CPU Based RayTracing

Kieran Bowsher
19036291

University of the West of England

April 5, 2022

The task set out was to create a cpu based ray-tracer that could render multiple objects with different materials, textures and lighting. Techniques like Bounding volume Hierarchy and multi-threading have been implemented to improve the ray-tracers performance.

1 Introduction

Ray-tracing has been around since the 60s and has made big advancements since then with systems like RTX allowing for real-time ray-tracing. There are lots of examples online of CPU based ray-tracer like Ray Tracing In One Weekend (Peter Shirley, 2020) and ScratchaPixel (scratchapixel, 2022). This project will be looking and implementing the many systems that are ubiquitous to ray-tracing.

2 Implementation

This is the main part of the report where each element that was implemented into the ray-tracer is discussed. The ray-tracer was developed using Visual Studio 2019 and includes the stb-image library for loading and saves images (Nothings, 2022) and the Assimp library (Kulling, 2021) for loading 3D meshes.

2.1 Overview

These are the main features of the ray-tracer:

- Move-able camera
- Custom Mesh loading
- Texture loading and saving
- Bounding Volume Hierarchy
- Lighting
- Different Materials
- Multi-threaded
- Tile-based rendering
- Physically based atmosphere rendering

With all of these features, the ray-tracer can produce images with custom meshes and textures place around the scene with a modest performance in render times.

2.2 Camera

Very advance ray-tracers and path-tracers simulate digital cameras for rendering. For projects like this one, a pin hole camera works fines. The camera used in this project allows a user to set its position, direction, image size and field of view (fov).

When a camera is defined in a 3D world, there is still the issue of visibility. One way and the most common way is to solve visibility is to produce a ray sometimes called a primary ray or visibility ray and cast it from the camera to a pixel and when this ray hits something, that pixel gets coloured in. There is another method of solving visibility and that is to use rasterization. However, for simplicity the first solution works fine.

2.3 Ray-casting

In order for a primary ray to have a direction towards a pixel. Each pixel first has to be converted from pixel/rasterizer space to world space as explained by (ScratchaPixel, 2022). Converting pixels from rasterizer space to world space is expressed as:

$$AspectRatio = \frac{ImageWidth}{ImageHeight}$$

$$Scale = \tan(\frac{fov * \pi}{180} * 0.5)$$

$$PixelCamera_x = (2 * PixelScreen_x - 1) * AspectRatio * Scale$$

$$PixelCamera_y = (1 - 2 * PixelScreen_y) * Scale$$

where $PixelScreen$ is expressed as:

$$PixelScreen_x = 2 * (\frac{Pixel_x + 0.5}{ImageWidth}) - 1$$

$$PixelScreen_y = 1 - 2 * (\frac{Pixel_y + 0.5}{ImageHeight})$$

Note that $Pixel_x + 0.5, Pixel_y + 0.5$ allows the ray to pass through the center of a pixel. Changing 0.5 to a random number generator would create different offsets for rays which is good for anti-aliasing and path tracing. Using these converted pixels positions and the forward vector of the camera, a direction can be made for each generated ray.

2.4 Geometry

Unlike some ray-tracer online which render implicit objects like sphere, this project uses triangulated meshes that are imported using Assimp. In order to check if a ray has intersected a triangulated mesh, algorithms like Moller-Trumbore (Tomas, 1997) are used.

2.4.1 Moller-Trumbore Ray Triangle Intersection

There are two ways to find the intersection of a triangle. The first one is called the geometric solution which computes the plane from the triangle normals and checks if the ray is inside the triangles 3 vertices. More of this is explained by ScratchaPixel (ScratchaPixel, 2022a). The second way and most popular way is using an algorithm proposed by Tomas Moller and Ben Trumbore titled "Fast, Minimum Storage Ray/Triangle Intersection" Tomas, 1997. As explained by them, a point $t(u, v)$ on a triangle can be given as:

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2 \qquad (1)$$

where u, v are the components of the barycentric coordinates which have to be $u \geq 0, v \geq 0$ and $u + v \geq 1$ in order to be valid. The w component is given as $(1 - u - v)$. Computing the ray(t) and the triangle's T(u, v) is given as R(t) = T(u, v) or

$$O + tD = (1 - u - v)V_0 + uV_1 + vV_2 \qquad (2)$$

However, there are three unknown values t, u, v and as explained by (ScratchaPixel, 2022a), the equation is rearranged into row-column vector multiplication. As well, (Tomas, 1997) uses a technique called "Cramer's rule" to solve for t, u, v.

2.5 Bounding Volume Hierarchy

Ray-tracing complex scenes with lots of objects and high triangle counts can become very slow to render. As proposed by (Kay and Kajiya, 1986), the usage of an accelerator structure like a Bounding Volume Hierarchy can greatly improve rendering times. A similar idea was also proposed by (Rubin and Whitted, 1980) who noted that using an accelerator can improve rendering times.

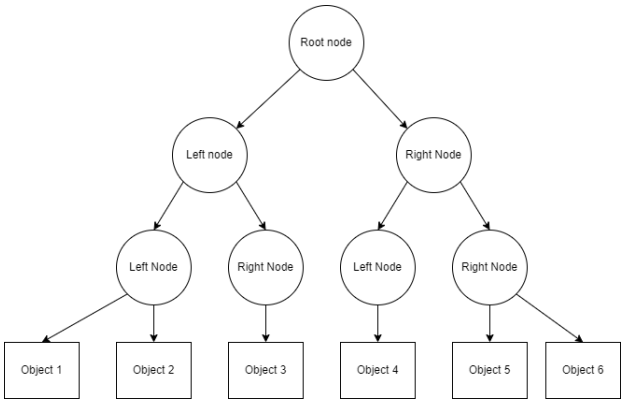

Figure 1: *An example of a bvh tree*

The premise for Bounding Volume Hierarchy is to traverse a scene and split up the objects into nodes that are then wrapped around a bounding box like an AABB. When a split happens, a node with a bounding box is created. These nodes will sort the objects into another left node and right node and generate more bounding boxes. This continues recursively until the number of objects in each list meets a requirement and once this happens the node that is created is called the leaf node which holds the objects. **Figure 1** shows an example of a node tree.

