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

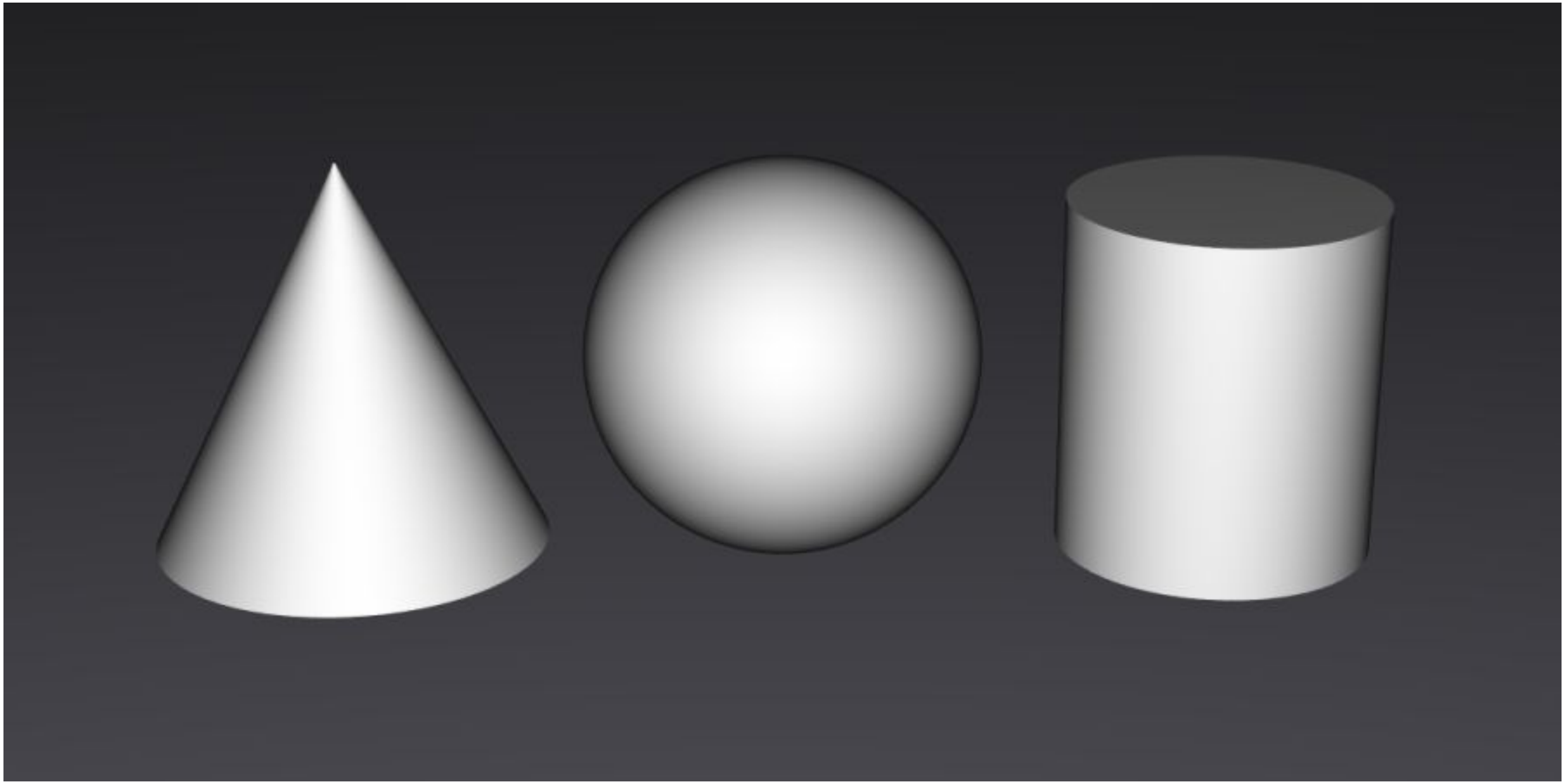
## a ray tracer by Kevin Beason http://kevinbeason.com/smallpt/

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

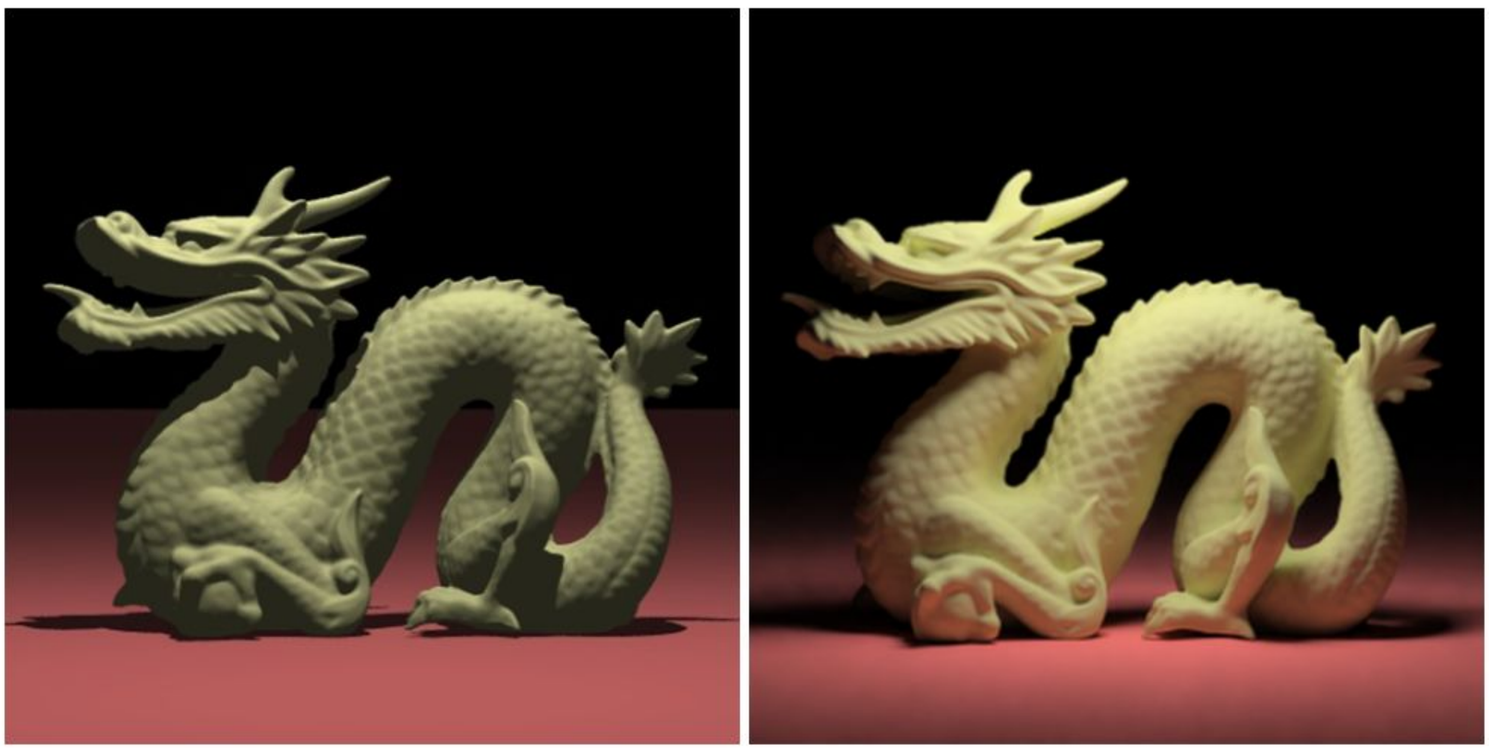
- “Ambient occlusion” = direct lighting of a cloudy day.

