Georgia Institute of Technology
School of Electrical and Computer Engineering

ECE 4122/6122 – Homework 2 Assignment

2D Ray Tracing with SFML

Due: March 5, 11:59 PM EST

---

1. Objective

In this assignment, you will implement a 2D ray tracer that renders a scene of geometric shapes (circles and line-segment walls) using SFML for real-time visualization. You will implement the ray tracing computation using three approaches:

- Single-threaded (baseline)
- Multi-threaded using std::thread
- Multi-threaded using OpenMP

You will then compare the performance of all three approaches and analyze the results in a written report.

This assignment reinforces concepts in concurrent programming with std::thread and OpenMP, ray-geometry intersection algorithms, real-time graphics with SFML, and performance measurement and analysis.

2. Background

2D Ray Tracing Overview

In 2D ray tracing, rays are cast from a light source position outward at evenly spaced angles across 360°. Each ray travels in a straight line until it intersects with a scene object (a circle or a line-segment wall) or exits the scene boundaries. The closest intersection point along each ray determines where the ray "hits."

By casting many rays (e.g., 3,600 to 36,000+), you can simulate how light illuminates a 2D scene. The resulting visualization shows lit regions and shadows cast by the objects.

Ray-Circle Intersection

Given a ray with origin O and unit direction D, and a circle with center C and radius r, substitute the parametric ray equation P(t) = O + tD into the circle equation:

$$|P - C|^2 = r^2$$

This yields a quadratic in t:

$$a = D \cdot D \qquad (\text{always } 1 \text{ for unit } D)$$

$$b = 2 \cdot D \cdot (O - C)$$

$$c = (O - C) \cdot (O - C) - r^2$$

$$\text{discriminant} = b^2 - 4ac$$

If discriminant < 0 there is no intersection. If discriminant ≥ 0, then t = (−b ± √discriminant) / 2a; take the smallest positive t.

Ray-Line Segment Intersection

Given a ray (origin O, direction D) and a line segment from P1 to P2, use the parametric form:

$$\text{Ray:} \quad R(t) = O + t \cdot D \qquad (t \geq 0)$$

$$\text{Segment:} \quad S(u) = P1 + u \cdot (P2 - P1) \quad (0 \leq u \leq 1)$$

Solve the 2×2 linear system for t and u. If t ≥ 0 and 0 ≤ u ≤ 1, the intersection is valid at distance t.

3. Requirements

3.1 Scene Description

Your program shall render a scene containing:

- One movable point light source (controlled by the mouse)
- At least 2 circles of varying sizes and positions
- At least 4 line-segment walls (e.g., forming partial enclosures)
- Scene boundary walls (the edges of the window)

You may hard-code the scene geometry or load it from a simple configuration file (bonus points for file loading – see Section 6).

3.2 Ray Casting

- Cast N rays from the light source uniformly distributed over 360°, where N is configurable (default: 10,800 rays, i.e., one ray per 0.0333°).
- For each ray, find the nearest intersection with any scene object.
- Draw each ray as an sf::Vertex line from the light source to the hit point using a color gradient (e.g., bright yellow at the source fading to dark orange/transparent at the hit point).

3.3 Threading Implementations

You must implement the ray computation in three modes, switchable at runtime via keyboard input:

Key	Mode
1	Single-threaded (baseline)
2	std::thread with configurable thread count
3	OpenMP with configurable thread count

↑ / ↓	Increase / decrease thread count (2, 4, 8, 16, ...)

std::thread Implementation:

- Divide the N rays evenly among T threads.
- Each thread computes intersections for its assigned range of rays and writes results into a shared (pre-allocated) results array. Since each thread writes to a distinct index range, no mutex is needed for the results array.
- Use std::thread::hardware_concurrency() to determine a reasonable default thread count.

OpenMP Implementation:

- Use #pragma omp parallel for with schedule (static) (and optionally experiment with dynamic scheduling) to parallelize the ray loop.
- Set thread count via omp_set_num_threads(T).

3.4 SFML Visualization

- Window size: 1920 × 1080 (or current desktop resolution)
- Display the following on screen using sf::Text:
  - current mode (Single, std::thread, OpenMP),
  - number of threads (when applicable),
  - frame time in milliseconds (for the ray computation only, excluding rendering), number of rays, and FPS.
- The light source follows the mouse cursor position.
- Scene objects (circles, walls) should be drawn with visible outlines.
- Rays should be rendered as lines from source to intersection point with an alpha gradient.
- Press ESC to exit the application.
- Press + / − to increase/decrease the number of rays by 3,600.

