RAY TRACING IN ONE WEEKEND: THE BOOK SERIES

Notes by Yuxing Shi

https://raytracing.github.io/

Ray Tracing in One Weekend

https://raytracing.github.io/books/RayTracingInOneWeekend.html

Output an Image

- The PPM Image Format

PPM example [edit]

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

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

Image
(magnified)
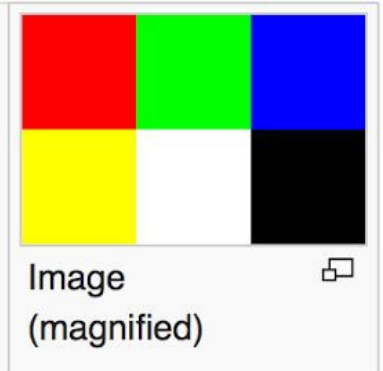
```cpp
// Render

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

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

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

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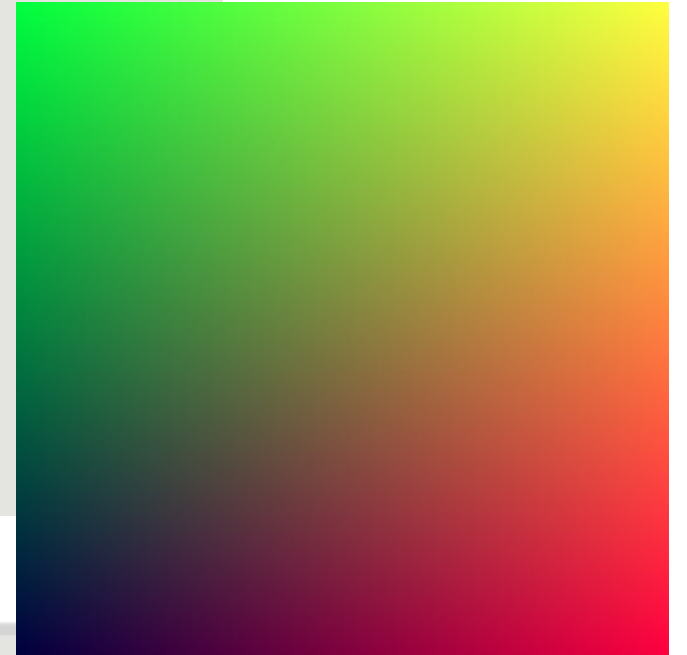
- Progress Indicator

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

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

Rays, a simple camera, and bg.

4.1. The ray Class

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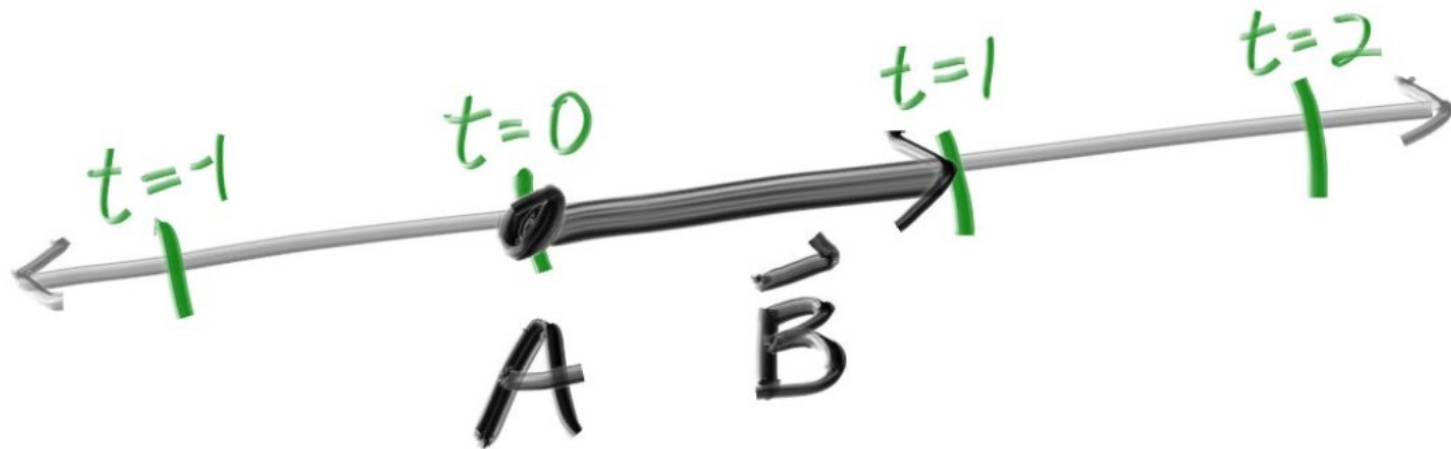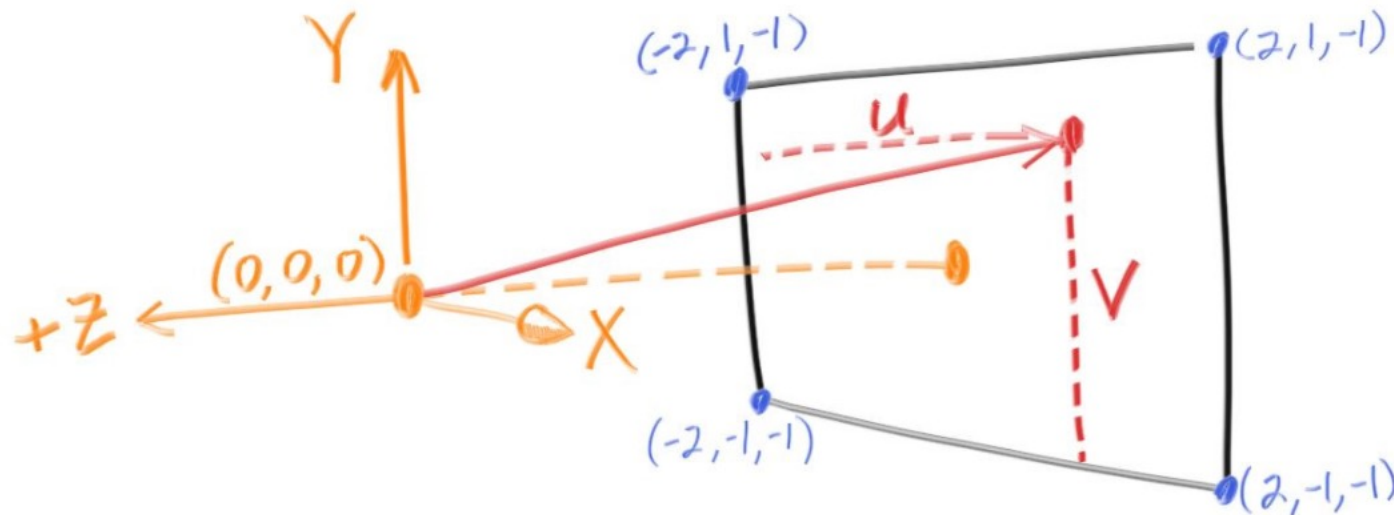

Figure 2: *Linear interpolation*

Sending Rays Into the Scene

- (1) calculate the ray from the eye to the pixel,
- (2) determine which objects the ray intersects, and
- (3) compute a color for that intersection point.

Figure 3: *Camera geometry*

Focal length = 1 unit (orange)
!= focus distance

Codes

- Viewpoint
  - through which to pass our scene rays
  - aspect ratio should be the same as our rendered image
  - Rectangle height of 2 at last slide.
  - Focal length: projection point to projection plane.

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

- linearly blends white and blue depending on the height of the y coordinate after scaling the ray direction to unit length (-1 ~ 1)

- And then scale 0 <= t <= 1.0

Codes

- Image + Camera (+ Render)

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

// Camera

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

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

Codes

- (Image + Camera +) Render

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

