PBRT 系列 23-专业知识理论与代码实战-图像重建与滤波

Dezeming Family

2022 年 11 月 22 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）。

源码见网址 [https://github.com/feimos32/PBRT3-DezemingFamily]。

前言

本文并不是为了描述一些数字滤波器，而是为了将 PBRT-V3 渲染器的整个图像构建流程描述清楚。

本文将从相机发射光线开始，不讲解实际渲染算法，而是在计算得到 Li 值以后，从图像滤波直到最终生成图像 buffer 的过程。因此，本文是对 PBRT 图像生成和像素滤波过程的一个串联。

目录

一 渲染启动与图像分辨率 **1**

二 图像重建 **3**

2 1 采样与滤波 3

2 2 Filter 类 4

三 相机胶片的工作 **5**

3 1 胶片类 5

3 2 把像素值提供给 Film 7

3 2.1 GetFilmTile() 函数 7

3 2.2 AddSample() 函数 8

3 2.3 MergeFilmTile() 函数 8

3 3 图像输出 9

四 整体步骤串联与总结概述 **10**

4 1 步骤串联 10

4 2 总结 10

参考文献 **11**

一　渲染启动与图像分辨率

PBRT 中，一幅图像被划分为多个小片段 (tiles)，启动多线程渲染，每个线程负责渲染一个小片段。把图像划分为小片段需要权衡线程切换的负担与每个小片段的计算负担，例如四核 CPU 运行四个小片段，那么有可能有的 CPU 核心已经运行完计算，需要等待其他 CPU 核心来运行结束，已经运行完的核心则没有被分配工作。小片段太多，则 CPU 切换线程也会比较费时，带来一些计算负担。

在 PBRT 中设一个小片段是 16×16 像素的。Film::GetSampleBounds() 函数会返回需要被渲染的像素范围（因为每个需要被渲染的像素都要分配样本来采样，所以 GetSampleBounds 意味着找到需要分配样本的像素范围）。下面在计算中加上 (tileSize-1) 是为了保证当要渲染的图像区域的分辨率不是 16 倍数时自动进位。

```
// 计算tiles的数量, _nTiles_, 为了进行并行执行渲染
Bounds2i sampleBounds = camera->film->GetSampleBounds();
Vector2i sampleExtent = sampleBounds.Diagonal();
const int tileSize = 16;
Point2i nTiles((sampleExtent.x + tileSize - 1) / tileSize, (sampleExtent.y +
    tileSize - 1) / tileSize);
```

在渲染的并行程序里计算当前 tile 的边界：

```
    // 计算像素的tile边界
    int x0 = sampleBounds.pMin.x + tile.x * tileSize;
    int x1 = std::min(x0 + tileSize, sampleBounds.pMax.x);
    int y0 = sampleBounds.pMin.y + tile.y * tileSize;
    int y1 = std::min(y0 + tileSize, sampleBounds.pMax.y);
    Bounds2i tileBounds(Point2i(x0, y0), Point2i(x1, y1));
```

然后对 tile 中的每个像素都进行渲染（do-while 结构中相当于循环多轮渲染），该代码见 SamplerIntegrator::Render() 函数：

```
<并行处理每个tile>{
    // 得到FilmTile
    std::unique_ptr<FilmTile> filmTile = camera->film->GetFilmTile(
        tileBounds);
    // 为tile中每个像素渲染
    for (Point2i pixel : tileBounds) {
        do{
            Spectrum L(0.f);
            <计算渲染着色值 L>
            // 给图像添加相机光线的贡献
            filmTile->AddSample(cameraSample.pFilm, L, rayWeight);
        } while(<迭代次数不足>);
    }
    camera->film->MergeFilmTile(std::move(filmTile));
}
// 把渲染结果写入到图像文件中
camera->film->WriteImage();
```

