Ray Tracing in One Weekend

Peter Shirley
edited by Steve Hollasch and Trevor David Black

Version 3.2.3, 2020-12-07

Contents

1. Overview

I’ ve taught many graphics classes over the years. Often I do them in ray tracing, because you are forced to write all the code, but you can still get cool images with no API. I decided to adapt my course notes into a how-to, to get you to a cool program as quickly as possible. It will not be a full-featured ray tracer, but it does have the indirect lighting which has made ray tracing a staple in movies. Follow these steps, and the architecture of the ray tracer you produce will be good for extending to a more extensive ray tracer if you get excited and want to pursue that.

When somebody says “ray tracing” it could mean many things. What I am going to describe is technically a path tracer, and a fairly general one. While the code will be pretty simple (let the computer do the work!) I think you’ ll be very happy with the images you can make.

I’ll take you through writing a ray tracer in the order I do it, along with some debugging tips. By the end, you will have a ray tracer that produces some great images. You should be able to do this in a weekend. If you take longer, don’t worry about it. I use C++ as the driving language, but you don’t need to. However, I suggest you do, because it’s fast, portable, and most production movie and video game renderers are written in C++. Note that I avoid most “modern features” of C++, but inheritance and operator overloading are too useful for ray tracers to pass on. I do not provide the code online, but the code is real and I show all of it except for a few straightforward operators in the `vec3` class. I am a big believer in typing in code to learn it, but when code is available I use it, so I only practice what I preach when the code is not available. So don’t ask!

I have left that last part in because it is funny what a 180 I have done. Several readers ended up with subtle errors that were helped when we compared code. So please do type in the code, but if you want to look at mine it is at:

https://github.com/RayTracing/raytracing.github.io/

I assume a little bit of familiarity with vectors (like dot product and vector addition). If you don’t know that, do a little review. If you need that review, or to learn it for the first time, check out Marschner’s and my graphics text, Foley, Van Dam, *et al.*, or McGuire’s graphics codex.

If you run into trouble, or do something cool you’d like to show somebody, send me some email at ptrshrl@gmail.com.

I’ll be maintaining a site related to the book including further reading and links to resources at a blog https://in1weekend.blogspot.com/ related to this book.

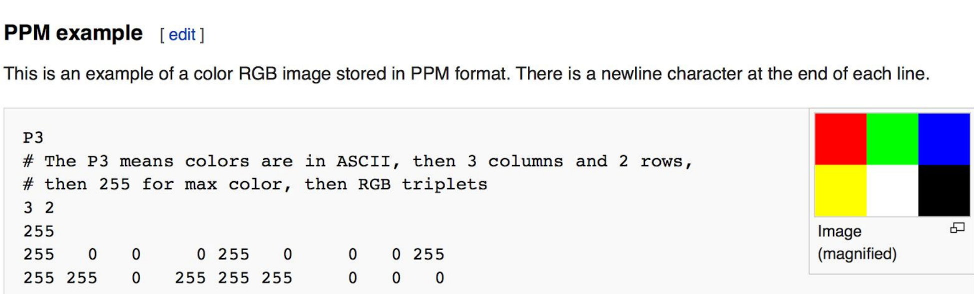
Thanks to everyone who lent a hand on this project. You can find them in the acknowledgments section at the end of this book.

Let’s get on with it!

2. Output an Image

2.1. The PPM Image Format

Whenever you start a renderer, you need a way to see an image. The most straightforward way is to write it to a file. The catch is, there are so many formats. Many of those are complex. I always start with a plain text ppm file. Here’s a nice description from Wikipedia:


PPM example [edit]

This is an example of a color RGB image stored in PPM format. There is a newline character at the end of each line.

```
P3
# The P3 means colors are in ASCII, then 3 columns and 2 rows,
# then 255 for max color, then RGB triplets
3 2
255
255   0   0     0 255   0     0   0 255
255 255   0   255 255 255     0   0   0
```

Image (magnified)

Figure 1: *PPM Example*

Let’ s make some C++ code to output such a thing:

```cpp
#include <iostream>

int main() {

    // Image

    const int image_width = 256;
    const int image_height = 256;

    // Render

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

    for (int j = image_height-1; j >= 0; --j) {
        for (int i = 0; i < image_width; ++i) {
            auto r = double(i) / (image_width-1);
            auto g = double(j) / (image_height-1);
            auto b = 0.25;

            int ir = static_cast<int>(255.999 * r);
            int ig = static_cast<int>(255.999 * g);
            int ib = static_cast<int>(255.999 * b);

            std::cout << ir << ' ' << ig << ' ' << ib << '\n';
        }
    }
}
```

Listing 1: [main.cc] *Creating your first image*

There are some things to note in that code:

1. The pixels are written out in rows with pixels left to right.
2. The rows are written out from top to bottom.
3. By convention, each of the red/green/blue components range from 0.0 to 1.0. We will relax that later when we internally use high dynamic range, but before output we will tone map to the zero to one range, so this code won’ t change.
4. Red goes from fully off (black) to fully on (bright red) from left to right, and green goes from black at the bottom to fully on at the top. Red and green together make yellow so we should expect the upper right corner to be yellow.

2.2. Creating an Image File

Because the file is written to the program output, you'll need to redirect it to an image file. Typically this is done from the command-line by using the `>` redirection operator, like so:

```
build\Release\inOneWeekend.exe > image.ppm
```

This is how things would look on Windows. On Mac or Linux, it would look like this:

```
build/inOneWeekend > image.ppm
```

Opening the output file (in `ToyViewer` on my Mac, but try it in your favorite viewer and Google "ppm viewer" if your viewer doesn't support it) shows this result:

Image 1: First PPM image

Hooray! This is the graphics "hello world". If your image doesn't look like that, open the output file in a text editor and see what it looks like. It should start something like this:

```
P3
256 256
255
0 255 63
1 255 63
2 255 63
3 255 63
4 255 63
5 255 63
6 255 63
7 255 63
8 255 63
9 255 63
...
```

Listing 2: *First image output*

If it doesn't, then you probably just have some newlines or something similar that is confusing the image reader.

If you want to produce more image types than PPM, I am a fan of `stb_image.h`, a header-only image library available on GitHub at https://github.com/nothings/stb.

2.3. Adding a Progress Indicator

Before we continue, let's add a progress indicator to our output. This is a handy way to track the progress of a long render, and also to possibly identify a run that's stalled out due to an infinite loop or other problem.

Our program outputs the image to the standard output stream (`std::cout`), so leave that alone and instead write to the error output stream (`std::cerr`):

```
    for (int j = image_height-1; j >= 0; --j) {
        std::cerr << "\rScanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            auto r = double(i) / (image_width-1);
            auto g = double(j) / (image_height-1);
            auto b = 0.25;

            int ir = static_cast<int>(255.999 * r);
            int ig = static_cast<int>(255.999 * g);
            int ib = static_cast<int>(255.999 * b);

            std::cout << ir << ' ' << ig << ' ' << ib << '\n';
        }
    }

    std::cerr << "\nDone.\n";
```

Listing 3: [main.cc] *Main render loop with progress reporting*

3. The vec3 Class

Almost all graphics programs have some class(es) for storing geometric vectors and colors. In many systems these vectors are 4D (3D plus a homogeneous coordinate for geometry, and RGB plus an alpha transparency channel for colors). For our purposes, three coordinates suffices. We'll use the same class `vec3` for colors, locations, directions, offsets, whatever. Some people don't like this because it doesn't prevent you from doing something silly, like adding a color to a location. They have a good point, but we're going to always take the "less code" route when not obviously wrong. In spite of this, we do declare two aliases for `vec3`: `point3` and `color`. Since these two types are just aliases for `vec3`, you won't get warnings if you pass a `color` to a function expecting a `point3`, for example. We use them only to clarify intent and use.

3.1. Variables and Methods

Here’s the top part of my `vec3` class:

```cpp
#ifndef VEC3_H
#define VEC3_H

#include <cmath>
#include <iostream>

using std::sqrt;

class vec3 {
    public:
        vec3() : e{0,0,0} {}
        vec3(double e0, double e1, double e2) : e{e0, e1, e2} {}

        double x() const { return e[0]; }
        double y() const { return e[1]; }
        double z() const { return e[2]; }

        vec3 operator-() const { return vec3(-e[0], -e[1], -e[2]); }
        double operator[](int i) const { return e[i]; }
        double& operator[](int i) { return e[i]; }

        vec3& operator+=(const vec3 &v) {
            e[0] += v.e[0];
            e[1] += v.e[1];
            e[2] += v.e[2];
            return *this;
        }

        vec3& operator*=(const double t) {
            e[0] *= t;
            e[1] *= t;
            e[2] *= t;
            return *this;
        }

        vec3& operator/=(const double t) {
            return *this *= 1/t;
        }

        double length() const {
            return sqrt(length_squared());
        }

        double length_squared() const {
            return e[0]*e[0] + e[1]*e[1] + e[2]*e[2];
        }

    public:
        double e[3];
};

// Type aliases for vec3
using point3 = vec3;   // 3D point
using color = vec3;    // RGB color
```

```
#endif
```

Listing 4: [vec3.h] *vec3 class*

We use `double` here, but some ray tracers use `float`. Either one is fine — follow your own tastes.

3.2. vec3 Utility Functions

The second part of the header file contains vector utility functions:

```
// vec3 Utility Functions

inline std::ostream& operator<<(std::ostream &out, const vec3 &v) {
    return out << v.e[0] << ' ' << v.e[1] << ' ' << v.e[2];
}

inline vec3 operator+(const vec3 &u, const vec3 &v) {
    return vec3(u.e[0] + v.e[0], u.e[1] + v.e[1], u.e[2] + v.e[2]);
}

inline vec3 operator-(const vec3 &u, const vec3 &v) {
    return vec3(u.e[0] - v.e[0], u.e[1] - v.e[1], u.e[2] - v.e[2]);
}

inline vec3 operator*(const vec3 &u, const vec3 &v) {
    return vec3(u.e[0] * v.e[0], u.e[1] * v.e[1], u.e[2] * v.e[2]);
}

inline vec3 operator*(double t, const vec3 &v) {
    return vec3(t*v.e[0], t*v.e[1], t*v.e[2]);
}

inline vec3 operator*(const vec3 &v, double t) {
    return t * v;
}

inline vec3 operator/(vec3 v, double t) {
    return (1/t) * v;
}

inline double dot(const vec3 &u, const vec3 &v) {
    return u.e[0] * v.e[0]
         + u.e[1] * v.e[1]
         + u.e[2] * v.e[2];
}

inline vec3 cross(const vec3 &u, const vec3 &v) {
    return vec3(u.e[1] * v.e[2] - u.e[2] * v.e[1],
                u.e[2] * v.e[0] - u.e[0] * v.e[2],
                u.e[0] * v.e[1] - u.e[1] * v.e[0]);
}

inline vec3 unit_vector(vec3 v) {
    return v / v.length();
}
```

Listing 5: [vec3.h] *vec3 utility functions*

3.3. Color Utility Functions

Using our new `vec3` class, we'll create a utility function to write a single pixel's color out to the standard output stream.

```
#ifndef COLOR_H
#define COLOR_H

#include "vec3.h"

#include <iostream>

void write_color(std::ostream &out, color pixel_color) {
    // Write the translated [0,255] value of each color component.
    out << static_cast<int>(255.999 * pixel_color.x()) << ' '
        << static_cast<int>(255.999 * pixel_color.y()) << ' '
        << static_cast<int>(255.999 * pixel_color.z()) << '\n';
}

#endif
```

Listing 6: [color.h] *color utility functions*

Now we can change our main to use this:

```
#include "color.h"
#include "vec3.h"

#include <iostream>

int main() {

    // Image

    const int image_width = 256;
    const int image_height = 256;

    // Render

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

    for (int j = image_height-1; j >= 0; --j) {
        std::cerr << "\rScanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            color pixel_color(double(i)/(image_width-1),
double(j)/(image_height-1), 0.25);
            write_color(std::cout, pixel_color);
        }
    }

    std::cerr << "\nDone.\n";
}
```

Listing 7: [main.cc] *Final code for the first PPM image*

4. Rays, a Simple Camera, and Background

4.1. The ray Class

The one thing that all ray tracers have is a ray class and a computation of what color is seen along a ray. Let' s think of a ray as a function $\mathbf{P}(t) = \mathbf{A} + t\mathbf{b}$. Here $\mathbf{P}$ is a 3D position along a line in 3D. $\mathbf{A}$ is the ray origin and $\mathbf{b}$ is the ray direction. The ray parameter t is a real number (`double` in the code). Plug in a different t and $\mathbf{P}(t)$ moves the point along the ray. Add in negative t values and you can go anywhere on the 3D line. For positive t, you get only the parts in front of $\mathbf{A}$, and this is what is often called a half-line or ray.

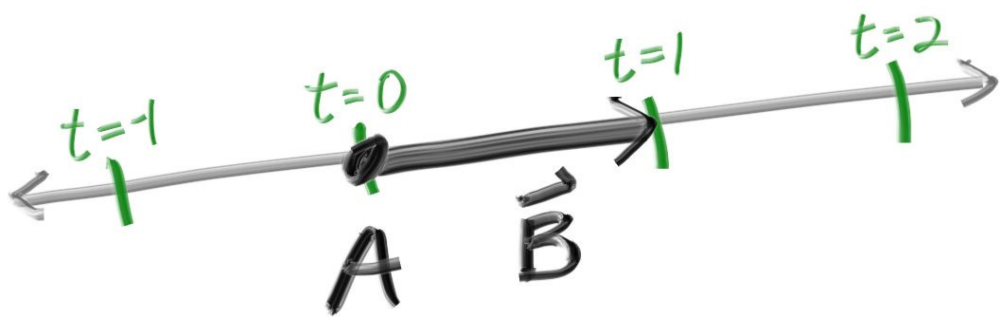


Figure 2: *Linear interpolation*

The function $\mathbf{P}(t)$ in more verbose code form I call `ray::at(t)`:

```
#ifndef RAY_H
#define RAY_H

#include "vec3.h"

class ray {
    public:
        ray() {}
        ray(const point3& origin, const vec3& direction)
            : orig(origin), dir(direction)
        {}

        point3 origin() const  { return orig; }
        vec3 direction() const { return dir; }

        point3 at(double t) const {
            return orig + t*dir;
        }

    public:
        point3 orig;
        vec3 dir;
};

#endif
```

Listing 8: [ray.h] *The ray class*

4.2. Sending Rays Into the Scene

Now we are ready to turn the corner and make a ray tracer. At the core, the ray tracer sends rays through pixels and computes the color seen in the direction of those rays. The involved steps are (1) calculate the ray from the eye to the pixel, (2) determine which objects the ray intersects, and (3) compute a color for that intersection point. When first developing a ray tracer, I always do a simple camera for getting the code up and running. I also make a simple `ray_color(ray)` function that returns the color of the background (a simple gradient).

I've often gotten into trouble using square images for debugging because I transpose x and y too often, so I'll use a non-square image. For now we'll use a 16:9 aspect ratio, since that's so common.

In addition to setting up the pixel dimensions for the rendered image, we also need to set up a virtual viewport through which to pass our scene rays. For the standard square pixel spacing, the viewport's aspect ratio should be the same as our rendered image. We'll just pick a viewport two units in height. We'll also set the distance between the projection plane and the projection point to be one unit. This is referred to as the "focal length", not to be confused with "focus distance", which we'll present later.

I'll put the "eye" (or camera center if you think of a camera) at $(0, 0, 0)$. I will have the y-axis go up, and the x-axis to the right. In order to respect the convention of a right handed coordinate system, into the screen is the negative z-axis. I will traverse the screen from the upper left hand corner, and use two offset vectors along the screen sides to move the ray endpoint across the screen. Note that I do not make the ray direction a unit length vector because I think not doing that makes for simpler and slightly faster code.

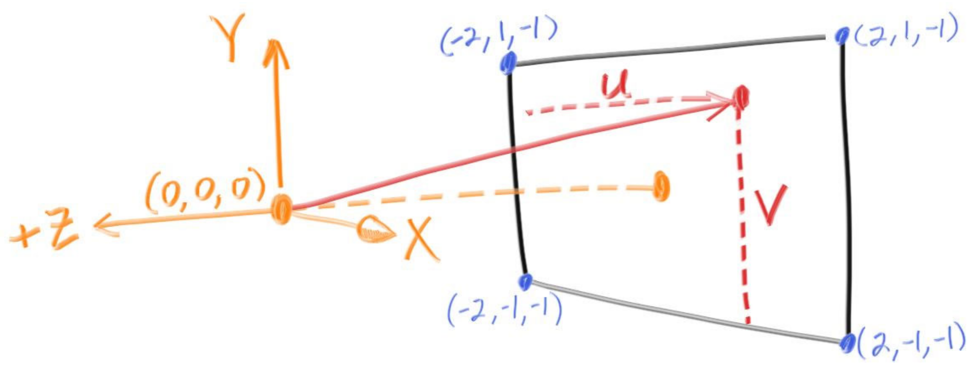


Figure 3: *Camera geometry*

Below in code, the ray `r` goes to approximately the pixel centers (I won' t worry about exactness for now because we' ll add antialiasing later):

```
#include "color.h"
#include "ray.h"
#include "vec3.h"

#include <iostream>

color ray_color(const ray& r) {
    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}

int main() {

    // Image
    const auto aspect_ratio = 16.0 / 9.0;
    const int image_width = 400;
    const int image_height = static_cast<int>(image_width / aspect_ratio);

    // Camera

    auto viewport_height = 2.0;
    auto viewport_width = aspect_ratio * viewport_height;
    auto focal_length = 1.0;

    auto origin = point3(0, 0, 0);
    auto horizontal = vec3(viewport_width, 0, 0);
    auto vertical = vec3(0, viewport_height, 0);
    auto lower_left_corner = origin - horizontal/2 - vertical/2 - vec3(0, 0,
focal_length);

    // Render

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

    for (int j = image_height-1; j >= 0; --j) {
        std::cerr << "\rScanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            auto u = double(i) / (image_width-1);
            auto v = double(j) / (image_height-1);
            ray r(origin, lower_left_corner + u*horizontal + v*vertical -
origin);
            color pixel_color = ray_color(r);
            write_color(std::cout, pixel_color);
        }
    }

    std::cerr << "\nDone.\n";
}
```

Listing 9: [main.cc] *Rendering a blue-to-white gradient*

The `ray_color(ray)` function linearly blends white and blue depending on the height of the y coordinate *after* scaling the ray direction to unit length (so $-1.0 < y < 1.0$). Because we're looking at the y height after normalizing the vector, you'll notice a horizontal gradient to the color in addition to the vertical gradient.

I then did a standard graphics trick of scaling that to $0.0 \leq t \leq 1.0$. When $t = 1.0$ I want blue. When $t = 0.0$ I want white. In between, I want a blend. This forms a "linear blend", or "linear interpolation", or "lerp" for short, between two things. A lerp is always of the form

$$\boxed{\text{blendedValue} = (1 - t) \cdot \text{startValue} + t \cdot \text{endValue},}$$

with t going from zero to one. In our case this produces:

Image 2: A blue-to-white gradient depending on ray Y coordinate

5. Adding a Sphere

Let's add a single object to our ray tracer. People often use spheres in ray tracers because calculating whether a ray hits a sphere is pretty straightforward.

