Ray Tracing Complex Scenes

Final Project Real-Time Rendering [CS7GV3]

Varun Kumar Gupta
Student ID: 23338877
Trinity College Dublin

Abstract

An implementation of a simple Path Tracer. Currently, it supports rendering cubes and spheres. Objects are bound by simple hulls, cubical [1], [2] and spherical [3]. The implementation is done in C++ and OpenGL [4] leveraging the power of the Graphics Processing Unit (GPU) via Fragment Shader [5]. By efficiently managing ray-object intersection calculations, it achieves significant improvements over legacy methods. Results show the quality with computation times to show the overall efficiency of the algorithm.

Keywords : Path Tracing, Ray Tracing, Extent, Bounding Volume, Global Illumination, Cornell Box

I. Introduction

The purpose of this project was to develop a simple Path Tracer (PT) with comparable results to existing software. Major focus was given on quality leading to sub-optimal performance. The implementation spends most of the time calculating the intersections for a given ray. It bounces the ray a fixed number of times before terminating it. It also supports progressive rendering (PR) [6], performing denoising [7] with increasing samples and time. The implementation is limited to solid shapes as bounding volumes, making it possible to run on a Fragment Shader [5]. If a particular ray doesn't intersect any object, it is discarded altogether.

In the reference paper [1], the author has made use of a hierarchical structure to provide even faster results. Due to the time constraint, this was not included in the current implementation. Focusing on specific aspects such as materials, including, Roughness, Reflection, Specular, and Emission [8]. Controlling various surface properties helps us give the scene a more real and interesting visual look.

II. Background

Ray Tracing (RT) is a technique for modelling light transport rendering algorithms for generating digital images [9]. Rasterizaton [10], is a technique where each 3D model is converted into pixels, on a 2D screen (image). Each pixel is assigned an initial color and further processed or ”shaded”, including changing pixel color, applying texture and more. In the end, everything is combined and apply a final color to the pixel, giving you a rendered picture.

Previous studies on this topic include dividing the objects into cells of an octree. Another approach is considering the sequence in which the ray passes these objects. There were many downsides, such as an object being intersected more than once, thus appearing twice in the octree. All these are examples of a space-partitioning approach. The reference and more recent papers approach this differently, partitioning objects rather than space. Popular approaches include using physics libraries such as Bullet [11], which is a physics library. This also happens to be the ”recommended” way to pick objects in game engines.

Using RT, the concept of Global Illumination (GI) [12] is also achieved. Indirect light is visible in the shadows of the objects inside the simulated Cornell Box [13]. Due to GI, there is inherently a lot of noise. For more open scenes, such as a simple HDRI, the noise is noticeably reduced due to more rays of light, direct and indirect, hitting each object.

III. Implementation

1) **Surface :**
A **Material** is used to define the look of a 3D Object. It is composed of various textures/maps (Images). Consider various surfaces. Not all perfectly reflect the light they receive. This is the basis of diffuse and rough (Specular) surfaces. The more irregular or "rough" a surface is, the more light scatters on it's surface. Emission is simply achieved by making the object emit light, similar to a point light. This time, the light actually has some geometry and it is not just a random point in space!

