

# 1 全局光照

## 1.1 光线投射与光线追踪

### 1.1.1 光线投射

大部分环境中的光都不能被镜头所捕捉, 因此相比于考虑每个光源发出的每条光线, 我们不妨采取逆向思维, 考虑那些最终到达镜头的光线. 由于光路是可逆的, 因此我们从镜头向屏幕上的点连一条线, 该射线在场景中击中的第一个物体将决定该点的颜色.

**定义 1.1 光线投射** 光线投射 (Ray Casting) 是一种通过从观察点向场景中投射射线来确定可见表面和颜色的技术.

因此, 光线投射算法的基本流程如下:

1. 对于屏幕的每个像素点  $(x, y)$ , 从相机位置向场景中投射一条穿过  $(x, y)$  的射线  $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ .
2. 计算射线  $\mathbf{r}(t)$  与场景中的物体第一次发生相交的位置  $\mathbf{p}$ .
3. 将交点  $\mathbf{p}$  与所有光源相连, 分别得到一根 shadow ray. 判断 shadow ray 在到达光源之前是否与其他物体相交, 如果相交则该光源对点  $\mathbf{p}$  不可见, 否则可见.
4. 在可见的情况下, 我们已经知道了指向光源的方向  $\mathbf{l}$ , 指向观察者的方向  $-\mathbf{d}$ , 以及表面法线  $\mathbf{n}$ . 使用在光照与着色中介绍的各种光照模型即可计算光源对点  $\mathbf{p}$  贡献的颜色.
5. 将所有光源的贡献累加, 得到最终的像素颜色.

上述过程的伪代码可以表示如下:

```
1 for each pixel (x, y) do
2     # Generate primary ray from camera through pixel
3     ray = generateRay(camera, x, y)
4     # Find intersection with scene.
5     hitInfo = intersectScene(ray)
6     if hitInfo.hit then
7         color = vec3(0, 0, 0)
8         # For each light source, check visibility and compute lighting.
9         for each light in scene.lights do
10             shadowRay = generateShadowRay(hitInfo.position, light.position)
11             if not intersectScene(shadowRay).hit then
12                 color += computeLighting(hitInfo, light)
13             setPixelColor(x, y, color)
14     else
15         setPixelColor(x, y, backgroundColor)
```

### 1.1.2 光线追踪

在实际情况下, 物体接受的光照可能并不直接来自于光源, 而可能经过其它物体的反射/折射再到达该物体上. 为了考虑这些间接光照, 我们需要引入光线追踪技术.

**定义 1.2 光线追踪** 光线追踪 (Ray Tracing) 通过模拟光线在场景中的传播路径, 包括反射和折射, 来生成更加逼真的图像.

光线追踪算法的基本流程如下:

1. 对于屏幕的每个像素点  $(x, y)$ , 从相机位置向场景中投射一条穿过  $(x, y)$  的射线  $r(t) = o + td$ .
2. 计算射线  $r(t)$  与场景中的物体第一次发生相交的位置  $p$ .
3. 按照光线投射中介绍的办法计算交点  $p$  处直接来自于光源形成的颜色.
4. 根据材质属性, 生成反射射线和折射射线, 递归地追踪这些射线以计算间接光照形成的颜色.
5. 将局部光照和间接光照形成的颜色累加, 得到最终的像素颜色.

上述过程的递归形式的伪代码可以表示如下:

```

1  # Define recursively ray tracing function.
2  function traceRay(ray, depth):
3      if depth > maxDepth then
4          return backgroundColor
5      hitInfo = intersectScene(ray)
6      if hitInfo.hit then
7          color = computeLocalLighting(hitInfo)
8          # Compute reflection
9          if hitInfo.material.reflective then
10             reflectRay = generateReflectRay(hitInfo)
11             color += hitInfo.material.reflectivity * traceRay(reflectRay, depth + 1)
12          # Compute refraction
13          if hitInfo.material.refractive then
14             refractRay = generateRefractRay(hitInfo)
15             color += hitInfo.material.transparency * traceRay(refractRay, depth + 1)
16          return color
17      else
18          return backgroundColor
19  for each pixel (x, y) do
20      ray = generateRay(camera, x, y)
21      color = traceRay(ray, 0)
22      setPixelColor(x, y, color)

```

光线追踪算法能够生成高度逼真的图像, 但计算开销较大, 因此在实际应用中通常会结合其他算法进行优化. 我们将在之后讲到优化的方式. 在此之前, 我们先来了解光线投射/追踪的基本步骤, 即光线求交.