需要注意的是，前面讲过，像素边界不一定等于整个图像的分辨率。例如我们定义了”crop window”来表示整个图像的子集（该子集才是需要被渲染的部分）。

Film 类中有变量“总分辨率”，表示完整的图像范围：

```
const Point2i fullResolution;
```

相机坐标系中的屏幕空间 (Screen space) 就是渲染的胶片的总分辨率（只不过多了一个 z 值表示深度范围），屏幕空间的坐标可以是小数，没必要是整数。将屏幕空间坐标转换到 NDC（标准化坐标，横纵范围都是从 0 到 1）的方法就是屏幕空间坐标的横纵坐标值分别除以屏幕空间分辨率，即除以下面的值：

```
    fullResolution.x
    fullResolution.y
```

但渲染中以及最终保存图像的结果并不一定是全分辨率，而是里面 crop 的部分，因此写入图像时会将全分辨率（fullResolution）与渲染的区域边界（croppedPixelBounds）同时传入到图像保存函数中：

```
pbrt::WriteImage(filename, &rgb[0], croppedPixelBounds, fullResolution);
```

FilmTile 类内有一个小 buffer 用来存储当前 tile 的像素值。渲染工作完成后，tile 将合并到 film 的存储中。

二　图像重建

渲染完的样本值（即辐射度值）需要转化为像素值并储存。依据信号处理原理，我们需要做三件事来计算输出图像每个像素的最终值：

- 从图像样本集中重建连续图像 $\widetilde{L}$。
- 对 $\widetilde{L}$ 预滤波来移除超过 Nyquist 限制的频率部分。
- 在像素位置处采样 $\widetilde{L}$ 来计算得到最终像素值。

我们实际操作中会把前两步合二为一，即使用一个滤波器函数来完成前两步。

2 1　采样与滤波

在采样中，如果采样频率高于 Nyquist 频率，可以用 sinc 滤波器完全重建原始信号。但是图像函数的频率总是比采样频率要高得多（因为场景中有大量边缘等结构），我们进行非均匀采样，通过噪声来抑制混叠现象 (aliasing)。

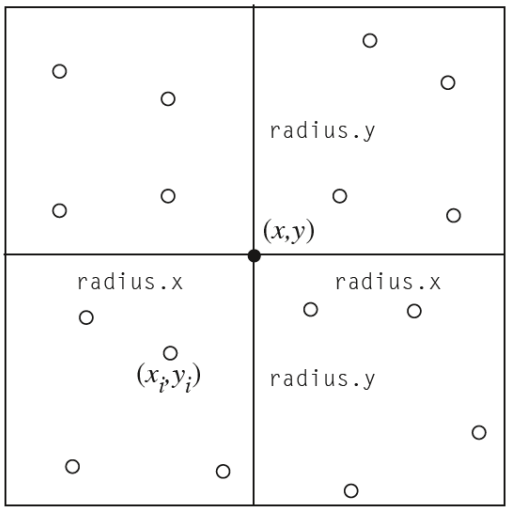
理想重建背后的理论取决于样本的采样间隔是均匀的，虽然已经进行了许多尝试将该理论扩展到非均匀采样，但尚未有一种公认的方法来解决这个问题。且在图像重建中，采样率永远是不够的，所以无法进行完美重建。

最近在抽样理论领域的研究重新审视了重建问题，明确承认在实践中通常无法实现完美的重建。这种视角的轻微转变导致了强大的新重建技术。例如，Unser(2000) 给出了有关这些发展的调研。特别是重建理论的研究目标已经从完美重建转向提高重建技术，无论原始函数是否受到频带限制，都可以证明重建函数和原始函数之间的误差最小化。

对于重建某个像素值，需要插值它周边的样本，设 f 是插值滤波器：

$$I(x,y)=\frac{\sum_i f(x-x_i,y-y_i)w(x_i,y_i)L(x_i,y_i)}{\sum_i f(x-x_i,y-y_i)} \tag{二.1}$$

$w(x_i,y_i)$ 是相机类返回的样本贡献权重（与相机测量有关，对于投影相机，所有相机射线采样的权重都是 1，该值就是 GenerateRayDifferential() 和 GenerateRay() 函数的返回值）。滤波器范围即当前点 (x,y) 扩展半径 radius.x 和 radius.y 构成的矩形中：



