Ray Tracing: The Rest of Your Life

Peter Shirley
edited by Steve Hollasch and Trevor David Black

Version 3.2.3, 2020-12-07

Contents

1. 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, https://in1weekend.blogspot.com/ will have further readings and references.

Thanks to everyone who lent a hand on this project. You can find them in the acknowledgments section at the end of this book.

2. 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.

2.1. Estimating Pi

As an example, let's estimate π. 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:

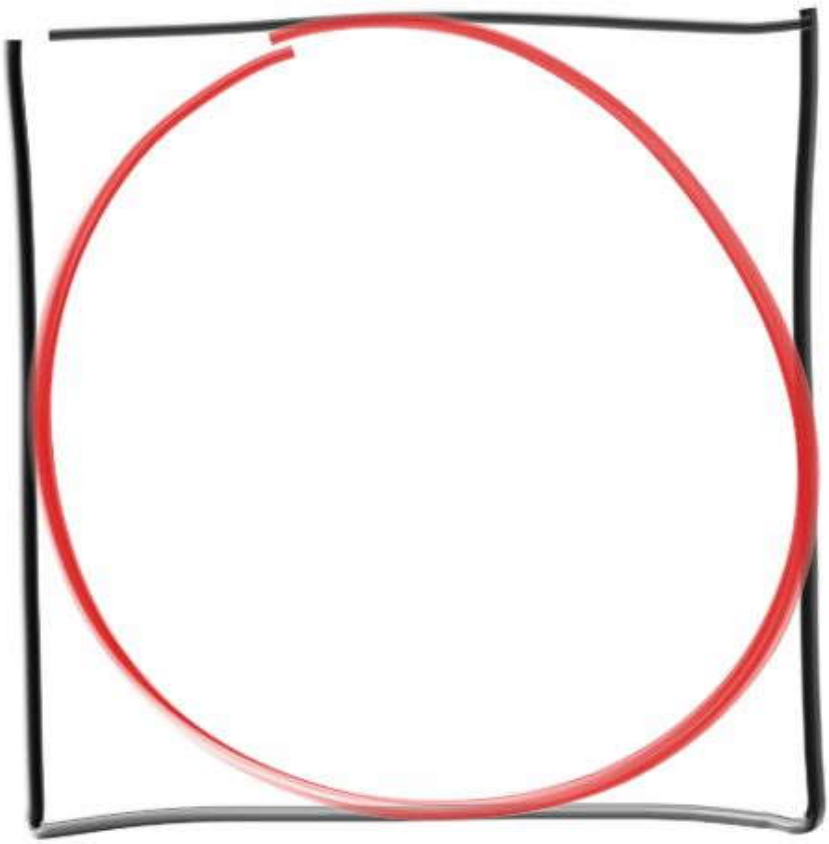

Figure 1: *Estimating π with a circle inside a square*

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:

$$\frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$$

Since the r cancels out, we can pick whatever is computationally convenient. Let's go with $r = 1$, centered at the origin:

```
#include "rtweekend.h"

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

int main() {
    int N = 1000;
    int inside_circle = 0;
    for (int i = 0; i < N; i++) {
        auto x = random_double(-1,1);
        auto y = random_double(-1,1);
        if (x*x + y*y < 1)
            inside_circle++;
    }
    std::cout << std::fixed << std::setprecision(12);
    std::cout << "Estimate of Pi = " << 4*double(inside_circle) / N << '\n';
}
```

Listing 1: `[pi.cc]` *Estimating π*

The answer of π found will vary from computer to computer based on the initial random seed. On my computer, this gives me the answer `Estimate of Pi = 3.0880000000`

2.2. Showing Convergence

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

```
#include "rtweekend.h"

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

int main() {
    int inside_circle = 0;
    int runs = 0;
    std::cout << std::fixed << std::setprecision(12);
    while (true) {
        runs++;
        auto x = random_double(-1,1);
        auto y = random_double(-1,1);
        if (x*x + y*y < 1)
            inside_circle++;

        if (runs % 100000 == 0)
            std::cout << "Estimate of Pi = "
                      << 4*double(inside_circle) / runs
                      << '\n';
    }
}
```

Listing 2: `[pi.cc]` *Estimating π, v2*

2.3. Stratified Samples (Jittering)

We get very quickly near π, 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:

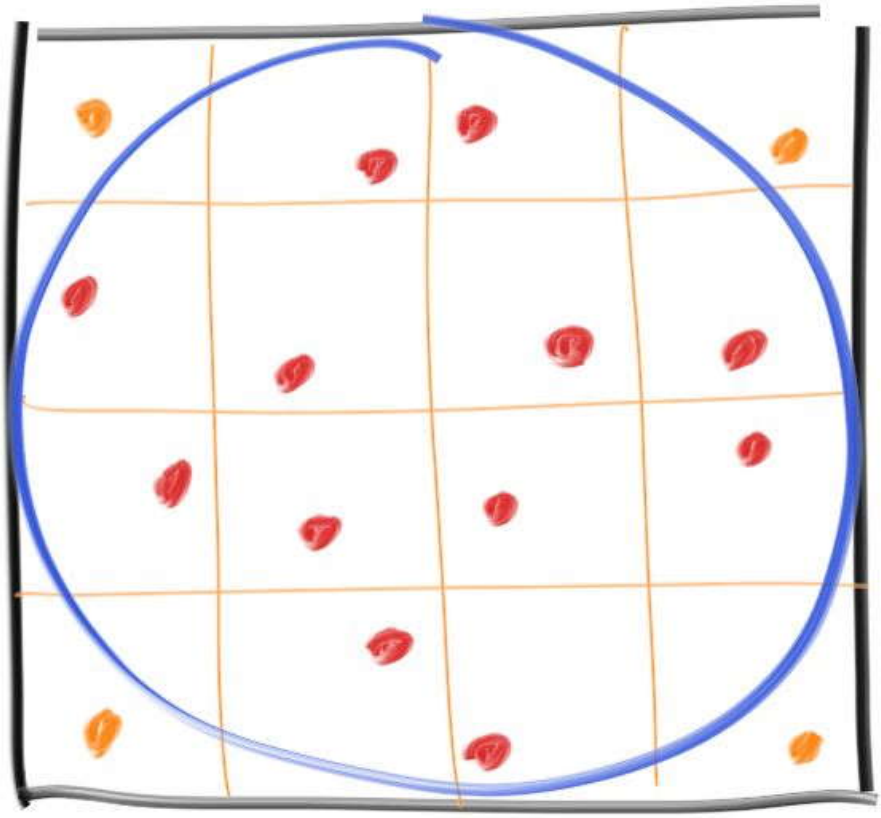

Figure 2: *Sampling areas with jittered points*

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 "rtweekend.h"

#include <iostream>
#include <iomanip>

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++) {
            auto x = random_double(-1,1);
            auto y = random_double(-1,1);
            if (x*x + y*y < 1)
                inside_circle++;
            x = 2*((i + random_double()) / sqrt_N) - 1;
            y = 2*((j + random_double()) / sqrt_N) - 1;
            if (x*x + y*y < 1)
                inside_circle_stratified++;
        }
    }

    auto N = static_cast<double>(sqrt_N) * sqrt_N;
    std::cout << std::fixed << std::setprecision(12);
    std::cout
        << "Regular    Estimate of Pi = "
        << 4*double(inside_circle) / (sqrt_N*sqrt_N) << '\n'
        << "Stratified Estimate of Pi = "
        << 4*double(inside_circle_stratified) / (sqrt_N*sqrt_N) << '\n';
}
```

Listing 3: `[pi.cc]` *Estimating π, v3*

On my computer, I get:

```
Regular    Estimate of Pi = 3.14151480
Stratified Estimate of Pi = 3.14158948
```

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.

3. One Dimensional MC Integration

Integration is all about computing areas and volumes, so we could have framed chapter 2 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.

3.1. Integrating x²

Let's look at a classic integral:

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

In computer sciency notation, we might write this as:

$$I = \text{area}(x^2, 0, 2)$$

We could also write it as:

$$I = 2 \cdot \text{average}(x^2, 0, 2)$$

This suggests a MC approach:

```
#include "rtweekend.h"

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

int main() {
    int N = 1000000;
    auto sum = 0.0;
    for (int i = 0; i < N; i++) {
        auto x = random_double(0,2);
        sum += x*x;
    }
    std::cout << std::fixed << std::setprecision(12);
    std::cout << "I = " << 2 * sum/N << '\n';
}
```

Listing 4: `[integrate_x_sq.cc]` *Integrating* x^2

This, as expected, produces approximately the exact answer we get with algebra, $I = 8/3$. 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 `ray_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 that small light sources create too much noise. This is because our uniform sampling doesn't sample these light sources often enough. Light sources are only sampled if a ray scatters toward them, but this can be unlikely for a small light, or a light that is far away. We could lessen this problem if we sent more random samples toward this light, but this will cause the scene to be inaccurately bright. We can remove this inaccuracy by downweighting these 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*.

3.2. Density Functions

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

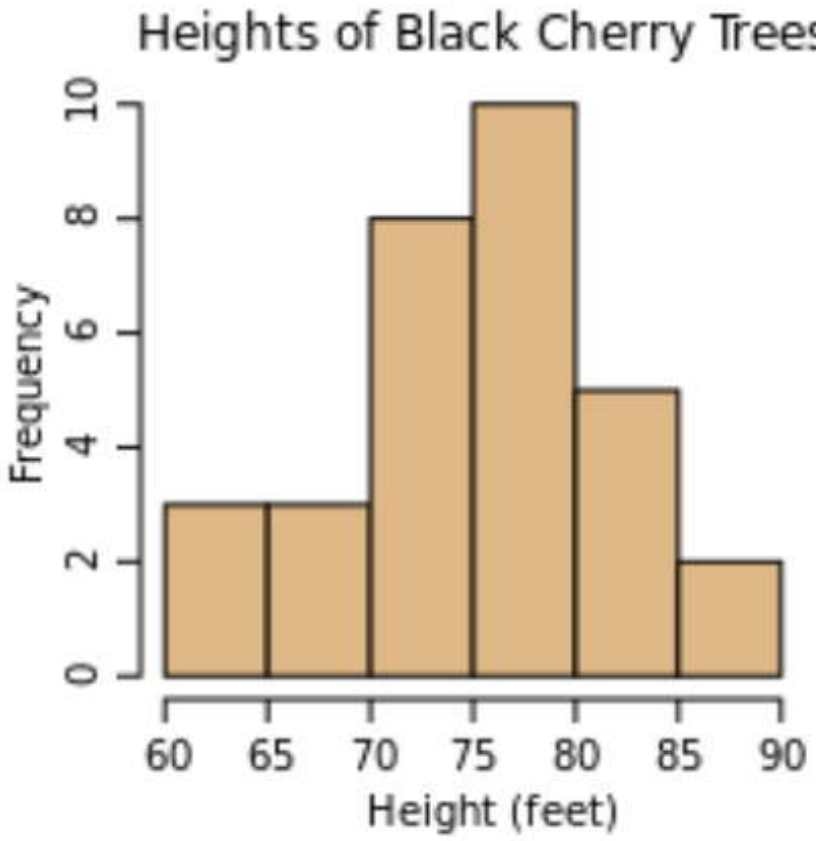

Figure 3: *Histogram example*

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:

$$\text{bin-height} = \frac{(\text{Fraction of trees between height } H \text{ and } H')}{(H - H')}$$

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

$$\text{Probability a random tree is between } H \text{ and } H' = \text{bin-height} \cdot (H - H')$$

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.

3.3. Constructing a PDF

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?

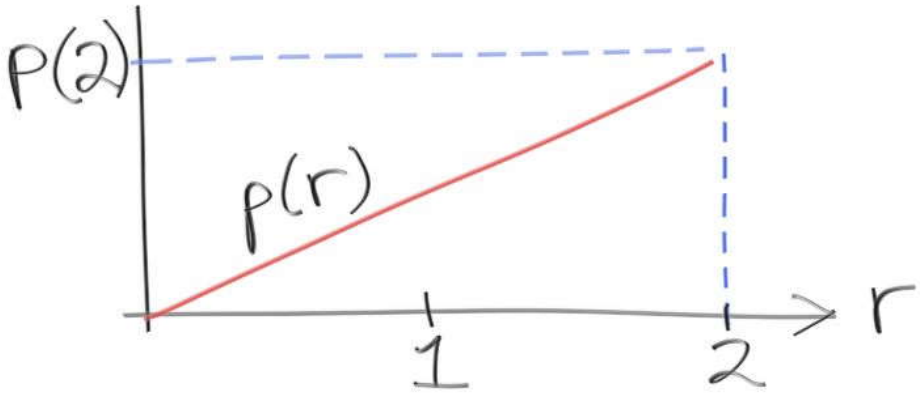

Figure 4: *A linear PDF*

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 (x_0, x_1):

$$\text{Probability } x_0 < r < x_1 = C \cdot \text{area}(p(r), x_0, x_1)$$

where C is a scaling constant. We may as well make $C = 1$ for cleanliness, and that is exactly what is done in probability. We also know the probability r has the value 1 somewhere, so for this case

$$\text{area}(p(r), 0, 2) = 1$$

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

$$area(C'r, 0, 2) = \int_0^2 C'r dr = \frac{C'r^2}{2}\Big|_{r=0}^{r=2} = \frac{C' \cdot 2^2}{2} - \frac{C' \cdot 0^2}{2} = 2C'$$

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 = random_double()` 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 $e = f(d) = d^2$. This is no longer a uniform PDF. The PDF of e will be bigger near 1 than it is near 0 (squaring a number between 0 and 1 makes it smaller). To convert this general observation to a function, we need the cumulative probability distribution function $P(x)$:

$$P(x) = \text{area}(p, -\infty, x)$$

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:

$$P(x) = 0 : x < 0$$

$$P(x) = \frac{x^2}{4} : 0 < x < 2$$

$$P(x) = 1 : x > 2$$

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 $x = 1.0$, we get:

$$P(1.0) = \frac{1}{4}$$

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(random_double())` we get a return value with a PDF $\frac{x^2}{4}$. We don't know what that is, but we do know that 25% of what it returns should be less than 1.0, and 75% should be above 1.0. 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, if we have PDF $p()$ and cumulative distribution function $P()$, we can use this "inverse function" with a random number to get what we want:

$$e = P^{-1}(\text{random\_double}())$$

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

$$y = \frac{x^2}{4}$$

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

$$x = \sqrt{4y}$$

Thus our random number with density p is found with:

$$e = \sqrt{4 \cdot \text{random\_double}()}$$

Note that this ranges from 0 to 2 as hoped, and if we check our work by replacing `random_double()` with $\frac{1}{4}$ we get 1 as expected.

We can now sample our old integral

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

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 $1/pdf$:

```
inline double pdf(double x) {
    return 0.5*x;
}

int main() {
    int N = 1000000;
    auto sum = 0.0;
    for (int i = 0; i < N; i++) {
        auto x = sqrt(random_double(0,4));
        sum += x*x / pdf(x);
    }
    std::cout << std::fixed << std::setprecision(12);
    std::cout << "I = " << sum/N << '\n';
}
```

Listing 5: `[integrate_x_sq.cc]` *Integrating x^2 with PDF*

3.4. Importance Sampling

Since we are sampling more where the integrand is big, we might expect less noise and thus faster convergence. In effect, we are steering our samples toward the parts of the distribution that are more *important*. 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 = $1/2$ over the range [0,2] we can use the machinery to get `x = random_double(0,2)`, and the code is:

```
inline double pdf(double x) {
    return 0.5;
}

int main() {
    int N = 1000000;
    auto sum = 0.0;
    for (int i = 0; i < N; i++) {
        auto x = random_double(0,2);
        sum += x*x / pdf(x);
    }
    std::cout << std::fixed << std::setprecision(12);
    std::cout << "I = " << sum/N << '\n';
}
```

Listing 6: `[integrate_x_sq.cc]` *Integrating x^2, v3*

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:

$$p(x) = \frac{3}{8}x^2$$

And we get the corresponding

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

and

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

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:

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

int main() {
    int N = 1;
    auto sum = 0.0;
    for (int i = 0; i < N; i++) {
        auto x = pow(random_double(0,8), 1./3.);
        sum += x*x / pdf(x);
    }
    std::cout << std::fixed << std::setprecision(12);
    std::cout << "I = " << sum/N << '\n';
}
```

Listing 7: `[integrate_x_sq.cc]` *Integrating x^2, final version*

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 $\frac{f(r)}{p(r)}$ where r is a random number with PDF p.

Any choice of PDF p will always converge to the right answer, but the closer that p approximates f, the faster that it will converge.

4. 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:

$$\int cos^2(\theta)$$

By MC integration, we should just be able to sample $\cos^2(\theta)/p(\text{direction})$, but what is *direction* in that context? We could make it based on polar coordinates, so p would be in terms of (θ, ϕ). However you do it, remember that a PDF has to integrate to 1 and represent the relative probability of that direction being sampled. Recall that we have vec3 functions to take uniform random samples in (`random_in_unit_sphere()`) or on (`random_unit_vector()`) a unit sphere.

Now what is the PDF of these uniform points? As a density on the unit sphere, it is $1/\text{area}$ of the sphere or $1/(4\pi)$. If the integrand is $\cos^2(\theta)$, and θ is the angle with the z axis:

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

int main() {
    int N = 1000000;
    auto sum = 0.0;
    for (int i = 0; i < N; i++) {
        vec3 d = random_unit_vector();
        auto cosine_squared = d.z()*d.z();
        sum += cosine_squared / pdf(d);
    }
    std::cout << std::fixed << std::setprecision(12);
    std::cout << "I = " << sum/N << '\n';
}
```

Listing 8: `[sphere_importance.cc]` *Generating importance-sampled points on the unit sphere*

The analytic answer (if you remember enough advanced calc, check me!) is $\frac{4}{3}\pi$, 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 ω and the projected area A on the unit sphere are the same thing.

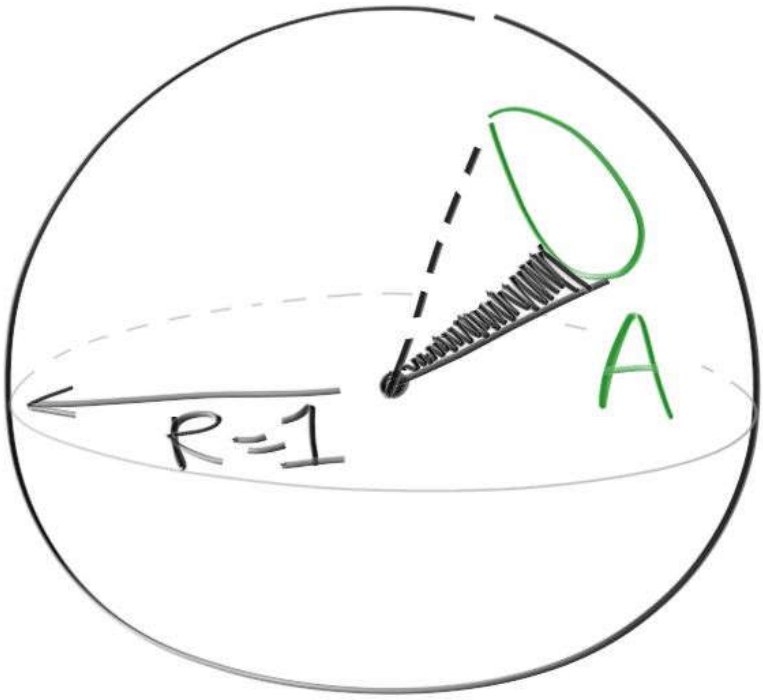

Figure 5: *Solid angle / projected area of a sphere*

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

5. Light Scattering

In this chapter we won't actually program anything. We will set up for a big lighting change in the next chapter.

5.1. Albedo

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 cases it is used to define some form of *fractional reflectance*. This *fractional reflectance* (or albedo) will vary with color and (as we implemented for our glass in book one) can vary with incident direction.

5.2. Scattering

In most physically based renderers, we would use a set of wavelengths for the light color rather than RGB. We can extend our intuition by thinking of R, G, and B as specific algebraic mixtures of 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*, which is the direction of the incoming ray. You can see this varying with incident direction when you look at reflections off a road — they become mirror-like as your viewing angle (incident angle) approaches grazing.

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

$$Color = \int A \cdot s(direction) \cdot \text{color}(direction)$$

Note that A and $s()$ may depend on the view direction or the scattering position (position on a surface or position within a volume). Therefore, the output color may also vary with view direction or scattering position.

5.3. The Scattering PDF

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

$$Color = \frac{A \cdot s(direction) \cdot \text{color}(direction)}{p(direction)}$$

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 θ 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 π.

To see that, remember that in spherical coordinates:

$$dA = \sin(\theta) d\theta d\phi$$

So:

$$Area = \int_0^{2\pi} \int_0^{\pi/2} cos(\theta) sin(\theta) d\theta d\phi = 2\pi \frac{1}{2} = \pi$$

So for a Lambertian surface the scattering PDF is:

$$s(direction) = \frac{\cos(\theta)}{\pi}$$

If we sample using a PDF that equals the scattering PDF:

$$p(direction) = s(direction) = \frac{\cos(\theta)}{\pi}$$

The numerator and denominator cancel out, and we get:

$$Color = A \cdot color(direction)$$

This is exactly what we had in our original `ray_color()` function! However, we need to generalize 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:

$$BRDF = \frac{A \cdot s(direction)}{\cos(\theta)}$$

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*.

6. 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:

$$p(direction) = \frac{1}{2} \cdot \text{Light}(direction) + \frac{1}{2} \cdot \text{pSurface}(direction)$$

As long as the weights are positive and add up to one, any such mixture of PDFs is a PDF. Remember, we can use any PDF: *all PDFs eventually converge to the correct answer*. So, the game is to figure out how to make the PDF larger where the product $s(direction) \cdot color(direction)$ 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.

6.1. Returning to the Cornell Box

Let's do a 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.

```
...
color ray_color(...) {
    ...
}

hittable_list cornell_box() {
    hittable_list objects;

    auto red   = make_shared<lambertian>(color(.65, .05, .05));
    auto white = make_shared<lambertian>(color(.73, .73, .73));
    auto green = make_shared<lambertian>(color(.12, .45, .15));
    auto light = make_shared<diffuse_light>(color(15, 15, 15));

    objects.add(make_shared<yz_rect>(0, 555, 0, 555, 555, green));
    objects.add(make_shared<yz_rect>(0, 555, 0, 555, 0, red));
    objects.add(make_shared<xz_rect>(213, 343, 227, 332, 554, light));
    objects.add(make_shared<xz_rect>(0, 555, 0, 555, 555, white));
    objects.add(make_shared<xz_rect>(0, 555, 0, 555, 0, white));
    objects.add(make_shared<xy_rect>(0, 555, 0, 555, 555, white));

    shared_ptr<hittable> box1 = make_shared<box>(point3(0,0,0), point3(165,330,165), white);
    box1 = make_shared<rotate_y>(box1, 15);
    box1 = make_shared<translate>(box1, vec3(265,0,295));
    objects.add(box1);

    shared_ptr<hittable> box2 = make_shared<box>(point3(0,0,0), point3(165,165,165), white);
    box2 = make_shared<rotate_y>(box2, -18);
    box2 = make_shared<translate>(box2, vec3(130,0,65));
    objects.add(box2);

    return objects;
}

int main() {
    // Image

    const auto aspect_ratio = 1.0 / 1.0;
    const int image_width = 600;
    const int image_height = static_cast<int>(image_width / aspect_ratio);
    const int samples_per_pixel = 100;
    const int max_depth = 50;

    // World

    auto world = cornell_box();

    color background(0,0,0);

    // Camera

    point3 lookfrom(278, 278, -800);
    point3 lookat(278, 278, 0);
    vec3 vup(0, 1, 0);
    auto dist_to_focus = 10.0;
    auto aperture = 0.0;
    auto vfov = 40.0;
    auto time0 = 0.0;
    auto time1 = 1.0;

    camera cam(lookfrom, lookat, vup, vfov, aspect_ratio, aperture, dist_to_focus, time0, time1);

    // Render

    std::cout << "P3\n" << image_width << ' ' << image_height << "\n255\n";

    for (int j = image_height-1; j >= 0; --j) {
        ...
}
```

Listing 9: `[main.cc]` *Cornell box, refactored*

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

Image 1: Cornell box, refactored

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: $\int f(x) \approx f(r)/p(r)$. 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, color& albedo, ray& scattered, double& pdf
        ) const {
            return false;
        }

        virtual double scattering_pdf(
            const ray& r_in, const hit_record& rec, const ray& scattered
        ) const {
            return 0;
        }

        virtual color emitted(double u, double v, const point3& p) const {
            return color(0,0,0);
        }
};
```

Listing 10: `[material.h]` *The material class, adding importance sampling*

And *Lambertian* material becomes:

```
class lambertian : public material {
    public:
        lambertian(const color& a) : albedo(make_shared<solid_color>(a)) {}
        lambertian(shared_ptr<texture> a) : albedo(a) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& alb, ray& scattered, double& pdf
        ) const override {
            auto scatter_direction = rec.normal + random_unit_vector();

            // Catch degenerate scatter direction
            if (scatter_direction.near_zero())
                scatter_direction = rec.normal;
            scattered = ray(rec.p, unit_vector(direction), r_in.time());
            alb = albedo->value(rec.u, rec.v, rec.p);
            pdf = dot(rec.normal, scattered.direction()) / pi;
            return true;
        }
        double scattering_pdf(
            const ray& r_in, const hit_record& rec, const ray& scattered
        ) const {
            auto cosine = dot(rec.normal, unit_vector(scattered.direction()));
            return cosine < 0 ? 0 : cosine/pi;
        }

    public:
        shared_ptr<texture> albedo;
};
```

Listing 11: `[material.h]` *Lambertian material, modified for importance sampling*

And the `ray_color` function gets a minor modification:

```
color ray_color(const ray& r, const color& background, const hittable& world, int depth) {
    hit_record rec;

    // If we've exceeded the ray bounce limit, no more light is gathered.
    if (depth <= 0)
        return color(0,0,0);

    // If the ray hits nothing, return the background color.
    if (!world.hit(r, 0.001, infinity, rec))
        return background;

    ray scattered;
    color attenuation;
    color emitted = rec.mat_ptr->emitted(rec.u, rec.v, rec.p);
    double pdf;
    color albedo;

    if (!rec.mat_ptr->scatter(r, rec, albedo, scattered, pdf))
        return emitted;

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

Listing 12: `[main.cc]` *The ray_color function, modified for importance sampling*

You should get exactly the same picture.

6.2. Random Hemisphere Sampling

Now, just for the experience, try a different sampling strategy. As in the first book, Let's choose randomly from the hemisphere above the surface. This would be $p(direction) = \frac{1}{2\pi}$.

