

真实感图像渲染报告

王征翊 2017011302

2019-6

目录 **Contents**

- 1 代码整体框架
- 2 功能概述
 - 2.1 渐进式光子映射
 - 2.2 随机渐进式光子映射
 - 2.3 贝塞尔旋转体
 - 2.4 景深
 - 2.5 贴图
 - 2.6 三角网格
- 3 加速
 - 3.1 openMP
 - 3.2 KDTree

1 代码整体框架

本次大作业，我渲染了一系列真实感图像。代码的整体框架为

- bezier.h 用于定义贝塞尔旋转体，并且进行求交等计算
- KDTTree.h 用于将三角网格组织为KDTTree，便于求交
- objects.h 定义了球体等常见几何体
- ppm.h 定义了渐进式光子映射的类
- ppm.cpp 增加了渐进式光子映射的类的有关函数的定义
- rand.hpp 用于产生随机数
- ray.h 定义了光线类
- scene.h 定义场景
- utils.h 一些常用的工具，如定义了M_PI，定义了全局求交的eps等
- vec.h 定义了三维向量，包括但不限于点积，叉积等功能。
- texture.h 定义了纹理贴图等有关内容
- stb/stb_image.h github上的库，用于处理纹理贴图中的图片读入。
- main.cpp 主函数

全部代码除了stb_image.h为外部的库以外，其余均为手动实现。

2 功能概述

2.1 渐进式光子映射

代码整体采用了渐进式光子映射的框架进行渲染。实现基于SIGGRAPH Asia 2008 的论文*Progressive Photon Mapping*。它的基本思想是从视点通过一轮ray tracing由每个像素点产生许多的viewpoint，viewpoint经过折射与镜面反射，撞击到理想漫反射面时立刻停止。之后再从光源发射多轮光子，光子经过photon tracing的过程，每次撞击到漫反射面时不停止，而是给撞击点周围的viewpoints 积累光能。

原始论文中提供的示意图：

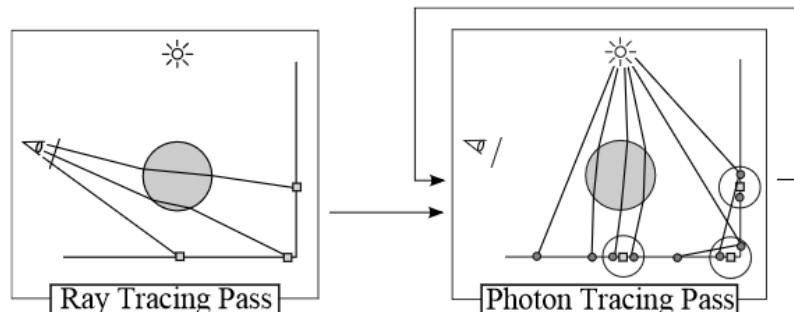


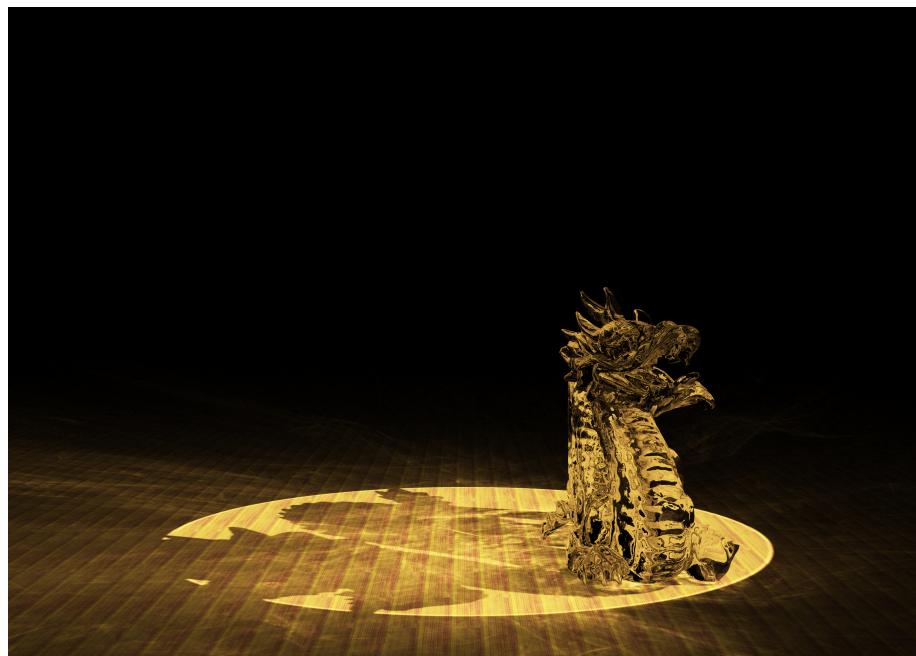
Figure 2: *Progressive photon mapping uses ray tracing in the first pass followed by one or more photon tracing passes.*

