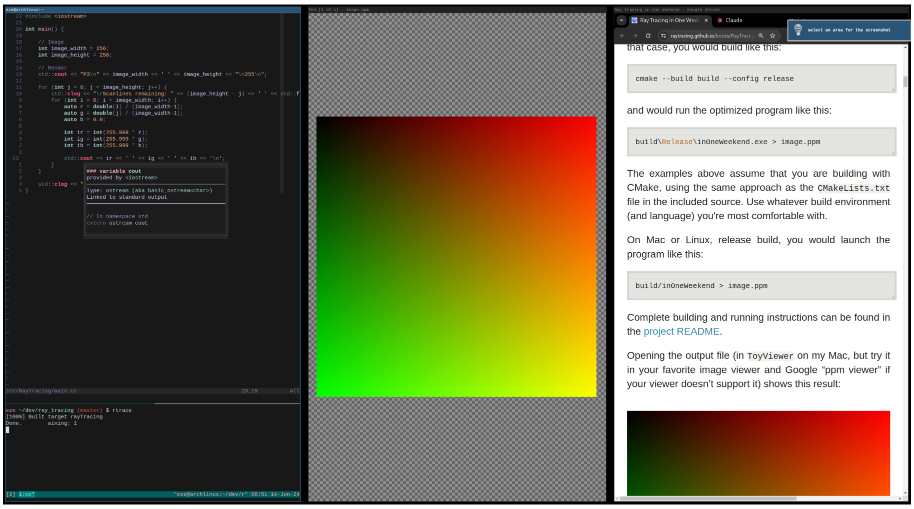
Ray Tracing One Weekend

Ian Turner

June 16, 2024

1 Ray Tracing in One Weekend

- this is my take on Marc Andreessen's anti-todo[1] list concept that i have been doing for years no matter the nature of the project or type of work i am doing (i also maintain a bite-sized version in a series of tiny moleskin)
- bib working lets go (broken on mac rn haha)
- now that bib is working, thanks to Peter Shirley, Trevor David Black, and Steve Hollasch for this incredible writeup! (course link)
- shoutout to @ludwigABAP for poasting this course (shoutout ml btw (for you page))
- this is my very first c or cpp project (op says it's c flavored cpp) beyond hello worlds and basic basic robotics stuff
- i love rust but i am not cracked at all so i would probably not be able to follow along in rust
- i will, however, follow op's advice to not copy pasta (besides most of makefile compiler flags hehehe) and build it up slowly by typing along
- going to try my best to thug this out by Sunday
- important setup for fresh arch install (not in order, and just off the dome, i likely am forgetting tons of things)
  - install unzip (will need for nvim clangd Mason lsp stuff)
  - install cmake, clangd, gcc stuff
  - setup debugger for nvim using dap, dap-ui, etc.
  - **build, compile, run:** (i think lol)
    1. `cmake -B build/Debug -DCMAKE_BUILD_TYPE=Debug`
    2. `cmake --build build/Debug`
    3. `build/rayTracing > image.ppm`
- op claims that if we can build project correctly in the beginning, then we are golden for rest of tutorial
- the provided cmake file is cash money and really easy to get working with my setup (Figure 1)



eze@archlinux:~
feh [1 of 1] - image.ppm
Ray Tracing in One Weekend - Google Chrome
Ray Tracing in One Week
Claude
raytracing.github.io/books/RayTraci...
select an area for the screenshot

```
#include <iostream>

int main() {

    // Image
    int image_width = 256;
    int image_height = 256;

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

    for (int j = 0; j < image_height; j++) {
        std::clog << "\rScanlines remaining: " << (image_height - j) << ' ' << std::f
        for (int i = 0; i < image_width; i++) {
            auto r = double(i) / (image_width-1);
            auto g = double(j) / (image_width-1);
            auto b = 0.0;

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

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

    std::clog << "
}
```

```
### variable cout
provided by <iostream>
Type: ostream (aka basic_ostream<char>)
Linked to standard output

// In namespace std
extern ostream cout
```

```
src/RayTracing/main.cc                                23,19        All

eze ~/dev/ray_tracing (master) $ rtrace
[100%] Built target rayTracing
Done.        aining: 1

[2] 1:cc*                           "eze@archlinux:~/dev/r" 06:51 14-Jun-24
```

that case, you would build like this:

```
cmake --build build --config release
```

and would run the optimized program like this:

```
build\Release\inOneWeekend.exe > image.ppm
```

The examples above assume that you are building with CMake, using the same approach as the `CMakeLists.txt` file in the included source. Use whatever build environment (and language) you're most comfortable with.

