Ray Tracing: The Next Week

Peter Shirley
edited by Steve Hollasch and Trevor David Black

Version 3.1.1, 2020-05-16

Contents

1 Overview

2 Motion Blur
2.1 Introduction of SpaceTime Ray Tracing
2.2 Updating the Camera to Simulate Motion Blur
2.3 Adding Moving Spheres
2.4 Tracking the Time of Ray Intersection
2.5 Putting Everything Together

3 Bounding Volume Hierarchies
3.1 The Key Idea
3.2 Hierarchies of Bounding Volumes
3.3 Axis-Aligned Bounding Boxes (AABBs)
3.4 Ray Intersection with an AABB
3.5 An Optimized AABB Hit Method
3.6 Constructing Bounding Boxes for Hittables
3.7 Creating Bounding Boxes of Lists of Objects
3.8 The BVH Node Class
3.9 Splitting BVH Volumes
3.10 The Box Comparison Functions

4 Solid Textures
4.1 The First Texture Class: Constant Texture
4.2 Texture Coordinates for Spheres
4.3 A Checker Texture
4.4 Rendering a Scene with a Checkered Texture

5 Perlin Noise
5.1 Using Blocks of Random Numbers
5.2 Smoothing out the Result

5.3 Improvement with Hermitian Smoothing
5.4 Tweaking The Frequency
5.5 Using Random Vectors on the Lattice Points
5.6 Introducing Turbulence
5.7 Adjusting the Phase

6 Image Texture Mapping
6.1 Storing Texture Image Data
6.2 Using an Image Texture

7 Rectangles and Lights
7.1 Emissive Materials
7.2 Adding Background Color to the Ray Color Function
7.3 Creating Rectangle Objects
7.4 Turning Objects into Lights
7.5 More Axis-Aligned Rectangles
7.6 Creating an Empty “Cornell Box”
7.7 Flipped Objects

8 Instances
8.1 Intance Translation
8.2 Instance Rotation

9 Volumes
9.1 Constant Density Mediums
9.2 Rendering a Cornell Box with Smoke and Fog Boxes

10 A Scene Testing All New Features

11 Acknowledgments

1. Overview

In Ray Tracing in One Weekend, you built a simple brute force path tracer. In this installment we'll add textures, volumes (like fog), rectangles, instances, lights, and support for lots of objects using a BVH. When done, you'll have a "real" ray tracer.

A heuristic in ray tracing that many people—including me—believe, is that most optimizations complicate the code without delivering much speedup. What I will do in this mini-book is go with the simplest approach in each design decision I make. Check https://in1weekend.blogspot.com/ for readings and references to a more sophisticated approach. However, I strongly encourage you to do no premature optimization; if it doesn't show up high in the execution time profile, it doesn't need optimization until all the features are supported!

The two hardest parts of this book are the BVH and the Perlin textures. This is why the title suggests you take a week rather than a weekend for this endeavor. But you can save those for last if you want a weekend project. Order is not very important for the concepts presented in this book, and without BVH and Perlin texture you will still get a Cornell Box!

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

2. Motion Blur

When you decided to ray trace, you decided that visual quality was worth more than run-time. In your fuzzy reflection and defocus blur you needed multiple samples per pixel. Once you have taken a step down that road, the good news is that almost all effects can be brute-forced. Motion blur is certainly one of those. In a real camera, the shutter opens and stays open for a time interval, and the camera and objects may move during that time. Its really an average of what the camera sees over that interval that we want.

2.1. Introduction of SpaceTime Ray Tracing

We can get a random estimate by sending each ray at some random time when the shutter is open. As long as the objects are where they should be at that time, we can get the right average answer with a ray that is at exactly a single time. This is fundamentally why random ray tracing tends to be simple.

The basic idea is to generate rays at random times while the shutter is open and intersect the model at that one time. The way it is usually done is to have the camera move and the objects move, but have each ray exist at exactly one time. This way the “engine” of the ray tracer can just make sure the objects are where they need to be for the ray, and the intersection guts don’t change much.

For this we will first need to have a ray store the time it exists at:

```
class ray {
    public:
        ray() {}
        ray(const point3& origin, const vec3& direction, double time = 0.0)
            : orig(origin), dir(direction), tm(time)
        {}

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

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

    public:
        point3 orig;
        vec3 dir;
        double tm;
};
```

Listing 1: `[ray.h]` *Ray with time information*

2.2. Updating the Camera to Simulate Motion Blur

Now we need to modify the camera to generate rays at a random time between `time1` and `time2`. Should the camera keep track of `time1` and `time2` or should that be up to the user of camera when a ray is created? When in doubt, I like to make constructors complicated if it makes calls simple, so I will make the camera keep track, but that's a personal preference. Not many changes are needed to camera because for now it is not allowed to move; it just sends out rays over a time period.

```
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,
            double t0 = 0,
            double t1 = 0
        ) {
            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;
            time0 = t0;
            time1 = t1;
        }

        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,
                random_double(time0, time1)
            );
        }

    private:
        point3 origin;
        point3 lower_left_corner;
        vec3 horizontal;
        vec3 vertical;
        vec3 u, v, w;
        double lens_radius;
        double time0, time1;  // shutter open/close times
};
```

Listing 2: `[camera.h]` *Camera with time information*

2.3. Adding Moving Spheres

We also need a moving object. I'll create a sphere class that has its center move linearly from `center0` at `time0` to `center1` at `time1`. Outside that time interval it continues on, so those times need not match up with the camera aperture open and close.

```
class moving_sphere : public hittable {
    public:
        moving_sphere() {}
        moving_sphere(
            point3 cen0, point3 cen1, double t0, double t1, double r,
shared_ptr<material> m)
            : center0(cen0), center1(cen1), time0(t0), time1(t1), radius(r),
mat_ptr(m)
        {};

        virtual bool hit(const ray& r, double tmin, double tmax, hit_record& rec)
const;

        point3 center(double time) const;

    public:
        point3 center0, center1;
        double time0, time1;
        double radius;
        shared_ptr<material> mat_ptr;
};

point3 moving_sphere::center(double time) const{
    return center0 + ((time - time0) / (time1 - time0))*(center1 - center0);
}
```

Listing 3: `[moving_sphere.h]` *A moving sphere*

An alternative to making a new moving sphere class is to just make them all move, while stationary spheres have the same begin and end position. I'm on the fence about that trade-off between fewer classes and more efficient stationary spheres, so let your design taste guide you. The intersection code barely needs a change: `center` just needs to become a function `center(time)`:

```
bool moving_sphere::hit(
    const ray& r, double t_min, double t_max, hit_record& rec) const {
    vec3 oc = r.origin() - center(r.time());
    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) {
        auto root = sqrt(discriminant);

        auto temp = (-half_b - root)/a;
        if (temp < t_max && temp > t_min) {
            rec.t = temp;
            rec.p = r.at(rec.t);
            auto outward_normal = (rec.p - center(r.time())) / radius;
            rec.set_face_normal(r, outward_normal);
            rec.mat_ptr = mat_ptr;
            return true;
        }

        temp = (-half_b + root) / a;
        if (temp < t_max && temp > t_min) {
            rec.t = temp;
            rec.p = r.at(rec.t);
            auto outward_normal = (rec.p - center(r.time())) / radius;
            rec.set_face_normal(r, outward_normal);
            rec.mat_ptr = mat_ptr;
            return true;
        }
    }
    return false;
}
```

Listing 4: `[moving-sphere.h]` *Moving sphere hit function*

2.4. Tracking the Time of Ray Intersection

Be sure that in the materials you have the scattered rays be at the time of the incident ray.

```
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 {
            vec3 scatter_direction = rec.normal + random_unit_vector();
            scattered = ray(rec.p, scatter_direction, r_in.time());
            attenuation = albedo;
            return true;
        }

        color albedo;
};
```

Listing 5: `[material.h]` *Lambertian matrial for moving objects*

2.5. Putting Everything Together

The code below takes the example diffuse spheres from the scene at the end of the last book, and makes them move during the image render. (Think of a camera with shutter opening at time 0 and closing at time 1.) Each sphere moves from its center $\mathbf{C}$ at time $t = 0$ to $\mathbf{C} + (0, r/2, 0)$ at time $t = 1$, where r is a random number in $[0, 1)$:

```
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 - vec3(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);
                    auto center2 = center + vec3(0, random_double(0,.5), 0);
                    world.add(make_shared<moving_sphere>(
                        center, center2, 0.0, 1.0, 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;
}
```

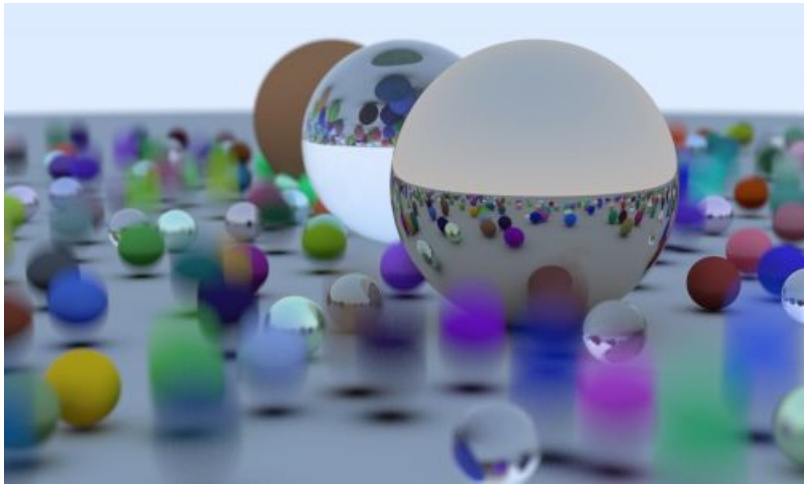
Listing 6: `[main.cc]` *Last book's final scene, but with moving spheres*

And with these viewing parameters:

```
const auto aspect_ratio = double(image_width) / image_height;
...
point3 lookfrom(13,2,3);
point3 lookat(0,0,0);
vec3 vup(0,1,0);
auto dist_to_focus = 10.0;
auto aperture = 0.0;

camera cam(lookfrom, lookat, vup, 20, aspect_ratio, aperture, dist_to_focus, 0.0,
1.0);
```

Listing 7: `[main.cc]` *Viewing parameters*

gives the following result:



Bouncing spheres

3. Bounding Volume Hierarchies

This part is by far the most difficult and involved part of the ray tracer we are working on. I am sticking it in this chapter so the code can run faster, and because it refactors `hittable` a little, and when I add rectangles and boxes we won't have to go back and refactor them.

The ray-object intersection is the main time-bottleneck in a ray tracer, and the time is linear with the number of objects. But it's a repeated search on the same model, so we ought to be able to make it a logarithmic search in the spirit of binary search. Because we are sending millions to billions of rays on the same model, we can do an analog of sorting the model, and then each ray intersection can be a sublinear search. The two most common families of sorting are to 1) divide the space, and 2) divide the objects. The latter is usually much easier to code up and just as fast to run for most models.

3.1. The Key Idea

The key idea of a bounding volume over a set of primitives is to find a volume that fully encloses (bounds) all the objects. For example, suppose you computed a bounding sphere of 10 objects. Any ray that misses the bounding sphere definitely misses all ten objects. If the ray hits the bounding sphere, then it might hit one of the ten objects. So the bounding code is always of the form:

```
if (ray hits bounding object)
    return whether ray hits bounded objects
else
    return false
```

A key thing is we are dividing objects into subsets. We are not dividing the screen or the volume. Any object is in just one bounding volume, but bounding volumes can overlap.

3.2. Hierarchies of Bounding Volumes

To make things sub-linear we need to make the bounding volumes hierarchical. For example, if we divided a set of objects into two groups, red and blue, and used rectangular bounding volumes, we'd have:

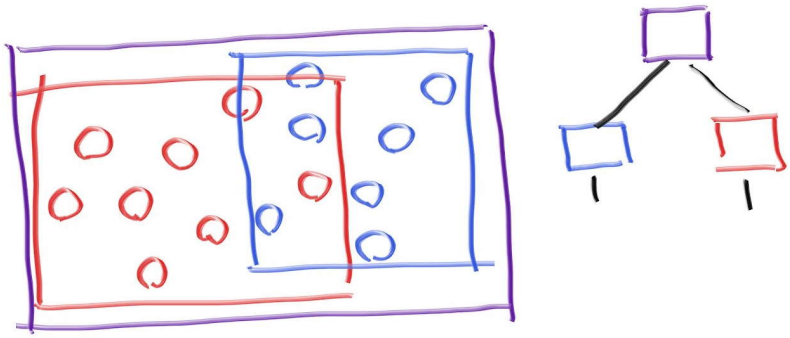


Figure 1: *Bounding volume hierarchy*

Note that the blue and red bounding volumes are contained in the purple one, but they might overlap, and they are not ordered — they are just both inside. So the tree shown on the right has no concept of ordering in the left and right children; they are simply inside. The code would be:

```
if (hits purple)
    hit0 = hits blue enclosed objects
    hit1 = hits red enclosed objects
    if (hit0 or hit1)
        return true and info of closer hit
return false
```

3.3. Axis-Aligned Bounding Boxes (AABBs)

To get that all to work we need a way to make good divisions, rather than bad ones, and a way to intersect a ray with a bounding volume. A ray bounding volume intersection needs to be fast, and bounding volumes need to be pretty compact. In practice for most models, axis-aligned boxes work better than the alternatives, but this design choice is always something to keep in mind if you encounter unusual types of models.

From now on we will call axis-aligned bounding rectangular parallelepiped (really, that is what they need to be called if precise) axis-aligned bounding boxes, or AABBs. Any method you want to use to intersect a ray with an AABB is fine. And all we need to know is whether or not we hit it; we don't need hit points or normals or any of that stuff that we need for an object we want to display.

Most people use the “slab” method. This is based on the observation that an n-dimensional AABB is just the intersection of n axis-aligned intervals, often called “slabs” An interval is just the points between two endpoints, *e.g.*, x such that $3 < x < 5$, or more succinctly x in $(3,5)$. In 2D, two intervals overlapping makes a 2D AABB (a rectangle):

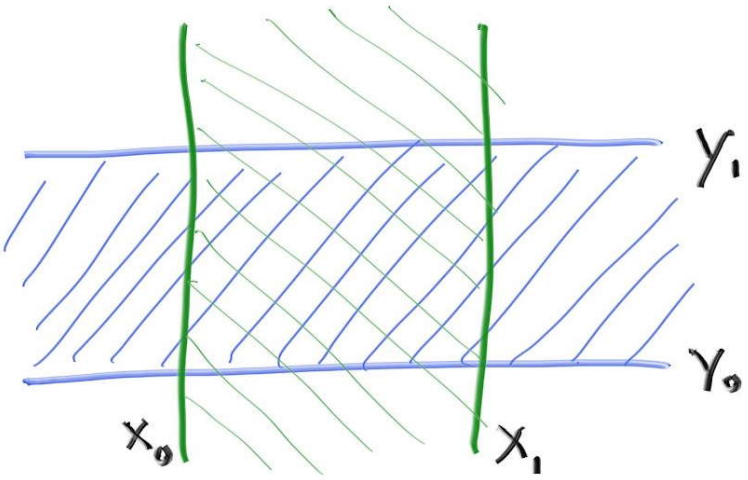


Figure 2: *2D axis-aligned bounding box*

For a ray to hit one interval we first need to figure out whether the ray hits the boundaries. For example, again in 2D, this is the ray parameters t_0 and t_1. (If the ray is parallel to the plane those will be undefined.)

