

**smallpt:  
Global Illumination in 99  
lines of C++**

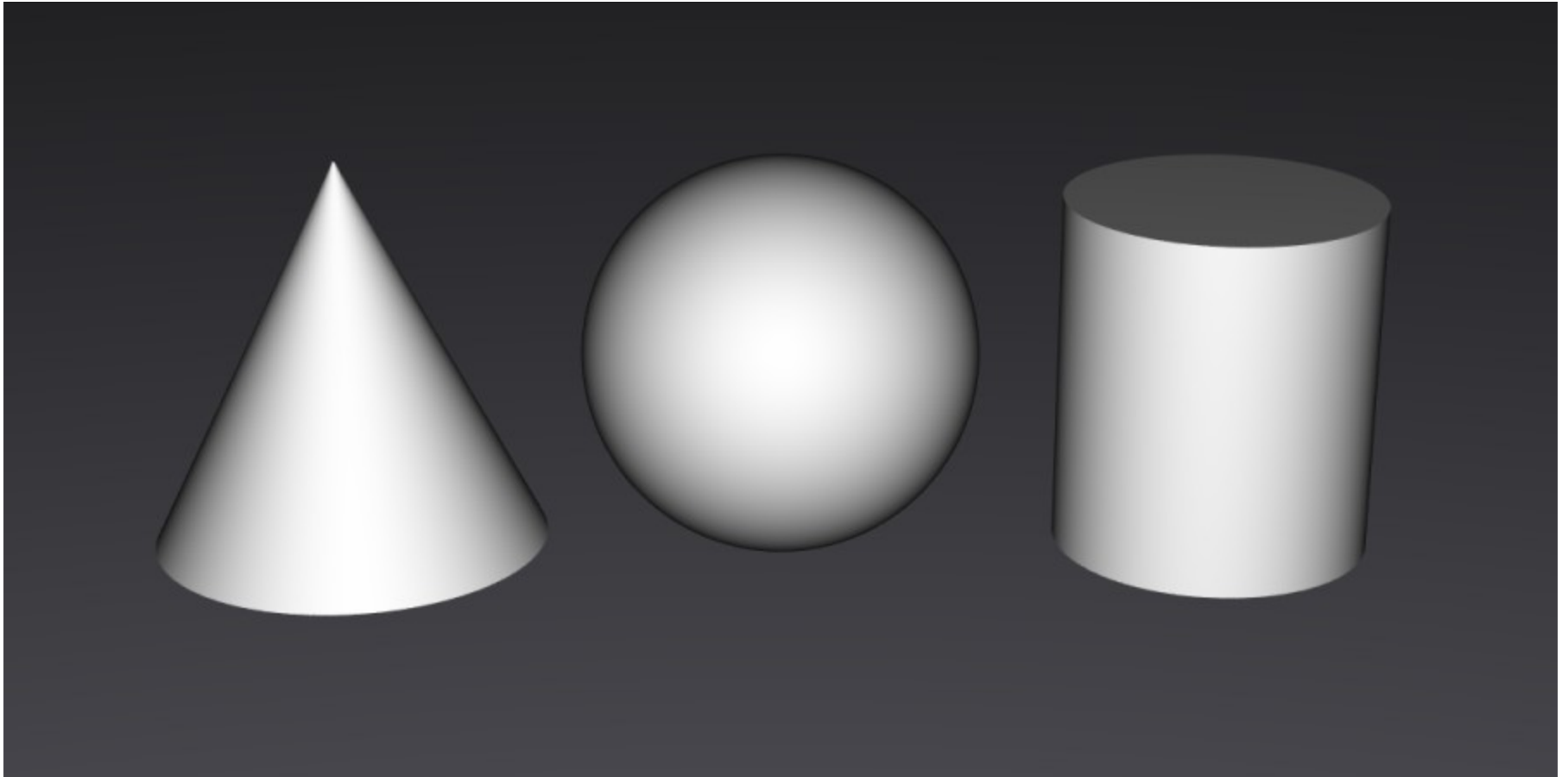
**a ray tracer by Kevin Beason  
<http://kevinbeason.com/smallpt/>**

Presentation by  
Dr. David Cline  
Oklahoma State University

# Global Illumination

- Global Illumination = “virtual photography”
  - Given a scene description that specifies the location of surfaces in a scene, the location of lights, and the location of a camera, take a virtual “photograph” of that scene.

- “Headlight” rendering of a simple scene



- Adding surface details



- Direct lighting with hard shadows



- “Ambient occlusion” = direct lighting of a cloudy day.



- Ambient Occlusion and depth of field





- Global illumination showing different surface types, glass surfaces, caustics (light concentrations), and depth of field.



