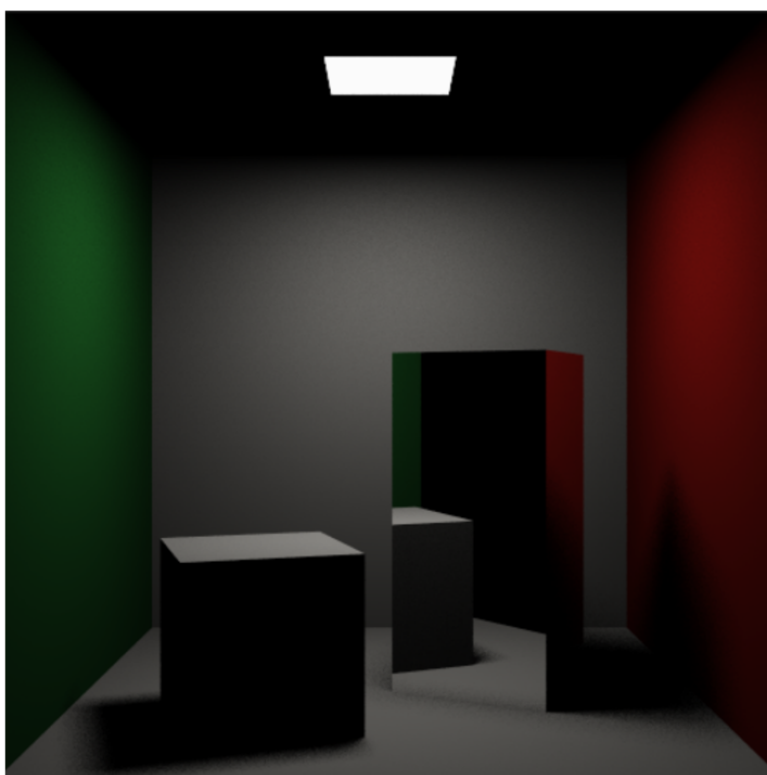
PBRT 代码实战-Whitted 光线追踪引擎

Dezeming Family

2021 年 1 月 19 日

因为本书是电子书，所以会不断进行更新和再版（更新频率会很高）。如果您从其他地方得到了这本书，最好从官方网站：https://dezeming.top/ 下载新的版本（免费下载）。

本书目标：根据 PBRT 中最简单的 Whitted 光线追踪器来学习 PBRT 的渲染流程，把渲染流程走通（主要目标）。然后在自己的引擎上按照 PBRT 的方式渲染出封面图效果（次要目标，选做，我们下一本书就完全开始移植和实现 PBRT 的引擎了）。



源码见网址 [https://github.com/feimos32/PBRT3-DezemingFamily]。

前言

本小书的是该系列的第二本，第一本是《PBRT 文件加载和设定》，已经完整地剖析了 PBRT 如何从文件加载和创建场景以及启动渲染，但是怎么在自己的系统上实现与 PBRT 同样功能的光线追踪效果是本书的目标。我们本文争取在实现了《光追三部曲》[2][3][4] 的基础上，扩展成 PBRT 风格光线追踪引擎。但请读者放心，本文不会是长篇大论，而是很容易就能阅读而且操作的。我们也只是把最简单的 Whitted 方法实现，不会有太多的新理论加入，Whitted 光线追踪器的实现与 [4] 中的“直接对光采样”一节的知识几乎完全一样，但是为了更好地研究 PBRT 里面的技术，我们需要通过 Whitted 弄清 PBRT 渲染的流程。

本书中最重要的部分是大家要对照着源码，把 Whitted 光线追踪的过程看几遍（虽然简单来说就是 Render 函数调用 Li 函数），包括求交、计算 BSDF，计算光照。最起码要有个模糊的印象。虽然我们下一本书就完全开始移植和实现 PBRT 的引擎，但如果能在自己的引擎上按照 PBRT 的方式走一遍，也是很好的选择。

本书的第二章仅作为了解，这一章不看也可以。该章属于知识回顾部分，仅仅作为蒙特卡洛方法的回顾和介绍，参考 [1] 和 [4] 以及 [5] 中的蒙特卡洛方法可以获得更详细的解释。Metropolis 采样和俄罗斯轮盘等会在 MLT 算法和路径追踪中应用到，因此我会在后续系列书中详细讲解。

本套系列书一共分为两个部分：

第一部分为 PBRT 基本知识《基础理论与代码实战》，一共 15 本书，售价 40 元，其中每本的售价都不同，本书的售价是 4 元（电子版）。我们不直接收取任何费用，如果其中某本书对大家学习有帮助，可以往我们的支付宝账户（17853140351，备注：PBRT）进行支持，您的赞助将是我们 Dezeming Family 继续创作各种图形学、机器学习、以及数学原理小册子的动力！

第二部分为专业理论知识部分《专业知识理论与代码实战》，一共 12 本书，暂定售价 120 元，每本价格也各不相同。我于 2022 年 7 月，开始着手写这个系列，但在此之前，我会将第一部分的每本书都配备源码，然后再开始第二个部分的写作。

目录

一 基本介绍 1

二 知识回顾 2
2 1 反函数方法 2
2 1.1 可计算反函数 2
2 1.2 分段常数函数 3
2 2 拒绝法 6
2 3 Metropolis 采样 7
2 4 多种概率分布之间的转换 7
2 5 俄罗斯轮盘 7
2 6 splitting 算法 8
2 7 分层抽样 8

三 PBRT 系统里光线追踪的流程 10

四 光线追踪使用的材料 11
4 1 光线追踪器场景 11
4 2 matte 材料 14
4 3 matte 反射 14
4 4 朗伯反射 BSDF 15

五 光线追踪中的采样 17
5 1 采样 BSDF 17
5 2 采样灯光 17

六 Whitted 光线追踪积分器 19
6 1 积分器概览 19
6 2 Whitted 积分器概览 19
6 3 Li() 函数 20
6 4 Li() 的反射和折射 21

七 自己编程来实现 Whitted 积分器 22
7 1 基本代码工程 22
7 2 架构设计 22
7 3 仅有朗伯反射物体的场景 23
7 4 有镜面物体的场景 24
7 5 关于采样器 25

八 本书结语 26

参考文献 27

一　基本介绍

首先，我们一开始使用的随机数不是低差异序列，而是用时钟 rand() 函数产生随机数：

```
#include "stdlib.h"
inline double getClockRandom() {
    return rand() / (RAND_MAX + 1.0);
}
```

渲染结果与使用低差异序列的准蒙特卡洛方法肯定有区别，我们在以后的系列书中再讲解和移植低差异随机数类。

本书是基于使用 [2][3][4] 进行扩展的，所以本质上说，只需要学会这三本小书，再学习这个系列就够了。虽然我们并不需要什么 BRDF，物理渲染之类的知识，我们只实现最简单的 matte 材料的渲染。但为了以后好叙述，先简单介绍一下什么是 BRDF 和 BSDF（这里不给出 radiance 和 irradiance 之类的关于辐射度的介绍，怕劝退读者）。

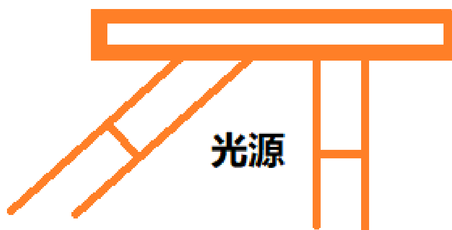
一般来说，凡是关于对光的入射和反射的能量进行建模的公式都可以称为 BRDF，即 bidirectional reflectance distribution function，双向反射分布函数。BRDF 是一个函数，一般表示为 $f_{BRDF}(\omega_i, \omega_r, n)$，其中 ω_i 表示入射光方向，ω_r 表示为光反射方向，n 表示为表面法向量，f 可以理解为从 ω_i 方向入射过来的光反射到 ω_r 方向的比重，因此可知如果光不被吸收，$\int_{\Omega_r} f \ d\omega_r = 1$，即反射方向的比重在半球上的积分为 1。

BSDF 可以认为是 BRDF 的拓展，因为散射可以包含内散射和外散射，比如在参与介质中的散射（不过在参与介质中的散射在 PBRT 的术语中一般被称为 BTDF），在 PBRT 中的散射都是用 BSDF 表示的。在 PBRT 系统中还有描述次表面散射的 BSSRDF 等，不过并不在我们这里要考虑的范围。我们的目标只是研究实现 PBRT 的 Whitted 光线追踪器，而不是一上来就把所有的材料纹理都学一遍，否则难度太大，容易劝退读者。

最后谈一下渲染方程：

$$L(\omega_o) = \int_{\Omega_i} f_{BRDF} L(\omega_i) cos(n, \omega_i) d\omega_i \tag{一.1}$$

即表面发射到某方向 ω_o 的光能表示为从整个半球面所有入射方向反射到 ω_o 方向的和。至于为什么要乘以一个 cos 值，是因为入射到表面的方向不同，则表面单位面积接受到的光能量大小也是不同的，如下图，同样强度的光，对于左边斜着方向发射到矩形的表面的光源和右边垂直发射到表面的光源，表面的单位面积接收到的光能量是不一样的。



