**Introduction to Physically Based Rendering(PBR) and Physically Based Ray Tracing(PBRT)**

练孙鸿

Notes of PBR 3rd Edition – chapter1

**Introduction**

渲染，是从3D场景生成图片的过程。这样的描述显然是过于宽泛的，因为还有很多种方法去完成这个任务。而基于物理(Physically Based)的方法会基于物理定律对光与物质之间的交互进行建模来模拟现实世界。虽然基于物理的方法看起来是一种非常直观的方法，但由于机器性能等原因，直到最近十来年才逐渐被广泛使用。1.7会介绍Physically Based Rendering(PBR)的简要发展历史。

这本书介绍了pbrt，也就是基于光线跟踪(RT, ray tracing)实现的基于物理的渲染系统(physically based rendering system)。

* 1. **Literate Programming**

(略)

* 1. **真实感渲染与光线跟踪算法**

真实感渲染(photo-realistic rendering)的目标是创造出一张让人难以分辨真假的3D场景的照片级图像（虽然书中也提到了“难以分辨真假”(indistinguishable)是一个不太准确的词，毕竟这个词包含着人类观察者的主观评价）。所以我们只会尽我们所能地去模拟光、光和物理的交互的物理过程，并努力给人类观察者献上品质最好的渲染图片。

几乎所有的PBR系统都是基于光线跟踪算法的。光线跟踪从原理和流程来说是比较简单易懂的：**它跟踪着光线在场景中的传播路径，并考察它是如何和周围环境交互然后反弹离开**。但无论要怎么写一个基于ray tracing的渲染系统，都起码需要模拟如下的对象或者现象：

* **摄像机、成像器(Camera)**：首先我们需要对摄像机进行建模，这决定了场景的光线会怎么被“观察者”通过成像器记录下来。非常多的渲染系统都从摄像机开始发射光线，然后跟踪到场景里面（如Path Tracer）。(pbrt 1.2.1)一般情况下我们最常用的就是针孔摄像机(pinhole camera)模型，场景物体一般会透视投影(perspective projection)到成像平面上。
* **射线与物体的相交(Ray-Object intersection)**：给定射线/光线，我们必须要知道他们在哪与物体相交，并且我们还需要交点处的一些几何信息如surface normal、材质等。大部分的ray tracing渲染系统可以测试一根射线跟多个物体的相交，一般情况下返回离射线起点最近的交点。如果我们把条射线和场景物体逐对求交的话，复杂度会是二次的，但如果我们用的代价建立加速结构之后，以后的Ray-Object intersection的复杂度可以优化成。
* **光源(Light Sources)**：光源是渲染里面非常重要的一个组成元素，如果没有了光源，渲染一个场景就变得毫无意义了。渲染系统必须对光在场景中的分布、光在空间中的能量分布进行建模。(pbrt 1.2.3)渲染系统一般需要场景中光照的几何(geometric)与辐射度学(radiometric)的分布，然后才能考察与计算到达某点的光线总能量。在渲染的时候，渲染系统会经常想要知道Ray-Object intersection交点处微元面积(differential area)接收并储存的能量。
* **可见性(Visibility)**：为了确定光在空间某一点上储存的能量是多少，我们必须要知道点到光源处是否有不被遮挡(un-interrupted)的路径。(pbrt 1.2.4)只有在光源和有不被遮挡(un-obstructed)的路径，才能被着色。只要光源还有一部分抵达点，那么它的影响就还需要考虑进去。
* **表面散射(Surface Scattering)**：每个物体都需要有一个表面的几何描述，无论是用解析形式还是网格描述；然后我们还需要知道光线与物体表面交互的方式（如反射、折射、散射），以及被再次辐射(re-radiated)(或者说被散射的)的光线的属性。其中[1]散射(scatter)广义上的含义是：某种形式的辐射（如光、声、运动粒子）因为空间中一些局部的突变而被迫偏离原来直线传播的轨道，从而向其他一个或多个方向继续传播。而日常生活当中说的“光线散射”现象如[2]丁达尔效应（Tynall Effect）则是散射的一种例子。(pbrt 1.2.5)渲染系统需要知道Ray-Object intersection交点处的材质(material)，一般用双向反射分布函数BRDF(Bi-directional Reflectance Distrution Function)来描述。BRDF是渲染方程的积分里面的一项，用来描述入射到物体在某点处**、**入射方向为的光线经过反射/散射在出射方向上的能量，一般记为。那么计算反射回摄像机的光量伪代码就是：

