隐藏表面去除

我们现在可以从任何角度渲染任何场景，但生成的图像在视觉上很简单：我们在线框中渲染对象，给人的印象是我们正在查看一组对象的蓝图，而不是对象本身。

本书的其余章节侧重于提高渲染场景的视觉质量。在本章结束时，我们将能够渲染看起来实体的对象（而不是线框）。我们已经开发了一种绘制填充三角形的算法，但正如我们将看到的，在 3D 场景中正确使用该算法并不像看起来那么简单！

渲染实体对象

当我们想要使固体对象看起来是实体时，首先想到的是使用我们在第 7 章（填充三角形）中开发的功能，使用随机颜色绘制对象的每个三角形（图 12-1）。`DrawFilledTriangle`

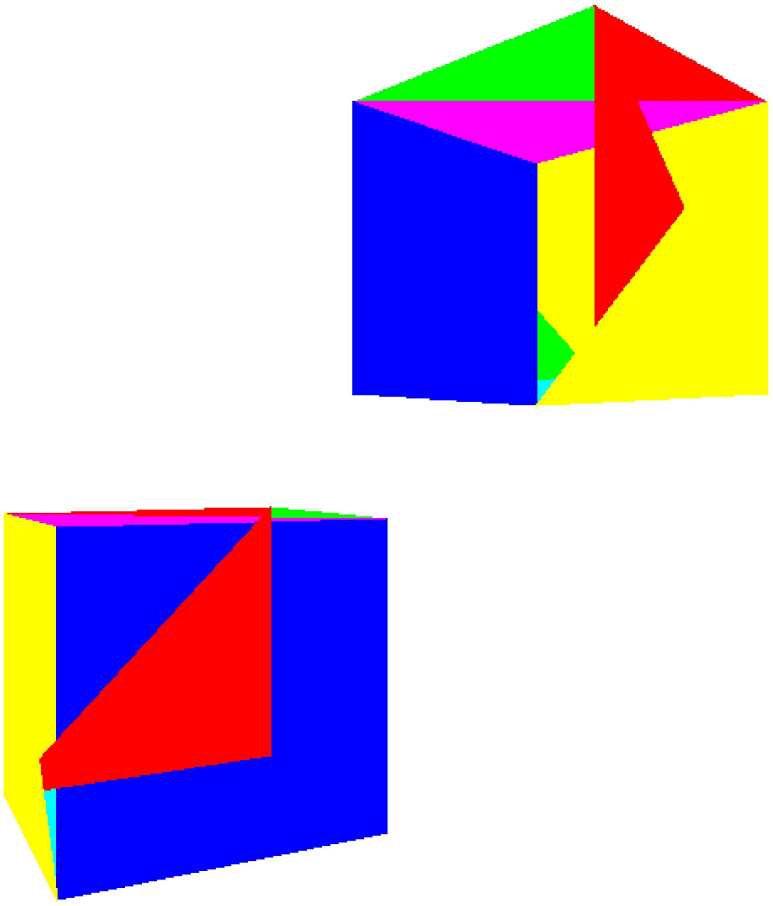

图 12-1：使用 而不是 不会产生我们预期的结果。`DrawFilledTriangle``DrawWireframeTriangle`

图 12-1 中的形状看起来不太像立方体，是吗？如果你仔细观察，你会发现问题出在哪里：立方体背面的一部分被画在正面的顶部！这是因为我们盲目地以“随机顺序”在画布上绘制 2D 三角形，或者更准确地说，按照它们恰好在模型列表中定义的顺序绘制，而不考虑它们之间的空间关系。`Triangles`

您可能想回到模型定义并更改三角形的顺序来解决此问题。但是，如果我们的场景包含旋转的立方体的另一个实例180°，我们会回到最初的问题。简而言之，没有一个“正确”的三角形顺序适用于每个实例和相机方向。我们应该怎么做？

画家的算法

这个问题的第一个解决方案被称为*画家算法*。现实生活中的画家首先绘制背景，然后用前景物体覆盖其中的一部分。我们可以通过从后到前绘制场景中的所有三角形来实现相同的效果。为此，我们将应用模型和相机变换，并根据三角形与相机的距离对三角形进行排序。

