

纹理

我们的光栅器可以渲染立方体或球体等对象。但我们通常不想渲染抽象的几何对象，如立方体和球体；相反，我们想要渲染现实世界的对象，如板条箱和行星或骰子和弹珠。在本章中，我们将了解如何使用纹理向对象表面添加视觉细节。

画板条箱

假设我们希望我们的场景有一个木箱。我们如何将立方体变成木箱？一种选择是添加大量三角形来复制木材的纹理、钉子的头部等。这将起作用，但它会给场景增加很多几何复杂性，从而导致性能大打折扣。

另一种选择是伪造细节：我们不是修改物体的几何形状，而是在上面“画”看起来像木头的东西。除非你近距离观察板条箱，否则你不会注意到差异，而且计算成本明显低于添加大量几何细节。

请注意，这两个选项并非不兼容：您可以在添加几何体和在该几何体上绘制之间选择适当的平衡，以实现所需的图像质量和性能。由于我们知道如何处理几何，我们将探索第二个选项。

首先，我们需要一个图像来绘制在我们的三角形上；在这种情况下，我们称此图像为纹理。图 14-1 显示了木箱纹理。

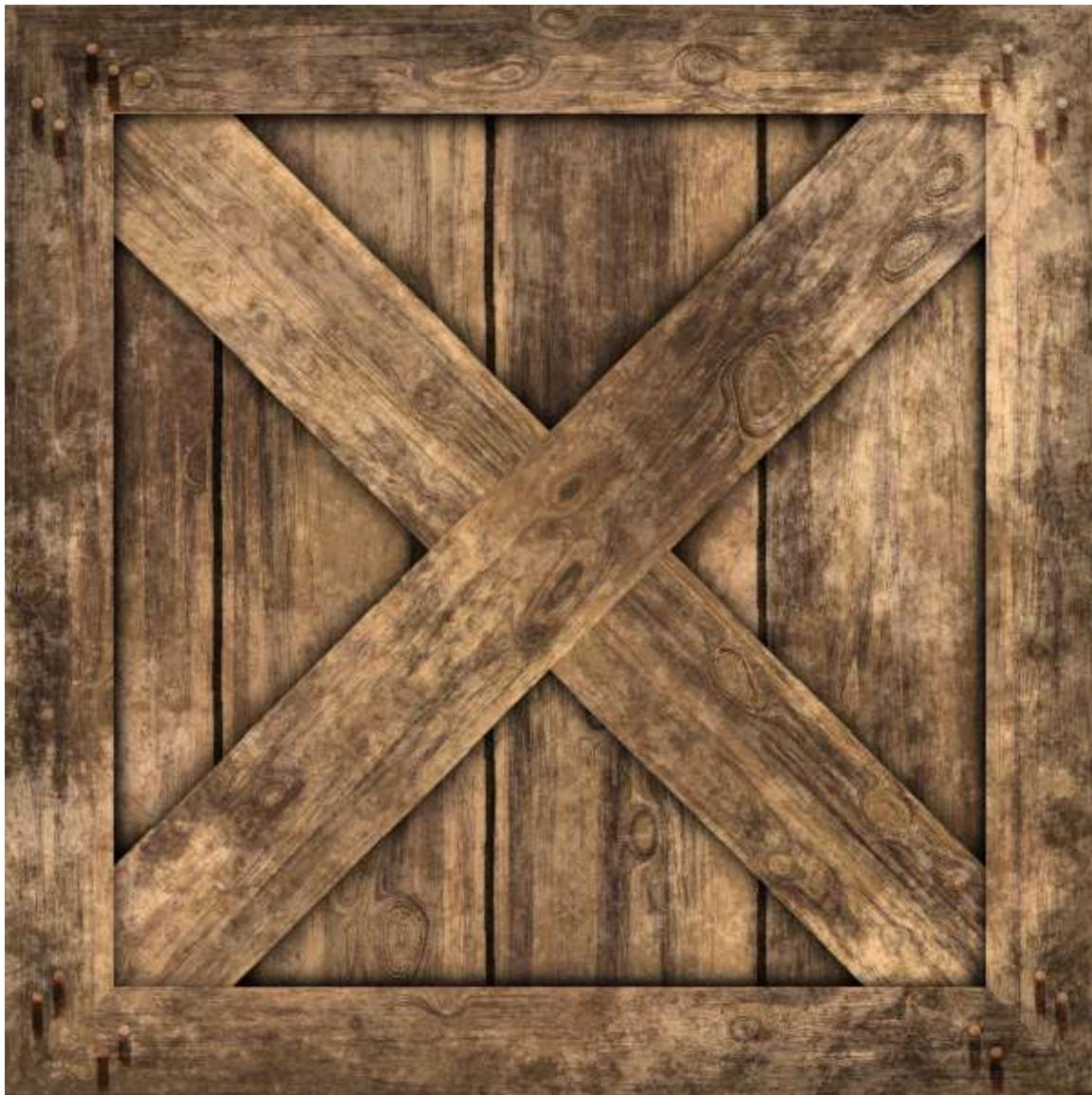


图 14-1: 木箱纹理 (通过过滤器锻造 — 署名 2.0 通用 (CC BY 2.0) 许可证)

接下来，我们需要指定如何将此纹理应用于模型。我们可以通过指定纹理的哪些点应该在三角形的每个顶点上来定义每个三角形的映射（图 14-2）。

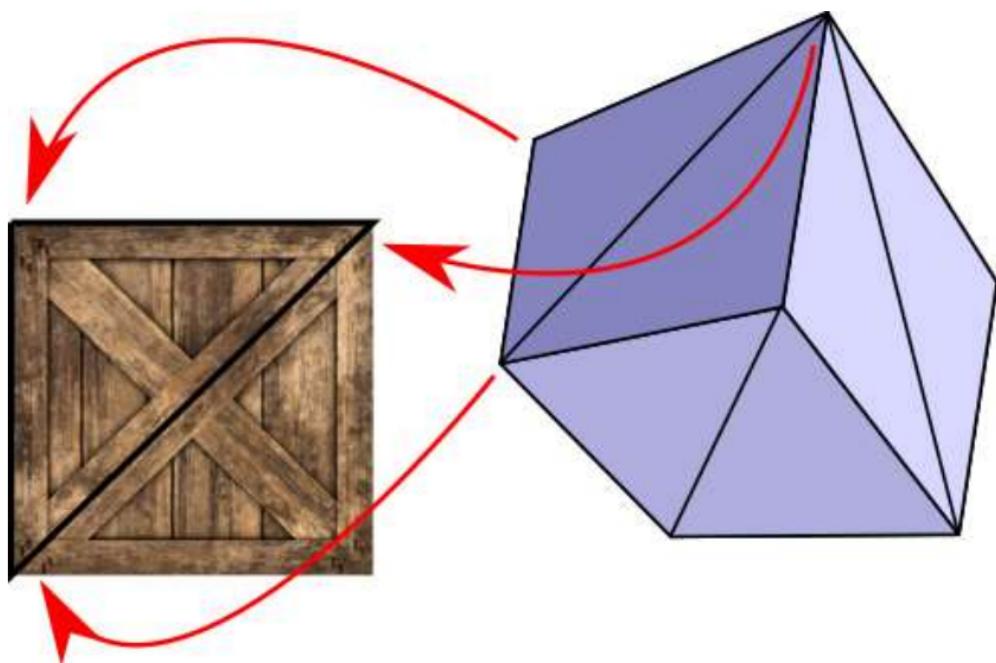


图 14-2：我们将纹理中的一个点与三角形的每个顶点相关联。

要定义此映射，我们需要一个坐标系来引用纹理中的点。请记住，纹理只是一个图像，表示为像素的矩形数组。我们可以使用 x 和 y 坐标并讨论纹理中的像素，但我们已经将这些名称用于画布。因此，我们使用 u 和 v 对于纹理坐标，我们称纹理的像素纹素（*texture elements*的收缩）。