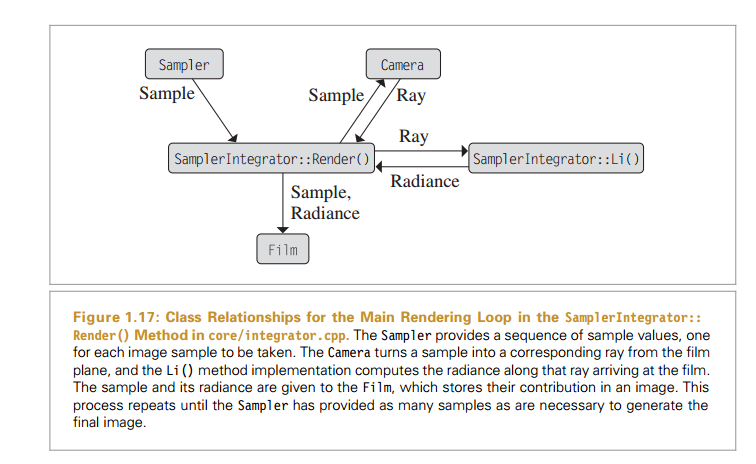
|  |
| --- |
| For each light:  If light is not blocked:  Incident\_light = light.***L***()  Amount\_reflected = surface.BRDF(***, ,* )**  ***L***+= Amount\_reflected \* Incident\_light |

* **间接光传输(indirect light transport)**：从物体表面反弹出射的光线也可能会落在其它物体的表面上，这些光线也要考虑到。(pbrt 1.2.6) [Whitted80]指出用ray tracing求解global illumination问题的递归(recursive)特性。而我们随时都可以向下递归并加上indirect light的影响。（然后书中稍微提了一下light transport equation/the rendering equation）
* **光线/射线传播(Ray Propagation)**：如果光线穿过的是真空，那么它在直线传播路径上的能量不会有变化。虽然真空在地球上并不常见，但是作为效率和效果的平衡，一般渲染就当光线是在真空中传播。在一些更加复杂精巧的渲染模型里面，光线可以穿过雾、烟、云、地球大气层等非真空介质（可能会产生散射与能量损失）。这就是体渲染(Volume Rendering)的由来，体渲染需要用数值方法求解体光照传输方程(volumn light transport equation)，这个会在pbrt chap11和chap15继续讲。
  1. **Pbrt-v3系统概览**

pbrt 3rd edition 在讲解ray tracing render system的时候会放上很多C style pseudo-code，这本书似乎也有配套的的开源系统pbrt-v3，可以在[4]github上下载源码。在它的设计里面，大概如下的基础抽象类：

