# Parallelization of Ray Marching for Efficient Rendering of Objects and Scenes

Jack Rowe (jbr2826, jackrowe)

April 2024

## 1 Overview

This project was created in C++ using OpenMP, and visualizations were further produced with Python and Matplotlib. The overall goal is to determine the effect of parallelization on the ray marching algorithm, which was also paired with several modifications to better mimic the natural world. This algorithm can produce highly realistic images at extreme ends, and this project is a great stepping stone to producing extremely efficient rendering. This also relates to scientific computing through several avenues, however the most relevant would be the study of optics and mirrors. Simulations like these can help researchers better understand various setups, especially with predicting outcomes of more complicated ones.

## 2 Ray Marching

The core idea behind ray marching is very similar to ray tracing. Rays are sent out from each pixel of the camera, and interact with the scene to produce colors for each pixel. Ray tracing solves intersection functions in order to determine collision points for these light rays, whereas ray marching uses signed difference functions to take appropriately sized steps towards surfaces. This process is done iteratively, and is an excellent candidate for parallelization through multithreading as each ray is completely independent from every other way.

## 3 Implementation

### 3.1 File Format

#### 3.1.1 CMakeLists.txt

This file contains the CMake instructions for building the project using OpenMP.

#### 3.1.2 output_times.sh

This Shell script is used to repeatedly run the compiled executable with different parameters in order to compile data into a `times.txt` file.

#### 3.1.3 visualize.py, graph.py

These Python scripts are used to generate the final rendered image in PNG format and to render the Matplotlib graphs of the parallelization timing data.

#### 3.1.4 raymarch.h, raymarch.cxx

These C++ scripts contain the headers and methods of the ray marching algorithm.

#### 3.1.5 main.cxx

This C++ script contains various signed distance functions and colors, and is used to set up and run the ray marching algorithm for various scenes. Output to a `output.txt` file is managed through this script as well.

### 3.2 Rays, Colors

A ray has six pieces of public data:

```
int iterations;
float x, y, z, theta, phi;
Color color;
```

Iterations is used to keep track of how many times the ray position has been moved through the scene. x, y, and z are positional variables in 3D space, and theta and phi are spherical angles to determine the rays direction. The ray is also assigned a Color object, which is comprised of three integers for RGB values. Colors are reassigned during collisions with opaque objects, and are used to generate the output from each ray.

### 3.3 SceneObjects, Scene

Objects are defined as SceneObjects, which have the following data:

```
SceneObjectType type;
Color color;
std::vector<float> extraParam;
std::function<float(float, float, float)> sDF;
```

SceneObjectType is an enumerator with only two values at the moment (`OPAQUE`, `MIRROR`), but was implemented for readability and for future expansion into other object types such as transparent objects like lenses. Color is also assigned here, and is used to change the color of rays as they impact the object. extraParam is unused in this project, but was originally implemented to store data such as permissivity and index of refraction. The signed distance function (sDF) represents the distance to the object from any point in 3D space. Scenes have the following data:

```
std::vector<SceneObject> objects;
Color color;
```

Scenes are a compiled vector of all the objects that the user wishes to render, along with a background color to assign to rays which do not end up colliding with a surface.

## 3.4 Camera

The camera is probably the most complicated object, and has the following data:

```
float x, y, z, theta, phi;
float fov;
int width, height;
Scene scene;
float collisionThreshold;
float distanceCutoff;
std::vector<std::vector<Ray>> rays;
```

Similarly to rays, the camera has its own positional and orientational data for 3D space. It also has field of view, width, and height parameters for determining the respective specifications of the final image. collisionThreshold is used to determine at what distance from the object causes rays to collide, and distanceCutoff is used in distance fog. The camera also has a respective scene and set of rays it uses to render the image data.

## 3.5 Visual Effects

### 3.5.1 Ambient Occlusion

Ambient Occlusion was implemented using the amount of iterations on each ray before they collided, which is roughly proportional to the surface complexity at that point (i.e. small crevices). This makes it harder for light to escape, as it is more likely to get absorbed with more bounces. Colors are linearly interpolated between the collided object's color and the scene's background color depending on a variable $t$ determined by the following equation:

$$t = 1 - \frac{1}{1 + 0.2log(iterations + 1)}$$

### 3.5.2 Distance Fog

Without distance fog, it is surprisingly difficult to tell whether a circle or sphere is being rendered. The further away from the camera a ray collides, the closer to the background color it becomes. Colors are linearly interpolated between the collided object's color and the scene's background color depending on a variable $t$ determined by the following equation:

$$t = \frac{distanceToCamera^2}{distanceCutoff^2}$$

## 4 Optimizations & Generalizations

### 4.1 Signed Distance Functions

Signed distance functions are probably the most beautiful and elegant section of this algorithm. They allow intensely complicated objects to be rendered at nearly no extra cost. Additionally, these functions can be used to render objects that would otherwise be extremely difficult to describe to a rasterizer, such as 3D fractals. An example of a S.D.F. for a sphere is given below:

```
float sphereSDF(float x_in, float y_in, float z_in,
                float x, float y, float z, float r) {
  float dx = (x_in - x);
  float dy = (y_in - y);
  float dz = (z_in - z);
  return std::sqrt( dx*dx + dy*dy + dz*dz) - r;
};
```

x, y, z, and r describe a sphere of radius r centered at these coordinates, and x_in, y_in, and z_in represent any point in 3D space. In the code, these are all passed as lambda arguments in order to provide maximum generalization to any shape describable under this format.

### 4.2 Modulo Operations

An interesting property of S.D.F.s is that they can contain modulo operators. This allows extremely easy ”cloning” of any object into an infinite swarm of them. A S.D.F. for such a swarm of spheres is given below:

```
float swarmSDF(float x_in, float y_in, float z_in,
               float x, float y, float z, float r, float spacing) {
  float dx = std::fmod(std::abs(x_in), spacing) - x;
  float dy = std::fmod(std::abs(y_in), spacing) - y;
  float dz = std::fmod(std::abs(z_in), spacing) - z;
  return std::sqrt(dx*dx + dy*dy + dz*dz) - r;
};
```

This is very similar to the sphereSDF function from earlier, however it includes an additional spacing argument for how much distance should be between each sphere on a grid.

### 4.3 Squared Distance

For the distance fog, squared distance was used for two reasons. Primarily, using the absolute distance requires a square root function, which when combined with floats caused occasional unpredictable casting to integers. Thankfully, using the squared version does not cause much difference visually, and is actually faster given that there is no square root involved.

## 4.4 Normal Vectors

In order to implement mirrors (and lenses in the future), every reflective or transparent object in the scene has to have associated normal vectors for every point on its surface. In order to keep the program's generalization to any object describable with an S.D.F., an approximate version was implemented using the gradient of the surface.

```
std::vector<float> SceneObject::CalculateNormal(float x, float y, float
    z, float eps=0.001) {
  float dx = sDF(x + eps, y, z) - sDF(x - eps, y, z);
  float dy = sDF(x, y + eps, z) - sDF(x, y - eps, z);
  float dz = sDF(x, y, z + eps) - sDF(x, y, z - eps);

  float mag = std::sqrt(dx*dx + dy*dy + dz*dz);
  if (mag != 0.0f) { dx /= mag; dy /= mag; dz /= mag; }

  return {dx, dy, dz};
};
```

This is then converted to spherical angles, and used to calculate reflection angles for rays.

## 5 Parallelization with OpenMP

Parallelization of the main loop in the `Camera::Marc()` method proves to be decently efficient. Given that each ray is unpredictable in how many iterations it might take to collide and terminate, a dynamic scheduling system was employed. Additionally, `nowait` is applicable here because the threads do not need to communicate information to each other. `collapse(2)` is used to flatten the 2D array to be parallelized.

```
void Camera::March(int iter) {
#pragma omp parallel
#pragma omp for schedule(dynamic) nowait collapse(2)
  for (int i = 0; i < width; i++) {
    for (int j = 0; j < height; j++) {
      bool terminate = false;
      for (int k = 0; k < iter && !terminate; k++) {
      ...
```

## 6 Examples, Time Comparison

### 6.1 Collection of Spheres

This scene is comprised of three different spheres of three different colors. This is the simplest example illustrating most of the components discussed so far. Notice the distance fog, which creates bright spots towards the middle of spheres and darkens their edges. Additionally, the region in between all three spheres (around just above the blue sphere and a little to the left) is somewhat darker as ambient occlusion becomes non negligible. These effects are best viewed on a PNG viewer with zoom tools.

