

# 作业二报告

## 项目概要

[https://github.com/xalang/cg\\_tracing](https://github.com/xalang/cg_tracing)

## 环境

整个项目用 Rust 编写，使用 Rayon 库的多线程加速（除了速度比 C/C++ 差点外，Rust 比 C/C++ 好用太多了）。

- cargo 1.35.0 (6f3e9c367 2019-04-04)
- rustc 1.35.0 (3c235d560 2019-05-20)
- stable-x86\_64-pc-windows-msvc

## 代码结构

assets/	资源文件
example/	某些配置文件和样例
result/	成果以及相应的配置文件
src/	
-- macros.rs	一些宏
-- main.rs	可执行程序入口代码，默认读入 ./example/test.json 作为配置文件
-- prelude.rs	引用后便能调用项目大部分功能
-- lib.rs	
-- geo/	包含物体渲染所需要的结构
---- mod.rs	定义了物体所需要实现的 trait，即要实现的接口
---- texture.rs	材质
---- collection/	各种物体的具体实现
----- bezier.rs	Bezier 曲线/曲面
----- mesh.rs	三角网格
----- mod.rs	
----- plane.rs	平面
----- sphere.rs	球
----- ds/	物体所用到的数据结构
----- bbox.rs	包围盒
----- bsptree.rs	BSP-Tree
----- kdtree.rs	KD-Tree
----- mod.rs	
-- linalg/	线性代数数学工具
---- mat.rs	4x4 矩阵以及一些矩阵变换
---- mod.rs	
---- ray.rs	光线（射线）
---- transform.rs	存储物体的一系列矩阵变换及逆变换
---- vct.rs	三维向量
-- scene/	场景
---- camera.rs	摄像机
---- mod.rs	
---- sppm.rs	渐进光子映射所需要的数据结构
---- world.rs	实现了路径追踪和渐进光子映射算法
-- utils/	一些常用工具

---- images.rs

存储材质图片的结构

---- mod.rs

随机数生成器、常用函数等

所有图形渲染的参数都能用 JSON 格式的配置文件来设置，整体代码结构也尽量保持了清晰易懂、尽量降低了不同文件之间的耦合性、尽量减少了一些常数开销。

## 使用方法

见 [README.md](#)

## 得分点

- PT/SPPM
- 网格化求交 / Bezier 参数曲面求交
- 算法型加速：Bezier 线性求多项式系数、下山牛顿迭代、KD-Tree/BSP-Tree、快速三角面求交、随机数生成器
- 景深、软阴影、抗锯齿、贴图

## 一张比较满意的图



配置文件见 [result\\_6.json](#)（注意，白光在正上方，视野外左边有红墙，右边有紫墙，所以有泛红和泛紫效果，同时焦平面在龙那里，所以红球模糊）。

## 功能实现

# 渲染算法

## Path Tracing

代码部分参考了 [small-pt](#) 这个项目。

原理是从摄像机成像面的某个像素点不断发出方向随机的射线，射线在经过不同材质表面的物体反射、折射后，最终抵达光源，然后回溯求出色彩信息。

可以发现射线抵达光源是一个小概率事件，因此该算法需要大量采样才能达到令人满意的效果。一般来说，一个像素点采样 8192 次就差不多了。同时由于反射、折射可能无限进行下去，因此还需要限制一下深度，简单场景设置 5 就差不多了。

