Raytracing in the Browser

Ryan Kustaborder

CIS 457

4/24/23

1. Executive Summary

In this project, I explored the efficacy of raytracing in the browser. By limiting myself to using pure JavaScript and HTML, I was able to get a sense of if this was a project that could be developed at an enterprise level, and the answer was no. While I was able to achieve some good results and solid base level raytracing, there are serious efficiency and development issues that limit the depth into which JavaScript raytracing can go. This paper also explains some core concepts of raytracing, such as Lambertian reflections, reflective materials, sampling rates, and more. I also developed a demo wrapper for my raytracing implementation that allows the user to control the camera when rendering a scene and save the output image.

2. Introduction

Raytracing is a method of rendering digital scenes which seeks to emulate the way that our eyes see the world. This physical approach can provide high degrees of realism, which has caused it to be a favorite method for digital 3D artists and other professionals who work with computer generated images. In this paper, the efficacy of raytracing in web browser environment is evaluated and potential solutions are proposed.

3. Raytracing

3.1 - Basic Principle

In the real world, light is emitted from light sources, and we can imagine the path it travels as a ray that comes from the light source. It then bounces around the world, reflecting off of objects as it travels. With each bounce, some of the light is absorbed by the object, which results in a change in the color of the light ray as it continues to propagate. Some of these rays will just bounce off back into space, but some will reflect into the viewer's eye where the brain can process the color information it carries.

Raytracing seeks to simulate this behavior, but instead of an eye, there is a virtual camera. Rays travel around the virtual scene, with the color of the ray being changed by its interactions with the digital objects. Below you can see a simple raycasting scenario:

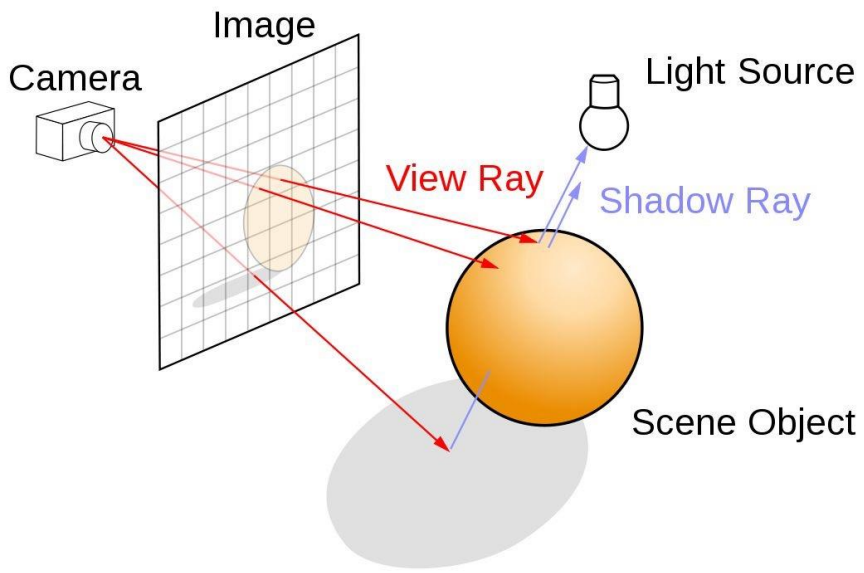


Figure 1: Simple Raycasting Example

You may notice that the rays are moving from the camera to the light source, instead of the reverse that we observe in the real world. While this may seem counterintuitive, we reverse the direction of the rays to avoid extraneous calculations. If we shot rays out from the light source, very few of the rays would end up passing through the camera's virtual lens, since most will bounce off in other directions and end up distributed across the scene. By shooting the rays out of the camera, we can ensure that every ray we calculate ends up in the camera lens.

3.2. Lambertian Materials

3.2.1. Base

While there are many different approaches to materials for raytracing, I chose to use Lambertian diffusion for my implementation of matte materials. Lambertian materials assume that rays are reflected non-uniformly, with more rays being redirected towards the surface normal. The distribution of the rays is rotationally symmetric, so the direction of the incoming ray does not affect the outgoing ray.