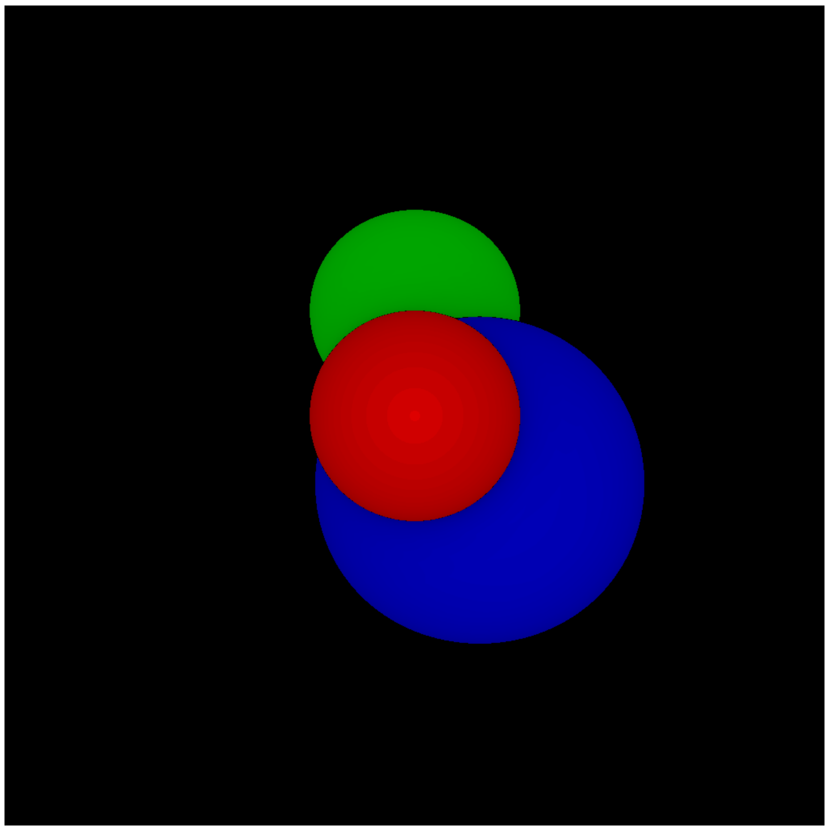

Figure 1: Collection of Spheres (1024x1024), rendered at 100 iterations in 3.16s on 32 threads

## 6.2 Mirror Plane

This scene is very similar to the previous, with the addition of a mirror plane just above the midsection of the green sphere. This over doubles computation time, but it provides a very interesting effect and preserves distance fog and ambient occlusion across the surface. Notice that ambient occlusion from the mirror is present on the spheres, which darkens them a bit more than the previous scene.

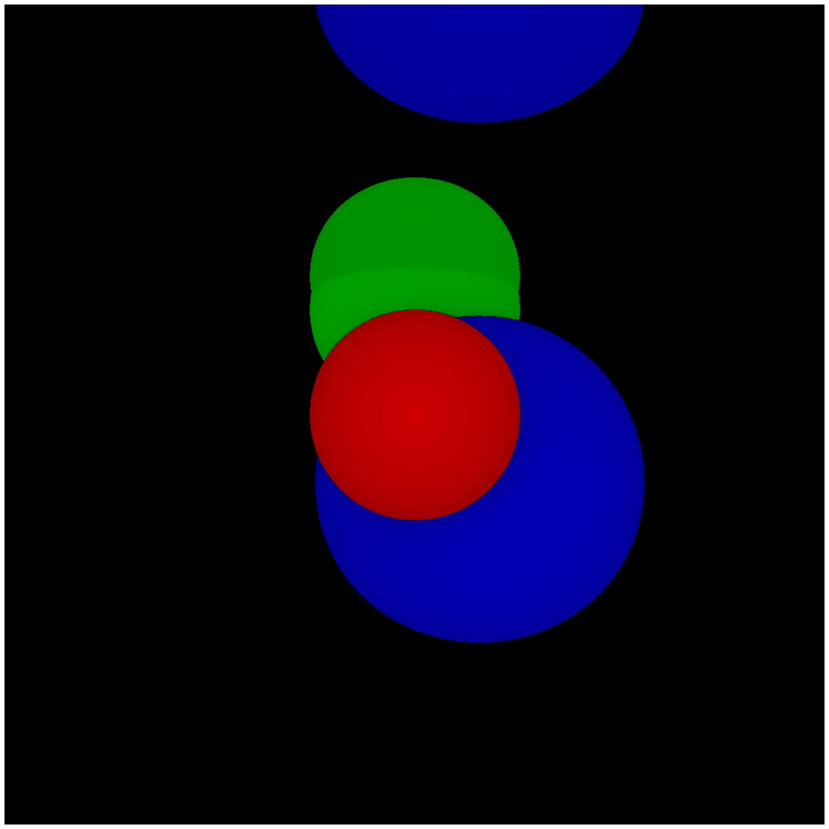

Figure 2: Mirror Plane (1024x1024), rendered in 7.49s at 100 iterations on 32 threads

## 6.3 Sphere Swarm

This scene is generated using the sphere swarm S.D.F. discussed earlier. It technically extends into infinity, however the distanceCutoff value for the distance fog limits how many spheres are visible here. Fractal-like patterns such as this one are some of the more beautiful visualizations possible with this algorithm. Of extreme importance is noting thaat this image required the least computation time to generate, clocking in at over a full second faster (2.07s) than Collection of Spheres (3.16s).

