

描述和渲染场景

在最后几章中，我们开发了算法，在给定 2D 坐标的情况下在画布上绘制 2D 三角形，并探讨了将场景中点的 3D 坐标转换为画布上点的 2D 坐标所需的数学运算。

在上一章的末尾，我们拼凑了一个程序，该程序使用两者在 3D 画布上渲染 2D 立方体。在本章中，我们将正式化和扩展这项工作，目标是渲染包含任意数量对象的整个场景。

表示多维数据集

让我们再考虑一下如何表示和操作多维数据集，这次的目标是找到一种更通用的方法。立方体的边缘长 2 个单位，平行于坐标轴，以原点为中心，如图 10-1 所示。

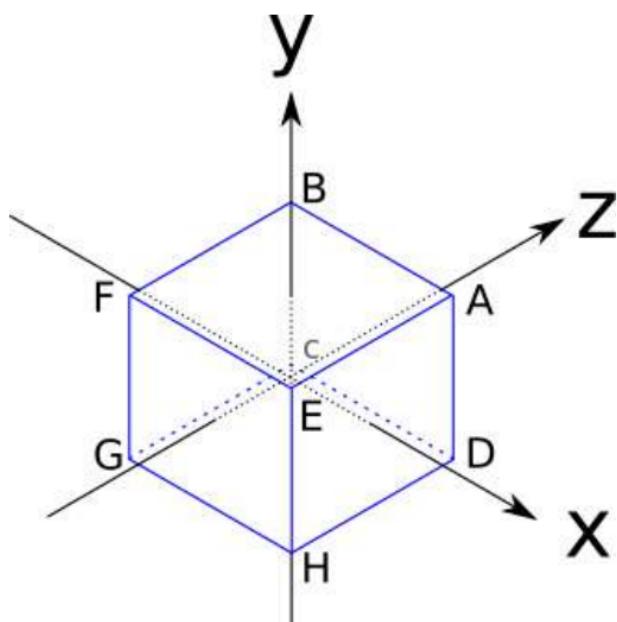


图 10-1：我们的标准立方体

这些是其顶点的坐标：

$$A = (1, 1, 1)$$

$$B = (-1, 1, 1)$$

$$C = (-1, -1, 1)$$

$$D = (1, -1, 1)$$

$$E = (1, 1, -1)$$

$$F = (-1, 1, -1)$$

$$G = (-1, -1, -1)$$

$$H = (1, -1, -1)$$

立方体的边是正方形的，但我们开发的算法适用于三角形。我们首先选择三角形的原因之一是，任何其他多边形，包括正方形，都可以分解为三角形。因此，我们将使用两个三角形表示立方体的每个正方形边。

但是，我们不能取立方体的任何三个顶点并期望它们描述其表面上的三角形（例如，ADG 位于立方体内部）。这意味着顶点坐标本身并不能完全描述立方体：我们还需要知道哪一组三个顶点描述了构成其边的三角形。

以下是我们立方体的三角形可能列表：

```
A, B, C
A, C, D
E, A, D
E, D, H
F, E, H
F, H, G
B, F, G
B, G, C
E, I, B
E, B, A
C, G, H
C, H, D
```

这表明我们可以用来表示由三角形组成的任何对象的通用结构：一个列表，保存每个顶点的坐标和一个列表，指定哪些三个顶点集描述对象表面上的三角形。`Vertices``Triangles`

列表中的每个条目除了构成它的顶点之外，还可能包含其他信息；例如，这将是指定每个三角形颜色的理想位置。`Triangles`

由于存储此信息的最自然方法是在两个列表中，因此我们将使用列表索引来引用顶点列表中的顶点。所以我们的立方体将表示如下：

```
Vertices
0 = ( 1, 1, 1)
1 = (-1, 1, 1)
2 = (-1, -1, 1)
3 = ( 1, -1, 1)
4 = ( 1, 1, -1)
5 = (-1, 1, -1)
6 = (-1, -1, -1)
7 = ( 1, -1, -1)

Triangles
0 = 0, 1, 2, red
1 = 0, 2, 3, red
2 = 4, 0, 3, green
3 = 4, 3, 7, green
4 = 5, 4, 7, blue
5 = 5, 7, 6, blue
6 = 1, 5, 6, yellow
7 = 1, 6, 2, yellow
8 = 4, 5, 1, purple
9 = 4, 1, 0, purple
10 = 2, 6, 7, cyan
11 = 2, 7, 3, cyan
```