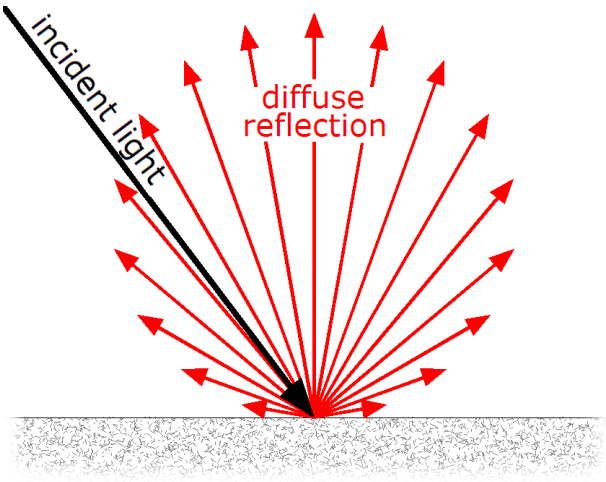


Figure 2 Lambertian Reflection Distribution

Lambertian reflections also accounts for the angle of incidence, or the angle between the incoming ray and the surface. This is important because a beam of light that is coming in at a lower angle (closer to parallel to the surface normal) will have cover a larger surface area on the object. This results in the light level per area being lower, and thus the outgoing ray will be dimmer. The opposite holds true for rays that come into the surface at a high angle (closer to perpendicular to the surface normal). The beam of light would cover less area, meaning more light per area, resulting in a brighter outgoing ray.

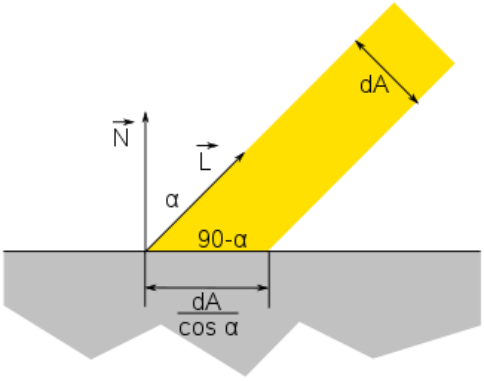


Figure 3: Affect of Incoming Angle on Surface Area

To actually calculate the outgoing ray, we take the dot product of the surface normal (N) with the unit vector pointing in the direction of the incoming light. This is what accounts for the angle of incidence, as the higher the angle, the higher the dot product will be. We then multiply this by the color of the object (C), simulating the object absorbing some of the colors of the incoming light. Lastly we multiply all this by the color of the incoming light (I_L) to blend them together.

$$B_D = (L \cdot N)CI_L$$

3.2.2. Reflective Lambertian Materials

Pure Lambertian reflections are perfect for matte materials, but they cannot simulate reflective materials like metal or plastic. This is due to the smoothness of these materials altering the way that the incoming ray is scattered by the material. With a completely smooth material, the incoming ray will scatter with the exact same angle that it came in with.

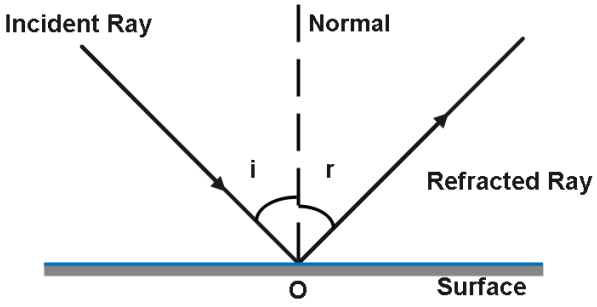


Figure 4: Perfect Reflections

Not all reflective materials are perfectly smooth, so we must account for that in our scattering calculation. Surfaces that are reflective but smooth behave as sort of a mix between the Lambertian reflection and the reflection seen with the perfect reflections. To achieve this effect, we can add slight perturbations to the outgoing ray, with the magnitude of these perturbations being proportionate to the roughness of the material.

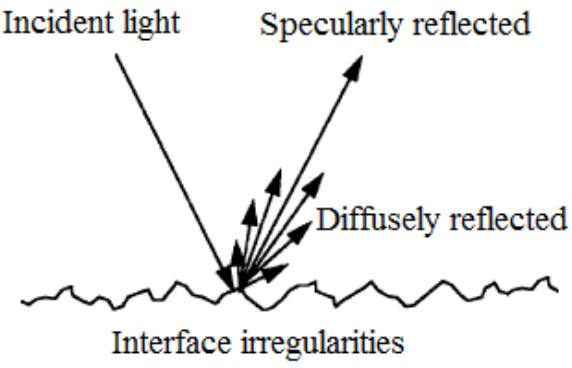


Figure 5: Effect of Surface Roughness on Reflection Distribution

3.3. Camera Properties

3.3.1. FOV

In addition to the materials, we must also consider our camera. Cameras use lenses, which allows us to use the properties of optics that we see in the real world to simulate how a virtual camera should behave. The first of these properties is the Field of View (FOV) which determines the area of a scene that the camera can see. It is defined as the angle formed by the outermost rays that come from the camera. By increasing the FOV we can get a perspective zoom out, and by decreasing the FOV we can get a perspective zoom in.

