# 真实感图像渲染
# 实验报告

计 97 郭昊 2019013292

2021 年 6 月 23 日

## 1 渲染结果

我设计了一个场景。这个场景采用 Progressive Photon Mapping 方法渲染，图片尺寸为 3840×2160，递归最大层数为 8。场景中要素包括：墙壁（纹理贴图、漫反射）、桌椅书架（复杂网格、纹理贴图）、兔子（简单网格）、桌上的球（漫反射、镜面反射）、椅子后面的球（折射）、书架上的酒杯（B 样条曲线迭代求交、Caustics、**B 样条的折射**）、相关渲染方法（抗锯齿、景深）。

图 1是我采用桌面 CPU：Core i5 9600KF 默频在 4 小时内渲染出的。其中光子数为 $4 \times 10^6$，图 2和图 3则是采用 Cascade Lake@3.0GHz 32 核 64 线程处理器分别在 4 小时内渲染出的，这两张图的总光子数均为 $10^7$，并且为了实现抗锯齿和景深效果，均运行了 100 次的视点记录并取平均值。图 4 则是采用 Cascade Lake@3.0GHz 32 核 64 线程处理器使用 10 小时渲染出，由于视点记录次数较多，花费的时间对应较多。

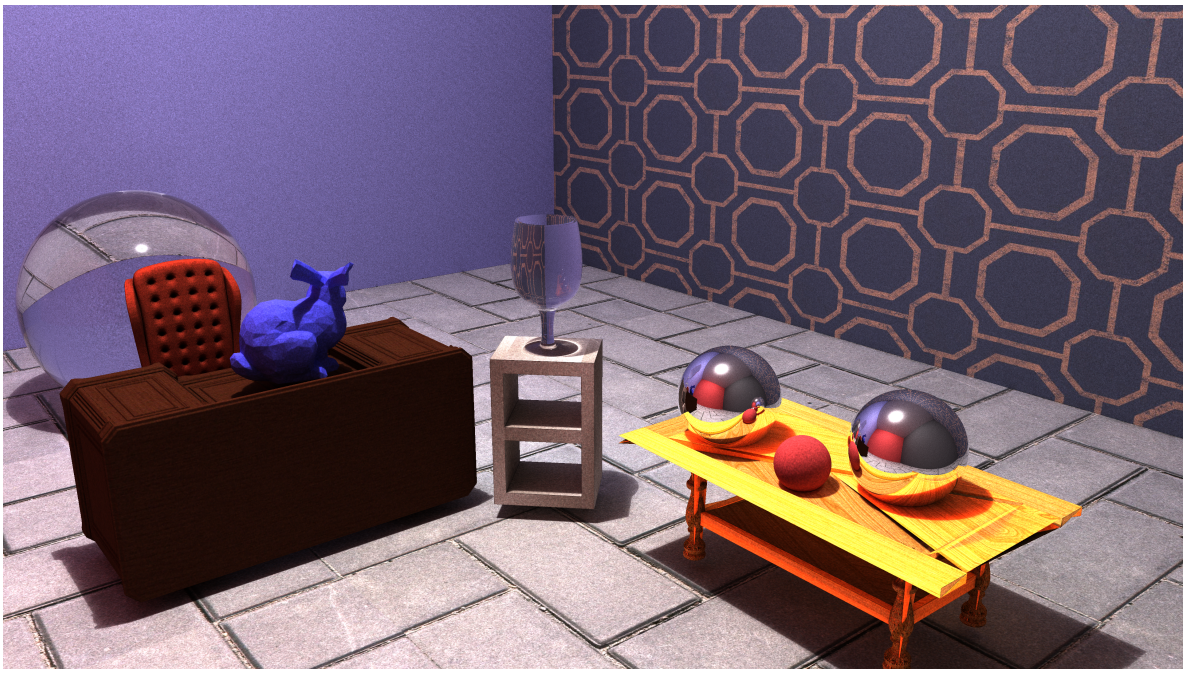

图 1: 光子数为 $4 \times 10^6$

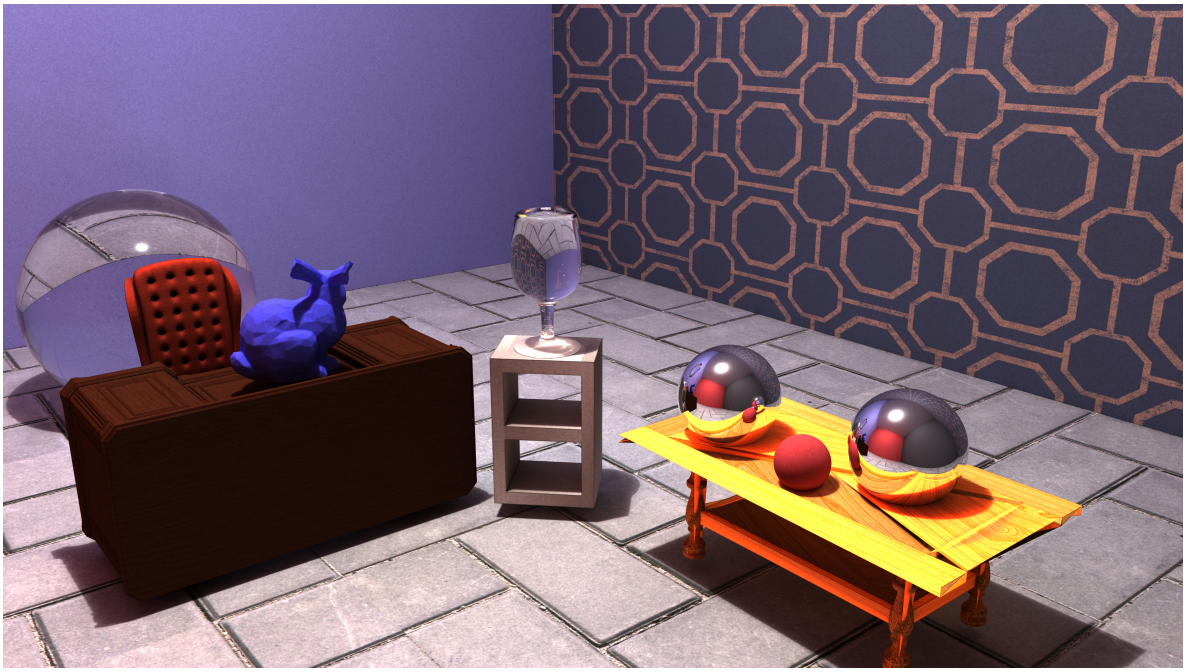

图 2: 光子数为 $10^7$，抗锯齿

