

# 期末项目报告

## ——基于渐进式光子映射 (SPPM) 的图形渲染

PB21010410 高凡

PB21010362 汪兆辰

2023 年 6 月 23 日

## 1 项目介绍

### 1.1 从基本全局光子映射 (Basic Photon Mapping) 讲起

有效的实现全局光照 (Global Illumination) 的模拟是计算机图形学中的一个经典问题，在 1998 年 Veach 的博士论文中给出了无偏 (unbiased) Monte-Carlo 模拟的 Path Tracing 算法，但 PT 算法路径追踪在模拟散焦 (caustics) 时并不准确，同时由于 Monte-Carlo 采样的误差阶是  $\mathcal{O}(n^{-\frac{1}{2}})$ ，增加采样点对最终渲染结果的效果有限。

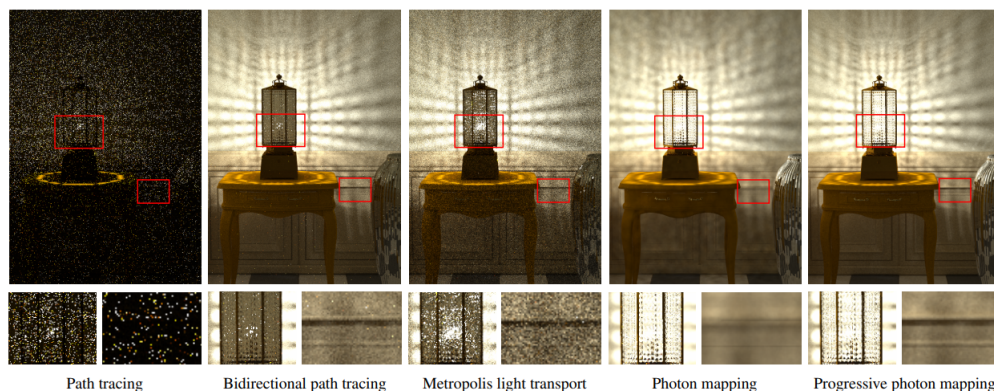


图 1: 几种光子映射算法的效果对比

对这种算法的一种改进是光子映射 (Photon Mapping), 最朴素的光子映射分为两个阶段: 第一个阶段需要构建一张光子贴图, 用来存储从光源发射出的所有光子的通量信息; 第二阶段从相机进行传统的路径追踪, 在追踪到漫反射表面时统计附近的光子信息, 并根据这些信息计算辐射率 (radiance).

### 1.1.1 光子贴图

构建光子贴图的基本思路是：让光源发出光子，并让光子在场景中反复弹射，直到被某个漫反射表面彻底吸收为止。首先需要基于光源采样以确定光子的射线与初始功率，其中射线按照光源随机采样，而初始功率根据

$$\Phi = \frac{L_e |\cos \theta|}{pdf_A(x) pdf_\omega(\omega)} \quad (1)$$

来确定，其中  $L_e$  为光源自身的辐射率。

在获得光子的初始状态后，需要对每一束光子在场景中反复迭代。当射线与表面相交时，按照表面的材质生成反射分布 BSDF，并基于该 BSDF 记录下反射的方向与类型。如果反射类型为漫反射，就将光子的位置、路径与功率记录到光子贴图中，并随机按照反射比例判断是否终止迭代，如果未被吸收，则更新光子的功率，算法如下：

首先计算当前材质的反射率：

$$R = \frac{f(x_{i-1} \rightarrow x_i \rightarrow x_{i+1}) |\cos \theta|}{pdf_\omega(x_i \rightarrow x_{i+1})} \quad (2)$$

然后得出光子的下一次功率：

$$\Phi_{i+1} = \Phi_i \frac{R}{N_{BSDF}} \quad (3)$$

其中  $N_{BSDF}$  为当前交点的反射分布下的反射模型的总数。

为了存储光子贴图，并加速光子映射，一种存储方式是按照光子图的位置构建平衡 kd 树，并用 kNN 算法进行优化，虽然建树的时间略微增加，但这种方式可以将后续查询的时间复杂度降到  $\mathcal{O}(\sqrt{N} + K)$ 。

