Dezeming Family

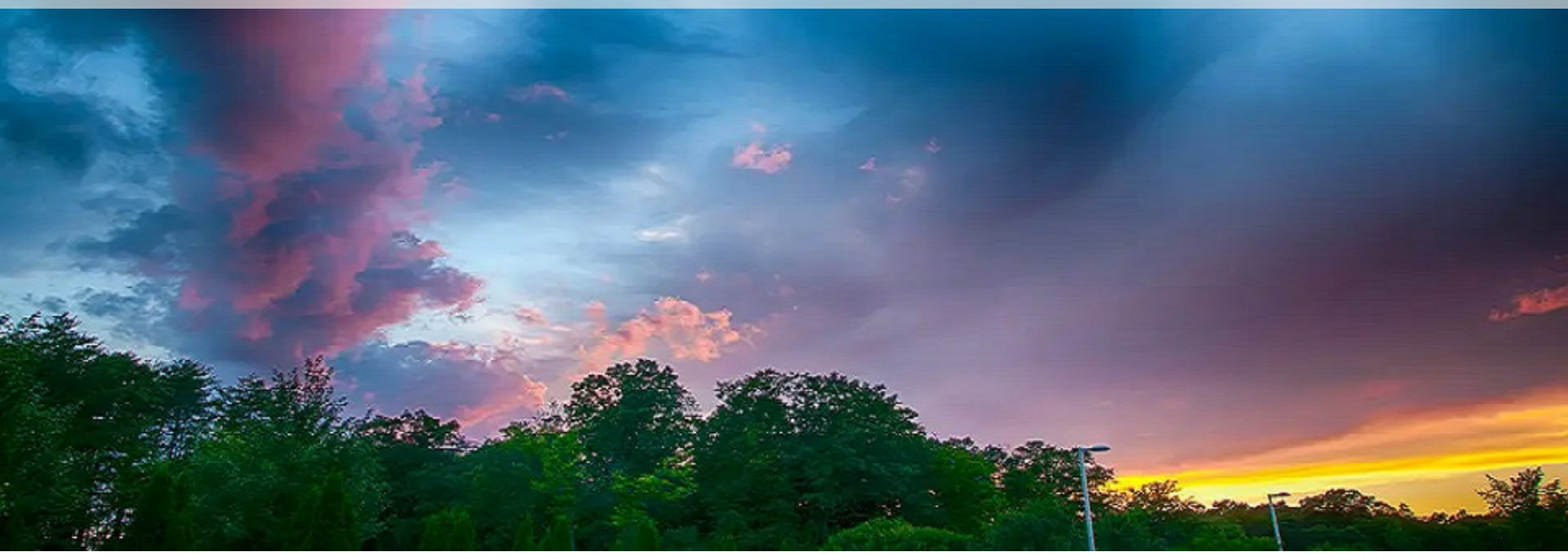
PBRT 系列 26-高级积分器-Metropolis 光传输

Art. No 0
ISBN 000–00–0000–00–0
Edition 0.0

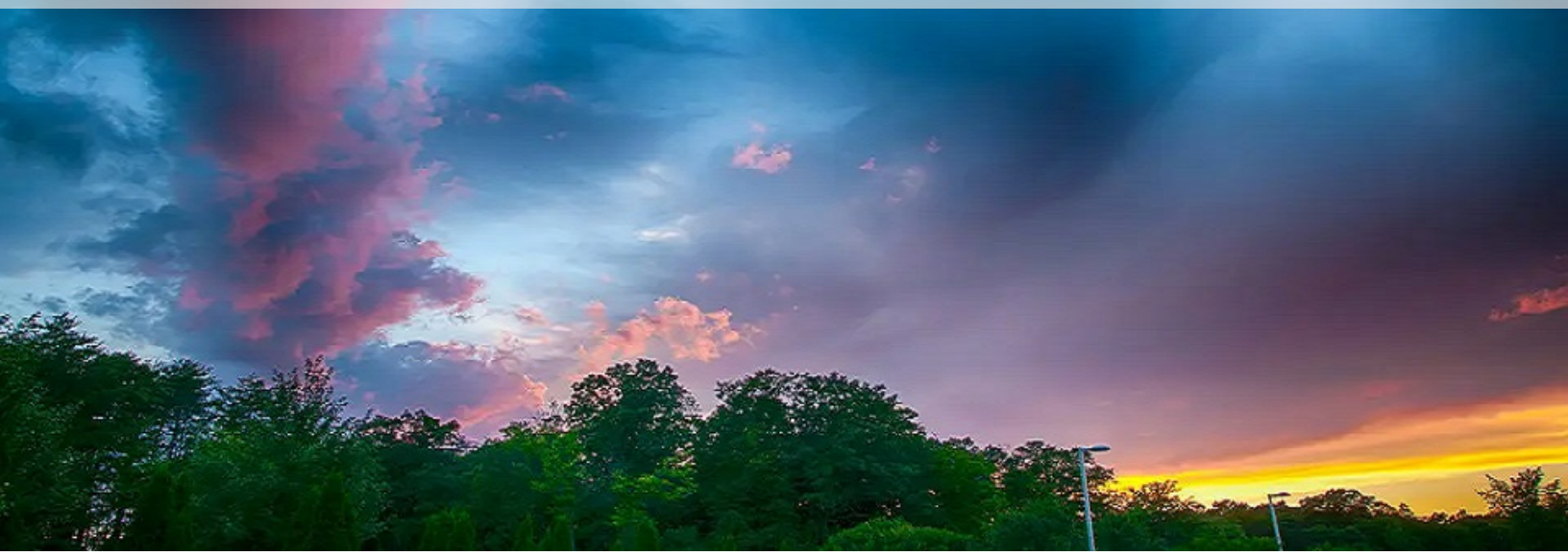
Cover design by Dezeming Family

Published by Dezeming
Printed in China

目录



前言及简介



DezemingFamily 系列文章和电子书**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* 的系列电子书，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。