```cpp
// Render

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

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

Adding a Sphere

Image 3: A simple red sphere

- Sphere: $(x-Cx)\text{^}2+(y-Cy)\text{^}2+(z-Cz)\text{^}2=r\text{^}2$

- Vector form: $(\mathbf{P}-\mathbf{C})\cdot(\mathbf{P}-\mathbf{C})=r\text{^}2$

- Ray hits sphere: $(\mathbf{P}(t)-\mathbf{C})\cdot(\mathbf{P}(t)-\mathbf{C})=r2$
  Where: $\mathbf{P}(t)=\mathbf{A}+t\mathbf{b}$

- **=>** $t\text{^}2\mathbf{b}\cdot\mathbf{b}+2t\mathbf{b}\cdot(\mathbf{A}-\mathbf{C})+(\mathbf{A}-\mathbf{C})\cdot(\mathbf{A}-\mathbf{C})-r\text{^}2=0$
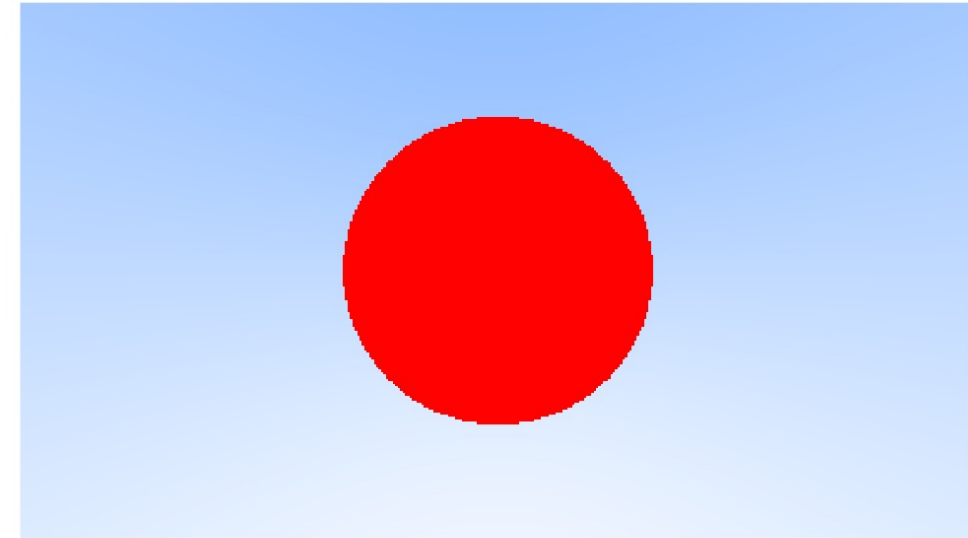
```cpp
bool hit_sphere(const point3& center, double radius, const ray& r) {
    return (discriminant > 0);
}
```

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

Surface Normals and Multiple Objects

- Surface normal: vector that is perpendicular to the surface at the point of intersection

- Code: configure hit_sphere to return smalltes t.