这解决了上面解释的“没有单一的正确顺序”问题，因为现在我们正在寻找对象和相机的特定相对位置的正确顺序。

虽然这确实会以正确的顺序绘制三角形，但它有一些缺点，使其不切实际。

首先，它不能很好地扩展。人类已知的最有效的排序算法是$O\ (n \cdot \log\ (n))$这意味着如果我们将三角形的数量加倍，运行时将增加一倍以上。（举例来说，对 100 个三角形进行排序大约需要 200 次操作;对 200 个三角形进行排序需要 460 次，而不是 400 次;对 800 个三角形进行排序需要 2，322 次操作，而不是 1，840 次！换句话说，这适用于小场景，但随着场景复杂性的增加，它很快就会成为性能瓶颈。

其次，它要求我们一次知道整个三角形列表。这需要大量内存，并阻止我们使用类似流的方法进行渲染。我们希望渲染器就像一个管道，其中模型三角形从一端进入，像素从另一端进入，但该算法在每个三角形被转换和排序之前不会开始绘制像素。

第三，即使我们愿意忍受这些限制，在某些情况下，三角形的正确顺序*根本不存在*。考虑图 12-2 中的情况。我们将*永远*无法以产生正确结果的方式对这些三角形进行排序。

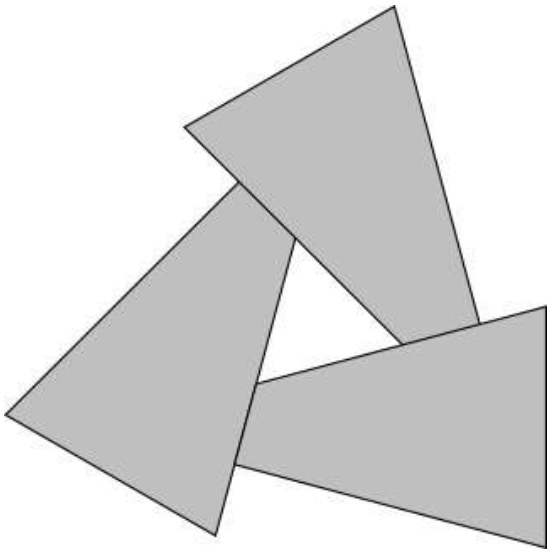

图 12-2：无法“从后到前”对这些三角形进行排序。

深度缓冲

我们无法在三角形级别解决排序问题，因此让我们尝试在像素级别解决它。

对于画布上的每个像素，我们希望用“正确”的颜色绘制它，其中“正确”颜色是最接近相机的对象的颜色。在图 12-3 中，这是P_1.

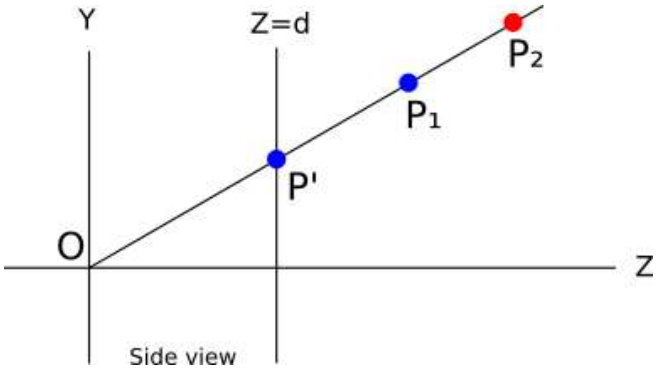

图 12-3：P_1 和 P_2 都投影到画布上的同一 P'。因为 P_1 比 P_2 更接近相机，所以我们想把 P' 涂成 P_1 的颜色。

在渲染过程中的任何时候，画布上的每个像素都代表场景中的一个点（在我们绘制任何东西之前，它代表无限远的点）。假设对于画布上的每个像素，我们保留了Z它当前表示的点的坐标。当我们需要决定是否用对象的颜色绘制像素时，我们只会在Z我们要绘制的点的坐标小于Z已存在的点的坐标。这保证了表示场景中某个点的像素永远不会被表示离相机较远的点的像素所绘制。

让我们回到图 12-3。假设由于模型中三角形的顺序，我们要绘制P_2首先和P_1第二。当我们画画时P_2，像素被涂成红色，其关联Z价值变为Z_{P_2}.然后我们想画画P_1，并且从$Z_{P_2} > Z_{P_1}$，我们将像素涂成蓝色，得到正确的结果。

