168 Final Report:

Environment Lighting and My Personal

Ray Tracing Renderer

By Tiane Maestas (A15948372)

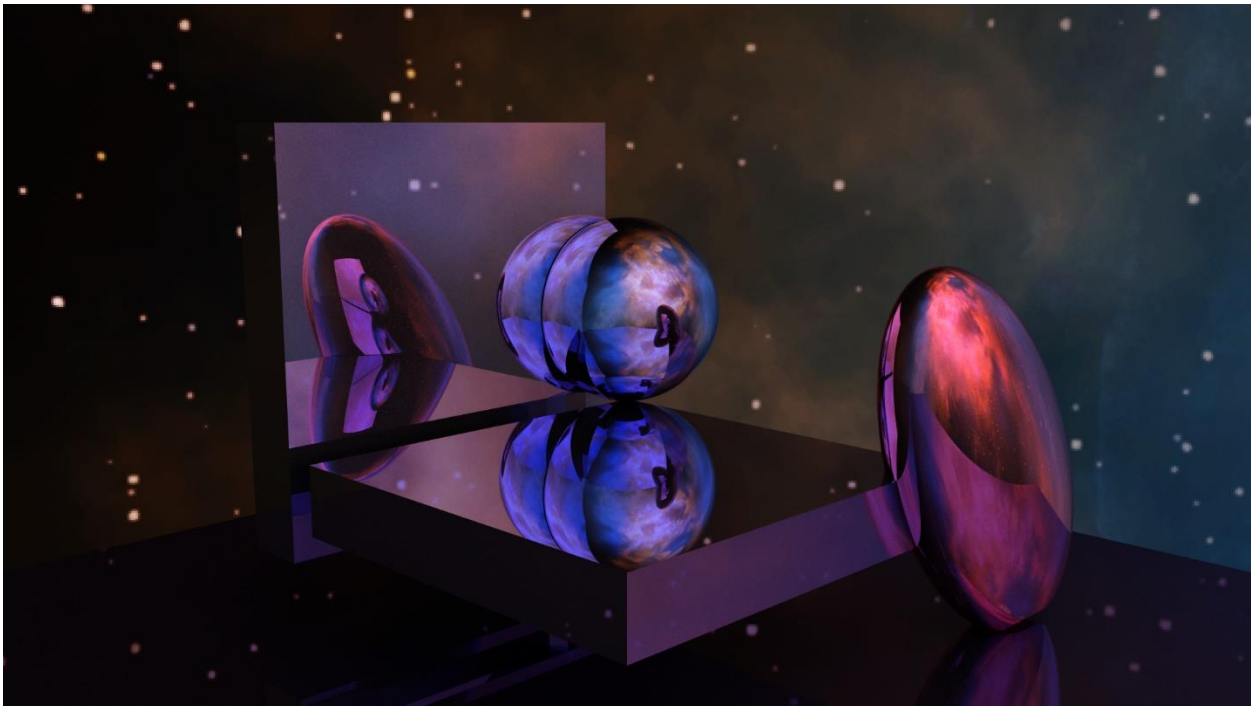

Figure 1: Ray Tracing Primitives lit by a Nebula.

1 Introduction

I started this renderer over spring break in preparation for this class. It can be found on my GitHub here. The goal of this project was to build a comprehensive understanding of raytracing and attempt to design a detailed and easy to use rendering pipeline; I used the homework as a poorly organized learning ground to build things that worked but I tried to implement a much better design in this project. With this said, this project isn't complete from a design perspective. There are always plenty of features that I could add but assuming the current set of features is complete I still would like to re-design how materials are handled and how lights are generally sampled.

From what I learned in this class I think I have found a good design for handling scenes and scene intersections but I have also learned about the depth of materials and light sampling. The way I have designed scenes and object intersections allows for no modification to the ray tracing

code and only simple implementations of my 'Traceable' interface. Furthermore, if you want to use a BVH for acceleration on something it needs to implement an 'Intersectable' interface giving more options for expansion in the future and no changes to the current pipeline. However, I know now that the same principles would be helpful if implemented for materials and lights. Currently, if new materials are added you must change the core ray tracing code and the same goes for lights. For example, every material has its own BSDF and sampling procedure that can be abstracted out of the core ray tracing pipeline. In a perfect world my renderer would be built in a way that only things directly pertaining to how rays are traced throughout the scene should be implemented in the core ray tracer such as MIS. If I had more time, I would have liked to do this before submitting my project but the best I can do is acknowledge these decisions.

The rest of this report will touch briefly on the core ray tracing pipeline I use in my renderer, my custom multithread design, and then on environment lighting since that is the only core feature that is different from the renderer, we made in our homeworks.

2 My Rendering Pipeline

Through trial and error, I think I found some good ideas when it comes to my ray tracing workflow. While this might not be very complicated it's taken a lot of time to find what was the most intuitive, easy to use, and easy to expand on for me. First let's start from the entry point to the program.

```
MyCustomSceneBuilder sceneBuilder;
std::shared_ptr<Scene> scene = sceneBuilder.BuildFromFile(sceneFile);
if (scene == nullptr)
{
    std::cout << "Scene Construction Failed." << std::endl;
    exit(1);
}

Renderer renderer(scene, sceneBuilder.GetOptions());
renderer.RenderScene();
renderer.SaveImage();
```

This is pretty much self-explanatory but it makes the flow of logic very explicit and straightforward. I build a scene and tell a renderer to render one frame then save the image.

