透视投影

到目前为止，我们已经学会了在画布上绘制 2D 三角形，给定其顶点的 2D 坐标。但是，本书的目标是渲染3D场景。因此，在本章中，我们将从 2D 三角形中休息一下，重点介绍如何将 3D 场景坐标转换为 2D 画布坐标。然后，我们将使用它在 3D 画布上绘制 2D 三角形。

基本假设

就像我们在第2章（基本光线追踪）开头所做的那样，我们将从定义*摄像机*开始。我们将使用与以前相同的约定：相机位于$O = (0, 0, 0)$看向$\vec{Z_+}$，其“向上”向量为$\vec{Y_+}$.我们还将定义一个大小的矩形*视口*V_w和V_h其边缘平行于X和Y，在远处d从相机。目标是在画布上绘制摄像机通过视口看到的任何内容。如果您需要复习这些概念，请参阅第 2 章（基本光线跟踪）。

考虑一个点P在镜头前的某个地方。我们有兴趣寻找P'，视口上摄像机通过该点看到的点P，如图9-1所示。

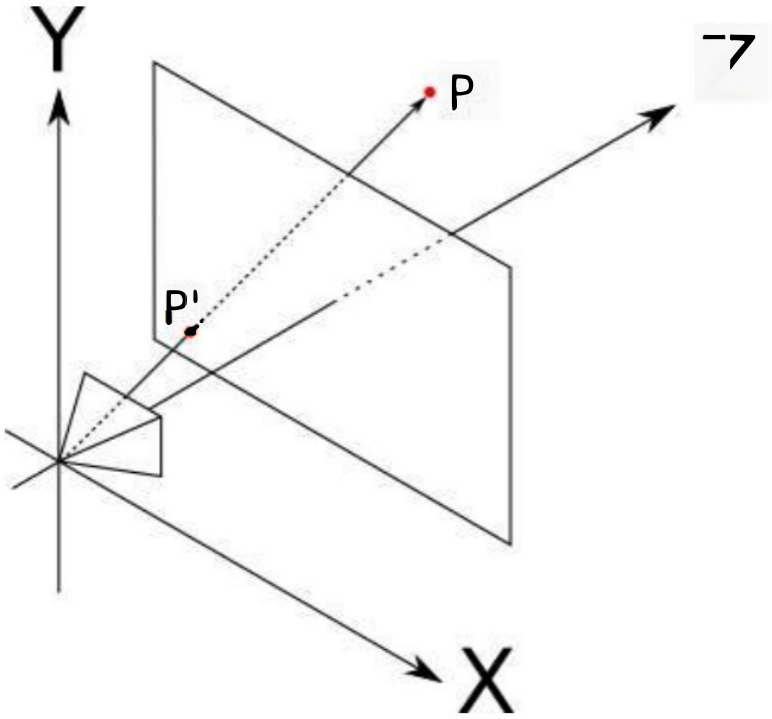

图 9-1：简单的透视投影设置。相机看到投影平面上的P到P'。

这与我们对光线追踪所做的相反。我们的光线追踪器从画布上的一个点开始，并确定它可以通过该点看到什么;在这里，我们从场景中的某个点开始，并希望确定它在视口

上的可见位置。

寻找 P'

找到P'，让我们从字面上从不同的角度看图 9-1 中所示的设置。图 9-2 显示了从“右侧”查看的设置图，就好像我们站在X轴：$\vec{Y}_+$点向上，$\vec{Z}_+$指向右侧，以及$\vec{X}_+$指着我们。

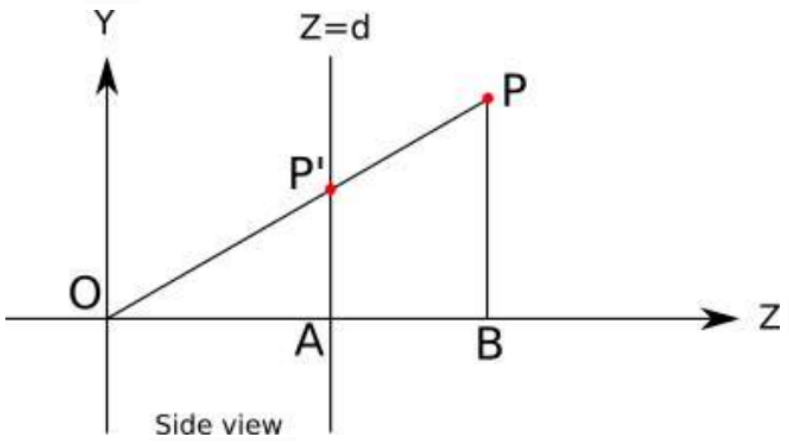

图 9-2：从右侧查看的透视投影设置

除了O,P和P'，此图还显示了要点A 和B，这有助于我们推理。

我们知道$P'_z = d$因为我们定义了P'成为视口上的一个点，我们知道视口嵌入在平面中$Z = d$.

