Ray Tracing: The Rest of Your Life

Peter Shirley, Trevor David Black, Steve Hollasch

Version 4.0.1, 2024-08-31

Contents

1. Overview

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

If you are motivated, you can take the source and information contained in those books to implement any visual effect you want. The source provides a meaningful and robust foundation upon which to build out a raytracer for a small hobby project. Most of the visual effects found in commercial ray tracers rely on the techniques described in these first two books. However, your capacity to add increasingly complicated visual effects like subsurface scattering or nested dielectrics will be severely limited by a missing mathematical foundation. In this volume, I assume that you are either a highly interested student, or are someone who is pursuing a career related to ray tracing. We will be diving into the math of creating a very serious ray tracer. When you are done, you should be well equipped to use and modify the various commercial ray tracers found in many popular domains, such as the movie, television, product design, and architecture industries.

There are many many things I do not cover in this short volume. For example, there are many ways of writing Monte Carlo rendering programs—I dive into only one of them. I don’t cover shadow rays (deciding instead to make rays more likely to go toward lights), nor do I cover bidirectional methods, Metropolis methods, or photon mapping. You'll find many of these techniques in the so-called “serious ray tracers”, but they are not covered here because it is more important to cover the concepts, math, and terms of the field. I think of this book as a deep exposure that should be your first of many, and it will equip you with some of the concepts, math, and terms that you'll need in order to study these and other interesting techniques.

I hope that you find the math as fascinating as I do.

See the project README file for information about this project, the repository on GitHub, directory structure, building & running, and how to make or reference corrections and contributions.

As before, see our Further Reading wiki page for additional project related resources.

These books have been formatted to print well directly from your browser. We also include PDFs of each book with each release, in the “Assets” section.

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 programs. If you're not familiar with Monte Carlo programs, then it'll be good to pause and catch you up. There are two kinds of randomized algorithms: Monte Carlo and Las Vegas. Randomized algorithms can be found everywhere in computer graphics, so getting a decent foundation isn't a bad idea. A randomized algorithm uses some amount of randomness in its computation. A Las Vegas random algorithm always produces the correct result, whereas a Monte Carlo algorithm *may* produce a correct result—and frequently gets it wrong! But for especially complicated problems such as ray tracing, we may not place as huge a priority on being perfectly exact as on getting an answer in a reasonable amount of time. Las Vegas algorithms will eventually arrive at the correct result, but we can't make too many guarantees on how long it will take to get there. The classic example of a Las Vegas algorithm is the *quicksort* sorting algorithm. The quicksort algorithm will always complete with a fully sorted list, but, the time it takes to complete is random. Another good example of a Las Vegas algorithm is the code that we use to pick a random point in a unit sphere:

```
inline vec3 random_in_unit_sphere() {
    while (true) {
        auto p = vec3::random(-1,1);
        if (p.length_squared() < 1)
            return p;
    }
}
```

Listing 1: `[vec3.h]` *A Las Vegas algorithm*

This code will always eventually arrive at a random point in the unit sphere, but we can't say beforehand how long it'll take. It may take only 1 iteration, it may take 2, 3, 4, or even longer. Whereas, a Monte Carlo program will give a statistical estimate of an answer, and this estimate will get more and more accurate the longer you run it. Which means that at a certain point, we can just decide that the answer is accurate *enough* and call it quits. This basic characteristic of simple programs producing noisy but ever-better answers is what Monte Carlo is all about, and is especially good for applications like graphics where great accuracy is not needed.

2.1. Estimating Pi

The canonical example of a Monte Carlo algorithm is estimating π, so let's do that. There are many ways to estimate π, with *Buffon's needle problem* being a classic case study. In Buffon's needle problem, one is presented with a floor made of parallel strips of floor board, each of the same width. If a needle is randomly dropped onto the floor, what is the probability that the needle will lie across two boards? (You can find more information on this problem with a simple Internet search.)

We’ll do a variation inspired by this method. Suppose you have a circle inscribed inside a square:

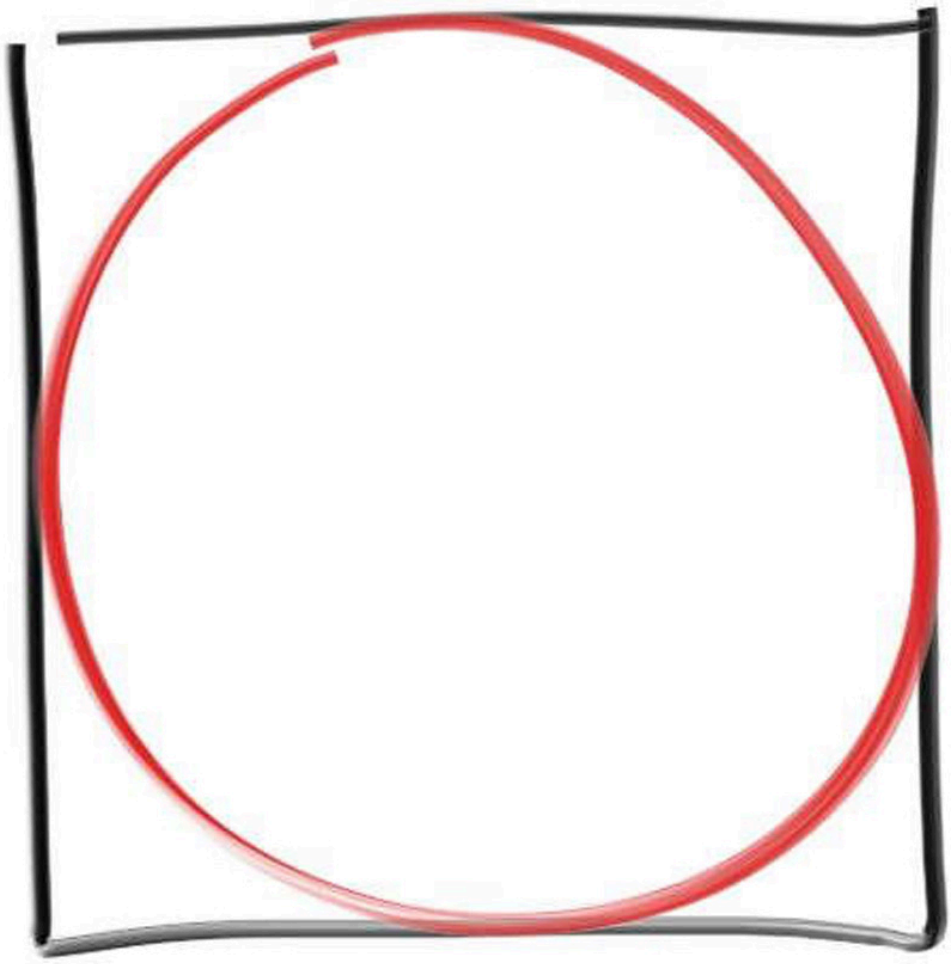

Figure 1: *Estimating π with a circle inside a square*

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:

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

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>

int main() {
    std::cout << std::fixed << std::setprecision(12);

    int inside_circle = 0;
    int N = 100000;

    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 << "Estimate of Pi = " << (4.0 * inside_circle) / N << '\n';
}
```

Listing 2: `[pi.cc]` *Estimating π — version one*

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 = 3.143760000000`.

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>

int main() {
    std::cout << std::fixed << std::setprecision(12);

    int inside_circle = 0;
    int runs = 0;
    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 << "\rEstimate of Pi = " << (4.0 * inside_circle) / runs;
    }

    std::cout << "Estimate of Pi = " << (4.0 * inside_circle) / N << '\n';
}
```

Listing 3: `[pi.cc]` *Estimating π — version two*

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 Monte Carlo. 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:

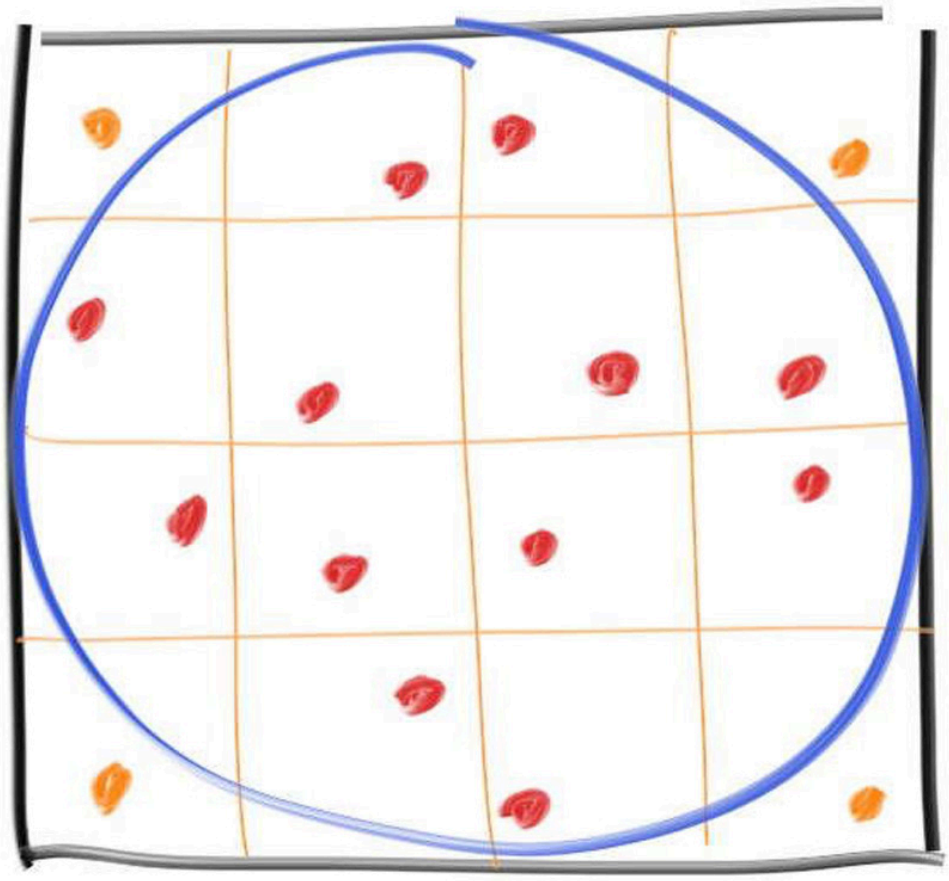

Figure 2: *Sampling areas with jittered points*

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 million and try it both ways:

```
#include "rtweekend.h"

#include <iostream>
#include <iomanip>

int main() {
    std::cout << std::fixed << std::setprecision(12);

    int inside_circle = 0;
    int inside_circle_stratified = 0;
    int sqrt_N = 1000;

    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++;
        }
    }

    std::cout
        << "Regular    Estimate of Pi = "
        << (4.0 * inside_circle) / (sqrt_N*sqrt_N) << '\n'
        << "Stratified Estimate of Pi = "
        << (4.0 * inside_circle_stratified) / (sqrt_N*sqrt_N) << '\n';
}
```

Listing 4: `[pi.cc]` *Estimating π — version three*

On my computer, I get:

```
Regular    Estimate of Pi = 3.141184000000
Stratified Estimate of Pi = 3.141460000000
```

Where the first 12 decimal places of pi are:

```
3.141592653589
```

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*. Ray tracing is a very high-dimensional algorithm, where each reflection adds two new dimensions: ϕ_o and θ_o. We won't be stratifying the output reflection angle in this book, simply because it is a little bit too complicated to cover, but there is a lot of interesting research currently happening in this space.

As an intermediary, we'll be stratifying the locations of the sampling positions around each pixel location. Let's start with our familiar Cornell box scene.

```
#include "rtweekend.h"

#include "camera.h"
#include "hittable_list.h"
#include "material.h"
#include "quad.h"
#include "sphere.h"

int main() {
    hittable_list world;

    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));

    // Cornell box sides
    world.add(make_shared<quad>(point3(555,0,0), vec3(0,0,555), vec3(0,555,0), green));
    world.add(make_shared<quad>(point3(0,0,555), vec3(0,0,-555), vec3(0,555,0), red));
    world.add(make_shared<quad>(point3(0,555,0), vec3(555,0,0), vec3(0,0,555), white));
    world.add(make_shared<quad>(point3(0,0,555), vec3(555,0,0), vec3(0,0,-555), white));
    world.add(make_shared<quad>(point3(555,0,555), vec3(-555,0,0), vec3(0,555,0), white));

    // Light
    world.add(make_shared<quad>(point3(213,554,227), vec3(130,0,0), vec3(0,0,105), light));

    // Box 1
    shared_ptr<hittable> box1 = 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));
    world.add(box1);

    // Box 2
    shared_ptr<hittable> box2 = 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));
    world.add(box2);

    camera cam;

    cam.aspect_ratio      = 1.0;
    cam.image_width       = 600;
    cam.samples_per_pixel = 64;
    cam.max_depth         = 50;
    cam.background        = color(0,0,0);

    cam.vfov     = 40;
    cam.lookfrom = point3(278, 278, -800);
    cam.lookat   = point3(278, 278, 0);
    cam.vup      = vec3(0, 1, 0);

    cam.defocus_angle = 0;

    cam.render(world);
}
```

Listing 5: `[main.cc]` *Cornell box, revisited*

Run this program to generate an un-stratified render and save for comparison.

Now make the following changes to implement a stratified sampling procedure:

```
class camera {
  public:
    ...

    void render(const hittable& world) {
        initialize();

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

        for (int j = 0; j < image_height; j++) {
            std::clog << "\rScanlines remaining: " << (image_height - j) << ' ' << std::flush;
            for (int i = 0; i < image_width; i++) {
                color pixel_color(0,0,0);
                for (int s_j = 0; s_j < sqrt_spp; s_j++) {
                    for (int s_i = 0; s_i < sqrt_spp; s_i++) {
                        ray r = get_ray(i, j, s_i, s_j);
                        pixel_color += ray_color(r, max_depth, world);
                    }
                }
                write_color(std::cout, pixel_samples_scale * pixel_color);
            }
        }

        std::clog << "\rDone.                 \n";
    }

  private:
    int    image_height;         // Rendered image height
    double pixel_samples_scale;  // Color scale factor for a sum of pixel samples
    int    sqrt_spp;             // Square root of number of samples per pixel
    double recip_sqrt_spp;       // 1 / sqrt_spp
    point3 center;               // Camera center
    ...

    void initialize() {
        image_height = int(image_width / aspect_ratio);
        image_height = (image_height < 1) ? 1 : image_height;

        sqrt_spp = int(std::sqrt(samples_per_pixel));
        pixel_samples_scale = 1.0 / (sqrt_spp * sqrt_spp);
        recip_sqrt_spp = 1.0 / sqrt_spp;

        center = lookfrom;
        ...
    }

    ray get_ray(int i, int j, int s_i, int s_j) const {
        // Construct a camera ray originating from the defocus disk and directed at a randomly
        // sampled point around the pixel location i, j for stratified sample square s_i, s_j.

        auto offset = sample_square_stratified(s_i, s_j);
        auto pixel_sample = pixel00_loc
                          + ((i + offset.x()) * pixel_delta_u)
                          + ((j + offset.y()) * pixel_delta_v);

        auto ray_origin = (defocus_angle <= 0) ? center : defocus_disk_sample();
        auto ray_direction = pixel_sample - ray_origin;
        auto ray_time = random_double();

        return ray(ray_origin, ray_direction, ray_time);
    }

    vec3 sample_square_stratified(int s_i, int s_j) const {
        // Returns the vector to a random point in the square sub-pixel specified by grid
        // indices s_i and s_j, for an idealized unit square pixel [-.5,-.5] to [+.5,+.5].

        auto px = ((s_i + random_double()) * recip_sqrt_spp) - 0.5;
        auto py = ((s_j + random_double()) * recip_sqrt_spp) - 0.5;

        return vec3(px, py, 0);
    }
```

```
    vec3 sample_square() const {
        ...
    }

    ...
};
```

Listing 6: `[camera.h]` *Stratifying the samples inside pixels*

If we compare the results from without stratification:

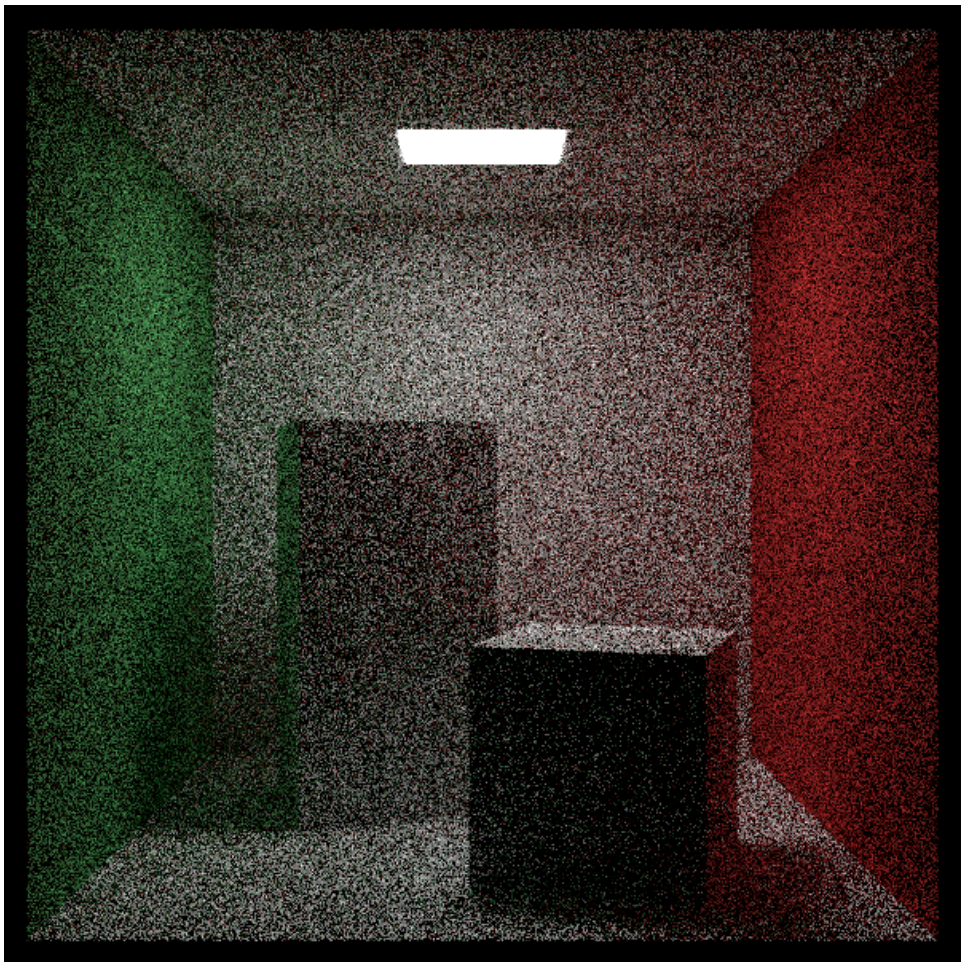

Image 1: *Cornell box, no stratification*

To after, with stratification:

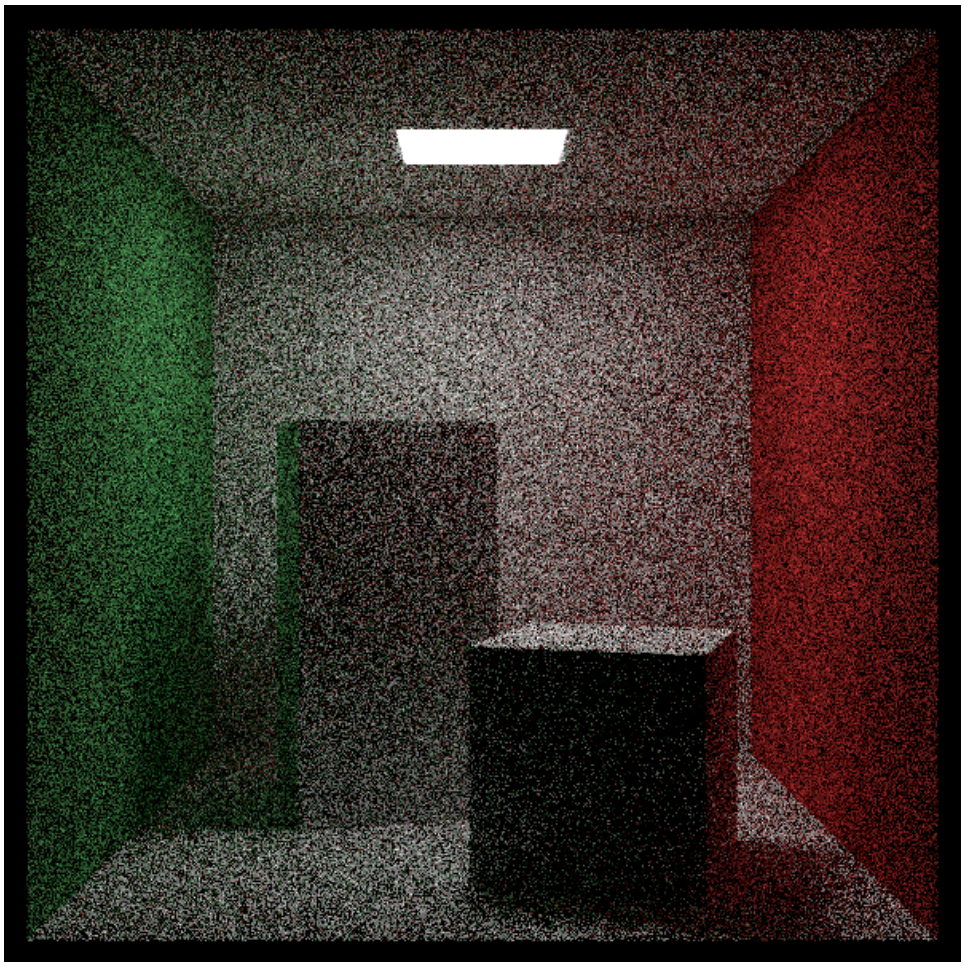

Image 2: *Cornell box, with stratification*

You should, if you squint, be able to see sharper contrast at the edges of planes and at the edges of boxes. The effect will be more pronounced at locations that have a higher frequency of change. High frequency change can also be thought of as high information density. For our cornell box scene, all of our materials are matte, with a soft area light overhead, so the only locations of high information density are at the edges of objects. The effect will be more obvious with textures and reflective materials.