0.1 本文前言

对于积分 $\int f(x)dx$，使用蒙特卡洛方法估计的方差是：

$$\sigma^2 = \frac{1}{N}\int \left(\frac{f(x)}{p(x)} - I\right)^2 p(x)dx \tag{0.1.1}$$

如果概率密度和函数恰好能成正比，则方差就是 0。否则，计算误差随着 $\sqrt{N}$ 降低，比如采样四倍的样本，才能使得估计结果减少一半。

双向方法虽然可以避免路径追踪中无法追踪到光源的问题，但由于可见性测试等原因，大量路径是不可见的，导致最终一个像素的有效的路径连接比较少。有些时候，尤其是在间接光为场景主要照明时，我们很难去找到一些合适的光路径，但当恰好找到比较合适的路径后，它对于图像的贡献又比较少，方差还是很大。

Metropolis 光传输算法可以理解为是试图使得样本分布趋近于真实分布的方法，通过将已经构建好的路径进行扰动，得到新的路径，因此很多时候，尤其是光路不好找时，它可以更有效。Metropolis-Hastings 算法的基本原理以及 Metropolis 光传输算法 (MLT) 在《模拟光传输的鲁棒的蒙特卡洛方法（Eric Veach）全文解读》中已经介绍过了，PBRT 并没有实现 Veach 的 MLT，而是另一种改进的 MLT，它实现起来更简单；Mitsuba 中有 MLT 的基本实现，以后有时间我会做一套关于 Mitsuba 的源码解读的电子书，但现在我们以 PBRT 为基准，学习改进的 MLT 算法。

1. MLT 的基本介绍

本章对 *PBRT* 中的 *MLT* 进行一些基本介绍。

1.1 最初的 MLT 应用

MLT 生成通过场景的一系列光传输路径，其中通过以某种方式改变前一路径来找到下一条路径。这些路径突变以确保采样路径的总体分布与路径对生成的图像贡献成比例，这种路径分布又可用于生成场景的图像。考虑到 Metropolis 抽样方法的灵活性，对可接受的突变规则类型的限制相对较少：可以用高度专门化的突变来抽样，否则很难探索有效光路径，这会很难或不可能实现而不在标准蒙特卡洛环境中引入偏差。

与基于独立样本生成的方法相比，MLT 的样本相关性提供了一个重要的优势，因为 MLT 能够执行路径空间的局部探索：当发现对图像有很大贡献的路径时，通过对其施加小扰动，很容易找到其他类似的路径（以这种方式基于先前状态值生成新状态的采样过程称为马尔可夫链）。

由此产生的短期记忆通常是有益的：当一个函数在其大部分域上具有较小的值，并且仅在较小的子集上具有较大的贡献时，局部探索通过从这部分路径空间中提取许多样本来摊销搜索重要区域的计算费用（以样本为单位）。该特性使 MLT 成为渲染特别具有挑战性的场景的一个很好的选择：虽然它通常无法在相对简单的照明问题上与样本不相关的积分器的性能相匹配，但它在照明更困难的场景中会很有用，在这些场景中，大多数光传输都沿着场景中所有可能路径的一小部分发生。

Veach 和 Guibas 的 MLT 技术基于光传输的路径空间理论，与生成简单的函数分布相比，该理论提出了额外的挑战：路径空间通常不是欧几里德域，因为曲面顶点被限制在 $\mathbb{R}^3$ 的 $2D$ 子集上。每当发生镜面反射或折射时，三个相邻顶点的序列必须满足精确的几何关系，这进一步降低了可用的自由度。

MLT 建立在一组五个突变规则的基础上，每个突变规则都针对特定的光路种类。其中三个突变对特别具有挑战性的路径类（例如焦散或包含镜面漫反射-镜面交互序列的路径）进行了局部探索，而两个突变则以相对较低的总体接受率执行更大的突变步骤。能够实施全套的 MLT 突变是

很重要的：有些挑战在于突变都不是对称的，因此必须为每一个突变计算转移密度函数。系统中任何部分的错误都可能导致难以调试的细微的收敛伪影。

1.2 PRIMARY SAMPLE SPACE MLT

Kelemen 于 2002 年提出了一种渲染技术，也是基于 Metropolis-Hastings 算法的，叫做初始样本空间 MLT(Primary Sample Space MLT)。PSSMLT 会探索光路空间，搜寻携带较高光能的路径。区别是 PSSMLT 并不会直接使用路径空间，它会间接地探索光路，搭载现有的蒙特卡洛渲染算法，如单向或双向路径跟踪，这既有优点也有缺点。主要优点是它可以利用欧式空间的对称转移（因此更容易实现）；缺点是它缺乏构建光路结构的详细信息，因此使得不可能重现原始 MLT 方法中实现的复杂突变策略。

这种方法的细节从以实现为中心的角度来看是最容易想到的，假设路径追踪从光栅空间产生了一个随机样本，并通过 PathIntegrator::Li() 估计出辐射度，Li() 函数会接收一堆随机样本来估计路径和光照。从概念上讲，这些样本都可以提前生成，并使用额外的参数传递给 PathIntegrator::Li()，从而从纯确定性函数中完全消除任何（伪）随机性。

