Ray Tracing: The Next Week

Peter Shirley, Trevor David Black, Steve Hollasch

Version 4.0.2, 2025-04-25

Contents

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. See our Further Reading wiki page for additional project related resources. 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!

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

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

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

2. Motion Blur

When you decided to ray trace, you decided that visual quality was worth more than run-time. When rendering fuzzy reflection and defocus blur, we used multiple samples per pixel. Once you have taken a step down that road, the good news is that almost *all* effects can be similarly brute-forced. Motion blur is certainly one of those.

In a real camera, the shutter remains open for a short time interval, during which the camera and objects in the world may move. To accurately reproduce such a camera shot, we seek an average of what the camera senses while its shutter is open to the world.

2.1. Introduction of SpaceTime Ray Tracing

We can get a random estimate of a single (simplified) photon by sending a single ray at some random instant in time while the shutter is open. As long as we can determine where the objects are supposed to be at that instant, we can get an accurate measure of the light for that ray at that same instant. This is yet another example of how random (Monte Carlo) ray tracing ends up being quite simple. Brute force wins again!

Since the “engine” of the ray tracer can just make sure the objects are where they need to be for each ray, the intersection guts don’t change much. To accomplish this, we need to store the exact time for each ray:

```
class ray {
  public:
    ray() {}

    ray(const point3& origin, const vec3& direction, double time)
      : orig(origin), dir(direction), tm(time) {}

    ray(const point3& origin, const vec3& direction)
      : ray(origin, direction, 0) {}

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

    double time() const { return tm; }

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

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

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

2.2. Managing Time

Before continuing, let's think about time, and how we might manage it across one or more successive renders. There are two aspects of shutter timing to think about: the time from one shutter opening to the next shutter opening, and how long the shutter stays open for each frame. Standard movie film used to be shot at 24 frames per second. Modern digital movies can be 24, 30, 48, 60, 120 or however many frames per second director wants.

Each frame can have its own shutter speed. This shutter speed need not be — and typically isn't — the maximum duration of the entire frame. You could have the shutter open for 1/1000th of a second every frame, or 1/60th of a second.

If you wanted to render a sequence of images, you would need to set up the camera with the appropriate shutter timings: frame-to-frame period, shutter/render duration, and the total number of frames (total shot time). If the camera is moving and the world is static, you're good to go. However, if anything in the world is moving, you would need to add a method to `hittable` so that every object could be made aware of the current frame's time period. This method would provide a way for all animated objects to set up their motion during that frame.

This is fairly straight-forward, and definitely a fun avenue for you to experiment with if you wish. However, for our purposes right now, we're going to proceed with a much simpler model. We will render only a single frame, implicitly assuming a start at time = 0 and ending at time = 1. Our first task is to modify the camera to launch rays with random times in $[0, 1]$, and our second task will be the creation of an animated sphere class.

2.3. Updating the Camera to Simulate Motion Blur

We need to modify the camera to generate rays at a random instant between the start time and the end time. Should the camera keep track of the time interval, or should that be up to the user of the 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 {
  ...
  private:
    ...
    ray get_ray(int i, int j) const {
        // Construct a camera ray originating from the defocus disk and directed at a randomly
        // sampled point around the pixel location i, j.

        auto offset = sample_square();
        auto pixel_sample = pixel00_loc
                          + ((i + offset.x()) * pixel_delta_u)
                          + ((j + offset.y()) * pixel_delta_v);

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

        return ray(ray_origin, ray_direction, ray_time);
    }

    ...
};
```

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

2.4. Adding Moving Spheres

Now to create a moving object. I'll update the sphere class so that its center moves linearly from `center1` at time=0 to `center2` at time=1. (It continues on indefinitely outside that time interval, so it really can be sampled at any time.) We'll do this by replacing the 3D center point with a 3D ray that describes the original position at time=0 and the displacement to the end position at time=1.

```
class sphere : public hittable {
  public:
    // Stationary Sphere
    sphere(const point3& static_center, double radius, shared_ptr<material> mat)
      : center(static_center, vec3(0,0,0)), radius(std::fmax(0,radius)), mat(mat) {}

    // Moving Sphere
    sphere(const point3& center1, const point3& center2, double radius,
           shared_ptr<material> mat)
      : center(center1, center2 - center1), radius(std::fmax(0,radius)), mat(mat) {}

    ...

  private:
    ray center;
    double radius;
    shared_ptr<material> mat;

};
#endif
```

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

The updated `sphere::hit()` function is almost identical to the old `sphere::hit()` function: we just need to now determine the current position of the animated center:

```
class sphere : public hittable {
  public:
    ...
    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {
        point3 current_center = center.at(r.time());
        vec3 oc = current_center - r.origin();
        auto a = r.direction().length_squared();
        auto h = dot(r.direction(), oc);
        auto c = oc.length_squared() - radius*radius;

        ...

        rec.t = root;
        rec.p = r.at(rec.t);
        vec3 outward_normal = (rec.p - current_center) / radius;
        rec.set_face_normal(r, outward_normal);
        get_sphere_uv(outward_normal, rec.u, rec.v);
        rec.mat = mat;

        return true;
    }
    ...
};
```

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

2.5. Tracking the Time of Ray Intersection

Now that rays have a time property, we need to update the `material::scatter()` methods to account for the time of intersection:

```
class lambertian : public material {
    ...
    bool scatter(const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered)
    const override {
        auto scatter_direction = rec.normal + random_unit_vector();

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

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

class metal : public material {
    ...
    bool scatter(const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered)
    const override {
        vec3 reflected = reflect(r_in.direction(), rec.normal);
        reflected = unit_vector(reflected) + (fuzz * random_unit_vector());
        scattered = ray(rec.p, reflected, r_in.time());
        attenuation = albedo;

        return (dot(scattered.direction(), rec.normal) > 0);
    }
    ...
};

class dielectric : public material {
    ...
    bool scatter(const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered)
    const override {
        ...
        scattered = ray(rec.p, direction, r_in.time());
        return true;
    }
    ...
};
```

Listing 5: `[material.h]` *Handle ray time in the material::scatter() methods*

2.6. 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. Each sphere moves from its center $\mathbf{C}$ at time $t = 0$ to $\mathbf{C} + (0, r/2, 0)$ at time $t = 1$:

```
int main() {
    hittable_list world;

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

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

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

                if (choose_mat < 0.8) {
                    // diffuse
                    auto albedo = color::random() * color::random();
                    sphere_material = make_shared<lambertian>(albedo);
                    auto center2 = center + vec3(0, random_double(0,.5), 0);
                    world.add(make_shared<sphere>(center, center2, 0.2, sphere_material));
                } else if (choose_mat < 0.95) {
                ...
    }
    ...

    camera cam;

    cam.aspect_ratio      = 16.0 / 9.0;
    cam.image_width       = 400;
    cam.samples_per_pixel = 100;
    cam.max_depth         = 50;

    cam.vfov     = 20;
    cam.lookfrom = point3(13,2,3);
    cam.lookat   = point3(0,0,0);
    cam.vup      = vec3(0,1,0);

    cam.defocus_angle = 0.6;
    cam.focus_dist    = 10.0;

    cam.render(world);
}
```

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

This gives the following result:

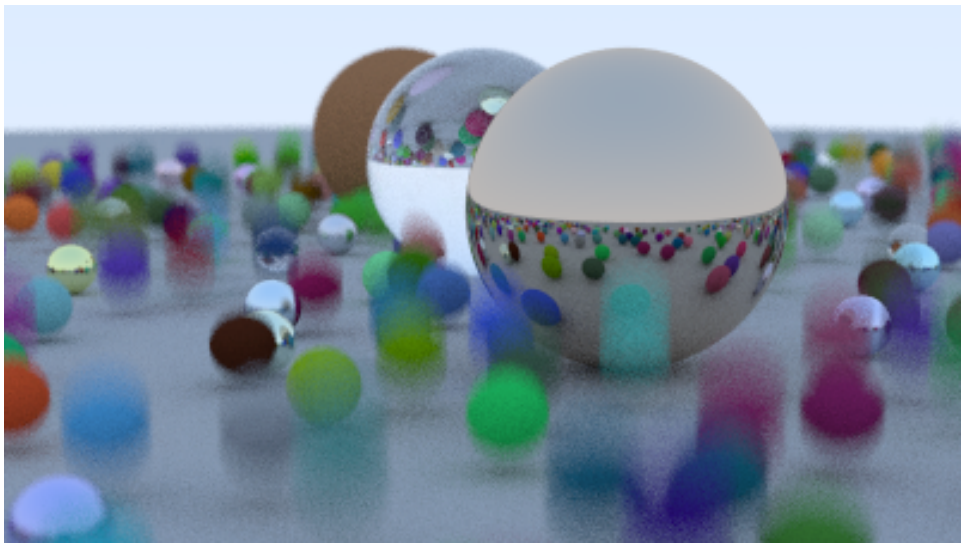

Image 1: *Bouncing spheres*

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.

Ray-object intersection is the main time-bottleneck in a ray tracer, and the run time is linear with the number of objects. But it's a repeated search on the same scene, 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 into the same scene, we can sort the objects in the scene, and then each ray intersection can be a sublinear search. The two most common methods of sorting are to 1) subdivide the space, and 2) subdivide 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 creating bounding volumes for a set of primitives is to find a volume that fully encloses (bounds) all the objects. For example, suppose you computed a sphere that bounds 10 objects. Any ray that misses the bounding sphere definitely misses all ten objects inside. 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
```

Note that we will use these bounding volumes to group the objects in the scene into subgroups. We are *not* dividing the screen or the scene space. We want any given object to be in just one bounding volume, though 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:

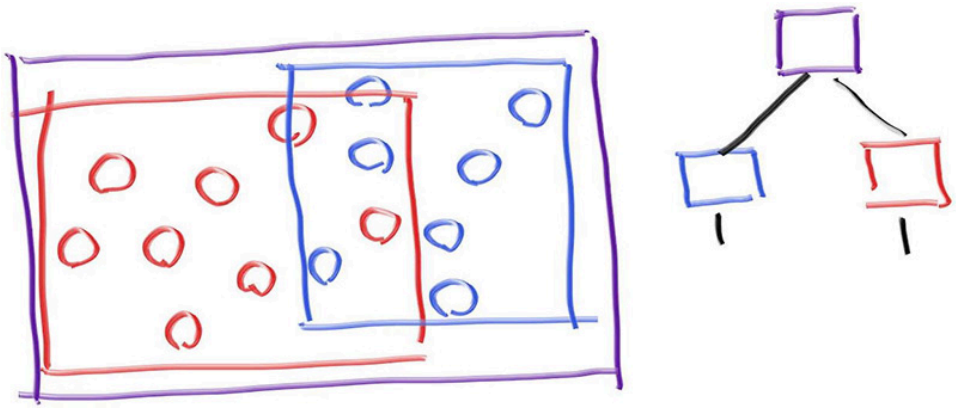

Figure 1: *Bounding volume hierarchy*

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 (such as the spherical bounds mentioned above), but this design choice is always something to keep in mind if you encounter other types of bounding models.

From now on we will call axis-aligned bounding rectangular parallelepipeds (really, that is what they need to be called if we're being precise) *axis-aligned bounding boxes*, or AABBs. (In the code, you will also come across the naming abbreviation “bbox” for “bounding box”.) 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 the stuff we need to display the object.

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”. Recall that an interval is just the points within two endpoints, for example, x such that $3 \leq x \leq 5$, or more succinctly x in $[3, 5]$. In 2D, an AABB (a rectangle) is defined by the overlap two intervals:

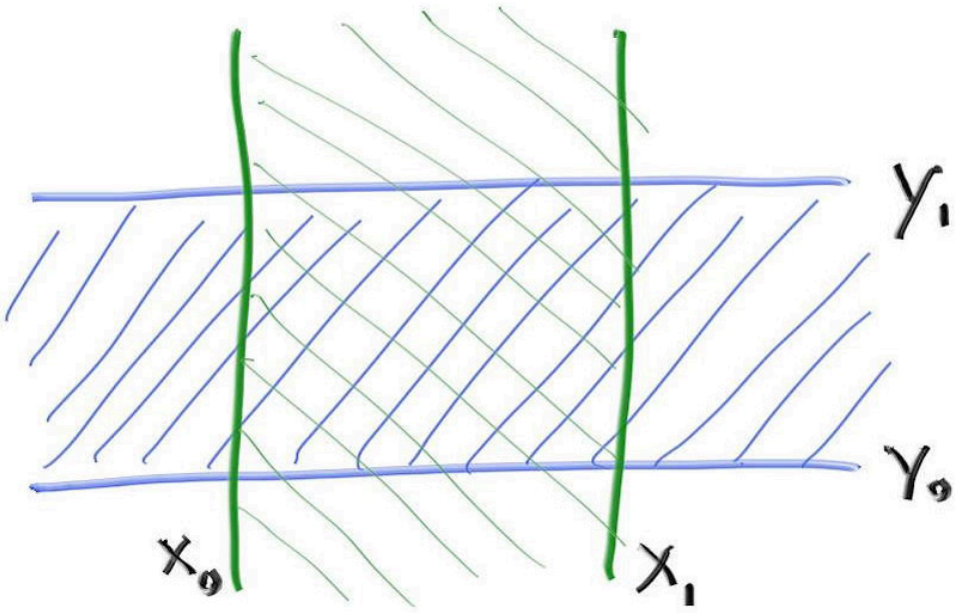

Figure 2: *2D axis-aligned bounding box*

To determine if a ray hits one interval, we first need to figure out whether the ray hits the boundaries. For example, in 1D, ray intersection with two planes will yield the ray parameters t_0 and t_1. (If the ray is parallel to the planes, its intersection with any plane will be undefined.)

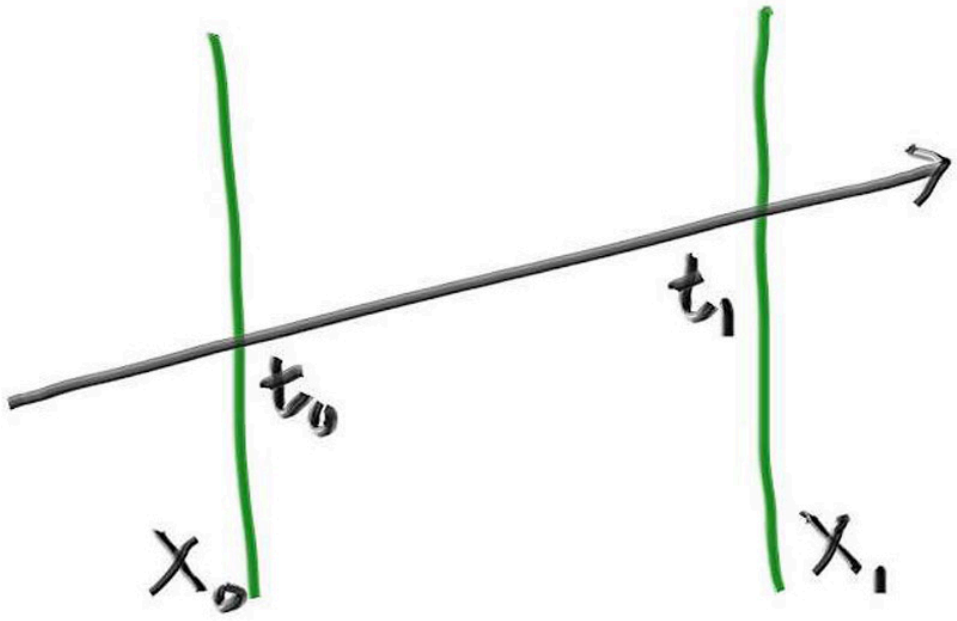

Figure 3: *Ray-slab intersection*

How do we find the intersection between a ray and a plane? Recall that the ray is just defined by a function that—given a parameter t—returns a location $\mathbf{P}(t)$:

$$\mathbf{P}(t) = \mathbf{Q} + t\mathbf{d}$$

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

$$x_0 = Q_x + t_0 d_x$$

So t at the intersection is given by

$$t_0 = \frac{x_0 - Q_x}{d_x}$$

We get the similar expression for x_1:

$$t_1 = \frac{x_1 - Q_x}{d_x}$$

The key observation to turn that 1D math into a 2D or 3D hit test is this: if a ray intersects the box bounded by all pairs of planes, then all t-intervals will overlap. For example, in 2D the green and blue overlapping only happens if the ray intersects the bounded box:

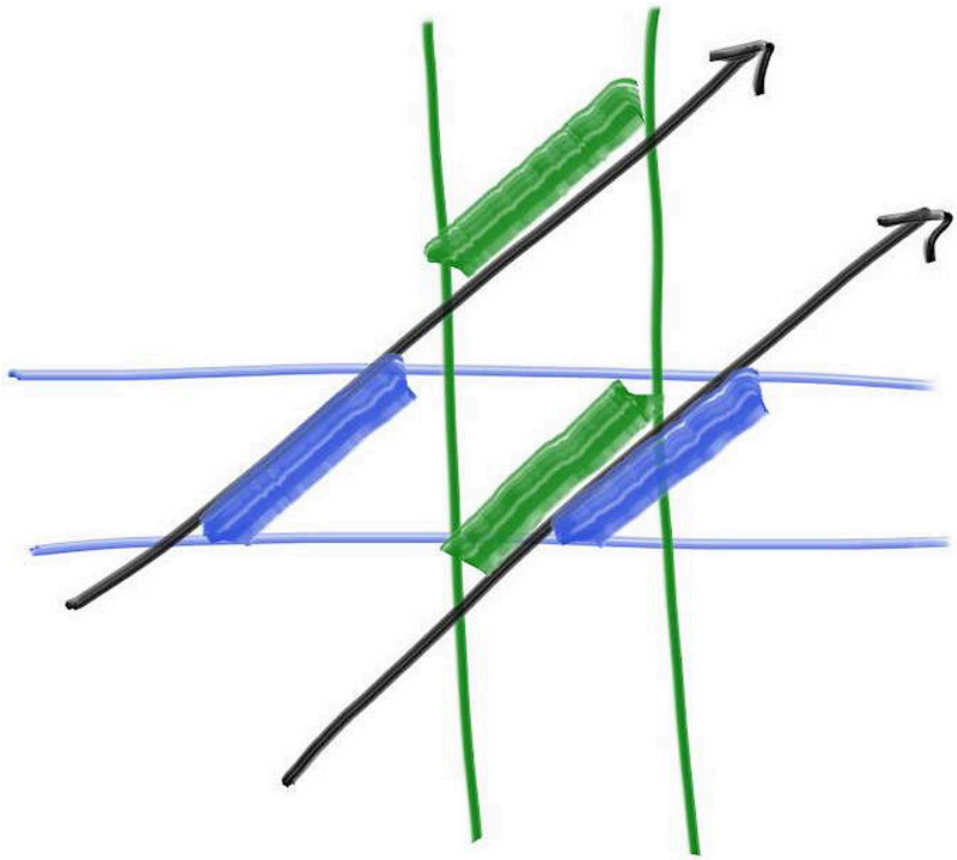

Figure 4: *Ray-slab t-interval overlap*

In this figure, the upper ray intervals do not overlap, so we know the ray does *not* hit the 2D box bounded by the green and blue planes. The lower ray intervals *do* overlap, so we know the lower ray *does* hit the bounded box.

3.4. Ray Intersection with an AABB

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

```
interval_x ← compute_intersection_x (ray, x0, x1)
interval_y ← compute_intersection_y (ray, y0, y1)
return overlaps(interval_x, interval_y)
```

That is awesomely simple, and the fact that the 3D version trivially extends the above is why people love the slab method:

```
interval_x ← compute_intersection_x (ray, x0, x1)
interval_y ← compute_intersection_y (ray, y0, y1)
interval_z ← compute_intersection_z (ray, z0, z1)
return overlaps(interval_x, interval_y, interval_z)
```

There are some caveats that make this less pretty than it first appears. Consider again the 1D equations for t_0 and t_1:

$$t_0 = \frac{x_0 - Q_x}{d_x}$$

$$t_1 = \frac{x_1 - Q_x}{d_x}$$

First, suppose the ray is traveling in the negative $\mathbf{x}$ direction. The interval (t_{x0}, t_{x1}) as computed above might be reversed, like $(7, 3)$ for example. Second, the denominator d_x could be zero, yielding infinite values. And if the ray origin lies on one of the slab boundaries, we can get a `NaN`, since both the numerator and the denominator can be zero. Also, the zero will have a ± sign when using IEEE floating point.

The good news for $d_x = 0$ is that t_{x0} and t_{x1} will be equal: both +∞ or -∞, if not between x_0 and x_1. So, using min and max should get us the right answers:

$$t_{x0} = \min(\frac{x_0 - Q_x}{d_x}, \frac{x_1 - Q_x}{d_x})$$

$$t_{x1} = \max(\frac{x_0 - Q_x}{d_x}, \frac{x_1 - Q_x}{d_x})$$

The remaining troublesome case if we do that is if $d_x = 0$ and either $x_0 - Q_x = 0$ or $x_1 - Q_x = 0$ so we get a `NaN`. In that case we can arbitrarily interpret that as either hit or no hit, but we'll revisit that later.

Now, let's look at the pseudo-function `overlaps`. Suppose we can assume the intervals are not reversed, and we want to return true when the intervals overlap. The boolean `overlaps()` function computes the overlap of the t intervals `t_interval1` and `t_interval2`, and uses that to determine if that overlap is non-empty:

```
bool overlaps(t_interval1, t_interval2)
    t_min ← max(t_interval1.min, t_interval2.min)
    t_max ← min(t_interval1.max, t_interval2.max)
    return t_min < t_max
```

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).

To accomplish this, we'll first add a new `interval` function `expand`, which pads an interval by a given amount:

```
class interval {
  public:
    ...
    double clamp(double x) const {
        if (x < min) return min;
        if (x > max) return max;
        return x;
    }

    interval expand(double delta) const {
        auto padding = delta/2;
        return interval(min - padding, max + padding);
    }

    static const interval empty, universe;
};
```

Listing 7: `[interval.h]` *interval::expand() method*

Now we have everything we need to implement the new AABB class.