在这种特殊情况下，无论Z，因为这些点恰好以方便的顺序出现。但是，如果我们想画画呢P_1首先和P_2第二？我们首先将像素涂成蓝色并存储Z_{P_1};但是当我们想画画时P_2，我们看到$Z_{P_2} > Z_{P_1}$，所以我们*不画*它——因为如果我们画了，P_1将被涵盖P_2，更远！我们再次得到一个蓝色像素，这是正确的结果。

在实现方面，我们需要一个缓冲区来存储Z画布上每个像素的坐标;我们称之为*深度缓冲区*。它具有与画布相同的尺寸，但其元素是表示深度值的实数，而不是像素。

但是在哪里Z价值观从何而来？这些应该是Z点在转换之后但在透视投影之前的值。但是，这只能给我们Z顶点的值;我们需要一个Z每个三角形的每个像素的值。

这是我们在第8章（阴影三角形）中开发的属性映射算法的另一个应用。为什么不使用Z作为属性并将其插值到三角形的表面，就像我们之前使用颜色强度值所做的那样？到现在为止，您已经知道该怎么做了：取 、 和 ;计算 、 和 ;将它们组合起来得到和;然后，对于每个水平段，计算 .最后，我们不是盲目调用，而是这样做：`z0z1z2z01z02z012z_leftz_rightz_segmentPutPixel(x, y, color)`

```
z = z_segment[x - xl]
if (z < depth_buffer[x][y]) {
    canvas.PutPixel(x, y, color)
    depth_buffer[x][y] = z
}
```

要使其正常工作，应将中的每个条目初始化为`depth_buffer`＋∞（或者只是“非常大的价值”）。这保证了我们第一次要绘制像素时，条件将为真，因为场景中的任何点都比无限远的点更接近相机。

我们现在得到的结果要好得多——看看图 12-4。

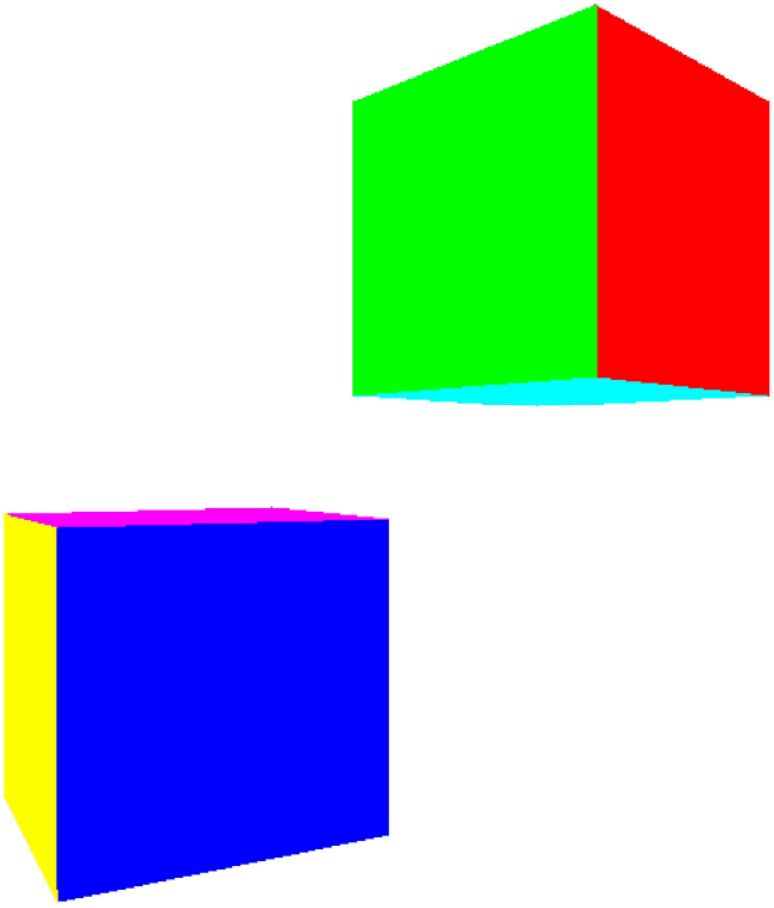

图 12-4：立方体现在看起来像立方体，无论其三角形的顺序如何。

源代码和现场演示>>

使用 1/Z 代替 Z

结果看起来好多了，但我们所做的是微妙的错误。的价值观Z因为顶点是正确的（毕竟它们来自数据），但在大多数情况下，线性插值为Z其余像素不正确。在这一点上，这甚至可能不会导致明显的差异，但以后会成为一个问题。