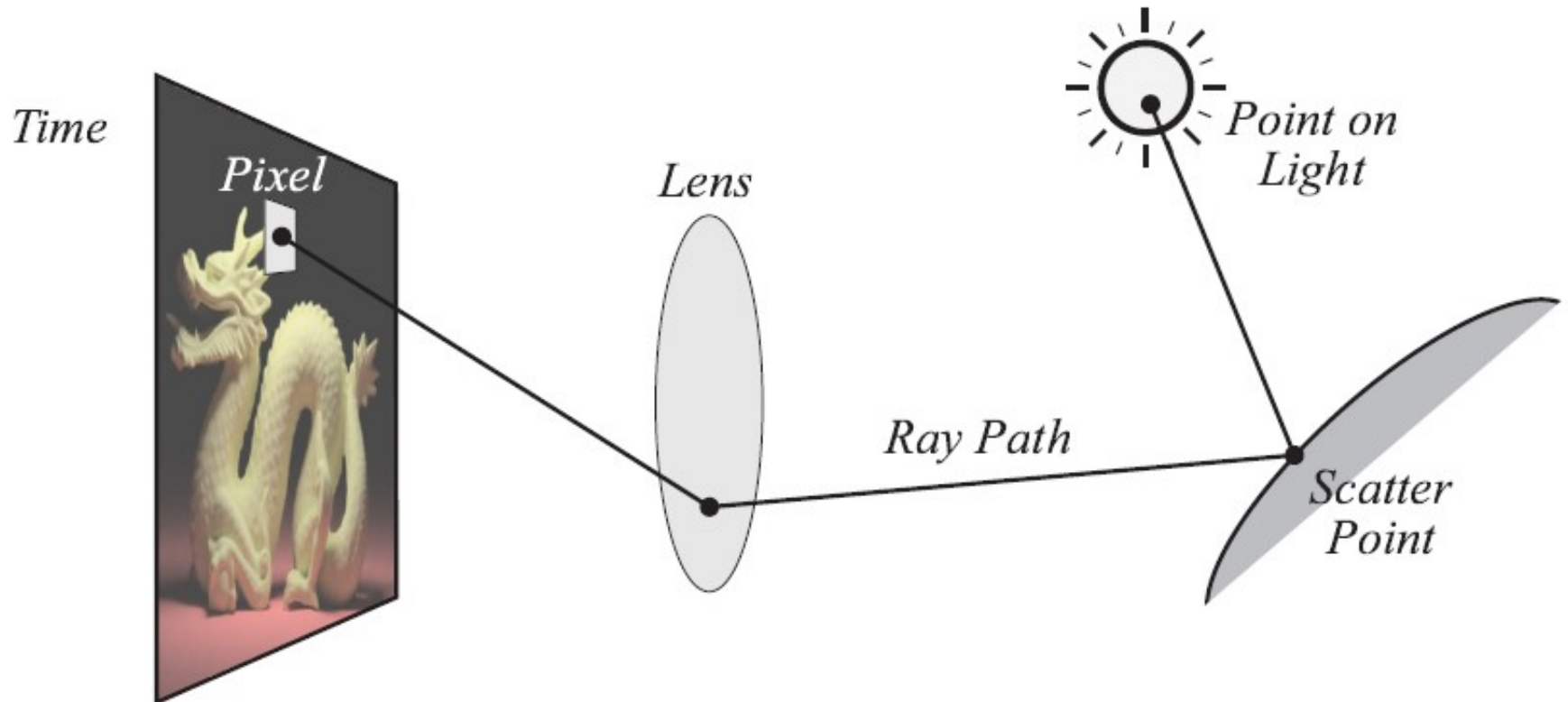


# Another Example

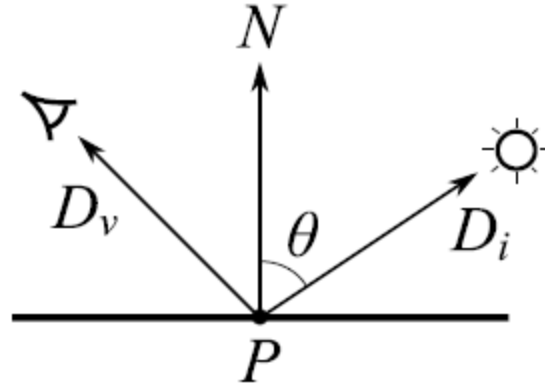
- Ad-hoc Lighting vs. Global Illumination



# How to form a GI image?

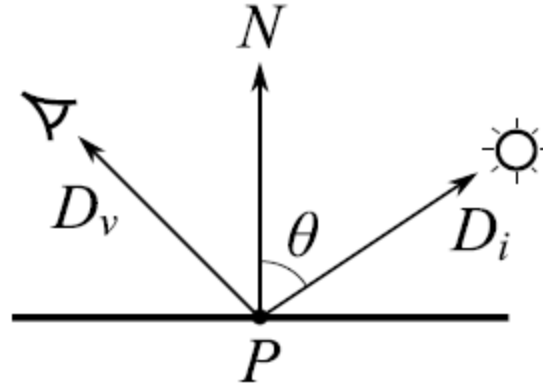


# The Rendering Equation



$$L(P \rightarrow D_v) = L_e(P \rightarrow D_v) + \int_{\Omega} F_s(D_v, D_i) |\cos \theta| L(Y_i \rightarrow -D_i) dD_i$$

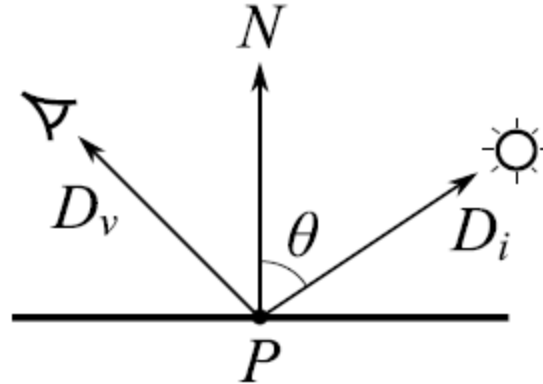
# The Rendering Equation



$$L(P \rightarrow D_v) = L_e(P \rightarrow D_v) + \int_{\Omega} F_s(D_v, D_i) |\cos \theta| L(Y_i \rightarrow -D_i) dD_i$$

The radiance (intensity of light)  
Coming from surface point P  
In direction  $D_v$ . This is what we  
Have to calculate.

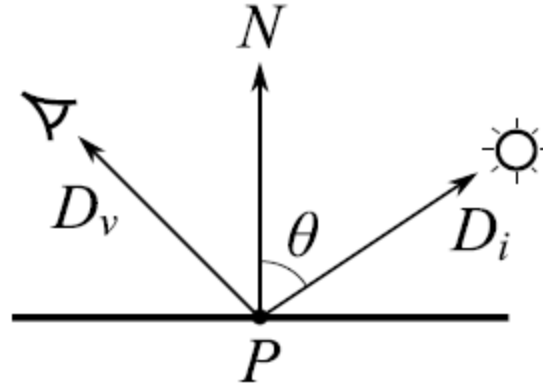
# The Rendering Equation



$$L(P \rightarrow D_v) = L_e(P \rightarrow D_v) + \int_{\Omega} F_s(D_v, D_i) |\cos \theta| L(Y_i \rightarrow -D_i) dD_i$$

The self-emitted radiance from  $P$   
 In direction  $D_v$  (0 unless point  $P$   
 Is a light source) This can be looked  
 Up as part of the scene description.

# The Rendering Equation

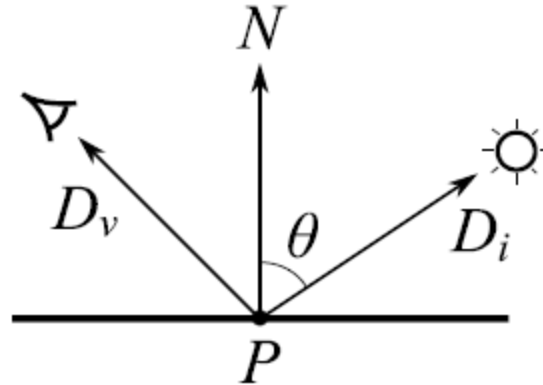


$$L(P \rightarrow D_v) = L_e(P \rightarrow D_v) + \int_{\Omega} F_s(D_v, D_i) |\cos \theta| L(Y_i \rightarrow -D_i) dD_i$$

The reflected light term. Here we must add  
Up (integrate) all of the light coming in to  
point P from all directions, modulated by the  
Chance that it scatters in direction  $D_v$   
(based on the BRDF function,  $F_s$ )



# Path Tracing Approximation



$$\hat{L}(P \rightarrow D_v) = L_e(P \rightarrow D_v) + \frac{F_s(D_v, D_i) |\cos \theta| \hat{L}(Y_i \rightarrow -D_i)}{p_{angle}^{tot}(D_i)}$$

Replace the ray integral with a Monte Carlo (random) Sample that has the same Expected (average) Value. Then average a bunch of samples for each pixel to create a smooth image.

# Path Tracing Algorithm

---

**Algorithm 3** Path Tracing Main Loop

---

```
1: for each pixel (i,j) do
2:   Vec3  $C = 0$ 
3:   for (k=0; k < samplesPerPixel; k++) do
4:     Create random ray in pixel:
5:       Choose random point on lens  $P_{lens}$ 
6:       Choose random point on image plane  $P_{image}$ 
7:        $D = \text{normalize}(P_{image} - P_{lens})$ 
8:       Ray ray = Ray( $P_{lens}$ ,  $D$ )
9:       castRay(ray, isect)
10:    if the ray hits something then
11:       $C += \text{radiance}(\text{ray}, \text{isect}, 0)$ 
12:    else
13:       $C += \text{backgroundColor}(D)$ 
14:    end if
15:  end for
16:  image(i,j) =  $C / \text{samplesPerPixel}$ 
17: end for
```

---

# SmallPT

- A 99 line Path Tracer by Kevin Beason
- (Expanded Version has 218 lines)
- Major Parts:
  - Vec: a vector class, used for points, normals, colors
  - Ray: a ray class (origin and direction)
  - Refl\_t: the surface reflection type
  - Sphere: SmallPT only supports sphere objects
  - spheres: the hard coded scene (some # of spheres)
  - intersect: a routine to intersect rays with the scene of spheres
  - radiance: recursive routine that solves the rendering equation
  - main: program start and main loop that goes over each pixel