```
#ifndef AABB_H
#define AABB_H

class aabb {
  public:
    interval x, y, z;

    aabb() {} // The default AABB is empty, since intervals are empty by default.

    aabb(const interval& x, const interval& y, const interval& z)
      : x(x), y(y), z(z) {}

    aabb(const point3& a, const point3& b) {
        // Treat the two points a and b as extrema for the bounding box, so we don't require a
        // particular minimum/maximum coordinate order.

        x = (a[0] <= b[0]) ? interval(a[0], b[0]) : interval(b[0], a[0]);
        y = (a[1] <= b[1]) ? interval(a[1], b[1]) : interval(b[1], a[1]);
        z = (a[2] <= b[2]) ? interval(a[2], b[2]) : interval(b[2], a[2]);
    }

    const interval& axis_interval(int n) const {
        if (n == 1) return y;
        if (n == 2) return z;
        return x;
    }

    bool hit(const ray& r, interval ray_t) const {
        const point3& ray_orig = r.origin();
        const vec3&   ray_dir  = r.direction();

        for (int axis = 0; axis < 3; axis++) {
            const interval& ax = axis_interval(axis);
            const double adinv = 1.0 / ray_dir[axis];

            auto t0 = (ax.min - ray_orig[axis]) * adinv;
            auto t1 = (ax.max - ray_orig[axis]) * adinv;

            if (t0 < t1) {
                if (t0 > ray_t.min) ray_t.min = t0;
                if (t1 < ray_t.max) ray_t.max = t1;
            } else {
                if (t1 > ray_t.min) ray_t.min = t1;
                if (t0 < ray_t.max) ray_t.max = t0;
            }

            if (ray_t.max <= ray_t.min)
                return false;
        }
        return true;
    }
};

#endif
```

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

3.5. 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.

Recall that `interval` values constructed without arguments will be empty by default. Since an `aabb` object has an interval for each of its three dimensions, each of these will then be empty by default, and therefore `aabb` objects will be empty by

default. Thus, some objects may have empty bounding volumes. For example, consider a `hittable_list` object with no children. Happily, the way we've designed our interval class, the math all works out.

Finally, recall that some objects may be animated. Such objects should return their bounds over the entire range of motion, from time=0 to time=1.

```
#include "aabb.h"

class material;

...

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

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

    virtual aabb bounding_box() const = 0;
};
```

Listing 9: `[hittable.h]` *Hittable class with bounding box*

For a stationary sphere, the `bounding_box` function is easy:

```
class sphere : public hittable {
  public:
    // Stationary Sphere
    sphere(const point3& static_center, double radius, shared_ptr<material> mat)
      : center(static_center, vec3(0,0,0)), radius(std::fmax(0,radius)), mat(mat)
    {
        auto rvec = vec3(radius, radius, radius);
        bbox = aabb(static_center - rvec, static_center + rvec);
    }

    ...

    aabb bounding_box() const override { return bbox; }

  private:
    ray center;
    double radius;
    shared_ptr<material> mat;
    aabb bbox;

    ...
};
```

Listing 10: `[sphere.h]` *Sphere with bounding box*

For a moving sphere, we want the bounds of its entire range of motion. To do this, we can take the box of the sphere at time=0, and the box of the sphere at time=1, and compute the box around those two boxes.

```
class sphere : public hittable {
  public:
    ...

    // Moving Sphere
    sphere(const point3& center1, const point3& center2, double radius,
           shared_ptr<material> mat)
      : center(center1, center2 - center1), radius(std::fmax(0,radius)), mat(mat)
    {
        auto rvec = vec3(radius, radius, radius);
        aabb box1(center.at(0) - rvec, center.at(0) + rvec);
        aabb box2(center.at(1) - rvec, center.at(1) + rvec);
        bbox = aabb(box1, box2);
    }

    ...
};
```

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

Now we need a new `aabb` constructor that takes two boxes as input. First, we'll add a new interval constructor to do this:

```
class interval {
  public:
    double min, max;

    interval() : min(+infinity), max(-infinity) {} // Default interval is empty

    interval(double _min, double _max) : min(_min), max(_max) {}

    interval(const interval& a, const interval& b) {
        // Create the interval tightly enclosing the two input intervals.
        min = a.min <= b.min ? a.min : b.min;
        max = a.max >= b.max ? a.max : b.max;
    }

    double size() const {
    ...
```

Listing 12: `[interval.h]` *Interval constructor from two intervals*

Now we can use this to construct an axis-aligned bounding box from two input boxes.

```
class aabb {
  public:
    ...

    aabb(const point3& a, const point3& b) {
        ...
    }

    aabb(const aabb& box0, const aabb& box1) {
        x = interval(box0.x, box1.x);
        y = interval(box0.y, box1.y);
        z = interval(box0.z, box1.z);
    }

    ...
};
```

Listing 13: `[aabb.h]` *AABB constructor from two AABB inputs*

3.6. Creating Bounding Boxes of Lists of Objects

Now we'll update the `hittable_list` object, computing the bounds of its children. We'll update the bounding box incrementally as each new child is added.

```
#include "aabb.h"
#include "hittable.h"

#include <vector>

class hittable_list : public hittable {
  public:
    std::vector<shared_ptr<hittable>> objects;

    ...
    void add(shared_ptr<hittable> object) {
        objects.push_back(object);
        bbox = aabb(bbox, object->bounding_box());
    }

    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {
        ...
    }

    aabb bounding_box() const override { return bbox; }

  private:
    aabb bbox;
};
```

Listing 14: `[hittable_list.h]` *Hittable list with bounding box*

3.7. 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. 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.

I am a fan of the one class design when feasible. Here is such a class:

```
#ifndef BVH_H
#define BVH_H

#include "aabb.h"
#include "hittable.h"
#include "hittable_list.h"

class bvh_node : public hittable {
  public:
    bvh_node(hittable_list list) : bvh_node(list.objects, 0, list.objects.size()) {
        // There's a C++ subtlety here. This constructor (without span indices) creates an
        // implicit copy of the hittable list, which we will modify. The lifetime of the copied
        // list only extends until this constructor exits. That's OK, because we only need to
        // persist the resulting bounding volume hierarchy.
    }

    bvh_node(std::vector<shared_ptr<hittable>>& objects, size_t start, size_t end) {
        // To be implemented later.
    }

    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {
        if (!bbox.hit(r, ray_t))
            return false;

        bool hit_left = left->hit(r, ray_t, rec);
        bool hit_right = right->hit(r, interval(ray_t.min, hit_left ? rec.t : ray_t.max), rec);

        return hit_left || hit_right;
    }

    aabb bounding_box() const override { return bbox; }

  private:
    shared_ptr<hittable> left;
    shared_ptr<hittable> right;
    aabb bbox;
};

#endif
```

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

3.8. 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. The following code uses three methods—`box_x_compare`, `box_y_compare`, and `box_z_compare`—that we haven't yet defined.

```
#include "aabb.h"
#include "hittable.h"
#include "hittable_list.h"

#include <algorithm>

class bvh_node : public hittable {
  public:
    ...

    bvh_node(std::vector<shared_ptr<hittable>>& objects, size_t start, size_t end) {
        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) {
            left = objects[start];
            right = objects[start+1];
        } else {
            std::sort(std::begin(objects) + start, std::begin(objects) + end, comparator);

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

        bbox = aabb(left->bounding_box(), right->bounding_box());
    }

    ...
};
```

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

This uses a new function: `random_int()`:

```
...

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

inline int random_int(int min, int max) {
    // Returns a random integer in [min,max].
    return int(random_double(min, max+1));
}

...
```

Listing 17: `[rtweekend.h]` *A function to return random integers in a range*

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.

3.9. The Box Comparison Functions

Now we need to implement the box comparison functions, used by `std::sort()`. To do this, create a generic comparator that 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.

```
class bvh_node : public hittable {
  ...

  private:
    shared_ptr<hittable> left;
    shared_ptr<hittable> right;
    aabb bbox;

    static bool box_compare(
        const shared_ptr<hittable> a, const shared_ptr<hittable> b, int axis_index
    ) {
        auto a_axis_interval = a->bounding_box().axis_interval(axis_index);
        auto b_axis_interval = b->bounding_box().axis_interval(axis_index);
        return a_axis_interval.min < b_axis_interval.min;
    }

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

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

    static 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*

At this point, we're ready to use our new BVH code. Let's use it on our random spheres scene.

```
#include "rtweekend.h"

#include "bvh.h"
#include "camera.h"
#include "hittable.h"
#include "hittable_list.h"
#include "material.h"
#include "sphere.h"

int main() {
    ...

    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));

    world = hittable_list(make_shared<bvh_node>(world));

    camera cam;

    ...
}
```

Listing 19: `[main.cc]` *Random spheres, using BVH*

The rendered image should be identical to the non-BVH version shown in image 1. However, if you time the two versions, the BVH version should be faster. I see a speedup of almost *six and a half times* the prior version.

3.10. Another BVH Optimization

We can speed up the BVH optimization a bit more. Instead of choosing a random splitting axis, let's split the longest axis of the enclosing bounding box to get the most subdivision. The change is straight-forward, but we'll add a few things to the `aabb` class in the process.

The first task is to construct an axis-aligned bounding box of the span of objects in the BVH constructor. Basically, we'll construct the bounding box of the `bvh_node` from this span by initializing the bounding box to empty (we'll define `aabb::empty` shortly), and then augment it with each bounding box in the span of objects.

Once we have the bounding box, set the splitting axis to the one with the longest side. We'll imagine a function that does that for us: `aabb::longest_axis()`. Finally, since we're computing the bounding box of the object span up front, we can delete the original line that computed it as the union of the left and right sides.

```
class bvh_node : public hittable {
  public:
    ...
    bvh_node(std::vector<shared_ptr<hittable>>& objects, size_t start, size_t end) {
        // Build the bounding box of the span of source objects.
        bbox = aabb::empty;
        for (size_t object_index=start; object_index < end; object_index++)
            bbox = aabb(bbox, objects[object_index]->bounding_box());

        int axis = bbox.longest_axis();

        auto comparator = (axis == 0) ? box_x_compare
                        : (axis == 1) ? box_y_compare
                                      : box_z_compare;

        ...

        bbox = aabb(left->bounding_box(), right->bounding_box());
    }

    ...
```

Listing 20: `[bvh.h]` *Building the bbox for the span of BVH objects*

Now to implement the empty `aabb` code and the new `aabb::longest_axis()` function:

```
class aabb {
  public:
    ...

    bool hit(const ray& r, interval ray_t) const {
        ...
    }

    int longest_axis() const {
        // Returns the index of the longest axis of the bounding box.

        if (x.size() > y.size())
            return x.size() > z.size() ? 0 : 2;
        else
            return y.size() > z.size() ? 1 : 2;
    }

    static const aabb empty, universe;
};

const aabb aabb::empty    = aabb(interval::empty,    interval::empty,    interval::empty);
const aabb aabb::universe = aabb(interval::universe, interval::universe, interval::universe);
```

Listing 21: `[aabb.h]` *New aabb constants and longest_axis() function*

As before, you should see identical results to image 1, but rendering a little bit faster. On my system, this yields something like an additional 18% render speedup. Not bad for a little extra work.

4. Texture Mapping

Texture mapping in computer graphics is the process of applying a material effect to an object in the scene. The "texture" part is the effect, and the "mapping" part is in the mathematical sense of mapping one space onto another. This effect could be any material property: color, shininess, bump geometry (called *Bump Mapping*), or even material existence (to create cut-out regions of the surface).

The most common type of texture mapping maps an image onto the surface of an object, defining the color at each point on the object's surface. In practice, we implement the process in reverse: given some point on the object, we'll look up the color defined by the texture map.

To begin with, we'll make the texture colors procedural, and will create a texture map of constant color. 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 it's great being able to make any color a texture.

In order to perform the texture lookup, we need a *texture coordinate*. This coordinate can be defined in many ways, and we'll develop this idea as we progress. For now, we'll pass in two dimensional texture coordinates. By convention, texture coordinates are named u and v. For a constant texture, every (u, v) pair yields a constant color, so we can actually ignore the coordinates completely. However, other texture types will need these coordinates, so we keep these in the method interface.

The primary method of our texture classes is the `color value(...)` method, which returns the texture color given the input coordinates. In addition to taking the point's texture coordinates u and v, we also provide the position of the point in question, for reasons that will become apparent later.

4.1. Constant Color Texture

```
#ifndef TEXTURE_H
#define TEXTURE_H

class texture {
  public:
    virtual ~texture() = default;

    virtual color value(double u, double v, const point3& p) const = 0;
};

class solid_color : public texture {
  public:
    solid_color(const color& albedo) : albedo(albedo) {}

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

    color value(double u, double v, const point3& p) const override {
        return albedo;
    }

  private:
    color albedo;
};

#endif
```

Listing 22: `[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.

```
class hit_record {
  public:
    vec3 p;
    vec3 normal;
    shared_ptr<material> mat;
    double t;
    double u;
    double v;
    bool front_face;

    ...
```

Listing 23: `[hittable.h]` *Adding u, v coordinates to the* `hit_record`

In the future, we'll need to compute (u, v) texture coordinates for a given point on each type of `hittable`. More on that later.

4.2. Solid Textures: A Checker Texture

A solid (or spatial) texture depends only on the position of each point in 3D space. You can think of a solid texture as if it's coloring all of the points in space itself, instead of coloring a given object in that space. For this reason, the object can move through the colors of the texture as it changes position, though usually you would to fix the relationship between the object and the solid texture.

To explore spatial textures, we'll implement a spatial `checker_texture` class, which implements a three-dimensional checker pattern. Since a spatial texture function is driven by a given position in space, the texture `value()` function ignores the `u` and `v` parameters, and uses only the `p` parameter.

To accomplish the checkered pattern, we'll first compute the floor of each component of the input point. We could truncate the coordinates, but that would pull values toward zero, which would give us the same color on both sides of zero. The floor function will always shift values to the integer value on the left (toward negative infinity). Given these

three integer results ($\lfloor x \rfloor, \lfloor y \rfloor, \lfloor z \rfloor$) we take their sum and compute the result modulo two, which gives us either 0 or 1. Zero maps to the even color, and one to the odd color.

Finally, we add a scaling factor to the texture, to allow us to control the size of the checker pattern in the scene.

```
class checker_texture : public texture {
  public:
    checker_texture(double scale, shared_ptr<texture> even, shared_ptr<texture> odd)
      : inv_scale(1.0 / scale), even(even), odd(odd) {}

    checker_texture(double scale, const color& c1, const color& c2)
      : checker_texture(scale, make_shared<solid_color>(c1), make_shared<solid_color>(c2)) {}

    color value(double u, double v, const point3& p) const override {
        auto xInteger = int(std::floor(inv_scale * p.x()));
        auto yInteger = int(std::floor(inv_scale * p.y()));
        auto zInteger = int(std::floor(inv_scale * p.z()));

        bool isEven = (xInteger + yInteger + zInteger) % 2 == 0;

        return isEven ? even->value(u, v, p) : odd->value(u, v, p);
    }

  private:
    double inv_scale;
    shared_ptr<texture> even;
    shared_ptr<texture> odd;
};
```

Listing 24: `[texture.h]` *Checkered texture*

Those checker odd/even parameters can point 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.

To support procedural textures, we'll extend the `lambertian` class to work with textures instead of colors:

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

...

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

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

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

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

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

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

If we add this to our main scene:

```
#include "rtweekend.h"

#include "bvh.h"
#include "camera.h"
#include "hittable.h"
#include "hittable_list.h"
#include "material.h"
#include "sphere.h"
#include "texture.h"


int main() {
    hittable_list world;

    auto checker = make_shared<checker_texture>(0.32, color(.2, .3, .1), color(.9, .9, .9));
    world.add(make_shared<sphere>(point3(0,-1000,0), 1000, make_shared<lambertian>(checker)));

    for (int a = -11; a < 11; a++) {
    ...
}
```

Listing 26: `[main.cc]` *Checkered texture in use*

we get:

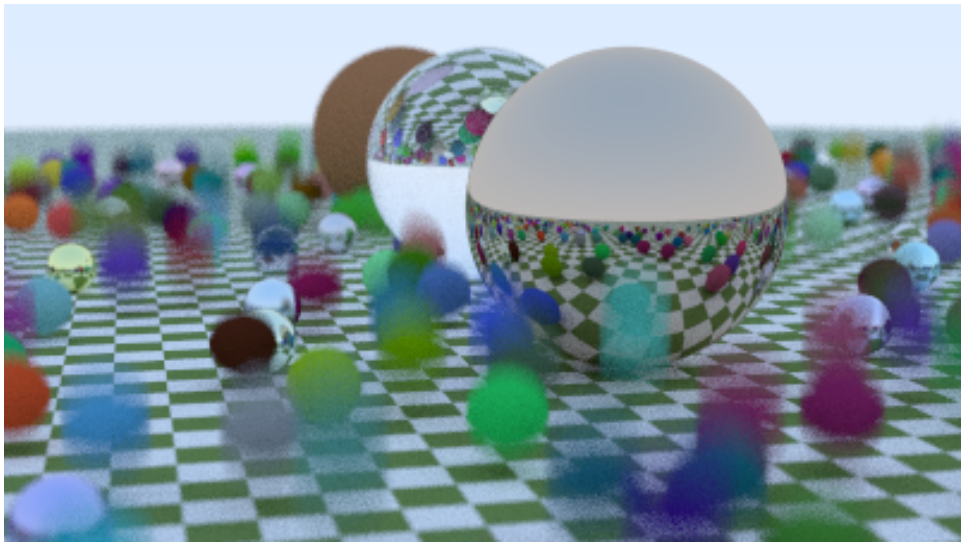

Image 2: *Spheres on checkered ground*

4.3. Rendering The Solid Checker Texture

We're going to add a second scene to our program, and will add more scenes after that as we progress through this book. To help with this, we'll set up a switch statement to select the desired scene for a given run. It's a crude approach, but we're trying to keep things dead simple and focus on the raytracing. You may want to use a different approach in your own raytracer, such as supporting command-line arguments.

