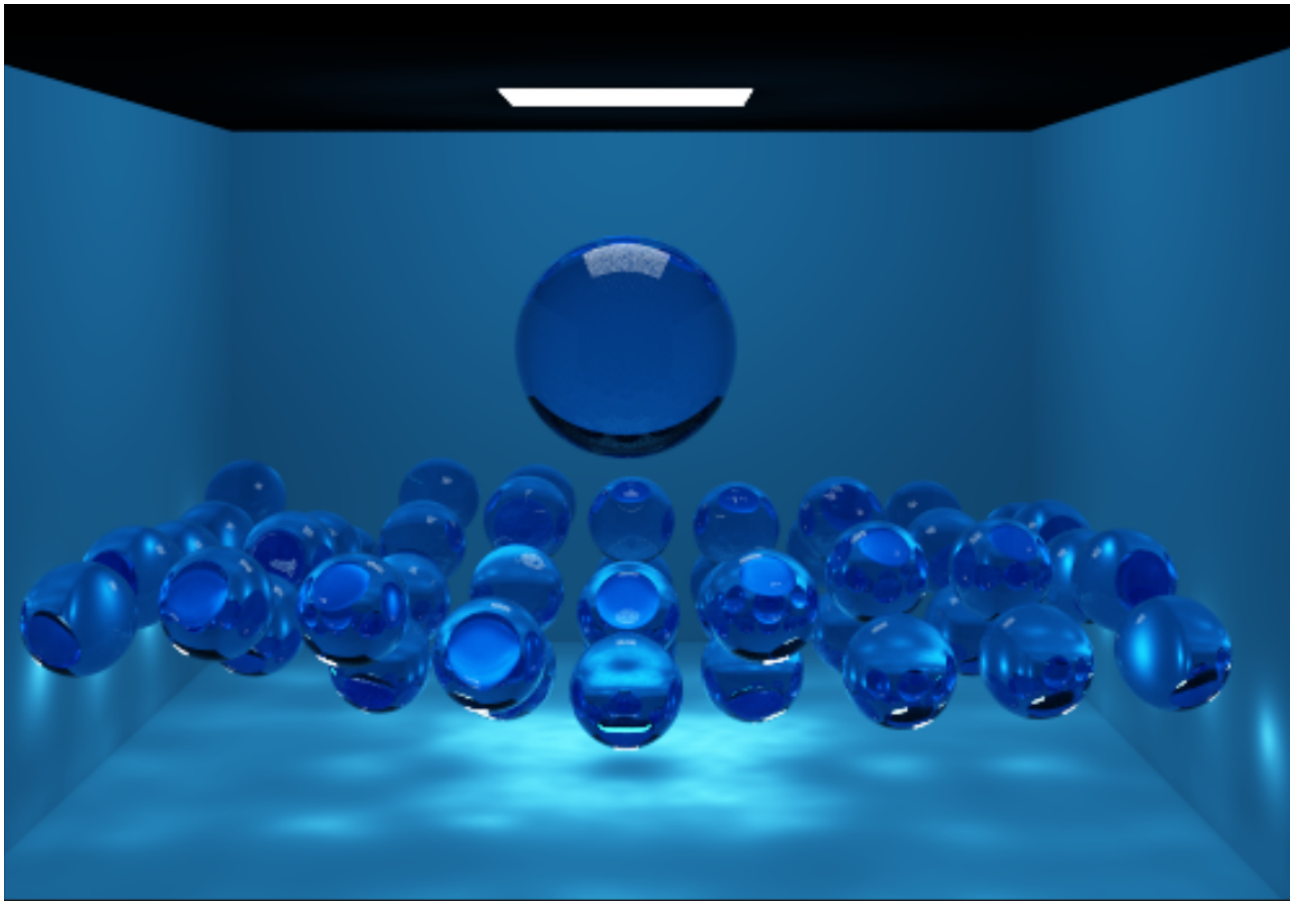
光子映射-从理论到实践-基础入门篇

Dezeming Family

2020 年 12 月 25 日

光子映射
从理论到实践



前言

以前总想写一个光子映射的实践教程，因为网上的资料都不是很亲民，最起码不像光追三部曲 [8, 9, 10]（后面会介绍这三本书）的入门这么简单，而且对于各种细节的描述也不是很透彻。不过因为科研比较忙，所以一直没有动笔。如今虽然还是特别忙，但决定开始写这个小册子，打算把这个有趣的技术，包括多种优化和升级，作为一个完备的教程来实现。我不喜欢读书的时候看到一堆当前完全看不懂的铺垫，而是喜欢走一步就弄懂一步，所以本书的风格也将是如此。大家在阅读的时候可能就会感觉到，一开始很多定义都不会特别严谨，而是根据直观理解进行解释，基于最简单的解释，我们会尽快做出一个完整的光子映射器。

本书的目标：希望读者能跟着本书，一步一步，从头到尾实现一个光子映射引擎，并在该引擎上逐步优化，达到更好的效果。本书的介绍方式是：首先介绍光子映射的机理，然后实现光子图的生成；之后再讲解利用光子图渲染光照的原理，然后把整个渲染管线制作完。在后续系列中，会在最简单的引擎基础上，进行更严格的定义和描述，然后丰富完善光子映射器。为了更准确，书中有些单词会保留英文，防止翻译不同造成较大差别，例如 radiance，kdTree 等。

其实我本想直接写完一整本书，把 PM，PPM 和 SPPM 以及基于物理的渲染都写进去，但是考虑到全都实现可能要花更长的时间，尤其是还要构思怎么写材质介绍，怎么安排章节，如果要写完完整的书估计又得几个月过去了。因为是电子书，所以比较灵活，我想，还是先把最基础的光子映射器先实现了吧。至于基于物理的光子映射器，等这本书完成以后（或者完成 PBRT 系列书以后），再开始着手去写。

Dezeming Family 网站：https://dezeming.top/ 。

本书的售价是 12 元（电子版），但是并不直接收取费用。如果您免费得到了这本书的电子版，在学习和实现时觉得有用，可以往我的支付宝账户（17853140351，备注：光子映射）支持 12 元，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！

本文于 20222 年 8 月 10 日进行小修，并为每个章节提供源代码。这是我在创建 DezemingFamily 网站之前就写作的第一本电子书，写作风格和文笔都不是很好，再次修订也很难达到自己满意，但应该是能让读者看懂的。

目录

一　本书的基本介绍

本书不会是一大堆知识的描述以及最后给出一大坨代码来劝退读者，这样更符合我们构建代码的过程，否则我们对代码如何逐步规范和设计总会感到不很得心应手。同时本书也不会给出一大堆代码，然后分段讲解（以 PBRT[11] 的体量，如果教你一步一步构建一个光追引擎，那那本书得几千页不止了，但是对于初学者而言，直接去啃大部头的书确实会比较难），而我喜欢光追三部曲 [8, 9, 10] 的风格，因此本书也会从零开始逐步构建，这是我写这个小册子的初衷。

本书在写作过程中参考了参考文献 [1]，这本书语言也比较通俗易懂，同时也有源码实现，但是是最后给出了一整个代码，没有中间构建的信息和每一步的调试过程，直接让初学者对照着后面的代码来实现恐怕有些困难，如果你没有在写代码中调试并显示一些中间结果，则很难对当前进度和目标有一个很好的把握。

在这个小册子里，我会逐步教大家一点一点从零开始构建一个光子映射器。尽管我认为这本书的零基础的，但是光子映射不同于光线追踪，它需要一个光线追踪的框架来实现光子映射算法。希望在读者开始学习光子映射之前，已经实现过光线追踪的引擎，并且学习过蒙特卡洛光线追踪的原理。

可能您没有实现过光线追踪引擎，这好办，去阅读下面的三个小册子吧：《Ray Tracing Minibooks》三部曲 [8, 9, 10]。读完这三个小册子，您就能够自己构建一个光线追踪引擎了：

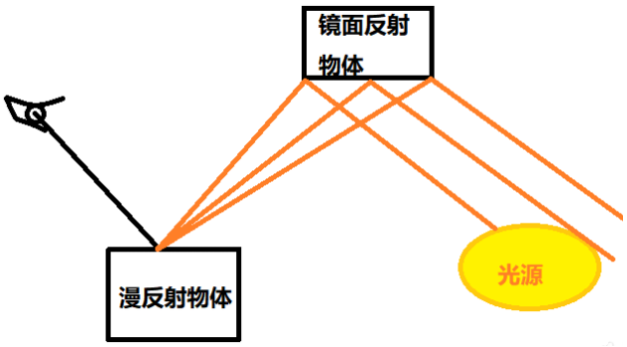
只需要三周就能学完这三本小册子，第三本就是关于蒙特卡洛光线追踪技术的。书里有代码，但是也有 BUG，您可以参考网上别人写的博客来对照着学习。我们本书的内容就是以这三部曲为基础框架的，当然，如果您以前自己实现过其他光追引擎，那么这本书您也一定能很轻松地读懂并实现在您的光线追踪引擎里。学完这三本小书以后，您已经具备了开始研究光子映射的基础，尽管这三本小书您可能需要学两到三遍才能吃透。

二　基础理论

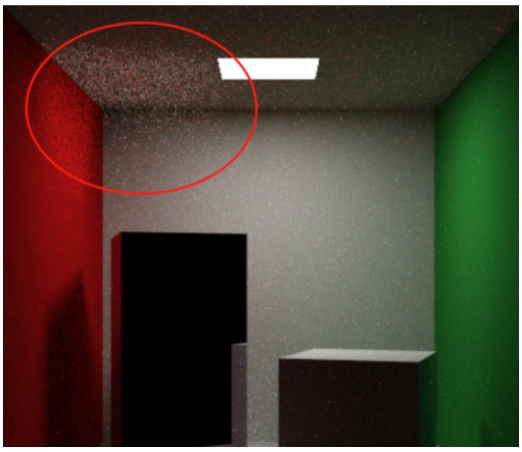
2 1　既然有了光线追踪，为什么要用光子映射

2 1.1　渲染镜面反射物体

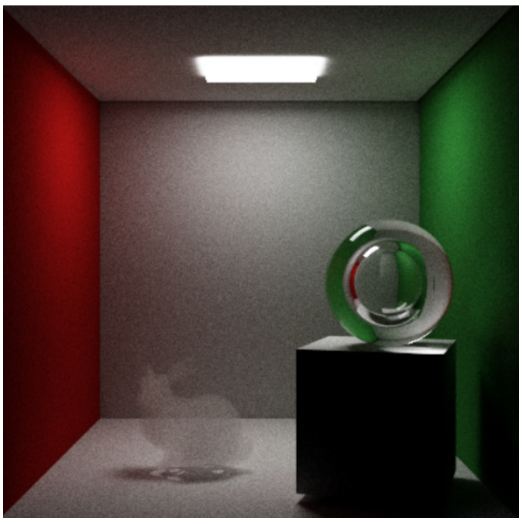
做光线追踪的时候，可能会遇到很多问题，第一个问题就是，在追踪镜面反射物体时：



按理来说光源朝镜面发射的光一定会打到漫反射物体，这个时候漫反射物体就会比较亮。但是实际采样的时候，是从眼睛开始发射光线，然后与漫反射物体相交，之后再随机打到场景内的其他物体上，从而没有对镜面反射物体进行足够的采样，因而噪声比较大：

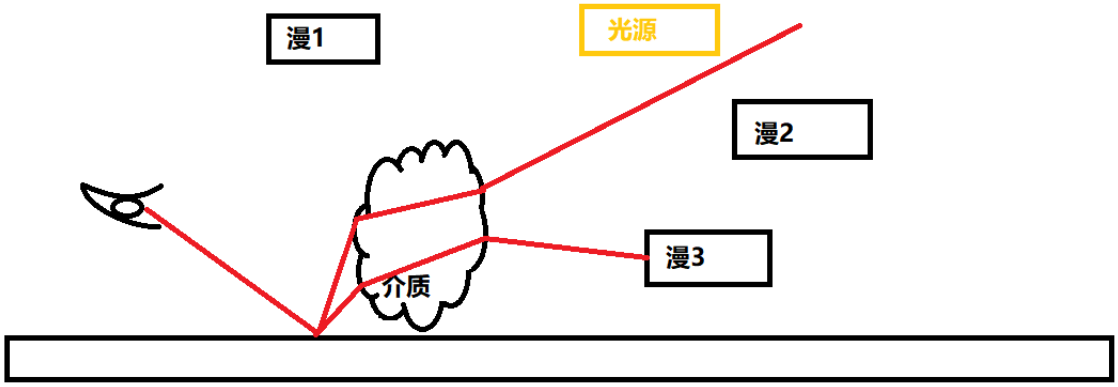


按理来说上面的区域应该很亮，而不是一堆跟噪点似的小白点。这种现象在处理不规则物体的折射和反射时更为常见。在折射类物体和镜面反射物体渲染时，光在这些物体的表面上会因为其表面曲面的反射、折射方向而形成特殊的现象，即焦散：