用这种表示渲染对象非常简单：我们首先投影每个顶点，将它们存储在临时投影顶点列表中（因为每个顶点平均使用四次，这避免了大量的重复工作）；然后我们浏览三角形列表，渲染每个单独的三角形。第一个近似值如下所示：

```
RenderObject(vertices, triangles) {
    projected = []
```

```

for V in vertices {
    projected.append(ProjectVertex(V))
}
for T in triangles {
    RenderTriangle(T, projected)
}
}

RenderTriangle(triangle, projected) {
    DrawWireframeTriangle(projected[triangle.v[0]],
                          projected[triangle.v[1]],
                          projected[triangle.v[2]],
                          triangle.color)
}

```

示例 10-1：一种渲染任何由三角形组成的对象的算法

我们可以继续将其直接应用于上面定义的立方体，但结果看起来并不好。这是因为它的一些顶点位于相机后面，正如我们在上一章中所讨论的，这是奇怪事物的秘诀。如果你看一下顶点坐标和图10-1，你会注意到坐标原点，我们相机的位置，在立方体内部。

要解决此问题，我们只需移动立方体。为此，我们需要将立方体的每个顶点沿同一方向移动。我们称这个方向为这个方向 \vec{T} ，表示“翻译”。我们将立方体向前平移 7 个单位，以确保它完全在相机前方。我们还将把它向左翻译 1.5 个单位，让它看起来更有趣。由于“前进”是方向 \vec{Z}_+ 而“左”是 \vec{X}_- ，平移向量很简单

$$\vec{T} = \begin{pmatrix} -1.5 \\ 0 \\ 7 \end{pmatrix}$$

计算已翻译版本的步骤 V' 每个顶点 V 在立方体中，我们只需要向其添加平移向量：

$$V' = V + \vec{T}$$

此时，我们可以获取立方体，平移每个顶点，然后应用示例 10-1 中的算法来获取我们的第一个 3D 立方体（图 10-2）。

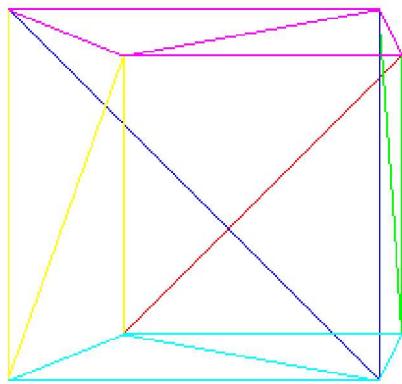


图 10-2：我们的立方体，在相机前平移，用线框三角形渲染

[源代码和现场演示>>](#)

模型和实例

如果我们想渲染两个立方体怎么办？一种天真的方法是创建一组描述第二个立方体的新顶点和三角形。这将起作用，但会浪费大量内存。如果我们想渲染 100 万个立方体怎么办？

更好的方法是从 **模型** 和 **实例** 的角度来思考。模型是一组顶点和三角形，以通用方式描述某个对象（想想“立方体有八个顶点和六个边”）。另一方面，模型的实例描述了该模型在场景中的具体出现（想想“在 (0, 0, 5) 处有一个立方体”）。

我们如何在实践中应用这一想法？我们可以对场景中的每个唯一对象进行单一描述，然后通过指定它们的坐标来放置它的多个副本。非正式地说，这就像是说，“这就是立方体的样子，这里，这里和那里都有立方体。”

这是我们如何使用这种方法描述场景的粗略近似值：

```
model {
    name = cube
    vertices {
        ...
    }
    triangles {
```

```

    ...
}

instance {
    model = cube
    position = (-1.5, 0, 7)
}

instance {
    model = cube
    position = (1.25, 2, 7.5)
}

```

为了呈现这一点，我们只需浏览实例列表;对于每个实例，我们复制模型的顶点，根据实例的位置转换它们，然后像以前一样渲染它们 (示例 10-2)：

```

RenderScene() {
    for I in scene.instances {
        RenderInstance(I);
    }
}

RenderInstance(instance) {
    projected = []
    model = instance.model
    for V in model.vertices {
        V' = V + instance.position
        projected.append(ProjectVertex(V'))
    }
    for T in model.triangles {
        RenderTriangle(T, projected)
    }
}

```

示例 10-2：一种渲染场景的算法，该场景可以包含多个对象的多个实例，每个实例位于不同的位置

如果我们希望它按预期工作，则应在对对象“有意义”的坐标系中定义模型上顶点的坐标;我们将此坐标系模型空间称为模型空间。例如，我们定义立方体，使其中心为 (0, 0, 0);这意味着当我们说“位于 (1, 2, 3) 的立方体”时，我们的意思是“以 (1, 2, 3) 为中心的立方体”。

