计算机图形学实验报告
基于 Bezier 曲线的三维造型与渲染

张钰晖

2015011372, yuhui-zh15@mails.tsinghua.edu.cn, 185-3888-2881

2017 年 6 月 25 日

目录

1 问题描述

本次作业可以认为分为两个环节，第一步是 Bezier 曲线的三维造型，第二步是图像渲染。

Bezier 曲线是由控制点组成的插值函数多项式，k 阶 Bezier 曲线由 (k + 1) 个控制点组成，Bezier 曲面可以由 Bezier 曲线旋转而得，也可以直接由二维的控制点组成。本实验中，我选择了比较有挑战性的双三次 Bezier 曲面，实现了直线与 Bezier 曲面的求交、Bezier 曲面法向的计算、Bezier 曲面生成 obj 网格等方法。

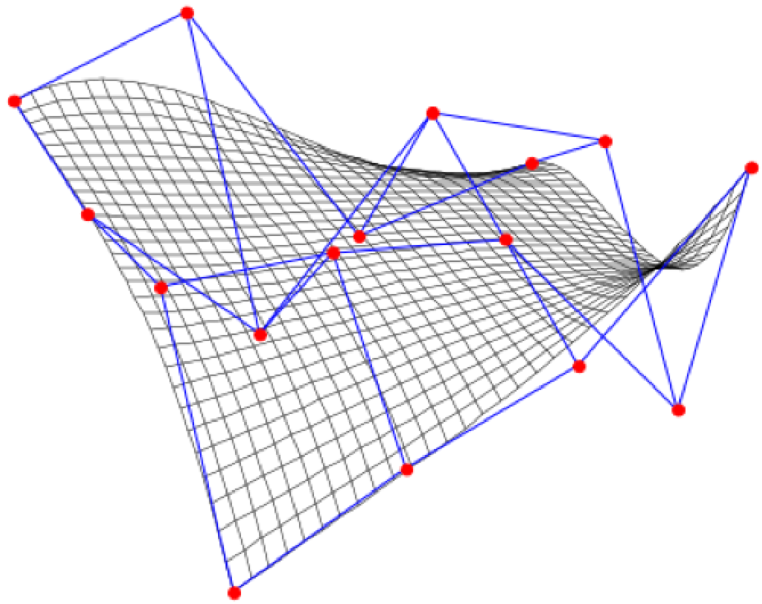

图 1: Bezier 双三次曲面

图像渲染是真实感图形学的重要部分，也是无数的学者研究了将近半个多世纪的问题。图像渲染有很多种算法，按照发展历史依次是光线投射、光线追踪、光子映射、渐进式光子映射，在本次作业中，我选择了近几年新提出的渐进式光子映射作为渲染引擎。

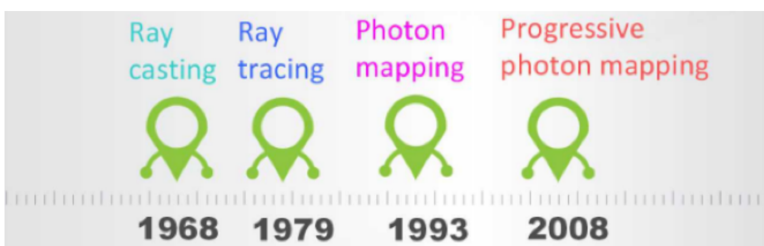

图 2: 图像渲染发展历史

下面，我将分两部分分别讲解 Bezier 双三次曲面与渐进式光子映射，以及贴图、景深、渲染加速的实现。

最终选题：

渲染算法：渐进式光子映射（90%），曲面求交：双三次曲面求交（40%），贴图（10%），景深（10%），渲染加速（5% 10%），共计 155%160%。

2 Bezier 双三次曲面

2.1 Bezier 曲线

由于 Bezier 曲面的固定某一维度后将退化为 Bezier 曲线，因此理解 Bezier 曲线是理解 Bezier 曲面的基石。Bezier 曲线是由控制点组成的曲线，k 阶 Bezier 曲线由 k+1 个控制点构成，可以认为曲线上每个点的值都是所有控制点的加权和。

$P(t) = \sum_{i=0}^{n} B_{i,n}(t) P_i, t \in [0,1]$

其中，$B_{i,n}(t)$ 是伯因斯坦多项式，

$B_{i,n}(t) = C_n^i t^i (1-t)^{n-i}, i = 0, ..., n$

对三阶 Bezier 曲线，我们简化标记，

$b_0(t) = (1-t)^3$

$b_1(t) = 3t(1-t)^2$

$b_2(t) = 3t^2(1-t)$

$b_3(t) = t^3$

该数学表达式可以写成矩阵和向量相乘的形式。

还有另一种曲线求值的方式，被称为 De Casteljau 算法，该算法数值稳定，但计算消耗量略大，不再赘述，可以参考维基百科。

曲线求值实现的代码如下：

```
Vec3f evalBezierCurve(const Vec3f *P, const float &t) {
   float b0 = (1 - t) * (1 - t) * (1 - t);
   float b1 = 3 * t * (1 - t) * (1 - t);
   float b2 = 3 * t * t * (1 - t);
   float b3 = t * t * t;
   return P[0] * b0 + P[1] * b1 + P[2] * b2 + P[3] * b3;
}
```

2.2 Bezier 曲面

Bezier 曲面是 Bezier 曲线在二维上的延伸，k 阶 Bezier 曲面由 (k+1)*(k+1) 个控制点构成，可以认为曲面上每个点的值都是所有控制点的加权和。

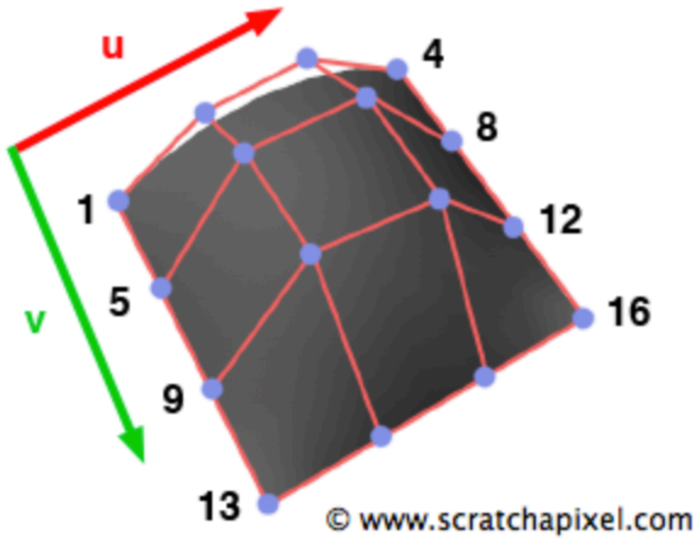

图 3: Bezier 曲面

$P(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) P_{ij}$

其中，$B_{i,n}(t)$ 依然是伯因斯坦多项式，

$B_{i,n}(t) = C_n^i t^i (1-t)^{n-i}, i = 0, ..., n$

因此，我们可以将 Bezier 曲面求值化归为 2 次曲线求值问题，先沿着 u 方向求值，再沿着 v 方向求值。

曲面求值实现的代码如下：

```
Vec3f evalBezierPatch(const Vec3f *controlPoints, const float &u, const float &v)
{
   Vec3f uCurve[4];
   for (int i = 0; i < 4; ++i) uCurve[i] = evalBezierCurve(controlPoints + 4 * i, u);
   return evalBezierCurve(uCurve, v);
}
```