如上图，玻璃球的底下有亮斑，光在进入到玻璃球后会折射，折射后的光线在射出玻璃球以后会产生焦散的效果。在波光粼粼的水面，这种焦散效果尤为常见：

其中，左边是水面因为水波动折射光，在水下形成的焦散，在夏天的泳池里非常常见；中间的图是光透过水杯形成的焦散效果；右边是曲面反射光形成的焦散效果。光线追踪在渲染这种效果的时候会特别慢，因为 Ray 从你的眼睛出发，然后在场景中到处采样，焦散是光在镜面或介质（本书所说的介质，都是指玻璃这种材料，而不是指烟雾这种参与介质）经过折射或者反射，最后在漫反射表面形成的——对于光线追踪这个逆过程而言，你追踪到了漫反射表面，这个时候你下一次迭代，就算你是对镜面或介质物体进行采样（见光追三部曲的最后一本书的最后一节），球面还好说，比较规则，但是对于复杂的物体，很难保证采样到的点最终能射到光源上：



就像上图所示，哪怕你用重要性采样来发射更多光线到介质上，在不规则介质中你也很难确保光能够打中光源，因此大量的光线就被浪费掉了，导致收敛极其慢。这个时候，我们就应该想到，从视角出发，追踪到漫反射物体再去追踪镜面反射很难，而光源则恰恰相反，它只需要朝四面八方发射光子，然后光子就能根据场景里的镜面反射物体自动聚集，岂不是相当方便吗？

2 1.2　噪声的表现

光线追踪产生的噪声都是一堆高频的小噪点，因为很多光线都打不中光源。而如果我们让光源去发射光子到场景中，场景遍布了光子，然后我们根据这些光子来估计亮度，那么这个亮度的噪声就不再是高频的噪点了，而是相对比较柔和的噪声，毕竟只要场景内有光，露在外面的物体，就一定会被直接（被光源直接照亮）或者间接（被其他物体反射的光）照亮。相对于高频的噪点，我们更容易接受比较低频的噪声，而且针对低频的噪声，进行图像后处理（滤波等）也更容易。

2 1.3　信息的重复利用

在做蒙特卡洛光线追踪的时候，当视角发生变动时，图像要从头生成和计算，而这样实在是麻烦。光子映射的好处是，通过将光照的信息进行有效保存，就可以在渲染中重复利用。哪怕视角变了，但是光照信息在空间的分布是没有变的，只需要在最后射入眼睛成像时考虑视角方向的光照计算就好了。

2 2　光子散射原理

其实上面对于光子映射的动机已经说的比较明确了，这里我们再详细地阐述一下内部机制。

2 2.1 BRDF 函数

因为我们最初的实现并不依赖于 BRDF 方程，所以这里并不会严格定义，仅仅是谈一下 BRDF 的作用。

BRDF 即双向反射分布函数，描述的是对于入射光 L_i，它在某方向的反射光的比重：

$$f_r(x, \vec{\omega}', \vec{\omega}) \tag{二.1}$$

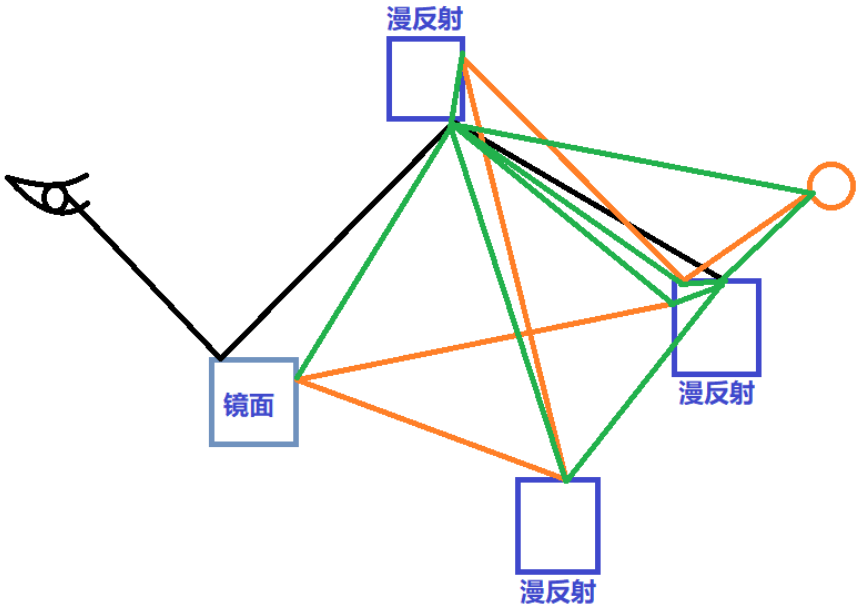
所以，从 $\vec{\omega}'$ 方向发来的光（强度为 L_i）传输到 $\vec{\omega}$ 方向的光强度为（其中，$\vec{n_x}$ 表示表面法向量）：

$$f_r(x, \vec{\omega}', \vec{\omega})(\vec{n_x}\vec{\omega}')L_i \tag{二.2}$$

为什么要乘以 $(\vec{n_x}\vec{\omega}')$ 因子我会在第四章简单讲解一下，其实这是与入射通量有关的。

2 2.2 从双向路径追踪到光子映射

可能您听过双向路径追踪这个技术，即从光发出光线，构成光路径，同时从相机发出光线，构成光路径，然后将两个路径的每个漫反射节点分别连接，根据联合概率密度，计算光照强度 [7]：



但是这种方法也有很多问题，比如光路径和视觉路径需要大量采样和优化。尽管有一些技术，例如 Metropolis 光传输算法，基于马尔科夫链来拟合目标函数，但是适用性也是比较有限的，计算过程中需要大量样本来收敛。光子映射也是一种双向路径的技术（光路径和相机采样路径），只不过它不需要存储路径，而仅仅需要把光与表面的交点进行记录保存，这样就算视点变动了，光信息还是可以复用的（虽然双向路径追踪的光信息某种程度上也可以复用，但是为了寻找贡献更多的路径以及采样收敛，还是需要多次采样和变异的，所以并不能算完全可复用信息）。

因为光子映射技术比双向路径追踪好很多，所以现在在渲染电介质材料时已经很少会使用双向路径追踪相关的方法了，而路径追踪和光子映射技术，以及结合路径追踪和光子映射技术已经显然成为了主流。

2 2.3 光子在物体表面散射

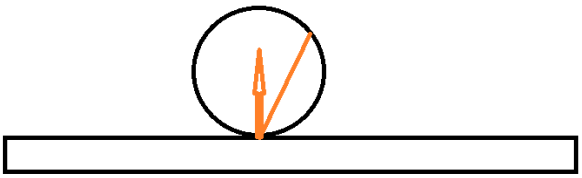
我们假设光源能够发出光子，光子的能量是根据光源特性而言的，比如对于点光源，距离越远，光子能量就越低。我们先不考虑各种光源对应的能量，而是简单阐述一下它的原理。

光子在理想镜面会直接反射，反射的公式大家肯定非常熟悉，这里就不过多介绍了：

$$\vec{\omega} = 2(\vec{n}\vec{\omega}')\vec{n} - \vec{\omega}' \tag{}$$

其中，$\vec{\omega'}$ 表示的是入射光的方向，$\vec{n}$ 表示的是表面法向量的方向。折射稍微复杂一点，因为还需要考虑全反射等现象，在之前的三部曲书中有比较基础的实现方法，大家可以参考，这里不过多解释。

漫反射就是向四面八方散射，在蒙特卡洛方法中，能量的计算是根据反射方向的概率密度来计算的，对于朗伯表面（漫反射表面）而言，反射方向的概率密度是与它和法向量的夹角的余弦成正比的（这里的夹角余弦即 $(\vec{n_x}\vec{\omega'})$ 项）：



对于任意表面 BRDF 的反射，则需要根据模型表面 BRDF 来计算相应的反射能量，或者采用接受-拒绝模式，即产生一个样本，根据 BRDF 和当前生成的随机数来决定这个样本是否被拒绝，再执行相应的迭代计算。我们一开始写程序时，当光遇到漫反射表面以后就直接停止并储存光子。这样的好处是实现简单，等我们先把最基础的渲染引擎搞定以后，读者可以再逐步完善，所以这个地方我暂时不用过多语言来阐述具体机制，大家有个印象就好了。

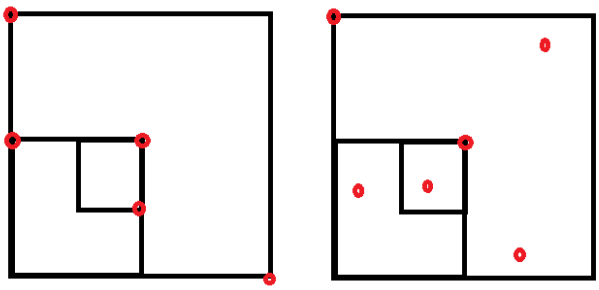
所以注意，我们当前的任务就是，构建一个光子发射器，遇到镜面物体就根据镜面反射/折射公式来计算反射/折射方向，遇到漫反射就直接停止，然后储存光子的位置（够简单了吧）。

2 3　光子的存储和表示

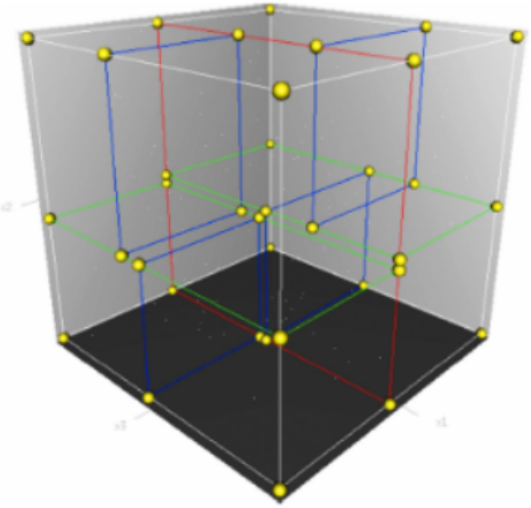
我们先从最基本的思路出发，而不是一开始就把所有的计划都实现，要不只会增加复杂性。光子的表示我们可以建立一个结构体（该结构体会在代码实现章节重新给出），注意 Vec3f 和 vec3 是同一个类型，只是一些历史遗留原因：

```
1 struct photon {
2     Vec3f Pos; //位置
3     Vec3f Dir; //入射方向
4     Vec3f power; //能量，通常用颜色值表示
5 }
```

因为可能光子图有几百万个光子，所以一定要紧凑地存储，而且需要比较快的数据结构。用不断细分的立方体网格不是一个很好的主意，因为点的位置可能非常不规则：

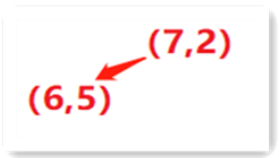


见上图，左图还好，可以细分到每个节点，适用于网格数据（体数据），但右图中，对于随机空间位置，细分的立方体就需要多次细分，才能定位到所有的位置，很不方便。而 k-dTree 这种数据结构则能够比较好得解决这个问题（图示来自网络）：



k-dTree 是一种数据结构（二叉树），在每个轴上，左子树小于中间值，右子树大于中间值。假如我们要构建 k-dTree，举个例子，有以下几个数据，它们按顺序开始建树：(7,2)(6,5)(4,8)(3,7)(6,4)(8,2)(3,3)。

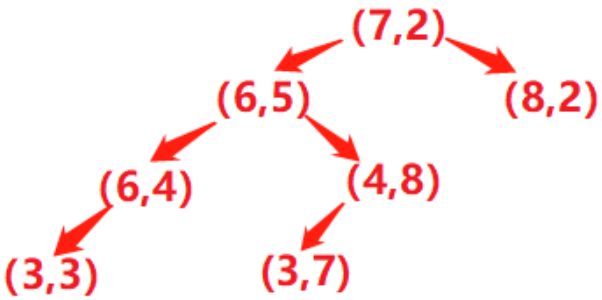
首先定位第一个点，然后我们用 x 轴作为分界面，则第二个点的 x 为 6，小于 7，插入左边（下图 (1)）。然后再插入第三个点，第三个点还是插在（7,2）的左边，但是它要插在（6,5）的左边还是右边呢？我们可以随机选择判断轴，也可以先比较 x，如果 x 相同再比较 y，如果 y 也相同再比较 z，但是这样可能会带来一个问题：如果某个光子图空间分布 x 值比较接近，而 y 值相差比较大，则根据 x 来划分不是一个很好的选择。因此我们选择用随机的轴（顺便提一句，这个选择的轴必须得记下来，不然后面没有办法去遍历搜索），我们可以第一级判断用 x，然后第二级用 y，然后第三级用 z（我们图示的例子是二维的，所以只有 x 和 y 坐标），之后循环改变。因此，照这种情况，（4,8）应该在（6,5）的右边（下图 (2)）。



图(1)

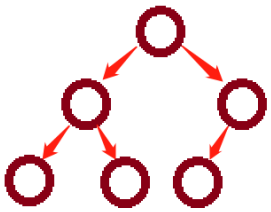
图(2)

