第十五章　全局光照

在之前的章节中我们已经了解到，当前计算机图形学中的渲染管线主要有光栅化和光线追踪两种形式．其中，光栅化方法由于其高效、易行的特点，广泛用于实时交互、电子游戏等领域，但却牺牲了渲染质量．特别地，光栅化方法难以处理软阴影、镜面反射、间接光照等全局现象．与之相反，光线追踪方法善于解决上述困难，而其代价则是渲染时间的成倍提升．

在本章中，我们讨论所谓的 Whitted 风格的光线追踪．它诞生于 20 世纪 80 年代伊始，是最简单的光线追踪算法形式，但却有至关重要的奠基作用．

15.1　光线投射

首先，我们介绍光线投射 *(ray casting)*．设想这样一个问题: 在渲染一幅画面的过程中，屏幕上任意一个点所对应的颜色应该来自于场景中的哪一个物体? 解决该问题的方法非常直接，即，我们从人眼 (摄像机) 向屏幕上的点连一条射线，该射线在场景中击中的第一个物体将决定该点的颜色．如图15.1所示．

上述这一发出射线、击中物体的过程，也就是所谓的光线投射．考虑到离散屏幕空间是由像素组成的，我们可以很容易地写出下面的算法流程:

1. 对于每一个像素点 (x, y)，从观察者的眼睛发出一条穿过 (x, y) 的光线．
2. 找出这条光线与场景中的物体第一次发生相交的位置．
3. 将这个交点与每一个光源相连，分别形成一根 shadow ray（这根光线也可能被障碍物遮挡）
4. 通过 shadow ray 和我们一开始发出的光线，我们得到了入射方向和出射方向，结合交点

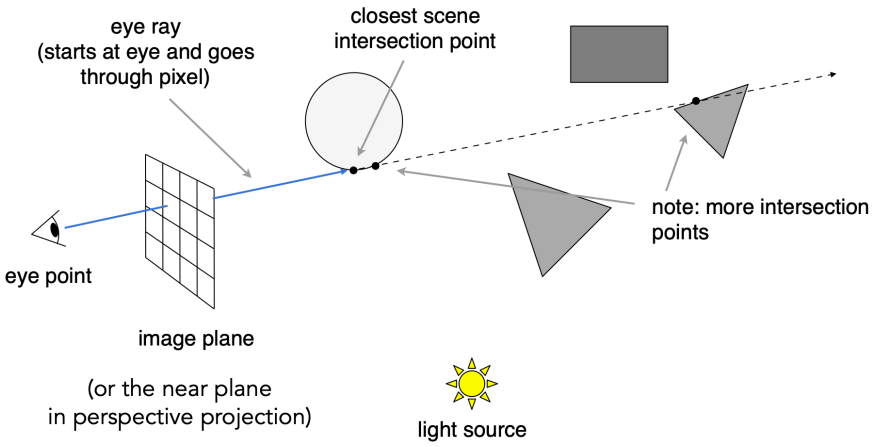

图 15.1: 光线投射的基本原理

的几何信息，我们也已知了法线方向，从而可以运用之前介绍过的着色模型计算出交点的颜色.

5. 将算出的颜色作为像素点 (x, y) 的颜色写入.

考虑到之前我们已介绍过的着色模型，即可得到利用光线投射法来渲染场景的伪代码：

```
Raycast()
    /* generate a picture */
    for each pixel x,y
        color(pixel) = Trace(ray_through_pixel(x,y))

Trace(ray)
    /* fire a ray, return RGB radiance of light traveling backward along it */
    object_point = Closest_intersection(ray)
    if object_point
        return Shade(object_point, ray)
    else
        return Background_Color

Closest_intersection(ray)
    for each surface in scene
        calc_intersection(ray, surface)
    return the closest point of intersection to viewer
    (also return other info about that point, e.g., surface normal, material properties, etc.)

Shade(point, ray)
    /* return radiance of light leaving point in opposite of ray direction */
    calculate surface normal vector
    use Phong illumination formula (or something similar)
    to calculate contributions of each light source
```

15.2　光线追踪

如图15.2所示，光线追踪就是在光线投射的基础上，进一步考虑光线的折射和反射，找出这个过程中与物体的每一个交点. 然后分别计算颜色，加权相加写入到对应的像素当中. 值得注意的是，在光线追踪算法中，我们只对可以反射和折射的材质进行递归处理，这些材质的颜色的权重将由其反射率和折射率决定；而当遇到其他材质时递归就会停止，此时它们的颜色将获得全部的权重.

