实验五　光线追踪

姓名 刘子言　学号 20002462　专业班级 计 203　成绩_____

实验日期 2022.05.26　实验地点 线上　指导教师(签名) 李建华

一．实验目的

1、　学习光线追踪原理。
2、　利用蒙特卡洛路径追踪，完成对模型的渲染。

二．实验工具与设备

计算机，开发软件为 CodeBlocks （配置 OpenGL）

三、实验内容

在本科学习平台（s.ecust.edu.cn）资料栏下，下载以下文件：smallpt.tar

1、 完成光线追踪实验。

①模型背景渲染：代码中包含了常见的结构体，如 Vec, Ray, Sphere (smallPT 只包含球体渲染)，以及一些功能函数和球体的初始化。

以 Vec 的结构体的详细介绍，其目的是为 POINTS， COLORS 以及 VECTORS 提供基础的结构。其中 norm()的意义是为了求射线的方向，点积是为了求余弦角，叉积是为了求正弦角。

```
// Vec STRUCTURE ACTS AS POINTS, COLORS, VECTORS
struct Vec {
    double x, y, z;  // position, also color (r,g,b)

    Vec(double x_=0, double y_=0, double z_=0) { x=x_; y=y_; z=z_; }
    Vec operator+(const Vec &b) const  { return Vec(x+b.x,y+b.y,z+b.z); }
    Vec operator-(const Vec &b) const  { return Vec(x-b.x,y-b.y,z-b.z); }
    Vec operator*(double b) const      { return Vec(x*b,y*b,z*b); }
    Vec mult(const Vec &b) const       { return Vec(x*b.x,y*b.y,z*b.z); }
    Vec& norm()                        { return *this = *this * (1/sqrt(x*x+y*y+z*z)); }
    double dot(const Vec &b) const     { return x*b.x+y*b.y+z*b.z; }
    Vec operator%(Vec&b)   { return Vec(y*b.z-z*b.y,z*b.x-x*b.z,x*b.y-y*b.x); } // cross
};
```

②主函数首先包括设置呈现平面，其中也包括了采样率，采样率越高，图片渲染效果越好。然后主函数中设置相机位置，相机的位置和方向都是非常重要的，如果设置有误，很容易导致渲染出全黑或者部分可见的效果。最后创建图像

1、实验代码：

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include<random>

std::default_random_engine generator;
std::uniform_real_distribution<double> distr(0.0,1.0);
//自己定义随机数生成函数，随机生成 0-1 间的随机数
double erand48(unsigned short int *X)
{
    return distr(generator);
}

struct Vec          //定义颜色混合生成的结构体
{
  double x, y, z;      //  位置依旧满足 color(R,G,B)
  Vec(double x_=0, double y_=0, double z_=0){ x=x_; y=y_; z=z_; }
  Vec operator+(const Vec &b) const { return Vec(x+b.x,y+b.y,z+b.z); }
  Vec operator-(const Vec &b) const { return Vec(x-b.x,y-b.y,z-b.z); }
  Vec operator*(double b) const { return Vec(x*b,y*b,z*b); }
  Vec mult(const Vec &b) const { return Vec(x*b.x,y*b.y,z*b.z); }
  Vec& norm(){ return *this = *this * (1/sqrt(x*x+y*y+z*z)); }
  double dot(const Vec &b) const { return x*b.x+y*b.y+z*b.z; }
  Vec operator%(Vec&b){return Vec(y*b.z-z*b.y,z*b.x-x*b.z,x*b.y-y*b.x);}
};

struct Ray          //定义光线结构 Ray，包含起点 o 与方向 d
{
    Vec o, d;
    Ray(Vec o_, Vec d_) : o(o_), d(d_){}
};

enum Refl_t         //用枚举类型表示不同的反射类型（即物体表面的材质）
{
    DIFF,  //漫反射材质
    SPEC,  //镜面材质
    REFR    //玻璃材质
};

struct Sphere       //定义球体类型
{
  double rad;        //半径
```

```
  Vec p, e, c;          //位置，自发光，自身颜色
  Refl_t refl;          //反射类型
  Sphere(double rad_, Vec p_, Vec e_, Vec c_, Refl_t refl_):
  rad(rad_), p(p_), e(e_), c(c_), refl(refl_) {}
  double intersect(const Ray &r) const     //判断光线是否与自身相交
  {
    Vec op = p-r.o;
//将光线公式代入球体公式，解方程判断是否为实根且是否大于 0
    double t, eps=1e-4, b=op.dot(r.d), det=b*b-op.dot(op)+rad*rad;
//如果相交，返回交点与光线原点的距离，否则返回 0
    if (det<0) return 0; else det=sqrt(det);
    return (t=b-det)>eps ? t : ((t=b+det)>eps ? t : 0);
  }
};