把树依次建完，就得到了下面的图：

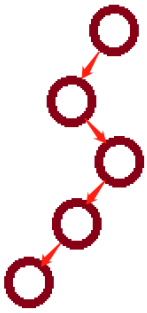


2 4　平衡 k-d 树以及定位最近的光子

上面的 k-dTree 有个问题：它的左边和右边高度不同。理想情况下，每个树干左右两边都应该是同样高的（或者左右只相差 1），这样 N 个光子的搜索复杂度就是 $O(log(N))$ 了，如下图 (1)，但是如果建树过程不太好，可能会出现极端情况，搜索复杂度变为 $O(N)$，如下图 (2)：



图(1)



图(2)

而左图，就是平衡 k-dTree。如何建立平衡 k-dTree 我们留在后面再说。

有了平衡 k-dTree，我们就要想办法来定位最近的光子了。什么是最近的光子呢？我们先不考虑关于场景的各种复杂的情况。而是直接用最简单的方法来做，即以搜索点 x 为球心，以 R 为半径的球内，找离着 x 点最近的 N 个光子。

所以我们需要一个函数，这个函数输入是当前位置 Pos，以及搜索半径 d，之后，它会返回给我们一个数据堆，堆里有搜索半径内的最近的 N 个光子，然后计算距离该点的最近的 N 个光子中最大的距离作为半径，用这 N 个光子的总功率除以面积来得到对 radiance 的估计。

这一章只是讲最浅层的原理，而没有任何与复杂的算法和实现有关的内容。如果一上来就给一大堆算法实现和理论，恐怕就会劝退很多人。下一章开始，我们就开始编程，来逐步把光子图构建起来。然后后续章节我们再考虑光子图可视化和辐射度估计，以及怎么实现它们。

三　我们的初始工程

见我们的示例代码 0-InitialWork 和 1-RayTracer。

编译环境为 VS2015，以及 Qt5.7，注意我在 CMakeLists.txt 中设置了我电脑的 QT_PATH，读者需要根据自己安装的 Qt 目录来设置，我们要求 Qt 的版本为 Qt5。除了 Qt，我们不需要安装其他第三方库（在 3rdLib 中是附加的第三方库文件：assimp 是与模型读取有关的，stb 图像库是与图像读取和写入有关的，我们暂时用不到这些功能）。

0-InitialWork 是一个最基础的工程，它可以显示使用的内存情况，以及每帧渲染使用的时间。在 Main-GUI 目录下的 RenderThread.cpp 文件中可以看到渲染循环，两层 for 循环来填充一个 framebuffer，这个 buffer 就是显示到屏幕上的结果。

1-RayTracer 是根据光追三部曲来实现的一个光线追踪器，里面的内容是完全符合光追三部曲的，故不做太多解释。但是有一点需要注意：material::scatter() 函数会 new 新的对象，但是都没有释放，所以程序运行时内存会不断累加，这是光追三部曲中自身存在的问题，可以手动释放，或者定义智能指针，但我们为了程序尽可能与原来保留一致，所以没有做什么修改，但是这一点很重要，因为你的程序可能跑几十分钟以后就发现电脑卡主，这就是因为内存被挤爆了。

四　代码 1：生成光子图和查找

代码部分是我最后再写的，我一开始先把第二章和第四章写完，才开始做第三章，对于光子映射而言，实践比理论难得多，但我相信，跟着本书一步一步把代码实现，光子映射也不会成为你的障碍！

在写程序的时候，一个很关键的地方就是调试：把我们已经写好的内容测试一下正确性。本章的方法就是，写一点程序，然后做一个调试显示，把已经写好的部分通过各种方法测试代码是否写对了。

4 1　光子追踪

本节实现一个基本的光子追踪系统，实现最基本的光子生成和存储。本节代码完成后，我们会把生成的光子图的所有点的位置显示出来。

4 1.1　场景准备

首先准备好您的光追引擎。这一节是基于已经建立的光线追踪引擎而写的，在学习本节时，可能部分代码直接抄到您的引擎里不能直接工作，还需要对照着您的引擎修改修改。如果您的引擎是根据前面介绍的光追三部曲实现的，那可能改十几分钟就能移植完，如果您的引擎和光追三部曲差别比较大，可能要改几个小时，甚至几天。总之，代码的实现过程我会详细地介绍，但是最终是否能够成功实现在您的引擎上还是得靠您的努力的。

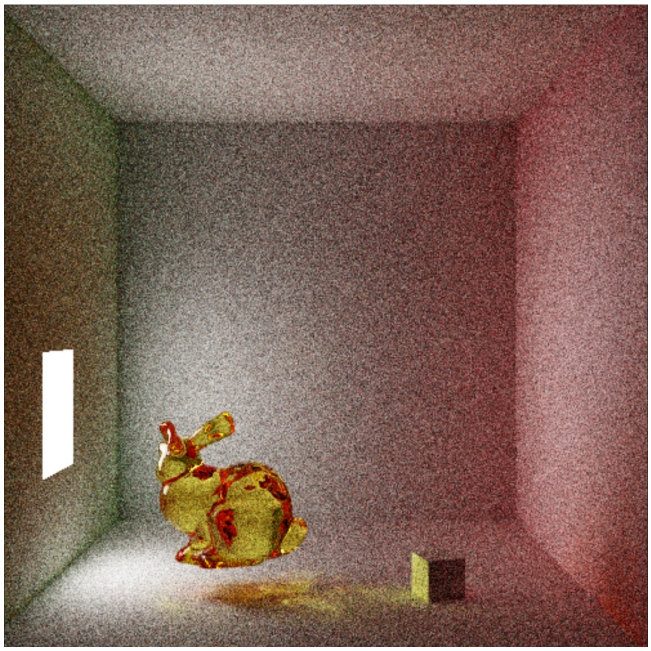
本节代码见 2-Bunny。相比于 1-RayTracer，本节代码增加了 readOffFile.h 和 readOffFile.cpp 两个文件，用来读取 bunny 模型。同时，本节将 scatter_record 的 attenuation 成员名称改为了 albedo。另外，本节把 dielectric 定义为了一个较简单的彩色玻璃（只是增加了一个 albedo，每当光射入电介质内，就会衰减一次。这种方式在物理上是错误的，真正的有色电介质的实现更为复杂，我们在这里只是为了颜色不那么单调而设置的颜色）。我还在 color() 函数里加上了释放渲染中内存的代码：

```
1 free(hrec.mat_ptr);
2 free(hrec.mat_ptr);
```

我们定义要渲染的场景：

```
1 material *light = new diffuse_light(new constant_texture(Vec3f(27.0f, 27.0f,
      27.0f)));
2 yz_rect *lightShape = new yz_rect(1.13, 2.43, 2.27, 3.32, 0, light);
3 hitable *cornell_box_PM() {
4         ......
5         return new bvh_node(list, index, 0.0, 1.0);
6 }
```

这个场景有一个面光源在左边。我们把这个场景用蒙特卡洛光线追踪的方法渲染出来。注意为了简单起见，没有使用重要性采样，所以收敛非常慢：



这幅图像我渲染了 1000 帧，才得到了上图这种一大堆噪点的效果。我们给的示例代码和上面的结果在亮暗上有点区别。

4 1.2 光子追踪框架

本部分代码见 3-Photon。

定义一个最原始的光子结构体表示：

```
#ifndef __PHOTONMAP_H__
#define __PHOTONMAP_H__
//包含的几何计算类，包括向量和矩阵等
#include "Core\Geometry.hpp"
struct Photon {
    Vec3f Pos; //位置
    Vec3f Dir; //入射方向
    Vec3f power; //能量，通常用颜色值表示
};
#endif
```

我们很难决定是否要将光子映射实现在一个类中，还是实现在一堆零散的函数里，不过放在类中倒是比较容易管理。

然后我们定义一个光子类 PhotonMap，考虑一下这个类里面应该有什么元素。首先肯定需要一个存储光子的数组，然后一个变量来记录光子的数量。同时需要一个包围盒来表示光子图的范围，用于以后定位方便。我们还需要一个函数来存储光子，就先定义这么多吧！

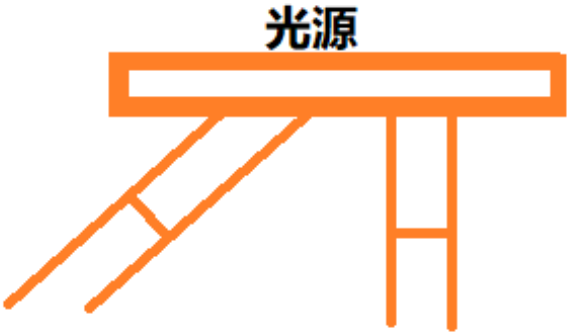
```
class PhotonMap {
    public:
        PhotonMap();
        ~PhotonMap();
        int PhotonNum;//光子数量
        int maxPhotonNum;//最大光子数量
        Photon *mPhoton;
        void store( Photon pn);
        Vec3f box_min, box_max;
```

```
        };
```

然后我们在 yz_rect 类中定义产生光子的函数（因为该场景的灯光是一个矩形面光源，该类表示为 yz_rect）：

```
//产生法向量所在的半球方向的向量
inline Vec3f random_in_half_sphere(const Vec3f &n) {
    Vec3f p;
    float mode;
    do {
        p = 2.0*Vec3f(random(), random(), random()) - Vec3f(1, 1, 1);
        mode = dot(p, p);
    } while (mode >= 1.0 || mode == 0.0 || dot(p,n)<0);
    return p;
}

virtual void generatePhoton(Vec3f& Orn, Vec3f& Dir, float& PowerScale,
    hit_record &rec) {
    Orn = Vec3f(y0 + random()*(y1 - y0), k, z0 + random()*(z1 - z0));
    Dir = random_in_half_sphere(rec.normal);
    PowerScale = dot(Dir, rec.normal);
}
```

注意为什么 PowerScale = dot(Dir, rec.normal)，这是因为光在不同角度发出的时候，功率是不同的，面光源单位面积发出的光在不同方向的强度与夹角有关，与接收面类似：



接下来实现追踪光子的函数，因为只是追踪较为简单，就不定义新的类了，而是直接定义函数。为了简单起见，我们追踪到漫反射表面就直接停止继续追踪，直接把能量全部保留在漫反射表面：

```
void tracePhoton(const Ray&r, hitable *world, int depth,Vec3f Power,
    PhotonMap* mPhotonMap) {
    //记录击中信息，例如击中点，表面向量，纹理坐标等
    hit_record hrec;
    if (world->hit(r, 0.001, MAXFLOAT, hrec)) {
        //记录当前点散射信息，比如是否为镜面反射等
        scatter_record srec;
        if (depth < 6 && hrec.mat_ptr->scatter(r, hrec, srec)) {
            //如果是镜面就继续反射/折射
            if (srec.is_specular) {
                tracePhoton(srec.specular_ray, world, depth + 1, Power,
```

```
                    mPhotonMap);
            }
            else {
                //注意我们假设介质并不吸收光子
                Photon pn;
                pn.Pos = hrec.p;
                pn.Dir = r.direction();
                pn.power = Power;
                mPhotonMap->store(pn);
            }
        }
    }
}
```

然后就可以写光子追踪的程序了。注意我们这里还没有实现 PhotonMap 里面的一些函数，不过先不急，把框架搭起来再说：

```
extern hitable *world;
PhotonMap * mPhotonMap = new PhotonMap(10000);
void generatePhotonMap() {
    // 生成物体场景
    world = cornell_box_PM();
    Vec3f Origin, Dir, Power = Vec3f(27.0f, 27.0f, 27.0f);
    float PowScale;
    for (int i = 0; i < 10000; i++) {
        lightShape->generatePhoton(Origin, Dir, PowScale);
        Ray r(Origin, Dir);
        tracePhoton(r, world, 0, PowScale*Power, mPhotonMap);
    }
}
```

框架有了以后，我们把光子图类给完善一下，然后写个调试语句，追踪一下光子，然后把调试信息输出，给大家直观感受一下。之后我会用 OpenGL 把所有的光子点进行点可视化，生成一个分布图。我们每完成一定量的内容，就会进行调试，来显示直观的结果，这样大家就能比较清楚地感受到我们正在一点一点的进步。

4 1.3 光子图类的完善

之后完善一下光子图的类目前主要需要完善两部分，一是构造函数和析构函数，二是存储函数：

```
PhotonMap::PhotonMap() {
    maxPhotonNum = 10000;
    PhotonNum = 0;
    mPhoton = new Photon[10000];
    box_min = Vec3f(1000000.0f, 1000000.0f, 1000000.0f);
    box_max = Vec3f(-1000000.0f, -1000000.0f, -1000000.0f);
}
PhotonMap::PhotonMap(int max) {
    maxPhotonNum = max;
    PhotonNum = 0;
```

