

# 阴影和反射

我们继续寻求以越来越逼真的方式渲染场景。在上一章中，我们模拟了光线与表面相互作用的方式。在本章中，我们将对光线与场景交互方式的两个方面进行建模：投射阴影的对象和反射在其他对象上的对象。

## 阴影

有灯光和物体的地方，就有阴影。我们有灯光和物体。那么我们的影子在哪里？

### 了解阴影

让我们从一个更基本的问题开始。为什么要~~有~~阴影？当光线无法到达物体时，就会发生阴影，因为有其他物体挡住了去路。

在上一章中，我们只关注光源和表面之间的局部交互，而忽略了场景中发生的所有其他事情。为了出现阴影，我们需要采取更全局的视角，并考虑光源、我们要绘制的表面以及场景中存在的其他对象之间的相互作用。

从概念上讲，我们要做的相对简单。我们想添加一点逻辑，说“如果点和光之间有一个物体，不要添加来自这个光的照明。”

我们要区分的两种情况如图 4-1 所示。

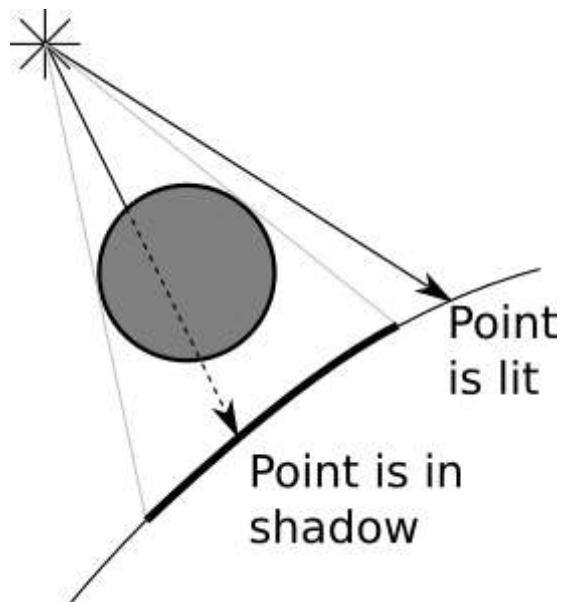


图 4-1：只要光源和该点之间有物体，阴影就会投射到该点上。

事实证明，我们已经拥有执行此操作所需的所有工具。让我们从定向光开始。我们知道  $P$ ; 这就是我们感兴趣的一点。我们知道  $\vec{L}$ ; 这是光定义的一部分。会意  $P$  和  $\vec{L}$ ，我们可以定义一条射线，即  $P + t\vec{L}$ ，即从表面上的点到无限远的光源。这条射线是否与任何其他物体相交？如果没有，则点和光之间没有任何内容，因此我们像以前一样计算该光的照明。如果是这样，则点在阴影中，因此我们忽略了来自此光的照明。

我们已经知道如何计算光线和球体之间的最近交点：我们用来追踪来自相机的光线的函数。我们可以重用其中的大部分来计算光线与场景其余部分之间的最近交点。

#### TraceRay

不过，此函数的参数略有不同：

- 光线不是从相机开始，而是从  $P$
- 射线的方向不是  $(V - O)$  但  $\vec{L}$
- 我们不希望后面有物体  $P$  在它上面投射阴影，所以我们需要  $t_{min} = 0$
- 由于我们处理的是无限远的定向光，因此非常远的物体仍然应该投射阴影。 $P$  所以  $t_{max} = +\infty$

图 4-2 显示了两点， $P_0$  和  $P_1$ . 追踪光线时  $P_0$  在光的方向上，我们发现与任何物体都没有交集；这意味着光线可以到达  $P_0$ ，所以上面没有阴影。在以下情况下： $P_1$ ，我们发现射线和球体之间有两个交点，其中  $T > 0$  (意思是相交点在表面和光之间)；因此，重点在阴影中。