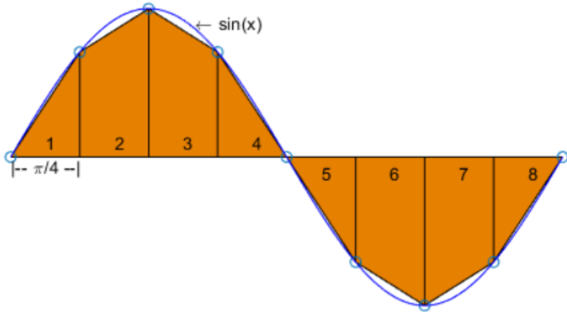
二　知识回顾

作为知识回顾，本节的理论知识大多摘抄整理自 PBRT[1]，而程序代码均为自己编写而非参考 [1] 的源码。因为 [2][3][4][5] 中已经有很好的入门介绍了，本节不再重点介绍这些基本细节，而是做一个简单的知识总结和回顾。

当读者对 [4] 中的蒙特卡洛方法了解得比较透彻时，本章可以忽略。但是为了完整性，这里还是做一些简单的介绍。里面的代码仅供参考，都是些简短的代码（大部分抽样方法我们在本书中是用不到的）。Metropolis 采样和俄罗斯轮盘等我们暂时还用不到，只是简单提一下，等以后的系列书介绍 PBRT 路径追踪和 MLT 的时候我们会详细介绍这些方法。

2 1　反函数方法

渲染方程是需要积分的，积分的方法有很多，低维空间可以用梯形法（如下）或者高斯求积公式法进行积分。



随机方法一般分为两类：拉斯维加斯方法（Las Vegas）以及蒙特卡罗方法。拉斯维加斯算法用于寻找精确解；而在解渲染方程时，我们主要采用蒙特卡洛方法。蒙特卡洛方法在高维空间中它的积分收敛速度与维度无关，仅与样本数量有关。

我们一共设计两种思路来产生符合某种 PDF 的样本，一种是已知函数，求反函数再生成符合该分布的样本，另一种是分段常数函数，生成符合该概率密度分布的样本。

2 1.1　可计算反函数

设 PDF 为 cx^2，区间在 0-2 范围内：

$$\int_0^2 cx^2 dx = 1 \tag{二.1}$$

解得：

$$c = \frac{3}{8} \tag{二.2}$$

计算 CDF 为

$$\frac{x^3}{8} \tag{二.3}$$

因此反函数为：

$$y = 2\sqrt[3]{x} \tag{二.4}$$

编程测试一下：

```
//待积函数
float func_test(float x) {
    return (1*x*x + 2 * x + sqrtf(x));
}
//random():产生0-1之间的随机数
inline float computable_a_pdf(float x) {
    return 3 * x*x / 8;
}
inline float computable_a_cal() {
    float sum = 0.0f;
    const int N = 1000000;
    for (int i = 0; i < N; i++) {
        float x = pow(8 * random(), 1. / 3.);
        if (x == 0) {
        }
        else {
            sum += func_test(x) / computable_a_pdf(x);
        }
    }
    return sum / (float)N;
}

//输出:
/*8.54673
8.54707
8.54566
8.5477
8.55197
8.54672*/
```

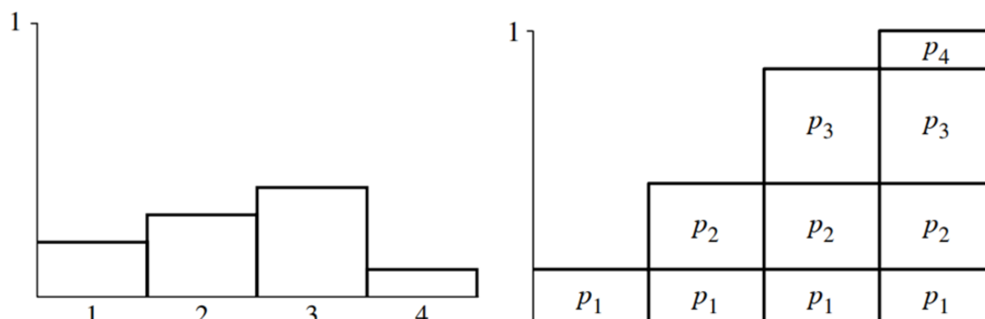
读者可以自己验证一下待积函数积分后的值，可以用科学计算软件计算。

2 1.2 分段常数函数

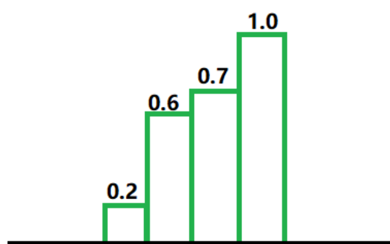
本节的代码能否看懂随缘 比如我们假设一共有 4 个可能的样本输出，每个概率分别为 p_i，即：

$$\sum_{i=1}^{4} p_i = 1 \tag{二.5}$$

概率密度函数（PDF）和累计概率密度函数（CDF）表示如下：



我们设一共有 4 段，CDF 我们设为实际数值：



设置每段区间为 0.5，分析一下：生成 0-0.5 之间的概率为 0.2，生成 0.5-1 之间的概率为 0.6-0.2=0.4，生成 1-1.5 之间的概率为 0.1，以此类推。假如我们生成的 0-1 随机数为 0.55，则要对应到图上的第二段，即生成的变量 x 为:

$$0.5 + \frac{0.55 - 0.2}{0.6 - 0.2} * 0.5 \tag{二.6}$$

注意这时候计算 PDF 不是两个 CDF 的差值，而要包含每段的长度。即用差值除以每段长度：

```
//分段常数
float *piecewise_c;
//实现过程中我们设置8段。注意 piece*pieceWidth 要等于积分的区间，这里是2，即在0到2上进行积分。
const int piece = 8;
const float pieceWidth = 0.25;
float genPiecewise(float* arr, float data, float pieceWidth, int piece, float &pdf) {
    float scale = pieceWidth;
    if (data < arr[0]) {
        pdf = arr[0] / scale;//注意要按照概率密度来计算，所以要除以每段的大小
        return (data) / arr[0] * pieceWidth;
    }
    else if (data > arr[piece - 2]) {
        pdf = (arr[piece - 1] - arr[piece - 2]) / scale;
        return ((piece - 1) + (data - arr[piece - 2]) / (arr[piece - 1] - arr[piece - 2])) * pieceWidth;
    }
    else {
        int index = binary_search(piecewise_c, data, 0, piece-1);
        pdf = (arr[index + 1] - arr[index]) / scale;
        return (index + 1 + (data - arr[index]) / (arr[index + 1] - arr[index])) * pieceWidth;
    }
}
inline float piecewise_a_cal() {
    //注意这里的 piece*pieceWidth 要等于积分空间的值，这里是2
    piecewise_c = (float*)malloc(piece * sizeof(float));
    piecewise_c[0] = 0.2; piecewise_c[1] = 0.3; piecewise_c[2] = 0.5; piecewise_c[3] = 0.6;
```

```
        piecewise_c[4] = 0.7; piecewise_c[5] = 0.75; piecewise_c[6] = 0.9;
            piecewise_c[7] = 1.0;
        float sum = 0.0f;
        const int N = 1000000;
        for (int i = 0; i < N; i++) {
            float pdf;
            float aa = genPiecewise(piecewise_c, random(), pieceWidth, piece
                , pdf);
            sum += func_test(aa) / pdf;
        }
        return sum / (float)N;
    }
```

上面代码中的 binary_search 就是用来找到当前段的前一个位置的索引。例如随机数为 0.85，那么就返回 2，因为 0.85 大于等于 piecewise_c[2]，但是又小于 piecewise_c[3]。因此我们可以设计出二进制搜索代码（注意我们的程序要保证 CDF 每段都不一样，而且后一个要比前一个大）：

```
    int binary_search(float* arr, float data, int low, int high) {
        int index;
        while (low <= high)
        {
            int middle = (low + high) / 2;
            if (data >= arr[middle]) {
                if (data >= arr[middle + 1]) {
                    low = middle;
                }
                else {
                    index = middle;
                    return index;
                }
            }
            else if (data < arr[middle]) {
                if (data >= arr[middle - 1]) {
                    index = middle - 1;
                    return index;
                }
                else {
                    high = middle-1;
                }
            }
        }
    }
```

我们把两个测试的结果相互对比，每组两个数，第一个是用计算反函数生成重要性样本后的计算结果，第二个是用分段常数 CDF 生成的重要性样本后的计算结果：

```
    8.54673
    8.55974

```

```
8.54062
8.55665

8.54963
8.54808

8.54567
8.53962

8.54943
8.54913

8.54699
8.54777
```

2 2 拒绝法

设概率密度函数为 $f(x)$。首先生成一个均匀分布：

$$X \sim Unit(x_{min}, x_{max}) \tag{二.7}$$

然后独立生成另一个均匀分布：

$$Y \sim Unit(y_{min}, y_{max}) \tag{二.8}$$

如果 $Y \leq f(X)$，则返回 X，否则回到第 1 步。

需要注意的是，这里 Y 的范围一定要大于 PDF 的范围。

```
inline float ar_pdf(float x) {
    return 3 * x*x / 8;
}
inline float accept_refuse_a_cal() {
    float sum = 0.0f;
    const int N = 1000000;
    for (int i = 0; i < N; i++) {
        while (1) {
            float r1 = 2.0f * random();
            float r2 = 3.5f * random();
            if (ar_pdf(r1) > r2) {
                if (r1 > 0) { //防止除以概率0
                    sum += func_test(r1) / ar_pdf(r1);
                }
                break;
            }
        }
    }
    return sum / (float)N;
}
```

