Ray Tracing in C++

Luke Tolchard
17024063

University of the West of England

July 24, 2020

The task assigned was to create a C++ CPU ray tracing engine that included support for materials, reflections and shadows. This final program contains a bounding volume hierarchy and Monte Carlo integration to help optimise the computational process.

1 Introduction

Ray Tracing is a 3D rendering technique for generating an image by tracing paths of light from a source and calculating how they would then encounter virtual objects. It affords a high level of realism but at significant computational cost. The challenge with creating a ray tracer comes in the form of adding realistic natural light physics to a coded environment without making the program impractical to operate in terms of hardware requirements or developing time.

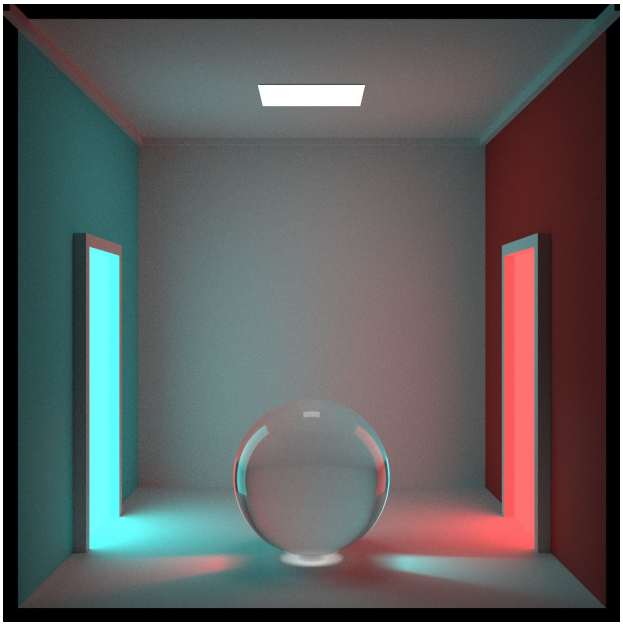

Figure 1: *An adapted Cornell box to showcase the final render from the completed ray tracer*

2 Related Work

Shirley, 2016, released several books on Ray Tracing and graphical computations for C++ implementation.

Smits, 1999, discussed the importance of a bounding volume hierarchy within all forms of ray tracing and offers a solution for low level and smaller scale ray tracers.

3 Method

3.1 Camera Rays

At a ray tracing engines core, there is a ray class creating rays through pixels of a scene and a computation of what colour can be seen along that ray returns to the camera to create a colour value for that pixel. (Shirley, 2016). Often, the ray hits an object within the scene, that will then redirect it or change its value, up to a total of x bounces before the colour is finally sampled. In Peter Shirley's example(2016), he describes a ray as a function:

$$p(t) = A + t * B \tag{1}$$

Here, 'p' is a 3D position along a line, 'A' is the ray origin, and 'B' is the ray direction. The parameter 't' is a value that represents an intersection in the ray's path, such as with a world object. How the ray then intersects and interacts with the hit object depends on it's shape, material, and nearby light sources.

3.2 Ray-Sphere Intersection

Kyle Halladay (2013) expands on the ray equation to explain how it could be used to return a ray hit on the surface of a sphere, returning the intersections position in world space to the renderer. This would then be used to display the sphere in the image, and allow the ray to continue to reflect and refract off its surface. Before calculating the intersection vectors however, it's a good idea to program ways for each ray to know early whether it has intersected with the sphere or not, and if it hasn't then it doesn't have to complete the full set of calculations, allowing the program to move on to the next ray faster and reducing computation time.

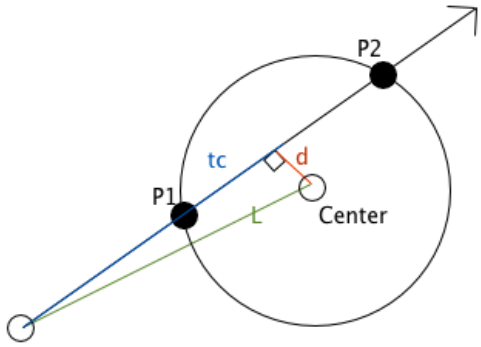

Figure 2: *A ray intersects with a sphere at point P1 and P2, from an origin (Kyle Halladay, 2013)*

