PBRT 系列 13-无限面光源

Dezeming Family

2021 年 3 月 19 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 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：

本文于 2022 年 7 月 15 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

这本书相对来说比较简单，我们研究的是无限环境光源。在《一些零散和琐碎的内容补充》中，我们实现了一个简单的环境光源，但是没有考虑到重要性采样等内容，因此渲染时收敛会很慢。而本书我们则会实现 PBRT 中的重要性采样环境光。

基本原理很简单，但是实现起来比较困难。无限面光源的贴图是 MIPMap 构造的，MIPMap 实现起来还是比较复杂的，但我们只需要掌握如何应用就可以了。

MIPMap 有多种实现和管理的方法，PBRT 中的功能比较完善，我们简单讲讲其中的原理之后就可以移植到自己的系统里。有了 MIPMap 就可以实现基于 MIPMap 的无限面光源了。

本书的售价是 3 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 MIPMap **1**

二 MIPMap 的移植 **2**
2 1 类定义的初步移植 2
2 2 构造函数的移植 2
2 3 移植和测试 MIPMap 类 3

三 无限面光源 **5**

四 色调映射 **7**
4 1 再议 Film 7
4 2 splatXYZ 8
4 3 gamma 矫正 8
4 4 移植与实现 8

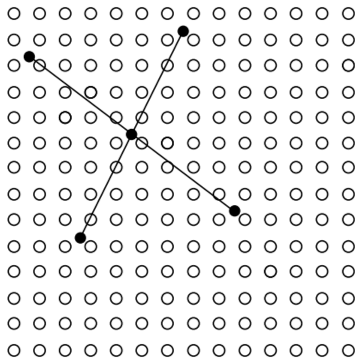
五 总结 **10**

参考文献 **11**

一 MIPMap

MIPMap（much in little Map）定义在 mipmap.h 和 mipmap.cpp 文件中。

如果图像函数具有比纹理采样率更高的频率细节，则会在最终图像中出现混淆。任何高于 Nyquist 极限的频率都必须在计算函数之前通过预滤波去除。考虑到有时候我们需要对纹理采样一大片范围（从分辨率角度考虑，远的物体相当于更大范围的滤波然后映入人眼中的）。



MIPMap 中像素的索引（Lookup 函数）是标准化坐标采样模式，即图像坐标限定在 $0-1$ 之间。当需要索引 MIPMap 中的像素值且当索引超过了 MIPMap 的范围时（例如当大于 1 或者小于 0 时），例如大于了索引分辨率，则根据 ImageWrap 方法选择如何进行索引：

```
// 重复 取0 截断
enum class ImageWrap { Repeat, Black, Clamp };
```

MIPMap 作为模板类，可以使用浮点数或者 Spectrum。

MIPMap 类可以帮助滤波，有两种滤波器，分别是三角滤波器和椭圆滤波器。

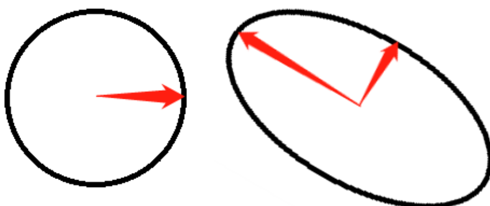
MIPMap 会构建图像金字塔，即不同分辨率的原图，用以进行滤波。原始图像纹理位于金字塔的底部，每个级别的图像都是上一级分辨率的一半，最高级别的分辨率为原始图像分辨率。金字塔背后的基本思想是，如果需要过滤大面积的 texel，一个合理的近似值是使用更高层次的金字塔，并在同一区域进行过滤，访问的 texel 更少。

构造函数要保证图像读入的分辨率是 2 的 n 次方，否则就用 RoundUpPow2 函数把宽或高 resize 到 2 的 n 次方。然后再初始化各个层级，该过程需要用到 resampleWeights 方法来得到采样各个 texels 的权重。

然后就开始初始化各个 MIPMap 层级。

之后初始化 EWA 权重，如果使用 EWA 进行滤波（椭圆滤波器）的话就能用到。

查找纹理值的函数有两个 MIPMap::Lookup 函数。第一个函数输入纹理坐标和查找范围宽度，如下图左。第二个函数输入坐标纹理以及两个向量，这两个向量用来描述滤波范围（如果只有宽度，那么只能滤波固定规则范围，且没有偏向），如下图右。



