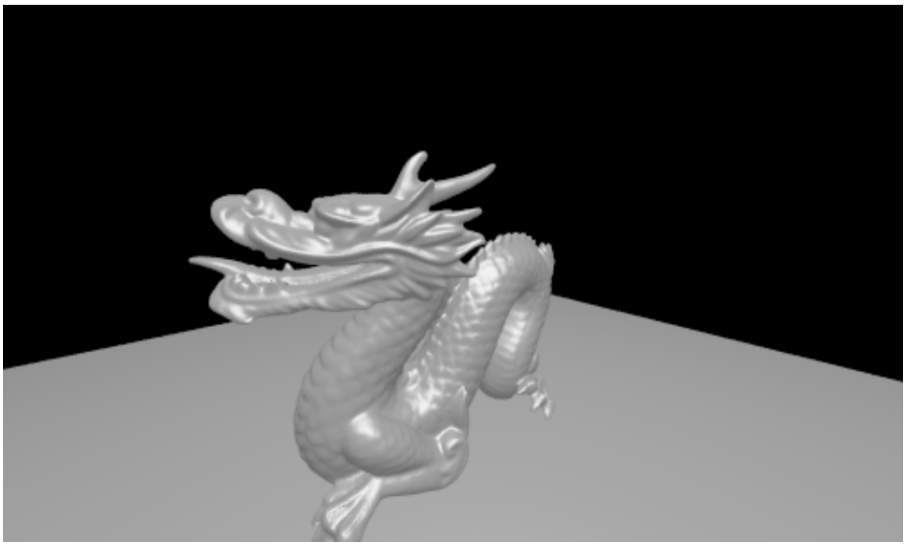
PBRT 系列 7-采样器与渲染器

Dezeming Family

2021 年 3 月 3 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，可以从官方网站：https://dezeming.top/ 下载新的版本（免费下载）。

本书目标：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：



本文于 2022 年 7 月 8 日进行再版，提供了源码。注意图形 GUI 界面和本文中展示的有点区别，但并不影响学习。源码见网址 [https://github.com/feimos32/PBRT3-DezemingFamily]。

前言

本书我们把采样器和渲染器流程学习和移植一下。

之前我们输出随机数都是使用的基于时钟产生的随机序列，在渲染时，该序列可能会产生聚集的随机数，比如大量随机数聚集在 0 附近，造成很差的视觉效果。我们的第二本书是在 [2][3][4] 书中实现的微型渲染器改造成的符合 PBRT 风格的 Whitted 渲染器，最后我们发现自己的渲染器渲染一帧的效果比 PBRT 的渲染器要差了很多，这是因为我们使用的并不是更适用于渲染的低差异序列（提一句，质量差跟光线微分技术无关，因为没有纹理采样）。

本书我们首先在我们构建的新的渲染器上实现采样器，然后我们回顾 Whitted 光线追踪器，移植和实现 PBRT 的积分器类到自己的系统中，对比使用时钟随机数和使用低差异序列渲染一帧的效果。本书不会对采样器做多么深的讲解，甚至是一笔带过。PBRT 是一个很复杂的系统，而我们的目标是让读者尽可能快地掌握和使用 PBRT，以及使用 PBRT 搭建一个自己的引擎，因此里面涉及到的很多算法细节我不会一上来就开始讲那么多，但不代表它们不需要了解（其实对于图形学而言，低差异序列的原理了解一下还是很有必要的）。

我们以前是借助了 [2][3][4] 中构建的引擎实现的 Whitted，如今，我们已经实现了自己的 PBRT 底层，我们已经完全抛弃 [2][3][4] 中使用的流程了，我们现在搭建的渲染器是完全遵从 PBRT 渲染器的版本。

本书的售价是 4 元（电子版），但是并不直接收取费用。如果您免费得到了这本书的电子版，在学习和实现时觉得有用，可以往我们的支付宝账户（17853140351，可备注：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！

目录

一 PBRT 中的采样器

自然界的图像是连续的，但是显示器的分辨率有限，所以需要使用有效的采样方法来显示大自然的图像。PBRT 中从连续值中采样的方法主要依赖于采样定理和低差异序列（分布比较均匀的序列）。PBRT 中一共介绍了 5 个采样器（分别对应 [1] 的 7.3 到 7.7 章），不过我们现在只移植和使用里面的一个就够了。

图像重建需要的类是 Filter，该类表示的是重建一副图像时，一个像素的着色值如何加权平均。之后，Film 负责将图像样本的贡献计算到像素中，我们在《相机系统》中简单看过 film 类，该类有成员变量 std::unique_ptr<Filter> filter。但图像重建并不是我们重点，我当前并不打算介绍它怎么重建的，因此，每个像素采样到什么值我们最后显示的时候就赋值给它什么值就行了。至于从渲染结果到实际成像结果的转换，各位有兴趣的话可以看看图像 Denoise，有各种高级而且效果还不错的技术。