This method can also be used on the triangles of a object to greatly improve rendering times.

2.5.1 AABB bounding Box

In order to generate a bounding box around a collection of object or just a single object, (Kay and Kajiya, 1986) proposed a method which has become popular called the "slab method". The idea is to create 6 planes all facing the major axis and project a vertex from the model onto the plane. The distance between the vertex and plane is checked and this value is store as $tnear$ or $tfar$. Looping over all the vertices in the model and projecting each vertex onto all planes will yield a bounding box. This can be expressed as:

$$Ax + By + Cz - d = 0$$
$$N_xP_x + N_yP_y + N_zP_z - d = 0$$
$$d = N_xP_x + N_yP_y + N_zP_z$$

where Ax, By, Cz are the normals of the planes and P being the vertex.

2.5.2 Middle split vs Surface Area Heuristics

Deciding how the scene or the object triangles are split is very important to the performance of the bvh. Most implementation of a bvh tend to use either mid point split or surface area heuristics. Surface area heuristics tends to provide the best performance for a bvh. This

project doesn't use surface area heuristics but instead uses mid-point split due to it easier implementation.

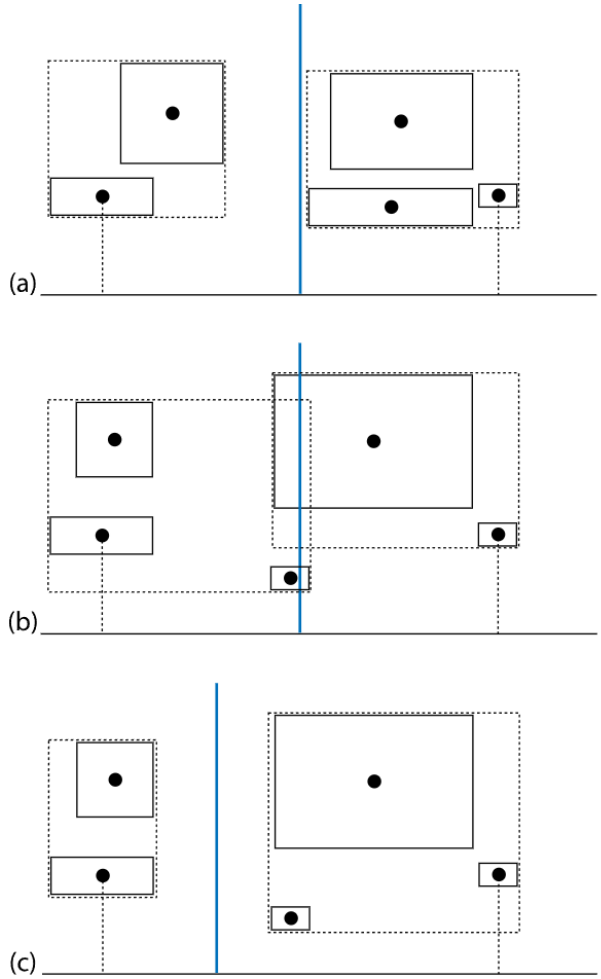

Figure 2: *A diagram from (Humphreys, 2021) showcasing midpoint split in three different scenarios*

Mid-point split works by taking the bounding box that surrounds all the objects and uses the longest axis of that bounding boxes centroid and sorts the objects based on that axis. Once all the objects have been sorted into another left node and right node, another bounding box is made for those nodes and midpoint-split is used again. As mentioned earlier, this will continue to happen recursively until a condition is met.

While this is fine for now, there are cases where mid-point split works poorly and why surface area heuristic should be used. As shown in **figure 2** and explained by (Humphreys, 2021), midpoint split works fine when the objects are divided equally on both sides as shown by **figure 2 (a)**. However, if there is a lot of overlap like **figure 2 (b)** shows, then midpoint split will work poorly. Finally, as **figure 2 (c)** shows, if the split was moved to the correct position for splitting, then all the objects will be split correctly.

2.6 Lighting the scene

This project mainly uses these 3 components for lighting which are

- light transport algorithm
- shadow rays
- shading

2.6.1 Whitted light transport algorithm

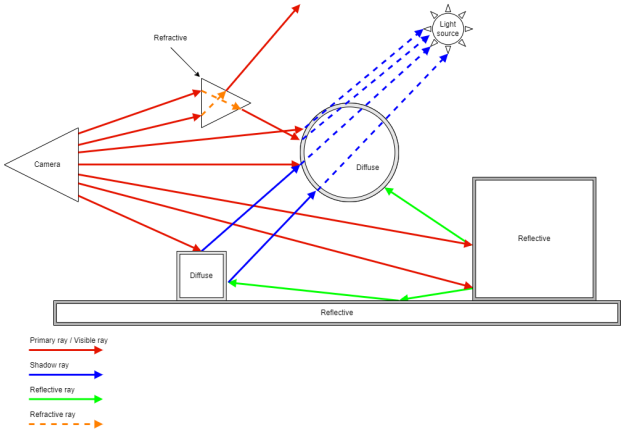

Figure 3: *A diagram showcasing an example of recursive ray tracing*

The project can be technical described as a recursive ray-tracer since each primary ray can cast other rays like reflective, refractive and shadow and have these rays continue to cast recursively until a condition is met.

