Jamin Valick

CSE 557

4/22/2024

Final Project: Parallelization of Ray Tracing Algorithm

Introduction:

Ray tracing is one of the most popular ways to render realistic images by simulating light ray as they travel through a virtual scene. As video game developers seek to make their products more realistic, real time ray tracing has become a hot topic for the industry. The algorithm is far more computer intensive than traditional rasterization, which poses a problem for rendering full frames quick enough to get 30 or more frames per second. The most fundamental method for reducing frame render time and thus increasing frame rate is to utilize parallel computing. This is typically done utilizing the many cores of a GPU, but for the sake of time constraints and simplicity, this project seeks to operate on OpenMP to utilize the CPU to its full potential and discover the best method of parallelization.

Background:

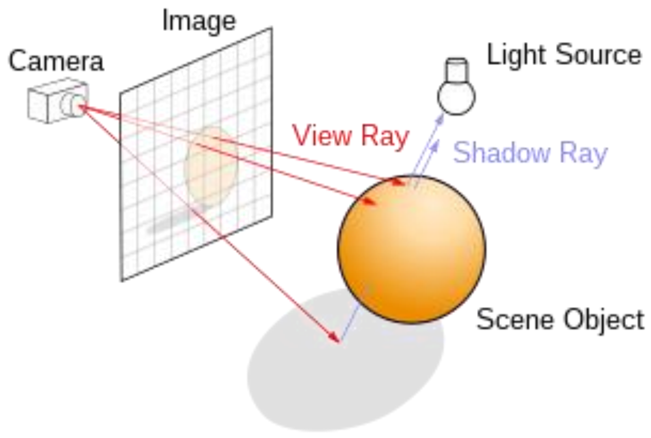

Figure 1.

Source: https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

In order to understand why we must parallelize ray tracing, we must understand how the algorithm itself functions. At a high level, ray tracing must find the color of each individual pixel on a screen. A virtual camera sits behind the screen in the scene and shoots a ray from the camera through every pixel. The ray is then checked for intersections with objects in the scene. If the ray does not intersect with anything on the scene, the background color is immediately returned with no further calculations. If the ray intersects a shape, the color of that location is calculated via Phong shading (ambient + specular + diffuse). A shadow ray is pointed from the location to the position of the light source. If the shadow ray reaches the light with no interference, then the light's color is added to the contributing color of that location via specular and diffuse lighting.

Additionally, a refraction and reflection ray are calculated at the position and the ray tracing function is recursively called for each new ray, and their results contribute to the overall color of that position. One pixel can end up recursively calling the ray trace function up to $2^{n+1} - 2$ where n is the maximum depth of the recursion.

For this project, it was decided to also add additional complexity to discover more interesting results. To reduce aliasing on edges, multiple samples can be taken for each pixel with slight directional offsets for each projected ray. The colors for each sample are added and averaged. This produces a smoothing effect that is essentially antialiasing. The edges of shadows can also be smoothed by averaging the results of multiple shadow samples. The side effects of this are a dramatic increase in runtime and the addition of noise. The more samples that are taken, the more work must be done, yet it results in a reduction of noise.

The following images represent the scene that was used for this project. Figure 2 is the scene rendered with only 1 sample per pixel and 1 shadow ray per ray trace call. Figure 3 is the scene with 8 samples per pixel and 8 shadow rays per ray trace call. Visual differences can be seen in the edges of the ceiling, edges of the spheres, and the edges of shadows cast by the spheres. This scene also includes intersections of triangles and spheres, one point light source, reflections, refractions, a maximum recursion depth of 3 (i.e. 14 calls),