我们本章学习 PBRT 使用的采样器的流程和函数接口。我提前说一句，本章内容会比较抽象，您可能不明白这里的函数接口究竟什么用，但是在这一章您可以先混个脸熟，下一章我会告诉大家 PBRT 渲染流程中采样器的使用过程，到时候大家就一目了然了。

1 1 采样器概述

我们需要的样本范围一般是要求在 $[0,1)$，之间，要保证严格小于 1，因此我们需要定义最大上界 OneMinusEpsilon，该值表示小于 1 的最大浮点数。

采样器最简单的实现就是返回 [0,1) 中的均匀随机值，这样的采样器可以产生正确的图像，但是需要更多的样本（因此，需要更多的光线跟踪路径和更多的时间）来创建与更复杂的采样器相同质量的图像。使用更好的采样模式的运行时开销与使用均匀随机数等质量较低的模式的运行时开销大致相同；因为评估每个图像样本的辐射度比生成随机值更耗费时间得多，所以做这项工作会带来好处。

我们可以把渲染用到的随机数表示为一个多维函数：

$$(x, y, t, u, v, [(u_0, u_1), (u_2, u_3), (u_4, u_5), (u_6, u_7)]) \tag{一.1}$$

其中，(x,y) 表示在图像像素上选择一个点，t 表示选择一个随机的时刻（运动模糊），(u,v) 表示在镜头上选择一个点（景深）。后面的 (u_n, u_{n+1}) 表示的是在渲染积分器中使用的随机数（比如产生随机方向的反射等）。我们认为多维函数中前面维度的采样重要性大于后面的维度，可能有的采样器更适合高维采样，但是有的采样器对低维更友好。

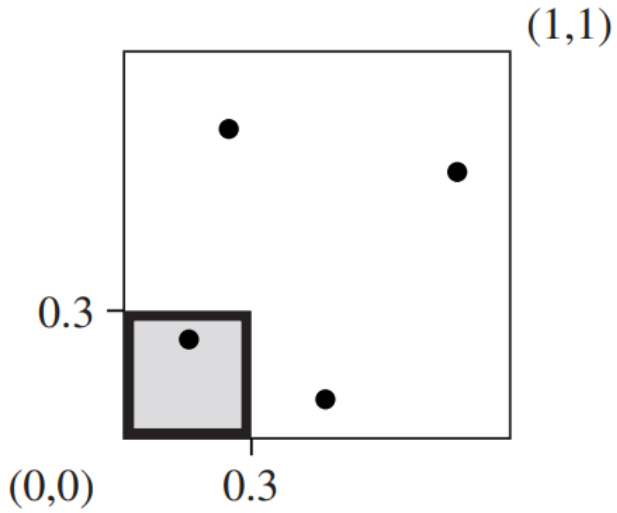
还要注意的是，由采样器生成的 n 维样本经常不显式地表示或全部存储，而是通常根据光传输算法的需要以增量的方式生成。

1 2 样本的差异

差异性是评估随机样本的方法。

数学家们提出了一个称为差异的概念，可以用来评估 n 维样本位置的质量。分布均匀的模式具有低差异值，因此样本生成问题可以被认为是找到一个合适的低差异点模式的问题之一。人们已经开发了许多确定性技术来生成低差异点集，包括在高维空间（后面使用的大多数采样算法都使用这些技术）。

所谓描述差异性的基本思想是，在 n 维空间 $[0,1)^n$ 中的一组点的质量可以通过观察 $[0,1)^n$ 内划分的更小的区域，计算每个区域内的点的数量，并将每个区域的体积（多维变量的超体积）与内部的样本点的数量进行比较来评估。一般来说，给定的体积分数应该与其中采样点总数的分数大致相同。虽然不可能总是这样，但我们仍然可以尝试使用使实际体积和由点估计的体积（差异）之间的最大差异最小化的模式。



如上图，小区域内有 1 个点，总共有 4 个点，所以估计面积为 0.25，但实际面积是 0.09。

1 3　采样器接口

样本的基类是 Sampler，在 core 文件夹的 sampler.cpp 和 sampler.h 文件里定义。所有采样器必须告诉它每个像素在最终图像上需要产生多少样本（有时候甚至可以认为整个成像面是 1 个像素构成的，需要采样海量的样本）。

当渲染算法准备好对给定的像素开始工作时，它首先调用 StartPixel()，提供图像中像素的坐标。一些采样器使用哪个像素被采样的信息来改进它们为该像素生成的样本的总体分布，而另一些采样器实现则忽略此信息。

