

基本光线追踪

在本章中，我们将介绍光线追踪，这是我们将介绍的第一个主要算法。我们首先激励算法并布置一些基本的伪代码。然后，我们看看如何在场景中表示光线和物体。最后，我们推导出一种方法来计算哪些光线构成了场景中每个对象的可见图像，并了解如何在画布上表示它们。

渲染瑞士风景

假设您正在访问某个异国情调的地方，并遇到令人惊叹的风景 - 如此令人惊叹，您只需要制作一幅捕捉其美丽的画。图 2-1 显示了这样一个景观。



图2-1：令人叹为观止的瑞士风景

你有画布和画笔，但你绝对缺乏艺术天赋。所有的希望都失去了吗？

不一定。你可能没有艺术天赋，但你是有条不紊的。所以你做了最明显的事情：你得到了一个防虫网。你切一个矩形的碎片，把它框起来，然后固定在一根棍子上。现在，您可以通过网状窗户查看风景。接下来，您选择最佳视角来欣赏这种景观，并种植另一根棍子来标记您的眼睛应该在的确切位置。

你还没有开始画，但现在你有一个固定的视角和一个固定的框架，通过它你可以看到风景。而且，这个固定的框架被昆虫网分成小方块。现在是有条不紊的部分。你在画布上画一个网格，给它与昆虫网相同数量的正方形。然后你看网的左上角正方形。你能透过它看到的主要颜色是什么？天蓝色。所以你把画布的左上角的方块涂成蓝色。你对每个正方形都这样做，很快画布就包含了一幅很好的风景画，就像通过框架看到的那样。所得绘画如图2-2所示。



图 2-2：景观的粗略近似值

当你想到它时，计算机本质上是一个非常有条理的机器，绝对缺乏艺术天赋。我们可以这样描述创作绘画的过程：

For each little square on the canvas
Paint it the right color

容易！但是，该公式过于抽象，无法直接在计算机上实现。我们可以更详细地介绍一下：

Place the eye and the frame as desired
For each square on the canvas
Determine which square on the grid corresponds to this square on the canvas
Determine the color seen through that grid square
Paint the square with that color

这仍然太抽象了，但它开始看起来像一个算法——也许令人惊讶的是，这是对完整光线跟踪算法的高级概述！是的，就是这么简单。

基本假设

计算机图形学的部分魅力在于在屏幕上绘制事物。为了尽快实现这一目标，我们将做出一些简化的假设。当然，这些假设对我们能做的事情施加了一些限制，但我们在后面的章节中取消这些限制。

首先，我们将假设一个固定的观看位置。观看位置，即您在瑞士风景类比中将目光放在位置的位置，通常称为相机位置；我们称之为 O 。我们假设相机占据空间中的单个点，它位于坐标系的原点，并且它永远不会从那里移动，因此 $O = (0, 0, 0)$

其次，我们将假设相机方向固定。相机方向决定了相机指向的位置。我们假设它朝正的方向看 Z 轴（我们将缩短为 \vec{Z}_+ ），正 Y 轴 (\vec{Y}_+) 向上和正 X 轴 (\vec{X}_+) 到右侧（图 2-3）。

