个 **DXGI_SAMPLE_DESC** 结构体。该结构体包含两个成员，其定义如下：

```
typedef struct DXGI_SAMPLE_DESC {
    UINT Count;
    UINT Quality;
} DXGI_SAMPLE_DESC, *LPDXGI_SAMPLE_DESC;
```

**Count** 成员用于指定每个像素的采样数量，**Quality** 成员用于指定希望得到的质量级别（不同硬件的质量级别表示的含义不一定相同）。质量级别越高，占用的系统资源就越多，所以我们必须在质量和速度之间权衡利弊。质量级别的取值范围由纹理格式和单个像素的采样数量决定。我们可以使用如下方法，通过指定纹理格式和采样数量来查询相应的质量级别：

```
HRESULT ID3D11Device::CheckMultisampleQualityLevels(
DXGI_FORMAT Format, UINT SampleCount, UINT *pNumQualityLevels);
```

如果纹理格式和采样数量的组合不被设备支持，则该方法返回 0。反之，通过

pNumQualityLevels 参数返回符合给定的质量等级数值。有效的质量级别范围为 0 到 pNumQualityLevels−1。

采样的最大数量可以由以下语句定义：

```
#define D3D11_MAX_MULTISAMPLE_SAMPLE_COUNT(32)
```

采样数量通常使用 4 或 8，可以兼顾性能和内存消耗。如果你不使用多重采样，可以将采样数量设为 1，将质量级别设为 0。所有符合 Direct3D 11 功能特性的设备都支持用于所有渲染目标格式的 4X 多重采样。

**注意**：我们需要为交换链缓冲区和深度缓冲区各填充一个 **DXGI_SAMPLE_DESC** 结构体。当创建后台缓冲区和深度缓冲区时，必须使用相同的多重采样设置；具体的代码会在下一节给出。

## 4.1.9 特征等级

Direct3D 11 提出了特征等级（feature levels，在代码中由枚举类型 **D3D_FEATURE_LEVEL** 表示）的概念，对应了定义了 d3d11 中定义了如下几个等级以代表不同的 d3d 版本：

```
typedef enum D3D_FEATURE_LEVEL {
  D3D_FEATURE_LEVEL_9_1 = 0x9100,
  D3D_FEATURE_LEVEL_9_2 = 0x9200,
  D3D_FEATURE_LEVEL_9_3 = 0x9300,
  D3D_FEATURE_LEVEL_10_0 = 0xa000,
  D3D_FEATURE_LEVEL_10_1 = 0xa100,
  D3D_FEATURE_LEVEL_11_0 = 0xb000
} D3D_FEATURE_LEVEL;
```

特征等级定义了一系列支持不同 d3d 功能的相应的等级(每个特征等级支持的功能可参见 SDK 文档),用意即如果一个用户的硬件不支持某一特征等级,程序可以选择较低的等级。例如，为了支持更多的用户，应用程序可能需要支持 Direct3D 11，10.1，9.3 硬件。程序会从最新的硬件一直检查到最旧的,即首先检查是否支持 Direct3D 11,第二检查 Direct3D 10.1，然后是 Direct3D 10，最后是 Direct3D 9。要设置测试的顺序，可以使用下面的特征等级数组（数组内元素的顺序即特征等级测试的顺序）：

```
D3D_FEATURE_LEVEL featureLevels [4] =
{
  D3D_FEATURE_LEVEL_11_0,   //  First check D3D 11 support
  D3D_FEATURE_LEVEL_10_1,   //  Second check D3D 10.1 support
  D3D_FEATURE_LEVEL_10_0,   //  Next,check D3D 10 support
  D3D_FEATURE_LEVEL_9_3    //  Finally,check D3D 9.3 support
} ;
```

这个数组可以放置在 Direct3D 初始化方法（4.2.1 节）中，方法会输出数组中第一个可被支持的特征等级。例如，如果 Direct3D 报告数组中第一个可被支持的特征等级是 **D3D_FEATURE_LEVEL_10_0**，程序就会禁用 Direct3D 11 和 Direct3D 10.1 的特征，而使用 Direct3D 10 的绘制路径。本书中我们要求必须能支持 **D3D_FEATURE_LEVEL_11_0**。

# 4.2 对 Direct3D 进行初始化

下面的各小节将讲解如何初始化 Direct3D。我们将 Direct3D 的初始化过程分为如下几个步骤：

9．使用 **D3D11CreateDevice** 方法创建 **ID3D11Device** 和 **ID3D11DeviceContext**。

10．使用 **ID3D11Device::CheckMultisampleQualityLevels** 方法检测设备支持的 4X 多重采样质量等级。

11．填充一个 **IDXGI_SWAP_CHAIN_DESC** 结构体，该结构体描述了所要创建的交换链的特性。

12．查询 **IDXGIFactory** 实例，这个实例用于创建设备和一个 **IDXGISwapChain** 实例。

13．为交换链的后台缓冲区创建一个渲染目标视图。

14．创建深度/模板缓冲区以及相关的深度/模板视图。

15．将渲染目标视图和深度/模板视图绑定到渲染管线的输出合并阶段，使它们可以被 Direct3D 使用。

16．设置视口。

## 4.2.1 创建设备（Device）和上下文（Context）

要初始化 Direct3D，首先需要创建 Direct3D 11 设备（**ID3D11Device**）和上下文（**ID3D11DeviceContext**）。它们是是最重要的 Direct3D 接口，可以被看成是物理图形设备硬件的软控制器；也就是说，我们可以通过该接口与硬件进行交互，命令硬件完成一些工作（比如：在显存中分配资源、清空后台缓冲区、将资源绑定到各种管线阶段、绘制几何体）。具体而言：

3．**ID3D11Device** 接口用于检测显示适配器功能和分配资源。

4．**ID3D11DeviceContext** 接口用于设置管线状态、将资源绑定到图形管线和生成渲染命令。

设备和上下文可用如下函数创建：

```
HRESULT  D3D11CreateDevice (
    IDXGIAdapter  *pAdapter,
    D3D_DRIVER_TYPE  DriverType,
    HMODULE  Software ,
    UINT  Flags ,
    CONST  D3D_FEATURE_LEVEL  *pFeatureLevels ,
    UINT  FeatureLevels ,
    UINT  SDKVersion,
    ID3D11Device  **ppDevice ,
    D3D_FEATURE_LEVE L  *pFeatureLevel,
    ID3D11DeviceContext  **ppImmediateContext
);
```

1．**pAdapter**：指定要为哪个物理显卡创建设备对象。当该参数设为空值时，表示使用主显卡。在本书的示例程序中，我们只使用主显卡。

2．**DriverType**：一般来讲，该参数总是指定为 **D3D_DRIVER_TYPE_HARDWARE**，

表示使用 3D 硬件来加快渲染速度。但是，也可以有两个其他选择：

**D3D_DRIVER_TYPE_REFERENCE**：创建所谓的引用设备（reference device）。引用设备是 Direct3D 的软件实现，它具有 Direct3D 的所有功能（只是运行速度非常慢，因为所有的功能都是用软件来实现的）。引用设备随 DirectX SDK 一起安装，只用于程序员，而不应该用于程序发布。使用引用设备有两个原因：

- 测试硬件不支持的代码；例如，在一块不支持 Direct3D 11 的显卡上测试一段 Direct3D 11 的代码。
- 测试驱动程序缺陷。当代码能在引用设备上正常运行，而不能在硬件上正常工作时，说明硬件的驱动程序可能存在缺陷。

**D3D_DRIVER_TYPE_SOFTWARE**：创建一个用于模拟 3D 硬件的软件驱动器。要使用软件驱动器，你必须自己创建一个，或使用第三方的软件驱动器。与下面要说的 WARP 驱动器不同，Direct3D 不提供软件驱动器。

**D3D_DRIVER_TYPE_WARP**：创建一个高性能的 Direct3D 10.1 软件驱动器。WARP 代表 Windows Advanced Rasterizati on Platform。因为 WARP 不支持 Direct3D 11，因此我们对它不感兴趣。

3．**Software**：用于支持软件光栅化设备（software rasterizer）。我们总是将该参数设为空值，因为我们使用硬件进行渲染。如果读者想要使用这一功能，那么就必须先安装一个软件光栅化设备。

4．**Flags**：可选的设备创建标志值。当以 release 模式生成程序时，该参数通常设为 0（无附加标志值）；当以 debug 模式生成程序时，该参数应设为：

**D3D11_CREATE_DEVICE_DEBUG**：用以激活调试层。当指定调试标志值后，Direct3D 会向 VC++的输出窗口发送调试信息；图 4.6 展示了输出错误信息的一个例子。

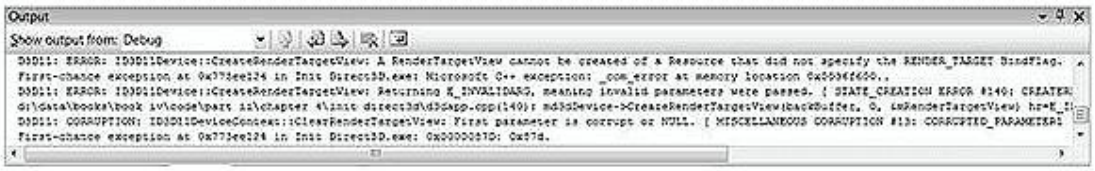



**图 4.6　Direct3D 11 调试输出的一个例子。**

5．**pFeatureLevels**：**D3D_FEATURE_LEVEL** 数组，元素的顺序表示要特征等级（见§4.1.9）的测试顺序。将这个参数设置为 null 表示选择可支持的最高等级。

6．**FeatureLevels**：**pFeatureLevels** 数组中的元素 **D3D_FEATURE_LEVELs** 的数量，若 **pFeatureLevels** 设置为 null，则这个值为 0。

7．**SDKVersion**：始终设为 **D3D11_SDK_VERSION**。

8．**ppDevice**：返回创建后的设备对象。

9．**pFeatureLevel**：返回 **pFeatureLevels** 数组中第一个支持的特征等级（如果 **pFeatureLevels** 为 null，则返回可支持的最高等级）。

10．**ppImmediateContext**：返回创建后的设备上下文。

下面是调用该函数的一个示例：

```
UINT createDeviceFlags = 0;

#if  defined(DEBUG)||defined(_DEBUG)
   createDeviceFlags  |= D3D11_CREATE_DEVICE_DEBUG;
```

```
#endif

D3D_FEATURE_LEVEL featureLevel;
ID3D11Device *  md3dDevice;
ID3D11Device Context*  md3dImmediate Context;
HRESULT  hr = D3D11CreateDevice(
    0,                  //  默认显示适配器
    D3D_DRIVER_TYPE_HARDWARE ,
    0,                  //  不使用软件设备
    createDeviceFlags ,
    0, 0,              //  默认的特征等级数组
    D3D11_SDK_VERSION,
    &  md3dDevice ,
    & featureLevel,
    &  md3dImmediateContext);
if(FAILED(hr) )
{
    MessageBox(0, L"D3D11CreateDevice Failed.", 0, 0);
    return  false ;
}
if(featureLevel != D3D_FEATURE_LEVEL_11_0)
{
    MessageBox(0, L"Direct3D FeatureLevel 11 unsupported.", 0, 0);
    return  false;
}
```

从上面的代码可以看到我们使用的是立即执行上下文（immediate context）：

**ID3D11DeviceContext* md3dImmediateContext;**

还有一种上下文叫做延迟执行上下文（**ID3D11Device::CreateDeferredContext**）。该上下文主要用于 Direct3D 11 支持的多线程程序。多线程编程是一个高级话题，本书并不会介绍，但下面介绍一点基本概念：

1．在主线程中使用立即执行上下文。

2．在工作线程总使用延迟执行上下文。

（a）每个工作线程可以将图形指令记录在一个命令列表（**ID3D11CommandList**）中。

（b）随后，每个工作线程中的命令列表可以在主渲染线程中加以执行。

在多核系统中，可并行处理命令列表中的指令，这样可以缩短编译复杂图形所需的时间。

## 4.2.2 检测 4X 多重采样质量支持

创建了设备后，我们就可以检查 4X 多重采样质量等级了。所有支持 Direct3D 11 的设备都支持所有渲染目标格式的 4X MSAA（支持的质量等级可能并不相同）。

```
UINT  m4xMsaaQuality;
HR(md3dDevice ->CheckMultisampleQualityLevels(
    DXGI_FORMAT_R8G8B8A8_UNORM, 4, &  m4xMsaaQuality));
```

```
assert(m4xMsaaQuality > 0);
```

因为 4X MSAA 总是被支持的，所以返回的质量等级总是大于 0。

### 4.2.3 描述交换链

下一步是创建交换链，首先需要填充一个 **DXGI_SWAP_CHAIN_DESC** 结构体来描述我们将要创建的交换链的特性。该结构体的定义如下：

```
typedef struct DXGI_SWAP_CHAIN_DESC {
  DXGI_MODE_DESC BufferDesc;
  DXGI_SAMPLE_DESC SampleDesc;
  DXGI_USAGE BufferUsage;
  UINT BufferCount;
  HWND OutputWindow;
  BOOL Windowed;
  DXGI_SWAP_EFFECT SwapEffect;
  UINT Flags;
} DXGI_SWAP_CHAIN_DESC;
```

**DXGI_MODE_DESC** 类型是另一个结构体，其定义如下：

```
typedef struct DXGI_MODE_DESC
{
  UINT Width;                    // 后台缓冲区宽度
  UINT Height;                   // 后台缓冲区高度
  DXGI_RATIONAL RefreshRate;     // 显示刷新率
  DXGI_FORMAT Format;            // 后台缓冲区像素格式
  DXGI_MODE_SCANLINE_ORDER ScanlineOrdering;// display scanline mode
  DXGI_MODE_SCALING Scaling;     // display scaling mode
} DXGI_MODE_DESC;
```

注意：在下面的数据成员描述中，我们只涵盖了一些常用的标志值和选项，它们对于初学者来说非常重要。对于其他标志值和选项的描述，请参阅 SDK 文档。

9．**BufferDesc**：该结构体描述了我们所要创建的后台缓冲区的属性。我们主要关注的属性有：宽度、高度和像素格式；其他属性的详情请参阅 SDK 文档。

10．**SampleDesc**：多重采样数量和质量级别（参阅 4.1.8 节）。

11．**BufferUsage**：设为 **DXGI_USAGE_RENDER_TARGET_OUTPUT**，因为我们要将场景渲染到后台缓冲区（即，将它用作渲染目标）。

12．**BufferCount**：交换链中的后台缓冲区数量；我们一般只用一个后台缓冲区来实现双缓存。当然，你也可以使用两个后台缓冲区来实现三缓存。

13．**OutputWindow**：我们将要渲染到的窗口的句柄。

14．**Windowed**：当设为 **true** 时，程序以窗口模式运行；当设为 **false** 时，程序以全屏（full-screen）模式运行。

15．**SwapEffect**：设为 **DXGI_SWAP_EFFECT_DISCARD**，让显卡驱动程序选择最高效的显示模式。

16．**Flags**：可选的标志值。如果设为 **DXGI_SWAP_CHAIN_FLAG_ALLOW_MODE_SWITCH**，那么当应用程序切换到全屏模

式时，Direct3D 会自动选择与当前的后台缓冲区设置最匹配的显示模式。如果未指定该标志值，那么当应用程序切换到全屏模式时，Direct3D 会使用当前的桌面显示模式。我们在示例框架中没有使用该标志值，因为对于我们的演示程序来说，在全屏模式下使用当前的桌面显示模式可以得到很好的效果。

下面是在我们的示例框架中填充 **DXGI_SWAP_CHAIN_DESC** 结构体的代码：

```
DXGI_SWAP_CHAIN_DESC sd;
sd.BufferDesc.Width    = mClientWidth;    // 使用窗口客户区宽度
sd.BufferDesc.Height = mClientHeight;
sd.BufferDesc.RefreshRate.Numerator = 60;
sd.BufferDesc.RefreshRate.Denominator = 1;
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
sd.BufferDesc.ScanlineOrdering                                        =
DXGI_MODE_SCANLINE_ORDER_UNSPECIFIED;
sd.BufferDesc.Scaling = DXGI_MODE_SCALING_UNSPECIFIED;
// 是否使用 4X MSAA?
if(mEnable4xMsaa)
{
  sd.SampleDesc.Count = 4;
  // m4xMsaaQuality 是通过 CheckMultisampleQualityLevels()方法获得的
  sd.SampleDesc.Quality = m4xMsaaQuality-1;
}
// NoMSAA
else
{
  sd.SampleDesc.Count = 1;
  sd.SampleDesc.Quality = 0;
}
sd.BufferUsage    = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.BufferCount    = 1;
sd.OutputWindow = mhMainWnd;
sd.Windowed     = true;
sd.SwapEffect    = DXGI_SWAP_EFFECT_DISCARD;
sd.Flags         = 0;
```

**注意**：如果你想在运行时改变多重采样的设置，那么必须销毁然后重新创建交换链。

**注意**：因为大多数显示器不支持超过 24 位以上的颜色，再多的颜色也是浪费，所以我们将后台缓冲区的像素格式设置为 **DXGI_FORMAT_R8G8B8A8_UNORM**（红、绿、蓝、alpha 各 8 位）。额外的 8 位 alpha 并不会输出在显示器上，但在后台缓冲区中可以用于特定的用途。

## 4.2.4 创建交换链

交换链（**IDXGISwapChain**）是通过 **IDXGIFactory** 实例的 **IDXGIFactory::CreateSwapChain** 方法创建的：

```
HRESULT  IDXGIFactory::CreateSwapChain(
    IUnknown *pDevice , // 指向 ID3D11Device 的指针
    DXGI_SWAP_CHAIN_DESC *pDesc, // 指向一个交换链描述的指针
    IDXGISwapChain **ppSwapChain); // 返回创建后的交换链
```

我们可以通过 **CreateDXGIFactory**（需要链接 dxgi.lib）获取指向一个 **IDXGIFactory** 实例的指针。但是使用这种方法获取 **IDXGIFactory** 实例，并调用 **IDXGIFactory::CreateSwapChain** 方法后，会出现如下的错误信息：

```
DXGI  Warning: IDXGIFactory::CreateSwapChain: This  function  is
being  called  with a device  from  a  different IDXGIFactory.
```

要修复这个错误，我们需要使用创建设备的那个 **IDXGIFactory** 实例，要获得这个实例，必须使用下面的 COM 查询（具体解释可参见 **IDXGIFactory** 的文档）：

```
IDXGIDevice *  dxgiDevice  =  0;
HR(md3dDevice ->QueryInterface(__uuidof(IDXGIDevice),
    (void**)&dxgiDevice ));
IDXGIAdapter* dxgiAdapter  =  0;
HR(dxgiDevice ->GetParent(__uuidof(IDXGIAdapter),
    (void**))&dxgiAdapte r ));
// 获得 IDXGIFactory 接口
IDXGIFactory*  dxgiFactory  =  0;
HR(dxgiAdapter->GetParent(__uuid of(IDXGIFactory),
    (void**))&dxgiFactor y));
// 现在，创建交换链
IDXGISwapChain*  mSwapChain;
HR(dxgiFactory->CreateSwapChain(md3dDevice, &sd , &mSw ap Chain);
// 释放 COM 接口
ReleaseCOM (dxgiDevice ;
ReleaseCOM (dxgiAdapter);
ReleaseCOM (dxgiFactory);
```

DXGI（DirectX Graphics Inf rastructure）是独立于 Direct3D 的 API，用于处理与图形关联的东西，例如交换链等。DXGI 与 Direct3D 分离的目的在于其他图形 API（例如 Direct2D）也需要交换链、图形硬件枚举、在窗口和全屏模式之间切换，通过这种设计，多个图形 API 都能使用 DXGI API。

**补充**：你也可以使用 **D3D11CreateDeviceAndSwapChain** 方法同时创建设备、设备上下文和交换链，详情请见 Direct3D 11 教程 1：Direct3D 11 基础。

## 4.2.5 创建渲染目标视图

如 4.1.6 节所述，资源不能被直接绑定到一个管线阶段；我们必须为资源创建资源视图，然后把资源视图绑定到不同的管线阶段。尤其是在把后台缓冲区绑定到管线的输出合并器阶段时（使 Direct3D 可以在后台缓冲区上执行渲染工作），我们必须为后台缓冲区创建一个**渲染目标视图（render target view）**。下面的代码说明了一实现过程：

```
ID3D11RenderTargetView* mRenderTargetView;
ID3D11Texture2D* backBuffer;
```

```
mSwapChain->GetBuffer(0,__uuidof(ID3D11Texture2D),
reinterpret_cast<void**>(&backBuffer));
md3dDevice->CreateRenderTargetView(backBuffer,                    0,
&mRenderTargetView);
ReleaseCOM(backBuffer);
```

4．**IDXGISwapChain::GetBuffer** 方法用于获取一个交换链的后台缓冲区指针。该方法的第一个参数表示所要获取的后台缓冲区的索引值（由于后台缓冲区的数量可以大于 1，所以这里必须指定索引值）。在我们的演示程序中，我们只使用一个后台缓冲区，所以该索引值设为 0。第二个参数是缓冲区的接口类型，它通常是一个 2D 纹理（**ID3D11Texture2D**）。第三个参数返回指向后台缓冲区的指针。

5．我们使用 **ID3D11Device::CreateRenderTargetView** 方法创建渲染目标视图。第一个参数指定了将要作为渲染目标的资源，在上面的例子中，渲染目标是后台缓冲区（即，我们为后台缓冲区创建了一个渲染目标视图）。第二个参数是一个指向 **D3D11_RENDER_TARGET_VIEW_DESC** 结构体的指针，该结构体描述了资源中的元素的数据类型。如果在创建资源时使用的是某种强类型格式（即，非弱类型格式），则该参数可以为空，表示以资源的第一个 mipmap 层次（后台缓冲区也只有一个 mipmap 层次）作为视图格式。第三个参数通过指针返回了创建后的渲染目标视图对象。

6．每调用一次 **IDXGISwapChain::GetBuffer** 方法，后台缓冲区的 COM 引用计数就会向上递增一次，这便是我们在代码片段的结尾处释放它（**ReleaseCOM**）的原因。

## 4.2.6 创建深度/模板缓冲区及其视图

我们现在需要创建深度/模板缓冲区。如 4.1.5 节所述，深度缓冲区只是一个存储深度信息的 2D 纹理（如果使用模板，则模板信息也在该缓冲区中）。要创建纹理，我们必须填充一个 **D3D11_TEXTURE2D_DESC** 结构体来描述所要创建的纹理，然后再调用 **ID3D11Device::CreateTexture2D** 方法。该结构体的定义如下：

```
typedef struct D3D11_TEXTURE2D_DESC {
  UINT Width;
  UINT Height;
  UINT MipLevels;
  UINT ArraySize;
  DXGI_FORMAT Format;
  DXGI_SAMPLE_DESC SampleDesc;
  D3D10_USAGE Usage;
  UINT BindFlags;
  UINT CPUAccessFlags;
  UINT MiscFlags;
} D3D11_TEXTURE2D_DESC;
```

11．**Width**：纹理的宽度，单位为纹理元素（texel）。

12．**Height**：纹理的高度，单位为纹理元素（texel）。

13．**MipLevels**：多级渐近纹理层（mipmap level）的数量。多级渐近纹理将在后面的章节“纹理”中进行讲解。对于深度/模板缓冲区来说，我们的纹理只需要一个多级渐近纹理层。

14．**ArraySize**：在纹理数组中的纹理数量。对于深度/模板缓冲区来说，我们只需要一个纹理。

15．**Format**：一个 **DXGI_FORMAT** 枚举类型成员，它指定了纹理元素的格式。对于深度/模板缓冲区来说，它必须是 4.1.5 节列出的格式之一。

16．**SampleDesc**：多重采样数量和质量级别；请参阅 4.1.7 和 4.1.8 节。

17．**Usage**：表示纹理用途的 **D3D11_USAGE** 枚举类型成员。有 4 个可选值：

- **D3D11_USAGE_DEFAULT**：表示 GPU（graphics processing unit，图形处理器）会对资源执行读写操作。CPU 不能读写这种资源。对于深度/模板缓冲区，我们使用 **D3D11_USAGE_DEFAULT** 标志值，因为 GPU 会执行所有读写深度/模板缓冲区的操作。
- **D3D10_USAGE_IMMUTABLE**：表示在创建资源后，资源中的内容不会改变。这样可以获得一些内部优化，因为 GPU 会以只读方式访问这种资源。除了在创建资源时 CPU 会写入初始化数据外，其他任何时候 CPU 都不会对这种资源执行任何读写操作。
- **D3D10_USAGE_DYNAMIC**：表示应用程序（CPU）会频繁更新资源中的数据内容（例如，每帧更新一次）。GPU 可以从这种资源中读取数据，而 CPU 可以向这种资源中写入数据。
- **D3D10_USAGE_STAGING**：表示应用程序（CPU）会读取该资源的一个副本（即，该资源支持从显存到系统内存的数据复制操作）。

18．**BindFlags**：指定该资源将会绑定到管线的哪个阶段。对于深度/模板缓冲区，该参数应设为 **D3D11_BIND_DEPTH_STENCIL**。其他可用于纹理的绑定标志值还有：

- **D3D11_BIND_RENDER_TARGET**：将纹理作为一个渲染目标绑定到管线上。
- **D3D11_BIND_SHADER_RESOURCE**：将纹理作为一个着色器资源绑定到管线上。

19．**CPUAccessFlags**：指定 CPU 对资源的访问权限。如果 CPU 需要向资源写入数据，则应指定 **D3D11_CPU_ACCESS_WRITE**。具有写访问权限的资源的 Usage 参数应设为 **D3D11_USAGE_DYNAMIC** 或 **D3D11_USAGE_STAGING**。如果 CPU 需要从资源读取数据，则应指定 **D3D11_CPU_ACCESS_READ**。具有读访问权限的资源的 Usage 参数应设为 **D3D11_USAGE_STAGING**。对于深度/模板缓冲区来说，只有 GPU 会执行读写操作；所以，我们将该参数设为 0，因为 CPU 不会在深度/模板缓冲区上执行读写操作。

20．**MiscFlags**：可选的标志值，与深度/模板缓冲区无关，所以设为 0。

**注意**：推荐避免使用 **D3D11_USAGE_DYNAMIC** 和 **D3D11_USAGE_STAGING**，因为有性能损失。要获得最佳性能，我们应创建所有的资源并将它们上传到 GPU 并保留其上，只有 GPU 在读取或写入这些资源。但是，在某些程序中必须有 CPU 的参与，因此这些标志无法避免，但你应该将这些标志的使用减到最小。

在本书中，我们会看到以各种不同选项来创建资源的例子；例如，使用不同的 Usage 标志值、绑定标志值和 CPU 访问权限标志值。但就目前来说，我们只需要关心那些与创建深度/模板缓冲区有关的标志值即可，其他选项可以以后再说。

另外，在使用深度/模板缓冲区之前，我们必须为它创建一个绑定到管线上的深度/模板视图。过程与创建渲染目标视图的过程相似。下面的代码示范了如何创建深度/模板纹理以及与它对应的深度/模板视图：

```
D3D11_TEXTURE2D_DESC depthStencilDesc;
depthStencilDesc.Width              = mClientWidth;
depthStencilDesc.Height            = mClientHeight;
depthStencilDesc.MipLevels          = 1;
depthStencilDesc.ArraySize          = 1;
depthStencilDesc.Format                                    =
```

```
DXGI_FORMAT_D24_UNORM_S8_UINT;
// 使用 4X MSAA?——必须与交换链的 MSAA 的值匹配
if( mEnable4xMsaa)
{
    depthStencilDesc.SampleDesc.Count  = 4;
    depthStencilDesc.SampleDesc.Quality  = m4xMsaaQuality-1;
}
//  不使用 MSAA
else
{
    depthStencilDesc.SampleDesc.Count  =  1;
    depthStencilDesc.SampleDesc.Quality  = 0;
}
depthStencilDesc.Usage              = D3D10_USAGE_DEFAULT;
depthStencilDesc.BindFlags          = D3D10_BIND_DEPTH_STENCIL;
depthStencilDesc.CPUAccessFlags     = 0;
depthStencilDesc.MiscFlags          = 0;
ID3D10Texture2D* mDepthStencilBuffer;
ID3D10DepthStencilView* mDepthStencilView;

HR(md3dDevice->CreateTexture2D(
    &depthStencilDesc, 0, &mDepthStencilBuffer));

HR(md3dDevice->CreateDepthStencilView(
    mDepthStencilBuffer, 0, &mDepthStencilView));
```

**CreateTexture2D** 的第二个参数是一个指向初始化数据的指针，这些初始化数据用来填充纹理。不过，由于个纹理被用作深度/模板缓冲区，所以我们不需要为它填充任何初始化数据。当执行深度缓存和模板操作时，Direct3D 会自动向深度/模板缓冲区写入数据。所以，我们在这里将第二个参数指定为空值。

**CreateDepthStencilView** 的第二个参数是一个指向 **D3D11_DEPTH_STENCIL_VIEW_DESC** 的指针。这个结构体描述了资源中这个元素数据类型（格式）。如果资源是一个有类型的格式（非 typeless），这个参数可以为空值，表示创建一个资源的第一个 mipmap 等级的视图（深度/模板缓冲也只能使用一个 mipmap 等级）。因为我们指定了深度/模板缓冲的格式，所以将这个参数设置为空值。

## 4.2.7 将视图绑定到输出合并器阶段

现在我们已经为后台缓冲区和深度缓冲区创建了视图，就可以将些视图绑定到管线的输出**合并器阶段（output merger stage）**，使些资源成为管线的渲染目标和深度/模板缓冲区：

```
md3dImmediateContext->OMSetRenderTargets(
    1,&mRenderTargetView,mDepthStencilView);
```

第一个参数是我们将要绑定的渲染目标的数量；我们在这里仅绑定了一个渲染目标，不过该参数可以为着色器同时绑定多个渲染目标（是一项高级技术）。第二个参数是我们将要

绑定的渲染目标视图数组中的第一个元素的指针。第三个参数是将要绑定到管线的深度/模板视图。

**注意**：我们可以设置一组渲染目标视图，但是只能设置一个深度/模板视图。使用多个渲染目标是一项高级技术，会在本书的第三部分加以介绍。

## 4.2.8 设置视口

通常我们会把 3D 场景渲染到整个后台缓冲区上。不过，有时我们只希望把 3D 场景渲染到后台缓冲区的一个子矩形区域中，如图 4.7 所示。

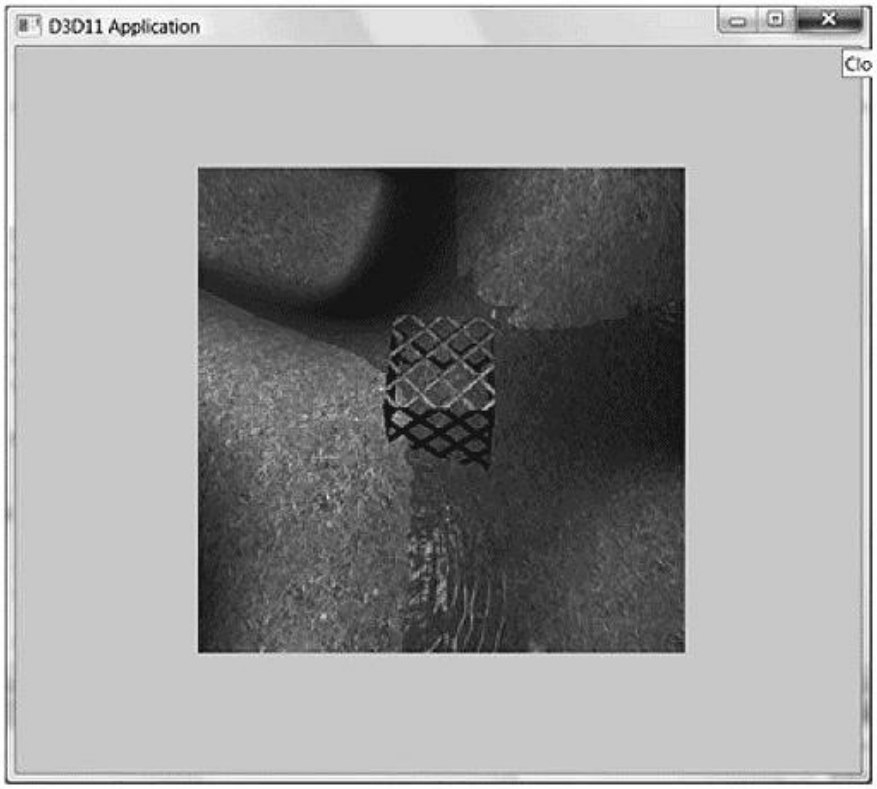



图 4.7：通过修改视口，我们可以把 3D 场景渲染到后台缓冲区的一个子矩形区域中。随后，后台缓冲区中的渲染结果会被呈现到窗口客户区上。

我们将后台缓冲区的子矩形区域称为**视口（viewport）**，它由如下结构体描述：

```
typedef struct D3D11_VIEWPORT {
  FLOAT TopLeftX;
  FLOAT TopLeftY;
  FLOAT Width;
  FLOAT Height;
  FLOAT MinDepth;
  FLOAT MaxDepth;
} D3D11_VIEWPORT;
```

前 4 个数据成员定义了相对于窗口客户区的视口矩形范围。MinDepth 成员表示深度缓冲区的最小值，**MaxDepth** 表示深度缓冲区的最大值。Direct3D 使用的深度缓冲区取值范围

是 0 到 1，除非你想要得到一些特殊效果，否则应将 **MinDepth** 和 **MaxDepth** 分别设为 0 和 1。

在填充了 **D3D11_VIEWPORT** 结构体之后，我们可以使用 **ID3D11Device::RSSetViewports** 方法设置 Direct3D 的视口。下面的例子创建和设置了一个视口，该视口与整个后台缓冲区的大小相同：

```
D3D11_VIEWPORT vp;
vp.TopLeftX = 0;
vp.TopLeftY = 0;
vp.Width     = static_cast<float>(mClientWidth);
vp.Height    = static_cast<float>(mClientHeight);
vp.MinDepth = 0.0f;
vp.MaxDepth = 1.0f;

md3dImmediateContext-->RSSetViewports(1, &vp);
```

第一个参数是绑定的视图的数量（可以使用超过 1 的数量用于高级的效果），第二个参数指向一个 viewports 的数组。

例如，你可以使用视口来实现双人游戏模式中的分屏效果。创建两个视口，各占屏幕的一半，一个居左，另一个居右。然后在左视口中以第一个玩家的视角渲染 3D 场景，在右视口中以第二个玩家的视角渲染 3D 场景。你也可以使用视口只绘制到屏幕的一个子矩形中，而在其他区域保留诸如按钮、列表框之类的 UI 控件。

# 4.3 计时和动画

要正确实现动画效果，我们就必须记录时间，尤其是要精确测量动画帧之间的时间间隔。当帧速率高时，帧之间的时间间隔就会很短；所以，我们需要一个高精确度计时器。

## 4.3.1 性能计时器

我们使用性能计时器（或性能计数器）来实现精确的时间测量。为了使用用于查询性能计时器的 Win32 函数，我们必须在代码中添加包含语句“#include<windows.h>”。

性能计时器采用的时间单位称为计数（count）。我们使用 **QueryPerformanceCounter** 函数来获取以计数测量的当前时间值：

```
__int64 currTime;
QueryPerformanceCounter((LARGE_INTEGER*)&currTime);
```

注意，该函数通过它的参数返回当前时间值，该参数是一个 64 位整数。

我们使用 **QueryPerformanceFrequency** 函数来获取性能计时器的频率（每秒的计数次数）：

```
__int64 countsPerSec;
QueryPerformanceFrequency((LARGE_INTEGER*)&countsPerSec);
```

而每次计数的时间长度等于频率的倒数（这个值很小，它只是百分之几秒或者千分之几秒）：

```
mSecondsPerCount = 1.0 / (double)countsPerSec;
```

这样，要把一个时间读数 **valueInCounts** 转换为秒，我们只需要将它乘以转换因子 **mSecondsPerCount**：

```
valueInSecs = valueInCounts * mSecondsPerCount;
```

由 **QueryPerformanceCounter** 函数返回的值本身不是非常有用。我们使用 **QueryPerformanceCounter** 函数的主要目的是为了获取两次调用之间的时间差——在执行一段代码之前记下当前时间，在该段代码结束之后再获取一次当前时间，然后计算两者之间的差值。也就是，我们总是查看两个时间戳之间的相对差，而不是由性能计数器返回的实际值。下面的代码更好地说明了这一概念：

```
__int64 A = 0;
QueryPerformanceCounter((LARGE_INTEGER*)&A);
/* Do work */
__int64 B = 0;
QueryPerformanceCounter((LARGE_INTEGER*)&B);
```

这样我们就可以知道执行这段代码所要花费的计数时间为（B−A），或者以秒表示的时间为（B−A）*mSecondsPerCount。

注意：MSDN 指出当使用 **QueryPerformanceCounter** 函数时，有以下注意事项：“在多处理器计算机中，任何一个处理器单独调用该函数都不会出现问题。但是，由于基础输入/输出系统（BIOS）或硬件抽象层（HAL）存在技术瓶颈，所以你在不同的处理器上调用该函数会得到不同的结果”。你可以使用 **SetThreadAffinityMask** 函数让主应用程序线程只运行在一个处理器上，不在处理器之间进行切换。

## 4.3.2 游戏计时器类

在下面的两节中，我们将讨论 **GameTimer** 类的实现。

```
class GameTimer
{
public:
    GameTimer();

    float TotalTime()const;  // 单位为秒
    float DeltaTime()const; // 单位为秒

    void Reset(); // 消息循环前调用
    void Start(); // 取消暂停时调用
    void Stop();  // 暂停时调用
    void Tick();  // 每帧调用

private:
    double mSecondsPerCount;
    double mDeltaTime;

    __int64 mBaseTime;
```

```
    __int64 mPausedTime;
    __int64 mStopTime;
    __int64 mPrevTime;
    __int64 mCurrTime;

    bool mStopped;
};
```

需要特别注意的是，构造函数查询了性能计数器的频率。其他成员函数将在随后的两节中讨论。

```
GameTimer::GameTimer()
: mSecondsPerCount(0.0), mDeltaTime(-1.0), mBaseTime(0),
  mPausedTime(0), mPrevTime(0), mCurrTime(0), mStopped(false)
{
    __int64 countsPerSec;
    QueryPerformanceFrequency((LARGE_INTEGER*)&countsPerSec);
    mSecondsPerCount = 1.0 / (double)countsPerSec;
}
```

**注意：GameTimer** 类的定义和实现部分都保存在了 GameTimer.h 和 GameTimer.cpp 文件中，你可以在示例代码的 Common 目录中找到它们。

### 4.3.3 帧之间的时间间隔

当渲染动画帧时，我们必须知道帧之间的时间间隔，以使我们根据逝去的时间长度来更新游戏中的物体。我们可以采用以下步骤来计算帧之间的时间间隔：设 $t_i$ 为第 $i$ 帧时性能计数器返回的时间值，设 $t_{i-1}$ 为前一帧时性能计数器返回的时间值，那么两帧之间的时间差为 $\Delta t = t_i - t_{i-1}$。对于实时渲染来说，我们至少要达到每秒 30 帧的频率才能得到比较平滑的动画效果（我们一般可以达到更高的频率）；所以，$\Delta t = t_i - t_{i-1}$ 通常是一个非常小的值。

下面的代码示范了 $\Delta t$ 的计算过程：

```
void GameTimer::Tick()
{
    if( mStopped )
    {
        mDeltaTime = 0.0;
        return;
    }

    __int64 currTime;
    QueryPerformanceCounter((LARGE_INTEGER*)&currTime);
    mCurrTime = currTime;

    // 当前帧和上一帧之间的时间差
    mDeltaTime = (mCurrTime - mPrevTime)*mSecondsPerCount;
```

```
    // 为计算下一帧做准备
    mPrevTime = mCurrTime;

    // 确保不为负值。DXSDK 中的 CDXUTTimer 提到：如果处理器进入了节电模式
    // 或切换到另一个处理器，mDeltaTime 会变为负值。
    if(mDeltaTime < 0.0)
    {
        mDeltaTime = 0.0;
    }
}

float GameTimer::getDeltaTime() const
{
    return (float)mDeltaTime;
}
```

函数 **Tick** 在应用程序消息循环中的调用如下：

```
int D3DApp::Run()
{
    MSG msg = {0};

    mTimer.Reset();

    while(msg.message != WM_QUIT)
    {
        // 如果接收到 Window 消息，则处理这些消息
        if(PeekMessage( &msg, 0, 0, 0, PM_REMOVE ))
        {
            TranslateMessage( &msg );
            DispatchMessage( &msg );
        }
        // 否则，则运行动画/游戏
        else
        {
            mTimer.Tick();

            if( !mAppPaused )
            {
                CalculateFrameStats();
                UpdateScene(mTimer.DeltaTime());
                DrawScene();
            }
            else
            {
                Sleep(100);
```

```
            }
        }
    }

    return (int)msg.wParam;
}
```

通过这一方式，每帧都会计算出一个$\Delta t$并将它传送给 **UpdateScene** 方法，根据当前帧与前一帧之间的时间间隔来更新场景。下面是 **Reset** 方法的实现代码：

```
void GameTimer::Reset()
{
    __int64 currTime;
    QueryPerformanceCounter((LARGE_INTEGER*)&currTime);

    mBaseTime = currTime;
    mPrevTime = currTime;
    mStopTime = 0;
    mStopped  = false;
}
```

这里包含一些还未讨论过的变量（请参见 4.3.3 节）。不过，我们可以看到，当调用 **Reset** 方法时，**mPrevTime** 被初始化为当前时间。这一点非常重要，因为对于动画的第一帧来说，没有前面的那一帧，也就是说没有前面的时间戳。所以个值必须在消息循环开始之前初始化。

## 4.3.4 游戏时间

另一个需要测量的时间是从应用程序开始运行时起经过的时间总量，其中不包括暂停时间；我们将这一时间称为游戏时间（game time）。下面的情景说明了游戏时间的用途。假设玩家有 300 秒的时间来完成一个关卡。当关卡开始时，我们会获取时间 $t_{start}$，它是从应用程序开始运行时起经过的时间总量。当关卡开始后，我们不断地将 $t_{start}$ 与总时间 $t$ 进行比较。如果 $t - t_{start} > 300$（如图 4.8 所示），就说明玩家在关卡中的用时超过了 300 秒，输掉了这一关。很明显，在一情景中我们不希望计算游戏的暂停时间。

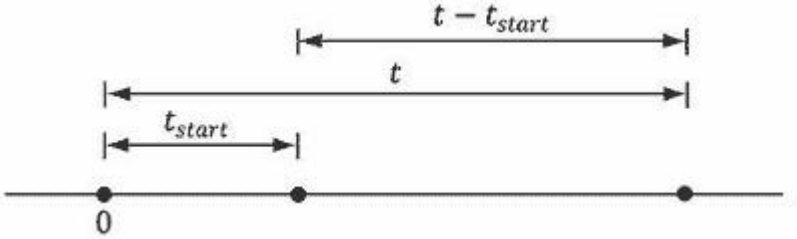



**图 4.8：计算从关卡开始时起的时间。注意，我们将应用程序的开始时间作为原点（0），测量相对于这个时间原点的时间值。**

游戏时间的另一个用途是通过时间函数来驱动动画运行。例如，我们希望一个灯光在时间函数的驱动下环绕着场景中的一个圆形轨道运动。灯光位置可由以下参数方程描述：

$x = 10\cos t$

$y = 20$

$z = 10\sin t$

这里 $t$ 表示时间，随着 $t$（时间）的增加，灯光的位置会发生改变，使灯光在平面 y = 20 上围绕着半径为 10 的圆形轨道运动。对于这种类型的动画，我们也不希望计算游戏的暂停时间；参见图 4.9。

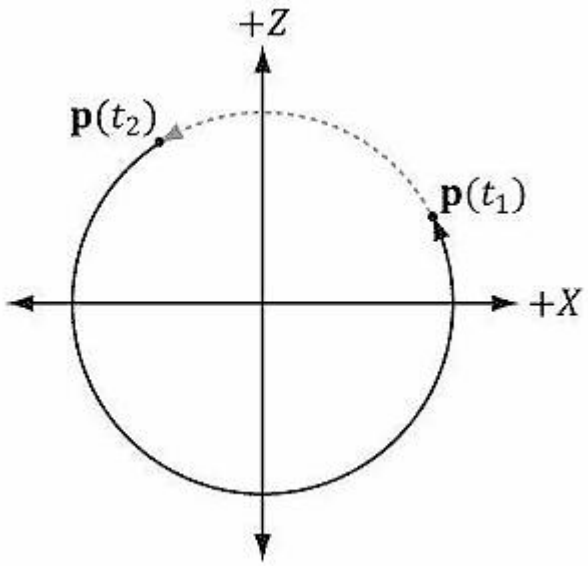



**图 4.9　如果我们在 $t_1$ 时暂停，在 $t_2$ 时取消暂停，并计算暂停时间，那么当我们取消暂停时，灯光的位置会从 $p(t_1)$ 突然跳到 $p(t_2)$。**

我们使用以下变量来实现游戏计时：

```
__int64 mBaseTime;
__int64 mPausedTime;
__int64 mStopTime;
```

如 4.3.3 节所述，当调用 **Reset** 方法时，**mBaseTime** 会被初始化为当前时间。我们可以把它视为从应用程序开始运行时起经过的时间总量。在多数情况下，你只会在消息循环开始之前调用一次 **Reset**，之后不会再调用个方法，因为 **mBaseTime** 在应用程序的整个运行周期中保持不变。变量 **mPausedTime** 用于累计游戏的暂停时间。我们必须累计这一时间，以使我们从总的运行时间中减去暂停时间。当计时器停止时（或者说，当暂停时），**mStopTime** 会帮我们记录暂停时间。

**GameTimer** 类包含两个重要的方法 **Stop** 和 **Start**，它们分别在应用程序暂停和取消暂停时调用，让 GameTimer 记录暂停时间。代码中的注释解释了这两个方法的实现思路。

```
void GameTimer::Stop()
{
    // 如果正处在暂停状态，则略过下面的操作
    if( !mStopped )
    {
        __int64 currTime;
        QueryPerformanceCounter((LARGE_INTEGER*)&currTime);

        // 记录暂停的时间，并设置表示暂停状态的标志
        mStopTime = currTime;
        mStopped  = true;
    }
}
```

```
void GameTimer::Start()
{
    __int64 startTime;
    QueryPerformanceCounter((LARGE_INTEGER*)&startTime);


    // 累加暂停与开始之间流逝的时间
    //
    //                     |<-------d------->|
    // ----*---------------*-----------------*------------> time
    //  mBaseTime       mStopTime        startTime


    // 如果仍处在暂停状态
    if( mStopped )
    {
        // 则累加暂停时间
        mPausedTime += (startTime - mStopTime);
        // 因为我们重新开始计时，因此 mPrevTime 的值就不正确了，
        // 要将它重置为当前时间
        mPrevTime = startTime;
        // 取消暂停状态
        mStopTime = 0;
        mStopped  = false;
    }
}
```

最后，成员函数 **TotalTime** 返回了自调用 **Reset** 之后经过的时间总量，其中不包括暂停时间。它的代码实现如下：

```
// 返回自调用 Reset()方法之后的总时间，不包含暂停时间
float GameTimer::TotalTime()const
{
    // 如果处在暂停状态，则无需包含自暂停开始之后的时间。
    // 此外,如果我们之前已经有过暂停,则 mStopTime - mBaseTime 会包含暂停时间,我们不想包含这个暂停时间,
    // 因此还要减去暂停时间：
    //
    //                     |<--paused time-->|
    //
----*---------------*-----------------*------------*------------*------> time
    //  mBaseTime        mStopTime          startTime      mStopTime  mCurrTime


    if( mStopped )
```

```
    {
        return (float)(((mStopTime - mPausedTime)-mBaseTime)*mSecondsPerCount);
    }

    // mCurrTime - mBaseTime 包含暂停时间,而我们不想包含暂停时间,
    // 因此我们从 mCurrTime 需要减去 mPausedTime:
    //
    //  (mCurrTime - mPausedTime) - mBaseTime
    //
    //                   |<--paused time-->|
    //  ----*---------------*-----------------*------------*------> time
    //  mBaseTime       mStopTime        startTime     mCurrTime

    else
    {
        return (float)(((mCurrTime-mPausedTime)-mBaseTime)*mSecondsPerCount);
    }
}
```

**注意**：我们的演示框架创建了一个 **GameTimer** 实例用于计算应用程序开始后的总时间和两帧之间的时间；你也可以创建额外的实例作为通用的秒表使用。例如，当点着一个炸弹时，你可以启动一个新的 **GameTimer**，当 **TotalTime** 达到 5 秒时，你可以引发一个事件让炸弹爆炸。

# 4.4 演示程序框架

本书中的演示程序均使用 d3dUtil.h、d3dApp.h、d3dApp.cpp 文件中的代码，这些文件可以从本书网站下载。由于本书的第 II 部分和第III部分的所有演示程序都会用到些常用文件，所以我们把些文件保存在了 Common 目录下，使些文件被所有的工程共享，避免多次复制文件。d3dUtil.h 文件包含了一些有用的工具代码，d3dApp.h 和 d3dApp.cpp 文件包含了 Direct3D 应用程序类的核心代码。我们希望读者在阅读本章之后，仔细研究一下些文件，因为我们不会涵盖些文件中的每一行代码（例如，我们不会讲解如何创建一个 Windows 窗口，因为基本的 Win32 编程是阅读本书的先决条件）。该框架的目标是隐藏窗口的创建代码和 Direct3D 的初始化代码；通过隐藏些代码，我们可以在设计演示程序时减少注意力的分散，把注意力集中在示例程序所要表达的特定细节上。

## 4.4.1 D3DApp

**D3DApp** 是所有 Direct3D 应用程序类的基类，它提供了用于创建主应用程序窗口、运行应用程序消息循环、处理窗口消息和初始化 Direct3D 的函数。另外，这个类还定义了一

些框架函数。所有的 Direct3D 应用程序类都继承于 **D3DApp** 类，重载它的 virtual 框架函数，并创建一个 **D3DApp** 派生类的单例对象。**D3DApp** 类的定义如下：

```
#ifndef D3DAPP_H
#define D3DAPP_H

#include "d3dUtil.h"
#include "GameTimer.h"
#include <string>

class D3DApp
{
public:
    D3DApp(HINSTANCE hInstance);
    virtual ~D3DApp();

    HINSTANCE AppInst()const;
    HWND      MainWnd()const;
    float     AspectRatio()const;

    int Run();

    // 框架方法。派生类需要重载这些方法实现所需的功能。

    virtual bool Init();
    virtual void OnResize();
    virtual void UpdateScene(float dt)=0;
    virtual void DrawScene()=0;
    virtual LRESULT MsgProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM
lParam);

    // 处理鼠标输入事件的便捷重载函数
    virtual void OnMouseDown(WPARAM btnState, int x, int y){ }
    virtual void OnMouseUp(WPARAM btnState, int x, int y)  { }
    virtual void OnMouseMove(WPARAM btnState, int x, int y){ }

protected:
    bool InitMainWindow();
    bool InitDirect3D();

    void CalculateFrameStats();

protected:

    HINSTANCE mhAppInst;     // 应用程序实例句柄
```

```
    HWND      mhMainWnd;     // 主窗口句柄
    bool      mAppPaused;    // 程序是否处在暂停状态
    bool      mMinimized;    // 程序是否最小化
    bool      mMaximized;    // 程序是否最大化
    bool      mResizing;     // 程序是否处在改变大小的状态
    UINT      m4xMsaaQuality;// 4X MSAA 质量等级

    // 用于记录"delta-time"和游戏时间(§4.3)
    GameTimer mTimer;

    //  D3D11 设备(§4.2.1)，交换链(§4.2.4)，用于深度/模板缓存的 2D 纹理(§4.2.6)，
    //  渲染目标(§4.2.5)和深度/模板视图(§4.2.6)，和视口(§4.2.8)。
    ID3D11Device* md3dDevice;
    ID3D11DeviceContext* md3dImmediateContext;
    IDXGISwapChain* mSwapChain;
    ID3D11Texture2D* mDepthStencilBuffer;
    ID3D11RenderTargetView* mRenderTargetView;
    ID3D11DepthStencilView* mDepthStencilView;
    D3D11_VIEWPORT mScreenViewport;

    //  下面的变量是在 D3DApp 构造函数中设置的。但是，你可以在派生类中重写这些值。

    //  窗口标题。D3DApp 的默认标题是"D3D11 Application"。
    std::wstring mMainWndCaption;

    //  Hardware device 还是 reference device？D3DApp 默认使用 D3D_DRIVER_TYPE_HARDWARE。
    D3D_DRIVER_TYPE md3dDriverType;
    // 窗口的初始大小。D3DApp 默认为 800x600。注意，当窗口大小在运行阶段改变时，这些值也会随之改变。
    int mClientWidth;
    int mClientHeight;
    //  设置为 true 则使用 4XMSAA(§4.1.8)，默认为 false。
    bool mEnable4xMsaa;
};
#endif // D3DAPP_H
```

在上面的代码中，我们使用注释描述了一些数据成员的含义；这些方法将在随后的几节中讨论。

### 4.4.2 非框架方法

**1．D3DApp**：构造函数，将数据成员简单地初始化为默认值。

**2．~D3DApp**：析构函数，释放 **D3DApp** 获取的 COM 接口。

**3．AppInst**：简单的取值函数，返回应用程序实例句柄的一个副本。

**4．MainWnd**：简单的取值函数，返回主窗口句柄的一个副本。

**5．AspectRatio**：后台缓存区的长宽比，这个比值会在下一章中用到，可以通过下面的代码获得：

```
float D3DApp::AspectRatio() const
{
    return static_cast<float>(mClientWidth)/mClientHeight;
}
```

**6．Run**：该方法封装了应用程序消息循环。它使用 Win32 **PeekMessage** 函数，当没有消息时，它让应用程序处理我们的游戏逻辑。该函数的实现代码请参见 4.3.3 节。

**7．InitMainWindow**：初始化主应用程序窗口；我们假定读者已经具备了基本的 Win32 编程知识，知道如何初始化一个 Windows 窗口。

**8．InitDirect3D**：通过 4.2 节描述的各个步骤初始化 Direct3D。

**9．CalculateFrameStats**：计算每秒的平均帧数和每帧的平均时间（单位为毫秒），这个方法的实现在 4.4.4 节中介绍。

### 4.4.3 框架方法

在本书的每个演示程序中，我们都会重载 **D3DApp** 的 5 个 virtual 函数。这 5 个函数用于实现特定示例中的代码细节。**D3DApp** 类实现的这种结构可以将所有的初始化代码、消息处理代码和其他代码安排得井井有条，使派生类专注于实现演示程序的特定代码。下面是对这些框架方法的描述：

**1．Init**：该方法包含应用程序的初始化代码，比如分配资源、初始化对象和设置灯光。该方法在 **D3DApp** 的实现中包含 **InitMainWindow** 和 **InitDirect3D** 方法的调用语句；所以，当在派生类中重载该方法时，应首先调用该方法的 D3DApp 版本，就像下面这样：

```
void TestApp::Init()
{
    if(!D3DApp::Init())
        return false;
    /* 剩下的初始化代码从这里开始 */
}
```

为你的后续初始化代码提供一个可用的 **ID3D11Device** 设备对象。（通常在获取 Direct3D 资源时都要传递一个有效的 **ID3D11Device** 设备对象。）

**2．OnResize**：该方法在 **D3DApp::MsgProc** 收到 **WM_SIZE** 消息时调用。当窗口的尺寸改变时，一些与客户区大小相关的 Direct3D 属性也需要改变。尤其是需要重新创建后台缓冲区和深度/模板缓冲区，使它们与窗口客户区的大小一致。后台缓冲区的大小可以通过调用 **IDXGISwapChain::ResizeBuffers** 方法来进行调整。而深度/模板缓冲区必须被销毁，然后根据新的大小重新创建。另外，渲染目标视图和深度/模板视图也必须重新创建。

**OnResize** 方法在 **D3DApp** 的实现中包含了调整后台缓冲区和深度/模板缓冲区的代码；详情请直接参见源代码。除缓冲区外，依赖于客户区大小的其他属性（例如，投影矩阵）也必须重新创建。我们把该方法作为框架的一部分是因为当窗口大小改变时，客户代码可能需要执行一些它自己的逻辑。

**3．UpdateScene**：该抽象方法每帧都会调用，用于随着时间更新 3D 应用程序（例如，实现动画和碰撞检测、检查用户输入、计算每秒帧数等等）。

**4．DrawScene**：该抽象方法每帧都会调用，用于将 3D 场景的当前帧绘制到后台缓冲区。当绘制当前帧时，我们调用了 **IDXGISwapChain::Present** 方法将后台缓冲区的内容呈现在屏幕上。

**5．MsgProc**：该方法是主应用程序窗口的消息处理函数。通常，当你只需重载该方法，就可以处理未由 **D3DApp::MsgProc** 处理（或者没按照你所希望的方式处理）的消息。该方法的 **D3DApp** 实现版本会在 4.4.5 节中讲解。如果你重载了这个方法，那么那些你没有处理的消息都会送到 **D3DApp::MsgProc** 中进行处理。

**注意**：除了上述的五个框架方法之外，为了使用起来更方便，我们还提供了三个虚函数，用于处理鼠标点击、释放和移动的事件。

```
virtual void OnMouseDown(WPARAM btnState, int x, int y){ }
virtual void OnMouseUp(WPARAM btnState, int x, int y)  { }
virtual void OnMouseMove(WPARAM btnState, int x, int y){ }
```

你可以重载这些方法处理鼠标事件，而用不着重载 **MsgProc** 方法。这些方法的第一个参数 **WPARAM** 都是相同的，保存了鼠标按键的状态（例如，哪个鼠标按键被按下），第二、三个参数是光标在客户区域的（x，y）坐标。

## 4.4.4 帧的统计数值

通常游戏和绘图应用程序都要测量每秒的渲染帧数（FPS）。要实现这一工作，我们只需计算在某一特定时间段 $t$ 中处理的总帧数（并存储在中变量 $n$ 中）。然后得到时间段 $t$ 中的平均 FPS 为 $fps_{avg}=n/t$。如果我们将 $t$ 设为 1，那么 $fps_{avg}=n/1=n$。在我们的代码中，我们将 $t$ 设为 1，这样可以减少一次除法操作，而且，以 1 秒为限可以得到一个最恰当的平均值——个时间间隔既不长也不短。计算 FPS 的代码由 **D3Dapp::CalculateFrameStats** 方法实现：

```
void D3DApp::CalculateFrameStats()
{
    // 计算每秒平均帧数的代码，还计算了绘制一帧的平均时间。
    // 这些统计信息会显示在窗口标题栏中。
    static int frameCnt = 0;
    static float timeElapsed = 0.0f;

    frameCnt++;

    // 计算一秒时间内的平均值
    if( (mTimer.TotalTime() - timeElapsed) >= 1.0f )
    {
        float fps = (float)frameCnt; // fps = frameCnt / 1
```

```
        float mspf = 1000.0f / fps;

        std::wostringstream outs;
        outs.precision(6);
        outs << mMainWndCaption << L"    "
            << L"FPS: " << fps << L"    "
            << L"Frame Time: " << mspf << L" (ms)";
        SetWindowText(mhMainWnd, outs.str().c_str());

        // 为了计算下一个平均值重置一些值。
        frameCnt = 0;
        timeElapsed += 1.0f;
    }
}
```

为了统计帧数，我们在每帧中都会调用该方法。

除了计算 FPS 外，上面的代码还计算了处理一帧所花费的平均时间，单位为毫秒：

```
float mspf = 1000.0f / fps;
```

**注意**：帧时间与 FPS 是倒数关系，通过乘以 1000ms/ 1s 可以将秒转换为毫秒（1 秒等于 1000 毫秒）。

这条语句的含义是：以毫秒为单位计算渲染一帧所花费的时间；是一个与 FPS 不同的值（虽然个值源于 FPS）。实际上，计算帧时间比计算 FPS 更有用，因为它可以更直观地反映出由于修改场景而产生的渲染时间变化（增加或减少）。另一方面，FPS 无法反映出这一变化。而且，[Dunlop03]在他的文章《FPS versus Frame Time》中指出：由于 FPS 曲线是非线性的，所以使用 FPS 可能会得到误导性的结果。例如，考虑情景一：假设我们的应用程序以 1000FPS 的速率运行，每 1ms（毫秒）渲染一帧。当帧速率下降到 250FPS 时，每 4ms 渲染一帧。现在，再考虑情景二：假设我们的应用程序以的 100FPS 的速率运行，每 10ms 渲染一帧。当帧速率下落到大约 76.9 FPS 时，大约为每 13ms 渲染一帧。在两个情景中，帧时间都是增加了 3 毫秒，增加的渲染时间完全相同。但是 FPS 的读数不够直观。从表面看上，似乎从 1000FPS 下降到 250FPS，要比从 100FPS 下降到 76.9FPS 更严重一些。然而，正如我们之前所说，它们实际表示的渲染时间的增长量是相同的。

## 4.4.5 消息处理函数

我们在消息处理函数中实现的代码与整个应用程序框架相比微不足道。通常，我们不会用到许多 Win32 消息。其实，我们的应用程序的核心代码会在处理器空闲执行（即，当没有窗口消息执行）。不过，有一些重要的消息我们必须处理。因为考虑到篇幅问题，我们不可能在这里列出所有的代码； 我们只能对本例使用的几个消息做以讲解。我们希望读者下载源代码文件，花一些时间熟悉应用程序框架代码，因为它是本书每个示例的基础。

我们处理的第 1 个消息是 **WM_ACTIVATE**。当应用程序获得焦点或失去焦点时，该消息被发送。我们这样来处理它：

```
// 当窗口被激活或非激活时会发送 WM_ACTIVATE 消息。
// 当非激活时我们会暂停游戏，当激活时则重新开启游戏。
case WM_ACTIVATE:
```

```
    if( LOWORD(wParam) == WA_INACTIVE )
    {
        mAppPaused = true;
        mTimer.Stop();
    }
    else
    {
        mAppPaused = false;
        mTimer.Start();
    }
    return 0;
```

可以看到，当应用程序失去焦点时，我们将数据成员 **mAppPaused** 设为 **true**，当应用程序获得焦点时，我们将数据成员 **mAppPaused** 设为 **false**。另外，当应用程序暂停时，计时器停止运行，当应用程序再次激活时，计时器恢复运行。如果回顾 4.3.3 节中 **D3DApp::Run** 方法，我们会发现当应用程序暂停时，我们并没有执行应用程序中的更新 3D 场景的代码，而是将空闲的 CPU 周期返回给了操作系统；通过这一方式，应用程序不会在处于非活动状态时独占 CPU 周期。

我们处理的第二个消息是 **WM_SIZE**。该消息在改变窗口大小时发生。我们处理该消息的主要原因是希望后台缓冲区和深度/模板缓冲区的大小与窗口客户区的大小相同（为了不出现图像拉伸）。所以，每次改变窗口大小时，我们希望同改变缓冲区的大小。这一任务由 **D3DApp::OnResize** 方法实现。如前所述，后台缓冲区的大小可以通过调用 **IDXGISwapChain::ResizeBuffers** 方法来进行调整。而深度/模板缓冲区必须被销毁，然后根据新的大小重新创建。另外，渲染目标视图和深度/模板视图也必须重新创建。当用户拖动窗口边框时，我们必须格外小心，因为此时会有接连不断的 WM_SIZE 消息发出，我们不希望连续地调整缓冲区大小。所以，当用户拖动窗口边框时，我们（除了暂停应用程序外）不应该执行任何代码，等到用户的拖动操作结束之后我们再调整缓冲区的大小。我们通过处理 **WM_EXITSIZEMOVE** 消息来完成一工作。该消息在用户释放窗口边框时发送。

```
// 当用户拖动窗口边框时会发送 WM_EXITSIZEMOVE 消息。
case WM_ENTERSIZEMOVE:
    mAppPaused = true;
    mResizing  = true;
    mTimer.Stop();
    return 0;

// 当用户是否窗口边框时会发送 WM_EXITSIZEMOVE 消息。
// 然后我们会基于新的窗口大小重置所有图形变量
case WM_EXITSIZEMOVE:
    mAppPaused = false;
    mResizing  = false;
    mTimer.Start();
    OnResize();
    return 0;
```

最后处理的 3 个消息的实现过程非常简单，所以我们直接来看代码：

```
// 窗口被销毁时发送 WM_DESTROY 消息
```

```
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;

// 如果使用者按下 Alt 和一个与菜单项不匹配的字符时，或者在显示弹出式菜单而
// 使用者按下一个与弹出式菜单里的项目不匹配的字符键时。
case WM_MENUCHAR:
    // 按下 alt-enter 切换全屏时不发出声响
    return MAKELRESULT(0, MNC_CLOSE);

// 防止窗口变得过小。
case WM_GETMINMAXINFO:
    ((MINMAXINFO*)lParam)->ptMinTrackSize.x = 200;
    ((MINMAXINFO*)lParam)->ptMinTrackSize.y = 200;
    return 0;
```

## 4.4.6 全屏模式

我们创建的 **IDXGISwapChain** 接口可以自动捕获 Alt+Enter 组合键消息，将应用程序切换到全屏模式（full-screen mode）。在全屏模式下，再次按下 Alt+Enter 组合键，可以返回到窗口模式。在这两种模式的切换中，应用程序的窗口大小会发生变化，会有一个 **WM_SIZE** 消息发送到应用程序的消息队列中；应用程序可以在此时调整后台缓冲区和深度/模板缓冲区的大小，使缓冲区与新的窗口大小匹配。另外，当切换到全屏模式时，窗口样式也会发生改变（即，窗口边框和标题栏会消失）。读者可以使用 Visual Studio 的 Spy++工具查看一下在按下 Alt+Enter 组合键时由演示程序产生的 Windows 消息。

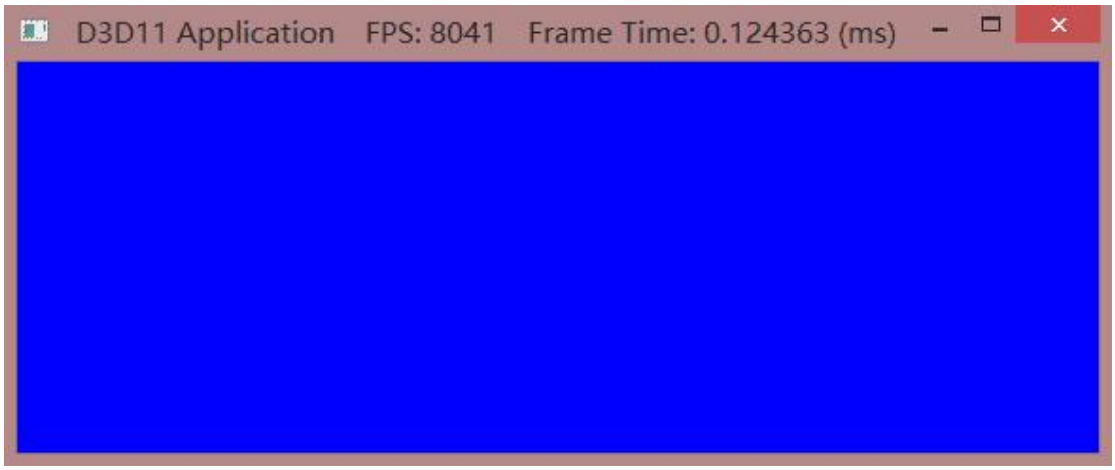



**图 4.10 第 4 章示例程序的屏幕截图。**

注意：读者可以回顾一下 4.2.3 节描述的 **DXGI_SWAP_CHAIN_DESC::Flags** 标志值。

## 4.4.7 初始化 Direct3D 演示程序

现在，我们已经讨论了应用程序框架的所有内容，下面让我们来使用该框架生成一个小

程序。基本上，我们用不着做任何实际工作就可以实现这个程序，因为基类 **D3DApp** 已经实现了它所需要的大部分功能。读者在这里应该关注是如何编写 D3DApp 的派生类以及实现框架方法，我们将要在这些框架方法中编写特定的示例代码。本书中的所有程序都使用这一模板。

```
#include "d3dApp.h"

class InitDirect3DApp : public D3DApp
{
public:
    InitDirect3DApp(HINSTANCE hInstance);
    ~InitDirect3DApp();

    bool Init();
    void OnResize();
    void UpdateScene(float dt);
    void DrawScene();
};

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE prevInstance,
                   PSTR cmdLine, int showCmd)
{
    // Enable run-time memory check for debug builds.
#if defined(DEBUG) | defined(_DEBUG)
    _CrtSetDbgFlag( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
#endif

    InitDirect3DApp theApp(hInstance);

    if( !theApp.Init() )
        return 0;

    return theApp.Run();
}

InitDirect3DApp::InitDirect3DApp(HINSTANCE hInstance)
: D3DApp(hInstance)
{
}

InitDirect3DApp::~InitDirect3DApp()
{
}

bool InitDirect3DApp::Init()
```

```
{
    if(!D3DApp::Init())
        return false;

    return true;
}

void InitDirect3DApp::OnResize()
{
    D3DApp::OnResize();
}

void InitDirect3DApp::UpdateScene(float dt)
{

}

void InitDirect3DApp::DrawScene()
{
    assert(md3dImmediateContext);
    assert(mSwapChain);

    md3dImmediateContext->ClearRenderTargetView(mRenderTargetView,
reinterpret_cast<const float*>(&Colors::Blue));
    md3dImmediateContext->ClearDepthStencilView(mDepthStencilView,
D3D11_CLEAR_DEPTH|D3D11_CLEAR_STENCIL, 1.0f, 0);

    HR(mSwapChain->Present(0, 0));
}
```

# 4.5 调试 Direct3D 应用程序

为了简化代码并突出重点，我们在本书的示例中省略了很多错误处理语句。不过，我们实现了一个宏，用它来检查许多 Direct3D 函数返回的 **HRESULT** 值。这个宏定义在 d3dUtil.h 文件中：

```
#if defined(DEBUG) | defined(_DEBUG)
    #ifndef HR
    #define HR(x)                                              \
    {                                                          \
        HRESULT hr = (x);                                      \
        if(FAILED(hr))                                         \
        {                                                      \
            DXTrace(__FILE__, (DWORD)__LINE__, hr, L#x, true); \
```

```
        }                                                   \
    }
    #endif

#else
    #ifndef HR
    #define HR(x) (x)
    #endif
#endif
```

当函数的返回值表明调用失败时，我们把返回值传递给 **DXTrace** 函数。请注意，当使用该函数时，我们必须在代码中添加包含语句“#include<dxerr.h>”，并链接 dxerr.lib 库文件，只有这样程序才能通过编译。

```
HRESULT WINAPI DXTraceW(const char* strFile, DWORD dwLine,
    HRESULT hr, const WCHAR* strMsg, BOOL bPopMsgBox);
```

该函数可以弹出一个消息框，显示出现错误的文件、行号、有关错误的描述信息以及导致错误的函数名。图 4.11 是它的一个例子。注意，当 **DXTrace** 函数的最后一个参数设为 **false** 时，该函数不会显示消息框，而是把调试信息输出到 Visual C++的输出窗口。当我们不使用调试模式时，**HR** 宏不执行任何代码。另外，**HR** 必须是一个宏而不能是一个函数；否则 **__FILE__**和**__LINE__**将无法引用调用 **HR** 宏的函数所在的文件和行。

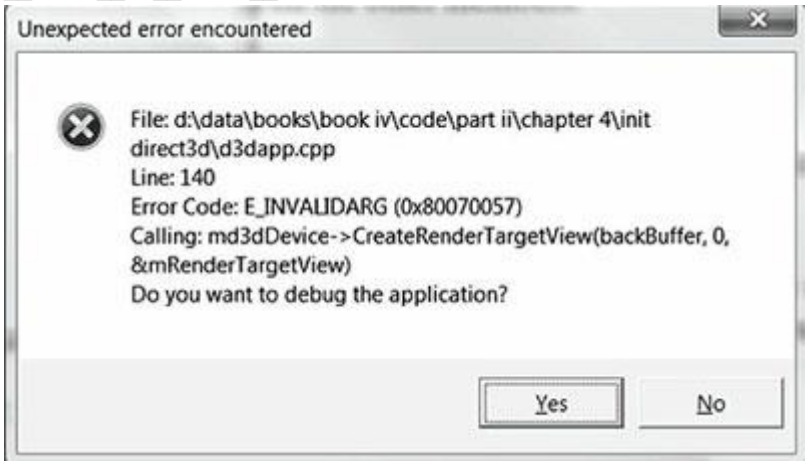



**图 4.11：当 Direct3D 函数返回一个错误时，通过 DXTrace 函数显示消息框。**

现在我们使用 **HR** 宏来包围返回 **HRESULT** 值的 Direct3D 函数，下面是一个例子：

```
HR(D3DX11CreateShaderResourceViewFromFile(md3dDevice,
    L"grass.dds", 0, 0, &mGrassTexRV, 0 ));
```

当我们调试演示程序时，这个宏可以很好地运作，但是对于一个实际的应用程序来说，我们应该使用更完善的错误处理机制。

注意：L#x 将 **HR** 宏的参数转换成一个 Unicode 字符串。通过这一方式，我们可以把导致错误的函数调用语句输出到消息框上面。

# 4.6 小结

**1.** Direct3D 可以被视为程序员和图形硬件之间的一个中介。例如，程序员调用 Direct3D 函数将资源视图绑定到硬件渲染管线、设定渲染管线的输出并绘制 3D 几何体。

**2.** 在 Direct3D 11 中，一个支持 Direct3D 11 的图形设备必须支持 Direct3D 11 规定的整个功能集合以及少量的额外功能。

**3.** 组件对象模型（COM）技术使 DirectX 独立于任何编程语言，并具有版本向后兼容的特性。Direct3D 程序员不必知道 COM 的实现细节及工作方式；只需要知道如何获取和释放 COM 接口即可。

**4.** 1D 纹理如同一维数据元素数组，2D 纹理如同二维数据元素数组，3D 纹理如同三维数据元素数组。纹理元素的格式由 **DXGI_FORMAT** 枚举类型成员描述。纹理通常用于存储图像数据，但是也可以用于存储其他数据，比如深度信息（例如，深度缓冲区）。GPU 可以在纹理上执行特殊运算，比如过滤器和多重采样。

**5.** 在 Direct3D 中，资源不能被直接绑定到一个管线阶段；我们只能把与资源关联的资源视图绑定到不同的管线阶段。我们可以为一个资源创建多个不同的视图。通过这一方式，一个资源可以被绑定到多个不同的渲染管线阶段。如果在创建资源时使用的是弱类型格式，那么在为该资源创建视图时必须指定明确的类型。

**6. ID3D11Device** 和 **ID3D11DeviceContext** 接口可以被视为物理图形设备硬件的软控制器；也就是，我们可以通过这些接口与硬件进行交互。**ID3D11Device** 接口负责检查硬件支持的功能、分配资源。**ID3D11DeviceContext** 接口负责设置渲染状态，将资源绑定到图形管线，发送渲染指令。

**7.** 为了避免动画出现闪烁，最好是将整个帧绘制到一个叫做后台缓冲区的离屏纹理中。当整个屏幕绘制到后台缓冲区之后，它就会以一个完整帧的形式呈现在屏幕上，通过这种方式，观察者就不会觉察到图像的绘制过程了。当帧绘制到后台缓冲区之后，后台缓冲和前台缓冲就会发生互换：后台变前台，前台变后台。交换两者的过程叫做呈现（presenting）。前台缓冲和后台缓冲构成一个交换链，由 IDXGISwapChain 接口表示，使用两个缓冲被称为双缓冲。

8. 对于屏幕上不透明的物体来说，离相机近的点会遮挡后面的点。深度缓冲就是一种判断哪个点离相机近的技术。通过这一方式，我们就无需关心对象的绘制顺序。

9. 性能计数器是一种高精度计时器，它为测量微小的时间差提供了准确无误的计时测量方法，比如帧之间的时间间隔。性能计时器采用的时间单位叫做计数。**QueryPerformanceFrequency** 函数用于输出性能计时器每秒的计数值，这个值的单位可以从计数转换为秒。我们可以通过 **QueryPerformanceCounter** 函数获取性能计时器的当前时间。

10. 我们通过累计某一时间段$\Delta t$ 内的帧数来计算 FPS（frames per second，每秒帧数）。设 $n$ 为时间段$\Delta t$ 中的帧数；在一段时间中的平均每秒帧数为 $fps_{avg}=n/\Delta t$。帧速率会对性能评定产生误导；帧时间是更有效的信息。以秒为单位的帧时间等于帧速率的倒数，即 $1/fps_{avg}$。

11. 示例框架用于为本书的所有演示程序提供统一的编程接口。这些代码保存在 d3dUtil.h、d3dApp.h 和 d3dApp.cpp 文件中，它们封装了每个应用程序必须实现的标准初始化代码。通过封装些代码，可以使示例程序更专注于所要演示的技术。

12. 当以调试模式生成程序时，我们可以使用 D3D11_CREATE_DEVICE_DEBUG 标志值创建 Direct3D 设备来启用调试层。在指定了调试标志值后，Direct3D 会把调试信息发送到 VC++的输出窗口。另外，当以调试模式生成程序时，我们应使用 D3DX 库的调试版本（即，

d3dx11d.lib）。

# 5.1 三维视觉

本章的主要讲解渲染管线（rendering pipeline）。渲染管线是指：在给定一个 3D 场景的几何描述及一架已确定位置和方向的虚拟摄像机（virtual camera）时，根据虚拟摄像机的视角生成 2D 图像的一系列步骤（如图 5.1 所示）。本章的内容大部分是理论性的——下一章才会把理论用于实践。在我们开始学习渲染管线之前，读者应该先了解两个基本概念：一是构成 3D 视觉的基本要素（也就是，研究如何通过 2D 屏幕来呈现 3D 场景）；二是讲解如何在 Direct3D 中以数学方式表示和使用颜色。

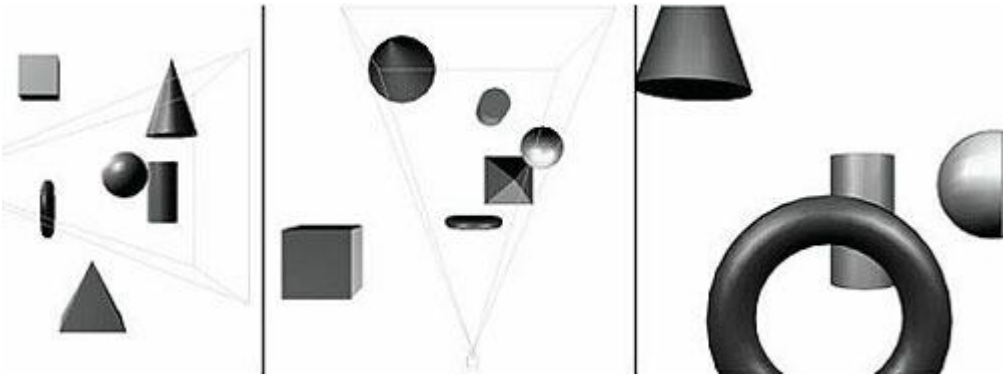


**图 5.1：左图是放置在 3D 场景中的一些物体以及一架已确定位置和方向的摄像机；中图表示的是同一个场景，但是是从上方向下看的视角。“棱锥体”表示观察者所能看到的视域空间；视域空间之外的物体（或者物体的一部分）不会被看到。右图是根据摄像机的“视角”生成的 2D 图像。**

## 学习目标

5. 了解如何在 2D 图像中表现物体的体积感和纵深感。
6. 了解如何在 Direct3D 中描述 3D 物体。
7. 学习如何模拟虚拟摄像机。
8. 理解渲染管线——以几何方式描述 3D 场景并生成 2D 图像的过程。

在踏上 3D 计算机绘图的旅程之前，我们必须先弄明白一个问题：如何在 2D 屏幕上表现 3D 场景的纵深感和体积感？幸运的是，这个问题已经得到了很好的解决，因为几个世纪以来艺术家们一直是在 2D 画布上绘制 3D 场景。本节我们将讲述几种表现图像立体感的关键技术，虽然它实际上是画在平面上的。

假设有一条笔直的铁路，它一直向远处延伸。两条铁轨之间彼此平行，互不相交。当你站在铁路上向远处望时，你会发现随着距离的增加，两条铁轨之间的间隔会越来越近，直至在一个无限远的地方相交为一点。这是通过观察总结出来的我们人类视觉系统的一个特性：平行线会汇集为一个零点（vanishing point），如图 5.2 所示。

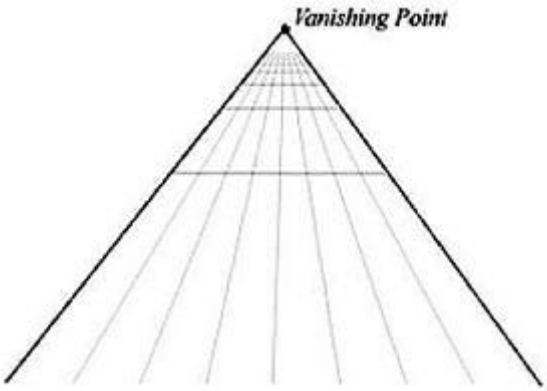



**图 5.2：平行线汇集为一个零点。艺术家们有时将它称为线性透视（linear perspective）。**

从观察中总结出来的人类视觉的另一个特性是物体的大小会随着深度的增加而减小；也就是，近处的物体比远处的物体大。例如，在远处山坡上的一栋房子看上去很小，而近处的一棵树看上去很大。图 5.3 展示了一个简单的场景，其中放置了几排柱子。这些柱子的大小实际上是一样的，但是从观察者的角度上看，随着深度的增加，柱子会变得越来越小。而且我们还可以看到，这些柱子在地平线上会相交为一个零点。

**图 5.3：这里，所有的柱子大小相同，但是由于景深现象（depth phenomenon），观察者会发现柱子越来越小。**

物体重叠（object overlap）是我们能感受到的另一种现象。物体重叠是指一个不透明的物体会挡住它后面的其他物体的一部分（或全部）。这一点非常重要，因为它告诉我们物体在场景中的远近关系。我们已经（在第 4 章中）讨论了如何使用 Direct3D 的深度缓冲区来判定哪些像素会被遮挡，不应该被绘制。出于完整性的考虑，我们在图 5.4 中再次展示了这一情形。

**图 5.4：彼此遮挡的一组物体。**

考虑图 5.5。左边是一个没有光线照射的球体，右边是一个有光线照射的球体。可以看到，左边的球体看上去相当单调——完全不像球体，只能算是一个 2D 圆！因此，在表现 3D 物体的立体感和体积感时光照（lighting）和阴影（shading）具有非常重要的作用。

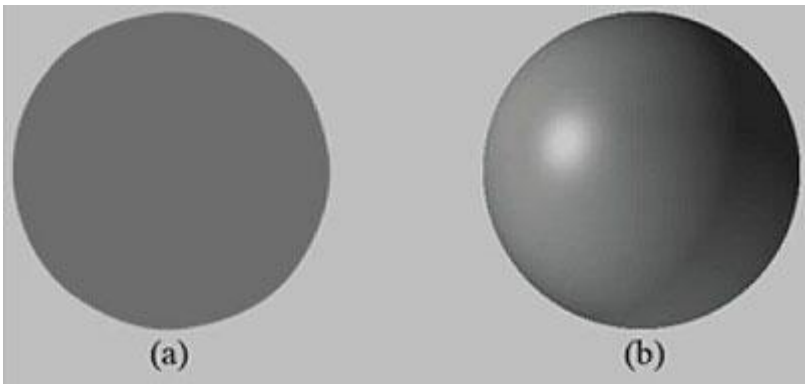



**图 5.5：（a）没有光线照射的球体看上去就像一个 2D 圆。（b）而有光线照射的球体看上去很立体。**

最后，图 5.6 展示了一艘飞船和它的阴影。阴影具有两个关键作用：一是告诉我们场景中的光源位置，二是告诉我们飞船距离地面的高度。

**图 5.6：一艘飞船和它的阴影。阴影间接地说明了光源在场景中的位置以及飞船距离地面的高度。**

刚才讨论的现象都很简单，都是我们在日常生活中能够观察到的现象。但是，更进一步地了解这些现象有助于我们学习和使用 3D 计算机绘图。

# 5.2 模型的表现形式

3D 物体可以通过三角形网格近似地模拟表示，三角形是构成物体模型的基本单位。图 5.7 说明，我们可以通过三角形网格来模拟真实世界中的任何 3D 物体。一般来说，网格的三角形密度越大，模拟出来的效果就越好。当然，我们使用的三角形越多，所要求的硬件性能也就越高，所以必须根据应用程序目标用户的硬件性能来决定模型的精度。除三角形外，有时还需要绘制点和直线。例如，通过绘制一系列 1 像素宽的短线段可以模拟出一条平滑曲线。

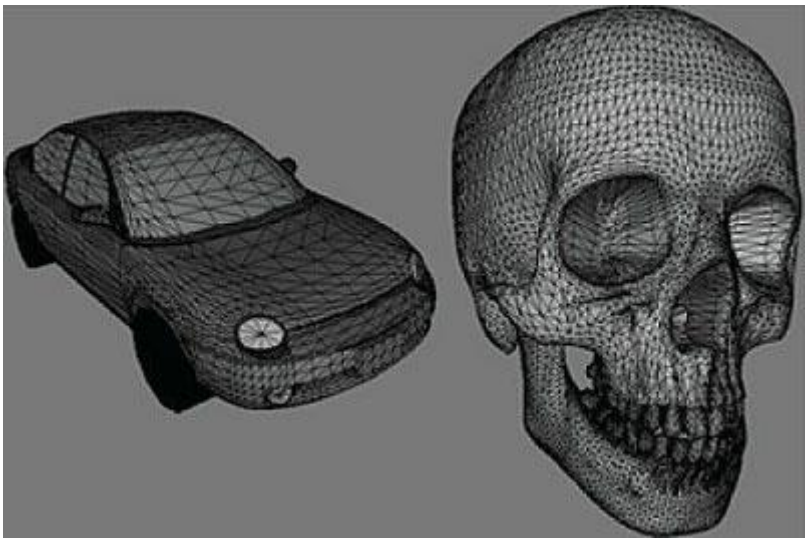


**图 5.7：（左）由低密度三角形网格模拟的轿车。（右）由高密度三角形网格模拟的头骨。**

图 5.7 中的大规则三角形网格说明了一件事情：要以手工方式编写一个 3D 模型的三角形列表是一件极其困难的事情。除了最简单的模型外，所有的模型都是用专门的 3D 建模软件生成的。这些建模软件提供了可视化的交互环境以及非常丰富的建模工具，用户可以使用些软件来创作复杂而逼真的网格模型，整个建模过程非常简单，很容易就能学会。现在在游戏开发领域中较为流行的建模软件有：3ds Max（http://usa.autodesk.com/3ds-max/）、LightWave 3D（http://www.newtek.com/lightwave/）、Maya（http://www.autodesk.com/maya）和 Softimage XSI（www.softimage.com）和 Blender（http://www.blender.org/）。不过，在本书的第一部分中，我们仍会通过手工方式或数学公式生成一些非常简单的 3D 模型（例如，使用参量公式可以很容易地生成圆柱体和球体的三角形列表）。在本书的第三部分中，我们会讲解如何载入和显示由 3D 建模软件导出的 3D 模型。

# 5.3 基本计算机颜色

计算机显示器通过每个像素发射红、绿、蓝光的混合光线。当混合光线进入人的眼睛时

会触碰到视网膜的某些区域，对锥感细胞产生刺激，神经触突会通过视觉神经传送到大脑。大脑会解释些信号并生成颜色。随着混合光线的变化，细胞会受到不同的刺激，从而在我们的思想意识中产生不同的颜色。图 5.8 说明了红、绿、蓝三色的混合方式以及不同强度的红色。通过为每个颜色分量指定不同的强度并对其进行混合，可以描述我们所要显示的真实图像中的所有颜色。

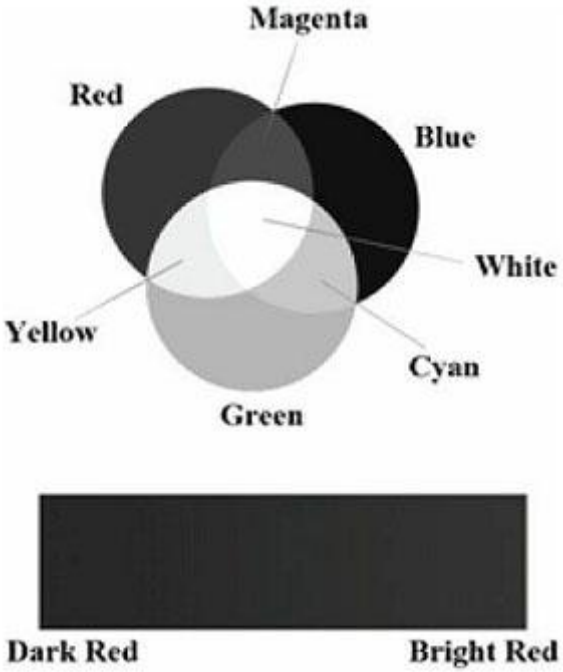



**图 5.8 （上）对纯红、纯绿、纯蓝三种颜色进行混合，得到新的颜色。（下）通过控制红光的强度得到不同明暗程度的红色。**

读者可以使用绘图软件（比如 Adobe Photoshop）或 Win32 颜色对话框（图 5.9）进一步了解如何使用 RGB（red、green、blue）值来描述颜色。尝试使用不同的 RGB 组合，看一看它们产生的颜色。

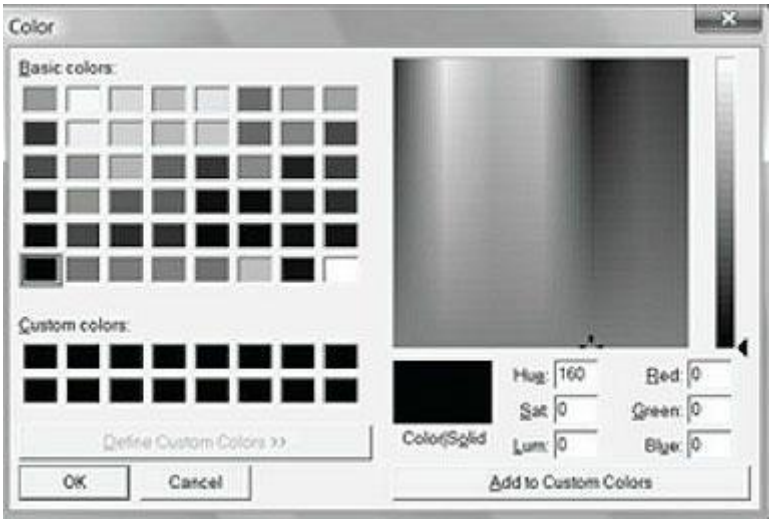



**图 5.9 颜色对话框。**

显示器所能发射的红、绿、蓝光的强度有最大限制。我们使用从 0 到 1 的规范化区间来描述光线强度。0 表示没有强度，1 表示最高强度。中间值表示中等强度。例如，值（0.25,0.67,1.0）表示混合光由强度为 25%的红光、强度为 67%的绿和强度为 100%的蓝光组成。如本例所示，

我们可以通过 3D 向量（r,g,b）来表示颜色，其中 0≤r，g，b≤1，三个颜色分量分别描述红、绿、蓝光的强度。

## 5.3.1 颜色运算

某些向量运算也适用于颜色向量。例如，我们可以把颜色向量加在一起得到一个新的颜色：

(0.0, 0.5, 0.0) + (0.0, 0.0, 0.25) = (0.0, 0.5, 0.25)

通过混合一个中等强度的绿色和一个低强度的蓝色，得到一个深绿色。

也可以通过颜色相减来得到一个新的颜色：

(1, 1,1) − (1, 1,0) = (0, 0,1)

也就是，我们从白色中减去它的红色和绿色部分，得到最终的蓝色。

标量乘法也有意义。例如：

0.5(1, 1,1) = (0.5, 0.5, 0.5)

也就是，将白色乘以 0.5，得到一个中等强度的灰色。而运算 2(0.25, 0.0, 0.0) = (0.5, 0.0, 0.0)，可使红色分量的强度增大一倍。

颜色向量的点积和叉积没有意义。不过，颜色向量有一种特殊的乘法运算，叫做分量乘法（componentwise multiplication）。其定义如下：

$(c_r,c_g,c_b)\otimes(k_r,k_g,k_b) = (c_rk_r,c_gk_g,c_bk_b)$

这一运算主要用于光照方程。例如，一个颜色为(r,g,b)的入射光，照射在一个平面上。该平面反射 50%的红光、75%的绿和 25%的蓝光，其余线均被平面吸收。则折回的反射光颜色为：

$(r,g,b)\otimes(0.5,0.75,0.25) = (0.5r,0.75g,0.25b)$

我们可以看到，由于平面吸收了一些线，所以当光线照射在平面上时会丢失一些颜色。

当进行颜色运算时，某些颜色分量可能会超出[0,1]区间；例如，方程(1,0.1,0.6) + (0.0,0.3,0.5) = (1,0.4,1.1)。由于 1.0 表示颜色分量的最大强度，任何分量都不能大于该值。所以，我们要把 1.1 截取为 1.0。同样，显示器不能发射负光，所以任何负的颜色分量（负值是由减法运算取得的结果）都必须截取为 0.0。

## 5.3.2 128 位颜色

通常，在颜色中会包含一个附加的颜色分量，叫做 alpha 分量。alpha 分量用于表示颜色的不透明度，我们会在第 9 章“混合”中使用 alpha 分量。（由于我们目前还用不到混合，所以现在暂且将 alpha 分量设置为 1。）