计算的结果与上面两种方法基本一致。

2 3 Metropolis 采样

Metropolis 采样方法是为了产生于非负函数 f 同样分布的概率密度的算法。它不需要知道 f 的具体表示方程，也不需要计算反函数。但是因为样本在生成的时候会相互关联，所以不要指望少量样本就能得到很好的分布。所以我们需要采样大量样本才行。同时，类似分层抽样这种方差衰减方案在 Metropolis 算法中难以实现（毕竟 Metropolis 算法就是为了生成不均匀的样本）。

因为当前我们的首要任务是实现光线追踪引擎，而不是 Metropolis 光传输算法，因此这种技术暂时不再介绍。我会在后续系列写一个关于 MLT 实现的内容。

2 4 多种概率分布之间的转换

如何从一种分布的变量转为另一种分布。设这种映射关系为 y，即 $Y_i = y(X_i)$。因为两种分布在 CDF 上应该是一一对应的，否则就会意义不明（从直觉上解释，就是这两个变量的分布需要满足这个关系，否则这种转换就毫无意义）。这种一一对应的关系，可以表述为：

$$P\{Y \leq y(x)\} = P\{X \leq x\} \tag{二.9}$$

然后转化为：

$$\begin{aligned} P_y(y) &= P_y(y(x)) = P_x(x) \\ p_y(y)\frac{dy}{dx} &= p_x(x) \\ p_y(y) &= (\frac{dy}{dx})^{-1}p_x(x) \end{aligned} \tag{二.10}$$

加入 $p(x)$ 在 [0-1] 上的概率密度函数为 $p(x) = 2x$，同时 $Y = sin(X)$，则 Y 的 PDF 就可以计算为：

$$p_y(y) = \frac{p_x(x)}{|cos(x)|} = \frac{2x}{cos(x)} = \frac{2arcsin(y)}{\sqrt{1-y^2}} \tag{二.11}$$

如果我们已有两个 CDF，要从一个分布计算另一个分布，则仅仅需要保证 $P_y(y) = P_x(x)$，那么：

$$y(x) = P_y^{-1}(P_x(x)) \tag{二.12}$$

如果 X 是均匀的 [0-1] 分布，那么 $P_x(x) = x$，因此 $y(x) = P_y^{-1}(x)$ 就是我们之前的反函数法。

2 5 俄罗斯轮盘

一个估计器的效率可以定为：

$$\epsilon[F] = \frac{1}{V[F]T[F]} \tag{二.13}$$

俄罗斯轮盘赌解决的问题是，样本的评估成本很高，但对最终结果的贡献很小。而 splitting 技术则可以将更多的样本放在被积函数的重要维度上。

作为使用俄罗斯轮盘赌的一个例子，考虑估算直接照明积分的问题，渲染方程的积分给出了由于场景中光源的直接照明产生的辐射而在某一点产生的反射辐射，但是计算非常耗时。在有些方向因为被阻挡了，所以积分贡献就是 0，我们在采样时应该避免采样这些方向。有些采样方向积分值非常小，也应该跳过，比如反射方向接近水平面时，这个时候余弦值就非常接近 0。

光线追踪的采样 Ray 可以经过多次反射，但是反射多少次终止呢？这就可以采用俄罗斯轮盘来实现。为了应用俄罗斯轮盘赌，我们选择一些终止概率 q。以这个概率，被积函数有 q 的概率被终止，1-q 的概

率继续做积分追踪。这个值几乎可以用任何方式选择；例如，它可以基于对所选特定样本的被积函数值的估计，随着被积函数值变小而增加。对于概率 q 发生的情况（光线 Ray 终止），被积函数不再继续跟踪光线 Ray，而是使用一个常数值 c 来代替计算值（通常使用 $c = 0$）。当概率为 1−q 时，被积函数仍被计算，但被一个项 $1/(1-q)$ 加权，该项有效地解释了跳过的所有样本。我们有了新的估计器：

$$F^{'} = \begin{cases} \frac{F-qc}{1-q} & \xi > q \\ c & otherwise \end{cases} \tag{二.14}$$

该估计器的期望值为：

$$F[F^{'}] = (1-q)(\frac{F[F]-qc}{1-q}) + qc = E[F] \tag{二.15}$$

俄罗斯轮盘赌不会减少方差，事实上，除非 $c = F$，否则它总是会增加方差。然而，如果根据概率选择是否进行跟踪，从而跳过可能对最终结果作出微小贡献的样本，则确实可以提高效率。

其中一个陷阱是，选择不当的俄罗斯轮盘赌权重可以大大增加方差。想象一下，将俄罗斯轮盘赌应用到所有的相机光线，终止概率为.99：我们只跟踪 1% 的相机光线，每个光线的权重为 $1/.01 = 100$。从严格的数学意义上讲，得到的图像仍然是“正确”的，尽管从视觉上看结果会很糟糕：大部分是黑色像素，还有一些非常明亮的像素。一种称为效率优化俄罗斯轮盘赌的技术试图以最小化方差增加的方式设置俄罗斯轮盘赌权重。

2 6 splitting 算法

虽然俄罗斯轮盘赌减少了评估不重要样本所花的精力，但为了提高效率，splitting 算法会增加样本的数量。再次考虑仅由直接照明计算反射的问题。忽略像素滤波，该问题可以被写为像素 a 区域和每个 (x, y) 像素位置的表面上的可见点处的方向 S^2 的球体上的二重积分：

$$\int_A \int_{S^2} L_d(x, y, \omega) dxdyd\omega \tag{二.16}$$

其中 L_d 是当前像素 (x, y) 射到的某个表面在方向 ω 上接受到的光照量。如果场景中有许多光源，或者如果有区域光投射软阴影，则可能需要数十或数百个样本来计算具有可接受方差水平的图像。不幸的是，每个样本都需要在场景中跟踪两条光线：一条是从成像面上的位置 (x, y) 计算第一个可见表面，另一条是沿着 ω 到光源的阴影光线。

这种方法的问题是，如果取 $N = 100$ 个样本来估计这个积分，那么将跟踪 200 条光线：100 条相机光线和 100 条阴影光线。然而，100 个相机光线可能比良好的像素抗锯齿所需的光线多得多，因此对最终结果中的方差减少的贡献相对较小。splitting 算法解决了这个问题，通过形式化的方法在集成的某些维度中为其他维度中的每个样本抽取多个样本。

通过 splitting 算法，可以用 N 个图像样本和 M 个光样本来编写该积分的估计器，在像素 (x, y) 采样到的每个样本的光贡献量为：

$$\frac{1}{M} \sum_{i=1}^{M} \frac{L_d(x, y, \omega_i)}{p(\omega_i)} \tag{二.17}$$

因此，我们可以只采集 5 个图像样本，但每个图像样本采集 20 个光样本，总共跟踪 105 条光线，而不是 200 条。采集的 100 个区域光样本用来计算高质量的软阴影。

2 7 分层抽样

分层抽样，将一个像素格分成 $k * k$ 个网格，每个网格抽样一个，比直接抽样 $k * k$ 个随机数要好很多。

分层抽样永远不会增加方差，而是总是在减少方差，除非只有当函数 f 在每个层 i 上具有相同的平均值时，才不会减少方差。事实上，为了使分层抽样发挥最佳效果，我们希望最好使每个层具有尽可能不相等的平均值。这就解释了为什么如果一个人对函数 f 一无所知，那么紧凑的抽样层是可取的。如果层较宽，则每个层内的平均值将更接近整个积分域的平均值。

分层抽样的主要缺点是它与标准的数值求积法有着相同的“维数灾难”。在 D 维中完全分层，每个维中有 S 层则需要 S^D 个样品，对高维积分非常不友好。

三 PBRT 系统里光线追踪的流程

本章是我在校准勘误时补充的内容。

光线追踪需要什么？需要对每个像素产生一个光线 Ray，需要 Ray 与场景里的物体计算求交，然后计算交点被光照亮后的光亮度，返回给当前像素，这样，一帧就渲染完了。

我们在上一本书《文件加载与设定》已经知道了 pbrtWorldEnd 函数里调用积分器的 Render 函数启动渲染的，Render 函数的输入是包含了光源和物体的场景类 Scene 对象。物体在场景里通过 Aggregate 类构造成了加速结构，用于快速计算光线求交。

Render 函数的作用就是为每个像素生成采样光线 Ray，只不过 PBRT 复杂一点，它将成像区域分为了多个方形小块，每个小块放到一个线程里去渲染（不过我们不必在意）。之后会 Render 函数会调用 Li 函数，该函数是真正用于渲染的算法并得到最后结果（除了图像后处理）的函数，Li 函数传入参数包括相机生成的光线 Ray 和场景 Scene 对象。

Li 函数会首先计算与场景是否相交，如果有交点，就根据材质计算反射属性。因此，我们下一章首先介绍材质。我们只使用出现在 [2][3][4] 中的最简单的漫反射材质：lambertian 漫反射材质。之后我们会介绍 [4] 中出现的两种采样方法，即对光采样和对物体采样。然后介绍 Whitted 光线追踪器的 Li 函数的实现。

下面的章节中使用到的代码大家一定要在 PBRT 源码中找到，尽量做到仔细理解（关于采样和 Pdf 实在不理解也没关系）。虽然我在系列 8《反射与材质初步了解》才开始真正详细地介绍 lambertian 材质与概率计算，但其实 [4] 中已经介绍过了，大家应该可以理解。

看完源码中 Whitted 光线追踪器之后，有兴趣的同学可以学习后面的一章，动手将 PBRT 风格的光追器实现在基于 [2][3][4] 中构建的系统上。在本书系列 9《代码实战-灯光基础与完整的光线追踪器》中我们才会将 PBRT 的 Whitted 光线追踪器真正移植到我们自己的系统上。

