Ray Tracing: The Rest of Your Life

Peter Shirley

Version 1.123

Pay What You Want. ptrshrl@gmail.com at paypal
50% of all payments donated to programming education related not-for-profits

Chapter 0: Overview

In *Ray Tracing In One Weekend* and *Ray Tracing: the Next Week*, you built a “real” ray tracer.

In this volume, I assume you will be pursuing a career related to ray tracing and we will dive into the math of creating a very serious ray tracer. When you are done you should be ready to start messing with the many serious commercial ray tracers underlying the movie and product design industries. There are many many things I do not cover in this short volume; I dive into only one of many ways to write a Monte Carlo rendering program. I don’t do shadow rays (instead I make rays more likely to go toward lights), bidirectional methods, Metropolis methods, or photon mapping. What I do is speak in the language of the field that studies those methods. I think of this book as a deep exposure that can be your first of many, and it will equip you with some of the concepts, math, and terms you will need to study the others.

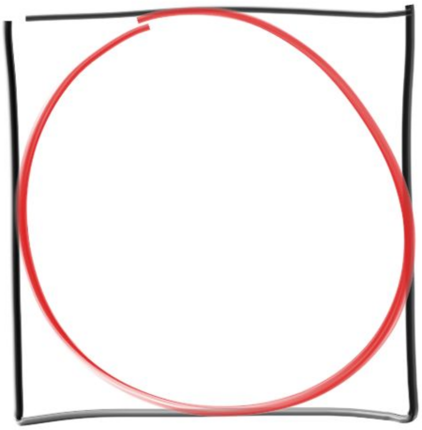
As before, www.in1weekend.com will have further readings and references.

Acknowledgements: thanks to Dave Hart and Jean Buckley for help on the original manuscript. Thanks to Paul Melis, Nakata Daisuke, Filipe Scur, Vahan Sosoyan, and Matthew Heimlich for finding errors.

Chapter 1: A Simple Monte Carlo Program

Let’s start with one of the simplest Monte Carlo (MC) programs. MC programs give a statistical estimate of an answer, and this estimate gets more and more accurate the longer you run it. This basic characteristic of simple programs producing noisy but ever-better answers is what MC is all about, and it is especially good for applications like graphics where great accuracy is not needed.

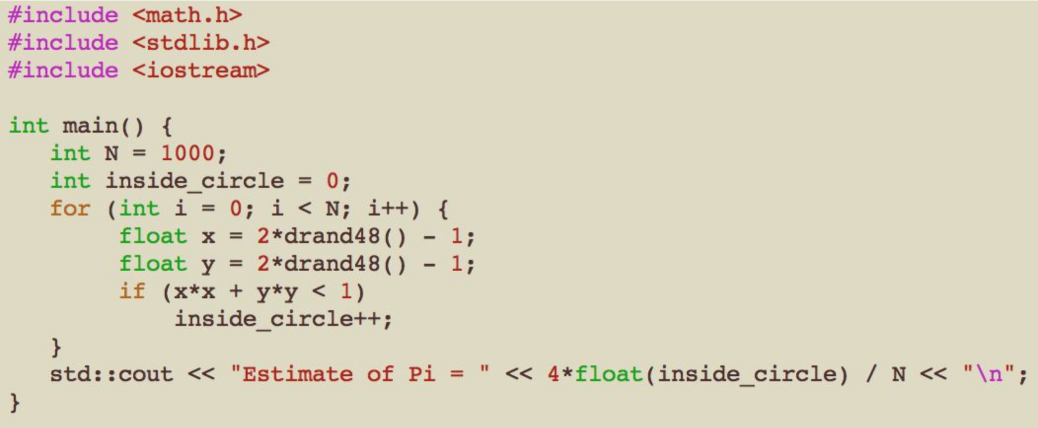
As an example, let’s estimate *Pi*. There are many ways to do this, with the Buffon Needle problem being a classic case study. We’ll do a variation inspired by that.   Suppose you have a circle inscribed inside a square:



Now, suppose you pick random points inside the square. The fraction of those random points that end up inside the circle should be proportional to the area of the circle. The exact fraction should in fact be the ratio of the circle area to the square area.

Fraction = *(Pi*R*R) / ((2R)*(2R)) = Pi/4*

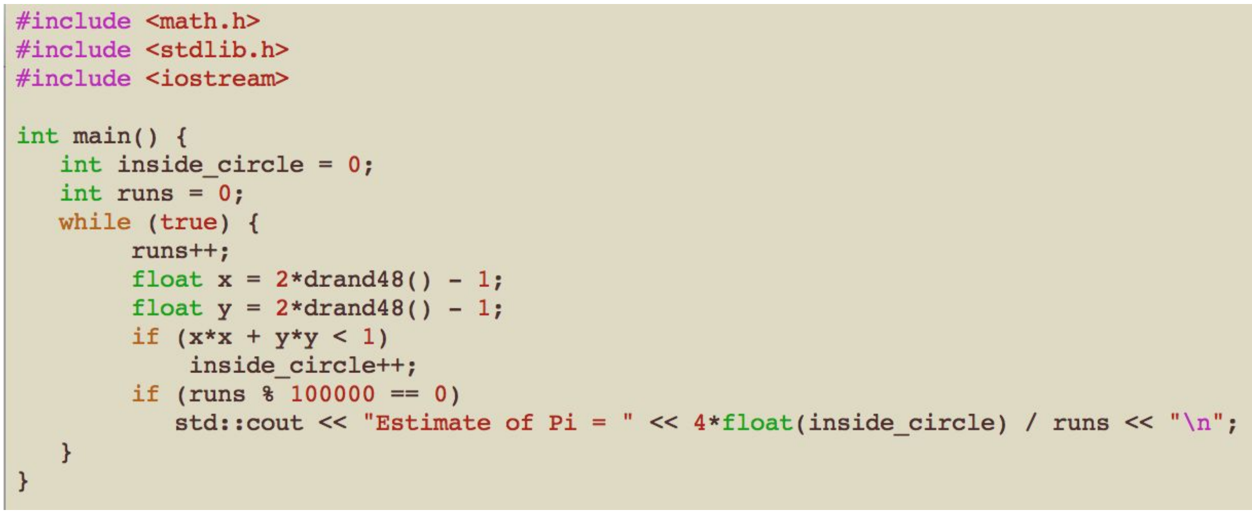
Since the *R* cancels out, we can pick whatever is computationally convenient. Let’s go with *R*=1 centered at the origin:



```cpp
#include <math.h>
#include <stdlib.h>
#include <iostream>

int main() {
   int N = 1000;
   int inside_circle = 0;
   for (int i = 0; i < N; i++) {
        float x = 2*drand48() - 1;
        float y = 2*drand48() - 1;
        if (x*x + y*y < 1)
            inside_circle++;
   }
   std::cout << "Estimate of Pi = " << 4*float(inside_circle) / N << "\n";
}
```

This gives me the answer *Estimate of Pi = 3.196*

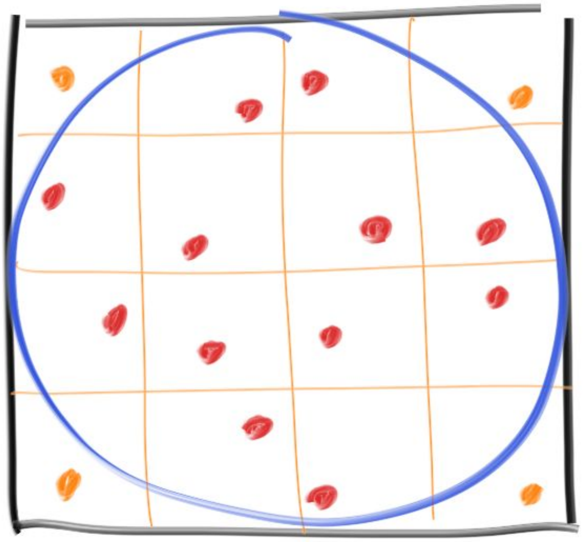
If we change the program to run forever and just print out a running estimate:



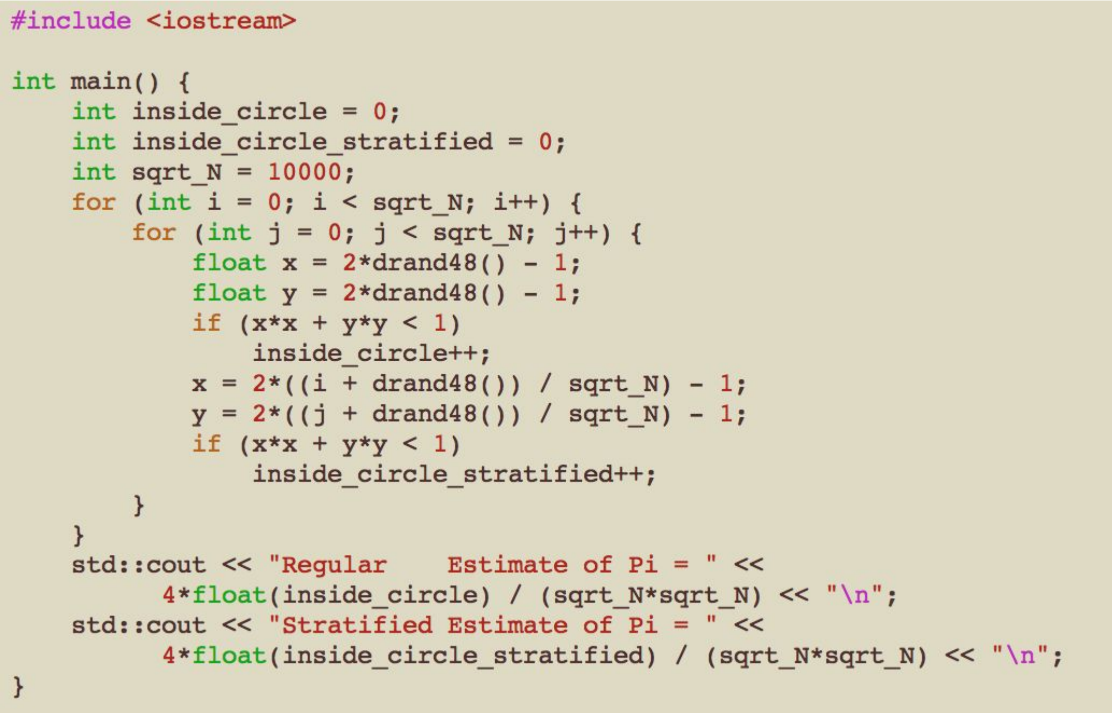
```cpp
#include <math.h>
#include <stdlib.h>
#include <iostream>

int main() {
   int inside_circle = 0;
   int runs = 0;
   while (true) {
        runs++;
        float x = 2*drand48() - 1;
        float y = 2*drand48() - 1;
        if (x*x + y*y < 1)
            inside_circle++;
        if (runs % 100000 == 0)
           std::cout << "Estimate of Pi = " << 4*float(inside_circle) / runs << "\n";
   }
}
```

We get very quickly near Pi, and then more slowly zero in on it. This is an example of the *Law of Diminishing Returns*, where each sample helps less than the last. This is the worst part of MC. We can mitigate this diminishing return by *stratifying* the samples (often called *jittering*), where instead of taking random samples, we take a grid and take one sample within each:

I get:



This changes the sample generation, but we need to know how many samples we are taking in advance because we need to know the grid. Let’s take a hundred million and try it both ways:



```
#include <iostream>

int main() {
    int inside_circle = 0;
    int inside_circle_stratified = 0;
    int sqrt_N = 10000;
    for (int i = 0; i < sqrt_N; i++) {
        for (int j = 0; j < sqrt_N; j++) {
            float x = 2*drand48() - 1;
            float y = 2*drand48() - 1;
            if (x*x + y*y < 1)
                inside_circle++;
            x = 2*((i + drand48()) / sqrt_N) - 1;
            y = 2*((j + drand48()) / sqrt_N) - 1;
            if (x*x + y*y < 1)
                inside_circle_stratified++;
        }
    }
    std::cout << "Regular    Estimate of Pi = " <<
          4*float(inside_circle) / (sqrt_N*sqrt_N) << "\n";
    std::cout << "Stratified Estimate of Pi = " <<
          4*float(inside_circle_stratified) / (sqrt_N*sqrt_N) << "\n";
}
```

I get:

Regular Estimate of Pi = 3.1415
Stratified Estimate of Pi = 3.14159

Interestingly, the stratified method is not only better, it converges with a better asymptotic rate! Unfortunately, this advantage decreases with the dimension of the problem (so for example, with the 3D sphere volume version the gap would be less). This is called the *Curse of Dimensionality*. We are going to be very high dimensional (each reflection adds two dimensions), so I won't stratify in this book. But if you are ever doing single-reflection or shadowing or some strictly 2D problem, you definitely want to stratify.

Chapter 2: One Dimensional MC Integration

Integration is all about computing areas and volumes, so we could have framed Chapter 1 in an integral form if we wanted to make it maximally confusing. But sometimes integration is the most natural and clean way to formulate things. Rendering is often such a problem. Let’s look at a classic integral:

$$I = \int_0^2 x^2 dx$$

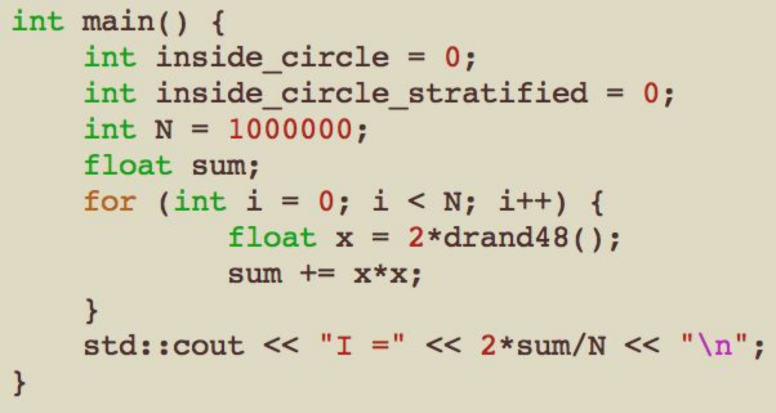
In computer sciency notation, we might write this as:

I = area(x^2, 0, 2)

We could also write it as:

*I = 2*average(x^2, 0, 2)*

This suggests a MC approach:

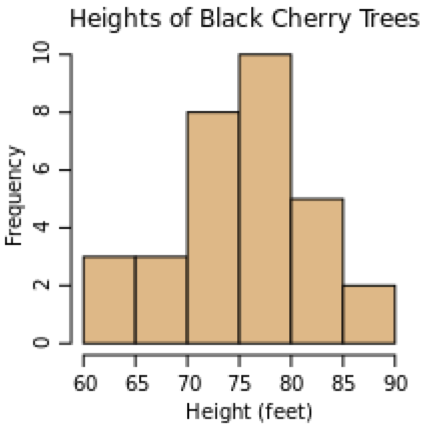


```
int main() {
    int inside_circle = 0;
    int inside_circle_stratified = 0;
    int N = 1000000;
    float sum;
    for (int i = 0; i < N; i++) {
            float x = 2*drand48();
            sum += x*x;
    }
    std::cout << "I =" << 2*sum/N << "\n";
}
```

This, as expected, produces approximately the exact answer we get with algebra, *I = 8/3*. But we could also do it for functions that we can't analytically integrate like *log(sin(x))*. In graphics, we often have functions we can evaluate but can't write down explicitly, or functions we can only probabilistically evaluate. That is in fact what the ray tracing *color()* function of the last two books is-- we don't know what color is seen in every direction, but we can statistically estimate it in any given dimension.

One problem with the random program we wrote in the first two books is small light sources create too much noise because our uniform sampling doesn't sample the light often enough. We could lessen that problem if we sent more random samples toward the light, but then we would need to downweight those samples to adjust for the over-sampling. How we do that adjustment? To do that we will need the concept of a *probability density function*.

First, what is a *density function*? It's just a continuous form of a histogram. Here's an example from the histogram Wikipedia page:



If we added data for more trees, the histogram would get taller. If we divided the data into more bins, it would get shorter. A discrete density function differs from a histogram in that it normalizes the frequency *y-axis* to a fraction or percentage (just a fraction times 100). A continuous histogram, where we take the number of bins to infinity, can't be a fraction because the height of all the bins would drop to zero. A density function is one where we take the bins and adjust them so they don't get shorter as we add more bins. For the case of the tree histogram above we might try:

Bin-height = (Fraction of trees between height H and H') / (H-H')

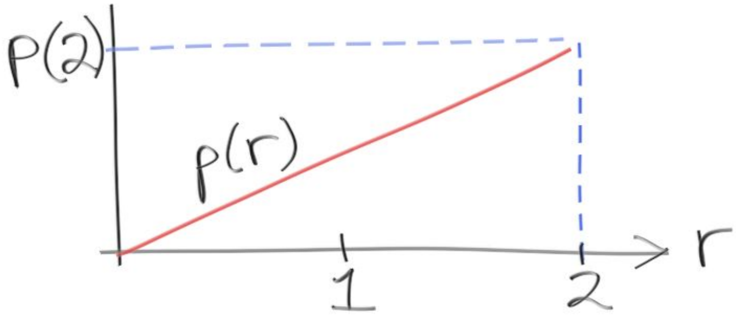
That would work! We could interpret that as a statistical predictor of a tree's height:

Probability a random tree is between H and H' = Bin-height(H-H')*

If we wanted to know about the chances of being in a span of multiple bins, we would sum.

A *probability density function*, henceforth *pdf*, is that fractional histogram made continuous.

Let's make a *pdf* and use it a bit to understand it more. Suppose I want a random number *r* between 0 and 2 whose probability is proportional to itself: *r*. We would expect the *pdf p(r)* to look something like the figure below. But how high should it be?



The height is just *p*(2). What should that be? We could reasonably make it anything by convention, and we should pick something that is convenient. Just as with histograms we can sum up (integrate) the region to figure out the probability that *r* is in some interval *(x0,x1)*:

*Probability x0 < r < x1 = C * area(p(r), x0, x1)* where C is a scaling constant. We may as well make *C = 1* for cleanliness, and that is exactly what is done in probability. And we know the probability *r* has the value 1 somewhere, so for this case

area(p(r), 0, 2) = 1.

Since *p(r)* is proportional to *r*, *i.e.*, *p = C' r* for some other constant *C'*

area(C'r, 0, 2) = INTEGRAL_0^2 C'r dr = (½)C'r^2_0^2 C'r = 2C'

So *p(r) = r/2*.

How do we generate a random number with that pdf *p(r)*? For that we will need some more machinery. Don't worry this doesn't go on forever!

Given a random number from *d = drand48()* that is uniform and between 0 and 1, we should be able to find some function f(d) that gives us what we want. Suppose *e = f(d) = d^2*. That is no longer a uniform *pdf*. The *pdf* of *e* will be bigger near 0 than it is near 1 (squaring a number between 0 and 1 makes it smaller). To take this general observation to a function, we need the cumulative probability distribution function *P(x)*:

P(x) = area(p, -infinity, x)

Note that for *x* where we didn't define *p(x), p(x) = 0*, i.e., the probability of an *x* there is zero. For our example *pdf p(r) = r/2*, the *P(x)* is:

P(x) = 0 : x < 0
P(x) = ¼ x^2 : 0 < x < 2
P(x) = 1 : x > 2

One question is, what's up with *x* versus *r*? They are dummy variables-- analogous to the function arguments in a program. If we evaluate *P* at *x = 0.5*, we get:

P(1.0) = ¼

This says *the probability that a random variable with our pdf is less than one is 25%*. This gives rise to a clever observation that underlies many methods to generate non-uniform random numbers. We want a function *f()* that when we call it as *f(drand48())* we get a return value with a pdf *¼ x*^2. We don't know what that is, but we do know that 25% of what it returns should be less than 1, and 75% should be above one. If *f()* is increasing, then we would expect *f(0.25) = 1.0.* This can be generalized to figure out *f()* for every possible input:

f(P(x)) = x

That means *f* just undoes whatever *P* does. So,

$f(x) = P^{-1}(x)$

The -1 means "inverse function". Ugly notation, but standard. For our purposes what this means is, if we have pdf *p()* and its cumulative distribution function *P()*, then if we do this to a random number we'll get what we want:

$e = P^{-1}(drand48())$

For our *pdf p(x) = x/2*, and corresponding *P(x)*, we need to compute the inverse of *P*. If we have

y = ¼ x^2

we get the inverse by solving for *x* in terms of *y*:

x = sqrt(4y)

Thus our random number with density p we get by:

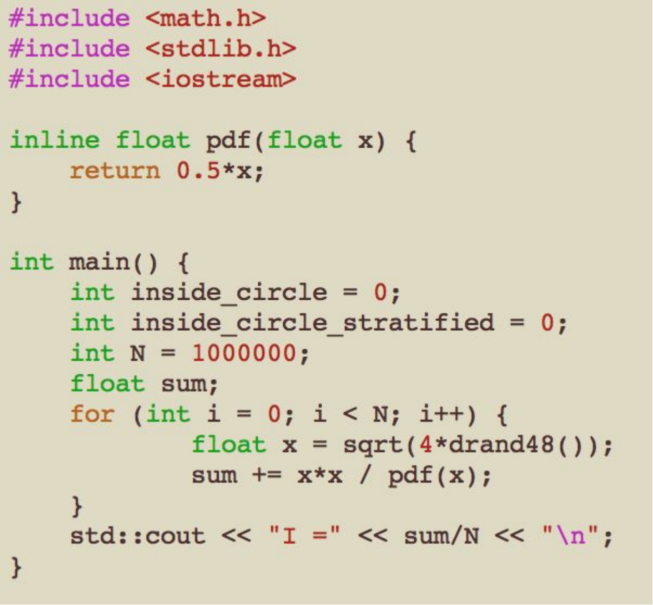
*e = sqrt(4*drand48())*

Note that does range 0 to 2 as hoped, and if we send in ¼ for *drand48()* we get 1 as desired.

