

Progressive Photon Mapping

Weng, Jian

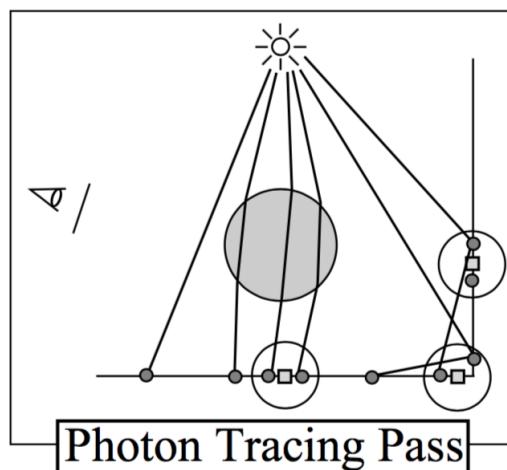
2016 年 6 月 20 日

1 算法介绍

Progressive Photon Mapping 是东京大学创意情报组蜂须贺惠也先生在 2008 年首次在 SIGGRAPH Asia 上提出的离线渲染算法，其改进版本 Stochastic Progressive Photon Mapping（在本篇的最后会提到）和 Parallel Progressive Photon Mapping，分别在 2009 和 2010 年的 SIGGRAPH Asia 上由他本人继续发表。该算法在进行有焦散线图片渲染的时候，有较好的效果。