包含 alpha 分量意味着我们要使用 4D 向量(r,g,b,a)来表示颜色，其中 0≤r,g,b,a≤1。要表示一个 128 位颜色，可以为每个分量指定一个浮点值。因为从数学上来说，颜色就是一个 4D 向量，所以我们可以在代码中使用 **XMVECTOR** 类型表示一个颜色，而且还可以利用 XNA 数学矢量函数所用的 SIMD 操作带来的优势进行颜色运算（例如颜色相加、相减、标量乘法）。对于分量乘法，XNA 数学库提供了以下方法：

```
XMVECTOR XMColorModulate(// Returns (cr, cg, cb, ca) ⊗ (kr,kg,kb,ka)
    FXMVECTOR C1, // (cr, cg, cb, ca)
    FXMVECTOR C2 // (kr, kg, kb, ka) );
```

### 5.3.3 32 位颜色

当使用 32 位表示一个颜色时，每个字节会对应于一个颜色分量。由于每个颜色分量占用一个 8 位字节，所以每个颜色分量可以表示 256 种不同的明暗强度——0 表示没有强度，255 表示最高强度，中间值表示中等强度。从表面上看，为每个颜色分量分配一个字节似乎很小，但是通过计算组合值（256×256×256 = 16,777,216）可以发现，这种方式可以表示上千万种不同的颜色。XNA 数学库提供了以下结构用于存储 32 位颜色：

```
// ARGB Color; 8-8-8-8 bit unsigned normalized integer components packed into
// a 32 bit integer.  The normalized color is packed into 32 bits using 8 bit
// unsigned, normalized integers for the alpha, red, green, and blue components.
// The alpha component is stored in the most significant bits and the blue
// component in the least significant bits (A8R8G8B8):
// [32] aaaaaaaa rrrrrrrr gggggggg bbbbbbbb [0]
typedef struct _XMCOLOR
{
    union
    {
        struct
        {
            UINT b    : 8;  // Blue:    0/255 to 255/255
            UINT g    : 8;  // Green:   0/255 to 255/255
            UINT r    : 8;  // Red:     0/255 to 255/255
            UINT a    : 8;  // Alpha:   0/255 to 255/255
        };
        UINT c;
    };

#ifdef __cplusplus

    _XMCOLOR() {};
    _XMCOLOR(UINT Color) : c(Color) {};
    _XMCOLOR(FLOAT _r, FLOAT _g, FLOAT _b, FLOAT _a);
    _XMCOLOR(CONST FLOAT *pArray);

    operator UINT () { return c; }

    _XMCOLOR& operator= (CONST _XMCOLOR& Color);
    _XMCOLOR& operator= (CONST UINT Color);
```

```
#endif // __cplusplus

} XMCOLOR;
```

通过将整数区间[0,255]映射到实数区间[0,1]，可以将一个 32 位颜色转换为一个 128 位颜色。这一映射工作是通过将每个分量除以 255 来实现。也就是，当 n 为 0 到 255 之间的一个整数时，对应于规范化区间[0,1]的分量值为 $0 \leqslant \frac{n}{255} \leqslant 1$。例如，32 位颜色(80,140,200,255)变为：

$$(80,140,200,255) \rightarrow \left(\frac{80}{255},\frac{140}{255},\frac{200}{255},\frac{255}{255}\right) \approx (0.31,0.55,0.78,1.0)$$

另一方面，通过将每个颜色分量乘以 255 并进行四舍五入，可以将一个 128 位颜色转换为一个 32 位颜色。例如：

(0.3,0.6,0.9,1.0) → (0.3*255,0.6*255,0.9*255,1.0*255) = (77,153,230,255)

当把一个 32 位颜色转换为一个 128 位颜色或者进行反向转换时，通常要执行额外的位运算，因为 8 位颜色分量通常会被封装在一个 32 位整数中（例如，无符号整数），即在 **XMCOLOR** 中。XNA 数学库使用以下函数处理一个 **XMCOLOR** 并以 **XMVECTOR** 的形式返回：

```
XMVECTOR XMLoadColor(CONST XMCOLOR* pSource);
```

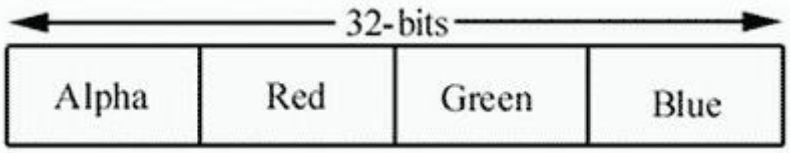



**图 5.10：一个 32 位颜色，它为每个颜色分量分配一个字节。**

图 5.10 说明了如何将 4 个 8 位颜色分量封装为一个无符号整数。注意，这只是用于封装颜色分量的方式之一。除使用 ARGB 外，还可以使用 ABGR 或 RGBA。不过，**XMCOLOR** 类使用 ARGB 格式。XNA 数学库提供了一个函数可以将一 **XMVECTOR** 颜色转化为一个 **XMCOLOR**：

```
VOID XMStoreColor(XMCOLOR*  pDestination,FXMVECTOR V);
```

通常，许多颜色运算（例如，在像素着色器中）使用的都是 128 位颜色值；通过这一方式，我们可以有足够多的二进制位来保证计算的精确度，减少算术错误的累积。不过，最终的像素颜色通常是存储在后台缓冲区的 32 位颜色值中；目前的物理显示设备还不能充分利用更高的分辨率颜色。

# 5.4 渲染管线概述

渲染管线（rendering pipeline）是指：在给定一个 3D 场景的几何描述及一架已确定位置和方向的虚拟摄像机时，根据虚拟摄像机的视角生成 2D 图像的一系列步骤（译者注：渲染管线由许多步骤组成，每个步骤称为一个阶段）。图 5.11 所示为构成渲染管线的各个阶段，以及与各个阶段相关的内存资源。从内存指向阶段的箭头表示该阶段可以从内存读取数据；例如，像素着色器阶段（pixel shader stage）可以从内存中的纹理资源中读取数据。从阶段指向内存的箭头表示该阶段可以向内存写入数据；例如，输出合并器阶段（output merger stage）可以将数据写入后台缓冲区和深度/模板缓冲区。我们还可以看到输出合并器阶段的箭头是

双向的（可以读取和写入 GPU 资源）。大多数阶段并不会写入 GPU 资源，它们只是将输出传递到下一阶段；例如，顶点着色器阶段（Vertex Shader Stage）读取输入装配阶段的数据，然后进行处理，接着将结果输出到几何着色器阶段（Geometry Shader Stage）。在随后的几节中，我们将分别讲解渲染管线的各个阶段。

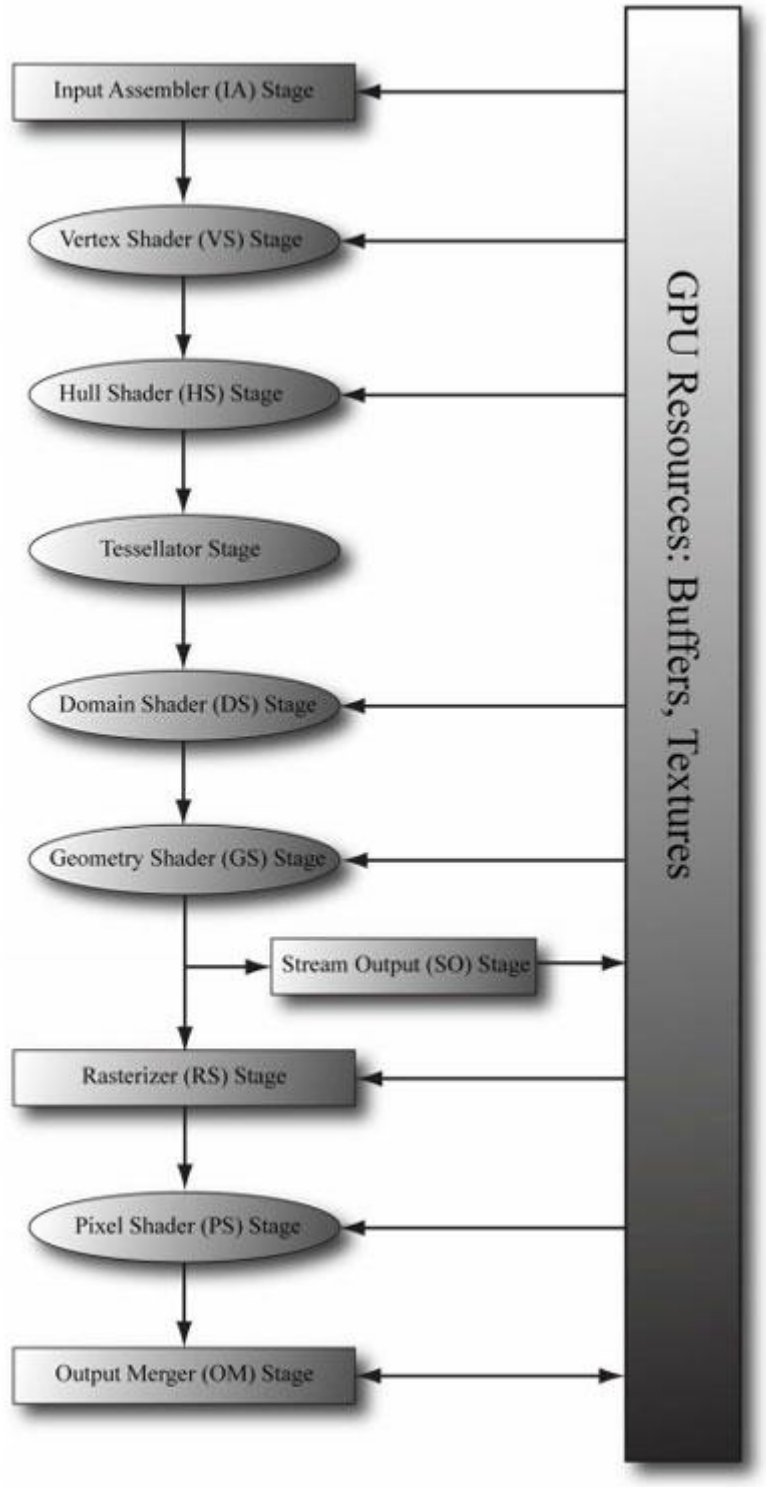



**图 5.11：渲染管线的各个阶段**

# 5.5 输入装配阶段

输入装配（Input Assembler，简称 IA）阶段从内存读取几何数据（顶点和索引）并将这些数据组合为几何图元（例如，三角形、直线）。（索引将在随后的小节中讲解。简单地说，索引规定了顶点的组织形式，解释了该以何种方式组成图元。）

## 5.5.1 顶点

从数学上讲，三角形的顶点位于两条边相交的位置上；而直线的顶点是端点。对于一个单个点来说，点本身就是顶点。图 5.12 说明了顶点的几何图形。

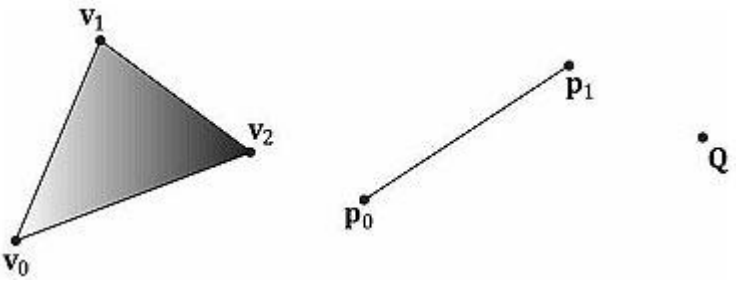



**图 5.12 由三个顶点 $V_1$、$V_2$、$V_3$ 组成的三角形；由两个顶点 $p_0$、$p_1$ 表示的直线；由顶点表示的点。**

图 5.12 说明顶点只是几何图元中的一个特殊的点。不过，在 Direct3D 中，顶点具有更多含义。本质上，Direct3D 中的顶点由空间位置和各种附加属性组成；例如，在第 7 章中，我们会在顶点中添加法线向量实现光照，在第 8 章中，在顶点中添加纹理坐标实现纹理。Direct3D 可以让我们灵活地建立属于我们自己的顶点格式(例如，允许我们定义顶点的分量)。在本书中，我们会基于要绘制的效果定义一些不同的顶点格式。

## 5.5.2 图元拓扑

顶点是以一个叫做顶点缓冲区的 Direct3D 数据结构的形式绑定到图形管线的。顶点缓冲区只是在连续的内存中存储了一个顶点列表。它并没有说明以何种方式组织顶点，形成几何图元。例如，是应该把顶点缓冲区中的每两个顶点解释为一条直线，还是应该把顶点缓冲区中的每三个顶点解释为一个三角形？我们通过指定图元拓扑来告诉 Direct3D 以何种方式组成几何图元：

```
void ID3D11Device::IASetPrimitiveTopology(
    D3D11_PRIMITIVE_TOPOLOGY Topology);

typedef enum D3D11_PRIMITIVE_TOPOLOGY
{
    D3D11_PRIMITIVE_TOPOLOGY_UNDEFINED = 0,
    D3D11_PRIMITIVE_TOPOLOGY_POINTLIST = 1,
    D3D11_PRIMITIVE_TOPOLOGY_LINELIST = 2,
    D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP = 3,
```

```
    D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST = 4,
    D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP = 5,
    D3D11_PRIMITIVE_TOPOLOGY_LINELIST_ADJ = 10,
    D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP_ADJ = 11,
    D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ = 12,
    D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP_ADJ= 13,
    D3D11_PRIMITIVE_TOPOLOGY_1_CONTROL_POINT_PATCHLIST = 33,
    D3D11_PRIMITIVE_TOPOLOGY_2_CONTROL_POINT_PATCHLIST = 34,
    .
    .
    .
    D3D11_PRIMITIVE_TOPOLOGY_32_CONTROL_POINT_PATCHLIST = 64,
} D3D11_PRIMITIVE_TOPOLOGY;
```

所有的绘图操作以当前设置的图元拓扑方式为准。在没有改变拓扑方式之前，当前设置的拓扑方式会一直有效。下面的代码说明了一点：

```
md3dDevice->IASetPrimitiveTopology(
    D3D11_PRIMITIVE_TOPOLOGY_LINELIST);
/* ...draw objects using line list... */
md3dDevice->IASetPrimitiveTopology(
    D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
/* ...draw objects using triangle list... */
md3dDevice->IASetPrimitiveTopology(
    D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP);
/* ...draw objects using triangle strip... */
```

下面的各小节会详细描述各种不同的图元拓扑方式。在本书中，我们主要使用三角形列表。

### 5.5.2.1 点列表

点列表（point list）由 **D3D11_PRIMITIVE_TOPOLOGY_POINTLIST** 标志值表示。当使用点列表时，每个顶点都会被绘制为一个独立的点，如图 5.13a 所示。

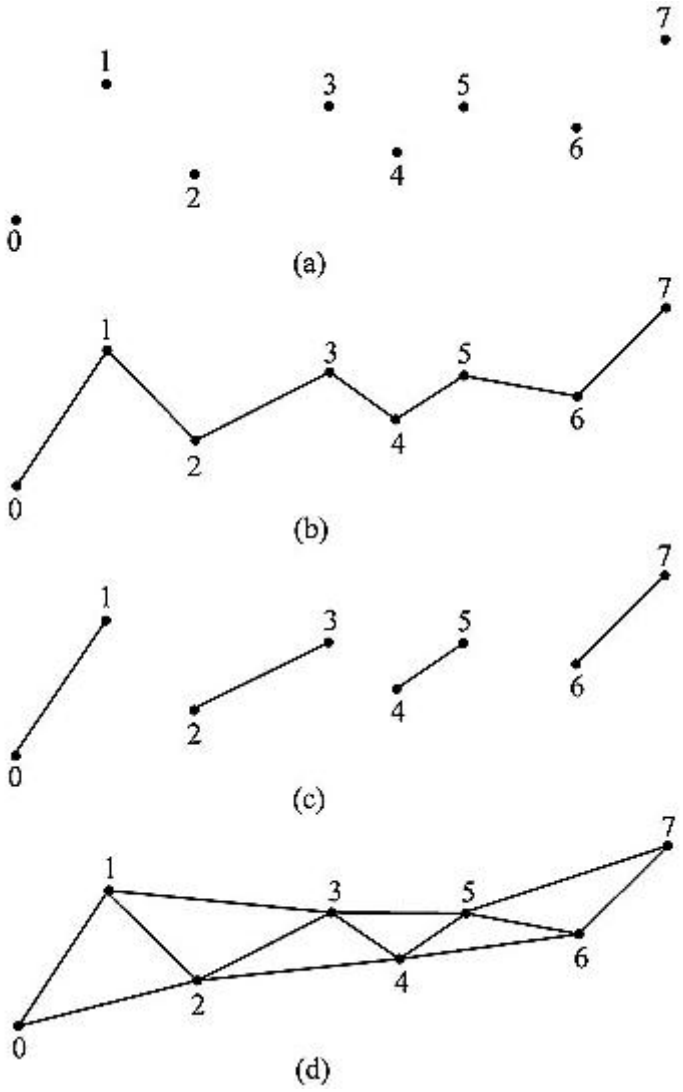



**图 5.13 (a)点列表。(b)线带。(c)线列表。(d)三角形带。**

## 5.5.2.2 线带

线带（line strip）由 **D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP** 标志值表示。当使用线带时，前后相邻的两个顶点会形成一条直线（参见图 5.13b）；这样，n+1 个顶点可以形成 n 条直线。

## 5.5.2.3 线列表

线列表（line list）由 **D3D11_PRIMITIVE_TOPOLOGY_LINELIST** 标志值表示。当使用线列表时，每两个顶点会形成一条独立的直线（参见图 5.13c）；这样，2n 个顶点可以形成 n 条直线。线列表和线带之间的区别是线列表中的直线可以断开，而线带中的直线会自动连在一起；因为内部的每个顶点由两条直线共享，所以线带使用的顶点数量更少。

## 5.5.2.4 三角形带

三角形带（triangle strip）由 **D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP** 标志值表示。当使用三角形带时，顶点会按照图 5.13d 所示的带状方式形成连续的三角形。我们可以看到顶点由相邻的三角形共享，n 个顶点可以形成 n-2 个三角形。

**注意**：偶数三角形和奇数三角形的顶点环绕顺序不同，由此会产生背面消隐问题（参见 5.10.2 节）。为了解决一问题，GPU 会在内部交换偶数三角形的前两个顶点的顺序，以使它们的环绕顺序与奇数三角形相同。

## 5.5.2.5 三角形列表

三角形列表（triangle list）由 **D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST** 标志值表示。当使用三角形列表时，每三个顶点会形成一个独立的三角形（参见图 5.14a）；这样，3n 个顶点可以形成 n 个三角形。三角形列表与三角形带之间的区别是：三角形列表中的三角形可以断开，而三角形带中的三角形会自动连在一起。

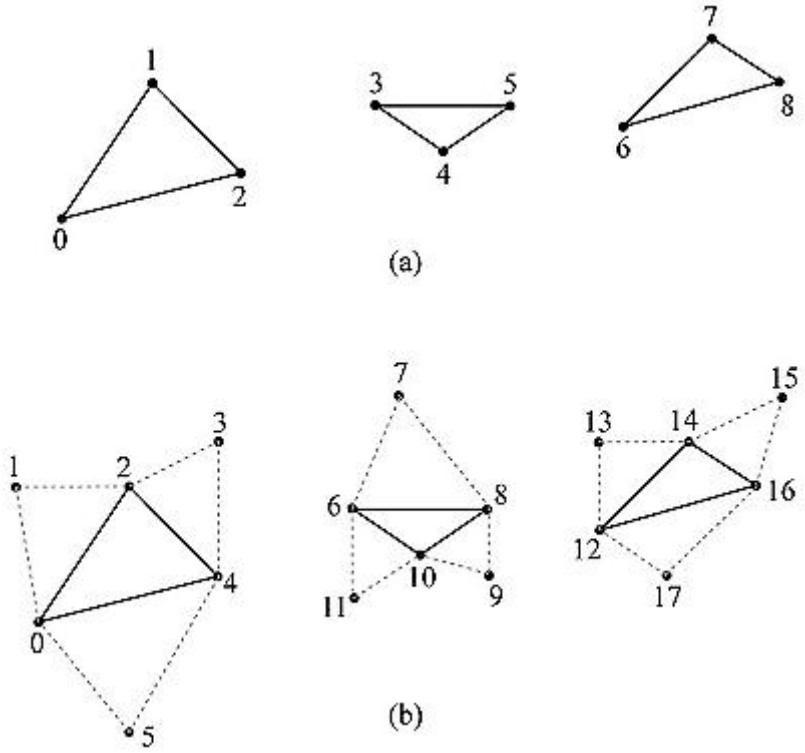



**图 5.14 (a)三角形列表。(b)邻接三角形列表——可以看到每个三角形需要 6 个顶点来描述它和它的邻接三角形。所以，6n 个顶点可以形成 n 个带有邻接信息的三角形。**

## 5.5.2.6 带有邻接信息的图元

在包含邻接信息的三角形列表中，每个三角形都有与之相邻的 3 个邻接三角形；图 5.14b 说明了这些三角形的定义方式。它们主要用于几何着色器，因为某些几何着色算法需要访问邻接三角形。为了实现这些算法，邻接三角形必须与原三角形一起通过顶点/索引缓冲区提

交给管线。通过指定 **D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ** 拓扑标志值可以使管线知道如何从顶点缓冲区中构建三角形以及它的邻接三角形。注意，邻接图元顶点只能作为几何着色器的输入数据——它们不会被绘制出来。如果没有几何着色器，那么邻接图元也不会被绘制出来。

线列表、线带和三角形带也可以包含邻接图元；详情请参见 SDK 文档。

#### 5.5.2.7 控制点面片列表

**D3D11_PRIMITIVE_TOPOLOGY_N_CONTROL_POINT_PATCHLIST** 拓扑标志表示将顶点数据作为 N 控制点的面片列表，这些点用于（可选）图形管线的曲面细分阶段（tessellation stage），我们要到第 13 章才会讨论到它。

### 5.5.3 索引

如前所述，三角形是构成 3D 物体的基本单位。下面的代码示范了使用三角形列表来构建四边形和八边形的顶点数组（即，每三个顶点构成一个三角形）。

```
Vertex quad[6] ={
    v0, v1, v2, // Triangle0
    v0, v2, v3, // Triangle1
};

Vertex octagon[24] ={
    v0, v1, v2, // Triangle0
    v0, v2, v3, // Triangle1
    v0, v3, v4, // Triangle2
    v0, v4, v5, // Triangle3
    v0, v5, v6, // Triangle4
    v0, v6, v7, // Triangle5
    v0, v7, v8, // Triangle6
    v0, v8, v1    // Triangle 7
};
```

**注意**：三角形的顶点顺序非常重要，我们将该顺序称为环绕顺序（winding order）；详情请参见 5.10.2 节。

如图 5.15 所示，构成 3D 物体的三角形会共享许多相同的顶点。更确切地说，在图 5.15a 中，四边形的每个三角形都会共享顶点 $\mathbf{v}_0$ 和 $\mathbf{v}_2$。当复制两个顶点时问题并不明显，但是在八边形的例子中问题就比较明显了（图 5.15b），八边形的每个三角形都会复制中间的顶点 $\mathbf{v}_0$，而且边缘上的每个顶点都由相邻的两个三角形共享。通常，复制顶点的数量会随着模型细节和复杂性的提高而骤然上升。

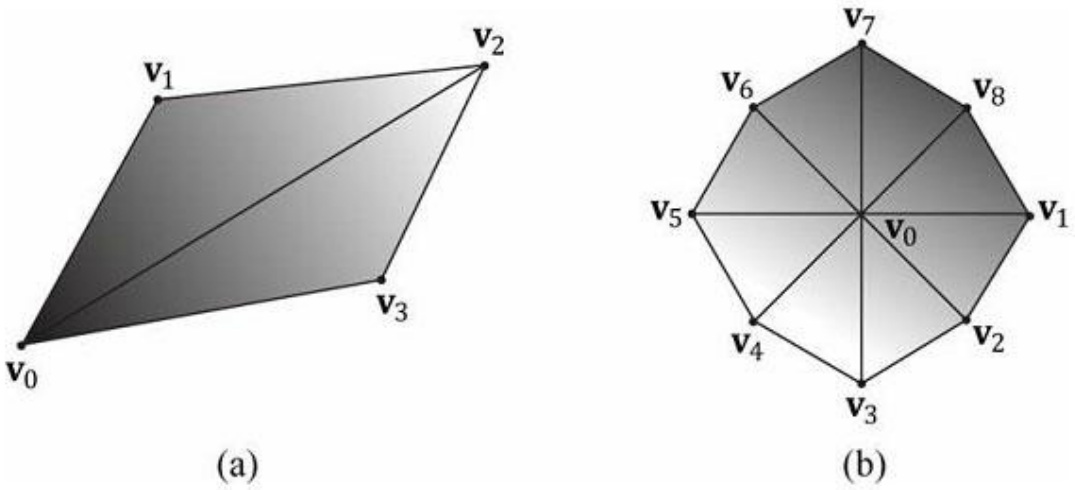



**图 5.15 (a)由 2 个三角形构成的四边形。(b)由 8 个三角形构成的八边形。**

我们不希望对顶点进行复制，主要有两个原因：

**3.** 增加内存需求量。（为什么要多次存储相同的顶点数据？）

**4.** 增加图形硬件的处理负担。（为什么要多次处理相同的顶点数据？）

三角形带在一定程度上可以解决复制顶点问题，但是几何体必须按照带状方式组织，实现起来难度较大。相比之下，三角形列表具有更好的灵活性（三角形不必彼此相连），如果能找到一种方法，即移除复制顶点，又保留三角形列表的灵活性，那么会是一件非常有价值的事情。索引（index）可以解决一问题。它的工作原理是：我们创建一个顶点列表和一个索引列表。顶点列表包含所有唯一的顶点，而索引列表包含指向顶点列表的索引值，这些索引定义了顶点以何种方式组成三角形。回顾图 5.15 中的图形，四边形的顶点列表可以这样创建：

```
Vertex v[4] = {v0, v1, v2, v3};
```

而索引列表需要定义如何将顶点列表中的顶点放在一起，构成两个三角形。

```
UINT indexList[6] = {0, 1, 2,    // Triangle0
    0, 2, 3}; // Triangle 1
```

在索引列表中，每 3 个元素表示一个三角形。所以上面的索引列表的含义为：“使用顶点 **v[0]**、**v[1]**、**v[2]**构成三角形 0，使用顶点 **v[0]**、**v[2]**、**v[3]**构成三角形 1”。

与之类似，八边形的顶点列表可以这样创建：

```
Vertex v[9] = {v0, v1, v2, v3, v4, v5, v6, v7, v8};
```

索引列表为：

```
UINT indexList[24] = {
    0, 1, 2,    // Triangle 0
    0, 2, 3,    // Triangle 1
    0, 3, 4,    // Triangle 2
    0, 4, 5,    // Triangle 3
    0, 5, 6,    // Triangle 4
    0, 6, 7,    // Triangle 5
    0, 7, 8,    // Triangle 6
    0, 8, 1    // Triangle7
};
```

当顶点列表中的唯一顶点得到处理之后，显卡可以使用索引列表把顶点放在一起构成三

角形。我们将“复制问题”转嫁给了索引列表，但是这种复制是可以让人接受的。因为：

3. 索引是简单的整数，不像顶点结构体那样占用很多内存（顶点结构体包含的分量越多，占用的内存就越多）。
4. 通过适当的顶点缓存排序，图形硬件不必重复处理顶点（在绝大多数的情况下）。

# 5.6 顶点着色器阶段

在完成图元装配后，顶点将被送往顶点着色器（vertex shader）阶段。顶点着色器可以被看成是一个以顶点作为输入输出数据的函数。每个将要绘制的顶点都会通过顶点着色器推送至硬件；实际上，我们可以概念性地认为在硬件上执行了如下代码：

```
for(UINT i = 0; i < numVertices; ++i)
    outputVertex[i] = VertexShader(inputVertex[i]);
```

顶点着色器函数由我们自己编写，但是它会在 GPU 上运行，所以执行速度非常快。

许多效果，比如变换（transformation）、光照（lighting）和置换贴图映射（displacement mapping）都是由顶点着色器来实现的。记住，在顶点着色器中，我们不仅可以访问输入的顶点数据，也可以访问在内存中的纹理和其他数据，比如变换矩阵和场景灯光。

我们将会在本书中看到许多不同的顶点着色器示例；当读完本书时，读者会对顶点着色器的功能有一个全面的认识。不过，我们的第一个示例会比较简单，只是用顶点着色器实现顶点变换。在随后的小节中，我们将讲解各种常用的变换算法。

## 5.6.1 局部空间和世界空间

现在让我们来假设一个情景：你正在参与一部影片的拍摄工作，剧组要为拍摄某些特殊效果而搭建一个微缩场景。你的具体任务是搭建一座小桥。现在，你不能在场景中搭建小桥，你必须另选地点，在远离场景的地方建立工作台，制作小桥，以避免弄乱场景中的其他微缩物品。当小桥建成后，你要按照正确的位置和角度把小桥放到场景中。

3D 美术师在创建 3D 场景时也采用同样的工作方式。他们不在全局场景坐标系（world space，世界空间）中建立物体，而是在局部坐标系（local space，局部空间）中建立物体；局部坐标系是最常用的实用坐标系，它的原点接近于物体中心，坐标轴的方向与物体的方向对齐。在完成 3D 模型的制作之后，美术师会将模型放到全局场景中；通过计算局部坐标系相对于世界坐标系的原点和轴向，实现相应的坐标转换变换（参见图 5.16，并回顾 3.4 节的内容）。将坐标从局部坐标系转换到世界坐标系的过程称为世界变换（world transform），相应的变换矩阵称为世界矩阵（world matrix）。当所有的物体都从局部空间变换到世界空间后，这些物体就会位于同一个坐标系（世界空间）中。如果你希望直接在世界空间中定义物体，那么可以使用单位世界矩阵（identity world matrix）。

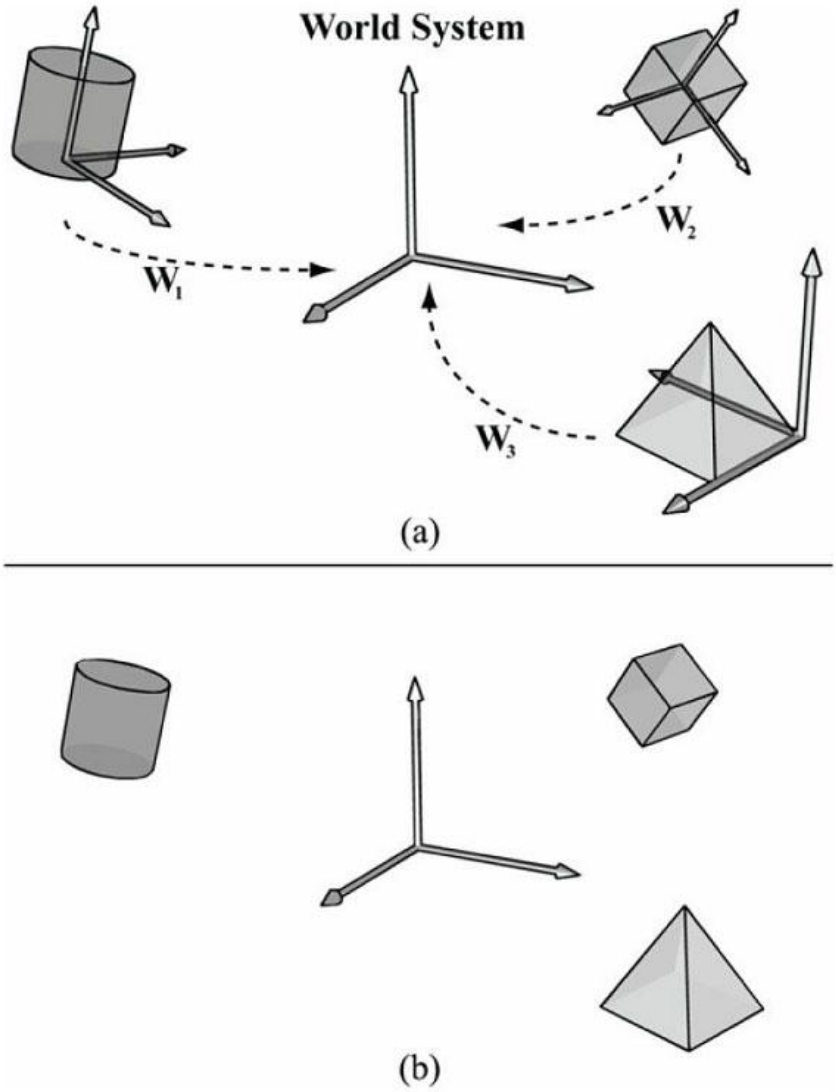



**图 5.16 (a)物体的每个顶点都是相对于它们自己的局部坐标系来定义的。我们根据物体在场景中的位置和方向来定义每个局部坐标系相对于世界坐标系的位置和方向。然后我们执行坐标转换变换，将所有坐标转换到世界坐标系中。(b)在世界变换后，所有物体的顶点都会位于同一个世界坐标系中。**

根据模型自身的局部坐标系定义模型，有以下几点好处：

**1．**简单易用。比如，在局部坐标系中，坐标系的原点通常会与物体的中心对齐，而某个主轴可能正是物体的对称轴。又如，当我们使用局部坐标系时，由于坐标系的原点与立方体的中心对齐，坐标轴垂直于立方体表面，所以可以更容易地描述立方体的顶点（参见图5.17）。

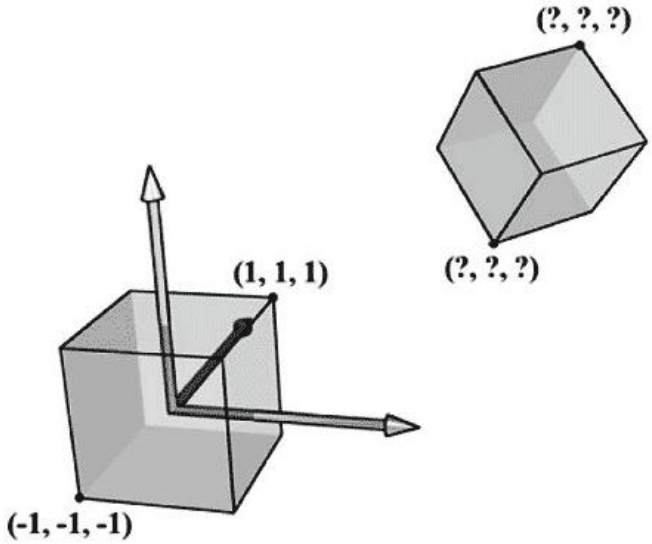



**图 5.17 当立方体的中心位于坐标系原点且与轴对齐（axis-aligned）时，可以很容易地描述立方体的顶点。当立方体位于坐标系的任意一个位置和方向上时，就很难描述这些坐标了。所以，当我们创建模型时，总是选择一种与物体位置接近且与物体方向对齐的实用坐标系。**

**2**．物体可以在多个场景中重复使用，对物体坐标进行相对于特定场景的硬编码是毫无意义的事情。较好的做法是：在局部坐标系中存储物体坐标，通过坐标转换矩阵将物体从局部坐标系变换到世界坐标系，建立物体与场景之间的联系。

**3**．最后，有时我们会多次绘制相同的物体，只是物体的位置、方向和大小有所不同（比如，将一棵树重绘多次形成一片森林）。在这种情况下，我们只需要一个相对于局部坐标系的单个副本，而不是多次复制物体数据，为每个实例创建一个副本。当绘制物体时，我们为每个物体指定不同的世界矩阵，改变它们在世界空间中的位置、方向和大小。这种方法叫做**instancing**。

如 3.4.3 节所述，世界矩阵描述的是一个物体的局部空间相对于世界空间的原点位置和坐标轴方向，这些坐标可以存放在一个行矩阵中。设 $\mathbf{Q}_w = (Q_x, Q_y, Q_z, 1)$、$\mathbf{u}_w = (u_x, u_y, u_z, 0)$、$\mathbf{v}_w = (v_x, v_y, v_z, 0)$、$\mathbf{w}_w = (w_x, w_y, w_z, 0)$分别表示局部空间相对于世界空间的原点、$x$ 轴、$y$ 轴、$z$ 轴的齐次坐标，由 3.4.3 节可知，从局部空间到世界空间的坐标转换矩阵为：

$$\mathbf{W} = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ Q_x & Q_y & u_z & 1 \end{pmatrix}$$

# 示例

假设一个正方形的顶点局部坐标在(−0.5,0,−0.5)和(0.5,0,0.5)之间，将它的边长变为 2，顺时针旋转 45°，并放置在世界空间的（10,0,10）坐标上，那如何求它在世界空间中的坐标呢？我们需要构建 **S**，**R**，**T** 矩阵，世界矩阵 **W** 如下所示：

$$\mathbf{S}=\begin{pmatrix}2&0&0&0\\0&1&0&0\\0&0&2&0\\0&0&0&1\end{pmatrix}\quad \mathbf{R}=\begin{pmatrix}\sqrt{2}/2&0&-\sqrt{2}/2&0\\0&1&0&0\\\sqrt{2}/2&0&\sqrt{2}/2&0\\0&0&0&1\end{pmatrix}\quad \mathbf{T}=\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\10&0&10&1\end{pmatrix}$$

$$\mathbf{W}=\mathbf{SRT}=\begin{pmatrix}\sqrt{2}&0&-\sqrt{2}&0\\0&1&0&0\\\sqrt{2}&0&\sqrt{2}&0\\10&0&10&1\end{pmatrix}$$

根据 3.5 节的讲解，**W** 中的行表示相对于世界空间的局部坐标系；即 $\mathbf{u}_w=(\sqrt{2}\ ,0,-\sqrt{2}\ ,0)$，$\mathbf{v}_w=(0,1,0\ ,0)$，$\mathbf{w}_w=(\sqrt{2}\ ,0,\sqrt{2}\ ,0)$，$\mathbf{Q}_w=(10,0,10,1)$。当我们使用 W 将局部坐标系转换到世界坐标系时，正方形就会处在期望的位置上（见图 5.18）。

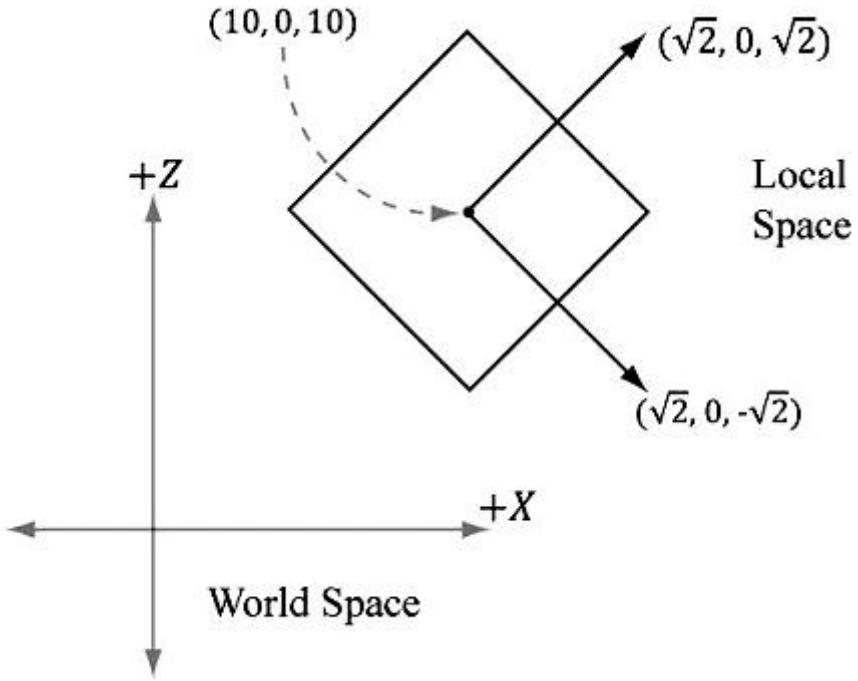



**图 5.18 世界矩阵的行向量表示相对于世界空间的局部坐标系。**

这个例子的要点是无需计算 $\mathbf{Q}_w$，$\mathbf{u}_w$，$\mathbf{v}_w$，$\mathbf{w}_w$ 直接获得世界矩阵，而是通过组合一系列简单的变换矩阵获得世界矩阵，这通常比直接求解 $\mathbf{Q}_w$，$\mathbf{u}_w$，$\mathbf{v}_w$，$\mathbf{w}_w$ 简单。我们只需确定：物体在世界空间中的尺寸多大，在世界空间中的朝向如何，我们要将该物体放置在世界空间中的何处。

还有一种考虑世界变换的方式是：只考虑局部坐标并把它作为世界坐标对待（这相当于使用一个单位矩阵作为世界变换矩阵）。这样，如果物体建模时就位于局部坐标的原点，那么它也在世界空间的坐标原点。通常，世界空间的坐标原点并不是我们想放置物体的位置，所以，对每个物体，我们只要施加一组变换用于缩放、选择、平移，将物体放置在世界空间中的确定位置。从数学上来说，这与将矩阵从局部空间转换到世界空间的变换效果是相同的。

## 5.6.2 观察空间

为了生成场景的 2D 图像，我们必须在场景中放置一架虚拟摄像机。虚拟摄像机指定了

观察者可以看到的场景范围，或者说是我们所要生成的 2D 图像所显示的场景范围。我们把一个局部坐标系（称为观察空间、视觉空间或摄像机空间）附加在摄像机上，如图 5.19 所示；该坐标系以摄像机的位置为原点，以摄像机的观察方向为 $z$ 轴正方向，以摄像机的右侧为 $x$ 轴，以摄像机的上方为 $y$ 轴。在渲染管线的随后阶段中，使用观察空间来描述顶点比使用世界空间来描述顶点要方便的多。从世界空间到观察空间的坐标转换称为观察变换（view transform），相应的矩阵称为观察矩阵（view matrix）。

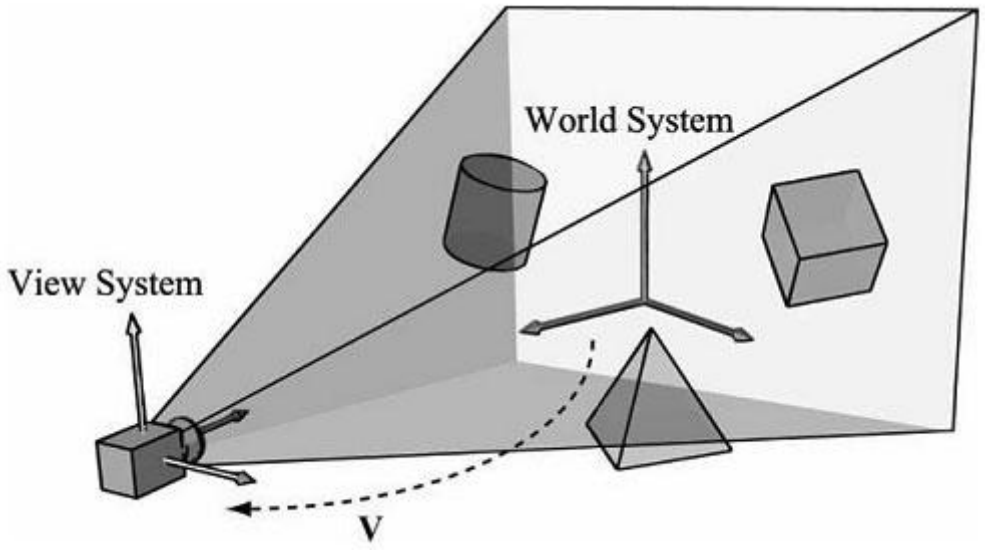



**图 5.19 将相对于世界空间的顶点坐标转换为相对于摄像机空间的顶点坐标。**

设 $\mathbf{Q}_w=(Q_x, Q_y, Q_z, 1)$、$\mathbf{u}_w=(u_x, u_y, u_z, 0)$、$\mathbf{v}_w=(v_x, v_y, v_z, 0)$、$\mathbf{w}_w=(w_x, w_y, w_z, 0)$分别表示观察空间相对于世界空间的原点、$x$ 轴、$y$ 轴、$z$ 轴的齐次坐标，我们由 3.4.3 节可知，从观察空间到世界空间的坐标转换矩阵为：

$$\mathbf{W}=\begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ Q_x & Q_y & u_z & 1 \end{bmatrix}$$

不过，这不是我们想要的结果。我们希望得到的是从世界空间到观察空间的反向变换。回顾 3.4.5 节可知，反向变换可由逆运算取得。也就是，$\mathbf{W}^{-1}$ 为世界空间到观察空间的变换矩阵。

世界坐标系和观察坐标系通常具有不同的位置和方向，所以凭直觉就可以知道 $\mathbf{W}=\mathbf{RT}$ 的含义（即，世界矩阵可以被分解为一个旋转矩阵和一个平移矩阵）。这种方式可以使逆矩阵的计算过程更简单一些：

$$\mathbf{V}=\mathbf{W}^{-1}=(\mathbf{RT})^{-1}=\mathbf{T}^{-1}\mathbf{R}^{-1}=\mathbf{T}^{-1}\mathbf{R}^{\mathrm{T}}$$

$$=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -Q_x & -Q_y & -Q_z & 1 \end{bmatrix}\begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}=\begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ -\mathbf{Q}\cdot\mathbf{u} & -\mathbf{Q}\cdot\mathbf{v} & -\mathbf{Q}\cdot\mathbf{w} & 1 \end{bmatrix}$$

所以，观察矩阵为：

$$\mathbf{V}=\begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ -\mathbf{Q}\cdot\mathbf{u} & -\mathbf{Q}\cdot\mathbf{v} & -\mathbf{Q}\cdot\mathbf{w} & 1 \end{bmatrix}$$

我们现在介绍一种更直观的方法来创建构成观察矩阵的向量。设 **Q** 为摄像机的位置，**T** 为摄像机瞄准的目标点，**j** 为描述世界空间“向上”方向的单位向量。参考图 5.20，摄像机的观察方向为：

$$\mathbf{w}=\frac{\mathbf{T}-\mathbf{Q}}{\|\mathbf{T}-\mathbf{Q}\|}$$

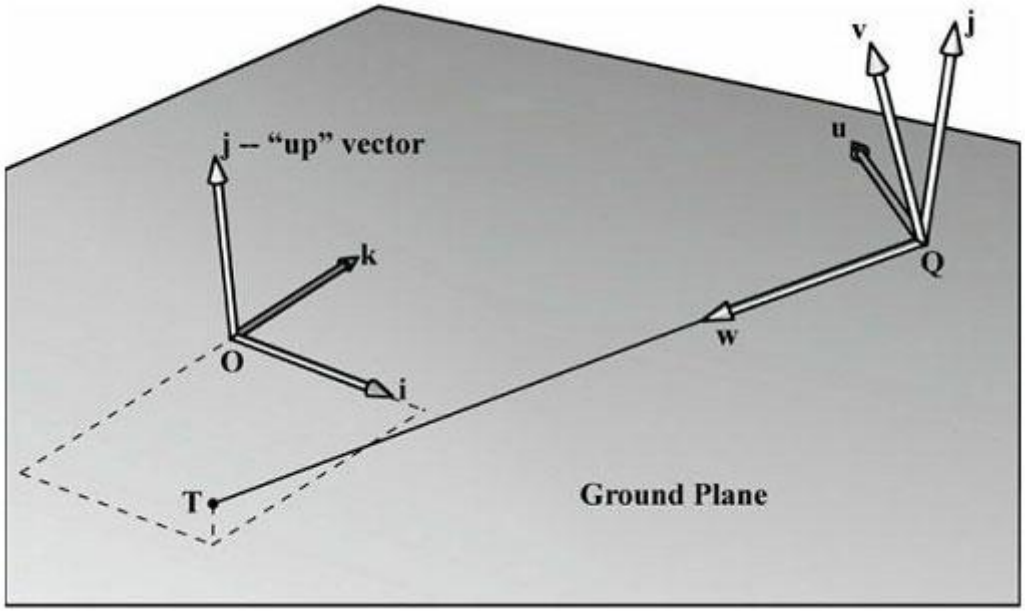



**图 5.20 通过指定摄像机的位置、目标点和世界“向上”向量来创建摄像机坐标系。**

向量 **w** 描述的是摄像机坐标系的 z 轴。指向 **w**“右边”的单位向量为：

$$\mathbf{u}=\frac{\mathbf{j}\times\mathbf{w}}{\|\mathbf{j}\times\mathbf{w}\|}$$

向量 **u** 描述的是摄像机坐标系的 *x* 轴。最后，描述摄像机坐标系 *y* 轴的向量为：

$$\mathbf{v}=\mathbf{w}\times\mathbf{u}$$

由于 **w** 和 **u** 是相互垂直的单位向量，所以 **w**×**u** 必定为单位向量，不需要对它做规范化处理。

这样，给出摄像机的位置、目标点和世界“向上”向量，我们就能够得到摄像机的局部坐标系，该坐标系可以用于创建观察矩阵。

XNA 库提供了如下函数，根据刚才描述的过程计算观察矩阵：

```
XMMATRIX XMMatrixLookAtLH( // Outputs resulting view matrix V
    FXMVECTOR EyePosition,   // Input camera position Q
    FXMVECTOR FocusPosition,   // Input target point T
    FXMVECTOR UpDirection);  // Input world up vector j
```

通常，世界坐标系的 *y* 轴就是“向上”方向，所以“向上”向量 **j** 通常设为(0, 1,0)。举

一个例子，假设摄像机相对于世界空间的位置为(5, 3, −10)，目标点为世界原点(0, 0,0)。我们可以使用如下代码创建观察矩阵：

```
XMVECTOR pos = XMVectorSet(5,3,-10,1.0f);
XMVECTOR target = XMVectorZero();
XMVECTOR up = XMVectorSet(0.0f,1.0f,0.0f,0.0f);
XMMATRIXV = XMMatrixLookAtLH(pos,target,up);
```

## 5.6.3 投影与齐次裁剪空间

到目前为止，我们已经知道了如何在场景中描述摄像机的位置和方向，下面我们来讲解如何描述摄像机所能看到的空间范围。该范围通过一个平截头体（frustum）来描述（图 5.21），它是一个在近平面处削去尖部的棱锥体。

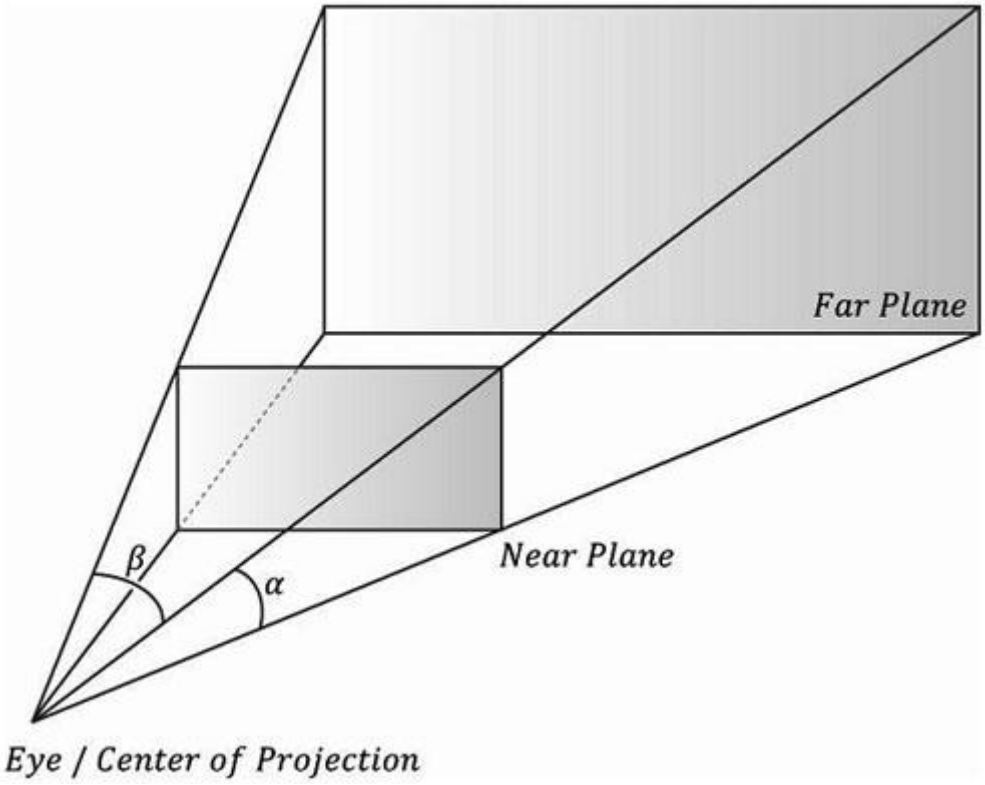



**图 5.21 平截头体描述了摄像机可以“看到”的空间范围。**

我们的下一个任务是把平截头体内的 3D 物体投影到 2D 投影窗口上。投影（projection）必须按照平行线汇集为零点的方式来实现，随着一个物体的 3D 深度增加，它的投影尺寸会越来越小；图 5.22 说明了透视投影的实现过程。我们将“从顶点连向观察点的直线”称为顶点的投影线。然后我们可以定义透视投影变换，将 3D 顶点 **v** 变换到它的投影线与 2D 投影平面相交的点 **v**′上；我们将 **v**′称为 **v** 的投影。对一个 3D 物体的投影就是对组成该物体的所有顶点的投影。

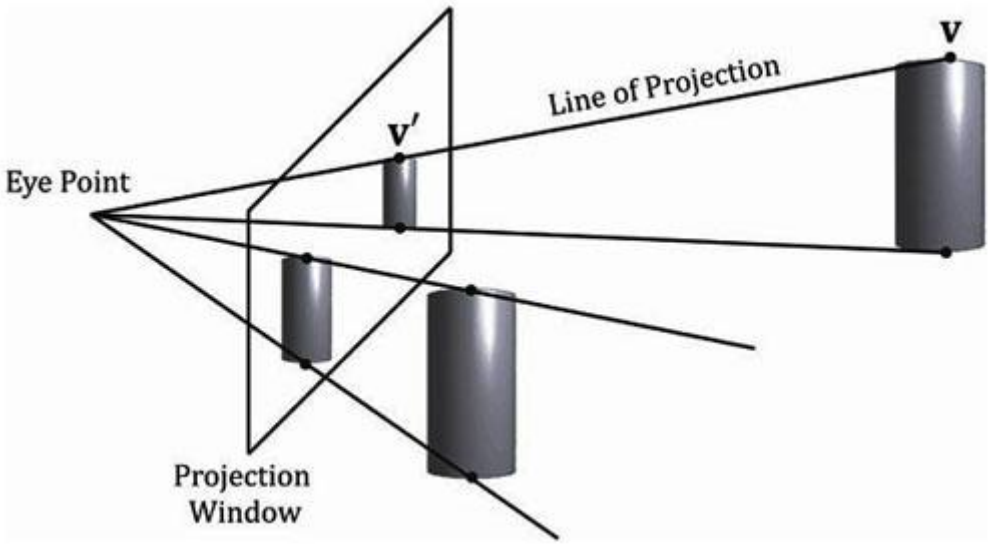



**图 5.22 在 3D 空间中，大小相同、深度不同的两个圆柱体。与观察点距离较近的圆柱体生成的投影较大。在平截头体内的物体可以被映射到投影窗口上；在平截头体外的物体可以映射到投影平面上，但是不会映射到投影窗口上。**

## 5.6.3.1 定义平截头体

我们可以在观察空间中使用如下 4 个参数来定义以原点为投影中心、以 $z$ 轴正方向为观察方向的平截头体：近平面 $n$、远平面 $f$、垂直视域角 $\alpha$ 和横纵比 $r$。注意，在观察空间中，近平面和远平面都平行于 $xy$ 平面；所以，我们只需要简单地指定它们沿 $z$ 轴方向到原点之间的距离即可表示这两个平面。横纵比由 $r = w/h$ 定义，其中 $w$ 表示投影窗口的宽度，$h$ 表示投影窗口的高度（单位由观察空间决定）。投影窗口本质上是指场景在观察空间中的 2D 图像。该图像最终会被映射到后台缓冲区中；所以，我们希望投影窗口的尺寸比例与后台缓冲区的尺寸比例保持相同。在大多数情况下，横纵比就是指后台缓冲区的尺寸比例（它是一个比例值，所以没有单位）。例如，当后台缓冲区的尺寸为 800×600 时，横纵比 $r = 800/600 \approx 1.333$。如果投影窗口的横纵比与后台缓冲区的横纵比不同，那么当投影窗口映射到后台缓冲区时，必然会出现比例失衡，导致图像变形（例如，投影窗口中的一个正圆会被拉伸为后台缓冲区中的一个椭圆）。

将水平视域角设为 $\beta$，它是由垂直视域角 $\alpha$ 和横纵比 $r$ 决定的。考虑图 5.23，分析一下如何通过 $\alpha$、$r$ 来求解 $\beta$。注意，投影窗口的实际尺寸并不重要，重要的只是横纵比。所以，我们将高度设定为 2，则对应的宽度为：

$$r=\frac{w}{h}=\frac{w}{2} \Rightarrow w=2r$$

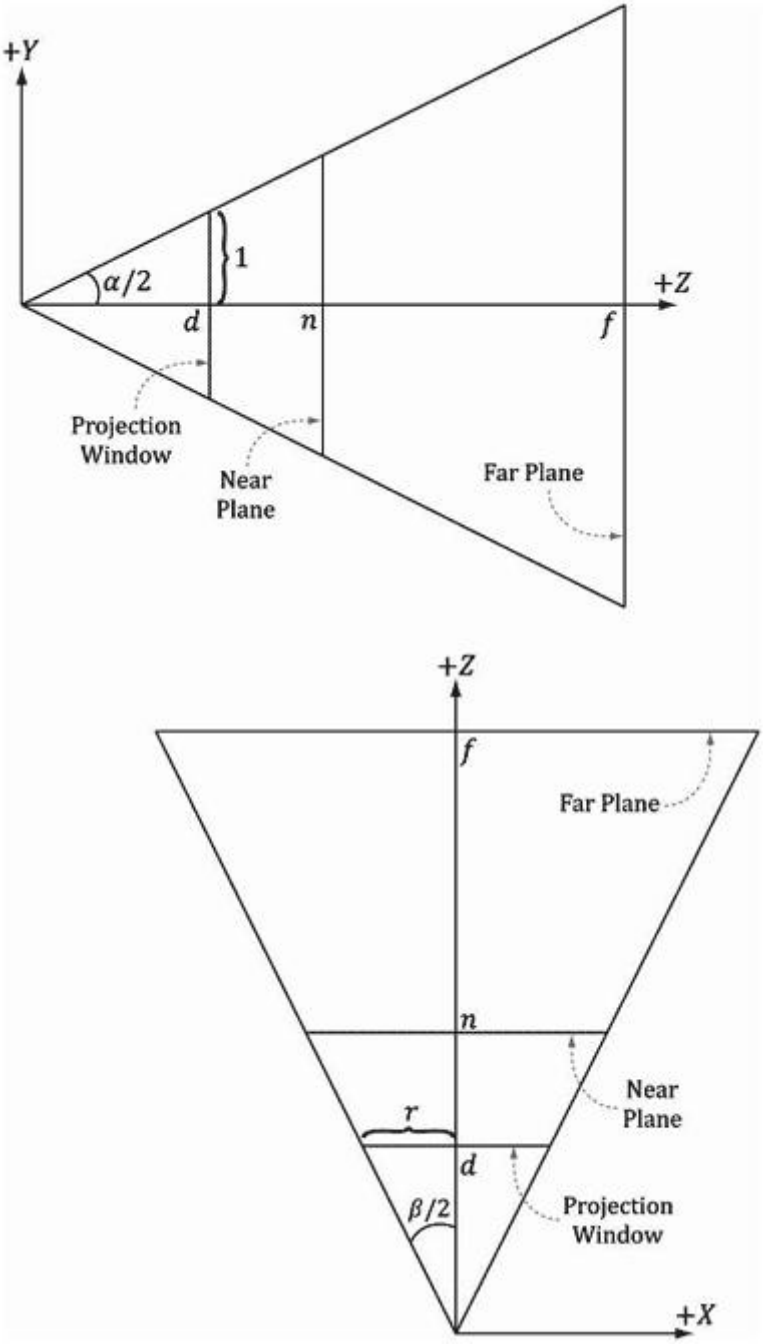



**图 5.23 给出垂直视域角$\alpha$和横纵比 $r$，求解水平视域角$\beta$。**

为了获得指定的垂直视域角$\alpha$，投影窗口必须放在与原点距离为 $d$ 的位置上：

$$\tan\frac{\alpha}{2} = \frac{1}{d} \Longrightarrow d = \cot\frac{\alpha}{2}$$

观察图 5.23 中的 xz 平面，可知：

$$\tan\frac{\beta}{2} = \frac{r}{d} = \frac{r}{\cot\frac{\alpha}{2}} = r \cdot \tan\frac{\alpha}{2}$$

所以，只要给出垂直视域角$\alpha$和横纵比 $r$，我们就能求出水平视域角$\beta$。

$$\beta = 2\tan^{-1}\left(r \cdot \tan\frac{\alpha}{2}\right)$$

## 5.6.3.2 对顶点进行投影

参见图 5.24。给出一个点 $(x, y, z)$，求它在投影平面 $z = d$ 上的投影点 $(x', y', d)$。通过分析 $x$、$y$ 坐标以及使用相似三角形，我们可以求出：

$$\frac{x'}{d} = \frac{x}{z} \Longrightarrow x' = \frac{xd}{z} = \frac{x \cot\frac{\alpha}{2}}{z} = \frac{x}{z\tan\frac{\alpha}{2}}$$

和

$$\frac{y'}{d} = \frac{y}{z} \Longrightarrow y' = \frac{yd}{z} = \frac{y \cot\frac{\alpha}{2}}{z} = \frac{y}{z\tan\frac{\alpha}{2}}$$

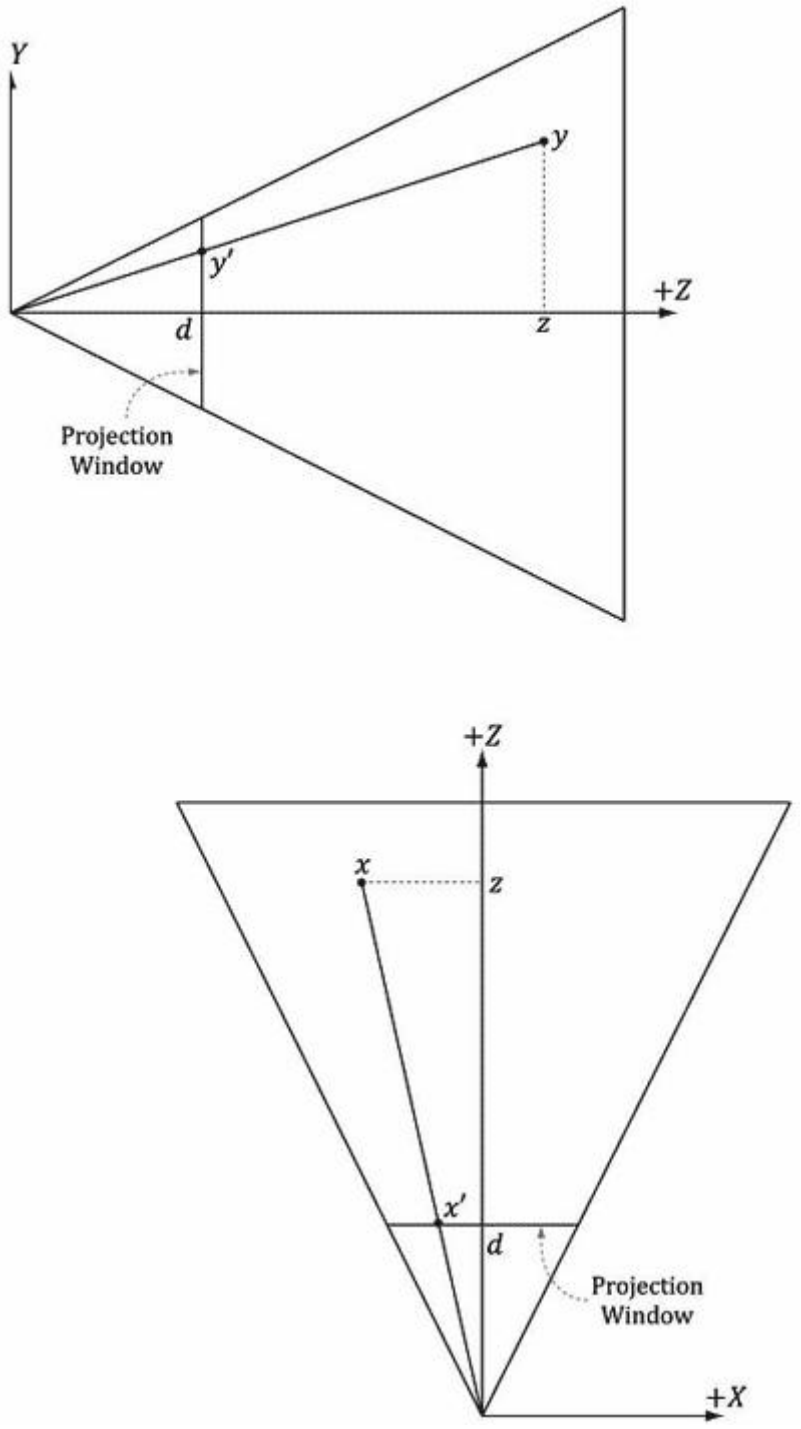



**图 5.24 相似三角形。**

当且仅当以下条件成立时，点(x, y , z)在平截头体内。

$-r \leqslant x' \leqslant r$

$-1 \leqslant y' \leqslant 1$

$n \leqslant z \leqslant f$

## 5.6.3.3 规范化设备坐标（NDC）

上一节我们讲解了如何在观察空间中计算点的投影坐标。在观察空间中，投影窗口的高

度为 2，宽度为 2*r*，其中 *r* 表示横纵比。这里存在的一个问题是：尺寸依赖于横纵比。这意味着我们必须为硬件指定横纵比，否则硬件将无法执行那些与投影窗口尺寸相关的运算（比如，将投影窗口映射到后台缓冲区）。如果我们能去除对横纵比的依赖性，那么会使相关的运算变得更加简单。为了解决一问题，我们将的投影 *x* 坐标从[−*r* , *r*] 区间缩放到[−1,1]区间：

$-r \leqslant x' \leqslant r$

$-1 \leqslant x'/r \leqslant 1$

在映射之后，*x*、*y* 坐标称为规范化设备坐标（normalized device coordinates，简称 NDC）（*z* 坐标还没有被规范化）。当且仅当以下条件成立时，点(*x* ,*y*, *z*)在平截头体内。

$-1 \leqslant x'/r \leqslant 1$

$-1 \leqslant y' \leqslant 1$

$n \leqslant z \leqslant f$

从观察空间到 NDC 空间的变换可以看成是一个单位转换。我们有这样一个关系式：在 *x* 轴上的一个 NDC 单位等于观察空间中的 *r* 个单位（即，1 ndc = *r* vs）。所以给出 *x* 观察空间单位，我们可以使用这个关系式来转换单位：

$$x \text{ vs} \frac{1 \text{ ndc}}{r \text{ vs}} = \frac{x}{r} \text{ ndc}$$

我们可以修改之前的投影公式，直接使用 NDC 空间中的 x、y 投影坐标：

$$x' = \frac{x}{rz\tan\frac{\alpha}{2}}$$

$$y' = \frac{y}{z\tan\frac{\alpha}{2}} \qquad \text{（方程 5.1）}$$

注意：在 NDC 空间中，投影窗口的高度和宽度都为 2。也就是说，现在的尺寸是固定的，硬件不需要知道横纵比，但是我们必须自己来完成投影坐标从观察空间到 NDC 空间的转换工作（图形硬件假定我们会完成一工作） 。

## 5.6.3.4 用矩阵来描述投影方程

为了保持一致，我们将用一个矩阵来描述投影变换。不过，方程 5.1 是非线性的，无法用矩阵描述。所以我们要使用一种“技巧”将它分为两部分来实现：一个线性部分和一个非线性部分。非线性部分要除以 *z*。我们会在下一节讨论“如何规范化 *z* 坐标”时讲解这一问题；现在读者只需要知道，我们会因为个除法操作而失去原始的 z 坐标。所以，我们必须在变换之前保存输入的 *z* 坐标；我们可以利用齐次坐标来解决一问题，将输入的 z 坐标复制给输出的 w 坐标。在矩阵乘法中，我们要将元素[2][3]设为 1、元素[3][3]设为 0（从 0 开始的索引）。我们的投影矩阵大致如下：

$$\mathbf{P}=\begin{bmatrix} \dfrac{1}{r\tan\dfrac{\alpha}{2}} & 0 & 0 & 0 \\ 0 & \dfrac{1}{\tan\dfrac{\alpha}{2}} & 0 & 0 \\ 0 & 0 & A & 1 \\ 0 & 0 & B & 0 \end{bmatrix}$$

注意矩阵中的常量 A 和 B（它们将在下一节讨论）；这些常量用于把输入的 z 坐标变换到规范化区间。将一个任意点(x , y, z,1)与该矩阵相乘，可以得到：

$$[x,y,z,1]\begin{bmatrix} \dfrac{1}{r\tan\dfrac{\alpha}{2}} & 0 & 0 & 0 \\ 0 & \dfrac{1}{\tan\dfrac{\alpha}{2}} & 0 & 0 \\ 0 & 0 & A & 1 \\ 0 & 0 & B & 0 \end{bmatrix}=\begin{bmatrix} \dfrac{x}{r\tan\dfrac{\alpha}{2}}, \dfrac{y}{\tan\dfrac{\alpha}{2}}, Az+B, z \end{bmatrix} \quad \text{（公式 5.2）}$$

在与投影矩阵（线性部分）相乘之后，我们要将每个坐标除以 w = z（非线性部分），得到最终的变换结果：

$$\begin{bmatrix} \dfrac{x}{r\tan\dfrac{\alpha}{2}}, \dfrac{y}{\tan\dfrac{\alpha}{2}}, Az+B, z \end{bmatrix} \xrightarrow{\text{除以w}} \begin{bmatrix} \dfrac{x}{rz\tan\dfrac{\alpha}{2}}, \dfrac{y}{z\tan\dfrac{\alpha}{2}}, A+\dfrac{B}{z}, 1 \end{bmatrix} \quad \text{（公式 5.3）}$$

顺便提一句，你可能会问：“如何处理除数为 0 的情况”；对于一问题我们不必担心，因为近平面总是大于 0 的，其他的点都会被裁剪掉（参见 5.9 节）。有时，与 w 相除的过程也称为透视除法（perspective divide）或齐次除法（homogeneous divide）。我们可以看到 $x$、$y$ 的投影坐标与方程 5.1 相同。

## 5.6.3.5 规范化深度值

你可能认为在投影之后可以丢弃原始的 3D z 坐标，因为所有的投影点已经摆放在 2D 投影窗口上，形成了我们最终看到的 2D 图像，不会再使用 3D z 坐标了。其实不然，我们仍然需要为深度缓存算法提供 3D 深度信息。就如同 Direct3D 希望我们把 x、y 投影坐标映射到一个规范化区间一样，Direct3D 也希望我们将深度坐标映射到一个规范化区间[0,1]中。所以，我们需要创建一个保序函数（order preserving function）g(x)把[*n*,*f*]区间映射到[0,1]区间。由于该函数是保序的，所以当 $z_1,z_2\in[n,f]$且 $z_1<z_2$ 时，必有 $g(z_1)<g(z_2)$。这样，即使深度值已经被变换过了，相对的深度关系还是会被完好无损地保留下来，我们依然可以在规范化区间中得到正确的深度测试结果，这就是我们要为深度缓存算法做的全部工作。

通过缩放和平移可以实现从[*n* ,*f*]到[0,1]的映射。但是，这种方式无法与我们当前的投影方程整合。我们可以从方程 5.3 中看到经过变换的 z 坐标为：

$$g(z) = A + \frac{B}{z}$$

我们现在需要让 A 和 B 满足以下条件：

- 条件 1：g(n) = A + B/n = 0（近平面映射为 0）
- 条件 2：g(f) = A +B/f = 1（远平面映射为 1）

由条件 1 得到 B 的结果为：B= −An。把它代入条件 2，得到 A 的结果为：

$$A + \frac{-An}{f} = 1$$

$$\frac{Af - An}{f} = 1$$

$$Af - An = f$$

$$A = \frac{f}{f - n}$$

所以，

$$g(x) = \frac{f}{f - n} - \frac{nf}{(f - n)z}$$

从 g(z)的曲线图（图 5.25）中可以看出，它会限制增长的幅度（保序）而且是非线性的。从图中我们还可以看到，区间中的大部分取值落在近平面附近。因此，大多数深度值被映射到了一个很窄的取值范围内。这会导致深度缓冲区出现精度问题（由于所能表示的数值范围有限，计算机将无法识别变换后的深度值之间的微小差异）。通常的建议是让近平面和远平面尽可能接近，把深度的精度性问题减小到最低程度。

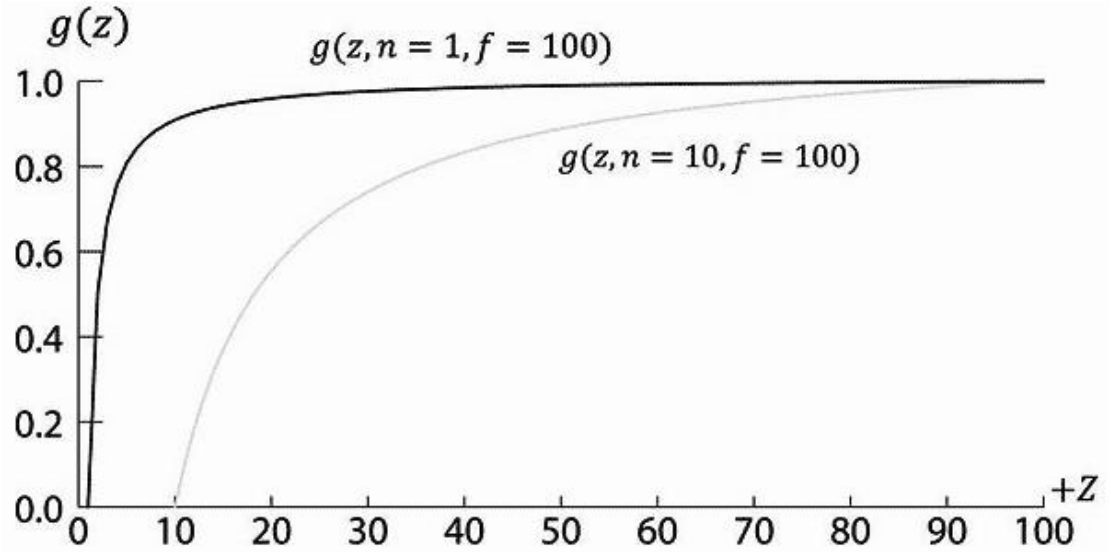



**图 5.25 相对于不同近平面的 g(z)曲线图。**

现在我们已经解出了 A 和 B，我们可以确定出完整的透视投影矩阵：

$$\mathbf{P}=\begin{bmatrix} \dfrac{1}{r\tan\dfrac{\alpha}{2}} & 0 & 0 & 0 \\ 0 & \dfrac{1}{\tan\dfrac{\alpha}{2}} & 0 & 0 \\ 0 & 0 & \dfrac{f}{f-n} & 1 \\ 0 & 0 & \dfrac{-nf}{f-n} & 0 \end{bmatrix}$$

在与投影矩阵相乘之后，进行透视除法之前，几何体所处的空间称为齐次裁剪空间（homogeneous clip space）或投影空间（projection space）。在透视除法之后，几何体所处的空间称为规范化设备空间（normalized device coordinates，简称 NDC）。

### 5.6.3.6 XMMatrixPerspectiveFovLH

透视投影矩阵可由如下 XNA 函数生成：

```
XMMATRIX XMMatrixPerspective FovLH(// returns projection matrix
    FLOAT FovAngleY, // vertical field of view angle in radians
    FLOAT AspectRatio, // aspect ratio =  width / height
    FLOAT NearZ, // distance to near plane
    FLOAT FarZ);  // distance to far plane
```

下面的代码片段示范了 XMMatrixPerspectiveFovLH 函数的使用方法。这里，我们将垂直视域角设为 45°，近平面 z 设为 1，远平面 z 设为 1000（这些长度是在观察空间中的）。

```
XMMATRIX P = XMMatrixPerspectiveFovLH(0.25f*MathX::Pi,
    AspectRatio(),1.0f,1000.0f);
```

横纵比要匹配窗口的横纵比：

```
float D3Dapp::AspectRatio() const
{
    return static_cast<float>(mClientWidth)/mClientHeight;
}
```

# 5.7 曲面细分阶段

曲面细分（Tessellation）是指通过添加三角形的方式对一个网格的三角形进行细分，这些新添加的三角形可以偏移到一个新的位置，让网格的细节更加丰富。（见图 5.26）。

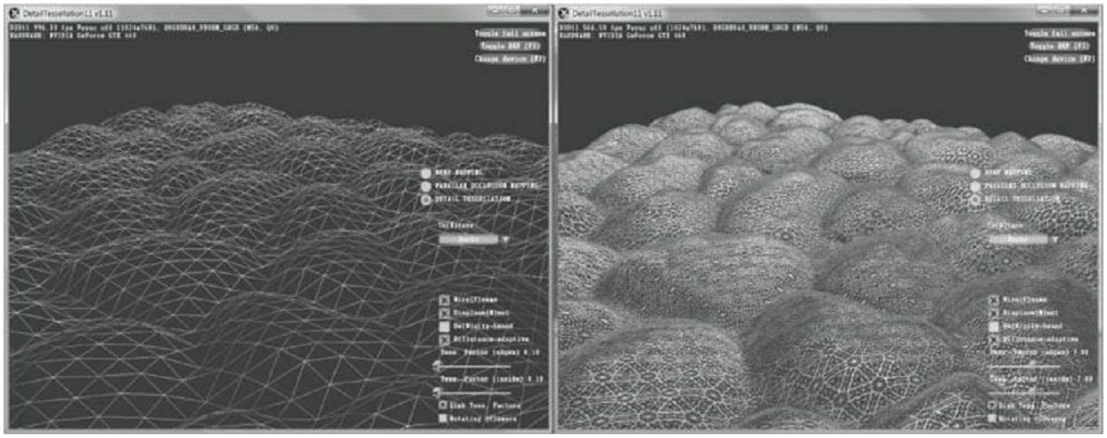


**图 5.26 左图是原始网格，右图是经过曲面细分处理后的网格**

下面是曲面细分的一些优点：

1．我们可以通过曲面细分实现细节层次（level -of-detai l，LOD），使靠近相机的三角形通过细分产生更多细节，而那些远离相机的三角形则保持不变。通过这种方式，我们只需在需要细节的地方使用更多的三角形就可以了。

2．我们可以在内存中保存一个低细节（低细节意味着三角形数量少）的网格，但可以实时地添加额外的三角形，这样可以节省内存。

3．我们可以在一个低细节的网格上处理动画和物理效果，而只在渲染时才使用细分过的高细节网格。

曲面细分阶段是 Direct3D 11 中新添加的，这样我们就可以在 GPU 上对几何体进行细分了。而在 Direct3D 11 之前，如果你想要实现曲面细分，则必须在 CPU 上完成，经过细分的几何体还要发送到 GPU 用于渲染。然而，将新的几何体从 CPU 内存发送到显存是很慢的，而且还会增加 CPU 的负担。因此，在 Direct3D 11 出现之前，曲面细分的方法在实时图形中并不流行。Direct3D 11 提供了一个可以完全在硬件上实现的曲面细分 API。这样曲面细分就成为了一个非常有吸引力的技术了。曲面细分阶段是可选的（即在需要的时候才使用它）。我们要在第 13 章才会详细介绍曲面细分。

# 5.8 几何着色器阶段

几何着色器阶段（geometry shader stage）是可选的，我们在第 11 章之前不会用到它，所以这里只做一个简短的概述。几何着色器以完整的图元作为输入数据。例如，当我们绘制三角形列表时，输入到几何着色器的数据是构成三角形的三个点。（注意，这三个点是从顶点着色器传递过来的。）几何着色器的主要优势是它可以创建或销毁几何体。例如，输入图元可以被扩展为一个或多个其他图元，或者几何着色器可以根据某些条件拒绝输出某些图元。这一点与顶点着色器有明显的不同：顶点着色器无法创建顶点，只要输入一个顶点，那么就必须输出一个顶点。几何着色器通常用于将一个点扩展为一个四边形，或者将一条线扩展为一个四边形。

我们可以在图 5.11 中看到一个“流输出（stream output）”箭头。也就是，几何着色器可以将顶点数据流输出到内存中的一个顶点缓冲区内，这些顶点可以在管线的随后阶段中渲染出来。这是一项高级技术，我们会在后面的章节中对它进行讨论。

**注意**：顶点位置在离开几何着色器之前，必须被变换到齐次裁剪空间。

# 5.9 裁剪阶段

我们必须完全丢弃在平截头体之外的几何体，裁剪与平截头体边界相交的几何体，只留下平截头体内的部分；图 5.27 以 2D 形式说明了一概念。

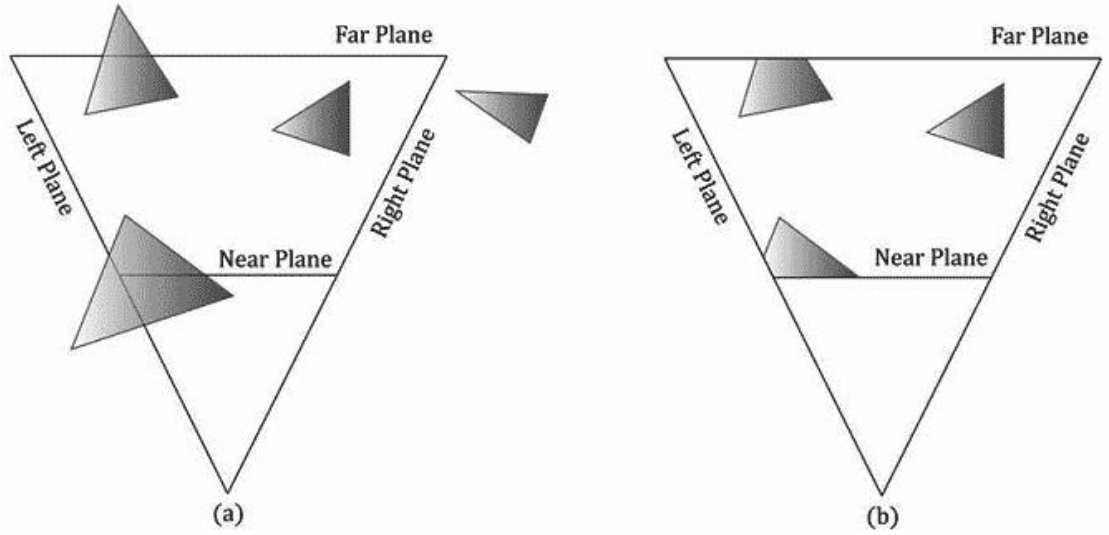



**图 5.27 (a)裁剪之前。(b)裁剪之后。**

我们可以将平截头体视为由 6 个平面界定的空间范围：顶、底、左、右、近、远平面。要裁剪与平截头体方向相反的多边形，其实就是逐个裁剪与每个平截头体平面方向相反的多边形，当裁剪一个与平面方向相反的多边形时（参见图 5.28），我们将保留平面正半空间中的部分，而丢弃平面负半空间中的部分。对一个与平面方向相反的凸多边形进行裁剪，得到的结果仍然会是一个凸多边形。由于硬件会自动完成所有的裁剪工作，所以我们不在这里讲解具体的实现细节；有兴趣的读者可以参阅[Sutherland74]，了解一下目前流行的 Sutherland-Hodgeman 裁剪算法。它基本思路是：求出平面与多边形边之间的交点，然后对顶点进行排序，形成新的裁剪后的多边形。

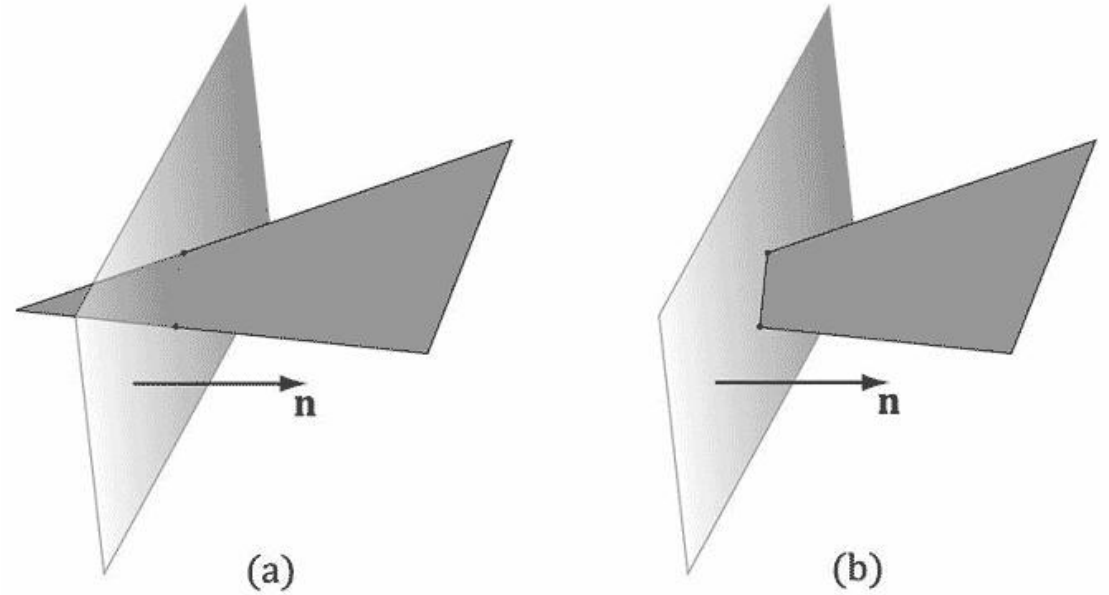



**图 5.28 (a)裁剪一个与平面方向相反的三角形。(b)裁剪后的三角形。注意，裁剪后的三角形已经不再是一个三角形了，它是一个四边形。所以，硬件必须将个四边形重新划分为三角**

**形，对于凸多边形来说这是一个非常简单的处理过程。**

[Blinn78]描述了如何在 4D 齐次空间中实现裁剪算法（图 5.29）。在透视除法之后，平截头体内的点$(x/w, y/w, z/w, 1)$将位于规范化设备空间，它的边界如下：

$-1 \leqslant x/w \leqslant 1$

$-1 \leqslant y/w \leqslant 1$

$0 \leqslant z/w \leqslant 1$

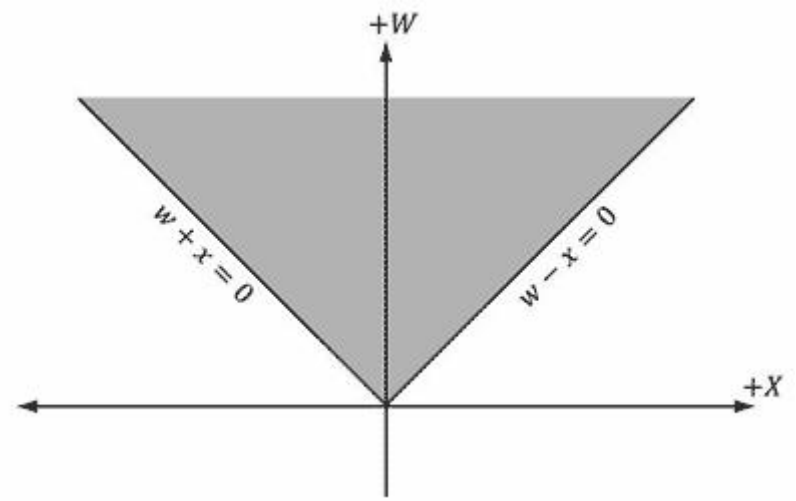



**图 5.29 齐次裁剪空间中 xw 平面上的截头体边界**

那么在透视除法之前，平截头体内的 4D 点$(x, y, z, w)$在齐次裁剪空间中的边界为：

$-w \leqslant x \leqslant w$

$-w \leqslant y \leqslant w$

$0 \leqslant z \leqslant w$

也就是，顶点被限定在以下 4D 平面构成的空间范围内：

左：$w = -x$

右：$w = x$

底：$w = -y$

顶：$w = y$

近：$z = 0$

远：$z = w$

只要我们知道齐次剪裁空间中的平截头体平面方程，我们就能使用任何一种裁剪算法（比如 Sutherland-Hodgeman）。注意，由线段/平面相交测试的数学推论可知，这个测试在$\mathbb{R}^4$也能使用，所以我们可以在齐次裁剪空间中进行 4D 点和 4D 平面的相交测试。

# 5.10 光栅化阶段

光栅化（rasterization）阶段的主要任务是为投影后的 3D 三角形计算像素颜色。

## 5.10.1 视口变换

在裁剪之后，硬件会自动执行透视除法，将顶点从齐次裁剪空间变换到规范化设备空间

（NDC）。一旦顶点进入 NDC 空间，构成 2D 图像的 2D *x*、*y* 坐标就会被变换到后台缓冲区中的一个称为视口的矩形区域内（回顾 4.2.8 节）。在该变换之后，*x*、*y* 坐标将以像素为单位。通常，视口变换不修改 *z* 坐标，因为 *z* 坐标还要由深度缓存使用，但是我们可以通过 **D3D11_VIEWPORT** 结构体的 **MinDepth** 和 **MaxDepth** 值修改 z 坐标的取值范围。**MinDepth** 和 **MaxDepth** 的值必须在 0 和 1 之间。

## 5.10.2 背面消隐

一个三角形有两个面。我们使用如下约定来区分这两个面。假设三角形的顶点按照 $\mathbf{v}_0$、$\mathbf{v}_1$、$\mathbf{v}_2$ 的顺序排列，我们这样来计算三角形的法线 **n**：

$\mathbf{e}_0 = \mathbf{v}_1 - \mathbf{v}_0$

$\mathbf{e}_1 = \mathbf{v}_2 - \mathbf{v}_1$

$$\mathbf{n} = \frac{\mathbf{e}_0 \times \mathbf{e}_1}{\|\mathbf{e}_0 \times \mathbf{e}_1\|}$$

带有法线向量的面为正面，而另一个面为背面。图 5.30 说明了这一概念。

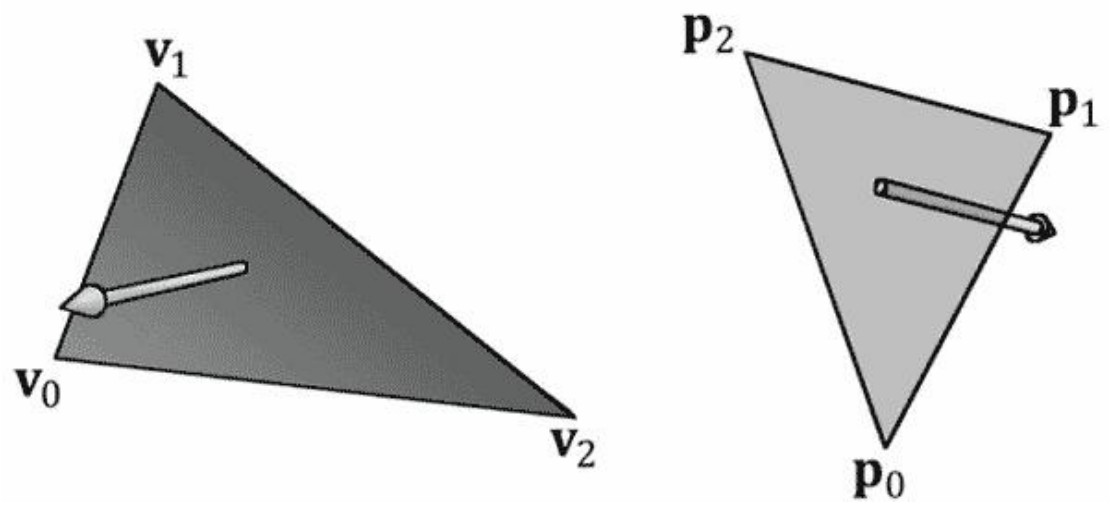



**图 5.30 左边的三角形正对我们的观察点，而右边的三角形背对我们的观察点。**

当观察者看到三角形的正面时，我们说三角形是朝前的；当观察者看到三角形的背面时，我们说三角形是朝后的。如图 5.30 所示，左边的三角形是朝前的，而右边的三角形是朝后的。而且，按照我们的观察角度，左边的三角形会按顺时针方向环绕，而右边的三角形会按逆时针方向环绕。这不是巧合：因为按照我们选择的约定（即，我们计算三角形法线的方式），按顺时针方向环绕的三角形（相对于观察者）是朝前的，而按逆时针方向环绕的三角形（相对于观察者）是朝后的。

现在，3D 空间中的大部分物体都是封闭实心物体。当我们按照这一方式将每个三角形的法线指向物体外侧时，摄像机就不会看到实心物体朝后的三角形，因为朝前的三角形挡住了朝后的三角形；图 5.31 和图 5.32 分别以 2D 和 3D 形式说明了一概念。由于朝前的三角形挡住了朝后的三角形，所以绘制它们是毫无意义的。背面消隐（backface culling）是指让管线放弃对朝后的三角形的处理。这可以将所要处理的三角形的数量降低到原数量的一半。