# Squashed Code 1:

```
1. #include <math.h> // smallpt, a Path Tracer by Kevin Beason, 2008
2. #include <stdlib.h> // Make : g++ -O3 -fopenmp smallpt.cpp -o smallpt
3. #include <stdio.h> // Remove "-fopenmp" for g++ version < 4.2
4. struct Vec { // Usage: time ./smallpt 5000 && xv image.ppm
5.     double x, y, z; // position, also color (r,g,b)
6.     Vec(double x_=0, double y_=0, double z_=0){ x=x_; y=y_; z=z_; }
7.     Vec operator+(const Vec &b) const { return Vec(x+b.x,y+b.y,z+b.z); }
8.     Vec operator-(const Vec &b) const { return Vec(x-b.x,y-b.y,z-b.z); }
9.     Vec operator*(double b) const { return Vec(x*b,y*b,z*b); }
10.    Vec mult(const Vec &b) const { return Vec(x*b.x,y*b.y,z*b.z); }
11.    Vec& norm(){ return *this = *this * (1/sqrt(x*x+y*y+z*z)); }
12.    double dot(const Vec &b) const { return x*b.x+y*b.y+z*b.z; } // cross:
13.    Vec operator%(Vec&b){return Vec(y*b.z-z*b.y,z*b.x-x*b.z,x*b.y-y*b.x);}
14. };
15. struct Ray { Vec o, d; Ray(Vec o_, Vec d_) : o(o_), d(d_) {} };
16. enum Refl_t { DIFF, SPEC, REFR }; // material types, used in radiance()
17. struct Sphere {
18.     double rad; // radius
19.     Vec p, e, c; // position, emission, color
20.     Refl_t refl; // reflection type (DIFFuse, SPECular, REFRactive)
21.     Sphere(double rad_, Vec p_, Vec e_, Vec c_, Refl_t refl_):
22.         rad(rad_), p(p_), e(e_), c(c_), refl(refl_) {}
23.     double intersect(const Ray &r) const { // returns distance, 0 if nohit
24.         Vec op = p-r.o; // Solve t^2*d.d + 2*t*(o-p).d + (o-p).(o-p)-R^2 = 0
25.         double t, eps=1e-4, b=op.dot(r.d), det=b*b-op.dot(op)+rad*rad;
26.         if (det<0) return 0; else det=sqrt(det);
27.         return (t=b-det)>eps ? t : ((t=b+det)>eps ? t : 0);
28.     }
29. };
30. Sphere spheres[] = { //Scene: radius, position, emission, color, material
31.     Sphere(1e5, Vec( 1e5+1,40.8,81.6), Vec(),Vec(.75,.25,.25),DIFF), //Left
32.     Sphere(1e5, Vec(-1e5+99,40.8,81.6),Vec(),Vec(.25,.25,.75),DIFF), //Rght
33.     Sphere(1e5, Vec(50,40.8, 1e5), Vec(),Vec(.75,.75,.75),DIFF), //Back
34.     Sphere(1e5, Vec(50,40.8,-1e5+170), Vec(),Vec(), DIFF), //Frnt
35.     Sphere(1e5, Vec(50, 1e5, 81.6), Vec(),Vec(.75,.75,.75),DIFF), //Botm
36.     Sphere(1e5, Vec(50,-1e5+81.6,81.6),Vec(),Vec(.75,.75,.75),DIFF), //Top
37.     Sphere(16.5,Vec(27,16.5,47), Vec(),Vec(1,1,1)*.999, SPEC), //Mirr
38.     Sphere(16.5,Vec(73,16.5,78), Vec(),Vec(1,1,1)*.999, REFR), //Glas
39.     Sphere(600, Vec(50,681.6-.27,81.6),Vec(12,12,12), Vec(), DIFF) //Lite
40. };
41. inline double clamp(double x){ return x<0 ? 0 : x>1 ? 1 : x; }
42. inline int toInt(double x){ return int(pow(clamp(x),1/2.2)*255+.5); }
43. inline bool intersect(const Ray &r, double &t, int &id){
44.     double n=sizeof(spheres)/sizeof(Sphere), d, inf=t-1e20;
45.     for(int i=int(n);i--;) if((d=spheres[i].intersect(r))&&d<t){t=d;id=i;}
46.     return t<inf;
47. }
```

# Squashed Code 2:

```
48. Vec radiance(const Ray &r, int depth, unsigned short *Xi){
49.     double t; // distance to intersection
50.     int id=0; // id of intersected object
51.     if (!intersect(r, t, id)) return Vec(); // if miss, return black
52.     const Sphere &obj = spheres[id]; // the hit object
53.     Vec x=r.o+r.d*t, n=(x-obj.p).norm(), nl=n.dot(r.d)<0?n:n*-1, f=obj.c;
54.     double p = f.x>f.y && f.x>f.z ? f.x : f.y>f.z ? f.y : f.z; // max refl
55.     if (++depth>5) if (erand48(Xi)<p) f=f*(1/p); else return obj.e; //R.R.
56.     if (obj.refl == DIFF){ // Ideal DIFFUSE reflection
57.         double r1=2*M_PI*erand48(Xi), r2=erand48(Xi), r2s=sqrt(r2);
58.         Vec w=nl, u=((fabs(w.x)>.1?Vec(0,1):Vec(1))%w).norm(), v=w%u;
59.         Vec d = (u*cos(r1)*r2s + v*sin(r1)*r2s + w*sqrt(1-r2)).norm();
60.         return obj.e + f.mult(radiance(Ray(x,d),depth,Xi));
61.     } else if (obj.refl == SPEC) // Ideal SPECULAR reflection
62.         return obj.e + f.mult(radiance(Ray(x,r.d-n*2*n.dot(r.d)),depth,Xi));
63.     Ray reflRay(x, r.d-n*2*n.dot(r.d)); // Ideal dielectric REFRACTION
64.     bool into = n.dot(nl)>0; // Ray from outside going in?
65.     double nc=1, nt=1.5, nnt=into?nc/nt:nt/nc, ddn=r.d.dot(nl), cos2t;
66.     if ((cos2t=1-nnt*nnt*(1-ddn*ddn))<0) // Total internal reflection
67.         return obj.e + f.mult(radiance(reflRay,depth,Xi));
68.     Vec tdir = (r.d*nnt - n*((into?1:-1)*(ddn*nnt+sqrt(cos2t)))).norm();
69.     double a=nt-nc, b=nt+nc, R0=a*a/(b*b), c = 1-(into?-ddn:tdir.dot(n));
70.     double Re=R0+(1-R0)*c*c*c*c*c,Tr=1-Re,P=.25+.5*Re,RP=Re/P,TP=Tr/(1-P);
71.     return obj.e + f.mult(depth>2 ? (erand48(Xi)<P ? // Russian roulette
72.         radiance(reflRay,depth,Xi)*RP:radiance(Ray(x,tdir),depth,Xi)*TP) :
73.         radiance(reflRay,depth,Xi)*Re+radiance(Ray(x,tdir),depth,Xi)*Tr);
74. }
75. int main(int argc, char *argv[]){
76.     int w=1024, h=768, samps = argc==2 ? atoi(argv[1])/4 : 1; // # samples
77.     Ray cam(Vec(50,52,295.6), Vec(0,-0.042612,-1).norm()); // cam pos, dir
78.     Vec cx=Vec(w*.5135/h), cy=(cx%cam.d).norm()**.5135, r, *c=new Vec[w*h];
79.     #pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP
80.     for (int y=0; y<h; y++){ // Loop over image rows
81.         fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1));
82.         for (unsigned short x=0, Xi[3]={0,0,y*y*y}; x<w; x++) // Loop cols
83.             for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++) // 2x2 subpixel rows
84.                 for (int sx=0; sx<2; sx++, r=Vec()){ // 2x2 subpixel cols
85.                     for (int s=0; s<samps; s++){
86.                         double r1=2*erand48(Xi), dx=r1<1 ? sqrt(r1)-1: 1-sqrt(2-r1);
87.                         double r2=2*erand48(Xi), dy=r2<1 ? sqrt(r2)-1: 1-sqrt(2-r2);
88.                         Vec d = cx*( (sx+.5 + dx)/2 + x)/w - .5) +
89.                             cy*( (sy+.5 + dy)/2 + y)/h - .5) + cam.d;
90.                         r = r + radiance(Ray(cam.o+d*140,d.norm()),0,Xi)*(1./samps);
91.                     } // Camera rays are pushed ^^^ forward to start in interior
92.                     c[i] = c[i] + Vec(clamp(r.x),clamp(r.y),clamp(r.z))**.25;
93.                 }
94.             }
95.     FILE *f = fopen("image.ppm", "w"); // Write image to PPM file.
96.     fprintf(f, "P3\n%d %d\n%d\n", w, h, 255);
97.     for (int i=0; i<w*h; i++)
98.         fprintf(f,"%d %d %d ", toInt(c[i].x), toInt(c[i].y), toInt(c[i].z));
99. }
```