采样器的成员函数 Get1D() 和 Get2D() 分别产生 1 维和 2 维的随机向量。但在一次渲染中，可能我们要产生 n 维的随机数，必须仔细编写使用样本值的代码，以便它总是以相同的顺序请求样本维度。考虑以下代码：

```
1    //如果这样的话，if判断失败，则当前循环只读入了2个低差异序列，可能打乱样
         本的低差异性
2    sampler->StartPixel(p);
3    do {
4        Float v = a(sampler->Get1D());
5        if (v > 0)
6            v += b(sampler->Get1D());
7        v += c(sampler->Get1D());
8    } while (sampler->StartNextSample());
9    //我们可以这么做，把固定维度数的样本都读出来，这样的话采样器就不会被打乱
         维度
10   sampler->StartPixel(p);
11   do {
12       //同时把多个随机数采样出来，不管是否会使用
13       //但这样不是好主意，因为有时候我们可能渲染一轮就终止了，但有的渲染需
             要好几轮，总共需要几十维的随机数，全都读出来对于渲染1轮就结束的情
             况非常浪费时间
14       Float v = a(sampler->Get1D());
15       Float b1 = sampler->Get1D();
16       Float c1 = sampler->Get1D();
17       if (v > 0)
18           v += b(b1);
```

```
        v += c(c1);
    } while (sampler->StartNextSample());
```

在这种情况下，样本 vector 的第一个维度将始终传递给函数 a()；当执行调用 b() 的代码路径时，b() 将接收第二个维度。但是，如果 if 测试不总是 true 或 false，那么 c() 有时会从样本向量的第二维度接收样本，否则会从第三维度接收样本。因此，采样器将不能为计算的每个维度提供均匀分布的样本点。因此，应该仔细编写使用采样器的代码，以便它始终使用同样的样本维度来避免这个问题。

比如，我们每个像素采样 100 次光路径，每次采样器最先生成的两个随机数就是相机在一个像素内的位置 (x, y)，我们希望这一百次采样到的位置 (x, y) 能够符合低差异特征，这就需要尽量保持相同的样本维度。通常就是，通过低差异技术生成 x 的 100 个样本，然后生成 y 的一百个低差异样本，以此类推。这也是为什么 PixelSampler::Get1D() 中：

```
    samples1D[current1DDimension++][currentPixelSampleIndex];
```

每次采样光路径时，每获取下一个样本则要在第 0 个维度计数加 1（即 current1DDimension++）——注意，以上面为例，100 个 x 样本和 y 样本也就是：

```
    samples1D[0][N];
    samples1D[1][N];
```

其中，$N \in [0, 100-1]$。

Request1DArray(int n) 和 Request2DArray(int n) 会直接生成低差异序列样本数组（要在渲染前就生成）。但是一些采样器更适合生成其他数量的低差异样本数，比如 ZeroTwoSequenceSampler 更适合生成 2 的幂的样本数，因此程序会修改传入的 n，想知道到底生成了多少样本可以使用 Sampler::RoundCount() 方法。Get1DArray 和 Get2DArray 可以获得生成的样本数组。在直接采样光中，假如我们希望对面光采样 10 个点，那么这 10 个样本点就需要尽可能均匀分布，因此我们需要 Request2DArray 来生成 10 个低差异序列样本。

1 4　采样器基类工作细节

下面的内容听起来可能有些抽象，等我们在源码中遇到时就明白是什么意思了，这里先初步有个印象。

当一个样本的工作完成时（即一个像素渲染完一帧时），调用 StartNextSample，通知采样器对样本组件的后续请求，应返回从当前像素的下一个样本的第一个维度开始的值（否则就被打乱了）。此方法返回 true，直到生成每个像素最初请求的样本数为止（此时调用者应开始处理另一个像素或停止使用更多样本）。

采样器存储当前样本的各种状态：哪个像素被采样，样本的多少维度被使用等等。因此，由多个线程同时使用单个采样器是不安全的。Clone() 方法生成一个初始采样器的新实例，供渲染线程使用；它为采样器的随机数生成器（如果有）获取一个种子值，以便不同的线程看到不同的随机数序列，因为如果多个图像块重用相同的伪随机数序列可能会导致细微的图像伪影，例如重复的噪声模式。一些光传输算法（特别是随机渐进光子映射）在进入下一个像素之前不使用一个像素中的所有样本，而是在像素周围跳跃：每次在每个像素中取一个样本。SetSampleNumber() 方法允许积分器设置当前像素中样本的索引以生成下一个样本。一旦 sampleNum 大于或等于每个像素最初请求的采样数，此方法将返回 false。

Sampler 基类在其接口中提供了一些方法的实现。首先，开始渲染第一帧图像之前，StartPixel() 方法传入当前采样像素的坐标，并将当前生成的像素中的采样数 currentPixelSampleIndex 重置为零。像素内的当前像素坐标和采样数可作为参数供采样器子类使用，但它们应被视为只读值。与 StartPixel() 一样，StartNextSample() 和 SetSampleNumber() 方法都是虚函数，子类继承以后会定义新的内容。

Sampler 基类实现还负责记录对样本数组的请求，并为它们的值分配存储。请求的样本数组的大小存储在 Samples1ArraySizes 和 Samples2ArraySizes 中，整个像素的数组样本的内存分配在 sampleArray1D 和 sampleArray2D 中。每个分配中的前 n 个值用于像素中第一个样本的对应数组，依此类推。

