Graphics like Pixar using Swift

https://github.com/kapsy/swift_ray_tracer

@kapsy1312

Firstly, a movie

https://github.com/kapsy/swift_ray_tracer

Thank you. (For emcees)

Hi.

My name is *Michael*, and I work for a company called *ZOZO* Technologies.

Firstly, I want to show you a movie I made.

This was made with the *same techniques* used in Pixar's animated films.

It was created using mathematics, and Swift.

No hardware or graphics APIs were used.

This is a rendering technique, called *ray tracing*.

Simply put, ray tracing, throws a multitude of rays from a *receptor point*, our eyes, into a *virtual scene*.

We observe how the rays of light behave when they hit different surfaces, and display the resulting color on the screen.

Why ray trace in Swift?

- Well, it's *try!* Swift…
- Try something I have *never* done before in the language
- Try Swift as a *bare bones* language
- No Platform Specific APIs or hardware
- Purpose was *learning*

So why ray trace in Swift?

Well, it is *try!* Swift, and I wanted to try programming something I have never done in the language.

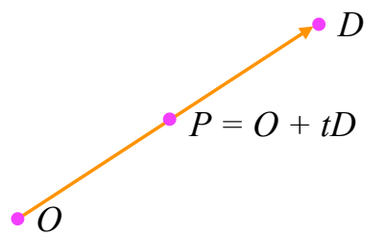
I wanted to see what Swift would be like as a *bare bones language*, rather than a medium for for calling Frameworks and system APIs.

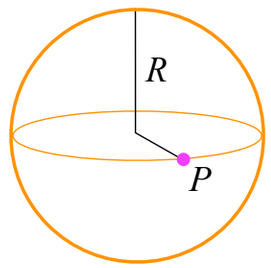
And, I focussed on building the ray tracer from *nothing*, in order to *really learn* how it works.

Ray tracing math in Swift

In the short time we have, I want to give you a taste of some *fundamental* ray tracing math, and how it would be done using Swift.

Line-sphere intersection





Equation for a sphere's surface: $P \cdot P = R^2$

Equation for a ray: $P = O + tD$

For instance, if we want to render one of the spheres in the movie, we *need to know* when our rays of light will *intersect* it.

We're shooting out thousands of rays of light into our scene, and we need to know which ones will hit the sphere.

That requires coming up with an *analytic solution* to two equations, one for the *point on a sphere*, and one for a *parametric ray*.

If we take the *dot product* of a point on the surface of a sphere, P, it will be *equal* to its radius squared, or R^2.

And a *parametric ray* is just an *origin* point O, a *direction* vector D, and a t parameter that tells us how far we travel along the ray.

We want to find t

Equation for a sphere's surface:	$P \cdot P = R^2$
Equation for a ray:	$P = O + tD$
So we are looking for any t where:	$(O + tD) \cdot (O + tD) - R^2 = 0$
We can rearrange into a quadratic:	$O^2 + t^2 D^2 + 2ODt - R^2 = 0$

So we want to find t.

For what t does our ray hit the surface of the sphere?

If we substitute P in each equation, the dot product of $O + tD$, minus R squared, should equal 0.

Working through the vector algebra, we can arrange this into a *quadratic*.

What's a quadratic?

A quadratic can have *one*, *two*, or *no* solutions.

$$ax^2 + bx + c = 0$$

We can find these solutions using the quadratic formula:

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

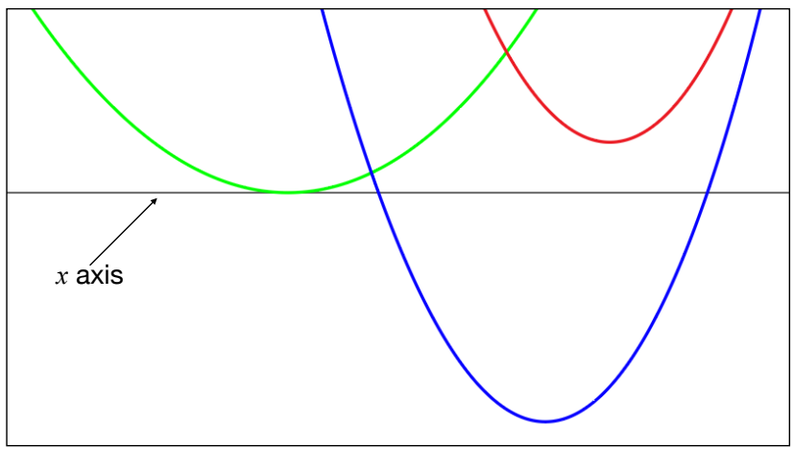
So what's a quadratic? A quadratic just follows the form $ax^2 + bx + c = 0$.

It can have *one*, *two*, or *no*, solutions.

A *solution* is any value of x where the quadratic *equals zero*.

We can find these *solutions* using the *quadratic formula*, which I won't cover here.