假设 L 是一个确定性函数，从无限维的样本序列 $X_1, X_2, \ldots$ 到辐射度估计 $L(X_1, X_2, \ldots)$，这里的 (X_1, X_2) 表示光栅空间的位置，其他参数都是 L 计算中使用的。通过产生一大堆样本，进行多次 L 估计并取平均，就能得到误差较小的积分值。

我们基于上面的分析，通过在相同域根据 Metropolis-Hastings 算法创建一个采样过程，样本分布正比于 L。这种分布在直觉上是可取的，更多的采样被分配在更多光传输发生的采样空间中。状态空间 Ω 包含无限维样本 $(X_1, X_2, \ldots) \in [0,1)^\infty$ 叫做初始样本空间：

$$\mathbf{X} = (X_1, X_2, \ldots) \tag{1.2.1}$$

所有的 X_i 都在区间 $[0,1)$ 内，当使用时，如果有必要会在 L 内缩放（比如对面积采样时，将原始的 $[0,1)$ 内的随机数缩放几倍，但概率什么的都是不变的）。

PSSMLT 使用两种不同类型的突变，第一种是用新的均匀分布的随机序列替换 $\mathbf{X}$（这对应于使用以前的蒙特卡洛渲染器）。这属于大突变，用于探索整个路径空间。第二种突变是对每个样本 X_i 做小的扰动，用来探索与当前光路相似的周边光路。当当前光路是很难遇到的的光路时，该方法会很有效，它可以探索周边的重要区域。这两种突变都是对称的，因此他们的转移概率都可以不用再计算。

Metropolis-Hastings 迭代和 Integrator 之间的接口仅涉及抽象样本向量的交换,这使得 PSSMLT 成为一种极其通用的方法：PSSMLT 理论上可以增强基于蒙特卡洛积分的任何类型的渲染方法。事实上，它甚至适用于与渲染完全无关的一般蒙特卡洛积分问题。

在实践中，PSSMLT 通常在现有双向路径跟踪器的基础上实现。PSSMLT 算法在每次迭代中生成一个新的主样本空间状态，并将其传递给 BDPT，BDPT 调用其一组连接策略，并使用 MIS 对结果重新加权。在这种情况下，突变不是处理单个连接策略，而是一次处理一组连接策略，然后再突变到另一组（每次用 BDPT 找到有效路径之后，分别对其 eye path 和 light path 进行 Primary Sample Space 上的扰动，形成新的 eye path 和 light path，由于 bdpt 中一条 eye path 和 light path 可能连接处多条有效路径，所以贡献度函数 f 的选择要考虑多条有效路径的贡献度，用 MIS 来加权）。

然而，这并非没有缺点：在许多情况下，只有策略的一小部分是真正有效的，因此 MIS 将仅为该子集分配较大的权重。然而该算法仍将相当一部分时间用于生成与权重较低的策略的连接并计算，这些连接对渲染图像的贡献很小，但是同样占用较多的计算时间。

1.3 MULTIPLEXED MLT

Hachisuka 在 2014 年对 PSSMLT 做了一些扩展，称为 Multiplexed Metropolis Light Transport(MMLT)。MMLT 并不改变 Metropolis-Hastings 概念，而是对 BDPT 进行了一些小的修改。它并不调用所有的 BDPT 连接策略，而是根据一个额外的状态维度来选择一个单个连接策略，并返回它的被反离散概率密度放缩后的贡献（反概率密度与选择哪个策略有关）。用于策略选择的附加维度可以使用小的或大的步骤进行突变，其方式与 $\mathbf{X}$ 的其他初始样本空间维度分量相同。

为了防止无意中的大型结构路径突变，Hachisuka 等人固定了马尔可夫链，使其仅探索固定深度值的路径;然后通过运行许多独立的马尔可夫链来处理一般的光传输问题。实际结果是，Metropolis 采样器将倾向于在有效策略上花费更多的计算，从而为图像产生更大的 MIS 加权贡献。此外，单个迭代速度更快，因为它们只涉及单个连接策略。这两个方面的结合提高了所得估计器的蒙特卡洛效率。

1.4 MMLT 应用于渲染

Metropolis 抽样从给定标量函数的分布生成样本，为了将其应用于渲染，我们必须解决两个问题：首先，我们需要为每个像素估计一个单独的积分，以将生成的样本转换为图像，其次，我们需要处理以下事实：L 是一个光谱值函数（可以简化为 RGB 分量），但 Metropolis 需要一个标量函数来计算接受概率。

我们首先定义图像的贡献函数 (image contribution function)，每个像素 I_j 都是像素滤波器 h_j 与贡献到图像的辐射度 L 的乘积的积分：

$$I_j = \int_\Omega h_j(\mathbf{X}) L(\mathbf{X}) d\Omega \tag{1.4.1}$$

h_j 只与 $\mathbf{X}$ 的两个组件有关（两个分量，描述了像素位置）。由于像素滤波器值不为 0 的范围有限，所以 $L(\mathbf{X})$ 对大部分像素的贡献都是 0。

如果 N 个样本序列 $\mathbf{X}_i$ 来自某个分布 $\mathbf{X}_i \sim p(\mathbf{X}_i)$，标准的蒙特卡洛估计就是：

$$I_j \approx \frac{1}{N} \sum_{i=1}^{N} \frac{h_j(\mathbf{X}_i) L(\mathbf{X}_i)}{p(\mathbf{X}_i)} \tag{1.4.2}$$

由于 Metropolis 采样需要一个定义了采样的理想分布的标量函数，然而 L 是一个光谱值函数，因此无法产生正比于 L 的样本，为此，我们定义标量贡献函数，$I(\mathbf{X})$，定义它可以保证样本的分布会与 L 的重要区域有关。因此，使用辐射度值的亮度是标量贡献函数的一个好选择。一般来说，当 L 为非零时，任何非零的函数都会产生正确的结果，只是可能不如与 L 更直接成比例的函数更有效。