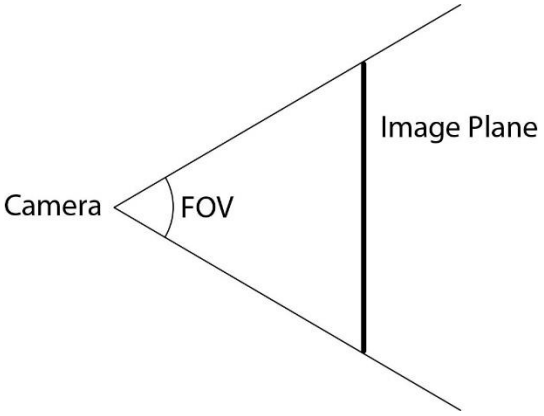


Figure 6: Camera FOV Angle

3.3.2. Focal Length

Another camera property we can emulate is the focal length of the lens. Convex lenses focus the light to a singular point, known as the focal point. As you go away from the focal point, the different rays begin to diverge, resulting in a slight blurring of the image. This effect gets more extreme the farther away from the focal point the object is. To achieve this effect in our raytracing program, we will need to add some random perturbations to the rays if the object they are intersecting, proportionate to the distance away from the focal length.

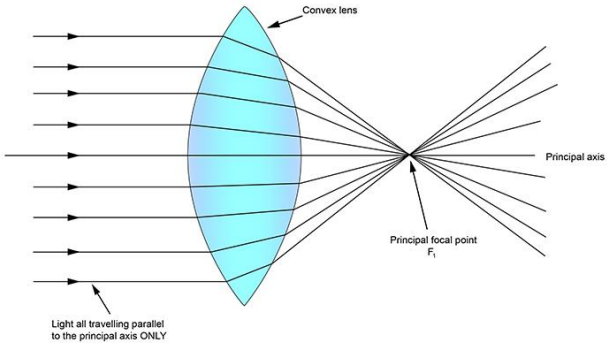


Figure 7: Lens Focal Length

3.3.3. Samples Per Pixel

The last camera property we will discuss is the samples per pixel. This number refers to how many rays are cast for each pixel of the output image. We do this to help increase the consistency across the image, as raytracing is not a perfect simulation. Sometimes, an individual ray will bounce in a peculiar direction, meaning that the color it returns is not what we would expect in the real world. To counteract this, we send many different rays that are all slightly different and then average them together at the end. This helps hide any outlier rays that through random chance ended up behaving strangely. While this drastically improves the image quality, it also increases run time, as you are effectively having to raytrace the image multiple times.

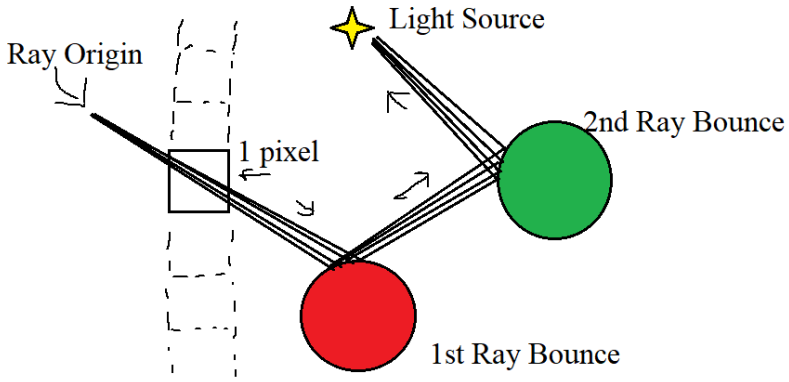


Figure 8: Effect of Perturbation on Initial Rays

4. My Implementation

4.1. Overview

For my implementation, I wanted to see how well pure JavaScript could be applied to raytracing. While this wouldn't be a smart choice for an enterprise level application, it was a fun experiment and worked better than expected. In terms of the architecture of the program, it has remained relatively simple. There are a few classes for representing mathematical objects such as points and rays, there are some classes that are effectively structs since JS does not have native struct support, and then there are my Hittable classes which represent our objects in the scene.