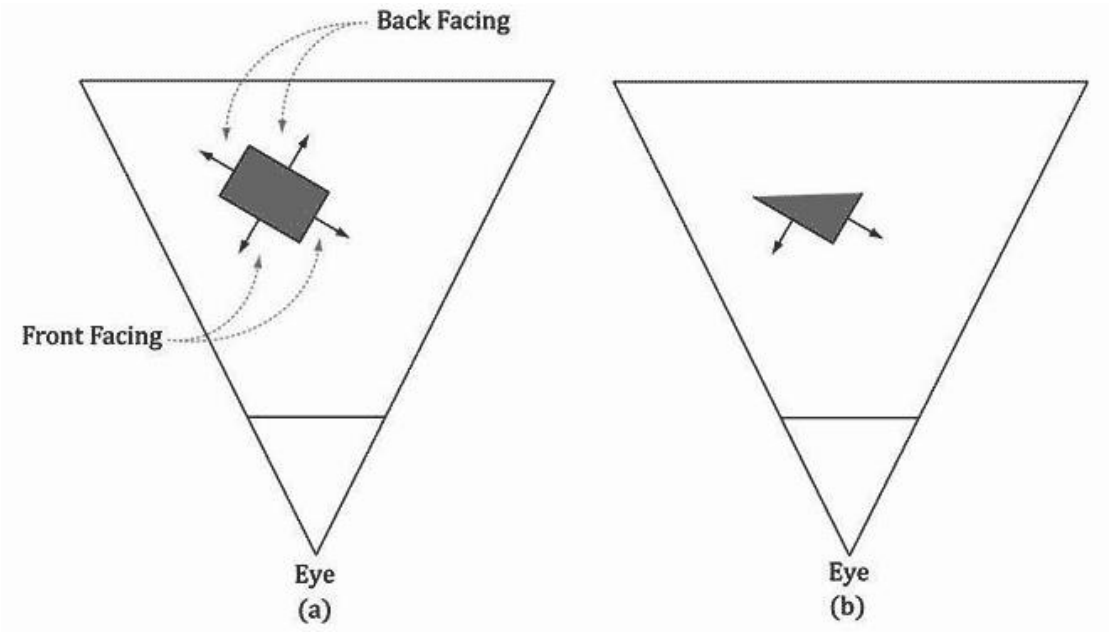



**图 5.31 (a)一个带有朝前和朝后三角形的实心物体。(b)在剔除了朝后的三角形之后的场景。注意，背面消隐不会影响最终的图像，因为朝后的三角形会被朝前的三角形阻挡。**

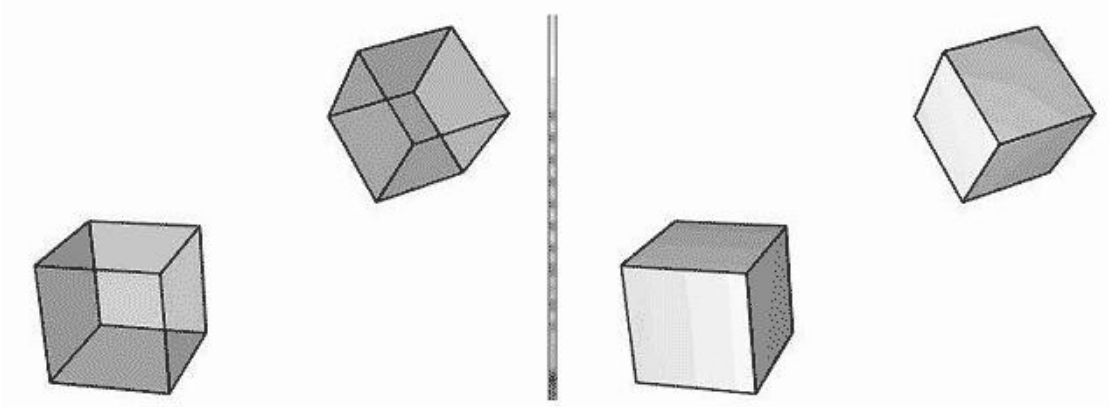


**图 5.32 （左图）当以透明方式绘制立方体时，我们可以看到所有的 6 个面。（右图）当以实心方式绘制立方体时，我们无法看到朝后的 3 个面，因为朝前的 3 个面挡住了它们——所以朝后的三角形可以被直接丢弃，不再接受后续处理，没人能看到些朝后的三角形。**

默认情况下，Direct3D 将（相对于观察者）顺时针方向环绕的三角形视为朝前的三角形，将（相对于观察者）逆时针方向环绕的三角形视为朝后的三角形。不过，这一约定可以通过修改 Direct3D 渲染状态颠倒过来。

## 5.10.3 顶点属性插值

如前所述，我们通过指定三角形的 3 个顶点来定义一个三角形。除位置外，顶点还可以包含其他属性，比如颜色、法线向量和纹理坐标。在视口变换之后，这些属性必须为三角形表面上的每个像素进行插值。顶点深度值也必须进行插值，以使每个像素都有一个可用于深度缓存算法的深度值。对屏幕空间中的顶点属性进行插值，其实就是对 3D 空间中的三角形表面进行线性插值（如图 5.33 所示）；这一工作需要借助所谓的透视矫正插值（perspective correct interpolation）来实现。本质上，三角形表面内部的像素颜色都是通过顶点插值得到的。

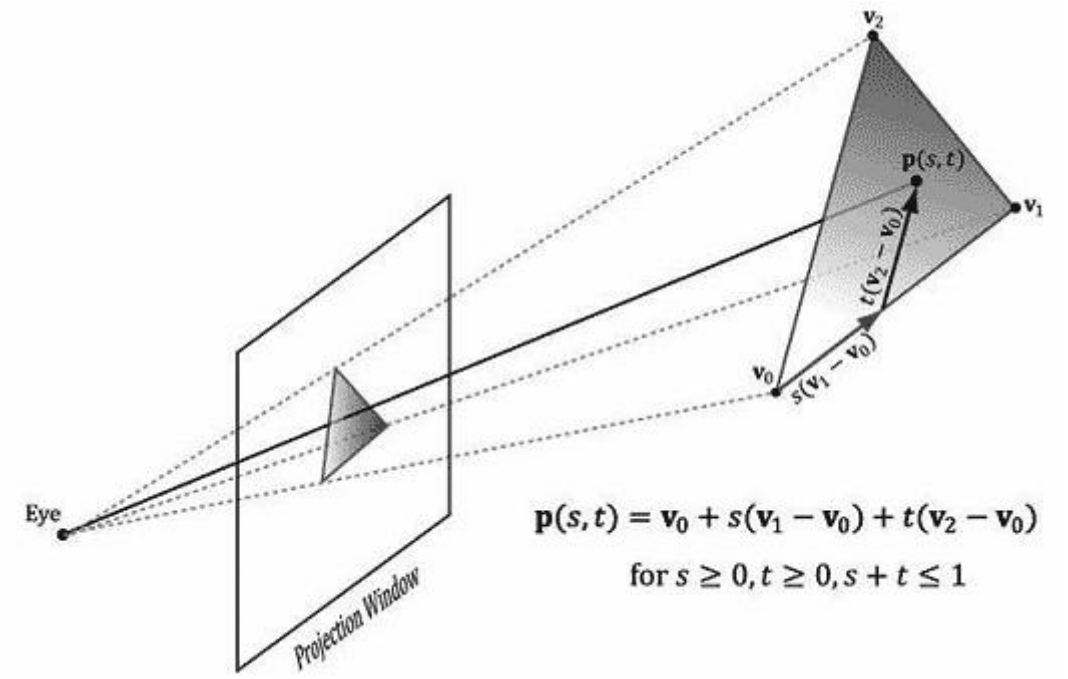



**图 5.33：通过对三角形顶点之间的属性值进行线性插值，可以得到三角形表面上的任一属性值 p(s,t)。**

我们不必关心透视精确插值的数学细节，因为硬件会自动完成这一工作；不过，有兴趣的读者可以在[Eberly01]中查阅相关的数学推导过程。图 5.34 介绍了一点基本思路：

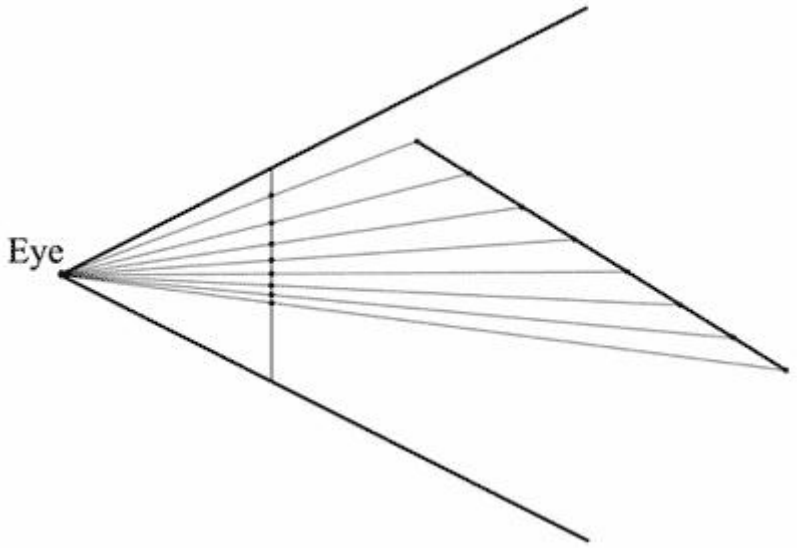



**图 5.34 一条 3D 线被投影到投影窗口上（在屏幕空间中投影是一条 2D 线）。我们看到，在 3D 线上取等距离的点，在 2D 屏幕空间上的投影点却不是等距离的。所以，我们在 3D 空间中执行线性插值，在屏幕空间需要执行非线性插值。**

# 5.11 像素着色器阶段

像素着色器（Pixel shader）是由我们编写的在 GPU 上执行的程序。像素着色器会处理每个像素片段（pixel fragment），它的输入是插值后的顶点属性，由此计算出一个颜色。像

素着色器可以非常简单地输出一个颜色，也可以很复杂，例如实现逐像素光照、反射和阴影等效果。

# 5.12 输出合并阶段

当像素片段由像素着色器生成之后，它们会被传送到渲染管线的输出合并（output merger，简称 OM）阶段。在该阶段中，某些像素片段会被丢弃（例如，未能通过深度测试或模板测试）。未丢弃的像素片段会被写入后台缓冲区。混合（blending）工作是在该阶段中完成的，一个像素可以与后台缓冲区中的当前像素进行混合，并以混合后的值作为该像素的最终颜色。某些特殊效果，比如透明度，就是通过混合来实现的；我们会在第 9 章专门讲解混合。

# 5.13 小结

**1.** 我们可以根据人眼的视觉特性，在 2D 图像上模拟 3D 场景。我们发现平行线会汇集为一个零点（或称消失点），物体的尺寸会随着深度的增加而减小，一个物体会挡住它后面的其他物体，灯光和阴影可以表现物体的立体感和体积感，阴影可以体现光源的位置并烘托物体之间的层次关系。

**2.** 我们使用三角形网络来模拟物体。我们可以通过指定三角形的 3 个顶点来定义一个三角形。在许多网格中，顶点是由多个三角形共享的；索引列表可以用于避免顶点重复。

**3.** 颜色可以通过红、绿、蓝的强度来描述。通过将这三种颜色以不同的强度进行混合，可以得到上千万种不同的颜色。我们通常使用规范化区间[0,1]描述红、绿、蓝的强度。0 表示没有强度，1 表示最高强度，中间值表示中等强度。通常在颜色中还会包含一个称为 alpha 分量的附加分量，它用于表示颜色的不透明度。当使用混合时，alpha 分量非常有用。我们可以使用 4D 向量$(r, g, b, a)$来表示带有 alpha 分量的颜色，其中 $0 \le r, g, b, a \le 1$。在 Direct3D 中，颜色由 **XMVECTOR** 类来表示，使用 XNA 数学库进行颜色操作可以发挥 SIMD 的优势。如果用 32 位表示颜色，则每个分量占一个字节，XNA 数学库提供了 **XMCOLOR** 结构用于存储 32 位颜色。颜色向量可以像普通向量那样进行加法、减法和标量乘法运算，只是每个分量的取值范围必须限定在[0, 1]区间内（或 32 位颜色的[0,255]区间内）。其他的向量运算（比如点积和叉积）对颜色向量来说没有意义。符号⊗表示分量乘法，它的含义为：$(c_1, c_2, c_3, c_4) \otimes (k_1, k_2, k_3, k_4) = (c_1k_1, c_2k_2, c_3k_3, c_4k_4)$。

**4.** 渲染管线是指：在给定一个 3D 场景的几何描述及一架已确定位置和方向的虚拟摄像机时，根据虚拟摄像机的视角生成 2D 图像的一系列步骤。

**5.** 渲染管线可以被分解为以下主要阶段：输入装配（IA）阶段；顶点着色器（VS）阶段；曲面细分阶段；几何着色器（GS）阶段；裁剪阶段、光栅化（RS）阶段、像素着色器（PS）阶段和输出合并器（OM）阶段。

# 6.1 顶点和顶点布局

5.5.1 节已经讲过，在 Direct3D 中，顶点由空间位置和各种附加属性组成，Direct3D 可以让我们灵活地建立属于我们自己的顶点格式；换句话说，它允许我们定义顶点的分量。要创建一个自定义的顶点格式，我们必须先创建一个包含顶点数据的结构体。例如，下面是两种不同类型的顶点格式：一个由位置和颜色组成，另一个由位置、法线和纹理坐标组成。

```
struct Vertex1
{
    XMFLOAT3 Pos;
    XMFLOAT4 Color;
};
struct Vertex2
{
    XMFLOAT3 Pos;
    XMFLOAT3 Normal;
    XMFLOAT2 Tex0;
    XMFLOAT2 Tex1;
};
```

在定义了顶点结构体之后，我们必须设法描述该顶点结构体的分量结构，使 Direct3D 知道该如何使用每个分量。这一描述信息是以输入布局（**ID3D11InputLayout**）的形式提供给 Direct3D 的。输入布局是一个 **D3D11_INPUT_ELEMENT_DESC** 数组。**D3D11_INPUT_ELEMENT_DESC** 数组中的每个元素描述了顶点结构体的一个分量。比如，当顶点结构体包含两个分量时，对应的 **D3D11_INPUT_ELEMENT_DESC** 数组会包含两个元素。我们将 **D3D11_INPUT_ELEMENT_DESC** 称为输入布局描述（input layout description）。**D3D11_INPUT_ELEMENT_DESC** 结构体定义如下：

```
typedef struct D3D11_INPUT_ELEMENT_DESC {
    LPCSTR SemanticName;
    UINT SemanticIndex;
    DXGI_FORMAT Format;
    UINT InputSlot;
    UINT AlignedByteOffset
    D3D11_INPUT_CLASSIFICATION InputSlotClass;
    UINT InstanceDataStepRate;
} D3D11_INPUT_ELEMENT_DESC;
```

**1．SemanticName**：一个与元素相关的字符串。它可以是任何有效的语义名。语义（semantic）用于将顶点结构体中的元素映射为顶点着色器参数（参见图 6.1）。

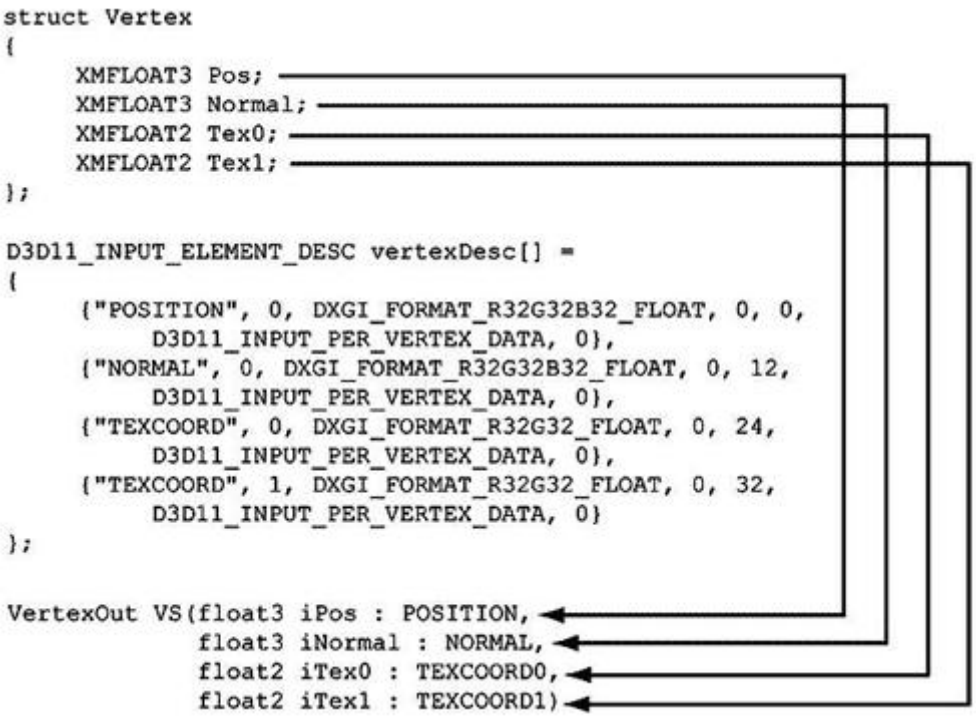



**图 6.1 顶点结构体中的每个元素分别由 D3D11_INPUT_ELEMENT_DESC 数组中的对应元素描述。语义名和语义索引提供了一种将顶点元素映射为顶点着色器参数的方法。**

**2．SemanticIndex**：附加在语义上的索引值。图 6.1 说明了使用该索引的原因；举例来说，当顶点结构体包含多组纹理坐标时，我们不是添加一个新的语义名，而是在语义名的后面加上一个索引值。在着色器代码中没有指定索引的语义默认索引为 0，例如，在图 6.1 中的 **POSITION** 相当于 **POSITION0**。

**3．Format**：一个用于指定元素格式的 DXGI_FORMAT 枚举类型成员；下面是一些常用的格式：

- DXGI_FORMAT_R32_FLOAT // 1D 32-bit float scalar
- DXGI_FORMAT_R32G32_FLOAT // 2D 32-bit float vector
- DXGI_FORMAT_R32G32B32_FLOAT // 3D 32-bit float vector
- DXGI_FORMAT_R32G32B32A32_FLOAT // 4D 32-bit float vector
- DXGI_FORMAT_R8_UINT // 1D 8-bit unsigned integer scalar
- DXGI_FORMAT_R16G16_SINT // 2D 16-bit signed integer vector
- DXGI_FORMAT_R32G32B32_UINT // 3D 32-bit unsigned integer vector
- DXGI_FORMAT_R8G8B8A8_SINT // 4D 8-bit signed integer vector
- DXGI_FORMAT_R8G8B8A8_UINT // 4D 8-bit unsigned integer vector

**4．InputSlot**：指定当前元素来自于哪个输入槽（input slot）。Direct3D 支持 16 个输入槽（索引依次为 0 到 15），通过这些输入槽我们可以向着色器传入顶点数据。例如，当一个顶点由位置元素和颜色元素组成时，我们既可以使用一个输入槽传送两种元素，也可以将两种元素分开，使用第一个输入槽传送顶点元素，使用第二个输入槽传送颜色元素。Direct3D 可以将来自于不同输入槽的元素重新组合为顶点。在本书中，我们只使用一个输入槽，但是在本章结尾的练习 2 中我们会引导读者做一个使用两个输入槽的练习。

**5．AlignedByteOffset**：对于单个输入槽来说，该参数表示从顶点结构体的起始位置到顶点元素的起始位置之间的字节偏移量。例如在下面的顶点结构体中，元素 **Pos** 的字节偏移量为 0，因为它的起始位置与顶点结构体的起始位置相同；元素 **Normal** 的字节偏移量为 12，

因为必须跳过由 **Pos** 占用的字节才能到达 **Normal** 的起始位置；元素 **Tex0** 的字节偏移量为 24，因为必须跳过由 **Pos** 和 **Normal** 占用的字节才能到达 **Tex0** 的起始位置；元素 **Tex1** 的字节偏移量为 32，因为必须跳过由 **Pos**，**Normal** 和 **Tex0** 占用的字节才能到达 **Tex1** 的起始位置。

```
struct Vertex2
{
    XMFLOAT3 Pos;       // 0-byte offset
    XMFLOAT3 Normal;    // 12-byte offset
    XMFLOAT2 Tex0;      // 24-byte offset
    XMFLOAT2 Tex1;      // 32-byte offset
};
```

**6．InputSlotClass**：目前指定为 **D3D11_INPUT_PER_VERTEX_DATA**；其他选项用于高级实例技术。

**7．InstanceDataStepRate**：目前指定为 0；其他值只用于高级实例技术。

对于前面的两个示例顶点结构体 Vertex1 和 Vertex2 来说，对应的输入布局描述为：

```
D3D11_INPUT_ELEMENT_DESC desc1[]=
{
    {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D11_INPUT_PER_VERTEX_DATA, 0},
    {"COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
        D3D11_INPUT_PER_VERTEX_DATA, 0}
};

D3D11_INPUT_ELEMENT_DESC desc2[]=
{
    {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D11_INPUT_PER_VERTEX_DATA, 0},
    {"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT,0, 12,
        D3D11_INPUT_PER_VERTEX_DATA, 0},
    {"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
        D3D11_INPUT_PER_VERTEX_DATA, 0}
    {"TEXCOORD", 1, DXGI_FORMAT_R32G32_FLOAT, 0, 32,
        D3D11_INPUT_PER_VERTEX_DATA, 0}
};
```

指定了输入布局描述之后，我们就可以使用 **ID3D11Device::CreateInputLayout** 方法获取一个表示输入布局的 **ID3D11InputLayout** 接口的指针：

```
HRESULT ID3D11Device::CreateInputLayout(
    const D3D11_INPUT_ELEMENT_DESC *pInputElementDescs,
    UINT NumElements,
    const void *pShaderBytecodeWithInputSignature,
    SIZE_T BytecodeLength,
    ID3D11InputLayout **ppInputLayout);
```

**1．pInputElementDescs**：一个用于描述顶点结构体的 D3D11_INPUT_ELEMENT_DESC 数组。

**2．NumElements：D3D11_INPUT_ELEMENT_DESC** 数组的元素数量。

**3．pShaderBytecodeWithInputSignature：**指向顶点着色器参数的字节码的指针。

**4．BytecodeLength：**顶点着色器参数的字节码长度，单位为字节。

**5．ppInputLayout：**返回创建后的 **ID3D11InputLayout** 指针。

我们需要进一步解释一下第 3 个参数的含义。本质上，顶点着色器以一组顶点元素作为它的输入参数——也就是所谓的输入签名（input signature）。自定义顶点结构体中的元素必须被映射为与它们对应的顶点着色器参数，图 6.1 解释了这一问题。通过在创建输入布局时传入顶点着色器签名，Direct3D 在创建时就可以验证输入布局是否与输入签名匹配，并建立从顶点结构体到着色器参数之间的映射关系。一个 **ID3D11InputLayout** 对象可以在多个参数完全相同的着色器中重复使用。

假设有下列输入参数和顶点结构：

```
VertexOut VS(float3 Pos:POSITION, float4 Color:COLOR,
    float3 Normal: NORMAL){ }
struct Vertex
{
    XMFLOAT3  Pos ;
    XMFLOAT4  Color;
};
```

这样会产生错误，VC++的调试输出窗口会显示以下信息：

```
D3D11:ERROR:ID3D11Device::CreateInputLayout:The provided input
signature expects to read an element with SemanticName/Index:
'NORMAL'/0, but the declaration doesn't provide a matching name.
```

假如顶点结构和输入参数与输入元素匹配，但类型不同：

```
VertexOut VS(int3 Pos:POSITION, float4 Color:COLOR) { }
struct Vertex
{
    XMFLOAT3  Pos;
    XMFLOAT4  Color;
} ;
```

这样做是可行的，因为 Direct3D 允许输入寄存器中的字节被重新解释。但是，VC ++调试输出窗口会显示以下信息：

```
D3D11:WARNING:ID3D11Device::CreateInputLayout:The provided input
signature expects to read an element with SemanticName/Index:
'POSITION'/0 and component(s) of the type 'int32'. However,the
matching  entry in the InputLayout declaration, element[0],
specifies mismatched format:'R32G32B32_FLOAT'.This is not an error,
since behavior is well defined :The element format determines what
data conversion algorithm gets applied before it shows up in a
shader register.Independently,the shader input signature defines
how the shader will interpret the data that has been placed in its
input registers,with no change in the bits stored.It is valid for
the application to reinterpret data as a different type once it is
in the vertex shader,so this warning is issued just in case reinte
```

```
rpretation  was not intended by the author.
```

下面的代码说明了该如何调用 **ID3D11Device::CreateInputLayout** 方法。注意，这些代码涉及了一些我们还未讨论的内容（比如 **ID3D11Effect**）。本质上，一个 effect 可以封装一个或多个 pass，而每个 pass 都会与一个顶点着色器相连。所以，我们可以从 effect 中得到有关 pass 的描述信息（**D3D11_PASS_DESC**），然后再从中得到顶点着色器的输入签名。

```
ID3D11Effect* mFX;
ID3D11EffectTechnique* mTech;
ID3D11InputLayout* mVertexLayout;
/* ...create the effect... */
mTech = mFX->GetTechniqueByName("Tech");
D3D11_PASS_DESC PassDesc;
mTech->GetPassByIndex(0)->GetDesc(&PassDesc);
HR(md3dDevice->CreateInputLayout(vertexDesc, 4,
   PassDesc.pIAInputSignature, PassDesc.IAInputSignatureSize,
   &mVertexLayout));
```

创建了输入布局对象之后，它不会自动绑定到设备上。我们必须调用下面的语句来实现绑定：

```
ID3D11InputLayout* mVertexLayout;
/* ...create the input layout... */
md3dImmediateContext->IASetInputLayout(mVertexLayout);
```

如果你打算用一个输入布局来绘制一些物体，然后再使用另一个的布局来绘制另一些物体，那你必须按照下面的形式来组织代码：

```
md3dImmediateContext->IASetInputLayout(mVertexLayout1);
/* ...draw objects using input layout 1... */
md3dImmediateContext->IASetInputLayout(mVertexLayout2);
/* ...draw objects using input layout 2... */
```

换句话说，当一个 **ID3D11InputLayout** 对象被绑定到设备上时，如果不去改变它，那么它会始终驻留在那里。

# 6.2 顶点缓冲

为了让 GPU 访问顶点数组，我们必须把它放置在一个称为缓冲（buffer）的特殊资源容器中，该容器由 **ID3D11Buffer** 接口表示。

用于存储顶点的缓冲区称为顶点缓冲（vertex buffer）。Direct3D 缓冲不仅可以存储数据，而且还说明了如何访问数据以及数据被绑定到图形管线的那个阶段。要创建一个顶点缓冲，我们必须执行以下步骤：

1．填写一个 **D3D11_BUFFER_DESC** 结构体，描述我们所要创建的缓冲区。

2．填写一个 **D3D11_SUBRESOURCE_DATA** 结构体，为缓冲区指定初始化数据。

3．调用 **ID3D11Device::CreateBuffer** 方法来创建缓冲区。

**D3D11_BUFFER_DESC** 结构体的定义如下：

```
typedef struct D3D11_BUFFER_DESC{
   UINT ByteWidth;
```

```
    D3D11_USAGE Usage;
    UINT BindFlags;
    UINT CPUAccessFlags;
    UINT MiscFlags;
    UINT StructureByteStride;
} D3D11_BUFFER_DESC;
```

**1．ByteWidth**：我们将要创建的顶点缓冲区的大小，单位为字节。

**2．Usage**：一个用于指定缓冲区用途的 **D3D11_USAGE** 枚举类型成员。有 4 个可选值：

（a）**D3D10_USAGE_DEFAULT**：表示 GPU 会对资源执行读写操作。在使用映射 API（例如 **ID3D11DeviceContext::Map**）时，CPU 在使用映射 API 时不能读写这种资源，但它能使用 **ID3D11DeviceContext::UpdateSubresource**。**ID3D11DeviceContext::Map** 方法会在 6.14 节中介绍。

（b）**D3D11_USAGE_IMMUTABLE**：表示在创建资源后，资源中的内容不会改变。这样可以获得一些内部优化，因为 GPU 会以只读方式访问这种资源。除了在创建资源时 CPU 会写入初始化数据外，其他任何时候 CPU 都不会对这种资源执行任何读写操作，我们也无法映射或更新一个 immutable 资源。

（c）**D3D11_USAGE_DYNAMIC**：表示应用程序（CPU）会频繁更新资源中的数据内容（例如，每帧更新一次）。GPU 可以从这种资源中读取数据，使用映射 API（**ID3D11DeviceContext::Map**）时，CPU 可以向这种资源中写入数据。因为新的数据要从 CPU 内存（即系统 RAM）传送到 GPU 内存（即显存），所以从 CPU 动态地更新 GPU 资源会有性能损失；若非必须，请勿使用 **D3D11_USAGE_DYNAMIC**。

（d）**D3D11_USAGE_STAGING**：表示应用程序（CPU）会读取该资源的一个副本（即，该资源支持从显存到系统内存的数据复制操作）。显存到系统内存的复制是一个缓慢的操作，应尽量避免。使用 **ID3D11DeviceContext::CopyResource** 和 **ID3D11DeviceContext::CopySubresourceRegion** 方法可以复制资源，在 12.3.5 节会介绍一个复制资源的例子。

**3．BindFlags**：对于顶点缓冲区，该参数应设为 **D3D11_BIND_VERTEX_BUFFER**。

**4．CPUAccessFlags**：指定 CPU 对资源的访问权限。设置为 0 则表示 CPU 无需读写缓冲。如果 CPU 需要向资源写入数据，则应指定 **D3D11_CPU_ACCESS_WRITE**。具有写访问权限的资源的 Usage 参数应设为 **D3D11_USAGE_DYNAMIC** 或 **D3D11_USAGE_STAGING**。如果 CPU 需要从资源读取数据，则应指定 **D3D11_CPU_ACCESS_READ**。具有读访问权限的资源的 Usage 参数应设为 **D3D11_USAGE_STAGING**。当指定这些标志值时，应按需而定。通常，CPU 从 Direct3D 资源读取数据的速度较慢。CPU 向资源写入数据的速度虽然较快，但是把内存副本传回显存的过程仍很耗时。所以，最好的做法是（如果可能的话）不指定任何标志值，让资源驻留在显存中，只用 GPU 来读写数据。

**5．MiscFlags**：我们不需要为顶点缓冲区指定任何杂项（miscellaneous）标志值，所以该参数设为 0。有关 **D3D11_RESOURCE_MISC_FLAG** 枚举类型的详情请参阅 SDK 文档。

**6．StructureByteStride**：存储在结构化缓冲中的一个元素的大小，以字节为单位。这个属性只用于结构化缓冲，其他缓冲可以设置为 0。所谓结构化缓冲，是指存储其中的元素大小都相等的缓冲。

**D3D11_SUBRESOURCE_DATA** 结构体的定义如下：

```
typedef struct D3D11_SUBRESOURCE_DATA {
    const void *pSysMem;
```

```
    UINT SysMemPitch;
    UINT SysMemSlicePitch;
} D3D11_SUBRESOURCE_DATA;
```

**1. pSysMem**：包含初始化数据的系统内存数组的指针。当缓冲区可以存储 n 个顶点时，对应的初始化数组也应至少包含 n 个顶点，从而使整个缓冲区得到初始化。

**2. SysMemPitch**：顶点缓冲区不使用该参数。

**3. SysMemSlicePitch**：顶点缓冲区不使用该参数。

下面的代码创建了一个只读的顶点缓冲区，并以中心在原点上的立方体的 8 顶点来初始化该缓冲区。之所以说该缓冲区是只读的，是因为当立方体创建后相关的几何数据从不改变——始终保持为一个立方体。另外，我们为每个顶点指定了不同的颜色；这些颜色将用于立方体着色，我们会在本章随后的小节中对此进行讲解。

```
//  定义在 d3dUtil.h 中的 Colors 命名空间
//
// #define XMGLOBALCONST extern CONST __declspec(selectany)
//   1. extern so there is only one copy of the variable, and not a separate
//      private copy in each .obj.
//   2. __declspec(selectany) so that the compiler does not complain about
//      multiple definitions in a .cpp file (it can pick anyone and discard
//      the rest because they are constant--all the same).

namespace Colors
{
    XMGLOBALCONST XMVECTORF32 White     = {1.0f, 1.0f, 1.0f, 1.0f};
    XMGLOBALCONST XMVECTORF32 Black     = {0.0f, 0.0f, 0.0f, 1.0f};
    XMGLOBALCONST XMVECTORF32 Red       = {1.0f, 0.0f, 0.0f, 1.0f};
    XMGLOBALCONST XMVECTORF32 Green     = {0.0f, 1.0f, 0.0f, 1.0f};
    XMGLOBALCONST XMVECTORF32 Blue      = {0.0f, 0.0f, 1.0f, 1.0f};
    XMGLOBALCONST XMVECTORF32 Yellow    = {1.0f, 1.0f, 0.0f, 1.0f};
    XMGLOBALCONST XMVECTORF32 Cyan      = {0.0f, 1.0f, 1.0f, 1.0f};
    XMGLOBALCONST XMVECTORF32 Magenta   = {1.0f, 0.0f, 1.0f, 1.0f};

    XMGLOBALCONST XMVECTORF32 Silver    = {0.75f, 0.75f, 0.75f, 1.0f};
    XMGLOBALCONST XMVECTORF32 LightSteelBlue = {0.69f, 0.77f, 0.87f, 1.0f};
}

// 创建顶点缓冲
Vertex vertices[] =
{
    {       XMFLOAT3(-1.0f,       -1.0f,       -1.0f),       (const
float*)&Colors::White   },
    {       XMFLOAT3(-1.0f,       +1.0f,       -1.0f),       (const
float*)&Colors::Black   },
    { XMFLOAT3(+1.0f, +1.0f, -1.0f), (const float*)&Colors::Red     },
    {       XMFLOAT3(+1.0f,       -1.0f,       -1.0f),       (const
```

```
float*)&Colors::Green   },
    { XMFLOAT3(-1.0f, -1.0f, +1.0f), (const float*)&Colors::Blue    },
    {       XMFLOAT3(-1.0f,       +1.0f,       +1.0f),       (const
float*)&Colors::Yellow  },
    { XMFLOAT3(+1.0f, +1.0f, +1.0f), (const float*)&Colors::Cyan    },
    { XMFLOAT3(+1.0f, -1.0f, +1.0f), (const float*)&Colors::Magenta }
};

D3D11_BUFFER_DESC vbd;
vbd.Usage = D3D11_USAGE_IMMUTABLE;
vbd.ByteWidth = sizeof(Vertex) * 8;
vbd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
vbd.CPUAccessFlags = 0;
vbd.MiscFlags = 0;
vbd.StructureByteStride = 0;
D3D11_SUBRESOURCE_DATA vinitData;
vinitData.pSysMem = vertices;

ID3D11Buffer * mVB;
HR(md3dDevice->CreateBuffer(&vbd, &vinitData, &mVB));
```

Vertex 类型和颜色由以下结构定义：

```
struct Vertex
{
    XMFLOAT3 Pos;
    XMFLOAT4 Color;
};
```

在创建顶点缓冲区后， 我们必须把它绑定到设备的输入槽上，只有这样才能将顶点送入管线。这一工作使用如下方法完成：

```
void ID3D11DeviceContext::IASetVertexBuffers(
    UINT StartSlot,
    UINT NumBuffers,
    ID3D10Buffer *const *ppVertexBuffers,
    const UINT *pStrides,
    const UINT *pOffsets);
```

**1．StartSlot**：顶点缓冲区所要绑定的起始输入槽。一共有 16 个输入槽，索引依次为 0 到 15。

**2．NumBuffers**：顶点缓冲区所要绑定的输入槽的数量，如果起始输入槽为索引 k，我们绑定了 n 个缓冲，那么缓冲将绑定在索引为 $I_k$，$I_{k+1}$……$I_{k+n-1}$ 的输入槽上。

**3．ppVertexBuffers**：指向顶点缓冲区数组的第一个元素的指针。

**4．pStrides**：指向步长数组的第一个元素的指针（该数组的每个元素对应一个顶点缓冲区，也就是，第 i 个步长对应于第 i 个顶点缓冲区）。这个步长是指顶点缓冲区中的元素的字节长度。

**5．pOffsets**：指向偏移数组的第一个元素的指针（该数组的每个元素对应一个顶点缓冲区，也就是，第 i 个偏移量对应于第 i 个顶点缓冲区）。这个偏移量是指从顶点缓冲区的

起始位置开始，到输入装配阶段将要开始读取数据的位置之间的字节长度。当希望跳过顶点缓冲区前面的一部分数据时，可以使用该参数。

因为 **IASetVertexBuffers** 方法支持将一个顶点缓冲数组设置到不同的输入槽中，因此这个方法看起来有点复杂。但是，大多数情况下我们只使用一个输入槽。本章最后的练习部分你会遇到使用两个输入插槽的情况。

顶点缓冲区会一直绑定在输入槽上时。如果不改变输入槽的绑定对象，那么当前的顶点缓冲区会一直驻留在那里。所以，当使用多个顶点缓冲区时，你可以按照下面的形式组织代码：

```
ID3D11Buffer* mVB1; // stores vertices of type Vertex1
ID3D11Buffer* mVB2; // stores vertices of type Vertex2
/*...Create the vertex buffers...*/
UINT stride = sizeof(Vertex1);
UINT offset = 0;
md3dImmediateContext->IASetVertexBuffers(0,  1,  &mVB1,  &stride,
&offset);
/* ...draw objects using vertex buffer 1... */
stride = sizeof(Vertex2);
offset = 0;
md3dImmediateContext->IASetVertexBuffers(0,  1,  &mVB2,  &stride,
&offset);
/* ...draw objects using vertex buffer 2... */
```

把顶点缓冲区指定给输入槽并不能实现顶点的绘制；它只是绘制前的准备工作（准备把顶点传送到管线）。顶点的实际绘制工作由 **ID3D11DeviceContext::Draw** 方法完成：

```
void     ID3D11DeviceContext::Draw(UINT     VertexCount,     UINT
StartVertexLocation);
```

这两个参数定义了在顶点缓冲区中所要绘制的顶点的范围，如图 6.2 所示。

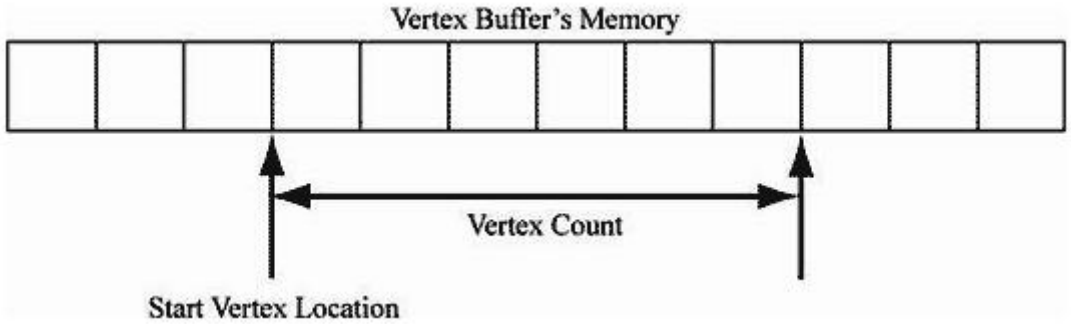



**图 6.2 StartVertexLocation 指定了在顶点缓冲区中所要绘制的第一个顶点的索引（从 0 开始）。VertexCount 指定了所要绘制的顶点的数量。**

# 6.3 索引和索引缓冲

由于索引要由 GPU 访问，所以它们必须放在一个特定的资源容器中，该容器称为索引缓冲（index buffer）。创建索引缓冲的过程与创建顶点缓冲的过程非常相似，只不过索引缓

冲存储的是索引而非顶点。所以，这里不再赘述之前讨论过的内容，我们直接给出一个创建索引缓冲区的示例：

```
UINT indices [24] = {
    0, 1, 2,   //  Triangle  0
    0, 2, 3,   //  Triangle  1
    0, 3, 4,   //  Triangle  2
    0, 4, 5,   //  Triangle  3
    0, 5, 6,   //  Triangle  4
    0, 6, 7,   //  Triangle  5
    0, 7, 8,   //  Triangle  6
    0, 8, 1    //  Triangle  7
} ;

// 要创建的索引的描述
D3D11_BUFFER_DESC ibd;
ibd.Usage = D3D11_USAGE_IMMUTABLE;
ibd.ByteWidth = sizeof(UINT) * 24;
ibd.BindFlags = D3D11_BIND_INDEX_BUFFER;
ibd.CPUAccessFlags = 0;
ibd.MiscFlags = 0;
ibd.StructureByteStride = 0;

// 设定用于初始化索引缓冲的数据
D3D11_SUBRESOURCE_DATA iinitData;
iinitData.pSysMem = indices;

// 创建索引缓冲
ID3D11Buffer* mIB;
HR(md3dDevice->CreateBuffer(&ibd, &iinitData, &mIB));
```

与顶点缓冲区相同，所有的 Direct3D 资源在使用之前都必须先绑定到管线上。我们使用 **ID3D11DeviceContext::IASetIndexBuffer** 方法将一个索引缓冲区绑定到输入装配阶段。下面是一个例子：

```
md3dImmediateContext->IASetIndexBuffer(mIB,  DXGI_FORMAT_R32_UINT,
0);
```

第 2 个参数表示索引格式。在本例中，我们使用的是 32 位无符号整数（DWORD）；所以，该参数设为 DXGI_FORMAT_R32_UINT。如果你希望节约一些内存，不需要这大的取值范围，那么可以改用 16 位无符号整数。还要注意的是，在 **IASetIndexBuffer** 方法中指定的格式必须与 **D3D11_BUFFER_DESC::ByteWidth** 数据成员指定的字节长度一致，否则会出现问题。索引缓冲区只支持 **DXGI_FORMAT_R16_UINT** 和 **DXGI_FORMAT_R32_UINT** 两种格式。第 3 个参数是一个偏移值，它表示从索引缓冲区的起始位置开始、到输入装配时实际读取数据的位置之间的字节长度。如果希望跳过索引缓冲区前面的一部分数据，那么可以使用该参数。

最后，当使用索引时，我们必须用 **DrawIndexed** 方法代替 **Draw** 方法：

```
void ID3D11DeviceContext::DrawIndexed(
```

```
    UINT IndexCount,
    UINT StartIndexLocation,
    INT BaseVertexLocation);
```

**1．IndexCount**：在当前绘图操作中使用的索引的数量。在一次绘图操作中不一定使用索引缓冲区中的全部索引；也就是说，我们可以绘制索引的一个连续子集。

**2．StartIndexLocation**：指定从索引缓冲区的哪个位置开始读取索引数据。

**3．BaseVertexLocation**：在绘图调用中与索引相加的一个整数。

我们通过分析如下情景来解释些参数。假设有三个物体：一个球体、一个立方体和一个圆柱体。每个物体都有它自己的顶点缓冲和索引缓冲。在每个独立的索引缓冲中的索引都与各自独立的顶点缓冲对应。现在，我们把这三个物体的顶点和索引数据合并到一个全局的顶点缓冲和一个全局的索引缓冲中，如图 6.3 所示（假如有许多小顶点缓冲和索引缓冲，并且它们可以很容易的合并，那么合并它们可以带来性能提升）。在合并之后，索引就不再正确了（因为这些索引是与各自独立的顶点缓冲区对应的，它们与全局顶点缓冲区没有对应关系）；所以，这些索引必须被重新计算，使它们与全局顶点缓冲区建立正确的对应关系。

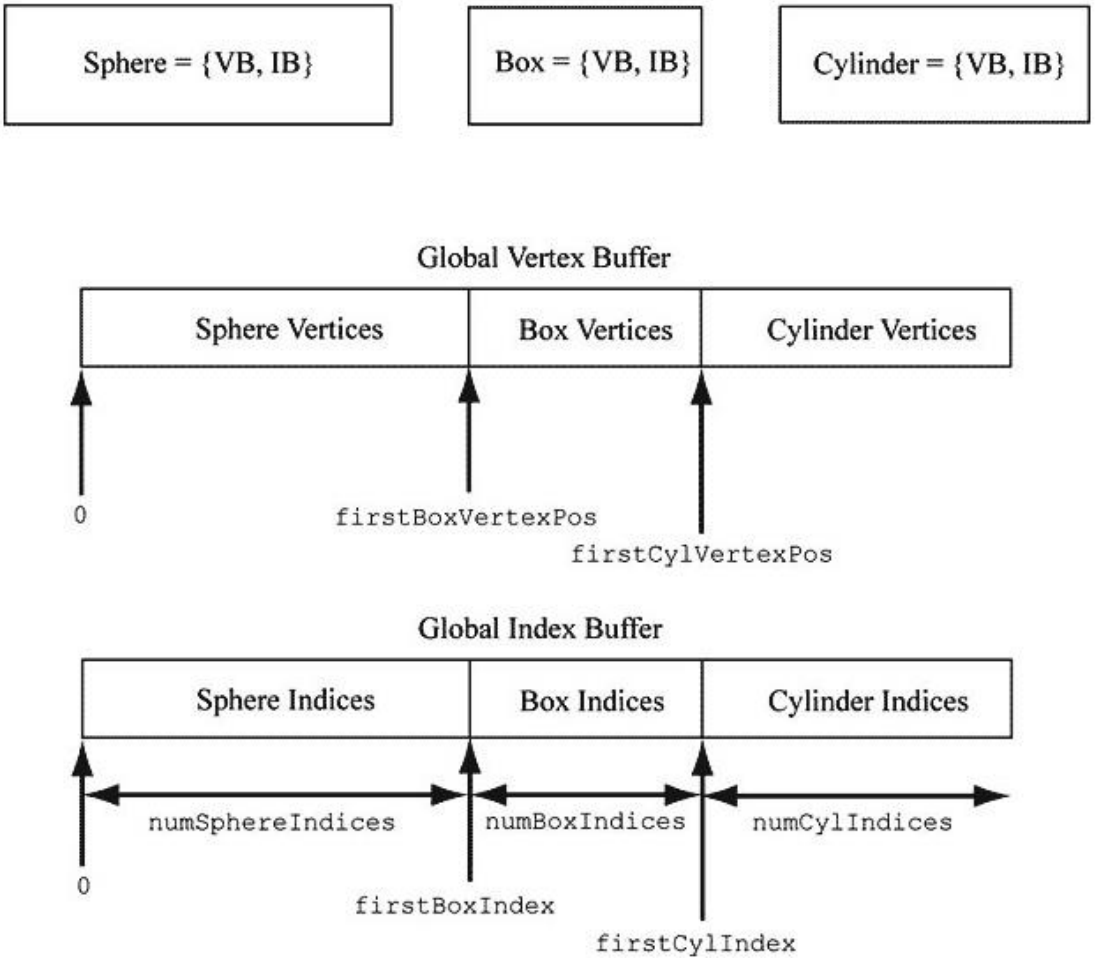



**图 6.3 将多个顶点缓冲合并为一个大的顶点缓冲，将多个索引缓冲合并为一个大的索引缓冲。**

假设原先的立方体索引为 0、1、...、numBoxVertices-1，通过个索引可以遍历立方体的所有顶点。在合并之后，索引应该变为 firstBoxVertexPos、firstBoxVertexPos+1、...、firstBoxVertexPos+ numBoxVertices -1。所以，要更新索引，我们必须将 firstBoxVertexPos 与

立方体的每个索引相加。而且，我们必须将 firstCylVertexPos 与圆柱体的每个索引相加。注意，球体的索引不需要修改（因为球体的顶点位置为 0）。通常，只要将一个物体在全局顶点缓冲区中的第一个顶点位置与原索引相加就可以得到该物体的新索引值。所以，只要给出物体在全局顶点缓冲中的第一个顶点的位置（例如，BaseVertexLocation），Direct3D 就可以在绘图调用中为我们重新计算索引。我们可以使用以下 3 条语句依次绘制球体、立方体和圆柱体：

```
md3dImmediateContext->DrawIndexed(numSphereIndices, 0, 0);
md3dImmediateContex->DrawIndexed(numBoxIndices,     firstBoxIndex,
firstBoxVertexPos);
md3dImmediateContex->DrawIndexed(numCylIndices,     firstCylIndex,
firstCylVertexPos);
```

后面的“Shape”示例程序就用到了这个技术。

# 6.4 顶点着色器示例

下面是一个顶点着色器的示例，它的代码非常简单：

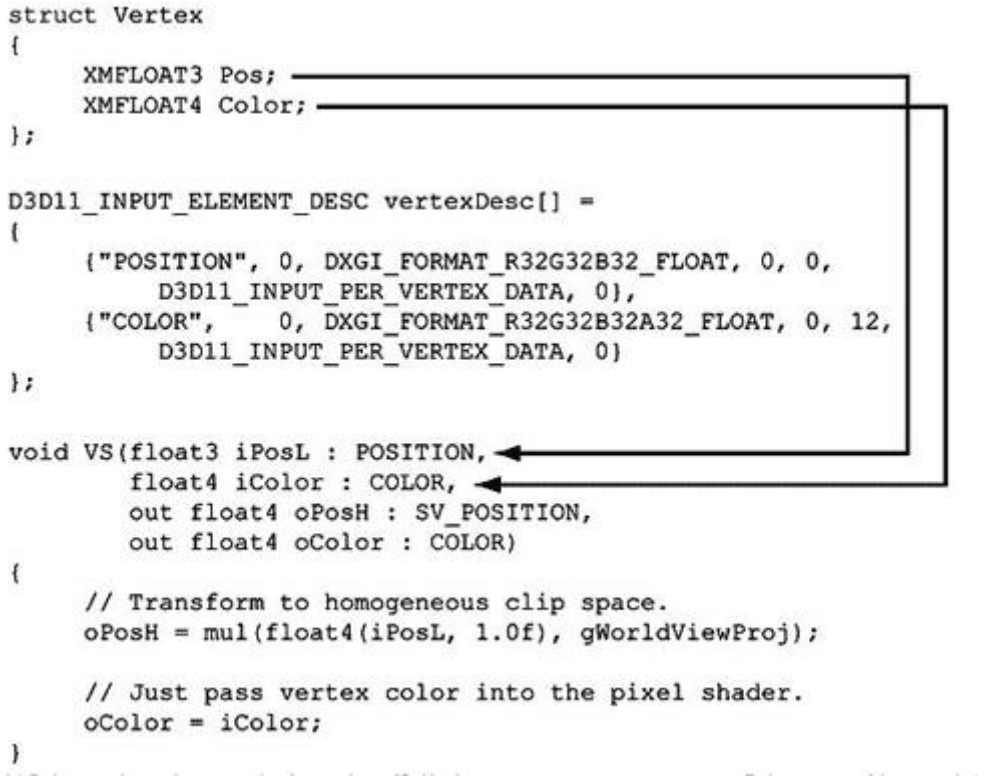
```
cbuffer cbPerObject
{
    float4x4 gWVP;
};

void VS(float3 iPosL : POSITION,
    float4 iColor : COLOR,
    out float4 oPosH : SV_POSITION,
    out float4 oColor : COLOR)
{
    // 转换到齐次裁剪空间
    oPosH = mul(float4(iPosL, 1.0f), gWVP);
    // 把顶点颜色直接传到像素着色器
    oColor = iColor;
}
```

着色器使用一种称为高级着色语言（High-Level Shading Language，简称 HLSL）的脚本语言来编写，它的语法与 C++相似，很容易就能学会。附录 B 提供了一些有关 HLSL 的简要概述。在本书中，我们将采用一种基于示例的方式讲解 HLSL 及着色器编程。也就是，根据贯穿本书的每个演示程序所涉及的技术讲解相关的 HLSL 概念。着色器通常保存在一种称为 effect 文件（.fx）的纯文本文件中。我们会在本章随后的小节中讨论 effect 文件，而现在我们主要讨论顶点着色器。

这里，顶点着色器是一个称为 VS 的函数。注意，你可以为顶点着色器指定任何有效的函数名。该顶点着色器包含 4 个参数；前两个是输入参数，后两个是输出参数（由 **out** 关键字表示）。HLSL 没有类似于 C++的引用和指针，所以当一个函数要返回多个值时，我们必须使用结构体或输出参数。

前两个输入参数对应于我们在顶点结构体中定义的数据成员。参数语义“**:POSITION**”

和“**:COLOR**”用于将顶点结构体的数据成员映射为顶点着色器的输入参数，如图 6.4 所示。



```
struct Vertex
{
    XMFLOAT3 Pos;
    XMFLOAT4 Color;
};

D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
{
    {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D11_INPUT_PER_VERTEX_DATA, 0},
    {"COLOR",    0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
        D3D11_INPUT_PER_VERTEX_DATA, 0}
};

void VS(float3 iPosL : POSITION,
        float4 iColor : COLOR,
        out float4 oPosH : SV_POSITION,
        out float4 oColor : COLOR)
{
    // Transform to homogeneous clip space.
    oPosH = mul(float4(iPosL, 1.0f), gWorldViewProj);

    // Just pass vertex color into the pixel shader.
    oColor = iColor;
}
```

**图 6.4 D3D11_INPUT_ELEMENT_DESC 数组为每个顶点元素指定了一个相关的语义，而顶点着色器的每个参数也都带有一个附加语义。语义用于建立顶点元素和顶点着色器参数之间的对应关系。**

输出参数也带有附加语义（“**:SV_POSITION**”和“**:COLOR**”）。这些语义用于将顶点着色器的输出数据映射为下一阶段（几何着色器或像素着色器）的输入数据。注意，**SV_POSITION** 是一个特殊的语义（SV 表示系统值，即 system value 的缩写）。它用于告诉顶点着色器该元素存储的是顶点位置。顶点位置的处理方式与其他顶点属性不同，因为它涉及到一些其他属性所没有的特殊运算，比如裁剪。若不是系统值，那么输出参数的语义名称可以是任何有效的语义名称。

该顶点着色器的代码非常简单。第一行通过与一个 4×4 矩阵 **gWorldViewProj** 相乘，将顶点位置从局部空间变换到齐次裁剪空间，矩阵 **gWorldViewProj** 是世界矩阵、观察矩阵和投影矩阵的组合矩阵：

```
// 转换到齐次裁剪空间
oPosH = mul(float4(iPosL, 1.0f), gWorldViewProj);
```

构造函数语法“float4(iPosL, 1.0f)”用于创建 4D 向量，它相当于“float4(iPosL.x, iPosL.y, iPosL.z, 1.0f)”；我们知道，顶点位置是一个点而不是一个向量，所以第 4 个分量应设为 1（即 w=1）。**float2** 和 **float3** 分别表示 2D 和 3D 向量。矩阵变量 **gWorldViewProj** 定义在一个常量缓冲区中，我们会在下一节讨论对它进行讨论。内置函数 **mul** 用于实现向量-矩阵乘法，它为不同维数的矩阵乘法定义了多个重载版本；例如，该函数可以实现 4×4 矩阵乘法、3×3 矩阵乘法、或者 1×3 向量与 3×3 矩阵的向量-矩阵乘法。最后一行是将输入的颜色赋值给输出参数，把颜色传递给管线的下一阶段：

```
oColor = iColor;
```

我们可以使用结构体来重写上面的顶点着色器，实现相同的功能：

```
cbuffer cbPerObject
```

```
{
    float4x4 gWVP;
};
struct VS_IN
{
    float3 posL    : POSITION;
    float4 color : COLOR;
};
struct VS_OUT
{
    float4 posH : SV_POSITION;
    float4 color : COLOR;
};
VS_OUT VS(VS_IN input)
{
    VS_OUT output;
    output.posH = mul(float4(input.posL, 1.0f), gWVP);
    output.color = input.color;
    return output;
}
```

**注意**：当没有几何着色器时，顶点着色器至少要实现投影变换，因为当顶点离开顶点着色器时（在没有几何着色器的情况下），硬件假定顶点位于投影空间。当包含一个几何着色器时，投影工作可以转嫁到几何着色器中完成。

**注意**：顶点着色器（或几何着色器）不执行透视除法；它只完成投影矩阵部分。透视除法会随后由硬件完成。

# 6.5 常量缓冲

在上一节的顶点着色器示例中包含如下代码：

```
cbuffer cbPerObject
{
    float4x4 gWorldViewProj;
};
```

这段代码定义了一个称为 cbPerObject 的 **cbuffer** 对象（constant buffer，常量缓冲）。常量缓冲只是一个用于存储各种变量的数据块，这些变量可以由着色器来访问。在本例中，常量缓冲区只存储了一个称为 **gWorldViewProj** 的 4×4 矩阵，它是世界矩阵、观察矩阵和投影矩阵的组合矩阵，用于将顶点从局部空间变换到齐次裁剪空间。在 HLSL 中，4×4 矩阵由内置的 **float4x4** 类型表示；与之类似，要定义一个 3×4 矩阵和一个 2×2 矩阵，可以分别使用 **float3x4** 和 **float2x2** 类型。顶点着色器不能修改常量缓冲中的数据，但是通过 effect 框架（6.9 节），C++应用程序代码可以在运行时修改常量缓冲中的内容。它为 C++应用程序代码和 effect 代码提供了一种有效的通信方式。例如，因为每个物体的世界矩阵各不相同，所以每个物体的“WVP”组合矩阵也各不相同；所以，当使用上述顶点着色器绘制多个物体

时，我们必须在绘制每个物体前修改 **gWorldViewProj** 变量。

通常的建议是根据变量修改的频繁程度创建不同的常量缓冲。比如，你可以创建下面的常量缓冲：

```
cbuffer cbPerObject
{
    float4x4 gWVP;
};
cbuffer cbPerFrame
{
    float3 gLightDirection;
    float3 gLightPosition;
    float4 gLightColor;
};
cbuffer cbRarely
{
    float4 gFogColor;
    float gFogStart;
    float gFogEnd;
};
```

在本例中，我们使用了 3 个常量缓冲区。第 1 个常量缓冲区存储“WVP”组合矩阵。该变量随物体而定，所以它必须在物体级别上更新。也就是，当我们每帧渲染 100 个物体时，每帧都要对这个变量更新 100 次。第 2 个常量缓冲存储了场景中的灯光变量。这里，我们假设要生成灯光动画，所以些变量必须在每帧中更新一次。最后一个常量缓冲存储了用于控制雾效的变量。这里，我们假设场景的雾效变化频率很低（例如，在游戏的一个特定时段中变化一次）。

对常量缓冲进行分组是为了提高运行效率。当一个常量缓冲区被更新时，它里面的所有变量都会同时更新；所以，根据它们的更新频率进行分组，可以减少不必要的更新操作，提高运行效率。

# 6.6 像素着色器示例

5.10.3 节说过，由顶点着色器（或几何着色器）输出的顶点属性都已经过了插值处理。这些插值随后会作为像素着色器（pixel shader）的输入数据传入像素着色器。假设这里没有几何着色器，图 6.5 说明了目前顶点数据的流动过程。

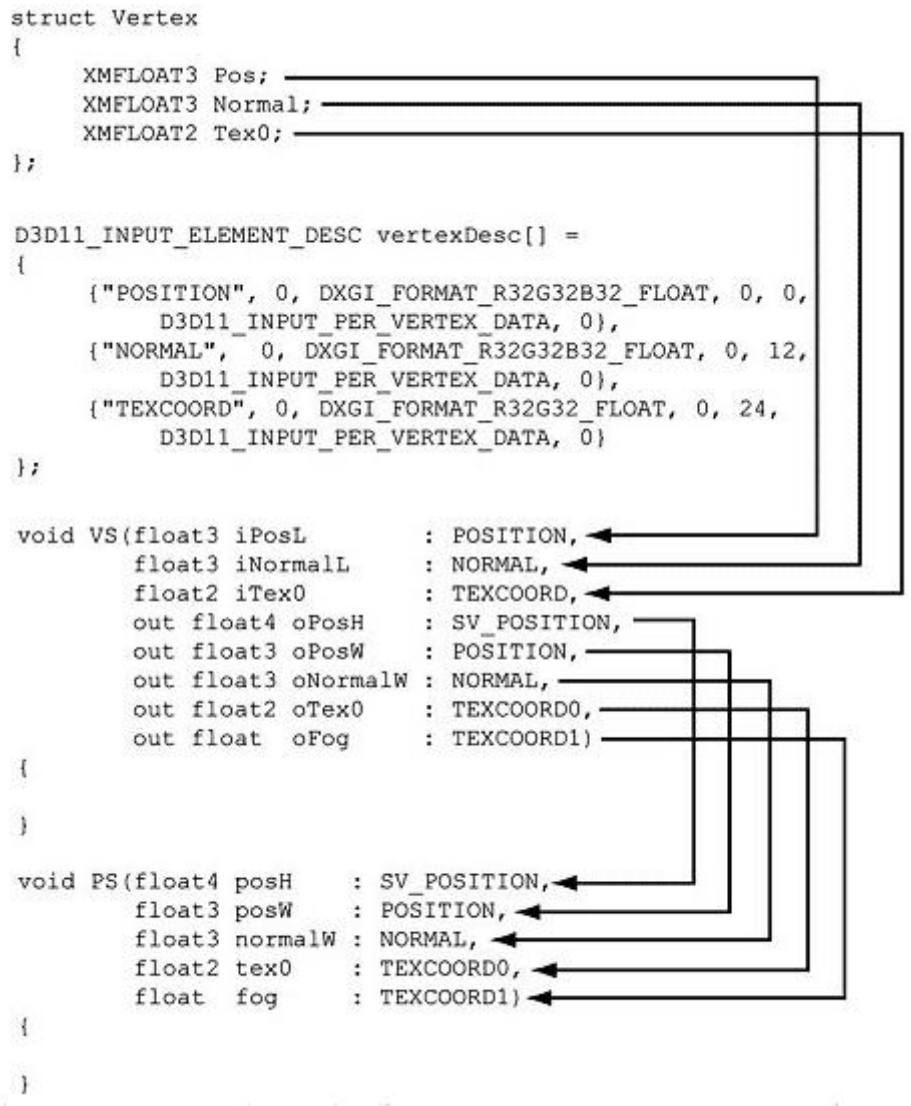



**图 6.5 D3D11_INPUT_ELEMENT_DESC 数组为每个顶点元素指定了一个关联语义，而顶点着色器的每个参数都有一个附加语义。这些语义描述了顶点元素和顶点着色器参数之间的对应关系。同样，顶点着色器的每个输出参数和像素着色器的每个输入参数也都有一个附加语义。这些语义用于将顶点着色器的输出参数映射为像素着色器的输入参数。**

与顶点着色器相似，像素着色器也是一个函数，只不过它处理的数据是像素片段（pixel fragment）。像素着色器的任务是为每个像素片段计算一个颜色值。请注意，像素和像素片段的含义不同，像素片段可能不会被存入后台缓冲区；例如，像素着色器可以对像素片段进行裁剪（在 HLSL 中的一个 **clip** 函数，它可以终止对一个像素片段的处理工作），或者当一个像素片段没有通过深度测试或模板测试时，它会被丢弃。所以，后台缓冲区中的一个像素可能会对应多个候选像素片段；这就是我们要区分“像素片段”和“像素”这两个术语的原因，虽然有时这两个术语会交替使用，但是读者应该明白它们在特定环境下的特定含义。

下面是一个简单的像素着色器，它与 6.4 节给出的顶点着色器对应。出于完整性的考虑，我们将该顶点着色器再次列了出来。

```
cbuffer cbPerObject
{
    float4x4 gWorldViewProj;
};
```

```
void VS(float3 iPosL : POSITION, float4 iColor : COLOR,
    out float4 oPosH    : SV_POSITION,
    out float4 oColor : COLOR)
{
    // 转换到齐次剪裁空间
    oPosH = mul(float4(iPosL, 1.0f), gWVP);
    // 将顶点颜色直接传递到像素着色器
    oColor = iColor;
}

float4 PS(float4 posH: SV_POSITION,float4 color : COLOR) : SV_TARGET
{
    return color;
}
```

在本例中，像素着色器只是简单地返回插值颜色。注意，像素着色器的输入参数和顶点着色器的输出参数是对应的；这是一项规定。该像素着色器返回了一个 4D 颜色值。函数参数列表中的 **SV_TARGET** 语义表示返回值与渲染目标视图的格式一致。

我们也可以使用输入/输出结构体的形式重写上面的顶点着色器和像素着色器代码。我们将语义附加到输入/输出结构的成员上，使用 return 语句用于输出代替输出参数。

```
cbuffer cbPerObject
{
    float4x4 gWorldViewProj;
};

struct VertexIn
{
    float3 PosL  : POSITION;
    float4 Color : COLOR;
};

struct VertexOut
{
    float4 PosH  : SV_POSITION;
    float4 Color : COLOR;
};

VertexOut VS(VertexIn vin)
{
    VertexOut vout;

    // 转换到齐次剪裁空间
    vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj);

    // 将顶点颜色直接传递到像素着色器
```

```
    vout.Color = vin.Color;

    return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
    return pin.Color;
}
```

# 6.7 渲染状态

从本质上讲，Direct3D 是一个状态机（state machine）。在我们改变它的状态之前，驻留在状态机内的当前状态是不会改变的。例如，我们在 6.1 节、6.2 节和 6.3 节中看到，当顶点缓冲和索引缓冲绑定到管线的输入装配阶段时，如果我们不绑定其他缓冲，那么它们就会一直驻留在那里；同样，在没有改变图元拓扑之前，当前的图元拓扑设置会一直有效。另外，Direct3D 将配置信息封装在状态组中，我们可以使用如下 3 种状态组配置 Direct3D：

**1．ID3D11RasterizerState**：该接口表示用于配置管线光栅化阶段的状态组。

**2．ID3D11BlendState**：该接口表示用于配置混合操作的状态组。我们将在有关混合的章节讨论这些状态；默认情况下，混合处于禁用状态，所以我们可以先不考虑这方面的问题。

**3．ID3D11DepthStencilState**：该接口表示用于配置深度测试和模板测试的状态组。我们将在有关模板缓冲的章节讨论这些状态；默认情况下，模板是禁用的，所以我们可以先不考虑这方面的问题。而默认的深度测试是我们在 4.1.5 节描述的标准深度测试。

目前，我们唯一需要关心的状态块接口是 **ID3D11RasterizerState**。我们可以通过填充一个 **D3D11_RASTERIZER_DESC** 结构体并调用如下方法来创建 **ID3D11RasterizerState** 对象：

```
HRESULT ID3D11Device::CreateRasterizerState(
    const D3D11_RASTERIZER_DESC *pRasterizerDesc,
    ID3D11RasterizerState **ppRasterizerState);
```

第 1 个参数是一个指向 **D3D11_RASTERIZER_DESC** 结构体的指针，该结构体用于描述所要创建的光栅化状态块；第二个参数用于返回创建后的 **ID3D11RasterizerState** 对象。

**D3D11_RASTERIZER_DESC** 结构体的定义如下：

```
typedef struct D3D11_RASTERIZER_DESC{
    D3D11_FILL_MODE FillMode;     // Default:D3D11_FILL_SOLID
    D3D11_CULL_MODE CullMode;     // Default:D3D11_CULL_BACK
    BOOL FrontCounterClockwise;  // Default:false
    INT DepthBias;   // Default:0
    FLOAT DepthBiasClamp;   // Default:0.0f
    FLOAT SlopeScaledDepthBias;   // Default:0.0f
    BOOL DepthClipEnable;   // Default:true
    BOOL ScissorEnable;   // Default:false
    BOOL MultisampleEnable;  // Default:false
```

```
    BOOL AntialiasedLineEnable;  // Default:false
} D3D11_RASTERIZER_DESC;
```

这里面的大部分成员是高级选项或者不常用的选项；因此，我们在这里只讲解前 3 个成员的含义，其他成员的详情请参见 SDK 文档。

**1．FillMode**：当指定为 **D3D11_FILL_WIREFRAME** 时，表示以线框模式渲染几何体；当指定为 **D3D11_FILL_SOLID** 时，表示以实心模式渲染几何体，这是默认值。

**2．CullMode**：当指定为 **D3D11_CULL_NONE** 时，表示禁用背面消隐功能；当指定为 D3D11_CULL_FRONT 时，表示消隐朝前的三角形；当指定为 **D3D11_CULL_BACK** 时，表示消隐朝后的三角形，这是默认值。

**3．FrontCounterClockwise**：当设为 **false** 时，表示按顺时针方向环绕的三角形（相对于观察者）是朝前的，而按逆时针方向环绕的三角形（相对于观察者）是朝后的，这是默认值。当设为 **true** 时，表示按逆时针方向环绕的三角形（相对于观察者）是朝前的，而按顺时针方向环绕的三角形（相对于观察者）是朝后的。

在创建 **ID3D11RasterizerState** 对象之后，我们可以使用个新的状态块来更新设备：

```
void         ID3D11DeviceContext::RSSetState(ID3D11RasterizerState
*pRasterizerState);
```

下面的代码示范了如何通过创建一个光栅化状态块来禁用背面消隐：

```
D3D11_RASTERIZER_DESC rsDesc;
ZeroMemory(&rsDesc, sizeof(D3D11_RASTERIZER_DESC));
rsDesc.FillMode = D3D11_FILL_SOLID;
rsDesc.CullMode = D3D11_CULL_NONE;
rsDesc.FrontCounterClockwise = false;
rsDesc.DepthClipEnable = true;

HR(md3dDevice->CreateRasterizerState(&rsDesc,&mNoCullRS));
```

**注意**：因为没有设置的属性的默认值是 0 或 false，所以使用 ZeroMemory 可以正常初始化这些属性。但是，若有些属性默认值不是 0 或是 true，那么你就必须显式地设置这些值。

注意，对于一个应用程序来说，你可能会用到多个不同的 **ID3D11RasterizerState** 对象。所以，你应该在初始化时把它们都创建出来，然后在应用程序的更新/绘图代码中切换些状态。例如，场景中有两个物体，你希望先以线框模式绘制第一个物体，然后再以实心模式绘制第二个物体。那么，你就应该创建两个 **ID3D11RasterizerState** 对象，当绘制物体时，切换这两种不同的状态：

```
// Create render state objects at initialization time.
ID3D11RasterizerState* mWireframeRS;
ID3D11RasterizerState* mSolidRS;
...
// Switch between the render state objects inthe draw function.
md3dDeviceContext->RSSetState(mSolidRS);
DrawObject();
md3dDeviceContext->RSSetState(mWireframeRS);
DrawObject();
```

注意，Direct3D 不会从一种状态自动恢复到先前状态。所以，当绘制物体时，你应该根据需要手工指定状态对象。错误地假设设备的当前状态必然会导致错误的渲染结果。

每个状态块都有一个默认状态。我们可以通过在调用 **RSSetState** 方法时指定空值来恢

复默认状态：

```
md3dDeviceContext->RSSetState( 0 );
```

**注意**：应用程序无需在运行时创建额外的渲染状态组。所以，应该在初始化时就定义并创建所有需要用到的状态组。而且，因为无需在运行时修改状态组，你可以在渲染代码中对这些状态组提供全局只读访问。例如，你可以将所有状态组对象放置在一个静态类中，通过这个方法，你就无需创建重复的状态组，渲染代码的不同部分都能共享这个渲染状态组对象。

# 6.8 Effects

effect 框架是一组用于管理着色器程序和渲染状态的工具代码。例如，你可能会使用不同的 effect 绘制水、云、金属物体和动画角色。每个 effect 至少要由一个顶点着色器、一个像素着色器和渲染状态组成。

在 Direct3D 11 中，effects 框架已从 D3DX 库中移除，你必须包含一个单独的头文件（**d3dx11Effect.h**），链接一个单独的库文件（**D3DX11Effects.lib** 用于 release 生成，而 **D3DX11EffectsD.lib** 用于 debug 生成）。

而且，在 Direct3D 11 中提供了 effect 库的完整源代码（DirectX SDK\Samples\C++\Effects11）。因此，你可以根据需要修改 effect 框架。本书中，我们只是使用、并不会修改 effect 框架。要使用这个库，首先需要生成 Effects11 项目的 Release 和 Debug 模式，用于获得 **D3DX11Effects.lib** 和 **D3DX11EffectsD.lib** 文件，除非 effect 框架进行了更新（例如，新版本的 DirectX SDK 可能会更新这些文件，这时就需要重新生成.lib 文件），这个步骤只需进行一次。**d3dx11Effect.h** 头文件可在 DirectX SDK\Samples\C++\Effects11\Inc 文件夹中找到。在示例代码中，我们将 **d3dx11Effect.h**，**D3DX11EffectsD.lib** 和 **D3DX11Effects.lib** 文件都放在 Common 文件夹中，这样所有的项目文件都能共享这些文件。

## 6.8.1 Effect 文件

我们已经讨论了顶点着色器、像素着色器，并对几何着色器、曲面细分着色器进行了简要概述。我们还讨论了常量缓冲，它可以用于存储由着色器访问的“全局”变量。这些代码通常保存在一个 effect 文件(.fx)中，它是一个纯文本文件中(就像是 C++代码保存在.h 和.cpp 文件中一样)。除了着色器和常量缓冲之外，每个 effect 文件至少还要包含一个 technique，而每个 technique 至少要包含一个 pass。

**1．technique11**：一个 technique 由一个或多个 pass 组成，用于创建一个渲染技术。每个 pass 实现一种不同的几何体渲染方式，按照某些方式将多个 pass 的渲染结果混合在一起就可以得到我们最终想要的渲染结果。例如，在地形渲染中我们将使用多通道纹理映射技术（multi-pass texturing technique）。注意，多通道技术通常会占用大量的系统资源，因为每个 pass 都要对几何体进行一次渲染；不过，要实现某些渲染效果，我们必须使用多通道技术。

**2．pass**：一个 pass 由一个顶点着色器、一个可选的几何着色器、一个像素着色器和一些渲染状态组成。这些部分定义了 pass 的几何体渲染方式。像素着色器也是可选的（很罕见）。例如，若我们只想绘制深度缓冲，不想绘制后台缓冲，在这种情况下我们就不需要像素着色器计算像素的颜色。

**注意**：techniques 也可以组合在一起成为 effect 组。如果你没有显式地定义一个 effect 组，那么编译器会创建一个匿名 effect 组，把所有 technique 包含在 effect 文件中。本书中，

我们不显式地定义 effect 组。

下面是本章演示程序使用的 effect 文件：

```
cbuffer cbPerObject
{
    float4x4 gWorldViewProj;
};

struct VertexIn
{
    float3 PosL  : POSITION;
    float4 Color : COLOR;
};

struct VertexOut
{
    float4 PosH  : SV_POSITION;
    float4 Color : COLOR;
};

VertexOut VS(VertexIn vin)
{
    VertexOut vout;

    // 转换到齐次剪裁空间
    vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj);

    // 将顶点颜色直接传递到像素着色器
    vout.Color = vin.Color;

    return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
    return pin.Color;
}

technique11 ColorTech
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_5_0, VS() ) );
        SetPixelShader( CompileShader( ps_5_0, PS() ) );
    }
```

```
}
```

**注意**：点和向量可以在许多不同的空间中描述（例如，局部空间、世界空间、观察空间、齐次裁剪空间）。当阅读代码时，有时很难看出点和向量的坐标系是相对于哪个坐标系的。所以，我们经常使用下面的后缀来表示空间：L（局部空间）、W（世界空间）、V（观察空间）、H（齐次裁剪空间）。下面是一些例子：

```
float3 iPosL;      // local space
float3 gEyePosW;    // world space
float3 normalV;    // view space
float4 posH;       // homogeneous clip space
```

前面提到，pass 可以包含渲染状态。也就是，状态块可以直接在 effect 文件中创建和指定。当 effect 需要特定的渲染状态时，这种方式非常实用；但是，当某些 effect 需要在运行过程中改变渲染状态时，我们更倾向于在应用程序层执行状态设定，因为这样进行状态切换更方便一些。下面的代码示范了如何在一个 effect 文件中创建和指定光栅化状态块。

```
RasterizerState Wireframe
{
    FillMode = Wireframe;
    CullMode = Back;
    FrontCounterClockwise = false;
    // 我们没有设置的属性使用默认值
};

technique11 ColorTech
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_5_0, VS() ) );
        SetPixelShader( CompileShader( ps_5_0, PS() ) );
        SetRasterizerState(Wireframe);
    }
}
```

可以看到，在光栅化状态对象中定义的常量与 C++中的枚举成员基本相同，只是省去了前缀而已（例如，**D3D11_FILL_**和 **D3D11_CULL_**）。

注意：由于 effect 通常保存在扩展名为.fx 的文件中，所以在修改 effect 代码之后，不必重新编译 C++源代码。

## 6.8.2 编译着色器

创建一个 effect 的第一步是编译定义在.fx 文件中的着色器程序，可以由下面的 D3DX 方法完成：

```
HRESULT D3DX11CompileFromFile (
    LPCTSTR pSrcFile ,
    CONST D3D10_SHADE R_MACRO *pDefines,
    LPD3D10INCLUDE  pInclude ,
```

```
    LPCSTR  pFunctionName ,
    LPCSTR pProfile,
    UINT Flags 1,
    UINT Flags 2,
    ID3DX11ThreadPump *pPump ,
    ID3D10Blob  **ppShader,
    ID3D10Blob  **ppErrorMsgs,
    HRESULT *pHResult);
```

**1．pSrcFile**：.fx 文件名，该文件包含了我们所要编译的效果源代码。

**2．pDefines**：高级选项，我们不使用；请参阅 SDK 文档。

**3．pInclude**：高级选项，我们不使用；请参阅 SDK 文档。

**4．pFunctionName**：着色器入口函数的名字。只用于单独编译着色器程序的情况。当使用 effect 框架时设置为 null，这是因为在 effect 文件中已经定义了入口点。

**5．pProfile**：用于指定着色器版本的字符串。对于 Direct3D 11 来说，我们使用的着色器版本为 5.0（“fx_5_0”）。

**6．Flags1**：用于指定着色器代码编译方式的标志值。SDK 文档列出了很多标志值，但本书只使用其中的 2 个：

- **D3D10_SHADER_DEBUG**：以调试模式编译着色器。
- **D3D10_SHADER_SKIP_OPTIMIZATION**：告诉编译器不做优化处理（用于进行调试）。

**7．Flags2**：高级选项，我们不使用；请参阅 SDK 文档。

**8．pPump**：指向线程泵的指针，多线程编程时使用，是高级选项，我们不使用；请参阅 SDK 文档。本书中这个值都设为 null。

**9．ppShader**：返回一个指向 **ID3D10Blob** 数据对象的指针，这个数据对象保存了经过编译的代码。

**10．ppErrorMsgs**：返回一个指向 **ID3D10Blob** 数据对象的指针，这个数据对象存储了一个包含错误信息的字符串。

**11．pHResult**：在使用异步编译时，用于获得返回的错误代码。仅当使用 pPump 时才使用该参数；我们在本书中将该参数设为空值。

**注意**：1．除了可以编译在.fx 文件内的着色器代码，这个方法也可以编译单独的着色器代码。有些程序不使用 effect 框架，它们会单独的定义和编译自己的着色器代码。

2. 方法中的指向“D3D10”的引用并不是打印错误。因为 D3D11 编译器是建立在 D3D10 的编译器之上的，所以 Direct3D 11 开发组就没有修改某些标识的名称。

3．**ID3D10Blob** 只是一个通用内存块，它有两个方法：

（a）LPVOID GetBufferPointer：返回指向数据的一个 void*，所以在使用时应该对它执行相应的类型转换（具有请参见下面的示例）。

（b）SIZE_T GetBufferSize：返回缓冲的大小，以字节为单位。

编译完成后，我们就可以使用下面的方法创建一个 effect（用 **ID3DXEffect11** 接口表示）：

```
HRESULT D3DX11CreateEffectFromMemory(
    void *pData,
    SIZE_T DataLength,
    UINT FXFlags ,
    ID3D11Device *pDevice,
```

```
    ID3DX11Effect **ppEffect);
```

**1．pData**：指向编译好的 effect 数据的指针。

**2．DataLength**：effect 数据的长度，以字节为单位。

**3. FXFlags**：Effect 标识必须与定义在 **D3DX11CompileFromFile** 方法中的 **Flags2** 匹配。

**4．pDevice**：指向 Direct3D 11 设备的指针。

**5．ppEffect**：指向创建好的 effect 的指针。

下面的代码演示了如何编译并创建一个 effect：

```
DWORD shaderFlags = 0;
#ifdefined (DEBUG)||defined(_DEBUG)
    shaderFlags |= D3D10_SHADER_DEBUG;
    shaderFlags |= D3D10_SHADER_SKIP_OPTIMIZATION ;
#endif
ID3D10Blob * compiledShader = 0;
ID3D10Blob * compilationMsgs = 0;
HRESULT hr = D3DX11CompileFromFile(L"color.fx", 0,
    0, 0, "fx_5_0", shaderFlags,
    0, 0, &compiledShader, &compilationMsgs, 0);

//  compilationMsgs 包含错误或警告的信息
if(compilationMsgs ! =  0)
{
    MessageBoxA(0,  (char*)compilationMsgs->GetBufferPointer(),  0,
0);
    ReleaseCOM(compilationMsgs);
}

// 就算没有 compilationMsgs，也需要确保没有其他错误
if(FAILED(hr))
{
    DXTrace(__FILE__,(DWORD)__LINE__,hr,L"D3DX11Compile
FromFile",true);
}

ID3DX11Effect*  mFX;
HR(D3DX11CreateEffectFromMemory(
    compiledShader->Ge tBufferPointer(),
    compiledShader->Ge tBufferSize(),
    0, md3dDevice, &mFX));

// 编译完成释放资源
ReleaseCOM(compiledShader);
```

**注意**：创建 Direct3D 资源代价昂贵，尽量在初始化阶段完成，即创建输入布局、缓冲、渲染状态对象和 effect 应该总在初始化阶段完成。

## 6.8.3 在 C++应用程序中与 Effect 进行交互

C++应用程序代码通常要与 effect 进行交互；尤其是 C++应用程序经常要更新常量缓冲中的变量。例如，在一个 effect 文件中，我们有如下常量缓冲定义：

```
cbuffer cbPerObject
{
    float4x4 gWVP;
    float4 gColor;
    float gSize;
    int gIndex;
    bool gOptionOn;
};
```

通过 **ID3D11Effect** 接口，我们可以获得指向常量缓冲变量的指针：

```
ID3D11EffectMatrixVariable* fxWVPVar;
ID3D11EffectVectorVariable* fxColorVar;
ID3D11EffectScalarVariable* fxSizeVar;
ID3D11EffectScalarVariable* fxIndexVar;
ID3D11EffectScalarVariable* fxOptionOnVar;
fxWVPVar     = mFX->GetVariableByName("gWVP")->AsMatrix();
fxColorVar   = mFX->GetVariableByName("gColor")->AsVector();
fxSizeVar    = mFX->GetVariableByName("gSize")->AsScalar();
fxIndexVar   = mFX->GetVariableByName("gIndex")->AsScalar();
fxOptionOnVar = mFX->GetVariableByName("gOptionOn")->AsScalar();
```

**ID3D11Effect::GetVariableByName** 方法返回一个 **ID3D11EffectVariable** 指针。它是一种通用 effect 变量类型；要获得指向特定类型变量的指针（例如，矩阵、向量、标量），你必须使用相应的 As-方法（例如，**AsMatrix**、**AsVector**、**AsScalar**）。

一旦我们获得变量指针，我们就可以通过 C++接口来更新它们了。下面是一些例子：

```
fxWVPVar->SetMatrix( (float*)&M ); // assume M is of type XMMATRIX
fxColorVar->SetFloatVector( (float*)&v ); // assume v is of type
XMVECTOR
fxSizeVar->>SetFloat( 5.0f );
fxIndexVar->SetInt( 77 );
fxOptionOnVar->SetBool( true );
```

注意，这些语句修改的只是 effect 对象在系统内存中的一个副本，它并没有传送到 GPU 内存中。所以在执行绘图操作时，我们必须使用 **Apply** 方法更新 GPU 内存（参见 6.8.4 节）。这样做的原因是为了提高效率，避免频繁地更新 GPU 内存。如果每修改一个变量就要更新一次 GPU 内存，那么效率会很低。

注意：effect 变量不一定要被类型化。例如，可以有如下代码：

```
ID3D11EffectVariable* mfxEyePosVar;
mfxEyePosVar = mFX->GetVariableByName("gEyePosW");
...
mfxEyePosVar->SetRawValue(&mEyePos, 0, sizeof(XMFLOAT3));
```

这种方式可以用来设置任意大小的变量（例如，普通结构体）。注意，

**ID3D11EffectVectorVariable** 接口使用 4D 向量。如果你希望使用 3D 向量的话，那应该像上面那样使用 **ID3D11EffectVariable** 接口。

除了常量缓冲变量之外，我们还需要获得指向 technique 对象的指针。实现方法如下：

```
ID3D11EffectTechnique* mTech;
mTech = mFX->GetTechniqueByName("ColorTech");
```

该方法只包含一个用于指定 technique 名称的字符串参数。

## 6.8.4 使用 effect 绘图

要使用 technique 来绘制几何体，我们只需要确保对常量缓冲中的变量进行实时更新。然后，使用循环语句来遍历 technique 中的每个 pass，使用 pass 来绘制几何体：

```
// 设置常量缓冲
XMMATRIX world  = XMLoadFloat4x4(&mWorld);
XMMATRIX view = XMLoadFloat4x4(&mView);
XMMATRIX proj = XMLoadFloat4x4(&mProj);
XMMATRIX worldViewProj = world*view*proj;

mfxWorldViewProj->SetMatrix(reinterpret_cast<float*>(&worldViewPr
oj));

D3DX11_TECHNIQUE_DESC techDesc;
mTech->GetDesc(&techDesc);
for(UINT p = 0;p < techDesc.Passes;++p )
{
    mTech->GetPassByIndex(p)->Apply(0,md3dImmediateContext);
    //  绘制几何体
    md3dImmediateContext->DrawIndexed(36, 0, 0);
}
```

当使用 pass 来绘制几何体时，Direct3D 会启用在 pass 中指定的着色器和渲染状态。**ID3D11EffectTechnique::GetPassByIndex** 方法返回一个指定索引的 pass 对象的 **ID3D11EffectPass** 接口指针。**Apply** 方法更新存储在 GPU 内存中的常量缓冲、将着色器程序绑定到管线、并启用在 pass 中指定的各种渲染状态。在当前版本的 Direct3D 11 中，**ID3D11EffectPass::Apply** 方法的第一个参数还未使用，应设置为 0；第二个参数指向 pass 使用的设备上下文的指针。

如果你需要在绘图调用之间改变常量缓冲中的变量值，那你必须在绘制几何体之前调用 **Apply** 方法：

```
for(UINT i = 0; i < techDesc.Passes; ++i)
{
    ID3D11EffectPass* pass = mTech->GetPassByIndex(i);

    //设置地面几何体的 WVP 组合矩阵
    worldViewProj = mLandWorld*mView*mProj;
    mfxWorldViewProj->SetMatrix(reinterpret_cast<float*>(&
```

```
worldViewProj);
    pass->Apply(0, md3dImmediateContext);
    mLand.draw();

    // 设置水波几何体的 WVP 组合矩阵
    worldViewProj = mWavesWorld*mView*mProj;
    mfxWorldViewProj->SetMatrix(reinterpret_cast<float*>(&
worldViewProj);
    pass->Apply(0 ,md3dImmediateContext);
    mWaves.draw();
}
```

## 6.8.5 在生成期间编译 effect

我们已经介绍了如何在运行时通过 **D3DX11CompileFromFile** 方法编译一个 effect。但这样做会带来一个小小的不便：如果你的 effect 文件有一个编译错误，直到程序运行时你才会发现这个错误。我们还可以使用 DirectX SDK 自带的 *fxc* 工具（位于 DirectX SDK\Utilities\bin\x86）离线编译你的 effect。而且，你还可以修改你的 VC++项目，将调用 fxc 编译 effect 的过程作为生成过程的一部分。步骤如下：

**1**．确保路径 DirectX SDK\Utilities\bin\x86 位于你的项目的 VC++目录的“可执行文件目录（Executable Directories）”之下。

**2**．在项目中添加 effect 文件。

**3**．在解决方案资源管理器中右击每个 effect 文件选择**属性**，添加自定义生成工具（见图 6.6）：

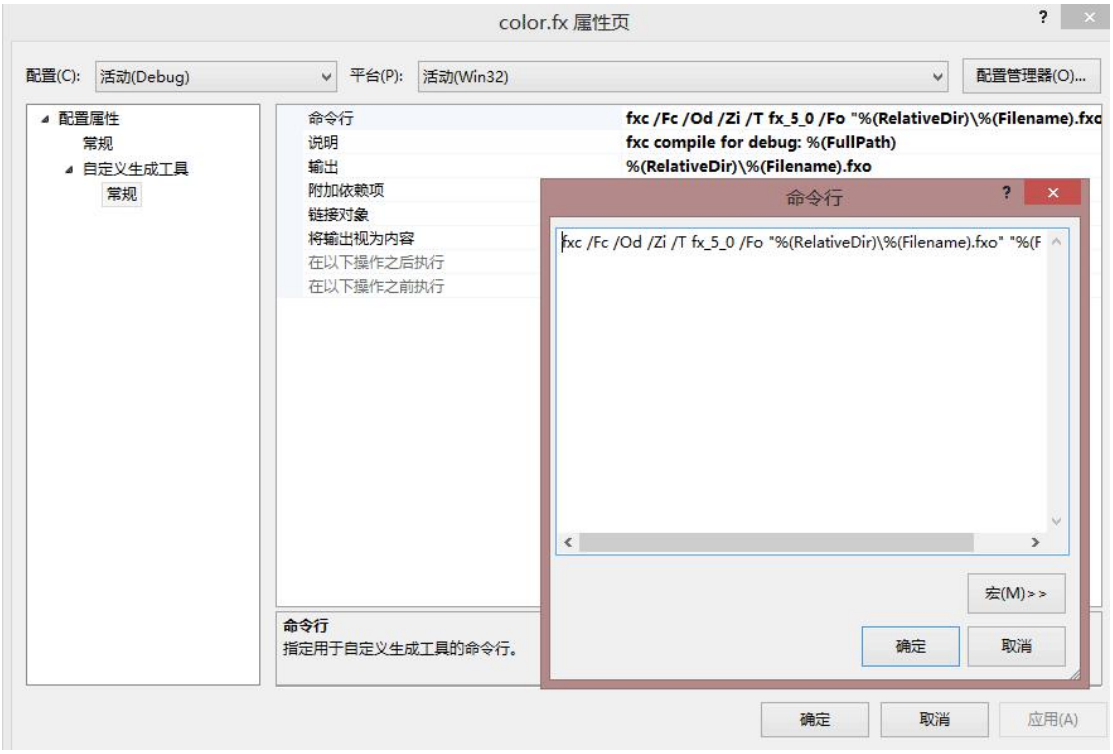



**图 6.6 在项目中添加自定义生成工具**

调试模式：

**fxc /Fc /Od /Zi /T fx_5_0 /Fo " % (RelativeDir)\% (Filename).fxo " " % (FuIIPath) "**

发布模式：

**fxc /T fx_5_0 /Fo " o/o (RelativeDir)\% (Filename).fxo" " % (FuIIPath) "**

你可以在 SDK 文档中找到 fxc 完整的编译参数说明。在调试模式中我们使用了以下三个参数，“/Fc /Od /Zi”分别对应输出汇编指令，禁用优化，开启调试信息。

现在当生成项目时，就会在每个 effect 上调用 *fxc* 并生成它的编译版本，以后缀为.fxo 的文件的形式保存。而且，如果有来自于 *fxc* 的编译警告或错误，会在调试输出窗口显示相关信息。例如，如果在 color.fx 文件中打错了一个变量的名称：

```
// 应该是 gWorldViewProj 而不是 worldViewProj!
vout.PosH = mul(float4(vin.Pos, l.Of), worldViewProj);
```

在调试输出窗口就会显示从这个错误引发的一系列错误的信息（第一条错误信息是修正的关键）：

```
error X3004: undeclared identifier  'worldViewProj'
error X3013: 'mul': intrinsic function does not take 2 parameters
error X3013: Possible intrinsic functions are:
error X3013: mul(float, float)…
```

在编译阶段获取错误信息要比运行时获取方便得多。现在我们在生成过程中编译 effect 文件（.fxo），再也不需要在运行时进行这个操作了（即，我们无须再调用 **D3DX11CompileFromFile** 方法了）。但是，我们仍需要从.fxo 文件中加载编译过的 shader，并将它们传递给 **D3DX11CreateEffectFromMemory** 方法。这个工作可以通过使用 C++的文件输入功能实现：

```
std::ifstream fin("fx/color.fxo",std::ios::binary);

fin.seekg(0, std::ios_base::end);
int size = (int)fin.tellg();
fin.seekg(0, std::ios_base::beg);
std::vector<char> compiledShader(size);

fin.read(&compiledShader[0],size);
fin.close();

HR(D3DX11CreateEffectFromMemory(&compiledShader[0],size,
   0, md3dDevice, &mFX));
```

除了在颜色立方体演示程序中我们在运行时编译了 shader 之外，本书的其他示例都是在生成过程中编译了所有 shader。

## 6.8.6 将 effect 框架作为“着色器生成器”

在本节的一开始我们提到过一个 effect 可以包含多个 technique。那为什么我们要使用多个 technique 呢？下面我们用阴影绘制为例子解释一下这个问题，但不会讨论实现阴影的细节内容。显然，阴影质量越高，要求的资源就越多。为了支持用户不同等级的显卡，我们可能会提供低、中、高不同质量的阴影技术。因此，即使只有一个阴影效果，我们也会使用多

个 technique 去实现它。我们的阴影 effect 文件如下所示：

```
// 省略了常量缓冲，顶点结构等代码...
VertexOut VS(Vertexln vin) {/* Omit implementation details */}

float4 LowQualityPS(VertexOut pin) : SV_Target
{
    /* Do work common to all quality levels */
    /* Do low quality specific stuff */
    /* Do more work common to all quality levels */
}

float4 MediumQualityPS(VertexOut pin) : SV_Target
{
    /* Do work common to all quality levels */
    /* Do medium quality specific stuff */
    /* Do more work common to all quality levels */
}

float4 HighQualityPS(VertexOut pin) : SV_Target
{
    /* Do work common to all quality levels */
    /* Do high quality specific stuff */
    /* Do more work common to all quality levels */
}

technique11 ShadowsLow
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
        SetPixeIShader(CompileShader(ps_5_0, LowQualityPS()));
    }
}

technique11 ShadowsMedium
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
        SetPixeIShader(CompileShader(ps_5_0, MediumQualityPS()));
    }
}

technique11 ShadowsHigh
```

```
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
        SetPixeIShader(CompileShader(ps_5_0, HighQualityPS ()));
    }
}
```

C++应用程序会侦测玩家的显卡等级，选择最合适的 technique 进行渲染。

**注意**：前面的代码假设三个不同的阴影 technique 只在像素着色器中有所区别，所有的 technique 共享相同的顶点着色器。但是，每个 technique 都有不同的顶点着色器也是有可能的。

前面的实现中还有一个问题：即使像素着色器的代码是不同的，但是还是有一些通用的代码是重复的。建议使用条件分支语句解决这个问题。在 shader 中使用动态分支语句代价不菲，所以只在必要时才使用它们。其实我们真正想要的是一个条件编译，它可以生成不同的 shader 代码，但又不使用分支指令。幸运的是，effect 框架提供了一个方法可以解决这个问题。下面是具体实现：

```
// 省略常量缓冲，顶点结构等...
VertexOut VS(VertexIn vin) {/* 省略代码细节 */}
#define LowQuality 0
#define MediumQuality 1
#define HighQuality 2

float4 PS(VertexOut pin, uniform int gQuality) : SV_Target
{
    /* Do work common to all quality levels */
    if(gQuality == LowQuality)
    {
       /* Do low quality specific stuff */
    }
    elseif(gQuality == MediumQuality)
    {
        /* Do medium quality specific stuff */
    }
    else
    {
        /* Do high quality specific stuff */
    }
    /* Do more work common to all quality levels */
}

technique11 ShadowsLow
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
```

```
        SetPixeIShader(CompileShader(ps_5_0, PS(LowQuality)));
    }
}

technique11 ShadowsMedium
{
   pass P0
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
        SetPixeIShader(CompileShader(ps_5_0, PS(MediumQuality)));
    }
}

techniquell ShadowsHigh
{
   pass P0
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
        SetPixeIShader(CompileShader(ps_5_0, PS(HighQuality)));
    }
}
```

我们在像素着色器中添加了一个额外的 uniform 参数，用来表示阴影质量等级。这个参数值是不同的，但对每个像素来说却是不变的，but is instead uniform/constant。Moreover，we do not change it at runtime either，like we change constant buffer variables。我们是在编译时设置这些参数的，而且这些值在编译时就是已知的，所以 effect 框架会基于这个值生成不同的 shader 变量。这样，我们不用复制代码（effect 框架帮我们在编译时复制了这些代码）就可以生成低、中、高三种不同阴影质量的 shader 代码，而且没有用到条件分支语句。

下面的两个例子是使用 shader 生成器的常见情景：

**1．是否需要纹理**？有个应用程序需要在一些物体上施加纹理，而另一些物体不使用纹理。一个解决方法是创建两个像素着色器，一个提供纹理而另一个不提供。或者我们也可以使用 shader 生成技巧创建两个像素着色器，然后在 C++程序中选择期望的 technique。

```
float4 PS(VertexOut pin, uniform bool gApplyTexture) : SV_Target
{
    /* Do common work */
    if(gApplyTexture)
    {
        /* Apply texture */
    }
    /* Do more common work */
}

technique11 BasicTech
{
    pass P0
```

```
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
        SetPixelShader(CompileShader(ps_5_0, PS(false)));
    }
}

technique11 TextureTech
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
        SetPixelShader(CompileShader(ps_5_0,PS(true)));
    }
}
```

**2．使用多少个光源？**一个游戏关卡可能会支持 1 至 4 个光源。光源越多，光照计算就越慢。我们可以基于光源数量设计不同的顶点着色器，或者也可以使用 shader 生成技巧创建四个顶点着色器，然后在 C++程序中根据当前激活的光源数量选择期望的 technique：

```
VertexOut VS(VertexOut pin, uniform int gLightCount)
{
    /* Do common work */
    for(int i = 0; i< gLightCount; ++i)
    {
        /* do lighting work */
    }
    /* Do more common work */
}

technique11 Light1
{
    P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS(1)));
        SetPixeIShader(CompileShader(ps_5_0, PS()));
    }
}

technique11 Light2
{
    P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS(2)));
        SetPixeIShader(CompileShader(ps_5_0, PS()));
    }
}
```

```
technique11 Light3
{
    P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS(3)));
        SetPixeIShader(CompileShader(ps_5_0, PS()));
    }
}
technique11 Light4
{
    P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS(4)));
        SetPixeIShader(CompileShader(ps_5_0, PS()));
    }
}
```

参数也可以不止一个。要将阴影质量，纹理和多个光源组合在一起，我们可以使用以下的顶点和像素着色器：

```
VertexOut VS(VertexOut pin, uniform int gLightCount)
{}
float4 PS(VertexOut pin,uniform int gQuality,uniform bool
gApplyTexture) : SV_Target
{}
```

要创建一个使用低质量阴影，两个光源，不使用纹理的 technique，我们可以这样写代码：

```
technique11 LowShadowsTwoLightsNoTextures
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0, VS(2)));
        SetPixeIShader(CompileShader(ps_5_0,PS(LowQuality,false)));
    }
}
```

## 6.8.7 对应的汇编代码

In case you do not bother to ever look at the assembly output of one of your effect files, we show what one looks like in this section.