由于 PT 是超采样的，所以自然就达到了软阴影的效果，同时也有了抗锯齿的效果。

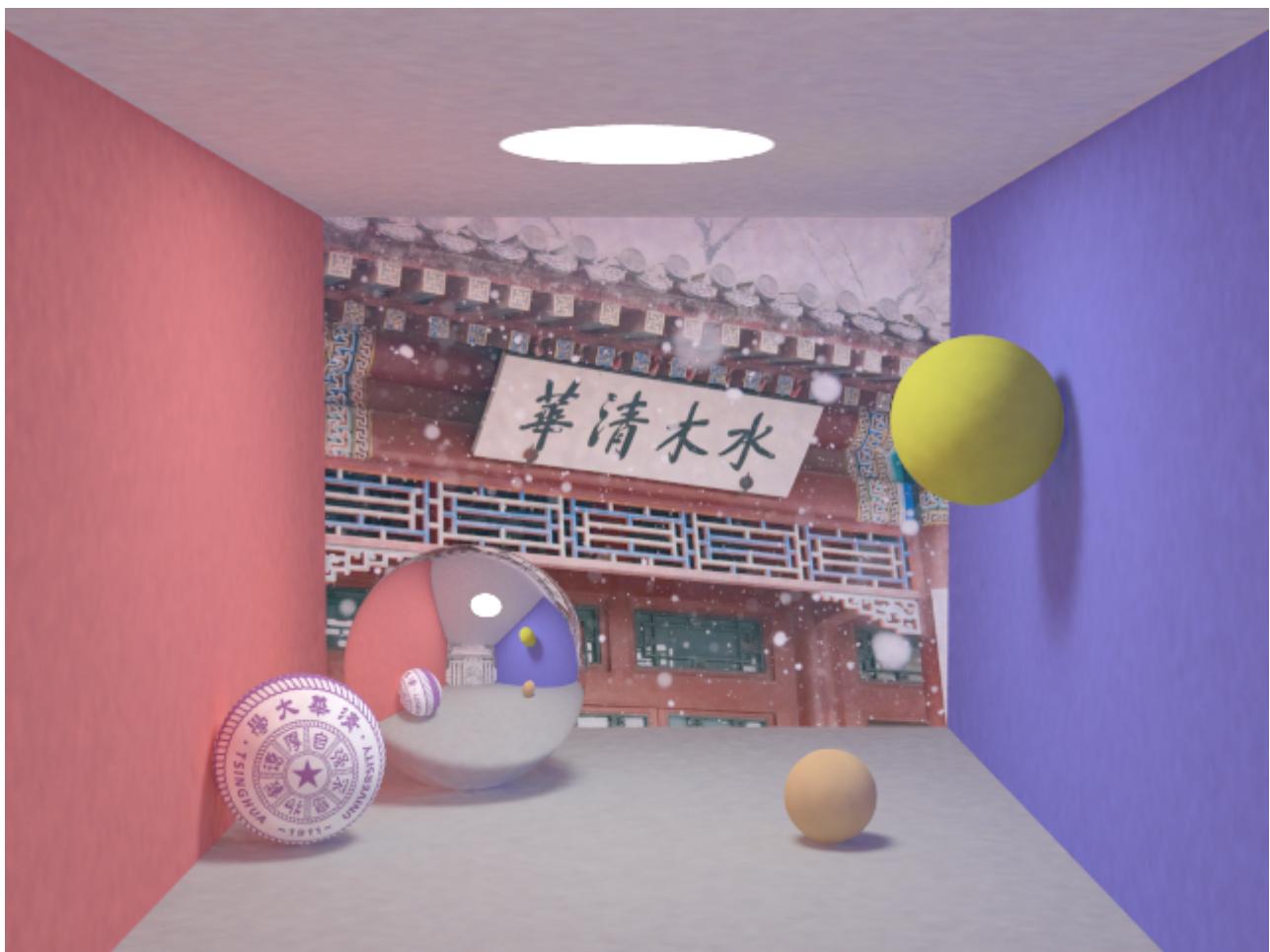
具体实现见 [src/scene/world.rs](#) 第 183 行以后。

## Progressive Photon Mapping

见论文 [Progressive Photon Mapping](#)

原理是先像路径追踪那样从摄像机发送射线，不同的是，PPM 在第一次漫反射的地方停止，并记录交点。然后再从光源随机发出光子，若光子打到了之前记录的某个交点的一个半径为 R 的球内，则统计该光子对这个交点的贡献。同时根据统计次数的增多，这个半径 R 应当适当减小。当光子足够多时，则这些贡献累加起来则会趋向于真实值。

## Stochastic Progressive Photon Mapping



配置文件见 [result\\_7.json](#) (这是在检查之后写的，由于实在没时间跑了，就跑了个低分辨率的图，然后光子数比较少、迭代次数比较少，所以还有点噪)。

见论文 [Stochastic Progressive Photon Mapping](#)