As per Figure 2, a ray intersecting with a sphere will almost always have 2 points of intersection, shown here as P_1 and P_2. In order to work out whether a ray intersects with a sphere or not, a calculation to determine the length of tc, L, and d, must be completed as the sphere and camera vector class will hold the world position for their implementation, this is a simple calculation of $L = Center - Origin$. tc can then be calculated by finding the dot product of L and the ray direction. If $tc < 0$, the ray does not intersect the sphere and does not have to be subject to anymore calculation, which saves on computational resources. Calculating the final unknown edge in Figure 2's inner triangle, d, can be solved using Pythagoras' Theorem $a^2 + b^2 = c^2$. To find b (d in the case of Figure 2), the formula must be rearranged to leave $b = \sqrt{(c^2 - a^2)}$ Which when substituted with the earlier values becomes:

$$d = \sqrt{(tc^2 - L^2} \tag{2}$$

If d returns as greater than the radius of the circle then a point on t_1c (Figure 3) will be outside of the sphere, proving the ray does not intersect. Once proving the ray does intersect the sphere with these two checks, the scope of the Pythagorean triangle can be moved within the confines of the sphere in order to calculate t_1c, t_1, and t_2 (the distance between the origin and P_1 and P_2 respectively). A rearrangement of Pythagoras's theorem to find t_1c gives:

$$t_1c = \sqrt{(radius^2 - d^2)} \tag{3}$$

Which can now be used with tc to determine the distance from the origin that the intersections take place.

$$t_1 = tc - t_1c \tag{4}$$

$$t_2 = tc + t_1c \tag{5}$$

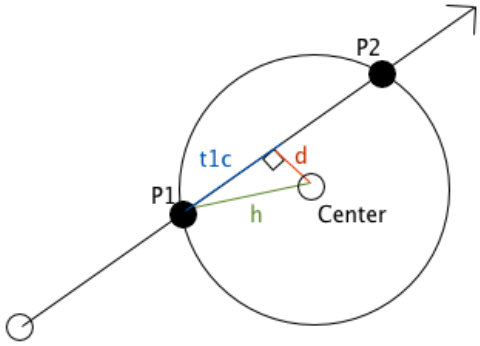

Figure 3: *Calculating the intersect points in world space*

Plugging the new data back into the original ray equation $p(t) = A + t * B$ returns:

$$P_1 = Origin + Direction * t_1 \tag{6}$$

$$P_2 = Origin + Direction * t_2 \tag{7}$$

The ray now has the functionality to calculate whether it intersects with a sphere, and where in world space those intersections would occur. Kyle Halladay (2013) muses that, when extrapolated over the potentially billions of rays that could be cast, "any optimisations you can make pay dividends".

3.3 Ray-Triangle Intersection

Computing the intersections of a ray with a triangle is necessary for ray tracing with more complex meshes and objects than just spheres (Scratchapixel Author, 2020). Möller and Trumbore, 1997, present a solution that first tests whether a ray intersects with the triangle, and can then proceed to determine the vector position in world space if it does. A point in a triangle can be defined as:

$$point(u, v) = (1 - u - v)V_0 + uV_1 + vV_2 \tag{8}$$

where u, v are the barycentric coordinates which must be individually ≥ 0 but not sum to be > 1. $p(t)$

from the ray equation is equal to $point(u, v)$ which allows substitution to yield:

$$A + t * B = (1 - u - v)V_0 + uV_1 + vV_2 \quad (9)$$

Möller and Trumbore (1997) describe this formulas effect as geometrically translating the triangle in question to the origin of the ray, and transforming it into a unit triangle with in the y and z axis with the ray direction aligned with x (Figure 4). They go on to use a mathematical technique known as Cramer's rule, which states that the multiplication of a matrix M by a column vector X is equal to a column vector C which can then find Xi by dividing the determinant of Mi by the determinant of M, where Mi is the matrix formed by replacing the ith column of M by the column vector C (Kensler and Shirley, 2006). After evaluating each determinant, the values of t, u and v would be calculated.

