Dezeming Family

PBRT 系列 25-高级积分器-随机渐进式光子映射

Art. No 0
ISBN 000–00–0000–00–0
Edition 0.0

Cover design by Dezeming Family

Published by Dezeming
Printed in China

目录

前言及简介

DezemingFamily 系列文章和电子书**全部都有免费公开的电子版**，可以很方便地进行修改和重新发布。如果您获得了 *DezemingFamily* 的系列电子书，可以从我们的网站 *[https://dezeming.top/]* 找到最新的版本。对文章的内容建议和出现的错误也欢迎在网站留言。

0.1 本文前言

简单的渲染积分器只需要从相机发出光线，而双向方法则同时从相机和光源发出光路径，然后将每次散射的顶点连接起来。光子映射也是一种双向方法，但是它不需要连接子路径，而是在场景中发射光子。双向路径追踪的原理可以参考论文 [2]，或者参考 DezemingFamily 的《模拟光传输的鲁棒的蒙特卡洛方法（Eric Veach）全文解读》。由于在 PBRT 系列 24 的第一章中介绍过双向方法的相机和测量的内容，因此我们在这里忽略这些内容。

光子映射的基本概念可以参考 DezemingFamily 的《零基础实现一个光子映射器》的内容。我们这里会讲解随机渐进式光子映射。

1. 粒子追踪理论基础

光子映射属于粒子追踪方法，本章讲解粒子追踪技术的基本理论。

1.1 粒子追踪

我们将首先介绍粒子跟踪算法的理论，并讨论粒子跟踪算法必须满足的条件，以便使用该算法创建的粒子正确计算任意测量值。

粒子追踪就是假设光源发出光粒子，这些粒子一部分会在模型表面沉积，一部分会被反射到其它方向。粒子追踪的难点在于如何理解这种粒子传输的效应。为了为粒子跟踪提供坚实的理论基础，我们将使用 Veach 引入的框架对其进行描述，该框架将存储的粒子历史解释为场景平衡辐射分布的样本。在粒子分布和合理计算权重的条件下，粒子可以用于计算场景中的光分布的几乎任何测量的估计值。

粒子追踪算法在点 p_j，$j \in N$ 个照明样本，记录这些照明样本的信息（ω_j 表示入射方向，β_j 表示吞吐量）：

$$(p_j, \omega_j, \beta_j) \tag{1.1.1}$$

β_j 包含了吞吐量函数 T 和相关采样 PDF 的比率，与路径追踪的 β 非常相似（下一节我们进行回顾）。我们想确定粒子位置的权重和分布的条件，以便我们可以使用它们正确计算任意测量的估计值。给定描述要进行的测量的重要性函数 $W_e(p,\omega)$，我们希望满足的自然条件是粒子应被分布和加权，以便使用它们计算估计值与相同重要性函数的测量方程具有相同的期望值：

$$E\left[\frac{1}{N}\sum_{j=1}^{N}\beta_j W_e(p_j,\omega_j)\right] = \int_A\int_{S^2} W_e(p,\omega)L_i(p,\omega)|\cos\theta|dAd\omega \tag{1.1.2}$$

例如，我们可能想使用粒子来计算入射到墙上的总通量。使用通量 (flux) 的定义：

$$\Phi = \int_{A_{wall}}\int_{\mathcal{H}^2(\mathbf{n})} L_i(p,\omega)|\cos\theta|dAd\omega \tag{1.1.3}$$

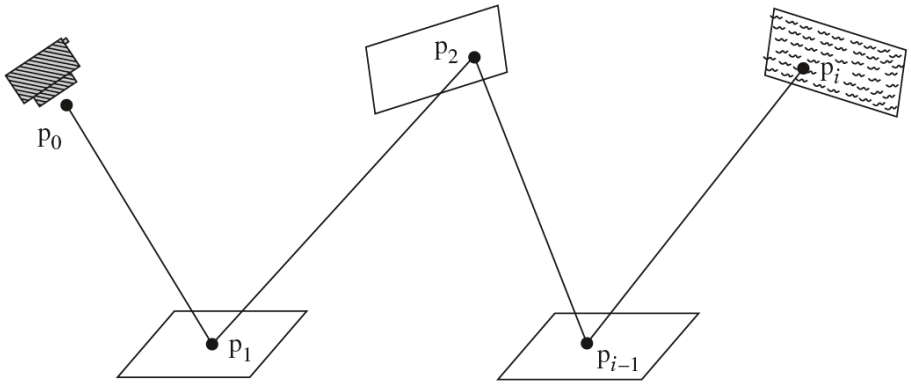
下面的 $W_e(p,\omega)$ 是用来筛选粒子的：

$$W_e(p,\omega)=\begin{cases}1 & p\ is\ on\ wall\ surface\ and\ \ (\omega\cdot\mathbf{n})>0\\ 0 & otherwise\end{cases} \tag{1.1.4}$$

如果式2.1.3成立，则通量估计可以直接计算为墙壁上粒子的粒子加权和。如果我们想估计不同墙上的通量，或者墙的一部分区域的通量，我们只需要用更新的重要性函数重新计算加权和。粒子和重量可以重复使用，我们对所有这些测量都有一个无偏的估计（然而，估计值将相互关联，这可能是图像 artifacts 的来源）。

1.2 路径追踪中 β 值的含义

在路径追踪算法里，路径从相机的某个顶点 p_0 开始沿着相机光线方向采样，在采样到的顶点处通过采样 BSDF 得到方向来找到新的顶点和方向。



对于一条路径，假设除了最后一个顶点以外其它顶点都是通过采样 BSDF 得到的（最后一个顶点是通过采样光源，即采样表面来得到的）：

$$\begin{aligned}L=&\frac{f(p_2\to p_1\to p_0)|\cos\theta_1|}{p_\omega(p_2-p_1)}\times\frac{f(p_3\to p_2\to p_1)|\cos\theta_2|}{p_\omega(p_3-p_2)}\times\frac{f(p_4\to p_3\to p_2)|\cos\theta_1|}{p_\omega(p_3-p_2)}\\&\times\cdots\cdots\times\frac{f(p_{i-1}\to p_{i-2}\to p_{i-3})|\cos\theta_{i-2}|}{p_\omega(p_{i-1}-p_{i-2})}\times\frac{L_e(p_i\to p_{i-1})f(p_i\to p_{i-1}\to p_{i-2})G(p_i\leftrightarrow p_{i-1})}{p_A(p_i)}\end{aligned} \tag{1.2.1}$$

$$=\left(\prod_{j=1}^{i-2}\frac{f(p_{j+1}\to p_j\to p_{j-1})|\cos\theta_j|}{p_\omega(p_{j+1}-p_j)}\right)\times\frac{L_e(p_i\to p_{i-1})f(p_i\to p_{i-1}\to p_{i-2})G(p_i\leftrightarrow p_{i-1})}{p_A(p_i)} \tag{1.2.2}$$

我们令：

$$\beta=\left(\prod_{j=1}^{i-2}\frac{f(p_{j+1}\to p_j\to p_{j-1})|\cos\theta_j|}{p_\omega(p_{j+1}-p_j)}\right) \tag{1.2.3}$$

即 β 值意味着路径吞吐量权重 (path throughput weight)，也就是吞吐量函数 $T(\overline{p}_{i-1})$（在 PBRT 书 [1] 的 14.4.4 节中定义的吞吐量函数是用路径面积分形式定义的）。

1.3 粒子与测量

要了解如何生成满足这些条件的粒子并对其进行加权，需要考虑估计测量方程积分：

$$\begin{aligned}&\int_A \int_{S^2} W_e(p_0,\omega) L(p_0,\omega) |\cos\theta| d\omega dA(p_0) \\ &= \int_A \int_A W_e(p_0 \to p_1) L(p_1 \to p_0) G(p_0 \leftrightarrow p_1) dA(p_0) dA(p_1)\end{aligned} \tag{1.3.1}$$

我们可以从场景中采样一系列的 p_0 点和 p_1 点，然后通过蒙特卡洛方法来估计上面的结果。假设采样 N 个样本的蒙特卡洛估计器（$p(\cdot)$ 表示采样到某点的概率）表示如下：

$$E\left[\frac{1}{N}\sum_{i=1}^{N} W_e(p_{i,0} \to p_{i,1}) \left\{\frac{L(p_{i,1} \to p_{i,0}) G(p_{i,0} \leftrightarrow p_{i,1})}{p(p_{i,0}) p(p_{i,1})}\right\}\right] \tag{1.3.2}$$

如果不考虑光的散射，而是光源直接照射到相机里，那么这个式子就可以直接使用，$L(p_{i,1} \to p_{i,0})$ 就是光源上采样的点 $p_{i,1}$ 发射到相机采样点 $p_{i,0}$ 的辐射度（这并不是直接采样光，而是对所有表面进行采样，而只有能够直接发光照射到相机镜头的顶点才会有贡献）。

我们可以把其中采样光路的估计项进行扩展，也就是扩展下式（PBRT 书上下式应该是写错了）：

$$E\left[\frac{L(p_{i,1} \to p_{i,0})}{p(p_{i,1})}\right] \tag{1.3.3}$$

假如我们目前关注第 i 个样本（即第 i 条光路，这是一个逐步构建的过程，相当于不断发射光子），将上面的式子可以扩展为 n_i 项的和：

$$\begin{aligned}\beta_i = \frac{L_e(p_{i,n_i} \to p_{i,n_i-1})}{p(p_{i,n_i})} &\times \frac{1}{1-q_{i,n_i-1}} \frac{f(p_{i,n_i} \to p_{i,n_i-1} \to p_{i,n_i-2}) G(p_{i,n_i} \leftrightarrow p_{i,n_i-1})}{p(p_{i,n_i-1})} \\ &\times \cdots\cdots \\ &\times \frac{1}{1-q_{i,2}} \frac{f(p_{i,3} \to p_{i,2} \to p_{i,1}) G(p_{i,3} \leftrightarrow p_{i,2})}{p(p_{i,2})} \\ &\times \frac{1}{1-q_{i,1}} \frac{f(p_{i,2} \to p_{i,1} \to p_{i,0}) G(p_{i,2} \leftrightarrow p_{i,1})}{p(p_{i,1})}\end{aligned}$$

规范一下可以表示为：

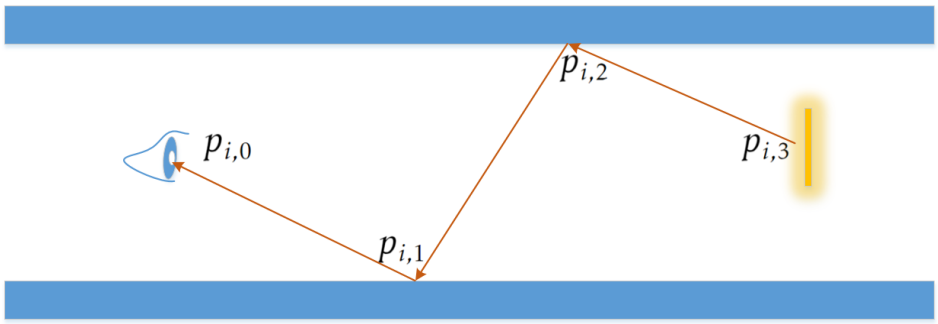
$$\beta_i = \frac{L_e(p_{i,n_i} \to p_{i,n_i-1})}{p(p_{i,n_i})} \prod_{j=1}^{n_i-1} \frac{1}{1-q_{i,j}} \frac{f(p_{i,j+1} \to p_{i,j} \to p_{i,j-1}) G(p_{i,j+1} \leftrightarrow p_{i,j})}{p(p_{i,j})} \tag{1.3.4}$$

