Joanna Li: Ray Tracer Reflection and Development Log

Reflection

I learned a lot about programming, development, and project management from creating this program. In order to implement the features, I had to understand how each step of ray tracing worked on a basic level, and figure out how they fit together. I also learned that it was easier to start with small and hard-coded examples first, and helpful to break bigger undertakings into small pieces.

Instead of thinking of goals like “implement shaded sphere by Tuesday,” I broke the steps down:

1. Review ray tracing slides
2. Do sphere intersection by hand
3. Write some lines that do the calculations on the same numbers, and then print the results to console

…

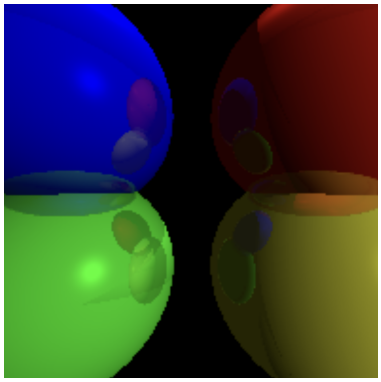
I tried to take shortcuts with writing testers and hardcoded examples near the end, but that may have been why it was so difficult for me to identify the error with t-min and t-max values– a situation which is in itself a lesson about not letting random constants that are easily forgotten control integral functions in your program.

To develop the program further, I would add a plane object, directional lighting, anti-aliasing, and more options for materials. I think it would be very satisfying to support transparent or refractive spheres and metallic objects. I would also have t-min and t-max be passed into each call to the intersectRay function instead of setting them as global variables, and research more about how to set their values correctly to reduce strange artifacts.
I also want to go over the program more carefully to add optimizations like accounting for shadow coherence and be more careful about normalizing vectors.

On the more extreme side, some advanced features I would want to implement if I had a lot more time would be using random sampling and probability density functions to shade more realistically, implementing support for other BRDFs (especially anisotropic for brushed metal), and writing the program so that it can be parallelized (I probably need to rewrite the ray tracer in another language and learn a lot more about parallelization).

12/14 File Input and Final Touches (3 hr)

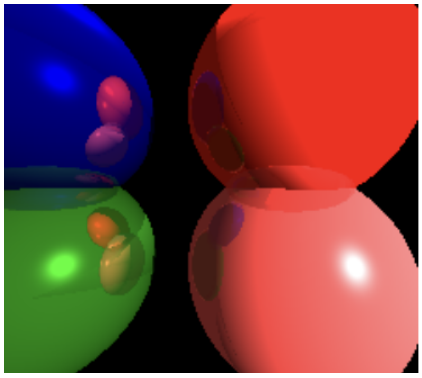
In the morning, I told the story of this project’s making to the class, and generated a demo image to show off the reflection:



Reflection!

I am pretty proud of the progress I made, and excited about the possibility of adding more features if I ever revisit the challenge of writing a ray tracer.

For final touches, I organized and commented the code better, made the color of a light affect the shading of the spheres, and made the reflectivity constant of the material factor into the shading math in a clearer way.

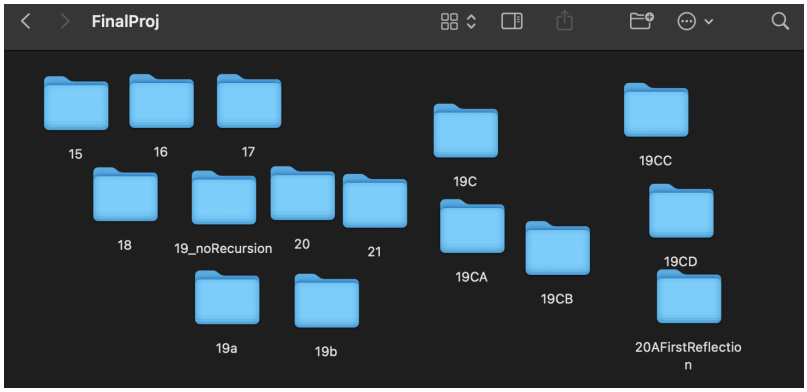


I turned up the red component of this light to intense levels, the lower right sphere is actually a white sphere, but it looks red under the light

12-13 Reflections (6 hr)

The next step was to implement reflections.
It was very challenging to implement this feature. I spent a very long time testing out different versions of the program, and development progressed in a non-linear way from this point.