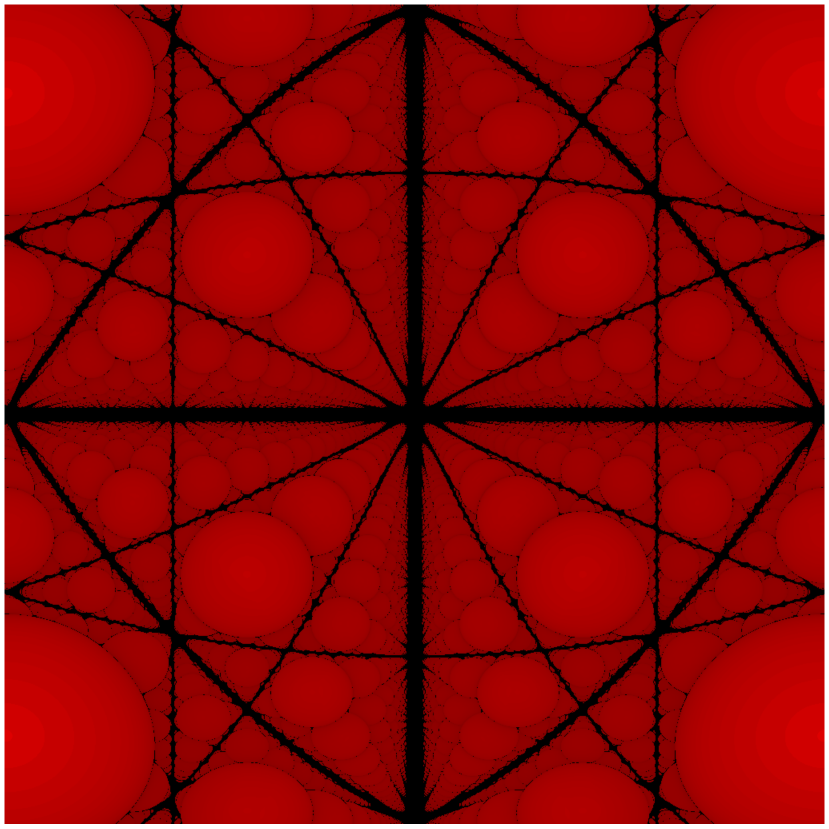

Figure 3: Sphere Swarm (1024x1024), rendered in 2.07s at 100 iterations on 32 threads

Additionally, this scene was re-rendered at 512x512 resolution across various thread counts (1-32). Two curves are shown, with one at a maximum of 25 iterations per ray, and one at 100. The image generated at 32 threads and 25 iterations is also shown below.