1 5　采样器子类-像素采样器和全局采样器

像素采样器

虽然一些采样算法可以轻松地以增量方式生成每个采样向量的元素，但其他一些算法更容易同时为一个像素的所有样本 vector 生成所有维度的随机值——PixelSampler 类就是这种类型的采样器的基类。这类采样器，比如：

```
class MaxMinDistSampler : public PixelSampler
class StratifiedSampler : public PixelSampler
```

渲染算法将使用的样本 vector 的维数事先是未知的（事实上，在渲染中只会根据 Get1D() 和 Get2D() 调用的数量以及请求的样本 Array 隐式地确定）。因此，PixelSampler 构造函数接受最大数量的维数，采样器将为这些维数计算非数组采样值。如果组件的所有这些维度都被消耗，那么 PixelSampler 只会为其他维度返回均匀的随机值：见 Get1D() 和 Get2D() 函数，这两个函数如果申请的随机数数量超过了预设值，则会调用 RNG 类来产生均匀随机数：

```
float PixelSampler::Get1D() {
    if (current1DDimension < samples1D.size())
        return samples1D[current1DDimension++][currentPixelSampleIndex];
    else
        return rng.UniformFloat();
}
```

对于每个预计算的维度，构造函数分配一个 vector 来存储样本值，像素中的每个样本对应一个值。这些向量被索引为 sample1D[dim][pixelSample]；虽然交换这些索引的顺序似乎更为合理，但这种内存布局（给定维度的所有样本的所有样本分量值在内存中是连续的）对于生成这些值的代码来说更为方便。

全局采样器

其他生成样本的算法基本上不是基于像素的，而是自然生成连续的样本，这些样本分布在整个图像上，连续访问完全不同的像素，GlobalSampler 就是这类采样器的基类。这类采样器，比如：

```
class SobolSampler : public GlobalSampler
class HaltonSampler : public GlobalSampler
```

这里就不再介绍它的实现机制了，有兴趣可以去 PBRT 书上的第七章参考一下。

我们下一章会跟踪采样器实际运行的代码，让读者对其有更充分的认识。

二　采样器的工作流程

采样器最初保存在哪里呢？

在《文件加载与设定》书中，我们知道 pbrtWorldEnd() 函数启动了渲染过程，MakeIntegrator() 函数根据参数创建了渲染器：

```
1    std::shared_ptr<Sampler> sampler =
2        MakeSampler(SamplerName, SamplerParams, camera->film);
3    //创建路径追踪器
4    if (IntegratorName == "path")
5        integrator = CreatePathIntegrator(IntegratorParams, sampler, camera)
             ;
```

Integrator 是所有积分器类的基类，SamplerIntegrator 类继承自 Integrator 类。注意 Render 函数在 SamplerIntegrator 类中就有实现，DirectLightingIntegrator 类和 PathIntegrator 类、VolPathIntegrator 类以及 WhittedIntegrator 类都是继承自 SamplerIntegrator 类的派生类。而 SPPMIntegrator 类、BDPTIntegrator 和 MLTIntegrator 类是直接继承自 Integrator 类的派生类，它们都实现了自己的 Render 函数。

需要注意的是，有些采样器定义的函数，例如 Request1DArray 和 Get1DArray，PBRT 程序中并没有用到，而 Request2DArray 和 Get2DArray 也只在环境光遮蔽渲染器 AOIntegrator 和直接光照渲染器 DirectLightingIntegrator 中使用到了。

2 1　Render 函数中的采样分配

SPPMIntegrator 等双向方法的积分器类过于复杂，我们暂时先不考虑，我们当前只研究 SamplerIntegrator 类中的 Render 函数的实现。

见 integrator.cpp 文件的函数：

```
1    void SamplerIntegrator::Render(const Scene &scene){
2        //分配n个tiles块，放入多线程渲染
3        ......
4        {
5            //启动多线程渲染
6            ParallelFor2D([&](Point2i tile)){
7                //开始渲染
8                ......
9            }
10       }
11       //保存图像
12       ......
13   }
```

在每个线程块开始渲染以后，为每个块提供一个采样器实例：

```
1    std::unique_ptr<Sampler> tileSampler = sampler->Clone(seed);
2    //遍历块中的每个像素点
3    for (Point2i pixel : tileBounds){
4        {
5            //根据每个像素点初始化采样器
6            tileSampler->StartPixel(pixel);
7        }
```

```
        do{
            // Initialize _CameraSample_ for current sample
            //会产生前5维的随机数，即像素采样点2维+相机时间1维+镜头采样2维
            CameraSample cameraSample =
                tileSampler->GetCameraSample(pixel);
            // Generate camera ray for current sample
            RayDifferential ray;
            Float rayWeight =
                camera->GenerateRayDifferential(cameraSample, &ray);
            ray.ScaleDifferentials(
                1 / std::sqrt((Float)tileSampler->samplesPerPixel));
            ++nCameraRays;
            // Evaluate radiance along camera ray
            Spectrum L(0.f);
            if (rayWeight > 0) L = Li(ray, scene, *tileSampler, arena);
        }while (tileSampler->StartNextSample());
    }
```