If you are ever doing single-reflection or shadowing or some strictly 2D problem, you definitely want to stratify.

3. One Dimensional Monte Carlo Integration

Our variation of Buffon's needle problem is a way of calculating π by solving for the ratio of the area of the circle and the area of the circumscribed square:

$$\frac{\text{area}(circle)}{\text{area}(square)} = \frac{\pi}{4}$$

We picked a bunch of random points in the circumscribed square and counted the fraction of them that were also in the unit circle. This fraction was an estimate that tended toward $\frac{\pi}{4}$ as more points were added. If we didn't know the area of a circle, we could still solve for it using the above ratio. We know that the ratio of areas of the unit circle and the circumscribed square is $\frac{\pi}{4}$, and we know that the area of a circumscribed square is $4r^2$, so we could then use those two quantities to get the area of a circle:

$$\frac{\text{area}(circle)}{\text{area}(square)} = \frac{\pi}{4}$$

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

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

$$\text{area}(circle) = \pi r^2$$

We choose a circle with radius $r = 1$ and get:

$$\text{area}(circle) = \pi$$

Our work above is equally valid as a means to solve for π as it is a means to solve for the area of a circle. So we could make the following substitution in one of the first versions of our pi program:

```
    std::cout << "Estimate of Pi = "                    << (4.0 * inside_circle) / N << '\n';
    std::cout << "Estimated area of unit circle = " << (4.0 * inside_circle) / N << '\n';
```

Listing 7: `[pi.cc]` *Estimating area of unit circle*

3.1. Expected Value

Let's take a step back and think about our Monte Carlo algorithm a little bit more generally.

If we assume that we have all of the following:

1. A list of values X that contains members x_i:

$$X = (x_0, x_1, \ldots, x_{N-1})$$

2. A continuous function $f(x)$ that takes members from the list:

$$y_i = f(x_i)$$

3. A function $F(X)$ that takes the list X as input and produces the list Y as output:

$$Y = F(X)$$

4. Where output list Y has members y_i:

$$Y = (y_0, y_1, \ldots, y_{N-1}) = (f(x_0), f(x_1), \ldots, f(x_{N-1}))$$

If we assume all of the above, then we could solve for the arithmetic mean—the average—of the list Y with the following:

$$\text{average}(Y_i) = E[Y] = \frac{1}{N} \sum_{i=0}^{N-1} y_i$$

$$= \frac{1}{N} \sum_{i=0}^{N-1} f(x_i)$$

$$= E[F(X)]$$

Where $E[Y]$ is referred to as the *expected value of* Y.

Note the subtle difference between *average value* and *expected value* here:

- A set may have many different subsets of selections from that set. Each subset has an *average value*, which is the sum of all selections divided by the count of selections. Note that a given item may occur zero times, one times, or multiple times in a subset.
- A set has only one *expected value*: the sum of *all* members of a set, divided by the total number of items in that set. Put another way, the *expected value* is the *average value* of *all* members of a set.

It is important to note that as the number of random samples from a set increases, the average value of a set will converge to the expected value.

If the values of x_i are chosen randomly from a continuous interval $[a, b]$ such that $a \leq x_i \leq b$ for all values of i, then $E[F(X)]$ will approximate the average of the continuous function $f(x')$ over the the same interval $a \leq x' \leq b$.

$$E[f(x')|a \leq x' \leq b] \approx E[F(X)|X = \{x_i|a \leq x_i \leq b\}]$$

$$\approx E[Y = \{y_i = f(x_i)|a \leq x_i \leq b\}]$$

$$\approx \frac{1}{N}\sum_{i=0}^{N-1} f(x_i)$$

If we take the number of samples N and take the limit as N goes to ∞, then we get the following:

$$E[f(x')|a \leq x' \leq b] = \lim_{N\to\infty} \frac{1}{N}\sum_{i=0}^{N-1} f(x_i)$$

Within the continuous interval $[a, b]$, the expected value of continuous function $f(x')$ can be perfectly represented by summing an infinite number of random points within the interval. As this number of points approaches ∞ the average of the outputs tends to the exact answer. This is a Monte Carlo algorithm.

Sampling random points isn't our only way to solve for the expected value over an interval. We can also choose where we place our sampling points. If we had N samples over an interval $[a, b]$ then we could choose to equally space points throughout:

$$x_i = a + i\Delta x$$

$$\Delta x = \frac{b-a}{N}$$

Then solving for their expected value:

$$E[f(x')|a \leq x' \leq b] \approx \frac{1}{N}\sum_{i=0}^{N-1} f(x_i)\Big|_{x_i=a+i\Delta x}$$

$$E[f(x')|a \leq x' \leq b] \approx \frac{\Delta x}{b-a}\sum_{i=0}^{N-1} f(x_i)\Big|_{x_i=a+i\Delta x}$$

$$E[f(x')|a \leq x' \leq b] \approx \frac{1}{b-a}\sum_{i=0}^{N-1} f(x_i)\Delta x\Big|_{x_i=a+i\Delta x}$$

Take the limit as N approaches ∞

$$E[f(x')|a \leq x' \leq b] = \lim_{N\to\infty} \frac{1}{b-a}\sum_{i=0}^{N-1} f(x_i)\Delta x\Big|_{x_i=a+i\Delta x}$$

This is, of course, just a regular integral:

$$E[f(x')|a \leq x' \leq b] = \frac{1}{b-a} \int_a^b f(x)dx$$

If you recall your introductory calculus class, the integral of a function is the area under the curve over that interval:

$$\text{area}(f(x), a, b) = \int_a^b f(x)dx$$

Therefore, the average over an interval is intrinsically linked with the area under the curve in that interval.

$$E[f(x)|a \leq x \leq b] = \frac{1}{b-a} \cdot \text{area}(f(x), a, b)$$

Both the integral of a function and a Monte Carlo sampling of that function can be used to solve for the average over a specific interval. While integration solves for the average with the sum of infinitely many infinitesimally small slices of the interval, a Monte Carlo algorithm will approximate the same average by solving the sum of ever increasing random sample points within the interval. Counting the number of points that fall inside of an object isn't the only way to measure its average or area. Integration is also a common mathematical tool for this purpose. If a closed form exists for a problem, integration is frequently the most natural and clean way to formulate things.

I think a couple of examples will help.

3.2. Integrating x²

Let's look at a classic integral:

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

We could solve this using integration:

$$I = \frac{1}{3}x^3\Big|_0^2$$

$$I = \frac{1}{3}(2^3 - 0^3)$$

$$I = \frac{8}{3}$$

Or, we could solve the integral using a Monte Carlo approach. In computer sciency notation, we might write this as:

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

We could also write it as:

$$E[f(x)|a \leq x \leq b] = \frac{1}{b-a} \cdot \text{area}(f(x), a, b)$$

$$\text{average}(x^2, 0, 2) = \frac{1}{2-0} \cdot \text{area}(x^2, 0, 2)$$

$$\text{average}(x^2, 0, 2) = \frac{1}{2-0} \cdot I$$

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

The Monte Carlo approach:

```
#include "rtweekend.h"

#include <iostream>
#include <iomanip>

int main() {
    int a = 0;
    int b = 2;
    int N = 1000000;
    auto sum = 0.0;

    for (int i = 0; i < N; i++) {
        auto x = random_double(a, b);
        sum += x*x;
    }

    std::cout << std::fixed << std::setprecision(12);
    std::cout << "I = " << (b - a) * (sum / N) << '\n';
}
```

Listing 8: `[integrate_x_sq.cc]` *Integrating* x^2

This, as expected, produces approximately the exact answer we get with integration, *i.e.* $I = 2.666\ldots = \frac{8}{3}$. You could rightly point to this example and say that the integration is actually a lot less work than the Monte Carlo. That might be true in the case where the function is $f(x) = x^2$, but there exist many functions where it might be simpler to solve for the Monte Carlo than for the integration, like $f(x) = \sin^5(x)$.

```
    for (int i = 0; i < N; i++) {
        auto x = random_double(a, b);
        sum += std::pow(std::sin(x), 5.0);
    }
```

Listing 9: *Integrating* $\sin^5$

We could also use the Monte Carlo algorithm for functions where an analytical integration does not exist, like $f(x) = \ln(\sin(x))$.

```
for (int i = 0; i < N; i++) {
    auto x = random_double(a, b);
    sum += std::log(std::sin(x));
}
```

Listing 10: *Integrating* $\ln(\sin)$

In graphics, we often have functions that we can write down explicitly but that have a complicated analytic integration, or, just as often, we have functions that *can* be evaluated but that *can't* be written down explicitly, and we will frequently find ourselves with a function that can *only* be evaluated probabilistically. The function `ray_color` from the first two books is an example of a function that can only be determined probabilistically. We can't know what color can be seen from any given place in all directions, but we can statistically estimate which color can be seen from one particular place, for a single particular direction.

3.3. Density Functions

The `ray_color` function that we wrote in the first two books, while elegant in its simplicity, has a fairly *major* problem. 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. If the background color is black, then the only real sources of light in the scene are from the lights that are actually placed about the scene. There might be two rays that intersect at nearby points on a surface, one that is randomly reflected toward the light and one that is not. The ray that is reflected toward the light will appear a very bright color. The ray that is reflected to somewhere else will appear a very dark color. The two intensities should really be somewhere in the middle. We could lessen this problem if we steered both of these rays toward the light, but this would cause the scene to be inaccurately bright.

For any given ray, we usually trace from the camera, through the scene, and terminate at a light. But imagine if we traced this same ray from the light source, through the scene, and terminated at the camera. This ray would start with a bright intensity and would lose energy with each successive bounce around the scene. It would ultimately arrive at the camera, having been dimmed and colored by its reflections off various surfaces. Now, imagine if this ray was forced to bounce toward the camera as soon as it could. It would appear inaccurately bright because it hadn't been dimmed by successive bounces. This is analogous to sending more random samples toward the light. It would go a long way toward solving our problem of having a bright pixel next to a dark pixel, but it would then just make *all* of our pixels bright.

We can remove this inaccuracy by downweighting those samples to adjust for the over-sampling. How do we do this adjustment? Well, we'll first need to understand the concept of a *probability density function*. But to understand the concept of a *probability density function*, we'll first need to know what a *density function* is.

A *density function* is just the continuous version of a histogram. Here's an example of a histogram from the histogram Wikipedia page:

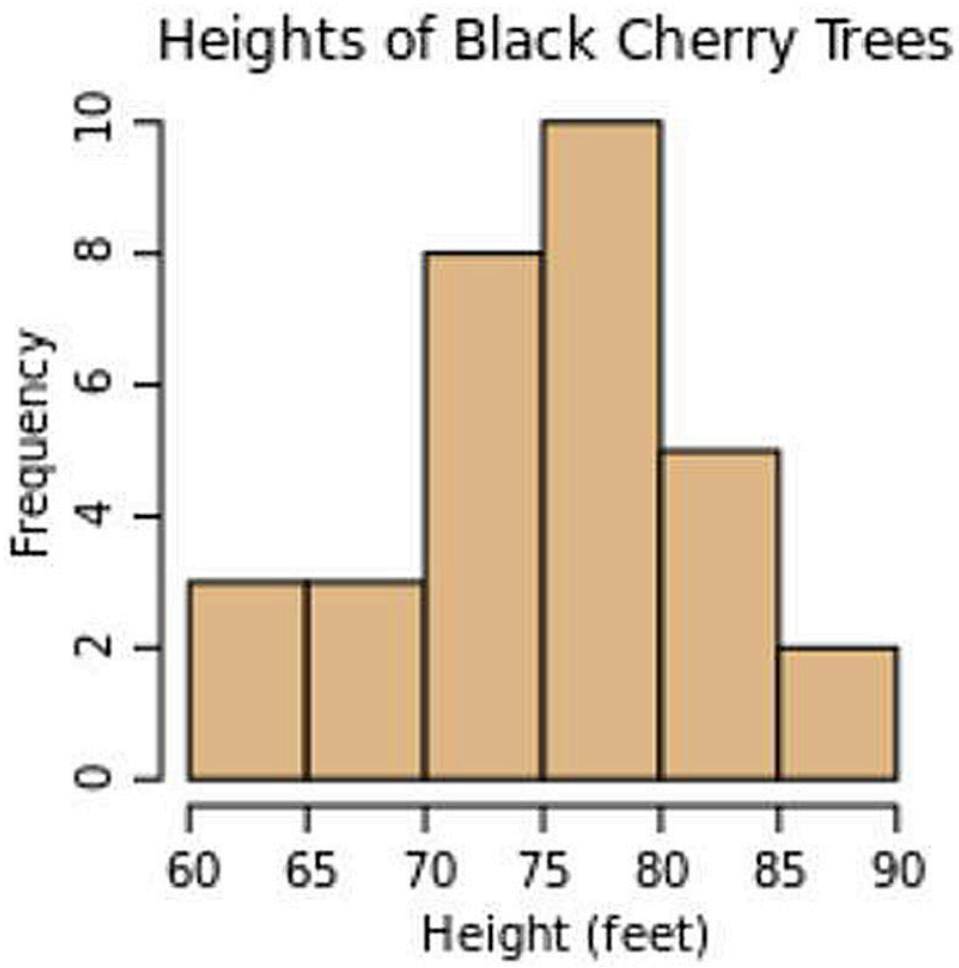

Figure 3: *Histogram example*

If we had more items in our data source, the number of bins would stay the same, but each bin would have a higher frequency of each item. If we divided the data into more bins, we'd have more bins, but each bin would have a lower frequency of each item. If we took the number of bins and raised it to infinity, we'd have an infinite number of zero-frequency bins. To solve for this, we'll replace our histogram, which is a *discrete function*, with a *discrete density function*. A *discrete density function* differs from a *discrete function* in that it normalizes the y-axis to a fraction or percentage of the total, *i.e* its density, instead of a total count for each bin. Converting from a *discrete function* to a *discrete density function* is trivial:

$$\text{Density of Bin i} = \frac{\text{Number of items in Bin i}}{\text{Number of items total}}$$

Once we have a *discrete density function*, we can then convert it into a *density function* by changing our discrete values into continuous values.

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

So a *density function* is a continuous histogram where all of the values are normalized against a total. If we had a specific tree we wanted to know the height of, we could create a *probability function* that would tell us how likely it is for our tree to fall within a specific bin.

$$\text{Probability of Bin i} = \frac{\text{Number of items in Bin i}}{\text{Number of items total}}$$

If we combined our *probability function* and our (continuous) *density function*, we could interpret that as a statistical predictor of a tree's height:

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

Indeed, with this continuous probability function, we can now say the likelihood that any given tree has a height that places it within any arbitrary span of multiple bins. This is a *probability density function* (henceforth *PDF*). In short, a PDF is a continuous function that can be integrated over to determine how likely a result is over an integral.

3.4. Constructing a PDF

Let's make a PDF and play around with it to build up an intuition. We'll use the following function:

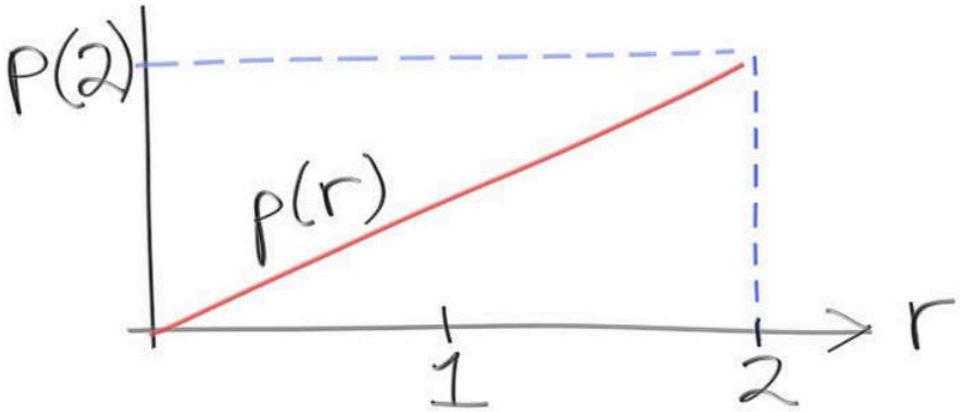

Figure 4: *A linear PDF*

What does this function do? Well, we know that a PDF is just a continuous function that defines the likelihood of an arbitrary range of values. This function $p(r)$ is constrained between 0 and 2 and linearly increases along that interval. So, if we used this function as a PDF to generate a random number then the *probability* of getting a number near zero would be less than the probability of getting a number near two.

The PDF $p(r)$ is a linear function that starts with 0 at $r = 0$ and monotonically increases to its highest point at $p(2)$ for $r = 2$. What is the value of $p(2)$? What is the value of $p(r)$? Maybe $p(2)$ is 2? The PDF increases linearly from 0 to 2, so guessing that the value of $p(2)$ is 2 seems reasonable. At least it looks like it can't be 0.

Remember that the PDF is a probability function. We are constraining the PDF so that it lies in the range [0,2]. The PDF represents the continuous density function for a probabilistic list. If we know that everything in that list is contained within 0 and 2, we can say that the probability of getting a value between 0 and 2 is 100%. Therefore, the area under the curve must sum to 1:

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

All linear functions can be represented as a constant term multiplied by a variable.

$$p(r) = C \cdot r$$

We need to solve for the value of C. We can use integration to work backwards.

$$\begin{aligned} 1 &= \text{area}(p(r), 0, 2) \\ &= \int_0^2 C \cdot r dr \\ &= C \cdot \int_0^2 r dr \\ &= C \cdot \frac{r^2}{2} \Big|_0^2 \end{aligned}$$

$$= C(\frac{2^2}{2} - \frac{0}{2})$$

$$C = \frac{1}{2}$$

That gives us the PDF of $p(r) = r/2$. Just as with histograms we can sum up (integrate) the region to figure out the probability that r is in some interval $[x_0, x_1]$:

$$\text{Probability}(r|x_0 \le r \le x_1) = \text{area}(p(r), x_0, x_1)$$

$$\text{Probability}(r|x_0 \le r \le x_1) = \int_{x_0}^{x_1} \frac{r}{2} dr$$

To confirm your understanding, you should integrate over the region $r = 0$ to $r = 2$, you should get a probability of 1.

After spending enough time with PDFs you might start referring to a PDF as the probability that a variable r is value x, *i.e.* $p(r = x)$. Don't do this. For a continuous function, the probability that a variable is a specific value is always zero. A PDF can only tell you the probability that a variable will fall within a given interval. If the interval you're checking against is a single value, then the PDF will always return a zero probability because its “bin” is infinitely thin (has zero width). Here's a simple mathematical proof of this fact:

$$\text{Probability}(r = x) = \int_x^x p(r) dr$$

$$= P(r)\Big|_x^x$$

$$= P(x) - P(x)$$

$$= 0$$

Finding the probability of a region surrounding x may not be zero:

$$\text{Probability}(r|x - \Delta x < r < x + \Delta x) = \text{area}(p(r), x - \Delta x, x + \Delta x)$$

$$= P(x + \Delta x) - P(x - \Delta x)$$

3.5. Choosing our Samples

If we have a PDF for the function that we care about, then we have the probability that the function will return a value within an arbitrary interval. We can use this to determine where we should sample. Remember that this started as a quest to determine the best way to sample a scene so that we wouldn't get very bright pixels next to very dark pixels. If we have a PDF for the scene, then we can probabilistically steer our samples toward the light without making the image inaccurately bright. We already said that if we steer our samples toward the light then we *will* make the image inaccurately bright. We need to figure out how to steer our samples without introducing this inaccuracy, this will be explained a little bit later, but for now we'll focus on generating samples if we have a PDF. How do we generate a random number with a PDF? For that we will need some more machinery. Don’t worry — this doesn’t go on forever!

Our random number generator `random_double()` produces a random double between 0 and 1. The number generator is uniform between 0 and 1, so any number between 0 and 1 has equal likelihood. If our PDF is uniform over a domain, say $[0, 10]$, then we can trivially produce perfect samples for this uniform PDF with

```
10.0 * random_double()
```

That's an easy case, but the vast majority of cases that we're going to care about are nonuniform. We need to figure out a way to convert a uniform random number generator into a nonuniform random number generator, where the distribution is defined by the PDF. We'll just *assume* that there exists a function $f(d)$ that takes uniform input and produces a nonuniform distribution weighted by PDF. We just need to figure out a way to solve for $f(d)$.

