smallpt

Path tracing in 99 lines of C++
(expanded version we use has 218 lines)

Written by Kevin Beason

- Spheres only
- Output .ppm
- OpenMP for multihreading

Note: we use “explicit.cpp” version

https://www.kevinbeason.com/smallpt/explicit.cpp

```cpp
#include <math.h>   // smallpt, a Path Tracer by Kevin Beason, 2008
#include <stdlib.h> // Make : g++ -O3 -fopenmp smallpt.cpp -o smallpt
#include <stdio.h>  //        Remove "-fopenmp" for g++ version < 4.2
struct Vec {        // Usage: time ./smallpt 5000 && xv image.ppm
  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; } // cross:
  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 { Vec o, d; Ray(Vec o_, Vec d_) : o(o_), d(d_) {} };
enum Refl_t { DIFF, SPEC, REFR };  // material types, used in radiance()
struct Sphere {
  double rad;       // radius
  Vec p, e, c;      // position, emission, color
  Refl_t refl;      // reflection type (DIFFuse, SPECular, REFRactive)
  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 { // returns distance, 0 if nohit
    Vec op = p-r.o; // Solve t^2*d.d + 2*t*(o-p).d + (o-p).(o-p)-R^2 = 0
    double t, eps=1e-4, b=op.dot(r.d), det=b*b-op.dot(op)+rad*rad;
    if (det<0) return 0; else det=sqrt(det);
    return (t=b-det)>eps ? t : ((t=b+det)>eps ? t : 0);
  }
};
Sphere spheres[] = {//Scene: radius, position, emission, color, material
  Sphere(1e5, Vec( 1e5+1,40.8,81.6), Vec(),Vec(.75,.25,.25),DIFF),//Left
  Sphere(1e5, Vec(-1e5+99,40.8,81.6),Vec(),Vec(.25,.25,.75),DIFF),//Rght
  Sphere(1e5, Vec(50,40.8, 1e5),     Vec(),Vec(.75,.75,.75),DIFF),//Back
  Sphere(1e5, Vec(50,40.8,-1e5+170), Vec(),Vec(),           DIFF),//Frnt
  Sphere(1e5, Vec(50, 1e5, 81.6),    Vec(),Vec(.75,.75,.75),DIFF),//Botm
  Sphere(1e5, Vec(50,-1e5+81.6,81.6),Vec(),Vec(.75,.75,.75),DIFF),//Top
  Sphere(16.5,Vec(27,16.5,47),       Vec(),Vec(1,1,1)*.999, SPEC),//Mirr
  Sphere(16.5,Vec(73,16.5,78),       Vec(),Vec(1,1,1)*.999, REFR),//Glas
  Sphere(600, Vec(50,681.6-.27,81.6),Vec(12,12,12),  Vec(), DIFF) //Lite
};
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); }
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;
}
Vec radiance(const Ray &r, int depth, unsigned short *Xi){
  double t;                               // distance to intersection
  int id=0;                               // id of intersected object
  if (!intersect(r, t, id)) return Vec(); // if miss, return black
  const Sphere &obj = spheres[id];        // the hit object
  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; // max refl
  if (++depth>5) if (erand48(Xi)<p) f=f*(1/p); else return obj.e; //R.R.
  if (obj.refl == DIFF){                  // Ideal DIFFUSE reflection
    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)            // Ideal SPECULAR reflection
    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));     // Ideal dielectric REFRACTION
  bool into = n.dot(nl)>0;                // Ray from outside going in?
  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)    // Total internal reflection
    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 ?   // Russian roulette
    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, samps = argc==2 ? atoi(argv[1])/4 : 1; // # samples
  Ray cam(Vec(50,52,295.6), Vec(0,-0.042612,-1).norm()); // cam pos, dir
  Vec cx=Vec(w*.5135/h), cy=(cx%cam.d).norm()*.5135, r, *c=new Vec[w*h];
#pragma omp parallel for schedule(dynamic, 1) private(r)       // OpenMP
  for (int y=0; y<h; y++){                       // Loop over image rows
    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++)   // Loop cols
      for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++)     // 2x2 subpixel rows
        for (int sx=0; sx<2; sx++, r=Vec()){        // 2x2 subpixel cols
          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);
          } // Camera rays are pushed ^^^^^ forward to start in interior
          c[i] = c[i] + Vec(clamp(r.x),clamp(r.y),clamp(r.z))*.25;
        }
  }
  FILE *f = fopen("image.ppm", "w");         // Write image to PPM file.
  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));
}
```

smallpt

Vec: vector class for points, normals, colors

Ray: ray class (origin and direction)

Refl_t: surface reflection type

spheres: hard coded scene

intersect: intersect rays with spheres

radiance: solves the rendering equation

main: main loop that goes over each pixel

Preliminaries (1-14)

```
1  // smallpt, a Path Tracer by Kevin Beason, 2009
2  // Make : g++ -O3 -fopenmp explicit.cpp -o explicit
3  //        Remove "-fopenmp" for g++ version < 4.2
4  // Reformatted by David Cline for illustrative purposes
5
6  #include <math.h>
7  #include <stdlib.h>
8  #include <stdio.h>
9
10 double M_PI = 3.1415926535;
11 double M_1_PI = 1.0 / M_PI;
12 double erand48(unsigned short xsubi[3]) {
13     return (double)rand() / (double)RAND_MAX;
14 }
```

Vectors (16-28)

```
16  // Vec STRUCTURE ACTS AS POINTS, COLORS, VECTORS
17  struct Vec {
18      double x, y, z;  // position, also color (r,g,b)
19
20      Vec(double x_=0, double y_=0, double z_=0) { x=x_; y=y_; z=z_; }
21      Vec operator+(const Vec &b) const  { return Vec(x+b.x,y+b.y,z+b.z); }
22      Vec operator-(const Vec &b) const  { return Vec(x-b.x,y-b.y,z-b.z); }
23      Vec operator*(double b) const      { return Vec(x*b,y*b,z*b); }
24      Vec mult(const Vec &b) const       { return Vec(x*b.x,y*b.y,z*b.z); }
25      Vec& norm()                        { return *this = *this * (1/sqrt(x*x+y*y+z*z)); }
26      double dot(const Vec &b) const     { return x*b.x+y*b.y+z*b.z; }
27      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
28  };
29
```

Ray Structure (30-34)

Parametric form with origin and direction

$$\vec{p}(t) = \vec{o} + t\vec{d}$$

```
30  // Ray STRUCTURE
31  struct Ray {
32      Vec o, d;
33      Ray(Vec o_, Vec d_) : o(o_), d(d_) {}
34  };
35
```

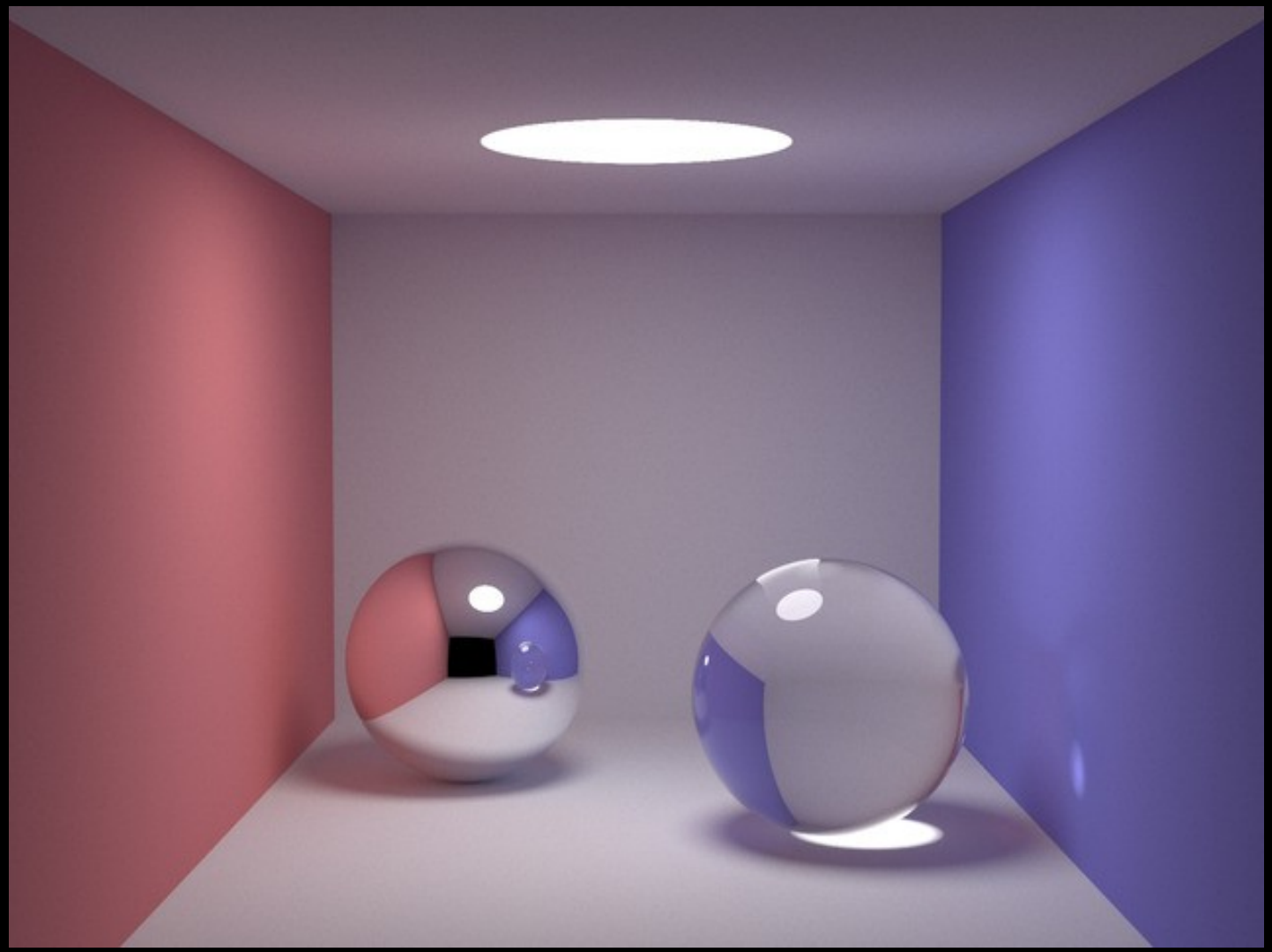
Sphere (36-60)

smallpt supports sphere objects only

Intersection code is as what we learned

```
36  // ENUM OF MATERIAL TYPES USED IN radiance FUNCTION
37  enum Refl_t { DIFF, SPEC, REFR };
38
39  // SMALLPT ONLY SUPPORTS SPHERES
40  struct Sphere {
41      double rad;       // radius
42      Vec p, e, c;      // position, emission, color
43      Refl_t refl;      // reflection type (DIFFuse, SPECular, REFRactive)
44
45      // constructor
46      Sphere(double rad_, Vec p_, Vec e_, Vec c_, Refl_t refl_):
47          rad(rad_), p(p_), e(e_), c(c_), refl(refl_) {}
48
49      // returns distance, 0 if nohit
50      double intersect(const Ray &r) const {
51          // Solve t^2*d.d + 2*t*(o-p).d + (o-p).(o-p)-R^2 = 0
52          Vec op = p-r.o;                          // p is sphere center (C)
53          double t, eps = 1e-4;                    // eps is a small fudge factor
54          double b = op.dot(r.d);                  // 1/2 b from quadradtic eq. setup
55          double det = b*b-op.dot(op)+rad*rad;     // (b^2-4ac)/4: a=1 because ray normalized
56          if (det<0) return 0;                     // ray misses sphere
57          else det = sqrt(det);
58          return (t=b-det)>eps ? t : ((t=b+det)>eps ? t : 0); // return smaller positive t
59      }
60  };
```

The Scene



Scene Description (62-76)

```
62  // HARD CODED SCENE DESCRIPTION
63  // THE SCENE DESCRIPTION CONSISTS OF A BUNCH OF SPHERES
64  // Scene: radius, position, emission, color, material
65  Sphere spheres[] = {
66      Sphere(1e5, Vec( 1e5+1,40.8,81.6), Vec(),Vec(.75,.25,.25),DIFF),//Left
67      Sphere(1e5, Vec(-1e5+99,40.8,81.6),Vec(),Vec(.25,.25,.75),DIFF),//Rght
68      Sphere(1e5, Vec(50,40.8, 1e5),     Vec(),Vec(.75,.75,.75),DIFF),//Back
69      Sphere(1e5, Vec(50,40.8,-1e5+170), Vec(),Vec(),           DIFF),//Frnt
70      Sphere(1e5, Vec(50, 1e5, 81.6),    Vec(),Vec(.75,.75,.75),DIFF),//Botm
71      Sphere(1e5, Vec(50,-1e5+81.6,81.6),Vec(),Vec(.75,.75,.75),DIFF),//Top
72      Sphere(16.5,Vec(27,16.5,47),       Vec(),Vec(1,1,1)*.999, SPEC),//Mirr
73      Sphere(16.5,Vec(73,16.5,78),       Vec(),Vec(1,1,1)*.999, REFR),//Glas
74      Sphere(1.5, Vec(50,81.6-16.5,81.6),Vec(4,4,4)*100,  Vec(), DIFF),//Lite
75  };
76  int numSpheres = sizeof(spheres)/sizeof(Sphere);
77
```

Tone Mapping & Gamma Correction (78-82)

Convert an input radiance into [0, 255]

Simple clamping for tone mapping

Gamma correction with 2.2 gamma

```
78  // CLAMP FUNCTION
79  inline double clamp(double x){ return x<0 ? 0 : x>1 ? 1 : x; }
80
81  // CONVERTS FLOATS TO INTEGERS TO BE SAVED IN PPM FILE
82  inline int toInt(double x){ return int(pow(clamp(x),1/2.2)*255+.5); }
83
```

Intersect Ray with Scene (84-97)

No acceleration data structure

Check all the intersections

```
84  // INTERSECTS RAY WITH SCENE
85  inline bool intersect(const Ray &r, double &t, int &id){
86      double n=sizeof(spheres)/sizeof(Sphere);
87      double d;
88      double inf=t=1e20;
89
90      for(int i=int(n);i--;) {
91          if((d=spheres[i].intersect(r))&&d<t) {
92              t=d;
93              id=i;
94          }
95      }
96      return t<inf;
97  }
98
```

Main Function

- Set up camera coordinates
- Initialize image array
- Parallel directive
- For each pixel
  - Do 2x2 subpixels
  - Average a number of radiance samples
- Write out image file

Initialization (176-188)

```
176  // MAIN FUNCTION, LOOPS OVER IMAGE PIXELS, CREATES IMAGE,
177  // AND SAVES IT TO A PPM FILE
178  //
179  int main(int argc, char *argv[])
180  {
181    int w=512, h=384; // image size
182    int samps = argc==2 ? atoi(argv[1])/4 : 1; // # samples (default of 1)
183    Ray cam(Vec(50,52,295.6), Vec(0,-0.042612,-1).norm()); // camera pos, dir
184    Vec cx=Vec(w*.5135/h); // x direction increment (uses implicit 0 for y, z)
185    Vec cy=(cx%cam.d).norm()*.5135; // y direction increment (note cross product)
186    Vec r; // used for colors of samples
187    Vec *c=new Vec[w*h]; // The image
188
```

