PBRTv4-系列3-PBRT的整体理解

Dezeming Family

2023年9月7日

DezemingFamily系列文章和电子文档**全部都有免费公开的电子版**，可以很方便地进行修改和重新发布。如果您获得了DezemingFamily的系列文章，可以从我们的网站[https://dezeming.top/]找到最新的版本。对文章的内容建议和出现的错误也欢迎在网站留言。

20230906：非常感谢图灵图书的王编辑提前将PBRT v4的电子书给我做参考。

20230908：本文完成第一版。

目录

一　PBRT的设计理念

PBRT的设计遵循了不少目标。

第一个目标是完整性。完整性意味着该系统不应缺少高质量商业渲染系统中的关键功能。特别是，这意味着重要的实际问题，如抗锯齿、鲁棒性、数值精度和有效渲染复杂场景的能力，都应该得到彻底解决。从系统设计之初就考虑这些问题是很重要的，因为这些特征可能对系统的所有组件都有微妙的影响，并且在实施的后期很难改造到系统中。

第二个目标是可读性。PBRT试图谨慎选择算法、数据结构和渲染技术，并着眼于可读性和清晰度。由于它们的实现将受到比其他渲染系统更多的读者的检查，pbrt试图选择所知道的最优雅的算法，并尽可能好地实现它们。这个目标还要求系统足够小，让一个人能够完全理解。PBRT使用可扩展的体系结构，系统的核心是根据一组精心设计的接口类实现的，并且在这些接口的实现中尽可能多地提供特定功能。结果是，不需要为了理解系统的基本结构而理解所有的具体实现。这使得深入研究感兴趣的部分和跳过其他部分变得更容易，而不会忽视整个系统是如何结合在一起的。完整性和说明性这两个目标之间存在着紧张关系。实施和描述每一种可能有用的技术不仅会让这本书变得长得令人无法接受，而且会让这个系统对大多数读者来说过于复杂。在pbrt缺乏特别有用的功能的情况下，PBRT试图设计架构，以便在不改变整个系统设计的情况下添加该功能。

第三个目标是，物理正确性。基于物理的渲染的基本基础是物理定律及其数学表达式。pbrt的设计目的是使用正确的物理单位和概念来计算量和实现算法。pbrt努力计算物理正确的图像；它们准确地反映了真实世界场景中的照明。决定使用物理基础的一个优点是，它给出了程序正确性的具体标准：对于简单的场景，预期结果可以以闭式计算，如果pbrt没有计算出相同的结果，就知道实现中一定存在错误。类似地，如果pbrt中不同的基于物理的照明算法对同一场景给出不同的结果，或者如果pbrt没有给出与另一个基于物理的渲染器相同的结果，那么其中一个肯定存在错误。最后，pbrt相信这种基于物理的渲染方法是有价值的，因为它是严格的。当不清楚特定的计算应该如何执行时，物理学会给出一个保证结果一致的答案。

第四个目标是，渲染效率。效率的优先级低于这三个目标。由于渲染系统通常在生成图像的过程中运行数分钟或数小时，因此效率显然很重要。然而，pbrt大多局限于算法效率，而不是低级别的代码优化。在某些情况下，明显的微观优化会退居清晰、组织良好的代码之后，尽管确实做出了一些努力来优化系统中大部分计算发生的部分。

1 1　PBRT第一版到第二版的变化

移除了插件架构。

移除了图像处理管线。

除此之外，PBRT追求任务并行性：多核架构变得无处不在，如果没有能力扩展到本地可用核的数量，pbrt就无法保持相关性。PBRT还希望本书中记录的并行编程实现细节将帮助图形程序员理解编写可扩展并行代码的一些微妙之处和复杂性。

“生产”渲染的适当性：pbrt的第一个版本专门作为教学工具和渲染研究的基石。事实上，在准备PBRT第一版时做出了许多与生产环境中使用相反的决定，例如对基于图像的照明的有限支持，对运动模糊的不支持，以及在复杂照明情况下不稳定的光子映射实现。随着对这些功能的支持以及对次表面散射和Metropolis光传输的支持大大提高，在第二版中，pbrt变得更适合渲染复杂环境的高质量图像。