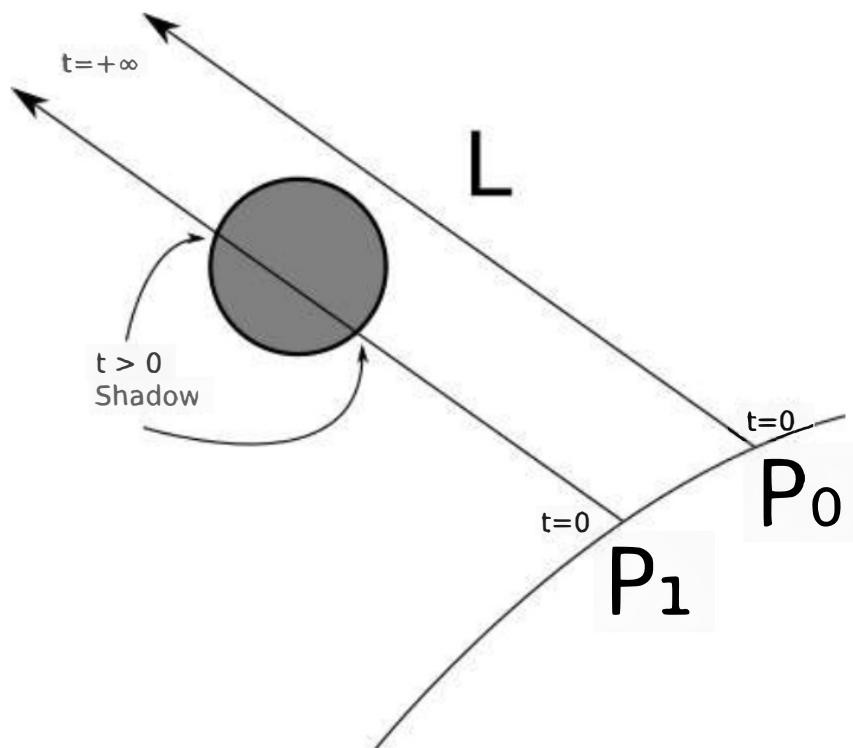


图 4-2: 球体在  $P_1$  上投射阴影，但不在  $P_0$  上投射阴影。

我们可以以非常相似的方式处理点光源，但有两个例外。第一  $\vec{L}$  不是恒定的，但我们已经知道如何从  $P$  以及光的位置。其次，我们不希望离光较远的物体能够投射阴影。 $P$ ，所以在这种情况下我们需要  $t_{max} = 1$  这样光线就会在光线下“停止”。

图 4-3 显示了这些情况。当我们投射光线  $P_0$  有方向  $L_0$ ，我们找到与小球体的交点；然而，这些有  $T > 1$ ，这意味着它们不在光和  $P_0$ ，所以我们忽略它们。因此  $P_0$  不在阴影中。另一方面，来自  $P_1$  有方向  $L_1$  与大球体相交  $0 < T < 1$ ，因此球体在上面投射出阴影  $P_1$ 。

我们需要考虑一个字面上的边缘情况。考虑射线  $P + t \vec{L}$ 。如果我们寻找从  $t_{min} = 0$ ，我们会在  $P$  本身！我们知道  $P$  在一个球体上，所以对于  $t = 0, P + 0 \vec{L} = P$ ；换句话说，每一点都会给自己投下阴影！