### 1.1.2 基本全局光子映射

由渲染方程：

$$L_r(x, \omega) = \int_{\Omega} f(x, \omega', \omega) L_i(x, \omega') |\cos \theta'| d\omega' \quad (4)$$

将辐射率

$$L_i(x, \omega') = \frac{d^2 \Phi_i(x, \omega')}{dA |\cos \theta'| d\omega'} \quad (5)$$

代入渲染方程得：

$$L_r(x, \omega) = \int_{\Omega} f(x, \omega, \omega') \frac{d^2 \Phi_i(x, \omega')}{dA |\cos \theta'| d\omega'} |\cos \theta'| d\omega' \quad (6)$$

$$= \int_{\Omega} f(x, \omega, \omega') \frac{d^2 \Phi_i(x, \omega')}{dA} \quad (7)$$

将其离散化得到：

$$L_r(x, \omega) = \sum_{p=1}^N f(x, \omega_p, \omega) \frac{\Delta \Phi_p(x_p, \omega_p)}{\Delta A} \quad (8)$$

其中  $\Phi_p, \omega_p$  分别为光子的功率与方向。

利用 kd 树和上述公式，取定合适的  $N$  就可以得到  $L_r$ ，样本  $N$  数量越多，估算的结果越接近真实值。而由 kd 树的 kNN 算法可以得到样本所占的面积  $\Delta A$ ，其中半径为

$$r = \max(\text{Dist}(x, x_1), \text{Dist}(x, x_2), \dots, \text{Dist}(x, x_N)) \quad (9)$$

### 1.1.3 全局光子映射的局限性

作为最原始的光子映射方案，全局光子映射保留了相当明显的光子映射特征：由于样本的随机性与密度估计的插值效应，在样本较少的情况下，光子映射会出现明显的块状斑点 (如图 1-4)。另一方面，当场景中的大多数表面为光滑材质时，第一阶段中生成光子贴图需要多次迭代才能使得光子最终被吸收，而单纯为迭代次数设置上限会使得图像整体偏暗，需要更好的改进办法。

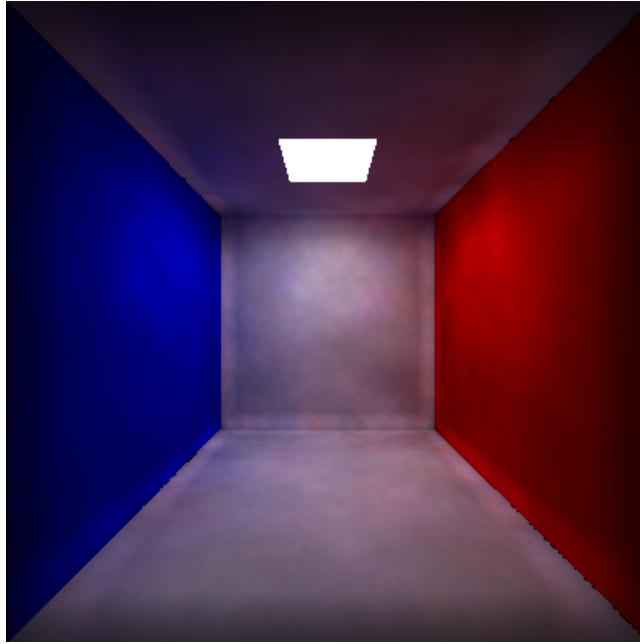


图 2: 全局光子映射渲染效果

其次，如图 2 所示，全局光子映射在场景的墙角处有明显的渗色情况，出现这种情况的原因是在密度估计过程中样本数量取的过大，使得部分与被估计光子不在同一平面的光子也参与了估计。在不占用大量内存与渲染时间的前提下很难有效果较好的优化，所以需要算法进一步改进。