- Ambient Occlusion and depth of field

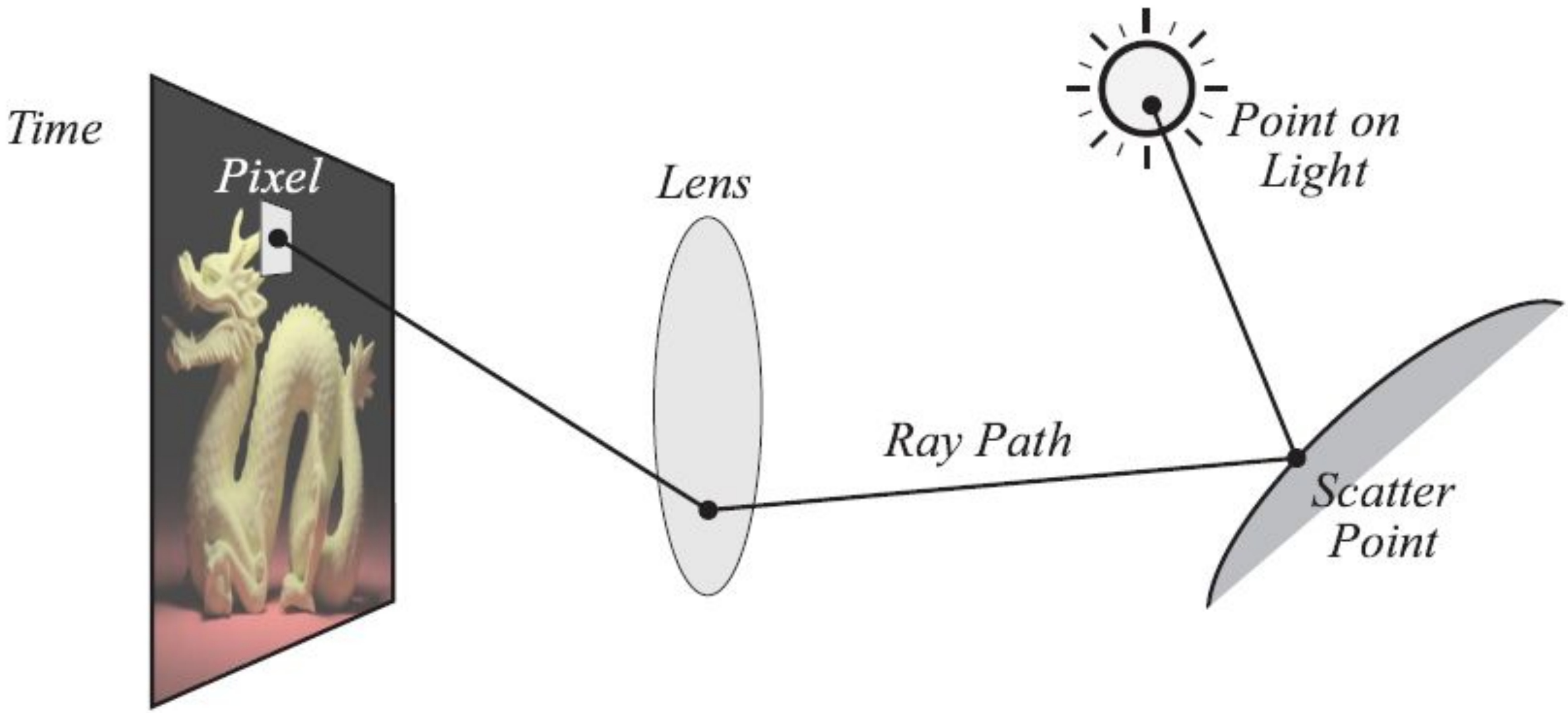
- Global illumination showing different surface types, glass surfaces, caustics (light concentrations), and depth of field.

# Another Example

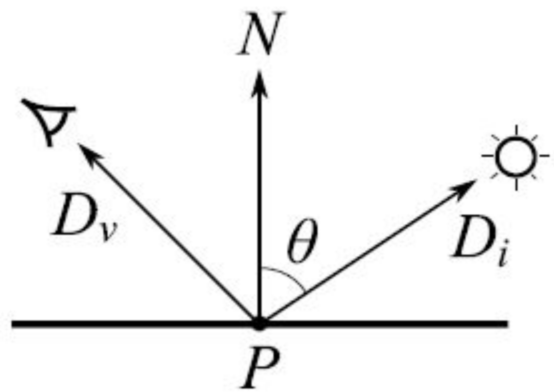
- Ad-hoc Lighting vs. Global Illumination



# How to form a GI image?

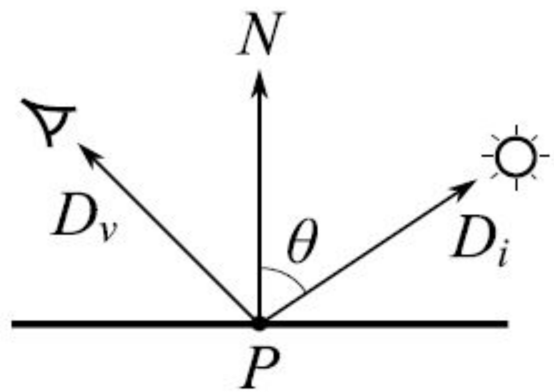


# The Rendering Equation




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

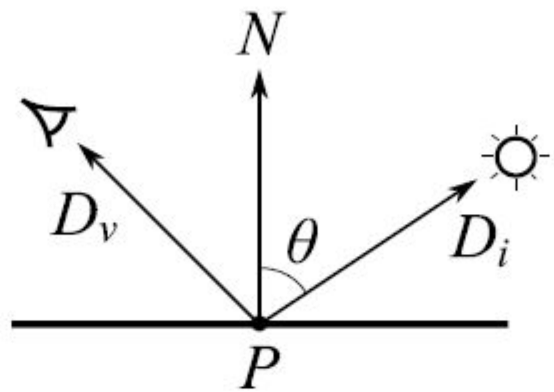
# The Rendering Equation




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

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

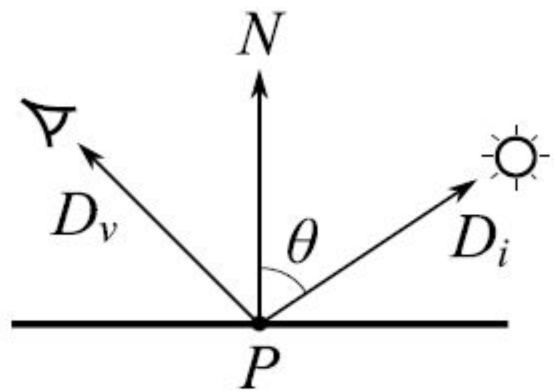
# The Rendering Equation




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

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.