给定一个合适的标量贡献函数 $I(\mathbf{X})$，Metropolis 方法从 I 的分布中产生一系列的样本 $\mathbf{X}_i$，I 的归一化版本（概率密度）就是：

$$p(\mathbf{X}) = \frac{I(\mathbf{X})}{\int_\Omega I(\mathbf{X}) d\Omega} \tag{1.4.3}$$

像素值就可以估计为：

$$I_j \approx \frac{1}{N} \sum_{i=1}^{N} \frac{h_j(\mathbf{X}_i) L(\mathbf{X}_i)}{I(\mathbf{X}_i)} \Big(\int_\Omega I(\mathbf{X}) d\Omega \Big) \tag{1.4.4}$$

我们可以先用传统方法估计出一个 $\int_\Omega I(\mathbf{X}) d\Omega$ 值，设为 b，则每个像素值就是：

$$I_j \approx \frac{b}{N} \sum_{i=1}^{N} \frac{h_j(\mathbf{X}_i) L(\mathbf{X}_i)}{I(\mathbf{X}_i)} \tag{1.4.5}$$

也就是说更亮的像素是由于它接收到了更多的样本（该像素滤波器值不为 0 的范围内样本更多）因此值会更大。

2. PBRT 中的 PSSMLT

本章对 *PBRT* 中的 *MLT* 算法的一些基本功能和函数做一些介绍。

2.1 MLTSampler 类的成员变量

MLTIntegrator 使用 Metropolis 采样和 MMLT 来渲染图像，使用前面的双向路径追踪方法估计路径。我们本章节主要介绍 MLTSampler，它用来管理初始样本空间的状态向量、变异以及接受和拒绝（双向路径追踪需要接受一个采样器来得到样本，因此，我们为 MLT 设计的采样器要继承自 Sampler）。

```
1 class MLTSampler : public Sampler
```

MLT 采样器实际上有三个单独的采样向量，一个用于相机子路径，一个为光源子路径，另一个用于连接步骤，我们将描述它们说是三个样本流（PBRT 定义为 const 型常量 nSampleStreams=3）。MLTSampler 的构造函数的 streamCount 参数允许调用者请求特定数量的此类样本流（但是 MLT 实现中都是输入参数为 3）。稍后在 MLTIntegrator 的初始化阶段，我们将创建许多独立的 MLTSampler 实例，用于为 Metropolis 采样器选择一组合适的启动状态。

```
1 MLTSampler(int mutationsPerPixel, int rngSequenceIndex, Float sigma,
    Float largeStepProbability, int streamCount)
```

重要的是，该过程要求每个 MLT 采样器产生不同的状态向量序列。pbrt 使用的 RNG(pseudo-random number generation) 伪随机数生成器有一个方便的特性，使其易于实现：RNG 构造函数接受一个序列索引 (sequence index)，该索引在 2^{63} 个唯一伪随机序列中进行选择。因此，我们向 MLTSampler 构造函数添加了一个 rngSequenceIndex 参数，该参数用于向内部 RNG 提供唯一的流索引，即构造函数中给一个初始种子（流索引值），它就产生一系列对应的伪随机数（种子值一旦确定了，伪随机数序列也就确定了）。程序中通过 UniformFloat() 来得到伪随机数值。

MLTSampler 中的 largeStepProbability 表示大步变异的概率（默认是 0.3）；sigma 控制的是小步变异的大小（默认是 0.01），后面会介绍 sigma 是高斯分布的方差（σ）。

MLTSampler::X 存储的是当前样本向量 **X**。由于我们提前不知道实际的 **X** 是多少维的，所以我们将会从一个空的 vector 开始，根据需求来扩展它（使用中会调用 MLTSampler::Get1D() 和 MLTSampler::Get2D()）：

```
1  std::vector<PrimarySample> X;
```

此数组的元素具有 PrimarySample 类型。PrimarySample 的主要任务是记录间隔 $[0,1)$ 上 **X** 的单个分量的当前值，它有一些功能，可以表示建议的突变，并在建议的突变被拒绝时恢复原始样本值。后面涉及 PrimarySample 的具体功能时我们再进行解释。

2.2 MLTSampler 类的成员函数

MLTSampler::Get1D() 函数返回 MLTSampler::X 内下一个索引的值（目前，我们可以将 GetNextIndex() 视为返回随每次调用而增加的运行计数器的值，实际上它跟当前的流索引有关系，但我们后面再统一解释）。EnsureReady() 根据需要扩展 MLTSampler::X，并确保其内容处于一致状态（而不是算法执行前先进行初始化），这些方法的细节将在进一步的了解之后更加清晰，所以我们现在还不介绍它们的实现。MLTSampler::Get2D() 就是调用两次 Get1D() 函数，得到一个二维随机数。

有些函数不属于 Sampler 基类的函数，而是新定义的函数。MLTSampler::StartIteration() 在 Metropolis 每次的迭代开始前被调用，增加 currentIteration 计数器并决定要给当前迭代的样本向量进行大的突变还是小的突变，赋值给 largeStep 这个 bool 型变量。

MLTSampler::lastLargeStepIteration 变量记录了最后一个成功的大突变的迭代索引。我们的实现选择初始的 $\mathbf{X}_0$ 是在样本空间 $[0,1)^\infty$ 均匀分布的，因此第一次迭代状态可以被解释为迭代 0 中的大突变的结果。

MLTSampler::Accept() 函数中，如果当前突变是大突变，就重新给 lastLargeStepIteration 变量赋值。

```
1 void MLTSampler::Accept() {
2     if (largeStep) lastLargeStepIteration = currentIteration;
3 }
```