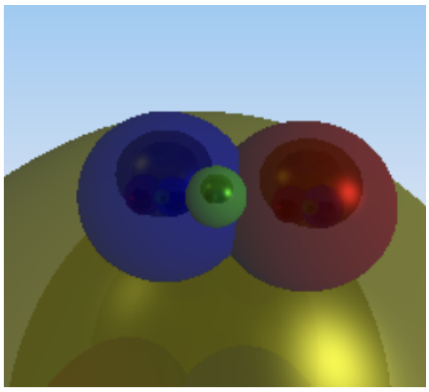
The difficulties were that there was no one simple function for calculating the shaded color given a certain ray through the scene. I wrote the program in a short time, so I made some ill-advised decisions to make the deadline and left some functions with too many parameters and spread out this color computation across many nested functions. This made writing the recursive reflection computation very, very difficult. I had to restructure all of my rendering functions.



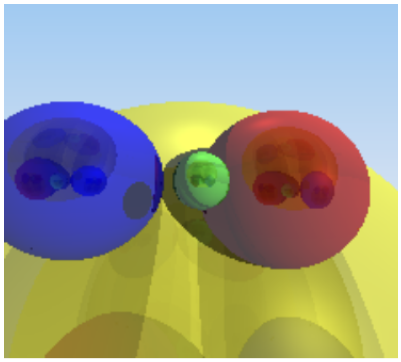
The Cambrian Explosion of final project file versions

I was led on many wild-goose chases through files and functions, trying to locate undefined values that broke the program by being passed in mysterious places on the rendering pipeline.

I was very excited when the first successful reflection image was generated!
Even if it did not seem correct (all the spheres should not reflect the same image), it was progress!



First generated reflection

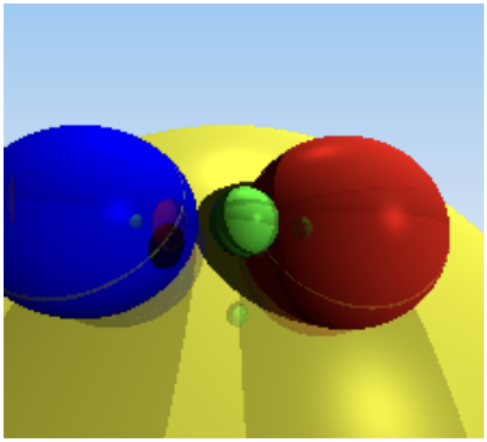


Incorrect: all the reflections were the same!

It was easy to identify why the reflections were incorrect: I had been calculating the new intersection points recursively from the view of the camera, and not each point from which reflected rays originate. It took a very, very long time to fix this error because changing the code to what seemed correct caused most of my image to disappear.

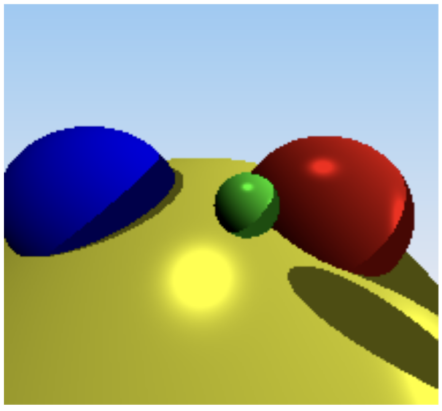
I eventually realized that the strange effect was a result of having set my t-min and t-max values inappropriately, and moreover that they need to be a parameter that is passed in every time intersections with spheres are calculated because the interval of relevant intersection distances should vary based on our ray tracing needs.

After tweaking the interval and changing the code…



It's not perfect, but the reflections are finally correct!