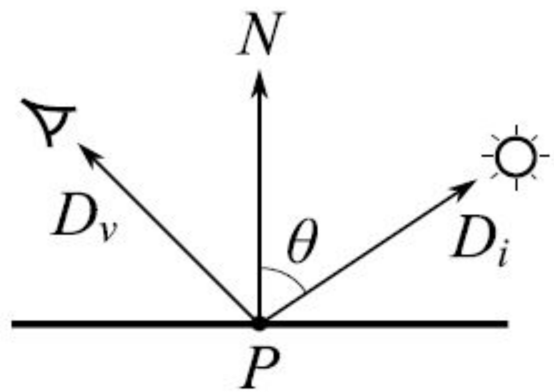
# The Rendering Equation




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

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




$$\widehat{L}(P \to D_v) = L_e(P \to D_v) + \frac{F_s(D_v, D_i)|\cos\theta|\widehat{L}(Y_i \to -D_i)}{p^{tot}_{angle}(D_i)}$$

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

**for** each pixel (i,j) **do**
    Vec3 $C = 0$
    **for** (k=0; k $<$ samplesPerPixel; k++) **do**
        Create random ray in pixel:
            Choose random point on lens $P_{lens}$
            Choose random point on image plane $P_{image}$
            $D$ = normalize($P_{image} - P_{lens}$)
            Ray ray = Ray($P_{lens}$, $D$)
        castRay(ray, isect)
        **if** the ray hits something **then**
            $C$ += radiance(ray, isect, 0)
        **else**
            $C$ += backgroundColor($D$)
        **end if**
    **end for**
    image(i,j) = $C$ / samplesPerPixel
**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:

```
#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;
}
```

# Squashed Code 2:

```
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));
}
```

# Expanded version (1)
# Preliminaries

```
// smallpt, a Path Tracer by Kevin Beason, 2009
// Make : g++ -O3 -fopenmp explicit.cpp -o explicit
//         Remove "-fopenmp" for g++ version < 4.2
// Reformatted by David Cline for illustrative purposes

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

double M_PI = 3.1415926535;
double M_1_PI = 1.0 / M_PI;
double erand48(unsigned short xsubi[3]) {
    return (double)rand() / (double)RAND_MAX;
}
```

# Expanded version (2)
# Vec (Points, Vectors, Colors)

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

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

# Normalize

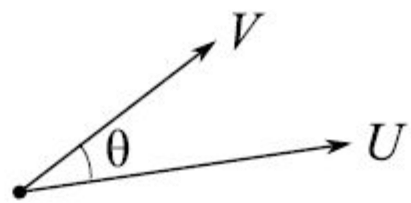
- “Normalize” a vector = divide by its length

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

```
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:



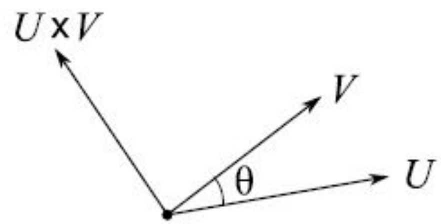

$$U \cdot V = (u_x v_x + u_y v_y + u_z v_z) = \|U\| \|V\| \cos\theta$$

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

```
double dot(const Vec &b) const     { return x*b.x+y*b.y+z*b.z; }
```

# 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:




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

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

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).

```
Vec operator%(Vec&b)    { return Vec(y*b.z-z*b.y,z*b.x-x*b.z,x*b.y-y*b.x); }
```

# 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: $P(t) = O + tD$
- In code we have:

```
// Ray STRUCTURE
struct Ray {
    Vec o, d;
    Ray(Vec o_, Vec d_) : o(o_), d(d_) {}
};

```

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

# Sphere

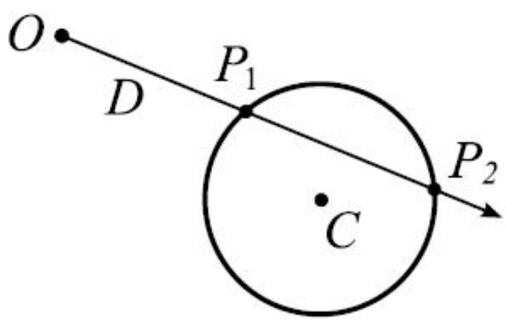
- SmallPT supports sphere objects only
- We can define a sphere based on
  - a center point, C
  - Radius, r
- The equation of the sphere:

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

- 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*d.d + 2*t*(o-p).d + (o-p).(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 quadradtic 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

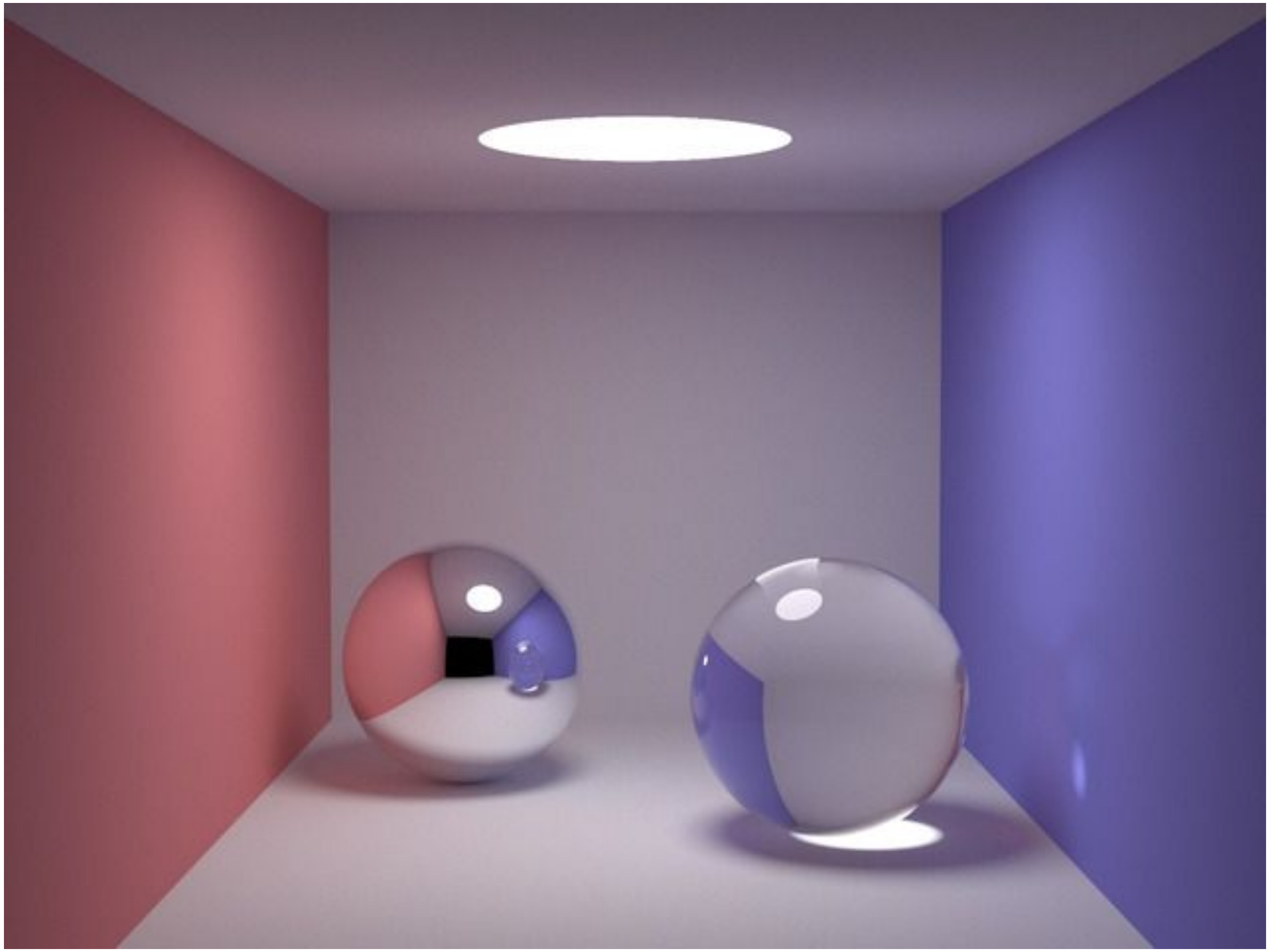
```
// ENUM OF MATERIAL TYPES USED IN radiance FUNCTION
enum Refl_t { DIFF, SPEC, REFR };