要了解这些值是如何错误的，请考虑从A（-1，0，2）到B（1，0，10）的线段的简单情况，其中点M位于（0，0，6）。具体地说，因为M是AB的中点，我们知道$M_z = (A_z + B_z)/2 = 6$ 图 12-5 显示了此线段。

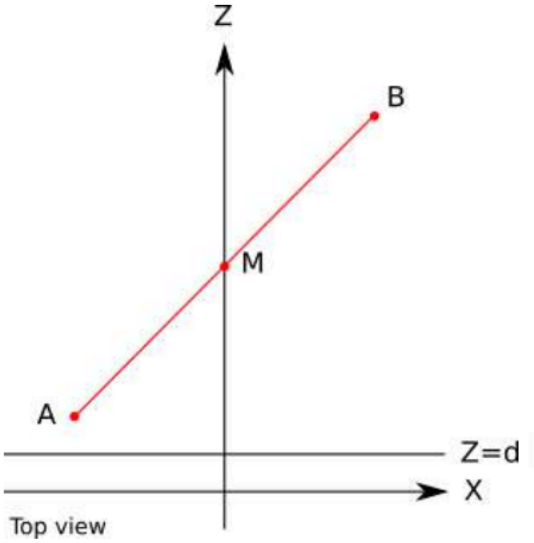

图 12-5：线段 AB 及其中点 M

让我们用$d = 1$.应用透视投影方程，应用透视投影方程，我们得到A'_x =A_x/A_z =1/2=0.5 类似地$B'_x = 0.1$和$M'_x = 0.$ 图 12-6 显示了投影点。