我们不会解释汇编代码，但是，如果你以前学过汇编，你可能会认得代码中的 mov 指令，and

maybe could guess thatdp4 does a 4D dot product. Even without understanding the assembly, the listing gives some useful

information. It clearly identifies the input and output signatures, and gives the approximate

mstruction count, which is one useful metric

to understand how expensive/complex your shader is. Moreover, we see that multiple versions of our shaders really were generated based on the compile time parameters with no branching instructions. The effect file we use is the same as the one shown in ~6.8.1,except that we add a simple uniform bool parameter to generate two techniques:

```
float4 PS(VertexOut pin, uniform bool gUseColor) : SV_Target
                {
 if(gUs e Color)
{
   return pin.Color;
else
{
   return float4(0, 0, 0, 1);          techniquell ColorTech
                {
pass PO
{
     SetVe rtexShade r(Compile Shade r(vs_5_0,   VSO》;
   SetPixeIShader(CompileShader(ps_5_0, PS(true》);
        techniquell NoColorTech
                {
pass PO
{
     SetVertexShader(CompileShade r(vs_5_0,   VS(》);
   SetPixeIShader(CompileShader(ps_5_0, PS(false》);
     //
            // FX Version: fx_5_0
     //
              11 1 local buffer(s)
     //
          cbuffer cbPerObject
                {
float4x4 gWorldViewProj; // Offset: O, size: 64
     //
              11 1 groups(s)
     //
fxgroup
                {
//
11 2 technique(s)
//
techniquell ColorTech
{
    pass PO
```

```
{
        VertexShader = asm {
        //
                    // Generated by Microsoft (R) HLSL Shader Compiler 9.29.952.3111
        //
        //
                      // Buffer Definitions:
        //
                    // cbuffer cbPerObject
                   // {
        //
                      // float4x4 gWorldViewProj; // Offset: O Size: 64
        //
                    // }
        //
        //
                      // Resource Bindings:
        //
                      // Name Type Format Dim Slot Elements
                    // -----.
                    // cbPerObject cbuffer NA NA 0 1
        //     //
  //
           // Input signature:
  //
      // Name Index Mask Register SysValue Format Used
      // -----
        // POSlrflON 0 xyz O NONE float xyz
        // COLOR O xyzw l NONE float xyzw
  //
  //
      // Output signature:
  //
      // Name Index Mask Register SysValue Format Used
      // -----
      // SV_POSITION O xyzw O POS float xyzw
        // COLOR O xyzw l NONE float xyzw
  //
vs_5_0
      dcl_globaIFlags refactoringAllowed
                  dcl_constantbuffe r    cb0 [4],    imme diatelnde xe d
        dcljnput vO.xyz
        dcl_input vl.xyzw
        dcl_output_siv oO.xyzw, position
```

```
        dcl_output ol.xyzw
       dcl_temps 1
        mov rO.xyz, vO.xyzx
           mov rO.w, l(l.000000)
        dp4 00.x, rO.xyzw, cbO[O].xyzw
        dp4 00.y, rO.xyzw, cbO[l].xyzw
        dp4 00.z, rO.xyzw, cb0[2].xyzw
        dp4 00.w, rO.xyzw, cb0[3].xyzw
        mov ol.xyzw, vl.xyzw
  ret
          // Approximately 8 instruction slots used
};
PixeIShader = asm {
    //
         // Generated by Microsoft (R) HLSL Shader Compiler 9.29.952.3111
    //
    //
    //
              // Input signature:
    //
         // Name Index Mask Register SysValue Format Used
         // -----
         // SV_POSIUON O xyzw o POS float
          // COLOR O xyzw l NONE float xyzw
    //
    //
         // Output signature:
    //
         // Name Index Mask Register SysValue Format Used
         // -----
         // SV_Target O xyzw O TARGET float xyzw
    //
ps_5_0
       dcl_globaIFlags refactoringAllowed
        dcl_input_ps linear vl.xyzw
        dcl_output oO.xyzw
        mov oO.xyzw, vl.xyzw
  ret                                  // Approximately 2 instruction slots used
         };
techniquell NoColorTech
{
    pass PO
     {
           VertexShader = asm {
```

```
//
    // Generated by Microsoft (R) HLSL Shader Compiler 9.29.952.3111
//
//
    // Buffer Definitions:
//
    // cbuffer cbPerObject
    // {
//
    // float4x4 gWorldViewProj; // Offset: O Size: 64
//
    // }
//
//
    // Resource Bindings:
//
    // Name Type Format Dim Slot Elements
    // -----.
    // cbPerObject cbuffer NA NA 0 1
//
//
//
        // Input signature:
//
    // Name Index Mask Register SysValue Format Used
    // -----.
      // POSlrflON O xyz O NONE float xyz
        // COLOR O xyzw l NONE float xyzw
//
//
    // Output signature:
//
    // Name Index Mask Register SysValue Format Used
    // -----.
    // SV_POSITION O xyzw O POS float xyzw
        // COLOR 0 xyzw l NONE float xyzw
//
vs_5_0
    dcl_globaIFlags refactoringAllowed
                                dcl_constantbuffe r  cb0 [4], imme
diatelnde xe d
    dcljnput vO.xyz
    dcl_input vl.xyzw
     dcl_output_siv oO.xyzw, position
```

```
                    dcl_output ol.xyzw
                   dcl_temps 1
                       mov rO.xyz, vO.xyzx
                           mov rO.w, 1(1.000000)
                    dp4 00.x, rO.xyzw, cbO[O].xyzw
                    dp4 00.y, rO.xyzw, cbO[l].xyzw
                     dp4 00.z, rO.xyzw, cb0[2].xyzw
                    dp4 00.w, rO.xyzw, cb0[3].xyzw
                     mov ol.xyzw, vl.xyzw}
}
}
   ret
          // Approximately 8 instruction slots used
};
PixeIShader = asm {
Generated by Microsoft (R) HLSL Shader Compiler 9.29.952.3111
Input signature:
Name Index Mask Register SysValue Format Used
SV_POSITION O xyzw O POS float
COLOR O xyzw l NONE float
Output signature:
Name Index Mask Register SysValue Format Used
SV_Target O xyzw O TARGET float xyzw
ps_5_0
       dcl_globaIFlags refactoringAllowed
        dcl_output oO.xyzw
          mov oO.xyzw, 1(0,0,0,1.000000)
   ret
          // Approximately 2 instruction slots used
```

6.9 BOX DEMO

At last, we have covered enough material to present a simple demo, which renders a colored box. This example essentially puts

everythmg we have discussed in this chapter up to now into a single program. The reader should study the code and refer back to the

previous sections of this chapter until every line is understood. Note that the program uses the "color.fx" effect, as written in ~6.8.1.

，／击击出士击击出士击出去击击出去击击出士击击出士击击击击击出去击击出去击击出士击击出士击出去击击出去击击古士击击出士击击出去击出去击击出去击

```
// BoxDemo.cpp by Frank Luna (C) 2011 All Rights Reserved.
//
 ／／Demonstrates rendering a colored box.
||
/ / Controls：
 ／／Hold the left mouse button down and move the mouse to rotate.
```

/ / Hold the right mouse button down to zoom in and out.

‖

，/ 击出去击击出去击击古去士击古去士出去击击出去击击出去士击古去士击古去击出去击击出去击击古去士击古去士击击击击出去击击出去士击古去士击古去击

#include "d3dApp.h"

#include "d3dxllEffect.h"

#include "MathHelper.h"

struct Vertex

XMFLOArl3 Pos;

XMFLOAT4 Color;

// // // // // // // // // // // // // // // // // // // // //

暂未整理。

# 6.9 颜色立方体演示程序

我们已经讲解了足够多的内容，现在我们可以开始编写一个简单的颜色立方体演示程序了。这个例子基本上包含了我们前面讲到的所有内容。读者应该对照前面的几节，仔细研究这些代码，直到把每一行代码都弄懂为止。注意，程序使用了 6.8.1 节编写的“color.fx”effect。

```
//***************************************************************
************************
// BoxDemo.cpp by Frank Luna (C) 2011 All Rights Reserved.
//
// Demonstrates rendering a colored box.
//
// Controls:
//     Hold the left mouse button down and move the mouse to rotate.
//      Hold the right mouse button down to zoom in and out.
//
//***************************************************************
************************

#include "d3dApp.h"
#include "d3dx11Effect.h"
#include "MathHelper.h"

struct Vertex
{
    XMFLOAT3 Pos;
    XMFLOAT4 Color;
};

class BoxApp : public D3DApp
```

```
{
public:
    BoxApp(HINSTANCE hInstance);
    ~BoxApp();

    bool Init();
    void OnResize();
    void UpdateScene(float dt);
    void DrawScene();

    void OnMouseDown(WPARAM btnState, int x, int y);
    void OnMouseUp(WPARAM btnState, int x, int y);
    void OnMouseMove(WPARAM btnState, int x, int y);

private:
    void BuildGeometryBuffers();
    void BuildFX();
    void BuildVertexLayout();

private:
    ID3D11Buffer* mBoxVB;
    ID3D11Buffer* mBoxIB;

    ID3DX11Effect* mFX;
    ID3DX11EffectTechnique* mTech;
    ID3DX11EffectMatrixVariable* mfxWorldViewProj;

    ID3D11InputLayout* mInputLayout;

    XMFLOAT4X4 mWorld;
    XMFLOAT4X4 mView;
    XMFLOAT4X4 mProj;

    float mTheta;
    float mPhi;
    float mRadius;

    POINT mLastMousePos;
};

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE prevInstance,
                   PSTR cmdLine, int showCmd)
{
    // Enable run-time memory check for debug builds.
```

```
#if defined(DEBUG) | defined(_DEBUG)
    _CrtSetDbgFlag( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
#endif

    BoxApp theApp(hInstance);

    if( !theApp.Init() )
        return 0;

    return theApp.Run();
}


BoxApp::BoxApp(HINSTANCE hInstance)
: D3DApp(hInstance), mBoxVB(0), mBoxIB(0), mFX(0), mTech(0),
  mfxWorldViewProj(0), mInputLayout(0),
  mTheta(1.5f*MathHelper::Pi),          mPhi(0.25f*MathHelper::Pi),
mRadius(5.0f)
{
    mMainWndCaption = L"Box Demo";

    mLastMousePos.x = 0;
    mLastMousePos.y = 0;

    XMMATRIX I = XMMatrixIdentity();
    XMStoreFloat4x4(&mWorld, I);
    XMStoreFloat4x4(&mView, I);
    XMStoreFloat4x4(&mProj, I);
}

BoxApp::~BoxApp()
{
    ReleaseCOM(mBoxVB);
    ReleaseCOM(mBoxIB);
    ReleaseCOM(mFX);
    ReleaseCOM(mInputLayout);
}

bool BoxApp::Init()
{
    if(!D3DApp::Init())
        return false;

    BuildGeometryBuffers();
```

```
    BuildFX();
    BuildVertexLayout();

    return true;
}

void BoxApp::OnResize()
{
    D3DApp::OnResize();

    // 当窗口大小改变时，需要更新横纵比，并重新计算投影矩阵
    XMMATRIX  P  =  XMMatrixPerspectiveFovLH(0.25f*MathHelper::Pi,
AspectRatio(), 1.0f, 1000.0f);
    XMStoreFloat4x4(&mProj, P);
}

void BoxApp::UpdateScene(float dt)
{
    // Convert Spherical to Cartesian coordinates.
    float x = mRadius*sinf(mPhi)*cosf(mTheta);
    float z = mRadius*sinf(mPhi)*sinf(mTheta);
    float y = mRadius*cosf(mPhi);

    // 创建视矩阵
    XMVECTOR pos    = XMVectorSet(x, y, z, 1.0f);
    XMVECTOR target = XMVectorZero();
    XMVECTOR up     = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);

    XMMATRIX V = XMMatrixLookAtLH(pos, target, up);
    XMStoreFloat4x4(&mView, V);
}

void BoxApp::DrawScene()
{
    md3dImmediateContext->ClearRenderTargetView(mRenderTargetView,
reinterpret_cast<const float*>(&Colors::LightSteelBlue));
    md3dImmediateContext->ClearDepthStencilView(mDepthStencilView,
D3D11_CLEAR_DEPTH|D3D11_CLEAR_STENCIL, 1.0f, 0);

    md3dImmediateContext->IASetInputLayout(mInputLayout);

md3dImmediateContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPO
LOGY_TRIANGLELIST);
```

```
    UINT stride = sizeof(Vertex);
    UINT offset = 0;
    md3dImmediateContext->IASetVertexBuffers(0, 1, &mBoxVB, &stride,
&offset);
    md3dImmediateContext->IASetIndexBuffer(mBoxIB,
DXGI_FORMAT_R32_UINT, 0);

    // Set constants
    XMMATRIX world = XMLoadFloat4x4(&mWorld);
    XMMATRIX view  = XMLoadFloat4x4(&mView);
    XMMATRIX proj  = XMLoadFloat4x4(&mProj);
    XMMATRIX worldViewProj = world*view*proj;

    mfxWorldViewProj->SetMatrix(reinterpret_cast<float*>(&worldVie
wProj));

    D3DX11_TECHNIQUE_DESC techDesc;
    mTech->GetDesc( &techDesc );
    for(UINT p = 0; p < techDesc.Passes; ++p)
    {
        mTech->GetPassByIndex(p)->Apply(0, md3dImmediateContext);

        // 立方体有 36 个索引
        md3dImmediateContext->DrawIndexed(36, 0, 0);
    }

    HR(mSwapChain->Present(0, 0));
}

void BoxApp::OnMouseDown(WPARAM btnState, int x, int y)
{
    mLastMousePos.x = x;
    mLastMousePos.y = y;

    SetCapture(mhMainWnd);
}

void BoxApp::OnMouseUp(WPARAM btnState, int x, int y)
{
    ReleaseCapture();
}

void BoxApp::OnMouseMove(WPARAM btnState, int x, int y)
{
```

```
    if( (btnState & MK_LBUTTON) != 0 )
    {
        // Make each pixel correspond to a quarter of a degree.
        float dx = XMConvertToRadians(0.25f*static_cast<float>(x -
mLastMousePos.x));
        float dy = XMConvertToRadians(0.25f*static_cast<float>(y -
mLastMousePos.y));

        // Update angles based on input to orbit camera around box.
        mTheta += dx;
        mPhi   += dy;

        // Restrict the angle mPhi.
        mPhi = MathHelper::Clamp(mPhi, 0.1f, MathHelper::Pi-0.1f);
    }
    else if( (btnState & MK_RBUTTON) != 0 )
    {
        // Make each pixel correspond to 0.005 unit in the scene.
        float dx = 0.005f*static_cast<float>(x - mLastMousePos.x);
        float dy = 0.005f*static_cast<float>(y - mLastMousePos.y);

        // Update the camera radius based on input.
        mRadius += dx - dy;

        // Restrict the radius.
        mRadius = MathHelper::Clamp(mRadius, 3.0f, 15.0f);
    }

    mLastMousePos.x = x;
    mLastMousePos.y = y;
}

void BoxApp::BuildGeometryBuffers()
{
    // 创建顶点缓冲
    Vertex vertices[] =
    {
        {       XMFLOAT3(-1.0f,       -1.0f,       -1.0f),       (const
float*)&Colors::White   },
        {       XMFLOAT3(-1.0f,       +1.0f,       -1.0f),       (const
float*)&Colors::Black   },
        {       XMFLOAT3(+1.0f,       +1.0f,       -1.0f),       (const
float*)&Colors::Red     },
        {       XMFLOAT3(+1.0f,       -1.0f,       -1.0f),       (const
```

```
float*)&Colors::Green   },
        {       XMFLOAT3(-1.0f,       -1.0f,       +1.0f),       (const
float*)&Colors::Blue    },
        {       XMFLOAT3(-1.0f,       +1.0f,       +1.0f),       (const
float*)&Colors::Yellow  },
        {       XMFLOAT3(+1.0f,       +1.0f,       +1.0f),       (const
float*)&Colors::Cyan    },
        {       XMFLOAT3(+1.0f,       -1.0f,       +1.0f),       (const
float*)&Colors::Magenta }
    };

    D3D11_BUFFER_DESC vbd;
    vbd.Usage = D3D11_USAGE_IMMUTABLE;
    vbd.ByteWidth = sizeof(Vertex) * 8;
    vbd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
    vbd.CPUAccessFlags = 0;
    vbd.MiscFlags = 0;
    vbd.StructureByteStride = 0;
    D3D11_SUBRESOURCE_DATA vinitData;
    vinitData.pSysMem = vertices;
    HR(md3dDevice->CreateBuffer(&vbd, &vinitData, &mBoxVB));


    // 创建索引缓冲

    UINT indices[] = {
        // 前表面
        0, 1, 2,
        0, 2, 3,

        // 后表面
        4, 6, 5,
        4, 7, 6,

        // 左表面
        4, 5, 1,
        4, 1, 0,

        // 右表面
        3, 2, 6,
        3, 6, 7,

        // 上表面
        1, 5, 6,
```

```
        1, 6, 2,

        // 下表面
        4, 0, 3,
        4, 3, 7
    };

    D3D11_BUFFER_DESC ibd;
    ibd.Usage = D3D11_USAGE_IMMUTABLE;
    ibd.ByteWidth = sizeof(UINT) * 36;
    ibd.BindFlags = D3D11_BIND_INDEX_BUFFER;
    ibd.CPUAccessFlags = 0;
    ibd.MiscFlags = 0;
    ibd.StructureByteStride = 0;
    D3D11_SUBRESOURCE_DATA iinitData;
    iinitData.pSysMem = indices;
    HR(md3dDevice->CreateBuffer(&ibd, &iinitData, &mBoxIB));
}

void BoxApp::BuildFX()
{
    DWORD shaderFlags = 0;
    #if defined( DEBUG ) || defined( _DEBUG )
        shaderFlags |= D3D10_SHADER_DEBUG;
        shaderFlags |= D3D10_SHADER_SKIP_OPTIMIZATION;
    #endif

    ID3D10Blob* compiledShader = 0;
    ID3D10Blob* compilationMsgs = 0;
    HRESULT hr = D3DX11CompileFromFile(L"FX/color.fx", 0, 0, 0,
"fx_5_0", shaderFlags,
        0, 0, &compiledShader, &compilationMsgs, 0);

    // compilationMsgs 中包含错误或警告信息
    if( compilationMsgs != 0 )
    {
        MessageBoxA(0, (char*)compilationMsgs->GetBufferPointer(), 0,
0);
        ReleaseCOM(compilationMsgs);
    }

    // 就算没有 compilationMsgs，也需要确保没有其他错误
    if(FAILED(hr))
    {
```

```
        DXTrace(__FILE__,              (DWORD)__LINE__,              hr,
L"D3DX11CompileFromFile", true);
    }

    HR(D3DX11CreateEffectFromMemory(compiledShader->GetBufferPoint
er(), compiledShader->GetBufferSize(),
        0, md3dDevice, &mFX));

    // 编译完成释放资源
    ReleaseCOM(compiledShader);

    mTech    = mFX->GetTechniqueByName("ColorTech");
    mfxWorldViewProj                                              =
mFX->GetVariableByName("gWorldViewProj")->AsMatrix();
}

void BoxApp::BuildVertexLayout()
{
    // 顶点输入布局描述
    D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
    {
        {"POSITION",    0,    DXGI_FORMAT_R32G32B32_FLOAT,    0,    0,
D3D11_INPUT_PER_VERTEX_DATA, 0},
        {"COLOR",       0,  DXGI_FORMAT_R32G32B32A32_FLOAT,  0,  12,
D3D11_INPUT_PER_VERTEX_DATA, 0}
    };

    // 创建顶点输入布局
    D3DX11_PASS_DESC passDesc;
    mTech->GetPassByIndex(0)->GetDesc(&passDesc);
    HR(md3dDevice->CreateInputLayout(vertexDesc,                   2,
passDesc.pIAInputSignature,
        passDesc.IAInputSignatureSize, &mInputLayout));
}
```

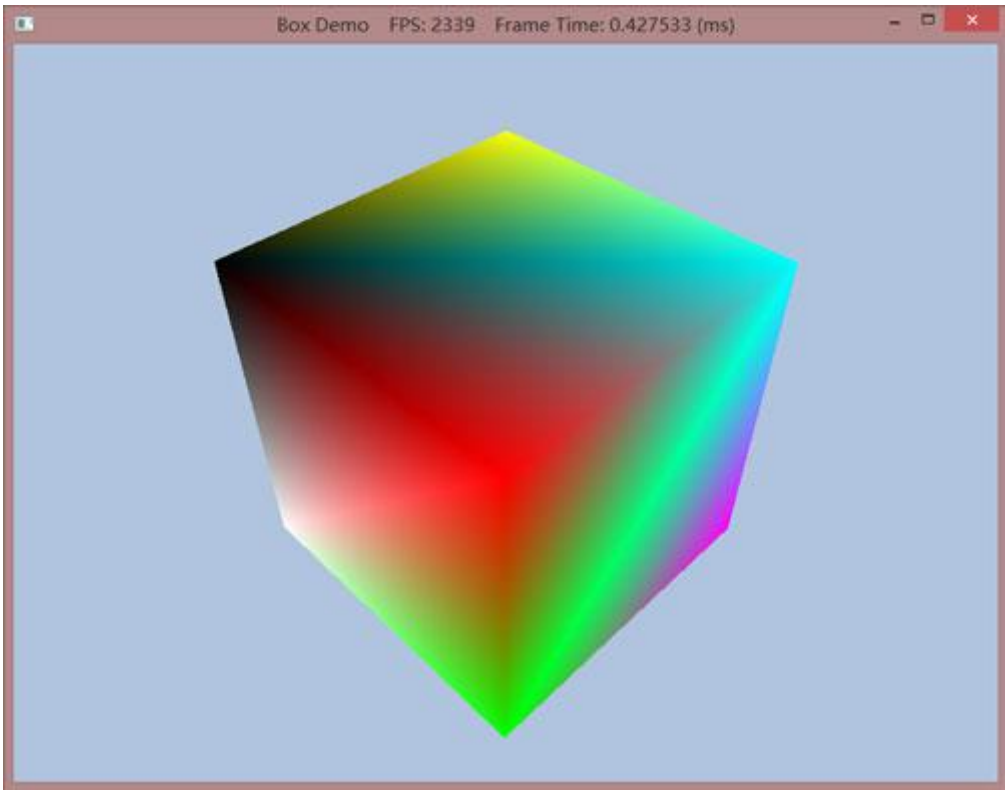



**图 6.7 立方体演示程序的屏幕截图**

# 6.10 山峰与河谷演示程序

本章还包含了一个“山峰与河谷”的例子。它使用了与颜色立方体演示程序相同的 Direct3D 方法，只是它绘制的几何体更复杂一些。它主要讲解的是如何使用代码来生成三角形网格；这种几何体在实现地形渲染和水体渲染时非常有用。

实数函数 $y = f(x, z)$可以生成一个“漂亮的”曲面。我们可以通过构造一个 $xz$ 平面上的网格来模拟该曲面，其中每个四边形都由两个三角形构成。然后，我们将每个网格点代入该函数；参见图 6.8。

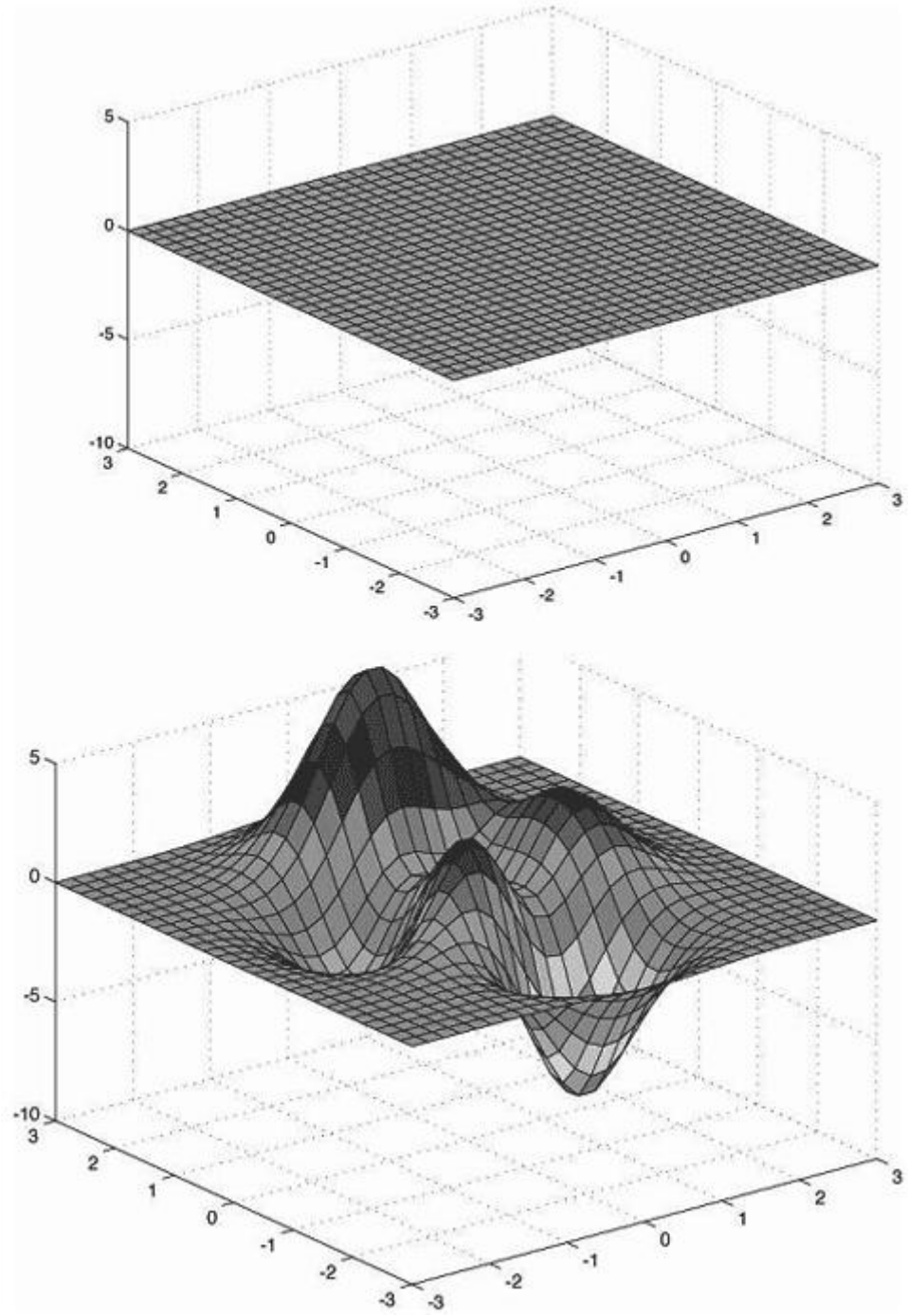



**图 6.8 （上）建立 *xz* 平面上的网格。（下）将每个网格点代入函数 *f*(*x* ,*z*)，得到 *y* 坐标。通过将大量的点(*x*, *f*(*x*, *z*), *z*)连接起来，即可形成上述曲面。**

## 6.10.1 生成网格顶点

下面的主要任务是创建 *xz* 平面上的网格。一个包含 $m \times n$ 个顶点的网格可以生成$(m-1)\times(n-1)$个多边形（或单元格），如图 6.9 所示。每个多边形由两个三角形组成，一共 $2\times(m-1)\times(n-1)$个三角形。如果网格的宽度为 *w*、深度为 *d*，则 *x* 轴方向上的单元格间距为 $dx = w/(n-1)$、 轴方向上的单元格间距为 $dz=d/(m-1)$。我们从左上角开始生成顶点，逐行计算每个顶点的坐标。在 *xz* 平面上，第 *ij* 个网格顶点的坐标为：

$v_{ij} = (-0.5w + j \cdot dx\ ,\ 0.0\ ,\ 0.5d - i \cdot dz)$

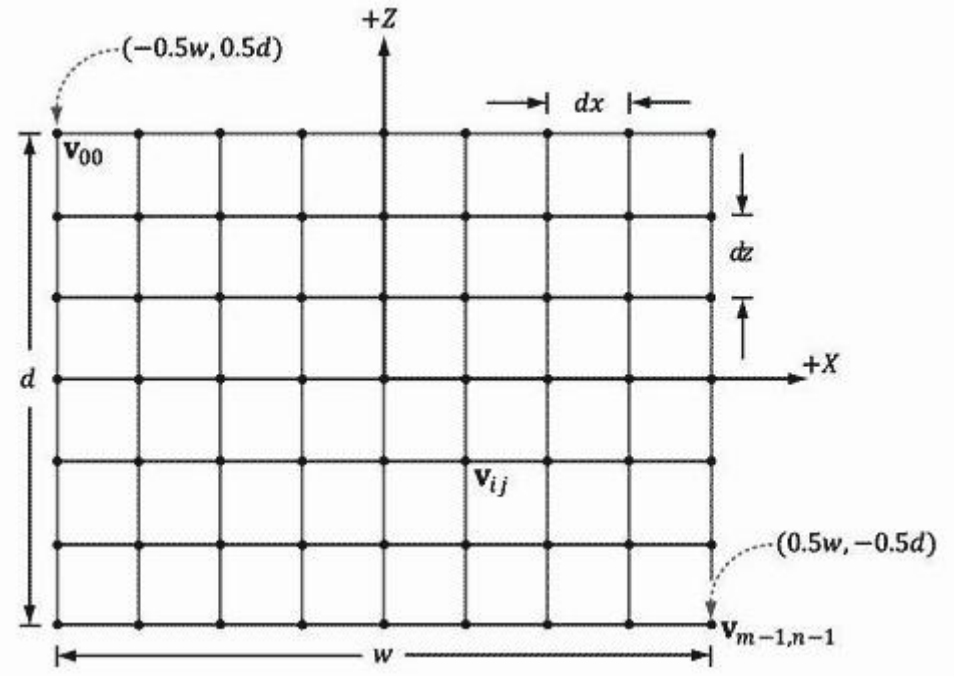



**图 6.9 网格结构。**

下面的代码实现了这一工作。

```
void GeometryGenerator::CreateGrid(float width, float depth, UINT m,
UINT n, MeshData& meshData)
{
    UINT vertexCount = m*n;
    UINT faceCount   = (m-1)*(n-1)*2;

    //
    // 创建顶点
    //

    float halfWidth = 0.5f*width;
    float halfDepth = 0.5f*depth;

    float dx = width / (n-1);
    float dz = depth / (m-1);

    float du = 1.0f / (n-1);
    float dv = 1.0f / (m-1);

    meshData.Vertices.resize(vertexCount);
    for(UINT i = 0; i < m; ++i)
    {
        float z = halfDepth - i*dz;
        for(UINT j = 0; j < n; ++j)
```

```
        {
            float x = -halfWidth + j*dx;

            meshData.Vertices[i*n+j].Position = XMFLOAT3(x, 0.0f, z);
            meshData.Vertices[i*n+j].Normal   = XMFLOAT3(0.0f, 1.0f,
0.0f);
            meshData.Vertices[i*n+j].TangentU = XMFLOAT3(1.0f, 0.0f,
0.0f);

            // Stretch texture over grid.
            meshData.Vertices[i*n+j].TexC.x = j*du;
            meshData.Vertices[i*n+j].TexC.y = i*dv;
        }
    }
}
```

**GeometryGenerator** 是一个工具类，用于生成诸如网格、球、圆柱体、盒子之类的几何形状，在本书的其他示例中都会用到这些形状。这个类在系统内存中生成数据，我们必须将这些数据复制到顶点和索引缓冲中。**GeometryGenerator** 创建的某些顶点数据在后面的章节中才会用到，这个演示程序不会用到，所以也无需将这些数据复制到顶点缓冲中。**MeshData** 结构体用于存储顶点和索引的集合列表。

```
class GeometryGenerator
{
public:
    struct Vertex
    {
        Vertex(){}
        Vertex(const XMFLOAT3& p, const XMFLOAT3& n, const XMFLOAT3&
t, const XMFLOAT2& uv)
            : Position(p), Normal(n), TangentU(t), TexC(uv){}
        Vertex(
            float px, float py, float pz,
            float nx, float ny, float nz,
            float tx, float ty, float tz,
            float u, float v)
            : Position(px,py,pz), Normal(nx,ny,nz),
              TangentU(tx, ty, tz), TexC(u,v){}

        XMFLOAT3 Position;
        XMFLOAT3 Normal;
        XMFLOAT3 TangentU;
        XMFLOAT2 TexC;
    };

    struct MeshData
```

```
    {
        std::vector<Vertex> Vertices;
        std::vector<UINT> Indices;
    };
…
};
```

## 6.10.2 生成网格索引

在完成顶点的计算之后，我们必须通过索引来定义网格三角形。我们再次从左上角开始逐行遍历每个四边形，通过计算索引来定义构成四边形的两个三角形。如图 6.10 所示，对于一个由 $m \times n$ 个顶点构成的网格来说，两个三角形的线性数组索引为：

$$\triangle ABC = (i \cdot n + j,\quad i \cdot n + j + 1,\quad (i+1) \cdot n + j)$$

$$\triangle CBD = ((i+1) \cdot n + j,\ i \cdot n + j + 1 \quad \cdot \quad (i+1) \cdot n + j + 1)$$

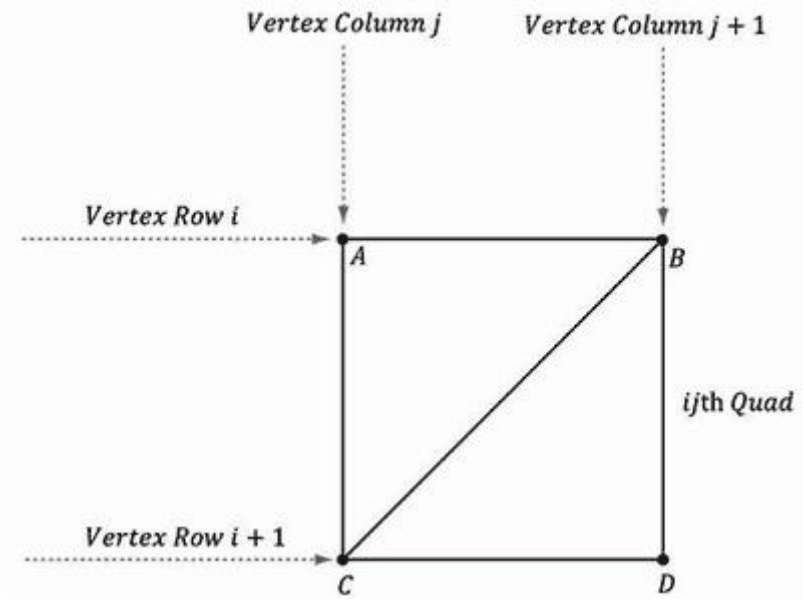



**图 6.10 第 *ij* 个四边形的顶点的索引。**

下面是对应的代码：

```
meshData.Indices.resize(faceCount*3); // 3 indices per face

// 遍历所有四边形并计算索引
UINT k = 0;
for(UINT i = 0; i < m-1; ++i)
{
    for(UINT j = 0; j < n-1; ++j)
    {
        meshData.Indices[k]   = i*n+j;
        meshData.Indices[k+1] = i*n+j+1;
        meshData.Indices[k+2] = (i+1)*n+j;
```

```
            meshData.Indices[k+3] = (i+1)*n+j;
            meshData.Indices[k+4] = i*n+j+1;
            meshData.Indices[k+5] = (i+1)*n+j+1;

            k += 6; // next quad
        }
    }
```

## 6.10.3 代入高度函数

创建了网格之后，我们要从 MeshData 中提取顶点元素，将平面网格转换为代表山丘的曲折表面，并基于顶点的高度（$y$ 坐标）设置它们的颜色。

```
struct Vertex
{
    XMFLOAT3 Pos;
    XMFLOAT4 Color;
};

void HillsApp::BuildGeometryBuffers()
{
    GeometryGenerator::MeshData grid;

    GeometryGenerator geoGen;

    geoGen.CreateGrid(160.0f, 160.0f, 50, 50, grid);

    mGridIndexCount = grid.Indices.size();

    //
    // 在每个顶点上附加高度函数。此外，还根据顶点的高度设置它们的颜色：
    // 沙滩为沙的颜色，小山为绿色，山顶为白色的雪。
    //

    std::vector<Vertex> vertices(grid.Vertices.size());
    for(size_t i = 0; i < grid.Vertices.size(); ++i)
    {
        XMFLOAT3 p = grid.Vertices[i].Position;

        p.y = GetHeight(p.x, p.z);

        vertices[i].Pos   = p;

        // 根据顶点高度设置颜色
```

```
        if( p.y < -10.0f )
        {
            // 沙滩色
            vertices[i].Color = XMFLOAT4(1.0f, 0.96f, 0.62f, 1.0f);
        }
        else if( p.y < 5.0f )
        {
            // 淡绿色
            vertices[i].Color = XMFLOAT4(0.48f, 0.77f, 0.46f, 1.0f);
        }
        else if( p.y < 12.0f )
        {
            // 深绿色
            vertices[i].Color = XMFLOAT4(0.1f, 0.48f, 0.19f, 1.0f);
        }
        else if( p.y < 20.0f )
        {
            // 棕色
            vertices[i].Color = XMFLOAT4(0.45f, 0.39f, 0.34f, 1.0f);
        }
        else
        {
            // 白色
            vertices[i].Color = XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f);
        }
    }

    D3D11_BUFFER_DESC vbd;
    vbd.Usage = D3D11_USAGE_IMMUTABLE;
    vbd.ByteWidth = sizeof(Vertex) * grid.Vertices.size();
    vbd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
    vbd.CPUAccessFlags = 0;
    vbd.MiscFlags = 0;
    D3D11_SUBRESOURCE_DATA vinitData;
    vinitData.pSysMem = &vertices[0];
    HR(md3dDevice->CreateBuffer(&vbd, &vinitData, &mVB));

    //
    // 将所有网格的索引放入一个索引缓冲中
    //

    D3D11_BUFFER_DESC ibd;
    ibd.Usage = D3D11_USAGE_IMMUTABLE;
    ibd.ByteWidth = sizeof(UINT) * mGridIndexCount;
```

```
    ibd.BindFlags = D3D11_BIND_INDEX_BUFFER;
    ibd.CPUAccessFlags = 0;
    ibd.MiscFlags = 0;
    D3D11_SUBRESOURCE_DATA iinitData;
    iinitData.pSysMem = &grid.Indices[0];
    HR(md3dDevice->CreateBuffer(&ibd, &iinitData, &mIB));
}
```

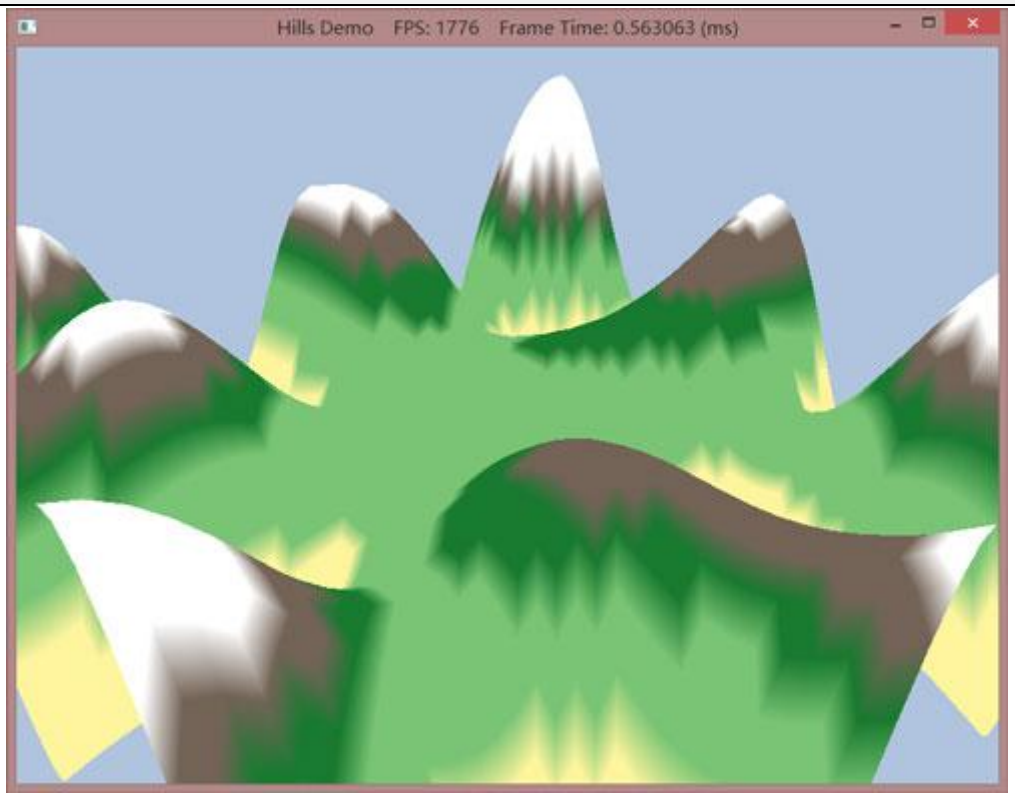



**图 6.11 山峰演示程序的截图。**

这个程序所用的函数 $f(x,z)$由以下代码给出：

```
float HillsApp::GetHeight(float x, float z)const
{
    return 0.3f*( z*sinf(0.1f*x) + x*cosf(0.1f*z) );
}
```

这个函数生成的图形看起了就像是山峰和山谷（见图 6.11），程序的其他部分与上一节颜色正方体的示例类似。

# 6.12 从文件加载几何体

虽然对于本书的某些示例来说，盒子、网格、球和圆柱形就足够了，但是还有些示例要绘制更加复杂的几何体。本书的后面我们会介绍如何从一个流行的 3D 模型格式加载 3D 网格。同时，我们已经将一个骷髅网格的几何体（见图 6.18）导出为一个顶点（只包含位置和法线向量）和索引的简单列表，可以使用标准的 C++文件 I/O 从文件中读取顶点和索引，并将它们复制到顶点和索引缓冲。文件的格式是一个非常简单的文本文件：

```
VertexCount:31076
TriangleCount:60339
VertexList(pos, normal)
{
    0. 592978  1. 92413  -2. 62486  0. 572276  0. 816877  0. 0721907
    0. 571224  1. 94331  -2. 66948  0. 572276  0. 816877  0. 0721907
    0. 609047  1. 90942  -2. 58578  0. 572276  0. 816877  0. 0721907
    …
}
TriangleList
{
    0  1  2
    3  4  5
    6  7  8
    …
}
```

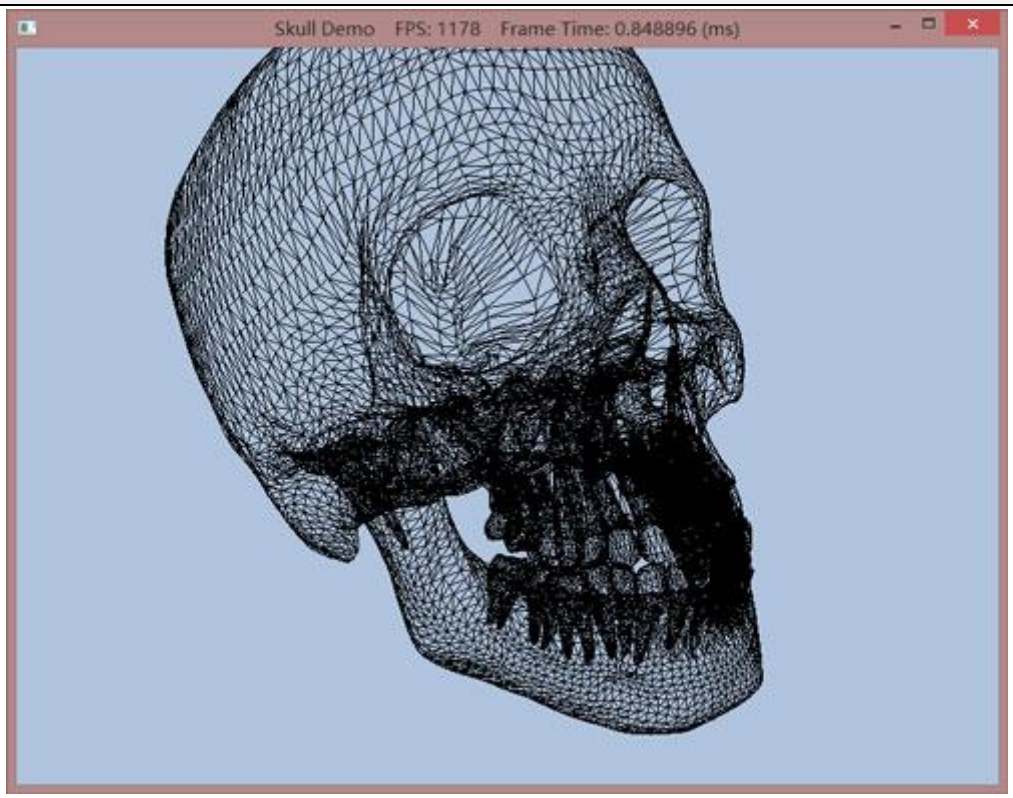



**图 6.18 “骷髅”演示示例的屏幕截图**

# 6.13 动态顶点缓冲

到目前为止，我们一直使用的是静态缓冲（static buffer），它的内容是在初始化时固定下来的。相比之下，动态缓冲（dynamic buffer）的内容可以在每一帧中进行修改。当实现一些动画效果时，我们通常使用动态缓冲区。例如，我们要模拟一个水波效果，并通过函数 $f(x, z, t)$来描述水波方程，计算当时间为 $t$ 时，$xz$ 平面上的每个点的高度。在这一情景中，我

们必须使用“山峰与河谷”中的那种三角形网格，将每个网格点代入 $f(x, z, t)$函数得到相应的水波高度。由于该函数依赖于时间 $t$（即，水面会随着时间而变化），我们必须在很短的时间内（比如 1/30 秒）重新计算这些网格点，以得到较为平滑的动画。所以，我们必须使用动态顶点缓冲区来实时更新三角形网格顶点的高度。

前面提到，为了获得一个动态缓冲区，我们必须在创建缓冲区时将 Usage 标志值指定为 **D3D11_USAGE_DYNAMIC**；同时，由于我们要向缓冲区写入数据，所以必须将 CPU 访问标志值指定为 **D3D11_CPU_ACCESS_WRITE**。

```
D3D11_BUFFER_DESC vbd;
vbd.Usage = D3D11_USAGE_DYNAMIC;
vbd.ByteWidth = sizeof(Vertex) * mWaves.VertexCount();
vbd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
vbd.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
vbd.MiscFlags = 0;
HR(md3dDevice->CreateBuffer(&vbd, 0, &mWavesVB));
```

**然后，使用 ID3D11Buffer::Map** 函数获取缓冲区内存的起始地址指针，并向它写入数据：

```
HRESULT ID3D11DeviceContext::Map(
    ID3D11Resource *pResource ,
    UINT Subresource,
    D3D11_MAP MapType,
    UINT MapFlags,
    D3D11_MAPPED_SUBRESOURCE *pMappedResource);
```

**1．pResource**：指向要访问的用于读/写的资源的指针。缓冲是一种 Direct3D 11 资源，其他类型的资源，例如纹理资源，也可以使用这个方法进行访问。

**2．Subresource**：包含在资源中的子资源的索引。后面我们会看到如何使用这个索引，而缓冲不包含子资源，所以设置为 0。

**3．MapType**：常用的标志有以下几个：

- **D3D11_MAP_WRITE_DISCARD**：让硬件抛弃旧缓冲，返回一个指向新分配缓冲的指针，通过指定这个标志，可以让我们写入新分配的缓冲的同时，让硬件绘制已抛弃的缓冲中的内容，可以防止绘制停顿。
- **D3D11_MAP_WRITE_NO_OVERWRITE**：我们只会写入缓冲中未初始化的部分；通过指定这个标志，可以让我们写入未初始化的缓冲的同时，让硬件绘制前面已经写入的内容，可以防止绘制停顿。
- **D3D11_MAP_READ**：表示应用程序（CPU）会读取 GPU 缓冲的的一个副本到系统内存中。

**4．MapFlags**：可选标志，这里不使用，所以设置为 0；具体细节可参见 SDK 文档。

**5．pMappedResource**：返回一个指向 **D3D11_MAPPED_SUBRESOURCE** 的指针，这样我们就可以访问用于读/写的资源数据。

**D3D11_MAPPED_SUBRESOURCE** 结构定义如下：

```
typedef struct D3D11_MAPPED_SUBRESOURCE{
    void  *pData;
    UINT Row Pitch;
    UINT DepthPitch;
}D3D11_MAPPED_SUBRESOURCE
```

**1．pData**：指向用于读/写的资源内存的指针，你必须将它转换为资源中存储的数据的格式。

**2．RowPitch**：资源中一行数据的字节大小。例如，对于一个 2D 纹理来说，这个大小为一行的字节大小。

**3．DepthPitch**：资源中一页数据的大小。例如，对于一个 3D 纹理来说，这个大小为 3D 纹理中一个 2D 图像的字节大小。

**RowPitch** 和 **DepthPitch** 的区别是针对 2D 和 3D 资源（类似于 2D 和 3D 数组）而言的。顶点/索引缓冲本质上是 1D 数组，**RowPitch** 和 **DepthPitch** 的值是相同的，都等于顶点/索引缓冲的字节大小。

下面的代码展示如何在水波演示程序中更新顶点缓冲：

```
D3D11_MAPPED_SUBRESOURCE mappedData;
HR(md3dImmediateContext->Map(mWavesVB, 0, D3D11_MAP_WRITE_DISCARD,
0, &mappedData));

Vertex* v = reinterpret_cast<Vertex*>(mappedData.pData);
for(UINT i = 0; i < mWaves.VertexCount(); ++i)
{
    v[i].Pos = mWaves[i];
    v[i].Color = XMFLOAT4(0.0f, 0.0f, 0.0f, 1.0f);
}

md3dImmediateContext->Unmap(mWavesVB, 0);
```

当你完成缓冲区的更新操作之后，必须调用 **ID3D11Buffer::Unmap** 函数。

当使用动态缓冲区时，必然会有一些额外开销，因为这里存在一个从 CPU 内存向 GPU 内存回传数据的过程。 所以，在实际工作中应尽可能多使用静态缓冲区，少使用动态缓冲区。在 Direct3D 的最新版本中已经引入了一些新特性用于减少对动态缓冲区的需求。例如：

1．可以在顶点着色器中实现简单动画。

2．通过渲染到纹理（render to texture）和顶点纹理推送（vertex texture fetch）功能，可以实现完全运行在 GPU 上的水波模拟动画。

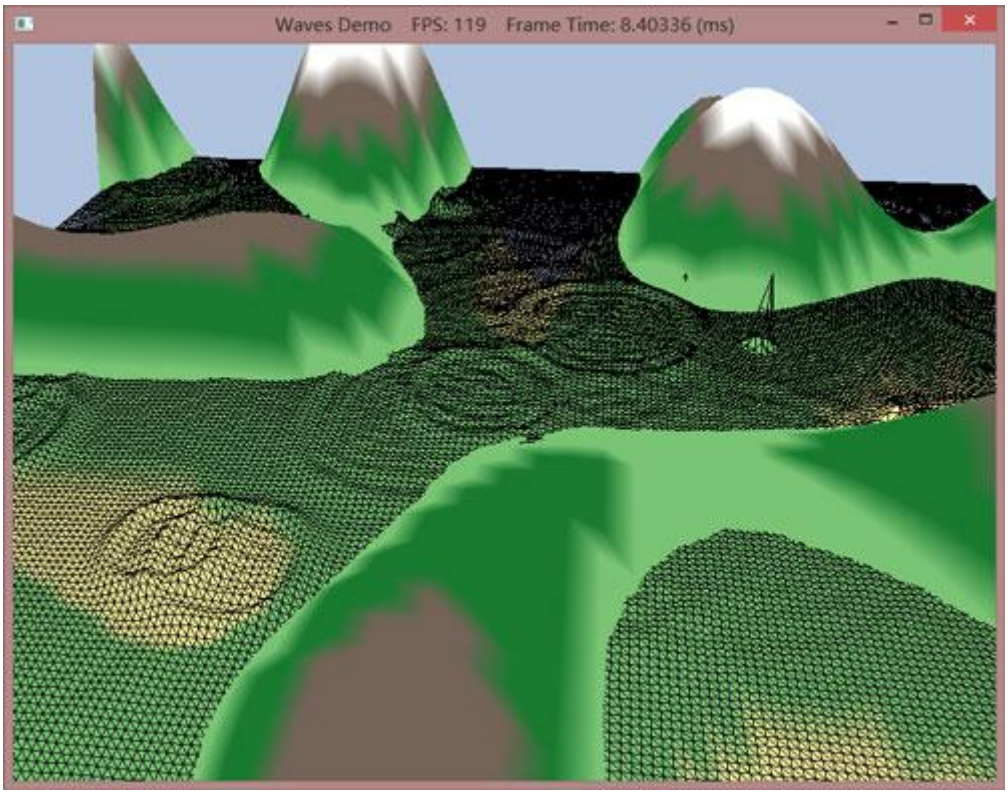
3．几何着色器为 GPU 提供了创建和销毁图元的能力，在以前没有几何着色器时，这些工作都是由 CPU 来完成的。

索引缓冲区可以是动态的。不过，在水波演示程序中，三角形的拓扑结构始终不变，只有顶点高度会发生变化；所以，这里只需要让顶点缓冲区变为动态缓冲区。

本章的水波演示程序使用了一个动态缓冲区来实现简单的水波效果。本书不会将重点放在水波模拟算法的实现细节上（有兴趣的读者可以参见[Lengyel02]），我们只是用它来说明动态缓冲区的用法：在 CPU 上更新模拟数据，然后调用 **Map/Unmap** 方法更新顶点缓冲区。

**注意**：在水波演示程序中，我们以线框模式渲染水波；是因为我们现在还没有讲到灯光的用法，在实体填充模式下，很难看出水波的运动效果。

**注意**：我们再次强调，这个示例应该在 GPU 上使用更高级的方式实现，比如渲染到纹理和顶点纹理推送。但是由于我们还没有讲到些技术，所以现在只能在 CPU 上实现，暂时使用动态顶点缓冲区来更新顶点。




**6.19 水波演示程序截图**

# 7.1 光照与材质的相互作用

图 7.1 展示了光照和阴影在表现物体的立体感和体积感时起到的重要作用。左边的球体没有灯照射，看上去就像是一个扁平的 2D 圆；而右边的球体有灯照射，看上去很立体。实际上，我们的视觉能力完全取决于光照及光照与材质之间的相互作用，所以，许多超写实场景的渲染工作都是通过精确的物理光照模型（lighting model）来实现的。（译者注：本章经常会提及“光照模型”个词，请读者不要将它与“3D 网格模型”的含义混淆。“光照模型”即不是说“用灯照射一个 3D 网格模型”，也不是指“支持光照运算的 3D 网格模型”。它的正确含义是“光照算法的数学模型（或者说数学公式）”。请读者一定要将“光照模型”与“3D 网格模型”的含义区分开。）

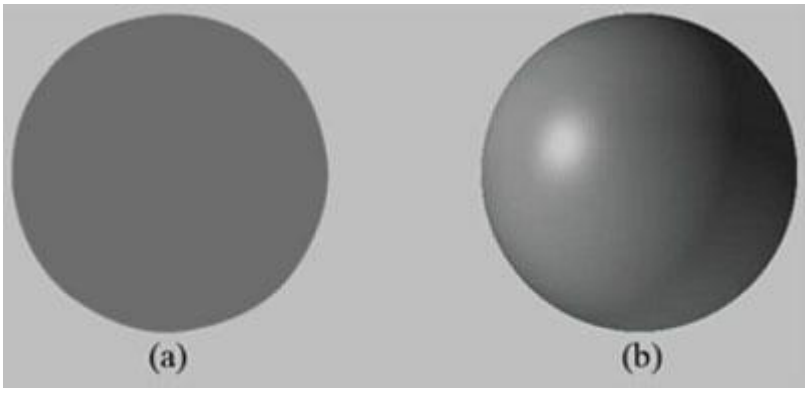



**图 7.1 (a)没有灯照明的球体看上去就像是一个 2D 圆。(b)有灯照明的球体看上去很立体。**

当然，一般来讲，越精确的光照模型，花费的计算时间就越长；我们必须在真实感和速

度之间寻求平衡。例如，电影中的 3D 特效场景可以做得非常复杂，可以使用更为写实的光照模型，因为电影中的帧是预渲染的（pre-rendered），电影制作者可以花费数小时甚至数日的时间来渲染一帧。而游戏是实时渲染应用程序，它至少要以每秒 30 帧的速度渲染场景。

注意，本书关于光照模型的解释和实现方法大部分来自于[Möller02]的描述。

# 学习目标

1．了解光照与材质之间的相互作用。

2．了解局部照明和全局照明之间的区别。

3．了解如何以数学方式描述平面上的点所“面对”的方向，以使我们确定线与平面之间的入射角。

4．学习如何正确变换法线向量。

5．了解环境光、漫反射和镜面光之间的区别。

6．学习如何实现方向光、点光和聚光灯。

7．了解如何通过深度来控制衰减参数，改变光照的强度。

当使用光照时，我们不再直接指定顶点颜色；而是指定材质和灯光，然后使用光照方程，根据灯光与材质的相互作用计算顶点颜色。这样可以产生非常逼真的物体颜色（再次比较图 7.1a 和 7.1b 中的球体）。

材质可以被认为是决定光照如何与物体表面相互作用的属性。例如，表面反射的灯光颜色、吸收的灯光颜色、反射率、透明度和光泽度都是构成表面材质的参数。不过，在本章中，我们主要讲解的是表面反射的灯光颜色、吸收的灯光颜色和光泽度。

在我们的光照模型中，光源可以发射各种强度的红、绿、蓝光；通过一方式，我们可以模拟很多灯光颜色。当光线从光源发出照射到一个物体上时，一部分光线会被物体吸收，另一部分线会被反射回来（对于透明物体，比如玻璃，还会有一部分光线会从物体中间穿过，不过在这里我们先不用考虑透明度的问题）。反射会沿着它的新路径传播，可能会照射在其他物体上，其中一部分线会被物体吸收，另一部分线会再次反射。在光线的能量完全耗尽之前，它会照射到许多物体。很可能会有一部分线最终传入人的眼睛（参见图 7.2），触碰到视网膜上的视感细胞（称为视锥细胞和视杆细胞）。

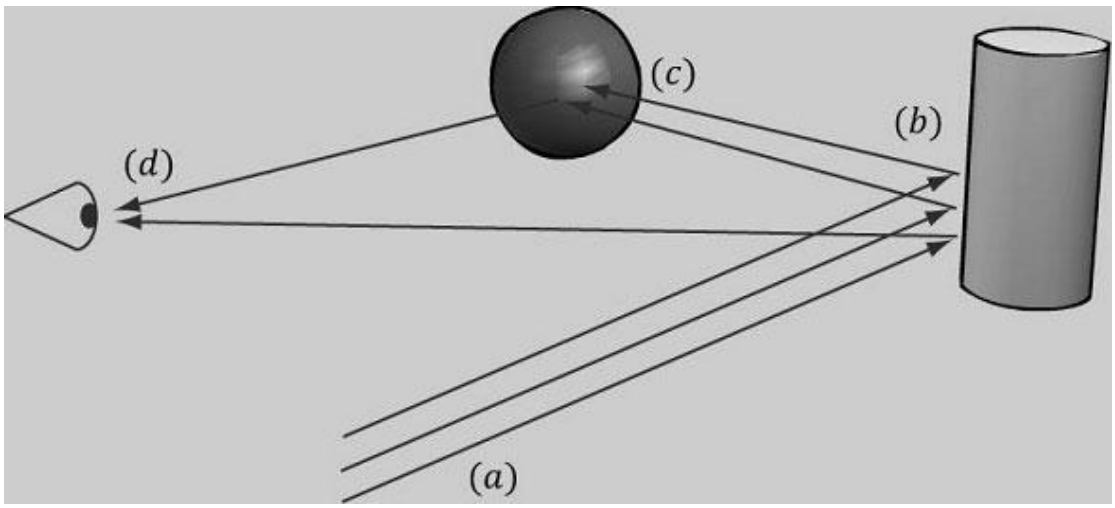



**图 7.2 (a)连续射入的白色光线。(b)当光线照射到圆柱体上时，一部分光线会被圆柱体吸收，另一部分光线分散传向眼睛和球体。(c)当圆柱体的反射光照射到球体上时，一部分光线会**

**被球体吸收，另一部分光线会再次反射，传入眼睛。(d)眼睛收到入射线，看到物体。**

根据三原色理论（参见[Santrock03]），视网膜包含三种类型的有色感受器，分别对红、绿、蓝光（以及某些重叠部分）敏感。根据光的波长改变射入的 RGB 光线强度，刺激相应的感受器。这样，感受器就会受到刺激（或者不受刺激），神经触突会通过视觉神经传送到大脑，大脑根据感受器产生的信号形成头脑中的最终图像。（当然，如果你闭上或盖上眼睛，感受器细胞就不会受到刺激，大脑就会认为是黑色。）

例如，再次考虑图 7.2。假设圆柱体的材质反射 75%的红和 75%的绿光，其余线均被圆柱体吸收；球体反射 25%的红光，其余线均被球体吸收。同时，假设光源发射的线为纯白色光线。当光线照射到圆柱体上时，所有的蓝会被吸收，只有 75%的红和 75%的绿光被反射回来（即，中高强度的黄色线）。这些光线会产生散射，其中一部分光线会传入眼睛，另一部分线会传向球体。传进眼睛的那一部分线主要刺激的是红色和蓝色圆锥细胞；因此，观察者看到的圆柱体为亮黄色。现在，另一部分线会传向球体，并照射在球体表面上。球体反射 25%的红光，其余线均被球体吸收；那些曾被稀释过的红色（中高强度的红色线）会被再次稀释，所有射入的绿光均被吸收，只有一部分红光反射回来。然后些剩下的红光传入眼睛，对红色视锥细胞产生刺激。因此，观察者看到的球体为暗红色。

本书（和许多实时渲染应用程序）采用的光照模型都是局部光照模型（local illumination model）。在局部光照模型中，每个物体的光照都是相互独立的，所有的光线都是从光源直接照射到物体上（即，本来应该被场景中其他物体阻隔的线，仍然会照射到后面的物体上）。图 7.3 说明了一光照模型的特点。

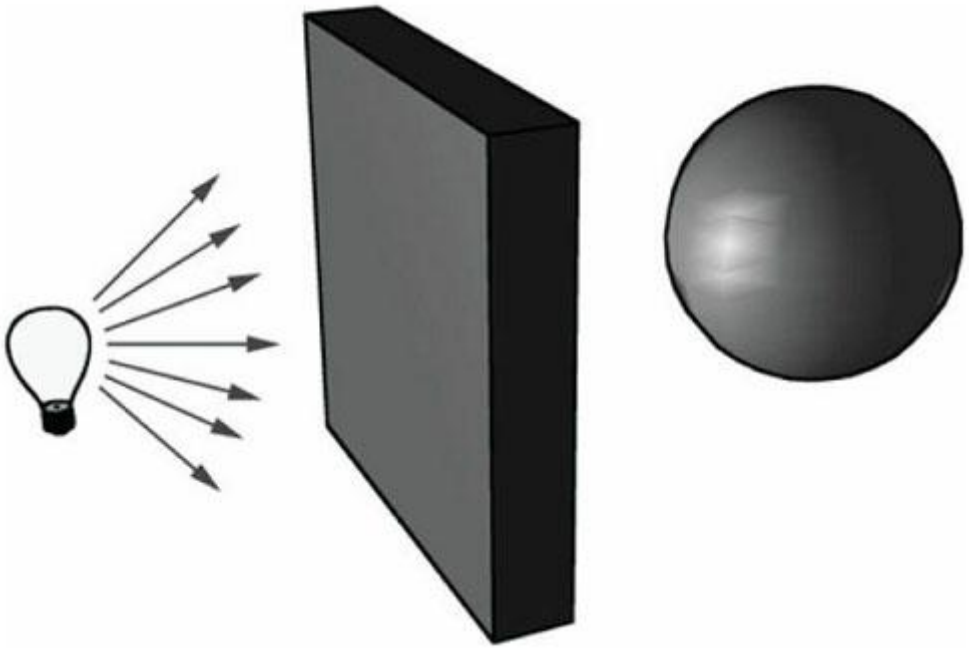


**图 7.3 在现实环境中，如果墙壁挡住了从灯泡射出的光线，那么球体就应该位于墙壁的阴影之中。然而，在局部光照模型中，无论墙壁是否存在，球体都会被照亮。**

与之相反，全局照明模型（global illumination model）不仅会考虑光源对物体的直接照明，也会考虑场景中的其他物体反射造成的间接照明。它之所以被称为全局照明模型，就是因为它在照亮一个物体的同时，也会考虑到整个场景中的其他因素。对于实时游戏来讲，全局照明模型占用的系统资源量过大（虽然它可以渲染出接近于照片的超写实场景）。目前，实时全局照明（real-time global illumination）的实现方法还处于探索阶段。

# 7.2 法线向量

平面法线（face normal）是描述多边形所朝方向的单位向量（即，它与多边形上的所有点相互垂直），如图 7.4a 所示。表面法线（surface normal）是与物体表面上的点的正切平面（tangent plane）相互垂直的单位向量，如图 7.4b 所示。表面法线确定了表面上的点“面对”的方向。

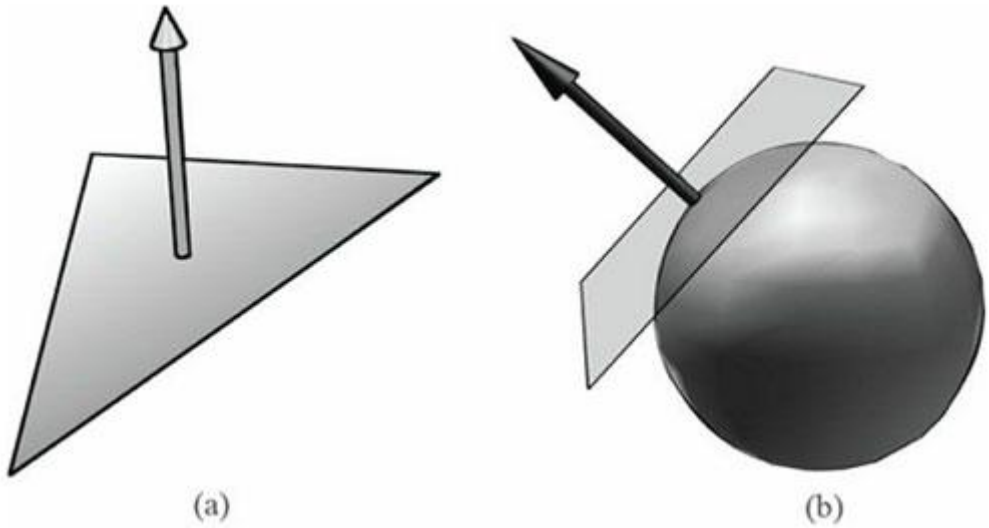



**图 7.4 (a)平面法线与平面上的所有点相互垂直。(b)表面法线与物体表面上的点的正切平面相互垂直。**

当进行光照计算时，我们必须为三角形网格表面上的每个点求解表面法线，以确定光线与网格表面在该点位置上的入射角度。为了获得表面法线，我们必须为每个顶点指定表面法线（这些法线称为顶点法线）。然后在光栅化阶段，这些顶点法线会在三角形表面上进行线性插值，使三角形表面上的每个点都获得一个表面法线（回顾 5.10.3 节并参见图 7.5）。

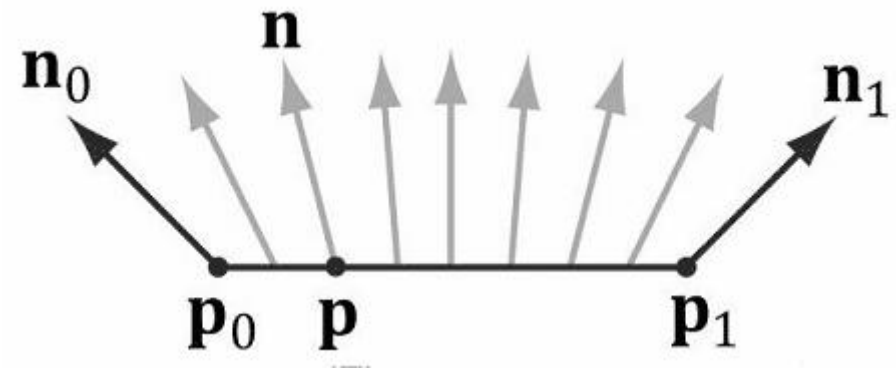



图 7.5 $\mathbf{p}_0$ 和 $\mathbf{p}_1$ 是线段的两个顶点，$\mathbf{n}_0$ 和 $\mathbf{n}_1$ 是对应的顶点法线。点 $\mathbf{p}$ 是通过线性插值（加权平均值）得到的线段上的一点，$\mathbf{n}$ 是点 $\mathbf{p}$ 的法线向量，它介于两个顶点法线之间。也就是，当存在一个位置使 $\mathbf{p} = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0)$ 时，$\mathbf{n} = \mathbf{n}_0 + t(\mathbf{n}_1 - \mathbf{n}_0)$。为了简单起见，我们只解释了线段的法线插值，但是一概念可以被直接扩展为 3D 三角形的法线插值。

**注意：**对每个像素的法线进行插值，并进行光照计算称为逐像素光照或 phong 光照。还有一种负担较轻，但不够精确的方法是对每个顶点进行光照运算，称为逐顶点光照，计算结果是从顶点着色器中输出的，在像素着色器中进行插值。将计算从像素着色器转移到顶点着色器是一种常见的性能优化措施，而且在很多情况下的视觉表现与逐像素光照差别不大。

## 7.2.1 计算法线向量

为了求解一个三角形 $\Delta \mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$ 的平面法线，我们必须先计算该三角形边上的两个向量：

$\mathbf{u} = \mathbf{p}_1 - \mathbf{p}_0$

$\mathbf{v} = \mathbf{p}_2 - \mathbf{p}_0$

然后求得平面法线为：

$$\mathbf{n} = \frac{\mathbf{u} \times \mathbf{v}}{\|\mathbf{u} \times \mathbf{v}\|}$$

下面的函数可以根据三角形的 3 个顶点来计算三角形正面（参见 5.10.2 节）的平面法线。

```
void ComputeNormal(const XMVector3& p0,
    const XMVector3& p1,
    const XMVector3& p2,
    XMVector3& out)
{
    XMVector3 u = p1 - p0;
    XMVector3 v = p2 - p0;
    XMVector3Cross(&out, &u, &v);
    XMVector3Normalize (&out, &out);
}
```

对于一个微分曲面来说，我们可以使用微积分计算曲面上的点的法线。但遗憾的是，三角形网格不是微分曲面。我们通常使用一种称为顶点法线平均值（vertex normal averaging）的技术求解三角形网格上的顶点法线。对于网格上的任意顶点 **v** 来说，**v** 的顶点法线 **n** 等于以 **v** 为共享顶点的每个多边形的平面法线的平均值。例如在图 7.6 中，网格上的四个多边形共享顶点 **v**；所以，**v** 的顶点法线为：

$$\mathbf{n}_{avg} = \frac{\mathbf{n}_0 + \mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3}{\|\mathbf{n}_0 + \mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3\|}$$

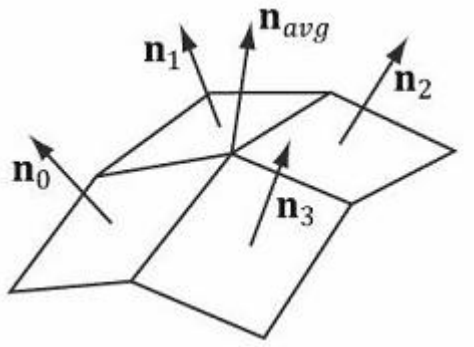



**图 7.6 中间的顶点由相邻的 4 个多边形共享，我们通过计算这 4 个多边形平面法线的平均值就可以估算出该顶点的法线。**

在上面的例子中，我们不需要除以 4，因为我们想要的是一个普通平均值，我们可以对结果进行规范化。注意，我们还可以构造更巧妙的平均值计算公式；例如，以每个多边形的面积作为权值，计算加权平均值（这样，面积较大的多边形会占有较大的权重，而面积较小的多边形会占有较小的权重）。

下面的伪代码说明了在给出一个三角形网格的顶点列表和索引列表时，如何计算该平均值：

```
// 输入：
// 1.一个顶点数组(mVertices)，每个顶点都有一个位置分量(pos)和
// 一个法线分量(normal)。
// 2.一个索引数组(mIndices)。
// 处理网格中的每个三角形：
for(DWORD i = 0; i < mNumTriangles; ++i)
{
    // 第 i 个三角形的索引
    UNIT i0 = mIndices[i*3+0];
    UNIT i1 = mIndices[i*3+1];
    UNIT i2 = mIndices[i*3+2];
    // 第 i 个三角形的顶点
    Vertex v0 = mVertices[i0];
    Vertex v1 = mVertices[i1];
    Vertex v2 = mVertices[i2];
    // 计算面法线
    Vector3 e0 = v1.pos - v0.pos;
    Vector3 e1 = v2.pos - v0.pos;
    Vector3 faceNormal Cross( &e0, &e1);
    // 这个三角形共享了以下三个顶点，
    // 所以要将面法线加入到这些顶点法线的平均值中。
    mVertices[i0].normal += faceNormal;
    mVertices[i1].normal += faceNormal;
    mVertices[i2].normal += faceNormal;
}
// 对每个顶点 v，我们已经将共享 v 的所有三角形的面法线相加了，
// 所以现在只需归一化即可。
for(UNIT i = 0; i < mNumVertices; ++i)
    mVertices[i].normal =  Normalize(&mVertices[i].normal));
```

## 7.2.2 对法线向量进行变换

在图 7.7a 中，正切向量 $\mathbf{u} = \mathbf{v}_1 - \mathbf{v}_0$ 垂直于法线向量 $\mathbf{n}$。当我们对这两个向量应用一个不成比例的缩放变换 $\mathbf{A}$ 时，我们可以从图 7.7b 中看到，变换之后的切线向量 $\mathbf{uA} = \mathbf{v}_1\mathbf{A} - \mathbf{v}_0\mathbf{A}$ 不再垂直于变换之后的法线向量 $\mathbf{nA}$。

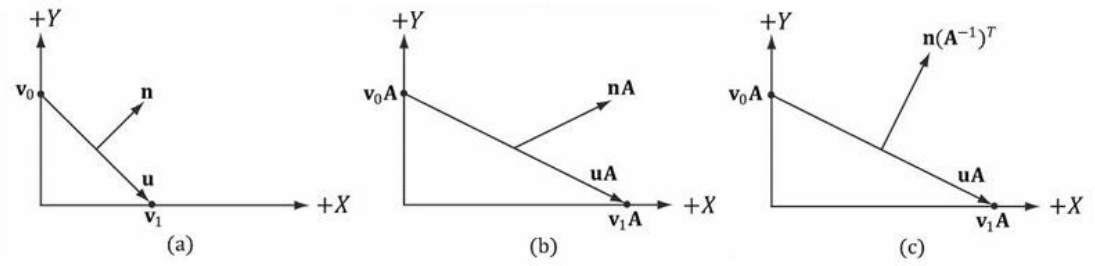



**图 7.7 (a)变换之前的表面法线。(b)当 *x* 轴上的单位长度增大两倍后，法线不再垂直于表面。(c)通过计算缩放变换的逆转置矩阵，我们可以得到正确的变换结果。**

所以我们的问题是：当给出一个用于变换点和（非法线）向量的变换矩阵 **A** 时，如何求出一个专门用来变换法线向量的变换矩阵 **B**，使变换之后的切线向量和法线向量依然保持垂直关系（即 **uA** • **uB** = 0）。要解决一问题，让我们先从一些已知条件开始：我们知道法线向量 **n** 直于切线向量 **u**。

| | |
|---|---|
| **u** • **v** = 0 | 切线向量垂直于法线向量 |
| $\mathbf{u}\mathbf{n}^T = 0$ | 用矩阵乘法来表示点积 |
| $\mathbf{u}(\mathbf{A}\mathbf{A}^{-1})\mathbf{n}^T = 0$ | 插入单位矩阵 $\mathbf{I} = \mathbf{A}\mathbf{A}^{-1}$ |
| $(\mathbf{uA})(\mathbf{A}^{-1}\mathbf{n}^T) = 0$ | 矩阵乘法的结合性 |
| $(\mathbf{uA})((\mathbf{A}^{-1}\mathbf{n}^T)^T)^T = 0$ | 转置特性$(\mathbf{A}^T)^T = \mathbf{A}$ |
| $(\mathbf{uA})(\mathbf{n}(\mathbf{A}^{-1})^T)^T = 0$ | 转置特性$(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$ |
| $\mathbf{uA} \cdot \mathbf{n}(\mathbf{A}^{-1})^T = 0$ | 用点积来表示矩阵乘法 |
| **uA** • **nB** = 0 | 变换之后的切线向量垂直于变换之后的法线向量 |

这样，使用 $\mathbf{B} = (\mathbf{A}^{-1})^T$（**A** 的逆转置矩阵）来变换法线向量，就可以使它与变换之后的切线向量依然保持垂直关系。

注意，当变换矩阵为正交矩阵（$\mathbf{A}^T = \mathbf{A}^{-1}$）时，$\mathbf{B} = (\mathbf{A}^{-1})^T = (\mathbf{A}^T)^T = \mathbf{A}$；也就是，我们不必计算逆转置矩阵，直接用 **A** 来代替 **B** 即可。总之，当以一个非等比变换矩阵对法线向量进行变换时，我们必须使用该矩阵的逆转置矩阵。

在 **MathHelper.h** 中有一个辅助方法用于计算逆转置矩阵：

```
static XMMATRIX InverseTranspose(CXMMATRIX M)
{
    // 逆转置矩阵仅用于法线。所以将矩阵中的平移行
    // 设置为 0，这样就不会影响逆转置矩阵的计算——因为我们
    // 不需要对平移进行逆转置运算。
    XMMATRIX A = M;
    A.r[3] = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);

    XMVECTOR det = XMMatrixDeterminant(A);
    return XMMatrixTranspose(XMMatrixInverse(&det, A));
}
```

因为逆转置只用于变换矢量，而平移是作用在点上的，因此需要从矩阵中排除平移因素。但是，3.2.1 节告诉我们，为了防止矢量被平移操作影响，它的 w 应设置为 0（使用齐次坐标）。因此，我们无须将矩阵中的平移行归零。但问题是，如果我们将逆转置矩阵和另一个不包含非等比例缩放的矩阵相乘，例如视矩阵$(\mathbf{A}^{-1})^T\mathbf{V}$，转置后位于$(\mathbf{A}^{-1})^T$ 第 4 列的平移项导致结果错误。所以，我们将平移项清零就是为了预防这个错误。正确的方法是使用$((\mathbf{AV})^{-1})^T$对法线进行变换。下面是一个缩放和平移矩阵的例子，第 4 列经过逆转置后并不是$[0,0,0,1]^T$：

$$\mathbf{A}=\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 0.5 & 0 & 0\\ 0 & 0 & 0.5 & 0\\ 1 & 1 & 1 & 1\end{pmatrix}$$

$$(\mathbf{A}^{-1})^{T}=\begin{pmatrix}1 & 0 & 0 & -1\\ 0 & 2 & 0 & -2\\ 0 & 0 & 2 & -2\\ 0 & 0 & 0 & 1\end{pmatrix}$$

**注意**：即使使用逆转置变换，法线向量也有可能会失去单位长度；所以，在变换之后必须重新规范化法线向量。

# 7.3 兰伯特余弦定理

垂直照向平面的线比从侧面照向平面的线更加强烈（参见图 7.8）。

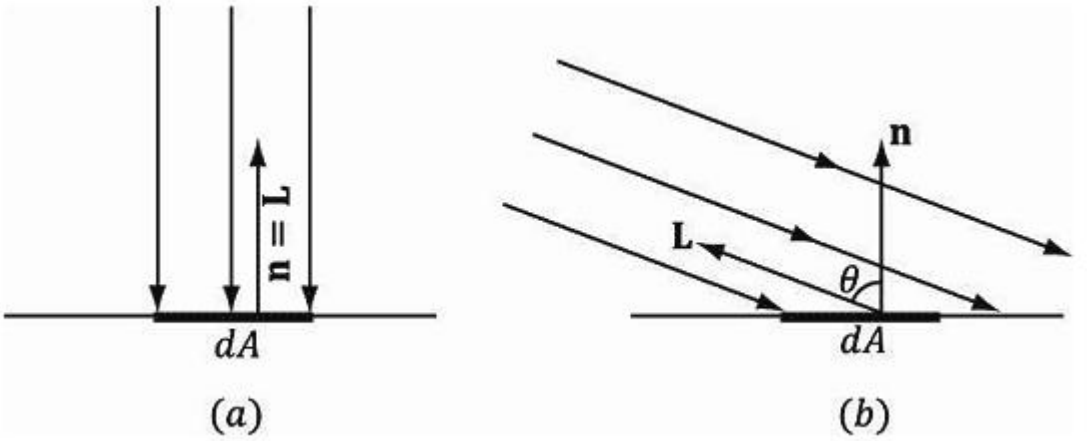



**图 7.8 假设有一块很小的区域 *dA*。当法线向量 n 与光照向量 L 平行时，区域 *dA* 受到的光线照射最多。随着 n 和 L 之间的夹角$\theta$逐渐增大，区域 *dA* 受到的光线照射量会越来越少（因为很多光线都无法照射到 *dA* 表面上了）。**

我们可以从这个概念中推导出一个函数，根据顶点法线和光照向量之间的夹角返回不同的光照强度。（注意，光照向量是从表面指向光源的向量；也就是，它与线的传播方向正好相反。）当顶点法线与照向量完全重叠时（即，它们的角度为 0º 时），该函数返回最大强度值；随着顶点法线与照向量之间的夹角逐渐增大，该函数返回的强度值会越来越小。当$\theta$>90º 时，说明光线照射的是物体背面，此时我们应该将强度设置为 0。兰伯特（Lambert）余弦定理给出了上述函数的定义：

$$f(\theta)=\max(\cos\theta,0)=\max(\mathbf{L}\cdot\mathbf{n},0)$$

其中，**L** 和 **n** 是单位向量。图 7.9 是$f(\theta)$的曲线图。我们可以看到，随着$\theta$的变化，强度在 0.0 到 1.0（即，0%到 100%）之间变化。

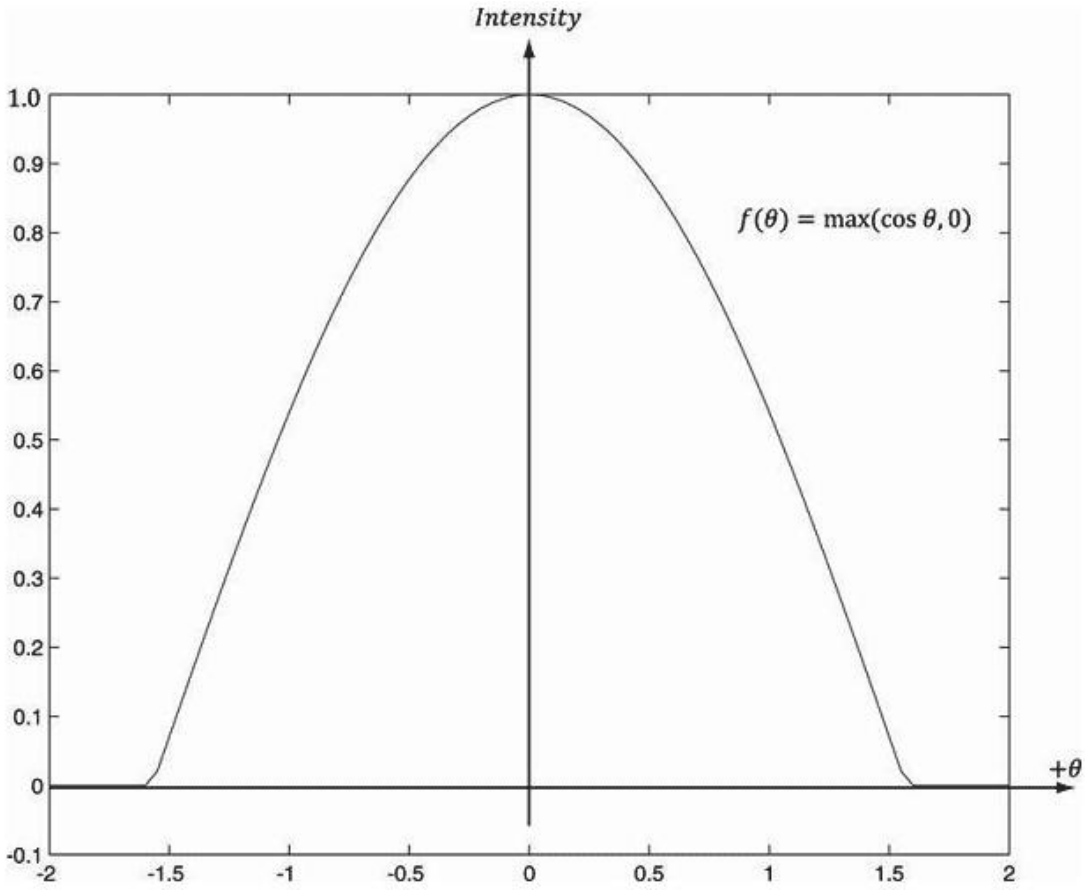



**图 7.9 当−2≤θ≤ 2 时，函数 $f(\theta) = \max(\cos\theta,0) = \max(L \cdot n,0)$的曲线图。注意，$\pi/2 \approx 1.57$。**

# 7.4 漫反射光

考虑图 7.10 所示的粗糙表面。当光线照射在这样一个表面上时，会在不同的随机方向上散开；我们将这种反射称为漫反射（diffuse reflection）。在我们的光照模型中模拟了灯光与表面之间的这种相互作用，我们约定线会在表面的各个方向上均匀散开；所以，无论观察点（眼睛的位置）在哪里，我们总能看到漫反射光。也就是，我们不需要考虑观察点的位置（即，漫反射光的计算与观察点的位置无关），无论观察点在哪里，我们总能看到表面上的点的颜色。

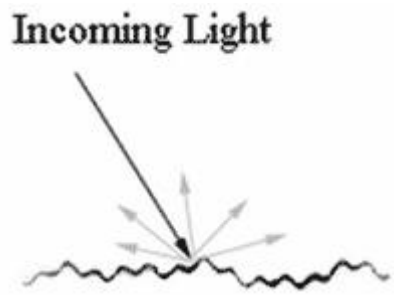



**图 7.10 当照射一个漫反射表面时，入射光会在随机方向上散开。在显微镜下，很多物体表面都是粗糙的。**

我们将漫反射光的计算过程分为两个部分。在第一部分中，我们指定一个入射光颜色和

一个漫反射材质颜色。漫反射材质指定了表面所能反射和吸收的漫反射光的总量；它使用分量颜色乘法来实现。例如，表面反射 50%的红光、100%的绿和 75%的蓝光，入射光是强度为 80%的白光。那么，入射光颜色 $\mathbf{l}_d$ = (0.8, 0.8, 0.8)，漫反射材质颜色 $\mathbf{m}_d$ = (0.5, 1.0, 0.75)；反射光的总量为：

$$\mathbf{D} = \mathbf{l}_d \otimes \mathbf{m}_d = (0.8, 0.8, 0.8) \otimes (0.5, 1.0, 0.75) = (0.4, 0.8, 0.6)$$

**注意**：表面上的漫反射材质可以变化；也就是，表面上的每个点可以有不同的漫反射材质颜色。例如，当渲染一个由沙滩、草地、土地和雪地构成的地形时，地形的每个部分反射和吸收的线数量是不一样的，所以材质颜色必须根据地形表面而变化。在本书的 Direct3D 光照实现中，我们为每个顶点指定一个不同的漫反射材质颜色。在光栅化阶段中，这些顶点属性会在三角形表面上的进行插值。通过一方式，三角形网格表面上的每个点都会得到一个漫反射材质颜色。

我们直接引用兰伯特余弦定理（根据表面法线与光照向量之间的夹角来控制表面的受光强度）来实现漫反射光照计算。设 $\mathbf{l}_d$ 入射光颜色，$\mathbf{m}_d$ 为漫反射材质颜色，$k_d = \max(\mathbf{L}\cdot\mathbf{n}, 0)$，其中 **L** 是光线向量，**n** 是表面法线。则，反射回来的漫反射光颜色为：

$$\mathbf{c}_d = k_d \cdot \mathbf{l}_d \otimes \mathbf{m}_d = k_d\mathbf{D} \qquad \text{（公式 7.1）}$$

# 7.5 环境光

前面提到，我们的光照模型不处理由场景中的其他物体反弹的间接光。不过，我们在现实生活中看到的很多光线都是间接光。例如，在一条通往房间的走廊里，我们不会直接看到房间里面的光源，但是光线会照射在墙壁上，通过墙壁把一部分线反弹到走廊中，间接地把走廊照亮。再比如，我们坐在一间屋子里，面前摆着桌子、茶壶和台灯。茶壶放在桌子上，并且只有一个侧面面向台灯；我们可以看到茶壶背面并不是全黑的。因为有一些光线会通过墙壁或其他物体间接地反弹到茶壶背面。