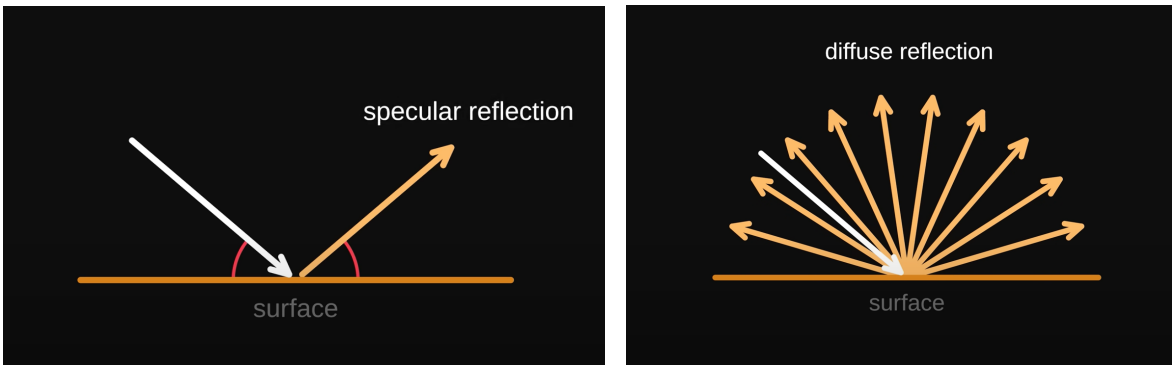


Fig. 1: Left : Perfect Reflection, $Roughness = 0$; Right : Irregular Reflection, $0 < Roughness \leq 1$; Sebastian Lague

Following the rendering equation:

$$I = g(k_\epsilon + \sum L^i_{df} k_{df} \mathbf{(n.l)} + \sum L^i_s k_s \mathbf{(h.n)}^\alpha + L^i_a k_a \tag{1}$$

where,

k_{df} is the Diffuse Map, giving the material it's base color.
k_ϵ is the Emission Map, telling the material to emit light instead of absorb it.
k_s, α is the Specular Map and Specular Exponent, defining the reflective properties of the material.
L^i_{df}, L^i_s, L^i_a are the Light and Environment Mapping, feeding the material with surrounding data. Including lights, objects; the environment.
g is the Visibility Mapping, specifically Shadow Mapping in this case.
No texture mapping has been done as of now. The objects support simple materials with basic control over properties. Equation (1) is used in the Phong Illumination model. Moving on to the Ray Tracing equation:

$$I = I_{local} + k_{rg}.I + k_{tg}.I \tag{2}$$

where,

I_{local} is the Direct Diffuse term, calculated using Phong Illumination. Globally, diffusion of light due to imperfections is ignored.
$k_{rg}I$ is the reflectance term.
$k_{tg}I$ is the transmittance term.

In this implementation, there is no use of the transmittance term. The only focus is Direct Diffuse term (GI) and Reflectance term.

2) **Intersection :**
The intersections for both sphere and cubes are calculated using custom functions. For Box/Cube Intersections [4], the following formulae are used:

$$t_1 = \max(\min_{xyz}(t_{0s}, t_{1s}), t_0) \tag{3}$$

$$t_2 = \max(\min_{xyz}(t_{0s}, t_{1s}), t_1) \tag{4}$$

$$\text{Hit} = (t_1 \geq 0) \wedge (t_1 \leq t_2) \tag{5}$$

where,

min_{xyz} is component wise minimum on a vector across x, y and z values.
max_{xyz} is component wise maximum on a vector across x, y and z values.

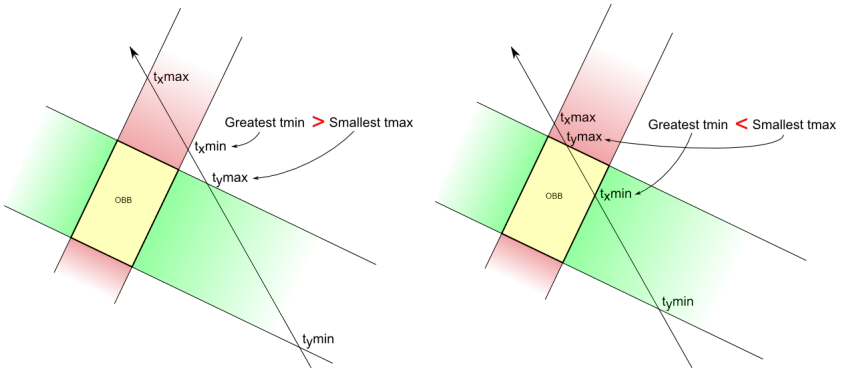


Fig. 2: Left : No Intersection, enter the green area → exit the green area → enter the red area → leave the red area; Right : Intersection, enter the green area → enter the red area! A sequence of events that are followed. Ray-OBB [14], [2]