将实例平移应用于模型空间中定义的顶点后，转换后的顶点现在以场景的坐标系表示;我们将此坐标系称为世界空间。

没有硬性规定来定义模型空间;这取决于您的应用程序的需求。例如，如果您有一个人的模型，那么将坐标系的原点放在他们的脚下可能是明智的。

图 10-3 显示了包含两个立方体实例的简单场景。

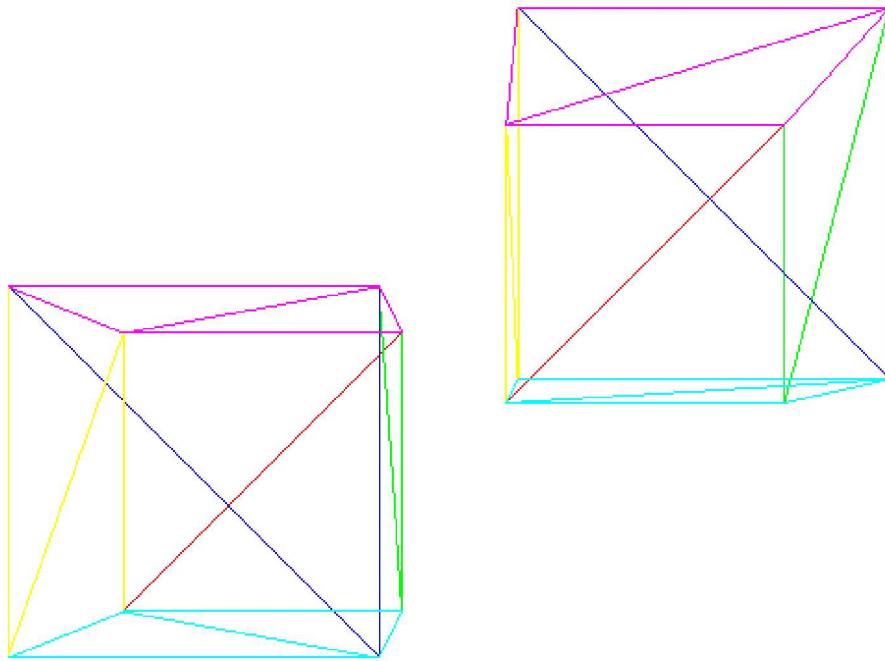


图 10-3：具有同一立方体模型的两个实例的场景，放置在不同的位置

[源代码和现场演示>>](#)

模型转换

我们上面描述的场景定义并没有给我们很大的灵活性。由于我们只能指定立方体的位置，因此我们可以实例化任意数量的立方体，但它们都将朝向同一方向。通常，我们希望对实例有更多的控制：我们还希望指定它们的方向和可能的规模。

从概念上讲，我们可以使用以下三个元素定义模型转换：比例因子、模型空间中围绕原点的旋转以及到场景中特定点的平移：

```
instance {
    model = cube
    transform {
        scale = 1.5
        rotation = <45 degrees around the Y axis>
        translation = (1, 2, 3)
    }
}
```

我们可以扩展示例 10-2 中的算法以适应新的转换。但是，我们应用转换的顺序很重要；特别是，翻译必须最后完成。这是因为大多数时候我们都希望在模型空间中围绕它们的原点旋转和缩放实例，因此我们需要在它们转换为世界空间之前执行此操作。