在这里，sinc 滤波器不是一个合适的选择：回想一下，当函数频率超过奈奎斯特极限（Gibbs 现象）时，理想的 sinc 滤波器容易出现振铃 (ringing) 现象（可以在冈萨雷斯的《数字图像处理》中找到很好的解释，也可以参考 [3]），这意味着图像中的边缘在附近像素中具有边缘的微弱复制副本。此外，sinc 滤波器具有无限的支撑 (support)：它不会在距离其中心有限的距离处下降到零，因此需要为每个输出像素过滤所有图像样本。实际上，没有单一的最佳过滤功能。为特定场景选择最佳场景需要定量评估和定性判断的结合。

2 2 Filter 类

Filter 类需要知道滤波半径，初始化其成员变量 radius。

Filter 类有一个函数 Evaluate，用来实现滤波方法，它接受一个参数 p，表示相对于滤波器中心的位置，返回该位置处的滤波函数值。

```
1     virtual Float Evaluate(const Point2f &p) const = 0;
```

PBRT 实现了 box 滤波器、三角滤波器等好几种滤波器，因为比较简单，这里不详细描述。

三　相机胶片的工作

Film 模拟了相机传感设备，它决定了样本对每个像素的贡献，以及在渲染结束后，将生成的图像写入到图像文件中（比如 png）文件。

对于现实相机模型，测量方程描述了相机中的传感器如何测量一段时间内到达传感器区域的能量。对于稍微简单一些的相机模型，可以考虑传感器测量的是一段时间内到达一个小区域的平均辐射度（相当于把着色计算的结果取平均）。传感器在代码中如何测量其实是取决于 Camera::GenerateRayDifferential() 函数的返回值的，对于投影相机类，因为返回权重都是 1，所以可以认为就是取平均值。

本节介绍应用像素重建公式计算最终像素值的 Film 实现。对于基于物理的渲染器，通常最好将生成的图像存储为浮点图像格式。与使用具有 8 位无符号整数值的传统图像格式相比，这样做在如何使用输出方面提供了更大的灵活性；浮点格式避免了将图像量化为 8 位所带来的大量信息损失。为了在现代显示设备上显示这样的图像，有必要将这些浮点像素值映射到离散值以进行显示。例如，计算机监视器通常期望每个像素的颜色由 RGB 三基色来描述，而不是任意的光谱功率分布。因此，由一般基函数系数描述的光谱必须在显示之前转换为 RGB 表示。一个相关的问题是，显示器的可显示辐射度值范围比许多真实场景中的范围小得多（比如显示器上显示太阳时，不可能也没必要比真的太阳还要亮，否则会伤害人眼）。

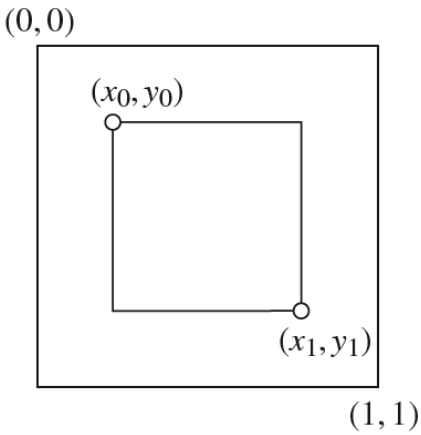
因此，像素值必须使最终显示的图像看起来尽可能接近其在理想显示设备上出现的方式，因此，映射到可显示范围时，通过对色调映射的研究来解决这些问题（即高动态范围图像转低动态范围图像）。