四　光线追踪使用的材料

在写这一章的过程中，我会假设自己对 PBRT 中除了《文件加载与设定》中关于启动流程和文件加载以外的内容一无所知，我们开始研究 PBRT 的渲染细节，构建出一个光线追踪渲染器。

4 1　光线追踪器场景

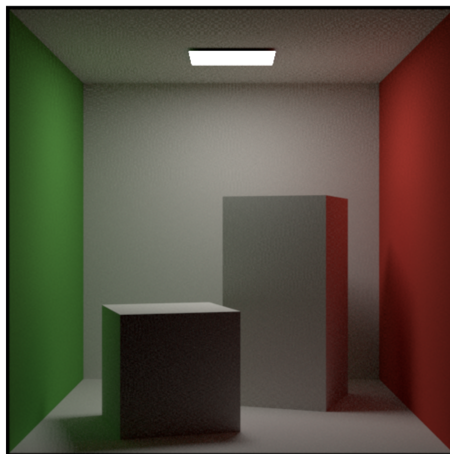
来个最简单的康奈尔盒场景吧，场景修改自 [5]。这个场景有个很不好的地方，就是里面的两个盒子都是变换后的顶点坐标，而不是原始顶点 + 变换矩阵的表示形式，看起来很乱，但我比较懒，而且人生短暂，我没有时间和精力去修改它，各位凑合着看吧。

```
1  Integrator "path" "integer maxdepth" [ 5 ]
2  LookAt 0 1 5   0 1 0   0 1 0
3  Sampler "sobol" "integer pixelsamples" [ 64 ]
4  PixelFilter "triangle" "float xwidth" [ 1.000000 ] "float ywidth" [
      1.000000 ]
5  Film "image" "integer xresolution" [ 512 ] "integer yresolution" [ 512 ]
       "string filename" [ "cornellBox.png" ]
6  Camera "perspective" "float fov" [ 28.500000 ]
7  WorldBegin
8
9      MakeNamedMaterial "LeftWall" "string type" [ "matte" ] "rgb Kd" [
          0.630000 0.065000 0.050000 ]
10     MakeNamedMaterial "RightWall" "string type" [ "matte" ] "rgb Kd" [
          0.140000 0.450000 0.091000 ]
11     MakeNamedMaterial "Floor" "string type" [ "matte" ] "rgb Kd" [
          0.725000 0.710000 0.680000 ]
12     MakeNamedMaterial "Ceiling" "string type" [ "matte" ] "rgb Kd" [
          0.725000 0.710000 0.680000 ]
13     MakeNamedMaterial "BackWall" "string type" [ "matte" ] "rgb Kd" [
          0.725000 0.710000 0.680000 ]
14     MakeNamedMaterial "ShortBox" "string type" [ "matte" ] "rgb Kd" [
          0.725000 0.710000 0.680000 ]
15     MakeNamedMaterial "TallBox" "string type" [ "matte" ] "rgb Kd" [
          0.725000 0.710000 0.680000 ]
16     MakeNamedMaterial "Light" "string type" [ "matte" ] "rgb Kd" [
          0.000000 0.000000 0.000000 ]
17
18     NamedMaterial "Floor"
19     Shape "trianglemesh" "integer indices" [ 0 1 2 0 2 3 ] "point P" [
          -1 0 -1 -1 0 1 1 0 1 1 0 -1 ] "normal N" [ 0 1 0 0 1 0 0 1 0 0 1
          0 ] "float uv" [ 0 0 1 0 1 1 0 1 ]
20     NamedMaterial "Ceiling"
21     Shape "trianglemesh" "integer indices" [ 0 1 2 0 2 3 ] "point P" [ 1
        2 1 -1 2 1 -1 2 -1 1 2 -1 ] "normal N" [ 0 -1 0 0 -1 0 0 -1 0 0
          -1 0 ] "float uv" [ 0 0 1 0 1 1 0 1 ]
22     NamedMaterial "BackWall"
23     Shape "trianglemesh" "integer indices" [ 0 1 2 0 2 3 ] "point P" [
          -1 0 -1 -1 2 -1 1 2 -1 1 0 -1 ] "normal N" [ 0 0 -1 0 0 -1 0 0 -1
```

```
      0 0 -1 ] "float uv" [ 0 0 1 0 1 1 0 1 ]
24    NamedMaterial "RightWall"
25    Shape "trianglemesh" "integer indices" [ 0 1 2 0 2 3 ] "point P" [ 1
      0 -1 1 2 -1 1 2 1 1 0 1 ] "normal N" [ 1 0 0 1 0 0 1 0 0 1 0 0 ]
      "float uv" [ 0 0 1 0 1 1 0 1 ]
26    NamedMaterial "LeftWall"
27    Shape "trianglemesh" "integer indices" [ 0 1 2 0 2 3 ] "point P" [
      -1 0 1 -1 2 1 -1 2 -1 -1 0 -1 ] "normal N" [ -1 0 0 -1 0 0 -1 0 0
      -1 0 0 ] "float uv" [ 0 0 1 0 1 1 0 1 ]
28
29    NamedMaterial "ShortBox"
30    Shape "trianglemesh" "integer indices" [ 0 2 1 0 3 2 4 6 5 4 7 6 8
      10 9 8 11 10 12 14 13 12 15 14 16 18 17 16 19 18 20 22 21 20 23
      22 ] "point P" [ -0.0460751 0.6 0.573007 -0.0460751 -2.98023e-008
      0.573007 0.124253 0 0.00310463 0.124253 0.6 0.00310463 0.533009
      0 0.746079 0.533009 0.6 0.746079 0.703337 0.6 0.176177 0.703337
      2.98023e-008 0.176177 0.533009 0.6 0.746079 -0.0460751 0.6
      0.573007 0.124253 0.6 0.00310463 0.703337 0.6 0.176177 0.703337
      2.98023e-008 0.176177 0.124253 0 0.00310463 -0.0460751 -2.98023e
      -008 0.573007 0.533009 0 0.746079 0.533009 0 0.746079 -0.0460751
      -2.98023e-008 0.573007 -0.0460751 0.6 0.573007 0.533009 0.6
      0.746079 0.703337 0.6 0.176177 0.124253 0.6 0.00310463 0.124253 0
      0.00310463 0.703337 2.98023e-008 0.176177 ] "normal N" [
      -0.958123 -4.18809e-008 -0.286357 -0.958123 -4.18809e-008
      -0.286357 -0.958123 -4.18809e-008 -0.286357 -0.958123 -4.18809e
      -008 -0.286357 0.958123 4.18809e-008 0.286357 0.958123 4.18809e
      -008 0.286357 0.958123 4.18809e-008 0.286357 0.958123 4.18809e
      -008 0.286357 -4.37114e-008 1 -1.91069e-015 -4.37114e-008 1
      -1.91069e-015 -4.37114e-008 1 -1.91069e-015 -4.37114e-008 1
      -1.91069e-015 4.37114e-008 -1 1.91069e-015 4.37114e-008 -1
      1.91069e-015 4.37114e-008 -1 1.91069e-015 4.37114e-008 -1 1.91069
      e-015 -0.286357 -1.25171e-008 0.958123 -0.286357 -1.25171e-008
      0.958123 -0.286357 -1.25171e-008 0.958123 -0.286357 -1.25171e-008
      0.958123 0.286357 1.25171e-008 -0.958123 0.286357 1.25171e-008
      -0.958123 0.286357 1.25171e-008 -0.958123 0.286357 1.25171e-008
      -0.958123 ] "float uv" [ 0 0 1 0 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1 0
      1 1 0 1 0 0 1 0 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1 0 1 1 0 1 ]
31    NamedMaterial "TallBox"
32    Shape "trianglemesh" "integer indices" [ 0 2 1 0 3 2 4 6 5 4 7 6 8
      10 9 8 11 10 12 14 13 12 15 14 16 18 17 16 19 18 20 22 21 20 23
      22 ] "point P" [ -0.720444 1.2 -0.473882 -0.720444 0 -0.473882
      -0.146892 0 -0.673479 -0.146892 1.2 -0.673479 -0.523986 0
      0.0906493 -0.523986 1.2 0.0906492 0.0495656 1.2 -0.108948
      0.0495656 0 -0.108948 -0.523986 1.2 0.0906492 -0.720444 1.2
      -0.473882 -0.146892 1.2 -0.673479 0.0495656 1.2 -0.108948
      0.0495656 0 -0.108948 -0.146892 0 -0.673479 -0.720444 0 -0.473882
      -0.523986 0 0.0906493 -0.523986 0 0.0906493 -0.720444 0
```

```
        -0.473882 -0.720444 1.2 -0.473882 -0.523986 1.2 0.0906492
        0.0495656 1.2 -0.108948 -0.146892 1.2 -0.673479 -0.146892 0
        -0.673479 0.0495656 0 -0.108948 ] "normal␣N" [ -0.328669 -4.1283e
        -008 -0.944445 -0.328669 -4.1283e-008 -0.944445 -0.328669 -4.1283
        e-008 -0.944445 -0.328669 -4.1283e-008 -0.944445 0.328669 4.1283e
        -008 0.944445 0.328669 4.1283e-008 0.944445 0.328669 4.1283e-008
        0.944445 0.328669 4.1283e-008 0.944445 3.82137e-015 1 -4.37114e
        -008 3.82137e-015 1 -4.37114e-008 3.82137e-015 1 -4.37114e-008
        3.82137e-015 1 -4.37114e-008 -3.82137e-015 -1 4.37114e-008
        -3.82137e-015 -1 4.37114e-008 -3.82137e-015 -1 4.37114e-008
        -3.82137e-015 -1 4.37114e-008 -0.944445 1.43666e-008 0.328669
        -0.944445 1.43666e-008 0.328669 -0.944445 1.43666e-008 0.328669
        -0.944445 1.43666e-008 0.328669 0.944445 -1.43666e-008 -0.328669
        0.944445 -1.43666e-008 -0.328669 0.944445 -1.43666e-008 -0.328669
         0.944445 -1.43666e-008 -0.328669 ] "float␣uv" [ 0 0 1 0 1 1 0 1
        0 0 1 0 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1 0 1 1 0 1 0
         0 1 0 1 1 0 1 ]
33
34      AttributeBegin
35          AreaLightSource "diffuse" "rgb␣L" [ 17.000000 17.000000
                17.000000 ]
36          NamedMaterial "Light"
37          Shape "trianglemesh" "integer␣indices" [ 0 1 2 0 2 3 ] "point␣P"
                [ -0.24 1.98 -0.22 0.23 1.98 -0.22 0.23 1.98 0.16 -0.24 1.98
                0.16 ] "normal␣N" [ -8.74228e-008 -1 1.86006e-007 -8.74228e
                -008 -1 1.86006e-007 -8.74228e-008 -1 1.86006e-007 -8.74228e
                -008 -1 1.86006e-007 ] "float␣uv" [ 0 0 1 0 1 1 0 1 ]
38      AttributeEnd
39
40  WorldEnd
```