2.3 Bezier 曲面偏导数

对三阶 Bezier 曲面，曲面上 (u,v) 点的值为，

$P(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u) b_j(v) P_{ij}$

其中，

$b_0(t) = (1-t)^3$

$b_1(t) = 3t(1-t)^2$

$b_2(t) = 3t^2(1-t)$

$b_3(t) = t^3$

对上面的伯因斯坦基函数求导得到，

$b_0^{'}(t) = -3(1-t)^2$

$b_1^{'}(t) = 3(1-t)^2 - 6t(1-t)$

$b_2^{'}(t) = 6t(1-t) - 3t^2$

$b_3^{'}(t) = 3t^2$

故对于 Bezier 曲线，其导数即为 $b^{'}$ 的加权和。

对 u 方向求偏导数，可以先通过 v 分量生成相应的 Bezier 曲线，再通过 Bezier 曲线求导的方式实现；对 v 方向求偏导数也是类似的，在这里略去数学推导。

曲面求偏导数实现的代码如下：

```
Vec3f dUBezier(const Vec3f *controlPoints, const float &u, const float &v)
{
  Vec3f P[4];
  Vec3f vCurve[4];
  for (int i = 0; i < 4; ++i) {
     P[0] = controlPoints[i];
     P[1] = controlPoints[4 + i];
     P[2] = controlPoints[8 + i];
     P[3] = controlPoints[12 + i];
     vCurve[i] = evalBezierCurve(P, v);
  }

  return -3 * (1 - u) * (1 - u) * vCurve[0] +
     (3 * (1 - u) * (1 - u) - 6 * u * (1 - u)) * vCurve[1] +
     (6 * u * (1 - u) - 3 * u * u) * vCurve[2] +
     3 * u * u * vCurve[3];
}

Vec3f dVBezier(const Vec3f *controlPoints, const float &u, const float &v)
{
  Vec3f uCurve[4];
  for (int i = 0; i < 4; ++i) {
     uCurve[i] = evalBezierCurve(controlPoints + 4 * i, u);
  }

  return -3 * (1 - v) * (1 - v) * uCurve[0] +
     (3 * (1 - v) * (1 - v) - 6 * v * (1 - v)) * uCurve[1] +
     (6 * v * (1 - v) - 3 * v * v) * uCurve[2] +
     3 * v * v * uCurve[3];
}
```

2.4 Bezier 曲面法向量

由下图可以清晰地看出，曲面 (u,v) 点的法向量即为其偏导数的叉积，即 $N(u,v) = dU(u,v) \times dV(u,v)$

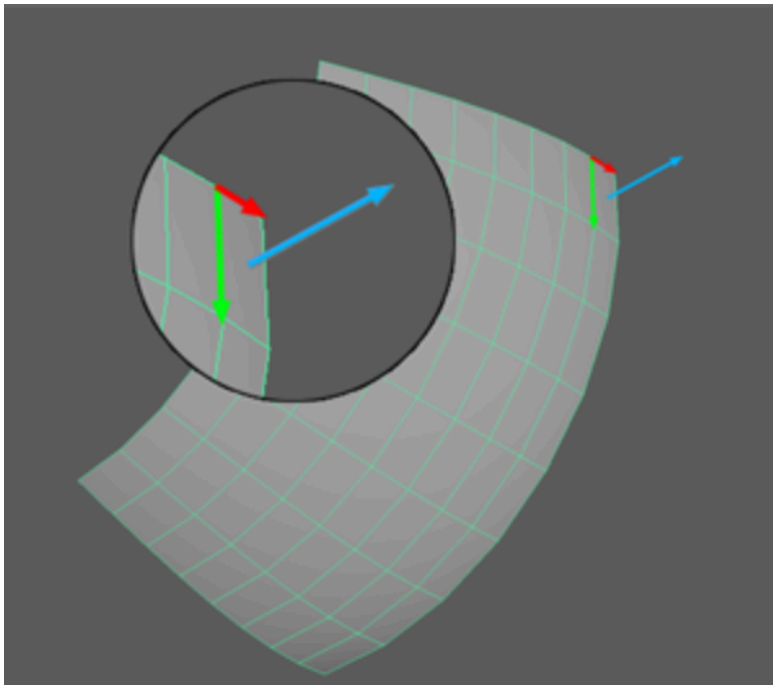

图 4: Bezier 曲面法向量

曲面法向量实现代码如下：

```
Vec3f normal(const Vec3f *controlPoints, const float &u, const float &v)
{
  Vec3f dU = dUBezier(controlPoints, u, v);
  Vec3f dV = dVBezier(controlPoints, u, v);
  return cross(dU, dV);
}
```

2.5 Bezier 曲面求交

Bezier 曲面求交问题是渲染能稳定运行的关键，由于 Bezier 曲面不具有平面、球、三角面简单的几何解析解法，如何设计一个数值稳定的算法直接影响了后期渲染的质量。

这部分我耗了大量的心血研究和尝试各种算法，在此进行一一介绍。

2.5.1 生成网格法

求交一种最为简单的方式便是把 Bezier 曲面通过一些均匀的点，通过计算其值和法向量，将曲面细分为网格 (如矩形面片或三角面片) 求交，这也是生成 obj 网格的原理。

代码原理较为简单，即曲面细化，但实现较为冗长，在此不再赘述，感兴趣的读者可以参考代码。

图 5: 生成的 Utah 壶网格

这种方式将曲面转化为简单几何面，求交是数值稳定的。但是由于 Bezier 曲面的光滑型优于面片，故渲染时需要再次对法向量插值确保渲染出来的曲面色彩均匀。这种方法虽然数值稳定，但是为了保证精度需要生成很多的面片，速度会很慢，而且如果不进行法向量插值渲染出来的图效果也不均匀。

总体评价：实现难度 2 稳定性 5 速度 2 效果 4

2.5.2 牛顿迭代法

对光线与曲面求交而言，设光线表达式为 $C(t) = O + t * N$，曲面表达式为 $P(u, v)$，则交点即满足方程

$F(t, u, v) = C(t) - P(u, v) = 0$

可以使用牛顿迭代法求解。

牛顿迭代法是不动点迭代法的一种，是数值分析的基本内容，在此不再进行赘述，可以参考维基百科。

对多元函数的牛顿迭代法，欲求 $F(x)=0$，其迭代公式可以写为

$x^{(i+1)} = x^{(i)} - J^{-1}F(x^{(i)})$

其中 J 为 Jacobi 矩阵，停止条件可以设为两次迭代差值的无穷范数小于 eps 或迭代次数大于 maxiter。需要实现矩阵求逆。

牛顿迭代法是局部收敛的，其收敛性依赖于初值的选取，是数值敏感的，而且得到的解只是近似解，不是解析解。对该问题而言，很难选择一个合适的初值，故无法控制牛顿迭代法收敛性。假设曲面与光线有多个交点，初值没有选好，则牛顿法将变得数值不稳定，导致渲染出来的图惨不忍睹。