3 1　胶片类

对于 Film 类，有许多值被传递给构造函数：图像的整体分辨率（以像素为单位）；裁剪窗口，其可以指定要渲染的图像的子集；胶片在实际物理区域对角线的长度，原本单位为毫米，此处转换为米（该值只对真实感相机有用，因此目前无需了解）；滤波函数；输出图像的文件名和控制图像像素值如何存储在文件中的参数；maxSampleLuminance 表示采样得到的样本值不能高于该值，该值默认是无穷大。

```
1  Film::Film(const Point2i &resolution, const Bounds2f &cropWindow, std::
       unique_ptr<Filter> filt, Float diagonal, const std::string &filename,
       Float scale, Float maxSampleLuminance) :
2      fullResolution(resolution),
3      diagonal(diagonal * .001),
4      filter(std::move(filt)),
5      filename(filename),
6      scale(scale),
7      maxSampleLuminance(maxSampleLuminance)
8  {
9      <计算Film图像边界>
10     <给Film图像分配存储>
11     <预计算滤波权重表>
12 }
```

初始给定的 cropWindow 是一个范围在 $[0,0]-[1,1]$ 之间的值：



在“计算 film 图像边界”中，Film::croppedPixelBounds 存储裁剪窗口左上角到右下角的像素边界。计算的像素坐标如果出现分数，就向上舍入，这确保了如果一个图像是用多个相邻的裁剪出的子窗口来呈现的，那么每个像素将只会出现在一个子图像中。

构造器对每个像素都分配一个 Pixel 结构：

```
1     pixels = std::unique_ptr<Pixel[]>(new Pixel[croppedPixelBounds.Area()]);
```

Pixel 结构定义为：

```
1 struct Pixel {
2     Pixel() { xyz[0] = xyz[1] = xyz[2] = filterWeightSum = 0; }
3     Float xyz[3];
4     Float filterWeightSum;
5     AtomicFloat splatXYZ[3];
6     Float pad;
7 };
```

光谱的像素贡献的加权和使用 XYZ 颜色表示，并存储在 xyz 成员变量中。filterWeightSum 保存像素采样贡献的过滤器权重值之和。splatXYZ 保存样本 splats 的未加权总和（splat 是 Metropolis 光传输算法的功能，这里不讲解它的意义），为了防止并行冲突，这里定义的是原子操作的变量。pad 变量未使用，其唯一目的是确保 Pixel 结构的大小为 32 字节，而不是 28 字节（假设为浮点数为 4 字节；否则，它确保 64 字节结构）。此填充确保 Pixel 不会跨越缓存线 (cache line)，因此在访问 Pixel 时不会发生多个缓存未命中（只要数组中的第一个 Pixel 在缓存线的开始处分配）。

我们选择使用 XYZ 颜色而不是 RGB 颜色，这是因为 XYZ 是颜色的独立于显示器的表示，而 RGB 需要假设一组特定的显示响应曲线。

对于典型的滤波器设置，每个样本可以贡献给最终图像中的 16 个或更多像素。特别是对于简单场景，在光线相交测试和着色计算上花费的时间相对较少，更新每个样本的图像所花费的时间相比于渲染着色的时间可能比较长。因此，Film 预先计算一个过滤器值表，这样我们就可以避免对 filter::Evaluate() 方法的虚函数调用以及对过滤器求值的开销，而可以使用表中的值进行滤波。没有在每个样本的精确位置计算滤波值而引入的误差在实践中并不明显。

由于滤波器中，$f(x,y) = f(x,|y|) = f(|x|,y) = f(|x|,|y|)$，因此只需要保存滤波器正值表即可，只需要原先 1/4 的存储空间。

Film 需要知道整数像素范围，用来分配采样样本。GetSampleBounds() 函数返回实际要渲染的区域。由于像素重建滤波器通常跨越多个像素，因此采样器必须生成超出实际输出像素范围的图像样本。这样，即使是图像边界处的像素，其周围的所有方向的样本密度也将相等，并且不会使得像素值只偏向图像内部的值。

当使用裁剪窗口渲染图像片段时，该细节也很重要，因为它消除了子图像边缘的伪影。计算样本边界涉及在从离散像素坐标转换为连续像素坐标时考虑半像素偏移（比如最左上角的像素坐标为 (0,0)，其实际像素中心的位置应该是 (0.5,0.5)），然后滤波器半径展开，然后向外舍入：

```
Bounds2i Film::GetSampleBounds() const {
    Bounds2f floatBounds(Floor(Point2f(croppedPixelBounds.pMin) + Vector2f
        (0.5f, 0.5f) - filter->radius), Ceil(Point2f(croppedPixelBounds.pMax)
        - Vector2f(0.5f, 0.5f) + filter->radius));
    return (Bounds2i)floatBounds;
}
```