要了解结果的差异, 请查看图 10-4, 其中显示了 45° 围绕原点旋转, 然后沿 Z 轴平移。

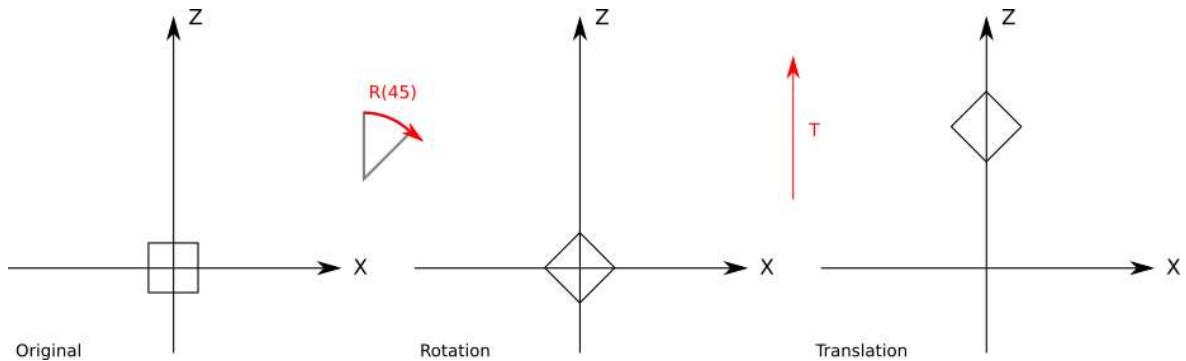


图 10-4: 应用旋转, 然后平移

图 10-5 显示了旋转前应用的平移

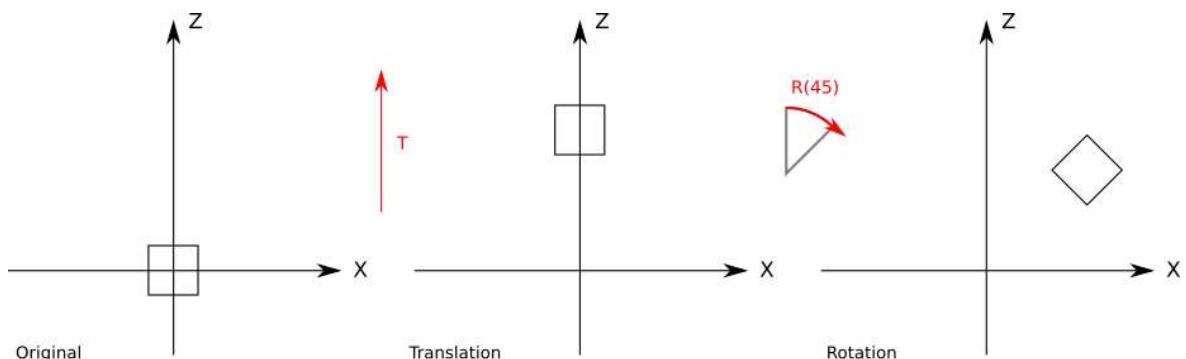


图 10-5: 应用平移然后旋转

严格来说, 给定一个旋转后进行平移, 我们可以找到一个平移, 然后旋转 (也许不是围绕原点), 达到相同的结果。但是, 使用第一种形式表达这种转换要自然得多。

我们可以编写一个支持缩放、旋转和位置的新版本: RenderInstance

```
RenderInstance(instance) {
    projected = []
    model = instance.model
    for V in model.vertices {
        V' = ApplyTransform(V, instance.transform)
        projected.append(ProjectVertex(V'))
    }
    for T in model.triangles {
        RenderTriangle(T, projected)
    }
}
```

示例 10-3: 一种渲染场景的算法, 该场景可以包含多个对象的多个实例, 每个实例都有不同的变换

该方法如下所示: ApplyTransform

```
ApplyTransform(vertex, transform) {
    scaled = Scale(vertex, transform.scale)
    rotated = Rotate(scaled, transform.rotation)
    translated = Translate(rotated, transform.translation)
    return translated
}
```

示例 10-4: 一个以正确顺序将变换应用于顶点的函数

相机变换

前面的部分探讨了如何在场景中的不同点定位模型实例。在本节中，我们将探讨如何在场景中移动和旋转摄像机。

想象一下，您是一台漂浮在完全空的坐标系中间的相机。突然，一个红色立方体正好出现在您面前（图 10-6）。

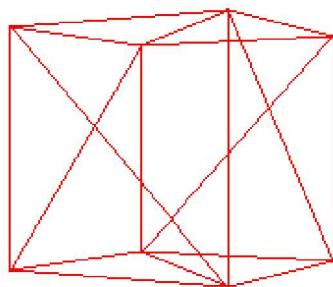


图 10-6：一个红色立方体出现在相机前。

一秒钟后，立方体朝你移动 1 个单位（图 10-7）。

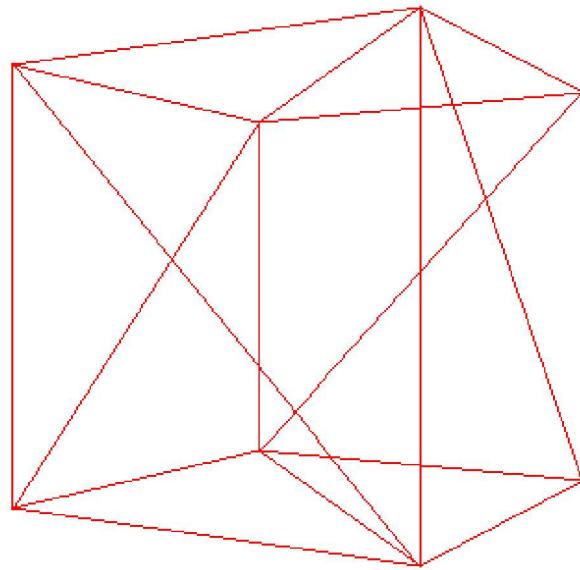


图 10-7：红色立方体向相机移动。还是吗？

但是立方体真的向你移动了 1 个单位吗？还是您将 1 个单位移向立方体？由于根本没有参考点，并且坐标系不可见，因此无法仅通过查看所看到的内容来判断，因为立方体和相机的相对位置在两种情况下都是相同的（图 10-8）。

