

线

在本书的第一部分中，我们广泛研究了光线追踪，并开发了一种光线追踪器，可以使用相对简单的算法和数学来渲染具有精确照明、材质属性、阴影和反射的测试场景。这种简单性是有代价的：性能。虽然非实时性能对于某些应用程序（例如建筑可视化或电影的视觉效果）很好，但对于其他应用程序（例如视频游戏）来说还不够。

在本书的这一部分中，我们将探索一组完全不同的算法，这些算法有利于性能而不是数学纯度。

我们的光线追踪器从摄像机开始，通过视口探索场景。对于画布的每个像素，我们回答这个问题，“**场景中的哪个对象在这里可见？**现在，我们将遵循一种在某种意义上相反的方法：对于场景中的每个对象，我们将尝试回答以下问题：“**该对象在画布的哪些部分可见？**

事实证明，我们可以开发算法来回答这个新问题，比光线追踪快得多，只要我们愿意做出一些准确性权衡。稍后，我们将探讨如何使用这些快速算法来获得与光线追踪器相当的质量的结果。

我们将从头开始：我们有一个维度画布 C_w 和 C_h ，我们可以设置单个像素的颜色，但不能设置其他颜色。让我们探索如何在画布上绘制最简单的元素：两点之间的一条线。`PutPixel()`

描述线条

假设我们有两个画布点， P_0 和 P_1 ，带坐标 (x_0, y_0) 和 (x_1, y_1) 分别。我们如何在两者之间画直线段 P_0 和 P_1 ？

让我们从用参数坐标表示一条线开始，就像我们之前对射线所做的那样（事实上，您可以将“射线”视为 3D 中的线）。任何点 P 上线可以通过从 P_0 并沿方向移动一段距离 P_0 自 P_1 ：

$$P = P_0 + t(P_1 - P_0)$$

我们可以将这个等式分解为两个，每个坐标一个：

$$x = x_0 + t \cdot (x_1 - x_0)$$

$$y = y_0 + t \cdot (y_1 - y_0)$$

让我们取第一个方程并求解 t ：

$$x = x_0 + t \cdot (x_1 - x_0)$$

$$x - x_0 = t \cdot (x_1 - x_0)$$

$$\frac{x - x_0}{x_1 - x_0} = t$$

我们现在可以将此表达式插入 t 进入第二个等式：

$$y = y_0 + t \cdot (y_1 - y_0)$$

$$y = y_0 + \frac{x - x_0}{x_1 - x_0} \cdot (y_1 - y_0)$$

稍微重新排列一下：

$$y = y_0 + (x - x_0) \cdot \frac{y_1 - y_0}{x_1 - x_0}$$

请注意 $\frac{y_1 - y_0}{x_1 - x_0}$ 是一个常量，是一个仅取决于线段端点的常量，因此，我们可以将上面的等式重写为

$$y = y_0 + a \cdot (x - x_0)$$

什么是一个？根据我们定义它的方式，它测量 y 单位变化坐标 x 坐标；换句话说，它是线斜率的度量。

让我们回到等式。分配乘法：

$$y = y_0 + ax - ax_0$$

对常量进行分组：

$$y = -x + (y_0 - ax_0)$$

再 $(y_0 - ax_0)$ 仅取决于段的端点；我们称之为 b . 最后我们得到

$$y = ax + b$$

这是线性函数的标准公式，可用于表示几乎任何线条。当我们解决 t ，我们添加了一个除法 $x_1 - x_0$ 不假思索，如果 $x_1 = x_0$. 我们不能除以零，这意味着这个公式不能表示 $x_1 = x_0$ ——即垂直线。

为了解决这个问题，我们现在只忽略垂直线，稍后再弄清楚如何处理它们。

绘制线条

我们现在有一种方法可以获取 y 对于 x 我们对此很感兴趣。这给了我们一对 (x, y) 满足直线方程。

现在我们可以编写一个函数的第一个近似值，该函数从中绘制线段 P_0 自 P_1 . 让并成为 $x_0 y_0$ 和 y 的坐标 P_0 分别和 $x_1 y_1 P_1$. 若 $x_0 < x_1$ ，我们可以从 x_0 自 x_1 ，计算值 y 对于 x ，并在以下坐标处绘制一个像素：

```

DrawLine(P0, P1, color) {
    a = (y1 - y0) / (x1 - x0)
    b = y0 - a * x0
    for x = x0 to x1 {
        y = a * x + b
        canvas.PutPixel(x, y, color)
    }
}