与 PPM 不同的是，统计的不再是对某个交点的贡献，而是对一个区域的贡献（比如说成像面的像素点），其余的和 PPM 大致相似。

交点我用 KD-Tree 来维护，然后每个像素点的信息用以下公式维护， $\alpha = 0.7$ ：

Our idea is to use shared statistics over a region that we would like to compute the average radiance value for. Using the shared statistics, the stochastic progressive radiance estimate approximates the average radiance value  $L(S, \vec{\omega})$  over the region  $S$  as:

$$L(S, \vec{\omega}) \approx \frac{\tau_i(S, \vec{\omega})}{N_e(i)\pi R_i(S)^2}, \quad (7)$$

where  $i$  is the number of photon passes as before,  $\tau_i(S, \vec{\omega})$  is the shared accumulated flux over the region  $S$ , and  $R_i(S)$  is the shared search radius. The updating procedure of the shared statistics is:

$$N_{i+1}(S) = N_i(S) + \alpha M_i(\vec{x}_i) \quad (8)$$

$$R_{i+1}(S) = R_i(S) \sqrt{\frac{N_i(S) + \alpha M_i(\vec{x}_i)}{N_i(S) + M_i(\vec{x}_i)}} \quad (9)$$

$$\Phi_i(\vec{x}_i, \vec{\omega}) = \sum_{p=1}^{M_i(\vec{x}_i)} f_r(\vec{x}_i, \vec{\omega}, \vec{\omega}_p) \Phi_p(\vec{x}_p, \vec{\omega}_p) \quad (10)$$

$$\tau_{i+1}(S, \vec{\omega}) = (\tau_i(S, \vec{\omega}) + \Phi_i(\vec{x}_i, \vec{\omega})) \frac{R_{i+1}(S)^2}{R_i(S)^2}, \quad (11)$$

where  $\vec{x}_i$  is a randomly generated position within  $S$  and  $N_i(S)$  is the shared local photon count. Note that the updating procedure

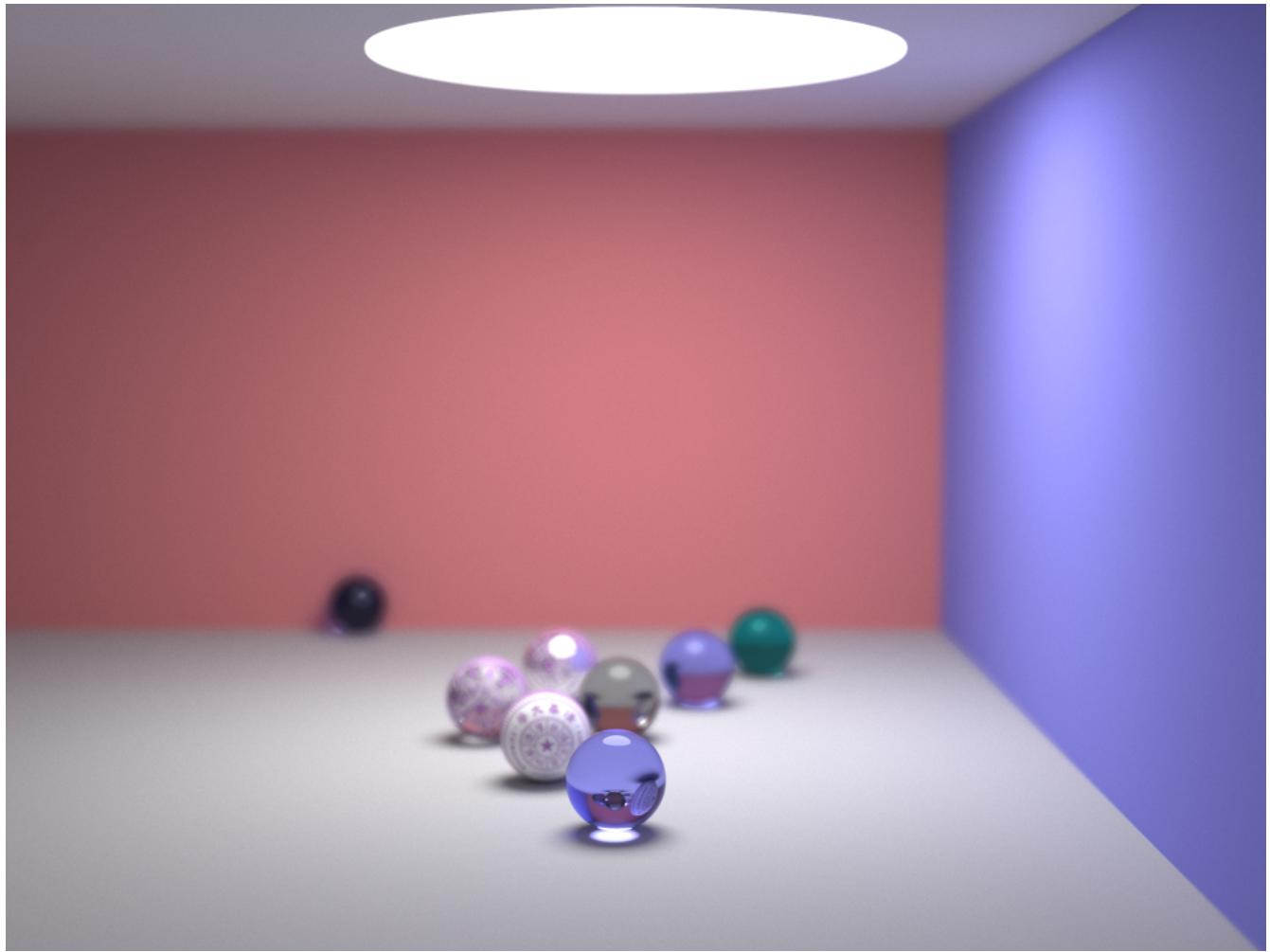
```
"renderer": {
  "type": "sppm",
  "view_point_sample": 4,           // 每个像素点 HitPoint 采样数
  "photon_sample": 300000,          // 光子数目
  "radius": 1,                     // 初始半径
  "radius_decay": 0.95,            // 半径衰减值
  "rounds": 100,                  // 迭代轮数
  "light_pos": { "x": 50, "y": 81.599999, "z": 81.6 },    // 光源位置
  "light_r": 15                   // 光源半径
}
```

