三维造型与渲染

计 72 谢兴宇 2017011326

June 2019

目录

1 场景渲染：Progressive Photon Mapping

1.1 算法简述

参考 [1] 实现，先做一遍 Ray Tracing，记录下所有的 Hit Point。再做多次 Photon Mapping，用得到的 Photon Mapping 更新 Hit Point 的光子统计半径和光能。

1.2 改进：跨代更新

在 Photon Mapping 比较小（光子比较少，分布稀疏）或随着迭代次数的增加 Hit Point 的光子统计半径已经减小到比较小的情况下，单次 Photn Mapping 统计到的光子数量极少，会以较大的概率出现出现较大的偏差。为解决这个问题，我再设置了一个阈值 P，对于每一个 Hit Point，只有其在多个 Photon Mapping 中统计到的光子数大于阈值 P 后，才会使用这些光子进行一次更新。

通过实验，最终我选取了 $P = 10$。

1.3 关键代码

```
1  //main.cpp
2  std::vector<HitPoint> hitPoints = hitPointMapping(w, h,
   ↪  samps, depth_of_field);
3  //...
4  for (int _number_of_photon_mappings = 1;
   ↪  _number_of_photon_mappings <=
   ↪  number_of_photon_mappings;
   ↪  ++_number_of_photon_mappings) {
5      PhotonMap photonMap = photonMapping(number_of_photons,
       ↪  source, Xi);
6      //...
7      if (hitPoint.m >= 10) {
8          hitPoint.r *= sqrt((hitPoint.n + alpha *
           ↪  hitPoint.m) / (hitPoint.n + hitPoint.m));
```

```
9           hitPoint.tau = (hitPoint.tau + hitPoint.tauM) *
            ↪  ((hitPoint.n + alpha * hitPoint.m) /
            ↪  (hitPoint.n + hitPoint.m));
10          hitPoint.n = hitPoint.n + alpha * hitPoint.m;
11
12          hitPoint.m = 0;
13          hitPoint.tauM = Vec(0, 0, 0);
14      }
15  }
```

1.4 效果图示

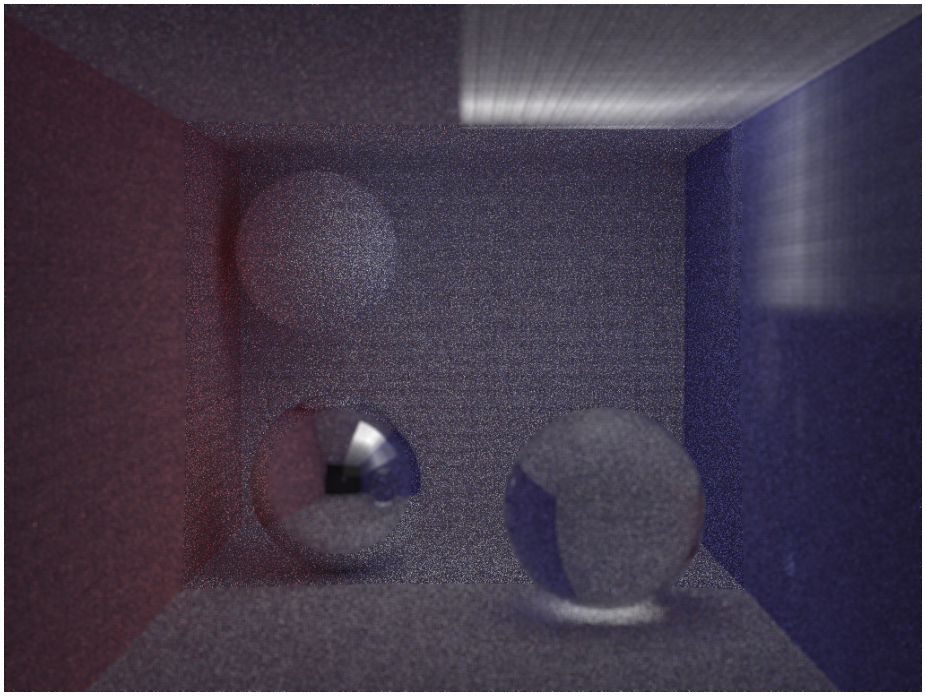

图 1: 3spheres.ppm

左下方的球的表面是纯反射镜面，右下方的球的表面是玻璃面（既会发生折射也会发生反射），左上方的球的表面是一个普通的毛面（漫反射表

面）。光源置于康奈尔盒中央。光线在左侧镜面球上镜面反射，在球的表面和右侧墙壁上均可看到光斑。右侧墙面及地板上有光线穿过玻璃球而产生的焦散效果。在墙壁和地板上可以看到三个球产生的软阴影。