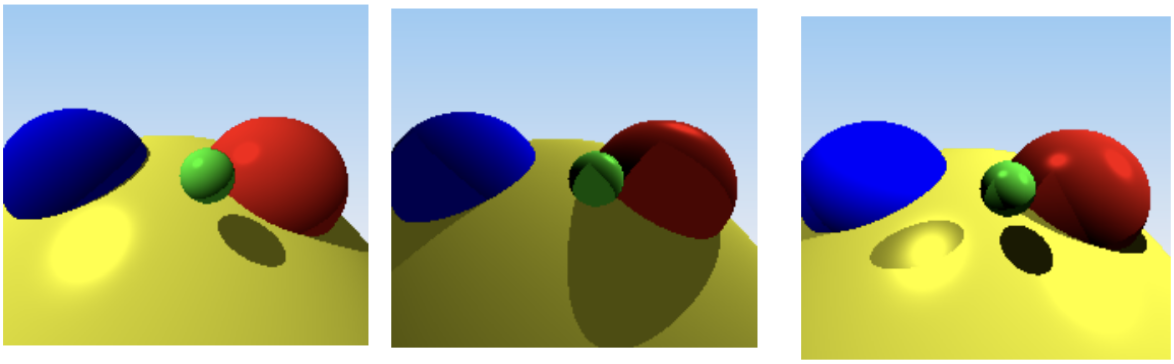
12/12-13 Shadows (4 hr)



At first, I was satisfied by these shadows, but I realized they were not computed correctly when they stayed the same even if I moved my light source.

I realized that I was using the wrong ray directions when computing the objects that intersected the ray cast from each point to a light source:

I used the directional vector from the point to the camera instead of the one from the point to the light.
Now, the shadows are dependent on the position of the light source.



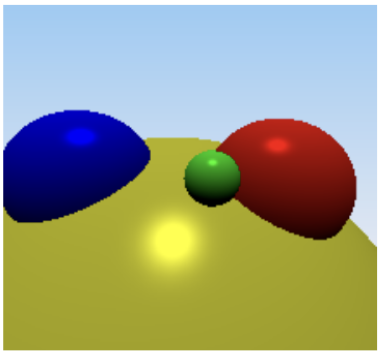
Light at positions [0,-1.3,-0.6] and [1,-1.3,-0.6], and example with multiple light sources

The shadows somewhat worked with multiple light sources, but there was some strange interference. Another improvement would be to scale the darkness of these flat shadows based on some factor so that they look more realistic. -

12/12 Phong Shading (5hr)

Implementing the Blinn-Phong shading model was a challenge because I had to refine my understanding of how the shading was calculated and also restructure much of my code.

- First, I implemented and tested basic Phong model shading by adding a specular component to calculations for the diffuse lighting model:

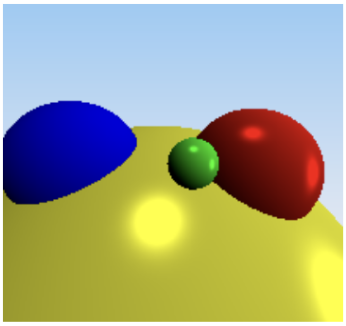


After that, I generalized the hardcoded example by:

- Making a shadeAt() function which computes the color of a point on any sphere given the position of the point, and a pointer to the sphere from which its material properties and normal vector can be accessed

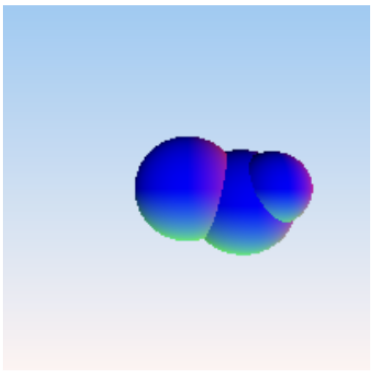
- This removed the need for the sphere itself or the material object to have functions that computed shading
- Making an array of lights for the scene and having shadeAt() add together the contributions of all the lights to the intensity of diffuse and specular shading to the scene.
- Because I don't anticipate the addition of multiple ambient lights in a scene, the ambient lighting on each material is represented by a constant ka
- Since lambertian diffuse materials can be created by setting the ks parameter in a Phong material to zero, I also removed this type of material from my program for simplicity of implementation

With these changes, the program produced the image below, exhibiting spheres shaded with the Phong model and the diffuse model, using 2 directional lights.