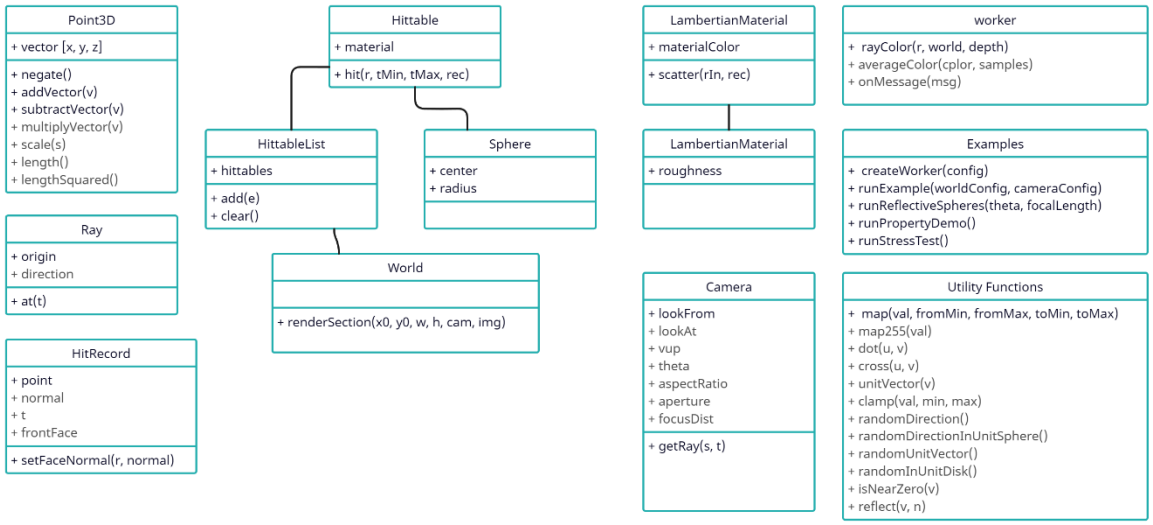


Figure 9: Class Diagram

4.2. Important Methods

4.2.1. The Worker Script

The worker script can be broken into two phases. The first is reconstructing the scene from the input data. Since the serialized data has all class context removed, we need to go through it and construct class objects for the camera, the scene, and the objects. Once we have constructed all these objects, we can then move on to the second phase of performing the raytracing.

```
self.onmessage = function (msg) {
    // Construct the world object from the input message
    const world = new World();

    for (let element of msg.data.world) {
        let location = new Point3D(element.x, element.y, element.z);

        let mat;
        let m = element.material;
        if (m.type == "lambertian") {
            mat = new LambertianMaterial();

            mat.materialColor = new Point3D(m.color.r, m.color.g,
m.color.b);else if (m.type == "reflective_lambertian") {
            mat = new ReflectiveLambertianMaterial();

            mat.materialColor = new Point3D(m.color.r, m.color.g,
m.color.b); mat.roughness = m.roughness;
        }

        // Add the object to the scene
        world.add(new Sphere(location, element.radius, mat));
    }

    let c = msg.data.camera;

    // Construct the camera from the input message
    const cam = new Camera(
        new Point3D(c.lookfrom.x, c.lookfrom.y, c.lookfrom.z),
        new Point3D(c.lookat.x, c.lookat.y, c.lookat.z),
        new Point3D(c.vup.x, c.vup.y, c.vup.z),
        c.theta,
        c.aspectRatio,
        c.aperture,
        c.dist_to_focus
    );
    const img = msg.data.image;
    const bounds = msg.data.bounds;

    // Begin rendering the scene
    self.postMessage(
        world.renderSection(
            bounds.x0,
            bounds.y0,
            bounds.width,
            bounds.height,
            cam,
            img
        )
    );

    // Close the worker when it is done
    self.close();
};
```

4.2.2. world.renderSection

The second phase of the worker script is where we do the raytracing. First, I defined a bunch of constants so that I could have faster read times inside the for loops. While this may seem trivial, it did have a noticeable effect on the runtime since the loops run so many times. We then iterate over each pixel in the output image, and then further iterate over the total number of samples per pixel. Inside these loops we cast the rays and get the color returned from its path. Lastly, we average out this pixel value and write it to the output array.

```
renderSection(x0, y0, w, h, cam, img) {
        // Define these as consts for slightly faster read times
        const imgWidth = img.imgWidth;
        const imgHeight = img.imgHeight;

        const maxDepth = img.maxDepth;
        const samplesPerPixel = img.samplesPerPixel;

        const totalPixels = w * h;
        const initialValue = [0, 0, 0, 1];
        const imgData = Array(totalPixels).fill(initialValue).flat();

        let i = 0;

        // Iterate over each pixel
        for (let y = y0 + h; y > y0; y--) {
            for (let x = x0; x < x0 + w; x++) {
                let pixelColor = new Point3D(0, 0.5, 0);

                // Cast samplesPerPixel rays for each pixel
                for (let s = 0; s < img.samplesPerPixel; s++) {
                    let u = (x + Math.random()) / (imgWidth - 1);
                    let v = (y + Math.random()) / (imgHeight - 1);
                    let r = cam.getRay(u, v);
                    // Cast the ray
                    let colorFromRay = rayColor(r, this, maxDepth);
                    // Add to cumulative color for this pixel
                    pixelColor = pixelColor.addVector(colorFromRay);
                }

                // Write the final pixel
                let color = averageColor(pixelColor,
samplesPerPixel);
                imgData[i + 0] = map255(color.x());
                imgData[i + 1] = map255(color.y());
                imgData[i + 2] = map255(color.z());
                imgData[i + 3] = 255;

                i += 4;
            }
        }
        return imgData;
    }
```