On Mac or Linux, release build, you would launch the program like this:

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

Complete building and running instructions can be found in the project README.

Opening the output file (in `ToyViewer` on my Mac, but try it in your favorite image viewer and Google "ppm viewer" if your viewer doesn't support it) shows this result:

Figure 1: *build test*

- make `rtrace` aliase for build, compile, run, then open image in feh, probably terrible idea but whatever
- got color header file with a `write_color` util function
- now working on a `ray` class which will use our `vec3` class.
- refresher on rays: think of them as functions (Equation 1):

$$\mathbf{P}(t) = \mathbf{A} + t\mathbf{b} \tag{1}$$

- 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 $\mathbf{P}(t)$ 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 a ray. (Figure 2)

[1] thanks pmarca! original blog post (archived by someone)

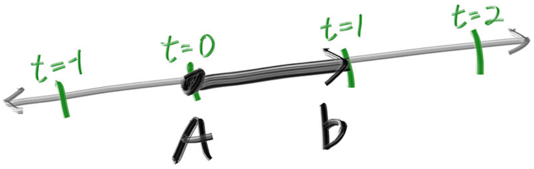


Figure 2: *linear interpolation*

- to make the actual ray tracer we will make simple camera with 16:9 aspect ratio since it will be easier to debug x and y transpositions.
- we need the height to be at least 1 since we divide width by height since it's easier to set the aspect ratio to width then divide by height. e.g. $width/height = 16/9 = 1.7778$
- apparently this is just an *optimistic* (my words) ratio since these values are not integers. we approximate the aspect ratio as best we can by rounding height to the nearest integer (and don't allow it to be less than one)

- now we have a camera center in 3d space from which all the rays will originate (commonly referred to as the *eye point*). we initially set the distance between the viewport of the camera center point to be one unit. This is often referred to as the *focal length.*

- we will use *right-handed coordinates* (right hand rule gang) Figure 3

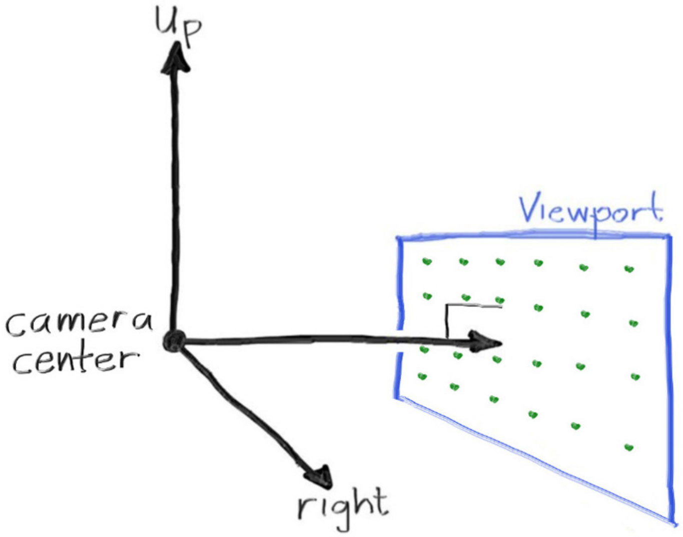


Figure 3: *camera geometry*

- unfortunately, the camera pose conflicts with the way we would like to render our image starting from the upper left pixel row by row scanning across from left to right. (Figure 4)