Image

```
176  // MAIN FUNCTION, LOOPS OVER IMAGE PIXELS, CREATES IMAGE,
177  // AND SAVES IT TO A PPM FILE
178  //
179  int main(int argc, char *argv[])
180  {
181    int w=512, h=384; // image size
182    int samps = argc==2 ? atoi(argv[1])/4 : 1; // # samples (default of 1)
183    Ray cam(Vec(50,52,295.6), Vec(0,-0.042612,-1).norm()); // camera pos, dir
184    Vec cx=Vec(w*.5135/h); // x direction increment (uses implicit 0 for y, z)
185    Vec cy=(cx%cam.d).norm()*.5135; // y direction increment (note cross product)
186    Vec r; // used for colors of samples
187    Vec *c=new Vec[w*h]; // The image
188
```

Camera

```
176  // MAIN FUNCTION, LOOPS OVER IMAGE PIXELS, CREATES IMAGE,
177  // AND SAVES IT TO A PPM FILE
178  //
179  int main(int argc, char *argv[])
180  {
181    int w=512, h=384; // image size
182    int samps = argc==2 ? atoi(argv[1])/4 : 1; // # samples (default of 1)
183    Ray cam(Vec(50,52,295.6), Vec(0,-0.042612,-1).norm()); // camera pos, dir
184    Vec cx=Vec(w*.5135/h); // x direction increment (uses implicit 0 for y, z)
185    Vec cy=(cx%cam.d).norm()*.5135; // y direction increment (note cross product)
186    Vec r; // used for colors of samples
187    Vec *c=new Vec[w*h]; // The image
188
```

Camera Setup

Look from and gaze direction:

```
183    Ray cam(Vec(50,52,295.6), Vec(0,-0.042612,-1).norm()); // camera pos, dir
```

Horizontal camera direction:

(assumes upright camera)

(0.5135 defines field of view angle)

```
184    Vec cx=Vec(w*.5135/h); // x direction increment (uses implicit 0 for y, z)
```

Vertical camera direction:

```
185    Vec cy=(cx%cam.d).norm()*.5135; // y direction increment
```

Main Loop (189-210)

```
189  #pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP
190
191  // LOOP OVER ALL IMAGE PIXELS
192  for (int y=0; y<h; y++) { // Loop over image rows
193    fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1)); // print progress
194    unsigned short Xi[3]={0,0,y*y*y};
195    for (unsigned short x=0; x<w; x++)   // Loop columns
196
197      // FOR EACH PIXEL DO 2x2 SUBSAMPLES, AND samps SAMPLES PER SUBSAMPLE
198      for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++)     // 2x2 subpixel rows
199        for (int sx=0; sx<2; sx++, r=Vec()){        // 2x2 subpixel cols
200          for (int s=0; s<samps; s++){
201            // I BELIEVE THIS PRODUCES A TENT FILTER
202            double r1=2*erand48(Xi), dx=r1<1 ? sqrt(r1)-1: 1-sqrt(2-r1);
203            double r2=2*erand48(Xi), dy=r2<1 ? sqrt(r2)-1: 1-sqrt(2-r2);
204            Vec d = cx*( ( (sx+.5 + dx)/2 + x)/w - .5) +
205                    cy*( ( (sy+.5 + dy)/2 + y)/h - .5) + cam.d;
206            r = r + radiance(Ray(cam.o+d*140,d.norm()),0,Xi)*(1./samps);
207          } // Camera rays are pushed ^^^^^ forward to start in interior
208          c[i] = c[i] + Vec(clamp(r.x),clamp(r.y),clamp(r.z))*.25;
209        }
210  }
211
```

OpenMP Parallelization

```
189  #pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP
190
191  // LOOP OVER ALL IMAGE PIXELS
192  for (int y=0; y<h; y++) { // Loop over image rows
193    fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1)); // print progress
194    unsigned short Xi[3]={0,0,y*y*y};
195    for (unsigned short x=0; x<w; x++)   // Loop columns
196
197      // FOR EACH PIXEL DO 2x2 SUBSAMPLES, AND samps SAMPLES PER SUBSAMPLE
198      for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++)     // 2x2 subpixel rows
199        for (int sx=0; sx<2; sx++, r=Vec()){        // 2x2 subpixel cols
200          for (int s=0; s<samps; s++){
201            // I BELIEVE THIS PRODUCES A TENT FILTER
202            double r1=2*erand48(Xi), dx=r1<1 ? sqrt(r1)-1: 1-sqrt(2-r1);
203            double r2=2*erand48(Xi), dy=r2<1 ? sqrt(r2)-1: 1-sqrt(2-r2);
204            Vec d = cx*( ( (sx+.5 + dx)/2 + x)/w - .5) +
205                    cy*( ( (sy+.5 + dy)/2 + y)/h - .5) + cam.d;
206            r = r + radiance(Ray(cam.o+d*140,d.norm()),0,Xi)*(1./samps);
207          } // Camera rays are pushed ^^^^^ forward to start in interior
208          c[i] = c[i] + Vec(clamp(r.x),clamp(r.y),clamp(r.z))*.25;
209        }
210  }
211
```

States that each loop iteration should be run in its own thread.

Loop Over All Pixels

```
189  #pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP
190
191  // LOOP OVER ALL IMAGE PIXELS
192  for (int y=0; y<h; y++) { // Loop over image rows
193    fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1)); // print progress
194    unsigned short Xi[3]={0,0,y*y*y};
195    for (unsigned short x=0; x<w; x++)   // Loop columns
196
197      // FOR EACH PIXEL DO 2x2 SUBSAMPLES, AND samps SAMPLES PER SUBSAMPLE
198      for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++)     // 2x2 subpixel rows
199        for (int sx=0; sx<2; sx++, r=Vec()){        // 2x2 subpixel cols
200          for (int s=0; s<samps; s++){
201            // I BELIEVE THIS PRODUCES A TENT FILTER
202            double r1=2*erand48(Xi), dx=r1<1 ? sqrt(r1)-1: 1-sqrt(2-r1);
203            double r2=2*erand48(Xi), dy=r2<1 ? sqrt(r2)-1: 1-sqrt(2-r2);
204            Vec d = cx*( ( (sx+.5 + dx)/2 + x)/w - .5) +
205                    cy*( ( (sy+.5 + dy)/2 + y)/h - .5) + cam.d;
206            r = r + radiance(Ray(cam.o+d*140,d.norm()),0,Xi)*(1./samps);
207          } // Camera rays are pushed ^^^^^ forward to start in interior
208          c[i] = c[i] + Vec(clamp(r.x),clamp(r.y),clamp(r.z))*.25;
209        }
210  }
211
```

Subpixel Sampling

```
189  #pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP
190
191  // LOOP OVER ALL IMAGE PIXELS
192  for (int y=0; y<h; y++) { // Loop over image rows
193    fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1)); // print progress
194    unsigned short Xi[3]={0,0,y*y*y};
195    for (unsigned short x=0; x<w; x++)   // Loop columns
196
197      // FOR EACH PIXEL DO 2x2 SUBSAMPLES, AND samps SAMPLES PER SUBSAMPLE
198      for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++)     // 2x2 subpixel rows
199        for (int sx=0; sx<2; sx++, r=Vec()){        // 2x2 subpixel cols
200          for (int s=0; s<samps; s++){
201            // I BELIEVE THIS PRODUCES A TENT FILTER
202            double r1=2*erand48(Xi), dx=r1<1 ? sqrt(r1)-1: 1-sqrt(2-r1);
203            double r2=2*erand48(Xi), dy=r2<1 ? sqrt(r2)-1: 1-sqrt(2-r2);
204            Vec d = cx*( ( (sx+.5 + dx)/2 + x)/w - .5) +
205                    cy*( ( (sy+.5 + dy)/2 + y)/h - .5) + cam.d;
206            r = r + radiance(Ray(cam.o+d*140,d.norm()),0,Xi)*(1./samps);
207          } // Camera rays are pushed ^^^^^ forward to start in interior
208          c[i] = c[i] + Vec(clamp(r.x),clamp(r.y),clamp(r.z))*.25;
209        }
210  }
211
```

Pixels composed of 2x2 subpixels.
The subpixel colors will be averaged.

Subpixel Filtering

```
189  #pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP
190
191  // LOOP OVER ALL IMAGE PIXELS
192  for (int y=0; y<h; y++) { // Loop over image rows
193    fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1)); // print progress
194    unsigned short Xi[3]={0,0,y*y*y};
195    for (unsigned short x=0; x<w; x++)   // Loop columns
196
197      // FOR EACH PIXEL DO 2x2 SUBSAMPLES, AND samps SAMPLES PER SUBSAMPLE
198      for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++)     // 2x2 subpixel rows
199        for (int sx=0; sx<2; sx++, r=Vec()){        // 2x2 subpixel cols
200          for (int s=0; s<samps; s++){
201            // I BELIEVE THIS PRODUCES A TENT FILTER
202            double r1=2*erand48(Xi), dx=r1<1 ? sqrt(r1)-1: 1-sqrt(2-r1);
203            double r2=2*erand48(Xi), dy=r2<1 ? sqrt(r2)-1: 1-sqrt(2-r2);
204            Vec d = cx*( ( (sx+.5 + dx)/2 + x)/w - .5) +
205                    cy*( ( (sy+.5 + dy)/2 + y)/h - .5) + cam.d;
206            r = r + radiance(Ray(cam.o+d*140,d.norm()),0,Xi)*(1./samps);
207          } // Camera rays are pushed ^^^^^ forward to start in interior
208          c[i] = c[i] + Vec(clamp(r.x),clamp(r.y),clamp(r.z))*.25;
209        }
210  }
211
```

Sample a location within each subpixel from tent filter.

Subpixel Filtering

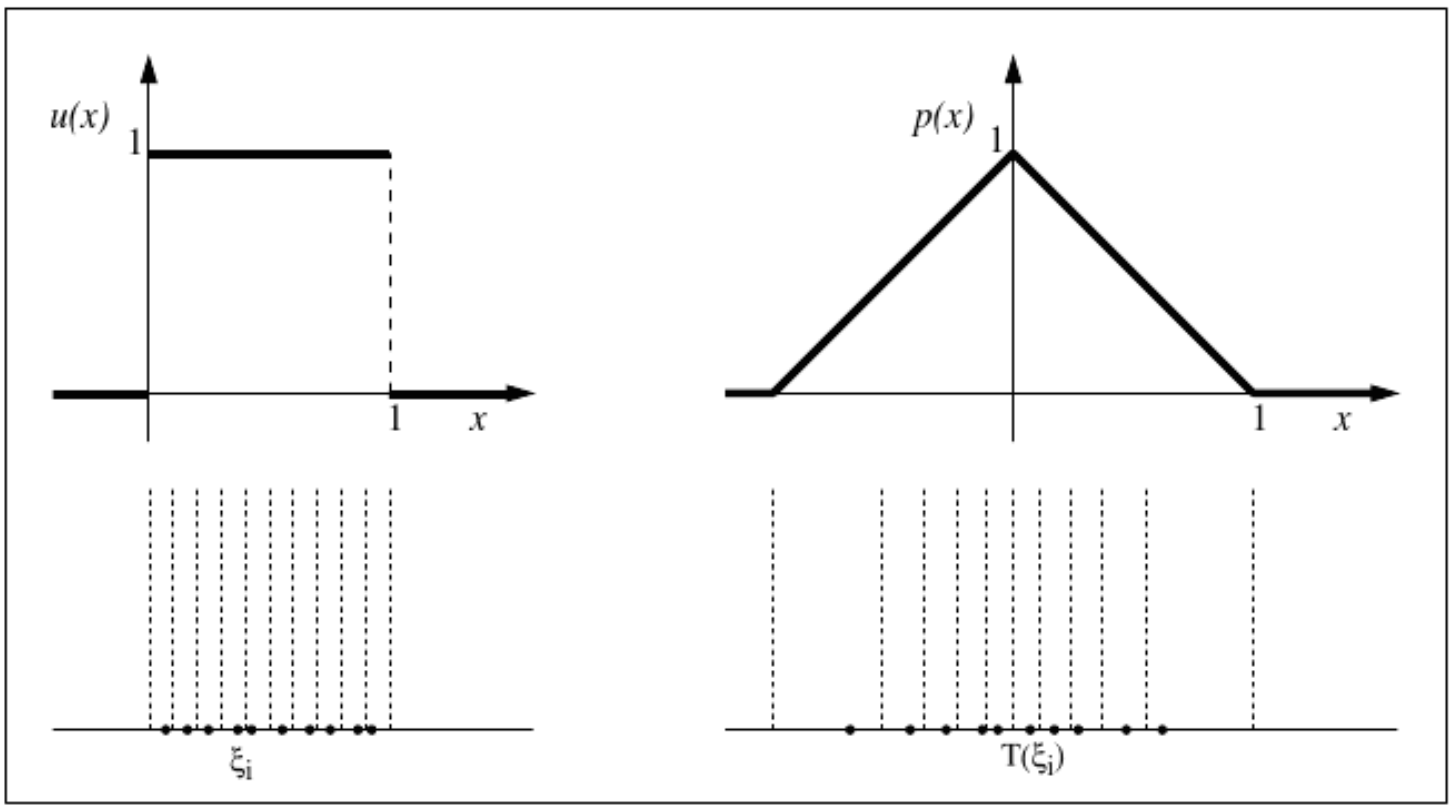


Figure 3.8. We can take a set of canonical random samples and transform them to nonuniform samples.