// SMALLPT ONLY SUPPORTS SPHERES
struct Sphere {
    double rad;       // radius
    Vec p, e, c;      // position, emission, color
    Refl_t refl;      // reflection type (DIFFuse, SPECular, REFRactive)

    // constructor
    Sphere(double rad_, Vec p_, Vec e_, Vec c_, Refl_t refl_):
        rad(rad_), p(p_), e(e_), c(c_), refl(refl_) {}

    // returns distance, 0 if nohit
    double intersect(const Ray &r) const {
        // Solve t^2*d.d + 2*t*(o-p).d + (o-p).(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 quadradtic 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
    }
};
```

# The Scene



# The Scene Description

```
// HARD CODED SCENE DESCRIPTION
// THE SCENE DESCRIPTION CONSISTS OF A BUNCH OF SPHERES
// Scene: radius, position, emission, color, material
Sphere spheres[] = {
    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(1.5, Vec(50,81.6-16.5,81.6),Vec(4,4,4)*100,  Vec(), DIFF),//Lite
};
int numSpheres = sizeof(spheres)/sizeof(Sphere);
```

# 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.

```
// CLAMP FUNCTION
inline double clamp(double x){ return x<0 ? 0 : x>1 ? 1 : x; }

// CONVERTS FLOATS TO INTEGERS TO BE SAVED IN PPM FILE
inline int toInt(double x){ return int(pow(clamp(x),1/2.2)*255+.5); }

```

# Intersect Ray with Scene

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

```
// INTERSECTS RAY WITH SCENE
inline bool intersect(const Ray &r, double &t, int &id){
    double n=sizeof(spheres)/sizeof(Sphere);
    double d;
    double inf=t=1e20;

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

```

# 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)

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

```

# main (1a: set up image)

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

```

# main (1b: set up camera)

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

```

# Camera Setup

- Look from and gaze direction:

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

- Horizontal (x) camera direction

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

(assumes upright camera)

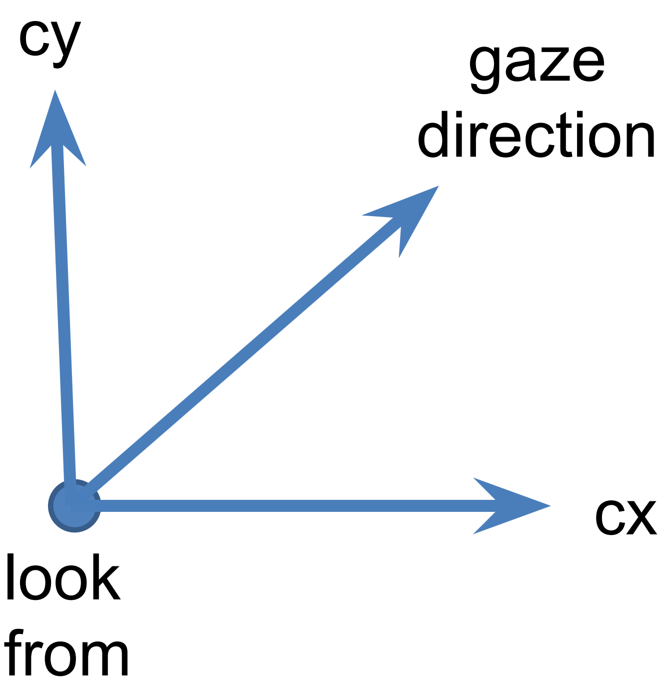
(0.5135 defines field of view angle)

- Vertical (vup) vector of the camera

```
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)

```
#pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP

// LOOP OVER ALL IMAGE PIXELS
for (int y=0; y<h; y++) { // Loop over image rows
  fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1)); // print progress
  unsigned short Xi[3]={0,0,y*y*y};
  for (unsigned short x=0; x<w; x++)   // Loop columns

    // FOR EACH PIXEL DO 2x2 SUBSAMPLES, AND samps SAMPLES PER SUBSAMPLE
    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++){
          // I BELIEVE THIS PRODUCES A TENT FILTER
          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;
      }
}
```

# main (2a: OpenMP directive)

```
#pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP

// LOOP OVER ALL IMAGE PIXELS
for (int y=0; y<h; y++) { // Loop over image rows
  fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1)); // print progress
  unsigned short Xi[3]={0,0,y*y*y};
  for (unsigned short x=0; x<w; x++)   // Loop columns

    // FOR EACH PIXEL DO 2x2 SUBSAMPLES, AND samps SAMPLES PER SUBSAMPLE
    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++){
          // I BELIEVE THIS PRODUCES A TENT FILTER
          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;
      }
}
```

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

# main (2b: Loop over image pixels)

```
#pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP

// LOOP OVER ALL IMAGE PIXELS
for (int y=0; y<h; y++) { // Loop over image rows
  fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1)); // print progress
  unsigned short Xi[3]={0,0,y*y*y};
  for (unsigned short x=0; x<w; x++)   // Loop columns

