

# 真实感图形渲染报告

李晨昊 2017011466

2019-6-26

## 目录

<b>1 项目概述</b>	<b>2</b>
<b>2 功能概述</b>	<b>3</b>
2.1 PathTracing . . . . .	3
2.2 PPM . . . . .	3
2.3 旋转 Bézier 曲面 . . . . .	4
2.4 拓展功能 . . . . .	6
2.4.1 各种几何体 . . . . .	6
2.4.2 景深 . . . . .	8
2.4.3 贴图 . . . . .	10
2.4.4 色散 . . . . .	10
2.4.5 雾 . . . . .	13
2.4.6 水波效果 . . . . .	14
<b>3 加速</b>	<b>16</b>
3.1 硬件加速 . . . . .	16
3.1.1 GPU 加速 . . . . .	16
3.1.2 多线程加速 . . . . .	17
3.2 算法加速 . . . . .	17
3.2.1 KDTree . . . . .	17
3.2.2 OctTree . . . . .	20
3.2.3 Hash 网格 . . . . .	21
3.2.4 低差异序列 . . . . .	21
3.2.5 三角形求交算法 . . . . .	23

# 1 项目概述

本项目最初用 rust 实现，已经达到了比较理想的效果和速度，充分利用了 rust 的 enum 机制来代替很多其它语言中的基类 + 虚函数机制（我大概能想象到那是怎么写的：一个抽象的 `Geometry` 基类，一堆 `Sphere` 之类的来继承它）。Tagged-union 机制的开销相比于虚函数调用已经减少了很多，但是我认为运行过程中仍然做了很多无谓的判断：显然一个几何体的类型能够在渲染前就确定下来，为什么每次运算的时候还需要判断它的类型呢？

于是，我另外用 rust 写了一套代码生成的代码，将几何场景转化成 C++/CUDA 代码，完全消除这些无谓的判断。在我的平台<sup>1</sup>上，C++ 版本对于任何场景相比 rust 版本都能有大约 40%-60% 的性能提升<sup>2</sup>，而 CUDA 版本视场景而定，相比于 C++ 版本又有了大约 300%-3000% 的性能提升。具体细节会在后面详述。

这种代码生成的思路给我的代码实现带来了很大的自由性，我可以提前用 rust 以一种更加紧凑高效的形式，准备好 C++/CUDA 程序所需要的数据结构。同时各种功能可以以代码片段的形式直接添加到生成好的 C++ 和 CUDA 程序中，灵活性很强。

使用了 rust 库 `serde` 来实现场景的读取和保存，可以很容易的将场景和 json 或者 `bincode` 格式互相转换。在 rust 端使用 rust 库 `png` 来输出 png 格式的图片。在 C++/CUDA 端使用了非常简短易用的 `svpng` 来输出 png 格式的图片，由于它的实现比较简单，可能同等尺寸的 png 图片会比别人的大出不少。

代码组织如下：

- `tools/ppm_util.hpp`: C++ 代码，提供 PPM 算法所需的基础设施
- `tools/tracer_util.hpp`: C++ 代码，兼容 CUDA，提供 PathTracing 算法和 PPM 算法都需要的基础设施
- `physics/`: 简单的物理场景计算，目前仅用于制作几个简单的视频，后面会提到
- `bezier.rs`: 实现 Bézier 曲线/旋转 Bézier 曲线/Bézier 曲面的相关算法
- `kd_tree.rs`: 实现用于组织三角形网格的 KDTree
- `mat44.rs`: 实现用于对三角形网格进行坐标变换的 4x4 矩阵
- `oct_tree.rs`: 实现用于组织三角形网格的 OctTree(Deprecated)
- `tri_aabb.rs`: 实现三角形和 AABB 的求交算法，用于 OctTree 的生成 (Deprecated)
- `world.rs`: 渲染的总控代码
- `codegen.rs`: 实现 C++/CUDA 代码生成，**以生成出来的代码实现了所有渲染的功能**
- `load.rs`: 加载文件，包括 obj 文件和贴图文件等
- `material.rs`: 实现表面的材质和颜色
- `util.rs`: 提供随机数生成器，定义误差限 EPS 等基础设施

---

<sup>1</sup>CPU: AMD R7-2700，超频至 3.9GHz; GPU: NVIDIA GTX 1060; C++ 编译器: g++ 8.2.1, 编译选项:-Ofast -fopenmp -march=native; CUDA 编译器:nvcc release 10.1, V10.1.105, 编译选项:-O3 -use\_fast\_math

<sup>2</sup>性能定义为耗时的倒数

- geo.rs: 封装所有几何体，后面会提到
- main.rs: 入口，可以选择直接用 rust 渲染或者生成 C++/CUDA 代码
- mesh.rs: 实现三角形网格
- pic.rs: 封装 ppm 和 png 文件的读写
- vec.rs: 实现二维/三维/四维向量，主要的功能实现是依赖于三维向量，其余两个只在较少场合用到

顺便说一句，在我看来传统的面向对象那一套真是相当恶心，尤其不适用于这种对于性能有着较高要求的应用。

## 2 功能概述

### 2.1 PathTracing

在 CPU 和 GPU 上实现了 PathTracing 算法。

它的思想非常简单：为了统计一个像素点的颜色，向这个像素点发射多条光线，光线在场景中按照光学规律发生反射和折射，而遇到理想粗糙表面时，随机选择一个方向继续前进；光线遇到光源时，即对本像素点的颜色产生贡献。

由于遇到理想粗糙表面时的随机选择，导致 PathTracing 算法可能需要较多的采样数才能收敛。根据我的实际测试，一般选择每个像素点采样 8000 次以上效果就已经相当不错了；如果采样 15000 次以上，图像的质量就相当理想，再增加采样次数也不会有很明显的提升了。不过后面提到的一些渲染特效还会引入其他的随机选择，因此这个数字只对于最基本的场景成立。

这个算法本身非常容易并行化，这也就意味着在 GPU 上实现时，难度的提升会比较小。最后的 GPU 的实现版本相比于 CPU 还是做了一定的修改，详见[GPU 加速](#)。

### 2.2 PPM

只在 CPU 上实现了 PPM 算法。

PPM 是一个两趟的过程：一趟 RayTracing Pass 和后续的多趟的 PhotonTracing Pass。前者的视角与 PathTracing 的视角相同，都是从像素点出发射光线，光线在场景中按照光学规律发生反射和折射，而遇到理想粗糙表面时，光线立即停止，并且将交点及其它一些辅助信息记录下来，后续再以某种数据结构对其进行组织，方便查询。后者的视角是从光源处发射光子，同样也进行正常的反射和折射，遇到粗糙表面时，根据它附近的交点信息来计算光强贡献。此时光子可以停止，也可以随机选择一个方向继续前进，后者的效果会好一些，但是如果直接套用 PathTracing 的随机数实现，会产生大量的噪点，这可以通过在低差异序列上采样来解决，详见[低差异序列](#)。

没有采用 KDTTree 来组织交点信息，而是换用了另一种更加高效，用途也更受限的空间数据结构：[Hash 网格](#)。

### 2.3 旋转 Bézier 曲面

我采用的是先在[KDTTree](#)上做光线求交，以此作为初值采用 Newton 迭代法求解非线性方程组的方案。

求解过程如下：

根据数值分析的知识，非线性方程组  $\vec{F}(\vec{x}) = \vec{0}$  对应的 Newton 迭代法的迭代公式为

$$\vec{x}^{(k+1)} = \vec{x}^{(k)} - J_{\vec{F}}(\vec{x}^{(k)})^{-1} \vec{F}(\vec{x}^{(k)})$$