Sphere spheres[] = {          //创建多个球体
//前六个，分别对应立体空间的左右、前后、下上六面，所以球的半径很大
  Sphere(1e5, Vec( 1e5+1,40.8,81.6), Vec(),Vec(.75,.25,.25),DIFF),
  Sphere(1e5, Vec(-1e5+99,40.8,81.6),Vec(),Vec(.25,.25,.75),DIFF),
  Sphere(1e5, Vec(50,40.8, 1e5),     Vec(),Vec(.75,.75,.75),DIFF),
  Sphere(1e5, Vec(50,40.8,-1e5+170), Vec(),Vec(),           DIFF),
  Sphere(1e5, Vec(50, 1e5, 81.6),    Vec(),Vec(.75,.75,.75),DIFF),
  Sphere(1e5, Vec(50,-1e5+81.6,81.6),Vec(),Vec(.75,.75,.75),DIFF),
//场景中，3 个不同材质的球
  Sphere(16.5,Vec(27,16.5,47),       Vec(),Vec(1,1,1)*.999, SPEC),    //镜面材质
  Sphere(16.5,Vec(73,16.5,78),       Vec(),Vec(1,1,1)*.999, REFR),    //玻璃材质
  Sphere(600, Vec(50,681.6-.27,81.6), Vec(12,12,12), Vec(),  DIFF)    //漫反射材质
};

  //伽马矫正：通过路线追踪算法计算得到的值是一系列无边界颜色值，我们要将其转化为
  //人眼能够感知到的亮度，将颜色值转入 0-255 之间（使用的伽马值为 2.2）
inline double clamp(double x)
{
    return x<0 ? 0 : x>1 ? 1 : x;
}
inline int toInt(double x)
{
    return int(pow(clamp(x),1/2.2)*255+.5);
}

//判断某条光线是否与场景中的球体相交
inline bool intersect(const Ray &r, double &t, int &id)
{
  double n=sizeof(spheres)/sizeof(Sphere), d, inf=t=1e20;
```

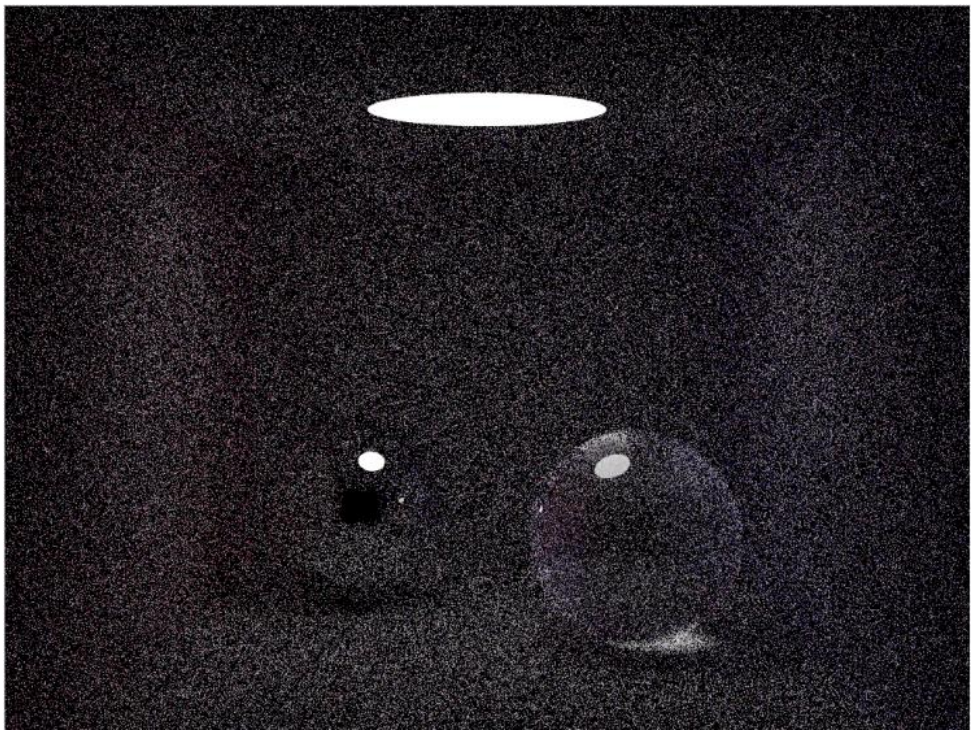
```
    for(int i=int(n);i--;)
        if((d=spheres[i].intersect(r))&&d<t){t=d;id=i;}
    return t<inf;
}

