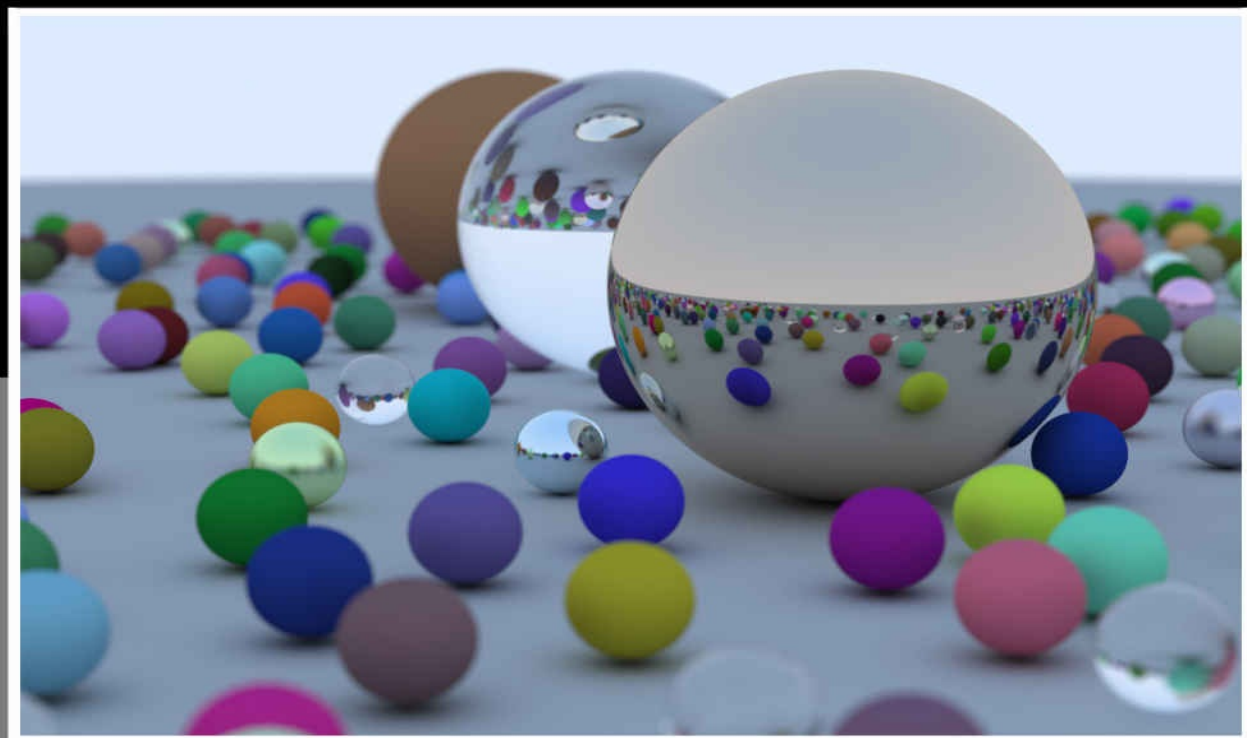


# RAY TRACING IN ONE WEEKEND



PETER SHIRLEY

# Ray Tracing in One Weekend

Peter Shirley

Copyright 2016. Peter Shirley. All rights reserved.

## Chapter 0: 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 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](mailto:ptrshrl@gmail.com)

I'll be maintaining a site related to the book including further reading and links to resources at a blog [in1weekend](#) related to this book.

Let's get on with it!


## Chapter 1: Output an image

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 and 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)

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

```

#include <iostream>

int main() {
    int nx = 200;
    int ny = 100;
    std::cout << "P3\n" << nx << " " << ny << "\n255\n";
    for (int j = ny-1; j >= 0; j--) {
        for (int i = 0; i < nx; i++) {
            float r = float(i) / float(nx);
            float g = float(j) / float(ny);
            float b = 0.2;
            int ir = int(255.99*r);
            int ig = int(255.99*g);
            int ib = int(255.99*b);
            std::cout << ir << " " << ig << " " << ib << "\n";
        }
    }
}

```

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 black to fully on 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.

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:



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
200 100
255
0 253 51
1 253 51
2 253 51
3 253 51
5 253 51
6 253 51
7 253 51
8 253 51
10 253 51
11 253 51
12 253 51
14 253 51
15 253 51
```

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](#) available [on github](#).

## Chapter 2: 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.

Here’s the top part of my vec3 class:

```

#ifndef VEC3H
#define VEC3H

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

class vec3 {

public:
    vec3() {}
    vec3(float e0, float e1, float e2) { e[0] = e0; e[1] = e1; e[2] = e2; }
    inline float x() const { return e[0]; }
    inline float y() const { return e[1]; }
    inline float z() const { return e[2]; }
    inline float r() const { return e[0]; }
    inline float g() const { return e[1]; }
    inline float b() const { return e[2]; }

    inline const vec3& operator+() const { return *this; }
    inline vec3 operator-() const { return vec3(-e[0], -e[1], -e[2]); }
    inline float operator[](int i) const { return e[i]; }
    inline float& operator[](int i) { return e[i]; }

    inline vec3& operator+=(const vec3 &v2);
    inline vec3& operator-=(const vec3 &v2);
    inline vec3& operator*=(const vec3 &v2);
    inline vec3& operator/=(const vec3 &v2);
    inline vec3& operator*=(const float t);
    inline vec3& operator/=(const float t);

    inline float length() const { return sqrt(e[0]*e[0] + e[1]*e[1] + e[2]*e[2]); }
    inline float squared_length() const { return e[0]*e[0] + e[1]*e[1] + e[2]*e[2]; }
    inline void make_unit_vector();

    float e[3];
};

```

I use floats here, but in some ray tracers I have used doubles. Neither is correct— follow your own tastes. Everything is in the header file, and later on in the file are lots of vector operations such as:

```

inline vec3 operator+(const vec3 &v1, const vec3 &v2) {
    return vec3(v1.e[0] + v2.e[0], v1.e[1] + v2.e[1], v1.e[2] + v2.e[2]);
}

inline vec3 operator-(const vec3 &v1, const vec3 &v2) {
    return vec3(v1.e[0] - v2.e[0], v1.e[1] - v2.e[1], v1.e[2] - v2.e[2]);
}

inline vec3 operator*(const vec3 &v1, const vec3 &v2) {
    return vec3(v1.e[0] * v2.e[0], v1.e[1] * v2.e[1], v1.e[2] * v2.e[2]);
}

inline vec3 operator/(const vec3 &v1, const vec3 &v2) {
    return vec3(v1.e[0] / v2.e[0], v1.e[1] / v2.e[1], v1.e[2] / v2.e[2]);
}

```

The / and \* operations are for colors and it's unlikely you want to use them for things like



locations. Similarly, we'll need some geometric operations occasionally:

```
inline float dot(const vec3 &v1, const vec3 &v2) {
    return v1.e[0] * v2.e[0] + v1.e[1] * v2.e[1] + v1.e[2] * v2.e[2];
}

inline vec3 cross(const vec3 &v1, const vec3 &v2) {
    return vec3( (v1.e[1]*v2.e[2] - v1.e[2]*v2.e[1]),
                -(v1.e[0]*v2.e[2] - v1.e[2]*v2.e[0]),
                (v1.e[0]*v2.e[1] - v1.e[1]*v2.e[0]));
}
```

And to make a unit vector in the same direction as the input vector:

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

Now we can change our main to use this:

```
#include <iostream>
#include "vec3.h"

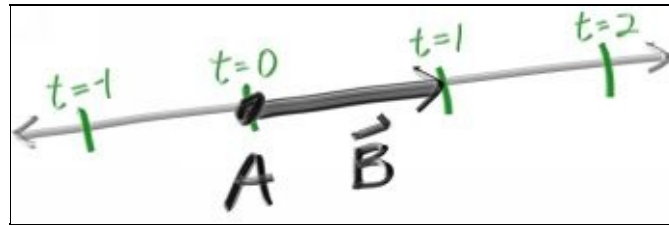
int main() {
    int nx = 200;
    int ny = 100;
    std::cout << "P3\n" << nx << " " << ny << "\n255\n";
    for (int j = ny-1; j >= 0; j--) {
        for (int i = 0; i < nx; i++) {
            vec3 col(float(i) / float(nx), float(j) / float(ny), 0.2);
            int ir = int(255.99*col[0]);
            int ig = int(255.99*col[1]);
            int ib = int(255.99*col[2]);