## 1.2 随机渐进式光子映射 (SPPM)

SPPM 在 PM 的基础上调整了计算的模型，使得计算的结果可以近似完全收敛。首先要从 PM 过渡到渐进式光子映射 (PPM)，将 PM 中单次的光子映射改为多次光子映射并叠加，实现渐进式的收敛；之后再进一步将射线追踪阶段也拆为多次，并将 PPM 中基于场景的计算方式改为基于像素计算，以获得更低的开销与更快的收敛速度。

### 1.2.1 从 PM 到 PPM

在 [7] 中, Jensen 提出了一种改进措施, 将 PM 的光子映射拆成多次, 形成多张光子图并分批执行密度估计, 对所有结果取平均值, 但这最多将估计范围收敛到对应的 kNN 范围  $r$  内, 而如果一开始将  $r$  取的很小, 会使得估计范围内光子密度很低, 降低估计效率, 所以实现对估计半径  $r$  的控制成为了重点的问题。

改进方案的前几步与 Jensen 2004 相同: 假设第一次密度估计需要  $N_1(x)$  个光子, 在第一轮密度估计后, 利用 kNN 算法可以得到估计范围的半径  $R_1(x)$ , 并得到第一次估计的密度  $d_1(x) = \frac{N_1(x)}{\pi R_1(x)^2}$ . 之后维持半径  $R_1(x)$  不变, 重新生成一张光子图, 并围绕新的光子图基于同样的半径  $R_1(x)$  做一次密度估计, 假设新的光子数目为  $M_1(x)$ , 则新的密度估计为

$$d_2(x) = \frac{N_1(x) + M_1(x)}{\pi R_1(x)^2} \quad (10)$$

在进行下一步密度估计之前, 先对估计半径进行收缩, 假设在收缩过程中密度是不变的, 可以计算出半径缩减后下一轮光子个数  $N_2(x)$  与密度  $d_2(x)$  的关系:

$$d_2(x) = \frac{N_2(x)}{\pi R_2(x)^2} = \frac{N_2(x)}{\pi(R_1(x) - \Delta R_1(x))^2} \quad (11)$$

其中  $\Delta R_1(x)$  为第一次迭代过渡到第二次过程中半径的缩减量. 而下一轮光子数量  $N_2(x)$  取决于上一轮光子  $N_1(x)$  与新增光子  $M_1(x)$ , 这里  $N_1(x)$  无法更改, 但可以控制新增部分的比例, 即

$$N_2(x) = N_1(x) + \alpha M_1(x) \quad (12)$$

其中缩减系数  $\alpha$  实际上控制了半径  $R_2(x)$  的衰减速度. 结合 (10)-(12) 三式, 可得:

$$R_2(x) = R_1(x) \sqrt{\frac{N_1(x) + \alpha M_1(x)}{N_1(x) + M_1(x)}} \quad (13)$$

重复此过程, 可以使得估算半径不断收敛。

记  $x$  处出射到相机方向  $\omega$  的总通量为:

$$\tau(x, \omega) := \sum_{p=1}^N f(x, \omega_p, \omega) \Delta \Phi_p(x, \omega_p) \quad (14)$$

假设已经迭代了  $i$  轮, 累积了  $N_i(x)$  个光子, 那么这些光子的总通量理论上可以计算为:

$$\tau_i(x, \omega) = \sum_{p=1}^{N_i(x)} f(x, \omega_p, \omega) \Delta \Phi_p(x, \omega_p) \quad (15)$$

然后下一步, 缩减前新增  $M_i(x)$  个光子, 尽管由于半径缩减最终只保留其中的一部分, 但依然需要计算全部新增光子的总通量, 记为:

$$\phi_i(x, \omega) = \sum_{p=1}^{M_i(x)} f(x, \omega_p, \omega) \Delta \Phi_p(x, \omega_p) \quad (16)$$