2 三维曲面：旋转 Bézier 曲面

2.1 算法简述

旋转 Bézier 曲面是通过将二维平面中的二维 Bézier 曲线绕一根轴旋转而成，实现旋转 Bézier 曲面的关键问题是解决光线如何与旋转 Bézier 曲面求交的问题。我的方法参考自 [2]，通过解析几何的方法，可将光线与 n 次旋转 Bézier 曲面求交的问题化为求 $2n$ 次多项式在 $[0,1]$ 之间的根的问题。

二维 Bézier 曲线的方程为：

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

对 x, y 坐标分别考虑，即为：

$$x(t) = \sum_{i=0}^{n} x_i B_{i,n}(t)$$

$$y(t) = \sum_{i=0}^{n} y_i B_{i,n}(t)$$

若光线方向与 y 轴不垂直，则光线方程可写作：

$$x = x_0 + a(y - y_0)$$

$$z = z_0 + c(z - z_0)$$

假设旋转 Bézier 曲面的旋转轴为 $x = x_1, z = z_1$，则光线与旋转 Bézier 曲面的交点应满足：

$$(x_0 + a(y(t) - y_0) - x_1)^2 + (z_0 + c(y(t) - y_0) - z_1)^2 = x^2(t) \qquad (1)$$

$x(t), y(t)$ 是 t 的 n 次多项式，所以这是一个 $2n$ 次多项式的求零点问题（$t \in [0,1]$），我们可以使用牛顿迭代法来求解。

2.2 关键代码

```
//bezier.hpp
struct Bezier
{
    std::vector<std::pair<double, double>> p;
    std::vector<double> x, y;
    double x0 = 40, z0 = 90;
    int n = 3;
    //...
};
//...
bool bezier_intersect(Ray r, double &s, Vec &x, Vec &normal){
    //...
    //计算待求解零点的多项式
    for (int i = 0; i <= n; ++i)
        for (int j = 0; j <= n; ++j)
            coe[i + j] += (a * a + c * c) * bezier.y[i] * bezier.y[j] - bezier.x[i] * bezier.x[j];
    for (int i = 0; i <= n; ++i)
        coe[i] += 2 * (a * (r.origin.x - bezier.x0 - a * r.origin.y) + c * (r.origin.z - bezier.z0 - c * r.origin.y)) * bezier.y[i];
    coe[0] += pow(r.origin.x - bezier.x0 - a * r.origin.y, 2) + pow(r.origin.z - bezier.z0 - c * r.origin.y, 2);
    //...
    auto newtonMethod = [f, df, coe](double t0, double &t) -> bool {
        //...
        while (--restDepth && fabs(f(t1)) > eps && fabs(t0 - t1) > eps)
```

```
24          {
25              t0 = t1;
26              t1 = t1 - f(t1) / df(t1);
27          }
28          //...
29      };
30      //...
31  }
```