Film::GetPhysicalExtent() 函数只用于真实感相机，这里我们不需要了解。

3 2 把像素值提供给 Film

本小节可以分为三个函数来描述，即 GetFilmTile() 函数、AddSample() 函数和 MergeFilmTile() 函数。

3 2.1 GetFilmTile() 函数

为 film 提供样本贡献中最直接的方式，是允许渲染器提供胶片像素位置和计算得到的相应光线的光谱值对胶片的直接贡献，但在存在多线程的情况下，提供这种方法的高性能实现并不容易，因为多个线程可能会同时尝试更新图像的同一部分。

因此，Film 定义了一个接口，线程可以在该接口中指定它们正在生成相对于整个图像的某个像素范围内的样本。给定样本边界，GetFilmTile() 会返回一个指向 FilmTile 对象的指针，该对象存储图像在相应区域中的像素的贡献。FilmTile 及其存储的数据的所有权是每个调用线程专有的，因此线程可以向 FilmTile 提供样本值而不必担心与其他线程的争用。当线程完成对 tile 的处理后，线程将完成的 tile 传回胶片，胶片将其安全地合并到最终图像中。

GetFilmTile() 函数会根据滤波半径来扩展当前 tile 的采样像素范围。但是在 crop 图像之外的部分不需要被考虑进来，所以通过 Intersect 求交：

```
    Vector2f halfPixel = Vector2f(0.5f, 0.5f);
    Bounds2f floatBounds = (Bounds2f)sampleBounds;
    Point2i p0 = (Point2i)Ceil(floatBounds.pMin - halfPixel - filter->radius
        );
    Point2i p1 = (Point2i)Floor(floatBounds.pMax - halfPixel + filter->
        radius) + Point2i(1, 1);
    Bounds2i tilePixelBounds = Intersect(Bounds2i(p0, p1),
        croppedPixelBounds);
```

FilmTile 构造器提供了 2D 边界框，边界框给出了最终图像中必须提供存储的像素的边界，以及指向重建滤波器表的指针。FilmTile 中存储像素值借用了 FilmTilePixel> 结构：

```
    pixels = std::vector<FilmTilePixel>(std::max(0, pixelBounds.Area()));
```

该结构有两个变量，用于存储加权贡献和与权重和：

```
struct FilmTilePixel {
    Spectrum contribSum = 0.f;
    Float filterWeightSum = 0.f;
};
```

3 2.2 AddSample() 函数

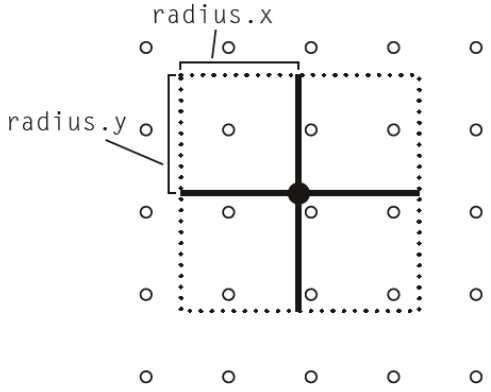
当一个辐射度样本被计算完后，会调用 FilmTile::AddSample() 来将样本贡献到相应像素上。滤波半径为 radius。

由于像素的横纵坐标都是 (0,1,...) 这种整数，而采样值对哪些像素有贡献需要考虑采样的中心位置。例如，对于某个像素 (11,13) 发射的光线，光线在屏幕空间采样范围是 (11-12,13-14)：我们在 Sampler::GetCameraSample() 函数中可以看到发射的光线范围在一个像素内部：

```
1     cs.pFilm = (Point2f)pRaster + Get2D();
```