我们可以写出光线追踪的伪代码如下：

```
Trace(ray)
    /* fire a ray, return RGB radiance of light traveling backward along it */
    object_point = Closest_intersection(ray)
    if object_point
        return Shade(object_point, ray)
    else
        return Background_Color

Shade(point, ray)
    /* return radiance along ray */
    radiance = black; /* initialize color vector */
    for each light source
        shadow_ray = calc_shadow_ray(point,light)
        if !in_shadow(shadow_ray,light)
            radiance += phong_illumination(point,ray,light)
        if material is specularly reflective
```

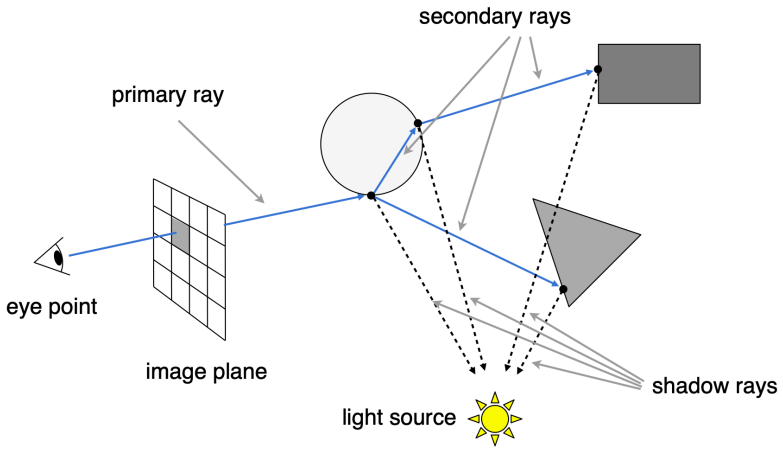

图 15.2: 光线追踪的基本原理

```
        radiance += spec_reflectance * Trace(reflected_ray(point,ray))
    if material is specularly transmissive
        radiance += spec_transmittance * Trace(refracted_ray(point,ray))
return radiance
```

15.3　数学: 射线与几何体相交

在光线追踪算法中，我们需要反复求解光线与物体的交点，所以我们有必要研究一番这里的数学. 首先，我们将光线定义为一条从原点发出，沿着某个方向延展的射线，可以用如下方程表示：

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \qquad 0 \leq t < \infty \tag{15.1}$$

这里 $\mathbf{o}$ 表示光线的原点，$\mathbf{d}$ 表示光线射出方向的单位向量，这样我们只要将这个方程与表示物体表面的方程联立，就可以解出交点.

15.3.1　球面

我们设球心的位置为 $\mathbf{c}$，球面的半径为 R，那么球面上任意一点 $\mathbf{p}$ 的位置满足方程：

$$\|\mathbf{p} - \mathbf{c}\|^2 - R^2 = 0 \tag{15.2}$$

与光线方程15.1联立，我们就可以得到交点满足：

$$\|\mathbf{o} + t\mathbf{d} - \mathbf{c}\|^2 - R^2 = 0 \tag{15.3}$$

我们可以将这个式子展开得到一个一元二次方程组：$at^2 + bt + c = 0$，其中 $a = \|\mathbf{d}\|^2$，$b = 2(\mathbf{o} - \mathbf{c}) \cdot \mathbf{d}$，$c = \|\mathbf{o} - \mathbf{c}\|^2 - R^2$，而最后解出交点：

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{15.4}$$

这里 $b^2 - 4ac < 0$、$b^2 - 4ac = 0$ 以及 $b^2 - 4ac > 0$ 分别对应光线与球面相离、相切和相割的三种情况.

15.3.2　长方体

为了简化光线与物体的求交计算，通常会对物体创建一个规则的几何外形将其包围. 其中，最常见的创建包围盒的方法就是 AABB（axis-aligned bounding box），即轴对称包围盒，它要求我们构建的长方体每条边都平行于一个坐标平面，且恰好能将物体包围.