为了模拟间接光，我们在光照方程中引入了一个环境光项：

$$\mathbf{A} = \mathbf{l}_a \otimes \mathbf{m}_a$$

颜色 $\mathbf{l}_a$ 指定了表面从一个光源收到的间接（环境）光的总量。环境材质颜色 $\mathbf{m}_a$ 指定了表面反射和吸收的入射环境光的总量。环境光只是将物体的亮度稍微提高一点儿——这根本不是真实的物理计算。之所以这样做是因为间接光会在场景中散开和多次反弹，从每个方向均匀地照亮物体。

将环境项与漫反射项组合在一起，得到新的光照方程：

$$\text{LitColor} = \mathbf{l}_a \otimes \mathbf{m}_a + \mathbf{k}_d \cdot \mathbf{l}_d \otimes \mathbf{m}_d \qquad \text{（公式 7.2）}$$

# 7.6 镜面光

考虑图 7.11 所示的光滑表面。当灯照射在这样一个表面上时，光线会在一个由反射系数描述的圆锥体区域内形成锐利的反射；我们将这种反射称为镜面高光反射（specular reflection，或直译为镜面反射）。与漫反射不同，高光可能不会传入眼睛，因为它只在一个特定的方向上反射；高光的计算过程与观察点的位置相关。也就是说，当场景中的观察点位置发生变化时，我们看到的高光强度也会跟着变化。

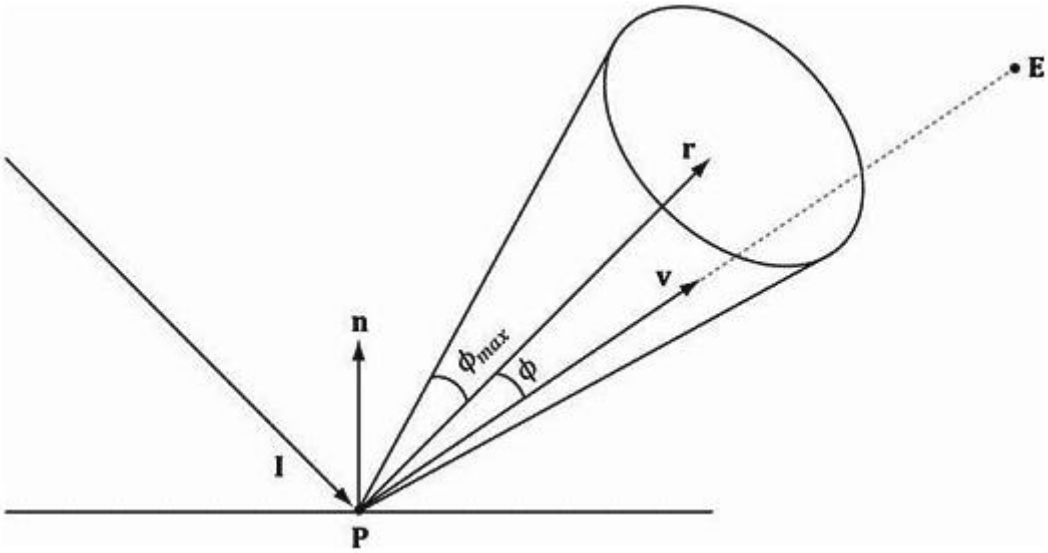



**图 7.11 镜面反射不会在所有的方向上散开，它只会将反射光集中在一个由反射系数描述的圆锥体区域内。当 v 在圆锥体内时，我们可以看到高光；反之，看不到高光。v 与 r 的夹角越小，我们看到的高光就越强。**

镜面反射的圆锥体区域由一个反射向量 **r** 和一个角度$\phi_{\max}$来定义。简单来说，反射光的强度可由反射向量 **r** 和观察向量 $\mathbf{v}=\dfrac{\mathbf{E}-\mathbf{P}}{\|\mathbf{E}-\mathbf{P}\|}$（即，从表面点 P 到观察点位置 **E** 的单位向量）之间的夹角$\phi$来决定。我们约定：当$\phi$ =0 时，高光强度最大；当$\phi$逐渐接近于时，高光强度逐渐降低为 0。为了以数学方式来描述一过程，我们需要对使用兰伯特余弦定理的函数做一些修改。图 7.12 说明了使用不同幂时的余弦函数曲线图，其中 $p \geqslant 1$。本质上，通过为 $p$ 指定不同的值，我们可以间接地控制当高光强度降低为 0 时的圆锥体角度$\phi_{\max}$。参数 $p$ 可以用来控制表面的平滑程度；也就是，非常精细的表面比缺乏光泽的表面的反射系数小（反射光更锐利）。所以，我们应该为光滑表面指定一个比不光滑表面更大的 $p$ 值。

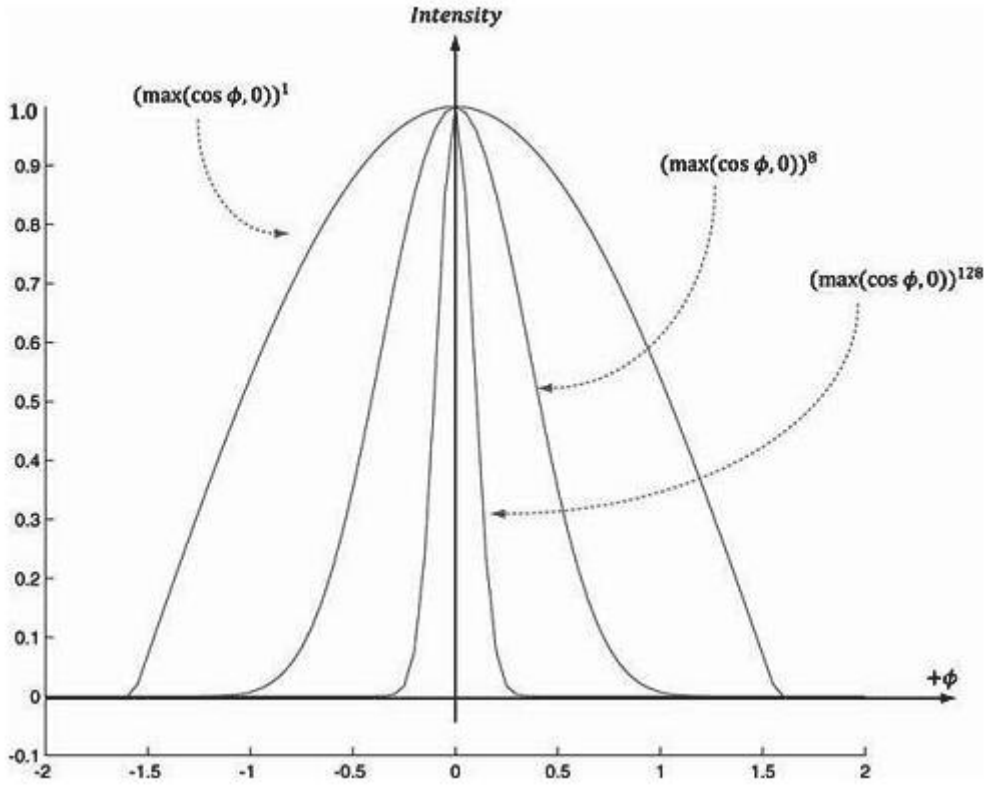



**图 7.12 使用不同幂时的余弦函数曲线图，其中 p≥1。**

注意，因为 v 和 r 是单位向量，所以 $\cos\phi = \mathbf{v} \cdot \mathbf{r}$。

我们现在定义照模型中的高光项：

$\mathbf{c}_s = k_s \cdot \mathbf{l}_s \otimes \mathbf{m}_s = k_s \mathbf{S}$

其中

$$k_s = \begin{cases} \max(\mathbf{v} \cdot \mathbf{r}, 0)^P, \mathbf{L} \cdot \mathbf{n} > 0 \\ 0, \mathbf{L} \cdot \mathbf{n} \le 0 \end{cases}$$

颜色 $\mathbf{l}_s$ 指定了光源发出的高光总量。镜面材质颜色 $\mathbf{m}_s$ 指定了表面反射和吸收的入射高光总量。系数 $k_s$ 根据 **r** 和 **v** 之间的夹角来决定高光强度。图 7.13 说明了一个表面可能接收不到漫反射光（$\mathbf{L} \cdot \mathbf{n} < 0$），但是却可以接收到高光。不过，在这种情况下，它收到的高光是毫无意义的，我们应该将 $k_s$ 设为 0。

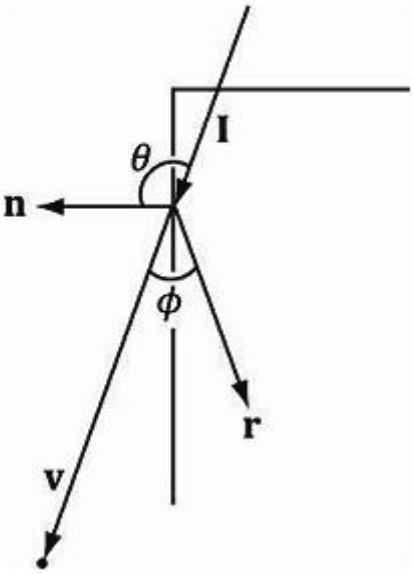



**图 7.13 虽然光线照射的是物体背面，但是在观察点上还是可以看到高光。是错误的结果。当出现一问题时，我们必须将 $k_s$ 设为 0。**

注意：高光幂 $p$ 的取值应该总是大于 1 的。

新的光照模型为：

$\text{LitColor} = \mathbf{l}_a \otimes \mathbf{m}_a + \mathbf{k}_d \cdot \mathbf{l}_d \otimes \mathbf{m}_d + k_s \cdot \mathbf{l}_s \otimes \mathbf{m}_s = \mathbf{A} + k_d\mathbf{D} + k_s\mathbf{S}$ （公式 7.3）

$k_d = \max(\mathbf{L}\cdot\mathbf{n}, 0)$

$$k_s = \begin{cases} \max(\mathbf{v}\cdot\mathbf{r}, 0)^P, \mathbf{L}\cdot\mathbf{n} > 0 \\ 0, \mathbf{L}\cdot\mathbf{n} \le 0 \end{cases}$$

**注意**：反射向量 $\mathbf{r} = \mathbf{I} - 2(\mathbf{n}\cdot\mathbf{I})\mathbf{n}$（参见图 7.14）。（这里假设 n 是一个单位向量。）不过，我们在着色器程序中总是使用 HLSL 的内置函数 **reflect** 来计算。

这里的入射光向量 **I** 是指入射光的方向（它与光线向量 **L** 的方向相反） 。

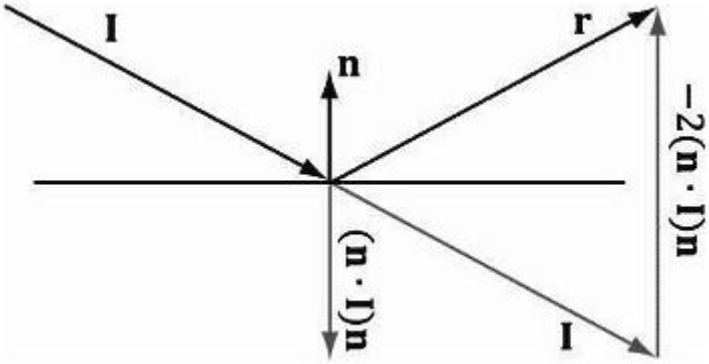



**图 7.14 反射的几何描述。**

# 7.7 重点摘要

在我们的模型中，光源可以发射 3 种不同类型的线：

1．环境光（ambient light）：模拟间接光照。

2．漫反射光（diffuse light）：模拟对粗糙表面的直接照。

3．高光（specular light）：模拟对光滑表面的直接光照。

同样，物体表面有以下材质属性与其对应：

1．环境材质：平面反射和吸收的环境光的总量。

2．漫反射材质：平面反射和吸收的漫反射光的总量。

3．高光材质：平面反射和吸收的高光的总量。

4．高光指数：它是在高光计算中使用的一个指数，它通过一个由反射系数描述的圆锥体区域来控制表面的光滑程度。圆锥体越小，表面越平滑/光亮。

把光照分为 3 个部分的原因是为了提高灵活性；可以让美术师从多个自由度来调整希望得到的渲染结果。图 7.15 说明了如何将这三个部分结合在一起使用。

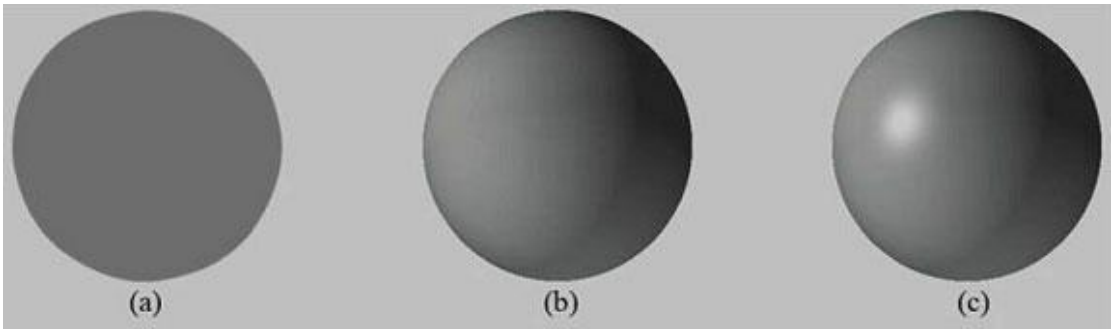



**图 7.15 (a)只有环境光的球体颜色，环境光只是均匀地提高物体的亮度。(b)环境和漫反射光的组合。兰伯特余弦定理使球体表面形成了从亮到暗的平滑过渡。(c)环境光、漫反射和高光的组合。高光在球体的受光面形成了一小块高亮区域。**

# 7.8 指定材质

我们如何指定材质的值？表面上的材质有可能会发生变化；也就是说，表面上不同的点可能会有不同的材质值（见图 7.16）。例如，一个轿车模型的车身、窗户、灯和轮胎反射和吸收光线的能力是不一样的，所以轿车表面的材质值也应该不一样。

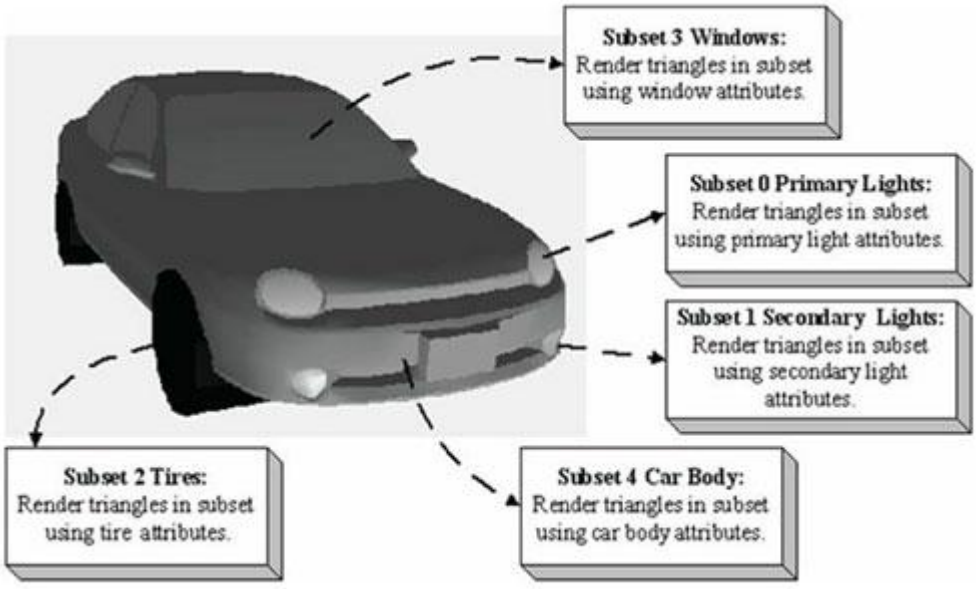



**图 7.16 将轿车网格分为 5 个材质属性组。**

要模拟材质值的不同，一种方法是在顶点级别上定义材质值。这些材质值会在三角形表

面进行线性插值，使三角形网格的每个表面点都拥有材质值。但是，从第 6 章中的“山峰与河谷演示程序”中可以看到，在顶点级别定义材质颜色模拟出的效果还是太粗糙。而且，顶点颜色还会在顶点结构中添加额外的数据，我们还需要给每个顶点上色的工具。更普遍的方法是使用纹理映射，将会在下一章中介绍。还有，在频繁调用绘制的过程中我们还要修改材质。因此，我们将材质值设置为常量缓冲的一个成员，除非在两次绘制之间改变了这个材质值，所有在设置之后绘制的几何体都会使用这个材质。下面的伪代码展示了如何绘制一辆轿车：

```
Set Primary  Lights material to constant buffer
Draw Primary  Lights geometry
Set Secondary Lights material to constant buffer
Draw Secondary Lights geometry
Set Tire material to constant buffer
Draw Tire geometry
Set Window material to constant buffer
Draw Windows geometry
Set Car Body material to constant buffer
Draw car body geometry
```

我们的材质结构体定义如下，位于 **LightHelper.h** 中：

```
struct Material
{
    Material(){ ZeroMemory(this,sizeof(this));}
    XMFLOAT4 Ambient;
    XMFLOAT4 Diffuse ;
    XMFLOAT4 Specular;  //  w 分量为高光强度
    XMFLOAT4 Reflect;
} ;
```

这里不讨论 **Reflect** 成员变量，这个变量会在以后模拟镜子时用到。我们在镜面高光指数 $p$ 放置在材质高光颜色的第 4 个分量中。这是因为光照不需要 alpha 分量，所以空出的这个位置可以储存一些有用的东西。漫反射材质的 alpha 分量在后面的章节中可用于 alpha 混合。

最后，我们提醒读者，三角形网格表面上的每个点都需要法线向量，以使网格表面上的每个点都可以（根据兰伯特余弦定理）计算入射光的角度。为了在三角形网格表面上估算每个点的法线向量，我们在顶点级别上指定法线。在光栅化阶段中，这些法线会在三角形表面上进行线性插值。

到目前为止，我们已经讨论了光线的组成原理，但是还没有讨论特定的光源类型。在下面的三节中，我们将讲解平行光（parallel light ）、点光（point light）和聚光灯（spotlight）的实现方法。

# 7.9 平行光

平行光（或方向光）用于模拟距离很远的光源，它产生的入射光是相互平行的（参见图 7.17）。平行光由一个描述线传播方向的向量来表示。因为它产生的线是平行的，所以所有

光线都使用相同的方向向量。光照向量与平行光的传播方向相反。在现实生活中，最常见的平行光源是太阳（参见图 7.18）。用于平行光的光照方程为方程 7.3。

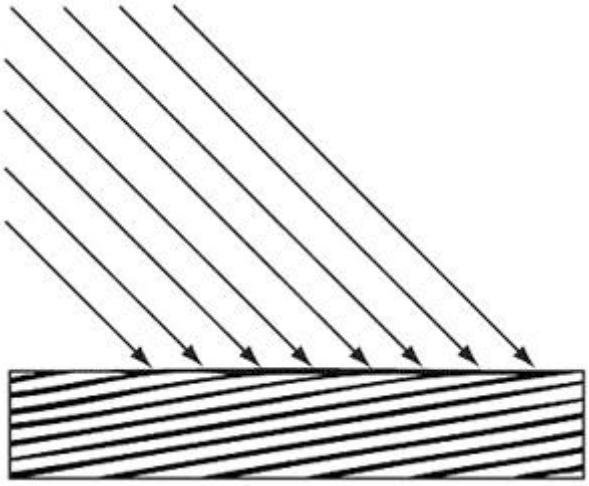


**图 7.17 平行光照射在一个表面上。**

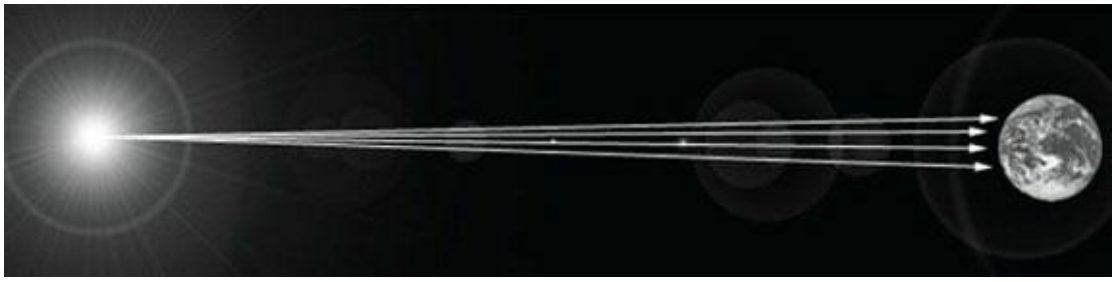


**图 7.18 该图未按比例绘制，但是当你在地球上选择一小块区域时，照在该区域上的太阳光几乎是平行的。**

# 7.10 点光

在现实生活中，最常见的点光源是灯泡；它可以向各个方向发射光线（参见图 7.19）。对于任意一点 P，都有一条从点光位置 Q 射向点 P 的线。通常，光照向量与点光的传播方向相反；也就是，该方向从点 P 指向点光源 Q。

$$\mathbf{L} = \frac{\mathbf{Q} - \mathbf{P}}{\|\mathbf{Q} - \mathbf{P}\|}$$

本质上，点光和平行光之间的唯一区别是光照向量的计算方式——点光会随着点的位置而变化，而平行光会保持为一个常量。

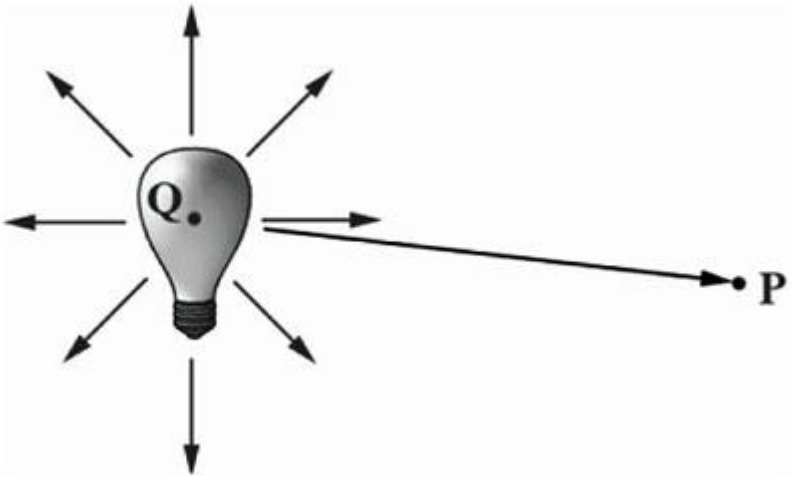



**图 7.19 点光向各个方向发射光线；对于任意一点 P，都有一条从点光源 Q 射向点 P 的线。**

## 7.10.1 衰减

按物理规律来说，光照强度会随着距离的增加而衰减，它与距离的平方成反比。也就是说，在一点上的光照强度由该点与光源之间的距离 $d$ 来决定：

$$I(d)=\frac{I_0}{d^2}$$

其中，$I_0$ 是 $d$=1 时的光照强度。不过，这个公式得到的计算结果不是总能令人满意。所以，我们打算使用一个更通用的函数，让美术师和程序员通过一些参数来控制照强度（即，可以让美术师和程序员尝试各种不同的参数，直至得到满意的效果为止），而不是单纯追求物理准确性。用于调节灯光强度的典型公式为：

$$I(d)=\frac{I_0}{a_0+a_1d+a_2d^2}$$

我们将 $a_0$、$a_1$ 和 $a_2$ 称为衰减参数，它们由美术师或程序员来指定。例如，当你希望光照强度与距离成反比时，可以设置 $a_0$=0、$a_1$=1、$a_2$=0。当你希望光照强度与距离的平方成反比时，可以设置 $a_0$=0、$a_1$=0、$a_2$=1。

把衰减参数引入光照方程，得到：

$$LitColor=\mathbf{A}+\frac{k_d\mathbf{D}+k_s\mathbf{S}}{a_0+a_1d+a_2d^2}\qquad \text{公式 7.4}$$

注意，因为环境光模拟的是充斥周围的间接光照，因此衰减参数并不会影响环境光

## 7.10.2 范围

对于点光来说，我们可以引入一个附加的范围参数。当一个点与点光源之间的距离大于指定的范围时，使它不接收该光源的照射。当需要让一个光源只对一个特定区域产生光照时，该参数非常有用。虽然使用衰减参数也可以让光照强度随着距离的增加而衰减，但是明确地指定光源范围仍然有益。范围参数可以用于优化着色器代码。我们马上就会看到，在着色器代码中，如果一个点的位置超出了光照范围，那我们就可以通过动态分支语句跳过该点的光

照计算。范围参数不影响平行光，因为这种光源的位置非常远。

# 7.11 聚光灯

在现实生活中，最常见的聚光灯是手电筒。本质上，聚光灯由一个位置 **Q**、一个方向向量 **d** 和一个圆锥体光照区域来描述（参见图 7.20）。

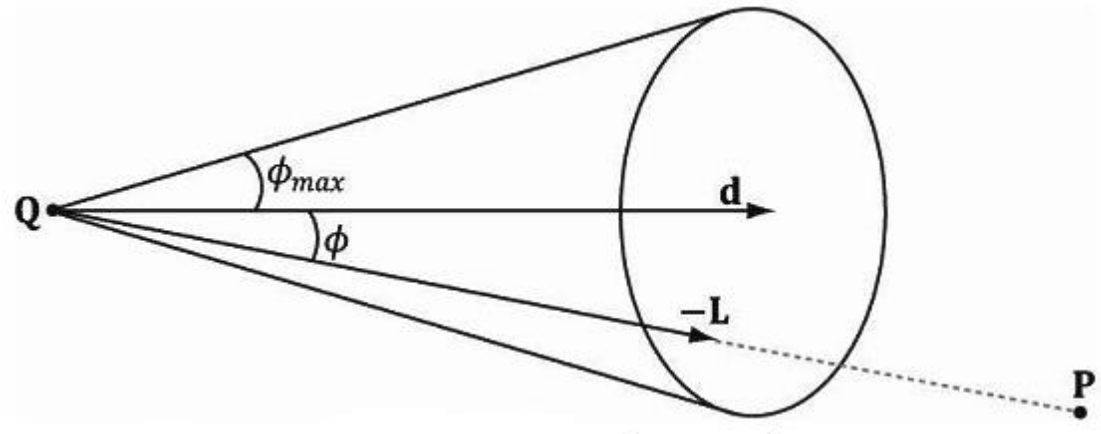



**图 7.20 聚光灯由一个位置 Q、一个方向向量 d 和一个半角角度为$\phi_{max}$的圆锥体照区域来描述。**

当实现一个聚光灯时，我们开始所做的事情与点光相同。光照向量可以由以下公式描述：

$$\mathbf{L} = \frac{\mathbf{Q}-\mathbf{P}}{\|\mathbf{Q}-\mathbf{P}\|}$$

其中，**P** 是接收照的点的位置，**Q** 是聚光灯的位置。从图 7.20 中可以看到，当且仅当，**−L** 与 **d** 之间的角度$\phi$小于圆锥角$\phi_{max}$时，**P** 在聚光灯的锥形范围内（所以它可以接收光照）。另外，在聚光灯的圆锥体区域中的线应该具有不同的强度；越靠近圆锥体中心的光线应该越强，随着角度$\phi$从 0 增加到$\phi_{max}$，光线强度应该逐渐衰退为 0（零）。

那么，我们应该如何通过一个$\phi$的函数来控制衰减强度，以及如何控制聚光灯的圆锥体区域大小呢？其实很简单，我们只需要故伎重演，直接套用控制镜面高光圆锥体反射系数的公式即可。也就是，使用如下函数：

$$k_{spot}(\phi) = \max(\cos\phi, 0)^s = \max(-\mathbf{L}\cdot\mathbf{d}, 0)^s$$

回顾图 7.12 所示的函数曲线图。我们可以看到，当$\phi$增加时，强度逐渐衰减，这是我们想要得到的结果；另外，通过修改指数 *s*，我们可以间接地控制$\phi_{max}$（它是当光照强度降低为 0 时的圆锥体角度）；也就是说，我们可以通过改变 *s* 来缩小或扩大聚光灯的圆锥体区域大小。例如，当我们将 *s* 设为 8 时，圆锥体的半角角度约为 45º。这样，聚光灯与点光的方程基本相同，只是要再乘以一个聚光灯因子，根据点与聚光灯圆锥体的相对位置按比例调整光照强度：

$$LitColor = k_{spot}\left(\mathbf{A} + \frac{k_d\mathbf{D} + k_s\mathbf{S}}{a_0 + a_1 d + a_2 d^2}\right) \qquad \text{（公式 7.5）}$$

**注意**：比较公式 7.4 和 7.5，我们可以看出聚光灯比点光耗费的资源要高得多，这是因为我们需要计算 $k_{spot}$ 因子。同理，比较公式 7.3 和 7.4，我们可以看出点光比平行光耗费的

资源要高得多，这是因为需要计算距离 $d$（这个计算涉及到平方根运算，非常耗时），而且需要除以衰减表达式。总而言之，从资源耗费角度而言，平行光要求最低，点光其次，聚光灯最高。

# 7.12 实现

## 7.12.1 光照结构体

在 **LightHelper.h** 文件中，我们定义了下面的结构体来表示平行光、点光或聚光灯。

```
struct DirectionalLight
{
    DirectionalLight() { ZeroMemory(this, sizeof(this)); }

    XMFLOAT4 Ambient;
    XMFLOAT4 Diffuse;
    XMFLOAT4 Specular;
    XMFLOAT3 Direction;
    float Pad; // 占位最后一个 float，这样我们就可以设置光源数组了。
};

struct PointLight
{
    PointLight() { ZeroMemory(this, sizeof(this)); }

    XMFLOAT4 Ambient;
    XMFLOAT4 Diffuse;
    XMFLOAT4 Specular;

    // 打包到 4D 矢量: (Position, Range)
    XMFLOAT3 Position;
    float Range;

    // 打包到 4D 矢量: (A0, A1, A2, Pad)
    XMFLOAT3 Att;
    float Pad; // 占位最后一个 float,，这样我们就可以设置光源数组了。
};

struct SpotLight
{
    SpotLight() { ZeroMemory(this, sizeof(this)); }

    XMFLOAT4 Ambient;
```

```
    XMFLOAT4 Diffuse;
    XMFLOAT4 Specular;

    // 打包到 4D 矢量: (Position, Range)
    XMFLOAT3 Position;
    float Range;

    // 打包到 4D 矢量: (Direction, Spot)
    XMFLOAT3 Direction;
    float Spot;

    // 打包到 4D 矢量: (Att, Pad)
    XMFLOAT3 Att;
    float Pad; // 占位最后一个 float,, 这样我们就可以设置光源数组了。
};
```

1. **Ambient**：由光源发射的环境光的数量。
2. **Diffuse**：由光源发射的漫反射光的数量。
3. **Specular**：由光源发射的高光的数量。
4. **Direction**：灯光方向。
5. **Position**：灯光位置。
6. **Range**：光照范围（离开光源的距离大于这个值的点不会被照亮）。
7. **Attenuation**：按照（$a_0$、$a_1$ 和 $a_2$）的顺序存储 3 个衰减常量。衰减常量只用于点和聚光灯，用于控制光强随距离衰减的程度。
8. **Spot**：该指数用于控制聚光灯的圆锥体区域大小；这个值只用于聚光灯。

我们会在下一节中讨论“pad”变量的必要性和“packing”格式。

定义在 LightHelper.fx 文件中的结构体镜像了上面的结构体：

```
struct DirectionalLight
{
    float4 Ambient;
    float4 Diffuse;
    float4 Specular;
    float3 Direction;
    float pad;
};

struct PointLight
{
    float4 Ambient;
    float4 Diffuse;
    float4 Specular;

    float3 Position;
    float Range;
```

```
    float3 Att;
    float pad;
};

struct SpotLight
{
    float4 Ambient;
    float4 Diffuse;
    float4 Specular;

    float3 Position;
    float Range;

    float3 Direction;
    float Spot;

    float3 Att;
    float pad;
};
```

## 7.12.2 结构 Packing

定义了 HLSL 结构体后，我们就可以像以下代码一样初始化常量缓冲：

```
cbuffer cbPerFrame
{
    DirectionalLight gDirLight;
    PointLight gPointLight;
    SpotLight gSpotLight;
    float3 gEyePosW;
} ;
```

在应用程序中初始化对应的灯光结构体，我们想在一次调用中将灯光实例设置到 effect 变量中，而不是单独地设置每个成员变量。可以用以下函数将一个结构实例设置到一个 effect 变量实例上：

```
ID3DX11EffectVariable::SetRawValue(void *pData,
    UINT Offset,UINT Count);
// Example call:
DirectionalLight  mDirLight;
mfxDirLight->SetRawValue(&mDirLight, 0, sizeof(mDirLight));
```

但是，由于这个函数只是简单地复制原始字节，因此如果你不仔细的话会导致很难找到错误，这里所说的要仔细对待的东西是指 C++的打包（packing）规则与 HLSL 不同。

在 HLSL 中，当元素打包到要给 4D 矢量中时会发生结构填充（padding），根据 HLSL 的规则，一个元素无法拆分到两个 4D 矢量中。考虑下面的例子：

```
//HLSL
```

```
struct S
{
    float3 Pos;
    float3 Dir;
} ;
```

如果将这个数据打包在 4D 矢量中，你可能以为会是这样：

```
vector1:(Pos.x,Pos.y,Pos.z,Dir.x)
vector2:(Dir.y,Dir.z,empty,empty)
```

但是，将元素 dir 打包在两个不同的 4D 矢量中违反了 HLSL 的规则，是不被允许的——一个元素不可以跨过一个 4D 矢量的范围，它只能这样被打包：

```
vector1:(Pos.x,Pos.y,Pos.z,empty)
vector2:(Dir.x,Dir.y,Dir.z,empty)
```

现在假设我们对应的 C++结构如下所示：

```
//  C++
struct S
{
    XMFLOAT3 Pos;
    XMFLOAT3 Dir;
};
```

如果我们不注意打包规则，只是盲目地调用并复制数据，我们就会得到错误的结果：

```
vector1:(Pos.x,Pos.y,Pos.z,Dir.x)
vector2:(Dir.y,Dir.z,empty,empty)
```

所以我们必须基于 HLSL 打包规则定义 C++结构，才能让元素正确地复制到 HLSL 结构中。我们要使用一个“pad”变量解决上述问题。让我们看几个 HLSL 打包的例子。

例如有下列一个结构：

```
struct S
{
    float3 v;
    float s;
    float2 p;
    float3 q;
} ;
```

这个结构会有空隙，数据会被打包到 3 个 4D 矢量中：

```
vector1:(v.x,v.y,v.z,s)
vector2:(p.x,p.y,empty,empty)
vector3:(q.x,q.y,q.z,empty)
```

我们可以将标量放在第 1 个矢量的第 4 个分量中。但是，我们无法在第 2 个矢量中匹配 q 的全部分量，所以需要为它分配单独的矢量。

最后一个例子，结构：

```
struct S
{
    float2  u;
    float2  v;
```

```
    float  a0;
    float  a1;
    float  a2;
} ;
```

会这样被打包：

```
vector1:(u.x,u.y,v.x,v.y)
vector2:(a0,a1,a2,empty)
```

**注意**：数组的处理方法有所不同。SDK 文档中这样说明：“数组中的每个元素都存储在一个 4 个分量的矢量中。”所以，若你有一个 float2 的数组：

```
float2 TexOffsets[8];
```

你可能会认为如上面的例子所说，两个 float2 会包装在一个 float4 中。但是，数组是例外，上面的代码等同于：

```
float4 TexOffsets[8];
```

所以，你需要在 C++中定义一个 8 个 XMFLOAT4 的数组，而不是 8 个 XMFLOAT2 的数组，这样才能正常运行。我们实际上只需要一个 float2 数组，所以每个元素浪费了两个 float 的空间。SDK 文档指出你可以使用转换和额外的地址计算指令提高内存的使用效率：

```
float4  array[4];
static float2  aggressivePackArray[8]  = (float2[8])array;
```

## 7.12.3 实现平行光

下面的 HLSL 函数根据给出的材质、平行光源、表面法线和由表面指向观察点的矢量，输出光照后的表面颜色。

```
void ComputeDirectionalLight(Material mat, DirectionalLight L,
                             float3 normal, float3 toEye,
                             out float4 ambient,
                             out float4 diffuse,
                             out float4 spec)
{
    // 初始化输出的变量
    ambient = float4(0.0f, 0.0f, 0.0f, 0.0f);
    diffuse = float4(0.0f, 0.0f, 0.0f, 0.0f);
    spec    = float4(0.0f, 0.0f, 0.0f, 0.0f);

    // 光照矢量与光线的传播方向相反
    float3 lightVec = -L.Direction;

    // 添加环境光
    ambient = mat.Ambient * L.Ambient;

    // 添加漫反射和镜面光
    float diffuseFactor = dot(lightVec, normal);
```

```
    // Flatten 避免动态分支
    [flatten]
    if( diffuseFactor > 0.0f )
    {
        float3 v         = reflect(-lightVec, normal);
        float    specFactor    =    pow(max(dot(v,    toEye),    0.0f),
mat.Specular.w);

        diffuse = diffuseFactor * mat.Diffuse * L.Diffuse;
        spec    = specFactor * mat.Specular * L.Specular;
    }
}
```

其中，使用的 HLSL 内置函数有：**dot**、**reflect**、**pow** 和 **max**，它们分别用来计算向量点积、向量反射、乘方和取最大值。读者可以在附录 B 中找到大部分 HLSL 内置函数的描述，以及有关 HLSL 语法的快速入门。另外还有一件事情需要注意，那就是当两个向量使用*运算符相乘时，实际执行的是分量乘法。

**注意**：PC 上的 HLSL 函数总是内联的，调用函数或传递参数不会有性能损失。

## 7.12.4 实现点光

下面的 HLSL 函数根据给出的材质、点光源、表面位置、表面法线和表面点到观察点的的矢量信息，输出光照后的表面颜色。

```
void ComputePointLight(Material mat, PointLight L, float3 pos, float3
normal, float3 toEye,
                out float4 ambient, out float4 diffuse, out float4
spec)
{
    // 初始化输出变量
    ambient = float4(0.0f, 0.0f, 0.0f, 0.0f);
    diffuse = float4(0.0f, 0.0f, 0.0f, 0.0f);
    spec    = float4(0.0f, 0.0f, 0.0f, 0.0f);

    // 表面指向光源的矢量
    float3 lightVec = L.Position - pos;

    // 表面离光源的距离
    float d = length(lightVec);

    // Range test.
    if( d > L.Range )
        return;

    // Normalize the light vector.
```

```
    lightVec /= d;

    // Ambient term.
    ambient = mat.Ambient * L.Ambient;

    // Add diffuse and specular term, provided the surface is in
    // the line of site of the light.

    float diffuseFactor = dot(lightVec, normal);

    // Flatten to avoid dynamic branching.
    [flatten]
    if( diffuseFactor > 0.0f )
    {
        float3 v         = reflect(-lightVec, normal);
        float    specFactor    =    pow(max(dot(v,    toEye),    0.0f),
mat.Specular.w);

        diffuse = diffuseFactor * mat.Diffuse * L.Diffuse;
        spec    = specFactor * mat.Specular * L.Specular;
    }

    // 衰减
    float att = 1.0f / dot(L.Att, float3(1.0f, d, d*d));

    diffuse *= att;
    spec    *= att;
}
```

## 7.12.5 实现聚光灯

下面的 HLSL 函数根据给出的材质、聚光灯、表面位置、表面法线、表面点指向观察点位置的矢量信息，输出光照后的表面颜色：

```
void ComputeSpotLight(Material mat, SpotLight L, float3 pos, float3
normal, float3 toEye,
                out float4 ambient, out float4 diffuse, out float4
spec)
{
    // 初始化输出变量.
    ambient = float4(0.0f, 0.0f, 0.0f, 0.0f);
    diffuse = float4(0.0f, 0.0f, 0.0f, 0.0f);
    spec    = float4(0.0f, 0.0f, 0.0f, 0.0f);
```

```
    // 从表面指向光源的光照矢量
    float3 lightVec = L.Position - pos;

    // 表面离开光源的距离
    float d = length(lightVec);

    // Range test.
    if( d > L.Range )
        return;

    // 规范化光照矢量
    lightVec /= d;

    // 计算环境光
    ambient = mat.Ambient * L.Ambient;

    // 计算漫反射和镜面光，provided the surface is in
    // the line of site of the light.

    float diffuseFactor = dot(lightVec, normal);

    // Flatten 避免动态分支
    [flatten]
    if( diffuseFactor > 0.0f )
    {
        float3 v         = reflect(-lightVec, normal);
        float    specFactor    =    pow(max(dot(v,    toEye),    0.0f),
mat.Specular.w);

        diffuse = diffuseFactor * mat.Diffuse * L.Diffuse;
        spec    = specFactor * mat.Specular * L.Specular;
    }

    // Scale by spotlight factor and attenuate.
    float spot = pow(max(dot(-lightVec, L.Direction), 0.0f), L.Spot);

    // Scale by spotlight factor and attenuate.
    float att = spot / dot(L.Att, float3(1.0f, d, d*d));

    ambient *= spot;
    diffuse *= att;
    spec    *= att;
}
```

# 7.13 光照演示程序

在我们的光照演示程序中，我们实现了 3 种个不同的光源：1 个平行光、1 个点光、1 个聚光灯。平行光的位置是固定的，点光绕着地形转圈，而聚光灯跟随相机移动，并指向相机的观察方向。光照演示程序只是对上一章的水波演示程序做了一些修改。effect 文件的内容在下一节中，其中使用的结构体和函数都已在 7.10 节中讨论过了。

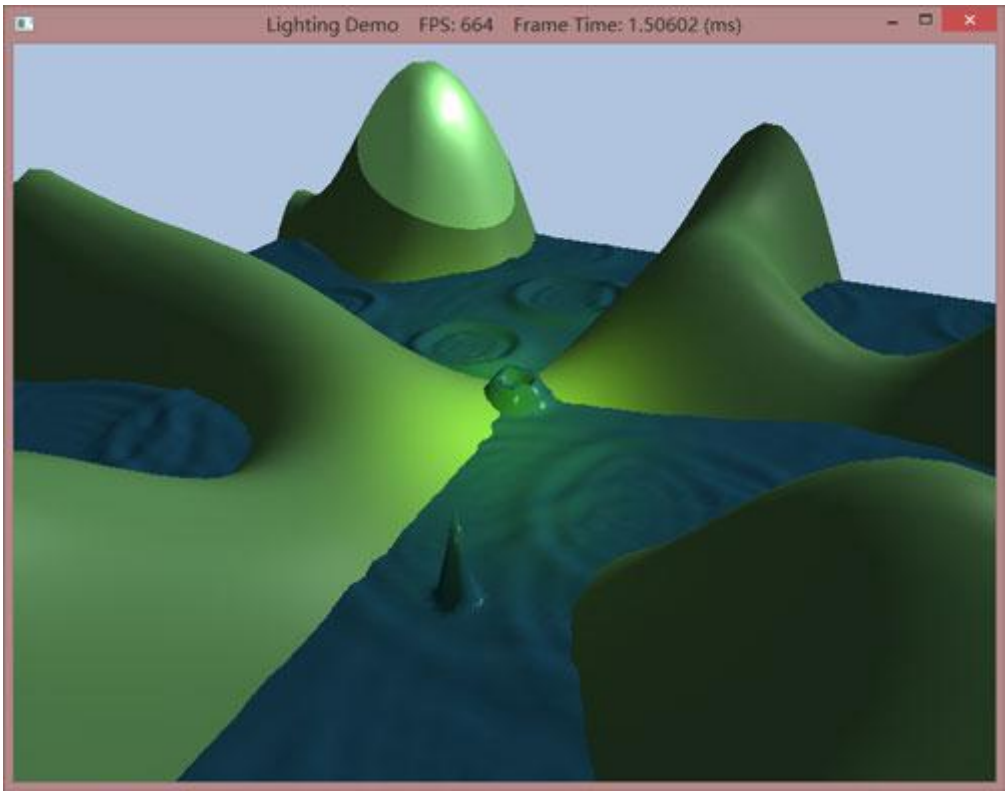



**图 7.21：光照演示程序的屏幕截图。**

## 7.13.1 effect 文件

```
//=============================================================
==============
// Lighting.fx by Frank Luna (C) 2011 All Rights Reserved.
//
// Transforms and lights geometry.
//=============================================================
==============

#include "LightHelper.fx"

cbuffer cbPerFrame
{
    DirectionalLight gDirLight;
```

```
    PointLight gPointLight;
    SpotLight gSpotLight;
    float3 gEyePosW;
};

cbuffer cbPerObject
{
    float4x4 gWorld;
    float4x4 gWorldInvTranspose;
    float4x4 gWorldViewProj;
    Material gMaterial;
};

struct VertexIn
{
    float3 PosL    : POSITION;
    float3 NormalL : NORMAL;
};

struct VertexOut
{
    float4 PosH    : SV_POSITION;
    float3 PosW    : POSITION;
    float3 NormalW : NORMAL;
};

VertexOut VS(VertexIn vin)
{
    VertexOut vout;

    // 转换到世界空间
    vout.PosW    = mul(float4(vin.PosL, 1.0f), gWorld).xyz;
    vout.NormalW = mul(vin.NormalL, (float3x3)gWorldInvTranspose);

    // 转换到齐次剪裁空间
    vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj);

    return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
    // 插值后的法线需要重新规范化
    pin.NormalW = normalize(pin.NormalW);
```

```
    float3 toEyeW = normalize(gEyePosW - pin.PosW);

    // 初始化光照变量
    float4 ambient = float4(0.0f, 0.0f, 0.0f, 0.0f);
    float4 diffuse = float4(0.0f, 0.0f, 0.0f, 0.0f);
    float4 spec    = float4(0.0f, 0.0f, 0.0f, 0.0f);

    // 每个光源的贡献
    float4 A, D, S;

    ComputeDirectionalLight(gMaterial,    gDirLight,    pin.NormalW,
toEyeW, A, D, S);
    ambient += A;
    diffuse += D;
    spec    += S;

    ComputePointLight(gMaterial, gPointLight, pin.PosW, pin.NormalW,
toEyeW, A, D, S);
    ambient += A;
    diffuse += D;
    spec    += S;

    ComputeSpotLight(gMaterial, gSpotLight, pin.PosW, pin.NormalW,
toEyeW, A, D, S);
    ambient += A;
    diffuse += D;
    spec    += S;

    float4 litColor = ambient + diffuse + spec;

    // 通常从漫反射材质中提取 alpha
    litColor.a = gMaterial.Diffuse.a;

    return litColor;
}

technique11 LightTech
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_5_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_5_0, PS() ) );
```

```
    }
}
```

## 7.13.2 C++程序代码

光照计算需要表面法线的信息。我们是在顶点层次定义法线的，然后对法线进行插值用于逐像素光照计算。此外，现在也不需要指定顶点颜色了，表面的颜色可以代入光照方程逐像素地求得。输入布局描述如下所示：

```
D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
{
    {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D11_INPUT_PER_VERTEX_DATA, 0},
    {"NORMAL",    0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
D3D11_INPUT_PER_VERTEX_DATA, 0}
};
```

在应用程序中，我们定义了三个光源和两个材质。

```
DirectionalLight mDirLight;
PointLight mPointLight;
SpotLight mSpotLight;
Material mLandMat;
Material mWavesMat;
```

它们在构造函数中进行初始化：

```
LightingApp::LightingApp(HINSTANCE hInstance)
:    D3DApp(hInstance),    mLandVB(0),    mLandIB(0),    mWavesVB(0),
mWavesIB(0),
  mFX(0),    mTech(0),    mfxWorld(0),    mfxWorldInvTranspose(0),
mfxEyePosW(0),
  mfxDirLight(0),        mfxPointLight(0),        mfxSpotLight(0),
mfxMaterial(0),
  mfxWorldViewProj(0),
  mInputLayout(0),        mEyePosW(0.0f,        0.0f,        0.0f),
mTheta(1.5f*MathHelper::Pi),            mPhi(0.1f*MathHelper::Pi),
mRadius(80.0f)
{
    …

    // Directional light.
    mDirLight.Ambient  = XMFLOAT4(0.2f, 0.2f, 0.2f, 1.0f);
    mDirLight.Diffuse  = XMFLOAT4(0.5f, 0.5f, 0.5f, 1.0f);
    mDirLight.Specular = XMFLOAT4(0.5f, 0.5f, 0.5f, 1.0f);
    mDirLight.Direction = XMFLOAT3(0.57735f, -0.57735f, 0.57735f);

    // Point light--position is changed every frame to animate in
```

```
UpdateScene function.
    mPointLight.Ambient  = XMFLOAT4(0.3f, 0.3f, 0.3f, 1.0f);
    mPointLight.Diffuse  = XMFLOAT4(0.7f, 0.7f, 0.7f, 1.0f);
    mPointLight.Specular = XMFLOAT4(0.7f, 0.7f, 0.7f, 1.0f);
    mPointLight.Att      = XMFLOAT3(0.0f, 0.1f, 0.0f);
    mPointLight.Range    = 25.0f;

    // Spot light--position and direction changed every frame to
animate in UpdateScene function.
    mSpotLight.Ambient  = XMFLOAT4(0.0f, 0.0f, 0.0f, 1.0f);
    mSpotLight.Diffuse  = XMFLOAT4(1.0f, 1.0f, 0.0f, 1.0f);
    mSpotLight.Specular = XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f);
    mSpotLight.Att      = XMFLOAT3(1.0f, 0.0f, 0.0f);
    mSpotLight.Spot     = 96.0f;
    mSpotLight.Range    = 10000.0f;

    mLandMat.Ambient  = XMFLOAT4(0.48f, 0.77f, 0.46f, 1.0f);
    mLandMat.Diffuse  = XMFLOAT4(0.48f, 0.77f, 0.46f, 1.0f);
    mLandMat.Specular = XMFLOAT4(0.2f, 0.2f, 0.2f, 16.0f);

    mWavesMat.Ambient  = XMFLOAT4(0.137f, 0.42f, 0.556f, 1.0f);
    mWavesMat.Diffuse  = XMFLOAT4(0.137f, 0.42f, 0.556f, 1.0f);
    mWavesMat.Specular = XMFLOAT4(0.8f, 0.8f, 0.8f, 96.0f);
}
```

当使用多光源时，必须要小心别让颜色超出范围。所以你需要测试环境光、漫反射光和镜面光的强度，对光照范围和衰减的测试也是必须的。有时很容易忘记使用带颜色的灯光，但是它们可以被用来产生不同的效果。例如，你在一个太阳模型上使用平行光，太阳本身被设置为橘黄色，你可以将光的颜色也设置为橘黄色，这样，场景中的物体都会带上橘黄色的色调。异形飞船爆炸时可以加点淡蓝色，红光通常表示紧急情况。

如前所述，点光源和聚光灯是活动的；这一工作在 **UpdateScene** 方法中实现：

```
void LightingApp::UpdateScene(float dt)
{
    …
    //
    // 光源动画
    //

    // 让点光源在地面之上转圈
    mPointLight.Position.x = 70.0f*cosf( 0.2f*mTimer.TotalTime() );
    mPointLight.Position.z = 70.0f*sinf( 0.2f*mTimer.TotalTime() );
    mPointLight.Position.y = MathHelper::Max(GetHillHeight(mPointLight.Position.x,
        mPointLight.Position.z), -3.0f) + 10.0f;
```

```
    // 聚光灯的位置与观察点的位置相同，光照方向与观察方向相同；
    // 这样产生的效果就像是观察者拿着一个手电筒在场景中照来照去一样。
    mSpotLight.Position = mEyePosW;
    XMStoreFloat3(&mSpotLight.Direction, XMVector3Normalize(target - pos));
}
```

基本上，点光源会沿着 *xz* 平面上的一个圆形轨迹来运动，只是它总保持在地面和水面之上。聚光灯的位置与观察点的位置相同，光照方向与观察方向相同；这样产生的效果就像是观察者拿着一个手电筒在场景中照来照去一样。

最后，我们要在渲染之前将灯光和材质指定给 effect：

```
void LightingApp::DrawScene()
{
    …

    // Set per frame constants.
    mfxDirLight->SetRawValue(&mDirLight, 0, sizeof(mDirLight));
    mfxPointLight->SetRawValue(&mPointLight,                    0,
sizeof(mPointLight));
    mfxSpotLight->SetRawValue(&mSpotLight, 0, sizeof(mSpotLight));
    mfxEyePosW->SetRawValue(&mEyePosW, 0, sizeof(mEyePosW));

    …
    D3DX11_TECHNIQUE_DESC techDesc;
    mTech->GetDesc( &techDesc );
    for(UINT p = 0; p < techDesc.Passes; ++p)
    {
        …
        mfxMaterial->SetRawValue(&mLandMat, 0, sizeof(mLandMat));
        /* Draw the hills...*/
        …
        mfxMaterial->SetRawValue(&mWavesMat, 0, sizeof(mWavesMat));
        /* Draw waves...*/
    }

    HR(mSwapChain->Present(0, 0));
}
```

## 7.13.3 法线计算

因为我们的地形表面是通过一个函数 $y = f(x,z)$生成的，所以我们可以直接使用微积分计算法线向量，而不是 7.2.1 节描述的法线平均值技术。要为表面上的每个点计算法线向量，我们必须通过偏导数生成+*x* 和+*z* 方向上的两个正切向量：

$$\mathbf{T}_x = \left(1, \frac{\partial f}{\partial x}, 0\right)$$

$$\mathbf{T}_z = \left(0, \frac{\partial f}{\partial z}, 1\right)$$

这两个向量位于表面点的正切平面上。我们可以通过计算叉积得到正切平面的法线向量：

$$\mathbf{n} = \mathbf{T}_z \times \mathbf{T}_x = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 0 & \frac{\partial f}{\partial z} & 1 \\ 1 & \frac{\partial f}{\partial x} & 0 \end{vmatrix}$$

$$= \left( \begin{vmatrix} \frac{\partial f}{\partial z} & 1 \\ \frac{\partial f}{\partial x} & 0 \end{vmatrix}, - \begin{vmatrix} 0 & 1 \\ 1 & 0 \end{vmatrix}, \begin{vmatrix} 0 & \frac{\partial f}{\partial z} \\ 1 & \frac{\partial f}{\partial x} \end{vmatrix} \right)$$

$$= \left( -\frac{\partial f}{\partial x}, 1, -\frac{\partial f}{\partial z} \right)$$

我们用于生成地面网格的函数为：

$$f(x,z) = 0.3z\sin(0.1x) + 0.3x\cos(0.1z)$$

偏导数为：

$$\frac{\partial f}{\partial x} = 0.03z\cos(0.1x) + 0.3\cos(0.1z)$$

$$\frac{\partial f}{\partial z} = 0.03\sin(0.1x) - 0.03x\sin(0.1z)$$

表面点$(x,f(x,z),z)$的表面法线为：

$$\mathbf{n}(x,z) = \left( -\frac{\partial f}{\partial x}, 1, -\frac{\partial f}{\partial z} \right) = \begin{bmatrix} -0.03z\cos(0.1x) - 0.3\cos(0.1z) \\ -0.3\sin(0.1x) + 0.03x\sin(0.1z) \end{bmatrix}^T$$

注意，表面法线不是规范化向量，在执行照计算之前必须对它进行规范化处理。

我们只在每个顶点上执行上述法线计算，获得顶点法线：

```
XMFLOAT3 LightingApp::GetHillNormal(float x, float z)const
{
    // n = (-df/dx, 1, -df/dz)
    XMFLOAT3 n(
        -0.03f*z*cosf(0.1f*x) - 0.3f*cosf(0.1f*z),
        1.0f,
        -0.3f*sinf(0.1f*x) + 0.03f*x*sinf(0.1f*z));

    XMVECTOR unitNormal = XMVector3Normalize(XMLoadFloat3(&n));
    XMStoreFloat3(&n, unitNormal);

    return n;
```

```
}
```

对于水体表面来说，法线向量的计算过程基本相同，只是我们没有用于生成水体的公式。不过，读者可以使用有限差异图（finite difference scheme）估算每个顶点上的正切向量（详情请参见[Lengyel02]或任何一本关于数值分析的书籍）。

**注意**：如果你的微积分已经荒废了，那也不用担心；因为它在本书中的作用不是非常重要。我们在这里使用微积分，只是为了以数学方式生成一个曲面几何体，使我们在演示程序中能够渲染一些比较有趣的物体。在本书最后，我们将为读者讲解如何载入和使用那些由3D 建模软件导出的 3D 网格文件。

# 8.1 纹理和资源概述

我们的演示程序正在变得越来越有趣，但是顶点颜色无法体现真实世界物体所具有的细节。纹理贴图映射（texture mapping）是一种将图像数据映射到三角形表面的技术，它可以显著提高场景的细节和真实感。例如，我们可以创建一个立方体，然后将一个板条箱纹理映射到立方体的每个面上，使它看上去更像是一个板条箱（参见图 8.1）。

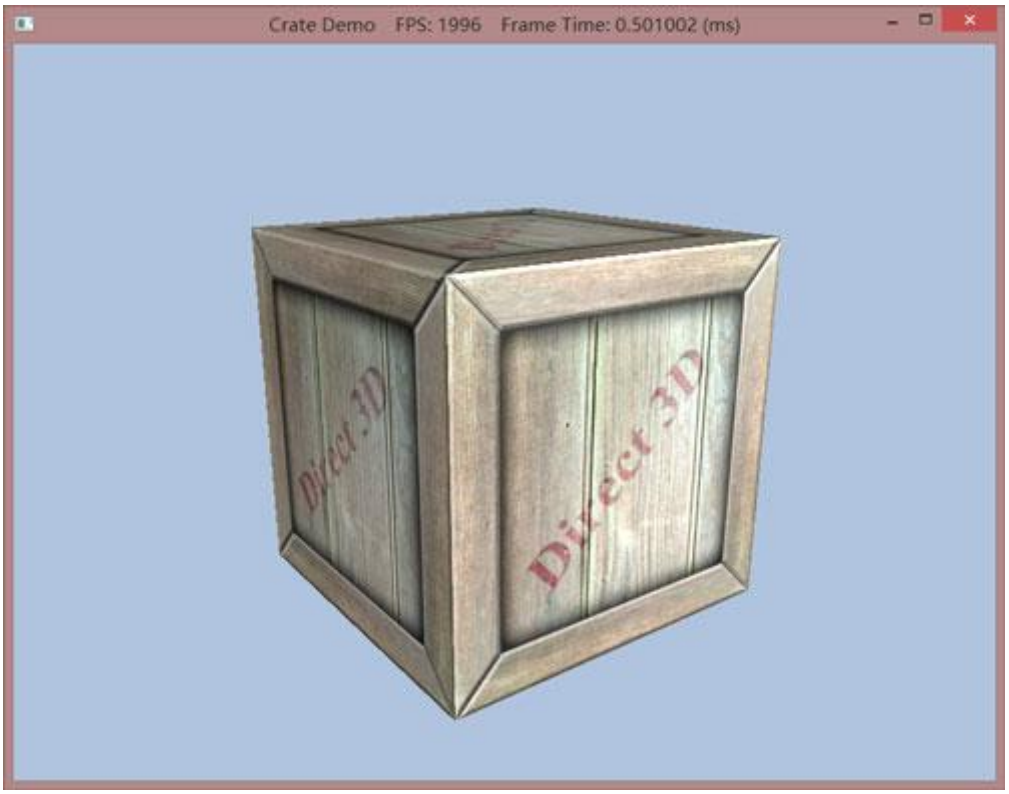



**图 8.1：板条箱演示程序创建了一个带有板条箱纹理的立方体。**

**学习目标**

1．学习如何将纹理片段映射到一个三角形上。
2．了解如何创建和启用纹理。
3．学习如何通过纹理过滤来消除图像中的锯齿，使整个图像看上去更平滑。
4．了解如何使用寻址模式实现纹理平铺。
5．了解如何将多重纹理合并为一幅新的纹理，实现一些特殊效果。
6．学习如何通过纹理动画来创建一些简单的效果。

回顾我们自从第 4 章之后学习和使用过的一些纹理；尤其是深度缓冲区和后台缓冲区，它们都是通过 **ID3D11Texture2D** 接口描述的 2D 纹理对象。在本节中，我们将回顾第 4 章讲过的一些有关纹理和材质的知识。

2D 纹理是一种数据元素矩阵。2D 纹理的用途之一是存储 2D 图像数据，在纹理的每个元素中保存一个像素颜色。但这不是纹理的唯一用途；例如，在一种称为法线贴图映射（normal mapping）的高级技术中，纹理存储的不是颜色，而是 3D 向量。因此，从通常意义上讲，纹理用来存储图像数据，但是在实际应用中纹理可以有更广泛的用途。1D 纹理（**ID3D11Texture1D**）类似于一个 1D 数据元素数组，3D 纹理（**ID3D11Texture3D**）类似一个 3D 数据元素数组。1D、2D、3D 纹理接口都继承自 **ID3D11Resource** 接口。

本章稍后会讨论纹理的更多特性。纹理比数据数组要复杂的多；纹理可以带有多级渐近纹理层（mipmap level），GPU 可以在纹理上执行特殊运算，比如使用过滤器（filter）和多重采样（multisampling）。但是并非所有数据块都能存入纹理；纹理只支持特定格式的数据存储，这些格式由 **DXGI_FORMAT** 枚举类型描述。其中一些常用的格式如下：

**1．DXGI_FORMAT_R32G32B32_FLOAT**：每个元素包含 3 个 32 位浮点分量。

**2．DXGI_FORMAT_R16G16B16A16_UNORM**：每个元素包含 4 个 16 位分量，分量的取值范围在[0,1]区间内。

**3．DXGI_FORMAT_R32G32_UINT**：每个元素包含 2 个 32 位无符号整数分量。

**4．DXGI_FORMAT_R8G8B8A8_UNORM**：每个元素包含 4 个 8 位无符号整数分量，分量的取值范围在[0,1]区间内。

**5．DXGI_FORMAT_R8G8B8A8_SNORM**：每个元素包含 4 个 8 位有符号整数分量，分量的取值范围在[−1,1] 区间内。

**6．DXGI_FORMAT_R8G8B8A8_SINT**：每个元素包含 4 个 8 位有符号整数分量，分量的取值范围在[−128, 127]区间内。

**7．DXGI_FORMAT_R8G8B8A8_UINT**：每个元素包含 4 个 8 位无符号整数分量，分量的取值范围在[0, 255]区间内。

注意，字母 R、G、B、A 分别表示 red（红）、green（绿）、blue（蓝）和 alpha（透明度）。不过，正如我们之前所述，纹理存储的不一定是颜色信息；例如，格式

**DXGI_FORMAT_R32G32B32_FLOAT**

包含 3 个浮点分量，可以存储一个使用浮点坐标的 3D 向量（而不一定是颜色向量）。另外，还有一种弱类型（typeless）格式，可以预先分配内存空间，然后在纹理绑定到管线时再指定如何重新解释数据内容（这一过程与数据类型转换颇为相似）。例如，下面的弱类型格式会为每个元素预留 4 个 8 位分量，且不指定数据类型（例如：整数、浮点数、无符号整数）：

**DXGI_FORMAT_R8G8B8A8_TYPELESS**

纹理可以被绑定到渲染管线的不同阶段；例如，比较常见的情况是将纹理作为渲染目标（即，Direct3D 渲染到纹理）和着色器资源（即，在着色器中对纹理进行采样）。当创建用于这两种目的的纹理资源时，应使用绑定标志值：

**D3D11_BIND_RENDER_TARGET | D3D11_BIND_SHADER_RESOURCE**

指定纹理所要绑定的两个管线阶段。其实，资源不能被直接绑定到一个管线阶段；我们只能把与资源关联的资源视图绑定到不同的管线阶段。无论以哪种方式使用纹理，Direct3D 始终要求我们在初始化时为纹理创建相关的资源视图。这样有助于提高运行效率，正如 SDK 文档指出的那样：“运行时环境与驱动程序可以在视图创建执行相应的验证和映射，减少绑定时的类型检查”。所以，当把纹理作为一个渲染目标和着色器资源时，我们要为它创建两种视图：渲染目标视图（**ID3D11RenderTargetView**）和着色器资源视图

（**ID3D11ShaderResourceView**）。资源视图主要有两个功能：（1）告诉 Direct3D 如何使用资源（即，指定资源所要绑定的管线阶段）；（2）如果在创建资源时指定的是弱类型（typeless）格式，那么在为它创建资源视图时就必须指定明确的资源类型。对于弱类型格式，纹理元素可能会在一个管线阶段中视为浮点数，而在另一个管线阶段中视为整数。

**注意**：2008 年 8 月的 SDK 文档指出："当创建资源时，为资源指定强类型（fully-typed）格式，把资源的用途限制在格式规定的范围内，有利于提高运行时环境对资源的访问速度……"。所以，你只应该在真正需要弱类型资源时，才创建弱类型资源；否则，应尽量创建强类型资源。

为了给一个资源创建特定的视图，该资源在创建时必须带有特定的绑定标志值。例如，在创建资源时如果没有使用 **D3D11_BIND_SHADER_RESOURCE** 绑定标志值（该标志值表示纹理将作为一个着色器资源绑定到管线上），那么我们就无法为这个资源创建 **ID3D11ShaderResourceView** 视图。如果你尝试一下就会发现 Direct3D 会报告如下调试错误：

```
D3D11: ERROR: ID3D11Device::CreateShaderResourceView:
A ShaderResourceView cannot be created of a Resource that
did not specify the D3D11_BIND_SHADER_RESOURCE BindFlag.
```

在本章中，我们主要讲解如何将纹理绑定为着色器资源，以使我们的像素着色器可以对纹理进行采样，并用纹理来实现几何体着色。

# 8.2 纹理坐标

Direct3D 的纹理坐标系由表示图像水平方向的 $u$ 轴和表示图像垂直方向的 $v$ 轴组成。坐标$(u,v)$指定了纹理上的一个元素，我们将该元素称为纹理元素（texel，译者注：texel 是 texture element 的缩写），其中 $0 \leqslant u,v \leqslant 1$。注意，$v$ 轴的正方向是"垂直向下"的（参见图 8.2）。另外，将规范化坐标区间设为[0,1]，是因为这样可以使 Direct3D 拥有一个独立于纹理尺寸的坐标空间；例如，无论纹理的实际尺寸是 256×256、512×1024 还是 2048×2048，(0.5, 0.5) 永远表示中间的纹理元素。同样，(0.25, 0.75)表示在水平方向上位于总宽度的 1/4 处、在垂直方向上位于总高度的 3/4 处的纹理元素。现在，我们只讨论[0,1]区间内的纹理坐标，稍后会讲解当纹理坐标超出一范围时的处理方法。

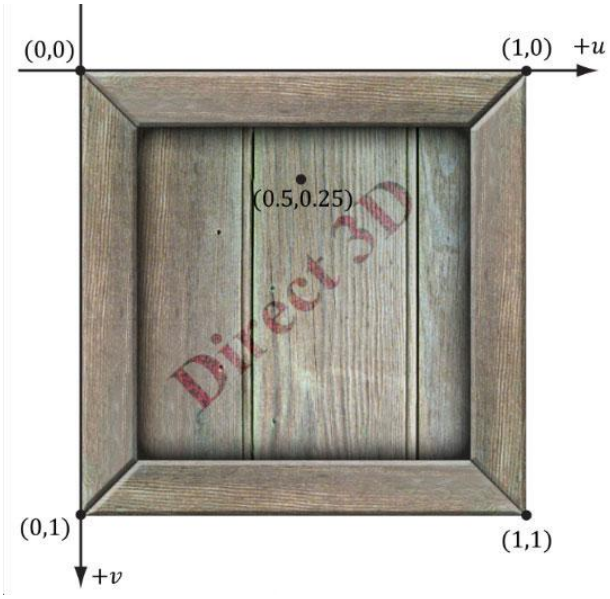



**图 8.2 纹理坐标系（texture coordinate system），有时也称为纹理空间（texture space）。**

对于每个 3D 三角形，我们都要在纹理上定义一个相应的 2D 三角形，以使纹理映射到 3D 三角形上（参见图 8.3）。

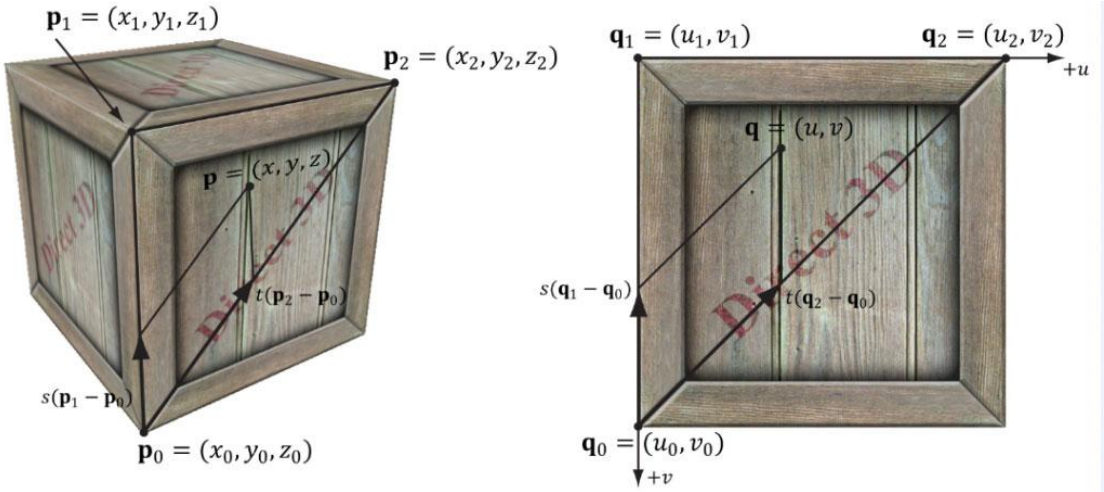



**图 8.3 左图是 3D 空间中的一个三角形，右图是我们在纹理上定义的 2D 三角形，它会被映射到 3D 三角形上。**

若三角形的顶点坐标为 $\mathbf{p}_0$，$\mathbf{p}_1$ 和 $\mathbf{p}_2$，对应的纹理坐标为 $\mathbf{q}_0$、$\mathbf{q}_1$ 和 $\mathbf{q}_2$。对于 3D 三角形上的任意一个点$(x,y,z)$，它的纹理坐标$(u,v)$都可以通过在 3D 三角形表面对顶点纹理坐标进行线性插值得到，而线性插值中的参数 $s$、$t$ 是相同的；也就是说：

$$(x,y,z) = \mathbf{p} = \mathbf{p}_0 + s(\mathbf{p}_1 - \mathbf{p}_0) + t(\mathbf{p}_2 - \mathbf{p}_0)$$

若 $s \geqslant 0$，$t \geqslant 0$，$s+t \leqslant 1$ 则，

$$(u,v) = \mathbf{q} = \mathbf{q}_0 + s(\mathbf{q}_1 - \mathbf{q}_0) + t(\mathbf{q}_2 - \mathbf{q}_0)$$

通过一方式，三角形上的每个点都可以得到一个相应的纹理坐标。

为了使用纹理，我们需要再次修改顶点结构体，添加一对纹理坐标，指定纹理上的点，使每个 3D 点都有一个相应的 2D 纹理点。这样，由 3 个顶点构成的每个 3D 三角形在纹理空间中都会有一个相应的 2D 纹理三角形（即，在每个 2D 纹理三角形和 3D 三角形之间建

立对应关系）。

```
// Basic 32-byte vertex structure.
struct Basic32
{
    XMFLOAT3 Pos;
    XMFLOAT3 Normal;
    XMFLOAT2 Tex;
};
const D3D11_INPUT_ELEMENT_DESC InputLayoutDesc::Basic32[3] =
{
    {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D11_INPUT_PER_VERTEX_DATA, 0},
    {"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
D3D11_INPUT_PER_VERTEX_DATA, 0},
    {"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
D3D11_INPUT_PER_VERTEX_DATA, 0}
};
```

**注意**：你可以创建一个 2D 纹理三角形和 3D 三角形差别很大的“奇怪”纹理映射。结果是，当这个 2D 纹理映射到 3D 三角形上时，会出现拉伸和扭曲，看起来不好。例如，将一个尖三角形映射到一个正三角形上就会发生拉伸现象。通常，纹理扭曲应尽量避免，除非艺术家期望这种扭曲的效果。

看一下图 8.3，我们将整张纹理映射到立方体的每个面上，但这并不是必须的。我们可以只将纹理的一部分映射到几何体上。我们可以将几张毫不相关的图像合并在一张大纹理贴图上（叫做纹理贴图集 texture atlas），然后将它用于不同的对象（图 8.4）。纹理坐标决定了将那一部分纹理映射到三角形上。

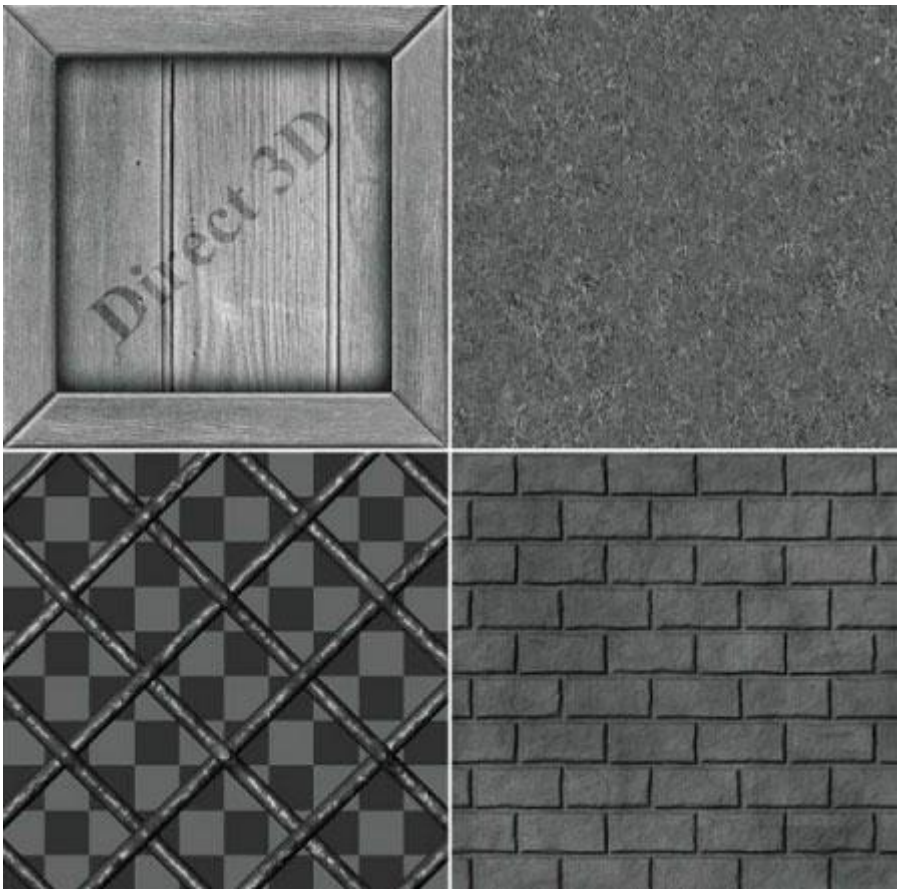



**图 8.4. 一张存储了 4 张纹理的纹理贴图集。设置每个顶点的纹理坐标就可以将期望的部分纹理映射到几何体上。**

# 8.3 创建和启用纹理

纹理数据通常是存储在磁盘上的图像文件。我们需要将它读取出来，并载入到一个 **ID3D11Texture2D** 对象中（参见 **D3DX11CreateTextureFromFile**）。不过，纹理资源是不能被直接绑定到渲染管线上的；我们需要为纹理创建一个着色器资源视图（**ID3D11ShaderResourceView**），然后将视图绑定到管线上。这个过程可分为两步：

1．调用 **D3DX11CreateTextureFromFile**，读取存储在磁盘上的图像文件，创建 **ID3D11Texture2D** 对象。

2．调用 **ID3D11Device::CreateShaderResourceView**，为纹理创建对应的着色器资源视图。

这两步也可通过如下 D3DX 函数一次完成：

```
HRESULT D3DX11CreateShaderResourceViewFromFile(
    ID3D11Device *pDevice,
    LPCTSTR pSrcFile,
    D3DX11_IMAGE_LOAD_INFO *pLoadInfo,
    ID3DX11ThreadPump *pPump,
    ID3D11ShaderResourceView **ppShaderResourceView,
    HRESULT *pHResult
```

```
);
```

**1．pDevice**：创建纹理时使用的 Direct3D 设备指针。

**2．pSrcFile**：所要载入的图像的文件名。

**3．pLoadInfo**：可选的图像信息；当设为空值时，该函数使用源图像文件信息。例如，我们在这里指定空值，那么源图像的尺寸就是纹理的尺寸；另外，该函数还会生成一个完整的多级渐近纹理链（8.4.2 节）。这个默认值很好，因为它所执行的一系列行为通常都是我们想要的。

**4．pPump**：在一个新的线程中载入资源。如果要在当前的工作线程中载入资源，请将它设为空值。在本书中，这个参数总是设为空值。

**5．ppShaderResourceView**：返回纹理的着色器资源视图指针。

**6．pHResult**：当 pPump 设为空值时，该参数也应设为空值。

该函数可以载入以下格式的图像文件：BMP、JPG、PNG、DDS、TIFF、GIF、WMP（参见 **D3DX11_IMAGE_FILE_FORMAT**）。

**注意**：有时，我们所说的“纹理”实际上是指“与它关联的着色器资源视图”。例如，我们会说：“将纹理绑定到管线上”，但实际上是指“将它的视图绑定到管线上”。

例如，要从图像文件 WoodCrate01.dds 创建一个纹理，我们可以编写如下代码：

```
ID3D11ShaderResourceView* mDiffuseMapSRV;
HR(D3DX11CreateShaderResourceViewFromFile(md3dDevice,
L"WoodCrate01.dds", 0, 0, &mDiffuseMapSRV, 0 ));
```

在纹理载入之后，我们需要把它指定给一个 effect 变量，使像素着色器中的代码可以访问到这个资源。在.fx 文件中，2D 纹理对象由 **Texture2D** 类型表示；例如，我们可以在 effect 文件中声明一个纹理变量：

```
// Nonnumeric values cannot be added to a cbuffer.
Texture2D gDiffuseMap;
```

如注释所述:“非数值对象不能放在常量缓冲区中”,纹理对象必须放在常量缓冲区之外。我们可以在 C++应用程序代码中获取这个 **Texture2D** 对象的指针（它是一个着色器资源变量）：

```
ID3D11EffectShaderResourceVariable* mfxDiffuseMapVar;
mfxDiffuseMapVar                                                    =
mFX->GetVariableByName("gDiffuseMap")->AsShaderResource();
```

在我们获取 **Texture2D** 对象的指针之后，可以通过如下 C++接口更新纹理对象：

```
// Set the C++ texture resource view to the effect texture variable.
mfxDiffuseMapVar->SetResource(mDiffuseMapSRV);
```

与其他的 effect 效果变量相同，当我们在绘图调用之间更新效果变量时，一定要调用 **Apply** 方法：

```
// set crate texture
mfxDiffuseMapVar->SetResource(mCrateMapSRV);
pass->Apply(0, md3dImmediateContext);
DrawCrate();

// set grass texture
mfxDiffuseMapVar->SetResource(mGrassMapSRV);
pass->Apply(0, md3dImmediateContext);
```

```
DrawGrass();

// set brick texture
mfxDiffuseMapVar->SetResource(mBrickMapSRV);
pass->Apply(0, md3dImmediateContext);
DrawBricks();
```

使用纹理贴图集可以在一次绘制调用中绘制更多的几何体，所以可以提高性能。例如，我们使用了如图 8.3 所示的包含条板、草地、砖墙纹理的贴图集，通过调整纹理坐标在每个对象上映射不同的子纹理，我们就可以在一次绘制调用中绘制整个几何体（假设每个对象都没有需要改变的其他参数）：

```
//  set texture atlas
mfxDiffuseMap ->SetResource(mAtlasSRV);
pass ->Apply(0, md3dImmediateContext);
DrawCrateGrassAndBricks();
```

调用绘制需要耗费资源，使用这个技术可以使绘制调用减到最少。

**注意**：纹理资源实际上可以由任何着色器（顶点、几何或像素）使用。但是目前我们只在像素着色器中使用纹理。如前所述，纹理本质上是一种特殊的数组，所以不难想象，这些数组数据也可以在顶点着色器和几何着色器中使用。

# 8.4 过滤器

## 8.4.1 倍增

纹理贴图元素应该被视为在一个连续图像上的离散颜色采样；不应该被视为矩形区域。那么问题是：当我们指定的纹理坐标$(u,v)$与任何一个纹理元素点都不对应时会产生什么结果？这一问题会发生在如下情景中：当观察点离场景中的一面墙很近时，墙会被放大，以至于会盖住整个屏幕。如果显示器的分辨率为 1024×1024，墙体纹理的分辨率为 256×256，那么就会出现倍增（magnification）问题——我们将要用很少的纹理元素来覆盖很多的像素。在本例中，每个纹理元素要覆盖 4 个像素。当顶点纹理坐标在三角形表面上进行插值时，每个像素都会得到一对唯一的纹理坐标。所以，像素的纹理坐标不会与任何一个纹理元素点对应。 我们可以使用插值方法来估算纹理元素之间的颜色。图形硬件提供了两种插值方法：常量插值和线性插值。线性插值是我们最常用的插值方法。

图 8.5 说明了在 1D 空间中的插值方法：假设有一个包含 256 个采样点的 1D 纹理以及一个插值纹理坐标 $u = 0.126484375$。那该纹理坐标对应的纹理元素为 $0.126484375 \times 256 = 32.38$。当然，这个值位于两个纹理采样点之间，我们必须使用插值来估算它。

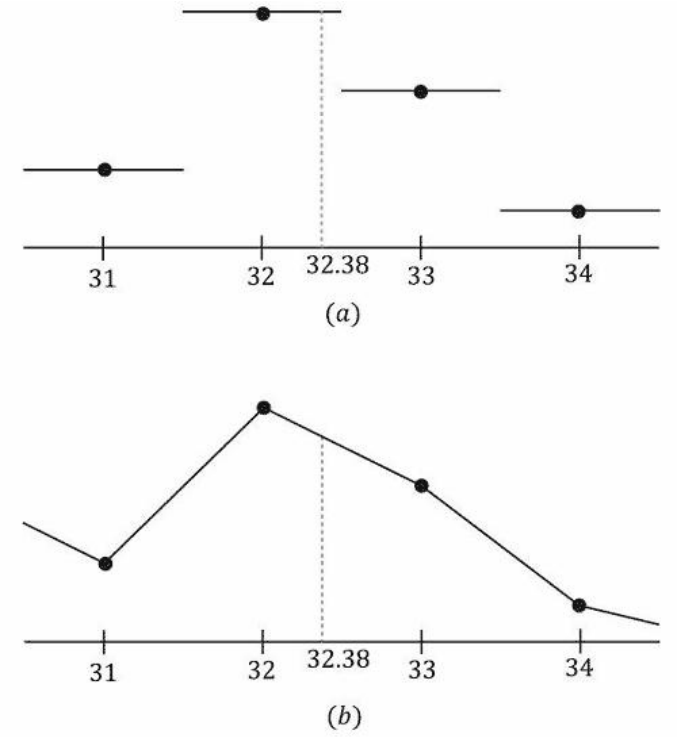



**图 8.5 (a)给出纹理元素点，我们构造一个分段常量函数来估算纹理元素点之间的值；这种方法有时也称为最近邻接点采样（nearest neighbor point sampling），因为它总是用最近的纹理元素点的值作为采样结果。(b)给出纹理元素点，我们构造一个分段线性函数来估算纹理元素点之间的值。**

2D 线性插值也称为双线性插值（bilinear interpolation），如图 8.6 所示。给出一对位于 4 个纹理元素之间的纹理坐标，我们在 *u* 方向上进行两次 1D 线性插值，然后在 *v* 方向上进行一次 1D 线性插值。

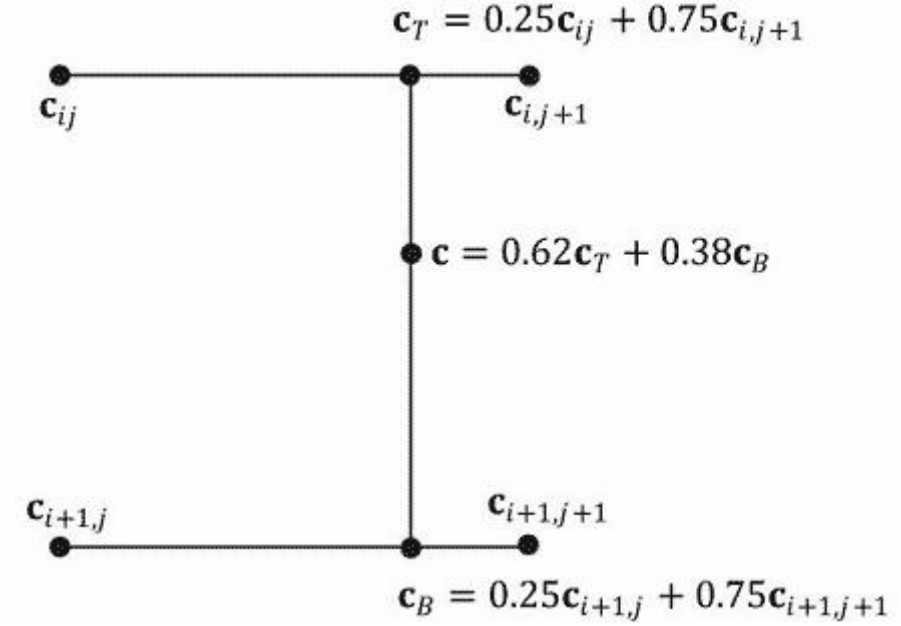



**图 8.6 这里有 4 个纹理元素点 $c_{ij}$、$c_{i,\,j+1}$,$c_{i+1,\,j}$、$c_{i+1,\,j+1}$。我们想要估算 c 点的颜色，它位于这 4 个纹理元素点之间。在本例中，c 位于 $c_{ij}$ 右侧 0.75 单位、$c_{ij}$ 下方 0.38 单位处。我们先在上面的两个颜色之间进行 1D 线性插值得到 $c_T$。然后，在下面的两个颜色之间进行 1D 线性插值得到 $c_B$。最后，在 $c_T$ 和 $c_B$ 之间进行线性插值得到 c。**

图 8.7 说明了常量插值和线性插值之间的区别。可以看到，常量插值会使图像出现明显的块状。而线性插值较为平滑，但是与真实数据（例如，一幅高分辨率的纹理）相比，通过插值得到的衍生数据仍然不够理想。

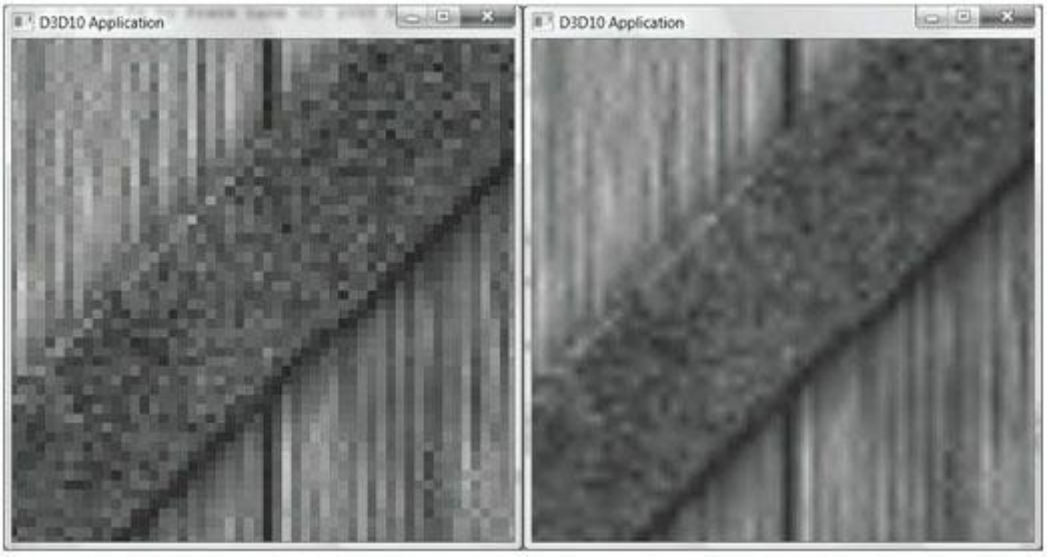



**图 8.7 我们在一个立方体放大一幅板条箱纹理，使倍增问题出现。左图使用常量插值，可以看到它产生了明显的斑块；这很容易理解，因为插值函数是离散的（图 8.5a），它所形成的颜色过渡显得生硬而不平滑。右图使用线性插值，由于插值函数是连续的，所以它产生的图像较为平滑。**

需要强调的是，由于在交互式 3D 程序中观察点的位置可以自由移动，所以我们对于倍增问题没有一种非常彻底的解决办法。在一定距离内，纹理可以有较好的效果，但是当观察点与物体之间的距离越来越近时，效果会急转直下。使用更高分辨率的纹理可以缓解一问题。

**注意**：在纹理映射的环境下，使用常量插值来求解纹理元素之间的颜色值的过程也称为点过滤（point filtering），使用线性插值来求解纹理元素之间的颜色值的过程也称为线性过滤（linear filtering）。点过滤和线性过滤是 Direct3D 使用的术语。

## 8.4.2 缩减

缩减（minification）与倍增的情况恰好相反。在缩减中，较多的纹理元素会被映射为较少的像素。例如，考虑下面的情景：我们将一幅 256×256 的纹理映射到墙体上。然后把观察点对准墙体，并向后移动观察点，使墙体逐渐变小，直到墙体在屏幕上只占 64×64 的像素区域为止。现在，我们要把 256×256 个纹理元素映射为 64×64 个屏幕像素。在这一情景中，像素的纹理坐标不会与纹理贴图中的任何一个纹理元素对应。常量和线性插值过滤器仍然适用于缩减情况。不过，缩减的处理工作稍多一些。简单的讲，我们要把 256×256 个纹理元素均匀地缩减为 64×64 个纹理元素。多级渐近贴图映射（mipmapping）为这一工作提供了一种高效的估值手段，只是要额外占用一些内存。在初始化时（或创建资源时），通过对图像进行降阶采样生成纹理的多个缩略版本来创建多级渐近纹理链（mipmap chain，参见图 8.8）。求平均值的工作是根据多级渐近贴图的大小提前计算出来的。在运行时，图形硬件会根据程序员指定的多级渐近贴图参数执行两种不同的操作：

1．为纹理映射挑选一个与屏幕几何体分辨率最匹配的多级渐近纹理层，根据需要在多级渐近纹理层上使用常量插值或线性插值。这种用于多级渐近纹理的操作称为点过滤（point filtering），因为它与常量插值很像——它只为纹理映射挑选一个最接近的多级渐近纹理层。

2．为纹理映射挑选两个与屏幕几何体分辨率最匹配的多级渐近纹理层（其中，一个比屏幕几何体分辨率大一些，另一个比屏幕几何体分辨率小一些）。然后，在这两个多级渐近纹理层上使用常量插值或线性插值，分别取出一个纹理颜色。最后，在这两个纹理颜色之间进行插值。这种用于多级渐近纹理的操作称为线性过滤（linear filtering），因为它与线性插值很像——它在两个最匹配的多级渐近纹理层之间进行线性插值。

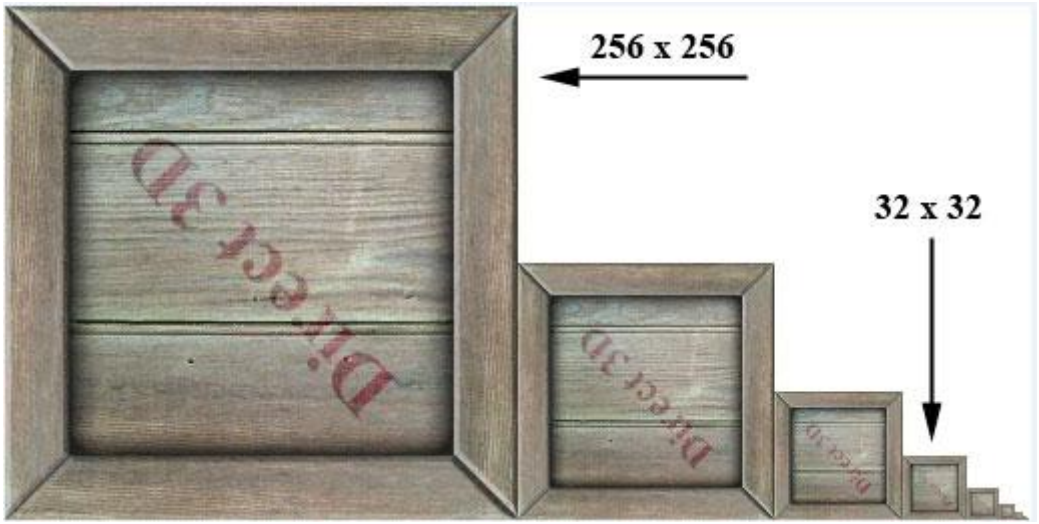



**图 8.8 多级渐近纹理链；每个后续多级渐近纹理层的尺寸都是前一层的 1/2，纹理层的下限尺寸是 1×1。**

通过在多级渐近纹理链中选择最佳纹理层，可以使缩减操作的总开销降至最低。

## 8.4.2.1 创建多级渐近纹理

多级渐近纹理层即可以由美术师手工创建，也可以由过滤算法生成。

DDS 图像格式（DirectDraw Surface 格式）可以直接把多级渐近纹理层存储在文件中；在这种情况下，只需要简单的读入数据——不需要在运行执行任何与多级渐近纹理层相关的算法。DirectX 纹理工具可以为纹理生成多级渐近纹理链，并导出 DDS 文件。当一个图像文件不包含完整的多级渐近纹理链时，函数 **D3DX11CreateShaderResourceViewFromFile** 或 **D3DX11CreateTextureFromFile** 会使用指定的过滤算法创建一个多级渐近纹理链（请参阅 SDK 文档中 **D3DX11_IMAGE_LOAD_INFO**，尤其是要注意 **MipFilter** 数据成员的描述）。我们可以看到，多级渐近纹理映射基本上是自动的。当资源文件不包含多级渐近纹理链时，**D3DX11** 会为我们自动生成一个纹理链。只要启用了多级渐近纹理映射，硬件就会在运行时自动选择正确的多级渐近纹理层。