## 1.2 光线求交

可以看出, 在光线投射与光线追踪算法中, 计算射线与场景中物体的交点是一个核心步骤. 高效地实现光线求交对于提升渲染性能至关重要. 下面我们将介绍几种常见的几何体与光线求交点的办法. 在本节的推导中, 我们假定光线的方程为

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}, \quad 0 \leq t < +\infty$$

其中  $\mathbf{o}$  意为 Origin, 即光线的起点, 而  $\mathbf{d}$  意为 Direction, 即光线的方向向量. 求得的  $t$  即与物体的交点到光线起点的距离.

### 1.2.1 光线与球面求交

考虑球心为  $\mathbf{c}$ , 半径为  $r$  的球面, 其隐式方程为

$$\|\mathbf{p} - \mathbf{c}\|^2 = r^2$$

代入光线方程, 我们有

$$\|\mathbf{o} + t\mathbf{d} - \mathbf{c}\|^2 = r^2$$

展开后得到关于  $t$  的二次方程

$$t^2 \mathbf{d} \cdot \mathbf{d} + 2t \mathbf{d} \cdot (\mathbf{o} - \mathbf{c}) + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 = 0$$

设  $a = \mathbf{d} \cdot \mathbf{d}$ ,  $b = 2\mathbf{d} \cdot (\mathbf{o} - \mathbf{c})$ ,  $c = (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2$ , 则方程的解为

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

根据判别式  $b^2 - 4ac$  的值, 我们可以判断光线与球面的交点情况是相离, 相切还是相交, 然后根据求根公式即可求出交点.

### 1.2.2 光线与长方体求交

为了简化光线与物体的求交计算, 我们通常对物体创建一个规则的几何外形将其包围. 最常见的包围体是轴对齐包围盒 (Axis-Aligned Bounding Box, AABB).

**定义 1.3 轴对齐包围盒** 轴对齐包围盒是指其边界与坐标轴平行的长方体, 通常由最小点  $\mathbf{p}_{\min} = (x_{\min}, y_{\min}, z_{\min})$  和最大点  $\mathbf{p}_{\max} = (x_{\max}, y_{\max}, z_{\max})$  定义.

这里介绍一种高效的光线与 AABB 求交的方法, 即 **Slabs Method**. 不失一般性地, 我们考虑长方体中平行于  $xy$  平面的表面. 设表面所在的方程分别为  $z = z_{\min}$  和  $z = z_{\max}$ , 则光线与这两个平面的交点  $t$  值分别为

$$t_{z_{\min}} = \frac{z_{\min} - \mathbf{o}_z}{\mathbf{d}_z}, \quad t_{z_{\max}} = \frac{z_{\max} - \mathbf{o}_z}{\mathbf{d}_z}$$

于是光线处于这两个平面之间时总有

$$t_{z_{\min}} \leq t \leq t_{z_{\max}}$$

类似地, 我们可以计算出光线与平行于  $xy$  平面和  $yz$  平面的交点  $t$  值, 分别记为  $t_{x_{\min}}, t_{x_{\max}}$  和  $t_{y_{\min}}, t_{y_{\max}}$ . 如果光线与长方体相交, 那么它同时处于这三组平面之间, 于是

$$[t_{x_{\min}}, t_{x_{\max}}] \cap [t_{y_{\min}}, t_{y_{\max}}] \cap [t_{z_{\min}}, t_{z_{\max}}] \neq \emptyset$$

于是令

$$t_{\min} = \max\{t_{x_{\min}}, t_{y_{\min}}, t_{z_{\min}}\}, \quad t_{\max} = \min\{t_{x_{\max}}, t_{y_{\max}}, t_{z_{\max}}\}$$

只需要  $t_{\min} \leq t_{\max}$ , 则光线与长方体相交, 交出的线段即为

$$\mathbf{r}'(t) = \mathbf{o} + t\mathbf{d}, \quad t_{\min} \leq t \leq t_{\max}$$