具体的实现细节我们不做表述，一是书中有详细的解释，二是因为这只是一个工具，我们只要会使用它就行了。下一节我会移植到自己建立的系统中，然后进行测试。

二　MIPMap 的移植

本章我们开始移植 MIPMap 类。

2 1　类定义的初步移植

类定义的移植非常简单，没有什么可说的。注意 clamp 函数一共有 3 个重载，对应了 float 型数据和两种光谱类数据。Lanczos 函数是用来做滤波的，即使用 Sinc 函数进行纹理滤波。注意下面的代码：

```
std::vector<std::unique_ptr<BlockedArray<T>>> pyramid;
```

pyramid 用来索引图像金字塔，该 vector 数组存储多级纹理。我们可以把 BlockedArray 类移植到自己的系统里，其中该类的有些地方我们需要修改一下：

```
//注意把PBRT内存管理相关的内容：
data = AllocAligned<T>(nAlloc);
FreeAligned(data);
//改为下面的内容
data = new T[nAlloc];
delete [] data;
```

别忘了在全局头文件声明：

```
template <typename T, int logBlockSize = 2>
class BlockedArray;
```

编译通过即可。

2 2　构造函数的移植

构造函数的移植：先把函数体清空。我们先在主程序中测试一下是否可以编译：

```
    float * mpdata = nullptr;
    MIPMap<float> mp(Point2i(100,100), mpdata, true);
```

可以编译则说明类定义移植正确。

构造函数中首先去掉调试日志，然后我们需要重点处理的是与 PBRT 内存管理和多线程相关的内容。我们先将下面的函数都实现了：

```
IsPowerOf2
RoundUpPow2
Log2Int
```

然后移植函数体以后，注释掉与 PBRT 多线程相关的内容，编译成功说明没有问题了。

ParallelFor 是启动多线程的函数，接受三个参数：func 表示要运行的函数，count 表示线程数，chunkSize 表示需要启动的线程数。如果不启动多线程，则循环执行 func 函数。因此我们可以将多线程改为循环执行：

```
// Apply _sWeights_ to zoom in $s$ direction
//注意ParallelFor的输入参数是t
for (int64_t t = 0; t < resolution[1]; ++t) {
    for (int s = 0; s < resPow2[0]; ++s) {
        // Compute texel $(s,t)$ in $s$-zoomed image
        resampledImage[t * resPow2[0] + s] = 0.f;
```

```
        for (int j = 0; j < 4; ++j) {
            int origS = sWeights[s].firstTexel + j;
            if (wrapMode == ImageWrap::Repeat)
                origS = Mod(origS, resolution[0]);
            else if (wrapMode == ImageWrap::Clamp)
                origS = Clamp(origS, 0, resolution[0] - 1);
            if (origS >= 0 && origS < (int)resolution[0])
                resampledImage[t * resPow2[0] + s] +=
                sWeights[s].weight[j] *
                img[t * resolution[0] + origS];
        }
    }
}
```

其他的修改也按照这样就可以了。有一段修改比较难：

```
T *workData = new T[resPow2[1]];
for (int64_t s = 0; s < resPow2[0]; ++s) {
    for (int t = 0; t < resPow2[1]; ++t) {
        workData[t] = 0.f;
        for (int j = 0; j < 4; ++j) {
            int offset = tWeights[t].firstTexel + j;
            if (wrapMode == ImageWrap::Repeat)
                offset = Mod(offset, resolution[1]);
            else if (wrapMode == ImageWrap::Clamp)
                offset = Clamp(offset, 0, (int)resolution[1] - 1);
            if (offset >= 0 && offset < (int)resolution[1])
                workData[t] += tWeights[t].weight[j] *
                resampledImage[offset * resPow2[0] + s];
        }
    }
    for (int t = 0; t < resPow2[1]; ++t)
        resampledImage[t * resPow2[0] + s] = clamp(workData[t]);
}
delete[] workData;
```