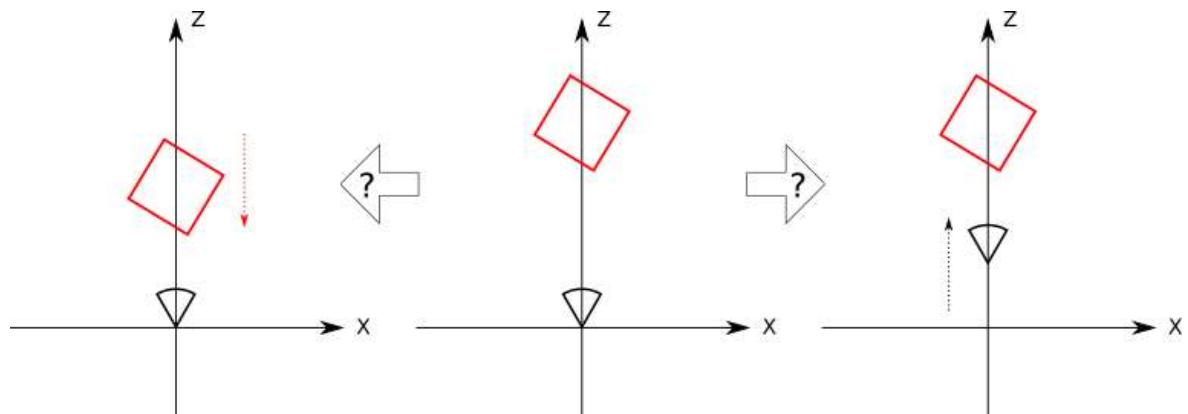


图 10-8：如果没有坐标系，我们无法判断移动的是物体还是相机。

现在立方体围绕你旋转 45° 顺时针。还是吗？也许是您旋转了 45° 反时针方向的？同样，没有办法判断（图 10-9）。

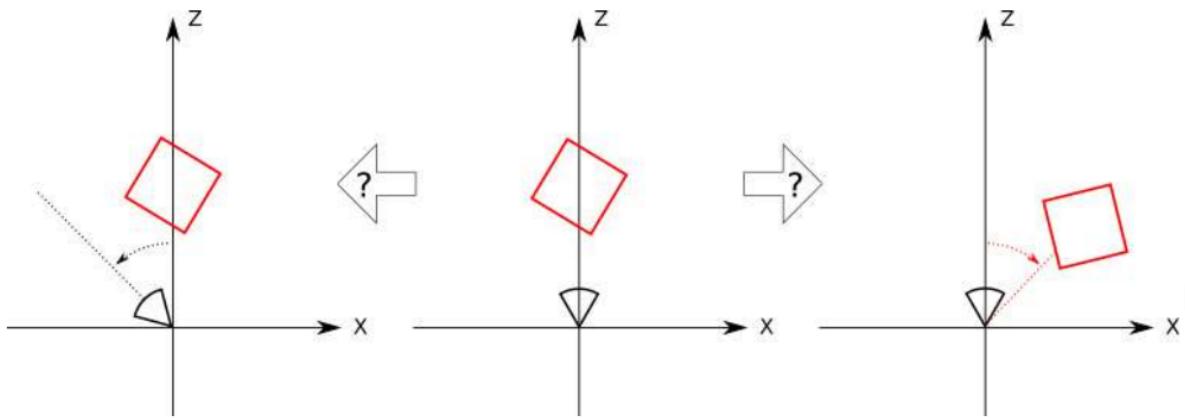


图 10-9：如果没有坐标系，我们无法判断旋转的是对象还是相机。

这个思想实验表明，在固定场景中移动摄像机与在旋转和平移周围场景时保持摄像机固定之间没有区别！

这种明显以自我为中心的宇宙愿景的好处是，通过将相机固定在原点并指向 Z_+ ，我们可以使用上一章中推导的投影方程，而无需进行任何修改。相机的坐标系称为相机空间。

假设相机还附加了一个转换，包括平移和旋转。为了从摄像机的角度渲染场景，我们需要对场景的每个顶点应用相反的变换：

$$V_{translated} = V_{scene} - camera.translation$$

$$V_{cam_space} = \text{inverse}(camera.rotation) \cdot V_{translated}$$

$$V_{projected} = \text{perspective_projection}(V_{cam_space})$$

请注意，我们使用旋转矩阵表示旋转。有关此内容的更多详细信息，请参阅附录 [ch: linear_algebra_appendix]。

变换矩阵

现在我们可以在场景中移动摄像机和实例，让我们退后一步，考虑顶点发生的一切。 V_{model} 模型空间中，直到它投影到画布点(Cx, cy)