# Expanded version (1)

## Preliminaries

```
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 }
```



# Expanded version (2)

## Vec (Points, Vectors, Colors)

```
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
```

# Normalize

- “Normalize” a vector = divide by its length

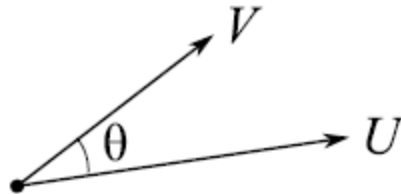
$$\|V\| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

25

```
Vec& norm() { return *this = *this * (1/sqrt(x*x+y*y+z*z)); }
```

# Dot Product

The *dot product* of two vectors,  $U \cdot V$ , is a scalar that describes the angle  $\theta$  between them:

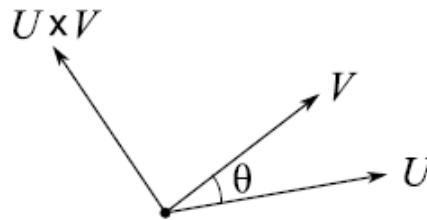


$$U \cdot V = (u_x v_x + u_y v_y + u_z v_z) = \|U\| \|V\| \cos \theta$$

Almost always in computer graphics when we write the cosine of an angle, it will be evaluated using the dot product.

# Cross Product

The *cross product* of two vectors,  $U \times V$  is a third vector that is perpendicular to both  $U$  and  $V$ , with direction defined by the *right hand rule*. The length of the cross product equals the product of the two vector lengths and the sine of the angle between them:



$$U \times V = (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x) \quad (3)$$

$$\|U \times V\| = \|U\| \|V\| \sin \theta$$

The length of the cross product is also the area of the parallelogram defined by  $U$  and  $V$ , (twice the area of the triangle they define).

# Ray Structure

- A ray is a parametric line with an origin (o) and a direction (d). A point along the ray can be defined using a parameter, t:
- In code we have:

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

$$P(t) = O + tD$$

- The core routines of the ray tracer intersect rays with geometric objects (spheres in our case)

# Sphere

- SmallPT supports sphere objects only
- We can define a sphere based on
  - a center point,  $C$
  - Radius,  $r$
- The equation of the sphere:

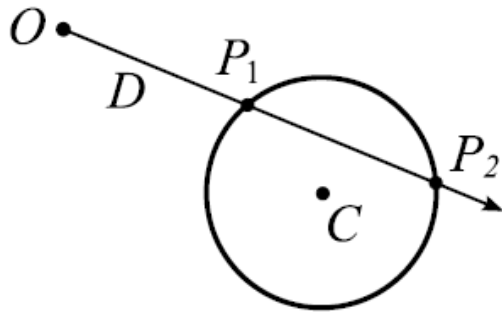
$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 - r^2 = 0$$

- In vector form:

$$(P - C) \cdot (P - C) - r^2 = 0$$



# Sphere Intersection



Start with vector equation of sphere

$$(P - C) \cdot (P - C) - r^2 = 0$$

Now, substitute the ray equation for P:  $P(t) = O + tD$

$$(O + tD - C) \cdot (O + tD - C) - r^2 = 0$$

$$(D \cdot D)t^2 + 2D \cdot (O - C)t + (O - C) \cdot (O - C) - r^2 = 0$$

...and solve for  $t$  using the quadratic formula:

$$a = (D \cdot D)$$

$$b = 2D \cdot (O - C)$$

$$c = (O - C) \cdot (O - C) - r^2$$

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Note that if the discriminant,  $b^2 - 4ac$  is negative, the ray misses the sphere completely. Also, if both  $t$  values are negative, the sphere is behind the ray.