具体在本应用中， $\vec{F} : R^3 \rightarrow R^3$ ,  $\vec{x} \in R^3$ , 记需要渲染的 Bézier 曲线为  $\vec{B}(t) : R^1 \rightarrow R^2$ , 则  $\vec{F}$  为

$$\vec{F}(u, v, t) = G_u [\vec{B}_x(v), \vec{B}_y(v), 0]^T - (\vec{o} + t\vec{d})$$

其中  $G_u$  是由旋转角  $u$  定义的 Givens 旋转矩阵：

$$G_u = \begin{bmatrix} \cos(u) & 0 & \sin(u) \\ 0 & 1 & 0 \\ -\sin(u) & 0 & \cos(u) \end{bmatrix}$$

从而计算出  $\vec{F}$  的表达式为：

$$\vec{F}(u, v, t) = \begin{bmatrix} \vec{B}_x(v) \cos(u) - \vec{o}_x - t\vec{d}_x \\ \vec{B}_y(v) - \vec{o}_y - t\vec{d}_y \\ -\vec{B}_x(v) \sin(u) - \vec{o}_z - t\vec{d}_z \end{bmatrix}$$

从而更新项为：

$$\vec{\Delta t} = \begin{bmatrix} \vec{B}'_x(v) \cos(u) & -\vec{B}_x(u) \sin(v) & -\vec{d}_x \\ \vec{B}'_y(v) & 0 & -\vec{d}_y \\ -\vec{B}'_y(v) \sin(u) & -\vec{B}_x(u) \cos(v) & -\vec{d}_z \end{bmatrix}^{-1} \vec{F}(u, v, t)$$

我本来是希望从这个表达式中直接推导出更新项的封闭形式的，但是推导后发现并不能得到比较简洁的形式。当然，通过应用 Cramer 法则的确可以得到封闭形式，课本上的更新公式

就可以通过我的推导加上 Cramer 法则得到。但是 Cramer 法则的运算次数<sup>3</sup>并不优化，课本 P107 上的解法需要 50 次运算。这里我选择采用 Gauss 消去法来求解线性方程组。对于 3x3 的线性方程组，Gauss 消去法需要 28 次浮点运算<sup>4</sup>。利用矩阵中有一个 0，可以将运算次数减少到 21 次（某一次行更新可以省去）。虽然看起来这个数字差别并不大，但是考虑到我使用的 Bézier 曲线的求值方法的计算量也不大，因此其速度相差的比例还是比较明显的。

关于 Bézier 曲线的求值，我没有使用更加简单，数值稳定性更好，但是需要  $O(n^2)$  时间的 de Casteljau's 算法，而是采用了更加直接的多项式求值方法。在 rust 端预先求出插值多项式的每一项，运行时通过应用秦九韶算法，只需要  $O(n)$  的时间即可完成对 Bézier 曲线上某点的求值。观察插值多项式的形式：

$$\vec{B}(t) = \sum_{i=0}^n \vec{P}_i C_n^i t^i (1-t)^{n-i}$$

其中很多系数的绝对值都较大，为了得到最终的系数可能发生严重的抵消现象。为了尽量减小抵消现象的影响，我使用了 128 位浮点数来存储中间结果，当然运算过程中我还是只会使用 32 位浮点数的，在 GPU 上 32 位浮点数的效率远高于 64 位浮点数，而在 CPU 上，由于 SIMD 指令的存在，32 位浮点数也往往较快一些。

实际代码中，只需要迭代更新 5 次左右即可收敛。Newton 迭代法的核心代码如下：

```
for (u32 i = 0; i < ITER; ++i) {
    EVAL_BEZIER(PS, v, bx, by);
    EVAL_BEZIER(DER, v, dbx, dby);
    // 填写增广矩阵，左侧为 Jacobi 矩阵，右侧为函数值
    a00 = -bx * sinf(u), a01 = dbx * cosf(u), a02 = -d.x;
    b0 = bx * cosf(u) - o.x - t * d.x;
    /* a10 = 0          , */ a11 = dby           , a12 = -d.y;
    b1 = by - o.y - t * d.y;
    a20 = -bx * cosf(u), a21 = -dbx * sinf(u), a22 = -d.z;
    b2 = -bx * sinf(u) - o.z - t * d.z;
    err = b0 * b0 + b1 * b1 + b2 * b2;
    // 填写完矩阵后，以下经过恰好 21 次运算，求出三个更新分量
    {
        f32 fac = a20 / a00;
        a21 -= fac * a01, a22 -= fac * a02, b2 -= fac * b0;
    }
}
```

<sup>3</sup>现代处理器上，各个浮点运算的耗时差距不太大，简单起见就用运算次数做性能的参考指标

<sup>4</sup>Comparison of Solution of 3x3 System of Linear Equation in Terms of Cost

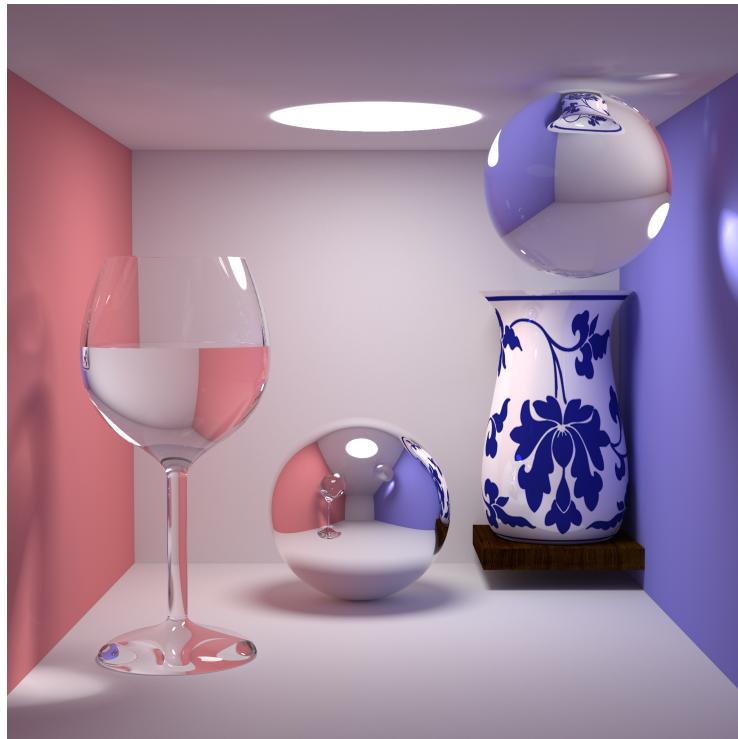
```

f32 fac = a21 / a11;
a22 -= fac * a12, b2 -= fac * b1;
}

f32 x2 = b2 / a22;
f32 x1 = (b1 - x2 * a12) / a11;
f32 x0 = (b0 - x2 * a02 - x1 * a01) / a00;
// 更新
u -= x0, v -= x1, t -= x2;
}

