PBRT-从零到完全吃透系列

Dezeming Family

2021 年 5 月 4 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（用户注册后即可免费下载）。本系列开始于 2020 年 11 月，2022 年 7 月，在我开始着手讲解专业知识理论系列与高级积分器系列开始前，我决定整体重新再版，并给每本小书都附带源码。

本书目标：本书是一个引子，告诉读者本系列书的作用和内容。

一　本系列书的基本介绍

研究图形学的人们常说，PBRT 是渲染“圣经”，将 PBRT 里面的内容掌握，意味着您在渲染领域可以算是登堂入室了。但是对于新手来说，PBRT 实在是有些难以学习。本身架构就比较庞大，各个基类之间相互影响，很难控制。基于此，我决定写这套书，教大家怎么一点一点学习 PBRT，并搭建自己的渲染引擎。

学习 PBRT 最困难的地方在于，当你看书 [1] 学习时，总会看完一章就忘了上一章讲的内容。一上来 [1] 就会告诉你一堆类和变量，学习者不清楚它们之间的关系和作用，很难掌握。因此，我们在学习和构建自己的基于 PBRT 的渲染器时，会将当前用到的东西抽离出来，然后不断改造和改编，慢慢形成一个复杂而功能齐全的系统。读者跟着一步一步走，就能真正吃透整个 PBRT 渲染器。

本系列书直接从 PBRT 第三代开始介绍，前两代太老了功能也不完善。大家不需要担心第三代是否会学不会，我可以保证的是，学习 PBRT3 并不会比大家大学时学习微积分要难。2021 年夏季据说 PBRT4 就要登场了，我觉得它和第三代应该没什么较大的区别，之后我会写一本书，用来介绍第三代到第四代的改变。

每本书都有前言，每个章节也都有前言，大家在学习每本书时，一定要先看前言中的叙述，例如每本书一定要学会什么，以及某些部分可以不用现在就弄懂，这样就能减轻学习压力。2022 年 7 月再版中，我们会每本书都提供源码，源码的内容一定要掌握好。

对于一些理论知识的介绍，虽然为了完整性在本系列书中也包含了原理与实现，但是大家要注意参考 [1][5] 和源码中的实现。同时对于一些基本概念，如果本系列书中讲述不明确，可以借助 [5] 和网络资料自行学会，也可以在本书网站上留言。

顺便提一句，PBRT 源码是会不断变化的，在 PBRT 书 [1] 发布以后 PBRT3 也会有一些新的内容，所以很可能您从书中看到的内容与源码不符，因此这个时候就需要您能够根据自己的知识和理解去判断。我写本系列书的源码是来自 2020 年 10 月的版本，在某些地方已经和书中的介绍不同了（其实都是些一眼就能看出来的地方，我也会在系列书里进行介绍）。

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。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！

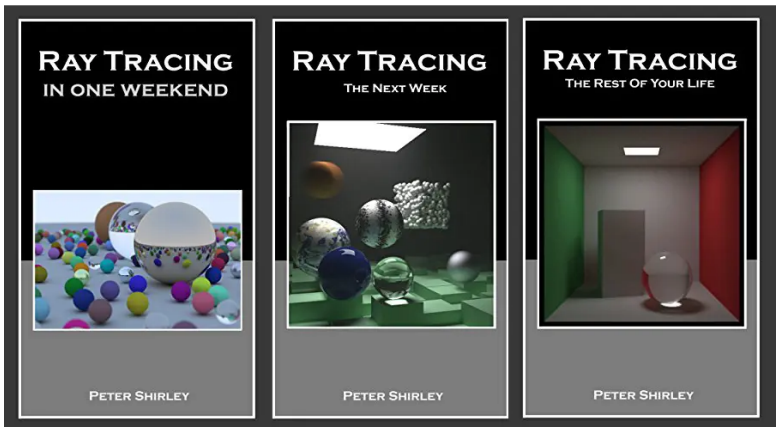
第二部分为专业理论知识部分《专业知识理论与代码实战》，一共 12 本书，暂定售价 120 元，每本价格也各不相同。我于 2022 年 7 月，开始着手写这个系列，但在此之前，我会将第一部分的每本书都配备源码，然后再开始第二个部分的写作。源码见网址 [https://github.com/feimos32/PBRT3-DezemingFamily]。