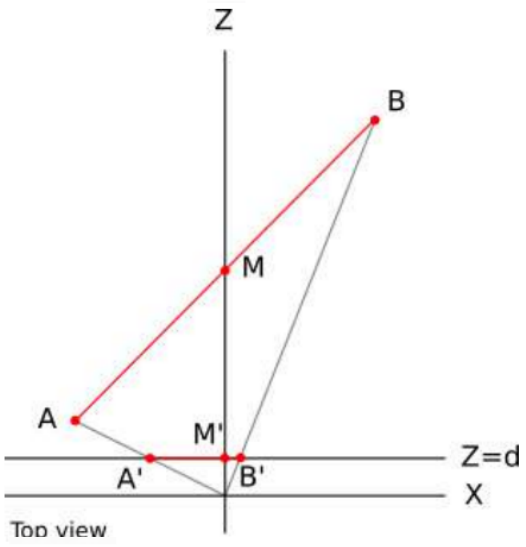

图 12-6：投影到投影平面上的点 A、B 和 M

$A'B'$是视口上的水平线段。我们知道A_z 和 B_z.让我们看看如果我们尝试计算M_z使用线性插值。隐含的线性函数如图 12-7 所示。

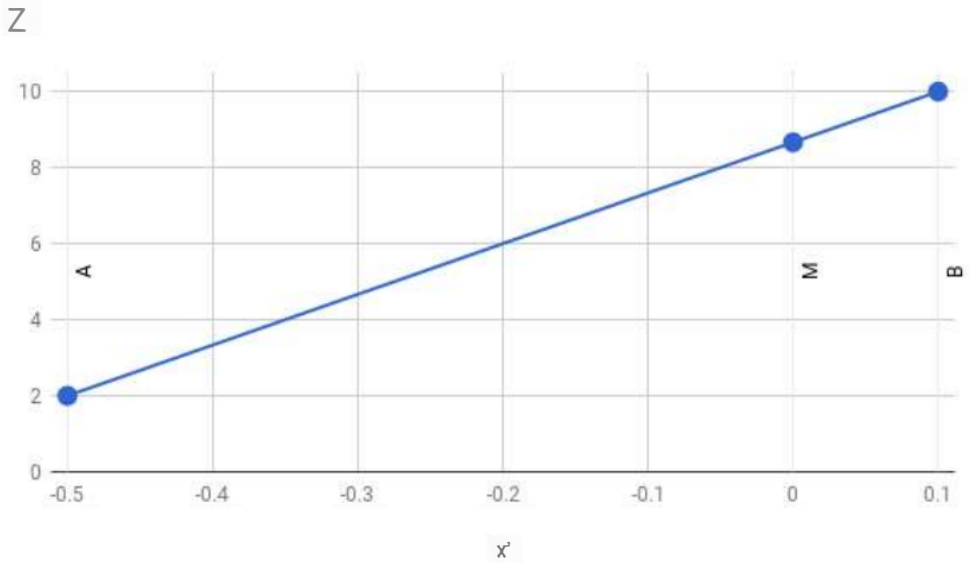

图 12-7：A x' 和 B x' 的 A z 和 B z 值定义了线性函数 z = f (x')。

函数的斜率是恒定的，所以我们可以写

$$\frac{M_z - A_z}{M'_x - A'_x} = \frac{B_z - A_z}{B'_x - A'_x}$$

我们可以操纵该表达式来解决M_z:

$$M_z = A_z + (M'_x - A'_x)(\frac{B_z - A_z}{B'_x - A'_x})$$

如果我们插入我们知道的值并做一些算术，我们会得到

$$M_z = 2 + (0 - (-0.5))(\frac{10 - 2}{0.1 - (-0.5)}) = 2 + (0.5)(\frac{8}{0.6}) = 8.666$$

这说明M_z是8.666，但我们知道它实际上是6!

我们哪里做错了？我们使用线性插值，我们知道它效果很好，我们给它提供来自数据的正确值，那么为什么结果是错误的呢？

我们的错误隐藏在我们使用线性插值时所做的隐含假设中：我们正在插值的函数一开始就是线性的！在这种情况下，事实证明并非如此。

如果$Z = f(x', y')$是x'和y'，对于A、B和C的某些值，我们可以把它写成$Z = Ax' + By' + C$.这类函数具有以下特性：两点之间的值之差取决于点之间的差异，而不是点本身的性质：

$$\begin{aligned} f(x' + \Delta x, y + \Delta y) - f(x', y') &= [A(x' + \Delta x) + B(y' + \Delta y) + C] - [A \cdot x' + B \cdot y' + C] \\ &= A(x' + \Delta x - x') + B(y' + \Delta y - y') + C - C \\ &= A\Delta x + B\Delta y \end{aligned}$$

也就是说，对于给定的屏幕坐标差异，在Z永远是一样的。

更正式地说，包含我们正在研究的线段的平面方程为

$$Ax + By + Cz + D = 0$$

另一方面，我们有透视投影方程：

$$x' = \frac{x \cdot d}{z}$$

$$y' = \frac{x \cdot d}{z}$$

我们可以得到x和y从这些回来：

$$x = \frac{z \cdot x'}{d}$$

$$y = \frac{z \cdot y'}{d}$$

如果我们替换x和y在具有这些表达式的平面方程中，我们得到

$$\frac{Ax'z + By'z}{d} + Cz + D = 0$$

乘以d然后求解z,

$$Ax'z + By'z + dCz + dD = 0$$

$$(Ax' + By' + dC)z + dD = 0$$

$$z = \frac{-dD}{Ax' + By' + dC}$$

这显然不是x'和y'，这就是为什么线性插值z给了我们一个不正确的结果。

但是，如果我们计算$1/z$而不是z，我们得到

$$1/z = \frac{Ax' + By' + dC}{-dD}$$

这显然是x'和y'.这意味着我们可以线性插值$1/z$并获得正确的结果。

为了验证这是否有效，让我们计算M_z，但这次使用的线性插值$1/z$:

$$\frac{M_{\frac{1}{z}} - A_{\frac{1}{z}}}{M'_x - A'_x} = \frac{B_{\frac{1}{z}} - A_{\frac{1}{z}}}{B'_x - A'_x}$$

$$M_{\frac{1}{z}} = A_{\frac{1}{z}} + (M'_x - A'_x)(\frac{B_{\frac{1}{z}} - A_{\frac{1}{z}}}{B'_x - A'_x})$$

$$M_{\frac{1}{z}} = \frac{1}{2} + (0 - (-0.5))(\frac{\frac{1}{10} - \frac{1}{2}}{0.1 - (-0.5)}) = 0.166666$$

