EC526 Final: Ray Tracing

Jordan Nichols

6 May 2022

Contents

1 Background Information

1.1 What is Ray Tracing?

Ray tracing is a rendering technique that generates images by tracing the path of light rays and simulating their interactions with virtual objects to color pixels. Compared to other rendering techniques, ray tracing can produce very high-fidelity images; however, this comes at the cost of a much longer run time. Typically, this makes ray tracing a suitable technique for applications such as animation, image creation, television effects, and other scenarios where time is not as much of a constraint. It is normally less suited, however, for real-time applications such as video games.

1.2 Parallelization Benefits

Ray tracing is well suited for parallel systems, such as GPUs and multi-core architectures, because each pixel's color can be calculated independently of every other pixel on the screen. This means that, with enough threads, large blocks of pixels (or even every individual pixel) can be calculated simultaneously.

1.3 Algorithm

Determining the color of a pixel is done by first computing the ray from the camera to the center of a pixel, then finding which, if any, objects in the scene the ray intersects with. Based on the material of the object, we can then reflect or refract scattered rays and return the color of the pixel.

1.3.1 Finding the Ray

Let E be the eye position, T the target position, θ the field of view, m, k the height and width of the viewport, i, j the indices of some pixel, and $\vec{v}$ a vector indicating up and down.

To calculate rays in a rectangular viewport, we use the following algorithm:
On input, the following are true and the values are known:

$$\begin{aligned} E &\in \mathbb{R}^3 \\ T &\in \mathbb{R}^3 \\ \theta &\in [0, \pi] \\ m, k &\in \mathbb{N} \\ i, j \in \mathbb{N}, 1 \leq i \leq k \wedge 1 \leq j &\leq m \\ \vec{v} &\in \mathbb{R}^3 \end{aligned}$$

The goal is to find the position of the center of each pixel P_{ij} to get the vector $\vec{p}_{ij} = P_{ij} - E$. First, we find vectors $\vec{t}$ and $\vec{b}$, then normalise them along with $\vec{v}$:

$$\vec{t} = T - E, \quad \vec{t}_n = \frac{\vec{t}}{||\vec{t}||}$$
$$\vec{b} = \vec{v} \times \vec{t}, \quad \vec{b}_n = \frac{\vec{b}}{||\vec{b}||}$$
$$\vec{v}_n = \vec{t}_n \times \vec{b}_n$$

Next, calculate the size of the viewport, h_x, h_y, divided by two:

$$g_x = \frac{h_x}{2} = d \tan\frac{\theta}{2}$$
$$g_y = \frac{h_y}{2} = g_x \frac{m-1}{k-1}$$

Then find the next-pixel shifting vectors, $\vec{q}_x$ and $\vec{y}_x$, parallel to the viewport and the vector for the center of the bottom-left pixel center, $\vec{p}_{1m}$

$$\vec{q}_x = \frac{2g_x}{k-1}\vec{b}_n$$
$$\vec{q}_y = \frac{2g_y}{m-1}\vec{v}_n$$
$$\vec{p}_{1m} = \vec{t}_n d - g_x \vec{b}_n - g_y \vec{v}_n$$

Now that the next-pixel shifting vectors and the bottom-left pixel center vector are found, scale both shifting vectors by $i-1$ and $j-1$ and add them to $\vec{p}_{1m}$ to find $\vec{p}_{ij}$.

$$\vec{p}_{ij} = \vec{p}_{1m} + \vec{q}_x(i-1) + \vec{q}_y(j-1)$$

1.3.2 Ray-Sphere Intersection

A point (x, y, z) on a sphere centered at (C_x, C_y, C_z) with radius r will satisfy the following equation:

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2$$

Converting this to vector form, where the vector from $\mathbf{C} = (C_x, C_y, C_z)$ to $\mathbf{P} = (x, y, z)$ is $(\mathbf{P} - \mathbf{C})$, we get the equation

$$(\mathbf{P} - \mathbf{C}) \cdot (\mathbf{P} - \mathbf{C}) = (x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2$$

or

$$(\mathbf{P} - \mathbf{C}) \cdot (\mathbf{P} - \mathbf{C}) = r^2$$

We take our ray $\mathbf{P}$ and rewrite it as $\mathbf{P(t)} = \mathbf{A} + t\mathbf{b}$. If $\mathbf{P(t)}$ hits our sphere, then there is some $\mathbf{t}$ for which $\mathbf{P(t)}$ satisfies our sphere equation. Rewriting the equation gives

$$(\mathbf{A} + t\mathbf{b} - \mathbf{C}) \cdot (\mathbf{A} + t\mathbf{b} - \mathbf{C}) = r^2$$

and moving all terms to the left-hand side gives

$$t^2\mathbf{b} \cdot \mathbf{b} + 2t\mathbf{b} \cdot (\mathbf{A} - \mathbf{C}) + (\mathbf{A} - \mathbf{C}) \cdot (\mathbf{A} - \mathbf{C}) - r^2 = 0$$