在这里我们介绍一种称之为 Slabs method 的方法来高效地解决光线与长方体求交的问题. 我们首先不失一般性地考虑长方体中平行于 xy 平面的一对表面，设表面所在平面的方程分别为 $z = z_1$ 和 $z = z_2$，接着结合我们的光线方程15.1我们可以得到光线与这两个平面相交时满足：

$$z_i = \hat{\mathbf{z}} \cdot \mathbf{r}(t) = \hat{\mathbf{z}} \cdot \mathbf{o} + t_i \hat{\mathbf{z}} \cdot \mathbf{d} \qquad i \in \{1, 2\} \tag{15.5}$$

我们解出 t_1 和 t_2 后，取其中的较小值记为 t_{near}，较大值记为 t_{far}，那么我们就可以得到光线在这两个平面内时 t 的取值为 $[t_{near}, t_{far}]$. 同样的，我们可以求出光线在另外两对平面内时 t 的取值范围，如果这三个取值范围有交集的话，那就说明光线在 t 取这个交集范围之内时同时处在这三对平面之间，也就是出现在了长方体内，此时就可以判定光线与长方体相交. 而判断三个取值范围是否有交集是不难做到的，我们只要取三组 t_{near} 的最大值和三组 t_{far} 的最小值进行比较，如果前者小于后者就说明存在交集，从而说明了光线会与长方体相交.

15.3.3　三角面片

我们通常会用三角面片来离散化地表示物体，所以光线与三角形求交是一个值得研究的问题. 历史上人们针对这个问题提出过许多算法，在这里我们介绍 Möller Trumbore Algorithm[1]. 首先，我们设三角形三个顶点的坐标分别为 $\mathbf{V}_0$、$\mathbf{V}_1$ 和 $\mathbf{V}_2$，对于三角形上的任意一点 $\mathbf{T}(u, v)$，它满足：

$$\mathbf{T}(u, v) = (1 - u - v)\mathbf{V}_0 + u\mathbf{V}_1 + v\mathbf{V}_2 \tag{15.6}$$

这里的 (u, v) 为三角形的重心坐标，满足 $u \geq 0$，$v \geq 0$ 和 $u + v \leq 1$，与光线方程15.1联立，我们可以得到交点满足：

$$\mathbf{o} + t\mathbf{d} = (1 - u - v)\mathbf{V}_0 + u\mathbf{V}_1 + v\mathbf{V}_2 \tag{15.7}$$

用矩阵的形式表示上式我们可以得到：

$$\begin{bmatrix} -\mathbf{D}, \mathbf{V}_1 - \mathbf{V}_0, \mathbf{V}_2 - \mathbf{V}_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \mathbf{O} - \mathbf{V}_0 \tag{15.8}$$

我们设 $\mathbf{E}_1 = \mathbf{V}_1 - \mathbf{V}_0$，$\mathbf{E}_2 = \mathbf{V}_2 - \mathbf{V}_0$ 和 $\mathbf{T} = \mathbf{O} - \mathbf{V}_0$，运用克拉默法则可以解得：

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-\mathbf{D}, \quad \mathbf{E}_1, \quad \mathbf{E}_2|} \begin{bmatrix} |\mathbf{T}, \quad \mathbf{E}_1, \quad \mathbf{E}_2| \\ |-\mathbf{D}, \quad \mathbf{T}, \quad \mathbf{E}_2| \\ |-\mathbf{D}, \quad \mathbf{E}_1, \quad \mathbf{T}| \end{bmatrix} \tag{15.9}$$

运用我们在线性代数中学到的知识，我们知道 $|\mathbf{A}, \quad \mathbf{B}, \quad \mathbf{C}| = -(\mathbf{A} \times \mathbf{C}) \cdot \mathbf{B} = -(\mathbf{C} \times \mathbf{B}) \cdot \mathbf{A}$，所以我们可以进一步将上式化简为：

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(\mathbf{D} \times \mathbf{E}_2) \cdot \mathbf{E}_1} \begin{bmatrix} (\mathbf{T} \times \mathbf{E}_1) \cdot \mathbf{E}_2 \\ (\mathbf{D} \times \mathbf{E}_2) \cdot \mathbf{T} \\ (\mathbf{T} \times \mathbf{E}_1) \cdot \mathbf{D} \end{bmatrix} = \frac{1}{\mathbf{P} \cdot \mathbf{E}_1} \begin{bmatrix} \mathbf{Q} \cdot \mathbf{E}_2 \\ \mathbf{P} \cdot \mathbf{T} \\ \mathbf{Q} \cdot \mathbf{D} \end{bmatrix} \tag{15.10}$$