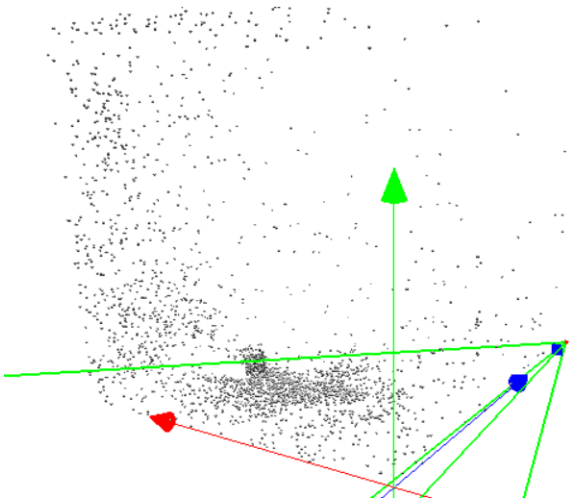
```
11      mPhoton = new Photon[max];
12      box_min = Vec3f(1000000.0f, 1000000.0f, 1000000.0f);
13      box_max = Vec3f(-1000000.0f, -1000000.0f, -1000000.0f);
14  }
15  PhotonMap::~PhotonMap() {
16
17  }
18  void PhotonMap::store(Photon photon) {
19      if (PhotonNum >= maxPhotonNum) return;
20      mPhoton[PhotonNum++] = photon;
21      box_min = Vec3f(std::min(box_min.x, photon.Pos.x), std::min(box_min.y,
            photon.Pos.y), std::min(box_min.z, photon.Pos.z));
22      box_max = Vec3f(std::max(box_max.x, photon.Pos.x), std::max(box_max.y,
            photon.Pos.y), std::max(box_max.z, photon.Pos.z));
23  }
```

我们可以把光子在数组存储在 0——光子总数-1 的索引上，也可以存储在 1——光子总数的索引上。上面的示例中选择了前者，但这样在写一些程序的时候可能不太符合人的主观感觉和计算，容易出错，所以更好的方法是存储在 1——光子总数的索引上。

之后我们就已经可以往场景内发射和存储光子了，光子存储在 mPhotonMap->mPhoton 里，可以自己写个函数，把结果输出或者存储在文件里，然后使用一些工具来可视化。我选择用 OpenGL 来把这些点可视化，程序比较长，而且跟本书的内容无关，我就不把程序放在这里了。

把结果显示一下，为了更清楚，我只记录焦散光子，即光子直接打到漫反射材料时我会直接舍弃，这其实只需要在光子追踪的程序里判断一下就好了：

```
1       //如果光子直接达到了漫反射材料（迭代深度=0）
2       if (depth == 0) return;
3       else {
4           //说明是先打到镜面物体再打到漫反射材料的
5           储存光子
6       }
```



上图可以比较清楚地看到光子的分布，在 bunny 的后下方聚集了很多光子。

4 2　平衡 k-dTree

这节牵扯到算法的知识，有一定难度，所以就单独弄一节出来。我们之前已经知道了平衡 k-dTree 长什么样子，但是距离能够构造出来还差很远呢。

在 PhotonMap 类内定义一个平衡方法：void PhotonMap::balance(); 参数暂时不能确定。

我来阐述一下平衡算法的伪代码描述：

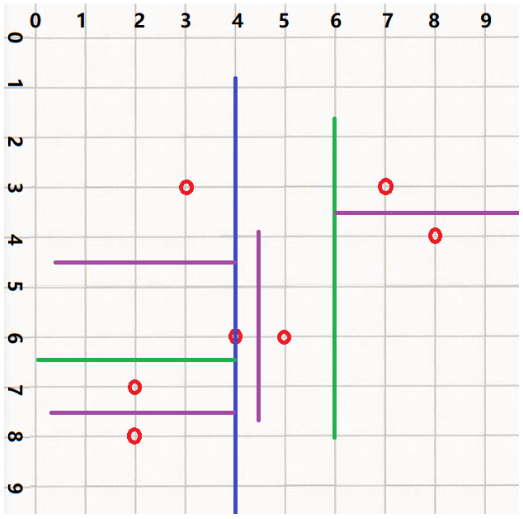
```
void PhotonMap::balance(points){
    找到包围这些点的包围盒。
    选择某个包围盒最长的边作为划分光子位置的维度
    找到该维度上的光子位置中间值，node = 中间光子
    left = 所有位置低于中间值的光子
    right = 所有位置高于中间值的光子
    node->left = balance(left)
    node->right = balance(right)
    得到node
}
```

4 2.1　平衡二叉树的中间节点位置

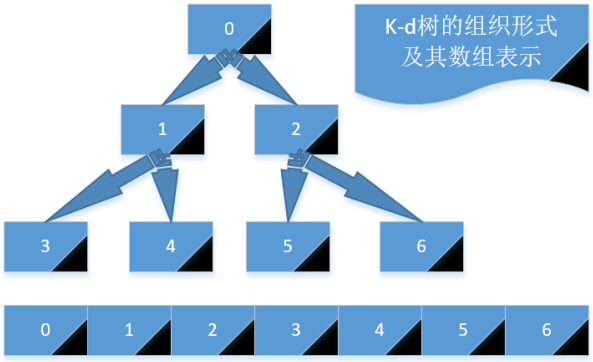
平衡 k-dTree 可以是使用常规的建树方法，即结构体包含两个指针和数据位：

```
struct kdNode{
    Photon pn;
    kdNode *left;
    kdNode *right;
};
```

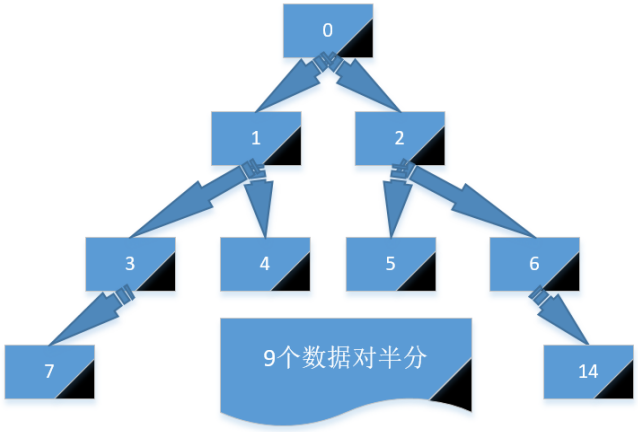
但是这样比较耗费内存，因为我们建树的方式比较特殊，不是获得一个节点然后插入一个节点，而是已经有了全部的数据，而且可以进行排序，因此我们可以将树直接存储在数组里。我们来实际操作一下，假设我们有之前的数据：(7,2)(6,5)(4,8)(3,7)(6,4)(8,2)(3,3)，包围盒的长为 6，宽为 5，所以以长边来区分，然后剩下的再逐步判断：



前 3 个分到 left 组里，后 3 个分到 right 组里。我们直接用数组来存储，注意数组的坐标和树节点对应关系：



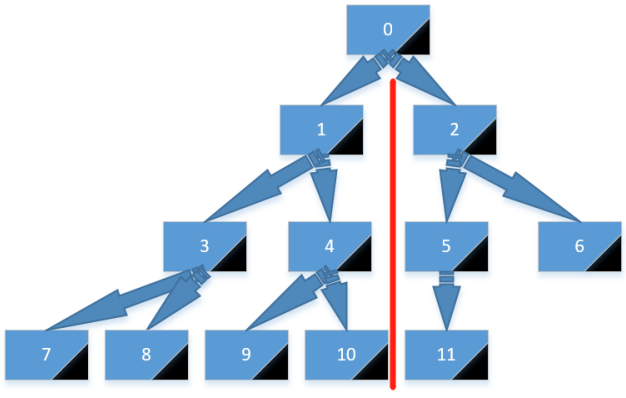
但是，假如我们有 9 个数据，我们一直对半分，则会出现类似这种情况：



对应的数组坐标里，坐标 7 到 14 里面有一些坐标缺失。我们知道一共有 9 个节点，对应数组坐标为 0-8，如果我们平衡一下中间点的位置，让树的最后一个位置对应的数组坐标正好是数据个数-1，所以需要换种方式计算中间数。

注意下面的叙述与数组坐标有关（坐标从 0 到数据总个数-1），假如我们有 3 个数，则中间数坐标是 1；4 个数，中间数坐标是 2；5 个数，中间数坐标是 3，6 个数，中间数坐标还是 3。这种树有个定义，叫做完全二叉树，大家可以动手画一下。怎么找到中间节点位置呢？我们还是假设有 12 个数据，那么，我们尝试铺满每一层，第一层是 1 个数据，第二层是 2 个数据，第三层是 4 个数据，以此类推。

当加完第 5 层的时候，铺满需要 15 个数据，但是我们只有 12 个，我们做个图表示一下：



红线将中间值位置分开，左边是左子树，右边是右子树。当铺满需要的 15 个数据大于我们已有数据 12 时，我们令 15 减去这一层的一半，15-4=11，而 11 小于 12，说明最后一层在红线右边有一个节点。把上面的过程编程表示为：

```
int calMed(int start, int end) {
    int num = end - start + 1;
    int med;
    int as = 1, b = 2;
    while (as < num) {
```

```
        as += b;
        b *= 2;
    }
    if (as == num)
        return start + num / 2;
    b /= 2;
    if (as - b / 2 < num) {
        return start + as / 2;
    }
    else
        return start + as / 2 - (as - b / 2 - num);
}
```

给出一个 start，一个 end，我们就能找到若要建立完全二叉树结构的中间值的位置，这个位置是相对于起始位置 start 的偏移，之后整个数据就可以分为两部分，一部分比中间值小，一部分比中间值大。

4 2.2 寻找平衡二叉树的中间节点

本小节主要内容为从一组数据中找到中位数（并不是严格意义的中位数，而是从小到大第 med 位置的数）。

因为要比较的是 x,y,z 三个轴的位置，方便起见，我们写个函数，来直接索引第 index 的每个坐标值：

```
float PhotonMap::getPhotonPosAxis(int index, int axis) {
    return mPhoton[index].Pos[axis];
}
```

该函数定义后只是备用，为了调试打印方便。我们给每个光子一个记录分界轴的信息：

```
    int Photon::axis;
```

仅仅找中位数还不行，还得令左边的值全都小于中位数，右边的值全都大于中位数，这就很难了（当然也有很简单的思路，只不过会特别浪费时间）。如果您学过数据结构与算法，可能能够想到使用堆排序等方法快速排序并找中位数，我给出的参考文献 [2]，您可以从中找自己的思路，为了简单起见，我直接给出根据参考文献 [1] 中修改的代码，为了方便操作，我们修改一下以前的设置，让数组的索引为从 1 到 n，索引为 0 的位置不放任何元素：

```
    void PhotonMap::MedianSplit(Photon* tempPhoton, int start, int end, int
        med, int axis) {
        int l = start, r = end;
        while (l < r) {
            double key = tempPhoton[r].Pos[axis];
            int i = l - 1, j = r;
            for (; ; ) {
                while (tempPhoton[++i].Pos[axis] < key);
                while (tempPhoton[--j].Pos[axis] > key && j > l);
                if (i >= j) break;
                std::swap(tempPhoton[i], tempPhoton[j]);
            }
            std::swap(tempPhoton[i], tempPhoton[r]);
            if (i >= med) r = i - 1;
            if (i <= med) l = i + 1;
```

```
        }
    }
```

上面算法的主要思想就是从数组两侧不断向内收缩，每轮收缩的过程中，得到末尾的值，并把小于末尾的值和大于末尾的值进行交换。自己拿几个数试试就能弄明白了。

4 2.3 生成平衡二叉树

这个部分的代码就比较简单了，上面已经叙述过，直接给代码就好了。

```
void PhotonMap::balance() {
    Photon* tempPhoton = new Photon[PhotonNum + 1];
    for (int i = 1; i <= PhotonNum; i++)
        tempPhoton[i] = mPhoton[i];
    balanceSegment(tempPhoton, 1, 1, PhotonNum);
    delete[] tempPhoton;
}

void PhotonMap::balanceSegment(Photon* tempPhoton, int index, int start,
    int end) {
    if (start == end) {
        mPhoton[index] = tempPhoton[start];
        return;
    }
    int med = calMed(start, end);
    int axis = 2;
    //选择使用哪个轴来进行分叉
    if (box_max.x - box_min.x > box_max.y - box_min.y && box_max.x -
       box_min.x > box_max.z - box_min.z) axis = 0; else
        if (box_max.y - box_min.y > box_max.z - box_min.z) axis = 1;
    MedianSplit(tempPhoton, start, end, med, axis);
    mPhoton[index] = tempPhoton[med];
    mPhoton[index].axis = axis;
    if (start < med) {
        double tmp = box_max[axis];
        box_max[axis] = mPhoton[index].Pos[axis];
        balanceSegment(tempPhoton, index * 2, start, med - 1);
        box_max[axis] = tmp;
    }
    if (med < end) {
        double tmp = box_min[axis];
        box_min[axis] = mPhoton[index].Pos[axis];
        balanceSegment(tempPhoton, index * 2 + 1, med + 1, end);
        box_min[axis] = tmp;
    }
}
```

