扩展光线追踪器

在本书的第一部分结束时，我们将快速讨论几个我们尚未涵盖的有趣主题：将相机放置在场景中的任何位置、性能优化、球体以外的基元、使用构造实体几何体对对象建模、支撑透明表面和超采样。我们不会实施所有这些更改，但我鼓励您尝试一下！前面的章节，加上下面提供的描述，为你自己探索和实现它们奠定了坚实的基础。

任意摄像机定位

在关于光线追踪的讨论之初，我们做了三个重要的假设：相机固定在（0，0，0）它是向$\vec{Z_+}$，并且它的“向上”方向是$\vec{Y_+}$.在本节中，我们将取消这些限制，以便我们可以将摄像机放置在场景中的任何位置，并将其指向任何方向。

让我们从相机位置开始。您可能已经注意到O在所有伪代码中只使用一次：作为顶级方法中来自相机的光线的来源。如果我们想改变相机的位置，我们*唯*一需要做的就是使用 不同的值O我们完成了。

*位置*的变化会影响光线*的方向*吗？一点也不。光线的方向是从相机到投影平面的矢量。当我们移动相机时，投影平面会随之移动，因此它们的相对位置不会改变。我们的写作方式与这个想法是一致的。`CanvasToViewport`

让我们把注意力转向相机方向。假设您有一个表示相机所需方向的旋转矩阵。如果您只是旋转相机，相机*的位置*不会改变，但它朝向的方向会改变;它经历与整个相机相同的旋转。所以如果你有光线方向$\vec{D}$和旋转矩阵R、旋转的D只是$R \cdot \vec{D}$

总之，唯一需要更改的函数是我们在示例 2-2 中写回的主函数。示例 5-1 显示了更新后的函数：

```
for x in [-Cw/2, Cw/2] {
    for y in [-Ch/2, Ch/2] {
        ❶D = camera.rotation * CanvasToViewport(x, y)
        ❷color = TraceRay(camera.position, D, 1, inf)
         canvas.PutPixel(x, y, color)
    }
}
```