对于viewpoint半径的减小有两种方法：1. 全局固定半径R，每轮以某一比率缩小R或者不缩小。它的特点是计算快，收敛快，但是不能保证无偏。2. 按照原始论文的公式 $\hat{R}(x) = R(x)\sqrt{\frac{N(x)+\alpha M(x)}{N(x)+M(x)}}$ 进行缩小。能保证无偏，但是计算较慢。经过比较和尝试，我最终采用了第一种方法。

path tracing与ppm本质上都在进行采样，是两种不同的采样方法。在采样率足够高的情况下，两者产生的图像是完全相同的。但是path tracing只能采集大光源（小光源很难被打到，这意味着需要很高的采样率才能收敛），而ppm对于小光源也能较快收敛。因为能采集小光源，所以ppm能产生pt难以产生的焦散效果。

ppm效果图如下。可以看到地板上的焦散效果非常好。（原始图像亮度较低，此图片经过了后期调亮，原始图像见于result文件夹。）

有关代码过于庞大，在此不贴代码了，详见ppm.h/ppm.cpp。



2.2 随机渐进式光子映射

将ppm稍加修改，我们就可以得到sppm（随机渐进式光子映射）。sppm基于SIGGRAPH Asia 2009的论文*Stochastic Progressive Photon Mapping*。sppm的想法非常简单，就是把ppm中的只发射一轮viewpoint改成每轮都发射新的viewpoint，并且每轮发射viewpoint时作出微小扰动。这样增加了计算量，但是在同等迭代次数下收敛效果更好。我在读论文前就已经独立想到了sppm，事实上我的代码中的ppm均为sppm。