1 2　PBRT第二版到第三版的变化

增加了双向光传输算法。增加了次表面散射。增加了数值鲁棒的求交（误差界定）。参与介质更好的支持。可测量的材质（稀疏频域空间基材质）。光子映射。更全面的采样器。

1.3 是否使用光学

由可见光组成的波非常小，从波峰到波谷只有几百纳米。光的复杂波状行为出现在这些小尺度上，但在以厘米或米的尺度模拟物体时，它几乎没有影响。这是一个好消息，因为对任何大于几微米的东西进行详细的波浪级模拟都是不切实际的：如果渲染图像需要这种细节级别，那么计算机图形就不会以目前的形式存在。

pbrt将主要研究16世纪至19世纪初开发的光学方程，这些方程将光建模为沿射线传播的粒子（几何光学）。这导致了一种基于称为光线跟踪的关键操作的更高效的计算方法。

二 系统基类和pbrt书的结构

2.1 系统基类

这些关键的基类有14种，汇总在下表。将其中一种类型的新实现添加到系统中非常简单；实现必须提供所需的方法，必须将其编译并链接到可执行文件中，并且必须修改场景对象创建例程，以便在解析场景描述文件时根据需要创建对象的实例。

Base type	Source Files	Section
Spectrum	base/spectrum.h, util/spectrum.{h,cpp}	4.5
Camera	base/camera.h, cameras.{h,cpp}	5.1
Shape	base/shape.h, shapes.{h,cpp}	6.1
Primitive	cpu/{primitive,accelerators}.{h,cpp}	7.1
Sampler	base/sampler.h, samplers.{h,cpp}	8.3
Filter	base/filter.h, filters.{h,cpp}	8.8.1
BxDF	base/bxdf.h, bxdfs.{h,cpp}	9.1.2
Material	base/material.h, materials.{h,cpp}	10.5
FloatTexture SpectrumTexture	base/texture.h, textures.{h,cpp}	10.3
Medium	base/medium.h, media.{h,cpp}	11.4
Light	base/light.h, lights.{h,cpp}	12.1
LightSampler	base/lightsampler.h, lightsamplers.{h,cpp}	12.6
Integrator	cpu/integrators.{h,cpp}	1.3.3

C++中的常规实践是使用定义纯虚拟函数的抽象基类为这些类型中的每一个指定接口，并使实现从这些基类继承并实现所需的虚拟函数。反过来，编译器将负责生成调用适当方法的代码，给定指向基类类型的任何对象的指针。之前的三个版本的pbrt中都是用的这种方法。

但在这个版本中，增加了对图形处理单元（GPU）渲染的支持，这激发了一种基于标记的分派(tag-based dispatch)的更便携的方法，其中每个特定类型的实现都被分配了一个唯一的整数，该整数在运行时确定其类型。在pbrt中以这种方式实现的多态类型都在base/目录中的头文件中定义，比如material/texture/camera。不过注意，有些类是没有基类的，比如Sphere/Disk/FloatConstantTexture；有些类有基类，比如OrthographicCamera/PerspectiveCamera都是有基类的。但是它们也都使用基于标记的分派方法来管理具体的实现。

这个版本的pbrt能够在支持C++ 17的GPU上运行，并为光线交叉测试提供API。pbrt的系统经过了仔细的设计，以便几乎所有pbrt的实现都在CPU和GPU上运行，因此大部分内容通常很少谈论CPU与GPU的对比，正如书[4]的第2章到第12章所述。

pbrt中CPU和GPU渲染路径之间的主要区别在于它们的数据流以及它们的并行化方式——是如何将各个部分连接在一起的。有些基本渲染算法和高级光传输算法（双向路径追踪等）都只能在CPU上使用。GPU渲染管线在[4]的第15章中进行了讨论，尽管它也能够在CPU上运行（GPU上使用ValPath积分器，然而，不如直接以CPU为目标设计的光传输算法高效）。