**注意**：有时通用的过滤算法可能会丢失你想要保留的细节。例如，在图 8.7 中，木板上的文字“Direct3D”在最低等级的渐进纹理中非常模糊。如果这不可接受，美工就需要手动创建/调整渐进层次以保留重要的细节。

## 8.4.3 各向异性过滤

我们还可以使用各向异性过滤（anisotropic filter）。当多边形的法线向量与摄像机的观察向量夹角过大时（例如，当多边形垂直于观察窗口时），这种过滤可以有效缓解图像的失真问题。只是它的资源占用量较大。不过，当你需要对失真进行校正时，这些消耗是值得的。

图 8.9 是对各向异性过滤和线性过滤的一个比较。

**图 8.9 板条箱的顶面几乎垂直于观察窗口。（左图）使用线性过滤，板条箱的顶面显得极其模糊。（右图）在同样的角度上，使用各向异性过滤得到的渲染结果要好很多。**

# 8.5 纹理采样

我们知道，**Texture2D** 对象用于在 effect 文件中表示纹理。不过，还有一种与纹理相关的 **SamplerState**（采样器）对象。它用于描述如何使用过滤器访问纹理资源。下面是它的一些例子：

```
// 在倍增、缩减、多级渐进纹理上使用线性过滤。
SamplerState mySampler0
{
    Filter = MIN_MAG_MIP_LINEAR;
};

// 在缩减上使用线性过滤，倍增和多级渐进纹理上使用点过滤。
SamplerState mySampler1
{
    Filter = MIN_LINEAR_MAG_MIP_POINT;
};

// 在缩减上使用点过滤，倍增上使用线性过滤，多级渐进纹理上使用点过滤。
SamplerState mySampler2
{
    Filter = MIN_POINT_MAG_LINEAR_MIP_POINT;
};

// 在倍增、缩减、多级渐进纹理上使用各向异性过滤。
SamplerState mySampler3
{
    Filter = ANISOTROPIC;
    MaxAnisotropy = 4;
};
```

注意，对于各向异性过滤，我们必须指定各向异性的最大值，这个值介于 1 至 16 之间。大数值需要耗费更多的资源，但效果更好。你可以从这些例子中猜到一些其他的标志值，或者你可以在 SDK 文档中查找 **D3D11_FILTER** 枚举类型。我们很快就会看到与采样器相关的

其他属性，但是对于我们的第一个演示程序来说现在的这些内容已经足够了。

现在，传入像素着色器的每个像素都有一对纹理坐标，我们使用如下语法完成实际的纹理采样工作：

```
Texture2D gDiffuseMap;

struct VertexOut
{
    float4 posH : SV_POSITION;
    float3 posW : POSITION;
    float3 normalW : NORMAL;
    float2 Tex : TEXCOORD;
};

SamplerState samAnisotropic
{
    Filter = ANISOTROPIC;
    MaxAnisotropy = 4;

};
float4 PS(VertexOut pin, uniform  int gLightCount) : SV_Target
{
    // 从纹理中提取颜色
    float4 texColor = gDiffuseMap.Sample(samAnisotropic, pin.Tex);
    ...
```

可以看到，我们使用 **Texture2D::Sample** 方法对一个纹理进行采样。通过第一个参数传递 SamplerState 对象，通过第二个参数传递像素的纹理坐标$(u,v)$。该方法使用由 **SamplerState** 对象指定的过滤方式从纹理贴图中返回位于点$(u,v)$处的插值颜色。

**注意**：HLSL 类型 **SamplerState** 对应 **ID3D11SamplerState** 接口。采样状态也可以在应用程序中由 **ID3DX11EffectSamplerVariable::SetSampler** 方法进行设置。见 **D3D11_SAMPLER_DESC** 和 **ID3D11Device::CreateSamplerState**。和渲染状态一样，必须在初始化阶段就创建采样状态。

# 8.6 把纹理作为材质

要将纹理整合到我们的材质/光照系统中，通常可以将环境光和漫反射光项调制到纹理颜色上，但不需要包括高光项（这常常被称为“后期添加调制，modulate with late add”）。

```
//  Modulate with late add.
litColor = texColor*(ambient + diffuse) + spec;
```

上述操作给每个像素添加了环境光和漫反射光的材质颜色，像素材质比顶点材质的分辨率更好，因为会有许多纹理元素映射到三角形上。也就是，每个像素都会得到一个插值后的纹理坐标$(u,v)$；这些纹理坐标可以用于纹理采样，为每个像素估算出一个表面颜色。

# 8.8 寻址模式

纹理必须与常量插值或线性插值一起使用以形成一个向量值函数 $T(u,v) = (r,g,b,a)$。也就是，当给定一个纹理坐标$(u,v) \in [0,1]^2$时，纹理函数 $T$ 返回颜色$(r,g,b,a)$。Direct3D 允许我们以 4 种不同的方式扩展该函数的值域（称为寻址模式）：重复（wrap）、边框颜色（border color）、截取（clamp）和镜像（mirror）。

**1．重复**：通过在每个整点连接处重复图像来扩展纹理（参见图 8.10）。

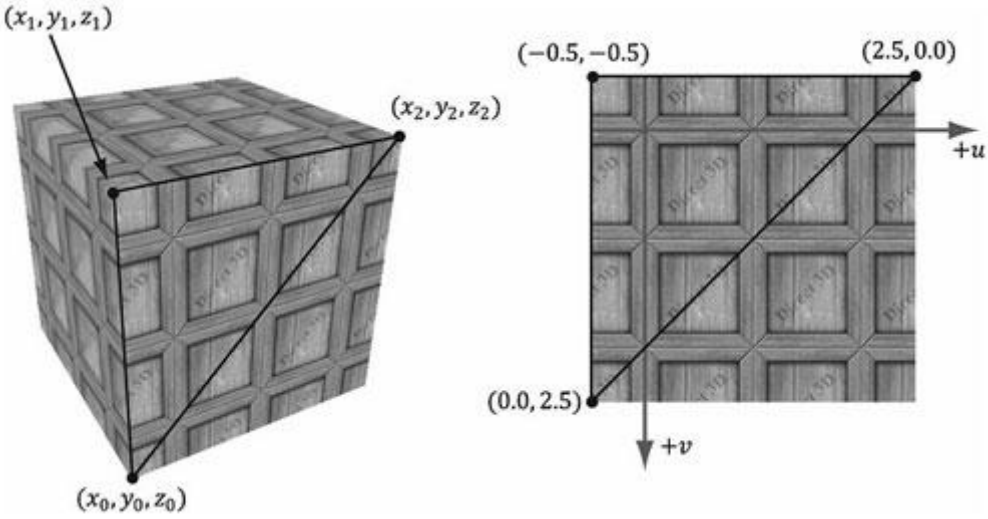



**图 8.10：重复寻址模式。**

**2．边框颜色**：通过将每个不在$[0,1]^2$区间内的$(u,v)$映射为程序员指定的某个颜色来扩展纹理（参见图 8.11）。

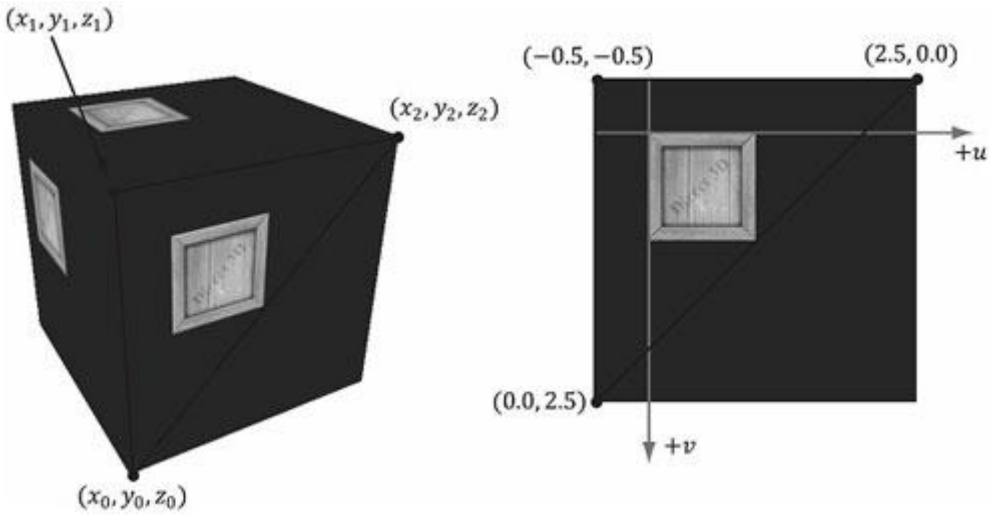



**图 8.11 边框颜色寻址模式。**

**3．截取**：通过将每个不在$[0,1]^2$区间内的$(u,v)$映射为颜色 $T(u_0,v_0)$来扩展纹理。其中，$(u_0,v_0) \in [0,1]^2$，$(u_0,v_0)$是与$(u,v)$距离最近的点（参见图 8.12）。

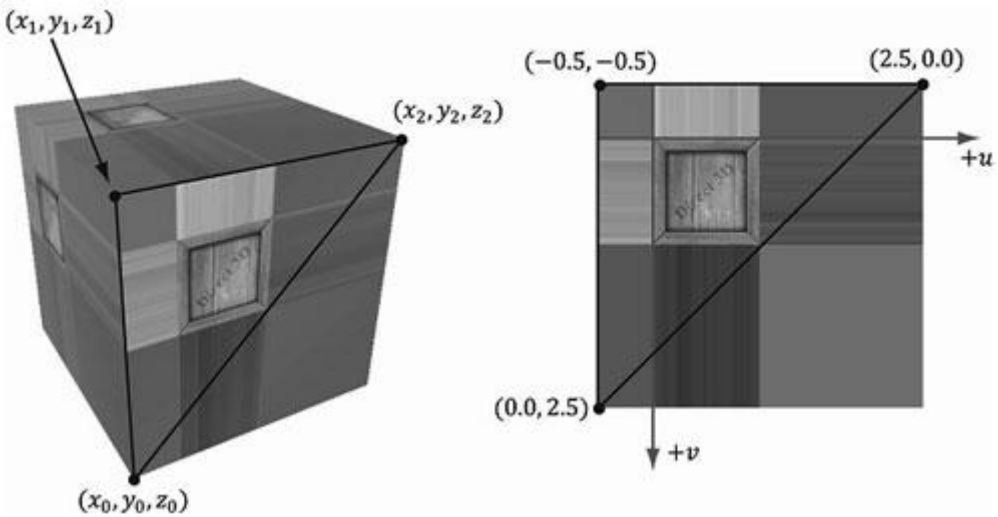



**图 8.12 截取寻址模式。**

**4．镜像**：通过在每个整点连接处镜像图像来扩展纹理（参见图 8.13）。

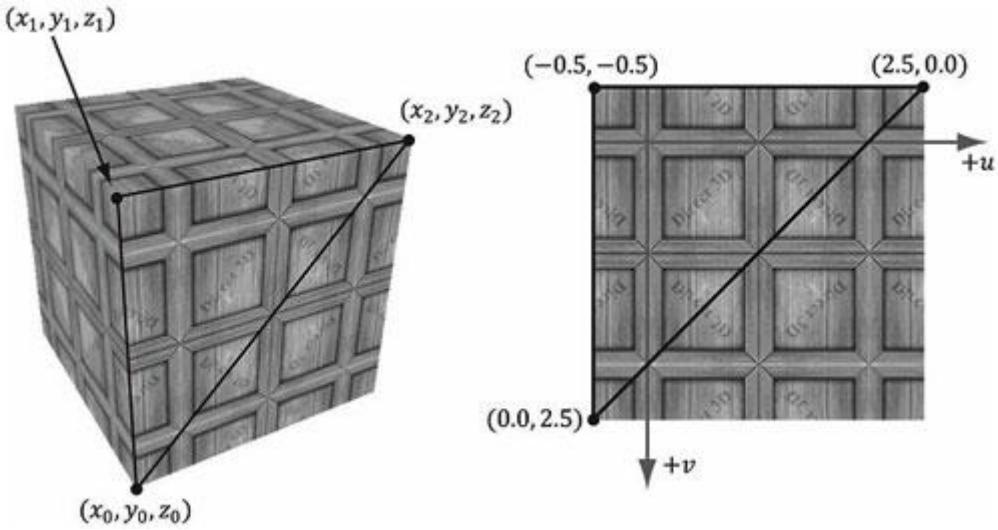



**图 8.13 镜像寻址模式。**

由于寻址模式始终驻留在 Direct3D 设备中（重复模式为默认值），所以当纹理坐标超出[0,1]区间时，我们仍然可以得到采样颜色。

重复寻址模式可能是最常用的寻址模式；因为它可以把纹理平铺在物体表面上，使我们在不提供额外数据的情况下有效地提高纹理分辨率（虽然些额外的分辨率是重复的）。当使用平铺时，我们通常希望纹理是无缝的（seamless）。例如，板条箱纹理不是无缝的，因为你可以清楚地看到它的每个副本（参见图 8.10）。图 8.14 展示了将一幅无缝的墙砖纹理在平铺 2×3 次后的结果。

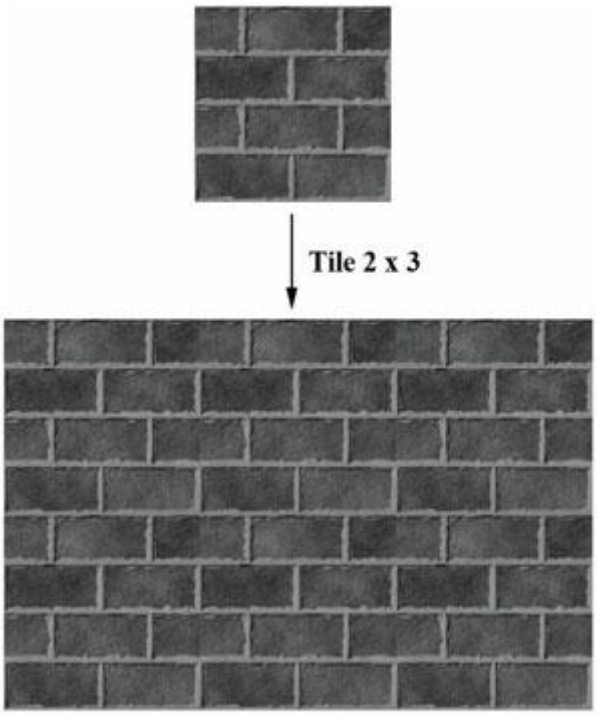



**图 8.14 将一幅墙砖纹理平铺 2×3 次。由于该纹理是无缝的，所以很难一下就看出它的重复图案。**

我们在采样器对象中指定寻址模式。下面是图 8.10 到 8.13 对应的寻址模式参数：

```
SamplerState samTriLinear
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

SamplerState samTriLinear
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = BORDER;
    AddressV = BORDER;
    // Blue border color
    BorderColor = float4(0.0f, 0.0f, 1.0f, 1.0f);
};

SamplerState samTriLinear
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = CLAMP;
    AddressV = CLAMP;
};
```

```
SamplerState samTriLinear
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = MIRROR;
    AddressV = MIRROR;
};
```

**注意**：可以单独控制 *u*、*v* 方向上的寻址模式。读者可以亲自试验一下。

# 8.9 对纹理进行变换

如前所述，纹理坐标表示纹理平面上的 2D 点。因此，我们可以像使用其他坐标一样，对纹理坐标进行平移、旋转和缩放。下面是一些会对纹理进行变换的例子：

1．沿着墙体拉伸一幅砖块纹理。该墙体顶点的纹理坐标在[0,1]区间内。我们将每个纹理坐标乘以 4，使区间扩大为[0,4]，让纹理在墙体上重复 4×4 次。

2．在一片晴朗的蓝天上（即，在一个天空球上）拉伸一幅白云纹理。通过一个时间函数控制纹理坐标的平移，形成白云在天上飘动的效果。

3．当实现粒子效果时，有时需要对纹理坐标进行旋转；例如，随着时间的推移旋转一幅火球纹理。

纹理坐标变换与普通坐标变换的实现方式相同。我们指定一个变换矩阵，将纹理坐标向量与该矩阵相乘。例如：

```
// 常量缓冲变量
float4x4 gTexMtx;

// 位于 shader 程序中的代码
vOut.texC = mul(float4(vIn.texC, 0.0f, 1.0f),gTexMtx);
```

注意，由于我们使用的是 2D 纹理坐标，所以只需要对前两个坐标进行变换。例如，当纹理矩阵对 *z* 坐标进行平移时，它不会对纹理坐标产生任何影响。

# 8.10 地形纹理演示程序

在本例中，我们要为地形和水体添加纹理。首先，我们要在地形上平铺一幅草地纹理。由于地形网格很大，如果我们直接拉伸纹理，那么每个三角形只能得到很少的几个纹理元素。换句话说，这里无法为表面提供足够高的纹理分辨率；我们会受到倍增问题的影响。所以，我们要在地面网格上平铺草地纹理，进而获得较高的分辨率。其次，我们要通过一个时间函数对水体纹理进行平移，使水体显得更真实一些。图 8.15 是该演示程序的屏幕截图。

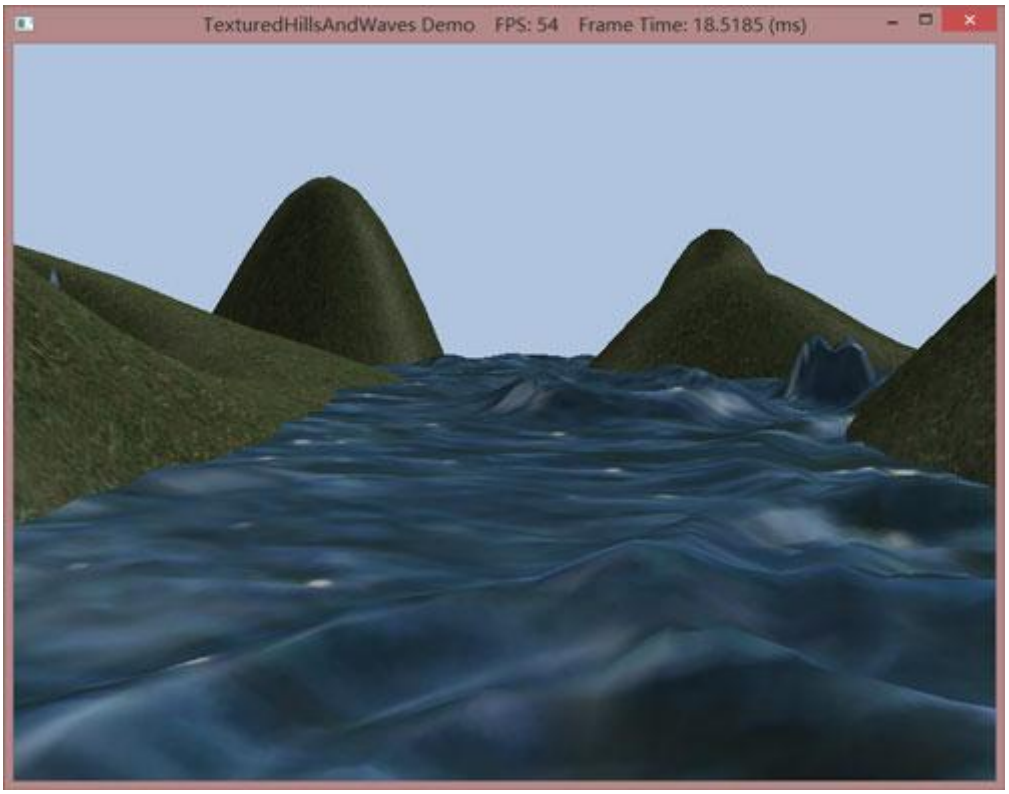



**图 8.15 地形纹理演示程序的屏幕截图。**

## 8.10.1 生成网格纹理坐标

图 8.16 是一个建立在 *xz* 平面上的 $m \times n$ 网格以及一个在规范化纹理空间$[0,1]^2$中的对应网格。可以看到，*xz* 平面上的第 *ij* 个网格顶点的纹理坐标对应于纹理空间中的第 *ij* 个网格顶点的坐标。第 *ij* 个顶点对应的纹理空间坐标为：

$u_{ij} = j \cdot \Delta u$

$v_{ij} = i \cdot \Delta v$

其中，$\Delta u = \frac{1}{n\text{-}1}$ ，$\Delta v = \frac{1}{m\text{-}1}$ 。

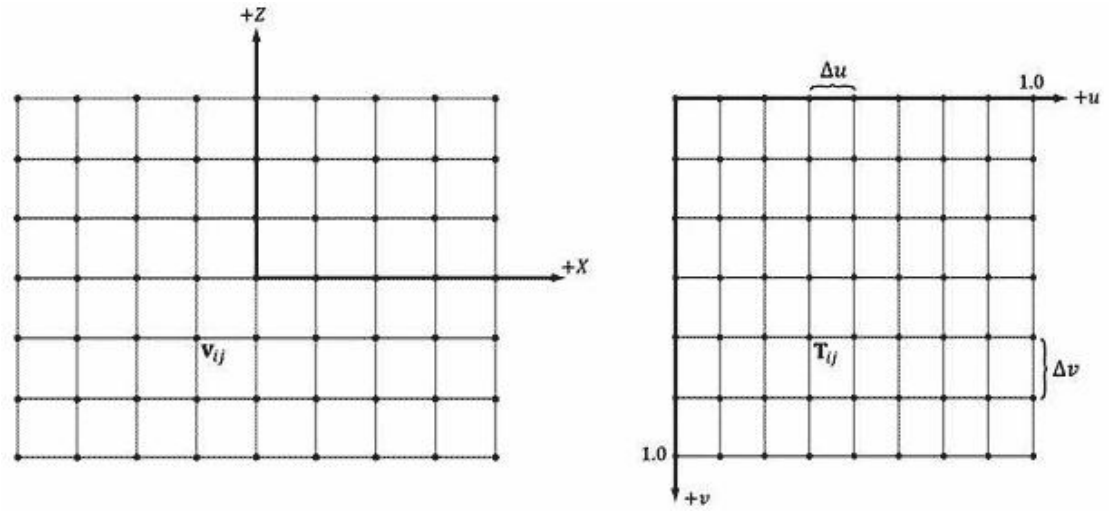



**图 8.16 空间中的网格顶点 $v_{ij}$ 的纹理坐标等于 *uv* 空间中的第 *ij* 个网格顶点 $T_{ij}$ 的坐标。**

因此，我们可以使用如下代码为地面网格生成纹理坐标：

```
void GeometryGenerator::CreateGrid(float width, float depth, UINT m,
UINT n, MeshData& meshData)
{
    UINT vertexCount = m*n;
    UINT faceCount   = (m-1)*(n-1)*2;

    //
    // 创建顶点
    //

    float halfWidth = 0.5f*width;
    float halfDepth = 0.5f*depth;

    float dx = width / (n-1);
    float dz = depth / (m-1);

    float du = 1.0f / (n-1);
    float dv = 1.0f / (m-1);

    meshData.Vertices.resize(vertexCount);
    for(UINT i = 0; i < m; ++i)
    {
        float z = halfDepth - i*dz;
        for(UINT j = 0; j < n; ++j)
        {
            float x = -halfWidth + j*dx;

            meshData.Vertices[i*n+j].Position = XMFLOAT3(x, 0.0f, z);
            meshData.Vertices[i*n+j].Normal   = XMFLOAT3(0.0f, 1.0f,
0.0f);
            meshData.Vertices[i*n+j].TangentU = XMFLOAT3(1.0f, 0.0f,
0.0f);

            // 在网格上拉伸纹理
            meshData.Vertices[i*n+j].TexC.x = j*du;
            meshData.Vertices[i*n+j].TexC.y = i*dv;
        }
    }

    …
}
```

### 8.10.2 纹理平铺

前面提到，我们希望在地形网格上平铺一幅草地纹理。但是，目前计算出来的纹理坐标是在单位区间$[0,1]^2$中的，无法产生平铺。所以，我们要指定重复寻址模式并通过一个纹理变换矩阵将纹理坐标扩大 5 倍。这样，纹理坐标会被映射到$[0,5]^2$区间内，使纹理在地形网格表面平铺 5×5 次：

```
XMMATRIX grassTexScale = XMMatrixScaling(5.0f, 5.0f, 0.0f);
XMStoreFloat4x4(&mGrassTexTransform, grassTexScale);
…
Effects::BasicFX->SetTexTransform(XMLoadFloat4x4(&mGrassTexTransf
orm));
…
activeTech->GetPassByIndex(p)->Apply(0, md3dImmediateContext);
md3dImmediateContext->DrawIndexed(mLandIndexCount, 0, 0);
```

### 8.10.3 纹理动画

我们要通过一个位于 **UpdateScene** 方法中的时间函数在纹理空间中平移纹理坐标，使水体纹理在网格上移动。我们为每帧提供一个很小的位移量，以得到一个平滑动画。我们同时使用无缝纹理和重复寻址模式，以使纹理坐标在平移时不出现间断。下面的代码示范了如何为水体纹理计算位移量，并生成和设定水体的纹理矩阵：

```
// 平铺水面纹理
XMMATRIX wavesScale = XMMatrixScaling(5.0f, 5.0f, 0.0f);

// 根据时间平移纹理
mWaterTexOffset.y += 0.05f*dt;
mWaterTexOffset.x += 0.1f*dt;
XMMATRIX wavesOffset = XMMatrixTranslation(mWaterTexOffset.x,
mWaterTexOffset.y, 0.0f);

// 组合缩放和平移
XMStoreFloat4x4(&mWaterTexTransform, wavesScale*wavesOffset);
…
Effects::BasicFX->SetTexTransform(XMLoadFloat4x4(&mWaterTexTransf
orm));
…
activeTech->GetPassByIndex(p)->Apply(0, md3dImmediateContext);
```

## 8.11 压缩纹理格式

一个虚拟场景可能会载入数百幅纹理，而些纹理会占用大量的 GPU 内存（记住，我们

必须让所有的纹理驻留在 GPU 内存中，只有这样着色器才能快速地访问纹理）。为了缓解 GPU 内存压力，Direct3D 提供了以下压缩纹理格式：BC1、BC2、BC3、BC4、BC5，BC6 和 BC7。

1．BC1（**DXGI_FORMAT_BC1_UNORM**）：该格式支持 3 个颜色通道，仅用 1 位（开/关）表示 alpha 分量。

2．BC2（**DXGI_FORMAT_BC2_UNORM**）：该格式支持 3 个颜色通道，仅用 4 位表示 alpha 分量。

3．BC3（**DXGI_FORMAT_BC3_UNORM**）：该格式支持 3 个颜色通道，以 8 位表示 alpha 分量。

4．BC4（**DXGI_FORMAT_BC4_UNORM**）：该格式支持 1 个颜色通道（例如，灰阶图像）。

5．BC5（**DXGI_FORMAT_BC5_UNORM**）：该格式支持两个颜色通道。

6．BC6（**DXGI_FORMAT_BC6_UF16**）：该格式用于压缩的 HDR（高动态范围，high dynamic range）图像数据。

7．BC7（**DXGI_FORMAT_BC7_UNORM**）：该格式用于高质量的 RGBA 压缩。特别的有，这个格式可以显著地减少由于压缩法线贴图带来的错误。

关于这些格式的更多信息，请读者在 SDK 文档的索引中查找“Block Compression（块压缩）”。

**注意**：压缩纹理只能作为输入数据传递给渲染管线的着色器阶段。

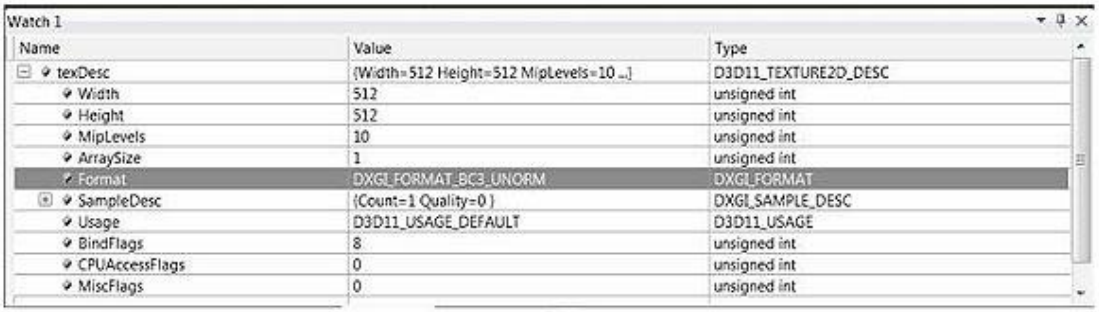
**注意**：因为块压缩算法使用 4×4 像素块，所以纹理尺寸必须为 4 的倍数。

使用这些格式的好处是它们可以压缩存储在 GPU 内存中，当使用时由 GPU 实时解压缩。

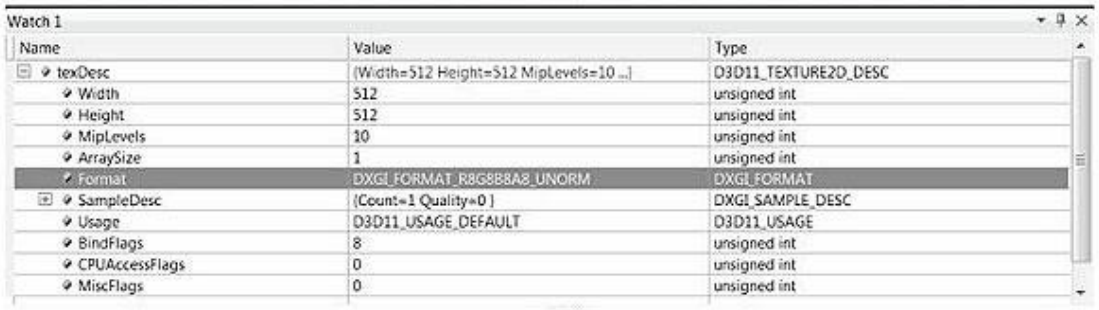
我们可以在载入纹理时使用 **D3DX11CreateShaderResourceViewFromFile** 函数的 **pLoadInfo** 参数，让 Direct3D 把纹理转换为某种压缩格式。例如下面的代码，它载入了一个 BMP 文件：

```
D3DX11_IMAGE_LOAD_INFO loadInfo;
loadInfo.Format = DXGI_FORMAT_BC3_UNORM;

HR(D3DX11CreateShaderResourceViewFromFile(md3dDevice,
    L"Textures/darkbrick.bmp", &loadInfo, 0, &mDiffuseMapSRV, 0 ));

// 从资源视图获取 2D 纹理
ID3D11Texture2D* tex;
mDiffuseMapSRV->GetResource((ID3D11Resource**)&tex);

// 从 2D 纹理获取纹理描述
D3D11_TEXTURE2D_DESC texDesc;
tex->GetDesc(&texDesc);
```

图 8.17a 是在调试器中看到的 **texDesc** 值；它包含了我们指定的压缩纹理格式。当参数 **pLoadInfo** 设为空值时，Direct3D 将使用源图像格式（图 8.17b），即非压缩格式 **DXGI_FORMAT_R8G8B8A8_UNORM**。




(a)




(b)

**图 8.17 (a) 使用压缩格式 DXGI_FORMAT_BC3_UNORM 创建纹理。(b)使用非压缩格式 DXGI_FORMAT_R8G8B8A8_UNORM 创建纹理。**

另外，你也可以使用 DDS（DirectDraw Surface）格式直接存储压缩纹理。操作步骤是运行 SDK 目录 D:\Microsoft DirectX SDK(June 2010)\Utilities\Bin\x86 中的 DirectX 纹理工具（DXTex.exe），打开你的图像文件。然后执行菜单命令 **Menu>Format>Change Surface Format**，选择 DXT1、DXT2、DXT3、DXT4 或 DXT5，并保存 DDS 文件。这些格式其实是 Direct3D 9 的压缩纹理格式，DXT1 相当于 BC1，DXT2 和 DXT3 相当于 BC2，DXT4 和 DXT5 相当于 BC3。例如，当我们用 **D3DX11CreateShaderResourceViewFromFile** 函数载入一个 DXT1 格式的 DDS 文件时，它的实际纹理格式为 DXGI_FORMAT_BC1_UNORM：

```
HR(D3DX11CreateShaderResourceViewFromFile(md3dDevice,
    L"Textures/darkbrick.dds", 0, 0, &mDiffuseMapSRV, 0 ));

// 从资源视图获取 2D 纹理
ID3D11Texture2D* tex;
mDiffuseMapSRV->GetResource((ID3D11Resource **)&tex);

// 从 2D 纹理获取纹理描述
D3D11_TEXTURE2D_DESC texDesc;
tex->GetDesc(&texDesc);
```

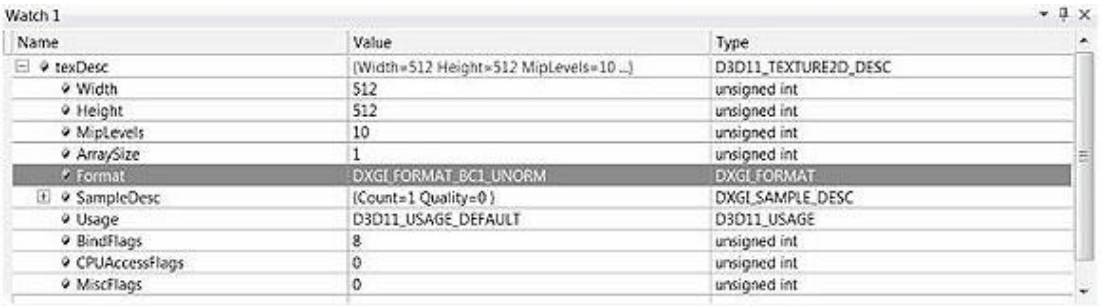


**图 8.18：使用 DXGI_FORMAT_BC1_UNORM 格式创建纹理。**

注意，如果 DDS 文件使用了某种压缩格式，那么我们可以将 **pLoadInfo** 参数设为空值，**D3DX11CreateShaderResourceViewFromFile** 会自动使用由文件指定的压缩格式。

对于 BC4 和 BC5 格式，你可以使用 NVIDIA Texture Tools（http://code.google.com/p/nvidia-texture-tools/ ）。对于 BC6 和 BC7 格式，DirectX SDK 包含了一个叫做“BC6HBC7EncoderDecoder11”的示例。这个程序可以用来将纹理转换为 BC6 或 BC7 格式。这个示例包含了完整的源代码，所以你可以将它整合到你自己的素材管道中。而且，若你的显卡支持计算着色器，这个示例还会使用 GPU 进行转换工作，这比通过 CPU 进行转换快得多。

你还可以用 DirectX 纹理工具生成多级渐近纹理层（**Menu>Format>Generate Mip Maps**），并保存为 DDS 文件。通过一方式，多级渐近纹理层可以被提前计算出来并保存在文件中，节省载入时的计算时间（它们只需要被载入 。

将纹理存储为 DDS 压缩文件的另一个好处是可以减少磁盘空间的占用量。

# 8.12 小结

纹理坐标用于定义纹理上的 2D 三角形，以使纹理映射到 3D 三角形表面。

我们可以使用 **D3DX11CreateShaderResourceViewFromFile** 函数从存储在磁盘上的图像文件创建纹理。

我们可以通过采样器状态来指定纹理的过滤方式：缩减、倍增和多级渐近纹理采样。

寻址模式定义了当纹理坐标超出[0,1]区间时 Direct3D 的处理方式。例如，纹理会被平铺、镜像、截取，还有一个叫什么来着？

纹理坐标可以像其他坐标一样进行缩放、旋转和平移。通过在每帧为纹理坐标增加一个很小的位移量，可以生成纹理动画。

通过使用 Direct3D 纹理压缩格式 BC1、BC2、BC3、BC4、BC5，BC6 和 BC7，可以节约大量的 GPU 内存。使用 DirectX Texture Tool 可以生成 BC1、BC2、和 BC3 格式的纹理。使用 NVIDIA Texture Tools（http://code.google.com/p/nvidia-texture-tools/）生成 BC4 和 BC5 格式的纹理。使用 SDK 中“BC6HBC7EncoderDecoder11”示例生成 BC6 和 BC7 格式的纹理。

# 9.1 混合方程

考虑图 9.1。我们从地形和板条箱开始渲染场景，先绘制地形，再绘制板条箱，使地形和板条箱的像素依次存入后台缓冲区。然后使用混合（blending），将水体绘制到后台缓冲区，使水体像素和后台缓冲区中的地形、板条箱像素融为一体。通过一方式，我们可以透过水体看到地形和板条箱。本章我们主要讲解混合技术，它可以将当前的光栅化像素（也称为源像素）与后台缓冲区中的像素（也称为目标像素）混合（融合）在一起。该技术通常用于渲染半透明物体，比如水和玻璃。

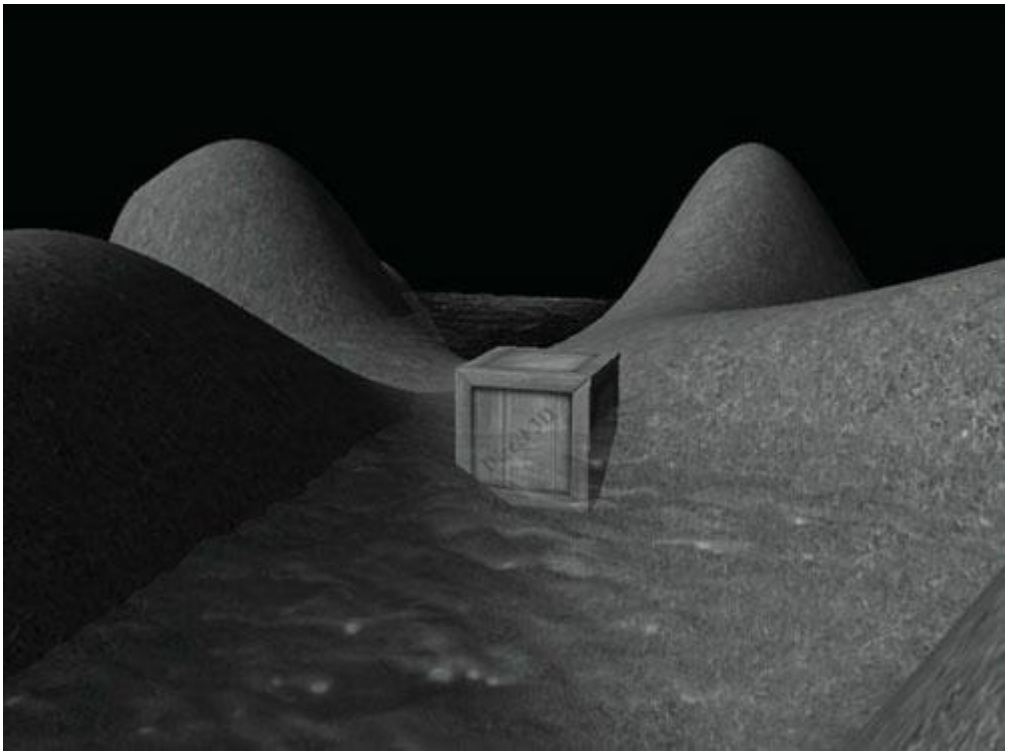


**图 9.1 半透明的水面。**

**注意**：为了叙述方便，当我们提到后台缓冲时，指的是渲染目标。但是，我们也可以绘制到一个“离屏，off screene”渲染目标中。混合作用在这些渲染目标的方式是相同的，目标像素的值就是之前光栅化后的离屏渲染目标中的值。

**学习目标**

1．理解混合的工作原理以及如何在 Direct3D 中使用混合。

2．学习 Direct3D 支持的各种混合模式。

3．了解如何通过 alpha 分量来控制图元的透明度。

4．学习如何使用 HLSL 的 **clip** 函数阻止像素绘制到后台缓冲区。

设 $\mathbf{C}_{src}$ 为当前正在进行光栅化处理的第 *ij* 个像素（源像素）的颜色，$\mathbf{C}_{dst}$ 为后台缓冲区中的第 *ij* 个像素（目标像素）的颜色。当不使用混合时，$\mathbf{C}_{src}$ 会覆盖 $\mathbf{C}_{dst}$ 的值（假设该值已通过深度/模板测试）并成为后台缓冲区中的第 *ij* 个像素的新颜色。但是当使用混合时，$\mathbf{C}_{src}$ 和 $\mathbf{C}_{dst}$ 会被组合为一个新颜色 **C** 并覆盖 $\mathbf{C}_{dst}$ 的值（即，混合颜色 **C** 会成为后台缓冲区中的第 *ij* 个像素的新颜色）。Direct3D 使用如下混合方程混合源像素和目标像素颜色：

$$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} \boxplus \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$$

$\mathbf{F}_{src}$（源混合系数）和 $\mathbf{F}_{dst}$（目标混合系数）可以是 9.3 节描述的任何一个值，它们按照各种不同的方式调整源像素和目标像素的比例，实现各种不同的混合效果。运算符⊗是定义在 5.3.1 节中定义的颜色的分量乘法；运算符⊞可以是 9.2 节描述的任何一个二进制运算符。

上述混合方程只负责处理颜色的 RGB 分量。alpha 分量要由另外一个方程来处理：

$$A = A_{src}F_{src} \boxplus A_{dst}F_{dst}$$

该方程与前一个方程基本相同，只是使用的混合系数和二进制运算符有些区别。我们将 RGB 和 alpha 分开的原因非常简单，就是为了独立地、按照不同的方式处理它们。

**注意**：我们很少混合 alpha 分量，多数情况下，我们只是对 RGB 分量进行混合。尤其是在本书中，我们从不混合源 alpha 值和目标 alpha 值，虽然 alpha 值会参与我们的 RGB 混合操作。这主要是因为我们暂时用不到后台缓冲区中的 alpha 值。只有当某些算法需要使用

目标 alpha 值时，后台缓冲区中的 alpha 值才有用。

# 9.2 混合操作

混合方程中的二进制运算符⊞可取以下各值之一：

```
typedef enum D3D11_BLEND_OP
{
    D3D11_BLEND_OP_ADD = 1,           C = Csrc ⊗ Fsrc + Cdst ⊗ Fdst
    D3D11_BLEND_OP_SUBTRACT = 2,      C = Cdst ⊗ Fdst - Csrc ⊗ Fsrc
    D3D11_BLEND_OP_REV_SUBTRACT = 3,  C = Csrc ⊗ Fsrc - Cdst ⊗ Fdst
    D3D11_BLEND_OP_MIN = 4,           C = min(Csrc , Cdst)
    D3D11_BLEND_OP_MAX = 5,           C = max(Csrc , Cdst)
} D3D11_BLEND_OP;
```

**注意**：在 min/max 操作中会忽略混合因子。

这些运算符也可用于 alpha 混合方程。注意，RGB 混合方程和 alpha 混合方程可以使用不同的运算符。例如，当两组 RGB 值相加时，对应的两组 alpha 值可以相减：

$$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} + \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$$

$$A = A_{dst}F_{dst} - A_{src}F_{src}$$

# 9.3 混合系数

通过为源混合系数和目标混合系数指定不同的组合值，可以实现不同的混合效果。我们将在 9.5 节讲解这里的一部分组合值，对于那些我们没有讲到的组合值，请读者自己做一些实验，看看它们可以产生哪些效果。下面的列表描述了基本的混合系数，它们都可以用于 $\mathbf{F}_{src}$ 和 $\mathbf{F}_{dst}$。对于一些额外的高级混合系数，请参阅 SDK 文档中的 **D3D11_BLEND** 枚举类型。设 $\mathbf{C}_{src} = (r,g,b)$、$A_{src} = a_s$（从像素着色器中输出的 RGBA 颜色）、$\mathbf{C}_{dst} = (r_d,g_d,b_d)$、$A_d = a_d$（已经储存在渲染目标中的 RGBA 颜色），**F** 既可以作为 $\mathbf{F}_{src}$ 也可以作为 $\mathbf{F}_{dst}$，$F$ 既可以作为 $F_{src}$ 也可以作为 $F_{dst}$，我们有：

- **D3D11_BLEND_ZERO**：$\mathbf{F} = (0, 0,0)$且 $F = 0$
- **D3D11_BLEND_ONE**：$\mathbf{F} = (1, 1,1)$ 且 $F = 1$
- **D3D11_BLEND_SRC_COLOR**：$\mathbf{F} = (r_s ,g_s ,b_s )$
- **D3D11_BLEND_INV_SRC_COLOR**：$\mathbf{F} = (1 - r_s ,1 - g_s ,1 - b_s )$
- **D3D11_BLEND_SRC_ALPHA**：$\mathbf{F} = (a_s ,a_s ,a_s )$且 $F = a_s$
- **D3D11_BLEND_INV_SRC_ALPHA**：$\mathbf{F} = (1 - a_s,1 - a_s,1 - a_s)$且 $F =1 - a_s$
- **D3D11_BLEND_DEST_ALPHA**：$\mathbf{F} = (a_d ,a_d ,a_d )$且 $F = a_d$
- **D3D11_BLEND_INV_DEST_ALPHA**：$\mathbf{F} = (1 - a_d,1 - a_d,1 - a_d)$且 $F =1 - a_d$
- **D3D11_BLEND_DEST_COLOR**：$\mathbf{F} = (r_d ,g_d ,b_d )$
- **D3D11_BLEND_INV_DEST_COLOR**：$\mathbf{F} = (1 - r_d ,1 - g_d ,1 - b_d )$
- **D3D11_BLEND_SRC_ALPHA_SAT**：$\mathbf{F} = (a_s', a_s', a_s')$且 $F = a_s'$，其中，$a_s' = \text{clamp}(a_s , 0, 1)$
- **D3D11_BLEND_BLEND_FACTOR**：$\mathbf{F} = (r,g,b)$ 且 $F = a$，其中，颜色 $(r,g,b,a)$ 由

**ID3D11DeviceContext::OMSetBlendState** 方法的第 2 个参数指定（9.4 节）。也就是说，我们可以将一个颜色指定为混合系数；在修改混合状态前，该颜色一直有效。

- **D3D11_BLEND_INV_BLEND_FACTOR**：**F** = (1-*r*,1-*g*,1-*b*)且 *F* = 1-*a*，其中，颜色(*r*,*g*,*b*,*a*)由 **ID3D11DeviceContext::OMSetBlendState** 方法的第 2 个参数指定（9.4 节）。也就是说，我们可以将一个颜色指定为混合系数；在修改混合状态前，该颜色一直有效。

  注意：**clamp** 函数的定义为：

$$clamp(x,a,b)=\begin{cases} x, a \le x \le b \\ a, x < a \\ b, x > b \end{cases}$$

所有的这些混合系数都可用于 RGB 混合方程。但是，以“**_COLOR**”结尾的混合系数不可用于 alpha 混合方程。

# 9.4 混合状态

我们已经讨论了混合运算符和混合系数，但是还没有说该如何在 Direct3D 中使用些值。这些混合参数要通过 **ID3D11BlendState** 接口来控制。我们可以通过填充一个 **D3D11_BLEND_DESC** 结构体并调用 **ID3D11Device::CreateBlendState** 方法来创建该接口：

```
HRESULT ID3D11Device::CreateBlendState(
    const     D3D11_BLEND_DESC     *pBlendStateDesc,ID3D10BlendState
**ppBlendState);
```

**1．pBlendStateDesc**：指向 D3D11_BLEND_DESC 结构体的指针，该结构体用于描述所要创建的混合状态。

**2．ppBlendState**：返回创建后的混合状态接口。

**D3D11_BLEND_DESC** 结构体的定义如下：

```
typedef struct D3D11_BLEND_DESC {
    BOOL AlphaToCoverageEnable; // 默认值：False
    IndependentBlendEnable      // 默认值：False
    D3D11_RENDER_TARGET_BLEND_DESC RenderTarget[8];
} D3D11_BLEND_DESC;
```

**1．AlphaToCoverageEnable**：当设为 true 时，表示启用 alpha-to-coverage 功能。它是一种多重采样技术，在渲染植物的叶子或铁丝网纹理时非常有用。当设为 false 时，表示禁用 alpha-to-coverage 功能。使用 alpha-to-coverage 需要开启多重采样（即，后台和深度缓冲创建时需要开启多重采样）。在第 11 章有一个使用 alpha-to-coverage 的示例。

**2．IndependentBlendEnable**：Direct3D 11 支持同时绘制到 8 个渲染目标。当这个标志设置为 true 时，表示可以在不同的渲染目标上进行不同的混合处理（不同的混合因子、混合操作，混合开启/关闭等）。如果设置为 false，则表示所有渲染目标都使用 **D3D11_BLEND_DESC::RenderTarget** 数组中第一个元素的混合状态，多重渲染目标用于高级的算法，目前为止，我们一次只使用一个渲染目标。

3．**RenderTarget**：包含 8 个 **D3D11_RENDER_TARGET_BLEND_DESC** 元素的数组，第 *i* 个元素描述了第 *i* 个多重渲染目标的混合方式。如果 **IndependentBlendEnable** 设置为 false，则所有的渲染目标都使用 **RenderTarget[0]**进行混合。

**D3D11_RENDER_TARGET_BLEND_DESC** 结构体的定义如下：

```
typedef struct D3D11_RENDER_TARGET_BLEND_DESC{
    BOOL BlendEnable;       // 默认值:False
    D3D11_BLEND SrcBlend; // 默认值:D3D11_BLEND_ONE
    D3D11_BLEND DestBlend; // 默认值 Default:D3D11_BLEND_ZERO
    D3D11_BLEND_OP BlendOp; // 默认值:D3D11_BLEND_OP_ADD
    D3D11_BLEND SrcBlendAlpha; // 默认值:D3D11_BLEND_ONE
    D3D11_BLEND DestBlendAlpha; // 默认值:D3D11_BLEND_ZERO
    D3D11_BLEND_OP BlendOpAlpha; // 默认值:D3D11_BLEND_OP_ADD
    UINT8        RenderTargetWriteMask;        //        默        认
值:D3D11_COLOR_WRITE_ENABLE_ALL
}D3D11_RENDER_TARGET_BLEND_DESC;
```

**1．BlendEnable**：当设为 true 时，表示启用混合功能；当设为 false 时，表示禁用混合功能。

**2．SrcBlend**：**D3D11_BLEND** 枚举类型成员，用于为 RGB 混合指定源混合系数 $\mathbf{F}_{src}$。

**3. DestBlend**: **D3D11_BLEND** 枚举类型成员，用于为 RGB 混合指定目标混合系数 $\mathbf{F}_{dst}$。

**4．BlendOp**：**D3D11_BLEND_OP** 枚举类型成员，用于指定 RGB 混合运算符。

**5．SrcBlendAlpha**：**D3D11_BLEND** 枚举类型成员，用于为 alpha 混合指定源混合系数 $F_{src}$。

**6．DestBlendAlpha**：**D3D11_BLEND** 枚举类型成员，用于为 alpha 混合指定目标混合系数 $F_{dst}$。

**7．BlendOpAlpha**：**D3D11_BLEND_OP** 枚举类型成员，用于指定 alpha 混合运算符。

**8．RenderTargetWriteMask**：一个或多个下列标志值的组合：

```
typedef enum D3D11_COLOR_WRITE_ENABLE
{
    D3D11_COLOR_WRITE_ENABLE_RED = 1,
    D3D11_COLOR_WRITE_ENABLE_GREEN = 2,
    D3D11_COLOR_WRITE_ENABLE_BLUE = 4,
    D3D11_COLOR_WRITE_ENABLE_ALPHA = 8,
    D3D11_COLOR_WRITE_ENABLE_ALL =
     (D3D11_COLOR_WRITE_ENABLE_RED|D3D11_COLOR_WRITE_ENABLE_GREEN|
    D3D11_COLOR_WRITE_ENABLE_BLUE |
    D3D11_COLOR_WRITE_ENABLE_ALPHA)
} D3D11_COLOR_WRITE_ENABLE;
```

这些标志值用于控制混合之后将哪些颜色分量写入后台缓冲区。例如，通过 **D3D11_COLOR_WRITE_ENABLE_ALPHA** 可以屏蔽 RGB 通道，只将 alpha 值写入后台缓冲区。这一功能在实现某些高级技术时非常有用。当禁用混合时，由像素着色器返回的颜色会被屏蔽掉。

要将混合状态对象绑定到管线的输出合并器阶段，我们可以调用：

```
void ID3D11DeviceContext::OMSetBlendState(
    ID3D11BlendState *pBlendState,
    const FLOAT BlendFactor,
    UINT SampleMask);
```

**1．pBlendState**：混合状态对象的指针。

**2．BlendFactor**：用于描述 RGBA 颜色向量的浮点数组。当混合因子指定为 **D3D11_BLEND_BLEND_FACTOR** 或 **D3D11_BLEND_INV_BLEND_FACTOR** 时，Direct3D 将以该颜色向量作为混合系数。

**3．SampleMask**：多重采样最多可以支持 32 个采样源。这个 32 位整数用于启用和禁用采样源。例如，当第 5 个二进制位设为 0 时，表示屏蔽第 5 个采样源。当然，如果实际使用的多重采样源数量少于 5 个，那么屏蔽第 5 个采样源是没有什么实际意义的。当应用程序只使用一个采样源时，该参数只有第 1 个二进制位有效（参见练习 1）。通常，该参数以 0xffffffff 作为默认值，表示不屏蔽任何采样源。

与其他状态块相同，这里有一个默认的混合状态（禁用混合）；如果以空值来调用 **OMSetBlendState** 方法，它就会将混合状态恢复为默认值。注意，混合会在每个像素上执行额外的计算工作，所以我们只有在用到混合时才启用它，用完之后应该立即关闭。

下面是创建和设置混合状态的一个例子：

```
D3D11_BLEND_DESC blendDesc = {0};
transparentDesc.AlphaToCoverageEnable = false;
transparentDesc.IndependentBlendEnable = false;

transparentDesc.RenderTarget[0].BlendEnable = true;
transparentDesc.RenderTarget[0].SrcBlend = D3D10_BLEND_SRC_ALPHA;
transparentDesc.RenderTarget[0] = D3D11_BLEND_INV_SRC_ALPHA;
transparentDesc.RenderTarget[0] = D3D11_BLEND_OP_ADD;
transparentDesc.RenderTarget[0] = D3D11_BLEND_ONE;
transparentDesc.RenderTarget[0].DestBlendAlpha = D3D11_BLEND_ZERO;
transparentDesc.RenderTarget[0].BlendOpAlpha = D3D11_BLEND_OP_ADD;
transparentDesc.RenderTarget[0].RenderTargetWriteMask[0]          =
D3D10_COLOR_WRITE_ENABLE_ALL;

ID3D11BlendState* TransparentBS;
HR(device->CreateBlendState(&transparentDesc, &TransparentBS));
...
float blendFactor[] = {0.0f, 0.0f, 0.0f, 0.0f};
md3dImmediateContect->OMSetBlendState(TransparentBS,  blendFactor,
0xffffffff);
```

与其他状态块接口一样，我们应该在应用程序初始化时创建它们，然后根据需要在些状态接口之间进行切换。

混合状态对象也可以在 effect 文件中创建和设定：

```
BlendState blend
{
    // 第一个渲染目标的混合状态
    BlendEnable[0] = TRUE;
    SrcBlend[0] = SRC_COLOR;
    DestBlend[0] = INV_SRC_ALPHA;
    BlendOp[0] = ADD;
    SrcBlendAlpha[0] = ZERO;
    DestBlendAlpha[0] = ZERO;
```

```
    BlendOpAlpha[0] = ADD;
    RenderTargetWriteMask[0] = 0x0F;

    // 第二个渲染目标的混合状态
    BlendEnable[1] = True;
    SrcBlend[1] = One;
    DestBlend [1] = Zero;
    BlendOp[1] = Add;
    SrcBlendAlpha[1] = Zero;
    DestBlendAlpha[1] = Zero;
    BlendOpAlpha[1] = Add;
    RenderTargetWriteMask[1] = 0x0F;
};

technique11 Tech
{
    pass P0
    {
        ...
        // 在这个 pass 中使用“混合”。

SetBlendState(blend,float4(0.0f,0.0f,0.0f,0.0f),0xffffffff);
    }
}
```

在混合状态对象中指定的这些值与在 C++结构体中指定的值基本相同，只是省去了一些前缀。例如，在 effect 文件中我们指定 **SRC_COLOR**，而不是 **D3D11_BLEND_SRC_COLOR**。

# 9.5 例子

在下面的几个小节中，我们将介绍一些用于实现特殊效果的混合系数组合值。在这些例子中，我们主要关注于 RGB 混合，alpha 混合的处理是类似的。

## 9.5.1 屏蔽颜色写入

假设我们希望原始目标像素保持不变，即不被任何其他数值覆盖，也不与当前的光栅化源像素进行混合。当我们希望屏蔽后台缓冲区、只向深度/模板缓冲区写入数据时，该算法非常有用。要实现这一算法，可将源像素混合系数设为 **D3D11_BLEND_ZERO**，目标混合系数设为 **D3D11_BLEND_ONE**，混合运算符设为 **D3D11_BLEND_OP_ADD**。使用一方案，混合方程可简化为：

$$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} \boxplus \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$$

$$\mathbf{C} = \mathbf{C}_{src} \otimes (0, 0,0) + \mathbf{C}_{dst} \otimes (1, 1,1)$$

$$\mathbf{C} = \mathbf{C}_{dst}$$

还有一种方法可以实现相同的结果，就是将 **D3D11_RENDER_TARGET_BLEND_DESC::RenderTargetWriteMask** 设置为 0，这样就不会写入任何颜色通道。

## 9.5.2 加法和减法

假设我们希望将源像素和目标像素相加（参见图 9.2）。要实现这一算法，可将源混合系数设为 **D3D11_BLEND_ONE**，目标混合系数设为 **D3D11_BLEND_ONE**，混合运算符设为 **D3D11_BLEND_OP_ADD**。使用一方案，混合方程可简化为：

$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} \boxplus \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$

$\mathbf{C} = \mathbf{C}_{src} \otimes (1, 1,1) + \mathbf{C}_{dst} \otimes (1, 1,1)$

$\mathbf{C} = \mathbf{C}_{src} + \mathbf{C}_{dst}$

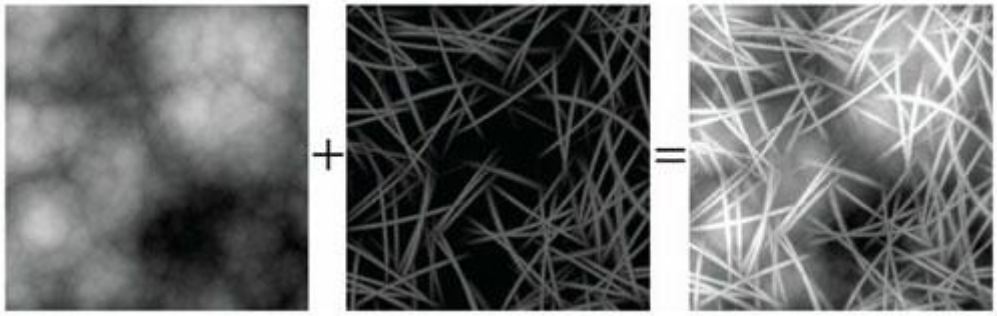


**图 9.2 将源颜色和目标颜色相加。由于颜色叠加，所以创建后的图像比原始图像的亮度要高一些。**

当把 **D3D11_BLEND_OP_ADD** 替换为 **D3D11_BLEND_OP_SUBTRACT** 或 **D3D11_BLEND_OP_REV_SUBTRACT** 时，可以实现源像素和目标像素的减法运算（参见图 9.3）。

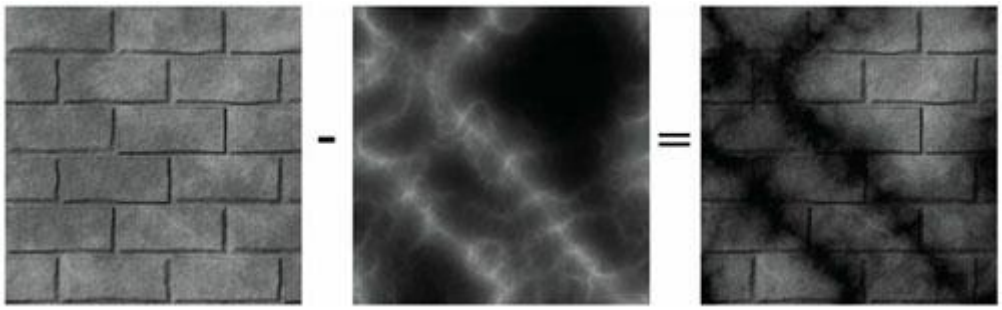


**图 9.3 从目标颜色中减去源颜色。由于颜色削减，所以创建后的图像比原始图像的亮度要低一些。**

## 9.5.3 乘法

假设我们要将源像素和目标像素相乘（参见图 9.4）。要实现这一算法，可将源混合系数设为 **D3D11_BLEND_ZERO**，目标混合系数设为 **D3D11_BLEND_SRC_COLOR**，混合运算符设为 **D3D11_BLEND_OP_ADD**。使用一方案，混合方程可简化为：

$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} \boxplus \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$

$\mathbf{C} = \mathbf{C}_{src} \otimes (0, 0,0) + \mathbf{C}_{dst} \otimes \mathbf{C}_{src}$

$\mathbf{C} = \mathbf{C}_{dst} \otimes \mathbf{C}_{src}$

**图 9.4 将源颜色和目标颜色相乘。**

## 9.5.4 透明度

我们可以使用源 alpha 分量 $a_s$ 控制源像素的不透明度（例如，当 alpha 为 0.0 时表示 0%不透明，为 0.4 时表示 40%不透明，为 1.0 时表示 100%不透明）。不透明和透明之间的关系可以简单地表示为 $T=1-A$，其中 $A$ 表示不透明度，$T$ 表示透明度。例如，当某物的不透明度为 0.4 时，它的透明度为 $1-0.4=0.6$。现在，我们希望根据源像素的不透明度来混合源像素和目标像素。要实现一算法，可将源混合系数设为 **D3D11_BLEND_SRC_ALPHA**，目标混合系数设为 D3D11_BLEND_INV_SRC_ALPHA，混合运算符设为 **D3D11_BLEND_OP_ADD**。使用这一方案，混合方程可简化为：

$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} \boxplus \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$

$\mathbf{C} = \mathbf{C}_{src} \otimes (a_s, a_s, a_s) + \mathbf{C}_{dst} \otimes (1 - a_s, 1 - a_s, 1 - a_s)$

$\mathbf{C} = a_s\mathbf{C}_{src} + (1 - a_s)\mathbf{C}_{dst}$

例如，当 $a_s = 0.25$ 时，表示源像素的不透明度为 25%， 或者说源像素的透明度为 75%。那么，当源像素和目标像素混合之后，我们最终得到的颜色应该是由 25%的源像素和 75%的目标像素组成（源像素将“挡住”一部分目标像素）。综上所述，此时的混合方程为：

$\mathbf{C} = a_s\mathbf{C}_{src} + (1 - a_s)\mathbf{C}_{dst}$

$\mathbf{C} = 0.25\mathbf{C}_{dst} + 0.75\mathbf{C}_{src}$

通过一混合方式，我们可以绘制如图 9.1 所示的透明物体。但需要注意的是，此时物体的绘制顺序非常重要。我们需要遵守如下规则：

*首先绘制非透明物体。然后，根据透明物体与摄像机之间的距离进行排序，按照从后向前的顺序绘制透明物体。*

之所以要按照从后向前的顺序进行绘制，是为了让前面的物体和后面的物体进行混合。如果一个物体是透明的，那么我们就会透过这个物体看到它后面的其他物体。所以，必须将透明物体后面的所有物体先绘制出来，然后才能将透明的源像素和后台缓冲区中的目标像素进行混合。

对于 9.5.1 节的混合方程来说，绘制顺序并不重要，因为它只是简单地阻止源像素向后台缓冲区的写入操作。对于 9.5.2 和 9.5.3 节的混合方程，我们必须先绘制非透明物体，再绘制透明物体；因为我们需要在混合之前将所有的非透明几何体存入后台缓冲区。不过，我们不需要对透明物体进行排序，因为这些运算满足交换律。也就是，从后台缓冲区中的某个像素颜色 **B** 开始执行 $n$ 次加/减/乘法运算，最终得到的混合颜色是一样的，与顺序无关：

$\mathbf{B}' = \mathbf{B} + \mathbf{C}_0 + \mathbf{C}_1 + \cdots + \mathbf{C}_{n-1}$

$\mathbf{B}' = \mathbf{B} - \mathbf{C}_0 - \mathbf{C}_1 - \cdots - \mathbf{C}_{n-1}$

$\mathbf{B}' = \mathbf{B} \otimes \mathbf{C}_0 \otimes \mathbf{C}_1 \otimes \cdots \otimes \mathbf{C}_{n-1}$

### 9.5.5 混合与深度缓冲

当使用加/减/乘法混合时会出现一个与深度测试相关的问题。这里，我们仅以加法混合为例进行说明，同样的情况也适用于减法和乘法混合。当我们使用加法混合来渲染一个粒子系统 *S* 时，每个粒子是否相互遮挡并不重要；我们只需要把粒子的颜色简单地累加起来（参见图 9.5）。我们并不希望对 *S* 中的每个粒子进行深度测试。在这一情景中，如果不按照从后向前的顺序绘图，那么当 *S* 中的某个粒子被另一个粒子遮挡时，该粒子将无法通过深度测试，它的像素片段将被丢弃，也就是说该粒子的像素颜色不会累加到最终的混合颜色中。我们应该在渲染 *S* 时禁用深度写入功能，使粒子的深度信息不写入深度缓冲区。在禁用深度写入功能之后，由加法混合生成的粒子深度信息不会写入到深度缓冲区；所以，当 *S* 中的一个粒子被其他粒子遮挡时，该粒子依然可以通过深度测试并绘制到后台缓冲区中。注意，我们只在绘制 *S* 时禁用深度写入功能（以便于使用加法混合绘制该粒子系统），而深度读取和深度测试功能依然有效。非透明物体（在绘制透明物体之前）仍然可以遮挡位于它后面的透明物体。例如，当你在一堵墙的后面绘制一个透明物体时，该物体不会显示出来，因为它被墙体挡住了。我们会在下一章讲解有关深度写入和深度测试的设置方法。

**图 9.5 当使用加法混合时，火焰中心的亮度比火焰边缘的亮度高，因为中间的粒子较多，它们彼此重叠并累加在一起。随着粒子逐渐散开，亮度会逐渐衰减，因为当重叠在一起的粒子越来越少时，累加值会相应降低。**

# 9.6 Alpha 通道

9.5.4 节的例子说明，在 RGB 混合中，源 alpha 分量可以用来控制透明度。混合方程中

的源颜色来自于像素着色器。我们会在最后一章中看到，我们将漫反射材质的 alpha 值作为像素着色器的 alpha 输出。也就是说，漫反射贴图的 alpha 通道可以用来控制透明度。

```
float4 PS(VertexOUT pin) : SV_Target
{
    …
    // 从漫反射材质和纹理中提取 alpha
    litColor. a  =  gMaterial.Diffuse.a * texColor.a;

    return litColor;
}
```

你可以使用任何一款流行的图像处理软件为纹理添加 alpha 通道，比如 Adobe Photoshop，然后使用一种支持 alpha 通道的格式（例如，32 位 BMP 格式或 DDS 格式）保存图像文件。不过，这里我们会展示另外一种方法，使用我们在前一章介绍的 DXTex 实用工具插入 alpha 通道。

假设我们有两幅图像：一幅是 RGB 彩色图像，另一幅是灰阶图像。这幅灰阶图像将作为 alpha 通道插入到彩色图像中（参见图 9.6）。

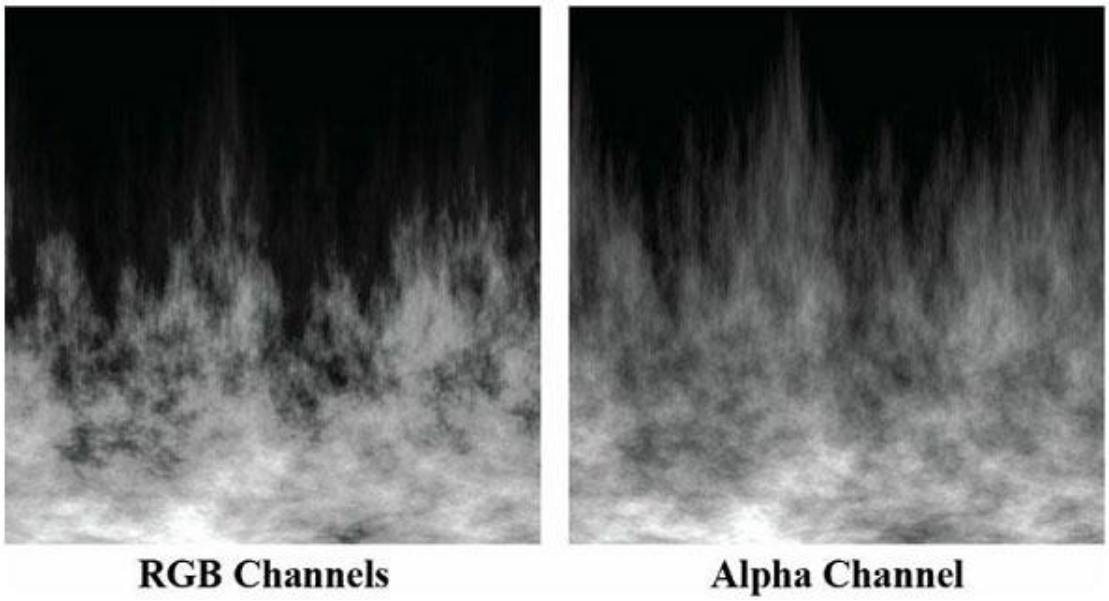



**图 9.6 RGB 图像（左）和灰阶图像（右）。灰阶图像将插入到纹理的 alpha 通道中。**

现在，运行 DXTex 工具程序，打开本章演示程序文件夹下的 *fire_rgb.bmp* 文件。火焰纹理会被自动载入为一个 24 位 RGB 纹理（即，**D3DFMT_R8G8B8**），其中红色、绿色和蓝色分量各占 8 位。我们必须为纹理指定一种支持 alpha 通道的格式，比如 32 位 ARGB 纹理格式 **D3DFMT_A8R8G8B8** 或支持 alpha 通道的 **D3DFMT_DXT5** 压缩格式。在菜单栏中选择 **Format>Change Surface Format** 命令，弹出如图 9.7 所示的对话框。选择 **DXT5** 格式，单击 **OK**。

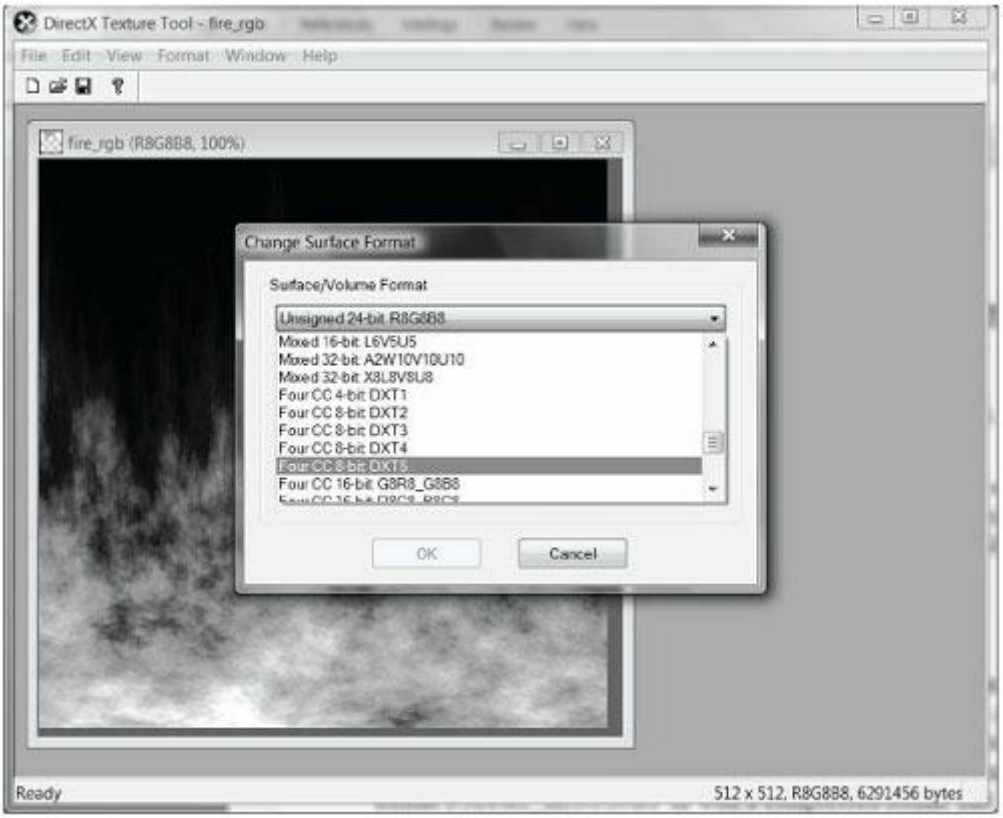



**图 9.7 修改纹理格式。**

这样就创建了一幅带有 alpha 通道的压缩纹理。我们下一步要做的是向 alpha 通道导入数据。我们要把图 9.6 所示的 8 位灰阶贴图导入到 alpha 通道中。在菜单栏中选择 **File>Open Onto Alpha Channel Of This Texture** 命令，然后再选择 **Format>Generate Mip Maps** 命令。此时会弹出一个对话框，要求我们指定包含 alpha 数据的图像文件的位置。选择本章演示程序文件夹下的 *fire_a.bmp* 文件。图 9.8 展示了插入 alpha 通道后的程序界面——可以看到纹理已经与背景颜色混合在一起了。我们可以通过选择 **View> Change Background Color…**菜单命令改变背景颜色。或者选择 **View>Alpha Channel Only** 命令，让窗口只显示 alpha 通道。

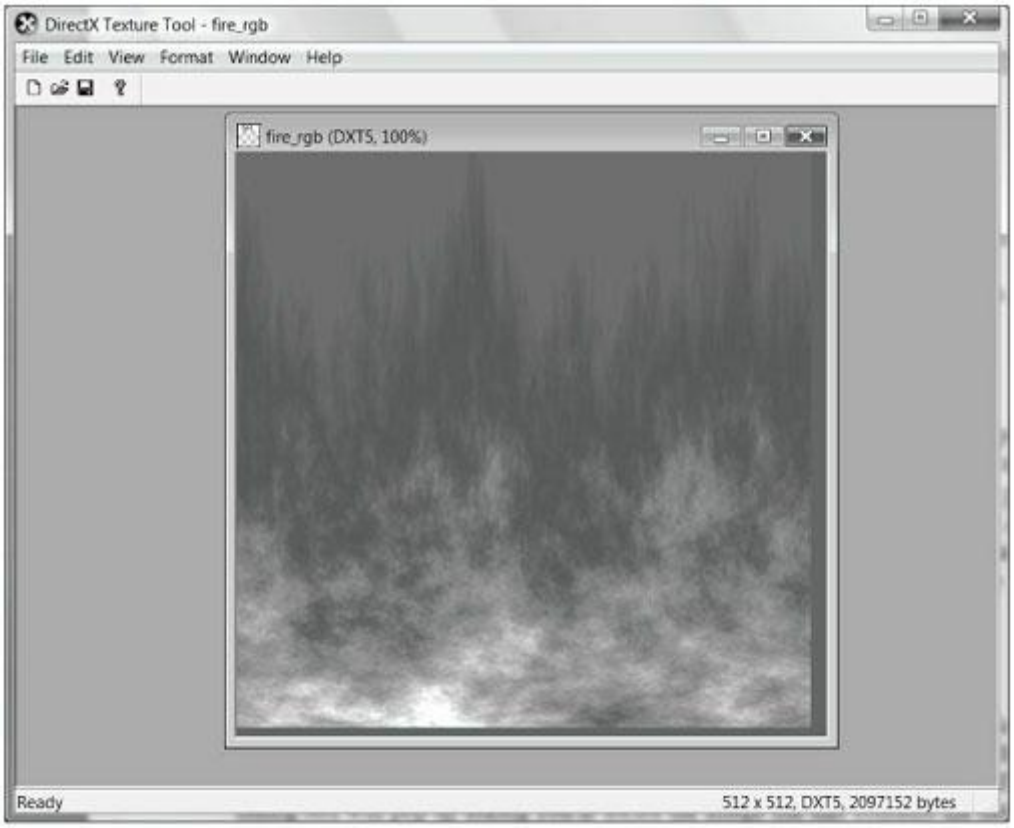



**图 9.8 带有一个 alpha 通道的纹理。火焰纹理已经与蓝色背景混合在一起了。**

现在，指定一个文件名（比如，fire.dds），保存纹理。

# 9.7 裁剪像素

有时，我们希望完全丢弃某个源像素，使它不再接受后续处理。这一工作可以由 HLSL 的内置函数 **clip(x)**来实现。该函数只能在像素着色器中使用，当 x<0 时丢弃当前像素，使之不再接受后续处理。该函数在渲染如图 9.9 所示的铁丝网纹理时非常有用。也就是说，它非常适合于渲染那些完全不透明或者完全透明的像素。

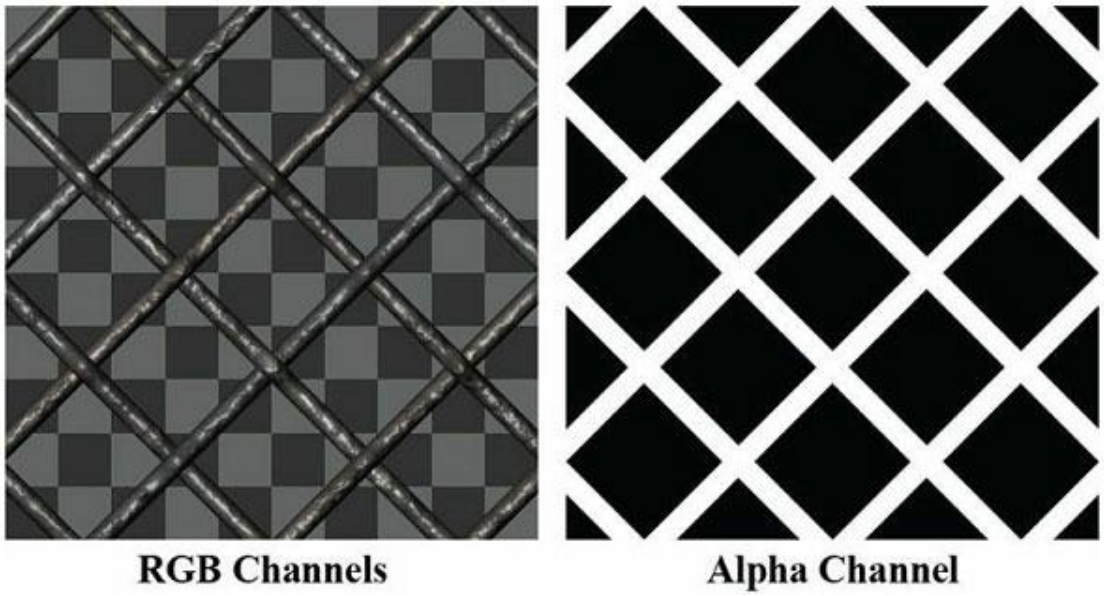



**图 9.9 带有 alpha 通道的铁丝网纹理。clip 函数将丢弃那些带有黑色 alpha 值的像素，不对这些像素进行绘制；只有铁丝网部分会保留下来。从本质上讲，alpha 通道剔除了纹理中的“非铁丝网”像素。**

在像素着色器中，我们攫取了漫反射纹理的 alpha 分量。当它的值接近于 0 时，我们将该像素视为完全透明，丢弃该像素，不再对它进行后续处理。

```
float4 PS(VertexOut pin, uniform int gLightCount, uniform bool
gUseTexure, uniform bool gAlphaClip, uniform bool gFogEnabled) :
SV_Target
{
    // 插值后的法线需要重新规范化
    pin.NormalW = normalize (pin.NormalW);
    // toEye 矢量用于光照计算
    float3 toEye = gEyePosW - pin.PosW;
    // 保存表面顶点离开相机的距离信息
    float distToEye = length(toEye);
    // 规范化
    toEye /= distToEye;
    // 初始化纹理颜色
    float4 texColor = float4(1, 1, 1, 1);
    if (gUseTexure)
    {
        // 采样纹理
        texColor = gDiffuseMap.Sample(samAnisotropic, pin.Tex);

        if (gAlphaClip)
        {
            // 如果纹理的 alpha<0.1，则丢弃像素。
            // 注意，我们应该尽可能早地进行这个测试，这样我们就可以及早退出
            // shader，忽略其余 shader 代码。
```

```
            clip(texColor.a - 0.1f);
        }
    }
…
```

因为我们可能只在某些几何体上进行裁剪操作，所以只有在参数 **gAlphaClip** 设置为 true 的情况下我们才进行裁剪，这样我们就可以根据特定的 shader 切换裁剪。

注意，使用混合也可以得到同样的效果，只是这种（裁剪）方式的运行效率更高一些。这种方式即不需要进行任何混合计算，也不需要考虑物体的绘制顺序。而且，它可以在像素着色器中尽早丢弃像素，避免执行不必要的像素着色器指令（被丢弃的像素不会参与任何计算）。

**注意**：由于过滤器的作用，alpha 通道可能会变得有些模糊，所以当裁剪像素时，你应该保留一些容差值。例如，裁剪 alpha 值接近于 0 的像素，而不必让 alpha 值精确为 0。

图 9.10 是“Blend”演示程序的屏幕截图。它使用透明混合绘制了半透明的水体，使用了新的铁丝网纹理，并且在像素着色器中加入了裁剪测试功能。另一个值得注意的地方是，由于我们现在要透过立方体看到背面的铁丝网纹理，所以我们希望禁用背面消隐功能：

```
D3D11_RASTERIZER_DESC noCullDesc;
ZeroMemory(&noCullDesc, sizeof(D3D11_RASTERIZER_DESC));
noCullDesc.FillMode  = D3D11_FILL_SOLID;
noCullDesc.CullMode = D3D11_CULL_NONE;
noCullDesc.FrontCounterClockwise = false;
noCullDesc.DepthClipEnable = true;
ID3D11RasterizerState * NoCullRS;
HR(device->CreateRasterizerState(&noCullDesc, &NoCullRS));
…
// 因为铁丝网纹理包含透明区域，我们可以透过立方体看到背面的三角形，所以我们希望禁用背面消隐功能
md3dImmediateContext->RSSetState(NoCullRS);
boxTech->GetPassByInde x(p)->Apply(0, md3dImmediateContext);
md3dImmediateContext->DrawIndexed(36, 0, 0);
// 恢复为默认的渲染状态
md3dImmediateContext->RSSetState(0);
```

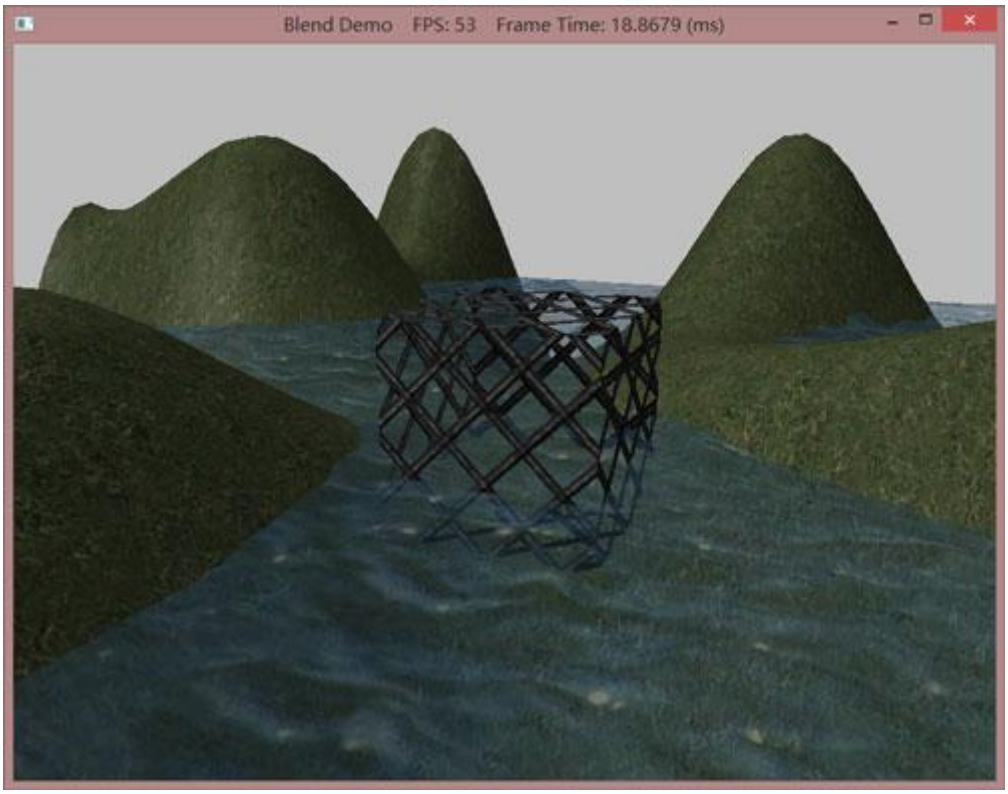



**图 9.10“Blend”演示程序的屏幕截图。**

# 9.8 雾

当我们在游戏中模拟某些类型的天气状况时，可能会用到雾效（参见图 9.11）。雾除了本身所具有的用途外，还具有一些附加效用。例如，雾可以用来掩盖渲染过程中出现的不自然的人工痕迹，避免蹿出问题的发生。蹿出（popping）是指由于摄像机的移动，使原本在远平面后面的物体突然进入平截头体内，从不可见变为可见；这看上去就像是突然“蹿”到场景里面一样。通过在一定距离内加入雾效，可以掩盖这一问题。注意，即使你的场景是在晴朗的白天，你也可以在较远的地方加入一些淡淡的雾气。因为就算是晴天，远处的物体（比如山岳）也会看上去有些模糊，就像是有一层薄薄的雾笼罩在上面一样，当深度增加时，物体的对比度会逐渐减小。我们可以用雾来模拟这种大气透视现象。

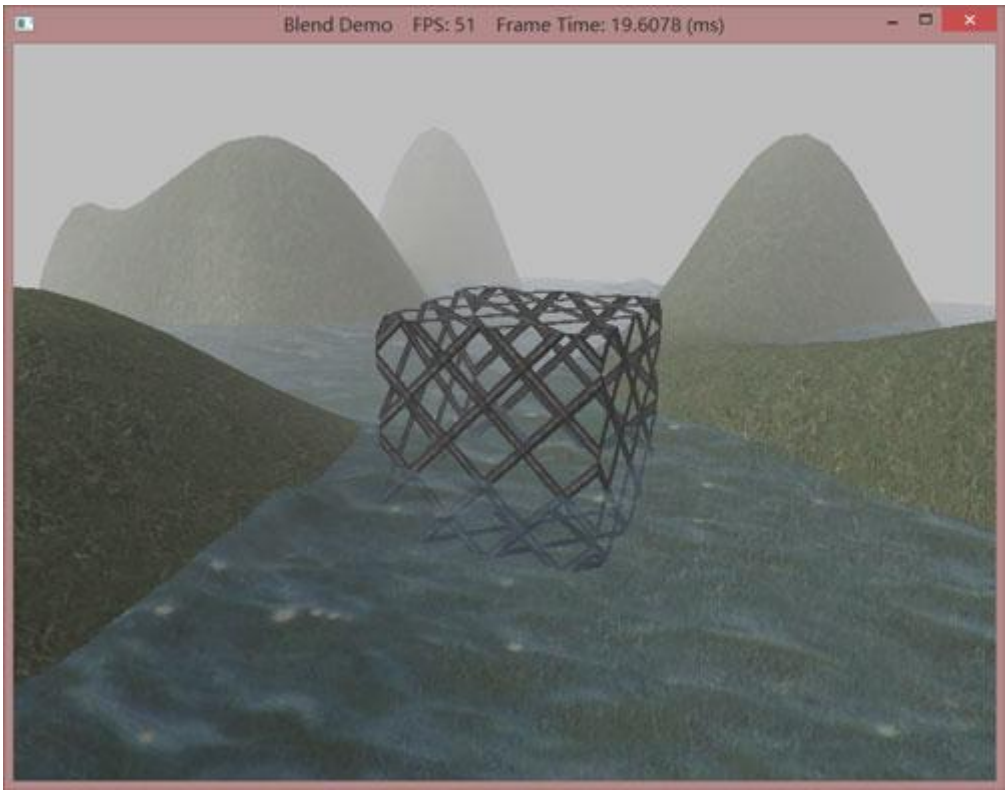



**图 9.11 雾演示程序的屏幕截图。**

我们用如下方法来实现雾效：我们为雾指定一个颜色、一个相对于摄像机的起始位置和一个范围（即，该范围从雾的起始位置开始到完全遮隐任何物体为止）。那么，三角形表面点的颜色等于点的照颜色与雾颜色的加权平均值：

$$foggedColor = litColor + s\,(fogColor - litColor) = (1 - s) \cdot litColor + s \cdot fogColor$$

参数 $s$ 的取值范围是从 0 到 1，它是一个以表面点和观察点之间的距离为自变量的函数。随着表面点和观察点之间的距离增大，雾在表面点颜色中所占的比例会越来越大。该参数的定义如下：

$$s = saturate\left(\frac{dist(\mathbf{p},\mathbf{E}) - fogStart}{fogRange}\right)$$

其中，dist(**p**,**E**)是表面点 **p** 和观察点 **E** 之间的距离。**saturate** 函数将自变量限定在[0,1]区间内：

$$saturate(x) = \begin{cases} x, 0 \le x \le 1 \\ 0, x < 0 \\ 1, x > 1 \end{cases}$$

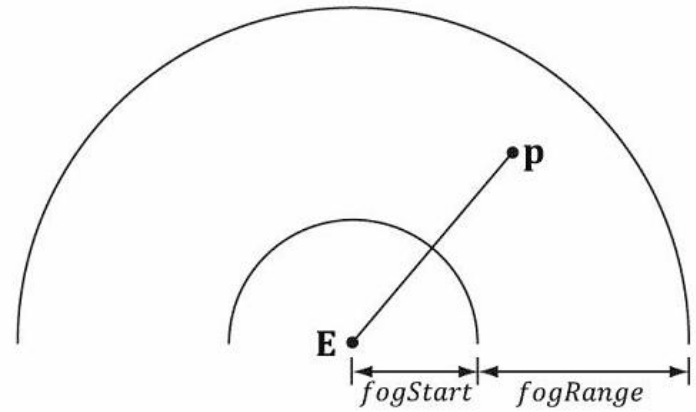



**图 9.12 从观察点到表面点的距离，以及 *fogStart* 和 *fogRange* 参数。**

图 9.13 是距离函数 $s$ 的曲线图。我们可以看到，当 dist(**p**,**E**)≤*fogStart* 时，$s = 0$ 且雾化颜色为：

$$foggedColor = litColor$$

换句话说，当顶点与观察点之间的距离小于 *fogStart* 时，雾不会影响顶点颜色。从 *fogStart* 这个名字就可以猜到：只有当顶点与观察点之间的距离超过了 *fogStart* 这个分界线时，雾才会开始影响顶点颜色。

设 $fogEnd = fogStart + fogRange$，当 dist(**p**,**E**)≥*fogEnd* 时，$s = 1$ 且雾化颜色为：

$$foggedColor = fogColor$$

换句话说，当表面点与观察点之间的距离大于等于 *fogEnd* 时，雾将完全取代表面点本身的光照颜色。

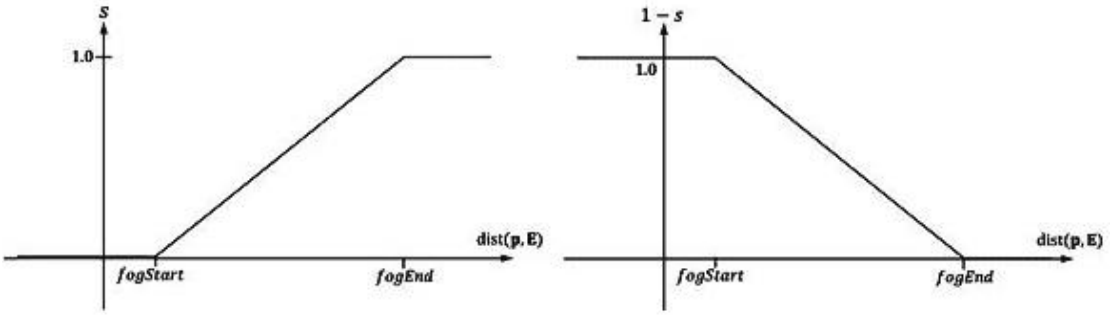



**图 9.13 （上图）距离函数 *s*（雾色权重）的曲线图。（下图）距离函数 1 − *s*（光照颜色权重）的曲线图。当 *s* 增大时，1 − *s* 会减小相同的量。**

我们可以看到，当 *fogStart* < dist(**p**,**E**) < *fogEnd* 时，随着 dist(**p**,**E**)从 *fogStart* 增加到 *fogEnd*，$s$ 会线性地从 0 增加到 1。这说明当距离增加时，雾色所占的比重会越来越大，而表面点的光照颜色所占的比重会越来越小。这很容易理解，因为距离越远，被雾色笼罩的表面点就越多。