原始论文的示意图：

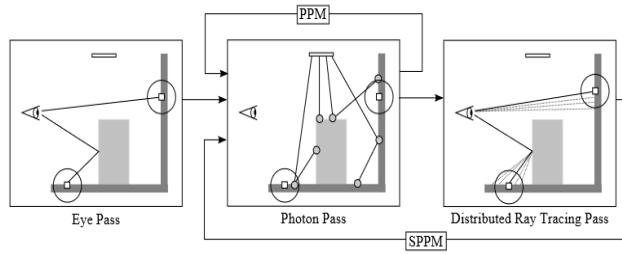


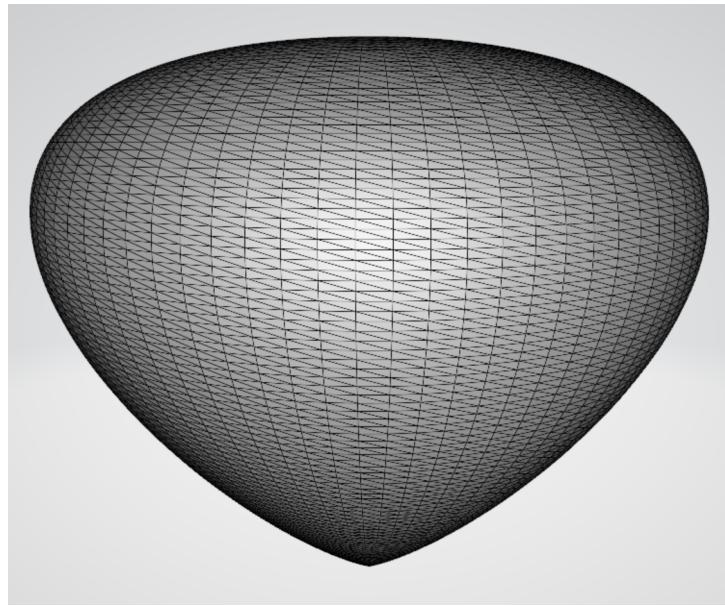
Figure 2: Difference between the algorithms of progressive photon mapping (PPM) and stochastic progressive photon mapping (SPPM). In order to compute the average radiance values, SPPM adds a new distributed ray tracing pass after each photon tracing pass. The photon tracing algorithm itself stays the same, but PPM uses a fixed set of hit points (shown as squares), whereas SPPM uses randomly generated hit points by the distributed ray tracing pass.

由于sppm本身进行了Distributed Ray Tracing Pass，我的代码中自然做到了抗锯齿和超采样。（效果见于各张图中）

软阴影取决于光源的大小，光源较小时，焦散明显；在光源较大时，自然实现了软阴影（效果见于各张图中，部分图片光源较大，软阴影效果明显）。

2.3 贝塞尔旋转体

我实现了光线与贝塞尔曲线旋转体的求交。我采用的方法是，先利用我自己写的一个独立的程序 `write_bezier_obj.cpp` 将贝塞尔旋转体生成为网格。只需要输入控制点，即可生成对应的贝塞尔旋转体的.obj文件。



图为利用该程序生成的bezier旋转体的网格。

网格由KDTree组织。光线先与网格求交，交点作为初始交点，之后再用牛顿迭代法求精细的交点。牛顿迭代法参考了课本P107页。（事实上课本P107页的算法印刷有错误，最后一个大括号右部的等式右边的加减号反了，关于dt的式子也错了）。

记 $f(\theta, t, l) = L(l) - S(\theta, t)$ ，其中 $L(l)$ 表示光线， l 为光线距离出发点的长度。 θ, t 为贝塞尔旋转体的控制参数。我们的目标是找到 θ, t, l ，使得函数 f 的值为0。

我们有 $\Delta f = \frac{dL(l)}{dl} \Delta l - \frac{\partial S(\theta, t)}{\partial \theta} \Delta \theta - \frac{\partial S(\theta, t)}{\partial t} \Delta t$ 。对该式两边同时叉积 $\frac{\partial S}{\partial \theta}$ ，再点积 $\frac{dL}{dl}$ ，我们可以得到

$$\frac{dL}{dl} \cdot \left(\frac{\partial S}{\partial \theta} \times \Delta f \right) = -D dt$$

其中, $D = \frac{dL}{dl} \cdot \left(\frac{\partial S}{\partial \theta} \times \frac{\partial S}{\partial t} \right)$ 。类似地, 我们可以得到

$$\begin{aligned}\frac{dL}{dt} \cdot \left(\frac{\partial S}{\partial t} \times \Delta f \right) &= D d\theta \\ \frac{\partial S}{\partial \theta} \cdot \left(\frac{\partial S}{\partial t} \times \Delta f \right) &= D dl\end{aligned}$$

于是我们可以得到关于 θ, t, l 的迭代方程

$$\begin{cases} l_{i+1} = l_i - \frac{\partial S}{\partial \theta} \cdot \left(\frac{\partial S}{\partial t} \times \Delta f \right) / D \\ \theta_{i+1} = \theta_i - \frac{dL}{dl} \cdot \left(\frac{\partial S}{\partial t} \times \Delta f \right) / D \\ t_{i+1} = t_i + \frac{dL}{dt} \cdot \left(\frac{\partial S}{\partial t} \times \Delta f \right) / D \end{cases}$$