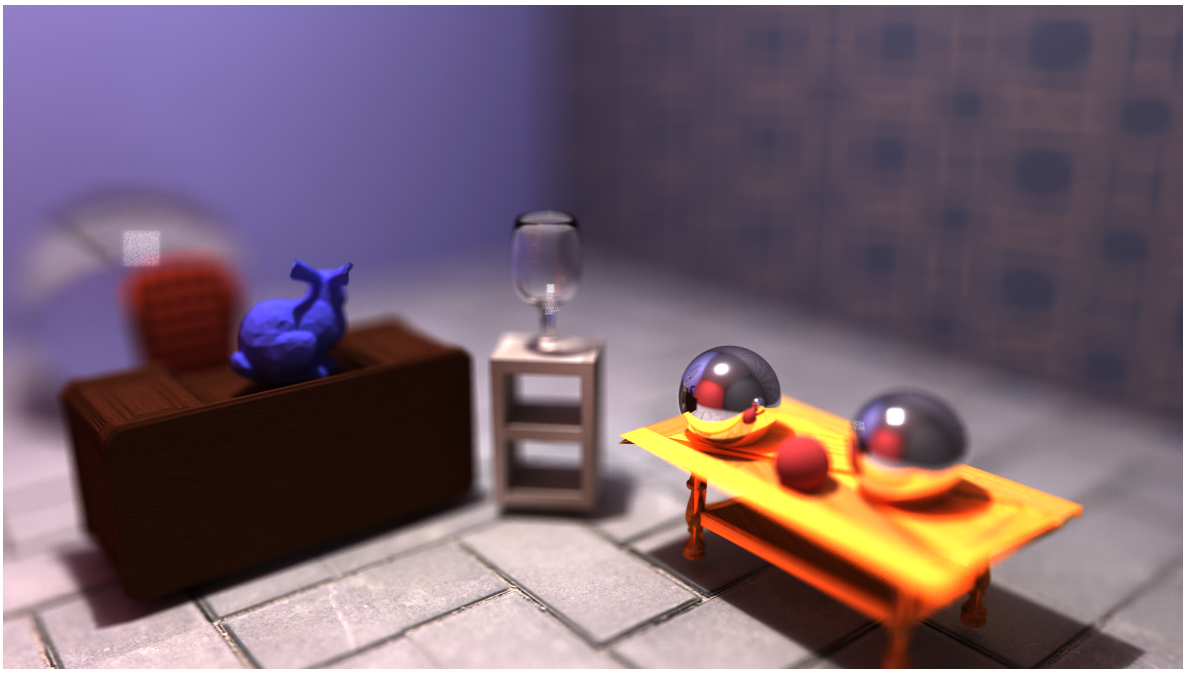

图 3: 光子数为 $10^7$，抗锯齿，带景深

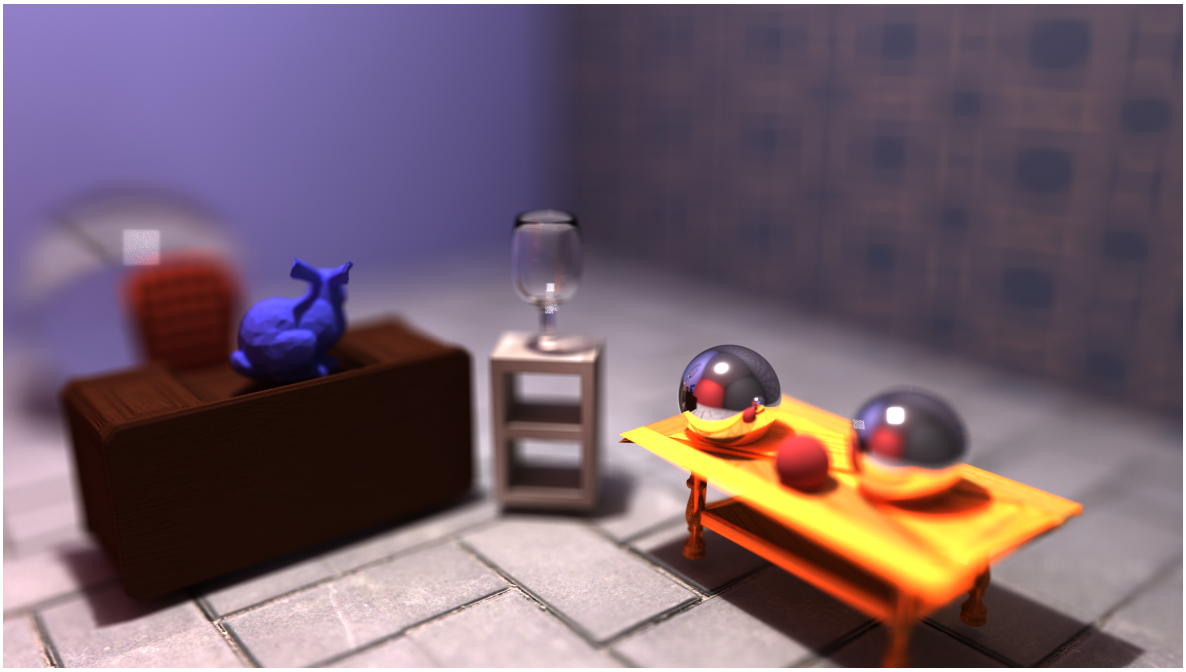

图 4: 光子数为 $10^7$，抗锯齿，带景深，500 次 HitpointPass

# 2 代码整体框架

整体代码框架继承了 PA1 及 PA3，并在此基础上作了尽可能小的改动，在保证功能完整实现的同时保留了很强的复用性及面向对象的设计思想。

## 2.1 渲染核心代码

Path Tracing 算法核心代码位于 `pt.hpp` 及 `pt.cpp` 中，而 Progressive Photon Mapping 算法核心代码位于 `ppm.hpp` 及 `ppm.cpp` 中。