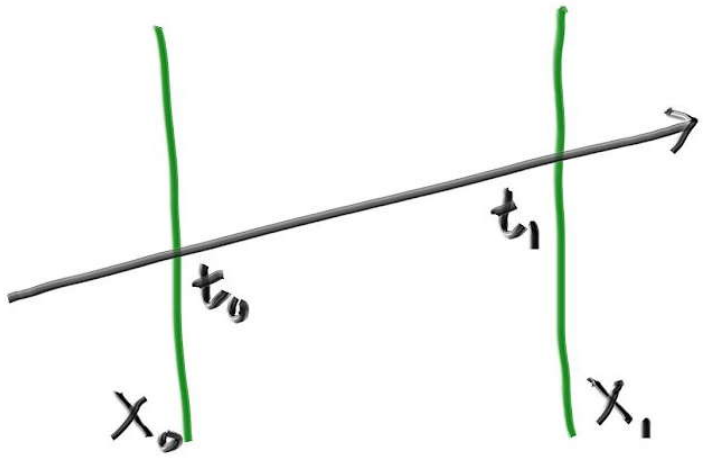


Figure 3: *Ray-slab intersection*

In 3D, those boundaries are planes. The equations for the planes are $x = x_0$, and $x = x_1$. Where does the ray hit that plane? Recall that the ray can be thought of as just a function that given a t returns a location $\mathbf{P}(t)$:

$$\mathbf{P}(t) = \mathbf{A} + t\mathbf{b}$$

That equation applies to all three of the x/y/z coordinates. For example, $x(t) = A_x + tb_x$. This ray hits the plane $x = x_0$ at the t that satisfies this equation:

$$x_0 = A_x + t_0 b_x$$

Thus t at that hitpoint is:

$$t_0 = \frac{x_0 - A_x}{b_x}$$

We get the similar expression for x_1:

$$t_1 = \frac{x_1 - A_x}{b_x}$$

The key observation to turn that 1D math into a hit test is that for a hit, the t-intervals need to overlap. For example, in 2D the green and blue overlapping only happens if there is a hit:

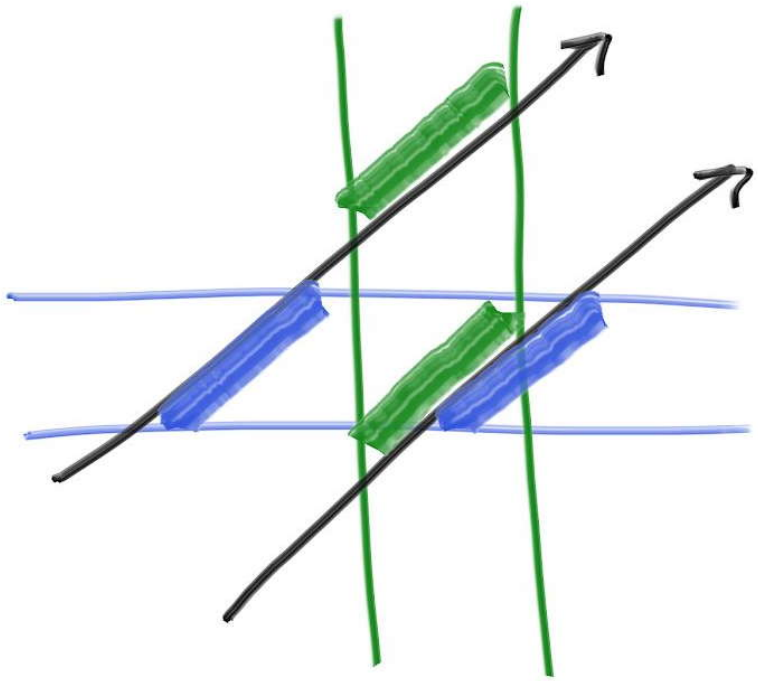


Figure 4: *Ray-slab t-interval overlap*

3.4. Ray Intersection with an AABB

The following pseudocode determines whether the t intervals in the slab overlap:

```
compute (tx0, tx1)
compute (ty0, ty1)
return overlap?( (tx0, tx1), (ty0, ty1))
```

That is awesomely simple, and the fact that the 3D version also works is why people love the slab method:

```
compute (tx0, tx1)
compute (ty0, ty1)
compute (tz0, tz1)
return overlap?( (tx0, tx1), (ty0, ty1), (tz0, tz1))
```

There are some caveats that make this less pretty than it first appears. First, suppose the ray is travelling in the negative x direction. The interval (t_{x0}, t_{x1}) as computed above might be reversed, *e.g.* something like $(7, 3)$. Second, the divide in there could give us infinities. And if the ray origin is on one of the slab boundaries, we can get a `NaN`. There are many ways these issues are dealt with in various ray tracers' AABB. (There are also vectorization issues like SIMD which we will not discuss here. Ingo Wald's papers are a great place to start if you want to go the extra mile in vectorization for speed.) For our purposes, this is unlikely to be a major bottleneck as long as we make it reasonably fast, so let's go for simplest, which is often fastest anyway! First let's look at computing the intervals:

$$t_{x0} = \frac{x_0 - A_x}{b_x}$$

$$t_{x1} = \frac{x_1 - A_x}{b_x}$$

One troublesome thing is that perfectly valid rays will have $b_x = 0$, causing division by zero. Some of those rays are inside the slab, and some are not. Also, the zero will have a ± sign under IEEE floating point. The good news for $b_x = 0$ is that t_{x0} and t_{x1} will both be +∞ or both be -∞ if not between x_0 and x_1. So, using min and max should get us the right answers:

$$t_{x0} = \min(\frac{x_0 - A_x}{b_x}, \frac{x_1 - A_x}{b_x})$$

$$t_{x1} = \max(\frac{x_0 - A_x}{b_x}, \frac{x_1 - A_x}{b_x})$$

The remaining troublesome case if we do that is if $b_x = 0$ and either $x_0 - A_x = 0$ or $x_1 - A_x = 0$ so we get a `NaN`. In that case we can probably accept either hit or no hit answer, but we'll revisit that later.

Now, let's look at that overlap function. Suppose we can assume the intervals are not reversed (so the first value is less than the second value in the interval) and we want to return true in that case. The boolean overlap that also computes the overlap interval (f, F) of intervals (d, D) and (e, E) would be:

```
bool overlap(d, D, e, E, f, F)
    f = max(d, e)
    F = min(D, E)
    return (f < F)
```

If there are any `NaN`s running around there, the compare will return false so we need to be sure our bounding boxes have a little padding if we care about grazing cases (and we probably should because in a ray tracer all cases come up eventually). With all three dimensions in a loop, and passing in the interval $[t_{min}, t_{max}]$, we get:

```
#include "rtweekend.h"

class aabb {
    public:
        aabb() {}
        aabb(const point3& a, const point3& b) { _min = a; _max = b;}

        point3 min() const {return _min; }
        point3 max() const {return _max; }

        bool hit(const ray& r, double tmin, double tmax) const {
            for (int a = 0; a < 3; a++) {
                auto t0 = fmin((_min[a] - r.origin()[a]) / r.direction()[a],
                               (_max[a] - r.origin()[a]) / r.direction()[a]);
                auto t1 = fmax((_min[a] - r.origin()[a]) / r.direction()[a],
                               (_max[a] - r.origin()[a]) / r.direction()[a]);
                tmin = fmax(t0, tmin);
                tmax = fmin(t1, tmax);
                if (tmax <= tmin)
                    return false;
            }
            return true;
        }

        point3 _min;
        point3 _max;
};
```

Listing 8: `[aabb.h]` *Axis-aligned bounding box class*

3.5. An Optimized AABB Hit Method

In reviewing this intersection method, Andrew Kensler at Pixar tried some experiments and proposed the following version of the code. It works extremely well on many compilers, and I have adopted it as my go-to method:

```
inline bool aabb::hit(const ray& r, double tmin, double tmax) const {
    for (int a = 0; a < 3; a++) {
        auto invD = 1.0f / r.direction()[a];
        auto t0 = (min()[a] - r.origin()[a]) * invD;
        auto t1 = (max()[a] - r.origin()[a]) * invD;
        if (invD < 0.0f)
            std::swap(t0, t1);
        tmin = t0 > tmin ? t0 : tmin;
        tmax = t1 < tmax ? t1 : tmax;
        if (tmax <= tmin)
            return false;
    }
    return true;
}
```

Listing 9: `[aabb.h]` *Axis-aligned bounding box hit function*

3.6. Constructing Bounding Boxes for Hittables

We now need to add a function to compute the bounding boxes of all the hittables. Then we will make a hierarchy of boxes over all the primitives, and the individual primitives—like spheres—will live at the leaves. That function returns a bool because not all primitives have bounding boxes (*e.g.*, infinite planes). In addition, moving objects will have a bounding box that encloses the object for the entire time interval [`time1`,`time2`].

```
class hittable {
    public:
        virtual bool hit(
            const ray& r, double t_min, double t_max, hit_record& rec) const = 0;
        virtual bool bounding_box(double t0, double t1, aabb& output_box) const = 0;
};
```

Listing 10: `[hittable.h]` *Hittable class with bounding-box*

For a sphere, that `bounding_box` function is easy:

```
bool sphere::bounding_box(double t0, double t1, aabb& output_box) const {
    output_box = aabb(
        center - vec3(radius, radius, radius),
        center + vec3(radius, radius, radius));
    return true;
}
```

Listing 11: `[sphere.h]` *Sphere with bounding box*

For `moving sphere`, we can take the box of the sphere at t_0, and the box of the sphere at t_1, and compute the box of those two boxes:

```
bool moving_sphere::bounding_box(double t0, double t1, aabb& output_box) const {
    aabb box0(
        center(t0) - vec3(radius, radius, radius),
        center(t0) + vec3(radius, radius, radius));
    aabb box1(
        center(t1) - vec3(radius, radius, radius),
        center(t1) + vec3(radius, radius, radius));
    output_box = surrounding_box(box0, box1);
    return true;
}
```

Listing 12: `[moving_sphere.h]` *Moving sphere with bounding box*

3.7. Creating Bounding Boxes of Lists of Objects

For lists you can store the bounding box at construction, or compute it on the fly. I like doing it the fly because it is only usually called at BVH construction.

```
bool hittable_list::bounding_box(double t0, double t1, aabb& output_box) const {
    if (objects.empty()) return false;

    aabb temp_box;
    bool first_box = true;

    for (const auto& object : objects) {
        if (!object->bounding_box(t0, t1, temp_box)) return false;
        output_box = first_box ? temp_box : surrounding_box(output_box, temp_box);
        first_box = false;
    }

    return true;
}
```

Listing 13: `[hittable_list.h]` *Hittable list with bounding box*

This requires the `surrounding_box` function for `aabb` which computes the bounding box of two boxes:

```
aabb surrounding_box(aabb box0, aabb box1) {
    point3 small(fmin(box0.min().x(), box1.min().x()),
                 fmin(box0.min().y(), box1.min().y()),
                 fmin(box0.min().z(), box1.min().z()));

    point3 big(fmax(box0.max().x(), box1.max().x()),
               fmax(box0.max().y(), box1.max().y()),
               fmax(box0.max().z(), box1.max().z()));

    return aabb(small,big);
}
```

Listing 14: `[aabb.h]` *Surrounding bounding box*

3.8. The BVH Node Class

A BVH is also going to be a `hittable` — just like lists of `hittable`s. It's really a container, but it can respond to the query "does this ray hit you?". One design question is whether we have two classes, one for the tree, and one for the nodes in the tree; or do we have just one class and have the root just be a node we point to. I am a fan of the one class design when feasible. Here is such a class:

```
class bvh_node : public hittable {
    public:
        bvh_node();

        bvh_node(hittable_list& list, double time0, double time1)
            : bvh_node(list.objects, 0, list.objects.size(), time0, time1)
        {}

        bvh_node(
            std::vector<shared_ptr<hittable>>& objects,
            size_t start, size_t end, double time0, double time1);

        virtual bool hit(const ray& r, double tmin, double tmax, hit_record& rec)
const;
        virtual bool bounding_box(double t0, double t1, aabb& output_box) const;

    public:
        shared_ptr<hittable> left;
        shared_ptr<hittable> right;
        aabb box;
};

bool bvh_node::bounding_box(double t0, double t1, aabb& output_box) const {
    output_box = box;
    return true;
}
```

Listing 15: `[bvh.h]` *Bounding volume hierarchy*

Note that the children pointers are to generic hittables. They can be other `bvh_nodes`, or `spheres`, or any other `hittable`.

The `hit` function is pretty straightforward: check whether the box for the node is hit, and if so, check the children and sort out any details:

```
bool bvh_node::hit(const ray& r, double t_min, double t_max, hit_record& rec) const
{
    if (!box.hit(r, t_min, t_max))
        return false;

    bool hit_left = left->hit(r, t_min, t_max, rec);
    bool hit_right = right->hit(r, t_min, hit_left ? rec.t : t_max, rec);

    return hit_left || hit_right;
}
```

Listing 16: `[bvh.h]` *Bounding volume hierarchy hit function*

3.9. Splitting BVH Volumes

The most complicated part of any efficiency structure, including the BVH, is building it. We do this in the constructor. A cool thing about BVHs is that as long as the list of objects in a `bvh_node` gets divided into two sub-lists, the hit function will work. It will work best if the division is done well, so that the two children have smaller bounding boxes than their parent's bounding box, but that is for speed not correctness. I'll choose the middle ground, and at each node split the list along one axis. I'll go for simplicity:

1. randomly choose an axis
2. sort the primitives (`using std::sort`)
3. put half in each subtree

When the list coming in is two elements, I put one in each subtree and end the recursion. The traversal algorithm should be smooth and not have to check for null pointers, so if I just have one element I duplicate it in each subtree. Checking explicitly for three elements and just following one recursion would probably help a little, but I figure the whole method will get optimized later. This yields:

```
#include <algorithm>
...

bvh_node::bvh_node(
    std::vector<shared_ptr<hittable>>& objects,
    size_t start, size_t end, double time0, double time1
) {
    int axis = random_int(0,2);
    auto comparator = (axis == 0) ? box_x_compare
                    : (axis == 1) ? box_y_compare
                                  : box_z_compare;

    size_t object_span = end - start;

    if (object_span == 1) {
        left = right = objects[start];
    } else if (object_span == 2) {
        if (comparator(objects[start], objects[start+1])) {
            left = objects[start];
            right = objects[start+1];
        } else {
            left = objects[start+1];
            right = objects[start];
        }
    } else {
        std::sort(objects.begin() + start, objects.begin() + end, comparator);

        auto mid = start + object_span/2;
        left = make_shared<bvh_node>(objects, start, mid, time0, time1);
        right = make_shared<bvh_node>(objects, mid, end, time0, time1);
    }

    aabb box_left, box_right;

    if (  !left->bounding_box (time0, time1, box_left)
       || !right->bounding_box(time0, time1, box_right)
    )
        std::cerr << "No bounding box in bvh_node constructor.\n";

    box = surrounding_box(box_left, box_right);
}
```

Listing 17: `[bvh.h]` *Bounding volume hierarchy node*

The check for whether there is a bounding box at all is in case you sent in something like an infinite plane that doesn't have a bounding

box. We don't have any of those primitives, so it shouldn't happen until you add such a thing.