需要注意的是，每次细分的时候都要根据轴来计算递归到下一级的包围盒大小，等递归回来之后再恢复原值。调试相对容易，尽管可能有偶然性，但是一般调试成功就不会有问题。我们把生成的平衡 kdTree

数组打印出来，包含它用来划分下一级的轴：

```
0    3.84745 0.0193548 2.01175 axis:1
1    3.30279 0 2.29711 axis:0
2    5.55 4.197 3.34284 axis:1
3    2.90595 0 1.75339 axis:2
4    3.68488 0 1.79798 axis:2
5    5.55 2.47523 0.0704501 axis:0
6    2.87025 5.55 0.114246 axis:0
7    2.97469 0 1.52212 axis:0
8    2.21122 0 2.45654 axis:0
9    4.57494 0 1.49264 axis:0
    3.62875 0 2.27056 axis:100
    3.84631 0.0252064 2.01523 axis:100
    5.55 1.64238 1.62909 axis:100
    0.0746229 5.55 1.34276 axis:100
    4.82198 5.55 0.251443 axis:100
    2.67924 -5.96046e-8 1.31797 axis:100
    3.20912 2.98023e-8 1.52446 axis:100
    2.19095 0 2.4808 axis:100
    2.5242 0 2.2247 axis:100
    3.56771 0 1.58068 axis:100
```

轴为 100 意味着属于叶节点，不再往下划分了。完全 kdTree 树的层次关系是：索引 0 左指针指向索引 1，右指针指向索引 2，索引 0 用来比较的轴是 y 轴，可以看到索引 1 的 y 轴小于索引 0 的 y 轴，而索引 2 的 y 轴大于索引 0 的 y 轴。然后，索引 1 的左指针指向索引 3，右指针指向索引 4，依次类推。

4 3 找最近的 N 个光子

找最近的 N 个光子的技术理解起来并不困难，尽管比较考验编程能力，它涉及标准二叉树搜索的扩展以及范围搜索。我们要在给定的体内（例如圆盘、球体等）找到最近的 N 个光子。

方法是在给定半径内，我们找到一个光子就塞进数组里，然后排序，如果超出数组范围，就删掉里面最远的光子。一般的排序算法时间复杂度比较高，最好的方法是使用最大堆来实现，这里推荐一个课程 [3]，里面有很多基础的数据结构与算法，大家可以学习一下。以及推荐我写的博客 [4]，我把这个系列课程所有的编程题全部做完并把答案和思考过程写在了里面。

4 3.1 表示最近 N 个光子的结构体

把结构体给出：

```
    struct Nearestphotons {
        Vec3f Pos; //搜索点位置
        int max_photons, found; //最大光子数和已找到的光子数
        bool got_heap; //最大堆是否已满（即found等于max_photons）
        float* dist2; //存储最大堆内光子距搜索点距离的平方，避免每次重复计算
        Photon** photons; //存储已经找到的光子的数组
        Nearestphotons() {
            max_photons = found = 0;
            got_heap = false;
            dist2 = NULL;
```

```
            photons = NULL;
        }
        ~Nearestphotons() {
            delete[] dist2;
            delete[] photons;
        }
    };
```

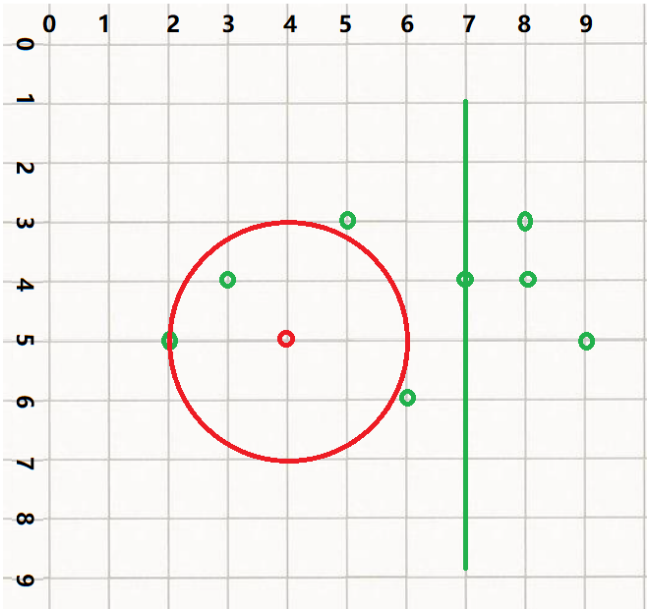
我们要找半径 R 内的 N 个光子，如果有两种可能，一种是半径 R 内光子不够 N 个，则一直往里面填充即可。另一种是半径内光子大于等于 N 个，如果大于 N 个，就要使用最大堆来增加距离更小的光子以及剔除距离更大的光子。注意存储的是搜索距离的平方，这是因为我们只需要比较平方距离即可，开平方运算过于耗费时间。

4 3.2 找最近的 N 个光子的算法

前面已经说得差不多了，直接用伪代码来表述。

```
//输入：光子图（平衡k-dTree）索引p,最近光子结构体np,最大搜索半径的平方d2
//输出：存储最近N个光子的最大堆h
getNearestPhotons(p, np, d2){
    if(2p+1 < 光子总数){
        //说明当前索引有子节点
        d_tmp = 光子x到当前光子p的分界轴的距离（有正负）
        if(d_tmp < 0){
            //我们在分界面左边,搜索左子树
            getNearestPhotons(2p, np, d2)
            if(d_tmp2 < d2){
                //搜索右子树
                getNearestPhotons(2p+1, np, d2)
            }
        }else{
            //我们在分界面右边，搜索右子树
            getNearestPhotons(2p+1, np, d2)
            if(d_tmp2 < d2){
                getNearestPhotons(2p, np, d2)
            }
        }
    }
    //计算当前索引的距离
    d_tmp2 = 光子p到x的距离平方
    if(d_tmp2 < d2){
        把光子插入到最大堆中
        d2 = 最大堆h根节点值（如果最大堆没有满，则根节点的值为d2）
    }
}
```

我们判断是否去递归子节点的方法是，判断 x 距离当前节点的分界线的正负值，如下图：红点代表检测点 x，x 的最大距离半径是 2；绿线经过的绿点代表当前点 p,p 划分子节点的轴是绿线。如果 x 到绿线的距离大于搜索半径，则绿线右边的点就不用再去搜索了，直接搜索左边的点即可。



实际代码是这个样子：

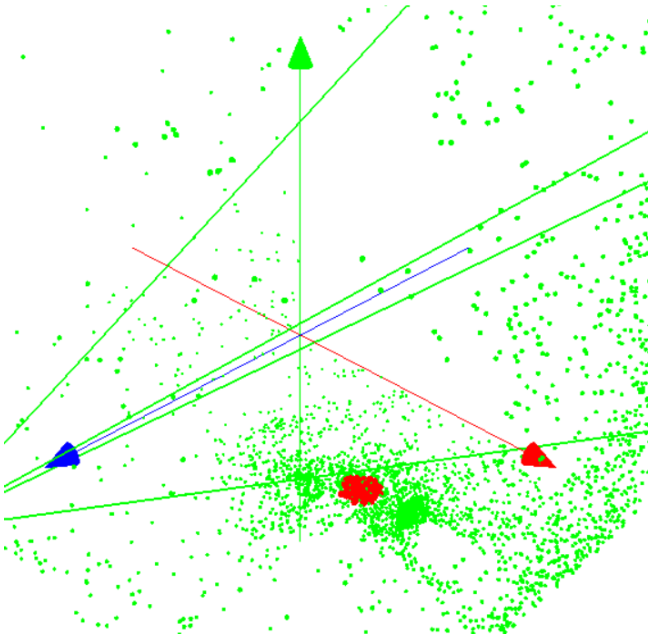
```
1   void PhotonMap::getNearestPhotons(Nearestphotons* np, int index) {
2       if (index > PhotonNum) return;
3       Photon *photon = &mPhoton[index];
4       if (index * 2 <= PhotonNum) {
5           double dist = np->Pos[photon->axis] - photon->Pos[photon->axis];
6           if (dist < 0) {
7               getNearestPhotons(np, index * 2);
8               if (dist * dist < np->dist2[0]) getNearestPhotons(np, index
                    * 2 + 1);
9           }
10          else {
11              getNearestPhotons(np, index * 2 + 1);
12              if (dist * dist < np->dist2[0]) getNearestPhotons(np, index
                    * 2);
13          }
14      }
15      float dist2 = squareDistance(photon->Pos, np->Pos);
16      if (dist2 > np->dist2[0]) return;
17      if (np->found < np->max_photons) {
18          np->found++;
19          np->dist2[np->found] = dist2;
20          np->photons[np->found] = photon;
21      }
22      else {
23          if (np->got_heap == false) {
24              for (int i = np->found >> 1; i >= 1; i--) {
25                  int par = i;
26                  Photon* tmp_photon = np->photons[i];
27                  float tmp_dist2 = np->dist2[i];
28                  while ((par << 1) <= np->found) {
29                      int j = par << 1;
```

```
                    if (j + 1 <= np->found && np->dist2[j] < np->dist2[j
                        + 1]) j++;
                    if (tmp_dist2 >= np->dist2[j]) break;

                    np->photons[par] = np->photons[j];
                    np->dist2[par] = np->dist2[j];
                    par = j;
                }
                np->photons[par] = tmp_photon;
                np->dist2[par] = tmp_dist2;
            }
            np->got_heap = true;
        }
        int par = 1;
        while ((par << 1) <= np->found) {
            int j = par << 1;
            if (j + 1 <= np->found && np->dist2[j] < np->dist2[j + 1]) j
               ++;
            if (dist2 > np->dist2[j]) break;

            np->photons[par] = np->photons[j];
            np->dist2[par] = np->dist2[j];
            par = j;
        }
        np->photons[par] = photon;
        np->dist2[par] = dist2;

        np->dist2[0] = np->dist2[1];
    }
}
```

为了调试，我们随机找一个点 (3.0,0.0,2.0)，然后找到最近的 N 个光子:

```
Nearestphotons npn;
    npn.Pos = Vec3f(3.0,0.0,2.0);
    npn.max_photons = 100;
    npn.dist2 = new float[100 + 1];
npn.photons = new Photon*[100 + 1];
//搜索距离的平方，存储在第0个索引位置，这样该最大堆的最顶端永远是堆中最
   大的值
    npn.dist2[0] = 0.6 * 0.6;
    mPhotonMap->getNearestPhotons(&npn, 1);
    for (int i = 1; i <= npn.found; i++) {
            Photon pn;
    pn = *npn.photons[i];
    //之后把光子位置打印出来或者存储在文件里
    }
```

我们像之前一样，使用 OpenGL 将收集到的最近的光子显示成红色，其他所有光子显示成绿色：



可以看到我们的搜索半径内最近 N 个光子的算法是成功的。

五　辐射度估计和光子图可视化

5 1　光照量的估计

这一节就该严肃认真地讨论一下光子和辐射度的估计了。

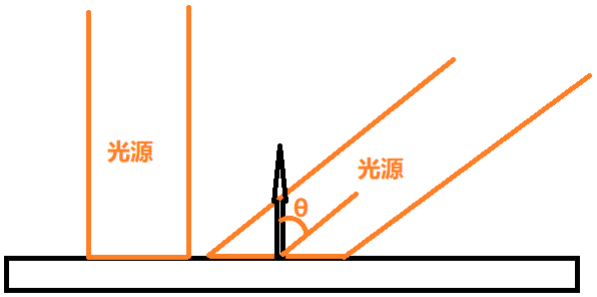
当光子图上的某个区域有光子的时候，就说明这个区域接受到了直接光或间接光。但是，仅仅从一个光子或者某个点处的光子数量我们无法得到这个区域（或者这个点）的光照量，因为光子的分布具有一定的随机性，而不是向自然界的光一样遍布整个模型（自然界里测光强，我们只需要在任意位置用一个光敏传感器，就能得到这个位置的光强了）。

密度的估计是一个统计学的概念，基于直方图思想的核密度估计方法最适用，但是一般的统计方法需要假设变量都是随机的，比如，你捉一些样本鱼来估计鱼缸里的各种鱼的比例，它不考虑哪种鱼是否好捉。而光照模型中，我们必须得考虑这一点，因为光子的入射方向和反射方向决定了反射的能量占入射的能量的比例（由 BRDF 决定）。

根据渲染方程:

$$L_r(x,\vec{\omega}) = \int_\Omega L_i f_r(x,\vec{\omega}',\vec{\omega})(\vec{n_x}\vec{\omega}')d\vec{\omega}' \tag{五.1}$$

其中，$\vec{\omega}'$ 表示的是入射光的方向，$\vec{n}$ 表示的是表面法向量的方向。这里的 L_i 表示的是入射光的 radiance，表示射入表面的单位面积单位时间的能量（单位时间射入的能量，我们叫做功率，用 Φ 表示）。因为射入可能不是垂直射入的，在某个表面，如果光斜着射入，则单位面积的入射功率会比较小（这就是上面的公式为什么要乘以 $cos(\theta) = (\vec{n_x}\vec{\omega}')$）：