具体实现见 [src/scene/world.rs](#) 第 402 行以后。

## 摄像头

```
"camera": {  
    "origin": { "x": 50.0, "y": 50.0, "z": 300.0 }, // 位置  
    "direct": { "x": 0.0, "y": -0.082612, "z": -1.0 }, // 方向  
    "view_angle_scale": 0.5135, // 成像平面的视角比例  
    "plane_distance": 140.0, // 成像平面与摄像机的距离  
    "focal_distance": 100.0, // 修正后的焦距, 成像平面到焦平面的距离  
    "aperture": 0.7 // 光圈半径  
},
```

## 景深



配置文件 [result\\_4.json](#) (在最前面的紫球聚焦)。



配置文件 [result\\_5.json](#) (在中间的紫球聚焦)。

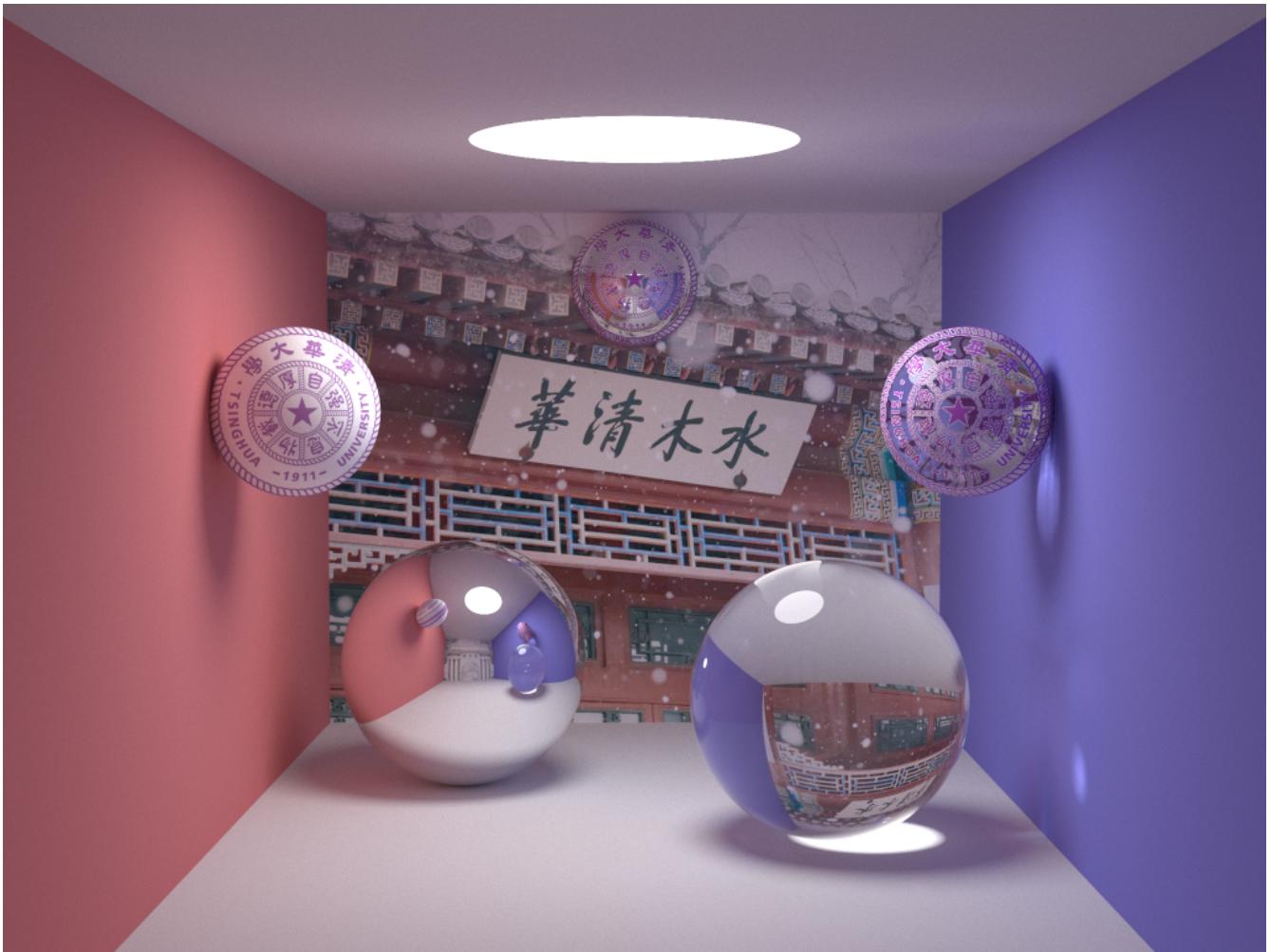
参考的是[这篇文章](#), 大概就是将摄像头看成一个圆而不是一个点, 然后在这个圆上随机生成一个点作为摄像头, 再根据焦距来确定焦平面, 即:

$$o = o_{camera} + r, d = normalize(focal\_distance \cdot normalize(d_{camera})) - r$$

其中  $r$  为上述所说圆内的一个随机向量。

具体实现见 [src/scene/world.rs](#) 第 236 行 (这里还综合了与成像平面的混合操作)。