//计算每个小方格最终亮度
Vec radiance(const Ray &r, int depth, unsigned short *Xi)
{
    double t;
    int id=0;
    if (!intersect(r, t, id)) return Vec();
    const Sphere &obj = spheres[id];

    //光照计算
    Vec x=r.o+r.d*t, n=(x-obj.p).norm(), nl=n.dot(r.d)<0?n:n*-1, f=obj.c;
    double p = f.x>f.y && f.x>f.z ? f.x : f.y>f.z ? f.y : f.z;
    if (++depth>5)
        if (erand48(Xi)<p)
            f=f*(1/p);
        else return obj.e;

    if (obj.refl == DIFF)                   //漫反射材质（采用了蒙特卡洛积分方法）
    {
        double r1=2*M_PI*erand48(Xi), r2=erand48(Xi), r2s=sqrt(r2);
        Vec w=nl, u=((fabs(w.x)>.1?Vec(0,1):Vec(1))%w).norm(), v=w%u;
        Vec d = (u*cos(r1)*r2s + v*sin(r1)*r2s + w*sqrt(1-r2)).norm();
        return obj.e + f.mult(radiance(Ray(x,d),depth,Xi));
    }
    else if (obj.refl == SPEC)          //镜面反射材质（计算了反射方向上的光照贡献）
        return obj.e + f.mult(radiance(Ray(x,r.d-n*2*n.dot(r.d)),depth,Xi));

    Ray reflRay(x, r.d-n*2*n.dot(r.d));
    bool into = n.dot(nl)>0;
    double nc=1, nt=1.5, nnt=into?nc/nt:nt/nc, ddn=r.d.dot(nl), cos2t;
    if ((cos2t=1-nnt*nnt*(1-ddn*ddn))<0)
        return obj.e + f.mult(radiance(reflRay,depth,Xi));

    Vec tdir = (r.d*nnt - n*((into?1:-1)*(ddn*nnt+sqrt(cos2t)))).norm();
    double a=nt-nc, b=nt+nc, R0=a*a/(b*b), c = 1-(into?-ddn:tdir.dot(n));
    double Re=R0+(1-R0)*c*c*c*c*c,Tr=1-Re,P=.25+.5*Re,RP=Re/P,TP=Tr/(1-P);
    return obj.e + f.mult(depth>2 ? (erand48(Xi)<P ?
    radiance(reflRay,depth,Xi)*RP:radiance(Ray(x,tdir),depth,Xi)*TP):
    radiance(reflRay,depth,Xi)*Re+radiance(Ray(x,tdir),depth,Xi)*Tr);
}
```

```
int main(int argc, char *argv[])
{
    int w=1024, h=768;                              //图像大小
    int samps = argc==2 ? atoi(argv[1])/4 : 1;      //设置采样率（此处为 4spp）

//根据位置方向定义相机 cam
    Ray cam(Vec(50,52,295.6), Vec(0,-0.042612,-1).norm());

//提前计算得到单个像素在相机 x,y 轴的偏移量 cx,cy
    Vec cx=Vec(w*.5135/h), cy=(cx%cam.d).norm()*.5135;

//定义变量 r 存储计算后像素的颜色值，数组 c 存储各个像素值
    Vec r, *c=new Vec[w*h];

//对某一具体像素，将该像素划分为四个小方格，每个小方格中随机采样，samps 条路径
    for (int y=0; y<h; y++)
    {
        fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1));
        for (unsigned short x=0, Xi[3]={0,0,y*y*y}; x<w; x++)
            for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++)
                for (int sx=0; sx<2; sx++, r=Vec())
                {
                //得到采样下不同路径后，调用 radiance 方法计算得到该小方格的最终亮度
                //这里将多次采样的结果取平均，最后再将四个小方格内的颜色取一个均值
                    for (int s=0; s<samps; s++)
                    {
                        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)/w - .5) +
                              cy*( ( (sy+.5 + dy)/2 + y)/h - .5) + cam.d;
                        r = r + radiance(Ray(cam.o+d*140,d.norm()),0,Xi)*(1./samps);
                    }
                    c[i] = c[i] + Vec(clamp(r.x),clamp(r.y),clamp(r.z))*.25;
                }
    }