由公式 (13) 可以计算出收敛后半径:

$$R_{i+1}(x) = R_i(x) \sqrt{\frac{N_i(x) + \alpha M_i(x)}{N_i(x) + M_i(x)}} \quad (17)$$

并估计出缩减后保留的总通量:

$$\tau_{i+1}(x, \omega) = (\tau_i(x, \omega) + \phi_i(x, \omega)) \frac{N_i(x) + \alpha M_i(x)}{N_i(x) + M_i(x)} \quad (18)$$

结合式 (8) 可以计算出第  $i+1$  轮的辐射率:

$$L_r(x, \omega) = \frac{1}{\Delta A} \sum_{p=1}^{N_{i+1}(x)} f(x, \omega_p, \omega) \Delta \Phi_p(x, \omega_p) \quad (19)$$

$$= \frac{1}{\pi R_{i+1}(x)^2} \frac{\tau_{i+1}(x, \omega)}{N_e(i+1)} \quad (20)$$

随着迭代次数不断增大, 总通量  $\tau_{i+1}(x, \omega)$  增大, 密度估计半径  $R_{i+1}(x)$  缩小, 全局光子总数  $N_e(i+1)$  越来越多, 能够完全收敛。

但由于 PPM 需要为每个像素点分配很多射线与漫反射面相交形成的命中点才能有效去噪, 仍然需要较大的内存开销。

### 1.2.2 从 PPM 到 SPPM

SPPM 在 PPM 的基础上改进了对命中点的生成, 在每个渲染阶段, 先进行一次射线追踪, 再对每次射线追踪生成一小批命中点, 并围绕这些命中点在一个像素范围内抖动更新, 从而当样本量足够大时, 估算的即为像素内的全部区域。而 SPPM 的计算公式与 PPM 的式 (20) 只需将命中点  $x$  改为像素区域  $S$ :

$$L_r(S, \omega) = \frac{1}{\pi R_{i+1}(S)^2} \frac{\tau_{i+1}(S, \omega)}{N_e(i+1)} \quad (21)$$

## 2 代码介绍

项目代码由以下部分组成:

- aabb.hpp
- bvh.hpp
- camera.hpp
- photon.hpp
- photon\_map.hpp

- ray.hpp
- renderable.hpp
- SppmManager.hpp
- SPPM.cpp
- 外部 Eigen 与 stb 库

## 2.1 aabb.hpp

该头文件定义了 Axis Aligned Bounding Box 对象，其中成员包括：

- v6f limits: 六个面的限定参数
- bool is\_ray\_inter(Ray& ray): 测试是否与射线相交
- AABB union\_with\_other(AABB& another): 融合两个 AABB
- v3f get\_center(): 获取 AABB 的中心

## 2.2 bvh.hpp

该头文件定义了 BVH 树对象以及相关的类型，包括：

- struct BVHNd: BVH 树节点结构体
  - BVHNd\* lc, \* rc; 左右孩子
  - AABB aabb; 节点对应的 AABB
  - Renderable \*rd\_obj; 包围盒中的物体。该指针仅有在节点为叶子时才非空
- struct IntersectInfo: 相交信息记录
  - Renderable\* obj; 相交的物体
  - v3f pos; 相交位置
  - float distance; 射线发出点到相交点的位置
  - bool do\_intersect; 是否真的相交的标志位。如果为 false 说明该射线事实上与场景并没有交点
- class BVH
  - BVHNd\* root; 根节点

- `std::vector<Renderable*> obj_ptr_list`; 可渲染物体指针的列表。该列表是参数的复制，因为需要对列表进行排序。
- `std::vector<BVHNd*> tree_nodes`; 所有的树节点
- `IntersectInfo get_first_intersection(Ray& ray, Renderable* obj)` 获取射线的交点，同时排除当前物体 `obj`