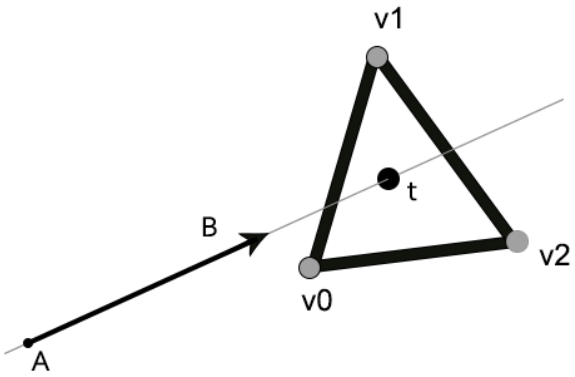

Figure 4: *A ray-triangle intersection as described by Möller and Trumbore, 1997*

$$[-B(V_0 - V_0)(V_2 - V_0)] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = A - V_0 \quad (10)$$

Giving that $E_1 = V_1 - V_0, E_2 = V_2 - V_0$ and $T = A - V_0$, Cramer's rule can be used to produce:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{-B, E_1, E_2} \begin{bmatrix} T, E_1, E_2 \\ -B, T, E_2 \\ -B, E_1, T \end{bmatrix} \quad (11)$$

Linear algebra also states that $|A, B, C| = -(A \times C) \cdot B = -(C \times B) \cdot A$, which means that the above matrix can be rewritten in a shorter form which includes 2 new variables P and Q for substituting out repeating calculations (Möller and Trumbore, 1997).

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(B \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (B \times E_2) \cdot T \\ (T \times E_1) \cdot B \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} (Q \cdot E_2 \\ (P \cdot T \\ (Q \cdot B \end{bmatrix} \quad (12)$$

This will improve the resource efficiency of this calculation and help prepare the ray tracer for larger scale projects.

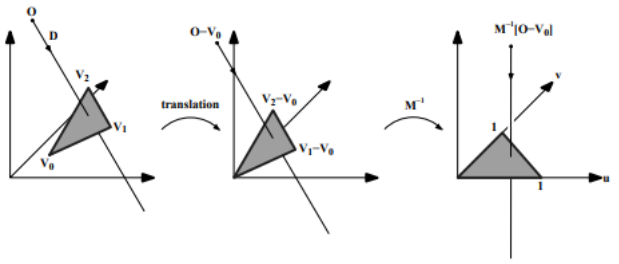

Figure 5: *The effective transposition of a triangle intersected by a ray (Möller and Trumbore, 1997)*

3.4 Bounding Volume Hierarchy (BVH)

The ray-object intersection is the main time-bottleneck in a ray tracer (Shirley, 2016). As these rays are repeated searches contained within the same environment, they should be able to realise when they are traversing an area where they have been before and know for sure that there is no object to be found in that direction. Grouping objects together in bounding volumes and sub-volumes allows for the intersection calculations to back out early in the case it realises there are no objects in the direction its progressing and this saves valuable computational resources.

In Figure 6, an example of a ray-lab intersection is

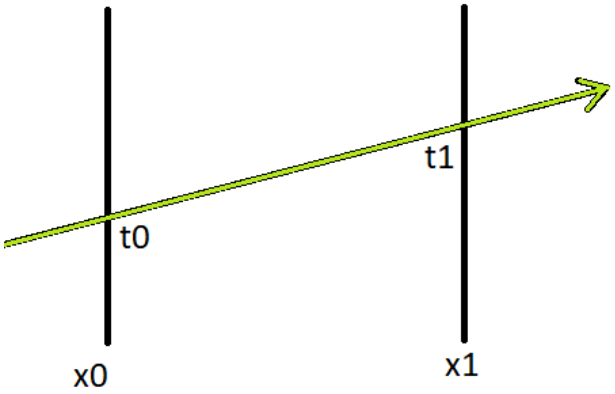

Figure 6: *A X plane in a ray-slab intersection*

shown. In its most basic form, a BVH is comprised of 3 plane equations for x,y,z that create a 'box' around the object that encompasses it entirely. Equation 1 can be applied to all three of the x/y/z coordinates to produce equations that look like:

$$x(t) = A_x + tb_x \quad (13)$$

This ray hits the plane $x = x_0$, which results in this solve for t:

$$t_0 = \frac{x_0 - A_x}{b_x} \quad (14)$$

These volumes need to be fast and compact, and how one might implement them into their own project might depend on what regularity of models they are using - if a bounding volume representing an object

fits too loosely, many rays intersecting and passing the check do not actually intersect with the primitive and are wasted. Using a more complex and compact bounding volume would give better results but is more costly to ray trace (Scratchapixel Author, 2020).