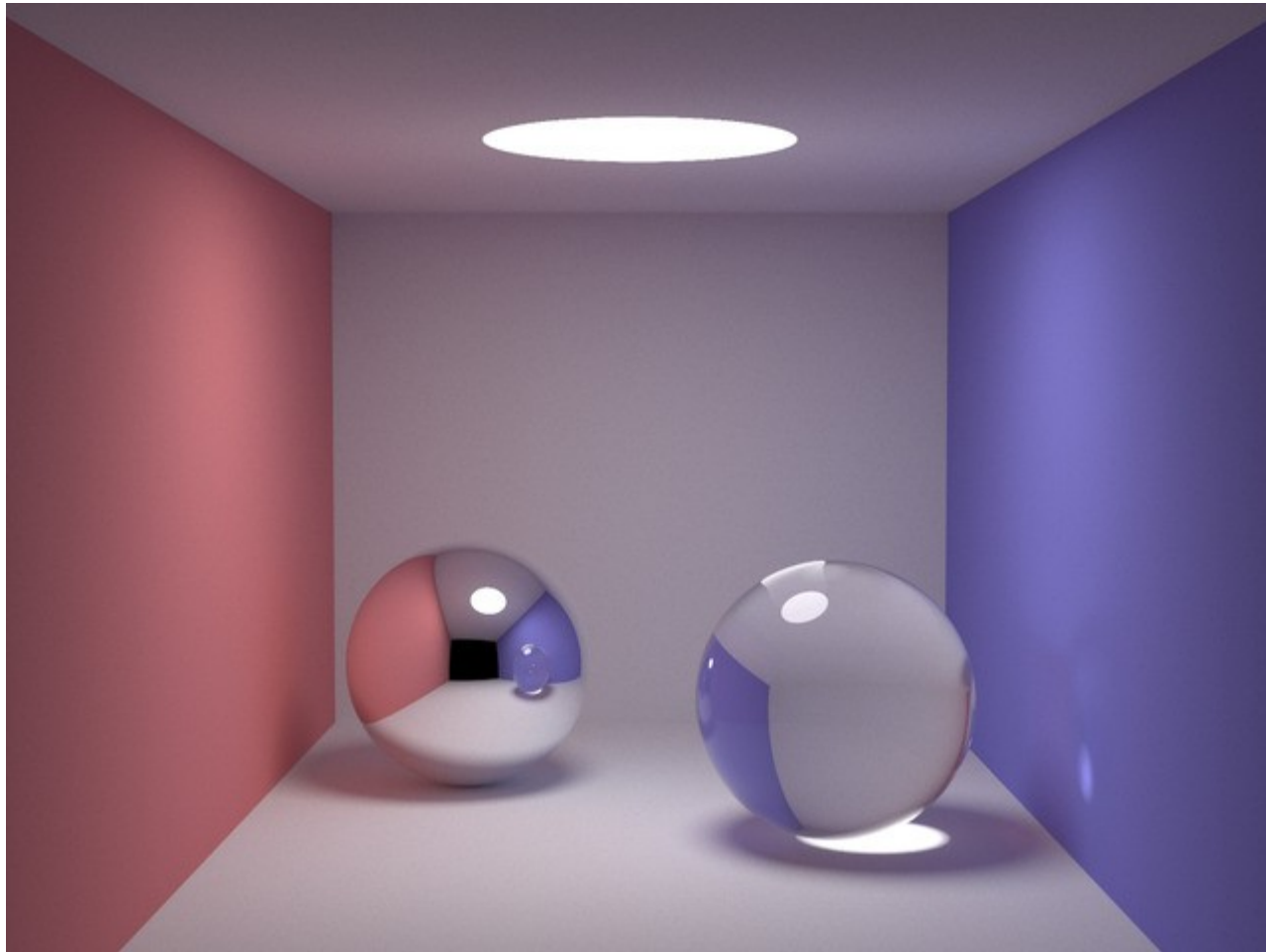
# Intersection Routine

```
// returns distance, 0 if nohit
double intersect(const Ray &r) const {
    // Solve  $t^2 \cdot d \cdot d + 2 \cdot t \cdot (o-p) \cdot d + (o-p) \cdot (o-p) - R^2 = 0$ 
    Vec op = p-r.o; // p is sphere center (C)
    double t, eps = 1e-4; // eps is a small fudge factor
    double b = op.dot(r.d); // 1/2 b from quadratic eq. setup
    double det = b*b-op.dot(op)+rad*rad; // (b^2-4ac)/4: a=1 because ray normalized
    if (det<0) return 0; // ray misses sphere
    else det = sqrt(det);
    return (t=b-det)>eps ? t : ((t=b+det)>eps ? t : 0); // return smaller positive t
}
```

# Full Sphere Code

```
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  $t^2 \cdot d \cdot d + 2 \cdot t \cdot (o-p) \cdot d + (o-p) \cdot (o-p) - R^2 = 0$ 
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 quadratic 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 };
61
```

# The Scene



# The Scene Description

```
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
```

# Convert Colors to Displayable Range

- The output of the “radiance” function is a set of unbounded colors. This has to be converted to be between 0 and 255 for display purposes. The following functions do this. The “toInt” function applies a gamma correction of 2.2.

```
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

- Check each sphere, one at a time. Keep the closest intersection.

```
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
```

# End Part 1

# The main Function

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

# main (1)

```
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
```

# main (1a: set up 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 }
```

# main (1b: set up 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 (x) camera direction

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

(assumes upright camera)

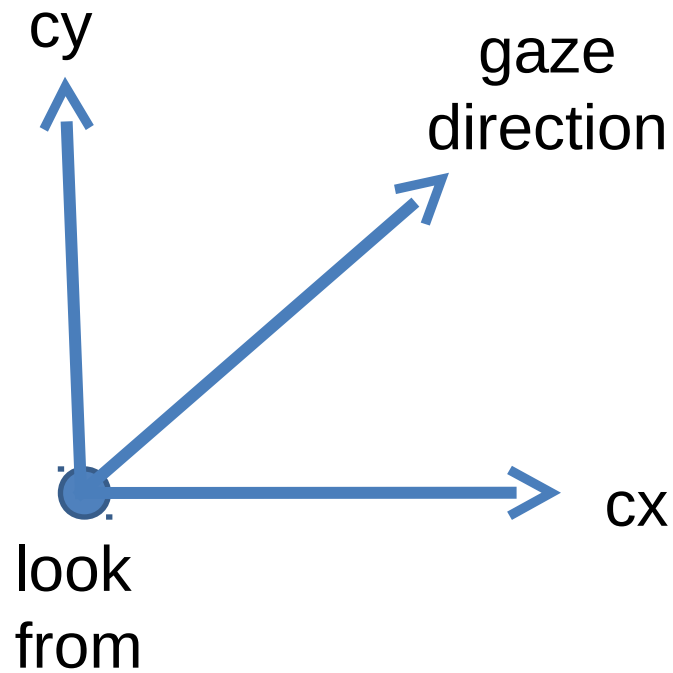
(0.5135 defines field of view angle)

- Vertical (vup) vector of the camera

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

(cross product gets vector perpendicular to both cx and gaze direction)

# Camera Setup





# main (2: Create Image)

```
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))*0.25;
209             }
210     }
211 }
```

# main (2a: OpenMP directive)

```
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))*0.25;
209             }
210     }
211 }
```

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

# main (2b: Loop over image 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))*0.25;
209             }
210     }
211 }
```

Loop over all pixels in the image.

# main (2c: Subpixels & samples)

```
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))*0.25;
209             }
210     }
211 }
```

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

# main (2d: Pixel Index)

```
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))*0.25;
209             }
210     }
211 }
```

Calculate array index for pixel(x,y)

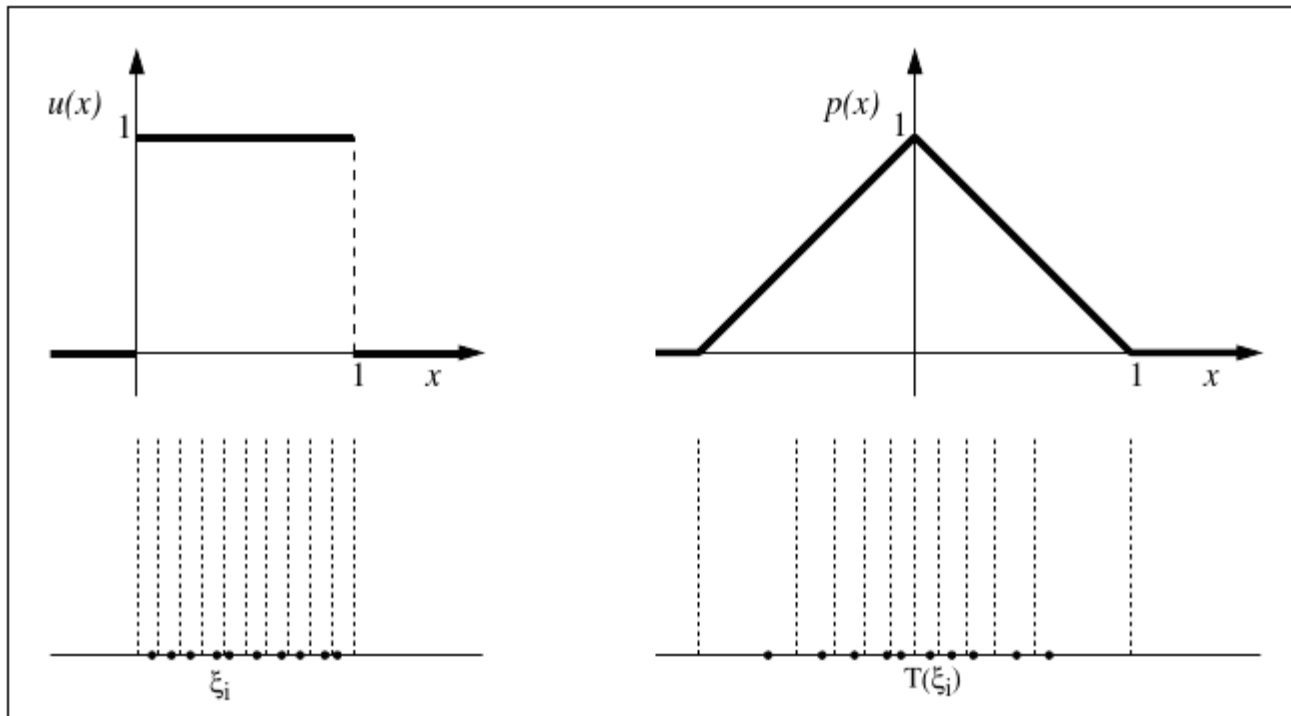
# main (2e: Tent Filter)