二　PBRT 基本知识

2 1　《PBRT 系列 1-文件加载和设定》

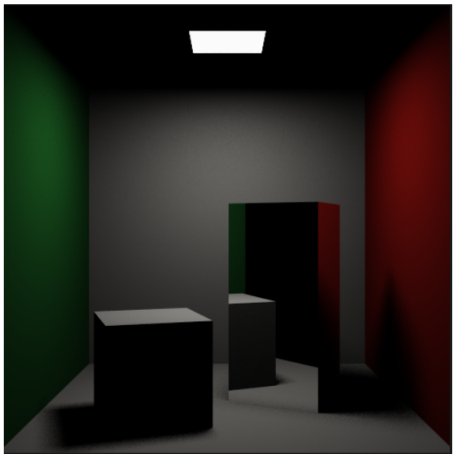
本系列书的第一本《PBRT 系列 1-文件加载和设定》，会首先教你怎么看懂 PBRT 使用的场景文件，然后告诉你场景文件是如何加载到 PBRT 各个类中，然后程序如何启动渲染，弄明白这些，你就对整个引擎有了初步的把握。

在下一步学习之前，希望读者已经自己实现过微型的光线追踪引擎。如果没有实现过，则将下面三本书 [2][3][4] 学习一遍以上，尤其是 [4] 的蒙特卡洛光线追踪部分，一定要认真学会：



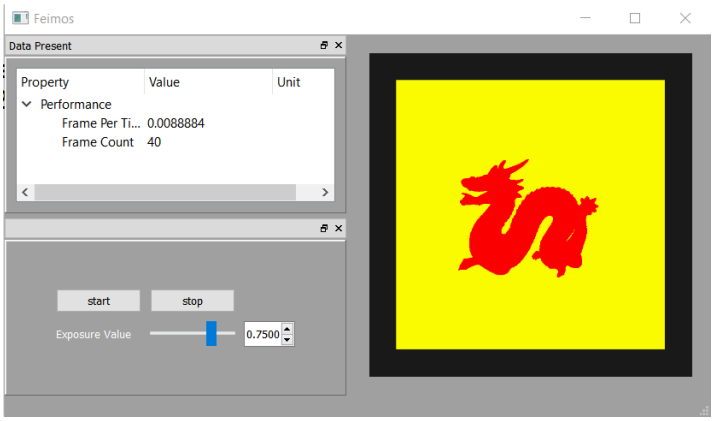
2 2　《PBRT 系列 2-代码实战-Whitted 光线追踪引擎》

我们的第二本书《PBRT 系列 2-代码实战-Whitted 光线追踪引擎》，就是基于 [2][3][4] 中实现的渲染器来学习的。我们首先研究 PBRT 源码中，最简单的光线追踪器——Whitted 光线追踪器的渲染流程，顺便学习了 PBRT 里面的一些接口和类，之后在我们自己的引擎上实现一个简单的 Whitted 光线追踪。尽管学完 [2][3][4] 后，Whitted 光线追踪器非常简单，而且并没有什么新知识，但是通过学习该追踪器，您就能对 PBRT 整个渲染流程有更深的认识。我们能够基于 PBRT 的代码风格渲染出下面的效果：