5.1. Ray-Sphere Intersection

Recall that the equation for a sphere centered at the origin of radius R is $x^2 + y^2 + z^2 = R^2$. Put another way, if a given point (x, y, z) is on the sphere, then $x^2 + y^2 + z^2 = R^2$. If the given point (x, y, z) is *inside* the sphere, then $x^2 + y^2 + z^2 < R^2$, and if a given point (x, y, z) is *outside* the sphere, then $x^2 + y^2 + z^2 > R^2$.

It gets uglier if the sphere center is at (C_x, C_y, C_z):

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2$$

In graphics, you almost always want your formulas to be in terms of vectors so all the x/y/z stuff is under the hood in the `vec3` class. You might note that the vector from center $\mathbf{C} = (C_x, C_y, C_z)$ to point $\mathbf{P} = (x, y, z)$ is $(\mathbf{P} - \mathbf{C})$, and therefore

$$(\mathbf{P} - \mathbf{C}) \cdot (\mathbf{P} - \mathbf{C}) = (x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2$$

So the equation of the sphere in vector form is:

$$(\mathbf{P} - \mathbf{C}) \cdot (\mathbf{P} - \mathbf{C}) = r^2$$

We can read this as "any point $\mathbf{P}$ that satisfies this equation is on the sphere". We want to know if our ray $\mathbf{P}(t) = \mathbf{A} + t\mathbf{b}$ ever hits the sphere anywhere. If it does hit the sphere, there is some t for which $\mathbf{P}(t)$ satisfies the sphere equation. So we are looking for any t where this is true:

$$(\mathbf{P}(t) - \mathbf{C}) \cdot (\mathbf{P}(t) - \mathbf{C}) = r^2$$

or expanding the full form of the ray $\mathbf{P}(t)$:

$$\boxed{(\mathbf{A} + t\mathbf{b} - \mathbf{C}) \cdot (\mathbf{A} + t\mathbf{b} - \mathbf{C}) = r^2}$$

The rules of vector algebra are all that we would want here. If we expand that equation and move all the terms to the left hand side we get:

$$\boxed{t^2\mathbf{b} \cdot \mathbf{b} + 2t\mathbf{b} \cdot (\mathbf{A} - \mathbf{C}) + (\mathbf{A} - \mathbf{C}) \cdot (\mathbf{A} - \mathbf{C}) - r^2 = 0}$$

The vectors and r in that equation are all constant and known. The unknown is t, and the equation is a quadratic, like you probably saw in your high school math class. You can solve for t and there is a square root part that is either positive (meaning two real solutions), negative (meaning no real solutions), or zero (meaning one real solution). In graphics, the algebra almost always relates very directly to the geometry. What we have is:

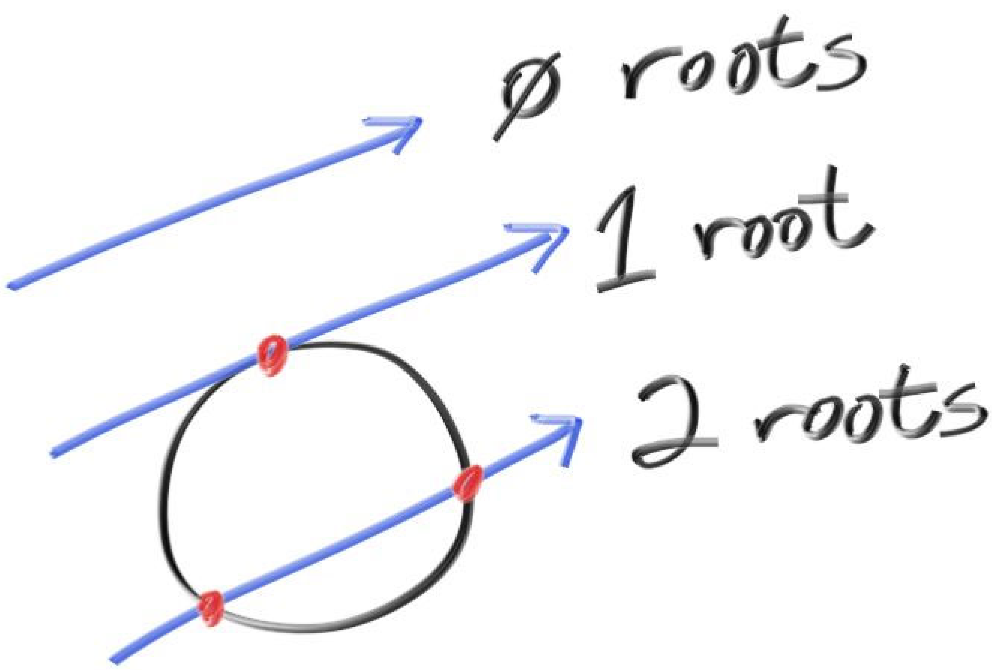


Figure 4: *Ray-sphere intersection results*

5.2. Creating Our First Raytraced Image

If we take that math and hard-code it into our program, we can test it by coloring red any pixel that hits a small sphere we place at −1 on the z-axis:

```
bool hit_sphere(const point3& center, double radius, const ray& r) {
    vec3 oc = r.origin() - center;
    auto a = dot(r.direction(), r.direction());
    auto b = 2.0 * dot(oc, r.direction());
    auto c = dot(oc, oc) - radius*radius;
    auto discriminant = b*b - 4*a*c;
    return (discriminant > 0);
}

color ray_color(const ray& r) {
    if (hit_sphere(point3(0,0,-1), 0.5, r))
        return color(1, 0, 0);
    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

Listing 10: [main.cc] *Rendering a red sphere*

What we get is this:

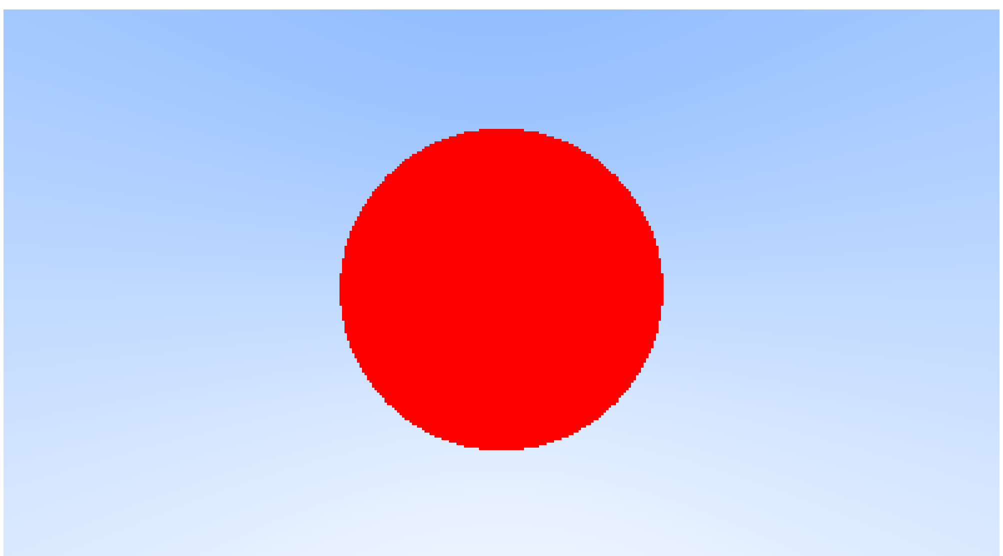


Image 3: A simple red sphere

Now this lacks all sorts of things — like shading and reflection rays and more than one object — but we are closer to halfway done than we are to our start! One thing to be aware of is that we tested whether the ray hits the sphere at all, but $t < 0$ solutions work fine. If you change your sphere center to $z = +1$ you will get exactly the same picture because you see the things behind you. This is not a feature! We'll fix those issues next.

6. Surface Normals and Multiple Objects

6.1. Shading with Surface Normals

First, let’s get ourselves a surface normal so we can shade. This is a vector that is perpendicular to the surface at the point of intersection. There are two design decisions to make for normals. The first is whether these normals are unit length. That is convenient for shading so I will say yes, but I won’t enforce that in the code. This could allow subtle bugs, so be aware this is personal preference as are most design decisions like that. For a sphere, the outward normal is in the direction of the hit point minus the center:

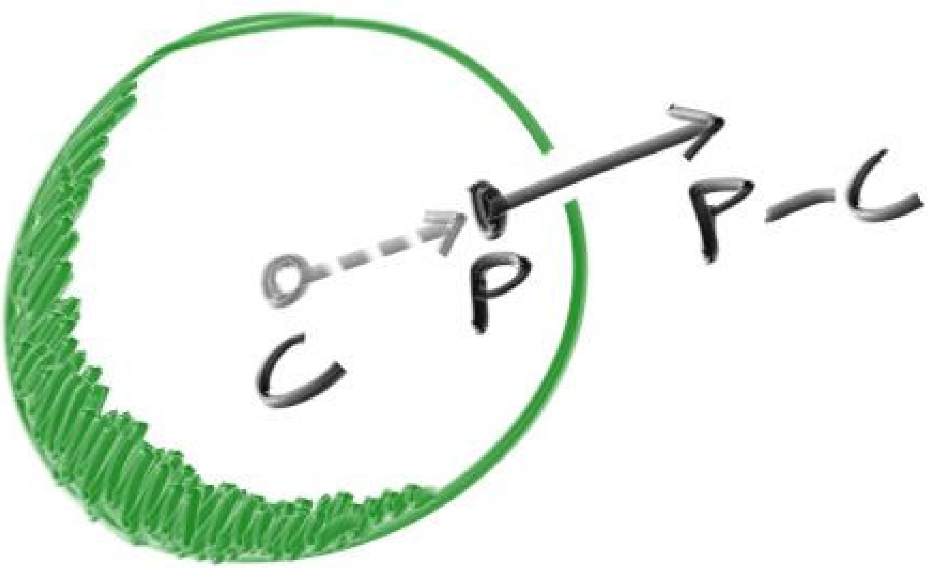


Figure 5: *Sphere surface-normal geometry*

On the earth, this implies that the vector from the earth’ s center to you points straight up. Let’ s throw that into the code now, and shade it. We don’ t have any lights or anything yet, so let’ s just visualize the normals with a color map. A common trick used for visualizing normals (because it’ s easy and somewhat intuitive to assume $\mathbf{n}$ is a unit length vector — so each component is between −1 and 1) is to map each component to the interval from 0 to 1, and then map x/y/z to r/g/b. For the normal, we need the hit point, not just whether we hit or not. We only have one sphere in the scene, and it's directly in front of the camera, so we won't worry about negative values of t yet. We'll just assume the closest hit point (smallest t). These changes in the code let us compute and visualize $\mathbf{n}$:

```
double hit_sphere(const point3& center, double radius, const ray& r) {
    vec3 oc = r.origin() - center;
    auto a = dot(r.direction(), r.direction());
    auto b = 2.0 * dot(oc, r.direction());
    auto c = dot(oc, oc) - radius*radius;
    auto discriminant = b*b - 4*a*c;
    if (discriminant < 0) {
        return -1.0;
    } else {
        return (-b - sqrt(discriminant) ) / (2.0*a);
    }
}

color ray_color(const ray& r) {
    auto t = hit_sphere(point3(0,0,-1), 0.5, r);
    if (t > 0.0) {
        vec3 N = unit_vector(r.at(t) - vec3(0,0,-1));
        return 0.5*color(N.x()+1, N.y()+1, N.z()+1);
    }
    vec3 unit_direction = unit_vector(r.direction());
    t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

Listing 11: [main.cc] *Rendering surface normals on a sphere*

And that yields this picture:

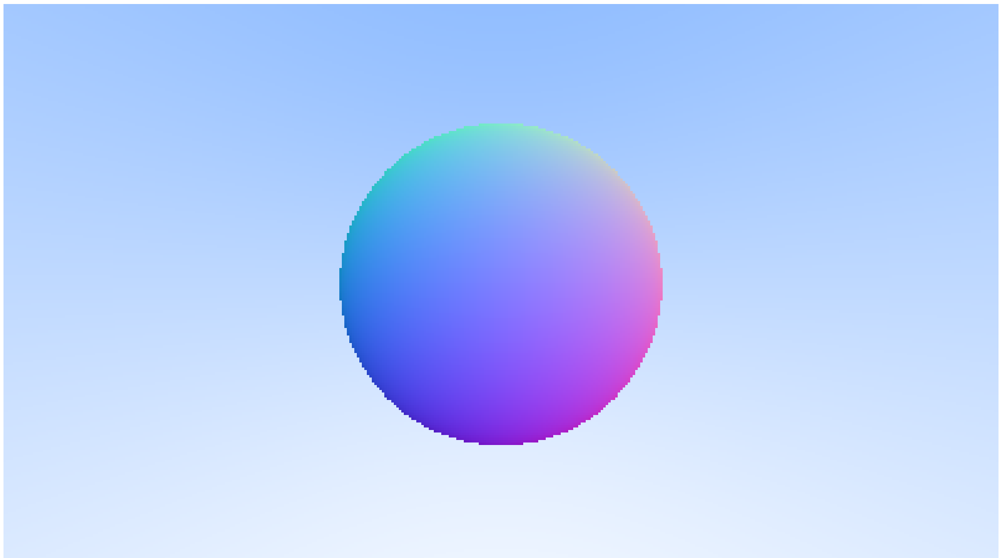


Image 4: A sphere colored according to its normal vectors

6.2. Simplifying the Ray-Sphere Intersection Code

Let' s revisit the ray-sphere equation:

```
double hit_sphere(const point3& center, double radius, const ray& r) {
    vec3 oc = r.origin() - center;
    auto a = dot(r.direction(), r.direction());
    auto b = 2.0 * dot(oc, r.direction());
    auto c = dot(oc, oc) - radius*radius;
    auto discriminant = b*b - 4*a*c;

    if (discriminant < 0) {
        return -1.0;
    } else {
        return (-b - sqrt(discriminant) ) / (2.0*a);
    }
}
```

Listing 12: [main.cc] *Ray-sphere intersection code (before)*

First, recall that a vector dotted with itself is equal to the squared length of that vector.

Second, notice how the equation for `b` has a factor of two in it. Consider what happens to the quadratic equation if $b = 2h$:

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

$$= \frac{-2h \pm \sqrt{(2h)^2 - 4ac}}{2a}$$

$$= \frac{-2h \pm 2\sqrt{h^2 - ac}}{2a}$$

$$= \frac{-h \pm \sqrt{h^2 - ac}}{a}$$

Using these observations, we can now simplify the sphere-intersection code to this:

```
double hit_sphere(const point3& center, double radius, const ray& r) {
    vec3 oc = r.origin() - center;
    auto a = r.direction().length_squared();
    auto half_b = dot(oc, r.direction());
    auto c = oc.length_squared() - radius*radius;
    auto discriminant = half_b*half_b - a*c;

    if (discriminant < 0) {
        return -1.0;
    } else {
        return (-half_b - sqrt(discriminant) ) / a;
    }
}
```

Listing 13: [main.cc] *Ray-sphere intersection code (after)*

6.3. An Abstraction for Hittable Objects

Now, how about several spheres? While it is tempting to have an array of spheres, a very clean solution is the make an "abstract class" for anything a ray might hit, and make both a sphere and a list of spheres just something you can hit. What that class should be called is something of a quandary — calling it an "object" would be good if not for "object oriented" programming. "Surface" is often used, with the weakness being maybe we will want volumes. "hittable" emphasizes the member function that unites them. I don't love any of these, but I will go with "hittable".

This `hittable` abstract class will have a hit function that takes in a ray. Most ray tracers have found it convenient to add a valid interval for hits t_{min} to t_{max}, so the hit only "counts" if $t_{min} < t < t_{max}$. For the initial rays this is positive t, but as we will see, it can help some details in the code to have an interval t_{min} to t_{max}. One design question is whether to do things like compute the normal if we hit something. We might end up hitting something closer as we do our search, and we will only need the normal of the closest thing. I will go with the simple solution and compute a bundle of stuff I will store in some structure. Here' s the abstract class:

```
#ifndef HITTABLE_H
#define HITTABLE_H

#include "ray.h"

struct hit_record {
    point3 p;
    vec3 normal;
    double t;
};

class hittable {
    public:
        virtual bool hit(const ray& r, double t_min, double t_max, hit_record&
rec) const = 0;
};

#endif
```

Listing 14: [hittable.h] *The hittable class*

And here’ s the sphere:

```
#ifndef SPHERE_H
#define SPHERE_H

#include "hittable.h"
#include "vec3.h"

class sphere : public hittable {
    public:
        sphere() {}
        sphere(point3 cen, double r) : center(cen), radius(r) {};

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

    public:
        point3 center;
        double radius;
};

bool sphere::hit(const ray& r, double t_min, double t_max, hit_record& rec)
const {
    vec3 oc = r.origin() - center;
    auto a = r.direction().length_squared();
    auto half_b = dot(oc, r.direction());
    auto c = oc.length_squared() - radius*radius;

    auto discriminant = half_b*half_b - a*c;
    if (discriminant < 0) return false;
    auto sqrtd = sqrt(discriminant);

    // Find the nearest root that lies in the acceptable range.
    auto root = (-half_b - sqrtd) / a;
    if (root < t_min || t_max < root) {
        root = (-half_b + sqrtd) / a;
        if (root < t_min || t_max < root)
            return false;
    }

    rec.t = root;
    rec.p = r.at(rec.t);
    rec.normal = (rec.p - center) / radius;

    return true;
}

#endif
```

Listing 15: [sphere.h] *The sphere class*

6.4. Front Faces Versus Back Faces

The second design decision for normals is whether they should always point out. At present, the normal found will always be in the direction of the center to the intersection point (the normal points out). If the ray intersects the sphere from the outside, the normal points against the ray. If the ray intersects the sphere from the inside, the normal (which always points out) points with the ray. Alternatively, we can have the normal always point against the ray. If the ray is outside the sphere, the normal will point outward, but if the ray is inside the sphere, the normal will point inward.