我们将修复此的起源 (u, v) 纹理左上角的坐标系。我们还将声明 u 和 v 是范围内的实数 $[0, 1]$ 而不考虑纹理的实际纹素尺寸。由于几个原因，这非常方便。例如，我们可能希望使用较低或更高分辨率的纹理，具体取决于我们可用的 RAM 量；因为我们不受实际像素尺寸的约束，所以我们可以更改分辨率而无需修改模型本身。我们可以乘以 u 和 v 分别通过纹理宽度和高度得到实际纹素指数 tx 和 ty

纹理映射的基本思想很简单：我们计算 (u, v) 三角形每个像素的坐标，从纹理中获取适当的纹素，并用该颜色绘制像素。但该模型仅指定 u 和 v 三角形三个顶点的坐标，我们需要它们用于每个像素……

到现在为止，您可能可以看到这是怎么回事。是的，这是我们的好朋友线性插值。我们可以使用属性映射来插值 u 和 v 穿过三角形的表面，给我们 (u, v) 在每个像素处。由此我们可以计算 (tx, ty) 获取纹素，应用着色，然后用生成的颜色绘制像素。您可以在图 14-3 中看到执行此操作的结果。



图 14-3：纹理在应用于对象时看起来变形。

结果有点平淡无奇。板条箱的外观形状看起来不错，但如果你仔细观察对角线的木板，你会发现它们看起来变形了，好像以奇怪的方式弯曲。出了什么问题？

精确纹理映射

与第 12 章（[隐藏表面去除](#)）一样，我们做了一个隐含的假设，结果证明是不正确的：即， u 和 v 在屏幕上呈线性变化。显然情况并非如此。考虑一个很长的走廊的墙壁，上面画着交替的垂直黑白条纹。随着墙壁向远处退去，垂直条纹应该看起来越来越薄。如果我们使 u 坐标随线性变化 x' ，我们得到不正确的结果，如图 14-4 所示。