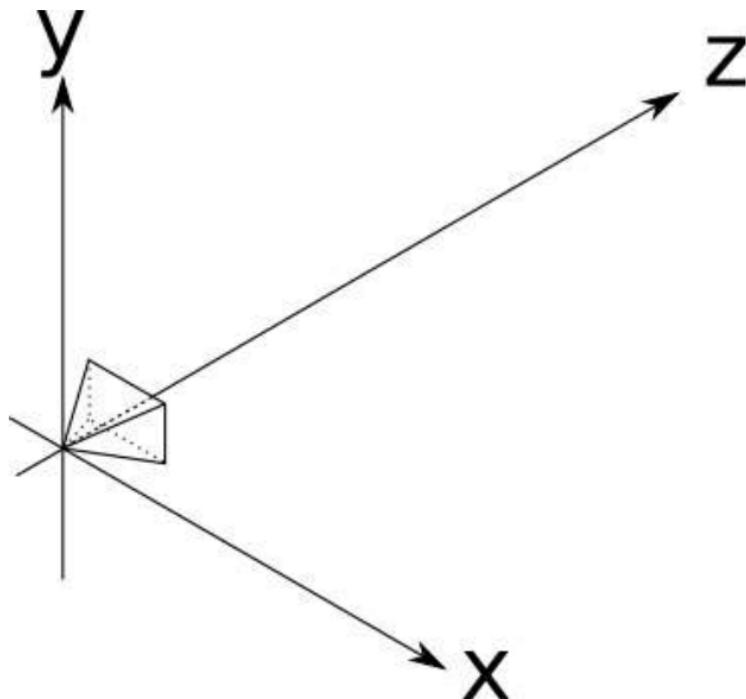


图 2-3：相机的位置和方向

相机位置和方向现已固定。类比中仍然缺少我们观察场景的“框架”。我们假设这个框架有尺寸 V_w 和 V_h ，并且正对摄像机方向，即垂直于 \vec{Z}_+ 。我们还假设它在远处 d ，其侧面平行于 X 和 Y 轴，并且它相对于 \vec{Z} 。这句话很拗口，但实际上很简单。请看图 2-4。

将充当我们世界窗口的矩形称为视口。从本质上讲，我们将在画布上绘制通过视口看到的任何内容。请注意，视口的大小和到摄像机的距离决定了摄像机的可见角度，称为视野，简称为 FOV。人类几乎 180° 水平视场（尽管其中大部分是模糊的周边视觉，没有深度感）。为简单起见，我们将设置 $V_w = V_h = d = 1$ ；这导致 FOV 大约 53° ，从而产生看起来合理且不会过度失真的图像。

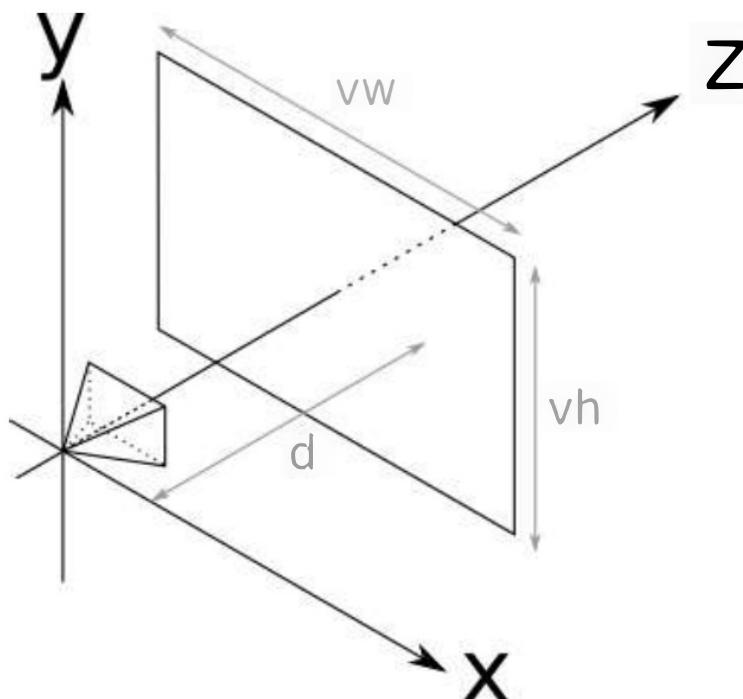


图 2-4: 视口的位置和方向

让我们回到前面介绍的“算法”，使用适当的技术术语，并对示例 2-1 中的步骤进行编号：

- ① Place the camera and the viewport as desired
For each pixel on the canvas
- ② Determine which square on the viewport corresponds to this pixel
- ③ Determine the color seen through that square
- ④ Paint the pixel with that color

示例 2-1：光线追踪算法的高级描述

我们刚刚完成了步骤①（或者，更准确地说，暂时将其排除在外）。步骤③是微不足道的：我们简单地使用`canvas.PutPixel(x, y, color)`快速完成步骤②，然后在接下来的几章中将注意力集中在越来越复杂的步骤③方法上。

画布到视口

示例 2-1 中算法的步骤②要求我们。我们知道像素的画布坐标 - 我们称之为 `Determine which square on the viewport corresponds to this pixel` C_x 和 C_y 。请注意我们如何方便地放置视口，以便其轴与画布的方向匹配，并且其中心与画布的中心匹配。因为视口以世界单位测量，画布以像素为单位，所以从画布坐标到空间坐标只是比例的变化！

$$V_x = C_x \cdot \frac{V_w}{C_w}$$

$$V_y = C_y \cdot \frac{V_h}{C_h}$$

还有一个额外的细节。虽然视口是 2D 的，但它嵌入在 3D 空间中。我们将其定义为远距离 d 从相机。根据定义，该平面（称为投影平面）中的每个点都具有 $z = d$ 。因此

$$V_z = d$$

我们已经完成了这一步。对于每个像素(C_x, C_y)在画布上，我们可以确定其在视口上的对应点(V_x, V_y, V_z).