    // FOR EACH PIXEL DO 2x2 SUBSAMPLES, AND samps SAMPLES PER SUBSAMPLE
    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++){
          // I BELIEVE THIS PRODUCES A TENT FILTER
          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;
      }
}
```

Loop over all pixels in the image.

# main (2c: Subpixels & samples)

```
#pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP

// LOOP OVER ALL IMAGE PIXELS
for (int y=0; y<h; y++) { // Loop over image rows
  fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1)); // print progress
  unsigned short Xi[3]={0,0,y*y*y};
  for (unsigned short x=0; x<w; x++)   // Loop columns

    // FOR EACH PIXEL DO 2x2 SUBSAMPLES, AND samps SAMPLES PER SUBSAMPLE
    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++){
          // I BELIEVE THIS PRODUCES A TENT FILTER
          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;
      }
}
```

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

# main (2d: Pixel Index)

```
#pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP

// LOOP OVER ALL IMAGE PIXELS
for (int y=0; y<h; y++) { // Loop over image rows
  fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1)); // print progress
  unsigned short Xi[3]={0,0,y*y*y};
  for (unsigned short x=0; x<w; x++)   // Loop columns

    // FOR EACH PIXEL DO 2x2 SUBSAMPLES, AND samps SAMPLES PER SUBSAMPLE
    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++){
          // I BELIEVE THIS PRODUCES A TENT FILTER
          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;
      }
}
```

Calculate array index for pixel(x,y)

# main (2e: Tent Filter)

```
#pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP

// LOOP OVER ALL IMAGE PIXELS
for (int y=0; y<h; y++) { // Loop over image rows
  fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1)); // print progress
  unsigned short Xi[3]={0,0,y*y*y};
  for (unsigned short x=0; x<w; x++)   // Loop columns

    // FOR EACH PIXEL DO 2x2 SUBSAMPLES, AND samps SAMPLES PER SUBSAMPLE
    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++){
          // I BELIEVE THIS PRODUCES A TENT FILTER
          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;
      }
}
```

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

# Tent Filter

- From Realistic Ray Tracing (Shirley and Mo

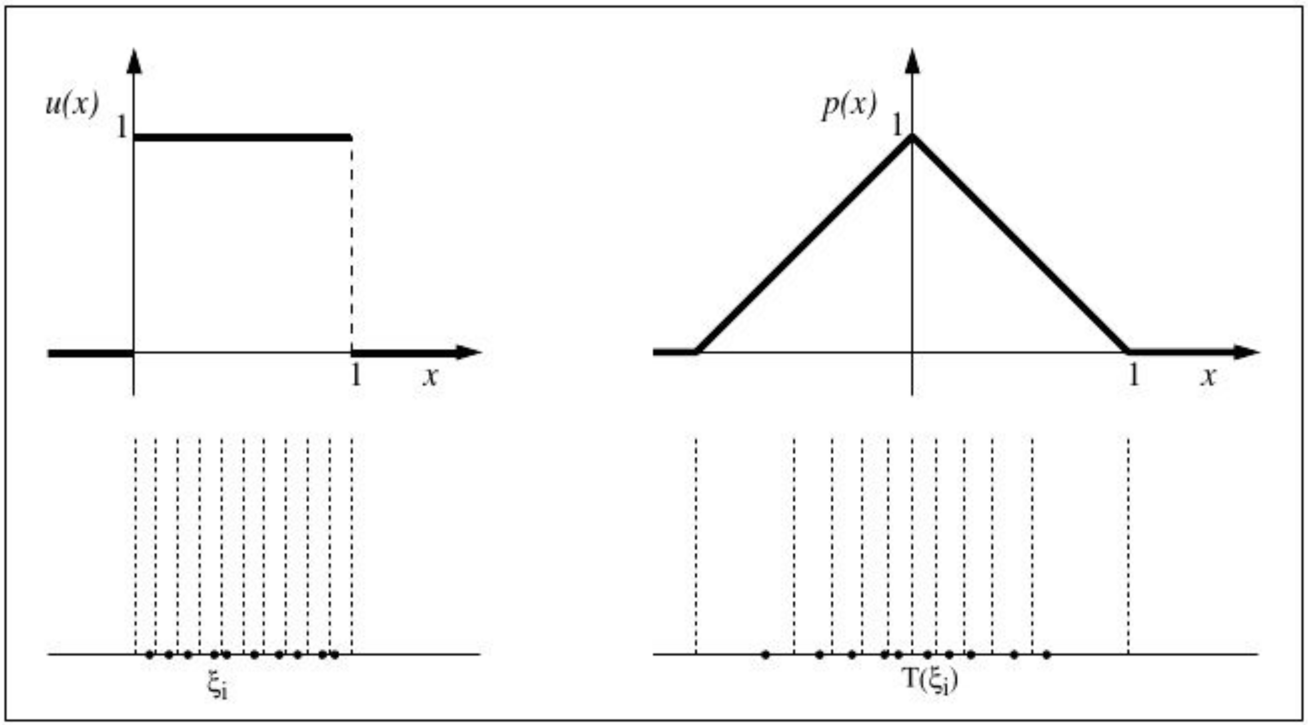



**Figure 3.8.** We can take a set of canonical random samples and transform them to nonuniform samples.

# Tent Filter

- From Realistic Ray Tracing (Shirley and Mor

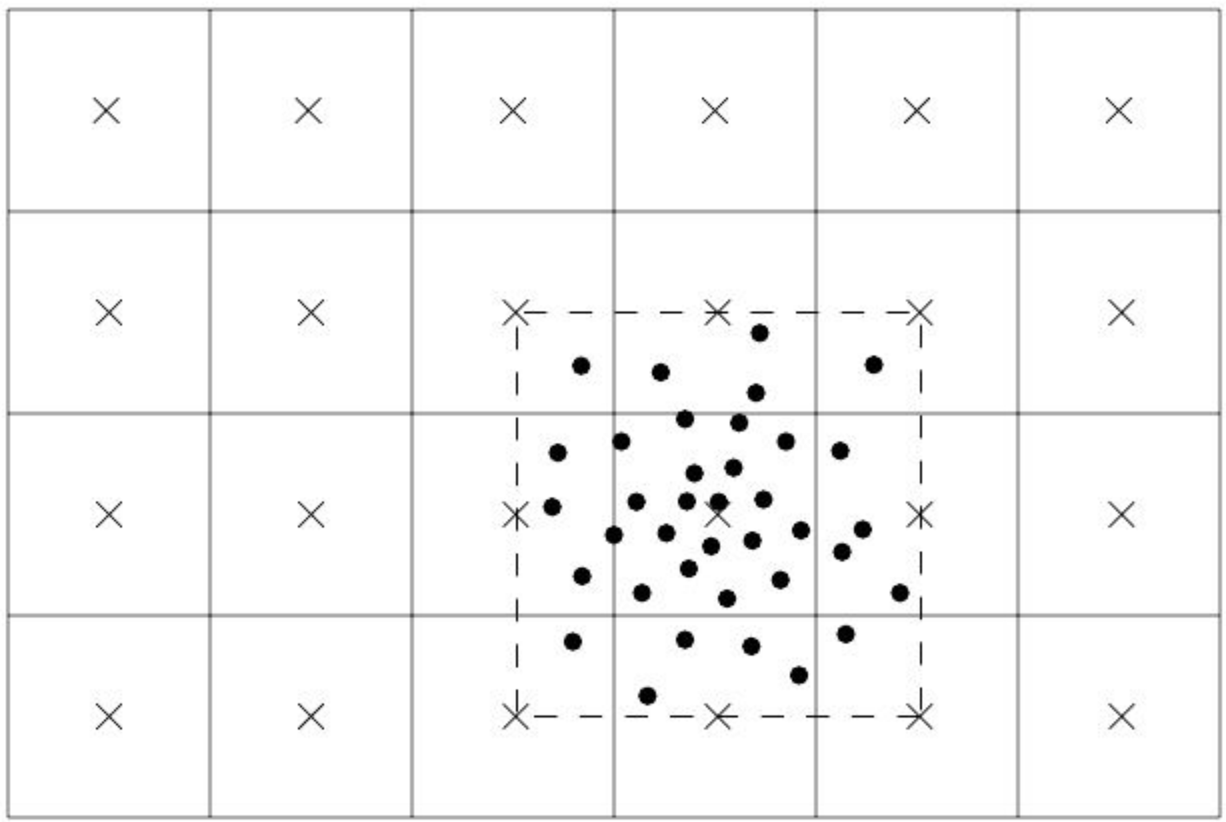


**Figure 3.7.** n can be used to create a estimate for pixel color.

# main (2f: Ray direction & radiance)