//得到表示图像各像素的颜色数组 c 以后，写入文件（即将结果绘制在 ppm 文件中）
  FILE *f = fopen("image.ppm", "w");
  fprintf(f, "P3\n%d %d\n%d\n", w, h, 255);
  for (int i=0; i<w*h; i++)
    fprintf(f,"%d %d %d ", toInt(c[i].x), toInt(c[i].y), toInt(c[i].z));
}
```

2、运行结果：

1）、运行上述代码，结果框中显示运行进程（采样率设置的为 4spp）：

2）、在对应文件夹中打开.ppm 文件，下图为 4spp 下的场景，可以看到由于采样率较低，渲染的效果不是很好，较为模糊：

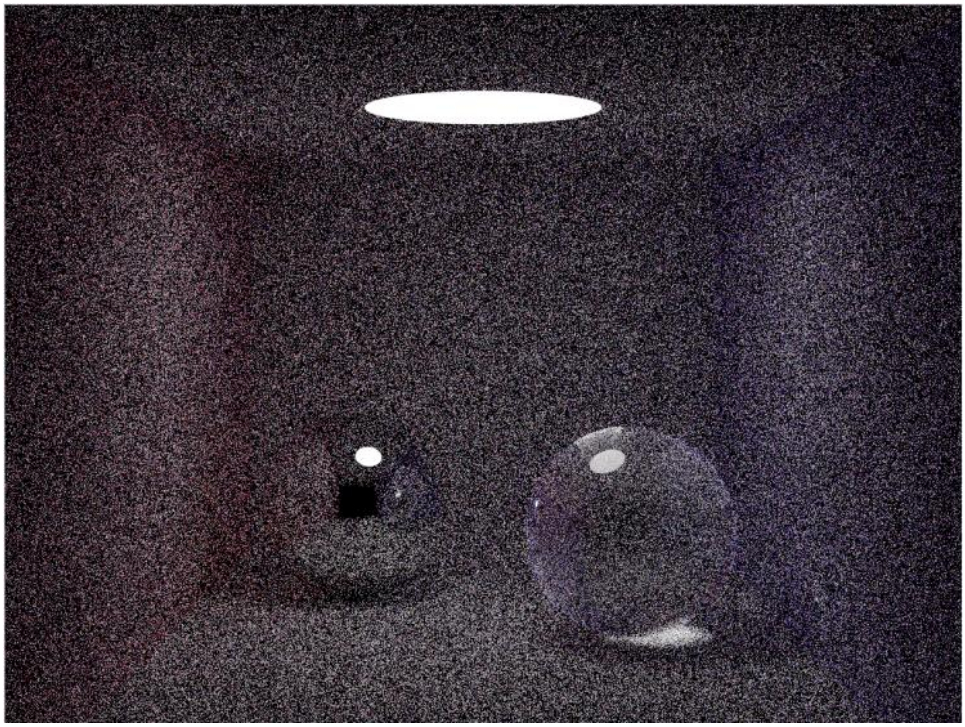


四、拓展实验

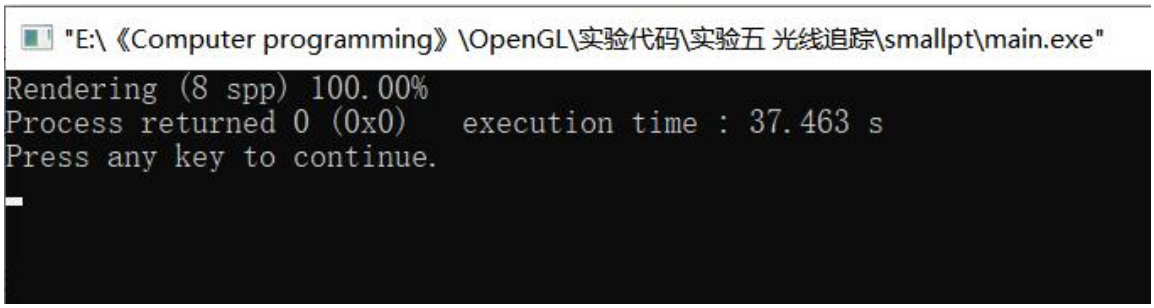
1、 举一反三，在场景中设置不同材质大小的小球实现光线追踪效果。比较不同的采样率下的渲染效果，分析采样率与渲染效果和时间成本的关系。

上述实验，我们采取用了三种不同的材质来设置球体实现光线追踪。其中墙体球为漫反射，场景中的两个小球，一个为玻璃材质，一个为镜面材质。可见不同材质的光线追踪效果也是不同的。

我们改变采样率的大小，将采样率设为 8spp，运行得到下列的渲染效果场景图：



通过与采样率 4spp 的渲染效果进行比较，可见采样率越高，渲染效果越好；但与此同时，时间成本也增加了许多：



其实通过渲染效果图，我们可以看出，即使是 8spp 下的渲染场景，画面依旧比较模糊，但要是再提高采样率进行运行，计算机运行则会超时。可见，采样率越高，光线追踪算法的计算量也会越来越大，因为光线追踪算法本质上是一个递归算法，每个像素的颜色和光强必须综合各级递归计算的结果才能获得。

五．思考题

1．光线追踪的优缺点？

答：

（1）光线追踪的优点

光线追踪基于几何光学原理，通过模拟光的传播路径，来确定反射、折射和阴影等，由于每个像素都单独计算，因此它能够很好地表现曲面细节。

光线跟踪技术为整体光照模型提供了一种简单有效的绘制手段，能够生成高度真实的图形。

（2）光线追踪的缺点

光线追踪算法的计算量非常大，需要运用其他的技术加以改进。

（3）关于光线追踪算法的计算量及改进

光线跟踪算法，本质上是一个递归算法，每个像素的颜色和光强必须综合各级递归计算的结果才能获得。

光线跟踪算法中最核心的运算是求交。早期算法中约有 95%的时间用于光线与物体表面的求交计算。所以想要优化该算法的时间，就需要设计高效率的求交算法：包围盒技术。

考虑到有时候射线与物体相距甚远，不必具体计算它们的交点，只要判断出它们不可能相交，即可以不必求交点，这就是包围盒技术或称空间细分技术，即将相邻物体用一个包围盒包起来，然后用光线与包围盒求交，若无交点，则无须对被包围的物体进行求交测试。