4.2.3. raycolor

The most important method in my implementation is the raycolor(ray, world, depth) method which is what casts the rays and performs the majority of the raytracing. It is called recursively, with the depth decreasing until the depth is 0 or no more scattering occurs. If the ray hits an object, we calculate the next ray in the series and add it to the final color.

```
function rayColor(r, world, depth) {
    // If we have hit the bounce limit, stop
    if (depth <= 0) {
        return new Point3D(0, 0, 0);
    }

    let rec = new HitRecord();

    // Cast the ray
    let result = world.hit(r, 0.001, Number.MAX_SAFE_INTEGER, rec);

    // If ray hits an object, calculate next bounce
    if (result.hit) {
        let resRec = result.record;

        // Calculate the new ray based off the material of the object
        let scatterResponse = resRec.material.scatter(r, resRec);

        if (scatterResponse.didScatter) {
            return scatterResponse.attenuation.multiplyVector(
                rayColor(scatterResponse.scattered, world, depth - 1)
            );
        } else {
            return new Point3D(0, 0, 0);
        }
    }

    // If no hit, color is based on background gradient (sky)
    let unitDirection = unitVector(r.direction);
    let t = 0.5 * (unitDirection.y() + 1.0);
    return new Point3D(1.0, 1.0, 1.0)
        .scale(1.0 - t)
        .addVector(new Point3D(0.5, 0.7, 1.0).scale(t));
}
```

4.3. Parallelism

One of the biggest drawbacks of raytracing as a rendering method is the time it takes to compute. One way to help decrease the run time is to render different portions of the image in parallel. Unfortunately, JS does not have great multithreading support, but it does have Web Workers. Web Workers are threads that can be initiated by the Main Thread of the program. They are incredibly limited in what they can do, but they will run in the background and can return some data in the end, so we will work with what we have.

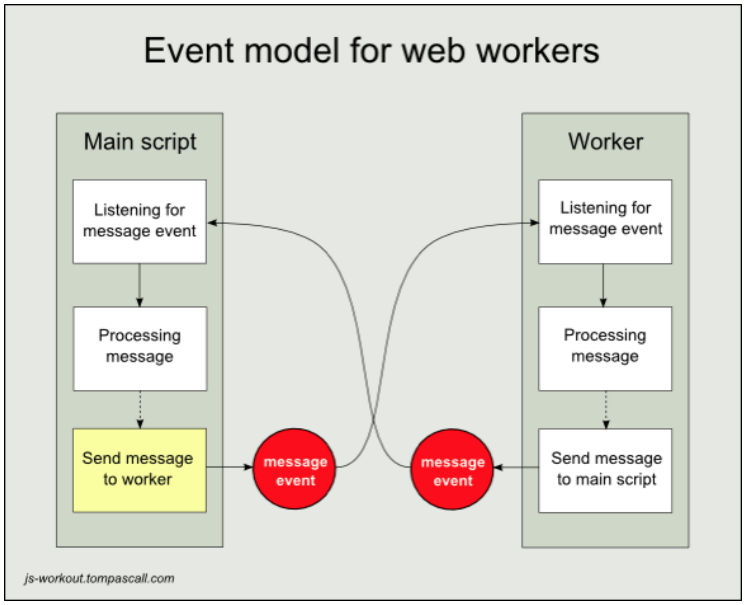


Figure 10: Communication Between the Main Thread and a Web Worker

Web Workers are relatively simple to manage. First you have to create a worker object, then give it some code to run, then tell it what to do when it finishes, and you're done. But while their simplicity allows for easy set up, it greatly limits what the worker can do. For instance, web workers cannot import any external code as they are no longer working in the ES6 module environment. They also cannot manipulate any DOM elements as they are detached from the webpage itself. But the downside that I had to deal with the most is that any data that is passed between the worker and the main thread must be serializable with JSON. This means that you cannot pass any functions and objects lose any of their meaning and become structs. To handle this, I ended up having to move most of the code into the singular worker file which made it more difficult to maintain and read.