```
#pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP

// LOOP OVER ALL IMAGE PIXELS
for (int y=0; y<h; y++) { // Loop over image rows
  fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1)); // print progress
  unsigned short Xi[3]={0,0,y*y*y};
  for (unsigned short x=0; x<w; x++)   // Loop columns

    // FOR EACH PIXEL DO 2x2 SUBSAMPLES, AND samps SAMPLES PER SUBSAMPLE
    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++){
          // I BELIEVE THIS PRODUCES A TENT FILTER
          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;
      }
}
```

Compute ray direction using cam.d, cx, cy

Use radiance function to estimate radiance

# main (2g: Add subpixel estimate)

```
#pragma omp parallel for schedule(dynamic, 1) private(r) // OpenMP

// LOOP OVER ALL IMAGE PIXELS
for (int y=0; y<h; y++) { // Loop over image rows
  fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1)); // print progress
  unsigned short Xi[3]={0,0,y*y*y};
  for (unsigned short x=0; x<w; x++)   // Loop columns

    // FOR EACH PIXEL DO 2x2 SUBSAMPLES, AND samps SAMPLES PER SUBSAMPLE
    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++){
          // I BELIEVE THIS PRODUCES A TENT FILTER
          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;
      }
}
```

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

# main (3: Write PPM image)

```
// WRITE OUT THE FILE TO A PPM
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));
}
```

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)

```
// COMPUTES THE RADIANCE ESTIMATE ALONG RAY R
//
Vec radiance(const Ray &r, int depth, unsigned short *Xi,int E=1)
{
  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

  if (depth>10) return Vec();

```

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)

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

```

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

```
// Use maximum reflectivity amount for Russian roulette
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||!p) if (erand48(Xi)<p) f=f*(1/p); else return obj.e*E;

```

# Diffuse Reflection

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

# Diffuse Reflection

```
// IDEAL DIFFUSE REFLECTION
if (obj.refl == DIFF){                  // Ideal DIFFUSE reflection
  double r1=2*M_PI*erand48(Xi); // angle around
  double r2=erand48(Xi), r2s=sqrt(r2); // distance from center
  Vec w = nl; // w = normal
  Vec u = ((fabs(w.x)>.1?Vec(0,1):Vec(1))%w).norm(); // u is perpendicular to w
  Vec v = w%u; // v is perpendicular to u and w
  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 Morle `

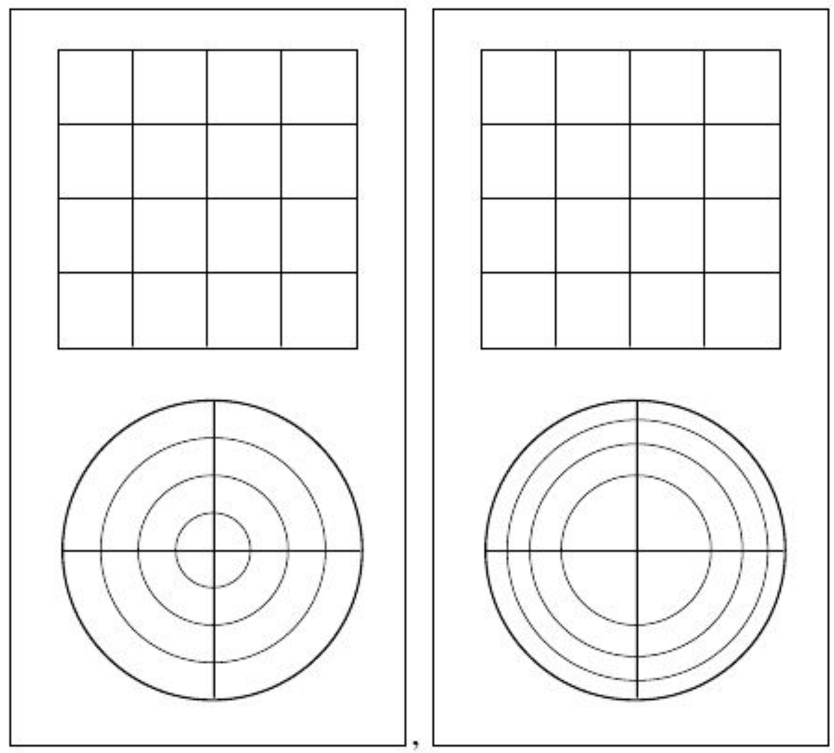
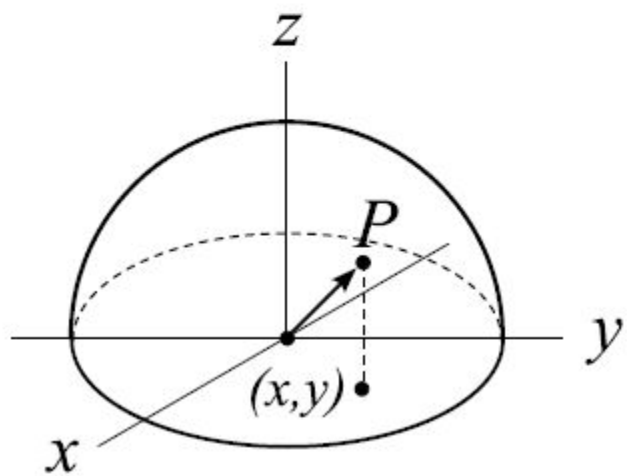


**Figure 10.1.** Left: The transform that takes the horizontal and vertical dimensions uniformly to $(r, \phi)$ does not preserve relative area; not all of the resulting areas are the same. Right: An area-preserving map.

# Sampling Unit Hemisphere

w=z
u=x
v=y




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

# Sampling Lights