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

## Chapter 3: Rays, a simple camera, and background

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 (float in the code). Plug in a different  $t$  and  $\mathbf{p}(t)$  moves the point along the ray. Add in negative  $t$  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. The example  $\mathbf{C} = \mathbf{p}(2)$

is shown here:



The function  $p(t)$  in more verbose code form I call “point\_at\_parameter(t)”:

```
#ifndef RAYH
#define RAYH
#include "vec3.h"

class ray
{
public:
    ray() {}
    ray(const vec3& a, const vec3& b) { A = a; B = b; }
    vec3 origin() const { return A; }
    vec3 direction() const { return B; }
    vec3 point_at_parameter(float t) const { return A + t*B; }

    vec3 A;
    vec3 B;
};

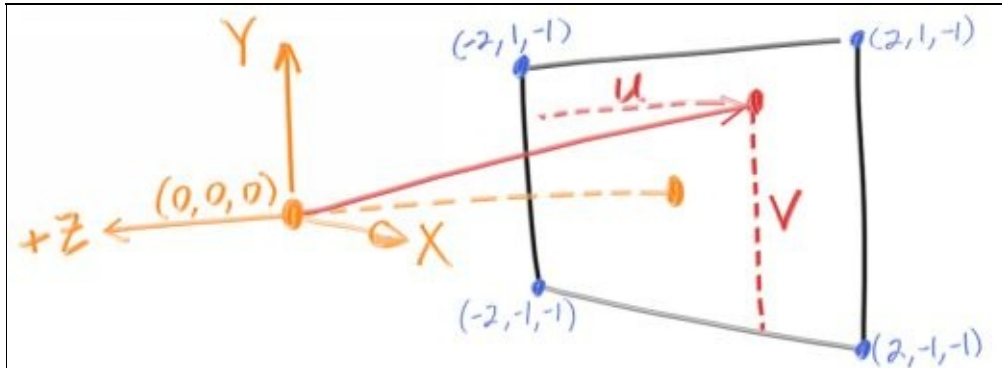
#endif
```

Now we are ready to turn the corner and make a ray tracer. At the core of a ray tracer is to send rays through pixels and compute what color is seen in the direction of those rays. This is of the form *calculate which ray goes from the eye to a pixel, compute what that ray intersects, and 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 *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 stick with a 200x100 image. 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 lower 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.



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

```
#include <iostream>
#include "ray.h"

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

int main() {
    int nx = 200;
    int ny = 100;
    std::cout << "P3\n" << nx << " " << ny << "\n255\n";
    vec3 lower_left_corner(-2.0, -1.0, -1.0);
    vec3 horizontal(4.0, 0.0, 0.0);
    vec3 vertical(0.0, 2.0, 0.0);
    vec3 origin(0.0, 0.0, 0.0);
    for (int j = ny-1; j >= 0; j--) {
        for (int i = 0; i < nx; i++) {
            float u = float(i) / float(nx);
            float v = float(j) / float(ny);
            ray r(origin, lower_left_corner + u*horizontal + v*vertical);
            vec3 col = color(r);
            int ir = int(255.99*col[0]);
            int ig = int(255.99*col[1]);
            int ib = int(255.99*col[2]);

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

The `color(ray)` function linearly blends white and blue depending on the up/downness of the y coordinate. I first made it a unit vector so  $-1.0 < y < 1.0$ . I then did a standard graphics trick of scaling that to  $0.0 < t < 1.0$ . 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:  $blended\_value = (1-t)*start\_value + t*end\_value$ , with  $t$  going from zero to one. In our case this produces:



## Chapter 4: 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. Recall that the equation for a sphere centered at the origin of radius  $R$  is  $x*x + y*y + z*z = R*R$ . The way you can read that equation is “for any  $(x, y, z)$ , if  $x*x + y*y + z*z = R*R$  then  $(x,y,z)$  is on the sphere and otherwise it is not”. It gets uglier if the sphere center is at  $(cx, cy, cz)$ :

$$(x-cx)*(x-cx) + (y-cy)*(y-cy) + (z-cz)*(z-cz) = R*R$$

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  $\mathbf{C} = (cx, cy, cz)$  to point  $\mathbf{p} = (x, y, z)$  is  $(\mathbf{p} - \mathbf{C})$ . And  $dot((\mathbf{p} - \mathbf{C}), (\mathbf{p} - \mathbf{C})) = (y-cy)*(y-cy) + (z-cz)*(z-cz)$ . So the equation of the sphere in vector form is:

$$dot((\mathbf{p} - \mathbf{c}), (\mathbf{p} - \mathbf{c})) = R*R$$

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:

$$\text{dot}((\mathbf{p}(t) - \mathbf{c}), (\mathbf{p}(t) - \mathbf{c})) = R * R$$

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

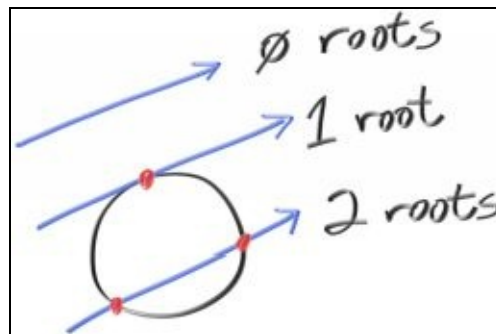
$$\text{dot}((\mathbf{A} + t * \mathbf{B} - \mathbf{C}), (\mathbf{A} + t * \mathbf{B} - \mathbf{C})) = R * R$$

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

$$t * t * \text{dot}(\mathbf{B}, \mathbf{B}) + \boxed{2 * t * \text{dot}(\mathbf{B}, \mathbf{A} - \mathbf{C})} + \text{dot}(\mathbf{C}, \mathbf{C}) - R * R = 0$$

↑

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:



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:

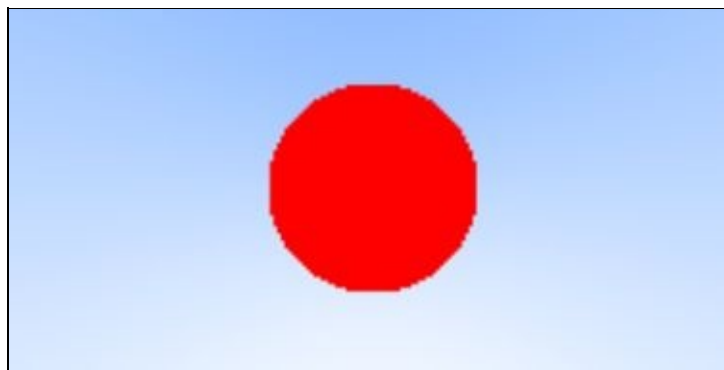
```

