WebGL Raytracer

Paul Maynard
Paul.Maynard001@umb.edu
University of Massachusetts Boston

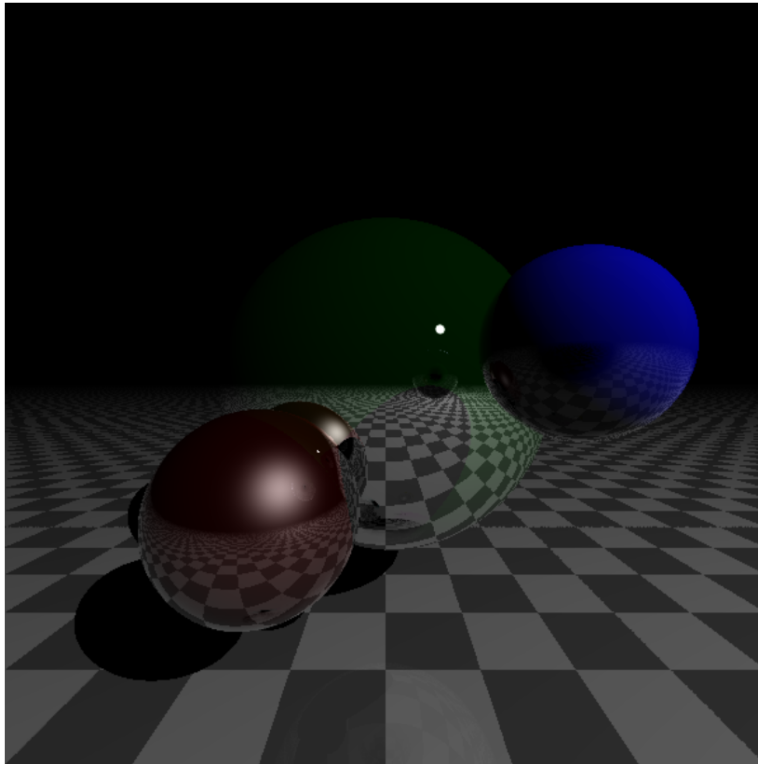

Figure 1: A simple scene rendered.

ABSTRACT

My project was to create a ray-tracing renderer in WebGL, capable of rendering simple scenes composed of graphics primitives.

KEYWORDS

WebGL, Visualization, Lighting, Raytracing

ACM Reference Format:
Paul Maynard. 2020. WebGL Raytracer. In *CS460: Computer Graphics at UMass Boston, Fall 2020*. Boston, MA, USA, 3 pages. https://CS460.org

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CS460, Fall 2020, Boston, MA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 1337.
https://CS460.org

1 INTRODUCTION

I think this project is important because it helped give me a fuller understanding of lighting calculations and methods. The project is somewhat limited - it is unable to render triangle-based geometry, because the implementation requires the geometry to be accessible from the fragment shader, and is limited to simple intersection calculations (spheres and planes). However, as a demo of code and a starting point for more advanced techniques, I think it is quite interesting, and I found it enjoyable to work on.

2 RELATED WORK

I found Scratch a Pixel's "Introduction to Ray Tracing" [2] to be a very useful resource to introduce the initial concepts for raytracing, such as shadow and reflection rays and how to calculate the cast rays.

The Graphics Programming Compendium's physically-based rendering guide [1] provided a good overview of the formulas I would need to implement PBR lighting calculations.

3 METHOD

The renderer is capable of rendering objects with physically-based materials, specified in a configuration file. It takes that configuration and passes relevant data through to the shader, which implements the following ray tracing algorithm:

```
for each pixel:
    color := 0
    pos := camera position
    ray := pixel ray
    cast ray from pos, if it hits something:
        color += lighting (lambert + cook-torrance)

        set new pos/ray to intercept and reflection

        reduce contribution of further iterations
        by Fresnel reflection factor
```

3.1 Implementation

One thing I did to make the code easier was to create a function for passing certain kinds of objects into WebGL structs, allowing the shader to easily read object and material data:

```
let struct: Struct = {
    diffuse: shape.diffuse,
    ...
};
uniformStruct(gl, program, `u_shapes[${i}]`, struct);
```

Once in the renderer, we cast a series of rays, one for each reflection iteration (the number is determined as part of the scene file and compiled into the shader).

The raycasting itself is fairly simple: the equation to calculate the intersection of a ray and a sphere or plane is pretty simple to solve, and once solved can simply be calculated:

The simpler of the two is intersection with a plane, defined by a point $\vec{c}$ and a normal vector $\vec{n}$ is:

$$\vec{n} \cdot (\vec{p}_0 + t\vec{v}_0) = \vec{n} \cdot \vec{c}$$

Where $\vec{p}_i$ is the camera (or reflection) position and $\vec{v}$ is the direction of the ray being cast.

$$\begin{aligned} \vec{n} \cdot \vec{p}_0 + t(\vec{n} \cdot \vec{v}_0) &= \vec{n} \cdot \vec{c} \\ t(\vec{n} \cdot \vec{v}_0) &= \vec{n} \cdot \vec{c} - \vec{n} \cdot \vec{p}_0 \\ t &= \frac{\vec{n} \cdot (\vec{c} - \vec{p}_0)}{\vec{n} \cdot \vec{v}_0} \end{aligned}$$

From there the intersection point can be found by

$$p_1 = \vec{p}_0 + t\vec{v}_0,$$

and the new ray can be found by reflecting it around the normal vector of the plane.