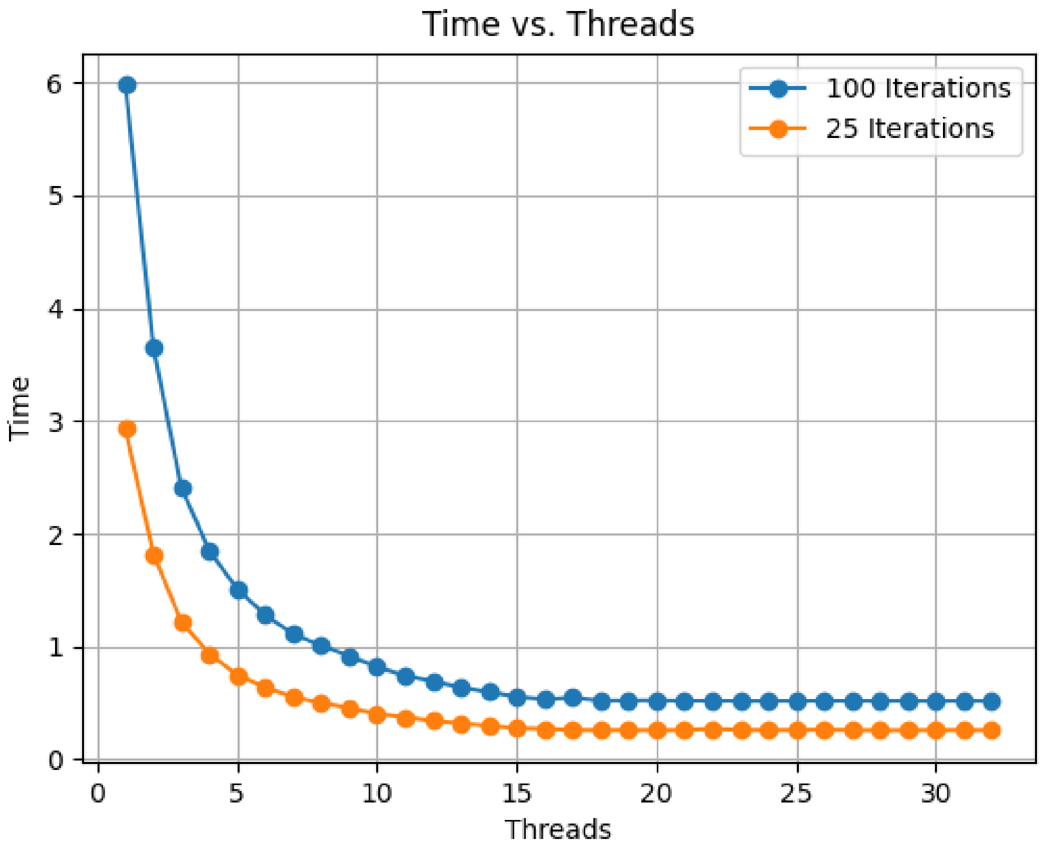

Figure 4: Sphere Swarm (512x512) Times on Multiple Threads for 25 and 100 Iterations

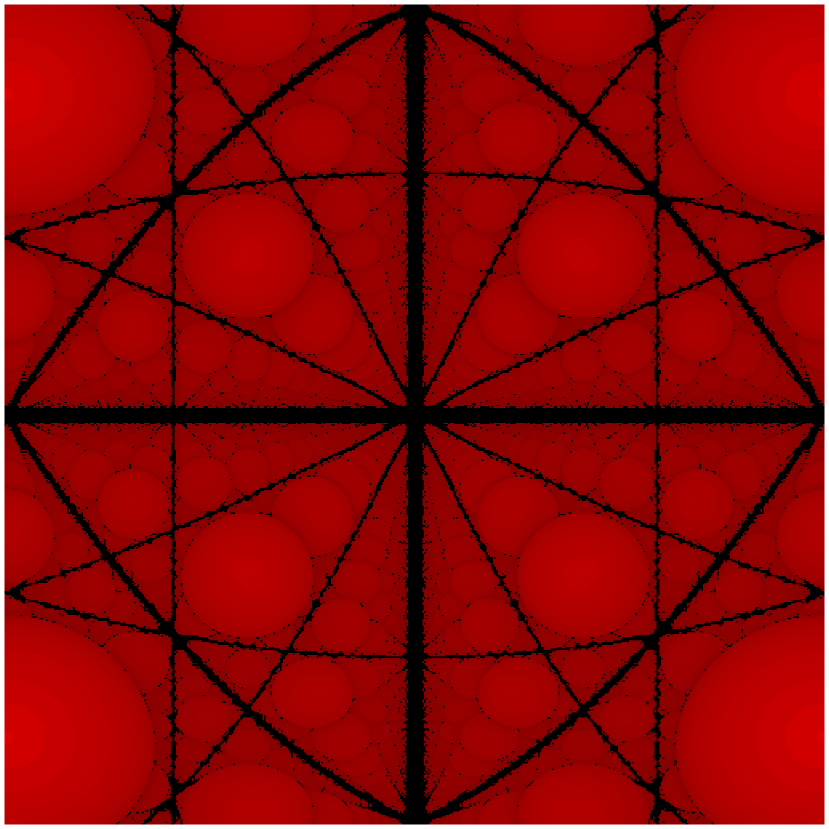

Figure 5: Sphere Swarm (512x512), rendered in 0.258s at 25 Iterations on 32 Threads

## 7 Future Explorations

In the future, it would be extremely interesting to try implementing lenses and transparent objects. Many of the S.D.Fs generally used for this support negative values inside the objects. Marching along the absolute value in this region after refraction would be a great approach to creating lenses, and very closely mimics how they work in the real world.

It would also be interesting to explore how well this algorithm adapts to real-time rendering. The iteration count used in these experiments was significantly higher than needed for a lot of video games or other retro-styled media. It is possible that this could be used to create physics over complex environments as well, but might be extremely costly computationally.