我们首先应用模型转换从模型空间到世界空间：

$$V_{model_scaled} = instance.scale \cdot V_{model}$$

$$V_{model_rotated} = instance.rotation \cdot V_{model_scaled}$$

$$V_{world} = V_{model_rotated} + instance.translation$$

然后我们应用相机变换从世界空间转到相机空间：

$$V_{translated} = V_{world} - camera.translation$$

$$V_{camera} = \text{inverse}(camera.rotation) \cdot V_{translated}$$

接下来，我们应用透视方程来获取视口坐标：

$$v_x = \frac{V_{camera}x \cdot d}{V_{camera}z}$$

$$v_y = \frac{V_{camera}y \cdot d}{V_{camera}z}$$

最后，我们将视口坐标映射到画布坐标：

$$c_x = \frac{v_x \cdot c_w}{v_w}$$

$$c_y = \frac{v_y \cdot c_h}{v_h}$$

如您所见，每个顶点都需要大量的计算和大量的中间值。如果我们能将所有这些简化为更紧凑、更高效的形式，那不是很好吗？

让我们将转换表示为采用顶点并返回转换顶点的函数。让 C_T 和 C_R 是相机平移和旋转； I_R , I_S , 和 I_T 实例旋转、缩放和平移； P 透视投影；和 M 视口到画布的映射。如果 V 是原始顶点，并且 V' 是画布上的点，我们可以像这样表达上面的所有方程：

$$V' = M(P(C_R^{-1}(C_T^{-1}(I_T(I_R(I_S(V)))))))$$

理想情况下，我们想要一个单一的转换 F 它执行一系列原始转换所做的一切，但这有一个更简单的表达式：

$$F = M \cdot P \cdot C_R^{-1} \cdot C_T^{-1} \cdot I_T \cdot I_R \cdot I_S$$

$$V' = F(V)$$

找到一种简单的表示方式 F 不是小事。我们的主要障碍是我们以不同的方式表达每个变换：我们将平移表示为点和向量的总和，旋转表示为矩阵和点的乘法，缩放表示为实数和点的乘法，透视投影表示为实数乘法和除法。但是，如果我们能以相同的方式表达所有变换，并且如果这种方式具有组合变换的机制，我们就会得到我们想要的简单变换。

均匀坐标

考虑表达式 $A = (1, 2, 3)$ A 表示 3D 点还是 3D 向量？如果我们不知道 A 的上下文是谁，没有办法知道 A 被谁使用。

但是，让我们添加第四个值，称为 w ，将 A 标记为点或向量。如果 $w = 0$ ，它是一个向量；如果 $w = 1$ ，这是一个点。所以当点 A 明确表示为 $A = (1, 2, 3, 1)$ 并且矢量 $A = (1, 2, 3, 0)$

由于点和向量共享相同的表示形式，因此这些四分量坐标称为齐次坐标。齐次坐标具有更深入、更复杂的几何解释，但这超出了本书的范围；在这里，我们只是将它们用作方便的工具。

操作以齐次坐标表示的点和向量与其几何解释兼容。例如，减去两点会生成一个向量：

$$(8, 4, 2, 1) - (3, 2, 1, 1) = (5, 2, 1, 0)$$

添加两个向量会产生另一个向量：

$$(0, 0, 1, 0) + (1, 0, 0, 0) = (1, 0, 1, 0)$$

同样，很容易看出，添加一个点和一个向量会产生一个点，将一个向量乘以标量会产生一个向量，依此类推，正如我们所期望的那样。

那么用一个坐标做什么 w 值 0 或 1 代表？它们还表示点。事实上，3D 中的任何点在齐次坐标中都有无限数量的表示。重要的是坐标和 w 价值。例如 $(1, 2, 3, 1)$ 和 $(2, 4, 6, 2)$ 的点，如 $(-3, -6, -9, -3)$

在所有这些表示中，我们称之为 $w = 1$ 点在齐次坐标中的规范表示。将任何其他表示转换为其规范表示或笛卡尔坐标是微不足道的：

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \end{pmatrix}$$

因此，我们可以将笛卡尔坐标转换为齐次坐标，然后再转换回笛卡尔坐标。但是，这如何帮助我们找到所有转换的单一表示形式呢？

均匀旋转矩阵

让我们从旋转矩阵开始。转换 3×3 笛卡尔坐标中的旋转矩阵为 4×4 齐次坐标中的旋转矩阵是微不足道的；自从 w 点的坐标不应该改变，我们在右边加一列，在底部加一行，用零填充它们，然后放置一个 1 以保持 w ：

$$\begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \rightarrow \begin{pmatrix} A & B & C & 0 \\ D & E & F & 0 \\ G & H & I & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

均匀尺度矩阵

缩放矩阵在齐次坐标中也是微不足道的，它的构造方式与旋转矩阵相同：

$$\begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \cdot S_x \\ y \cdot S_y \\ z \cdot S_z \end{pmatrix} \rightarrow \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \cdot S_x \\ y \cdot S_y \\ z \cdot S_z \\ 1 \end{pmatrix}$$

同构翻译矩阵

旋转和缩放矩阵很容易；它们已经表示为笛卡尔坐标中的矩阵乘法，我们只需要添加一个 1 以保留 w 坐标。但是，我们能用笛卡尔坐标表示为加法的翻译做什么呢？

我们正在寻找一个 4×4 矩阵使得

$$\begin{pmatrix} T_x \\ T_y \\ T_z \\ 0 \end{pmatrix} + \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

让我们专注于获得 $x + T_x$ 第一。此值是将矩阵的第一行与点相乘的结果，即

$$(A \ B \ C \ D) \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = x + T_x$$

如果我们扩展向量乘法，我们得到

$$Ax + By + Cz + D = x + T_x$$

从这里我们可以推断出 $A = 1, B = C = 0$ 和 $D = T_x$.