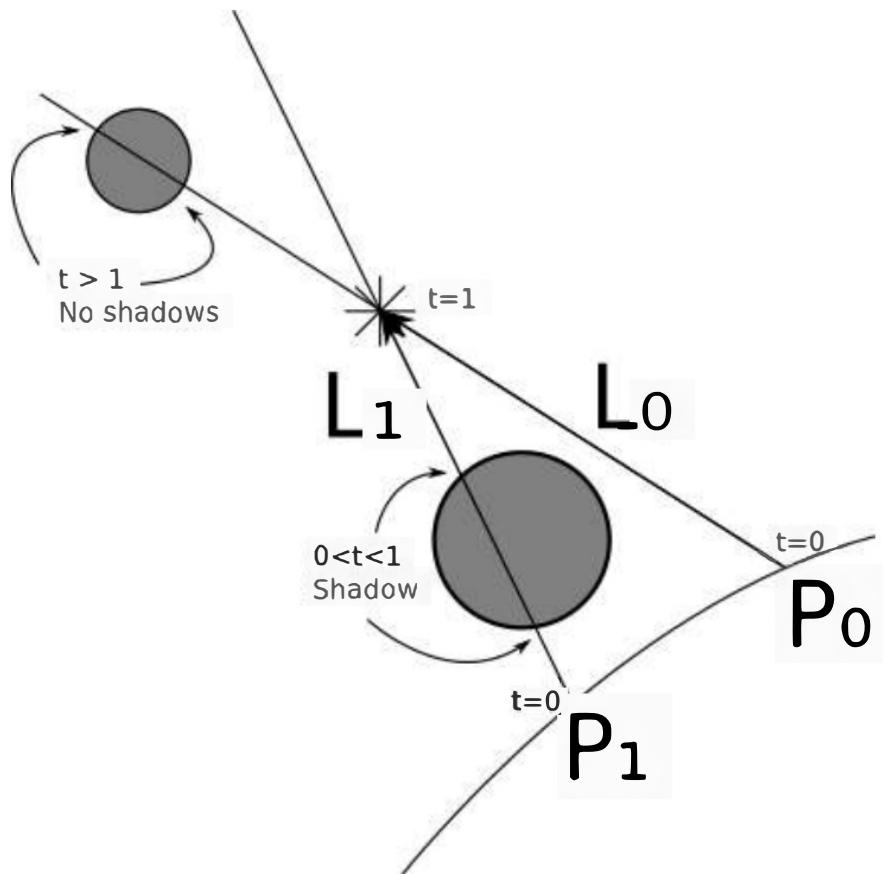


图 4-3：我们使用交叉点处的  $t$  值来确定它们是否在点上投射阴影。

最简单的解决方法是设置  $t_{min}$  到非常小的值  $\epsilon$  而不是 0。从几何上讲，我们说我们希望光线从表面开始一点点。 $P$  是，而不是完全在  $P$ 。所以范围将是  $[\epsilon, +\infty]$ 。用于定向灯和  $[\epsilon, 1]$  用于点光源。

通过不计算射线和球体之间的交点来解决这个问题可能很诱人  $P$  属于。这将适用于球体，但对于具有更复杂形状的对象会失败。例如，当你用手保护眼睛免受阳光照射时，你的手在你的脸上投下阴影，两个表面都是同一个物体的一部分 - 你的身体。

## 使用阴影渲染

让我们把上面的讨论变成伪代码。

在其以前的版本中，计算最近的射线球交点，然后计算交点上的照明。我们需要提取最近的交集代码，因为我们希望重用它来计算阴影（示例 4-1）。TraceRay

```
ClosestIntersection(0, 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
    }
}
return closest_sphere, closest_t
}

```

示例 4-1：计算最近的交集

我们可以重写以重用该函数，生成的版本要简单得多（示例 4-2）。TraceRay

```

TraceRay(0, D, t_min, t_max) {
    closest_sphere, closest_t = ClosestIntersection(0, D, t_min, t_max)
    if closest_sphere == NULL {
        return BACKGROUND_COLOR
    }
    P = 0 + closest_t * D
    N = P - closest_sphere.center
    N = N / length(N)
    return closest_sphere.color * ComputeLighting(P, N, -D, closest_sphere.specular)
}

```

示例 4-2：分解后的简化版本 TraceRayClosestIntersection

然后，我们需要将影子检查 ① 添加到（示例 4-3）。ComputeLighting

```

ComputeLighting(P, N, V, s) {
    i = 0.0
    for light in scene.Lights {
        if light.type == ambient {
            i += light.intensity
        } else {
            if light.type == point {
                L = light.position - P
                t_max = 1
            } else {
                L = light.direction
                t_max = inf
            }

            // Shadow check
① shadow_sphere, shadow_t = ClosestIntersection(P, L, 0.001, t_max)
            if shadow_sphere != NULL {
                continue
            }

            // Diffuse
            n_dot_l = dot(N, L)
            if n_dot_l > 0 {
                i += light.intensity * n_dot_l / (length(N) * length(L))
            }

            // Specular
            if s != -1 {
                R = 2 * N * dot(N, L) - L
                r_dot_v = dot(R, V)
                if r_dot_v > 0 {
                    i += light.intensity * pow(r_dot_v / (length(R) * length(V)), 8)
                }
            }
        }
    }
}