## 物体求交与贴图



配置文件 [result\\_1.json](#) (中间是反射的球，材质有色部分漫反射，无色部分反射；右边是透明的球，材质有色部分不透明，无色部分透明)。

以下均约定射线  $r = (o, d)$  为  $r(k) = o + kd$ , 其中  $o = (x_o, y_o, z_o)^T$  为起点,  $d = (x_d, y_d, z_d)^T$  为方向向量,  $k > 0$ 。

为了优化一点常数，我的代码里物体求交分为两步：光线求交检测中，先对所有物体只求出  $k$  和临时信息，并不多做额外计算。比较  $k$  确定了碰撞的物体后，再用该物体的  $k$  和临时信息去求法向量和贴图坐标。贴图会全部以 RGBA 的形式读入到内存中，存放到相应的物体结构体里，由于加载图片调的是第三方库，所以支持大部分图片格式。

## 球

设球圆心为  $c$ , 半径为  $R$ , 则解下一元二次方程即可 (但要注意  $k > 0$  且取较小的那个解) :

$$||r(k) - c|| = R^2$$

贴图的话我的做法比较蠢，先将球看成单位球，然后将半球的  $(x, y)$  坐标线性映射到  $[0, 1] \times [0, 1]$ 。

具体实现见 [src/geo/collection/sphere.rs](#)。

## 平面

平面用点-法式描述，点为  $p$ , 法向量为  $n$ 。先算出夹角余弦值，再根据三角关系解出  $k$

$$\begin{aligned} \cos \theta &= n \cdot d \\ k \cos \theta &= n \cdot (p - o) \end{aligned}$$