理论上，X 数组的所有元素必须在每次 Metropolis 采样中被小的或大的突变来更新。然而，这么做的效率可能并不高，考虑大部分迭代仅仅会访问 **X** 中的一小部分维度，因此 MLTSampler::X 的大小还没有扩展到很大。如果有一次迭代调用了多个 Get1D() 或 Get2D()，那么动态数组 MLTSampler::X 比如相应扩展，则这会增加每次后续迭代的计算消耗。

因此，根据需求来更新 PrimarySample 效率更高，也就是说，当在实际调用 Get1D() 或者 Get2D() 的时候再去更新。这样做可以避免前面提到的低效率，尽管可能很长时间 PrimarySample 都没有成功突变，但我们必须能够去查看给定 PrimarySample 错过的所有突变。lastModificationIteration 变量就是记录了上一次修改的对应迭代次数。有了这些背景，我们下一节就介绍 EnsureReady() 方法。

2.3 EnsureReady() 方法

该方法是为了更新独立的样本值。

```
void MLTSampler::EnsureReady(int index) {
  如果有必要就扩大MLTSampler::X并获取当前索引的X元素
  如果同时有大的突变就重置当前索引的X元素
  把剩余的突变序列应用到样本中
}
```

该过程相对比较简单，只是有些地方需要注意。我们希望对组件生成前一个状态的正态分布：

$$\mathbf{X}_i' \sim N(\mathbf{X}_i, \sigma^2) \tag{2.3.1}$$

这有助于小突变，可以探索临近区域。但另一方面，它也可以进行更大的突变，避免了在一个小区域太长时间。我们计算出从上一次成功突变到当前为止经历的迭代次数 nSmall，我们希望当前再尝试突变时，将前面所有的未成功的突变步骤都囊括进来（也就是从上一次成功突变开始，失败几次，当前就突变几次）。由于多个正态分布的样本的和相加以后也是正态分布，当对 $\mathbf{X}_i$ 应用 n 次突变时，相当于直接采样：

$$\mathbf{X}_i' \sim N(\mathbf{X}_i, n\sigma^2) \tag{2.3.2}$$

正态分布的概率密度的采样：

$$\begin{aligned} p(x) &= \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \\ \xi &\sim U[0,1) \\ P^{-1}(\xi) &= \sqrt{2} erf^{-1}(2\xi - 1) \end{aligned} \tag{2.3.3}$$

在这里是通过 ErfInv() 函数实现的，具体原理可以参考 PBRT 源码。

最后，代码会将超出 1.0 或者低于 0.0，此时需要将样本值进行 warp。比如 1.1 比 1.0 超出了 0.1，那么就变为 0.1；−0.2 比 0.0 低了 0.2，就变为 0.8：

```
Float effSigma = sigma * std::sqrt((Float)nSmall);
Xi.value += normalSample * effSigma;
Xi.value -= std::floor(Xi.value);
```

2.4 开始与结束状态

PrimarySample::Backup() 函数在突变前保存当前样本值。

PrimarySample::Restore() 函数在突变被拒绝时恢复到之前的样本值。

GenerateCameraSubpath() 和 GenerateLightSubpath() 都会需要 $1D$ 和 $2D$ 样本。在 MLT 中，每个 $\mathbf{X}$ 的分量都能够对应于路径空间的顶点。由于相机子路径和光源子路径中的样本流是分

开的，所以对于不同的流需要用到不同的样本序列。MLTSampler::StartStream() 函数接受当前的流索引（前面介绍过，MLT 中有三个流，分别用于相机子路径、光源子路径和连接路径）。

因此，MLTSampler::X 中样本排序就是：第 0 流的第 0 个样本、第 1 流的第 0 个样本、第 2 流的第 0 个样本；第 0 流的第 1 个样本、第 1 流的第 1 个样本、第 2 流的第 1 个样本；第 0 流的第 2 个样本、第 1 流的第 2 个样本、第 2 流的第 2 个样本...

```
1 int GetNextIndex() { return streamIndex + streamCount * sampleIndex++;
    }
```

2.5 MLTSampler 在代码中的使用

在正式介绍积分器之前，我用一节来描述一下 MLTSampler 的使用流程。

程序正式开始 MLT 变异之前，需要估计 b 和生成初始路径。MLTIntegrator::Render() 会有一个多线程结构，用来根据 nBootstrap 次估计来得到 b，每次估计中，都会遍历所有深度：

```
1 for (int depth = 0; depth <= maxDepth; ++depth) {
2   int rngIndex = i * (maxDepth + 1) + depth;
3   MLTSampler sampler(mutationsPerPixel, rngIndex, sigma,
      largeStepProbability, nSampleStreams);
4   Point2f pRaster;
5   bootstrapWeights[rngIndex] = L(scene, arena, lightDistr,
      lightToIndex, sampler, depth, &pRaster).y();
6   arena.Reset();
7 }
```

i 是当前第几次估计的索引，不同的深度都会定义一个新的 sampler。sampler 传入 rngIndex 就是用来表示 RNG 序列的索引的。

3. MLT 积分器的 PBRT 实现

本章介绍 *MLT* 积分器在 *PBRT* 中是如何实现的。

3.1 MLTIntegrator 总述

我们本章节介绍 MLTIntegrator 实现的核心。

MLTIntegrator 的构造函数接受一些基本参数，这些基本参数主要是：

```
maxdepth //路径最大深度（最大散射次数），默认为5
bootstrapsamples // 求b值估计的样本量，默认为100000
chains //并行的马尔科夫链数（后面再讲），默认为1000
mutationsperpixel // 每个像素路径突变次数，默认为100
largestepprobability //大突变的概率，默认为0.3
sigma // 小突变的尺寸
```

