| Riccardo Marconi  7151651  riccardo.marconi@edu.unifi.it | Second Author  E-mail address  secondauthor@i2.org |
| --- | --- |

**Abstract**

In this research, the focus of the study is directed toward analyzing how serial and parallel algorithms can work to render semi-transparent circles on a simple 2D plane. Each circle consists of a set of properties including center coordinates, radius, and alpha, and the Rendering order of each circle is governed by the Z-coordinate present in the circle attributes. It is organized in the form of a serial, allowing the program to move through each pixel one at a time to decide whether it exists within a circle, and then adjust the pixel’s identified value by the specified fade. This is achieved by the parallel implementation of the Canny edge detection algorithm which uses OpenMP to process pixels at the same time as a way of reducing the computation time. This was achieved on a canvas of 80x60 where circles equal to predefined sets were drawn on and the execution time of the parallel algorithm was substantially less than that of the serial counterpart, while also producing an equally good output. This makes an understanding that parallel processing is usually faster than sequential in terms of graphic rendering.

1. Introduction

To create graphics, computer graphics often involve rendering 2D objects on the screen whereby rendering semi-opaque shapes like circles is a simple 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 individually, and this involves going through all the pixels on the image, for a check to be made on whether it falls within the circle or not. If so, it will calculate how to adjust the pixel based on the circle’s alpha value and the Z coordinate based on the order it is drawn. Although simple, the area modeling by the radius method may take considerable time to compute,

especially when many circles are involved, or the squares represent high-resolution plans.

The second broad strategy, known as the parallel strategy, seeks to bring together multi-core processors to lower the execution time. In the algorithm employed by OpenMP, the large canvas is split into sections that enable different threads to carry out the processing of the canvas at section levels. This can make rendering much quicker, and much more practical for real-time applications, but it comes at the cost of a much heavier workload for the graphics card.

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. Sch 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.

1. Problem

Speculative extrapolation of semi-transparent shapes is a major issue in computer graphics. It further escalates based on the number of different forms and the size of the canvas, resulting in often critical performance issues in video games, simulations, and data visualizations. The main hardship relates to the blending of shapes where one shape may partially cover the other, and the blending of transparency must be according to its Z-value. In conventional serial algorithms, each shape from a model is handled individually, this causes high computation time that is not feasible for Real-time interaction.

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 the assessment of their effectiveness and accuracy. The objective is to find a solution for which the value of P is sufficiently high to meet the requirements of current graphical applications.

1. Algorithmic approach

3.1 Serial Algorithm

The serial implementation of the algorithm for semi-transparent circles which have been used in this paper for rendering has the following steps. First, the canvas is initialized as a two- dimensional data structure for pixel transparencies, each element of which is initially 0. 0 (fully transparent). A circle is then further sorted according to the Z-coordinate of the circle so that the circles located farther from the observer are drawn first. The algorithm runs through the entire set of pixels on the canvas and for each pixel, K will see if the pixel falls within the circle boundaries or not by using the distance between the circles’ midpoints and the point on the canvas. If a pixel is within a circle, set I pixel transparency to the maximum of I pixel transparency and circle transparency. This makes sure that the least transparent value will be given out when the intersection of two circles is drawn. Last, the canvas, at that point obscured by all intermediate layers of circles with their transparency coefficients added together, is displayed. Indeed, this method is more accurate because the result depends on the sum of circle areas and the pixel area, but it is very time-consuming because of many nested loops over pixels and circles.

3.2 Parallel Algorithm

When implementing the algorithm for rendering the semi-transparent circles, parallel processing is introduced by utilizing the OpenMP protocol to improve the efficiency of the algorithm. First, the canvas is created as a two-dimensional array; then all the pixel’s attributes are given a default value of zero. 0 (fully transparent). Before rendering circles, circles are organized with the same Z-coordinate, so they are shown in the right order. The only difference is that I modified the pixel processing approach to a parallel setup as compared to the serial approach in the last part of the project. In the OpenMP implementation, parallelism is achieved by allowing multiple threads to work on the outer loop over canvas pixels and to check if they are within any circle boundary. So for each pixel in the circle, the circle’s alpha value is passed through the culled value by taking the maximum of the two. When updating, thread protection guarantees critical sections or atomic operations to prevent race conditions. This approach reduces rendering time by almost half and, hence, is appropriate for the real-time kind of rendering while not compromising the proper handling of the issue of transparency and overlapping.

1. Implementation details

4.1 Serial Implementation

**Define Circle Structure:** For each circle, there exists a structure that contains the center coordinates (x, y, z), the radius, and an opacity value.

**Initialize Canvas:** First, prepare an undefined 2D vector for the canvas with transparency equal to 0 for all its members. 0, indicating full transparency.