Before the reflection ray is calculated, a shadow ray is cast to determine if the area is in light, and if so, the lighting is calculated using a very simple version of the Cook-Torrance BRDF:

$$r = c_d(\vec{n} \cdot \vec{l}) + c_s \frac{DGF}{4(-\vec{r} \cdot \vec{n})}$$

Where c_d, c_s are the diffuse and specular colors of the material, $\vec{l}$ is the direction of the light, and D, G, and F are the distribution, geometric, and fresnel components of the specular reflectance. A similar version of the fresnel component of this equation is used in the reflection attenuation calculations to determine how much the reflected ray should be taken into account.

Finally, all these components, the lighting and reflection, are added together to get the final color of the pixel.

3.2 Milestones

How did you structure the development?

3.2.1 Milestone 1. I created the initial renderer as a side project while trying to learn some graphics concepts, and eventually chose it from a few other things I had tried because I thought it was the most interesting. I had actually implmented reflection calculations before lighting, since I didn't really understand the lighting concepts yet.

3.2.2 Milestone 2. I implemented a lighting model into the canvas-based renderer and started laying the groundwork for the WebGL version (it took me an embarrassingly long time to realize that I was putting my data in an attribute instead of a uniform and that's why it wasn't working).

3.2.3 Milestone 3. I fully implemented the WebGL renderer and brought it up to feature parity with the canvas one, and then implemented more complex (and physically accurate) lighting calculations.

3.3 Challenges

Describe the challenges you faced.

- Challenge 1: Porting the code. Because I didn't feel comfortable enough with the restrictions of shader code (like no recursion!) and I wanted easier debugging, I wrote my initial code in pure javascript. There are still a few artifacts of that method in the GLSL version, and I suspect that if I had started it out purely in shader code I would see better performance. Of course, without comparative the ease of debugging of javascript, I may never have gotten that far
- Challenge 2: Floating point issues. My initial version of reflection nad shadow code failed to exclude the current object from it's caclulations, resulting in errors where the reflection or shadow ray would simply hit the same object again: In this image you can also see lighting on the back face of the object, an issue that was caused when I messed up the specular lighting calculations.
- Challenge 3: Optimization. Initially I didn't think I had gotten any speed improvement in the WebGL version, but once I scaled the canvas up I realized that the GPU renderer took almost the same time, while the old canvas one, still running in parallel, was unable to keep up. Despite this, it still takes over a tenth of a second to render my simple scene, though I'm sure I could improve that by optimization.

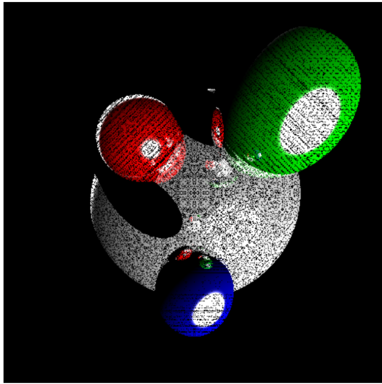

Figure 2: Floating point errors in action.

4 RESULTS

The final result is a fairly simple scene, but I have to say I think it looks quite nice. If I had had more time, there are a lot of features I would like to add, but I think the final result is pretty good on it's own. The project is not hosted in my cs460 student repository, but can be found at https://ironcretin.github.io/raytracer/ for the demo, and https://github.com/IronCretin/raytracer for the source. It currently renders a (rather low fps) animation of rotating around the simple scene depicted in the header image.

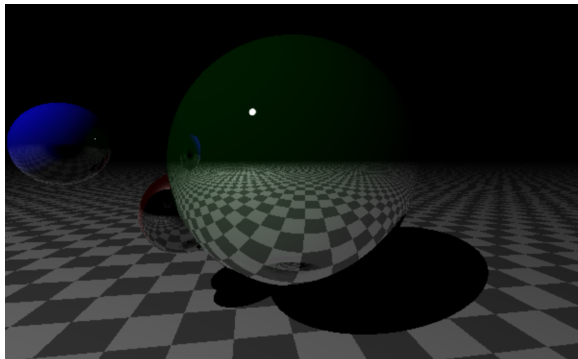

Figure 3: Another angle on the scene.

5 CONCLUSIONS

I really learned a lot, both about how to do semi-general purpose computing in GLSL (it is graphical, obviously, but I'm not taking proper advantage of all the graphical API's available to me). I also learned a lot about lighting models and how they work.

Both of these are topics I want to pursue further, both from a programming perspective and a user view (I do some 3d modeling/texturing, but I've never made PBR assets before and I want to try my hand at making some now that I know how they work.

I may also expand the capabilities of the renderer if I feel like going back to it later (it started out as a side project, and there's no reason it can't go back to being one, it's nowhere close to being "complete"), such as adding more robust lighting calculations, more primitives, and perhaps a way of loading regular models or files meant for other raytracing software.

I also looked into pathtracing, but ultimately the added complexity wasn't worth it for this project, but that is definitely something I want to try out, if not in this renderer, in another project, especially since I know some of the statistical techniques I learned in other classes can be applied.

All in all, this project, and this class, have been an illuminating (no pun intended) experience, and I am glad for the chance to delve into a topic I was somewhat unfamiliar with, both as a programmer and mathematician.

REFERENCES

[1] Wood Zoe. Dunn, Ian. [n.d.]. Physically-Based Rendering. https://graphicscompendium.com/gamedev/15-pbr.

[2] Scratchapixel. [n.d.]. Introduction to Ray Tracing: a Simple Method for Creating 3D Images. https://graphicscompendium.com/gamedev/15-pbr.