2.3 效果图示

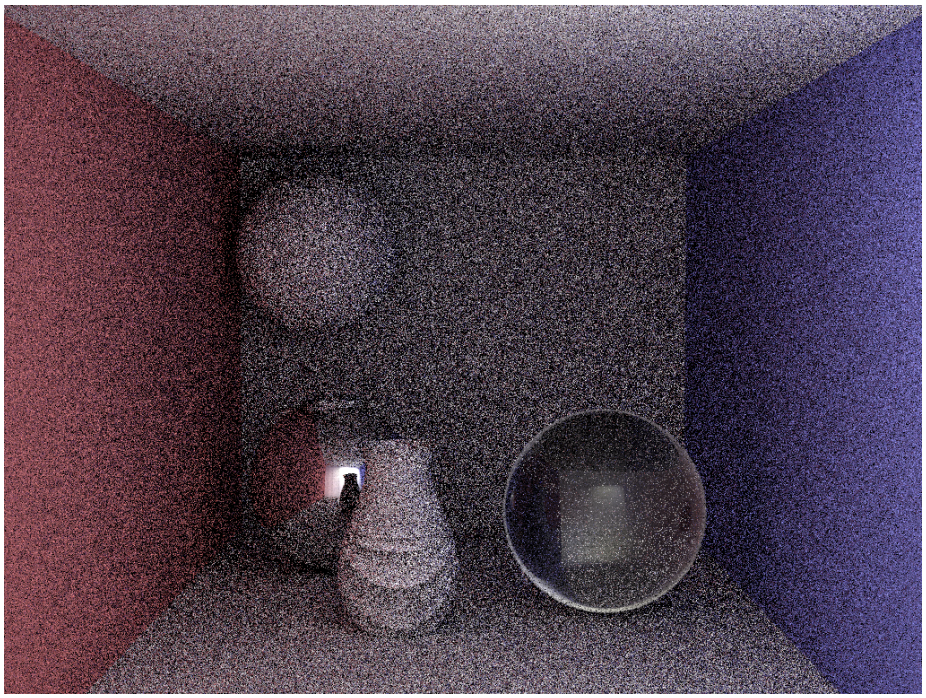

图 2: bezier.ppm

中间放置了一个 3 级旋转 Bézier 曲面，光源位于视点附近。

3 渲染加速：均匀立方体切分

3.1 算法简述

PPM 算法需要在 hit point 周围寻找 photon，或是在 photon 周围寻找 hit point。我的实现是选择组织 photon map，枚举 hit point，查询 hit point 周围的 photon。(虽然写这份报告时才意识到，组织 hit points，枚举 photon 应当是效率更高的选择)

这里我没有使用传统的 kd-tree, oc-tree 等数据结构来组织 photon map，而是将有限空间切分为 $n^{1/3} * n^{1/3} * n^{1/3}$ 个长方体，每个长方体中存储落在这个长方体中的所有光子。枚举 hit point 时，找到所有与这个 hit point 的统计球相交的长方体，检验其中每一个光子落在该统计球中。假设光子是在有限空间中均匀随机分布的，那么这样找到每一个合法光子的期望时间复杂度是 $O(1)$。

由于光子并非在三维空间中均匀随机分布（事实上，光子只能分布在物体的漫反射表面上），我采取了另一种实现。切分长方体时，用三个递增数列 $\{x_i\}, \{y_i\}, \{z_i\}$ 来切分三维空间，并保证 $x \in [x_i, x_{i+1}), y \in [y_i, y_{i+1}), z \in [z_i, z_{i+1})$ 的光子均有 $O(n^{1/3})$ 个。此时，查询与统计球相交的长方体时，需要在每一维坐标上二分。

3.2 关键代码

```
1  //photonmap.hpp
2  PhotonMap photonMapping(int number_of_photons, Vec source,
   ↪  unsigned short *Xi) {
3      //...
4      int V = photons.size();
5      int block_size = pow(V, 2.0 / 3);
6      //...
7      //划分 x 轴，并求出每一个光子在 x 轴长方体坐标
8      std::sort(photon_id.begin(), photon_id.end(),
       ↪  [&photons](int u, int v) { return photons[u].pos.x
       ↪  < photons[v].pos.x; });
9      for (int i = 0; i < V; ++i)
```

```
    {
        if (i % block_size == 0)
            photonMap.x.push_back(
            ↪ photons[photon_id[i]].pos.x);
        photon_i[photon_id[i]] = photonMap.x.size() - 1;
    }
    //...
    //存下每一个长方体中有哪些光子
    for (int i = 0; i < photons.size(); ++i)
        photonMap.photonList[photon_i[i]][photon_j[i]][
        ↪ photon_k[i]].push_back(photons[i]);
}

//main.cpp
for (auto &hitPoint : hitPoints) {
    //...
    // 求出与统计球相交的 x 轴长方体坐标
    int xl =
    ↪ std::max((int)(std::upper_bound(photonMap.x.begin(),
    ↪ photonMap.x.end(), hitPoint.pos.x - hitPoint.r) -
    ↪ photonMap.x.begin() - 1), 0);
    int xr = std::lower_bound(photonMap.x.begin(),
    ↪ photonMap.x.end(), hitPoint.pos.x + hitPoint.r) -
    ↪ photonMap.x.begin();
    // 枚举相交的长方体中的每一个光子，通过距离来判断是否在统
    ↪ 计球中
    for (int i = xl; i < xr; ++i)
        for (int j = yl; j < yr; ++j)
            for (int k = zl; k < zr; ++k)
                for (Photon photon :
                ↪ photonMap.photonList.at(i).at(j).at(k))
                    if (distance(hitPoint.pos, photon.pos)
                    ↪ < hitPoint.r) {
```

```
                    //...
                }
}
```