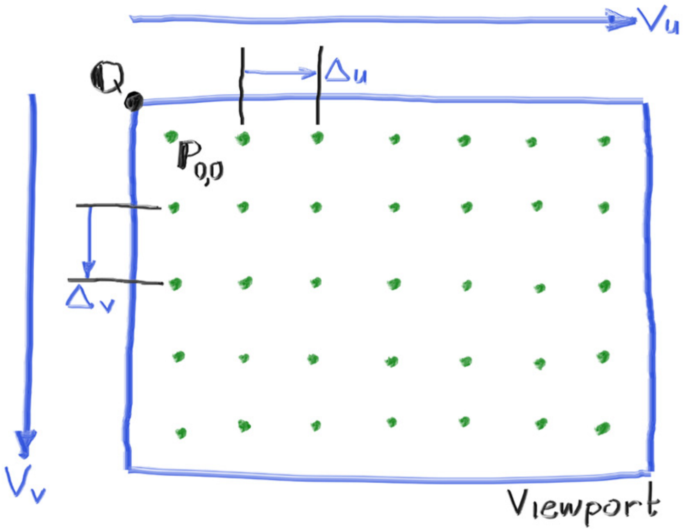


Figure 4: *viewport and pixel grid*

- we have example 7x5 resolution image, the viewport upper left corner $\mathbf{Q}$, the pixel $\mathbf{P_{0,0}}$ location, the viewport vector $\mathbf{V_u}$ (viewport_u), the viewport vector $\mathbf{V_v}$ (viewport_v), and the pixel delta vectors $\mathbf{\Delta u}$ and $\mathbf{\Delta v}$.

- now we add a simple gradient to the `ray_color` function which will linearly blend white and blue depending on the height of y coordinate $after$ scaling the ray direction to unit length. op uses standard graphics trick to linearly scale $0.0 \leq a \leq 1.0$. when $a = 1.0$, i want blue, when $a = 0.0$, i want white. in between, i want a blend. here's a *linear interpolation* or *linear interpolation*; commonly reffered to as a *lerp* between two values. a lerp is always of the form

$$blendedValue = (1 - a) \cdot startValue + a \cdot endValue \tag{2}$$

  with a going from zero to one (ayy lmao).

- now it's time to add a sphere (folks use spheres in ray tracers because calculating whether a ray hits a sphere is relatively simple.

- the equation for a radius r that is centered at the origin is an important mathematical equation

$$x^2 + y^2 + z^2 \tag{3}$$

- this of this as saying that if a given point (x, y, z) is on the surface of the sphere, then $x^2 + y^2 + z^2 = r^2$. if a given point (x, y, z) is *inside* the sphere, then $x^2 + y^2 + z^2 < r^2$, and if a given point (x, y, z) is *outside* the sphere, then $x^2 + y^2 + z^2 > r^2$.
- if we want to allow the sphere center to be at an arbitrary point (C_x, C_y, C_z), then the equation is not so nice:

$$(C_x - x)^2 + (C_y - y)^2 + (C_z - z)^2 = r^2 \tag{4}$$

- in graphics, you almost always want formulas to be in terms of vectors so we don't need to write out so many terms.
- note that the vector from point $\mathbf{P} = (x, y, z)$ to center $\mathbf{C} = (C_x, C_y, C_z)$ is $(\mathbf{C} - \mathbf{P})$
- we can ust the definition of the dot product:

$$(\mathbf{C} - \mathbf{P}) \cdot (\mathbf{C} - \mathbf{P}) = (C_x - x)^2 + (C_y - y)^2 + (C_z - z)^2 = r^2 \tag{5}$$

- then we can rewrite the equation of the sphere in vector form as:

$$(\mathbf{C} - \mathbf{P}) \cdot (\mathbf{C} - \mathbf{P}) = r^2 \tag{6}$$

- we can read this as "any point $\mathbf{P}$ that satisfies this equation is on the sphere". we want to know if our ray $\mathbf{P}(t) = \mathbf{Q} + t\mathbf{d}$ ever hits the sphere anywhere. if it does hit the sphere, there is some t for which $\mathbf{P}(t)$ satisfies the sphere equation. So we are looking for any t where this is true:

$$(\mathbf{C} - \mathbf{P}(t)) \cdot (\mathbf{C} - \mathbf{P}(t)) = r^2 \tag{7}$$