```
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))*0.25;
209             }
210     }
211 }
```

r1 and r2 are random values of a tent filter  
(Determine location of sample within pixel)

# Tent Filter

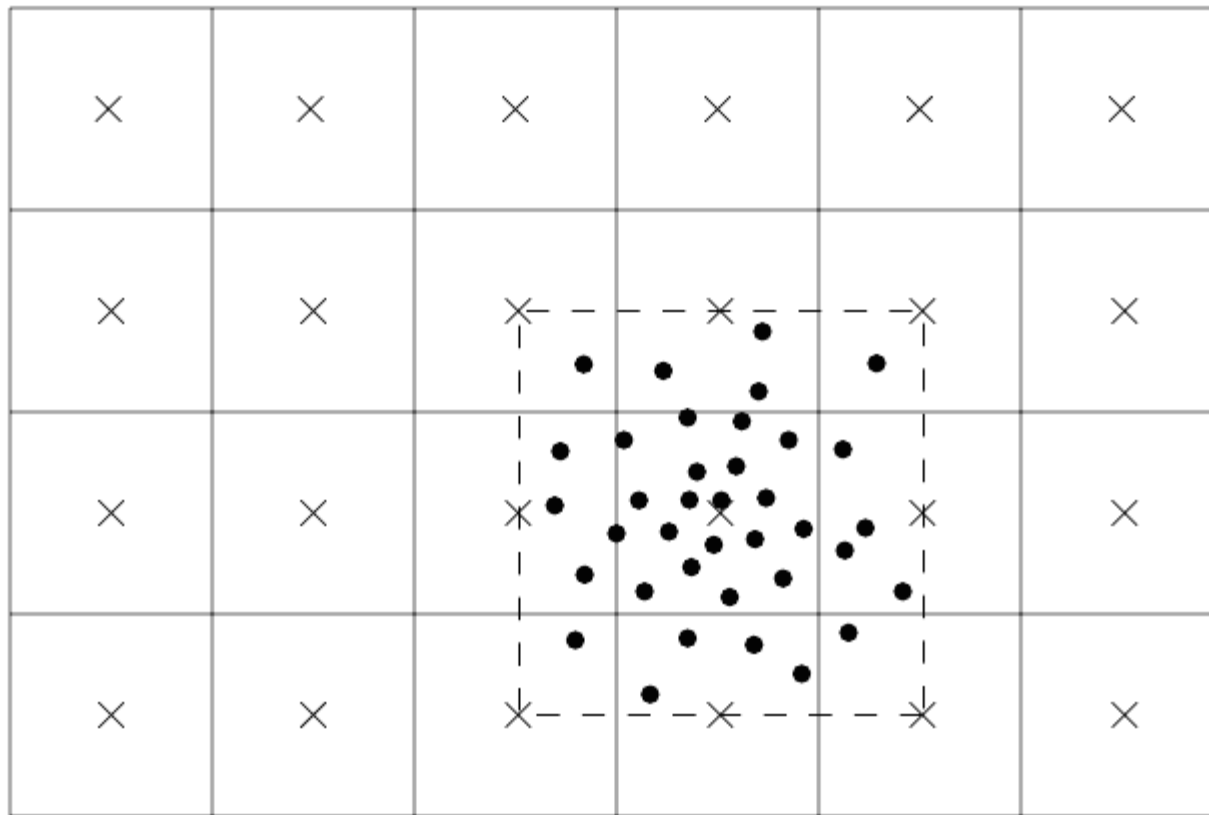
- From Realistic Ray Tracing (Shirley and van Dam)



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

# Tent Filter

- From Realistic Ray Tracing (Shirley and Morley)



**Figure 3.7.**  $n$  can be used to create an estimate for pixel color.



# main (2f: Ray direction & radiance)

```
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))*0.25;
209             }
210     }
211 }
```

Compute ray direction using cam.d, cx, cy  
Use radiance function to estimate radiance

# main (2g: Add subpixel estimate)

```
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))*0.25;
209             }
210     }
211 }
```

Add the gamma-corrected subpixel color estimate to the Pixel color  $c[i]$

# main (3: Write PPM image)

```
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 (1: do intersection)

```
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

# radiance (2: surface properties)

```
110 Vec x=r.o+r.d*t; // ray intersection point
111 Vec n=(x-obj.p).norm(); // sphere normal
112 Vec n1=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

- When a ray hits a glass surface, the ray tracer must determine if it is entering or exiting glass to compute the refraction ray.
- The dot product of the normal and ray direction tells this:

```
Vec nl=n.dot(r.d)<0?n:n*-1;
```

# Russian Roulette

- Stop the recursion randomly based on the surface reflectivity.
  - Use the maximum component (r,g,b) of the surface color.
  - Don't do Russian Roulette until after depth 5

```
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
```

# Diffuse Reflection

- For diffuse (not shiny) reflection
  - Sample all lights (non-recursive)
  - Send out additional random sample (recursive)



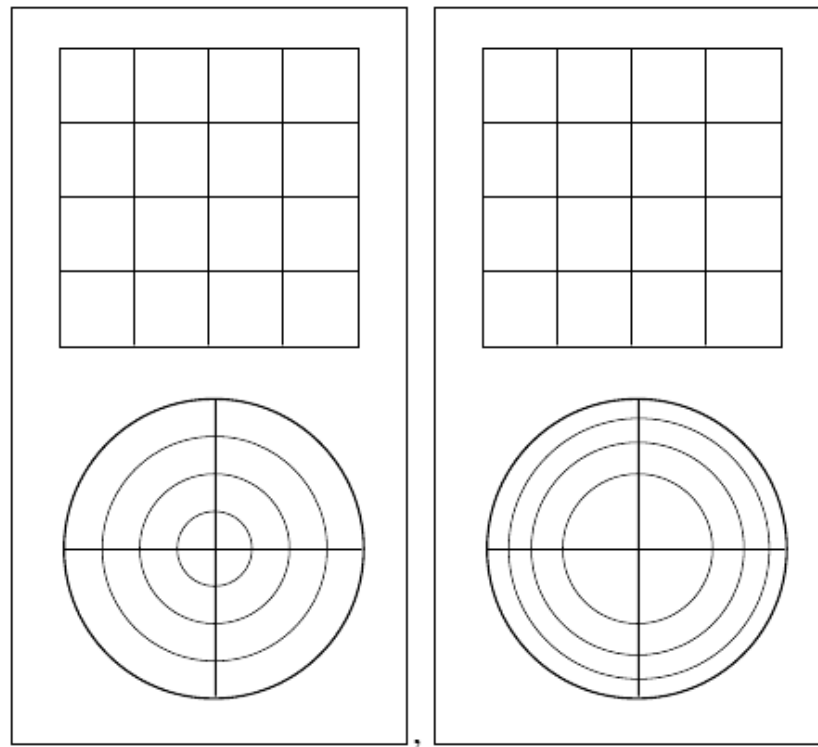
# Diffuse Reflection

```
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 = n1; // 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
```