```

将若干个 Bézier 曲面和圆盘组合在一起可以得到更加复杂的图形，例如一个高脚杯：



## 2.4 拓展功能

### 2.4.1 各种几何体

已经实现的几何体包括：

```

pub enum Geo {
    Sphere(Sphere),
    InfPlane(InfPlane),
}

```

```

Circle(Circle),
Rectangle(Rectangle),
Mesh(Mesh),
RotateBezier(RotateBezier),
}

```

其中 `Circle` 和 `Rectangle` 都是平面图形，但是也可以用它们组合成立体图形。

`Mesh` 封装了三角形 mesh。三角形 mesh 是从 obj 文件中读取得到的，读取完成后可以使用变换矩阵对其在空间中进行平移和缩放，变换完成后使用一棵 KDTTree 将其组织起来，然后将此 KDTTree 以某种方式写入文件，让 C++/CUDA 程序可以调用，详见[KDTTree](#)。

求交方法列举如下：

### 1. Sphere

球的方程为  $\|\vec{x} - \vec{c}\|_2 = r$ ，带入射线方程  $\vec{x} = \vec{o} + t\vec{d}$ ，化简得到：

$$\|\vec{d}\|_2 t^2 + 2((\vec{o} - \vec{c}) \cdot \vec{d})t + \|\vec{o} - \vec{c}\|_2^2 - r^2 = 0$$

这是一个关于  $t$  的一元二次方程，求解其最小正根即可。

### 2. InfPlane

除非平行，否则直线与无穷大平面总有交点，只要这交点对于射线来说是  $t > 0$  的点即可。设平面法向量为  $\vec{n}$ ，平面上任一点为  $\vec{p}$ ，则方程为：

$$(\vec{o} + t\vec{d} - \vec{p}) \cdot \vec{n} = 0$$

假定不是平行情形，即  $\vec{n} \cdot \vec{d} \neq 0$ ，解得

$$t = \frac{\vec{n} \cdot (\vec{p} - \vec{o})}{\vec{n} \cdot \vec{d}}$$

实际代码中无需判断平行情况，只需要判断解出来的  $t$  是否满足  $0 < t < t_{prev}$ ，而根据 IEEE 浮点数标准，平行情况中解得的  $t$  只可能是 inf, -inf, nan 中的一个，它们都不满足这个条件。这样可以减少一个分支判断。

### 3. Circle

先按照无穷大平面处理，解得交点后判断它到圆心的距离是否小于  $r$  即可。

### 4. Rectangle

先按照无穷大平面处理，解得交点后判断它到圆心的距离是否在矩形之内即可。判断方法为：设输入的矩形两边为  $\vec{u}$  和  $\vec{v}$ ，这两边的交点为  $\vec{p}$ ，则预先计算

$$\vec{u}' = \frac{\vec{u}}{\|\vec{u}\|_2^2}; \quad \vec{v}' = \frac{\vec{v}}{\|\vec{v}\|_2^2}$$

运行时，设交点为  $\vec{h}$ ，计算：

$$u = (\vec{h} - \vec{p}) \cdot \vec{u}'; \quad v = (\vec{h} - \vec{p}) \cdot \vec{v}'$$

这样运算次数和需要存储的变量较少，而且计算出来的  $uv$  直接就是平面参数方程的参数，可以用来在贴图上访问像素。

#### 5. Mesh

见[KDTree](#)。

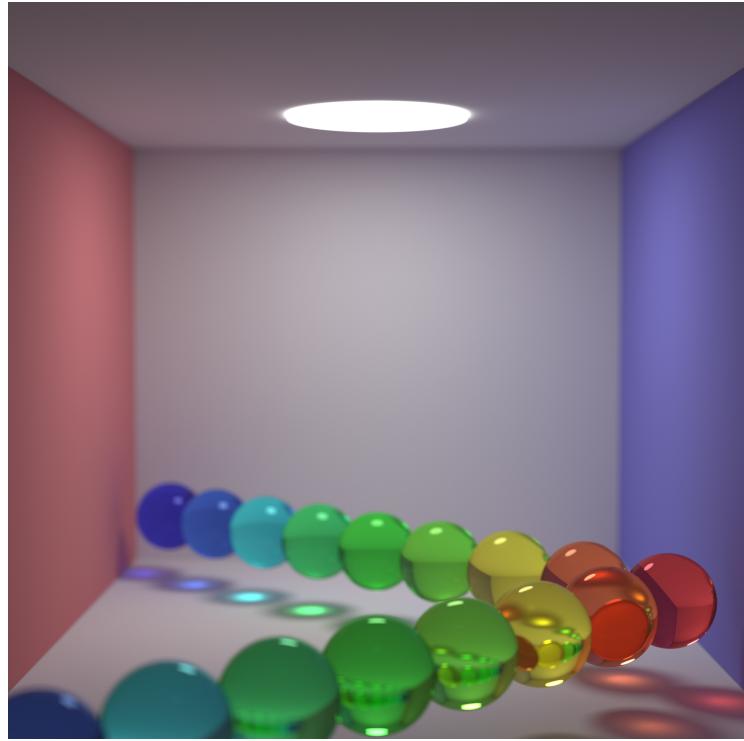
#### 6. RotateBezier

前面已经论述过。

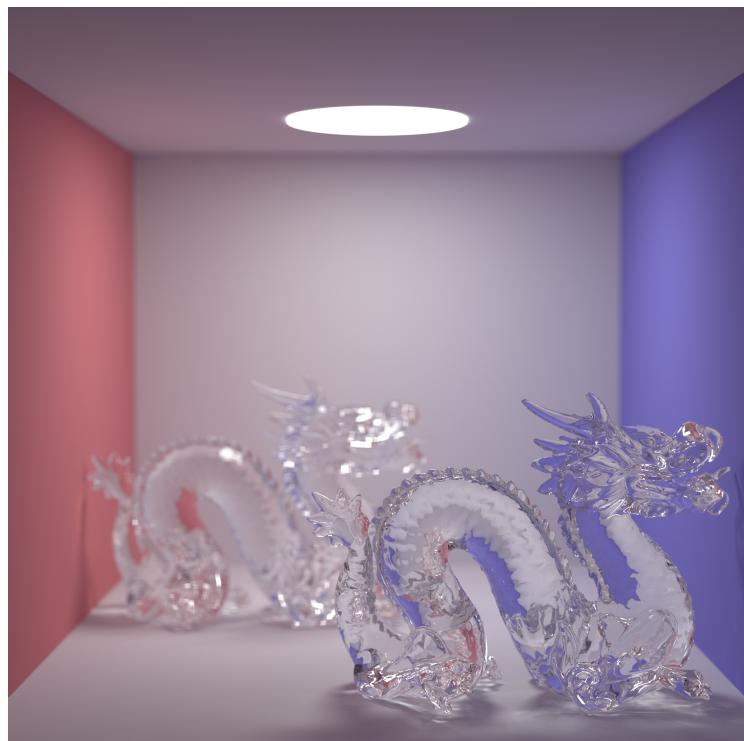
### 2.4.2 景深

大致实现思路是，将摄像头不再看成一个点，而是一个小圆盘，再确定一个焦距。光线都从圆盘上随机采样发出，经过焦点后在射入场景内，后续的操作没有变化。除了焦平面上的点外，其它平面上的点都是多个平面上的点对应于一个像素，因此产生了模糊效果。如果光圈的半径趋近于 0，则等效于原始没有光圈的情况。

