## **Comparative Analysis of Serial and Parallel Rendering Techniques for Semi-Transparent Circles**

**Riccardo Marconi**  
E-mail: [riccardo.marconi@edu.unifi.it](mailto:riccardo.marconi@edu.unifi.it)

Mat: 7151651

### **Abstract**

This study analyzes the performance of serial versus parallel algorithms for rendering semi-transparent circles on an 800x600 2D canvas. Each circle is characterized by center coordinates, radius, and RGBA color. This report incorporates performance data from both implementations to evaluate their efficiency.

### **1. Introduction**

To create graphics, computer graphics often involve rendering 2D objects on the screen whereby rendering semi-opaque shapes like circles is a problem that can be used in games, simulations, and data visualization. The transparency of shapes introduces another level of challenges to renderability. This is due to the fact of handling the transition through opaque boundaries and adjusting the color of the shapes depending on the degree of opacity. This report discusses and analyzes the implementation of serial and parallel rendering techniques for shapes such as circles having transparency effects.

In the serial approach, each circle is processed one by one. For each circle, the algorithm iterates over a square bounding box defined by the circle's radius. Every pixel within that box is checked to determine if it lies inside the circle using the circle equation. If the pixel is inside the circle, its color is updated using alpha blending, where the circle’s color (weighted by its alpha value) is combined with the pixel’s existing color. Although this method is straightforward, computing the affected area based on the circle’s radius can be quite time-consuming, especially when a large number of circles or high-resolution images are involved.

The second broad strategy, known as the parallel strategy, leverages the GPU's many cores via CUDA to significantly reduce execution time. In this approach, each circle is processed concurrently by a separate GPU thread, which iterates over its bounding box and performs alpha blending with the existing pixel colors. This method enables rapid rendering even for large numbers of circles, although it places a heavier workload on the graphics card and its memory bandwidth.

As detailed and described in this report, it presents the implementation aspects of the serial as well as the parallel algorithms, together with the performance analysis. Such time and graph comparison have been set to reveal the benefits and possible shortcomings of using parallel computing to render transparent shapes. It is hoped that the contents of this investigation will prove useful when considering state-of-the-art rendering methodologies in parallel with the strength of parallelism in graphical applications.

### **2. Problem Statement**

The handling of semi-transparent shapes poses significant challenges in computer graphics. As the number of shapes increases and the canvas size grows, the computational load can become critical—impacting performance in video games, simulations, and data visualizations. One key difficulty is correctly blending overlapping shapes: when one shape partially covers another, the final color is determined by alpha blending based solely on the drawing order rather than an explicit Z-value. In conventional serial algorithms, each shape is processed individually, resulting in high computation times that may not be feasible for real-time applications.

To address this, we require a method to take advantage of enhanced multi-core processors in rendering in parallel and thus minimize the time while preserving a vast resemblance with the first frame’s high-quality picture. The main objectives of this work include the creation of algorithms for the elements of serial and parallel rendering and accuracy**.**

### **3. Algorithmic Approach**

#### **3.1 Serial Algorithm**

The serial implementation follows an algorithmic method to semi-transparent circle generation:

**Initialization:** The canvas is allocated as an array of pixels in RGBA format and is initialized to zero, resulting in a completely transparent black background.

**Circle Properties and Ordering:** Each circle is defined by its center coordinates (x, y), its radius, and its RGBA color components. In this implementation, no depth-based ordering (using a z-coordinate) is applied; the circles are rendered sequentially in the order they are generated.

**Pixel Iteration:** For each circle, the algorithm iterates only over the pixels within its bounding box, thereby limiting the number of pixels processed and improving efficiency instead of iterating over every pixel of the entire canvas.

**Test:** Within the bounding box, the algorithm determines if a pixel lies inside the circle by comparing the squared distance from the circle's center to the squared radius. This method avoids the need to compute the square root.

**Color Update:** The key steps of the serial algorithm are implemented in the function renderCirclesSequential(). Each circle is processed individually, with the inner loop iterating solely over the pixels within its bounding box. Consequently, the time complexity is approximately O(m⋅c), where m is the number of circles and c is the average number of pixels in a circle's bounding box, while the space complexity is O(n), with n representing the total number of pixels in the canvas.

#### **3.2 Parallel Algorithm with CUDA**

The parallel implementation uses CUDA to enhance performance by assigning each thread to process an individual circle. Within each thread, the circle's bounding box is iterated over to update the pixel colors accordingly. This approach allows for the simultaneous rendering of multiple circles, significantly improving performance through effective thread configuration and memory management.

### **4. Performance Evaluation**

#### **4.1 Performance Data Summary**

Parallel Rendering Results:

|  |  |  |
| --- | --- | --- |
| **Threads** | **Circles** | **Time (seconds)** |
| 256 | 10 | 0.0011997 |
| 512 | 10 | 0.000999256 |
| 1024 | 10 | 0.000999453 |
| 256 | 100 | 0.0017571 |
| 512 | 100 | 0.00177516 |
| 1024 | 100 | 0.00175615 |
| 256 | 1000 | 0.0032944 |
| 512 | 1000 | 0.00626617 |
| 1024 | 1000 | 0.0118584 |
| 256 | 10000 | 0.0116749 |
| 512 | 10000 | 0.0112669 |
| 1024 | 10000 | 0.0124591 |
| 256 | 100000 | 0.138711 |
| 512 | 100000 | 0.0988683 |
| 1024 | 100000 | 0.0979236 |
| 256 | 1000000 | 0.967553 |
| 512 | 1000000 | 0.972618 |
| 1024 | 1000000 | 0.969739 |