最后我们再移植 Texel 函数。移植好以后，编译和运行都没有问题了，就说明构造函数移植成功。

2 3 移植和测试 MIPMap 类

其他几个函数的移植都非常简单，因此不详细说明是如何修改的。

测试我选择在 SamplerIntegrator 类的 Render 函数中进行。

我们先构建一张图像：

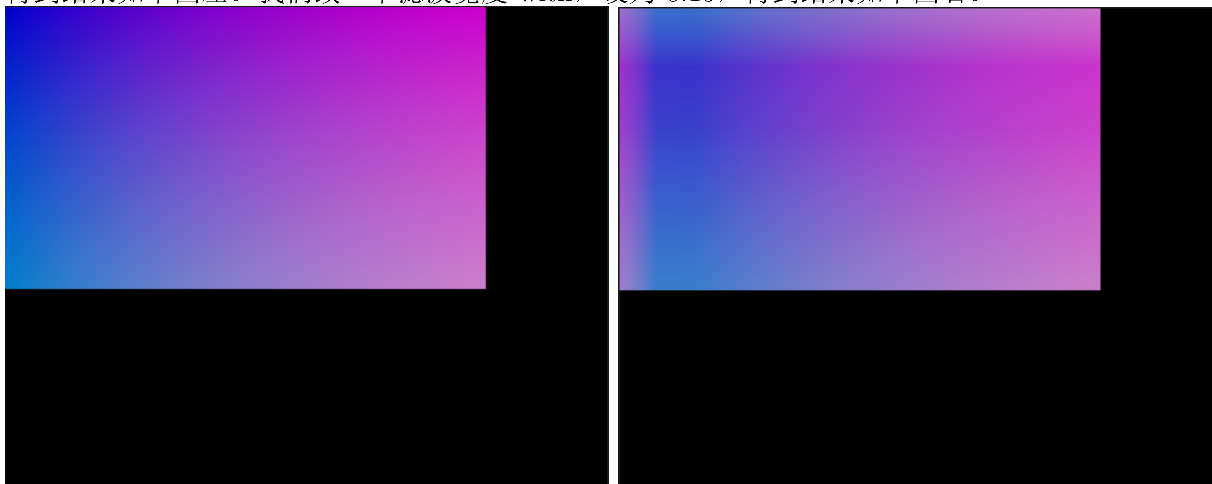
```
int mW = 400, mH = 235;
Spectrum * mpdata = new Spectrum[mW * mH];
for (int j = 0; j < mH; j++) {
    for (int i = 0; i < mW; i++) {
        int offset = i + j * mW;
        Spectrum s;
```

```
            s[0] = i / (float)800;
            s[1] = j / (float)800;
            s[2] = 0.4f;
            mpdata[offset] = s;
        }
}
MIPMap<Spectrum> mp(Point2i(mW, mH), mpdata, true);
```

然后在渲染的程序里：

```
    //i,j表示当前像素的索引，渲染图像分辨率为500*400
if (i < mW - 1 && j < mH - 1)
    colObj = mp.Lookup(Point2f((i + 1) / (float)RasterWidth, (j + 1) / (
        float)RasterHeight), 0.0f);
else
    colObj = Spectrum(0.f);
```