一个例子如下，焦平面在中间的球处。可以看出，中间的球最清晰，前后的球都产生模糊效果：



另一个例子如下，焦平面在前面的龙处：



### 2.4.3 贴图

在光线求交的过程中顺便求出交点在几何体上用参数方程表示的坐标  $uv$ ，然后使用  $uv$  在预先准备好的贴图上访问下标即可。

GPU 上的贴图可以使用 GPU 的纹理内存，其访存效率高于访问普通内存，而且 CUDA 的运行时库还定制了一些常用的贴图效果，例如可以选择让下标越界自动转化成回绕访问，这样可以减少一些数值判断。

一个带贴图的旋转 Bézier 曲面的例子如下：



### 2.4.4 色散

光在折射和薄膜干涉时都会发生色散，后者需要利用光的波动性，在现有框架下实现起来不太现实，这里就只实现了在折射时的色散。

实现思路是先预处理出一张频率到 RGB 颜色的映射表，随机选择一个频率，将其对应的颜色作为此次折射的表面颜色，利用频率来计算折射率。折射率的计算并不要求精确，只需要大致反映出（可见光范围内）随着频率增大，折射率增大的事实即可。因为这个随机选择的存在，随机采样需要更多采样次数才能达到将方差降到理想的范围内了。事实上原本大约 8000 次就能达到不错效果的 PathTracing，这时至少要 20000 次才能比较理想，不过得益于 GPU 的加速，这也是不难做到的。这里截取生成的代码中的一个片段：

```

u32 sel = rng.gen_u32() % 1150;
res.col = RGB_TABLE[sel] * 3;
res.n = 1.7 - (sel / 1150.0f) * 0.5; // 紫 => 1.7, 红 => 1.2

```

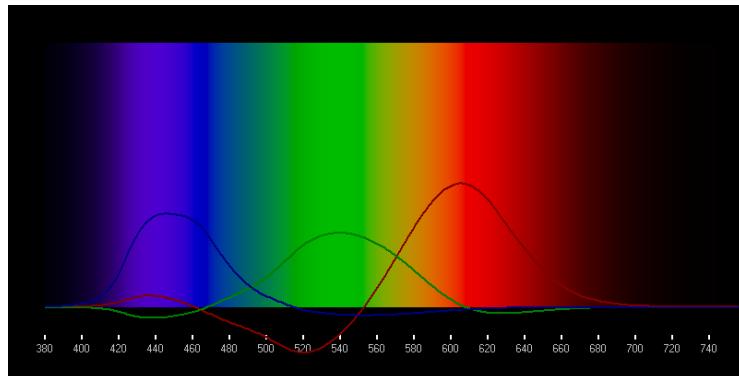
其中 `RGB_TABLE` 的生成方法如下，这代码参考了[Convert light frequency to RGB?](#)中的一些回答，其思路大致是在 RGB 的三个分量上轮流步进，遍历颜色空间：

```

std::vector<Vec3> RGB_TABLE = [] {
    std::vector<Vec3> ret;
    auto range_ck = [](i32 x) { return x > 255 ? 255 : x < 0 ? 0 : x; };
    i32 r = 0, g = 0, b = 255;
    i32 r_f = 0, g_f = 0, b_f = 1;
    while (true) {
        ret.push_back(Vec3{r / 255.0f, g / 255.0f, b / 255.0f});
        if (b == 255) g_f = 1;
        if (g == 255) b_f = -1;
        if (b == 0) r_f = 1;
        if (r == 255) g_f = -1;
        if (g == 0 && b == 0) r_f = -1;
        if (r < 127 && g == 0 && b == 0) break;
        r = range_ck(r + r_f);
        g = range_ck(g + g_f);
        b = range_ck(b + b_f);
    }
    return ret;
}();

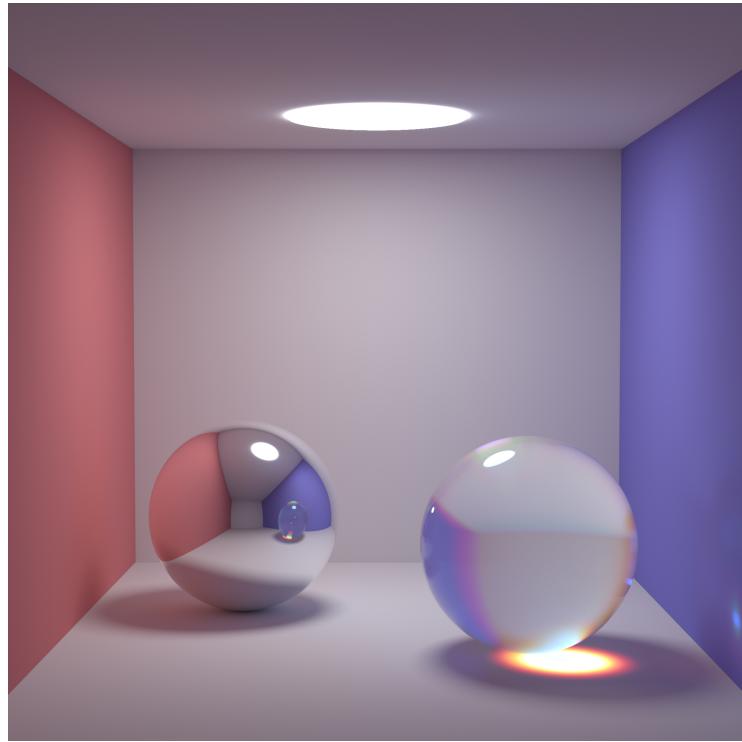
```