```

请注意，除法运算符应执行实除法，而不是整数除法。尽管 x 和 y 在此上下文中为整数，因为它们表示画布上像素的坐标。

另请注意，我们认为循环包含范围的最后一个值。在C, C++, Java和JavaScript等中，这将写为。我们将在本书中使用此约定。`for (x = x0; x <= x1; ++x)`

此函数是上述等式的直接、朴素的实现。它有效，但我们能让它更快吗？

我们不计算以下值 y 对于任何任意 x .相反，我们仅以整数增量计算它们 x 我们这样做是有顺序的。计算后立即 $y(x)$ ，我们计算 $y(x + 1)$:

$$y(x) = ax + b$$

$$y(x + 1) = a \cdot (x + 1) + b$$

我们可以稍微操纵一下第二个表达式：

$$y(x + 1) = ax + a + b$$

$$y(x + 1) = (ax + b) + a$$

$$y(x + 1) = y(x) + a$$

这应该不足为奇;毕竟，斜率 a 是 x 增加1时 y 变化的程度，这正是我们在这里所做的。

这意味着我们可以计算下一个值 y 只需取以前的值 y 并添加斜率;不需要每像素乘法，这使得函数更快。一开始没有“以前的值 y ”，所以我们从 (x_0, y_0) .然后我们继续添加1到 x ，把 a 加到 y ，直到我们到达 x_1 .

再次假设 $x_0 < x_1$ ，我们可以按如下方式重写函数：

```

DrawLine(P0, P1, color) {
    a = (y1 - y0) / (x1 - x0)
    y = y0
    for x = x0 to x1 {
        canvas.PutPixel(x, y, color)
        y = y + a
    }
}

```

到目前为止，我们一直假设 $x_0 < x_1$.有一个简单的解决方法来支持不成立的线条：由于我们绘制像素的顺序无关紧要，如果我们得到一条从右到左的线，我们可以交换并将其转换为同一行的从左到右版本，并像以前一样绘制它： $P0P1$

```
DrawLine(P0, P1, color) {
    // Make sure x0 < x1
    if x0 > x1 {
        swap(P0, P1)
    }
    a = (y1 - y0) / (x1 - x0)
    y = y0
    for x = x0 to x1 {
        canvas.PutPixel(x, y, color)
        y = y + a
    }
}
```