For the PDF given above, where $p(r) = \frac{r}{2}$, the probability of a random sample is higher toward 2 than it is toward 0. There is a greater probability of getting a number between 1.8 and 2.0 than between 0.0 and 0.2. If we put aside our mathematics hat for a second and put on our computer science hat, maybe we can figure out a smart way of partitioning the PDF. We know that there is a higher probability near 2 than near 0, but what is the value that splits the probability in half? What is the value that a random number has a 50% chance of being higher than and a 50% chance of being lower than? What is the x that solves:

$$50\% = \int_0^x \frac{r}{2} dr = \int_x^2 \frac{r}{2} dr$$

Solving gives us:

$$0.5 = \frac{r^2}{4}\Big|_0^x$$

$$0.5 = \frac{x^2}{4}$$

$$x^2 = 2$$

$$x = \sqrt{2}$$

As a crude approximation we could create a function `f(d)` that takes as input `double d = random_double()`. If `d` is less than (or equal to) 0.5, it produces a uniform number in $[0, \sqrt{2}]$, if `d` is greater than 0.5, it produces a uniform number in $[\sqrt{2}, 2]$.

```
double f(double d)
{
    if (d <= 0.5)
        return std::sqrt(2.0) * random_double();
    else
        return std::sqrt(2.0) + (2 - std::sqrt(2.0)) * random_double();
}
```

Listing 11: *A crude, first-order approximation to nonuniform PDF*

While our initial random number generator was uniform from 0 to 1:

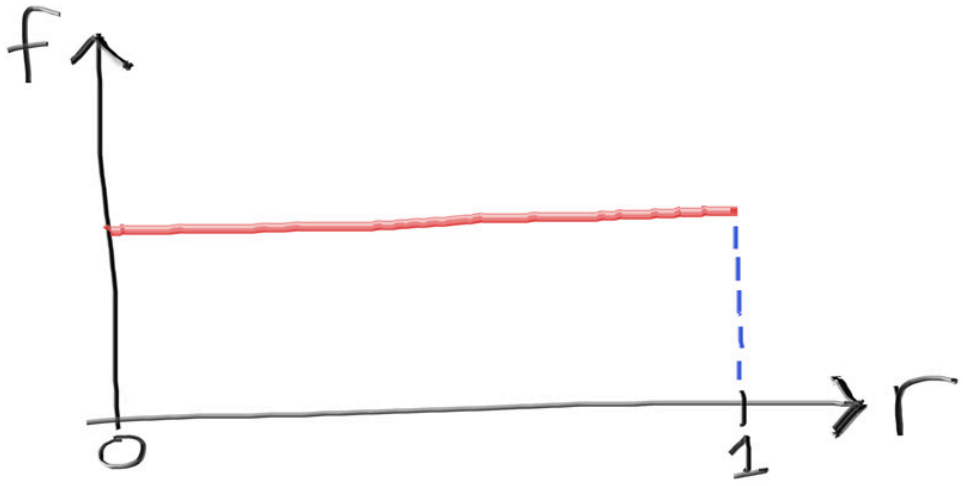

Figure 5: *A uniform distribution*

Our, new, crude approximation for $\frac{r}{2}$ is nonuniform (but only just):

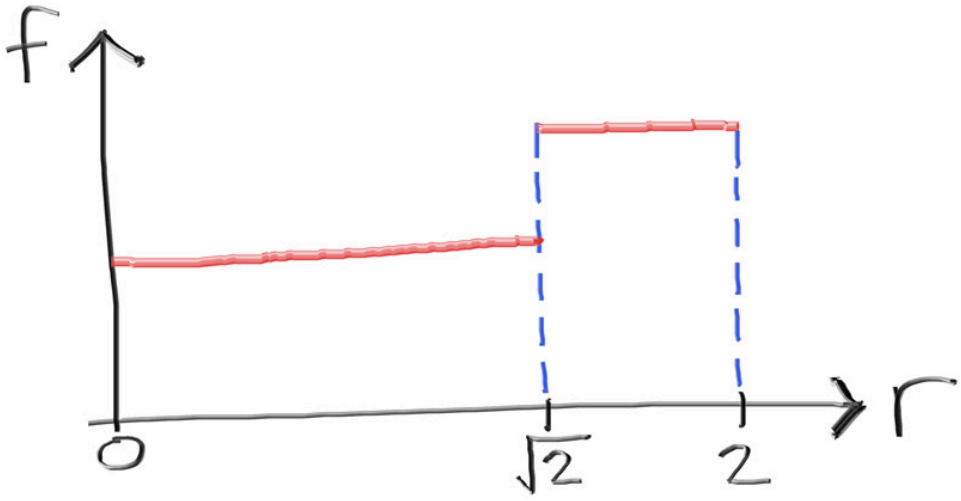

Figure 6: *A nonuniform distribution for $r/2$*

We had the analytical solution to the integration above, so we could very easily solve for the 50% value. But we could also solve for this 50% value experimentally. There will be functions that we either can't or don't want to solve for the integration. In these cases, we can get an experimental result close to the real value. Let's take the function:

$$p(x) = e^{\frac{-x}{2\pi}} \sin^2(x)$$

Which looks a little something like this:

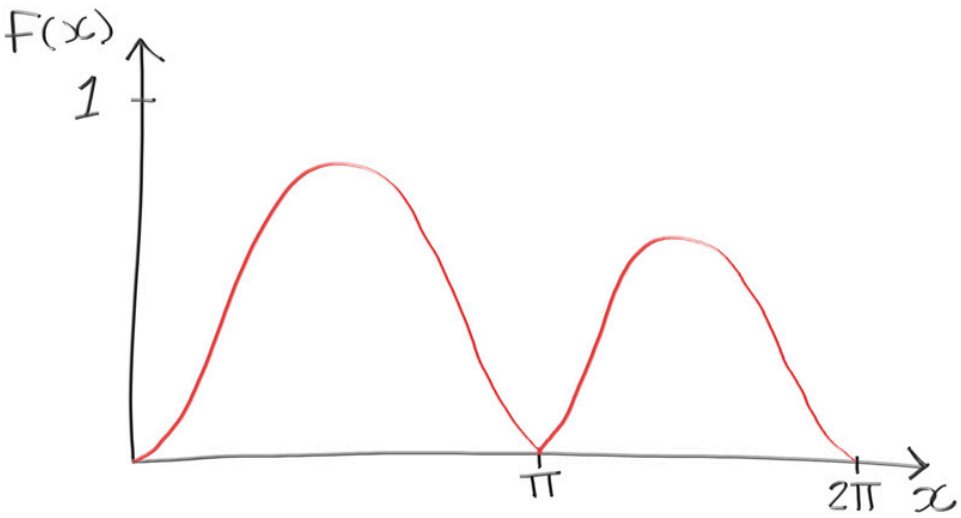

Figure 7: *A function that we don't want to solve analytically*

At this point you should be familiar with how to experimentally solve for the area under a curve. We'll take our existing code and modify it slightly to get an estimate for the 50% value. We want to solve for the x value that gives us half of the total area under the curve. As we go along and solve for the rolling sum over N samples, we're also going to store each individual sample alongside its `p(x)` value. After we solve for the total sum, we'll sort our samples and add them up until we have an area that is half of the total. From 0 to 2π for example:

```
#include "rtweekend.h"

#include <algorithm>
#include <vector>
#include <iostream>
#include <iomanip>

struct sample {
    double x;
    double p_x;
};

bool compare_by_x(const sample& a, const sample& b) {
    return a.x < b.x;
}

int main() {
    const unsigned int N = 10000;
    sample samples[N];
    double sum = 0.0;

    // Iterate through all of our samples.

    for (unsigned int i = 0; i < N; i++) {
        // Get the area under the curve.
        auto x = random_double(0, 2*pi);
        auto sin_x = std::sin(x);
        auto p_x = exp(-x / (2*pi)) * sin_x * sin_x;
        sum += p_x;

        // Store this sample.
        sample this_sample = {x, p_x};
        samples[i] = this_sample;
    }

    // Sort the samples by x.
    std::sort(std::begin(samples), std::end(samples), compare_by_x);

    // Find out the sample at which we have half of our area.
    double half_sum = sum / 2.0;
    double halfway_point = 0.0;
    double accum = 0.0;
    for (unsigned int i = 0; i < N; i++){
        accum += samples[i].p_x;
        if (accum >= half_sum) {
            halfway_point = samples[i].x;
            break;
        }
    }

    std::cout << std::fixed << std::setprecision(12);
    std::cout << "Average = " << sum / N << '\n';
    std::cout << "Area under curve = " << 2 * pi * sum / N << '\n';
    std::cout << "Halfway = " << halfway_point << '\n';
}
```

Listing 12: `[estimate_halfway.cc]` *Estimating the 50% point of a function*

This code snippet isn't too different from what we had before. We're still solving for the sum over an interval (0 to 2π). Only this time, we're also storing and sorting all of our samples by their input and output. We use this to determine the point at which they subtotal half of the sum across the entire interval. Once we know that our first j samples sum up to half of the total sum, we know that the $j\text{th}$ x roughly corresponds to our halfway point:

```
Average = 0.314686555791
Area under curve = 1.977233943713
Halfway = 2.016002314977
```

If you solve for the integral from 0 to 2.016 and from 2.016 to 2π you should get almost exactly the same result for both.

We have a method of solving for the halfway point that splits a PDF in half. If we wanted to, we could use this to create a nested binary partition of the PDF:

1. Solve for halfway point of a PDF
2. Recurse into lower half, repeat step 1
3. Recurse into upper half, repeat step 1

Stopping at a reasonable depth, say 6–10. As you can imagine, this could be quite computationally expensive. The computational bottleneck for the code above is probably sorting the samples. A naive sorting algorithm can have an algorithmic complexity of $\mathcal{O}(\mathbf{n^2})$ time, which is tremendously expensive. Fortunately, the sorting algorithm included in the standard library is usually much closer to $\mathcal{O}(\mathbf{n}\log\mathbf{n})$ time, but this can still be quite expensive, especially for millions or billions of samples. But this will produce decent nonuniform distributions of nonuniform numbers. This divide and conquer method of producing nonuniform distributions is used somewhat commonly in practice, although there are much more efficient means of doing so than a simple binary partition. If you have an arbitrary function that you wish to use as the PDF for a distribution, you'll want to research the *Metropolis-Hastings Algorithm*.

3.6. Approximating Distributions

This was a lot of math and work to build up a couple of notions. Let's return to our initial PDF. For the intervals without an explicit probability, we assume the PDF to be zero. So for our example from the beginning of the chapter, $p(r) = 0$, for $r \notin [0, 2]$. We can rewrite our $p(r)$ in piecewise fashion:

$$p(r) = \begin{cases} 0 & r < 0 \\ \frac{r}{2} & 0 \leq r \leq 2 \\ 0 & 2 < r \end{cases}$$

If you consider what we were trying to do in the previous section, a lot of math revolved around the *accumulated* area (or *accumulated* probability) from zero. In the case of the function

$$f(x) = e^{\frac{-x}{2\pi}} \sin^2(x)$$

we cared about the accumulated probability from 0 to 2π (100%) and the accumulated probability from 0 to 2.016 (50%). We can generalize this to an important term, the *Cumulative Distribution Function* $P(x)$ is defined as:

$$P(x) = \int_{-\infty}^{x} p(x')dx'$$

Or,

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

Which is the amount of *cumulative* probability from $-\infty$. We rewrote the integral in terms of x' instead of x because of calculus rules, if you're not sure what it means, don't worry about it, you can just treat it like it's the same. If we take the integration outlined above, we get the piecewise $P(r)$:

$$P(r) = \begin{cases} 0 & r < 0 \\ \frac{r^2}{4} & 0 \le r \le 2 \\ 1 & 2 < r \end{cases}$$

The *Probability Density Function* (PDF) is the probability function that explains how likely an interval of numbers is to be chosen. The *Cumulative Distribution Function* (CDF) is the distribution function that explains how likely all numbers smaller than its input is to be chosen. To go from the PDF to the CDF, you need to integrate from $-\infty$ to x, but to go from the CDF to the PDF, all you need to do is take the derivative:

$$p(x) = \frac{d}{dx} P(x)$$

If we evaluate the CDF, $P(r)$, at $r = 1.0$, we get:

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

This says *a random variable plucked from our PDF has a 25% chance of being 1 or lower*. We want a function $f(d)$ that takes a uniform distribution between 0 and 1 (*i.e* `f(random_double())`), and returns a random value according to a distribution that has the CDF $P(x) = \frac{x^2}{4}$. We don't know yet know what the function $f(d)$ is analytically, but we do know that 25% of what it returns should be less than 1.0, and 75% should be above 1.0. Likewise, we know that 50% of what it returns should be less than $\sqrt{2}$, and 50% should be above $\sqrt{2}$. If $f(d)$ monotonically increases, then we would expect $f(0.25) = 1.0$ and $f(0.5) = \sqrt{2}$. This can be generalized to figure out $f(d)$ for every possible input:

$$f(P(x)) = x$$

Let's take some more samples:

$$P(0.0) = 0$$

$$P(0.5) = \frac{1}{16}$$

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

$$P(1.5) = \frac{9}{16}$$

$$P(2.0) = 1$$

so, the function $f()$ has values

$$f(P(0.0)) = f(0) = 0$$

$$f(P(0.5)) = f(\frac{1}{16}) = 0.5$$

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

$$f(P(1.5)) = f(\frac{9}{16}) = 1.5$$

$$f(P(2.0)) = f(1) = 2.0$$

We could use these intermediate values and interpolate between them to approximate $f(d)$:

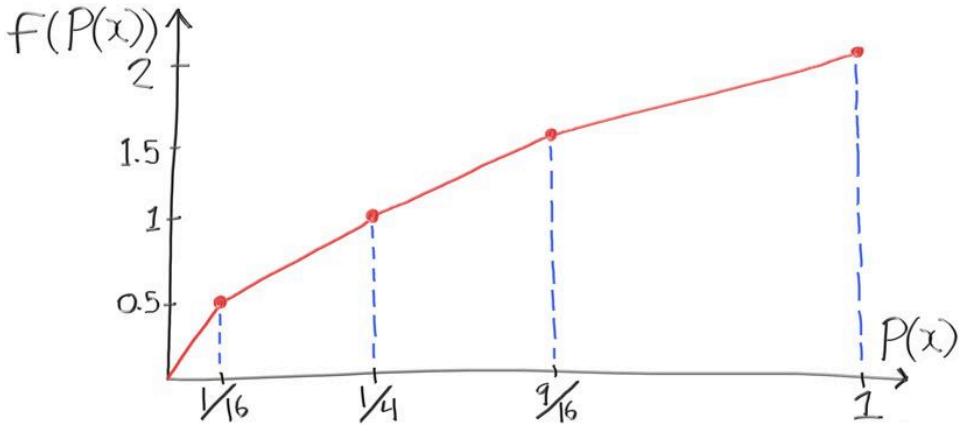

Figure 8: *Approximating the nonuniform f()*

If you can't solve for the PDF analytically, then you can't solve for the CDF analytically. After all, the CDF is just the integral of the PDF. However, you can still create a distribution that approximates the PDF. If you take a bunch of samples from the random function you want the PDF from, you can approximate the PDF by getting a histogram of the samples and then converting to a PDF. Alternatively, you can do as we did above and sort all of your samples.

Looking closer at the equality:

$$f(P(x)) = x$$

That just means that $f()$ just undoes whatever $P()$ does. So, $f()$ is the inverse function:

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

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:

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

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

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

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

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

Which means the inverse of our CDF (which we'll call $ICD(x)$) is defined as

$$P^{-1}(r) = \text{ICD}(r) = \sqrt{4y}$$

Thus our random number generator with density $p(r)$ can be created with:

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

Note that this ranges from 0 to 2 as we hoped, and if we check our work, we replace `random_double()` with $1/4$ to get 1, and also replace with $1/2$ to get $\sqrt{2}$, just as expected.

3.7. Importance Sampling

You should now have a decent understanding of how to take an analytical PDF and generate a function that produces random numbers with that distribution. We return to our original integral and try it with a few different PDFs to get a

better understanding:

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

The last time that we tried to solve for the integral we used a Monte Carlo approach, uniformly sampling from the interval $[0, 2]$. We didn't know it at the time, but we were implicitly using a uniform PDF between 0 and 2. This means that we're using a PDF = $1/2$ over the range $[0, 2]$, which means the CDF is $P(x) = x/2$, so $\text{ICD}(d) = 2d$. Knowing this, we can make this uniform PDF explicit:

```
#include "rtweekend.h"

#include <iostream>
#include <iomanip>

double icd(double d) {
    return 2.0 * d;
}

double pdf(double x) {
    return 0.5;
}

int main() {
    int a = 0;
    int b = 2;
    int N = 1000000;
    auto sum = 0.0;

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

Listing 13: `[integrate_x_sq.cc]` *Explicit uniform PDF for* x^2

There are a couple of important things to emphasize. Every value of x represents one sample of the function x^2 within the distribution $[0, 2]$. We use a function ICD to randomly select samples from within this distribution. We were previously multiplying the average over the interval (`sum / N`) times the length of the interval (`b - a`) to arrive at the final answer. Here, we don't need to multiply by the interval length—that is, we no longer need to multiply the average by 2.

We need to account for the nonuniformity of the PDF of x. Failing to account for this nonuniformity will introduce bias in our scene. Indeed, this bias is the source of our inaccurately bright image. Accounting for the nonuniformity will yield accurate results. The PDF will “steer” samples toward specific parts of the distribution, which will cause us to converge faster, but at the cost of introducing bias. To remove this bias, we need to down-weight where we sample more frequently, and to up-weight where we sample less frequently. For our new nonuniform random number generator, the PDF defines how much or how little we sample a specific portion. So the weighting function should be proportional to $1/pdf$. In fact it is *exactly* $1/pdf$. This is why we divide `x*x` by `pdf(x)`.

We can try to solve for the integral using the linear PDF, $p(r) = \frac{r}{2}$, for which we were able to solve for the CDF and its inverse, ICD. To do that, all we need to do is replace the functions $\text{ICD}(d) = \sqrt{4d}$ and $p(x) = x/2$.

```
double icd(double d) {
    return std::sqrt(4.0 * d);
}

double pdf(double x) {
    return x / 2.0;
}

int main() {
    int N = 1000000;
    auto sum = 0.0;

    for (int i = 0; i < N; i++) {
        auto z = random_double();
        if (z == 0.0)  // Ignore zero to avoid NaNs
            continue;

        auto x = icd(z);
        sum += x*x / pdf(x);
    }

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

Listing 14: `[integrate_x_sq.cc]` *Integrating x^2 with linear PDF*

If you compared the runs from the uniform PDF and the linear PDF, you would have probably found that the linear PDF converged faster. If you think about it, a linear PDF is probably a better approximation for a quadratic function than a uniform PDF, so you would expect it to converge faster. If that's the case, then we should just try to make the PDF match the integrand by turning the PDF into a quadratic function:

$$p(r) = \begin{cases} 0 & r < 0 \\ C \cdot r^2 & 0 \leq r \leq 2 \\ 0 & 2 < r \end{cases}$$

Like the linear PDF, we'll solve for the constant C by integrating to 1 over the interval:

$$1 = \int_0^2 C \cdot r^2 dr$$

$$= C \cdot \int_0^2 r^2 dr$$

$$= C \cdot \frac{r^3}{3}\Big|_0^2$$

$$= C(\frac{2^3}{3} - \frac{0}{3})$$

$$C = \frac{3}{8}$$

Which gives us:

$$p(r) = \begin{cases} 0 & r < 0 \\ \frac{3}{8}r^2 & 0 \leq r \leq 2 \\ 0 & 2 < r \end{cases}$$

And we get the corresponding CDF:

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

and

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

For just one sample we get:

```
double icd(double d) {
    return 8.0 * std::pow(d, 1.0/3.0);
}

double pdf(double x) {
    return (3.0/8.0) * x*x;
}

int main() {
    int N = 1;
    auto sum = 0.0;

    for (int i = 0; i < N; i++) {
        auto z = random_double();
        if (z == 0.0)  // Ignore zero to avoid NaNs
            continue;

        auto x = icd(z);
        sum += x*x / pdf(x);
    }
    std::cout << std::fixed << std::setprecision(12);
    std::cout << "I = " << sum / N << '\n';
}
```

Listing 15: `[integrate_x_sq.cc]` *Integrating x^2, final version*

This always returns the exact answer.

A nonuniform PDF "steers" more samples to where the PDF is big, and fewer samples to where the PDF is small. By this sampling, we would expect less noise in the places where the PDF is big and more noise where the PDF is small. If we choose a PDF that is higher in the parts of the scene that have higher noise, and is smaller in the parts of the scene that have lower noise, we'll be able to reduce the total noise of the scene with fewer samples. This means that we will be able to converge to the correct scene *faster* than with a uniform PDF. In effect, we are steering our samples toward the parts of the distribution that are more *important*. This is why using a carefully chosen nonuniform PDF is usually called *importance sampling*.

In all of the examples given, we always converged to the correct answer of $8/3$. We got the same answer when we used both a uniform PDF and the "correct" PDF (that is, $\text{ICD}(d) = 8d^{\frac{1}{3}}$). While they both converged to the same answer, the uniform PDF took much longer. After all, we only needed a single sample from the PDF that perfectly matched the integral. This should make sense, as we were choosing to sample the important parts of the distribution more often, whereas the uniform PDF just sampled the whole distribution equally, without taking importance into account.

Indeed, this is the case for any PDF that you create—they will all converge eventually. This is just another part of the power of the Monte Carlo algorithm. Even the naive PDF where we solved for the 50% value and split the distribution into two halves: $[0, \sqrt{2}]$ and $[\sqrt{2}, 2]$. That PDF will converge. Hopefully you should have an intuition as to why that PDF will converge faster than a pure uniform PDF, but slower than the linear PDF (that is, $\text{ICD}(d) = \sqrt{4d}$).

The 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.

Let's review the main concepts that underlie Monte Carlo 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 and non-negative 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. Monte Carlo Integration on the Sphere of Directions

In chapter One Dimensional Monte Carlo Integration we started with uniform random numbers and slowly, over the course of a chapter, built up more and more complicated ways of producing random numbers, before ultimately arriving at the intuition of PDFs, and how to use them to generate random numbers of arbitrary distribution.

