

Tony Gupta

10 December 2025

CAP5705 Computer Graphics

FINAL PROJECT REPORT

LINK TO CODE

<https://github.com/guptat07/unity-gpu-raytracing>

INTRODUCTION

The goal of this project was to explore how real time ray tracing can be implemented on the GPU. I took what we implemented in Assignments 1 and 2 and got it to work within the modern graphics pipeline. I saw a noticeable performance gain; four spheres rendering 9 reflective bounces still ran flawlessly (I could not notice any stuttering). This is a huge leap forward from our CPU ray tracer.

I think a project of this nature is the natural next step following what we have covered in class. We saw ray tracing's elegance and fidelity; we saw the graphics pipeline's processing capabilities; we should see how we can combine them.

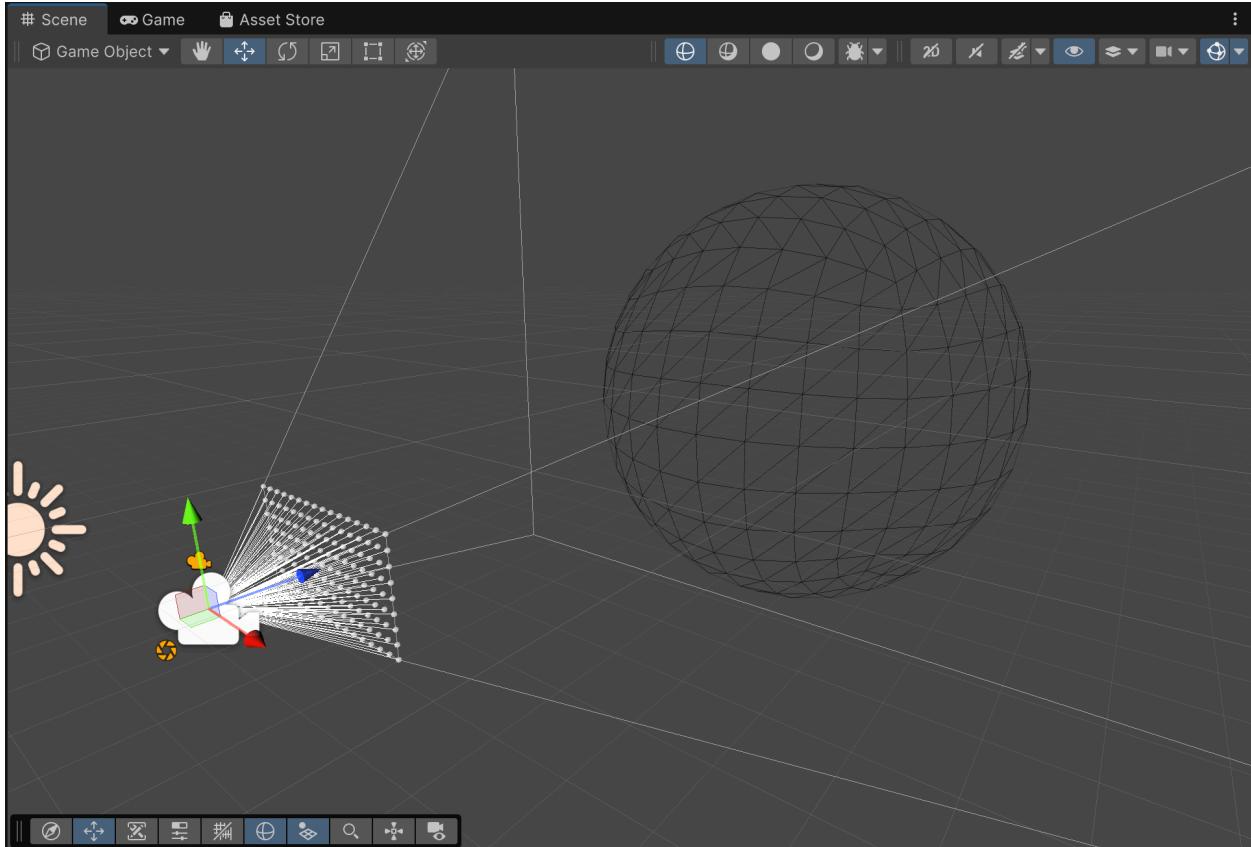
I used Unity to implement this project as it will allow me to create scenes easily. This ended up being kind of a mistake, but I still think it went a little smoother than going without Unity as there are more resources on how to go about accomplishing the goals of this project in Unity than there are in raw C++. Unity also implements things like cameras and spheres for us so that we don't have to start from zero, which is nice. We simply have to modify the behavior of these things.

There were associated challenges with learning HLSL and the Unity workflow for preparing and dispatching shaders, but I found it doable to some extent. I have never been great with documentation, and Unity's is kind of a mess when it comes to some things (e.g., I still cannot understand how to display the skybox because Unity documentation does not make it clear how to send a cube map to the shader). To minimize shader challenges, I used the Built-in Render Pipeline.

IMPLEMENTATION

Sebastian Lague's video on GPU Ray Tracing [1] was a boon when starting this project. It helped me understand the overall structure to follow; I was able to understand where to write scripts and what to attach them to. He suggested creating a RayTracingManager script to attach to the camera, which made things really easy by putting all the preparation and shader dispatch code in one place. Following along with

his code, I was quickly able to whip up this visualization representing how rays will go through the projection plane:



I was able to learn about some aspects of Unity that I hadn't previously known about, like Attributes, which let you provide additional information about structs, variables, etc. I was able to set minimum values and ranges on my public variables (which are exposed to the editor's inspector view).

Shaders are shaders. There are some differences in how you go about doing things, but ultimately, you put data in a buffer and ship it off to the GPU. For the spheres, I use a StructuredBuffer, since there can be many of those, but for directional lights, I used an array with the condition that there is a hard limit on the number of lights you can add (defined within the shader file).