This type of light transport has been in ray-tracing since the 80s with Turner Whitted (Whitted, 1980) being one of the most popular. Whitted's paper has inspired other works like distributed ray tracing and unbiased path tracing. This project uses Whitted light transport algorithm to allow objects with reflective and refractive surface to render realistically.

In programming, this light transport algorithm is just a switch case with a set of cases that are based off different surface which a ray could hit. If a ray hits a reflective surface, then the case which holds the reflective code is used and generates another ray. In turn that ray is sent into the scene and potentially hit another surface. If it does, then the calculation for another surface is done and another ray can be generated. If the ray doesn't hit anything, then the background is rendered and the process stops for that ray. This process can go on forever due to its recursive nature but checks like depth can be used to limit the number of recursions made. **figure 3** show this process.

2.6.2 Shadows

Rendering shadows in ray-tracing is slightly more different than other methods of rendering like rasterization. In rasterization, as explained by (rastertek,

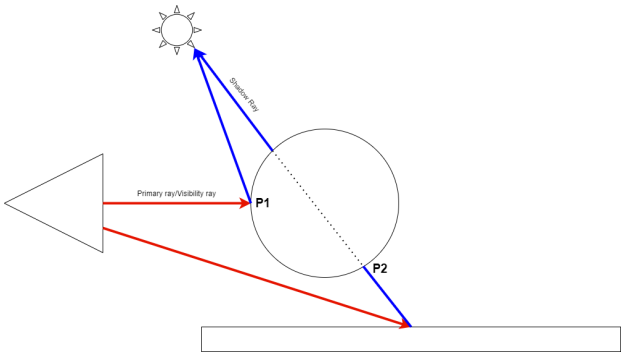

Figure 4: *A diagram showing the process of rendering shadows using shadow rays*

n.d.), a basic form of rendering the shadows is to capture a texture called the "shadow map" at the point and direction of the light source. This "shadow map" which is similar to a depth map is then compared with the position of the light and camera on a per-pixel bases. Doing this will produce very basic shadows.

Ray-tracing is a little different since the observer casts rays into the scene and this can be used for shadowing. Rendering shadows in ray-tracing is done when the primary ray or visibility ray hits an object, another ray called the shadow ray is created at the intersection point with a direction towards the light. As shown by **Figure 4**, when a shadow ray is created at intersection point of the sphere **P1** it reaches the light source without hitting any objects. This means that P1 is illuminated.

However, this is different for the second shadow ray which is created when it hits the plane. This ray hits the sphere at the intersection point **P2** which means that it obscured by the sphere. When this happens, the algorithm can exit early and render a black pixel for that shadow ray.

An issue called shadow acne can arise when creating shadow rays. The issue happens due to floating point precision and the fact that the primary ray doesn't accurately hit the object but instead, hit either inside or outside the object by a small margin. This issue can be fixed by using this equation which offsets the the shadow rays origin.

$$ShadowrayOrigin = hitpoint + normal * bias \quad (3)$$

2.7 Surface Materials

The project allows multiple objects to render with different surface material. There can be objects that are reflective, refractive and diffuse. There are two separate diffuse materials called Phong and Cook-Torrance which use two methods and this will be discussed more in section 2.7.1 and 2.7.2.

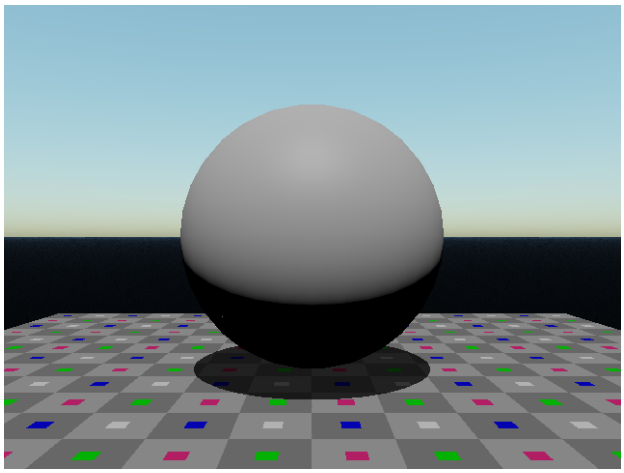

Figure 5

2.7.1 Phong Model

In order to create surfaces that have both diffuse and specular properties, the Phong reflection model was made for the project.

The Phong reflection model was developed by Bui Tuong Phong in 1975 (Phong, 1975). This reflection model has been used a lot in renders and real time applications. He noted that many materials can be computed by the sum of the weighted diffuse and specular components and that shiny surfaces have small but intense specular highlight whereas dull surfaces will have larger highlights that fall off more gradually. The equation for the Phong reflection model is shown as:

$$I_p = k_a i_a + \sum_{m \epsilon lights} (k_d(L \cdot N) i_{m,d} + k_s(R \cdot V)^{\alpha} i_{m,s}) \quad (4)$$

where k_a is the ambient value, i_a is the intensity of the ambient value. m is the light source with k_d as the diffuse reflection. $i_{m,d}$ is the intensity of the diffuse reflection. k_s is the specular reflection, α is the value of "shininess" and $i_{m,s}$ is the intensity of the specular reflection.