示例 5-1： 主循环，更新为支持任意相机位置和方向

我们将相机的旋转矩阵❶（描述其在空间中的方向）应用于我们将要追踪的光线方向。然后我们使用相机位置作为光线的起点❷。

图 5-1 显示了从不同位置和不同摄像机方向渲染场景时的外观。

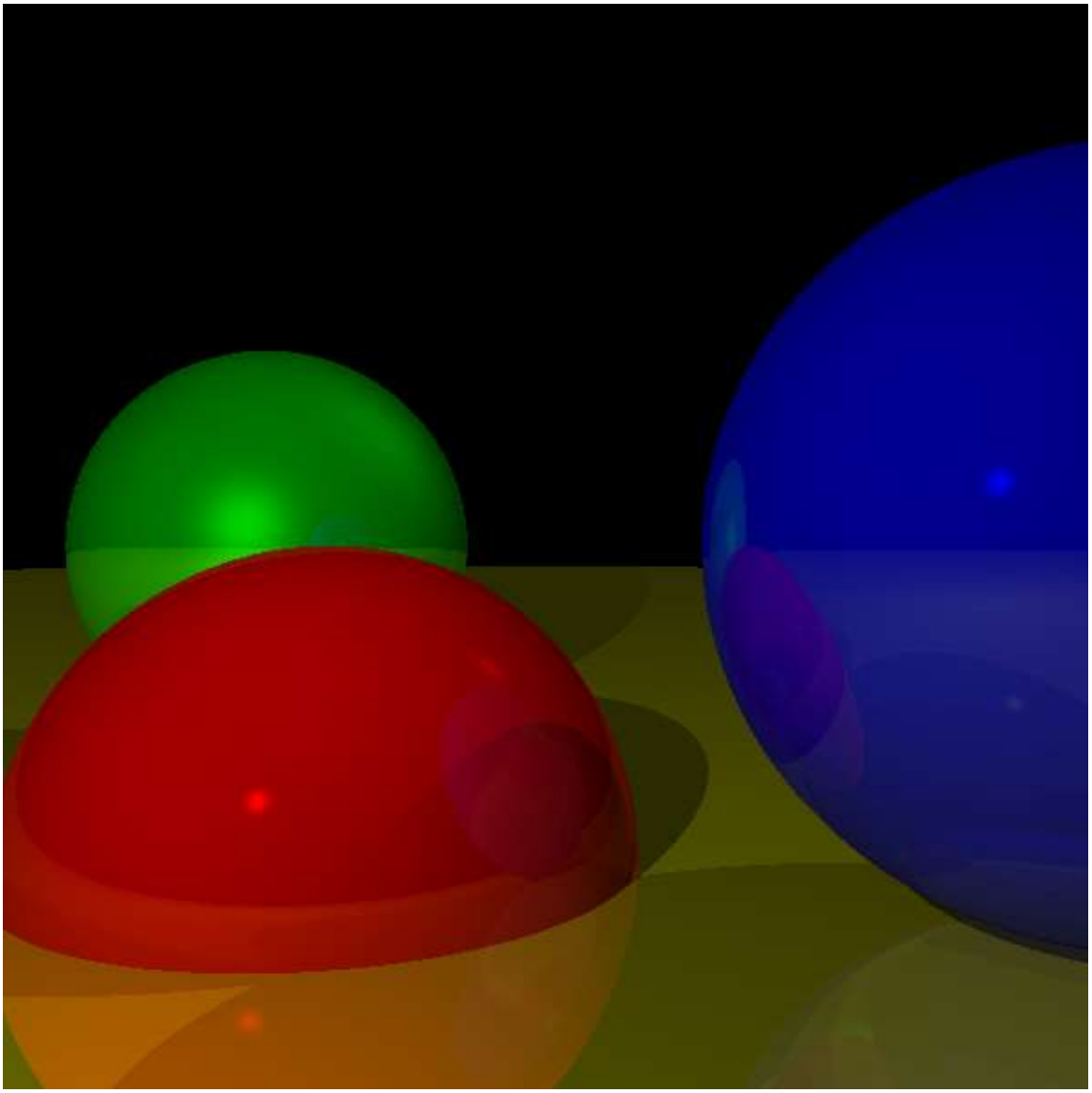

图 5-1：我们熟悉的场景，使用不同的摄像机位置和方向渲染

源代码和现场演示>>

性能优化

前面的章节重点介绍了解释和实现光线追踪器不同功能的最清晰方法。因此，它功能齐全，但不是特别快。以下是您可以自己探索的一些想法，以使光线追踪器更快。只是为了好玩，测量每个前后的时间。你会对结果感到惊讶！

并行

使光线追踪器更快的最明显方法是一次追踪多条光线。由于离开摄像机的每条光线都独立于其他每条光线，并且场景数据是只读的，因此您可以跟踪每个 CPU 内核的一束光线，而不会造成很多损失或太多的同步复杂性。事实上，光线追踪器属于一类被称为尴*尬并行化的*算法，正是因为它们的本质使它们非常容易并行化。

不过，每条射线生成一个线程可能不是一个好主意;管理潜在数百万个线程的开销可能会抵消您获得的加速。一个更明智的想法是创建一组“任务”，每个任务负责对画布

的一部分（矩形区域，低至单个像素）进行光线跟踪，并将它们调度到物理内核上运行的工作线程。

缓存不可变值

*缓存*是一种避免一遍又一遍重复相同计算的方法。每当有昂贵的计算并且您希望重复使用此计算的结果时，最好存储（缓存）此结果并在下次需要时重用它，尤其是在此值不经常更改的情况下。

考虑在 中计算的值，光线追踪器通常花费大部分时间：`IntersectRaySphere`

```
a = dot(D, D)
b = 2 * dot(OC, D)
c = dot(OC, OC) - r * r
```

不同的值在不同的时间段是不可变的。

加载场景并知道球体的大小后，就可以计算 .除非球体的大小发生变化，否则该值不会更改。`r * r`

至少，某些值对于整个帧是不可变的。一个这样的值是，如果相机或球体移动，它只需要在帧之间更改。（请注意，阴影和反射会跟踪不是从相机开始的光线，因此需要小心以确保在这种情况下不使用缓存值。`dot(OC, OC)`

对于整个光线，某些值不会改变。例如，您可以计算 并将其传递给 。
`dot(D, D)ClosestIntersectionIntersectRaySphere`