```

```
s)  
    }  
    }  
}  
return i  
}
```

示例 4-3: 支持影子ComputeLighting

图 4-4 显示了新渲染的场景的外观。

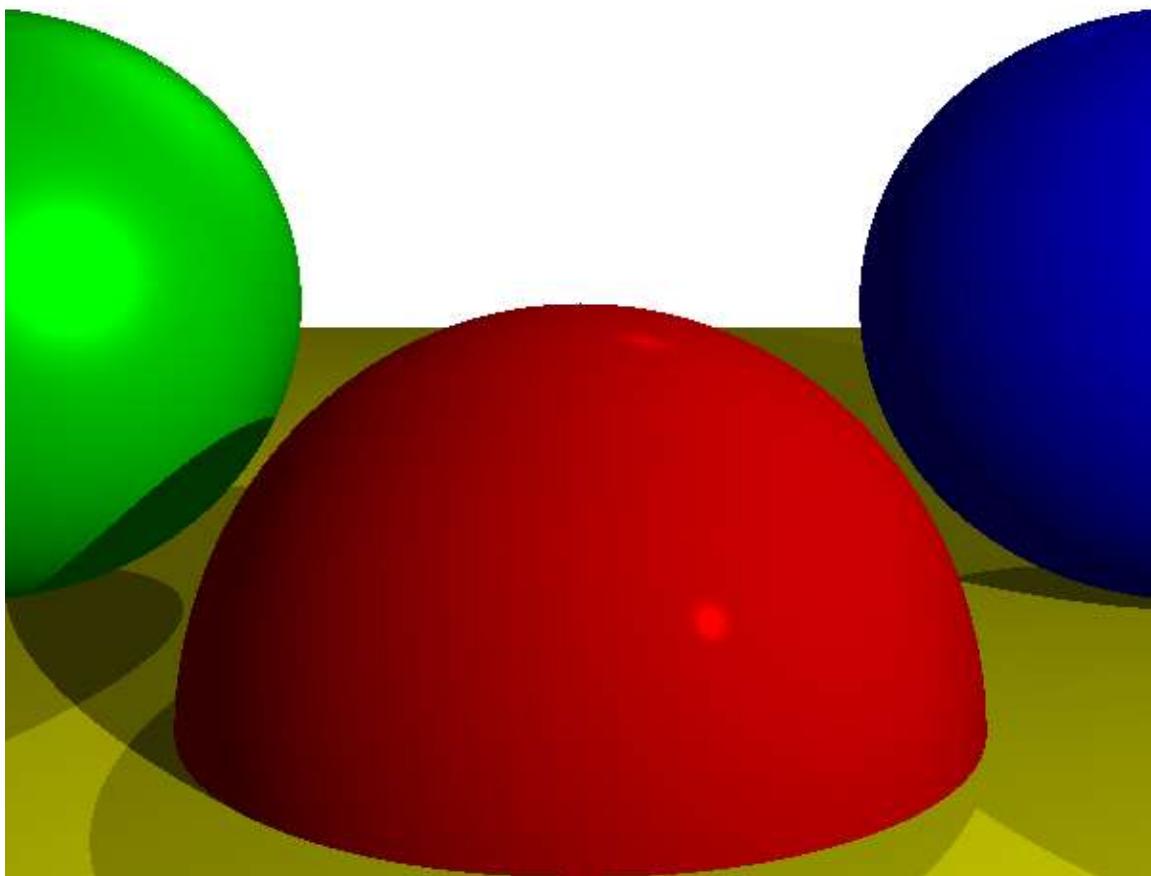


图 4-4: 光线追踪场景, 现在带有阴影

[源代码和现场演示>>](#)