3.10. The Box Comparison Functions

Now we need to implement the box comparison functions, used by `std::sort()`. To do this, create a generic comparator returns true if the first argument is less than the second, given an additional axis index argument. Then define axis-specific comparison functions that use the generic comparison function.

```
inline bool box_compare(const shared_ptr<hittable> a, const shared_ptr<hittable> b,
int axis) {
    aabb box_a;
    aabb box_b;

    if (!a->bounding_box(0,0, box_a) || !b->bounding_box(0,0, box_b))
        std::cerr << "No bounding box in bvh_node constructor.\n";

    return box_a.min().e[axis] < box_b.min().e[axis];
}


bool box_x_compare (const shared_ptr<hittable> a, const shared_ptr<hittable> b) {
    return box_compare(a, b, 0);
}

bool box_y_compare (const shared_ptr<hittable> a, const shared_ptr<hittable> b) {
    return box_compare(a, b, 1);
}

bool box_z_compare (const shared_ptr<hittable> a, const shared_ptr<hittable> b) {
    return box_compare(a, b, 2);
}
```

Listing 18: `[bvh.h]` *BVH comparison function, X-axis*

4. Solid Textures

A texture in graphics usually means a function that makes the colors on a surface procedural. This procedure can be synthesis code, or it could be an image lookup, or a combination of both. We will first make all colors a texture. Most programs keep constant rgb colors and textures in different classes, so feel free to do something different, but I am a big believer in this architecture because being able to make any color a texture is great.

4.1. The First Texture Class: Constant Texture

```
#include "rtweekend.h"

class texture {
    public:
        virtual color value(double u, double v, const point3& p) const = 0;
};

class solid_color : public texture {
    public:
        solid_color() {}
        solid_color(color c) : color_value(c) {}

        solid_color(double red, double green, double blue)
          : solid_color(color(red,green,blue)) {}

        virtual color value(double u, double v, const vec3& p) const {
            return color_value;
        }

    private:
        color color_value;
};
```

Listing 19: `[texture.h]` *A texture class*

We'll need to update the `hit_record` structure to store the U,V surface coordinates of the ray-object hit point.

```
struct hit_record {
    vec3 p;
    vec3 normal;
    shared_ptr<material> mat_ptr;
    double t;
    double u;
    double v;
    bool front_face;
    ...
```

Listing 20: `[hittable.h]` *Adding U,V coordinates to the* `hit_record`

We will also need to compute (u, v) texture coordinates for hittables.

4.2. Texture Coordinates for Spheres

For spheres, this is usually based on some form of longitude and latitude, *i.e.*, spherical coordinates. So if we have a (θ, ϕ) in spherical coordinates, we just need to scale θ and ϕ to fractions. If θ is the angle down from the pole, and ϕ is the angle around the axis through the poles, the normalization to $[0, 1]$ would be:

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

$$v = \frac{\theta}{\pi}$$

To compute θ and ϕ, for a given hitpoint, the formula for spherical coordinates of a unit radius sphere on the origin is:

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

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

$$z = \sin(\theta)$$

We need to invert that. Because of the lovely `<cmath>` function `atan2()` which takes any number proportional to sine and cosine and returns the angle, we can pass in x and y (the $\cos(\theta)$ cancel):

$$\phi = \text{atan2}(y, x)$$

The $atan2$ returns values in the range $-\pi$ to π, so we need to take a little care there. The θ is more straightforward:

$$\theta = \text{asin}(z)$$

which returns numbers in the range $-\pi/2$ to $\pi/2$.

So for a sphere, the (u, v) coord computation is accomplished by a utility function that expects things on the unit sphere centered at the origin. The call inside sphere::hit should be:

```
get_sphere_uv((rec.p-center)/radius, rec.u, rec.v);
```

Listing 21: `[sphere.h]` *Sphere UV coordinates from hit*

The utility function is:

```
void get_sphere_uv(const vec3& p, double& u, double& v) {
    auto phi = atan2(p.z(), p.x());
    auto theta = asin(p.y());
    u = 1-(phi + pi) / (2*pi);
    v = (theta + pi/2) / pi;
}
```

Listing 22: `[sphere.h]` *get_sphere_uv function*

Now we can make textured materials by replacing the `const color& a` with a texture pointer:

```
class lambertian : public material {
    public:
        lambertian(shared_ptr<texture> a) : albedo(a) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray&
scattered
        ) const {
            vec3 scatter_direction = rec.normal + random_unit_vector();
            scattered = ray(rec.p, scatter_direction, r_in.time());
            attenuation = albedo->value(rec.u, rec.v, rec.p);
            return true;
        }

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

Listing 23: `[material.h]` *Lambertian material with texture*

Where you used to have code like this:

```
    ...make_shared<lambertian>(color(0.5, 0.5, 0.5))
```

Listing 24: `[main.cc]` *Lambertian material with solid color*

now you should replace the `color(...)` with `make_shared<solid_color>(...)`

```
    ...make_shared<lambertian>(make_shared<solid_color>(0.5, 0.5, 0.5))
```

Listing 25: `[main.cc]` *Lambertian material with texture*

Update all three occurrences of lambertian in the `random_scene()` function in `main.cc`.

4.3. A Checker Texture

We can create a checker texture by noting that the sign of sine and cosine just alternates in a regular way, and if we multiply trig functions in all three dimensions, the sign of that product forms a 3D checker pattern.

```
class checker_texture : public texture {
    public:
        checker_texture() {}
        checker_texture(shared_ptr<texture> t0, shared_ptr<texture> t1): even(t0),
odd(t1) {}

        virtual color value(double u, double v, const point3& p) const {
            auto sines = sin(10*p.x())*sin(10*p.y())*sin(10*p.z());
            if (sines < 0)
                return odd->value(u, v, p);
            else
                return even->value(u, v, p);
        }

    public:
        shared_ptr<texture> odd;
        shared_ptr<texture> even;
}
```

Listing 26: `[texture.h]` *Checkered texture*

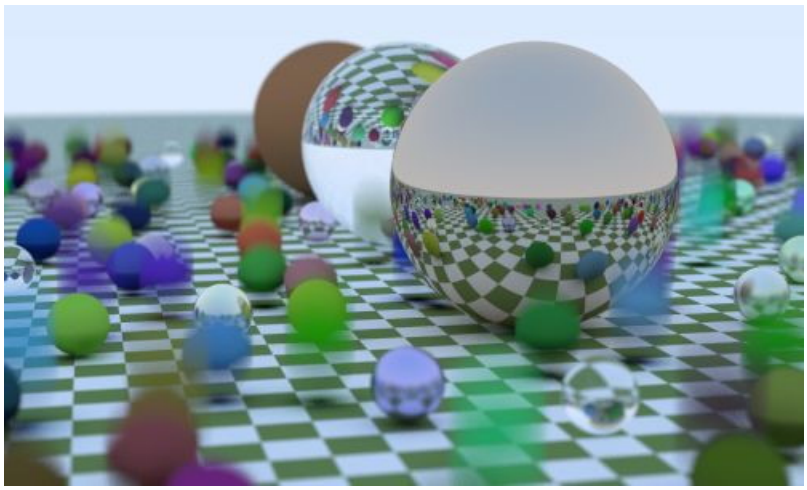
Those checker odd/even pointers can be to a constant texture or to some other procedural texture. This is in the spirit of shader networks introduced by Pat Hanrahan back in the 1980s.

If we add this to our `random_scene()` function's base sphere:

```
auto checker = make_shared<checker_texture>(
    make_shared<solid_color>(0.2, 0.3, 0.1),
    make_shared<solid_color>(0.9, 0.9, 0.9)
);

world.add(make_shared<sphere>(point3(0,-1000,0), 1000, make_shared<lambertian>
(checker)));
```

Listing 27: `[main.cc]` *Checkered texture in use*

We get:



Spheres on checkered ground

4.4. Rendering a Scene with a Checkered Texture

If we add a new scene:

```
hittable_list two_spheres() {
    hittable_list objects;

    auto checker = make_shared<checker_texture>(
        make_shared<solid_color>(0.2, 0.3, 0.1),
        make_shared<solid_color>(0.9, 0.9, 0.9)
    );

    objects.add(make_shared<sphere>(point3(0,-10, 0), 10, make_shared<lambertian>
(checker)));
    objects.add(make_shared<sphere>(point3(0, 10, 0), 10, make_shared<lambertian>
(checker)));

    return objects;
}
```

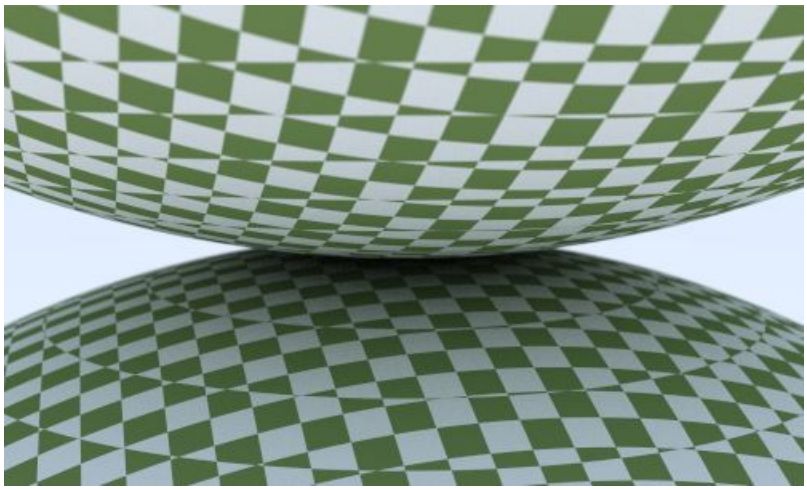
Listing 28: `[main.cc]` *Scene with two checkered spheres*

With camera:

```
const auto aspect_ratio = double(image_width) / image_height;
...
point3 lookfrom(13,2,3);
point3 lookat(0,0,0);
vec3 vup(0,1,0);
auto dist_to_focus = 10.0;
auto aperture = 0.0;

camera cam(lookfrom, lookat, vup, 20, aspect_ratio, aperture, dist_to_focus, 0.0,
1.0);
```

Listing 29: `[main.cc]` *Viewing parameters*

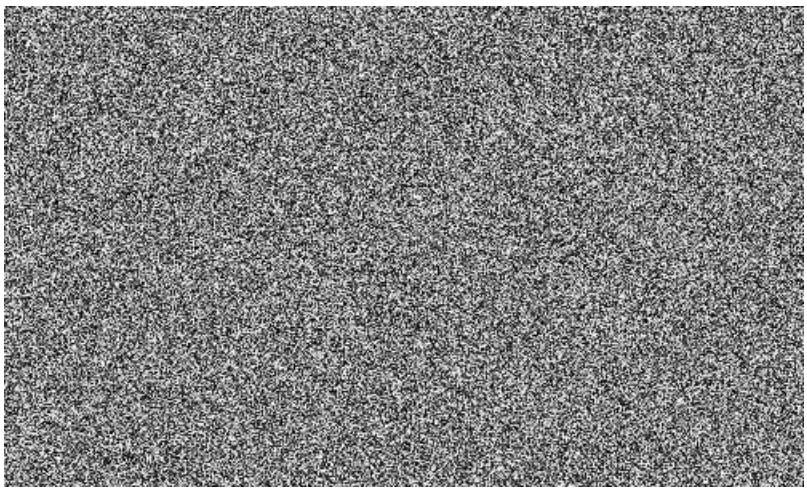
We get:



Checkered spheres

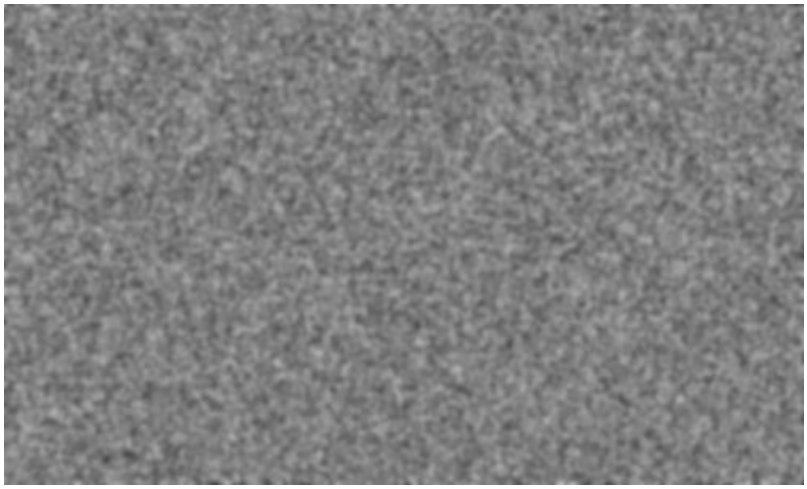
5. Perlin Noise

To get cool looking solid textures most people use some form of Perlin noise. These are named after their inventor Ken Perlin. Perlin texture doesn't return white noise like this:



White noise

Instead it returns something similar to blurred white noise:



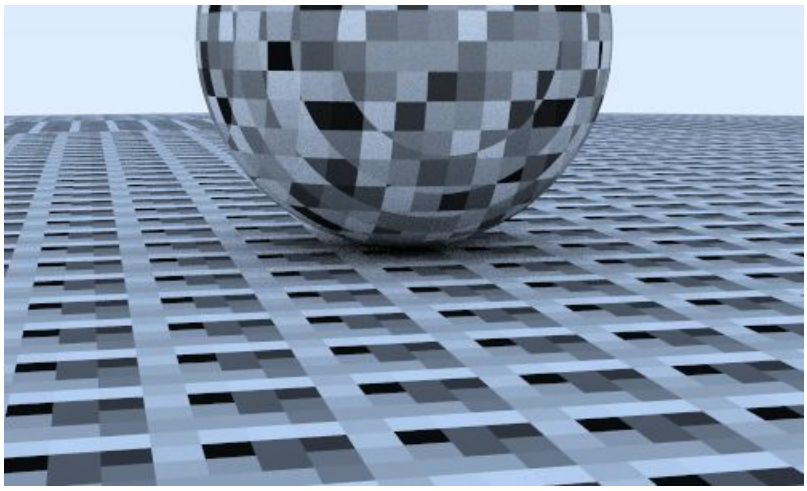
White noise, blurred

A key part of Perlin noise is that it is repeatable: it takes a 3D point as input and always returns the same randomish number. Nearby points return similar numbers. Another important part of Perlin noise is that it

be simple and fast, so it's usually done as a hack. I'll build that hack up incrementally based on Andrew Kensler's description.

5.1. Using Blocks of Random Numbers

We could just tile all of space with a 3D array of random numbers and use them in blocks. You get something blocky where the repeating is clear:



Tiled random patterns

Let's just use some sort of hashing to scramble this, instead of tiling. This has a bit of support code to make it all happen:

```
class perlin {
    public:
        perlin() {
            ranfloat = new double[point_count];
            for (int i = 0; i < point_count; ++i) {
                ranfloat[i] = random_double();
            }

            perm_x = perlin_generate_perm();
            perm_y = perlin_generate_perm();
            perm_z = perlin_generate_perm();
        }

        ~perlin() {
            delete[] ranfloat;
            delete[] perm_x;
            delete[] perm_y;
            delete[] perm_z;
        }

        double noise(const point3& p) const {
            auto u = p.x() - floor(p.x());
            auto v = p.y() - floor(p.y());
            auto w = p.z() - floor(p.z());

            auto i = static_cast<int>(4*p.x()) & 255;
            auto j = static_cast<int>(4*p.y()) & 255;
            auto k = static_cast<int>(4*p.z()) & 255;

            return ranfloat[perm_x[i] ^ perm_y[j] ^ perm_z[k]];
        }

    private:
        static const int point_count = 256;
        double* ranfloat;
        int* perm_x;
        int* perm_y;
        int* perm_z;

        static int* perlin_generate_perm() {
            auto p = new int[point_count];

            for (int i = 0; i < perlin::point_count; i++)
                p[i] = i;

            permute(p, point_count);

            return p;
        }

        static void permute(int* p, int n) {
            for (int i = n-1; i > 0; i--) {
                int target = random_int(0, i);
                int tmp = p[i];
                p[i] = p[target];
                p[target] = tmp;
            }
        }
};
```

Listing 30: `[perlin.h]` *A Perlin texture class and functions*

Now if we create an actual texture that takes these floats between 0 and 1 and creates grey colors:

```
#include "perlin.h"

class noise_texture : public texture {
    public:
        noise_texture() {}

        virtual color value(double u, double v, const point3& p) const {
            return color(1,1,1) * noise.noise(p);
        }

    public:
        perlin noise;
};
```

Listing 31: `[texture.h]` *Noise texture*

We can use that texture on some spheres:

```
hittable_list two_perlin_spheres() {
    hittable_list objects;

    auto pertext = make_shared<noise_texture>();
    objects.add(make_shared<sphere>(point3(0,-1000,0), 1000, make_shared<lambertian>
(pertext)));
    objects.add(make_shared<sphere>(point3(0, 2, 0), 2, make_shared<lambertian>
(pertext)));

    return objects;
}
```

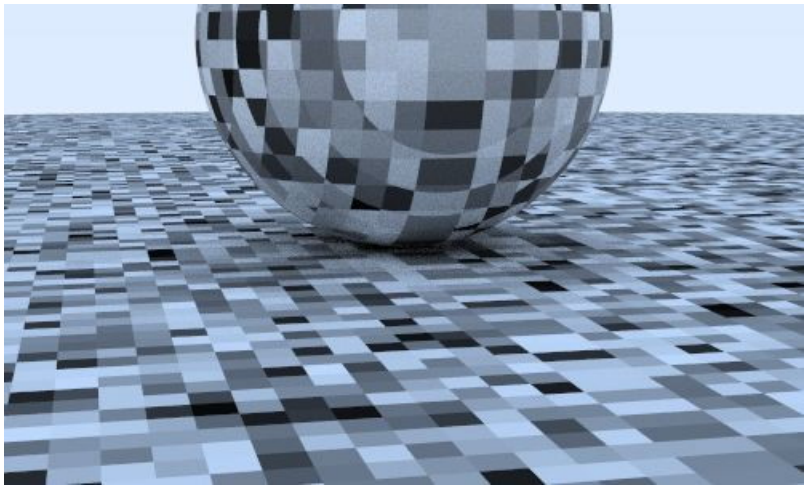
Listing 32: `[main.cc]` *Scene with two Perlin-textured spheres*

With the same camera as before:

```
const auto aspect_ratio = double(image_width) / image_height;
...
point3 lookfrom(13,2,3);
point3 lookat(0,0,0);
vec3 vup(0,1,0);
auto dist_to_focus = 10.0;
auto aperture = 0.0;

camera cam(lookfrom, lookat, vup, 20, aspect_ratio, aperture, dist_to_focus, 0.0,
1.0);
```

Listing 33: `[main.cc]` *Viewing parameters*

Add the hashing does scramble as hoped:



Hashed random texture

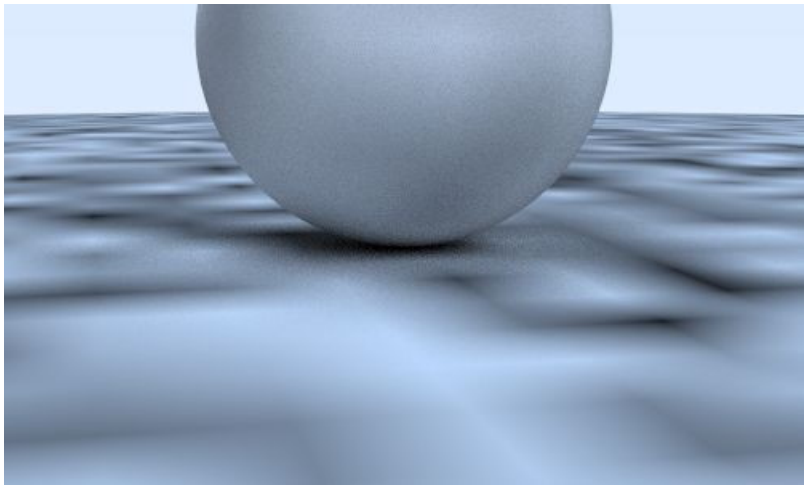
5.2. Smoothing out the Result

To make it smooth, we can linearly interpolate:

```
inline double trilinear_interp(double c[2][2][2], double u, double v, double w) {
    auto accum = 0.0;
    for (int i=0; i < 2; i++)
        for (int j=0; j < 2; j++)
            for (int k=0; k < 2; k++)
                accum += (i*u + (1-i)*(1-u))*
                         (j*v + (1-j)*(1-v))*
                         (k*w + (1-k)*(1-w))*c[i][j][k];

    return accum;
}

class perlin {
    public:
        ...
        double noise(point3 vec3& p) const {
            auto u = p.x() - floor(p.x());
            auto v = p.y() - floor(p.y());
            auto w = p.z() - floor(p.z());
            int i = floor(p.x());
            int j = floor(p.y());
            int k = floor(p.z());
            double c[2][2][2];

            for (int di=0; di < 2; di++)
                for (int dj=0; dj < 2; dj++)
                    for (int dk=0; dk < 2; dk++)
                        c[di][dj][dk] = ranfloat[
                            perm_x[(i+di) & 255] ^
                            perm_y[(j+dj) & 255] ^
                            perm_z[(k+dk) & 255]
                        ];

            return trilinear_interp(c, u, v, w);
        }
        ...
    }
```

Listing 34: `[perlin.h]` *Perlin with trilienear interpolation*

And we get:



Perlin texture with trilinear interpolation

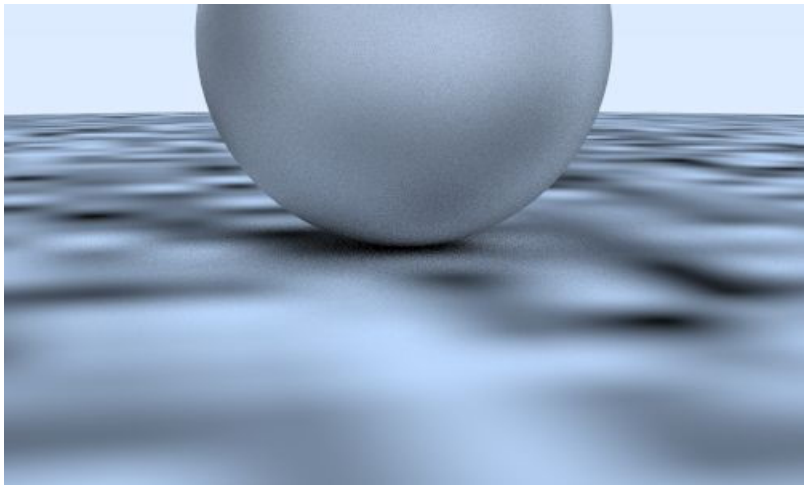
5.3. Improvement with Hermitian Smoothing

Smoothing yields an improved result, but there are obvious grid features in there. Some of it is Mach bands, a known perceptual artifact of linear interpolation of color. A standard trick is to use a Hermite cubic to round off the interpolation:

```
class perlin (
    public:
        ...
        double noise(const point3& p) const {
            auto u = p.x() - floor(p.x());
            auto v = p.y() - floor(p.y());
            auto w = p.z() - floor(p.z());
            u = u*u*(3-2*u);
            v = v*v*(3-2*v);
            w = w*w*(3-2*w);

            int i = floor(p.x());
            int j = floor(p.y());
            int k = floor(p.z());
            ...
```

Listing 35: `[perlin.h]` *Perlin smoothed*

This gives a smoother looking image:



Perlin texture, trilinearly interpolated, smoothed

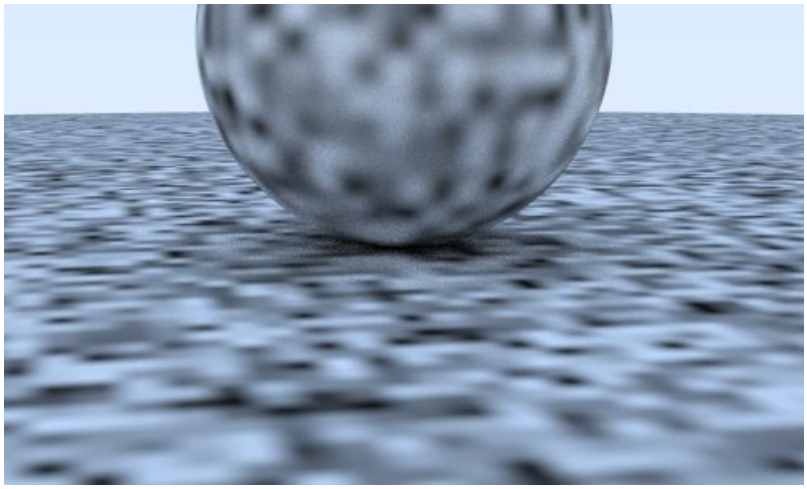
5.4. Tweaking The Frequency

It is also a bit low frequency. We can scale the input point to make it vary more quickly:

```
class noise_texture : public texture {
    public:
        noise_texture() {}
        noise_texture(double sc) : scale(sc) {}

        virtual color value(double u, double v, const point3& p) const {
            return color(1,1,1) * noise.noise(scale * p);
        }

    public:
        perlin noise;
        double scale;
};
```

Listing 36: `[texture.h]` *Perlin smoothed, higher frequency*

which gives:



Perlin texture, higher frequency

5.5. Using Random Vectors on the Lattice Points

This is still a bit blocky looking, probably because the min and max of the pattern always lands exactly on the integer x/y/z. Ken Perlin's very clever trick was to instead put random unit vectors (instead of just floats) on the lattice points, and use a dot product to move the min and max off the lattice. So, first we need to change the random floats to random vectors. These vectors are any reasonable set of irregular directions, and I won't bother to make them exactly uniform:

```
class perlin {
    public:
        perlin() {
            ranvec = new vec3[point_count];

            for (int i = 0; i < point_count; ++i) {
                ranvec[i] = unit_vector(vec3::random(-1,1));
            }

            perm_x = perlin_generate_perm();
            perm_y = perlin_generate_perm();
            perm_z = perlin_generate_perm();
        }

        ~perlin() {
            delete[] ranvec;
            delete[] perm_x;
            delete[] perm_y;
            delete[] perm_z;
        }
    ...
    private:
        vec3* ranvec;
        int* perm_x;
        int* perm_y;
        int* perm_z;
        ...
}
```

Listing 37: `[perlin.h]` *Perlin with random unit translations*

The Perlin class `noise()` method is now:

```
class perlin {
    public:
        ...
        double noise(const point3& p) const {
            auto u = p.x() - floor(p.x());
            auto v = p.y() - floor(p.y());
            auto w = p.z() - floor(p.z());
            int i = floor(p.x());
            int j = floor(p.y());
            int k = floor(p.z());
            vec3 c[2][2][2];

            for (int di=0; di < 2; di++)
                for (int dj=0; dj < 2; dj++)
                    for (int dk=0; dk < 2; dk++)
                        c[di][dj][dk] = ranvec[
                            perm_x[(i+di) & 255] ^
                            perm_y[(j+dj) & 255] ^
                            perm_z[(k+dk) & 255]
                        ];

            return perlin_interp(c, u, v, w);
        }
        ...
    }
```

Listing 38: `[perlin.h]` *Perlin class with new noise() method*

And the interpolation becomes a bit more complicated:

```
class perlin {
    ...
    private:
        ...
        inline double perlin_interp(vec3 c[2][2][2], double u, double v, double w) {
            auto uu = u*u*(3-2*u);
            auto vv = v*v*(3-2*v);
            auto ww = w*w*(3-2*w);
            auto accum = 0.0;

            for (int i=0; i < 2; i++)
                for (int j=0; j < 2; j++)
                    for (int k=0; k < 2; k++) {
                        vec3 weight_v(u-i, v-j, w-k);
                        accum += (i*uu + (1-i)*(1-uu))
                               * (j*vv + (1-j)*(1-vv))
                               * (k*ww + (1-k)*(1-ww))
                               * dot(c[i][j][k], weight_v);
                    }

            return accum;
        }
    ...
}
```

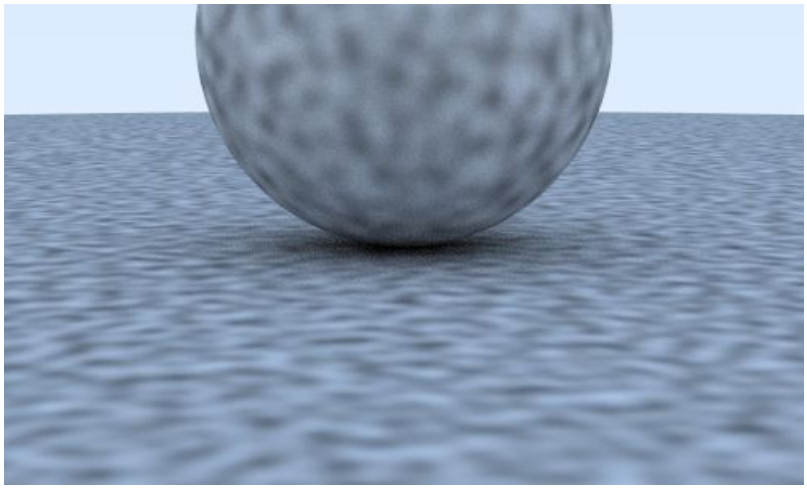
Listing 39: `[perlin.h]` *Perlin interpolation function so far*

The output of the perlin interpretation can return negative values. These negative values will be passed to the `sqrt()` function of our gamma function and get turned into `NaN`s. We will cast the perlin output back to between 0 and 1.