相机射线采样的过程简单来说是先在镜头采样一个点，再在屏幕空间采样一个点（即 pFilm）。如果采样到的屏幕空间的点是 (11.5,13.5)，我们会认为该点恰好就是像素 (11,13) 的中心点，在该像素范围内，该点的样本权重最大。所以，当计算与像素 (11,13) 的距离时，需要将采样到的屏幕空间的点减去 (0.5,0.5)。

减去 (0.5,0.5) 以后的某个点对周围有贡献的区域可以用下图来描述。其中黑实心圆表示当前采样点的位置，空心圆表示一个个的像素，可以看到该采样点会对周围 9 个像素有贡献：



但是下面的代码中，p1 值还加上了 1：

```
1 Point2f pFilmDiscrete = pFilm - Vector2f(0.5f, 0.5f);
2 Point2i p0 = (Point2i)Ceil(pFilmDiscrete - filterRadius);
3 Point2i p1 = (Point2i)Floor(pFilmDiscrete + filterRadius) + Point2i(1, 1);
4 p0 = Max(p0, pixelBounds.pMin);
5 p1 = Min(p1, pixelBounds.pMax);
```

这是因为循环遍历有贡献区域的过程中不包括坐标最大值：

```
1 for (int y = p0.y; y < p1.y; ++y) {
2     for (int x = p0.x; x < p1.x; ++x) {
3         ......
4     }
5 }
```

滤波器权重 filterWeight 在这里是通过插值滤波值表得到的，而不用再调用滤波器的 Evaluate 函数了。样本权重 sampleWeight 在投影相机中都是 1。

FilmTilePixel::GetPixel() 函数用于给定的 (x,y) 坐标从 vector 中获得对应 Pixel 对象的引用。

3 2.3 MergeFilmTile() 函数

渲染 FilmTiles 得到的结果会被合并到图像中，MergeFilmTile() 函数也是在并行中执行的。执行完以后，FilmTiles 就会被释放掉。

为了防止并行冲突，这里使用了互斥锁：

```
1     std::lock_guard<std::mutex> lock(mutex);
```

防止多个线程同时修改 Film 类中的 Pixel 数组 pixels。

当所有线程的 MergeFilmTile() 都执行完以后，Film 对象的 pixels 数组里就存储了每个像素的对应 Pixel 对象的最终结果。Pixel 结构中包括加权和、滤波权重和等。

3 3 图像输出

并行程序结束后，调用 Film::WriteImage() 程序来写入图像。

在一个大循环中，依次处理每个像素，把每个像素的 XYZ 颜色转换为 RGB，然后通过权重和进行归一化（与 MLT 光传输算法相关的内容这里暂且不提）。

四　整体步骤串联与总结概述

4 1　步骤串联

首先，在屏幕空间采样光线，屏幕空间大小就是整个图像的分辨率大小。但是我们要渲染的只是整个图像的一个子集，该子集由 croppedPixelBounds 定义。

渲染时，将整个要渲染的像素区分成多个 tiles，每个 tile 由 16×16 个像素构成。由于每个 tile 内的样本可能会对临近 tiles 的像素产生贡献（由于滤波器核的宽度导致），因此需要定义 FilmTile 对象，该对象相当于扩大的 tile，里面有存储像素值的内存空间。

渲染中每渲染一个样本，就调用一次 filmTile->AddSample 函数，将样本贡献到 filmTile 的相应像素位置上。

当前 tile 渲染结束以后，就调用 camera->film->MergeFilmTile() 函数，将结果合并，之后 filmTile 内存被自动释放。

等所有线程都结束渲染以后，调用 camera->film->WriteImage() 函数写入图像。

4 2　总结

本文共使用一天时间写完，目的其实是为了将 PBRT 的图像写入和保存机制进行详细地描述。虽然很多时候我们的渲染结果的呈现都是借助我们自己的系统需求，但 PBRT 的实现方式也是值得借鉴的。

本文由于比较简单，且功能我们暂时不太需要，所以并不提供源码，大家有需要的话可以自己移植和使用。

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[D]. Stanford University. 1998.

[3] https://www.cnblogs.com/wxl845235800/p/7692788.html