2.7.2 Physically Based Materials

In real life, there are mainly two types of surfaces that exist which are conductive and dialectic. Dialectic surfaces like wood, diffuse the light in different direction while conductive surfaces like metal reflect the light back giving metals a mirror like appearance. To allow this behaviour to exist in computer graphics, a Bi-Directional Reflectance Distribution Function (BRDF) is used. There are many BRDF models like Phong reflectance model (Phong, 1975). For this project, Cook-Torrance (Robert, 1982) was used since it used heavily within real time application due to it low performance cost and allows for objects to have different surface types like dialectic, conductive or a mix of both.

The Cook-Torrance BRDF can be describe as:

$$f_r = \frac{FDG}{4 \cdot (N \cdot L)(N \cdot V)} \tag{5}$$

where FDG are the micro-facet functions like the Normal Distribution Function D, the Geometry function G and the Fresnel function F.

Applying the BRDF into the reflectance equation looks like this:

$$L_o(p, \omega_o) = \int_\Omega (k_d \frac{c}{\pi} + \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}) L_i(p, \omega_i) n \cdot \omega_i d_{\omega i} \tag{6}$$

The Schlick Fresnel approximation is used for the Fresnel function F and can be explained as:

$$F(\theta) = F_0 + (1 - F_0)(1 - N \cdot V)^5 \tag{7}$$

where F_0 is the specular reflectance at the normal incidence and this value is depended on the index of refraction. In most implementation of this model, F_0 is usually valued as 0.04.

Disney's GGX/Trowbridge-Reitz is used for the normal distribution function D and is calculated as:

$$D(h) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2} \tag{8}$$

with α as the surface roughness.

For the geometry function G, GGX and Schlick-GGX is used and is described as:

$$G(n, v, \alpha) = \frac{n \cdot v}{(n \cdot v)(1 - k) + k} \tag{9}$$

with k is the remapped value of α.

$$k = \frac{(\alpha + 1)^2}{8} \tag{10}$$

As noted by (Burely, 2012), $(\alpha + 1)$ is a modification done to reduce roughness "hotness". Finally, Smith's method combines the two Schlick-GGX function to create the geometry function.

$$G_{Smith}(n, v, l, a) = G(n, v, a)G(n, l, a) \tag{11}$$

The BRDF is implemented similar to how it would be implemented inside a pixel shader in that this BRDF only shades the pixel where the model lies in and these effects can be seen in **figure 6**.

2.7.3 Reflective Materials

Rendering reflective materials is very easy to do due to the laws of reflection (britannica, n.d.). In real life, when a ray of light hits a smooth object, it bounces off in the equal opposite direction. The ray hitting the surface is called the "Incident ray" θ_i and when that ray bounces off the surface its called the "Reflected ray" θ_r. For this to work in code, the following equation is used:

$$\theta_r = \theta_i - 2(N \cdot \theta_i)N \tag{12}$$

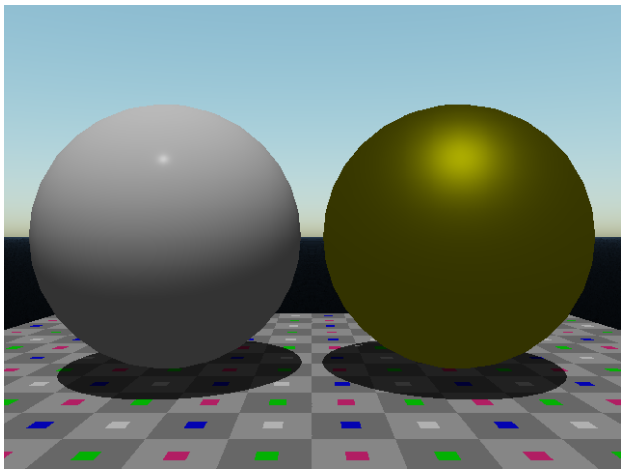

Figure 6: *Left: object with roughness set to 0.2. Right: object with metallic set to 1.0 and roughness set to 0.5*

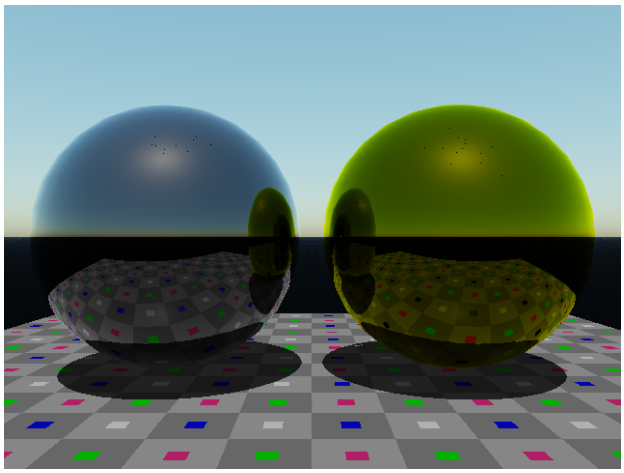

Figure 7: *Two metallic spheres next to each other causing a recursive reflecting effect to happen.*

Using this equation will produce the results as shown in **figure 7**. Do note that in closer inspection of the image, there are recursive reflection happening. This mainly happens due to the castRay() function which is recursive and allows multiple reflective rays to be cast. However, this can be an issue since the rays can technically be cast an infinite amount of times. An easy fix is to have a variable called $depth$ which iterates every time a reflective ray is cast and check if it is equal to the max_depth which in this example is set to 3.