We can now sample our old integral

I = integral(x^2, 0, 2)

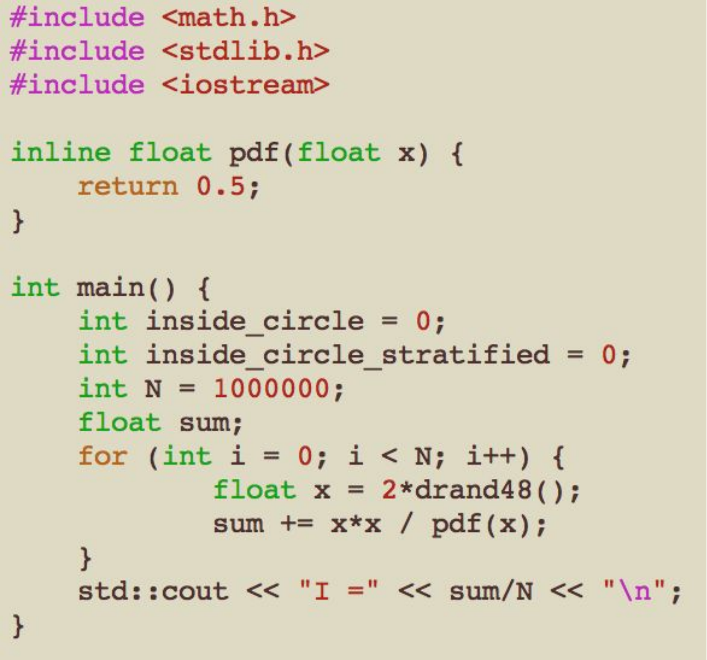
We need to account for the non-uniformity of the *pdf* of *x*. Where we sample too much we should down-weight. The *pdf* is a perfect measure of how much or little sampling is being done. So the weighting function should be proportional to *1/pdf*. In fact it is exactly *1/pdf*:



```
#include <math.h>
#include <stdlib.h>
#include <iostream>

inline float pdf(float x) {
    return 0.5*x;
}

int main() {
    int inside_circle = 0;
    int inside_circle_stratified = 0;
    int N = 1000000;
    float sum;
    for (int i = 0; i < N; i++) {
            float x = sqrt(4*drand48());
            sum += x*x / pdf(x);
    }
    std::cout << "I =" << sum/N << "\n";
}
```

SInce we are sampling more where the integrand is big, we might expect less noise and thus faster convergence. This is why using a carefully chosen non-uniform *pdf* is usually called *importance sampling*.

If we take that same code with uniform samples so the *pdf* = ½ over the range [0,2] we can use the machinery to get *x = 2*drand48()* and the code is:



```
#include <math.h>
#include <stdlib.h>
#include <iostream>

inline float pdf(float x) {
    return 0.5;
}

int main() {
    int inside_circle = 0;
    int inside_circle_stratified = 0;
    int N = 1000000;
    float sum;
    for (int i = 0; i < N; i++) {
            float x = 2*drand48();
            sum += x*x / pdf(x);
    }
    std::cout << "I =" << sum/N << "\n";
}
```

Note that we don't need that 2 in the *2*sum/N* anymore-- that is handled by the *pdf*, which is 2 when you divide by it. You'll note that importance sampling helps a little, but not a ton. We could make the *pdf* follow the integrand exactly:

$p(x) = (3/8)x^2$

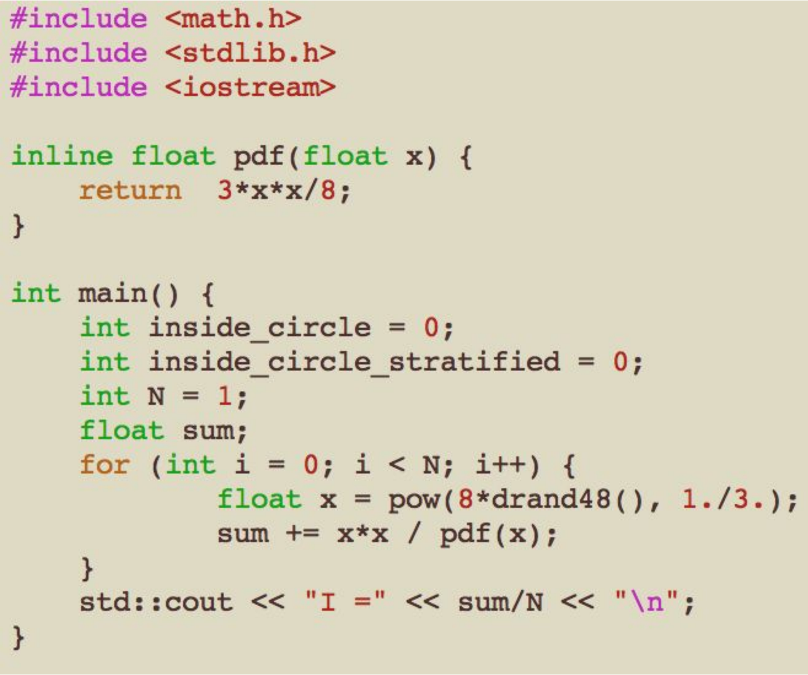
And we get the corresponding

$P(x) = (\tfrac{1}{8})x^3$

and

$P^{-1}(x) = pow(8x,\tfrac{1}{3})$

This perfect importance sampling is only possible when we already know the answer (we got *P* by integrating *p* analytically), but it's a good exercise to make sure our code works. For just 1 sample we get:



```
#include <math.h>
#include <stdlib.h>
#include <iostream>

inline float pdf(float x) {
    return  3*x*x/8;
}

int main() {
    int inside_circle = 0;
    int inside_circle_stratified = 0;
    int N = 1;
    float sum;
    for (int i = 0; i < N; i++) {
            float x = pow(8*drand48(), 1./3.);
            sum += x*x / pdf(x);
    }
    std::cout << "I =" << sum/N << "\n";
}
```

Which always returns the exact answer.

Let’s review now because that was most of the concepts that underlie MC ray tracers.

1. You have an integral of *f(x)* over some domain *[a,b]*
2. You pick a pdf *p* that is non-zero over *[a,b]*
3. You average a whole ton of *f(r)/p(r)* where r is a random number *r* with pdf *p*.

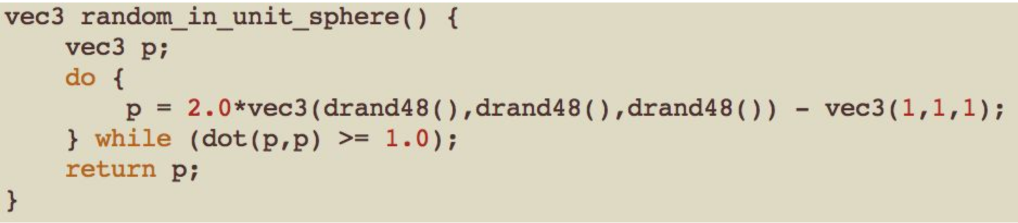
This will always converge to the right answer. The more *p* follows *f*, the faster it converges.

Chapter 2: MC Integration on the Sphere of Directions

In our ray tracer we pick random directions, and directions can be represented as points on the unit-sphere. The same methodology as before applies. But now we need to have a *pdf* defined over 2D. Suppose we have this integral over all directions:

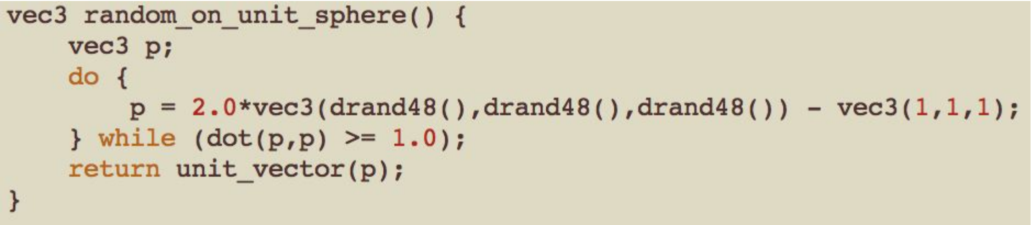
INTEGRAL(cos^2(theta))

By MC integration, we should just be able to sample *cos^2(theta) / p(direction)*. But what is direction in that context? We could make it based on polar coordinates, so *p* would be in terms of *(theta, phi)*. However you do it, remember that a pdf has to integrate to 1 and represent the relative probability of that direction being sampled. We have a method from the last books to take uniform random samples in a unit sphere:



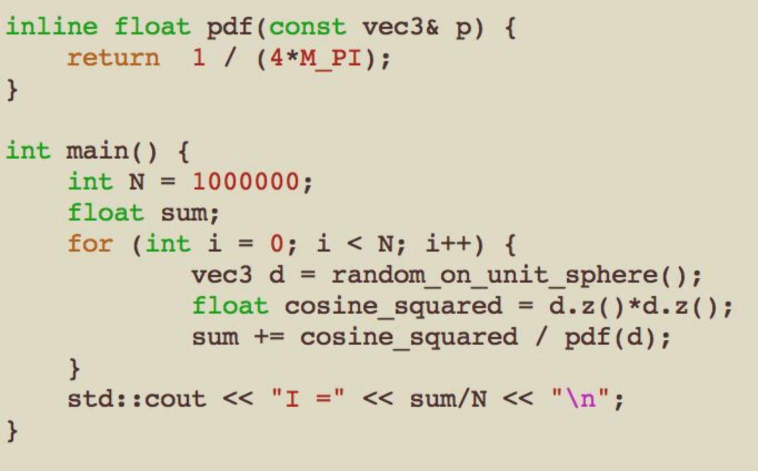
```
vec3 random_in_unit_sphere() {
    vec3 p;
    do {
        p = 2.0*vec3(drand48(),drand48(),drand48()) - vec3(1,1,1);
    } while (dot(p,p) >= 1.0);
    return p;
}
```

To get points on a unit sphere we can just normalize the vector to those points:



```
vec3 random_on_unit_sphere() {
    vec3 p;
    do {
        p = 2.0*vec3(drand48(),drand48(),drand48()) - vec3(1,1,1);
    } while (dot(p,p) >= 1.0);
    return unit_vector(p);
}
```

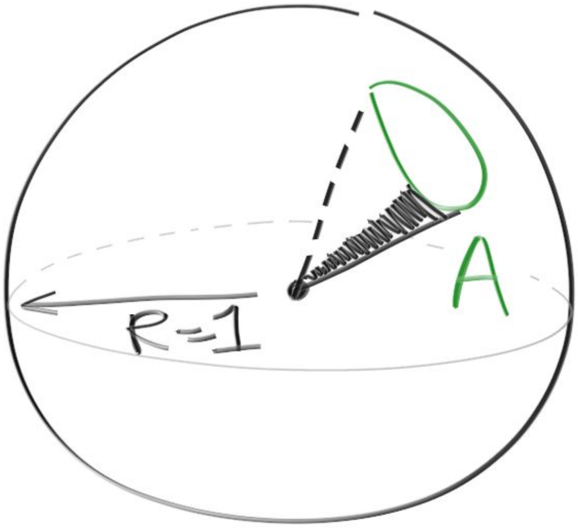
Now what is the pdf of these uniform points? As a density on the unit sphere, it is 1/area of the sphere or 1/(4*Pi). If the integrand is *cos^2(theta)* and *theta* is the angle with the *z* axis:



```
inline float pdf(const vec3& p) {
    return  1 / (4*M_PI);
}

int main() {
    int N = 1000000;
    float sum;
    for (int i = 0; i < N; i++) {
            vec3 d = random_on_unit_sphere();
            float cosine_squared = d.z()*d.z();
            sum += cosine_squared / pdf(d);
    }
    std::cout << "I =" << sum/N << "\n";
}
```

The analytic answer (if you remember enough advanced calc, check me!) is (4/3)*Pi, and the code above produces that. Next, we are ready to apply that in ray tracing!

The key point here is that all the integrals and probability and all that are over the unit sphere. The area on the unit sphere is how you measure the directions. Call it direction, solid angle, or area-- it’s all the same thing. Solid angle is the term usually used. If you are comfortable with that, great! If not, do what I do and imagine the area on the unit sphere that a set of directions goes through. The solid angle *omega* and the projected area *A* on the unit sphere are the same thing.



Now let’s go on to the light transport equation we are solving.

Chapter 3: Light Scattering

In this chapter we won't actually implement anything. We will set up for a big lighting change in our program in Chapter 4.

Our program from the last books already scatters rays from a surface or volume. This is the commonly used model for light interacting with a surface. One natural way to model this is with probability. First, is the light absorbed?

Probability of light scattering: A
Probability of light being absorbed: 1-A

Here *A* stands for *albedo* (latin for *whiteness*). Albedo is a precise technical term in some disciplines, but in all uses it means some form of fractional reflectance. As we implemented for glass, the albedo may vary with incident direction, and it varies with color. In the most physically based renderers, we would use a set of wavelengths for the light color rather than

RGB. We can almost always harness our intuition by thinking of R, G, and B as specific long, medium, and short wavelengths.

If the light does scatter, it will have a directional distribution that we can describe as a pdf over solid angle. I will refer to this as its *scattering pdf: s(direction)*. The scattering pdf can also vary with incident direction, as you will notice when you look at reflections off a road-- they become mirror-like as your viewing angle approaches grazing.

The color of a surface in terms of these quantities is:

*color = INTEGRAL   A * s(direction) * color(direction)*

Note that *A* and *s()* might depend on the view direction, so of course *color* can vary with view direction. Also *A* and *s()* may vary with position on the surface or within the volume.

If we apply the MC basic formula we get the following statistical estimate:

*Color = (A * s(direction) * color(direction)) / p(direction)*

Where *p(direction)* is the pdf of whatever direction we randomly generate.

For a Lambertian surface we already implicitly implemented this formula for the special case where p() is a cosine density. The *s()* of a Lambertian surface is proportional to *cos(theta),* where theta is the angle relative to the surface normal. Remember that all pdf need to integrate to one.   For *cos(theta) < 0* we have *s(direction) = 0*, and the integral of *cos* over the hemisphere is *Pi*.   To see that remember that in spherical coordinates remember that dA = sin(theta) dtheta dphi, so area = INT(phi from 0 to 2PI) INT(theta from 0 to PI/2) cos(theta)sin(theta) = 2PI * (½) = PI.

So for a Lambertian surface the scattering pdf is:

s(direction) = cos(theta) / Pi

If we sample using the same pdf, so *p(direction) = cos(theta) / Pi,* the numerator and denominator cancel out and we get:

*Color = A * s(direction)*

This is exactly what we had in our original *color()* function! But we need to generalize now so we can send extra rays in important directions such as toward the lights.

The treatment above is slightly non-standard because I want the same math to work for surfaces and volumes. To do otherwise will make some ugly code.

If you read the literature, you'll see reflection described by the *bidirectional reflectance distribution function (BRDF).* It relates pretty simply to our terms*:*

*BRDF = A * s(direction) / cos(theta)*

So for a Lambertian surface for example, BRDF = A / Pi.   Translation between our terms and BRDF is easy.

For participation media (volumes), our albedo is usually called *scattering albedo*, and our scattering pdf is usually called *phase function*.

Chapter 4: Importance Sampling Materials

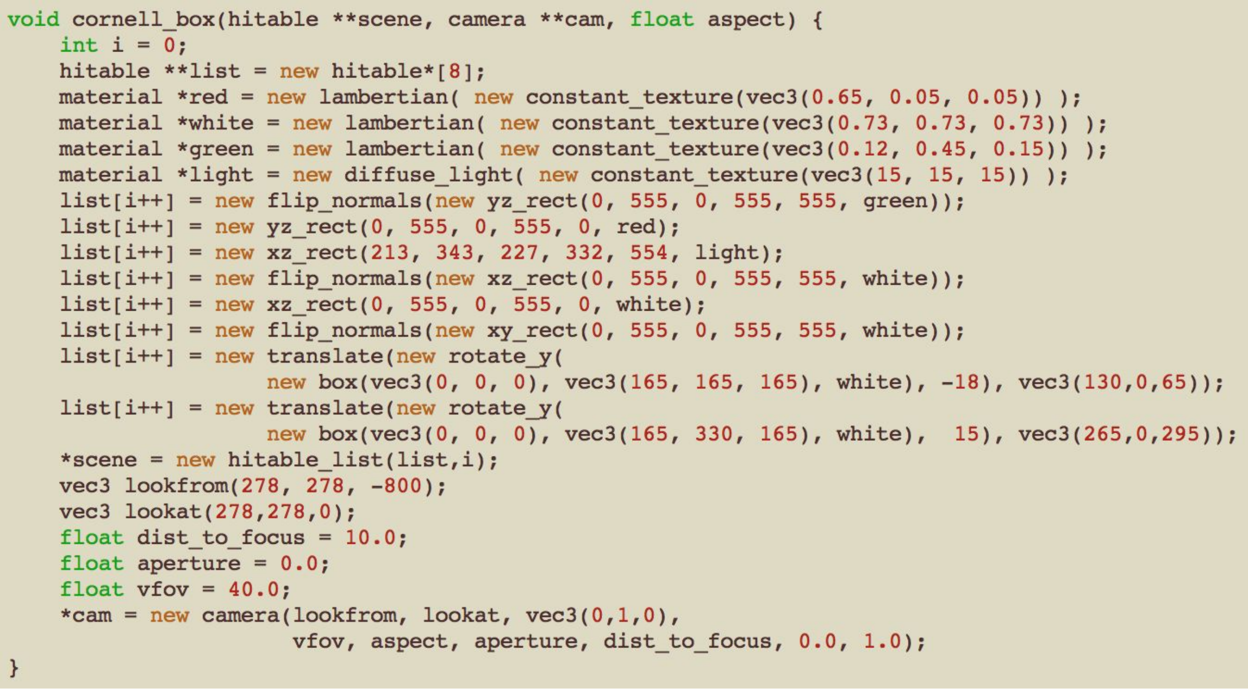
Our goal over the next two chapters is to instrument our program to send a bunch of extra rays toward light sources so that our picture is less noisy. Let's assume we can send a bunch of rays toward the light source using a pdf *pLight(direction)*. Let's also assume we have a pdf related to *s,* and let's call that *pSurface(direction)*. A great thing about pdfs is that you can just use linear mixtures of them to form mixture densities that are also pdfs. For example, the simplest would be:

*p(direction) = 0.5*pLight(direction) + 0.5*pSurface(direction)*

As long as the weights are positive and add up to one, any such mixture of pdfs is a pdf.   And remember, we can use any pdf-- it doesn't change the answer we converge to!   So, the game is to figure out how to make the pdf larger where the product *s(direction)*color(direction)* is large. For diffuse surfaces, this is mainly a matter of guessing where *color(direction)* is high.

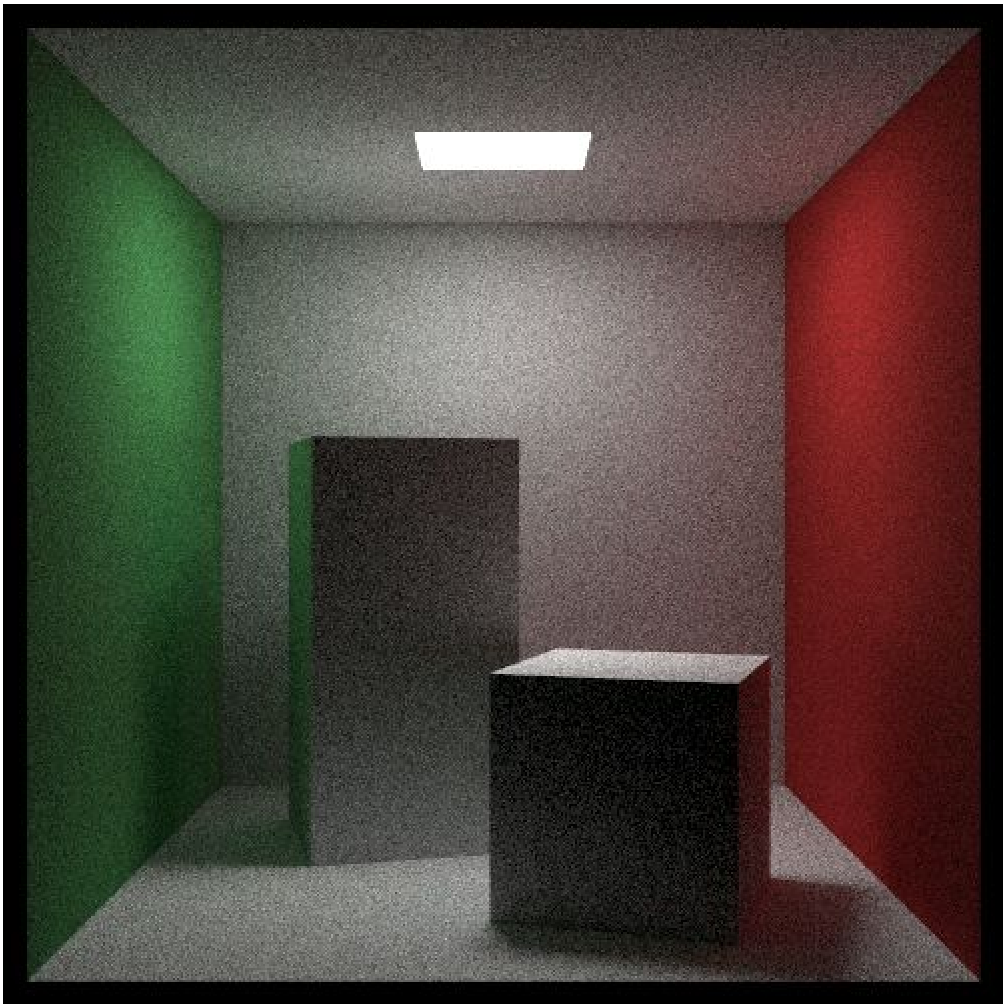
For a mirror, *s()* is huge only near one direction, so it matters a lot more. Most renderers in fact make mirrors a special case and just make the s/p implicit-- our code currently does that.