让我们使用我们的函数绘制几条线。图 6-1 显示了线段
(-200, -100) – (240, 120)，图6-2显示了该线的特写。

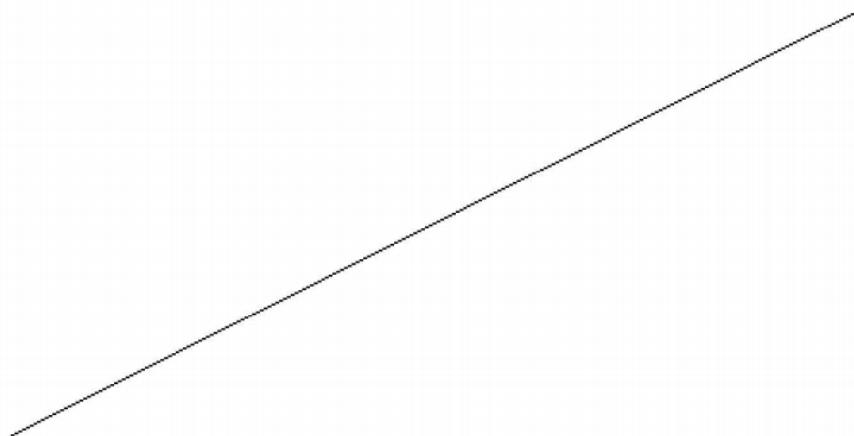


图 6-1: 直线

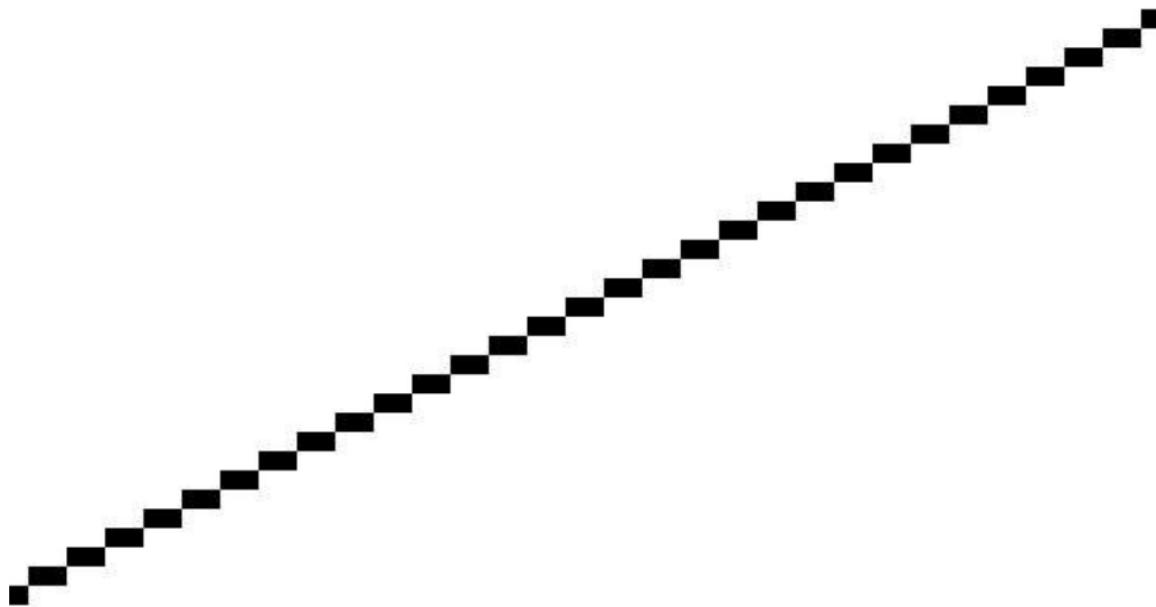


图 6-2: 放大直线

这条线出现锯齿状，因为我们只能在整数坐标上绘制像素，而数学线实际上宽度为零；我们绘制的是理想线的量化近似值 $(-200, -100) - (240, 120)$ 。方法可以绘制更漂亮的近似线（您可能希望查看 MSAA、FXAA、SSAA 和 TAA 作为一组有趣的兔子洞的可能入口点）。我们不会去那里有两个原因：(1) 它更慢，(2) 我们的目标不是画漂亮的线条，而是开发一些基本的算法来渲染3D场景。

让我们尝试另一行，让我们尝试另一行， $(-50, -200) - (-60, -240)$ 。显示了结果，图 6-4 显示了相应的特写镜头。

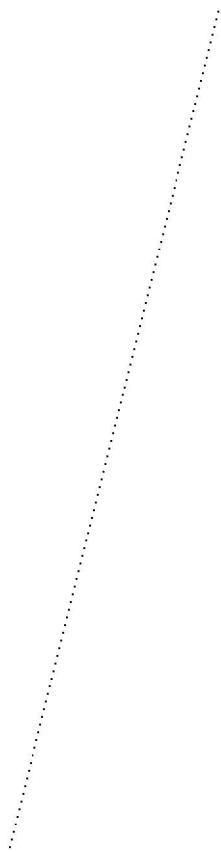


图 6-3：另一条斜率较高的直线

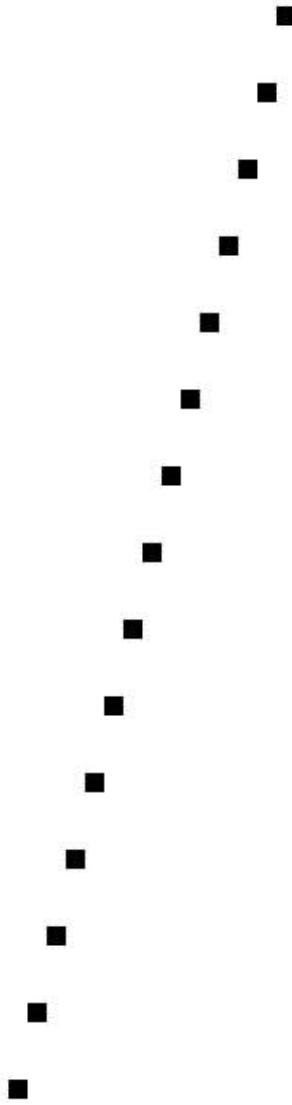


图 6-4: 放大第二条直线

哎呀。发生了什么事?

该算法完全按照我们的要求去做;它从左到右,计算出一个值 y 对于 x ,并绘制了相应的像素。问题是它计算了一个值 y 对于 x ,而在这种情况下,我们实际上需要几个值 y 对于某些值 x .