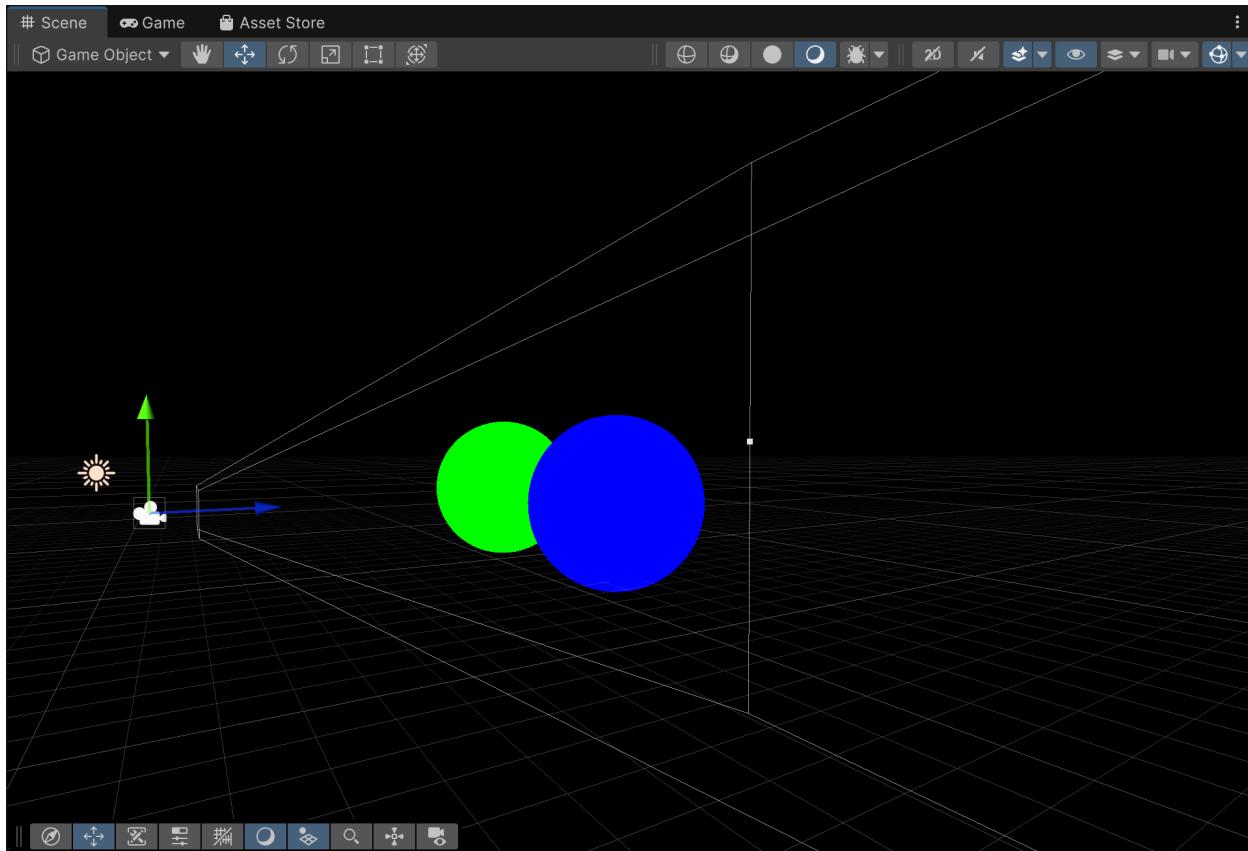
Unfortunately, the term “shader” is overloaded in Unity and refers to objects as well as the programs. Shader objects hold shader program(s). Most of the work is done with/by Material objects, which need references to Shader objects. You use the Material to set things in the shader program. So, for example, you can call SetVector(); you pass a string in and Unity will look for that variable with that name in the shader program to pass data there (just like OpenGL glGetUniformLocation). Material objects basically hold the shader programs WITH the data that they will receive. Ultimately, when we use

Graphics.Blit() to draw, we pass in the RayTracingMaterial, which we have constructed with a RayTracingShader Shader object, which currently is just the material-less given “Unlit/Texture” shader.

With this set up, we can move on. We will need to take our spheres and pass them all into the shader with buffers (this part is like Assignment 3 and 4) and the shader will use the spheres to do the closest intersection test (like Assignment 1 and 2).

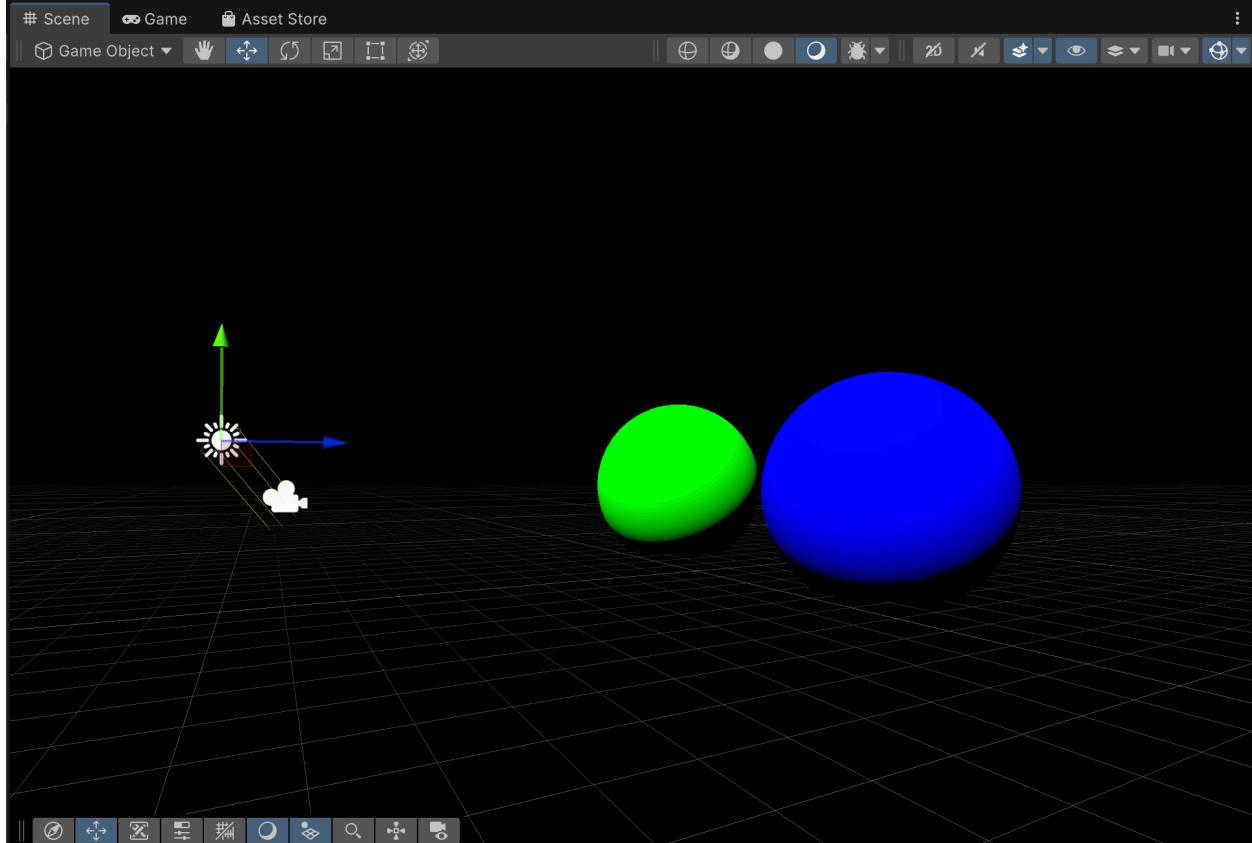
C# needs structs, not classes—it is not simply a matter of public/private. C# structs are contiguous memory. When you create a struct, you need to mark it [System.Serializable] to be able to modify it in the inspector when you use the struct in other places.

With this information, we can create a sphere struct again. This is not a sphere object; it is just the data for the calculation generated from the sphere objects. We already have our ray-sphere intersection code and HitRecord struct from before. We just translate those to C# and create equivalent structs in the shader.