```
// Loop over any lights (explicit lighting)
Vec e;
for (int i=0; i<numSpheres; i++){
  const Sphere &s = spheres[i];
  if (s.e.x<=0 && s.e.y<=0 && s.e.z<=0) continue; // skip non-lights

  // Create random direction towards sphere using method from Realistic Ray Tracing
  Vec sw=s.p-x, su=((fabs(sw.x)>.1?Vec(0,1):Vec(1))%sw).norm(), sv=sw%su;
  double cos_a_max = sqrt(1-s.rad*s.rad/(x-s.p).dot(x-s.p));
  double eps1 = erand48(Xi), eps2 = erand48(Xi);
  double cos_a = 1-eps1+eps1*cos_a_max;
  double sin_a = sqrt(1-cos_a*cos_a);
  double phi = 2*M_PI*eps2;
  Vec l = su*cos(phi)*sin_a + sv*sin(phi)*sin_a + sw*cos_a;
  l.norm();

  // Shoot shadodw ray
  if (intersect(Ray(x,l), t, id) && id==i){  // shadow ray
    double omega = 2*M_PI*(1-cos_a_max);
    e = e + f.mult(s.e*l.dot(nl)*omega)*M_1_PI;  // 1/pi for brdf
  }
}
```

# Sampling Sphere by Solid Angle

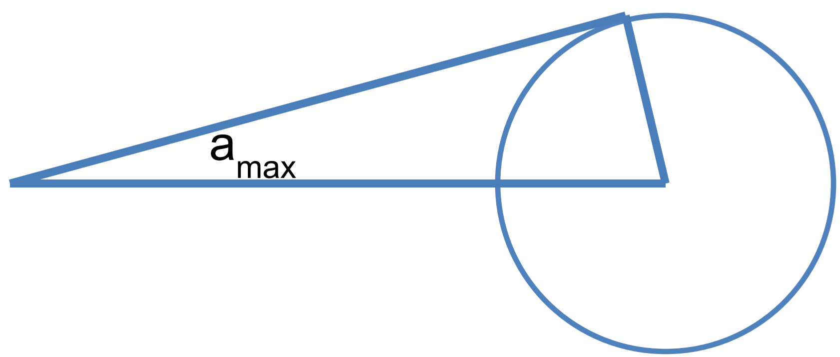
```
// Create random direction towards sphere using method from Realistic Ray Tracing
Vec sw=s.p-x, su=((fabs(sw.x)>.1?Vec(0,1):Vec(1))%sw).norm(), sv=sw%su;
double cos_a_max = sqrt(1-s.rad*s.rad/(x-s.p).dot(x-s.p));
double eps1 = erand48(Xi), eps2 = erand48(Xi);
double cos_a = 1-eps1+eps1*cos_a_max;
double sin_a = sqrt(1-cos_a*cos_a);
double phi = 2*M_PI*eps2;
Vec l = su*cos(phi)*sin_a + sv*sin(phi)*sin_a + sw*cos_a;
l.norm();
```

- Create coordinate system for sampling: sw, su, sv

# Sampling Sphere by Solid Angle

```
// Create random direction towards sphere using method from Realistic Ray Tracing
Vec sw=s.p-x, su=((fabs(sw.x)>.1?Vec(0,1):Vec(1))%sw).norm(), sv=sw%su;
double cos_a_max = sqrt(1-s.rad*s.rad/(x-s.p).dot(x-s.p));
double eps1 = erand48(Xi), eps2 = erand48(Xi);
double cos_a = 1-eps1+eps1*cos_a_max;
double sin_a = sqrt(1-cos_a*cos_a);
double phi = 2*M_PI*eps2;
Vec l = su*cos(phi)*sin_a + sv*sin(phi)*sin_a + sw*cos_a;
l.norm();
```

- Determine max angle



# Sampling Sphere by Solid Angle

```
// Create random direction towards sphere using method from Realistic Ray Tracing
Vec sw=s.p-x, su=((fabs(sw.x)>.1?Vec(0,1):Vec(1))%sw).norm(), sv=sw%su;
double cos_a_max = sqrt(1-s.rad*s.rad/(x-s.p).dot(x-s.p));
double eps1 = erand48(Xi), eps2 = erand48(Xi);
double cos_a = 1-eps1+eps1*cos_a_max;
double sin_a = sqrt(1-cos_a*cos_a);
double phi = 2*M_PI*eps2;
Vec l = su*cos(phi)*sin_a + sv*sin(phi)*sin_a + sw*cos_a;
l.norm();
```

- 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 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     }
```

- 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:

```
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

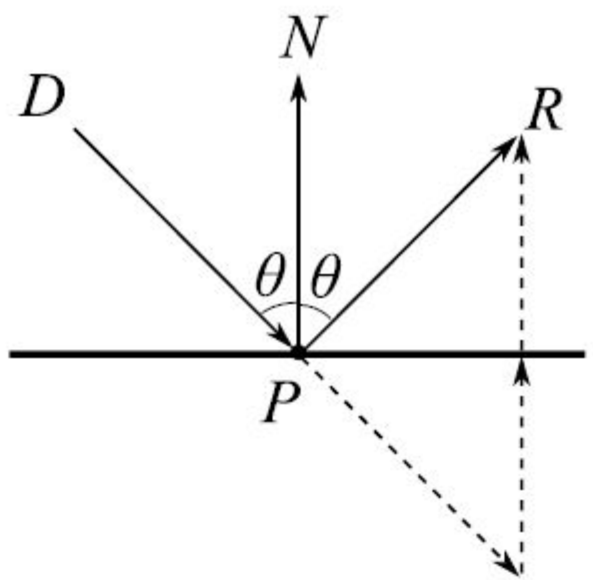
```
// IDEAL SPECULAR REFLECTION
} else if (obj.refl == SPEC) {                // Ideal SPECULAR reflection
  return obj.e + f.mult(radiance(Ray(x,r.d-n*2*n.dot(r.d)),depth,Xi));
}

```

# Ideal Specular (Mirror) Reflection

```
// IDEAL SPECULAR REFLECTION
} 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));
}
```

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




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

# Glass (Dielectric)

```
    // OTHERWISE WE HAVE A DIELECTRIC (GLASS) SURFACE
    Ray reflRay(x, r.d-n*2*n.dot(r.d));     // Ideal dielectric REFLECTION
    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 TOTAL INTERNAL REFLECTION, REFLECT
    if ((cos2t=1-nnt*nnt*(1-ddn*ddn))<0)    // Total internal reflection
      return obj.e + f.mult(radiance(reflRay,depth,Xi));

    // OTHERWISE, CHOOSE REFLECTION OR REFRACTION
    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);
}
```

# 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, 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

```
// IF TOTAL INTERNAL REFLECTION, REFLECT
if ((cos2t=1-nnt*nnt*(1-ddn*ddn))<0)    // Total internal reflection
  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.