2 3　《PBRT 系列 3-代码实战-形状和加速器》

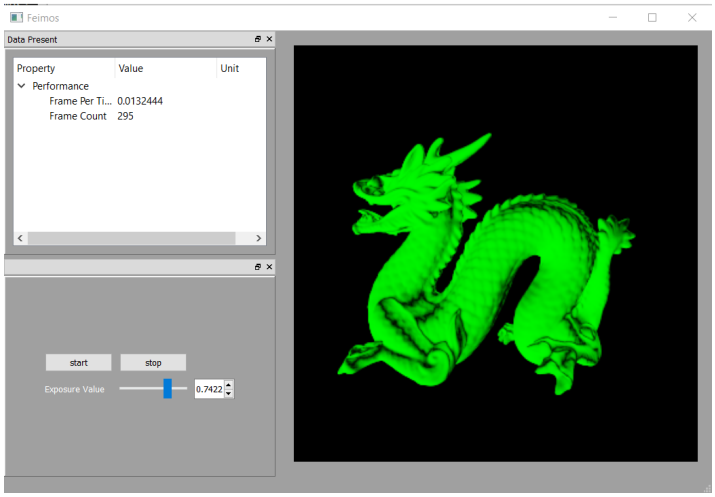
之后是我们的第三本书：《PBRT 系列 3-代码实战-形状和加速器》，第三本书则不会再使用 [2][3][4] 的渲染结构了，而是学习 PBRT3 的矩阵向量工具类、形状类和加速器包围盒。之后，将它们实现在我们自己的光线追踪器上。因为我们只会显示一下图形，所以不需要光线追踪结构，因此您完全可以建立一个新的工程开始移植。



从第三本书开始，我们的所有工具和类都是基于 PBRT 构建的，一开始我们的引擎没有材料纹理类，没有颜色与光谱类，没有相机类等，所以我们从 [2][3][4] 中实现的材质类、光源类纹理类等都可以完全删除了。我们实现的新系统也会先去掉 PBRT 源码中与这些有关的内容，书中会告诉大家如何取舍。