参见下图，左边的曲面和右边的曲面是完全一样的曲面，迭代初值完全相同，右边的曲面求交完好，左边的曲面求交几乎全军覆没。

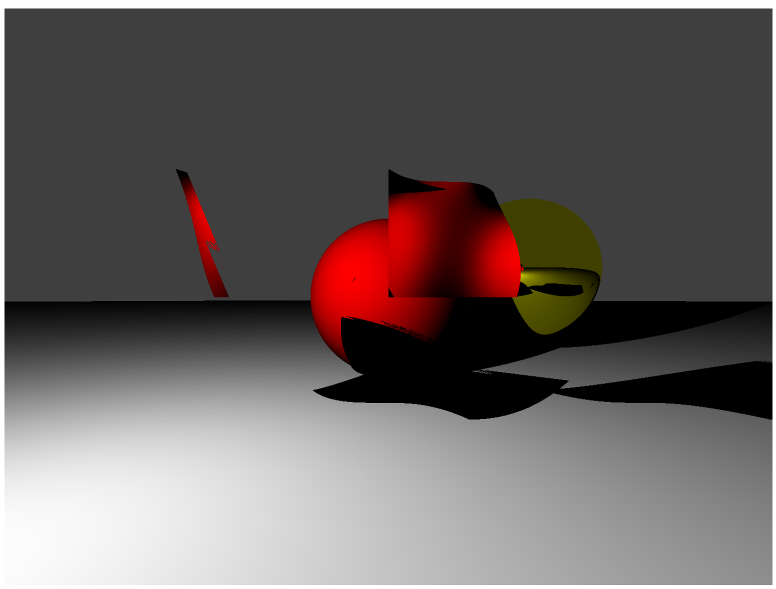

图 6: 牛顿迭代法的问题

由于牛顿迭代法严重依赖于初值的选取，因此我们可以先通过对 Bezier 曲面建立包围盒大致确定交点位置再进行迭代，但笔者尝试过后发现对一些扭曲程度较高的曲面其仍然数值不稳定。

但牛顿迭代法相对于生成网格法而言，速度较快，求交正常的话渲染较为均匀，然而在渲

染中，数值稳定性远远要比速度重要。

总体评价：实现难度 3 稳定性 1 速度 3 效果 1

2.5.3 随机牛顿迭代法

正如前文所说，由于牛顿迭代法严重依赖于初值的选取，因此我们可以随机生成一些点（取的点数即为取样次数），从这些不同的点开始迭代，选出符合要求的且光走过距离最短的点。

但是由于其随机性，取样次数较高时（例如 20）对一些简单曲面应对良好，但笔者尝试过后发现对一些复杂的曲面拼合体其仍然数值不稳定。如下图著名的 Utah 壶，改变取样次数可以看出其噪点明显减少，说明求交准确度提高，但仍有噪点不可根除。如果不加随机过程的话，壶几乎全部渲染不出来，已经是一个较大的进步，但不能使人满意。

