Circle Renderer Project Report

Mattia Marilli

Abstract

This work explores the rendering of overlapping circles in a 2D image, considering their attributes such as position, size, color, transparency, and layering depth. We present a sequential rendering version and analyze its limitations in terms of performance and scalability. Than we implement a parallelized approach using OpenMP, incorporating spatial partitioning to optimize rendering efficiency.

1. Introduction

This project aims to develop a rendering system in C++ to display overlapping circles on a two-dimensional image. Each circle has unique properties, such as position, size, color, transparency, and depth, where depth is represented by a z-coordinate. To achieve a realistic visual effect, circles closer to the viewer need to be displayed on top of those further away, simulating a correct depth perspective.

The importance of maintaining accurate visual layering can be illustrated with an example using three circles of different colors: red, green, and blue. Suppose the circles are placed at varying depths along the z-axis, with the red circle having the highest z-coordinate, followed by the green circle, and the blue circle having the lowest z-coordinate. In this case, the red circle, being closest to the viewer, should be rendered on top, followed by the green circle, and finally, the blue circle at the bottom. If the circles are rendered without considering their depth, the result will be incorrect overlaps, with the blue circle potentially covering the red one, even though the red circle should be in front of all others.

Figure 1 demonstrates the difference in visual clarity when depth is properly maintained. On the left, circles are correctly ordered based on their depth, resulting in a clear and coherent image, where the circles' layering respects their relative positions in space. On the right, rendering without depth ordering causes random overlaps, losing the intended depth effect.

To further enhance the efficiency of this rendering process, two versions of the rendering system will be implemented: a sequential version and a parallel version using OpenMP directives. By leveraging parallel processing, this project will explore how OpenMP can improve rendering speed and efficiency, while ensuring that both versions produce the same visual output.

Figure 1. Comparison of rendering with correct (left) and incorrect (right) depth perception. The correct rendering order places circles with higher z-values (R) in front, followed by green and blue circles, while the incorrect rendering order fails to account for depth.

This report will cover the steps taken to achieve accurate visual layering and assess the performance impact of parallelization using OpenMP.

2. Objectives

The objectives of this project are as follows:

- Develop a rendering system in C++ to display overlapping circles with accurate depth-based layering.
- Implement a sequential version of the rendering system to establish a baseline for correctness and performance.
- Implement a parallel version using OpenMP to improve rendering speed while maintaining the same output as the sequential version.
- Evaluate and compare with different kinds of metrics the performance of both versions to understand the impact of parallelization.

3. Implementation

3.1. Sequential Implementation

The program consists of several key parts:

- **Circle Structure**:
  - Defines the attributes of each circle, including its position (`x`, `y`, `z`), size (`radius`), and color (`r`, `g`, `b`, `a`).
  - This structure holds all the information necessary to render the circles in 3D space and apply color with transparency.
- **Comparator Function (`compareByZ`)**:
  - Used to sort the circles based on their `z`-depth values.
  - Circles closer to the viewer are drawn last, thus appearing on top of those farther away.
- **Helper Functions (`isPointInCircle`)**:
  - Checks if a point lies within a circle's radius.
  - This function is essential for determining which pixels belong to which circles when rendering the image.
- **Rendering Function (`renderCircles`)**:
  1. **Image Initialization**: Initializes a blank `width` x `height` image with a white background (RGB = 255, 255, 255).
  2. **Sorting Circles by Depth**: Sorts circles based on the `z` value, so closer circles render on top of those farther away.
  3. **Rendering**: Loops through each pixel in the image:
     - **Circle-Pixel Check**: Calls `isPointInCircle` to determine if a given pixel lies within the circle's radius.
     - **Color Blending**: Uses the circle's alpha transparency to blend the circle's color with the background color of each pixel. This blending calculation considers each color channel (RGB) individually, producing smooth transitions.
- **Image Saving (`saveImageToFile`)**
  - The rendered image is saved in the **PPM (Portable Pixmap)** format.
  - The function writes the image file pixel by pixel.
- **Random Circle Generation (`generateRandomCircles`)**:
  - Creates a specified number of circles with randomized attributes (circle's center, depth (`z`), radius, color, and alpha transparency using uniform distributions).

3.1.1 Naive Rendering vs Grid-Based Rendering

The original implementation of the rendering process employed a *naive rendering* approach. In this method, the distance from each circle to every pixel in the image was calculated to determine whether the pixel was inside any of the circles. While straightforward, this approach suffered from inefficiencies due to redundant calculations: many circles did not influence certain pixels, yet were still unnecessarily checked. This inefficiency became increasingly problematic as the number of circles and pixels grew.

