

计算机图形学：三维造型与渲染

计算机科学与技术 75 班 赵成钢 2017011362

2019 年 6 月

1 代码实现

代码采用 C++ 实现，共 1687 行，Github 地址是 <https://github.com/LyricZhao/BoArtist>。

1.1 代码框架

1. main.cpp: 程序的主入口。
2. renderer.cpp/h: 渲染器，包含了主要渲染算法的实现。
3. sppm.cpp/h: SPPM 加速中的 KD-Tree 数据结构等的实现。
4. stb_image.cpp: 调用 STB 库实现贴图文件的读取。
5. utils.h: 一些辅助性的小函数。
6. Makefile: 方便编译、运行的一些命令。
7. stb/*: STB 库实现，这部分文件是直接从网络上下载的。
8. objects/*: 对场景中物体的全部实现，包括 Bezier 曲线、长方体、网格、平面、球体和对应贴图。
9. scenes/*: 场景描述，一个场景为一个头文件并封装在一个 namespace 中。
10. sources/*: 我渲染的场景所用到的一些资源文件。

1.2 代码采分点

1. Path Tracing: renderer.cpp/h (已经用 ifndef SPPM_MODE 宏标记)
2. SPPM: renderer.cpp/h (已经用 SPPM_MODE 宏标记)
3. 网格求交: objects/mesh.h
4. Bezier 求交: objects/bezier.h
5. AABB 包围盒 +KD 树加速网格求交: objects/mesh.h
6. KD 树加速 Hit Point 查找: sppm.cpp/h
7. 景深: renderer.cpp[274-276]
8. 网格贴图: objects/mesh.h objects/texture.h
9. 平面贴图: objects/plane.h objects/texture.h
10. 超采样抗锯齿: renderer.cpp[214-215, 268-269]
11. OpenMP 加速: renderer.cpp[208, 263, 291]

1.3 运行方式

本程序采用了把配置封装到 namespace 的方式来实现场景的读取。这样既没有 hardcode，也比较方便调试，同时提供 SPPM 模式和 PT 模式，通过 renderer.h 中的 SPPM_MODE 宏来控制，打开即为 SPPM 模式否则为 PT 模式。

在 scenes 中提供了几个场景的范例，在使用渲染器时需要在 renderer.cpp 中包含指定的头文件，并把 TO_RENDER 宏改为对于场景的 namespace 名。

在 namespace 中需要声明下面的变量：

1. width, height: 宽度和高度。
2. samples: 采样数（在 SPPM 模式下为每次迭代发射的光子数，在 PT 模式下是单像素点采样数）。
3. camera: 为 Ray 结构，指定了视点位置和方向。

4. camera_scale: 控制视点所视缩放的参数。
5. dof: 光圈大小 (仅 SPPM 模式可用)。
6. focal_distance: 焦平面距离 (仅 SPPM 模式可用)。
7. iteration_time: SPPM 模式下的迭代次数。
8. r_alpha: SPPM 模式下的 α 值。
9. sppm_radius: SPPM 模式下的初始半径。
10. energy: SPPM 模式下的光子能量。
11. ray_generator: SPPM 模式下的光子发射函数, 每调用一次需要返回随机的光线。
12. objects: 场景中的物体。
13. output: 输出的文件位置。

通过 make 命令编译后, 直接运行 main 程序即可运行。

2 场景类

2.1 基础: objects/base.h

定义了 2 维、3 维向量、颜色向量、射线、3 维范围、表面材质和像素结构等。

其中重点部分是 3 维向量的折射、反射等函数, 具体数学方法不再赘述, 另还有 3 维范围类采用了 Slab Method 定义了 AABB 包围盒并实现了和射线的求交判断。

2.2 长方体: cube.h

同上的 AABB 包围盒算法, 我使用了 Slab Method 对求交进行了判断, 这里不再赘述。

2.3 平面类: plane.h

把平面表示为 $ax + by + cz = 1$, 解方程 $n(o + td) = 1$ 得 $t = \frac{1-n\cdot o}{n\cdot d}$ 即可。