得到结果如下图左。我们改一下滤波宽度 with，设为 0.15，得到结果如下图右。

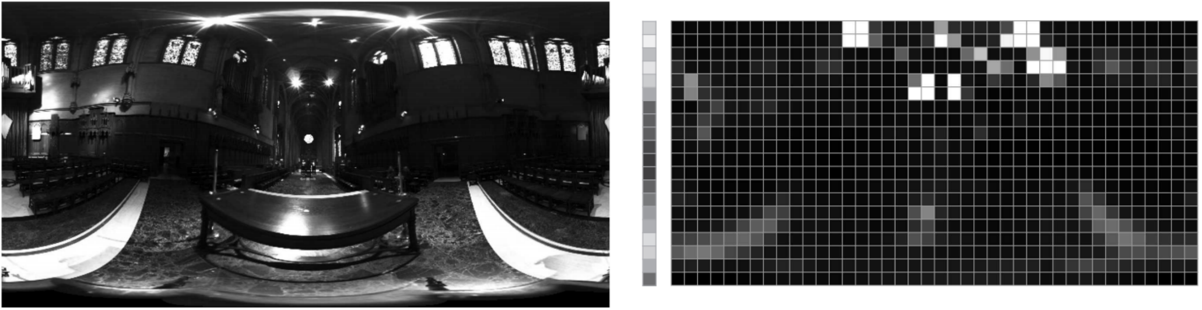


还是能看出 with 变大以后图更模糊的。

三 无限面光源

无限面光源的使用还是比较容易的。

InfiniteAreaLight 类会将整个区域进行统计，得到亮度二维分布概率密度：



在有了 MIPMap 以后，我们只需要注意一下天空盒的坐标即可。移植过程非常简单，这里就不再赘述了。读取文件我采用的是 stb_image.h（该文件见《一些零散和琐碎的内容》）。

```
float *data;
RGBSpectrum *data_s;
if (texmap != "") {
    //不再使用PBRT的文件读取程序
    //texels = ReadImage(texmap, &resolution);
    int imageWidth, imageHeight, nrComponents;
    data = stbi_loadf(texmap.c_str(), &imageWidth, &imageHeight, &
        nrComponents, 0);
    data_s = new RGBSpectrum[imageWidth * imageHeight];
    for (int j = 0; j < imageHeight; j++) {
        for (int i = 0; i < imageWidth; i++) {
            RGBSpectrum r;
            r[0] = L[0] * data[(i + j * imageWidth) * nrComponents + 0];
            r[1] = L[1] * data[(i + j * imageWidth) * nrComponents + 1];
            r[2] = L[2] * data[(i + j * imageWidth) * nrComponents + 2];
            data_s[i + j * imageWidth] = r;
        }
    }
    resolution.x = imageWidth;
    resolution.y = imageHeight;
}
free(data);
std::unique_ptr<RGBSpectrum[]> texels(data_s);
```

要注意的是，因为在这里 SphericalTheta 和 SphericalPhi 函数认为 theta 角是与 z 轴的夹角，phi 角是 x-y 平面上的角，因此我们要想显示正确，需要将值进行反转（见下面代码）。我不打算将 SphericalTheta 和 SphericalPhi 改成以前我们实现的方式了，怕后面再遇到的时候移植 PBRT 出错很难定位原因。

注意 Sample_Le 与 Pdf_Le 是为了实现双向光传输的，这里因为我们用不到，就不用真正实现。

我们实现一下：

```
//无限环境光
Transform InfinityLightToWorld;
//因为theta角度问题，需要沿着x轴转90度。
InfinityLightToWorld = RotateX(90) * InfinityLightToWorld;
```

```
5 Spectrum power(1.0f);
6 std::shared_ptr<Light> infinityLight =
7     std::make_shared<InfiniteAreaLight>(InfinityLightToWorld, power, 10,
        hdrFile);
8 lights.push_back(infinityLight);
```

渲染得到结果如下图左。但是阴影不太明显。因此我们将数据取 1.5 次方：

```
1     data_s[i + j * imageWidth] = r + Sqrt(r);
```

得到结果如下图有：

四　色调映射

我们之前渲染的图像都是直接使用 RGB，而没有做其他转换，本节我们研究的是颜色空间的转换和计算。

4 1　再议 Film

本节先阐述一下过程，里面的细节会在下一节再讲解。

SamplerIntegrator 积分器的 Render 函数在渲染完每一个像素的一次采样以后，将结果进行保存：

```
1 filmTile->AddSample(cameraSample.pFilm, L, rayWeight);
```

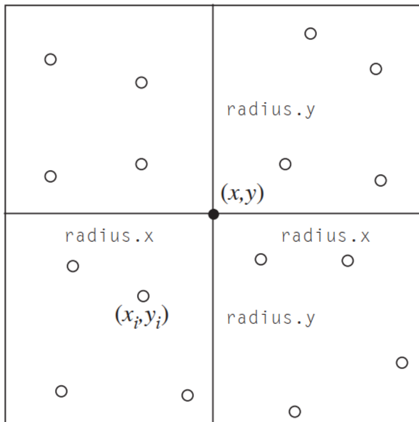
等渲染完一个 tile 里的所有像素以后，执行：

```
1 camera->film->MergeFilmTile(std::move(filmTile));
```

等像素全都 Merge 到一起以后，就保存图像。

```
1 camera->film->WriteImage();
```

AddSample 函数会根据当前点在像素内的偏移来进行滤波，计算滤波权重：



执行时，Film 类的 pixel.contribSum 保存了渲染值的加权总和，pixel.filterWeightSum 保存了权重的总和。

等执行到 MergeFilmTile 时，加权和会被转化为 xyz 表示，并保存到 mergePixel.xyz[i] 里：

```
1 tilePixel.contribSum.ToXYZ(xyz);
2 for (int i = 0; i < 3; ++i) mergePixel.xyz[i] += xyz[i];
```