还有许多其他计算可以重用。发挥你的想象力！但是，并非每个缓存值都会使整体上更快，因为有时簿记开销可能高于节省的时间。始终使用基准来评估优化是否真正有帮助。

阴影优化

当表面的某个点由于有另一个物体挡在路上而处于阴影中时，它旁边的点很可能也会在同一对象的阴影中（这称为*阴影相干*）。您可以在图 5-2 中看到一个示例。

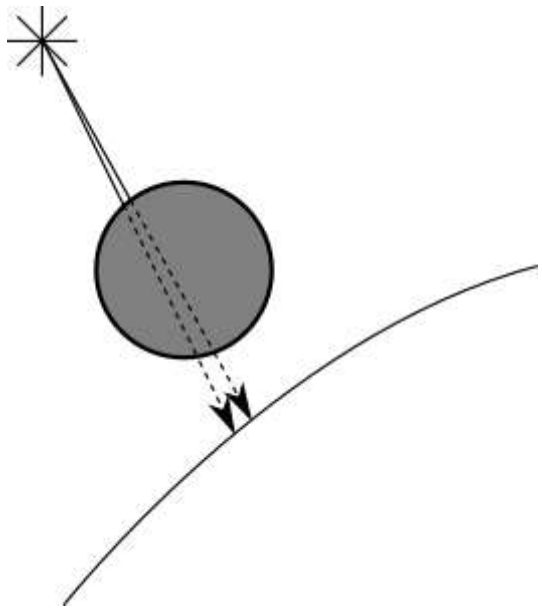

图 5-2：靠近的点可能位于同一对象的阴影中。

当搜索点和光源之间的对象时，为了确定该点是否在阴影中，我们通常会检查与其他每个对象的交集。但是，如果我们知道紧挨着它的点在特定对象的阴影中，我们可以先检查与该对象的交集。如果我们找到一个，我们就完成了，我们不需要检查所有其他对象！如果我们没有找到与该对象的交集，我们只是恢复到检查每个对象。

同样，在寻找光线对象相交以确定点是否在阴影中时，您实际上并不需要*最近的*交点;知道*至少有一个*十字路口就足够了，因为这足以阻止光线到达点！因此，您可以在找到*任何*交集后立即编写该返回的专用版本。您也不需要计算和返回;相反，您可以只返回一个布尔值。`ClosestIntersectionclosest_t`

空间结构

计算光线与场景中每个球体的交点有点浪费。有许多数据结构允许您一次丢弃整个对象组，而无需单独计算交集。

假设您有几个彼此靠近的球体。您可以计算包含所有这些球体的最小球体的中心和半径。如果射线不与这个边界球相交，则可以确定它不会与它包含的任何*球*体相交，代价是单次相交测试。当然，如果是这样，您仍然需要检查它是否与它包含的任何球体相交。

您可以更进一步，拥有多个级别的边界球体（即球体组），形成一个层次结构，仅当其中一个实际球体很有可能与射线相交时，才需要一直遍历到底部。

虽然这一系列技术的确切细节不在本书的讨论范围之内，但您可以在名称*边界卷层次结构*下找到更多信息。

子抽样

这是制作光线追踪器的简单方法N倍快：计算速度N像素少几倍！

对于画布中的每个像素，我们通过视口跟踪一条光线，以*采样*来自该方向的光线的颜色。如果我们的光线少于像素，我们将对场景进行*子采样*。但是我们如何才能做到这一点并仍然正确渲染场景呢？

假设您跟踪像素的光线$(10, 100)(12, 100)$它们碰巧碰到了同一个物体。您可以合理地假设光线为像素$(11, 100)$击中同一个对象，因此您可以跳过与场景中所有对象的交集的初始搜索，直接跳转到计算该点的颜色。

如果您在水平和垂直方向上跳过每隔一个像素，则可以减少多达 75% 的主要光线场景交叉计算——这是 4 倍的加速！

当然，您很可能会错过一个非常薄的物体;这是一种“不纯”的优化，从某种意义上说，与前面讨论的优化不同，它产生的图像与未优化的图像非常相似，但不能保证与图像相同。在某种程度上，这是偷工减料的“作弊”。诀窍是知道在保持令人满意的结果的同时可以削减哪些角落;在计算机图形学的许多领域，重要的是结果的主观质量。