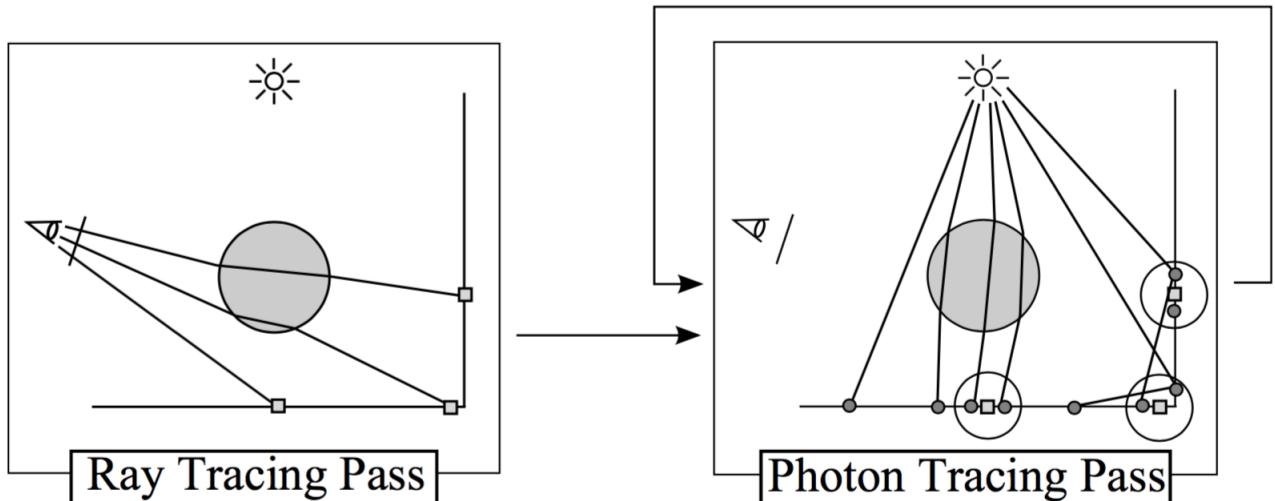
1.1 传统算法

在正式讲 Progressive Photon Mapping（以下简称 PPM）之前，允许我先介绍一下传统的光子贴图算法，这是一个 2-pass Algorithm，简而言之就是：第一趟，首先由光源释放出足够多的光子，对画面进行渲染；第二趟，从视角出发，收集相应位置邻域内的光子，进行色彩贡献的计算。对于光子数量的选取，时空面临着严峻的考验，如果光子数量过少图片渲染不收敛，如果光子过多会对存储产生光子的内存空间一个严峻的考验。这个数量是难于选取的，我们只有在释放完了光子之后才能看到渲染的效果；而本算法使用一个渐进式的手段进行渲染，可以在线看到渲染的效果，随时停止渲染。同时，邻域的大小也有其问题，因为传统的光子贴图算法邻域是一个固定值，所以无法对于邻域以内的细节进行进一步刻画，而且邻域大小的确定从经验的角度来说也是一个难题，如果邻域过小，要求更多数量的光子，对内存也是一种考验，邻域过大将无法正确刻画细节；而本算法使用一个渐进式的手段进行渲染，可以在渲染的过程中动态调整邻域的大小，不断缩小搜集光子的邻域，具体的过程会在下面几节详述。



1.2 算法流程

PPM 算法在光子贴图算法流程上进行了改进，首先进行一遍视点追踪，收集可以看见的地方，然后再进行光子播撒，待光子击中物体立马对“可以看见的地方”进行比对，如果在邻域内，就计算色彩的贡献。随着光子的播撒，邻域的大小会不断收缩（后面会详细介绍收缩的步骤）。下面几节会详细介绍如何收集“可以看见的地方”，如何快速地进行比对，如何进行色彩贡献计算的时候进行校正（从直觉上来说，因为邻域越小，一块地方会被渲染的次数就会增加，那么就会变得过亮产生违和）。



1.2.1 Eye/Photon Tracing Pass

正如上左图所示，逐像素收集可以看到的视点并不是简单的收集第一次看到的点，而是根据折射和反射，第一次走到的漫反射平面或者是到达一个反射次数的上界的尽头。这么做是为了能够做出焦散线的效果，那些更加容易被走到的漫反射平面上的点，从直觉上来说，更加会拥有更加明亮的色彩，这就是“焦散”。而光子播撒光子的过程中，几乎和发射视点的过程一模一样，色彩贡献的计算也是光子到达一个漫反射平面时才进行的，除了发射的起始点不同之外，还有一个不同是视线的光路一定要走完，但是的光路在几何体中折射可能会被“俄罗斯轮盘赌”杀死，或者被距离限制死。（利用距离杀死光子的光路是有特殊的效果可以做的，但是这里用“俄罗斯轮盘赌”杀死光路仅仅是为了节约复杂度）

1.2.2 半径/色彩校准

作者在文章中洋洋洒洒推了一页半近十个公式，但是色彩校准和半径校准要做的事情很简单，动机在于因为光子是逐步添加的，所以在校准的时候不能够有一个类似于传统光子贴图的辐射校准方法，对于已知所有光子进行加权校准，加之为了节约空间，前面的光子在计算完贡献之后就再也没有它们的信息了，所以在这些限制下面该怎么校准呢？假设原有的光子数量为 n ，每次新加一组 m 个光子的时候，要保证邻域里面光子的密度不变（即单位面积内的光子数量不变），假设此时还有有 $n + \alpha m$ ($\alpha \in (0, 1)$)，因为直接控制一个领域里面的光子数量可能会很多，加上缺乏一个感性的概念，所以我们控制参数 α 。我们假设原来的半径为 r ，新的半径为 $r' = r - dr$ ，那么 $r' = gr$ ，其中 $g = \sqrt{\frac{n + \alpha m}{n + m}}$ ，同样的，在计算色彩贡献的时候也是 $\text{color}_{\text{pos}} += g \cdot \text{color}_{\text{photon}}$ 。（这段公式的合理性详见 ppt）