3.5 Timing

Use std::chrono::high_resolution_clock to measure the wall-clock time of the ray-casting computation for each frame (do not include the SFML rendering time). Display a running average (over the last 60 frames) of the computation time.

4. Program Structure

You are free to organize your code, but the following structure is recommended:

```
project/
├── main.cpp              // Main loop, SFML window, input handling
├── RayTracer.h/.cpp      // Ray casting logic (single, std::thread, OpenMP)
├── Scene.h/.cpp          // Scene geometry (circles, walls)
├── Geometry.h/.cpp       // Intersection math utilities
├── Makefile
└── report.pdf
```

Makefile Requirements

Your Makefile must compile on the PACE-ICE cluster or a standard Ubuntu system with SFML installed. Required flags:

```
CXX = g++
CXXFLAGS = -std=c++17 -O2 -fopenmp -pthread
LDFLAGS = -lsfml-graphics -lsfml-window -lsfml-system -fopenmp -pthread
```

5. Report (30 points)

Submit a PDF report (2–4 pages) that includes:

5.1 Performance Analysis (20 points)

- Timing table showing average frame computation time (ms) for each mode. Vary number of rays (3,600 / 10,800 / 36,000 / 108,000) and number of threads (1, 2, 4, 8, 16) for std::thread and OpenMP.
- Speedup plot: For each ray count, plot speedup (T_1/T_n) vs. number of threads for both std::thread and OpenMP. Include the ideal linear speedup line for reference.
- Efficiency analysis: Compute parallel efficiency (Speedup / N_threads) and discuss where efficiency drops off and why.

5.2 Discussion (10 points)

- How does the performance of std::thread compare to OpenMP? Which is easier to implement? Which gives better performance for this workload, and why?
- At what number of rays does parallelization become worthwhile (i.e., where does the overhead of thread creation/management get amortized)?
- What is the effect of increasing thread count beyond the number of physical cores? Explain in terms of hardware threading and context switching.

6. Grading Rubric

Component	ECE 4122/6122
Correct single-threaded ray tracing with SFML visualization	20
Correct std::thread implementation	15
Correct OpenMP implementation	15
Real-time mouse-controlled light source	5
On-screen HUD (mode, threads, timing, FPS)	5
Keyboard controls working correctly	5
Code quality (structure, comments, naming)	5
Report – Performance analysis	20
Report – Discussion	10
Total	**100**

Bonus (up to 5 points)

- (+3) Reflections: when a ray hits a circle, spawn a reflection ray (up to 2 bounces), render in a different color
- (+2) Colored lighting: assign colors to rays based on which object they hit

7. Starter Code Snippets

Ray Structure

```
struct Ray {
    sf::Vector2f origin;
    sf::Vector2f direction; // unit vector
};

struct HitResult {
    bool hit = false;
    float distance = std::numeric_limits<float>::max();
    sf::Vector2f point;
};
```

Casting Rays – Single-Threaded Baseline

```
void castRaysSingleThreaded(const sf::Vector2f& lightPos,
                            int numRays,
                            const Scene& scene,
                            std::vector<HitResult>& results)
{
    results.resize(numRays);
    for (int i = 0; i < numRays; ++i) {
        float angle = (2.0f * M_PI * i) / numRays;
        Ray ray;
        ray.origin = lightPos;
        ray.direction = {std::cos(angle), std::sin(angle)};
        results[i] = scene.closestIntersection(ray);
    }
}
```