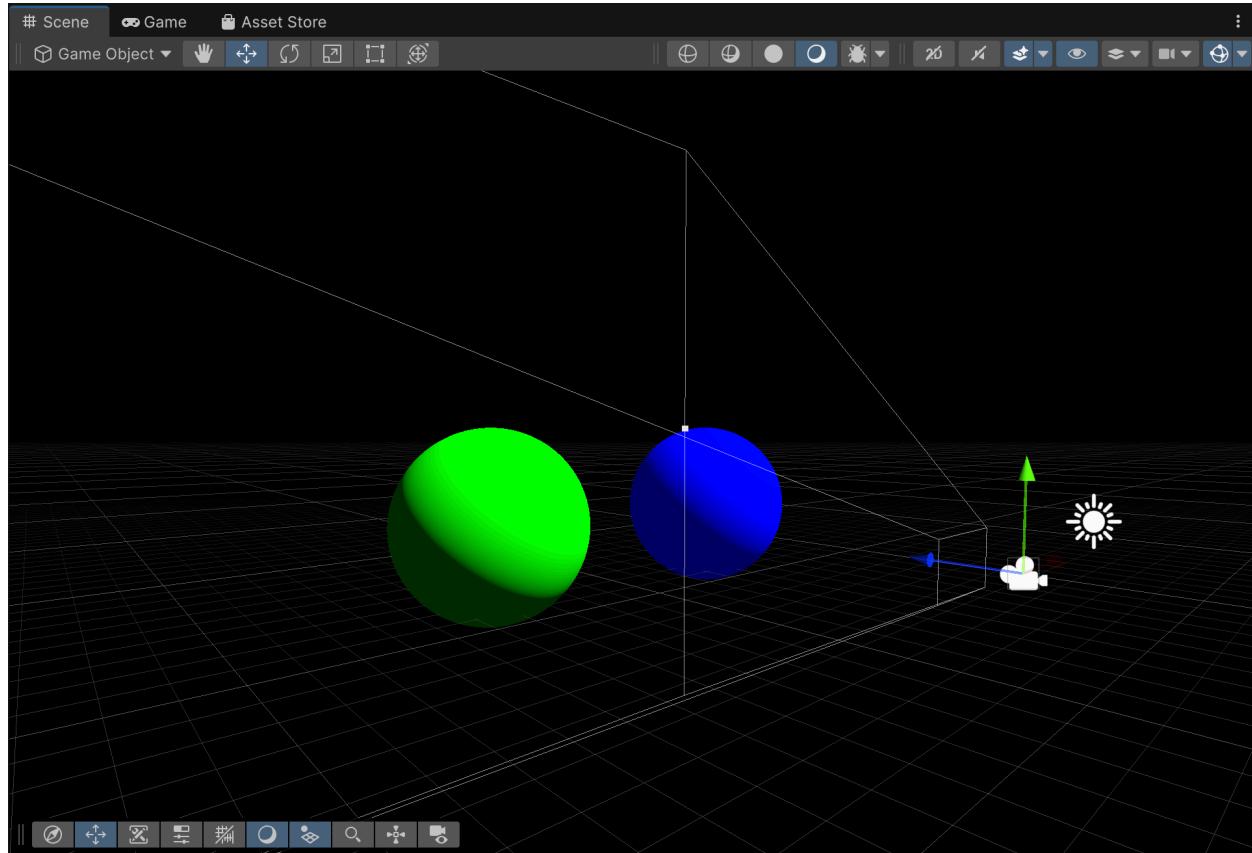
At this point, Assignment 1 was finished. This was harder than it should have been due to the learning curve, but now I understood how to do this work. Moving on to the shading, I had a good understanding of what to do, but Lague's video became less helpful from here on out as he took a different approach than what I was interested in.

Just like I needed something to attach to the sphere objects so that I could collect them and pass them to the shader, I needed a small data tag to attach to the lights in the scene. I sent the data over as arrays, and implementing the shading calculation on the shader was simple.

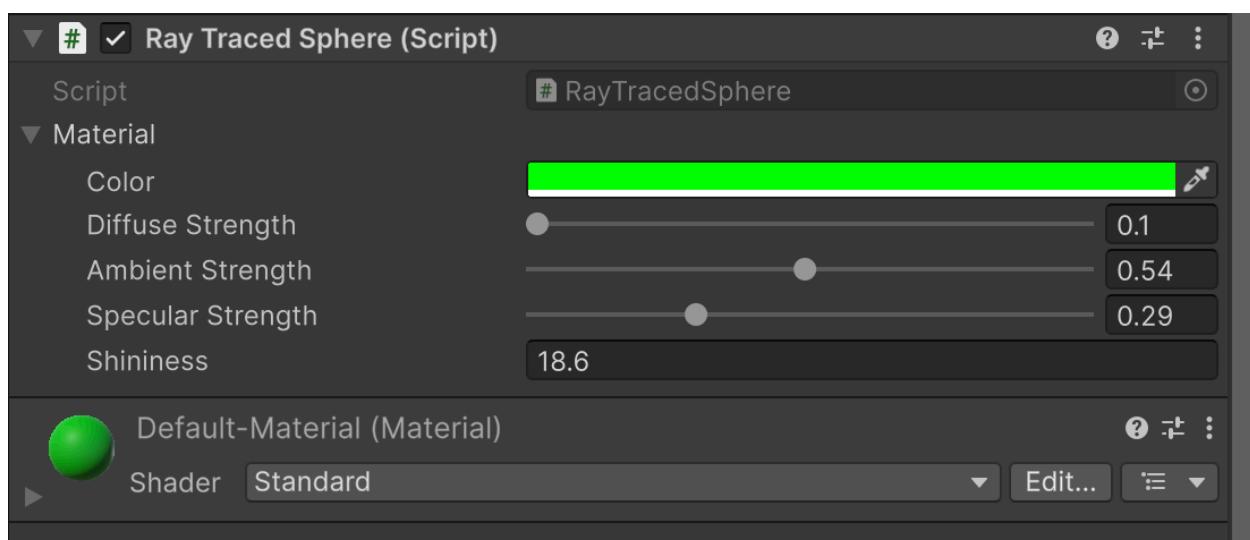
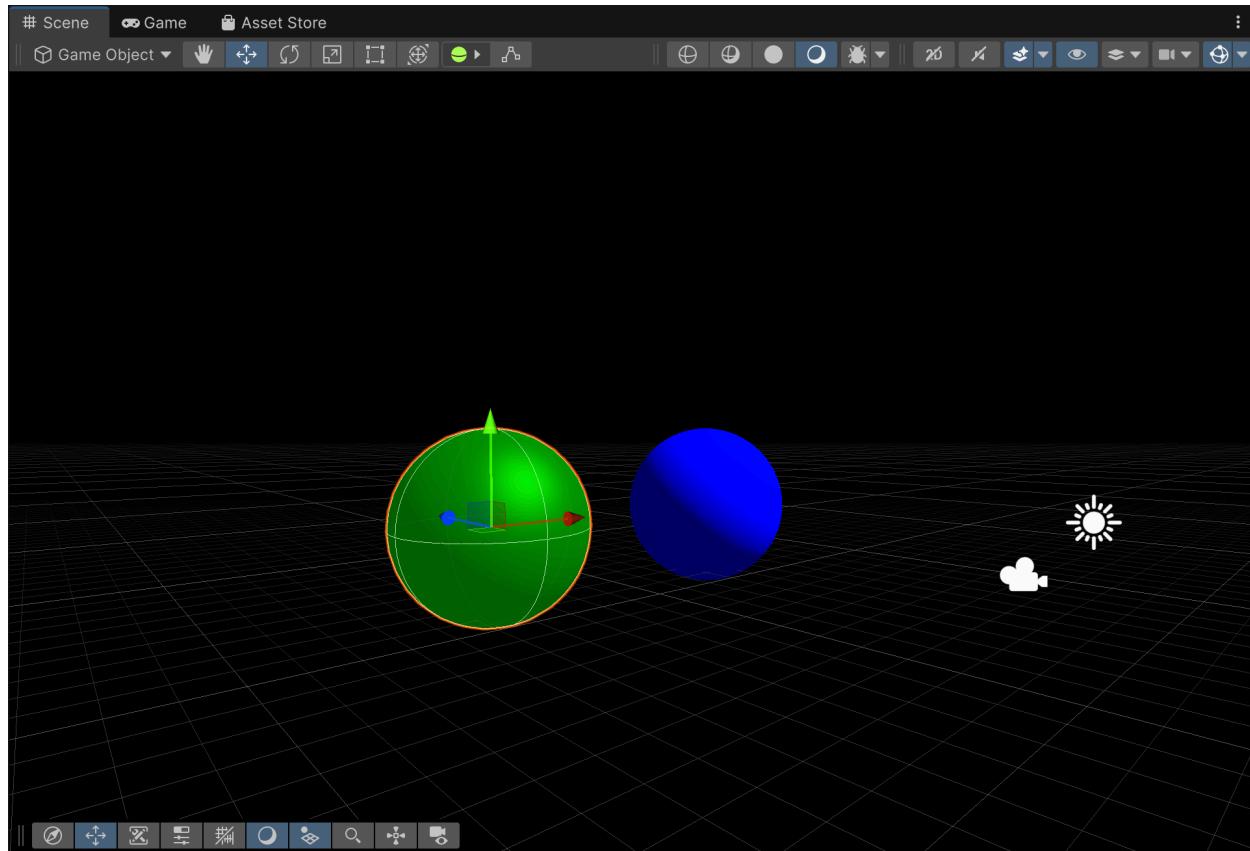