- which can be found by replacing $\mathbf{P}(t)$ with its expanded form:

$$(\mathbf{C} - (\mathbf{Q} + t\mathbf{d})) \cdot (\mathbf{C} - (\mathbf{Q} + t\mathbf{d})) = r^2 \tag{8}$$

- we have three vecs on the left dotted by three vecs on right, if we solved for the full dot product we would get nine vectors (slop), we need to solve for t with quadratic equation, so first shape equation above into quadratic by first isolating t terms, then distribute dot product, then move r^2 to left hand side:
- boom

$$(-t\mathbf{d} + (\mathbf{C} - \mathbf{Q})) \cdot (-t\mathbf{d} + (\mathbf{C} - \mathbf{Q})) = r^2 \tag{9}$$

- bop

$$t^2\mathbf{d} \cdot \mathbf{d} - 2t\mathbf{d} \cdot (\mathbf{C} - \mathbf{Q}) + (\mathbf{C} - \mathbf{Q}) \cdot (\mathbf{C} - \mathbf{Q}) = r^2 \tag{10}$$

- pow

$$t^2\mathbf{d} \cdot \mathbf{d} - 2t\mathbf{d} \cdot (\mathbf{C} - \mathbf{Q}) + (\mathbf{C} - \mathbf{Q}) \cdot (\mathbf{C} - \mathbf{Q}) - r^2 = 0 \tag{11}$$

- we can now run quadratic formula on this bad boy and all the vecs are reduced to scalars by dot prod.

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{12}$$

- can now for t using the terms a, b, and c:

$$a = \mathbf{d} \cdot \mathbf{d} \tag{13}$$

$$b = -2\mathbf{d} \cdot (\mathbf{C} - \mathbf{C}) \tag{14}$$

$$c = (\mathbf{C} - \mathbf{Q}) \cdot (\mathbf{C} - \mathbf{Q}) - r^2 \tag{15}$$

- this will yield either positive (2 real solutions), negative (no real solutions) or zero (1 real solution)

- the algebra almost always relates very directly to the geometry (Figure 5)

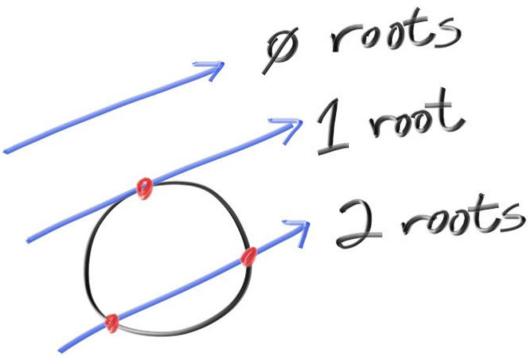
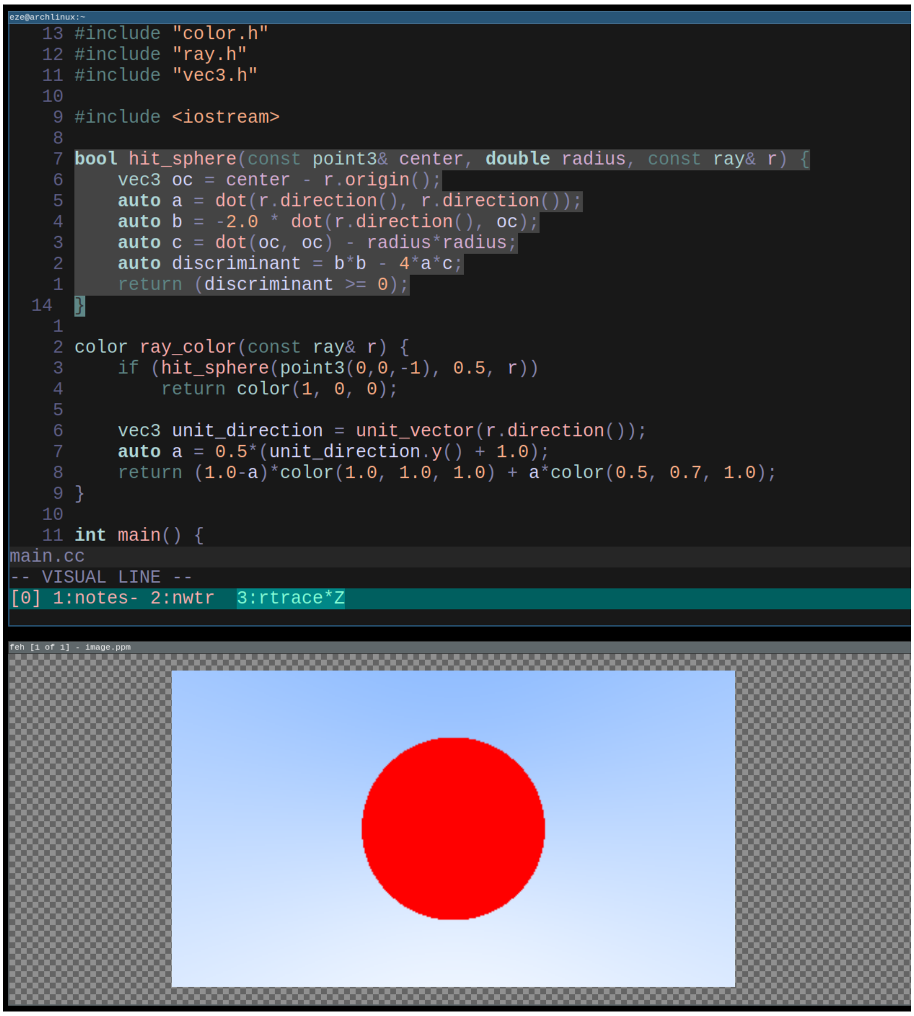