samplesPerPixel 就是我们在场景文件中设置的一个像素采样多少个点。可以看到我们上一章讲述过的 StartPixel 和 StartNextSample。假如我们渲染一个像素的一个样本是一个 13 维的随机向量（如下，我们上一章提到过），而我们每个像素需要渲染 64 个样本点，则我们可以知道，sample1D[dim][pixelSample] 中，dim 是 13，pixelSample 是 64。

$$(x, y, t, u, v, [(u_0, u_1), (u_2, u_3), (u_4, u_5), (u_6, u_7)]) \tag{二.1}$$

samples1D 和 samples2D 是两个 vector:

```
    std::vector<std::vector<Float>> samples1D;
    std::vector<std::vector<Point2f>> samples2D;
```

2 2　渲染算法中使用的采样器

以 Whitted 渲染器为例，我们来看一下它使用的采样器：

```
    Spectrum WhittedIntegrator::Li(const RayDifferential &ray, const Scene &
        scene,
    Sampler &sampler, MemoryArena &arena,
    int depth);
```

Li() 函数中使用的采样器是用来采样光源上的随机点的：

```
    Spectrum Li =
    light->Sample_Li(isect, sampler.Get2D(), &wi, &pdf, &visibility);
```

我们现在已经知道了采样器的流程，下一章就开始介绍如何移植和使用。

三　不同采样器的描述和移植

本章重点是将 PBRT 中的采样器应用到我们自己的系统上去。我们只移植一个 halton 采样器，其他采样器都可以照搬就好了。

虽然 PBRT 书 [1] 中关于采样器的理论有一大堆，但是使用的话还是非常简单的，如果想学习一下内部原理，可以从网上看各种与低差异序列有关的博客和论坛，有很多非常通俗易懂的解释，这里不再涉及。

3 1　采样器的移植

我们首先移植 sobolmatrices.h 和 sobolmatrices.cpp 两个文件，没什么可说的，直接复制过来就好了。

然后是 lowdiscrepancy.h 和 lowdiscrepancy.cpp 两个文件，用于产生低差异序列，以及 sampling.h 和 sampling.cpp 文件。注意这些文件里面的函数我们不必全都移植到自己系统里，而是用到里面哪个变量或者函数就拷贝哪个。

Sampler、PixelSampler 和 GlobalSampler 是可以直接移植上来的，只需要将里面的 Float 写为小写 float（同以前一样，我们不考虑自定的浮点数或者有时候选择用 double 代替 float），另外根据自己的需求删除或者修改调试语句。

移植的过程会比较复杂和麻烦，有许多 PBRT 预定义的量需要修改。

CreateHaltonSampler 函数我们这么移植：

```
1  HaltonSampler *CreateHaltonSampler(const Bounds2i &sampleBounds) {
2      //我们是一轮一轮渲染的，每轮只使用一个样本点
3      int nsamp = 1;
4      bool sampleAtCenter = false;
5      return new HaltonSampler(nsamp, sampleBounds, sampleAtCenter);
6  }
```

sampleBounds 来自于 MakeSampler 函数（api.cpp）传入的参数 film->GetSampleBounds()。这里样本的边界我们就按照生成图像分辨率的最大值和 0 来表示了，即如果我们生成的图是 500*300 的，那么 sampleBounds.pMin=(0,0),sampleBounds.pMax=(500,300)。