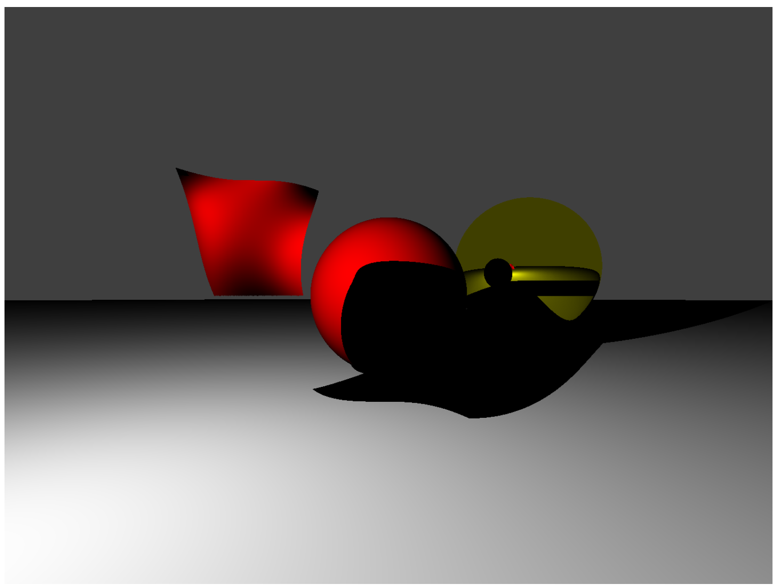

图 7: 渲染出来的左边的面片

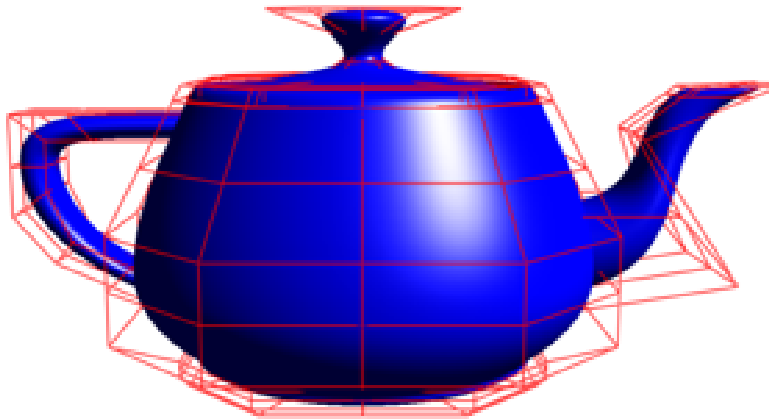

图 8: 由 32 个 Bezier 曲面和 306 个控制点构成的 Utah 壶

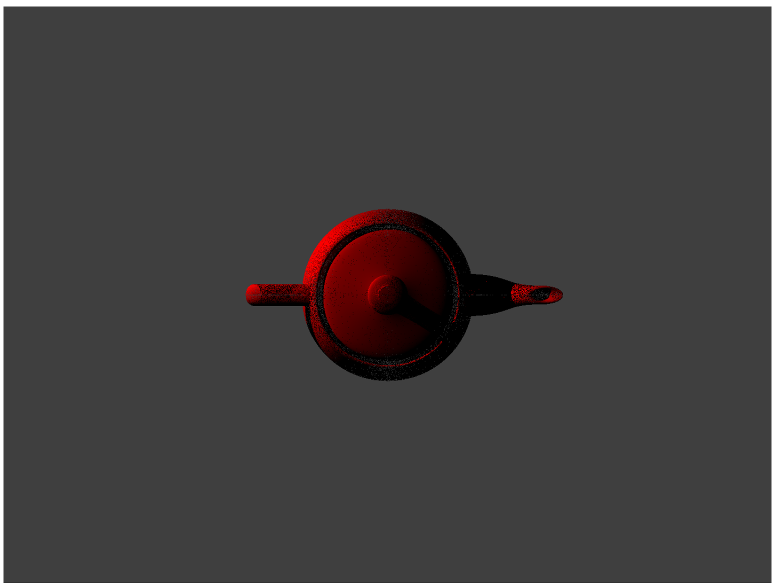

图 9: 采样次数为 3 的随机牛顿迭代法

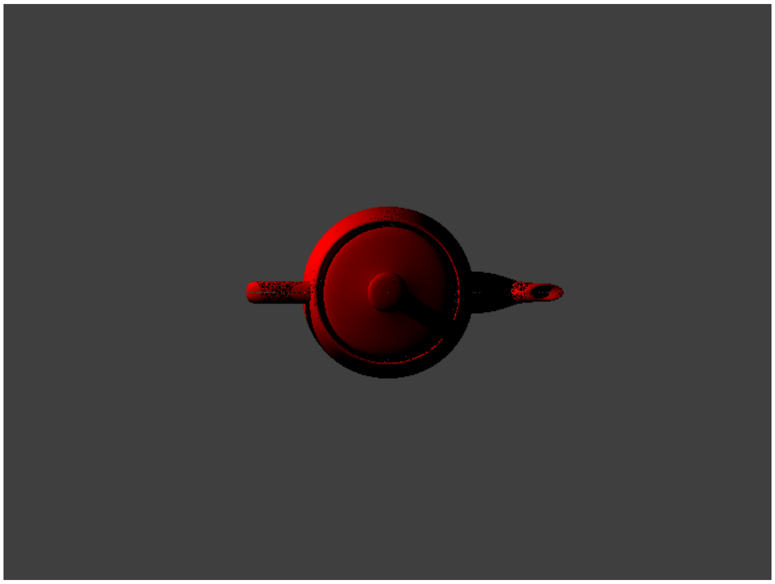

图 10: 采样次数为 10 的随机牛顿迭代法

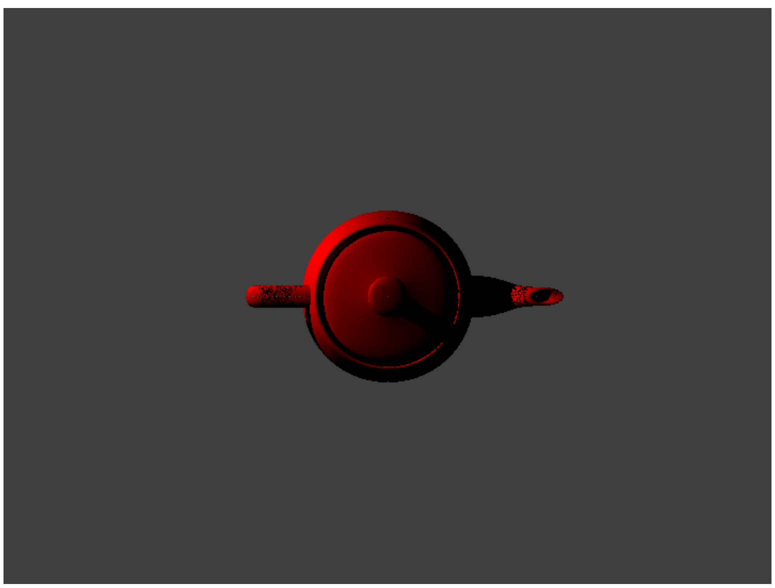

图 11: 采样次数为 20 的随机牛顿迭代法

由于渲染时间正比于取样次数，取样次数很高的时候，渲染速度及其慢，而且效果也不是很好。

总体评价：实现难度 4 稳定性 3 速度 1 效果 3

2.5.4 曲面剖分法

尝试到这里，看着这残缺不全的壶我已经感到绝望了。但是通过阅读教材我发现教材提供了一种四叉树的方式求交，即把曲面不断四分，直到每个曲面足够小为止，构成四叉树，然后对每个节点包围盒求交（对叶子用牛顿迭代法求交）。

因为叶片表示的曲面足够小，所以牛顿迭代法数值稳定，即使初值选取不好也基本能收敛到解。但是这个算法一是要实现一颗四叉树，代码便会显得很冗长，二是这样剖分下去速度也不一定会很快，如果当前节点的包围盒不与光线相交，该当前节点代表的曲面便没有必要继续剖分下去。

因此，我想出了一种更为简单的方式，直接采用队列。对当前节点代表的曲面，四分后对每个 1/4 曲面，用包围盒判断是否相交，如果相交加入队列，否则弃之。这样深度优先搜索下去，直到队列首元素表示的曲面足够小为之。这时对队列中的所有元素（可以保证所有元素代表的曲面都足够小）一一采用牛顿法求交，选出光走过距离最小的面片即可。

在实现过程中，要注意两个细节。

一是包围盒的实现，对 Bezier 曲面来说，由于其具有凸包性，直接记录 AABB 包围盒的 max(x, y, z) 和 min(x, y, z) 即可，然后用 Slabs 算法快速判断光线是否与其相交，提高程序运行速度。

二是四分曲面需要使用 De Casteljau 算法，而不是直接等距离四分，如果直接等距离四分则曲面不再是原曲面，故渲染出来的图将是不均匀的面片。

经过这一系列的重构，一个完整而漂亮的 Utah 壶终于展现在了我的眼前。而且由于在大部分情况下，曲面与光线只有一个交点，这样迭代的时间复杂度仅仅只有 O(logn)，并且是和包围盒判断求交，其速度又远远快于牛顿法。只有最后一步需要牛顿迭代法，此时面片足够小，牛顿迭代法是稳定安全的，收敛也非常快。

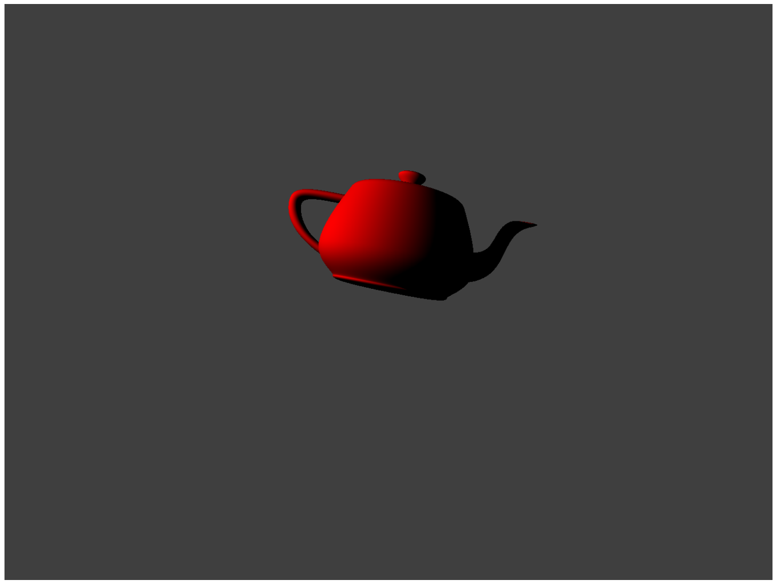

图 12: 完整的 Utah 壶

故整体速度有了质的飞跃，在 PPM 引擎下，在场景中放入 Utah 壶（32 个 Bezier 面片），每轮 500W 光子，3 小时便可以运行 6 轮，这再一次让我感受到算法的力量。

总体评价：实现难度 5 稳定性 5 速度 5 效果 5

3 渲染算法

正如引言所描述的那样，渲染算法一直是一代计算机图形学科学家的研究的核心。我将从光线追踪讲起，谈到路径跟踪，再到光子映射，最后谈到最终所使用的渐进式光子映射算法。由于前三个不是本文的重点，故只大致介绍原理，而且渐进式光子映射算法需要用到光线跟踪，必须对其有大致的了解。

3.1 光线追踪

光线追踪（Ray Tracing）是一种最为经典的渲染算法，其原理大致描述如下：

（1）要得到屏幕上像素 P 的颜色，从视点发出一条经过 P 的光线，找到与场景中物体的第一个交点 Q，交点 Q 的颜色就是像素 P 的颜色。求出物体上 Q 点的颜色，便也就求得了像素 P 的颜色。

（2）场景中所有的光源对 Q 的颜色有影响（直接光照），场景中其他物体对光源发出的

光线进行反射或是折射（间接光照），经过反射或折射后再次发射的光线同样对 Q 的颜色有影响。这些因素确定了交点 Q 的颜色。

（3）产生间接光照的物体表面的颜色同样是需要求解的，只需要递归追踪打在 Q 点上的光线即可，在算法的每一步追溯这条光线，对最后得到的效果与直接光照做一叠加，便可以求得 Q 点的颜色。