其原理在于，将太阳光谱在 RGB 坐标下分解，可以得到如下频率-RGB 图谱：

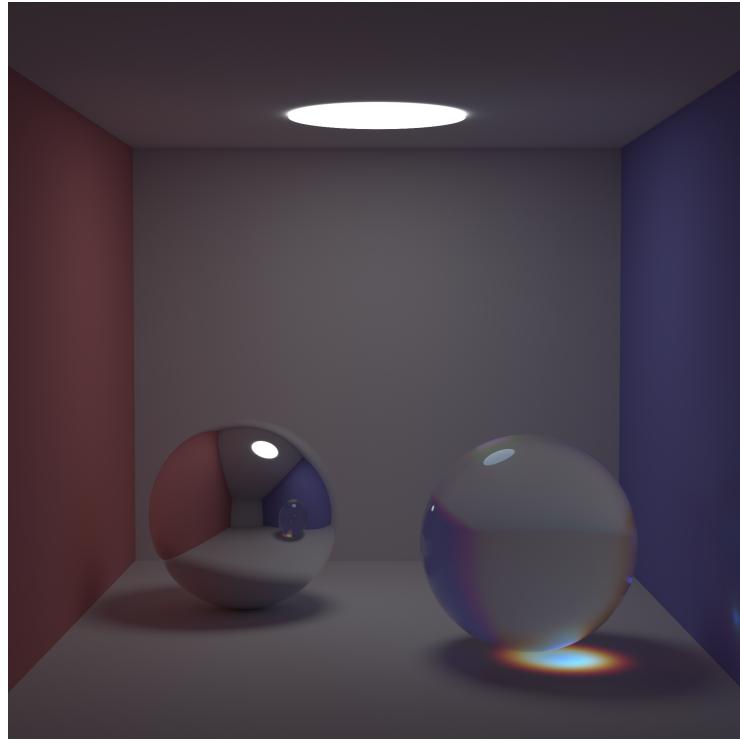


而 RGB\_TABLE 的计算方法则是用几根直线去近似 RGB 三个分量的变化。

一个例子，可以看出色散，但是聚焦的中心因为太亮了显示为白色的：



调暗了光线，可以看到光谱中的各个颜色：



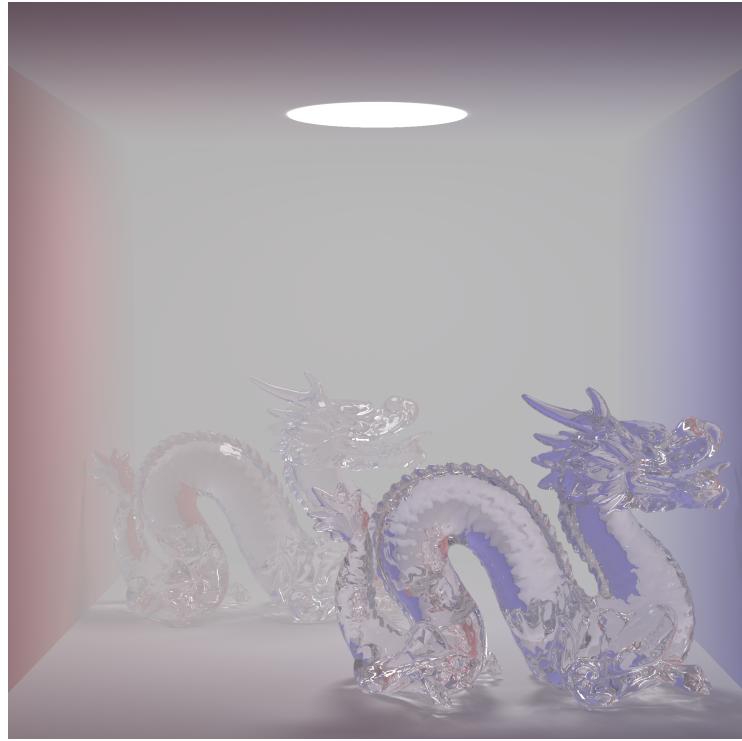
#### 2.4.5 雾

参考了[这篇博客](#)，实现雾的主要原理来自于以下两个式子：

$$f_{atmo}(d) = e^{-(\rho d)^2}$$

$$I = f_{atmo}(d)I_{obj} + [1 - f_{atmo}(d)]I_{atmo}$$

这些都可以比较直接的对应到代码上，其中  $\rho$  和  $I_{atmo}$  是两个需要调整的参数，对于不同尺寸的图也许最佳值不一定相同。一个例子如下，可看出前面的龙比后面的龙清晰受雾的影响小一些。

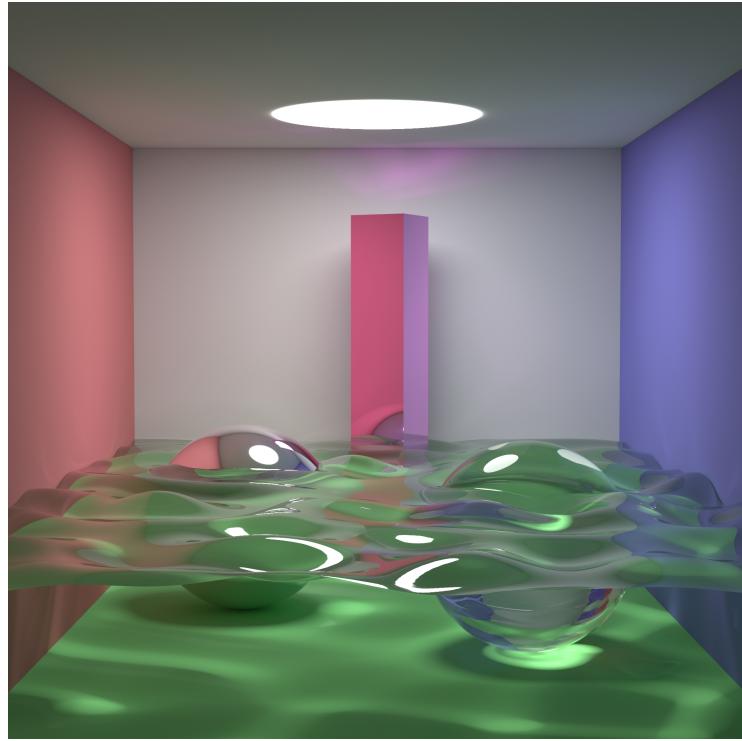


#### 2.4.6 水波效果

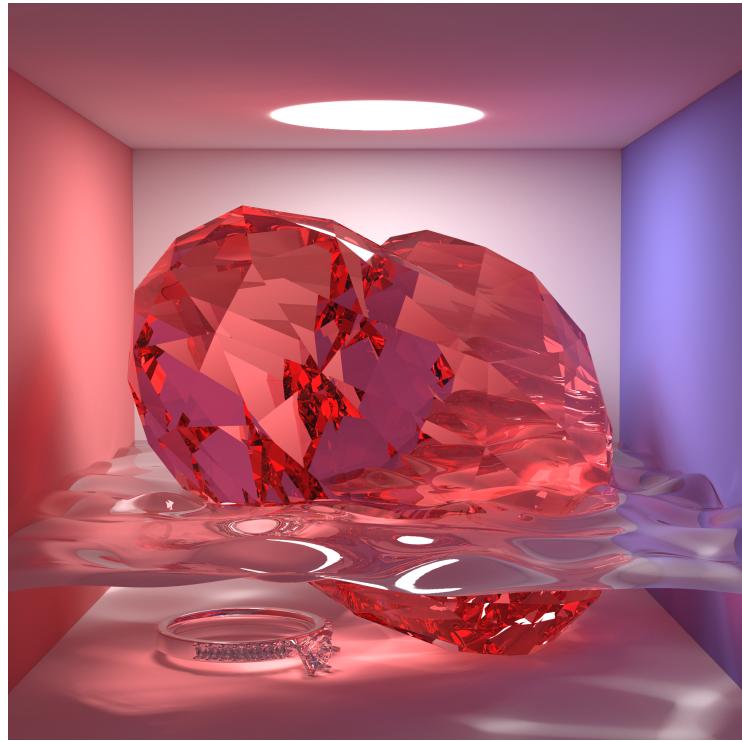
写了一个额外的小工具，用于生成一个包含水波效果的 obj 文件。最终还是转化成 mesh 处理，本质上没有什么新东西。

根据傅里叶变换的原理，水波可以转化成多个幅度，频率，相位不同的平面简谐波。由于我们的目的仅仅是生成一个看起来比较真实的水面，而非对一个已经给定的水面进行近似，所以没有必要进行傅里叶变换，只需要随机选择一些水波的参数再大力调参即可。