All of the concepts covered in that chapter continue to work as we extend beyond a single dimension. Moving forward, we might need to be able to select a point from a two, three, or even higher dimensional space and then weight that selection by an arbitrary PDF. An important case of this—at least for ray tracing—is producing a random direction. In the first two books we generated a random direction by creating a random vector and rejecting it if it fell outside of the unit sphere. We repeated this process until we found a random vector that fell inside the unit sphere. Normalizing this vector produced points that lay exactly on the unit sphere and thereby represent a random direction. This process of generating samples and rejecting them if they are not inside a desired space is called *the rejection method*, and is found all over the literature. The method covered in the last chapter is referred to as *the inversion method* because we invert a PDF.

Every direction in 3D space has an associated point on the unit sphere and can be generated by solving for the vector that travels from the origin to that associated point. You can think of choosing a random direction as choosing a random point in a constrained two dimensional plane: the plane created by mapping the unit sphere to Cartesian coordinates. The same methodology as before applies, but now we might have a PDF defined over two dimensions. Suppose we want to integrate this function over the surface of the unit sphere:

$$f(\theta, \phi) = \cos^2(\theta)$$

Using Monte Carlo integration, we should just be able to sample $\cos^2(\theta)/p(r)$, where the $p(r)$ is now just $p(direction)$. But what is *direction* in that context? We could make it based on polar coordinates, so p would be in terms of θ and ϕ for $p(\theta, \phi)$. It doesn't matter which coordinate system you choose to use. Although, however you choose to do it, remember that a PDF must integrate to one over the whole surface and that the PDF represents the *relative probability* of that direction being sampled. Recall that we have a `vec3` function to generate uniform random samples on the unit sphere d (`random_unit_vector()`). What is the PDF of these uniform samples? As a uniform density on the unit sphere, it is $1/area$ of the sphere, which is $1/(4\pi)$. If the integrand is $\cos^2(\theta)$, and θ is the angle with the z axis, we can use scalar projection to re-write $\cos^2(\theta)$ in terms of the d_z:

$$d_z = \|d\| \cos\theta = 1 \cdot \cos\theta$$

We can then substitute $1 \cdot \cos\theta$ with d_z giving us:

$$f(\theta, \phi) = \cos^2(\theta) = {d_z}^2$$

```
#include "rtweekend.h"

#include <iostream>
#include <iomanip>

double f(const vec3& d) {
    auto cosine_squared = d.z()*d.z();
    return cosine_squared;
}

double pdf(const vec3& d) {
    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 f_d = f(d);
        sum += f_d / pdf(d);
    }
    std::cout << std::fixed << std::setprecision(12);
    std::cout << "I = " << sum / N << '\n';
}
```

Listing 16: `[sphere_importance.cc]` *Generating importance-sampled points on the unit sphere*

The analytic answer is $\frac{4}{3}\pi = 4.188790204786391$ — if you remember enough advanced calc, check me! And the code above produces that. The key point here is that all of the integrals and the probability and everything else is over the unit sphere. The way to represent a single direction in 3D is its associated point on the unit sphere. The way to represent a range of directions in 3D is the amount of area on the unit sphere that those directions travel through. Call it direction, area, or *solid angle* — it's all the same thing. Solid angle is the term that you'll usually find in the literature. You have radians (r) in θ over one dimension, and you have *steradians* (sr) in θ *and* ϕ over two dimensions (the unit sphere is a three dimensional object, but its surface is only two dimensional). Solid Angle is just the two dimensional extension of angles. If you are comfortable with a two dimensional angle, 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.

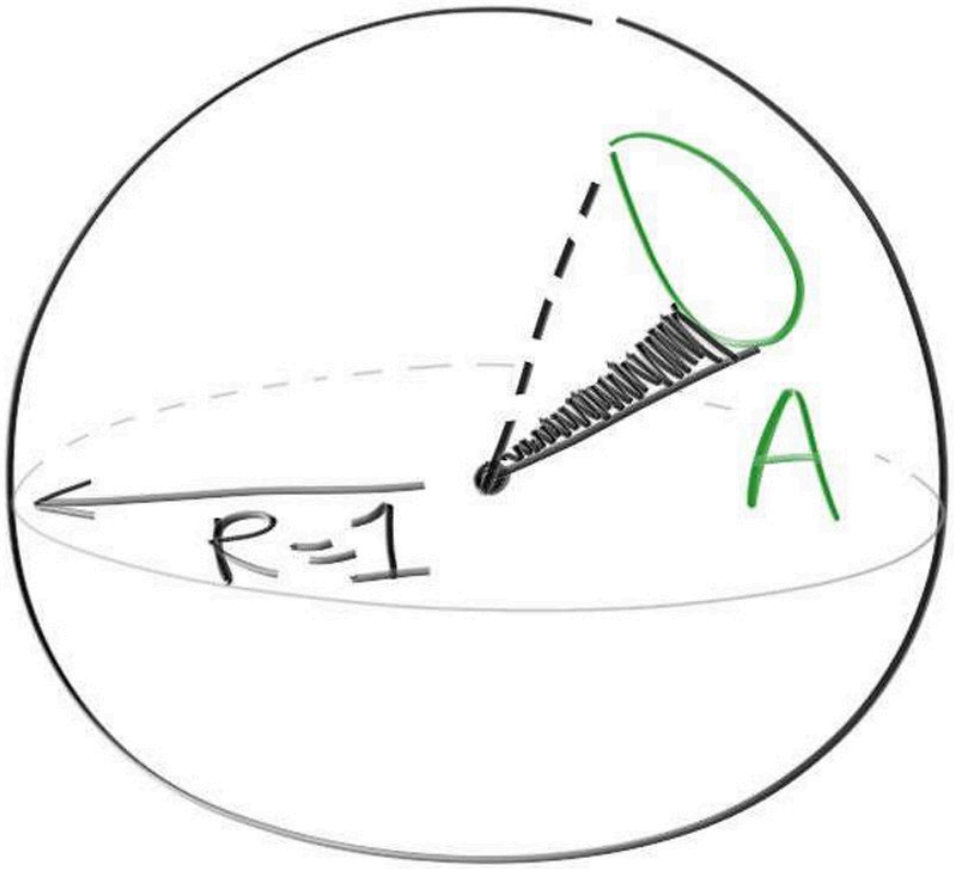

Figure 9: *Solid angle / projected area of a sphere*

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'll just be setting up for a big lighting change in the next chapter. Our ray tracing program from the first two books scatters a ray when it interacts with a surface or a volume. Ray scattering is the most commonly used model for simulating light propagation through a scene. This can naturally be modeled probabilistically. There are many things to consider when modeling the probabilistic scattering of rays.

5.1. Albedo

First, is the light absorbed?

Probability of light being scattered: A

Probability of light being absorbed: $1 - A$

Where here A stands for *albedo*. As covered in our first book, recall that albedo is a form of fractional reflectance. It can help to stop and remember that when we simulate light propagation, all we're doing is simulating the movement of photons through a space. If you remember your high school Physics then you should recall that every photon has a unique energy and wavelength associated by the Planck constant:

$$E = \frac{hc}{\lambda}$$

Each individual photon has a *tiny* amount of energy, but when you add enough of them up you get all of the illumination in your rendering. The absorption or scattering of a photon with a surface or a volume (or really anything that a photon can interact with) is probabilistically determined by the albedo of the object. Albedo can depend on color because some objects are more likely to absorb some wavelengths.

In most physically based renderers, we would use a predefined set of specific wavelengths for the light color rather than RGB. As an example, we would replace our *tristimulus* RGB renderer with something that specifically samples at 300nm, 350nm, 400nm, ..., 700nm. We can extend our intuition by thinking of R, G, and B as specific algebraic mixtures of wavelengths where R is *mostly* red wavelengths, G is *mostly* green wavelengths, and B is *mostly* blue wavelengths. This is an approximation of the human visual system which has 3 unique sets of color receptors, called *cones*, that are each sensitive to different algebraic mixtures of wavelengths, roughly RGB, but are referred to as long, medium, and short cones (the names are in reference to the wavelengths that each cone is sensitive to, not the length of the cone). Just as colors can be represented by their strength in the RGB color space, colors can also be represented by how excited each set of cones is in the *LMS color space* (long, medium, short).

5.2. Scattering

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*: $\text{pScatter}()$. The scattering PDF will vary with outgoing direction: $\text{pScatter}(\omega_o)$. The scattering PDF can also vary with *incident direction*: $\text{pScatter}(\omega_i, \omega_o)$. 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 scattering PDF can vary with the wavelength of the light: $\text{pScatter}(\omega_i, \omega_o, \lambda)$. A good example of this is a prism refracting white light into a rainbow. Lastly, the scattering PDF can also depend on the scattering position: $\text{pScatter}(\mathbf{x}, \omega_i, \omega_o, \lambda)$. The $\mathbf{x}$ is just math notation for the scattering position: $\mathbf{x} = (x, y, z)$. The albedo of an object can also depend on these quantities: $A(\mathbf{x}, \omega_i, \omega_o, \lambda)$.

The color of a surface is found by integrating these terms over the unit hemisphere by the incident direction:

$$\text{Color}_o(\mathbf{x}, \omega_o, \lambda) = \int_{\omega_i} A(\mathbf{x}, \omega_i, \omega_o, \lambda) \cdot \text{pScatter}(\mathbf{x}, \omega_i, \omega_o, \lambda) \cdot \text{Color}_i(\mathbf{x}, \omega_i, \lambda)$$

We've added a Color_i term. The scattering PDF and the albedo at the surface of an object are acting as filters to the light that is shining on that point. So we need to solve for the light that is shining on that point. This is a recursive algorithm, and is the reason our `ray_color` function returns the color of the current object multiplied by the color of the next ray.

5.3. The Scattering PDF

If we apply the Monte Carlo basic formula we get the following statistical estimate:

$$\text{Color}_o(\mathbf{x}, \omega_o, \lambda) \approx \sum \frac{A(\dots) \cdot \text{pScatter}(\dots) \cdot \text{Color}_i(\dots)}{p(\mathbf{x}, \omega_i, \omega_o, \lambda)}$$

where $p(\mathbf{x}, \omega_i, \omega_o, \lambda)$ is the PDF of whatever outgoing direction we randomly generate.

For a Lambertian surface we already implicitly implemented this formula for the special case where $pScatter(\dots)$ is a cosine density. The $\text{pScatter}(\dots)$ of a Lambertian surface is proportional to $\cos(\theta_o)$, where θ_o is the angle relative to the surface normal ($\theta_o \in [0, \pi]$). An angle of 0 indicates an outgoing direction in the same direction as the surface normal, and an angle of π indicates an outgoing direction exactly opposite the normal vector.

Let's solve for C once more:

$$\text{pScatter}(\mathbf{x}, \omega_i, \omega_o, \lambda) = C \cdot \cos(\theta_o)$$

All two dimensional PDFs need to integrate to one over the whole surface (remember that pScatter is a PDF). We set $\text{pScatter}(\frac{\pi}{2} < \theta_o \leq \pi) = 0$ so that we don't scatter below the horizon. Given this, we only need to integrate $\theta \in [0, \frac{\pi}{2}]$.

$$1 = \int_{\phi=0}^{2\pi} \int_{\theta=0}^{\frac{\pi}{2}} C \cdot \cos(\theta) dA$$

To integrate over the hemisphere, remember that in spherical coordinates:

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

So:

$$1 = C \cdot \int_{0}^{2\pi} \int_{0}^{\frac{\pi}{2}} \cos(\theta) \sin(\theta) d\theta d\phi$$

$$1 = C \cdot 2\pi \frac{1}{2}$$

$$1 = C \cdot \pi$$

$$C = \frac{1}{\pi}$$

The integral of $\cos(\theta_o)$ over the hemisphere is π, so we need to normalize by $\frac{1}{\pi}$. The PDF pScatter is only dependent on outgoing direction (ω_o), so we'll simplify its representation to just $\text{pScatter}(\omega_o)$. Put all of this together and you get the scattering PDF for a Lambertian surface:

$$\text{pScatter}(\omega_o) = \frac{\cos(\theta_o)}{\pi}$$

We'll assume that the $p(\mathbf{x}, \omega_i, \omega_o, \lambda)$ is equal to the scattering PDF:

$$p(\omega_o) = \text{pScatter}(\omega_o) = \frac{\cos(\theta_o)}{\pi}$$

The numerator and denominator cancel out, and we get:

$$\text{Color}_o(\mathbf{x}, \omega_o, \lambda) \approx \sum A(\dots) \cdot \text{Color}_i(\dots)$$

This is exactly what we had in our original `ray_color()` function!

```
    return attenuation * ray_color(scattered, depth-1, world);
```

The treatment above is slightly non-standard because I want the same math to work for surfaces and volumes. If you read the literature, you'll see reflection defined by the *Bidirectional Reflectance Distribution Function* (BRDF). It relates pretty simply to our terms:

$$BRDF(\omega_i, \omega_o, \lambda) = \frac{A(\mathbf{x}, \omega_i, \omega_o, \lambda) \cdot \text{pScatter}(\mathbf{x}, \omega_i, \omega_o, \lambda)}{\cos(\theta_o)}$$

So for a Lambertian surface for example, $BRDF = A/\pi$. Translation between our terms and BRDF is easy. For participating media (volumes), our albedo is usually called the *scattering albedo*, and our scattering PDF is usually called the *phase function*.

All that we've done here is outline the PDF for the Lambertian scattering of a material. However, we'll need to generalize so that we can send extra rays in important directions, such as toward the lights.

6. Playing with Importance Sampling

Our goal over the next several 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 $\text{pLight}(\omega_o)$. Let's also assume we have a PDF related to pScatter, and let's call that $\text{pSurface}(\omega_o)$. 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:

$$p(\omega_o) = \frac{1}{2}\text{pSurface}(\omega_o) + \frac{1}{2}\text{pLight}(\omega_o)$$

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

$$\text{pScatter}(\mathbf{x}, \omega_i, \omega_o) \cdot \text{Color}_i(\mathbf{x}, \omega_i)$$

is largest. For diffuse surfaces, this is mainly a matter of guessing where $\text{Color}_i(\mathbf{x}, \omega_i)$ is largest. Which is equivalent to guessing where the most light is coming from.

For a mirror, $\text{pScatter}()$ is huge only near one direction, so $\text{pScatter}()$ matters a lot more. In fact, most renderers just make mirrors a special case, and make the $\text{pScatter}()/p()$ implicit — our code currently does that.

6.1. Returning to the Cornell Box

Let's adjust some parameters for the Cornell box:

```
int main() {
    ...
    cam.samples_per_pixel = 1000;
    ...
}
```

Listing 17: `[main.cc]` *Cornell box, refactored*

At 600×600 my code produces this image in 15min on 1 core of my Macbook:

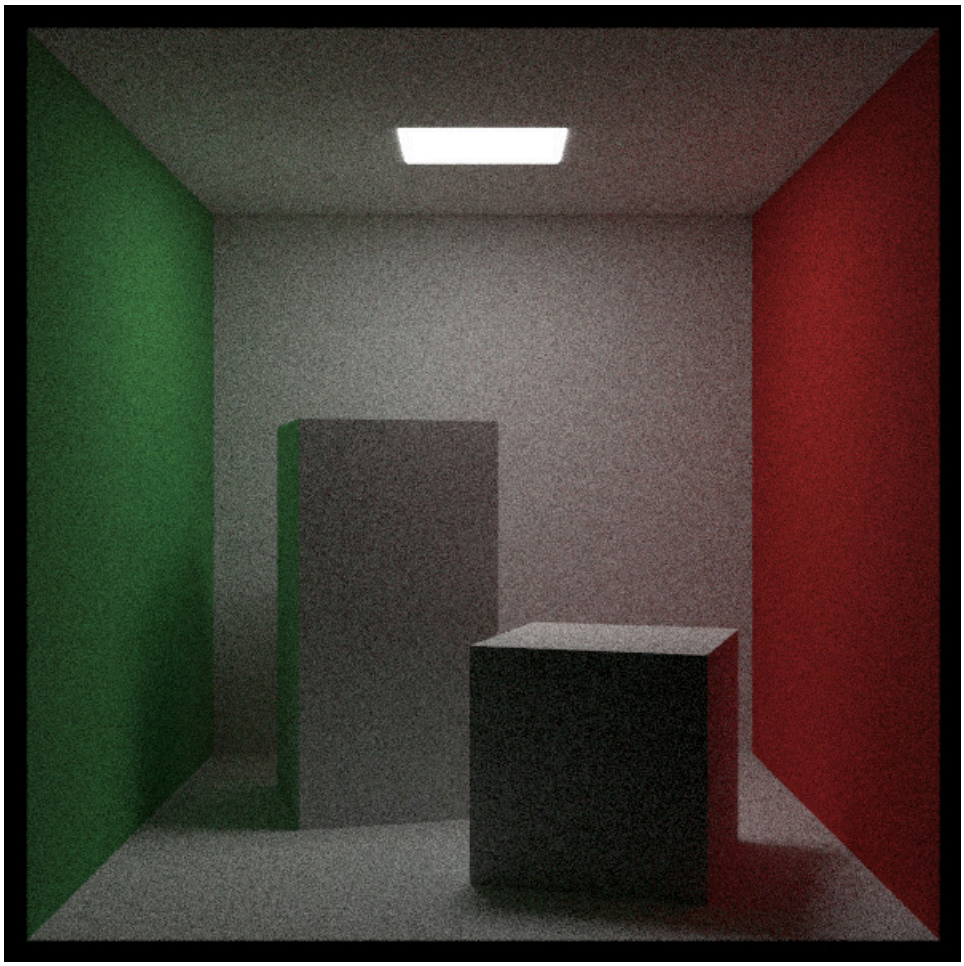

Image 3: *Cornell box, refactored*

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 Monte Carlo basics: $\int f(x) \approx \sum f(r)/p(r)$. For the Lambertian material, let's sample like we do now: $p(\omega_o) = \cos(\theta_o)/\pi$.

We modify the base-class `material` to enable this importance sampling, and define the scattering PDF function for Lambertian materials:

```
class material {
  public:
    ...

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

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

    bool scatter(const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered)
    const override {
        ...
    }

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

  private:
    shared_ptr<texture> tex;
};
```

Listing 18: `[material.h]` *Lambertian material, modified for importance sampling*

And the `camera::ray_color` function gets a minor modification:

```
class camera {
  ...
  private:
    ...
    color ray_color(const ray& r, int depth, const hittable& world) const {
        // If we've exceeded the ray bounce limit, no more light is gathered.
        if (depth <= 0)
            return color(0,0,0);

        hit_record rec;

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

        ray scattered;
        color attenuation;
        color color_from_emission = rec.mat->emitted(rec.u, rec.v, rec.p);

        if (!rec.mat->scatter(r, rec, attenuation, scattered))
            return color_from_emission;

        double scattering_pdf = rec.mat->scattering_pdf(r, rec, scattered);
        double pdf_value = scattering_pdf;

        color color_from_scatter =
            (attenuation * scattering_pdf * ray_color(scattered, depth-1, world)) / pdf_value;

        return color_from_emission + color_from_scatter;
    }
};
```

Listing 19: `[camera.h]` *The ray_color function, modified for importance sampling*

You should get exactly the same picture. Which *should make sense*, as the scattered part of `ray_color` is getting multiplied by `scattering_pdf / pdf_value`, and as `pdf_value` is equal to `scattering_pdf` is just the same as multiplying by one.

6.2. Using a Uniform PDF Instead of a Perfect Match

Now, just for the experience, let's try using a different sampling PDF. We'll continue to have our reflected rays weighted by Lambertian, so $\cos(\theta_o)$, and we'll keep the scattering PDF as is, but we'll use a different PDF in the denominator. We will sample using a uniform PDF about the hemisphere, so we'll set the denominator to $1/2\pi$. This will still converge on the correct answer, as all we've done is change the PDF, but since the PDF is now less of a perfect match for the real distribution, it will take longer to converge. Which, for the same number of samples means a noisier image:

```
class camera {
  ...
  private:
    ...
    color ray_color(const ray& r, int depth, const hittable& world) const {
        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, interval(0.001, infinity), rec))
            return background;

        ray scattered;
        color attenuation;
        color color_from_emission = rec.mat->emitted(rec.u, rec.v, rec.p);

        if (!rec.mat->scatter(r, rec, attenuation, scattered))
            return color_from_emission;

        double scattering_pdf = rec.mat->scattering_pdf(r, rec, scattered);
        double pdf_value = 1 / (2*pi);

        color color_from_scatter =
            (attenuation * scattering_pdf * ray_color(scattered, depth-1, world)) / pdf_value;

        return color_from_emission + color_from_scatter;
    }
```

Listing 20: `[camera.h]` *The ray_color function, now with a uniform PDF in the denominator*

You should get a very similar result to before, only with slightly more noise, it may be hard to see.