假设只考虑光反射一次：

$$\beta_{i,1} = \frac{L_e(p_{i,n_i} \to p_{i,n_i-1})}{p(p_{i,n_i})} \times \frac{1}{1-q_{i,n_i-1}} \frac{f(p_{i,n_i} \to p_{i,n_i-1} \to p_{i,n_i-2}) G(p_{i,n_i} \leftrightarrow p_{i,n_i-1})}{p(p_{i,n_i-1})}$$

注意这里的几何项 G 说明了此时度量的概率密度采样是相当于表面面积的采样概率，而不是基于 BSDF 的立体角采样。虽然方法是从灯光上的点开始，并使用路径顶点处的 BSDF 对路径进行增量式采样，类似于路径跟踪积分器生成路径的方式（这里从灯光开始，而不是从相机开始），但我们可以使用任何数目的不同采样策略来生成路径。

我们再用图示更好地描述一下，假设有一条光路：



一个样本来估计的光照可以写为：

$$\begin{aligned} & W_e(p_{i,0} \to p_{i,1}) \frac{G(p_{i,0} \leftrightarrow p_{i,1})}{p(p_{i,0})} \frac{L(p_{i,1} \to p_{i,0})}{p(p_{i,1})} \\ = W_e(p_{i,0} \to p_{i,1}) & \frac{G(p_{i,0} \leftrightarrow p_{i,1})}{p(p_{i,0})} \times \frac{f(p_{i,2} \to p_{i,1} \to p_{i,0}) G(p_{i,2} \leftrightarrow p_{i,1})}{p(p_{i,1})} \\ & \times \frac{f(p_{i,3} \to p_{i,2} \to p_{i,1}) G(p_{i,3} \leftrightarrow p_{i,2})}{p(p_{i,2})} \\ & \times \frac{L_e(p_{i,3} \to p_{i,3-1})}{p(p_{i,3})} \end{aligned} \tag{1.3.5}$$

注意在该式中，对于某一个像素而言，估计该像素的光子需要考虑两个方面，一是镜头上的采样点，二是光源发射过来的位置，如果光源采样点 $p_{i,1}$ 与镜头采样点 $p_{i,0}$ 之间的连线并没有通过该像素，则我们认为 $W_e(p_{i,0} \to p_{i,1})$ 为 0 或者非常小。我们再把上式展开的过程写得更详细一点，这里将 $L(p_{i,1} \to p_{i,0})$ 展开为：

$$\begin{aligned} & L(p_{i,1} \to p_{i,0}) \\ = & \int_A f(p_{i,2} \to p_{i,1} \to p_{i,0}) G(p_{i,2} \leftrightarrow p_{i,1}) L(p_{i,2} \to p_{i,1}) dA(p_{i,2}) \\ \approx & f(p_{i,2} \to p_{i,1} \to p_{i,0}) G(p_{i,2} \leftrightarrow p_{i,1}) \frac{L(p_{i,2} \to p_{i,1})}{p(p_{i,2})} \end{aligned} \tag{1.3.6}$$

重复上述过程：

$$\begin{aligned} & L(p_{i,2} \to p_{i,1}) \\ = & \int_A f(p_{i,3} \to p_{i,2} \to p_{i,1}) G(p_{i,3} \leftrightarrow p_{i,2}) L(p_{i,3} \to p_{i,2}) dA(p_{i,3}) \\ \approx & f(p_{i,3} \to p_{i,2} \to p_{i,1}) G(p_{i,3} \leftrightarrow p_{i,2}) \frac{L(p_{i,3} \to p_{i,2})}{p(p_{i,3})} \end{aligned} \tag{1.3.7}$$

这样就能得到前面的式子1.3.5了。

我们应该在每个点处如何记录光子信息呢？因为对于 $p_{i,1}$ 点来说，我们不知道光的反射方向，所以我们可以在 $p_{i,1}$ 点存储下面的值：

$$\begin{aligned} \beta_{i,1} = \frac{L_e(p_{i,3} \to p_{i,3-1})}{p(p_{i,3})} & \times \frac{1}{1-q_{i,2}} \frac{f(p_{i,3} \to p_{i,2} \to p_{i,1}) G(p_{i,3} \leftrightarrow p_{i,2})}{p(p_{i,2})} \\ & \times \frac{1}{1-q_{i,1}} \frac{G(p_{i,2} \leftrightarrow p_{i,1})}{p(p_{i,1})} \end{aligned}$$

当然，在实际光子追踪过程中，我们并不是采样表面，而是采样光子散射方向，因此前面的内容需要转化为立体角采样的形式，我们会在最后一个章节将整个流程串起来。

我们也可以把光子传播的过程理解为功率的传播，在网上针对光子映射技术的入门描述中基本上都是用功率来描述的，有很多这方面的资料。

粒子跟踪带来的关键优势是，我们可以只生成一次样本和相关权重，然后重复使用它们，有效地减少计算量。

2. 光子映射理论基础

本章讲解光子映射技术的基本理论。

2.1 光子映射基本思想

光子映射算法基于将粒子跟踪到场景中并模糊它们的贡献，来近似着色点处的入射照明。为了与算法的其他描述保持一致，我们将为光子映射生成的粒子称为光子。

为了去计算一个点处的反射辐射度，我们需要去估计出射辐射度方程（在点 p 以方向 ω_o 出射的辐射度），可以等效地被写为场景中表面上所有点的测量（并使用 Dirac delta 分布来表示仅仅在点 p 的反射）：

$$\begin{aligned}&\int_{S^2} L_i(p,\omega_i) f(p,\omega_o,\omega_i)|\cos\theta_i| d\omega_i \\ &= \int_A \int_{S^2} \delta(p-p') L_i(p',\omega_i) f(p',\omega_o,\omega_i)|\cos\theta_i| d\omega_i dA(p')\end{aligned} \tag{2.1.1}$$

也就是测量方程写为：

$$W_e(p',\omega) = \delta(p'-p) f(p,\omega_o,\omega) \tag{2.1.2}$$

该式有一个 delta 项。

在漫反射表面根据 BSDF 采样方向时，永远不可能与点光源相交，因此只能采样点光源。基于无法直接采样到 delta 项的这种思维，我们考虑下式估计：

$$E\left[\frac{1}{N}\sum_{j=1}^{N}\beta_j W_e(p_j,\omega_j)\right] = \int_A \int_{S^2} W_e(p,\omega) L_i(p,\omega)|\cos\theta| dA d\omega \tag{2.1.3}$$

假如我们的相机采样光线与平面中的某个点相交，我们想估计这个点处的反射光，这是不可能的，因为 $W_e(p_j,\omega_j)$ 中存在 delta 项，而我们发射并存储的光子恰好在该点的概率为 0（delta 分布）。

我们假设当前采样点的光照是周围区域的光照的近似，因此光子映射其实可以理解为对周围区域的光照进行插值（这个插值和近似的过程就会带来偏差）。我们可以把这个 delta 函数近似为一个滤波器函数（比如一个高斯函数），用周围邻域的光照值加权作为当前点的光照值（离着当前点越近的光源点权重越高）。

有助于光子映射效率的一个因素是光子的重复使用：费尽心思计算光传输路径，允许其在多个点潜在地贡献照明，从而分摊了生成光的成本。虽然光子映射从光子复用这种效率的改进中获得了一些好处，但在某个点使用附近的光子有一个更微妙但更重要的好处：一些光传输路径不可能使用基于增量路径控制的无偏算法进行采样。

例如，渲染一张有一堆玻璃的图像，而只有点光源来照明，则在玻璃上是无法使用采样光源这种策略的，且采样 BSDF 也永远不会与点光源相交。即使是面光源，折射的光线与光源相交时也会带来很大的误差。使用光子映射我们可以跟踪离开光线的光子，让它们在玻璃中折射，并在漫反射表面上沉积照明。有了足够数量的光子，表面将被密集覆盖，某点周围的光子可以很好地估计在该点的入射光照。

2.2 光子与密度估计

先明确目标，我们需要测量：

$$E\left[\frac{1}{N}\sum_{j=1}^{N}\beta_j\delta(p'-p)f(p,\omega_o,\omega)\right] \tag{2.2.1}$$

注意这里的 W_e 是包含 $BSDF$ 项的，原因是因为前面计算 β_i 时，在每个点记录的 β_i 值不能包含反射到下一个方向的 BSDF（因为我们并不知道相机光线是从哪个方向采样到当前点的）。

一种称为密度估计 (density estimation) 的统计技术提供了执行这种插值的数学工具。密度估计在假设样本根据某个感兴趣函数的总体分布来分布的情况下，构造给定一组样本点的 PDF。

直方图就是这个想法的一个简单例子。在 1D 中，线被划分为具有一定宽度的间隔，人们可以计算每个间隔中有多少个采样，并进行归一化使间隔的面积总和为 1。

核方法是一种更复杂的密度估计技术。它们通常提供更好的结果和更平滑的 PDF，不会像直方图那样出现不连续性（相当于给一定邻域范围内的样本不同的权重）。给定一个积分到 1 的核函数 $k(x)$：

$$\int_{-\infty}^{\infty}k(x)dx=1 \tag{2.2.2}$$

对于在位置 x_i 的 N 个样本的核估计器来说：

$$\hat{p}(x)=\frac{1}{Nh}\sum_{i=1}^{N}k\left(\frac{x-x_i}{h}\right) \tag{2.2.3}$$

其中 h 是窗口宽度（也称为平滑参数或内核带宽）。内核方法可以被认为是在观察点放置一系列凸起，其中凸起的总和形成 PDF，因为它们各自积分为 1，并且总和被归一化。h 如何选择是一个很重要的方法，如果它太宽，概率密度函数就会被模糊，如果太窄，概率密度函数就会很陡峭，覆盖区域的样本就不会很多。

假设 $h=1$（单位宽度），当前点设为 x，所有存在光子的点 x_i 都恰好在 x 上，那么估计式就是：

$$\hat{p}(x)=\frac{1}{N}\sum_{i=1}^{N}k\Big(0\Big)=k(0) \tag{2.2.4}$$

有时候可能期望 $k(0)=1$（并不是必须的），表示当所有点都集中在一个点上时，该点密度为 1。

最近邻技术是一种很好的自适应局部密度的方式，当样本量很多时，核宽度就会比较小，当样本量很少时，核宽度就会很大。例如，定义一个 N 值，在当前点 x 处找到最近的 N 个样本，使用距离 $d_N(x)$ 来作为窗宽。这是一般化的 n 个最近邻估计：

$$\hat{p}(x)=\frac{1}{Nd_N(x)}\sum_{i=1}^{N}k\Big(\frac{x-x_i}{d_N(x)}\Big) \tag{2.2.5}$$