发生这种情况是因为我们选择了一种配方,其中 $y = f(x)$;事实上,这与我们无法绘制垂直线的原因相同 - 这是一个极端情况,其中所有值 y 对应于相同的值 x .

绘制具有任何坡度的线

选择 $y = f(x)$ 是一个任意的选择;我们同样可以选择将这条线表示为 $x = f(y)$.通过交换重新计算所有方程 x 和 y ,我们得到以下算法:

```
DrawLine(P0, P1, color) {
    // Make sure y0 < y1
    if y0 > y1 {
        swap(P0, P1)
    }
    a = (x1 - x0) / (y1 - y0)
```

```

x = x0
for y = y0 to y1 {
    canvas.PutPixel(x, y, color)
    x = x + a
}
}

```

这与前面相同，除了`DrawLine`x和y计算已交换。这个可以处理垂直线并会绘制 $(0, 0) - (50, 100)$ 正确，但是当然，它根本无法处理水平或绘制 $(0, 0) - (100, 50)$ 怎么办呢？

我们可以保留函数的两个版本，并根据我们尝试绘制的线选择使用哪个版本。标准很简单：该行是否具有更多不同的值x比不同的值y？如果有更多值x比y，我们使用第一个版本；否则，我们使用第二个。

示例 6-1 显示了处理所有情况的版本`DrawLine`

```

DrawLine(P0, P1, color) {
    dx = x1 - x0
    dy = y1 - y0
    if abs(dx) > abs(dy) {
        // Line is horizontal-ish
        // Make sure x0 < x1
        if x0 > x1 {
            swap(P0, P1)
        }
        a = dy/dx
        y = y0
        for x = x0 to x1 {
            canvas.PutPixel(x, y, color)
            y = y + a
        }
    } else {
        // Line is vertical-ish
        // Make sure y0 < y1
        if y0 > y1 {
            swap(P0, P1)
        }
        a = dx/dy
        x = x0
        for y = y0 to y1 {
            canvas.PutPixel(x, y, color)
            x = x + a
        }
    }
}

```

示例 6-1：一个处理所有情况的版本`DrawLine`

这当然有效，但它并不漂亮。有大量的代码重复，选择要使用的函数的逻辑、计算函数值的逻辑以及像素绘制本身都是交织在一起的。当然，我们可以做得更好！

线性插值函数

我们有两个线性函数 $y = f(x)$ 和 $x = f(y)$ 。为了抽象出我们正在处理像素的事实，让我们以更通用的方式将其编写为 $d = f(i)$ ，其中 i 是自变量，我们为其选择值的自变量，并且 d 是因变量，其值取决于另一个变量并且我们要计算的变量。在水平情况下 x 是自变量，并且 y 是因变量；在垂直情况下，情况正好相反。

当然，任何函数都可以写成 $d = f(i)$ 我们知道另外两件事可以完全定义我们的函数：它是线性的，以及它的两个值，即： $d_0 = f(i_0)$ 和 $d_1 = f(i_1)$ 我们可以编写一个简单的函数来获取这些值并返回所有中间值的列表 d ，假设和以前一样 $i_0 < i_1$

```
Interpolate (i0, d0, i1, d1) {
    values = []
    a = (d1 - d0) / (i1 - i0)
    d = d0
    for i = i0 to i1 {
        values.append(d)
        d = d + a
    }
    return values
}
```

此函数与 的前两个版本具有相同的“形状”，但调用变量而不是 和，而不是绘制像素，而是将值存储在列表中。DrawLineidx

请注意，值 d 对应于 i_0 在 中返回，则 $values[0]$ 在 中，依此类推；通常 i_n 的值为 $values[1]$ ，在 $values[i_n - i_0]$ 中返回，假设 $values[i_n - i_0]$ 在 范围内 $[i_0 - i_1]$

我们需要考虑一个极端情况：我们可能需要计算 $d = f(i)$ 对于单个值 i 也就是说，当 $i_0 = i_1$ 在这种情况下，我们甚至无法计算 a ，此我们将其视为特例：

```
Interpolate (i0, d0, i1, d1) {
    if i0 == i1 {
        return [d0]
    }
    values = []
    a = (d1 - d0) / (i1 - i0)
    d = d0
    for i = i0 to i1 {
        values.append(d)
        d = d + a
    }
    return values
}
```