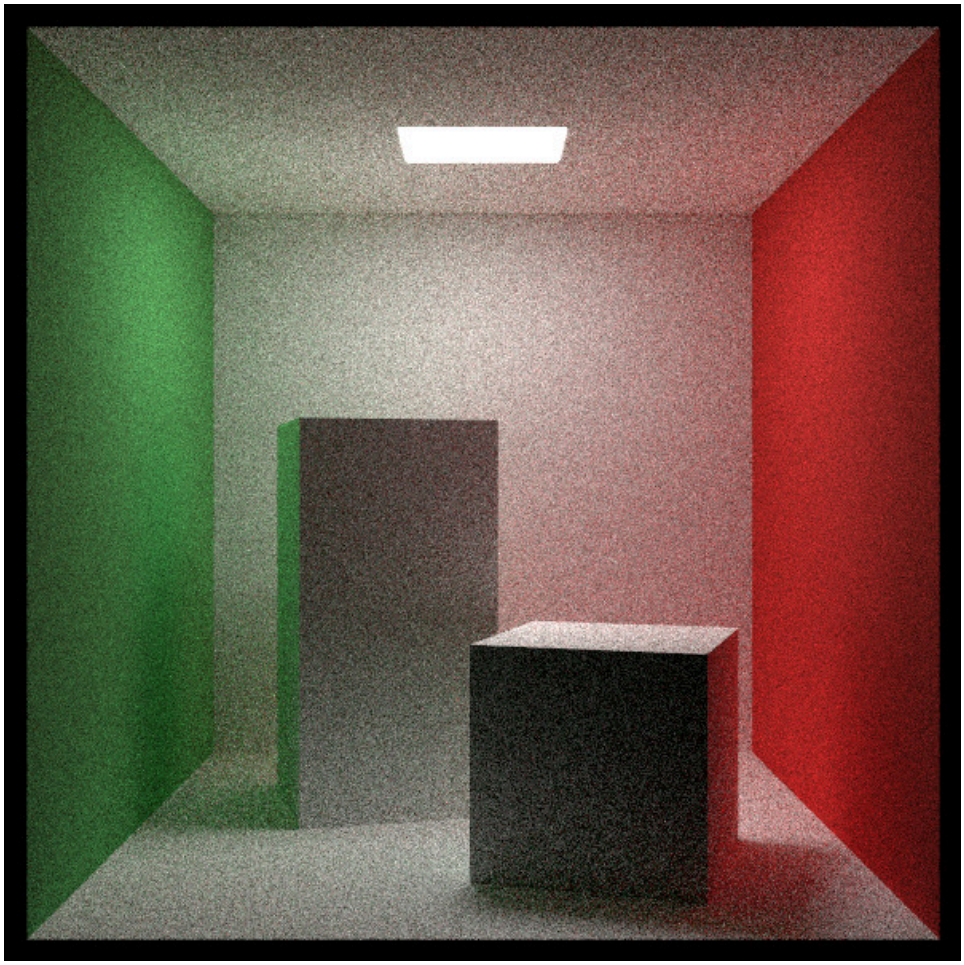

Image 4: *Cornell box, with imperfect PDF*

Make sure to return the PDF to the scattering PDF.

```
        ...

        double scattering_pdf = rec.mat->scattering_pdf(r, rec, scattered);
        double pdf_value = scattering_pdf;

        ...
```

Listing 21: `[camera.h]` *Return the PDF to the same as scattering PDF*

6.3. Random Hemispherical Sampling

To confirm our understanding, let's try a different scattering distribution. For this one, we'll attempt to repeat the uniform hemispherical scattering from the first book. There's nothing wrong with this technique, but we are no longer treating our objects as Lambertian. Lambertian is a specific type of diffuse material that requires a $\cos(\theta_o)$ scattering distribution. Uniform hemispherical scattering is a different diffuse material. If we keep the material the same but change the PDF, as we did in last section, we will still converge on the same answer, but our convergence may take more or less samples. However, if we change the material, we will have fundamentally changed the render and the algorithm will converge on a different answer. So when we replace Lambertian diffuse with uniform hemispherical diffuse we should expect the outcome of our render to be *materially* different. We're going to adjust our scattering direction and scattering PDF:

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

    bool scatter(const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered)
    const override {
        auto scatter_direction = random_on_hemisphere(rec.normal);

        // Catch degenerate scatter direction
        if (scatter_direction.near_zero())
            scatter_direction = rec.normal;

        scattered = ray(rec.p, scatter_direction, r_in.time());
        attenuation = tex->value(rec.u, rec.v, rec.p);
        return true;
    }

    double scattering_pdf(const ray& r_in, const hit_record& rec, const ray& scattered)
    const override {
        return 1 / (2*pi);
    }

    ...
```

Listing 22: `[material.h]` *Modified PDF and scatter function*

This new diffuse material is actually just $p(\omega_o) = \frac{1}{2\pi}$ for the scattering PDF. So our uniform PDF that was an imperfect match for Lambertian diffuse is actually a perfect match for our uniform hemispherical diffuse. When rendering, we should get a slightly different image.

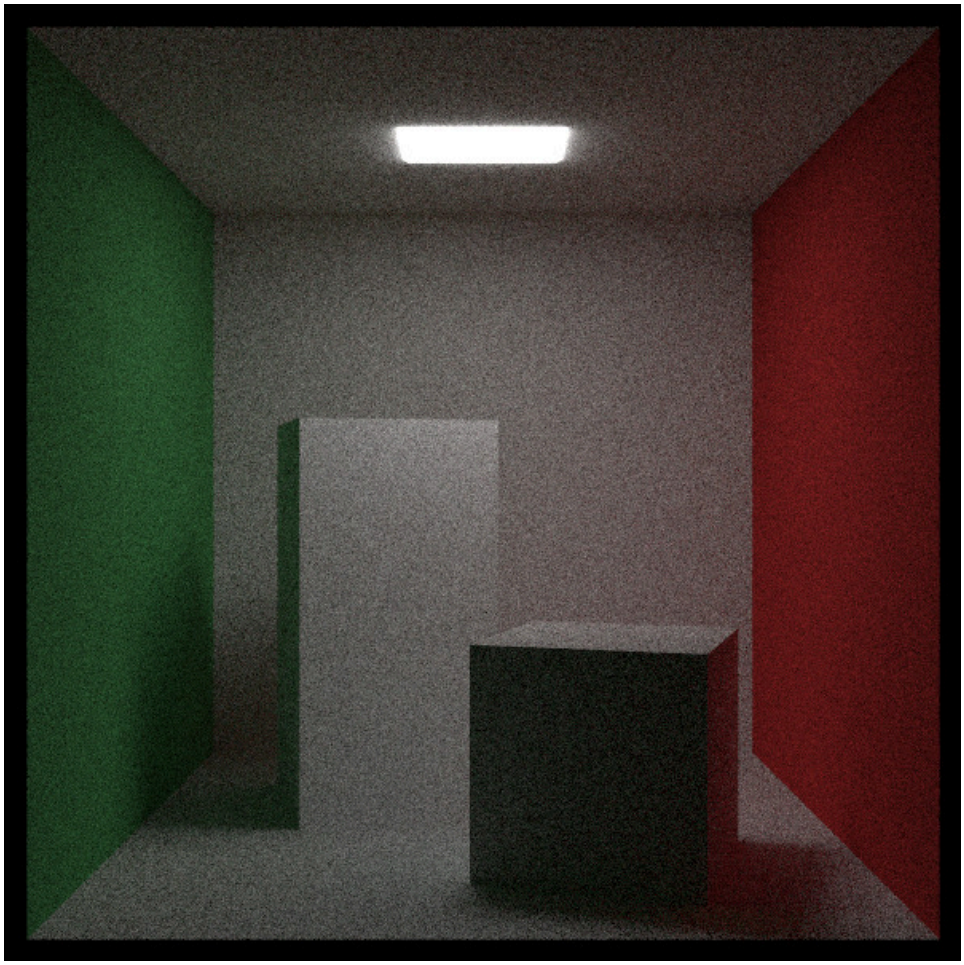

Image 5: *Cornell box, with uniform hemispherical sampling*

It’s pretty close to our old picture, but there are differences that are not just noise. The front of the tall box is much more uniform in color. If you aren't sure what the best sampling pattern for your material is, it's pretty reasonable to just go ahead and assume a uniform PDF, and while that might converge slowly, it's not going to ruin your render. That said, if you're not sure what the correct sampling pattern for your material is, your choice of PDF is not going to be your biggest concern, as incorrectly choosing your scattering function *will* ruin your render. At the very least it will produce an incorrect result. You may find yourself with the most difficult kind of bug to find in a Monte Carlo program — a bug that produces a reasonable looking image! You won’t know if the bug is in 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, we'll harden our understanding and our tools.

7.1. Random Directions Relative to the Z Axis

Let’s first figure out how to generate random directions. We already have a method to generate random directions using the rejection method, so let's create one using the inversion method. To simplify things, assume the z axis is the surface normal, and θ is the angle from the normal. We'll set everything up in terms of the z axis this chapter. Next chapter we’ll

get them oriented to the surface normal vector. We will only deal with distributions that are rotationally symmetric about z. So $p(\omega) = f(\theta)$.

Given a directional PDF on the sphere (where $p(\omega) = f(\theta)$), the one dimensional PDFs on θ and ϕ are:

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

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

For uniform random numbers r_1 and r_2, we solve for the CDF of θ and ϕ so that we can invert the CDF to derive the random number generator.

$$r_1 = \int_0^\phi a(\phi')d\phi'$$

$$= \int_0^\phi \frac{1}{2\pi}d\phi'$$

$$= \frac{\phi}{2\pi}$$

Invert to solve for ϕ:

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

This should match with your intuition. To solve for a random ϕ you can take a uniform random number in the interval [0,1] and multiply by 2π to cover the full range of all possible ϕ values, which is just [0,2π]. You may not have a fully formed intuition for how to solve for a random value of θ, so let's walk through the math to help you get set up. We rewrite ϕ as ϕ' and θ as θ' just like before, as a formality. For θ we have:

$$r_2 = \int_0^\theta b(\theta')d\theta'$$

$$= \int_0^\theta 2\pi f(\theta')\sin(\theta')d\theta'$$

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 $p(\omega) = \frac{1}{4\pi}$ on the unit sphere.

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

$$= \int_0^\theta \frac{1}{2}\sin(\theta')d\theta'$$

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

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

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 $\cos^2 + \sin^2 = 1$, 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:

```
#include "rtweekend.h"

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

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

Listing 23: [sphere_plot.cc] *Random points on the unit sphere*

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

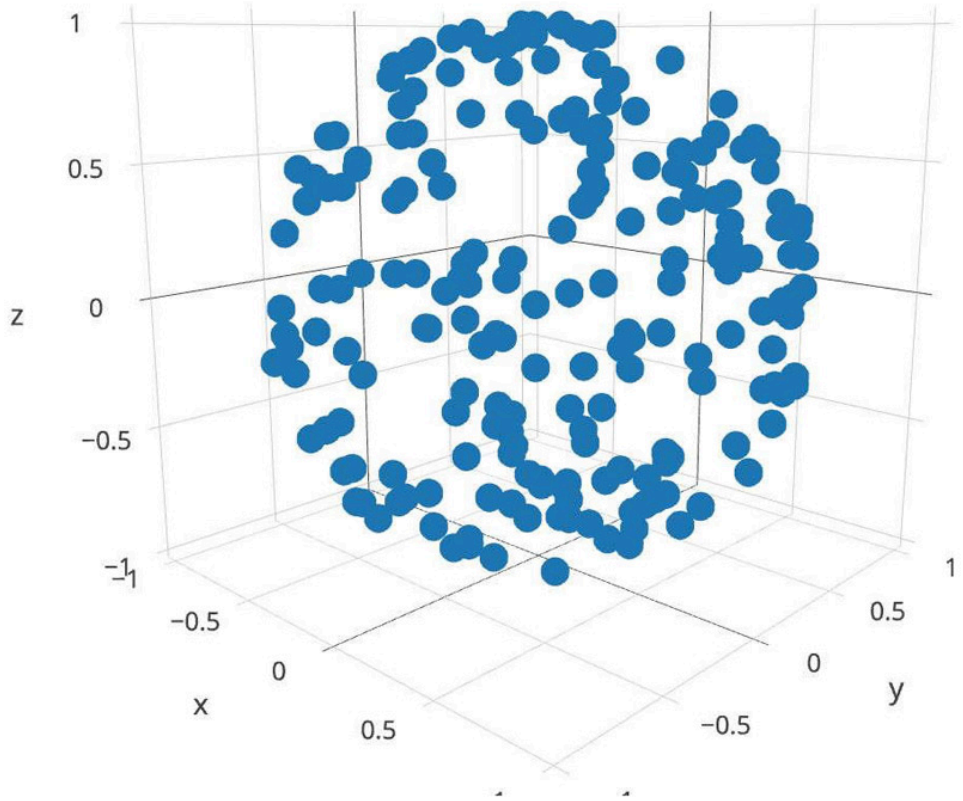

Figure 10: *Random points on the unit sphere*

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 $p(\omega) = f(\theta) = \frac{1}{2\pi}$. Just changing the constant in the theta equations yields:

$$r_2 = \int_0^\theta b(\theta')d\theta'$$

$$= \int_0^\theta 2\pi f(\theta')\sin(\theta')d\theta'$$

$$= \int_0^\theta 2\pi \frac{1}{2\pi}\sin(\theta')d\theta'$$

$$\dots$$

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

This means that $\cos(\theta)$ will vary from 1 to 0, so θ will vary from 0 to $\pi/2$, which means that nothing will go below the horizon. Rather than plot it, we'll solve for a 2D integral with a known solution. Let's integrate cosine cubed over the hemisphere (just picking something arbitrary with a known solution). First we'll solve the integral by hand:

$$\int_\omega \cos^3(\theta)dA$$

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

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

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

```
#include "rtweekend.h"

#include <iostream>
#include <iomanip>

double f(double r2) {
    // auto x = std::cos(2*pi*r1) * 2 * std::sqrt(r2*(1-r2));
    // auto y = std::sin(2*pi*r1) * 2 * std::sqrt(r2*(1-r2));
    auto z = 1 - r2;
    double cos_theta = z;
    return cos_theta*cos_theta*cos_theta;
}

double pdf() {
    return 1.0 / (2.0*pi);
}

int main() {
    int N = 1000000;

    auto sum = 0.0;
    for (int i = 0; i < N; i++) {
        auto r2 = random_double();
        sum += f(r2) / pdf();
    }

    std::cout << std::fixed << std::setprecision(12);
    std::cout << "PI/2 = " << pi / 2.0 << '\n';
    std::cout << "Estimate = " << sum / N << '\n';
}
```

Listing 24: [cos_cubed.cc] *Integration using* $\cos^3(x)$

7.3. Cosine Sampling a Hemisphere

We'll now continue trying to solve for cosine cubed over the horizon, but we'll change our PDF to generate directions with $p(\omega) = f(\theta) = \cos(\theta)/\pi$.

$$r_2 = \int_0^{\theta} b(\theta') d\theta'$$

$$= \int_0^{\theta} 2\pi f(\theta') \sin(\theta') d\theta'$$

$$= \int_0^{\theta} 2\pi \frac{\cos(\theta')}{\pi} \sin(\theta') d\theta'$$

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

So,

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

We can save a little algebra on specific cases by noting

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

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

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

Here's a function that generates random vectors weighted by this PDF:

```
inline vec3 random_cosine_direction() {
    auto r1 = random_double();
    auto r2 = random_double();

    auto phi = 2*pi*r1;
    auto x = std::cos(phi) * std::sqrt(r2);
    auto y = std::sin(phi) * std::sqrt(r2);
    auto z = std::sqrt(1-r2);

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

Listing 25: `[vec3.h]` *Random cosine direction utility function*

```
#include "rtweekend.h"

#include <iostream>
#include <iomanip>

double f(const vec3& d) {
    auto cos_theta = d.z();
    return cos_theta*cos_theta*cos_theta;
}

double pdf(const vec3& d) {
    return d.z() / pi;
}

int main() {
    int N = 1000000;

    auto sum = 0.0;
    for (int i = 0; i < N; i++) {
        vec3 d = random_cosine_direction();
        sum += f(d) / pdf(d);
    }

    std::cout << std::fixed << std::setprecision(12);
    std::cout << "PI/2 = " << pi / 2.0 << '\n';
    std::cout << "Estimate = " << sum / N << '\n';
}
```

Listing 26: `[cos_density.cc]` *Integration with cosine density function*

We can generate other densities later as we need them. This `random_cosine_direction()` function produces a random direction weighted by $\cos(\theta)$ where θ is the angle from the z axis.

8. Orthonormal Bases

In the last chapter we developed methods to generate random directions relative to the z axis. If we want to be able to produce reflections off of any surface, we are going to need to make this more general: Not all normals are going to be perfectly aligned with the z axis. So in this chapter we are going to generalize our methods so that they support arbitrary surface normal vectors.

8.1. Relative Coordinates

An *orthonormal basis* (ONB) is a collection of three mutually orthogonal unit vectors. It is a strict subtype of coordinate system. The Cartesian xyz axes are one example of an orthonormal basis. All of our renders are the result of the relative positions and orientations of the objects in a scene projected onto the image plane of the camera. The camera and objects must be described in the same coordinate system, so that the projection onto the image plane is logically defined, otherwise the camera has no definitive means of correctly rendering the objects. Either the camera must be redefined in the objects' coordinate system, or the objects must be redefined in the camera's coordinate system. It's best to start with both in the same coordinate system, so no redefinition is necessary. So long as the camera and scene are described in the same coordinate system, all is well. The orthonormal basis defines how distances and orientations are represented in the space, but an orthonormal basis alone is not enough. The objects and the camera need to described by their displacement from a mutually defined location. This is just the origin $\mathbf{O}$ of the scene; it represents the center of the universe for everything to displace from.

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 to graphics course, there will be a lot of time spent on coordinate systems and 4×4 coordinate transformation matrices. Pay attention, it's really important stuff! But we won't be needing it for this book and we'll make do without it. What we do need is to generate random directions with a set distribution relative to the surface normal vector $\mathbf{n}$. We won't be needing an origin for this because a direction is relative and has no specific origin. To start off with, we need two cotangent vectors that are each perpendicular to $\mathbf{n}$ and that are also perpendicular to each other.

Some 3D object models will come with one or more cotangent vectors for each vertex. 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 nonparallel with $\mathbf{n}$. We can get vectors $\mathbf{s}$ and $\mathbf{t}$ perpendicular to $\mathbf{n}$ by using the property of the cross product that $\mathbf{n} \times \mathbf{a}$ is perpendicular to both $\mathbf{n}$ and $\mathbf{a}$:

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

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

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 our current program doesn't have a way to generate one. If we went ahead and picked an arbitrary $\mathbf{a}$ to use as an initial vector we may get an $\mathbf{a}$ that is parallel to $\mathbf{n}$. So a common method is to pick an arbitrary axis and check to see if it's parallel to $\mathbf{n}$ (which we assume to be of unit length), if it is, just use another axis:

```
if (std::fabs(n.x()) > 0.9)
    a = vec3(0, 1, 0)
else
    a = vec3(1, 0, 0)
```

We then take the cross product to get $\mathbf{s}$ and $\mathbf{t}$

```
vec3 s = unit_vector(cross(n, a));
vec3 t = cross(n, s);
```

Note that we don't need to take the unit vector for $\mathbf{t}$. Since $\mathbf{n}$ and $\mathbf{s}$ are both unit vectors, their cross product $\mathbf{t}$ will be also. 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 $\mathbf{n}$ with:

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

If you remember, we used similar math to produce rays from a camera. You can think of that 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(const vec3& n) {
        axis[2] = unit_vector(n);
        vec3 a = (std::fabs(axis[2].x()) > 0.9) ? vec3(0,1,0) : vec3(1,0,0);
        axis[1] = unit_vector(cross(axis[2], a));
        axis[0] = cross(axis[2], axis[1]);
    }

    const vec3& u() const { return axis[0]; }
    const vec3& v() const { return axis[1]; }
    const vec3& w() const { return axis[2]; }

    vec3 transform(const vec3& v) const {
        // Transform from basis coordinates to local space.
        return (v[0] * axis[0]) + (v[1] * axis[1]) + (v[2] * axis[2]);
    }

  private:
    vec3 axis[3];
};


#endif
```

Listing 27: [onb.h] *Orthonormal basis class*

We can rewrite our Lambertian material using this to get:

```
#include "hittable.h"
#include "onb.h"
#include "texture.h"

class material {
  public:
    ...

    virtual bool scatter(
        const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered, double& pdf
    ) const {
        return false;
    }

    ...
};

class lambertian : public material {
  public:
    ...

    bool scatter(
        const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered, double& pdf
    ) const override {
        onb uvw(rec.normal);
        auto scatter_direction = uvw.transform(random_cosine_direction());

        scattered = ray(rec.p, unit_vector(scatter_direction), r_in.time());
        attenuation = tex->value(rec.u, rec.v, rec.p);
        pdf = dot(uvw.w(), scattered.direction()) / pi;
        return true;
    }

    ...
};

class metal : public material {
  public:
    ...

    bool scatter(
        const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered, double& pdf
    ) const override {
        ...
    }
};

class dielectric : public material {
  public:
    ...

    bool scatter(
        const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered, double& pdf
    ) const override {
        ...
    }
};

class diffuse_light : public material {
    ...
};

class isotropic : public material {
  public:
    ...

    bool scatter(
        const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered, double& pdf
    ) const override {
        ...
```

```
    }
};
```

Listing 28: `[material.h]` *Scatter function, with orthonormal basis*

And here we add the accompanying changes to the camera class:

```
class camera {
  ...
  private:
    ...

    color ray_color(const ray& r, int depth, const hittable& world) const {
        ...

        ray scattered;
        color attenuation;
        double pdf_value;
        color color_from_emission = rec.mat->emitted(rec.u, rec.v, rec.p);

        if (!rec.mat->scatter(r, rec, attenuation, scattered, pdf_value))
            return color_from_emission;

        double scattering_pdf = rec.mat->scattering_pdf(r, rec, scattered);
        pdf_value = scattering_pdf;

        ...
    }

    ...
};
```

Listing 29: `[camera.h]` *Updated ray_color function with returned PDF value*

Which produces:

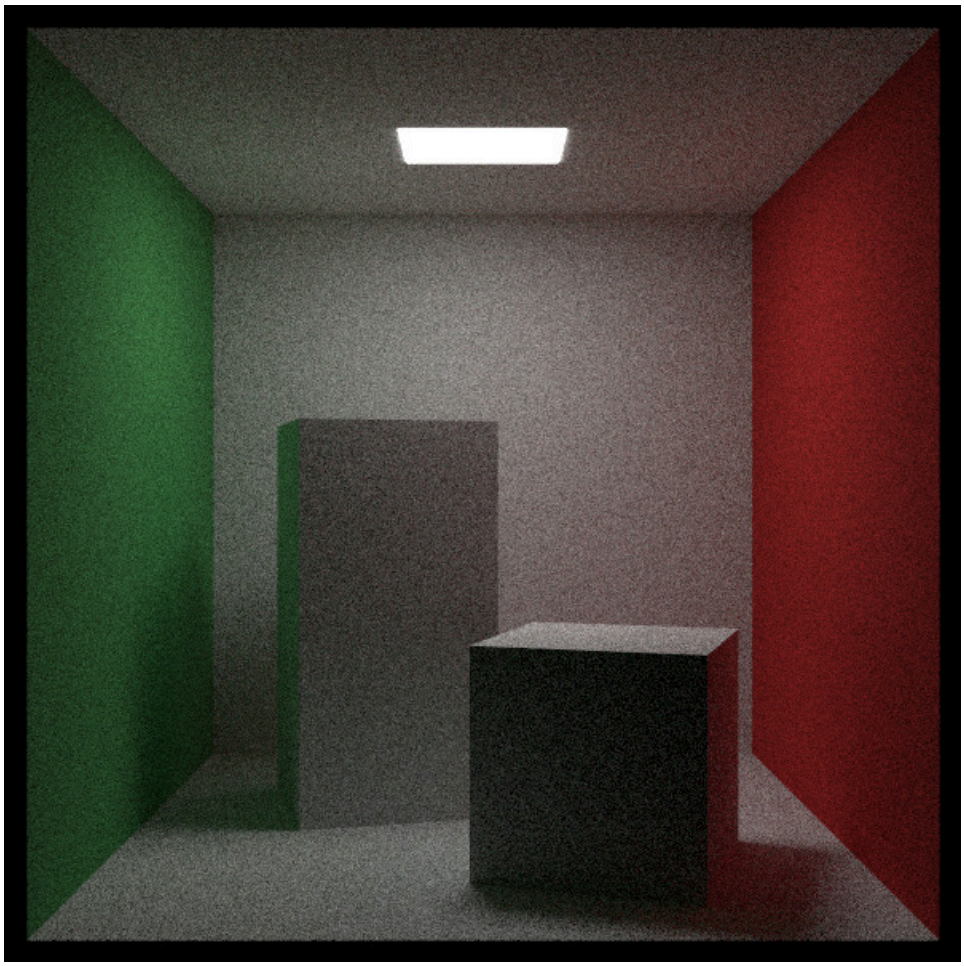

Image 6: *Cornell box, with orthonormal basis scatter function*

Let’s get rid of some of that noise.