To address these limitations, a *grid-based rendering* approach was introduced. In this method, the image is divided into smaller cells forming a grid. Each cell contains a list of circles that intersect it, and only these circles are considered for the pixels within the cell. This optimization led to significant performance improvements, as outlined below:

- **Reduction of unnecessary checks:** Unlike the naive approach, where every circle is checked for every pixel, the grid-based method limits the checks to circles that are relevant to a specific grid cell. This significantly reduces the computational overhead.
- **Improved memory locality:** Circles located near each other in the image are processed together because they belong to the same or adjacent grid cells. This enhances cache efficiency and reduces memory latency during computations.
- **Optimized performance:** By narrowing the scope of circles considered for each pixel, the grid-based rendering approach is much faster, particularly when the number of circles is large. This makes it a more scalable solution.

N.B. The size of the grid is dynamically calculated based on the dimensions of the image. Specifically, the grid size is set as a fraction of the smaller dimension (either width or height) of the image. This calculation ensures that the grid size is proportionate to the size of the image, which helps balance the granularity of the grid cells and the computational efficiency.

In summary, while the naive rendering approach is simple and straightforward, it becomes computationally expensive as the problem size grows. The grid-based rendering method mitigates these inefficiencies by leveraging spatial partitioning, resulting in a significant reduction in computational complexity and improved rendering performance.

3.2. Parallel Implementation

To further improve performance, parallelization was introduced using OpenMP. The part of the program that iterated over all pixels of the image was parallelized to execute multiple computations simultaneously on separate threads. This was achieved using the OpenMP directive `pragma omp parallel for`.

In particular:

- The directive `pragma omp parallel for collapse(2)` was used to parallelize the nested loops that iterated over the image's height and width. The `collapse(2)` clause merges the two loops (height and width) into a single loop, improving load balancing and better utilizing multiple cores.
- The condition `if(numThreads > 1)` ensures that parallelization occurs only when more than one thread is available, avoiding unnecessary overhead when running on a single thread.

The inner loop calculates the distance from each pixel to the relevant circles and handles color blending if the pixel is within a circle. Since the operations are sequential for each pixel,and each pixel is indipendent by others, no thread synchronization is needed. Each thread processes its own pixel without shared data. This sequential approach ensures accurate blending, proper handling of overlapping circles and transparency, and consistent visual results without the need for complex synchronization, guaranteeing both efficiency and accurate rendering.

3.2.1 Randomization and File Writing Not Parallelized

The randomization of circles and the file writing processes were not parallelized for specific reasons:

- **Randomization of Circles**: the randomization of circles is performed once before the rendering process and is independent of it.
- **File Writing**: writing to a file inherently involves shared resources (e.g., the file handle), requiring thread synchronization to avoid data corruption or race conditions. This synchronization would introduce significant overhead, negating the benefits of parallelization. Parallelizing this process would complicate the implementation without yielding meaningful performance gains.

3.2.2 Advantages of Parallelization

Parallelization using OpenMP, combined with the grid approach, provided several benefits:

- Parallelization with OpenMP divides the task of processing the pixels across multiple threads, significantly reducing rendering times on multi-core systems.
- The combination of grid optimization and OpenMP parallelization makes the program scalable and efficient, even with a high number of circles and pixels.

In summary, the introduction of the grid approach improved the efficiency of the sequential implementation by reducing the number of circles checked for each pixel. When combined with OpenMP parallelization for the outer loops, the rendering process was significantly accelerated, resulting in improved performance on multi-core systems. By keeping the inner loop sequential, we ensured that the program remained efficient without introducing unnecessary overhead for thread synchronization.

4. Experimentation and Results

4.1. Test Setup

In this experiment, we used the parallelized version of the program to assess the impact of parallelization on the rendering performance. To establish a baseline, we performed tests using a single thread, where OpenMP was not activated. This allowed us to compare the performance of the parallelized version with that of the sequential (single-threaded) version.

The tests were executed on a system with an Apple M1 chip, which features a total of 8 cores (4 high-performance cores and 4 high-efficiency cores) with no hyper-threading. The number of threads was varied in the parallel version to evaluate its scalability and performance improvements across different configurations. This is an important aspect as the heterogeneous architecture of the M1 chip—combining high-performance and high-efficiency cores—impacts the test results and highlights how the program leverages these resources. The effects of this architecture will be visible in the tests, particularly in scenarios with varying thread counts.

The images used for testing have a fixed size of 2000x2000 pixels, and the number of circles rendered on the image varies between 10 and over 100000. This range allows us to observe how performance scales as the workload increases. Specifically, we compared the following configurations:

- **Single-threaded execution**: Only one thread was used, which does not trigger OpenMP parallelization. This setup serves as the baseline.
- **Multi-threaded execution**: The number of threads was varied (2, 4, 8, 16), and OpenMP parallelization was used to distribute the workload across multiple

cores. These tests aimed to measure the speedup and efficiency as the number of threads increased.