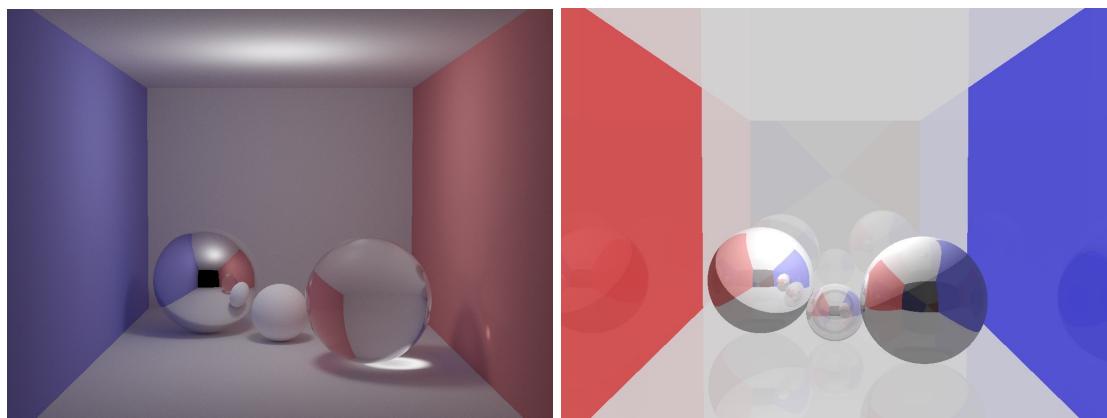
1.2.3 哈希表

怎么快速找到一个点的邻域呢？这里介绍一个廉价的做法，不需要 k-d Tree 那么繁琐的流程就可以完成的一个做法——哈希表。对于最早 Eye tracing pass 的每一个点，进行一个 R 的抖动（所有点邻域最大的初始值），把它都塞进所有抖动点所对应的哈希表里面，每次 Photon pass 的时候只要把目标点作为 Value 去哈希表里面寻址，就能找到所有可能相邻的点。当然这个做法有一定的局限性，就是渲染的空间不能很大，要把一个实数空间在精度允许的情况下，拉到整数上，这本质是一个精度损失。

2 实现效果

2.1 与光线追踪

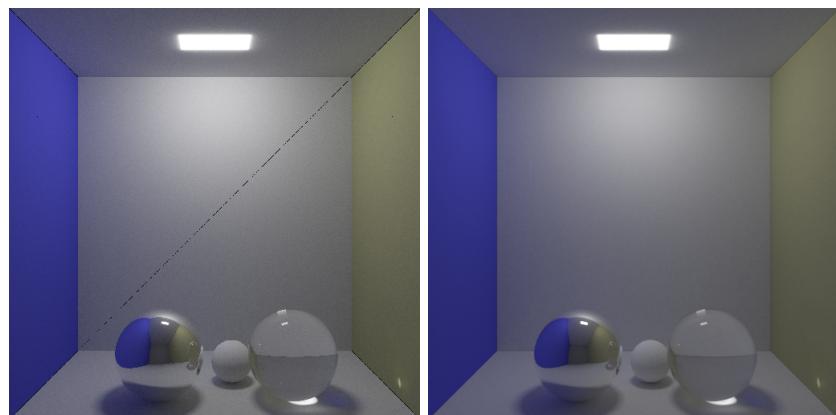
下图是作者在 PPM 里面所用的布景和我 Assignment 1 用光路追踪跑的效果对比。



不要在意为什么光线追踪的影子被不见了，主要是为了体现光线追踪在同样的光源的影响下产生光晕的性能本身就弱于 PPM，更不要说“焦散”了。显而易见的，我们可以从求上明亮的渐变看出 PPM 对于渐变的处理要优越于 Raytrace。

2.2 与我的复现

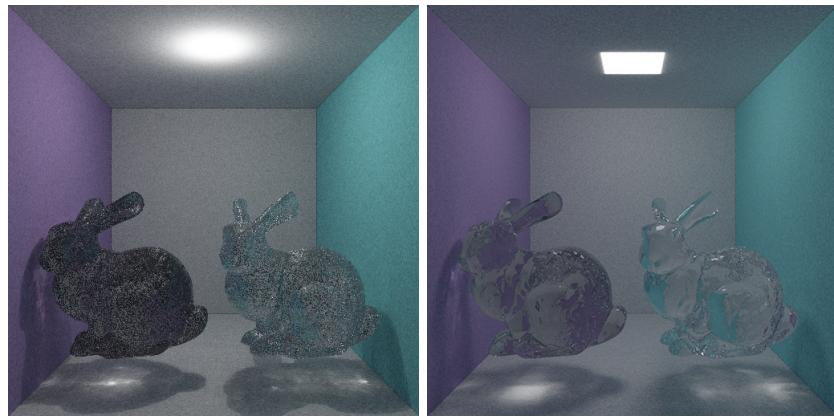
但是这个算法本身就是有其缺陷所在的一由于视线过于死板，甚至是三角面片的拼接（见下图左，是我用作者在其个人主页上放出的演示代码修改后生成的图像），都有可能因为“看不见”而产生“裂隙”：