from “Realistic Ray Tracing” by Shirley and Morley

Subpixel Filtering

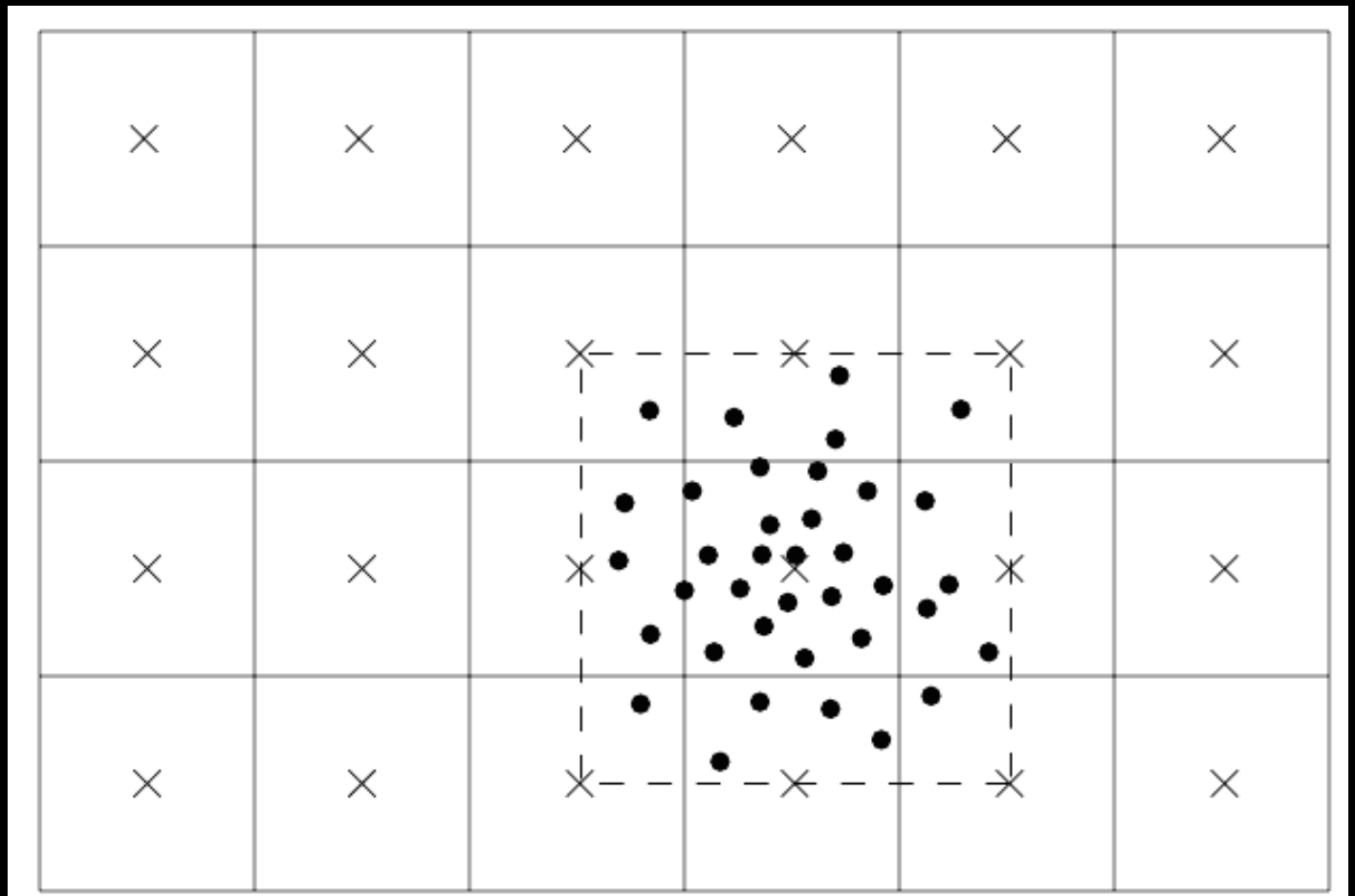


Figure 3.7. n can be used to create a estimate for pixel color.

from “Realistic Ray Tracing” by Shirley and Morley

Path Tracing

```
189  #pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP
190
191  // LOOP OVER ALL IMAGE PIXELS
192  for (int y=0; y<h; y++) { // Loop over image rows
193    fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1)); // print progress
194    unsigned short Xi[3]={0,0,y*y*y};
195    for (unsigned short x=0; x<w; x++)   // Loop columns
196
197      // FOR EACH PIXEL DO 2x2 SUBSAMPLES, AND samps SAMPLES PER SUBSAMPLE
198      for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++)     // 2x2 subpixel rows
199        for (int sx=0; sx<2; sx++, r=Vec()){        // 2x2 subpixel cols
200          for (int s=0; s<samps; s++){
201            // I BELIEVE THIS PRODUCES A TENT FILTER
202            double r1=2*erand48(Xi), dx=r1<1 ? sqrt(r1)-1: 1-sqrt(2-r1);
203            double r2=2*erand48(Xi), dy=r2<1 ? sqrt(r2)-1: 1-sqrt(2-r2);
204            Vec d = cx*( ( (sx+.5 + dx)/2 + x)/w - .5) +
205                    cy*( ( (sy+.5 + dy)/2 + y)/h - .5) + cam.d;
206            r = r + radiance(Ray(cam.o+d*140,d.norm()),0,Xi)*(1./samps);
207          } // Camera rays are pushed ^^^^^ forward to start in interior
208          c[i] = c[i] + Vec(clamp(r.x),clamp(r.y),clamp(r.z))*.25;
209        }
210  }
211
```

Note: moves the camera ray forward to be inside the scene.

Subpixel Averaging

```
189   #pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP
190
191   // LOOP OVER ALL IMAGE PIXELS
192   for (int y=0; y<h; y++) { // Loop over image rows
193     fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1)); // print progress
194     unsigned short Xi[3]={0,0,y*y*y};
195     for (unsigned short x=0; x<w; x++)   // Loop columns
196
197       // FOR EACH PIXEL DO 2x2 SUBSAMPLES, AND samps SAMPLES PER SUBSAMPLE
198       for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++)     // 2x2 subpixel rows
199         for (int sx=0; sx<2; sx++, r=Vec()){        // 2x2 subpixel cols
200           for (int s=0; s<samps; s++){
201             // I BELIEVE THIS PRODUCES A TENT FILTER
202             double r1=2*erand48(Xi), dx=r1<1 ? sqrt(r1)-1: 1-sqrt(2-r1);
203             double r2=2*erand48(Xi), dy=r2<1 ? sqrt(r2)-1: 1-sqrt(2-r2);
204             Vec d = cx*( ( (sx+.5 + dx)/2 + x)/w - .5) +
205                     cy*( ( (sy+.5 + dy)/2 + y)/h - .5) + cam.d;
206             r = r + radiance(Ray(cam.o+d*140,d.norm()),0,Xi)*(1./samps);
207           } // Camera rays are pushed ^^^^^ forward to start in interior
208           c[i] = c[i] + Vec(clamp(r.x),clamp(r.y),clamp(r.z))*.25;
209         }
210   }
211
```

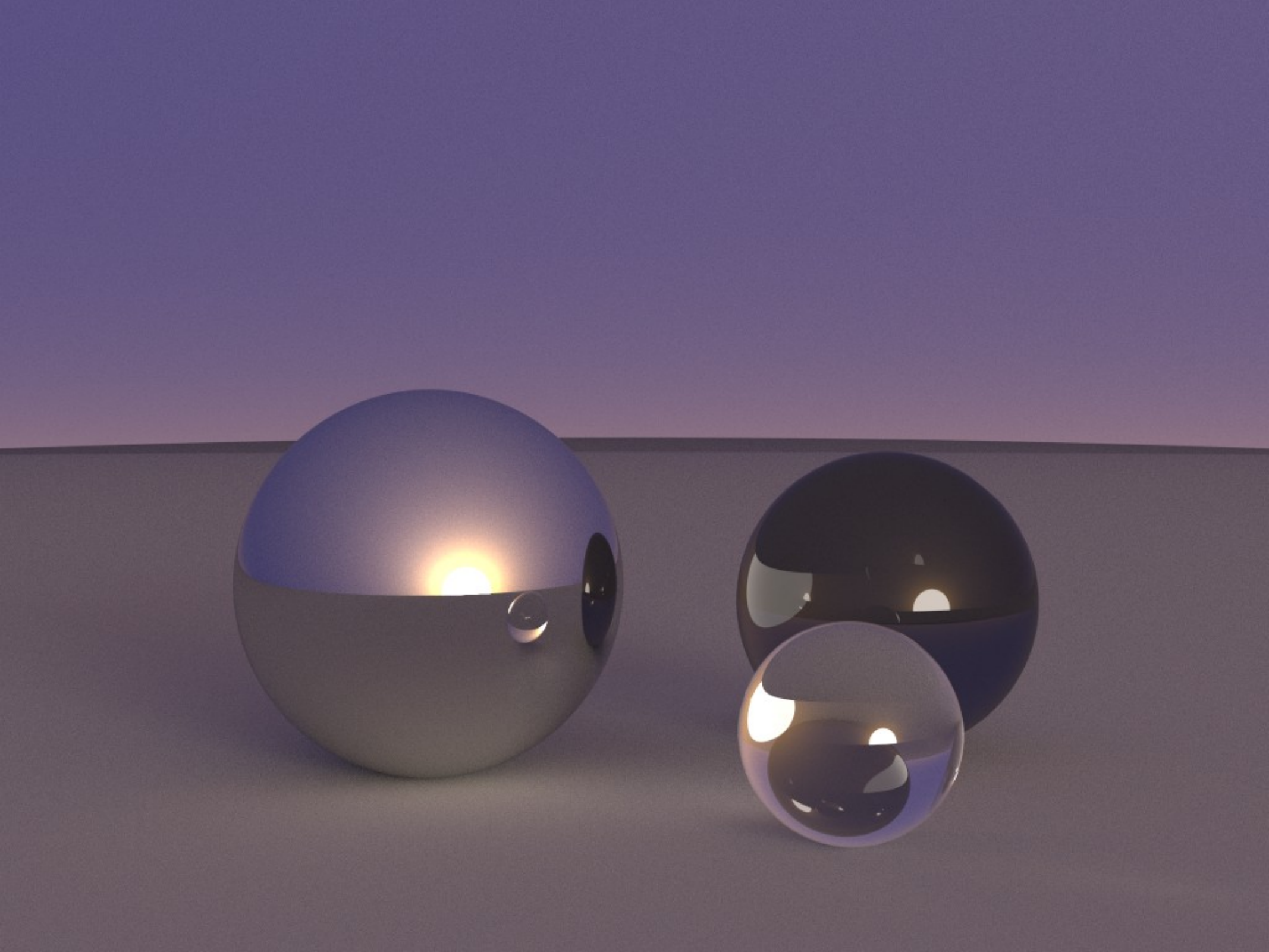
Averaging of subpixels happens after tone mapping.

Write PPM image (212-217)

```
212   // WRITE OUT THE FILE TO A PPM
213   FILE *f = fopen("image.ppm", "w");          // Write image to PPM file.
214   fprintf(f, "P3\n%d %d\n%d\n", w, h, 255);
215   for (int i=0; i<w*h; i++) {
216     fprintf(f,"%d %d %d ", toInt(c[i].x), toInt(c[i].y), toInt(c[i].z));
217   }
```

PPM Format: http://netpbm.sourceforge.net/doc/ppm.html

```
P3
# feep.ppm
4 4
15
 0  0  0    0  0  0    0  0  0   15  0 15
 0  0  0    0 15  7    0  0  0    0  0  0
 0  0  0    0  0  0    0 15  7    0  0  0
15  0 15    0  0  0    0  0  0    0  0  0
```



“radiance” function = path tracing

```
 99  // COMPUTES THE RADIANCE ESTIMATE ALONG RAY R
100  //
101  Vec radiance(const Ray &r, int depth, unsigned short *Xi,int E=1)
102  {
103    double t;                                 // distance to intersection
104    int id=0;                                 // id of intersected object
105    if (!intersect(r, t, id)) return Vec(); // if miss, return black
106    const Sphere &obj = spheres[id];          // the hit object
107
108    if (depth>10) return Vec();
109
```

return value	Vec the radiance estimate
r	the ray we are casting
depth	the ray depth
Xi	random number seed
E	whether to include emissive color

Scene Intersection

```
110   Vec x=r.o+r.d*t; // ray intersection point
111   Vec n=(x-obj.p).norm(); // sphere normal
112   Vec nl=n.dot(r.d)<0?n:n*-1; // properly oriented surface normal
113   Vec f=obj.c; // object color (BRDF modulator)
114
```

Surface properties include:

- intersection point (x)
- Normal (n)
- Oriented normal (n1)
- Object color (f)

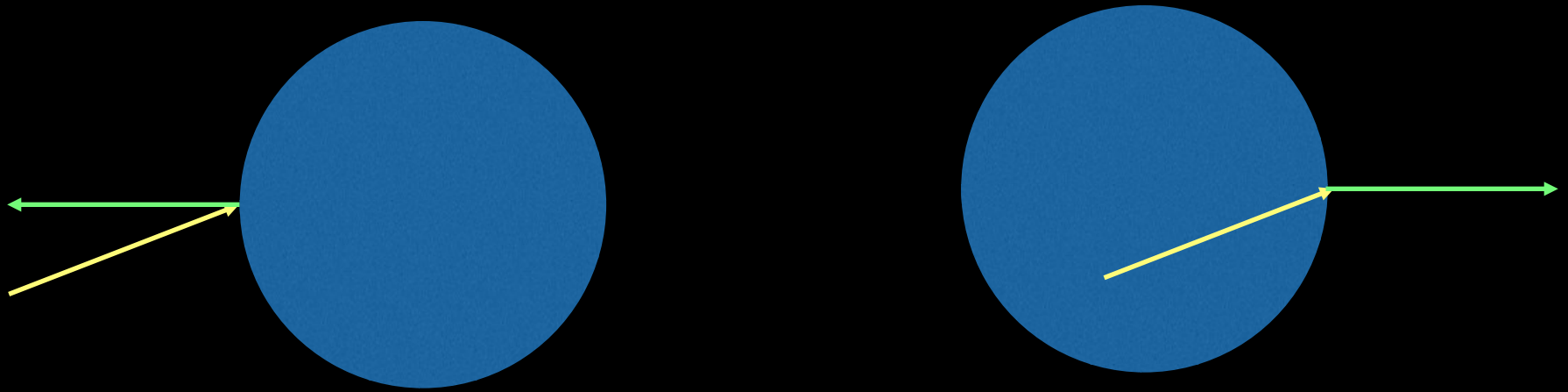
Orienting Normal

```
110   Vec x=r.o+r.d*t; // ray intersection point
111   Vec n=(x-obj.p).norm(); // sphere normal
112   Vec nl=n.dot(r.d)<0?n:n*-1; // properly oriented surface normal
113   Vec f=obj.c; // object color (BRDF modulator)
114
```

For a glass surface, need to know if the ray

Enters glass - n.dot(r.d) < 0

Exits glass - n.dot(r.d) >= 0



Russian Roulette

- Stop the recursive ray tracing randomly
  - Otherwise, we have an infinite recursion
  - Use the max(r,g,b) of the surface color
  - Applies it only after five bounces

```
115   // Use maximum reflectivity amount for Russian roulette
116   double p = f.x>f.y && f.x>f.z ? f.x : f.y>f.z ? f.y : f.z; // max refl
117   if (++depth>5||!p) if (erand48(Xi)<p) f=f*(1/p); else return obj.e*E;
118
```

$$L(y \to x) \approx L_i(y \to x) + \frac{f(\omega, y \to x) L(\omega) \cos\theta}{p(\omega)}$$

Diffuse Reflection

```
119   // IDEAL DIFFUSE REFLECTION
120   if (obj.refl == DIFF){                  // Ideal DIFFUSE reflection
121     double r1=2*M_PI*erand48(Xi); // angle around
122     double r2=erand48(Xi), r2s=sqrt(r2); // distance from center
123     Vec w = nl; // w = normal
124     Vec u = ((fabs(w.x)>.1?Vec(0,1):Vec(1))%w).norm(); // u is perpendicular to w
125     Vec v = w%u; // v is perpendicular to u and w
126     Vec d = (u*cos(r1)*r2s + v*sin(r1)*r2s + w*sqrt(1-r2)).norm();  // d is random reflection ray
```

Importance sampling according to cosine

Orthonormal basis transformation

Two random numbers (r1, r2)