Then, through the renderer, the following occurs.

```
if (m_numThreads == 0) // Run on the main thread.
{
    RayTracer rayTracer(&m_image, (float)m_options.ImageWidth, (float)m_options.ImageHeight);
    rayTracer.UpdateOptions(m_options);
    PixelBlock block{ 0, 0, m_options.ImageWidth, m_options.ImageHeight };
    rayTracer.TraceImage(m_scene.get(), block);
    return;
}
```

The renderer simply configures in what manner the scene should be traced based on rendering options. The above code shows the general flow of logic but is split amongst threads if told to do so. A “RayTracer” is built and is given a block of pixels in the total image to render; since this is the non-multithreaded example the pixel block is the entire image.

Then, the core ray tracing pipeline occurs all in the ray tracer “TraceImage” method.

```
vec3 pixelCol(0.0f);
for (int i = 0; i < m_raysPerPixel; i++)
{
    float xoffset = (i == 0) ? 0.5f : random_float(m_randomGenerator);
    float yoffset = (i == 0) ? 0.5f : random_float(m_randomGenerator);
    Ray ray = RayThroughPixel(scene, x + xoffset, y + yoffset);
    Intersection intersection = FindIntersection(scene, ray);
    pixelCol += FindColor(scene, intersection, 0);
}

(*m_imagePixels)(x, y) = pixelCol / (float)m_raysPerPixel;
```

For each pixel in the given “PixelBlock” the above is executed. Generate a ray, intersect with the scene, and calculate a color given the intersection…simple.

“FindIntersection” either calls intersect on the scene’s BVH or loops over all of the “Traceable” object stored in the scene depending on configuration. What makes the design behind my intersection more interesting is that the scene and every triangle mesh has its own BVH; so, when the scene checks to see if it is intersecting with a leaf triangle mesh it calls the mesh’s BVH. Currently, you can create any number of layers of BVH as you desire (i.e. the meshes could have other objects within them that have their own BVH and so on) and I personally really like this freedom of implementation choice. Primitive objects such as spheres and ellipsoids, for instance, don’t have their own BVH by design but if I were to implement any other “Traceable” object in the future I could choose whether its own BVH is needed.

3 My Multithreading Solution

I had some fun coming up with my own multithreading solution rather than using parallel for loops like we did in the homeworks. My design is inspired by a thread pool architecture of sorts.

```
void RayTracer::TraceImage(Scene* scene, const std::vector<PixelBlock>& blocks)
{
    m_thread = std::make_shared<std::thread>(&RayTracer::TraceThreadedImage, this, scene, blocks);
}

void RayTracer::TraceThreadedImage(Scene* scene, const std::vector<PixelBlock>& blocks)
{
    for (const PixelBlock& block : blocks)
    {
        TraceImage(scene, block);
        if (ProgressReport::ReportProgress) { ... }
    }
}
```

Every “RayTracer” has an overloaded “TraceImage” method that takes in a list of pixel blocks. Therefore, multithreading is enabled by the “RayTracer” simply by passing a list of pixel blocks rather than a single pixel block. It’s like reserving a list of jobs for a thread pool to complete. In my case, the renderer simply traces an image on separate “RayTracers” for each list of blocks it creates and each list will be on a separate thread. For a more detailed look on how it works check out the “RenderScene” method in the renderer. In the end, I had fun coming up with my own solution.

4 Environment Lighting

Like I said in the introduction and the project check-ins I started this project over spring break and worked slowly throughout the quarter on updating it. Overall, a lot went into getting my final images for this project submission but I will only elaborate on the environment lighting feature since it is the most significant difference from our homework renderers.

At a high level, I simply load images as textures and “surround” the scene with them. I currently support both cube maps and sphere maps. I treat the lighting from these maps essentially as directional lights and simply query the texture given an outgoing ray direction from an intersection point (assuming the direction isn’t blocked and is in shadow). It was quite annoying finding the right standards for image maps that I could find online but eventually followed the same standard as Unity since I had some free asset skyboxes that I could re-purpose for my images. This implementation can be found in the skybox “Query” method.

Diving deeper, I used what we learned in homework 4 to build the skybox sampling method. Currently, I am doing cosine lobe sampling around the z-direction and transforming the direction sample to the surface normal basis of the intersection. Then given the direction and PDF I sample the skybox and perform lighting calculations as I normally would for a directional light.

```
// Uniform Random Sample an outgoing direction on cosine weighted hemisphere.
float r1 = random_float(m_randomGenerator);
float r2 = random_float(m_randomGenerator);
float z = sqrt(1 - r2);
float phi = 2 * pi * r1;
float x = cos(phi) * sqrt(r2);
float y = sin(phi) * sqrt(r2);

vec3 direction(x, y, z);

// Convert to surface normal basis.
vec3 u, v, w;
w = intersection.normal;
v = (fabs(w.x) > 0.9f) ? vec3(0.0f, 1.0f, 0.0f) : vec3(1.0f, 0.0f, 0.0f);
v = normalize(cross(w, v));
u = cross(w, v);

direction = normalize(direction.x * u + direction.y * v + direction.z * w);

float PDF = dot(w, direction) / pi;
```

The major downside of this sampling method is that it takes more time because I must increase the samples per pixel for the image to converge. It isn't extremely unreasonable though because it does converge relatively quickly. The dragon scenes that I rendered are only 50 spp and a resolution of 1920x1080. They take about 27 seconds to render. The dragon has a little over 300,000 triangles.

Figure 2: Four Dragon Scenes

5 Conclusion

Finally, I just want to say that this has been an extremely fun and educational experience. This is definitely the most complicated project I have ever worked on and I am extremely proud of what I could produce. Seeing how the time of day in the environment maps change the lighting on the dragon in the smallest of ways and make the scenes feel so much more real is awesome. It's my new desktop wallpaper lol. Thank you!

Figure 3: Dragon at Midday

Figure 4: Dragon at Sunset

Figure 5: Dragon at Dusk

Figure 6: Dragon at Night