Here's what our main.cc looks like after refactoring for our single random spheres scene. Rename `main()` to `bouncing_spheres()`, and add a new `main()` function to call it:

```
#include "rtweekend.h"

#include "bvh.h"
#include "camera.h"
#include "hittable.h"
#include "hittable_list.h"
#include "material.h"
#include "sphere.h"
#include "texture.h"

int main() {
void bouncing_spheres() {
    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));

    ...

    cam.render(world);
}

int main() {
    bouncing_spheres();
}
```

Listing 27: `[main.cc]` *Main dispatching to selected scene*

Now add a scene with two checkered spheres, one atop the other.

```
#include "rtweekend.h"

#include "bvh.h"
#include "camera.h"
#include "hittable.h"
#include "hittable_list.h"
#include "material.h"
#include "sphere.h"
#include "texture.h"


void bouncing_spheres() {
    ...
}

void checkered_spheres() {
    hittable_list world;

    auto checker = make_shared<checker_texture>(0.32, color(.2, .3, .1), color(.9, .9, .9));

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

    camera cam;

    cam.aspect_ratio      = 16.0 / 9.0;
    cam.image_width       = 400;
    cam.samples_per_pixel = 100;
    cam.max_depth         = 50;

    cam.vfov     = 20;
    cam.lookfrom = point3(13,2,3);
    cam.lookat   = point3(0,0,0);
    cam.vup      = vec3(0,1,0);

    cam.defocus_angle = 0;

    cam.render(world);
}

int main() {
    switch (2) {
        case 1: bouncing_spheres();  break;
        case 2: checkered_spheres(); break;
    }
}
```

Listing 28: [main.cc] *Two checkered spheres*

We get this result:

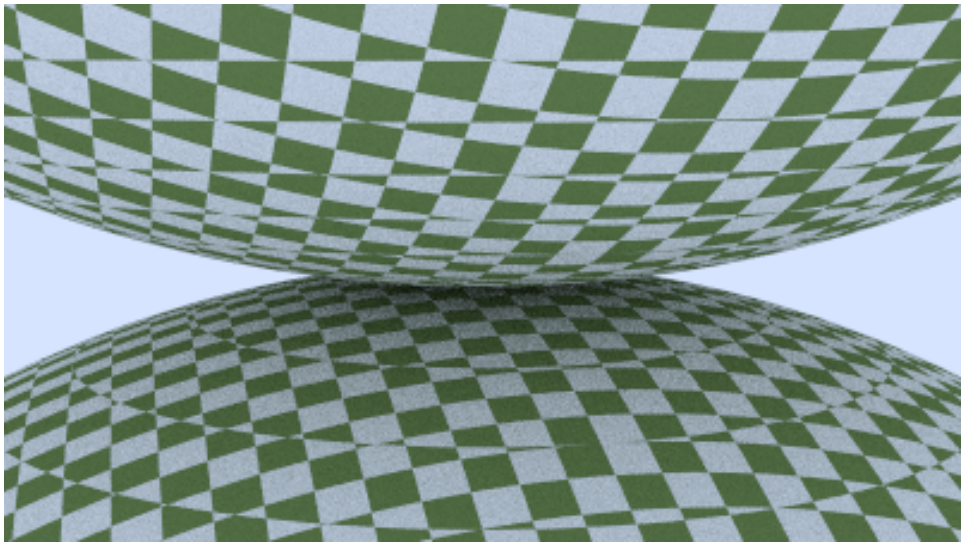

Image 3: *Checkered spheres*

You may think the result looks a bit odd. Since `checker_texture` is a spatial texture, we're really looking at the surface of the sphere cutting through the three-dimensional checker space. There are many situations where this is perfect, or at least sufficient. In many other situations, we really want to get a consistent effect on the surface of our objects. That approach is covered next.

4.4. Texture Coordinates for Spheres

Constant-color textures use no coordinates. Solid (or spatial) textures use the coordinates of a point in space. Now it's time to make use of the u, v texture coordinates. These coordinates specify the location on 2D source image (or in some 2D parameterized space). To get this, we need a way to find the u, v coordinates of any point on the surface of a 3D object. This mapping is completely arbitrary, but generally you'd like to cover the entire surface, and be able to scale, orient and stretch the 2D image in a way that makes some sense. We'll start with deriving a scheme to get the u, v coordinates of a sphere.

For spheres, texture coordinates are usually based on some form of longitude and latitude, *i.e.*, spherical coordinates. So we compute (θ, ϕ) in spherical coordinates, where θ is the angle up from the bottom pole (that is, up from -Y), and ϕ is the angle around the Y-axis (from -X to +Z to +X to -Z back to -X).

We want to map θ and ϕ to texture coordinates u and v each in $[0, 1]$, where $(u = 0, v = 0)$ maps to the bottom-left corner of the texture. Thus the normalization from (θ, ϕ) to (u, v) would be:

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

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

To compute θ and ϕ for a given point on the unit sphere centered at the origin, we start with the equations for the corresponding Cartesian coordinates:

$$\begin{aligned} y &= -\cos(\theta) \\ x &= -\cos(\phi)\sin(\theta) \\ z &= \sin(\phi)\sin(\theta) \end{aligned}$$

We need to invert these equations to solve for θ and ϕ. Because of the lovely `<cmath>` function `std::atan2()`, which takes any pair of numbers proportional to sine and cosine and returns the angle, we can pass in x and z (the $\sin(\theta)$ cancel) to solve for ϕ:

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

`std::atan2()` returns values in the range $-\pi$ to π, but they go from 0 to π, then flip to $-\pi$ and proceed back to zero. While this is mathematically correct, we want u to range from 0 to 1, not from 0 to $1/2$ and then from $-1/2$ to 0. Fortunately,

$$\text{atan2}(a, b) = \text{atan2}(-a, -b) + \pi,$$

and the second formulation yields values from 0 continuously to 2π. Thus, we can compute ϕ as

$$\phi = \text{atan2}(-z, x) + \pi$$

The derivation for θ is more straightforward:

$$\theta = \arccos(-y)$$

So for a sphere, the (u, v) coord computation is accomplished by a utility function that takes points on the unit sphere centered at the origin, and computes u and v:

```
class sphere : public hittable {
  ...
  private:
    ...

    static void get_sphere_uv(const point3& p, double& u, double& v) {
        // p: a given point on the sphere of radius one, centered at the origin.
        // u: returned value [0,1] of angle around the Y axis from X=-1.
        // v: returned value [0,1] of angle from Y=-1 to Y=+1.
        //     <1 0 0> yields <0.50 0.50>       <-1  0  0> yields <0.00 0.50>
        //     <0 1 0> yields <0.50 1.00>       < 0 -1  0> yields <0.50 0.00>
        //     <0 0 1> yields <0.25 0.50>       < 0  0 -1> yields <0.75 0.50>

        auto theta = std::acos(-p.y());
        auto phi = std::atan2(-p.z(), p.x()) + pi;

        u = phi / (2*pi);
        v = theta / pi;
    }
};
```

Listing 29: `[sphere.h]` *get_sphere_uv function*

Update the `sphere::hit()` function to use this function to update the hit record UV coordinates.

```
class sphere : public hittable {
  public:
    ...
    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {
        ...

        rec.t = root;
        rec.p = r.at(rec.t);
        vec3 outward_normal = (rec.p - current_center) / radius;
        rec.set_face_normal(r, outward_normal);
        get_sphere_uv(outward_normal, rec.u, rec.v);
        rec.mat = mat;

        return true;
    }
    ...
};
```

Listing 30: `[sphere.h]` *Sphere UV coordinates from hit*

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.

4.5. Accessing Texture Image Data

Now it's time to create a texture class that holds an image. I am going to use my favorite image utility, stb_image. It reads image data into an array of 32-bit floating-point values. These are just packed RGBs with each component in the range [0,1] (black to full white). In addition, images are loaded in linear color space (gamma = 1) — the color space in which we do all our computations.

To help make loading our image files even easier, we provide a helper class to manage all this: `rtw_image`. It provides a helper function — `pixel_data(int x, int y)` — to get the 8-bit RGB byte values for each pixel. The following listing assumes that you have copied the `stb_image.h` header into a folder called `external`. Adjust according to your directory structure.

```
#ifndef RTW_STB_IMAGE_H
#define RTW_STB_IMAGE_H

// Disable strict warnings for this header from the Microsoft Visual C++ compiler.
#ifdef _MSC_VER
    #pragma warning (push, 0)
#endif

#define STB_IMAGE_IMPLEMENTATION
#define STBI_FAILURE_USERMSG
#include "external/stb_image.h"

#include <cstdlib>
#include <iostream>

class rtw_image {
  public:
    rtw_image() {}

    rtw_image(const char* image_filename) {
        // Loads image data from the specified file. If the RTW_IMAGES environment variable is
        // defined, looks only in that directory for the image file. If the image was not found,
        // searches for the specified image file first from the current directory, then in the
        // images/ subdirectory, then the _parent's_ images/ subdirectory, and then _that_
        // parent, on so on, for six levels up. If the image was not loaded successfully,
        // width() and height() will return 0.

        auto filename = std::string(image_filename);
        auto imagedir = getenv("RTW_IMAGES");

        // Hunt for the image file in some likely locations.
        if (imagedir && load(std::string(imagedir) + "/" + image_filename)) return;
        if (load(filename)) return;
        if (load("images/" + filename)) return;
        if (load("../images/" + filename)) return;
        if (load("../../images/" + filename)) return;
        if (load("../../../images/" + filename)) return;
        if (load("../../../../images/" + filename)) return;
        if (load("../../../../../images/" + filename)) return;
        if (load("../../../../../../images/" + filename)) return;

        std::cerr << "ERROR: Could not load image file '" << image_filename << "'.\n";
    }

    ~rtw_image() {
        delete[] bdata;
        STBI_FREE(fdata);
    }

    bool load(const std::string& filename) {
        // Loads the linear (gamma=1) image data from the given file name. Returns true if the
        // load succeeded. The resulting data buffer contains the three [0.0, 1.0]
        // floating-point values for the first pixel (red, then green, then blue). Pixels are
        // contiguous, going left to right for the width of the image, followed by the next row
        // below, for the full height of the image.

        auto n = bytes_per_pixel; // Dummy out parameter: original components per pixel
        fdata = stbi_loadf(filename.c_str(), &image_width, &image_height, &n, bytes_per_pixel);
        if (fdata == nullptr) return false;

        bytes_per_scanline = image_width * bytes_per_pixel;
        convert_to_bytes();
        return true;
    }

    int width()  const { return (fdata == nullptr) ? 0 : image_width; }
    int height() const { return (fdata == nullptr) ? 0 : image_height; }

    const unsigned char* pixel_data(int x, int y) const {
        // Return the address of the three RGB bytes of the pixel at x,y. If there is no image
        // data, returns magenta.
        static unsigned char magenta[] = { 255, 0, 255 };
        if (bdata == nullptr) return magenta;
```

```
        x = clamp(x, 0, image_width);
        y = clamp(y, 0, image_height);

        return bdata + y*bytes_per_scanline + x*bytes_per_pixel;
    }

  private:
    const int      bytes_per_pixel = 3;
    float         *fdata = nullptr;         // Linear floating point pixel data
    unsigned char *bdata = nullptr;         // Linear 8-bit pixel data
    int            image_width = 0;         // Loaded image width
    int            image_height = 0;        // Loaded image height
    int            bytes_per_scanline = 0;

    static int clamp(int x, int low, int high) {
        // Return the value clamped to the range [low, high).
        if (x < low) return low;
        if (x < high) return x;
        return high - 1;
    }

    static unsigned char float_to_byte(float value) {
        if (value <= 0.0)
            return 0;
        if (1.0 <= value)
            return 255;
        return static_cast<unsigned char>(256.0 * value);
    }

    void convert_to_bytes() {
        // Convert the linear floating point pixel data to bytes, storing the resulting byte
        // data in the `bdata` member.

        int total_bytes = image_width * image_height * bytes_per_pixel;
        bdata = new unsigned char[total_bytes];

        // Iterate through all pixel components, converting from [0.0, 1.0] float values to
        // unsigned [0, 255] byte values.

        auto *bptr = bdata;
        auto *fptr = fdata;
        for (auto i=0; i < total_bytes; i++, fptr++, bptr++)
            *bptr = float_to_byte(*fptr);
    }
};

// Restore MSVC compiler warnings
#ifdef _MSC_VER
    #pragma warning (pop)
#endif

#endif
```

Listing 31: `[rtw_stb_image.h] The rtw_image helper class`

If you are writing your implementation in a language other than C or C++, you'll need to locate (or write) an image loading library that provides similar functionality.

The `image_texture` class uses the `rtw_image` class:

```
#include "rtw_stb_image.h"

...

class checker_texture : public texture {
    ...
};

class image_texture : public texture {
  public:
    image_texture(const char* filename) : image(filename) {}

    color value(double u, double v, const point3& p) const override {
        // If we have no texture data, then return solid cyan as a debugging aid.
        if (image.height() <= 0) return color(0,1,1);

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

        auto i = int(u * image.width());
        auto j = int(v * image.height());
        auto pixel = image.pixel_data(i,j);

        auto color_scale = 1.0 / 255.0;
        return color(color_scale*pixel[0], color_scale*pixel[1], color_scale*pixel[2]);
    }

  private:
    rtw_image image;
};
```

Listing 32: `[texture.h]` *Image texture class*

4.6. Rendering The Image Texture

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

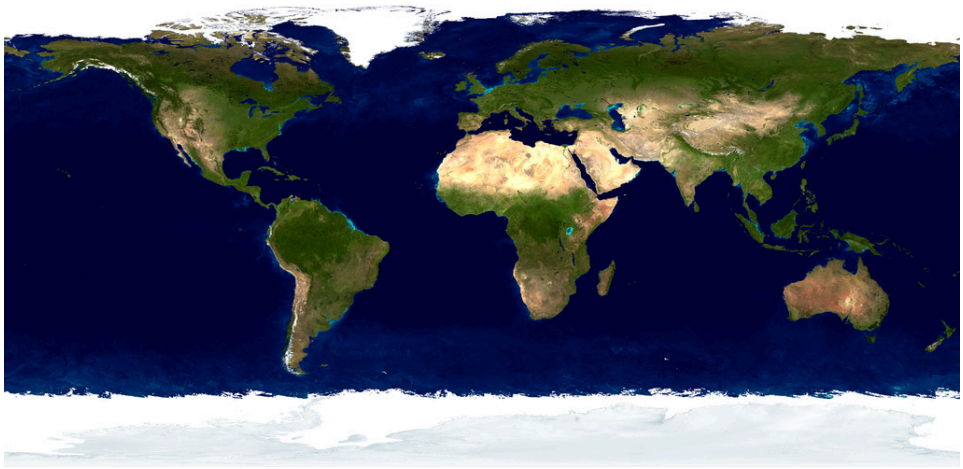

Image 4: *earthmap.jpg*

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

```
void 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);

    camera cam;

    cam.aspect_ratio      = 16.0 / 9.0;
    cam.image_width       = 400;
    cam.samples_per_pixel = 100;
    cam.max_depth         = 50;

    cam.vfov     = 20;
    cam.lookfrom = point3(0,0,12);
    cam.lookat   = point3(0,0,0);
    cam.vup      = vec3(0,1,0);

    cam.defocus_angle = 0;

    cam.render(hittable_list(globe));
}

int main() {
    switch (3) {
        case 1:  bouncing_spheres();  break;
        case 2:  checkered_spheres(); break;
        case 3:  earth();             break;
    }
}
```

Listing 33: `[main.cc]` *Rendering with a texture map of Earth*

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.

If the photo comes back with a large cyan sphere in the middle, then `stb_image` failed to find your Earth map photo. The program will look for the file in the same directory as the executable. Make sure to copy the Earth into your build directory, or rewrite `earth()` to point somewhere else.

Image 5: *Earth-mapped sphere*

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:

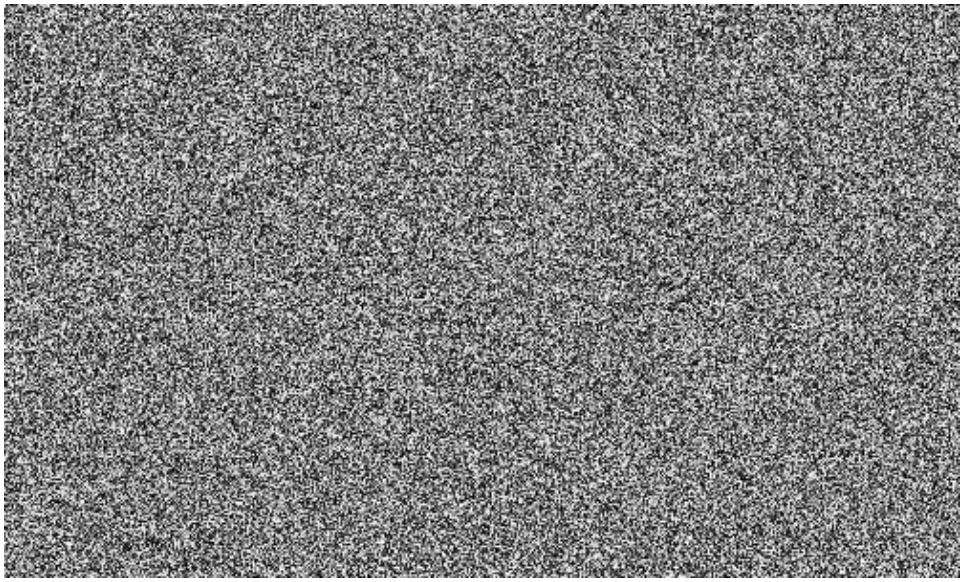

Image 6: *White noise*

Instead it returns something similar to blurred white noise:

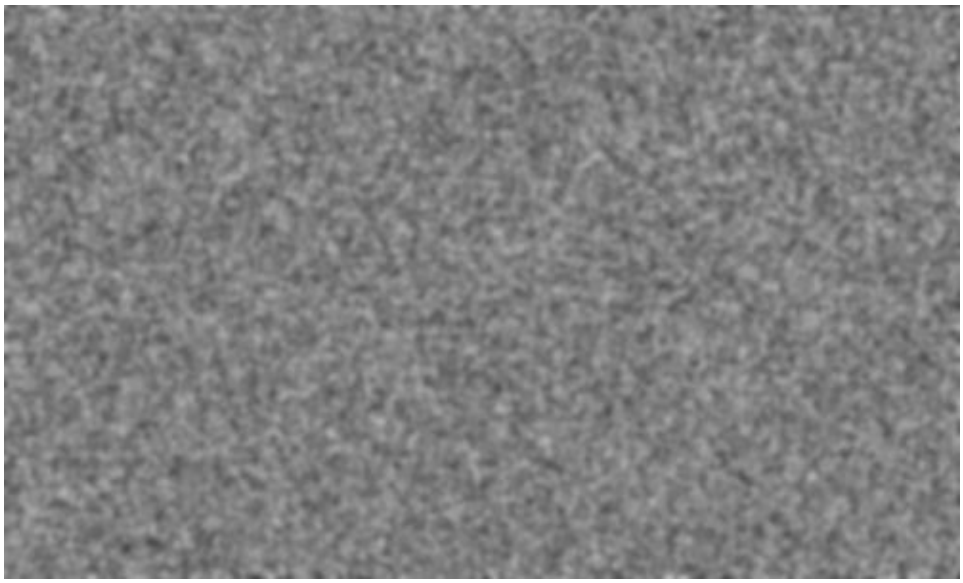

Image 7: *White noise, blurred*

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:

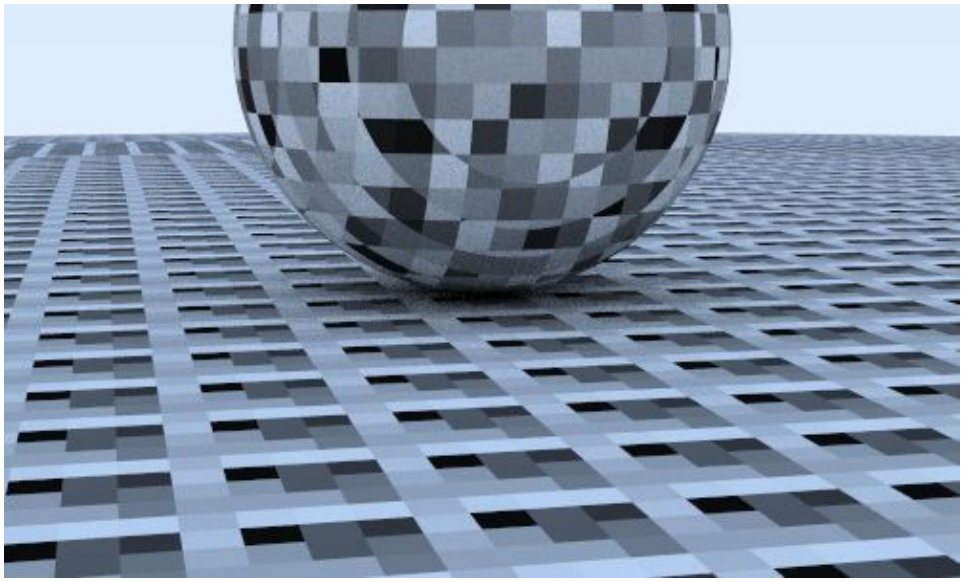

Image 8: *Tiled random patterns*

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:

```
#ifndef PERLIN_H
#define PERLIN_H

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

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

    double noise(const point3& p) const {
        auto i = int(4*p.x()) & 255;
        auto j = int(4*p.y()) & 255;
        auto k = int(4*p.z()) & 255;

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

  private:
    static const int point_count = 256;
    double randfloat[point_count];
    int perm_x[point_count];
    int perm_y[point_count];
    int perm_z[point_count];

    static void perlin_generate_perm(int* p) {
        for (int i = 0; i < point_count; i++)
            p[i] = i;

        permute(p, point_count);
    }

    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;
        }
    }
};

#endif
```

Listing 34: `[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"
#include "rtw_stb_image.h"

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

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

  private:
    perlin noise;
};
```

Listing 35: `[texture.h]` *Noise texture*

We can use that texture on some spheres:

```
void perlin_spheres() {
    hittable_list world;

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

    camera cam;

    cam.aspect_ratio      = 16.0 / 9.0;
    cam.image_width       = 400;
    cam.samples_per_pixel = 100;
    cam.max_depth         = 50;

    cam.vfov     = 20;
    cam.lookfrom = point3(13,2,3);
    cam.lookat   = point3(0,0,0);
    cam.vup      = vec3(0,1,0);

    cam.defocus_angle = 0;

    cam.render(world);
}

int main() {
    switch (4) {
        case 1:  bouncing_spheres();   break;
        case 2:  checkered_spheres();  break;
        case 3:  earth();              break;
        case 4:  perlin_spheres();     break;
    }
}
```

Listing 36: `[main.cc]` *Scene with two Perlin-textured spheres*

And the hashing does scramble as hoped:

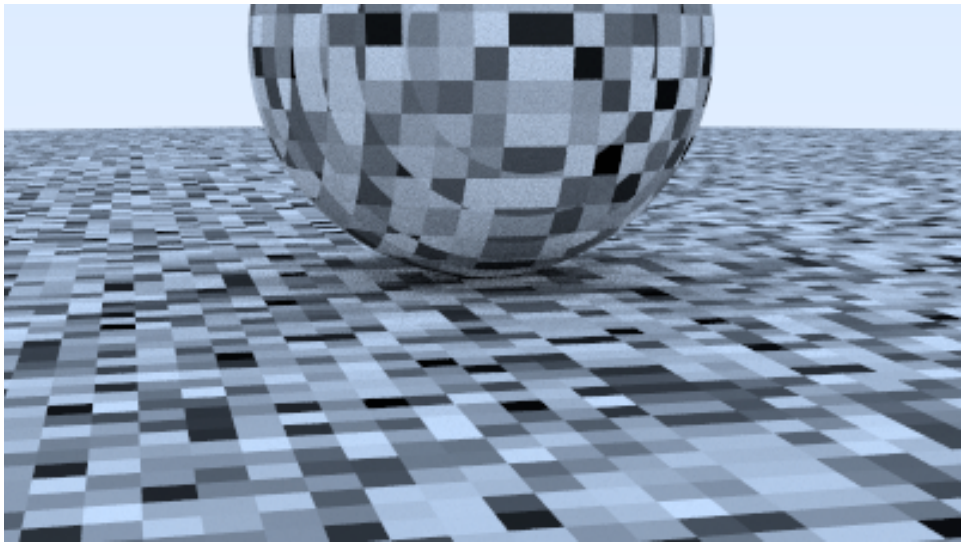

Image 9: *Hashed random texture*

5.2. Smoothing out the Result

To make it smooth, we can linearly interpolate:

```
class perlin {
  public:
    ...

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

        auto i = int(std::floor(p.x()));
        auto j = int(std::floor(p.y()));
        auto k = int(std::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] = randfloat[
                        perm_x[(i+di) & 255] ^
                        perm_y[(j+dj) & 255] ^
                        perm_z[(k+dk) & 255]
                    ];

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

    ...

  private:
    ...

    static void permute(int* p, int n) {
        ...
    }

    static 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;
    }
};
```

Listing 37: `[perlin.h]` *Perlin with trilinear interpolation*

And we get:

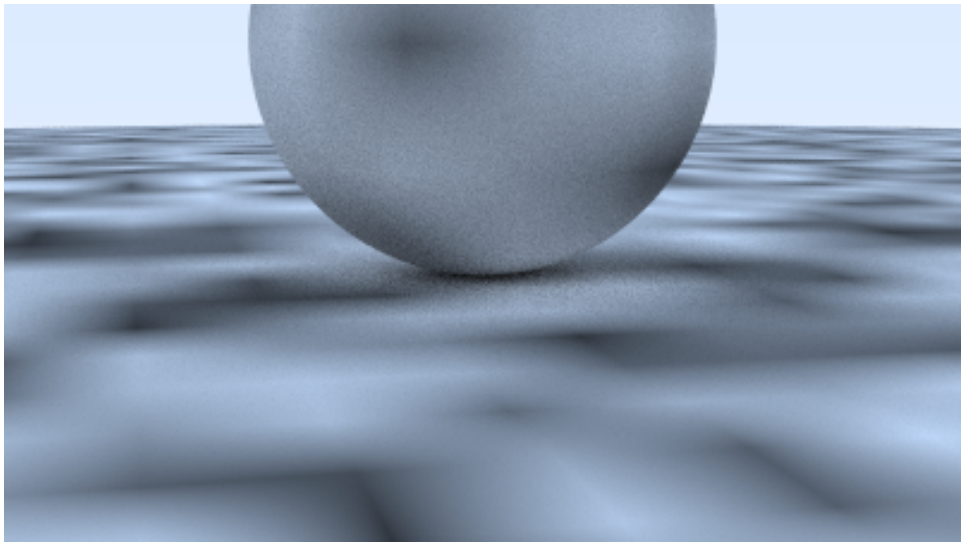

Image 10: *Perlin texture with trilinear interpolation*

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() - std::floor(p.x());
        auto v = p.y() - std::floor(p.y());
        auto w = p.z() - std::floor(p.z());
        u = u*u*(3-2*u);
        v = v*v*(3-2*v);
        w = w*w*(3-2*w);

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

Listing 38: `[perlin.h]` *Perlin with Hermitian smoothing*

This gives a smoother looking image:

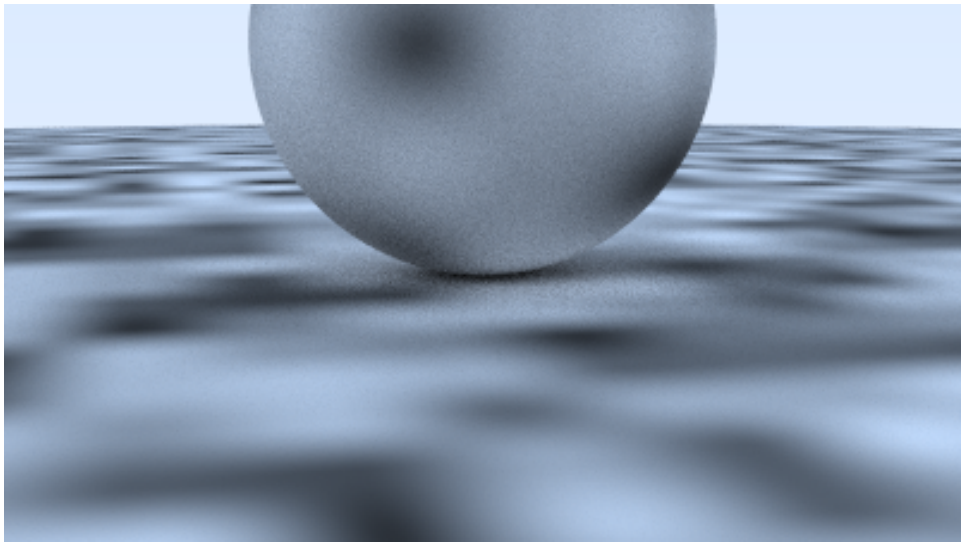

Image 11: *Perlin texture, trilinearly interpolated, smoothed*

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(double scale) : scale(scale) {}

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

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

Listing 39: `[texture.h]` *Perlin smoothed, higher frequency*

We then add that scale to the `perlin_spheres()` scene description:

```
void perlin_spheres() {
    ...
    auto pertext = make_shared<noise_texture>(4);
    world.add(make_shared<sphere>(point3(0,-1000,0), 1000, make_shared<lambertian>(pertext)));
    world.add(make_shared<sphere>(point3(0, 2, 0), 2, make_shared<lambertian>(pertext)));

    camera cam;
    ...
}
```

Listing 40: `[main.cc]` *Perlin-textured spheres with a scale to the noise*

This yields the following result:

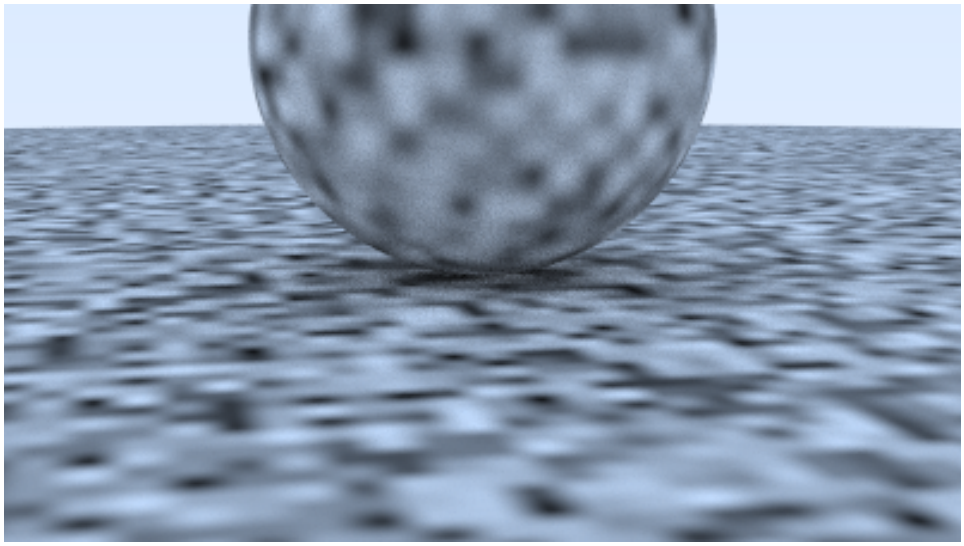

Image 12: *Perlin texture, higher frequency*

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

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

    ...

  private:
    static const int point_count = 256;
    vec3 randvec[point_count];
    int perm_x[point_count];
    int perm_y[point_count];
    int perm_z[point_count];
    ...
};
```

Listing 41: `[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() - std::floor(p.x());
        auto v = p.y() - std::floor(p.y());
        auto w = p.z() - std::floor(p.z());
        u = u*u*(3-2*u);
        v = v*v*(3-2*v);
        w = w*w*(3-2*w);

        auto i = int(std::floor(p.x()));
        auto j = int(std::floor(p.y()));
        auto k = int(std::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] = randvec[
                        perm_x[(i+di) & 255] ^
                        perm_y[(j+dj) & 255] ^
                        perm_z[(k+dk) & 255]
                    ];

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

    ...
};
```

Listing 42: `[perlin.h]` *Perlin class with new noise() method*

And the interpolation becomes a bit more complicated:

```
class perlin {
  ...
  private:
    ...
    static double trilinear_interp(double c[2][2][2], double u, double v, double w) {
        ...
    }

    static double perlin_interp(const 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 43: `[perlin.h]` *Perlin interpolation function so far*

The output of the Perlin interpolation function can return negative values. These negative values will be passed to our `linear_to_gamma()` color function, which expects only positive inputs. To mitigate this, we'll map the $[-1,+1]$ range of values to $[0,1]$.

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

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

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

Listing 44: `[texture.h]` *Perlin smoothed, higher frequency*

This finally gives something more reasonable looking:

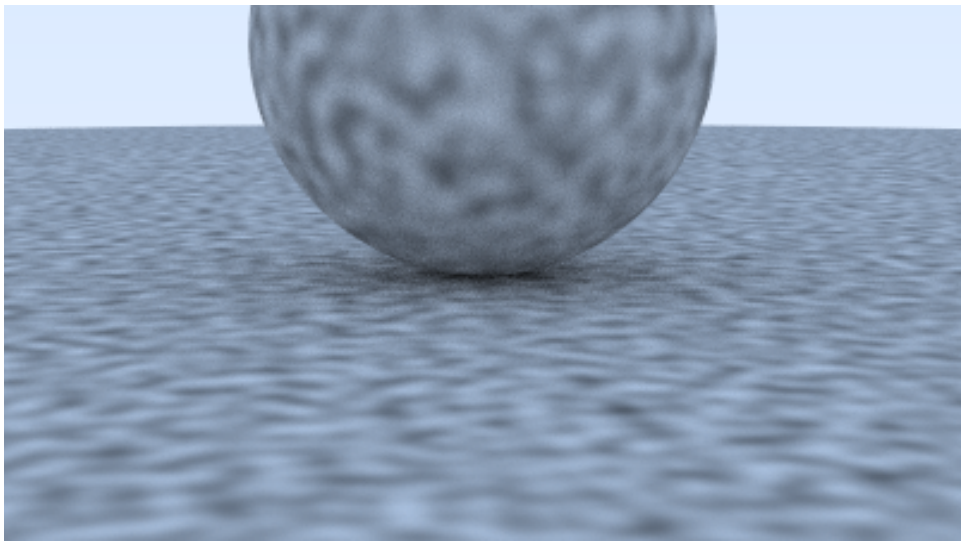

Image 13: *Perlin texture, shifted off integer values*

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 noise(const point3& p) const {
        ...
    }

    double turb(const point3& p, int depth) 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 std::fabs(accum);
    }

    ...
```

Listing 45: `[perlin.h]` *Turbulence function*

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

    color value(double u, double v, const point3& p) const override {
        return color(1,1,1) * noise.turb(p, 7);
    }

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

Listing 46: `[texture.h]` *Noise texture with turbulence*

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

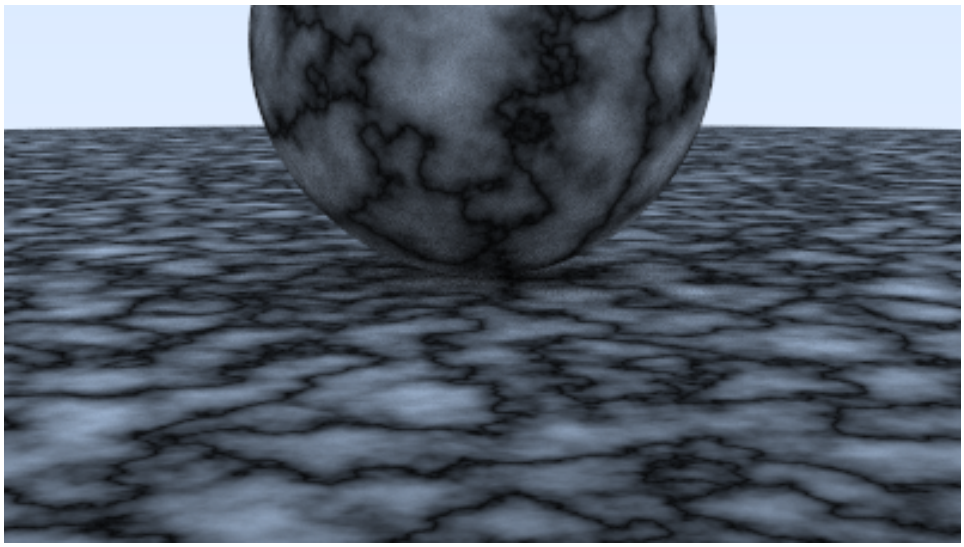

Image 14: *Perlin texture with turbulence*

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(double scale) : scale(scale) {}

    color value(double u, double v, const point3& p) const override {
        return color(.5, .5, .5) * (1 + std::sin(scale * p.z() + 10 * noise.turb(p, 7)));
    }

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

Listing 47: `[texture.h]` *Noise texture with marbled texture*

Which yields:

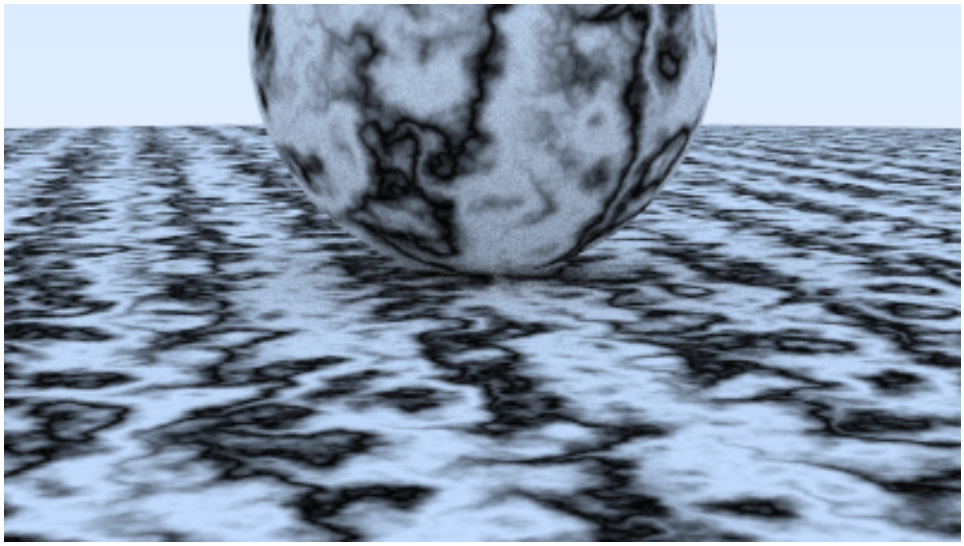

Image 15: *Perlin noise, marbled texture*

6. Quadrilaterals

We've managed to get more than half way through this three-book series using spheres as our only geometric primitive. Time to add our second primitive: the quadrilateral.

6.1. Defining the Quadrilateral

Though we'll name our new primitive a `quad`, it will technically be a parallelogram (opposite sides are parallel) instead of a general quadrilateral. For our purposes, we'll use three geometric entities to define a quad:

1. $\mathbf{Q}$, the starting corner.
2. $\mathbf{u}$, a vector representing the first side. $\mathbf{Q} + \mathbf{u}$ gives one of the corners adjacent to $\mathbf{Q}$.
3. $\mathbf{v}$, a vector representing the second side. $\mathbf{Q} + \mathbf{v}$ gives the other corner adjacent to $\mathbf{Q}$.

The corner of the quad opposite $\mathbf{Q}$ is given by $\mathbf{Q} + \mathbf{u} + \mathbf{v}$. These values are three-dimensional, even though a quad itself is a two-dimensional object. For example, a quad with corner at the origin and extending two units in the Z direction and one unit in the Y direction would have values $\mathbf{Q} = (0, 0, 0), \mathbf{u} = (0, 0, 2), \text{and } \mathbf{v} = (0, 1, 0)$.

The following figure illustrates the quadrilateral components.

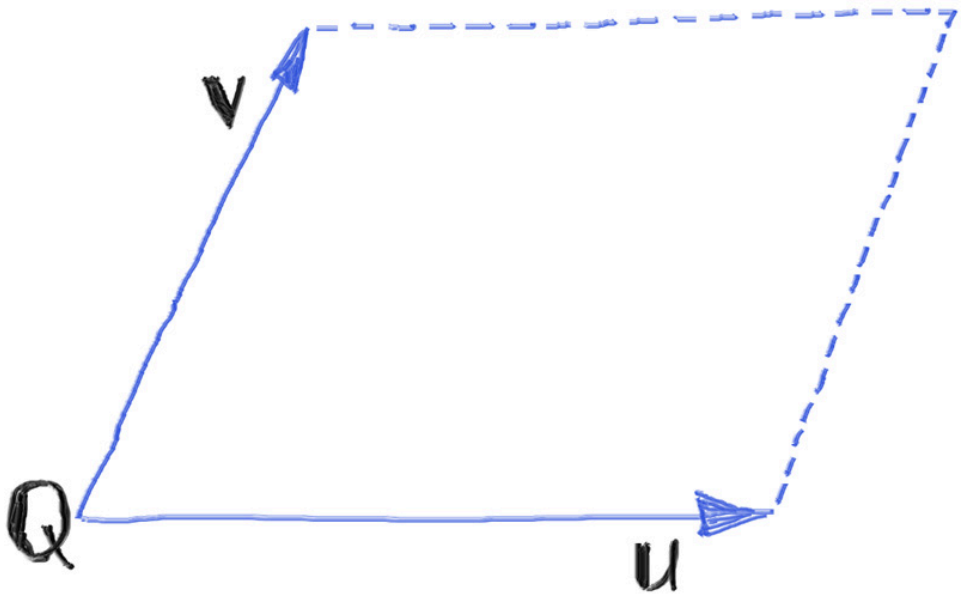

Figure 5: *Quadrilateral Components*

Quads are flat, so their axis-aligned bounding box will have zero thickness in one dimension if the quad lies in the XY, YZ, or ZX plane. This can lead to numerical problems with ray intersection, but we can address this by padding any zero-sized dimensions of the bounding box. Padding is fine because we aren't changing the intersection of the quad; we're only expanding its bounding box to remove the possibility of numerical problems, and the bounds are just a rough approximation to the actual shape anyway. To remedy this situation, we insert a small padding to ensure that newly constructed AABBs always have a non-zero volume:

```
...
class aabb {
  public:
    ...
    aabb(const interval& x, const interval& y, const interval& z)
      : x(x), y(y), z(z)
    {
        pad_to_minimums();
    }


    aabb(const point3& a, const point3& b) {
        // Treat the two points a and b as extrema for the bounding box, so we don't require a
        // particular minimum/maximum coordinate order.

        x = interval(std::fmin(a[0],b[0]), std::fmax(a[0],b[0]));
        y = interval(std::fmin(a[1],b[1]), std::fmax(a[1],b[1]));
        z = interval(std::fmin(a[2],b[2]), std::fmax(a[2],b[2]));

        pad_to_minimums();
    }
    ...
    static const aabb empty, universe;

  private:

    void pad_to_minimums() {
        // Adjust the AABB so that no side is narrower than some delta, padding if necessary.

        double delta = 0.0001;
        if (x.size() < delta) x = x.expand(delta);
        if (y.size() < delta) y = y.expand(delta);
        if (z.size() < delta) z = z.expand(delta);
    }
};
```

Listing 48: `[aabb.h]` *New aabb::pad_to_minimums() method*

Now we're ready for the first sketch of the new `quad` class:

```
#ifndef QUAD_H
#define QUAD_H

#include "hittable.h"

class quad : public hittable {
  public:
    quad(const point3& Q, const vec3& u, const vec3& v, shared_ptr<material> mat)
      : Q(Q), u(u), v(v), mat(mat)
    {
        set_bounding_box();
    }

    virtual void set_bounding_box() {
        // Compute the bounding box of all four vertices.
        auto bbox_diagonal1 = aabb(Q, Q + u + v);
        auto bbox_diagonal2 = aabb(Q + u, Q + v);
        bbox = aabb(bbox_diagonal1, bbox_diagonal2);
    }

    aabb bounding_box() const override { return bbox; }

    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {
        return false; // To be implemented
    }

  private:
    point3 Q;
    vec3 u, v;
    shared_ptr<material> mat;
    aabb bbox;
};

#endif
```

Listing 49: `[quad.h]` *2D quadrilateral (parallelogram) class*

6.2. Ray-Plane Intersection

As you can see in the prior listing, `quad::hit()` remains to be implemented. Just as for spheres, we need to determine whether a given ray intersects the primitive, and if so, the various properties of that intersection (hit point, normal, texture coordinates and so forth).

Ray-quad intersection will be determined in three steps:

1. finding the plane that contains that quad,
2. solving for the intersection of a ray and the quad-containing plane,
3. determining if the hit point lies inside the quad.

We'll first tackle the middle step, solving for general ray-plane intersection.

Spheres are generally the first ray tracing primitive taught because their implicit formula makes it so easy to solve for ray intersection. Like spheres, planes also have an implicit formula, and we can use their implicit formula to produce an algorithm that solves for ray-plane intersection. Indeed, ray-plane intersection is even *easier* to solve than ray-sphere intersection.

You may already know this implicit formula for a plane:

$$Ax + By + Cz + D = 0$$

where A, B, C, D are just constants, and x, y, z are the values of any point (x, y, z) that lies on the plane. A plane is thus the set of all points (x, y, z) that satisfy the formula above. It makes things slightly easier to use the alternate

formulation:

$$Ax + By + Cz = D$$

(We didn't flip the sign of D because it's just some constant that we'll figure out later.)

Here's an intuitive way to think of this formula: given the plane perpendicular to the normal vector $\mathbf{n} = (A, B, C)$, and the position vector $\mathbf{v} = (x, y, z)$ (that is, the vector from the origin to any point on the plane), then we can use the dot product to solve for D:

$$\mathbf{n} \cdot \mathbf{v} = D$$

for any position on the plane. This is an equivalent formulation of the $Ax + By + Cz = D$ formula given above, only now in terms of vectors.

Now to find the intersection with some ray $\mathbf{R}(t) = \mathbf{P} + t\mathbf{d}$. Plugging in the ray equation, we get

$$\mathbf{n} \cdot (\mathbf{P} + t\mathbf{d}) = D$$

Solving for t:

$$\mathbf{n} \cdot \mathbf{P} + \mathbf{n} \cdot t\mathbf{d} = D$$

$$\mathbf{n} \cdot \mathbf{P} + t(\mathbf{n} \cdot \mathbf{d}) = D$$

$$t = \frac{D - \mathbf{n} \cdot \mathbf{P}}{\mathbf{n} \cdot \mathbf{d}}$$

This gives us t, which we can plug into the ray equation to find the point of intersection. Note that the denominator $\mathbf{n} \cdot \mathbf{d}$ will be zero if the ray is parallel to the plane. In this case, we can immediately record a miss between the ray and the plane. As for other primitives, if the ray t parameter is less than the minimum acceptable value, we also record a miss.

All right, we can find the point of intersection between a ray and the plane that contains a given quadrilateral. In fact, we can use this approach to test *any* planar primitive, like triangles and disks (more on that later).

6.3. Finding the Plane That Contains a Given Quadrilateral

We've solved step two above: solving the ray-plane intersection, assuming we have the plane equation. To do this, we need to tackle step one above: finding the equation for the plane that contains the quad. We have quadrilateral parameters $\mathbf{Q}$, $\mathbf{u}$, and $\mathbf{v}$, and want the corresponding equation of the plane containing the quad defined by these three values.

Fortunately, this is very simple. Recall that in the equation $Ax + By + Cz = D$, (A, B, C) represents the normal vector. To get this, we just use the cross product of the two side vectors $\mathbf{u}$ and $\mathbf{v}$:

$$\mathbf{n} = \text{unit\_vector}(\mathbf{u} \times \mathbf{v})$$

The plane is defined as all points (x, y, z) that satisfy the equation $Ax + By + Cz = D$. Well, we know that $\mathbf{Q}$ lies on the plane, so that's enough to solve for D:

$$\begin{aligned} D &= n_x Q_x + n_y Q_y + n_z Q_z \\ &= \mathbf{n} \cdot \mathbf{Q} \end{aligned}$$

Add the planar values to the `quad` class:

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

        set_bounding_box();
    }
    ...

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

Listing 50: `[quad.h]` *Caching the planar values*

We will use the two values `normal` and `D` to find the point of intersection between a given ray and the plane containing the quadrilateral.

As an incremental step, let's implement the `hit()` method to handle the infinite plane containing our quadrilateral.

```
class quad : public hittable {
    ...

    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {
        auto denom = dot(normal, r.direction());

        // No hit if the ray is parallel to the plane.
        if (std::fabs(denom) < 1e-8)
            return false;

        // Return false if the hit point parameter t is outside the ray interval.
        auto t = (D - dot(normal, r.origin())) / denom;
        if (!ray_t.contains(t))
            return false;

        auto intersection = r.at(t);

        rec.t = t;
        rec.p = intersection;
        rec.mat = mat;
        rec.set_face_normal(r, normal);

        return true;
    }

    ...
}
```

Listing 51: `[quad.h]` *hit() method for the infinite plane*

6.4. Orienting Points on The Plane

At this stage, the intersection point is on the plane that contains the quadrilateral, but it could be *anywhere* on the plane: the ray-plane intersection point will lie inside or outside the quadrilateral. We need to test for intersection points that lie

inside the quadrilateral (hit), and reject points that lie outside (miss). To determine where a point lies relative to the quad, and to assign texture coordinates to the point of intersection, we need to orient the intersection point on the plane.

To do this, we'll construct a *coordinate frame* for the plane — a way of orienting any point located on the plane. We've already been using a coordinate frame for our 3D space — this is defined by an origin point $\mathbf{O}$ and three basis vectors $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$.

Since a plane is a 2D construct, we just need a plane origin point $\mathbf{Q}$ and *two* basis vectors: $\mathbf{u}$ and $\mathbf{v}$. Normally, axes are perpendicular to each other. However, this doesn't need to be the case in order to span the entire space — you just need two axes that are not parallel to each other.

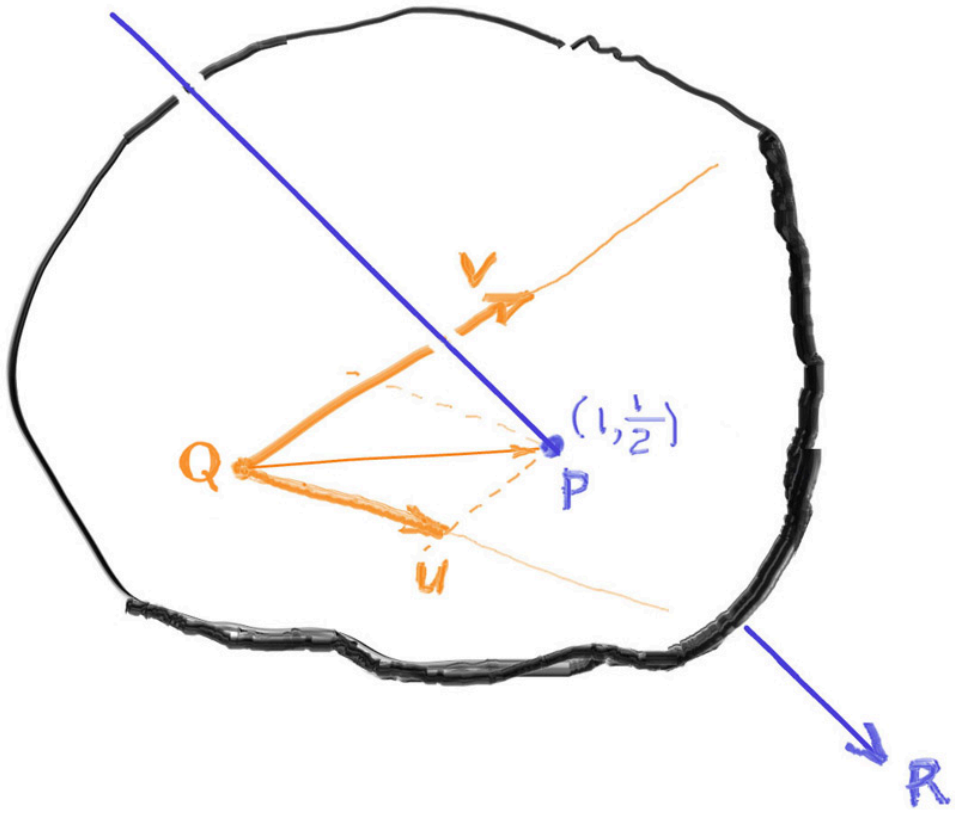

Figure 6: *Ray-plane intersection*

Consider figure 6 as an example. Ray $\mathbf{R}$ intersects the plane, yielding intersection point $\mathbf{P}$ (not to be confused with the ray origin point $\mathbf{P}$ above). Measuring against plane vectors $\mathbf{u}$ and $\mathbf{v}$, the intersection point $\mathbf{P}$ in the example above is at $\mathbf{Q} + (1)\mathbf{u} + (\frac{1}{2})\mathbf{v}$. In other words, the $\mathbf{UV}$ (plane) coordinates of intersection point $\mathbf{P}$ are $(1, \frac{1}{2})$.

Generally, given some arbitrary point $\mathbf{P}$, we seek two scalar values α and β, so that

$$\mathbf{P} = \mathbf{Q} + \alpha\mathbf{u} + \beta\mathbf{v}$$

Pulling a rabbit out of my hat, the planar coordinates α and β are given by the following equations:

$$\alpha = \mathbf{w} \cdot (\mathbf{p} \times \mathbf{v})$$

$$\beta = \mathbf{w} \cdot (\mathbf{u} \times \mathbf{p})$$

where

$$\mathbf{p} = \mathbf{P} - \mathbf{Q}$$

$$\mathbf{w} = \frac{\mathbf{n}}{\mathbf{n} \cdot (\mathbf{u} \times \mathbf{v})} = \frac{\mathbf{n}}{\mathbf{n} \cdot \mathbf{n}}$$

The vector $\mathbf{w}$ is constant for a given quadrilateral, so we'll cache that value.

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

        set_bounding_box();
    }

    ...

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

Listing 52: `[quad.h]` *Caching the quadrilateral's w value*

6.5. Deriving the Planar Coordinates

(This section covers the derivation of the equations above. Feel free to skip to the next section if you're not interested.)

Refer back to figure 6. If the planar basis vectors $\mathbf{u}$ and $\mathbf{v}$ were guaranteed to be orthogonal to each other (forming a 90° angle between them), then solving for α and β would be a simple matter of using the dot product to project $\mathbf{P}$ onto each of the basis vectors $\mathbf{u}$ and $\mathbf{v}$. However, since we are not restricting $\mathbf{u}$ and $\mathbf{v}$ to be orthogonal, the math's a little bit trickier.

To set things up, consider that

$$\mathbf{P} = \mathbf{Q} + \alpha\mathbf{u} + \beta\mathbf{v}$$

$$\mathbf{p} = \mathbf{P} - \mathbf{Q} = \alpha\mathbf{u} + \beta\mathbf{v}$$

Here, $\mathbf{P}$ is the *point* of intersection, and $\mathbf{p}$ is the *vector* from $\mathbf{Q}$ to $\mathbf{P}$.

Cross the equation for $\mathbf{p}$ with $\mathbf{u}$ and $\mathbf{v}$, respectively:

$$\begin{aligned}\mathbf{u} \times \mathbf{p} &= \mathbf{u} \times (\alpha\mathbf{u} + \beta\mathbf{v}) \\ &= \mathbf{u} \times \alpha\mathbf{u} + \mathbf{u} \times \beta\mathbf{v} \\ &= \alpha(\mathbf{u} \times \mathbf{u}) + \beta(\mathbf{u} \times \mathbf{v})\end{aligned}$$

$$\begin{aligned}\mathbf{v} \times \mathbf{p} &= \mathbf{v} \times (\alpha\mathbf{u} + \beta\mathbf{v}) \\ &= \mathbf{v} \times \alpha\mathbf{u} + \mathbf{v} \times \beta\mathbf{v} \\ &= \alpha(\mathbf{v} \times \mathbf{u}) + \beta(\mathbf{v} \times \mathbf{v})\end{aligned}$$

Since any vector crossed with itself yields zero, these equations simplify to

$$\mathbf{v} \times \mathbf{p} = \alpha(\mathbf{v} \times \mathbf{u})$$

$$\mathbf{u} \times \mathbf{p} = \beta(\mathbf{u} \times \mathbf{v})$$

Now to solve for the coefficients α and β. If you're new to vector math, you might try to divide by $\mathbf{u} \times \mathbf{v}$ and $\mathbf{v} \times \mathbf{u}$, but you can't divide by vectors. Instead, we can take the dot product of both sides of the above equations with the plane normal $\mathbf{n} = \mathbf{u} \times \mathbf{v}$, reducing both sides to scalars, which we *can* divide by.

$$\mathbf{n} \cdot (\mathbf{v} \times \mathbf{p}) = \mathbf{n} \cdot \alpha(\mathbf{v} \times \mathbf{u})$$

$$\mathbf{n} \cdot (\mathbf{u} \times \mathbf{p}) = \mathbf{n} \cdot \beta(\mathbf{u} \times \mathbf{v})$$

Now isolating the coefficients is a simple matter of division:

$$\alpha = \frac{\mathbf{n} \cdot (\mathbf{v} \times \mathbf{p})}{\mathbf{n} \cdot (\mathbf{v} \times \mathbf{u})}$$

$$\beta = \frac{\mathbf{n} \cdot (\mathbf{u} \times \mathbf{p})}{\mathbf{n} \cdot (\mathbf{u} \times \mathbf{v})}$$

Reversing the cross products for both the numerator and denominator of α (recall that $\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$) gives us a common denominator for both coefficients:

$$\alpha = \frac{\mathbf{n} \cdot (\mathbf{p} \times \mathbf{v})}{\mathbf{n} \cdot (\mathbf{u} \times \mathbf{v})}$$

$$\beta = \frac{\mathbf{n} \cdot (\mathbf{u} \times \mathbf{p})}{\mathbf{n} \cdot (\mathbf{u} \times \mathbf{v})}$$

Now we can perform one final simplification, computing a vector $\mathbf{w}$ that will be constant for the plane's basis frame, for any planar point $\mathbf{P}$:

$$\mathbf{w} = \frac{\mathbf{n}}{\mathbf{n} \cdot (\mathbf{u} \times \mathbf{v})} = \frac{\mathbf{n}}{\mathbf{n} \cdot \mathbf{n}}$$

$$\alpha = \mathbf{w} \cdot (\mathbf{p} \times \mathbf{v})$$

$$\beta = \mathbf{w} \cdot (\mathbf{u} \times \mathbf{p})$$

6.6. Interior Testing of The Intersection Using UV Coordinates

Now that we have the intersection point's planar coordinates α and β, we can easily use these to determine if the intersection point is inside the quadrilateral — that is, if the ray actually hit the quadrilateral.

The plane is divided into coordinate regions like so:

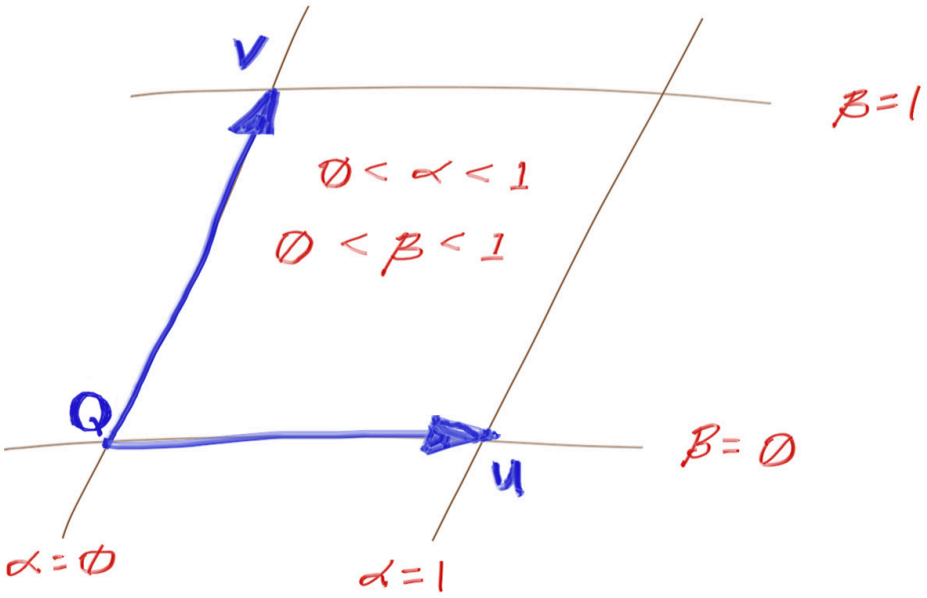

Figure 7: *Quadrilateral coordinates*

Thus, to see if a point with planar coordinates (α, β) lies inside the quadrilateral, it just needs to meet the following criteria:

1. $0 \leq \alpha \leq 1$
2. $0 \leq \beta \leq 1$

That's the last piece needed to implement quadrilateral primitives.

In order to make such experimentation a bit easier, we'll factor out the (α, β) interior test method from the hit method.