Let’s do simple refactoring and temporarily remove all materials that aren’t Lambertian.   We can use our Cornell Box scene again, and let’s generate the camera in the function that generates the model:



```
void cornell_box(hitable **scene, camera **cam, float aspect) {
    int i = 0;
    hitable **list = new hitable*[8];
    material *red = new lambertian( new constant_texture(vec3(0.65, 0.05, 0.05)) );
    material *white = new lambertian( new constant_texture(vec3(0.73, 0.73, 0.73)) );
    material *green = new lambertian( new constant_texture(vec3(0.12, 0.45, 0.15)) );
    material *light = new diffuse_light( new constant_texture(vec3(15, 15, 15)) );
    list[i++] = new flip_normals(new yz_rect(0, 555, 0, 555, 555, green));
    list[i++] = new yz_rect(0, 555, 0, 555, 0, red);
    list[i++] = new xz_rect(213, 343, 227, 332, 554, light);
    list[i++] = new flip_normals(new xz_rect(0, 555, 0, 555, 555, white));
    list[i++] = new xz_rect(0, 555, 0, 555, 0, white);
    list[i++] = new flip_normals(new xy_rect(0, 555, 0, 555, 555, white));
    list[i++] = new translate(new rotate_y(
                    new box(vec3(0, 0, 0), vec3(165, 165, 165), white), -18), vec3(130,0,65));
    list[i++] = new translate(new rotate_y(
                    new box(vec3(0, 0, 0), vec3(165, 330, 165), white),  15), vec3(265,0,295));
    *scene = new hitable_list(list,i);
    vec3 lookfrom(278, 278, -800);
    vec3 lookat(278,278,0);
    float dist_to_focus = 10.0;
    float aperture = 0.0;
    float vfov = 40.0;
    *cam = new camera(lookfrom, lookat, vec3(0,1,0),
                      vfov, aspect, aperture, dist_to_focus, 0.0, 1.0);
}
```

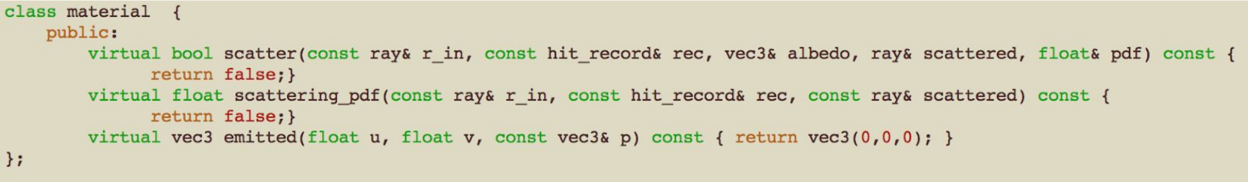
At 500x500 my code produces this image in 10min on 1 core of my Macbook:



Reducing that noise is our goal. We'll do that by constructing a pdf that sends more rays to the light.

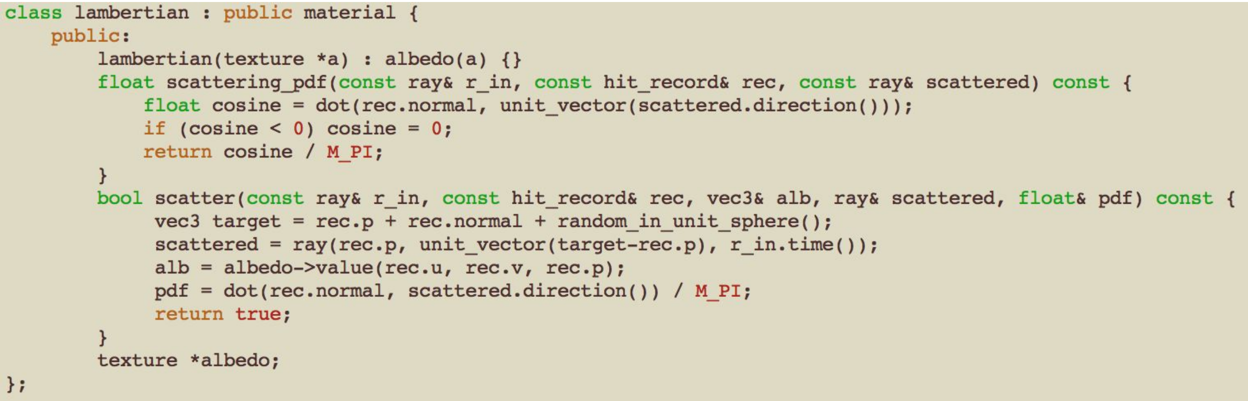
First, let's instrument the code so that it explicitly samples some pdf and then normalizes for that. Remember MC basics: *integral f(x)* is approximately *f(r)/p(r)*. For the Lambertian material, let's sample like we do now: p(direction) = cos(theta) / Pi.

We modify the base-class *material* to enable this importance sampling:



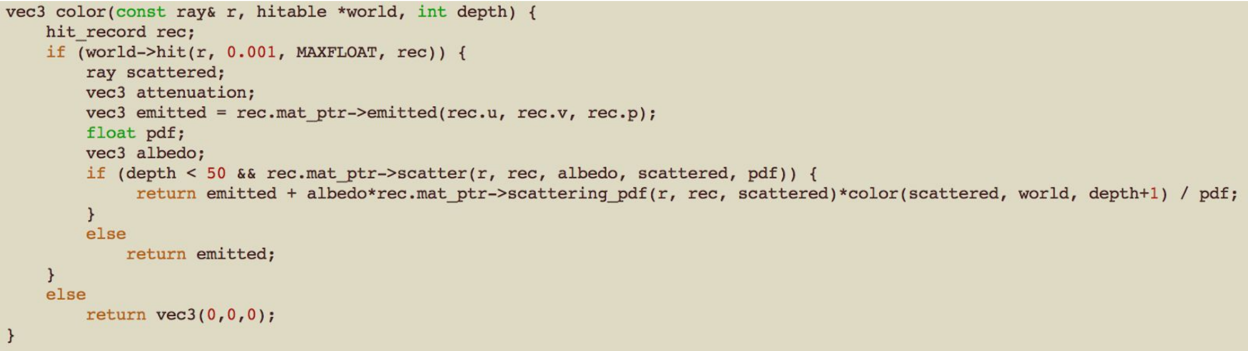
```
class material  {
    public:
        virtual bool scatter(const ray& r_in, const hit_record& rec, vec3& albedo, ray& scattered, float& pdf) const {
              return false;}
        virtual float scattering_pdf(const ray& r_in, const hit_record& rec, const ray& scattered) const {
              return false;}
        virtual vec3 emitted(float u, float v, const vec3& p) const { return vec3(0,0,0); }
};
```

And *Lambertian* material becomes:



```
class lambertian : public material {
    public:
        lambertian(texture *a) : albedo(a) {}
        float scattering_pdf(const ray& r_in, const hit_record& rec, const ray& scattered) const {
            float cosine = dot(rec.normal, unit_vector(scattered.direction()));
            if (cosine < 0) cosine = 0;
            return cosine / M_PI;
        }
        bool scatter(const ray& r_in, const hit_record& rec, vec3& alb, ray& scattered, float& pdf) const {
             vec3 target = rec.p + rec.normal + random_in_unit_sphere();
             scattered = ray(rec.p, unit_vector(target-rec.p), r_in.time());
             alb = albedo->value(rec.u, rec.v, rec.p);
             pdf = dot(rec.normal, scattered.direction()) / M_PI;
             return true;
        }
        texture *albedo;
};
```

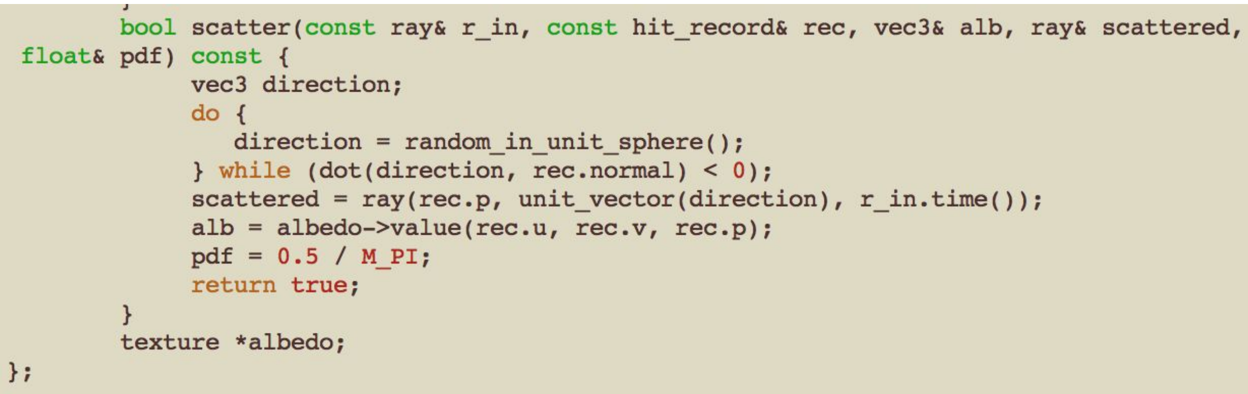
And the color function gets a minor modification.



```
vec3 color(const ray& r, hitable *world, int depth) {
    hit_record rec;
    if (world->hit(r, 0.001, MAXFLOAT, rec)) {
        ray scattered;
        vec3 attenuation;
        vec3 emitted = rec.mat_ptr->emitted(rec.u, rec.v, rec.p);
        float pdf;
        vec3 albedo;
        if (depth < 50 && rec.mat_ptr->scatter(r, rec, albedo, scattered, pdf)) {
             return emitted + albedo*rec.mat_ptr->scattering_pdf(r, rec, scattered)*color(scattered, world, depth+1) / pdf;
        }
        else
            return emitted;
    }
    else
        return vec3(0,0,0);
}
```

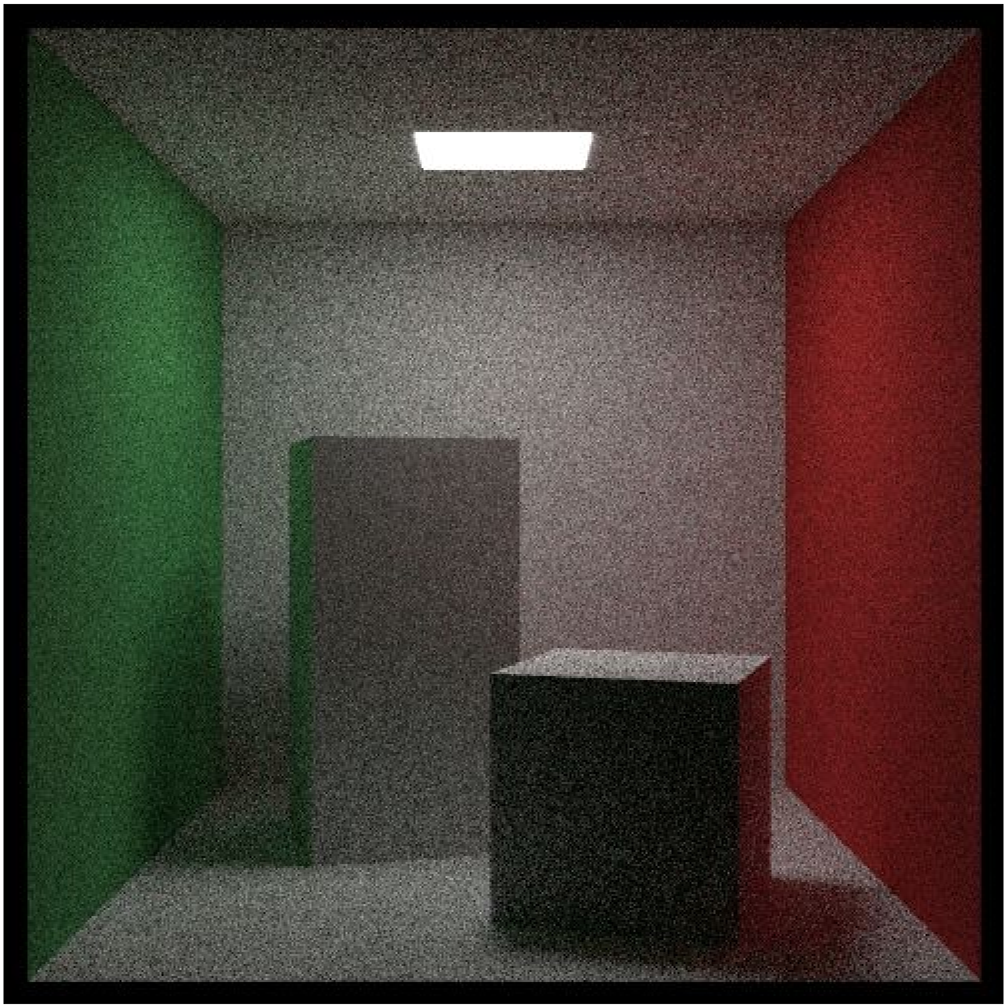
You should get exactly the same picture.

Now, just for the experience, try a different sampling strategy. Let’s choose randomly from the hemisphere that is above the surface. This would be *p(direction) = 1/(2*Pi)*.



```
        }
        bool scatter(const ray& r_in, const hit_record& rec, vec3& alb, ray& scattered,
 float& pdf) const {
              vec3 direction;
              do {
                  direction = random_in_unit_sphere();
              } while (dot(direction, rec.normal) < 0);
              scattered = ray(rec.p, unit_vector(direction), r_in.time());
              alb = albedo->value(rec.u, rec.v, rec.p);
              pdf = 0.5 / M_PI;
              return true;
        }
        texture *albedo;
};
```

And again I **should** get the same picture except with different variance, but I don’t!



It’s pretty close to our old picture, but there are differences that are not noise. The front of the tall box is much more uniform in color. So I have the most difficult kind of bug to find in a Monte Carlo program-- a bug that produces a reasonable looking image. And I don’t know if the bug is the first version of the program or the second, or even in both!

Let’s build some infrastructure to address this.

Chapter 5: Generating Random Directions

In this and the next two chapters let’s harden our understanding and tools and figure out which Cornell Box is right. Let’s first figure out how to generate random directions, but to simplify things let's assume the *z-axis* is the surface normal and *theta* is the angle from the normal. We'll get them oriented to the surface normal vector in the next chapter. We will only deal with distributions that are rotationally symmetric about *z*. So *p(direction) = f(theta)*. If you have had advanced calculus, you may recall that on the sphere in spherical coordinates dA = sin(theta)*dtheta*dphi. If you haven't, you'll have to take my word for the next step, but you'll get it when you take advanced calc.

Given a directional *pdf, p(direction) = f(theta)*on the sphere, the 1D *pdfs* on *theta* and *phi* are:

a(phi) = 1/(2Pi) (uniform)
b(theta) = 2*Pi*f(theta)sin(theta)

For uniform random numbers r1 and r2, the material presented in Chapter 2 leads to:

r1 = INTEGRAL_0^phi (1/(2*Pi)) = phi/(2*Pi)

Solving for *phi* we get:

phi = 2*Pi*r1

For *theta* we have:

r2 = INTEGRAL_0^theta 2*Pi*f(t)sin(t)

Here, *t* is a dummy variable. Let's try some different functions for *f()*. Let's first try a uniform density on the sphere. The area of the unit sphere is 4*Pi*, so a uniform *p(direction) = 1/(4Pi)* on the unit sphere.

r2 = -cos(theta)/2 - (-cos(0)/2)= (1 - cos(theta))/2

Solving for *cos(theta)* gives:

*cos(theta) = 1 - 2*r2*

We don't solve for *theta* because we probably only need to know *cos(theta)* anyway and don't want needless *acos()* calls running around.

To generate a unit vector direction toward *(theta,phi)* we convert to Cartesian coordinates:

x = cos(phi)*sin(theta)
y = sin(phi)*sin(theta)
z = cos(theta)

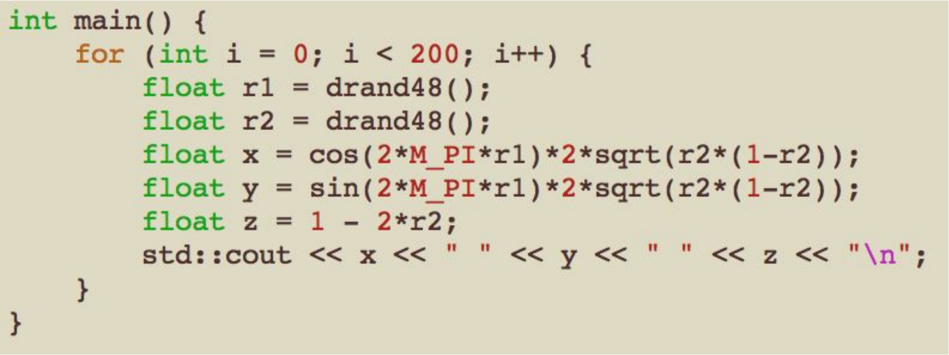
And using the identity that *cos^2* + *sin^2* = *1*, we get the following in terms of random *(r1,r2)*:

x = cos(2*Pi*r1)*sqrt(1 - (1-2*r2)^2)
y = sin(2*Pi*r1)*sqrt(1 - (1-2*r2)^2)
z = 1 - 2*r2

Simplifying a little, *(1-2*r2)^2 = 1 - 4*r2 + 4*r2^2,* so:

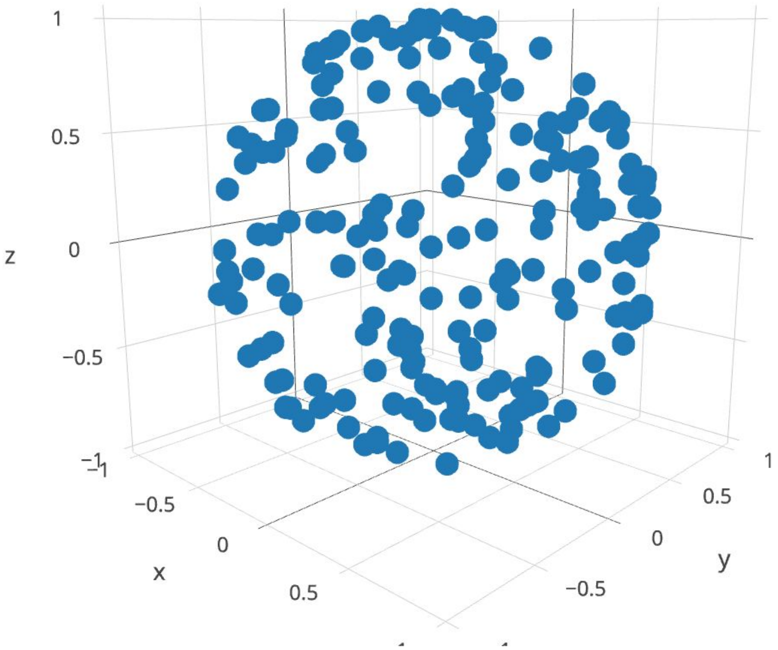
x = cos(2*Pi*r1)*2*sqrt(r2*(1-r2))
y = sin(2*Pi*r1)*2*sqrt(r2*(1-r2))
z = 1 - 2*r2

We can output some of these:



```
int main() {
    for (int i = 0; i < 200; i++) {
        float r1 = drand48();
        float r2 = drand48();
        float x = cos(2*M_PI*r1)*2*sqrt(r2*(1-r2));
        float y = sin(2*M_PI*r1)*2*sqrt(r2*(1-r2));
        float z = 1 - 2*r2;
        std::cout << x << " " << y << " " << z << "\n";
    }
}
```

And plot them for free on plot.ly (a great site with 3D scatterplot support):



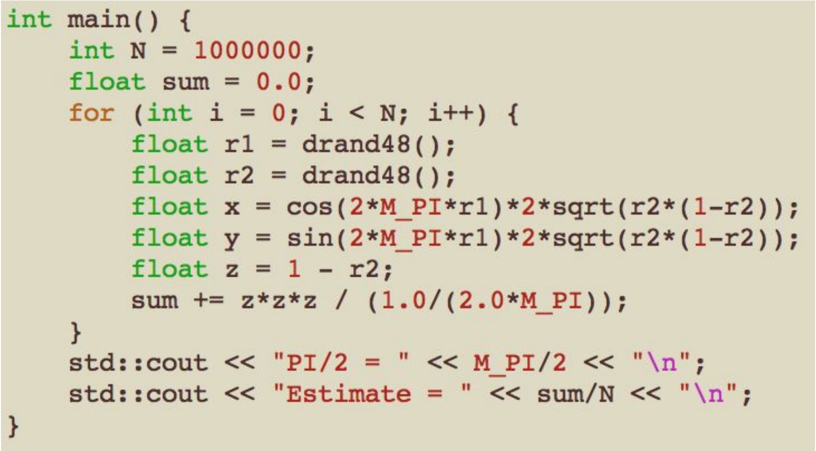
On the plot.ly website you can rotate that around and see that it appears uniform.

Now let's derive *uniform on the hemisphere*.   The density being uniform on the hemisphere means *p(direction) = 1/(2*Pi)* and just changing the constant in the *theta* equations yields:

cos(theta) = 1 - r2

It is comforting that *cos(theta)* will vary from 1 to 0, and thus *theta* will vary from 0 to *Pi/2.*   Rather than plot it, let's do a 2D integral with a known solution.   Let's integrate *cosine* cubed over the hemisphere (just picking something arbitrary with a known solution).   First let's do it by hand: *INTEGRAL cos^3   = 2*Pi*INTEGRAL_0^PI/2 cos^3 sin = PI/2.*

Now for integration with importance sampling.   *p(direction) = 1/(2*Pi)*, so we average f/p which is *cos^3/ (1/(2*Pi))*, and we can test this:



```
int main() {
    int N = 1000000;
    float sum = 0.0;
    for (int i = 0; i < N; i++) {
        float r1 = drand48();
        float r2 = drand48();
        float x = cos(2*M_PI*r1)*2*sqrt(r2*(1-r2));
        float y = sin(2*M_PI*r1)*2*sqrt(r2*(1-r2));
        float z = 1 - r2;
        sum += z*z*z / (1.0/(2.0*M_PI));
    }
    std::cout << "PI/2 = " << M_PI/2 << "\n";
    std::cout << "Estimate = " << sum/N << "\n";
}
```

Now let’s generate directions with *p(directions) = cos(theta) / Pi*.