对于 d 维的情况：

$$\hat{p}(x)=\frac{1}{N(d_N(x))^d}\sum_{i=1}^{N}k\Big(\frac{x-x_i}{d_N(x)}\Big) \tag{2.2.6}$$

代入到测量方程中，可以得到对测量的合适的估计，即点 p 的出射辐射度为：

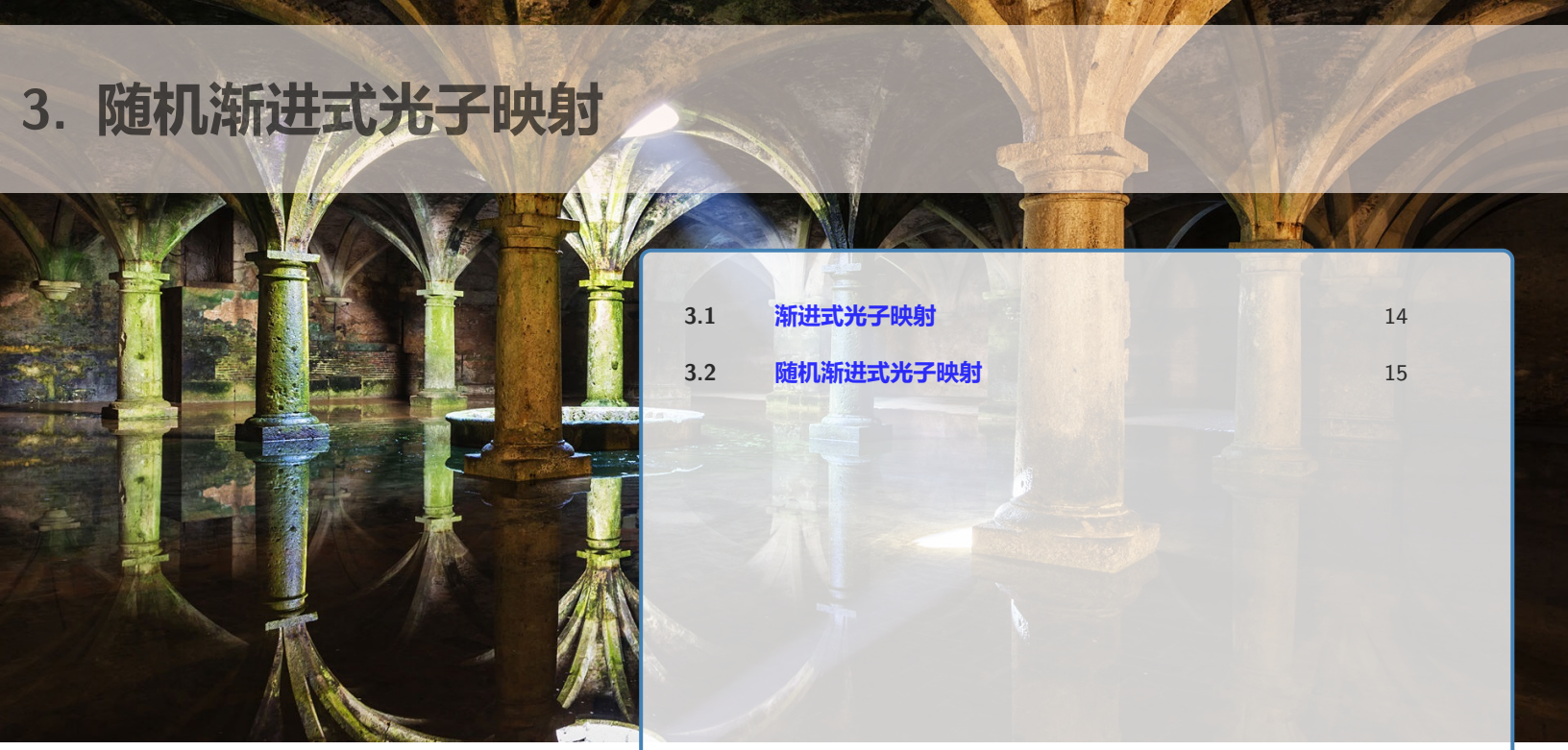
$$L_o(p,\omega_o)\approx\frac{1}{N_pd_N(p)^2}\sum_{j}^{N_p}k\Big(\frac{p-p_j}{d_N(p)}\Big)\beta_jf(p,\omega_o,\omega_j) \tag{2.2.7}$$

N_p 表示发射光子的总数，由于较远处的光子的核函数值较小，所以我们只需要考虑计算最近的 N 个光子和。

光子映射的误差是很难量化的，但我们知道，增加追踪的光子量总能够提高结果的质量。当光子越多时，不必在最近邻估计中使用更远处的光子。通常，任何给定点的误差将取决于照明在其周围的变化速度，在实践中，这种错误通常不会太糟。由于插值的步骤很倾向于模糊照明，因此有时使用光子映射来重建照明中高频变化的效果不佳。如果在一个渲染系统中，传统方法用于直接照明，光子映射用于简介照明，则会效果较好，因为间接照明往往是低频的。

光子映射最初是基于一种两步算法，首先从光源跟踪光子。光子相互作用被记录在场景的表面上，然后被组织在空间数据结构（通常是 kd 树）中，以便在渲染时使用。第二遍遵循从相机开始的路径；在每个路径顶点，附近的光子用于估计间接照明。虽然这种方法是有效的，但由于必须存储所有光子，因此可以使用的光子数量受到可用内存的限制。一旦内存满了，即使你想通过使用更多的光子获得更高质量的结果，也无法进一步提升效果。（相比之下，例如，在路径跟踪中，每个像素总是可以添加更多的样本，而不会增加任何存储成本，只会增加计算成本。）

3. 随机渐进式光子映射



本章讲解从光子映射技术过渡到随机渐进式光子映射的原理和方法。

3.1 渐进式光子映射

渐进光子映射算法通过对算法重构解决了这个问题：**首先**，相机过程跟踪从相机开始的路径。对于每个像素而言，像素范围内会发出一系列的采样射线，进行路径采样。这些路径的所有非镜面反射路径顶点都被存储在一个像素内。（例如，如果相机光线射到漫反射表面，我们可能会记录关于交点和漫反射的几何信息。如果光线射到完美镜面反射表面，然后再射到漫反射表面，我们会记录由镜面反射 BSDF 值缩放的漫反射信息，以此类推。）我们将在下面将这些存储的路径顶点称为可见点。**然后**，跟踪来自光源的光子；在每个光子与表面相交处，光子都会对附近可见点的反射辐射估计做出贡献。

为了理解渐进光子映射是如何工作的，我们考虑将 LTE 分解为每个顶点处的直接入射辐射度 L_d 和间接入射辐射度 L_i 的独立积分（正如路径追踪中的那样）：

$$\begin{aligned} L(p,\omega_o) &= L_e(p,\omega_o) + \int_{S^2} f(p,\omega_o,\omega_i) L(p,\omega_i) |\cos\theta_i| d\omega_i \\ &= L_e(p,\omega_o) + \int_{S^2} f(p,\omega_o,\omega_i) L_d(p,\omega_i) |\cos\theta_i| d\omega_i \\ &\quad + \int_{S^2} f(p,\omega_o,\omega_i) L_i(p,\omega_i) |\cos\theta_i| d\omega_i \end{aligned} \tag{3.1.1}$$

间接光照的计算有两种方式，第一种是像路径追踪那样不断跟踪路径和方向；第二种是通过光子来估计简介光照，即通过将光子权重与路径中的顶点序列的路径吞吐量权重的乘积相加，可以计算出该点对相机胶片平面上的辐射的最终贡献。对于完全镜面反射的 BSDF，唯一合理的选择是继续跟踪路径：镜面反射中光子永远不会到达与 BSDF 中的增量分布完全匹配的正确方向。对于高光泽的表面也建议继续跟踪路径，因为它需要很多光子才能到达狭窄的镜面波瓣以计算准确的估计值。

对于漫反射表面，使用光子是很有效的方式，尽管跟踪最后一次反弹也是值得的。这个过程被叫做最终聚集 (final gathering)，如果光子仅仅被一次漫反射反弹后就使用，那么光子密度不足导致的任何误差通常都不太明显，尽管可能需要跟踪更多的相机路径以消除噪声。

这种方法不需要存储任何光子，可以追踪任意数量的光子，而内存限制转为了存储可见点和它们的反射信息。对于高分辨率的图像需要求解运动模糊或者景深时，内存容量会受限（每个像素需要更多相机路径，因此需要存储更多顶点）。

3.2 随机渐进式光子映射

随机渐进式光子映射 (SPPM, Stochastic progressive photon mapping) 不会受到任何内存限制、正如渐进式光子映射，它会产生一堆相机可见顶点，但是相对来说采样率较低；然后会发出一系列光子，在临近的可见点处积累贡献。之后，丢弃以前的可见顶点，继续生成新的顶点和位置。

SPPM 一共做了两点改进。

第一：因为估计只是在 2D 空间中进行的（只在可见点所处的局部的切线平面），因此（注意 N_p 是光源发射的光子总数，需要注意的是，对于一个固定功率的光源，每个光子携带的能量需要除以它发出的光子数量）：

$$L_o(p,\omega_o) \approx \frac{1}{N_p \pi r^2} \sum_j^{N_p} \beta_j f(p,\omega_o,\omega_j) \tag{3.2.1}$$

相当于除以圆碟形区域的面积。

第二：基于首先在渐进光子映射中实现的方法，随着更多光子贡献到可见点，逐渐减小光子搜索半径。一般的想法是，随着在搜索半径内发现更多的光子，我们就会有更多的证据表明有足够的光子到达来估计入射辐射度分布。通过减小半径，我们确保未来使用的光子将更接近该点，从而有助于更准确地估计入射辐射分布。

减小半径意味着如何调整反射辐射度估计，因为下式计算的光子和会来自于不同的辐射度：

$$L_o(p,\omega_o) \approx \frac{1}{N_p d_N(p)^2} \sum_j^{N_p} k\left(\frac{p-p_j}{d_N(p)}\right) \beta_j f(p,\omega_o,\omega_j) \tag{3.2.2}$$

设 N_i 表示在 i 此迭代以后，贡献到某个点的光子数；M_i 表示当前迭代期间贡献的光子数；r_i 是第 i 次迭代的搜索半径；τ 是光子与 BSDF 值的乘积之和；Φ_i 是在 i 次迭代计算的：