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

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

```

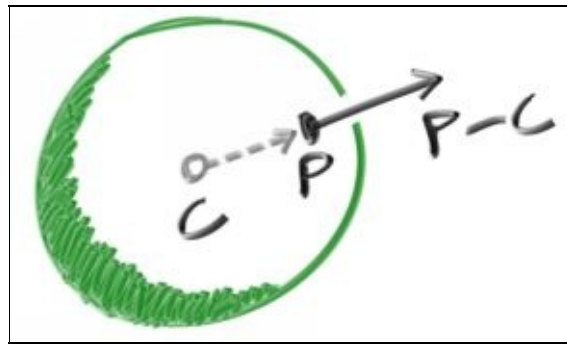
What we get is this:



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.

## Chapter 5: Surface normals and multiple objects.

First, let's get ourselves a surface normal so we can shade. This is a vector that is perpendicular to the surface, and by convention, points out. One design decision is whether these normals (again by convention) 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 normal is in the direction of the hitpoint minus the center:

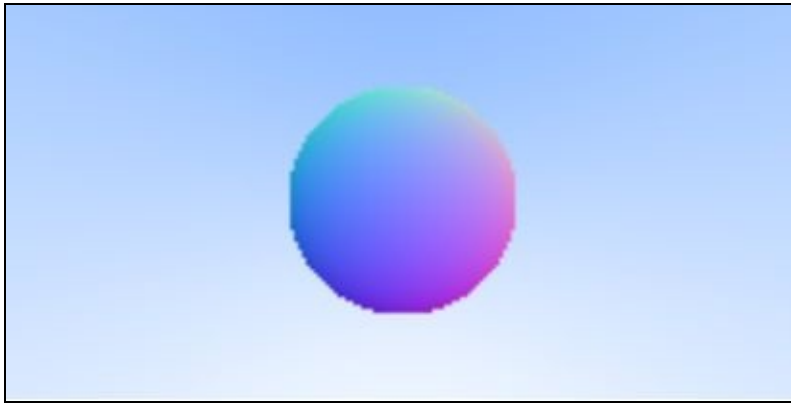


On the earth, this implies that the vector from the earth's center to you points straight up. Let's throw that into the code now, and shade it. We don't have any lights or anything yet, so let's just visualize the normals with a color map. A common trick used for visualizing normals (because it's easy and somewhat intuitive to assume  $N$  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. Let's assume the closest hit point (smallest  $t$ ). These changes in the code let us compute and visualize  $N$ :

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

vec3 color(const ray& r) {
    float t = hit_sphere(vec3(0,0,-1), 0.5, r);
    if (t > 0.0) {
        vec3 N = unit_vector(r.point_at_parameter(t) - vec3(0,0,-1));
        return 0.5*vec3(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)*vec3(1.0, 1.0, 1.0) + t*vec3(0.5, 0.7, 1.0);
}
```

And that yields this picture:



Now, how about several spheres? While it is tempting to have an array of spheres, a very clean solution is to 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. “Hitable” emphasizes the member function that unites them. I don’t love any of these but I will go with “hitable”.

This hitable 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. I know we’ll want motion blur at some point, so I’ll add a time input variable. Here’s the abstract class:

```
#ifndef HITABLEH
#define HITABLEH

#include "ray.h"

struct hit_record {
    float t;
    vec3 p;
    vec3 normal;
};

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

#endif
```

And here’s the sphere (note that I eliminated a bunch of redundant 2’s that cancel each other



```

#ifndef SPHEREH
#define SPHEREH

#include "hitable.h"

class sphere: public hitable {
public:
    sphere() {}
    sphere(vec3 cen, float r) : center(cen), radius(r) {};
    virtual bool hit(const ray& r, float tmin, float tmax, hit_record& rec) const;
    vec3 center;
    float radius;
};

bool sphere::hit(const ray& r, float t_min, float t_max, hit_record& rec) const {
    vec3 oc = r.origin() - center;
    float a = dot(r.direction(), r.direction());
    float b = dot(oc, r.direction());
    float c = dot(oc, oc) - radius*radius;
    float discriminant = b*b - a*c;
    if (discriminant > 0) {
        float temp = (-b - sqrt(b*b-a*c))/a;
        if (temp < t_max && temp > t_min) {
            rec.t = temp;
            rec.p = r.point_at_parameter(rec.t);
            rec.normal = (rec.p - center) / radius;
            return true;
        }
        temp = (-b + sqrt(b*b-a*c))/a;
        if (temp < t_max && temp > t_min) {
            rec.t = temp;
            rec.p = r.point_at_parameter(rec.t);
            rec.normal = (rec.p - center) / radius;
            return true;
        }
    }
    return false;
}

#endif

```

out):

And a list of objects:

```

#ifndef HITABLELISTH
#define HITABLELISTH

#include "hitable.h"

class hitable_list: public hitable {
public:
    hitable_list() {}
    hitable_list(hitable **l, int n) {list = l; list_size = n; }
    virtual bool hit(const ray& r, float tmin, float tmax, hit_record& rec) const;
    hitable **list;
    int list_size;
};

bool hitable_list::hit(const ray& r, float t_min, float t_max, hit_record& rec) const {
    hit_record temp_rec;
    bool hit_anything = false;
    double closest_so_far = t_max;
    for (int i = 0; i < list_size; i++) {
        if (list[i]->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

```

And the new main:

```

#include <iostream>
#include "sphere.h"
#include "hitable_list.h"
#include "float.h"

vec3 color(const ray& r, hitable *world) {
    hit_record rec;
    if (world->hit(r, 0.0, MAXFLOAT, rec)) {
        return 0.5*vec3(rec.normal.x()+1, rec.normal.y()+1, rec.normal.z()+1);
    }
    else {
        vec3 unit_direction = unit_vector(r.direction());
        float t = 0.5*(unit_direction.y() + 1.0);
        return (1.0-t)*vec3(1.0, 1.0, 1.0) + t*vec3(0.5, 0.7, 1.0);
    }
}

int main() {
    int nx = 200;
    int ny = 100;
    std::cout << "P3\n" << nx << " " << ny << "\n255\n";
    vec3 lower_left_corner(-2.0, -1.0, -1.0);
    vec3 horizontal(4.0, 0.0, 0.0);
    vec3 vertical(0.0, 2.0, 0.0);
    vec3 origin(0.0, 0.0, 0.0);
    hitable *list[2];
    list[0] = new sphere(vec3(0,0,-1), 0.5);
    list[1] = new sphere(vec3(0,-100.5,-1), 100);
    hitable *world = new hitable_list(list,2);
    for (int j = ny-1; j >= 0; j--) {
        for (int i = 0; i < nx; i++) {
            float u = float(i) / float(nx);
            float v = float(j) / float(ny);
            ray r(origin, lower_left_corner + u*horizontal + v*vertical);

            vec3 p = r.point_at_parameter(2.0);
            vec3 col = color(r, world);
            int ir = int(255.99*col[0]);
            int ig = int(255.99*col[1]);
            int ib = int(255.99*col[2]);

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

```

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.



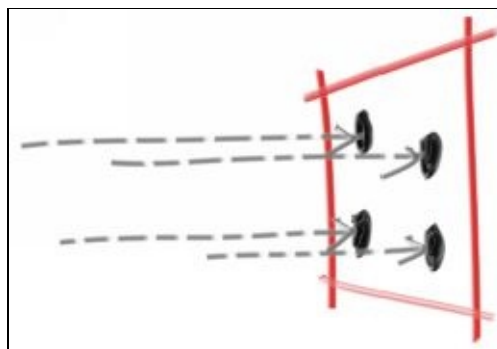
## Chapter 6: Antialiasing

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, which 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.

One thing we need is a random number generator that returns real random numbers. C++ did not traditionally have a standard random number generator but most systems have `drand48()` tucked away someplace and that is what I use here. However, newer versions of C++ have addressed this issue with the `<random>` header (if imperfectly according to some experts). Whatever your infrastructure, find a function that returns a canonical random number which by convention returns random real in the range  $0 \leq \text{ran} < 1$ . The "less than" before the 1 is important as we will sometimes take advantage of that.

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:



Putting that all together yields a camera class encapsulating our simple axis-aligned camera from before:

```

#ifndef CAMERAH
#define CAMERAH

#include "ray.h"

class camera {
public:
    camera() {
        lower_left_corner = vec3(-2.0, -1.0, -1.0);
        horizontal = vec3(4.0, 0.0, 0.0);
        vertical = vec3(0.0, 2.0, 0.0);
        origin = vec3(0.0, 0.0, 0.0);
    }
    ray get_ray(float u, float v) { return ray(origin, lower_left_corner + u*horizontal + v*vertical - origin); }

    vec3 origin;
    vec3 lower_left_corner;
    vec3 horizontal;
    vec3 vertical;
};

#endif