*r2 = INTEGRAL_0^theta 2*Pi*(cos(t)/Pi)*sin(t) = 1-cos^2(theta)*

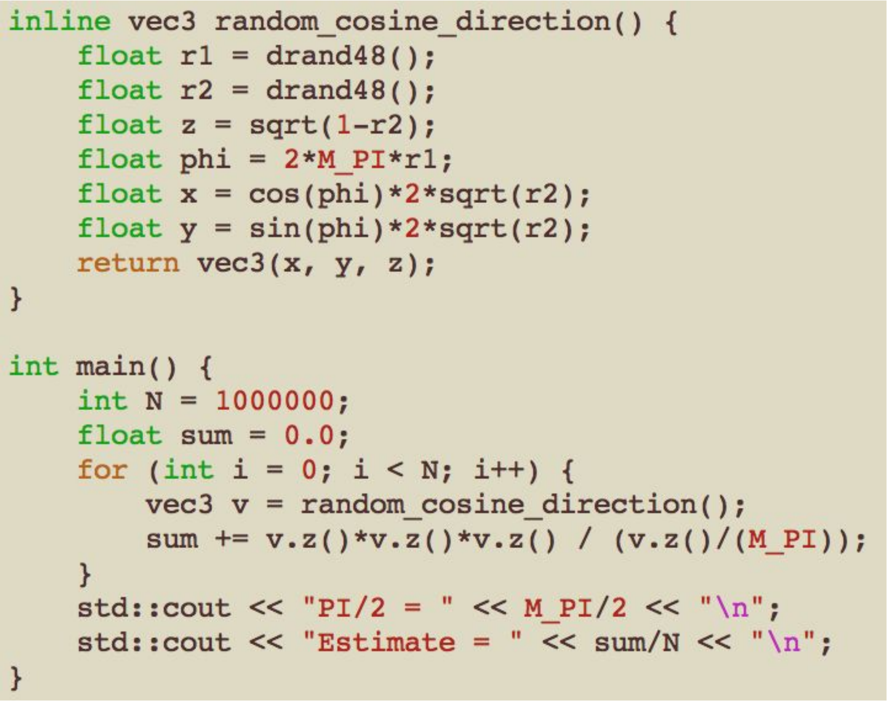
So,

cos(theta) = sqrt(1-r2)

We can save a little algebra on specific cases by noting

z = cos(theta) = sqrt(1-r2)
*x = cos(phi)*sin(theta) = cos(2*Pi*r1)*sqrt(1-z^2) = cos(2*Pi*r1)*sqrt(r2)*
*y = sin(phi)*sin(theta) = sin(2*Pi*r1)*sqrt(1-z^2) = sin(2*Pi*r1)*sqrt(r2)*

Let’s also start generating them as random vectors:



```
inline vec3 random_cosine_direction() {
    float r1 = drand48();
    float r2 = drand48();
    float z = sqrt(1-r2);
    float phi = 2*M_PI*r1;
    float x = cos(phi)*2*sqrt(r2);
    float y = sin(phi)*2*sqrt(r2);
    return vec3(x, y, z);
}

int main() {
    int N = 1000000;
    float sum = 0.0;
    for (int i = 0; i < N; i++) {
        vec3 v = random_cosine_direction();
        sum += v.z()*v.z()*v.z() / (v.z()/(M_PI));
    }
    std::cout << "PI/2 = " << M_PI/2 << "\n";
    std::cout << "Estimate = " << sum/N << "\n";
}
```

We can generate other densities later as we need them. In the next chapter we'll get them aligned to the surface normal vector.

Chapter 6: Ortho-normal Bases

In the last chapter we developed methods to generate random directions relative to the z-axis. We'd like to do that relative to a surface normal vector . An ortho-normal basis (ONB) is a collection of three mutually orthogonal unit vectors. The Cartesian xyz axes are one such ONB and I sometimes forget that it has to sit in some real place with real orientation to have meaning in the real world, and some virtual place and orientation in the virtual world. A picture is a result of the relative positions/orientations of the camera and scene, so as long as the camera and scene are described in the same coordinate system, all is well.

Suppose we have an origin **o** and cartesian unit vectors **x**/**y**/**z**. When we say a location is (3, -2, 7), we really are saying:

Location is **o** + 3***x** - 2***y** + 7***z**

If we want to measure coordinates in another coordinate system with origin **o’** and basis vectors **u**/**v**/**w,** we can just find the numbers *(u, v, w)* such that:

Location is **o’** + u**u** + v**v** + **w**.

If you take an intro graphics course, there will be a lot of time spent on coordinate systems and 4x4 coordinate transformation matrices. Pay attention, it’s important stuff in graphics! But we won’t need it. What we need to is generate random directions with a set distribution relative to **n**. We don’t need an origin because a direction is relative to no specified origin. We do need two cotangent vectors that are mutually perpendicular to **n** and each other.

Some models will come with at least one cotangent vector. The hard case of making an ONB is when we just have one vector. Suppose we have any vector **a** that is not zero length or parallel to **n**. We can get to vectors **s** and **t** parallel to **n** by using the property of the cross product that cross(**c**,**d**) is perpendicular to both **c** and **d:**

t = *unit_vector(cross(***a, n***))*
s = *cross(***t,n***)*

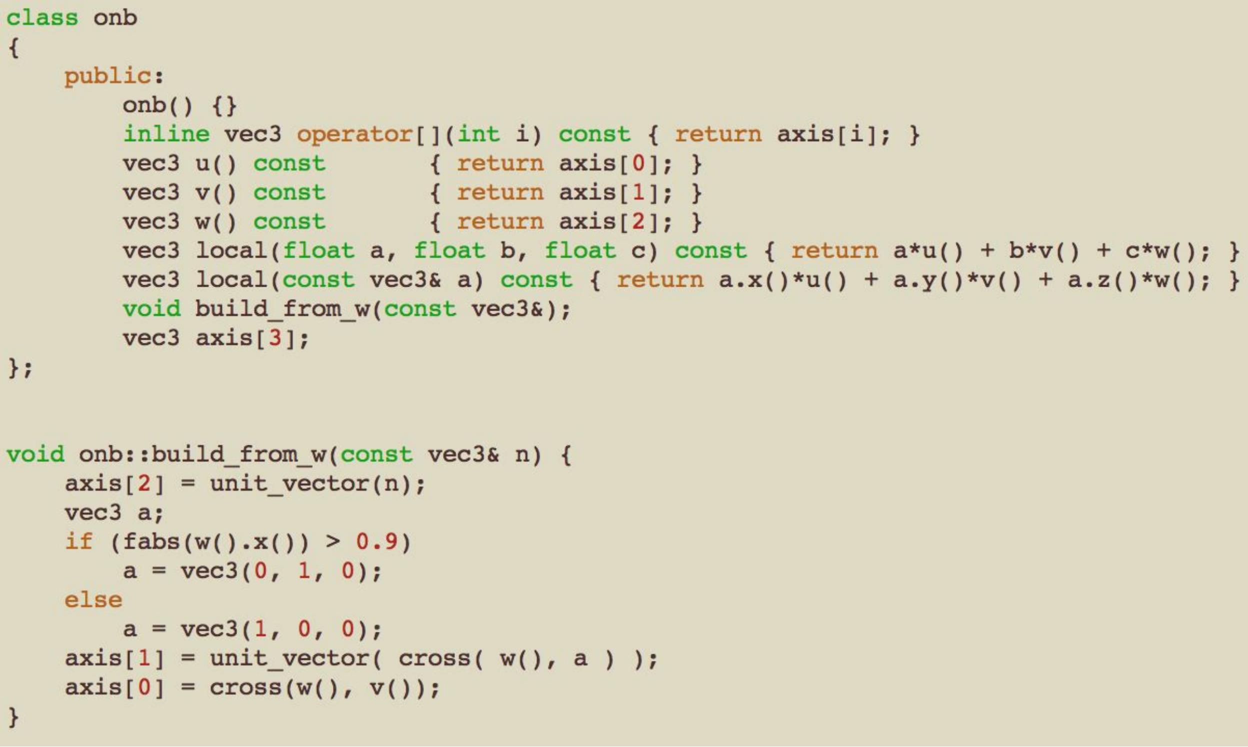
The catch is, we don’t have an **a** and if we pick a particular one at some point we will get an **n** parallel to **a**. A common method is to use an *if statement* to determine whether **n** is a particular axis, and if not, use that axis.

```
If (fabs(n.x()) > 0.9)
    a = (0,1,0)
else
    a = (1,0 ,0)
```

Once we have an ONB and we have a random *(x,y,z)* relative the the z-axis we can get the vector relative to **n** as:

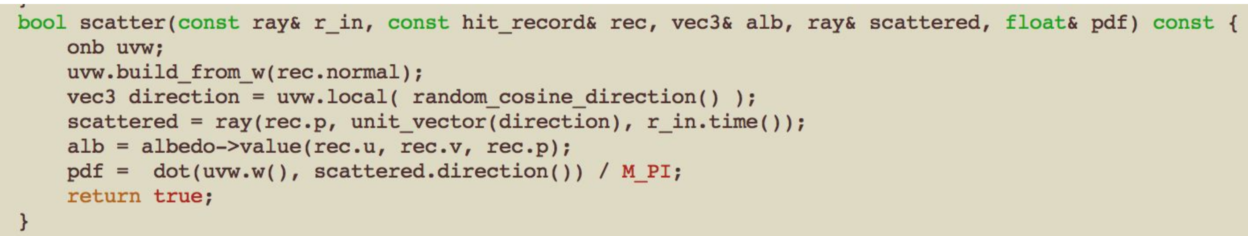
Random vector = x*s + y*t + z*n

You may notice we used similar math to get rays from a camera. That could be viewed as a change to the camera’s natural coordinate system. Should we make a class for ONBs or are utility functions enough?   I’m not sure, but let’s make a class because it won't really be more complicated than utility functions:



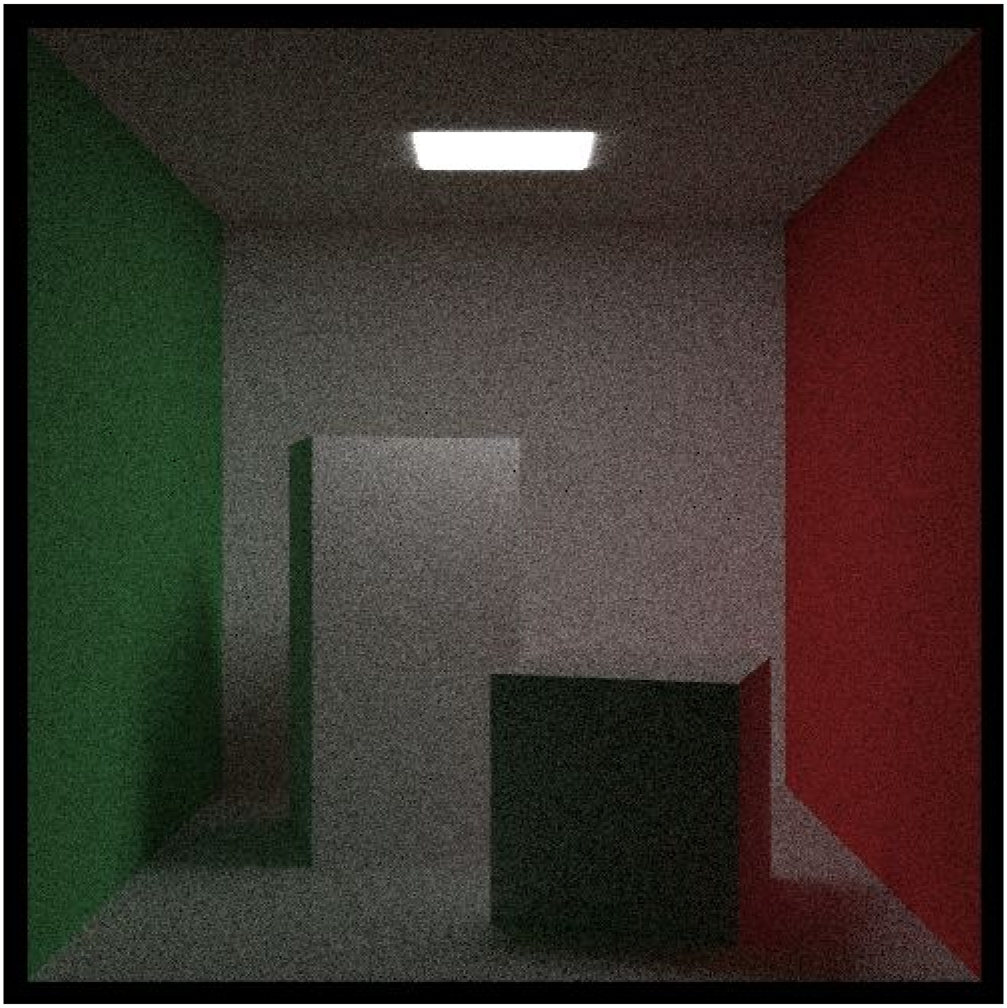
```
class onb
{
    public:
        onb() {}
        inline vec3 operator[](int i) const { return axis[i]; }
        vec3 u() const       { return axis[0]; }
        vec3 v() const       { return axis[1]; }
        vec3 w() const       { return axis[2]; }
        vec3 local(float a, float b, float c) const { return a*u() + b*v() + c*w(); }
        vec3 local(const vec3& a) const { return a.x()*u() + a.y()*v() + a.z()*w(); }
        void build_from_w(const vec3&);
        vec3 axis[3];
};


void onb::build_from_w(const vec3& n) {
    axis[2] = unit_vector(n);
    vec3 a;
    if (fabs(w().x()) > 0.9)
        a = vec3(0, 1, 0);
    else
        a = vec3(1, 0, 0);
    axis[1] = unit_vector( cross( w(), a ) );
    axis[0] = cross(w(), v());
}
```

We can rewrite our Lambertian material using this to get:



```
bool scatter(const ray& r_in, const hit_record& rec, vec3& alb, ray& scattered, float& pdf) const {
    onb uvw;
    uvw.build_from_w(rec.normal);
    vec3 direction = uvw.local( random_cosine_direction() );
    scattered = ray(rec.p, unit_vector(direction), r_in.time());
    alb = albedo->value(rec.u, rec.v, rec.p);
    pdf =  dot(uvw.w(), scattered.direction()) / M_PI;
    return true;
}
```

Which produces:



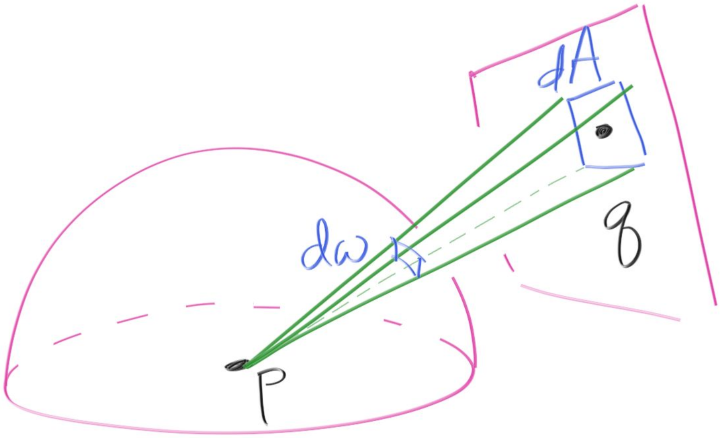
Is that right? We still don't know for sure. Tracking down bugs is hard in the absence of reliable reference solutions. Let's table that for now and move on to get rid of some of that noise.

Chapter 7: Sampling Lights Directly

The problem with sampling almost uniformly over directions is that lights are not sampled any more than unimportant directions. We could use shadow rays and separate out direct lighting. But instead, I'll just send more rays to the light.   We can then use that later to send more rays in whatever direction we want.

It's really easy to pick a random direction toward the light; just pick a random point on the light and send a ray in that direction.   But we also need to know the *pdf(direction)*.   What is that?

For a light of area *A,* if we sample uniformly on that light, the *pdf* on the surface of the light is 1/*A*. But what is it on the area of the unit sphere that defines directions? Fortunately, there is a simple correspondence, as outlined in the diagram:



If we look at a small area *dA* on the light, the probability of sampling it is p_q(**q**)*dA. On the sphere, the probability of sampling the small area *dw* on the sphere is p(direction)*dw. There is a geometric relationship between *dw* and *dA:*

*dw = dA cos(alpha) / (distance(**p,q**)^2)*

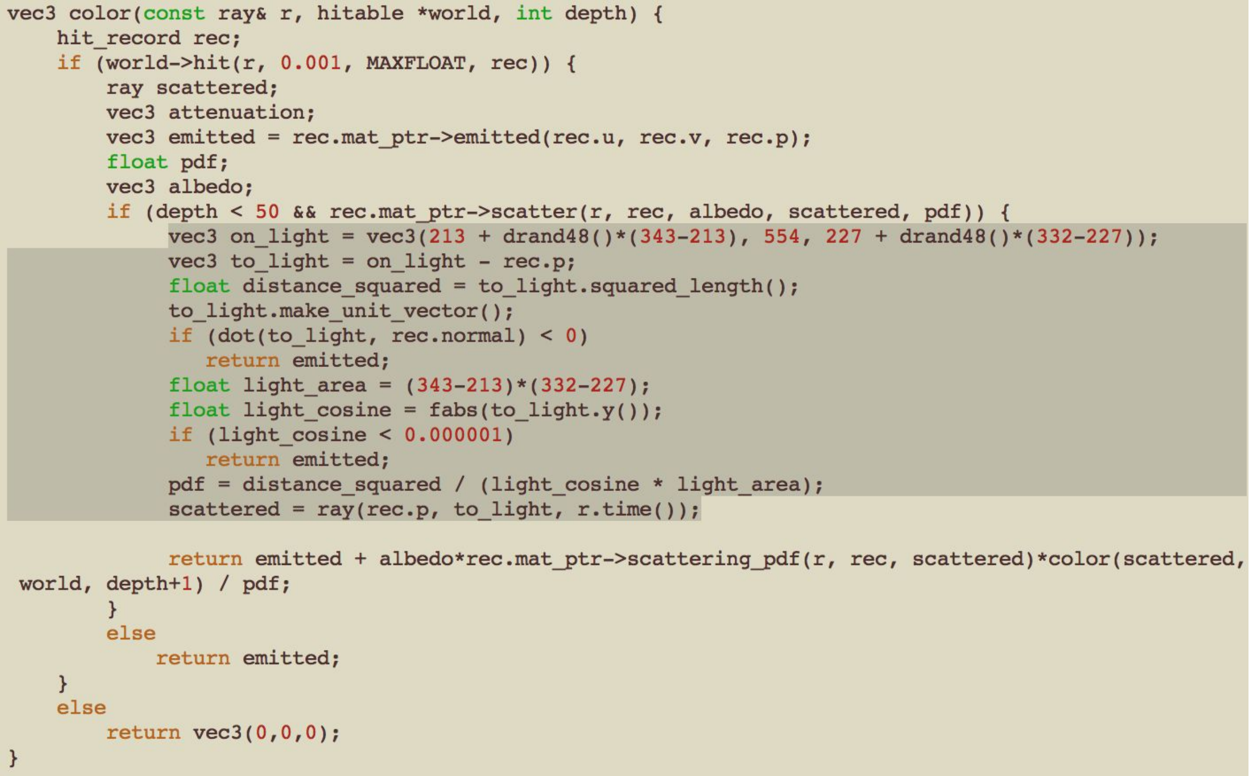
Since the probability of sampling dw and dA must be the same, we have

p(direction)*cos(alpha)*dA / (distance(**p,q**)^2) = p_q(**q**)*dA = dA / A

So

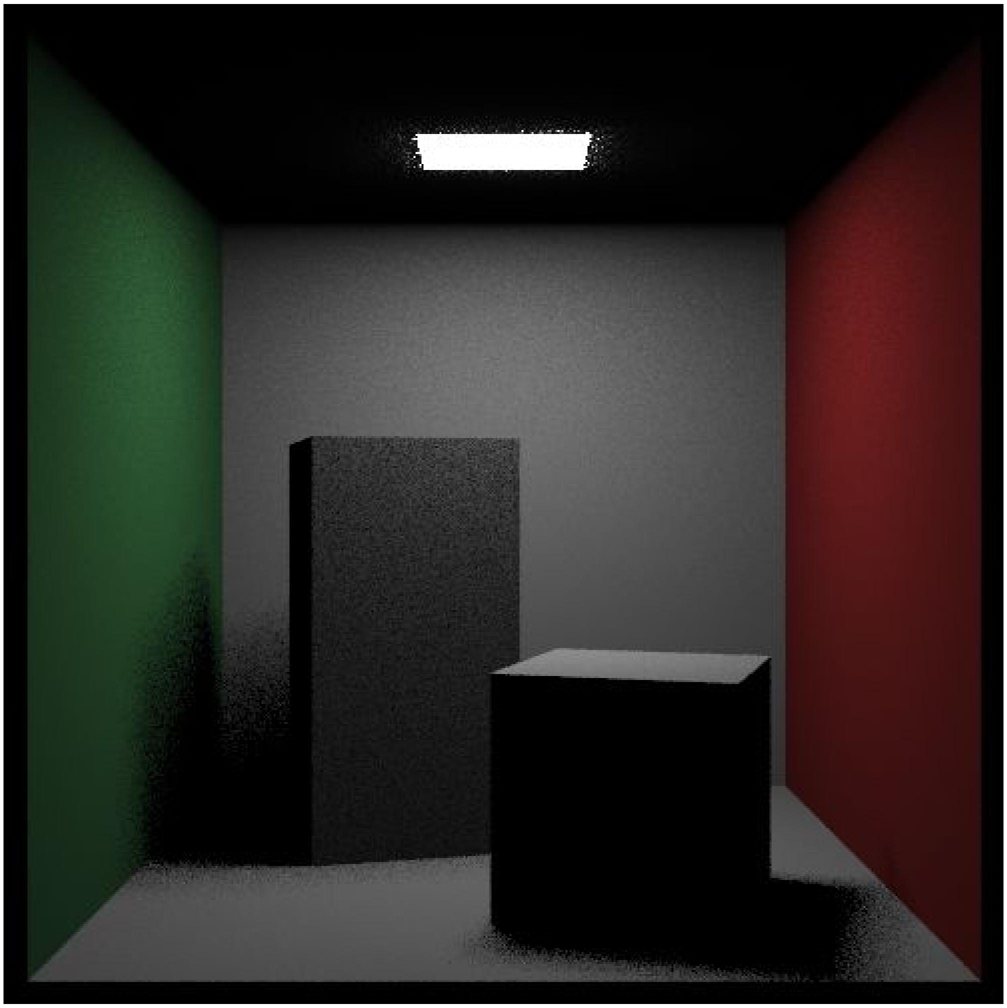
*p(direction) = distance(**p,q**)^2 / (cos(alpha) *A)*