贴图的话比较简单，设定  $p$  为图片左上角或左下角，然后以交点位置与  $p$  的相对位置作为贴图坐标。

具体实现见 [src/geo/collection/plane.rs](#)。

## 包围盒

包围盒用  $\min, \max$  两个顶点表示，则两个交点解  $k_{\min}, k_{\max}$ （注意与 0 的关系）分别为

$$\begin{aligned} a &= \min\{(min - o)/d, (max - o)/d\} \\ b &= \min\{(min - o)/d, (max - o)/d\} \\ k_{\min} &= \min\{x_a, y_a, z_a\}, k_{\max} = \max\{x_b, y_b, z_b\} \end{aligned}$$

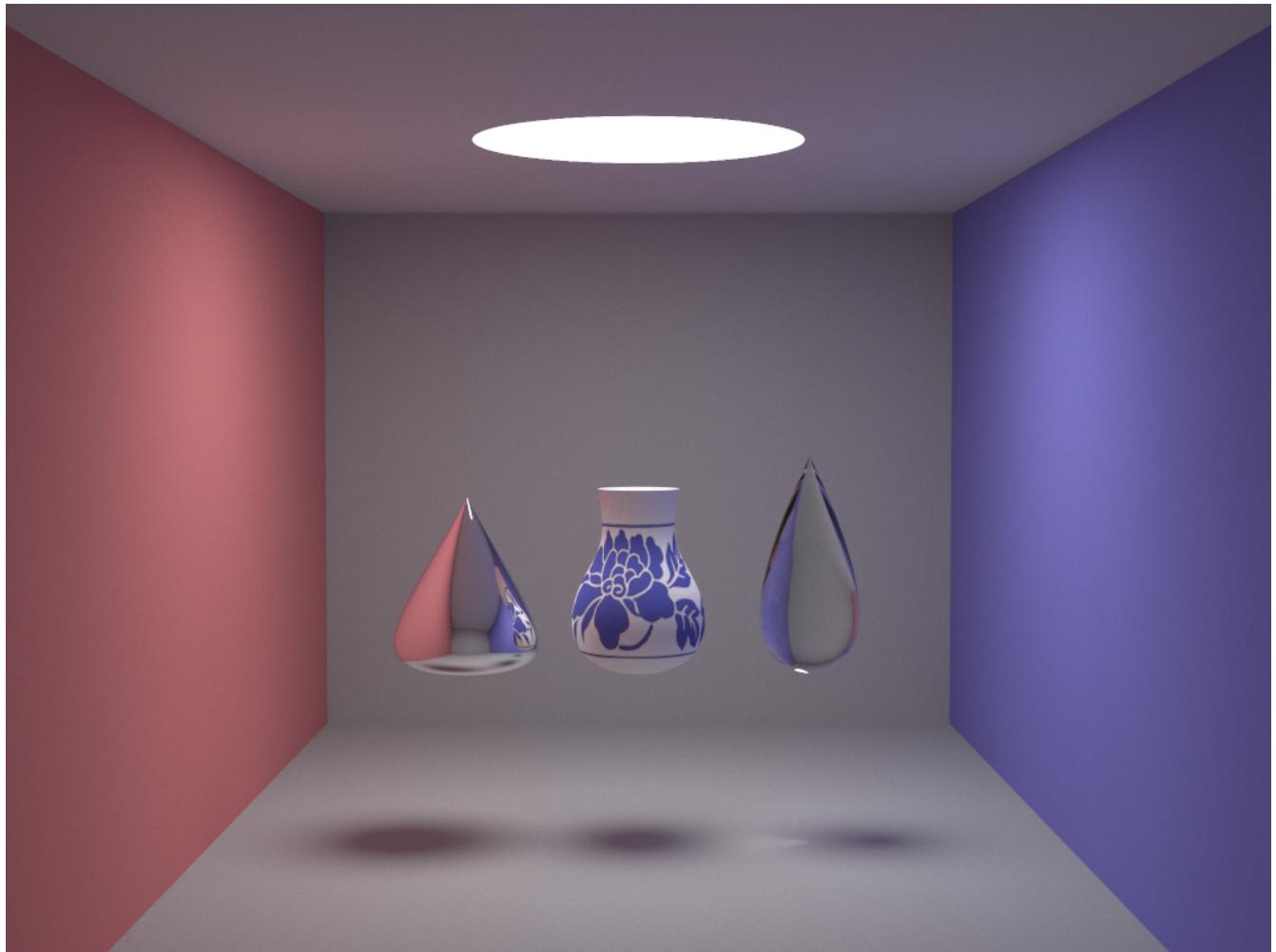
在这里两个向量的除法定义为各个分量作除法后得到的新向量。两个向量取  $\min$  定义为各个分量取  $\min$  后得到的新向量。