Geometrically explained



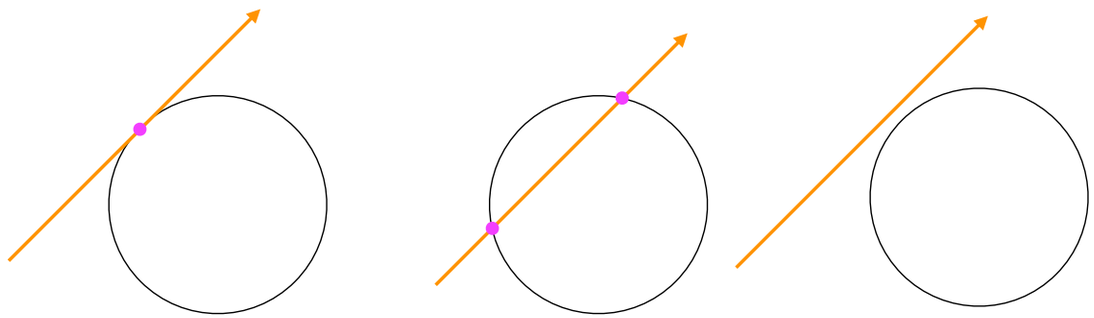
$$ax^2 + bx + c = 0$$

We can also look at a quadratic *geometrically*, as a *parabola* shape.

And any time the parabola crosses the x axis, we have a *root*, or *solution*, to the quadratic equation.

So now we can *clearly see* how our quadratic can have *one*, *two*, or *no* solutions.

As a sphere and ray



We can have *one*, *two*, or *no* intersections.

In terms of a ray, we can also have *one*, *two*, or *no* intersections with our sphere.

The ray can touch the surface of the sphere *exactly*, it can intersect *right through it*, or *not touch at all*.

So the math of a quadratic models what is happening here.

In Swift…

```
let oc = ray.origin - sphere.center
// Our quadratic constants
let a = dot(ray.dir, ray.dir)
let b = dot(oc, ray.dir)
let c = dot(oc, oc) - sphere.radius*sphere.radius
// Obtain discriminant and check sign
let discriminant = b*b - a*c;
if discriminant > 0.0 {
    var t = (-b - sqrt(discriminant))/a
    if (tnear < t && t < tfar) {
        P = t*ray.dir // Intersection, find P using ray equation
        tfar = t
    }
    t = (-b + sqrt(discriminant))/a
    if (tnear < t && t < tfar) {
        P = t*ray.dir // Intersection, find P using ray equation
        tfar = t
    }
}
```

So time for some Swift.

Firstly, we obtain our quadratic constants, a, b, and c, from the origin and direction of the ray, and the radius and center of the sphere.

We then use those to work out something called the *discriminant*.

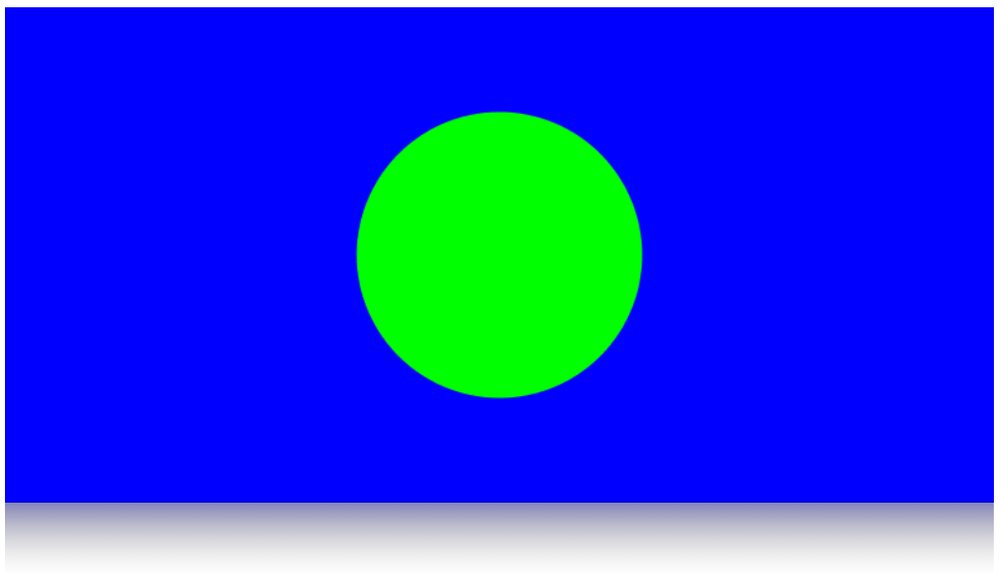
The *discriminant* is just a part of the *quadratic formula* that tells us how many solutions we have.

If it's *greater than zero*, then we have one or two intersections, and we use the *quadratic formula* to find the t parameter for the ray.

We then perform a distance check, and use the *ray equation* to find the point of the intersection.

So how does it look?