一个例子如下，可以看出水中反射了紫色的柱子的影子：



另一个例子如下，可以更强一些的焦散效果；

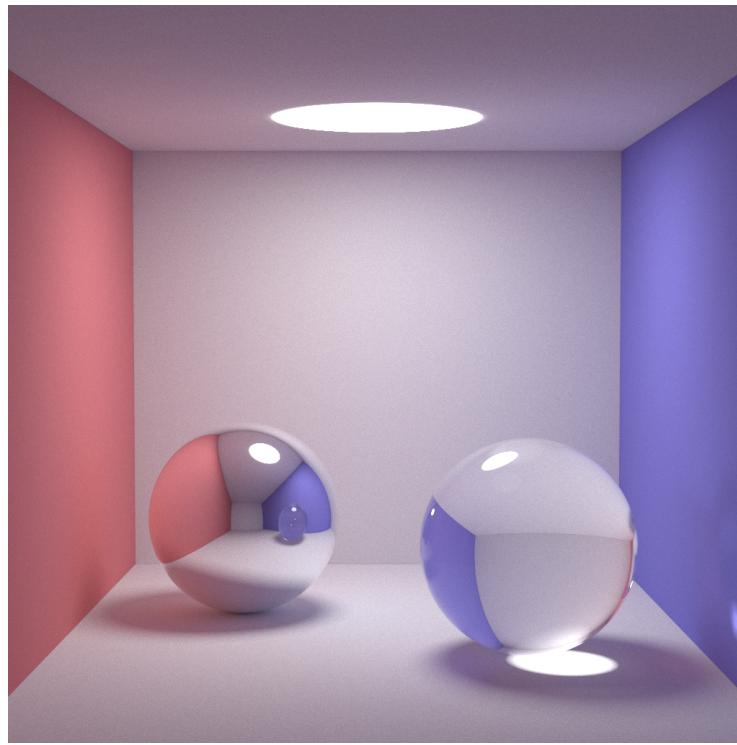


### 3 加速

#### 3.1 硬件加速

##### 3.1.1 GPU 加速

我自学了基本的 CUDA 编程的方法，在我的显卡上渲染了这个简单的场景。分辨率 1024px \* 1024px，每个像素点发射 8192 条光线，大约 8s 内完成渲染。简单计算一下，渲染速度达到了  $1024 * 1024 * 8192 \div 8 \approx 1.07 * 10^9 (\text{rays/s})$ 。作为对比，使用我的 CPU 耗时约 4min35s，使用服务器 CPU E5-2680 v3，耗时约 3min30s（“AMD, No!”）。我的显卡相比于我的 CPU 的性能提升为  $\frac{\frac{1}{8s} - \frac{1}{4min35s}}{\frac{1}{4min35s}} = 3337.5\%$ 。



实际上基本的 CUDA 编程很简单，与 C++ 没有太大的区别，不过为了能够让程序运行地更高效，还需要额外了解一些 CUDA 的内部原理。例如：

1. 使用 `__ldg` 将内存读取到 constant cache，从而加快后续访存速度
2. 使用 GPU 的纹理内存来实现贴图
3. 改变线程之间工作量的分配方式，从 CPU 的一个线程完整负责一个像素点修改成了多个线程分摊一个像素，这是为了让一个 sm 中的线程的所执行的工作尽量相似
4. 需要修改算法的实现形式，例如 PathTracing 和 PPM 的两个 pass 都不能使用递归，全部用一个循环来代替；KDTree 也不能再用简单的递归实现，必须手动显式地维护栈，栈的深度也比 CPU 上更加受限。详见[KDTree](#)。

还有很多别的细微的改变，具体实现方式可以参考我的代码。

在 GPU 加速如此好的效果下，我可以拿它来做一些有趣的东西，比如说渲染几个视频。以其中的 gravity.mp4 为例，该视频总共 10s，帧率为 60，总计张 600 图片，渲染耗时约 1h25min，平均每张图片耗时 8.5s(考虑到编译和 IO 时间，每张图片渲染实际耗时应该小于 8s)，参数都是 1024px \* 1024px \* 8192spp。没有做什么额外的技巧，只是简单地重复渲染一帧一帧的图片。

需要承认的是，虽然我花了很多时间来实现和优化 GPU 上的 mesh 渲染，包括 KDTree 部分着重讨论了这个问题，但是 GPU 渲染 mesh 的速度提升还是达不到 GPU 渲染普通图形的速度提升，只能达到 CPU 的四倍左右。我认为这可能是因为 GPU 本身不太擅长复杂逻辑的处理，而访问 mesh 需要在 KDTree 上进行搜索，其中包含一定的逻辑判断。在 KDTree 查找过程中，内存访问的连续性也不如渲染简单图形时好，而 GPU 显存的特性在于高吞吐，高延迟，如果每次只访问一小块区域，则访存效率可能不够理想。

### 3.1.2 多线程加速

虽然习题课上已经约定好了使用 `openmp` 这种非常 naive(但也很实用) 的多线程加速不会获得加分，这里还是顺便提一句我使用了它。

除了在 C++ 版本上使用了 `openmp` 加速外，在原始的 rust 版本上，我另外使用了一个 rust 的并行库 `rayon` 来实现多线程加速。

虽然图形渲染这个工作很适合并行处理，但是写代码还是要一直保持线程安全的思想，尽量减少或消除共享可变变量，这也是 rust 提倡的编程思想。

## 3.2 算法加速

### 3.2.1 KDTree

rust 在生成 KDTree 的时候采用了按照方差最大的维度分割的策略，这样可以尽可能地保证分割的均匀性。

C++/CUDA 中的 KDTree 节点数据结构如下：

```
struct TriMat {
    f32 m00, m01, m02, m03;
    f32 m10, m11, m12, m13;
    f32 m20, m21, m22, m23;
};

struct KDNode {
```

```

Vec3 min, max;
union {
    struct { // leaf, len = actual len | (1 << 31)
        u32 len;
        TriMat ms[0]; // also store n & uv after ms
    };
    struct { // internal
        u32 ch1, sp_d;
        f32 sp;
    };
};
};

```

其中 `min/max` 定义了包围此节点的 AABB，用于和光线求交的时候进行剪枝。下面的一个 `union` 为叶子/内部节点的数据存放处，采用字段 `len` 的最高位来标记叶子/内部节点，如果最高位为 1 则为叶子，此时字段 `ms` 有效，否则为内部节点，此时字段 `ch1, sp_d, sp` 有效。

`ms` 是一个柔性数组，它存储了用于三角形-光线求交的三角形矩阵信息，详见[三角形求交算法](#)。实际上不仅 `ms` 的长度是可变的，它的末尾处还会接上存储法向量，参数 `uv` 的数组。`ch1` 表示本节点的右侧孩子相对于根节点在内存中的偏移量，`sp_d, sp` 分别表示分割维度和分割坐标。

整个 KDTree 都存储在从根节点开始的一块连续的内存区域中。rust 代码会先生成 KDTree，将它的二进制格式用一个 linux 工具 `ld` 做成一个.o 文件，在 C++/CUDA 代码中链接此.o 文件即可，使用方法大致是：

```

// 实际上用 ld 生成的.o 文件的符号信息依赖于平台
// 不保证其它平台上也是这个名字，不过我也不在乎跨平台性
extern const KDNode _binary_mesh0_start;
kd_node_hit(&_binary_mesh0_start, ray, res, 2, Vec3{0.999, 0.999, 0.999});