Figure 5: Sphere surface-normal geometry

Image 4: A sphere colored according to its normal vectors
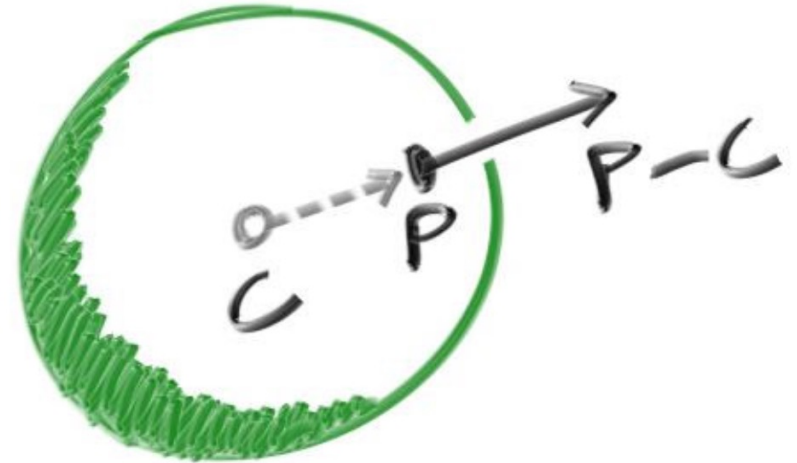
An Abstraction for Hittable Objects

- Object, surface, hittable

- Struct hit_record: P, N, t
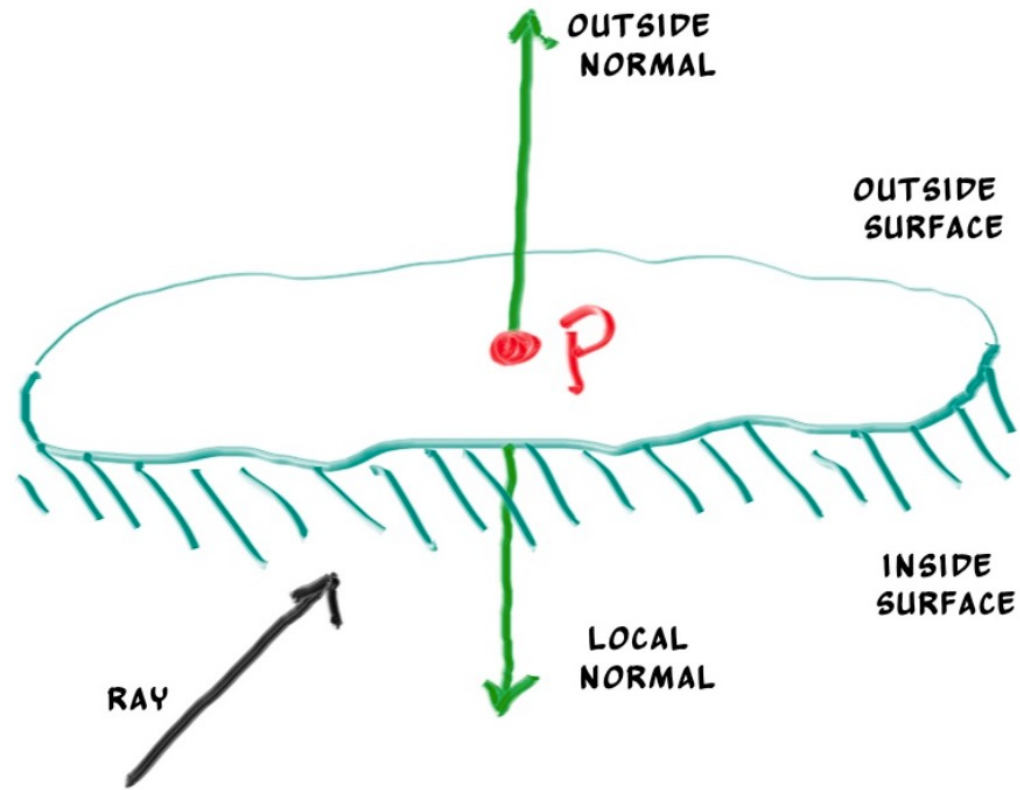
- Normal directions
  - Always out
  - Against ray

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

Add hittable_list

```cpp
Vector3 ray_color(Ray r, hittable_list world) {
    hit_record rec;
    if (world.hit(r, 0, infinity, rec)) {
        return 0.5 * (rec.N + Vector3(1,1,1));
        std::cout<<"hit"<<std::endl;
    }
    Vector3 unit_direction = r.direction().unit();
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*Vector3(1.0, 1.0, 1.0) + t*Vector3(0.5, 0.7, 1.0);
}
```

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

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

Antialiasing

- Real camera: no jaggies along edges because the edge pixels are a blend of some foreground and some background.