Figure 5: *ray sphere intersection roots*

- create first ray traced image by placing a small sphere at -1 on the z-axis and then coloring red any pixel that intersects it. (Figure 6)



```
#include "color.h"
#include "ray.h"
#include "vec3.h"

#include <iostream>

bool hit_sphere(const point3& center, double radius, const ray& r) {
    vec3 oc = center - r.origin();
    auto a = dot(r.direction(), r.direction());
    auto b = -2.0 * dot(r.direction(), oc);
    auto c = dot(oc, oc) - radius*radius;
    auto discriminant = b*b - 4*a*c;
    return (discriminant >= 0);
}

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 a = 0.5*(unit_direction.y() + 1.0);
    return (1.0-a)*color(1.0, 1.0, 1.0) + a*color(0.5, 0.7, 1.0);
}

int main() {
main.cc
-- VISUAL LINE --
[0] 1:notes- 2:nwtr  3:rtrace*Z
```

Figure 6: *simple red sphere*

- this lacks all sorts like shading, reflection rays, and more than a single object, also this solution doesn't account for the camera really since it will also work with the sphere center at +1 (behind camera)

- i should start writing section titles... maybe later
- shading with surface normals, a normal is just perpendicular to the surface at the point of intersection.
- we have key design decision to make for normal vectors in code: whether normal vecs will have arbitrary length, or should we normalize? *much to think about...*
- it is tempting to skip the expensive sqrt op involved in normalizing the vector, in case it's not needed. in practice, however, there are three important observations. first, if a unit-length normal vector is *ever* required, then you might as well do it up from once, instead of over and over again "just in case" for every location where unit-length is required. Second, we *do* require unit-length normal vecors in several places. Third, if you require normal vectors to be unit length, then you can often efficiently generate that vec with an understanding of the specific geometry class, in its consructor, or in the `hit()` function. e.g., sphere normals can be made unit length simply by dividing by the sphere radius, avoiding the sqrt entirely.
- unit length sphere normals will be used up front for these reasons
- the outward normal for a sphere is in the direction of the hit point minus the center (Figure 7)