If we hack our color function to sample the light in a very hard-coded fashion just to check that math and get the concept, we can add it (see the highlighted region):

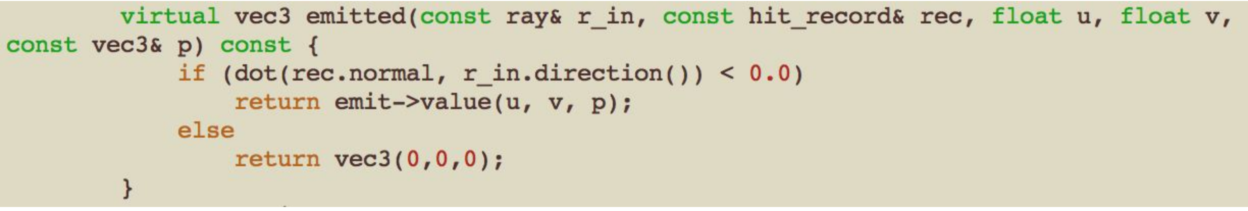


```
vec3 color(const ray& r, hitable *world, int depth) {
    hit_record rec;
    if (world->hit(r, 0.001, MAXFLOAT, rec)) {
        ray scattered;
        vec3 attenuation;
        vec3 emitted = rec.mat_ptr->emitted(rec.u, rec.v, rec.p);
        float pdf;
        vec3 albedo;
        if (depth < 50 && rec.mat_ptr->scatter(r, rec, albedo, scattered, pdf)) {
             vec3 on_light = vec3(213 + drand48()*(343-213), 554, 227 + drand48()*(332-227));
             vec3 to_light = on_light - rec.p;
             float distance_squared = to_light.squared_length();
             to_light.make_unit_vector();
             if (dot(to_light, rec.normal) < 0)
                return emitted;
             float light_area = (343-213)*(332-227);
             float light_cosine = fabs(to_light.y());
             if (light_cosine < 0.000001)
                return emitted;
             pdf = distance_squared / (light_cosine * light_area);
             scattered = ray(rec.p, to_light, r.time());

             return emitted + albedo*rec.mat_ptr->scattering_pdf(r, rec, scattered)*color(scattered,
 world, depth+1) / pdf;
        }
        else
            return emitted;
    }
    else
        return vec3(0,0,0);
}
```

With 10 samples per pixel this yields:

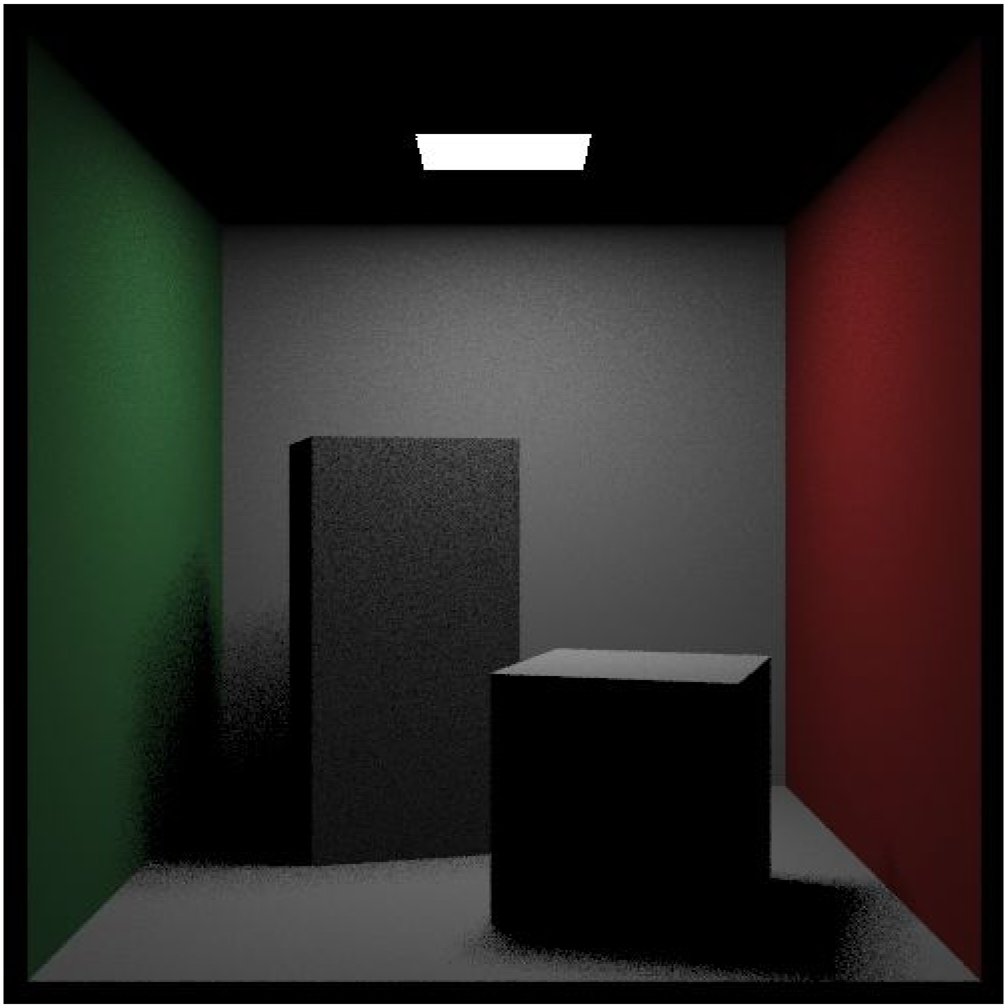


This is about what we would expect from something that samples only the light sources, so this appears to work. The noisy pops around the light on the ceiling are because the light is two-sided and there is a small space between light and ceiling. We probably want to have the light just emit down. We can do that by letting the emitted member function of *hitable* take extra information:



```
        virtual vec3 emitted(const ray& r_in, const hit_record& rec, float u, float v,
const vec3& p) const {
            if (dot(rec.normal, r_in.direction()) < 0.0)
                return emit->value(u, v, p);
            else
                return vec3(0,0,0);
        }
```

We also need to flip the light so its normals point in the -y direction and we get:



Chapter 8: Mixture Densities

We have used a *pdf* related to *cosine theta*, and a *pdf* related to sampling the light.   We would like a *pdf* that combines these.   A common tool in probability is to mix the densities to form a *mixture density*. Any weighted average of *pdf*s is a *pdf*.   For example, we could just average the two densities:

pdf_mixture(direction) = ½ pdf_reflection(direction) + ½ pdf_light(direction)

How would we instrument our code to do that? There is a very important detail that makes this not quite as easy as one might expect. Choosing the random direction is simple:

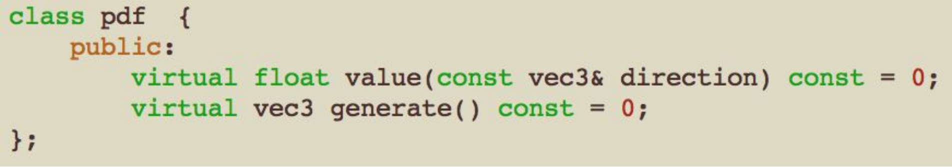
```
if (drand48() < 0.5)
    Pick direction according to pdf_reflection
else
    Pick direction according to pdf_light
```

But evaluating *pdf_mixture* is slightly more subtle.   We need to evaluate both *pdf_reflection* and *pdf_light* because there are some directions where either *pdf* could have generated the direction. For example, we might generate a direction toward the light using *pdf_reflection*.

If we step back a bit, we see that there are two functions a *pdf* needs to support:

1. What is your value at this location?
2. Return a random number that is distributed appropriately.

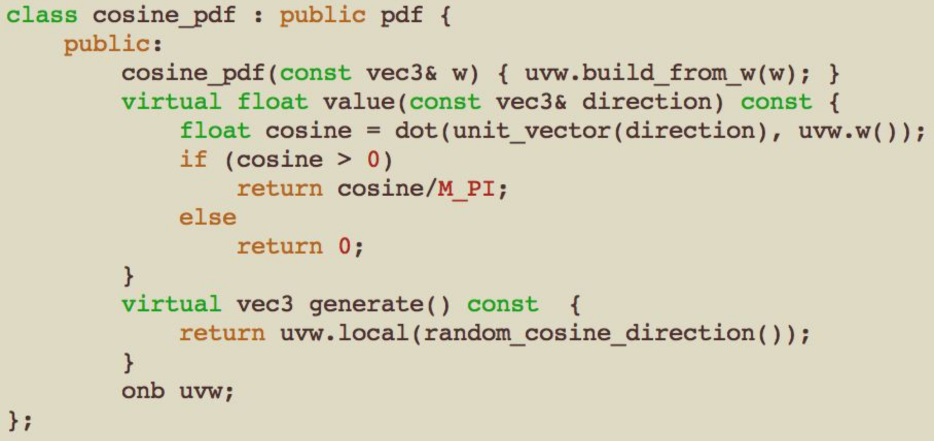
The details of how this is done under the hood varies for the *pdf_reflection* and the *pdf_light* and the mixture density of the two of them, but that is exactly what class hierarchies were invented for! It's never obvious what goes in an abstract class, so my approach is to be greedy and hope a minimal interface works, and for the *pdf* this implies:



```
class pdf  {
    public:
        virtual float value(const vec3& direction) const = 0;
        virtual vec3 generate() const = 0;
};
```

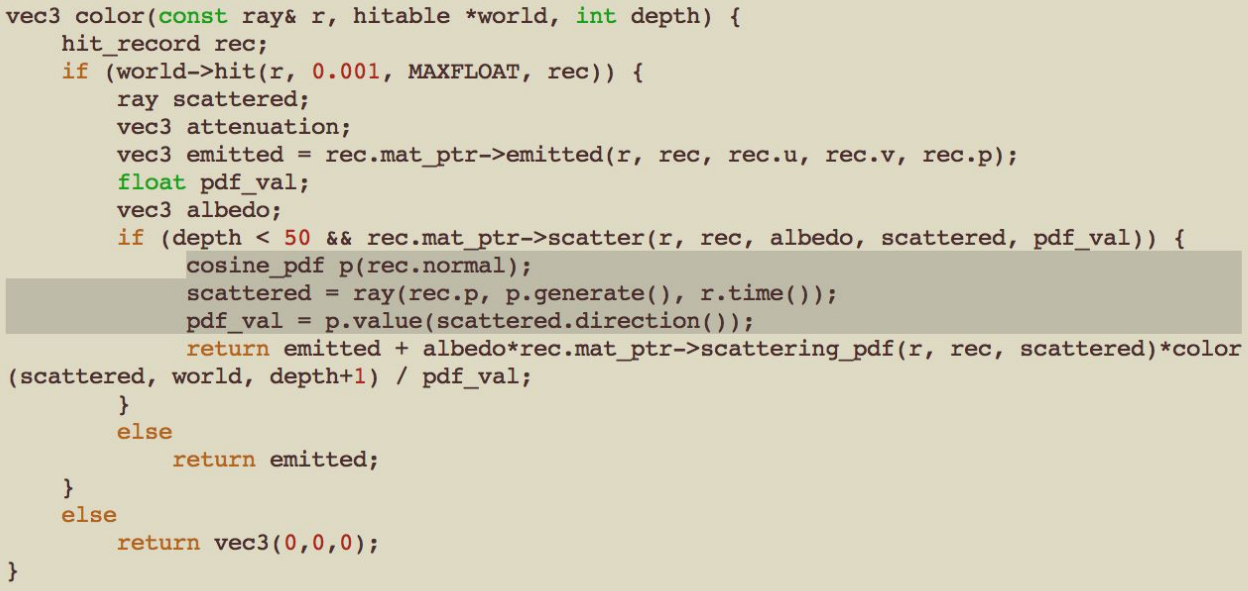
We'll see if that works by fleshing out the subclasses.   For sampling the light, we will need *hitable* to answer some queries that it doesn't have an interface for.   We'll probably need to mess with it too, but we can start by seeing if we can put something in *hitable* involving sampling the bounding box that works with all its subclasses.

First, let's try a cosine density:



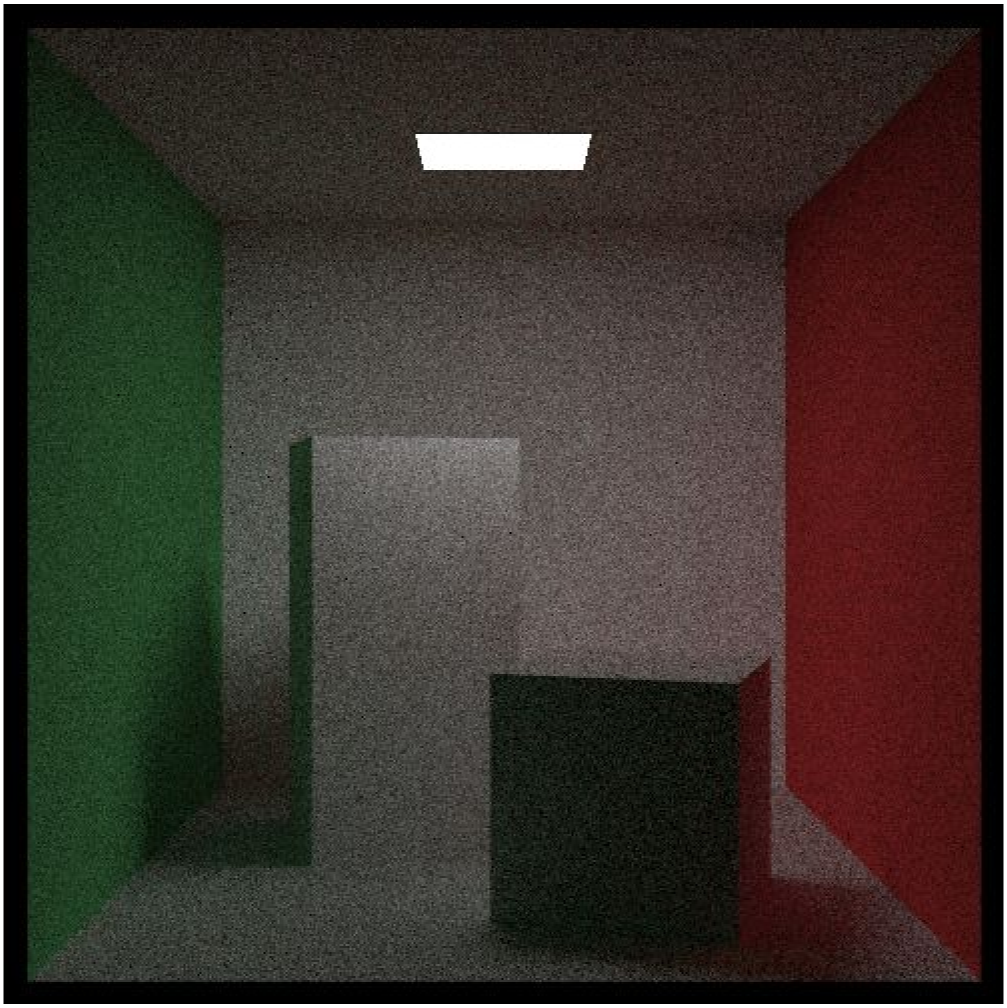
```
class cosine_pdf : public pdf {
    public:
        cosine_pdf(const vec3& w) { uvw.build_from_w(w); }
        virtual float value(const vec3& direction) const {
            float cosine = dot(unit_vector(direction), uvw.w());
            if (cosine > 0)
                return cosine/M_PI;
            else
                return 0;
        }
        virtual vec3 generate() const  {
            return uvw.local(random_cosine_direction());
        }
        onb uvw;
};
```

We can try this in the *color() function*, with the main changes highlighted.   We also need to change variable *pdf* to some other variable name to avoid a name conflict with the new *pdf* class.

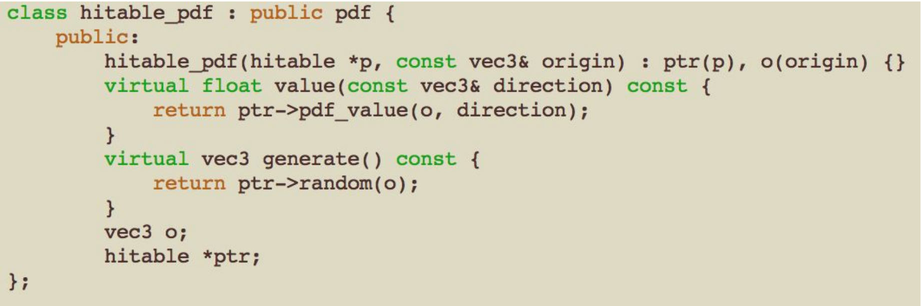


```
vec3 color(const ray& r, hitable *world, int depth) {
    hit_record rec;
    if (world->hit(r, 0.001, MAXFLOAT, rec)) {
        ray scattered;
        vec3 attenuation;
        vec3 emitted = rec.mat_ptr->emitted(r, rec, rec.u, rec.v, rec.p);
        float pdf_val;
        vec3 albedo;
        if (depth < 50 && rec.mat_ptr->scatter(r, rec, albedo, scattered, pdf_val)) {
             cosine_pdf p(rec.normal);
             scattered = ray(rec.p, p.generate(), r.time());
             pdf_val = p.value(scattered.direction());
             return emitted + albedo*rec.mat_ptr->scattering_pdf(r, rec, scattered)*color
(scattered, world, depth+1) / pdf_val;
        }
        else
            return emitted;
    }
    else
        return vec3(0,0,0);
}
```

This yields an apparently matching result so all we’ve done so far is refactor where *pdf* is computed:

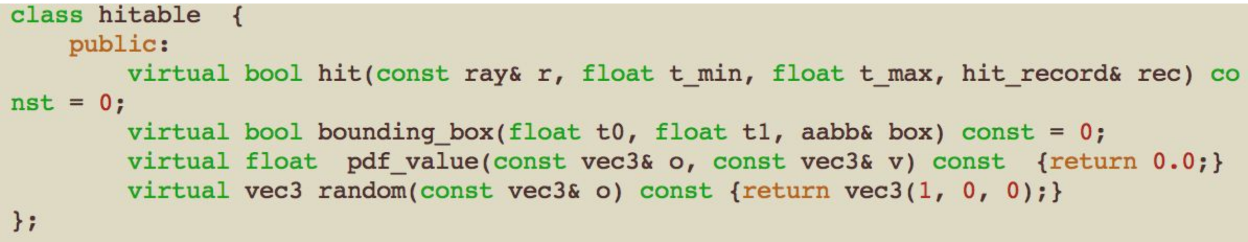


Now we can try sampling directions toward a *hitable* like the light.