Algorithm 1 Ray-Box Intersection Algorithm

procedure RAYBOXINTERSECTION(position, size, ray.origin, ray.direction, hitDistance)
  $t1 \leftarrow -1000000000000.0$ ▷ Start of intersection interval
  $t2 \leftarrow 1000000000000.0$ ▷ End of intersection interval
  $boxMin \leftarrow$ position $-$ size$/2.0$
  $boxMax \leftarrow$ position $+$ size$/2.0$
  $t0s \leftarrow$ (boxMin $-$ ray.origin)$/$ray.direction
  $t1s \leftarrow$ (boxMax $-$ ray.origin)$/$ray.direction
  $tsmaller \leftarrow \min(t0s, t1s)$
  $tbigger \leftarrow \max(t0s, t1s)$
  $t1 \leftarrow \max(t1, \max(tsmaller.x, \max(tsmaller.y, tsmaller.z)))$
  $t2 \leftarrow \min(t2, \min(tbigger.x, \min(tbigger.y, tbigger.z)))$
  $hitDistance \leftarrow t1$ ▷ Distance to intersection
  if ($t1 \geq 0$ AND $t1 \leq t2$) **then**
    return True
  else
    return False
end procedure

These check the direction of the ray, with entry, and exit points of the bounding volume. Since a relatively simple shape is being used, it keeps the process robust and fast to compute. All of these computation are carried out in the Fragment Shader on the GPU.

For spherical intersections [3], first the vector from the ray origin to sphere center is calculated $(v = c - o)$. The resultant vector is then projected onto the ray direction $(t_{ca} = \mathbf{d}.v)$, giving the closest ray to the sphere center. Then the distance from the sphere center to the closest ray is calculated $(d = || \, v - t_{ca}\mathbf{d} \, ||$. After that, checks for intersection are made. If $d > r$, then there is no intersection. Otherwise, when $d \leq r$ The final equation looks like:

$$t_{entry,exit} = t_{ca} \pm \sqrt{r^2 - d^2} \tag{6}$$

where, $\sqrt{r^2 - d^2} \geq 0$ for a valid (real) solution.

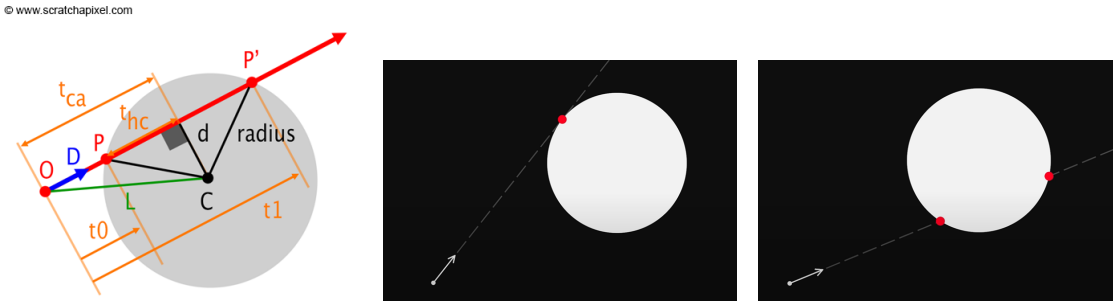


Fig. 3: Left : A ray intersection a sphere, with various terms used to solve sphere intersection. For geometrical and analytical solutions; Middle : A single ray touching the edge of a sphere. No intersection; Right : A single ray intersecting a sphere. Two contact points are generated. (Left) Sphere-OBB [3] (Middle, Right) Intersection [15]