Moving on to ambient shading, I felt like trying to add a skybox to remove my spheres from the black void. I found a CC0 HDRI at [2] and converted it into a cubemap at [3]. Adding it to my scene was easy enough—this even had the benefit of giving me an ambient light color automatically that already matched this map. However, since my viewport renders the output of the shader, I needed to get the skybox into the shader.

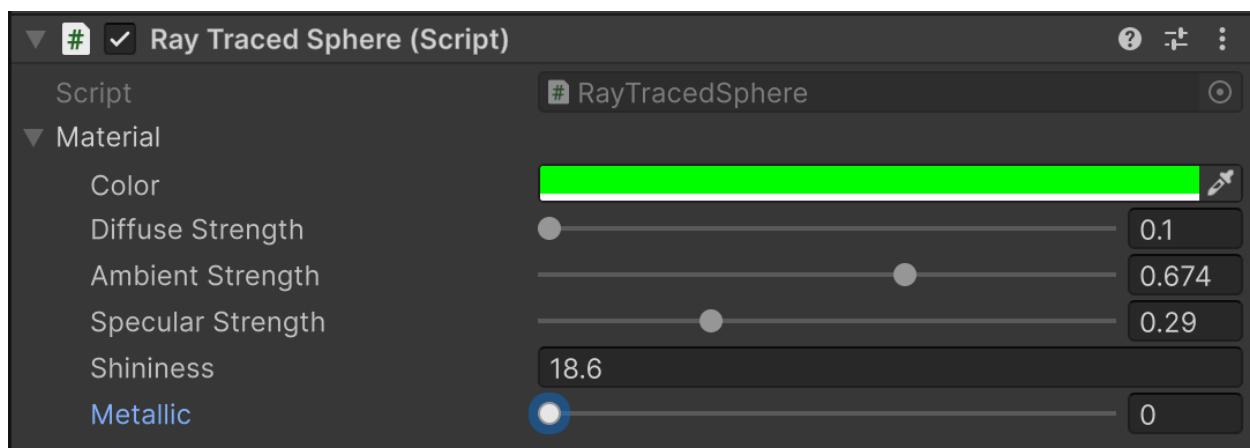
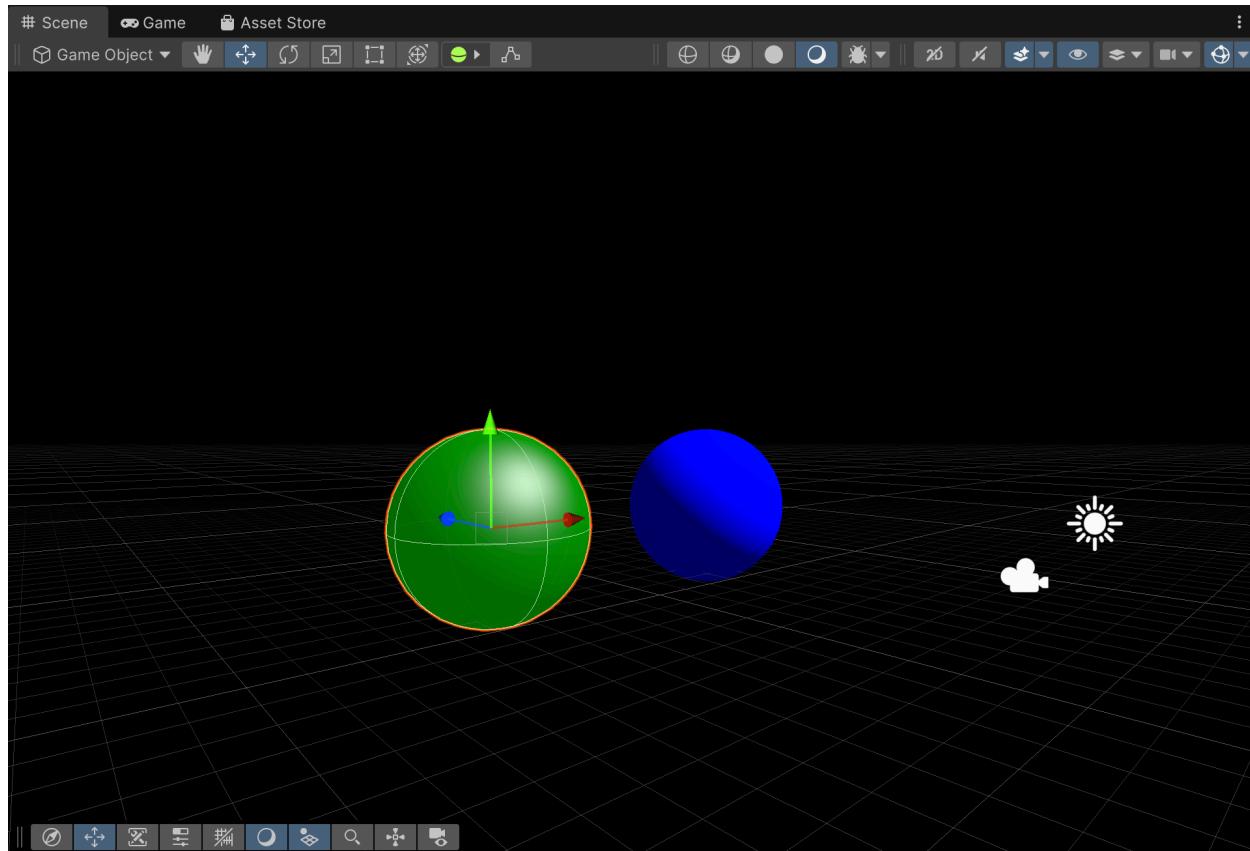
I couldn't really figure out how to do this, so I abandoned it to do the ambient lighting shading first. Ambient light took a very short time to implement since there's no real calculation. After my diffuse contribution from each of my lights is added up, I just add the ambient contribution to the end.