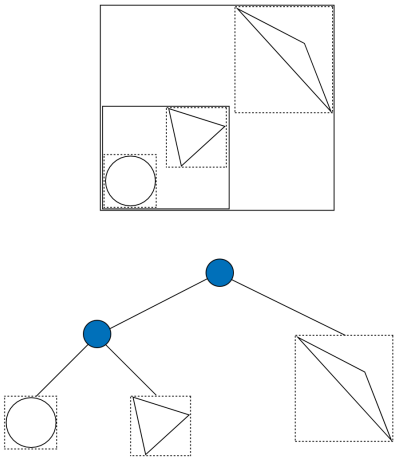

图 15.3: 层次包围盒的基本结构．图源：PBRT 第三版

这里我们令 $\mathbf{P} = \mathbf{D} \times \mathbf{E}_2$ 和 $\mathbf{Q} = \mathbf{T} \times \mathbf{E}_1$，通过这些变量代换，我们的计算效率将得到提高．当我们根据方程15.10算出了 t, u, v 后，就可以通过判断是否满足 $0 \leq t < \infty$ 来得知交点是否在光线上，以及是否满足 $u \geq 0$，$v \geq 0$ 和 $u + v \leq 1$ 来得知交点是否在三角形内．

15.4　加速结构

虽然光线追踪算法可以为我们提供高质量的渲染结果，但是渲染时间会成倍地增加．其中最耗费时间的是需要不断地去求光线与物体的交点．最简单的算法就是对每一条光线都遍历所有三角形，但是这显然开销太大．人们为此提出了各种各样的空间加速结构来优化这一过程．希望借助它们预测出一条光线可能相交的三角形集合，从而减少对大量不可能相交的三角形的求交计算．

15.4.1　网格化

网格化是指将空间切分为等大的体素．对于每个体素，我们会记录在体素内存在顶点的所有图元．这样在光线追踪时，我们可以从光线原点开始，由近及远地依次查询光线是否与穿过的体素内的图元相交．如果在某个体素内发生相交，则只需找出其中最近的交点返回即可．

但总的来说，网格化在作为加速结构时对于网格的大小往往会有比较高的要求，同时当场景中的物体分布比较不均匀时表现也会比较差．人们一般会用其他的加速结构来提升渲染的速度．

15.4.2　层次包围盒

层次包围盒（Bounding Volume Hierarchies，BVH）是一种基于图元（即构成场景的基本元素，如三角形、球面等）划分的空间索引结构．如图15.3所示，实现层次包围盒时我们会构建一棵层次包围盒树，图元被存储在叶节点上，每一个节点都存储了一个包围了其所有子节点内图元的包围盒．需要注意的是，每一个图元虽然在整个结构中只会出现一次．但是同一块空间区域可能会被多个节点所包含．由于我们构建的层次包围盒树是一棵二叉树，

在这棵二叉树中，所有的内部节点都有两个子节点，所以当有 n 个叶节点时，我们构建的层次包围盒树会有 $n-1$ 个内部节点，其消耗的内存是确定的.

层次包围盒相比于 KD 树构建效率更高，但是遍历效率略低. 另外，层次包围盒相较于 KD 树拥有更好的数值鲁棒性，更不容易因为浮点数舍入误差而导致弃真（实际相交，但是计算结果为不相交）情况的出现.

层次包围盒的构建

层次包围盒的构建大致可以分为两步：

1. 计算每个图元的包围盒（如 AABB 包围盒）和质心位置并存储.
2. 按照划分策略构建树索引结构

整个构建过程是递归的. 每次递归都会处理一组图元，当图元数量小于某一阈值时（叶节点允许的最大图元数），递归就会终止，返回一个叶节点. 不然，我们就需要考虑将图元按照一定的策略划分为两部分，形成两个子节点，并递归地对这两部分进行处理.