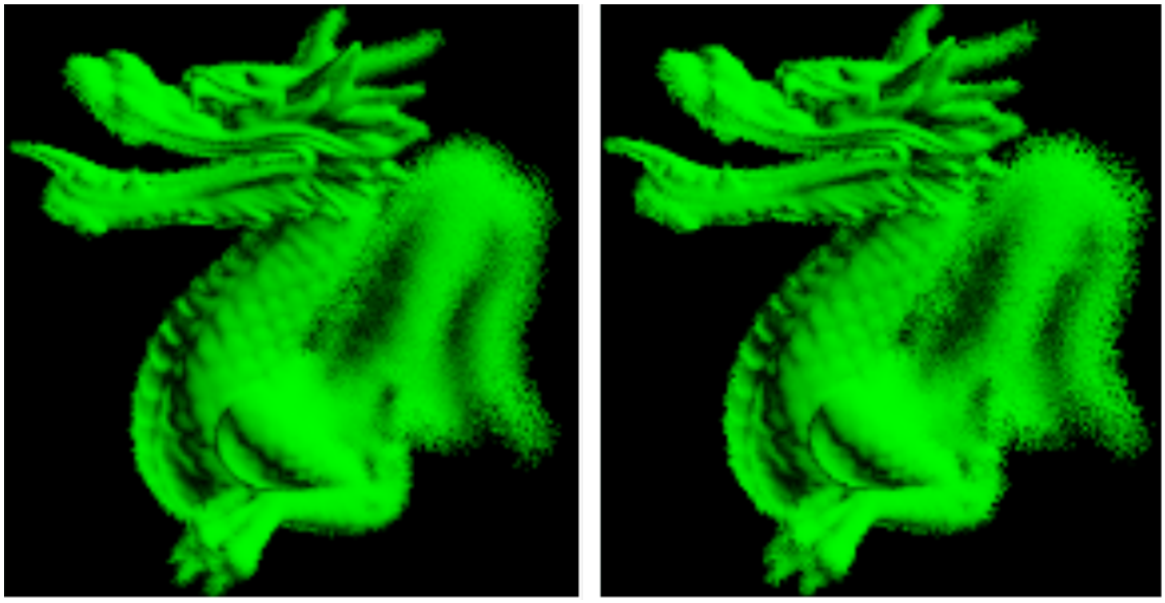
3 2　采样器的初步使用

我们做一个简单的测试：

```
1      Bounds2i imageBound(Point2i(0, 0), Point2i(RasterWidth, RasterHeight
           ));
2      std::shared_ptr<HaltonSampler> hns= std::make_unique<HaltonSampler
           >(8, imageBound, false);
3      for (int j = 0; j < RasterHeight; j++) {
4          for (int i = 0; i < RasterWidth; i++) {
5              int offset = (i + RasterWidth * j);
6              std::unique_ptr<Sampler> pixel_sampler = hns->Clone(
                   offset);
7              Point2i pixel(i, j);
8              pixel_sampler->StartPixel(pixel);//开始准备随机数
9              Spectrum colObj(0.0f);
10             do{
11                 CameraSample cs;
12                 cs= pixel_sampler->GetCameraSample(pixel);//
                       随机数初始化相机采样点
```

```
                                Ray r;
                                cam->GenerateRay(cs,&r);
                                float tHit;
                                SurfaceInteraction isect;
                                if (agg->Intersect(r, &isect)){
                                        float Li = Dot(Light, isect.n);
                                        colObj[1] += std::abs(Li);
                                }
                        } while (pixel_sampler->StartNextSample());//开始下
                            一个样本的采样
                        colObj[1] = colObj[1] / (float)pixel_sampler->
                            samplesPerPixel;
                        //更新到缓冲区
                ......
                    }
        }
```

实现的结果如下，我们对比不使用低差异序列的相机参数生成和使用低差异序列的相机参数生成，每个像素渲染 8 个样本：



左边是 halton 采样器生成的结果，右边是时钟产生的随机数生成的结果，可以看到左边比右边均匀一些。

3 3　基于时钟的随机数

由于我们的 GUI 允许我们每次只渲染一帧然后逐帧把结果累加，所以前面的低差异序列我们难以使用。我们包装一个类 ClockRandSampler，见 clockRand.h 和 clockRand.cpp 文件，实现了里面的一些功能接口：

```
std::unique_ptr<Sampler> Clone(int seed) {
    return std::unique_ptr<Sampler>(new ClockRandSampler(*this));
}
int64_t GetIndexForSample(int64_t sampleNum) const {
    return 0;
}
float SampleDimension(int64_t index, int dimension) const {
```

```
8       return getClockRandom();
9   }
```

这样就能像使用 PBRT 类一样使用时钟随机数了。

四　渲染积分器

我们虽然在《文件的加载与设定》和《Whitted 光线追踪器》中已经学习过渲染积分器了，但仅仅是初步了解，哪怕最终我们实现了一个简单的渲染积分器，也是基于 [2][3][4] 引擎上实现的。现在我们开始实现真正的 PBRT 渲染积分器。

4 1　Scene 类的作用和移植

PBRT 的积分器需要输入场景 Scene 对象（scene.h 和 scene.cpp 文件）：

```
if (scene && integrator) integrator->Render(*scene);
```

因此我们先实现 Scene 类到自己的引擎上，我移除了里面与灯光类有关的成员变量，IntersectTr 与体渲染透光有关，我们也暂时移除。

```
class Scene {
    public:
        // Scene Public Methods
        Scene(std::shared_ptr<Primitive> aggregate)
            : aggregate(aggregate) {
            // Scene Constructor Implementation
            worldBound = aggregate->WorldBound();
        }
        const Bounds3f &WorldBound() const { return worldBound; }
        bool Intersect(const Ray &ray, SurfaceInteraction *isect) const;
        bool IntersectP(const Ray &ray) const;
    private:
        // Scene Private Data
        std::shared_ptr<Primitive> aggregate;
        Bounds3f worldBound;
};
```

两个求交的方法都是调用的 aggregate 的求交方法：

```
bool Scene::Intersect(const Ray &ray, SurfaceInteraction *isect) const {
    ++nIntersectionTests;
    return aggregate->Intersect(ray, isect);
}
bool Scene::IntersectP(const Ray &ray) const {
    ++nShadowTests;
    return aggregate->IntersectP(ray);
}
```