### 1.2.3 光线与三角面片求交

三角面片是计算机图形学中最常用的基本几何体之一, 因此高效地实现光线与三角面片的求交对于光线追踪算法至关重要. 这里介绍一种常用的光线与三角面片求交算法, 即 **Möller-Trumbore 算法**. 对于三角形上任意一点  $\mathbf{t}(u, v)$ , 其满足

$$\mathbf{t}(u, v) = (1 - u - v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2$$

这里的  $u, v$  即为点在三角形中的重心坐标, 满足  $u \geq 0, v \geq 0$  且  $u + v \leq 1$ . 将上述方程与光线方程联立, 可得

$$\mathbf{o} + t\mathbf{d} = (1 - u - v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2$$

整理后得到

$$-t\mathbf{d} + u(\mathbf{v}_1 - \mathbf{v}_0) + v(\mathbf{v}_2 - \mathbf{v}_0) = \mathbf{o} - \mathbf{v}_0$$

这一线性方程组的矩阵形式为

$$\begin{bmatrix} -\mathbf{d} & \mathbf{v}_1 - \mathbf{v}_0 & \mathbf{v}_2 - \mathbf{v}_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \mathbf{o} - \mathbf{v}_0$$

令

$$\mathbf{D} = \mathbf{d}, \quad \mathbf{E}_1 = \mathbf{v}_1 - \mathbf{v}_0, \quad \mathbf{E}_2 = \mathbf{v}_2 - \mathbf{v}_0, \quad \mathbf{T} = \mathbf{o} - \mathbf{v}_0$$

根据 Cramer 法则可知线性方程组的解为

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\det \begin{bmatrix} -\mathbf{D} & \mathbf{E}_1 & \mathbf{E}_2 \end{bmatrix}} \begin{bmatrix} \det \begin{bmatrix} \mathbf{T} & \mathbf{E}_1 & \mathbf{E}_2 \end{bmatrix} \\ \det \begin{bmatrix} -\mathbf{D} & \mathbf{T} & \mathbf{E}_2 \end{bmatrix} \\ \det \begin{bmatrix} -\mathbf{D} & \mathbf{E}_1 & \mathbf{T} \end{bmatrix} \end{bmatrix}$$

根据我们在线性代数中所学的知识有

$$\det \begin{bmatrix} \mathbf{A} & \mathbf{B} & \mathbf{C} \end{bmatrix} = -(\mathbf{A} \times \mathbf{C}) \cdot \mathbf{B} = -(\mathbf{C} \times \mathbf{B}) \cdot \mathbf{A}$$

于是上式可以改写为

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(\mathbf{D} \times \mathbf{E}_2) \cdot \mathbf{E}_1} \begin{bmatrix} (\mathbf{T} \times \mathbf{E}_1) \cdot \mathbf{E}_2 \\ (\mathbf{D} \times \mathbf{E}_2) \cdot \mathbf{T} \\ (\mathbf{T} \times \mathbf{E}_1) \cdot \mathbf{D} \end{bmatrix}$$

令  $\mathbf{P} = \mathbf{D} \times \mathbf{E}_2$ ,  $\mathbf{Q} = \mathbf{T} \times \mathbf{E}_1$ , 则有

$$t = \frac{\mathbf{Q} \cdot \mathbf{E}_2}{\mathbf{P} \cdot \mathbf{E}_1}, \quad u = \frac{\mathbf{P} \cdot \mathbf{T}}{\mathbf{P} \cdot \mathbf{E}_1}, \quad v = \frac{\mathbf{Q} \cdot \mathbf{D}}{\mathbf{P} \cdot \mathbf{E}_1}$$

最后判断  $u \geq 0, v \geq 0, u + v \leq 1$  且  $t \geq 0$  即可确定光线与三角面片是否相交, 如果相交则交点为  $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ , 在三角面片上的重心坐标为  $(u, v)$ .