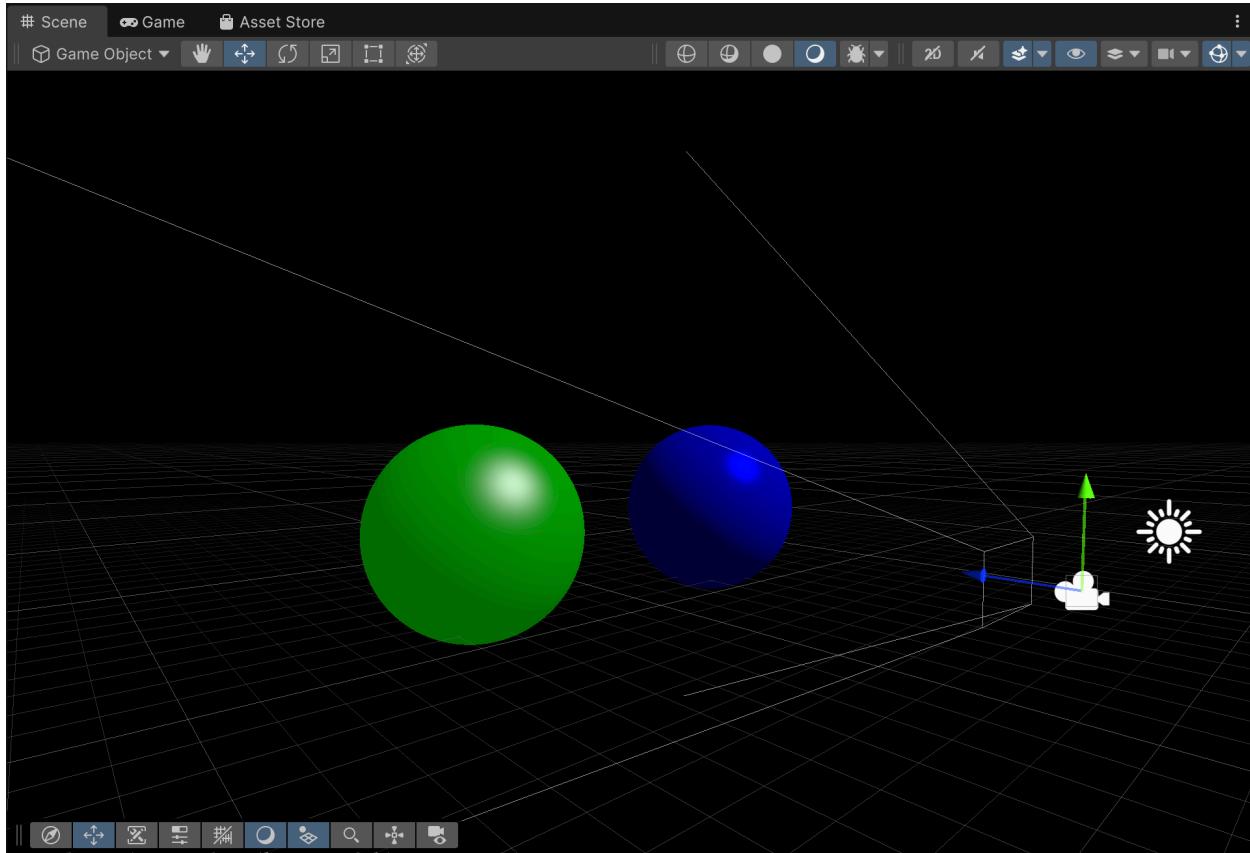


Finishing off shading, specular reflection just adds onto our diffuse calculation. We only need to add a parameter for specular strength and then use it (we already have the formula for this from Assignment 2) in our shading function.



I wasn't happy with this look. I first realized that it was metallic-only, so I added the option to choose if the object gave a metallic specular highlight or not. Strangely, I had to use an int that was limited to 0 or 1 because bools can't be passed via the StructuredBuffer.

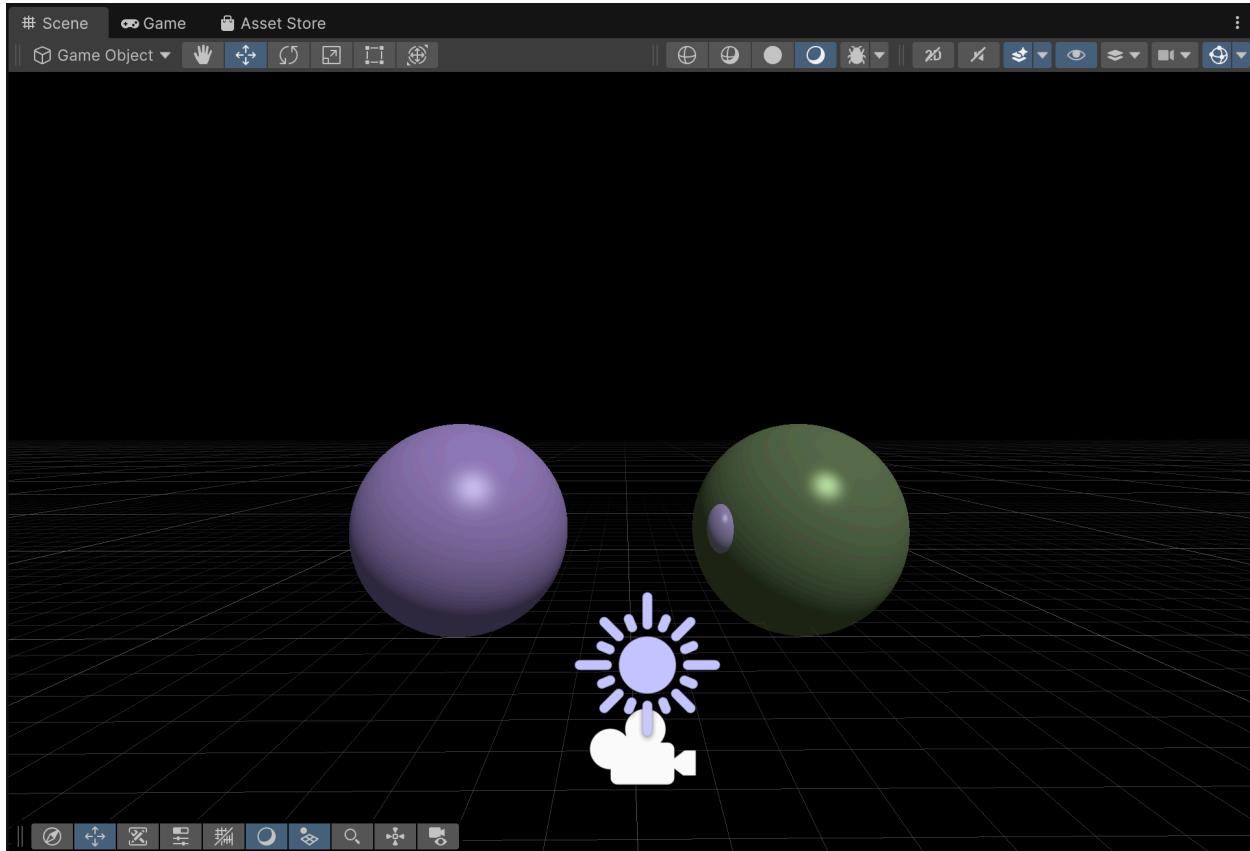




This was almost all of Assignment 2 done. I have some pretty nice-looking spheres and the performance is rock-solid. However, as with the blue sphere on the right, it is metallic without a reflection. Time to implement the last aspect, which is also the most challenging.