## 2.3 camera.hpp

该头文件定义了相机对象

- class Cam
  - `v3f top, lookat, pos`; 相机几何参数
  - `int w, h`; 图像参数
  - `float focus, fov`; 焦点与 `fovY`
  - `float ccd_w, ccd_h`; 相机 `ccd` 参数
  - `v3f local_x_ax`; 相机坐标系未显式给出的轴
  - `Ray sample_ray_xy(int x, int y, float fltx, float flty)` 计算某一个像素点采样射线

## 2.4 photon.hpp

该头文件定义了光子

- class Photon
  - `v3f power_rgb`; 光子携带的能量
  - `Ray ray`; 光子对应的传播射线
  - `v3f end_at`; 光子击中位置
  - `bool bounce`; 是否击中某个目标

## 2.5 photon\_kdt.hpp

该头文件定义了光子对象的 `kdtree`

- struct KDTNd kdt 节点
  - `Photon *ph` 对应的光子
  - `int axis`; 分割轴

- KDTNd\* lc, \* rc; 左右孩子
- class KDT
  - KDTNd \*root; 根节点
  - std::vector<KDTNd\*> ptrs; 节点指针
  - KDTNd\* recursive\_construction(int st, int ed, std::vector<Photon>& photons, int axis) 构建 kdt
  - void recursive\_kdt\_search(pq\_kdt& pq, int N, KDTNd\* nd, v3f& pos) 搜索给定数目的节点
  - std::vector<Photon\*> N\_near(int N, v3f pos) 搜索定额
  - std::vector<Photon\*> N\_near\_with\_R(int N, v3f pos, float R) 搜索定额以及半径以内

## 2.6 photon\_map.hpp

该头文件定义了光子图的生成

- class PhotonMap
  - std::vector<Photon> photon\_record; 光子记录
  - void get\_map\_mt(int n\_th, int n, Scene& sc, int single\_max\_iter = 32, float rr\_prob = 0.8) 多线程生成光子，计算传播，记录光子

## 2.7 ray.hpp

该头文件定义了射线

- class Ray
  - v3f pos; 起点
  - v3f dir; 方向
  - v3f get\_t(float t) 获取在距离 t 处的位置

## 2.8 renderable.hpp

该头文件定义了所有可渲染物体的基类 Renderable 以及对应的受支持模型类型——三角网格 & 球体的派生类 Triangle&Ball

- class Renderable



- int n\_materials; 材质的数目
  - bool has\_spec;
  - bool has\_lamb;
  - bool has\_tran; 是否具有支持的三种材质：镜面反射、理想漫反射、透射
  - bool is\_light; 是否是光源
  - float spec\_ratio;
  - float lamb\_ratio;
  - float tran\_ratio; 采样 BSDF 时的概率
  - v3f color; 颜色参数
  - float emit\_power; 发光强度
  - float n\_t; 折射率
  - virtual InteractType sample\_reflect\_type() 采样 BSDF
  - virtual Ray get\_next\_ray(InteractType type, Ray& in, v3f pos) 根据反射类型以及入射信息计算出射射线
  - virtual v3f get\_bsdf(InteractType type, v3f in\_dir, v3f out\_dir, v3f pos) 获取对应 BSDF
  - virtual std::pair<float, v3f> try\_intersect\_ray(Ray& in) 获取交点以及距离
  - virtual Photon emit\_a\_photon() 发射光子，仅当是光源时才使用
  - virtual v3f get\_norm(v3f pos) 获取曲面某点的法向
  - virtual v3f sample\_a\_pos() 随机采样曲面上一点
  - virtual float A() 面积
  - std::vector<std::pair<InteractType, float>> available\_mat; 可使用的材质
- class Triangle: public Renderable
    - v3f v0, v1, v2; 顶点
    - v3f norm; 预先计算的法向
  - class Ball :public Renderable
    - v3f center; 球心
    - float radius; 半径