```
virtual bool scatter(
    const ray& r_in, const hit_record& rec, color& alb, ray& scattered, double& pdf
) const override {
    auto direction = random_in_hemisphere(rec.normal);
    scattered = ray(rec.p, unit_vector(direction), r_in.time());
    alb = albedo->value(rec.u, rec.v, rec.p);
    pdf = 0.5 / pi;
    return true;
}
```

Listing 13: `[material.h]` *Modified scatter function*

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

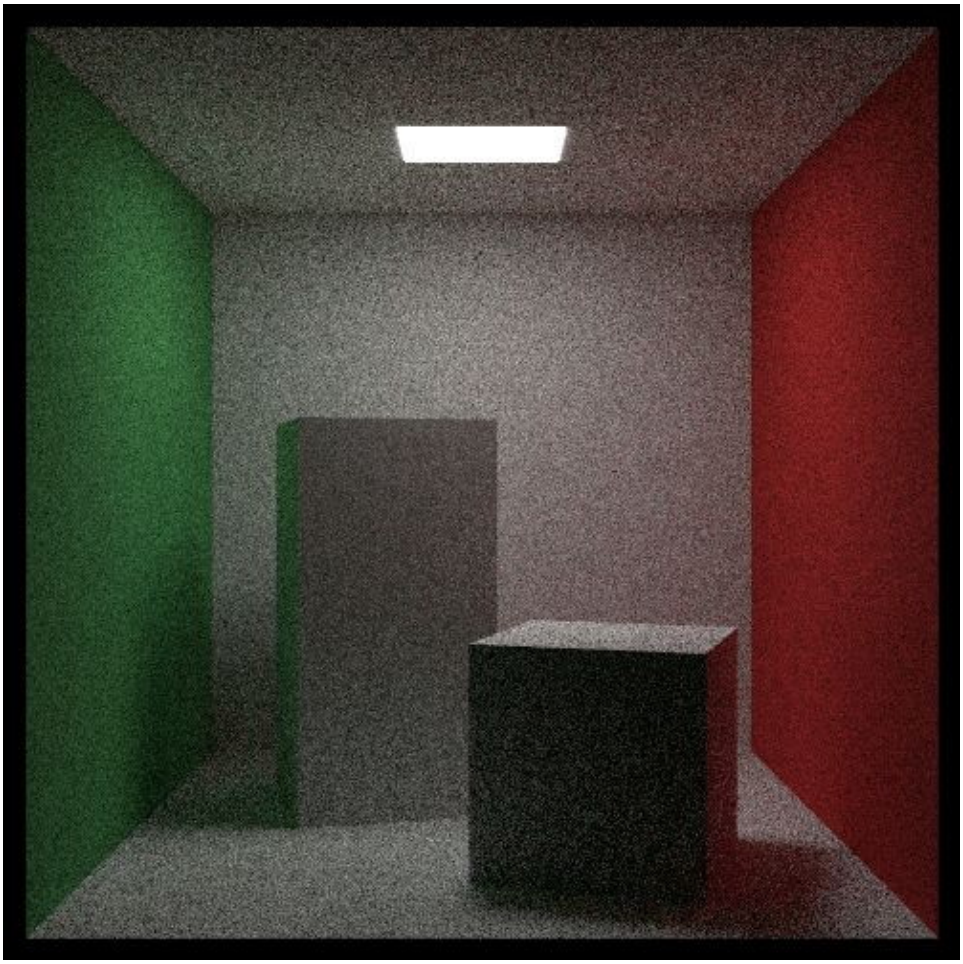

Image 2: Cornell box, with different sampling strategy

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. I also don't know if the bug is the first version of the program, or the second, or both!

Let's build some infrastructure to address this.

7. 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.

7.1. Random Directions Relative to the Z Axis

Let's first figure out how to generate random directions. To simplify things, let's assume the z-axis is the surface normal, and θ 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 $dA = \sin(\theta) \cdot d\theta \cdot d\phi$. If you haven't, you'll have to take my word for the next step, but you'll get it when you take advanced calculus.

Given a directional PDF, $p(direction) = f(\theta)$ on the sphere, the 1D PDFs on θ and ϕ are:

$$a(\phi) = \frac{1}{2\pi}$$

(uniform)

$$b(\theta) = 2\pi f(\theta) \sin(\theta)$$

For uniform random numbers r_1 and r_2, the material presented in the One Dimensional MC Integration chapter leads to:

$$r_1 = \int_0^\phi \frac{1}{2\pi} dt = \frac{\phi}{2\pi}$$

Solving for ϕ we get:

$$\phi = 2\pi \cdot r_1$$

For θ we have:

$$r_2 = \int_0^\theta 2\pi f(t) \sin(t) dt$$

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 4π, so a uniform $p(direction) = \frac{1}{4\pi}$ on the unit sphere.

$$r_2 = \int_0^\theta 2\pi \frac{1}{4\pi} \sin(t) dt$$

$$= \int_0^\theta \frac{1}{2} \sin(t) dt$$

$$= \frac{-\cos(\theta)}{2} - \frac{-\cos(0)}{2}$$

$$= \frac{1 - \cos(\theta)}{2}$$

Solving for $\cos(\theta)$ gives:

$$\cos(\theta) = 1 - 2r_2$$

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

To generate a unit vector direction toward (θ, ϕ) we convert to Cartesian coordinates:

$$x = \cos(\phi) \cdot \sin(\theta)$$

$$y = \sin(\phi) \cdot \sin(\theta)$$

$$z = \cos(\theta)$$

And using the identity that $\cos^2 + \sin^2 = 1$, we get the following in terms of random (r_1, r_2):

$$x = \cos(2\pi \cdot r_1) \sqrt{1 - (1 - 2r_2)^2}$$

$$y = \sin(2\pi \cdot r_1) \sqrt{1 - (1 - 2r_2)^2}$$

$$z = 1 - 2r_2$$

Simplifying a little, $(1 - 2r_2)^2 = 1 - 4r_2 + 4r_2^2$, so:

$$x = \cos(2\pi r_1) \cdot 2\sqrt{r_2(1 - r_2)}$$

$$y = \sin(2\pi r_1) \cdot 2\sqrt{r_2(1 - r_2)}$$

$$z = 1 - 2r_2$$

We can output some of these:

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

Listing 14: `[sphere_plot.cc]` *Random points on the unit sphere*

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

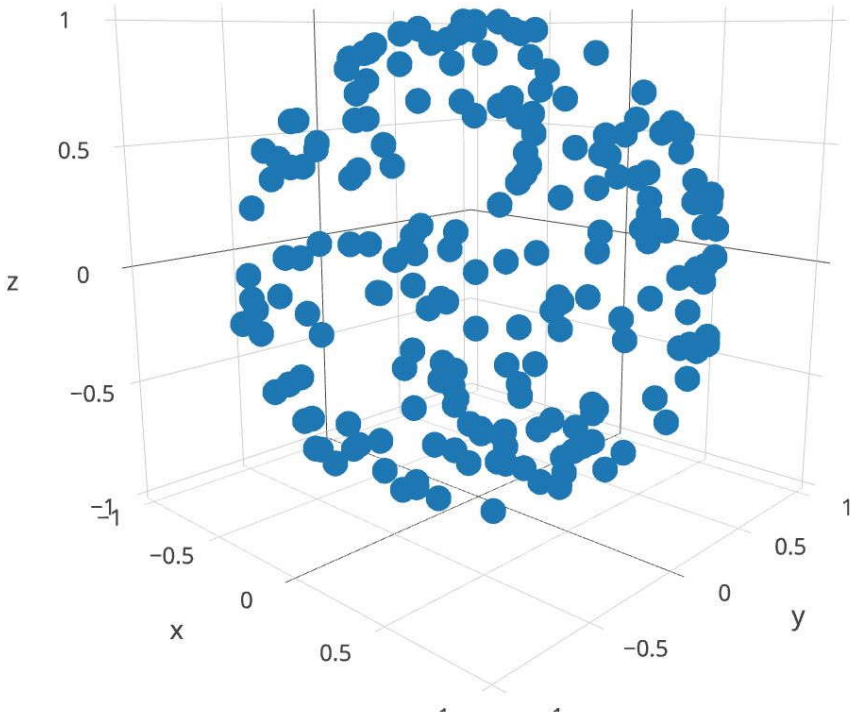

Figure 6: *Random points on the unit sphere*

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

7.2. Uniform Sampling a Hemisphere

Now let's derive uniform on the hemisphere. The density being uniform on the hemisphere means $p(direction) = \frac{1}{2\pi}$. Just changing the constant in the theta equations yields:

$$\cos(\theta) = 1 - r_2$$

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:

$$\int \cos^3(\theta) dA$$

$$= \int_0^{2\pi} \int_0^{\pi/2} \cos^3(\theta) \sin(\theta) d\theta d\phi$$

$$= 2\pi \int_0^{\pi/2} \cos^3(\theta) \sin(\theta) = \frac{\pi}{2}$$

Now for integration with importance sampling. $p(direction) = \frac{1}{2\pi}$, so we average f/p which is $\cos^3(\theta)/(1/2\pi)$, and we can test this:

```
int main() {
    int N = 1000000;
    auto sum = 0.0;
    for (int i = 0; i < N; i++) {
        auto r1 = random_double();
        auto r2 = random_double();
        auto x = cos(2*pi*r1)*2*sqrt(r2*(1-r2));
        auto y = sin(2*pi*r1)*2*sqrt(r2*(1-r2));
        auto z = 1 - r2;
        sum += z*z*z / (1.0/(2.0*pi));
    }
    std::cout << std::fixed << std::setprecision(12);
    std::cout << "Pi/2     = " << pi/2 << '\n';
    std::cout << "Estimate = " << sum/N << '\n';
}
```

Listing 15: [cos_cubed.cc] *Integration using* $cos^3(x)$

Now let's generate directions with $p(directions) = \cos(\theta)/\pi$.

$$r_2 = \int_0^\theta 2\pi \frac{\cos(t)}{\pi} \sin(t) = 1 - \cos^2(\theta)$$

So,

$$\cos(\theta) = \sqrt{1 - r_2}$$

We can save a little algebra on specific cases by noting

$$z = \cos(\theta) = \sqrt{1 - r_2}$$

$$x = \cos(\phi)\sin(\theta) = \cos(2\pi r_1)\sqrt{1 - z^2} = \cos(2\pi r_1)\sqrt{r_2}$$

$$y = \sin(\phi)\sin(\theta) = \sin(2\pi r_1)\sqrt{1 - z^2} = \sin(2\pi r_1)\sqrt{r_2}$$

Let's also start generating them as random vectors:

```
#include "rtweekend.h"