我们只有两个函数需要讲，一个是 MLTIntegrator::L()，另一个是 MLTIntegrator::Render()。我们先讲 L() 函数，然后再描述在 Render() 函数的逻辑流程。

3.2 L() 函数

给定样本 **X**，L() 函数计算对应的辐射度估计值。如果成功地找到了携带光的路径，pRaster 参数就会赋值给路径在光栅空间的位置。

一开始会初始化 MLTSampler 以激活三个样本流中的第一个：

```
// cameraStreamIndex是常量0
sampler.StartStream(cameraStreamIndex);
```

然后构建相机子路径。如果构建成功（否则返回 Spectrum(0.f)），就再初始化 MLTSampler 以激活三个样本流中的第二个：

```
// lightStreamIndex是常量1
sampler.StartStream(lightStreamIndex);
```

然后构建光源子路径。如果构建成功（否则返回 Spectrum(0.f)），就再初始化 MLTSampler 以激活三个样本流中的第三个：

```
// connectionStreamIndex是常量2
sampler.StartStream(connectionStreamIndex);
```

然后进行子路径连接。进行子路径连接的样本流主要用于当 $s = 1$ 或者 $t = 1$ 的时候来采样光源或者相机镜头（即 Camera::Sample_Wi() 函数和 Light::Sample_Li() 函数）。

在开始构建路径之前，需要计算所有可选择的策略，并选择其中一个策略（MMLT 对 PSSMLT 的修改就在于只选择其中一个策略）。对于没有散射事件的光路（光直接射入相机），仅有的策略就是 BDPT 从相机发出光线与光源相交。

```
if (depth == 0) {
  nStrategies = 1;
  s = 0;
  t = 2;
} else {
  nStrategies = depth + 2;
  s = std::min((int)(sampler.Get1D() * nStrategies), nStrategies - 1);
  t = nStrategies - s;
}
```

对于长点的路径，有 depth+2 种策略。比如当 depth=1，表示要散射一次，那么要么相机 1 个顶点，光源 2 个顶点；要么相机 2 个顶点，光源 1 个顶点，要么相机 3 个顶点（镜头一个，散射位置处一个，以及采样到的下一个点），光源 0 个顶点。

之后，采样一个 film 的位置（范围在 $[0,1)^2$ 内）映射到光栅坐标，尝试生成一个 t 个顶点的相机子路径，如果子路径长度不是 t 就返回 Spectrum(0.f)。pRaster 在传入 GenerateCameraSubpath() 之前只是限定在 $[0,1)^2$ 的对应绑定边界内的值（该边界值对 $[0,1)^2$ 坐标进行限定）。Lerp 就是线性插值，根据随机数作为比例计算选中的点：

```
Vertex *cameraVertices = arena.Alloc<Vertex>(t);
Bounds2f sampleBounds = (Bounds2f)camera->film->GetSampleBounds();
*pRaster = sampleBounds.Lerp(sampler.Get2D());
```

最后，计算完连接权重，还需要除以抽取到当前策略 (s,t) 的概率，其实就是乘以 nStrategies（后面还会再解释一下这里为什么要除以抽取概率）。

其他内容都比较简单和清晰，所以不再赘述。

3.3 Render() 函数

这是本文最后一个知识点，除非还会添加一些其他内容，否则也将是本系列最后一个知识点，不免有些感慨。

Render() 第一阶段会产生一大堆样本，用于初始马尔科夫链的候选，以及用于估计 $b = \int I(\mathbf{X})d\Omega$。第二阶段会运行一系列的 Markov 链，每个链选择一个样本作为初始样本向量，然后应用 Metropolis 采样。

```
Render() {
  <生成初始样本，计算归一化常量b>
  <并行运行nChains条马尔科夫链>
  <存储最终图像>
}
```

3.3.1 生成初始样本并计算归一化常量 b

我们通过使用标准蒙特卡洛估计器计算一组起始样本，并使用它们创建提供马尔可夫链初始状态的分布，来避免启动偏差问题。这个过程建立在先前实现的样本生成和估计的基础上。每个初始样本在实现上是具有不同路径深度的 $maxDepth + 1$ 个样本的序列，因此初始样本量总数就是：

```
int nBootstrapSamples = nBootstrap * (maxDepth + 1);
```

nBootstrap 给出了执行循环体的次数，chunkSize 给出了每个线程要执行多少次循环体（该值要小于 nBootstrap）。bootstrapWeights 存储的内容是：

```
第0个样本depth=0的估计结果，第0个样本depth=1的估计结果，...，第0个样本depth=maxDepth的估计结果
第1个样本depth=0的估计结果，第1个样本depth=1的估计结果，...，第1个样本depth=maxDepth的估计结果
......
第(nBootstrap-1)个样本depth=0的估计结果，第(nBootstrap-1)个样本depth=1的估计结果，...，第(nBootstrap-1)个样本depth=maxDepth的估计结果
```

而每个结果都是当前路径计算的辐射度乘以 nStrategies 以后的结果。

每次迭代中都会实例化一个专用的 MLTSampler，每个样本向量都有专门的索引 rngIndex，该向量均匀分布在主样本空间中。接下来，我们估计 L 以获得当前路径深度的对应辐射估计，并将其亮度写入 bootstrapWeights。mutationsPerPixel 类似于 Sampler::samplesPerPixel，表示 MLT 中变异的次数（这个变异次数指的是每个像素的平均变异次数，而不是每个像素都完全遵循变异 mutationsPerPixel 次）。由于 Metropolis 的特性是在函数值较高的区域采集更多样本，因此单个像素将接收到与其亮度相关的实际样本数，Metropolis 样本的总数是像素数和每个像素的平均变异数的乘积。