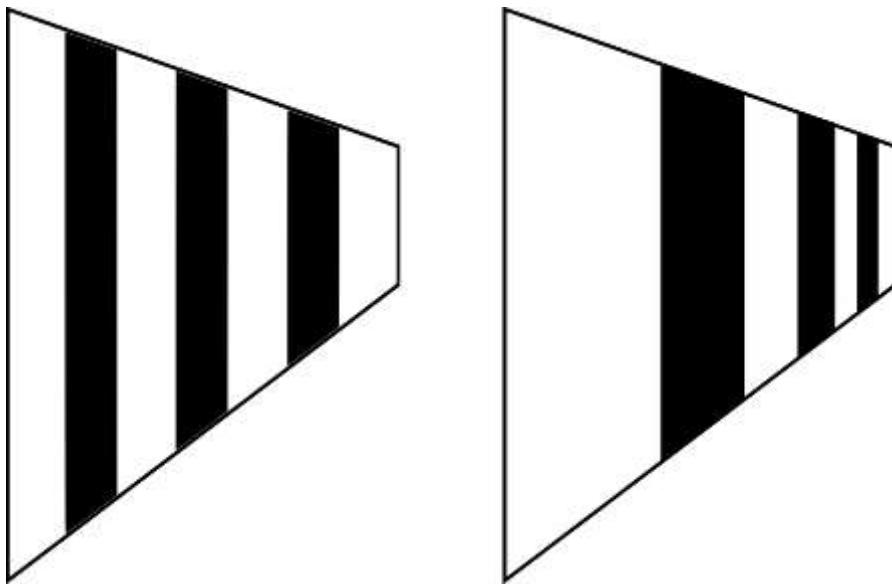


图 14-4：线性插值 u 和 v (左) 不会产生预期的透视校正结果 (右)。

这种情况与我们在[第12章（隐藏表面去除）](#)中遇到的情况非常相似，解决方案也非常相似：虽然 u 和 v 屏幕坐标不是线性的， $\frac{u}{z}$ 和 $\frac{v}{z}$ 是。（证明与 $\frac{1}{z}$ 证明：考虑一下 u 在 3D 空间中线性变化，并替换 x 和 y 及其屏幕空间表达式。由于我们已经有 $\frac{1}{z}$ 在每个像素上，插值就足够了 $\frac{u}{z}$ 和 $\frac{v}{z}$ 并得到 u 和 v 返回：

$$u = \frac{\frac{u}{z}}{\frac{1}{z}}$$

$$v = \frac{\frac{v}{z}}{\frac{1}{z}}$$

这将产生我们预期的结果，如图 14-5 所示。



图 14-5: u/z 和 v/z 的线性插值确实会产生透视校正结果。

图 14-6 并排显示了两个结果，以便更容易理解差异。



图 14-6: “线性 u 和 v ” 结果 (左) 和 “线性 u/z 和 v/z ” 结果 (右) 的比较

这些示例看起来不错，因为纹理的大小和我们应用纹理的三角形的大小（以像素为单位）大致相似。但是，如果三角形比纹理大或小几倍，会发生什么？接下来我们将探讨这些情况。

双线性滤波

假设我们将相机放置在非常靠近其中一个立方体的位置。我们将看到如图 14-7 所示的内容。



图 14-7：从近距离渲染的纹理对象

图像看起来非常块状。为什么会这样？屏幕上的三角形具有的像素数多于纹理具有像素的像素数，因此每个纹素映射到许多连续的像素。

我们正在插值纹理坐标 u 和 v ，它们是介于 0.0 和 1.0。稍后，给定纹理尺寸 w 和 h ，我们映射 u 和 v 坐标到 tx 和 ty 纹素坐标乘以 w 和 h 分别。但是因为纹理是具有整数索引的像素数组，所以我们舍入 tx 和 ty 向下到最接近的整数。因此，这种基本技术称为最近邻过滤。

便 (u, v) 在主三角形的表面上平滑变化，生成的纹素坐标从一个完整的像素“跳”到下一个像素，导致我们在图 14-7 中看到的块状外观。

我们可以做得更好。而不是四舍五入 tx 和 ty 向下，我们可以解释分数纹素坐标 (tx, ty) 描述四个整数纹素坐标之间的位置（通过四舍五入的组合获得 tx 和 ty 上下）。我们可以取周围整数纹素的四种颜色，并计算分数纹素的线性插值颜色。这将产生明显更平滑的结果（图 14-8）。



图 14-8：使用插值颜色从近距离渲染的纹理对象

我们来称周围的四个像素 TL, TR, BL, BR (分别适用于左上角、右上角、左下角和右下角)。让我们取 tx 和 ty 并打电话给他们 fx 和 fy . 图 14-9 显示 C , 描述的确切位置(tx, ty) 在整数坐标处被纹素包围，以及它与它们的距离。