12/11 Diffuse Lighting and Shadow (3hr)

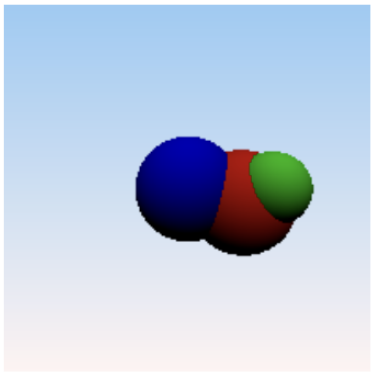
- I started out by shading a sphere based on its normalized surface normal vectors:
- I decided that each object type would have a shade() method which would then call the colorAt() method of its material. I am working on a diffuse Lambert material first
- Calling shade() from the renderer and colorAt() in the shade() method of each object will make use of JavaScript's duck-typing to structure the code. I learned about this from Source 2.
- This was a test which shaded each point on a sphere using its unit normal vector (which points outwards from the surface).
- The r,g, and b values used to color each point was determined by multiplying the x, y, and z components of the normal vector at the point by 255. On the overlapping spheres example, it looks like:



The blue RGB value corresponds to the size of the z-component, so it makes sense that surfaces that faced the positive z-direction were more blue. On the Javascript canvas, the y-axis is inverted, so it makes sense for downward-facing surfaces to be shaded green

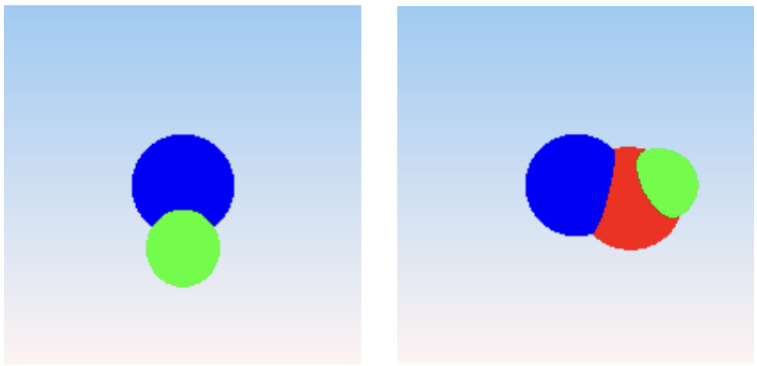
- After that, I implemented a hard-coded example with a direction light source coming from the top right

```
class lambertMaterial {
    kd;
    color;
    constructor(kd, color) {
        this.kd = kd;
        this.color = color;
    }
    //Returns color of material at a point, given the normal vector of a surface
    //and the direction of the light source
    colorAt(point, normal, lightColor, lightDirection) {
        //console.log("shaded lambert");
        //HARDCODED:
        lightColor = 1;
        lightDirection = [0,-1,1];
        let constant = this.kd * lightColor * (Vec3.dot(normal, lightDirection));
        //console.log(normal);
        return (Vec3.scale(constant, this.color))
        //return(Vec3.scale(255,normal));
    }
}
```



12/7-10 Overlapping Spheres (4hr)

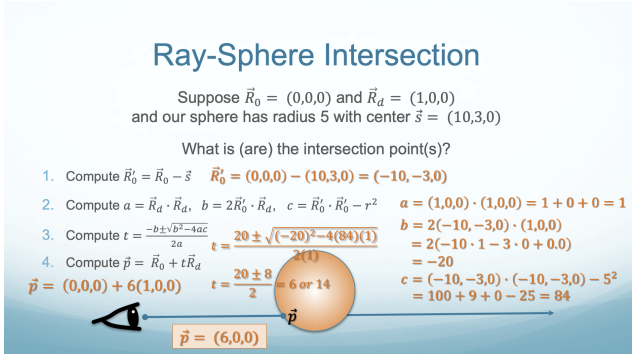
- The next feature I implemented for the program was checking for intersections with an array of all objects in the scene with every ray that is cast, and rendering their overlap correctly.
- I created materials objects with color attributes for the spheres to use as a parameter.
- I had the sphere intersection return both the t-value calculated and the point of intersection so that the draw() function would render the pixel at (x,y) based on the color of the first object intersected by the ray through (x,y, z), where z describes the location of the viewport.
- With this, the ray tracer now renders overlapping spheres!