- Construct random ray:
  - Get random angle (r1)
  - Get random distance from center (r2s)
  - Use normal to create orthonormal coordinate frame (w,u,v)

# Sampling Unit Disk

- From Realistic Ray Tracing (Shirley and van Dam)



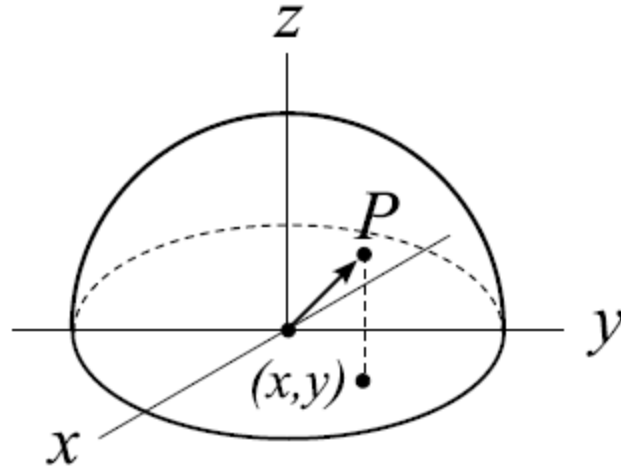
**Figure 10.1.** Left: The transform that takes the horizontal and vertical dimensions uniformly to  $(r, \phi)$  does not preserve relative area; not all of the resulting areas are the same. Right: An area-preserving map.

# Sampling Unit Hemisphere

$$w=z$$

$$u=x$$

$$v=y$$



126

```
Vec d = (u*cos(r1)*r2s + v*sin(r1)*r2s + w*sqrt(1-r2)).norm();
```

# Sampling Lights

```
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 shadow 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 Sphere by Solid Angle

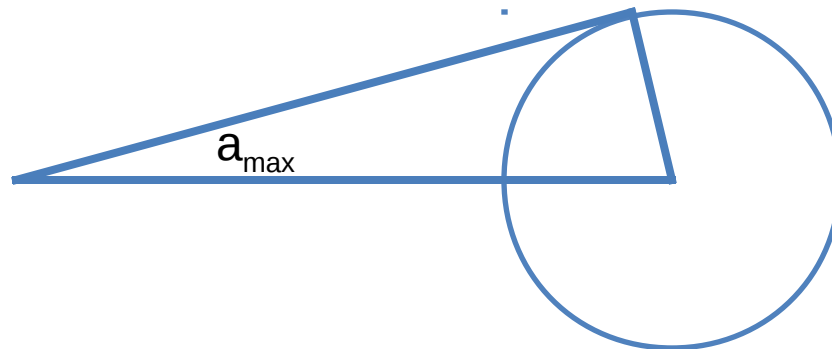
```
134 // Create random direction towards sphere using method from Realistic Ray Tracing
135 Vec sw=s.p-x, su=((fabs(sw.x)>.1?Vec(0,1):Vec(1))%sw).norm(), sv=sw%su;
136 double cos_a_max = sqrt(1-s.rad*s.rad/(x-s.p).dot(x-s.p));
137 double eps1 = erand48(Xi), eps2 = erand48(Xi);
138 double cos_a = 1-eps1+eps1*cos_a_max;
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
```

- Create coordinate system for sampling:  
SW, SU, SV

# Sampling Sphere by Solid Angle

```
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
```

- Determine max angle



# Sampling Sphere by Solid Angle

```
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
```

- Calculate sample direction based on random numbers according to equation from Realistic Ray Tracing:

$$\begin{bmatrix} \cos \alpha \\ \phi \end{bmatrix} = \begin{bmatrix} 1 + \xi_1 (\cos \alpha_{\max} - 1) \\ 2\pi \xi_2 \end{bmatrix}$$

$$\mathbf{a} = \mathbf{u} \cos \phi \sin \alpha + \mathbf{v} \sin \phi \sin \alpha + \mathbf{w} \cos \alpha.$$

# Shadow Ray

```
144 // Shoot shadow 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 }
```

- 145: Check for occlusion with shadow ray
- 146: Compute 1/probability with respect to solid angle
- 147: Calculate lighting and add to current value



# Diffuse Recursive Call

- Make recursive call with random ray direction computed earlier:

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

- Note that the 0 parameter at the end turns off the emissive term at the next recursion level.

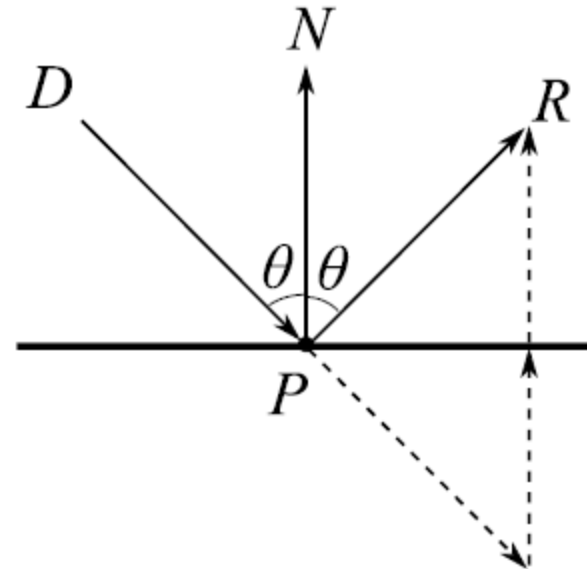
# Ideal Specular (Mirror) Reflection

```
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
```

# Ideal Specular (Mirror) Reflection

```
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
```

- Reflected Ray:
  - Angle of incidence = Angle of reflection



$$R = D - 2(N \cdot D)N$$

# Glass (Dielectric)

```
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 }
```

# Reflected Ray & Orientation

```
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, ddh=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));
```

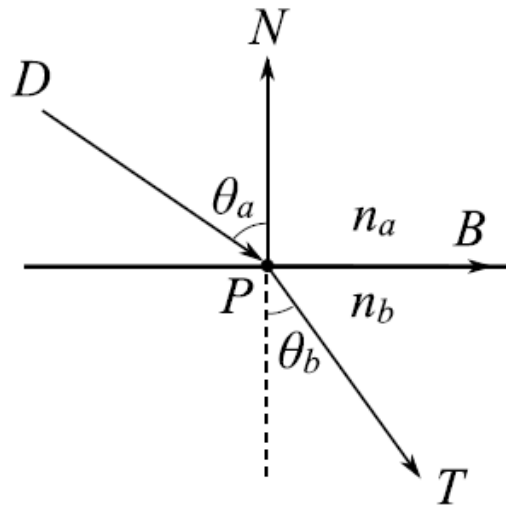
- Total internal reflection occurs when the light ray attempts to leave glass at too shallow an angle.
- If the angle is too shallow, all the light is reflected.

# Reflect or Refract using Fresnel Term