所以我们可以把入射光 radiance 表示为：

$$L_i(x,\vec{\omega}') = \frac{d^2\Phi_i(x,\vec{\omega}')}{(\vec{n_x}\vec{\omega}')d\vec{\omega}'dA_i} \tag{五.2}$$

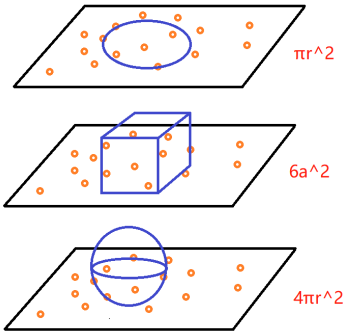
然后把这个公式代入到渲染方程，得到：

$$L_r(x,\vec{\omega}) = \int_\Omega f_r(x,\vec{\omega}',\vec{\omega})\frac{d^2\Phi_i(x,\vec{\omega}')}{d\vec{\omega}'dA_i}d\vec{\omega}' \tag{五.3}$$

这里的单位面积的功率，即 radiance 可以表示为最近的 n 个光子的功率除以这 n 个光子的所在范围，令 n 包含最近的 n 个光子的最小半径为 r，每个光子的功率分别为 Φ_p，则空间中某个点 x 射到方向 ω 的 radiance 为：

$$L_r(x,\vec{\omega}) \approx \sum_{p=1}^{N} f_r(x,\vec{\omega}',\vec{\omega})\frac{\Phi_p}{\Delta A} \tag{五.4}$$

包含最近的光子的范围可能是一个圆盘（面积 πr^2），也可能是一个球面，或者是一个正方体：



如果是圆盘的话，ΔA 就是 πr^2 了：

$$L_r(x,\vec{\omega}) \approx \sum_{p=1}^{N} f_r(x,\vec{\omega}',\vec{\omega})\frac{\Phi_p}{\pi r^2} \tag{五.5}$$

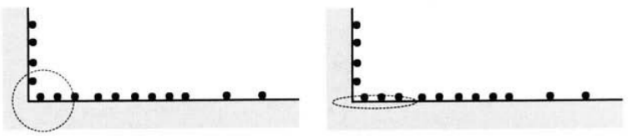
算法描述：

```
//输入：当前点位置，光反射方向，法向量
radianceEstimate(Pos, outv, n){
    定位最近的n个光子;
    r = Pos 到这n个光子中最远的距离;
    flux = 0;
    for(每个光子p){
        flux += BRDF * p.power;
    }
    Lr = flux / (2 * pi * r*r);
    return Lr;
}
```

不过上面对于光照量的物理表示非常不规范，但是我们的初步目标是先实现一个效果图，所以就算不规范也没关系。

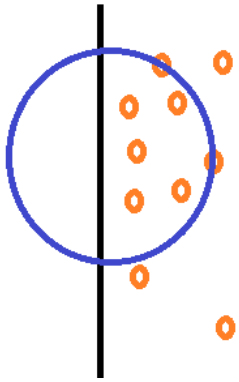
5 2 一些应该注意的问题

当估计光照量时，在墙角可能会遇到这种情况：

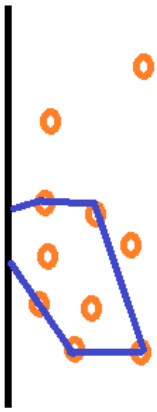


如果用球体包含范围来得到最近的 N 个光子，则墙角会特别亮（如左图）。所以可以使用一个以表面法向量方向压缩的圆盘（矮圆柱）来表示范围，这样就会比较准确。

但是在墙边的时候，可能仍然会有比较大的误差：



因为这时，为了包含 N 个光子，圆盘半径会很大（有很大的区域在墙外），针对这种情况，我们可以使用计算多边形凸包的方法：



这样就不会导致使用圆盘而半径过大了。但是这种方法还是比较麻烦，其实比较简单的方法就是进行滤波。关于滤波的内容我们放在后面再描述，因为它毕竟算是一种优化。我还是坚持认为实现一个引擎应该一步一步走，先把整体框架打好，然后再慢慢优化里面的内容。

5 3　渲染光学现象

对于光子映射图来说，它包含的全部信息是：光源经过 0 次到多次镜面反射或者漫反射，最终到达漫反射的路径。而对于以往的光线追踪方法，它除了包含漫反射路径，还包括光源直接到眼睛，以及光源经过镜面反射直接到眼睛的过程。所以我们使用光子映射来“渲染”间接光照（也就是说，光子没有经过任何镜面反射直接打中漫反射表面的光我们不能记录为光子），然后用光线追踪来渲染直接光照，把直接光照和存储的间接光照组合起来。

对于前面章节提到过的焦散效果，这非常适用，因为焦散就是光子被介质到处折射，在漫反射（不光滑）表面形成的效果。而对于渲染焦散，仅仅需要少量的光子就能得到非常好的效果，这是蒙特卡洛光线追踪根本无法实现的。为了做个对比，我会在下一章代码实现部分放一张蒙特卡洛光线追踪实现的图，等您完成光子映射渲染以后，相互对比一下，就能感受到光子映射在渲染焦散效果上的巨大优势。

Color Bleeding 是一种现象：当你在红纱包围的婚房里，你会感觉到眼前的物体都会泛红，即使是雪白的墙壁。这是由于光在漫反射表面相互反射造成的。这对于光子映射也很容易实现，因为光子映射会记录漫反射物体相互反射的信息。不过在传统的光线追踪中，Color Bleeding 也可以被比较容易地渲染出来，只是收敛可能比较慢，毕竟间接光总是比直接光要弱很多，而且光线追踪渲染的 Color Bleeding 主要是高频噪声多，而光子映射的 Color Bleeding 主要是低频噪声，视觉上更容易接受。

六　渲染步骤

这一节的叙述比较多得参考了参考文献 [1]，但是对内容的讲解做了新的规划，本节的意义是，在前面已经构建的知识的基础上，给出比较合理的渲染理论和方法。

6 1　渲染方程再议

BRDF 方程可以拆分成两个项：朗伯漫反射和完美的镜面反射：

$$f_r(x, \overrightarrow{\omega}', \overrightarrow{\omega}) = f_{r,S}(x, \overrightarrow{\omega}', \overrightarrow{\omega}) + f_{r,D}(x, \overrightarrow{\omega}', \overrightarrow{\omega}) \tag{六.1}$$

同理，入射的 radiance 也可以分为几个项：

$$L_i(x, \overrightarrow{\omega}') = L_{i,l}(x, \overrightarrow{\omega}') + L_{i,c}(x, \overrightarrow{\omega}') + L_{i,d}(x, \overrightarrow{\omega}') \tag{六.2}$$

其中，$L_{i,l}$ 表示直接光照，$L_{i,c}$ 表示直接光经过镜面反射或者介质折射以后的间接光，$L_{i,d}$ 表示直接光经过至少一次漫反射以后的反射光。

因此，整个渲染方程就可以写为：

$$L_i(x, \overrightarrow{\omega}') = L_{i,l}(x, \overrightarrow{\omega}') + L_{i,c}(x, \overrightarrow{\omega}') + L_{i,d}(x, \overrightarrow{\omega}') \tag{六.3}$$

于是，整个渲染方程就可以写为：

$$L_r = \int_\Omega f_r L_{i,l} cos(\theta) + \int_\Omega f_{r,S}(L_{i,c} + L_{i,d}) cos(\theta) + \int_\Omega f_{r,D} L_{i,c} cos(\theta) + \int_\Omega f_{r,D} L_{i,d} cos(\theta) \tag{六.4}$$

6 2　光子图的生成

一般情况下，我们会使用两种光子图，一种是用于渲染焦散的焦散光子图，另一种是用于渲染全局光照的全局光子图。

焦散光子图是由直接光经过反射或者介质折射以后在漫反射表面生成的。所以在焦散光子图生成时，光子碰到漫反射物体就停止了，因为漫反射物体是不会发生散射到别处产生焦散的。不过对于特别强的间接光，也是会有焦散现象，这个时候可以用蒙特卡洛光线追踪来渲染这种焦散。

而全局光子图则是包括全部光学现象的，包含直接光，间接光和焦散，所以焦散光子图和全局光子图不能直接相加，而是经过计算来结合。

6 3　渲染光子图

渲染光子图需要用到光线追踪的方法，光线在不断迭代中，需要经历两种类型的计算，一种是准确计算，另一种是近似。

需要准确计算的场合：（1）眼睛直接能看到该表面，或者是经过镜面反射（介质折射）后能直接看到表面。（2）Ray 的出发点距离相交点非常近，比如从靠近墙角的位置出发，然后击中了墙角附近的位置。因为这个时候，精确计算可以消除一些潜在的错误的颜色溢出（近似计算的话很可能使墙角过亮）现象（前面已经提到过）。

需要近似计算的场合：迭代中的 Ray 与漫反射表面相交，或者从漫反射表面出发，如果光照贡献比较小（毕竟是漫反射嘛），这个时候就得近似计算了。

6 3.1 直接光照与镜面反射

对于直接光照：

$$\int_{\Omega} f_r L_{i,l} cos(\theta) \tag{六.5}$$

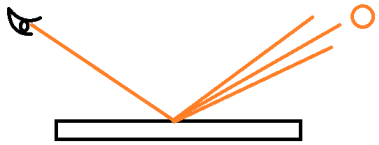
传统光线追踪时，通过 Ray 发射个阴影光线就好了，如果阴影光线被物体遮挡，就不被照亮，非常简单，但是可能会比较慢。

而采用 radiance 估计的方法，从全局光子图中就可以得到直接光照的信息。

对于镜面反射（包括光滑曲面反射）：

$$\int_{\Omega} f_{r,S}(L_{i,c} + L_{i,d}) cos(\theta) + \tag{六.6}$$

这个计算就没法使用光子图了，毕竟光子图生成的光子并不在镜面上，否则就得用一大堆光子才能模拟在镜面处的分布（因为分布确实比较尖锐）：



比如上图，只有圆锥内的光能够反射到眼睛中，所以如果要在镜面生成光子图，就得用海量光子来模拟这个光的分布，同时计算的时候还得分别计算，特别浪费时间和内存。所以计算采用传统的 MonteCarlo 光线追踪就可以了，根据 BRDF（即 $f_{r,S}$）来进行重要性采样，就能很快得到比较精确的结果（因为概率密度比较集中）。因为我们暂时只关注完美镜面反射和完全漫反射，所以我们只简单介绍这么多。等我们把最简单的光子映射器实现以后，再扩充功能。

6 3.2 焦散和多次反射

对于焦散：

$$\int_{\Omega} f_{r,D} L_{i,c} cos(\theta) \tag{六.7}$$

焦散直接使用焦散光子图来进行 radiance 估计就好了；也可以在全局光照图中进行近似计算，只是得需要更多的光子。

对于多次漫反射：

$$\int_{\Omega} f_{r,D} L_{i,d} cos(\theta) \tag{六.8}$$

精确计算就是使用蒙特卡洛光线追踪了，但是过于费时，不过也有不少方法可以改进。而近似计算则是使用全局光子图的 radiance 估计，全局光子图包含的是所有直接光、焦散和间接光的信息。

七　代码 2：光子图可视化

可视化的代码并不是非常复杂，因为基本上就是光线追踪。但是可以看到我们之前做的关于辐射度的理论和铺垫少得可怜，几乎没有什么几何光学和能量表示的物理方程，也没有非常严格的数学物理定义，这并不是说这些不重要，只是我们如果一开始就顾及一大堆知识，企图一口气就实现最完备的光子映射引擎，无疑是不太可能的——更准确的内容我放在了 PBRT 系列书里。

本节代码见 4-PhotonMapping。

7 1　辐射度估计

估计辐射度的代码还是直接放在 PhotonMap 类里比较合适，给一个坐标位置，计算其辐射度，还需要输入当前位置所在表面的法向量，最大搜索半径以及最大光子数 N

```
Vec3f PhotonMap::getIrradiance(Vec3f Pos, Vec3f Norm, float max_dist, float
    N);
```

接下来是能量的计算，但是鉴于我们还没有对能量的表示做很好的规划和设计，所以就简单点实现：

```
Vec3f PhotonMap::getIrradiance(Vec3f Pos, Vec3f Norm, float max_dist, float
    N) {
    Vec3f ret(0.0,0.0,0.0);
    Nearestphotons np;
    np.Pos = Pos;
    np.max_photons = N;
    np.dist2 = new float[N + 1];
    np.photons = new Photon*[N + 1];
    np.dist2[0] = max_dist * max_dist;
    getNearestPhotons(&np, 1);
    if (np.found <= 8) return ret;
    for (int i = 1; i <= np.found; i++) {
        Vec3f dir = np.photons[i]->Dir;
        //先不管BRDF，只计算一共有多少个光子
        if (dot(Norm,dir) < 0) ret = ret + np.photons[i]->power;
    }
    //先不管发射了多少光子，直接除以10000。这样的好处是防止能量一开始设置的
        不对，导致图像特别暗啥都看不出来
    ret = ret * (1.0f / (10000 * PI * np.dist2[0]));
    return ret;
}
```