2.4 球面: sphere.h

内定义了球心 c 和半径 r , 代入射线方程是一个有关 t 的一元二次方程, 解出大于 0 且较小的 t 即可。

2.5 网格: mesh.h

其中定义了面片类 Surface, 为一个立体的三角形, 和光线求交利用了三点的比例构建方程, 即 $O + tD = (1 - u - v)V_0 + uV_1 + vV_2$, 设 $E_1 = V_1 - V_0, E_2 = V_2 - V_0, T = O - V_0$, 可以把方程改为 $[-D \ E_1 \ E_2][t \ u \ v]^T = T$, 根据 Crammer 法则解出 $t = \frac{1}{|-D \ E_1 \ E_2|}|T \ E_1 \ E_2|, u = \frac{1}{|-D \ E_1 \ E_2|}|-D \ T \ E_2|, v = \frac{1}{|-D \ E_1 \ E_2|}|D \ E_1 \ T|$ 。进一步的, 令 $P = D \times E_2, Q = T \times E_1$, 可得 $t = \frac{1}{|P \cdot E_1|}(Q \cdot E_2), u = \frac{1}{|P \cdot E_1|}(P \cdot T), v = \frac{1}{|P \cdot E_1|}(Q \cdot D)$ 。

同时我写了对 OBJ 文件的贴图支持, 贴图的具体算法是已知面片三点的对应贴图坐标, 对光线交点的贴图坐标进行插值即可。

而为了快速判断光线和大量面片的交点, 我使用了 KD 树来加速, 把面片的重心位置作为依据建立 KD 树, 后有光线要进行查询时先判断整个子树的 AABB 包围盒和光线的相交情况进行剪枝。最后的效果就是支持快速的超大规模的网格计算, 如图 1 有将近千万的面片 (后面的天空是个贴图), 我用 24 线程渲染一张 2560x1440 的图, 每个像素采样 200000 次, 只用了 4 小时左右, 效果比较好。

2.6 Bezier 曲线: bezier.h

我开始写 Bezier 时一直在犹豫最后要渲染一个什么样的曲线或者曲面出来, 我也通过一些软件进行过简单的设计, 比如画一头猪的轮廓, 可是我发现用控制点来求拟合总是非常耗费时间, 一是控制点和真实曲线的对应关系不明显导致非常难设计, 二是方程太复杂需要牛顿迭代来解非常不优美和困难。

这里我从二维的 Bezier 曲线出发, 参考了网络上一些做法也结合了自己的思考设计了一个通过输入锚点来反推 Beizer 曲线控制点实现拟合曲线



图 1: 大规模网格渲染结果

而不是设计曲线的方法，最后效果图中的猪就是通过输入原图片的 49 个锚点坐标让程序生成了 50 个分段且光滑连续的 Bezier 三次曲线做出的，下面阐释下具体的做法。

首先，我们有大量的有顺序的锚点，我希望在每两个锚点之间用一条 Bezier 三次曲线来拟合，这样锚点足够多的时候所有的曲线连接起来就是完整的图形。为了求出 Bezier 曲线，我们需要知道每两个锚点之间曲线的控制点，而且要保证两个相邻的曲线是相切的。先计算出相邻锚点的中点，后连接相邻边的中点，这样求出了新的 $n-1$ 个线段，然后核心思想是把线段的两端作为两条边的交点的左右控制点，于是我们把线段上按照两边的比例求出新的点，把这个点移动到两边的交点处，把原来的两端作为交点的左右控制点。之后对于相邻的两个锚点 A、B，我们把 A、A 的右控制点、B 的左控制点和 B 作为连接的 Bezier 曲线的 4 个控制点，这样可以求出一条新的三次 Bezier 曲线，可以证明是连续的，同时还可以不连接相邻边的中点用其他比例改变最后曲线的曲率。

这样说可能比较无力，我举一个具体的例子来说明这件事情。如图 2，为了求一条 AB 之间的 Bezier 曲线，我们先找到 AD、AB 和 BC 的中点 I、J 和 K，并连接相邻边的中点 (I 和 J, J 和 K)，之后在两个线段上找到 E、F 使得 $\frac{|IE|}{|EJ|} = \frac{|AD|}{|AB|}$ ，即靠近的点分开的线段的长度比是相邻两边的长度比，然后把线段 IJ 和线段 JK 移动到 A、B 的位置，具体方法就是在保证平行的情况下让 E 和 A 重合，让 F 和 B 重合，我们把露出来的线段两端作为控制点，比如 P 为 B 的“右控制点”、Q 是 A 的“左控制点”，这