```
class quad : public hittable {
  public:
    ...

    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {
        auto denom = dot(normal, r.direction());

        // No hit if the ray is parallel to the plane.
        if (std::fabs(denom) < 1e-8)
            return false;

        // Return false if the hit point parameter t is outside the ray interval.
        auto t = (D - dot(normal, r.origin())) / denom;
        if (!ray_t.contains(t))
            return false;

        // Determine if the hit point lies within the planar shape using its plane coordinates.
        auto intersection = r.at(t);
        vec3 planar_hitpt_vector = intersection - Q;
        auto alpha = dot(w, cross(planar_hitpt_vector, v));
        auto beta = dot(w, cross(u, planar_hitpt_vector));

        if (!is_interior(alpha, beta, rec))
            return false;

        // Ray hits the 2D shape; set the rest of the hit record and return true.
        rec.t = t;
        rec.p = intersection;
        rec.mat = mat;
        rec.set_face_normal(r, normal);

        return true;
    }

    virtual bool is_interior(double a, double b, hit_record& rec) const {
        interval unit_interval = interval(0, 1);
        // Given the hit point in plane coordinates, return false if it is outside the
        // primitive, otherwise set the hit record UV coordinates and return true.

        if (!unit_interval.contains(a) || !unit_interval.contains(b))
            return false;

        rec.u = a;
        rec.v = b;
        return true;
    }

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

Listing 53: `[quad.h]` *Final quad class*

And now we add a new scene to demonstrate our new `quad` primitive:

```
#include "rtweekend.h"

#include "bvh.h"
#include "camera.h"
#include "hittable.h"
#include "hittable_list.h"
#include "material.h"
#include "quad.h"
#include "sphere.h"
#include "texture.h"

...

void quads() {
    hittable_list world;

    // Materials
    auto left_red     = make_shared<lambertian>(color(1.0, 0.2, 0.2));
    auto back_green   = make_shared<lambertian>(color(0.2, 1.0, 0.2));
    auto right_blue   = make_shared<lambertian>(color(0.2, 0.2, 1.0));
    auto upper_orange = make_shared<lambertian>(color(1.0, 0.5, 0.0));
    auto lower_teal   = make_shared<lambertian>(color(0.2, 0.8, 0.8));

    // Quads
    world.add(make_shared<quad>(point3(-3,-2, 5), vec3(0, 0,-4), vec3(0, 4, 0), left_red));
    world.add(make_shared<quad>(point3(-2,-2, 0), vec3(4, 0, 0), vec3(0, 4, 0), back_green));
    world.add(make_shared<quad>(point3( 3,-2, 1), vec3(0, 0, 4), vec3(0, 4, 0), right_blue));
    world.add(make_shared<quad>(point3(-2, 3, 1), vec3(4, 0, 0), vec3(0, 0, 4), upper_orange));
    world.add(make_shared<quad>(point3(-2,-3, 5), vec3(4, 0, 0), vec3(0, 0,-4), lower_teal));

    camera cam;

    cam.aspect_ratio      = 1.0;
    cam.image_width       = 400;
    cam.samples_per_pixel = 100;
    cam.max_depth         = 50;

    cam.vfov     = 80;
    cam.lookfrom = point3(0,0,9);
    cam.lookat   = point3(0,0,0);
    cam.vup      = vec3(0,1,0);

    cam.defocus_angle = 0;

    cam.render(world);
}

int main() {
    switch (5) {
        case 1:  bouncing_spheres();   break;
        case 2:  checkered_spheres();  break;
        case 3:  earth();              break;
        case 4:  perlin_spheres();     break;
        case 5:  quads();              break;
    }
}
```

Listing 54: `[main.cc]` *A new scene with quads*

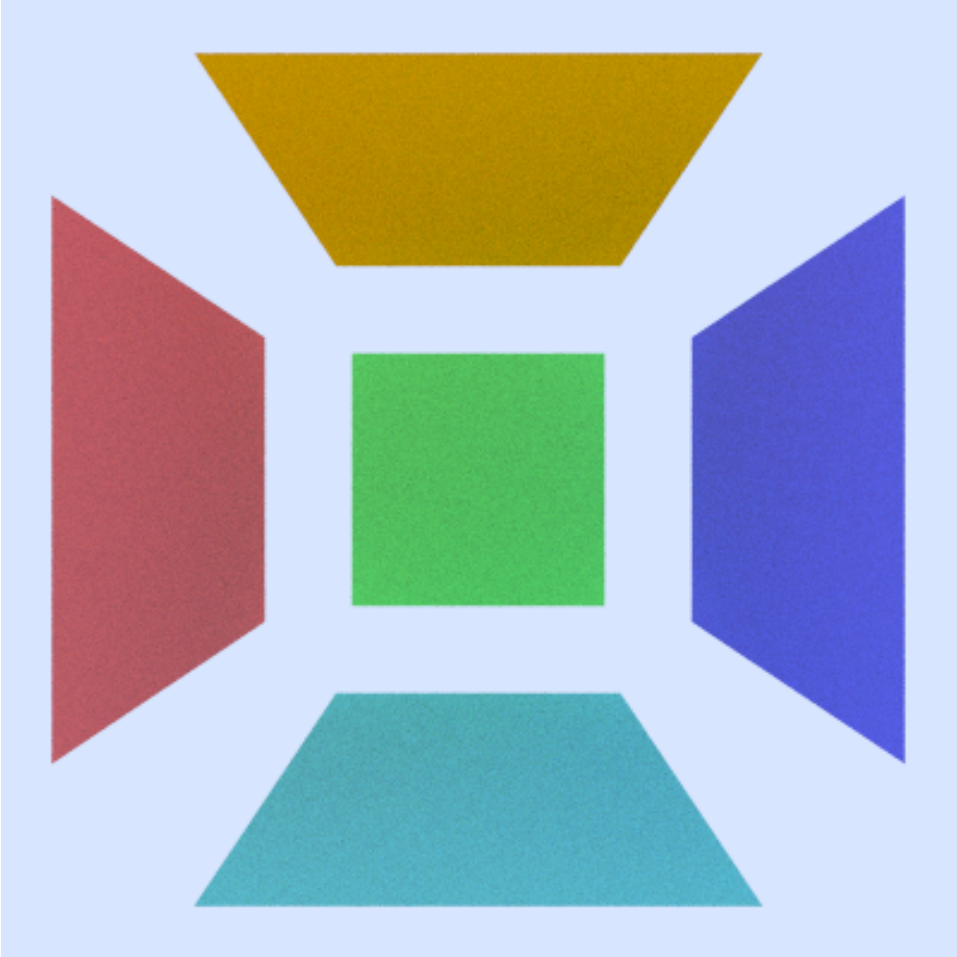

Image 16: *Quads*

6.7. Additional 2D Primitives

Pause a bit here and consider that if you use the (α, β) coordinates to determine if a point lies inside a quadrilateral (parallelogram), it's not too hard to imagine using these same 2D coordinates to determine if the intersection point lies inside *any* other 2D (planar) primitive!

For example, suppose we change the `is_interior()` function to return true if `sqrt(a*a + b*b) < r`. This would then implement disk primitives of radius `r`. For triangles, try `a > 0 && b > 0 && a + b < 1`.

We'll leave additional 2D shape possibilities as an exercise to the reader, depending on your desire to explore. You could even create cut-out stencils based on the pixels of a texture map, or a Mandelbrot shape! As a little Easter egg, check out the `alternate-2D-primitives` tag in the source repository. This has solutions for triangles, ellipses and annuli (rings) in `src/TheNextWeek/quad.h`

7. 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 dielectric : public material {
    ...
}

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

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

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

Listing 55: `[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 ~material() = default;

    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 {
        return false;
    }
};
```

Listing 56: `[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 `color_from_emission` value.

```
class camera {
  public:
    double aspect_ratio      = 1.0;  // Ratio of image width over height
    int    image_width       = 100;  // Rendered image width in pixel count
    int    samples_per_pixel = 10;   // Count of random samples for each pixel
    int    max_depth         = 10;   // Maximum number of ray bounces into scene
    color  background;               // Scene background color

    ...

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

        hit_record rec;

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

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

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

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

        return color_from_emission + color_from_scatter;
    }
};
```

Listing 57: `[camera.h]` *ray_color function with background and emitting materials*

`main()` is updated to set the background color for the prior scenes:

```
void bouncing_spheres() {
    ...
    camera cam;

    cam.aspect_ratio      = 16.0 / 9.0;
    cam.image_width       = 400;
    cam.samples_per_pixel = 100;
    cam.max_depth         = 20;
    cam.background        = color(0.70, 0.80, 1.00);
    ...
}

void checkered_spheres() {
    ...
    camera cam;

    cam.aspect_ratio      = 16.0 / 9.0;
    cam.image_width       = 400;
    cam.samples_per_pixel = 100;
    cam.max_depth         = 50;
    cam.background        = color(0.70, 0.80, 1.00);
    ...
}

void earth() {
    ...
    camera cam;

    cam.aspect_ratio      = 16.0 / 9.0;
    cam.image_width       = 400;
    cam.samples_per_pixel = 100;
    cam.max_depth         = 50;
    cam.background        = color(0.70, 0.80, 1.00);
    ...
}

void perlin_spheres() {
    ...
    camera cam;

    cam.aspect_ratio      = 16.0 / 9.0;
    cam.image_width       = 400;
    cam.samples_per_pixel = 100;
    cam.max_depth         = 50;
    cam.background        = color(0.70, 0.80, 1.00);
    ...
}

void quads() {
    ...
    camera cam;

    cam.aspect_ratio      = 1.0;
    cam.image_width       = 400;
    cam.samples_per_pixel = 100;
    cam.max_depth         = 50;
    cam.background        = color(0.70, 0.80, 1.00);
    ...
}
```

Listing 58: `[main.cc]` *Specifying new background color*

Since we're removing the code that we used to determine the color of the sky when a ray hit it, we need to pass in a new color value for our old scene renders. We've elected to stick with a flat bluish-white for the whole sky. You could always pass in a boolean to switch between the previous skybox code versus the new solid color background. We're keeping it simple here.

7.3. Turning Objects into Lights

If we set up a rectangle as a light:

```
void simple_light() {
    hittable_list world;

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

    auto difflight = make_shared<diffuse_light>(color(4,4,4));
    world.add(make_shared<quad>(point3(3,1,-2), vec3(2,0,0), vec3(0,2,0), difflight));

    camera cam;

    cam.aspect_ratio      = 16.0 / 9.0;
    cam.image_width       = 400;
    cam.samples_per_pixel = 100;
    cam.max_depth         = 50;
    cam.background        = color(0,0,0);

    cam.vfov     = 20;
    cam.lookfrom = point3(26,3,6);
    cam.lookat   = point3(0,2,0);
    cam.vup      = vec3(0,1,0);

    cam.defocus_angle = 0;

    cam.render(world);
}

int main() {
    switch (6) {
        case 1:  bouncing_spheres();   break;
        case 2:  checkered_spheres();  break;
        case 3:  earth();              break;
        case 4:  perlin_spheres();     break;
        case 5:  quads();              break;
        case 6:  simple_light();       break;
    }
}
```

Listing 59: `[main.cc]` *A simple rectangle light*

We get:

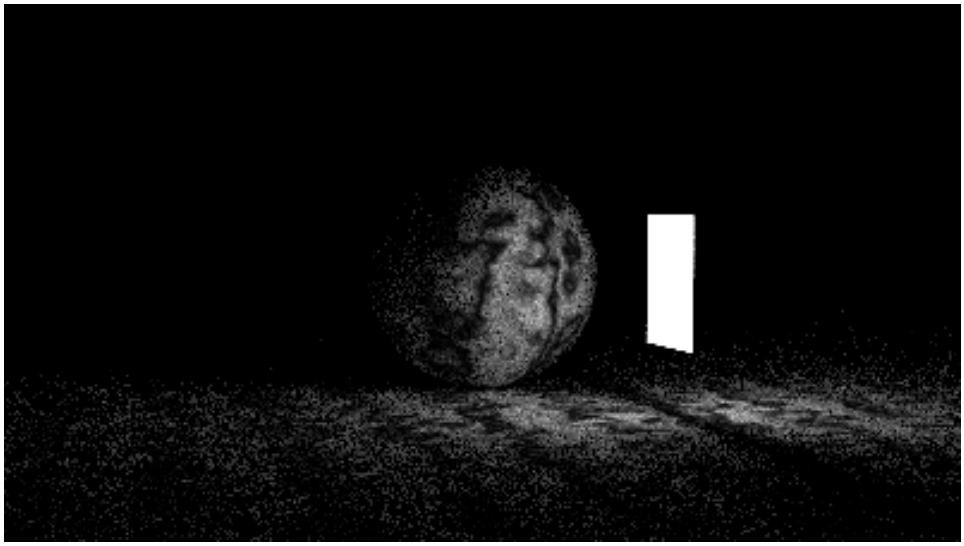

Image 17: *Scene with rectangle light source*

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.

```
void simple_light() {
    ...
    auto difflight = make_shared<diffuse_light>(color(4,4,4));
    world.add(make_shared<sphere>(point3(0,7,0), 2, difflight));
    world.add(make_shared<quad>(point3(3,1,-2), vec3(2,0,0), vec3(0,2,0), difflight));
    ...
}
```

Listing 60: `[main.cc]` *A simple rectangle light plus illuminating ball*

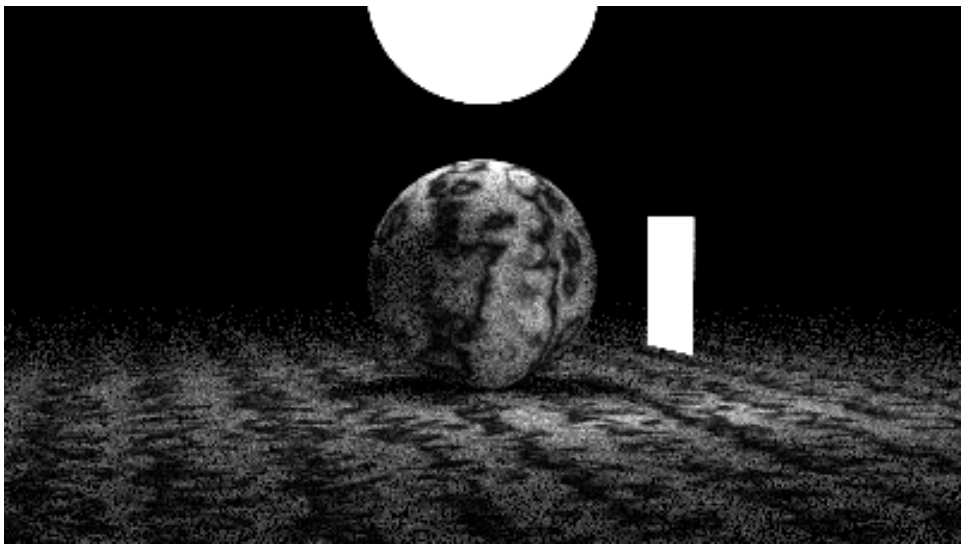

Image 18: *Scene with rectangle and sphere light sources*

7.4. 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:

```
void cornell_box() {
    hittable_list world;

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

    world.add(make_shared<quad>(point3(555,0,0), vec3(0,555,0), vec3(0,0,555), green));
    world.add(make_shared<quad>(point3(0,0,0), vec3(0,555,0), vec3(0,0,555), red));
    world.add(make_shared<quad>(point3(343, 554, 332), vec3(-130,0,0), vec3(0,0,-105), light));
    world.add(make_shared<quad>(point3(0,0,0), vec3(555,0,0), vec3(0,0,555), white));
    world.add(make_shared<quad>(point3(555,555,555), vec3(-555,0,0), vec3(0,0,-555), white));
    world.add(make_shared<quad>(point3(0,0,555), vec3(555,0,0), vec3(0,555,0), white));

    camera cam;

    cam.aspect_ratio      = 1.0;
    cam.image_width       = 600;
    cam.samples_per_pixel = 200;
    cam.max_depth         = 50;
    cam.background        = color(0,0,0);