我们还可以看到三角形$OP'A$ 和OPB相似，因为它们对应的边（$P'A$和PB,OP和OP'和OA和OB） 是平行的。这意味着它们侧面的比例是相同的;例如：

$$\frac{|P'A|}{|OA|} = \frac{|PB|}{|OB|}$$

由此，我们得到

$$|P'A| = \frac{|PB| \cdot |OA|}{|OB|}$$

该等式中每个段的（有符号）长度是我们知道或我们感兴趣的点的坐标：$|P'A| = P'_y$,$|PB| = P_y$, $OA| = P'_z = d$和 $OB| = P_z$.如果我们在等式中替换这些，我们得到

$$P'_y = \frac{P_y \cdot d}{P_z}$$

我们可以画一个类似的图，这次从上面查看设置：$\vec{Z}_+$点向上，$\vec{X}_+$指向右侧，以及$\vec{Y}_+$指向我们（图 9-3）。

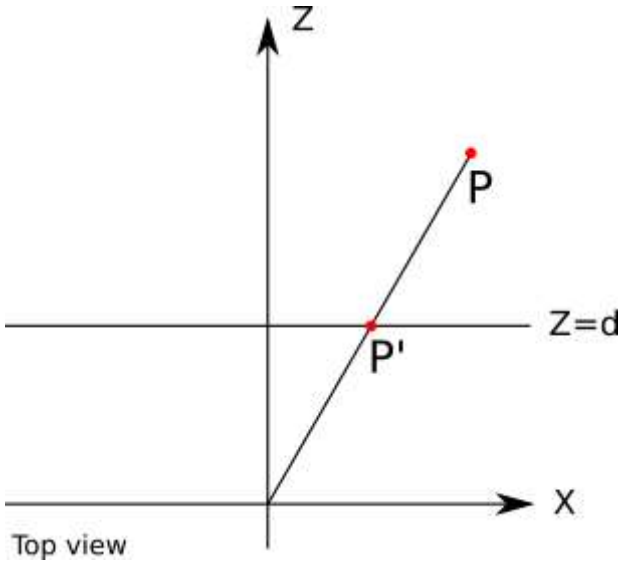

图 9-3：透视投影设置的俯视图

以相同的方式再次使用类似的三角形，我们可以推断出

$$P'_x = \frac{P_x \cdot d}{P_z}$$

我们现在拥有所有三个坐标P'.

投影方程

让我们把所有这些放在一起。给定一个点P在场景和标准摄像机和视口设置中，我们可以计算P在视口上，我们称之为P'如下：

$$P'_x = \frac{P_x \cdot d}{P_z}$$

$$P'_y = \frac{P_y \cdot d}{P_z}$$

$$P'_z = d$$

P'在视口上，但它仍然是 3D 空间中的一个点。我们如何在画布中获得相应的点？

我们可以立即放弃P'_z，因为每个投影点都在视口平面上。接下来我们需要转换P'_x和P'_y到画布坐标C_x和C_y.P'仍然是场景中的点，因此其坐标以场景单位表示。我们可以将它们除以视口的宽度和高度。这些也以场景单位表示，因此我们获得临时无单位的值。最后，我们将它们乘以画布的宽度和高度，以像素表示：

$$C_x = \frac{P'_x \cdot C_w}{V_w}$$

$$C_y = \frac{P'_y \cdot C_h}{V_h}$$

这种视口到画布的变换与我们在本书光线追踪部分使用的画布到视口变换完全相反。有了这个，我们终于可以从场景中的一个点变成屏幕上的像素了！

投影方程的性质

在我们继续之前，投影方程的一些有趣的属性值得讨论。

上面的方程式应该与我们看待现实世界事物的日常经验兼容。例如，物体离得越远，它看起来越小;事实上，如果我们增加P_z，我们得到较小的值P'_x和P'_y.