But first, let's quickly update the `isotropic` material:

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

    bool scatter(
        const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered, double& pdf
    ) const override {
        scattered = ray(rec.p, random_unit_vector(), r_in.time());
        attenuation = tex->value(rec.u, rec.v, rec.p);
        pdf = 1 / (4 * pi);
        return true;
    }

    double scattering_pdf(const ray& r_in, const hit_record& rec, const ray& scattered)
    const override {
        return 1 / (4 * pi);
    }

  private:
    shared_ptr<texture> tex;
};
```

Listing 30: `[material.h]` *Isotropic material, modified for importance sampling*

9. Sampling Lights Directly

The problem with sampling uniformly over all directions is that lights are no more likely to be sampled than any arbitrary or unimportant direction. We could use shadow rays to solve for the direct lighting at any given point. Instead, I’ll just use a PDF that sends more rays to the light. We can then turn around and change that PDF 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'll need to know the PDF, $p(\omega)$, so that we're not biasing our render. But what is that?

9.1. Getting the PDF of a Light

For a light with a surface area of A, if we sample uniformly on that light, the PDF on the surface is just $\frac{1}{A}$. How much area does the entire surface of the light take up if its projected back onto the unit sphere? Fortunately, there is a simple correspondence, as outlined in this diagram:

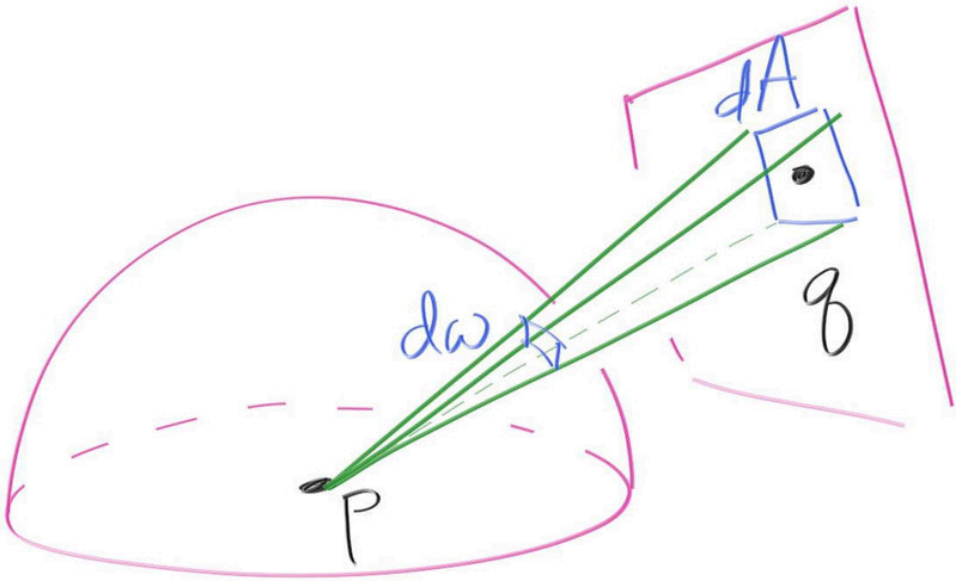

Figure 11: *Projection of light shape onto PDF*

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

$$d\omega = \frac{dA \cdot \cos(\theta)}{\text{distance}^2(p, q)}$$

Since the probability of sampling $d\omega$ and dA must be the same, then

$$\mathrm{p}(\omega) \cdot d\omega = \mathrm{p_q}(q) \cdot dA$$

$$\mathrm{p}(\omega) \cdot \frac{dA \cdot \cos(\theta)}{\text{distance}^2(p, q)} = \mathrm{p_q}(q) \cdot dA$$

We know that if we sample uniformly on the light the PDF on the surface is $\frac{1}{A}$:

$$\mathrm{p_q}(q) = \frac{1}{A}$$

$$\mathrm{p}(\omega) \cdot \frac{dA \cdot \cos(\theta)}{\text{distance}^2(p, q)} = \frac{dA}{A}$$

So

$$\mathrm{p}(\omega) = \frac{\text{distance}^2(p, q)}{\cos(\theta) \cdot A}$$

9.2. Light Sampling

We can hack our `ray_color()` function to sample the light in a very hard-coded fashion just to check that we got the math and concept right:

```
class camera {
  ...
  private:
    ...

    color ray_color(const ray& r, int depth, const hittable& world) const {
        // If we've exceeded the ray bounce limit, no more light is gathered.
        if (depth <= 0)
            return color(0,0,0);

        hit_record rec;

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

        ray scattered;
        color attenuation;
        double pdf_value;
        color color_from_emission = rec.mat->emitted(rec.u, rec.v, rec.p);

        if (!rec.mat->scatter(r, rec, attenuation, scattered, pdf_value))
            return color_from_emission;

        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 color_from_emission;

        double light_area = (343-213)*(332-227);
        auto light_cosine = std::fabs(to_light.y());
        if (light_cosine < 0.000001)
            return color_from_emission;

        pdf_value = distance_squared / (light_cosine * light_area);
        scattered = ray(rec.p, to_light, r.time());

        double scattering_pdf = rec.mat->scattering_pdf(r, rec, scattered);

        color color_from_scatter =
            (attenuation * scattering_pdf * ray_color(scattered, depth-1, world)) / pdf_value;

        return color_from_emission + color_from_scatter;
    }
};
```

Listing 31: `[camera.h]` *Ray color with light sampling*

We'll test this scene with just ten samples per pixel:

```
int main() {
    ...
    cam.aspect_ratio      = 1.0;
    cam.image_width       = 600;
    cam.samples_per_pixel = 10;
    cam.max_depth         = 50;
    cam.background        = color(0,0,0);
    ...
}
```

Listing 32: `[main.cc]` *Ray color with light sampling at 10spp*

With 10 samples per pixel this yields:

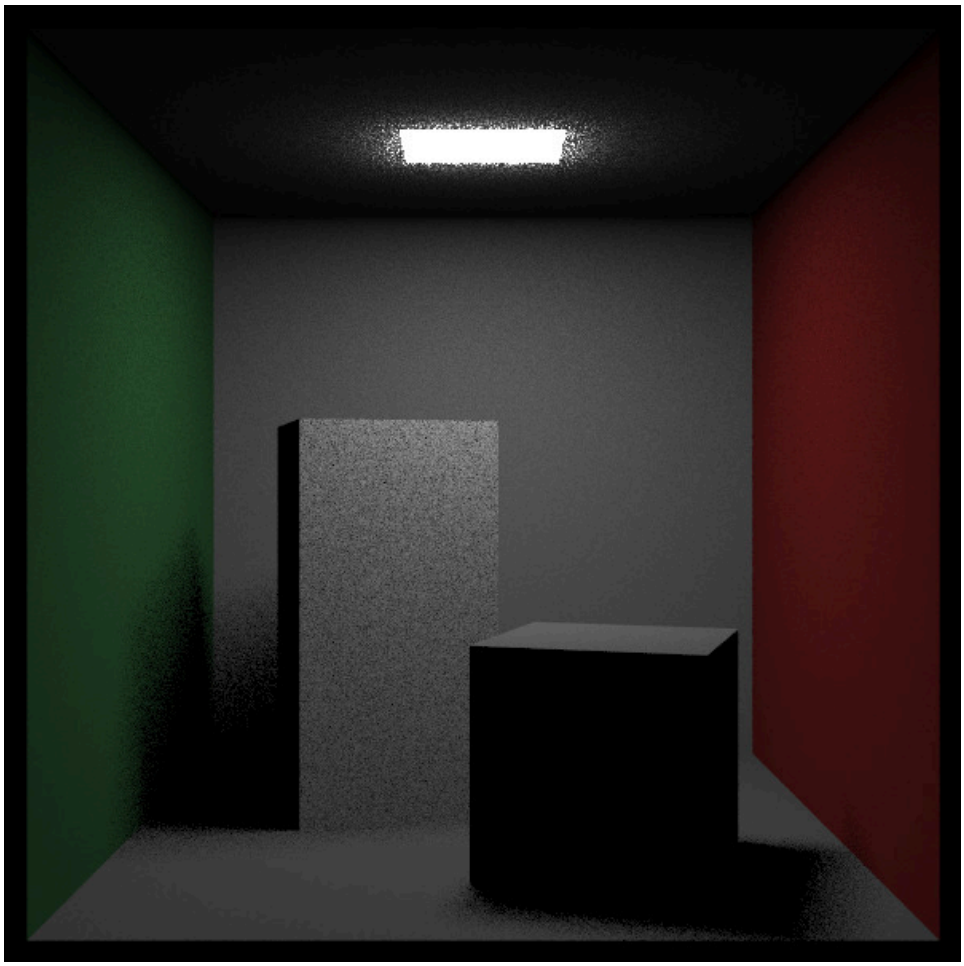

Image 7: *Cornell box, sampling only the light, 10 samples per pixel*

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 `hittable::emitted()` function take extra information:

```
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);
    }
    ...
};

class diffuse_light : public material {
  public:
    ...

    color emitted(const ray& r_in, const hit_record& rec, double u, double v, const point3& p)
    const override {
        if (!rec.front_face)
            return color(0,0,0);
        return tex->value(u, v, p);
    }

    ...
};
```

Listing 33: `[material.h]` *Material emission, directional*

```
class camera {
  ...
  private:
    color ray_color(const ray& r, int depth, const hittable& world) const {
        ...

        color color_from_emission = rec.mat->emitted(r, rec, rec.u, rec.v, rec.p);

        ...
    }
};
```

Listing 34: `[camera.h]` *Material emission, camera::ray_color() changes*

This gives us:

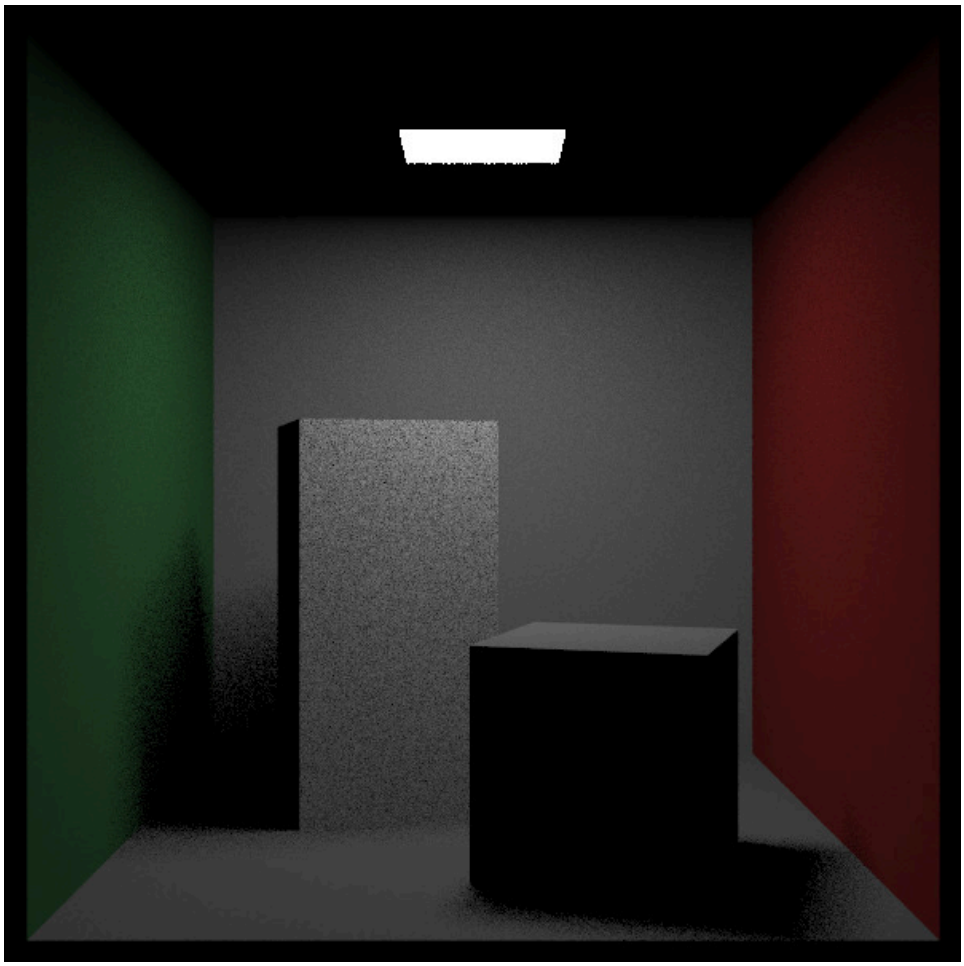

Image 8: *Cornell box, light emitted only in the downward direction*

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. The PDF Class

We've worked with PDFs in quite a lot of code already. I think that now is a good time to figure out how we want to standardize our usage of PDFs. We already know that we are going to have a PDF for the surface and a PDF for the light, so let's create a `pdf` base class. So far, we've had a `pdf()` function that took a direction and returned the PDF's distribution value for that direction. This value has so far been one of $1/4\pi$, $1/2\pi$, and $\cos(\theta)/\pi$. In a couple of our examples we generated the random direction using a different distribution than the distribution of the PDF. We covered this quite a lot in the chapter Playing with Importance Sampling. In general, if we know the distribution of our random directions, we should use a PDF with the same distribution. This will lead to the fastest convergence. With that in mind, we'll create a `pdf` class that is responsible for generating random directions and determining the value of the PDF.

From all of this, any `pdf` class should be responsible for

1. returning a random direction weighted by the internal PDF distribution, and

2. returning the corresponding PDF distribution value in that direction.

The details of how this is done under the hood varies for $\mathrm{pSurface}$ and pLight, 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 `pdf` this implies:

```
#ifndef PDF_H
#define PDF_H

#include "onb.h"


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

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

#endif
```

Listing 35: `[pdf.h]` *The abstract PDF class*

We'll see if we need to add anything else to `pdf` by fleshing out the subclasses. First, we'll create a uniform density over the unit sphere:

```
class sphere_pdf : public pdf {
  public:
    sphere_pdf() {}

    double value(const vec3& direction) const override {
        return 1/ (4 * pi);
    }

    vec3 generate() const override {
        return random_unit_vector();
    }
};
```

Listing 36: `[pdf.h]` *The sphere_pdf class*

Next, let's try a cosine density:

```
class cosine_pdf : public pdf {
  public:
    cosine_pdf(const vec3& w) : uvw(w) {}

    double value(const vec3& direction) const override {
        auto cosine_theta = dot(unit_vector(direction), uvw.w());
        return std::fmax(0, cosine_theta/pi);
    }

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

  private:
    onb uvw;
};
```

Listing 37: `[pdf.h]` *The cosine_pdf class*

We can try this cosine PDF in the `ray_color()` function:

```
#include "hittable.h"
#include "pdf.h"
#include "material.h"

class camera {
  ...
  private:
    ...
    color ray_color(const ray& r, int depth, const hittable& world) const {
        // If we've exceeded the ray bounce limit, no more light is gathered.
        if (depth <= 0)
            return color(0,0,0);

        hit_record rec;

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

        ray scattered;
        color attenuation;
        double pdf_value;
        color color_from_emission = rec.mat->emitted(r, rec, rec.u, rec.v, rec.p);

        if (!rec.mat->scatter(r, rec, attenuation, scattered, pdf_value))
            return color_from_emission;

        cosine_pdf surface_pdf(rec.normal);
        scattered = ray(rec.p, surface_pdf.generate(), r.time());
        pdf_value = surface_pdf.value(scattered.direction());

        double scattering_pdf = rec.mat->scattering_pdf(r, rec, scattered);

        color color_from_scatter =
            (attenuation * scattering_pdf * ray_color(scattered, depth-1, world)) / pdf_value;

        return color_from_emission + color_from_scatter;
    }
};
```

Listing 38: `[camera.h]` *The ray_color function, using cosine PDF*

And set the render back to 1000 samples per pixel:

```
int main() {
    ...
    cam.aspect_ratio      = 1.0;
    cam.image_width       = 600;
    cam.samples_per_pixel = 1000;
    cam.max_depth         = 50;
    cam.background        = color(0,0,0);
    ...
}
```

Listing 39: `[main.cc]` *Reset sampling back to 1000spp*

This yields an exactly matching result so all we've done so far is move some computation up into the `cosine_pdf` class:

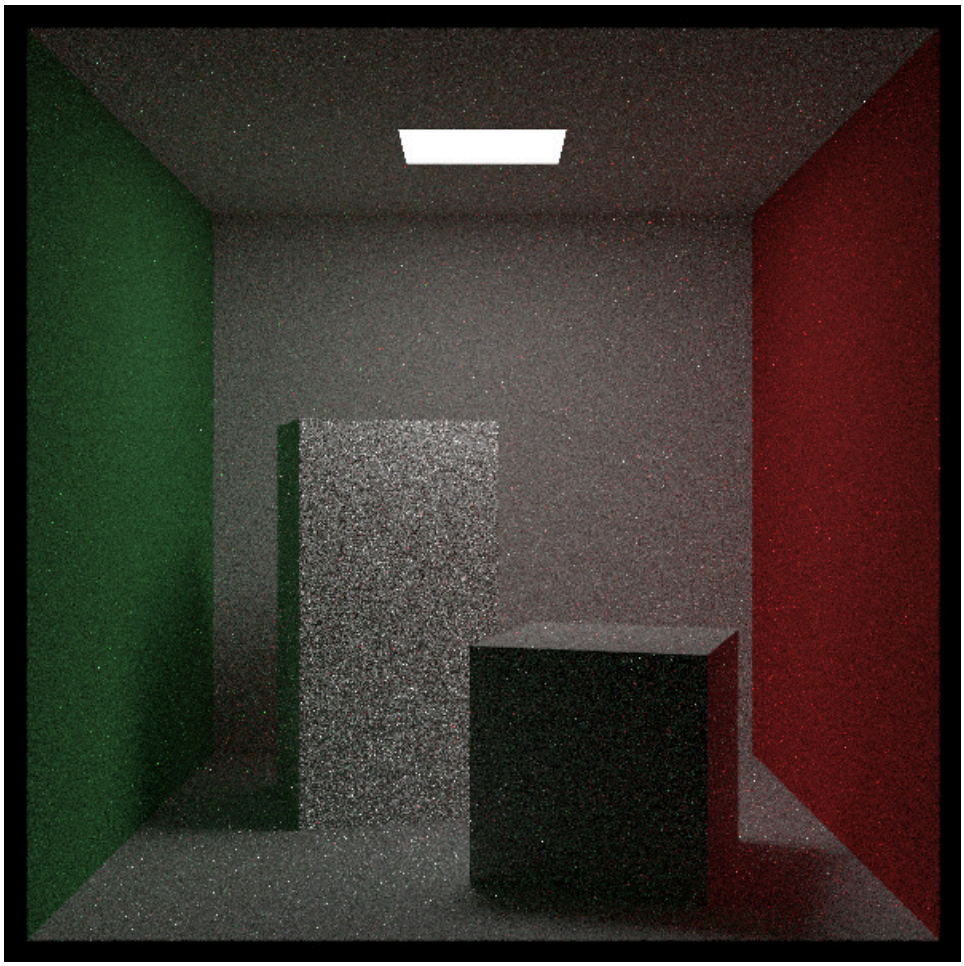

Image 9: *Cornell box with a cosine density PDF*

10.2. Sampling Directions towards a Hittable

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

```
#include "hittable_list.h"
#include "onb.h"

...

class hittable_pdf : public pdf {
  public:
    hittable_pdf(const hittable& objects, const point3& origin)
      : objects(objects), origin(origin)
    {}

    double value(const vec3& direction) const override {
        return objects.pdf_value(origin, direction);
    }

    vec3 generate() const override {
        return objects.random(origin);
    }

  private:
    const hittable& objects;
    point3 origin;
};
```

Listing 40: `[pdf.h]` *The hittable_pdf class*

If we want to sample the light, we will need `hittable` to answer some queries that it doesn't yet have an interface for. The above code assumes the existence of two as-of-yet unimplemented functions in the `hittable` class: `pdf_value()` and `random()`. We need to add these functions for the program to compile. We could go through all of the `hittable` subclasses and add these functions, but that would be a hassle, so we'll just add two trivial functions to the `hittable` base class. This breaks our previously pure abstract implementation, but it saves work. Feel free to write these functions through to subclasses if you want a purely abstract `hittable` interface class.