2 2 pbrt书的结构

书[4]的在线版包含由于页面限制而无法包含在印刷书籍中的其他内容。所有这些材料都是对本书内容的补充。例如，它包括一个额外的相机模型、kd树加速结构的实现，以及关于双向光传输算法的完整章节。

[4]的其余部分分为四个主要部分，每个部分各有几章。

- 第2章至第4章介绍了该系统的基础。第2章简要介绍了蒙特卡洛积分的关键思想，第3章描述了广泛使用的几何类，如Point3f、Ray和Bounds3f。第4章介绍了用于测量光的物理单位以及pbrt用于表示光谱分布的SampledSpectrum类。它还讨论了颜色，即人类对光谱的感知，它影响了如何向渲染器提供输入以及如何生成输出。

- 第二部分介绍了图像的形成以及场景几何体是如何表示的。第5章定义了相机接口和一些不同的相机实现，然后讨论了将到达film的光谱辐射转换为图像的整个过程。第6章介绍了Shape界面，并给出了许多形状的实现，包括展示如何使用它们进行射线相交测试。第7章描述了加速结构的实现，这些加速结构通过跳过光线绝对不相交的基元测试来提高光线跟踪的效率。最后，第8章的主题是Sampler类，它将样本放置在图像平面上，并为蒙特卡罗积分提供随机样本。

- 第三部分是关于光以及它如何从表面和参与介质中散射。第9章包括一组类，这些类定义了表面反射的各种类型。第10章中描述的材料使用这些反射函数来实现许多不同的表面类型，如塑料、玻璃和金属。材质特性（颜色、粗糙度等）的空间变化由纹理建模，第10章也对其进行了描述。第11章介绍了描述光在参与介质中如何散射和吸收的抽象概念。第12章描述了光源的接口和各种光源实现。

- 最后一部分将书其余部分的所有想法结合在一起，实现了一些有趣的光传输算法。第13章和第14章中的积分器代表了蒙特卡罗积分的各种不同应用，以计算比RandomWalkIntegrator更精确的光传输方程近似值。第15章描述了在GPU上运行的高性能积分器的实现。

- 第16章是书的最后一章，对系统设计决策进行了简短的回顾和讨论，并对扩展项目提出了一些建议。

- 附录包含更多的蒙特卡罗采样算法，描述了实用函数，并解释了在解析输入文件时如何创建场景描述的细节。

三 渲染的执行

3 1 执行阶段

注意以前的PBRT版本是一边parse一边创建对象，但是PBRT v4是先parse完对象以后，得到一个BasicScene类，然后再在第二阶段去创建对象实例，这是因为加入了GPU支持以后，需要考虑更多内容，所以拆分为了两个步骤（本系列前两本小册子详细介绍过流程）。

加载好场景文件以后，就调用RenderWavefront()或RenderCPU()来执行渲染。注意RenderWavefront()既能运行CPU渲染，也能运行GPU渲染。运行CPU渲染时和RenderCPU()一样都是CPU并行渲染，但并行程度比RenderCPU()也要高一些。

在RenderCPU()渲染路径中，实现Integrator接口的类的实例负责渲染。因为Integrator实现只在CPU上运行，所以将Integrator定义为具有纯虚拟方法的标准基类。Integrator和各种实现分别在文件cpu/Integrator.h和cpu/Integrator.cpp中定义。

场景中的每个光源都由一个实现“Light”接口的对象表示，该对象允许Light指定其shape和发射的能量分布。某些Light需要知道整个场景的bounding box，而在首次创建时，该bounding box是不可用的。因此，Integrator的构造函数调用它们的Preprocess()方法，提供这些边界。有时，只需要在这些Light上循环处理照明，而对于具有数千个光源的场景，在所有光源上循环是低效的。