所以在构建过程中最重要的问题就是如何对图元进行划分. 这个策略应该尽可能减少划分后两部分包围盒重叠的体积. 因为重叠的体积越大，光线穿过重叠区域的可能性也越大，遍历两棵子树的可能性就越高，计算消耗也越多. 为此，我们一般会在图元跨度最大的坐标轴上进行划分. 这里，图元的跨度可以用图元的包围盒或图元的质心位置来进行衡量.

于是很自然的，我们可以首先提出两种简单的划分策略. 一种是根据坐标轴跨度的中点进行划分，将位于中点左侧的图元划分到左节点，而将位于中点右侧的图元划分到右节点. 另一种是进行等量划分，将最左边的一半图元划分到左节点，剩下的一半则划分到右节点. 但是这两种划分策略在图元分布不均匀的时候会得到很差的结果.

一种更为常用且效果更好的方法是基于表面积的启发式评估划分方法（Surface Area Heuristic，SAH），这种方法通过对求交代价和遍历代价进行评估，给出了每一种划分的代价，而我们的目的便是去寻找代价最小的划分.

假设当前节点中存在 n 个物体，设对每个物体求交的代价为 $t(i)$，如果不做划分依次求交的总代价为：

$$\sum t(i) = t(1) + t(2) + \cdots + t(n) \tag{15.11}$$

如果我们将他们分为 2 组，这两组物体分别有包围盒 A 和包围盒 B. 设光线击中它们的概率分别为 $p(A)$ 和 $p(B)$，设 t_{trav} 为遍历这层树结构（即判断是否会与包围盒 A 和包围盒 B 相交）的时间开销，那么划分后对每个物体求交的代价就变为：

$$c(A,B) = p(A)\sum_{i\in A} t(i) + p(B)\sum_{i\in B} t(i) + t_{trav} \tag{15.12}$$

假设对于每个物体求交的代价都近乎相同，所以设对于任意 i 有 $t(i)=1$，又考虑到遍历树结构的开销会低于与物体求交的代价，所以设 $t_{trav}=0.125$（关于这个数字的估计可以具体根据代码实现进行调整）. 再设包围盒 A 中的图元数量为 a，包围盒 B 中的图元数量为 b，我们就可以得到：

$$c(A,B) = p(A)a + p(B)b + 0.125 \tag{15.13}$$

最后我们需要对 $p(A)$ 和 $p(B)$ 进行估计，值得注意的是，由于包围盒 A 和包围盒 B 可能有重叠，同时它们也可能并不会占据当前节点包围盒的全部空间，所以 $p(A)+p(B)$ 并不一定严格等于 1. 但是，当包围盒的表面积越大时，它也就越有可能被光线击中，所以我们从

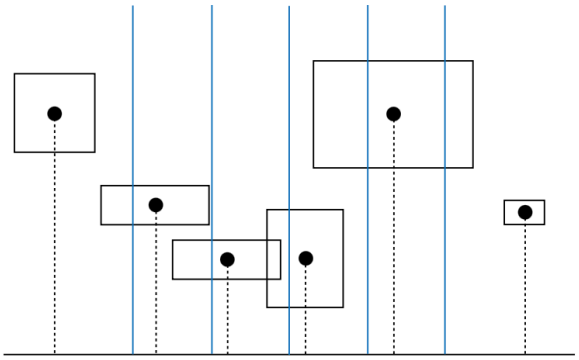

图 15.4: 在实际应用中，我们会将当前节点的包围盒空间沿着跨度最长的坐标轴的方向均匀地划分为若干个桶．图源：PBRT 第三版

这一点出发，取子包围盒表面积与父包围盒表面积的比值来估计 $p(A)$ 和 $p(B)$，从而有：

$$c(A,B)=\frac{S(A)}{S(C)}a+\frac{S(B)}{S(C)}b+0.125 \tag{15.14}$$

在这里，我们设包围盒 A 和包围盒 B 以及当前节点的包围盒的表面积分别为 $S(A)$，$S(B)$ 和 $S(C)$．而 SAH 就是选择这其中代价最小的划分作为最终划分．