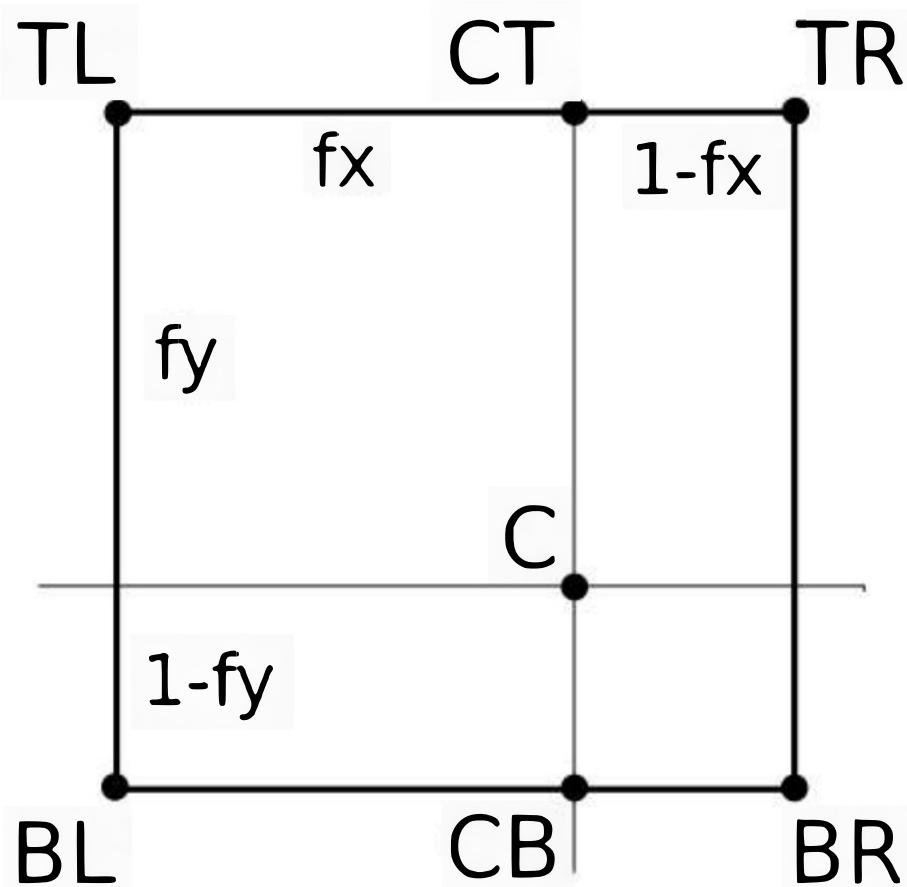


图 14-9: 我们从围绕它的四个纹素中线性插值 C 处的颜色。

首先，我们线性插值颜色 CT ，介于 TL 和 TR :

$$CT = (1 - fx) \cdot TL + fx \cdot TR$$

请注意，重量为 TR 是 fx ，不是 $(1 - fx)$ 。这是因为作为 fx 接近 1.0，我们想要 CT 更贴近 TR 。事实上，如果 $fx = 0.0$ ，然后 $CT = TL$ ，如果 $fx = 1.0$ ，然后 $CT = TR$ 。我们可以计算 CB 之间 TL 和 TR ，以类似的方式：

$$CB = (1 - fy) \cdot BL + fy \cdot BR$$

最后，我们计算 C ，在 CT 和 CB :

$$C = (1 - fy) \cdot CT + fy \cdot CB$$

在伪代码中，我们可以编写一个函数来获取对应于分数纹素的插值颜色：

```
GetTexel(texture, tx, ty) {
    fx = frac(tx)
    fy = frac(ty)
    tx = floor(tx)
    ty = floor(ty)

    TL = texture[tx][ty]
    TR = texture[tx+1][ty]
    BL = texture[tx][ty+1]
    BR = texture[tx+1][ty+1]

    CT = fx * TR + (1 - fx) * TL
    CB = fy * BR + (1 - fy) * BL

    return fy * CB + (1 - fy) * CT
}
```