CPU上，Integrator是渲染积分器的基类，派生类ImageTileIntegrator，将整个图像分成多个tiles，然后分别对每个tiles进行渲染。

3 2 ImageTileIntegrator::Render()函数

ImageTileIntegrator::Render()以每波每个像素几个样本来渲染图像。对于前两个波，在每个像素中只采集一个样本。在下一波中，采集两个样本，每个样本的数量在达到当前上限后会在下次采样时翻倍。与一次把一个像素全部样本都渲染完然后再移动到下一个像素相比，这种计算方式意味着在渲染期间可以看到图像的预览。因为pbrt是使用多个线程并行运行的，所以这种方法需要取得平衡。线程获取新图像tile的工作是有成本的，并且一旦没有更多的工作要做，而其他线程仍在对其分配的tiles进行工作，则一些线程在每波结束时都会空闲。这些考虑因素推动了每波上限加倍的方法。

在开始渲染之前，需要一些额外的变量。首先，在计算每条采样射线的贡献的过程中，积分器实现将需要分配少量的临时存储器来存储表面散射特性。由此产生的大量分配很容易淹没系统的常规内存分配方式（例如，new关键字），这些方式必须协调复杂数据结构的多线程维护，以跟踪空闲内存。过于简单的实现可能会在内存分配中花费相当大的计算时间。为了解决这个问题，pbrt提供了一个ScratchBuffer类，用于管理一个小的预先分配的内存缓冲区。ScratchBuffer的分配非常有效，只需要增加一个偏移量。ScratchBuffer不允许独立释放分配；相反，必须同时释放所有对象，但这样做只需要重置该偏移量。

由于ScratchBuffer对于多个线程同时使用是不安全的，因此会使用ThreadLocal模板类为每个线程创建一个单独的ScratchBuffers。ThreadLocal的构造函数接受一个lambda函数，该函数返回它所管理类型的对象的新实例；在这里，调用默认的ScratchBuffer构造函数就足够了。ThreadLocal然后处理为每个线程维护对象的不同副本的细节，并根据需要进行分配。在ImageTileIntegrator::Render()函数里，会在并行线程中用ThreadLocal对象给每个线程分配单独的ScratchBuffer，我们传给ThreadLocal对象的参数就是构造ScratchBuffer的函数。

```
1 ThreadLocal<ScratchBuffer> scratchBuffers(
2     []() { return ScratchBuffer(); } );
```

大多数采样器实现都发现维护一些状态很有用，比如当前像素的坐标。这意味着多个线程不能同时使用一个采样器，ThreadLocal也用于采样器管理。采样器提供了一个Clone()方法，用于创建其采样器类型的新实例。

```
1 ThreadLocal<Sampler> samplers(
2     [this]() { return samplerPrototype.Clone(); });
```

向用户提供完成了多少渲染工作的指示以及需要多长时间的估计是有帮助的。此任务由ProgressReporter类处理，该类将工作项的总数作为其第一个参数。这里，总工作量是在每个像素中采集的样本数量乘以像素总数。使用64位精度来计算此值很重要，因为32位int可能不足以用于每个像素具有许多样本的高分辨率图像。

对于渲染循环，ParallelFor2D()自动选择tile大小来平衡两个问题：

- 一方面，希望拥有比系统中处理器多得多的tiles。一些tiles可能比其他tiles花费更少的处理时间，因此，例如，如果处理器和tiles之间存在1:1的映射，则一些处理器在完成工作后将处于空闲状态，而其他处理器则继续处理其图像区域。
- 另一方面，tiles过多也会影响效率。线程在并行for循环中获得更多工作的固定开销很小，而且tiles越多，必须支付的开销就越多。因此，ParallelFor2D()选择一个tile大小，该大小既考虑了要处理的区域的范围，也考虑了系统中处理器的数量。

渲染线程会调用EvaluatePixelSample(...)函数来渲染每个像素。该函数在ImageTileIntegrator类是一个纯虚函数，ImageTileIntegrator类的派生类RayIntegrator提供了实现。

3 3 RayIntegrator类

RayIntegrator实现了ImageTile Integrator中的纯虚拟EvaluatePixelSample()方法。在给定的像素上，它使用相机和采样器生成一条进入场景的射线，然后调用派生类提供的Li()方法来确定沿着该射线到达图像平面的光量。该值将传递给Film，Film记录射线采样的光量对图像的贡献。