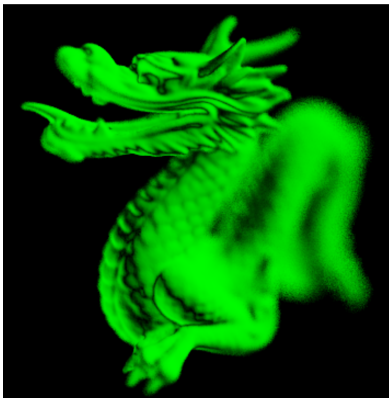
2 4 《PBRT 系列 4-代码实战-颜色与光谱》

本书学习颜色空间与光谱的表示，并移植到自己的渲染系统上。然后我们会渲染出下面的图像：



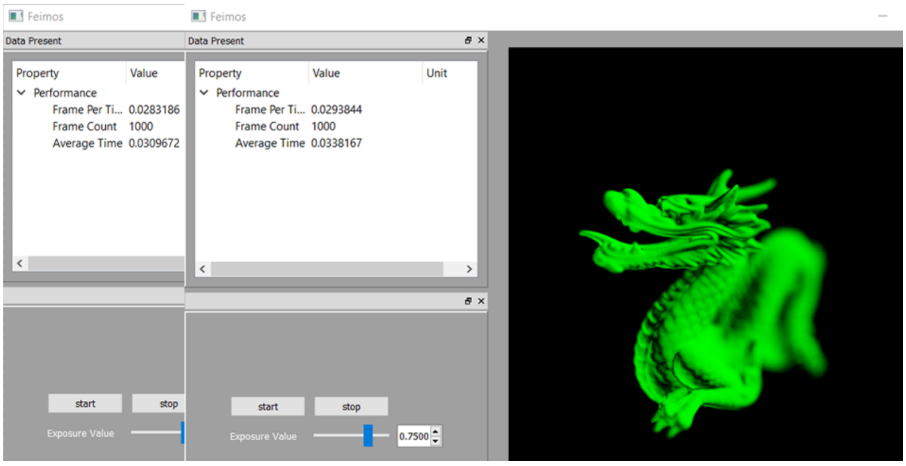
2 5 《PBRT 系列 5-代码实战-相机系统》

学习 PBRT 的相机系统，包括透视投影相机和正交投影相机。学习薄透镜模型和景深。在自己的引擎上移植 PBRT 相机并渲染出如下离焦模糊结果：



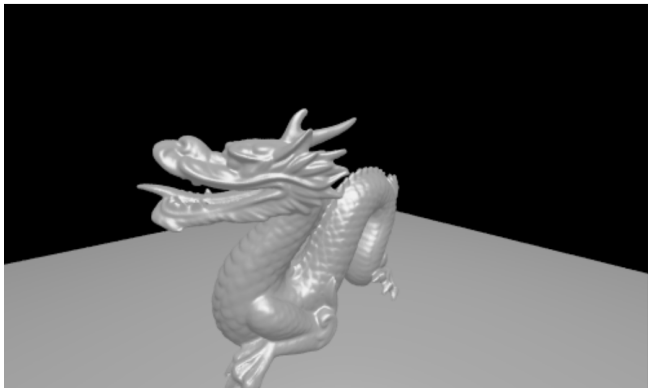
2 6 《PBRT 系列 6-代码实战-误差界定和内存管理》

学习 PBRT 的内存管理类和误差界定类，当在 PBRT 程序中遇到时知道它是在干什么就行了。如果想移植可以移植到自己的系统上，不想移植就算了。因为本书没有新渲染的东西，所以随便渲染一张使用误差管理和不使用误差管理的时间对比图：



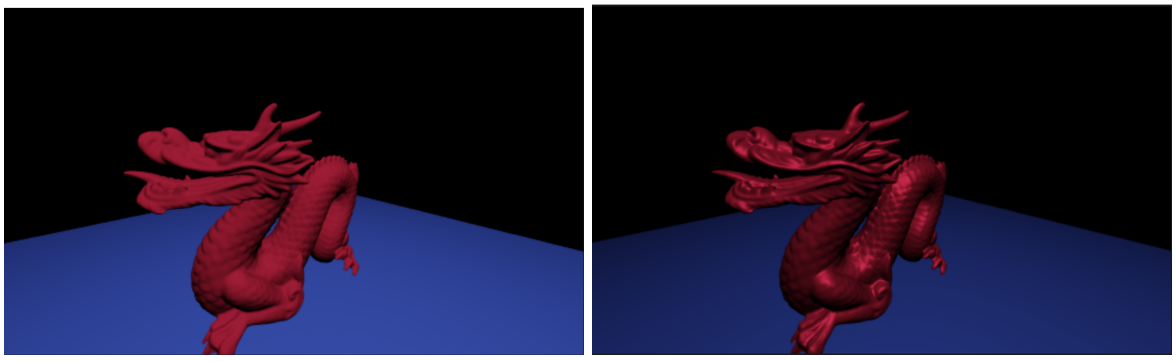
2 7 《PBRT 系列 7-代码实战-采样器和渲染器》

学习 PBRT 的采样器。明白使用专门的采样器的好处，移植 PBRT 的采样器到我们的系统中。明白采样器的使用流程。学习渲染积分器的几个基本接口，移植渲染积分器到我们自己的系统上，渲染出下面的图像：



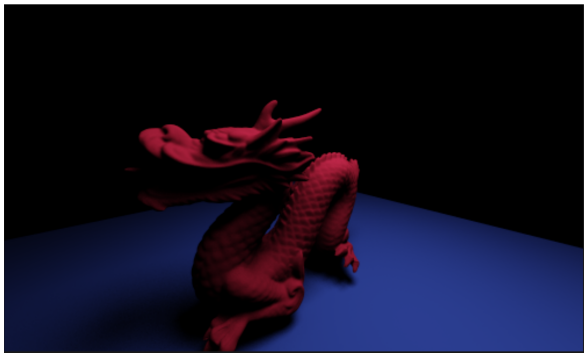
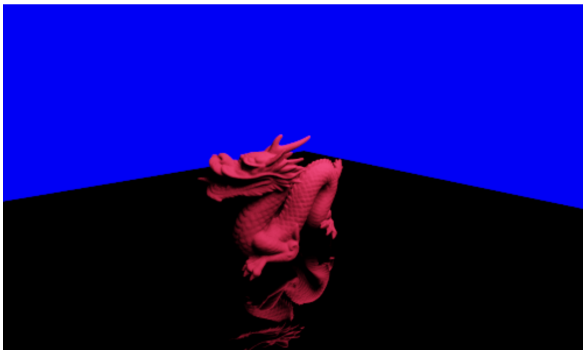
2 8 《PBRT 系列 8-代码实战-反射与材质初步了解》