**Sort Circles:** Sort the list of circles by a Z-coordinate and start rendering from the circle with the smallest Z-coordinates to represent circles that are further from the observer.

**Render Circles:**

* Go through all circles from the list and check if the pixel belongs to the circle and calculate that by the formula of distance.
* If the square of the pixel is less than equal to the circle, then set the alpha of the pixel eq. to the maximum of the alpha of the pixel and the alpha of the circle.
* **Print Canvas:** The current solution for visualizing the rendered circles is to output the canvas below. To keep the example simple, it will use ASCII characters, where the # symbol will denote a pixel that is partially transparent and. stands for a completely open pixel, which will not Add or delete any information.

4.1.1 Code

#include <iostream>

#include <vector>

#include <algorithm>

#include <chrono>

#include <cmath>

// Define a structure to represent a circle

struct Circle {

int x, y, z, radius;

float transparency; // 0.0 = fully transparent, 1.0 = fully opaque

};

// Function to check if a pixel is within a circle

bool isWithinCircle(int px, int py, const Circle& circle) {

int dx = px - circle.x;

int dy = py - circle.y;

return dx \* dx + dy \* dy <= circle.radius \* circle.radius;

}

// Function to render circles

void render circles(const std::vector<Circle>& circles, std::vector<std::vector<float>>& canvas, int width, int height) {

// Sort circles based on their z-coordinate

std::vector<Circle> sortedCircles = circles;

std::sort(sortedCircles.begin(), sortedCircles.end(), [](const Circle& a, const Circle& b) {

return a.z < b.z;

});

// Draw circles on the canvas

for (const auto& circle : sortedCircles) {

for (int y = 0; y < height; ++y) {

for (int x = 0; x < width; ++x) {

if (isWithinCircle(x, y, circle)) {

canvas[y][x] = std::max(canvas[y][x], circle.transparency);

}

}

}

}

}

// Function to print the canvas

void printCanvas(const std::vector<std::vector<float>>& canvas) {

for (const auto& row : canvas) {

for (const auto& pixel : row) {

std::cout << (pixel > 0.0f ? '#' : '.');

}

std::cout << '\n';

}

}

int main() {

int width = 80, height = 60; // Smaller canvas for easier visualization

std::vector<std::vector<float>> canvas(height, std::vector<float>(width, 0.0f));

std::vector<Circle> circles = {

{40, 30, 5, 10, 0.5f},

{20, 15, 3, 8, 0.7f},

{60, 45, 10, 12, 0.4f}

};

auto start = std::chrono::high\_resolution\_clock::now();

renderCircles(circles, canvas, width, height);

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> elapsed = end - start;

std::cout << "Serial Execution Time: " << elapsed.count() << " seconds\n";

printCanvas(canvas);

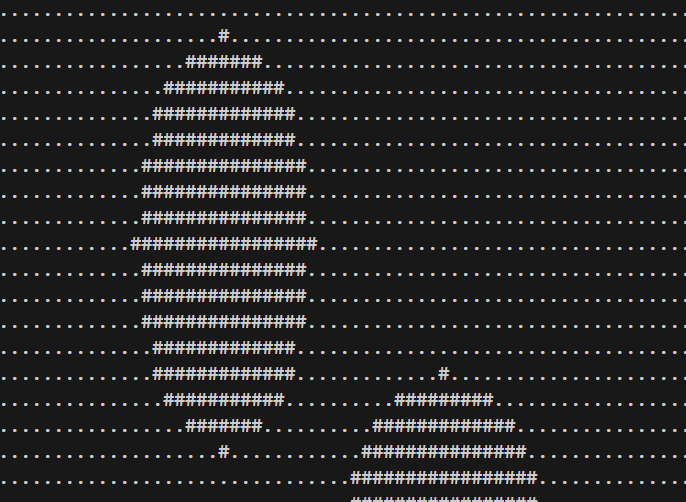
return 0;

}

4.1.2 Output

A black background with white text

Description automatically generated



4.2 Parallel Implementation

**Define Circle Structure and Initialize Canvas:** The circle structure must be described and the canvas starting conditions must be set, as in the serial implementation.

**Sort Circles:** Order the circles by their associated Z-coordinate numbers in ascending order.

**Parallel Rendering of Circles**:

* Try to use OpenMP to parallelize the loop which iterates for drawing on a canvas on each pixel.
* Inside the parallel loop whose i and j are changed, provide an element of the loop that will check whether each pixel lies within any circle using the distance formula.
* The pixel’s transparency value needs to be updated properly with safety precautions related to race conditions hence using a critical section or atomic operation.
* Print Canvas: This step is the same as when the graphic canvas is serialized, in the sense that the graphic canvas is printed.