```

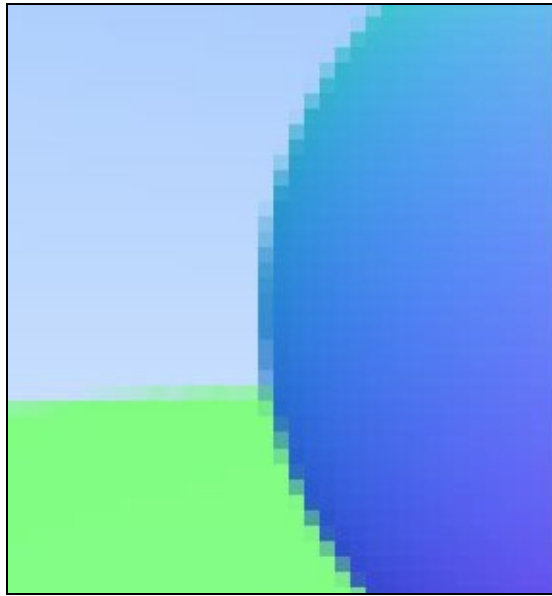
Main is also changed:

```

int main() {
    int nx = 200;
    int ny = 100;
    int ns = 100;
    std::cout << "P3\n" << nx << " " << ny << "\n255\n";
    hitable *list[2];
    list[0] = new sphere(vec3(0,0,-1), 0.5);
    list[1] = new sphere(vec3(0,-100.5,-1), 100);
    hitable *world = new hitable_list(list,2);
    camera cam;
    for (int j = ny-1; j >= 0; j--) {
        for (int i = 0; i < nx; i++) {
            vec3 col(0, 0, 0);
            for (int s=0; s < ns; s++) {
                float u = float(i + drand48()) / float(nx);
                float v = float(j + drand48()) / float(ny);
                ray r = cam.get_ray(u, v);
                vec3 p = r.point_at_parameter(2.0);
                col += color(r, world);
            }
            col /= float(ns);
            int ir = int(255.99*col[0]);
            int ig = int(255.99*col[1]);
            int ib = int(255.99*col[2]);
            std::cout << ir << " " << ig << " " << ib << "\n";
        }
    }
}

```

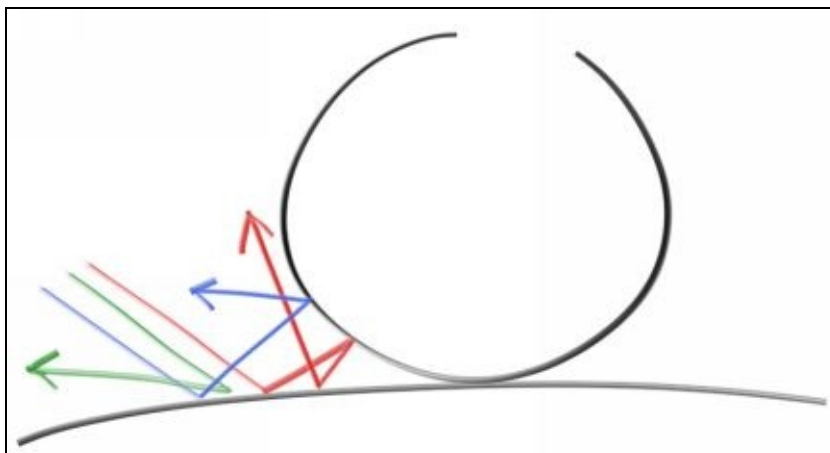
Zooming into the image that is produced, the big change is in edge pixels that are part background and part foreground:



## Chapter 7: 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 can mix and match shapes and materials (so we assign a sphere a material) or if it's put together so the 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.

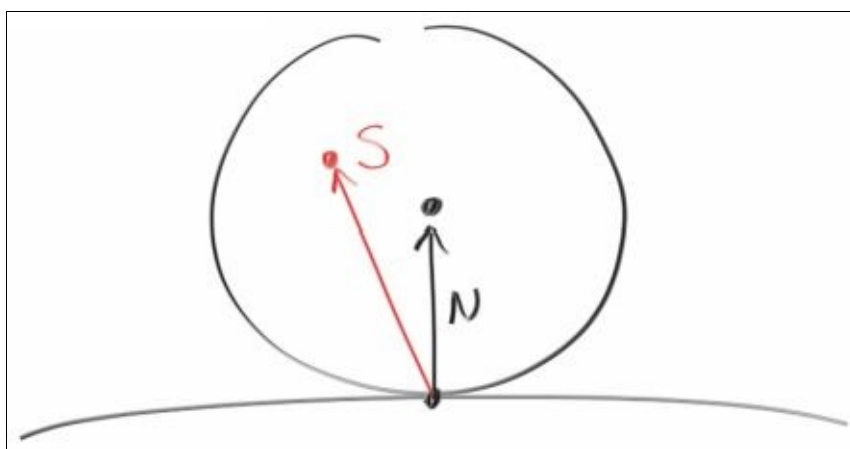
Diffuse objects that don't emit light merely take on the color of their surroundings, but they modulate that with their own intrinsic color. Light that reflects off a diffuse surface has its direction randomized. So, if we send three rays into a crack between two diffuse surfaces they will each have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





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, but a commenter on my blog showed that it was in fact mathematically ideal Lambertian.)

Pick a random point  $s$  from the unit radius sphere that is tangent to the hitpoint, and send a ray from the hitpoint  $p$  to the random point  $s$ . That sphere has center  $(p+N)$ :

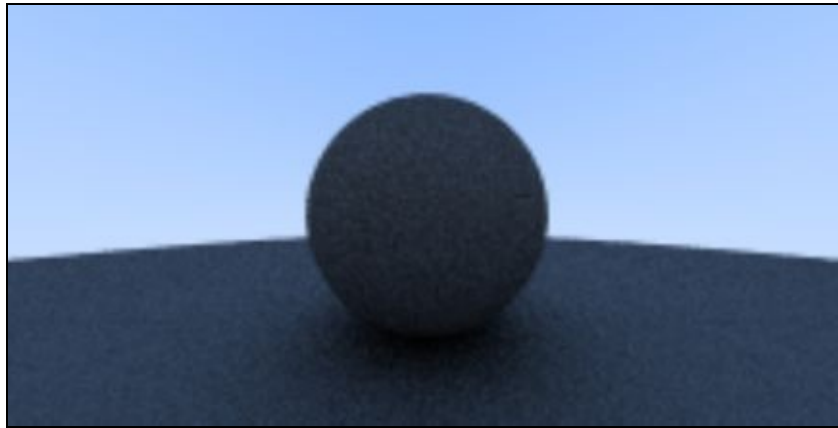


We also need a way to pick a random point in a unit radius sphere centered at the origin. We'll use what is usually the easiest algorithm: a rejection method. First, we pick a random point in the unit cube where  $x$ ,  $y$ , and  $z$  all range from  $-1$  to  $+1$ . We reject this point and try again if the point is outside the sphere. A do/while construct is perfect for that:

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

vec3 color(const ray& r, hitable *world) {
    hit_record rec;
    if (world->hit(r, 0.0, MAXFLOAT, rec)) {
        vec3 target = rec.p + rec.normal + random_in_unit_sphere();
        return 0.5*color( ray(rec.p, target-rec.p), world);
    }
    else {
        vec3 unit_direction = unit_vector(r.direction());
        float t = 0.5*(unit_direction.y() + 1.0);
        return (1.0-t)*vec3(1.0, 1.0, 1.0) + t*vec3(0.5, 0.7, 1.0);
    }
}
```

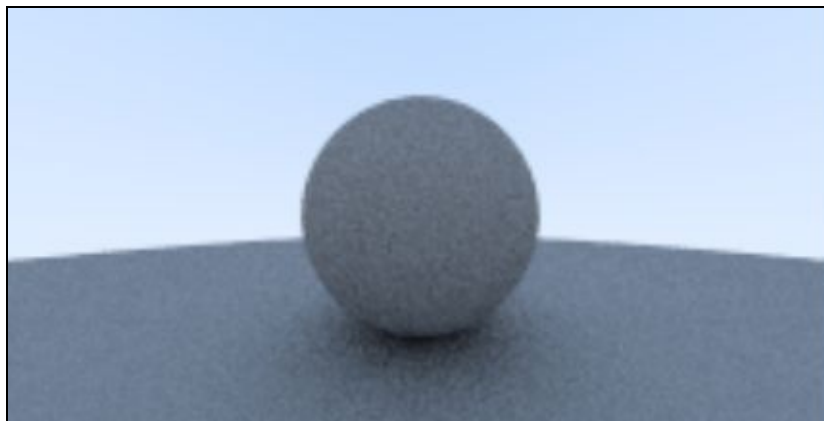
This gives us:



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  $1/\text{gamma}$ , or in our simple case  $1/2$ , which is just square-root:

```
col /= float(ns);
col = vec3( sqrt(col[0]), sqrt(col[1]), sqrt(col[2]) );
int ir = int(255.99*col[0]);
int ig = int(255.99*col[1]);
int ib = int(255.99*col[2]);
std::cout << ir << " " << ig << " " << ib << "\n";
```

That yields light grey, as we desire:



## Chapter 8: Metal

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:

```
class material {
public:
    virtual bool scatter(const ray& r_in, const hit_record& rec, vec3& attenuation, ray& scattered) const = 0;
};
```

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. Hitable 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 hitable class below does:

```
#ifndef HITABLEH
#define HITABLEH
#include "ray.h"

class material;

struct hit_record
{
    float t;
    vec3 p;
    vec3 normal;
    material *mat_ptr;
};

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

#endif
```

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:

```

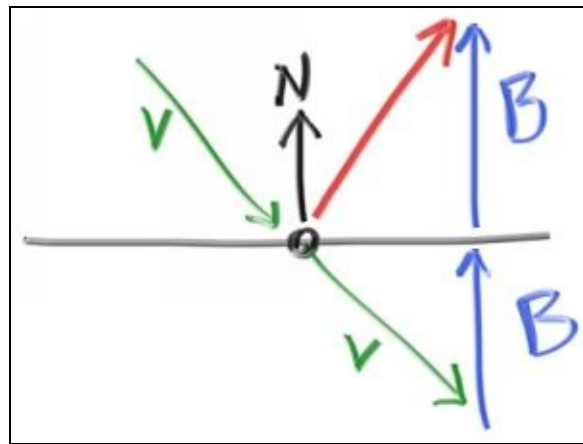
class lambertian : public material {
public:
    lambertian(const vec3& a) : albedo(a) {}
    virtual bool scatter(const ray& r_in, const hit_record& rec, vec3& attenuation, ray& scattered) const {
        vec3 target = rec.p + rec.normal + random_in_unit_sphere();
        scattered = ray(rec.p, target-rec.p);
        attenuation = albedo;
        return true;
    }

    vec3 albedo;
};

```

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

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:



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  $\text{dot}(\mathbf{v}, \mathbf{N})$ . 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;
}

```

The metal material just reflects rays using that formula:

```

class metal : public material {
public:
    metal(const vec3& a) : albedo(a) {}
    virtual bool scatter(const ray& r_in, const hit_record& rec, vec3& attenuation, ray& scattered) const {
        vec3 reflected = reflect(unit_vector(r_in.direction()), rec.normal);
        scattered = ray(rec.p, reflected);
        attenuation = albedo;
        return (dot(scattered.direction(), rec.normal) > 0);
    }

    vec3 albedo;
};

```

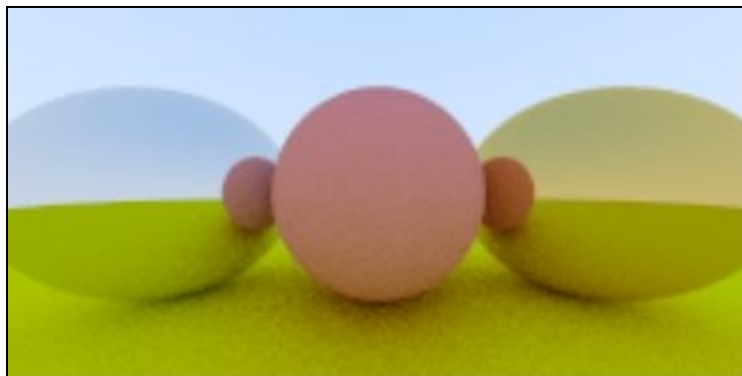
We need to modify the color function to use this:

```
vec3 color(const ray& r, hitable *world, int depth) {
    hit_record rec;
    if (world->hit(r, 0.001, MAXFLOAT, rec)) {
        ray scattered;
        vec3 attenuation;
        if (depth < 50 && rec.mat_ptr->scatter(r, rec, attenuation, scattered)) {
            return attenuation*color(scattered, world, depth+1);
        }
        else {
            return vec3(0,0,0);
        }
    }
    else {
        vec3 unit_direction = unit_vector(r.direction());
        float t = 0.5*(unit_direction.y() + 1.0);
        return (1.0-t)*vec3(1.0, 1.0, 1.0) + t*vec3(0.5, 0.7, 1.0);
    }
}
```

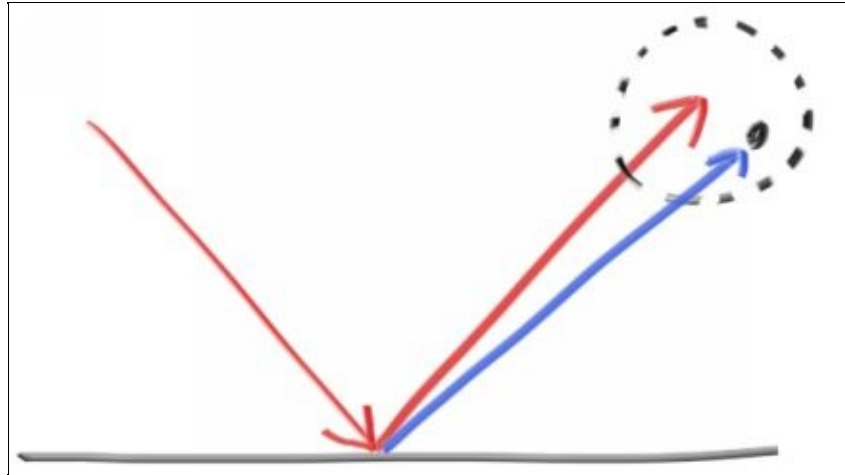
And add some metal spheres:

```
int main() {
    int nx = 200;
    int ny = 100;
    int ns = 100;
    std::cout << "P3\n" << nx << " " << ny << "\n255\n";
    hitable *list[4];
    list[0] = new sphere(vec3(0,0,-1), 0.5, new lambertian(vec3(0.8, 0.3, 0.3)));
    list[1] = new sphere(vec3(0,-100.5,-1), 100, new lambertian(vec3(0.8, 0.8, 0.0)));
    list[2] = new sphere(vec3(1,0,-1), 0.5, new metal(vec3(0.8, 0.6, 0.2)));
    list[3] = new sphere(vec3(-1,0,-1), 0.5, new metal(vec3(0.8, 0.8, 0.8)));
    hitable *world = new hitable_list(list,4);
    camera cam;
    for (int j = ny-1; j >= 0; j--) {
        for (int i = 0; i < nx; i++) {
            vec3 col(0, 0, 0);
            for (int s=0; s < ns; s++) {
                float u = float(i + drand48()) / float(nx);
                float v = float(j + drand48()) / float(ny);
                ray r = cam.get_ray(u, v);
                vec3 p = r.point_at_parameter(2.0);
                col += color(r, world,0);
            }
            col /= float(ns);
            col = vec3( sqrt(col[0]), sqrt(col[1]), sqrt(col[2]) );
            int ir = int(255.99*col[0]);
            int ig = int(255.99*col[1]);
            int ib = int(255.99*col[2]);
            std::cout << ir << " " << ig << " " << ib << "\n";
        }
    }
}
```

Which gives:



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



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. We'll put a maximum of 1 on the radius of the sphere which yields:

```
class metal : public material {
public:
    metal(const vec3& a, float f) : albedo(a) { if (f < 1) fuzz = f; else fuzz = 1; }
    virtual bool scatter(const ray& r_in, const hit_record& rec, vec3& attenuation, ray& scattered) const {
        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);
    }
    vec3 albedo;
    float fuzz;
};
```

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

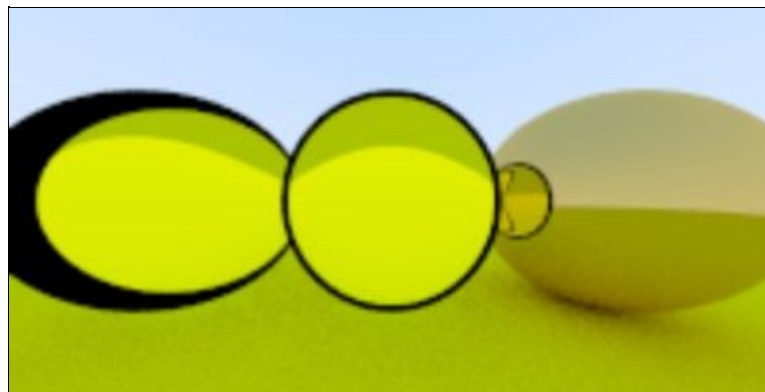




## Chapter 9: Dielectrics

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.