每条射线都具有多个离散波长λ（默认为四个）的辐射度。在计算每个像素的颜色时，pbrt在不同的像素样本中选择不同的波长，以便最终结果更好地反映所有波长的正确结果。为了选择这些波长，采样器首先提供采样值lu，lu值在$[0,1)$的范围内均匀分布。然后，Film::SampleWavellength()方法将该样本映射到一组特定的波长，并将其薄膜传感器响应模型作为波长的函数。大多数采样器的实现都确保：如果在一个像素中采集多个样本，则这些样本在集合中均匀分布[0,1)。反过来，它们确保采样波长也在有效波长范围内良好分布，从而提高图像质量。

pbrt v4已经删除了所有使用RGB颜色进行的照明计算，现在只根据波长相关光谱分布的样本来执行照明计算。这种方法不仅在物理上比使用RGB更准确，而且还允许pbrt对色散等效果进行精确建模。关于光谱采样的内容会在后面的系列文章中介绍。

CameraSample结构记录Film上相机应为其生成光线的位置。该位置受采样器提供的采样位置和用于将多个采样值滤波为像素的单个值的重建滤波器的影响。GetCameraSample()处理这些计算。CameraSample还存储与光线相关的时间以及镜头位置样本，分别用于渲染具有移动对象的场景和模拟非针孔光圈的相机模型。

四　一些程序和架构设计

指针/引用

关于是否用指针还是引用，pbrt中的约定是，当参数将被函数或方法完全更改时使用指针，当其某些内部状态将被更改但不会完全重新初始化时使用引用，当它根本不会更改时使用常量引用。该规则的一个重要例外是，当我们希望能够传递nullptr来指示参数不可用或不应使用时，将始终使用指针。

抽象/效率

在为软件系统设计接口时，主要的紧张关系之一是在抽象和效率之间进行合理的权衡。例如，许多程序员虔诚地将所有类中的所有数据设为私有数据，并提供获取或修改数据项值的方法。对于简单的类（例如Vector3f），pbrt认为这种方法不必要地隐藏了它的基本属性——拥有三个浮点坐标值。当然，不使用信息隐藏和公开所有类内部的所有细节会导致代码维护噩梦，但pbrt相信，在整个系统中明智地公开基本设计决策并没有错。例如，光线用一个点、一个向量、一个时间和它所处的介质来表示，这是一个不需要隐藏在抽象层后面的决定。当这些细节暴露出来时，其他地方的代码会更短，更容易理解。

在编写软件系统并进行这些权衡时，需要记住的一件重要事情是系统的预期最终大小。pbrt大约有70000行代码，它永远不会增长到一百万行代码；这一事实应该反映在系统中使用的信息隐藏量中。设计接口以适应更高复杂度的系统将是对程序员时间的浪费（可能也是运行时效率低下的原因）。

pstd

在pstd命名空间中重新实现了C++标准库的一个子集；为了在CPU和GPU上可互换地使用它的这些部分，这是必要的。为了阅读pbrt的源代码，pstd中的任何东西都提供了与std中相应实体相同的功能，具有相同的类型和方法。因此，书中没有再记录pstd的用法。

Allocator

在pbrt中表示场景的对象的几乎所有动态内存分配都是使用提供给对象创建方法的Allocator实例执行的。在pbrt中，Allocator是C++标准库的pmr::polymorphic_allocator类型的简写。它的定义在pbrt.h中，因此它可用于所有其他源文件。

```
1 using Allocator = pstd::pmr::polymorphic_allocator<std::byte>;
```

它的实现提供了少量用于分配和释放内存的函数，这在PBRT中广泛使用：

```
void *allocate_bytes(size_t nbytes, size_t alignment);
template <class T> T *allocate_object(size_t n = 1);
template <class T, class... Args> T *new_object(Args &&... args);
```