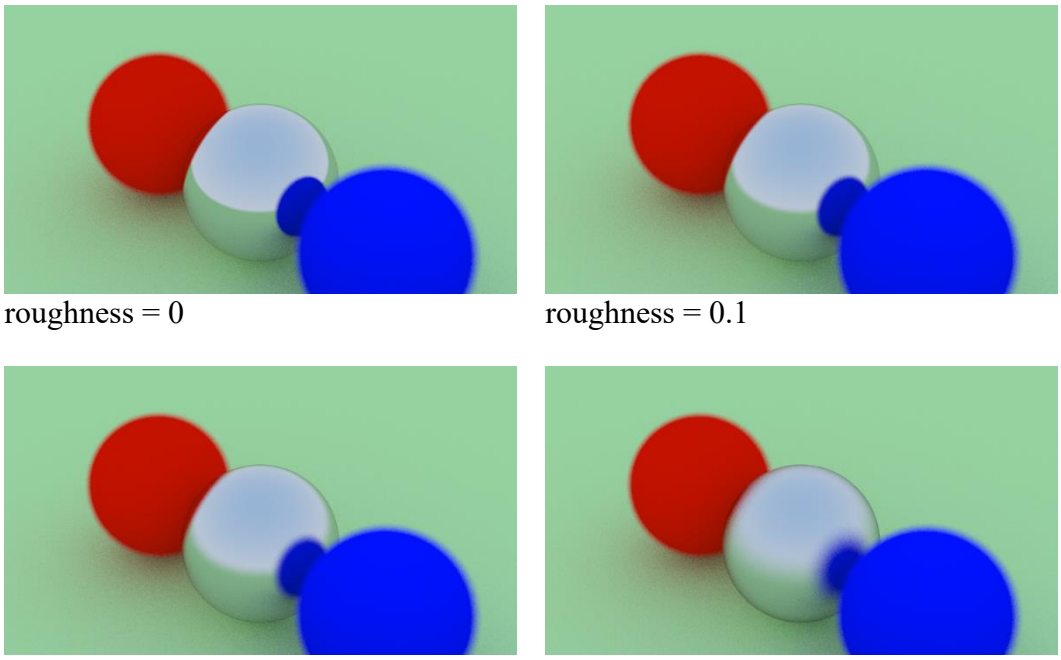
4.4. Demo Wrapper

Lastly, I needed to create a user interface that would allow the user to control the rendering script. The wrapper I came up with is very simple and made just for the demo in the presentation since I wanted to focus on the raytracing as that is the scope of the course.