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:



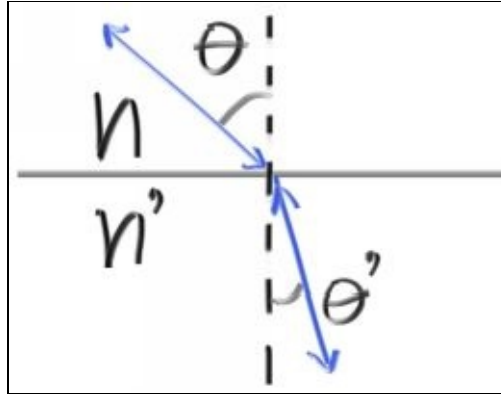
Is that right? Glass balls look odd in real life. But no, it isn't right. The world should be flipped upside down and no weird black stuff. I just printed out the ray straight through the middle of the image and it was clearly wrong. That often does the job.

The refraction is described by Snell's law:

$$n \sin(\theta) = n' \sin(\theta')$$

Where  $n$  and  $n'$  are the refractive indices (typically air = 1, glass = 1.3-1.7, diamond = 2.4)

and the geometry is:



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. 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. The code for refraction is thus a bit more complicated than for reflection:

$$\begin{aligned} \sin\theta &= n_i \text{ over } n_t * \sin\theta' \\ \cos\theta' &= dt \quad \rightarrow \quad \sin\theta' = \sqrt{1 - dt * dt} \\ \sin\theta * \sin\theta &= n_i \text{ over } n_t * n_i \text{ over } n_t * (1 - dt * dt) \leq 1 \end{aligned}$$

```
bool refract(const vec3& v, const vec3& n, float ni_over_nt, vec3& refracted) {
    vec3 uv = unit_vector(v);
    float dt = dot(uv, n);
    float discriminant = 1.0 - ni_over_nt*ni_over_nt*(1-dt*dt);
    if (discriminant > 0) {
        refracted = ni_over_nt*(v - n*dt) - n*sqrt(discriminant);
        return true;
    }
    else
        return false;
}
```

uv

And the dielectric material that always refracts when possible is:

```

class dielectric : public material {
public:
    dielectric(float ri) : ref_idx(ri) {}
    virtual bool scatter(const ray& r_in, const hit_records& rec, vec3& attenuation, ray& scattered) const {
        vec3 outward_normal;
        vec3 reflected = reflect(r_in.direction(), rec.normal);
        float ni_over_nt;
        attenuation = vec3(1.0, 1.0, 0.0);
        vec3 refracted;
        if (dot(r_in.direction(), rec.normal) > 0) {
            outward_normal = -rec.normal;
            ni_over_nt = ref_idx;
        }
        else {
            outward_normal = rec.normal;
            ni_over_nt = 1.0 / ref_idx;
        }
        if (refract(r_in.direction(), outward_normal, ni_over_nt, refracted)) {
            scattered = ray(rec.p, refracted);
        }
        else {
            scattered = ray(rec.p, reflected);
            return false;
        }
        return true;
    }

    float ref_idx;
};

```

Attenuation is always 1—the glass surface absorbs nothing.

If we try that out with these parameters:

```

list[0] = new sphere(vec3(0,0,-1), 0.5, new lambertian(vec3(0.1, 0.2, 0.5)));
list[1] = new sphere(vec3(0,-100.5,-1), 100, new lambertian(vec3(0.8, 0.8, 0.0)));
list[2] = new sphere(vec3(1,0,-1), 0.5, new metal(vec3(0.8, 0.6, 0.2)));
list[3] = new sphere(vec3(-1,0,-1), 0.5, new dielectric(1.5));

```

We get:



(The reader Becker has pointed out that when there is a reflection ray the function returns false so there are no reflections. He is right and that is why there are none in the image above. I am leaving this in rather than correcting this because it is a very interesting example of a major bug that still leaves a reasonably plausible image. These sleeper bugs are the hardest bugs to find because we humans are not designed to find fault with what we see.)

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 simple and surprisingly simple polynomial approximation by Christophe Schlick:

```
float schlick(float cosine, float ref_idx) {
    float r0 = (1-ref_idx) / (1+ref_idx);
    r0 = r0*r0;
    return r0 + (1-r0)*pow((1 - cosine),5);
}
```

This yields our full glass material:

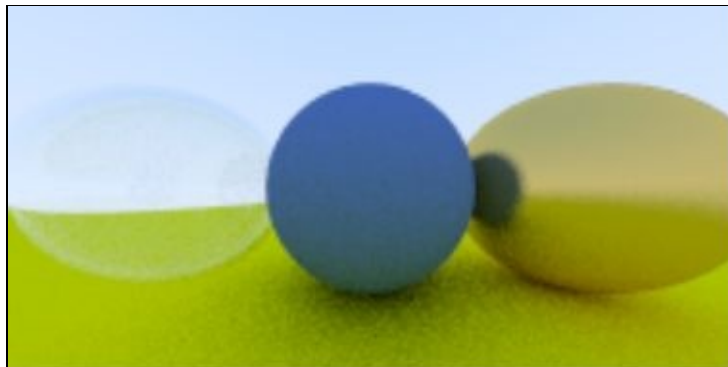
```
class dielectric : public material {
public:
    dielectric(float ri) : ref_idx(ri) {}
    virtual bool scatter(const ray& r_in, const hit_record& rec, vec3& attenuation, ray& scattered) const {
        vec3 outward_normal;
        vec3 reflected = reflect(r_in.direction(), rec.normal);
        float ni_over_nt;
        attenuation = vec3(1.0, 1.0, 1.0);
        vec3 refracted;
        float reflect_prob;
        float cosine;
        if (dot(r_in.direction(), rec.normal) > 0) {
            outward_normal = -rec.normal;
            ni_over_nt = ref_idx;
            cosine = ref_idx * dot(r_in.direction(), rec.normal) / r_in.direction().length();
        }
        else {
            outward_normal = rec.normal;
            ni_over_nt = 1.0 / ref_idx;
            cosine = -dot(r_in.direction(), rec.normal) / r_in.direction().length();
        }
        if (refract(r_in.direction(), outward_normal, ni_over_nt, refracted)) {
            reflect_prob = schlick(cosine, ref_idx);
        }
        else {
            scattered = ray(rec.p, reflected);
            reflect_prob = 1.0;
        }
        if (drand48() < reflect_prob) {
            scattered = ray(rec.p, reflected);
        }
        else {
            scattered = ray(rec.p, refracted);
        }
        return true;
    }

    float ref_idx;
};
```

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, so it can be used as a bubble to make a hollow glass sphere:

```
list[0] = new sphere(vec3(0,0,-1), 0.5, new lambertian(vec3(0.1, 0.2, 0.5)));
list[1] = new sphere(vec3(0,-100.5,-1), 100, new lambertian(vec3(0.8, 0.8, 0.0)));
list[2] = new sphere(vec3(1,0,-1), 0.5, new metal(vec3(0.8, 0.6, 0.2)));
list[3] = new sphere(vec3(-1,0,-1), 0.5, new dielectric(1.5));
list[4] = new sphere(vec3(-1,0,-1), -0.45, new dielectric(1.5));
```

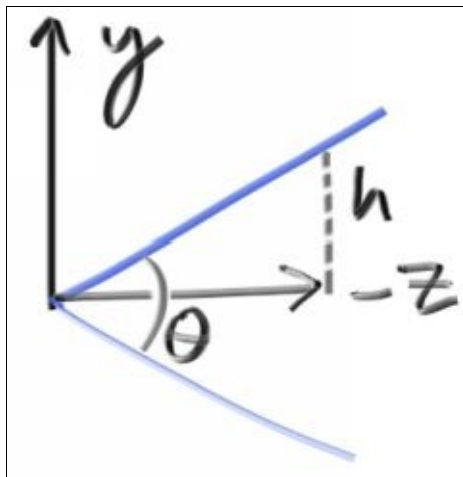
This gives:



## Chapter 10: 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.

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:



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

```
#ifndef CAMERAH
#define CAMERAH

#include "ray.h"

class camera {
public:
    camera(float vfov, float aspect) { // vfov is top to bottom in degrees
        float theta = vfov*M_PI/180;
        float half_height = tan(theta/2);
        float half_width = aspect * half_height;
        lower_left_corner = vec3(-half_width, -half_height, -1.0);
        horizontal = vec3(2*half_width, 0.0, 0.0);
        vertical = vec3(0.0, 2*half_height, 0.0);
        origin = vec3(0.0, 0.0, 0.0);
    }
    ray get_ray(float u, float v) { return ray(origin, lower_left_corner + u*horizontal + v*vertical - origin); }

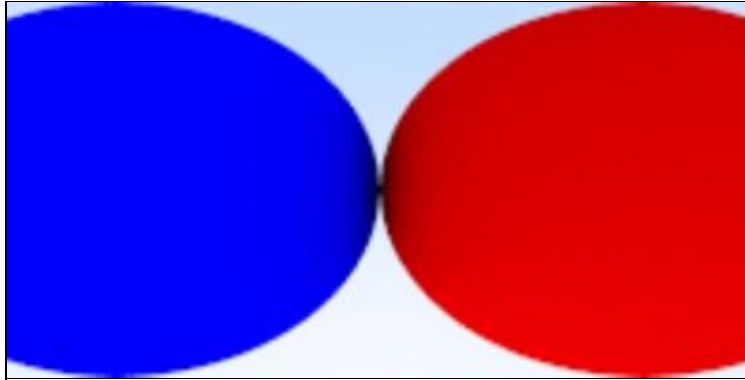
    vec3 origin;
    vec3 lower_left_corner;
    vec3 horizontal;
    vec3 vertical;
};

#endif
```

With calling it camera cam(90, float(nx)/float(ny)) and these spheres:

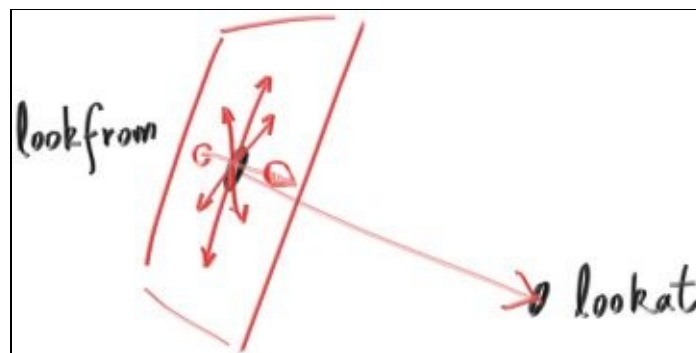
```
float R = cos(M_PI/4);  
list[0] = new sphere(vec3(-R,0,-1), R, new lambertian(vec3(0, 0, 1)));  
list[1] = new sphere(vec3( R,0,-1), R, new lambertian(vec3(1, 0, 0)));  
hitable *world = new hitable_list(list,2);
```

gives:



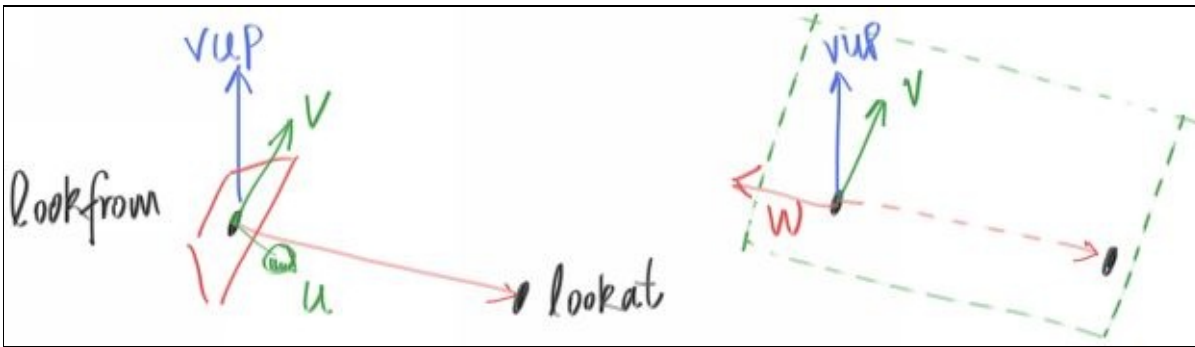
To get an arbitrary viewpoint, let's first name the points we care about. We'll call the position where we place the camera *lookfrom*, and the point we look at *lookat*. (Later, if you want, you could define a direction to look in instead of a point to look at.)

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 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. Notice we already have a plane that the up vector should be in, the plane orthogonal to the 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.





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.

```
#ifndef CAMERAH
#define CAMERAH

#include "ray.h"

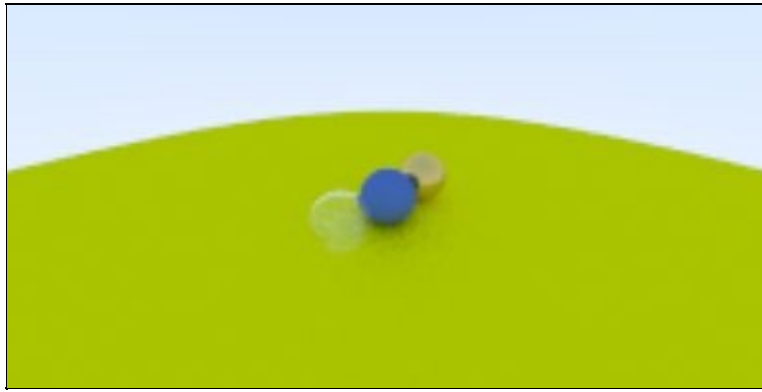
class camera {
public:
    camera(vec3 lookfrom, vec3 lookat, vec3 vup, float vfov, float aspect) { // vfov is top to bottom in degrees
        vec3 u, v, w;
        float theta = vfov*M_PI/180;
        float half_height = tan(theta/2);
        float half_width = aspect * half_height;
        origin = lookfrom;
        w = unit_vector(lookfrom - lookat);
        u = unit_vector(cross(vup, w));
        v = cross(w, u);
        lower_left_corner = vec3(-half_width, -half_height, -1.0);
        lower_left_corner = origin - half_width*u - half_height*v - w;
        horizontal = 2*half_width*u;
        vertical = 2*half_height*v;
    }
    ray get_ray(float s, float t) { return ray(origin, lower_left_corner + s*horizontal + t*vertical - origin); }

    vec3 origin;
    vec3 lower_left_corner;
    vec3 horizontal;
    vec3 vertical;
};
#endif
```

This allows us to change the viewpoint:

```
camera cam(vec3(-2,2,1), vec3(0,0,-1), vec3(0,1,0), 90, float(nx)/float(ny));
```

to get:



And we can change field of view to get:

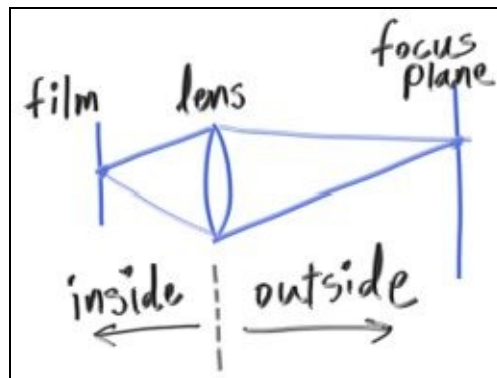


## Chapter 11: Defocus Blur

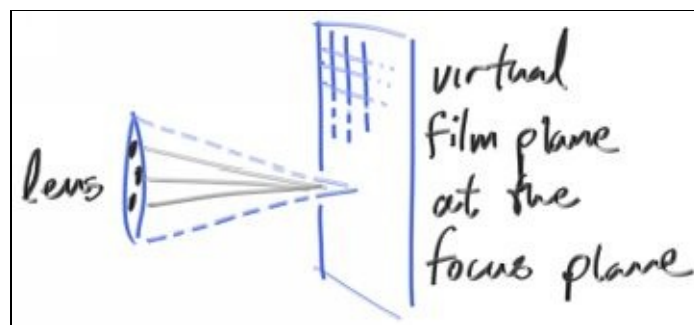
Now our final feature: *defocus blur*. Note, all photographers will call it “depth of field” so be aware of only using “defocus blur” among friends.