Algorithm 2 Sphere-Ray Intersection Algorithm

```
procedure SphereIntersection(position, radius, ray, hitDistance)
    t ← dot(position - ray.origin, ray.direction)        ▷ Projection onto ray direction
    p ← ray.origin + ray.direction · t                   ▷ Nearest point on ray to sphere center
    y ← length(position - p)                             ▷ Distance from sphere center to p
    if y < radius then
        x ← sqrt(radius^2 − y^2)                         ▷ Distance from p to intersection point
        t1 ← t − x
        if t1 > 0 then
            hitDistance ← t1
            return True
        end if
    end if
    return False
end procedure
```

Additionally, there is a plane for the floor. First we calculate the denominator, by calculating the dot product between the normal of the plane and direction of a ray $(denom = \mathbf{n.d})$. Check if the ray is parallel to the plane $(denom > \epsilon)$, given the denom $\neq 0$. If the ray is not is not parallel, compute the distance from ray origin to intersection point on the plane.

$$t = \frac{n.(\mathbf{p\text{-}o})}{denom}; denom \neq 0 \tag{7}$$

$$return(t \geq \epsilon) \tag{8}$$

3) **Ray Casting :**
Rays are generated from the camera and shot into the 3D Scene [16]. It is not possible to bounce the rays an infinite number of times. For this, the implementation has a parameter that can be changed to change the number of bounces per ray. Furthermore, sampling all points on a hull is wasting resources and performance. To improve that, first some noise is introduced in sampling the bounce points. A pseudo-random number is generated for each UV position on the object using the following equation:

$$result = x - \lfloor x \rfloor \implies x \in [0, 1] \tag{9}$$

$$x = \sin(co_x * 14.4527 + co_y * 76.8761) * 39282.6275 \tag{10}$$

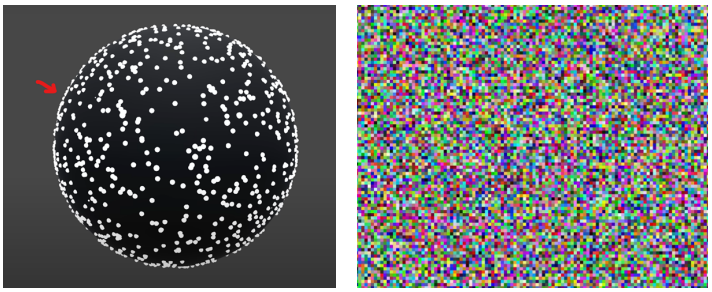


Fig. 4: Left : Randomly sampled points on a sphere using noise. Right : Result of the custom noise function. Noise [15]

Further optimizations include sampling the hemisphere [17] based on the surface normal. Looking at statistics, GI mostly effects only the hemisphere along the surface normal, making it safe to ignore half the points! This in turn, speeds up computation. Any vector (V) that makes an $angle < 90°$ with surface normal N lies within the hemisphere.

$$N.V = N_x * V_x + N_y * V_y; N.V > 0 \tag{11}$$

Algorithm 3 Sample Point on Hemisphere Algorithm

function SAMPLEHEMISPHERE(normal, alpha, seed)
  $cosTheta \leftarrow$ pow(rand(seed), $1.0/(\text{alpha} + 1.0)$) ▷ Using equation (9)
  $sinTheta \leftarrow$ sqrt$(1.0 - cosTheta * cosTheta)$
  $phi \leftarrow 2 * \pi *$ rand(seed.yx)
  $tangentSpaceDir \leftarrow$ vec3$(\cos(phi) * sinTheta, \sin(phi) * sinTheta, cosTheta)$
  return getTangentSpace(normal) $* tangentSpaceDir$ ▷ Transform direction to world space
end function

Making use of equation (9), the hemisphere is sampled. Fig5 gives a visual representation of the algorithm.