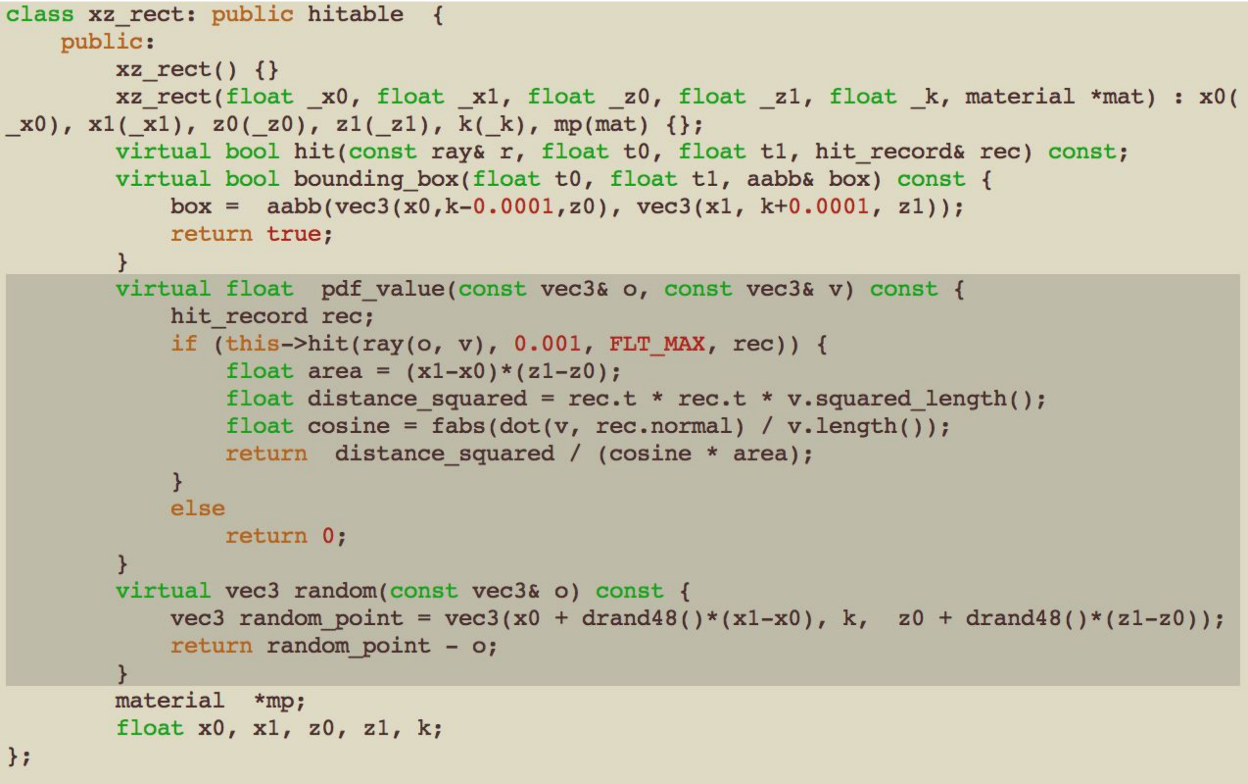
```
class hitable_pdf : public pdf {
    public:
        hitable_pdf(hitable *p, const vec3& origin) : ptr(p), o(origin) {}
        virtual float value(const vec3& direction) const {
            return ptr->pdf_value(o, direction);
        }
        virtual vec3 generate() const {
            return ptr->random(o);
        }
        vec3 o;
        hitable *ptr;
};
```

This assumes two as-yet not implemented functions in the *hitable* class. To avoid having to add instrumentation to all of *hitable* subclasses, we'll add two dummy functions to the *hitable* class:



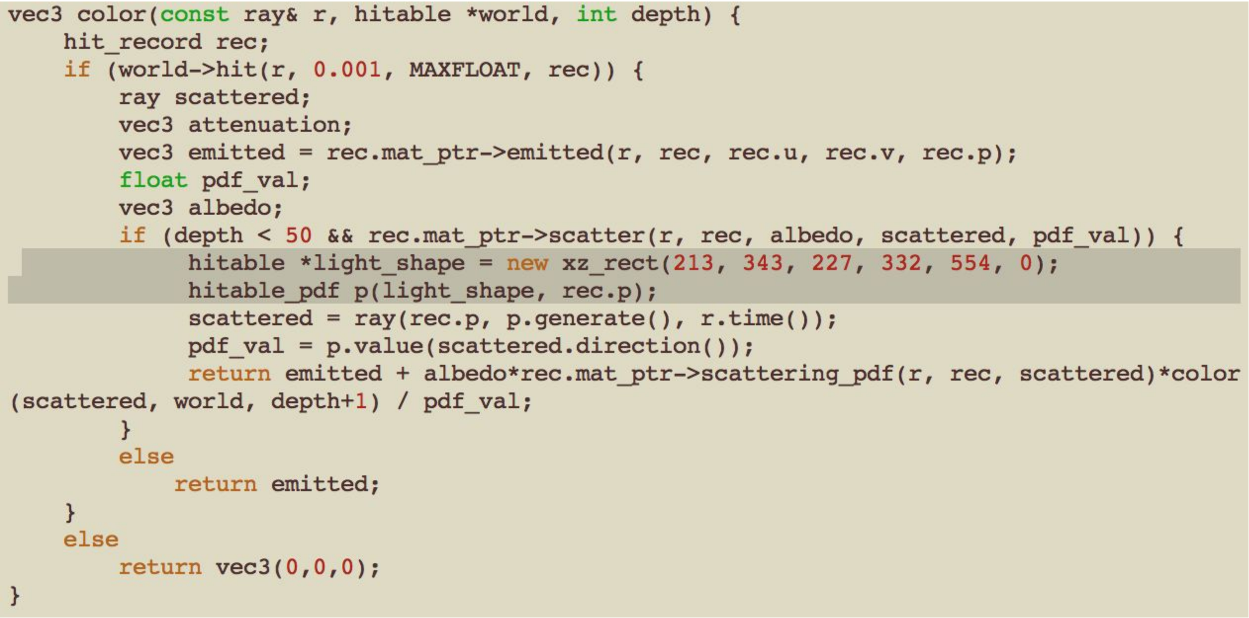
```
class hitable  {
    public:
        virtual bool hit(const ray& r, float t_min, float t_max, hit_record& rec) co
nst = 0;
        virtual bool bounding_box(float t0, float t1, aabb& box) const = 0;
        virtual float  pdf_value(const vec3& o, const vec3& v) const  {return 0.0;}
        virtual vec3 random(const vec3& o) const {return vec3(1, 0, 0);}
};
```

And we change *xz_rect* to implement those functions:



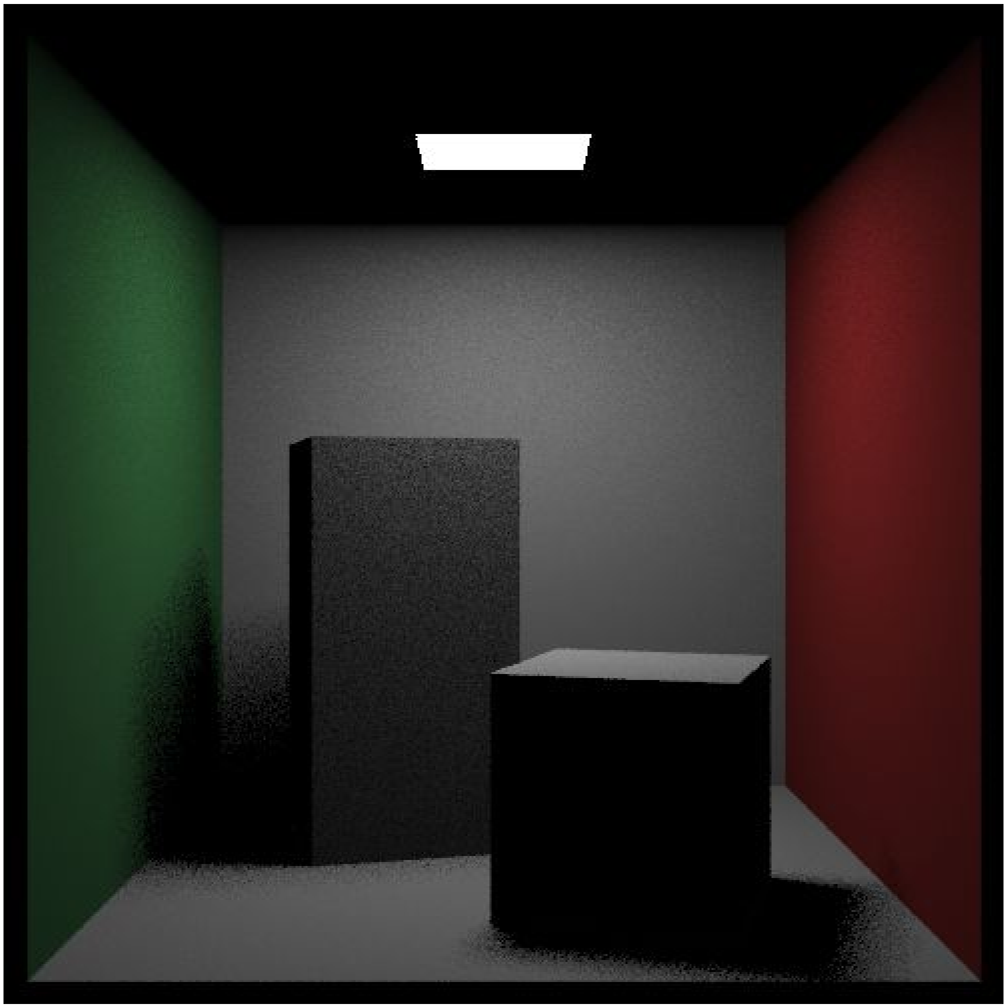
```
class xz_rect: public hitable  {
    public:
        xz_rect() {}
        xz_rect(float _x0, float _x1, float _z0, float _z1, float _k, material *mat) : x0(
_x0), x1(_x1), z0(_z0), z1(_z1), k(_k), mp(mat) {};
        virtual bool hit(const ray& r, float t0, float t1, hit_record& rec) const;
        virtual bool bounding_box(float t0, float t1, aabb& box) const {
            box =  aabb(vec3(x0,k-0.0001,z0), vec3(x1, k+0.0001, z1));
            return true;
        }
        virtual float  pdf_value(const vec3& o, const vec3& v) const {
            hit_record rec;
            if (this->hit(ray(o, v), 0.001, FLT_MAX, rec)) {
                float area = (x1-x0)*(z1-z0);
                float distance_squared = rec.t * rec.t * v.squared_length();
                float cosine = fabs(dot(v, rec.normal) / v.length());
                return  distance_squared / (cosine * area);
            }
            else
                return 0;
        }
        virtual vec3 random(const vec3& o) const {
            vec3 random_point = vec3(x0 + drand48()*(x1-x0), k,  z0 + drand48()*(z1-z0));
            return random_point - o;
        }
        material  *mp;
        float x0, x1, z0, z1, k;
};
```

And then change *color()* :

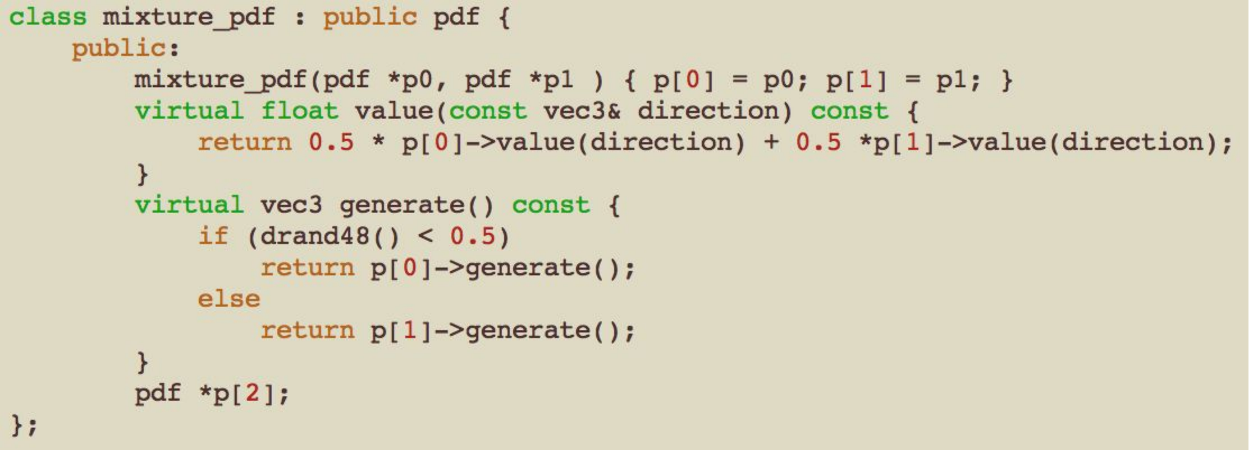


```
vec3 color(const ray& r, hitable *world, int depth) {
    hit_record rec;
    if (world->hit(r, 0.001, MAXFLOAT, rec)) {
        ray scattered;
        vec3 attenuation;
        vec3 emitted = rec.mat_ptr->emitted(r, rec, rec.u, rec.v, rec.p);
        float pdf_val;
        vec3 albedo;
        if (depth < 50 && rec.mat_ptr->scatter(r, rec, albedo, scattered, pdf_val)) {
             hitable *light_shape = new xz_rect(213, 343, 227, 332, 554, 0);
             hitable_pdf p(light_shape, rec.p);
             scattered = ray(rec.p, p.generate(), r.time());
             pdf_val = p.value(scattered.direction());
             return emitted + albedo*rec.mat_ptr->scattering_pdf(r, rec, scattered)*color
(scattered, world, depth+1) / pdf_val;
        }
        else
            return emitted;
    }
    else
        return vec3(0,0,0);
}
```

At 10 samples per pixel we get:

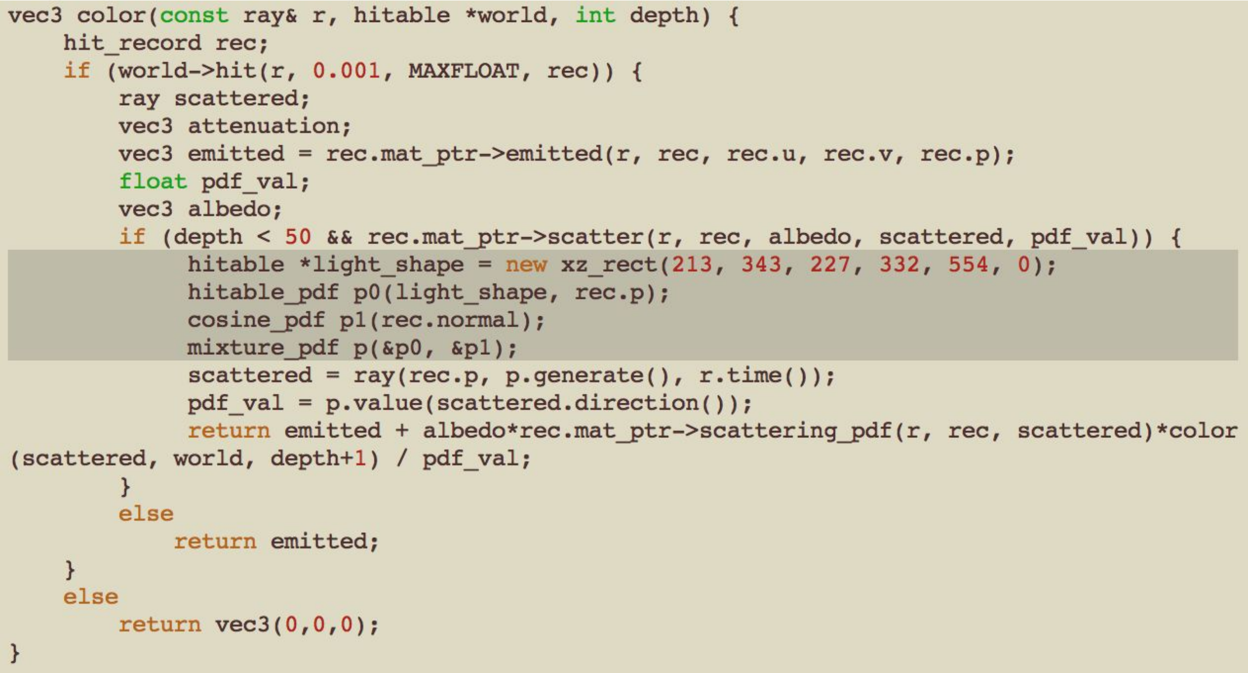


Now we would like to do a mixture density of the cosine and light sampling. The mixture density class is straightforward:



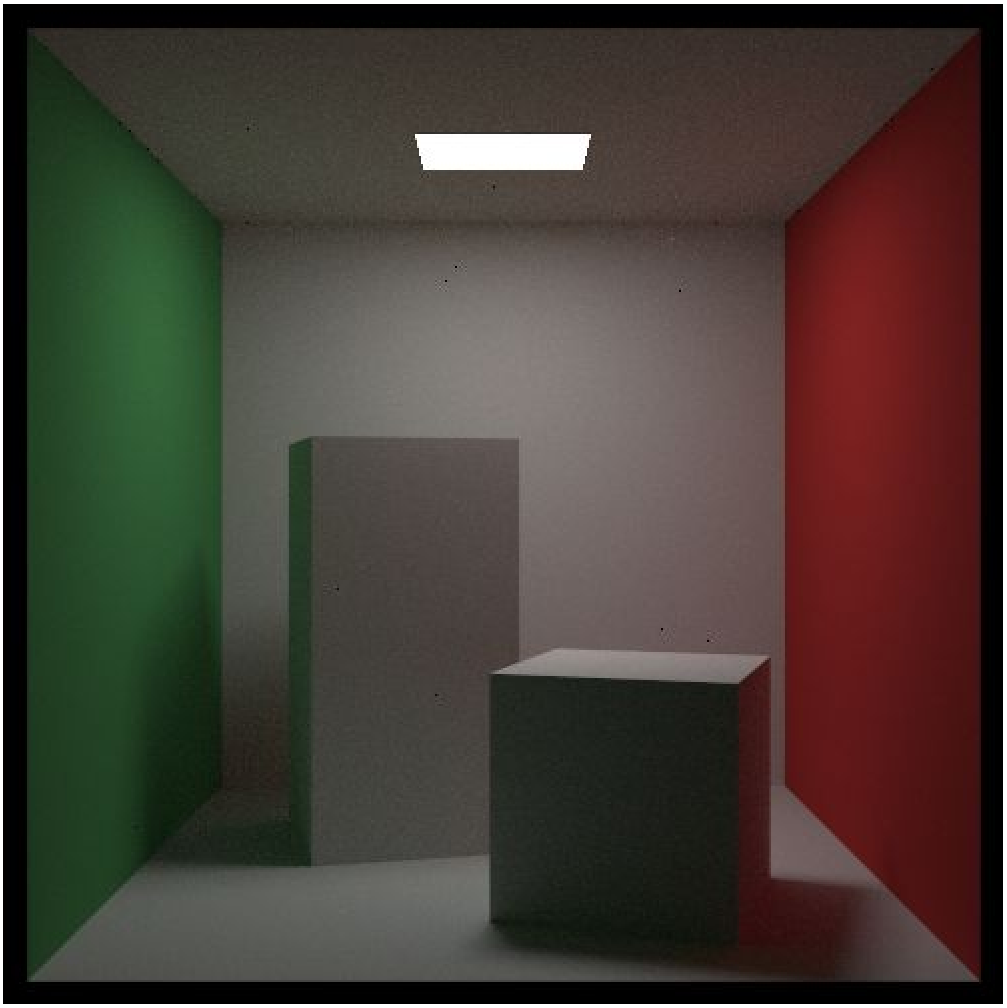
```
class mixture_pdf : public pdf {
    public:
        mixture_pdf(pdf *p0, pdf *p1 ) { p[0] = p0; p[1] = p1; }
        virtual float value(const vec3& direction) const {
            return 0.5 * p[0]->value(direction) + 0.5 *p[1]->value(direction);
        }
        virtual vec3 generate() const {
            if (drand48() < 0.5)
                return p[0]->generate();
            else
                return p[1]->generate();
        }
        pdf *p[2];
};
```

And plugging it into *color()*:



```
vec3 color(const ray& r, hitable *world, int depth) {
    hit_record rec;
    if (world->hit(r, 0.001, MAXFLOAT, rec)) {
        ray scattered;
        vec3 attenuation;
        vec3 emitted = rec.mat_ptr->emitted(r, rec, rec.u, rec.v, rec.p);
        float pdf_val;
        vec3 albedo;
        if (depth < 50 && rec.mat_ptr->scatter(r, rec, albedo, scattered, pdf_val)) {
             hitable *light_shape = new xz_rect(213, 343, 227, 332, 554, 0);
             hitable_pdf p0(light_shape, rec.p);
             cosine_pdf p1(rec.normal);
             mixture_pdf p(&p0, &p1);
             scattered = ray(rec.p, p.generate(), r.time());
             pdf_val = p.value(scattered.direction());
             return emitted + albedo*rec.mat_ptr->scattering_pdf(r, rec, scattered)*color
(scattered, world, depth+1) / pdf_val;
        }
        else
            return emitted;
    }
    else
        return vec3(0,0,0);
}
```

1000 samples per pixel yields:



We’ve basically gotten this same picture (with different levels of noise) with several different sampling patterns. It looks like the original picture was slightly wrong! Note by “wrong” here I mean not a correct Lambertian picture. But Lambertian is just an ideal approximation to matte, so our original picture was some other accidental approximation to matte. I don’t think the new one is any better, but we can at least compare it more easily with other Lambertian renderers.

Chapter 9: Some Architectural Decisions

I won't write any code in this chapter. We’re at a crossroads where I need to make some architectural decisions. The mixture-density approach is to not have traditional shadow rays and is something I personally like, because in addition to lights you can sample windows or bright cracks under doors or whatever else you think might be bright. But most programs branch, and send one or more terminal rays to lights explicitly and one according to the reflective distribution of the surface. This could be a time you want faster convergence on more restricted scenes and add shadow rays; that's a personal design preference.

There are some other issues with the code.

The *pdf* construction is hard coded in the *color()* function and we should clean that up. Probably we should pass something into *color* about the lights. Unlike bvh construction, we should be careful about memory leaks as there are an unbounded number of samples.

The specular rays (glass and metal) are no longer supported. The math would work out if we just made their scattering function a *delta function*. But that would be floating point disaster. We could either separate out specular reflections, or have surface roughness never be zero and have almost-mirrors that look perfectly smooth but don't generate NaNs. I don't have an opinion on which way to do it-- I have tried both and they both have their advantages-- but we have smooth metal and glass code anyway, so I add perfect specular surfaces that do not do explicit *f()/p()* calculations.

We also lack a real background function infrastructure in case we want to add an environment map or more interesting functional background. Some environment maps are *HDR* (the R/G/B components are floats rather than 0-255 bytes usually interpreted as 0-1). Our output has been HDR all along and we've just been truncating it.

Finally, our renderer is RGB and a more physically-based one-- like an automobile manufacturer might use-- would probably need to use spectral colors and maybe even polarization. For a movie renderer, you would probably want RGB. You can make a hybrid renderer that has both modes, but that is of course harder. I'm going to stick to RGB for now, but I will revisit this near the end of the book.

Chapter 10: Cleaning Up pdf Management.

So far I have the color function create two hard-coded *pdfs*:

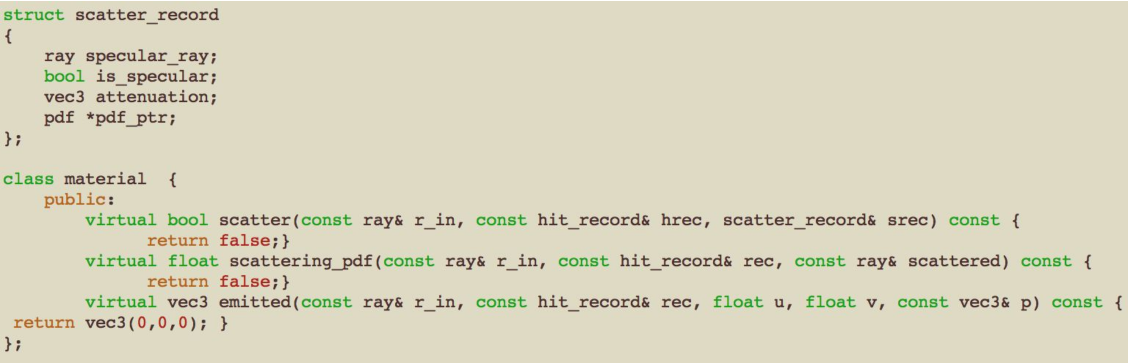
1. *p0()* related to the shape of the light

2. *p1()* related to the normal vector and type of surface

We can pass information about the light (or whatever *hitable* we want to sample) into the *color* function, and we can ask the *material* function for a *pdf* (we would have to instrument it to do that).   We can also either ask *hit* function or the *material* class to supply whether there is a specular vector.