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. The distance to that plane where things are in focus 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.

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



We also 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 surface of the lens, and send them toward a virtual film plane, by finding the projection of the film on the plane that is in focus (at the distance `focus_dist`).



For that we just need to have the ray origins be on a disk around lookfrom rather than from a point:

```

#ifndef CAMERAH
#define CAMERAH
#include "ray.h"

vec3 random_in_unit_disk() {
    vec3 p;
    do {
        p = 2.0*vec3(drand48(),drand48(),0) - vec3(1,1,0);
    } while (dot(p,p) >= 1.0);
    return p;
}

class camera {
public:
    camera(vec3 lookfrom, vec3 lookat, vec3 vup, float vfov, float aspect, float aperture, float focus_dist) {
        lens_radius = aperture / 2;
        float theta = vfov*M_PI/180;
        float half_height = tan(theta/2);
        float half_width = aspect * half_height;
        origin = lookfrom;
        w = unit_vector(lookfrom - lookat);
        u = unit_vector(cross(vup, w));
        v = cross(w, u);
        lower_left_corner = origin - half_width*focus_dist*u - half_height*focus_dist*v - focus_dist*w;
        horizontal = 2*half_width*focus_dist*u;
        vertical = 2*half_height*focus_dist*v;
    }
    ray get_ray(float s, float t) {
        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);
    }

    vec3 origin;
    vec3 lower_left_corner;
    vec3 horizontal;
    vec3 vertical;
    vec3 u, v, w;
    float lens_radius;
};
#endif

```

Using a big aperture:

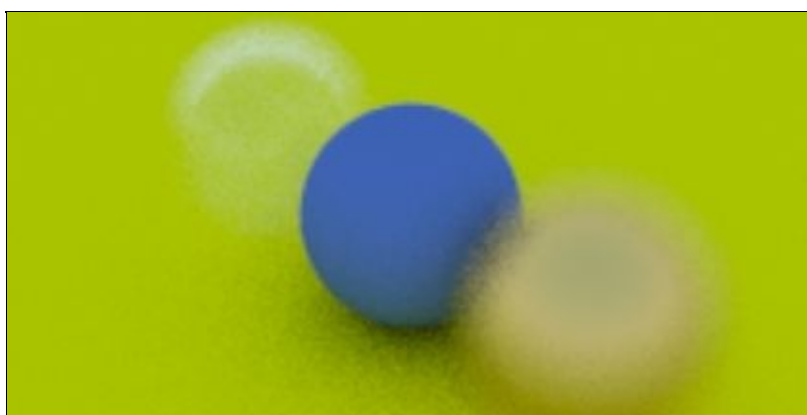
```

vec3 lookfrom(3,3,2);
vec3 lookat(0,0,-1);
float dist_to_focus = (lookfrom-lookat).length();
float aperture = 2.0;

camera cam(lookfrom, lookat, vec3(0,1,0), 20, float(nx)/float(ny), aperture, dist_to_focus);

```

We get:



## Chapter 12: Where next?

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

```

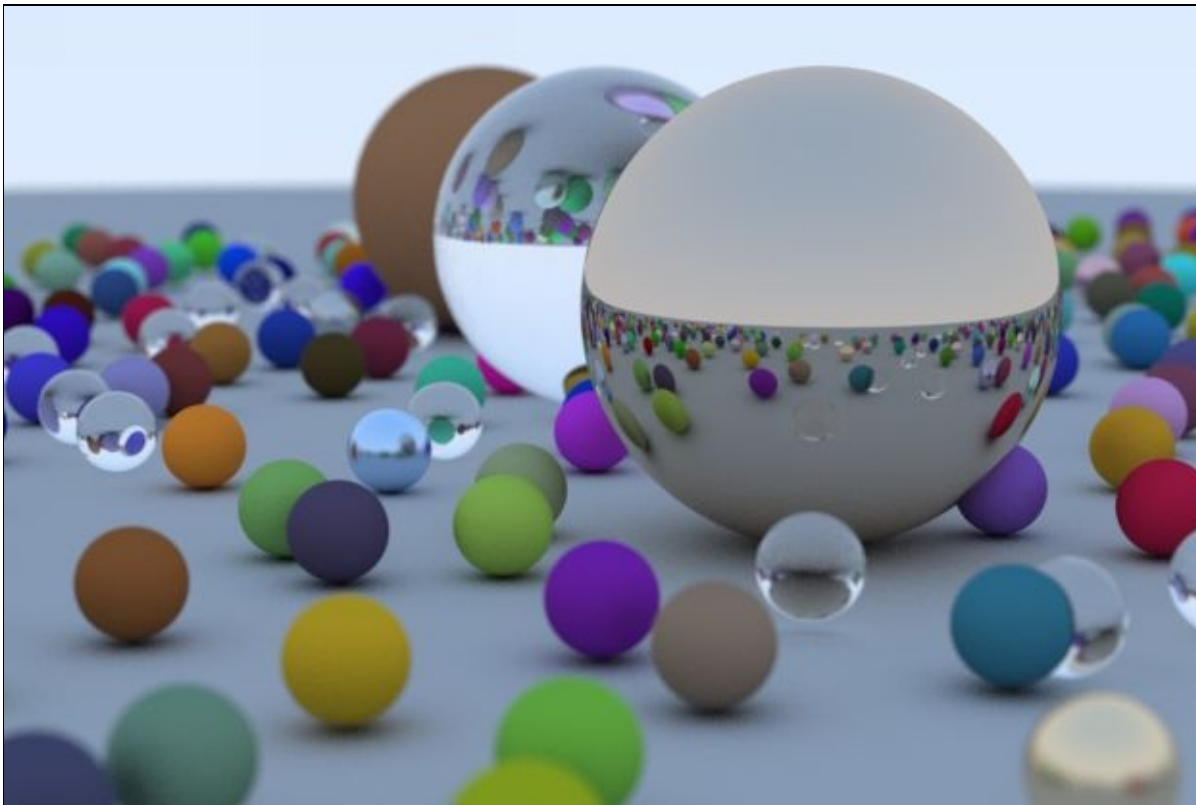
hitable *random_scene() {
    int n = 500;
    hitable **list = new hitable*[n+1];
    list[0] = new sphere(vec3(0,-1000,0), 1000, new lambertian(vec3(0.5, 0.5, 0.5)));
    int i = 1;
    for (int a = -11; a < 11; a++) {
        for (int b = -11; b < 11; b++) {
            float choose_mat = drand48();
            vec3 center(a+0.9*drand48(),0.2,b+0.9*drand48());
            if ((center-vec3(4,0.2,0)).length() > 0.9) {
                if (choose_mat < 0.8) { // diffuse
                    list[i++] = new sphere(center, 0.2, new lambertian(vec3(drand48()*drand48(), drand48()*drand48(), drand48()*drand48())));
                }
                else if (choose_mat < 0.95) { // metal
                    list[i++] = new sphere(center, 0.2,
                        new metal(vec3(0.5*(1 + drand48()), 0.5*(1 + drand48()), 0.5*(1 + drand48()), 0.5*drand48()));
                }
                else { // glass
                    list[i++] = new sphere(center, 0.2, new dielectric(1.5));
                }
            }
        }
    }

    list[i++] = new sphere(vec3(0, 1, 0), 1.0, new dielectric(1.5));
    list[i++] = new sphere(vec3(-4, 1, 0), 1.0, new lambertian(vec3(0.4, 0.2, 0.1)));
    list[i++] = new sphere(vec3(4, 1, 0), 1.0, new metal(vec3(0.7, 0.6, 0.5), 0.0));

    return new hitable_list(list,i);
}

```

This gives:



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.

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,
2. 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.
3. 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.
4. Surface textures. This lets you paste images on like wall paper. Pretty easy and a good thing to do.
5. Solid textures. Ken Perlin has his code online. Andrew Kensler has some very cool info at his blog.
6. Volumes and media. Cool stuff and will challenge your software architecture. I favor making volumes have the hitable interface and probabilistically have intersections based on density. Your rendering code doesn't even have to know it has volumes with that method.
7. 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!

## Acknowledgements

Thanks to readers Becker and Lorenzo Mancini for finding bugs, and to the limnu.com team with help on the figures.