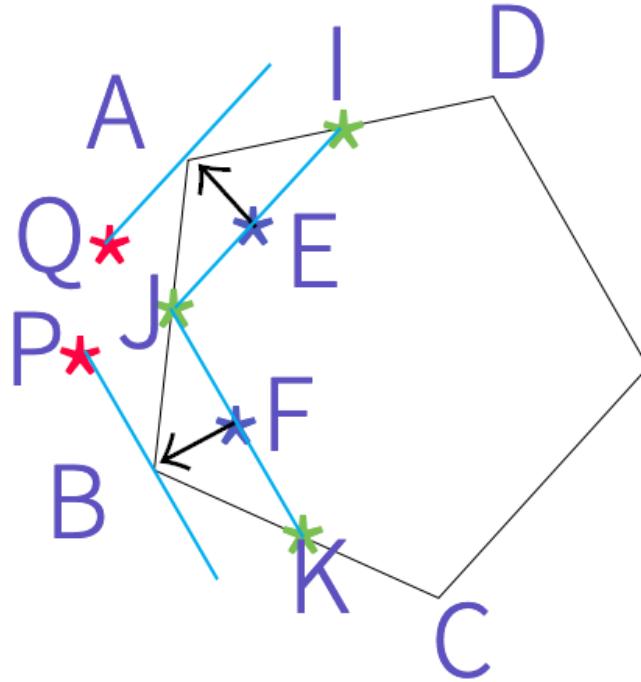


图 2: Bezier 控制点计算示意图

样对于 AB 直接的 Bezier 曲线，我们就用 B、P、Q 和 A 作为控制点来拟合，实践证明效果非常好。

这样，我们构造出了二维的情况，对于三维的曲面，我没有让它旋转（比较旋转没有什么意义，因为这样构造的曲线是闭合的），而我让它构成了一个柱体，在 XZ 平面上是曲线，在 Y 轴上是柱体，下面我来描述如何直接在 O(锚点个数) 的复杂度内直接求解。

首先上面构造出的曲线的方程形式都是 $B(t) = P_0 * (1-t)^3 + 3 * P_1 * t * (1-t)^2 + 3 * P_2 * t^2 * (1-t) + P_3 * t^3$ ，其中 P 是 4 个控制点（而且是二维的），我们把第一维映射到三维的 X，第二维映射到 Z 上，可以得到两个方程 $B_x(u)$ 和 $B_z(u)$ 。同时因为是柱体，所以我们对 Y 进行一个 $[y_1, y_2]$ 的限制，下面就可以开始解方程了，要解的方式比较显然是 $[B_x(u) \ v \ B_z(u)]^T = o + td$ ，于是我们可以先联立一个关于 u 和 t 的方程，把 t 消去，这样就成了一个

只有 u 的三次方程。

这个方程形如 $ax^3+bx^2+cx+d = 0$, 令 $x = y - \frac{b}{3a}$, 代入有 $y^3 + 3py + 2q = 0$, $p = \frac{c}{3a} - \frac{b^2}{9a^2}$, $q = \frac{d}{2a} + \frac{b^3}{27a^3} - \frac{bc}{6a^2}$, 我们的问题就成了解 $y^3 + 3py + 2q = 0$ 的实数根, 根据盛金公式, 另 $\Delta = q^2 + p^3$, 分为 3 个情况: A、 $\Delta > 0$, 则方程只有一个实根 $y = (-q + \sqrt{\Delta})^{\frac{1}{3}} + (-q - \sqrt{\Delta})^{\frac{1}{3}}$; B、 $\Delta = 0$ 时, 有三个实根, $y_1 = -2q^{\frac{1}{3}}, y_2 = y_3 = q^{\frac{1}{3}}$; C、 $\Delta < 0$ 时, 令 $\alpha = \frac{1}{3}\arccos\left(\frac{-q\sqrt{-p}}{p^2}\right)$, 则有 $y_1 = 2\sqrt{-p}\cos(\alpha), y_2 = 2\sqrt{-p}\cos(\alpha + \frac{2\pi}{3}), y_3 = 2\sqrt{-p}\cos(\alpha + \frac{4\pi}{3})$ 。于是我们 $O(1)$ 解出对应方程, 判断 t 的范围和 y 的范围并取最小的 t 即可。

还有一种情况是, 如果打在了柱的上下两侧, 需要直接根据 y 来解出 t, 解出 t 后需要判断是不是交点在曲线内, 这时候的做法就是从交点引一条射线, 再循环一遍所有的方程, 如果有奇数个交点就是在曲线的内部, 否则在曲线的外部。



图 3: 没加贴图的 Bezier 柱体

最后的结果就是达到了图 3, 为了明显我把贴图去掉了, 还可以看到这是一个立体的图形, 加了贴图的如图 4。

3 渲染

3.1 三种材质

在初期学习时我参考了 smallpt 的代码, 材质也是参考 smallpt 写的, 具体分为全漫反射的面 (如地板)、全反射的面 (如镜子) 和反射折射都有



图 4: 加了贴图的 Bezier 柱体

的特殊材质（如玻璃）。漫反射就是随机一个新方向射出，反射就是根据法线计算一个入射对称的反射光线，而折射根据材料的不同计算对应的折射光线，对于第三个材质，根据物体的属性，根据 Fresnel 效应，计算出折射和反射的概率即可，具体原理不再赘述。