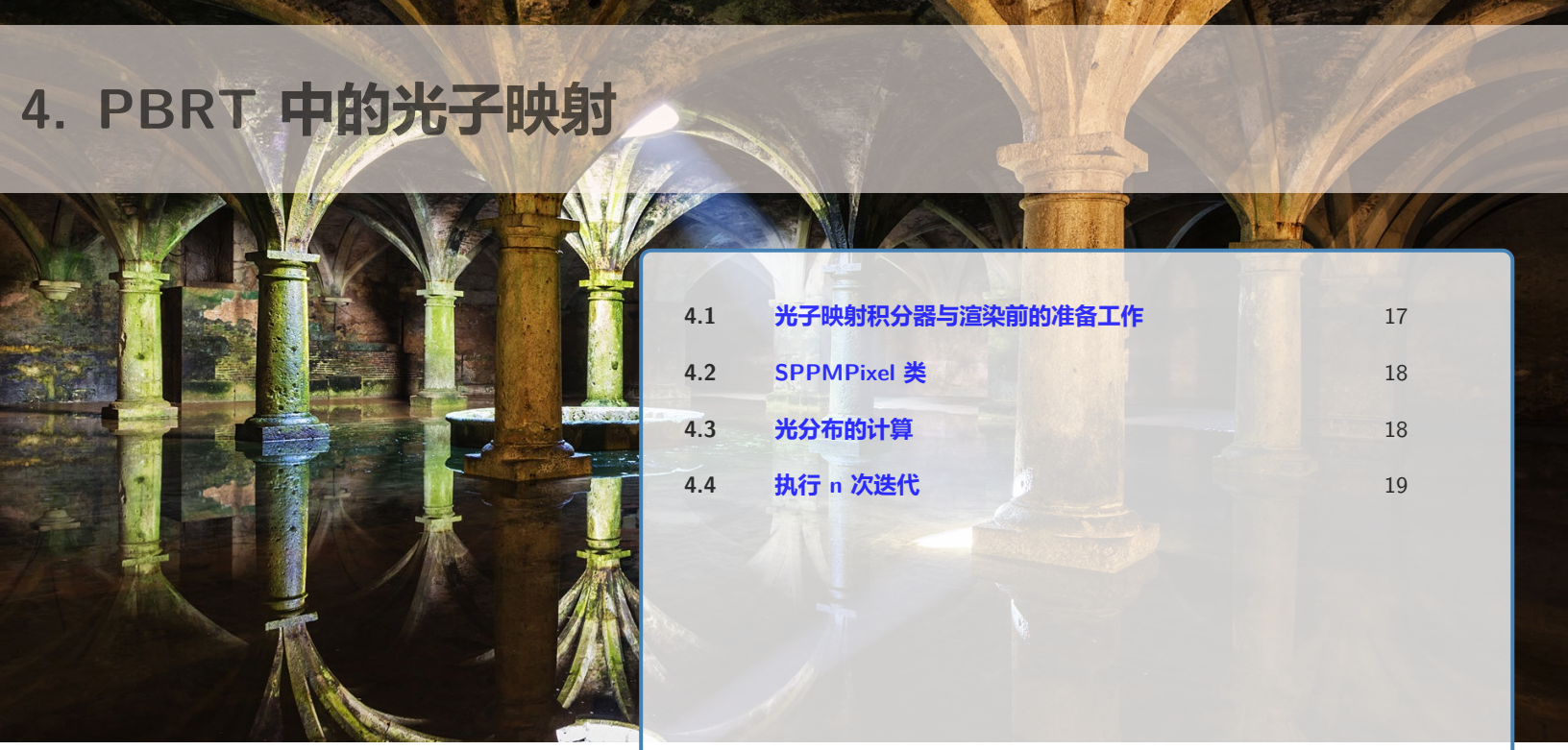
$$\Phi_i = \sum_j^{M_j} \beta_j f(p,\omega_o,\omega_j) \tag{3.2.3}$$

设 γ 参数 (一般设 2/3) 决定了早期迭代中的光子的贡献（具有更宽的搜索半径）被淡出的速度。更新公式为：

$$\begin{aligned} N_{i+1} &= N_i + \gamma M_i \\ r_{i+1} &= r_i \sqrt{\frac{N_{i+1}}{N_i + M_i}} \\ \tau_{i+1} &= (\tau_i + \Phi_i)\frac{r_{i+1}^2}{r_i^2} \end{aligned} \tag{3.2.4}$$

注意半径是每像素属性而不是每可见点属性（也就是说一个像素的所有可见点都要共用同一个半径）。值得注意的是，即使在像素中的所有可见点序列上共用该半径值，也可以计算出反射辐射度的一致的估计值（暂不证明）。当追踪的光子数量 $N_p \to \infty$，半径 $r \to \infty$，反射辐射度估计是一致的，并收敛到正确值。

4. PBRT 中的光子映射



本章讲解 *PBRT* 中的光子映射的代码。

4.1 光子映射积分器与渲染前的准备工作

SPPMIntegrator 直接继承自 Integrator，它有自己的 Render() 方法。

在执行了一些初始设置之后，它运行 SPPM 算法的多次迭代，找到一组可见点，然后在它们的所处位置上通过光子累积照明。每次迭代都会在每个像素中创建一个从相机开始的新路径，这有助于消除几何边缘的锯齿，并对运动模糊和景深进行采样。Render() 函数描述如下：

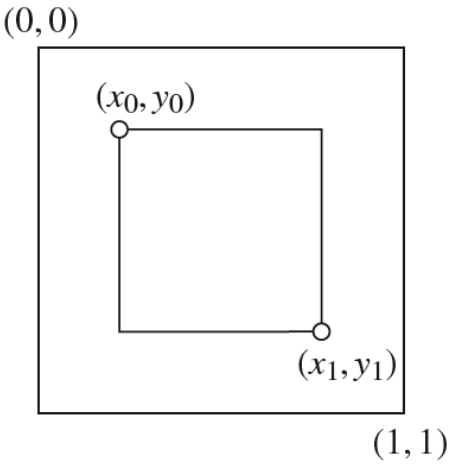
```
1 void SPPMIntegrator::Render(const Scene &scene) {
2   // 初始化SPPM的像素数组
3   ......
4   // 计算正比于能量的光分布，用于采样光源
5   ......
6   // 执行SPPM的n次迭代
7   ......
8 }
```

定义 pixels 数组来为每个像素存储一个 SPPMPixel 对象（会在后面定义）。并给每个 pixels 中的元素赋予一个初始的搜索半径。这个半径不能太大，也不能太小。初始化 SPPM 的像素数组的代码为：

```
1   Bounds2i pixelBounds = camera->film->croppedPixelBounds;
2   int nPixels = pixelBounds.Area();
3   std::unique_ptr<SPPMPixel[]> pixels(new SPPMPixel[nPixels]);
4   for (int i = 0; i < nPixels; ++i) pixels[i].radius =
        initialSearchRadius;
```

```
5   // invSqrtSPP的作用仅用于缩放光线微分，类似于samplesPerPixel的平方根
        对：
6   // ray.ScaleDifferentials(1 / std::sqrt((Float)tileSampler->
        samplesPerPixel));
7   const Float invSqrtSPP = 1.f / std::sqrt(nIterations);
8   pixelMemoryBytes = nPixels * sizeof(SPPMPixel);
```

在相机的 film 上，剪切窗口 (crop window) 用来指定需要被渲染的图像区域，它的坐标是 (0,0)–(1,1) 区间：



Film::croppedPixelBounds 存储坐标边界（与分辨率有关），当计算出分数像素坐标时，需要向上取整（防止每个子区域渲染的像素之间会重叠）。

4.2 SPPMPixel 类

SPPMPixel 类存储三个内容，第一，存储当前估计的一个像素范围内可见的平均辐射度（包括快门打开的时间和景深（如果存在））；第二，存储与像素的光子密度估计相关的参数（N_i、M_i 以及 τ 等）；第三，存储一个像素中相机路径得到的可见点的几何和反射信息。

因为这些存储的信息的形式等需要依赖于实际操作相关（比如并行中的原子操作），等后面讲到源码时再详细介绍 SPPMPixel 类。

4.3 光分布的计算

使用加权的光子可以实现两种完全不同的总体采样方法。

第一种方法中，我们可以尝试以近似于表面辐照度 (irradiance) 的权重均匀分布光子，同时考虑不连续性和其他重要的几何和照明特征。然而，这种类型的光子分布对于一般输入来说很难实现（因为我们很难根据表面辐照度来分布光子，我们也很难控制光子的分布）。因此，我们将努力生成具有相同（或相似）权重的光子，从而使其在整个场景中的变化密度代表照明的变化（辐照度越高的地方光子密度越高）。

此外，如果表面上的光子权重存在与辐照度无关的实质性变化，则可能会出现令人不快的图像伪影：如果一个光子的权重比其他光子大得多，则在该光子对内插辐射有贡献的场景区域会显示出明亮的圆形伪影。

因此，我们希望从更亮的光中发射更多的光子，这样离开所有光的光子的初始权重将具有相似的量级，因此，根据由光的各自功率定义的 PDF 来选择每个光子路径的起始点。来自较亮灯光的光子数量较多，说明它们对场景中照明的贡献更大（而不是来自所有灯光的相同数量的光子，更强大的灯光权重更大）。

4.4 执行 n 次迭代

SPPM 算法的每次迭代都会在每个像素处跟踪来自相机的新路径，然后在每个路径的端点收集入射光子。

通常，迭代次数越多，可以改善可见边、运动模糊和景深的采样；随着每个像素中累积的光子越多，间接照明估计就越精确。请注意，这里的欠采样伪影是低频斑点，这是一种与路径跟踪中的欠采样高频噪声截然不同的视觉伪影。

这里的实现使用 HaltonSampler 生成相机路径；这样做可以确保在聚合 (aggregate) 的所有迭代中使用分布良好的样本。总流程参考如下表示：

```
1  // 执行_nIterations_次的SPPM迭代
2    // 采样器
3    HaltonSampler sampler(nIterations, pixelBounds);
4    // 计算需要用来渲染的tiles
5    ......
6    for (int iter = 0; iter < nIterations; ++iter) {
7      // 产生SPPM的可见点
8      // 追踪光子并积累贡献
9      // 根据本次过程的光子来更新像素值
10     // 将SPPM图像存储在胶片中
11     // 重置内存存储
12   }
```

以上就是 SPPM 的工作框架。下一章中我们详细解释迭代中的这些步骤是如何实现的。

5. 光子映射代码详解

本章讲解 *PBRT* 中的光子映射的可见点构建和光子积累等代码具体是如何工作的。

5.1 并发与采样

与路径跟踪器不同，在估计每个顶点的直接照明并对每个顶点的出射方向进行采样后，可以丢弃 BSDF，这里我们需要存储可见点的 BSDF 直到完成当前迭代的光子传递。因此，用于在相机路径跟踪期间分配 BSDF 的 MemoryArenas 不会在相机路径的循环迭代结束时重置（在计算完光子贡献后才重置）。