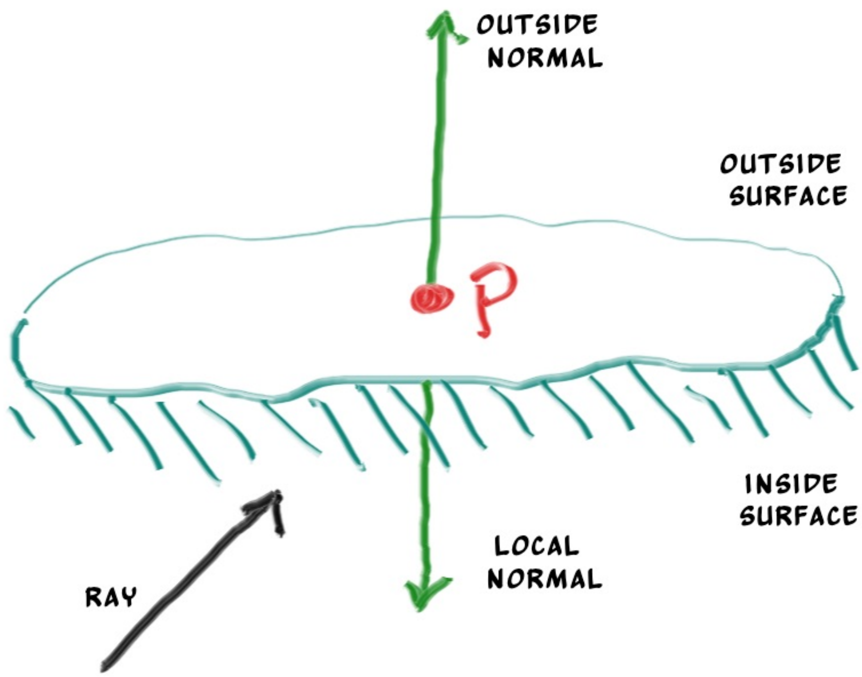


Figure 6: *Possible directions for sphere surface-normal geometry*

We need to choose one of these possibilities because we will eventually want to determine which side of the surface that the ray is coming from. This is important for objects that are rendered differently on each side, like the text on a two-sided sheet of paper, or for objects that have an inside and an outside, like glass balls.

If we decide to have the normals always point out, then we will need to determine which side the ray is on when we color it. We can figure this out by comparing the ray with the normal. If the ray and the normal face in the same direction, the ray is inside the object, if the ray and the normal face in the opposite direction, then the ray is outside the object. This can be determined by taking the dot product of the two vectors, where if their dot is positive, the ray is inside the sphere.

```
if (dot(ray_direction, outward_normal) > 0.0) {
    // ray is inside the sphere
    ...
} else {
    // ray is outside the sphere
    ...
}
```

Listing 16: *Comparing the ray and the normal*

If we decide to have the normals always point against the ray, we won't be able to use the dot product to determine which side of the surface the ray is on. Instead, we would need to store that information:

```
bool front_face;
if (dot(ray_direction, outward_normal) > 0.0) {
    // ray is inside the sphere
    normal = -outward_normal;
    front_face = false;
} else {
    // ray is outside the sphere
    normal = outward_normal;
    front_face = true;
}
```

Listing 17: *Remembering the side of the surface*

We can set things up so that normals always point "outward" from the surface, or always point against the incident ray. This decision is determined by whether you want to determine the side of the surface at the time of geometry intersection or at the time of coloring. In this book we have more material types than we have geometry types, so we'll go for less work and put the determination at geometry time. This is simply a matter of preference, and you'll see both implementations in the literature.

We add the `front_face` bool to the `hit_record` struct. We'll also add a function to solve this calculation for us.

```
struct hit_record {
    point3 p;
    vec3 normal;
    double t;
    bool front_face;

    inline void set_face_normal(const ray& r, const vec3& outward_normal) {
        front_face = dot(r.direction(), outward_normal) < 0;
        normal = front_face ? outward_normal :-outward_normal;
    }
};
```

Listing 18: [hittable.h] *Adding front-face tracking to hit_record*

And then we add the surface side determination to the class:

```
bool sphere::hit(const ray& r, double t_min, double t_max, hit_record& rec)
const {
    ...

    rec.t = root;
    rec.p = r.at(rec.t);
    vec3 outward_normal = (rec.p - center) / radius;
    rec.set_face_normal(r, outward_normal);

    return true;
}
```

Listing 19: [sphere.h] *The sphere class with normal determination*

6.5. A List of Hittable Objects

We have a generic object called a `hittable` that the ray can intersect with. We now add a class that stores a list of `hittables`:

```
#ifndef HITTABLE_LIST_H
#define HITTABLE_LIST_H

#include "hittable.h"

#include <memory>
#include <vector>

using std::shared_ptr;
using std::make_shared;

class hittable_list : public hittable {
    public:
        hittable_list() {}
        hittable_list(shared_ptr<hittable> object) { add(object); }

        void clear() { objects.clear(); }
        void add(shared_ptr<hittable> object) { objects.push_back(object); }

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

    public:
        std::vector<shared_ptr<hittable>> objects;
};

bool hittable_list::hit(const ray& r, double t_min, double t_max, hit_record&
rec) const {
    hit_record temp_rec;
    bool hit_anything = false;
    auto closest_so_far = t_max;

    for (const auto& object : objects) {
        if (object->hit(r, t_min, closest_so_far, temp_rec)) {
            hit_anything = true;
            closest_so_far = temp_rec.t;
            rec = temp_rec;
        }
    }

    return hit_anything;
}

#endif
```

Listing 20: [hittable_list.h] *The hittable_list class*

6.6. Some New C++ Features

The `hittable_list` class code uses two C++ features that may trip you up if you're not normally a C++ programmer: `vector` and `shared_ptr`.

`shared_ptr<type>` is a pointer to some allocated type, with reference-counting semantics. Every time you assign its value to another shared pointer (usually with a simple assignment), the reference count is incremented. As shared pointers go out of scope (like at the end of a block or function), the reference count is decremented. Once the count goes to zero, the object is deleted.

Typically, a shared pointer is first initialized with a newly-allocated object, something like this:

```
shared_ptr<double> double_ptr = make_shared<double>(0.37);
shared_ptr<vec3>   vec3_ptr   = make_shared<vec3>(1.414214, 2.718281,
1.618034);
shared_ptr<sphere> sphere_ptr = make_shared<sphere>(point3(0,0,0), 1.0);
```

Listing 21: *An example allocation using shared_ptr*

`make_shared<thing>(thing_constructor_params ...)` allocates a new instance of type `thing`, using the constructor parameters. It returns a `shared_ptr<thing>`.

Since the type can be automatically deduced by the return type of `make_shared<type>(...)`, the above lines can be more simply expressed using C++'s `auto` type specifier:

```
auto double_ptr = make_shared<double>(0.37);
auto vec3_ptr   = make_shared<vec3>(1.414214, 2.718281, 1.618034);
auto sphere_ptr = make_shared<sphere>(point3(0,0,0), 1.0);
```

Listing 22: *An example allocation using shared_ptr with auto type*

We'll use shared pointers in our code, because it allows multiple geometries to share a common instance (for example, a bunch of spheres that all use the same texture map material), and because it makes memory management automatic and easier to reason about.

`std::shared_ptr` is included with the `<memory>` header.

The second C++ feature you may be unfamiliar with is `std::vector`. This is a generic array-like collection of an arbitrary type. Above, we use a collection of pointers to `hittable`. `std::vector` automatically grows as more values are added: `objects.push_back(object)` adds a value to the end of the `std::vector` member variable `objects`.

`std::vector` is included with the `<vector>` header.

Finally, the `using` statements in listing 20 tell the compiler that we'll be getting `shared_ptr` and `make_shared` from the `std` library, so we don't need to prefex these with `std::` every time we reference them.

6.7. Common Constants and Utility Functions

We need some math constants that we conveniently define in their own header file. For now we only need infinity, but we will also throw our own definition of pi in there, which we will need later. There is no standard portable definition of pi, so we just define our own constant for it. We'll throw common useful constants and future utility functions in `rtweekend.h`, our general main header file.

```
#ifndef RTWEEKEND_H
#define RTWEEKEND_H

#include <cmath>
#include <limits>
#include <memory>


// Usings

using std::shared_ptr;
using std::make_shared;
using std::sqrt;

// Constants

const double infinity = std::numeric_limits<double>::infinity();
const double pi = 3.1415926535897932385;

// Utility Functions

inline double degrees_to_radians(double degrees) {
    return degrees * pi / 180.0;
}

// Common Headers

#include "ray.h"
#include "vec3.h"

#endif
```

Listing 23: [rtweekend.h] *The rtweekend.h common header*

And the new main:

```
#include "rtweekend.h"

#include "color.h"
#include "hittable_list.h"
#include "sphere.h"

#include <iostream>
color ray_color(const ray& r, const hittable& world) {
    hit_record rec;
    if (world.hit(r, 0, infinity, rec)) {
        return 0.5 * (rec.normal + color(1,1,1));
    }
    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}

int main() {

    // Image

    const auto aspect_ratio = 16.0 / 9.0;
    const int image_width = 400;
    const int image_height = static_cast<int>(image_width / aspect_ratio);

    // World
    hittable_list world;
    world.add(make_shared<sphere>(point3(0,0,-1), 0.5));
    world.add(make_shared<sphere>(point3(0,-100.5,-1), 100));

    // Camera

    auto viewport_height = 2.0;
    auto viewport_width = aspect_ratio * viewport_height;
    auto focal_length = 1.0;

    auto origin = point3(0, 0, 0);
    auto horizontal = vec3(viewport_width, 0, 0);
    auto vertical = vec3(0, viewport_height, 0);
    auto lower_left_corner = origin - horizontal/2 - vertical/2 - vec3(0, 0,
focal_length);

    // Render

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

    for (int j = image_height-1; j >= 0; --j) {
        std::cerr << "\rScanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            auto u = double(i) / (image_width-1);
            auto v = double(j) / (image_height-1);
            ray r(origin, lower_left_corner + u*horizontal + v*vertical);
            color pixel_color = ray_color(r, world);
            write_color(std::cout, pixel_color);
        }
    }
```

```
    std::cerr << "\nDone.\n";
}
```

Listing 24: [main.cc] *The new main with hittables*

This yields a picture that is really just a visualization of where the spheres are along with their surface normal. This is often a great way to look at your model for flaws and characteristics.

Image 5: Resulting render of normals-colored sphere with ground

7. Antialiasing

When a real camera takes a picture, there are usually no jaggies along edges because the edge pixels are a blend of some foreground and some background. We can get the same effect by averaging a bunch of samples inside each pixel. We will not bother with stratification. This is controversial, but is usual for my programs. For some ray tracers it is critical, but the kind of general one we are writing doesn’t benefit very much from it and it makes the code uglier. We abstract the camera class a bit so we can make a cooler camera later.

7.1. Some Random Number Utilities

One thing we need is a random number generator that returns real random numbers. We need a function that returns a canonical random number which by convention returns a random real in the range $0 \leq r < 1$. The “less than” before the 1 is important as we will sometimes take advantage of that.

A simple approach to this is to use the `rand()` function that can be found in `<cstdlib>`. This function returns a random integer in the range 0 and `RAND_MAX`. Hence we can get a real random number as desired with the following code snippet, added to `rtweekend.h`:

```
#include <cstdlib>
...

inline double random_double() {
    // Returns a random real in [0,1).
    return rand() / (RAND_MAX + 1.0);
}

inline double random_double(double min, double max) {
    // Returns a random real in [min,max).
    return min + (max-min)*random_double();
}
```

Listing 25: [rtweekend.h] *random_double() functions*

C++ did not traditionally have a standard random number generator, but newer versions of C++ have addressed this issue with the `<random>` header (if imperfectly according to some experts). If you want to use this, you can obtain a random number with the conditions we need as follows:

```
#include <random>

inline double random_double() {
    static std::uniform_real_distribution<double> distribution(0.0, 1.0);
    static std::mt19937 generator;
    return distribution(generator);
}
```

Listing 26: [rtweekend.h] *random_double(), alternate implemenation*

7.2. Generating Pixels with Multiple Samples

For a given pixel we have several samples within that pixel and send rays through each of the samples. The colors of these rays are then averaged:

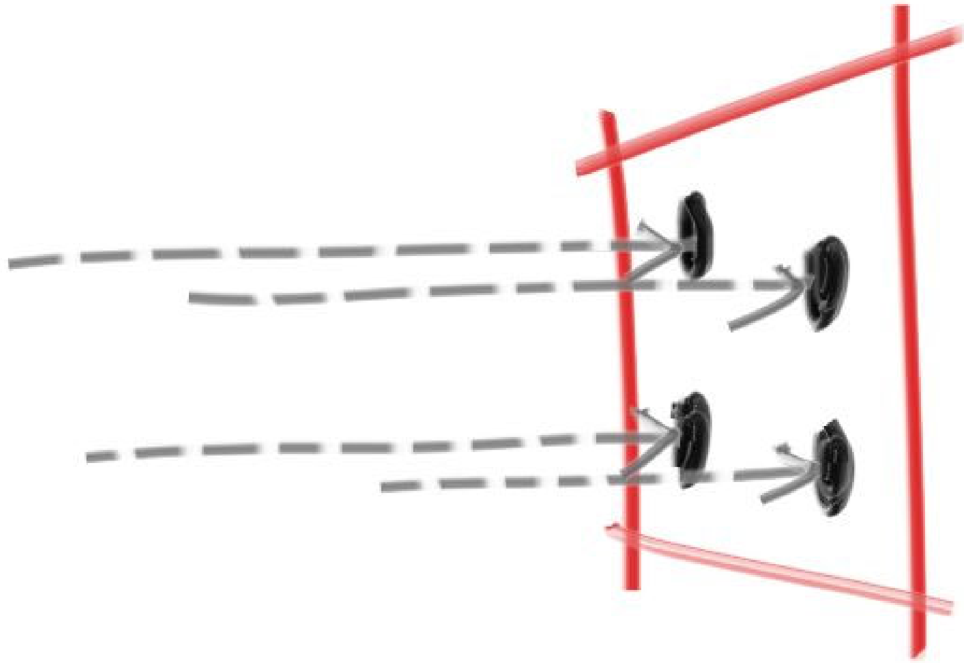


Figure 7: *Pixel samples*

Now's a good time to create a `camera` class to manage our virtual camera and the related tasks of scene scampling. The following class implements a simple camera using the axis-aligned camera from before:

```
#ifndef CAMERA_H
#define CAMERA_H

#include "rtweekend.h"

class camera {
    public:
        camera() {
            auto aspect_ratio = 16.0 / 9.0;
            auto viewport_height = 2.0;
            auto viewport_width = aspect_ratio * viewport_height;
            auto focal_length = 1.0;

            origin = point3(0, 0, 0);
            horizontal = vec3(viewport_width, 0.0, 0.0);
            vertical = vec3(0.0, viewport_height, 0.0);
            lower_left_corner = origin - horizontal/2 - vertical/2 - vec3(0, 0,
focal_length);
        }

        ray get_ray(double u, double v) const {
            return ray(origin, lower_left_corner + u*horizontal + v*vertical -
origin);
        }

    private:
        point3 origin;
        point3 lower_left_corner;
        vec3 horizontal;
        vec3 vertical;
};
#endif
```

Listing 27: [camera.h] *The camera class*

To handle the multi-sampled color computation, we'll update the `write_color()` function. Rather than adding in a fractional contribution each time we accumulate more light to the color, just add the full color each iteration, and then perform a single divide at the end (by the number of samples) when writing out the color. In addition, we'll add a handy utility function to the `rtweekend.h` utility header: `clamp(x,min,max)`, which clamps the value `x` to the range [min,max]:

```
inline double clamp(double x, double min, double max) {
    if (x < min) return min;
    if (x > max) return max;
    return x;
}
```

Listing 28: [rtweekend.h] *The clamp() utility function*

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

    // Divide the color by the number of samples.
    auto scale = 1.0 / samples_per_pixel;
    r *= scale;
    g *= scale;
    b *= scale;

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

Listing 29: [color.h] *The multi-sample write_color() function*

Main is also changed:

```
#include "camera.h"

...

int main() {

    // Image

    const auto aspect_ratio = 16.0 / 9.0;
    const int image_width = 400;
    const int image_height = static_cast<int>(image_width / aspect_ratio);
    const int samples_per_pixel = 100;

    // World

    hittable_list world;
    world.add(make_shared<sphere>(point3(0,0,-1), 0.5));
    world.add(make_shared<sphere>(point3(0,-100.5,-1), 100));

    // Camera
    camera cam;

    // Render

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

    for (int j = image_height-1; j >= 0; --j) {
        std::cerr << "\rScanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            color pixel_color(0, 0, 0);
            for (int s = 0; s < samples_per_pixel; ++s) {
                auto u = (i + random_double()) / (image_width-1);
                auto v = (j + random_double()) / (image_height-1);
                ray r = cam.get_ray(u, v);
                pixel_color += ray_color(r, world);
            }
            write_color(std::cout, pixel_color, samples_per_pixel);
        }
    }

    std::cerr << "\nDone.\n";
}
```

Listing 30: [main.cc] *Rendering with multi-sampled pixels*

Zooming into the image that is produced, we can see the difference in edge pixels.

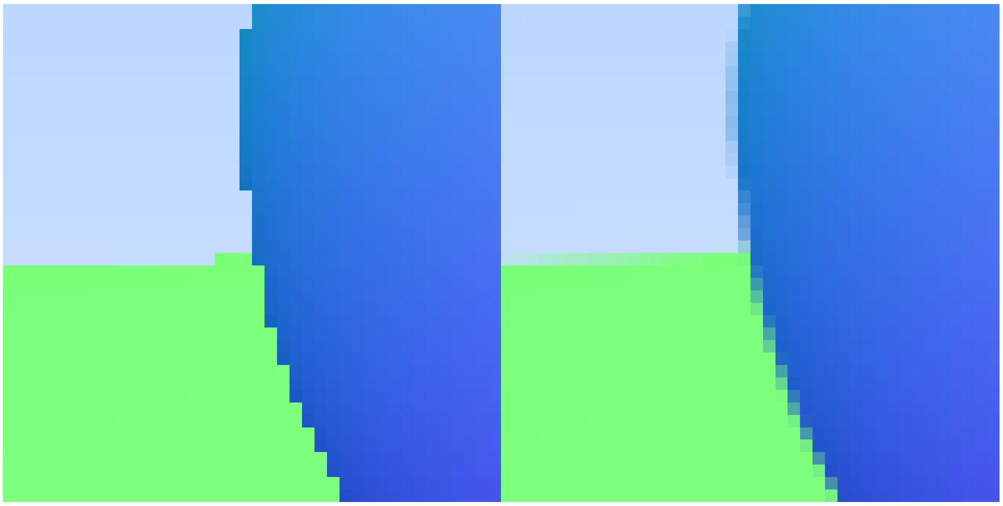


Image 6: Before and after antialiasing

8. Diffuse Materials

Now that we have objects and multiple rays per pixel, we can make some realistic looking materials. We' ll start with diffuse (matte) materials. One question is whether we mix and match geometry and materials (so we can assign a material to multiple spheres, or vice versa) or if geometry and material are tightly bound (that could be useful for procedural objects where the geometry and material are linked). We' ll go with separate — which is usual in most renderers — but do be aware of the limitation.