我们测试一下 Scene 类是否能够正确运行：

```
std::unique_ptr<Scene> worldScene;
//std::vector<std::shared_ptr<Primitive>> prims;是基元数组
//std::make_shared<BVHAccel>构造了加速结构
worldScene = std::make_unique<Scene>(std::make_shared<BVHAccel>(prims,
    1));
if (worldScene->Intersect(r, &isect)){
    //测试到相交
```

```
        ......
    }
```

正确渲染得到之前的图像即说明移植成功。

4 2 渲染积分器基类的移植

Integrator 基类（integrator.h 头文件）非常简单，直接拷贝到自己的引擎上即可。

SamplerIntegrator 类我们先只保留下面的这些函数，之后会慢慢填充进去：

```
// SamplerIntegrator Declarations
class SamplerIntegrator : public Integrator {
public:
    // SamplerIntegrator Public Methods
    SamplerIntegrator(std::shared_ptr<const Camera> camera,
        std::shared_ptr<Sampler> sampler,
        const Bounds2i &pixelBounds)
        : camera(camera), sampler(sampler), pixelBounds(pixelBounds) {}
    virtual void Preprocess(const Scene &scene, Sampler &sampler) {}
    void Render(const Scene &scene);
protected:
    // SamplerIntegrator Protected Data
    std::shared_ptr<const Camera> camera;
private:
    // SamplerIntegrator Private Data
    std::shared_ptr<Sampler> sampler;
    const Bounds2i pixelBounds;
};
```

我们只有一个需要实现的函数，也就是：

```
    void Render(const Scene &scene);
```

在该函数里我们可以把之前的渲染函数都塞到 Render 函数里：

```
        Vector3f Light(1.0, 1.0, -1.0);
        Light = Normalize(Light);
        for (int j = 0; j < RasterHeight; j++) {
                for (int i = 0; i < RasterWidth; i++) {
                        int offset = (i + RasterWidth * j);
                        std::unique_ptr<Sampler> pixel_sampler = sampler->
                            Clone(offset);
                        Point2i pixel(i, j);
                        pixel_sampler->StartPixel(pixel);
                        Spectrum colObj(0.0f);
                        do {
                                CameraSample cs;
                                cs = pixel_sampler->GetCameraSample(pixel);
                                Ray r;
                                camera->GenerateRay(cs, &r);
                                float tHit;
```

```
                                SurfaceInteraction isect;
                                if (scene.Intersect(r, &isect)) {
                                        float Li = Dot(Light, isect.n);
                                        colObj[1] += std::abs(Li);
                                }
                        } while (pixel_sampler->StartNextSample());
                        colObj[1] = colObj[1] / (float)pixel_sampler->
                            samplesPerPixel;
                //保存到图像缓存或保存图像
                ......
                    }
            }
```

需要注意的是，我们之前提到过，PBRT 的图像是保存在 Camera 对象的 Film 成员变量里。我们可以按照自己系统的方法来保存，比如我是直接显示在 GUI 界面上。

我们把构造积分器的函数实现一下：

```
    //将下面的内容进行初始化
    std::unique_ptr<Scene> worldScene;
    std::shared_ptr<Camera> camera;
    std::shared_ptr<Sampler> sampler;
    std::shared_ptr<Integrator> integrator;
    //构造积分器:
    integrator = std::make_shared<SamplerIntegrator>(camera, sampler,
        ScreenBound);
    //渲染函数:
    integrator->Render(*worldScene);
```

需要注意的是，halton 构造器输出序列对于固定的种子值，顺序是固定的：

```
    //offset是根据像素位置提供的种子值
    std::unique_ptr<Sampler> pixel_sampler = sampler->Clone(offset);
    //对于固定的offset，使用pixel_sampler产生的随机数序列也是固定的
```

因此我们同时调用两次 Render 函数以后，产生的结果是完全一样的。

五　积分器的使用和测试

在《Whitted 光线追踪引擎》一书，我们曾经说过，所有的灯光类，只要是面光源，也会被存储在加速结构中。Whitted 光线追踪分为三步，第一步检测场景如果相机没有和物体相交，则返回环境光的值（如果有环境光）。第二步计算反射的 BxDF，并判断该物体是否为发光物。第三步遍历所有光源，计算光照量。如果有镜面物体就执行镜面反射或折射。

5 1　场景设置

我们搭建一下场景，除了最初的三角面片化物体，我们再添加一个地板：

```
1        //地板
2        int nTrianglesFloor = 2;
3        int vertexIndicesFloor[6] = { 0,1,2,3,4,5 };
4        int nVerticesFloor = 6;
5        const float yPos_Floor = -2.0;
6        Point3f P_Floor[6] = {
7                Point3f(-6.0,yPos_Floor,6.0),Point3f(6.0,yPos_Floor,6.0),
                     Point3f(-6.0,yPos_Floor,-6.0),
8                Point3f(6.0,yPos_Floor,6.0),Point3f(6.0,yPos_Floor,-6.0),
                     Point3f(-6.0,yPos_Floor,-6.0)
9        };
10       Transform tri_Object2World2, tri_World2Object2;
11       std::shared_ptr<TriangleMesh> meshFloor = std::make_shared<
             TriangleMesh>
12               (tri_Object2World2, nTrianglesFloor, vertexIndicesFloor,
                     nVerticesFloor, P_Floor, nullptr, nullptr, nullptr,
                     nullptr);
13       std::vector<std::shared_ptr<Shape>> trisFloor;
14       for (int i = 0; i < nTrianglesFloor; ++i)
15               trisFloor.push_back(std::make_shared<Triangle>(&
                     tri_Object2World2, &tri_World2Object2, false, meshFloor,
                     i));
16       //将物体填充到基元
17       for (int i = 0; i < nTrianglesFloor; ++i)
18           prims.push_back(std::make_shared<GeometricPrimitive>(trisFloor[i
                 ]));
```