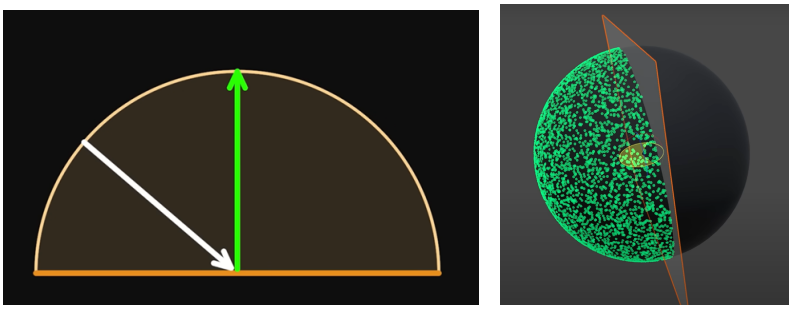


Fig. 5: Left : Surface Normal N (Green) and Vector V (White). Right : Result of the hemisphere sampling. Hemisphere [15]

4) **Indirect Illumination :**
Coming to this part, the algorithm transitions into a Path Tracer [18]. The amount of rays starting from initial hitpoints and bouncing around are limited [17]. Probabilistically choosing between Specular and Diffuse lighting using a Russian Roulette technique, using equation (9). The reflectance type is selected based on material properties. Energy losses and gains are factored into further calculations at each step. This produces much more complicated light interactions. Modifying equation (9), we get:

$$I = I_{local} + k_{rg}.I_{reflect} + k_{gi}.I_{indirect} \tag{12}$$

where,
I is the total illumination, resultant from local, reflection and indirect illuminations. $I_{reflect}$ is the illumination resukting from reflective surfaces. This is calculated by casting new rays in the direction of reflection and evaluating their effect on the scene.
$I_{indirect}$ is the indirect illumination component. This is calculated by summing the contributions of light paths that bounce off of other surfaces.

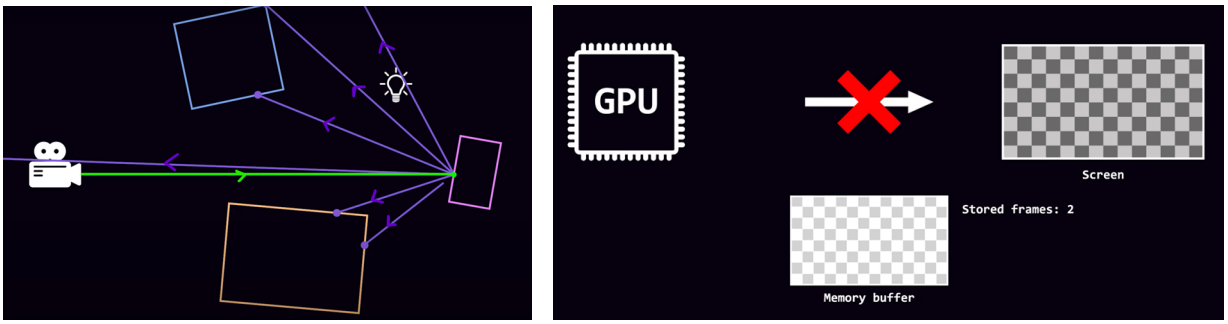


Fig. 6: Left : Indirect Illumination : A single ray bouncing off an object after 1 pass. Right : Progressive sampling to reduce noise and improve render times. Ray Tracing Engine [14]

5) **Progressive Rendering :**
Progressive Rendering incrementally refines image quality over time. It calculates the same frame multiple times within a time period and compounds them on top of each other to give better results. A random noise is sampled for each pass, making it converge even faster! This approach helps achieve real-time ray tracing while making the scene look good and interactive. The equation for progressive rendering is:

$$C = \frac{1}{N} \sum_{p=1}^{N} I^{(p)} \tag{13}$$

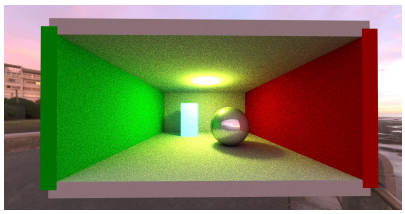
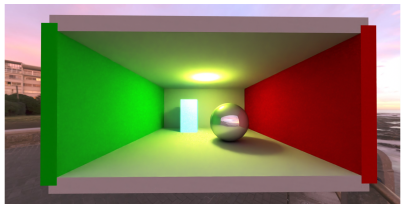
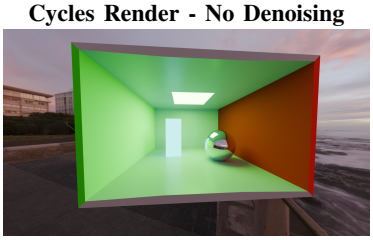
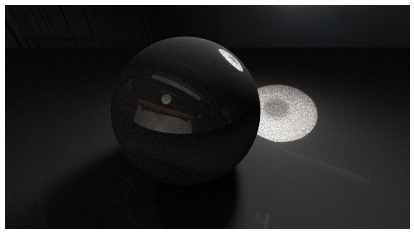
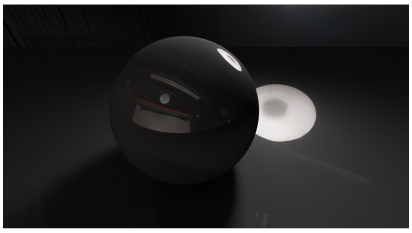
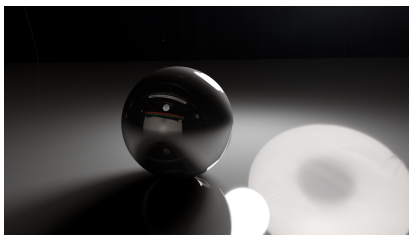
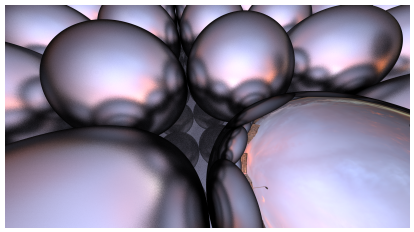
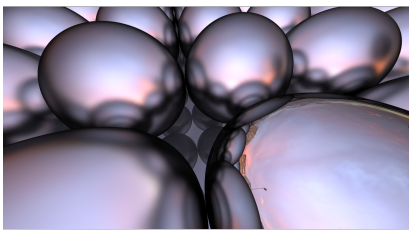
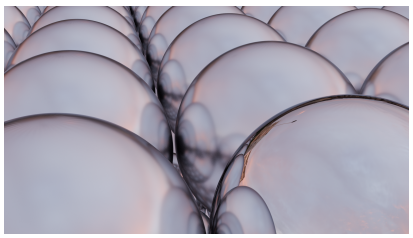
where,
C is the final color of the pixel after PR.
N is the number of rendering passes.
$I^(p)$ is the color calculated for the pixel at p^{th} pass

All these steps complete the main algorithm. There are more functions which use similar concepts. For object picking, Cube Intersection [2] (Algorithm 1) and Sphere Intersection [3] (Algorithm 2) are used.

IV. Results and Discussion

In this section, the renders of the implemented Path Tracer is compared with existing state of the art Path Tracer(s). Cycles [19], a physically based Path Tracer from Blender [20] has been used. Comparing various scenes by closely re-creating them. These are not one to one mappings in terms of color-space and gamma correction, and modelling. Color space $sRGB - AGX$ has been used in Blender [20], which is only available in versions 4.1.0 and above.

TABLE I: Comparing renders from Custom Path Tracer & Cycles [19] Path Tracer. The differences can be perceived visually, along with time to render metrics give in *(HH:MM:SS)* format. All renders in Cycles have been carried out using the following settings : **Samples:** 1024; **Light Bounces:** 12; **Denoising:** No. These are similar to the custom renderer where, **Light Bounces:** 6, **Passes per frame:** 3, **Frames per render:** 360. $TotalSamples = Passes * Frames \implies 3 * 360 = 1080$.