|  |  |  |
| --- | --- | --- |
| **抽象接口类** | **pbrt章节** | **备注** |
| Shape | 3 | 基本的图形接口，有诸如求图形面积等的接口函数 |
| Aggregate | 4 | 加速结构 |
| Camera | 6 | 针孔摄像机模型，Pinhole Camera. Noise3D已封装 |
| Sampler | 7 | 采样器，提供对各种对象的各种采样（估计还有低差异化序列之类的东西） |
| Filter | 7 | 滤波器，应该就是Anti-aliasing、信号重建的那个滤波器 |
| Material | 9 | 材质，虽然Noise3D已经有封装，但是reflectance model还是Blinn-Phong的，估计要搞搞Physically Based一点的光照模型。 |
| Texture | 10 | 纹理，Noise3D已封装。或许可以给Texture类弄个只加载到系统内存的选项，毕竟不一定要实现GPU-acceleration |
| Medium | 11 | 用于体渲染(volume rendering)的介质，但暂时不打算实现体渲染 |
| Light | 12 | 光源，需要实现。Noise3D有用于real time rendering的精确光源，那么GI render system就需要区域光源了 |
| Integrator | 1 | 据说是必须实现void Render(Scene&)…这设计不是很懂，打算先抛弃一下。（之所以叫积分器就是因为它在解渲染方程的积分吧…） |

* 主渲染循环：在场景和渲染器都初始化了以后，就进入了pbrt-v3的主渲染循环，也是离线渲染图片耗费时间的主体。在渲染里面首先先把Camera的成像传感器矩阵（像素矩阵）分块，**分成很多的image-tile并行渲染**（p.s：用cpu多线程、OpenMP、gpu来做并行化parallelization都可以，总之path tracer在render main function的入口处看起来就挺适合做并行化的）。



图：pbrt-v3的渲染大致流程，虽然设计不是很想抄，但流程大致就是生成采样—发射射线—求解交点反射到camera的radiance—取得radiance记录在渲染图片上

至于image tile的大小，就要权衡下负载平衡(load balance)和每一块的额外开销(per-tile overhead)。出于简单起见，pbrt-v3采用了16x16的分块作为并行处理的单元。但是还要注意内存的问题，一次过分配太多内存很可能会把系统的allocator搞爆，所以可能会需要频繁地分配和回收内存块（或者复用一下也行）

Pbrt 3rd edition原书的chapter 14, 15讲light transport，包含了很多不同精度不同原理的“积分器”(integrator)（也就是不同的渲染方法）。比较简单的一种就是***Whitted Ray Tracing***：它递归地沿着反射和折射的方向去求解radiance。第一步是求相机发射的射线与场景的交点（然后调用像DXR一样的hit shader），如果没有交点那么便做出相应处理（像DXR的miss shader）。如果有有效的交点，我们就要知道这个物体表面的点是如何散射(scatter)光线的，通常用[5]**双向散射分布函数(BSDF, Bi-directional Scattering Distribution Function)**来表示。BSDF一般包括BRDF(Reflectance)与BTDF(Transmittance)。如果点有自发光(emissive)，那么积分器也会把自发光考虑上。对于每个光源，积分器会把光源发射并落在点的radiance进行求和，但如何生成采样方向还需要进一步的讨论，因为[6]辐射亮度(radiance)的定义是：

而辐射照度(irradiance)的定义是：

且radiance就是某一方向（即单位立体角）上的irradiance，所以irradiance也可以定义为radiance在半球面上的积分：

笔者的疑惑就在这里，如果用采样的方式来求解光源落在点上的光线radiance的积分，那么每个采样所代表的立体角应该是要斟酌一下的。不然本应该在点出生成半球面上的均匀、无偏采样的方向向量来采样radiance。Pbrt 3rd edition的说法是shaded point生成采样区域灯光(area light)的采样时它会同时返回一个**概率密度(probability density)**，这样子才能正确地求解 对radiance 的蒙特卡洛积分(Monte-Carlo Integration)。（蒙特卡洛积分把一个积分转化成为离散的求数学期望的问题，for more info plz refer to [7]SH Lighting notes）。至于怎么求area light采样的probability density就要看看pbrt 3rd editon第13、14章。

对于每个求交得到的着色点，我们已经用可以用Monte-Carlo Integration求得它的直接光照(direct lighting)的结果了，无论光源是精确的平行光、点光源、聚光灯，还是带有面积的区域光源、自发光物体。Whitted Ray Tracing最后还会处理完美镜面和透明体，如镜子、玻璃，也就是光线经过镜面反射或者无损失地透射射到shaded point上的光线也会被考虑到。