- => averaging a bunch of samples inside each pixel

- Canonical random number $0 \le r < 1$

- => use rand() in <cstdlib> returns a random integer 0, RAND_MAX

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

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

Add camera class

```cpp
class camera {
private:
    Vector3 _origin;
    Vector3 _lower_left_corner;
    Vector3 _horizontal;
    Vector3 _vertical;

public:
    camera();
    Ray get_ray(double u, double v);
};
```

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

- Write_color: add the full color each iteration, and then perform a single divide at the end (by the number of samples) when writing out the color

Add samples

```cpp
const int image_height = static_cast<in
const int samples_per_pixel = 100;

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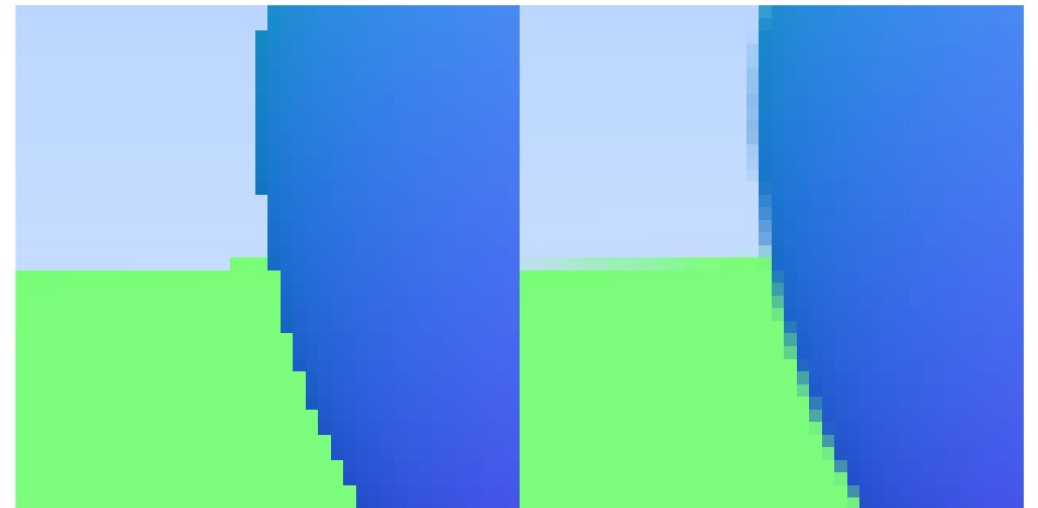

Image 6: Before and after antialiasing

Diffuse Materials

- take on the color of their surroundings, but they modulate that with their own intrinsic color

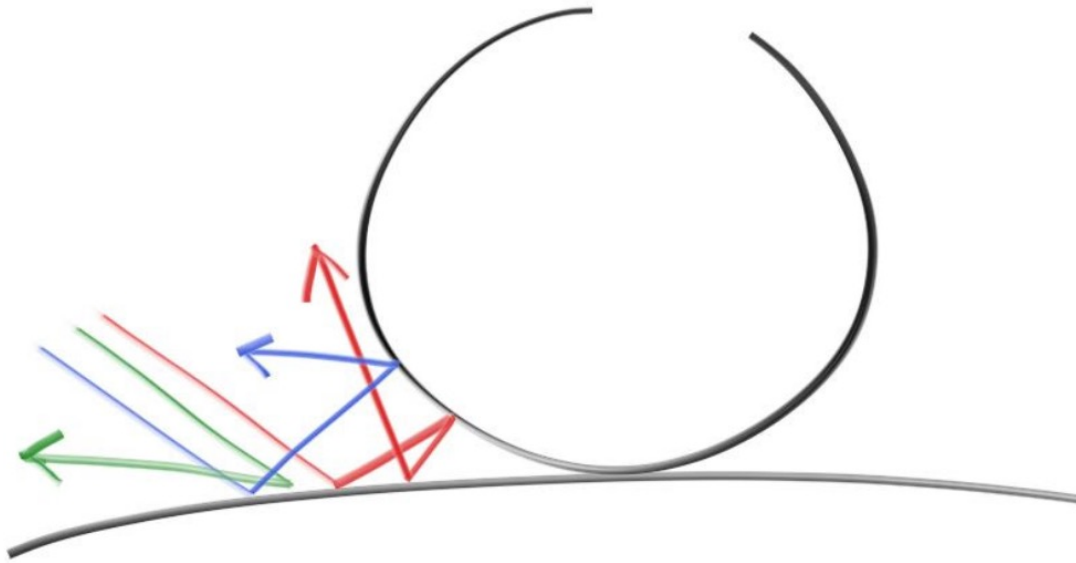- Light that reflects off a diffuse surface has its direction randomized

Figure 8: *Light ray bounces*

Diffuse Materials

- Also absorbed. The darker the surface, the more likely absorption is.
- algorithm randomizes direction will produce surfaces that look matte
- The sphere with a center at ($\mathbf{P}-\mathbf{n}$) is considered

inside the surface, whereas the sphere with

center ($\mathbf{P}+\mathbf{n}$) is considered *outside* the surface.

- Select radius on the same side as the ray origin
- Pick random **S** inside this unit radius.
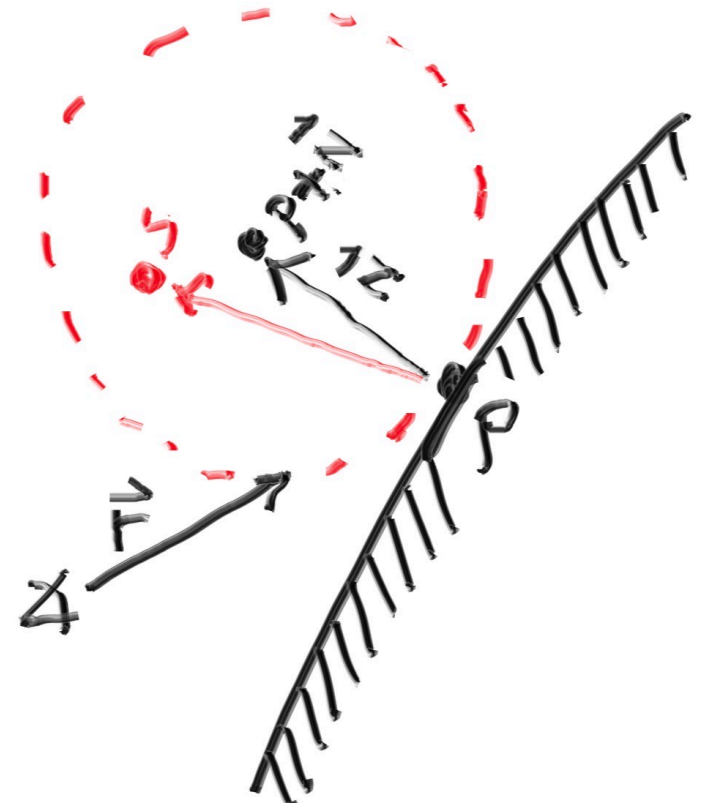
Pick random S.

- Rejection method:
- First, pick a random point in the unit cube where x, y, and z all range from −1 to +1. Reject this point and try again if the point is outside the sphere.