学习 PBRT 的反射模型以及最基本的朗伯反射（Lambertian）。学习材料类的基本接口。学习纹理类基本接口，实现最简单的常量纹理类。实现整个包含了材质和纹理的渲染流程，并渲染出下面的图像（朗伯反射和加入高光的朗伯反射）：



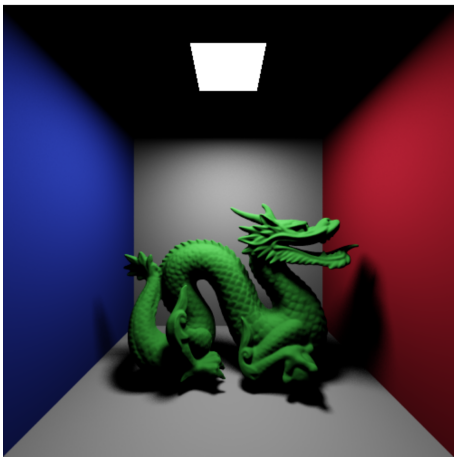
2 9 《PBRT 系列 9-代码实战-灯光基础与完整的光线追踪器》

学习 PBRT 的光源系统，移植点光源和面光源到自己的引擎上。学习和移植 PBRT 的完美镜面物体。实现 Whitted 光线追踪器到我们的系统中。渲染出下面的图像（面光源软阴影和镜面反射）：