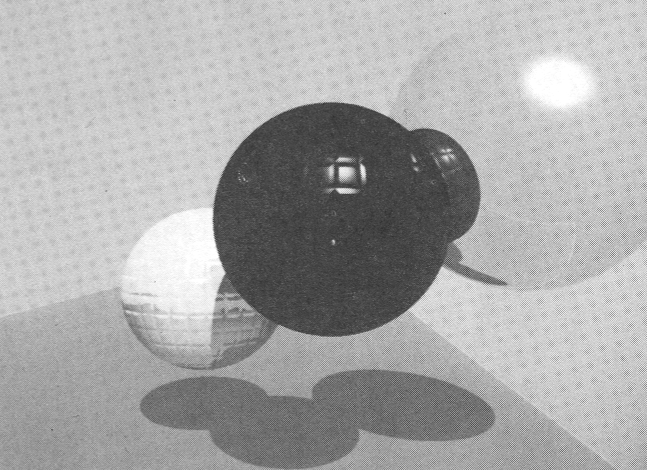
**1.4 并行化(Parallelization)**

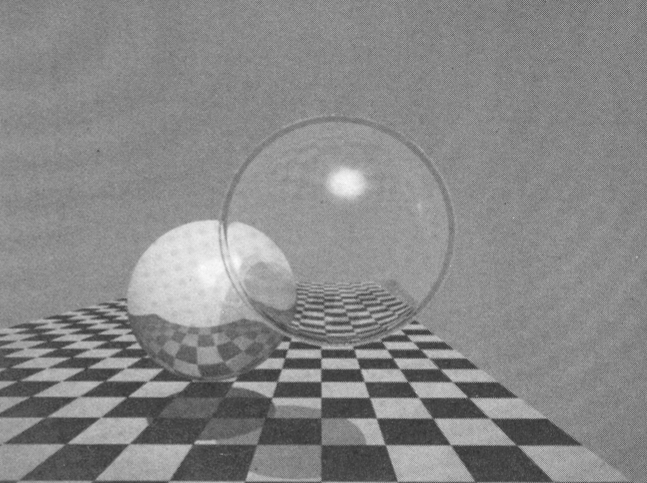
(略)，Pbrt-v3看起来只是用了CPU多核并行，没有用GPU/GPGPU来做加速。这一章简单讲了下c++和STL的多线程及线程安全，先不看了。

**1.7 A Brief History of Physically Based Rendering**

在图形学发展的早期，也就是70年代，那个时候大部分主要研究的问题都是非常基础的，例如可视性(Visibility)和物体的几何表示形式。因为在那个时候受限于机器性能，计算性能和内存都是极其珍贵的资源，而不是像现在(2019)连手机都是多核的，电脑内存8GB、16GB都是标配了。随着电脑性能的大幅增长，研究者和开发者们终于可以采用基于物理的方法进行渲染。正如Blinn所说：“随着技术的进步，我们渲染的时间还是在维持恒定”。当有更充足的机器性能可用时，内容制作者倾向于用同样的时间渲染出更加复杂、精美的场景，而不是把老场景渲染得更快。

在学术界，基于物理的渲染方法从20世纪80年代开始被重视起来并严肃地研究。[Whitted80]提出用光线跟踪的方法来模拟全局光照(Global Lighting)的效果，这篇论文打开了精确模拟光照的大门，paper里面的实验结果渲染图跟以往的差别很大，研究者对此方法表示了巨大的兴趣。