直接让能量除以 10000：在我们本章做渲染的时候，我们先不关注能量大小，而是根据视觉效果和亮度来调节，这样比较简单和容易实现。

7 2　光线追踪

为了表现出正常的颜色，我把兔子模型设置称为无色透明。同时加上光源：

```
material *bunnymtrl = new dielectric(1.5, Vec3f(1.0, 1.0, 1.0));
list[index++] = new yz_rect(1.13, 2.43, 2.27, 3.32, 0.01, light);
```

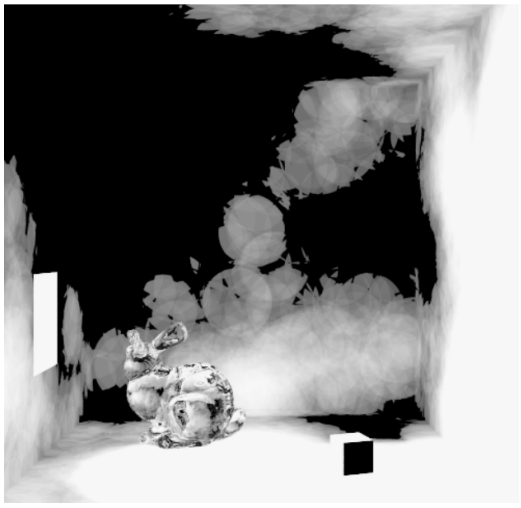
光线追踪的程序很简单：

```
Vec3f color_PMPT(const Ray&r, hitable *world, int depth) {
    hit_record hrec;
    if (world->hit(r, 0.001, MAXFLOAT, hrec)) {
        scatter_record srec;
        Vec3f emitted = hrec.mat_ptr->emitted(r, hrec, hrec.texU, hrec.texV,
            hrec.p);
        if (depth < 10 && hrec.mat_ptr->scatter(r, hrec, srec)) {
            if (srec.is_specular) {
                return srec.albedo*color_PMPT(srec.specular_ray, world,
                    depth + 1);
            }
            else {
                Vec3f col = mPhotonMap->getIrradiance(hrec.p, hrec.normal,
                    0.6, 100);
                return col;
            }
        }
        else
            return emitted;
    }
    else
        return Vec3f(0, 0, 0);
}
```

在上面的程序中，如果折射次数太多（超过 3 次）就不返回颜色了。否则，如果是镜面，就继续追踪，到漫反射表面的时候把光子能量计算出来。

```
float renderFrame_PMPT() {
    for (int i = 0; i < ThreadNum; i++) {
        for (int j = 0; j < ThreadNum; j++) {
            int offset = (i + ThreadNum * j);
            float u = float(i + rand() / (RAND_MAX + 1.0)) / float(ThreadNum
                );
            float v = float(j + rand() / (RAND_MAX + 1.0)) / float(ThreadNum
                );
            Ray r = cam.get_ray(u, v);
            Vec3f colLgt = de_nan(color_PMPT(r, world, 0));
            //压缩范围
            clamp(colLgt.x, 0.0, 1.0);
            clamp(colLgt.y, 0.0, 1.0);
            clamp(colLgt.z, 0.0, 1.0);
            //存储colLgt
        }
    }
}
```

渲染以后，结果果然特别亮：



我们把能量除以一个更大的值，比如我设置的 1000000，得到：

焦散的效果看着还挺好。

7 3　追踪全局光子

tracePhoton 可以设为捕捉全局光子，也可以设为只捕捉焦散光子（即下面注释的部分，我在程序里写成了两个函数：traceGlobalPhoton() 和 traceCausticsPhoton()）：

```
1  void traceGlobalPhoton(const Ray&r, hitable *world, int depth,Vec3f Power,
       PhotonMap* mPhotonMap) {
2      //记录击中信息，例如击中点，表面向量，纹理坐标等
3      hit_record hrec;
4      if (world->hit(r, 0.001, MAXFLOAT, hrec)) {
5          //记录当前点散射信息，比如是否为镜面反射等
6          scatter_record srec;
7          if (depth < 4 && hrec.mat_ptr->scatter(r, hrec, srec)) {
8              if (srec.is_specular) {
9                  traceGlobalPhoton(srec.specular_ray, world, depth + 1, Power
                       , mPhotonMap);
10             }
11             else {
12                 //注释掉迭代为0就返回的情况。表示保留直接射到漫反射表面的光
                       子
13                 //if (depth == 0) return;
```

```
                //else
                //{
                    Photon pn;
                    pn.Pos = hrec.p;
                    pn.Dir = r.direction();
                    pn.power = Power;
                    mPhotonMap->store(pn);
                //}
            }
        }
    }
}
```

修改一下函数，我们希望单独用 10 万个光子记录全部种类的反射，并用另外 10000 个光子来记录焦散：

```
mPhotonMap = new PhotonMap(200000);
Vec3f Origin, Dir, Power = Vec3f(27.0f, 27.0f, 27.0f);
float PowScale;
while(mPhotonMap->PhotonNum < 100000) {
    lightShape->generatePhoton(Origin, Dir, PowScale);
    Ray r(Origin, Dir);
    traceGlobalPhoton(r, world, 0, PowScale*Power, mPhotonMap);
}
while (mPhotonMap->PhotonNum < 110000) {
    lightShape->generatePhoton(Origin, Dir, PowScale);
    Ray r(Origin, Dir);
    tracePhotonCaustics(r, world, 0, PowScale*Power, mPhotonMap);
}
```

得到结果：

把兔子 Bunny 的颜色改回之前的黄色，然后我们在焦散能量这里修改一下能量的表示：

```
    tracePhotonCaustics(r, world, 0, PowScale*Power*Vec3f(0.87, 0.49, 0.173)
        , mPhotonMap);
```

虽然如果从真实物理角度理解我们的行为一定是错误的，但是目前我只是想搞个还不错的视觉效果，所以谁在乎物理呢？

这一章我就打算写这么多，这是休闲和娱乐的一章，Enjoy 你的光子映射引擎吧！

八　光子映射引擎的完善

上面的章节里，我们已经拥有了一个不算很正规的光子映射器。尽管不是很正规，但是麻雀虽小五脏俱全。而这一章，我们就要争取扩展一下这个光子映射引擎，让它更丰富更完善。这一章主要是从一些细节上入手优化，而 PBRT 系列书中则会从物理的角度进行严谨的定义和实现。

8 1　滤波

一看到前面渲染的图像满满的噪声，心里就不是很舒服，于是决定先滤波。

虽然场景是三维的，但是因为光子主要是定位在表面上的，所以应该使用二维滤波器。在以前的光照图算法中已经有很多二维滤波器了，比如高斯滤波器和圆锥滤波器。以后我们还会研究更好的滤波器。

圆锥滤波器和高斯滤波器主要用于焦散滤波。对于圆锥滤波器，首先定义一个权重（其实就是距离越近的占的比重越大）：

$$\omega_{pc} = 1 - \frac{d_p}{k * r} \tag{八.1}$$

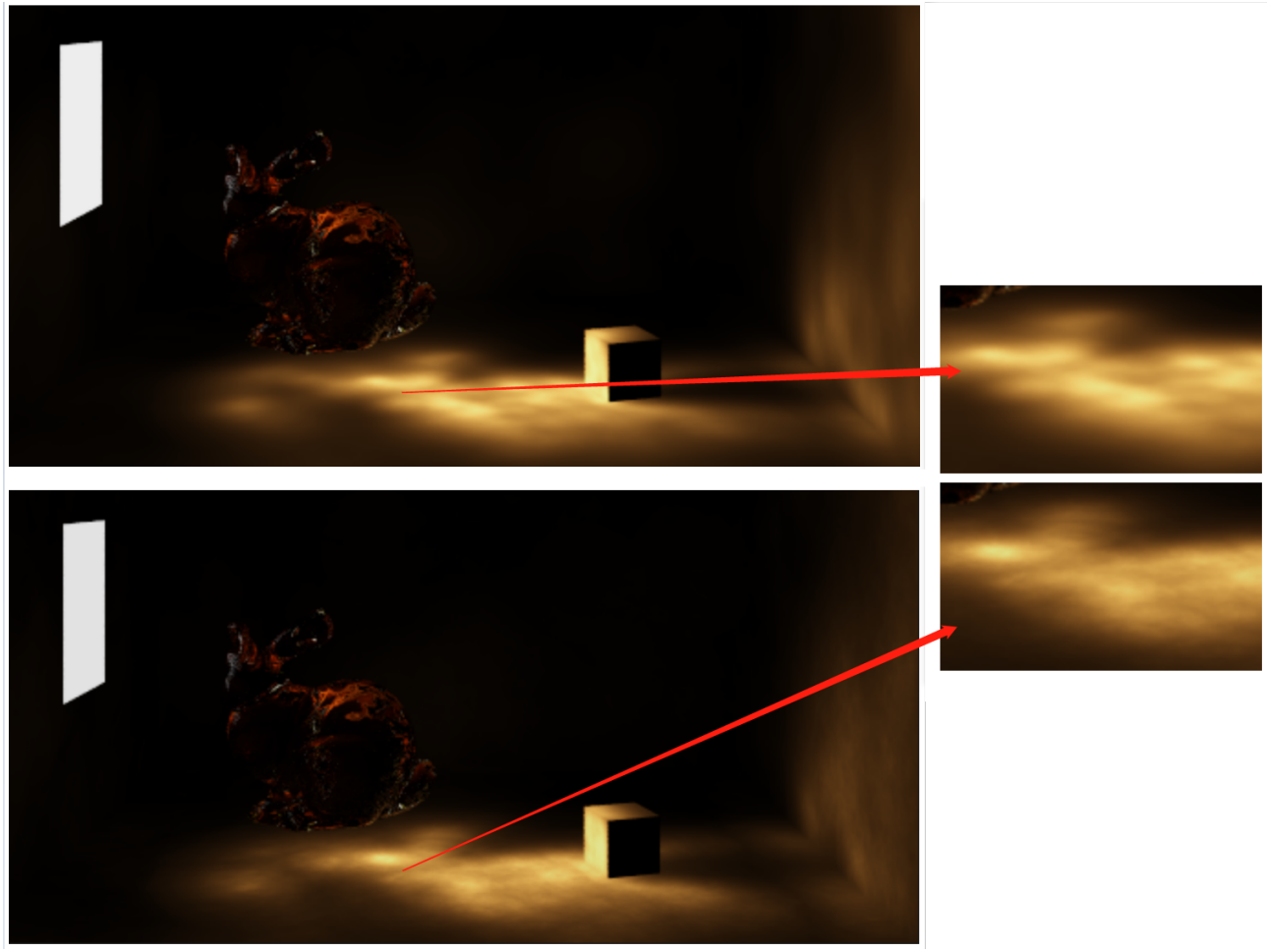
其中，d_p 表示的是从当前位置 x 到光子 p 的距离，k 是一个自定义的常数，r 是最大距离。因此最终的 L_r 表示为：

$$L_r(x, \vec{\omega}) \approx \sum_{p=1}^{N} f_r(x, \vec{\omega}', \vec{\omega}) \frac{\Phi_p}{\pi r^2} \frac{\omega_{pc}}{(1 - \frac{2}{3k})} \tag{八.2}$$

写成代码也比较简单，选择 k 的值为 1.1：

```
1 for (int i = 1; i <= np.found; i++) {
2     Vec3f dir = np.photons[i]->Dir;
3     float wht = 1.0f - Distance(Pos, np.photons[i]->Pos) / (1.1f * max_dist)
          ;
4     if (dot(Norm,dir) < 0) ret = ret + wht * np.photons[i]->power;
5 }
6 ret = ret * (1.0f / (1000000 * PI * np.dist2[0] * (1 - 2.0f / (3 * 1.1f))));
```

下面是渲染的对比图，其中上图是使用了圆锥滤波的结果，下图是不使用圆锥滤波的结果：



可以看到，使用了圆锥滤波器的焦散边界更明显，而且内部低频噪声也更小了。

我们将兔子 bunny 换成金属材质，可以渲染出这种效果：

至于高斯滤波器也同样简单，也是基于距离调整比重的方法。我们直接使用一个归一化的高斯滤波器，则光照量局就可以表示为：

$$L_r(x,\vec{\omega}) \approx \sum_{p=1}^{N} f_r(x,\vec{\omega}',\vec{\omega})\Phi_p\omega_{pg} \tag{八.3}$$

其中，高斯权重 ω_{pg} 表示为：

$$\omega_{pg} = \alpha[1-\frac{1-e^{-\beta\frac{d_p^2}{2r^2}}}{1-e^{-\beta}}] \tag{八.4}$$