于是我们便可以求解了。由于初始值选取得好, 面片的交点非常贴近旋转体的实际位置, 所以绝大多数情况都能收敛。一般来说, 迭代3次以内就可以达到我们所需要的精度。此外, 我采用了阻尼牛顿迭代进行优化——当迭代过程中发现 $|f|$ 并未缩小时, 便放弃这一迭代, 不断利用原步长的二分之一进行尝试, 直到迭代后的 $|f|$ 比之前的小。保证 $|f|$ 在迭代过程中单调递减。

迭代部分关键代码如下。

```
int cnt=0;
do
{
    cnt++;
    //...略去了部分内容, 详见bezier.h
    f=ray.o+ray.d*(vec(cos(theta)*_P_bezier[n_bezier]
[0].x,_P_bezier[n_bezier][0].y,
sin(theta)*_P_bezier[n_bezier][0].x)+position);
    if (sqrt(f.dot(f))<_eps) break;
    vec ds_dtheta=Vec(-sin(theta)*_P_bezier[n_bezier]
[0].x,0,cos(theta)*_P_bezier[n_bezier][0].x);
    vec ds_dt=-vec(cos(theta)*_Pd_bezier[n_bezier-1]
[0].x,_Pd_bezier[n_bezier-1]
[0].y,sin(theta)*_Pd_bezier[n_bezier-1][0].x);
    double D=ray.d.dot(ds_dtheta%ds_dt);
    double dl=((ds_dtheta.dot(ds_dt%f))/D);
    double dtheta=(ray.d.dot(ds_dt%f)/D);
    double dt=(ray.d.dot(ds_dtheta%f)/D);
    int cntt=0;
    while (cntt<10){
        double tmpt=t+dt;
        //...
        vec tmpf=ray.o+ray.d*(1-dl)-(vec(cos(theta-
dtheta)*_P_bezier[n_bezier][0].x,_P_bezier[n_bezier][0].y,
sin(theta-dtheta)*_P_bezier[n_bezier][0].x))-_
position;
        if (tmpf.dot(tmpf)<f.dot(f)){
            break;
        }
        dl*=0.5;
        dtheta*=0.5;
    }
}
```

```

dt*=0.5;
cntt++;
}
if(cntt==10) break;
l=d1;
theta-=dtheta;
t+=dt;
theta=std::fmod(theta,2*M_PI);
if (cnt>15){
    return 0;
}
} while (1);

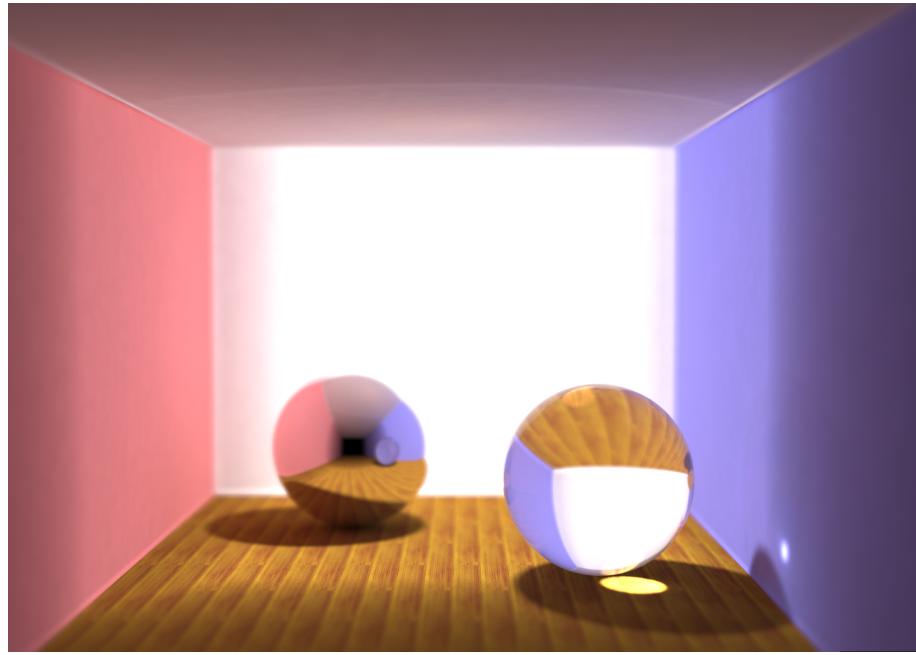
```