3.5 Materials and Textures

Materials and textures can be assigned to objects within the scene to change the light rays behaviour and produce different effects. Within the program, there is a base material class that contains functions for the scattering of rays after an object has been hit, and the emission of colour from the material. From there, sub classes that derive from the base class are created to differentiate between the different types of material the objects might have.

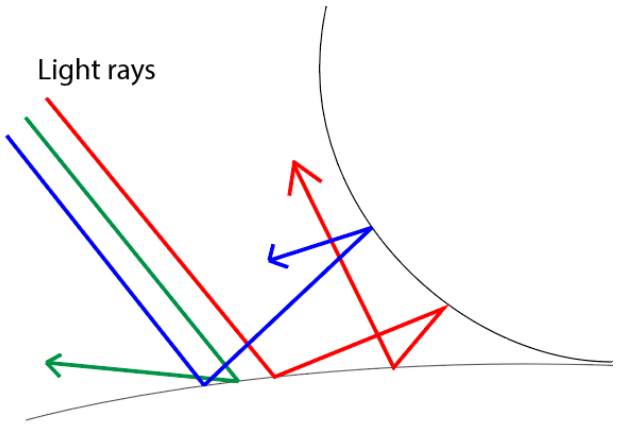

Figure 7: *Light rays striking a diffuse material are randomly reflected*

Figure 7 shows how a diffuse material might influence the ray calculation. As diffuse materials don't emit any light of their own but modulate the colour of their surroundings with their own intrinsic colour, the rays are randomly scattered along its surface. They also might be absorbed rather than reflected, which prompts a material to appear dark in a final render (Shirley, 2016). Changing the material property to give a cleaner, less random reflection makes the material appear more metallic and reflective, and a reflective return that is only slightly different from the true reflection allows for a fuzziness parameter to the metallic finish.

A texture could be applied to a surface from an external file, or written from a function, and they make the colours of a surface procedural (Jacobs, 2020). In order to create these textures, the spherical co-ordinates were calculated using this utility function:

```
1 void return_sphere_uv(const vector3&
      p, double& u, double &v)
2 {
3   auto phi = atan2(p.z(), p.x());
4     auto theta = asin(p.y());
5     u = 1-(phi + pi) / (2*pi);
6     v = (theta + pi/2) / pi;
7 }
```

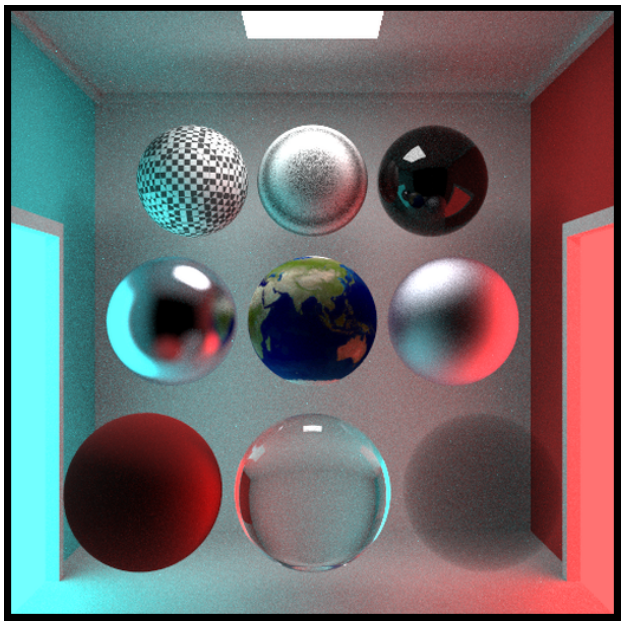

Figure 8: *From top left across: Checker texture, Perlin texture, Coloured Dielectric, Metal with low fuzz, Earth texture, Metal with high fuzz, Red Lambertian , Clear Dielectric, Participating Volume(Smoke)*

Figure 8 shows a render of all the implemented materials and textures that have their own subclass side by side from the completed ray tracer.