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

inline vec3 random_cosine_direction() {
    auto r1 = random_double();
    auto r2 = random_double();
    auto z = sqrt(1-r2);

    auto phi = 2*pi*r1;
    auto x = cos(phi)*sqrt(r2);
    auto y = sin(phi)*sqrt(r2);

    return vec3(x, y, z);
}

int main() {
    int N = 1000000;

    auto sum = 0.0;
    for (int i = 0; i < N; i++) {
        auto v = random_cosine_direction();
        sum += v.z()*v.z()*v.z() / (v.z()/pi);
    }

    std::cout << std::fixed << std::setprecision(12);
    std::cout << "Pi/2     = " << pi/2 << '\n';
    std::cout << "Estimate = " << sum/N << '\n';
}
```

Listing 16: `[cos_density.cc]` *Integration with cosine density function*

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

8. Orthonormal Bases

In the last chapter we developed methods to generate random directions relative to the Z-axis. We'd like to be able to do that relative to a surface normal vector.

8.1. Relative Coordinates

An orthonormal 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 $\mathbf{O}$ and cartesian unit vectors $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$. When we say a location is (3,-2,7), we really are saying:

$$\text{Location is } \mathbf{O} + 3\mathbf{x} - 2\mathbf{y} + 7\mathbf{z}$$

If we want to measure coordinates in another coordinate system with origin $\mathbf{O}'$ and basis vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$, we can just find the numbers (u, v, w) such that:

$$\text{Location is } \mathbf{O}' + u\mathbf{u} + v\mathbf{v} + w\mathbf{w}$$

8.2. Generating an Orthonormal Basis

If you take an intro graphics course, there will be a lot of time spent on coordinate systems and 4×4 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 $\mathbf{n}$. 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 $\mathbf{n}$ and to each other.

Some models will come with one or more cotangent vectors. If our model has only one cotangent vector, then the process of making an ONB is a nontrivial one. Suppose we have any vector $\mathbf{a}$ that is of nonzero length and not parallel to $\mathbf{n}$. We can get vectors $\mathbf{s}$ and $\mathbf{t}$ perpendicular to $\mathbf{n}$ by using the property of the cross product that $\mathbf{c} \times \mathbf{d}$ is perpendicular to both $\mathbf{c}$ and $\mathbf{d}$:

$$\mathbf{t} = \text{unit\_vector}(\mathbf{a} \times \mathbf{n})$$

$$\mathbf{s} = \mathbf{t} \times \mathbf{n}$$

This is all well and good, but the catch is that we may not be given an $\mathbf{a}$ when we load a model, and we don't have an $\mathbf{a}$ with our existing program. If we went ahead and picked an arbitrary $\mathbf{a}$ to use as our initial vector we may get an $\mathbf{a}$ that is parallel to $\mathbf{n}$. A common method is to use an if-statement to determine whether $\mathbf{n}$ is a particular axis, and if not, use that axis.

```
if absolute(n.x > 0.9)
    a ← (0, 1, 0)
else
    a ← (1, 0, 0)
```

Once we have an ONB of $\mathbf{s}$, $\mathbf{t}$, and $\mathbf{n}$, and we have a $random(x, y, z)$ relative to the Z-axis, we can get the vector relative to $\mathbf{n}$ as:

$$\text{Random vector} = x\mathbf{s} + y\mathbf{t} + z\mathbf{n}$$

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.

8.3. The ONB Class

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:

```
#ifndef ONB_H
#define ONB_H

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(double a, double b, double 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&);

    public:
        vec3 axis[3];
};


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

#endif
```

Listing 17: `[onb.h]` *Orthonormal basis class*

We can rewrite our Lambertian material using this to get:

```
virtual bool scatter(
    const ray& r_in, const hit_record& rec, color& alb, ray& scattered, double& pdf
) const override {
    onb uvw;
    uvw.build_from_w(rec.normal);
    auto 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()) / pi;
    return true;
}
```

Listing 18: `[material.h]` *Scatter function, with orthonormal basis*

Which produces:

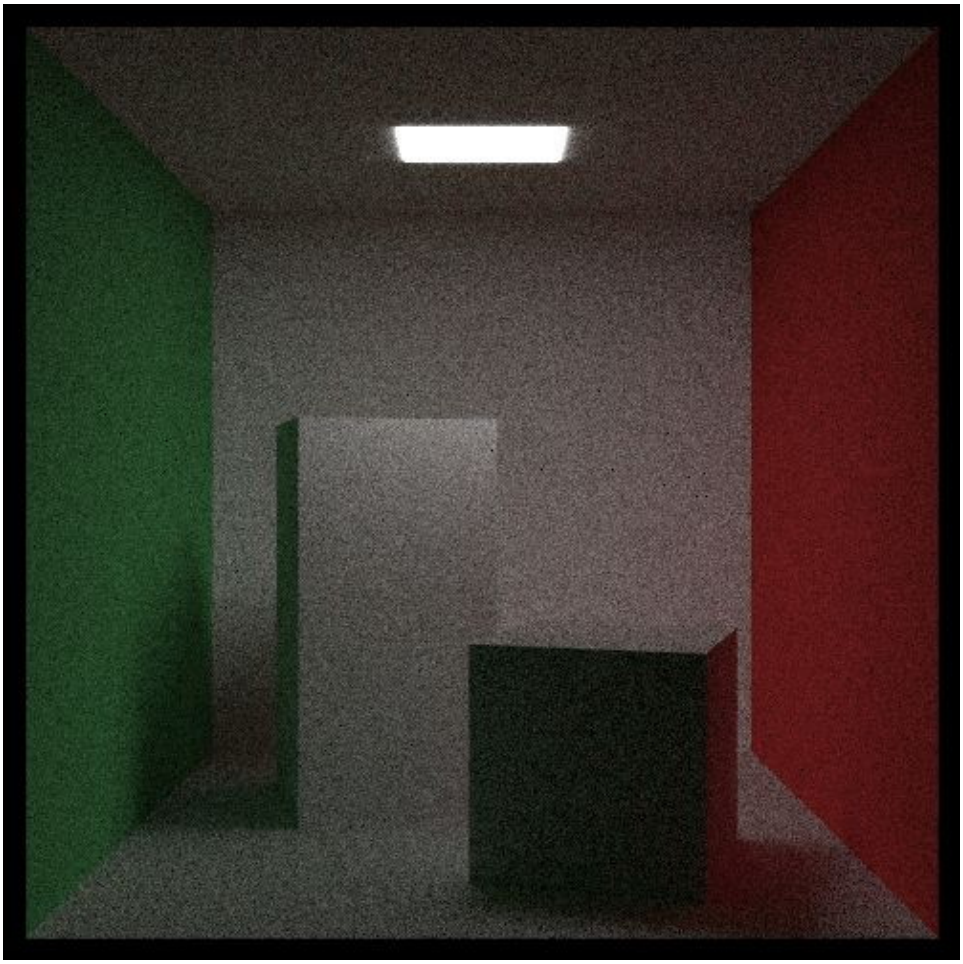

Image 3: Cornell box, with orthonormal basis scatter function

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 get rid of some of that noise.

9. 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. 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. We also need to know the PDF, $p(direction)$. What is that?

9.1. Getting the PDF of a Light

For a light of area A, if we sample uniformly on that light, the PDF on the surface of the light is $\frac{1}{A}$. What is it on the area of the unit sphere that defines directions? Fortunately, there is a simple correspondence, as outlined in the diagram:

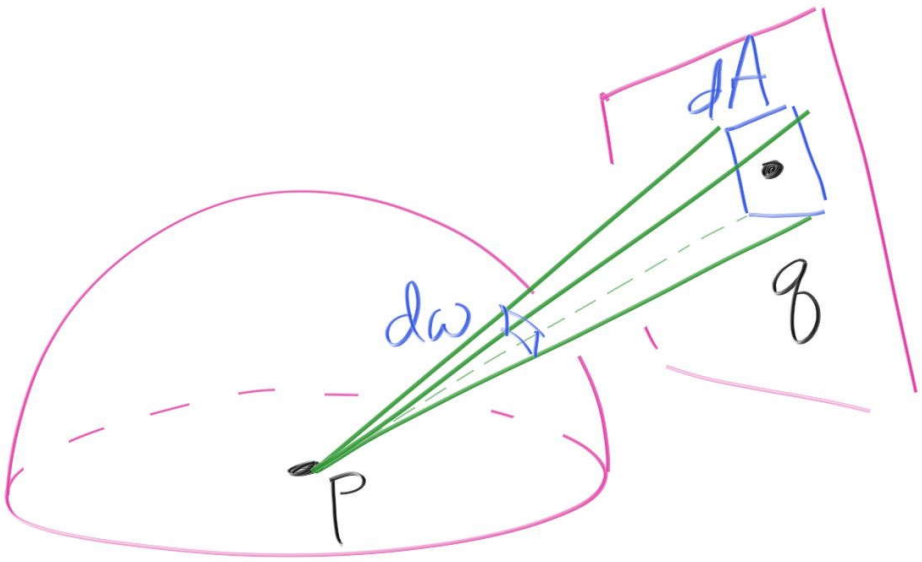

Figure 7: *Projection of light shape onto PDF*

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

$$dw = \frac{dA \cdot \cos(alpha)}{distance^2(p,q)}$$

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

$$p(direction) \cdot \frac{dA \cdot \cos(alpha)}{distance^2(p,q)} = p_q(q) \cdot dA = \frac{dA}{A}$$

So

$$p(direction) = \frac{distance^2(p,q)}{\cos(alpha) \cdot A}$$

9.2. Light Sampling

If we hack our `ray_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):

```
color ray_color(const ray& r, const color& background, const hittable& world, int depth) {
    hit_record rec;

    // If we've exceeded the ray bounce limit, no more light is gathered.
    if (depth <= 0)
        return color(0,0,0);

    // If the ray hits nothing, return the background color.
    if (!world.hit(r, 0.001, infinity, rec))
        return background;

    ray scattered;
    color attenuation;
    color emitted = rec.mat_ptr->emitted(rec.u, rec.v, rec.p);
    double pdf;
    color albedo;
    if (!rec.mat_ptr->scatter(r, rec, albedo, scattered, pdf))
        return emitted;
    auto on_light = point3(random_double(213,343), 554, random_double(227,332));
    auto to_light = on_light - rec.p;
    auto distance_squared = to_light.length_squared();
    to_light = unit_vector(to_light);

    if (dot(to_light, rec.normal) < 0)
        return emitted;

    double light_area = (343-213)*(332-227);
    auto 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)
                  * ray_color(scattered, background, world, depth-1) / pdf;
}
```

Listing 19: `[main.cc]` *Ray color with light sampling*

With 10 samples per pixel this yields:

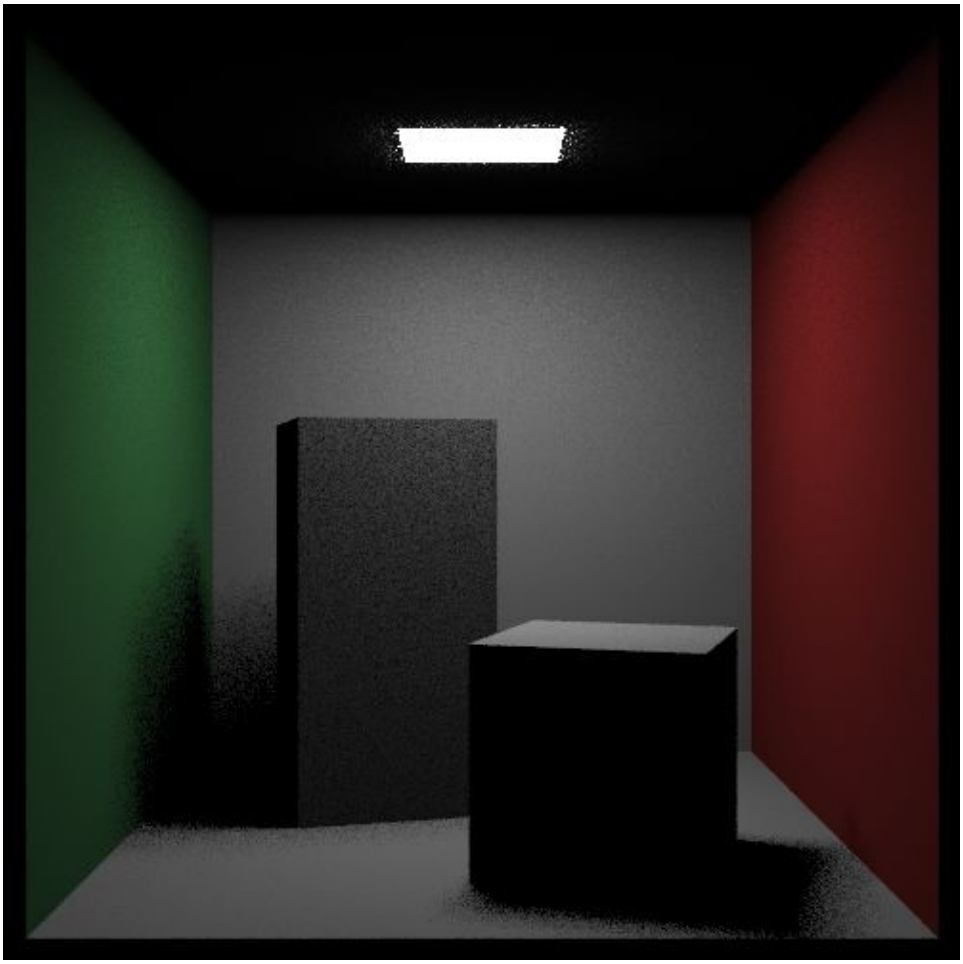

Image 4: Cornell box, sampling only the light, 10 samples per pixel

This is about what we would expect from something that samples only the light sources, so this appears to work.

9.3. Switching to Unidirectional Light

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 hittable take extra information:

```
virtual color emitted(const ray& r_in, const hit_record& rec, double u, double v,
    const point3& p) const override {

    if (rec.front_face)
        return emit->value(u, v, p);
    else
        return color(0,0,0);
}
```

Listing 20: `[material.h]` *Material emission, directional*

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

```
class flip_face : public hittable {
    public:
        flip_face(shared_ptr<hittable> p) : ptr(p) {}

        virtual bool hit(
            const ray& r, double t_min, double t_max, hit_record& rec) const override {

            if (!ptr->hit(r, t_min, t_max, rec))
                return false;

            rec.front_face = !rec.front_face;
            return true;
        }

        virtual bool bounding_box(double time0, double time1, aabb& output_box) const override {
            return ptr->bounding_box(time0, time1, output_box);
        }

    public:
        shared_ptr<hittable> ptr;
};
```

Listing 21: `[hittable.h]` *We use a hittable object to flip the light*

Making sure to call this in our world definition:

```
hittable_list cornell_box() {
    hittable_list objects;

    auto red   = make_shared<lambertian>(color(.65, .05, .05));
    auto white = make_shared<lambertian>(color(.73, .73, .73));
    auto green = make_shared<lambertian>(color(.12, .45, .15));
    auto light = make_shared<diffuse_light>(color(15, 15, 15));

    objects.add(make_shared<yz_rect>(0, 555, 0, 555, 555, green));
    objects.add(make_shared<yz_rect>(0, 555, 0, 555, 0, red));
    objects.add(make_shared<flip_face>(make_shared<xz_rect>(213, 343, 227, 332, 554, light)));
    objects.add(make_shared<xz_rect>(0, 555, 0, 555, 555, white));
    objects.add(make_shared<xz_rect>(0, 555, 0, 555, 0, white));
    objects.add(make_shared<xy_rect>(0, 555, 0, 555, 555, white));

    ...
```

Listing 22: `[main.cc]` *Flip the light in our cornell box scene*

This gives us:

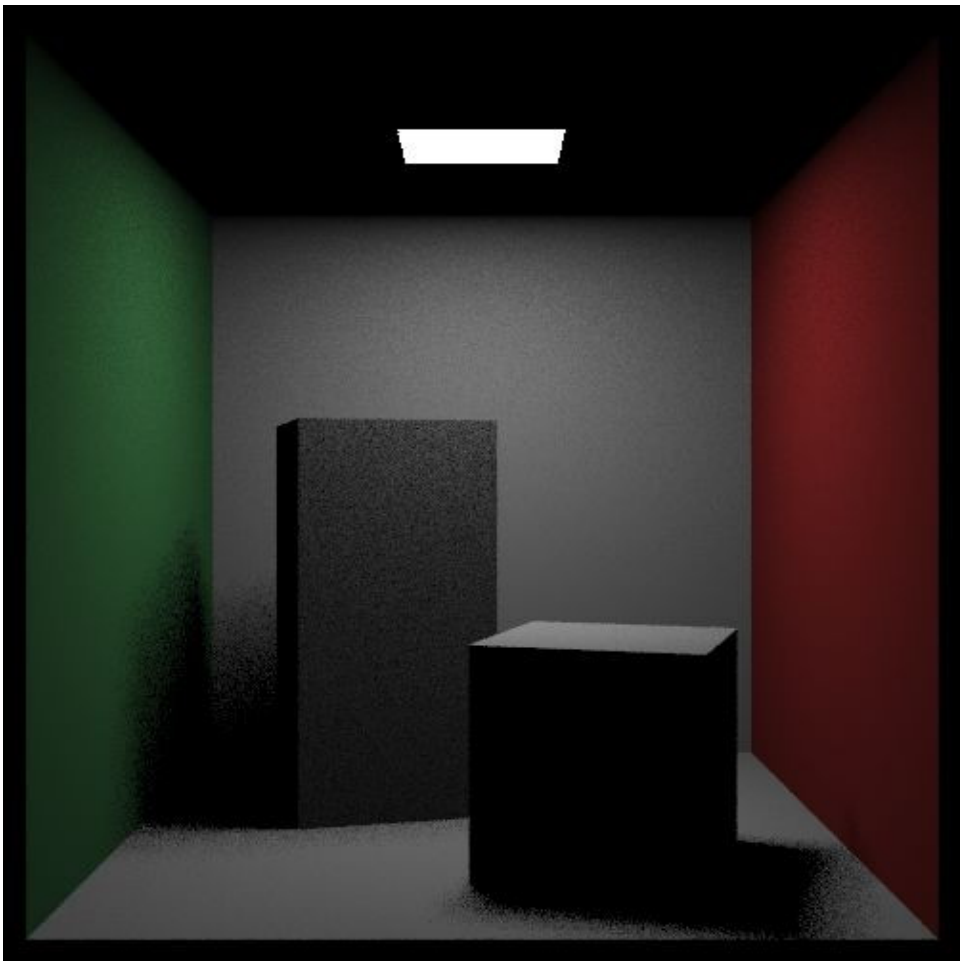

Image 5: Cornell box, light emitted only in the downward direction

10. Mixture Densities

We have used a PDF related to $\cos(\theta)$, and a PDF related to sampling the light. We would like a PDF that combines these.

10.1. An Average of Lighting and Reflection

A common tool in probability is to mix the densities to form a mixture density. Any weighted average of PDFs is a PDF. For example, we could just average the two densities:

$$\text{mixture}_{\text{pdf}}(\textit{direction}) = \frac{1}{2}\text{reflection}_{\text{pdf}}(\textit{direction}) + \frac{1}{2}\text{light}_{\text{pdf}}(\textit{direction})$$

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 (random_double() < 0.5)
    pick direction according to pdf_reflection
else
    pick direction according to pdf_light
```

But evaluating $\text{mixture}_{\text{pdf}}$ is slightly more subtle. We need to evaluate both $\text{reflection}_{\text{pdf}}$ and $\text{light}_{\text{pdf}}$ because there are some directions where either PDF could have generated the direction. For example, we might generate a direction toward the light using $\text{reflection}_{\text{pdf}}$.

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 $\text{reflection}_{\text{pdf}}$ and the $\text{light}_{\text{pdf}}$ 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:

```
#ifndef PDF_H
#define PDF_H

class pdf {
    public:
        virtual ~pdf() {}

        virtual double value(const vec3& direction) const = 0;
        virtual vec3 generate() const = 0;
};

#endif
```

Listing 23: [pdf.h] *The `pdf` class*

We'll see if that works by fleshing out the subclasses. For sampling the light, we will need `hittable` 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 `hittable` involving sampling the bounding box that works with all its subclasses.

First, let's try a cosine density:

```
inline vec3 random_cosine_direction() {
    auto r1 = random_double();
    auto r2 = random_double();
    auto z = sqrt(1-r2);

    auto phi = 2*pi*r1;
    auto x = cos(phi)*sqrt(r2);
    auto y = sin(phi)*sqrt(r2);

    return vec3(x, y, z);
}

class cosine_pdf : public pdf {
    public:
        cosine_pdf(const vec3& w) { uvw.build_from_w(w); }

        virtual double value(const vec3& direction) const override {
            auto cosine = dot(unit_vector(direction), uvw.w());
            return (cosine <= 0) ? 0 : cosine/pi;
        }

        virtual vec3 generate() const override {
            return uvw.local(random_cosine_direction());
        }

    public:
        onb uvw;
};
```

Listing 24: `[pdf.h]` *The cosine_pdf class*

We can try this in the `ray_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.

```
color ray_color(const ray& r, const color& background, const hittable& world, int depth) {
    hit_record rec;

    // If we've exceeded the ray bounce limit, no more light is gathered.
    if (depth <= 0)
        return color(0,0,0);

    // If the ray hits nothing, return the background color.
    if (!world.hit(r, 0.001, infinity, rec))
        return background;

    ray scattered;
    color attenuation;
    color emitted = rec.mat_ptr->emitted(r, rec, rec.u, rec.v, rec.p);
    double pdf_val;
    color albedo;
    if (!rec.mat_ptr->scatter(r, rec, albedo, scattered, pdf_val))
        return emitted;
    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)
                  * ray_color(scattered, background, world, depth-1) / pdf_val;
}
```

Listing 25: `[main.cc]` *The ray_color function, using cosine pdf*

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

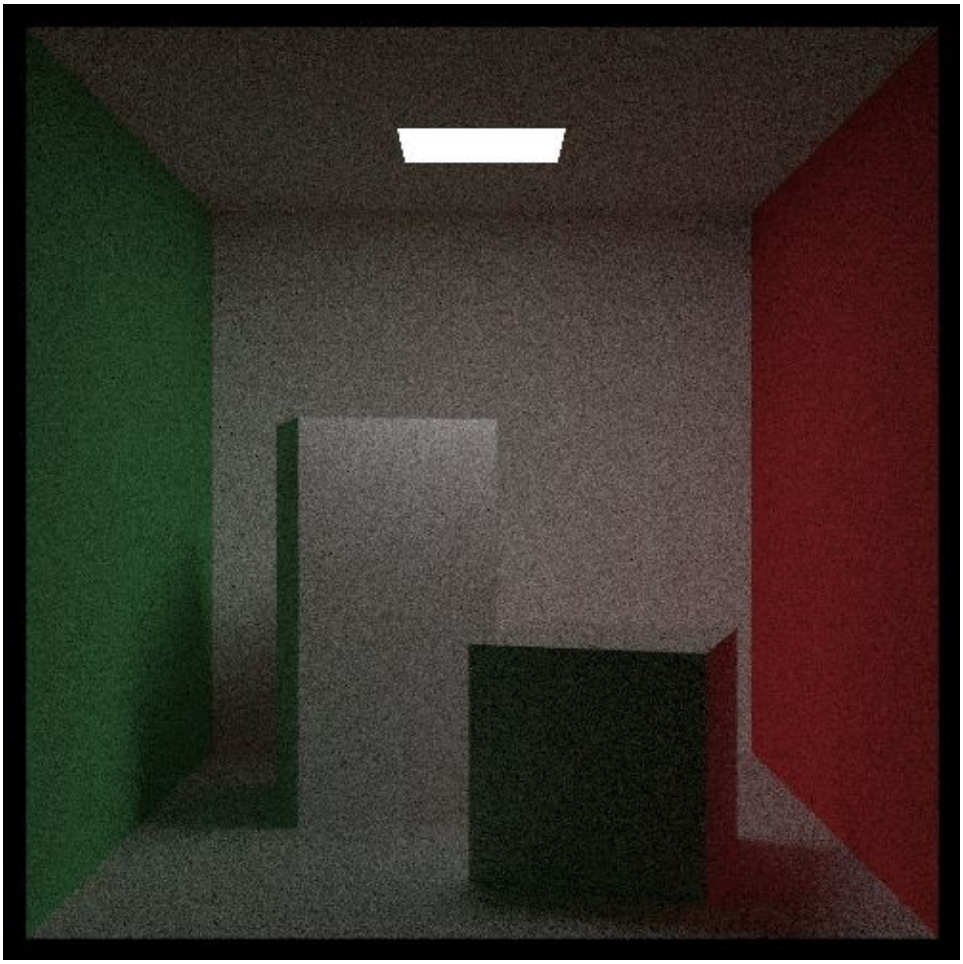

Image 6: Cornell box with a cosine density pdf

10.2. Sampling Directions towards a Hittable

Now we can try sampling directions toward a `hittable`, like the light.

```
class hittable_pdf : public pdf {
    public:
        hittable_pdf(shared_ptr<hittable> p, const point3& origin) : ptr(p), o(origin) {}

        virtual double value(const vec3& direction) const override {
            return ptr->pdf_value(o, direction);
        }

        virtual vec3 generate() const override {
            return ptr->random(o);
        }

    public:
        point3 o;
        shared_ptr<hittable> ptr;
};
```

Listing 26: `[pdf.h]` *The hittable_pdf class*

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

```
class hittable {
    public:
        virtual bool hit(const ray& r, double t_min, double t_max, hit_record& rec) const = 0;
        virtual bool bounding_box(double time0, double time1, aabb& output_box) const = 0;
        virtual double pdf_value(const point3& o, const vec3& v) const {
            return 0.0;
        }

        virtual vec3 random(const vec3& o) const {
            return vec3(1, 0, 0);
        }
};
```

Listing 27: `[hittable.h]` *The hittable class, with two new methods*

And we change `xz_rect` to implement those functions:

```
class xz_rect : public hittable {
    public:
        ...
        virtual double pdf_value(const point3& origin, const vec3& v) const override {
            hit_record rec;
            if (!this->hit(ray(origin, v), 0.001, infinity, rec))
                return 0;

            auto area = (x1-x0)*(z1-z0);
            auto distance_squared = rec.t * rec.t * v.length_squared();
            auto cosine = fabs(dot(v, rec.normal) / v.length());

            return distance_squared / (cosine * area);
        }

        virtual vec3 random(const point3& origin) const override {
            auto random_point = point3(random_double(x0,x1), k, random_double(z0,z1));
            return random_point - origin;
        }
```