下图展示了利用旋转贝塞尔曲面生成的水滴。图中背景里的房屋是一整个的三角网格。



2.4 景深

景深的大致思路是，发射视点时，不再从镜头简单地发射，而是想象镜头前有一个圆形的凸透镜，光线的出发点在圆形上均匀采样。确定一个焦平面，计算原始光线打在焦平面上的位置，该位置与透镜上的采样点两点确定一条光线，即为发出的光线。



上图为景深的例子。可以看到，后面的球比前面的球模糊。

景深部分对应代码如下

```
double f1=2,aperture=1,f2=220;
for (int y=0; y<height; y++){
    fprintf(stderr, "\rCreating (%d spp)
%5.2f%%\n", samps*4,100.*y/(height-1));
    for (unsigned short x=0; x<width; x++)
    {
        for (int s=0; s<samps; s++){
            int sy=erand48(xi)<0.5?0:1;
            int sx=erand48(xi)<0.5?0:1;
            double r1=2*erand48(xi), dx=r1<1 ?
sqrt(r1)-1: 1-sqrt(2-r1);
            double r2=2*erand48(xi), dy=r2<1 ?
sqrt(r2)-1: 1-sqrt(2-r2);
            vec d = cx*( (sx+.5 + dx)/2 +
x)/width - .5) +
                    cy*( (sy+.5 + dy)/2 +
y)/height - .5) + cam.d;

            double theta=erand48(xi)*2*M_PI;
            vec point1=cam.o+cam.d*f1+
(cx*cos(theta)*aperture+cy*sin(theta)*aperture)*
(erand48(xi)-0.5);
            RayTrace(Ray(point1+(cam.o+d*f2-
point1).norm()*123,(cam.o+d*f2-
point1).norm()),0,xi,Vec(1,1,1),x,y);
        }
    }
}
```

2.5 贴图

许多.obj文件中带了贴图坐标。在光线与网格（也可以是其他物体）求交的过程中，计算出交点的贴图坐标，在贴图中提取颜色即可。对于光线与三角网格交点的贴图坐标的计算，我利用了重心坐标进行插值，让贴图变得平滑。



图中的汽车是带有贴图的三角网格，花瓶是带有贴图的贝塞尔旋转体，地板是带有贴图的半径很大的球。

相关代码如下

```
class Texture{
public:

    //略去了大量内容，详见texture.h

    Vec get_texture(Vec pos,Vec centre=Vec()){
        if (filename=="texture/wood.jpg"){
            double x,z;
            double scaling=1.2;
            if (fmod(pos.x,8*scaling)<4*scaling)

                x=fmod(pos.x,4*scaling)/(4.0*scaling),z=fmod(pos.z,30*scal
                ing)/(30.0*scaling);
            else

                x=fmod(pos.x,4*scaling)/(4.0*scaling),z=fmod(pos.z+15*scal
                ing,30.0*scaling)/(30.0*scaling);
        }
    }
}
```

```

        return get_color(x,z);
    }
    else if (filename=="texture/vase.png"){
        return get_color(pos.x,pos.y);
    }
    return get_color(pos.x,1-pos.y);
    std::cout<<"Using non-exist texture!\n";
    return Vec();
}

;

```

2.6 三角网格

我实现了光线与三角网格的求交。三角网格利用KDTree进行组织，见于3.2 KDTree。我对于三角网格的法向，利用重心坐标进行了法向插值，这使得三角网格更加平滑自然。