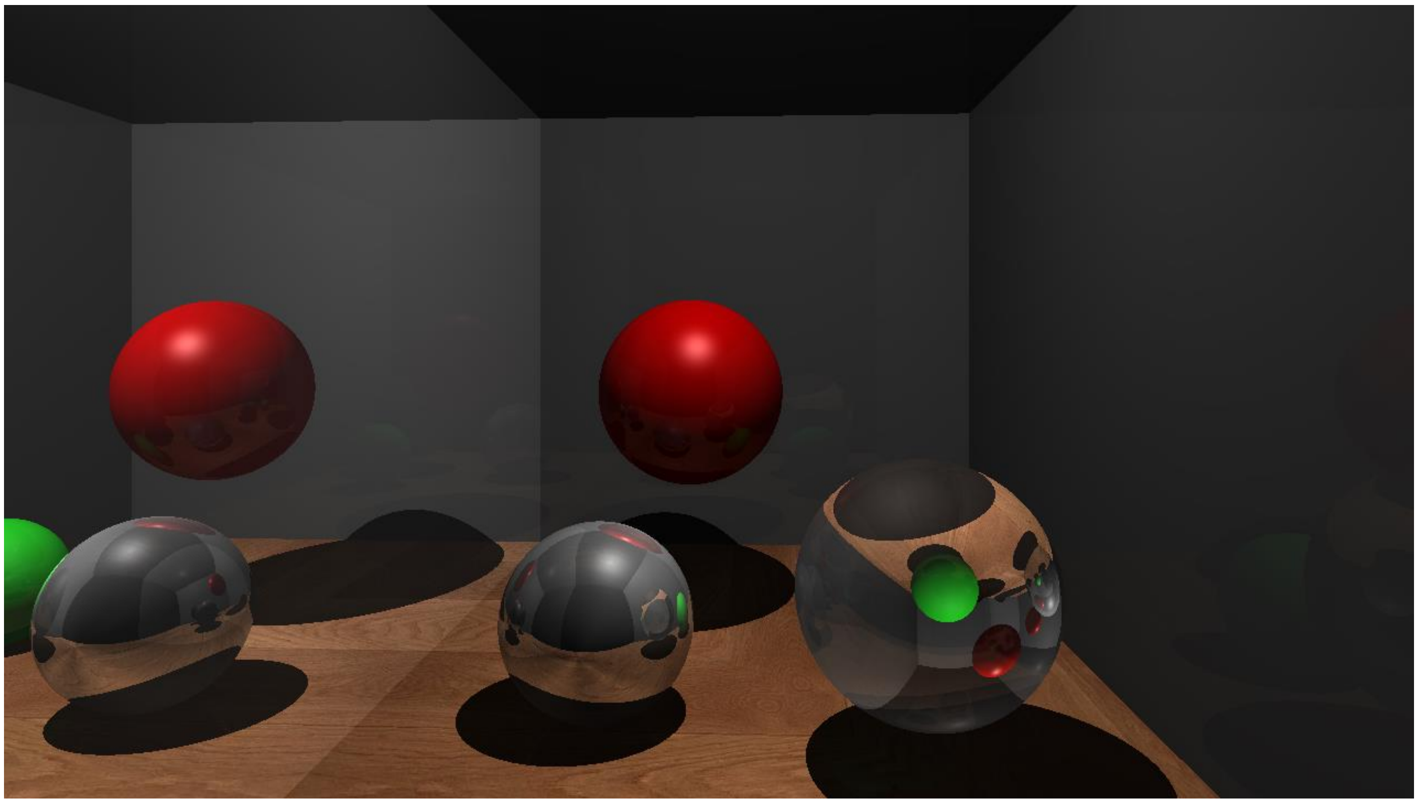

Figure 2.

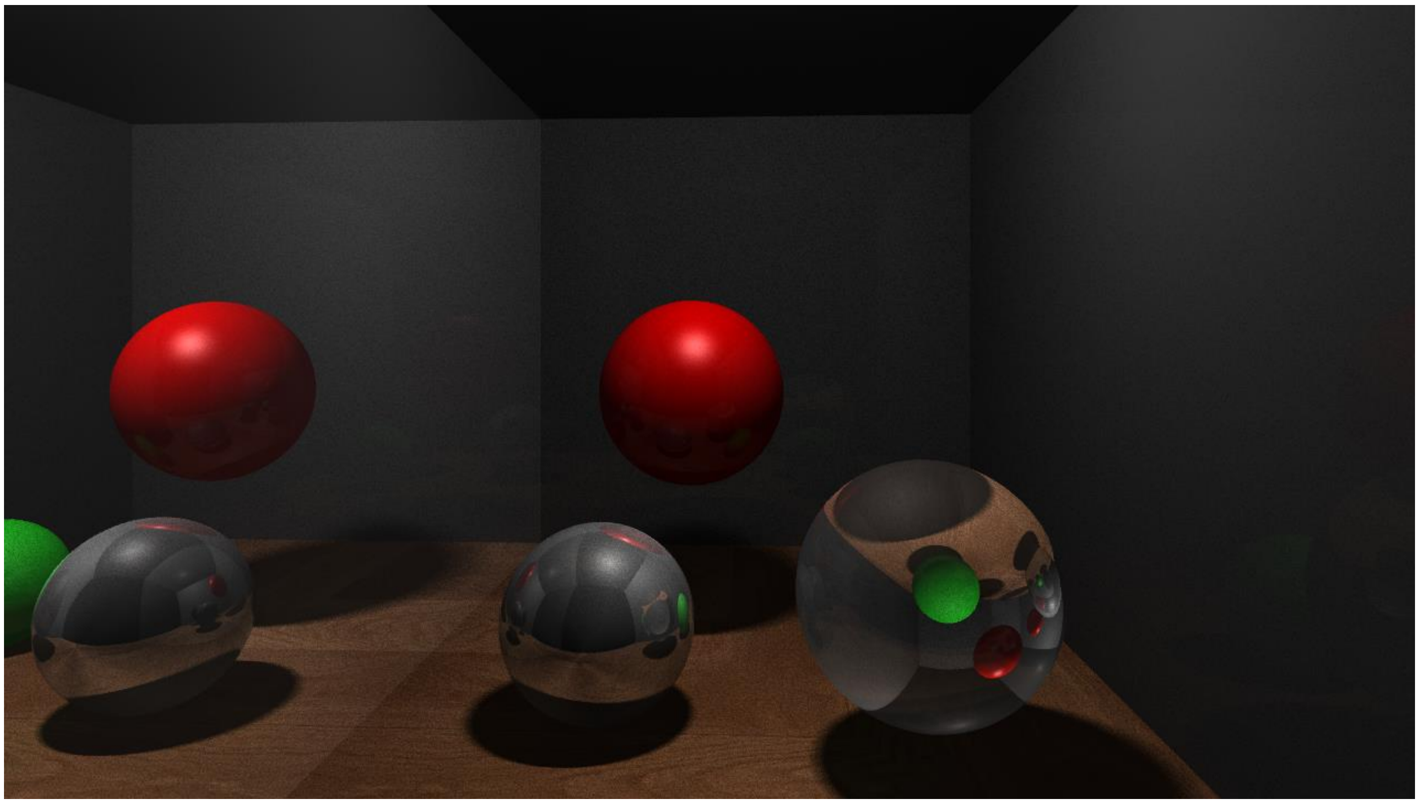
Figure 3.