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

    cam.defocus_angle = 0;

    cam.render(world);
}

int main() {
    switch (7) {
        case 1:  bouncing_spheres();   break;
        case 2:  checkered_spheres();  break;
        case 3:  earth();              break;
        case 4:  perlin_spheres();     break;
        case 5:  quads();              break;
        case 6:  simple_light();       break;
        case 7:  cornell_box();        break;
    }
}
```

Listing 61: `[main.cc]` *Cornell box scene, empty*

We get:

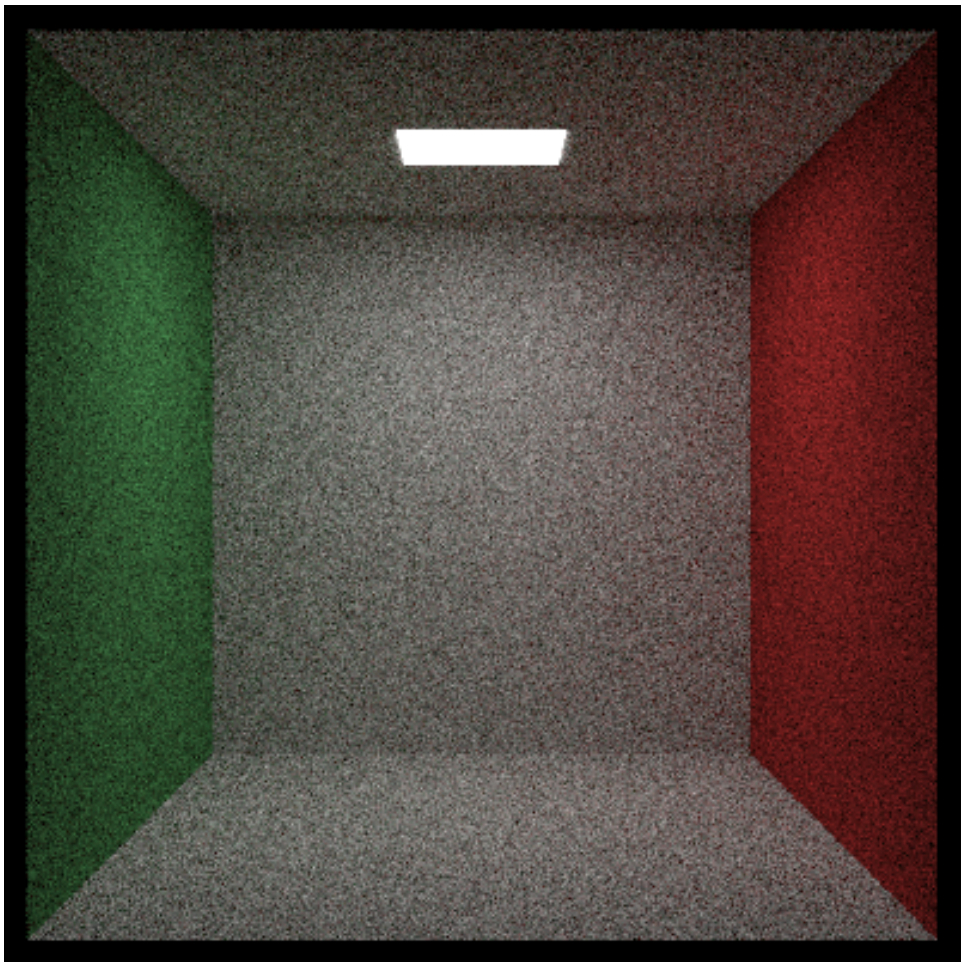

Image 19: *Empty Cornell box*

This image is very noisy because the light is small, so most random rays don't hit the light source.

8. Instances

The Cornell Box usually has two blocks in it. These are rotated relative to the walls. First, let's create a function that returns a box, by creating a `hittable_list` of six rectangles:

```
#include "hittable.h"
#include "hittable_list.h"

class quad : public hittable {
    ...
};

inline shared_ptr<hittable_list> box(const point3& a, const point3& b, shared_ptr<material> mat)
{
    // Returns the 3D box (six sides) that contains the two opposite vertices a & b.

    auto sides = make_shared<hittable_list>();

    // Construct the two opposite vertices with the minimum and maximum coordinates.
    auto min = point3(std::fmin(a.x(),b.x()), std::fmin(a.y(),b.y()), std::fmin(a.z(),b.z()));
    auto max = point3(std::fmax(a.x(),b.x()), std::fmax(a.y(),b.y()), std::fmax(a.z(),b.z()));

    auto dx = vec3(max.x() - min.x(), 0, 0);
    auto dy = vec3(0, max.y() - min.y(), 0);
    auto dz = vec3(0, 0, max.z() - min.z());

    sides->add(make_shared<quad>(point3(min.x(), min.y(), max.z()),  dx,  dy, mat)); // front
    sides->add(make_shared<quad>(point3(max.x(), min.y(), max.z()), -dz,  dy, mat)); // right
    sides->add(make_shared<quad>(point3(max.x(), min.y(), min.z()), -dx,  dy, mat)); // back
    sides->add(make_shared<quad>(point3(min.x(), min.y(), min.z()),  dz,  dy, mat)); // left
    sides->add(make_shared<quad>(point3(min.x(), max.y(), max.z()),  dx, -dz, mat)); // top
    sides->add(make_shared<quad>(point3(min.x(), min.y(), min.z()),  dx,  dz, mat)); // bottom

    return sides;
}
```

Listing 62: `[quad.h]` *A box object*

Now we can add two blocks (but not rotated).

```
void cornell_box() {
    ...
    world.add(make_shared<quad>(point3(555,0,0), vec3(0,555,0), vec3(0,0,555), green));
    world.add(make_shared<quad>(point3(0,0,0), vec3(0,555,0), vec3(0,0,555), red));
    world.add(make_shared<quad>(point3(343, 554, 332), vec3(-130,0,0), vec3(0,0,-105), light));
    world.add(make_shared<quad>(point3(0,0,0), vec3(555,0,0), vec3(0,0,555), white));
    world.add(make_shared<quad>(point3(555,555,555), vec3(-555,0,0), vec3(0,0,-555), white));
    world.add(make_shared<quad>(point3(0,0,555), vec3(555,0,0), vec3(0,555,0), white));

    world.add(box(point3(130, 0, 65), point3(295, 165, 230), white));
    world.add(box(point3(265, 0, 295), point3(430, 330, 460), white));

    camera cam;
    ...
}
```

Listing 63: `[main.cc]` *Adding box objects*

This gives:

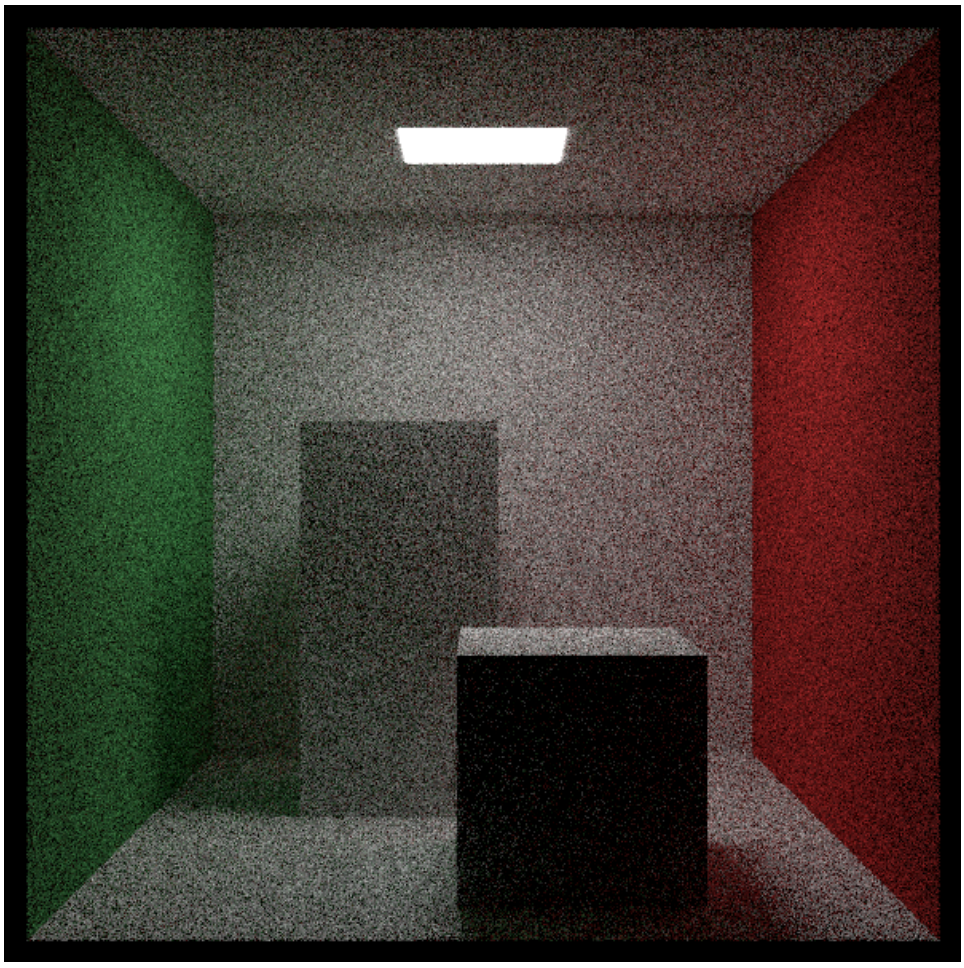

Image 20: *Cornell box with two blocks*

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 copy of a geometric primitive that has been placed into the scene. This instance is entirely independent of the other copies of the primitive and can be moved or rotated. In this case, our geometric primitive is our hittable `box` object, and we want to rotate it. This is especially easy in ray tracing because we don't actually need to move objects in the scene; 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 two 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 two off the x-component of the ray origin.

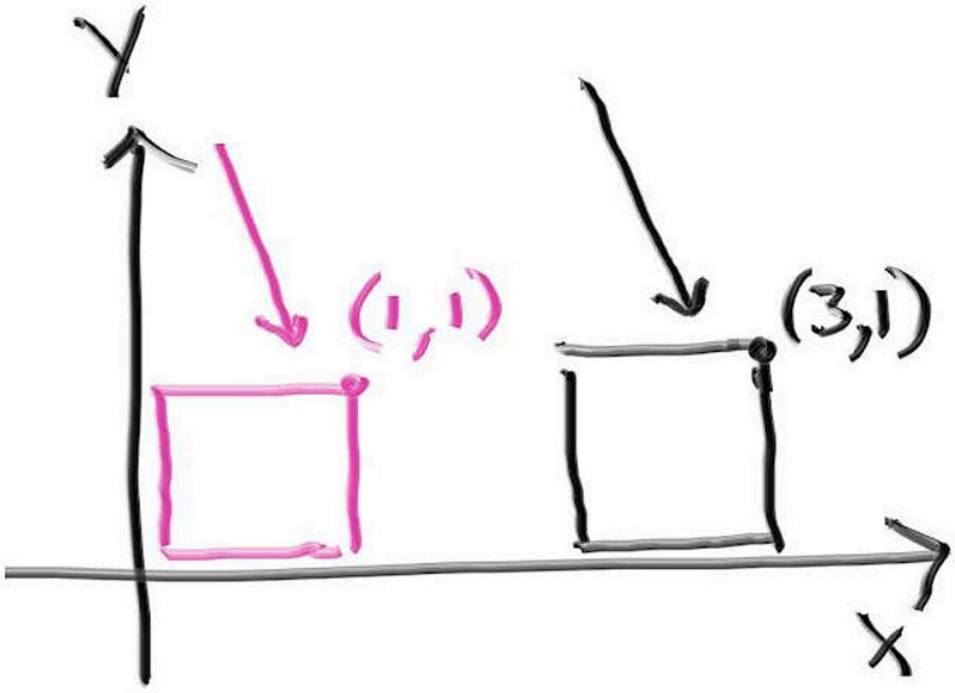

Figure 8: *Ray-box intersection with moved ray vs box*

8.1. Instance Translation

Whether you think of this as a move or a change of coordinates is up to you. The way to reason about this is to think of moving the incident ray backwards the offset amount, determining if an intersection occurs, and then moving that intersection point forward the offset amount.

We need to move the intersection point forward the offset amount so that the intersection is actually in the path of the incident ray. If we forgot to move the intersection point forward then the intersection would be in the path of the offset ray, which isn't correct. Let's add the code to make this happen.

```
class hittable {
    ...
};

class translate : public hittable {
  public:
    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {
        // Move the ray backwards by the offset
        ray offset_r(r.origin() - offset, r.direction(), r.time());

        // Determine whether an intersection exists along the offset ray (and if so, where)
        if (!object->hit(offset_r, ray_t, rec))
            return false;

        // Move the intersection point forwards by the offset
        rec.p += offset;

        return true;
    }

  private:
    shared_ptr<hittable> object;
    vec3 offset;
};
```

Listing 64: `[hittable.h]` *Hittable translation hit function*

... and then flesh out the rest of the `translate` class:

```
class translate : public hittable {
  public:
    translate(shared_ptr<hittable> object, const vec3& offset)
      : object(object), offset(offset)
    {
        bbox = object->bounding_box() + offset;
    }

    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {
        ...
    }

    aabb bounding_box() const override { return bbox; }

  private:
    shared_ptr<hittable> object;
    vec3 offset;
    aabb bbox;
};
```

Listing 65: `[hittable.h]` *Hittable translation class*

We also need to remember to offset the bounding box, otherwise the incident ray might be looking in the wrong place and trivially reject the intersection. The expression `object->bounding_box() + offset` above requires some additional support.

```
class aabb {
    ...
};

const aabb aabb::empty    = aabb(interval::empty,    interval::empty,    interval::empty);
const aabb aabb::universe = aabb(interval::universe, interval::universe, interval::universe);

aabb operator+(const aabb& bbox, const vec3& offset) {
    return aabb(bbox.x + offset.x(), bbox.y + offset.y(), bbox.z + offset.z());
}

aabb operator+(const vec3& offset, const aabb& bbox) {
    return bbox + offset;
}
```

Listing 66: `[aabb.h]` *The aabb + offset operator*

Since each dimension of an `aabb` is represented as an interval, we'll need to extend `interval` with an addition operator as well.

```
class interval {
    ...
};

const interval interval::empty    = interval(+infinity, -infinity);
const interval interval::universe = interval(-infinity, +infinity);

interval operator+(const interval& ival, double displacement) {
    return interval(ival.min + displacement, ival.max + displacement);
}

interval operator+(double displacement, const interval& ival) {
    return ival + displacement;
}
```

Listing 67: `[interval.h]` *The interval + displacement operator*

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.

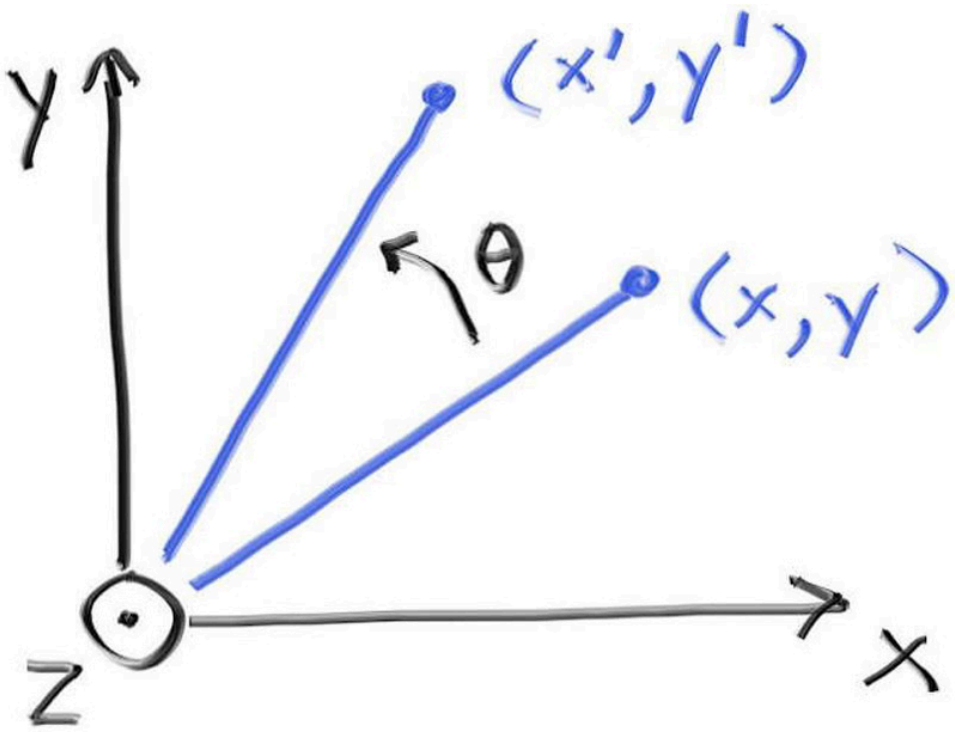

Figure 9: *Rotation about the Z axis*

This involves some basic trigonometry using formulas that I will not cover here. 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 if we want to rotate about the x-axis:

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

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

Thinking of translation as a simple movement of the initial ray is a fine way to reason about what's going on. But, for a more complex operation like a rotation, it can be easy to accidentally get your terms crossed (or forget a negative sign), so it's better to consider a rotation as a change of coordinates.

The pseudocode for the `translate::hit` function above describes the function in terms of *moving*:

1. Move the ray backwards by the offset
2. Determine whether an intersection exists along the offset ray (and if so, where)
3. Move the intersection point forwards by the offset

But this can also be thought of in terms of a *changing of coordinates*:

1. Change the ray from world space to object space
2. Determine whether an intersection exists in object space (and if so, where)
3. Change the intersection point from object space to world space

Rotating an object will not only change the point of intersection, but will also change the surface normal vector, which will change the direction of reflections and refractions. So we need to change the normal as well. Fortunately, the normal will rotate similarly to a vector, so we can use the same formulas as above. While normals and vectors may appear identical for an object undergoing rotation and translation, an object undergoing scaling requires special attention to keep the normals orthogonal to the surface. We won't cover that here, but you should research surface normal transformations if you implement scaling.

We need to start by changing the ray from world space to object space, which for rotation means rotating by $-\theta$.

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

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

We can now create a class for y-rotation. Let's tackle the hit function first:

```
class translate : public hittable {
    ...
};

class rotate_y : public hittable {
  public:

    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {

        // Transform the ray from world space to object space.

        auto origin = point3(
            (cos_theta * r.origin().x()) - (sin_theta * r.origin().z()),
            r.origin().y(),
            (sin_theta * r.origin().x()) + (cos_theta * r.origin().z())
        );

        auto direction = vec3(
            (cos_theta * r.direction().x()) - (sin_theta * r.direction().z()),
            r.direction().y(),
            (sin_theta * r.direction().x()) + (cos_theta * r.direction().z())
        );

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

        // Determine whether an intersection exists in object space (and if so, where).

        if (!object->hit(rotated_r, ray_t, rec))
            return false;

        // Transform the intersection from object space back to world space.

        rec.p = point3(
            (cos_theta * rec.p.x()) + (sin_theta * rec.p.z()),
            rec.p.y(),
            (-sin_theta * rec.p.x()) + (cos_theta * rec.p.z())
        );

        rec.normal = vec3(
            (cos_theta * rec.normal.x()) + (sin_theta * rec.normal.z()),
            rec.normal.y(),
            (-sin_theta * rec.normal.x()) + (cos_theta * rec.normal.z())
        );

        return true;
    }
};
```

Listing 68: `[hittable.h]` *Hittable rotate-Y hit function*

... and now for the rest of the class:

```
class rotate_y : public hittable {
  public:
    rotate_y(shared_ptr<hittable> object, double angle) : object(object) {
        auto radians = degrees_to_radians(angle);
        sin_theta = std::sin(radians);
        cos_theta = std::cos(radians);
        bbox = object->bounding_box();

        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.x.max + (1-i)*bbox.x.min;
                    auto y = j*bbox.y.max + (1-j)*bbox.y.min;
                    auto z = k*bbox.z.max + (1-k)*bbox.z.min;

                    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] = std::fmin(min[c], tester[c]);
                        max[c] = std::fmax(max[c], tester[c]);
                    }
                }
            }
        }

        bbox = aabb(min, max);
    }

    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {
        ...
    }

    aabb bounding_box() const override { return bbox; }

  private:
    shared_ptr<hittable> object;
    double sin_theta;
    double cos_theta;
    aabb bbox;
};
```

Listing 69: `[hittable.h]` *Hittable rotate-Y class*

And the changes to Cornell are:

```
void cornell_box() {
    ...
    world.add(make_shared<quad>(point3(0,0,555), vec3(555,0,0), vec3(0,555,0), white));