遵循其余坐标的类似推理，我们得出以下用于翻译的矩阵表达式：

$$\begin{pmatrix} T_x \\ T_y \\ T_z \\ 0 \end{pmatrix} + \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

均匀投影矩阵

和乘法很容易表示为矩阵和向量的乘法，因为它们毕竟涉及和乘法。但是透视投影方程的除法为 z 我们该如何表达呢？

你可能会想，除以 z 与乘以相同 $1/z$ ，您可能希望通过放置来解决此问题 $1/z$ 在矩阵中。然而，其中 z 我们会把坐标放在那里吗？我们希望这个投影矩阵适用于每个输入点，因此对 z 任何一点的坐标都不会给我们想要的东西。

幸运的是，齐次坐标确实有一个除法实例：除以 w 转换回笛卡尔坐标时的坐标。如果我们能设法使 z 原始点的坐标显示为 w “投影” 点的坐标，我们将得到投影 x 和 y 一旦我们将点转换回笛卡尔坐标：

$$\begin{pmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \cdot d \\ y \cdot d \\ z \cdot d \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} \frac{x \cdot d}{z} \\ \frac{y \cdot d}{z} \\ \frac{z \cdot d}{z} \\ 1 \end{pmatrix}$$

请注意，此矩阵是 3×4 ；它可以乘以四元素向量（齐次坐标中转换的 3D 点），它将产生一个三元素向量（齐次坐标中的投影 2D 点），然后通过除以转换为 2D 笛卡尔坐标 w 。这给了我们确切的值 x' 和 y' 我们一直在寻找。这里缺少的元素是 z' ，我们知道等于 d 根据定义。

应用我们用来推导平移矩阵的相同推理，我们可以将透视投影表示如下：

$$\begin{pmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \cdot d \\ y \cdot d \\ z \cdot d \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} \frac{x \cdot d}{z} \\ \frac{y \cdot d}{z} \\ \frac{z \cdot d}{z} \\ 1 \end{pmatrix}$$

均匀的视口到画布矩阵

最后一步是将视口上的投影点映射到画布。这只是一个 2D 缩放转换 $S_x = \frac{c_w}{v_w}$ 和 $S_y = \frac{c_h}{v_h}$ 。因此，该矩阵是

$$\begin{pmatrix} \frac{cw}{vw} & 0 & 0 \\ 0 & \frac{ch}{vh} & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \frac{x \cdot cw}{vw} \\ \frac{y \cdot ch}{vh} \\ z \end{pmatrix}$$

事实上，很容易将其与投影矩阵相结合，以获得一个简单的 3D 到画布矩阵：

$$\begin{pmatrix} \frac{d \cdot cw}{vw} & 0 & 0 & 0 \\ 0 & \frac{d \cdot ch}{vh} & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{x \cdot d \cdot cw}{vw} \\ \frac{y \cdot d \cdot ch}{vh} \\ z \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} (\frac{x \cdot d}{z})(\frac{cw}{vw}) \\ (\frac{y \cdot d}{z})(\frac{ch}{vh}) \\ z \\ 1 \end{pmatrix}$$

重新审视转换矩阵

完成所有这些工作后，我们可以表达转换模型顶点所需的每个转换 V 转换为画布像素 V' 作为矩阵。此外，我们可以通过乘以相应的矩阵来组合这些变换。因此，我们可以将整个变换序列表示为单个矩阵：

$$F = M \cdot P \cdot C_R^{-1} \cdot C_T^{-1} \cdot I_T \cdot I_R \cdot I_S$$

现在转换顶点只需计算以下逐点矩阵乘法：

$$V' = F \cdot V$$

此外，我们可以将变换分解为三个部分：

$$\begin{aligned} M_{Projection} &= M \cdot P \\ M_{Camera} &= C_R^{-1} \cdot C_T^{-1} \\ M_{Model} &= I_T \cdot I_R \cdot I_S \\ M &= M_{Projection} \cdot M_{Camera} \cdot M_{Model} \end{aligned}$$