```

以上是 KDTree 的数据结构，在此基础上我实现了[Interactive k-D Tree GPU Raytracing](#)中的 short-stack 算法。算法伪代码如下：

#### A.4 Modifications for short-stack traversal

```
tMin=tMax=sceneMin tHit=infinity
while (tMax<sceneMax):
    if stack.empty():
        node=root
        tMin=tMax
        tMax=sceneMax
        pushdown=True
    else:
        (node,tMin,tMax)=stack.pop()
        pushdown=False
    while (not node.isLeaf()):
        a = node.axis
        tSplit = (node.value - ray.origin[a] ) / ray.direction[a]
        (first, sec) = order(ray.direction[a], node.left, node.right)
        if( tSplit ≥ tMax or tSplit < 0)
            node=first
        else if( tSplit ≤ tMin)
            node=second
        else
            stack.push(sec,tSplit,tMax)
            node=first
            tMax=tSplit
            pushdown=False
        if pushdown:
            root=node
    for tri in node.triangles():
        tHit=min(tHit,tri.Intersect(ray))
        if tHit<tMax:
            return tHit //early exit
return tHit
```

对算法原理简要解释如下：

非递归的 KDTree 查找算法大致可以分成两个思路：第一个思路是模拟调用栈，在其中压入所有递归的查找算法所必须的信息。其优点在于计算量完全等同于递归的查找算法。然而，具体实现时，在 CPU 上还算可行，但是在 GPU 上效率会严重降低，原因在于根据GPU 内存模型，GPU 的 local memory 所处的地位并不完全等同于 CPU 的 stack，相对各自平台的其它存储层级 (cache/register) 而言，前者的速度要远慢于后者。所以如果在 GPU 上开太大的数组，速度是会受较大影响的。第二个思路是重启。它并不保存路径上遇到的需要访问的兄弟节点，一旦在叶子节点处查找失败，则重启查找过程。下一次查找之所以可能和前一次不同，是因为查找过程中更新了光线的参数  $t$  的上下界，这个界会用来决定访问哪个节点。其优点在于只需要  $O(1)$  的额外空间，但是它显然引入了额外的计算量。一个简单的优化 (push-down) 是，重启查找并不从根节点开始，而是从第一次光线同时位于两个子树的节点开始。

short-stack 算法整合了这两种实现方法的优点。它使用一个小小的，固定大小的栈。当栈已满时若需要 push 节点，则丢弃栈底部的节点；当栈空时若需要 pop 节点时，则触发重新启动。这个栈实际上起到一个缓存的作用，希望能在尽量不使用 GPU local memory 的情况下减少计算量，当然对于 CPU 来说也存在一定价值，例如较小的栈可能可以装进一个 cache line，访存的速度也会更快。

根据测试，short-stack 算法的确很明显地提升了 KDTree 的查找效率。事实上，在使用这个算法之前我实现了 OctTree，也按照常见的写法实现了 KDTree，但是前者大约比后者快两倍；

而在使用了这个算法后，变成了 KDtree 明显快一些，所以现在 OctTree 已经不再使用了。

### 3.2.2 OctTree

我曾经使用过 OctTree 来组织三角形 mesh，现在已经不再使用，但是还是简单介绍一下实现思路和剪枝方法。

我预先定义了这样一个数组，表示光线在 OctTree 中的搜索顺序。其中数字 0-7 为三维空间中的八个卦限的编号，二维数组中的每个一维数组内排序的依据是：数组元素与一维数组所在下标的不同的二进制 bit 数。

```
pub const PROB: [[u8; 8]; 8] = [
    [0, 1, 2, 4, 3, 5, 6, 7],
    [1, 0, 3, 5, 2, 4, 7, 6],
    [2, 0, 3, 6, 1, 4, 7, 5],
    [3, 1, 2, 7, 0, 5, 6, 4],
    [4, 0, 5, 6, 1, 2, 7, 3],
    [5, 1, 4, 7, 0, 3, 6, 2],
    [6, 2, 4, 7, 0, 3, 5, 1],
    [7, 3, 5, 6, 1, 2, 4, 0],
];
```

在八叉树中查找时，按照这个搜索顺序进行搜索

```
let mid = (self.aabb.min + self.aabb.max) / 2.0;
let index = ((ray.o.0 > mid.0) as usize) +
            (((ray.o.1 > mid.1) as usize) << 1) +
            (((ray.o.2 > mid.2) as usize) << 2);
for child_idx in &PROB[index] {
    if let Some(result) = children[*child_idx as usize]
        .hit_impl(ray, inv_d, mesh) {
        return Some(result);
    }
}
```

分析得知，如果光线的起始点位于某个卦限，按照这个编号对应的搜索序列对孩子节点递归搜索，只要在某个孩子中找到交点，这交点一定是此光线与本节点所有三角形的交点中最最近的，便可以立即返回，而不用遍历完整的八个子树。

### 3.2.3 Hash 网格

用于在 PPM 算法中存储和组织交点。之所以使用 KDTree 而使用 Hash 网格，原因在于后者的性能更好。但是它不适用于光线求交的场合，只适用于为某点查询临近点，所以没有用它来组织 Mesh 中的三角形。

根据 PPM 算法的要求，每次光子与理想粗糙表面发生碰撞时需要统计碰撞点附近的 Ray-Tracing Pass 留下的交点信息。RayTracing Pass 结束后，初始化 Hash 网格，为每个交点设置一个初始半径，选择一个散列表大小，将一个交点分配到所有与它相交的网格中。为了尽可能使内存紧凑，这里进行一个两遍的统计操作：先统计出每个网格包含多少个交点，用另一个额外的下标数组来表示每个网格的起始下标，然后分配一片大的连续内存区域，分别填入每个交点。经测试，这样相比于为每个网格都分配自己的动态内存可减少一半左右的内存消耗。

得到 Hash 网格后在 Photon Pass 中，直接查询与碰撞点所在网格相交的交点。理想情况下（散列碰撞尽可能少，每个交点所管辖的半径都较小），Hash 网格能够达到理论最优的复杂度  $O(r)$ ， $r$  表示实际上需要处理的交点数（这数目很可能并不小，但是这已经不能再优化了）。每轮 Photon Pass 结束后，由于交点的半径可能发生了变化，所以需要重建 Hash 网格。

值得注意的是，交点的半径和光强的更新，这都要求必须对某个共享的变量进行同步的修改操作，严格来说应该使用原子变量或者锁来保护。但是考虑到图形学的目标并不是完全的正确性，而是以尽量高的效率达到尽量好的效果，所以 Photon Pass 中没有做任何相关的保护（RayTracing Pass 有一个地方使用了 `mutex` 进行保护：每个线程向一个共享的 `std::vector<HitPoint>` 中添加元素时，当然这个操作可以用线程私有变量 + 规约之类的方案来减少同步互斥的开销，但是这并非性能瓶颈，没什么优化的必要）。