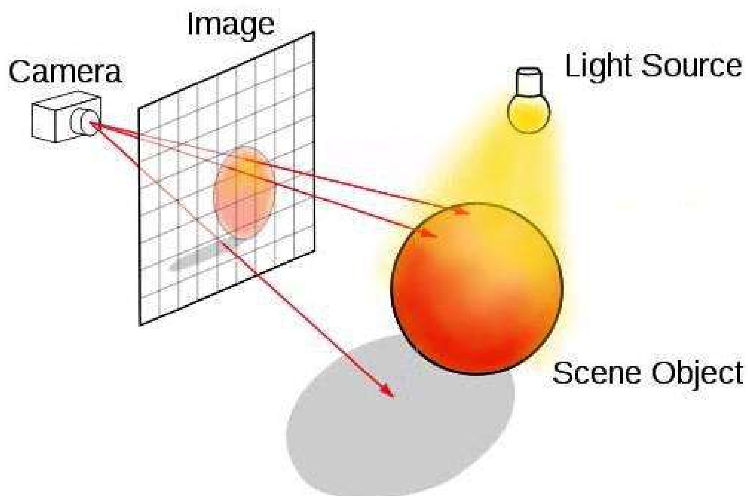

图 13: 光线追踪原理

光线追踪的主要瓶颈在于：

（1）漫反射入射光线无法追踪。

（2）物体表面的多种属性难以模拟。

第一个问题可以采取 Phong 模型解决，Phong 模型只考虑漫反射物体与光源的位置，且模拟效果比较理想，是一种经典的方法，在此不再赘述。

综合评价：实现难度 2 速度 5 效果 3

3.2 路径跟踪

路径追踪（Path Tracing）是一种基于光线跟踪的改进算法，和光线追踪有很大的类似之处，主要是基于光线追踪的两大问题进行改进：

（1）漫反射入射光线无法追踪

在路径追踪中，通过大量的随机模拟，每次在表面 Q 随机选定一个方向作为追踪的方向。我们对每个像素做充足的采样，并对采样的结果取平均值，从而模拟出较为真实的漫反射效果。

（2）物体表面的多种属性难以模拟

在路径追踪中,假设物体的表面属性为 70% 漫反射 +30% 镜面反射,在每次递归时,以 70% 的概率追踪漫反射光线（追踪的时候在每一个被追踪的物体表面随机取一个追踪的方向）, 以 30% 的概率进行镜面反射追踪即可。

综合评价：没有实现，故不进行评价

3.3 渐进式光子映射

渐进式光子映射（Progressive Photon Mapping）是近几年比较流行的算法，其原理大致描述如下：

（1）通过光线追踪，建立整个场景的碰撞点，记录碰撞点的各种信息，并使用碰撞点图（KDTree）存储这些点。

（2）使用光子发射器从光源发射指定多个光子，每当光子碰到漫反射表面，就在碰撞点图中询问光子位置是否在碰撞点半径内部，如果是，则将光子信息存入，否则丢弃光子。光子信息的存入实际上就是指将光子的能量加到碰撞点的属性上。

（3）顺序扫描碰撞点图，根据公式更新碰撞点的各项信息值。

$k = \frac{N+\alpha M}{N+M}$ $R^2 * = k$ $\Phi * = k$ $N + = M$ $M = 0$

N 为累计光子数，M 为该轮新增光子数，R^2 为碰撞点半径，Φ 为累计光通量，α 为半径收敛系数

（4）重复执行 (2)(3) 若干轮后（详见下文参数选择），对图像进行渲染，根据每个像素对应的碰撞点，及各碰撞点累计光子数，便可以得到该像素点的颜色。

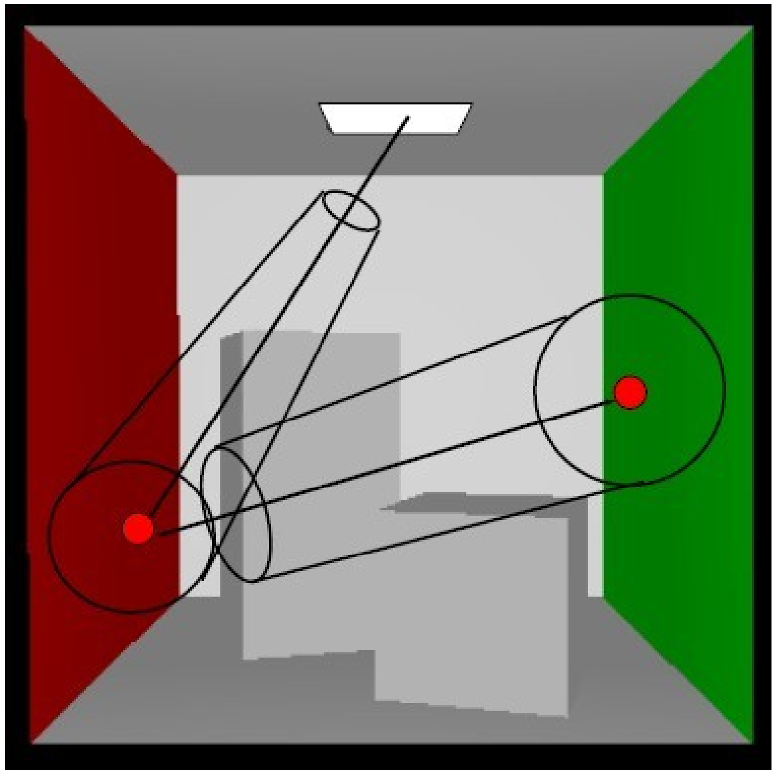

图 14: 光子映射部分原理

使用渐进式光子映射作为渲染算法，渲染效果非常出众，可以实现很多路径追踪实现不出来的效果，缺点是渲染非常慢，必须发射足够多的光子噪点才能收敛，不能作为实时渲染算法。另外，bug 非常难调试，只能一遍又一遍读代码以及逐个像素点 debug。