还要注意，我们只为每个工作线程分配一个 arena，用于运行并行 for 循环，并使用 ThreadIndex 全局索引到向量中，而不是为每个循环迭代分配一个单独的 arena。通过这种方式，我们避免了拥有多个独立的 MemoryArena 的开销，同时仍然确保每个 arena 不被多个处理线程使用（被多个处理线程使用将导致 MemoryArrena 方法中的竞争条件）。注意我们这 nIterations 次迭代是顺序执行的，并没有开启多线程。

```
1  std::vector<MemoryArena> perThreadArenas(MaxThreadIndex());
2  for (int iter = 0; iter < nIterations; ++iter) {
3    // 往SPPM中增加可见点
4    ParallelFor([&](int pixelIndex){.....}, nPixels, 4096);
5    // 追踪光子并积累贡献
6    ParallelFor([&](int photonIndex){}, photonsPerIteration, 8192);
7    // 从这个过程的光子中更新像素值
8    ParallelFor([&](int i){}, nPixels, 4096);
9    // 重置 arenas
10   for (int i = 0; i < perThreadArenas.size(); ++i)
11     perThreadArenas[i].Reset();
12 }
```

我们回顾一下并发的两个函数：

```
void ParallelFor(std::function<void(int64_t)> func, int64_t count, int
    chunkSize);
void ParallelFor2D(std::function<void(Point2i)> func, const Point2i &
   count);
```

我们以前学习过 ParallelFor2D 的用法：PBRT 会把整个图像分成一堆 16×16 的小片段 (tiles)，每个线程负责渲染一个小片段里面的内容。当图像分辨率的长或者宽不是 16 的倍数时，会通过下面的方式进位：

```
const int tileSize = 16;
Point2i nTiles((sampleExtent.x + tileSize - 1) / tileSize, (
   sampleExtent.y + tileSize - 1) / tileSize);
```

对于 ParallelFor 函数，参数 count 给出了执行循环体的次数，chunkSize 参数表示每个线程执行多少次循环（count 需要大于 chunkSize）。

关于采样器的设置，我们也单独捋出来：

```
// 执行nIterations次迭代
HaltonSampler sampler(nIterations, pixelBounds);
for (int iter = 0; iter < nIterations; ++iter) {
  ParallelFor2D([&](Point2i tile) {
    int tileIndex = tile.y * nTiles.x + tile.x;
    // tileIndex作为随机数种子
    std::unique_ptr<Sampler> tileSampler = sampler.Clone(tileIndex);
    Bounds2i tileBounds(Point2i(x0, y0), Point2i(x1, y1));
    for (Point2i pPixel : tileBounds) {
      tileSampler->StartPixel(pPixel);
      tileSampler->SetSampleNumber(iter);
      // 对每个像素产生相机光线
      CameraSample cameraSample = tileSampler->GetCameraSample(pPixel)
          ;
    }
  }, nTiles);
}
```

在追踪光子时都是使用 RadicalInverse 函数来作为采样随机数：

```
  uint64_t haltonIndex = (uint64_t)iter * (uint64_t)
     photonsPerIteration + photonIndex;
  RadicalInverse(haltonDim, haltonIndex);
```

5.2 积累相机可见点

积累可见点的全部代码可见：

```
for (Point2i pPixel : tileBounds) {
  <(1) 为每个pPixel准备采样器>
  <(2) 为每个像素发射相机射线>
  <(3) 沿着相机射线走，直到发现一个可见点>
}
```

前两步和以前的方法没有什么太大区别，关键在于第三步，这是一个循环的过程。在循环开始前，需要计算像素在存储区内的偏置（线性索引），即 pixelOffset 值：

```
  Point2i pPixelO = Point2i(pPixel - pixelBounds.pMin);
  int pixelOffset =
      pPixelO.x +
      pPixelO.y * (pixelBounds.pMax.x - pixelBounds.pMin.x);
  SPPMPixel &pixel = pixels[pixelOffset];
```

然后开始循环：

```
bool specularBounce = false;
for (int depth = 0; depth < maxDepth; ++depth) {
  SurfaceInteraction isect;
  ......
}
```

循环可以分为下面几个步骤：

```
  <如果与场景没有交点，就积累外部光源的辐射度>
  <处理SPPM相机的射线相交>
  {
    <(1) 计算交点处的BSDF>
    <(2) 在相机射线交点处积累直接照明>
    <(3) 可能会创建一个可见点，并终止相机路径>
    <(4) 从顶点位置再次产生(spawn)光线>
  }
```

配合源码，SPPMPixel::Ld 内存储了相机路径总所有顶点的发射的和反射的直接光照，也就是下

式的前两项：

$$\begin{aligned} L(p,\omega_o) &= L_e(p,\omega_o) + \int_{S^2} f(p,\omega_o,\omega_i) L(p,\omega_i) |\cos\theta_i| d\omega_i \\ &= L_e(p,\omega_o) + \int_{S^2} f(p,\omega_o,\omega_i) L_d(p,\omega_i) |\cos\theta_i| d\omega_i \\ &\quad + \int_{S^2} f(p,\omega_o,\omega_i) L_i(p,\omega_i) |\cos\theta_i| d\omega_i \end{aligned} \tag{5.2.1}$$

前两项也在通过采样 BSDF 和跟踪第三项的光线而找到的顶点处进行评估，因此，Ld 不只是在第一个顶点存储直接照明，而是直到创建某个点为可见点时该点的直接光照都要囊括进来。由于这个辐射度记录值 Ld 包含了一个像素中的所有样本的和，所以必须将此值除以 SPPMIntegrator::nIterations 来得到最终像素辐射度估计的直接光照的平均值。

(1) 计算交点处的 BSDF，该过程较为简单：

```
1  isect.ComputeScatteringFunctions(ray, arena, true);
2  if (!isect.bsdf) {
3      ray = isect.SpawnRay(ray.d);
4      --depth;
5      continue;
6  }
7  const BSDF &bsdf = *isect.bsdf;
```

SPPMIntegrator 不支持参与介质的渲染，所以当不存在 bsdf 时（一般是参与介质的表面），光线穿过表面继续前进。

(2) 在相机射线交点处积累直接照明，该过程将表面发出的光和直接光照进行累加。

Knowledge 5.1 (光源辐射度重复累加的问题) 我们可能会担心采样到下一个方向后，相机射线与表面交点处恰好是光源，则上一个点计算的直接光照和光源表面的发光会重复累加。我们要注意改代码防止了这种情况：

```
1      Vector3f wo = -ray.d;
2      if (depth == 0 || specularBounce)
3          pixel.Ld += beta * isect.Le(wo);
4      pixel.Ld +=
5          beta * UniformSampleOneLight(isect, scene, arena, *
               tileSampler);
```

depth 为 0，说明相机第一次追踪光线；上一次发生的是镜面反射，意味着也没有直接光照被捕获。所以在这两种情况下才会累加表面发出的光照。

(3) 可能会创建一个可见点，并终止相机路径。如果当前点的反射 BSDF 是漫反射则直接退出循环，将当前点作为可见顶点；或者光滑表面（非镜面表面）且当前已经达到了最后一次追踪的深度（depth == maxDepth - 1），那么将当前点作为可见顶点；否则就继续循环追踪。

```
// Possibly create visible point and end camera path
bool isDiffuse = bsdf.NumComponents(BxDFType(
                     BSDF_DIFFUSE | BSDF_REFLECTION |
                     BSDF_TRANSMISSION)) > 0;
bool isGlossy = bsdf.NumComponents(BxDFType(
                    BSDF_GLOSSY | BSDF_REFLECTION |
                    BSDF_TRANSMISSION)) > 0;
if (isDiffuse || (isGlossy && depth == maxDepth - 1)) {
    pixel.vp = {isect.p, wo, &bsdf, beta};
    break;
}
```

(4) 从顶点位置再次产生 (spawn) 光线，当还没有达到最后一次追踪的深度（depth < maxDepth - 1）时，就采样散射方向，此时和路径追踪一样，会使用俄罗斯轮盘来决定光线的终止。

5.3 构建可见点网格

当我们追踪光子的时候，我们需要求追踪的光子对哪些相机可见点会有贡献，因此需要对得到的可见点数组（即数组变量 pixels，pixels 为每个像素点都存储了一个可见顶点）进行一些处理。

当光子与表面相交时，为了有效地找到离着光子交点最近的距离小于当前搜索半径 r_i 的可见点所在的像素，在所有可见点的包围盒上使用统一网格。大致过程如下：

```
<创建所有SPPM可见点的网格>
{
  <(1) 给SPPM可见点分配网格>
  <(2) 给SPPM可见点计算网格边界>
  <(3) 计算SPPM网格在每个维度上的分辨率>
  <(4) 把所有可见点加入到网格中>
}
```

5.3.1 给 SPPM 可见点分配网格

这个网格通常是稀疏结构，许多体素里都没有可见点。（首先，网格中的任何体素中没有表面结构时，都不可能有可见点）因此，网格不是为所有体素分配存储空间，而是由哈希表表示，其中哈希函数将三维体素坐标转换为网格阵列中的索引：

```
int hashSize = nPixels;
std::vector<std::atomic<SPPMPixelListNode *>> grid(hashSize);
```

由于需要用多线程方法来并行构建网格，这些线程需要通过原子操作来更新网格，所以使用了std::atomic。

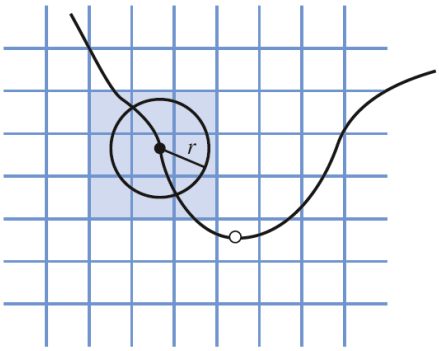
每个网格小格子 (grid cell) 内都会存储一个链接列表，列表的每个节点都指向一个 SPPMPixel 对象，该对象的可见点的搜索体空间与网格单元是有重叠区域的。注意一个可见点可能会跟很多网格小格子都有重叠，因此保证节点的紧凑表示很有用（即只存储指向 SPPMPixel 对象的指针，而不是存储整个 SPPMPixel 对象的备份）。

```
struct SPPMPixelListNode {
  SPPMPixel *pixel;
  SPPMPixelListNode *next;
};
```

5.3.2 给 SPPM 可见点计算网格边界

如果当前迭代的像素没有可见点（相机射线没有与任何物体相交，或者是在与漫反射表面相交之前终止），则该像素将具有路径吞吐量权重 $\beta = 0$（并且不应尝试在该像素的网格中放置可见点）。

如果当前迭代的像素有可见点，此处计算以可见点为中心的边界框，范围为 $\pm r_i$，即像素的当前光子搜索半径（给定搜索半径为 r 的可见点（下图实心小圆点），可见点将被添加到半径为 r_i 的球体的边界框重叠的所有网格单元中的链接列表中）：



当我们稍后产生了一个光子交点时，我们只需要考虑光子所在的网格单元的可见点，就可以找到光子可能贡献的可见点（给定一个光子入射到场景中的一个表面上（上图中空心小圆点），我们只需要检查光子所在的体素中的可见点，就可以找到它可能贡献的可见点）。由于不同的可见点将具有不同的搜索半径，这取决于迄今为止为其像素贡献的光子数量。如果存储可见点而不考虑它们将从中接收光子的空间体积，则查找光子的可能相关的可见点将非常困难。