此函数使用，将数字向下舍入到最接近的整数，以及返回数字的小数部分，可以定义为。`floor()frac(x) - floor(x)`

这种技术称为双线性滤波（因为我们进行两次线性插值，每个维度一次）。

米普映射

让我们考虑相反的情况，从远处渲染对象。在这种情况下，纹理的纹素比三角形的像素多得多。为什么这是一个问题可能不太明显，因此我们将使用精心选择的情况来说明。

考虑一个方形纹理，其中一半像素是黑色的，一半的像素是白色的，以棋盘图案布局（图 14-10）。

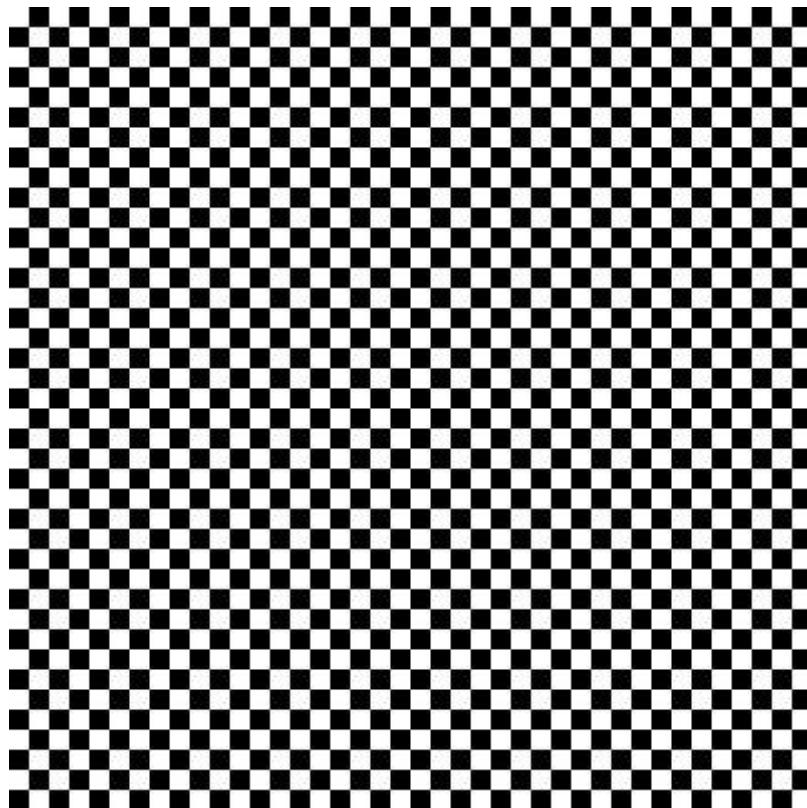


图 14-10：黑白棋盘格纹理

假设我们将此纹理映射到视口中的正方形上，这样当它在画布上绘制时，正方形的宽度（以像素为单位）正好是纹理宽度（纹素）的一半。这意味着实际上只会使用四分之一的纹素。

我们直觉地期望正方形看起来是灰色的。但是，考虑到我们进行纹理映射的方式，我们可能会不走运，并得到所有的白色像素或所有黑色像素。的确，我们可能很幸运，得到了50/50的黑白像素组合，但我们期望的50%的灰色并不能保证。请看图 14-11，其中显示了不幸的情况。