支持其他基元

在前面的章节中，我们使用球体作为基元，因为它们在数学上易于操作;也就是说，找到射线和球体之间交点的方程相对简单。但是，一旦您拥有了可以渲染球体的基本光线追踪器，添加对渲染其他基元的支持就不需要太多额外的工作。

请注意，只需要能够计算光线和任何给定对象的两件事：值`TraceRay`t对于它们与该交叉点处的法线之间的最近交叉点。光线追踪器中的其他所有内容都是与对象无关的！

三角形是一个很好的基元支撑。三角形是最简单的多边形，因此您可以用三角形构建任何其他多边形。它们在数学上易于操作，因此它们是表示更复杂表面近似值的好方法。

要向光线追踪器添加三角形支持，您只需更改 .首先，计算光线（由其原点和方向给出）与包含三角形的平面（由其法线和与原点的距离给出）之间的交点。`TraceRay`

由于平面无限大，光线几乎总是与任何给定平面相交（除非它们完全平行）。所以第二步是确定射线平面相交点是否真的在三角形内。有许多方法可以做到这一点，包括使用重心坐标或使用叉积来检查点相对于三角形的三条边中的每一条是否“在内侧”。

一旦确定点在三角形内，交点处的法线就是平面的法线。返回适当的值，无需进一步更改！`TraceRay`

构造实体几何

假设我们要渲染比球体或弯曲对象更复杂的对象，这些对象很难使用一组三角形准确建模。两个很好的例子是镜头（比如放大镜里的镜头）和死星（那不是月亮）。

我们可以很容易地用简单的语言描述这些对象。放大镜看起来像粘在一起的两片球体;死星看起来像一个球体，从中取出一个较小的球体。

我们可以更正式地将其表示为将集合运算（例如并集、交集或差分）应用于其他对象的结果。继续我们上面的例子，透镜可以被描述为两个球体的交点，死星被描述为一个大球体，我们从中减去一个小球体（见图5-3）。

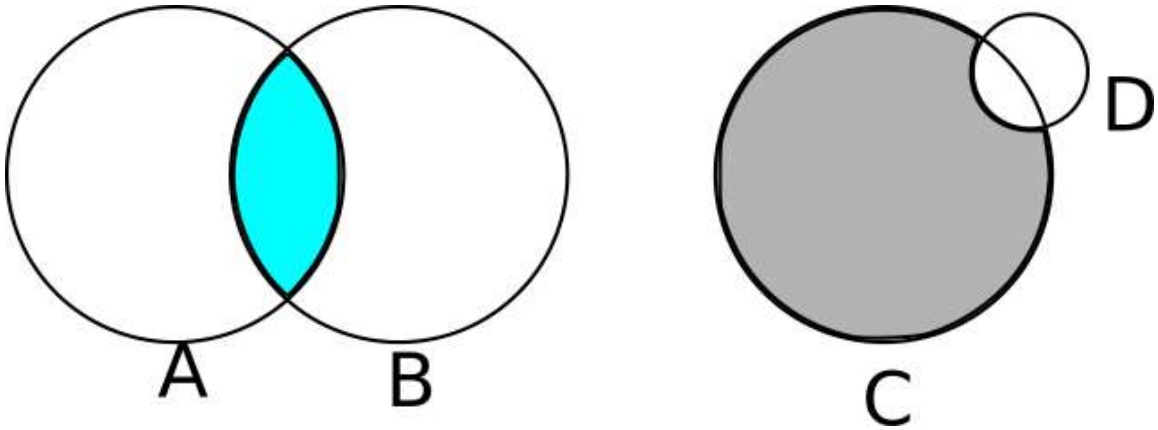

图 5-3：构造实体几何体的实际应用。 $A \cap B$给了我们一个镜头。$C - D$给了我们死星。

您可能会认为计算固体对象的布尔运算是一个非常棘手的几何问题。你会完全正确！幸运的是，*事实证明，建设性实体几何*体允许我们渲染对象之间的设置操作的结果，而无需显式计算这些结果！

我们如何在光线追踪器中做到这一点？对于每个对象，您可以计算光线进入和离开对象的点;例如，在球体的情况下，光线进入米·(t_1, t_2) 并在米阿$x(t_1, t_2)$假设您要计算两个球体的交点;当光线在*两个*球体内时，它位于交点内，当它位于*任一*球体外时，它位于交点外。在减法的情况下，当光线在第一个物体内而不是第二个物体内时，它就在里面。对于两个物体的并集，当光线在任何一个物体内时，它就在里面。