2.7.4 Refractive Materials

Similar to reflective materials, refractive materials bend the incoming light based on a value called the Index of Refraction. Refraction is based off Snell's law which explains that for a pair of media, the ratio between the sines of each angle of incidence θ_1 and the angle of refraction θ_2 are equivalent to the opposite ratio of the indices of refraction. This can be expressed

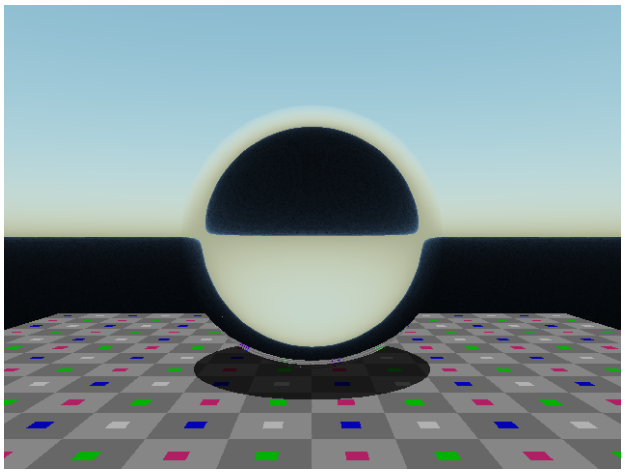

Figure 8: *A sphere with an index of refraction value set to 1.33 causing the light to refract inside the sphere*

as:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{\eta 2}{\eta 1} \tag{13}$$

where θ_1 is the angle of incidence and θ_2 is the indices of refraction.

As explained by (Greve, 2006), calculating refraction is expressed as:

$$t = \frac{\eta_1}{\eta_2} i + (\frac{\eta_1}{\eta_2}\cos\theta_i - \sqrt{1 - \sin^2\theta_t}) \tag{14}$$

$$\sin^2\theta_t = (\frac{\eta_1}{\eta_2})^2 \sin^2\theta_i = (\frac{\eta_1}{\eta_2})^2(1 - \cos^2\theta_i) \tag{15}$$

Computing this gives the results as shown in **figure 8**.

2.8 Physically Based sky-box

Figure 9: *Render showcasing the usage of a sky-box (Wenson, 2014)*

In most cases for ray-tracing, the background or sky-box as it called is usually just a solid colour. However, to add more visual detail to the renders, a textures of the sky is used. Normally, these textures would be a unwrapped and applied on a mesh sphere or an implicit sphere and each ray that hits this sphere would sample the pixel and render it to screen as shown in **figure 9**.

However, there are other method proposed by (Nishita et al., 1993) and (A.J, Peter, and Brian, 1999) which simulate the colours of the sky using an algorithm. As mentioned by (ScratchaPixel, 2022b), the model proposed by (A.J, Peter, and Brian, 1999) is restricted as it only works for the observer on the ground and not in the air or outer space.
The implementation of the physically based sky-box in the project mainly follows the work of (ScratchaPixel, 2022b) who base their implementation off (Nishita et al., 1993).

2.8.1 Atmosphere

The atmosphere on most planets like Mars or Earth have a layer of gas surrounding them called an Atmosphere. The characteristics of the atmosphere is generally defined by its thickness and composition (the different elements that make up the atmosphere).

As mentioned by (ScratchaPixel, 2022b), simulating an atmosphere for a renderer only accounts for the two layers which are called the troposphere and stratosphere. When light rays travels through the atmosphere, the rays get deflected and scattered by the particles in the atmosphere which is a phenomenon called scattering. This scattering is controlled by the particle size and density. Since the atmosphere is also affected by the planets gravity, in turn so do the particles. This means that at sea-level, there is more particles near the sea-level than there are particles 60km above sea level.
Papers like (Nishita et al., 1993) make the assumption that the density decreases exponentially with height and this can be described as:

$$density(h) = density(0)e^{-\frac{h}{H}} \tag{16}$$

2.8.2 Phase functions

In order to simulate the scattering of the light in the atmosphere, phase functions are used. The first one is Rayleigh, which was discovered by Rayleigh in 19th century. He discovered that air molecules scatter blue light more than green or red and this has a dependency on the wavelength strength.

This can be described as:

$$P_R(\mu) = \frac{3}{16\pi}(1 + \mu^2) \tag{17}$$

where μ is the cosine of the angle between the light and viewing direction.

Another phase function that is used is called Mie scattering. This is similar to Rayleigh with the only difference being that its only applied to particles that are larger than the wavelength like aerosols. This equation can be seen as:

$$P_M(\mu) = \frac{3}{8\pi}\frac{(1 - g^2)(1 + \mu^2)}{(2 + g^2)(1 + g^2 - 2g\mu)^{\frac{3}{2}}} \tag{18}$$

where the mean cosine g, controls the anisotropy. By default in most papers, $g = 0.76$.

2.8.3 Rendering the atmosphere and sun light

To render the atmosphere, a ray is cast from the observer to the atmosphere. Once the ray hits the atmosphere, light is sampled at intervals along the ray to compute the amount of light that has travelled along the ray due to single scattering. To do this, the volume rendering equation is used and is computed as:

$$L(P_c, P_a) = \int_{P_c}^{P_a} T(P_c, X) L_{sun}(X) ds \tag{19}$$

where L is the amount of light in the volume, T is the transmittance between the intersection point P_c and X which is the sample position along the viewing direction. This equation explains that the amount of light that reaches the observer is equal to all the light scattered along the viewing direction.
Since light is attenuated when it travels through a medium like the atmosphere, transmittance needs to be calculated and it looks like this:

$$T(P_a, P_b) = \exp(-\beta_e(0) \sum_{P_a}^{P_b} \exp(-\frac{h}{H}) ds) \tag{20}$$

This equation is a modified version of the transmittance equation as seen by (ScratchaPixel, 2022b) who explain that for Rayleigh scattering β_e, absorption can be ignored and instead written as $\beta_e(h) = \beta_s(h) + \beta_a(h) = \beta_s(h) + 0 = \beta_s(h)$.