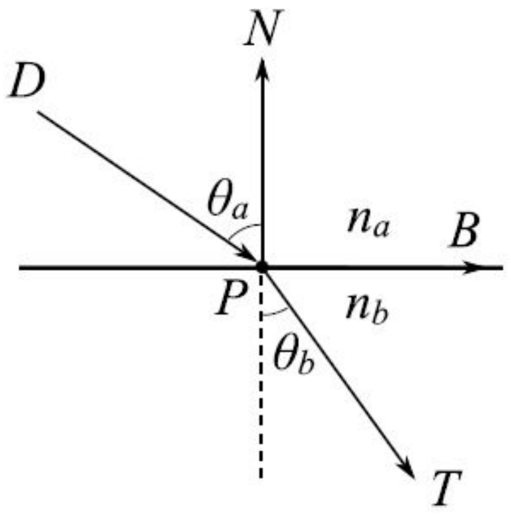
- If the angle is too shallow, all the light is reflected.

# Reflect or Refract using Fresnel Term

```
// OTHERWISE, CHOOSE REFLECTION OR REFRACTION
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);
```

- Compute the refracted ray

# Refraction Ray




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

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

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

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

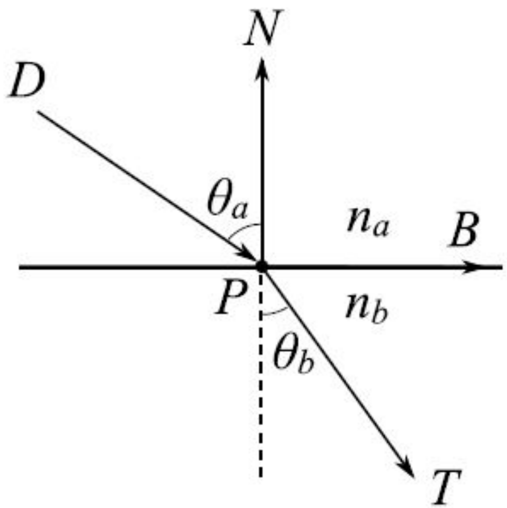
$$T = B \sin \theta_b - N \cos \theta_b$$

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

```
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

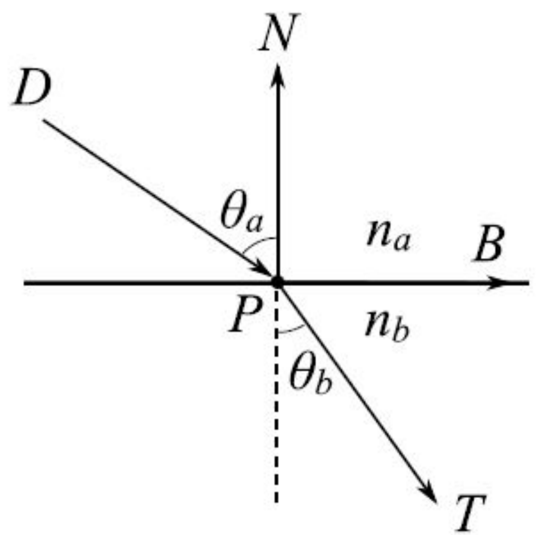
Glass: 1.5 – 1.7

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", where ($n=n_a/n_b$)

$$F_0 = \frac{(n-1)^2}{(n+1)^2}$$

- Reflectance at other angles:

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

# Reflect or Refract using Fresnel Term

```
// OTHERWISE, CHOOSE REFLECTION OR REFRACTION
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);
```

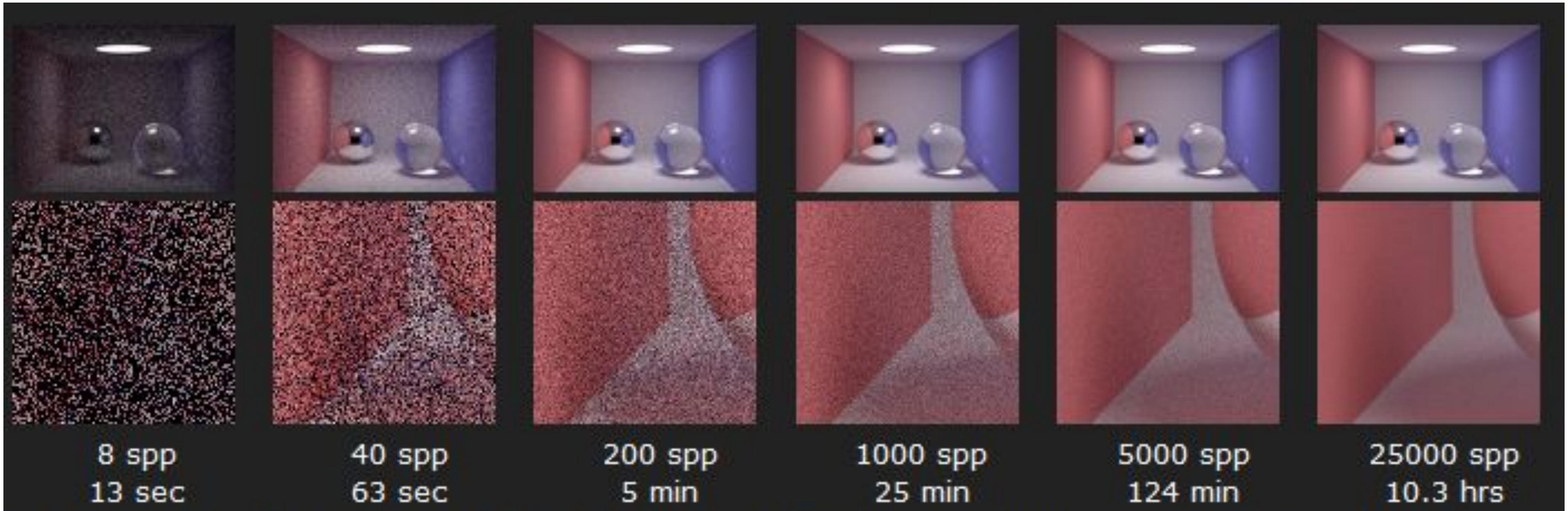
- Fresnel Reflectance
  - R0 = reflectance at normal incidence based on IOR
  - c = 1-cos(theta)
  - Re = fresnel reflectance

# Reflect or Refract using Fresnel Term

```
// OTHERWISE, CHOOSE REFLECTION OR REFRACTION
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);
```

- P = probability of reflecting
- Finally, make 1 or 2 recursive calls
  - Make 2 if depth is <= 2
  - Make 1 randomly if depth > 2

# Convergence




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.

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