8.1. A Simple Diffuse Material

Diffuse objects that don’t emit light merely take on the color of their surroundings, but they modulate that with their own intrinsic color. Light that reflects off a diffuse surface has its direction randomized. So, if we send three rays into a crack between two diffuse surfaces they will each have different random behavior:

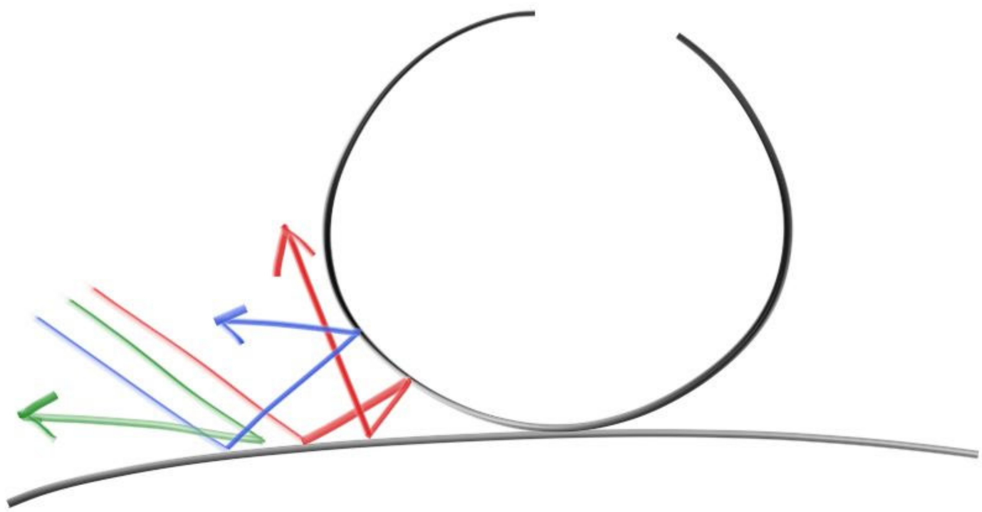


Figure 8: *Light ray bounces*

They also might be absorbed rather than reflected. The darker the surface, the more likely absorption is. (That’s why it is dark!) Really any algorithm that randomizes direction will produce surfaces that look matte. One of the simplest ways to do this turns out to be exactly correct for ideal diffuse surfaces. (I used to do it as a lazy hack that approximates mathematically ideal Lambertian.)

(Reader Vassillen Chizhov proved that the lazy hack is indeed just a lazy hack and is inaccurate. The correct representation of ideal Lambertian isn't much more work, and is presented at the end of the chapter.)

There are two unit radius spheres tangent to the hit point p of a surface. These two spheres have a center of $(\mathbf{P}+\mathbf{n})$ and $(\mathbf{P}-\mathbf{n})$, where $\mathbf{n}$ is the normal of the surface. The sphere with a center at $(\mathbf{P}-\mathbf{n})$ is considered *inside* the surface, whereas the sphere with center $(\mathbf{P}+\mathbf{n})$ is considered *outside* the surface. Select the tangent unit radius sphere that is on the same side of the surface as the ray origin. Pick a random point $\mathbf{S}$ inside this unit radius sphere and send a ray from the hit point $\mathbf{P}$ to the random point $\mathbf{S}$ (this is the vector $(\mathbf{S}-\mathbf{P})$):

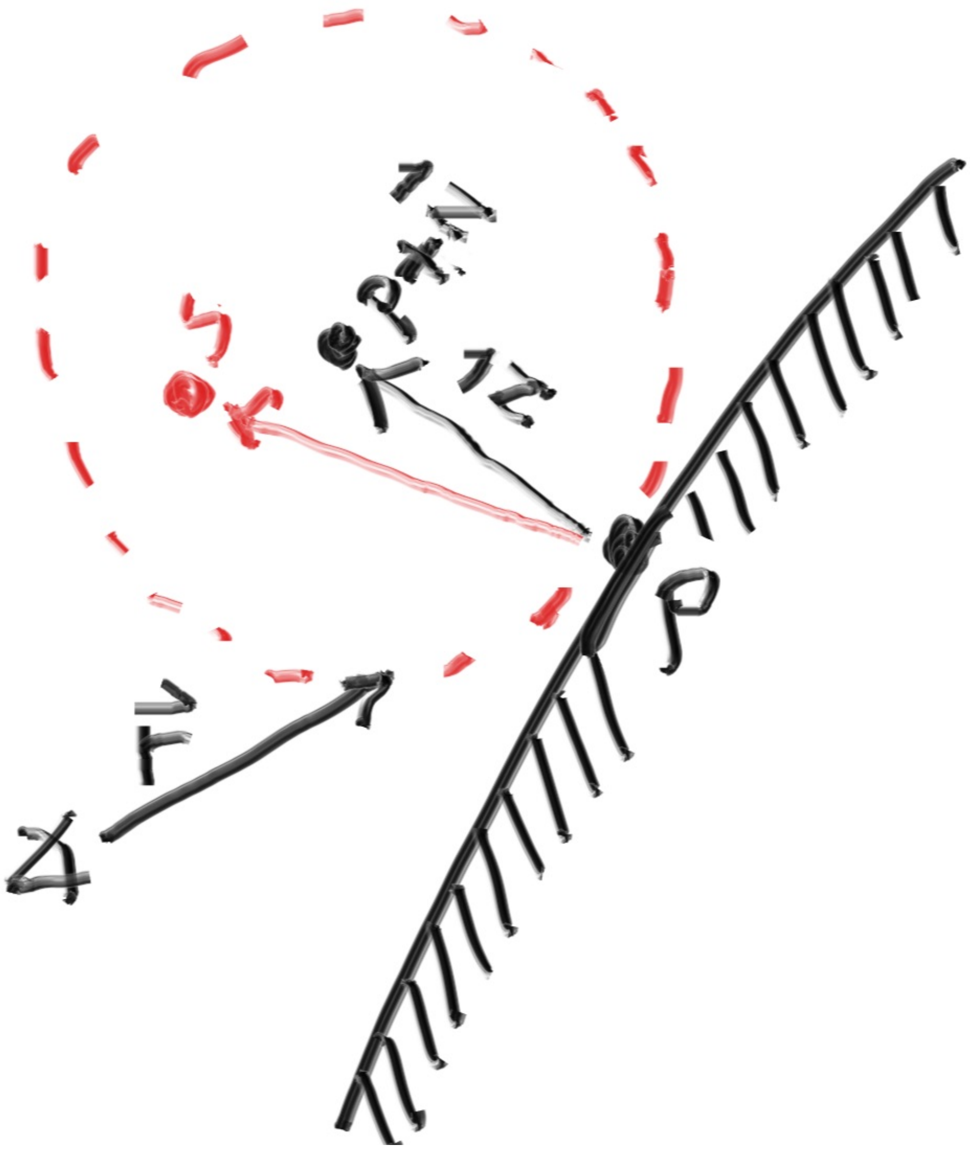


Figure 9: *Generating a random diffuse bounce ray*

We need a way to pick a random point in a unit radius sphere. We'll use what is usually the easiest algorithm: a rejection method. First, pick a random point in the unit cube where x, y, and z all range from −1 to +1. Reject this point and try again if the point is outside the sphere.

```
class vec3 {
  public:
    ...
    inline static vec3 random() {
        return vec3(random_double(), random_double(), random_double());
    }

    inline static vec3 random(double min, double max) {
        return vec3(random_double(min,max), random_double(min,max),
random_double(min,max));
    }
```

Listing 31: [vec3.h] *vec3 random utility functions*

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

Listing 32: [vec3.h] *The random_in_unit_sphere() function*

Then update the `ray_color()` function to use the new random direction generator:

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

    if (world.hit(r, 0, infinity, rec)) {
        point3 target = rec.p + rec.normal + random_in_unit_sphere();
        return 0.5 * ray_color(ray(rec.p, target - rec.p), world);
    }

    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

Listing 33: [main.cc] *ray_color() using a random ray direction*

8.2. Limiting the Number of Child Rays

There's one potential problem lurking here. Notice that the `ray_color` function is recursive. When will it stop recursing? When it fails to hit anything. In some cases, however, that may be a long time — long enough to blow the stack. To guard against that, let's limit the maximum recursion depth, returning no light contribution at the maximum depth:

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

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

    if (world.hit(r, 0, infinity, rec)) {
        point3 target = rec.p + rec.normal + random_in_unit_sphere();
        return 0.5 * ray_color(ray(rec.p, target - rec.p), world, depth-1);
    }

    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}

...

int main() {

    // Image

    const auto aspect_ratio = 16.0 / 9.0;
    const int image_width = 400;
    const int image_height = static_cast<int>(image_width / aspect_ratio);
    const int samples_per_pixel = 100;
    const int max_depth = 50;
    ...

    // Render

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

    for (int j = image_height-1; j >= 0; --j) {
        std::cerr << "\rScanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            color pixel_color(0, 0, 0);
            for (int s = 0; s < samples_per_pixel; ++s) {
                auto u = (i + random_double()) / (image_width-1);
                auto v = (j + random_double()) / (image_height-1);
                ray r = cam.get_ray(u, v);
                pixel_color += ray_color(r, world, max_depth);
            }
            write_color(std::cout, pixel_color, samples_per_pixel);
        }
    }

    std::cerr << "\nDone.\n";
}
```

Listing 34: [main.cc] *ray_color() with depth limiting*

This gives us:

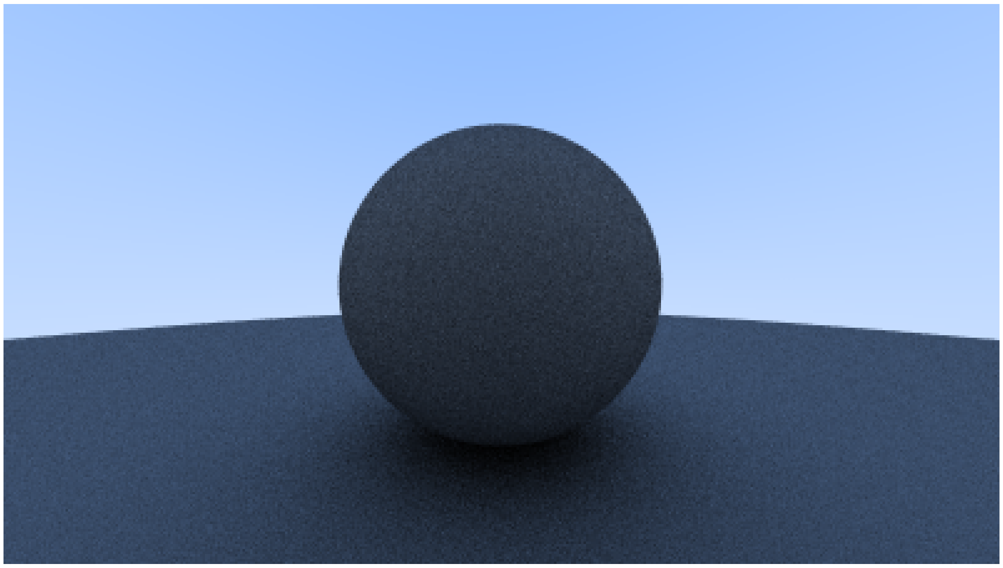


Image 7: First render of a diffuse sphere

8.3. Using Gamma Correction for Accurate Color Intensity

Note the shadowing under the sphere. This picture is very dark, but our spheres only absorb half the energy on each bounce, so they are 50% reflectors. If you can't see the shadow, don't worry, we will fix that now. These spheres should look pretty light (in real life, a light grey). The reason for this is that almost all image viewers assume that the image is "gamma corrected", meaning the 0 to 1 values have some transform before being stored as a byte. There are many good reasons for that, but for our purposes we just need to be aware of it. To a first approximation, we can use "gamma 2" which means raising the color to the power $1/gamma$, or in our simple case ½, which is just square-root:

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

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

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

Listing 35: [color.h] *write_color(), with gamma correction*

That yields light grey, as we desire:

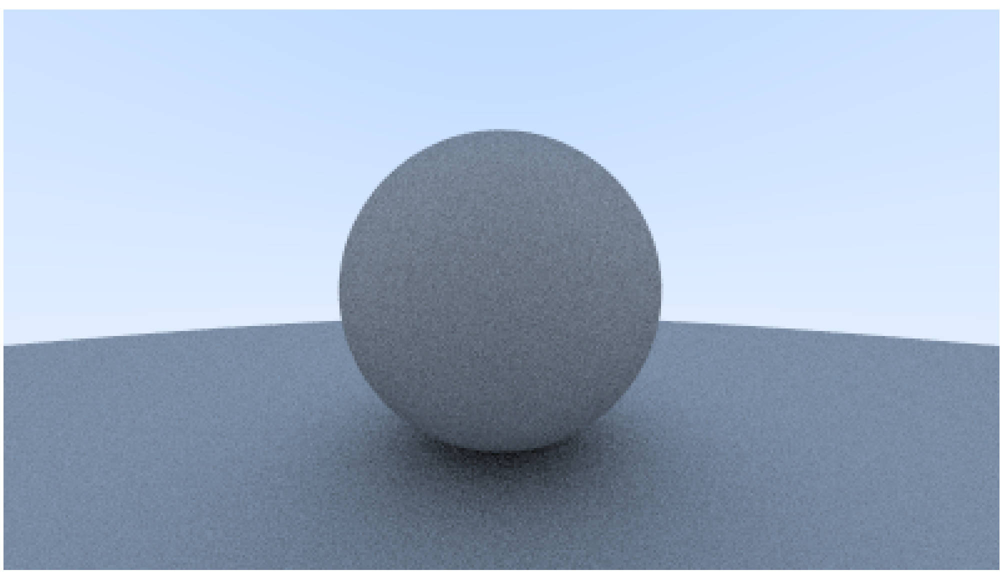


Image 8: Diffuse sphere, with gamma correction

8.4. Fixing Shadow Acne

There’ s also a subtle bug in there. Some of the reflected rays hit the object they are reflecting off of not at exactly $t = 0$, but instead at $t = -0.0000001$ or $t = 0.00000001$ or whatever floating point approximation the sphere intersector gives us. So we need to ignore hits very near zero:

```
if (world.hit(r, 0.001, infinity, rec)) {
```

Listing 36: [main.cc] *Calculating reflected ray origins with tolerance*

This gets rid of the shadow acne problem. Yes it is really called that.

8.5. True Lambertian Reflection

The rejection method presented here produces random points in the unit ball offset along the surface normal. This corresponds to picking directions on the hemisphere with high probability close to the normal, and a lower probability of scattering rays at grazing angles. This distribution scales by the $\cos^3(\phi)$ where ϕ is the angle from the normal. This is useful since light arriving at shallow angles spreads over a larger area, and thus has a lower contribution to the final color.

However, we are interested in a Lambertian distribution, which has a distribution of $\cos(\phi)$. True Lambertian has the probability higher for ray scattering close to the normal, but the distribution is more uniform. This is achieved by picking random points on the surface of the unit sphere, offset along the surface normal. Picking random points on the unit sphere can be achieved by picking random points *in* the unit sphere, and then normalizing those.

```
inline vec3 random_in_unit_sphere() {
    ...
}
vec3 random_unit_vector() {
    return unit_vector(random_in_unit_sphere());
}
```

Listing 37: [vec3.h] *The random_unit_vector() function*

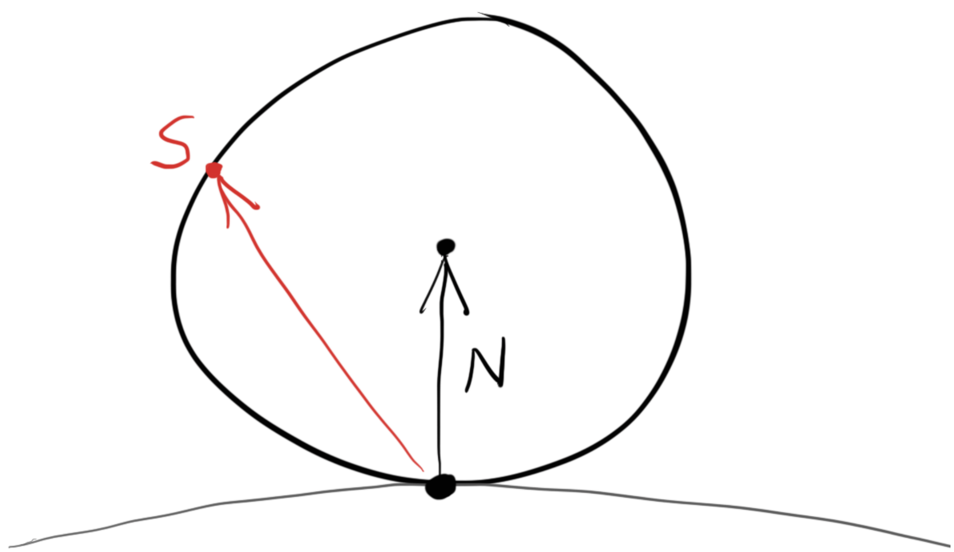


Figure 10: *Generating a random unit vector*

This `random_unit_vector()` is a drop-in replacement for the existing `random_in_unit_sphere()` function.

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

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

    if (world.hit(r, 0.001, infinity, rec)) {
        point3 target = rec.p + rec.normal + random_unit_vector();
        return 0.5 * ray_color(ray(rec.p, target - rec.p), world, depth-1);
    }

    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

Listing 38: [main.cc] *ray_color() with replacement diffuse*

After rendering we get a similar image:

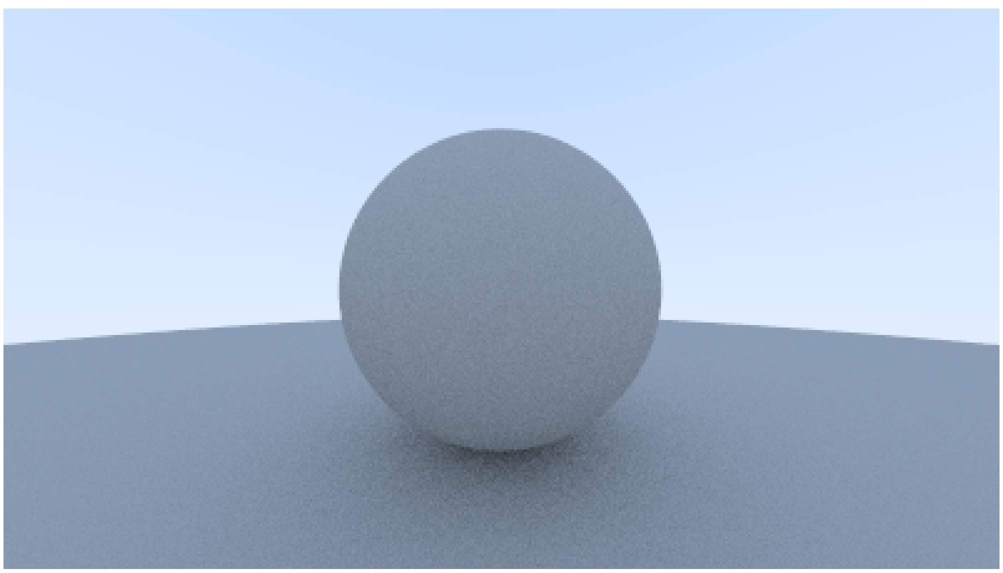


Image 9: Correct rendering of Lambertian spheres

It's hard to tell the difference between these two diffuse methods, given that our scene of two spheres is so simple, but you should be able to notice two important visual differences:

1. The shadows are less pronounced after the change
2. Both spheres are lighter in appearance after the change

Both of these changes are due to the more uniform scattering of the light rays, fewer rays are scattering toward the normal. This means that for diffuse objects, they will appear *lighter* because more light bounces toward the camera. For the shadows, less light bounces straight-up, so the parts of the larger sphere directly underneath the smaller sphere are brighter.

8.6. An Alternative Diffuse Formulation

The initial hack presented in this book lasted a long time before it was proven to be an incorrect approximation of ideal Lambertian diffuse. A big reason that the error persisted for so long is that it can be difficult to:

1. Mathematically prove that the probability distribution is incorrect
2. Intuitively explain why a $\cos(\phi)$ distribution is desirable (and what it would look like)

Not a lot of common, everyday objects are perfectly diffuse, so our visual intuition of how these objects behave under light can be poorly formed.

In the interest of learning, we are including an intuitive and easy to understand diffuse method. For the two methods above we had a random vector, first of random length and then of unit length, offset from the hit point by the normal. It may not be immediately obvious why the vectors should be displaced by the normal.

A more intuitive approach is to have a uniform scatter direction for all angles away from the hit point, with no dependence on the angle from the normal. Many of the first raytracing papers used this diffuse method (before adopting Lambertian diffuse).