In order to render the sunlight, the function $L_{sun}(X)$ from equation 19 is used.

$$L_{sun}(X) = SunIntensity * T(X, P_s) * P(V, L) * \beta_s(h) \tag{21}$$

Finally, to compute the sky colour which is mainly a result from Rayleigh and Mie scattering. This equation is used in combination with equation 21 as shown below.

$$SkyColour(P_c, P_a) = SunIntensity * P(V, L) \int_{P_c}^{P_a} T(P_c, X) * T(X, P_s) * \beta_s(h) ds$$

As shown by (ScratchaPixel, 2022b), this can be simplified even more since the scattering has to be calculated twice and this simplification looks like this:

$$SkyColour(P_c, P_a) = SkyColour_{Rayleigh}(P_c, P_a) + SkyColour_{Mie}(P_c, P_a)$$

Here are a few examples of the atmosphere at different altitudes and light directions as shown in **figures 10, 11, 12**.

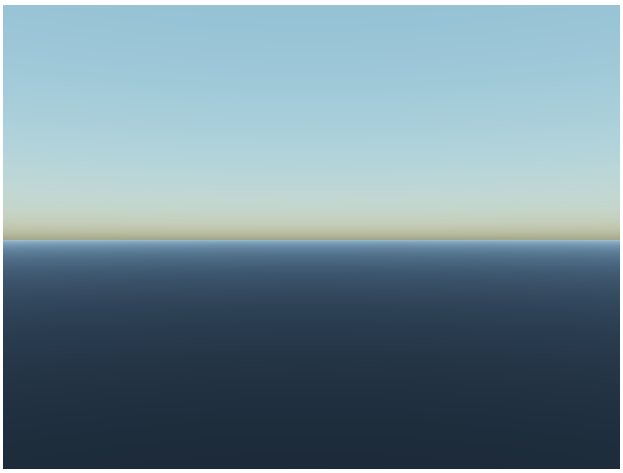

Figure 10: *A render of the sun directly above the observer*

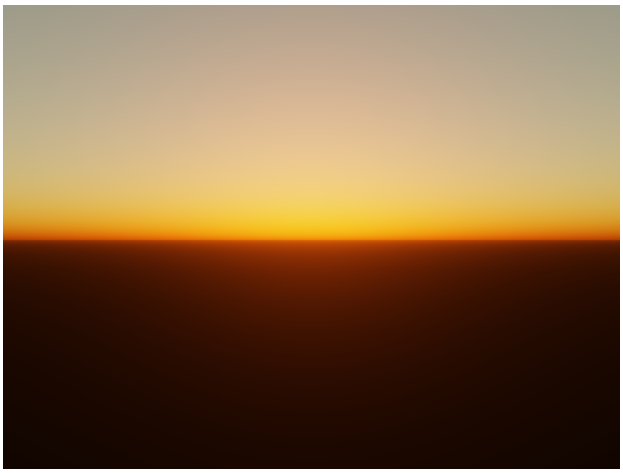

Figure 11: *A sunset*

2.9 Optimizations

2.9.1 Tile based rendering

Another solution to optimizing the performance of ray-tracing is to use a technique called tile rendering. The idea behind tile rendering as explained by (Quilez, 2008) is to break up the image into different tile which contain a number of pixel and render each tile. Usually the number of pixels within a tile is based off the width and height of that tile. For example, each tile could be 16x16 pixel meaning that each tile would contain 256 pixels as shown in **figure 13**.

2.9.2 Parallel Processing

To gain the most performance and optimization for a ray-tracer, multi-threading is used. The concept behind multi-threading is to execute parts of a software at different cores within the CPU and doing this will improve the performance of that software since now different pieces of code is executed on different cores instead of just a single core.
Through research, there were two methods uncovered for multi-threading. The first method is to use

Figure 12: *The atmosphere of a planet when the observer is above the planet*

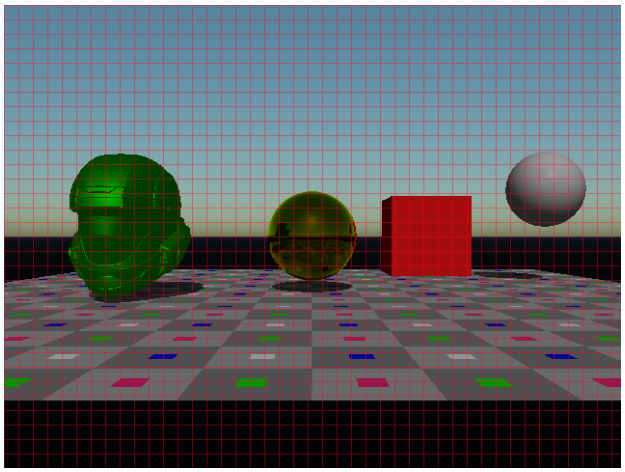

Figure 13: *An example image showing a render being split up into tiles*

$std :: thread$ to create threads and $std :: mutex$ to lock and unlock data for those threads. The other method as shown by (Brereton, 2014) is to use C++ functions called $std :: async$ and $std :: future$. This is what the project implemented since (Brereton, 2014) showcases these function in use for a ray-tracer. As explained by (cppreference, 2022), $std :: async$ is a function template that runs a function that is passed into it and executes it on another thread and returns a $std :: future$ which holds the results of the function call. By using a lambda function within $std :: async$, the code for generating a primary ray and casting the ray recursively can be held and execute within the lambda function. Doing this will allow the rendering to be multi-threaded.