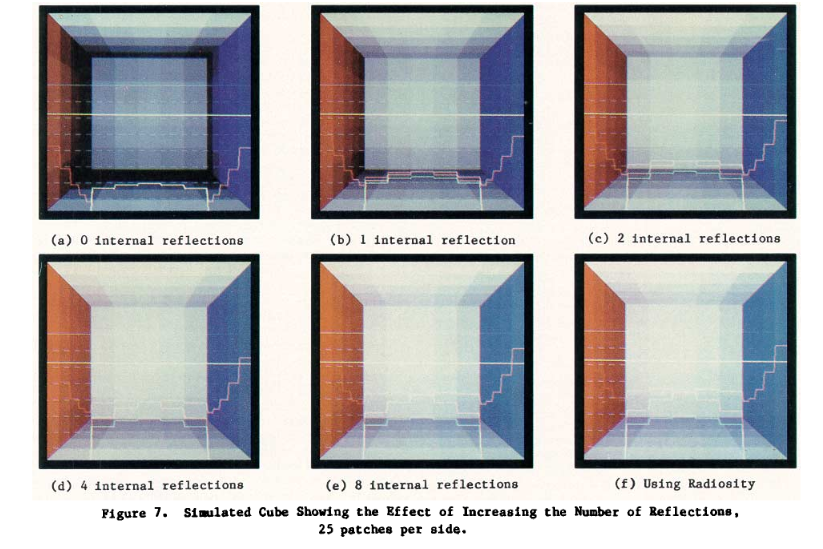
****



另外一个早期的PBR领域的关键进步是[Cook-Torrance81] Cook-Torrance的反射模型，引入了微表面(Microfacet)反射模型到图形学里面。他们还在这篇论文里面说明了如果精确模拟microfacet模型，可以渲染金属表面。



接着在1984年，[Goral84]把热传递(thermal transfer)的专著和渲染联系在一起，用近似的光照传递来实现全局的漫反射光照效果。这个方法基于有限元方法(FEM, Finte Element Method)，场景中的区域在互相的交换能量直到达到平衡。这个方法也被称作**光能传递(Radiosity)**。



虽然光能传递的方法是基于物理单元(physical unit)和能量守恒(energy conservation)的，但它完全不能做到在较短的时间内渲染，因为它的渐近时间复杂度是。

在光能传递研究很火热的时候，有一些学者在研究基于光线跟踪和蒙特卡洛积分方法的PBR。但因为蒙特卡洛积分的原因，产生的图像有不少噪点，不少人用怀疑的眼光看待这一种方法。反观光能传递在场景比较简单的时候，很快就可以渲染出效果不错的图片。

在1984年，[10]Cook, Porter, Carpenter引入了Distributive Ray Tracing，把[Whitted80]的方法泛化了一下，去除了只能跟踪单条光线的限制，避免了早期ray tracing方法渲染出的“尖锐(sharp)”的效果，使得它可以用了计算半影/软阴影(penumbra)、动态模糊、失焦模糊、模糊表面的反射、区域光源的光照等。他们证明ray tracing方法有能力模拟不少重要的光照效果。

1986年，Kajiya提出著名的**渲染方程(The Rendering Equation)**，严格地给出了渲染问题的数学形式（光照传递的积分方程）。这篇文章用了数学物理的一些观点去看待渲染问题，例如对渲染方程进行neumann series expansion，给出了一些较高视角的观点。最后提出了**路径跟踪(Path Tracing)**方法。

Cook和Kajiya的工作都让人们看到了基于物理的方法的重要性，因为他们的方法都渲染出了之前从未有过的图片效果。接下来的一些年都是关于Monte Carlo方法的研究。例如Kirk和Arvo在1991年提出[Kirk-Arvo91]的用于图像合成(image synthesis)的无偏采样(unbiased sample)方法。[Shirley96]提出的直接光照的蒙特卡洛方法也是一个重要的进展。

时间到了1997年，PBR又有重大的进展，Veach在他的phd论文[Veach97]里面描述多种关键的算法（这些算法在之前也有部分发过SIGGRAPH）。例如多重重要性采样(Multiple Importance Sampling)、双向路径跟踪(Bidirectional path tracing)、Metropolis Light Transport等。这些算法都极大地提高了PBR的计算效率。