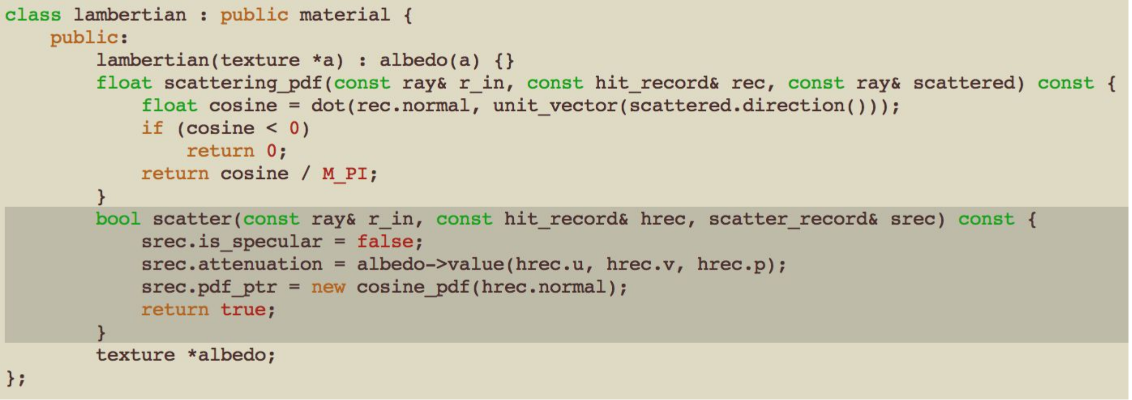
One thing we would like to allow for is a material like varnished wood that is partially ideal specular (the polish) and partially diffuse (the wood). Some renderers have the material generate two rays: one specular and one diffuse. I am not fond of branching, so I would rather have the material randomly decide whether it is diffuse or specular. The catch with that approach is that we need to be careful when we ask for the *pdf* value and be aware of whether for this evaluation of *color()* it is diffuse or specular.   Fortunately, we know that we should only call the *pdf value()* if it is diffuse so we can handle that implicitly.

We can redesign *material* and stuff all the new arguments into a *struct* like we did for *hitable*:



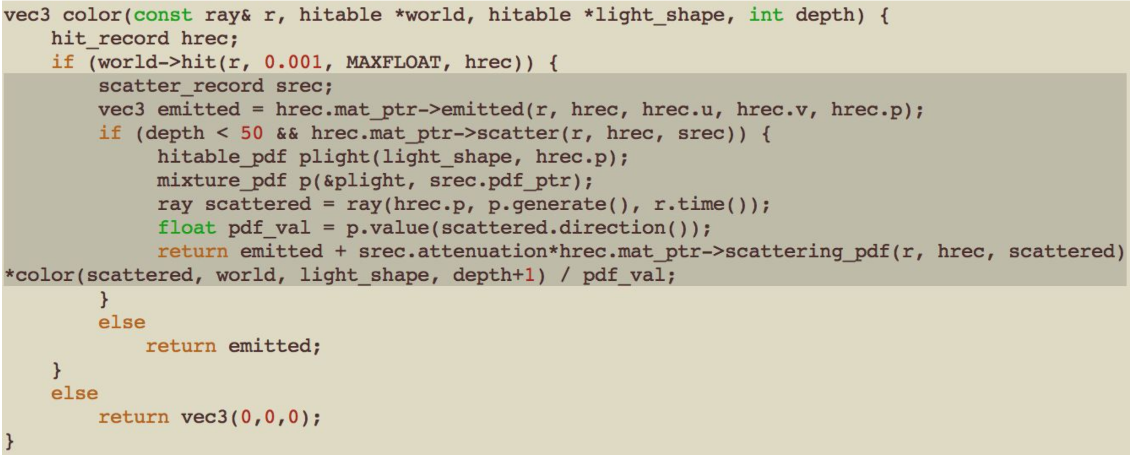
```
struct scatter_record
{
    ray specular_ray;
    bool is_specular;
    vec3 attenuation;
    pdf *pdf_ptr;
};

class material  {
    public:
        virtual bool scatter(const ray& r_in, const hit_record& hrec, scatter_record& srec) const {
              return false;}
        virtual float scattering_pdf(const ray& r_in, const hit_record& rec, const ray& scattered) const {
              return false;}
        virtual vec3 emitted(const ray& r_in, const hit_record& rec, float u, float v, const vec3& p) const {
 return vec3(0,0,0); }
};
```

The Lambertian material becomes simpler (highlighted where the change is):



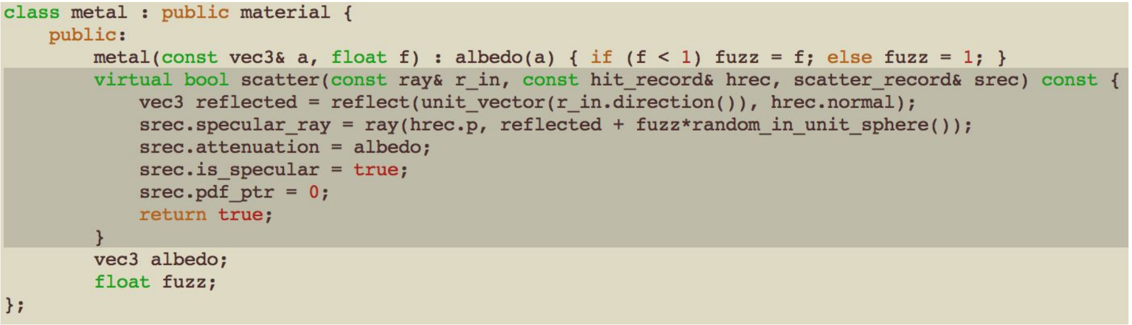
```
class lambertian : public material {
    public:
        lambertian(texture *a) : albedo(a) {}
        float scattering_pdf(const ray& r_in, const hit_record& rec, const ray& scattered) const {
            float cosine = dot(rec.normal, unit_vector(scattered.direction()));
            if (cosine < 0)
                return 0;
            return cosine / M_PI;
        }
        bool scatter(const ray& r_in, const hit_record& hrec, scatter_record& srec) const {
            srec.is_specular = false;
            srec.attenuation = albedo->value(hrec.u, hrec.v, hrec.p);
            srec.pdf_ptr = new cosine_pdf(hrec.normal);
            return true;
        }
        texture *albedo;
};
```

And *color()* changes are small:



```
vec3 color(const ray& r, hitable *world, hitable *light_shape, int depth) {
    hit_record hrec;
    if (world->hit(r, 0.001, MAXFLOAT, hrec)) {
        scatter_record srec;
        vec3 emitted = hrec.mat_ptr->emitted(r, hrec, hrec.u, hrec.v, hrec.p);
        if (depth < 50 && hrec.mat_ptr->scatter(r, hrec, srec)) {
             hitable_pdf plight(light_shape, hrec.p);
             mixture_pdf p(&plight, srec.pdf_ptr);
             ray scattered = ray(hrec.p, p.generate(), r.time());
             float pdf_val = p.value(scattered.direction());
             return emitted + srec.attenuation*hrec.mat_ptr->scattering_pdf(r, hrec, scattered)
*color(scattered, world, light_shape, depth+1) / pdf_val;
        }
        else
            return emitted;
    }
    else
        return vec3(0,0,0);
}
```

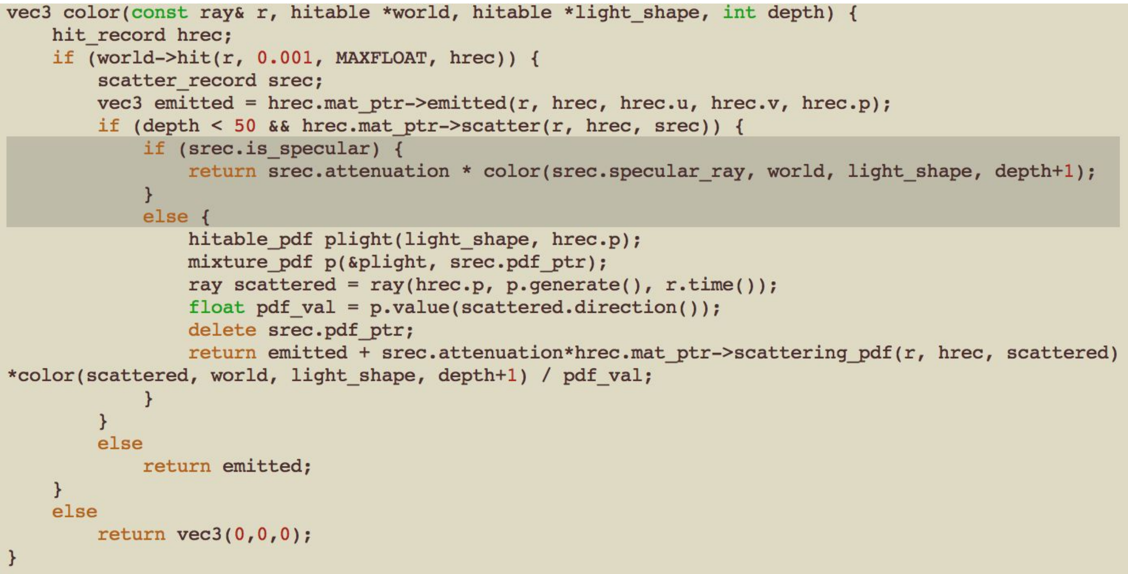
We have not yet dealt with specular surfaces, nor instances that mess with the surface normal, and we have added a memory leak by calling *new* in Lambertian material. But this design is clean overall, and those are all fixable. For now, I will just fix *specular*. Metal is easy to fix.



```
class metal : public material {
    public:
        metal(const vec3& a, float f) : albedo(a) { if (f < 1) fuzz = f; else fuzz = 1; }
        virtual bool scatter(const ray& r_in, const hit_record& hrec, scatter_record& srec) const {
            vec3 reflected = reflect(unit_vector(r_in.direction()), hrec.normal);
            srec.specular_ray = ray(hrec.p, reflected + fuzz*random_in_unit_sphere());
            srec.attenuation = albedo;
            srec.is_specular = true;
            srec.pdf_ptr = 0;
            return true;
        }
        vec3 albedo;
        float fuzz;
};
```

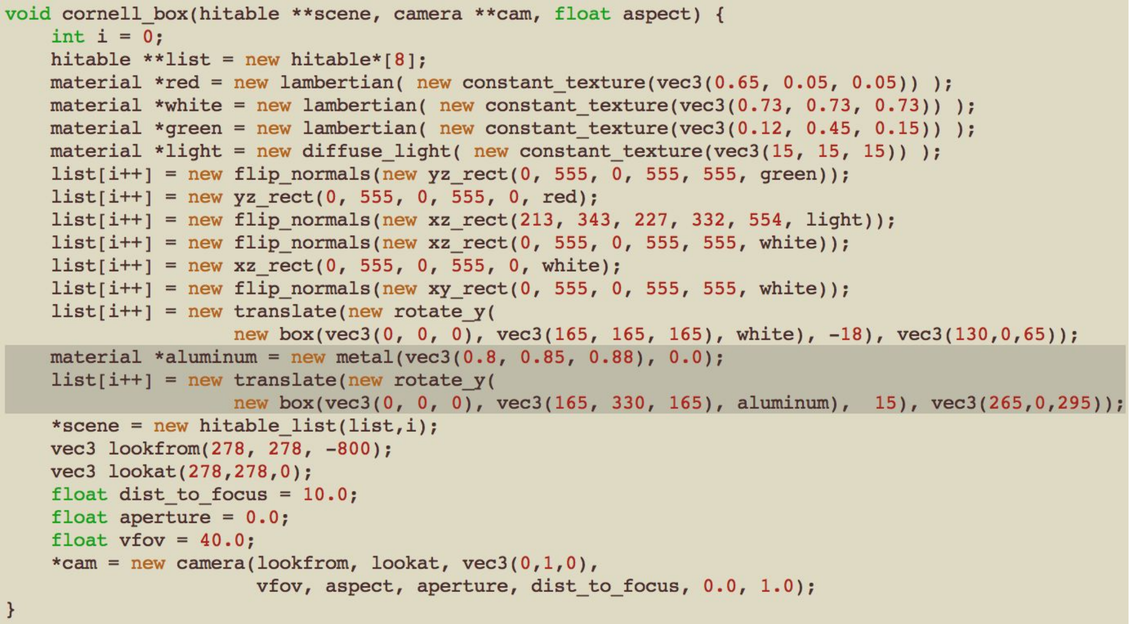
Note that if fuzziness is high, this surface isn’t ideally specular, but the implicit sampling works just like it did before.

Color just needs a new case to generate an implicitly sampled ray:



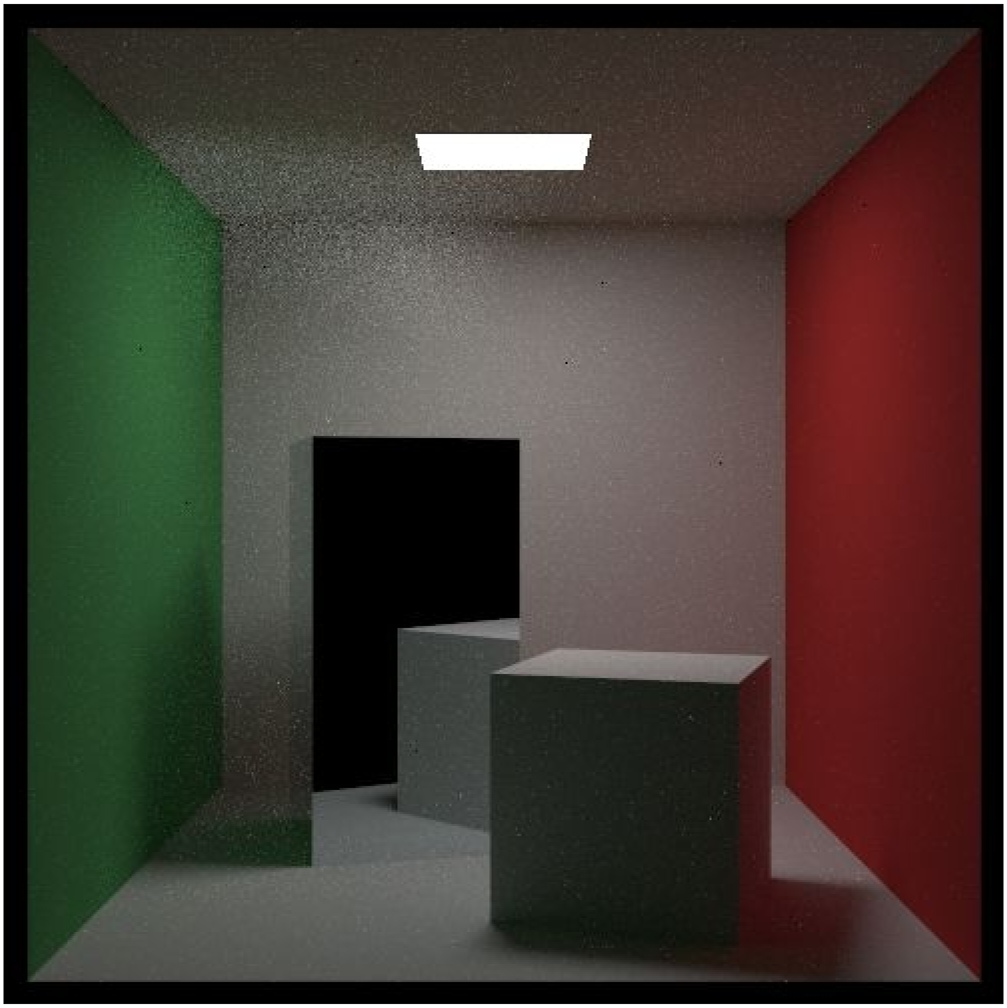
```
vec3 color(const ray& r, hitable *world, hitable *light_shape, int depth) {
    hit_record hrec;
    if (world->hit(r, 0.001, MAXFLOAT, hrec)) {
        scatter_record srec;
        vec3 emitted = hrec.mat_ptr->emitted(r, hrec, hrec.u, hrec.v, hrec.p);
        if (depth < 50 && hrec.mat_ptr->scatter(r, hrec, srec)) {
            if (srec.is_specular) {
                return srec.attenuation * color(srec.specular_ray, world, light_shape, depth+1);
            }
            else {
                hitable_pdf plight(light_shape, hrec.p);
                mixture_pdf p(&plight, srec.pdf_ptr);
                ray scattered = ray(hrec.p, p.generate(), r.time());
                float pdf_val = p.value(scattered.direction());
                delete srec.pdf_ptr;
                return emitted + srec.attenuation*hrec.mat_ptr->scattering_pdf(r, hrec, scattered)
*color(scattered, world, light_shape, depth+1) / pdf_val;
            }
        }
        else
            return emitted;
    }
    else
        return vec3(0,0,0);
}
```

We also need to change the block to metal.



```
void cornell_box(hitable **scene, camera **cam, float aspect) {
    int i = 0;
    hitable **list = new hitable*[8];
    material *red = new lambertian( new constant_texture(vec3(0.65, 0.05, 0.05)) );
    material *white = new lambertian( new constant_texture(vec3(0.73, 0.73, 0.73)) );
    material *green = new lambertian( new constant_texture(vec3(0.12, 0.45, 0.15)) );
    material *light = new diffuse_light( new constant_texture(vec3(15, 15, 15)) );
    list[i++] = new flip_normals(new yz_rect(0, 555, 0, 555, 555, green));
    list[i++] = new yz_rect(0, 555, 0, 555, 0, red);
    list[i++] = new flip_normals(new xz_rect(213, 343, 227, 332, 554, light));
    list[i++] = new flip_normals(new xz_rect(0, 555, 0, 555, 555, white));
    list[i++] = new xz_rect(0, 555, 0, 555, 0, white);
    list[i++] = new flip_normals(new xy_rect(0, 555, 0, 555, 555, white));
    list[i++] = new translate(new rotate_y(
                    new box(vec3(0, 0, 0), vec3(165, 165, 165), white), -18), vec3(130,0,65));
    material *aluminum = new metal(vec3(0.8, 0.85, 0.88), 0.0);
    list[i++] = new translate(new rotate_y(
                    new box(vec3(0, 0, 0), vec3(165, 330, 165), aluminum),  15), vec3(265,0,295));
    *scene = new hitable_list(list,i);
    vec3 lookfrom(278, 278, -800);
    vec3 lookat(278,278,0);
    float dist_to_focus = 10.0;
    float aperture = 0.0;
    float vfov = 40.0;
    *cam = new camera(lookfrom, lookat, vec3(0,1,0),
                      vfov, aspect, aperture, dist_to_focus, 0.0, 1.0);
}
```

The resulting image has a noisy reflection on the ceiling because the directions toward the box are not sampled with more density.



We could make the *pdf* include the block. Let's do that instead with a glass sphere because it's easier. When we sample a sphere's solid angle uniformly from a point outside the sphere, we are really just sampling a cone uniformly (the cone is tangent to the sphere). Let's say the code has theta_max. Recall from Chapter 5, that to sample *theta* we have:

*r2 = INTEGRAL_0^theta 2*Pi*f(t)sin(t)*

Here *f(t)* is an as yet uncalculated constant *C*, so:

*r2 = INTEGRAL_0^theta 2*Pi*C*sin(t)*

Doing some algebra/calculus this yields:

r2 = 2*Pi*C*(1-cos(theta))

So

cos(theta) = 1 - r2/(2*Pi*C)

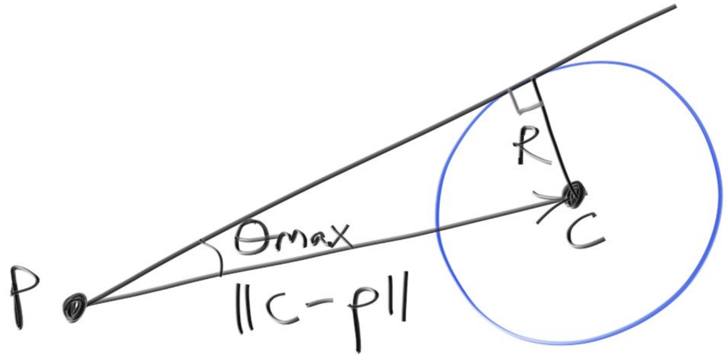
We know that for *R2 = 1* we should get *theta_max,* so we can solve for *C:*

cos(theta) = 1 + r2*(cos(theta_max)-1)

Phi we sample like before, so:

z = cos(theta) = 1 + r2*(cos(theta_max)-1)
x = cos(phi)*sin(theta) = cos(2*Pi*r1)*sqrt(1-z^2)
y = sin(phi)*sin(theta) = sin(2*Pi*r1)*sqrt(1-z^2)

Now what is *theta_max?*



We can see from the figure that *sin(theta_max) = R / length(**c**-**p**).* So*:*

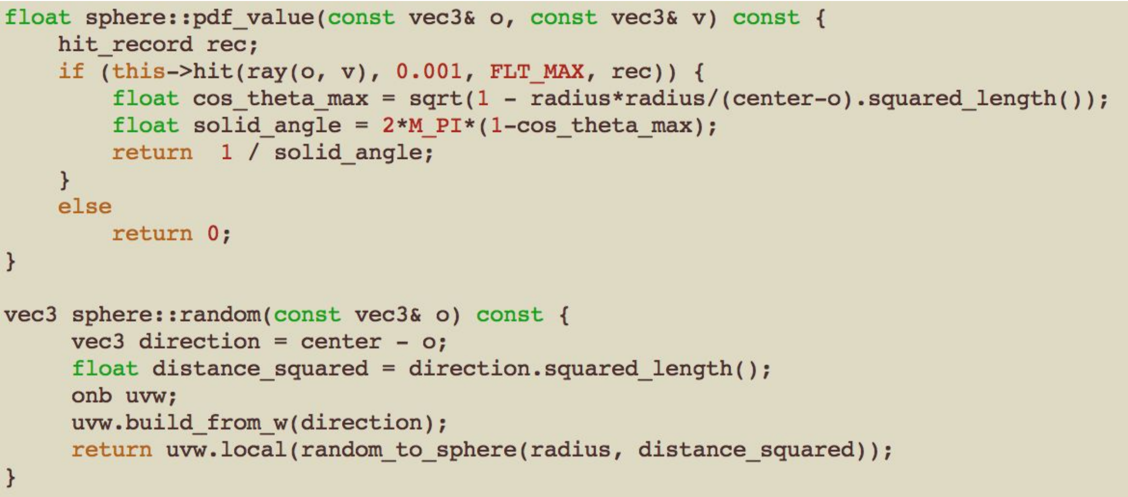
cos(theta_max) = sqrt(1- R^2/lenghth_squared(**c**-**p**))

We also need to evaluate the *pdf* of directions. For directions toward the sphere this is *1/solid_angle.* What is the solid angle of the sphere? It has something to do with the *C* above. It, by definition, is the area on the unit sphere, so the integral is

*Solid_angle = INT_0^2Pi INT_0^theta_max sin(theta) = 2*Pi (1-cos(theta_max))*

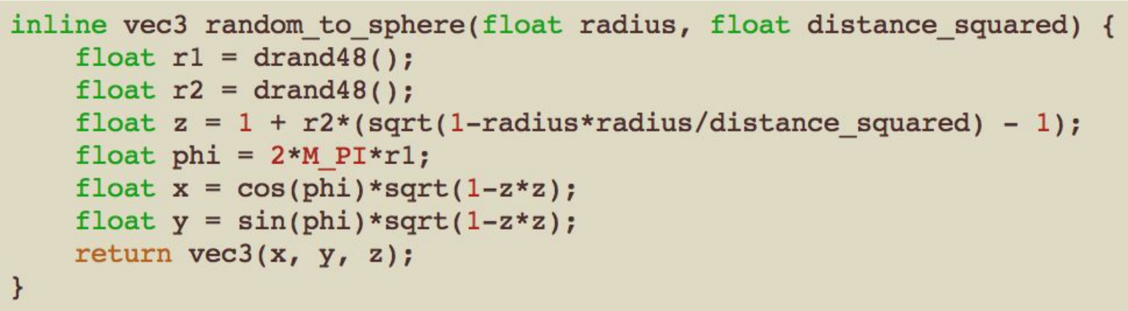
It’s good to check the math on all such calculations. I usually plug in the extreme cases (thank you for that concept, Mr. Horton-- my high school physics teacher). For a zero radius sphere *cos(theta_max) = 0,* and that works. For a sphere tangent at ***p,*** *cos(theta_max) = 0,* and *2*Pi* is the area of a hemisphere, so that works too.