然而，当我们降低P_z太多了;对于负值P_z也就是说，当一个物体在相机*后面*时，物体仍然是投影的，只是颠倒的！而且，当然，当$P_z = 0$我们除以零，宇宙就会崩溃。我们需要找到一种方法来避免这些不愉快的情况;现在，我们将假设每个点都在镜头前，并在后面的章节中处理这个问题。

透视投影的另一个基本属性是它保留了点对齐：如果三个点在空间中对齐，则它们的投影将在视口上对齐。换句话说，直线始终投影为直线。这听起来可能太明显了，不值得一提，但请注意，例如，两条线之间的*角度*没有保留：在现实生活中，我们看到平行线在地平线上“汇聚”，例如在高速公路上行驶时。

直线总是投影为直线这一事实对我们来说非常方便：到目前为止，我们已经讨论了投影一个点，但是投影线段甚至三角形怎么样？由于此属性，两点之间的线段投影是两点投影之间的线段;三角形的投影是由其顶点的投影形成的三角形。

投影我们的第一个 3D 对象

这意味着我们可以继续绘制我们的第一个 3D 对象：立方体。我们定义其 8 个顶点的坐标，并在构成立方体边缘的 12 对顶点的投影之间绘制线段，如示例 9-1 所示：

```
ViewportToCanvas(x, y) {
  return (x * Cw/Vw, y * Ch/Vh);
}

ProjectVertex(v) {
  return ViewportToCanvas(v.x * d / v.z, v.y * d / v.z)
}

// The four "front" vertices
vAf = [-2, -0.5, 5]
vBf = [-2,  0.5, 5]
vCf = [-1,  0.5, 5]
vDf = [-1, -0.5, 5]

// The four "back" vertices
vAb = [-2, -0.5, 6]
vBb = [-2,  0.5, 6]
vCb = [-1,  0.5, 6]
vDb = [-1, -0.5, 6]

// The front face
DrawLine(ProjectVertex(vAf), ProjectVertex(vBf), BLUE);
DrawLine(ProjectVertex(vBf), ProjectVertex(vCf), BLUE);
DrawLine(ProjectVertex(vCf), ProjectVertex(vDf), BLUE);
DrawLine(ProjectVertex(vDf), ProjectVertex(vAf), BLUE);

// The back face
DrawLine(ProjectVertex(vAb), ProjectVertex(vBb), RED);
```

```
DrawLine(ProjectVertex(vBb), ProjectVertex(vCb), RED);
DrawLine(ProjectVertex(vCb), ProjectVertex(vDb), RED);
DrawLine(ProjectVertex(vDb), ProjectVertex(vAb), RED);

// The front-to-back edges
DrawLine(ProjectVertex(vAf), ProjectVertex(vAb), GREEN);
DrawLine(ProjectVertex(vBf), ProjectVertex(vBb), GREEN);
DrawLine(ProjectVertex(vCf), ProjectVertex(vCb), GREEN);
DrawLine(ProjectVertex(vDf), ProjectVertex(vDb), GREEN);
```

示例 9-1： 绘制立方体

我们得到如图 9-4 所示的内容。

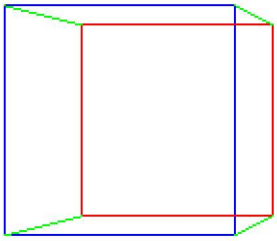

图 9-4：投影在 3D 画布上的第一个 2D 对象：立方体

源代码和现场演示>>

成功！我们已经成功地从物体的几何3D表示变成了从我们的合成相机看到的2D表示！

不过，我们的方法非常手工。它有很多限制。如果我们想渲染*两个*立方体怎么办？我们是否必须复制大部分代码？如果我们想渲染立方体以外的内容怎么办？如果我们想让用户从文件加载 3D 模型怎么办？我们显然需要一种更加数据驱动的方法来表示 3D 几何体。

总结

在本章中，我们开发了从场景中的 3D 点到画布上的 2D 点的数学方法。由于透视投影的属性，我们可以立即将其扩展到投影线段，然后扩展到 3D 对象。

但是，我们留下了两个重要问题没有解决。首先，示例 9-1 将透视投影逻辑与立方体的几何形状混合在一起;这种方法显然无法扩展。其次，由于透视投影方程的局限性，它无法处理相机后面的对象。我们将在接下来的两章中讨论这些问题。