```
class hittable {
  public:
    virtual ~hittable() = default;

    virtual bool hit(const ray& r, interval ray_t, hit_record& rec) const = 0;

    virtual aabb bounding_box() const = 0;

    virtual double pdf_value(const point3& origin, const vec3& direction) const {
        return 0.0;
    }

    virtual vec3 random(const point3& origin) const {
        return vec3(1,0,0);
    }
};
```

Listing 41: `[hittable.h]` *The hittable class, with two new methods*

And then we change `quad` to implement those functions:

```
class quad : public hittable {
  public:
    quad(const point3& Q, const vec3& u, const vec3& v, shared_ptr<material> mat)
      : Q(Q), u(u), v(v), mat(mat)
    {
        auto n = cross(u, v);
        normal = unit_vector(n);
        D = dot(normal, Q);
        w = n / dot(n,n);

        area = n.length();

        set_bounding_box();
    }

    ...

    double pdf_value(const point3& origin, const vec3& direction) const override {
        hit_record rec;
        if (!this->hit(ray(origin, direction), interval(0.001, infinity), rec))
            return 0;

        auto distance_squared = rec.t * rec.t * direction.length_squared();
        auto cosine = std::fabs(dot(direction, rec.normal) / direction.length());

        return distance_squared / (cosine * area);
    }

    vec3 random(const point3& origin) const override {
        auto p = Q + (random_double() * u) + (random_double() * v);
        return p - origin;
    }

  private:
    point3 Q;
    vec3 u, v;
    vec3 w;
    shared_ptr<material> mat;
    aabb bbox;
    vec3 normal;
    double D;
    double area;
};
```

Listing 42: `[quad.h]` *quad with PDF*

We only need to add `pdf_value()` and `random()` to `quad` because we're using this to importance sample the light, and the only light we have in our scene is a `quad`. if you want other light geometries, or want to use a PDF with other objects, you'll need to implement the above functions for the corresponding classes.

Add a `lights` parameter to the camera `render()` function:

```
class camera {
  public:
    ...

    void render(const hittable& world, const hittable& lights) {
        initialize();

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

        for (int j = 0; j < image_height; j++) {
            std::clog << "\rScanlines remaining: " << (image_height - j) << ' ' << std::flush;
            for (int i = 0; i < image_width; i++) {
                color pixel_color(0,0,0);
                for (int s_j = 0; s_j < sqrt_spp; s_j++) {
                    for (int s_i = 0; s_i < sqrt_spp; s_i++) {
                        ray r = get_ray(i, j, s_i, s_j);
                        pixel_color += ray_color(r, max_depth, world, lights);
                    }
                }
                write_color(std::cout, pixel_samples_scale * pixel_color);
            }
        }

        std::clog << "\rDone.                 \n";
    }

    ...
  private:
    ...

    color ray_color(const ray& r, int depth, const hittable& world, const hittable& lights)
    const {
        ...

        ray scattered;
        color attenuation;
        double pdf_value;
        color color_from_emission = rec.mat->emitted(r, rec, rec.u, rec.v, rec.p);

        if (!rec.mat->scatter(r, rec, attenuation, scattered, pdf_value))
            return color_from_emission;

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

        double scattering_pdf = rec.mat->scattering_pdf(r, rec, scattered);

        color sample_color = ray_color(scattered, depth-1, world, lights);
        color color_from_scatter = (attenuation * scattering_pdf * sample_color) / pdf_value;

        return color_from_emission + color_from_scatter;
    }
};
```

Listing 43: `[camera.h]` *ray_color function with light PDF*

Create a light in the middle of the ceiling:

```
int main() {
    ...

    // Box 2
    shared_ptr<hittable> box2 = 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));
    world.add(box2);

    // Light Sources
    auto empty_material = shared_ptr<material>();
    quad lights(point3(343,554,332), vec3(-130,0,0), vec3(0,0,-105), empty_material);

    camera cam;

    cam.aspect_ratio      = 1.0;
    cam.image_width       = 600;
    cam.samples_per_pixel = 10;
    cam.max_depth         = 50;
    cam.background        = color(0,0,0);

    ...

    cam.render(world, lights);
}
```

Listing 44: `[main.cc]` *Adding a light to the Cornell box*

At 10 samples per pixel we get:

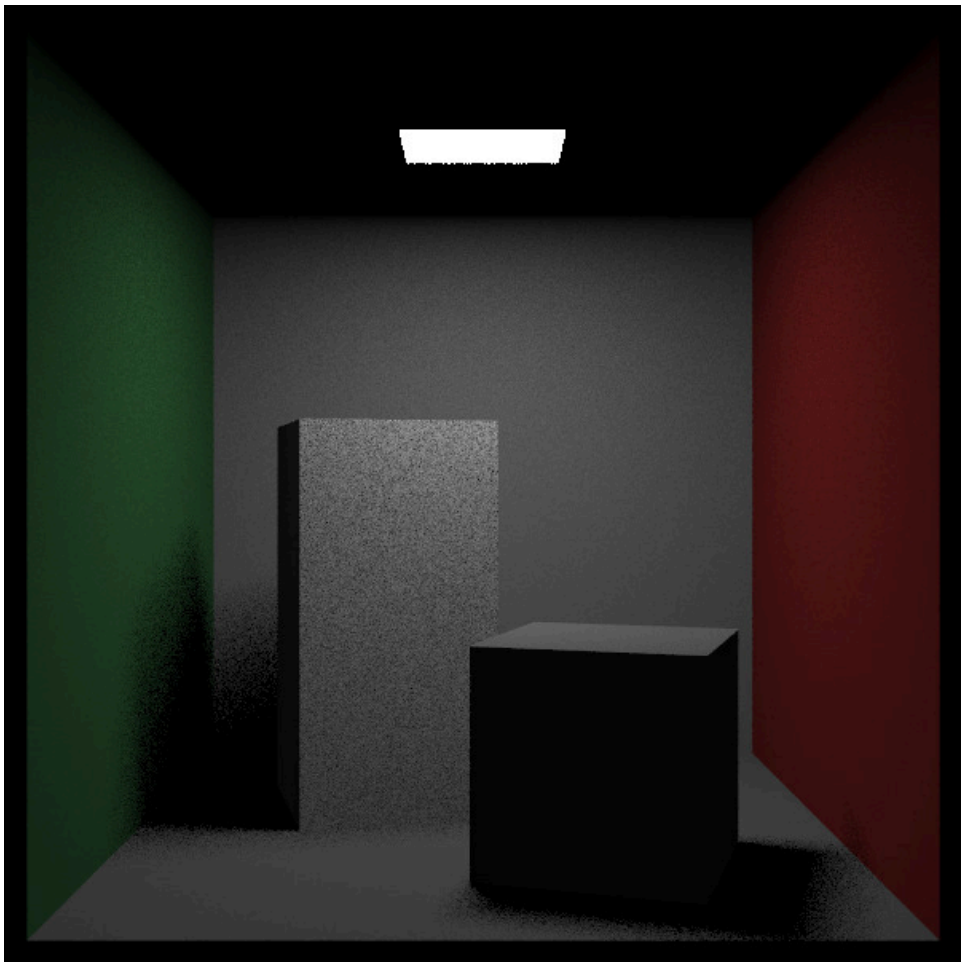

Image 10: *Cornell box, sampling a hittable light, 10 samples per pixel*

10.3. The Mixture PDF Class

As was briefly mentioned in the chapter Playing with Importance Sampling, we can create linear mixtures of any PDFs to form mixture densities that are also PDFs. Any weighted average of PDFs is also a PDF. As long as the weights are positive and add up to any one, we have a new PDF.

$$\text{pMixture}() = w_0 p_0() + w_1 p_1() + w_2 p_2() + \ldots + w_{n-1} p_{n-1}()$$

$$1 = w_0 + w_1 + w_2 + \ldots + w_{n-1}$$

For example, we could just average the two densities:

$$\text{pMixture}(\omega_o) = \frac{1}{2}\text{pSurface}(\omega_o) + \frac{1}{2}\text{pLight}(\omega_o)$$

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. Generating the random direction for a mixture PDF is simple:

```
if (random_double() < 0.5)
    pick direction according to pSurface
else
    pick direction according to pLight
```

But solving for the PDF value of $\mathrm{pMixture}$ is slightly more subtle. We can't just

```
if (direction is from pSurface)
    get PDF value of pSurface
else
    get PDF value of pLight
```

For one, figuring out which PDF the random direction came from is probably not trivial. We don't have any plumbing for `generate()` to tell `value()` what the original `random_double()` was, so we can't trivially say which PDF the random direction comes from. If we thought that the above was correct, we would have to solve backwards to figure which PDF the direction could come from. Which honestly sounds like a nightmare, but fortunately we don't need to do that. There are some directions that both PDFs could have generated. For example, a direction toward the light could have been generated by either pLight *or* $\mathrm{pSurface}$. It is sufficient for us to solve for the PDF value of $\mathrm{pSurface}$ and of pLight for a random direction and then take the PDF mixture weights to solve for the total PDF value for that direction. The mixture density class is actually pretty straightforward:

```
class mixture_pdf : public pdf {
  public:
    mixture_pdf(shared_ptr<pdf> p0, shared_ptr<pdf> p1) {
        p[0] = p0;
        p[1] = p1;
    }

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

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

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

Listing 45: `[pdf.h]` *The mixture_pdf class*

Now we would like to do a mixture density of the cosine sampling and of the light sampling. We can plug it into `ray_color()`:

```
class camera {
  ...
  private:
    ...

    color ray_color(const ray& r, int depth, const hittable& world, const hittable& lights)
    const {
        ...

        if (!rec.mat->scatter(r, rec, attenuation, scattered, pdf_value))
            return color_from_emission;

        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_value = mixed_pdf.value(scattered.direction());

        double scattering_pdf = rec.mat->scattering_pdf(r, rec, scattered);

        color sample_color = ray_color(scattered, depth-1, world, lights);
        color color_from_scatter = (attenuation * scattering_pdf * sample_color) / pdf_value;

        return color_from_emission + color_from_scatter;
    }
```

Listing 46: `[camera.h]` *The ray_color function, using mixture PDF*

Updating `main.cc` to 1000 samples per pixel (not listed) yields:

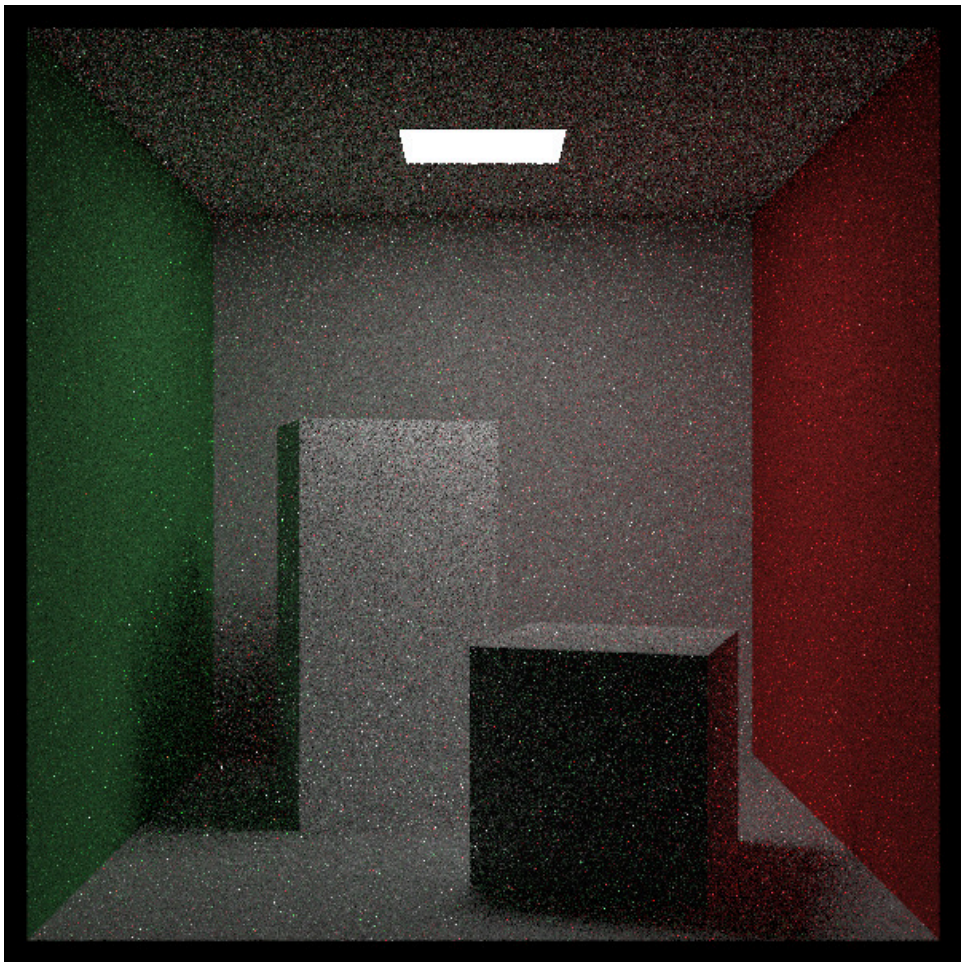

Image 11: *Cornell box, mixture density of cosine and light sampling*

11. Some Architectural Decisions

We won't write any code in this chapter. We're at a crossroads and we need to make some architectural decisions.

The mixture-density approach is an alternative to having more traditional shadow rays. These are rays that check for an unobstructed path from an intersection point to a given light source. Rays that intersect an object between a point and a given light source indicate that the intersection point is in the shadow of that particular light source. The mixture-density approach is something that I personally prefer, because in addition to lights, you can sample windows or bright cracks under doors or whatever else you think might be bright — or important. But you'll still see shadow rays in most professional path tracers. Typically they'll have a predefined number of shadow rays (*e.g* 1, 4, 8, 16) where over the course of rendering, at each place where the path tracing ray intersects, they'll send these terminal shadow rays to random lights in the scene to determine if the intersection is lit by that random light. The intersection will either be lit by that light, or completely in shadow, where more shadow rays lead to a more accurate illumination. After all of the shadow rays terminate (either at a light or at an occluding surface), the inital path tracing ray continues on and more shadow rays are sent at the next intersection. You can't tell the shadow rays what is important, you can only tell them what is emissive, so shadow rays work best on simpler scenes that don't have overly complicated photon distribution. That said, shadow rays terminate at the first thing they run into and don't bounce around, so one shadow ray is cheaper than one path tracing ray, which is the reason that you'll typically see a lot more shadow rays than path tracing rays (*e.g* 1, 4, 8, 16). You could choose shadow rays over mixture-density in a more restricted scene; that's a personal design preference. Shadow rays tend to be cheaper for a crude result than mixture-density and is becoming increasingly common in realtime.

There are some other issues with the code.

The PDF construction is hard coded in the `ray_color()` function. We should clean that up.

We've accidentally broken the specular rays (glass and metal), and they are no longer supported. The math would continue to work out if we just made their scattering function a delta function, but that would lead to all kinds of floating point disasters. We could either make specular reflection a special case that skips $f()/p()$, or we could set surface roughness to a very small — but nonzero — value and have almost-mirrors that look perfectly smooth but that 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 we'll add perfect specular surfaces that just skip over explicit $f()/p()$ calculations.

We also lack a real background function infrastructure in case we want to add an environment map or a more interesting functional background. Some environment maps are HDR (the RGB components are normalized floats rather than 0–255 bytes). Our output has been HDR all along; we've just been truncating it.

Finally, our renderer is RGB. 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, most studios still get away with 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 touch on this at 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 add instrumentation to do that). We also need to know if the scattered ray is specular, and we can do this either by asking the `hit()` function or the `material` class.

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 `ray_color()` needs to be aware of whether this ray is diffuse or specular. Fortunately, we have decided 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 class like we did for `hittable`:

```
#include "hittable.h"
#include "onb.h"
#include "texture.h"

class scatter_record {
  public:
    color attenuation;
    shared_ptr<pdf> pdf_ptr;
    bool skip_pdf;
    ray skip_pdf_ray;
};

class material {
  public:
    ...

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

    ...
};
```

Listing 47: `[material.h]` *Refactoring the material class*

The `lambertian` material becomes simpler:

```
#include "hittable.h"
#include "onb.h"
#include "pdf.h"
#include "texture.h"

...

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

    bool scatter(const ray& r_in, const hit_record& rec, scatter_record& srec) const override {
        srec.attenuation = tex->value(rec.u, rec.v, rec.p);
        srec.pdf_ptr = make_shared<cosine_pdf>(rec.normal);
        srec.skip_pdf = false;
        return true;
    }

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

  private:
    shared_ptr<texture> tex;
};
```

Listing 48: `[material.h]` *New lambertian scatter() method*

As does the `isotropic` material:

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

    bool scatter(const ray& r_in, const hit_record& rec, scatter_record& srec) const override {
        srec.attenuation = tex->value(rec.u, rec.v, rec.p);
        srec.pdf_ptr = make_shared<sphere_pdf>();
        srec.skip_pdf = false;
        return true;
    }

    double scattering_pdf(const ray& r_in, const hit_record& rec, const ray& scattered)
    const override {
        return 1 / (4 * pi);
    }

  private:
    shared_ptr<texture> tex;
};
```

Listing 49: `[material.h]` *New isotropic scatter() method*

And `ray_color()` changes are small:

```
class camera {
  ...
  private:
    ...

    color ray_color(const ray& r, int depth, const hittable& world, const hittable& lights)
    const {
        // If we've exceeded the ray bounce limit, no more light is gathered.
        if (depth <= 0)
            return color(0,0,0);

        hit_record rec;

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

        scatter_record srec;
        color color_from_emission = rec.mat->emitted(r, rec, rec.u, rec.v, rec.p);

        if (!rec.mat->scatter(r, rec, srec))
            return color_from_emission;

        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_value = p.value(scattered.direction());

        double scattering_pdf = rec.mat->scattering_pdf(r, rec, scattered);

        color sample_color = ray_color(scattered, depth-1, world, lights);
        color color_from_scatter =
            (srec.attenuation * scattering_pdf * sample_color) / pdf_value;

        return color_from_emission + color_from_scatter;
    }
};
```

Listing 50: `[camera.h]` *The ray_color function, using mixture PDF*

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& albedo, double fuzz) : albedo(albedo), fuzz(fuzz < 1 ? fuzz : 1) {}

    bool scatter(const ray& r_in, const hit_record& rec, scatter_record& srec) const override {
        vec3 reflected = reflect(r_in.direction(), rec.normal);
        reflected = unit_vector(reflected) + (fuzz * random_unit_vector());

        srec.attenuation = albedo;
        srec.pdf_ptr = nullptr;
        srec.skip_pdf = true;
        srec.skip_pdf_ray = ray(rec.p, reflected, r_in.time());

        return true;
    }

  private:
    color albedo;
    double fuzz;
};

class dielectric : public material {
  public:
    dielectric(double refraction_index) : refraction_index(refraction_index) {}

    bool scatter(const ray& r_in, const hit_record& rec, scatter_record& srec) const override {
        srec.attenuation = color(1.0, 1.0, 1.0);
        srec.pdf_ptr = nullptr;
        srec.skip_pdf = true;
        double ri = rec.front_face ? (1.0/refraction_index) : refraction_index;

        vec3 unit_direction = unit_vector(r_in.direction());
        double cos_theta = std::fmin(dot(-unit_direction, rec.normal), 1.0);
        double sin_theta = std::sqrt(1.0 - cos_theta*cos_theta);

        bool cannot_refract = ri * sin_theta > 1.0;
        vec3 direction;

        if (cannot_refract || reflectance(cos_theta, ri) > random_double())
            direction = reflect(unit_direction, rec.normal);
        else
            direction = refract(unit_direction, rec.normal, ri);

        srec.skip_pdf_ray = ray(rec.p, direction, r_in.time());
        return true;
    }

  ...
};
```

Listing 51: `[material.h]` *The metal and dielectric scatter methods*

Note that if the fuzziness is nonzero, this surface isn't really ideally specular, but the implicit sampling works just like it did before. We're effectively skipping all of our PDF work for the materials that we're treating specularly.

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

```
class camera {
  ...
  private:
    ...
    color ray_color(const ray& r, int depth, const hittable& world, const hittable& lights)
    const {
        ...

        if (!rec.mat->scatter(r, rec, srec))
            return color_from_emission;

        if (srec.skip_pdf) {
            return srec.attenuation * ray_color(srec.skip_pdf_ray, depth-1, world, lights);
        }

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

        ...
    }
};
```

Listing 52: `[camera.h]` *Ray color function with implicitly-sampled rays*

We'll check our work by changing a block to metal. We'd also like to swap out one of the blocks for a glass object, but we'll push that off for the next section. Glass objects are difficult to render well, so we'd like to make a PDF for them, but we have some more work to do before we're able to do that.

```
int main() {
    ...
    // Light
    world.add(make_shared<quad>(point3(213,554,227), vec3(130,0,0), vec3(0,0,105), light));

    // Box 1
    shared_ptr<material> aluminum = make_shared<metal>(color(0.8, 0.85, 0.88), 0.0);
    shared_ptr<hittable> box1 = 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));
    world.add(box1);

    // Box 2
    shared_ptr<hittable> box2 = 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));
    world.add(box2);

    // Light Sources
    auto empty_material = shared_ptr<material>();
    quad lights(point3(343,554,332), vec3(-130,0,0), vec3(0,0,-105), empty_material);

    ...
}
```

Listing 53: `[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.