追踪光线

下一步是弄清楚光线通过什么颜色(V_x, V_y, V_z)是，从相机的角度来看(O_x, O_y, O_z).

在现实世界中，光线来自光源（太阳、灯泡等），从几个物体上反弹，然后最终到达我们的眼睛。我们可以尝试模拟每个光子离开模拟光源的路径，但这非常耗时。我们不仅要模拟数量惊人的光子（单个100 W灯泡发射 10^{20} 每秒光子！），其中只有极少数碰巧达到(O_x, O_y, O_z)通过视口后。这种技术称为光子追踪或光子映射；不幸的是，这超出了本书的范围。

相反，我们将考虑“反向”的光线；我们将从来自摄像机的光线开始，穿过视口中的点，并跟踪其路径，直到它击中场景中的某个对象。此对象是摄像机通过视口的该点“看到”的对象。因此，作为第一个近似值，我们只需将该物体的颜色视为“通过该点的光的颜色”，如图 2-5 所示。

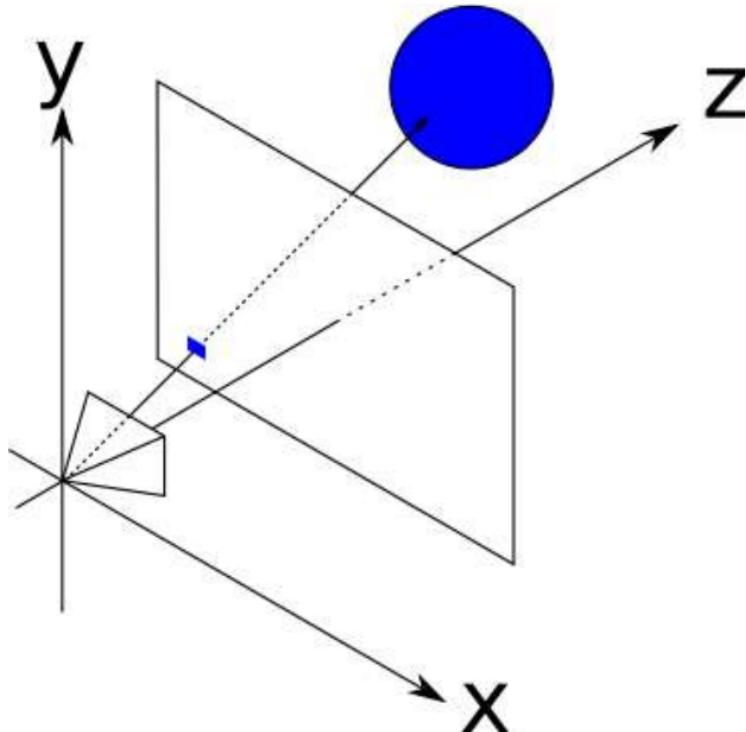


图 2-5：视口中的一个小方块，代表画布中的单个像素，绘制有摄像机通过它看到的对象的颜色

现在我们只需要一些方程式。

射线方程

就我们的目的而言，表示射线的最方便方法是使用参数方程。我们知道光线通过 O ，我们知道它的方向（从 O 自 V ），因此我们可以将射线中的任何点 P 表示为

$$P = O + t(V - O)$$

哪里 t 是任何实数。通过插入每个值 t 从 $-\infty$ 自 $+\infty$ 在这个等式中，我们得到每一分 P 沿着射线。

让我们打电话($V - O$)，光线的方向， \vec{D} .等式变为

$$P = O + t\vec{D}$$

理解这个方程的一种直观方法是，我们在原点(O)并沿射线方向“前进”(\vec{D})按一定量(t)很容易看出，这包括沿射线的所有点。您可以在线性代数附录中阅读有关这些向量运算的更多详细信息。图 2-6 显示了我们的等式。