The *sphere* class needs the two *pdf*-related functions:



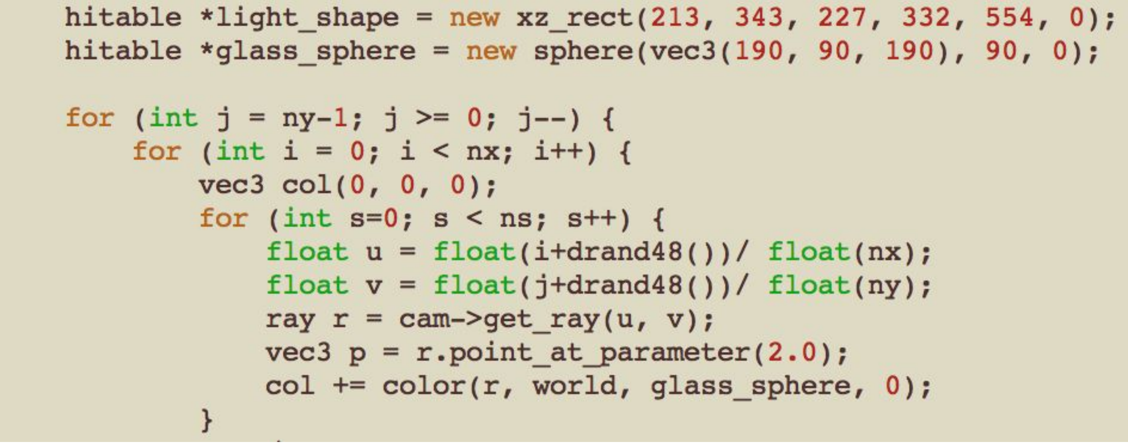
```
float sphere::pdf_value(const vec3& o, const vec3& v) const {
    hit_record rec;
    if (this->hit(ray(o, v), 0.001, FLT_MAX, rec)) {
        float cos_theta_max = sqrt(1 - radius*radius/(center-o).squared_length());
        float solid_angle = 2*M_PI*(1-cos_theta_max);
        return  1 / solid_angle;
    }
    else
        return 0;
}

vec3 sphere::random(const vec3& o) const {
     vec3 direction = center - o;
     float distance_squared = direction.squared_length();
     onb uvw;
     uvw.build_from_w(direction);
     return uvw.local(random_to_sphere(radius, distance_squared));
}
```

With the utility function:



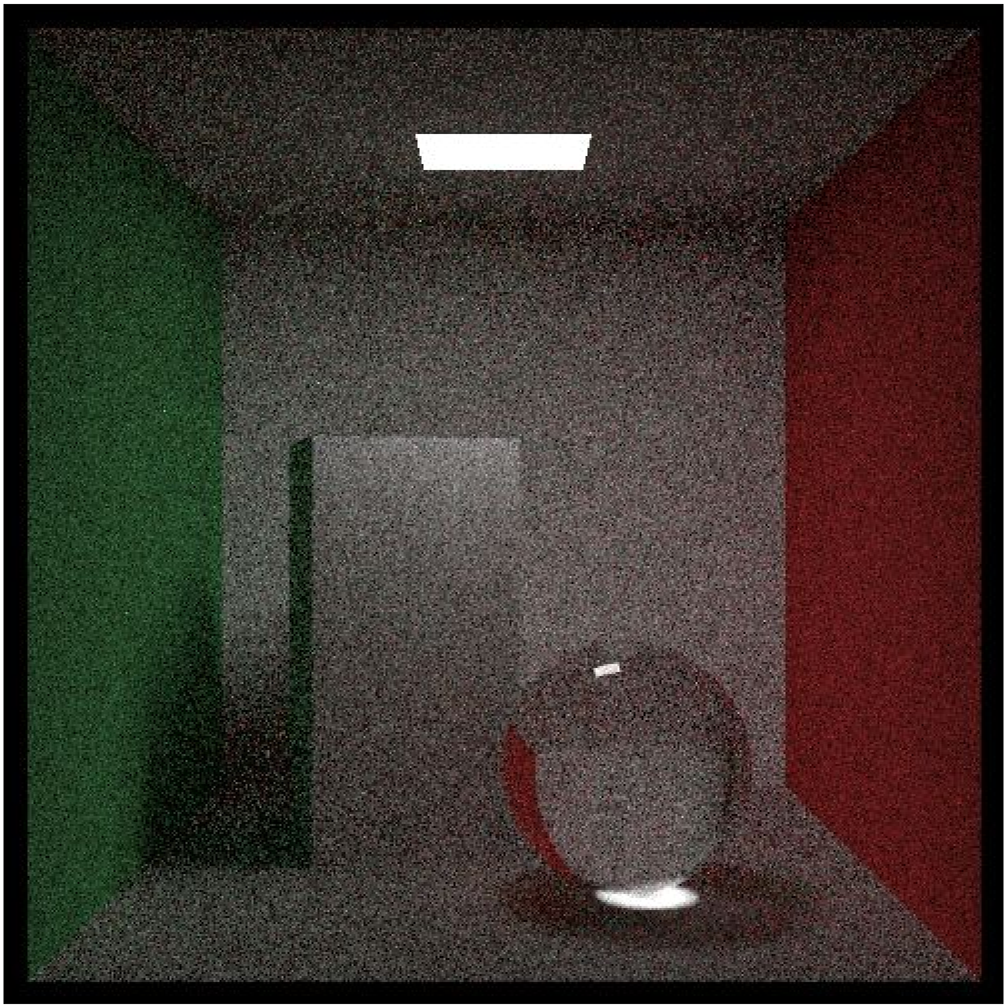
```
inline vec3 random_to_sphere(float radius, float distance_squared) {
     float r1 = drand48();
     float r2 = drand48();
     float z = 1 + r2*(sqrt(1-radius*radius/distance_squared) - 1);
     float phi = 2*M_PI*r1;
     float x = cos(phi)*sqrt(1-z*z);
     float y = sin(phi)*sqrt(1-z*z);
     return vec3(x, y, z);
}
```

We can first try just sampling the sphere rather than the light:

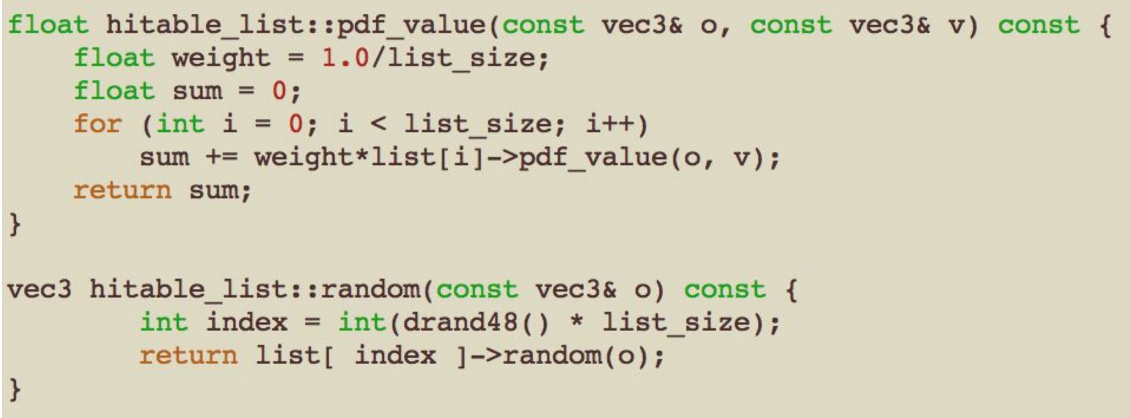


```
    hitable *light_shape = new xz_rect(213, 343, 227, 332, 554, 0);
    hitable *glass_sphere = new sphere(vec3(190, 90, 190), 90, 0);

    for (int j = ny-1; j >= 0; j--) {
        for (int i = 0; i < nx; i++) {
            vec3 col(0, 0, 0);
            for (int s=0; s < ns; s++) {
                float u = float(i+drand48())/ float(nx);
                float v = float(j+drand48())/ float(ny);
                ray r = cam->get_ray(u, v);
                vec3 p = r.point_at_parameter(2.0);
                col += color(r, world, glass_sphere, 0);
            }
```

This yields a noisy box, but the caustic under the sphere is good. It took five times as long as sampling the light did for my code. This is probably because those rays that hit the glass are expensive!

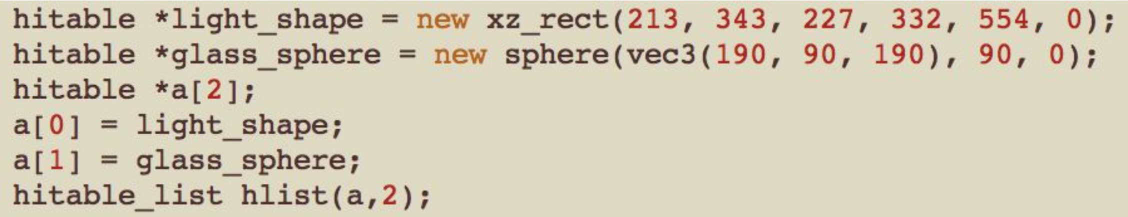


We should probably just sample both the sphere and the light. We can do that by creating a mixture density of their two densities. We could do that in the *color* function by passing a list of hitables in and building a mixture *pdf*, or we could add *pdf* functions to *hitable_list*.   I think both tactics would work fine, but I will go with instrumenting *hitable_list*.



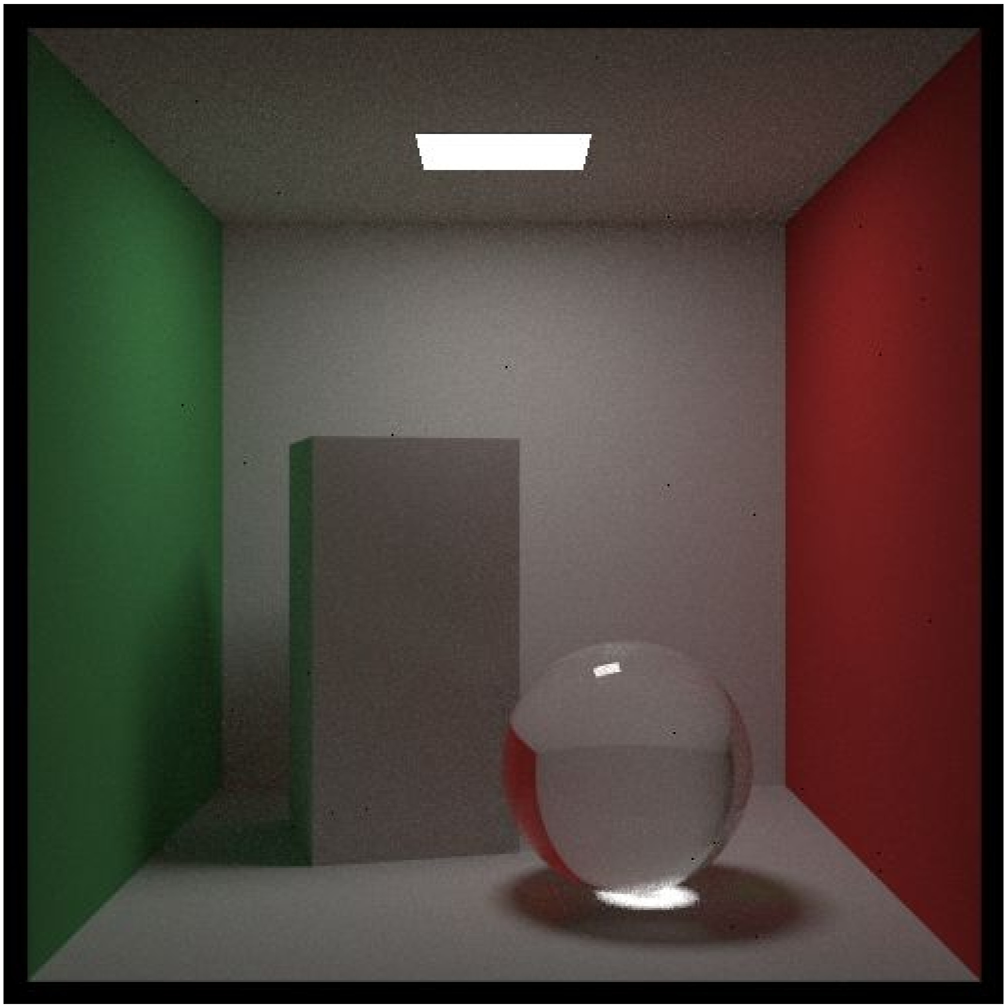
```
float hitable_list::pdf_value(const vec3& o, const vec3& v) const {
    float weight = 1.0/list_size;
    float sum = 0;
    for (int i = 0; i < list_size; i++)
        sum += weight*list[i]->pdf_value(o, v);
    return sum;
}

vec3 hitable_list::random(const vec3& o) const {
        int index = int(drand48() * list_size);
        return list[ index ]->random(o);
}
```

We assemble a list to pass in to *color*.



```
hitable *light_shape = new xz_rect(213, 343, 227, 332, 554, 0);
hitable *glass_sphere = new sphere(vec3(190, 90, 190), 90, 0);
hitable *a[2];
a[0] = light_shape;
a[1] = glass_sphere;
hitable_list hlist(a,2);
```

And we get a decent image with 1000 samples as before:

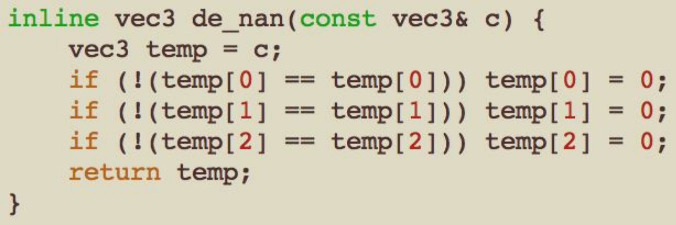


An astute reader pointed out there are some black specks in the image above. All Monte Carlo Ray Tracers have this as a main loop:

pixel_color = average(many many samples)

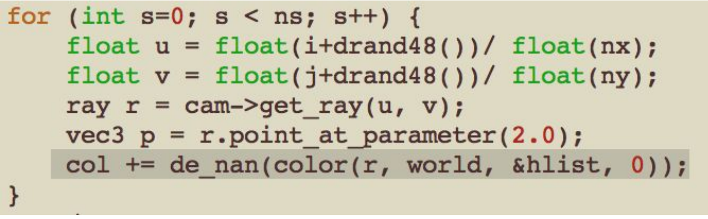
If you find yourself getting some form of acne in the images, and this acne is white or black, so one "bad" sample seems to kill the whole pixel, that sample is probably a huge number or a NaN. This particular acne is probably a NaN. Mine seems to come up once in every 10-100 million rays or so.

So big decision: sweep this bug under the rug and check for NaNs, or just kill NaNs and hope this doesn't come back to bite us later. I will always opt for the lazy strategy, especially when I know floating point is hard. First, how do we check for a NaN? The one thing I always remember for NaNs is that any *if* test with a NaN in it is false. This leads to the common trick:



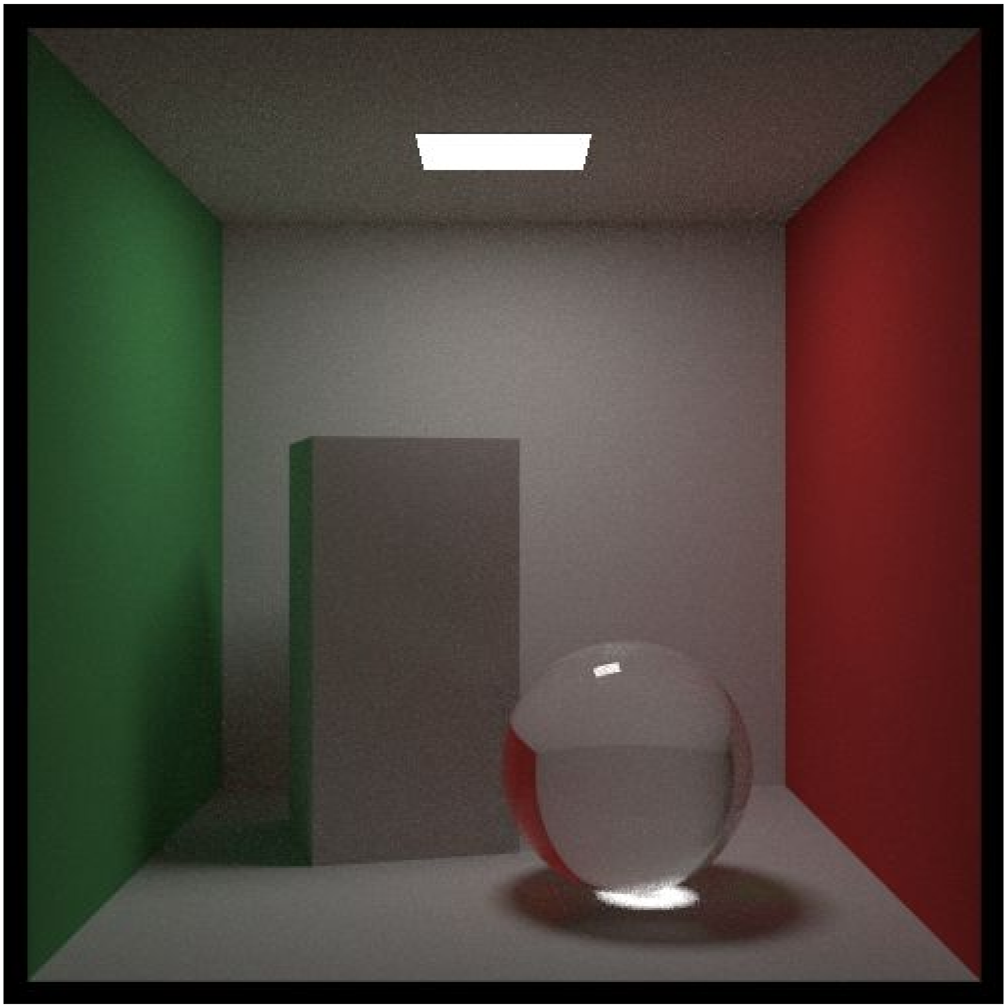
```
inline vec3 de_nan(const vec3& c) {
    vec3 temp = c;
    if (!(temp[0] == temp[0])) temp[0] = 0;
    if (!(temp[1] == temp[1])) temp[1] = 0;
    if (!(temp[2] == temp[2])) temp[2] = 0;
    return temp;
}
```

And we can insert that in the main loop:



```
for (int s=0; s < ns; s++) {
    float u = float(i+drand48())/ float(nx);
    float v = float(j+drand48())/ float(ny);
    ray r = cam->get_ray(u, v);
    vec3 p = r.point_at_parameter(2.0);
    col += de_nan(color(r, world, &hlist, 0));
}
```

Happily, the black specks are gone:



Chapter 11: The Rest of Your Life

The purpose of this book was to show the details of dotting all the i’s of the math on one way of organizing a physically-based renderer’s sampling approach. Now you can explore a lot of different potential paths.

If you want to explore Monte Carlo methods, look into bidirectional and path spaced approaches such as Metropolis. Your probability space won't be over solid angle but will instead be over path space, where a path is a multidimensional point in a high-dimensional space. Don’t let that scare you-- if you can describe an object with an array of numbers, mathematicians call it a *point* in the *space* of all possible arrays of such points. That’s not just for show. Once you get a clean abstraction like that, your code can get clean too. Clean abstractions are what programming is all about!

If you want to do movie renderers, look at the papers out of studios and Solid Angle. They are surprisingly open about their craft.

If you want to do high-performance ray tracing, look first at papers from Intel and NVIDIA. Again, they are surprisingly open.

If you want to do hard-core physically-based renderers, convert your renderer from RGB to spectral.   I am a big fan of each ray having a random wavelength and almost all of the RGBs in your program turning into floats. It sounds inefficient, but it isn't!

Regardless of what direction you take, add a glossy BRDF model. There are many to choose from and each has its advantages.

Finally, please send me your images and fixes and complaints! I will maintain links to further reading etc. at in1weekend.com

Have fun!

Peter Shirley
Salt Lake City, March, 2016