然后用其对应的亮度值初始化阵列引导权重。最后，我们创建了一个 Distribution1D 实例，以对与其亮度成比例进行初始化路径的采样，并将常数 b 设置为每个深度的平均亮度之和（待会再详细介绍）：

```
Distribution1D bootstrap(&bootstrapWeights[0], nBootstrapSamples);
Float b = bootstrap.funcInt * (maxDepth + 1);
```

因为我们保持了不同路径采样深度的贡献不同，所以我们可以优先采样对图像贡献最大的路径长度。并行计算初始样本很简单，因为所有的循环迭代都是彼此独立的。

程序代码中的 b 值

我们前面讲过 $b = \int_\Omega I(\mathbf{X})d\Omega$，也就是对整幅图像的估计辐射度积分。

我们总共对图像有 nBootstrap*(maxDepth+1) 次样本估计，每次样本估计都包含对应不同 (s,t) 策略的估计（随机抽取某一个策略）。

L() 计算的结果存储在 bootstrapWeights[] 数组里，用于生成一维分布 bootstrap。

b 值则是把 bootstrapWeights[] 数组全部元素都加起来,除以总元素数,然后乘以 (maxDepth+1)。在生成 1D 分布时（Distribution1D() 构造函数）：

```
// n就是nBootstrapSamples
for (int i = 1; i < n + 1; ++i) cdf[i] = cdf[i - 1] + func[i - 1] / n;
funcInt = cdf[n];
```

且需要注意到，在我们计算 L() 中，光照值会乘以策略数，也就意味着比如路径策略多（depth 比较大）且抽取的某个策略得到的光照值比较高时，作为候选的比重就比较大。

而且因为没有乘以像素总数，也就意味着是每个像素单独进行估计的。即每个像素每个 depth 的平均，再乘以 (maxDepth+1)，得到每个像素的辐射度的平均（一个像素的辐射度是各种 depth 得到的辐射度的和）。

本节解释到这种程度我已经尽力了，各个量之间的关系究竟怎么跟公式相匹配，还请大家多对照源码和公式看几遍。

3.3.2 并行运行 nChains 条马尔科夫链

nTotalMutations 表示变异总数，即平均每像素变异数乘以像素总数。我们有 nChain 条链，可能无法被 nTotalMutations 整除，因此可以假设最后一条 Markov 链只迭代有限次数，每个链的迭代数计算为 nChainMutations。nChains 条链意味着这些链之间都是独立的，默认 100，这是在足够的并行化与每条链运行时间之间的一个权衡。

一个样本贡献的像素并不一定是指定的像素（何况 MLT 中也没有指定像素），因此需要用到 Film::AddSplat() 函数，用于将样本存储和后续的合并。

每条马尔科夫链都会实例化它自己的伪随机序列生成器，这个 RNG 实例是与 MLTSampler 中分离的，因为 MLTSampler 只是一个状态（用以计算 L() 的伪随机状态序列），而它不能自己产生突变，需要另一个外部随机数生成器来对其进行突变。

由于在初始化 bootstrap 数组的每一项时深度都是排序的，我们可以立即得到采样的索引 bootstrapIndex 的路径深度：

```
int bootstrapIndex = bootstrap.SampleDiscrete(rng.UniformFloat());
int depth = bootstrapIndex % (maxDepth + 1);
```

用于生成初始分布的方法的一个重要结果是，给定 depth 的采样的初始状态的预期数量与其对图像的贡献成比例（有的 depth 对图像贡献很大，比如可能直接光照占主要部分，那么 depth=1 的路径初始状态就会很多）。

对于伪随机序列，我们可以在前面将所有的 MLTSampler 样本存储在一起，然后根据 rngSequenceIndex 来索引。但其实只要确定了 bootstrapIndex，就相当于 MLTSampler 对象是唯一的。

Metropolis 采样的程序的实施遵循期望值技术：提出突变，计算突变样本的函数值和接受概率，并记录新样本和旧样本的加权贡献（新样本权重为 a，旧样本权重为 (1-a)）。然后基于其接受概率随机接受所提出的突变。MMLT 的一个特点是它对每条 Markov 链都会限制固定的 depth 值，第一个样本维度用于从不同的策略中选择，但只有等长的路径深度会被考虑，这样会通过使得突变更局部性来提高效率。通过启动具有不同初始状态的多个马尔可夫链可以来解释所有路径深度的贡献。

```
for (int64_t j = 0; j < nChainMutations; ++j) {
  sampler.StartIteration();
  Point2f pProposed;
  Spectrum LProposed=L(scene, arena, lightDistr, sampler, depth, &
      pProposed);
  <计算突变样本的接受率>
  <把当前和样本和突变的样本splat到film中>
  <计算接受或者拒绝>
}
```

StartIteration() 会标定是大突变还是小突变，然后在实际执行中会在 Get1D() 和 Get2D() 函数中执行相关的变异（前面我们详细描述过）。

给定标量贡献函数的值，由于原始样本空间上突变的对称性，接受概率计算就会简化。如本节开头所述，光谱辐射亮度值 $L(\mathbf{X})$ 必须转换为标量贡献函数给出的值，以便可以计算 Metropolis 采样算法的接受概率，这里我们计算路径的亮度是一个合理的选择。