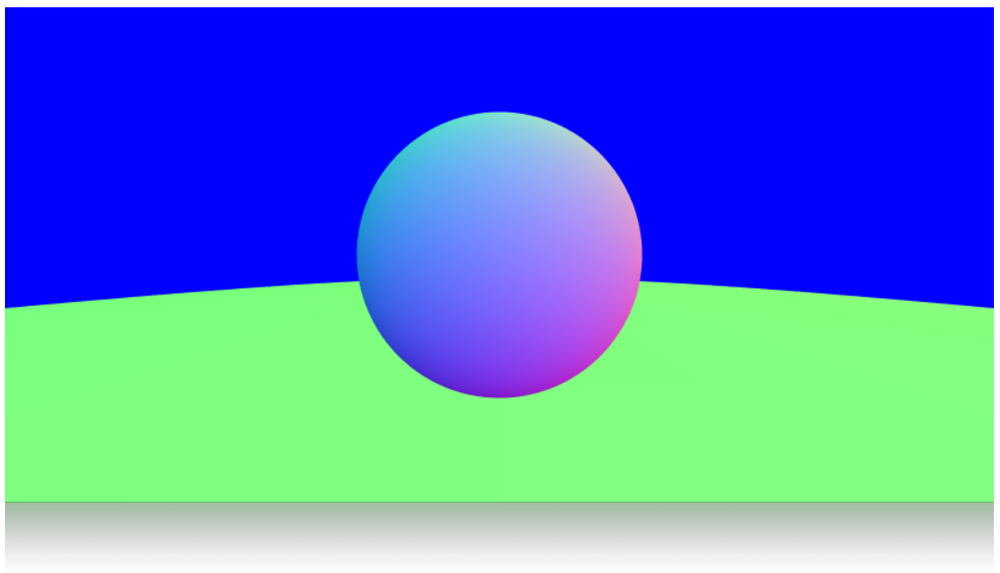
How does it look?



Ok, so we can’t exactly tell it’s a sphere.

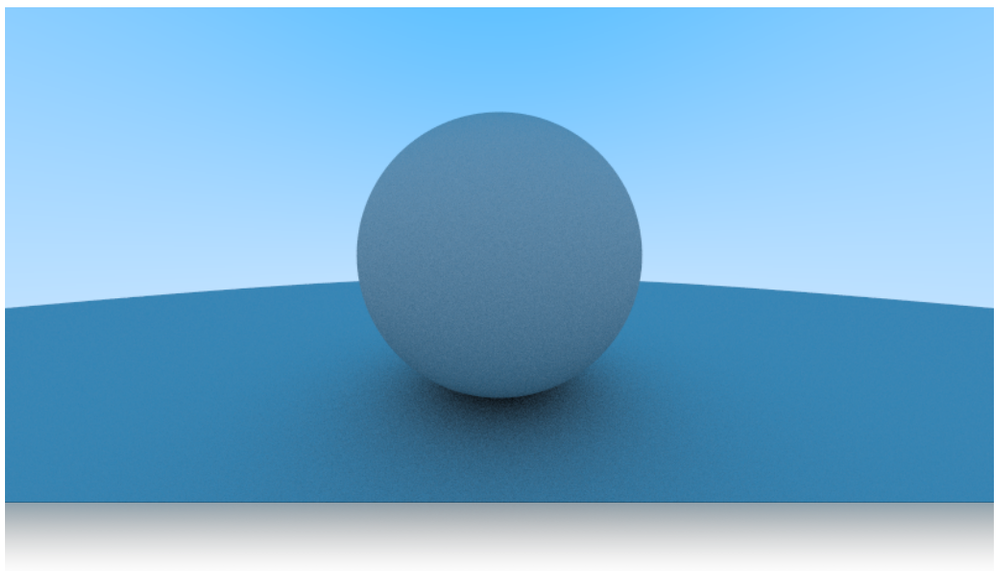
But, it is *round*, so that’s encouraging.

With normals



If we render the *normals*, we *can* confirm it is a sphere.

With Lambertian material



Add *Lambertian scattering*, and it starts to look pretty good.

Finally…

- *Really* try to understand your domain…
- …No matter what that may be
- Time invested in knowledge is *never* wasted
- ***Always*** challenge yourself!

https://github.com/kapsy/swift_ray_tracer

I've only covered one of the *many* algorithms used in the movie.

If you're interested in the others, please check out the Github.

Finally, I don't want you to worry if you didn't understand the math here at all.

If there's something I want everyone to take away, it's this:

As a developer, time spent really getting to know your domain, whatever it may be, is extremely valuable.

Always challenge yourself.

Finally, if you would like to talk about ray tracing, don't hesitate to come and visit me at the ZOZO booth!

Thank you.

References

- Ray Tracing Minibooks, Pete Shirley
- Scratchapixel 2.0
- Ray-Triangle Intersection, Möller–Trumbore
- Wikipedia: Cramer’s rule
- On building fast kd-Trees for Ray Tracing, Wald-Havran
- Realtime Ray Tracing and Interactive Global Illumination, Wald