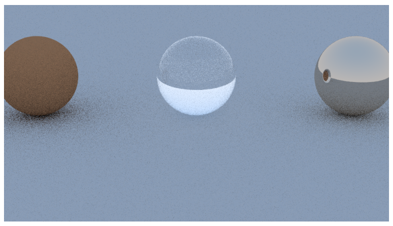
The only unknown is t, giving us a quadratic equation to solve. Therefore, we can calculate the discriminant of the equation to determine whether there are two solutions (the ray passes through the circle), one solution (the ray is tangent to the sphere), or no solutions (the ray does not intersect the sphere). We can simplify our discriminant as we know there is a factor of 2 in b. Replacing b with $2h$ gives the following:

$$\begin{aligned}
&\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\
&\frac{-2h \pm \sqrt{(2h)^2 - 4ac}}{2a} \\
&\frac{-2h \pm \sqrt{4(h^2 - ac)}}{2a} \\
&\frac{-2h \pm 2\sqrt{h^2 - ac}}{2a} \\
&\frac{-h \pm \sqrt{h^2 - ac}}{a}
\end{aligned}$$

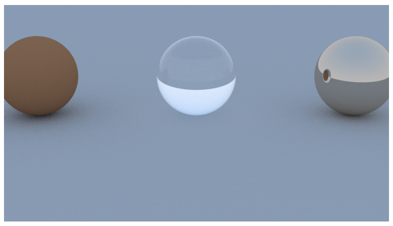
To find whether a ray intersects a sphere, all we must solve is $(\frac{b}{2})^2 - ac$.

1.3.3 Differences between Algorithm and Code

The main difference between the algorithm and the implementation of the algorithm is that a set number of rays (defined as `SAMPLES_PER_PIXEL`) are calculated at each pixel, each with slightly offset endpoints by a random float value between 0 and 1. This change provides an anti-aliasing effect and creates more realistic shadows. The below images demonstrate the difference between relatively low and high values of `SAMPLES_PER_PIXEL`).



(a) `SAMPLES_PER_PIXEL = 5`



(b) `SAMPLES_PER_PIXEL = 100`

Figure 1: Difference between high and low values of `SAMPLES_PER_PIXEL`. Notice the lack of shadow blurring in (a), along with the jagged edges along the spheres.

2 Code

2.1 Scalar

The following pseudo-code is the implementation of ray tracing used.

```
for j in IMG_HEIGHT:
    for i in IMG_WIDTH:
        for s in SAMPLES_PER_PIXEL:
            float u = (i + random_float()) / (IMG_WIDTH - 1);
            float v = (j + random_float()) / (IMG_HEIGHT - 1);
            ray r = get_ray(u, v);
            pixel_colors[i][j] += ray_color(r, world, MAX_DEPTH);
```

2.2 Runtime Efficiency

As seen above, we generate a set number of rays per pixel, each offset slightly by a random float. The efficiency is $O(nms)$, where n is the height of the image, m is the width of the image, and s is the number of samples taken for each pixel. In edge cases, the number of reflections per ray can also affect the runtime significantly, for example in a scene with many low-fuzz metallic objects or dielectric objects. However, for the purposes of this project, the contribution of the reflections will not be considered.

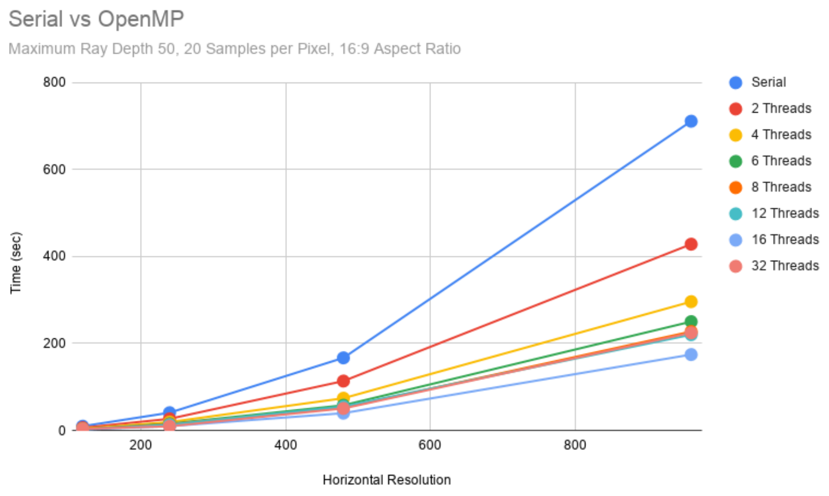
2.3 OpenMP

Using OpenMP, we can parallelize each loop in the main function, giving each thread a chunk of the rays to calculate. The number of threads tested where 2, 4, 6, 8, 12, 16, and 32. A static scene was used, and different horizontal resolutions were tested as well. It is worth noting that the vertical resolution scaled with the horizontal resolution because the aspect ratio is set to 4:3. Because of this, the runtime will increase exponentially no matter the number of threads.

2.4 Parallelization Results

Table 1: Timing of different OpenMP Thread Counts

Horizontal Resolution (px)	120	240	480	960
Basic	9.75s	40.86s	167.06s	711.46s
2 Threads	6.28s	26.78s	113.66s	428.56s
4 Threads	4.05s	19.30s	74.15s	296.31s
6 Threads	2.87s	15.46s	57.96s	250.39s
8 Threads	2.53s	14.46s	52.20s	227.62s
12 Threads	2.36s	13.35s	55.03s	220.34s
16 Threads	2.19s	9.90s	39.82s	174.61s
32 Threads	3.67s	9.56s	50.70s	224.64s



The 16-threaded code outperformed all other thread counts. Compared to the scalar code, it performed nearly 4 times as fast at every tested resolution. It is interesting to note that thread count was correlated with an increase in runtime all the way up to 32 threads, where performance began to decrease. This could be due to the CPU on the SCC not having enough cores to support the use of 32 threads, and the overhead of context switching impacting the runtime.