其他步骤都很浅显易懂，所以就不再赘述了。

3.3.3 存储最终图像

在最后写入图像时，需要将样本值都用因子加权。

首先需要像素值乘以 b，原因可见第一章的公式。

运行 nChains 马尔科夫链中的每个 Metropolis 迭代都加权单位亮度对胶片 splat 贡献，从而使 Film::WriteImage() 之前的最终平均胶片亮度完全等于每个像素的变化（最终结果需要再除以 mutationsPerPixel）。

所以最后缩放因子就是 (b/mutationsPerPixel)。

4. 总结

本章对本文和 *PBRT* 系列电子书做一个简单的总结和概括。

4.1 总结

疫情重新开放后的第二天，大概还要工作一两周，也算是个特殊的时间点。

PBRT 系列从 2021 年 1 月 24 日就开始正式写作，直到 2022 年年底才即将结束，今天是 12 月 14 日。中间因为很多事情，包括科研上和生活上，所以一直断断续续的，虽然本系列也肯定会存在不少问题，尤其是在言语组织上，但也算是本人的投入不少精力的心血之作，不免有些感慨。

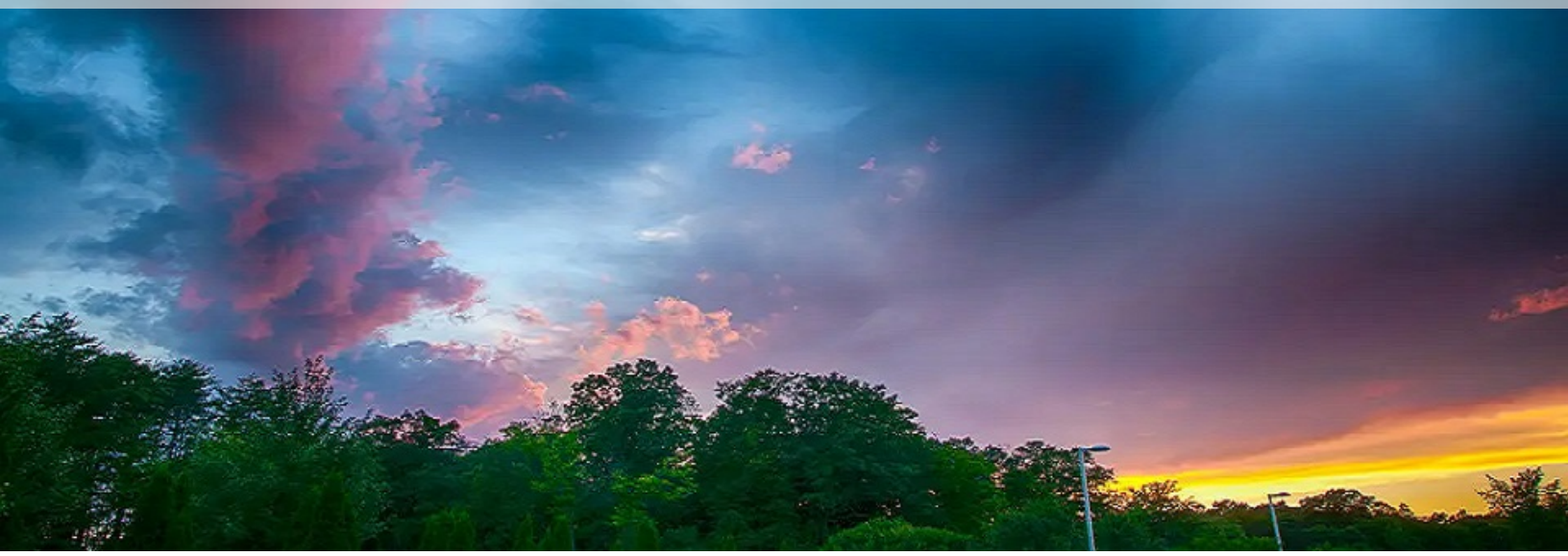
以后的规划，一方面我会做一个 Mitsuba 的精讲，另一方面我会做一个论文精讲系列，解释和详细分析一下最近这些年来一些大佬组们在 SIGGRAPH 和 TOG、CGF 等期刊上发表的重要工作。并且对一些相对专业的技术，比如布料渲染、各向异性材质渲染以及毛发渲染等做一些介绍和代码实现。这些就会涉及到一些更实际的物理模型。

至于图形学的科研方面，虽然我认为图形学在顶刊的发表已经完全变成了一个互惠互利的小圈子，圈外的人很难分一杯羹，但从技术角度，里面还是出现了不少有意义和价值的技术的。由于自己条件所限（无论是人脉上还是科研条件上），很多工作都很难去探索，尤其是最近一直很火的神经辐射场或者可微渲染领域，自己能使用的数据和硬件资源实在太少，深感无力。自己写 PBRT 系列一方面也是为了尽可能帮助像我一样想努力融入图形学圈子，但是没有人指导，只能自己一点一点去查资料和摸索的人罢了。

另外，由于本系列的非盈利性质，使得我可以把更多重点放在内容组织和安排，而不是刻意处处避开别人的解释。有些地方网上或其他书籍基于的解释我认为非常棒，我就会直接在书中引用，并且附带参考文献目录或网站地址。

最后，感谢大家能够坚持看这个系列，也非常感谢一些购买电子书的网友。虽然这些钱对我而言并不算什么经济来源，但如果有人愿意为此支付一定的金额，不论多少，都能够让我获得很多成就感，也是我能够坚持创作下去的动力。也祝大家能在计算机图形学或其他科研中找到自己的科研方法，建立好自己的一套方法论和科研体系！

参考文献



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

[2] Veach E . Robust Monte Carlo Methods for Light Transport Simulation[J]. Ph.d.thesis Stanford University Department of Computer Science, 1998.