下面的着色器代码示范了雾的实现方法。我们在顶点级别上计算距离和插值参数，然后进行插值，在像素级别上完成照颜色的计算。

```
cbuffer cbPerFrame
{
    DirectionalLight gDirLights[3];
    float3 gEyePosW;

    float  gFogStart;
    float  gFogRange;
```

```
    float4 gFogColor;
};

cbuffer cbPerObject
{
    float4x4 gWorld;
    float4x4 gWorldInvTranspose;
    float4x4 gWorldViewProj;
    float4x4 gTexTransform;
    Material gMaterial;
};

// Nonnumeric values cannot be added to a cbuffer.
Texture2D gDiffuseMap;

SamplerState samAnisotropic
{
    Filter = ANISOTROPIC;
    MaxAnisotropy = 4;

    AddressU = WRAP;
    AddressV = WRAP;
};

struct VertexIn
{
    float3 PosL    : POSITION;
    float3 NormalL : NORMAL;
    float2 Tex     : TEXCOORD;
};

struct VertexOut
{
    float4 PosH    : SV_POSITION;
    float3 PosW    : POSITION;
    float3 NormalW : NORMAL;
    float2 Tex     : TEXCOORD;
};

VertexOut VS(VertexIn vin)
{
    VertexOut vout;

    // 转换到世界空间
```

```
    vout.PosW    = mul(float4(vin.PosL, 1.0f), gWorld).xyz;
    vout.NormalW = mul(vin.NormalL, (float3x3)gWorldInvTranspose);

    // 转换到齐次剪裁空间
    vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj);

    // Output vertex attributes for interpolation across triangle.
    vout.Tex = mul(float4(vin.Tex, 0.0f, 1.0f), gTexTransform).xy;

    return vout;
}

float4 PS(VertexOut pin, uniform int gLightCount, uniform bool
gUseTexure, uniform bool gAlphaClip, uniform bool gFogEnabled) :
SV_Target
{
    // 插值后的法线需要重新归一化
    pin.NormalW = normalize(pin.NormalW);

    // toEye 矢量用于光照计算
    float3 toEye = gEyePosW - pin.PosW;

    // 保存观察点到表面的距离
    float distToEye = length(toEye);

    // 规范化
    toEye /= distToEye;

    // Default to multiplicative identity.
    float4 texColor = float4(1, 1, 1, 1);
    if(gUseTexure)
    {
        // 采样纹理
        texColor = gDiffuseMap.Sample( samAnisotropic, pin.Tex );

        if(gAlphaClip)
        {
            // 如果纹理的 alpha<0.1，则丢弃像素。
            // 注意，我们应该尽可能早地进行这个测试，这样我们就可以及早退出
            // shader，忽略其余 shader 代码。
            clip(texColor.a - 0.1f);
        }
    }
```

```
    //
    // 光照
    //

    float4 litColor = texColor;
    if( gLightCount > 0  )
    {
        // Start with a sum of zero.
        float4 ambient = float4(0.0f, 0.0f, 0.0f, 0.0f);
        float4 diffuse = float4(0.0f, 0.0f, 0.0f, 0.0f);
        float4 spec    = float4(0.0f, 0.0f, 0.0f, 0.0f);

        // Sum the light contribution from each light source.
        [unroll]
        for(int i = 0; i < gLightCount; ++i)
        {
            float4 A, D, S;
            ComputeDirectionalLight(gMaterial,        gDirLights[i],
pin.NormalW, toEye,
                A, D, S);

            ambient += A;
            diffuse += D;
            spec    += S;
        }

        // Modulate with late add.
        litColor = texColor*(ambient + diffuse) + spec;
    }

    //
    // 雾化
    //

    if( gFogEnabled )
    {
        float  fogLerp  =  saturate(  (distToEye  -  gFogStart)  /
gFogRange );

        // 混合雾颜色和光照颜色
        litColor = lerp(litColor, gFogColor, fogLerp);
    }

    // 从漫反射材质和纹理中提取 alpha
```

```
    litColor.a = gMaterial.Diffuse.a * texColor.a;

    return litColor;
}
```

**注意**：在雾效计算中，我们使用了 **distToEye**，这个值还用来归一化 **toEye** 矢量，下面的代码也可以用来归一化 toEye 矢量，但不够优化：

```
float3 toEye = normalize(gEyePosW - pin.PosW);
float distToEye = distance(gEyePosW, pin.PosW);
```

上述代码必须计算两次 **toEye** 矢量的长度，一次在 **normalize** 函数中，一次在 **distance** 函数中。

**注意**：“Blend”演示程序支持三种渲染模式，可以通过按‘1’，‘2’，‘3’键进行切换。第一个模式只绘制了光照（见图 9.14）。光照前面已经讨论过了，但看一下没有纹理的场景还是有用处的。第二个模式绘制了光照和纹理（图 9.10）。第三个模式绘制了光照、纹理和雾效（图 9.11）。这些渲染模式的切换是为了说明我们是如何在 effect 框架中通过 uniform 参数生成不同效果。读者也可以看一下 Basic.fx 的汇编代码，也可以比较一下三种模式下的帧频。

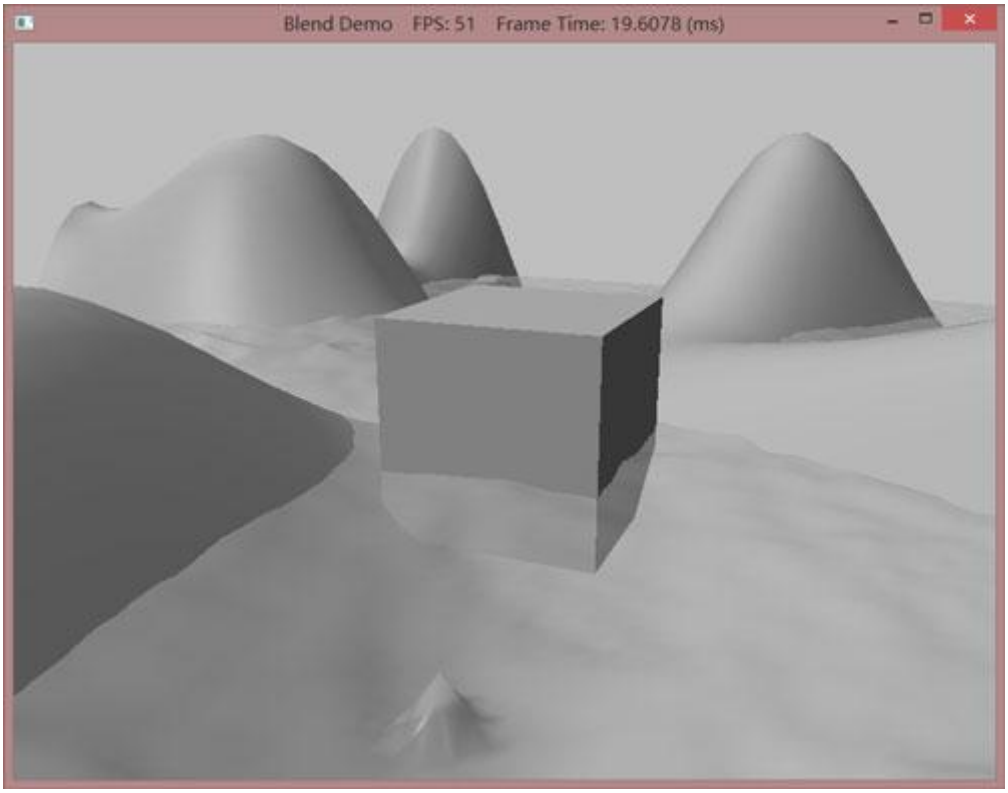



**图 9.14 只开启光照时的“Blend”示例程序截图**

# 9.9 小结

**1.** 混合技术可以将当前的光栅化像素（也称为源像素）与后台缓冲区中的像素（也称为目标像素）混合（融合）在一起。该技术通常用来渲染半透明物体，比如水和玻璃。

**2.** 混合方程为：

$$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} \boxplus \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$$

$$A = A_{src}F_{src} \boxplus A_{dst}F_{dst}$$

注意，RGB 分量与 alpha 分量的混合方程是分开的。二进制运算符⊞可以是 **D3D11_BLEND_OP** 枚举类型定义的任何一个运算符。

**3．**$\mathbf{F}_{src}$、$\mathbf{F}_{dst}$、$F_{src}$和$F_{dst}$称为混合系数，它们提供了一种自定义混合方程的途径。它们可以是 **D3D11_BLEND** 枚举类型定义的任何一个成员。另外，以“**_COLOR**”结尾的混合系数不能用于 alpha 混合方程。

**4．**源 alpha 信息来自于漫反射材质。在我们的框架中，漫反射材质是一个纹理贴图，纹理的 alpha 通道存储了 alpha 信息。

**5．**HLSL 的内置函数 **clip(x)**可以用来完全丢弃源像素，使源像素不再接受后续处理。该函数只能在像素着色器中使用，当 x<0 时丢弃当前像素，不再对它进行后续处理。另外，该函数非常适合于渲染那些完全不透明或者完全透明的像素（它用于丢弃完全透明的像素——这些像素的 alpha 值接近于 0）。

**6．**雾可以用来模拟各种天气效果和大气透视，掩盖渲染过程中出现的不自然的人工痕迹，避免“蹿出”问题的发生。在我们模拟的线性雾中，我们为雾指定了一个颜色、一个相对于摄像机的起始位置和一个范围。三角形表面点的颜色等于照颜色与雾颜色的加权平均值：

$$foggedColor = litColor + s\,(fogColor - litColor) = (1 - s) \cdot litColor + s \cdot fogColor$$

参数 $s$ 的取值范围是从 0 到 1，它是一个以表面点和观察点之间的距离为自变量的函数。随着表面点和观察点之间的距离增大，雾在表面点颜色中所占的比例会越来越大。

# 10.1 深度/模板格式及清空操作

模板缓冲区（stencil buffer）是一种用来实现特殊效果的离屏（off-screen）缓冲区。模板缓冲的大小与后台缓冲及深度缓冲的大小相同，也就是说，模板缓冲的第 *ij* 个像素对应于后台缓冲和深度缓冲第 *ij* 个像素。我们在 4.1.5 节的“注意”中提到，当指定一个模板缓冲时，它总是与深度缓冲共享相同的内存空间。尤如名字所指出的，模板缓冲区的用法就像是模板一样，它可以挡住某些像素片段，不让它们存入后台缓冲。（译者注：比如喷油漆时使用的图案模板，先把模板贴在汽车上或者其他什么地方，然后开始喷油漆。在模板镂空的地方会有油漆喷到汽车上，而没有镂空的地方会挡住油漆。在喷完之后，揭下模板，图案就喷涂在汽车上了。）

例如，当实现一个镜像效果时，我们需要反射镜子对面的物体；不过，我们希望镜像只显示在镜子里面。我们可以使用模板缓冲区来控制镜像范围，阻止镜像绘制到镜子之外的区域（参见图 10.1）。

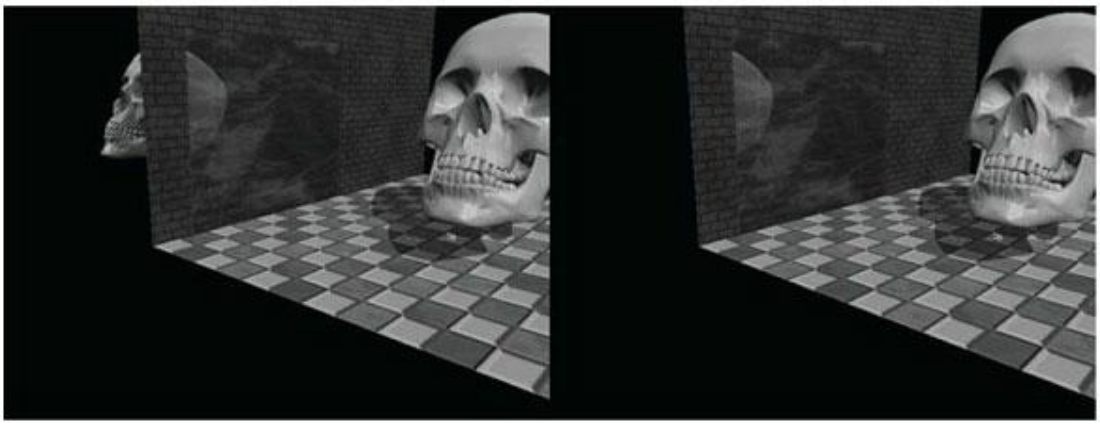


**图 10.1 （左）头骨的镜像在镜中的显示是正确的，由于没有通过深度测试，镜像也正确地被砖墙阻挡。但是，我们能看到墙后的镜像，这样就不对了（镜像应只出现在镜子中）。（右）通过使用模板缓冲，我们可以阻止头骨的镜像绘制到镜子之外的区域。**

我们可以通过 **ID3D11DepthStencilState** 接口控制模板缓冲（和深度缓冲）。与混合一样，该接口也提供了一套灵活而强大的功能集合。要学习如何高效地使用模板缓冲区，最有效的方法是仔细研究现有的示例应用程序。当你弄懂了几个使用模板缓冲区的应用程序之后，就会对它有一个更清晰的认识，知道该如何用它来解决实际工作问题。

**学习目标：**

1．了解如何使用 ID3D11DepthStencilState 接口控制深度缓冲和模板缓冲。

2．学习如何通过模板缓冲来实现镜像效果，阻止镜像绘制到镜子之外的区域。

3．会辨认双重混合以及理解模板缓冲如何防止出现这种情况。

4．解释什么是深度复杂性，介绍两种方法测量深度复杂性。

前面讲过，深度/模板缓冲区是一个纹理，在创建它时必须为它指定某种数据格式。可用于深度/模板缓冲区的格式有：

**1．DXGI_FORMAT_D32_FLOAT_S8X24_UINT**：32 位浮点深度缓冲区。其中 8 位用于模板缓冲区，每个模板元素的取值范围是[0,255]。其余 24 位闲置。

**2．DXGI_FORMAT_D24_UNORM_S8_UINT**：24 位用于深度缓冲区，每个深度元素

的取值范围是[0,1]；8 位用于模板缓冲区，每个模板元素的取值范围是[0,255]。

在 D3DApp 框架中，我们使用如下格式创建深度缓冲区：

```
depthStencilDesc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
```

另外，在每帧开始时，模板缓冲区应该被重置为某些初始值。该操作由如下方法完成（它可以同清空深度/模板缓冲区）：

```
void ID3D11Device::ClearDepthStencilView(
    ID3D11DepthStencilView *pDepthStencilView,
    UINT ClearFlags, FLOAT Depth, UINT8 Stencil);
```

**1．pDepthStencilView**：所要清空的深度/模板缓冲区的视图的指针。

**2．ClearFlags**：当设为 **D3D11_CLEAR_DEPTH** 时，表示只清空深度缓冲区；当设为 **D3D11_CLEAR_STENCIL** 时，表示只清空模板缓冲区；当设为 **D3D11_CLEAR_DEPTH|D3D11_CLEAR_STENCIL** 时，表示同时清空深度/模板缓冲区。

**3．Depth**：深度缓冲区元素的初始值，它必须是一个在 $0 \leqslant x \leqslant 1$ 之间的浮点值。

**4．Stencil**：模板缓冲区元素的初始值，它必须是一个在 $0 \leqslant n \leqslant 255$ 之间的整数值。

我们在示例的每帧中都会调用这个方法，例如：

```
void MirrorApp::DrawScene()
{
    md3dImmediateContext->ClearRenderTargetView(mRenderTargetView,
              reinterpret_cast<const float*>(&Colors::Black));
    md3dImmediateContext->ClearDepthStencilView(mDepthStencilView,
              D3D11_CLEAR_DEPTH|D3D11_CLEAR_STENCIL,1.0f,0);
}
```

# 10.2 模板测试

如前所述，我们可以使用模板缓冲区来阻止像素片段渲染到后台缓冲区的某些区域。判断一个特定像素是否可以写入后台缓冲区的操作称为模板测试（stencil test），其实现过程为：

```
if( StencilRef & StencilReadMask ⊴ Value &StencilReadMask)
    accept pixel
else
    reject pixel
```

模板测试是在像素光栅化时（即输出合并阶段）进行的。在启用模板功能之后，每个光栅化像素都要与下面的两个操作数进行模板测试：

1．左操作数（LHS）由应用程序指定的一个模板参考值（**StencilRef**）和一个模板掩码（**StencilReadMask**）进行按位**与**运算得到。

2．右操作数（RHS）由当前像素在模板缓冲区中的对应值（**Value**）和一个模板掩码（**StencilReadMask**）进行按位**与**运算得到。

注意，LHS 和 RHS 中的 **StencilReadMask** 是相同的。然后，通过应用程序指定的比较函数⊴对 LHS 和 RHS 进行比较，返回 true 或 false。当测试结果为 true 时，说明该像素可以写入后台缓冲区（假设深度测试也通过）。当测试结果为 false 时，说明应该丢弃该像素，不把它写入后台缓冲区。当然，如果一个像素未能通过模板测试，那么它的深度值也不会被写入深度缓冲区。

运算符⊴可以是 **D3D11_COMPARISON_FUNC** 枚举类型定义的任何一个函数：

```
typedef enum D3D11_COMPARISON_FUNC
{
    D3D11_COMPARISON_NEVER = 1,
    D3D11_COMPARISON_LESS = 2,
    D3D11_COMPARISON_EQUAL = 3,
    D3D11_COMPARISON_LESS_EQUAL = 4,
    D3D11_COMPARISON_GREATER = 5,
    D3D11_COMPARISON_NOT_EQUAL = 6,
    D3D11_COMPARISON_GREATER_EQUAL = 7,
    D3D11_COMPARISON_ALWAYS = 8,
} D3D11_COMPARISON_FUNC;
```

1. **D3D11_COMPARISON_NEVER**：始终返回 false。
2. **D3D11_COMPARISON_LESS**：用<运算符代替⊴。
3. **D3D11_COMPARISON_EQUAL**：用==运算符代替⊴。
4. **D3D11_COMPARISON_LESS_EQUAL**：用≤运算符代替⊴。
5. **D3D11_COMPARISON_GREATER**：用>运算符代替⊴。
6. **D3D11_COMPARISON_NOT_EQUAL**：用!=运算符代替⊴。
7. **D3D11_COMPARISON_GREATER_EQUAL**：用≥运算符代替⊴。
8. **D3D11_COMPARISON_ALWAYS**：始终返回 true。

# 10.3 深度/模板状态块

当创建 **ID3D11DepthStencilState** 接口时，第一步是要填充一个 **D3D11_DEPTH_STENCIL_DESC** 实例：

```
typedef struct D3D11_DEPTH_STENCIL_DESC{
    BOOL DepthEnable;//默认 True

    // 默认:D3D11_DEPTH_WRITE_MASK_ALL
    D3D11_DEPTH_WRITE_MASK DepthWriteMask;
    // 默认:D3D11_COMPARISON_LESS
    D3D11_COMPARISON_FUNC DepthFunc;

    BOOL StencilEnable;// 默认:False
    UINT8 StencilReadMask;// 默认:0xff
    UINT8 StencilWriteMask;// 默认:0xff
    D3D11_DEPTH_STENCILOP_DESC FrontFace;
    D3D11_DEPTH_STENCILOP_DESC BackFace;
}D3D11_DEPTH_STENCIL_DESC;
```

## 10.3.1 深度设置

**1．DepthEnable**：当设为 true 时，表示启用深度测试；当设为 false 时，表示禁用深度测试。当禁用深度测试时，绘图顺序非常重要，因为在这种情况下障碍物后面的像素片段也会被绘制出来（回顾 4.1.5 节）。如果禁用深度测试，那么无论 **DepthWriteMask** 设定何值，深度缓冲区中的元素都不会被更新。

**2．DepthWriteMask**：可设为 **D3D11_DEPTH_WRITE_MASK_ZERO** 或 **D3D11_DEPTH_WRITE_MASK_ALL**。这两个标志值不能同时使用。当 **DepthEnable** 设为 true 时，**D3D11_DEPTH_WRITE_MASK_ZERO** 表示禁用深度缓冲区的写入功能，但深度测试依然有效。**D3D11_DEPTH_WRITE_MASK_ALL** 表示启用深度缓冲区的写入功能；当深度/模板测试都通过时，新的深度值会被写入深度缓冲区。

**3．DepthFunc**：一个用于描述深度测试函数的 **D3D11_COMPARISON_FUNC** 枚举类型成员。我们一般使用 **D3D11_COMPARISON_LESS** 实现普通的深度测试，如 4.1.5 节所述。也就是，当像素片段的深度值比之前写入后台缓冲区的像素的深度值小时，接受该像素片段（即，将该像素片段写入后台缓冲区，将该像素片段的深度值写入深度缓冲区）。另外，Direct3D 支持自定义的深度测试函数。如果有必要的话，你可以自定义深度测试函数。

## 10.3.2 模板设置

**1．StencilEnable**：当设为 true 时，表示启用模板测试；当设为 false 时，表示禁用模板测试。

**2．StencilReadMask**：在模板测试时使用的掩码：

```
if( StencilRef & StencilReadMask ⊴ Value &StencilReadMask)
    accept pixel
else
    reject pixel
```

默认掩码不屏蔽任何二进制位：

```
#define D3D11_DEFAULT_STENCIL_READ_MASK (0xff)
```

**3．StencilWriteMask**：当更新模板缓冲区时，我们可以通过掩码来屏蔽某些二进制位，不让它们存入模板缓冲区。例如，当你希望屏蔽前 4 位数据时，可将掩码设为 0x0f。默认掩码不屏蔽任何二进制位：

```
#define D3D11_DEFAULT_STENCIL_WRITE_MASK (0xff)
```

**4．FrontFace**：一个已填充的 **D3D11_DEPTH_STENCILOP_DESC** 结构体，它告诉模板缓冲区如何处理朝前的三角形。

**5．BackFace**：一个已填充的 **D3D11_DEPTH_STENCILOP_DESC** 结构体，它告诉模板缓冲区如何处理朝后的三角形。

```
typedef struct D3D11_DEPTH_STENCILOP_DESC {
    D3D11_STENCIL_OP StencilFailOp;// Default:D3D11_STENCIL_OP_KEEP
    D3D11_STENCIL_OP                          StencilDepthFailOp;//
Default:D3D11_STENCIL_OP_KEEP
    D3D11_STENCIL_OP StencilPassOp; // Default:D3D11_STENCIL_OP_KEEP
    D3D11_COMPARISON_FUNC     StencilFunc;     //     Default:D3D11_
```

```
COMPARISON_ALWAYS
} D3D11_DEPTH_STENCILOP_DESC;
```

**1．StencilFailOp：D3D11_STENCIL_OP** 枚举类型成员，它描述了当一个像素片段的模板测试失败时，应如何更新模板缓冲区。

**2．StencilDepthFailOp：D3D11_STENCIL_OP** 枚举类型成员，它描述了当一个像素片段的模板测试成功而深度测试失败时，应如何更新模板缓冲区。

**3．StencilPassOp：D3D11_STENCIL_OP** 枚举类型成员，它描述了当一个像素片段的模板测试和深度测试均成功时，应如何更新模板缓冲区。

**4．StencilFunc：D3D11_COMPARISON_FUNC** 枚举类型成员，指定模板测试时使用的比较函数。

```
typedef enum D3D11_STENCIL_OP
{
    D3D11_STENCIL_OP_KEEP = 1,
    D3D11_STENCIL_OP_ZERO = 2,
    D3D11_STENCIL_OP_REPLACE = 3,
    D3D11_STENCIL_OP_INCR_SAT = 4,
    D3D11_STENCIL_OP_DECR_SAT = 5,
    D3D11_STENCIL_OP_INVERT = 6,
    D3D11_STENCIL_OP_INCR = 7,
    D3D11_STENCIL_OP_DECR = 8,
} D3D11_STENCIL_OP;
```

**1．D3D11_STENCIL_OP_KEEP**：不更新模板缓冲区；也就是，当前值保持不变。

**2．D3D11_STENCIL_OP_ZERO**：将模板缓冲区元素设为 0。

**3．D3D11_STENCIL_OP_REPLACE**：以模板测试中的模板参考值（**StencilRef**）替换模板缓冲区元素。注意，当我们将深度/模板状态块绑定到渲染管线上时（参见 10.3.3 节），**StencilRef** 值就已经被确定下来了。

4．**D3D11_STENCIL_OP_INCR_SAT**：递增模板缓冲区元素。如果递增之后的值大于最大值（比如，255 是 8 位模板缓冲区的最大值），则将其舍入为最大值。

**5．D3D11_STENCIL_OP_DECR_SAT**：递减模板缓冲区元素。如果递减之后的值小于 0，则将其舍入为 0。

**6．D3D11_STENCIL_OP_INVERT**：反转模板缓冲区元素的二进制位。

**7．D3D11_STENCIL_OP_INCR**：递增模板缓冲区元素。如果递增之后的值大于最大值（比如，255 是 8 位模板缓冲区的最大值），则将其折反为 0。

**8．D3D11_STENCIL_OP_DECR**：递减模板缓冲区元素。如果递减之后的值小于 0，则将其折反为最大值。

**注意**：我们一般不使用 **BackFace** 参数，因为当启用背面消隐时，Direct3D 根本不会渲染朝后的多边形。不过，有时我们会为了实现某些绘图算法或绘制透明几何体（比如在绘制铁丝立方体时，我们希望看到铁丝网立方体的背面）而渲染朝后的多边形。在这种情况下应该使用 **BackFace** 参数。

## 10.3.3 创建和绑定深度/模板状态

在填充 **D3D11_DEPTH_STENCIL_DESC** 结构体之后，我们可以调用如下方法获取一

个指向 **ID3D11DepthStencilState** 接口的指针：

```
HRESULT ID3D11Device::CreateDepthStencilState(
    const D3D11_DEPTH_STENCIL_DESC *pDepthStencilDesc,
    ID3D11DepthStencilState **ppDepthStencilState);
```

**1. pDepthStencilDesc**：一个已填充的 **D3D11_DEPTH_STENCIL_DESC** 结构体的指针，该结构体描述了所要创建的深度/模板状态块。

**2．ppDepthStencilState**：返回创建后的 **ID3D11DepthStencilState** 接口的指针。

在创建 **ID3D11DepthStencilState** 接口后，我们使用如下方法将它绑定到管线的输出合并器阶段：

```
void ID3D11DeviceContext::OMSetDepthStencilState(
    ID3D11DepthStencilState *pDepthStencilState,
    UINT StencilRef);
```

**1．pDepthStencilState**：深度/模板状态块的指针。

**2．StencilRef**：模板测试使用的 32 位模板参考值。

与其他状态块相同，深度/模板状态也有一个默认值（它使用普通的深度测试，并禁用模板测试）。通过给 **OMSetDepthStencilState** 方法的第 1 个参数传递一个空值就可以将深度/模板状态恢复为默认值。

```
// 恢复为默认值
md3dImmediateContext->OMSetDepthStencilState(0, 0);
```

## 10.3.4 effect 文件中的深度/模板状态

在 effect 文件中可以直接定义和设置深度/模板状态：

```
DepthStencilState DSS
{
    DepthEnable = true;
    DepthWriteMask = Zero;
    StencilEnable = true;
    StencilReadMask = 0xff;
    StencilWriteMask = 0xff;
    FrontFacetencilFunc = Always;
    FrontFaceStencilPass =  Incr;
    FrontFaceStencilFail = Keep;
    BackFaceStencilFunc = Always;
    BackFaceStencilPass =  Incr;
    BackFaceStencilFail = Keep;
} ;
...
technique11 Tech
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_5_0,VS()));
```

```
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,PS()));
        SetDepthStencilState(DSS, 0);
    }
}
```

在深度/模板状态对象中指定的些值与在 C++结构体中指定的值基本相同，只是省去了一些前缀。例如，我们在效果代码中指定是 **INCR**，而不是 **D3D11_STENCIL_OP_INCR**。顺便提一句，我们指定的状态值是不区分大小写的；例如，INCR 等价于 Incr。

# 10.4 平面镜像的实现

在自然界中有许多物体的表面都非常光滑，可以像镜子一样反射周围的物体。本节介绍了如何在 3D 应用程序中模拟镜像效果。为简单起见，我们降低了任务难度，只在平面上实现镜像效果。例如，一辆光滑的汽车可以反射周围的物体；但是，车身是一个平滑曲面，而非平面。我们不选择这样的物体。我们将在光滑的大理石地板或挂在墙上的镜子中渲染物体的映像——换句话说，我们只实现平面上的镜像效果。

要在程序中实现镜像效果，必须解决两个问题。首先，我们必须知道如何在一个任意平面上反射物体，正确地绘制该物体的映像。其次，我们只能在镜子里面显示映像；也就是，我们必须以某种方式将一个表面“标记”为镜子，然后在渲染时只在镜子里面绘制物体映像。回顾图 10.1，它最先引入了这一概念。

第一个问题可以很容易地通过解析几何来解决，具体请参见附录 C。第二个问题可以通过模板缓冲区来解决。

## 10.4.1 概述

**注意**：当绘制映像时，我们还需要在镜子平面上反射光源。否则，映像中的光照会显得很不真实。

图 10.2 说明了要绘制一个物体的映像，我们必须在镜子平面上对它进行反射。不过，这会出现图 10.1 所示的问题。即，物体映像（在本例中是头骨）会被渲染到镜子之外的区域（例如，墙面）。映像只应该显示在镜子里面。我们可以使用模板缓冲区来解决一问题，因为模板缓冲区可以阻止像素渲染到后台缓冲区的某些区域上。所以，我们可以使用模板缓冲区来控制头骨的映像，避免映像渲染到镜子之外的区域。下面给出了具体的实现步骤：

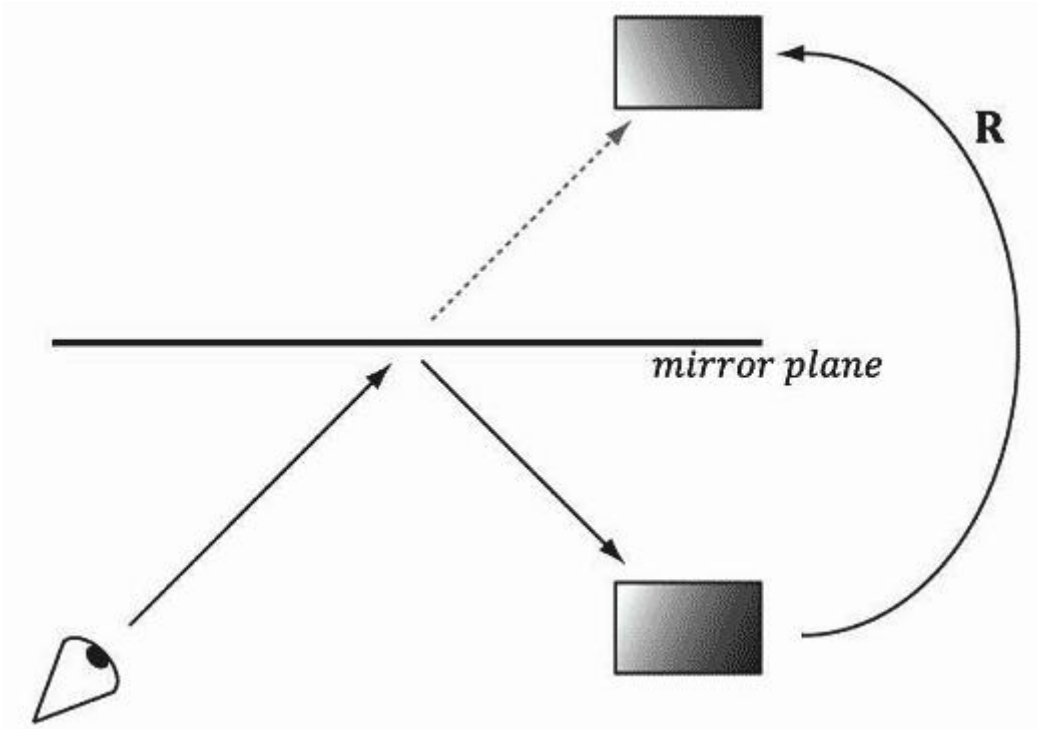



**图 10.2 观察点从镜子中看到的立方体映像。要模拟一效果，我们需要在镜子平面上反射立方体，然后将立方体映像绘制出来。**

1．将地板、墙壁和头骨（不包括镜子）渲染到后台缓冲区。注意，一步不修改模板缓冲区。

2．将模板缓冲区清为 0。图 10.3 展示了此时的后台缓冲区和模板缓冲区。

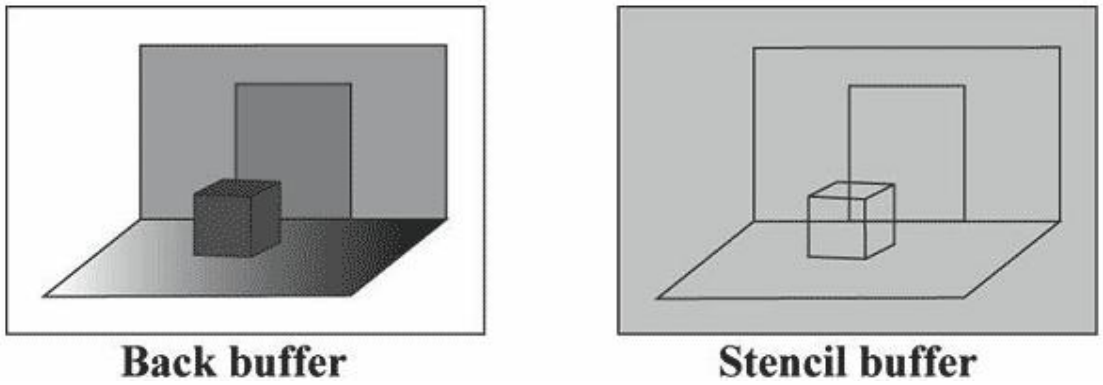



**图 10.3 将场景渲染到后台缓冲区，并将模板缓冲区清为 0（由浅灰色表示）。模板缓冲区上的黑色轮廓线用于说明后台缓冲区像素与模板缓冲区像素之间的对应关系——它们并不代表绘制在模板缓冲区上的任何数据。**

3．把镜子只渲染到模板缓冲区。我们可以通过设置以下混合状态禁止颜色写入到后台缓冲区中：

```
D3D11_RENDER_TARGET_BLEND_DESC::RenderTargetWriteMask = 0;
```

通过以下设置禁止写入到深度缓冲区中：

```
D3D11_DEPTH_STENCIL_DESC::DepthWriteMask                    =
D3D11_DEPTH_WRITE_MASK_ZERO;
```

在将镜子绘制到模板缓冲时，我们将模板测试函数设为 **D3D11_COMPARISON_ALWAYS**（始终成功），并指定当模板测试成功时将模板缓冲区元

素替换（**D3D11_STENCIL_OP_REPLACE**）为 1（**StencilRef**）。将 **StencilDepthFailOp** 设为 **D3D11_STENCIL_OP_KEEP**，当深度测试失败时不更新模板缓冲区（如果头骨挡住了镜子的某一部分，那么就会发生这种情况）。由于我们只将镜子绘制到模板缓冲区，所以在模板缓冲区中只有镜子的可视区域对应的像素为 1，而其他像素均为 0。图 10.4 展示了更新后的模板缓冲区。实际上，我们是给镜子在模板缓冲区中的可视区域做了标记。

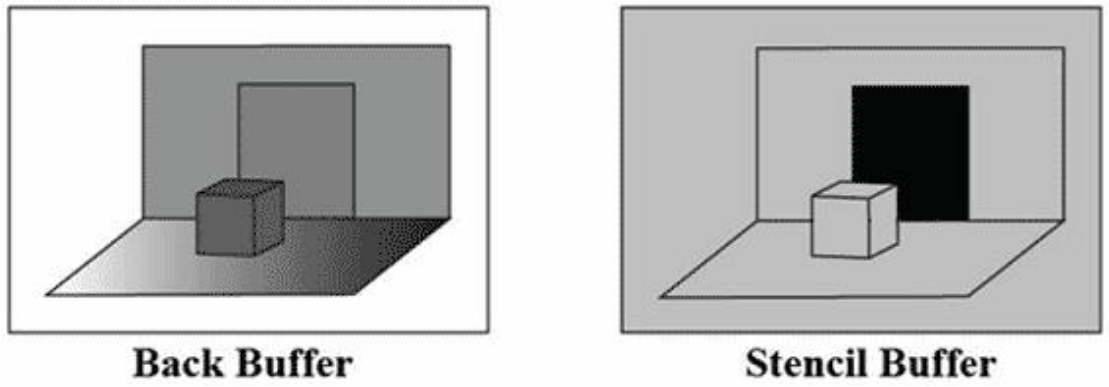



**图 10.4 把镜子渲染到模板缓冲区，这个操作的实际上是在模板缓冲区中标记了镜子的可视区域。实心黑色区域的模板元素值为 1。注意，被盒子挡住的区域不会设为 1，因为这一部分根本无法通过深度测试（盒子挡住了镜子前面的这一部分）。**

4．现在我们将头骨映像渲染到后台缓冲区和模板缓冲区。但是要记住，只有通过了模板测试的像素片段才能渲染到后台缓冲区中。这次，我们要将模板测试函数设为 **D3D11_COMPARISON_EQUAL**，使模板元素为 1 时测试成功。通过一方式，头骨映像只会渲染到模板元素为 1 的区域中。由于在模板缓冲区中只有镜子的可视区域的模板元素为 1，所以头骨映像只会被渲染到镜子里面。

5．最后，我们将镜子绘制到后台缓冲区。但是，为了为了能显示镜子之后的头骨镜像，我们需要使用透明混合绘制镜子，如果不这样做，那么镜子就会挡住位于它后面的头骨镜像。要实现这个效果，我们只需定义一个镜子用的材质实例，将漫反射分量的 alpha 通道设置为 0.5，这样镜子表示镜子是半透明的，然后就可以像 9.5.4 节那样绘制半透明的镜子了：

```
mMirrorMat.Ambient  = XMFLOAT4(0.5f, 0.5f, 0.5f, 1.0f);
mMirrorMat.Diffuse  = XMFLOAT4(1.0f, 1.0f, 1.0f, 0.5f);
mMirrorMat.Specular = XMFLOAT4(0.4f, 0.4f, 0.4f, 16.0f);
```

上述设置给出以下混合方程：

$\mathbf{C} = 0.5 \cdot \mathbf{C}_{src} + 0.5 \cdot \mathbf{C}_{dst}$

假设我们已将头骨镜像的像素发送到后台缓冲区中，则 50%的颜色来自于镜子（源），50%的颜色来自于头骨（目标）。

## 10.4.2 定义深度/模板状态

要实现上述算法，我们必须定义两个深度/模板状态。一个用于在模板缓冲区上绘制镜子，标记镜子的可视区域。另一个用于绘制头骨映像，使映像只显示在镜子里面。

```
//
// MarkMirrorDSS
//
```

```
D3D11_DEPTH_STENCIL_DESC mirrorDesc;
mirrorDesc.DepthEnable      = true;
mirrorDesc.DepthWriteMask   = D3D11_DEPTH_WRITE_MASK_ZERO;
mirrorDesc.DepthFunc        = D3D11_COMPARISON_LESS;
mirrorDesc.StencilEnable    = true;
mirrorDesc.StencilReadMask  = 0xff;
mirrorDesc.StencilWriteMask = 0xff;

mirrorDesc.FrontFace.StencilFailOp      = D3D11_STENCIL_OP_KEEP;
mirrorDesc.FrontFace.StencilDepthFailOp = D3D11_STENCIL_OP_KEEP;
mirrorDesc.FrontFace.StencilPassOp      = D3D11_STENCIL_OP_REPLACE;
mirrorDesc.FrontFace.StencilFunc        = D3D11_COMPARISON_ALWAYS;

// We are not rendering backfacing polygons, so these settings do not
matter.
mirrorDesc.BackFace.StencilFailOp       = D3D11_STENCIL_OP_KEEP;
mirrorDesc.BackFace.StencilDepthFailOp  = D3D11_STENCIL_OP_KEEP;
mirrorDesc.BackFace.StencilPassOp       = D3D11_STENCIL_OP_REPLACE;
mirrorDesc.BackFace.StencilFunc         = D3D11_COMPARISON_ALWAYS;

HR(device->CreateDepthStencilState(&mirrorDesc, &MarkMirrorDSS));

//
// DrawReflectionDSS
//

D3D11_DEPTH_STENCIL_DESC drawReflectionDesc;
drawReflectionDesc.DepthEnable      = true;
drawReflectionDesc.DepthWriteMask   = D3D11_DEPTH_WRITE_MASK_ALL;
drawReflectionDesc.DepthFunc        = D3D11_COMPARISON_LESS;
drawReflectionDesc.StencilEnable    = true;
drawReflectionDesc.StencilReadMask  = 0xff;
drawReflectionDesc.StencilWriteMask = 0xff;

drawReflectionDesc.FrontFace.StencilFailOp                      =
D3D11_STENCIL_OP_KEEP;
drawReflectionDesc.FrontFace.StencilDepthFailOp                 =
D3D11_STENCIL_OP_KEEP;
drawReflectionDesc.FrontFace.StencilPassOp = D3D11_STENCIL_OP_KEEP;
drawReflectionDesc.FrontFace.StencilFunc                        =
D3D11_COMPARISON_EQUAL;

// We are not rendering backfacing polygons, so these settings do not
matter.
```

```
drawReflectionDesc.BackFace.StencilFailOp                                =
D3D11_STENCIL_OP_KEEP;
drawReflectionDesc.BackFace.StencilDepthFailOp                           =
D3D11_STENCIL_OP_KEEP;
drawReflectionDesc.BackFace.StencilPassOp = D3D11_STENCIL_OP_KEEP;
drawReflectionDesc.BackFace.StencilFunc                                  =
D3D11_COMPARISON_EQUAL;

HR(device->CreateDepthStencilState(&drawReflectionDesc,
&DrawReflectionDSS));
```

## 10.4.3 绘制场景

下面是 draw 方法的代码。为了突出重点，我们略去了不相关的细节，比如设置常量缓冲区的值（完整的细节请参见本例源代码）。

```
//
// 镜子只绘制到模板缓冲
//

activeTech->GetDesc( &techDesc );
for(UINT p = 0; p < techDesc.Passes; ++p)
{
    ID3DX11EffectPass* pass = activeTech->GetPassByIndex( p );

    md3dImmediateContext->IASetVertexBuffers(0, 1, &mRoomVB, &stride,
&offset);

    // Set per object constants.
    XMMATRIX world = XMLoadFloat4x4(&mRoomWorld);
    XMMATRIX                    worldInvTranspose                    =
MathHelper::InverseTranspose(world);
    XMMATRIX worldViewProj = world*view*proj;

    Effects::BasicFX->SetWorld(world);
    Effects::BasicFX->SetWorldInvTranspose(worldInvTranspose);
    Effects::BasicFX->SetWorldViewProj(worldViewProj);
    Effects::BasicFX->SetTexTransform(XMMatrixIdentity());

    // 不写入渲染目标
    md3dImmediateContext->OMSetBlendState(RenderStates::NoRenderTa
rgetWritesBS, blendFactor, 0xffffffff);

    // 将镜子可见部分的像素绘制到模板缓冲中。
```

```
    // 但不要将镜子的深度信息写入深度缓冲中，否则会遮挡之后的头骨镜像。
    md3dImmediateContext->OMSetDepthStencilState(RenderStates::MarkMirrorDSS, 1);

    pass->Apply(0, md3dImmediateContext);
    md3dImmediateContext->Draw(6, 24);

    // 恢复之前的状态
    md3dImmediateContext->OMSetDepthStencilState(0, 0);
    md3dImmediateContext->OMSetBlendState(0, blendFactor, 0xffffffff);
}


//
// 绘制头骨镜像
//
activeSkullTech->GetDesc( &techDesc );
for(UINT p = 0; p < techDesc.Passes; ++p)
{
    ID3DX11EffectPass* pass = activeSkullTech->GetPassByIndex( p );

    md3dImmediateContext->IASetVertexBuffers(0, 1, &mSkullVB, &stride, &offset);
    md3dImmediateContext->IASetIndexBuffer(mSkullIB, DXGI_FORMAT_R32_UINT, 0);

    XMVECTOR mirrorPlane = XMVectorSet(0.0f, 0.0f, 1.0f, 0.0f); // xy plane
    XMMATRIX R = XMMatrixReflect(mirrorPlane);
    XMMATRIX world = XMLoadFloat4x4(&mSkullWorld) * R;
    XMMATRIX worldInvTranspose = MathHelper::InverseTranspose(world);
    XMMATRIX worldViewProj = world*view*proj;

    Effects::BasicFX->SetWorld(world);
    Effects::BasicFX->SetWorldInvTranspose(worldInvTranspose);
    Effects::BasicFX->SetWorldViewProj(worldViewProj);
    Effects::BasicFX->SetMaterial(mSkullMat);

    // 保存之前的光照方向，然后反射光照方向
    XMFLOAT3 oldLightDirections[3];
    for(int i = 0; i < 3; ++i)
    {
```

```
        oldLightDirections[i] = mDirLights[i].Direction;

        XMVECTOR lightDir = XMLoadFloat3(&mDirLights[i].Direction);
        XMVECTOR reflectedLightDir =
XMVector3TransformNormal(lightDir, R);
        XMStoreFloat3(&mDirLights[i].Direction, reflectedLightDir);
    }

    Effects::BasicFX->SetDirLights(mDirLights);

    // 在反射中剔除顺时针绕行的三角形
    md3dImmediateContext->RSSetState(RenderStates::CullClockwiseRS)
;

    // 根据模板标记绘制镜子可见部分中的头骨镜像
    md3dImmediateContext->OMSetDepthStencilState(RenderStates::Dra
wReflectionDSS, 1);
    pass->Apply(0, md3dImmediateContext);
    md3dImmediateContext->DrawIndexed(mSkullIndexCount, 0, 0);

    // 恢复默认状态。
    md3dImmediateContext->RSSetState(0);
    md3dImmediateContext->OMSetDepthStencilState(0, 0);

    // 恢复光照方向
    for(int i = 0; i < 3; ++i)
    {
        mDirLights[i].Direction = oldLightDirections[i];
    }

    Effects::BasicFX->SetDirLights(mDirLights);
}

//
// 将镜子绘制到后台缓冲区，需要镜像透明混合，
// 这样才能显示镜子后面的镜像
//

activeTech->GetDesc( &techDesc );
for(UINT p = 0; p < techDesc.Passes; ++p)
{
    ID3DX11EffectPass* pass = activeTech->GetPassByIndex( p );

    md3dImmediateContext->IASetVertexBuffers(0, 1, &mRoomVB, &stride,
```

```
&offset);

    // Set per object constants.
    XMMATRIX world = XMLoadFloat4x4(&mRoomWorld);
    XMMATRIX                    worldInvTranspose                    =
MathHelper::InverseTranspose(world);
    XMMATRIX worldViewProj = world*view*proj;

    Effects::BasicFX->SetWorld(world);
    Effects::BasicFX->SetWorldInvTranspose(worldInvTranspose);
    Effects::BasicFX->SetWorldViewProj(worldViewProj);
    Effects::BasicFX->SetTexTransform(XMMatrixIdentity());
    Effects::BasicFX->SetMaterial(mMirrorMat);
    Effects::BasicFX->SetDiffuseMap(mMirrorDiffuseMapSRV);

    // 镜子
    md3dImmediateContext->OMSetBlendState(RenderStates::Transparen
tBS, blendFactor, 0xffffffff);
    pass->Apply(0, md3dImmediateContext);
    md3dImmediateContext->Draw(6, 24);
}
```

## 10.4.4 环绕顺序和反射

当三角形被反射到平面上时，它的环绕顺序和平面法线都不会翻转。所以，在反射之后，原来向外的法线会变成向内的法线（参见图 10.5）。为了纠正一错误，我们必须告诉 Direct3D 将逆时针环绕的三角形视为朝前的三角形，将顺时针环绕的三角形视为朝后的三角形（与我们在 5.10.2 节的约定恰好相反）。这样可以有效地翻转法线方向，使它们在反射之后仍然向外。我们通过设置如下光栅器状态来翻转环绕顺序：

```
//
// CullClockwiseRS
//

// 注意：把朝前的三角形定义为逆时针方向绕行使我们仍然可以进行背面剔除。
// 如果我们关闭背面剔除，就不得不要考虑 D3D11_DEPTH_STENCIL_DESC 中的
BackFace 属性了。
D3D11_RASTERIZER_DESC cullClockwiseDesc;
ZeroMemory(&cullClockwiseDesc, sizeof(D3D11_RASTERIZER_DESC));
cullClockwiseDesc.FillMode = D3D11_FILL_SOLID;
cullClockwiseDesc.CullMode = D3D11_CULL_BACK;
cullClockwiseDesc.FrontCounterClockwise = true;
cullClockwiseDesc.DepthClipEnable = true;
```

```
HR(device->CreateRasterizerState(&cullClockwiseDesc,
&CullClockwiseRS));
```

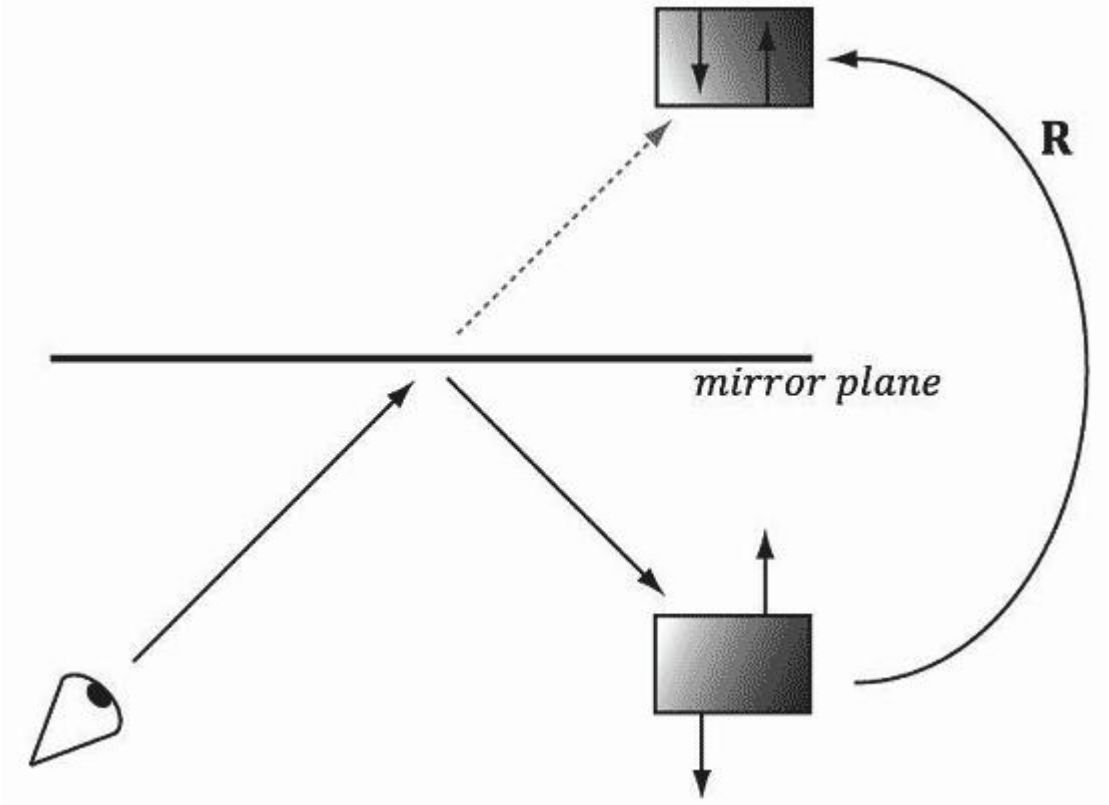



**图 10.5 多边形法线不会随着映像而翻转，在反射之后，原来向外的法线会变成向内的法线。**

# 11.1 几何着色器编程

若我们不使用曲面细分阶段，则几何着色器（geometry shader）阶段是一个可选阶段，它位于顶点着色器和像素着色器阶段之间。顶点着色器以顶点作为输入数据，而几何着色器以完整的图元作为输入数据。例如，当我们绘制三角形列表时，几何着色器处理的是列表中的每个三角形 **T**：

```
for(UINT i = 0; i < numTriangles; ++i)
    OutputPrimitiveList = GeometryShader(T[i].vertexList);
```

注意，这里是将每个三角形的 3 个顶点作为几何着色器的输入数据，几何着色器的输出为图元列表。顶点着色器无法创建或销毁顶点，而几何着色器的主要优点就是它可以创建或销毁几何体；这样就可以在 GPU 上实现一些有趣的效果。例如，几何着色器可以将输入图元扩展为一个或多个其他图元，或者根据一些条件屏蔽某些图元的输出。注意，输出图元可以与输入图元的类型不同；例如，几何着色器的常见用途是将一个点扩展为一个四边形（即，两个三角形）。

几何着色器的输出图元由一个顶点列表来描述。在顶点离开几何着色器之前，顶点坐标必须变换到齐次裁剪空间。在几何着色器阶段之后，顶点列表描述的是齐次裁剪空间中的图元。与往常一样，这些顶点会被投影（齐次除法），随后进行光栅化处理。

**学习目标**

1．学习如何编写几何着色器。
2．理解如何使用几何着色器实现高效的广告牌算法。
3．了解自动生成的图元 ID 以及它的一些用途。
4．学习如何创建和使用纹理数组，以及纹理数组的一些用途。
5．理解 alpha-to-coverage 是如何改进 alpha 剪裁中的锯齿问题的。

# 10.1 几何着色器编程

几何着色器编程与顶点/像素着色器编程非常相似，只是略有一些差异。下面的代码展示了它的一般格式：

```
[maxvertexcount(N)]
void ShaderName (
    PrimitiveType InputVertexType InputName [NumElements],
    inout StreamOutputObject<OutputVertexType>OutputName)
{
    // Geometry shader body...
}
```

首先，我们必须指定每次调用几何着色器时所能输出的顶点的最大数量。这一工作通过在着色器定义之前指定 **maxvertexcount** 属性来实现：

```
[maxvertexcount(N)]
```

其中，**N** 是每次调用几何着色器时所能输出的顶点的最大数量。几何着色器每次输出的顶点数量都可以不同，只要不超过指定的最大值就没问题。从性能方面考虑，**maxvertexcount** 应竟可能小，[NVIDIA08]指出当 GS 的输出介于 1-20 个标量之间时性能最佳，在 27-40 个标量之间则性能损失 50%。每次调用输出的标量大小是指 **maxvertexcount** 的生成和在输出顶点类型结构中的标量数量。有了这个限定，实际工作会非常困难，你要么接受性能的损失，要么选择不使用几何着色器而用别的方法代替；但是，我们必须要考虑到别的方法也会有缺点，可能几何着色器反而是个较好的选择。而且，[NVIDIA08]中的推荐设置发表于 2008（几何着色器第一次发布），所以现在可能已经改进过了。

几何着色器有两个参数：一个输入参数和一个输出参数。（其实，它的参数不只两个，我们会在后面的 11.2.4 节专门讨论个话题。）输入参数总是一个顶点数组，它可以表示：单个顶点、由两个顶点构成的直线、由 3 个顶点构成的三角形、由 4 个顶点构成的带有邻接信息的直线、由 6 个顶点构成的带有邻接信息的三角形。输入的顶点类型与顶点着色器返回的顶点类型相同（例如，**VertexOut**）。输入参数必须加上一个图元类型前缀，描述将要输入到几何着色器的图元类型。可以使用的图元类型包括：

**1．point**：输入图元为点。
**2．line**：输入图元为直线（列表或线带）。
**3．triangle**：输入图元为三角形（列表或线带）。
**4．lineadj**：输入图元为带有邻接信息的直线（列表或线带）。
**5．triangleadj**：输入图元为带有邻接信息的三角形（列表或线带）。

**注意**：几何着色器的输入图元总是一个完整的图元（例如，由两个顶点构成一条直线、由三个顶点构成一个三角形）。这样，几何着色器就不需要区分列表和线带了。例如，在绘制三角形线带时，几何着色器会处理线带中的每个三角形，而且每个三角形的 3 个顶点都会

作为输入数据传入到几何着色器中。这会导致额外的工作，因为几何着色器会重复处理被多个图元共享的顶点。

输出参数总是带有 **inout** 修饰符，并且是一个流类型（stream type）对象。流类型用于存储由几何着色器输出的几何体顶点列表。几何着色器使用内置的 **Append** 方法向输出流添加顶点：

```
void StreamOutputObject<OutputVertexType>::Append(OutputVertexType
v);
```

流类型是一种模板类型（template type），其中的模板参数用于指定输出顶点的类型（例如，**GeoOUT**）。这里有 3 种可以使用的流类型：

**1．PointStream<OutputVertexType>**：描述单个点的顶点列表。

**2．LineStream<OutputVertexType>**：描述直线线带的顶点列表。

**3．TriangleStream<OutputVertexType>**：描述三角形线带的顶点列表。

几何着色器以图元为单位输出顶点；输出图元的类型由流类型（**PointStream**、**LineStream**、**TriangleStream**）决定。对于直线和三角形来说，输出图元总是一个线带。不过，我们也可使用内置的 **RestartStrip** 方法模拟输出直线列表和三角形列表：

```
void StreamOutputObject<OutputVertexType>::RestartStrip();
```

例如，当你希望输出一个三角形列表时，你应该每输出 3 个顶点，调用一次 **RestartStrip** 方法（也就是，在每调用 3 次 Append 方法之后，调用一次 **RestartStrip** 方法）。

下面是一些几何着色器签名的例子：

```
// 例 1：GS 最多输出 4 个顶点。输入图元为一条线，输出为一个三角形线带。
//
[maxvertexcount(4)]
void GS(line VertexOutT gin[2],
    inout TriangleStream<GeoOut> triStream)
{
    // Geometry shader body...
}
//
// EXAMPLE 2: GS outputs at most 32 vertices.The input primitive is
a triangle. The output is a triangle strip.
//
[maxvertexcount(32)]
void GS(triangle VertexOut gin[3],
inout TriangleStream<GeoOut> triStream)
{
    // Geometry shader body...
}
//
// EXAMPLE 3: GS outputs at most 4 vertices. The input primitive
// is a point. The output is a triangle strip.
//
[maxvertexcount(4)]
void GS(point VertexOut gin[1],
inout TriangleStream<GeoOut> triStream)
```

```
{
    // Geometry shader body...
}
```

下面的几何着色器解释了 **Append** 和 **RestartStrip** 方法的用法；它输入一个三角形，对它进行细分（参见图 11.1），并输出 4 个细分后的三角形：

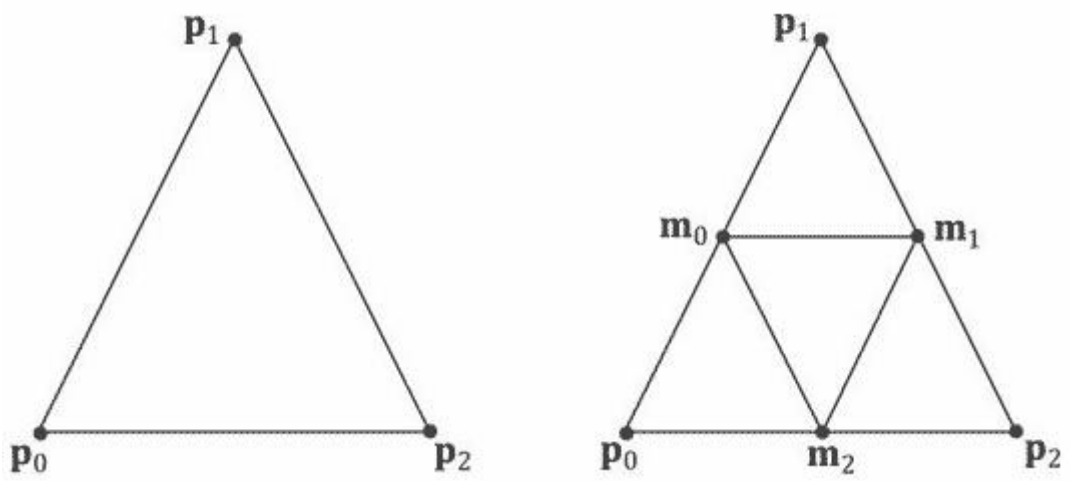



**图 11.1 将一个三角形细分为 4 个大小相同的小三角形。注意，3 个新的顶点是原三角形边的中点。**

```
struct VertexOut
{
    float3 posL     : POSITION;
    float3 normalL : NORMAL;
    float2 Tex : TEXCOORD;
};
struct GeoOut
{
     loat4 posH     : SV_POSITION;
    float3 posW     : POSITION;
    float3 normalW : NORMAL;
    float2 Tex      : TEXCOORD;
    float FogLerp   :  FOG;
}

void  Subdivide (VertexOut inVerts[3], out VertexOut outVerts[6])
{
    //       1
    //       *
    //      /  \
    //     /    \
    //   m0*-----*m1
    //    / \    /\
    //   /   \ /   \
```

```
    // *-----*-----*
    //0     m2     2

    VertexOut  m[3];
    // 计算每条边的中点
    m[0].PosL = 0.5f*(inVerts[0].PosL+inVerts[1].PosL);
    m[1].PosL  = 0.5f*(inVerts[1].PosL+inVerts[2].PosL);
    m[2].PosL = 0.5f*(inVerts[2].PosL+inVerts[0].PosL);
    // 投影到一个单位圆上
    m[0].PosL = normalize(m[0].PosL);
    m[1].PosL = normalize(m[1].PosL);
    m[2].PosL  = normalize(m[2].PosL);
    //  求得法线
    m[0].NormalL = m[0]. Pos L;
    m[1].NormalL  = m[1]. Pos L;
    m[2].NormalL  = m[2]. Pos L;
    // 插值求得纹理坐标
    m[0].Tex =  0.5f*(inVerts[0].Tex+inVerts[1].Tex);
    m[1].Tex =  0.5f*(inVerts[1].Tex+inVerts[2].Tex);
    m[2].Tex =  0.5f*(inVerts[2].Tex+inVerts[0].Tex);
    outVerts[0] = inVerts[0];
    outVerts[1] = m[0];
    outVerts[2] = m[2];
    outVerts[3] = m[1];
    outVerts[4] = inVerts[2];
    outVerts[5] = inVerts[1];
} ;

void OutputSubdivision(VertexOut v[6],inout TriangleStream<GeoOut>
triStream)
{
    GeoOut gout[6];
    [unroll]
    for(int i = 0; i  < 6; ++i)
    {
        // 转换到世界空间
        gout[i].PosW = mul(float4(v[i].PosL,1.0f), gWorld).xyz;
        gout[i].NormalW   =   mul(v[i].NormalL,(float3x3)gWorldInvT
ranspose);
        // 转换到齐次剪裁空间
        gout[i].PosH = mul(float4(v[i].PosL,1.0f), gWorldViewProj);
        gout[i].Tex = v[i].Tex;
    }
    //       1
```

```
    //        *
    //       /  \
    //      /    \
    //    m0*-----*m1
    //    / \    /\
    //   /   \ /   \
    // *-----*-----*
    //0     m2     2

    //  我们可以使用两个线带绘制细分三角形:
    //  第一个：底部的三个三角形
    //  第二个:顶部的一个三角形
     [unroll]
    for(int  j   =  0;  j   <  5;  ++j )
    {
        triStream.Append(gout[j]);
    }
    triStream.RestartStrip();
    triStream.Append(gout[1]);
    triStream.Append(gout[5]);
    triStream.Append(gout[3]);
}

[maxvertexcount(8)]
void GS(triangle VertexOut gin[3], inout TriangleStream<GeoOut>)
{
    VertexOut v[6];
    Subdivide(gin,v);
    OutputSubdivision(v, triStream);
}
```

**注意**：给定一个输入图元，几何着色器可以不对它进行输出。通过这一方式，几何着色器可以将输入的几何体“销毁”，这一功能在某些算法中非常有用。

**注意**：当几何着色器输出的顶点无法构成一个完整的图元时，这部分图元将被丢弃。

# 11.2 树广告牌演示程序

## 11.2.1 概述

当树与观察点的距离很远时，我们可以通过广告牌（billboard）技术来提高渲染效率。也就是，我们只在一个四边形上绘制树的 3D 图片，而不是渲染一个完整的 3D 树模型（参见图 11.2）。从远处看，你根本分辨不出是否使用了广告牌。不过，你必须确保广告牌始终面对摄像机（否则个假象就会被拆穿）。

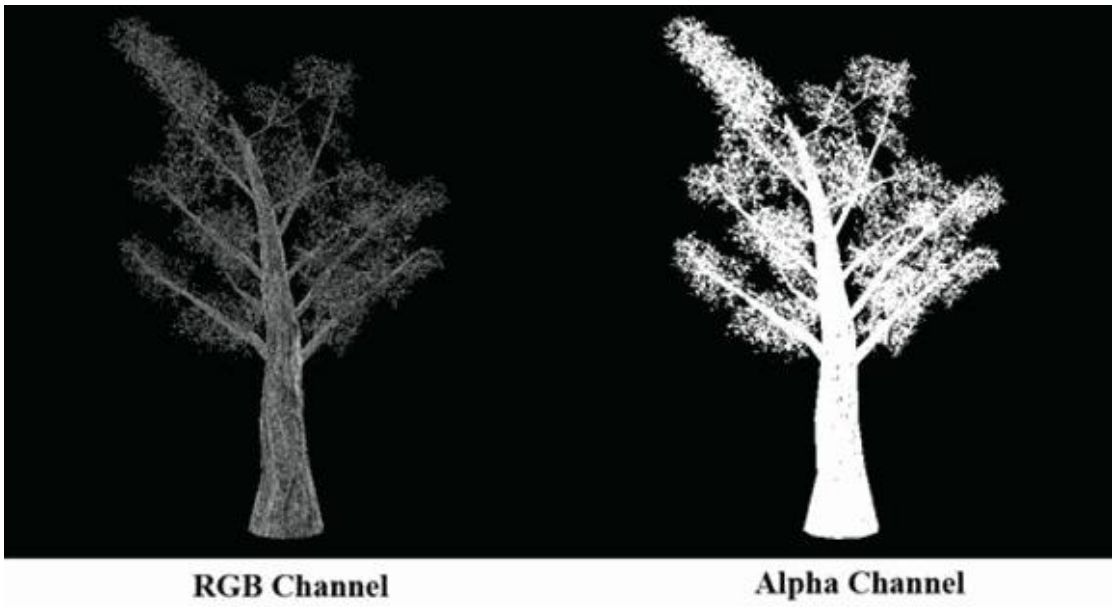



**图 11.2 带有 alpha 通道的树广告牌纹理。**

假设 *y* 轴垂直向上，*xz* 平面为地平面。树广告牌总与 *y* 轴对齐，只在 *xz* 平面上面对摄像机。图 11.3 展示了鸟瞰视图中的几个广告牌的局部坐标系——注意，广告牌始终“面对”摄像机。

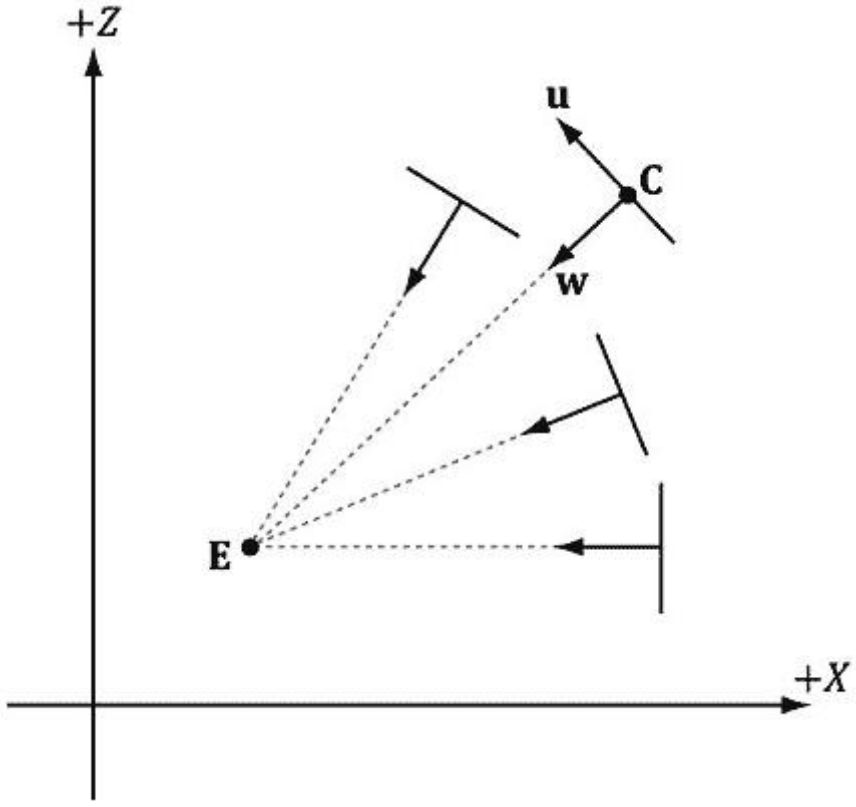



**图 11.3 广告牌面对摄像机。**

这样，只要在世界空间中给定广告牌的中心位置 **C**=($C_x$,$C_y$,$C_z$ )和摄像机的位置 **E**=($E_x$,$E_y$,$E_z$)，我们就有足够的信息来描述广告牌相对于世界空间的局部坐标系：

$$\mathbf{w} = \frac{(E_x - C_x, 0, E_z - C_z)}{\left\|(E_x - C_x, 0, E_z - C_z)\right\|}$$

$$\mathbf{v} = (0, 1,0)$$

$$\mathbf{u} = \mathbf{v} \times \mathbf{w}$$

给定上述局部坐标系以及公告牌的大小，公告牌四个顶点的坐标就可以由以下方式确定（如图 11.4 所示）：

```
v[0] = float4(gin[0].CenterW + halfWidth*right - halfHeight*up,
1.0f);
v[1] = float4(gin[0].CenterW + halfWidth*right + halfHeight*up,
1.0f);
v[2] = float4(gin[0].CenterW - halfWidth*right - halfHeight*up,
1.0f);
v[3] = float4(gin[0].CenterW - halfWidth*right + halfHeight*up,
1.0f);
```

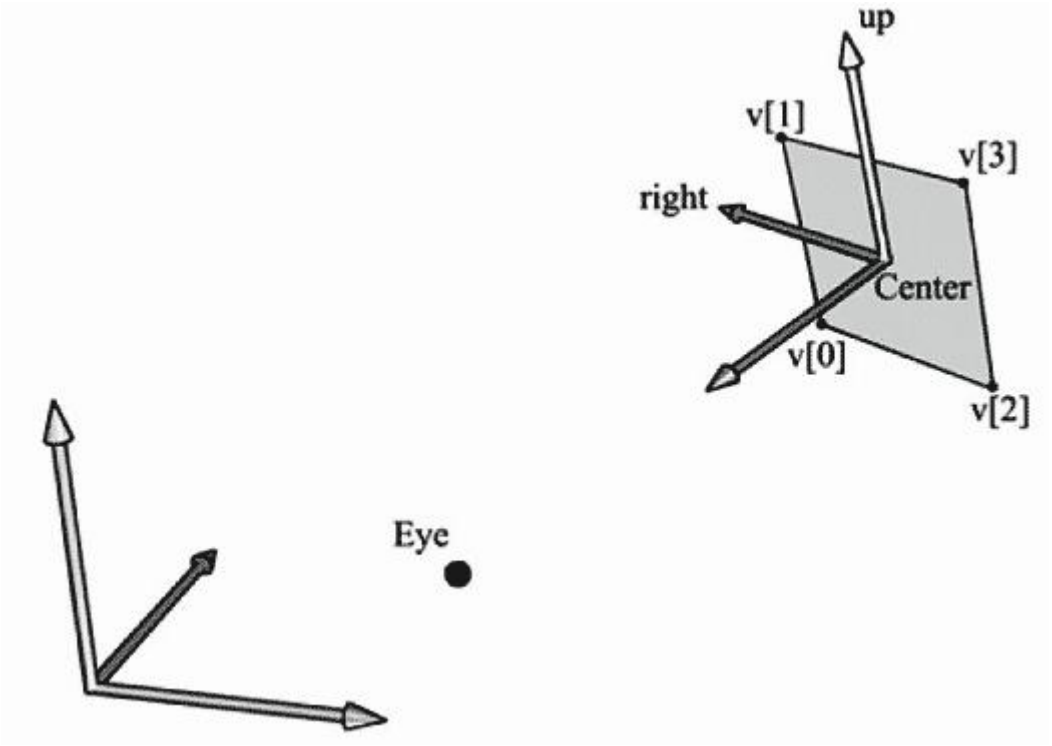



**图 11.4 根据公告牌的局部坐标系统和大小计算四个顶点的位置**

注意，每个广告牌的局部坐标系统都不一样，我们必须为每个广告牌分别计算广告牌矩阵。

在本例中，我们将创建一个点图元列表（**D3D11_PRIMITIVE_TOPOLOGY_POINTLIST**），每个点图元的位置都会比地形网格略高一些。这些点描述了我们所要绘制的广告牌的中心位置。在几何着色器中，我们将这些点扩展为广告牌四边形，并计算广告牌的世界矩阵。图 11.5 展示了该程序的屏幕截图。

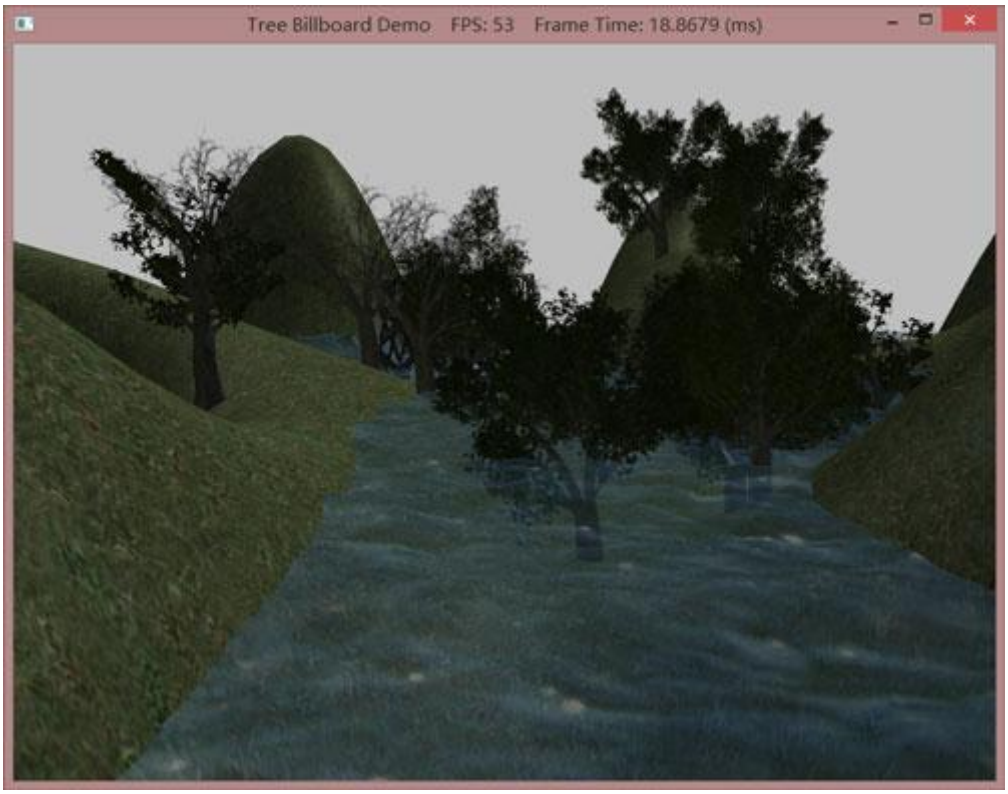



**图 11.5 树广告牌演示程序的屏幕截图。**

如图 11.5 所示，本例建立在第 9 章的“Blend”演示程序基础之上。

**注意**：公告牌的 CPU 实现版本要将公告牌的四个顶点放置在一个动态顶点缓冲中，当相机移动时，顶点坐标就需要通过 CPU 进行更新，然后使用 **ID3D11DeviceContext::Map** 方法发送到 GPU 上。这个方法必须将每个公告牌的四个顶点绑定到 IA 阶段，并更新动态缓冲区时，这是非常耗时的。而使用几何着色器的方法，GPU 会计算公告牌的四个顶点并使之朝向相机，所以只需使用静态缓冲区即可，而且，每个公告牌只需使用一个顶点，所需内存也较少。

## 11.2.2 顶点结构体

我们用下面的顶点结构体来描述广告牌的位置：

```
struct TreePointSprite
{
    XMFLOAT3 Pos;
    XMFLOAT2 Size;
} ;

const D3D11_INPUT_ELEMENT_DESC InputLayoutDesc::TreePointSprite[2]
=
{
    {"POSITION",0,DXGI
_FORMAT_R32G32B32_FLOAT,0,0,D3D11_INPUT_PER_VERTEX_DATA,0},

{"SIZE",0,DXGI_FORMAT_R32G32_FLOAT,0,12,D3D11_INPUT_PER_VERTEX_DA
```

```
TA,0}
} ;
```

该顶点结构体存储了广告牌在世界空间中的中心位置、宽度和高度（单位以世界空间为准），这样就可以使几何着色器知道广告牌应该被放在哪里，以及扩展为多大尺寸（参见图 11.6）。通过改变每个顶点的尺寸，可以很容易地让广告牌呈现出各种不同的大小。

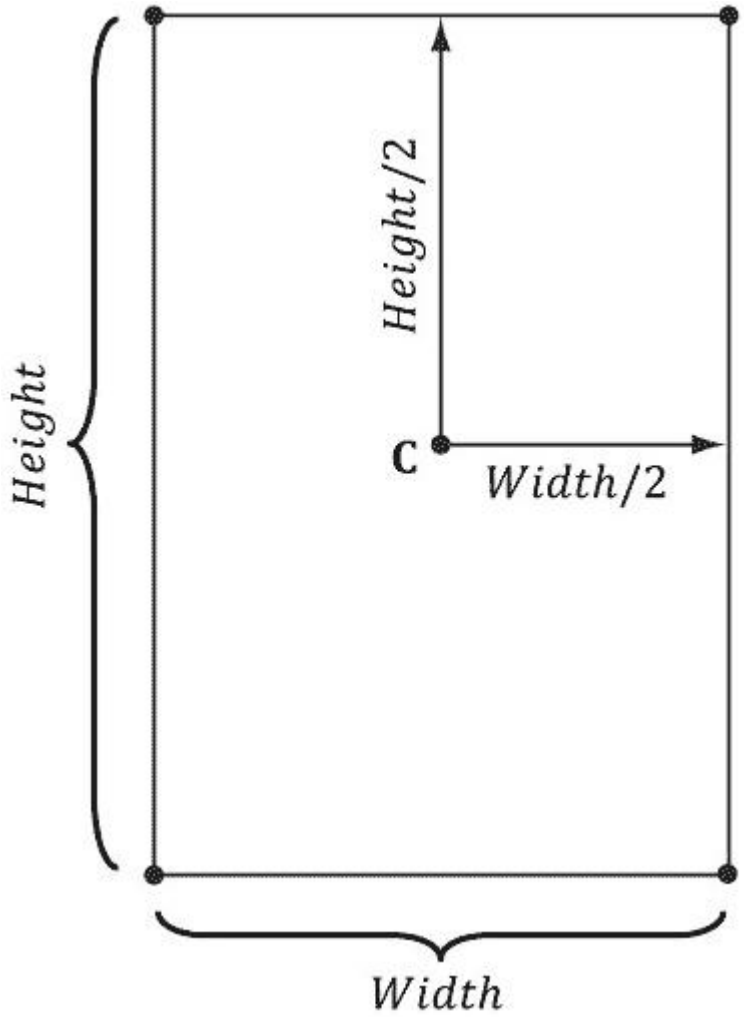



**图 11.6 将一个点扩展为一个四边形。**

除纹理数组（参见 11.3 节）外，“Tree Billboard”示例中的其他 C++代码都是普通的 Direct3D 代码（创建顶点缓冲区、effect、调用绘图方法等等）。所以，我们现在将讲解的重点转向 tree.fx 文件。

## 11.2.3 Effect 文件

由于这是我们的第一个几何着色器程序，所以我们把整个效果文件的内容都列了出来，以使你更清楚地看到顶点着色器、几何着色器、像素着色器以及其他效果对象是如何协同工作的。这个 effect 文件中还有一些我们未讨论过的对象（**SV_PrimitiveID** 和 **Texture2DArray**）；这些内容会在稍后进行讲解。现在，我们主要讲解几何着色器程序 **GS**，它按照 11.2.1 节讨论的方法，把一个点扩展为一个四边形，并将四边形的法线方向对准摄像机所在的位置。

```
//***************************************************************
```

```
************************
// TreeSprite.fx by Frank Luna (C) 2011 All Rights Reserved.
//
// Uses the geometry shader to expand a point sprite into a y-axis
aligned
// billboard that faces the camera.
//***********************************************************************
************************

#include "LightHelper.fx"

cbuffer cbPerFrame
{
    DirectionalLight gDirLights[3];
    float3 gEyePosW;

    float  gFogStart;
    float  gFogRange;
    float4 gFogColor;
};

cbuffer cbPerObject
{
    float4x4 gViewProj;
    Material gMaterial;
};

cbuffer cbFixed
{
    //
    // 计算方块上的纹理坐标
    //

    float2 gTexC[4] =
    {
        float2(0.0f, 1.0f),
        float2(0.0f, 0.0f),
        float2(1.0f, 1.0f),
        float2(1.0f, 0.0f)
    };
};

// Nonnumeric values cannot be added to a cbuffer.
Texture2DArray gTreeMapArray;
```

```
SamplerState samLinear
{
    Filter   = MIN_MAG_MIP_LINEAR;
    AddressU = CLAMP;
    AddressV = CLAMP;
};

struct VertexIn
{
    float3 PosW  : POSITION;
    float2 SizeW : SIZE;
};

struct VertexOut
{
    float3 CenterW : POSITION;
    float2 SizeW   : SIZE;
};

struct GeoOut
{
    float4 PosH    : SV_POSITION;
    float3 PosW    : POSITION;
    float3 NormalW : NORMAL;
    float2 Tex     : TEXCOORD;
    uint   PrimID  : SV_PrimitiveID;
};

VertexOut VS(VertexIn vin)
{
    VertexOut vout;

    // 直接将数据传送到几何着色器
    vout.CenterW = vin.PosW;
    vout.SizeW   = vin.SizeW;

    return vout;
}

// 因为我们将一个点扩展为一个方块（4 个顶点），
// 所以每次调用几何着色器输出顶点的最大数量为 4。
[maxvertexcount(4)]
void GS(point VertexOut gin[1],
```

```
        uint primID : SV_PrimitiveID,
        inout TriangleStream<GeoOut> triStream)
{
    //
    // 计算方块在世界空间中的局部坐标系统，
    // 公告牌与 y 轴对齐，且面对相机。
    //

    float3 up = float3(0.0f, 1.0f, 0.0f);
    float3 look = gEyePosW - gin[0].CenterW;
    look.y = 0.0f; // 公告牌与 y 轴对齐，只在 xz 平面上面对摄像机
    look = normalize(look);
    float3 right = cross(up, look);

    //
    // 在世界空间中计算顶点坐标
    //
    float halfWidth  = 0.5f*gin[0].SizeW.x;
    float halfHeight = 0.5f*gin[0].SizeW.y;

    float4 v[4];
    v[0] = float4(gin[0].CenterW + halfWidth*right - halfHeight*up,
1.0f);
    v[1] = float4(gin[0].CenterW + halfWidth*right + halfHeight*up,
1.0f);
    v[2] = float4(gin[0].CenterW - halfWidth*right - halfHeight*up,
1.0f);
    v[3] = float4(gin[0].CenterW - halfWidth*right + halfHeight*up,
1.0f);

    //
    // 将方块顶点转换到世界空间，并以三角形带的形式输出
    //
    GeoOut gout;
    [unroll]
    for(int i = 0; i < 4; ++i)
    {
        gout.PosH     = mul(v[i], gViewProj);
        gout.PosW     = v[i].xyz;
        gout.NormalW  = look;
        gout.Tex      = gTexC[i];
        gout.PrimID   = primID;

        triStream.Append(gout);
```

```
    }
}

float4 PS(GeoOut pin, uniform int gLightCount, uniform bool gUseTexure,
uniform bool gAlphaClip, uniform bool gFogEnabled) : SV_Target
{
    // Interpolating normal can unnormalize it, so normalize it.
    pin.NormalW = normalize(pin.NormalW);

    // The toEye vector is used in lighting.
    float3 toEye = gEyePosW - pin.PosW;

    // Cache the distance to the eye from this surface point.
    float distToEye = length(toEye);

    // Normalize.
    toEye /= distToEye;

    // Default to multiplicative identity.
    float4 texColor = float4(1, 1, 1, 1);
    if(gUseTexure)
    {
        // Sample texture.
        float3 uvw = float3(pin.Tex, pin.PrimID%4);
        texColor = gTreeMapArray.Sample( samLinear, uvw );

        if(gAlphaClip)
        {
            // Discard pixel if texture alpha < 0.05.  Note that we do
this
            // test as soon as possible so that we can potentially exit
the shader
            // early, thereby skipping the rest of the shader code.
            clip(texColor.a - 0.05f);
        }
    }

    //
    // Lighting.
    //

    float4 litColor = texColor;
    if( gLightCount > 0  )
    {
```

```
        // Start with a sum of zero.
        float4 ambient = float4(0.0f, 0.0f, 0.0f, 0.0f);
        float4 diffuse = float4(0.0f, 0.0f, 0.0f, 0.0f);
        float4 spec    = float4(0.0f, 0.0f, 0.0f, 0.0f);

        // Sum the light contribution from each light source.
        [unroll]
        for(int i = 0; i < gLightCount; ++i)
        {
            float4 A, D, S;
            ComputeDirectionalLight(gMaterial,        gDirLights[i],
pin.NormalW, toEye,
                A, D, S);

            ambient += A;
            diffuse += D;
            spec    += S;
        }

        // Modulate with late add.
        litColor = texColor*(ambient + diffuse) + spec;
    }

    //
    // Fogging
    //

    if( gFogEnabled )
    {
        float  fogLerp  =  saturate(  (distToEye  -  gFogStart)  /
gFogRange );

        // Blend the fog color and the lit color.
        litColor = lerp(litColor, gFogColor, fogLerp);
    }

    // Common to take alpha from diffuse material and texture.
    litColor.a = gMaterial.Diffuse.a * texColor.a;

    return litColor;
}

//-----------------------------------------------------------------------
------------------------
```

```
// Techniques--just define the ones our demo needs; you can define
the other
//   variations as needed.
//--------------------------------------------------------------------
------------------------
technique11 Light3
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_5_0, VS() ) );
        SetGeometryShader( CompileShader( gs_5_0, GS() ) );
        SetPixelShader( CompileShader( ps_5_0, PS(3, false, false,
false) ) );
    }
}

technique11 Light3TexAlphaClip
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_5_0, VS() ) );
        SetGeometryShader( CompileShader( gs_5_0, GS() ) );
        SetPixelShader( CompileShader( ps_5_0, PS(3, true, true,
false) ) );
    }
}

technique11 Light3TexAlphaClipFog
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_5_0, VS() ) );
        SetGeometryShader( CompileShader( gs_5_0, GS() ) );
        SetPixelShader( CompileShader( ps_5_0, PS(3, true, true,
true) ) );
    }
}
```

## 11.2.4 SV_PrimitiveID

在本例中，几何着色器包含了一个由 **SV_PrimitiveID** 语义修饰的特殊的无符号整数参数。

```
[maxvertexcount(4)]
```

```
void GS(point VS_OUT gIn[1],
uint primID : SV_PrimitiveID,
inout TriangleStream<GS_OUT> triStream);
```

当指定该语义时，输入汇编器阶段会为每个图元自动生成一个图元 ID。当调用 draw 方法绘制 *n* 个图元时，第 1 个图元被标记为 0，第 2 个图元被标记为 1，依次类推，直至最后一个图元被标记为 *n*−1。图元 ID 只有在每一次绘制调用中才是唯一的。在本例中，几何着色器没有使用图元 ID（虽然它可以使用这个 ID）；几何着色器把图元 ID 写入了输出顶点，传给了像素着色器阶段。像素着色器通过图元 ID 来建立广告牌和纹理数组之间的对应关系，这些内容将在下一节中讲解。

**注意**：当没有几何着色器时，图元 ID 参数可以添加到像素着色器的参数列表中：

```
float4 PS(VertexOut pin, uint primID : SV_PrimitiveID) : SV_Target
{
    // Pixel shader body...
}
```

不过，当有几何着色器时，图元 ID 参数必须定义在几何着色器的参数列表中。然后，几何着色器可以使用图元 ID 或者把图元 ID 传给像素着色器阶段（或者两者皆有）。

**注意**：输入装配器还可以生成一个顶点 ID。要使用这个 ID，必须在顶点着色器的签名中添加一个由 **SV_VertexID** 语义修饰的无符号整数参数。

下面的顶点着色器签名说明了应该如何完成这一工作：

```
VertexOut VS(VertexIn vin, uint vertID : SV_VertexID)
{
    // vertex shader body...
}
```

在每次调用 **Draw** 方法时，所要绘制的顶点都会被加上 0、1、…、*n*−1 这样的 ID 标记，其中 *n* 表示当前绘图调用中的顶点数量。在调用 **DrawIndexed** 方法时，顶点 ID 是相应的顶点索引值。

# 11.3 纹理数组

## 11.3.1 概述

纹理数组（texture array）对象用于存储一个纹理阵列。在 C++代码中，纹理数组对象由 **ID3D11Texture2D** 接口表示（该接口也用于表示单个纹理对象）。其实，在创建 **ID3D11Texture2D** 对象时有一个称为 **ArraySize** 的属性可以设置所要存储的纹理对象的数量。不过，我们总是使用 D3DX 来创建纹理，所以不必直接设置这个数据成员。在 effect 文件中，纹理数组对象由 **Texture2DArray** 类型表示：

```
Texture2DArray gTreeMapArray;
```

现在，你可能不明白我们为什么要使用纹理数组对象。为什么不这样用：

```
Texture2D TexArray[4];
...
float4 PS(GeoOut pin) : SV_Target
```

```
{
    float4 c = TexArray[pin.PrimID%4].Sample(samLinear, pin.Tex);
```

这会引发一个错误：“采样器数组索引必须是一个常量表达式”。换句话说，它不希望数组索引随着像素而变化。当我们指定一个常量数组索引时，该语句可以正常运行：

```
float4 c = TexArray[2].Sample(samLinear, pin.Tex);
```

但是与第一种方案相比，这条语句没多大用处。

## 11.3.2 对纹理数组进行采样

在树广告牌演示程序中，我们用如下代码对纹理数组进行采样：

```
float3 uvw = float3(pin.Tex, pin.PrimID%4);
float4 diffuse = gTreeMapArray.Sample(samLinear,uvw);
```

当使用纹理数组时，我们需要 3 个纹理坐标。前两个纹理坐标是普通的 2D 纹理坐标；第 3 个纹理坐标是纹理数组的索引。例如，0.0 是数组中的第 1 个纹理的索引，1.0 是数组中的第 2 个纹理的索引，2.0 是数组中的第 3 个纹理的索引，以此类推。

在树广告牌演示程序中，我们使用了一个包含 4 个元素的纹理数组，每个元素带有一种不同的树图像（参见图 11.7）。不过，我们绘制的图元数量不只 4 个，图元 ID 的值会大于 3。所以，我们让图元 ID 以 4 为模（pin.primID%4），把图元 ID 映射为 0、1、2、3，得到一个有效的数组索引。

**图 11.7 树广告牌图像。**

使用纹理数组的好处之一是，我们可以在一次绘图调用中使用不同的纹理来绘制一组图

元。以前，我们必须这样做（伪代码）：

```
SetTextureA();
DrawPrimitivesWithTextureA();
SetTextureB();
DrawPrimitivesWithTextureB();
...
SetTextureZ();
DrawPrimitivesWithTextureZ();
```

每个 Set 和 Draw 调用都会带来一些额外的系统开销。通过使用纹理数组，我们可以将其简化为一次 Set 调用和一次 Draw 调用：

```
SetTextureArray();
DrawPrimitivesWithTextureArray();
```