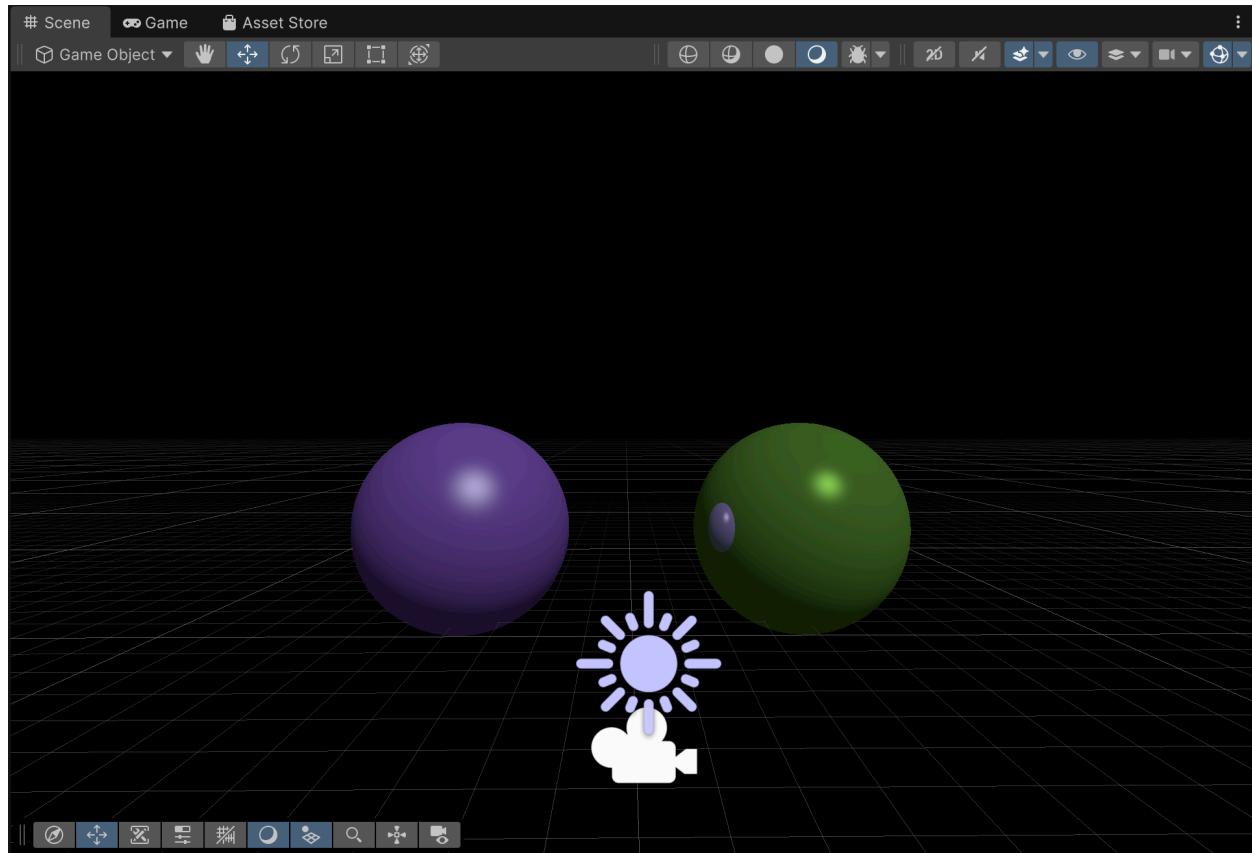
Since fragment shaders cannot use recursion, I had to find another way to limit the number of bounces (Even in my CPU ray tracer, I had a bounce limit, but the code did make a true recursive call) and get the logic to work. I created an editor-modifiable parameter (which I capped at 10 but could likely exceed) to adjust the number of bounces that I passed into the fragment shader. Then, inside the fragment shader, I have a for loop that executes as many times as the parameter is set to.

On each bounce, we calculate the regular shading by calling the shading function. Then, we create our reflected ray. The origin is the intersection point and we calculate the direction using the formula from Assignment 2 (although HLSL had a built-in function that I used). Then, simply enough, we assign this reflection ray to our original ray, so that the next loop calculates the reflection from that surface. The last thing we need to account for is how kMirror in our CPU calculation tends the reflection value towards zero. Due to the way the recursive call is set up, kMirror is essentially raised to the number-of-bounces power. I handle this by multiplying the material's mirror strength by itself each loop.