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

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

    camera cam;
    ...
}
```

Listing 70: `[main.cc]` *Cornell scene with Y-rotated boxes*

Which yields:

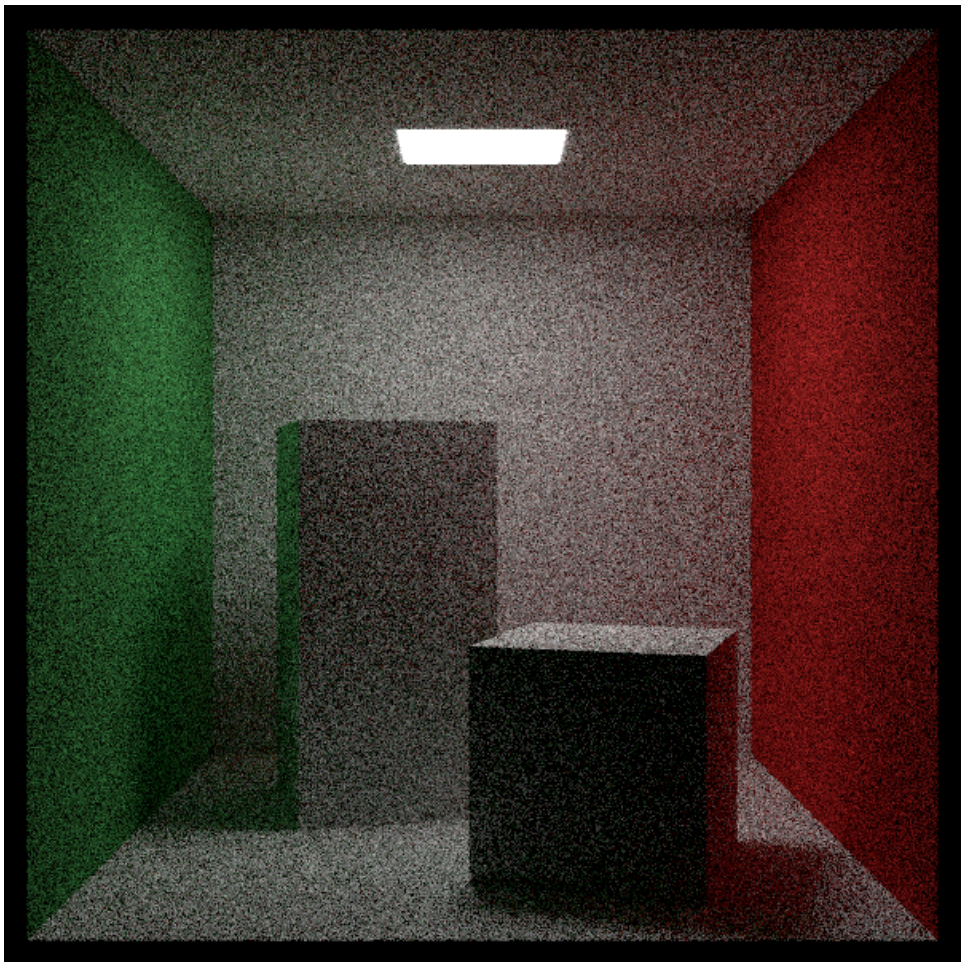

Image 21: *Standard Cornell box scene*

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.

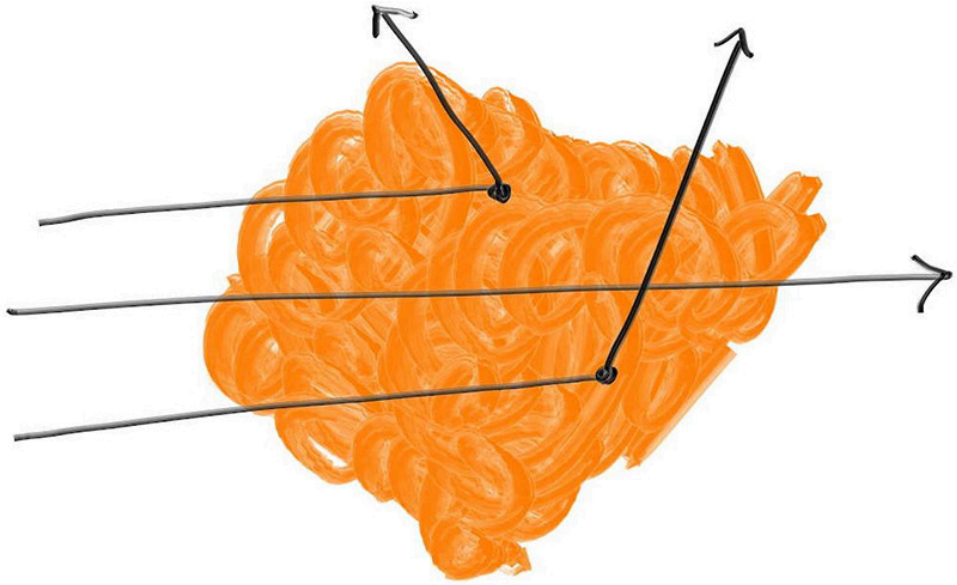

Figure 10: *Ray-volume interaction*

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:

$$probability = C \cdot \Delta L$$

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:

```
#ifndef CONSTANT_MEDIUM_H
#define CONSTANT_MEDIUM_H

#include "hittable.h"
#include "material.h"
#include "texture.h"

class constant_medium : public hittable {
  public:
    constant_medium(shared_ptr<hittable> boundary, double density, shared_ptr<texture> tex)
      : boundary(boundary), neg_inv_density(-1/density),
        phase_function(make_shared<isotropic>(tex))
    {}

    constant_medium(shared_ptr<hittable> boundary, double density, const color& albedo)
      : boundary(boundary), neg_inv_density(-1/density),
        phase_function(make_shared<isotropic>(albedo))
    {}

    bool hit(const ray& r, interval ray_t, hit_record& rec) const override {
        hit_record rec1, rec2;

        if (!boundary->hit(r, interval::universe, rec1))
            return false;

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

        if (rec1.t < ray_t.min) rec1.t = ray_t.min;
        if (rec2.t > ray_t.max) rec2.t = ray_t.max;

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

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

        auto ray_length = r.direction().length();
        auto distance_inside_boundary = (rec2.t - rec1.t) * ray_length;
        auto hit_distance = neg_inv_density * std::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);

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

        return true;
    }

    aabb bounding_box() const override { return boundary->bounding_box(); }

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

#endif
```

Listing 71: `[constant_medium.h]` *Constant medium class*

The scattering function of isotropic picks a uniform random direction:

```
class diffuse_light : public material {
    ...
};

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

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

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

Listing 72: `[material.h]` *The isotropic class*

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:

```
#include "bvh.h"
#include "camera.h"
#include "constant_medium.h"
#include "hittable.h"
#include "hittable_list.h"
#include "material.h"
#include "quad.h"
#include "sphere.h"
#include "texture.h"

...

void cornell_smoke() {
    hittable_list world;

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

    world.add(make_shared<quad>(point3(555,0,0), vec3(0,555,0), vec3(0,0,555), green));
    world.add(make_shared<quad>(point3(0,0,0), vec3(0,555,0), vec3(0,0,555), red));
    world.add(make_shared<quad>(point3(113,554,127), vec3(330,0,0), vec3(0,0,305), light));
    world.add(make_shared<quad>(point3(0,555,0), vec3(555,0,0), vec3(0,0,555), white));
    world.add(make_shared<quad>(point3(0,0,0), vec3(555,0,0), vec3(0,0,555), white));
    world.add(make_shared<quad>(point3(0,0,555), vec3(555,0,0), vec3(0,555,0), white));

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

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

    world.add(make_shared<constant_medium>(box1, 0.01, color(0,0,0)));
    world.add(make_shared<constant_medium>(box2, 0.01, color(1,1,1)));

    camera cam;

    cam.aspect_ratio      = 1.0;
    cam.image_width       = 600;
    cam.samples_per_pixel = 200;
    cam.max_depth         = 50;
    cam.background        = color(0,0,0);

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

    cam.defocus_angle = 0;

    cam.render(world);
}

int main() {
    switch (8) {
        case 1:  bouncing_spheres();   break;
        case 2:  checkered_spheres();  break;
        case 3:  earth();              break;
        case 4:  perlin_spheres();     break;
        case 5:  quads();              break;
        case 6:  simple_light();       break;
        case 7:  cornell_box();        break;
        case 8:  cornell_smoke();      break;
    }
}
```

Listing 73: `[main.cc]` *Cornell box, with smoke*

We get:

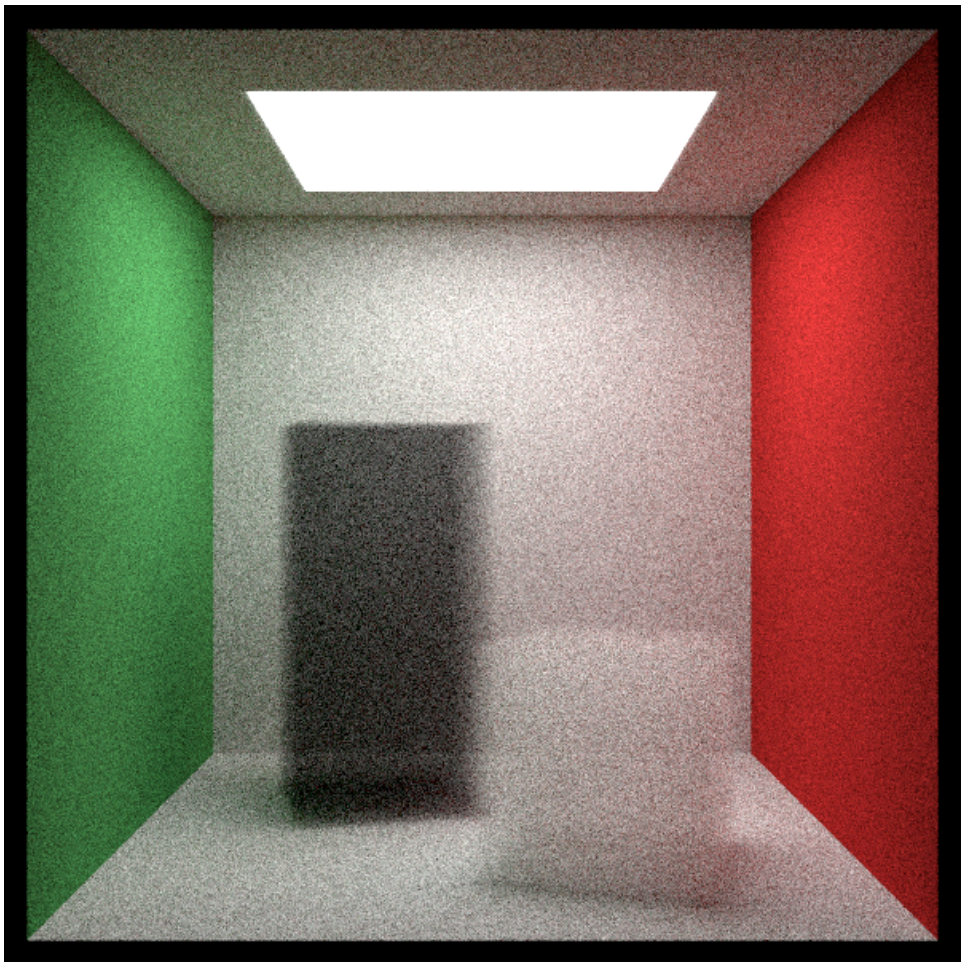

Image 22: *Cornell box with blocks of smoke*

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.

Also note that we'll parameterize this final scene to support a lower quality render for quick testing.

```
void final_scene(int image_width, int samples_per_pixel, int max_depth) {
    hittable_list boxes1;
    auto ground = make_shared<lambertian>(color(0.48, 0.83, 0.53));

    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(box(point3(x0,y0,z0), point3(x1,y1,z1), ground));
        }
    }

    hittable_list world;

    world.add(make_shared<bvh_node>(boxes1));

    auto light = make_shared<diffuse_light>(color(7, 7, 7));
    world.add(make_shared<quad>(point3(123,554,147), vec3(300,0,0), vec3(0,0,265), light));

    auto center1 = point3(400, 400, 200);
    auto center2 = center1 + vec3(30,0,0);
    auto sphere_material = make_shared<lambertian>(color(0.7, 0.3, 0.1));
    world.add(make_shared<sphere>(center1, center2, 50, sphere_material));

    world.add(make_shared<sphere>(point3(260, 150, 45), 50, make_shared<dielectric>(1.5)));
    world.add(make_shared<sphere>(
        point3(0, 150, 145), 50, make_shared<metal>(color(0.8, 0.8, 0.9), 1.0)
    ));

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

    auto emat = make_shared<lambertian>(make_shared<image_texture>("earthmap.jpg"));
    world.add(make_shared<sphere>(point3(400,200,400), 100, emat));
    auto pertext = make_shared<noise_texture>(0.2);
    world.add(make_shared<sphere>(point3(220,280,300), 80, make_shared<lambertian>(pertext)));

    hittable_list boxes2;
    auto white = make_shared<lambertian>(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));
    }

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

    camera cam;

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

    cam.vfov     = 40;
    cam.lookfrom = point3(478, 278, -600);
    cam.lookat   = point3(278, 278, 0);
    cam.vup      = vec3(0,1,0);
```

```
    cam.defocus_angle = 0;

    cam.render(world);
}

int main() {
    switch (9) {
        case 1:  bouncing_spheres();          break;
        case 2:  checkered_spheres();         break;
        case 3:  earth();                     break;
        case 4:  perlin_spheres();            break;
        case 5:  quads();                     break;
        case 6:  simple_light();              break;
        case 7:  cornell_box();               break;
        case 8:  cornell_smoke();             break;
        case 9:  final_scene(800, 10000, 40); break;
        default: final_scene(400,   250,  4); break;
    }
}
```

Listing 74: `[main.cc]` *Final scene*

Running it with 10,000 rays per pixel (sweet dreams) yields:

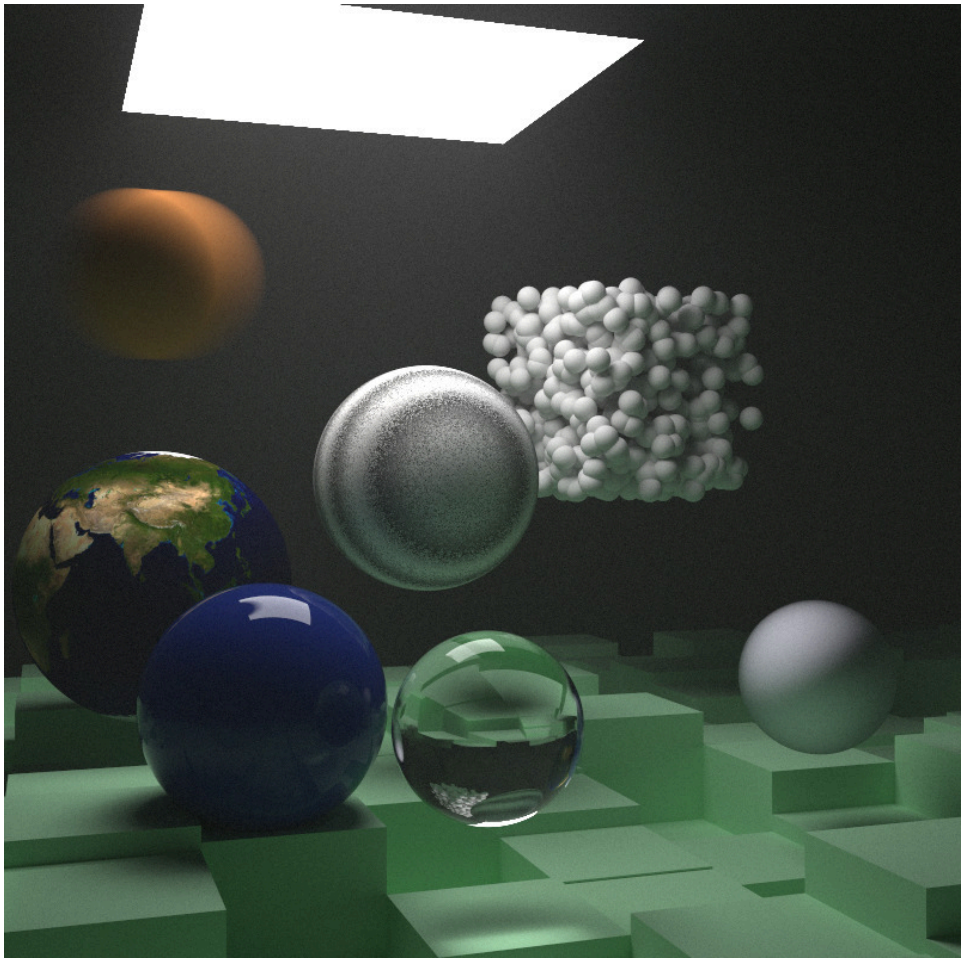

Image 23: *Final scene*

Now go off and make a really cool image of your own! See our Further Reading wiki page for additional project related resources. 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 Hollasch
Lorenzo Mancini
Ronald Wotzlaw

Corrections and Improvements

Aaryaman Vasishta
Andrew Kensler
Antonio Gamiz
Apoorva Joshi
Aras Pranckevičius
Arman Uguray
Becker
Ben Kerl
Benjamin Summerton
Bennett Hardwick
Benny Tsang
Dan Drummond
David Chambers
David Hart
Dimitry Ishenko
Dmitry Lomov
Eric Haines
Fabio Sancinetti
Filipe Scur
Frank He
Gareth Martin
Gerrit Wessendorf
Grue Debry
Gustaf Waldemarson
Ingo Wald
Jason Stone
JC-ProgJava
Jean Buckley
Jeff Smith
Joey Cho
John Kilpatrick
Kaan Eraslan
Lorenzo Mancini
Manas Kale
Marcus Ottosson
Mark Craig
Markus Boos
Matthew Heimlich
Nakata Daisuke
Nate Rupsis
Niccolò Tiezzi
Paul Melis
Phil Cristensen
LollipopFt
Ronald Wotzlaw
Shaun P. Lee
Shota Kawajiri
Tatsuya Ogawa
Thiago Ize
Thien Tran
Vahan Sosoyan
WANG Lei
Yann Herklotz
ZeHao Chen

Special Thanks

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

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

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

12. Citing This Book

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

12.1. Basic Data

- **Title (series)**: “Ray Tracing in One Weekend Series”
- **Title (book)**: “Ray Tracing: The Next Week”
- **Author**: Peter Shirley, Trevor David Black, Steve Hollasch
- **Version/Edition**: v4.0.2
- **Date**: 2025-04-25
- **URL (series)**: https://raytracing.github.io
- **URL (book)**: https://raytracing.github.io/books/raytracingthenextweek.html

12.2. Snippets

12.2.1 Markdown

```
[_Ray Tracing: The Next Week_](https://raytracing.github.io/books/RayTracingTheNextWeek.html)
```

12.2.2 HTML

```
<a href="https://raytracing.github.io/books/RayTracingTheNextWeek.html">
    <cite>Ray Tracing: The Next Week</cite>
</a>
```

12.2.3 LaTeX and BibTex

```
~\cite{Shirley2025RTW2}

@misc{Shirley2025RTW2,
   title = {Ray Tracing: The Next Week},
   author = {Peter Shirley, Trevor David Black, Steve Hollasch},
   year = {2025},
   month = {April},
   note = {\small \texttt{https://raytracing.github.io/books/RayTracingTheNextWeek.html}},
   url = {https://raytracing.github.io/books/RayTracingTheNextWeek.html}
}
```

12.2.4 BibLaTeX

```
\usepackage{biblatex}

~\cite{Shirley2025RTW2}

@online{Shirley2025RTW2,
   title = {Ray Tracing: The Next Week},
   author = {Peter Shirley, Trevor David Black, Steve Hollasch},
   year = {2025},
   month = {April},
   url = {https://raytracing.github.io/books/RayTracingTheNextWeek.html}
}
```

12.2.5 IEEE

```
“Ray Tracing: The Next Week.” raytracing.github.io/books/RayTracingTheNextWeek.html
(accessed MMM. DD, YYYY)
```

12.2.6 MLA

```
Ray Tracing: The Next Week. raytracing.github.io/books/RayTracingTheNextWeek.html
Accessed DD MMM. YYYY.
```