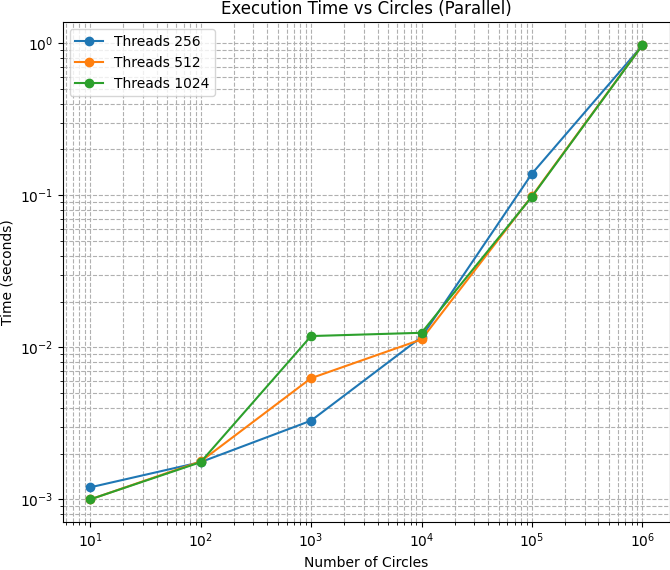
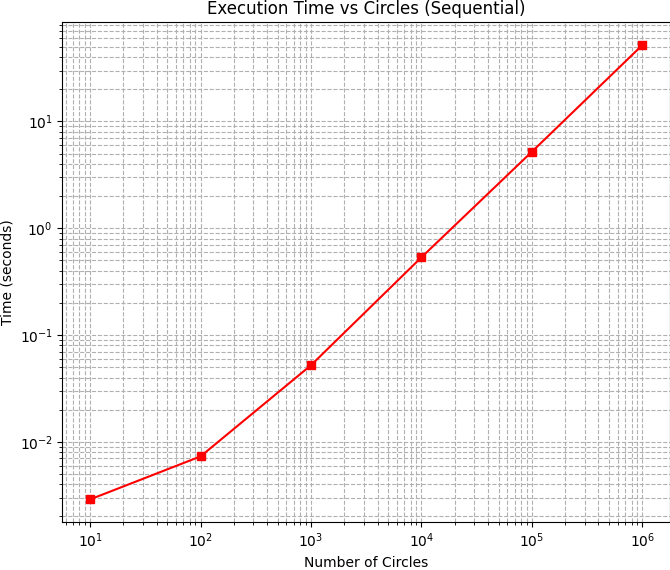
Serial Rendering Results:

|  |  |
| --- | --- |
| **Circles** | **Time (seconds)** |
| 10 | 0.0028892 |
| 100 | 0.0073041 |
| 1000 | 0.0521712 |
| 10000 | 0.53489 |
| 100000 | 5.21369 |
| 1000000 | 51.9802 |

### **4.2 Analysis of Results**

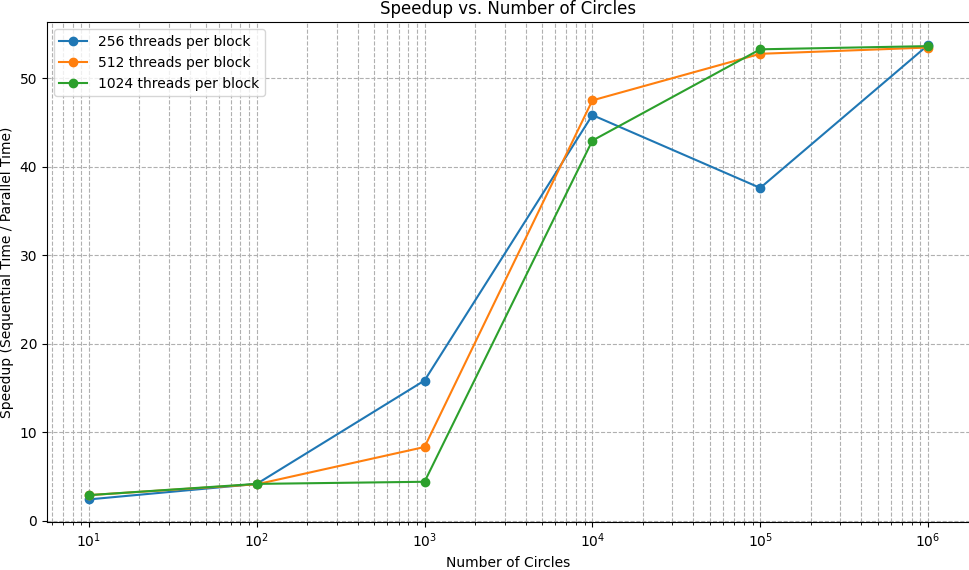


**Parallel Performance:** The parallel implementation demonstrates a significant reduction in rendering time as the number of circles increases. For example, rendering 1,000,000 circles with 1024 threads took approximately 0.970 seconds, while the same task in the serial implementation took over 50 seconds. This illustrates the substantial efficiency gains achieved through parallel processing.



**Efficiency of Small Circle Counts**: For smaller numbers of circles (10, 100, and 1000), the rendering times are minimal in both implementations, but the parallel approach still shows a clear advantage, especially with higher thread counts.

**Speedup:** The speedup rapidly increases as the problem size grows, highlighting the benefits of GPU parallelization for large workloads. The graph also shows that as the number of circles increases, the speedup converges to a similar value across all thread configurations.



### **5. Conclusion**

The findings confirm the effectiveness of using CUDA for rendering semi-transparent circles. The parallel algorithm consistently outperformed the serial implementation, especially as the complexity of the rendering task increased. The analysis highlights the relationship between thread count and rendering performance, underscoring the potential for further optimizations in parallel processing.