现在我们到了某个地方。场景中的对象以更逼真的方式相互交互，相互投射阴影。接下来，我们将探索对象之间的更多交互，即反射其他对象的对象。

## 思考

在上一章中，我们谈到了“镜面状”的表面，但这只会使它们具有光泽的外观。我们能不能让物体看起来像真正的镜子——也就是说，我们能看到其他物体反射在它们的

表面上吗？我们可以，事实上，在光线追踪器中做到这一点非常简单，但当你第一次看到它是如何完成的时，它也可能会令人费解。

## 镜子和反射

让我们看看镜子是如何工作的。当你照镜子时，你看到的是从镜子上反射的光线。光线相对于表面法线对称反射，如图 4-5 所示。

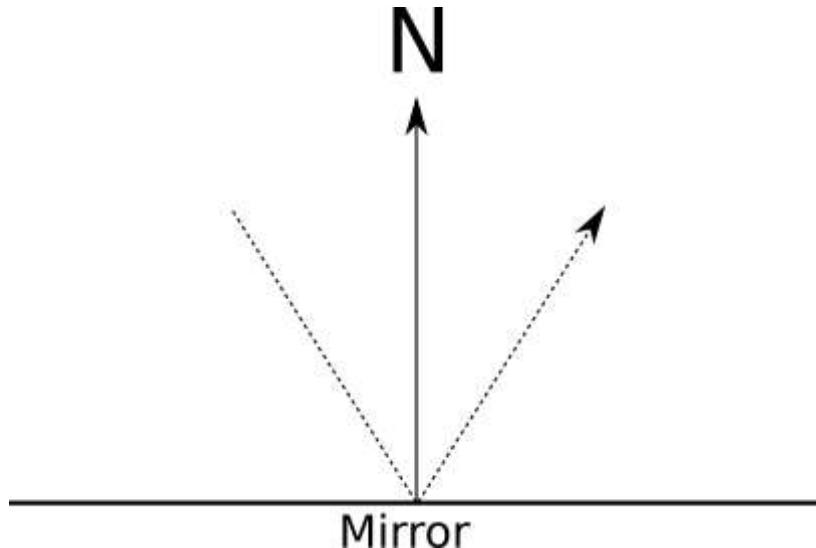


图 4-5：一束光线以与镜子法线对称的方向从镜子反射。

假设我们正在追踪一条光线，而最近的交点恰好是一面镜子。这束光是什么颜色的？这不是镜子本身的颜色，因为我们在看反射光。所以我们需要弄清楚这种光来自哪里，它是什么颜色。因此，我们所要做的就是计算反射光线的方向，并找出来自该方向的光的颜色。

如果我们有一个函数，给定一条光线，返回来自其方向的光的颜色.....

哦，等等！我们确实有一个，它叫！TraceRay

在主循环中，对于每个像素，我们创建一条从摄像机到场景的光线，然后我们调用以找出摄像机在该方向上“看到”的颜色。如果确定相机看到的是镜子，它只需要计算反射光线的方向并找出来自该方向的光的颜色；它必须调用。本身。TraceRayTraceRay

在这一点上，我建议你再读一遍最后几段，直到你明白为止。如果这是您第一次阅读有关递归光线跟踪的信息，则可能需要阅读几次并挠头，直到您真正理解它。

去吧，我会等——一旦这个美丽的啊哈！的兴奋开始减弱，让我们正式化一下。

当我们设计递归算法（调用自身的算法）时，我们需要确保不会导致无限循环（也称为“此程序已停止响应。是否要终止它？”）该算法有两个自然退出条件：当光线击中非反射物体时和当它没有击中任何东西时。但有一个简单的情况，我们可能会陷入无限循环：无限霍尔效应。当你把一面镜子放在另一个镜子面前，看着它时，就会发生这种情况——无限的你自己副本！

有许多方法可以防止无限递归。我们只在算法中引入递归限制；这将控制它的“深度”程度。我们称之为 $r$ 。什么时候 $r = 0$ ，我们看到了物体，但没有反射。什么时候 $r = 1$ ，我们看到物体和一些物体在其上的反射（图 4-6）。