## 2.2 真实感物体类

同 PA1 类似，所有物体类均继承自 `Object3D` 类，包括平面、三角形、椭圆，`Mesh`，以及 `curve,revsurface`。

## 2.3 其他添加或修改的部分

为了组织复杂三角网格，我使用了 `K-D Tree`，其代码位于 `kdtree.hpp` 及 `kdtree.cpp`。

`tiny_obj_loader.hpp` 则源自 tinyobjloader 项目[1]，我对其作了符合 PA1 框架数据结构的移植。

`utils.hpp` 储存了一些常量以及常用的函数。

并且，我还修改了 `scene_parser.cpp`，使之适应符合 Path Tracing 及 Photon Mapping 的数据储存方式。解除了对于 Light 类的依赖，转而使用 Sphere 作为光源。

在 `image.cpp` 中，我移植了 `LoadBMP` 函数[2]，以填补原本 `Image` 类不存在加载 BMP 文件函数的空缺。

# 3 光线与 B 样条旋转曲面求交及加速

由于 B 样条曲线相较 Bezier 曲线具有更加良好的性质，本代码主要对于 B 样条旋转曲面求交进行了优化，并且优化后代码相比平凡实现具有极为明显的速度提升，打破了程序的性能瓶颈，使程序能够以极高的效率完成 B 样条的折射计算。（本程序也支持 Bezier 旋转曲面的求交）

[1] https://github.com/tinyobjloader/tinyobjloader

[2] https://github.com/vallentin/LoadBMP/blob/master/loadbmp.h

### 3.1 核心思想

在 $y_d \neq 0$ 时，我们有

$$x(t)\cos\theta = x_o + \frac{(y(t)-y_o)x_d}{y_d}$$
$$x(t)\sin\theta = z_o + \frac{(y(t)-y_o)z_d}{y_d}$$

整理得

$$\left(\frac{(y(t)-y_o)x_d}{y_d} + x_o\right)^2 + \left(\frac{(y(t)-y_o)z_d}{y_d} + z_o\right)^2 - x(t)^2 = 0$$

其中 $x(t), y(t)$ 为样条曲线上 $t$ 处的坐标值。在 3 阶 B 样条曲线中，该方程为一 6 次多项式方程，可使用牛顿迭代法求解 $t$ 的值。解出 $t$ 值后，回代即可反解出 $\theta$ 及 $t_r$。

### 3.2 牛顿迭代

代码位于 `revsurface.cpp` 中第 30-129 行（包含了 $dy = 0$ 的特殊情况及一般情况）

传统的牛顿迭代法对初值要求极高，在本次作业中，经常出现无法收敛的情况。因此，我采用类似“牛顿下山法”的思想，对于每次迭代，将要下降的值除以一个系数，即将方程变为

$$t_{i+1} = t_i - \frac{f(t_i)}{x \times f'(t_i)}$$

通过这种方式平缓牛顿迭代的过程，可以防止出现局部梯度过小导致迭代时迅速超出边界的情况。

本次作业，我采用 32 作为 $x$ 的值，并且每次均匀在 B 样条定义域中取 32 个点作为初值进行迭代，这样加强了解的准确性。

### 3.3 针对 3 次 B 样条求值的 AVX 指令集优化

代码位于 `curve.hpp` 中 153-274 行。以下讨论中提到的 B 样条均指 3 次 B 样条。

在 de Boor 算法的朴素动态规划实现中，3 次 B 样条需要循环 3 轮，且每轮均需从 0 循环到 $n+k+1-p$（其中 $p$ 为轮次数），时间复杂度极高，且不易被编译器优化。经 Visual Studio 性能分析，如果 B 样条曲线存在折射，该求值过程会占据整个程序运行时间的 70% 以上。

事实上，由于 B 样条基函数中 $B_{i,0}(t)$ 仅在 $t_i \leq t < t_{i+1}$ 的 $t_i$ 处为 1，其余位置均为 0，并且每个位置的 $B_{i,p}$ 仅由前一个对应位置 $B_{i-1,p}$ 及其后一位置 $B_{i-1,p+1}$ 决定，因此，基函数仅在 $[B_{i-p,p}, B_{i,p}]$ $(0 \leq p \leq 3)$ 不为 0。通过仅对这些位置进行迭代，程序运行的效率有了显著的提高，但该求值过程仍然会占据整个程序运行时间的约 50%。