综合评价：实现难度 5 速度 1 效果 5

值得注意的是，渐进式光子映射模型参数的选择是个很有技巧的问题。

3.3.1 初始半径 r

初始半径 r 小，则噪点起初很平均，起初收敛速度快，但最后噪点很难被完全消除。

初始半径 r 大，则噪点起初会成黑块，起初收敛速度慢，但最后噪点基本全部被消除。

下面这两幅图可以清晰地看出这个关系，两张图光子散播数均足够大，确保已收敛。

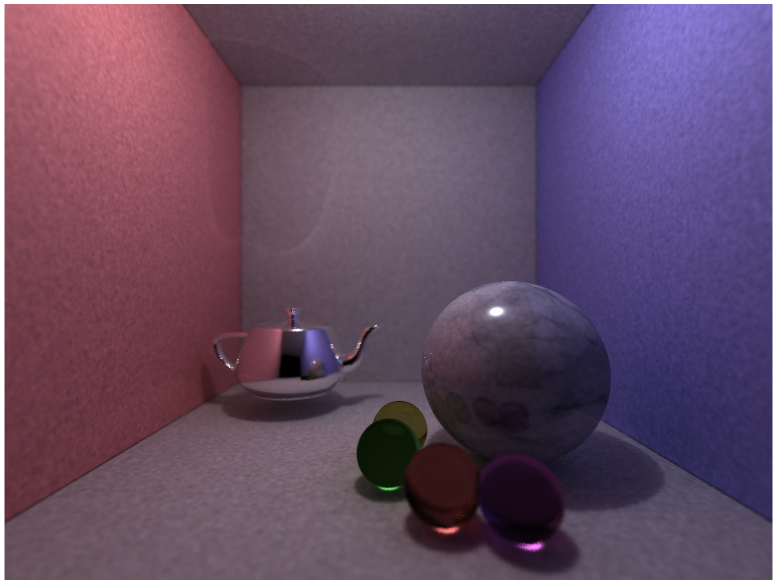

图 15: 初始半径为 0.5

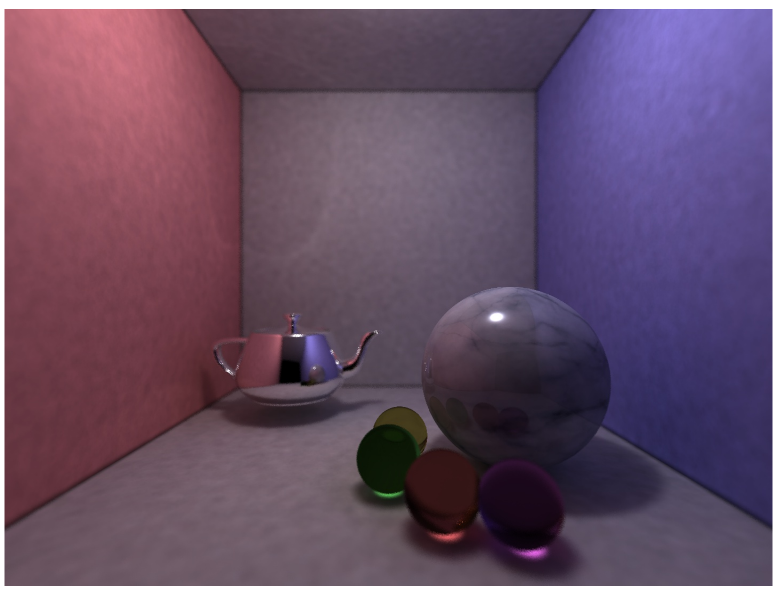

图 16: 初始半径为 1.0

在最终程序中，初始半径选择了 1.2（这个和图片建模的空间大小有关，图中空间尺寸约为 100x100x100）

3.3.2 半径收敛系数 α

半径收敛系数 α 小，采样半径收缩快，收敛速度快，但图片噪声较大。

半径收敛系数 α 大，采样半径收缩慢，收敛速度慢，渲染结果较精细。

在最终程序中，半径收敛系数选择了 0.7

3.3.3 光子散播数 n

理论上，光子散播次数越大，每次散播光子数越多，图片噪声越小，渲染越精细，但所耗时间也越长。

下面这五张图可以清晰地看出这个关系，但由于初始采样半径设置太小，故最后噪点较重，且无法继续收敛。

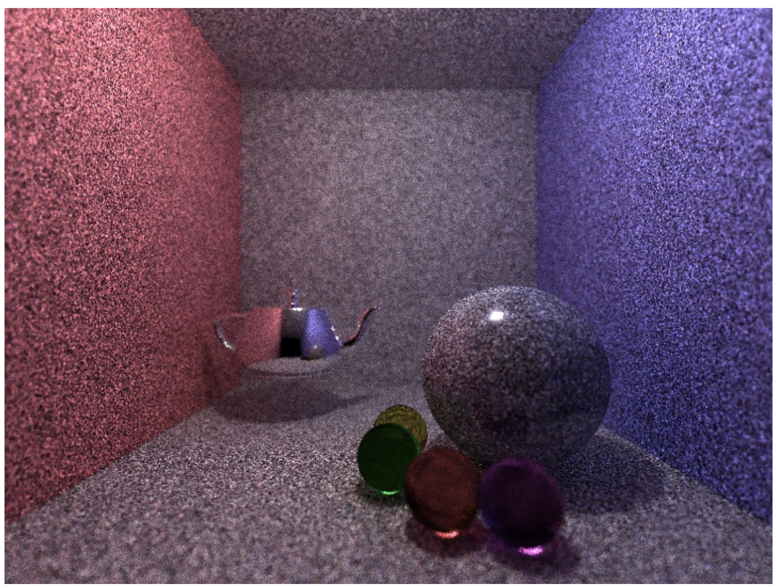

图 17: 10 万光子

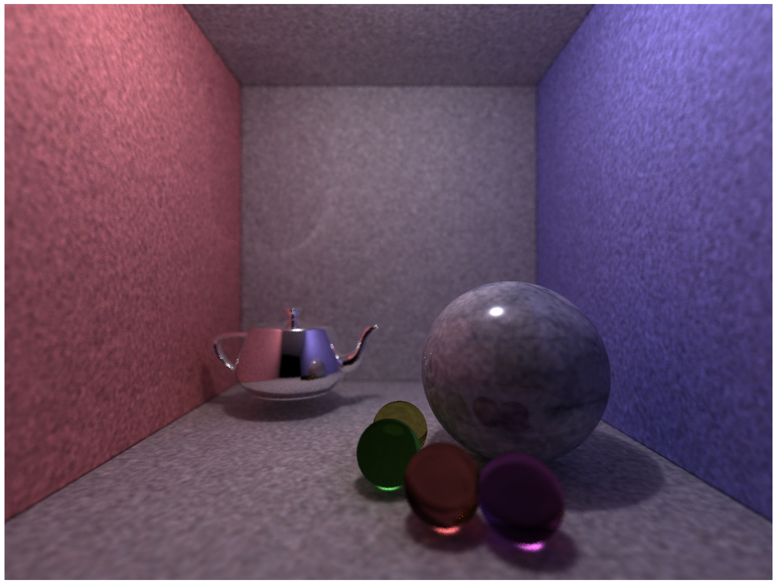

图 18: 500 万光子

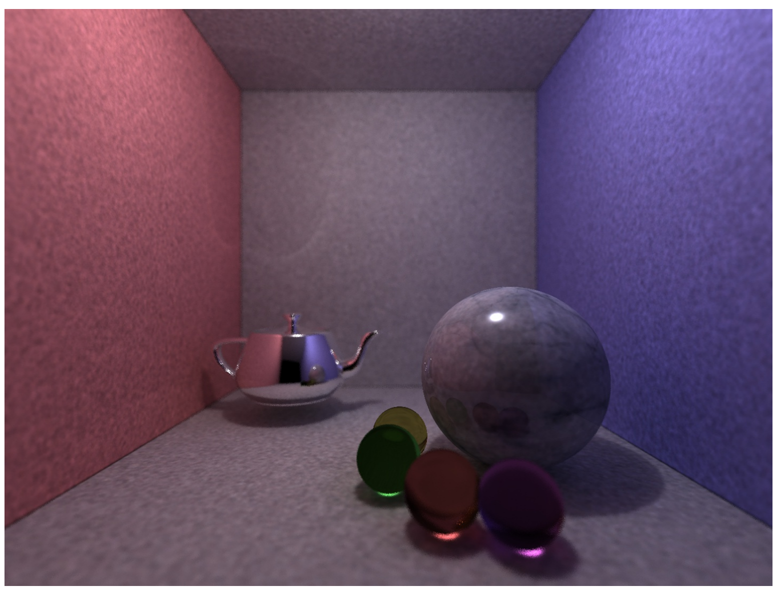

图 19: 1000 万光子

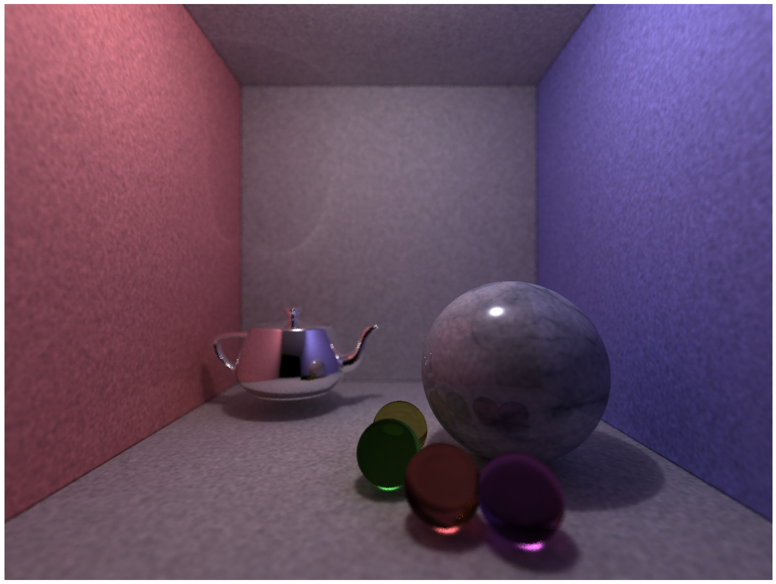