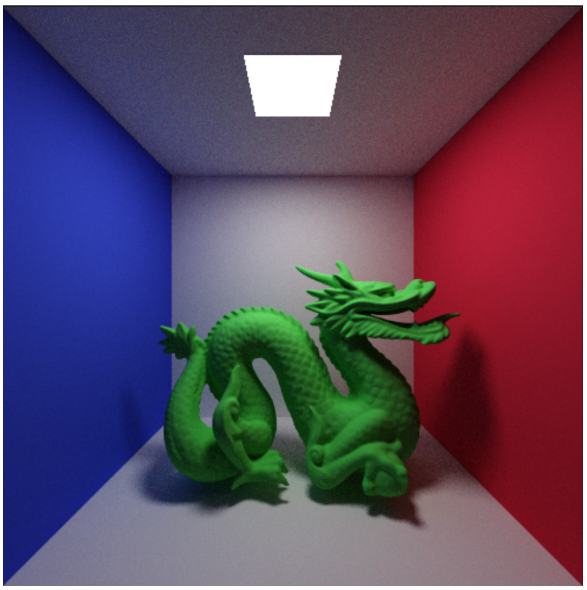
2 10 《PBRT 系列 10-代码实战-一些零散和琐碎的内容补充》

学习法向量和坐标系统的反转。移植和实现完美镜面的透明材质。学习采样 BSDF，显示面光源。自己创建一个简单的环境光源。实现一个康奈尔盒：

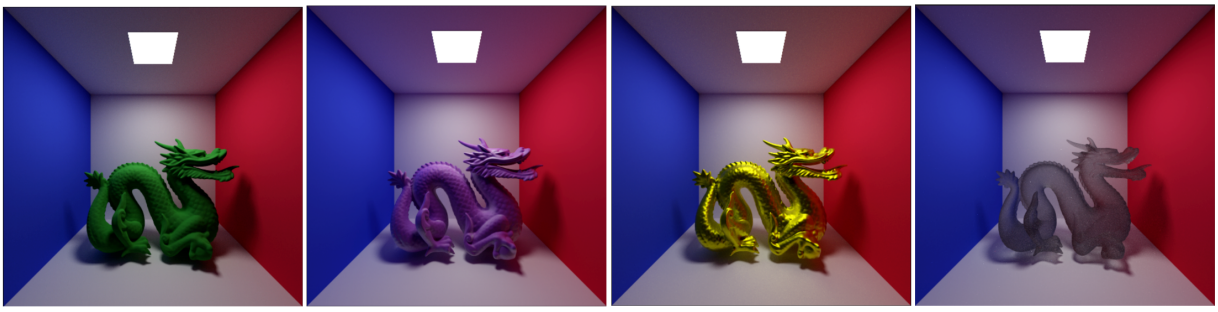
2 11 《PBRT 系列 11-代码实战-路径追踪》

学习光传输方程，路径积分形式。俄罗斯轮盘与路径追踪算法。学习直接采样光和多重重要性采样。实现直接采样光积分器和路径追踪积分器。渲染出下面的效果：



2 12 《PBRT 系列 12-代码实战-微表面材质》

学习多种材质的基本概念，微表面模型的基本概念。移植和测试成功微表面模型。跟踪内存。检查内存释放问题，保证没有内存泄漏。



2 13 《PBRT 系列 13-代码实战-无限面光源》

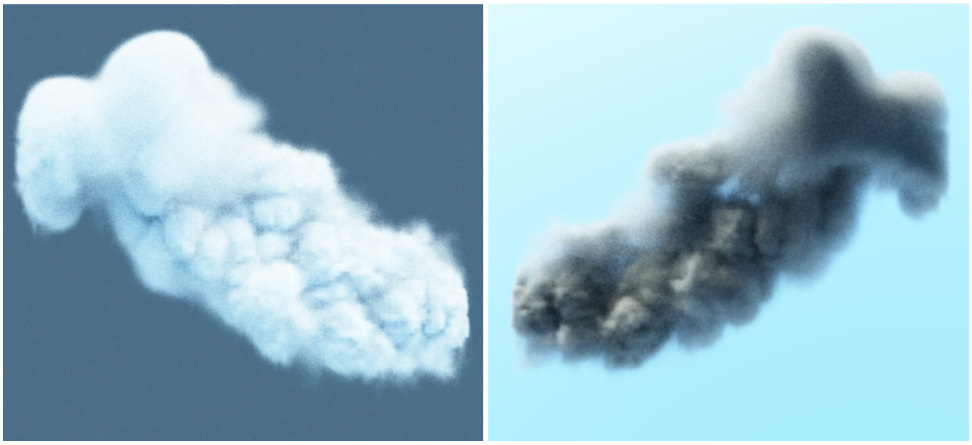
学习 PBRT 中的 MIPMap 的原理和使用方法。移植和测试成功 MIPMap。学习无限面光源的表示，移植并构造场景测试，得到下图渲染效果：

2 14 《PBRT 系列 14-代码实战-光线微分与纹理》

学习 PBRT 中的光线微分技术，实现光线微分并对比效果。实现 PBRT 的图像纹理，渲染得到如下效果：

2 15 《PBRT 系列 15-代码实战-参与介质与体渲染》

学习 PBRT 中的体渲染原理，实现两种不同的参与介质类型，渲染得到如下效果：



2 16 《PBRT 系列 19-系统功能扩展-复杂模型的读取接口》

学习模型读取库 Assimp 的使用，编写一个模型读取接口，渲染出下面的图像：

2 17 小结

您跟着本系列走，会不断学习到新的类和算法，这些类会不断移植和实现到我们的系统中，当您学完本系列，并最终移植实现了渲染器后，就真正掌握了主要的 PBRT 的源码架构。PBRT 遵循的是 BSD license，也就是说您可以随便使用 PBRT 中的源码应用到自己的工程中，但是要保留源码中的 BSD 协议，并且不能用 PBRT 的名字来做宣传。

《基础理论与代码实战》部分于 2022 年 7 月开始重新再版，我们会给每本系列书提供源码。源码见网址 [https://github.com/feimos32/PBRT3-DezemingFamily]。为的是让有疑惑的读者能够更清楚地掌握每本系列书的内容，以及后面讲解高级积分器系列的时候更方便以及更好叙述。

三　PBRT 专业理论知识

本节只是专业理论知识中的目录。

四 PBRT 的设计和历史

因为 PBRT 也不是我写的，所以关于它的历史我也只能参考网络和 [1]。

4 1 关于 PBRT 设计的简单介绍

下面的内容摘抄自 [1]。

pbrt 设计中的一个基本假设是渲染的有趣的图像类型是具有复杂几何体和照明的，支持各种形状、材料、光源和光传输算法非常重要。PBRT 还假设使用良好的采样模式、光线微分（虽然也有差分的意思，但通常情况下“差分”只表示离散情况，而“微分”表示的更广泛一点，因此这里我统称为光线微分）和抗锯齿纹理很好地渲染这些图像是值得的。这些假设的一个结果是，pbrt 在渲染简单场景时效率相对较低，而更专业的系统可以做得更好。