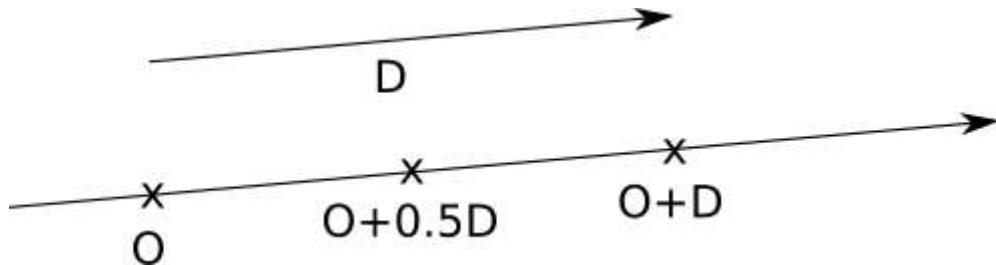


图 2-6：射线 $O + t\vec{D}$ 的一些点对于不同的 t 值。

图 2-6 显示了沿射线的点，对应于 $t = 0.5$ 和 $t = 1.0$ 。每个值 t 沿射线产生不同的点。

球体方程

现在我们需要在场景中有某种物体，以便我们的光线可以击中某些东西。我们可以选择任意几何基元作为场景的构建块；对于光线追踪，我们将使用球体，因为它们很容易用方程式操作。

什么是球体？球体是与固定点保持固定距离的点集。该距离称为球体的半径，该点称为球体的中心。图 2-7 显示了一个球体，由其中心定义 C 及其半径 r 。

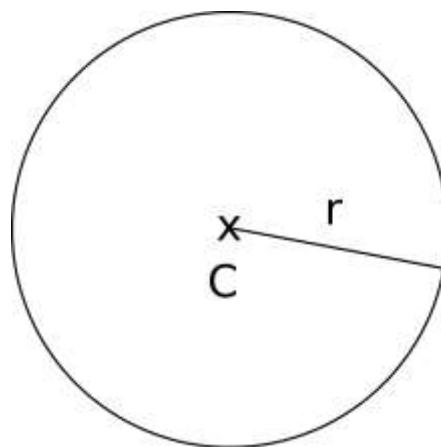


图 2-7：由其中心和半径定义的球体

根据我们上面的定义，如果 C 是中心和 r 是球体的半径，点 P 在该球体的表面上必须满足以下等式：

$$dInstance(P, C) = r$$

让我们玩一下这个等式。如果您发现这些数学中的任何一个不熟悉，请通读线性代数附录。

之间的距离 P 和 C 是向量的长度 P 自 C :

$$|P - C| = r$$

向量的长度 (表示 $|\vec{V}|$) 是其点积与自身的平方根 (表示为 $\langle \vec{V}, \vec{V} \rangle$):

$$\sqrt{\langle P - C, P - C \rangle} = r$$

为了摆脱平方根，我们可以对两边进行平方：

$$\langle P - C, P - C \rangle = r^2$$

球体方程的所有这些公式都是等价的，但最后一个公式在以下步骤中最方便操作。

射线遇见球体

我们现在有两个方程：一个描述球体上的点，另一个描述射线上的点：

$$\langle P - C, P - C \rangle = r^2$$

$$P = O + t\vec{D}$$

射线和球体相交吗？如果是，在哪里？

假设射线和球体确实在某一点相交 P .该点既沿着射线又在球体表面，因此它必须同时满足两个方程。请注意，这些方程中的唯一变量是参数 t 因为 O, \vec{D}, C 和 r 被给予和 P 是我们试图找到的点。

因为 P 表示两个方程中的同一点，我们可以代入 P 在第一个表达式中，表达式为 P 在第二个。这给了我们

$$\langle O + t\vec{D} - C, O + t\vec{D} - C \rangle = r^2$$

如果我们能找到 t 为了满足这个方程，我们可以将它们放在射线方程中，以找到光线与球体相交的点。

以目前的形式，这个等式有些笨拙。让我们做一些代数操作，看看我们能从中得到什么。

首先，让 $\vec{CO} = O - C$.然后我们可以把等式写成

$$\langle \vec{CO} + t\vec{D}, \vec{CO} + t\vec{D} \rangle = r^2$$

然后，我们使用其分布属性将点积扩展为其组件（再次，请随时查阅线性代数附录）：

$$\langle \vec{CO} + t\vec{D}, \vec{CO} \rangle + \langle \vec{CO} + t\vec{D}, t\vec{D} \rangle = r^2$$

$$\langle \vec{CO}, \vec{CO} \rangle + \langle t\vec{D}, \vec{CO} \rangle + \langle \vec{CO}, t\vec{D} \rangle + \langle t\vec{D}, t\vec{D} \rangle = r^2$$