上面第二个函数调用类默认构造函数来创建对象；第三个函数可以使用一些初始化参数提供给构造函数来创建对象。

与使用具有C++标准库中数据结构的allocator有关的一个棘手细节是，容器的allocator在其构造函数运行后就固定了。因此，如果将一个容器赋值给另一个容器，则目标容器的allocator保持不变，即使它存储的所有值都更新了。（即使是C++的move语义也是如此。）因此，在pbrt中，通常会看到对象的构造函数在它们存储的容器的成员初始值设定项列表中传递allocator，即使它们还没有准备好设置存储在其中的值。Allocator是C++的STL中一个一般用户不会直接用到的概念，一些细节可以参考[5]。

使用显式内存allocator而不是直接调用new和delete有一些优点。它不仅使跟踪已分配的内存总量等操作变得容易，而且还使替换针对许多小分配进行优化的allocator变得容易，这在构建加速结构时非常有用。以这种方式使用allocator还可以轻松地将场景对象存储在使用GPU渲染时GPU可见的内存中。

dynamic dispatch

虚函数通常不用于pbrt中多态类型的动态分派（主要例外是Integrators）。与之不同的是，TaggedPointer类用于表示指向指定类型集之一的指针；它包括用于运行时类型识别和动态分派的机制。（其实施可在附录B.4.4中找到。）

有两个考虑因素促使其使用。首先，在C++中，从抽象基类继承的对象的实例包括一个隐藏的虚函数表指针，该指针用于解析虚拟函数调用。在大多数现代系统上，此指针使用8字节的内存。虽然八个字节看起来可能不多，但pbrt发现，当使用以前版本的pbrt渲染复杂场景时，大量内存将仅用于形状和基元的虚函数指针。而使用TaggedPointer类就不会增加类型信息的存储成本。虚函数表的另一个问题是，它们存储指向可执行代码的函数指针，这一特性意味着虚函数表可以对来自CPU或GPU的方法调用有效，但不能同时来自两者，因为不同处理器的可执行代码存储在不同的存储位置。然而，当使用GPU进行渲染时，能够从两个处理器调用方法是很有用的。

对于所有调用多态对象方法的代码，使用pbrt的Tagged Pointer代替虚函数没有什么不同，只是使用进行方法调用“.”运算符，就像用于C++引用一样。[4]第4.5.1节介绍了Spectrum，这是书[4]中出现的第一个基于TaggedPointer的类，它详细介绍了pbrt的动态分派方案是如何实现的。

code optimization

pbrt试图通过使用精心选择的算法而不是通过局部微观优化来提高pbrt的效率，以便更容易理解系统。然而，效率是渲染不可或缺的一部分，因此在整本书[4]中都讨论了性能问题。

对于CPU和GPU来说，处理性能的增长速度都快于将数据从主存加载到处理器的速度。这意味着等待从内存中提取值可能是一个主要的性能限制。[4]讨论的最重要的优化涉及最大限度地减少不必要的内存访问，并以导致一致访问模式的方式组织算法和数据结构；关注这些问题可以比减少执行的指令总数更快地加快程序执行。

debugging and logging

调试渲染器可能很有挑战性，尤其是在大多数情况下结果都是正确的，但并不总是正确的情况下。pbrt包括许多方便调试的工具。其中最重要的是一套单元测试。单元测试在pbrt的开发中是非常宝贵的，因为它保证了测试的功能很可能是正确的。

有了这种保证，就可以缓解调试过程中的问题，例如“我确定这里使用的哈希表本身不是我的错误源吗？”或者，失败的单元测试几乎总是比渲染器生成的错误图像更容易调试；在调试pbrt的过程

中，添加了许多测试。文件code.cpp的单元测试可以在code_tests.cpp中找到。所有单元测试都是通过调用pbrt_test可执行文件来执行的，可以通过命令行选项选择特定的单元测试。

pbrt代码库中有许多断言，其中大多数都没有包含在书中。这些检查条件永远不应该为true，如果在运行时发现它们为true，则会发出错误并立即退出。（有关pbrt中使用的断言宏的定义，请参见[4]的第B.3.6节。）失败的断言提供了有关错误源的第一个提示；与单元测试一样，断言有助于集中调试，至少有一个起点。pbrt中一些计算成本更高的断言仅针对调试构建启用；如果渲染器崩溃或以其他方式产生不正确的输出，那么值得尝试运行调试构建，看看这些附加断言中的一个是否失败并产生线索。