例如，设计优先级的一个性能含义是，在光线相交处查找 BSDF 比在渲染器中查找 BSDF 的计算成本更高，渲染器不需要花费大量精力过滤纹理和计算光线微分。通过减少跟踪更多摄影机光线以解决纹理混叠的需要，这种努力总体上是有回报的，尽管对于简单场景，纹理混叠通常不是问题。另一方面，pbrt 中的大多数渲染积分器都假设在每个像素中采集数百甚至数千个样本，以获得高质量的全局照明；在这种情况下，高质量滤波的好处会减少，因为高像素采样率最终也会以高速率采样纹理。

为电影呈现高质量的图像带来了许多挑战，这些挑战超出了 PBRT 所讨论的主题。能够以几何和纹理的复杂性渲染高度复杂的场景是一个要求，大多数产品级渲染器都延迟了纹理和几何体的加载和缓存，这是它们实现的核心。可编程曲面着色器对于允许用户指定复杂的材质外观也很重要。

另一个实际挑战是集成交互式建模和着色工具：艺术家能够快速看到他们对模型、曲面和灯光所做更改的效果非常重要。与工具的深度集成是必要的，而像在 pbrt 中所做的那样，每次渲染场景时用文本文件从零开始使用场景描述就不是一种可行的方法。

由 Parker 等人（2010）描述的 OptiX 光线跟踪系统有一个非常有趣的系统结构：它是一个内置功能的组合（例如，用于构建加速结构和穿过它们的光线），可以通过用户提供的代码（用于基本实现、表面着色功能等）进行扩展。多年来，许多渲染器都允许用户进行这种扩展，通常是通过某种插件体系结构。OptiX 的独特之处在于它是使用运行时编译系统构建的，该系统将所有这些代码编译在一起。因为编译器在生成代码时具有整个系统的视图，所以生成的自定义呈现器可以以多种方式自动专用化。例如，如果表面着色代码从不使用（u，v）纹理坐标，则在三角形形状相交测试中计算它们的代码可以作为死代码进行优化。或者，如果 Ray 的 t 变量从未被访问，那么设置它的代码甚至结构成员本身都可以被删除。因此，这种方法允许一定程度的专业化（以及由此产生的性能），这将很难手动实现，至少对于一个以上的系统变量来说是如此。

PBRT 的重点一直是传统的多核 cpu 作为系统的目标，此外忽略了通过使用 CPU SIMD 硬件，每个指令最多可以执行八个浮点操作的可能性。虽然其他计算架构，如 GPU 或专用光线追踪硬件是渲染器值得考虑的目标，但它们的特性往往会迅速改变，并且它们的编程语言和模型比 CPU 上的 C++ 语言更不被广泛了解。虽然 PBRT 没有用 pbrt 针对这些体系结构，但是讨论它们的特性是有用的。

由于多核 cpu 在单个芯片上提供了少量独立的延迟处理器），从 2003 年前后可编程图形处理器的出现开始，吞吐量处理器（throughput processors）（以 gpu 为例）已日益成为许多计算机系统中可用的大部分计算能力的来源。这些处理器关注的不是单线程性能，而是以高聚合计算吞吐量高效地并行运行数百或数千个计算，而不试图最小化任何单个计算的时间。

不关注单线程性能，吞吐量处理器（throughput processors）能够在芯片上为缓存、分支预测硬件、无序执行单元和其他为提高 CPU 单线程性能而发明的功能占用更少的空间。因此，给定固定数量的芯片面积，这些处理器能够提供比 CPU 多得多的算术逻辑单元（ALUs）。对于能够提供大量独立并行工作的计算类型，吞吐量处理器可以使这些 ALUs 保持忙碌，并且非常高效地执行计算。截至本文撰写之时，gpu 提供的峰值触发器大约是高端 CPU 的十倍；这使得它们对于许多处理密集型任务（包括光线跟踪）非常有吸引力。