这些矩阵不需要为每个顶点从头开始计算（毕竟这是使用矩阵的意义所在）。因为矩阵乘法是关联的，所以我们可以重用表达式中不变的部分。

$M_{Projection}$ 应该很少改变；它仅取决于视口的大小和画布的大小。例如，当应用程序从窗口变为全屏时，画布的大小会发生变化。仅当摄像机的视野发生变化时，视口的大小才会更改；这种情况并不经常发生。

M_{Camera} 可能会更改每一帧；这取决于相机的位置和方向，因此如果相机正在移动或转动，则需要重新计算。但是，一旦计算出来，它对于帧中绘制的每个对象都保持不变，因此每帧最多计算一次。

M_{Model} 对于场景中的每个实例，将有所不同；但是，对于不移动的实例（例如，树木和建筑物），它将随着时间的推移保持不变，因此可以计算一次并存储在场景本身中。对于确实移动的对象（例如，赛车游戏中的汽车），需要在它们每次移动时（可能是每一帧）进行计算。

非常高级别的场景渲染伪代码类似于示例 10-5。

```
RenderModel(model, transform) {
```

```

for V in model.vertices {
    projected.append(ProjectVertex(transform * V))
}
for T in model.triangles {
    RenderTriangle(T, projected)
}
}

RenderScene() {
    M_camera = MakeCameraMatrix(camera.position, camera.orientation)

    for I in scene.instances {
        M = M_camera * I.transform
        RenderModel(I.model, M)
    }
}
}

```

示例 10-5：一种使用变换矩阵渲染场景的算法

现在，我们可以绘制一个包含不同模型的多个实例的场景，可以四处移动和旋转，并且可以在整个场景中移动摄像机。图 10-10 显示了我们的立方体模型的两个实例，每个实例都有不同的转换（包括平移和旋转），从平移和旋转的相机渲染。

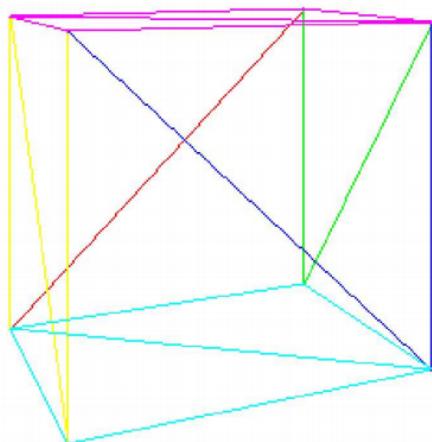
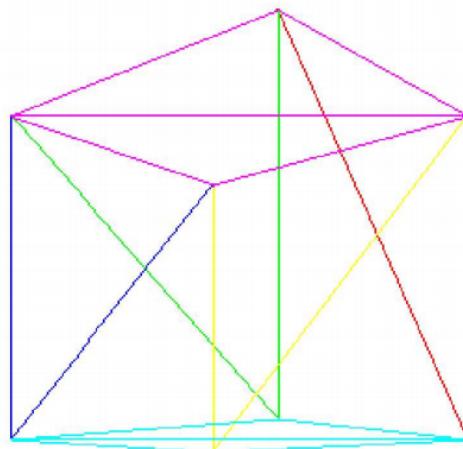


图 10-10：具有同一立方体模型的两个实例、具有不同实例转换和转换相机的场景

[源代码和现场演示>>](#)

总结

我们在本章中涵盖了很多内容。我们首先探索了如何表示由三角形组成的模型。然后我们想出了如何将上一章推导的透视投影方程应用于整个模型，这样我们就可以从抽象的3D模型到它在屏幕上的表现。

接下来，我们开发了一种方法，可以在场景中拥有同一模型的多个实例，而无需模型本身的多个副本。然后，我们找到了如何消除我们迄今为止一直在使用的限制之一：我们的相机不再需要固定在坐标系的原点或指向坐标系的原点 \vec{Z}^+

最后，我们探索了如何将需要应用于顶点的所有变换表示为齐次坐标中的矩阵乘法，这使我们能够通过将许多连续变换压缩为三个矩阵来减少渲染场景所需的计算：一个用于透视投影和视口到画布映射，一个用于实例变换，一个用于相机变换。

这为我们在场景中的表现提供了很大的灵活性，它还允许我们在场景中移动摄像机。但我们仍然有两个重要的限制。首先，移动相机意味着我们最终可能会看到它后面的物体，这会导致各种问题。其次，渲染看起来不是那么好：它仍然是线框图像。

请注意，出于实际原因，我们不会在本书的其余部分使用完整的投影矩阵。相反，我们将分别使用模型和相机变换，然后将其结果转换回笛卡尔坐标，如下所示：

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

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

这让我们可以在 3D 中执行更多操作，这些操作在投影点之前无法表示为矩阵变换。

在下一章中，我们将处理不应该可见的对象，然后我们将用本书的其余部分使渲染的对象看起来更好。