Next-Event Estimation

```
128     // Loop over any lights (explicit lighting)
129     Vec e;
130     for (int i=0; i<numSpheres; i++){
131       const Sphere &s = spheres[i];
132       if (s.e.x<=0 && s.e.y<=0 && s.e.z<=0) continue; // skip non-lights
133
134       // Create random direction towards sphere using method from Realistic Ray Tracing
135       Vec sw=s.p-x, su=((fabs(sw.x)>.1?Vec(0,1):Vec(1))%sw).norm(), sv=sw%su;
136       double cos_a_max = sqrt(1-s.rad*s.rad/(x-s.p).dot(x-s.p));
137       double eps1 = erand48(Xi), eps2 = erand48(Xi);
138       double cos_a = 1-eps1+eps1*cos_a_max;
139       double sin_a = sqrt(1-cos_a*cos_a);
140       double phi = 2*M_PI*eps2;
141       Vec l = su*cos(phi)*sin_a + sv*sin(phi)*sin_a + sw*cos_a;
142       l.norm();
143
144       // Shoot shadodw ray
145       if (intersect(Ray(x,l), t, id) && id==i){  // shadow ray
146         double omega = 2*M_PI*(1-cos_a_max);
147         e = e + f.mult(s.e*l.dot(nl)*omega)*M_1_PI;  // 1/pi for brdf
148       }
149     }
```

Next-Event Estimation

```
128     // Loop over any lights (explicit lighting)
129     Vec e;
130     for (int i=0; i<numSpheres; i++){
131       const Sphere &s = spheres[i];
132       if (s.e.x<=0 && s.e.y<=0 && s.e.z<=0) continue; // skip non-lights
133
134       // Create random direction towards sphere using method from Realistic Ray Tracing
135       Vec sw=s.p-x, su=((fabs(sw.x)>.1?Vec(0,1):Vec(1))%sw).norm(), sv=sw%su;
136       double cos_a_max = sqrt(1-s.rad*s.rad/(x-s.p).dot(x-s.p));
137       double eps1 = erand48(Xi), eps2 = erand48(Xi);
138       double cos_a = 1-eps1+eps1*cos_a_max;
139       double sin_a = sqrt(1-cos_a*cos_a);
140       double phi = 2*M_PI*eps2;
141       Vec l = su*cos(phi)*sin_a + sv*sin(phi)*sin_a + sw*cos_a;
142       l.norm();
143
144       // Shoot shadodw ray
145       if (intersect(Ray(x,l), t, id) && id==i){  // shadow ray
146         double omega = 2*M_PI*(1-cos_a_max);
147         e = e + f.mult(s.e*l.dot(nl)*omega)*M_1_PI;  // 1/pi for brdf
148       }
149     }
```

Sampling a point on all the light sources

Next-Event Estimation

```
128     // Loop over any lights (explicit lighting)
129     Vec e;
130     for (int i=0; i<numSpheres; i++){
131       const Sphere &s = spheres[i];
132       if (s.e.x<=0 && s.e.y<=0 && s.e.z<=0) continue; // skip non-lights
133
134       // Create random direction towards sphere using method from Realistic Ray Tracing
135       Vec sw=s.p-x, su=((fabs(sw.x)>.1?Vec(0,1):Vec(1))%sw).norm(), sv=sw%su;
136       double cos_a_max = sqrt(1-s.rad*s.rad/(x-s.p).dot(x-s.p));
137       double eps1 = erand48(Xi), eps2 = erand48(Xi);
138       double cos_a = 1-eps1+eps1*cos_a_max;
139       double sin_a = sqrt(1-cos_a*cos_a);
140       double phi = 2*M_PI*eps2;
141       Vec l = su*cos(phi)*sin_a + sv*sin(phi)*sin_a + sw*cos_a;
142       l.norm();
143
144       // Shoot shadodw ray
145       if (intersect(Ray(x,l), t, id) && id==i){  // shadow ray
146         double omega = 2*M_PI*(1-cos_a_max);
147         e = e + f.mult(s.e*l.dot(nl)*omega)*M_1_PI;  // 1/pi for brdf
148       }
149     }
```

Check if the connection is not blocked

Diffuse Interreflecion

Recursive call with random ray direction

```
151     return obj.e*E+e+f.mult(radiance(Ray(x,d),depth,Xi,0));
```

0 at the end turns off the emissive term

Implementation specific

Can be done differently

Mirror

```
153   // IDEAL SPECULAR REFLECTION
154   } else if (obj.refl == SPEC) {              // Ideal SPECULAR reflection
155     return obj.e + f.mult(radiance(Ray(x,r.d-n*2*n.dot(r.d)),depth,Xi));
156   }
157
```

Same as what we have learned

- Compute reflected direction
- Trace a new ray recursively
- f is the surface color

Glass

```
158   // OTHERWISE WE HAVE A DIELECTRIC (GLASS) SURFACE
159   Ray reflRay(x, r.d-n*2*n.dot(r.d));     // Ideal dielectric REFLECTION
160   bool into = n.dot(nl)>0;                // Ray from outside going in?
161   double nc=1, nt=1.5, nnt=into?nc/nt:nt/nc, ddn=r.d.dot(nl), cos2t;
162
163   // IF TOTAL INTERNAL REFLECTION, REFLECT
164   if ((cos2t=1-nnt*nnt*(1-ddn*ddn))<0)    // Total internal reflection
165     return obj.e + f.mult(radiance(reflRay,depth,Xi));
166
167   // OTHERWISE, CHOOSE REFLECTION OR REFRACTION
168   Vec tdir = (r.d*nnt - n*((into?1:-1)*(ddn*nnt+sqrt(cos2t)))).norm();
169   double a=nt-nc, b=nt+nc, R0=a*a/(b*b), c = 1-(into?-ddn:tdir.dot(n));
170   double Re=R0+(1-R0)*c*c*c*c*c,Tr=1-Re,P=.25+.5*Re,RP=Re/P,TP=Tr/(1-P);
171   return obj.e + f.mult(depth>2 ? (erand48(Xi)<P ?   // Russian roulette
172     radiance(reflRay,depth,Xi)*RP:radiance(Ray(x,tdir),depth,Xi)*TP) :
173     radiance(reflRay,depth,Xi)*Re+radiance(Ray(x,tdir),depth,Xi)*Tr);
174 }
```

Reflection

```
158    // OTHERWISE WE HAVE A DIELECTRIC (GLASS) SURFACE
159    Ray reflRay(x, r.d-n*2*n.dot(r.d));     // Ideal dielectric REFLECTION
160    bool into = n.dot(nl)>0;                // Ray from outside going in?
161    double nc=1, nt=1.5, nnt=into?nc/nt:nt/nc, ddn=r.d.dot(nl), cos2t;
```

159: Glass is both reflective and refractive, so we compute the reflected ray here.

160: Determine if ray is entering or exiting glass

161: IOR for glass is 1.5.

nnt is either 1.5 or 1/1.5

Total Internal Reflection

```
163   // IF TOTAL INTERNAL REFLECTION, REFLECT
164   if ((cos2t=1-nnt*nnt*(1-ddn*ddn))<0)    // Total internal reflection
165     return obj.e + f.mult(radiance(reflRay,depth,Xi));
```

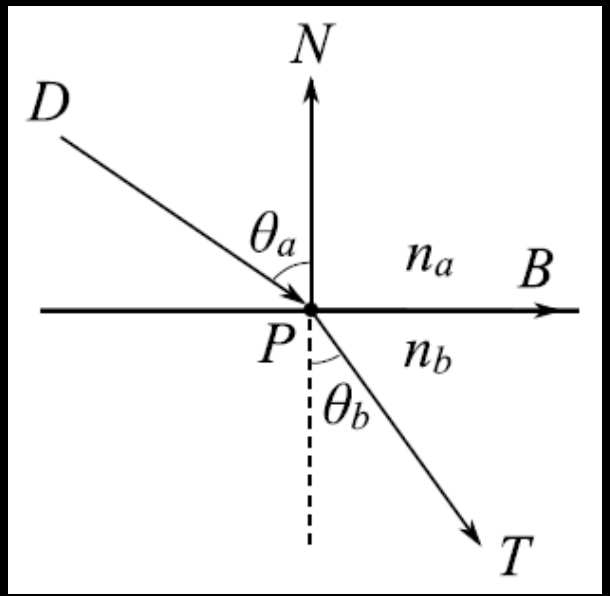
See why and when it happens in the lecture slides for shading models

If it happens, trace reflected ray only

Refraction

```
167   // OTHERWISE, CHOOSE REFLECTION OR REFRACTION
168   Vec tdir = (r.d*nnt - n*((into?1:-1)*(ddn*nnt+sqrt(cos2t)))).norm();
169   double a=nt-nc, b=nt+nc, R0=a*a/(b*b), c = 1-(into?-ddn:tdir.dot(n));
170   double Re=R0+(1-R0)*c*c*c*c*c,Tr=1-Re,P=.25+.5*Re,RP=Re/P,TP=Tr/(1-P);
171   return obj.e + f.mult(depth>2 ? (erand48(Xi)<P ?   // Russian roulette
172     radiance(reflRay,depth,Xi)*RP:radiance(Ray(x,tdir),depth,Xi)*TP) :
173     radiance(reflRay,depth,Xi)*Re+radiance(Ray(x,tdir),depth,Xi)*Tr);
```

tdir is the refracted direction



Approximate Fresnel

```
167   // OTHERWISE, CHOOSE REFLECTION OR REFRACTION
168   Vec tdir = (r.d*nnt - n*((into?1:-1)*(ddn*nnt+sqrt(cos2t)))).norm();
169   double a=nt-nc, b=nt+nc, R0=a*a/(b*b), c = 1-(into?-ddn:tdir.dot(n));
170   double Re=R0+(1-R0)*c*c*c*c*c,Tr=1-Re,P=.25+.5*Re,RP=Re/P,TP=Tr/(1-P);
171   return obj.e + f.mult(depth>2 ? (erand48(Xi)<P ?   // Russian roulette
172     radiance(reflRay,depth,Xi)*RP:radiance(Ray(x,tdir),depth,Xi)*TP) :
173     radiance(reflRay,depth,Xi)*Re+radiance(Ray(x,tdir),depth,Xi)*Tr);
```

Approximation proposed by Schlick

R0 = reflectance at normal incidence

c = 1- cos(theta)

Re = fresnel reflectance

Works fine, if IOR is not too close to 1.0
(in fact, not a good approach so don’t do it)

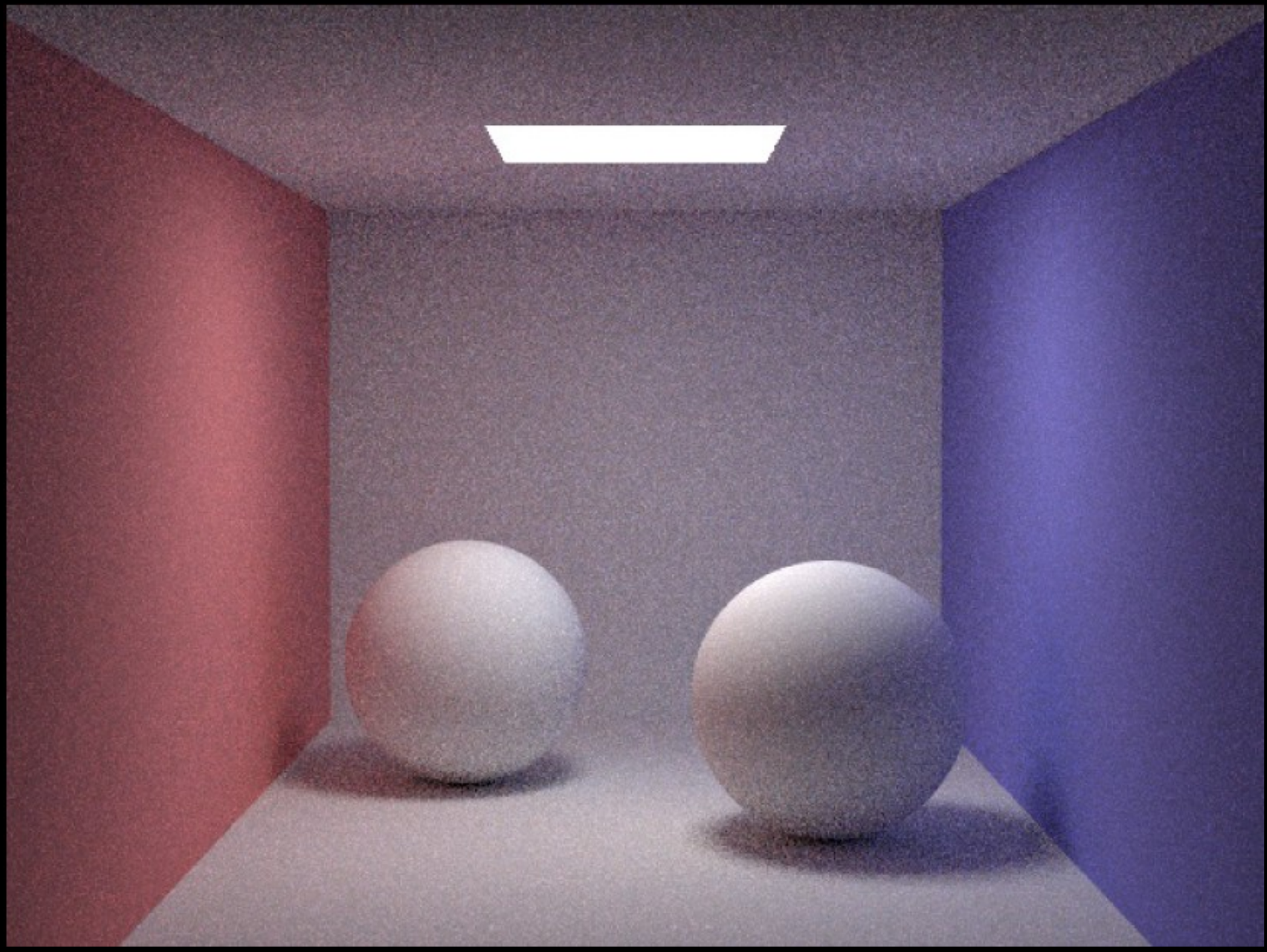
Reflect or Refract

```
167   // OTHERWISE, CHOOSE REFLECTION OR REFRACTION
168   Vec tdir = (r.d*nnt - n*((into?1:-1)*(ddn*nnt+sqrt(cos2t)))).norm();
169   double a=nt-nc, b=nt+nc, R0=a*a/(b*b), c = 1-(into?-ddn:tdir.dot(n));
170   double Re=R0+(1-R0)*c*c*c*c*c,Tr=1-Re,P=.25+.5*Re,RP=Re/P,TP=Tr/(1-P);
171   return obj.e + f.mult(depth>2 ? (erand48(Xi)<P ?   // Russian roulette
172     radiance(reflRay,depth,Xi)*RP:radiance(Ray(x,tdir),depth,Xi)*TP) :
173     radiance(reflRay,depth,Xi)*Re+radiance(Ray(x,tdir),depth,Xi)*Tr);
```