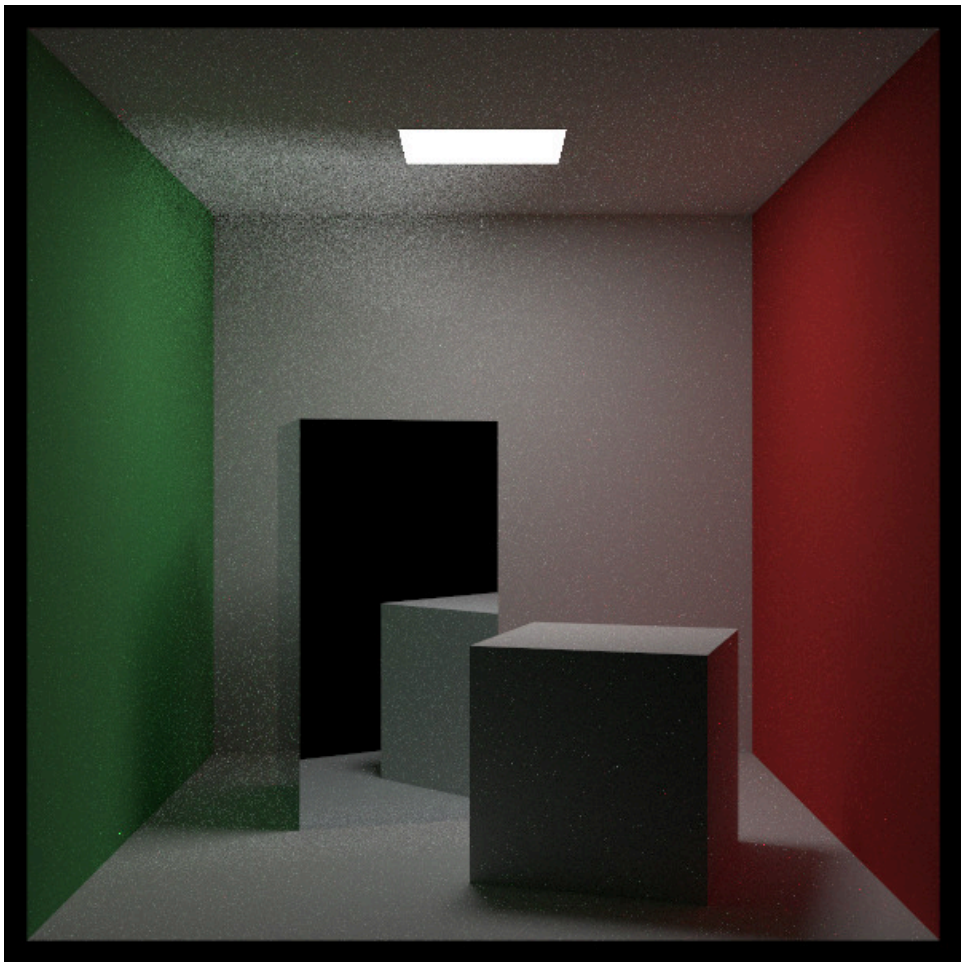

Image 12: *Cornell box with arbitrary PDF functions*

12.3. Sampling a Sphere Object

The noisiness on the ceiling could be reduced by making a PDF of the metal block. We would also want a PDF for the block if we made it glass. But making a PDF for a block is quite a bit of work and isn't terribly interesting, so let's create a PDF for a glass sphere instead. It's quicker and makes for a more interesting render. We need to figure out how to sample a sphere to determine an appropriate PDF distribution. If we want to sample a sphere from a point outside of the sphere, we can't just pick a random point on its surface and be done. If we did that, we would frequently pick a point on the far side of the sphere, which would be occluded by the front side of the sphere. We need a way to uniformly sample the side of the sphere that is visible from an arbitrary point. 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 axis goes from the ray origin through the sphere center, with the sides of the cone tangent to the sphere — see illustration below. Let's say the code has `theta_max`. Recall from the Generating Random Directions chapter that to sample θ we have:

$$r_2 = \int_0^\theta 2\pi f(\theta') \sin(\theta') d\theta'$$

Here $f(\theta')$ is an as-of-yet uncalculated constant C, so:

$$r_2 = \int_0^\theta 2\pi C \sin(\theta') d\theta'$$

If we solve through the calculus:

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

So

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

We are constraining our distribution so that the random direction must be less than θ_{max}. This means that the integral from 0 to θ_{max} must be one, and therefore $r_2 = 1$. We can use this to solve for C:

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

$$1 = 2\pi \cdot C \cdot (1 - \cos(\theta_{max}))$$

$$C = \frac{1}{2\pi \cdot (1 - \cos(\theta_{max})}$$

Which gives us an equality between θ, θ_{max}, and r_2:

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

We sample ϕ 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}?

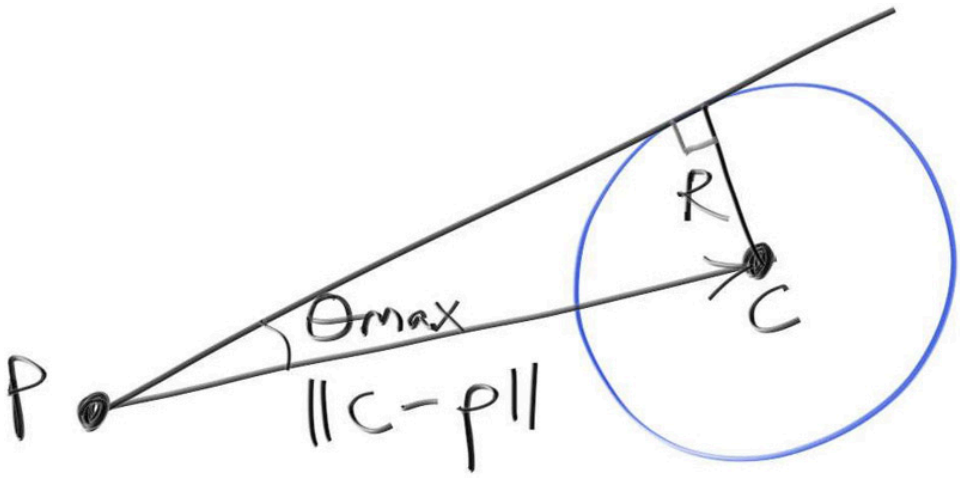

Figure 12: *A sphere-enclosing cone*

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 a uniform distribution toward the sphere the PDF is $1/\text{solid\_angle}$. What is the solid angle of the sphere? It has something to do with the C above. It is — by definition — the area on the unit sphere, so the integral is

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

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 $\cos(\theta_{max}) = 1$, 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:

```
#include "hittable.h"
#include "onb.h"

class sphere : public hittable {
  public:
    ...

    aabb bounding_box() const override { return bbox; }

    double pdf_value(const point3& origin, const vec3& direction) const override {
        // This method only works for stationary spheres.

        hit_record rec;
        if (!this->hit(ray(origin, direction), interval(0.001, infinity), rec))
            return 0;

        auto dist_squared = (center.at(0) - origin).length_squared();
        auto cos_theta_max = std::sqrt(1 - radius*radius/dist_squared);
        auto solid_angle = 2*pi*(1-cos_theta_max);

        return  1 / solid_angle;
    }

    vec3 random(const point3& origin) const override {
        vec3 direction = center.at(0) - origin;
        auto distance_squared = direction.length_squared();
        onb uvw(direction);
        return uvw.transform(random_to_sphere(radius, distance_squared));
    }

  private:
    ...

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

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

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

Listing 54: `[sphere.h]` *Sphere with PDF*

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

```
int main() {
    ...

    // Light
    world.add(make_shared<quad>(point3(213,554,227), vec3(130,0,0), vec3(0,0,105), light));

    // Box
    shared_ptr<hittable> box1 = 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));
    world.add(box1);

    // Glass Sphere
    auto glass = make_shared<dielectric>(1.5);
    world.add(make_shared<sphere>(point3(190,90,190), 90, glass));

    // Light Sources
    auto empty_material = shared_ptr<material>();
    quad lights(point3(343,554,332), vec3(-130,0,0), vec3(0,0,-105), empty_material);

    ...
}
```

Listing 55: `[main.cc]` *Sampling just the sphere*

This yields a noisy room, 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!

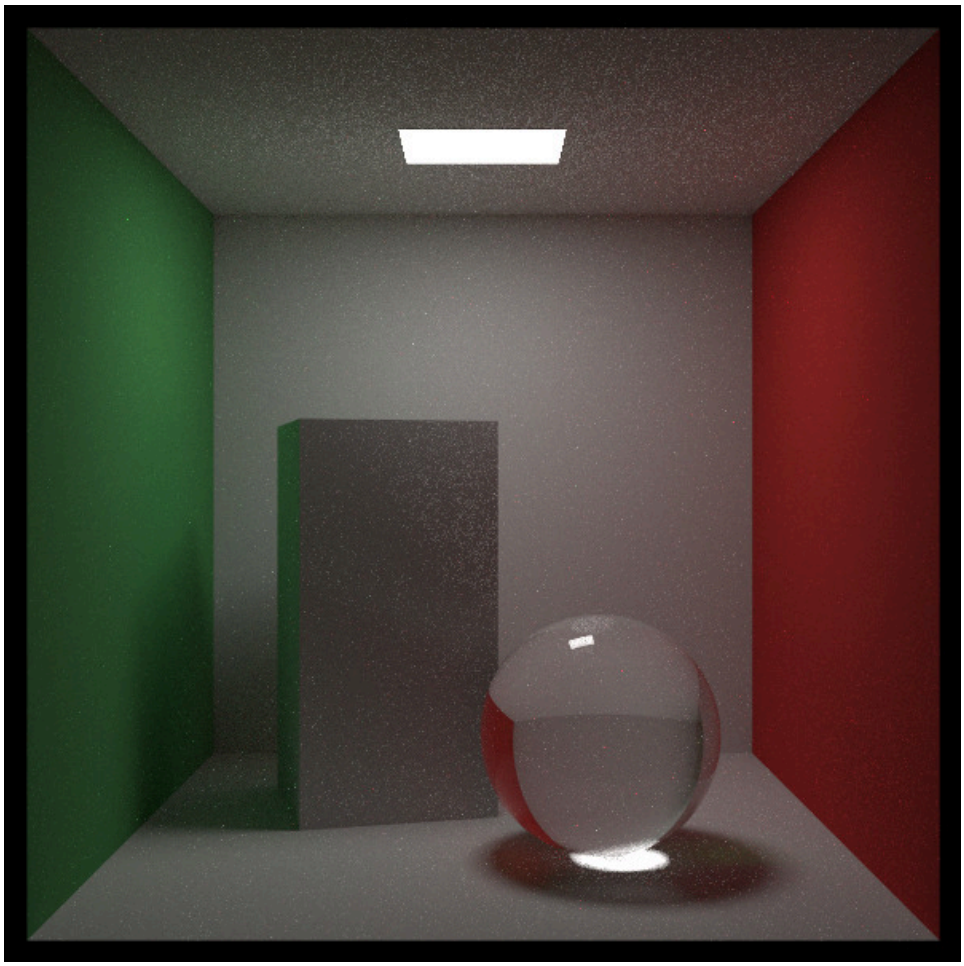

Image 13: *Cornell box with glass sphere, using new PDF functions*

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 distributions. 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`.

```
class hittable_list : public hittable {
  public:
    ...

    aabb bounding_box() const override { return bbox; }

    double pdf_value(const point3& origin, const vec3& direction) const override {
        auto weight = 1.0 / objects.size();
        auto sum = 0.0;

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

        return sum;
    }

    vec3 random(const point3& origin) const override {
        auto int_size = int(objects.size());
        return objects[random_int(0, int_size-1)]->random(origin);
    }

    ...
};
```

Listing 56: `[hittable_list.h]` *Creating a mixture of densities*

We assemble a list of light sources to pass to `camera::render()`:

```
int main() {
    ...

    // Light Sources
    auto empty_material = shared_ptr<material>();
    hittable_list lights;
    lights.add(
        make_shared<quad>(point3(343,554,332), vec3(-130,0,0), vec3(0,0,-105), empty_material));
    lights.add(make_shared<sphere>(point3(190, 90, 190), 90, empty_material));

    ...
}
```

Listing 57: `[main.cc]` *Updating the scene*

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

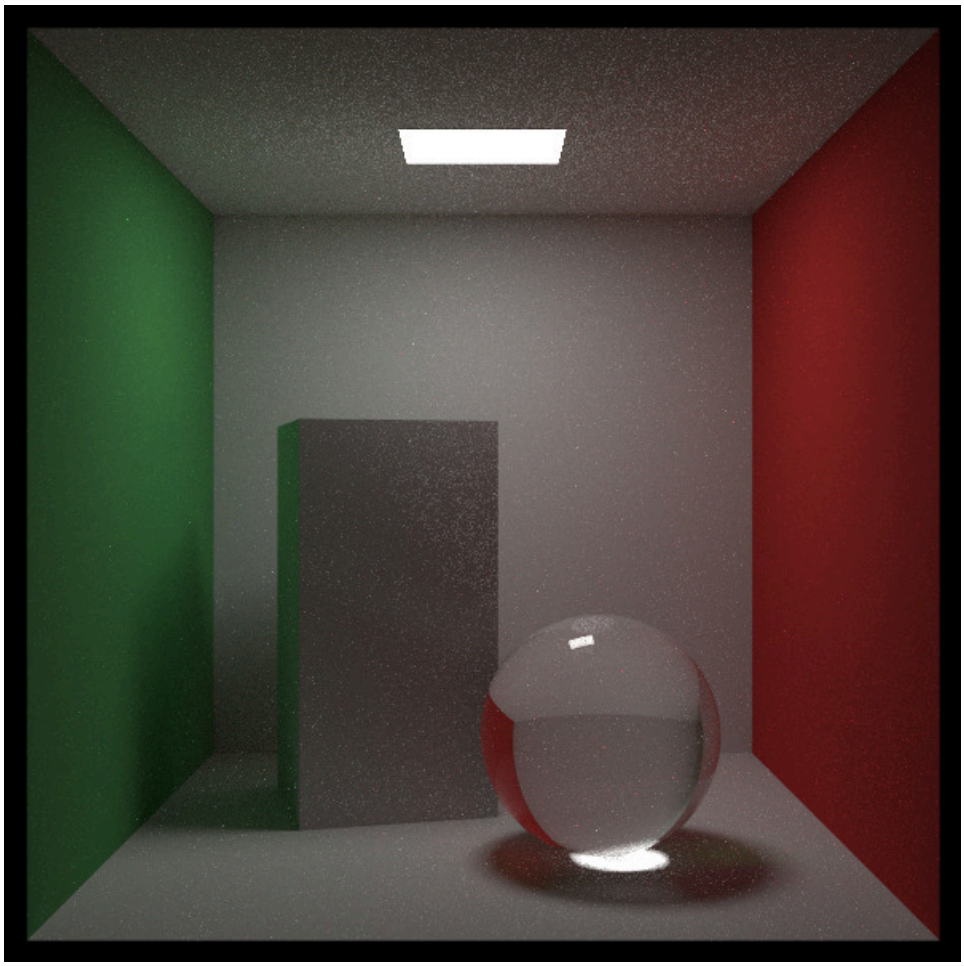

Image 14: *Cornell box using a mixture of glass & light PDFs*

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 your renders, and this acne is white or black — where one "bad" sample seems to kill the whole pixel — then 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 that working with 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, const color& pixel_color) {
    auto r = pixel_color.x();
    auto g = pixel_color.y();
    auto b = pixel_color.z();

    // Replace NaN components with zero.
    if (r != r) r = 0.0;
    if (g != g) g = 0.0;
    if (b != b) b = 0.0;

    // Apply a linear to gamma transform for gamma 2
    r = linear_to_gamma(r);
    g = linear_to_gamma(g);
    b = linear_to_gamma(b);

    // Translate the [0,1] component values to the byte range [0,255].
    static const interval intensity(0.000, 0.999);
    int rbyte = int(256 * intensity.clamp(r));
    int gbyte = int(256 * intensity.clamp(g));
    int bbyte = int(256 * intensity.clamp(b));

    // Write out the pixel color components.
    out << rbyte << ' ' << gbyte << ' ' << bbyte << '\n';
}
```

Listing 58: `[color.h]` *NaN-tolerant write_color function*

Happily, the black specks are gone:

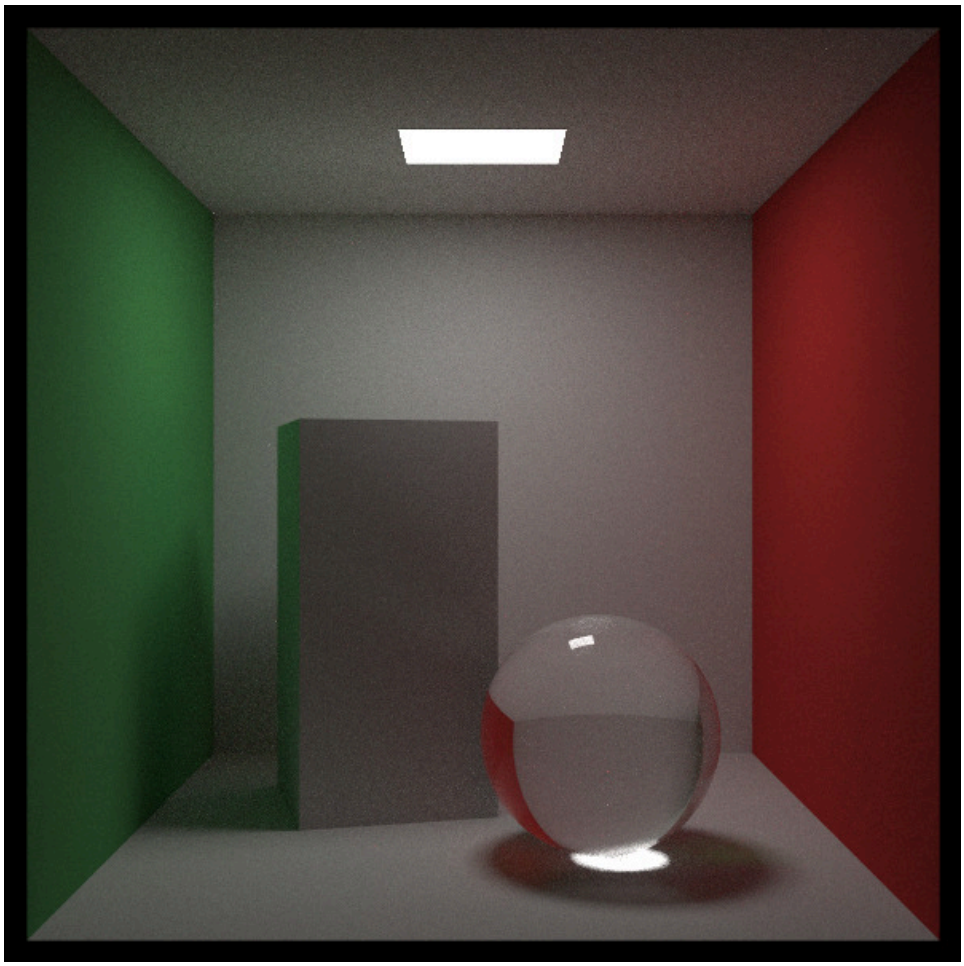

Image 15: *Cornell box with anti-acne color function*

13. The Rest of Your Life

The purpose of this book was to walk through all of the little details (dotting all the i's and crossing all of the t's) necessary when organizing a physically based renderer's sampling approach. You should now be able to take all of this detail and 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. They are also 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
Arman Uguray
Becker
Ben Kerl
Benjamin Summerton
Bennett Hardwick
Benny Tsang
Dan Drummond
David Chambers
David Hart
Dimitry Ishenko
Dmitry Lomov
Eric Haines
Fabio Sancinetti
Filipe Scur
Frank He
Gareth Martin
Gerrit Wessendorf
Grue Debry
Gustaf Waldemarson
Ingo Wald
Jason Stone
JC-ProgJava
Jean Buckley
Jeff Smith
Joey Cho
John Kilpatrick
Kaan Eraslan
Lorenzo Mancini
Manas Kale
Marcus Ottosson
Mark Craig
Markus Boos
Matthew Heimlich
Nakata Daisuke
Nate Rupsis
Paul Melis
Phil Cristensen
LollipopFt
Ronald Wotzlaw
Shaun P. Lee
Shota Kawajiri
Tatsuya Ogawa
Thiago Ize
Thien Tran
Vahan Sosoyan
WANG Lei
Yann Herklotz
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, Trevor David Black, Steve Hollasch
- **Version/Edition**: v4.0.1
- **Date**: 2024-08-30
- **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{Shirley2024RTW3}

@misc{Shirley2024RTW3,
   title = {Ray Tracing: The Rest of Your Life},
   author = {Peter Shirley, Trevor David Black, Steve Hollasch},
   year = {2024},
   month = {August},
   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{Shirley2024RTW3}

@online{Shirley2024RTW3,
   title = {Ray Tracing: The Rest of Your Life},
   author = {Peter Shirley, Trevor David Black, Steve Hollasch},
   year = {2024},
   month = {August},
   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.
```