MRMA

February 5, 2025

Note: Even though this paper is produced for new developers; Ray Tracing is a calculus and physics focused subject and requires understanding Caluclus at an introductiary level (for example University Calculus I or equivalent)

Ray tracing is a modern technique to model the transportation of light. However, mostly used in pre-rendered applications; it is now making its way into video games. By its core ray tracing is an intensive task, often this term is keyed as "expensive". Therefore, to, increase the accessibility of video games to computers with a lower specification it is important to optimize light casting while retaining good results.

The time complexity of ray tracing depends on the number of rays, objects in the scene, and the iterations of tests. For this visualization, let variables: (n) be number of object(s), (r) be number of ray cast(s), and (k) be cost of intersection test(s) for a single object. The current method of ray tracing adopted is Naive Ray Tracing, which states: for every ray, the algorithm tests all (n) objects. The time complexity of the aforementioned can be plotted as such:

$$TimeComplexity = o(r \cdot n \cdot k)$$

On the other side of the spectrum, flood fill for lighting is a technique used to simulate light propagation by treating light as a "filling" process across a grid or scene. It starts from a light source and spreads outward to connected regions, filling areas within a certain distance or until blocked by obstacles. The algorithm typically uses depth-first search (DFS) or breadth-first search (BFS) to determine which areas are illuminated, making it suitable for simple 2D or voxel-based environments where realistic lighting isn't required. For this visualization, let variables: (n) be number of pixels (or nodes) in the region to fill and (m) be number of edges in the graph/grid.

Algorithm	Time Complexity (Naive)	Time Complexity (Optimized)
Ray Tracing	$(o(r \cdot n \cdot k))$	$(o(r \cdot \log(n) \cdot k))$
Flood Fill	$(o(n))$ *or* $(o(w \cdot h))$	N/A (typically not optimized further)

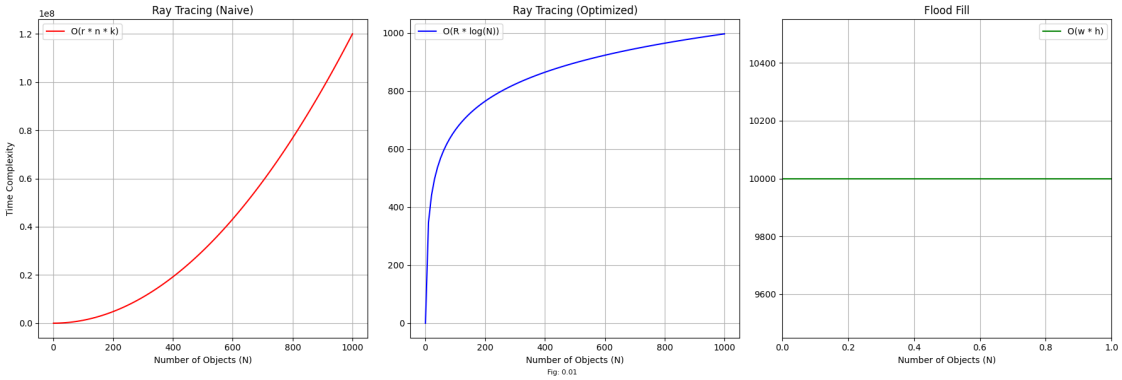

Fig: 0.01

0.1 Shadow casting

As seen in Figure 0.01 the time complexity of ray tracing is exponential, and is, therefore, expensive. To prevent this we can either: 1. Optimize the ray tracing algorithm by iterating over static non-updating objects and assigning a "skip" flag for the compute shader to ignore. However, this approach has limitations, creating a flag would populate the PCB bus with unnecessary effects and triggers that would further worsen the performance of other non-CPU dependent computational loads. Alternatively, "baking" in shadows for an immobile object is a viable option for real-time jobs with static weather and lighting effects as the need to compute shadows every frame is redundant. This, obviously has its own set of limitations, most notable of which are an unrealistic weather and lighting cycle, as well as, non-disruptive shadows without excessive math and collision testing. This is why a lot of graphically intensive and realistic games require strong processors and or graphic units. However, real-time rendering using ray-marching is not the only option. It is often unnecessary to use precise mapping techniques in parts of the video game where attention is not usually drawn. Therefore, it is common for competitive video game developers to approximate shadow rending in favor of ray-tracing. However, approximation at the end of the day is not a precise result and tends to fall apart in extreme circumstances.