为了加速计算，在概率较小或者光通量较小时可以直接剪枝，我也参考了 smallpt 的轮盘赌方式进行加速。

3.2 Path Tracing

为了学习和理解，我首先实现了 Path Tracing 算法，从视点出发，对光线进行反向追踪，但是毕竟是反向追踪有些刁钻的光源射出的光线产生的一些特殊的如漫反射焦散不容易采集，但是也达到了比较好的效果，如上文中的城堡（图 1）和下面图 5 就是用 Path Tracing 实现的，缺点是有比较多的噪点。

3.3 PPM 和 SPPM

实现了 Path Tracing 后，我根据 Toshiya Hachisuka 等人的 Progressive Photon Mapping 论文在原基础上实现了 PPM，其精髓就是一个双向的光的发送和吸收，利用 Path Tracing 算法 Backtrace 到漫反射面，后在光源处打光子送到漫反射面。这里不能精确地命中，所以在每个 Backtrace 到的点上都有一个吸收的半径，随着每轮迭代过后吸收的光子越来越多，根据论

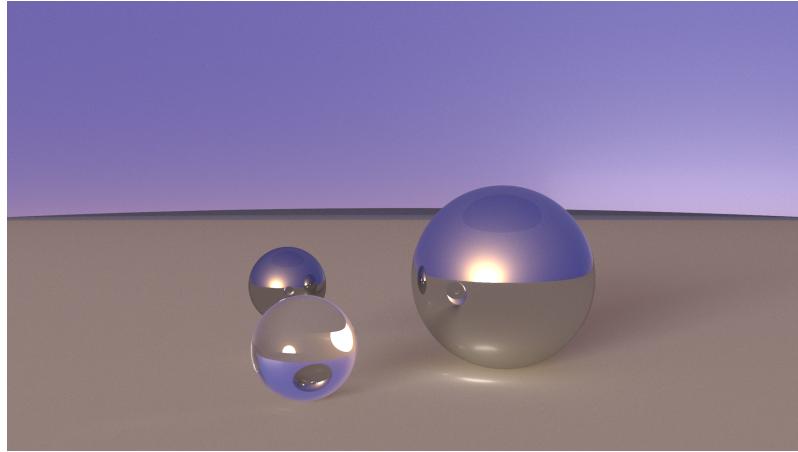


图 5: Path Tracing 效果图

文中的公式调整半径和已经积累的光通量，可以达到一些 Path Tracing 做不到的效果，在理论上也可以收敛。而 SPPM 把半径的维护放在了区域上，并加了一些扰动，实现了更好的效果。这里我直接把半径和光通量直接维护到对于的像素上，一个像素对应多个 Hit Point，每次迭代后都重新找一遍这些 Hit Point。

比如下图 6是用 Path Tracing 渲染出的猪，而上面 Bezier 章节中的图 4是用 SPPM 渲染出的，可以看到对于同样的总采样次数，Path Tracing 更暗，有大量噪点，而 SPPM 的图几乎完胜而且渲染出了一些漫反射焦散和软阴影的效果，图 7和图 1是分别用 SPPM 和 PT 渲染出的结果，可以看到面对大规模的场景有些过分强调光的效果也有些不真实了，相比 PT 的图没有那么多细节，但是看起来光效比较足，进一步的可能需要更加细致的参数调节，不过没出来的效果是都出来了而且非常明显，也不失为一个好的结果。

3.4 特效

3.4.1 超采样抗锯齿

类似第一个大作业，我们把一个像素点分隔成 4 个小像素点，计算后进行加权和（Path Tracing），或者分别把四个小像素点的 Hit Point 加到大像素上来维护（SPPM），这样就完成了 4 倍的超采样抗锯齿。