稍微重新排列术语，我们得到

$$\langle t\vec{D}, t\vec{D} \rangle + 2\langle \vec{CO}, t\vec{D} \rangle + \langle \vec{CO}, \vec{CO} \rangle = r^2$$

移动参数 t 走出点积和移动 r^2 等式的另一边给了我们

$$t^2 \langle \vec{D}, \vec{D} \rangle + t(2\langle \vec{CO}, \vec{D} \rangle) + \langle \vec{CO}, \vec{CO} \rangle - r^2 = 0$$

请记住，两个向量的点积是一个实数，因此尖括号之间的每个项都是实数。如果我们给它们起名字，我们会得到更熟悉的东西：

$$a = \langle \vec{D}, \vec{D} \rangle$$

$$b = 2\langle \vec{CO}, \vec{D} \rangle$$

$$c = \langle \vec{CO}, \vec{CO} \rangle - r^2$$

$$at^2 + bt + c = 0$$

这只不过是一个古老的二次方程！其解决方案是参数的值 t 光线与球体相交的地方：

$$\{t_1, t_2\} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

幸运的是，这在几何上是有意义的。您可能还记得，二次方程可以没有解、一个双解或两个不同的解，具体取决于判别式的值 $b^2 - 4ac$ 。这与光线不与球体相交、光线与球体相切以及光线分别进入和离开球体的情况完全对应（图 2-8）。

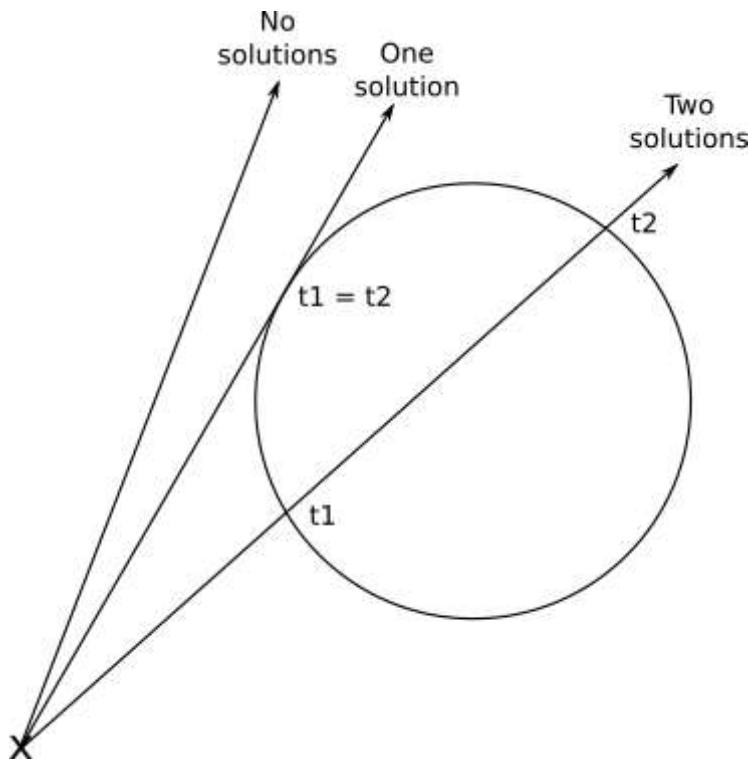


图 2-8：二次方程解的几何解释：无解、一个解或两个解。

一旦我们找到了 t , 我们可以把它代回射线方程, 我们终于得到了交点 P 对应于该值 t .

渲染我们的第一个球体

回顾一下, 对于画布上的每个像素, 我们可以计算视口上的相应点。给定相机的位置, 我们可以表示从相机开始并穿过视口该点的光线方程。给定一个球体, 我们可以计算光线与该球体相交的位置。

因此, 我们需要做的就是计算光线和每个球体的交点, 保持交点最靠近相机, 并在画布上绘制适当的颜色。我们几乎准备好渲染我们的第一个球体了!