渲染出来的场景为：



可以看到里面都是材质 matte，这是无高光的漫反射材质。我们这本书的工作，就是理解这个盒子究竟是怎么被渲染出来的，然后可以使用 PBRT 的方法把这个场景渲染出来。注意上面的渲染方法是路径追踪，所以存在间接光，而我们学习的是 Whitted 光线追踪，因此被物体挡住光源的部分是全黑的。

4 2 matte 材料

根据 PBRT[1] 的第 9 章描述，最简单最基本的材料就是 matte 了，因为是纯漫反射材料，所以这就作为我们首要研究的材料（也是除了完美的镜面反射以外，这本书里学习光线追踪渲染器唯一需要研究的材料）。

matte 是基本的漫反射材料，class MatteMaterial : public Material。构造函数用来初始化三个量：Kd，sigma，bumpMap。Kd 表示的是漫反射物体的颜色（漫反射颜色）。

创建 matte 材料的函数是 MakeMaterial()：

```
if (name == "matte")
    material = CreateMatteMaterial(mp);
```

在该函数中，为材料赋值的语句为：

```
std::shared_ptr<Texture<Float>> sigma = mp.GetFloatTexture("sigma", 0.f);
std::shared_ptr<Texture<Float>> bumpMap =
    mp.GetFloatTextureOrNull("bumpmap");
```

sigma 默认值为 0.0，由 PBRT 书中所述，当 sigma 为 0 时创建 Lambertian（朗伯）材料，当为其他值是创建 OrenNayar（粗糙材质）材料，粗糙材质是更精确的粗糙度材质，但因为我们的场景文件并没有设置，所以我们只研究渲染朗伯材质就好啦，毕竟我们的目标只是复现光线追踪器，而不是研究材料属性。

bump map 是凹凸贴图，如果提供了就使用，但是我们提供的场景没有凹凸贴图，因此我们不需要处理它。

综上所述，在我们的场景中，我们只创建了一堆朗伯反射物体，而且都只有漫反射颜色属性。

4 3 matte 反射

大家可以从 PBRT 书看到，所有的材质类都得实现一个 ComputeScatteringFunctions() 函数，该函数接受一个表示表面交点的类 SurfaceInteraction，然后计算出该交点处的反射特性：

```
virtual void ComputeScatteringFunctions(SurfaceInteraction *si,
                    MemoryArena &arena, TransportMode mode, bool
                        allowMultipleLobes) const = 0;
```

这个函数接受 4 个参数，我们这一节就来讲解一下这四个参数。

SurfaceInteraction 继承自 Interaction，表示采样光线与物体相交的属性，我们有两种主要物质，一种是表面类物质，另一种是参与介质（体渲染物体），因此 Interaction 的另一个派生类便是 MediumInteraction。SurfaceInteraction 类（interaction.h）包含了很多信息，例如当前击中的基元，当前击中点的法向量，当前击中位置的 BRDF。

memoryRena 用于为 BSDF 和 BSSRDF 分配内存，我们后面的书中会学到。如果不在意速度的话，使用 C++ 的 new 关键字就可以了，但是要记得释放内存。

TransportMode 参数表示曲面交点是沿着从相机开始的路径还是沿着从光源开始的路径找到的（在双向路径追踪中会很有用）；该细节对如何评估 BSDF 和 BSSRDF 有影响。不过在 matte 材质中，该参数没有起到任何作用，因此我们不必理会。

allowMultipleLobes 参数指示当 BxDF 可用时，材质是否应使用将多种散射类型聚合为单个 BxDF。不过在 matte 材质中，该参数没有起到任何作用，因此我们不必理会。

在 matte 材质的 ComputeScatteringFunctions 函数中，我们可以看到创建 bsdf 的语句：

```
    si->bsdf = ARENA_ALLOC(arena, BSDF)(*si); //为bsdf分配内存
    Spectrum r = Kd->Evaluate(*si).Clamp(); //把漫反射颜色值clamp在大于0的正
        数，赋值给r
    Float sig = Clamp(sigma->Evaluate(*si), 0, 90);
```

```
4     if (!r.IsBlack()) { //只要r的RGB分量有任何一个值大于0，便不是黑色
5         if (sig == 0)
6             si->bsdf->Add(ARENA_ALLOC(arena, LambertianReflection)(r));
7         else
8             si->bsdf->Add(ARENA_ALLOC(arena, OrenNayar)(r, sig));
9     }
```

该函数执行完后，表面交点属性 SurfaceInteraction 就携带了当前点的反射特性，在朗伯材料中就是朗伯 BSDF 函数。

4 4 朗伯反射 BSDF

为了比较方便叙述，这里把整个朗伯材料的代码都发一下：

```
1     class LambertianReflection : public BxDF {
2         public:
3           LambertianReflection(const Spectrum &R)
4               : BxDF(BxDFType(BSDF_REFLECTION | BSDF_DIFFUSE)), R(R) {}
5           Spectrum f(const Vector3f &wo, const Vector3f &wi) const;
6           Spectrum rho(const Vector3f &, int, const Point2f *) const {
                return R; }
7           Spectrum rho(int, const Point2f *, const Point2f *) const { return
                R; }
8           std::string ToString() const;
9         private:
10          const Spectrum R;
11      };
```

这里唯一的变量的 R，被初始化参数 r 赋值，即漫反射颜色 Kd。

ToString() 函数返回描述该朗伯材料的信息的字符串，用于调试。

当提供了方向信息 ω_o 时，rho（上述代码第 6 行）返回的是整个半球方向的反射光的比例（hemispherical-directional reflectance ）：

$$rho = \int_{hemispherical(n)} f_r(p, \omega_o, \omega_i)|cos\theta_i|d\omega_i \tag{四.1}$$

其中，$hemispherical(n)$ 表示的是以法向量 n 表示的半球区域。

当没有提供方向时，rho（上述代码第 7 行）返回的是反射和入射比：

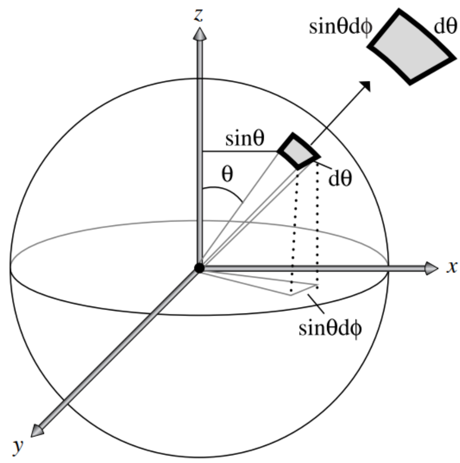
$$rho = \frac{1}{\pi}\int_{hemispherical(n)}\int_{hemispherical(n)} f_r(p, \omega_o, \omega_i)|cos\theta_o cos\theta_i|d\omega_o d\omega_i \tag{四.2}$$

上面这两个 rho 函数我们暂时都用不到，在 BSSDRF 材质时会需要，我们暂时不必理会。

f() 函数计算 BSDF。该值与光的入射方向 ω_i 和光的出射方向 ω_o 有关，但是对于朗伯物体，各个方向反射的光量都是一样的，因此下面的计算式中的 $f(\omega_i, \omega_o)$ 其实是一个常数：

$$\int\int f(\omega_i, \omega_o)cos(\theta)sin(\theta)d\theta d\phi = 1 \tag{四.3}$$

上式的 θ 和 ϕ 的表示如下，我们积分的区间是一个半球，因此 θ 积分区域是 $[0, 0.5\pi]$，而 ϕ 积分区域是 $[0, 2\pi]$：



计算得到：

$$f\int\int cos(\theta)sin(\theta)d\theta d\phi = f\int_0^{2\pi}\Big(\int_0^{\frac{\pi}{2}}\frac{1}{2}sin(2\theta)d\theta\Big)d\phi = 1$$
$$f = \frac{1}{\pi} \qquad (四.4)$$