当然，如果要比较所有可行的划分找出其中的最小值未免开销太大．如图15.4，在实际应用中会将当前节点的包围盒空间沿着跨度最长的坐标轴的方向均匀地划分为若干个桶，考虑的划分只可能出现在桶与桶之间．这样，如果桶的数量为 n，那么可能的划分就只有 $n-1$ 种．虽然这么做，丢失了一定的精度，但是事实表明，这样得到的划分也足够好．按照这个划分，我们可以按照图元质心的位置将它们分别放入到对应的桶中，接着就可以算出每个桶中所有图元构成的包围盒．最后，对于每一种 $mid\in[1,n]$ 的划分的代价就可以表示为：

$$c(A,B)=\left(\sum_{i<mid}n_i\right)\frac{\bigcup_{i<mid}S(i)}{S(C)}+\left(\sum_{j\geq mid}n_j\right)\frac{\bigcup_{j\geq mid}S(j)}{S(C)}+t_{trav} \tag{15.15}$$

这里，$\bigcup S(i)$ 代表了各个桶 i 的包围盒一起形成的大包围盒的表面积．我们从中选出代价最小的一种划分方案进行划分即可．

另外，如果当前节点中所有图元质心的位置都相同的话，就需要采取特殊处理，我们可以直接建立一个叶节点，让这个叶节点包含所有图元．因为如果他们的质心位置已经相同，再进行分割将没有意义，最终都需要对它们逐个求交．

层次包围盒的遍历

对于层次包围盒的遍历是容易的．对于每一个节点，首先检查光线和该节点的包围盒是否相交．如果不相交则不再需要对该节点的子节点进行遍历．若相交且该节点为内部节点则对其子节点进行遍历．若相交且该节点为叶节点，则对该叶节点下的每一个图元进行相交测试．

一般情况下需要遍历完整个层次包围盒才能得到最近的交点，因为各节点之间是可能存在重叠区域的．但是如果光线是在对光源进行遮挡测试时发射的 Shadow Ray，则不需要遍历整个层次包围盒，因为此时只需要确认是否存在交点即可，一旦发现有一个图元和 Shadow Ray 相交即可返回．

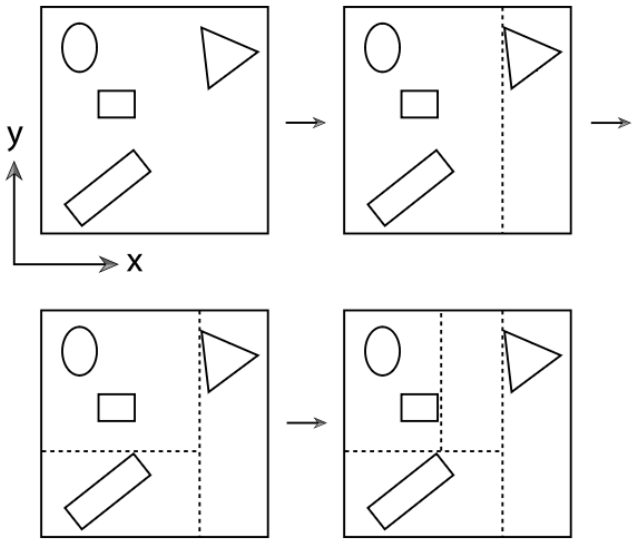

图 15.5: KD 树的构建过程．图源：PBRT 第三版

同时，为了剪枝，每当发现一个光线与图元的交点时，都需要进行记录更新，在之后的搜索中只需要搜索比这个点更近的交点即可．同时在遍历内部节点时，我们也可以记录光线的传播方向，优先对距离光线较近的节点进行遍历．

15.4.3　KD 树

二元空间分割（binary space partitioning，BSP）树是一种使用平面对空间进行自适应划分的空间索引结构．BSP 树的根节点是一个包裹整个场景的大包围盒．如果包围盒中图元的数目大于一个特定的阈值，该包围盒就会从中被一个平面分为两半．图元则会被关联到与其相重叠的半边包围盒上．若一个图元同时与两边的包围盒相重叠，则会同时被关联到两个包围盒上，而在层次包围盒中一个图元最终只会被关联到左右两个的包围盒的其中一个之上．正是由于这样的特性使得在层次包围盒在构建开始之前就能够知道需要消耗多少内存并提前分配，而 BSP 树则没有办法做到这一点．