```cpp
static Vector3 random();
static Vector3 random(double min, double max);

Vector3 random_in_unit_sphere();
```

Recursive ray_color

- Limit the number of child rays by depth.

```cpp
Vector3 ray_color(Ray r, hittable_list world, int depth) {
    hit_record rec;

    if (depth <= 0) return Vector3(0, 0, 0);

    if (world.hit(r, 0, infinity, rec)) {
        Vector3 target = rec.P + rec.N + Vector3::random_in_unit_sphere();
        return 0.5 * ray_color(Ray(rec.P, target - rec.P), world, depth - 1);
    }
}
```
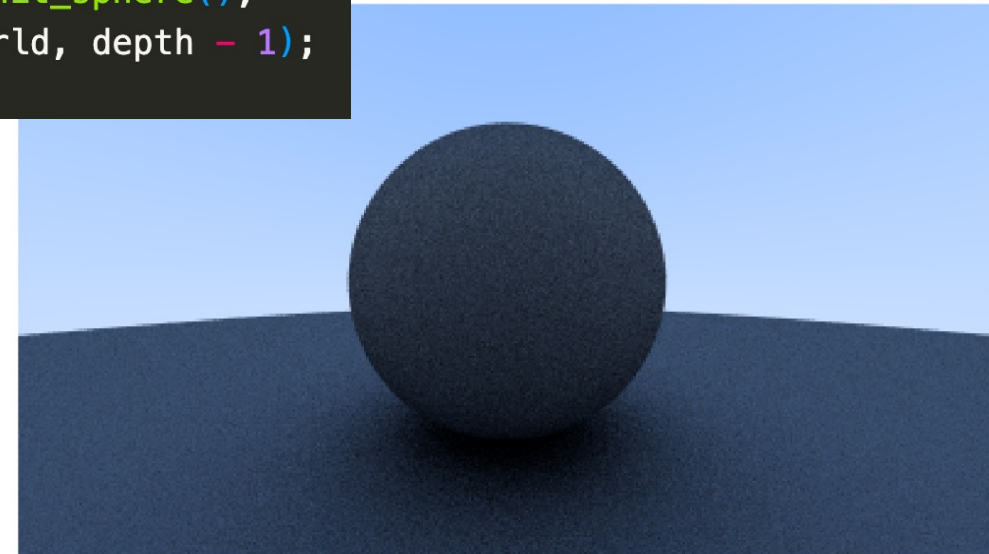

Image 7: First render of a diffuse sphere

Gamma correction for accurate color intensity

- meaning the 0 to 1 values have some transform before being stored as a byte

- use "gamma 2" which means raising the color to the power $1/gamma$, or in our simple case ½, which is just square-root:

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

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

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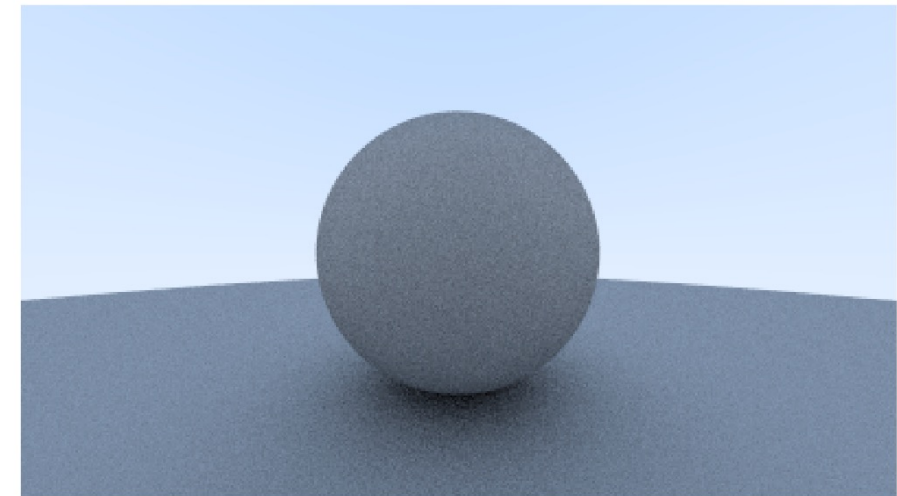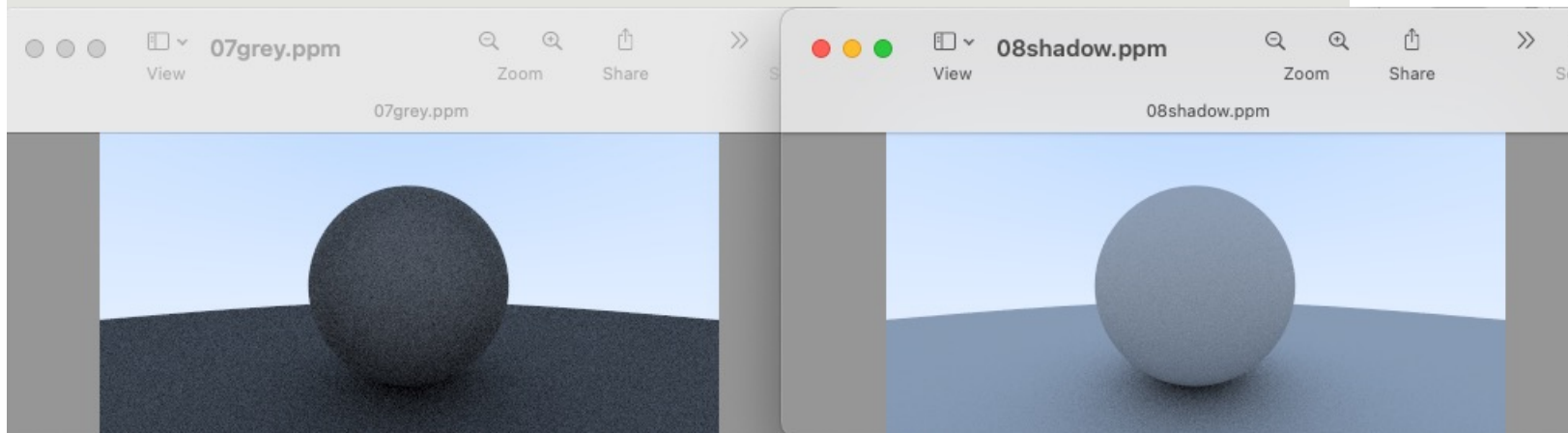

Image 8: Diffuse sphere, with gamma correction

Shadow acne problem

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

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

True Lambertian Reflection

- Rejection method: scales by the cos3^(ϕ) where ϕ is the angle from the normal. This is useful since light arriving at shallow angles spreads over a larger area, and thus has a lower contribution to the final color.

- Lambertian distribution: distribution of cos(ϕ).

has the probability higher for ray scattering close

to the normal, the distribution is more uniform.

picking random on the surface of the unit sphere,

offset along the surface normal.