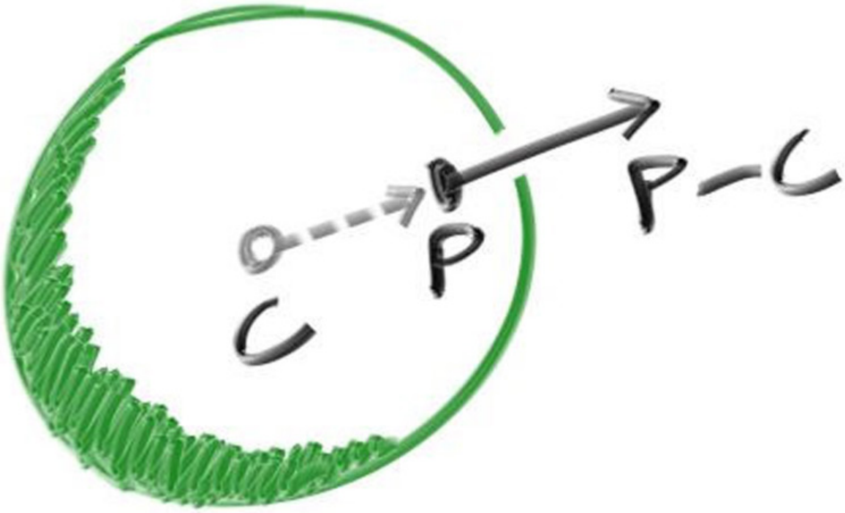
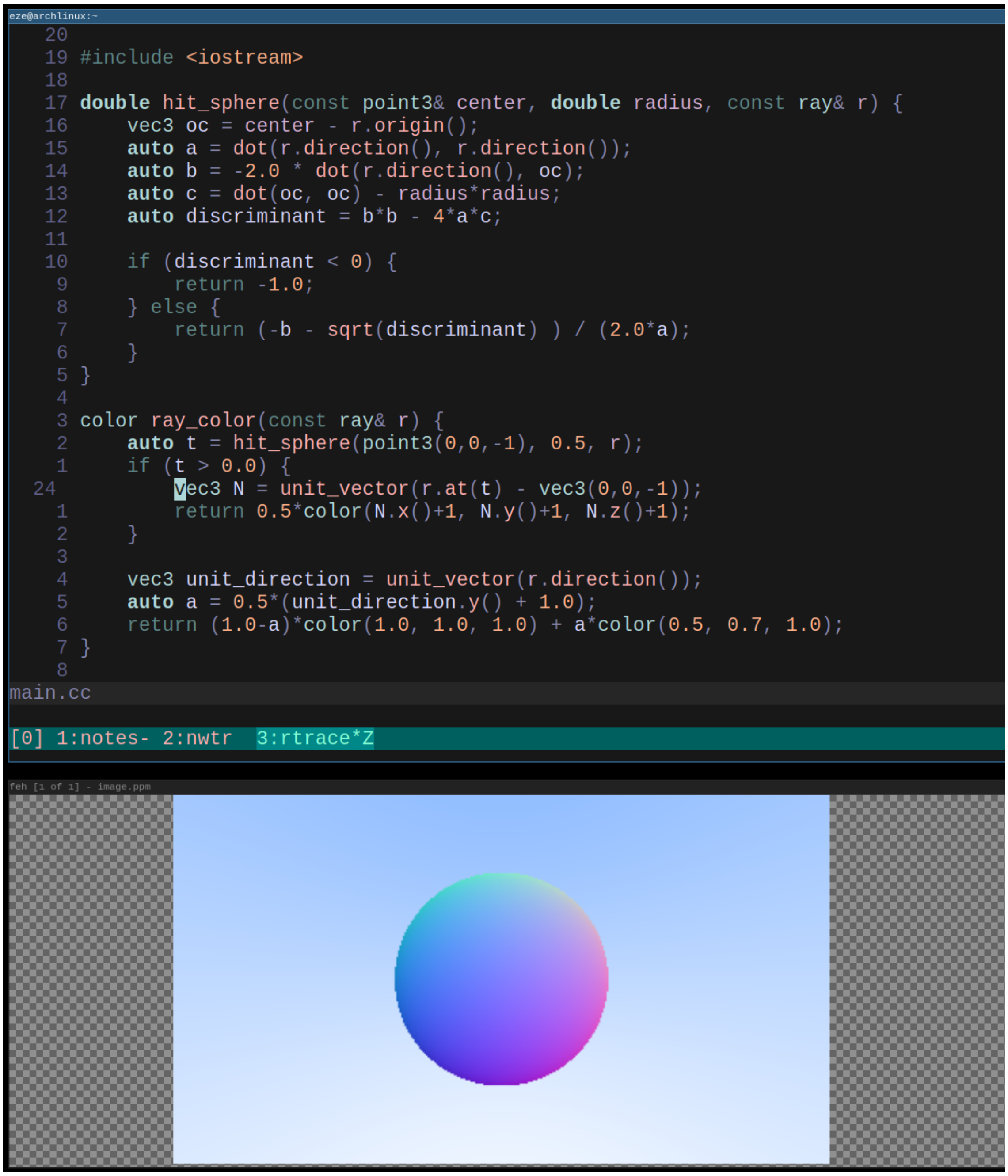