KD 树（KD-Trees）和八叉树（Octrees）是两种常见的 BSP 树的变种．如图15.5所示，KD 树与 BSP 树的区别在于它限定了划分平面必须与某一个坐标轴垂直．这样做的好处在于它可以使得遍历和构建的效率得到极大提高，代价则是在空间划分时失去了一定的的灵活度．八叉树则是使用三个分别与 x，y 和 z 轴垂直的平面同时将整个包围盒划分为八部分．我们在这里将重点介绍 KD 树．

KD 树的构建

KD 树的构建与层次包围盒类似，同样是一个自顶向下的递归过程．或将当前节点包围盒所包围的区域划分为两个子区域；或当前节点中的图元数目低于阈值，将其转化为叶节点并将与包围盒重叠的图元与该节点相关联，终止递归．同时，我们需要规定一个 KD 树的最大深度，因为在某些极特殊的情况下，KD 树可能会无限递归下去，在 PBRT[2] 中建议的最大深度为 $8 + 1.3\log(N)$，N 代表图元的数目．

在对 KD 树进行划分的时候，我们同样可以用我们之前在介绍层次包围盒时提到的

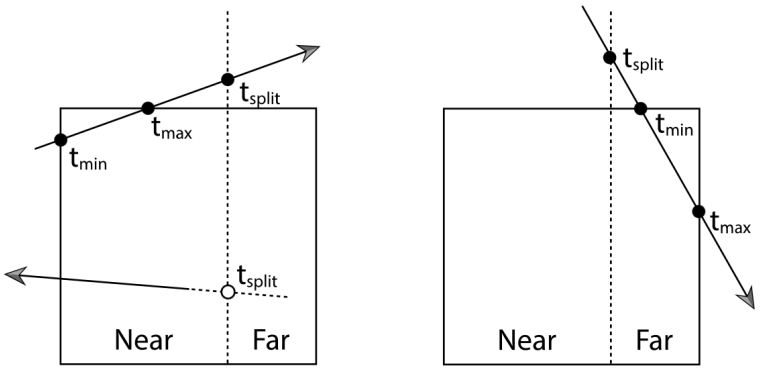

图 15.6: 判断光线与几个子节点相交．图源：PBRT 第三版

SAH 方法来估计代价．但一个不同的地方在于此时我们更加偏向于划分出的两个子节点中存在一个空节点，这样当光线通过该节点所包围区域的时候就能不经过求交快速地舍弃该节点并前进至下一个节点．因此，这里需要对代价函数进行一定的修改，增加一个奖励项 b_e，仅当两个待划分节点中的任意一个为空节点时 $b_e = 1$.

$$c(A,B) = (1 - b_e)\left(p(A)\sum_{i\in A} t(i) + p(B)\sum_{i\in B} t(i)\right) + t_{trav} \tag{15.16}$$

我们观察这个代价的计算式子，我们就会发现，如果分割平面不是出现在包围盒的某一个面上，那么其代价一定会不低于在面上进行分割的代价，所以我们只需要比较所有在包围盒面上进行划分的情况，从中选出代价最小的即可．当然如果在构建过程中，遇到了无法找到划分的位置；或求出的最优划分的代价高于不进行划分的代价时，我们只能放弃对当前节点的划分直接将当前节点变为叶节点．

KD 树的遍历

KD 树遍历的流程大致如下：首先光线与 KD 树的根节点的包围盒求交，若不相交则函数可以直接返回；若相交则继续遍历．对于每一个内部节点，首先计算光线和划分平面的交点，据此可以判断光线是与该节点的一个子节点相交还是同时与两个子节点相交，若同时相交于两个子节点需要按照远近顺序由近及远地依次遍历子节点．对于叶节点只需要对与其关联的图元进行逐一地求交即可．这里与层次包围盒不同的是，因为 KD 树是根据空间进行自适应划分的空间索引结构，我们可以保证优先与较近的物体进行求交，所以并不需要遍历完整个 KD 树就可以找到交点．

关于判断光线是与几个子节点相交，我们可以通过是否满足 $t_{split} \in [\max(0, t_{min}), t_{max}]$ 来判断，这里 t_{min} 表示光线相交的较近点，t_{max} 表示光线相交的较远点，t_{split} 表示光线与分隔平面的相交点．如果如图15.6所示，不在这个范围之内，那么就说明光线仅与一个子节点相交．