设置 $\alpha = 0.918$ 以及 $\beta = 1.953$，表示归一化的高斯权重 [6]。

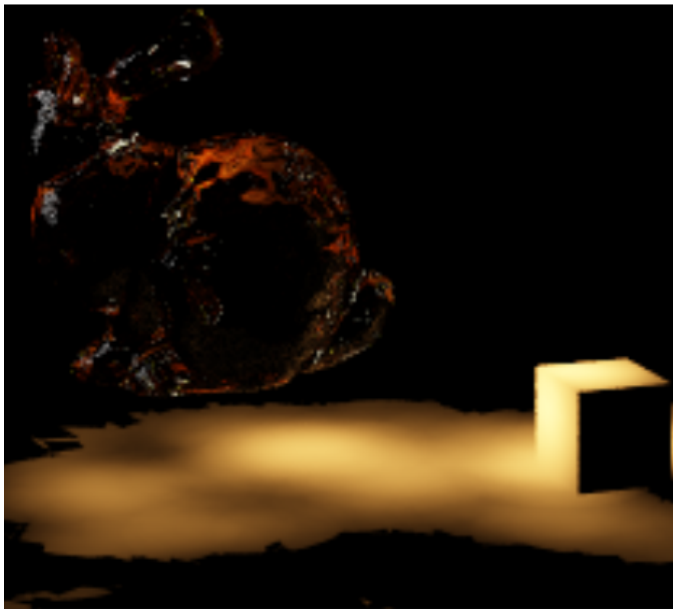
大家可以自己改改，试试结果。

8 2 光子映射的光子量和光子搜索半径

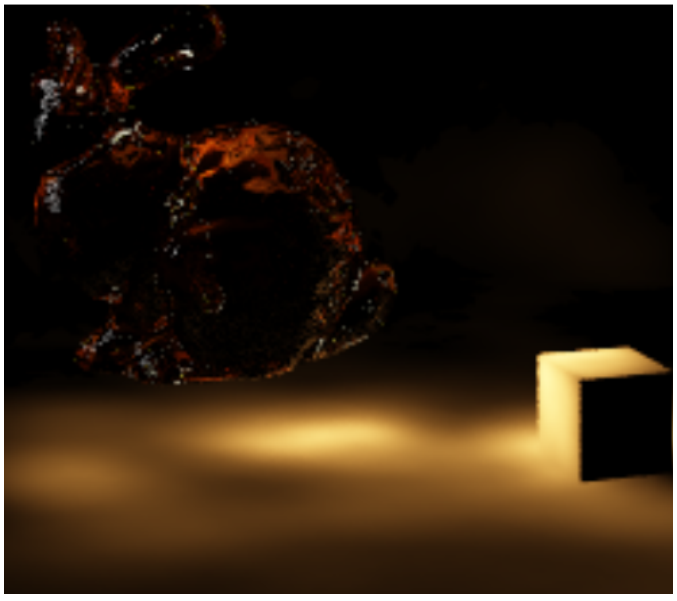
前面倒是还没有聊聊关于光子映射的搜索半径的问题。

我们一共有三个控制量：总光子数 S，最大搜索半径 R，最近光子数 N。这一小节我们简单说说这三个量应该怎么控制。

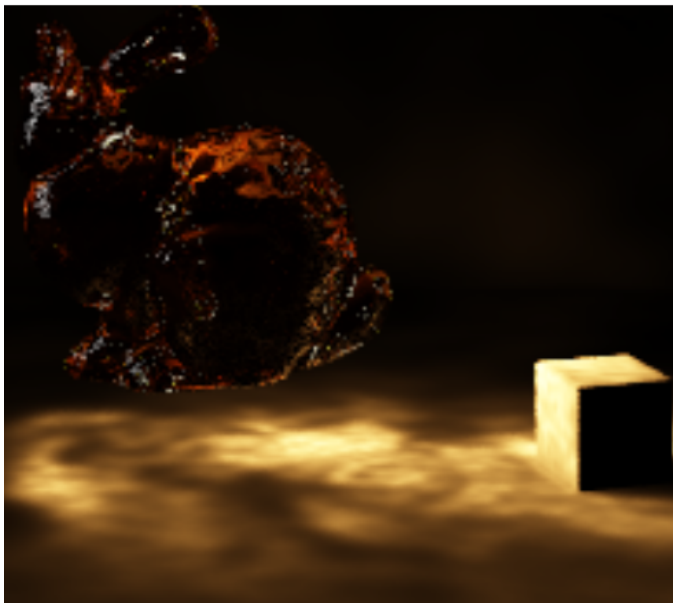
当光子数 S 比较少，同时搜索半径 R 比较小的时候，我们很容易得到这种结果：



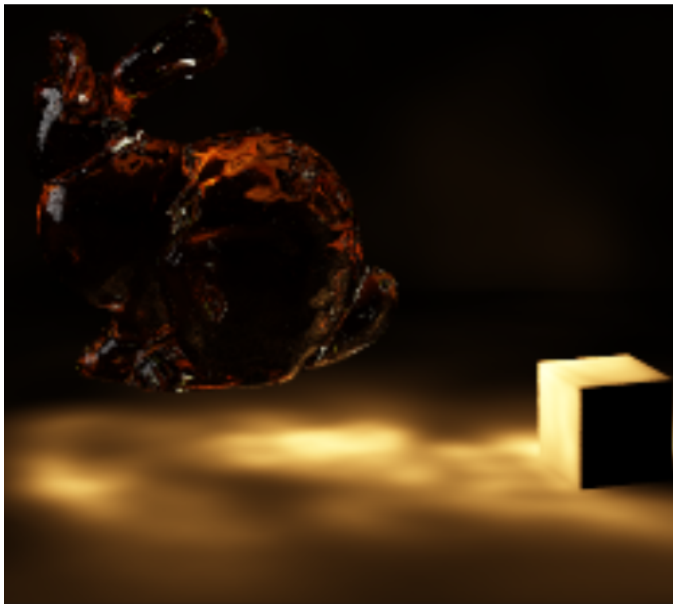
焦散光斑非常丑，就像牛奶洒在地上晒干的样子。增加光子数 S，但是搜索半径 R 和最近光子数 N 不要太大，否则可能会比较糊。尽管有焦散滤波，但毕竟滤波的效果是有限的。



如果最近光子数 N 比较小，哪怕使用了滤波，也会出现一些类似低频噪声的光斑：



设置 10 万个焦散光子，并设置最大搜索半径为 0.3（这个值是跟你场地尺寸有关的，比如我的康奈尔盒长宽是 5.55，所以最大搜索半径也不能太大），最近光子数为 400，得到效果还不错的焦散图：



8 3 光穿过有色介质的颜色变化

当光穿过透明的介质时，我们知道，白光穿过蓝玻璃后变成蓝光，所以我们直接用白光 (1.0,1.0,1.0) 乘以蓝色玻璃 (0.1,0.1,0.9) 就好了。

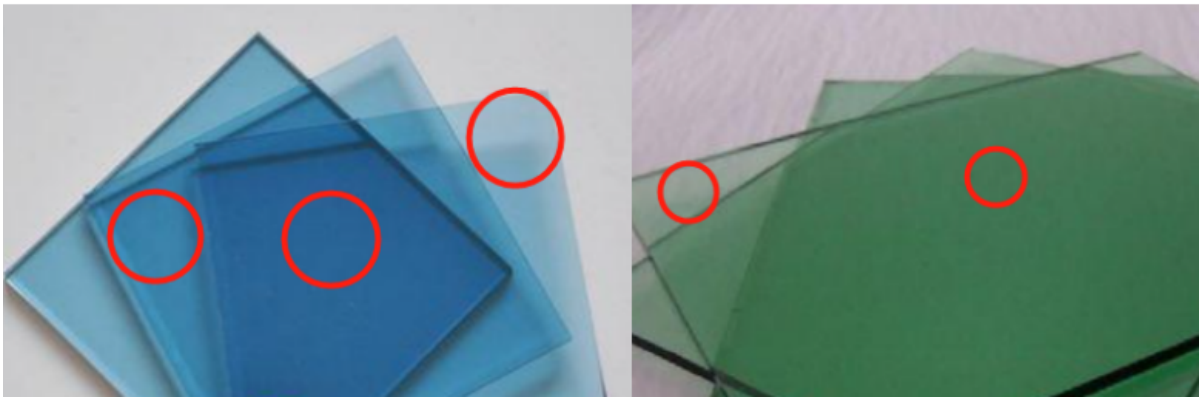
但是如果入射光不是白色呢？假如我们使用之前的黄色玻璃材质的 Bunny（RGB:0.87, 0.49, 0.173），白光两次入射到 bunny 以后，就会变成这种颜色：

222 125 44 193 61 8

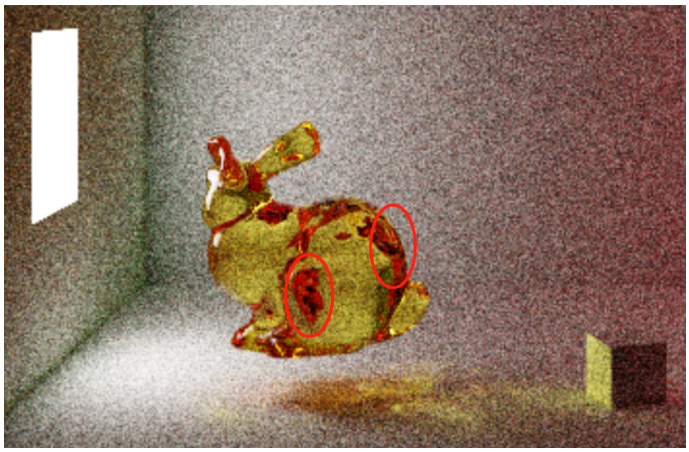
其中左图表示的是白光一次通过 bunny 后的颜色，右图表示的是白光两次通过 bunny 以后，如果都是采用直接相乘的方法得到的颜色。于是我们需要思考，如果其他颜色的光通过有色金属以后，应该是什么颜色呢？我们最大概知道，黄色光通过黄色兔子出来以后还是黄色光，而不会像上面的那样，两次相乘就变成了其他颜色的光。

尽管我认为解决这个问题比较重要，但我从网上还有论文里，以及一些专著中都没有找到合理的直接解决方案。真正的介质折射包含很多物理效应，例如光的吸收衰减，这种衰减造成颜色改变是因为衰减了不同波长的光导致的，因此可以知道，对于特别薄的有色玻璃，光穿过以后颜色并不会有较大改变，即使是红光穿过超薄蓝色玻璃，也不会全都被阻挡。

因此，我们需要计算光在介质内对不同的光分量（这里使用 RGB 表示不同的光分量）的衰减系数，然后计算衰减路径（光路径），根据衰减路径来最终得到光透过有色玻璃的颜色：



可以看到，同样颜色的玻璃叠在一起以后，光透过的颜色会变，这就是光衰减造成的。我们应该也有这样的体会，明明看着是透明的鱼缸玻璃，但是比较厚或者多个玻璃叠加在一起时，就会偏绿色。为了防止渲染器复杂化，我们把该部分放在基于物理的光方法中，当前如果想要实现有色金属，可以如此做：设置灯光为白光，当穿过有色介质时，变色，变色后的光穿过有色介质时，颜色不变。尽管效果不太对，那也比光穿过两次玻璃后颜色大变样要好得多。还记得我一开始给出的蒙特卡洛光线追踪处理的焦散效果吗，那张图上的兔子就有大量红色区域，这不是反射的墙上的光，而是错误的多次折射造成的效果（在复杂物体的褶皱区域很容易多次折射）：



九　结语

终于写完了这本并不是很厚的小书。多亏前人的辛苦劳作，使得我能够很快搭建了光子映射引擎，现在，我也想把这个技术较快地呈现给大家。

我本想将整本书的源码给放到网站上，但是有几点考虑，一是自己因为写这本小书比较仓促，源码在注释和调试以及代码结构都非常潦草，二是因为自己在调试时使用了 GUI 界面和图形显示窗口，所以多了一堆无用的信息，三是自己的代码里还实现了一堆其他渲染技术，例如双向光传输，MLT，球谐光照等，结构有些混乱，所以就不再提供完整的代码了。但我相信，这本四十页的小书已经可以帮助您从一个简单的光追引擎 [8, 9, 10] 快速构建一个光子映射引擎了。

最后，谢谢您能看到最后一章，如果您对本书有什么意见和建议，欢迎在网上留言。

参考文献

[1] Jensen H W. Realistic image synthesis using photon mapping[M]. AK Peters/CRC Press, 2001.

[2] Demaine E , Devadas S . Introduction to Algorithms[M]// Introduction to algorithms /. MIT Press, 2001.

[3] URL: https://www.icourse163.org/course/zju0901-93001《数据结构》浙江大学 MOOC 课程陈越，何钦铭

[4] URL：https://feimo.blog.csdn.net/article/details/104987342 数据结构浙江大学全部思考题 + 每周练习答案（已完结）

[5] URL：https://blog.csdn.net/zsyzgu/article/details/46592041 图像渲染技术（从光线跟踪到光子映射）

[6] Pavicic M J . Convenient Anti-Aliasing Filters That Minimize "Bumpy" Sampling[C]// Graphics gems. 1990.

[7] Veach E. Robust Monte Carlo methods for light transport simulation[M]. PhD thesis: Stanford University, 1997.

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

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

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

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