因此，我们可以理解为，当辐射度为 L 的光 ω_i 方向照射到颜色为 Kd 的朗伯物体上时，反射到 ω_o 方向的光能量为：

$$\frac{L*Kd}{\pi} \qquad (四.5)$$

而 f() 函数的输入参数没有辐射度为 L，因此它只返回 BSDF 中的材质反照率部分，即：

```
1 Spectrum LambertianReflection::f(const Vector3f &wo, const Vector3f &wi)
      const {
2      return R * InvPi;
3 }
```

五　光线追踪中的采样

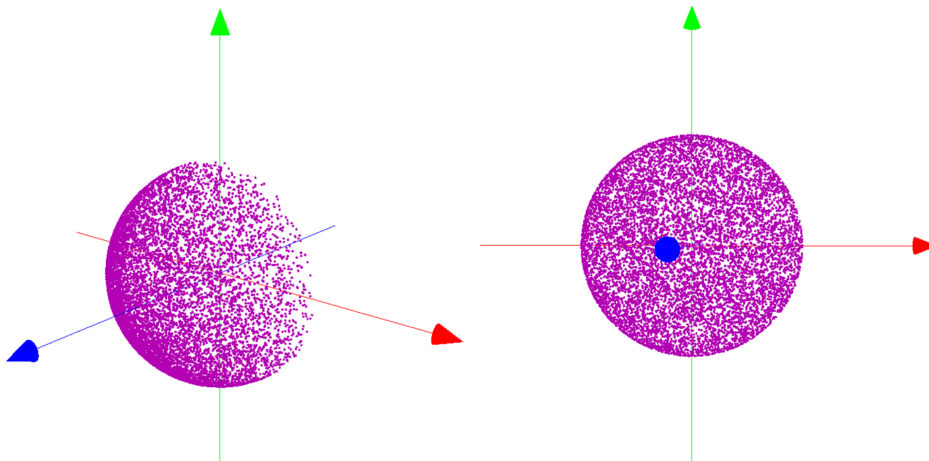
因为我们只有朗伯反射物体，光源也只有面光源，所以这一章比较容易。PBRT 的 Whitted 光线追踪技术和 [2][3][4] 就没有本质的区别，只是我们需要明白 PBRT 里面的结构，比如前一章描述的关于材料的设置。本章我们了解一下 PBRT 中的对 BSDF 采样和对灯光采样两种采样方式。

5 1　采样 BSDF

函数 BxDF::Sample_f()（reflection.h）实现了在给定出射方向（Ray 的反方向），按照 BSDF 的概率分布生成入射方向（光源射到物体的方向）以及其概率密度，朗伯反射物体的 Sample_f() 函数直接继承自 BxDF 父类：

```
1 Spectrum BxDF::Sample_f(const Vector3f &wo, Vector3f *wi, const Point2f &u,
2     Float *pdf, BxDFType *sampledType) const
3 {
4     // Cosine-sample the hemisphere, flipping the direction if necessary
5     *wi = CosineSampleHemisphere(u);
6     if (wo.z < 0) wi->z *= -1;
7     *pdf = Pdf(wo, *wi);
8     return f(wo, *wi);
9 }
```

输入参数中，u 表示一个二维随机数，我把 CosineSampleHemisphere 移植到了 OpenGL 项目里，生成的随机数点如下，左右两张图只是用了不同的视角：



可以看到与 z 轴夹角越小，生成的采样点越密集，至于 CosineSampleHemisphere 到底算法是如何实现的我们就暂时不管了，这里我们只知道它的作用就好（光追三部曲中的第三本 [4] 中有生成半球面随机方向的讲解）。

BxDF::Pdf() 计算得到的概率密度值是（注意 PBRT 采用了世界坐标到局部坐标的变换，因此用到了 AbsCosTheta，貌似有些难以理解，但其实就是计算的下面的公式）：

$$f_{BRDF} * cos(n, \omega_o) = \frac{cos(n, \omega_o)}{\pi} \tag{五.1}$$

5 2　采样灯光

直接采样光的方法在 [4] 中已经介绍的足够详细了，PBRT 上的方法和书中是完全一样的。

```
1 Spectrum DiffuseAreaLight::Sample_Li(const Interaction &ref, const Point2f &
    u,
2     Vector3f *wi, Float *pdf, VisibilityTester *vis) const {
```

```
3     ProfilePhase _(Prof::LightSample);
4     Interaction pShape = shape->Sample(ref, u, pdf);
5     pShape.mediumInterface = mediumInterface;
6     if (*pdf == 0 || (pShape.p - ref.p).LengthSquared() == 0) {
7         *pdf = 0;
8         return 0.f;
9     }
10    *wi = Normalize(pShape.p - ref.p);
11    *vis = VisibilityTester(ref, pShape);
12    return L(pShape, -*wi);
13 }
```

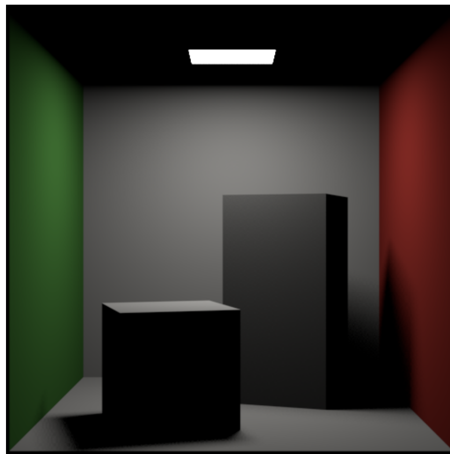
上面的函数中，ref 就是当前需要计算照明的物体的交点，shape 是光源所表示的形状，可能是球体，也可能是三角形。pShape 就是在光源上随便选一个采样点。shape->Sample 根据当前交点和光上的采样点，计算两者之间的 pdf，这个 pdf 计算方法和 [4] 中直接采样光一节的方法完全一样，这里不再赘述。wi 计算出光源的方向，而后 VisibilityTester 是测试可见性的，就是光源采样点到当前物体点之间有没有遮挡。

六　Whitted 光线追踪积分器

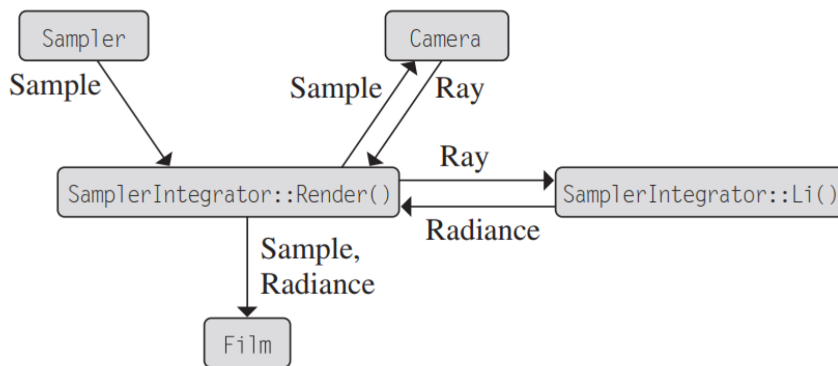
Whitted 积分器是在 PBRT 书的第一章讲述的积分器，也是整个 PBRT 渲染中最简单的积分器，但是它可以渲染出较好的镜面反射和折射物体。我们目前只有最简单的朗伯材料，所以我们先只参考朗伯材料来研究 Whitted 积分器的渲染。

6 1　积分器概览

积分器的基类是 Integrator，不过基类是一个抽象类，里面只有一个纯虚函数 Render()。SamplerIntegrator 继承自该类，SamplerIntegrator 也是抽象类，它实现了 Render() 函数，但多包含了一个纯虚函数 Li()，因此无法直接构建该类。我们可以看到，Whitted 光线追踪器继承自 SamplerIntegrator，并实现了 Li() 函数，因此我们可以说，SamplerIntegrator 类的 Render() 函数加上 whitted 积分器实现的 Li() 函数就构成了 Whitted 积分器类。使用 Whitted 光追渲染结果如下：



下图截取自 PBRT 书 [1]，在渲染器中，Camera 类产生采样 Ray，Li() 方法计算有多少光照量沿着该 Ray 到达成像平面，并把光照量（radiance）保存在 Film 内。



Li() 方法接受四个参数：

```
virtual Spectrum Li(const RayDifferential &ray, const Scene &scene,
        Sampler &sampler, MemoryArena &arena, int depth = 0) const;
```

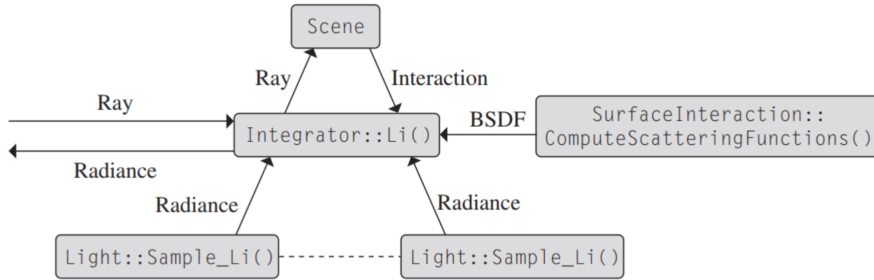
RayDifferential 是光线微分的采样 Ray（光线微分的作用是抗锯齿和反混淆），我们暂且当成一般的采样 Ray 就好了。MemoryArena 是在渲染中用来高效分配内存的类，我们暂时也不用在意。