上述过程的算法流程如下:

1. 按照定义计算辅助向量  $\mathbf{D}, \mathbf{E}_1, \mathbf{E}_2, \mathbf{T}, \mathbf{P}, \mathbf{Q}$ .
2. 判断行列式  $\mathbf{P} \cdot \mathbf{E}_1$  是否接近于 0, 如果是则光线与三角面片平行, 不相交.
3. 计算参数  $t, u, v$  的值.
4. 判断  $u \geq 0, v \geq 0, u + v \leq 1$  且  $t \geq 0$  是否成立, 如果成立则光线与三角面片相交, 否则不相交.

### 1.3 空间加速结构

如果每一次光线求交都要遍历场景中的所有三角面片, 计算效率将非常低下. 为了提升光线求交的效率, 我们通常会使用空间加速结构对场景进行划分, 从而减少每次求交时需要检查的三角面片数量. 下面介绍几种常见的空间加速结构. 从原理上而言, 它们都是将空间划分为若干个区域, 先判断光线是否与区域相交, 如果相交才会进一步检查该区域内的物体, 从而避免了很多不必要的求交计算. 它们之间的区别主要在于区域划分的方式.

#### 1.3.1 层次包围盒

**层次包围盒 (Hierarchical Bounding Boxes)** 是一种通过构建包围盒<sup>1</sup>树来组织场景中物体的空间加速结构. 包围盒树的每个节点表示一个包围盒, 其子节点表示该包围盒内包含的更小的包围盒或物体. 在光线求交时, 首先检查光线与根节点的包围盒是否相交, 如果相交则递归地检查子节点, 否则跳过该子树.

容易看出建立查找树可以使得求交的时间复杂度从  $O(n)$  降低到  $O(\log n)$ , 因此对于渲染效率有显著提升. 我们现在来介绍层次包围盒构建的具体办法:

1. 计算当前节点包含的所有物体的包围盒并存储.
2. 按照一定的划分策略将物体划分为两个子集, 作为该节点的子节点.

<sup>1</sup>一般而言, 包围盒是一个包含其中所有物体的最小的长方体. 光线与包围盒的求交可以用前面介绍的 **Slabs** 算法实现.

3. 递归地对每个子节点重复上述过程, 直到满足终止条件 (如节点包含的物体数量小于某个阈值).

在光线求交时, 我们可以按照如下流程进行:

1. 从根节点开始, 检查光线与当前节点的包围盒是否相交.
2. 如果相交, 则递归地检查子节点; 如果不相交, 则跳过该子树.
3. 当到达叶子节点时, 对该节点包含的所有物体进行求交测试, 记录光源到交点的距离.
4. 递归结束后对最近的交点进行处理, 计算颜色等信息, 然后返回.

需要注意的是, 在递归的过程中并不需要对每个交点进行精确的计算 (例如插值计算颜色等信息), 这会严重影响效率, 只需要更新最近的交点即可.

### 1.3.2 八叉树

我们在前面已经介绍过八叉树这一数据结构, 它也可以用作空间加速. 八叉树的每个节点表示一个立方体空间, 其子节点表示该立方体被划分后的八个子立方体. 在光线求交时, 首先检查光线与根节点的立方体是否相交, 如果相交则递归地检查子节点, 否则跳过该子树.

### 1.3.3 BSP 树和 KD 树

BSP 树 (Binary Space Partitioning Tree) 和 KD 树 (K-Dimensional Tree) 是两种基于空间划分的加速结构. 它们通过选择一个平面<sup>2</sup>将空间划分为两个子空间, 并递归地对每个子空间进行划分, 直到满足终止条件. 在光线求交时, 首先计算光线与当前节点的划分平面的交点, 据此可以判断它与该平面划分出的两个子节点的相交关系, 然后递归地检查相交的子节点.

两者的区别主要在于 KD 树的划分平面要求平行于  $xy$ ,  $yz$  或  $xz$  平面 (也即只能是  $x = c$ ,  $y = c$  或  $z = c$  中的一种), 而 BSP 树的划分平面可以是任意方向的平面. 一般而言, KD 树的构建和求交效率更高, 而 BSP 树仅在一些特殊的场景下使用 (例如场景中有很多倾斜的物体).

---

<sup>2</sup>与 BVH 树不同的是 BSP 树和 KD 树并不需要重新计算每个子节点的包围盒, 子节点的几何形状已经由划分平面天然地确定了.