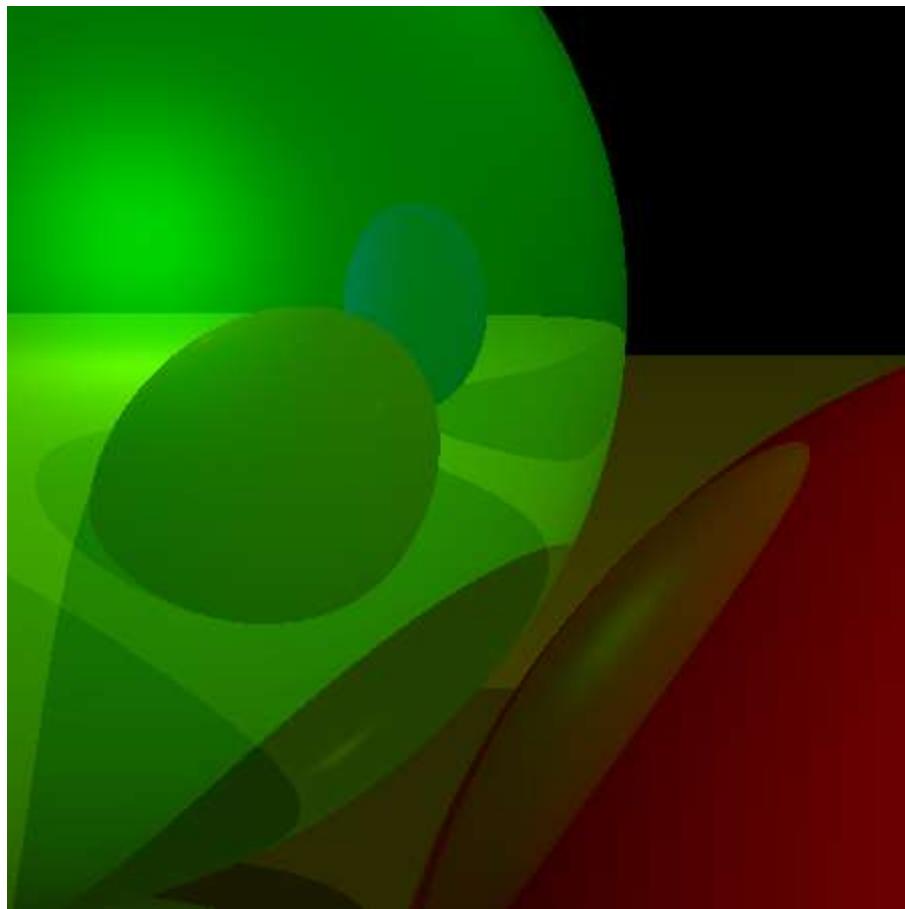


图 4-6: 反射仅限于一个递归调用 ( $r = 1$ )。我们看到球体反射在球体上，但反射球体本身看起来并不反射。

什么时候  $r = 2$ ，我们看到对象，某些对象的反射，以及某些对象的反射的反射（依此类推，对于更大的值  $r$ ）。图 4-7 显示了  $r = 3$ 。一般来说，超过三个级别没有多大意义，因为此时差异几乎不明显。

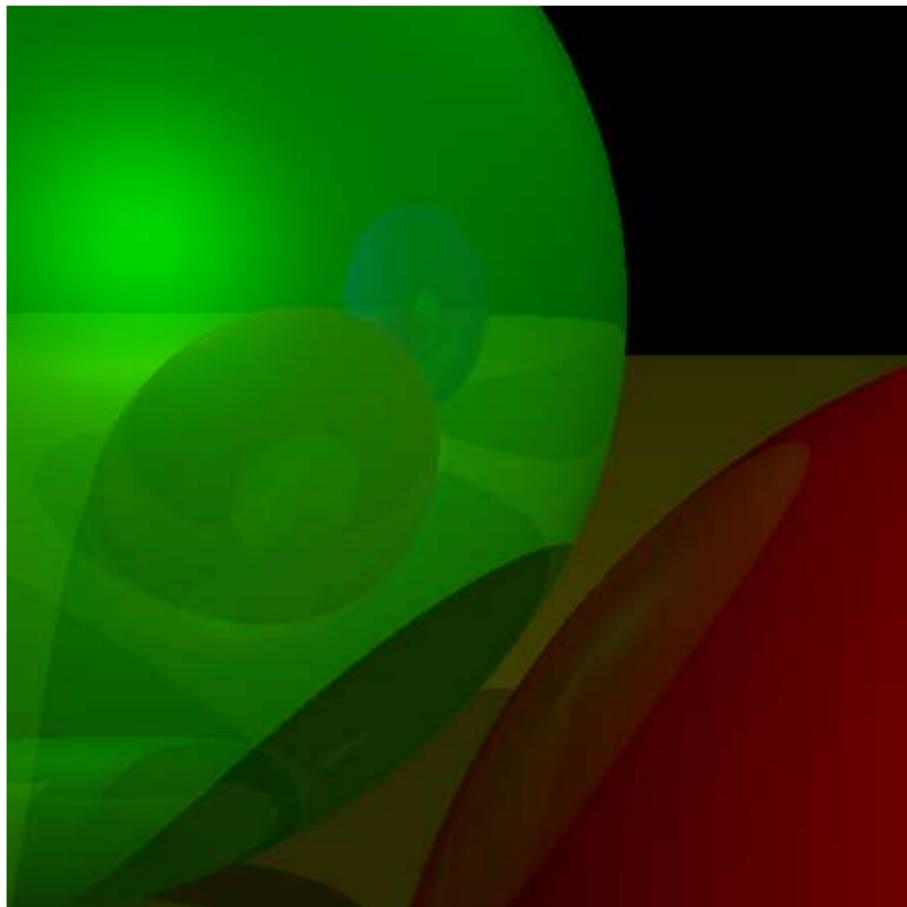


图 4-7: 反射仅限于三个递归调用 ( $r = 3$ )。现在我们可以看到球体反射的反射。

我们将做另一个区别。“反思性”不一定是一个全有或全无的命题;物体可能只有部分反射。我们将在0和1到每个表面，指定其反射程度。然后，我们将使用该数字作为权重来计算局部照明颜色和反射颜色的加权平均值。