在渲染完场景以后，PBRT 系统会把渲染结果进行整合和归并，这些流程在这我不打算介绍。我们完全可以直接处理每个像素点，而没有必要分成小块然后启动多线程来加速，否则学习一些优化方案在初期很容易占用我们过多的精力。

6 2　Whitted 积分器概览

Whitted 积分器非常简单，它只去考虑了直接光照以及镜面反射和折射，但是没有间接光照。下图表示了 Whitted 积分器的示意图，相机产生的 Ray 与场景计算交点，然后再计算交点位置的 BSDF，之后

根据 BSDF 来计算光照量。



Whitted 积分器提供了一个变量 WhittedIntegrator::maxDepth 来确定最大的递归次数（默认是 5），防止像类似两个镜子正对着放，可能光线就会来回反射，永不停止了。

为了简洁，我把 render() 函数最重要的几行代码写在了下面（注意这些代码我们其实可以完全不用理解）。在积分器的 render() 函数中，可以看到，首先相机产生 Ray，返回当前 Ray 的权重 rayWeight（见下面代码的第一行），该权重用于最后求光照量的加权平均值（权重主要是用在真实感相机的，在一般的相机完全可以理解为所有 Ray 权重相同）。之后计算该 Ray 的光照量（见下面代码的第二行）；然后检测 L 不是非法数据后，加入到 Film 的样本里（见下面代码第三行）。等所有 Film 样本都计算完以后，将样本融合（见下面代码第四行）

```
1  Float rayWeight = camera->GenerateRayDifferential(cameraSample, &ray);
2  L = Li(ray, scene, *tileSampler, arena);
3  filmTile->AddSample(cameraSample.pFilm, L, rayWeight);
4  camera->film->MergeFilmTile(std::move(filmTile));
```

我们目前要实现的话会非常简单（毕竟我们忽略了分块渲染和并行加速），里面唯一需要我们弄清楚的函数就是 Li(ray, scene, *tileSampler, arena)，这是我们渲染的关键。

6 3 Li() 函数

Whitted 积分器的 Li() 函数非常短，而且很好理解。

首先测试 Ray 与物体的相交，如果没有相交，就返回跟场景中的无限环境光源（如果有的话）相交的亮度值；没有环境光源的话就直接返回，这个时候返回的颜色是黑色：

```
1      if (!scene.Intersect(ray, &isect)) {
2          //除了环境光以外，别的light->Le()函数返回值都是Spectrum(0.f)
3          for (const auto &light : scene.lights) L += light->Le(ray);
4          return L;
5      }
```

如果有交点，就计算相交处的 BSDF 函数，需要为它分配内存，但因为是局部量，所以使用 MemoryArena 来分配（再不厌其烦地提一句，我们目前只需要知道 MemoryArena 可以高效地分配内存即可）。

顺便提一下（大家不了解也没有任何关系），如果交点计算 BSDF 没有计算出来，说明可能是在两个参与介质（比如烟雾，用于体渲染）的相接处，就使用 SpawnRay() 方法生成 Ray 去再次求交（见下面的代码）。

```
1  isect.ComputeScatteringFunctions(ray, arena);
2  if (!isect.bsdf)
3      return Li(isect.SpawnRay(ray.d), scene, sampler, arena, depth);
```

之后，如果我们的 Ray 相交处是一个光源，就计算其亮度值：

```
1      L += isect.Le(wo);
```

然后，我们对该表面计算每个光源的贡献：

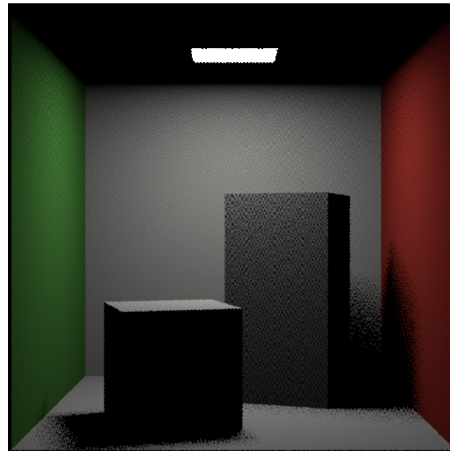
```
for (const auto &light : scene.lights) {
    Vector3f wi;
    Float pdf;
    VisibilityTester visibility;
    Spectrum Li =
        light->Sample_Li(isect, sampler.Get2D(), &wi, &pdf, &visibility)
            ;
    if (Li.IsBlack() || pdf == 0) continue;
    Spectrum f = isect.bsdf->f(wo, wi);
    if (!f.IsBlack() && visibility.Unoccluded(scene))
        L += f * Li * AbsDot(wi, n) / pdf;
}
```

light->Sample_Li() 函数之前已经介绍过了，这里会把所有光源都采样一遍。

我们打开场景文件，设置在场景文件中每个像素点只采样一次：

```
Sampler "sobol" "integer pixelsamples" [ 1 ]
```

渲染得到下图：



可以看到，即使每像素只采样一次，渲染的结果也挺不错的。

6 4 Li() 的反射和折射

Whitted 还可以计算反射和折射：

```
if (depth + 1 < maxDepth) {
    // Trace rays for specular reflection and refraction
    L += SpecularReflect(ray, isect, scene, sampler, arena, depth);
    L += SpecularTransmit(ray, isect, scene, sampler, arena, depth);
}
```

我们需要注意的是，在反射和折射函数中，都是递归地去计算光照量的，例如反射中的：

```
return f * Li(rd, scene, sampler, arena, depth + 1) * AbsDot(wi, ns) / pdf;
```

所以如果是镜面来回反射的情况，又会在 Li() 中继续调用 SpecularReflect()，反射和折射并不复杂，因此这里不再赘述。

七　自己编程来实现 Whitted 积分器

本章我们的额外目标是自己实现一个 Whitted 积分器。该引擎建立在 [2][3][4] 的基础上，但是本质来说只要有一个基础的光线追踪器就可以了。我们不使用三角面片模型来构建复杂的场景，只使用简单的球体和矩形即可。我会在下一本书中重新开始，实现 PBRT 的 Shape 和 Accelerator。

7 1　基本代码工程

我们的代码是使用 Qt 作为可视化界面的，但大家没有学习过 Qt 也没有任何关系，因为我们只需要知道几个接口即可。

[0 - OriginProject] 是我们的初始工程代码，我们的代码是用 CMake 生成的，在 CMakeList.txt 中，注意下面的 Qt 安装目录要修改为你的 Qt 安装目录：

```
set(QT_PATH "D:/DevTools/QT5/5.7/msvc2015_64" CACHE PATH "qt5 cmake dir")
```

这里不需要其他配置，我们只需要 Qt 这一个需要额外指定的库工具。

编译好以后，就是我们的显示界面，点击左边的按钮，就会弹出调试显示框，不断打印 Rendering。

该代码中，MainGUI/RenderThread 类中的 run() 函数就是运行渲染循环的程序。我们的渲染工作放在另一个线程中进行。在渲染中需要打印调试信息时，则调用 Qt 的信号函数：

```
emit PrintString("Rendering");
```

它会在主线程的调试框 DebugText 对象中打印输出信息。

渲染的结果会通过下面的信号输出显示：

```
emit PaintBuffer(p_framebuffer->getUCbuffer(), 800, 600, 4);
```

FrameBuffer 类是存储渲染结果图像的类，里面的函数功能都非常简单。

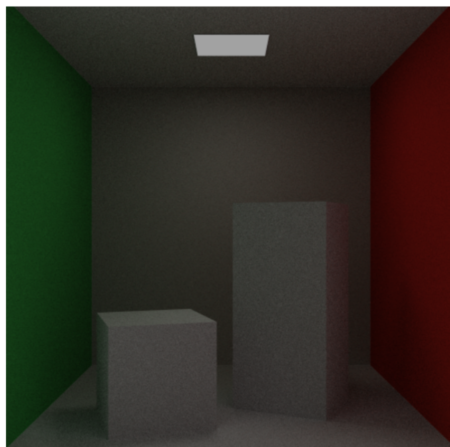
本文实现的内容见代码 [2 - PBRT2WhittedRayTracer]（注意没有以 1 开头的代码，因为第一本介绍场景加载的小书是没有代码的）。

7 2　架构设计

在 [2][3][4] 中构建的渲染器中，物体的空间变换非常简单，没有使用任何空间三维变换矩阵。如果现在就开始实现 Whitted 好像不是特别明智，不过鉴于 Whitted 积分器实在是简单，我们也暂时不追求一次就做得跟 PBRT 一样完备，所以那些矩阵变换什么的就先不考虑了，用简单的平移和旋转-反旋转操作（见 [3]）也是可以凑合着用的。

我把全部与光追有关的内容都放在了 RayTracer 目录下，其中，里面除了 RayTracer.h 以外，其他代码都是照搬自 [2][3][4]。RayTracer.h 定义了基本场景以及 Whitted 光追程序。

我们先实现书中的包围盒，构造一个类似的场景，见 cornell_box() 函数。使用 [4] 中最后一节做出的渲染器渲染出来的结果如下：



但是 [2][3][4] 的光线 Ray 在渲染时是在场景里不断反弹的，没有实现计算是否遮挡的函数。PBRT 在计算直接光照时，要计算光照是否被遮挡的话就直接发送一个光线过去，如果没有打到光上，就返回黑色。我们也可以写一个类似的函数。