因此

$$M_z = \frac{1}{M_{\frac{1}{z}}} = \frac{1}{0.166666} = 6$$

这个值是正确的，因为它符合我们最初的计算M_z基于线段的几何形状。

所有这些都意味着我们需要使用$1/z$而不是值z用于深度缓冲。伪代码中唯一实际的区别是缓冲区中的每个条目都应初始化为0（这在概念上是$1/+\infty$），并且比较应该颠倒（我们保留较大的值$1/z$，对应于较小的值z).

背面剔除

深度缓冲产生所需的结果。但是我们能让事情变得更快吗?

回到立方体，即使每个像素最终都有正确的颜色，其中许多像素也会被绘制好几次。例如，如果在正面之前渲染立方体的背面，则将绘制两次许多像素。这可能代价高昂。到目前为止，我们一直在计算$1/z$对于每个像素，但很快我们将添加更多属性，例如照明。随着我们需要执行的每像素操作数量的增加，永远不会可见的计算像素变得越来越浪费。

在我们进行所有这些计算之前，我们可以更早地丢弃像素吗？事实证明，我们可以在开始渲染之前丢弃*整个三角形*!

到目前为止，我们一直在非正式地谈论*正面*和*背面*。想象一下，每个三角形都有两条不同的边;不可能同时看到三角形的两条边。为了区分两条边，我们将在每个三角形上贴一个假想的箭头，垂直于其表面。然后我们将拿立方体并确保每个箭头都指向。图 12-8 显示了这个想法。

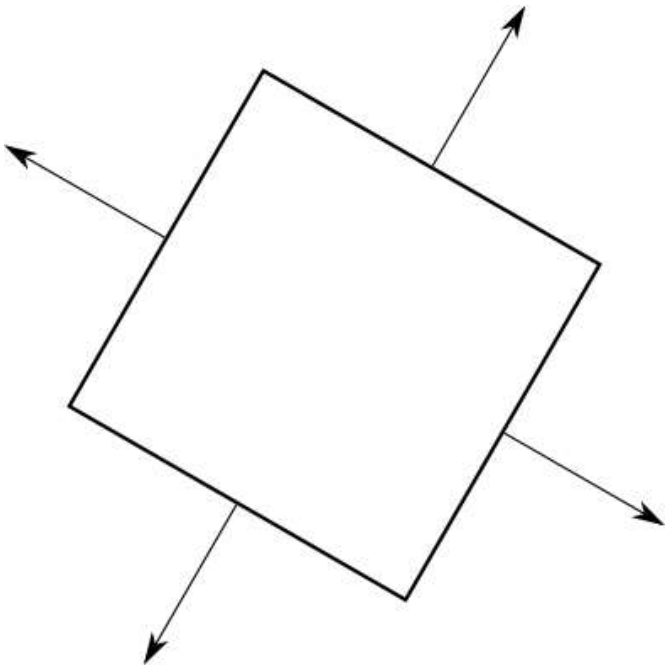

图 12-8：从上方查看的立方体，每个三角形上的箭头指向

这些箭头让我们可以将每个三角形分类为“前”或“后”，具体取决于它们是指向相机还是远离相机。更正式地说，如果视图向量和这个箭头（实际上是三角形的法向量）形成小于90°，三角形朝前;否则，它是朝后的（图 12-9）。