(2) 给 SPPM 可见点计算网格边界：

```
  Float maxRadius = 0.;
  for (int i = 0; i < nPixels; ++i) {
    const SPPMPixel &pixel = pixels[i];
    // 如果可见点beta==0，则跳过
    if (pixel.vp.beta.IsBlack()) continue;
    // 这是一个不断扩张的过程，查看需要囊括全部可见点所需的包围盒范围
```

```
    Bounds3f vpBound = Expand(Bounds3f(pixel.vp.p), pixel.radius);
    gridBounds = Union(gridBounds, vpBound);
    // 计算所有可见点中最大的搜索半径
    maxRadius = std::max(maxRadius, pixel.radius);
  }
```

5.3.3 计算 SPPM 网格在每个维度上的分辨率

上一节的代码中可以计算出网格的总体边界，我们需要决定体素应该有多大，从而决定细分空间的精细程度。一方面，如果体素太大将效率低下，因为每个光子都必须检查许多可见点，以查看它是否对每个点都有贡献。如果体素太小，那么每个可见点将重叠许多体素，网格所需的内存将会过多。

因此这里我们计算一个初始分辨率，使得网格维度中最大的一维的体素宽度大致等于所有可见点的最大当前搜索半径。这限制了任何可见点可以重叠的体素的最大数量。然后设置网格的其他两个维度的分辨率，以便体素在所有维度上具有大致相同的宽度。

```
Vector3f diag = gridBounds.Diagonal();
Float maxDiag = MaxComponent(diag);
int baseGridRes = (int)(maxDiag / maxRadius);
CHECK_GT(baseGridRes, 0);
for (int i = 0; i < 3; ++i)
  gridRes[i] = std::max((int)(baseGridRes * diag[i] / maxDiag), 1);
```

5.3.4 把所有可见点加入到网格中

然后就要把可见点塞入到网格中。由于网格和可见点的 BSDF 都需要一直被存储着，直到光子追踪完毕才释放，因此我们可以重用每个线程的 MemoryArenas。代码过程如下：

```
ParallelFor([&](int pixelIndex) {
  MemoryArena &arena = perThreadArenas[ThreadIndex];
  SPPMPixel &pixel = pixels[pixelIndex];
  if (!pixel.vp.beta.IsBlack()) {
    <处理每个可见点，添加到网格中>
  }
}, nPixels, 4096);
```

之后要把每个可见点加入到它的包围盒重叠到的网格小格子里：

```
Float radius = pixel.radius;
Point3i pMin, pMax;
ToGrid(pixel.vp.p - Vector3f(radius, radius, radius),
       gridBounds, gridRes, &pMin);
```

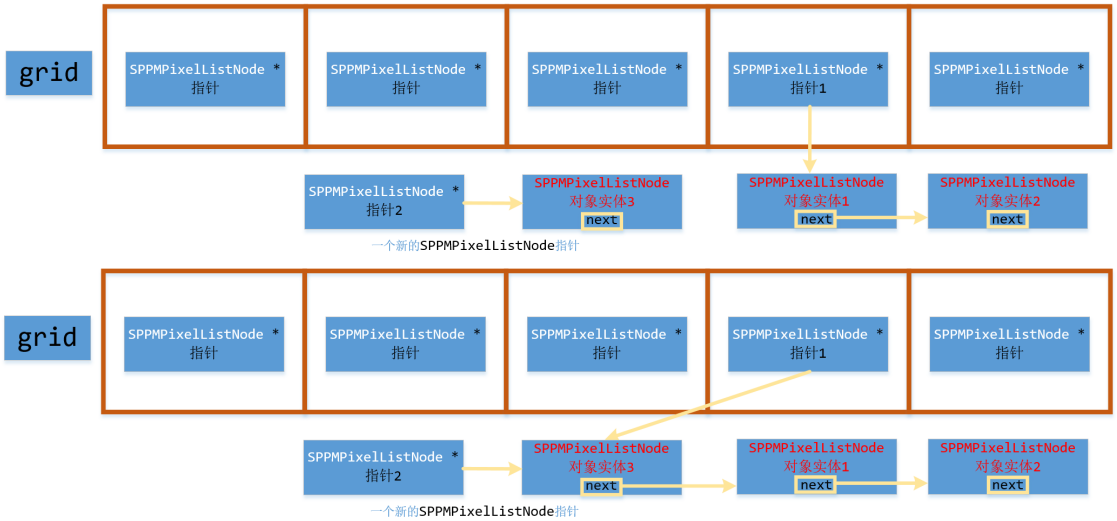
```
ToGrid(pixel.vp.p + Vector3f(radius, radius, radius),
       gridBounds, gridRes, &pMax);
for (int z = pMin.z; z <= pMax.z; ++z)
    for (int y = pMin.y; y <= pMax.y; ++y)
        for (int x = pMin.x; x <= pMax.x; ++x) {
          <把可见点加入到网格小格子$(x, y, z)$上>
        }
```

ToGrid() 函数给出了给定点所在的体素坐标，这里通过将可见点的三个坐标分量减去最大半径，得到最小坐标；并通过将可见点的三个分量分别加上半径，得到最大坐标，最小坐标和最大坐标就构成了所在网格范围。

在“把可见点加入到网格小格子 (x,y,z) 上”中，首先计算 hash 码，用来找到对应的格子的索引（注意这里的 hash 码计算结果可能也会有冲突，意味着不同的网格点存储到表中相同的位置里，但这样只是会增加光子的查找量，并不会引起错误）。

```
int h = hash(Point3i(x, y, z), hashSize);
SPPMPixelListNode *node = arena.Alloc<SPPMPixelListNode>();
node->pixel = &pixel;
// 原子地把node加到grid[h]的链接表的开始位置
node->next = grid[h];
while (grid[h].compare_exchange_weak(node->next, node) == false);
```

原子操作可以在多线程中将可见点加入到网格中，而不需要去支持任何线程锁。我们先说一下过程，我们的目标是将上面的结果变到下面的结果（next 是指向表中下一个实体的指针）。



在没有并发的情况下，我们仅仅希望 node->next 指向 grid[h]，并且把 node 归于 grid[h]。当多线程并发地更新链表时，这种方法将不会很好地工作：比如当一个 node->next 指向 grid[h] 后，另一个线程又修改了 grid[h]，会带来错误。上面的 compare_exchange_weak() 方法可以有效解决这个问题，它会在其他线程来修改该值时做出判断，通过循环最终得到目标结果（希望进一步了解的读者可以自行去 C++ 标准库官方文档查询）。

5.4 追踪光子并积累贡献

光子同样通过多线程方式来追踪，每个线程重新分配内存空间，而不是使用之前 BSDF 和网格构建时使用的那个内存池。

```
std::vector<MemoryArena> photonShootArenas(MaxThreadIndex());
ParallelFor([&](int photonIndex) {
  MemoryArena &arena = photonShootArenas[ThreadIndex];
  <追踪光子路径>
  arena.Reset();
}, photonsPerIteration, 8192);
```

在为每次 SPPM 迭代选择要跟踪的光子数量时，需要达到一个平衡：太多的话，则像素的半径不会随着更多光子的到达和太多太远的光子的使用而减少（每次迭代用更多光子，减少迭代次数，因此半径更新的次数就会变少，注意当半径没有有效减少的时候，由于半径较大，估计出来的焦散效果会比较模糊）；太少的话，找到可见点并制作它们的网格的时间成本开销将无法在足够的光子上分摊。在实践中，每次迭代中几十万到几百万的光子数量通常效果很好。

前面说过，光子映射会使用 RadicalInverse 来生成随机数。Halton 序列为所有迭代中的所有光子路径提供了一组分布良好的采样点。haltonIndex 记录当前光子的 Halton 序列的索引；它也可以被视为追踪光子的全局指数。换句话说，对于第一个光子，haltonIndex 从 0 开始，并逐步增加。使用 64 位整数作为该值很重要，因为 32 位整数在大约 20 亿个光子之后会溢出；高质量图像可能需要更多的光子。

```
uint64_t haltonIndex = (uint64_t)iter * (uint64_t)
    photonsPerIteration + photonIndex;
int haltonDim = 0;
```

追踪光子并积累贡献的过程分为下面几个步骤，我们分为四个小结来介绍：

```
<(1) 选择发射光子的光源>
<(2) 计算光子离开光源的采样值>
<(3) 从光源生成photonRay，并初始化beta>
<(4) 光子路径生成，并记录交点>
```

5.4.1 选择发射光子的光源

lightPdf 记录了要采样哪个光源的概率，Halton 序列的第一个维度被用来作为采样随机数（对于 m 个 Halton 序列来说，在每一个维度上，样本分布都是非常均匀的，这保证了初始采样中样本的均匀分布）。

```
Float lightPdf;
Float lightSample = RadicalInverse(haltonDim++, haltonIndex);
int lightNum =
```

```
4       lightDistr->SampleDiscrete(lightSample, &lightPdf);
5   const std::shared_ptr<Light> &light = scene.lights[lightNum];
```

5.4.2 计算光子离开光源的随机样本值

并且需要两个二维随机数，一个采样发光点位置，一个采样方向。这三个随机数就是 uLight0（二维随机数，采样位置）、uLight1（二维随机数，采样方向）和 uLightTime（采样时间）。因为光源可能也是移动的，所以需要在快门内采样光源的时间，使得发出的光线会携带时间信息。

值得一提的是，PBRT-V3 我目前的版本里，程序代码中，DiffuseAreaLight::Sample_Le() 函数并没有用到 time，我认为是代码中的一个 bug。

5.4.3 从光源生成 photonRay，并初始化 beta

β 值计算基于：

$$\beta = \frac{|\cos\omega_0| L_e(p_0, \omega_0)}{p(light)p(p_0, \omega_0)} \tag{5.4.1}$$

$p(light)$ 是采样到某个光源的概率密度，$p(p_0, \omega_0)$ 是在这个光源上采样到这个点和这个方向的概率密度。

注意，几何项 $G(p_0 \leftrightarrow p_1)$ 中还包含光源光线与物体求交以后距离因素，但是这个余弦因子 $|\cos\omega_0|$ 必须要集成到 β 中（这是因为我们生成的路径是采样光线方向来得到的，因此我们其实并不需要用采样面积的表示法，也就是说我们并不需要几何项）。在最后一章中我们还会对 SPPM 中的各个部分的权重进行更精确地分析和描述。

```
1   SurfaceInteraction isect;
2   for (int depth = 0; depth < maxDepth; ++depth) {
3     if (!scene.Intersect(photonRay, &isect)) break;
4     if (depth > 0) {
5       <把光子贡献到周围的可见点上>
6     }
7     <采样新的光子方向>
8   }
```

5.4.4 光子路径生成，并记录交点

该过程即光子生成路径，并不断更新 β 值。

注意光子在第一个交点处不会做任何贡献，这是因为这种情况是属于直接照明的，当相机与该处也有交点时，可以通过直接照明来计算结果。在后续的交点处才会对临近可见点贡献照明。