4 渲染特效：景深

4.1 算法简述

对于每一个像素，我将其分为 4 个子像素，在每个子像素中发出一条射线来采样。首先，我求出了每个像素看到的东西的平均距离。将所有像素按这个平均距离排序，对处于前 1/3 的像素应用一个 5×5 的平均卷积核，对处于中间 1/3 的像素应用一个 3×3 的平均卷积核。通过这种方式近似实现了焦平面在康奈尔盒最前一面的景深效果。

4.2 关键代码

```
//main.cpp
std::sort(td, td + w * h);
//...
for (int k = 1, _k = 3; k < _k; ++k)
    for (int x = k; x < w - k; ++x)
        for (int y = k; y < h - k; ++y)
        {
            int i = (h - y - 1) * w + x;
            if (depth_of_field[i] < td[w * h * k / _k] &&
            ↪  (k == _k - 1 || depth_of_field[i] > td[w *
            ↪  h * (k + 1) / 10]))
            {
                new_c[i] = 0;
                for (int tx = x - k; tx <= x + k; ++tx)
                    for (int ty = y - k; ty <= y + k;
                    ↪  ++ty)
                    {
```

```
15                          int j = (h - ty - 1) * w + tx;
16                          new_c[i] = new_c[i] + c[j];
17                      }
18                  new_c[i] = new_c[i] * (1.0 / (2 * k + 1) /
                    ↪  (2 * k + 1));
19              }
20          }
```

4.3 效果图示

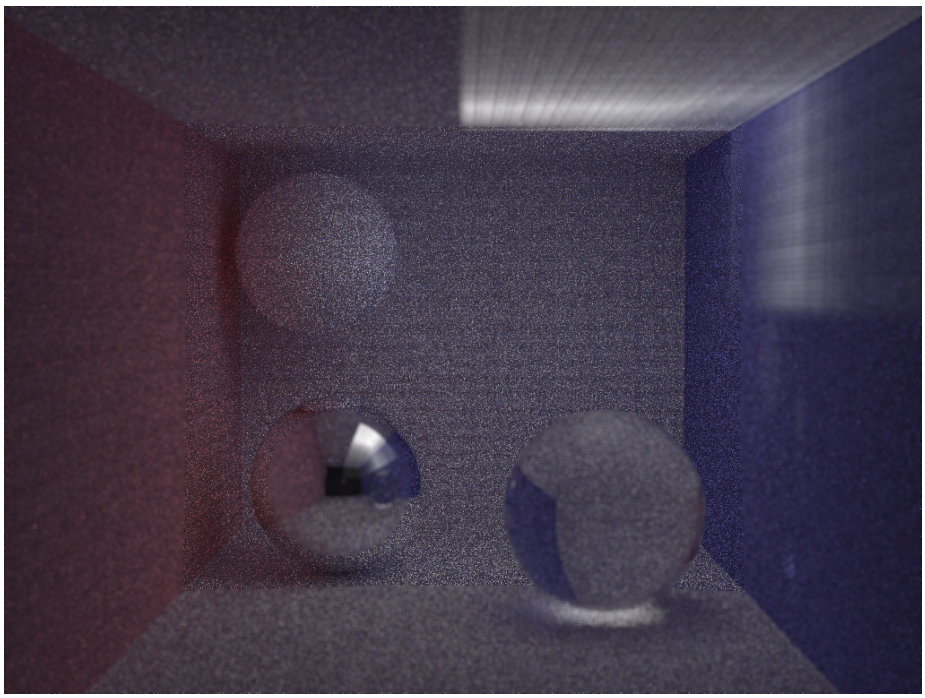

图 3: 3spheres.ppm

通过左右两侧墙壁与地板和天花板的夹线，可以看到随远及近夹线逐渐模糊，前面的两个球也要比后面的一个球要模糊，这便是景深效果。

参考文献

[1] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. In *ACM Transactions on Graphics (TOG)*, volume 27, page 130. ACM, 2008.

[2] Jiayi Weng. `https://github.com/Trinkle23897/Computational-Graphics-THU-2018`, 2018.