```
vec3 random_in_hemisphere(const vec3& normal) {
    vec3 in_unit_sphere = random_in_unit_sphere();
    if (dot(in_unit_sphere, normal) > 0.0) // In the same hemisphere as the normal
        return in_unit_sphere;
    else
        return -in_unit_sphere;
}
```

Listing 39: [vec3.h] *The random_in_hemisphere(normal) function*

Plugging the new formula into the `ray_color()` function:

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

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

    if (world.hit(r, 0.001, infinity, rec)) {
        point3 target = rec.p + random_in_hemisphere(rec.normal);
        return 0.5 * ray_color(ray(rec.p, target - rec.p), world, depth-1);
    }

    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

Listing 40: [main.cc] *ray_color() with hemispherical scattering*

Gives us the following image:

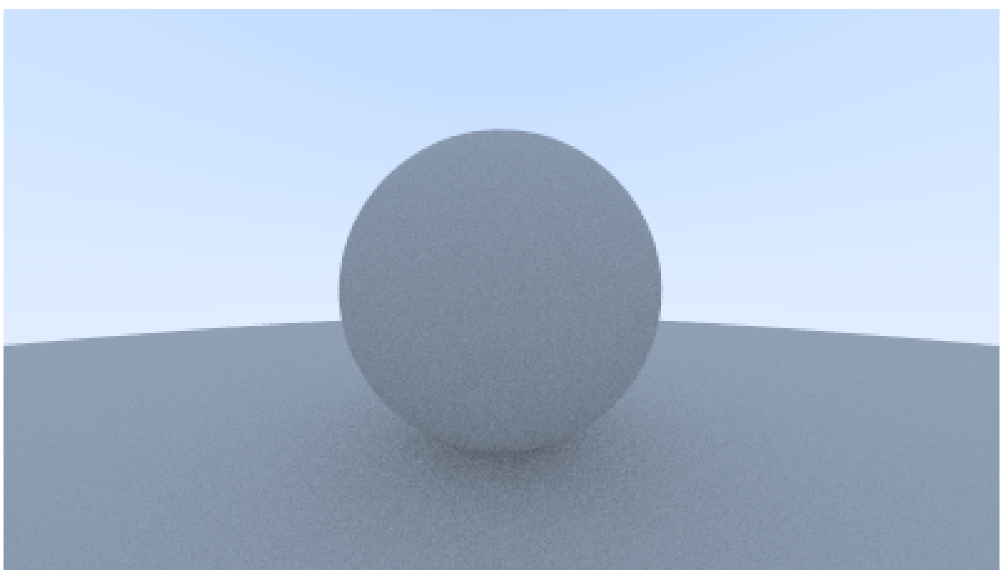


Image 10: Rendering of diffuse spheres with hemispherical scattering

Scenes will become more complicated over the course of the book. You are encouraged to switch between the different diffuse renderers presented here. Most scenes of interest will contain a disproportionate amount of diffuse materials. You can gain valuable insight by understanding the effect of different diffuse methods on the lighting of the scene.

9. Metal

9.1. An Abstract Class for Materials

If we want different objects to have different materials, we have a design decision. We could have a universal material with lots of parameters and different material types just zero out some of those parameters. This is not a bad approach. Or we could have an abstract material class that encapsulates behavior. I am a fan of the latter approach. For our program the material needs to do two things:

1. Produce a scattered ray (or say it absorbed the incident ray).
2. If scattered, say how much the ray should be attenuated.

This suggests the abstract class:

```
#ifndef MATERIAL_H
#define MATERIAL_H

#include "rtweekend.h"

struct hit_record;

class material {
    public:
        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray&
scattered
        ) const = 0;
};

#endif
```

Listing 41: [material.h] *The material class*

9.2. A Data Structure to Describe Ray-Object Intersections

The `hit_record` is to avoid a bunch of arguments so we can stuff whatever info we want in there. You can use arguments instead; it’ s a matter of taste. Hittables and materials need to know each other so there is some circularity of the references. In C++ you just need to alert the compiler that the pointer is to a class, which the “class material” in the hittable class below does:

```
#include "rtweekend.h"

class material;

struct hit_record {
    point3 p;
    vec3 normal;
    shared_ptr<material> mat_ptr;
    double t;
    bool front_face;

    inline void set_face_normal(const ray& r, const vec3& outward_normal) {
        front_face = dot(r.direction(), outward_normal) < 0;
        normal = front_face ? outward_normal :-outward_normal;
    }
};
```

Listing 42: [hittable.h] *Hit record with added material pointer*

What we have set up here is that material will tell us how rays interact with the surface. `hit_record` is just a way to stuff a bunch of arguments into a struct so we can send them as a group. When a ray hits a surface (a particular sphere for example), the material pointer in the `hit_record` will be set to point at the material pointer the sphere was given when it was set up in `main()` when we start. When the `ray_color()` routine gets the `hit_record` it can call member functions of the material pointer to find out what ray, if any, is scattered.

To achieve this, we must have a reference to the material for our sphere class to returned within `hit_record`. See the highlighted lines below:

```
class sphere : public hittable {
    public:
        sphere() {}
        sphere(point3 cen, double r, shared_ptr<material> m)
            : center(cen), radius(r), mat_ptr(m) {};

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

    public:
        point3 center;
        double radius;
        shared_ptr<material> mat_ptr;
};

bool sphere::hit(const ray& r, double t_min, double t_max, hit_record& rec)
const {
    ...

    rec.t = root;
    rec.p = r.at(rec.t);
    vec3 outward_normal = (rec.p - center) / radius;
    rec.set_face_normal(r, outward_normal);
    rec.mat_ptr = mat_ptr;

    return true;
}
```

Listing 43: [sphere.h] *Ray-sphere intersection with added material information*

9.3. Modeling Light Scatter and Reflectance

For the Lambertian (diffuse) case we already have, it can either scatter always and attenuate by its reflectance R, or it can scatter with no attenuation but absorb the fraction $1 - R$ of the rays, or it could be a mixture of those strategies. For Lambertian materials we get this simple class:

```
class lambertian : public material {
    public:
        lambertian(const color& a) : albedo(a) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray&
scattered
        ) const override {
            auto scatter_direction = rec.normal + random_unit_vector();
            scattered = ray(rec.p, scatter_direction);
            attenuation = albedo;
            return true;
        }

    public:
        color albedo;
};
```

Listing 44: [material.h] *The lambertian material class*

Note we could just as well only scatter with some probability p and have attenuation be $albedo/p$. Your choice.

If you read the code above carefully, you'll notice a small chance of mischief. If the random unit vector we generate is exactly opposite the normal vector, the two will sum to zero, which will result in a zero scatter direction vector. This leads to bad scenarios later on (infinities and NaNs), so we need to intercept the condition before we pass it on.

In service of this, we'll create a new vector method — `vec3::near_zero()` — that returns true if the vector is very close to zero in all dimensions.

```
class vec3 {
    ...
    bool near_zero() const {
        // Return true if the vector is close to zero in all dimensions.
        const auto s = 1e-8;
        return (fabs(e[0]) < s) && (fabs(e[1]) < s) && (fabs(e[2]) < s);
    }
    ...
};
```

Listing 45: [vec3.h] *The vec3::near_zero() method*

```
class lambertian : public material {
    public:
        lambertian(const color& a) : albedo(a) {}

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

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

            scattered = ray(rec.p, scatter_direction);
            attenuation = albedo;
            return true;
        }

    public:
        color albedo;
};
```

Listing 46: [material.h] *Lambertian scatter, bullet-proof*

9.4. Mirrored Light Reflection

For smooth metals the ray won’ t be randomly scattered. The key math is: how does a ray get reflected from a metal mirror? Vector math is our friend here:

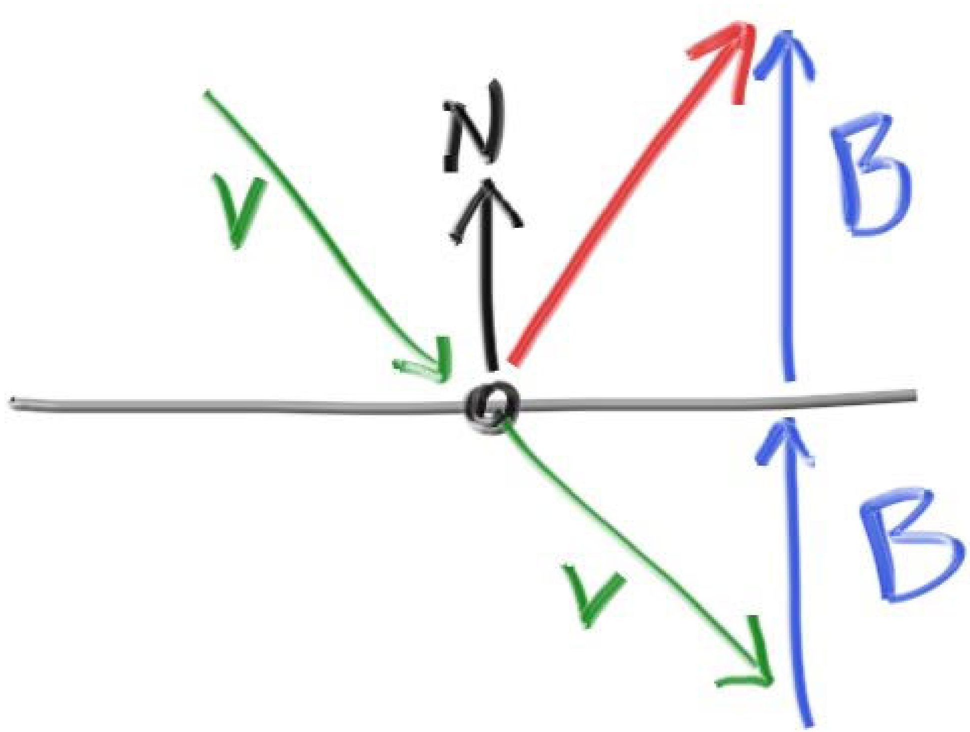


Figure 11: *Ray reflection*

The reflected ray direction in red is just $\mathbf{v}+2\mathbf{b}$. In our design, $\mathbf{n}$ is a unit vector, but $\mathbf{v}$ may not be. The length of $\mathbf{b}$ should be $\mathbf{v}\cdot\mathbf{n}$. Because $\mathbf{v}$ points in, we will need a minus sign, yielding:

```
vec3 reflect(const vec3& v, const vec3& n) {
    return v - 2*dot(v,n)*n;
}
```

Listing 47: [vec3.h] *vec3 reflection function*

The metal material just reflects rays using that formula:

```
class metal : public material {
    public:
        metal(const color& a) : albedo(a) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray&
scattered
        ) const override {
            vec3 reflected = reflect(unit_vector(r_in.direction()),
rec.normal);
            scattered = ray(rec.p, reflected);
            attenuation = albedo;
            return (dot(scattered.direction(), rec.normal) > 0);
        }

    public:
        color albedo;
};
```

Listing 48: [material.h] *Metal material with reflectance function*

We need to modify the `ray_color()` function to use this:

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

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

    if (world.hit(r, 0.001, infinity, rec)) {
        ray scattered;
        color attenuation;
        if (rec.mat_ptr->scatter(r, rec, attenuation, scattered))
            return attenuation * ray_color(scattered, world, depth-1);
        return color(0,0,0);
    }

    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

Listing 49: [main.cc] *Ray color with scattered reflectance*

9.5. A Scene with Metal Spheres

Now let’s add some metal spheres to our scene:

```
...

#include "material.h"

...

int main() {

    // Image

    const auto aspect_ratio = 16.0 / 9.0;
    const int image_width = 400;
    const int image_height = static_cast<int>(image_width / aspect_ratio);
    const int samples_per_pixel = 100;
    const int max_depth = 50;

    // World

    hittable_list world;

    auto material_ground = make_shared<lambertian>(color(0.8, 0.8, 0.0));
    auto material_center = make_shared<lambertian>(color(0.7, 0.3, 0.3));
    auto material_left   = make_shared<metal>(color(0.8, 0.8, 0.8));
    auto material_right  = make_shared<metal>(color(0.8, 0.6, 0.2));

    world.add(make_shared<sphere>(point3( 0.0, -100.5, -1.0), 100.0,
material_ground));
    world.add(make_shared<sphere>(point3( 0.0,    0.0, -1.0),   0.5,
material_center));
    world.add(make_shared<sphere>(point3(-1.0,    0.0, -1.0),   0.5,
material_left));
    world.add(make_shared<sphere>(point3( 1.0,    0.0, -1.0),   0.5,
material_right));

    // Camera

    camera cam;

    // Render

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

    for (int j = image_height-1; j >= 0; --j) {
        std::cerr << "\rScanlines remaining: " << j << ' ' << std::flush;
        for (int i = 0; i < image_width; ++i) {
            color pixel_color(0, 0, 0);
            for (int s = 0; s < samples_per_pixel; ++s) {
                auto u = (i + random_double()) / (image_width-1);
                auto v = (j + random_double()) / (image_height-1);
                ray r = cam.get_ray(u, v);
                pixel_color += ray_color(r, world, max_depth);
            }
            write_color(std::cout, pixel_color, samples_per_pixel);
        }
    }
```

```
    std::cerr << "\nDone.\n";
}
```

Listing 50: [main.cc] *Scene with metal spheres*

Which gives:

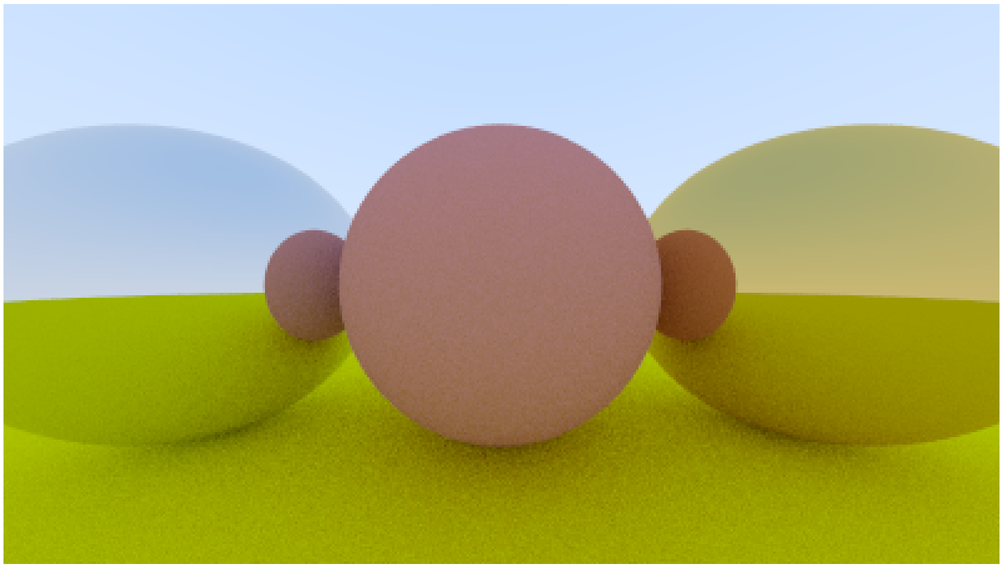


Image 11: Shiny metal

9.6. Fuzzy Reflection

We can also randomize the reflected direction by using a small sphere and choosing a new endpoint for the ray:

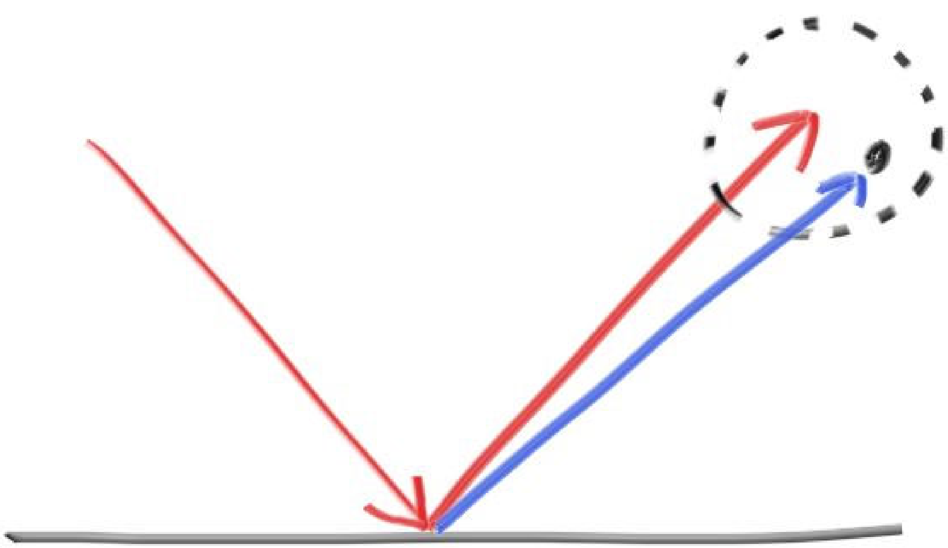


Figure 12: *Generating fuzzed reflection rays*

The bigger the sphere, the fuzzier the reflections will be. This suggests adding a fuzziness parameter that is just the radius of the sphere (so zero is no perturbation). The catch is that for big spheres or grazing rays, we may scatter below the surface. We can just have the surface absorb those.