另外我发现我们的三角面片求交函数里并没有设置相交点位置，我们补充上：

```
1        isect->p = b0 * p0 + b1 * p1 + b2 * p2;
2        isect->n = Normal3f(Normalize(Cross(p1-p0,p2-p0)));
```

之后我们会用到这个位置。

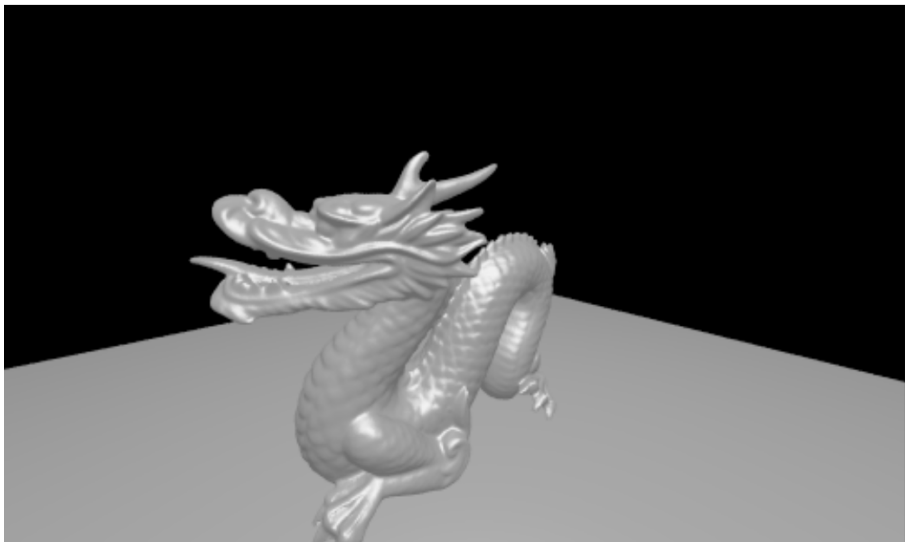
5 2　Blinn-Phong 模型渲染

基于点光源或方向光源的 Blinn-Phong 是最简单的渲染模型之一了。其实我本想实现面光源蒙特卡洛渲染的，但我发现没有材质信息确实有点不太好办，因此我们就实现一个基于 Blinn-Phong 的渲染效果。

这里不讲什么是 Blinn-Phong 模型了，过于简单，直接附上实现代码：

```
    Point3f Light(-2.0, 4.0, -3.0);
    if (scene.Intersect(r, &isect)) {
        Vector3f LightNorm = Light - isect.p;
        LightNorm = Normalize(LightNorm);
        Vector3f viewInv = -r.d;
        Vector3f H = Normalize(viewInv + LightNorm); //半程向量
        //高光
        float Ls = Dot(H, isect.n); Ls = (Ls > 0.0f)? Ls:0.0f;
        Ls = pow(Ls, 32);
        //漫反射光
        float Ld = Dot(LightNorm, isect.n); Ld = (Ld > 0.0f) ? Ld : 0.0f;
        float Li = (0.2 + 0.2 * Ld + 0.6*Ls);
        colObj += Spectrum(Li);
    }
```

得到如下渲染结果：



注意我们提供的代码的显示效果与本文中的描述有些不符，但对于 PBRT 移植的部分则是完全一样的，并不影响读者学习和使用。

六　本书结语

这本书说难也不难，说简单也不简单。关于低差异序列的原理（n 进制反转之类的）我并没有讲，但就算不知道原理，我们也知道了低差异序列的好处以及如何使用低差异序列。

我们学习了基本的渲染器接口，但是并没有实现最简单的 Whitted 光线追踪器到新系统上，是因为我们需要同时实现面光源和材质，会比较麻烦。因此我们只实现了一个简单的 Blinn-Phong 照明模型。

读者可能感受到，我们已经构建了一个有模有样的渲染器了——恭喜您已经从这套系列书中得到了真传。因为篇幅和本人时间原因，很多细节我都没有仔细讲解，但配合着 [1] 和各种网络资料，学起来应该也没有那么难。

下一本书，就到了 PBRT 的重头戏了——反射与材质。还是以知识讲解和移植实现为主，尽量能让读者更容易的学习基本的框架。

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.