5. Results

5.1. Variance of Parameters

5.1.1. Material Roughness

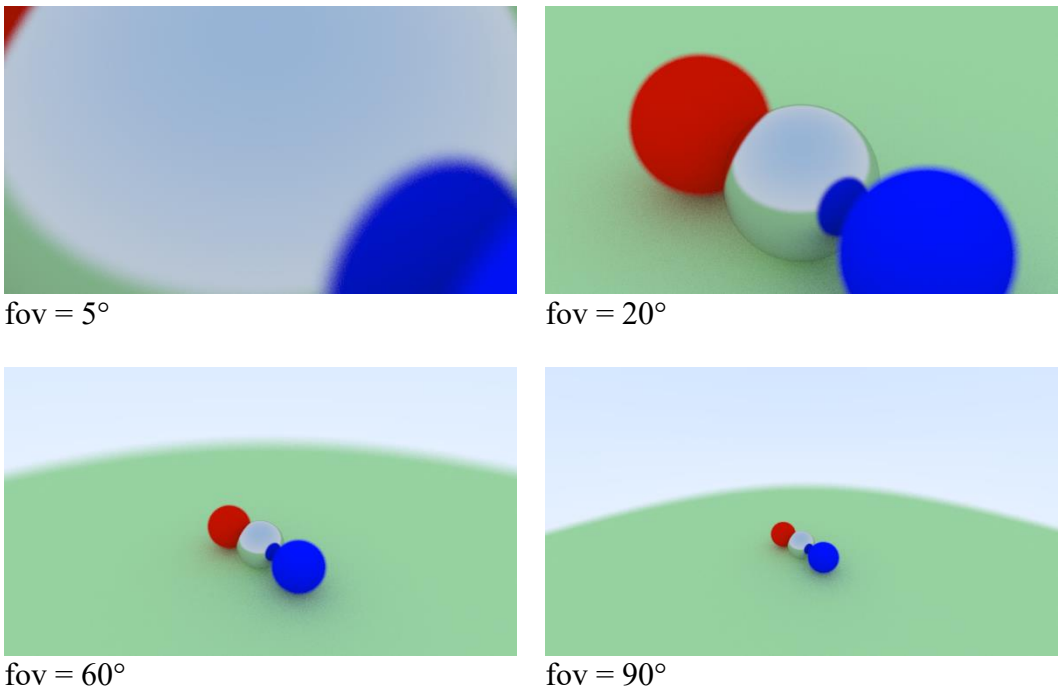


roughness = 0 | roughness = 0.1

roughness = 0.2 | roughness = 0.5

Figure 11: Variation of Material Roughness

5.1.2. Camera FOV

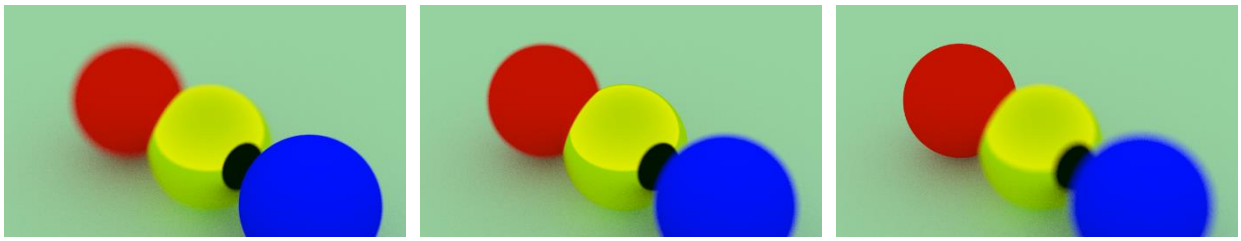


fov = 5° | fov = 20°

fov = 60° | fov = 90°

Figure 12: Variation of Camera FOV

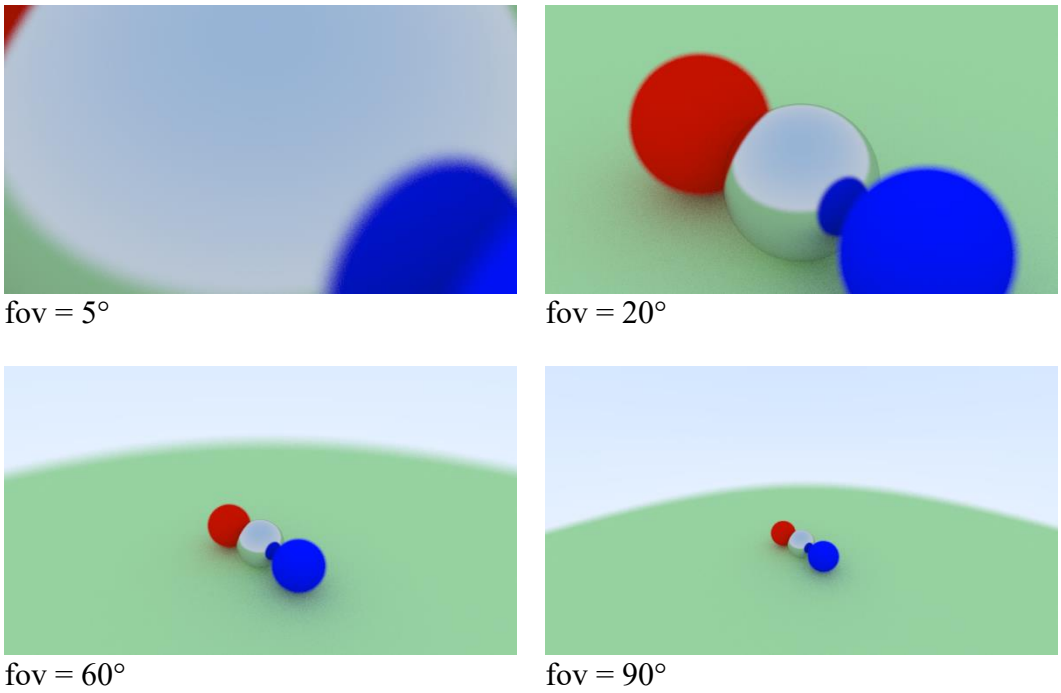
5.1.3. Camera Focal Length



focal length offset = -0.05 focal length offset = 0 focal length offset = +0.05

Figure 13: Variation of Camera Focal Length

5.1.4. Camera Samples per Pixel



fov = 5° fov = 20°

fov = 60° fov = 90°

Figure 14: Variation of Camera Samples per Pixel

5.2. Stress Test

I wanted to test to see how efficient my implementation was, so I generated a scene with lots of different objects to see how long it would take. I used a mixture of materials and positions to make sure it would be representative of a real scene. Rendering the scene on my laptop took on average 373 seconds, or about six minutes. This is not ideal, but there are a few things to consider. The first is that I was running this on a not so great laptop, which definitely limits the speed of the application since its running locally on the computer. The other is that there are some objects that are outside of the scene which do not have any affect on the final rendering. These ideally would be ignored to improve future efficiency.