造成问题的原因很简单，像素是一个小方块，而视线与其的交点是一点，用一点来代替一个面是极为不科学的

2.2.1 改进

事实上，上图仅仅是因为摆放的布景正对着视线造成的，但是如果我们使用一个更加复杂的几何体，就会因为几何体互相反射多次没法找到一个漫反射平面作为递归边界而发生黑点，如下图所示：

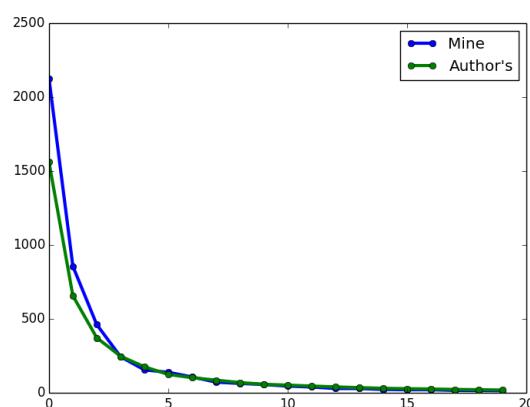


这两个问题产生的原因都是因为我们默认了像素是一个点造成的，而事实上像素是有其大小的，用一条视线与面的交点就来代表它是及其不科学的。当然解决的方法也很简单，具体做法就是，先进行一次随机发出的视线（随机的扰动限制在这一个像素的范围内），接下来释放 1000 万个光子渲染出一张图片后，重新随机一组视线，借此重新渲染一张图片（依然需要 1000 万个光子），然后对这两张图片取平均值，如是者重复数次。

2.2.2 改进评估

感性上说，显而易见的，这个改进手段对于算法的局限性有补充作用，可以修复图像中的坏点。

理性上说，量化的，改进对于性能的牺牲仅仅是在每一个新纪元开始时要重新进行一次视线追踪，而事实上视线追踪相较于大量的光子播撒来说只是一个很小的计算量（以我做的实验，视线追踪全部代价相当于逐像素进行光子追踪，一共 $512 \times 512 = 262144$ 次，相对于每个纪元播撒光子 1000 万次的追踪，仅仅只占其 2% 多一点，这样的性能损失几乎可以忽略不计。两者的图像收敛速度也相当（但是我改进后的算法图像收敛速度在前期会略慢于原算法，这是因为修复坏点需要较大的增量来进行），如下图所示：

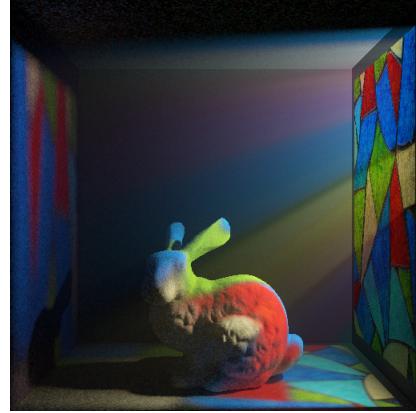


平均值可以跑出非常细腻的图像来，如下图所示（陶瓷的效果是漫反射和镜面反射体的结合，水波依赖于一个生成的三角面片）：



2.3 附加特效

在此我尝试实现了一种特效，因为作者在文中提到的是某些 Photon tracing pass 的光路会被俄罗斯轮盘赌杀死，这仅仅是为了在牺牲一小部分效果的前提下降低一点时间复杂度以提升性能。但是如果我们强制加入一个随机距离的限制的话可以做出体光的效果，如下图：



具体做法是在 Eye tracing pass 和 Photon tracing pass 的时候分别随机一个距离，根据距离加权进行色彩贡献的加权衰减。超出随机距离的点进行正常的光路追踪，而在距离以内的点，寻找周围的视线进行加权渲染。否则进行正常的渲染。