单指令多数据（SIMD）处理，即处理单元跨多个数据元素执行一条指令，是吞吐量处理器用于高效交付计算的关键机制；今天的 CPU 和 GPU 的处理核心都有 SIMD 向量单元。现代 CPU 通常有几个处

理核心，并在其向量指令集中支持四个或八个 32 位浮点运算（例如，SSE、NEON 或 AVX）。GPU 目前有几十个处理核，每个核有两个 SIMD 矢量单元，宽度在 8 到 64 个元素之间。（Intel 的 Xeon Phi 体系结构具有 50 多个相对简单的 CPU 内核，每个内核都有一个 16 宽 32 位浮点 SIMD 单元，位于这两点之间。）很可能所有这些处理器体系结构中的处理内核的数量和向量单元的宽度都会随着时间的推移而增加，因为硬件设计者利用了摩尔定律所能提供的额外晶体管。

在渲染系统中使用吞吐量处理器的一大挑战是如何找到有效使用 SIMD 向量元素的一致计算集合。假设蒙特卡罗路径跟踪器跟踪一组光线；在第一次反弹时进行随机采样后，每条光线通常会与完全不同的对象相交，很可能与完全不同的曲面着色器相交。在这一点上，运行曲面明暗器可能会对 SIMD 硬件的利用率很低，因为每条光线都需要执行不同的计算。另一个挑战是，gpu 上相对有限的本地内存使得实现光传输算法具有挑战性，这些算法需要为每条光线提供更多的存储空间。（例如，即使为双向路径跟踪算法存储一对子路径的所有顶点也不简单。）

Hachisuka 的路径跟踪器展示了渲染器开发人员要考虑的一个有趣的权衡，它使用具有平行投影的光栅化器来跟踪光线，有效地计算所有着色点在同一方向上的可见性（Hachisuka 2005）。他的见解是，尽管这种方法不能为蒙特卡罗路径跟踪提供特别好的采样分布，但由于每个点都不能执行重要采样来选择出射方向，因此计算非常相干的光线集合的可见性所提高的效率得到了总体回报。换言之，对于固定的计算量，与使用光线跟踪相比，使用光栅化可以获取更多的样本，因此数量更多的分布不均匀的样本生成的图像比数量更少的精心选择的样本生成的图像更好。我们怀疑，在计算单个点的精确本地期望结果与计算可以在多个点上非常有效地进行全局计算之间进行权衡的一般问题，将是开发人员在未来 SIMD 处理器上要考虑的一个重要问题。

4 2 PBRT 的历史

PBRT 的想法诞生于 1999 年 10 月。在接下来的五年里，它从一个只为斯坦福 CS348b 课程的学生提供支持的系统发展到一个健壮、功能丰富、可扩展的渲染系统。从一开始就学到了很多关于如何构建一个渲染系统的知识，这个系统不仅可以生成漂亮的图片，而且其他人也喜欢使用和修改它。然而，最困难的是设计一个其他人可能喜欢阅读的大型软件。这是一项远比实现任何渲染算法本身更具挑战性（和回报）的任务。

在第一次出版之后，这本书在世界范围内的高级图形课程中得到了广泛的采用。从头开始编写光线跟踪器是一项艰巨的任务（许多本科图形课程的学生都可以证明这一点），而创建一个健壮的基于物理的渲染器更是困难得多。PBRT 降低了有抱负的研究人员进入渲染领域的门槛，使研究人员更容易尝试和展示渲染领域新思想的价值。我们仍然很高兴在 SIGGRAPH、欧洲图形渲染研讨会、高性能图形和其他图形研究场所看到论文，这些论文要么建立在 pbrt 上以实现其目标，要么将其图像与 PBRT 进行比较，作为 ground truth。

最近，在离线渲染实践中，基于物理的方法得到了迅速的采用，游戏和交互式应用程序也得到了迅速的采用，在屏幕上看到令人难以置信的图形，并惊叹于数十亿的伪随机（或准随机）样本、数十亿的光线跟踪，以及每幅经过的图像中所包含的复杂数学知识，是一件特别令人高兴的事。

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