## 2.9 scene.hpp

该头文件定义了场景对象

- class Scene
  - std::vector<Renderable\* > objs; 可渲染物体
  - std::vector<Renderable\* > light\_src; 光源
  - BVH \*bvh; BVH 树
  - void append\_obj(Renderable\* obj) 添加物体
  - void construct\_bvh() 建立 BVH
  - static std::vector<Renderable\*> parse\_triangle\_file(std::string filename) 解析三角网格文件

## 2.10 SppmManager.hpp

该头文件定义了所有与随机渐进光子映射有关的过程

- constexpr float dec\_fac = 0.75f; 半径衰减系数
- struct Q 采样路径的一些记录
  - v3f pos, dir; 采样的几何信息
  - Renderable\* obj; 击中物体
  - v3f beta; 吞吐量
  - bool islt; 是否击中光源，如果是，则光源的 radiance 额外计算
- struct SppmPix SPPM 像素
  - v3f direct\_radiance; 直接光照 radiance
  - int direct\_radiance\_cnt; 光照计数
  - bool is\_first\_estimate; 是否第一次估计
  - float global\_shared\_radius 估计半径
  - v3f global\_cumulate\_pow; 积累的通量
  - int global\_N; 累计光子数目
  - int needed\_N; 下次搜索时的搜索上界，目的是优化搜索速度
  - std::vector<Q> sample\_position\_dir\_obj\_beta; 预采样信息

- class SppmManager SPPM 的主要处理逻辑
  - Cam camera; 相机
  - std::vector<SppmPix> sppm\_pix\_list; 像素
  - int single\_sp; 单点采样上限
  - inline int get\_pix\_id(int x, int y) return y \* camera.w + x; 获取像素点位置
  - void initialize\_samples\_mt(Scene & sc, int th = 8, int single\_samples = 32, int max\_ray\_reflect = 16) 多线程预采样
  - void single\_run(Scene& sc, int th = 8, const int global\_scale = 50000, const int global\_map\_iter = 32, const int global\_search\_N = 100, const float global\_search\_R = 0.05, const int all\_sp = 32) 多线程单次迭代
  - v3b get\_value(float scale, int x, int y) 获取最后的像素值

## 2.11 SPPM.cpp 程序入口

1. 加载渲染物体到场景
2. 构建 bvh
3. 定义相机
4. 创建 SppmManager
5. 预采样
6. 根据迭代次数进行计算
7. 根据给定比例计算输出图像并储存