As evident by the explanation of ray tracing, there is a lot of processing that goes into calculating the color of each pixel. Thus, it is imperative to parallelize this project to render frames quicker. The system used for this project is an AMD Ryzen 7 3700X with 8 cores and 16 threads. While it will still not be possible to render frames quick enough for real time interaction, it would be much easier to render a short video in a reasonable time.

Parallelization:

The overall ray tracing program is driven by a nested for loop. The simplified version is shown below.

```
for (int y = 0; y < SCR_HEIGHT; y++)
    for (int x = 0; x < SCR_WIDTH; x++)
        for (int sample = 0; sample < SAMPLES_PER_PIXEL; sample++)
            currentColor += rayTrace(eye, direction);
        currentColor /= SAMPLES_PER_PIXEL;
        image[x + y * SCR_WIDTH] = currentColor
```

Figure 4.

The first outermost loop is for every row y and the next loop is for every pixel x in a row y. The innermost loop is for every sample of pixel (x,y). Each pixel is calculated independently, so any combination of OpenMP pragmas can be used on these three loops. This means that the program can be divided by rows, pixels, or samples. For the y loop, each thread will be assigned a row to work on. For the x loop, each thread will be assigned a pixel to work on for the current row. For the sample loop, each thread will be assigned a sample to work on for the current pixel.

Results:

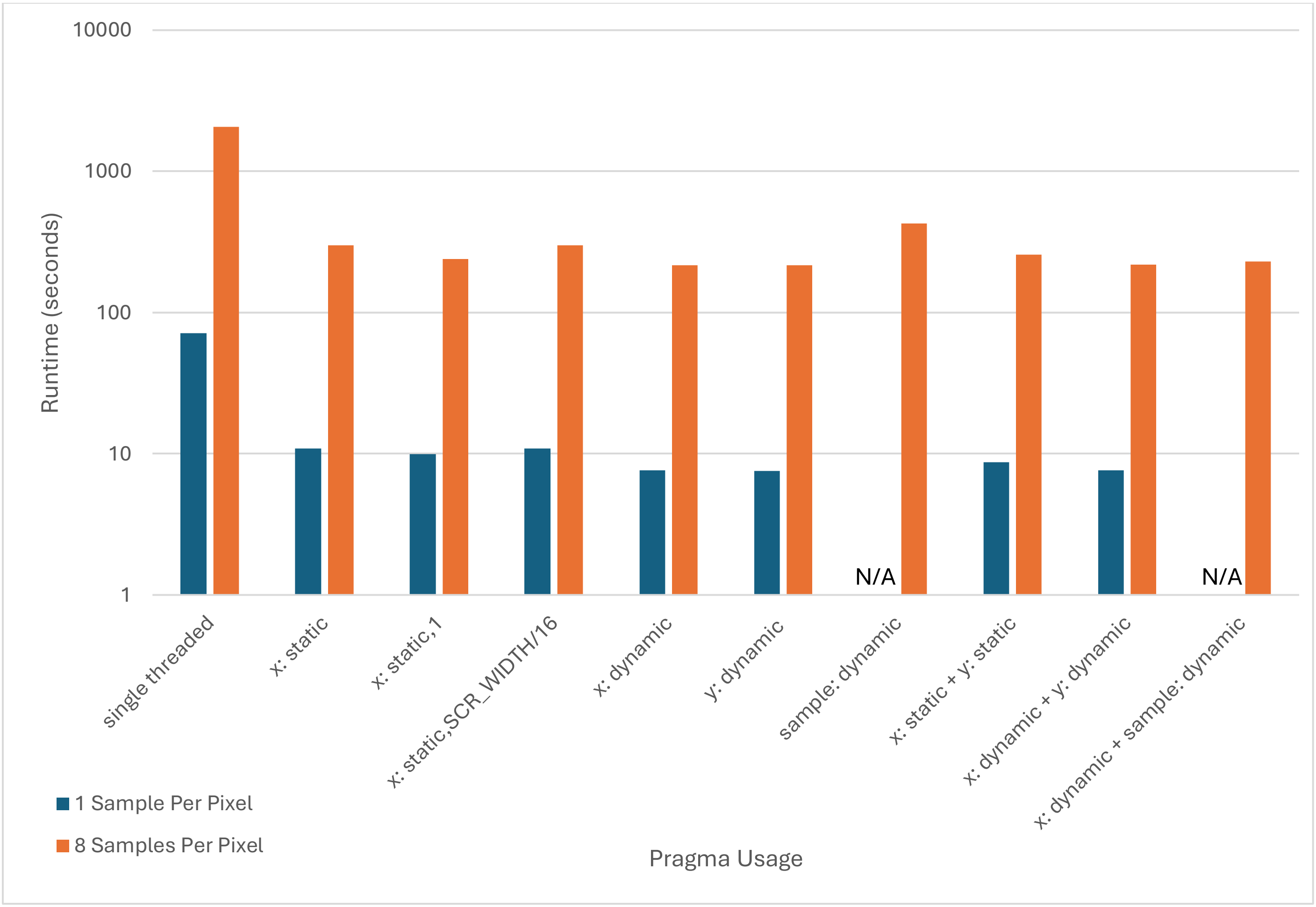

Figure 5.

The graph above shows the different types of OpenMP pragmas used and the runtimes for both 1 and 8 samples per pixel. Overall, there was little variance when comparing types of pragmas used among 1 and 8 samples per pixel. This shows that increasing the number of samples does not improve the performance of any type of pragma when comparing with the others. This may not be true with increasing the samples even more, but time did not allow for more testing.

Predictably, the single threaded test performed the worst as there is no parallelization used. When comparing the performance of static scheduling with dynamic scheduling, dynamic scheduling shows a considerable advantage. This is possibly due to the fact that each pixel can have varying workloads depending on the depth of recursion. Static scheduling is best for well known work distribution, which would not be ideal when certain threads could finish much quicker with pixels that return the background early. OpenMP dynamic scheduling will adapt to varying workloads per thread.

Using pragmas for the x and y loop fundamentally changes the way pixels are assigned to threads, but this did not seem to change the runtimes. This shows that there is enough work to keep all threads busy throughout the pragmas iteration no matter which method was used. In fact, these two methods had the best performance out of all the tests.

Pragmas for the sample loop are not possible for the single sample tests, as there is no need for parallelizing a loop that only executes once. The 8 sample tests did not compete with the x and y loop as it took about twice as long. This is because only half of the threads in the CPU were being used for each pixel iteration. If the number of samples were to increase to 16 or more then it is predictable that performance would become competitive with the other pragmas.

Combining x and y loop pragmas was comparable to just x or just y loop pragmas. And combining x and sample loop pragmas closer than only the sample loop pragmas to the best performing pragmas.

Conclusion:

More tests could be performed on higher maximum recursion depths and higher sample rates. This may show differences between x and y loop pragmas or increase comparable performance of the sample loop pragma. It would also be beneficial to explore the use of CUDA programming to use even more threads at once. Due to time constraints this was not possible. All of this would be important for future work.

Overall, OpenMP made a dramatic improvement to the performance of the ray tracing program. The best pragma methods were to use dynamically scheduled pragmas around the x and y loops. In the end, the best pragma is one that keeps all threads busy throughout the program. The program is most efficient when no threads are left hanging.