Custom - No Denoising	**Custom - Blender Denoising**	**Cycles Render - No Denoising**
		
Time: 00:03:47	Time: NA	Time: 00:00:37.34
		
Time: 00:00:12	Time: NA	Time: 00:00:14.60
		
Time: 00:00:37	Time: NA	Time: 00:00:23.25

There are few major differences in all the renders, some of which are stated below:

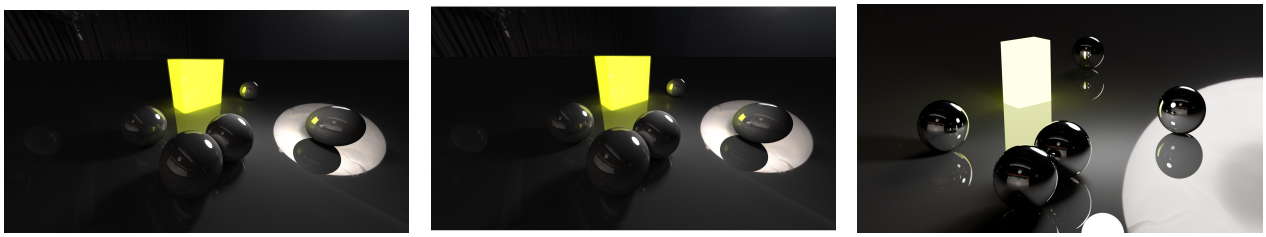


Fig. 7: More renders comparing the custom Path Tracer and Cycles Path Tracer. From left to right - Time: 00:00:59 & Time: NA & Time: 00:00:14.60

.

1) **Noise:** Noise is by far the most noticeable difference. Cycles produces less noise compared to the custom Path Tracer, even when it is given less samples.
2) **Indirect Illumination:** Cycles provides better indirect illumination and control over more granular surface properties such as Ambient Occlusion, Anisotropy, better HDRI mapping and support for various color modes. In the Cornell Box (Table I), the quality of shadows, GI, soft falloff of the area light, and color blending in shadows is much more pronounced and realistic.
3) **Speed:** Cycles is performing exponentially better with more details and less rendering times. Cycles uses various acceleration structures such as Optix [21] and BVH to speed up computation and increase intractability. The custom Path Tracer is running on a single Fragment Shader, without any acceleration structures, making the difference very apparent.
4) **Reflections:** The overall quality of reflections in Cycles is way better. Even with the same number of bounces of a ray, it gives more accurate results. More details are reflected, improving GI as a side product.

V. Improvements

There are many areas for improvement! The most obvious one would be using a Compute Shader [22], which utilizes the high core count of the GPU to perform calculations parallelly on multiple threads. Next, including more objects that can be Ray Traced, this would include using hierarchical structures to divide the objects, as discussed in [1]. Furthermore, the Russian Roulette method of selecting the surface properties for rays is inefficient. A weighted approach should be used, wherein, the vertices closer to a light source should focus more on the reflecting light and relevant surface properties, compared to others.

The shader used for viewing the scene outside of Ray Tracing is not suitable, it only represents the objects with a simple outline and object colors. This makes it harder to compose the overall scene as there is no depth perception shading done.

VI. Conclusions

Making a Path Tracer is not easy, yet a very fun task! Facing many difficulties such as calculating hulls, intersections, limiting light bounces and other technical aspects made very good use of many concepts. Ranging from fundamental coding principles to eliminate recursion, to more advanced concepts.

Overall, Cycles performs better in all aspects compared to the custom Path Tracer. For simple renders with better environment mapping (Strength) and an open setup (Not closed within a box), the custom Path Tracer does well, giving very less noise and decent looking renders. It also renders quickly, while maintaining low usage of resources. For more complex scenes with multiple objects bound in a smaller area, Cycles is better.

The project lays foundation for a better Path Tracer! Optimizations and some more clever mathematics [2], [3], [17] can improve it by a lot! The project was hugely inspired and affected by previous work done by Sebastian Lague in Unity [15].