```
class noise_texture : public texture {
    public:
        noise_texture() {}
        noise_texture(double sc) : scale(sc) {}

        virtual color value(double u, double v, const point3& p) const {
            return color(1,1,1) * 0.5 * (1.0 + noise.noise(scale * p));
        }

    public:
        perlin noise;
        double scale;
};
```

Listing 40: `[perlin.h]` *Perlin smoothed, higher frequency*

This finally gives something more reasonable looking:



Perlin texture, shifted off integer values

5.6. Introducing Turbulence

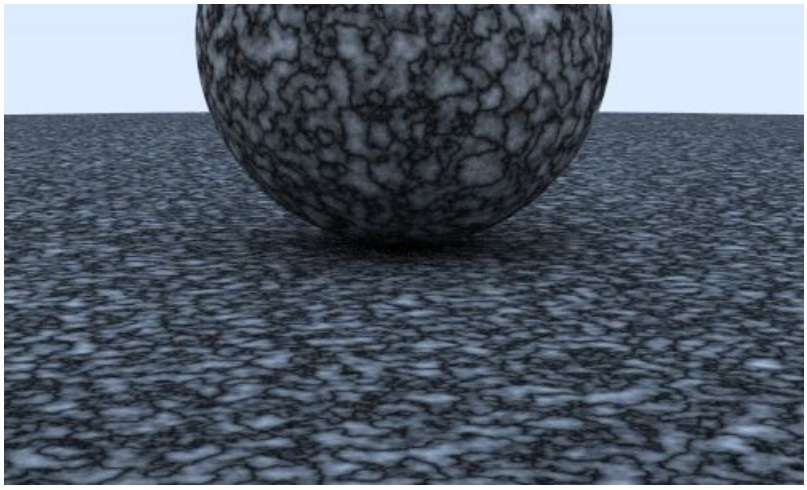
Very often, a composite noise that has multiple summed frequencies is used. This is usually called turbulence, and is a sum of repeated calls to noise:

```
class perlin {
    ...
    public:
        ...
        double turb(const point3& p, int depth=7) const {
            auto accum = 0.0;
            auto temp_p = p;
            auto weight = 1.0;

            for (int i = 0; i < depth; i++) {
                accum += weight*noise(temp_p);
                weight *= 0.5;
                temp_p *= 2;
            }

            return fabs(accum);
        }
        ...
```

Listing 41: `[perlin.h]` *Turbulence function*

Here `fabs()` is the absolute value function defined in `<cmath>`.

Used directly, turbulence gives a sort of camouflage netting appearance:



Perlin texture with turbulence

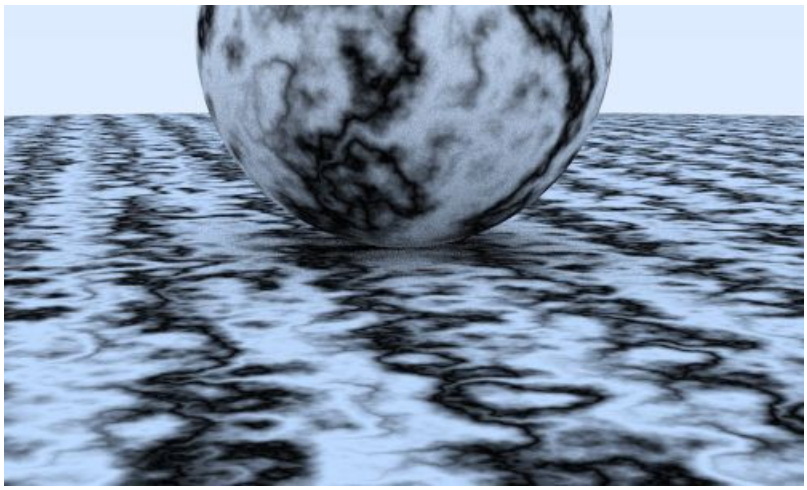
5.7. Adjusting the Phase

However, usually turbulence is used indirectly. For example, the “hello world” of procedural solid textures is a simple marble-like texture. The basic idea is to make color proportional to something like a sine function, and use turbulence to adjust the phase (so it shifts x in $\sin(x)$) which makes the stripes undulate. Commenting out straight noise and turbulence, and giving a marble-like effect is:

```
class noise_texture : public texture {
    public:
        noise_texture() {}
        noise_texture(double sc) : scale(sc) {}

        virtual color value(double u, double v, const point3& p) const {
            return color(1,1,1) * 0.5 * (1 + sin(scale*p.z() + 10*noise.turb(p)));
        }

    public:
        perlin noise;
        double scale;
};
```

Listing 42: `[texture.h]` *Noise texture with turbulence*

Which yields:



Perlin noise, marbled texture

6. Image Texture Mapping

From the hitpoint $\mathbf{P}$, we compute the surface coordinates (u, v). We then use these to index into our procedural solid texture (like marble). We can also read in an image and use the 2D (u, v) texture coordinate to index into the image.

A direct way to use scaled (u, v) in an image is to round the u and v to integers, and use that as (i, j) pixels. This is awkward, because we don't want to have to change the code when we change image resolution. So instead, one of the the most universal unofficial standards in graphics is to use texture coordinates instead of image pixel coordinates. These are just some form of fractional position in the image. For example, for pixel (i, j) in an N_x by N_y image, the image texture position is:

$$u = \frac{i}{N_x - 1}$$

$$v = \frac{j}{N_y - 1}$$

This is just a fractional position.

6.1. Storing Texture Image Data

Now we also need to create a texture class that holds an image. I am going to use my favorite image utility stb_image. It reads in an image into a big array of unsigned char. These are just packed RGBs with each component in the range [0,255] (black to full white). The `image_texture` class uses the resulting image data.

```
#include "rtweekend.h"
#include "rtw_stb_image.h"

#include <iostream>

class image_texture : public texture {
    public:
        const static int bytes_per_pixel = 3;

        image_texture()
          : data(nullptr), width(0), height(0), bytes_per_scanline(0) {}

        image_texture(const char* filename) {
            auto components_per_pixel = bytes_per_pixel;

            data = stbi_load(
                filename, &width, &height, &components_per_pixel,
components_per_pixel);

            if (!data) {
                std::cerr << "ERROR: Could not load texture image file '" <<
filename << "'.\n";
                width = height = 0;
            }

            bytes_per_scanline = bytes_per_pixel * width;
        }

        ~image_texture() {
            delete data;
        }

        virtual color value(double u, double v, const vec3& p) const {
            // If we have no texture data, then return solid cyan as a debugging
aid.
            if (data == nullptr)
                return color(0,1,1);

            // Clamp input texture coordinates to [0,1] x [1,0]
            u = clamp(u, 0.0, 1.0);
            v = 1.0 - clamp(v, 0.0, 1.0);  // Flip V to image coordinates

            auto i = static_cast<int>(u * width);
            auto j = static_cast<int>(v * height);

            // Clamp integer mapping, since actual coordinates should be less than
1.0
            if (i >= width)  i = width-1;
            if (j >= height) j = height-1;

            const auto color_scale = 1.0 / 255.0;
            auto pixel = data + j*bytes_per_scanline + i*bytes_per_pixel;

            return color(color_scale*pixel[0], color_scale*pixel[1],
color_scale*pixel[2]);
        }

    private:
        unsigned char *data;
        int width, height;
        int bytes_per_scanline;
};
```

Listing 43: `[texture.h]` *Image texture class*

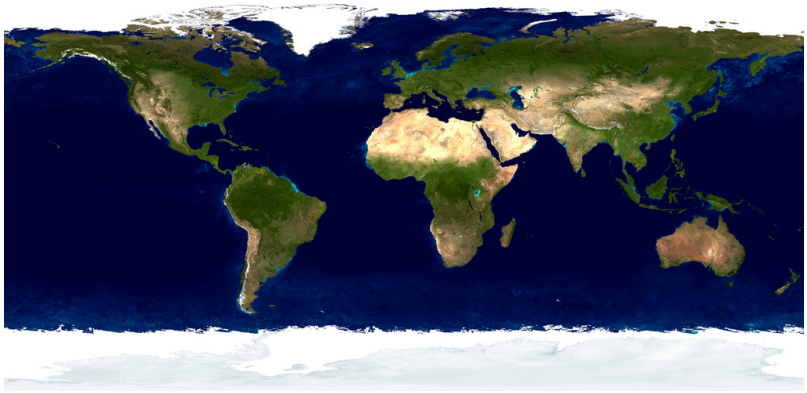
The representation of a packed array in that order is pretty standard. Thankfully, the stb_image package makes that super simple — just include the header `rtw_stb_image.h` (found in the project source code at `src/common/rtw_stb_image.h`) in `main.h`:

```
#include "rtw_stb_image.h"
```

Listing 44: *Including the STB image package*

6.2. Using an Image Texture

I just grabbed a random earth map from the web — any standard projection will do for our purposes.



earthmap.jpg

Here's the code to read an image from a file and then assign it to a diffuse material:

```
hittable_list earth() {
    auto earth_texture = make_shared<image_texture>("earthmap.jpg");
    auto earth_surface = make_shared<lambertian>(earth_texture);
    auto globe = make_shared<sphere>(point3(0,0,0), 2, earth_surface);

    return hittable_list(globe);
}
```

Listing 45: `[main.cc]` *Using stbi_load() to load an image*

We start to see some of the power of all colors being textures — we can assign any kind of texture to the lambertian material, and lambertian doesn't need to be aware of it.

To test this, assign it to a sphere, and then temporarily cripple the `ray_color()` function in main to just return attenuation. You should get something like:

Earth-mapped sphere

7. Rectangles and Lights

Lighting is a key component of raytracing. Early simple raytracers used abstract light sources, like points in space, or directions. Modern approaches have more physically based lights, which have position and size. To create such light sources, we need to be able to take any regular object and turn it into something that emits light into our scene.

7.1. Emissive Materials

First, let's make a light emitting material. We need to add an emitted function (we could also add it to `hit_record instead` — that's a matter of design taste). Like the background, it just tells the ray what color it is and performs no reflection. It's very simple:

```
class diffuse_light : public material  {
    public:
        diffuse_light(shared_ptr<texture> a) : emit(a) {}

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

        virtual color emitted(double u, double v, const point3& p) const {
            return emit->value(u, v, p);
        }

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

Listing 46: `[material.h]` *A diffuse light class*

So that I don't have to make all the non-emitting materials implement `emitted()`, I have the base class return black:

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

Listing 47: `[material.h]` *New emitted function in class material*

7.2. Adding Background Color to the Ray Color Function

Next, we want a pure black background so the only light in the scene is coming from the emitters. To do this, we'll add a background color parameter to our `ray_color` function, and pay attention to the new `emitted` value.

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

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

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

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

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

    return emitted + attenuation * ray_color(scattered, background, world, depth-1);
}
...

int main() {
    ...
    const color background(0,0,0);
    ...
                pixel_color += ray_color(r, background, world, max_depth);
    ...
}
```

Listing 48: `[main.cc]` *ray_color function for emitting materials*

7.3. Creating Rectangle Objects

Now, let's make some rectangles. Rectangles are often convenient for modeling man-made environments. I'm a fan of doing axis-aligned rectangles because they are easy. (We'll get to instancing so we can rotate them later.)

First, here is a rectangle in an xy plane. Such a plane is defined by its z value. For example, $z = k$. An axis-aligned rectangle is defined by the lines $x = x_0$, $x = x_1$, $y = y_0$, and $y = y_1$.

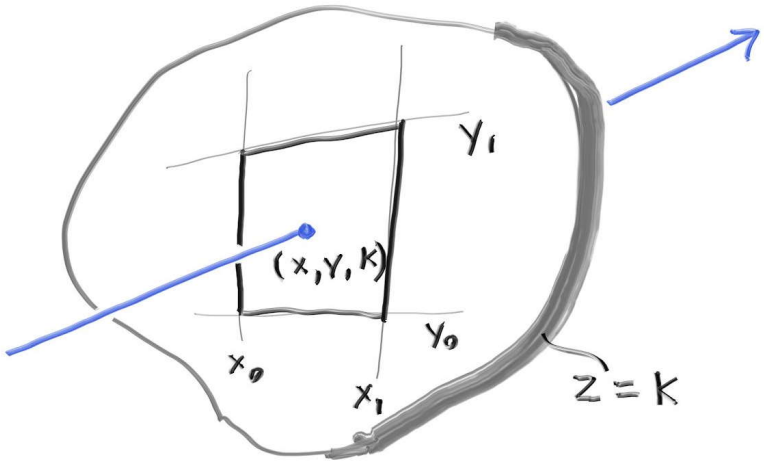


Figure 5: *Ray-rectangle intersection*

To determine whether a ray hits such a rectangle, we first determine where the ray hits the plane. Recall that a ray $\mathbf{P}(t) = \mathbf{A} + t\mathbf{b}$ has its z component defined by $P_z(t) = A_z + tb_z$. Rearranging those terms we can solve for what the t is where $z = k$.

$$t = \frac{k - A_z}{b_z}$$

Once we have t, we can plug that into the equations for x and y:

$$x = A_x + tb_x$$

$$y = A_y + tb_y$$

It is a hit if $x_0 < x < x_1$ and $y_0 < y < y_1$.

Because our rectangles are axis-aligned, their bounding boxes will have an infinitely-thin side. This can be a problem when dividing them up with our axis-aligned bounding volume hierarchy. To counter this, all hittable objects should get a bounding box that has finite width along every dimension. For our rectangles, we'll just pad the box a bit on the infnitely-thin side.

The actual `xy_rect` class is thus:

```
class xy_rect: public hittable {
    public:
        xy_rect() {}

        xy_rect(double _x0, double _x1, double _y0, double _y1, double _k,
shared_ptr<material> mat)
            : x0(_x0), x1(_x1), y0(_y0), y1(_y1), k(_k), mp(mat) {};

        virtual bool hit(const ray& r, double t0, double t1, hit_record& rec) const;

        virtual bool bounding_box(double t0, double t1, aabb& output_box) const {
            // The bounding box must have non-zero width in each dimension, so pad
the Z
            // dimension a small amount.
            output_box = aabb(point3(x0,y0, k-0.0001), point3(x1, y1, k+0.0001));
            return true;
        }

    public:
        shared_ptr<material> mp;
        double x0, x1, y0, y1, k;
};
```

Listing 49: `[aarect.h]` *XY-plane rectangle objects*

And the hit function is:

```
bool xy_rect::hit(const ray& r, double t0, double t1, hit_record& rec) const {
    auto t = (k-r.origin().z()) / r.direction().z();
    if (t < t0 || t > t1)
        return false;
    auto x = r.origin().x() + t*r.direction().x();
    auto y = r.origin().y() + t*r.direction().y();
    if (x < x0 || x > x1 || y < y0 || y > y1)
        return false;
    rec.u = (x-x0)/(x1-x0);
    rec.v = (y-y0)/(y1-y0);
    rec.t = t;
    auto outward_normal = vec3(0, 0, 1);
    rec.set_face_normal(r, outward_normal);
    rec.mat_ptr = mp;
    rec.p = r.at(t);
    return true;
}
```

Listing 50: `[aarect.h]` *Hit function for XY rectangle objects*

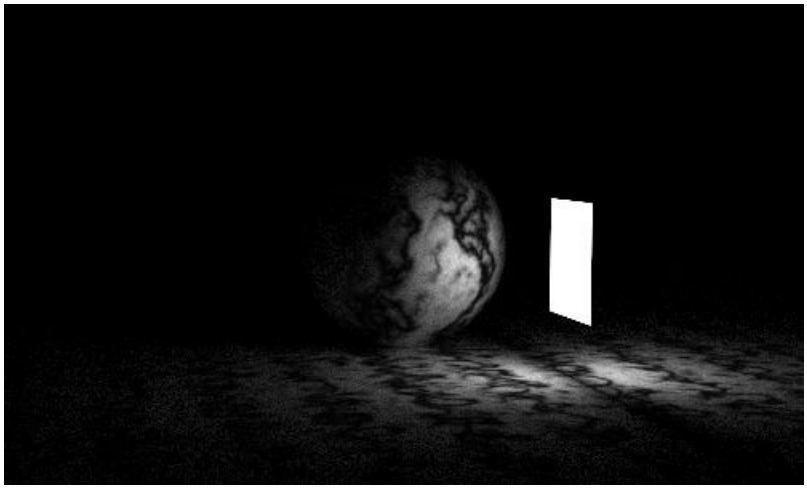
7.4. Turning Objects into Lights

If we set up a rectangle as a light:

```
hittable_list simple_light() {
    hittable_list objects;

    auto pertext = make_shared<noise_texture>(4);
    objects.add(make_shared<sphere>(point3(0,-1000,0), 1000, make_shared<lambertian>
(pertext)));
    objects.add(make_shared<sphere>(point3(0,2,0), 2, make_shared<lambertian>
(pertext)));

    auto difflight = make_shared<diffuse_light>(make_shared<solid_color>(4,4,4));
    objects.add(make_shared<sphere>(point3(0,7,0), 2, difflight));
    objects.add(make_shared<xy_rect>(3, 5, 1, 3, -2, difflight));

    return objects;
}
```

Listing 51: `[main.cc]` *A simple rectangle light*

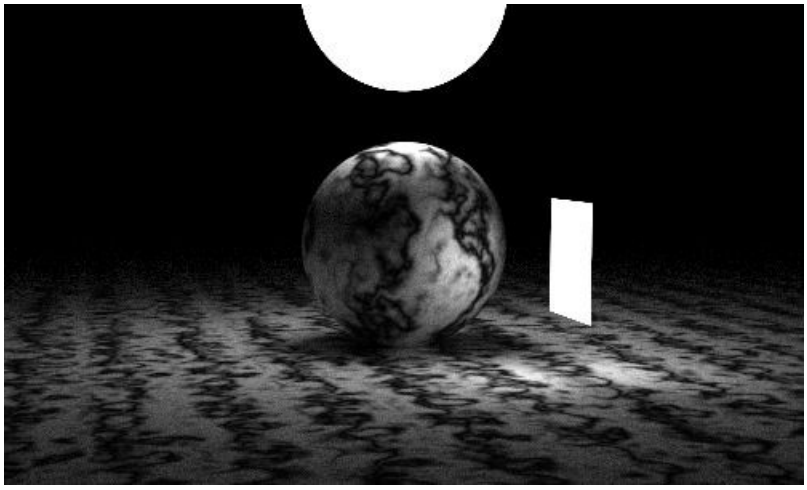
We get:



Scene with rectangle light source

Note that the light is brighter than $(1, 1, 1)$. This allows it to be bright enough to light things.

Fool around with making some spheres lights too.



Scene with rectangle and sphere light sources

7.5. More Axis-Aligned Rectangles

Now let’s add the other two axes and the famous Cornell Box.

This is xz and yz:

```
class xz_rect: public hittable {
    public:
        xz_rect() {}

        xz_rect(double _x0, double _x1, double _z0, double _z1, double _k,
shared_ptr<material> mat)
            : x0(_x0), x1(_x1), z0(_z0), z1(_z1), k(_k), mp(mat) {};

        virtual bool hit(const ray& r, double t0, double t1, hit_record& rec) const;

        virtual bool bounding_box(double t0, double t1, aabb& output_box) const {
            // The bounding box must have non-zero width in each dimension, so pad
the Y
            // dimension a small amount.
            output_box = aabb(point3(x0,k-0.0001,z0), point3(x1, k+0.0001, z1));
            return true;
        }

    public:
        shared_ptr<material> mp;
        double x0, x1, z0, z1, k;
};

class yz_rect: public hittable {
    public:
        yz_rect() {}

        yz_rect(double _y0, double _y1, double _z0, double _z1, double _k,
shared_ptr<material> mat)
            : y0(_y0), y1(_y1), z0(_z0), z1(_z1), k(_k), mp(mat) {};

        virtual bool hit(const ray& r, double t0, double t1, hit_record& rec) const;

        virtual bool bounding_box(double t0, double t1, aabb& output_box) const {
            // The bounding box must have non-zero width in each dimension, so pad
the X
            // dimension a small amount.
            output_box = aabb(point3(k-0.0001, y0, z0), point3(k+0.0001, y1, z1));
            return true;
        }

    public:
        shared_ptr<material> mp;
        double y0, y1, z0, z1, k;
};
```

Listing 52: `[aarect.h]` *XZ and YZ rectangle objects*

With unsurprising hit functions:

```
bool xz_rect::hit(const ray& r, double t0, double t1, hit_record& rec) const {
    auto t = (k-r.origin().y()) / r.direction().y();
    if (t < t0 || t > t1)
        return false;
    auto x = r.origin().x() + t*r.direction().x();
    auto z = r.origin().z() + t*r.direction().z();
    if (x < x0 || x > x1 || z < z0 || z > z1)
        return false;
    rec.u = (x-x0)/(x1-x0);
    rec.v = (z-z0)/(z1-z0);
    rec.t = t;
    auto outward_normal = vec3(0, 1, 0);
    rec.set_face_normal(r, outward_normal);
    rec.mat_ptr = mp;
    rec.p = r.at(t);
    return true;
}

bool yz_rect::hit(const ray& r, double t0, double t1, hit_record& rec) const {
    auto t = (k-r.origin().x()) / r.direction().x();
    if (t < t0 || t > t1)
        return false;
    auto y = r.origin().y() + t*r.direction().y();
    auto z = r.origin().z() + t*r.direction().z();
    if (y < y0 || y > y1 || z < z0 || z > z1)
        return false;
    rec.u = (y-y0)/(y1-y0);
    rec.v = (z-z0)/(z1-z0);
    rec.t = t;
    auto outward_normal = vec3(1, 0, 0);
    rec.set_face_normal(r, outward_normal);
    rec.mat_ptr = mp;
    rec.p = r.at(t);
    return true;
}
```

Listing 53: `[aarect.h]` *XZ and YZ rectangle object hit functions*

7.6. Creating an Empty “Cornell Box”

The “Cornell Box” was introduced in 1984 to model the interaction of light between diffuse surfaces. Let’s make the 5 walls and the light of the box:

```
hittable_list cornell_box() {
    hittable_list objects;

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

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

    return objects;
}
```

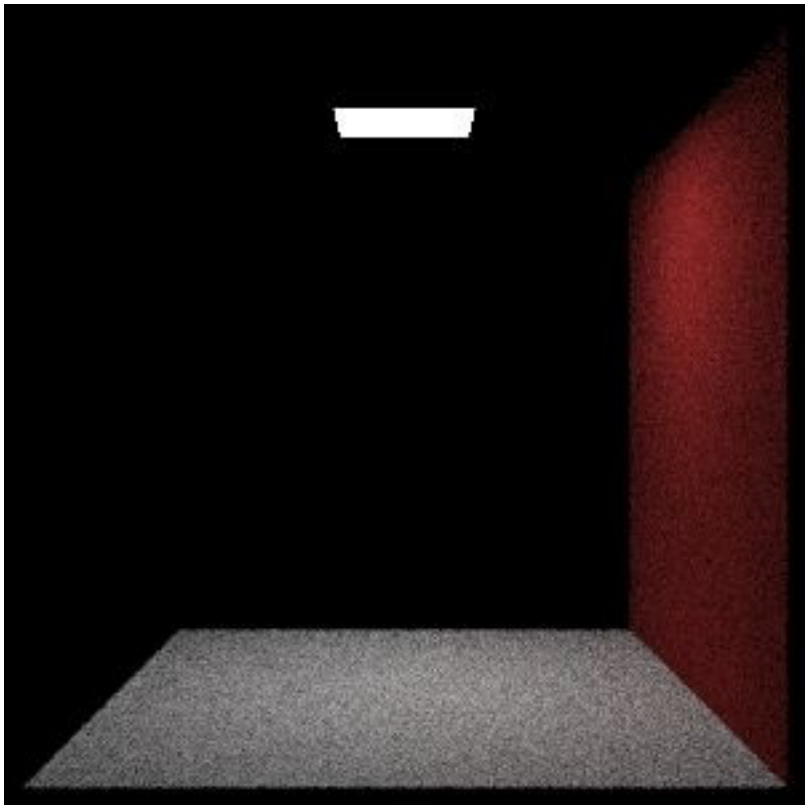
Listing 54: `[main.cc]` *Cornell box scene, empty*

And the view info:

```
const auto aspect_ratio = double(image_width) / image_height;
...
point3 lookfrom(278, 278, -800);
point3 lookat(278,278,0);
vec3 vup(0,1,0);
auto dist_to_focus = 10.0;
auto aperture = 0.0;
auto vfov = 40.0;

camera cam(lookfrom, lookat, vup, vfov, aspect_ratio, aperture, dist_to_focus, 0.0,
1.0);
```

Listing 55: `[main.cc]` *Viewing parameters*

We get:



Empty Cornell box

7.7. Flipped Objects

This is very noisy because the light is small. But we have a problem: some of the walls are facing the wrong way. We haven't specified that a diffuse material should behave differently on different faces of the object, but what if the Cornell box had a different pattern on the inside and outside walls? The rectangle objects are described such that their front faces are always in the directions $(1,0,0)$, $(0,1,0)$, or $(0,0,1)$. We need a way to switch the faces of an object. Let's make a hittable that does nothing but hold another hittable, and flips the face:

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

        virtual bool hit(const ray& r, double t_min, double t_max, hit_record& rec)
const {
            if (!ptr->hit(r, t_min, t_max, rec))
                return false;

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

        virtual bool bounding_box(double t0, double t1, aabb& output_box) const {
            return ptr->bounding_box(t0, t1, output_box);
        }

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

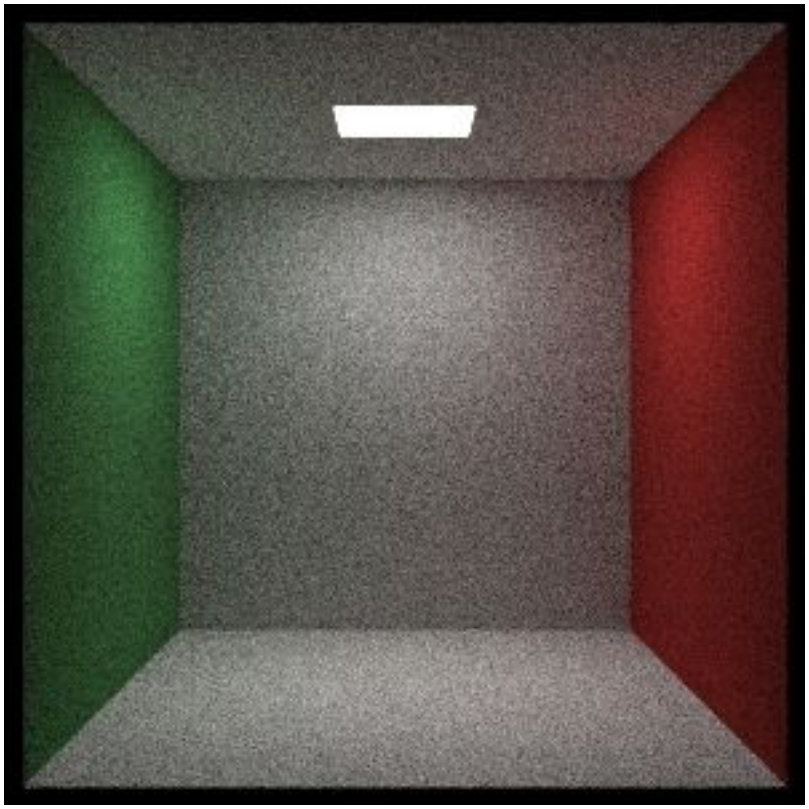
Listing 56: `[hittable.h]` *Flip-Face function*

This makes Cornell:

```
hittable_list cornell_box() {
    hittable_list objects;

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

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

    return objects;
}
```

Listing 57: `[main.cc]` *Empty Cornell box with flipped rectangles*

And voila:



Empty Cornell box with fixed walls

8. Instances

The Cornell Box usually has two blocks in it. These are rotated relative to the walls. First, let's make an axis-aligned block primitive that holds 6 rectangles:

```
class box: public hittable  {
    public:
        box() {}
        box(const point3& p0, const point3& p1, shared_ptr<material> ptr);

        virtual bool hit(const ray& r, double t0, double t1, hit_record& rec) const;

        virtual bool bounding_box(double t0, double t1, aabb& output_box) const {
            output_box = aabb(box_min, box_max);
            return true;
        }

    public:
        point3 box_min;
        point3 box_max;
        hittable_list sides;
};

box::box(const point3& p0, const point3& p1, shared_ptr<material> ptr) {
    box_min = p0;
    box_max = p1;

    sides.add(make_shared<xy_rect>(p0.x(), p1.x(), p0.y(), p1.y(), p1.z(), ptr));
    sides.add(make_shared<flip_face>(
        make_shared<xy_rect>(p0.x(), p1.x(), p0.y(), p1.y(), p0.z(), ptr)));

    sides.add(make_shared<xz_rect>(p0.x(), p1.x(), p0.z(), p1.z(), p1.y(), ptr));
    sides.add(make_shared<flip_face>(
        make_shared<xz_rect>(p0.x(), p1.x(), p0.z(), p1.z(), p0.y(), ptr)));

    sides.add(make_shared<yz_rect>(p0.y(), p1.y(), p0.z(), p1.z(), p1.x(), ptr));
    sides.add(make_shared<flip_face>(
        make_shared<yz_rect>(p0.y(), p1.y(), p0.z(), p1.z(), p0.x(), ptr)));
}

bool box::hit(const ray& r, double t0, double t1, hit_record& rec) const {
    return sides.hit(r, t0, t1, rec);
}
```

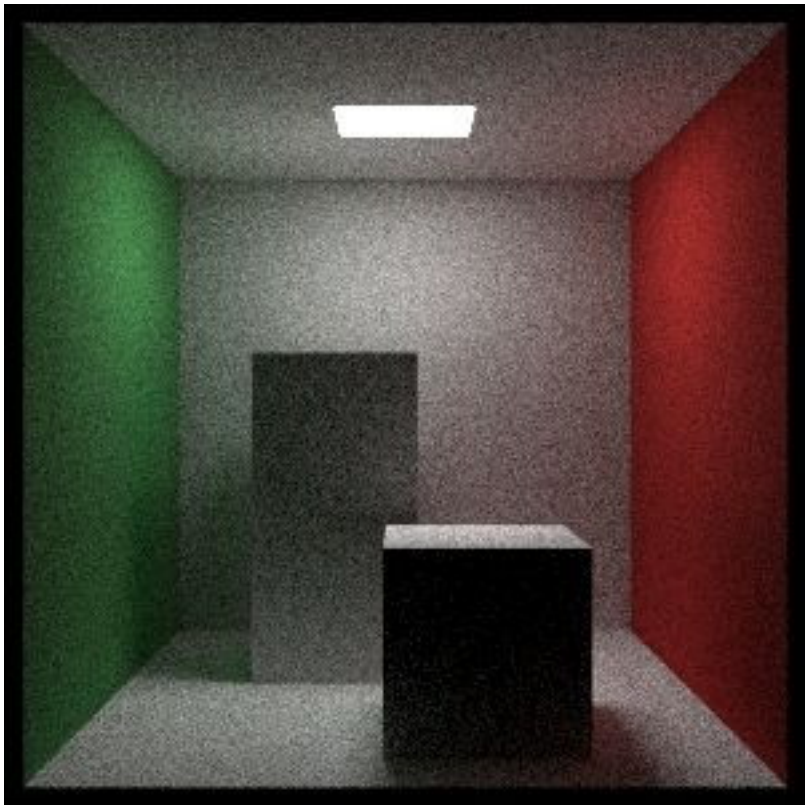
Listing 58: `[box.h]` *A box object*

Now we can add two blocks (but not rotated)

```
objects.add(make_shared<box>(point3(130, 0, 65), point3(295, 165, 230), white));
objects.add(make_shared<box>(point3(265, 0, 295), point3(430, 330, 460), white));
```

Listing 59: `[main.cc]` *Adding box objects*

This gives:



Cornell box with two blocks

Now that we have boxes, we need to rotate them a bit to have them match the *real* Cornell box. In ray tracing, this is usually done with an *instance*. An instance is a geometric primitive that has been moved or rotated somehow. This is especially easy in ray tracing because we don't move anything; instead we move the rays in the opposite direction. For example, consider a *translation* (often called a *move*). We could take the pink box at the origin and add 2 to all its x components, or (as we almost always do in ray tracing) leave the box where it is, but in its hit routine subtract 2 off the x-component of the ray origin.

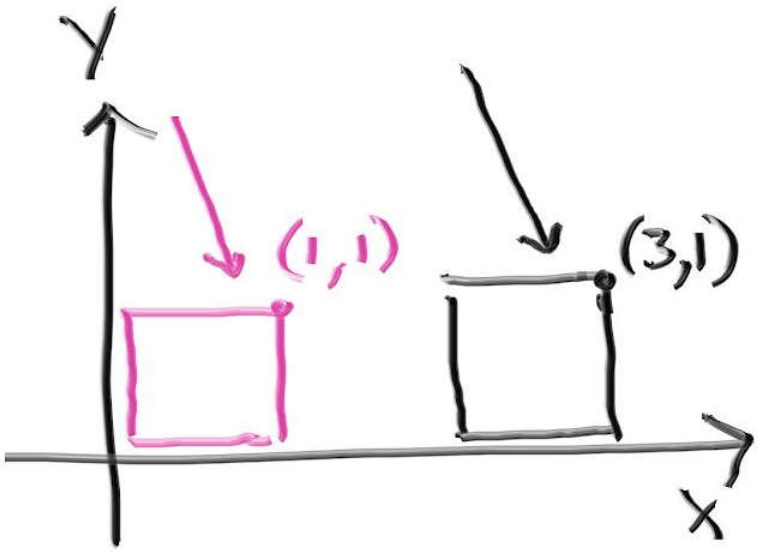


Figure 6: *Ray-box intersection with moved ray vs box*

8.1. Intance Translation

Whether you think of this as a move or a change of coordinates is up to you. The code for this, to move any underlying hittable is a *translate* instance.

```
class translate : public hittable {
    public:
        translate(shared_ptr<hittable> p, const vec3& displacement)
            : ptr(p), offset(displacement) {}

        virtual bool hit(const ray& r, double t_min, double t_max, hit_record& rec)
const;
        virtual bool bounding_box(double t0, double t1, aabb& output_box) const;

    public:
        shared_ptr<hittable> ptr;
        vec3 offset;
};

bool translate::hit(const ray& r, double t_min, double t_max, hit_record& rec) const
{
    ray moved_r(r.origin() - offset, r.direction(), r.time());
    if (!ptr->hit(moved_r, t_min, t_max, rec))
        return false;

    rec.p += offset;
    rec.set_face_normal(moved_r, rec.normal);

    return true;
}

bool translate::bounding_box(double t0, double t1, aabb& output_box) const {
    if (!ptr->bounding_box(t0, t1, output_box))
        return false;

    output_box = aabb(
        output_box.min() + offset,
        output_box.max() + offset);

    return true;
}
```

Listing 60: `[hittable.h]` *Hittable translation class*

8.2. Instance Rotation

Rotation isn't quite as easy to understand or generate the formulas for. A common graphics tactic is to apply all rotations about the x, y, and z axes. These rotations are in some sense axis-aligned. First, let's rotate by theta about the z-axis. That will be changing only x and y, and in ways that don't depend on z.

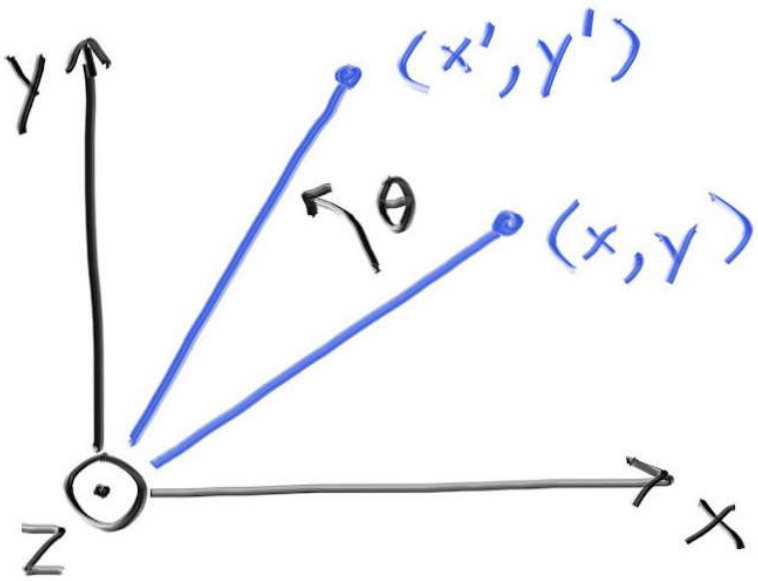


Figure 7: *Rotation about the Z axis*

This involves some basic trigonometry that uses formulas that I will not cover here. That gives you the correct impression it's a little involved, but it is straightforward, and you can find it in any graphics text and in many lecture notes. The result for rotating counter-clockwise about z is:

$$x' = \cos(\theta) \cdot x - \sin(\theta) \cdot y$$

$$y' = \sin(\theta) \cdot x + \cos(\theta) \cdot y$$

The great thing is that it works for any θ and doesn't need any cases for quadrants or anything like that. The inverse transform is the opposite geometric operation: rotate by $-\theta$. Here, recall that $\cos(\theta) = \cos(-\theta)$ and $\sin(-\theta) = -\sin(\theta)$, so the formulas are very simple.

Similarly, for rotating about y (as we want to do for the blocks in the box) the formulas are:

$$x' = \cos(\theta) \cdot x + \sin(\theta) \cdot z$$

$$z' = -\sin(\theta) \cdot x + \cos(\theta) \cdot z$$

And about the x-axis:

$$y' = \cos(\theta) \cdot y - \sin(\theta) \cdot z$$

$$z' = \sin(\theta) \cdot y + \cos(\theta) \cdot z$$

Unlike the situation with translations, the surface normal vector also changes, so we need to transform directions too if we get a hit. Fortunately for rotations, the same formulas apply. If you add scales, things get more complicated. See the web page https://in1weekend.blogspot.com/ for links to that.

For a y-rotation class we have:

```
class rotate_y : public hittable {
    public:
        rotate_y(shared_ptr<hittable> p, double angle);

        virtual bool hit(const ray& r, double t_min, double t_max, hit_record& rec)
const;
        virtual bool bounding_box(double t0, double t1, aabb& output_box) const {
            output_box = bbox;
            return hasbox;
        }

    public:
        shared_ptr<hittable> ptr;
        double sin_theta;
        double cos_theta;
        bool hasbox;
        aabb bbox;
};
```

Listing 61: `[hittable.h]` *Hittable rotate-Y class*

With constructor:

```
rotate_y::rotate_y(shared_ptr<hittable>, double angle) : ptr(p) {
    auto radians = degrees_to_radians(angle);
    sin_theta = sin(radians);
    cos_theta = cos(radians);
    hasbox = ptr->bounding_box(0, 1, bbox);

    point3 min( infinity,  infinity,  infinity);
    point3 max(-infinity, -infinity, -infinity);

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 2; k++) {
                auto x = i*bbox.max().x() + (1-i)*bbox.min().x();
                auto y = j*bbox.max().y() + (1-j)*bbox.min().y();
                auto z = k*bbox.max().z() + (1-k)*bbox.min().z();

                auto newx =  cos_theta*x + sin_theta*z;
                auto newz = -sin_theta*x + cos_theta*z;

                vec3 tester(newx, y, newz);

                for (int c = 0; c < 3; c++) {
                    min[c] = fmin(min[c], tester[c]);
                    max[c] = fmax(max[c], tester[c]);
                }
            }
        }
    }

    bbox = aabb(min, max);
}
```

Listing 62: `[hittable.h]` *Rotate-Y rotate method*

And the hit function:

```
bool rotate_y::hit(const ray& r, double t_min, double t_max, hit_record& rec) const
{
    auto origin = r.origin();
    auto direction = r.direction();

    origin[0] = cos_theta*r.origin()[0] - sin_theta*r.origin()[2];
    origin[2] = sin_theta*r.origin()[0] + cos_theta*r.origin()[2];

    direction[0] = cos_theta*r.direction()[0] - sin_theta*r.direction()[2];
    direction[2] = sin_theta*r.direction()[0] + cos_theta*r.direction()[2];

    ray rotated_r(origin, direction, r.time());

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

    auto p = rec.p;
    auto normal = rec.normal;

    p[0] =  cos_theta*rec.p[0] + sin_theta*rec.p[2];
    p[2] = -sin_theta*rec.p[0] + cos_theta*rec.p[2];

    normal[0] =  cos_theta*rec.normal[0] + sin_theta*rec.normal[2];
    normal[2] = -sin_theta*rec.normal[0] + cos_theta*rec.normal[2];

    rec.p = p;
    rec.set_face_normal(rotated_r, normal);

    return true;
}
```

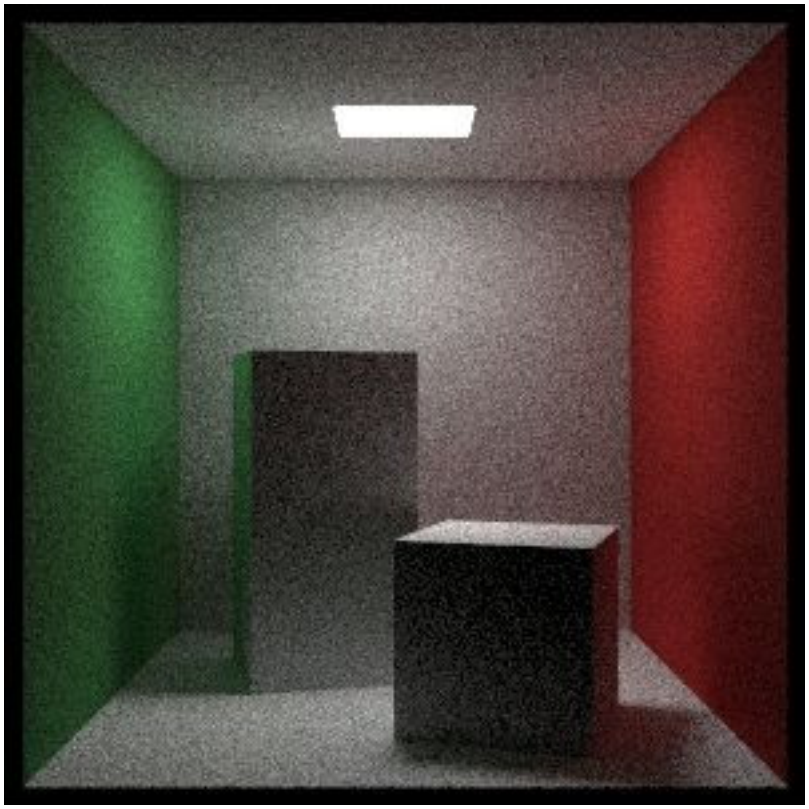
Listing 63: `[hittable.h]` *Hittable Y-rotate hit function*

And the changes to Cornell are:

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

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

Listing 64: `[main.cc]` *Cornell scene with Y-rotated boxes*

Which yields:



Standard Cornell box scene

9. Volumes

One thing it's nice to add to a ray tracer is smoke/fog/mist. These are sometimes called *volumes* or *participating media*. Another feature that is nice to add is subsurface scattering, which is sort of like dense fog inside an object. This usually adds software architectural mayhem because volumes are a different animal than surfaces, but a cute technique is to make a volume a random surface. A bunch of smoke can be replaced with a surface that probabilistically might or might not be there at every point in the volume. This will make more sense when you see the code.

9.1. Constant Density Mediums

First, let's start with a volume of constant density. A ray going through there can either scatter inside the volume, or it can make it all the way through like the middle ray in the figure. More thin transparent volumes, like a light fog, are more likely to have rays like the middle one. How far the ray has to travel through the volume also determines how likely it is for the ray to make it through.

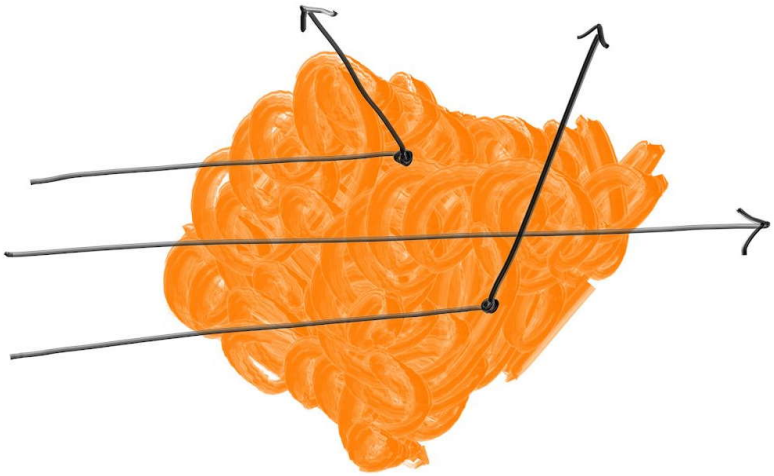


Figure 8: *Ray-volume interaction*

As the ray passes through the volume, it may scatter at any point. The denser the volume, the more likely that is. The probability that the ray scatters in any small distance ΔL is:

$$\text{probability} = C \cdot \Delta L$$

where C is proportional to the optical density of the volume. If you go through all the differential equations, for a random number you get a distance where the scattering occurs. If that distance is outside the volume, then there is no “hit”. For a constant volume we just need the density C and the boundary. I’ll use another hittable for the boundary. The resulting class is:

```
class constant_medium : public hittable {
    public:
        constant_medium(shared_ptr<hittable> b, double d, shared_ptr<texture> a)
            : boundary(b), neg_inv_density(-1/d)
        {
            phase_function = make_shared<isotropic>(a);
        }

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

        virtual bool bounding_box(double t0, double t1, aabb& output_box) const {
            return boundary->bounding_box(t0, t1, output_box);
        }

    public:
        shared_ptr<hittable> boundary;
        shared_ptr<material> phase_function;
        double neg_inv_density;
};
```

Listing 65: `[constant_medium.h]` *Constant medium class*

The scattering function of isotropic picks a uniform random direction:

```
class isotropic : public material {
    public:
        isotropic(shared_ptr<texture> a) : albedo(a) {}

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color& attenuation, ray&
scattered
        ) const  {
            scattered = ray(rec.p, random_in_unit_sphere(), r_in.time());
            attenuation = albedo->value(rec.u, rec.v, rec.p);
            return true;
        }

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

Listing 66: `[material.h]` *The isotropic class*

And the hit function is:

```
bool constant_medium::hit(const ray& r, double t_min, double t_max, hit_record& rec)
const {
    // Print occasional samples when debugging. To enable, set enableDebug true.
    const bool enableDebug = false;
    const bool debugging = enableDebug && random_double() < 0.00001;

    hit_record rec1, rec2;

    if (!boundary->hit(r, -infinity, infinity, rec1))
        return false;

    if (!boundary->hit(r, rec1.t+0.0001, infinity, rec2))
        return false;

    if (debugging) std::cerr << "\nt0=" << rec1.t << ", t1=" << rec2.t << '\n';

    if (rec1.t < t_min) rec1.t = t_min;
    if (rec2.t > t_max) rec2.t = t_max;

    if (rec1.t >= rec2.t)
        return false;

    if (rec1.t < 0)
        rec1.t = 0;

    const auto ray_length = r.direction().length();
    const auto distance_inside_boundary = (rec2.t - rec1.t) * ray_length;
    const auto hit_distance = neg_inv_density * log(random_double());

    if (hit_distance > distance_inside_boundary)
        return false;

    rec.t = rec1.t + hit_distance / ray_length;
    rec.p = r.at(rec.t);

    if (debugging) {
        std::cerr << "hit_distance = " <<  hit_distance << '\n'
                  << "rec.t = " <<  rec.t << '\n'
                  << "rec.p = " <<  rec.p << '\n';
    }

    rec.normal = vec3(1,0,0);  // arbitrary
    rec.front_face = true;     // also arbitrary
    rec.mat_ptr = phase_function;

    return true;
}
```

Listing 67: `[constant_medium.h]` *Constant medium hit method*

The reason we have to be so careful about the logic around the boundary is we need to make sure this works for ray origins inside the volume. In clouds, things bounce around a lot so that is a common case.

In addition, the above code assumes that once a ray exits the constant medium boundary, it will continue forever outside the boundary. Put another way, it assumes that the boundary shape is convex. So this particular implementation will work for boundaries like boxes or spheres, but will not work with toruses or shapes that contain voids. It's possible to write an implementation that handles arbitrary shapes, but we'll leave that as an exercise for the reader.

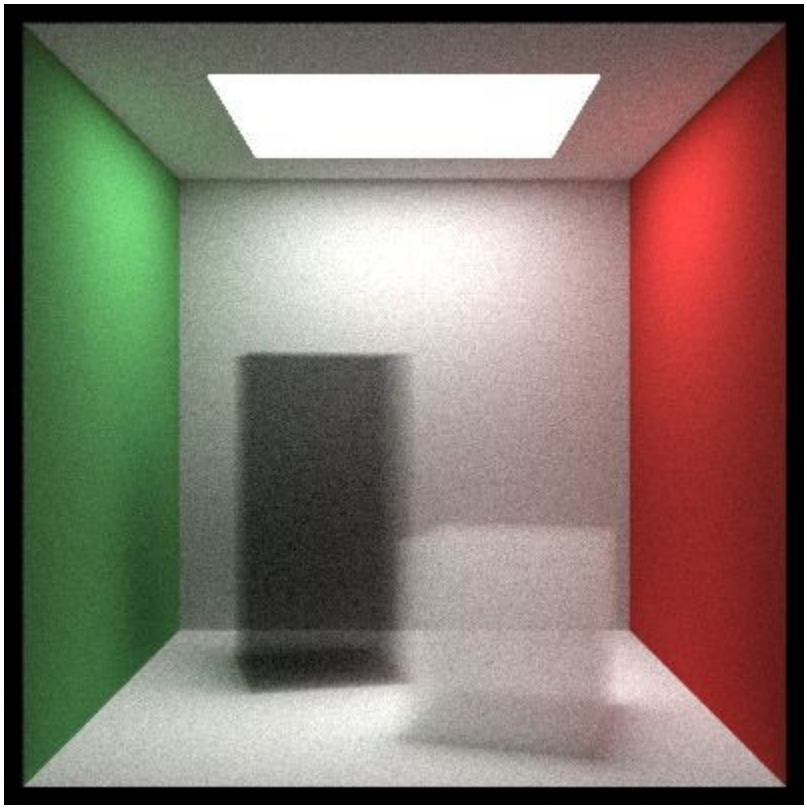
9.2. Rendering a Cornell Box with Smoke and Fog Boxes

If we replace the two blocks with smoke and fog (dark and light particles), and make the light bigger (and dimmer so it doesn't blow out the scene) for faster convergence:

```
hittable_list cornell_smoke() {
    hittable_list objects;

    auto red   = make_shared<lambertian>(make_shared<solid_color>(.65, .05, .05));
    auto white = make_shared<lambertian>(make_shared<solid_color>(.73, .73, .73));
    auto green = make_shared<lambertian>(make_shared<solid_color>(.12, .45, .15));
    auto light = make_shared<diffuse_light>(make_shared<solid_color>(7, 7, 7));

    objects.add(make_shared<flip_face>(make_shared<yz_rect>(0, 555, 0, 555, 555,
green)));
    objects.add(make_shared<yz_rect>(0, 555, 0, 555, 0, red));
    objects.add(make_shared<xz_rect>(113, 443, 127, 432, 554, light));
    objects.add(make_shared<flip_face>(make_shared<xz_rect>(0, 555, 0, 555, 555,
white)));
    objects.add(make_shared<xz_rect>(0, 555, 0, 555, 0, white));
    objects.add(make_shared<flip_face>(make_shared<xy_rect>(0, 555, 0, 555, 555,
white)));

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

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

    objects.add(make_shared<constant_medium>(box1, 0.01, make_shared<solid_color>
(0,0,0)));
    objects.add(make_shared<constant_medium>(box2, 0.01, make_shared<solid_color>
(1,1,1)));

    return objects;
}
```

Listing 68: `[main.cc]` *Cornell box, with smoke*

We get:



Cornell box with blocks of smoke

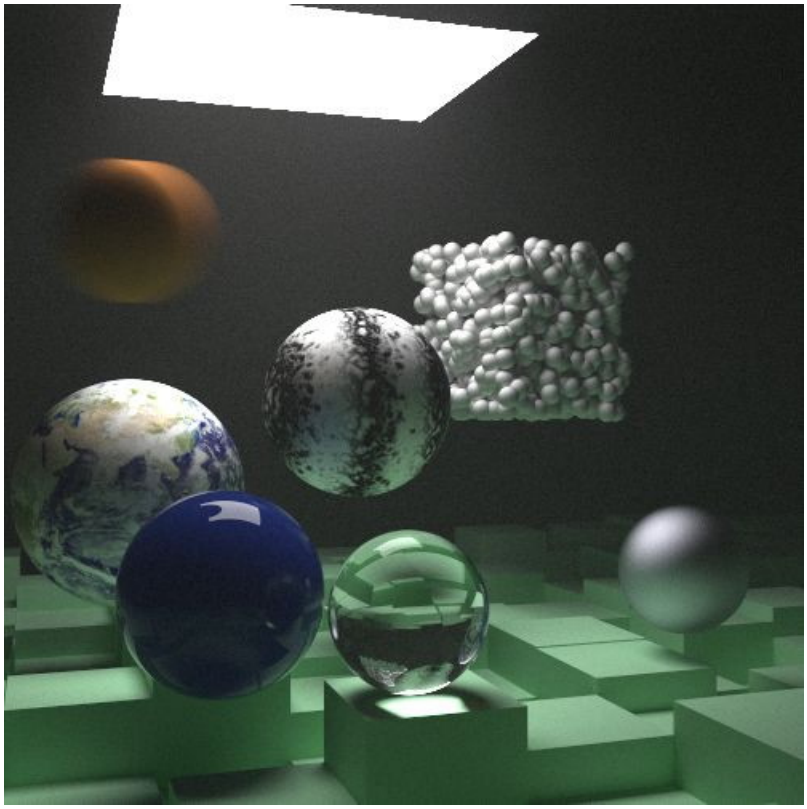
10. A Scene Testing All New Features

Let’s put it all together, with a big thin mist covering everything, and a blue subsurface reflection sphere (we didn’t implement that explicitly, but a volume inside a dielectric is what a subsurface material is). The biggest limitation left in the renderer is no shadow rays, but that is why we get caustics and subsurface for free. It’s a double-edged design decision.

```
hittable_list final_scene() {
    hittable_list boxes1;
    auto ground = make_shared<lambertian>(make_shared<solid_color>(0.48, 0.83,
0.53));

    const int boxes_per_side = 20;
    for (int i = 0; i < boxes_per_side; i++) {
        for (int j = 0; j < boxes_per_side; j++) {
            auto w = 100.0;
            auto x0 = -1000.0 + i*w;
            auto z0 = -1000.0 + j*w;
            auto y0 = 0.0;
            auto x1 = x0 + w;
            auto y1 = random_double(1,101);
            auto z1 = z0 + w;

            boxes1.add(make_shared<box>(point3(x0,y0,z0), point3(x1,y1,z1),
ground));
        }
    }

    hittable_list objects;

    objects.add(make_shared<bvh_node>(boxes1, 0, 1));

    auto light = make_shared<diffuse_light>(make_shared<solid_color>(7, 7, 7));
    objects.add(make_shared<xz_rect>(123, 423, 147, 412, 554, light));

    auto center1 = point3(400, 400, 200);
    auto center2 = center1 + vec3(30,0,0);
    auto moving_sphere_material =
        make_shared<lambertian>(make_shared<solid_color>(0.7, 0.3, 0.1));
    objects.add(make_shared<moving_sphere>(center1, center2, 0, 1, 50,
moving_sphere_material));

    objects.add(make_shared<sphere>(point3(260, 150, 45), 50,
make_shared<dielectric>(1.5)));
    objects.add(make_shared<sphere>(
        point3(0, 150, 145), 50, make_shared<metal>(color(0.8, 0.8, 0.9), 10.0)
    ));

    auto boundary = make_shared<sphere>(point3(360,150,145), 70,
make_shared<dielectric>(1.5));
    objects.add(boundary);
    objects.add(make_shared<constant_medium>(
        boundary, 0.2, make_shared<solid_color>(0.2, 0.4, 0.9)
    ));
    boundary = make_shared<sphere>(point3(0, 0, 0), 5000, make_shared<dielectric>
(1.5));
    objects.add(make_shared<constant_medium>(
        boundary, .0001, make_shared<solid_color>(1,1,1)));

    auto emat = make_shared<lambertian>(make_shared<image_texture>("earthmap.jpg"));
    objects.add(make_shared<sphere>(point3(400,200,400), 100, emat));
    auto pertext = make_shared<noise_texture>(0.1);
    objects.add(make_shared<sphere>(point3(220,280,300), 80, make_shared<lambertian>
(pertext)));

    hittable_list boxes2;
    auto white = make_shared<lambertian>(make_shared<solid_color>(.73, .73, .73));
    int ns = 1000;
    for (int j = 0; j < ns; j++) {
        boxes2.add(make_shared<sphere>(point3::random(0,165), 10, white));
    }

    objects.add(make_shared<translate>(
        make_shared<rotate_y>(
            make_shared<bvh_node>(boxes2, 0.0, 1.0), 15),
            vec3(-100,270,395)
        )
    );

    return objects;
}
```

Listing 69: [main.cc] *Final scene*

Running it with 10,000 rays per pixel yields:



Final scene

Now go off and make a really cool image of your own! See https://in1weekend.blogspot.com/ for pointers to further reading and features, and feel free to email questions, comments, and cool images to me at ptrshrl@gmail.com.

11. Acknowledgments

Original Manuscript Help

Dave Hart Jean Buckley

Web Release

Berna Kabadayı Lori Whippler Ronald Wotzlaw
Lorenzo Mancini Hollasch

Corrections and Improvements

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

Tools

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.

formatted by Markdeep 1.10