```
class metal : public material {
    public:
        metal(const color& a, double f) : albedo(a), fuzz(f < 1 ? f : 1) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray&
scattered
        ) const override {
            vec3 reflected = reflect(unit_vector(r_in.direction()),
rec.normal);
            scattered = ray(rec.p, reflected + fuzz*random_in_unit_sphere());
            attenuation = albedo;
            return (dot(scattered.direction(), rec.normal) > 0);
        }

    public:
        color albedo;
        double fuzz;
};
```

Listing 51: [material.h] *Metal material fuzziness*

We can try that out by adding fuzziness 0.3 and 1.0 to the metals:

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

    auto material_ground = make_shared<lambertian>(color(0.8, 0.8, 0.0));
    auto material_center = make_shared<lambertian>(color(0.7, 0.3, 0.3));
    auto material_left   = make_shared<metal>(color(0.8, 0.8, 0.8), 0.3);
    auto material_right  = make_shared<metal>(color(0.8, 0.6, 0.2), 1.0);
    ...
}
```

Listing 52: [main.cc] *Metal spheres with fuzziness*

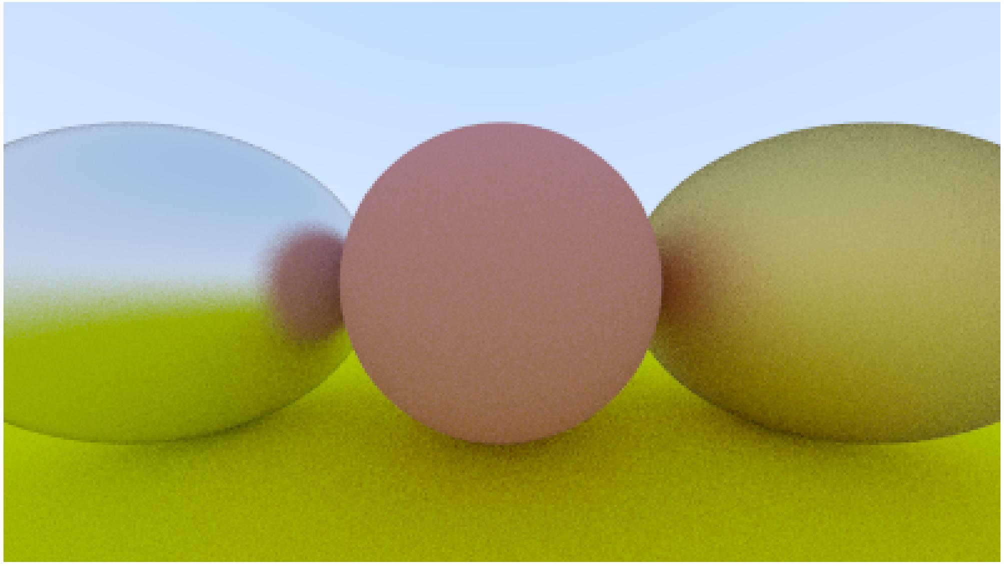


Image 12: Fuzzed metal

10. Dielectrics

Clear materials such as water, glass, and diamonds are dielectrics. When a light ray hits them, it splits into a reflected ray and a refracted (transmitted) ray. We' ll handle that by randomly choosing between reflection or refraction, and only generating one scattered ray per interaction.

10.1. Refraction

The hardest part to debug is the refracted ray. I usually first just have all the light refract if there is a refraction ray at all. For this project, I tried to put two glass balls in our scene, and I got this (I have not told you how to do this right or wrong yet, but soon!):

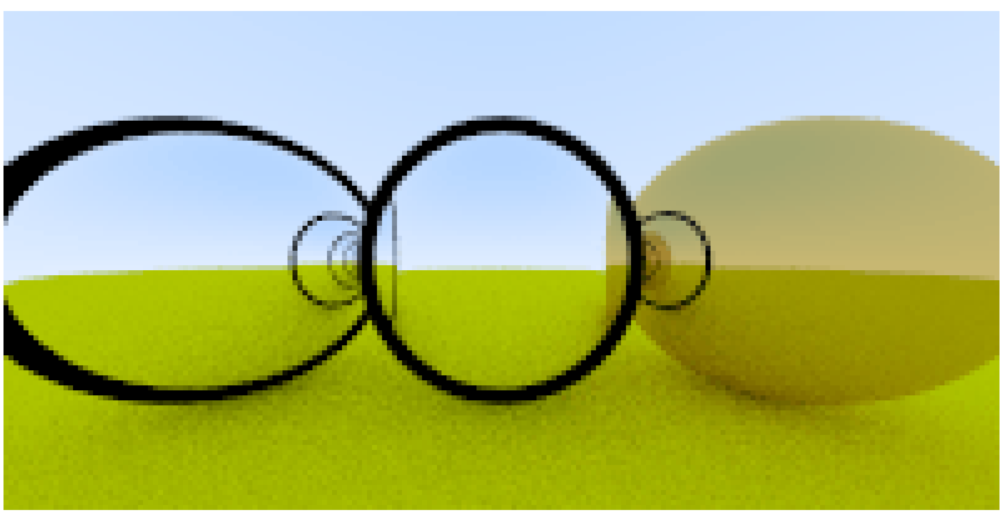


Image 13: Glass first

Is that right? Glass balls look odd in real life. But no, it isn' t right. The world should be flipped upside down and no weird black stuff. I just printed out the ray straight through the middle of the image and it was clearly wrong. That often does the job.

10.2. Snell's Law

The refraction is described by Snell' s law:

$$\boxed{\eta \cdot \sin\theta = \eta' \cdot \sin\theta'}$$

Where θ and θ' are the angles from the normal, and η and η' (pronounced "eta" and "eta prime") are the refractive indices (typically air = 1.0, glass = 1.3–1.7, diamond = 2.4). The geometry is:

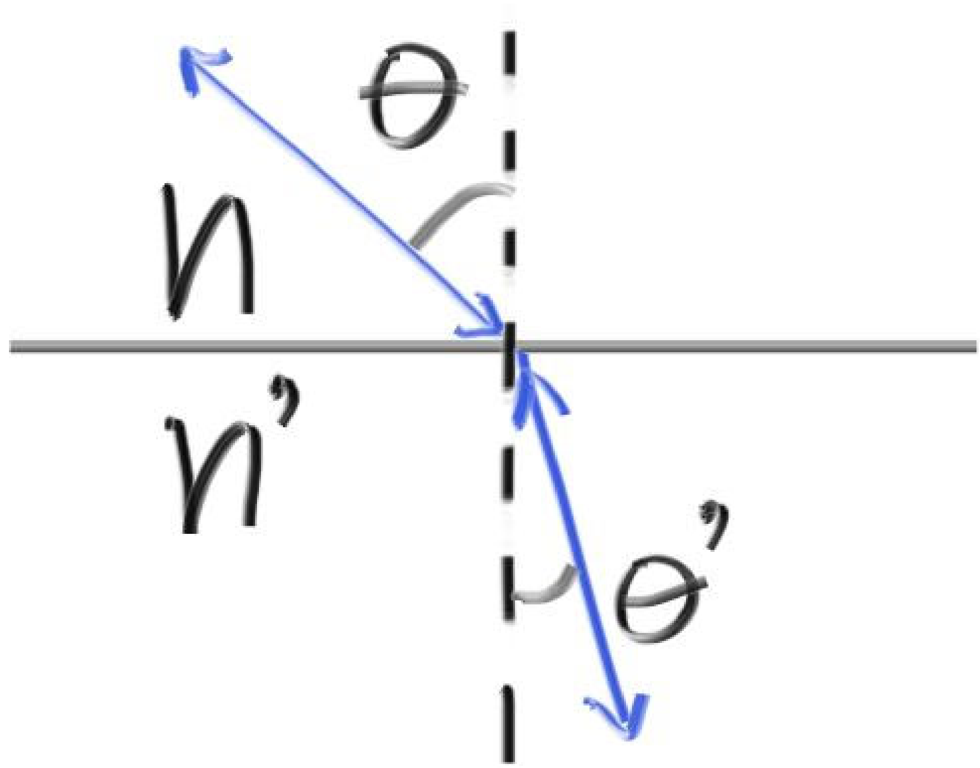


Figure 13: *Ray refraction*

In order to determine the direction of the refracted ray, we have to solve for $\sin\theta'$:

$$\boxed{\sin\theta' = \frac{\eta}{\eta'} \cdot \sin\theta}$$

On the refracted side of the surface there is a refracted ray $\mathbf{R'}$ and a normal $\mathbf{n'}$, and there exists an angle, θ', between them. We can split $\mathbf{R'}$ into the parts of the ray that are perpendicular to $\mathbf{n'}$ and parallel to $\mathbf{n'}$:

$$\boxed{\mathbf{R'} = \mathbf{R'}_{\perp} + \mathbf{R'}_{\parallel}}$$

If we solve for $\mathbf{R'}_{\perp}$ and $\mathbf{R'}_{\parallel}$ we get:

$$\boxed{\begin{aligned}\mathbf{R'}_{\perp} &= \frac{\eta}{\eta'}(\mathbf{R} + \cos\theta\mathbf{n}) \\ \mathbf{R'}_{\parallel} &= -\sqrt{1 - |\mathbf{R'}_{\perp}|^2}\mathbf{n}\end{aligned}}$$

You can go ahead and prove this for yourself if you want, but we will treat it as fact and move on. The rest of the book will not require you to understand the proof.

We still need to solve for $\cos\theta$. It is well known that the dot product of two vectors can be explained in terms of the cosine of the angle between them:

$$\boxed{\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos\theta}$$

If we restrict $\mathbf{a}$ and $\mathbf{b}$ to be unit vectors:

$$\boxed{\mathbf{a} \cdot \mathbf{b} = \cos\theta}$$

We can now rewrite $\mathbf{R'}_{\perp}$ in terms of known quantities:

$$\boxed{\mathbf{R'}_{\perp} = \frac{\eta}{\eta'}(\mathbf{R} + (-\mathbf{R} \cdot \mathbf{n})\mathbf{n})}$$

When we combine them back together, we can write a function to calculate $\mathbf{R'}$:

```
vec3 refract(const vec3& uv, const vec3& n, double etai_over_etat) {
    auto cos_theta = fmin(dot(-uv, n), 1.0);
    vec3 r_out_perp =  etai_over_etat * (uv + cos_theta*n);
    vec3 r_out_parallel = -sqrt(fabs(1.0 - r_out_perp.length_squared())) * n;
    return r_out_perp + r_out_parallel;
}
```

Listing 53: [vec3.h] *Refraction function*

And the dielectric material that always refracts is:

```
class dielectric : public material {
    public:
        dielectric(double index_of_refraction) : ir(index_of_refraction) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray&
scattered
        ) const override {
            attenuation = color(1.0, 1.0, 1.0);
            double refraction_ratio = rec.front_face ? (1.0/ir) : ir;

            vec3 unit_direction = unit_vector(r_in.direction());
            vec3 refracted = refract(unit_direction, rec.normal,
refraction_ratio);

            scattered = ray(rec.p, refracted);
            return true;
        }

    public:
        double ir; // Index of Refraction
};
```

Listing 54: [material.h] *Dielectric material class that always refracts*

Now we'll update the scene to change the left and center spheres to glass:

```
auto material_ground = make_shared<lambertian>(color(0.8, 0.8, 0.0));
auto material_center = make_shared<dielectric>(1.5);
auto material_left   = make_shared<dielectric>(1.5);
auto material_right  = make_shared<metal>(color(0.8, 0.6, 0.2), 1.0);
```

Listing 55: [main.cc] *Changing left and center spheres to glass*

This gives us the following result:

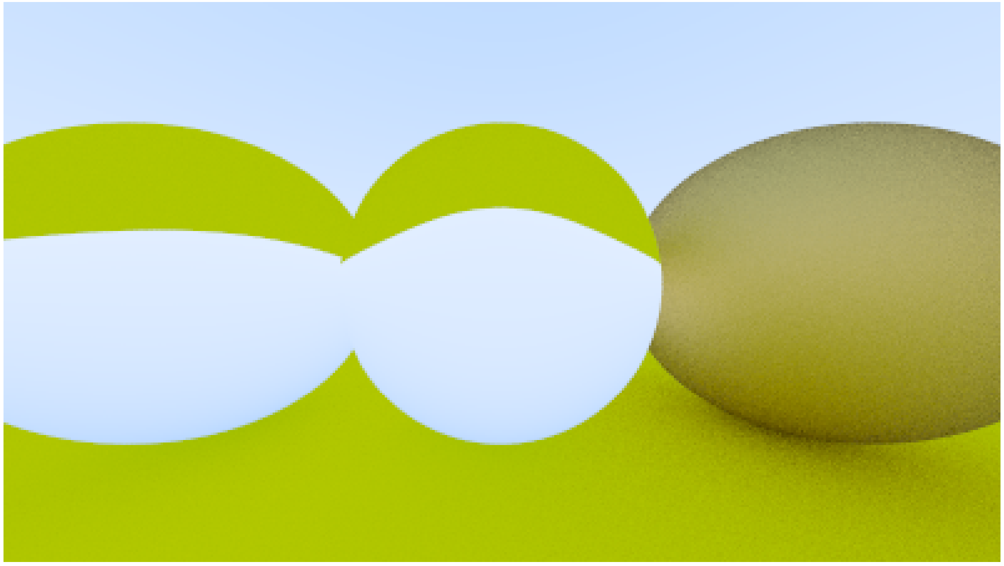


Image 14: Glass sphere that always refracts

10.3. Total Internal Reflection

That definitely doesn't look right. One troublesome practical issue is that when the ray is in the material with the higher refractive index, there is no real solution to Snell’ s law, and thus there is no refraction possible. If we refer back to Snell's law and the derivation of $\sin\theta'$:

$$\sin\theta' = \frac{\eta}{\eta'} \cdot \sin\theta$$

If the ray is inside glass and outside is air ($\eta = 1.5$ and $\eta' = 1.0$):

$$\sin\theta' = \frac{1.5}{1.0} \cdot \sin\theta$$

The value of $\sin\theta'$ cannot be greater than 1. So, if,

$$\frac{1.5}{1.0} \cdot \sin\theta > 1.0$$

,

the equality between the two sides of the equation is broken, and a solution cannot exist. If a solution does not exist, the glass cannot refract, and therefore must reflect the ray:

```
if (refraction_ratio * sin_theta > 1.0) {
    // Must Reflect
    ...
} else {
    // Can Refract
    ...
}
```

Listing 56: [material.h] *Determining if the ray can refract*

Here all the light is reflected, and because in practice that is usually inside solid objects, it is called "total internal reflection". This is why sometimes the water-air boundary acts as a perfect mirror when you are submerged.

We can solve for `sin_theta` using the trigonometric qualities:

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

and

$$\boxed{\cos\theta = \mathbf{R} \cdot \mathbf{n}}$$

```
double cos_theta = fmin(dot(-unit_direction, rec.normal), 1.0);
double sin_theta = sqrt(1.0 - cos_theta*cos_theta);

if (refraction_ratio * sin_theta > 1.0) {
    // Must Reflect
    ...
} else {
    // Can Refract
    ...
}
```

Listing 57: `[material.h]` *Determining if the ray can refract*

And the dielectric material that always refracts (when possible) is:

```
class dielectric : public material {
    public:
        dielectric(double index_of_refraction) : ir(index_of_refraction) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray&
scattered
        ) const override {
            attenuation = color(1.0, 1.0, 1.0);
            double refraction_ratio = rec.front_face ? (1.0/ir) : ir;

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

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

            if (cannot_refract)
                direction = reflect(unit_direction, rec.normal);
            else
                direction = refract(unit_direction, rec.normal,
refraction_ratio);

            scattered = ray(rec.p, direction);
            return true;
        }

    public:
        double ir; // Index of Refraction
};
```

Listing 58: [material.h] *Dielectric material class with reflection*

Attenuation is always 1 — the glass surface absorbs nothing. If we try that out with these parameters:

```
auto material_ground = make_shared<lambertian>(color(0.8, 0.8, 0.0));
auto material_center = make_shared<lambertian>(color(0.1, 0.2, 0.5));
auto material_left   = make_shared<dielectric>(1.5);
auto material_right  = make_shared<metal>(color(0.8, 0.6, 0.2), 0.0);
```

Listing 59: `[main.cc]` *Scene with dielectric and shiny sphere*

We get:

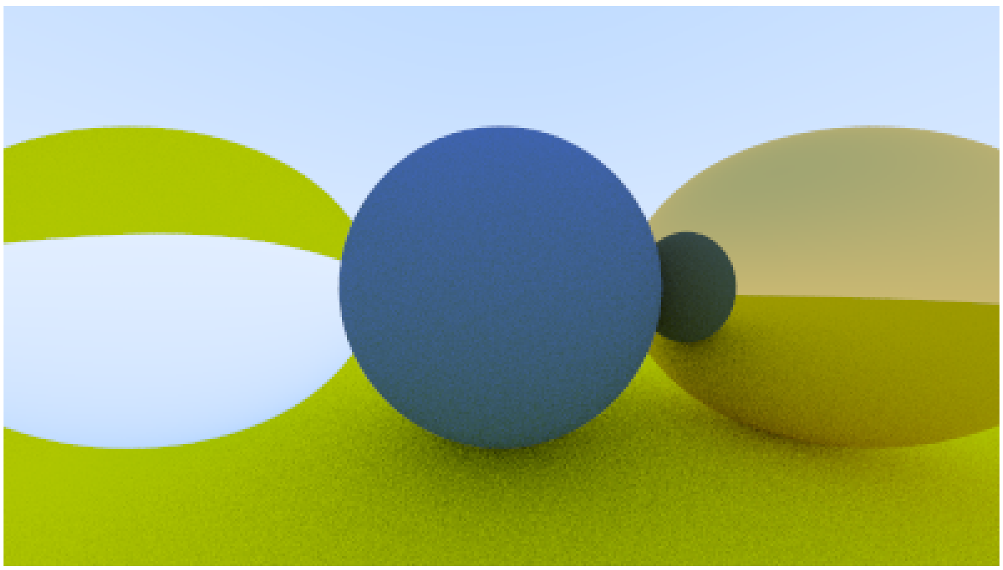


Image 15: Glass sphere that sometimes refracts

10.4. Schlick Approximation

Now real glass has reflectivity that varies with angle — look at a window at a steep angle and it becomes a mirror. There is a big ugly equation for that, but almost everybody uses a cheap and surprisingly accurate polynomial approximation by Christophe Schlick. This yields our full glass material:

```
class dielectric : public material {
    public:
        dielectric(double index_of_refraction) : ir(index_of_refraction) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray&
scattered
        ) const override {
            attenuation = color(1.0, 1.0, 1.0);
            double refraction_ratio = rec.front_face ? (1.0/ir) : ir;

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

            bool cannot_refract = refraction_ratio * sin_theta > 1.0;
            vec3 direction;
            if (cannot_refract || reflectance(cos_theta, refraction_ratio) >
random_double())
                direction = reflect(unit_direction, rec.normal);
            else
                direction = refract(unit_direction, rec.normal,
refraction_ratio);

            scattered = ray(rec.p, direction);
            return true;
        }

    public:
        double ir; // Index of Refraction

    private:
        static double reflectance(double cosine, double ref_idx) {
            // Use Schlick's approximation for reflectance.
            auto r0 = (1-ref_idx) / (1+ref_idx);
            r0 = r0*r0;
            return r0 + (1-r0)*pow((1 - cosine),5);
        }
};
```