参数 t 不过, 值得一些额外的关注。让我们回到射线方程:

$$P = O + t(V - O)$$

由于射线的原点和方向是固定的, 因此变化很大 t 在所有实数中将产生每个点 P 在这射线中。请注意, 对于 $t = 0$ 我们得到 $P = O$, 以及 $t = 1$ 我们得到 $P = V$.负值 t 让步点朝相反的方向, 即在相机后面。因此, 我们可以将参数空间分为三个部分, 如表 2-1 所示。图 2-9 显示了参数空间的示意图。

表 2-1：参数空间的细分

$t < 0$	镜头背后
$0 \leq t \leq 1$	在摄像机和投影平面/视口之间
$t > 1$	在投影平面/视口前面

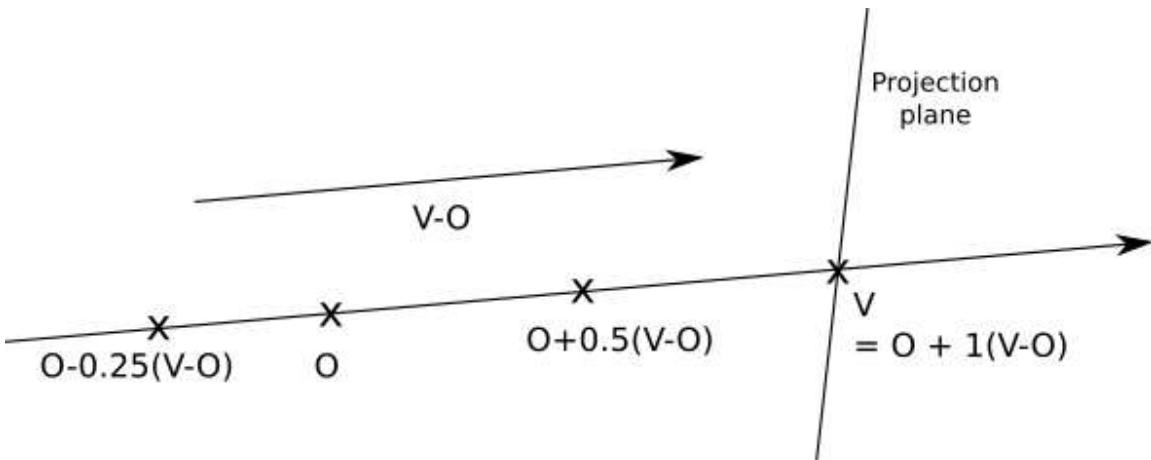


图 2-9：参数空间中的几个点

请注意，相交方程中没有任何内容表明球体必须在相机前面；该方程将愉快地为相机后面的交叉点生成解决方案。显然，这不是我们想要的，所以我们应该忽略任何解决方案 $T < 0$ 。为了避免进一步的数学不愉快，我们将解决方案限制为 $T > 1$ ；也就是说，我们将渲染投影平面之外的任何内容。

另一方面，我们不想对 t ；我们希望看到相机前的所有物体，无论它们有多远。但是，因为在后期阶段，我们将希望缩短光线，我们现在将引入这种形式主义并给出 t 上限值为 $+\infty$ （对于不能直接表示“无穷大”的语言，一个非常大的数字就可以了）。

我们现在可以用一些伪代码形式化我们到目前为止所做的一切。作为一般规则，我们假设代码可以访问它需要的任何数据，因此我们不会费心显式传递参数（如画布），而是专注于真正必要的参数。

主方法现在类似于示例 2-2。

```

O = (0, 0, 0)
for x = -Cw/2 to Cw/2 {
    for y = -Ch/2 to Ch/2 {
        D = CanvasToViewport(x, y)
        color = TraceRay(O, D, 1, inf)
        canvas.PutPixel(x, y, color)
    }
}

```

示例 2-2：主要方法

该函数非常简单，如示例 2-3 所示。常量表示相机和投影平面之间的距离。

CanvasToViewportd

```

CanvasToViewport(x, y) {
    return (x*Vw/Cw, y*Vh/Ch, d)
}

```

示例 2-3：函数 CanvasToViewport

该方法（示例 2-4）计算射线与每个球体的交点，并返回球体在请求范围内的最近交点处的颜色 $\text{TraceRay}(t)$ 。

```

TraceRay(O, D, t_min, t_max) {
    closest_t = inf
}