作为实现细节，在本书的其余部分，自变量的值 i 终是整数，因为它们表示像素，而因变量的值 d 始终是浮点值，因为它们表示通用线性函数的值。

现在我们可以使用 .DrawLineInterpolate

```
DrawLine(P0, P1, color) {
    if abs(x1 - x0) > abs(y1 - y0) {
        // Line is horizontal-ish
        // Make sure x0 < x1
        if x0 > x1 {
            swap(P0, P1)
        }
        ys = Interpolate(x0, y0, x1, y1)
        for x = x0 to x1 {
            canvas.PutPixel(x, ys[x - x0], color)
        }
    } else {
        // Line is vertical-ish
        // Make sure y0 < y1
        if y0 > y1 {
            swap(P0, P1)
        }
    }
}
```

```

xs = Interpolate(y0, x0, y1, x1)
for y = y0 to y1 {
    canvas.PutPixel(xs[y - y0], y, color)
}
}

```

示例 6-2：一个使用 DrawLineInterpolate

这可以正确处理所有情况（图 6-5）。DrawLine

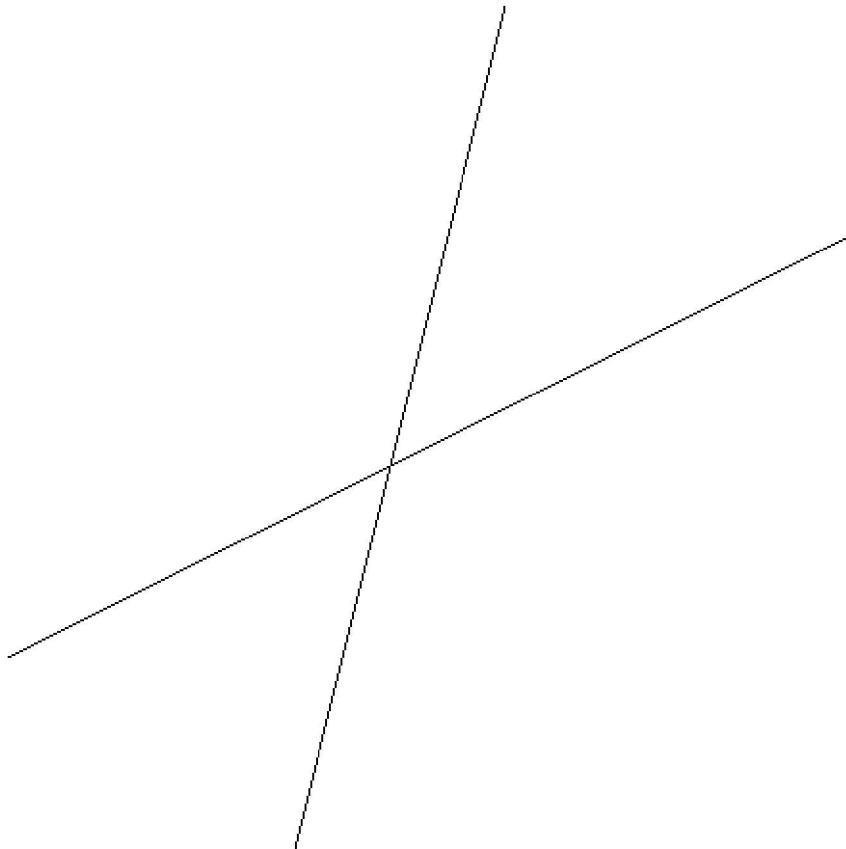


图 6-5：重构算法正确处理所有情况。

[源代码和现场演示>>](#)

虽然这个版本并不比前一个版本短多少，但它干净地分离了中间值的计算 y 和 x 从哪个决定是自变量和像素绘制代码本身。

令人惊讶的是，这种行算法不是最好的或最快的；这种区别可能属于布雷森纳姆算法。提出这种算法的原因有两个。首先，它更容易理解，这是本书的首要原则。其次，它给了我们这个功能，我们将在本书的其余部分广泛使用。Interpolate

总结

在本章中，我们迈出了构建光栅器的第一步。使用我们唯一的工具，我们开发了一种可以在画布上绘制直线段的算法。PutPixel

我们还开发了辅助方法，这是一种有效计算线性函数值的方法。在继续之前，请确保您很好地理解了它，因为我们将大量使用它。Interpolate

在下一章中，我们将用于在画布上绘制更复杂和有趣的形状：三角形。Interpolate