OpenMP Version

```
void castRaysOpenMP(const sf::Vector2f& lightPos,
                    int numRays,
                    const Scene& scene,
                    std::vector<HitResult>& results,
                    int numThreads)
{
    results.resize(numRays);
    omp_set_num_threads(numThreads);
    #pragma omp parallel for schedule(static)
    for (int i = 0; i < numRays; ++i) {
        float angle = (2.0f * M_PI * i) / numRays;
```

```
        Ray ray;
        ray.origin = lightPos;
        ray.direction = {std::cos(angle), std::sin(angle)};
        results[i] = scene.closestIntersection(ray);
    }
}
```

std::thread Version

```
void castRaysStdThread(const sf::Vector2f& lightPos,
                       int numRays,
                       const Scene& scene,
                       std::vector<HitResult>& results,
                       int numThreads)
{
    results.resize(numRays);
    std::vector<std::thread> threads;
    int chunkSize = numRays / numThreads;

    for (int t = 0; t < numThreads; ++t) {
        int start = t * chunkSize;
        int end = (t == numThreads - 1) ? numRays : start + chunkSize;
        threads.emplace_back([&, start, end]() {
            for (int i = start; i < end; ++i) {
                float angle = (2.0f * M_PI * i) / numRays;
                Ray ray;
                ray.origin = lightPos;
                ray.direction = {std::cos(angle), std::sin(angle)};
                results[i] = scene.closestIntersection(ray);
            }
        });
    }
    for (auto& th : threads)
        th.join();
}
```

Rendering Rays with SFML

```
sf::VertexArray lines(sf::Lines, 2 * numRays);
for (int i = 0; i < numRays; ++i) {
    lines[2 * i].position = lightPos;
    lines[2 * i].color = sf::Color(255, 200, 50, 180);  // bright at source

    lines[2 * i + 1].position = results[i].point;
    lines[2 * i + 1].color = sf::Color(255, 100, 0, 30); // dim at hit
}
window.draw(lines);
```

8. Hints and Tips

- **False sharing:** Since each thread writes to a distinct portion of the results array, false sharing on cache lines at chunk boundaries is possible but unlikely to be a major factor with large ray counts. For bonus insight, discuss this in your report.

- **Timing precision:** Wrap only the ray computation in timing calls, not the rendering.
- **Compiler optimization:** Use -O2 for all timing comparisons. Debug builds will produce misleading results.

9. Submission Checklist

- ☐ All .cpp and .h source files
- ☐ Makefile that compiles cleanly with make
- ☐ report.pdf with timing data, plots, and discussion
- ☐ Code compiles and runs on pace-ice with SFML 2.5+, g++ 9+
- ☐ All three modes (single, std::thread, OpenMP) are functional
- ☐ Keyboard and mouse controls work as specified

10. Academic Integrity

This is an individual assignment. You may discuss concepts with classmates but all code must be your own. Use of AI-generated code (ChatGPT, GitHub Copilot, etc.) is not permitted for the core ray tracing and threading implementations. You may use AI tools for understanding concepts, debugging assistance, or report writing. Violations will be reported to the Office of Student Integrity.

11. LATE POLICY

Element	Percentage Deduction	Details
Late Deduction Function	score - 0.5*H	H = number of hours (ceiling function) passed deadline

Appendix A: Coding Standards

Indentation:

When using *if/for/while* statements, make sure you indent 4 spaces for the content inside those. Also make sure that you use spaces to make the code more readable.
For example:

```
for (int i; i < 10; i++)
{
    j = j + i;
}
```

If you have nested statements, you should use multiple indentions. Each { should be on its own line (like the *for* loop) If you have *else* or *else if* statements after your *if* statement, they should be on their own line.

```
for (int i; i < 10; i++)
{
   if (i < 5)
   {
       counter++;
       k -= i;
   }
   else
   {
       k +=1;
   }
   j += i;
}
```

Camel Case:

This naming convention has the first letter of the variable be lower case, and the first letter in each new word be capitalized (e.g. firstSecondThird). This applies for functions and member functions as well! The main exception to this is class names, where the first letter should also be capitalized.

Variable and Function Names:

Your variable and function names should be clear about what that variable or function is. Do not use one letter variables, but use abbreviations when it is appropriate (for example: “imag" instead of “imaginary”). The more descriptive your variable and function names are, the more readable your code will be. This is the idea behind self-documenting code.

File Headers:

Every file should have the following header at the top

/*

Author: <your name>

Class: ECE4122 or ECE6122

Last Date Modified: <date>

Description:

What is the purpose of this file?

*/

Code Comments:

1. Every function must have a comment section describing the purpose of the function, the input and output parameters, the return value (if any).
2. Every class must have a comment section to describe the purpose of the class.
3. Comments need to be placed inside of functions/loops to assist in the understanding of the flow of the code.