```cpp
Vector3 Vector3::random_unit_sphere() {
    // on sphere S
    return Vector3::random_in_unit_sphere().unit();
}
```
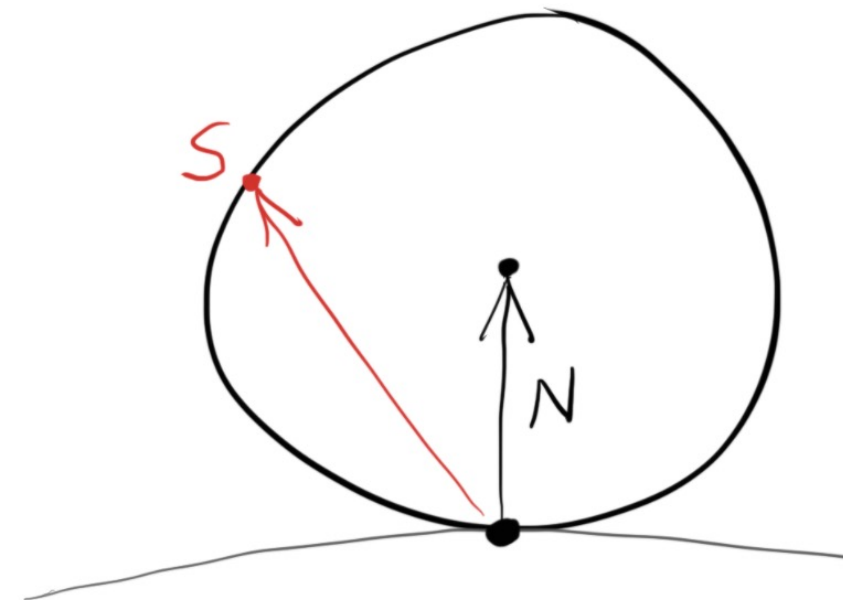
Figure 10: *Generating a random unit vector*

Differences

- The shadows are less pronounced after the change

- Both spheres are lighter in appearance after the change

<= due to the more uniform scattering of the light rays, fewer rays are scattering toward the normal, appear *lighter* because more light bounces toward the camera. the shadows less light bounces straight-up
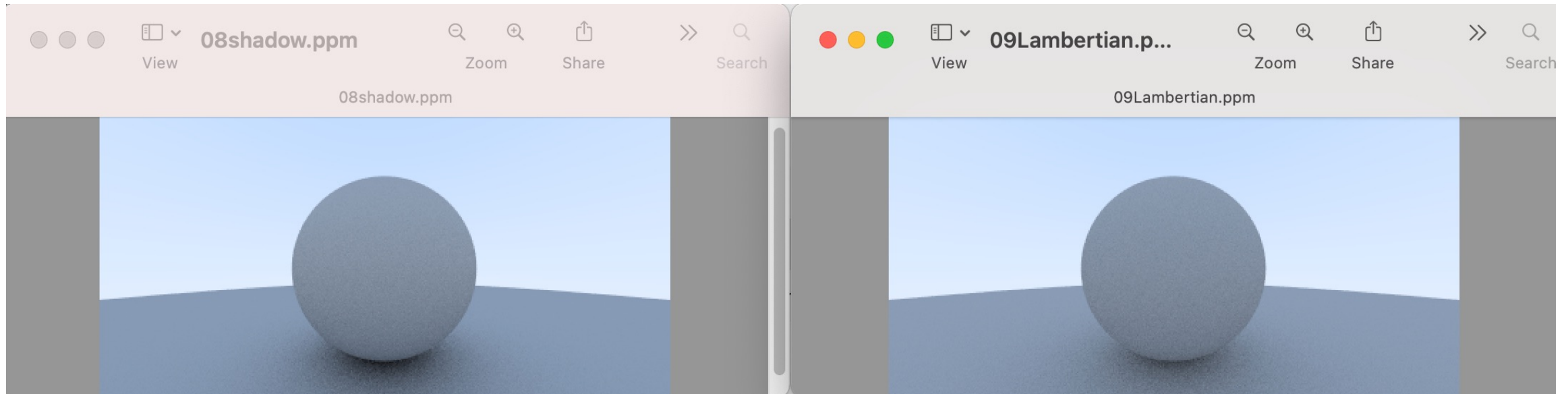
Abstract class for materials

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

```cpp
#ifndef MATERIAL_H
#define MATERIAL_H

#include "rtweekend.h"

struct hit_record;

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

#endif
```

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

Material and hittable

- Hittables and materials need to know each other so there is some circularity of the references. In C++ you just need to alert the compiler that the pointer is to a class, which the "class material" in the hittable class below does:

```
#include "rtweekend.h"

class material;

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

- the material pointer in the hit_record will be set to point at the material pointer, when hits.
- Add `shared_ptr<material> mat_ptr` to Sphere.

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

Modeling Light Scatter and Reflectance

scatter with some probability p and have attenuation be $albedo/p$.