```

```

closest_sphere = NULL
for sphere in scene.spheres {
    t1, t2 = IntersectRaySphere(0, D, sphere)
    if t1 in [t_min, t_max] and t1 < closest_t {
        closest_t = t1
        closest_sphere = sphere
    }
    if t2 in [t_min, t_max] and t2 < closest_t {
        closest_t = t2
        closest_sphere = sphere
    }
}
if closest_sphere == NULL {
    ①return BACKGROUND_COLOR
}
return closest_sphere.color
}

```

示例 2-4：方法TraceRay

在示例 2-4 中，表示射线的起源；虽然我们正在追踪来自放置在原点的相机的光线，但在后期阶段不一定是这种情况，因此它必须是一个参数。这同样适用于 和。

0t_min_t_max

请注意，当光线不与任何球体相交时，我们仍然需要返回一些颜色 ① - 我在大多数示例中都选择了白色。

最后，（示例 2-5）只是求解二次方程。IntersectRaySphere

```

IntersectRaySphere(0, D, sphere) {
    r = sphere.radius
    C0 = 0 - sphere.center

    a = dot(D, D)
    b = 2*dot(C0, D)
    c = dot(C0, C0) - r*r

    discriminant = b*b - 4*a*c
    if discriminant < 0 {
        return inf, inf
    }

    t1 = (-b + sqrt(discriminant)) / (2*a)
    t2 = (-b - sqrt(discriminant)) / (2*a)
    return t1, t2
}

```

示例 2-5：方法IntersectRaySphere

为了将所有这些付诸实践，让我们定义一个非常简单的场景，如图 2-10 所示。

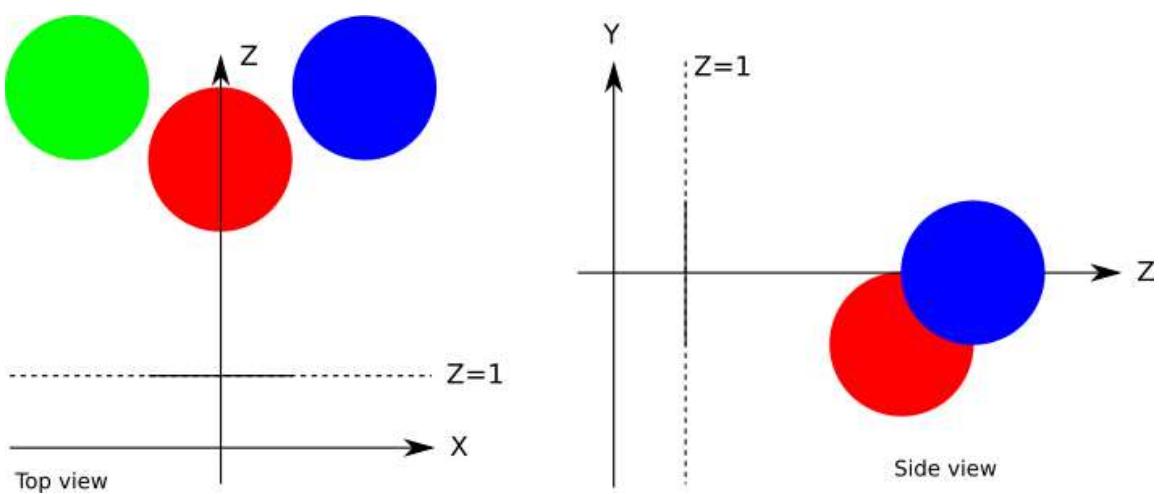


图 2-10: 一个非常简单的场景，从上方（左）和右侧（右）查看

在伪场景语言中，它是这样的：

```
viewport_size = 1 x 1
projection_plane_d = 1
sphere {
    center = (0, -1, 3)
    radius = 1
    color = (255, 0, 0) # Red
}
sphere {
    center = (2, 0, 4)
    radius = 1
    color = (0, 0, 255) # Blue
}
sphere {
    center = (-2, 0, 4)
    radius = 1
    color = (0, 255, 0) # Green
}
```