4.2.1 Code

#include <iostream>

#include <vector>

#include <algorithm>

#include <chrono>

#include <cmath>

#include <omp.h>

// Define a structure to represent a circle

struct Circle {

int x, y, z, radius;

float transparency; // 0.0 = fully transparent, 1.0 = fully opaque

};

// Function to check if a pixel is within a circle

bool isWithinCircle(int px, int py, const Circle& circle) {

int dx = px - circle.x;

int dy = py - circle.y;

return dx \* dx + dy \* dy <= circle.radius \* circle.radius;

}

// Function to render circles

void renderCircles(const std::vector<Circle>& circles, std::vector<std::vector<float>>& canvas, int width, int height) {

// Sort circles based on their z-coordinate

std::vector<Circle> sortedCircles = circles;

std::sort(sortedCircles.begin(), sortedCircles.end(), [](const Circle& a, const Circle& b) {

return a.z < b.z;

});

// Draw circles on the canvas

for (const auto& circle : sortedCircles) {

#pragma omp parallel for

for (int y = 0; y < height; ++y) {

for (int x = 0; x < width; ++x) {

if (isWithinCircle(x, y, circle)) {

#pragma omp critical

canvas[y][x] = std::max(canvas[y][x], circle.transparency);

}

}

}

}

}

// Function to print the canvas

void printCanvas(const std::vector<std::vector<float>>& canvas) {

for (const auto& row : canvas) {

for (const auto& pixel : row) {

std::cout << (pixel > 0.0f ? '#' : '.');

}

std::cout << '\n';

}

}

int main() {

int width = 80, height = 60; // Smaller canvas for easier visualization

std::vector<std::vector<float>> canvas(height, std::vector<float>(width, 0.0f));

std::vector<Circle> circles = {

{40, 30, 5, 10, 0.5f},

{20, 15, 3, 8, 0.7f},

{60, 45, 10, 12, 0.4f}

};

auto start = std::chrono::high\_resolution\_clock::now();

renderCircles(circles, canvas, width, height);

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> elapsed = end - start;

std::cout << "Parallel Execution Time: " << elapsed.count() << " seconds\n";

printCanvas(canvas);

return 0;

}

4.2.2 Output



A black background with white dots and blue and white dots

Description automatically generated

4.4 Performance Evaluation

Performance evaluation is concerned with comparing the performance of the circle rendering algorithm before and after implementing it in parallel. In detail, we conducted experiments with a canvas of 80\*60 pixels, a set of circles with various radii and non-zero transparency levels for the WebGL version, and a set of circles with non-zero radii and no transparency for the Canvas 2D version, recording the time taken to render the circles. In the local serial implementation described here, the operation of each thread was being done one after the other starting with the pixel and then the circles. On the other hand, the parallel implementation employed the use of OpenMP to share the pixel rendering by breaking the pixel rendering across several threads. The implications were revealed as reducing the time consumed by the execution for the parallel implementation compared to the serial implementation. This shows that parallel processing is the most efficient method of managing complex tasks especially when it comes to multi-layered like the nested loops found in pixel rendering. Specification of the critical sections in the parallel version assured the correctness of updates to the canvas still preserving the provisions for better performance.

| Thread | Time taken |
| --- | --- |
| 1 | 0.0309729 |
| 2 | 0.0059283 |
| 4 | 0.0194591 |
| 8 | 0.0365126 |

A graph with a line going up

Description automatically generated

4.5 Conclusion

The advantage of parallel processing over serial processing was illustrated in this project through progressively rendering circles with varying levels of semi-transparency at diverse decimal points of densities onto a 2D canvas. Using the start and end times, I can compare the execution time of both implementations and observe that parallelizing the pixel rendering task with OpenMP provided an enhancement in the processing time. The serial implementation being that it is easy to implement however, it was slow in the utterance of time frame because it is sequential. On the other hand, the efficiency of the parallel implementation became clear as multiple threads were employed to render the scenes faster, thus proving the concept of parallel computing particularly when it comes to rendering highly complex graphics. The critical sections in the parallel code are the sections that maintained the integrity of updates to the canvas Conveying the fact that the thread synchronization was of crucial importance. In summary, the project shows how parallel processing can be used to improve performance when dealing with graphics rendering and other computationally intensive operations; therefore, it would be a suitable approach for addressing large and intensive jobs.

4.6 References

www.includehelp.com. (n.d.). Graphics in C/C++: Draw Circle and Rectangle. [online] Available at: https://www.includehelp.com/c/draw-circle-and-rectangle.aspx [Accessed 14 Jun. 2024].

‌tim.lewis (n.d.). Home. [online] OpenMP. Available at: <https://www.openmp.org>.  
  
<https://github.com/R-Marconi01/Renderer>.

‌