Listing 28: `[aarect.h]` *XZ rect with pdf*

And then change `ray_color()`:

```
color ray_color(
    const ray& r, const color& background, const hittable& world,
    shared_ptr<hittable>& lights, int depth
) {
    ...

    ray scattered;
    color attenuation;
    color emitted = rec.mat_ptr->emitted(r, rec, rec.u, rec.v, rec.p);
    double pdf_val;
    color albedo;
    if (!rec.mat_ptr->scatter(r, rec, albedo, scattered, pdf_val))
        return emitted;

    hittable_pdf light_pdf(lights, rec.p);
    scattered = ray(rec.p, light_pdf.generate(), r.time());
    pdf_val = light_pdf.value(scattered.direction());

    return emitted
         + albedo * rec.mat_ptr->scattering_pdf(r, rec, scattered)
                  * ray_color(scattered, background, world, lights, depth-1) / pdf_val;
}

...
int main() {
    ...
    // World

    auto world = cornell_box();
    shared_ptr<hittable> lights =
        make_shared<xz_rect>(213, 343, 227, 332, 554, shared_ptr<material>());

    ...
    for (int j = image_height-1; j >= 0; --j) {
        std::cerr << "\rScanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            ...
            pixel_color += ray_color(r, background, world, lights, max_depth);
    ...
```

Listing 29: `[main.cc]` *ray_color function with hittable PDF*

At 10 samples per pixel we get:

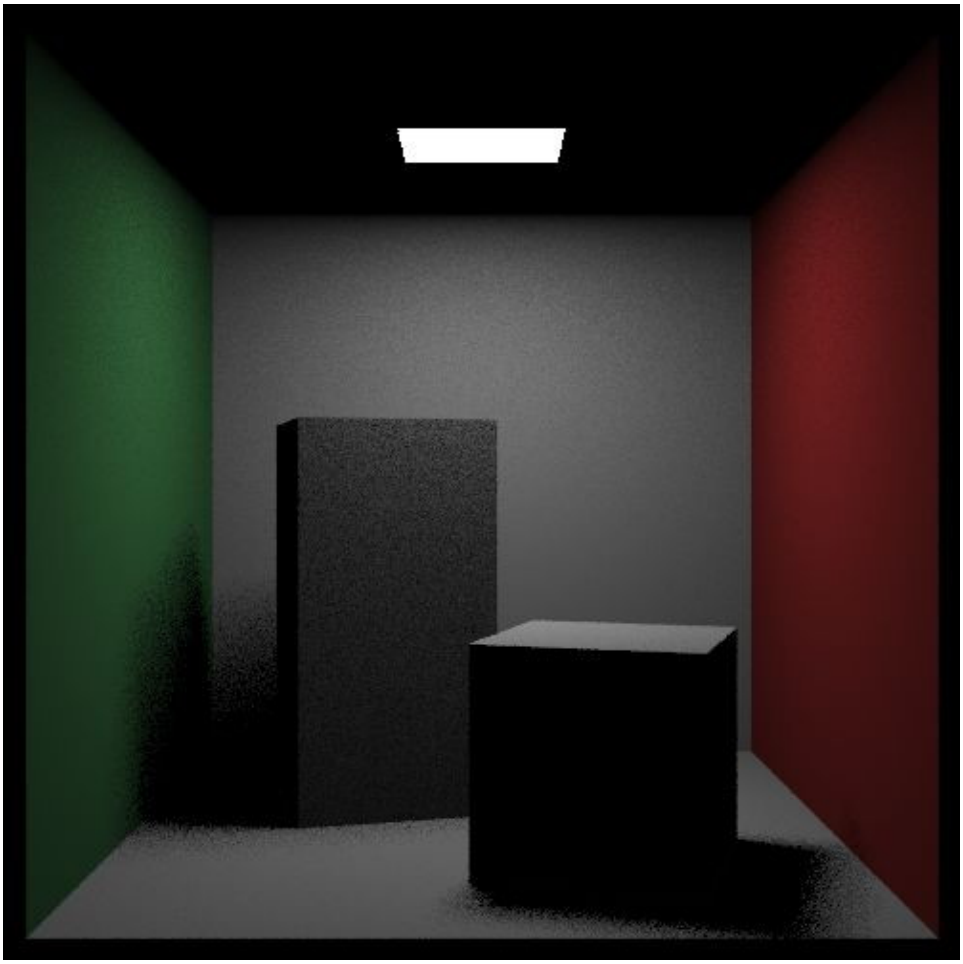

Image 7: Cornell box, sampling a hittable light, 10 samples per pixel

10.3. The Mixture PDF Class

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(shared_ptr<pdf> p0, shared_ptr<pdf> p1) {
            p[0] = p0;
            p[1] = p1;
        }

        virtual double value(const vec3& direction) const override {
            return 0.5 * p[0]->value(direction) + 0.5 *p[1]->value(direction);
        }

        virtual vec3 generate() const override {
            if (random_double() < 0.5)
                return p[0]->generate();
            else
                return p[1]->generate();
        }

    public:
        shared_ptr<pdf> p[2];
};
```

Listing 30: [pdf.h] *The `mixture_pdf` class*

And plugging it into `ray_color()`:

```
color ray_color(
    const ray& r, const color& background, const hittable& world,
    shared_ptr<hittable>& lights, int depth
) {
    ...

    ray scattered;
    color attenuation;
    color emitted = rec.mat_ptr->emitted(r, rec, rec.u, rec.v, rec.p);
    double pdf_val;
    color albedo;
    if (!rec.mat_ptr->scatter(r, rec, albedo, scattered, pdf_val))
        return emitted;
    auto p0 = make_shared<hittable_pdf>(lights, rec.p);
    auto p1 = make_shared<cosine_pdf>(rec.normal);
    mixture_pdf mixed_pdf(p0, p1);

    scattered = ray(rec.p, mixed_pdf.generate(), r.time());
    pdf_val = mixed_pdf.value(scattered.direction());

    ...
}
```

Listing 31: [main.cc] *The ray_color function, using mixture PDF*

1000 samples per pixel yields:

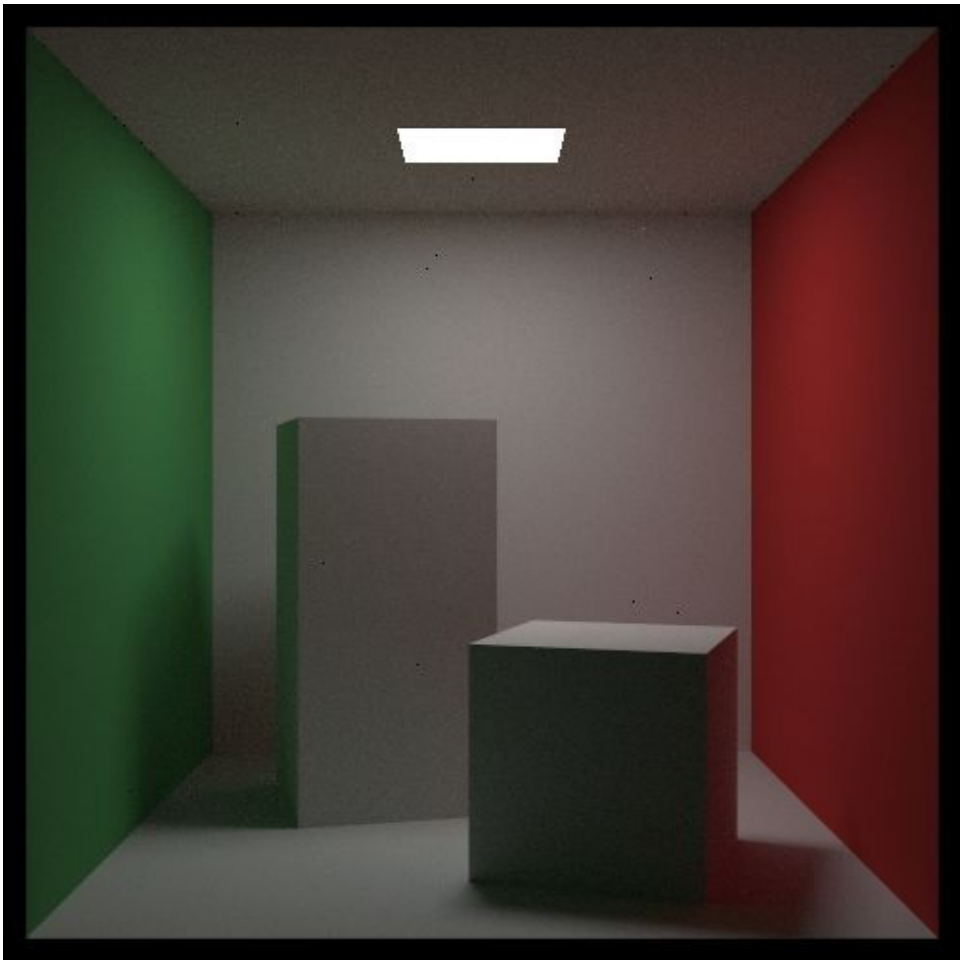

Image 8: Cornell box, mixture density of cosine and light sampling

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. Yet 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.

11. 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 `ray_color()` function. We should clean that up, probably by passing 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 RGB components are floats rather than 0–255 bytes usually interpreted as 0-1). Our output has been HDR all along; 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.

12. Cleaning Up PDF Management

So far I have the `ray_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 `hittable` we want to sample) into the `ray_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.

12.1. Diffuse Versus Specular

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 `ray_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 `hittable`:

```
struct scatter_record {
    ray specular_ray;
    bool is_specular;
    color attenuation;
    shared_ptr<pdf> pdf_ptr;
};

class material {
    public:
        virtual color emitted(
            const ray& r_in, const hit_record& rec, double u, double v, const point3& p
        ) const {
            return color(0,0,0);
        }

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, scatter_record& srec
        ) const {
            return false;
        }

        virtual double scattering_pdf(
            const ray& r_in, const hit_record& rec, const ray& scattered
        ) const {
            return 0;
        }
};
```

Listing 32: `[material.h]` *Refactoring the material class*

The Lambertian material becomes simpler:

```
class lambertian : public material {
    public:
        lambertian(const color& a) : albedo(make_shared<solid_color>(a)) {}
        lambertian(shared_ptr<texture> a) : albedo(a) {}
        virtual bool scatter(
            const ray& r_in, const hit_record& rec, scatter_record& srec
        ) const override {
            srec.is_specular = false;
            srec.attenuation = albedo->value(rec.u, rec.v, rec.p);
            srec.pdf_ptr = new cosine_pdf(rec.normal);
            return true;
        }

        double scattering_pdf(
            const ray& r_in, const hit_record& rec, const ray& scattered
        ) const {
            auto cosine = dot(rec.normal, unit_vector(scattered.direction()));
            return cosine < 0 ? 0 : cosine/pi;
        }

    public:
        shared_ptr<texture> albedo;
};
```

Listing 33: `[material.h]` *New lambertian scatter() method*

And `ray_color()` changes are small:

```
color ray_color(
    const ray& r,
    const color& background,
    const hittable& world,
    shared_ptr<hittable>& lights,
    int depth
) {
    hit_record rec;

    // If we've exceeded the ray bounce limit, no more light is gathered.
    if (depth <= 0)
        return color(0,0,0);

    // If the ray hits nothing, return the background color.
    if (!world.hit(r, 0.001, infinity, rec))
        return background;
    scatter_record srec;
    color emitted = rec.mat_ptr->emitted(r, rec, rec.u, rec.v, rec.p);
    if (!rec.mat_ptr->scatter(r, rec, srec))
        return emitted;

    auto light_ptr = make_shared<hittable_pdf>(lights, rec.p);
    mixture_pdf p(light_ptr, srec.pdf_ptr);

    ray scattered = ray(rec.p, p.generate(), r.time());
    auto pdf_val = p.value(scattered.direction());

    return emitted
        + srec.attenuation * rec.mat_ptr->scattering_pdf(r, rec, scattered)
                           * ray_color(scattered, background, world, lights, depth-1) / pdf_val;
}

...