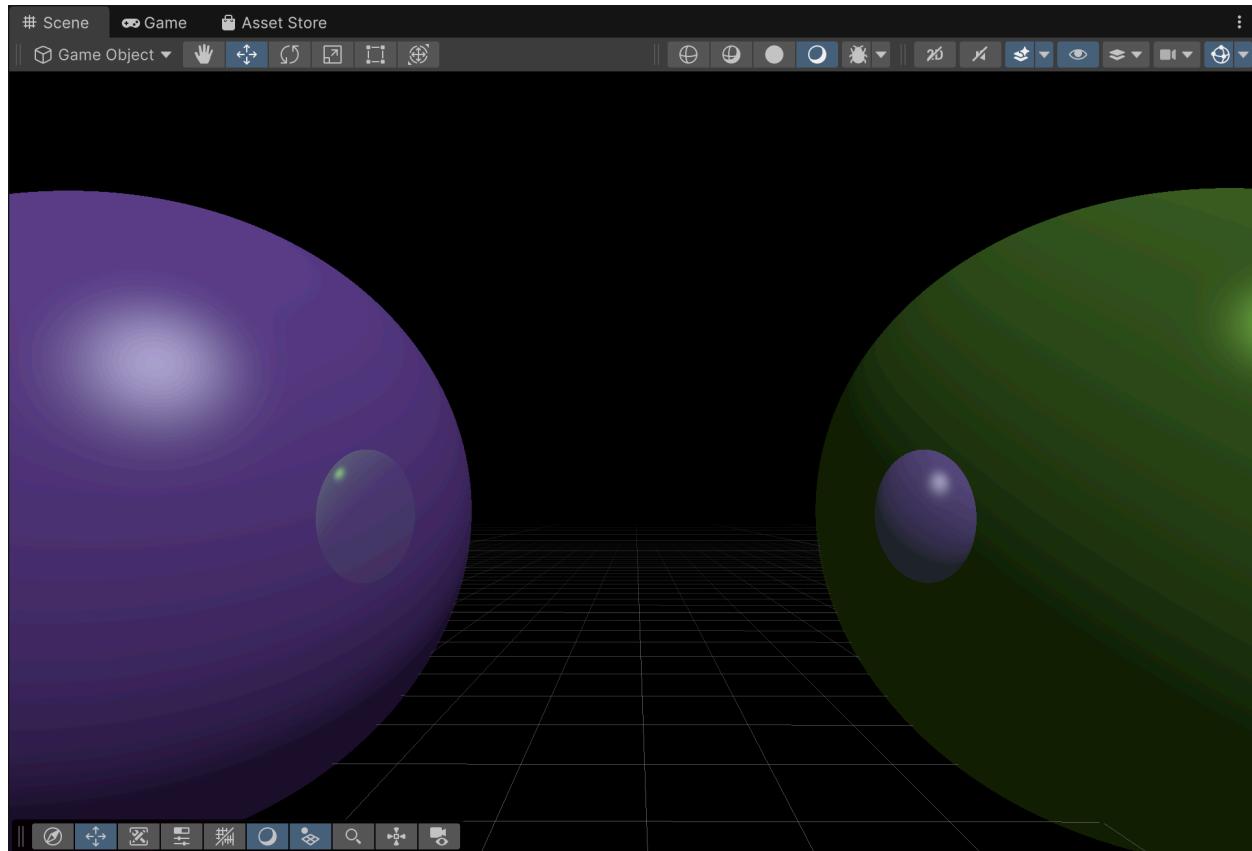


Reflection was working, meaning I had completed Assignments 1 and 2 in the GPU with a fragment shader at this point. But I still wasn't satisfied with the look, so I went in to tweak some more things.

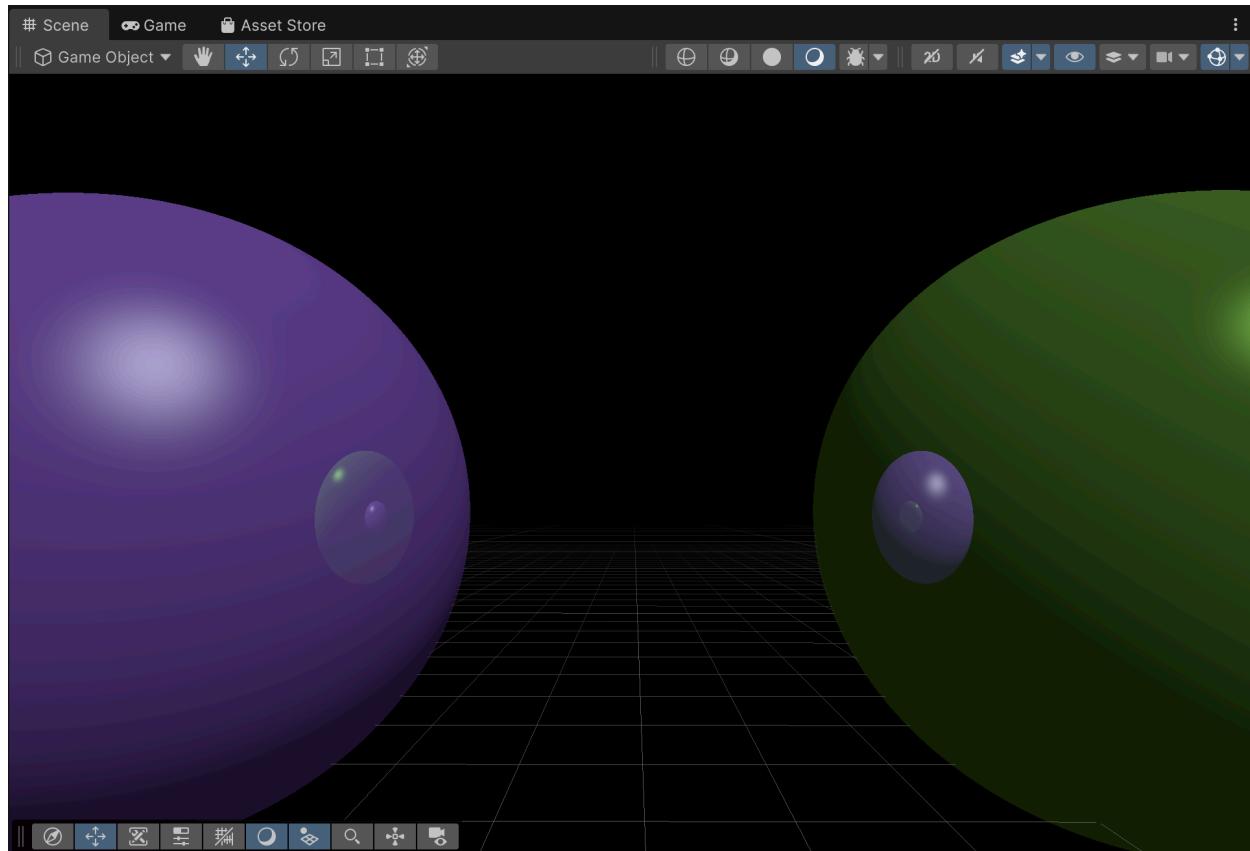
I picked the exact same colors I used in the CPU assignment, but as you can see, they looked really washed out. This was irritating me—I spent far too long on what turned out to be a color space issue. Unity is in linear color space by default, while our CPU ray tracer was Gamma 2.2. We can quickly correct this in code [4].



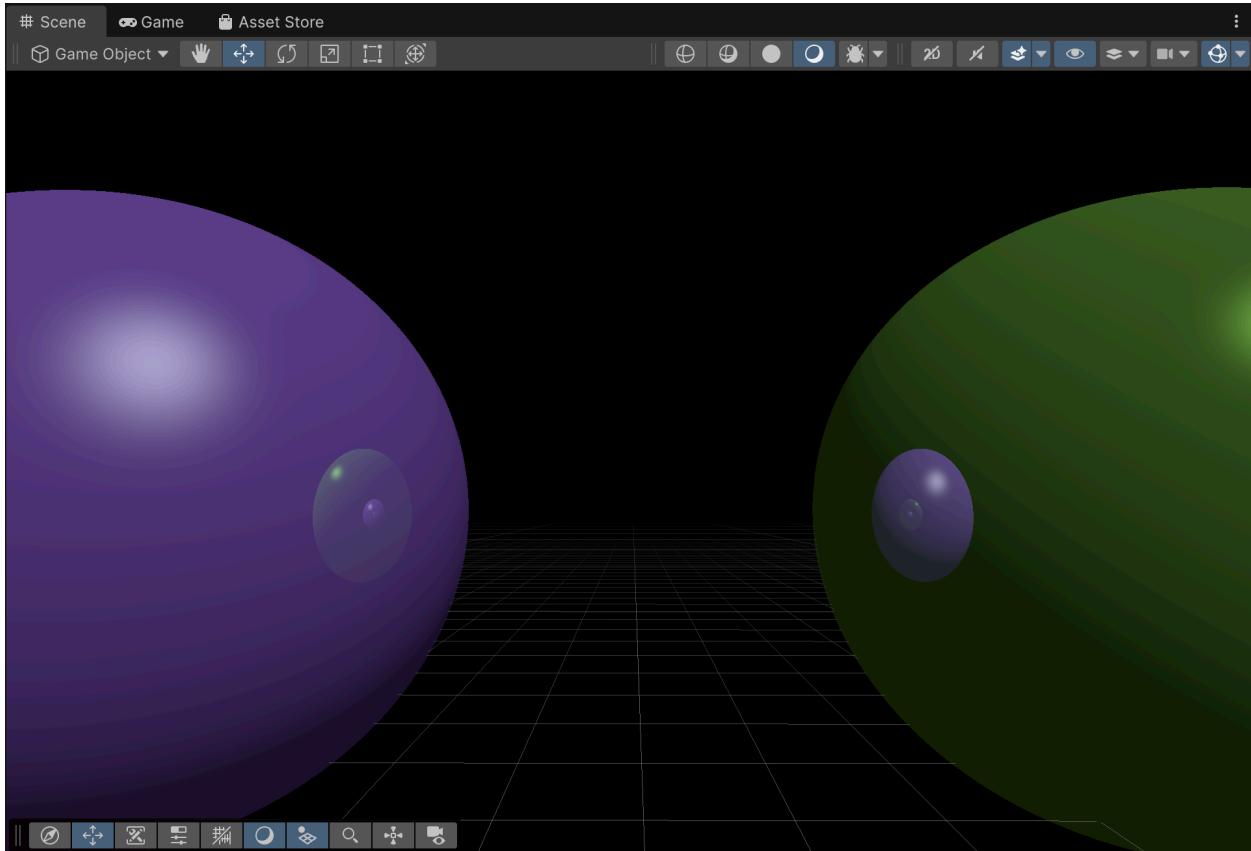
This looked much better, and the colors are closer to the values from my original ray tracer. From here, I realized that it was making no difference whether I had 2 bounces or 10. This was because I forgot to make the purple sphere on the left reflective. I made this change and took a few screenshots to illustrate the change in reflection depth:



One bounce (the parameter is set to two, so there is one level of reflection).



Two bounces.



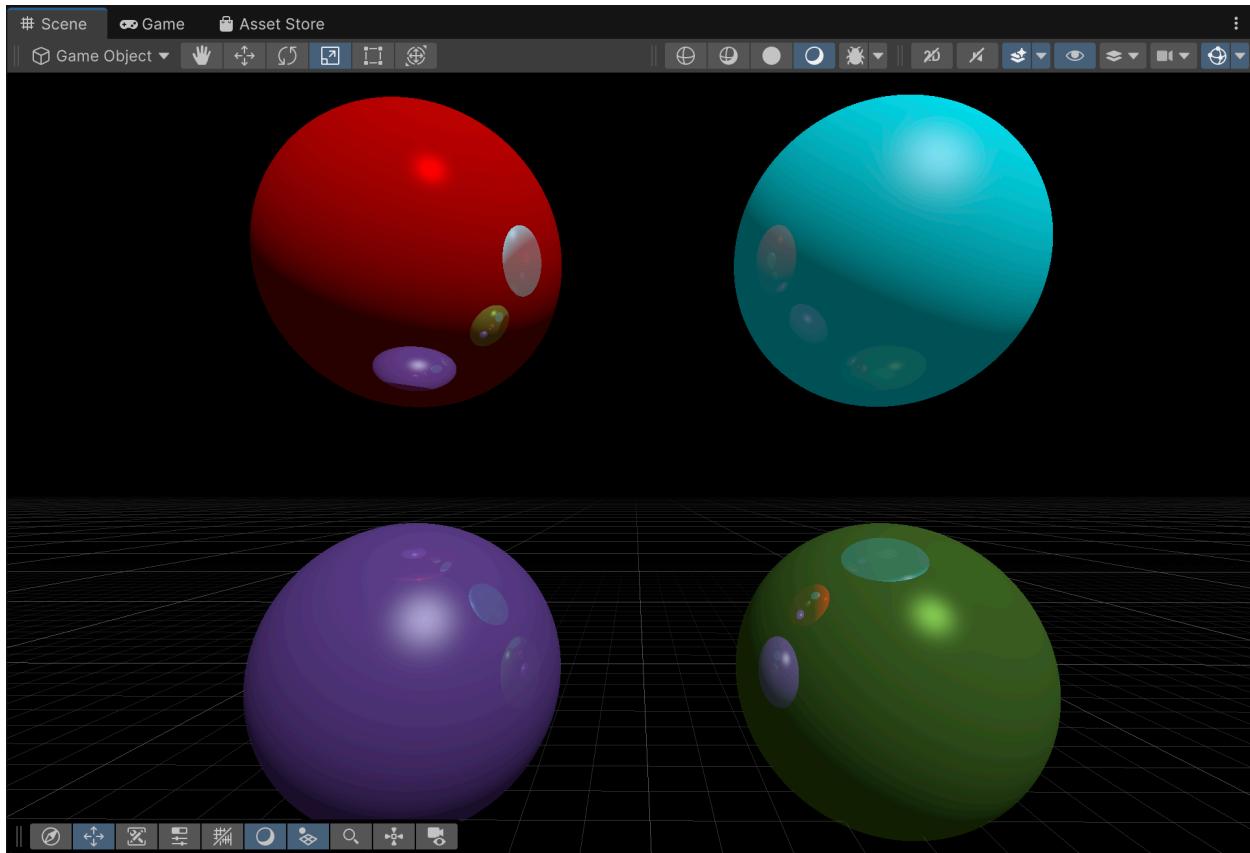
Nine bounces.

The coolest part about having nine reflective bounces was that there was an imperceptible change to performance—if any at all—and this ran in real-time (especially compared to the CPU ray tracer, which needed ages to render out an animation with smooth motion).

At this point, I felt that I had accomplished my stated goal. Since Unity did not provide intersection testing for me (as I had hoped when proposing the project), I tried and failed to successfully implement ray-triangle intersection. I am still not sure what the problem is, as I think the logic is sound.

So, I went back to the one thing that eluded me before: the skybox. Ultimately, I was not able to figure this out, either. What's frustrating is that I understand both how to pass data to the shader in Unity AND how to use the skybox (it is a cube map, so we use our ray to sample it if it misses the spheres).

I decided the best thing to do to end this project would be to try and do a minor stress test on my program by adding two more spheres with the maximum bounce count set.



It still runs flawlessly in the editor view. I particularly like that red material. I think there is a good foundation here for even more tinkering, and I would like to return to this project in my spare time to continue building.

Obviously, the first things I would add are the skybox and ray-triangle intersection (and the associated work required to break a model down into triangles, etc.). From there, I would return to [1] to understand Lague's approach to modeling reflection, as it seemed to be a lot more resource-intensive than my approach even for simple effects. I also know that he is able to extend it to create a more sophisticated result with more interesting effects, including glass-style refractions; I would love to tackle those.

REFERENCES

- [1] Sebastian Lague. 2023. Coding Adventure: Ray Tracing. Video. (1 April 2023).
<https://www.youtube.com/watch?v=Qz0KTGYJtUk>
- [2] Greg Zaal. Shanghai Bund HDRI. https://polyhaven.com/a/shanghai_bund
- [3] Mateusz Wisniowski. HDRI to Cubemap. <https://matheowis.github.io/HDRI-to-CubeMap>
- [4] pabloandinopla. 2023. Response in RenderTexture's color look washed out.
<https://discussions.unity.com/t/rendertextures-color-look-washed-out/927043>