如果有交点，就把光子贡献到周围的可见点上。ToGrid 函数将光子定位到相关网格，然后计算该网格的哈希值定位到网格内容索引，再在该网格小格子内查找光子是否在某些可见点半径范围内：

```
// 把光子贡献到周围的可见点上
Point3i photonGridIndex;
if (ToGrid(isect.p, gridBounds, gridRes, &photonGridIndex)) {
    int h = hash(photonGridIndex, hashSize);
    < 把光子贡献加到grid[h]位置处的可见点上。>
}
```

回想一下，网格存储指向 SPMPixelListNodes 的 std::atomic 指针。通常，从 std::：atomic 值读取内容意味着编译器必须小心，不要在读取 grid[h] 值时重新排序读取或写入内存的指令；这个约束是必要的，这样无锁算法就可以按预期工作。但在当前的这种情况下，网格已经构建，没有其他线程同时修改它，因此，使用 std::atomic.load() 方法是值得的，并让它知道没有这些约束的“松弛 (relaxed)”内存模型可以用于读取初始网格指针。这种方法具有显著的性能优势：对于几百个三角形的简单场景（其中没有太多时间用于跟踪光线），在 2015 年的 CPU 上使用这种内存模型，光子过程的运行时间减少了 20%。

```
for (SPPMPixelListNode *node = grid[h].load(std::memory_order_relaxed)
    ; node != nullptr; node = node->next) {
    SPPMPixel &pixel = *node->pixel;
    Float radius = pixel.radius;
    if (DistanceSquared(pixel.vp.p, isect.p) > radius * radius)
        continue;
    // 更新pixel的$\Phi$和$M$
    Vector3f wi = -photonRay.d;
    Spectrum Phi = beta * pixel.vp.bsdf->f(pixel.vp.wo, wi);
    for (int i = 0; i < Spectrum::nSamples; ++i) pixel.Phi[i].Add(Phi[
        i]);
    ++pixel.M;
}
```

上面的 pixel.vp.wo 就是根据相机采样到可见点时的采样射线的反方向得到的。pixel.M 存储了本次迭代中的光子的总贡献数。注意 SPPMPixel::M 和 SPPMPixel::Phi[] 是定义了原子操作的（因为 Spectrum 不支持原子更新，因此使用 AtomicFloat 来实现）：

```
AtomicFloat Phi[Spectrum::nSamples];
std::atomic<int> M;
```

这里提及一下前面描述过的 Φ 的计算式：

$$\Phi_i = \sum_j^{M_j} \beta_j f(p, \omega_o, \omega_j) \tag{5.4.2}$$

记录光子贡献后，需要选择光子新的散射方向并更新 β 值。在前面描述中，$\beta_{i,j}$ 表示第 i 个

光子的第 j 次相交的 β 权重，散射后，一个新的顶点 $p_{i,j+1}$ 被采样到，权重设置为：

$$\beta_{i,j+1} = \beta_{i,j} \frac{1}{1-q_{i,j+1}} \frac{f(p_{i,j+1} \to p_{i,j} \to p_{i,j-1}) G(p_{i,j+1} \leftrightarrow p_{i,j})}{p(p_{i,j+1})} \tag{5.4.3}$$

需要将面积表示的采样权重转换为采样角度的权重（实际路径是通过采样立体角来构建的，采样场景的表面积实施起来不但困难，而且方差会很大，容易采样到不可见表面）：

$$\beta_{i,j+1} = \beta_{i,j} \frac{1}{1-q_{i,j+1}} \frac{f(p,\omega,\omega')|\cos\theta'|}{p(\omega')} \tag{5.4.4}$$

下一章我们会通过一个实例，来对随机渐进式光子映射中的权重计算与统计进行解释。

程序代码中剩下的步骤我们已经遇到了太多次，因此这里就不再赘述了。但是需要注意的是，光子路径采样中计算表面散射 BSDF 时，会用到重要性传输（折射中的非对称散射问题也要考虑进来）：

```
1 // 在光子交点处计算散射BSDF
2 isect.ComputeScatteringFunctions(photonRay, arena, true, TransportMode
    ::Importance);
```

顺便值得一提的是，代码应尽量小心地执行光子散射步骤，以使光子权重尽可能彼此相似。Jensen 提出了一种方法，在所有光子的权重完全相等的情况下，给出光子的分布。首先，计算交点处的反射率。然后随机决定是否继续光子的路径，其概率与该反射率成比例。如果光子继续，其散射方向将通过从 BSDF 的分布中采样来找到，继续构建光子路径但是它的权重不变（除了根据表面的颜色调整光谱分布之外）。因此，那些反射很少光的表面将反射很少的到达它的光子，但那些被散射的光子将以不变的贡献继续传播，依此类推。（这种特殊的方法在 pbrt 中是不可能的，因为 PBRT 会均匀采样里面的每一种 BxDF，其采样分布可能并不和真实的 BSDF 分布完全一致。况且对很多复杂的 BSDF 来说，很难根据其分布进行精确采样。）

5.5 从光子中更新像素值

追踪迭代完全部光子以后，就从光子中更新像素值：

```
1  for (int i = 0; i < nPixels; ++i){
2    SPPMPixel &p = pixels[i];
3    if (p.M > 0) {
4      <更新像素内光子数、搜索半径和$\tau$值>
5      // 重置M值和Phi
6      p.M = 0;
7      for (int j = 0; j < Spectrum::nSamples; ++j)
8          p.Phi[j] = (Float)0;
9    }
10   // 重置像素内的VisiblePoint
11   p.vp.beta = 0.;
```

```
12   p.vp.bsdf = nullptr;
13 }
```

更新公式在之前已经全都介绍过了。

5.6 存储 SPPM 图像

存储图像的步骤如下：

```
1  int x0 = pixelBounds.pMin.x;
2  int x1 = pixelBounds.pMax.x;
3  uint64_t Np = (uint64_t)(iter + 1) * (uint64_t)photonsPerIteration;
4  std::unique_ptr<Spectrum[]> image(new Spectrum[pixelBounds.Area()]);
5  int offset = 0;
6  for (int y = pixelBounds.pMin.y; y < pixelBounds.pMax.y; ++y) {
7    for (int x = x0; x < x1; ++x) {
8      // 从pixel中计算辐射度L
9      const SPPMPixel &pixel = pixels[(y - pixelBounds.pMin.y) * (x1 -
           x0) + (x - x0)];
10     Spectrum L = pixel.Ld / (iter + 1);
11     L += pixel.tau / (Np * Pi * pixel.radius * pixel.radius);
12     image[offset++] = L;
13   }
14 }
```

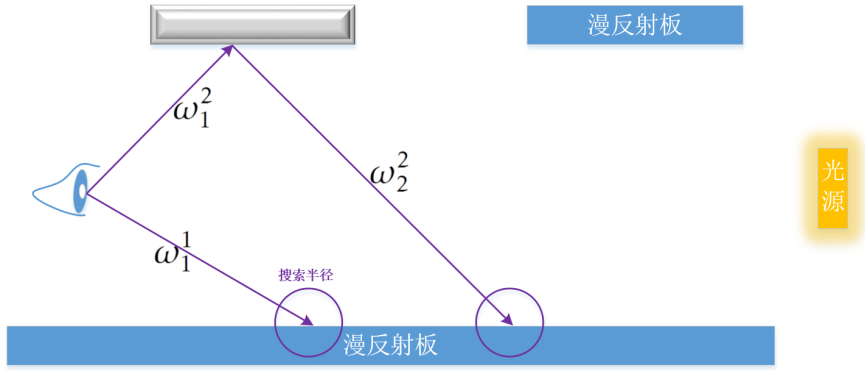
注意 pixel.Ld 值是像素捕获到的直接光照，该值会在每次迭代中不断累加，所以估计结果需要除以 (iter + 1)。

6. 路径与权重的精确分析

本章通过使用一个具体的例子来更好地分析路径与权重方面的内容。当初笔者也是对路径积分、双向方法等内容进行了多次实例分析，才渐渐明白了里面的内容。

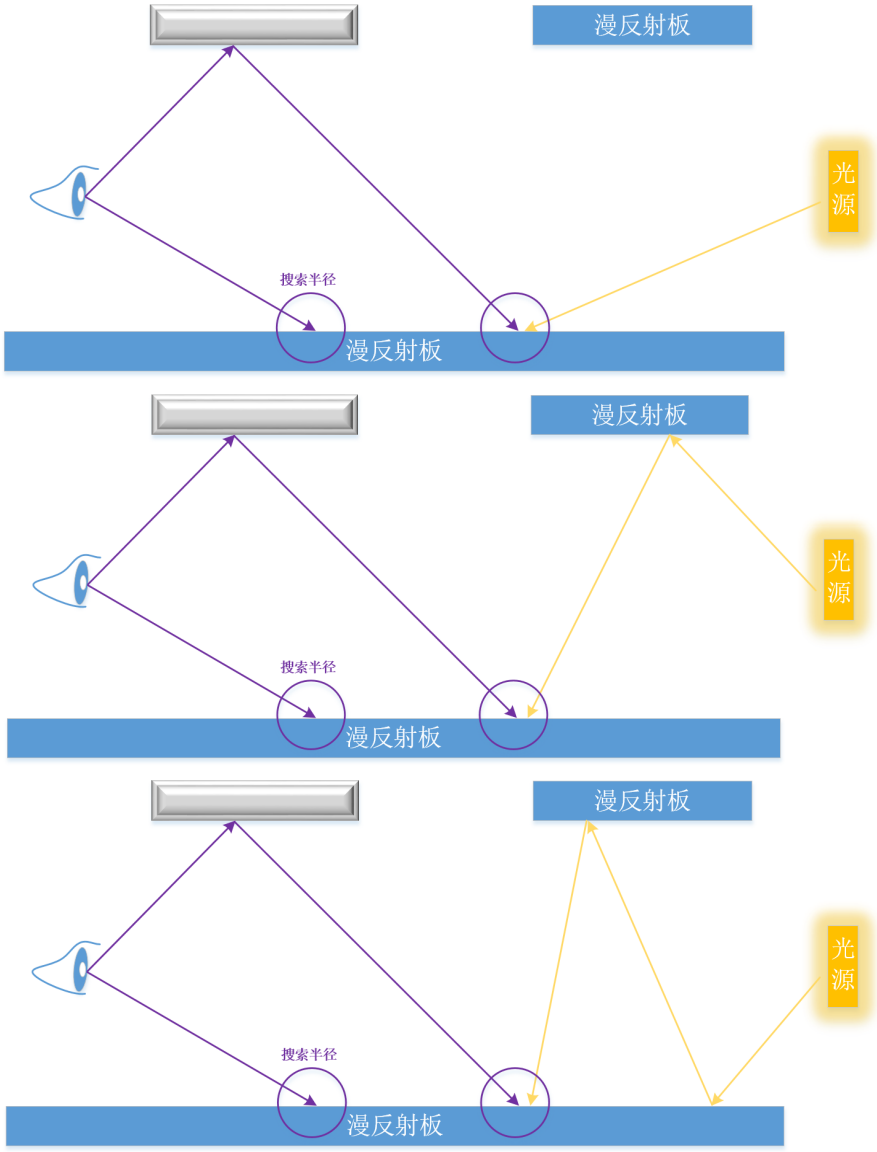
6.1 实例构建

我们构建一个场景如下：



该场景中，有两个漫反射面板，一个白光泽面和一个光源。该场景模拟了两条相机射线，下面这一条相机射线直接与漫反射板相交，记为 p^b，上面这一条相机射线经过光泽反射以后与下面的漫反射板相交，记为 p^t（两个交点分别记录为 p_1^t 和 p_2^t）。