Listing 60: [material.h] *Full glass material*

10.5. Modeling a Hollow Glass Sphere

An interesting and easy trick with dielectric spheres is to note that if you use a negative radius, the geometry is unaffected, but the surface normal points inward. This can be used as a bubble to make a hollow glass sphere:

```
world.add(make_shared<sphere>(point3( 0.0, -100.5, -1.0), 100.0,
material_ground));
world.add(make_shared<sphere>(point3( 0.0,    0.0, -1.0),   0.5,
material_center));
world.add(make_shared<sphere>(point3(-1.0,    0.0, -1.0),   0.5,
material_left));
world.add(make_shared<sphere>(point3(-1.0,    0.0, -1.0),  -0.4,
material_left));
world.add(make_shared<sphere>(point3( 1.0,    0.0, -1.0),   0.5,
material_right));
```

Listing 61: [main.cc] *Scene with hollow glass sphere*

This gives:

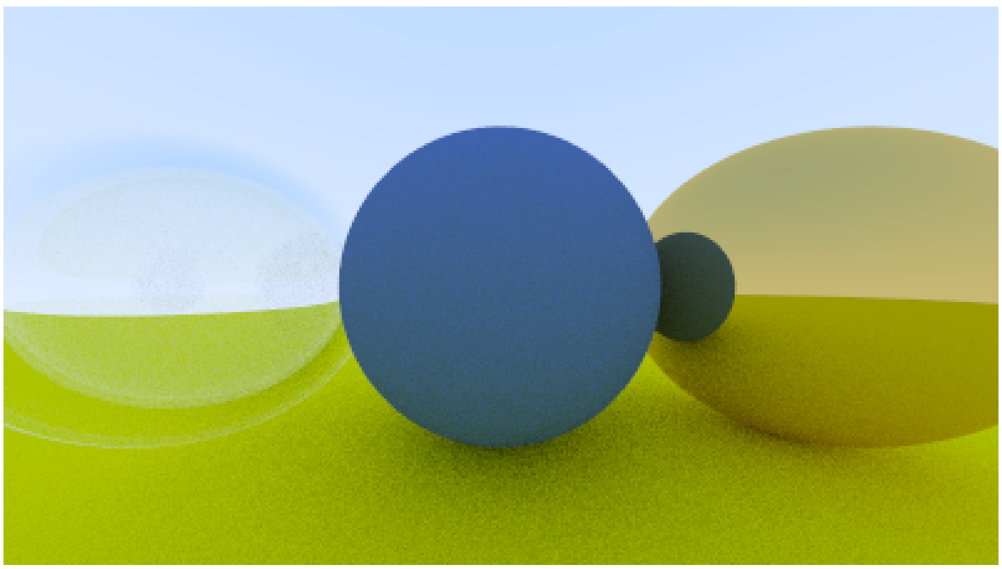


Image 16: A hollow glass sphere

11. Positionable Camera

Cameras, like dielectrics, are a pain to debug. So I always develop mine incrementally. First, let's allow an adjustable field of view (*fov*). This is the angle you see through the portal. Since our image is not square, the fov is different horizontally and vertically. I always use vertical fov. I also usually specify it in degrees and change to radians inside a constructor — a matter of personal taste.

11.1. Camera Viewing Geometry

I first keep the rays coming from the origin and heading to the $z = -1$ plane. We could make it the $z = -2$ plane, or whatever, as long as we made h a ratio to that distance. Here is our setup:

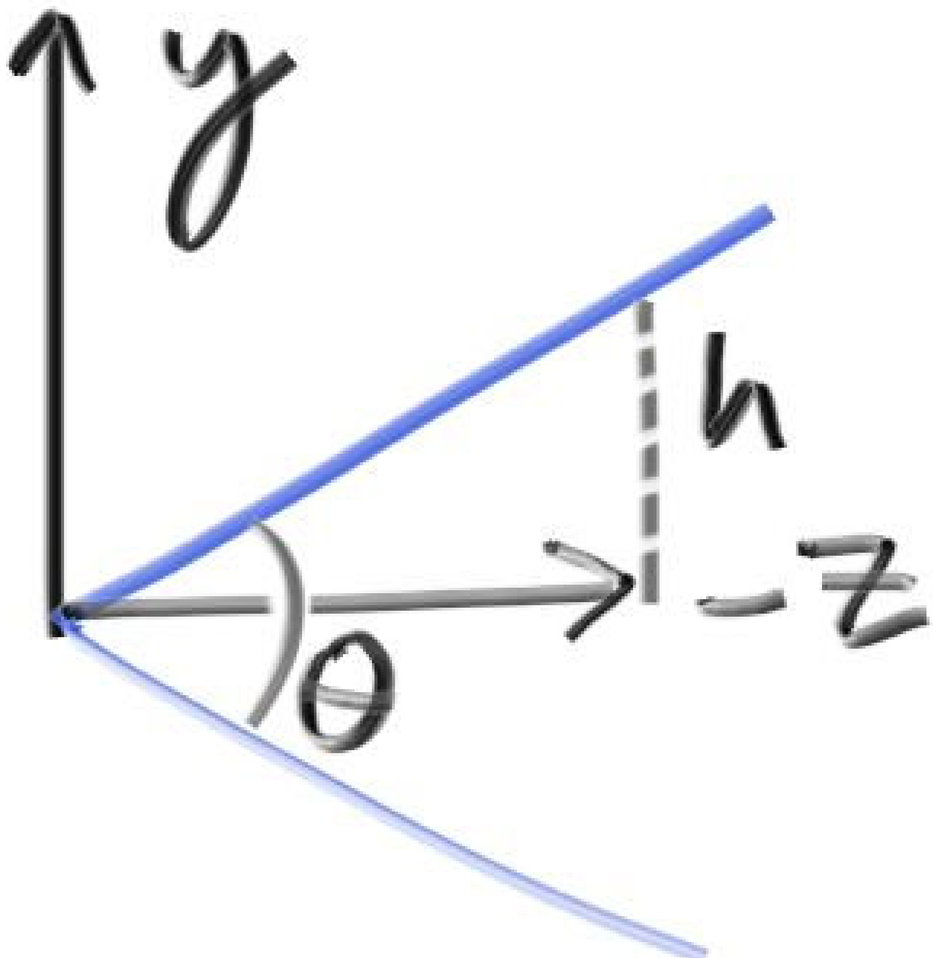


Figure 14: *Camera viewing geometry*

This implies $h = \tan(\frac{\theta}{2})$. Our camera now becomes:

```
class camera {
    public:
        camera(
            double vfov, // vertical field-of-view in degrees
            double aspect_ratio
        ) {
            auto theta = degrees_to_radians(vfov);
            auto h = tan(theta/2);
            auto viewport_height = 2.0 * h;
            auto viewport_width = aspect_ratio * viewport_height;

            auto focal_length = 1.0;

            origin = point3(0, 0, 0);
            horizontal = vec3(viewport_width, 0.0, 0.0);
            vertical = vec3(0.0, viewport_height, 0.0);
            lower_left_corner = origin - horizontal/2 - vertical/2 - vec3(0, 0,
focal_length);
        }

        ray get_ray(double u, double v) const {
            return ray(origin, lower_left_corner + u*horizontal + v*vertical -
origin);
        }

    private:
        point3 origin;
        point3 lower_left_corner;
        vec3 horizontal;
        vec3 vertical;
};
```

Listing 62: [camera.h] *Camera with adjustable field-of-view (fov)*

When calling it with camera `cam(90, aspect_ratio)` and these spheres:

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

    auto R = cos(pi/4);
    hittable_list world;

    auto material_left  = make_shared<lambertian>(color(0,0,1));
    auto material_right = make_shared<lambertian>(color(1,0,0));

    world.add(make_shared<sphere>(point3(-R, 0, -1), R, material_left));
    world.add(make_shared<sphere>(point3( R, 0, -1), R, material_right));

    // Camera

    camera cam(90.0, aspect_ratio);

    // Render

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

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

Listing 63: [main.cc] *Scene with wide-angle camera*

gives:

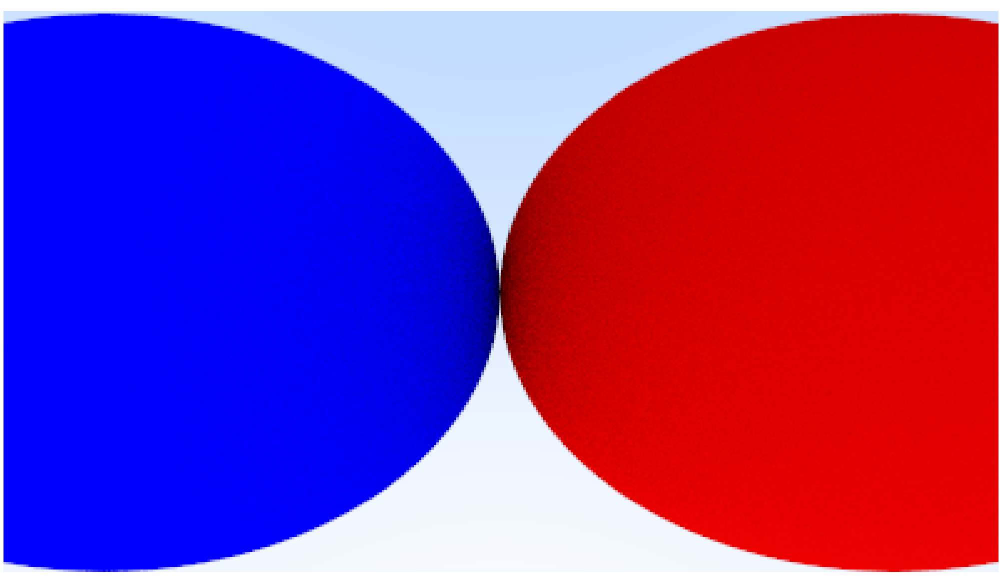


Image 17: A wide-angle view

11.2. Positioning and Orienting the Camera

To get an arbitrary viewpoint, let' s first name the points we care about. We' ll call the position where we place the camera *lookfrom*, and the point we look at *lookat*. (Later, if you want, you could define a direction to look in instead of a point to look at.)

We also need a way to specify the roll, or sideways tilt, of the camera: the rotation around the lookat-lookfrom axis. Another way to think about it is that even if you keep `lookfrom` and `lookat` constant, you can still rotate your head around your nose. What we need is a way to specify an "up" vector for the camera. This up vector should lie in the plane orthogonal to the view direction.

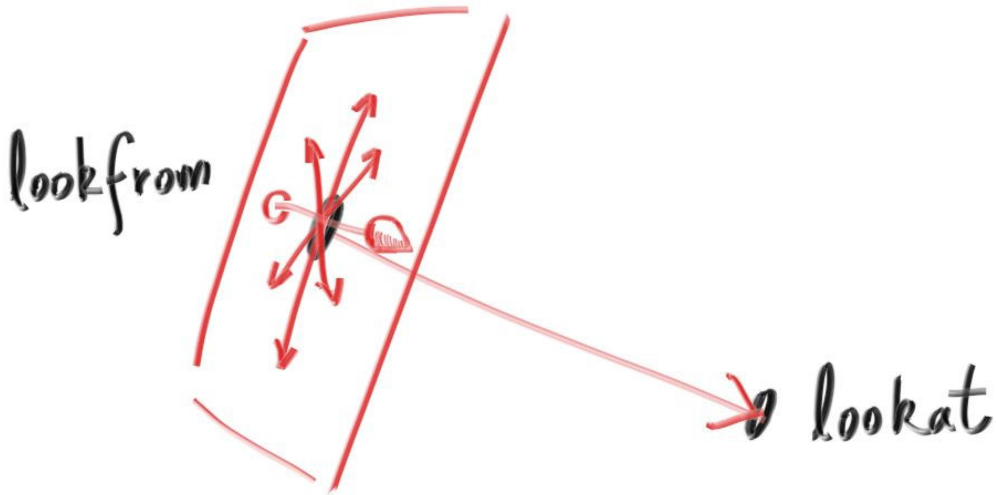


Figure 15: *Camera view direction*

We can actually use any up vector we want, and simply project it onto this plane to get an up vector for the camera. I use the common convention of naming a "view up" (*vup*) vector. A couple of cross products, and we now have a complete orthonormal basis (u, v, w) to describe our camera's orientation.

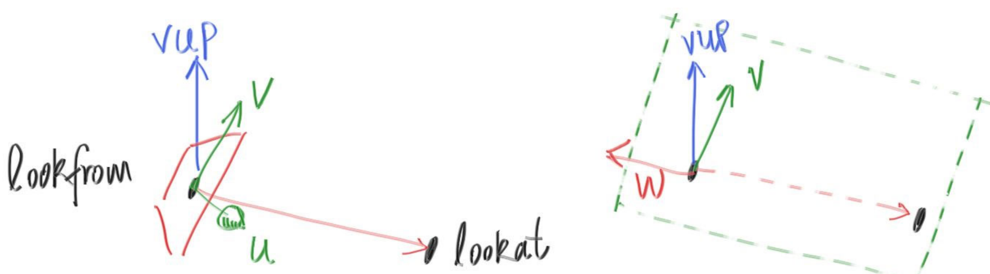


Figure 16: *Camera view up direction*

Remember that `vup`, `v`, and `w` are all in the same plane. Note that, like before when our fixed camera faced -Z, our arbitrary view camera faces -w. And keep in mind that we can — but we don't have to — use world up $(0, 1, 0)$ to specify vup. This is convenient and will naturally keep your camera horizontally level until you decide to experiment with crazy camera angles.

```
class camera {
    public:
        camera(
            point3 lookfrom,
            point3 lookat,
            vec3   vup,
            double vfov, // vertical field-of-view in degrees
            double aspect_ratio
        ) {
            auto theta = degrees_to_radians(vfov);
            auto h = tan(theta/2);
            auto viewport_height = 2.0 * h;
            auto viewport_width = aspect_ratio * viewport_height;

            auto w = unit_vector(lookfrom - lookat);
            auto u = unit_vector(cross(vup, w));
            auto v = cross(w, u);

            origin = lookfrom;
            horizontal = viewport_width * u;
            vertical = viewport_height * v;
            lower_left_corner = origin - horizontal/2 - vertical/2 - w;
        }

        ray get_ray(double s, double t) const {
            return ray(origin, lower_left_corner + s*horizontal + t*vertical -
origin);
        }

    private:
        point3 origin;
        point3 lower_left_corner;
        vec3 horizontal;
        vec3 vertical;
};
```

Listing 64: [camera.h] *Positionable and orientable camera*

We'll change back to the prior scene, and use the new viewpoint:

```
hittable_list world;

auto material_ground = make_shared<lambertian>(color(0.8, 0.8, 0.0));
auto material_center = make_shared<lambertian>(color(0.1, 0.2, 0.5));
auto material_left   = make_shared<dielectric>(1.5);
auto material_right  = make_shared<metal>(color(0.8, 0.6, 0.2), 0.0);

world.add(make_shared<sphere>(point3( 0.0, -100.5, -1.0), 100.0,
material_ground));
world.add(make_shared<sphere>(point3( 0.0,    0.0, -1.0),   0.5,
material_center));
world.add(make_shared<sphere>(point3(-1.0,    0.0, -1.0),   0.5,
material_left));
world.add(make_shared<sphere>(point3(-1.0,    0.0, -1.0), -0.45,
material_left));
world.add(make_shared<sphere>(point3( 1.0,    0.0, -1.0),   0.5,
material_right));

camera cam(point3(-2,2,1), point3(0,0,-1), vec3(0,1,0), 90, aspect_ratio);
```

Listing 65: [main.cc] *Scene with alternate viewpoint*

to get:

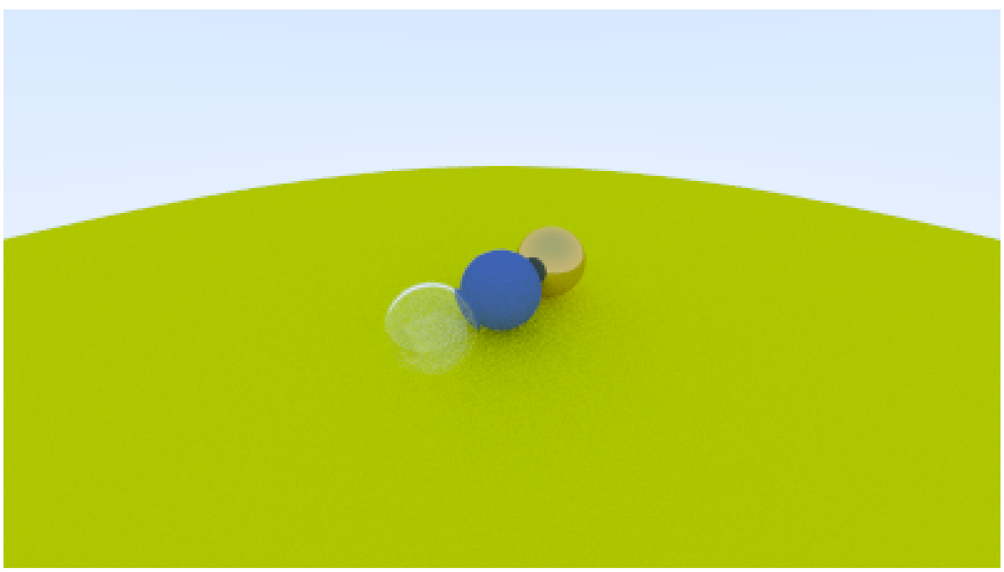


Image 18: A distant view

And we can change field of view:

```
camera cam(point3(-2,2,1), point3(0,0,-1), vec3(0,1,0), 20, aspect_ratio);
```

Listing 66: [main.cc] *Change field of view*

to get:

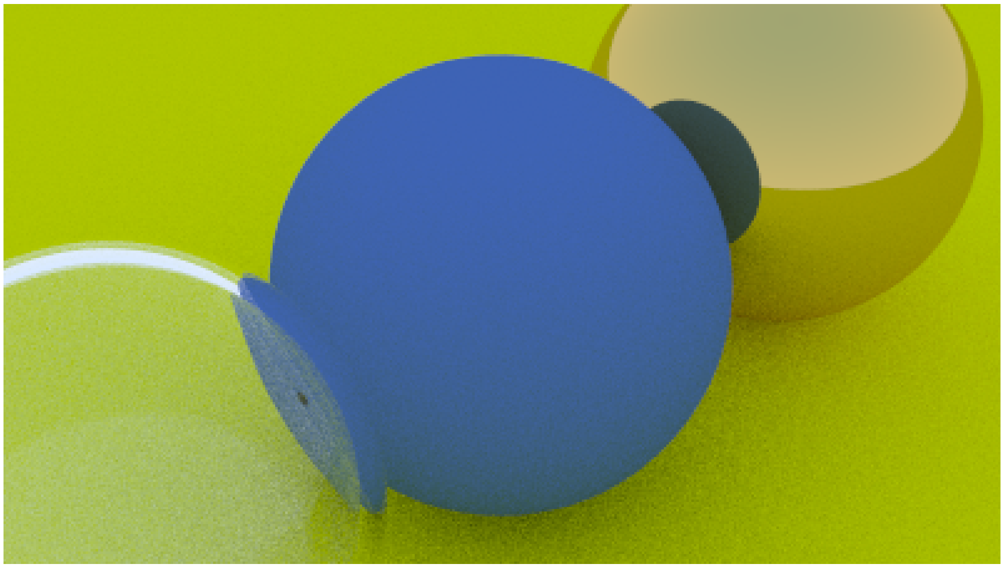


Image 19: Zooming in

12. Defocus Blur

Now our final feature: defocus blur. Note, all photographers will call it "depth of field" so be aware of only using "defocus blur" among friends.