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

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

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

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

    public:
        color albedo;
};
```

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

Mirrored Light Reflection

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

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

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

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

    public:
        color albedo;
};
```
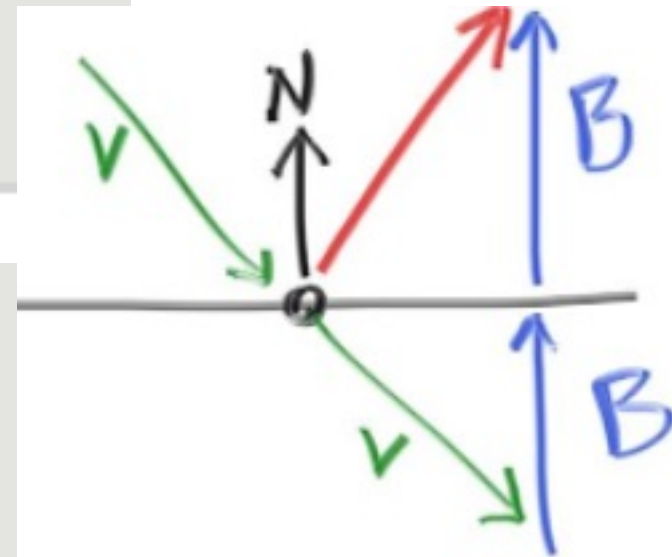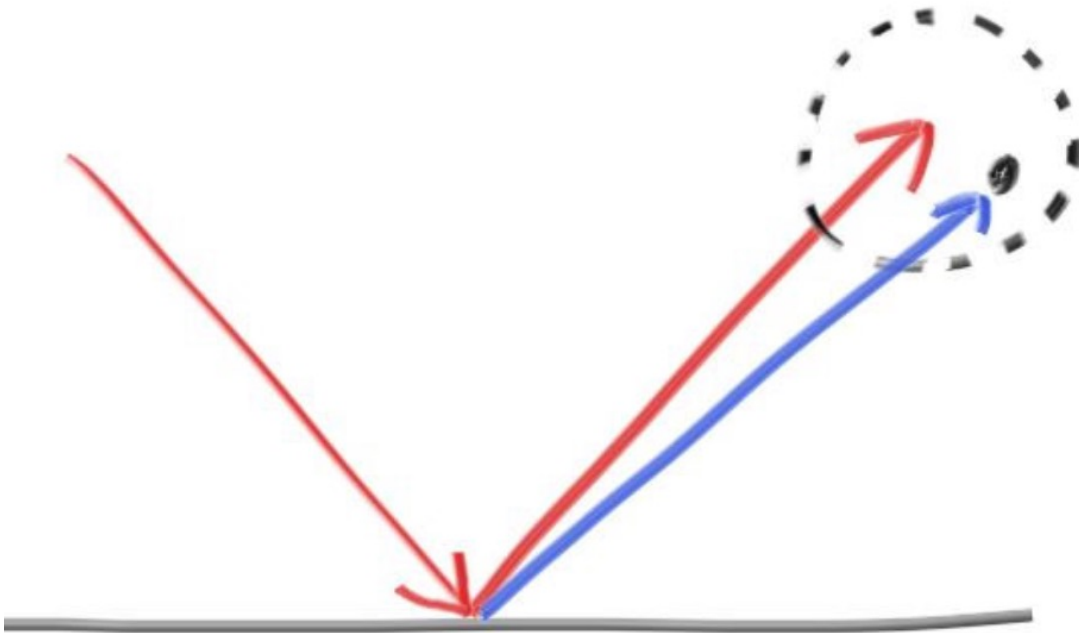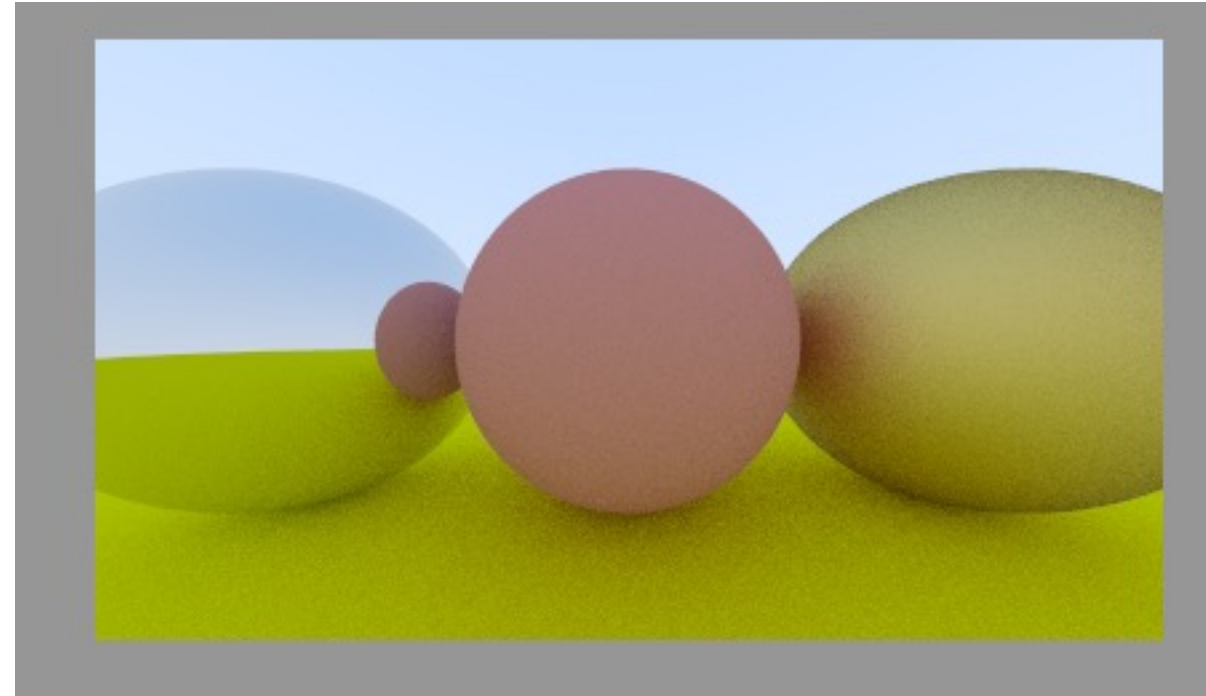
Fuzz Mirror

Figure 12: *Generating fuzzed reflection rays*

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

Fuzz = 0 & 1.