To prevent that a second and very appealing option surfaces with promising results

2. Mathematical Ray Marching Approximation, or M.R.M.A is a mathematical approximation and ray marching hybrid algorithm focused on obtaining realistic shadows at a much lower computational cost, the algorithm can be defined by a group of equations:

$$S(P)_{intensity} = a \cdot S_{MSM}(P) + (1 - a) \cdot S_{SDF}(P)$$

1. The Moment Shadow Mapping (MSM) Component can be modeled as:

$$\mu_1(x, y) = \frac{1}{N} \sum_{i=1}^{N} z_i \mu_2(x, y) = \frac{1}{N} \sum_{i=1}^{N} z_i^2$$

Where:

- (x, y) are light-space coordinates of the texel
- z_i are depth values of the geometry in light space.

- N is number of samples per texel.

Furthermore, it is also important to model an equation to check the probability of occlusion

$$P_{occlusion}(P) = \frac{\mu_2 + d^2 - 2\mu_1 \cdot d}{\mu_2 - \mu_1^2}$$

or

$$P_{occlusion}(P) = \frac{\frac{1}{N}\sum_{i=1}^{N} z_i^2 + d^2 - 2\frac{1}{N}\sum_{i=1}^{N} z_i \cdot d}{\frac{1}{N}\sum_{i=1}^{N} z_i^2 - \left(\frac{1}{N}\sum_{i=1}^{N} z_i\right)^2}$$

2. Signed Distance Fields (SDF) Component

   The Signed Distance Field (SDF) defines the distance from any point P to the nearest surface Q:

$$\text{SDF}(P) = \min_{Q \in S} \|P - Q\| \cdot \text{sign}(P)$$

   Where:

   - S: The set of all surfaces in the scene.
   - sign(P):
   - $+1$ if P is outside the geometry.
   - -1 if P is inside the geometry.

   To compute local shadows, march from P along the light direction L in steps:

   1. Initialize $d_{\text{occlusion}} = 0$.
   2. At each step t, update $P_t = P + tL$.
   3. Query the SDF: $d_t = \text{SDF}(P_t)$.
   4. Accumulate $d_{\text{occlusion}}$ until:

   - $d_t < \epsilon$ (hit surface), or
   - $t > t_{\max}$ (no occluder).

   The shadow intensity from the SDF is:

$$S_{\text{SDF}}(P) = e^{-\beta \cdot d_{\text{occlusion}}} \cdot \cos(\theta)$$

   Where:

   - β: Softness control parameter.
   - $\cos(\theta) = \max(0, L \cdot N)$: The cosine of the angle between the light direction L and the surface normal N.

Here are few other things to keep in mind as you complete your algorithm:

Preprocessing:

1. Render the scene from the light's perspective to compute (μ_1) and (μ_2) for the shadow map.
2. Precompute the SDF for the scene geometry.

Per-Pixel Shadow Calculation:

1. Transform the pixel (P) into light space.
2. Compute $(S_{MSM}(P))$ using the MSM equations.
3. Perform SDF-based light marching to compute $(S_{SDF}(P))$.
4. Blend $(S_{MSM}(P))$ and $(S_{SDF}(P))$ using the unified shadow equation.

Optimization:

1. Use hierarchical grids or mipmaps for efficient SDF queries.
2. Limit SDF refinement to areas near the camera or shadow edges.

And finally here is the completely modeled equation:

$$S(P) = \sum_{i=k}^{\infty} \frac{d+k}{d} \cdot \max\left(0, 1 - \frac{\frac{1}{N}\sum_{i=1}^{N} z_i^2 + d^2 - 2\frac{1}{N}\sum_{i=1}^{N} z_i \cdot d}{\frac{1}{N}\sum_{i=1}^{N} z_i^2 - \left(\frac{1}{N}\sum_{i=1}^{N} z_i\right)}\right) + \left(1 - \frac{d+k}{d}\right) \cdot e^{-\beta \cdot d_{\text{occlusion}}} \cdot \max(0, L \cdot N)$$

The equation models light traveling through ray marching and still offers a greater performance than optimized ray tracing. The time complexity can be modeled as such: $O(R \cdot S)$ and is linear