For each configuration, we measured the rendering duration, i.e., the total time taken to process the image with a given number of circles, excluding the file writing step, as it is a sequential operation. The execution time was measured multiple times for each configuration to account for variability in performance. The results were then averaged to ensure more reliable and consistent data. The rendering time was used to compute performance metrics such as speedup and efficiency.

4.2. Performance Results

The performance results of the parallelized rendering implementation are presented in Table 1, which details the rendering duration, speedup, and efficiency for different numbers of circles and varying thread counts. The graph in Figure 2 illustrates the speedup achieved with increasing numbers of threads for different circle counts. It shows the relationship between the number of threads (from 1 to 16) and the corresponding speedup for various numbers of circles (10, 100, 1000, 10000, 100000).

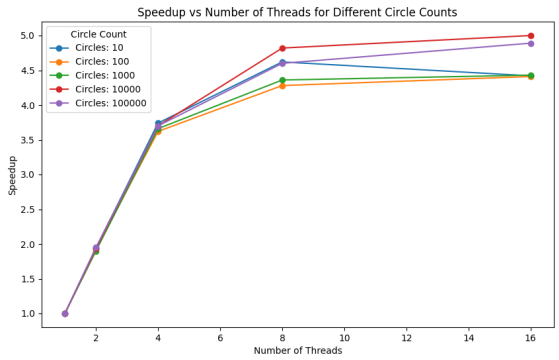

Figure 2. Speedup vs. Number of Threads for Different Numbers of Circles.

From the data and the graph, we can observe the following:

- As the number of threads increases, the rendering time generally decreases, indicating that parallelization improves performance.
- Speedup is most significant when moving from 1 to 4 threads, and it continues to improve, although at a slower rate, as more threads are added.
- Efficiency decreases as the number of threads increases, especially when the number of threads exceeds the available cores, reflecting diminishing returns on parallelization.

Circles	Threads	Time (s)	Speedup	Efficiency
10	1	0.13	1.00	1.00
10	2	0.06	1.94	0.97
10	4	0.03	3.74	0.94
10	8	0.02	4.62	0.58
10	16	0.02	4.42	0.28
100	1	0.15	1.00	1.00
100	2	0.07	1.93	0.96
100	4	0.04	3.62	0.91
100	8	0.03	4.28	0.54
100	16	0.03	4.41	0.28
1000	1	0.35	1.00	1.00
1000	2	0.18	1.90	0.95
1000	4	0.09	3.66	0.91
1000	8	0.08	4.36	0.54
1000	16	0.07	4.43	0.28
10k	1	2.43	1.00	1.00
10k	2	1.26	1.93	0.96
10k	4	0.65	3.70	0.92
10k	8	0.50	4.82	0.60
10k	16	0.48	5.00	0.31
100k	1	22.93	1.00	1.00
100k	2	11.78	1.95	0.97
100k	4	6.19	3.70	0.93
100k	8	4.99	4.60	0.57
100k	16	4.68	4.89	0.31

Table 1. Performance results for varying numbers of circles and threads.

- Between 4 and 8 threads, speedup continues to increase but at a slower rate. This is due to the fact that the Apple M1 chip includes 4 high-performance cores and 4 high-efficiency cores. The latter are less performant, and their involvement begins to reduce the overall speedup gain.
- For 16 threads, the performance remains comparable to 8 threads. This is because the M1 chip does not support hyper-threading, meaning that additional threads do not lead to further performance improvements. In fact, the operating system efficiently manages the threads, ensuring that performance degradation does not occur when using more threads than the available cores.
- The performance improvements are notable but tend to level off after a certain point, demonstrating the trade-off between the number of threads and the system's capacity to handle them efficiently.
- We can also observe that, especially with 8 and 16 threads, the best speedup (even if not by much) occurs when rendering a high number of circles. With fewer circles, the overhead of thread creation likely

outweighs the benefits of parallelization, limiting the speedup.

In summary, the OpenMP-based parallelization provides substantial performance improvements for both small and large workloads. While scalability plateaus at higher thread counts, the approach remains effective across various circle counts, ensuring an optimal balance of speedup and efficiency.

5. Conclusions

In conclusion, the project developed a system for rendering overlapping circles, addressing both depth management and rendering efficiency. The sequential implementation ensured accurate visualization, while the grid-based rendering optimization improved performance. The introduction of parallelization using OpenMP further reduced rendering times, particularly on multi-core systems. The results showed consistent performance improvements, even as the complexity of the rendered scenes increased. This approach demonstrated how optimization and parallelization can greatly enhance graphical rendering systems without causing performance degradation.