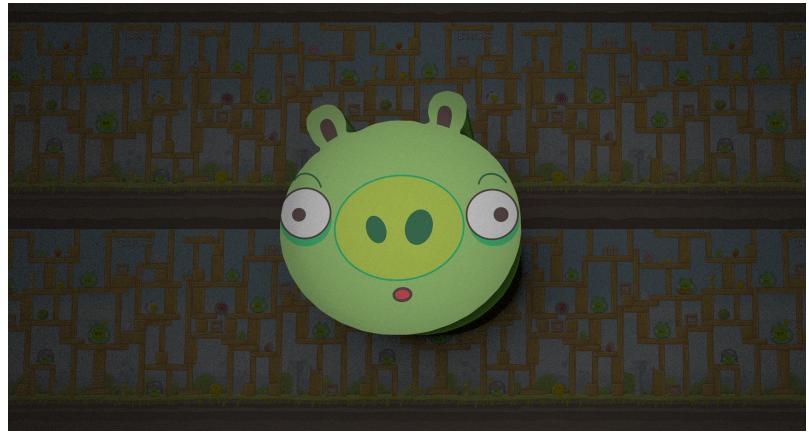


图 6: Path Tracing 渲染的猪



图 7: SPPM 渲染的城堡

3.4.2 贴图

这里我完成了平面的贴图和 OBJ 网格的贴图，具体平面的贴图就是建立一个平面点到贴图的映射关系，这样可以实现平铺。而对于 OBJ 网格，如上面所说，对击中点所在面片的三个顶点的贴图坐标插值出新的贴图坐标即可。

3.4.3 景深

对于景深的实现，我参考了 <http://cn.voidcc.com/question/p-ndxrzfdgbcu.html> 的做法，如图 8，在视点处加一个光圈半径，视点每次随机在光圈上采样，并设计一个焦平面，这样如图原红线和蓝线的光线会变成红色和蓝色的光锥，只有在进而模糊，而在焦距上的点还保持线所以很清晰。

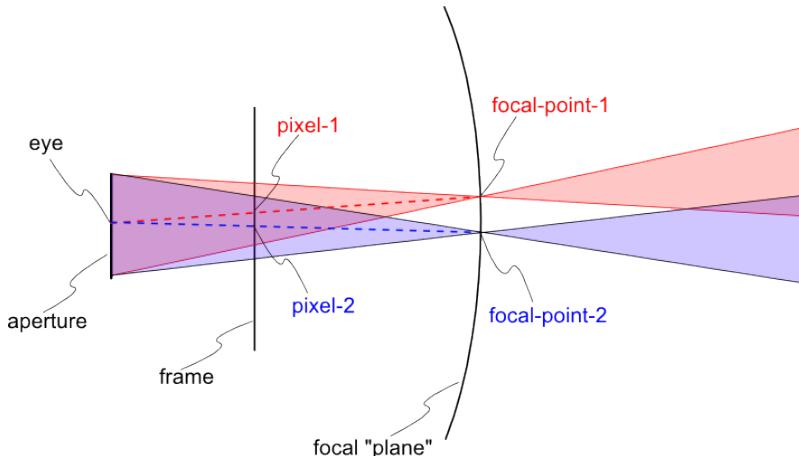


图 8: 景深的原理

如图 9 和 10 分别是没加和加了景深的恐龙，可以发现明显第二个图片聚焦在了第一个恐龙的头上，效果比较明显。

3.5 加速

3.5.1 OpenMP

最便利的加速方式，在 PT 中对像素并行，在 SPPM 中对像素采样和光线射出并行即可。



图 9: 没加景深的恐龙

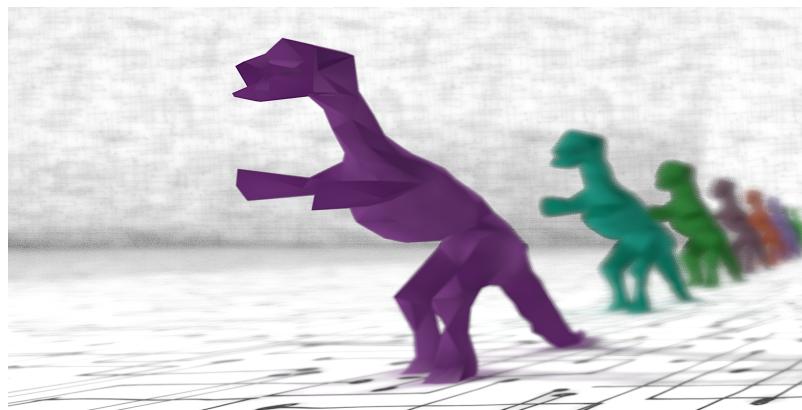


图 10: 加景深的恐龙

3.5.2 面片包围盒

在对大规模网格求交中，使用了 KD 树来维护网格，上面已经提到了，这里不再赘述。

3.5.3 Hit Point 维护

在 SPPM 中，为了维护 Backtrace 到的漫反射平面上的点，这里也采用了 KD 树来维护，在射出光线时可以直接调用 KD 树查询临近的在半径范围内的 Hit Point。

4 感谢

本次作业对我来说比较困难，参考了大量资料也问了很多同学，渲染了不下千张图片，收获很大。在此，感谢 smallpt 的作者和计六年级的翁家翌同学（Honor Code），感谢计 75 班的王征翊对我的一些思路上的讲解和帮助，也感谢老师一学期的讲解和助教的工作。