

# Ray Tracing One Weekend

Ian Turner

June 15, 2024

# 1 Week 0: Ray Tracing Class

Thursday, 06/13/2023

- [course link](#)
- shoutout to **TODO** add x user that pointed me to this (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
- test bib [1]
- bib working lets go
- 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
- i hope this is true since the provided cmake file is cash money and really easy to get working with my setup (Figure ??)
- make `rtrace` alias 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):

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

- here  $\mathbf{P}$  is a 3D position along a line in 3D.  $\mathbf{A}$  is the ray origin and  $\mathbf{b}$  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  $\mathbf{A}$ , and this is what is often called a half-line or a ray. (Figure 1)

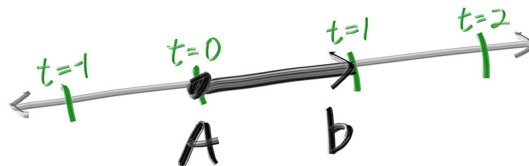


Figure 1: *linear interpolation*

- 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 2

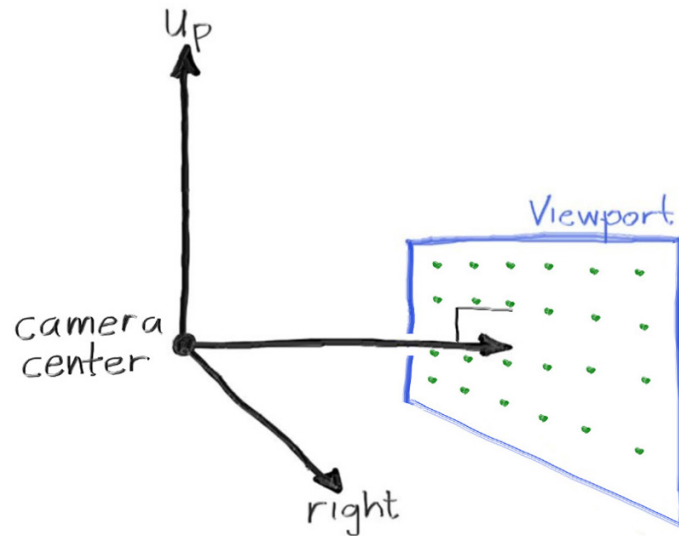


Figure 2: camera geometry

- 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 3)

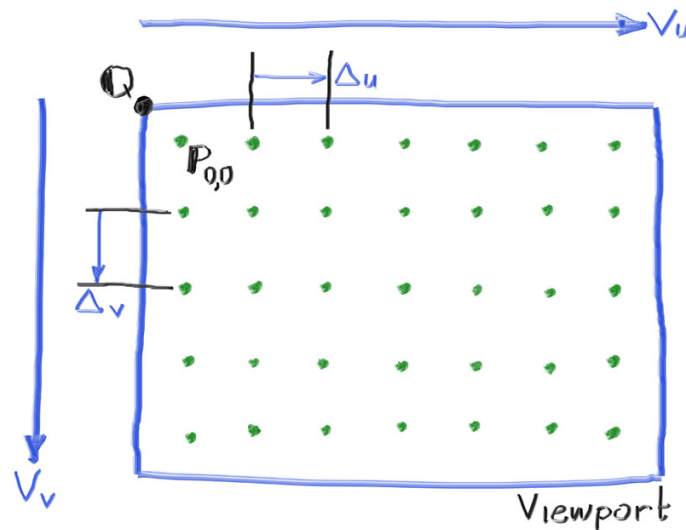


Figure 3: viewport and pixel grid

- we have example 7x5 resolution image, the viewport upper left corner  $Q$ , the pixel  $P_{0,0}$  location, the viewport vector  $V_u$  (viewport\_u), the viewport vector  $V_v$  (viewport\_v), and the pixel delta vectors  $\Delta u$  and  $\Delta v$ .
- 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 referred to as a *lerp* between two values. a lerp is always of the form

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

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

$$x^2 + y^2 + z^2 \quad (3)$$

- 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:

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

- 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 use the definition of the dot product:

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

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

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

- 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:

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

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

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

- 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

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

- bop

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

- pow

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

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

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (12)$$

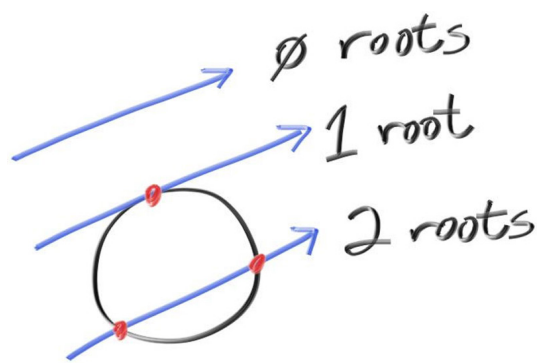
- can now for  $t$  using the terms  $a, b$ , and  $c$ :

$$a = \mathbf{d} \cdot \mathbf{d} \quad (13)$$

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

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

- 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 4)
- 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 4:** *ray sphere intersection roots*

## References

- [1] S. H. Peter Shirley, Trevor David Black, “Ray tracing in one weekend,” April 2024.