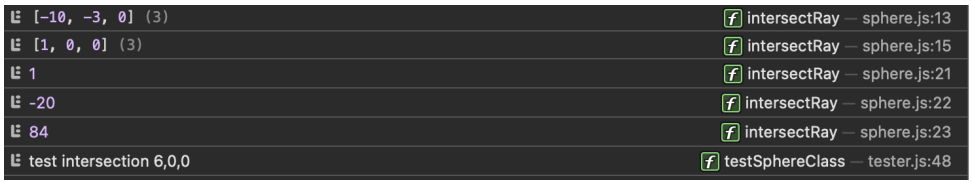


Overlapping spheres

12/6 Sphere Intersections Pt. 2 (3hr)

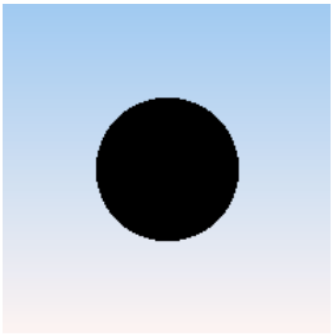
- Today, I wrote a Sphere class and tested out the implementation of sphere intersections. Towards this goal, I wrote vector functions to return a dot product and a scaled vector.





Verified that the sphere intersection code works on an example from class

With this, I was able to render my first sphere with ray tracing.



Lines left:

0

Ray-traced sphere!

My next step is to have the color of the sphere be taken from a color attribute of the Sphere “object,” and render multiple overlapping spheres correctly.

I also want an alternative to generating images with the HTML canvas, so I aim to implement the ability to export results as a ppm file later in the project.

12/5 Sphere intersections Pt. 1(2hr)

Today, I worked to implement a ray caster, and worked towards implementing spheres in the program.
At first, I imported the glMatrix library to perform matrix math for my program, but I decided to write my own Vec3 class with static methods to perform basic vector math on 3D vectors to reduce the bloat of adding a package, and remove the convention of having each operation require the output vector as a parameter.

Also, it was just more fun writing everything from scratch.

I can now do something like: add(vec1, vec2) instead of add(resultVector, vec1, vec2).

To test the code, I verified that a ray cast straight out from the camera pointed straight down into the viewport, with value (0,0,100) and a camera placed at the center of the viewport at (ymax / 2, xmax/ 2) to adjust for the way that HTML canvas coordinates for the viewport run from 0 to 200 instead of -100 to 100, as presented in lecture.

I used a linear interpolation between 2 RGB colors to make a gradient background, and stopped here for the day.

12/4 Drawing on the Canvas and Making the Viewport (2 hr)

Today, I wrote a function to loop over the viewport and request its color from a function called getPixelColor(x,y) which just returns the rgb values for black. I also wrote the logic for viewport size to be calculated from camera parameters, which I have set to create a 300 x 200 viewport located at z = -100. Inspired by “Ray Tracing in One Weekend” [Source 1], I also added a counter for the number of lines left to render

A sanity check that simple code to fill the viewport works

One problem that came up is that the viewport size calculations had some errors, and did not return whole numbers. I rounded the viewport height value to the nearest integer as a temporary fix.

11/30 Planning the Program: What do I Even Have to Do? (1hr 15 min)

Today, I planned out the structure for my ray tracer program
I will need a class to read in the scene description, a class for the ray caster, and some classes with specialized functions, such as one for all general math equations.

A challenge was how overwhelming and complex many online examples were, but to simplify my task, I will aim to implement an object oriented JavaScript program with a fixed canvas size and fixed camera settings at first.

I will implement the file reader last, and use an example of a 200 x 200 viewport with a camera at N=1, aspect ratio = 1, field of view = 90 degrees. With these settings, the viewport will be a 2 x 2 plane at z = -1.

Sources

- I used these guides to review the equations needed and for guidance on deciding the order in which to implement features
- I did not use or refer to the starter code from these sources, most of which was not written in JavaScript, but C++

1. https://raytracing.github.io/books/RayTracingInOneWeekend.html
2. https://faculty.washington.edu/joelross/courses/archive/f14/cs315/hwk/7/
3. https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/diffuse-lambertian-shading.html
4. CSC 240 Class Slides 14, 15, 18, 19
   https://www.science.smith.edu/~nhowe/teaching/csc240/lect/vid14.pdf
   https://www.science.smith.edu/~nhowe/teaching/csc240/lect/vid15.pdf
   https://www.science.smith.edu/~nhowe/teaching/csc240/lect/vid18.pdf
   https://www.science.smith.edu/~nhowe/teaching/csc240/lect/vid19.pdf
5. https://www.cs.cornell.edu/courses/cs4620/2011fa/lectures/08raytracingWeb.pdf
6. https://gabrielgambetta.com/ - inspiration for test images and development order