最后，递归调用的参数是什么？`TraceRay`

- 光线从物体表面开始， $P$ .
- 反射光线的方向是入射光线反弹的方向 $P$ ;在`TraceRay`里，我们有 $\vec{D}$ ，入射光朝向的方向 $P$ ，所以反射光线的方向是 $-\vec{D}$ 反映在 $\vec{N}$ .
- 与阴影发生的情况类似，我们不希望物体反射自己，所以 $t_{min} = \epsilon$ .
- 我们希望看到物体被反射，无论它们有多远，所以 $t_{max} = +\infty$ .
- 递归限制比当前递归限制小 1 (以避免无限递归)。

现在我们已经准备好将其转换为实际的伪代码。

## 使用反射渲染

让我们向光线追踪器添加反射。首先，我们通过向每个表面添加一个属性来修改场景定义，描述它的反射程度，从 0.0 (完全不反射) 到 1.0 (完美镜子)：`reflective`

```
sphere {
    center = (0, -1, 3)
```

```

radius = 1
color = (255, 0, 0) # Red
specular = 500 # Shiny
reflective = 0.2 # A bit reflective
}
sphere {
    center = (-2, 0, 4)
    radius = 1
    color = (0, 0, 255) # Blue
    specular = 500 # Shiny
    reflective = 0.3 # A bit more reflective
}
sphere {
    center = (2, 0, 4)
    radius = 1
    color = (0, 255, 0) # Green
    specular = 10 # Somewhat shiny
    reflective = 0.4 # Even more reflective
}
sphere {
    color = (255, 255, 0) # Yellow
    center = (0, -5001, 0)
    radius = 5000
    specular = 1000 # Very shiny
    reflective = 0.5 # Half reflective
}

```

在计算镜面反射时，我们已经使用了“反射光线”公式，因此我们可以将其分解掉。它需要射线  $\vec{R}$  和正常  $\vec{N}$  和退货  $\vec{R}$  反映在  $\vec{N}$ 。

```

ReflectRay(R, N) {
    return 2 * N * dot(N, R) - R;
}

```

我们需要做的唯一更改是将反射方程替换为对这个新 .ComputeLightingReflectRay  
main 方法有一个小的变化——我们需要将递归限制传递给顶级调用：TraceRay

```
color = TraceRay(0, D, 1, inf, recursion_depth)
```

我们可以将 的初始值设置为合理的值，例如 3，如前所述。recursion\_depth

唯一的重大变化发生在 的末尾，我们以递归方式计算反射。您可以在示例 4-4 中看到  
更改。TraceRay

```

TraceRay(0, D, t_min, t_max, recursion_depth) {
    closest_sphere, closest_t = ClosestIntersection(0, D, t_min, t_max)

    if closest_sphere == NULL {
        return BACKGROUND_COLOR
    }

    // Compute local color
    P = 0 + closest_t * D
    N = P - closest_sphere.center
    N = N / length(N)
    local_color = closest_sphere.color * ComputeLighting(P, N, -D, closest_sphere.s
pecular)

    // If we hit the recursion limit or the object is not reflective, we're done
    ❶ r = closest_sphere.reflective
}

```

```

if recursion_depth <= 0 or r <= 0 {
    return local_color
}

// Compute the reflected color
R = ReflectRay(-D, N)
❷ reflected_color = TraceRay(P, R, 0.001, inf, recursion_depth - 1)

❸ return local_color * (1 - r) + reflected_color * r
}