Figure 7: *sphere surface-normal geometry*

- we don't have light let so let's use a common trick for visualizing normals: a color map. we can assume **n** is a unit length vec, so each component is between -1 and 1; we just map eacho component to the interval fro m0 to 1, and then map (x, y, z) to $(red, green, blue)$. for the normal, we need the hit point, not just whether we hit or not (which is all we are doing currently: Figure 6). we only have 1 sphere in the schene and it's right in front of the camera, so we won't worry about negative values of t yet. We'll just assume the closest hit point (smallest t) is the one that we want. (Figure 8)



```
#include <iostream>

double hit_sphere(const point3& center, double radius, const ray& r) {
    vec3 oc = center - r.origin();
    auto a = dot(r.direction(), r.direction());
    auto b = -2.0 * dot(r.direction(), oc);
    auto c = dot(oc, oc) - radius*radius;
    auto discriminant = b*b - 4*a*c;

    if (discriminant < 0) {
        return -1.0;
    } else {
        return (-b - sqrt(discriminant) ) / (2.0*a);
    }
}

color ray_color(const ray& r) {
    auto t = hit_sphere(point3(0,0,-1), 0.5, r);
    if (t > 0.0) {
        vec3 N = unit_vector(r.at(t) - vec3(0,0,-1));
        return 0.5*color(N.x()+1, N.y()+1, N.z()+1);
    }

    vec3 unit_direction = unit_vector(r.direction());
    auto a = 0.5*(unit_direction.y() + 1.0);
    return (1.0-a)*color(1.0, 1.0, 1.0) + a*color(0.5, 0.7, 1.0);
}
```

Figure 8: *sphere colored according to its normal vectors*

- testing cpp listing style:

```cpp
double hit_sphere(const point3& center, double radius, const ray& r) {
  vec3 oc = center - r.origin();
  auto a = dot(r.direction(), r.direction());
  auto b = -2.0 * dot(r.direction(), oc);
  auto c = dot(oc, oc) - radius*radius;
  auto discriminant = b*b - 4*a*c;

  if (discriminant < 0) {
    return -1.0;
  } else {
    return (-b - sqrt(discriminant)) / (2.0*a);
  }
}
```

- first, recall that a vector dotted with itself is equal to the equared length of that vector.
- second, notice how the eq for `b` has a factor of negative two in it. consider what happens to the quadratic equation if $b = -2h$:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$= \frac{-(-2h) \pm \sqrt{(-2h)^2 - 4ac}}{2a}$$

$$= \frac{2h \pm 2\sqrt{h^2 - ac}}{2a}$$

$$= \frac{h \pm \sqrt{h^2 - ac}}{a}$$

- this simplifies nicely so solving for h:

$$b = -2\mathbf{d} \cdot (\mathbf{C} - \mathbf{Q})$$

$$b = -2h$$

$$h = \frac{b}{-2} = \mathbf{d} \cdot (\mathbf{C} - \mathbf{Q})$$

- using these observations, we can now simplify the sphere-intersection code to this:

```cpp
double hit_sphere(const point3& center, double radius, const ray& r) {
    vec3 oc = center - r.origin();
    auto a = r.direction().length_squared();
    auto h = dot(r.direction(), oc);
    auto c = oc.length_squared() - radius*radius;
    auto discriminant = h*h - a*c;

    if (discriminant < 0) {
        return -1.0;
    } else {
        return (-h - sqrt(discriminant) ) / a;
    }
}
```