3.6 Lighting and Monte Carlo Integration

Another modification that can be made to a material function is to give it emissive properties (Shirley, 2016). In order for objects in a scene to be visible in the final render, there must be a source of illumination so that some light is reflected from them to the camera sensor (Pharr, Jakob, and Humphreys, 2016).

The light rendering equation as confirmed by Kajiya, 1986, is:

$$L_o(\omega_o) = L_e(\omega_o) + \int_\Omega f(\omega_i, \omega_o) L_i(\omega_i)(\omega_i n) d\omega_i$$

To find the light towards the viewer from a specific point, we must sum the light emitted from that point plus the integral within the unit hemisphere of the light coming from any given direction multiplied by the chances of the light rays bouncing towards the

origin, and also by the irradiance factor over the normal at the point.

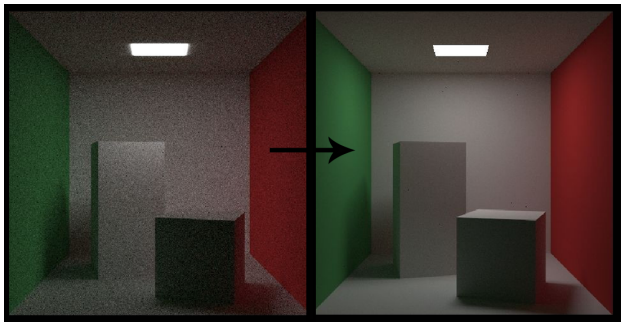

Figure 9: *The difference between a noisy distributed ray tracing algorithm (left) and a cleaner, Monte Carlo algorithm with a mixture of cosine and light sampling (right)*

Ray tracers soon face a problem with classic distributed ray tracing: An increase in scene complexity leads to images with more noise unless you sample many more rays per pixel to counteract the problem, which leads to a sharp increase in computational requirements. Monte Carlo integration is a method for using random sampling to estimate the values of integrals, which would help to optimise render to produce a smoother return of pixel colour with significantly less stress on the program (Shirley, 2016).

4 Evaluation

In order to gain an understanding of the effect of BVH's and Monte Carlo Integration, A simple comparison test was carried out to determine the efficiency of such data structures within the ray tracer. All rendered images were set to 500px by 500px, with 200 rays per pixel.

Data Structure Evaluation			
Test Scene	Distributed Ray Tracing	BVH enabled	BVH + Monte Carlo
1 Earth Sphere	59 s	55 s	51s
2 Perlin Spheres	121 s	107 s	95 s
Emissive Light	98 s	93 s	64 s
Cornell Box	456 s	398 s	332 s
Material Render (Figure 8)	1014 s	928 s	645 s
Final Render (Figure 1)	863 s	736 s	482 s
Average Time	435.16 s	386.16 s	278.16 s

5 Conclusion

With optimisation of ray casting code comes a double edged design decision. Careful profiling can result in significant speedups and allow for far more complex and effective scenes to be produced, but it can also lead to code that is slower and more complicated. Eventually, a developer will arrive at the point where further optimisations made to the engine will no longer have any significant impact and they will have to go back and try and create better trees requiring fewer primitive bounding tests (Smits, 1999).

Bibliography

Jacobs, Noah (2020). *Ray Tracing - Noah Jacobs*. URL: `https://noahmjacobs.com/graphics/ray-tracing/`.

Kajiya, James T (1986). "The rendering equation". In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pp. 143–150.

Kensler, Andrew and Peter Shirley (2006). "Optimizing ray-triangle intersection via automated search". In: *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE, pp. 33–38.

Möller, Tomas and Ben Trumbore (1997). "Fast, minimum storage ray-triangle intersection". In: *Journal of graphics tools* 2.1, pp. 21–28.

Pharr, Matt, Wenzel Jakob, and Greg Humphreys (2016). *Physically based rendering: From theory to implementation*. Morgan Kaufmann.

Shirley, Peter (2016). *Ray tracing in one weekend*.

Smits, Brian (1999). "Efficiency Issues for Ray Tracing". In: *Journal of Graphics Tools* 3, pp. 1–14.