（4）光线追踪可能产生走样

对于屏幕上的每个像素点，都可以从视点穿过这个像素点发出这样的一条射线，利用光线跟踪算法，就可求出这个像素点的颜色。

简单的方法是使射线穿过像素点的中心，但意味着用一根无限细的线对环境中的物体进行采样，会产生走样。不过，这样做优点是射线之间是相互独立的，适合并行处理。

附件 1

1.构造函数

①Vec{}:可以用来表示位置、方向、颜色等，定义 mult、norm、dot、cross 方法，分别用于向量各项相乘、归一化、点乘、叉乘。

②Ray{}:定义光线结构 Ray，包含起点 o 与方向 d。

③Refl_t{}:定义一个枚举类型用于表示不同的反射类型

④Sphere{}:构造球体

2.场景与光线求交

①Intersect():单独定义一个方法 intersect 用于判断某条光线 r 是否与场景中的球体相交

3.伽马矫正

①Clamp():通过路径追踪算法计算得到的值为一系列没有边界的颜色值，我们需要将其转化为人眼能够感知到的亮度

4.主函数

①参数设置：定义生成的图像大小 w 和 h，设置采样率 samps，定义变量 r 存储通过计算后每一个像素最终的颜色值，数组 c 存储整张图像各个位置的像素值。

②相机设置 cam():根据位置和方向定义相机 cam，给定 FOV 为 0.5135，提前计算得到单个像素在相机的 x、y 轴的偏移量 cx、cy。

③像素样点采样:利用 OpenMP 并行遍历每一个像素，遍历方式为从下往上、从左往右。

```
for (int y = 0; y < h; y++)
{
    for (unsigned short x = 0, Xi[3] = { 0, 0, y * y * y }; x < w; x++)
    {
        ...
     }
}
```

对于某一个具体像素而言，将该像素划分为 4 个小方格，在这四个小方格中随机采样 samps 条路径。

```
for (int sy = 0, i = (h - y - 1) * w + x; sy < 2; sy++)
{
    for (int sx = 0; sx < 2; sx++, r = Vec3d())
    {
         for (int s = 0; s < samps; s++)
        {
             ...
         }
     }
}
```

④Radiance():调用 radiance 方法计算得到该小方格最终的亮度，这里是将多次采样的结果取平均，最后再将这四个小方格内的颜色取一个平均值。

⑤文件写入：得到了表示图像各像素的颜色数组 c 后，将其写入 PMM 文件