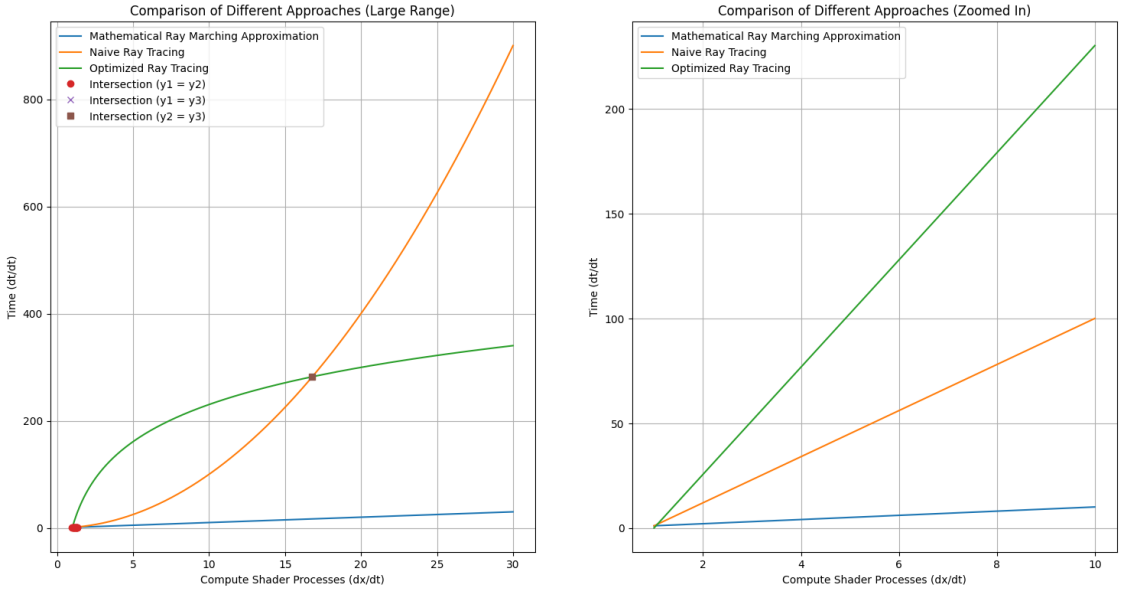

Fig: 0.02

0.2 Refraction, critical point, and other light effects

However, the aforementioned equations only deal with shadows and are not inclusive of other aspects often associated with Ray Tracing, such as, specular highlghts, normal maps, refraction, and defraction. Lets first tackle with the most compuuttionally expensive task, refraction.

Let's first start with understand how refraction works at it's core. The point of refraction can be defined with Snell's Law:

$$n_i \cdot \sin\theta_i = n_f \cdot \sin\theta_f$$

Which can be visualized as such.

The line between the vaccum and medium can be represented as S_n, or the surfce normal. where the shift in the ray (angle of incidient) is called the Index of Refraction (IOR) and can be represented as θ this number is modified by n in the formula above and is independent to each medium.

However, doing trignometric functions on a GPU shader can be extremely slow due to its low clock-speed, high quantity core set-up. To mitigate this problem we can sub in the taylor's series for $\sin\theta$:

$$\sin x = \sum_{n=0}^{\infty} (-1)^n \frac{x^(2n+1)}{(2n+1)!} \quad for \quad |x| \ \leq \ z$$

However, this is only for case zero, assuming that the angle of attack is not critical or negative

to deal with those exceptions let us set up a piecewise function, which can be modeled using a switch case model.

let the function modeled be $f(x)$

$$f(x) \ =$$

$$\sin x \ = \ \sum_{n=0}^{\infty} (-1)^n \frac{x^(2n+1)}{(2n+1)!} \quad for \quad |x| \ \leq \ 1$$

$$\theta_{critical} \ = \ 0 \quad for \quad \theta_{critical} \ = \ \arcsin(x) \ \Rightarrow \ \arcsin\left(\frac{n_f}{n_i}\right) \ \Rightarrow \ \sum_{n=0}^{a} \frac{(2n)!}{2^2 n \cdot (n!)^2 \cdot (2n+1)} \cdot 2n{+}1$$

$$-\sin x \ = \ \sum_{n=0}^{\infty} (-1)^n \frac{x^(2n+1)}{(2n+1)!} \quad for \quad x \ \leq \ 0$$

The conversion from an arcsin into a taylor's seris function is extremeley benifical in hardware acceleration based workloads where a Graphics Processing Unity (GPU) pipeline with multiple low boost clock cores are doing (a)synchronus tasks to reudce at real time rendering costs.

the equation of diffrence can be modeled using an integral between the diffrence of the two equations, in a high fidelity real time computational load the time complexity of the functions can be modeled as such:

$$T_{taylor} \quad = \quad O(n)$$

whilst, raw computation of arcsin using something like the Newton-Rhapson algorithim or the CORDIC is possible it creates a logarythimic time complexity which can be modeled as such:

$$T_{CORDIC} \quad = \quad O(\log_z(m))$$

As mentioned before the diffence in computations can be modeled using the integral of the two previous functions, which can be modeled as such:

Note: the use of symbol ϕ (phi) in this case is to display an optional use case

$$C_{diffrence} = \int_{a_\phi}^{b_\phi} \left(T_{taylor}(x) - T_{CORDIC}(x)\right)\, dx$$

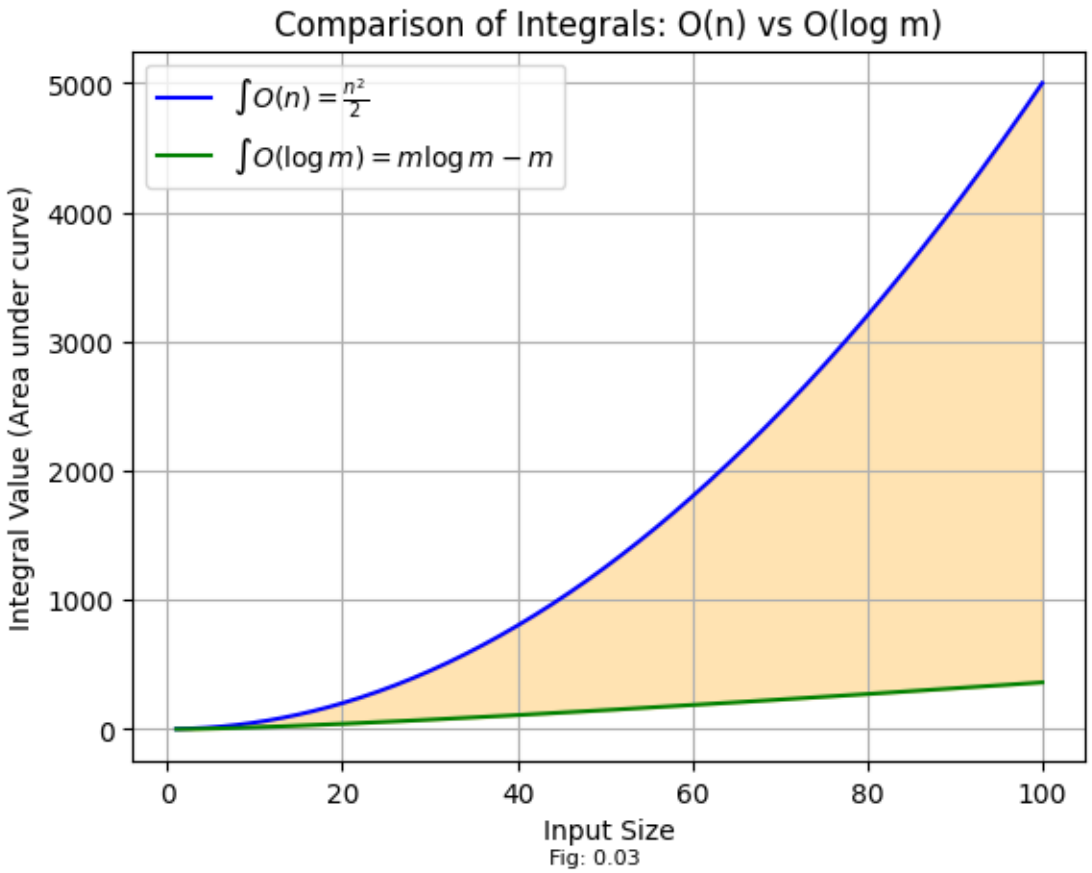

Fig: 0.03

As seen in Figure 0.03 conversion into Taylor's series is much more benifical for the algorithim. This was because of the previousley mentioned structure of GPU shaders and pipeline effects. An easy way to visualize this by thinking of the GPU as a sieve, whilst the CPU as a single pipe.

0.3 Specular Highlights, Bloom, Anistrophic Filtering, and DLSS/XeSS/RSR.. Super Resolution

To understand casting and maping specular higlights on 3D Objects it is important to have a core understanding for 3D Voxels. Voxels are simillar to a pixel in the sense that they can hold a set amount of data in a 3D space.

We can better visualize a voxel as a class:

```
private class Voxel3D {
    private Voxel3D (float[] { x, y, z }, float r ...) {
        ...
    }
}
```

Obviously, you wont be writing a GPU shader in C#, however, it provides easy to visualize code that can be understood with minimal programing knowledge