P = probability of reflection

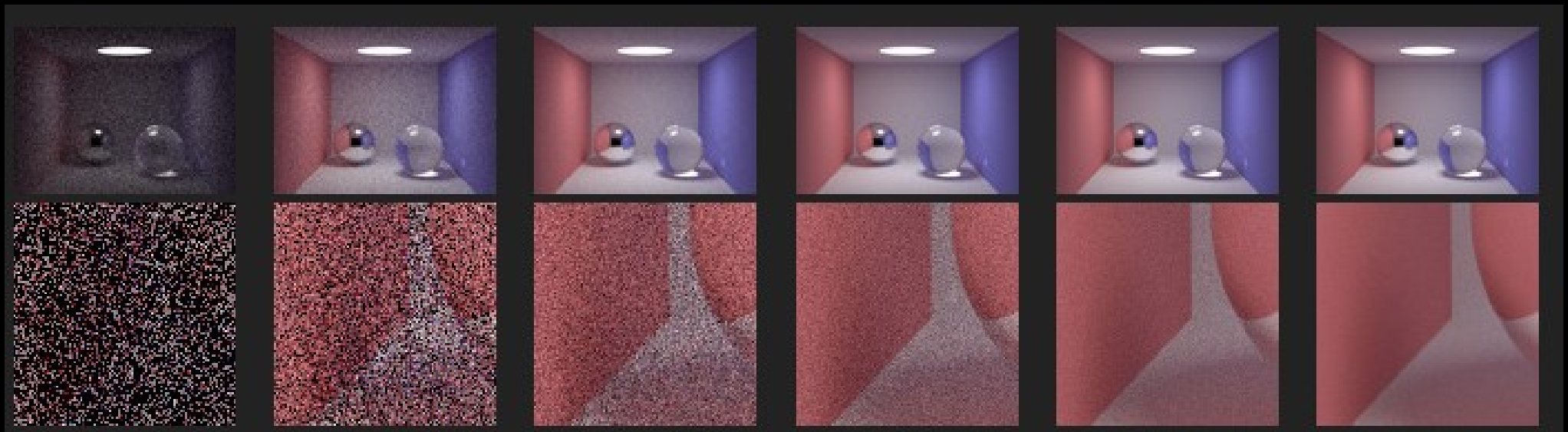
Finally, make 1 or 2 recursive calls

- Make 2 if depth is <= 2
- Make 1 randomly if depth > 2
- Avoids exponential number of rays



10 paths/pixel

Convergence



8 spp	40 spp	200 spp	1000 spp	5000 spp	25000 spp
13 sec	63 sec	5 min	25 min	124 min	10.3 hrs

Timings and resulting images for different numbers of samples per pixel (spp) on a 2.4 GHz Intel Core 2 Quad CPU using 4 threads.

http://kevinbeason.com/smallpt/

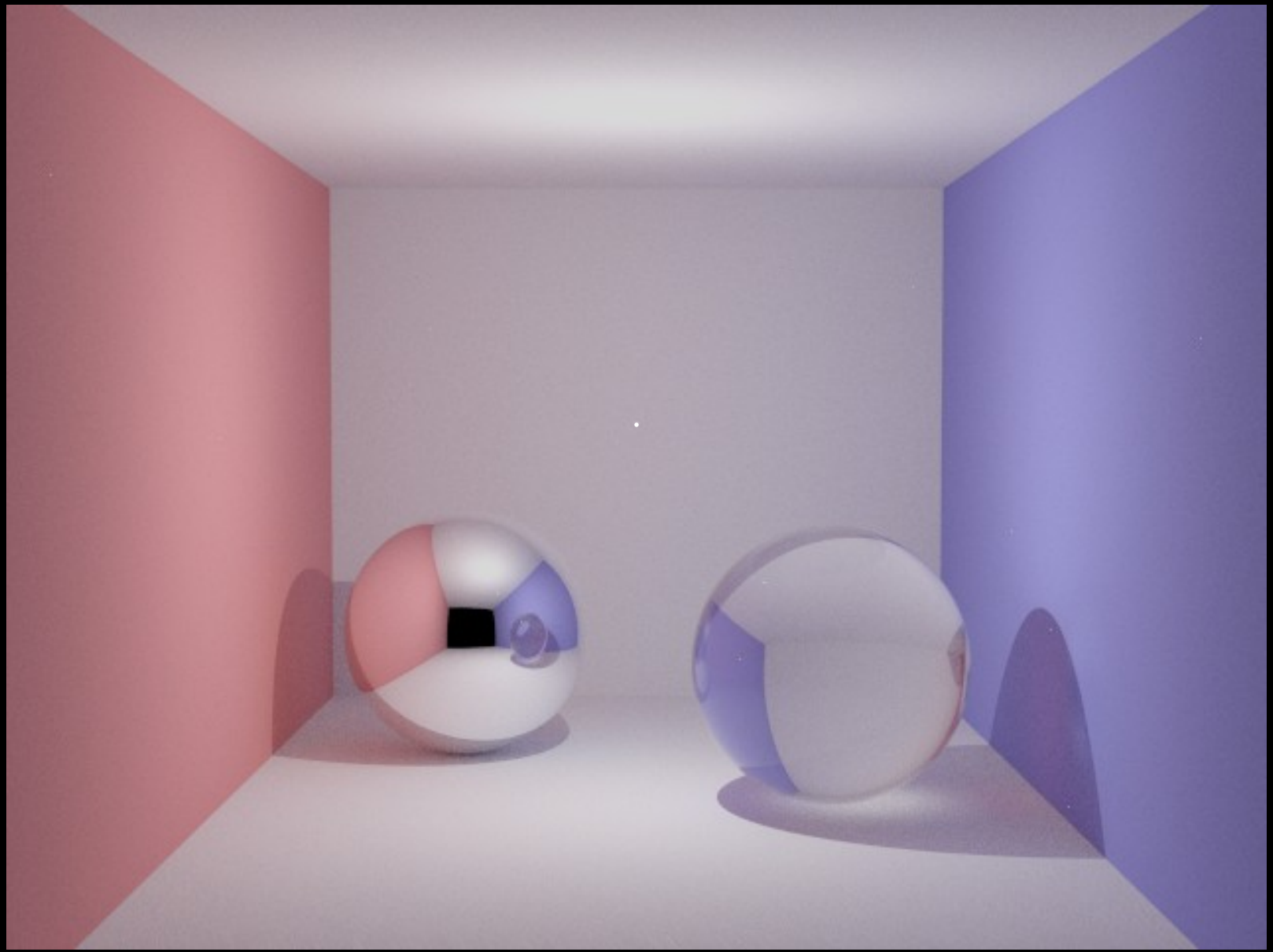
smallppm

- Photon density estimation in 128 lines of C++ (extended version has 335 lines)
- Modification of smallpt by Kevin Beason
- Majority of code is similar to smallpt
  - Spheres only
  - Output .ppm
  - OpenMP for multihreading

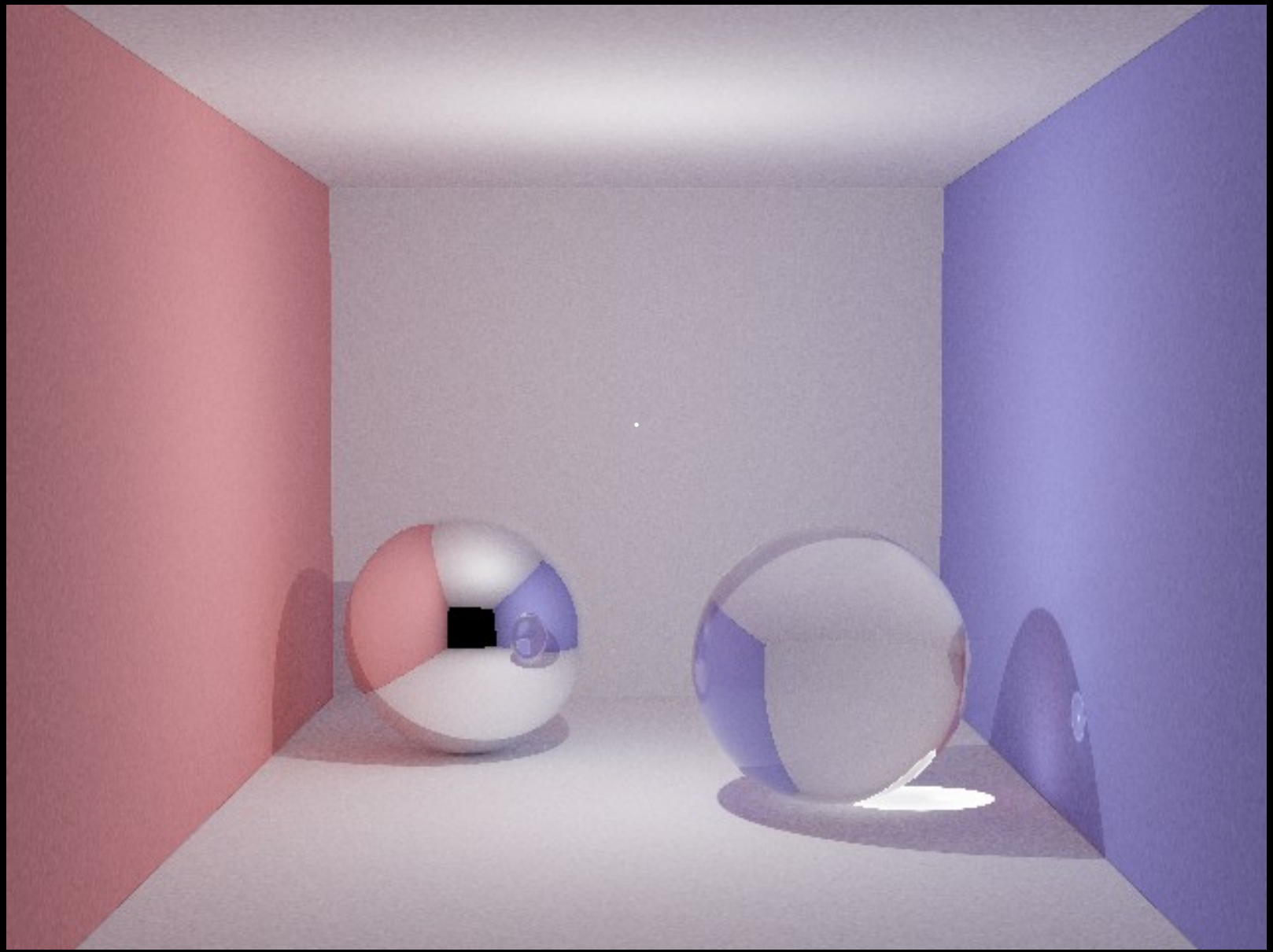
https://cs.uwaterloo.ca/~thachisu/smallppm_exp.cpp