```
167 // OTHERWISE, CHOOSE REFLECTION OR REFRACTION
168 Vec tdir = (r.d*nnt - n*((into?-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);
```

- Compute the refracted ray

# Refraction Ray



$$B = \frac{D - |D \cdot N|N}{\sqrt{1 - (D \cdot N)^2}}$$

$$\sin \theta_a = \sqrt{1 - (D \cdot N)^2}$$

$$\sin \theta_b = \frac{n_a}{n_b} \sqrt{1 - (D \cdot N)^2}$$

$$\cos \theta_b = \sqrt{1 - \sin^2 \theta_b}$$

$$T = B \sin \theta_b - N \cos \theta_b$$

$$T = \frac{n_a(D + N(D \cdot N))}{n_b} - N \sqrt{1 - \frac{n_a^2(1 - (D \cdot N)^2)}{n_b^2}}$$

168

```
Vec tdir = (r.d*nnt - n*((into?-1:-1)*(ddn*nnt+sqrt(cos2t)))) .norm();
```



# Refractive Index

- Refractive index gives the speed of light within a medium compared to the speed of light within a vacuum:

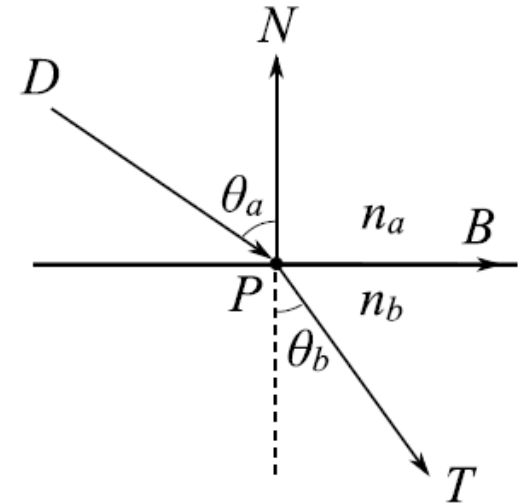
Water: 1.33

Plastic: 1.5

Glass: 1.5 – 1.7

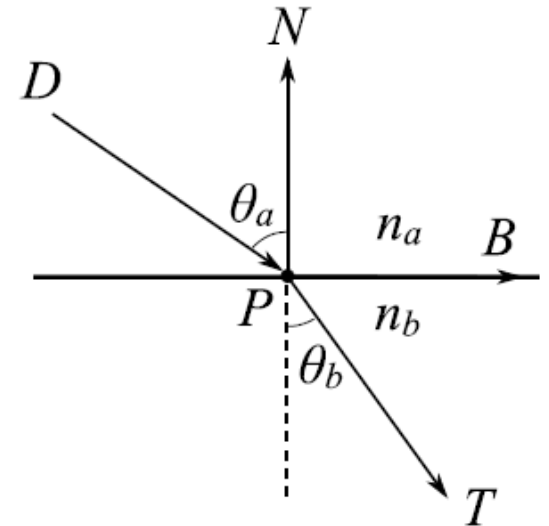
Diamond: 2.5

Note that this does not account for dispersion (prisms). To account for these, vary index by wavelength.



# Fresnel Reflectance

- Percentage of light is reflected (and what refracted) from a glass surface based on incident angle ( $\theta_a$ )



- Reflectance at “normal incidence”,  $F_0 = \frac{(n - 1)^2}{(n + 1)^2}$  /  $n_b$
- Reflectance at other angles:  

$$Fr(\theta) = F_0 + (1 - F_0)(1 - \cos \theta)^5$$

# Reflect or Refract using Fresnel Term

```
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);
```

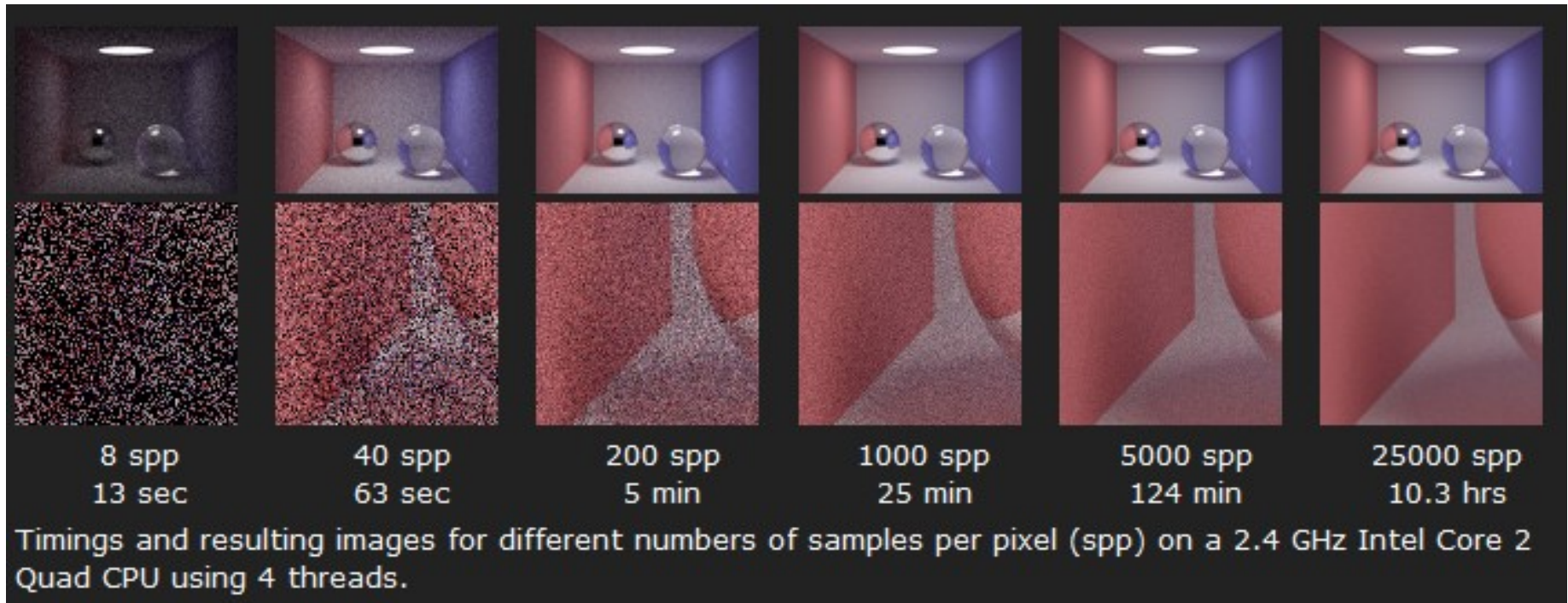
- Fresnel Reflectance
  - $R_0$  = reflectance at normal incidence based on IOR
  - $c = 1 - \cos(\theta)$
  - $R_e$  = fresnel reflectance

# Reflect or Refract using Fresnel Term

```
167 // OTHERWISE, CHOOSE REFLECTION OR REFRACTION
168 Vec tdir = (r.d*nnt - n*((into?-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 reflecting
- Finally, make 1 or 2 recursive calls
  - Make 2 if depth is  $\leq 2$
  - Make 1 randomly if depth  $> 2$

# Convergence



From: <http://kevinbeason.com/smallpt/>