int main() {
    ...
    // World

    auto world = cornell_box();
    auto lights = make_shared<hittable_list>();
    lights->add(make_shared<xz_rect>(213, 343, 227, 332, 554, shared_ptr<material>()));
    lights->add(make_shared<sphere>(point3(190, 90, 190), 90, shared_ptr<material>()));
    ...
```

Listing 34: `[main.cc]` *Ray color with scatter*

12.2. Handling Specular

We have not yet dealt with specular surfaces, nor instances that mess with the surface normal. But this design is clean overall, and those are all fixable. For now, I will just fix `specular`. Metal and dielectric materials are easy to fix.

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

...

class dielectric : public material {
    public:
        ...
        virtual bool scatter(
            const ray& r_in, const hit_record& rec, scatter_record& srec
        ) const override {
            srec.is_specular = true;
            srec.pdf_ptr = nullptr;
            srec.attenuation = color(1.0, 1.0, 1.0);
            double refraction_ratio = rec.front_face ? (1.0/ir) : ir;
            ...
            srec.specular_ray = ray(rec.p, direction, r_in.time());
            return true;
        }
    ...
};
```

Listing 35: `[material.h]` *The metal and dielectric scatter methods*

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

`ray_color()` just needs a new case to generate an implicitly sampled ray:

```
color ray_color(
    const ray& r,
    const color& background,
    const hittable& world,
    shared_ptr<hittable>& lights,
    int depth
) {
    ...

    scatter_record srec;
    color emitted = rec.mat_ptr->emitted(r, rec, rec.u, rec.v, rec.p);
    if (!rec.mat_ptr->scatter(r, rec, srec))
        return emitted;

    if (srec.is_specular) {
        return srec.attenuation
             * ray_color(srec.specular_ray, background, world, lights, depth-1);
    }

    ...
}
```

Listing 36: `[main.cc]` *Ray color function with implicitly-sampled rays*

We also need to change the block to metal. We'll also swap out the short block for a glass sphere.

```
hittable_list cornell_box() {
    hittable_list objects;

    auto red   = make_shared<lambertian>(color(.65, .05, .05));
    auto white = make_shared<lambertian>(color(.73, .73, .73));
    auto green = make_shared<lambertian>(color(.12, .45, .15));
    auto light = make_shared<diffuse_light>(color(15, 15, 15));

    objects.add(make_shared<yz_rect>(0, 555, 0, 555, 555, green));
    objects.add(make_shared<yz_rect>(0, 555, 0, 555, 0, red));
    objects.add(make_shared<flip_face>(make_shared<xz_rect>(213, 343, 227, 332, 554, light)));
    objects.add(make_shared<xz_rect>(0, 555, 0, 555, 555, white));
    objects.add(make_shared<xz_rect>(0, 555, 0, 555, 0, white));
    objects.add(make_shared<xy_rect>(0, 555, 0, 555, 555, white));

    shared_ptr<material> aluminum = make_shared<metal>(color(0.8, 0.85, 0.88), 0.0);
    shared_ptr<hittable> box1 = make_shared<box>(point3(0,0,0), point3(165,330,165), aluminum);
    box1 = make_shared<rotate_y>(box1, 15);
    box1 = make_shared<translate>(box1, vec3(265,0,295));
    objects.add(box1);

    auto glass = make_shared<dielectric>(1.5);
    objects.add(make_shared<sphere>(point3(190,90,190), 90 , glass));

    return objects;
}
```

Listing 37: `[main.cc]` *Cornell box scene with aluminum material*

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

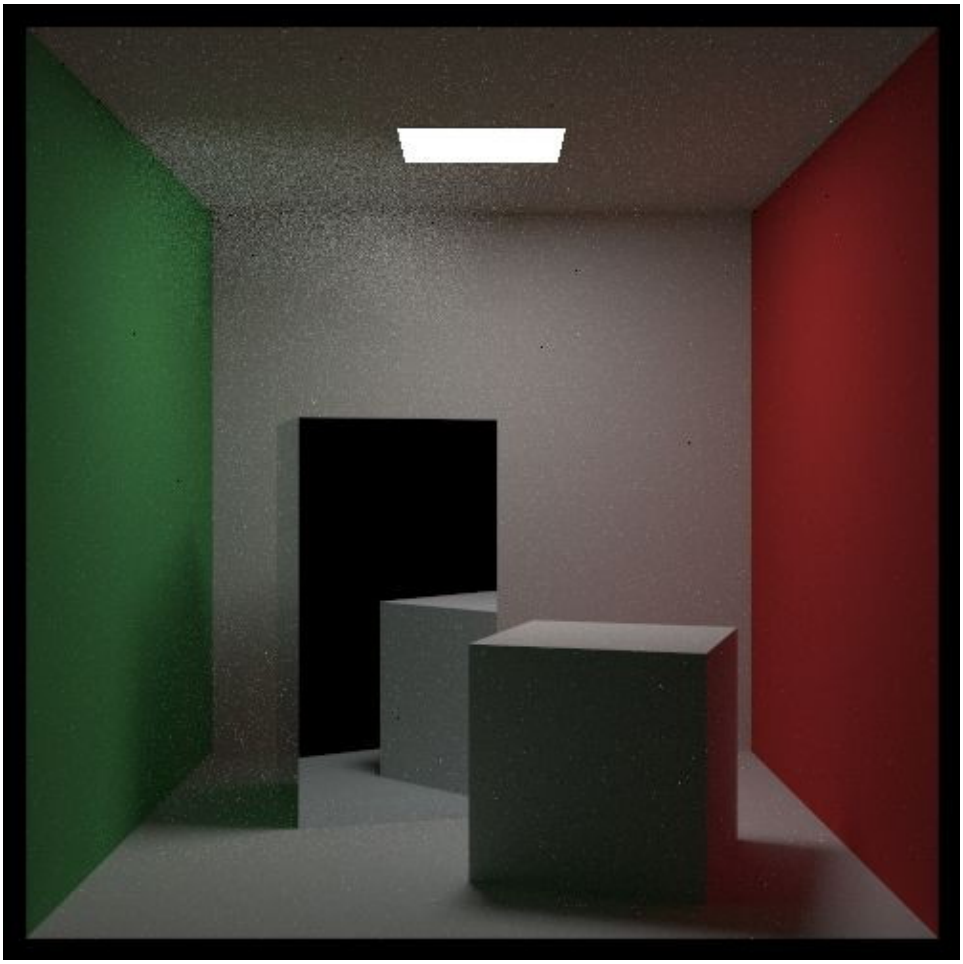

Image 9: Cornell box with arbitrary PDF functions

We could make the PDF include the block. Let's do that instead with a glass sphere because it's easier.

12.3. Sampling a Sphere Object

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 the Generating Random Directions chapter that to sample θ we have:

$$r_2 = \int_0^\theta 2\pi \cdot f(t) \cdot \sin(t) dt$$

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

$$r_2 = \int_0^\theta 2\pi \cdot C \cdot \sin(t) dt$$

Doing some algebra/calculus this yields:

$$r_2 = 2\pi \cdot C \cdot (1 - \cos(\theta))$$

So

$$cos(\theta) = 1 - \frac{r_2}{2\pi \cdot C}$$

We know that for $r_2 = 1$ we should get θ_{max}, so we can solve for C:

$$\cos(\theta) = 1 + r_2 \cdot (\cos(\theta_{max}) - 1)$$

ϕ we sample like before, so:

$$z = \cos(\theta) = 1 + r_2 \cdot (\cos(\theta_{max}) - 1)$$

$$x = \cos(\phi) \cdot \sin(\theta) = \cos(2\pi \cdot r_1) \cdot \sqrt{1 - z^2}$$

$$y = \sin(\phi) \cdot \sin(\theta) = \sin(2\pi \cdot r_1) \cdot \sqrt{1 - z^2}$$

Now what is θ_{max}?

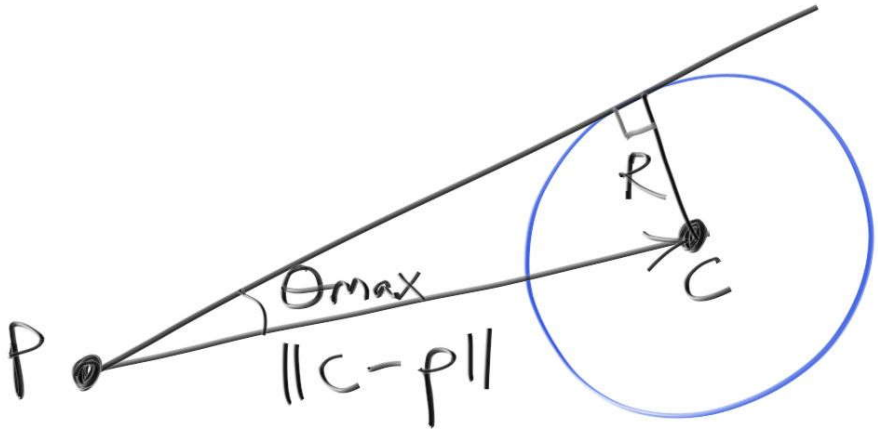

Figure 8: *A sphere-enclosing cone*

We can see from the figure that $\sin(\theta_{max}) = R/length(\mathbf{c} - \mathbf{p})$. So:

$$\cos(\theta_{max}) = \sqrt{1 - \frac{R^2}{length^2(\mathbf{c} - \mathbf{p})}}$$

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

$$solid\_angle = \int_0^{2\pi} \int_0^{\theta_{max}} \sin(\theta) = 2\pi \cdot (1 - \cos(\theta_{max}))$$

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 $\mathbf{p}$, $\cos(\theta_{max}) = 0$, and 2π is the area of a hemisphere, so that works too.

12.4. Updating the Sphere Code

The sphere class needs the two PDF-related functions:

```
double sphere::pdf_value(const point3& o, const vec3& v) const {
    hit_record rec;
    if (!this->hit(ray(o, v), 0.001, infinity, rec))
        return 0;

    auto cos_theta_max = sqrt(1 - radius*radius/(center-o).length_squared());
    auto solid_angle = 2*pi*(1-cos_theta_max);

    return  1 / solid_angle;
}

vec3 sphere::random(const point3& o) const {
    vec3 direction = center - o;
    auto distance_squared = direction.length_squared();
    onb uvw;
    uvw.build_from_w(direction);
    return uvw.local(random_to_sphere(radius, distance_squared));
}
```

Listing 38: `[sphere.h]` *Sphere with PDF*

With the utility function:

```
inline vec3 random_to_sphere(double radius, double distance_squared) {
    auto r1 = random_double();
    auto r2 = random_double();
    auto z = 1 + r2*(sqrt(1-radius*radius/distance_squared) - 1);

    auto phi = 2*pi*r1;
    auto x = cos(phi)*sqrt(1-z*z);
    auto y = sin(phi)*sqrt(1-z*z);

    return vec3(x, y, z);
}
```

Listing 39: `[pdf.h]` *The random_to_sphere utility function*

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

```
int main() {
    ...
    // World

    auto world = cornell_box();
    shared_ptr<hittable> lights =
    //  make_shared<xz_rect>(213, 343, 227, 332, 554, shared_ptr<material>());
        make_shared<sphere>(point3(190, 90, 190), 90, shared_ptr<material>());
    ...
```

Listing 40: `[main.cc]` *Sampling just the sphere*

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!

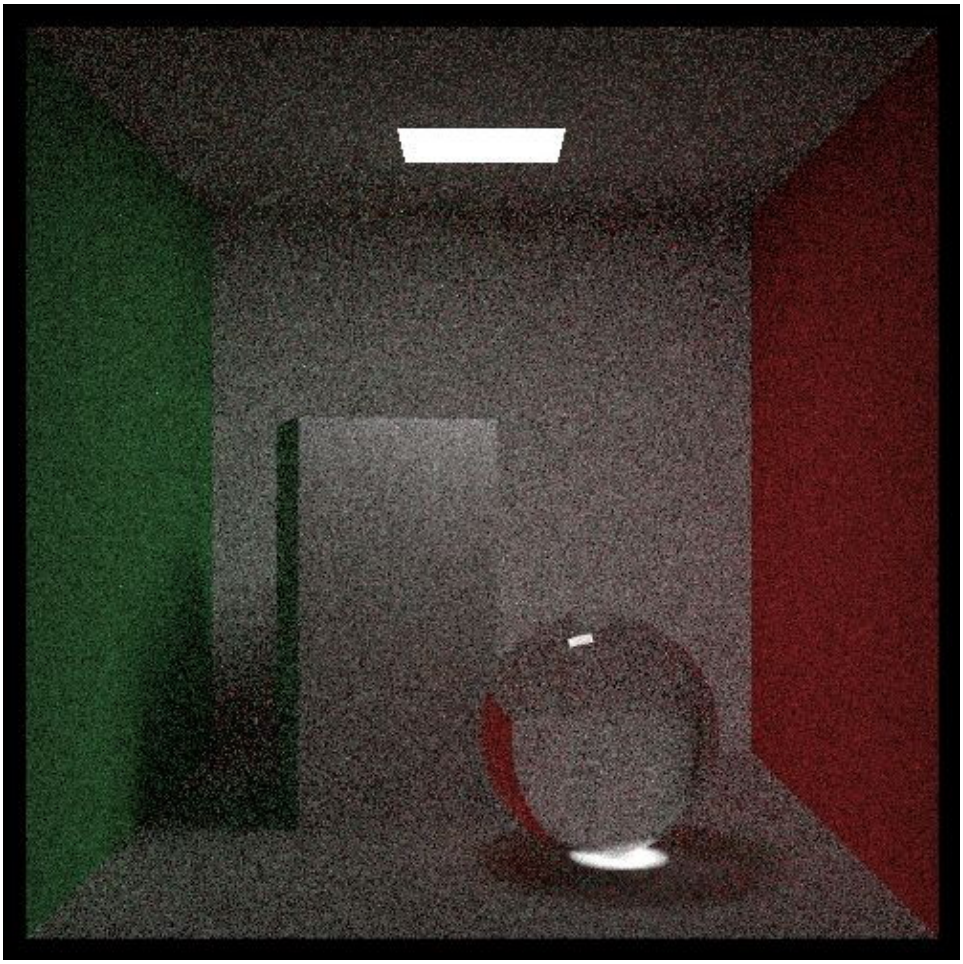

Image 10: Cornell box with glass sphere, using new PDF functions

12.5. Adding PDF Functions to Hittable Lists

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 `ray_color()` function by passing a list of hittables in and building a mixture PDF, or we could add PDF functions to `hittable_list`. I think both tactics would work fine, but I will go with instrumenting `hittable_list`.

```
double hittable_list::pdf_value(const point3& o, const vec3& v) const {
    auto weight = 1.0/objects.size();
    auto sum = 0.0;

    for (const auto& object : objects)
        sum += weight * object->pdf_value(o, v);

    return sum;
}

vec3 hittable_list::random(const vec3& o) const {
    auto int_size = static_cast<int>(objects.size());
    return objects[random_int(0, int_size-1)]->random(o);
}
```

