Concepts:

I wouldn't say I was surprised but more so shocked that we could use the quadratic formula to solve a ray-sphere intersection (like you said, the formula always comes back in some sort of way). Once you look at how to format a,b,c variables. It makes perfect sense, at least after a few tries with the formula it does start to. Definitely have to be careful and make sure I do my vector math properly. I ran into the issue of having a vector for b on our quiz this week. I thought "How in the world did I end up with a vector for b." I then realized I sped through the formula for b too fast. For I had forgotten to get the dot product of dHat and (o-c). It also helped me realize how the coefficients for a,b,c make sense to put them there.

Another concept I enjoyed learning were explicit equations and implicit equations. From my understanding basically implicit equations are equations that can tell us whether a point lands on the surface of that area you imply. It cannot generate points, but you can just insert any point you want. From there you can figure out if it is a valid point or not for that equation.
Explicit equations are kind of the opposite of implicit equations. Instead of checking if a point lands on the surface. You can generate points most of the time, at least you can for a ray explicit equation. However, it is hard to justify if that generated point is on the ray. I think this is where you generate a point, then combine it with the implicit equation to see if that generated point lands on the surface that we have. If this is right, that's awesome to see how the two equations can work together to solve some problems, not all but some.

Experience:

For this week, my partner and I utilized vscode's live share extension to work together on the assignment. We solved the code equally, but 100% we were confused at a lot of points in writing the code. Both of us really had to talk about "what does this equation mean" or "what does alpha_1 (b_positive in our code)." Since we had to re-use a lot of variables for various parts of equations and vector math. We seriously had to sit down and learn what every variable, and coefficients, meant for our program.

One area we got stuck at, at least I for sure was, using the discriminant, alpha 1, and alpha 2 variables.

```
// alpha 1
var b_positive = (-b + Math.sqrt((b*b) - (4*a*c)))/(2 * a);

// alpha 2
var b_negative = (-b - Math.sqrt((b*b) - (4*a*c)))/(2 * a);

//console.log(a + " : " + b + " : " + c);
//b^2 - 4*a*c
var discriminant = (b * b) - 4 * a * c;

// check if we should even do more computation
    // if any of these are negative, then we 100% won't have a hit on our sphere
      // this also checks, if our raycast originates inside our sphere
        // we don't care for any raycast that originates within our sphere, at this moment
if(discriminant < 0 || (b_positive < 0 || b_negative < 0)){
  return {hit: false, point: null};
}
```

From our discussions, in this case since we didn't care for any rays that originated inside the sphere, we learned if the discriminant, alpha 1 (b_positive), or alpha 2 (b_negative) were actually negative. That meant the ray's origin was inside the sphere (please correct me if I am wrong). It did pass the inside sphere test case, and it wouldn't without. (Maybe a luck guess, but that's what we were thinking when coding this part)

Next confusing topic is when we actually had a valid ray/intersection.

```
var scaledDirection = r1.direction.clone().multiplyScalar(alpha);
// add our scaled direction to our raycast origin
    // this basically is like adding new vectors, if you add a+b you'll get a c.
      // so our RaycastHit is now c
var RaycastHit = r1.origin.clone().add(scaledDirection);
```

This is how I interpreted my thought process of writing this code. If we find a valid ray/intersection, we want to scale our direction vector by our valid alpha. This makes the direction vector scaled up to our valid intersection point. So the tip of this direction vector now lands on the intersection point. Now we need to store where the hit actually is, but the Raycast hitPoint isn't on the scaled-direction vector. We have to add our scaled-direction with our raycast's origin. I also put it in terms of, we have a and b, our actual hitPoint is c. Since we want to find where the hit was from the origin of the ray, which is why we want to add our scaled-direction and raycast origin.