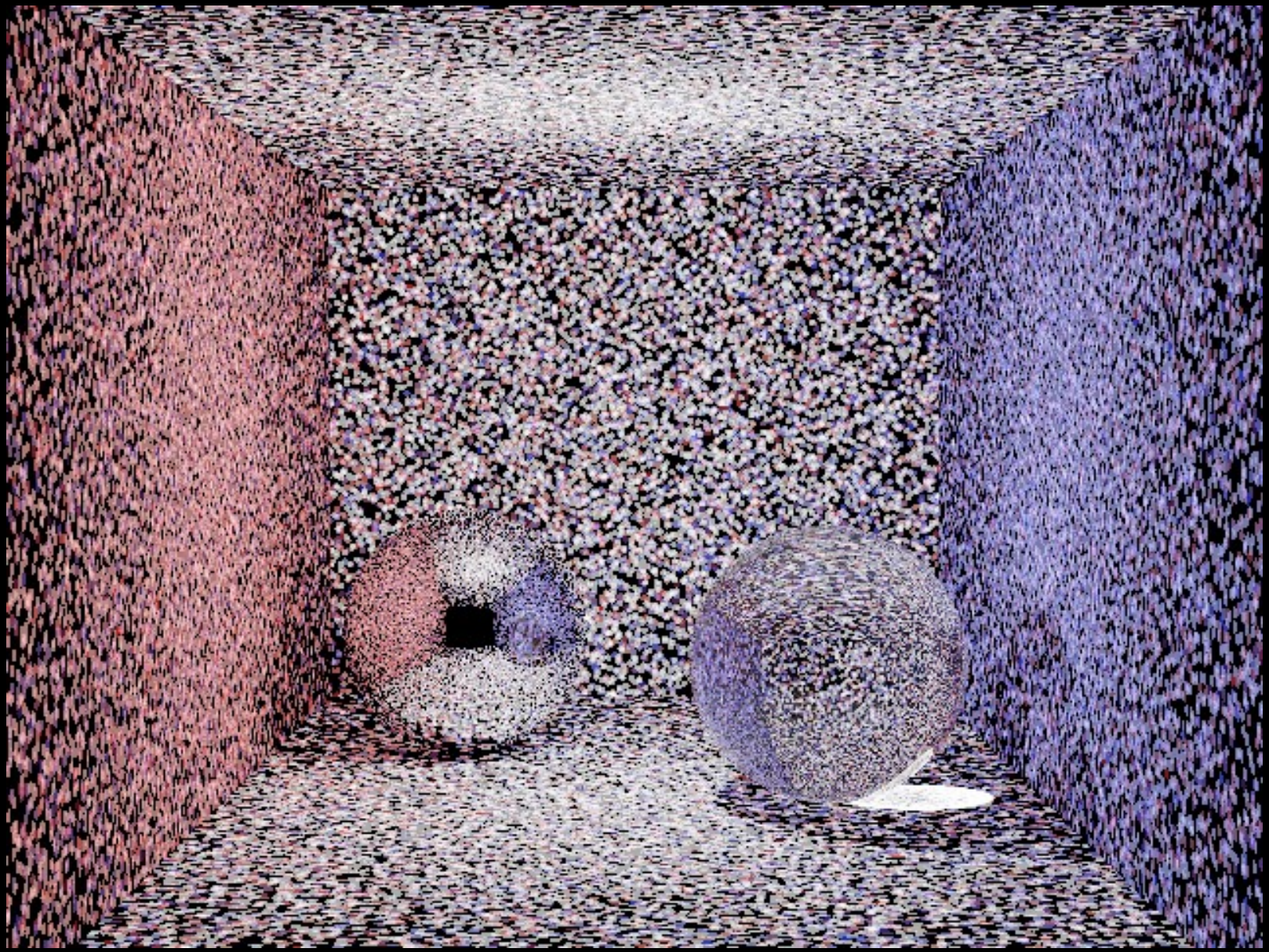
```
#include <math.h>   // smallppm, Progressive Photon Mapping by T. Hachisuka
#include <stdlib.h> // originally smallpt, a path tracer by Kevin Beason, 2008
#include <stdio.h>  // Usage: ./smallppm 100000 && xv image.ppm
#define PI ((double)3.14159265358979) // ^^^^^^:number of photons emitted
#define ALPHA ((double)0.7) // the alpha parameter of PPM
int primes[61]={2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,
83,89,97,101,103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,
191,193,197,199,211,223,227,229,233,239,241,251,257,263,269,271,277,281,283};
inline int rev(const int i,const int p) {if (i==0) return i; else return p-i;}
double hal(const int b, int j) { // Halton sequence with reverse permutation
  const int p = primes[b]; double h = 0.0, f = 1.0 / (double)p, fct = f;
  while (j > 0) {h += rev(j % p, p) * fct; j /= p; fct *= f;} return h;}
struct Vec {double x, y, z; // vector: position, also color (r,g,b)
  Vec(double x_ = 0, double y_ = 0, double z_ = 0) {x = x_; y = y_; z = z_;}
  inline Vec operator+(const Vec &b) const {return Vec(x+b.x, y+b.y, z+b.z);}
  inline Vec operator-(const Vec &b) const {return Vec(x-b.x, y-b.y, z-b.z);}
  inline Vec operator+(double b) const {return Vec(x + b, y + b, z + b);}
  inline Vec operator-(double b) const {return Vec(x - b, y - b, z - b);}
  inline Vec operator*(double b) const {return Vec(x * b, y * b, z * b);}
  inline Vec mul(const Vec &b) const {return Vec(x * b.x, y * b.y , z * b.z);}
  inline Vec norm() {return (*this) * (1.0 / sqrt(x*x+y*y+z*z));}
  inline 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);}};
#define MAX(x, y) ((x > y) ? x : y)
struct AABB {Vec min, max; // axis aligned bounding box
  inline void fit(const Vec &p) {
    if (p.x<min.x)min.x=p.x; if (p.y<min.y)min.y=p.y; if (p.z<min.z)min.z=p.z;
    max.x=MAX(p.x, max.x); max.y=MAX(p.y, max.y); max.z=MAX(p.z, max.z);}
  inline void reset() {min=Vec(1e20,1e20,1e20); max=Vec(-1e20,-1e20,-1e20);}};
struct HPoint {Vec f,pos,nrm,flux; double r2; unsigned int n; int pix;};
struct List {HPoint *id; List *next;};
List* ListAdd(HPoint *i,List* h){List* p=new List;p->id=i;p->next=h;return p;}
unsigned int num_hash, pixel_index, num_photon;
double hash_s; List **hash_grid; List *hitpoints = NULL; AABB hpbbox;
inline unsigned int hash(const int ix, const int iy, const int iz) {
return (unsigned int)((ix*73856093)^(iy*19349663)^(iz*83492791))%num_hash;}
void build_hash_grid(const int w, const int h) {
  hpbbox.reset(); List *lst = hitpoints; while (lst != NULL) {
    HPoint *hp=lst->id; lst=lst->next; hpbbox.fit(hp->pos);}
  Vec ssize = hpbbox.max - hpbbox.min; // compute initial radius
  double irad = ((ssize.x + ssize.y + ssize.z) / 3.0) / ((w + h) / 2.0) * 2.0;
  hpbbox.reset(); lst = hitpoints; int vphoton = 0; // determine hash size
    while (lst != NULL) {HPoint *hp = lst->id; lst = lst->next;
    hp->r2=irad *irad; hp->n = 0; hp->flux = Vec();
    vphoton++; hpbbox.fit(hp->pos-irad); hpbbox.fit(hp->pos+irad);}
  hash_s=1.0/(irad*2.0); num_hash = vphoton; hash_grid=new List*[num_hash];
  for (unsigned int i=0; i<num_hash;i++) hash_grid[i] = NULL;
  lst = hitpoints; while (lst != NULL) { // store hitpoints in the hashed grid
    HPoint *hp = lst->id; lst = lst->next;
    Vec BMin = ((hp->pos - irad) - hpbbox.min) * hash_s;
    Vec BMax = ((hp->pos + irad) - hpbbox.min) * hash_s;
    for (int iz = abs(int(BMin.z)); iz <= abs(int(BMax.z)); iz++)
      for (int iy = abs(int(BMin.y)); iy <= abs(int(BMax.y)); iy++)
        for (int ix = abs(int(BMin.x)); ix <= abs(int(BMax.x)); ix++)
          {int hv=hash(ix,iy,iz); hash_grid[hv]=ListAdd(hp,hash_grid[hv]);}}}
struct Ray {Vec o, d; Ray(){}; Ray(Vec o_, Vec d_) : o(o_), d(d_) {}};
enum Refl_t {DIFF, SPEC, REFR};  // material types, used in radiance()
struct Sphere {double rad; Vec p, c; Refl_t refl;
  Sphere(double r_,Vec p_,Vec c_,Refl_t re_) : rad(r_),p(p_),c(c_),refl(re_){}
  inline double intersect(const Ray &r) const { // returns distance
    Vec op=p-r.o; double t, b=op.dot(r.d), det=b*b-op.dot(op)+rad*rad;
    if (det < 0) return 1e20; else det = sqrt(det);
    return (t=b-det) > 1e-4 ? t : ((t=b+det)>1e-4 ? t : 1e20);}};
Sphere sph[] = { // Scene: radius, position, color, material
  Sphere(1e5, Vec( 1e5+1,40.8,81.6), Vec(.75,.25,.25),DIFF),//Left
  Sphere(1e5, Vec(-1e5+99,40.8,81.6),Vec(.25,.25,.75),DIFF),//Rght
  Sphere(1e5, Vec(50,40.8, 1e5),     Vec(.75,.75,.75),DIFF),//Back
  Sphere(1e5, Vec(50,40.8,-1e5+170), Vec(),           DIFF),//Frnt
  Sphere(1e5, Vec(50, 1e5, 81.6),    Vec(.75,.75,.75),DIFF),//Botm
  Sphere(1e5, Vec(50,-1e5+81.6,81.6),Vec(.75,.75,.75),DIFF),//Top
  Sphere(16.5,Vec(27,16.5,47),       Vec(1,1,1)*.999, SPEC),//Mirr
  Sphere(16.5,Vec(73,16.5,88),       Vec(1,1,1)*.999, REFR),//Glas
  Sphere(8.5, Vec(50,8.5,60),        Vec(1,1,1)*.999, DIFF)};//Mid
int toInt(double x){return int(pow(1-exp(-x),1/2.2)*255+.5);} //tone mapping
inline bool intersect(const Ray &r,double &t,int &id){//ray-sphere intrsect.
  int n = sizeof(sph) / sizeof(Sphere); double d, inf = 1e20; t = inf;
  for(int i=0;i<n;i++){d=sph[i].intersect(r);if(d<t){t=d;id=i;}}return t<inf;}
void genp(Ray* pr, Vec* f, int i) {*f = Vec(2500,2500,2500)*(PI*4.0);
  double p=2.*PI*hal(0,i),t=2.*acos(sqrt(1.-hal(1,i)));
  double st=sin(t);pr->d=Vec(cos(p)*st,cos(t),sin(p)*st);pr->o=Vec(50,60,85);}
void trace(const Ray &r,int dpt,bool m,const Vec &fl,const Vec &adj,int i) {
  double t;int id; dpt++;if(!intersect(r,t,id)||(dpt>=20))return;int d3=dpt*3;
  const Sphere &obj = sph[id]; Vec x=r.o+r.d*t, n=(x-obj.p).norm(), f=obj.c;
  Vec nl=n.dot(r.d)<0?n:n*-1; double p=f.x>f.y&&f.x>f.z?f.x:f.y>f.z?f.y:f.z;
  if (obj.refl == DIFF) {double r1=2.*PI*hal(d3-1,i),r2=hal(d3+0,i);
    double r2s=sqrt(r2);Vec w=nl,u=((fabs(w.x)>.1?Vec(0,1):Vec(1))%w).norm();
    Vec v=w%u, d = (u*cos(r1)*r2s + v*sin(r1)*r2s + w*sqrt(1-r2)).norm();
    if (m) {HPoint* hp=new HPoint; hp->f=f.mul(adj); hp->pos=x;
      hp->nrm=n; hp->pix = pixel_index; hitpoints = ListAdd(hp, hitpoints);
    } else {Vec hh = (x-hpbbox.min) * hash_s;
      int ix = abs(int(hh.x)), iy = abs(int(hh.y)), iz = abs(int(hh.z));
      // strictly speaking, we should use #pragma omp critical here
      {List* hp = hash_grid[hash(ix, iy, iz)]; while (hp != NULL) {
        HPoint *hitpoint = hp->id; hp = hp->next; Vec v = hitpoint->pos - x;
        if ((hitpoint->nrm.dot(n) > 1e-3) && (v.dot(v) <= hitpoint->r2)) {
            double g = (hitpoint->n*ALPHA+ALPHA) / (hitpoint->n*ALPHA+1.0);
            hitpoint->r2=hitpoint->r2*g; hitpoint->n++;
            hitpoint->flux=(hitpoint->flux+hitpoint->f.mul(fl)*(1./PI))*g;}}}
      if (hal(d3+1,i)<p) trace(Ray(x,d),dpt,m,f.mul(fl)*(1./p),adj,i);}
  } else if (obj.refl == SPEC) {
    trace(Ray(x, r.d-n*2.0*n.dot(r.d)), dpt, m, f.mul(fl), f.mul(adj),i);
  } else {Ray lr(x,r.d-n*2.0*n.dot(r.d)); bool into = (n.dot(nl)>0.0);
    double nc = 1.0, 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 trace(lr,dpt,m,fl,adj,i);
    Vec td = (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:td.dot(n));
    double Re=R0+(1-R0)*c*c*c*c*c,P=Re;Ray rr(x,td);Vec fa=f.mul(adj);
    if(m){trace(lr,dpt,m,fl,fa*Re,i);trace(rr,dpt,m,fl,fa*(1.0-Re),i);}
    else {(hal(d3-1,i)<P)?trace(lr,dpt,m,fl,fa,i):trace(rr,dpt,m,fl,fa,i);}}}
int main(int argc, char *argv[]) {
  int w=1024, h=768, samps = (argc==2) ? MAX(atoi(argv[1])/1000,1) : 1;
  Ray cam(Vec(50,48,295.6), Vec(0,-0.042612,-1).norm());
  Vec cx=Vec(w*.5135/h), cy=(cx%cam.d).norm()*.5135, *c=new Vec[w*h], vw;
  for (int y=0; y<h; y++){
    fprintf(stderr, "\rHitPointPass %5.2f%%", 100.0*y/(h-1));
    for (int x=0; x<w; x++) {pixel_index = x + y * w;
      Vec d = cx * ((x + 0.5) / w - 0.5) + cy * (-(y + 0.5) / h + 0.5)+cam.d;
      trace(Ray(cam.o + d * 140, d.norm()), 0, true, Vec(), Vec(1, 1, 1),0);}}
 fprintf(stderr,"\n"); build_hash_grid(w,h); num_photon=samps; vw=Vec(1,1,1);
 #pragma omp parallel for schedule(dynamic, 1)
 for(unsigned int i=0;i<num_photon;i++) {double p=100.*(i+1)/num_photon;
   fprintf(stderr,"\rPhotonPass %5.2f%%",p); int m=1000*i; Ray r; Vec f;
   for(int j=0;j<1000;j++){genp(&r,&f,m+j); trace(r,0,0>1,f,vw,m+j);}}
 List* lst=hitpoints; while (lst != NULL) {HPoint* hp=lst->id;lst=lst->next;
   int i=hp->pix;c[i]=c[i]+hp->flux*(1.0/(PI*hp->r2*num_photon*1000.0));}
 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));}}
```