3 Evaluation

This evaluation will mainly look at the reduction of render times with comparison made with each systems in place.

3.0.1 Ray-Geometry optimization using bounding boxes

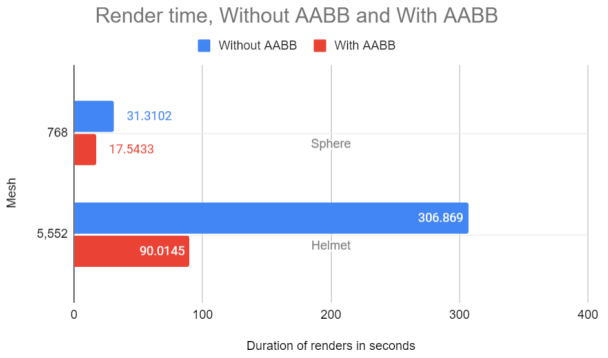

Figure 14: *Bar chart showing render duration with AABB and without AABB*

When rendering objects with just using (Tomas, 1997) ray triangle intersection, the time it takes to render is very slow and this becomes even slower when using models with higher triangle counts. An easy solution to reduce render times and prevent all rays having to check with all triangles, a bounding box can be used.

As shown by **figure 14**, a test was made with two separate meshes to see how long it takes to render each mesh. Clearly, using a bounding box help reduce render times by a big margin. However, while the results are good, it can be improved as mentioned by (scratchapixel, 2022) and (Kay and Kajiya, 1986) who noted that a normal AABB bounding boxes like the one developed for this project, encompassing an object loosely which makes the bounding box test for rays that may never intersect with the object and wasted computation. In order to improve this, a bounding box that fits more closely with the object may improve performance.

3.0.2 Bounding Volume Hierarchy performance

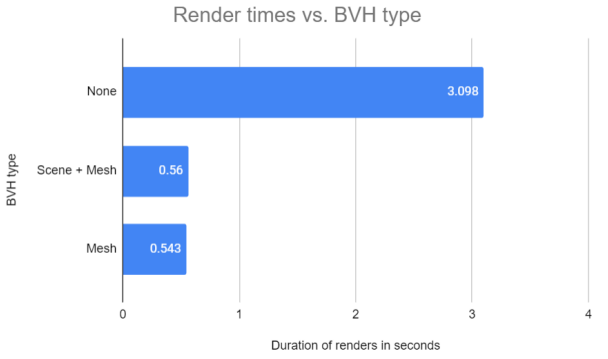

Figure 15: *Bar chart showing render duration with each different bvh developed for the project*

The project has implemented two types of bvh. One that splits the objects in the scene into a different

bounding boxes and another bvh for splitting the objects triangles into different bounding boxes. This can be seen in **figure 15**.

Obviously, there would be a big decrease in rendering time when using a bvh. However, between using a bvh for both the scene and mesh and just using a bvh for the meshes, there is a small increase in performance when just using a bvh for the meshes. This might be because having a bvh for both the scene and mesh's forces a ray to traverse and check more bounding boxes.

It should also be noted that these tests were made using a bvh that uses midpoint split and using a bvh that uses Surface Area Heuristics would provided better results. As well, adding more meshes to the scene could negate the performance result found with the bvh for both the scene and mesh.

3.0.3 Multi-threaded performance

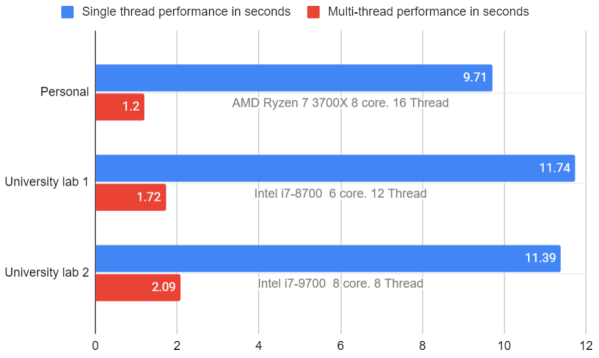

Figure 16: *Bar chart showing the different CPU being tested on single thread and multi-threaded performance*

To see how well multi-threading did for rendering times, tests were made on three different computers with different CPUs. Each computer had to render the same scene with and without multi-threading. **Figure 16** shows the results from this test. From a glance, multi-threading makes a big improvement overall for rendering times on all CPUs and a CPU that has a higher core and thread count will have better performance.

However, these tests were made using the C++ functions $std :: async$ and $std :: future$ and having another test, this time using $std :: thread$ and $std :: mutex$ could yield better results since $std :: async$ is a "fire and forget" type of function and using something low level like $std :: thread$ could provide better performance.

3.0.4 Overall performance

Figure 17 showcases all the optimization made to reduce rendering times for the ray-tracer. Most of these optimization like bvh, multi-threading and bounding boxes have already been discussed. However, this graph was made to show visually each optimization technique that has impacted on the rendering time.

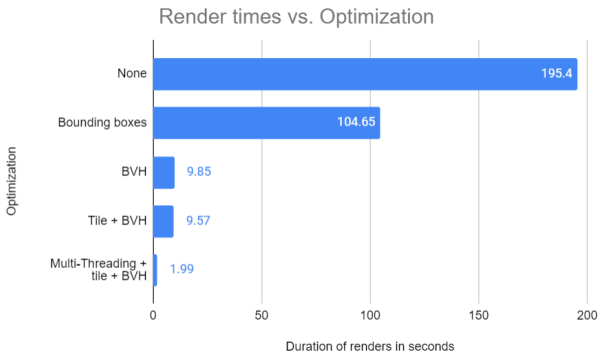

Figure 17: *Bar chart showing all the systems which reduce rendering time*

Looking at the performance of no optimization vs Multi-threading + tile + BVH shows a 98.98% reduction in rendering times.