```

示例 4-4：光线追踪器伪代码，现在带有反射

对代码的更改非常简单。首先，我们检查是否需要计算反射❶。如果球体没有反射，或者我们达到了递归限制，我们就完成了，我们可以返回球体自己的颜色。

最有趣的变化是递归调用❷；调用自身，具有适当的反射参数，重要的是，递归深度计数器递减；这与检查❶相结合，可以防止无限循环。TraceRay

最后，一旦我们有了球体的局部颜色和反射的颜色，我们就将它们混合在一起❸，使用“这个球体的反射率”作为混合权重。

我会让结果自己说话。查看图 4-8。

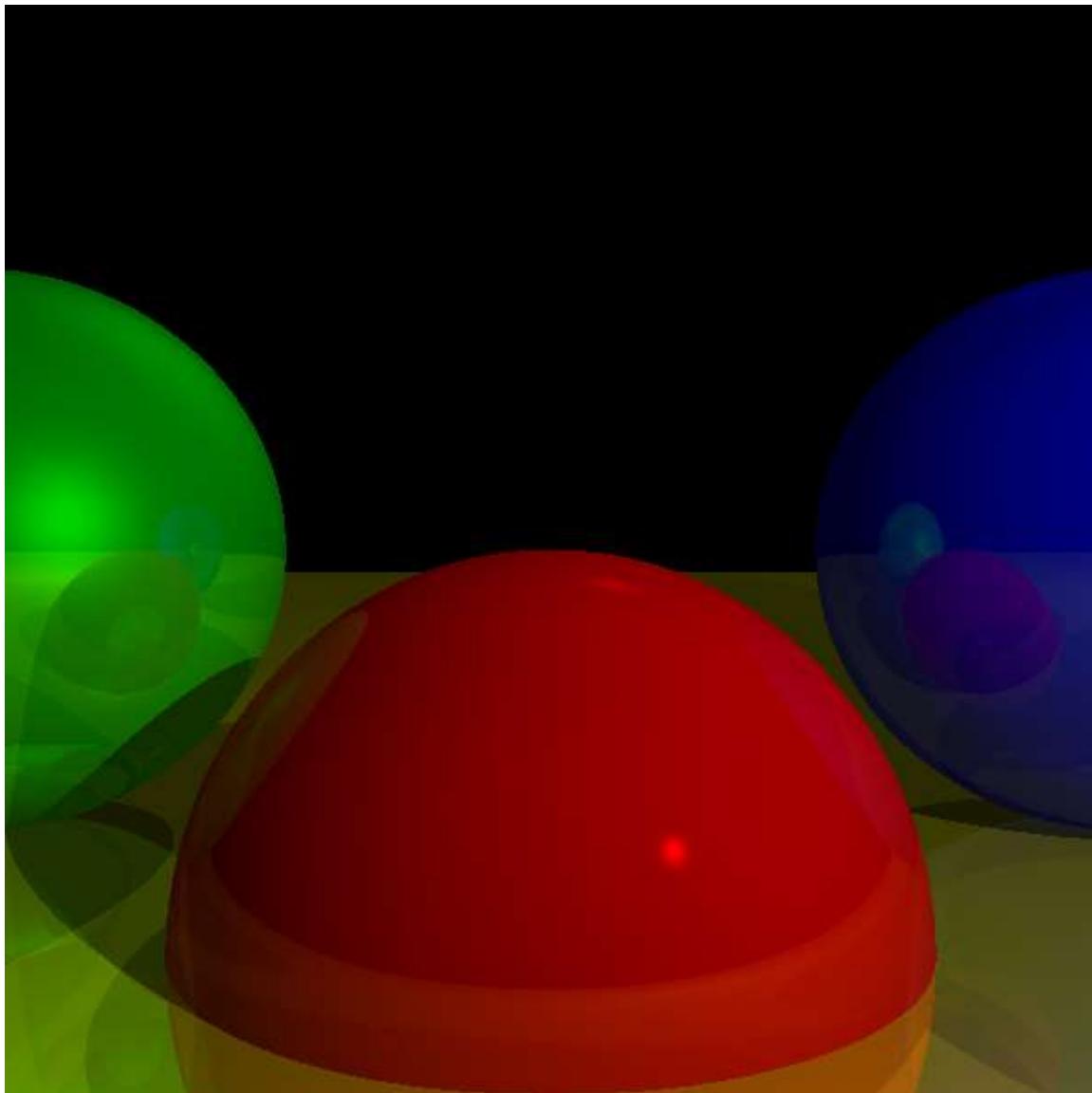


图 4-8：光线追踪场景，现在带有反射

## 总结

---

在前面的章节中，我们开发了一个基本框架来在3D画布上渲染2D场景，对光线与物体表面交互的方式进行建模。这给了我们一个简单的场景初始表示。

在本章中，我们扩展了这个框架，以模拟场景中的不同对象如何不仅与光线交互，而且通过相互投射阴影和相互反射来相互作用。因此，渲染的场景看起来更加逼真。

在下一章中，我们将简要讨论扩展这项工作的不同方法，从表示球体以外的对象到渲染性能等实际考虑因素。