更一般地说，如果要计算射线和对象之间的交集$A \odot B$（其中$\odot$是任意集合运算），您首先计算射线和A和B 单独，这为您提供了t每个物体都在“内部”，R_A和R_B.然后你计算$R_A \odot R_B$，这是 的“内部”范围$A \odot B$.一旦你有了这个，射线和$A \odot B$是 的最小值t这既在对象的“内部”范围内，又介于$t_{米}$ 和$t_{米阿x}$图 5-4 显示了两个球体的并集、交集和减法的内部范围。

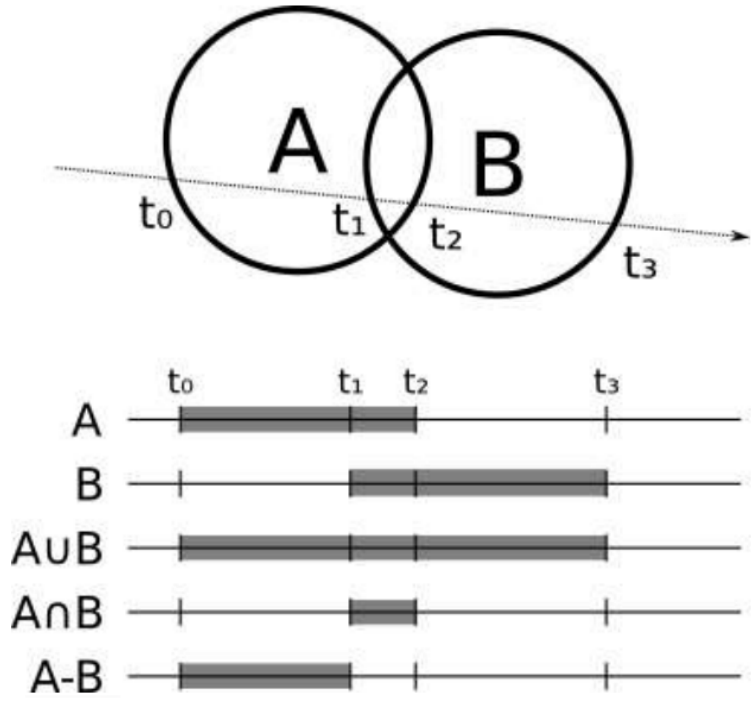

图 5-4：两个球体的并集、交集和减法

交叉点处的法线要么是产生相交对象的法线，要么是相反的，具体取决于您是在看原始对象的“外部”还是“内部”。

答案是肯定的A和B 不必是原始人;它们可以是集合操作本身的结果！如果你干净地实现这一点，你甚至不需要知道*什么*是A和B ，只要你能从中得到交叉点和法线。通过这种方式，您可以采用三个球体并计算，例如，$(A \cup B) \cap C$

透明度

到目前为止，我们已经渲染了每个对象，就好像它是完全不透明的一样，但事实并非如此。我们可以渲染部分透明的对象，比如鱼缸。

实现这一点与实现反射非常相似。当光线照射到部分透明的表面时，您可以像以前一样计算局部颜色和反射颜色，但您还会计算另一种颜色，即*通过对象进行*另一次调用获得的光线的颜色。然后，根据对象的透明度，将此颜色与局部和反射颜色混合，这与我们在计算对象反射时所做的方式大致相同。TraceRay

折射

在现实生活中，当一束光线穿过透明物体时，它会改变方向（这就是为什么当你将吸管浸入一杯水中时，它看起来“破碎”）。更准确地说，当光线穿过材料（如空气）并进入不同的材料（如水）时，它会改变方向。