- now how about more than one sphere? let's create a “hittable” class to abstract away the sphere creation logic.
-

- the second design decision for normals is whether they should always point out. at present, the normal found will always be in the direction of the center to the intersection point (the normal points out). if the ray intersects the sphere from the outside, the normal points against the ray. if the ray intersects the sphere from the inside, the normal (which always points out) points with the ray. alternatively, we can have the normal always point against the ray. if the ray is outside the sphere, the normal will point outward, but if the ray is inside the sphere, the normal will point inward. (Figure 9)

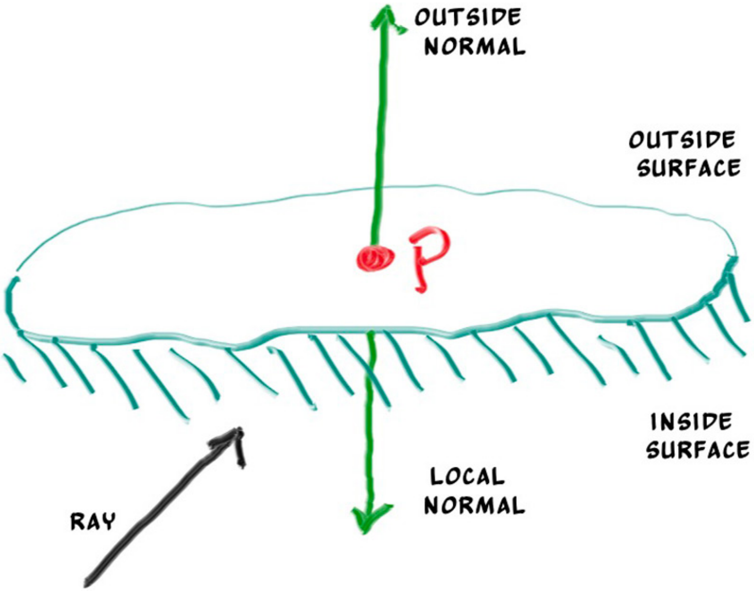


Figure 9: *possible directions for sphere surface-normal geometry*

- we need to choose one of these possibilities because we will eventually want to determine which side of the surface that the ray is coming from. this is important for objects that are rendered differently on each side, like the text on a two-sided sheet of paper, or for objects that have an inside and an outside, like glass balls.
- if we decide to have the normals always point out, then we will need to determine which side the ray is on when we color it. we can figure this out by comparing the ray with the normal. if the ray and the normal face in the same direction, the ray is inside the object, if the ray and the normal face in the opposite direction, then the ray is outside the object. this can be determined by taking the dot product of the two vectors, where if their dot is positive, the ray is inside the sphere.
-
- if we decide to have the normals always point against the ray, we won't be able to use the dot product to determine which side of the surface the ray is on. instead, we would need to store the info:
-
- we can set things up so that norms always point out from the surface, or always point against the incident ray. this decision is determined by whether you want to determine the side of the surface at the time of geometry intersection or at the time of coloring. in this book we have more material types than we have geometry types, so we'll go for less work and put the determination at geometry time. this is simply a matter of preference.
-
- the `hittable_list` class code uses two C++ features that may trip you up if you're not normally a C++ programmer: `vector` and `shared_ptr`.
- `shared_ptr<type>` is a pointer to some allocated type, with reference-counting semantics. every time you assign its value to another shared pointer (usually with a simple assignment), the reference count is incremented. as shared pointers go out of the scope (like at the end of a block or function), the reference count is decremented. once the count goes to zero, the object is safely deleted.
- typically, a shared pointer is first initialized with a newly-allocated object, something like this:
-
- make common header file `rtweekend.h`