smallpt

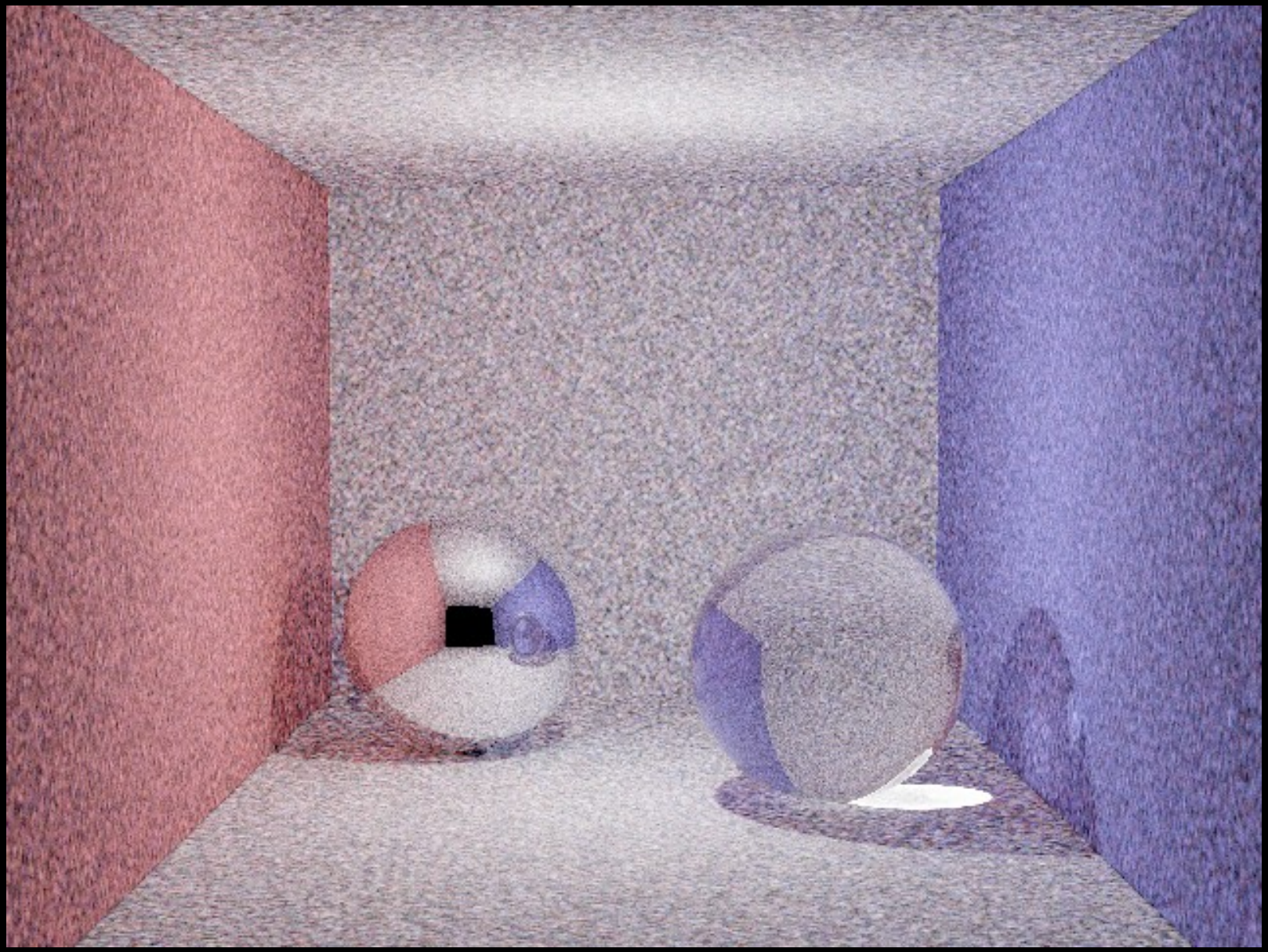


smallppm

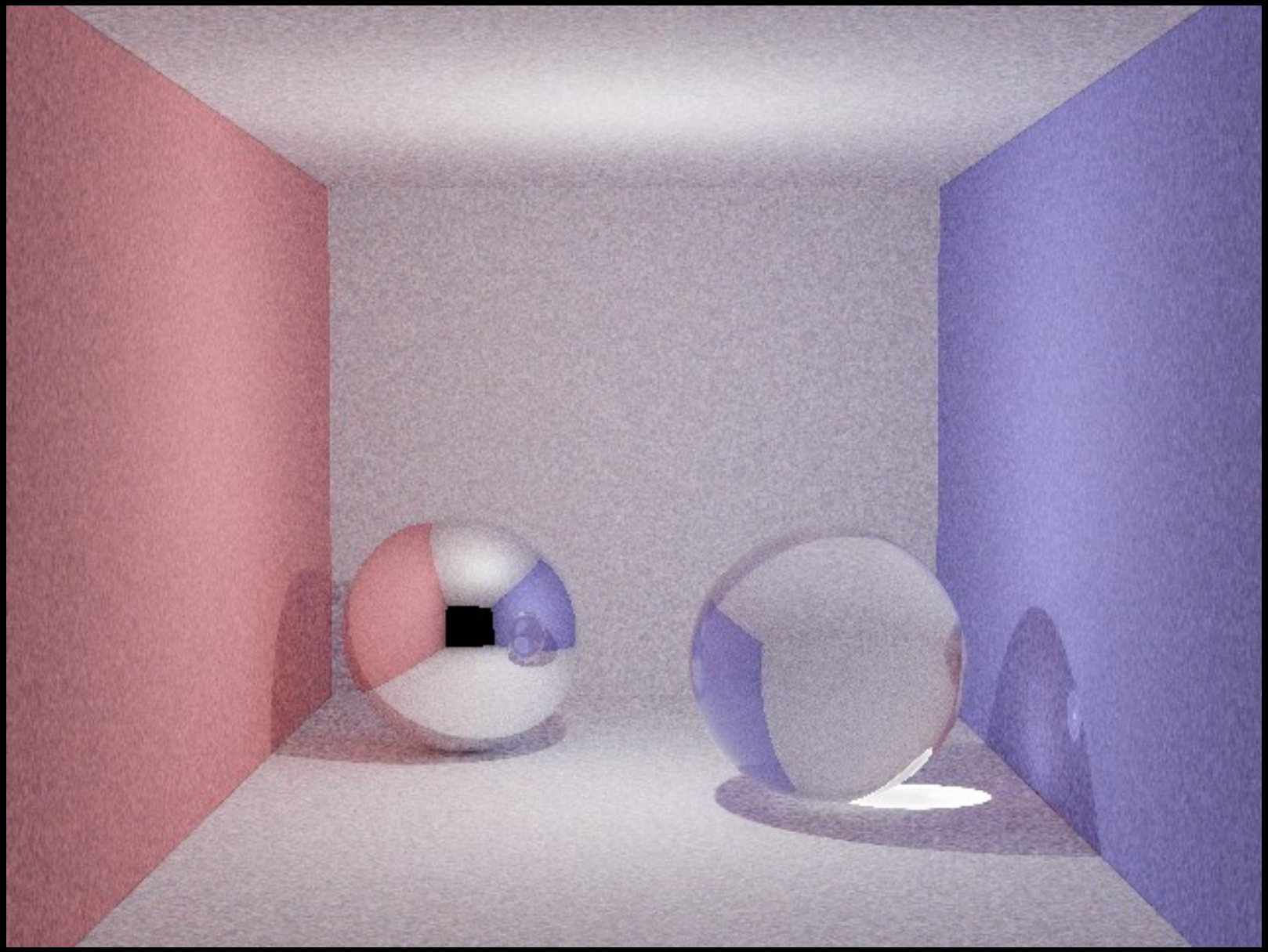




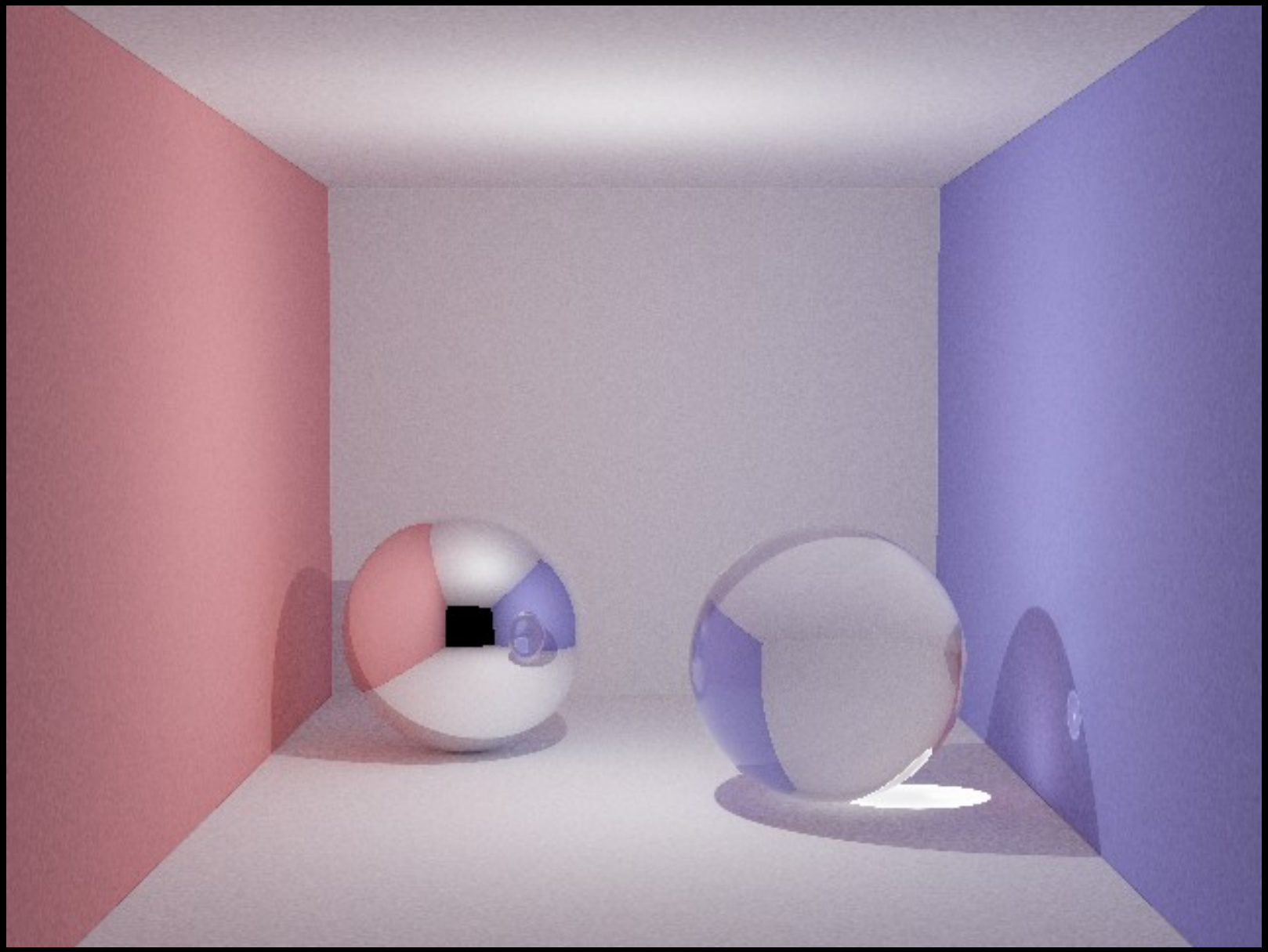
40K photons



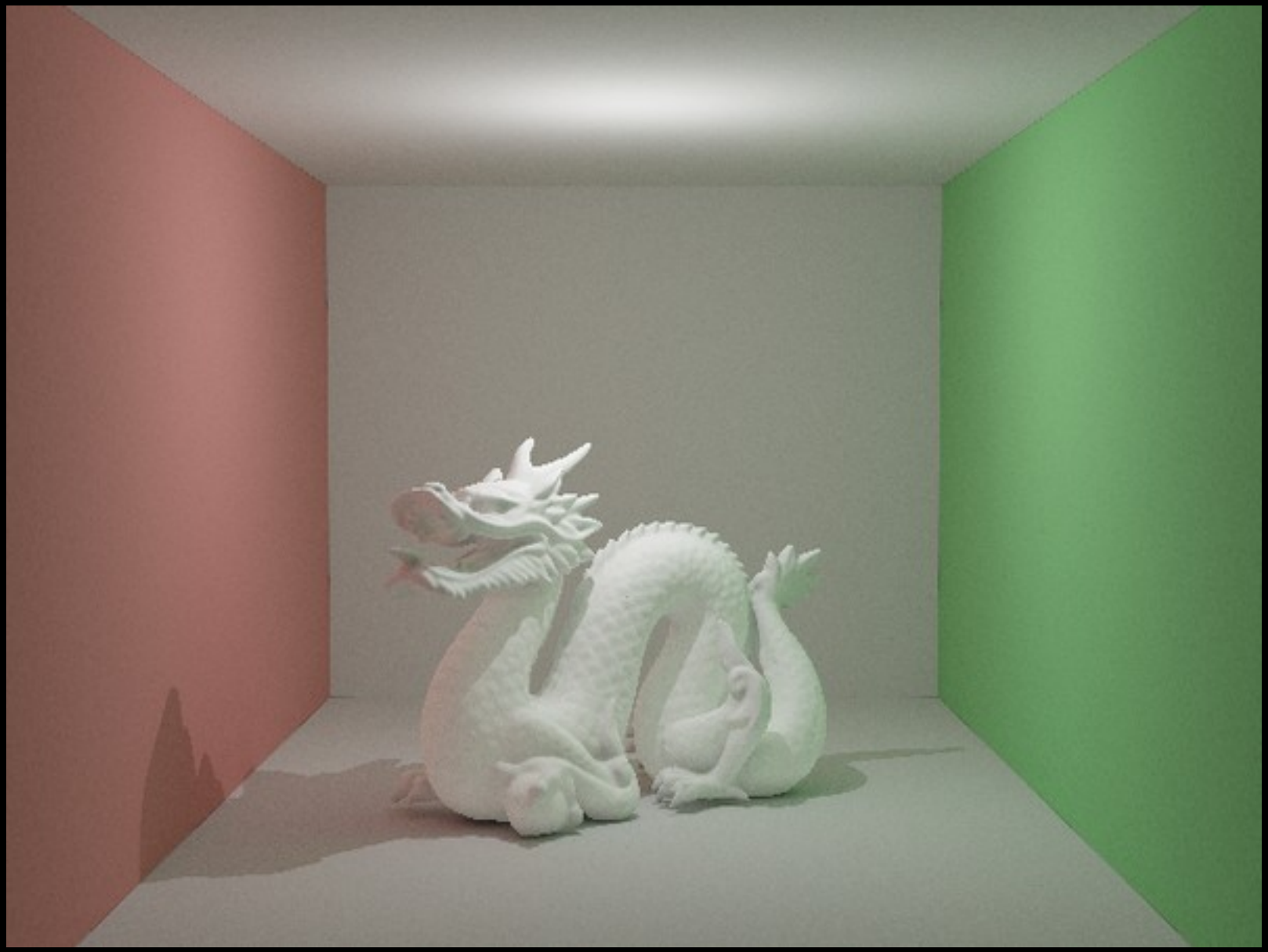
400K photons



4M photons



40M photons



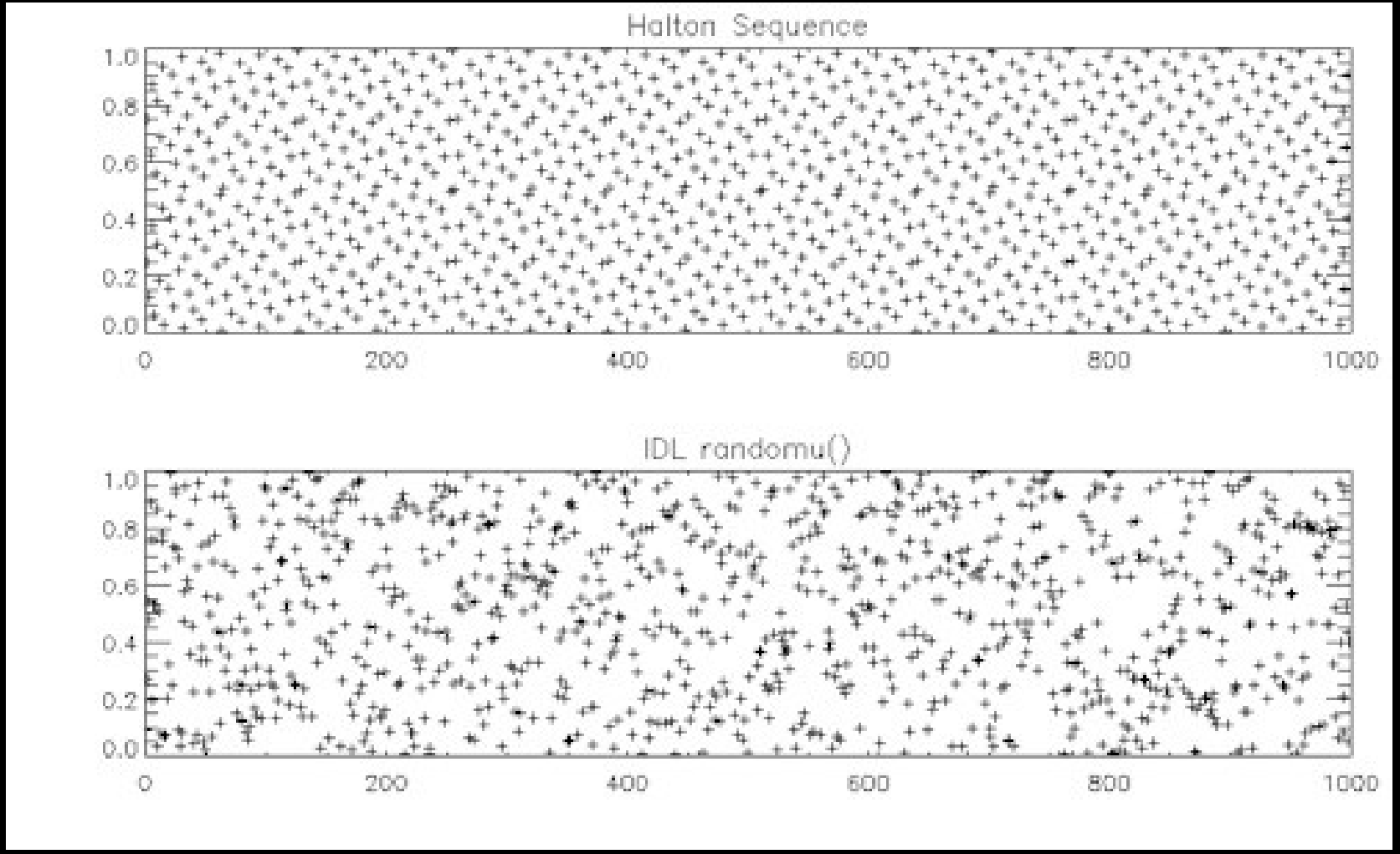
Extended smallppm @ 140M photons

Quasi Monte Carlo Sequence (6-12)