方向变化的方式取决于每种材料的特性，称为*其折射率*，根据以下方程，称为斯涅尔定律：

$$\frac{\sin(\alpha_1)}{\sin(\alpha_2)} = \frac{n_2}{n_1}$$

这里α_1和α_2是光线与穿过表面前后的法线之间的角度，以及n_1和n_2是物体外部和内部材料的折射率。

例如n_{air}大约是1.0和n_{Water}大约是1.33.因此，对于一束光线进入水中$60°$角度，我们有

$$\frac{\sin(60)}{\sin(\alpha_2)} = \frac{1.33}{1.0}$$

$$\sin(\alpha_2) = \frac{\sin(60)}{1.33}$$

$$\alpha_2 = \arcsin\left(\frac{\sin(60)}{1.33}\right) = 40.628°$$

此示例如图 5-5 所示。

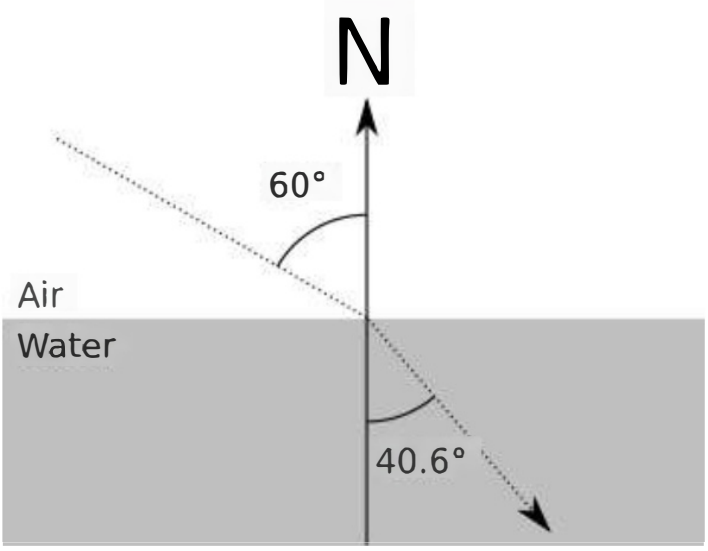

图 5-5：光线在离开空气并进入水中时折射（改变方向）。

在实现层面，每条射线都必须携带一条额外的信息：它当前正在经历的材料的折射率。当光线与部分透明对象相交时，您可以根据当前材质和新材质的折射率从该点计算光线的新方向，然后像以前一样继续。

停下来考虑一下：如果您实现了建设性实体几何体和透明度，则可以对放大镜（两个球体的交集）进行建模，其行为类似于物理上正确的放大镜！

超级采样

*超级*采样或多或少与子采样相反。在这种情况下，您正在寻找准确性而不是性能。假设对应于两个相邻像素的光线击中不同的物体。您将用相应的颜色绘制每个像素。

但请记住我们开始的类比：每条光线都应该确定我们正在查看的“网格”的每个*正方形*的“代表性”颜色。通过每像素使用一条光线，我们武断地决定穿过正方形中间的光线的颜色代表整个正方形，但这可能不是真的。

解决这个问题的方法只是在每个像素上追踪更多的光线——4、9、16，你想要的——然后平均它们以获得像素的颜色。

当然，这会使您的光线追踪器慢 4、9 或 16 倍，原因与子采样使其完全相同。N倍快。幸运的是，有一个中间立场。您可以假设对象属性在其表面上平滑变化，因此每像素拍摄四条光线，在非常不同的位置击中同一对象可能不会对场景产生太大改善。因此，您可以从每个像素一条光线开始，然后比较相邻的光线：如果它们击中不同的物体或颜色差异超过某个阈值，则对两者应用像素细分。

总结

在本章中，我们简要介绍了您可以自己探索的几个想法。这些以新的和有趣的方式修改了我们一直在开发的基本光线追踪器——使其更高效，能够表示更复杂的物体，或者以更好地接近我们的物理世界的方式对光线进行建模。

本书的第一部分应该证明光线追踪器是漂亮的软件，只需使用简单直观的算法和简单的数学即可生成令人惊叹的美丽图像。

可悲的是，这种纯度是有代价的：性能。虽然有许多方法可以优化和并行化光线追踪器，但如本章所述，它们对于实时性能来说仍然过于昂贵;虽然硬件每年都在变快，但一些应用程序要求图片速度快 100 倍，而且质量没有损失。在所有这些应用程序中，游戏的要求最高：我们期望每秒至少绘制 60 次完美的图像。光线追踪器只是不剪切它。

那么，自90年代初以来，电子游戏是如何做到的呢?

答案在于一个完全不同的算法家族，我们将在本书的第二部分探讨。