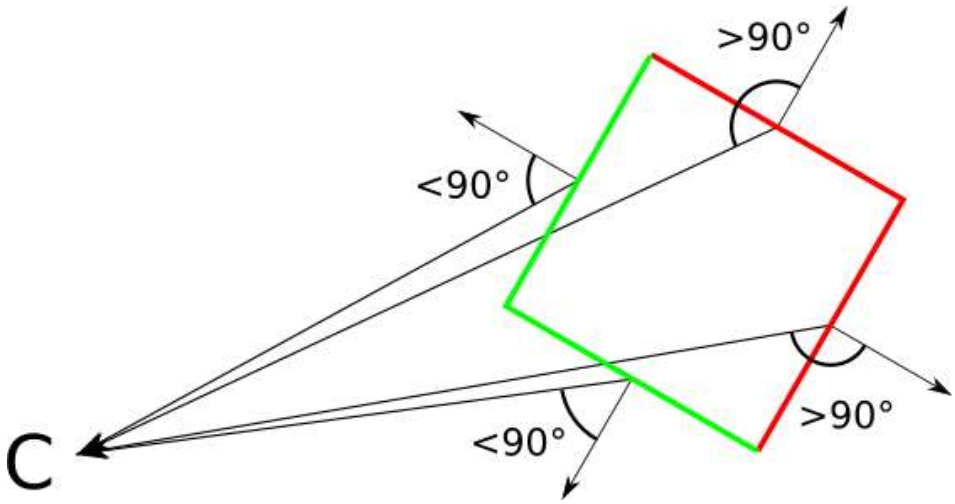

图 12-9：视图向量和三角形法向量之间的角度允许我们将其分类为正面或背面。

在这一点上，我们需要对3D模型施加限制：它们是*封闭的*。封闭的确切定义相当复杂，但幸运的是，直观的理解就足够了。我们一直在使用的多维数据集已关闭;我们只能看到它的外观。如果我们去掉它的一张脸，它就不会被关闭，因为我们可以看到它的内部。这并不意味着我们不能有孔或凹陷的物体;我们只会用薄薄的“墙”来建模这些。有关一些示例，请参见图 12-10。

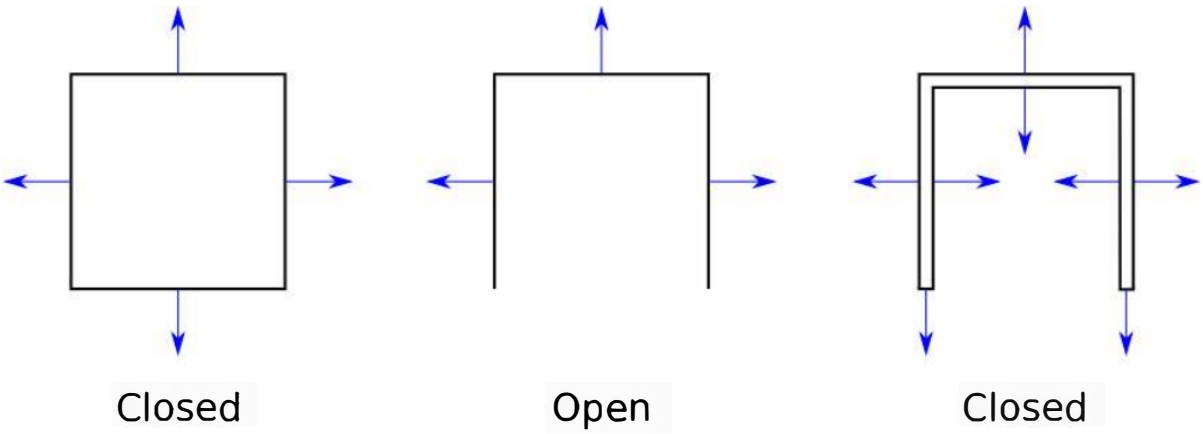

图 12-10：打开和关闭对象的一些示例

为什么要施加此限制？闭合对象具有有趣的特性，即无论模型或相机的方向如何，正面集都完全覆盖了背面集。这意味着我们根本不需要绘制背面，从而节省了宝贵的计算时间。

由于我们可以丢弃（剔除）所有背面，因此此算法称为*背面剔除*。它的伪代码对于可以将我们的渲染时间缩短一半的算法来说非常简单！

```
CullBackFaces(object, camera) {
  for T in object.triangles {
    if T is back-facing {
      remove T from object.triangles
    }
  }
}
```

示例 12-1：背面剔除算法

让我们更详细地了解如何确定三角形是正面还是背面。

对三角形进行分类

假设我们有法线向量$\vec{N}$三角形和向量$\vec{V}$从三角形的顶点到相机。现在假设$\vec{N}$指向对象的外部。为了将三角形分类为正面或背面，我们计算了两者之间的角度$\vec{N}$和$\vec{V}$并检查它们是否在$90°$彼此之间。