法向插值的核心代码：

```

void get_ord(Vec point,Vec
&my_n_ans,double&my_ord_x,double&my_ord_y,int&my_group){
    Vec E1=v[1]-v[0];
    Vec E0=v[2]-v[0];
    Vec E2=point-v[0];
    double lambda1=(E1.dot(E1)*E0.dot(E2)-
E0.dot(E1)*E1.dot(E2))/(E0.dot(E0)*E1.dot(E1)-
E0.dot(E1)*E0.dot(E1));
    double lambda2=(E0.dot(E0)*E1.dot(E2)-
E0.dot(E1)*E0.dot(E2))/(E0.dot(E0)*E1.dot(E1)-
E0.dot(E1)*E0.dot(E1));
    double lambda3=1-lambda1-lambda2;
    my_n_ans=
(vn[0]*lambda3+vn[2]*lambda1+vn[1]*lambda2).norm();
    Vec tmp=vt[0]*lambda3+vt[2]*lambda1+vt[1]*lambda2;
    my_ord_x=tmp.x,my_ord_y=tmp.y;
    my_group=group;
}

```

3 加速

3.1 openMP

openMP加速广泛分布于代码中。对于sppm，在创建视点阶段，直接并行即可；对于发射光子阶段，我在每个线程中创建了buffer_image，所有光子发射完毕后，把buffer_image叠加起来就得到了最终的图片。这样可以在不加锁的情况下快速渲染。

3.2 KDTree

有两个地方运用了KDTree进行加速。

1. 光线与三角面片进行求交，三角面片运用KDTree进行组织。
2. ppm发射光子过程中，所有视点运用KDTree进行组织。方便光子在撞击漫反射面后查询R距离内的所有视点。

有关代码如下：

三角网格中的KDTree

```
class KDTree{
public:
    static int D;

    //.....略去了大量内容，详见KDTree.h

    void _query(const Ray& ray, int rt, double&
_query_ans, Vec
&my_n_ans, double&my_ord_x, double&my_ord_y, int&my_group){
        if (!meshes[rt].intersect_box(ray)) return;
        double t=meshes[rt].intersect(ray);
        if (t!=0&&t>eps&&t<_query_ans)
_query_ans=t, meshes[rt].get_ord(ray.o+ray.d*t, my_n_ans, my_o
rd_x, my_ord_y, my_group);
        if (meshes[rt].son[0]!=-1)
_query(ray, meshes[rt].son[0], _query_ans, my_n_ans, my_ord_x, m
y_ord_y, my_group);
        if (meshes[rt].son[1]!=-1)
_query(ray, meshes[rt].son[1], _query_ans, my_n_ans, my_ord_x, m
y_ord_y, my_group);
    }
    double intersect(const Ray& ray, Vec&
my_n_ans, double&my_ord_x, double&my_ord_y, int&my_group){
        double _query_ans=INF;

        _query(ray, root, _query_ans, my_n_ans, my_ord_x, my_ord_y, my_
group);
        if (_query_ans<INF/2) return _query_ans;
        else return 0;
    }
};
```

ppm中的KDTree

```
//略去了大量内容，完整部分见于ppm.h
void build_tree(){
    printf("Building KDTree...\n");
    //if (tree!=NULL) delete[] tree;
    int v_size=vector_vp.size();
```

```
tree=new KDTreeNode[v_size+10];
for (int i=1;i<=v_size;i++){
    tree[i].view_point=vector_vp[i-1];
}
kdt_root=_build(1,v_size,0);
vector_vp.clear();
}

void _query(Vec x,int rt,std::vector<ViewPoint>
&result){
    if (!tree[rt].intersect_box(x)) return;
    if (dist2(x,tree[rt].view_point.position)
<all_radius*all_radius){
        result.push_back(tree[rt].view_point);
    }
    if (tree[rt].son[0]!=-1)
_query(x,tree[rt].son[0],result);
    if (tree[rt].son[1]!=-1)
_query(x,tree[rt].son[1],result);
}
void query(Vec x,std::vector<ViewPoint> &result){
    result.clear();
    _query(x,kdt_root,result);
}
```