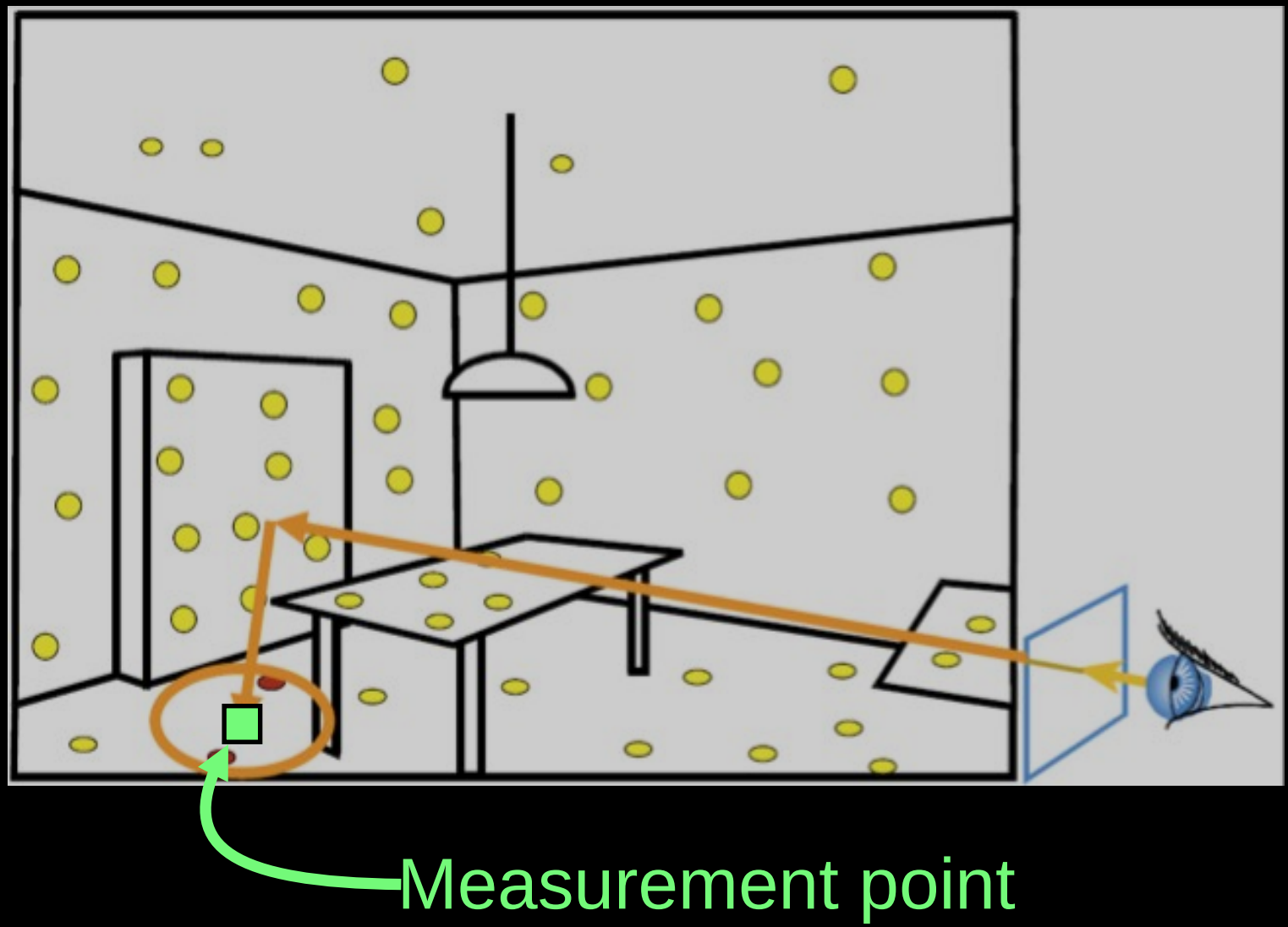
- Halton sequence with the reverse permutation
  - Random but well distributed numbers
  - Better convergence than rand()
  - hal(dimensions, index to sample)
  - Optional - we can use rand() instead

```
6  int primes[61]={2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,
7  83,89,97,101,103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,
8  191,193,197,199,211,223,227,229,233,239,241,251,257,263,269,271,277,281,283};
9  inline int rev(const int i,const int p) {if (i==0) return i; else return p-i;}
10 double hal(const int b, int j) { // Halton sequence with reverse permutation
11   const int p = primes[b]; double h = 0.0, f = 1.0 / (double)p, fct = f;
12   while (j > 0) {h += rev(j % p, p) * fct; j /= p; fct *= f;} return h;}
```

Quasi Monte Carlo Sequence



Measurement Points (30)



Spatial Hashing (31-55)

- Faster counting of points in a region

```
31  struct List {HPoint *id; List *next;};
32  List* ListAdd(HPoint *i,List* h){List* p=new List;p->id=i;p->next=h;return p;}
33  unsigned int num_hash, pixel_index, num_photon;
34  double hash_s; List **hash_grid; List *hitpoints = NULL; AABB hpbbox;
35  inline unsigned int hash(const int ix, const int iy, const int iz) {
36  return (unsigned int)((ix*73856093)^(iy*19349663)^(iz*83492791))%num_hash;}
37  void build_hash_grid(const int w, const int h) {
38    hpbbox.reset(); List *lst = hitpoints; while (lst != NULL) {
39      HPoint *hp=lst->id; lst=lst->next; hpbbox.fit(hp->pos);}
40    Vec ssize = hpbbox.max - hpbbox.min; // compute initial radius
41    double irad = ((ssize.x + ssize.y + ssize.z) / 3.0) / ((w + h) / 2.0) * 2.0;
42    hpbbox.reset(); lst = hitpoints; int vphoton = 0; // determine hash size
43      while (lst != NULL) {HPoint *hp = lst->id; lst = lst->next;
44      hp->r2=irad *irad; hp->n = 0; hp->flux = Vec();
45      vphoton++; hpbbox.fit(hp->pos-irad); hpbbox.fit(hp->pos+irad);}
46    hash_s=1.0/(irad*2.0); num_hash = vphoton; hash_grid=new List*[num_hash];
47    for (unsigned int i=0; i<num_hash;i++) hash_grid[i] = NULL;
48    lst = hitpoints; while (lst != NULL) { // store hitpoints in the hashed grid
49      HPoint *hp = lst->id; lst = lst->next;
50      Vec BMin = ((hp->pos - irad) - hpbbox.min) * hash_s;
51      Vec BMax = ((hp->pos + irad) - hpbbox.min) * hash_s;
52      for (int iz = abs(int(BMin.z)); iz <= abs(int(BMax.z)); iz++)
53        for (int iy = abs(int(BMin.y)); iy <= abs(int(BMax.y)); iy++)
54          for (int ix = abs(int(BMin.x)); ix <= abs(int(BMax.x)); ix++)
55            {int hv=hash(ix,iy,iz); hash_grid[hv]=ListAdd(hp,hash_grid[hv]);}}}
```

Eye Pass (112-119)

- Basically the same as smallpt, however,
  - “trace” stops at the first diffuse hit point
  - Store hit points as measurement points
  - Build spatial hash over measurement points

```
112   Ray cam(Vec(50,48,295.6), Vec(0,-0.042612,-1).norm());
113   Vec cx=Vec(w*.5135/h), cy=(cx%cam.d).norm()*.5135, *c=new Vec[w*h], vw;
114   for (int y=0; y<h; y++){
115     fprintf(stderr, "\rHitPointPass %5.2f%%", 100.0*y/(h-1));
116     for (int x=0; x<w; x++) {pixel_index = x + y * w;
117       Vec d = cx * ((x + 0.5) / w - 0.5) + cy * (-(y + 0.5) / h + 0.5)+cam.d;
118       trace(Ray(cam.o + d * 140, d.norm()), 0, true, Vec(), Vec(1, 1, 1),0);}}
119   fprintf(stderr,"\n"); build_hash_grid(w,h); num_photon=samps; vw=Vec(1,1,1);
```

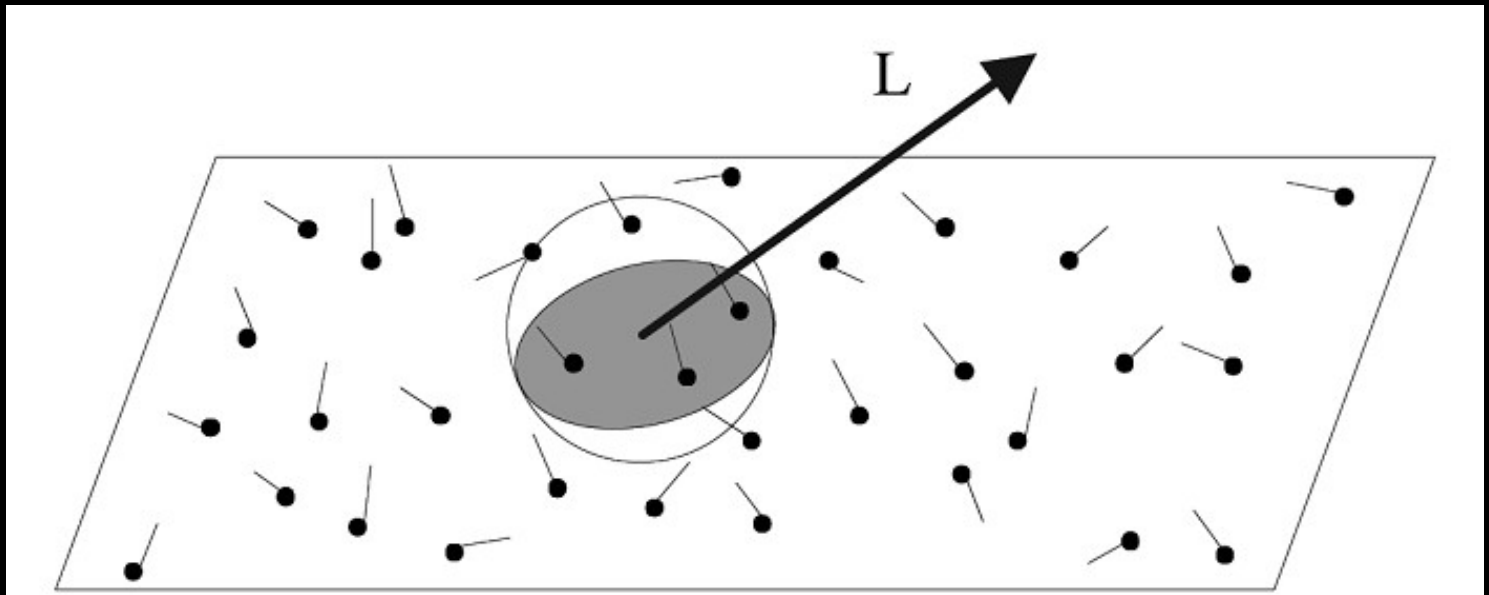
Photon Pass (120-125)

- Trace rays from the light source (120-123)
  - Same “trace” function but from light
  - Accumulate photons onto measurement points
- Estimate radiance on the image (124-125)

```
120     #pragma omp parallel for schedule(dynamic, 1)
121     for(unsigned int i=0;i<num_photon;i++) {double p=100.*(i+1)/num_photon;
122       fprintf(stderr,"\rPhotonPass %5.2f%%",p); int m=1000*i; Ray r; Vec f;
123       for(int j=0;j<1000;j++){genp(&r,&f,m+j); trace(r,0,0>1,f,vw,m+j);}}
124     List* lst=hitpoints; while (lst != NULL) {HPoint* hp=lst->id;lst=lst->next;
125       int i=hp->pix;c[i]=c[i]+hp->flux*(1.0/(PI*hp->r2*num_photon*1000.0));}
```

Radiance Estimation (124-125)

- Estimate density of photons within the circle
  - Division by the area (PI * r^2)
- Each photon carries equal flux from the light
  - Division by the number of emitted photons




```
124   List* lst=hitpoints; while (lst != NULL) {HPoint* hp=lst->id;lst=lst->next;
125     int i=hp->pix;c[i]=c[i]+hp->flux*(1.0/(PI*hp->r2*num_photon*1000.0));}
```

Tips

- Adjust the initial radius if not working well

```
31  struct List {HPoint *id; List *next;};
32  List* ListAdd(HPoint *i,List* h){List* p=new List;p->id=i;p->next=h;return p;}
33  unsigned int num_hash, pixel_index, num_photon;
34  double hash_s; List **hash_grid; List *hitpoints = NULL; AABB hpbbox;
35  inline unsigned int hash(const int ix, const int iy, const int iz) {
36  return (unsigned int)((ix*73856093)^(iy*19349663)^(iz*83492791))%num_hash;}
37  void build_hash_grid(const int w, const int h) {
38    hpbbox.reset(); List *lst = hitpoints; while (lst != NULL) {
39      HPoint *hp=lst->id; lst=lst->next; hpbbox.fit(hp->pos);}
40    Vec ssize = hpbbox.max - hpbbox.min; // compute initial radius
41    double irad = ((ssize.x + ssize.y + ssize.z) / 3.0) / ((w + h) / 2.0) * 2.0;
42    hpbbox.reset(); lst = hitpoints; int vphoton = 0; // determine hash size
43      while (lst != NULL) {HPoint *hp = lst->id; lst = lst->next;
44      hp->r2=irad *irad; hp->n = 0; hp->flux = Vec();
45      vphoton++; hpbbox.fit(hp->pos-irad); hpbbox.fit(hp->pos+irad);}
46    hash_s=1.0/(irad*2.0); num_hash = vphoton; hash_grid=new List*[num_hash];
47    for (unsigned int i=0; i<num_hash;i++) hash_grid[i] = NULL;
48    lst = hitpoints; while (lst != NULL) { // store hitpoints in the hashed grid
49      HPoint *hp = lst->id; lst = lst->next;
50      Vec BMin = ((hp->pos - irad) - hpbbox.min) * hash_s;
51      Vec BMax = ((hp->pos + irad) - hpbbox.min) * hash_s;
52      for (int iz = abs(int(BMin.z)); iz <= abs(int(BMax.z)); iz++)
53        for (int iy = abs(int(BMin.y)); iy <= abs(int(BMax.y)); iy++)
54          for (int ix = abs(int(BMin.x)); ix <= abs(int(BMax.x)); ix++)
55            {int hv=hash(ix,iy,iz); hash_grid[hv]=ListAdd(hp,hash_grid[hv]);}}}
```

Tips

- Trace O(100M) photons for noiseless images

- Do not close up a small part of the scene
  - Photons do not “know” where you look at
  - Need many more photons

PT or PM?

Path tracing is more efficient in general when it works (e.g., no caustics, Lambertian surfaces, large area light sources)

Photon density estimation can render scenes that are nearly impossible to render with path tracing (i.e., reflection of caustics)

You can actually combine them, but its implementation would not be trivial