尽管速度已经得到了显著的提升，但我没有止步，而是尝试寻找 CPU 上更加进阶的优化方式，即指令集优化。注意到：基函数不为 0 的点至多有 4 处，而 AVX 指令集的寄存器大小为 256

字节，刚好可以装下 4 个双浮点数。并且，每个位置相同的加法及乘法运算，恰好适合使用 `simd` 指令集进行批量计算。

因此，我在 `curve.hpp` 第 194-271 行实现了 AVX 版本的 3 阶 B 样条曲线求值函数，并且该函数会在已定义 `__AVX2__` 宏时自动替换上述优化后的传统实现[3]。经测试，在 Core i5 9600KF 处理器上渲染相同图片（B 样条约占图片的 1/50，折射材质），渲染速度相比优化后的传统实现，有接近 2 倍的提升；相比朴素动态规划实现，速度提升则超过 10 倍。

该优化的优势在于：充分利用了 CPU 的指令集优势，并且能够无障碍迁移到任何支持 AVX 的 CPU 平台（或其他 B 样条曲线代码中），无需如 Cuda 般考虑数据结构方面的设计（去除 OOP 元素，采用类 C 方式存储数据，在得到速度提升的同时代码可扩展性及风格等必然受到影响）。

# 4 渲染核心

本次作业中，我实现了路径追踪算法[4]及渐进式光子映射算法[5]，并且使其适合于 PA1 及 PA3 的框架。

## 4.1 Path Tracing

代码位于 `pt.cpp`。

对于每个像素，从视点发射一条光线，如果碰到光源则停止，反之，碰到的表面有三种类型：漫反射、镜面反射及折射。对于漫反射，则随机选择一个反射方向继续追踪；镜面反射则按照理想方式，反射角等于入射角；折射则视递归深度，追踪反射方向和透射方向，或二选一进行追踪。

长时间渲染后，该算法效果较好，但收敛相对较慢。

## 4.2 Progressive Photon Mapping

代码位于 `ppm.cpp`。

首先进行视点的追踪，在给定递归深度的情况下，对于每个像素，从视点发射一条光线，碰到三种类型的表面时，与路径追踪类似地继续追踪，但折射光线无视深度，两个方向均进行追踪。追踪到的漫反射点会对当前像素颜色有影响，将其加入列表中。

接着，利用列表创建一个哈希表，维护三维空间中距离每个坐标为 $r$ 的视点。

然后，随机选择光源发射一条随机方向的光线，当其与漫反射表面相交时，查询所有与交点距离小于其受光影响的半径 $r$ 的视点，增加这些点受到光的强度，并降低该点受到光影响的半径；而对于反射和折射表面，则选择一个方向继续进行追踪。这种方法收敛效率极高，效果也比较好。

## 4.3 光线与各类物体求交

代码分散在各物体名的 `hpp` 及 `cpp` 中。

[3]使用该版本求值函数时程序会输出"Using Intel AVX to accelerate BSpline curve..."

[4]参考 smallpt:https://www.kevinbeason.com/smallpt/，进行了面向对象的重写

[5]参考 smallppm: https://github.com/snowroll/smallppm，进行了面向对象的重写

除样条曲线外，与单个物体求交同 PA1 及 PA3，并加入了包围盒求交（见 5.1）这里不再赘述。

对于 `Group` 类及 `Mesh` 类，则采用 `K-D Tree` 维护所有数据（见 5.2），通过包围盒（见 5.1）实现了求交的粗粒度快速判断，以达到加速目的。

# 5 渲染加速

## 5.1 包围盒

代码位于 `aabb.cpp` 中。

为了加速复杂网格及 B 样条曲面的求交，我采用包围盒进行加速。对于每一个三角形，其包围盒为三个顶点的坐标的最小值到坐标最大值的 AABB；对于 B 样条旋转曲面，则为

$(-\max(x_{max}, -x_{min}), y_{min}, -\max(x_{max}, -x_{min}))$ 到

$(\max(x_{max}, -x_{min}), y_{max}, \max(x_{max}, -x_{min}))$。

对于其他类型，我也做了包围盒（进行减速），但效果并不明显。

在与包围盒求交时，还有一个小 trick：正常情况下需要对光线的方向坐标作除法。但是除法效率较乘法而言极低，因此在储存光线时，先将其方向的倒数计算好存起来，这样只需作乘法即可。