The reason we defocus blur in real cameras is because they need a big hole (rather than just a pinhole) to gather light. This would defocus everything, but if we stick a lens in the hole, there will be a certain distance where everything is in focus. You can think of a lens this way: all light rays coming *from* a specific point at the focus distance — and that hit the lens — will be bent back *to* a single point on the image sensor.

We call the distance between the projection point and the plane where everything is in perfect focus the *focus distance*. Be aware that the focus distance is not the same as the focal length — the *focal length* is the distance between the projection point and the image plane.

In a physical camera, the focus distance is controlled by the distance between the lens and the film/sensor. That is why you see the lens move relative to the camera when you change what is in focus (that may happen in your phone camera too, but the sensor moves). The "aperture" is a hole to control how big the lens is effectively. For a real camera, if you need more light you make the aperture bigger, and will get more defocus blur. For our virtual camera, we can have a perfect sensor and never need more light, so we only have an aperture when we want defocus blur.

12.1. A Thin Lens Approximation

A real camera has a complicated compound lens. For our code we could simulate the order: sensor, then lens, then aperture. Then we could figure out where to send the rays, and flip the image after it's computed (the image is projected upside down on the film). Graphics people, however, usually use a thin lens approximation:

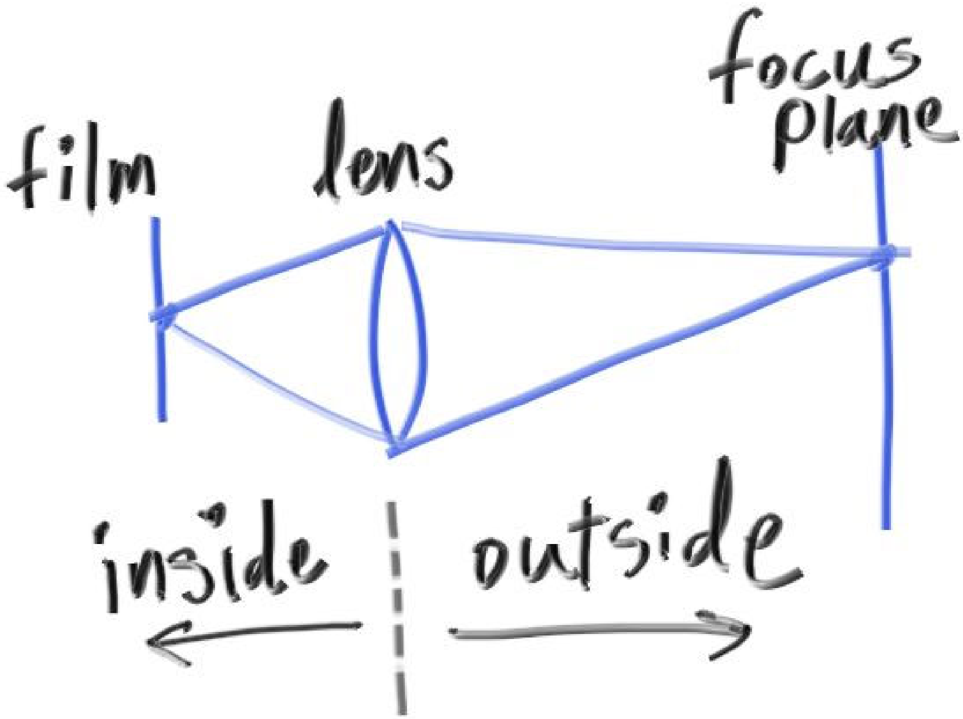


Figure 17: *Camera lens model*

We don' t need to simulate any of the inside of the camera. For the purposes of rendering an image outside the camera, that would be unnecessary complexity. Instead, I usually start rays from the lens, and send them toward the focus plane (`focus_dist` away from the lens), where everything on that plane is in perfect focus.

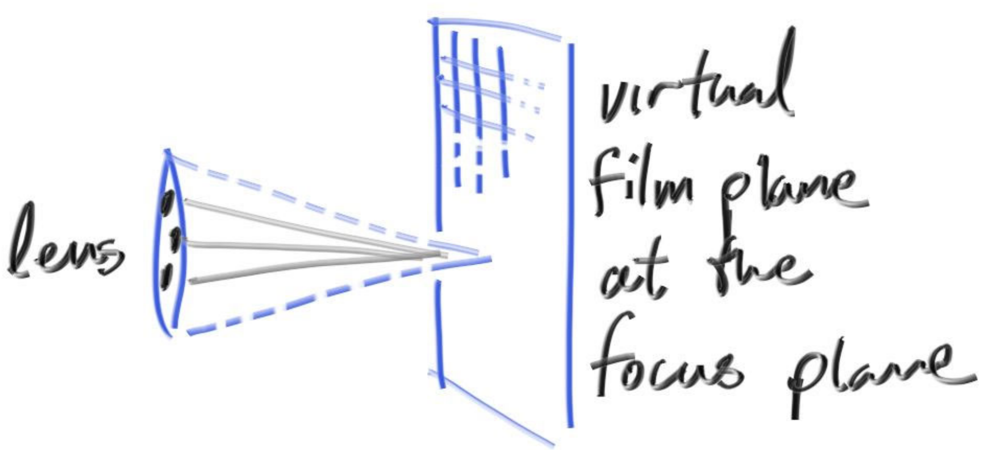


Figure 18: *Camera focus plane*

12.2. Generating Sample Rays

Normally, all scene rays originate from the `lookfrom` point. In order to accomplish defocus blur, generate random scene rays originating from inside a disk centered at the `lookfrom` point. The larger the radius, the greater the defocus blur. You can think of our original camera as having a defocus disk of radius zero (no blur at all), so all rays originated at the disk center (`lookfrom`).

```
vec3 random_in_unit_disk() {
    while (true) {
        auto p = vec3(random_double(-1,1), random_double(-1,1), 0);
        if (p.length_squared() >= 1) continue;
        return p;
    }
}
```

Listing 67: [vec3.h] *Generate random point inside unit disk*

```
class camera {
    public:
        camera(
            point3 lookfrom,
            point3 lookat,
            vec3   vup,
            double vfov, // vertical field-of-view in degrees
            double aspect_ratio,
            double aperture,
            double focus_dist
        ) {
            auto theta = degrees_to_radians(vfov);
            auto h = tan(theta/2);
            auto viewport_height = 2.0 * h;
            auto viewport_width = aspect_ratio * viewport_height;

            w = unit_vector(lookfrom - lookat);
            u = unit_vector(cross(vup, w));
            v = cross(w, u);

            origin = lookfrom;
            horizontal = focus_dist * viewport_width * u;
            vertical = focus_dist * viewport_height * v;
            lower_left_corner = origin - horizontal/2 - vertical/2 -
focus_dist*w;

            lens_radius = aperture / 2;
        }


        ray get_ray(double s, double t) const {
            vec3 rd = lens_radius * random_in_unit_disk();
            vec3 offset = u * rd.x() + v * rd.y();

            return ray(
                origin + offset,
                lower_left_corner + s*horizontal + t*vertical - origin - offset
            );
        }

    private:
        point3 origin;
        point3 lower_left_corner;
        vec3 horizontal;
        vec3 vertical;
        vec3 u, v, w;
        double lens_radius;
};
```

Listing 68: [camera.h] *Camera with adjustable depth-of-field (dof)*

Using a big aperture:

```
point3 lookfrom(3,3,2);
point3 lookat(0,0,-1);
vec3 vup(0,1,0);
auto dist_to_focus = (lookfrom-lookat).length();
auto aperture = 2.0;

camera cam(lookfrom, lookat, vup, 20, aspect_ratio, aperture, dist_to_focus);
```

Listing 69: [main.cc] *Scene camera with depth-of-field*

We get:

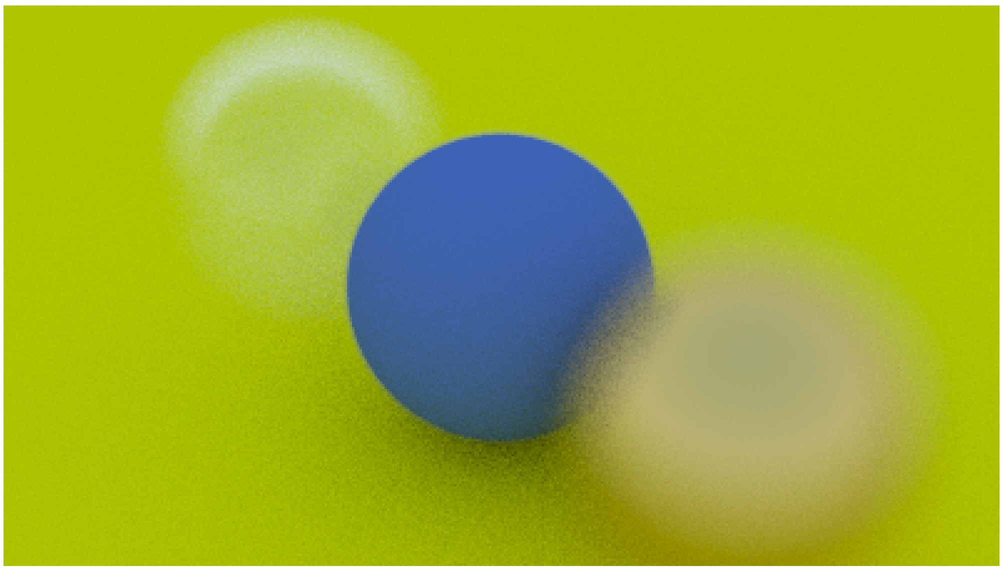


Image 20: Spheres with depth-of-field

13. Where Next?

13.1. A Final Render

First let’s make the image on the cover of this book — lots of random spheres:

```
hittable_list random_scene() {
    hittable_list world;

    auto ground_material = make_shared<lambertian>(color(0.5, 0.5, 0.5));
    world.add(make_shared<sphere>(point3(0,-1000,0), 1000, ground_material));

    for (int a = -11; a < 11; a++) {
        for (int b = -11; b < 11; b++) {
            auto choose_mat = random_double();
            point3 center(a + 0.9*random_double(), 0.2, b +
0.9*random_double());

            if ((center - point3(4, 0.2, 0)).length() > 0.9) {
                shared_ptr<material> sphere_material;

                if (choose_mat < 0.8) {
                    // diffuse
                    auto albedo = color::random() * color::random();
                    sphere_material = make_shared<lambertian>(albedo);
                    world.add(make_shared<sphere>(center, 0.2,
sphere_material));
                } else if (choose_mat < 0.95) {
                    // metal
                    auto albedo = color::random(0.5, 1);
                    auto fuzz = random_double(0, 0.5);
                    sphere_material = make_shared<metal>(albedo, fuzz);
                    world.add(make_shared<sphere>(center, 0.2,
sphere_material));
                } else {
                    // glass
                    sphere_material = make_shared<dielectric>(1.5);
                    world.add(make_shared<sphere>(center, 0.2,
sphere_material));
                }
            }
        }
    }

    auto material1 = make_shared<dielectric>(1.5);
    world.add(make_shared<sphere>(point3(0, 1, 0), 1.0, material1));

    auto material2 = make_shared<lambertian>(color(0.4, 0.2, 0.1));
    world.add(make_shared<sphere>(point3(-4, 1, 0), 1.0, material2));

    auto material3 = make_shared<metal>(color(0.7, 0.6, 0.5), 0.0);
    world.add(make_shared<sphere>(point3(4, 1, 0), 1.0, material3));

    return world;
}

int main() {

    // Image

    const auto aspect_ratio = 3.0 / 2.0;
    const int image_width = 1200;
```

```
    const int image_height = static_cast<int>(image_width / aspect_ratio);
    const int samples_per_pixel = 500;
    const int max_depth = 50;

    // World

    auto world = random_scene();

    // Camera

    point3 lookfrom(13,2,3);
    point3 lookat(0,0,0);
    vec3 vup(0,1,0);
    auto dist_to_focus = 10.0;
    auto aperture = 0.1;

    camera cam(lookfrom, lookat, vup, 20, aspect_ratio, aperture,
dist_to_focus);

    // Render

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

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

Listing 70: [main.cc] *Final scene*

This gives:

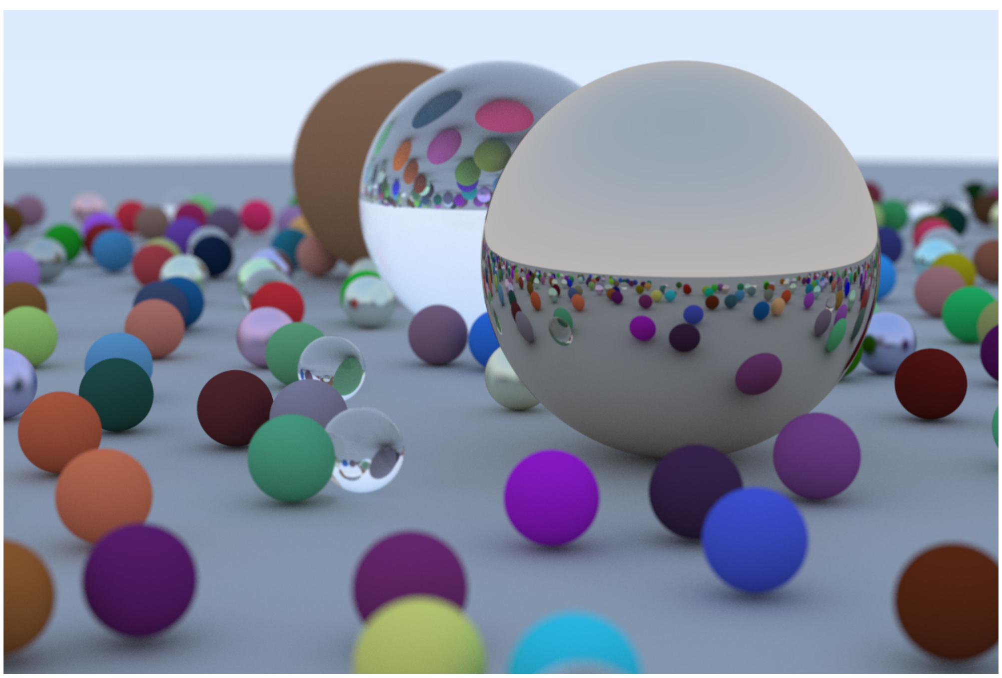


Image 21: Final scene

An interesting thing you might note is the glass balls don't really have shadows which makes them look like they are floating. This is not a bug — you don't see glass balls much in real life, where they also look a bit strange, and indeed seem to float on cloudy days. A point on the big sphere under a glass ball still has lots of light hitting it because the sky is re-ordered rather than blocked.

13.2. Next Steps

You now have a cool ray tracer! What next?

1. Lights — You can do this explicitly, by sending shadow rays to lights, or it can be done implicitly by making some objects emit light, biasing scattered rays toward them, and then downweighting those rays to cancel out the bias. Both work. I am in the minority in favoring the latter approach.

2. Triangles — Most cool models are in triangle form. The model I/O is the worst and almost everybody tries to get somebody else's code to do this.

3. Surface Textures — This lets you paste images on like wall paper. Pretty easy and a good thing to do.

4. Solid textures — Ken Perlin has his code online. Andrew Kensler has some very cool info at his blog.

5. Volumes and Media — Cool stuff and will challenge your software architecture. I favor making volumes have the hittable interface and probabilistically have intersections based on density. Your rendering code doesn't even have to know it has volumes with that method.

6. Parallelism — Run N copies of your code on N cores with different random seeds. Average the N runs. This averaging can also be done hierarchically where $N/2$ pairs can be averaged to get $N/4$ images, and pairs of those can be averaged. That method of parallelism should extend well into the thousands of cores with very little coding.

Have fun, and please send me your cool images!

14. Acknowledgments

Original Manuscript Help

Dave Hart
Jean Buckley

Web Release

Berna Kabadayı
Lorenzo Mancini
Lori Whippler Hollasch
Ronald Wotzlaw

Corrections and Improvements

Aaryaman Vasishta
Andrew Kensler
Antonio Gamiz
Apoorva Joshi
Aras Pranckevičius
Becker
Ben Kerl
Benjamin Summerton
Bennett Hardwick
Dan Drummond
David Chambers
David Hart
Eric Haines
Fabio Sancinetti
Filipe Scur
Frank He
Gerrit Wessendorf
Grue Debry
Ingo Wald
Jason Stone
Jean Buckley

Joey Cho
John Kilpatrick
Kaan Eraslan
Lorenzo Mancini
Manas Kale
Marcus Ottosson
Mark Craig
Matthew Heimlich
Nakata Daisuke
Paul Melis
Phil Cristensen
Ronald Wotzlaw
Shaun P. Lee
Shota Kawajiri
Tatsuya Ogawa
Thiago Ize
Vahan Sosoyan
ZeHao Chen

Special Thanks

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

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

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

15. Citing This Book

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

15.1. Basic Data

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

15.2. Snippets

15.2.1 Markdown

```
[_Ray Tracing in One Weekend_]
(https://raytracing.github.io/books/RayTracingInOneWeekend.html)
```

15.2.2 HTML

```
<a href="https://raytracing.github.io/books/RayTracingInOneWeekend.html">
    <cite>Ray Tracing in One Weekend</cite>
</a>
```

15.2.3 LaTeX and BibTex

```
~\cite{Shirley2020RTW1}

@misc{Shirley2020RTW1,
   title = {Ray Tracing in One Weekend},
   author = {Peter Shirley},
   year = {2020},
   month = {December},
   note = {\small
\texttt{https://raytracing.github.io/books/RayTracingInOneWeekend.html}},
   url = {https://raytracing.github.io/books/RayTracingInOneWeekend.html}
}
```

15.2.4 BibLaTeX

```
\usepackage{biblatex}

~\cite{Shirley2020RTW1}

@online{Shirley2020RTW1,
   title = {Ray Tracing in One Weekend},
   author = {Peter Shirley},
   year = {2020},
   month = {December}
   url = {https://raytracing.github.io/books/RayTracingInOneWeekend.html}
}
```

15.2.5 IEEE

```
“Ray Tracing in One Weekend.”
raytracing.github.io/books/RayTracingInOneWeekend.html
(accessed MMM. DD, YYYY)
```

15.2.6 MLA:

```
Ray Tracing in One Weekend.
raytracing.github.io/books/RayTracingInOneWeekend.html
Accessed DD MMM. YYYY.
```