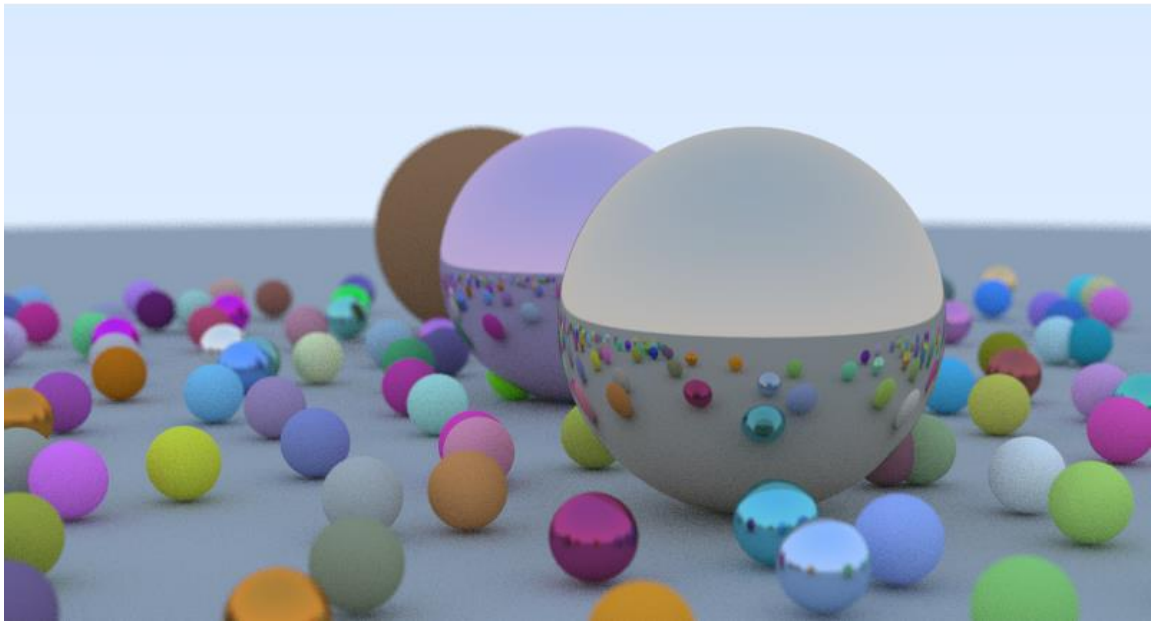


Figure 15: Stress Test Scene Rendered

6. Conclusion

6.1. Advantages / Disadvantages

When it comes to the advantages of raytracing in the browser with pure JavaScript, there are not many. If I had to pick one, it would be that JS is very flexible and allows for rapid prototyping and development, but that is a very small part of the development process and does not outweigh the many drawbacks.

The first of these drawbacks is the inefficiency. This inefficiency comes from a couple sources. The first is that JS is not a particularly efficient language. Since it is interpreted, the code must be compiled each time which slows down the run time. One way to mitigate this while staying in the browser would be to use something like Blazor WebAssembly apps which are written in C# which is a must faster language. The other source of the inefficiency is that the raytracing is running locally on the user's machine. This means that the speed of the program is limited by the hardware the user has, which is typically not very fast. One way to mitigate this

would be to introduce server side rendering, which would let the raytracing code run on very fast server machines.

The second big drawback is that JS uses automatic memory management and garbage collection. This can be nice when it doesn't matter, but it limits the amount of optimization that can be done. If pure JS could support features like shared pointers or global states, this issue would be resolved, but that does not seem likely to happen anytime soon.

The last main issue I ran into was one that is more an issue with web development generally which may present problems in an enterprise level raytracing application. When viewing the internet, the user might be using any combination of hardware, software, and browsers. This means that all of these cases must be considered when developing the program. Each browser interprets the webpage differently and may run the web workers slightly differently. For this project, this was mitigated by using the same browser and machine for all the development, but this could meant that there are compatibility issues.

6.2. Further Work

There are lots of interesting further improvements that can be made to this project. The biggest improvement would be server-side rendering of some kind, which would speed up the runtime and also free up the local machine's resources to do other tasks while waiting for the render to finish. Another improvement would be supporting more geometry types in addition to the spheres. This would allow more interesting scenes to be produced and allow for the importing of models made in other programs. I would also like to add emissive objects, or objects that produce their own light. Currently, the project only uses the sky as lighting, which limits the scenes to outdoor scenes only.

There are also improvements that can be made to the wrapper that is given to the user. One thing I found is that it took a long time to get the positioning of the camera and scene objects just right, since you must wait for a complete render to see where the objects are in the final image. One way to get around this would be a preview rendering mode. This would be a much less expensive rendering method that would show generally the layout of the final image without needing to fully raytrace the scene. I would also like to allow the user to edit the scene fully from the browser, but this would require a whole other project's work of effort so it will wait until another day.

7. References and Source Code

7.1. References

[1] P. Shirley, “Ray Tracing in One Weekend,” *Ray Tracing in one weekend series*, 07-Dec-2020. [Online]. Available: https://raytracing.github.io/books/RayTracingInOneWeekend.html. [Accessed: 22-Apr-2023].

[2] *Ray Tracing*, 18-Apr-2019. [Online]. Available: https://developer.nvidia.com/discover/ray-tracing. [Accessed: 22-Apr-2023].

7.2. Source Code

The source code for this project can be found at https://github.com/ryan-kustaborder/raytracing.