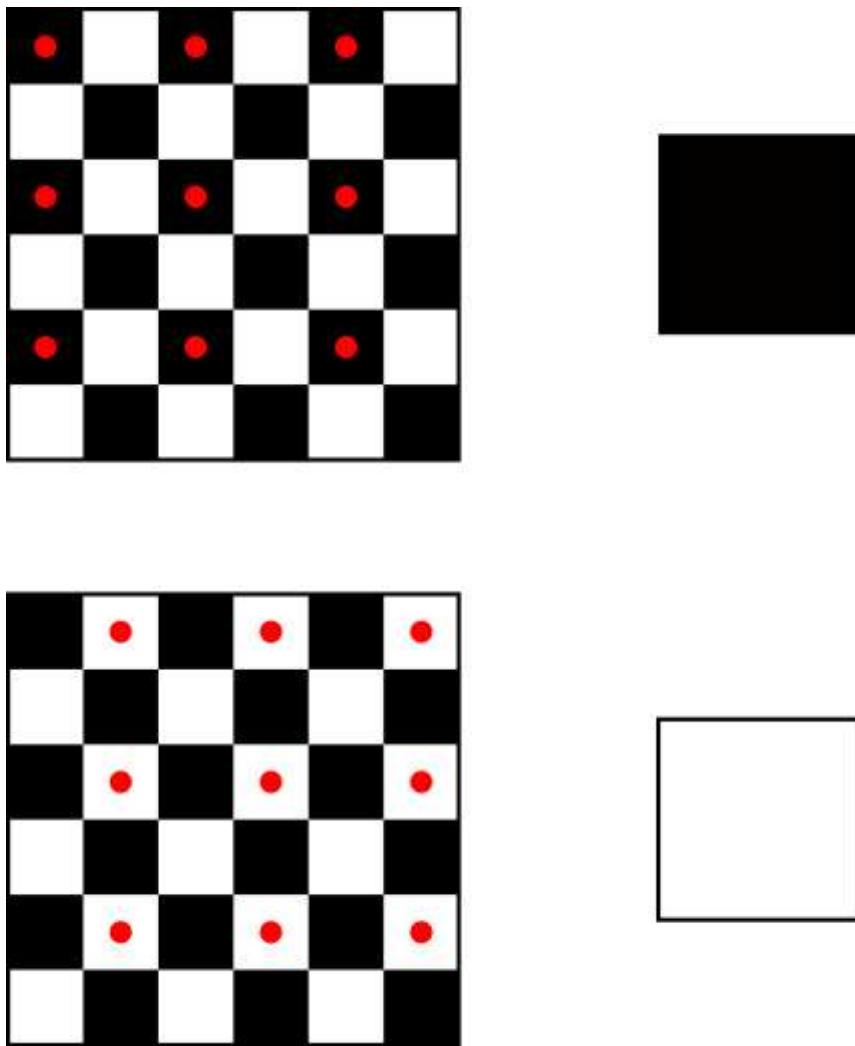


图 14-11：在小对象上映射大纹理可能会导致意外结果，具体取决于恰好选择了哪些纹素。

如何解决这个问题？从某种意义上说，正方形的每个像素都代表一个 2×2 纹理的纹素区域，因此我们可以计算该区域的平均颜色并将该颜色用于像素。平均黑白像素会给我们带来我们正在寻找的灰色。

但是，这可能会很快变得非常昂贵。假设正方形更远，因此它是纹理宽度的十分之一。这意味着正方形中的每个像素都代表一个 10×10 纹理的纹素区域。我们必须为要渲染的每个像素计算 100 个纹素的平均值！

幸运的是，这是我们可以用一点额外内存替换大量计算的情况之一。让我们回到最初的情况，其中正方形是纹理宽度的一半。与其计算我们想要一次又一次地为每个像素渲染的四个纹素的平均值，我们可以预先计算原始大小一半的纹理，其中半大小纹理中的每个纹素都是原始纹理中相应四个纹素的平均值。稍后，当需要渲染像素时，我们可以在这个较小的纹理中查找纹素，甚至可以应用上一节中所述的双线性过滤。

这样，我们获得了平均四个像素的更好的渲染质量，但代价是单个纹理查找的计算成本。这确实需要一点预处理时间（例如，在加载纹理时）和更多的内存（用于存储全尺寸和半尺寸纹理），但总的来说，这是一个值得的权衡。

那 $10 \times$ 我们上面讨论的大小方案？我们可以进一步采用这种技术，还可以预先计算原始纹理的四分之一、八分之一和十六分之一大小版本（精确到 1×1 纹理，如果我们愿意的话）。然后，在渲染三角形时，我们将使用比例与其大小最匹配的纹理，并获得平均数百甚至数千像素的所有好处，而无需额外的运行时成本。

这种强大的技术称为 *mipmapping*。这个名字来源于拉丁语 *multum in parvo*，意思是“小中多”。

计算所有这些较小规模的纹理确实需要内存成本，但它比您想象的要小得多。

假设纹理的原始区域（以纹素为单位）为 A ，其宽度为 w 。半角纹理的宽度为 $\frac{w}{2}$ ，但它只需要 $\frac{A}{4}$ 纹素；四分之一宽度的纹理需要 $\frac{A}{16}$ 纹素；等等。图 14-12 显示了原始纹理和前三个简化版本。