Obviously, adding or change optimization techniques like using Surface Area Heuristics, rasterization for visibility or look up tables when rendering the atmosphere can provide even better results but the techniques used at the moment have provided a massive decrease in overall render times.

4 Conclusion

The project that was produced has allowed for a varied study into the different techniques and methods used in ray-tracing. The end product from the project allows any one to make simply scenes with different types of surface material and lighting. While there can be improvement made into performance, the project that was produced has satisfied most of the requirement set out by the task.

Bibliography

A.J, Preetham, Shirley Peter, and Smits Brian (1999). *A Practical Analytic Model for Daylight*. URL: https://courses.cs.duke.edu/cps124/spring08/assign/07_papers/p91-preetham.pdf. (accessed: 27.03.2022).

Brereton, Foster (2014). *Solving multithreaded raytracing issues with C++11*. URL: https://medium.com/@phostershop/solving-multithreaded-raytracing-issues-with-c-11-7f018ecd76fa. (accessed: 29.03.2022).

britannica (n.d.). *Reflection and refraction*. URL: https://www.britannica.com/science/light/Reflection-and-refraction. (accessed: 25.03.2022).

Burely, Brent (2012). *Physically Based Shading at Disney*. URL: https://media.disneyanimation.com/uploads/production/publication_asset/48/asset/s2012_pbs_disney_brdf_notes_v3.pdf. (accessed: 29.12.2021).

cppreference (2022). *std::async*. URL: https://en.cppreference.com/w/cpp/thread/async. (accessed: 27.03.2022).

Greve, Bram de (2006). *Reflections and Refractions in Ray Tracing*. URL: https://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction.pdf. (accessed: 27.03.2022).

Humphreys, Matt Pharr. Wenzel Jakob. Greg (2021). *Bounding Volume Hierarchies*. URL: https://pbr-book.org/3ed-2018/Primitives_and_Intersection_Acceleration/Bounding_Volume_Hierarchies. (accessed: 15.03.2022).

Kay, Timothy L. and James T. Kajiya (1986). "Ray Tracing Complex Scenes". In: *SIGGRAPH Comput. Graph.* 20.4, 269–278. ISSN: 0097-8930. DOI: 10.1145/15886.15916. URL: https://doi.org/10.1145/15886.15916.

Kulling, Kim (2021). *The Asset-Importer-Lib*. URL: https://assimp-docs.readthedocs.io/en/v5.1.0/. (accessed: 02.03.2022).

Nishita, Tomoyuki et al. (1993). "Display of the Earth Taking into Account Atmospheric Scattering". In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '93. Anaheim, CA: Association for Computing Machinery, 175–182. ISBN: 0897916018. DOI: 10.1145/166117.166140. URL: https://doi.org/10.1145/166117.166140.

Nothings (2022). *stb*. URL: https://github.com/nothings/stb. (accessed: 20.03.2022).

Peter Shirley Steve Hollasch, Trevor David Black (2020). *Ray Tracing in One Weekend*. URL: https://raytracing.github.io/books/RayTracingInOneWeekend.html. (accessed: 10.03.2022).

Phong, Bui Tuong (1975). *Illumination for Computer Generated Pictures*. URL: https://users.cs.northwestern.edu/~ago820/cs395/Papers/Phong_1975.pdf. (accessed: 01.04.2022).

Quilez, Inigo (2008). *tracing in tiles - 2008*. URL: https://www.iquilezles.org/www/articles/cputiles/cputiles.htm. (accessed: 17.03.2022).

rastertek (n.d.). *Tutorial 40: Shadow Mapping*. URL: http://www.rastertek.com/dx11tut40.html. (accessed: 28.03.2022).

Robert Cook. Kenneth, Torrance (1982). *A Reflectance Model for Computer Graphics*. URL: https://graphics.pixar.com/library/ReflectanceModel/paper.pdf. (accessed: 02.01.2022).

Rubin, Steven M. and Turner Whitted (1980). "A 3-Dimensional Representation for Fast Rendering of Complex Scenes". In: *SIGGRAPH Comput. Graph.* 14.3, 110–116. ISSN: 0097-8930. DOI: 10.1145/965105.807479. URL: https://doi.org/10.1145/965105.807479.

ScratchaPixel (2022). *generating camera rays*. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays. (accessed: 10.03.2022).

scratchapixel (2022). *Introduction to Acceleration Structures*. URL: https://www.scratchapixel.com/lessons/advanced-rendering/introduction-acceleration-structure/bounding-volume-hierarchy-BVH-part1. (accessed: 15.03.2022).

ScratchaPixel (2022a). *moller trumbore ray triangle intersection*. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection. (accessed: 10.03.2022).

– (2022b). *Simulating the Colors of the Sky*. URL: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/simulating-sky. (accessed: 27.03.2022).

Tomas Moller. Ben, Trumbore (1997). *Fast, Minimum Storage Ray/Triangle Intersection*. URL: http://www.graphics.cornell.edu/pubs/1997/MT97.pdf. (accessed: 03.03.2022).

Wenson (2014). *Raytracer*. URL: http://whsieh.github.io/raytracer/. (accessed: 27.03.2022).

Whitted, Turner (1980). "An Improved Illumination Model for Shaded Display". In: *Commun. ACM* 23.6, 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882. URL: https://doi.org/10.1145/358876.358882.