当我们在这个场景上运行我们的算法时，我们最终得到了一个令人难以置信的令人敬畏的光线追踪场景的奖励（图 2-11）。

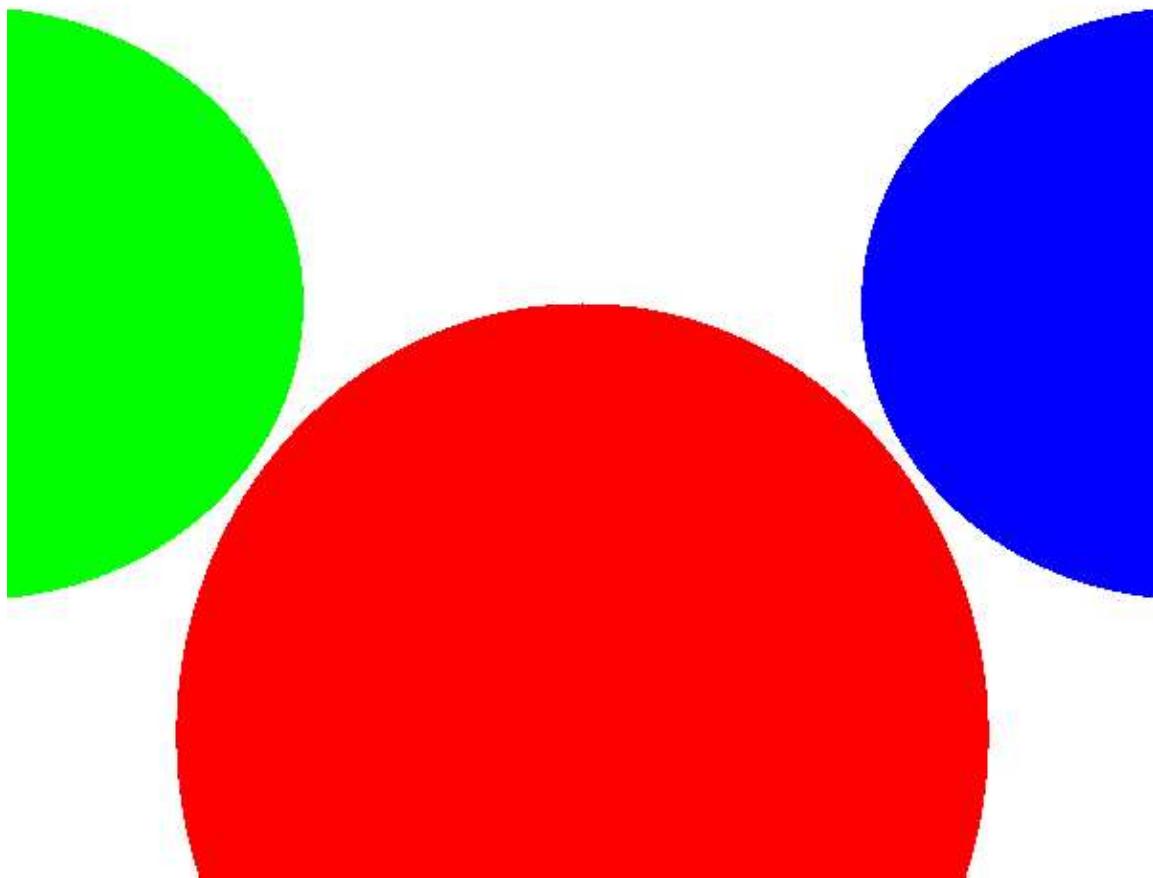


图 2-11：令人难以置信的光线追踪场景

[源代码和现场演示>>](#)

我知道，这有点令人失望，不是吗？反射、阴影和抛光的外观在哪里？别担心，我们会到达那里的。这是一个很好的第一步。球体看起来像圆圈，这比它们看起来像猫要好。它们看起来不太像球体的原因是，我们错过了人类如何确定物体形状的一个关键组成部分：它与光相互作用的方式。我们将在下一章中介绍这一点。

总结

在本章中，我们奠定了光线追踪器的基础。我们选择了一个固定的设置（摄像机和视口的位置和方向，以及视口的大小）；我们选择了球体和光线的表示形式；我们已经探索了弄清楚球体和光线如何相互作用所需的数学；我们将所有这些放在一起，用纯色在画布上绘制球体。

接下来的章节将以此为基础，对光线与场景中物体交互的方式进行建模，并越来越详细。