图 20: 1500 万光子

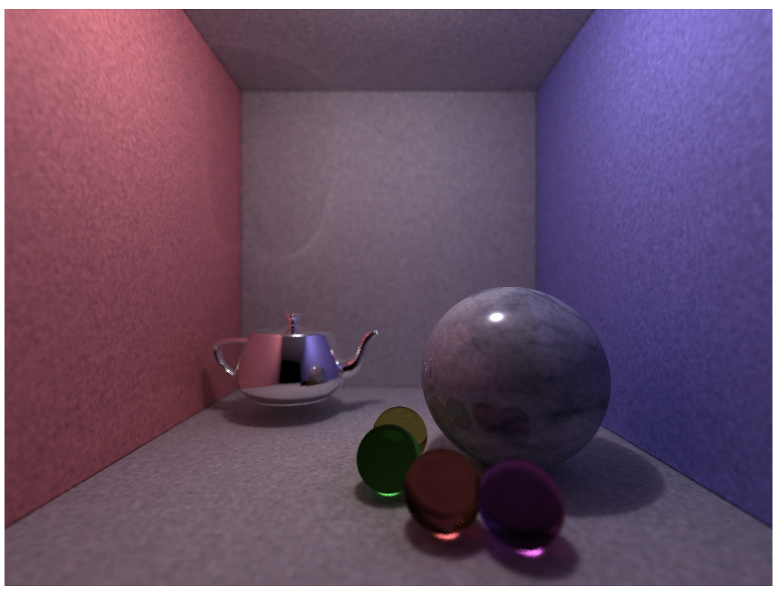

图 21: 2000 万光子

在最终程序中，每轮散播 500 万个光子，散播 5 轮后基本图像噪声不再有明显减小。

4 其他功能

4.1 贴图

贴图的本质就是建立起三维坐标 (x, y, z)-> 平面坐标 (u, v) 的映射关系，球面贴图根据球坐标系进行映射，平面贴图根据三维坐标系进行映射，Bezier 曲面贴图根据曲面控制系数进行映射即可。

最终贴图效果详见效果图。

4.2 景深

景深的本质是对相机的模拟，对光线的出发点在光圈上做随机扰动，并保证光线通过焦平面上。由于随机扰动采样，开启景深后渲染时间会大幅增加。

最终景深效果详见效果图。

4.3 渲染加速

在整个算法流程中两个地方用到了数据结构进行渲染加速。

4.3.1 AABB 包围盒

AABB 包围盒用于在 Bezier 曲面求交中实现加速，由于曲面的凸包性质，直接记录 16 个控制点的最大 x,y,z 和最小 x,y,z 六个参数，然后用 Slabs 算法求交，大大降低计算成本。

4.3.2 KD 树

KD 树是一种分割 K 维数据空间的数据结构，主要应用于多维空间关键数据的搜索，本程序中用 KD 树记录碰撞节点的信息实现加速。

加速算法详见代码。

5 实验结果

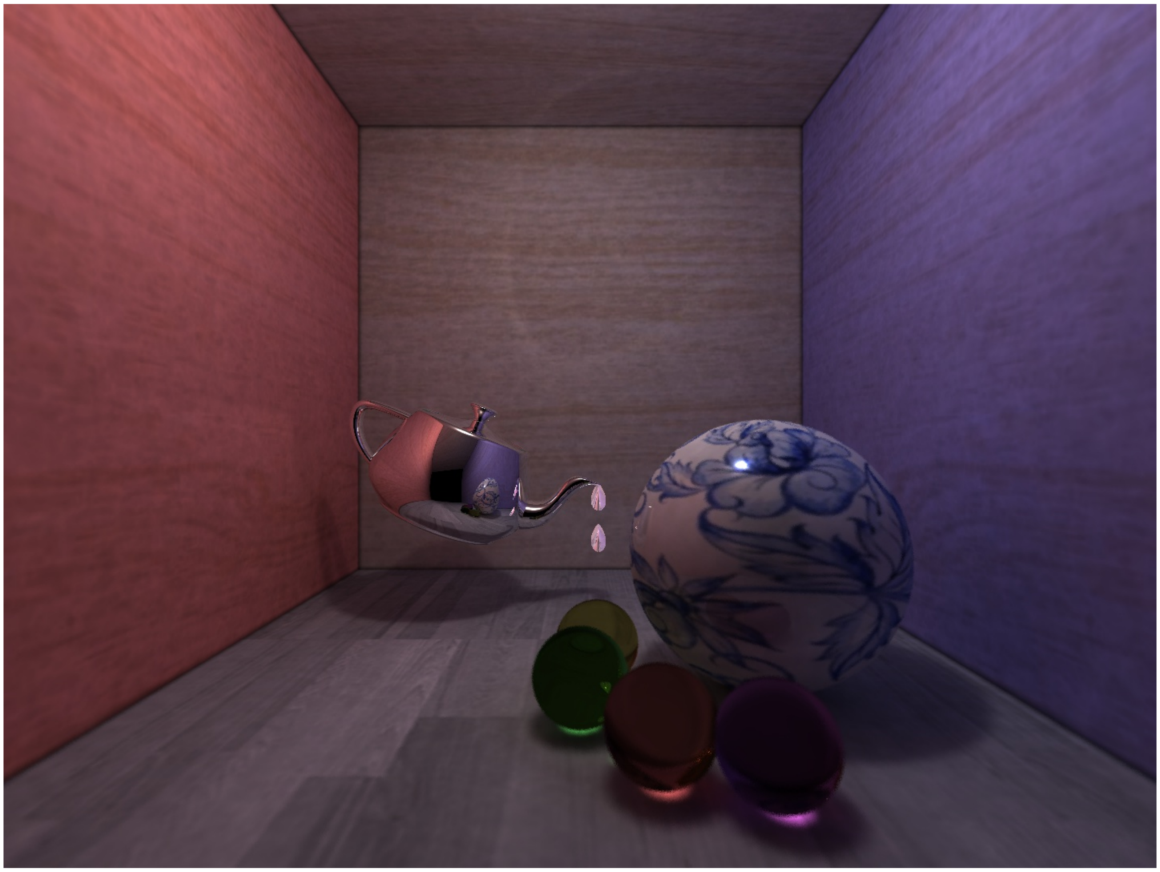

图 22: 最终效果图

从图中，我们不仅可以看出渐进式光子映射较为漂亮的渲染效果，而且可以清晰地看出景深（紫色折射玻璃球、红色折射玻璃球）、贴图（青花瓷漫反射加镜面反射大球、地板、天花板、前后左右的墙）等效果，并且由于景深本身就是多次采样，所以也有一定的抗锯齿成分（所有球边缘）。