我们可以再次使用点积的属性来简化此操作。请记住，如果α是两者之间的角度$\vec{N}$和$\vec{V}$然后

$$\frac{\langle \vec{N}, \vec{V} \rangle}{|\vec{N}||\vec{V}|} = \cos(\alpha)$$

因为cos（ α ）对于$|\alpha| \leq 90°$是非负的，我们只需要知道这个表达式的符号即可将三角形分类为正面或背面。请注意，$|\vec{N}|$和$|\vec{V}|$始终为正数，因此它们不会影响表达式的符号。因此

$$\text{sign}(\langle \vec{N}, \vec{V} \rangle) = \text{sign}(\cos(\alpha))$$

分类标准很简单：

$$\begin{aligned} \langle \vec{N}, \vec{V} \rangle \leq 0 \text{ Back-facing} \\ \langle \vec{N}, \vec{V} \rangle > 0 \text{ Front-facing} \end{aligned}$$

边缘案例$\langle \vec{N}, \vec{V} \rangle = 0$对应于我们正面看三角形边缘的情况，即相机和三角形是共面的。我们可以对这个三角形进行分类，而不会对结果产生太大影响，所以我们选择将其分类为反向的，以避免处理退化的三角形。

我们从哪里得到法线向量？原来有一个向量运算，*交叉乘积* $\vec{A} \times \vec{B}$，这需要两个向量 $\vec{A} \times \vec{B}$ 并生成垂直于两者的向量（有关此操作的定义，请参阅附录 A（线性代数））。换句话说，三角形表面上两个向量的

叉积是该三角形的法向量。我们可以通过彼此减去三角形的顶点来轻松获得三角形上的两个向量。所以计算三角形ABC法向量的方向,简单明了：

$$\vec{V_1} = B - A$$

$$\vec{V_2} = C - A$$

$$\vec{N} = \vec{V_1} \times \vec{V_2}$$

请注意，“法线向量的方向”与“法线向量”不同。这有两个原因。第一个是$|\vec{N}|$不一定等于1.这并不重要，因为规范化$\vec{N}$将是微不足道的，因为我们只关心的迹象$\langle \vec{N}, \vec{V} \rangle$.

第二个原因是，如果$\vec{N}$是ABC所以是$-\vec{N}$，在这种情况下，我们非常关心方向$\vec{N}$点，因为这正是我们可以将三角形分类为正面或背面的原因。

此外，两个向量的叉积不是可交换的：$\vec{V_1} \times \vec{V_2} = -(\vec{V_2} \times \vec{V_1})$.换句话说，此操作中向量的顺序很重要。既然我们定义了V_1和V_2就A_1 B_1和C，这意味着三角形中顶点的顺序很重要。我们不能处理三角形ABC和ACB不再是同一个三角形。
幸运的是，这些都不是随机的。给定交叉乘积操作的定义，我们定义的方式V_1和V_2，以及我们使用的坐标系（X向右，Y向上，Z向前），有一个非常简单的规则来确定法向量的方向：如果三角形的顶点ABC按顺时针顺序排列 当您从相机看它们时，上面计算的法线矢量将指向相机，也就是说，相机正在看三角形的正面。

我们只需要在手动设计 3D 模型时牢记这一规则，并在查看其正面时按顺时针顺序列出每个三角形的顶点，以便在我们以这种方式计算它们时它们的法线指向“外”。当然，到目前为止我们一直使用的示例立方体模型遵循此规则。

总结

在本章中，我们制作了渲染器，它以前只能渲染线框对象，能够渲染实体对象。这比仅仅使用代替 更复杂，因为我们需要靠近相机的三角形来掩盖远离相机的三角形。`DrawFilledTriangle``DrawWireframeTriangle`

我们探索的第一个想法是从后到前绘制三角形，但这有一些我们讨论过的缺点。更好的主意是在像素级别工作;这个想法使我们采用了一种称为深度缓冲的技术，无论我们绘制三角形的顺序如何，它都会产生正确的结果。

我们最终探索了一种可选但有价值的技术，它不会改变结果的正确性，但可以避免渲染场景大约一半的三角形：背面剔除。由于闭合对象的所有反向三角形都被其所有正面三角形覆盖，因此根本不需要绘制背面三角形。我们提出了一种简单的代数方法来确定三角形是正面还是背面。

现在我们可以渲染看起来很立体的对象，我们将在本书的其余部分专门讨论如何使这些对象看起来更逼真。