Listing 41: `[hittable_list.h]` *Creating a mixture of densities*

We assemble a list to pass to `ray_color()` `from main()`:

```
hittable_list lights;
lights.add(make_shared<xz_rect>(213, 343, 227, 332, 554, 0));
lights.add(make_shared<sphere>(point3(190, 90, 190), 90, 0));
```

Listing 42: `[main.cc]` *Updating the scene*

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

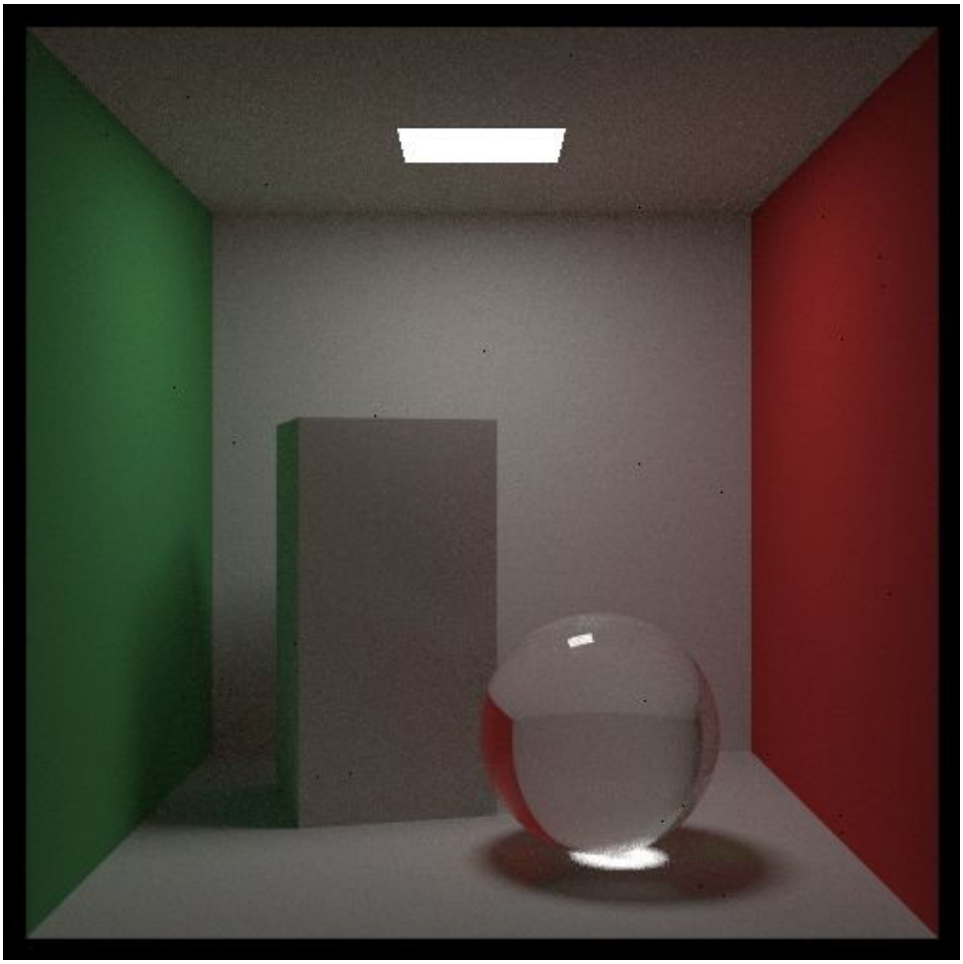

Image 11: Cornell Cornell box, using a mixture of glass & light PDFs

12.6. Handling Surface Acne

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` (Not A Number). 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 `NaN`s, or just kill `NaN`s 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 `NaN`s is that a `NaN` does not equal itself. Using this trick, we update the `write_color()` function to replace any NaN components with zero:

```
void write_color(std::ostream &out, color pixel_color, int samples_per_pixel) {
    auto r = pixel_color.x();
    auto g = pixel_color.y();
    auto b = pixel_color.z();

    // Replace NaN components with zero. See explanation in Ray Tracing: The Rest of Your Life.
    if (r != r) r = 0.0;
    if (g != g) g = 0.0;
    if (b != b) b = 0.0;

    // Divide the color by the number of samples and gamma-correct for gamma=2.0.
    auto scale = 1.0 / samples_per_pixel;
    r = sqrt(scale * r);
    g = sqrt(scale * g);
    b = sqrt(scale * b);

    // Write the translated [0,255] value of each color component.
    out << static_cast<int>(256 * clamp(r, 0.0, 0.999)) << ' '
        << static_cast<int>(256 * clamp(g, 0.0, 0.999)) << ' '
        << static_cast<int>(256 * clamp(b, 0.0, 0.999)) << '\n';
}
```

Listing 43: `[color.h]` *NaN-tolerant write_color function*

Happily, the black specks are gone:

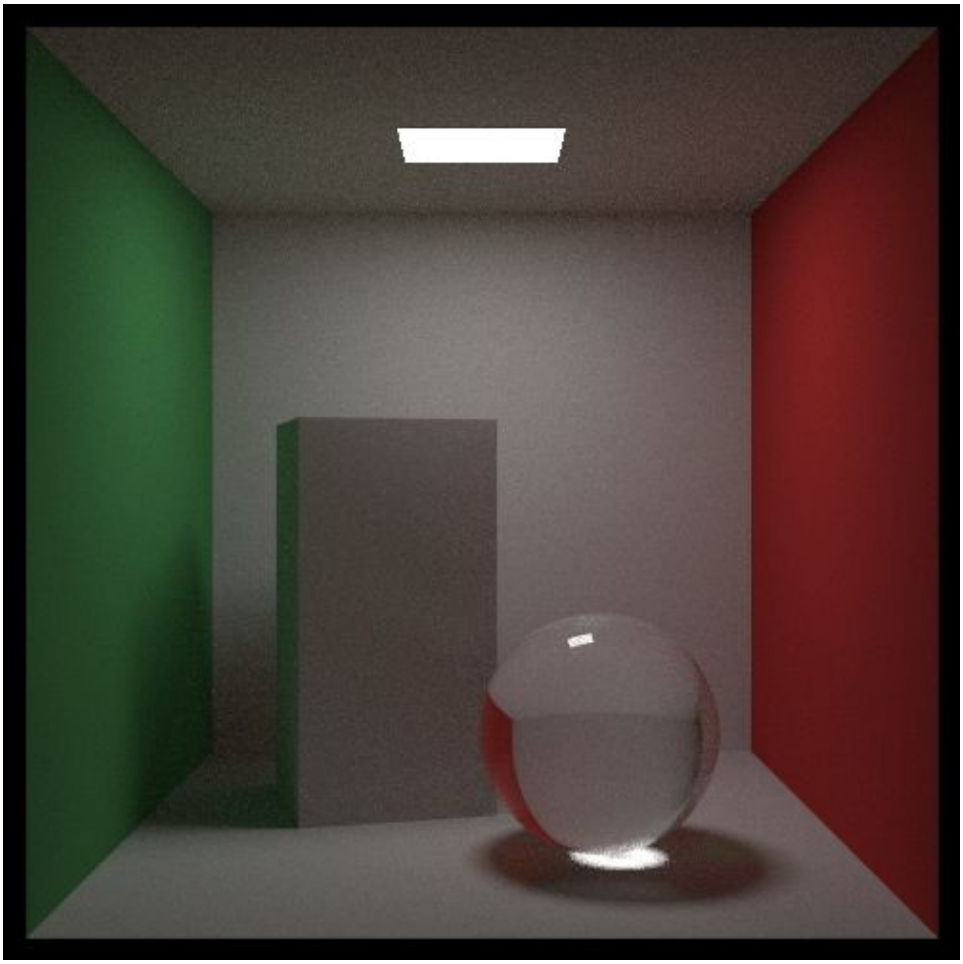

Image 12: Cornell box with anti-acne color function

13. 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 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.

Have fun!

Peter Shirley
Salt Lake City, March, 2016

14. Acknowledgments

Original Manuscript Help

Dave Hart Jean Buckley

Web Release

Berna Kabadayı
Lori Whippler Hollasch
Lorenzo Mancini
Ronald Wotzlaw

Corrections and Improvements

Aaryaman Vasishta
Andrew Kensler
Antonio Gamiz
Apoorva Joshi
Aras Pranckevičius
Becker
Ben Kerl
Benjamin Summerton
Bennett Hardwick
Dan Drummond
David Chambers
David Hart
Eric Haines
Fabio Sancinetti
Filipe Scur
Frank He
Gerrit Wessendorf
Grue Debry
Ingo Wald
Jason Stone
Jean Buckley
Joey Cho
John Kilpatrick
Kaan Eraslan
Lorenzo Mancini
Manas Kale
Marcus Ottosson
Mark Craig
Matthew Heimlich
Nakata Daisuke
Paul Melis
Phil Cristensen
Ronald Wotzlaw
Shaun P. Lee
Shota Kawajiri
Tatsuya Ogawa
Thiago Ize
Vahan Sosoyan
ZeHao Chen

Special Thanks

Thanks to the team at Limnu for help on the figures.

These books are entirely written in Morgan McGuire's fantastic and free Markdeep library. To see what this looks like, view the page source from your browser.

Thanks to Helen Hu for graciously donating her https://github.com/RayTracing/ GitHub organization to this project.

15. Citing This Book

Consistent citations make it easier to identify the source, location and versions of this work. If you are citing this book, we ask that you try to use one of the following forms if possible.

15.1. Basic Data

- **Title (series)**: “Ray Tracing in One Weekend Series”
- **Title (book)**: “Ray Tracing: The Rest of Your Life”
- **Author**: Peter Shirley
- **Editors**: Steve Hollasch, Trevor David Black
- **Version/Edition**: v3.2.3
- **Date**: 2020-12-07
- **URL (series)**: https://raytracing.github.io/
- **URL (book)**: https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html

15.2. Snippets

15.2.1 Markdown

```
[_Ray Tracing: The Rest of Your Life_](https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html)
```

15.2.2 HTML

```
<a href="https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html">
    <cite>Ray Tracing: The Rest of Your Life</cite>
</a>
```

15.2.3 LaTeX and BibTex

```
~\cite{Shirley2020RTW3}

@misc{Shirley2020RTW3,
   title = {Ray Tracing: The Rest of Your Life},
   author = {Peter Shirley},
   year = {2020},
   month = {December}
   note = {\small \texttt{https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html}},
   url = {https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html}
}
```

15.2.4 BibLaTeX

```
\usepackage{biblatex}

~\cite{Shirley2020RTW3}

@online{Shirley2020RTW3,
   title = {Ray Tracing: The Rest of Your Life},
   author = {Peter Shirley},
   year = {2020},
   month = {December}
   url = {https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html}
}
```

15.2.5 IEEE

```
“Ray Tracing: The Rest of Your Life.”
raytracing.github.io/books/RayTracingTheRestOfYourLife.html
(accessed MMM. DD, YYYY)
```

15.2.6 MLA:

```
Ray Tracing: The Rest of Your Life. raytracing.github.io/books/RayTracingTheRestOfYourLife.html
Accessed DD MMM. YYYY.
```

formatted by Markdeep 1.13