等到 WriteImage 函数时，会再次进行转化：

```
1 XYZToRGB(pixel.xyz, &rgb[3 * offset]);
2 Float filterWeightSum = pixel.filterWeightSum;
3 //根据加权系数求权重和
4 ......
5 XYZToRGB(splatXYZ, splatRGB);
```

Film::WriteImage 最后再调用 pbrt::WriteImage 函数。该函数定义了一个宏：

```
1 TO_BYTE(v) (uint8_t) Clamp(255.f * GammaCorrect(v) + 0.5f, 0.f, 255.f)
```

该宏将 rgb 值先经过 gamma 矫正，然后 clamp 到 0 到 255 范围内。

4 2 splatXYZ

首先注意 WriteImage 函数里，rgb 分别加了两次值，i 分别为 0,1,2：

```
//第一次
rgb[3 * offset + i] = std::max((Float)0, rgb[3 * offset + i] * invWt);
//第二次
rgb[3 * offset + i] += splatScale * splatRGB[i];
```

第一次加的是赋给的加权平均值，第二次加的是 splatXYZ[3] 转换为的 splatRGB[i]。该值在双向路径追踪中会遇到，在路径追踪中值为 0，所以我们暂且不提。

4 3 gamma 矫正

在《颜色空间理论》中我们详细介绍了颜色理论和 Gamma 矫正的原理，这里我们仅仅简单讲解一下关键点。

像素值可以通过 gamma 校正以将它们映射到所需的范围。Gamma 校正尤其重要而且要小心处理：计算机显示器通常不会显示要显示的像素值和它们的 radiance 之间的线性关系。因此，比如艺术家可以创建一个图像，在 LCD 显示器上看到的图像的一部分显示亮度为另一部分的两倍，但是相应的像素值实际上没有 2:1 的关系。（实际上恰恰相反，其值具有 2:1 关系的像素不会导致 2:1 的亮度比。）对于使用此类图像作为纹理贴图的渲染器来说，这种差异是一个问题，因为渲染器通常期望 texel 值与其表示的数量之间存在线性关系。

pbrt 遵循 sRGB 标准，该标准规定了与 CRT 显示器一般行为相匹配的特定传输曲线。此标准在 2015-era 设备上得到广泛支持；其他（非 CRT）设备（如 LCD 或喷墨打印机）通常接受 sRGB 伽马校正值作为输入，然后在内部重新映射以匹配设备特定的行为。

sRGB 伽马曲线是一个分段函数，低值为线性关系，中大值为幂关系：

$$\gamma(x) = \begin{cases} 12.92x, & x \leq 0.0031308 \\ (1.055)x^{1/2.4} - 0.055, & x > 0.0031308 \end{cases} \tag{四.1}$$