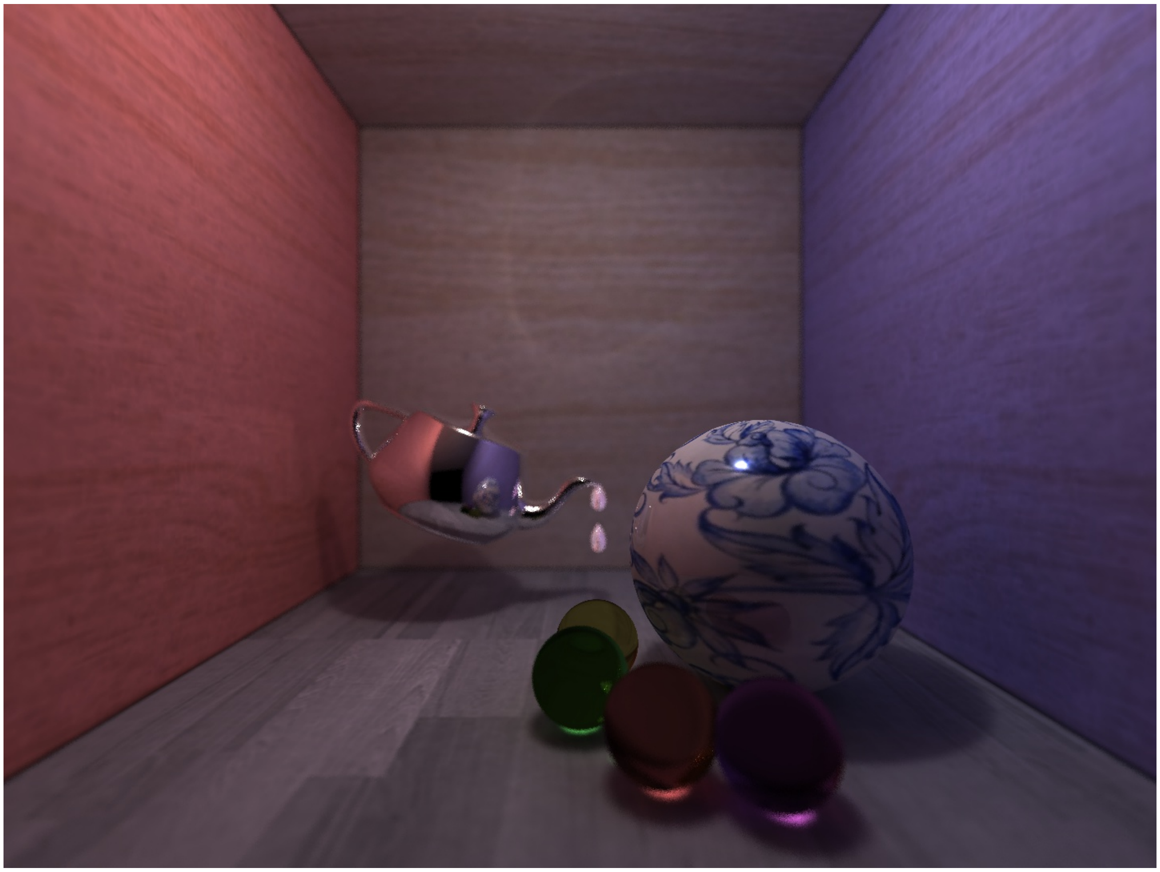

图 23: 改变了景深的效果图

6 实验心得

中小学时代玩 3D 游戏的时候，比如经典的单机游戏仙剑奇侠传，我的心中一直有一个疑问，这些场景到底是怎么渲染出来的？物体到底是怎么构成的？为什么能做出来和真实世界一致的东西？这也是我修计算机图形学的最重要的原因，感谢这次作业，让我真真正正明白了光线跟踪、路径追踪、光子映射、渐进式光子映射的原理，算是入了图形学世界的大门。

从写 Bezier 曲面求交的过程中，我补了很多数学知识，改了很多次算法，最终实现了四分求交稳定而快速的算法，这让我感受到了算法的力量。

从光线跟踪到渐进式光子映射，渲染效果有了很大的进步，但是这是时间代价换来的，光线跟踪可以在短短几秒跑出结果的图，渐进式光子映射需要几千倍的时间才能稳定，做出精美的画面，我想这也是为什么渐进式光子映射算法无法做成游戏引擎的原因——无法实现实时渲

染。

同时学习了抗锯齿、阴影、软阴影、景深、贴图、凹凸贴图等技术后，尽管我没有时间全部实现，但这些概念让我更加深入理解了游戏中的原理，

这个大作业是我本学期完成所耗时间最长的作业，在期中过后开始从光线追踪调起，调过之后先改了一部分光子映射，后来发现渐进式光子映射和光子映射实现原理较为类似，就开始彻底重构为渐进式光子映射，由于平常时间断断续续，一直到十四五周代码仍有些许 Bug，渲染效果也不是很理想。

本次大作业具有算法复杂、工程量大、Bug 难调等诸多特点，在此感谢和我一起搜集资料、研读渐进光子映射算法的同学，在此感谢认真听了我的曲面求交算法后告诉我四分区面必须用 De Casteljau 算法的同学，在此感谢看到我渲染图给我提出问题的同学，在此感谢在我迷茫时给我鼓励的同学，在此感谢每天提醒我身体最要紧的父母，在此感谢助教对我问题认真的答疑，感谢所有帮助我的人！

7 程序使用

7.1 编译方式

注意：编译本工程前需要配置 C++ opencv 和 eigen 的库。

本工程内含 Makefile 文件，在终端中输入 make 即可。

7.2 使用方式

在终端中输入./main 即开始渲染，第一步光线跟踪大约耗时 5 分钟，之后进行无限轮的渐进式光子映射过程，每一轮大约耗时 45 分钟，每轮结束都会更新 update.jpg 文件。

7.3 文件组织

- code 文件夹下为源代码、贴图文件和 makefike 文件
- picture 文件夹下为渲染结果
- obj 文件夹下为双三曲面生成的 obj 文件
- result.pdf 为说明文档

7.4 代码结构

- Vec3.h 内含三维 double 向量 Vec3 类

- Object.h 内含物品 Object 基类、球面 Sphere 类、平面 Plane 类、双三次曲面 Bezier 类、包围盒 AABB 类
- Light.h 内含光源 Light 基类和点光源 DirectionalLight 类
- Camera.h 内含相机 Camera 类
- HitPoint.h 内含光子 Photon 类和撞击点 HitPoint 类
- KdTree.h 内含 Kd 树节点 KdNode 类和 Kd 树 KdTree 类
- Scene.h 内含场景 Scene 类
- Texture.h 内含贴图 Texture 类
- PPM.h 内含渐进式光子映射 PPM 类
- teapotdata.h 内含双三曲面 Utah 壶控制点数据
- Object.cpp 内含各种物体求交算法
- KdTree.cpp 内含 Kd 树各种算法
- PPM.cpp 内含光线追踪与渐进式光子映射算法
- main.cpp 内含场景布局与程序的入口

参考文献

[1] 计算机图形学, 胡事民, 2017.

[2] 计算机图形学第二次习题课, 杨晟, 2017.

[3] 各种论文、博客、网站, 2017.