pbrt还努力使在给定像素样本上的执行具有确定性。调试渲染器的一个挑战是有时候在渲染计算数分钟或数小时后才会发生崩溃。通过确定性执行，可以在单个像素采样处重新启动渲染，以便更快地返回到崩溃点。此外，在崩溃时，pbrt将打印诸如“在像素（16，27）采样821处渲染失败”之类的消息。使用–debugstart 16,27821“进行调试。“debugstart”之后打印的值取决于所使用的积分器，但足以在接近崩溃点时重新启动其计算。

最后，在调试过程中打印出存储在数据结构中的值通常很有用。pbrt已经为几乎所有的类实现了ToString()方法。它们返回它们的 std::string 表示，这样在程序执行期间就可以很容易地打印出它们的完整对象状态。此外，当在格式化字符串中找到%s说明符时，pbrt的自定义Printf()和StringPrintf()函数（[4]的第B.3.3节）会自动使用ToString()为对象返回的字符串。

parallelism and thread safety

在pbrt中（与大多数光线跟踪器的情况一样），渲染时的绝大多数数据都是只读的（例如，场景描述和纹理图像）。场景文件的大部分解析和内存中场景表示的创建都是通过单个执行线程完成的，因此在执行阶段几乎没有同步问题。在渲染期间，多个线程对所有只读数据的并发读取访问在CPU和GPU上都没有问题；我们只需要关注内存中的数据被修改的情况。

一般来说，系统中的低级类和结构不是线程安全的。例如，Point3f类存储三个浮点值来表示三维空间中的一个点，对于多个线程同时调用修改它的方法来说是不安全的。（当然，多个线程可以同时使用Point3f作为只读数据。）使Point3f线程安全的运行时开销会对性能产生重大影响，但收效甚微。Vector3f、Normal3f、SampledSpectrum、Transform、Quaternion和SurfaceInteraction等类也是如此。这些类通常在场景构建时创建，然后用作只读数据，或者在渲染期间在堆栈上分配，仅由单个线程使用。

实用程序类ScratchBuffer（用于高性能临时内存分配）和RNG（伪随机数生成）对于多个线程使用也不安全；这些类存储在调用其方法时修改的状态，并且相对于它们执行的计算量，通过互斥来保护对其状态的修改的开销将过大。因此，在像前面的ImageTileIntegrator::Render()方法这样的代码中，pbrt在堆栈上为这些类的每个线程分配实例。

除了两个例外的类，本文2.1节的表中列出的基类型的实现对于多个线程同时使用是安全的。只要稍微注意，通常可以直接实现这些基类的新实例，这样它们就不会修改方法中的任何共享状态。

第一个例外是Light Preprocess()方法的实现。这些由系统在场景构建过程中调用，Preprocess()的实现通常会修改其对象中的共享状态。因此假设只有一个线程会调用这些方法。（考虑这些计算密集型方法的实现可能会使用ParallelFor()来并行化它们的计算是另一个单独的问题。）

第二个例外是Sampler类的实现；他们的方法也不是线程安全的。这是另一个这种要求会对性能和可扩展性造成过度影响的例子；多个线程同时试图从单个采样器获取样本将限制系统的整体性能。因此，将使用Sampler::Clone()为每个渲染线程创建一个唯一的Sampler。

pbrt中的所有独立函数都是线程安全的（只要多个线程不向它们传递指向相同数据的指针）。

五　小结

PBRT v4是一个相对比较复杂的系统，里面也有很多底层的复杂实现。本系列将会把一些底层实现剥离出来，放在各个小专题文章里，正文主要描述PBRT v4系统的代码架构以及关键原理。

参考文献

[1] https://github.com/mmp/pbrt-v4

[2] https://pbrt.org/

[3] https://pbrt.org/resources

[4] Pharr, Matt, Wenzel Jakob, and Greg Humphreys. Physically based rendering: From theory to implementation. MIT Press, 2023.

[5] https://www.cnblogs.com/pointer-smq/p/7522416.html