### 3.2.4 低差异序列

用于在 PPM 算法中生成随机数。之所以在 PPM 算法中使用，主要原因是 PPM 算法的随机数采样次数较少，如果仍然采用传统的随机数方法（对于 PathTracing 而言，我采用了 Xorshift 方法来获取随机数），会导致较大的方差，在现有的交点数和光子数下不能有效收敛。而如果使用低差异序列，例如 Halton sequence 则能达到理想的效果。一个素数对应于一个 Halton sequence，在其上进行采样会得到一系列很明显不随机的数字，但是它们基本是在  $(0, 1)$  上均匀分布的，所以只要在每一个需要发生随机数的地方，使用同一个 Halton sequence 来生成相对不太的随机数，就可以达到比较均匀的采样效果。

我也尝试过在 PathTracing 中使用它，但是相比于原来的随机数生成方法并没有明显的提升，我猜测可能是因为 PPM 的亮度贡献主要来自于光子碰撞点处的计算，而随机采样并不是主要决定因素；但 PathTracing 主要依赖于随机采样和后续的光线求交过程。

Halton sequence 的实现伪代码如下：

```

algorithm Halton-Sequence is
    inputs: index i
              base b
    output: result r

    f  $\leftarrow$  1
    r  $\leftarrow$  0

    while i  $>$  0 do
        f  $\leftarrow$  f/b
        r  $\leftarrow$  r + f * (i mod b)
        i  $\leftarrow$   $\lfloor i/b \rfloor$ 

    return r

```

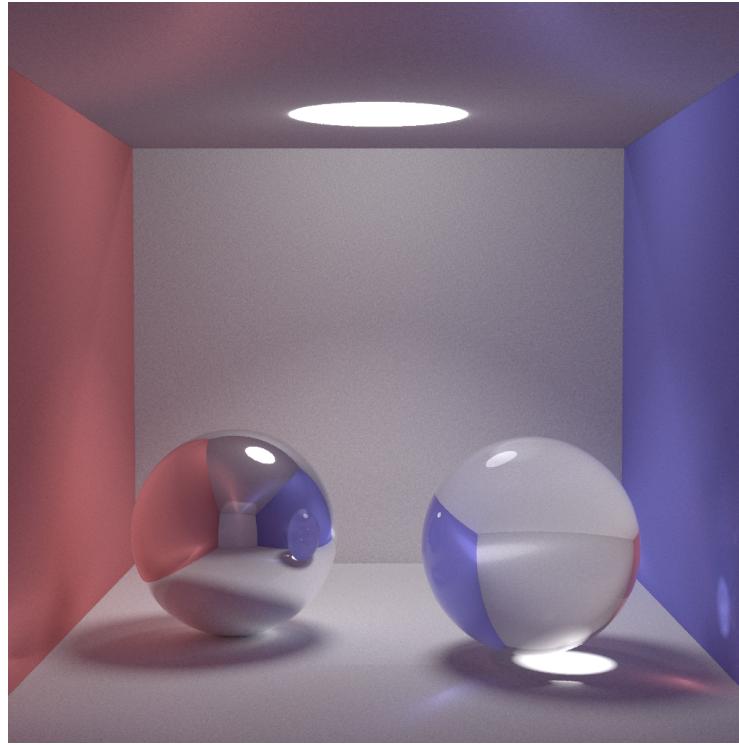
例如，对于  $base = 2$ ,  $index = 1, 2, \dots, 10$ , 算法会输出：

0.5 0.25 0.75 0.125 0.625 0.375 0.875 0.0625 0.5625 0.3125

可见它的输出规律大致是按照一定的步长反复遍历  $(0, 1)$ , 每次的起始点略有差异。

具体实现时，我进行了常数除法优化，具体优化方法参见[编译器是如何实现 32 位整型的常量整数除法优化的？](#)。在目前大多数的硬件平台上，整数除法/取模都是一个非常昂贵的操作，但是对于常整数的除法/取模，可以预先计算一些中间变量，然后使用乘法/加法/位运算来代替除法/取模。一般而言这个优化会被编译器自动完成，这里之所以需要自己手动执行，是因为这里的“常数”只是在运行时不会改变的数，并不是在编译期就已知的数。也许未来某一天这种级别的优化也可以由编译器来代劳，但至少目前还是需要人类智慧的。

需要注意的是，低差异序列的使用需要比平常的随机数更加小心。如果在不同的随机数发生点采用了同一个素数对应的 Halton 生成器，二者产生的结果就会有很强的关联性，这会导致比较严重的 artifact。即是不是用一个素数，二者可能也会有很强的关联，例如 wikipedia 上给出来的例子：对于 17 和 19，生成的前 16 个数字都是成比例的。因为这个原因，我曾经在调试代码时渲染出来的质量不好的一张图：



原因就是有两个地方错误使用了同一个 Halton 生成器，因此过于强化了某些原本应该是相对均匀分布的光效，所以产生了过度的焦散效果（所以或许这图也有一点独特的美感？）。

### 3.2.5 三角形求交算法

我实现了[Fast Ray-Triangle Intersections by Coordinate Transformation](#)中提出的光线-三角形求交算法。

其大致思路为，使用一个变换矩阵可以将一个三点坐标为  $(0, 0, 0)$ ,  $(1, 0, 0)$ ,  $(0, 1, 0)$  的三角形 (canonical right triangle) 变换到待求交的三角形，如果预先求出这个矩阵的逆，在光线求交时将逆矩阵应用在光线上，就可以转化成一个光线和简单三角形的求交问题。

由于变换矩阵中存在一些自由项，论文中讨论了三角形法向量元素的模的最大值的分布的 4 种情况，目的是尽量减小数值误差。因为这个步骤是在渲染前做的，所以耗时稍微高一些也没有关系。代码大致如下：

```
if norm.0.abs() > norm.1.abs() && norm.0.abs() > norm.2.abs() {
    m[0][0] = 0.0;
    m[1][0] = 0.0;
    m[2][0] = 1.0;
    m[0][1] = e2.2 / norm.0;
    m[1][1] = -e1.2 / norm.0;
```

```

m[2][1] = norm.1 / norm.0;
m[0][2] = -e2.1 / norm.0;
m[1][2] = e1.1 / norm.0;
m[2][2] = norm.2 / norm.0;
m[0][3] = p3.cross(p1).0 / norm.0;
m[1][3] = -p2.cross(p1).0 / norm.0;
m[2][3] = -p1.dot(norm) / norm.0;
} else if norm.1.abs() > norm.2.abs() {
    ...
} else if norm.2.abs() > 0.0 {
    ...
} else {
    // degenerate triangle
}

```

最终的逆矩阵包括 13 个可能非 0 的位置，其中一个位置固定为 1，所以只需要存储 12 个浮点数即可描述该矩阵，这也就是 TriMat 中的 12 个浮点数。空间消耗相比于直接存储坐标增加了  $\frac{1}{3}$ ，还算可以接受。

这个算法渲染时的浮点运算次数略少于广为人知的 Möller-Trumbore 光线-三角形求交算法，同时代码比较紧凑，分支判断较少，编译器也比较容易自动地使用 SIMD 来优化这个矩阵向量乘法。

经测试，对于 Mesh 的渲染，相比于 Möller-Trumbore 算法，这个算法有大概 5% 的性能提升。这个提升其实不是很明显，原因在于论文中提到，这个算法的主要优势在于可以将对于场景应用的变换矩阵预先乘到三角形矩阵上，从而可以将变换 + 求交压缩成一次变换，这样节省的计算量就比较可观了。但是在我的算法框架中，没有在渲染时再对场景应用变换矩阵的需求，所以提升就比较有限了。