但该算法会有较多的比较运算，而且很多无解的情况下根本不需要这么多比较。可以发现上述操作基本都是对每个分量来做的，因此可以将各个分量单独来做并判断。具体实现见 [src/geo/collection/ds/bbox.rs](#) 第 30 行以后。

贴图的话，就 6 个面当平面分别贴一下就好（但我没写）。

长方体可以由一个包围盒通过矩阵变换得到，因此不做讨论（也没写）。

## Bezier 曲线/曲面



配置文件 [result\\_3.json](#)（注意左边反射物体的顶端是光源，底部黑区域是影子，右边折射物体的影子上的光斑是光线折射得到的）。

一个二维  $n$  阶的 Bezier 曲线由  $n + 1$  个控制点  $\{p_i | 0 \leq i \leq n\} = \{(x_0, y_0), \dots, (x_n, y_n)\}$  得到，对于  $0 \leq t \leq 1$ ，有

$$p(t) = \sum_{i=0}^n p_i \binom{n}{i} t^i (1-t)^{n-i}$$

显然这个  $p(t) = (x(t), y(t))$  的两个分量分别是一个  $n$  次多项式，因此可以在  $O(n^3)$  的时间内用拉格朗日插值法求出系数，之后便能在  $O(n)$  的时间内求出  $p(t)$  和  $p'(t)$  了。

但这个系数是可以优美的解出来的，具体看 n+e 同学写的[解法](#)，再根据解法里的式子稍微简化后可以得到

```
let mut a = vec![];
let mut t = 1.0;
for i in 0..=n {
    a.push((x[0] * t, y[0] * t));
    t = t * (n - i) as f64 / (i + 1) as f64;
    for j in 0..n - i {
        x[j] = x[j + 1] - x[j];
        y[j] = y[j + 1] - y[j];
    }
}
```

$a$  数组即系数，就是这么简单！

接下来考虑旋转曲面求交，只考虑 Bezier 曲线绕  $y$  轴旋转的情况，其余情况可以通过矩阵变换得到。

光线与曲面相交当且仅当交点到  $y$  轴的距离等于  $x(t)$ ，因此可以得到

$$k = \frac{y(t) - y_o}{y_d}, x(t) = \sqrt{(x_o + kx_d)^2 + (z_o + kz_d)^2}$$

平方后两边乘  $y_d^2$ ，有

$$y_d^2 x^2(t) = [x_o y_d + (y(t) - y_o) x_d]^2 + [z_o y_d + (y(t) - y_o) z_d]^2$$

则令

$$\begin{aligned} f(t) &= [x_o y_d + (y(t) - y_o) x_d]^2 + [z_o y_d + (y(t) - y_o) z_d]^2 - y_d^2 x^2(t) \\ &= (x_d^2 + z_d^2) y^2(t) + 2[(x_o y_d - y_o x_d) x_d + (z_o y_d - y_o z_d) z_d] y(t) + \\ &\quad (x_o y_d - y_o x_d)^2 + (z_o y_d - y_o z_d)^2 - y_d^2 x^2(t) \\ &= a y^2(t) + b y(t) + c + w x^2(t) \end{aligned}$$

其中

$$\begin{aligned} a &= x_d^2 + z_d^2, \quad b = 2[(x_o y_d - y_o x_d) x_d + (z_o y_d - y_o z_d) z_d] \\ c &= (x_o y_d - y_o x_d)^2 + (z_o y_d - y_o z_d)^2, \quad w = -y_d^2 \end{aligned}$$

则

$$f'(t) = 2ay(t)y'(t) + by'(t) + 2wx(t)x'(t)$$

那么就可以用牛顿迭代求出  $t$ 。但其实没那么简单。

由于可能有多解，某些特殊情况两个解甚至挨得特别近，那么一个初始值迭代得到的解往往不是最优解（使得  $k$  最小的解）。我的做法是  $[0, 1]$  区间内均匀取  $2n$  个初始值，然后做十几轮下山牛顿迭代法，当精度达到很高时结束，然后比较得到个最优的  $k$ 。