4 4 移植与实现

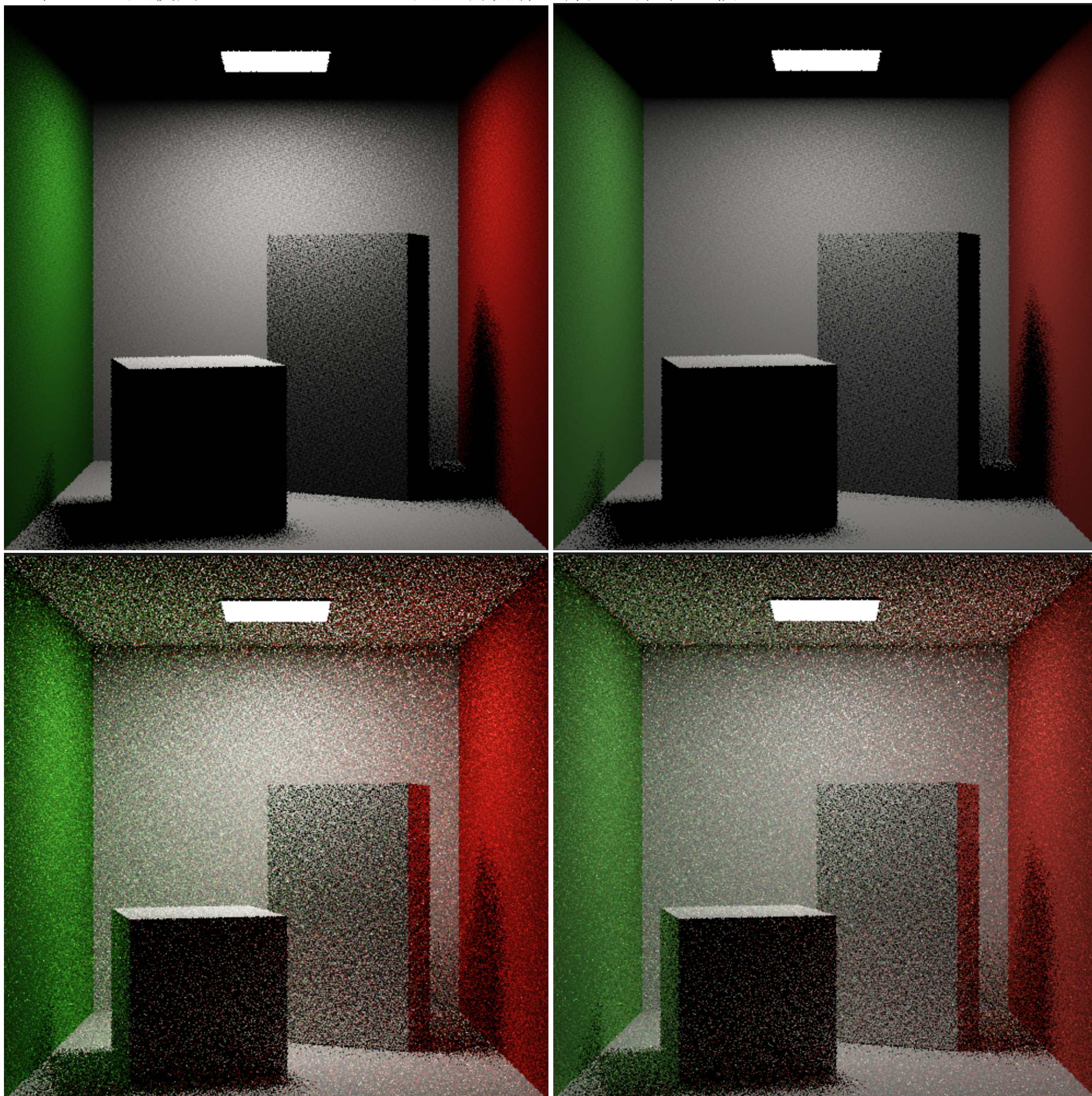
我们在《一些零散和琐碎的内容补充》中曾经使用过 HDRtoLDR 的方法，将大于 1 的 radiance 转换为 0-1 之间的值。

我们和 PBRT 的区别是，PBRT 的 radiance 先转化为 xyz，再取加权平均，再转为 RGB；而我们的步骤是先取加权平均（因为我们没有使用滤波器，因此加权平均其实就是取平均，下面程序中的 colorA 就是取平均以后的值），然后再转为 XYZ，最后再转为 RGB 值。因为是线性变换，所以顺序并没有什么影响。

```
//buffer_f: 存储了float类型的值
Spectrum colorA = Spectrum::FromRGB(buffer_f + offset);
float xyz[3];
colorA.ToXYZ(xyz);
float rgb[3];
XYZToRGB(xyz, rgb);
#define TO_BYTE(v) (uint8_t) Clamp(255.f * GammaCorrect(v) + 0.5f, 0.f, 255.f)
buffer[offset + 0] = TO_BYTE(rgb[0]);
buffer[offset + 1] = TO_BYTE(rgb[1]);
buffer[offset + 2] = TO_BYTE(rgb[2]);
```

```
11 #undef TO_BYTE
```

我们对比一下使用 PBRT 色调映射和使用之前的 HDRtoLDR 的区别，下面的图分别是直接光照和路径追踪，左边是使用 HDRtoLDR，右边是色调映射，都只渲染了 1 帧。



可以看到，使用 PBRT 的色调映射方法之后，显示的结果更柔和了。

五　总结

在写本书时，我一直考虑写多少。以前总觉得写的内容不够详略得当，很多 [1] 中出现的内容我又写了一遍，但是还不如 [1] 详细。

其实在完成《形状与加速器》一节以后，大家完全就可以自己去学习去移植了，也不一定非要按照我的顺序来。不过我也是为了自己再从头到尾摸索一遍，因此打算完整地叙述 PBRT 系统的移植和实现过程。并且，PBRT 的好处在于它实现了非常出色的高级积分器，比如光子映射器和这让我很舒服

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.