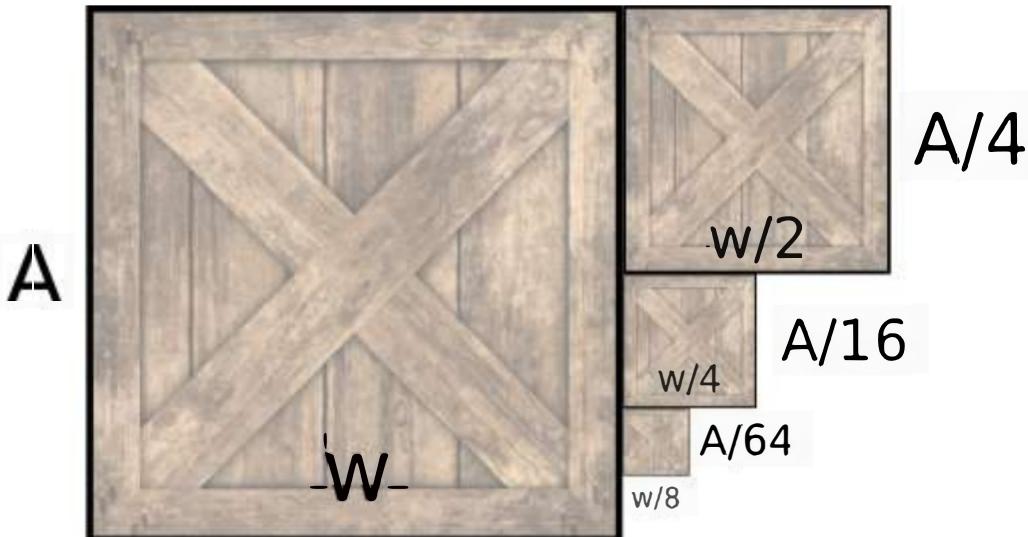


图 14-12：纹理及其逐渐变小的 mipmap

我们可以将纹理大小的总和表示为无穷级数：

$$A + \frac{A}{4} + \frac{A}{16} + \frac{A}{64} + \dots = \sum_{n=0}^{\infty} \frac{A}{4^n}$$

本系列收敛到 $A \cdot 4/3$ ，或 $A \cdot 1.3333$ ，这意味着所有较小的纹理都下降到 1×1 特塞尔只比原始纹理多占用三分之一的空间。

三线性滤波

让我们更进一步。想象一个远离相机的物体。我们使用最适合其大小的 mipmap 级别来渲染它。

现在想象一下相机向物体移动。在某些时候，最合适的选择会从一帧更改为下一帧，这将导致微妙但明显的差异。

在选择 mipmap 级别时，我们选择与纹理和正方形的相对大小最匹配的级别。例如，对于比纹理小 10 倍的正方形，我们可以选择比原始纹理小 8 倍的 mipmap 级别，并对其应用双线性过滤。但是，我们也可以考虑与相对大小最匹配的两个 mipmap 级别（在这种情况下，小 8 倍和 16 倍），并在它们之间进行线性插值，具体取决于 mipmap 大小比和实际大小比之间的“距离”。

因为来自每个 mipmap 级别的颜色是双倍插值的，我们在上面应用另一种线性插值，所以这种技术称为三线性过滤。

总结

在本章中，我们让光栅器的质量有了巨大的飞跃。在本章之前，每个三角形可以有单一的颜色；现在我们可以在它们上面绘制任意复杂的图像。

我们还讨论了如何确保带纹理的三角形看起来不错，无论三角形的相对大小和纹理如何。我们提出了双线性滤波、mipmapping和三线性滤波作为低质量纹理最常见原因的解决方案。