VII. Declaration

I, Varun Kumar Gupta, hereby declare that I have read and fully understood the plagiarism provisions as outlined in the General Regulations of the University Calendar for the current academic year, accessible at https://www.tcd.ie/calendar/. I certify that all work submitted in this assignment is my own, except where I have clearly acknowledged the use of works of other authors. I understand the consequences of violating these regulations.

References

[1] T. L. Kay and J. T. Kajiya, "Ray tracing complex scenes," *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, p. 269–278, aug 1986. [Online]. Available: https://doi.org/10.1145/15886.15916

[2] Shtille, "Box normal - aabbnormal," https://gist.github.com/Shtille/1f98c649abeeb7a18c5a56696546d3cf, 2021, accessed: 2024-04-10.

[3] Scratchapixel, "Ray-sphere intersection," https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection.html, accessed: 2024-04-10.

[4] OpenGL Tutorial, "Picking with a custom ray-obb function," https://www.opengl-tutorial.org/miscellaneous/clicking-on-objects/picking-with-custom-ray-obb-function/, accessed: 2024-04-10.

[5] K. Group, "Fragment shader," https://www.khronos.org/opengl/wiki/Fragment_Shader, accessed: 2024-04-10.

[6] iRendering, "Bucket vs. progressive rendering: Which one is better for your final renders?" https://irendering.net/bucket-vs-progressive-rendering-which-one-is-better-for-your-final-renders/, accessed: 2024-04-10.

[7] Intel, "Open image denoise," https://www.openimagedenoise.org/, accessed: 2024-04-10.

[8] LearnOpenGL, "Materials." [Online]. Available: https://learnopengl.com/Lighting/Materials

[9] Wikipedia, "Ray tracing (graphics)," https://en.wikipedia.org/wiki/Ray_tracing_(graphics), accessed: 2024-04-13.

[10] Nvidia, "What's the difference between ray tracing and rasterization?" https://blogs.nvidia.com/blog/whats-difference-between-ray-tracing-rasterization/, accessed: 2024-04-13.

[11] OpenGL Tutorial, "Picking with a physics library," https://www.opengl-tutorial.org/miscellaneous/clicking-on-objects/picking-with-a-physics-library/, accessed: 2024-04-13.

[12] Nvidia GPU Gems, "Chapter 38. high-quality global illumination rendering using rasterization," 2005. [Online]. Available: https://developer.nvidia.com/gpugems/gpugems2/part-v-image-oriented-computing/chapter-38-high-quality-global-illumination

[13] Cornell University, "The cornell box," 1998. [Online]. Available: https://www.graphics.cornell.edu/online/box/

[14] Carl von Bonin, "Opengl ray-tracing," https://github.com/carl-vbn/opengl-raytracing, Tech. Rep., 2022.

[15] Sebastian Lague, "Coding adventure: Ray-tracing," https://github.com/SebLague/Ray-Tracing, Tech. Rep., 2023.

[16] *Ray Tracing in One Weekend Series*, 2018. [Online]. Available: https://raytracing.github.io/

[17] D. Kuri, "Gpu ray tracer in unity," Tech. Rep., 2019. [Online]. Available: https://bitbucket.org/Daerst/gpu-ray-tracing-in-unity/src/Tutorial_Pt2/Assets/RayTracingShader.compute

[18] Nvidia, "What is path tracing?" Tech. Rep., 2022. [Online]. Available: https://blogs.nvidia.com/blog/what-is-path-tracing/

[19] Cycles Renderer, "Cycles renderer: Official website," https://www.cycles-renderer.org/, 2024, accessed: 2024-04-10.

[20] Ton Roosendaal, "Blender," https://www.blender.org/, 2024, accessed: 2024-04-10.

[21] Nvidia, "Nvidia optix: Ray tracing engine," https://developer.nvidia.com/rtx/ray-tracing/optix, accessed: 2024-04-11.

[22] Khronos Group, "Compute shader," https://www.khronos.org/opengl/wiki/Compute_Shader, 2019, accessed: 2024-04-12.