简单回顾一下 [2][3][4] 中使用的一些类。hit_record 相当于 PBRT 中的 SurfaceInteraction 类，记录了交点信息，交点法向量材质等，scatter_record 类用来生成根据表面材质进行散射的信息，例如散射方向，交点面的颜色值（变量名为 albedo）等。在 [2][3][4] 中，所有的形状，包括 BVH 树节点，包围盒等都继承自 hitable 类，该类主要函数是 hit()，用来计算采样 Ray 是否与该形状或者 BVH 树节点的 AABB 包围盒相交。

下一节我们就讲述如何实现一个自己的 Whitted 光线追踪器。

7 3　仅有朗伯反射物体的场景

我们的采样光的函数很简单：

```
vec3 getLight(const ray&r, hitable *world) {
    hit_record hrec;
    scatter_record srec;
    if (world->hit(r, 0.001, 100000.0f, hrec)) {
        //计算发出的光。如果是非光源物体，emitted就是vec3(0.f)
        vec3 emitted = hrec.mat_ptr->emitted(r, hrec, hrec.u, hrec.v, hrec.p
            );
        return emitted;
    }
    else {
        return vec3(0, 0, 0);
    }
}
```

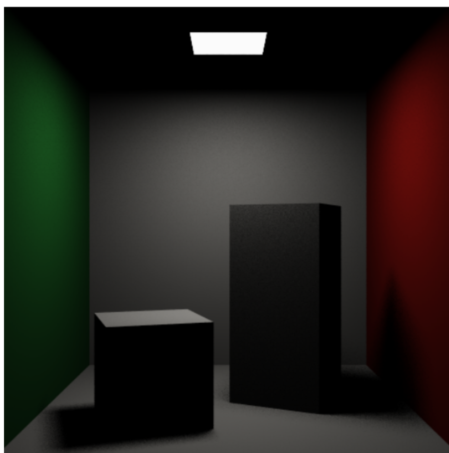
如果光源被物体挡住了，则 hit_record 记录的就是物体的信息，那么 hrec.mat_ptr->emitted 返回的就是黑色，因此该函数不但用来返回光源颜色，还包含了光源是否被遮挡的信息。

Whitted 光线追踪器的实现也不复杂：

```
vec3 WhittedRT(const ray&r, hitable *world, int depth) {
    vec3 finalColor(0.0, 0.0, 0.0);
    ray tempR = r;
    hit_record hrec;
    scatter_record srec;
    if (world->hit(tempR, 0.001, 100000.0f, hrec)) {
        vec3 emitted = hrec.mat_ptr->emitted(tempR, hrec, hrec.u, hrec.v,
            hrec.p);
        //击中了物体
        if (hrec.mat_ptr->scatter(tempR, hrec, srec)) {
            //计算对光采样的pdf
            hitable_pdf p(light_shape, hrec.p);
            ray scattered = ray(hrec.p, p.generate(), r.time());
            float pdf_val = p.value(scattered.direction());//对光重要性采样
                的pdf
            float mpdf = hrec.mat_ptr->scattering_pdf(r, hrec, scattered);//
                计算散射pdf
```

```
15              //计算光照强度
16              vec3 Li = getLight(scattered, world);
17              finalColor = srec.albedo * mpdf * Li / pdf_val;
18          }
19          //击中了光源
20          else {
21              finalColor = emitted;
22          }
23      }
24      return finalColor;
25  }
```

输入参数中，depth 表示递归深度，该参数虽然没有用到，但暂时保留。对光源进行重要性采样的方法可以参考 [4]，之前提到过，该实现与 PBRT 是完全一样的。渲染出的结果如下：



我们定义了 HDR2LDR 函数，用来将高动态范围的值缩小到 0 到 1 之间。

7 4 有镜面物体的场景

我们在 PBRT 中的场景并没有设置折射和反射物，是因为我不想花太多章节去介绍材质。但是因为 [2][3][4] 中有关于镜面物体和介质的实现，所以我们可以很简单就能实现一个包含了镜面反射和折射场景的 Whitted 渲染器。

```
1 //计算镜面反射光照
2 Vec3f calSpecularReflection(const Ray&r, hitable *world, int maxDepth);
3 //计算折射光照
4 Vec3f calSpecularRefraction(const Ray&r, hitable *world, int maxDepth);
```

但是因为 [2][3][4] 中并没有记录是反射还是折射，但是计算了反射或折射的方向，即 srec.specular_ray，同时 srec.is_specular 来判断当前材质是否是镜面物体。因此我们只使用一个函数来计算镜面反射或折射，我们修改一下 WhittedRT 函数：

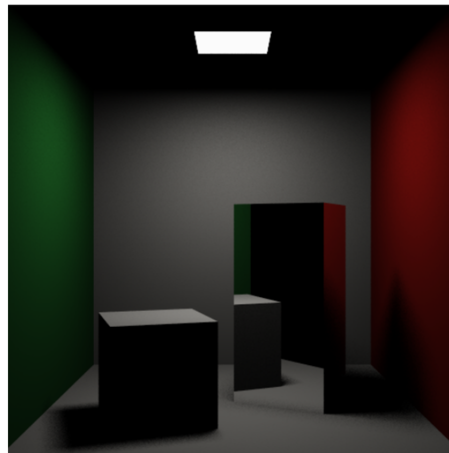
```
1 if (hrec.mat_ptr->scatter(tempR, hrec, srec)) {
2     if (srec.is_specular) {
3         finalColor = calSpecularLi(srec.specular_ray, world, depth + 1);
4     }
5     else {
6         //计算对光采样的pdf
7         hitable_pdf p(light_shape, hrec.p);
8         Ray scattered = Ray(hrec.p, p.generate(), r.time());
```

```
9           float pdf_val = p.value(scattered.direction());//对光重要性采样的pdf
10          float mpdf = hrec.mat_ptr->scattering_pdf(r, hrec, scattered);//计算散射pdf
11          Vec3f Li = getLight(scattered, world);
12          finalColor = srec.albedo * mpdf * Li / pdf_val;
13      }
14  }
```

calSpecularLi 函数就是计算镜面反射的光照的函数：

```
1  const int maxDepth = 5;
2  Vec3f calSpecularLi(const Ray&r, hitable *world, int depth) {
3      if (depth > maxDepth)
4          return Vec3f(0.0, 0.0, 0.0);
5      return WhittedRT(r,world,depth);
6  }
```

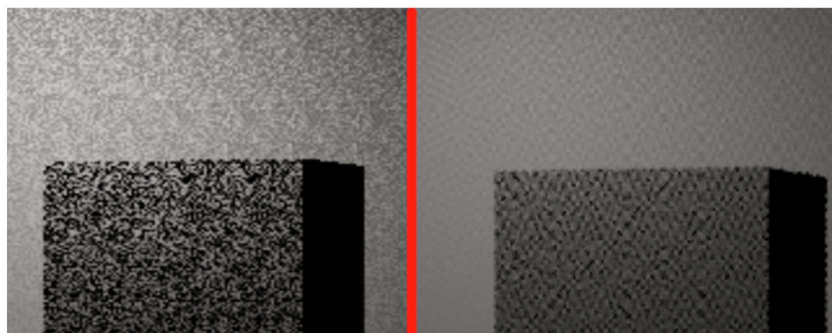
虽然完全可以直接用 WhittedRT 函数递归去实现，没必要再使用一个新的函数 calSpecularLi() 去调用 WhittedRT，但为了模仿一下 PBRT，我还是选择这么写。渲染出来的结果如下：



到目前为止，PBRT 中的 Whitted 光线追踪器终于全部实现了。

7 5 关于采样器

一开始我们说过，我们选择的采样器是 rand() 函数来生成随机数，但是 PBRT 中使用一些算法来生成伪随机序列，单帧渲染结果对比如下：



左图是我们的 Whitted 积分器渲染一帧（单像素采样一次）生成的效果，右图是 PBRT 的 Whitted 积分器渲染单像素采样一次生成的效果。可以看到右图比左图噪声分布更均匀，效果也更好。在以后的系列中我会对 PBRT 的采样器的基本原理和移植方法进行讲解。

八　本书结语

如果您对 PBRT 本身并不了解，但花了一两天把《PBRT 文件加载和设定》学会，然后又把这本书花两三天看懂，那么您就已经可以自豪的宣称，自己已经把 PBRT 的 Whitted 光线追踪器搞懂了！

但是，PBRT 还有好多东西需要研究，任重而道远啊。

这一本书如果您没能实现在自己的引擎上也没有关系，下一本书我们就从零开始移植 PBRT 了，主要介绍 PBRT 的基本工具的实现，比如矩阵变换；形状和加速器结构。这些内容很基础，但是是构建 PBRT 的基石。之后的系列中我会开始讲解 PBRT 的材料和纹理，然后大家就能随心所欲地创造自己想要的场景了。

大概写这本小书用了四天的时间。PBRT 第三代已经变得比第二代多了很多内容，功能也更加完善了，但我还是写第三代的解析书，因为我相信只要找准了方法，PBRT3 的学习其实可以很简单。而且我相信等 PBRT 第四代更新后，和第三代相比基本架构和功能不会有太大变化，所以我们直接开始学第三代 PBRT 是一个很不错的选择。

参考文献

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

[2] Shirley P. Ray Tracing in One Weekend[J]. 2016.

[3] Shirley P. Ray Tracing The Next Week[J]. 2016.

[4] Shirley P. Ray Tracing The Rest Of Your Life[J]. 2016.

[5] https://benedikt-bitterli.me/resources/

[6] Marschner S , Shirley P . Fundamentals of computer graphics. 4th edition.[J]. World Scientific Publishers Singapore, 2009, 9(1):29-51.