## 11.3.3 载入纹理数组

在撰写本书时，Direct3D 并没有提供将一批图像文件同时载入纹理数组的 D3DX 函数。所以，我们必须自己完成一任务。实现过程归纳如下：

1．分别创建每个纹理对象。

2．创建纹理数组对象。

3．将每个纹理对象依次复制到纹理数组元素中去。

4．为纹理数组对象创建一个着色器资源视图。

我们在 *d3dUtil.h/cpp* 中实现了一个辅助函数用来从一个文件名列表创建纹理数组。这些纹理的大小要求是相同的。具体的实现代码如下。

```
ID3D11ShaderResourceView* d3dHelper::CreateTexture2DArraySRV(
        ID3D11Device* device, ID3D11DeviceContext* context,
        std::vector<std::wstring>& filenames,
        DXGI_FORMAT format,
        UINT filter,
        UINT mipFilter)
{
    //
    // 从文件加载单独的纹理元素。这些纹理不能被 GPU 使用(0 bind flags)，
    // 只是用来从文件中加载图像数据。我们使用 STAGING 让 CPU 可以访问资源。
    //

    UINT size = filenames.size();

    std::vector<ID3D11Texture2D*> srcTex(size);
    for(UINT i = 0; i < size; ++i)
    {
        D3DX11_IMAGE_LOAD_INFO loadInfo;

        loadInfo.Width  = D3DX11_FROM_FILE;
```

```
        loadInfo.Height = D3DX11_FROM_FILE;
        loadInfo.Depth  = D3DX11_FROM_FILE;
        loadInfo.FirstMipLevel = 0;
        loadInfo.MipLevels = D3DX11_FROM_FILE;
        loadInfo.Usage = D3D11_USAGE_STAGING;
        loadInfo.BindFlags = 0;
        loadInfo.CpuAccessFlags    =    D3D11_CPU_ACCESS_WRITE    |
D3D11_CPU_ACCESS_READ;
        loadInfo.MiscFlags = 0;
        loadInfo.Format = format;
        loadInfo.Filter = filter;
        loadInfo.MipFilter = mipFilter;
        loadInfo.pSrcInfo  = 0;

        HR(D3DX11CreateTextureFromFile(device,
filenames[i].c_str(),
            &loadInfo, 0, (ID3D11Resource**)&srcTex[i], 0));
    }

    //
    // 创建纹理数组。每个纹理元素都具有相同的格式/大小。
    //

    D3D11_TEXTURE2D_DESC texElementDesc;
    srcTex[0]->GetDesc(&texElementDesc);

    D3D11_TEXTURE2D_DESC texArrayDesc;
    texArrayDesc.Width              = texElementDesc.Width;
    texArrayDesc.Height             = texElementDesc.Height;
    texArrayDesc.MipLevels          = texElementDesc.MipLevels;
    texArrayDesc.ArraySize          = size;
    texArrayDesc.Format             = texElementDesc.Format;
    texArrayDesc.SampleDesc.Count   = 1;
    texArrayDesc.SampleDesc.Quality = 0;
    texArrayDesc.Usage              = D3D11_USAGE_DEFAULT;
    texArrayDesc.BindFlags          = D3D11_BIND_SHADER_RESOURCE;
    texArrayDesc.CPUAccessFlags     = 0;
    texArrayDesc.MiscFlags          = 0;

    ID3D11Texture2D* texArray = 0;
    HR(device->CreateTexture2D( &texArrayDesc, 0, &texArray));

    //
    // 将单独的纹理元素复制到纹理数组中。
```

```
    //

    // 处理每个纹理元素...
    for(UINT texElement = 0; texElement < size; ++texElement)
    {
        // 处理每个渐进纹理层...
        for(UINT mipLevel = 0; mipLevel < texElementDesc.MipLevels;
++mipLevel)
        {
            D3D11_MAPPED_SUBRESOURCE mappedTex2D;
            HR(context->Map(srcTex[texElement],              mipLevel,
D3D11_MAP_READ, 0, &mappedTex2D));

            context->UpdateSubresource(texArray,
                D3D11CalcSubresource(mipLevel,              texElement,
texElementDesc.MipLevels),
                0,      mappedTex2D.pData,      mappedTex2D.RowPitch,
mappedTex2D.DepthPitch);

            context->Unmap(srcTex[texElement], mipLevel);
        }
    }

    //
    // 创建纹理数组的资源视图
    //

    D3D11_SHADER_RESOURCE_VIEW_DESC viewDesc;
    viewDesc.Format = texArrayDesc.Format;
    viewDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2DARRAY;
    viewDesc.Texture2DArray.MostDetailedMip = 0;
    viewDesc.Texture2DArray.MipLevels = texArrayDesc.MipLevels;
    viewDesc.Texture2DArray.FirstArraySlice = 0;
    viewDesc.Texture2DArray.ArraySize = size;

    ID3D11ShaderResourceView* texArraySRV = 0;
    HR(device->CreateShaderResourceView(texArray,        &viewDesc,
&texArraySRV));

    //
    // 清除--我们要的只是资源视图
    //

    ReleaseCOM(texArray);
```

```
    for(UINT i = 0; i < size; ++i)
        RereleaseCOM(srcTex[i]);

    return texArraySRV;
}
```

**ID3D11DeviceContext::UpdateSubresource** 方法通过 CPU 将一个子资源复制给另一个子资源（该方法的参数描述请参见 SDK 文档）。

```
void ID3D11DeviceContext::UpdateSubresource(
    ID3D11Resource *pDstResource,
    UINT DstSubresource,
    const D3D11_BOX *pDstBox,
    const void *pSrcData,
    UINT SrcRowPitch,
    UINT SrcDepthPitch);
```

1．**pDstResource**：目标资源对象。

2．**DstSubresource**：我们要更新的目标资源中的子资源索引（见 11.3.4 节）。

3．**pDstBox**：指向一个 **D3D11_BOX** 实例的指针，指定要更新的目标子资源的大小，若设定为 null 则更新整个子资源。

4．**pSrcData**：指向源数据的指针。

5．**SrcRowPitch**：源数据一行的大小，以字节为单位。

6．**SrcDepthPitch**：源数据一个深度的大小，以字节为单位。

注意，对于 2D 纹理而言，**SrcDepthPitch** 参数看起来是不必要的；但是这个方法还要用于更新 3D 纹理，这种情况下此参数可被视为一个 2D 纹理堆。

我们在初始化时调用前面的函数创建树图像的纹理数组：

```
ID3D11ShaderResourceView* mTreeTextureMapArraySRV;
mTreeTextureMapArraySRV                                          =
d3dHelper::CreateTexture2DArraySRV(md3dDevice,md3dImmediateContex
t, treeFilenames ,DXGI_FORMAT_R8G8B8A8_UNORM);
```

## 11.3.4 纹理子资源

现在我们已经讨论了纹理数组，下面我们来讨论子资源（subresources）的概念。图 11.8 展示了一个包含多个纹理的纹理数组。其中的每个纹理都有它自己的多级渐近纹理链。Direct3D API 使用术语“数组切片（array slice）”表示一个完整的多级渐近纹理链中的一个元素，使用术语 “多级渐近切片（mip slice）”表示纹理数组中的所有多级渐近纹理链的特定层。“子资源”表示纹理数组元素中的单个多级渐近纹理层。

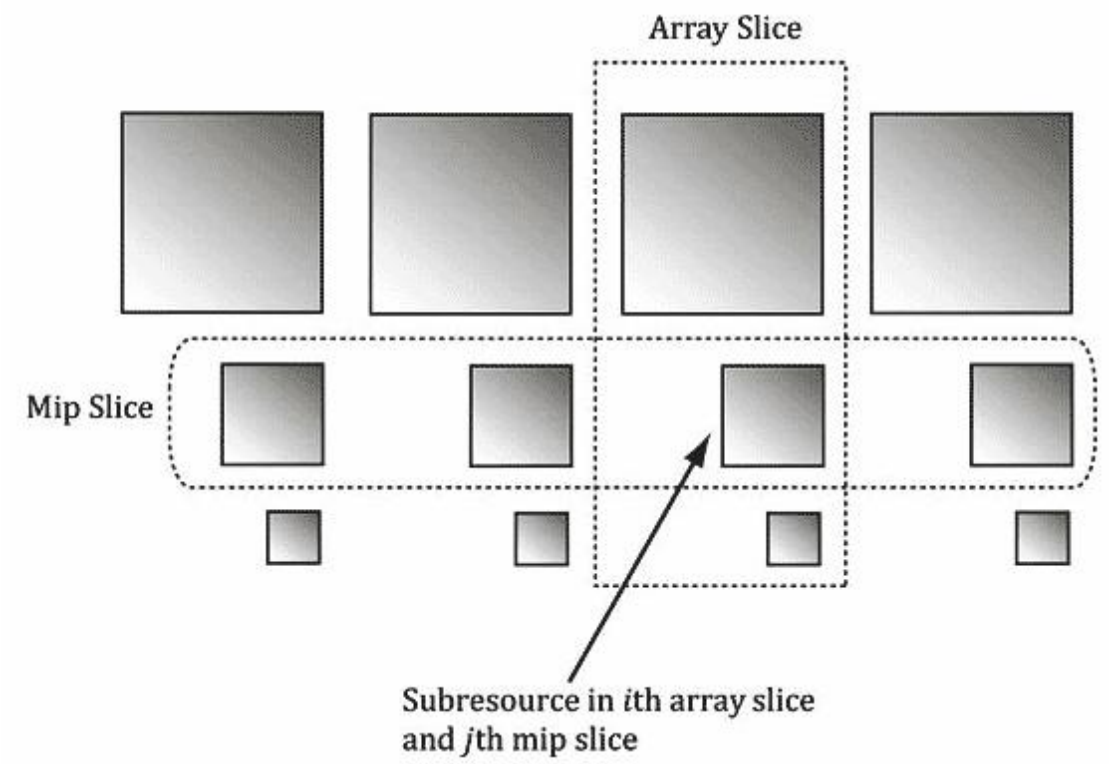



**图 11.8 包含 4 个纹理的纹理数组。每个纹理带有 3 个多级渐近纹理层。**

只要给定一个数组切片索引和一个多级渐近切片索引，我们就可以访问纹理数组中的任何一个子资源。不过，子资源也可以通过线性索引来标识；Direct3D 按照如图 11.9 所示的顺序标识线性索引。

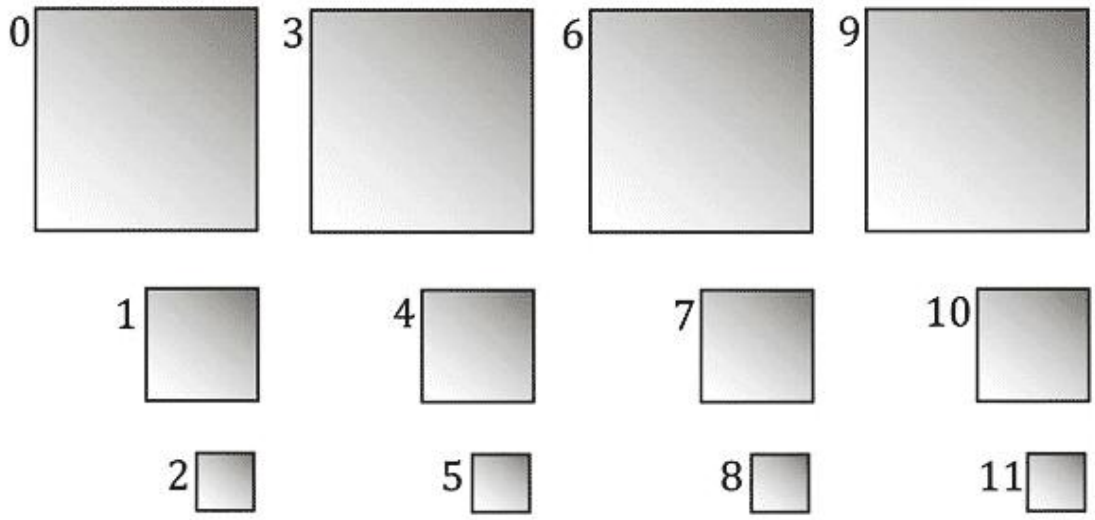



**图 11.9 通过线性索引标识纹理数组中的子资源。**

下面的工具函数用于计算线性子资源索引，它的 3 个参数分别表示多级渐近切片的索引、数组切片的索引和多级渐近纹理层的数量：

```
inline UINT D3D11CalcSubresource(
    UINT MipSlice, UINT ArraySlice, UINT MipLevels);
```

使用的公式很简单：$k = ArraySlice * MipLevels + MipSlice$。

# 16.1 屏幕到投影窗口的变换

在本章中，我们要讨论如何对用户使用鼠标拾取的 3D 物体或图元进行测定（参见图 16.1）。换言之，当给定鼠标的 2D 屏幕坐标时，我们是否能够推断出位于该投影点上的 3D 物体？从某种意义上说，我们要解决这一问题就必须做一些与之前相反的工作；也就是说，我们通常都是从 3D 空间变换到屏幕空间，而这里我们要从屏幕空间变换回 3D 空间。当然，我们还必须解决另外一个小问题：不存在一个与 2D 屏幕点唯一对应的 3D 点（即，可以有任意多个 3D 点投影在同一个 2D 点上——参见图 16.2）。所以，在测定实际拾取的物体时存在一些不确定性。不过，这不是什么大问题，因为通常与摄像机距离最近的物体就是我们实际拾取的物体。

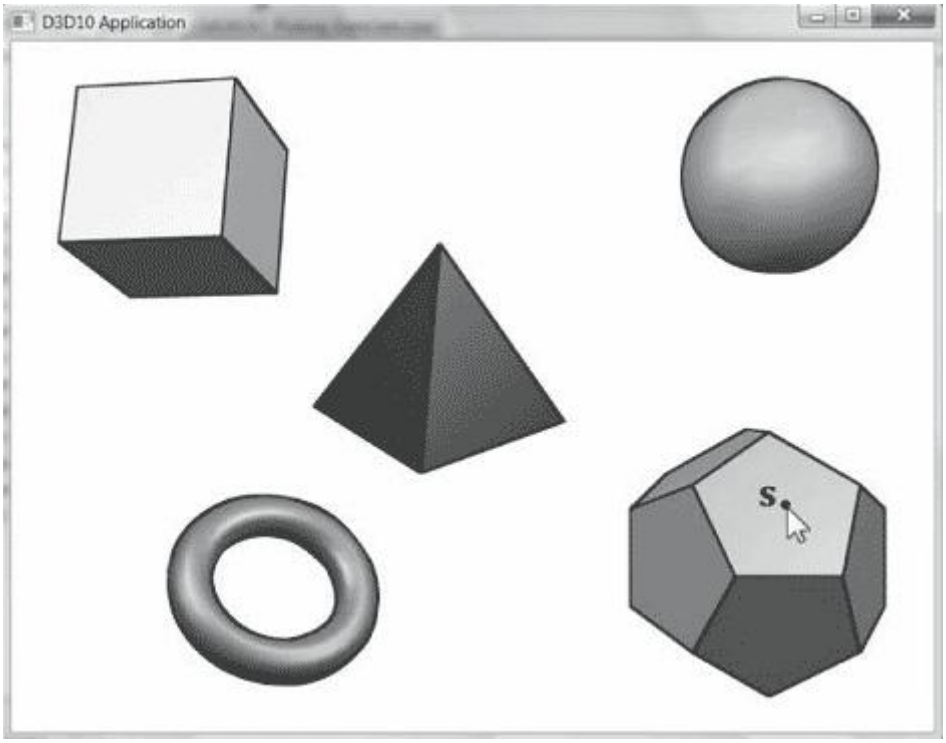



**图 16.1 用户拾取了十二面体。**

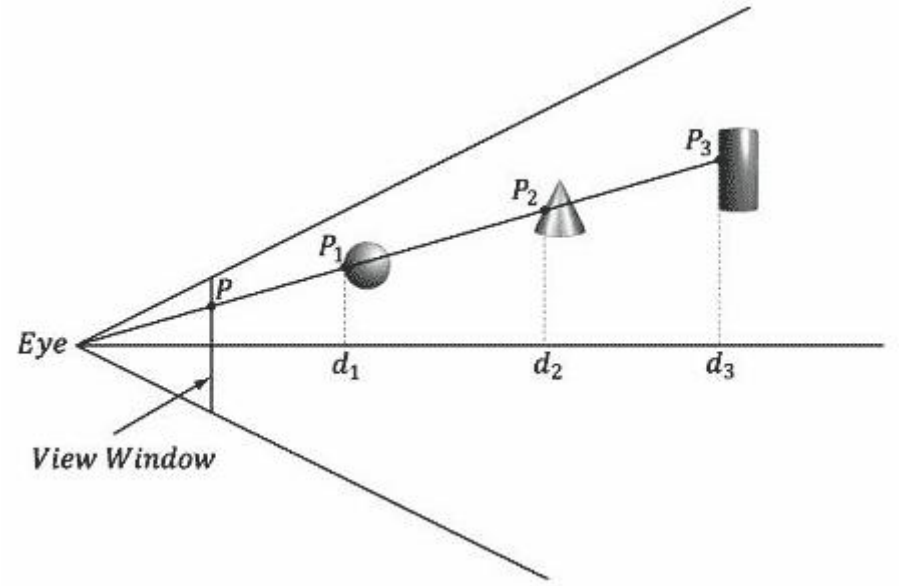



**图 16.2：平截头体的侧视图。可以看到 3D 空间中的多个点投影在了投影窗口的同一个点上。**

考虑图 16.3 所示的视域体。这里，**p** 是屏幕坐标 **s** 在投影窗口上的位置。现在，如果我们从观察点引出一条穿过点 **p** 的拾取射线，那该射线将会与所有投影到点 **p** 上的物体相交，在本例中与射线相交的是圆柱体。所以，我们的实现思路是：只要我们计算出一条拾取射线，就可以遍历场景中的每个物体，测试物体是否与该射线相交。与射线相交的物体就是被用户选中的物体。前面提到，射线可能会与场景中的多个物体相交（当然，也有可能不会与任何物体相交）。如果我们沿着射线的路径观察物体，那么就会发现它们具有不同的深度值。既然这样，我们就可以将与摄像机距离最近的相交物体作为最终的拾取物体。

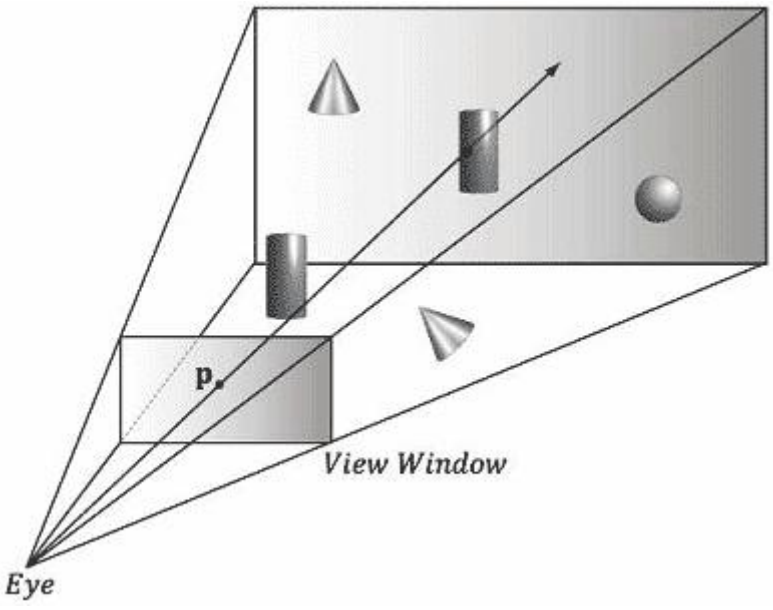



**图 16.3 穿过点 p 的射线会与投影在 p 点上的物体相交。注意，投影点 p 是屏幕坐标 s 在投影窗口上的位置。**

**学习目标**

学习如何实现拾取算法，理解拾取算法的工作原理。我们将拾取算法分解为如下 4 个步骤：

1．给定屏幕坐标 **s**，求出它在投影窗口上的对应点 **p**。

2．在观察空间中计算拾取射线。该射线从观察空间的原点射出，并穿过点 **p**。

3．把拾取射线和模型变换到同一个空间，测试模型是否与拾取射线相交。

4．确定与射线相交的物体。（与摄像机距离）最近的物体就是用户拾取的屏幕物体。

第一步是把单击的屏幕坐标变换为规范化设备坐标（参见 5.6.3.3 节）。回顾前文，视口矩阵（viewport matrix）可以把顶点从 NDC 空间变换到屏幕空间：

$$\mathbf{M}=\begin{bmatrix} \frac{Width}{2} & 0 & 0 & 0 \\ 0 & -\frac{Height}{2} & 0 & 0 \\ 0 & 0 & MaxDepth-MinDepth & 0 \\ TopLeftX+\frac{Width}{2} & TopLeftY+\frac{Height}{2} & MinDepth & 1 \end{bmatrix}$$

视口矩阵中的这些变量由 **D3D11_VIEWPORT** 结构体指定：

```
typedef struct D3D11_VIEWPORT {
    FLOAT TopLeftX;
    FLOAT TopLeftY;
    FLOA Width;
    FLOA Height;
    FLOAT MinDepth;
    FLOAT MaxDepth;
} D3D11_VIEWPORT;
```

对于游戏来说，视口通常是整个后台缓冲区，深度缓冲区的取值范围是[0,1]。所以，该结构体的成员应分别设置为：*TopLeftX* = 0、*TopLeftY* = 0、*MinDepth* = 0，*MaxDepth* = 1，*Width* = *w*，*Height* = *h*，其中 *w* 和*h*是后台缓冲区的宽度和高度。此时的视口矩阵可简化为：

$$\mathbf{M}=\begin{bmatrix} \frac{w}{2} & 0 & 0 & 0 \\ 0 & -\frac{h}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{w}{2} & \frac{h}{2} & 0 & 1 \end{bmatrix}$$

现在，设 $\mathbf{p}_{ndc}=(x_{ndc},y_{ndc},z_{ndc},1)$是 NDC 空间中的一个点（即，$-1\leqslant x_{ndc}\leqslant 1$、$-1\leqslant y_{ndc}\leqslant 1$、$0\leqslant z_{ndc}\leqslant 1$）。将 $\mathbf{p}_{ndc}$ 变换到屏幕空间后的结果为：

$$\left[x_{ndc},y_{ndc},z_{ndc},1\right]\begin{bmatrix} \frac{w}{2} & 0 & 0 & 0 \\ 0 & -\frac{h}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{w}{2} & \frac{h}{2} & 0 & 1 \end{bmatrix}=\left[\frac{x_{ndc}w+w}{2},\frac{-y_{ndc}h+h}{2},z_{ndc},1\right]$$

坐标 $z_{ndc}$ 只由深度缓冲区使用。在拾取中，我们不需要考虑任何深度坐标。2D 屏幕坐标 $\mathbf{p}_s=(x_s,y_s)$仅与 $\mathbf{p}_{ndc}$ 变换后的 $x$、$y$ 坐标有对应关系：

$$x_s=\frac{x_{ndc}w+w}{2}$$
$$y_s=\frac{-y_{ndc}h+h}{2}$$

上述方程说明，只要给定规范化设备坐标 $\mathbf{p}_{ndc}$ 和视口大小，我们就可以得到屏幕坐标 $\mathbf{p}_s$。

不过，在拾取过程中我们最初得到的是屏幕坐标 $\mathbf{p}_s$ 和视口大小，想要求出的是 $\mathbf{p}_{ndc}$。所以，由上述方程解得：

$$x_{ndc}=\frac{2x_s}{w}-1$$
$$y_{ndc}=-\frac{2y_s}{h}+1$$

我们现在有了 NDC 空间中的屏幕坐标。不过，在计算拾取射线时，我们实际想要的是观察空间中的屏幕坐标。回顾 5.6.3.3 节，我们通过将 $x$ 坐标除以横纵比 $r$，使投影点从观察空间变换到 NDC 空间：

$$-r\le x'\le r$$
$$-1\le\frac{x'}{r}\ge 1$$

那么，要回到观察空间，我们只需要将 NDC 空间中的 $x$ 坐标乘以横纵比 $r$。现在，观察空间中的屏幕坐标为：

$$x_v=r\left(\frac{2x_s}{w}-1\right)$$
$$y_v=-\frac{2y_s}{h}+1$$

**注意**：观察空间和 NDC 空间中的 $y$ 坐标相同。这是因为我们把观察空间中的投影窗口高度限定在了[−1,1] 区间内。

现在回顾 5.6.3.1 节，投影窗口与原点的距离为 $d=\cot\frac{\alpha}{2}$ ，其中$\alpha$为垂直视域角。这样我们可以引出一条穿过投影窗口上的点$(x_v,y_v,d)$的拾取射线。不过，这需要我们计算 $d=\cot\frac{\alpha}{2}$ 。图 16.4 给出了一种更简单的方法：

$$x_v'=\frac{x_v}{d}=\frac{x_v}{\cos\frac{\alpha}{2}}=x_v\tan\frac{\alpha}{2}=\left(\frac{2x_s}{w}-1\right)r\tan\frac{\alpha}{2}$$
$$y_v'=\frac{y_v}{d}=\frac{y_v}{\cos\frac{\alpha}{2}}=y_v\tan\frac{\alpha}{2}=\left(-\frac{2y_s}{h}+1\right)\tan\frac{\alpha}{2}$$

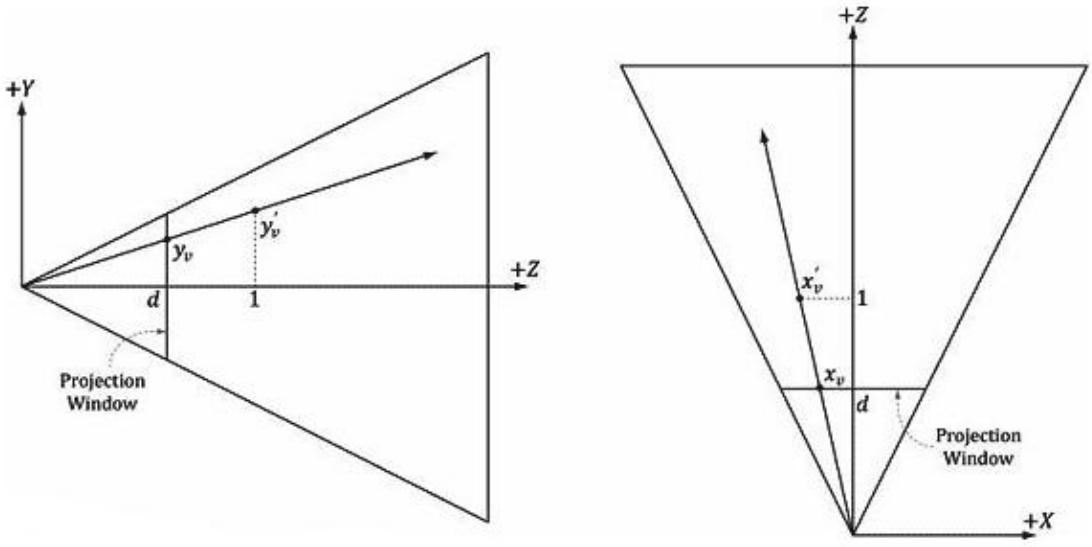



**图 16.4 由相似三角形可知 $\frac{y_v}{d}=\frac{y_v'}{1}$ 和 $\frac{x_v}{d}=\frac{x_v'}{1}$。**

回忆一下，在投影矩阵中 $\mathbf{P}_{00}=\frac{1}{r\tan\frac{\alpha}{2}}$ 和 $\mathbf{P}_{11}=\frac{1}{\tan\frac{\alpha}{2}}$。我们可以将上述方程改写为：

$$x_v'=\left(\frac{2x_s}{w}-1\right)/\mathbf{P}_{00}$$

$$y_v'=\left(-\frac{2y_s}{h}+1\right)/\mathbf{P}_{11}$$

这样，我们可以引出一条穿过点$(x_v', y_v', 1)$的拾取射线，它与穿过点$(x_v, y_v, d)$的拾取射线是同一条射线。下面给出了在观察空间中计算拾取射线的代码：

```
void PickingApp::pick(int sx, int sy)
{
    XMMATRIX P = mCam.Proj();

    // 在视空间中计算拾取射线
    float vx = (+2.0f*sx/mClientWidth - 1.0f)/P(0,0);
    float vy = (-2.0f*sy/mClientHeight + 1.0f)/P(1,1);
    // 视空间中的射线定义
    XMVECTOR rayOrigin = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);
    XMVECTOR rayDir = XMVectorSet (vx, vy, 1.0f, 0.0f);
```

注意，该射线的起点是观察空间的原点，因为观察点位于观察空间的原点上。

# 16.2 世界/本地空间中的拾取射线

现在，我们已经知道了如何计算观察空间中的拾取射线，但是它的用途非常有限，因为只有当物体也在观察空间中时，我们才能使用该射线进行相交测试。我们知道，观察矩阵可

以将几何体从世界空间变换到观察空间，所以它的逆矩阵可以将几何体从观察空间变换回世界空间。设 $\mathbf{r}_v(t) = \mathbf{q} + t\mathbf{u}$ 为观察空间中的拾取射线，**V** 为观察矩阵，则世界空间中的拾取射线为：

$$\mathbf{r}_w(t) = \mathbf{q}\mathbf{V}^{-1} + t\mathbf{u}\mathbf{V}^{-1} = \mathbf{q}_w + t\mathbf{u}_w$$

注意，射线起点 **q** 是按照“点的方式”来变换的（即，$\mathbf{q}_w = 1$），而射线方向 **u** 是按照“向量的方式”来变换的（即，$\mathbf{u}_w = 0$）。

世界空间中的拾取射线可以与世界空间中的物体进行相交测试。不过，在大多数情况下，一个物体中的几何体都是相对于该物体自身的局部空间来定义的。所以，我们必须把射线变换到物体的局部空间后再进行射线与物体之间的相交测试。设 **W** 为物体的世界矩阵，矩阵 $\mathbf{W}^{-1}$ 可以将几何体从世界空间变换到物体的局部空间。则局部空间中的拾取射线为：

$$\mathbf{r}_L(t) = \mathbf{q}_W\mathbf{W}^{-1} + t\mathbf{u}_W\mathbf{W}^{-1}$$

通常，场景中的每个物体都有它自身的局部空间。 所以，必须把射线变换到每个物体的局部空间后再进行相交测试。

有些读者可能会想：是否可以把网格变换到世界空间，然后在世界空间中进行相交测试呢？方法可行，但不可取。因为这样的计算量会非常大。一个网格可能会包含几千个顶点，如果把些顶点逐个变换到世界空间，那代价会非常大。在效率上，这种方式远不如将一条射线变换到物体的局部空间更为高效。

下面的代码示范了如何将一条拾取射线从观察空间变换到一个物体的局部空间：

```
//  将设置转换到网格的本地空间
XMMATRIXV = mCam.View();
XMMATRIXinvView = XMMatrixInverse(&XMMatrixDeterminant(V),V);
XMMATRIXW = XMLoadFloat4x4(&mMeshWorld);

XMMATRIXinvWorld = XMMatrixInverse(&XMMatrixDeterminant(W),W);
XMMATRIXtoLocal = XMMatrixMultiply(invView,invWorld);

rayOrigin = XMVector3TransformCoord(rayOrigin,toLocal);
rayDir = XMVector3TransformNormal(rayDir,toLocal);

// 规范化射线方向用于相交测试
rayDir = XMVector3Normalize(rayDir);
```

**XMVec3TransformCoord** 和 **XMVec3TransformNormal** 函数都是以 3D 向量作为参数。但是请注意，**XMVec3TransformCoord** 函数将向量的第 4 个分量视为 1；而 **XMVec3TransformNormal** 函数将向量的第 4 个分量视为 0。所以，我们使用 **XMVec3TransformCoord** 函数变换点；使用 **XMVec3TransformNormal** 函数变换向量。

# 17.1 立方体贴图映射

在本章中，我们学习立方体贴图（cube map），它本质上是一种由 6 幅纹理组成的、按特殊方式解释的纹理数组。通过使用立方体贴图映射，我们可以很容易地对一个天空进行纹理映射或模拟反射。

**学习目标**

1．了解什么是立方体贴图以及如何在 HLSL 代码中对它们进行采样。

2．学习如何使用 DirectX 纹理工具创建立方体贴图。

3．学习如何使用立方体贴图来模拟反射。

4．学习如何使用立方体贴图来对一个天空穹顶进行纹理映射。

立方体贴图映射的实现思路是：将 6 幅纹理想像为关于某个坐标系原点和轴对齐的立方体的 6 个平面（“立方体贴图”这个名字正是由此而来）。由于立方体纹理是轴对齐的，它的每个平面都沿着 3 个主轴的方向放置；所以，我们可以根据与平面相交的主轴方向（±X，±Y，±Z）标识立方体贴图的每个平面。Direct3D 提供了 D3D11_TEXTURECUBE_FACE 枚举类型来完成这一工作：

```
typedef enum D3D11_TEXTURECUBE_FACE
{
    D3D11_TEXTURECUBE_FACE_POSITIVE_X = 0,
    D3D11_TEXTURECUBE_FACE_NEGATIVE_X = 1,
    D3D11_TEXTURECUBE_FACE_POSITIVE_Y = 2,
    D3D11_TEXTURECUBE_FACE_NEGATIVE_Y = 3,
    D3D11_TEXTURECUBE_FACE_POSITIVE_Z = 4,
    D3D11_TEXTURECUBE_FACE_NEGATIVE_Z = 5,
} D3D11_TEXTURECUBE_FACE;
```

立方体贴图存储在一个纹理数组中，它包含 6 幅纹理：

**1．**索引 0 表示+X 平面。

**2．**索引 1 表示－X 平面。

**3．**索引 2 表示+Y 平面。

**4．**索引 3 表示－Y 平面。

**5．**索引 4 表示+Z 表面。

**6．**索引 5 表示－Z 表面。

与 2D 纹理映射不同，我们在这里不能使用 2D 纹理坐标指定纹理元素。要指定一个立方体贴图中的纹理元素，我们必须定义一个从原点引出的查找向量 **v**，通过 3D 纹理坐标指定纹理元素。查找向量 v 与立方体贴图相交的地方（参见图 11.1）就是 3D 纹理坐标 **v** 对应的纹理元素。我们在第 8 章讨论的纹理过滤概念同样适用于立方体贴图采样。

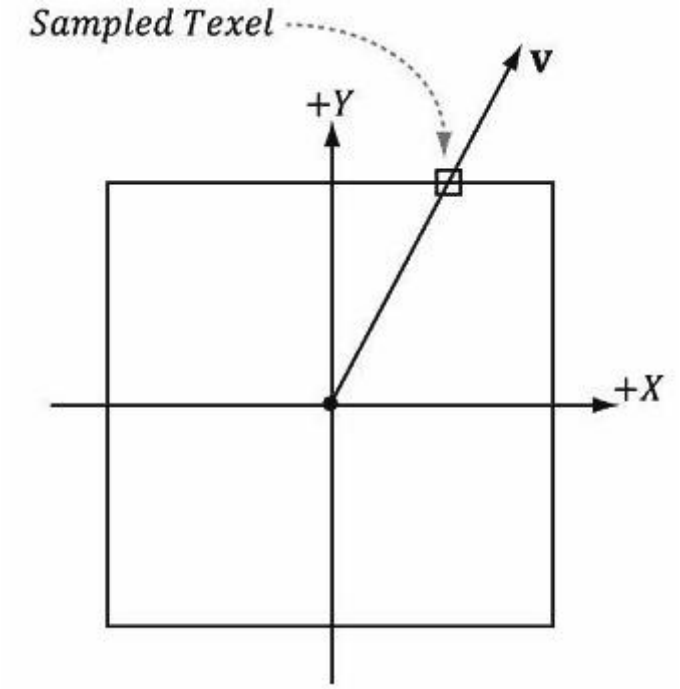



**图 17.1 为简单起见，我们在 2D 空间中描述一概念；读者可以把个正方形想像为一个 3D 空间中的立方体。该正方形表示一个与坐标系原点和轴对齐的立方体贴图。我们从原点引出一个向量 v。与 v 相交的纹理元素就是所要采样的纹理元素。在本图中，v 与立方体贴图的+Y 平面相交。**

**注意**：查找向量的大小并不重要；重要的只是它的方向。使用方向相同、大小不同的两个向量对立方体贴图进行采样得到的结果完全相同。

在 HLSL 中，立方体纹理由 **TextureCube** 类型表示。下面的代码片段说明了应该如何对立方体贴图进行采样：

```
TextureCube gCubeMap;
SamplerState gTriLinearSam
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
};
...
// 在像素着色器中
float3 v = float3(x,y,z); // 用于查找纹理坐标的向量
float4 color = gCubeMap.Sample(gTriLinearSam,v);
```

**注意**：查找向量和立方体贴图应该使用相同的坐标系。例如，当立方体贴图使用世界坐标系时（即，立方体平面与世界空间的主轴对齐），查找向量也应该使用世界坐标系。

# 17.2 环境贴图

立方体贴图的主要用途是实现环境贴图映射（environment mapping）。它的实现思路是：在场景中的某个物体 O 的中心位置放置一架摄像机，将（水平和垂直）视域角设为 90º。然

后沿着±X 轴、±Y 轴和±X 轴方向，从 6 种不同的角度各拍摄一张照片（在照片中不包含物体 O）。因为视域角为 90º，所以这 6 张照片完全可以从物体 O 的角度捕捉到各个方向上的环境信息（参见图 17.2）。我们把这 6 张照片存入到一个立方体贴图中，就得到了所谓的环境贴图。换句话说，环境贴图就是在立方体平面上存入一个环境的全景照片。

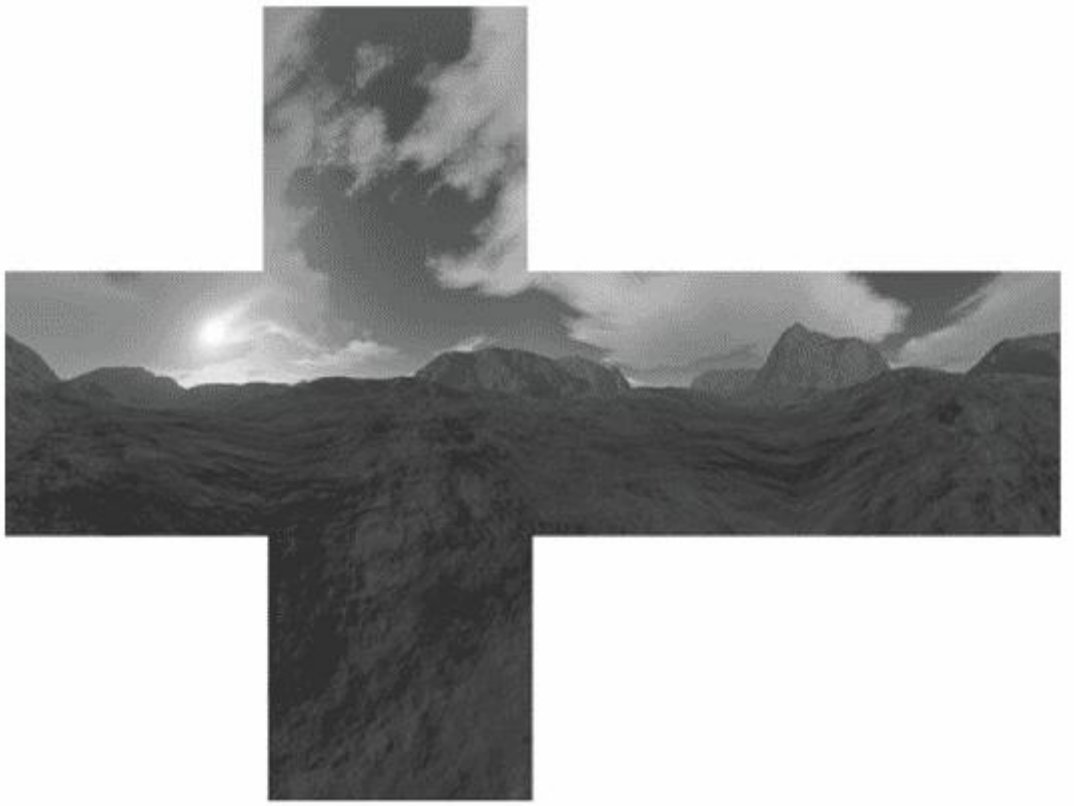


**图 17.2 将立方体贴图“展平”后就得到了一幅环境贴图。设想，将这 6 个平面重新折叠为一个 3D 立方体，然后站在立方体的中心。从每个方向上，你都可以看到一个连续的场景环境。**

上述内容表明，在场景中有多少个使用环境贴图映射的物体，我们就必须创建多少个环境贴图。不过，环境贴图通常只用于表现远处的“背景”信息，而近景物体可以共享相同的环境贴图。例如，在图 17.3 中，所有的球体都共享图 17.2 所示的环境贴图。注意，这个环境贴图并没有拍摄场景中的石柱和地板；它只拍摄了远处的山峰和天空（即，场景背景）。虽然在某些场景中，背景环境贴图不能表现场景的所有细节，但是在实践中它的渲染结果还是比较令人满意的。


**图 17.3 立方体贴图演示程序的屏幕截图。**

当创建环境贴图时，摄像机的拍摄方向必须与世界空间的主轴方向对齐，只有这样，环境贴图才能直接在世界空间中使用。当然，你也可以采用其他的方向进行拍摄（比方说某个物体的局部空间）。但是此时，查找向量必须被转换到立方体贴图空间之后，才能进行纹理映射。

注意，环境贴图使用的这 6 幅图像一般不是在 Direct3D 程序中生成的，虽然有时也可以这么做（参见 17.5 节）。因为立方体贴图只存储纹理数据，它们的内容通常是由美术师提前画好的（就像我们之前使用的 2D 纹理一样）。所以，我们不需要使用实时渲染计算一个立方体贴图的图像。也就是，我们可以在一个 3D 世界编辑器中创建场景，然后在编辑器中预渲染立方体贴图平面上的这 6 幅图像。对于室外环境贴图来说，我们通常使用 Terragen 软件（http://www.planetside.co.uk/terragen/，该软件有免费的个人使用版本），它可以创建照片级的室外场景。本书使用的环境贴图都是使用 Terragen 软件生成的，比如图 17.2 所示的环境贴图。

**注意**：当你初次使用 Terragen 软件时，一定要在 **Camera Settings** 对话框中将缩放系数（**zoom factor**）设为 1.0，以得到一个 90º 的视域角。另外，输出图像的宽度和高度也必须设为相同的值，只有这样摄像机在水平和垂直方向上的视域角才会相同，也就是 90º 。

**注意**：网上（https://developer.valvesoftware.com/wiki/Skybox_(2D)_with_Terragen）还有一个很好的 Terragen 脚本，它可以使用当前相机的位置并以 90º 的视域角渲染出周围的六张图像。

当我们使用某种软件创建了立方体贴图的 6 幅图像之后，我们必须创建一幅立方体贴图纹理来存储这 6 幅图像。我们之前使用的 DDS 纹理图像格式就支持立方体贴图，我们可以很容易地使用 DirectX 纹理工具将这 6 幅图像合并为一个立方体贴图。运行 DirectX 纹理工具（它位于 DirectX SDK 的 DXSDXJune10\Utilities\Bin\x86 目录下）。在 **File** 菜单中选择 **New Texture** 命令，弹出如图 17.4 所示的对话框。在 Texture Type 中选择 **Cubemap Texture** 选项，输入源图像的尺寸大小，并选择一种表面格式。（最好使用像 DXT1 这样的压缩格式；因为

这里要存储 6 幅纹理，高分辨率的立方体贴图会占用很多内存。）

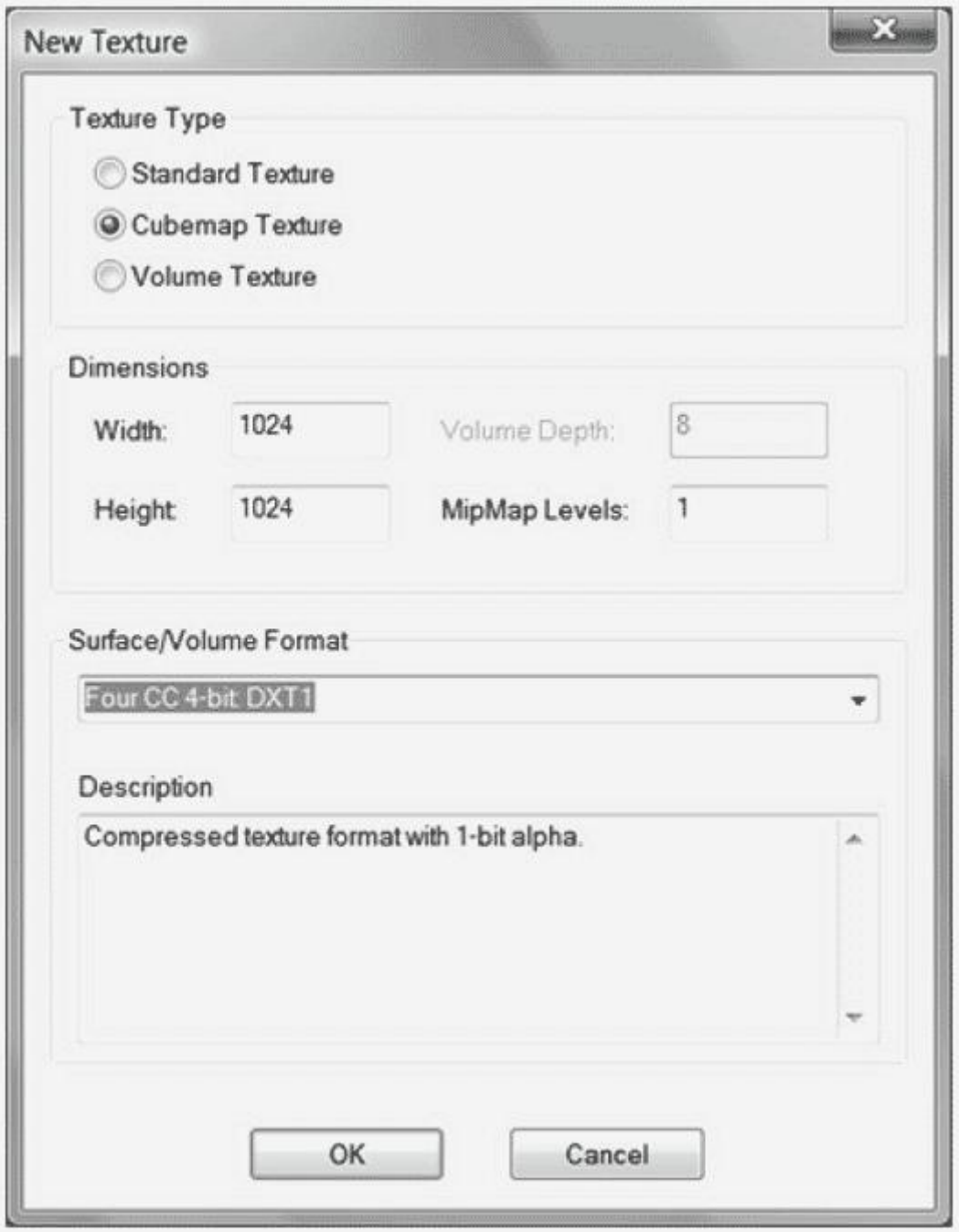



**图 17.4 使用 DirectX 纹理工具创建一幅新的立方体纹理。**

现在我们有了一幅空的立方体贴图。将鼠标定位在 **View** 菜单上，选择 **Cube Map Face** 子菜单，挑选一个你希望在窗口中显示的立方体贴图平面（图 17.5）。（起初，这些平面都是空的。）现在随便选择一个平面，然后单击 **File** 菜单，选择 **Open Onto This Cubemap Face** 命令，在弹出的对话框中选择对应于个立方体平面的图像文件，将它导入到当前的立方体贴图平面中。重复这一操作，向其余的 5 个立方体贴图平面导入相应的图像文件。当全部操作完成之后，将立方体贴图保存为 DDS 文件。


**图 17.5 在 DirectX 纹理工具中，挑选一个希望在窗口中显示的立方体贴图平面。**

**注意**：NVIDIA 提供了一个 Photoshop 插件用来保存.DDS 和立方体贴图，可见 http://developer.nvidia.com/nvidia-texture-tools-adobe-photoshop。

## 11.2.1 在 Direct3D 中载入和使用立方体贴图

使用 **D3DX11CreateShaderResourceViewFromFile** 函数可以很方便地将一个存储了立方体贴图的 DDS 文件载入到 **ID3D11Texture2D** 对象中并生成对应的 shader 视图：

```
ID3D11ShaderResourceView * mCubeMapSRV;
HR(D3DX11CreateShaderResourceViewFromFile(device,cubemapFilename.
c_str(),0,0,&mCubeMapSRV,0));
```

**ID3D11Texture2D** 对象会将立方体贴图的 6 个纹理存储为一个纹理数组。

在我们获得了一个指向立方体纹理的 **ID3D11ShaderResourceView***指针之后，我们可以使用 **ID3D11EffectShaderResourceVariable::SetResource** 方法将它指定给 effet 文件中的 **TextureCube** 变量：

```
// .fx 变量
TextureCube gCubeMap;

// .cpp 代码
ID3D11EffectShaderResourceVariable* CubeMap;
CubeMap =
mfx -> GetVariableByName("gCubeMap")->AsShaderResource();
...
CubeMap->SetResource(cubemap);
```

# 17.3 对天空进行纹理映射

在本节中， 我们使用一个环境贴图对天空进行纹理映射。 我们创建了一个包围整个场景的椭圆体（用椭圆体来模拟一个较为平坦的天空表面）。我们通过图 17.6 所示的方法，使用一个环境贴图对椭圆体进行纹理映射，将环境贴图投影到椭圆体表面上，模拟天地相接处的远山假象。

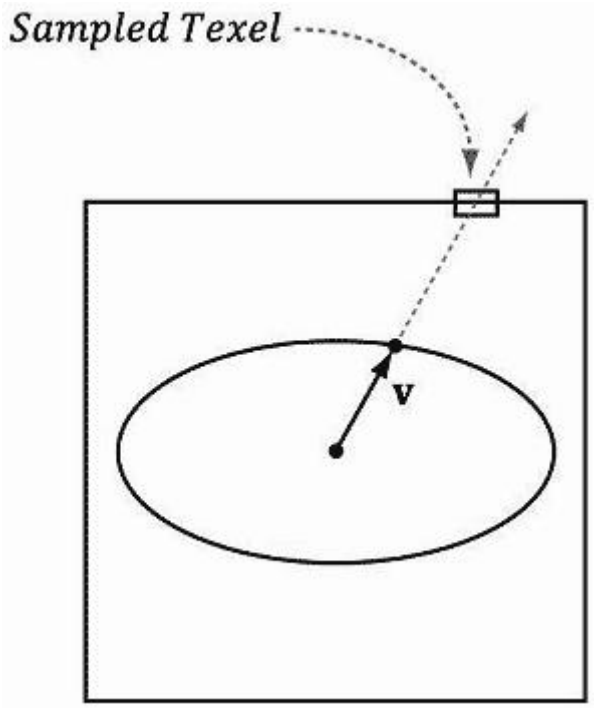



**图 17.6：为简单起见，我们在 2D 空间中描述一概念；读者可以把个正方形想像为一个 3D 空间中的立方体，把个椭圆形想像为一个 3D 空间中的椭圆体。我们假设天空和环境贴图的中心均与坐标系原点对齐。然后，我们对椭圆体表面上的点进行纹理映射，使用从原点指向表面点的向量作为立方体贴图的查找向量，将立方体贴图投影到椭圆体上。**

我们假设天空椭圆体无限遥远（即，它的中心与世界空间的原点对齐，但是有无限大的半径），无论摄像机如何在世界空间中移动，我们永远都无法靠近或远离天空椭圆体表面。要实现这个无限遥远的天空，我们只需要在世界空间中将天空椭圆体的中心和摄像机的位置对齐，让它跟着摄像机移动。这样无论摄像机移动到哪，我们都不会靠近天空椭圆体表面。如果不这样做，那么当摄像机与天空椭圆体表面越来越近时，整个假象就会被拆穿，我们用来模拟天空的这个小把戏就没什么意思了。

用来实现天空的 effect 文件如下：

```
//=============================================================
==============
// Sky.fx by Frank Luna (C) 2011 All Rights Reserved.
//
// Effect used to shade sky dome.
//=============================================================
==============
```

```
cbuffer cbPerFrame
{
    float4x4 gWorldViewProj;
};

// Nonnumeric values cannot be added to a cbuffer.
TextureCube gCubeMap;

SamplerState samTriLinearSam
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
};

struct VertexIn
{
    float3 PosL : POSITION;
};

struct VertexOut
{
    float4 PosH : SV_POSITION;
    float3 PosL : POSITION;
};

VertexOut VS(VertexIn vin)
{
    VertexOut vout;

    // 将 z 设置为 w，这样 z/w = 1 (即天空球总是在位于平面).
    vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj).xyww;

    // 使用局部顶点位置作为立方体贴图的查找向量
    vout.PosL = vin.PosL;

    return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
    return gCubeMap.Sample(samTriLinearSam, pin.PosL);
}
```

```
RasterizerState NoCull
{
    CullMode = None;
};

DepthStencilState LessEqualDSS
{
    // 将深度测试函数设置为 LESS_EQUAL 而不是 LESS.
    // 否则，当深度缓冲清除为 1 时，z=1(NDC)处规范化深度值将通不过深度测试。
    DepthFunc = LESS_EQUAL;
};

technique11 SkyTech
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_5_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_5_0, PS() ) );

        SetRasterizerState(NoCull);
        SetDepthStencilState(LessEqualDSS, 0);
    }
}
```

**注意**：在以前，应用程序会先绘制天空，把它作为一个替代物来清空渲染目标和深度/模板缓冲区。不过，现在出于以下原因《ATI Radeon HD 2000 编程指南》（http://ati.amd.com/developer/SDK/AMD_SDK_Samples_May2007/Documentations/ATI_Radeon_HD_2000_programming_guide.pdf）建议我们不要再使用这种方式。首先，深度/模板缓冲区应该被直接清空，这样可以获得硬件内部的深度优化，有利于提高性能。这一点与渲染目标的情形相似。其次，天空通常会被其他物体阻挡，比如建筑物和地形。当我们先绘制天空时，天空图像会被与摄像机距离更近的其他物体覆盖，是对资源的浪费。所以，现在的首选方式是始终调用 Clear***方法清空各个缓冲区，最后绘制天空。

# 17.4 模拟反射

如前所述，环境贴图在实现天空纹理映射时可以达到比较好的效果。环境贴图的另一个主要用途是为任意物体模拟反射（该技术只能反射环境贴图中的图像）。图 17.7 说明了如何使用环境贴图来实现镜面反射。这个表面就像是一面镜子：观察点 e 可以看到由 p 点反射回来的环境图像。

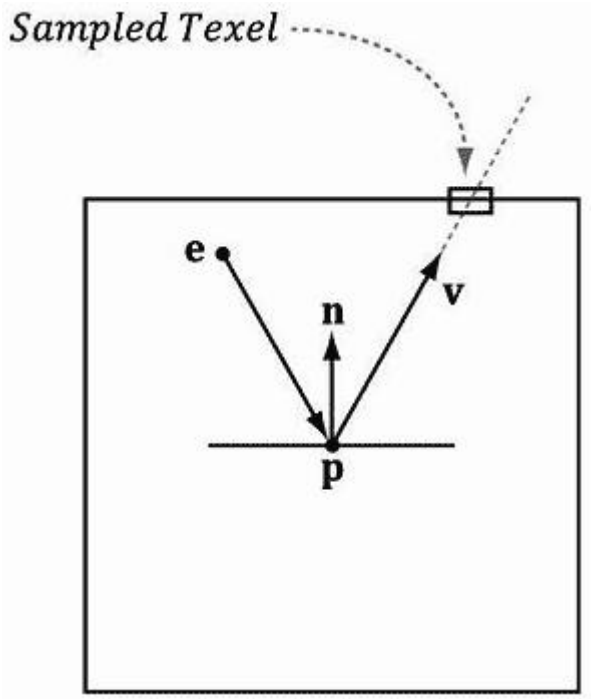



图 17.7 这里 **e** 是观察点，**n** 是点 **p** 的表面法线，**v** 是 **ep** 向量的反射向量。我们可以通过反射向量 **v** 将纹理元素映射到表面点 **p** 上（即，我们可以把 v 作为查找向量）。通过一方式，观察点就可以看到反射后的环境图像了。

我们为每个像素计算反射向量，然后用它来对环境贴图进行采样：

```
litColor = texColor*(ambient + diffuse) + spec;
if(gReflectionEnabled)
{
    float3 incident = -toEye;
    float3 reflectionVector = reflect(incident, pin.NormalW);
    float4  reflectionColor  =  gCubeMap.Sample(samAnisotropic,refle
ctionVector);
    litColor += gMaterial.Reflect*reflectionColor;
}
```

通常，像素的颜色不完全取决于反射颜色（除非镜子的反射率是 100%）。所以，我们必须修改一下光照方程，加入一个反射项 $\mathbf{m}_R \otimes \mathbf{c}_R$。这里 $\mathbf{c}_R$ 是环境贴图的采样颜色，$\mathbf{m}_R$ 是应用程序指定的材质颜色，它决定了表面对 $\mathbf{c}_R$ 的反射数量。例如，当表面只反射红光时，你应该将 $\mathbf{m}_R$ 设为(1,0,0)，使表面只从环境贴图中反射红光。我们之前定义的 **Material** 结构已经包含了一个反射属性，本章终于可以派上用场了：

```
struct Material
{
    float4 Ambient;
    float4 Diffuse;
    float4 Specular;
    float4 Reflect;
};
```

现在存在的一个问题是在光照方程引入附加的反射项后，物体颜色显得过于饱和。由于反射项的添加，每个像素颜色值都会增大，物体表面的亮度会显得过高。本质上，如果我们

要将一个额外的反射颜色添加进来，那么就必须从其他项中减去一些颜色，以求得平衡。一般可以通过降低环境材质系数和漫反射材质系数来实现，使表面反射的环境光和漫反射光更少一些。另一种方式是计算环境贴图的采样颜色 $\mathbf{c}_R$ 和普通光照颜色 **s** 之间的加权平均值：

$$\mathbf{f} = t\mathbf{c}_R + (1 - t)\mathbf{s} \qquad \text{其中 } 0 \leqslant t \leqslant 1$$

通过一方式，我们给环境贴图的采样颜色添加了一个权值 $t$，从普通光照颜色中减去了等量的颜色，以保持平衡。这里的参数 $t$ 可用来控制表面的反射率。

图 17.8 说明通过环境贴图映射无法达到令人满意的平面反射效果。这是因为反射向量无法表达完整的信息（它不包含坐标位置）；我们实际需要的是一条反射射线，并且要让该射线与环境贴图相交。射线具有位置和方向，而向量只具有方向。我们可以从图中看到，两条反射射线 $\mathbf{r}(t) = \mathbf{p} + t\mathbf{v}$ 和 $\mathbf{r}'(t) = \mathbf{p}' + t\mathbf{v}$ 与不同的立方体贴图元素相交，所以应该得到不同的颜色。不过，由于两条射线具有相同的方向向量 **v**，而方向向量 **v** 又被唯一地用于查找立方体贴图，所以当观察点位于 **e** 和 **e**′时，**p** 和 **p**′会映射到同一个纹理元素上。对于平面物体来说，这是环境贴图映射的一个严重缺陷。对于曲面来说，环境贴图映射的这一缺陷并不明显，因为曲面会产生反射向量的变化。[Brennan02]在该问题上给出了一个基本可行的解决方案。

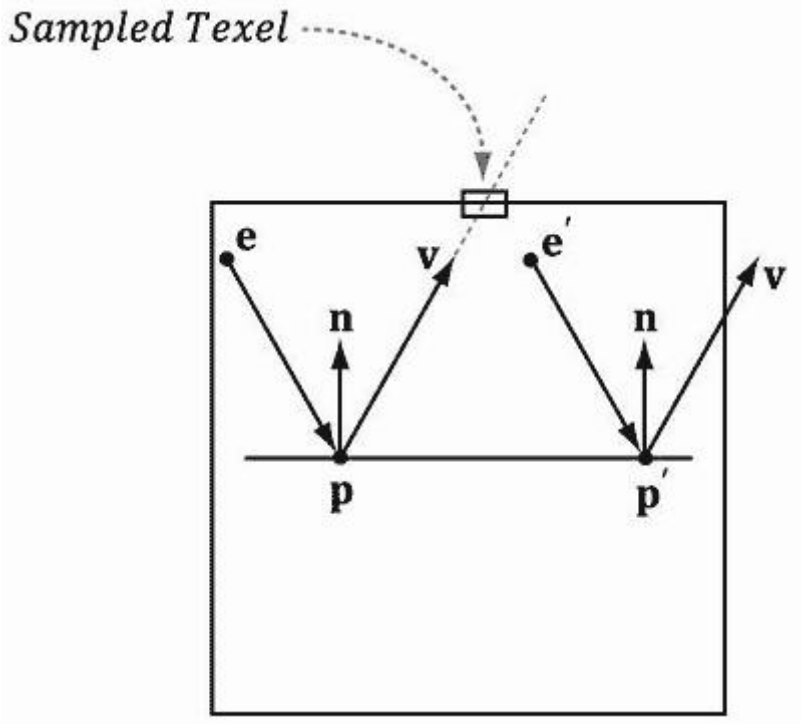



**图 17.8 当视点位于 e 和 e′时，反射向量 v 分别对应于两个不同点 p 和 p′。**

# 18.1 为什么要使用法线贴图映射

我们在第 8 章中介绍了纹理贴图映射，它可以将纹理细节映射到三角形表面上。然而，我们现在的法线向量仍然定义在比较粗糙的顶点级别上，使用的是三角形表面插值。在本章中，我们将介绍一种目前流行的方法，在更高分辨率下指定表面法线，这样可以增加光照的细节，但是网格的几何细节仍然没有得到提高。位移映射与曲面细分的组合，可以让我们增加网格的细节。

**学习目标**

1．了解为什么要使用法线贴图映射。

2．了解法线贴图的存储方式。

3．学习如何创建法线贴图。

4．了解法线向量在法线贴图中的存储方式，以及如何在 3D 三角形的物体空间中使用法线向量。

5．学习如何在顶点着色器和像素着色器中实现法线贴图映射。

6．学习组合位移映射和曲面细分改进网格的细节。

如图 18.1 所示，圆柱体的高光看上去有些问题——光滑的柱体表面与凹凸不平的砖块纹理放在一起显得极不谐调。这是因为网格几何体是平滑的，我们只是在平滑的柱体表面上使用了一个凹凸不平的砖块纹理而已。而光照计算是根据网格几何体（尤其是顶点法线插值）来实现的，没有考虑到纹理图像的内容。所以，光照和纹理表现出来的质感不完全一致。

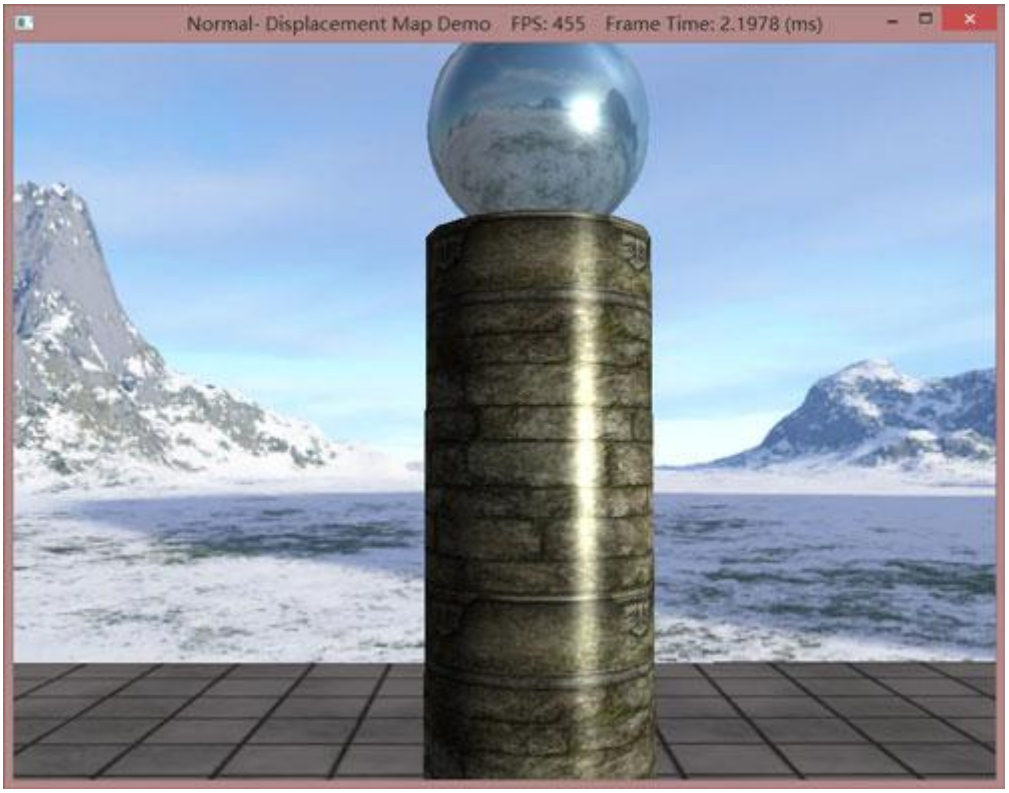



**图 18.1 平滑的镜面高光。**

在理想状态下，我们可以对网格几何体进行细化，通过网格来模拟砖块实际具有的凹凸裂痕，使光照和纹理表现出一致的质感。硬件曲面细分可以实现这一目的，但是，我们仍需要一种指定由细分器产生的顶点法线（通过插值得到的法线并不能增加法线的分辨率）的方法。

另一种可能的解决办法是将光照细节直接烘焙到纹理上。不过，当灯光在场景中移动时，这种方法会产生错误的光照结果。因为固定在纹理元素中的光照颜色不会随着灯光的移动而变化。

所以，我们的目标是要寻求一种实现动态光照的方法，使物体表面即能体现自身的纹理细节，又能表现正确的光照质感。由于所有的细节都来源于纹理，所以我们可以转变一下思路，从纹理贴图映射入手寻求问题的答案。图 18.2 是同一个场景在使用法线贴图映射时效果；我们可以看到，动态光照与砖块纹理表现出了相当一致的效果，感觉上比较协调。

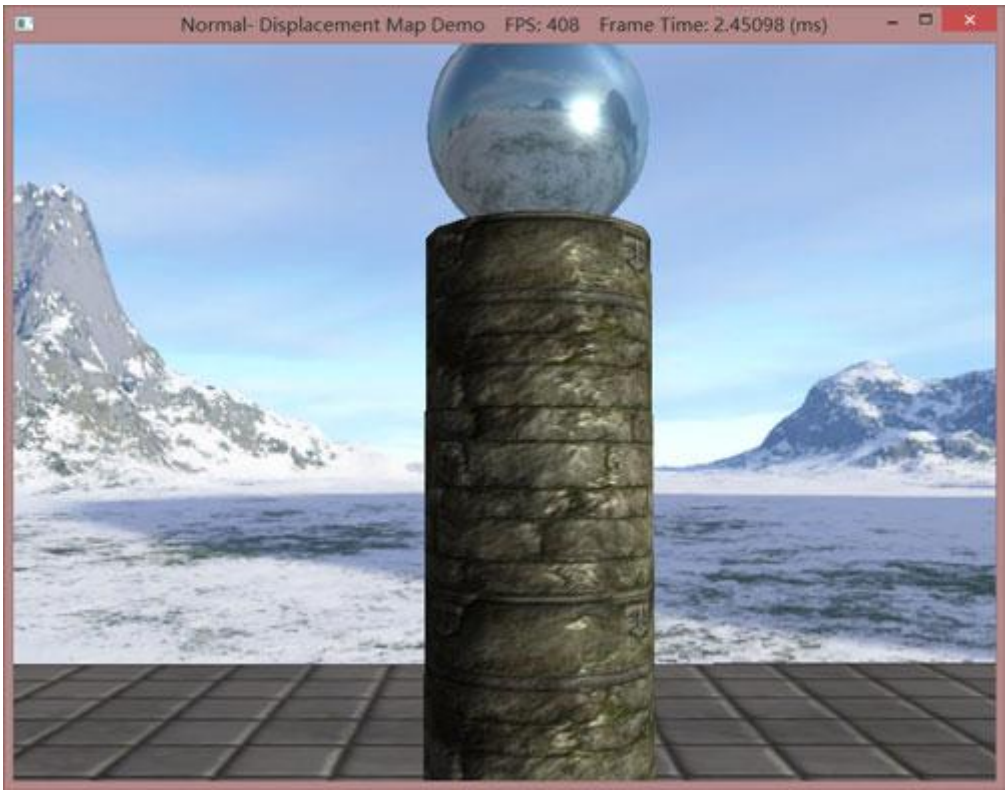



**图 18.2 凹凸不平的镜面高光**

# 18.2 法线贴图

法线贴图（normal map）是一种特殊的纹理，它的每个纹理元素存储的不是 RGB 数据，而是压缩后的 *x*、*y*、*z* 坐标（分别存储在 R、G、B 分量中）。这些坐标定义了一个法线向量；法线贴图的每个像素存储的都是这种法线向量。图 18.3 说明了法线贴图的实现方式。

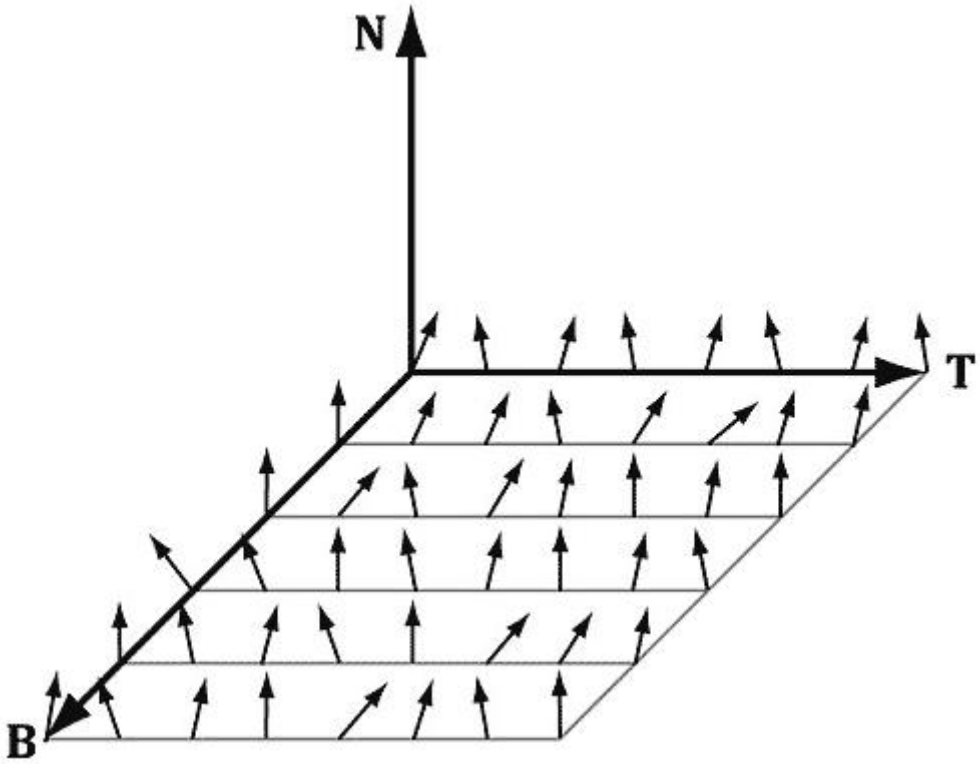



**图 18.3 法线向量存储在一个由向量 T（*x* 轴）、B（*y* 轴）、N（*z* 轴）构成的纹理空间坐标系中。向量 T 水平向右遍历法线贴图，向量 B 垂直向下遍历法线贴图，向量 N 垂直于法线贴图平面。**

为便于讲解，我们将采用 24 位图像格式，为每个颜色分量分配一个字节，故每个颜色分量的取值范围在 0 到 255 之间。（当然，也可以采用 32 位格式，只是 alpha 分量会被闲置，或者考虑用它存储一些其他的诸如高度贴图或高光贴图之类的标量值。在不需要压缩内存的情况下，也可以采用浮点格式，只是它占用的内存较多。）

**注意**：如图 18.3 所示，向量一般与 z 轴对齐。也就是，*z* 坐标的值最大。所以，当把法线贴图视为一个彩色图像时（例如在 Photoshop 中查看法线贴图），它一般呈现为偏蓝的颜色。这是因为 *z* 坐标存储在蓝色通道中，该通道的数值最大，蓝色占据首要地位。

那么，我们如何将一个单位向量压缩到这种格式中呢首先，对于单位向量来说，每个坐标都在[−1,1]区间内。如果我们将该区间调整为[0,1]、乘以 255 并舍去小数，那结果就是一个在[0,255]区间内的整数。也就是，如果 *x* 是一个在[−1,1]区间内的浮点数，那么 *f*(*x*)的整数部分就是一个在[0,255]区间内的整数。其中，*f* 的定义为：

$$f(x) = (0.5x + 0.5) \cdot 255$$

所以要将一个单位向量存入 24 位图像，我们只需要将函数 *f* 应用于每个坐标，并将坐标写入纹理贴图的对应颜色通道即可。

第二个问题是如何解压缩；也就是，当给出一个在[0,255]区间内的压缩纹理坐标时，如何将它恢复为[−1,1]区间内的原值。答案是将函数 *f* 简单的反转过来，只要稍加思索就可以看出：

$$f^{-1}(x) = \frac{2x}{255} - 1$$

也就是，如果 *x* 是一个在[0,255]区间内的整数，那么 $f^{-1}(x)$就是一个在[−1,1]区间内的浮点数。

我们不必自己实现压缩处理，因为我们可以用一个 Photoshop 插件来完成从图像到法线贴图的转换工作。不过，当我们在像素着色器中对一个法线贴图进行采样时，必须自己实现反转处理，对法线贴图进行解压缩。当我们在着色器中对一个法线贴图进行采样时，可使用如下代码：

```
float3 normalT = gNormalMap.Sample(gTriLinearSam,pIn.texC);
```

颜色向量 **normalT** 将会包含规范化分量$(r,g,b)$，其中 $0 \leqslant r$，$g$，$b \leqslant 1$。这样，该方法已经替我们完成了一部分解压缩工作（也就是除以 255、将[0,255]区间内的整数变换为[0,1]区间内的浮点数）。我们只需要使用函数 $g(x)$将每个分量从[0,1]区间调整到[−1,1]区间，即可完成的转换工作：

$$g(x) = 2x - 1$$

在代码中，我们将每个颜色分量代入该函数：

```
// 将每个分量从[0,1]解压到[-1,1].
normalT = 2.0f*normalT - 1.0f;
```

这条语句可以运行，因为标量 1.0 会被自动扩展为向量(1,1,1)，以使整个表达式完整有效，并完成分量运算。

**注意**：Photoshop 插件可以从 http://developer.nvidia.com/nvidia-texture-tools-adobe-photoshop 下载。还有一些其他的工具也可以生成法线贴图，比如 http://www.crazybump.com/。另外，还有一些工具可以根据高精度网格来生成法线贴图（参见 http://developer.nvidia.com/object/melody_home.html）。

**注意**：如果你想使用压缩纹理格式存储法线贴图，请使用 BC7（**DXGI_FORMAT_BC7_UNORM**）格式，因为这种格式可以显著地减少压缩法线贴图带来的错误，所以质量最高。对于 BC6 和 BC7 格式来说，DirectX SDK 中一个叫做“BC6HBC7EncoderDecoder11”的示例可以用来将纹理文件转换为 BC6 或 BC7。

# 18.3 纹理/正切空间

考虑一个 3D 纹理映射三角形。为便于讨论，我们假设在进行纹理贴图映射没有拉伸问题；也就是说，将纹理三角形映射到 3D 三角形的过程只是一个刚体变换过程（平移和旋转）。现在，假设纹理是一张彩色帖纸。我们对它进行平移和旋转，把它贴到 3D 三角形上。图 18.4 说明了如何在纹理坐标轴与 3D 三角形之间建立联系：它们正切于三角形，并位于三角形平面上。当然，三角形的纹理坐标是相对于纹理坐标系的。现在把三角形平面法线 **N** 合并进来，我们便得到了一个在三角形平面上的 3D TBN 基（3D TBN-basis），我们将它称为纹理空间（texture space）或正切空间（tangent space）。注意，正切空间通常会随着三角形而变化（参见图 18.5）。

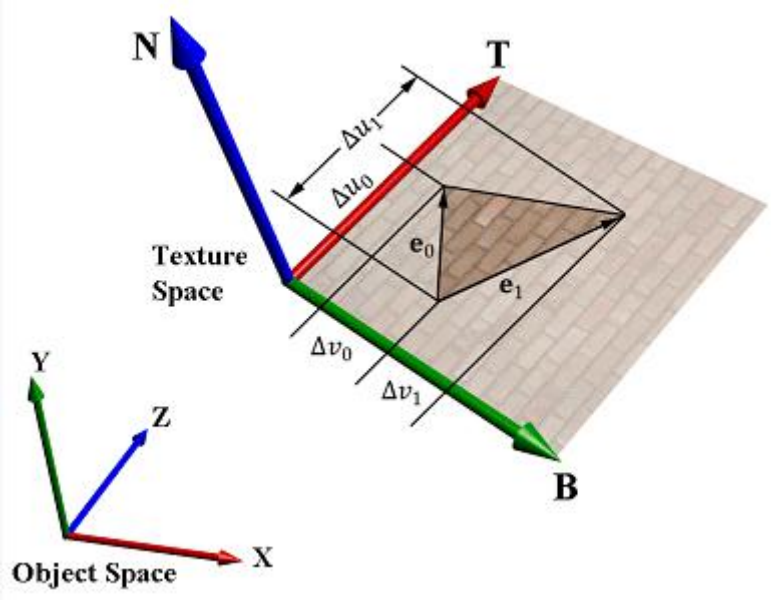



**图 18.4 三角形纹理空间与物体空间之间的关系。3D 切线向量 T 指向纹理坐标系统的 u 轴方向，3D 切线向量 B 指向纹理坐标系统的 v 轴方向。**

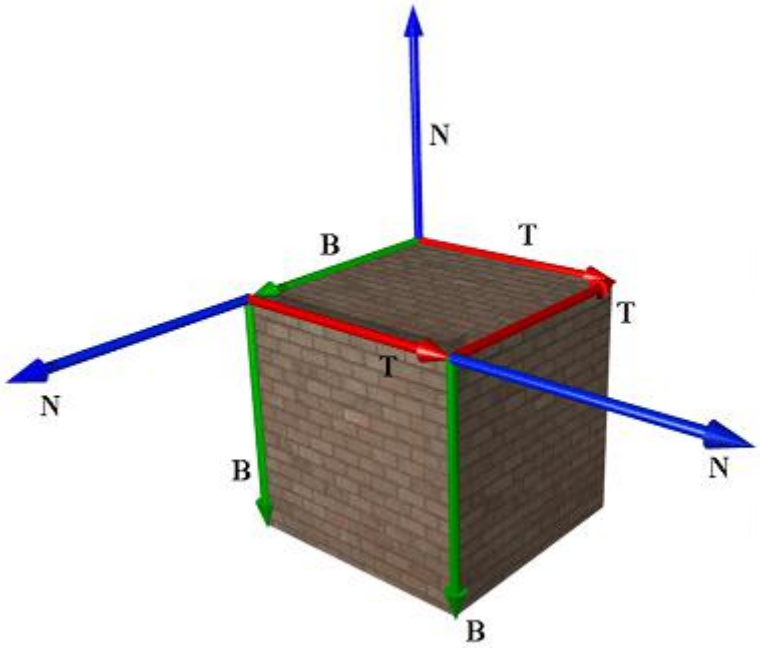



**图 18.5 在立方体上，每个平面的纹理空间都不一样。**

如图 18.3 所示，我们存储在法线贴图中的法线向量使用的是纹理空间坐标系。而场景中的光照使用的是世界空间坐标系。为了进行光照，我们必须把法线向量和灯光变换到同一个坐标系中。那么我们的第一步是要在正切空间坐标系和三角形顶点的物体空间坐标系之间建立联系。只要进入物体空间，我们就可以通过世界矩阵实现从物体空间到世界空间的坐标变换（具体实现过程在下一节讲解）。现在，我们设 $\mathbf{v}_0$、$\mathbf{v}_1$、$\mathbf{v}_2$ 为 3D 三角形的 3 个顶点，对应的纹理坐标为$(u_0,v_0)$、$(u_1,v_1)$、$(u_2,v_2)$，这些纹理坐标定义了纹理坐标轴（即，**T** 和 **B**）上的纹理三角形。设 $\mathbf{e}_0=\mathbf{v}_1-\mathbf{v}_0$、$\mathbf{e}_1=\mathbf{v}_2-\mathbf{v}_0$ 为 3D 三角形的两个边向量，对应的纹理三角形的边向量为$(\Delta u_0,\Delta v_0)=(u_1-u_0,v_1-v_0)$、$(\Delta u_1,\Delta v_1)=(u_2-u_0,v_2-v_0)$。从图 18.4 中可以看出：

$$\mathbf{e}_0=\Delta u_0\mathbf{T}+\Delta v_0\mathbf{B}$$
$$\mathbf{e}_1=\Delta u_1\mathbf{T}+\Delta v_1\mathbf{B}$$

使用相对于物体空间的坐标表示向量，可以得到矩阵方程：

$$\begin{bmatrix} e_{0,x} & e_{0,y} & e_{0,z} \\ e_{1,x} & e_{1,y} & e_{1,z} \end{bmatrix} = \begin{bmatrix} \Delta u_0 & \Delta v_0 \\ \Delta u_1 & \Delta v_1 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

注意，只要我们知道三角形顶点的物体空间坐标，那么就等于知道了边向量的物体空间坐标。所以矩阵

$$\begin{bmatrix} e_{0,x} & e_{0,y} & e_{0,z} \\ e_{1,x} & e_{1,y} & e_{1,z} \end{bmatrix}$$

是已知的。同样，我们知道纹理坐标，所以矩阵

$$\begin{bmatrix} \Delta u_0 & \Delta v_0 \\ \Delta u_1 & \Delta v_1 \end{bmatrix}$$

是已知的。求解 **T** 和 **B** 的物体空间坐标：

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \begin{bmatrix} \Delta u_0 & \Delta v_0 \\ \Delta u_1 & \Delta v_1 \end{bmatrix}^{-1} \begin{bmatrix} e_{0,x} & e_{0,y} & e_{0,z} \\ e_{1,x} & e_{1,y} & e_{1,z} \end{bmatrix}$$

$$= \frac{1}{\Delta u_0 \Delta v_1 - \Delta v_0 \Delta u_1} \begin{bmatrix} \Delta v_1 & -\Delta v_0 \\ -\Delta u_1 & \Delta u_0 \end{bmatrix} \begin{bmatrix} e_{0,x} & e_{0,y} & e_{0,z} \\ e_{1,x} & e_{1,y} & e_{1,z} \end{bmatrix}$$

我们在上述方程中使用了一个逆矩阵运算的推论，即给定一个矩阵 $\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$，可有：

$$\mathbf{A}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

注意，物体空间中的向量 **T** 和 **B** 通常不是单位向量，而且当纹理出现拉伸问题时，它们彼此并不垂直。

**T**、**B**、**N** 向量分别称为切线向量、副法线向量（或副切线向量）、法线向量。

# 18.4 顶点正切空间

在上一节中，我们为三角形推导了一个正切空间。不过，当进行法线贴图映射时，我们希望以三角形的形式描述该正切空间，因为正切空间在三角形平面上是一个常量。所以，我们在每个顶点上指定切线向量，就像是使用顶点法线来模拟一个光滑表面时所做的事情一样：

1．对于网格中的任意一个顶点 **v**，它的切线向量 **T** 等于共享该顶点的每个三角形的切线向量的平均值。

2．对于网格中的任意一个顶点 **v**，它的副切（法）线向量 **B** 等于共享该顶点的每个三角形的副切（法）线向量的平均值。

在计算平均值之后，我们通常需要对 TBN 基进行正交化处理，使这 3 个向量彼此垂直并为单位向量。这一工作通常使用 Gram-Schmidt 算法来完成。读者可以在网上找到能为任意三角形网格生成顶点切线空间的代码：http://www.terathon.com/code/tangent.php。

这里，我们不直接在内存中存储副切线向量 **B**。而是，在需要 B 时计算 **B**=**N**×**T**，其中

**N** 是常规的经过平均后的顶点法线。此时，我们的顶点结构体为：

```
namespace Vertex
{
    struct NormalMap
    {
        XMFLOAT3 Pos;
        XMFLOAT3 Normal;
        XMFLOAT2 Tex;
        XMFLOAT3 TangentU;
    } ;
}
```

回忆一下由 **GeometryGenerator** 程序生成的网格计算了对应 *u* 轴的切线向量 **T**。对于 Grid 和 Box 来说，我们可以直接在每个顶点中指定切线向量 **T** 的物体空间坐标（参见图 18.5）。对于 Cylinder 和 Sphere 来说，我们可以通过定义向量值函数 **P**(*u*,*v*)和计算∂**P**/∂u 来求出每个顶点上的切线向量，其中参数 *u* 还被用作为 *u* 纹理坐标。

# 18.5 在切线空间和物体空间之间变换

现在，我们在网格的每个顶点上都有一个正交 TBN 基，而且还有相对于网格物体空间的 TBN 向量坐标。所以，我们现在可以得到一个相对于物体空间坐标系的 TBN 矩阵，通过该矩阵我们可以将坐标从切线空间变换到物体空间：

$$\mathbf{M}_{object} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{bmatrix}$$

由于该矩阵是正交矩阵，它的逆矩阵和转置矩阵相同。所以，从物体空间到正切空间的坐标转换矩阵为：

$$\mathbf{M}_{tangent} = \mathbf{M}_{object}^{-1} = \mathbf{M}_{object}^{T} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

为了在着色器代码中进行光照计算，我们希望将法线向量从正切空间变换到世界空间。一种实现方式是，先将法线向量从正切空间变换到物体空间，然后再从物体空间变换到世界空间：

$$\mathbf{n}_{world} = (\mathbf{n}_{tangent}\mathbf{M}_{object})\mathbf{M}_{world}$$

不过，矩阵乘法满足结合律，所以我们可以把它改写为：

$$\mathbf{n}_{world} = \mathbf{n}_{tangent}(\mathbf{M}_{object}\mathbf{M}_{world})$$

注意

$$\mathbf{M}_{object}\mathbf{M}_{world} = \begin{bmatrix} \leftarrow \mathbf{T} \rightarrow \\ \leftarrow \mathbf{B} \rightarrow \\ \leftarrow \mathbf{N} \rightarrow \end{bmatrix}\mathbf{M}_{world} = \begin{bmatrix} \leftarrow \mathbf{T}' \rightarrow \\ \leftarrow \mathbf{B}' \rightarrow \\ \leftarrow \mathbf{N}' \rightarrow \end{bmatrix} = \begin{bmatrix} T'_x & T'_y & T'_z \\ B'_x & B'_y & B'_z \\ N'_x & N'_y & N'_z \end{bmatrix}$$

其中，$\mathbf{T}' = \mathbf{T} \cdot \mathbf{M}_{world}$、$\mathbf{B}' = \mathbf{B} \cdot \mathbf{M}_{world}$、$\mathbf{N}' = \mathbf{N} \cdot \mathbf{M}_{world}$。这样，法线向量可以直接从正

切空间变换到世界空间。我们只需要描述在世界空间中的 TBN 基，就可以将法线向量从物体空间变换到世界空间。

**注意**：我们只对向量进行变换（不考虑点的变换）。所以，我们只需要一个 3×3 矩阵。前面讲过，仿射矩阵的第 4 行用于平移，但是我们在这里不需要平移向量。

# 18.6 着色器代码

我们将法线贴图映射的一般处理过程总结如下：

1．使用某个绘图软件或工具软件创建法线贴图，并保存为图像文件。当程序初始化时，从这些文件创建 2D 纹理。

2．为每个三角形计算切线向量 **T**。在网格中，顶点 **v** 的切线向量等于共享该顶点的每个三角形的切线向量的平均值。（在我们的演示程序中，我们为简单几何体直接指定切线向量，为那些从 3D 建模软件导出的复杂三角形网格计算平均值。）

3．在顶点着色器中将顶点的法线向量和切线向量变换到世界空间，然后将结果输出到像素着色器。

4．使用插值后的切线向量和法线向量，为三角形表面上的每个像素点生成 TBN 基。使用 TBN 基，将从法线贴图采样得到的法线向量从切线空间变换到世界空间。最后将法线向量用于光照计算。

要实现法线法线贴图映射，我们需要在 *lighthelper.fx* 中添加以下函数：

```
//---------------------------------------------------------------------
// 将法线贴图的采样值转换到世界空间.
//---------------------------------------------------------------------
float3 NormalSampleToWorldSpace(float3 normalMapSample, float3 unitNormalW, float3 tangentW)
{
    // 将每个分量从[0,1]解压到[-1,1].
    float3 normalT = 2.0f*normalMapSample - 1.0f;

    // 创建 TBN 基.
    float3 N = unitNormalW;
    float3 T = normalize(tangentW - dot(tangentW, N)*N);
    float3 B = cross(N, T);

    float3x3 TBN = float3x3(T, B, N);

    // 从切线空间转换到世界空间.
    float3 bumpedNormalW = mul(normalT, TBN);

    return bumpedNormalW;
}
```

这个函数在像素着色器中的使用方法如下：

```
float3 normalMapSample = gNormalMap.Sample(samLinear, pin.Tex).rgb;
float3 bumpedNormalW = NormalSampleToWorldSpace(normalMapSample,
pin.NormalW, pin.TangentW);
```

这里有两行代码需要特别说明一下：

```
float3 N = unitNormalW;
float3 T = normalize(tangentW - dot(tangentW, N)*N);
```

在插值之后，切线向量和法线向量可能不再相互垂直。这两行代码的用途是通过从 **T** 中减去偏向于 **N** 的部分（见图 18.6），使 T 重新垂直于 **N**。注意，我们假设 **unitNormalW** 已被规范化。

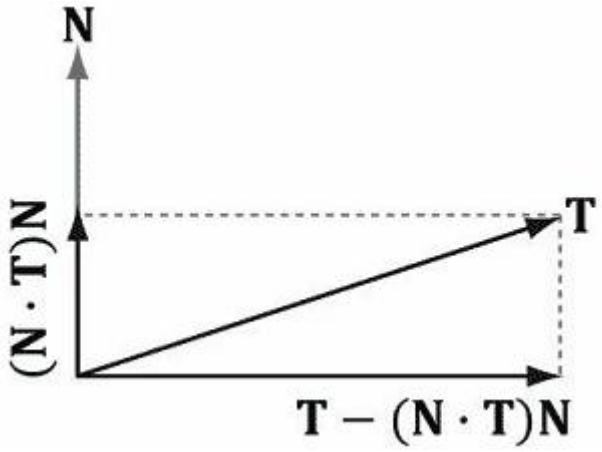



**图 18.6 因为 ‖N‖=1，所以 $proj_N(T) = (T \cdot N)N$。向量 $T - proj_N(T)$是 T 垂直于 N 的部分。**

在从法线贴图中获取了法线之后，就可以将它用在所有包含法线向量的计算中（即，光照、立方贴图映射）。我们将整个法线贴图映射的 effect 代码都列了出来。

```
#include "LightHelper.fx"

cbuffer cbPerFrame
{
    DirectionalLight gDirLights[3];
    float3 gEyePosW;

    float  gFogStart;
    float  gFogRange;
    float4 gFogColor;
};

cbuffer cbPerObject
{
    float4x4 gWorld;
    float4x4 gWorldInvTranspose;
    float4x4 gWorldViewProj;
    float4x4 gTexTransform;
    Material gMaterial;
};
```

```
// Nonnumeric values cannot be added to a cbuffer.
Texture2D gDiffuseMap;
Texture2D gNormalMap;
TextureCube gCubeMap;

SamplerState samLinear
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

struct VertexIn
{
    float3 PosL     : POSITION;
    float3 NormalL  : NORMAL;
    float2 Tex      : TEXCOORD;
    float3 TangentL : TANGENT;
};

struct VertexOut
{
    float4 PosH     : SV_POSITION;
    float3 PosW     : POSITION;
    float3 NormalW  : NORMAL;
    float3 TangentW : TANGENT;
    float2 Tex      : TEXCOORD;
};

VertexOut VS(VertexIn vin)
{
    VertexOut vout;

    // Transform to world space space.
    vout.PosW     = mul(float4(vin.PosL, 1.0f), gWorld).xyz;
    vout.NormalW  = mul(vin.NormalL, (float3x3)gWorldInvTranspose);
    vout.TangentW = mul(vin.TangentL, (float3x3)gWorld);

    // Transform to homogeneous clip space.
    vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj);

    // Output vertex attributes for interpolation across triangle.
    vout.Tex = mul(float4(vin.Tex, 0.0f, 1.0f), gTexTransform).xy;
```

```
    return vout;
}

float4 PS(VertexOut pin,
          uniform int gLightCount,
          uniform bool gUseTexure,
          uniform bool gAlphaClip,
          uniform bool gFogEnabled,
          uniform bool gReflectionEnabled) : SV_Target
{
    // Interpolating normal can unnormalize it, so normalize it.
    pin.NormalW = normalize(pin.NormalW);

    // The toEye vector is used in lighting.
    float3 toEye = gEyePosW - pin.PosW;

    // Cache the distance to the eye from this surface point.
    float distToEye = length(toEye);

    // Normalize.
    toEye /= distToEye;

    // Default to multiplicative identity.
    float4 texColor = float4(1, 1, 1, 1);
    if(gUseTexure)
    {
        // Sample texture.
        texColor = gDiffuseMap.Sample( samLinear, pin.Tex );

        if(gAlphaClip)
        {
            // Discard pixel if texture alpha < 0.1.  Note that we do
this
            // test as soon as possible so that we can potentially exit
the shader
            // early, thereby skipping the rest of the shader code.
            clip(texColor.a - 0.1f);
        }
    }

    //
    // 法线映射
    //
```

```
    float3 normalMapSample = gNormalMap.Sample(samLinear,
pin.Tex).rgb;
    float3 bumpedNormalW = NormalSampleToWorldSpace(normalMapSample,
pin.NormalW, pin.TangentW);

    //
    // Lighting.
    //

    float4 litColor = texColor;
    if( gLightCount > 0  )
    {
        // Start with a sum of zero.
        float4 ambient = float4(0.0f, 0.0f, 0.0f, 0.0f);
        float4 diffuse = float4(0.0f, 0.0f, 0.0f, 0.0f);
        float4 spec    = float4(0.0f, 0.0f, 0.0f, 0.0f);

        // Sum the light contribution from each light source.
        [unroll]
        for(int i = 0; i < gLightCount; ++i)
        {
            float4 A, D, S;
            ComputeDirectionalLight(gMaterial, gDirLights[i],
bumpedNormalW, toEye,
                A, D, S);

            ambient += A;
            diffuse += D;
            spec    += S;
        }

        litColor = texColor*(ambient + diffuse) + spec;

        if( gReflectionEnabled )
        {
            float3 incident = -toEye;
            float3 reflectionVector = reflect(incident,
bumpedNormalW);
            float4 reflectionColor  = gCubeMap.Sample(samLinear,
reflectionVector);

            litColor += gMaterial.Reflect*reflectionColor;
        }
    }
```

```
    //
    // Fogging
    //

    if( gFogEnabled )
    {
        float fogLerp = saturate( (distToEye - gFogStart) /
gFogRange );

        // Blend the fog color and the lit color.
        litColor = lerp(litColor, gFogColor, fogLerp);
    }

    // Common to take alpha from diffuse material and texture.
    litColor.a = gMaterial.Diffuse.a * texColor.a;

    return litColor;
}
```