## 5.2 K-D Tree

代码位于 `kdtree.cpp` 中。

本次作业中，`K-D Tree` 主要用于复杂网格的求交加速。每个 `KDTNode` 类继承于 `Object3D`，具有自己的包围盒，其值为左右两孩子包围盒坐标的最小值及最大值。通过重载其 `intersect` 函数为先判断是否与包围盒有交，若有交，则返回两个孩子求交的或，实现了求交的多态以及代码的简洁。

`K-D Tree` 对于 `Mesh` 类求交加速极为显著，在测试超过 $10^5$ 个三角网格时，渲染速度相较无网格渲染仅减慢至约 $\frac{1}{3}$。

## 5.3 CPU 级优化

对于样条曲线的指令集优化在 3.3部分已详细表述，这里不再赘述。

并且，我采用 `OpenMP` 库进行了算法核心的多线程加速（虽然没有分数，但渲染更快还是好的）。

我也尝试了采用 AVX 指令集重写 Vecmath 库，但反而收到了负效果，因此放弃。

# 6 特殊效果

## 6.1 景深

代码位于 `camera.hpp`。

为了模拟实际相机，采用 `focal` 和 `aperture` 两个参量。`focal` 表示相机焦平面位置，而 `aperture` 表示光圈大小，理想情况下光圈大小为 0。在产生光线时，实际上是在相机中心周围半径为 `aperture` 的圆上随机选点作为光路起点，该点与焦平面上一点的连线即为产生光线的方向。这样做，位于焦平面上的物体显示正常，但其他位置的物体，则会因光线的偏移，在取平均值后变得模糊。实际调整效果时，应将需要“对焦”的物体置于焦平面上。

在 PPM 中，由于 HitpointPass 只有一次，所以会出现很多的噪声，因此，我将 HitpointPass 改为多次，每次对应的 PhotonPass 光子数减少，将这些“单次 PPM”得到的结果取平均，即可以得到更好的景深效果，景深效果见图 3及 4。其中左边球上的白色方块显得较为“异常”，这可能是由最右侧球反射过玻璃杯，经过玻璃杯折射若干次后得到的。由于 HitpointPass 次数仍然较少（100 次和 500 次），且光路较为复杂，故噪声较多，这也是 PPM 中实现景深的一个小缺陷。

## 6.2 抗锯齿

代码位于 `ppm.cpp` 第 154-162 行。

我通过对附近点采样的方法实现了抗锯齿的功能。在 PT 中，与 smallpt 相同，采用对采样点周围 $2 \times 2$ 的点分别作 tent filter[6]的方法进行采样，得到的结果取平均值；在 PPM 中，则是提高 HitpointPass 的轮数，并在每次 HitpointPass 时，对于对应像素点作一次 tent filter 的偏移，对最终每轮 PPM 的结果取平均，即可实现抗锯齿。抗锯齿后的效果见图 2。

## 6.3 纹理

代码位于 `texture.hpp`，并对 `material.hpp` 进行修改，改变获取颜色的逻辑，并同时对 `mesh.cpp`，`revsurface.cpp`，`plane.cpp` 设置 `hit` 的部分进行了修改，并且对 `hit.hpp` 进行修改，以储存纹理坐标。我分别实现了平面、B 样条曲面及复杂网格上的纹理。

对于 B 样条曲面，其 $t$ 和 $\theta$ 两个参数可以与 $[0,1]$ 区间上的纹理坐标 $u$ 和 $v$ 完美对应，对于复杂网格，tinyobjloader 提供了读取到每个顶点对应的纹理坐标的方法，利用三角形上重心坐标的线性组合，即可得三角形上任意一点的纹理坐标。而对于平面，我采用的方法是将平面投影到坐标轴对应平面，取某两个坐标作为纹理坐标。但这样操作后，常常会出现“场景中纹理重复次数过多”的情况，因此我给这两个坐标分别除了一个系数，使平面上 $5 \times 5$ 代表一轮纹理坐标。通过对于纹理图片的精心选择，使之可以拼接，即可得到很好的效果。

[6] http://www.kevinbeason.com/smallpt/smallpt_cline_slides.ppt