这个下山牛顿迭代具体是这样，在迭代  $t = t_0 - \frac{f(t_0)}{f'(t_0)}$  时，给  $f(t_0)/f'(t_0)$  前加个常数  $\lambda (0 < \lambda \leq 1)$ ，若满足  $0 \leq t \leq 1, f(t) < f(t_0)$  则终止，否则  $\lambda = \alpha\lambda$  继续下去， $\lambda$ 一开始为 1， $\alpha = 0.5$ （或者当  $t$  很靠边缘时设为 0.9）。

牛顿迭代求出  $t$  后，便能求出  $k$ 。但若  $y_d = 0$ ，此时  $k$  无法求出，但可以发现此时是和一个半径为  $x(t)$  的圆求交，那么再解个一元二次方程即可（此时  $t$  已经由上面迭代求出）：

$$\begin{aligned} (x_o + kx_d)^2 + (z_o + kz_d)^2 &= x^2(t) \\ \Rightarrow (x_d^2 + z_d^2)k^2 + 2(x_o x_d + z_o z_d)k + x_o^2 + z_d^2 - x^2(t) &= 0 \end{aligned}$$

然后是求法向量。若  $x(t) \neq 0$ ，由于

$$p(t, \theta) = (x(t) \cos \theta, y(t), x(t) \sin \theta), \quad (0 \leq \theta < 2\pi)$$

则法向量为

$$\begin{aligned} n &= \frac{\partial p(t, \theta)}{\partial t} \times \frac{\partial p(t, \theta)}{\partial \theta} \\ &= (x'(t) \cos \theta, y'(t), x'(t) \sin \theta) \times (-x(t) \sin \theta, 0, x(t) \cos \theta) \end{aligned}$$

若  $x(t) = 0$ ，法向量直接为  $(0, -y_d, 0)$ 。

具体实现见 [src/geo/collection/bezier.rs](#)。

## 网格物体



配置文件 [result\\_2.json](#)。

只实现了三角面片的网格物体。用 KD-Tree 或 BSP-Tree 来维护所有三角面片。以下仅说明 BSP-Tree（因为KD-Tree 是 BSP-Tree 的特例，划分用的是平行坐标轴的平面）。

BSP-Tree 如何选取划分的平面是一个难题，首先选所有 KD-Tree 会选的面（即平行坐标轴且左右点集到平面距离方差最大的那个），然后再考虑每个三角形面片的面，以及这个面绕着三条边旋转 45°、-45°得到的面。得到面之后再划分，跨越平面的三角面两边都会加入进去。当面片个数小于设定阈值便直接划分为叶子。

这样预处理的复杂度是平方级的，特别慢，但目前我没有找到一种好的办法（而 KD-Tree 的预处理理想情况下是  $O(n \log n)$  的，但跨越两个面的三角片特别多的话也可能退化到平方级，不过这种情况一般不会出现）。

查询的话就比较普通了，根据射线与划分平面的位置关系，确定左右孩子的先后递归顺序，然后在递归的同时根据目前最优答案再剪枝一下。实际测试下来速度还是很快的（我用的是手工栈实现的递归）。

三角面与光线求交我用的不是 Möller-Trumbore intersection algorithm 而是 [Fast Ray-Triangle Intersections by Coordinate Transformation](#)，可以通过预处理得到一些变换矩阵，之后只需要最多进行 15 次乘法、15 次加法、1 次除法（浮点数）便能求出解（也并不是直接矩阵乘法），具体见代码第 171 行以后。

BSP-Tree 和 KD-Tree 在我的测试中，查询速度差别并不大，面比较少时 BSP-Tree 会慢一些（因为查询时 BSP-Tree 要进行两次点积，而 KD-Tree 只需要一个乘法，所以深度优势便体现不出来了，更何况深度也不会比 KD-Tree 浅到哪里去）。同时由于预处理实在太慢，所以我最终渲染用的数据结构均为 KD-Tree 而非 BSP-Tree。

具体实现见 [src/geo/collection/mesh.rs](#)。

## 其他

## 随机数生成器

生成一个  $(0, 1]$  内的均匀分布伪随机数有一个经典做法

```
u32 x = seed
function rand()
    x ^= x << 13
    x ^= x >> 17
    x ^= x << 5
    return x / MAX_U32
```