21世纪初，电脑变得更快，也更支持并行化了，于是学者们开始追逐实时光线跟踪(realtime ray tracing)了。[Wald01]写了一篇很有影响力的论文，他们的ray tracer的优化很好，比之前的ray tracer效率都要高很多。虽然它的渲染器不是physically based的，但是这篇论文在加速结构方面的贡献还是很好的。

再之后就是GPGPU加速的ray tracing如[16]。至此，PBR的学术研究主要发展的简要介绍就结束了。

最近这几年有一些工业界上的发展。2018年微软在Direct3D 12的基础上推出了光线跟踪API DirectX RayTracing；Nvidia也推出了支持硬件光线跟踪的显卡系列Nvidia Geforce RTX系列。

**Reference**

[1] Wikipedia, <https://en.wikipedia.org/wiki/Scattering> [DB/OL]2019-3

[2] Wikipedia, <https://en.wikipedia.org/wiki/Tyndall_effect> [DB/OL]2019-3

[3] Whitted T. An Improved Illumination Model for Shaded Display[C] Communication of the ACM, Volume 23 Issue 6, June 1980 Pages 343-349

[4] <https://github.com/mmp/pbrt-v3>

[5] Bartell, F. O.; Dereniak, E. L.; Wolfe, W. L. (1980). ["The theory and measurement of bidirectional reflectance distribution function (BRDF) and bidirectional transmittance distribution function (BTDF)"](http://proceedings.spiedigitallibrary.org/proceeding.aspx?articleid=1230783). Radiation Scattering in Optical Systems. **0257**. Proceedings of SPIE Vol. 257 Radiation Scattering in Optical Systems: 154–160. [doi](https://en.wikipedia.org/wiki/Digital_object_identifier):[10.1117/12.959611](https://doi.org/10.1117%2F12.959611)

[6] ["Thermal insulation — Heat transfer by radiation — Physical quantities and definitions"](http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=16943). ISO 9288:1989. [ISO](https://en.wikipedia.org/wiki/International_Organization_for_Standardization) catalogue. 1989.

[7] 鸡哥, https://zhuanlan.zhihu.com/p/49746076 [DB/OL]2019-3

[8] Cook R. L., K.E. Torrance. A reflectance model for computer graphics [C] Computer Graphics(SIGGRAPH ’81 Proceedings), 18, 137-145

[9] Goral C M , Torrance K E , Greenberg D P , et al. Modelling the Interaction of Light Between Diffuse Surfaces[J]. Acm Siggraph Computer Graphics, 1984, 18(3):213-222.

[10] Cook R L, Porter T, Carpenter L. Distributed ray tracing[J]. Acm Siggraph Computer Graphics, 1984, 18(3):137-145.

[11] Kajiya J T. The rendering equation[J]. Acm Computer Graphics, 1986, 20(4):143-150.

[12] Kirk D, Arvo J. Unbiased sampling techniques for image synthesis[J]. Acm Siggraph Computer Graphics, 1991, 25(4):153-156.

[13] Shirley P, Wang C, Zimmerman K. Monte Carlo techniques for direct lighting calculations[J]. Acm Transactions on Graphics, 1996, 15(1):1-36.

[14] Veach E., L. Robust Monte Carlo methods for light transport simulation.[D] Standford University, 1997. 1-432

[15] Wald I., P. Slusallek, C. Benthin. Interactive distributed ray tracing of highly complex models.[C]Rendering Techniques, 2001, 277-288

[16] Wang R , Wang R , Zhou K , et al. An efficient GPU-based approach for interactive global illumination[J]. Acm Transactions on Graphics, 2009, 28(3):1-8.

[17] NVIDIA, <https://www.nvidia.com/en-us/geforce/20-series/rtx/> [DB/OL]2019-3