我们模拟三条光子路径，分别是光子直接射到可见点搜索半径内、光子反射一次以后射到可见点搜索半径内、光子反射两次以后射到可见点搜索半径内：



但由于光子第一次反射不被记录

6.2 构建可见点

可见点的构建中，相机射线采样到上面的白光泽面时不会停下，而是继续反射；采样到下面的的漫反射表面时，记录为可见点。

6.2.1 p^b 的构建

由于没有经过反射过程，p^b 点存储的 β 值为 1。

直接光照部分就是：

```
1  pixel.Ld += beta * UniformSampleOneLight(isect, scene, arena, *
       tileSampler);
```

6.2.2 p^t 的构建

在光泽表面上反射时，首先计算直接光照，此时 $\beta = 1$：

```
1  pixel.Ld += beta * UniformSampleOneLight(isect, scene, arena, *
       tileSampler);
```

然后采样散射方向 $\omega_i = \omega_2^2$，β 变为了：

$$\begin{aligned} \beta &:= \beta \cdot \frac{f_1(\omega_o, \omega_2^2) \cdot |\omega_2^2 \cdot n_{p_1^t}|}{p(\omega_2^2)} \\ \Longrightarrow \beta &= \frac{f_1(-\omega_1^2, \omega_2^2) \cdot |\omega_2^2 \cdot n_{p_1^t}|}{p(\omega_2^2)} \end{aligned} \tag{6.2.1}$$

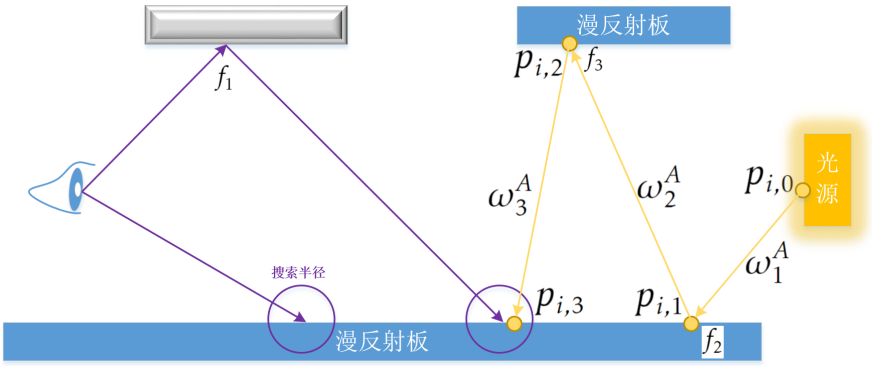
其中，$\omega_o = -\omega_1^2$ 是相机射线方向的反方向，$n_{p_1^t}$ 是在 p_1^t 点处的法向量；$f_1(\omega_o, \omega_2^2)$ 表示在 p_1^t 点处材质的 BSDF 函数，BSDF 函数的第一个参数表示光的射出方向，第一个参数是射入方向的反方向（因为对于一个表面来说，BSDF 要求入射和出射方向都是从表面点向外发出的）。

然后相机射线随着 ω_2^2 方向追踪到点 p_2^t，记录为可见点，并再次采样直接光照。

```
1  pixel.Ld += beta * UniformSampleOneLight(isect, scene, arena, *
       tileSampler);
```

6.3 光子追踪的立体角形式

我们再定义一些符号来描述，下面的 f 表示在各个表面交点处的材质 BSDF：



在各个点所在表面的法向量还是用 n 来表示。此时光子路径的 β 值用 β^L 来描述。

光子路径是用来估计可见点间接照明的，现在开始跟踪光子。设光源为漫反射面光源，首先在光源上采样位置和方向，由于场景中只有一个光源，所以 lightPdf=1。一开始的 β^L 值表示为：

$$\beta^L = \frac{|\omega_1^A \cdot n_{p_{i,0}}| \cdot L}{lightPdf \cdot p(p_{i,0}) p(\omega_1^A)} \tag{6.3.1}$$

光子与 $p_{i,1}$ 相交时，采样伴随 BSDF。注意对于非折射材质来说，采样伴随 BSDF 就是采样原 BSDF，所以不太需要注意些什么。由于使用了俄罗斯轮盘，所以需要除以 $1 - q_1$，我们假设光

线没有被终止：

$$\beta^L := \beta^L \cdot \frac{1}{1-q_1} \frac{f_2(-\omega_1^A, \omega_2^A)|\omega_2^A \cdot n_{p_{i,1}}|}{p(\omega_2^A)} \tag{6.3.2}$$

光子与 $p_{i,2}$ 相交时，采样伴随 BSDF：

$$\beta^L := \beta^L \cdot \frac{1}{1-q_2} \frac{f_3(-\omega_2^A, \omega_3^A)|\omega_2^A \cdot n_{p_{i,2}}|}{p(\omega_3^A)} \tag{6.3.3}$$

光子与 $p_{i,2}$ 相交时，贡献到点 p_2^t（注意这里使用的是 BSDF 而不是伴随 BSDF）：

$$\Phi = \beta^L \cdot f_{p_2^t}(-\omega_3^A, -\omega_2^2) \tag{6.3.4}$$

其中，$f_{p_2^t}$ 就是点 p_2^t 处所在表面的 BSDF。

6.4 计算贡献

假设我们只发射了一个光子，恰好就是上一节中的流程，那么估计得到的贡献就是：

$$\begin{aligned}
&r_{new} = r \times \sqrt{\frac{2/3}{1}} \\
&\tau = \beta \times \Phi \times \frac{r_{new}^2}{r^2} \\
&N_p = 1 \\
&L = Ld + \frac{r_{new}^2}{r^2} \times \frac{1}{N_p \pi r_{new}^2} \beta \times \Phi \approx Ld + \beta \times \Phi
\end{aligned} \tag{6.4.1}$$

$\beta \times \Phi$ 就是估计的间接光照，我们可以写成展开式：

$$\begin{aligned}
\beta \times \Phi = &\frac{f_1(-\omega_1^2, \omega_2^2) \cdot |\omega_2^2 \cdot n_{p_1^t}|}{p(\omega_2^2)} \times f_{p_2^t}(-\omega_3^A, -\omega_2^2) \times \\
&\frac{1}{1-q_2} \frac{f_3(-\omega_2^A, \omega_3^A)|\omega_3^A \cdot n_{p_{i,2}}|}{p(\omega_3^A)} \times \\
&\frac{1}{1-q_1} \frac{f_2(-\omega_1^A, \omega_2^A)|\omega_2^A \cdot n_{p_{i,1}}|}{p(\omega_2^A)} \times \\
&\frac{|\omega_1^A \cdot n_{p_{i,0}}| \cdot L}{lightPdf \cdot p(p_{i,0}) p(\omega_1^A)}
\end{aligned} \tag{6.4.2}$$

大家应该察觉到，它跟双向路径追踪技术计算的光路的式子很是非常一致的，但有一个不同点，就是光源发光时的权重多了一个参数 $|\omega_1^A \cdot n_{p_{i,0}}|$，这个 cos 项其实可以理解为功率，也就是说，下式可以理解为初始的光发射功率：

$$\beta^L = \frac{|\omega_1^A \cdot n_{p_{i,0}}| \cdot L}{lightPdf \cdot p(p_{i,0}) p(\omega_1^A)} \tag{6.4.3}$$

我们在本文中并没有介绍功率和光子追踪之间的关系，因为我们是按照 PBRT 的讲解方法进行的，我们会用一些补充专题的方式来从功率的角度描述一下光子映射的原理。

6.5 表面积采样与立体角采样

再回到原式：

$$W_e(p_{i,0} \to p_{i,1}) \frac{G(p_{i,0} \leftrightarrow p_{i,1})}{p(p_{i,0})} \frac{L(p_{i,1} \to p_{i,0})}{p(p_{i,1})} \tag{6.5.1}$$

我们思考双向过程，即相机发出采样射线，并与场景中的某个点 $p_{i,1}$ 相交。

情况一：如果 $p_{i,1}$ 是光源点，那么6.5.3就可以写为：

$$W_e(p_{i,0} \to p_{i,1}) \frac{1}{p(p_{i,0})} \frac{L(p_{i,1} \to p_{i,0})}{p(\omega_{p_{i,0} \to p_{i,1}})} \tag{6.5.2}$$

顺便提一句，对于光源来说，发光值不需要几何项 G（比如对于漫散射面光源来说，相同距离下，从各个方向看向面光源，单位立体角接收到的辐射是相同的）。

情况二：如果 $p_{i,1}$ 不是光源点，那么6.5.3就可以写为：

$$W_e(p_{i,0} \to p_{i,1}) \frac{G(p_{i,0} \leftrightarrow p_{i,1})}{p(p_{i,0})} \frac{L(p_{i,1} \to p_{i,0})}{p(p_{i,1})} \tag{6.5.3}$$

以上是关于表面积采样的过程。

现在转换一下思路，假设相机已经发出了光线，镜头点在 $p_{i,0}$，击中了场景中的 $p_{i,1}$ 我们想要估计（假设相机是投影相机而非真实感相机，则 W_e 的值为 1）：

$$W_e(p_{i,0} \to p_{i,1}) L(p_{i,1} \to p_{i,0}) = L(p_{i,1} \to p_{i,0}) \tag{6.5.4}$$

假设此时 $p_{i,1}$ 并不是发光表面的点，则 $L(p_{i,1} \to p_{i,0})$ 由两部分构成，一是直接光照到该表面然后反射到 $\omega_{p_{i,0}-p_{i,1}}$ 方向的光，二是间接光照射到该表面然后反射到 $\omega_{p_{i,0}-p_{i,1}}$ 的光。

光子映射方法估计的是间接光照，前面所有的过程本质上都是描述怎么估计间接光照的内容。

6.6 小结

本文开始写于 11 月 9 日，结束于 11 月 24 日。中间一直在修改论文和为各种生活上的事情所累，11 月是一个很奇特的月份，这一月中发生了太多事情。

关于光子映射，其实还有不少基础理论没有介绍，我打算将光子映射相关的几篇论文串联一下，使这个知识体系更加完善。即使读者读完本文还有不少疑惑，甚至没有完全读懂也没有关系，我们只需要明确一点——光子映射作为一种双向方法，它的意义在于计算间接照明（更进一步，尤其是为了计算间接照明中的焦散效果）这就够了，目前绝大部分优化技术都是在研究对间接照明的优化，因为直接光照求解非常简单，误差也非常小。

参考文献

[1] Pharr M, Jakob W, Humphreys G. Physically based rendering: From theory to implementation[M]. Morgan Kaufmann, 2016.

[2] Veach E . Robust Monte Carlo Methods for Light Transport Simulation[J]. Ph.d.thesis Stanford University Department of Computer Science, 1998.