## 3 效果展示

渲染的图像效果如下：

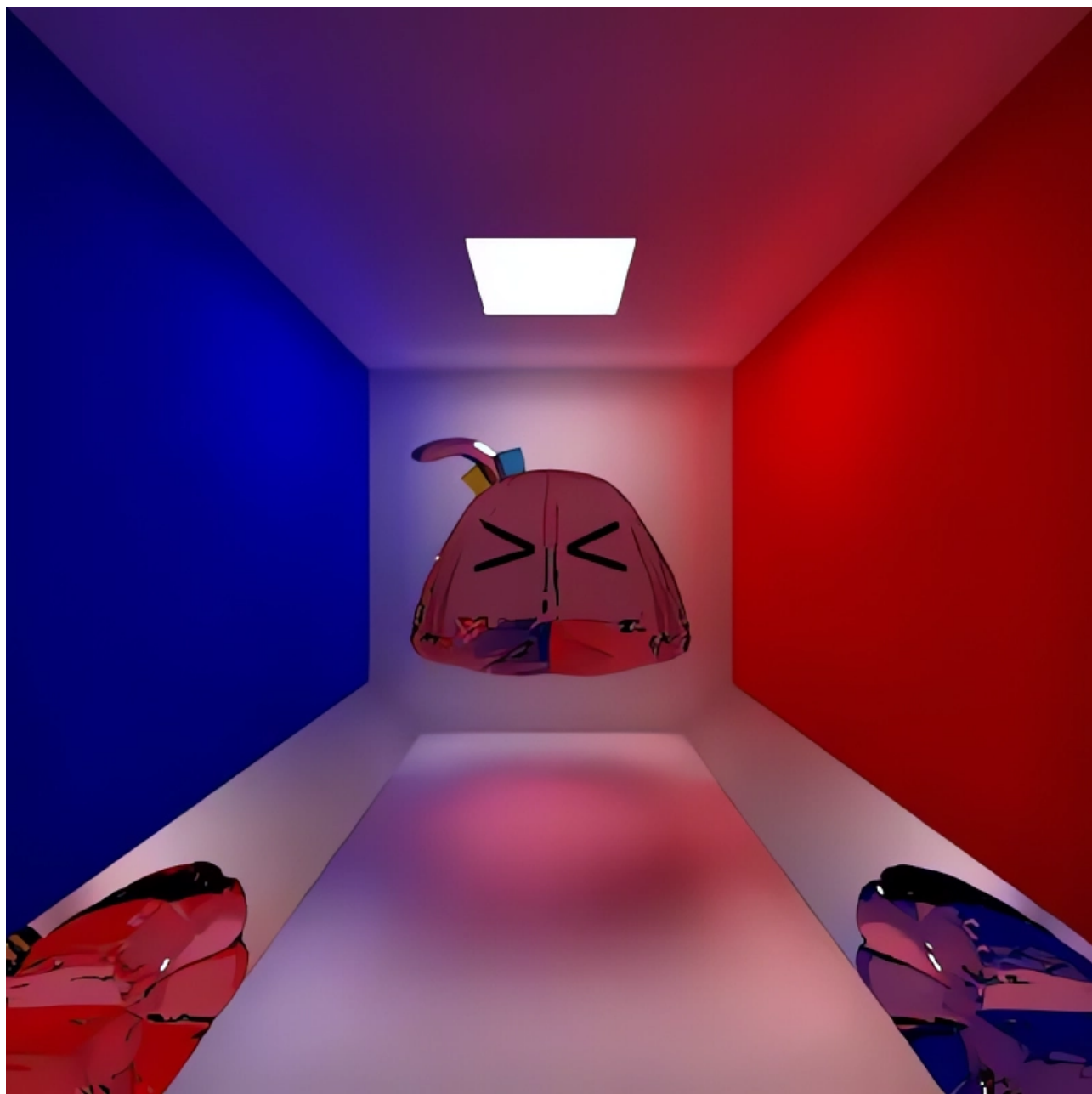


图 3: bocchi pudding

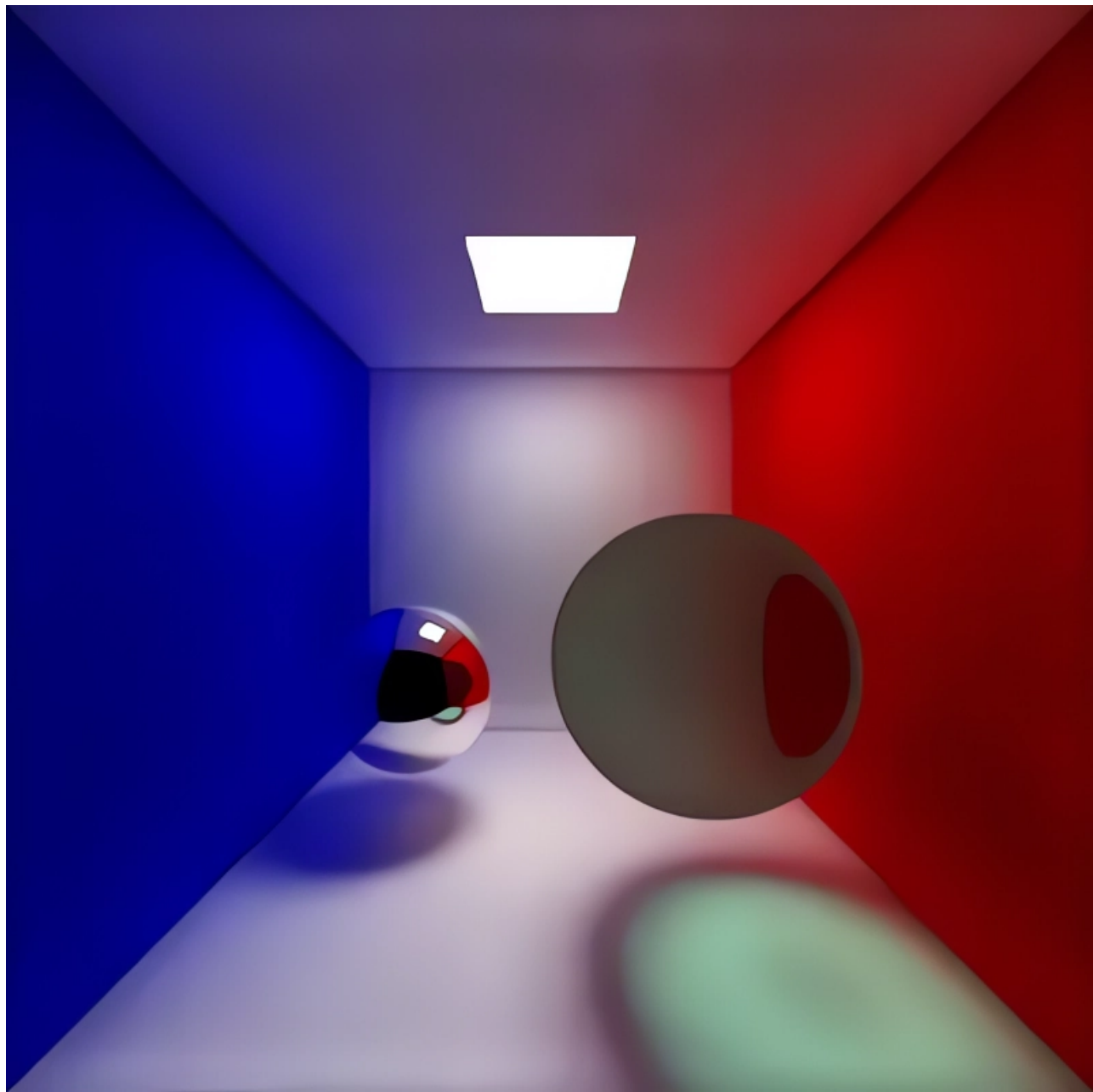


图 4: greenball

## 致谢

- 感谢陈老师与两位助教一学期的悉心教导
- 感谢知乎答主 @ 我好饱啊的光子映射总结. Talk is not cheap. Talk can be powerful.
- 感谢临近期末中区自习室晚上不断电可以跑一晚上的图
- 感谢《孤独摇滚》及其全体制作人员，有你们业界完不了
- 感谢三井律郎在「忘れてやらない」编曲的巨大升级，让我练不下去来写报告
- 感谢肥西路，南边第一家热干面和凉皮做的真不错
- 首先我是一个无神论者，然后感谢所有保佑代码不出 bug 的神灵，我愿称之为代码仙人

## 参考文献

- [1] <https://zhuanlan.zhihu.com/p/208356944>
- [2] <https://zhuanlan.zhihu.com/p/259565623>
- [3] <https://graphics.stanford.edu/courses/cs348b-00>
- [4] M.Pharr, W.Jakob, and G.Humphreys. Physically Based Rendering: From Theory To Implementation. 2018.
- [5] T.Hachisuka, S.Ogaki, and H.W.Jensen. Progressive Photon Mapping. ACM SIGGRAPH Asia 2008 Papers, 2008.
- [6] T.Hachisuka, H.W.Jensen. Stochastic progressive photon mapping. ACM Transactions on Graphics, 2009.
- [7] H.W.Jensen. ACM SIGGRAPH 2004 Course Notes. A practical guide to global illumination using ray tracing and photon mapping, 2004.
- [8] E.Veach. Stanford University. Robust monte carlo methods for light transport simulation, 1998.