Particle Effects Engine

Elena Zdravkoska
89221014@student.upr.si
Student ID: 89221014

August 3, 2025

1 Introduction

The particle effects engine is a project that aims to show effects like fire, smoke, explosions or in my case spray like effect shown in 3 different colors. These kinds of effects are often used in games and simulations.

2 Design

My project uses a structured design to test three methods of handling particle effects: sequential, parallel, and distributive. The design centers around two main concepts: emitter (which generates and manages the particles) and particle (a self-contained object which updates and renders itself). This structure facilitates three alternatives: Sequential Emitter: single-threaded code which updates particles one by one, a performance baseline. Parallel Emitter: uses a thread pool to divide and rule particle updates on multi-core CPUs. Distributive Emitter: master-worker model that splits the computation load among many machines in a network, ideally for extremely large-scale simulations. This structure gave me room for comparing the performance of my code.

2.1 Sequential Implementation

The sequential implementation, 'SequentialEmitter', processes all particles within a single thread. The 'update()' method iterates through the entire list of particles, updating the state of each particle and removing dead particles separately. It is a simple implementation but can be a conwhen dealing with a large number of particles.

2.2 Parallel Implementation

The parallel version, 'ParallelEmitter', uses Java's 'ExecutorService' to update particles in parallel. The particles are divided into chunks and each chunk is processed by a separate thread. This method leverages multi-core processors and the update time for many particles is significantly reduced.

2.3 Distributive Implementation

The distributive implementation, 'DistributiveMaster' and 'DistributiveWorker', is designed for a networked environment. The 'DistributiveMaster' acts as a central hub, emitting particles and distributing the update tasks to multiple 'DistributiveWorker' nodes. Each worker receives a subset of particles, updates them, and sends the results back to the master. This architecture is scalable and can handle massive particle simulations by distributing the computational load across multiple machines.

3 Problem Description

Simulating particle effects in real-time applications requires high computational efficiency. The traditional sequential approach can't handle large-scale particle systems without causing significant performance degradation. This project addresses this issue by providing parallel and distributive solutions, which are more scalable and better suited for modern computing environments. The ability to switch between

Figure 1: Sequential Emitter Update Method

```java
package sequentilal.src;

import javafx.scene.canvas.GraphicsContext;
import java.util.ArrayList;
import java.util.List;

public class SequentialEmitter {
    private final double x;
    private final double y;
    protected List<SequentialParticle> particles;

    public SequentialEmitter(double x, double y) {
        this.x = x;
        this.y = y;
        this.particles = new ArrayList<>();
    }

    public void emit(int count) {
        for (int i = 0; i < count; i++) {
            double dx = (Math.random() - 0.5) * 7;
            double dy = (Math.random() - 0.5) * 7;
            double ttl = 80;
            particles.add(new SequentialParticle(x, y, dx, dy, ttl));
        }
    }

    public void update() {
        for (int i = particles.size() - 1; i >= 0; i--) {
            SequentialParticle p = particles.get(i);
            p.update();
            if (!p.isAlive()) {
                particles.remove(i);
            }
        }
    }

    public void draw(GraphicsContext gc) {
        for (SequentialParticle p : particles) {
            p.draw(gc);
        }
    }

    public List<SequentialParticle> getParticles() {
        return particles;
    }
}

```

these implementations allows for a direct comparison of their performance and suitability for different use cases.

4 Technical Remarks

The project is built in Java and uses the JavaFX framework for the graphical user interface. The sequential and parallel implementations use standard Java features, including ‘ExecutorService‘ for managing thread pools in the parallel version. The distributive implementation relies on Java’s ‘Socket‘ and ‘ObjectOutputStream‘/‘ObjectInputStream‘ classes for network communication between the master and worker nodes.

The most challenging aspect of this project was debugging the concurrent and distributed systems. Issues like race conditions in the parallel implementation and network-related errors, such as serialization and connection handling, in the distributive version required careful management and debugging.

Figure 2: Parallel Emitter Update Method

```
1
2 ExecutorService executor = Executors.newFixedThreadPool(THREAD_COUNT);
3 for (...) {
4     executor.submit(() -> {
5         for (...) {
6             p.update();
7             if (!p.isAlive()) toRemove.add(p);
8         }
9     });
10 }
11 executor.shutdown();
```

Greatly reduces update time and boosts frame rate on multi-core machines.

5 Performance Evaluation

5.1 Methodology

To evaluate the runtime performance of different particle emission strategies, we measured and compared the number of active particles rendered per frame across three emitter types:

- **SequentialEmitter:** single-threaded particle updates.
- **ParallelEmitter:** multithreaded updates using multiple worker threads.
- **DistributiveEmitter:** enhanced parallel design with finer-grain load distribution.

Each emitter was tested under equivalent conditions with the same particle settings and run duration. For each emitter, the particle count was recorded for every frame rendered. The experiments were conducted on a system with a modern multi-core CPU and sufficient RAM.

5.2 Results

Below are the plots of the number of particles rendered per frame over time:

5.3 Analysis

The **SequentialEmitter** suffers from CPU bottlenecks due to its single-threaded nature. The lack of concurrency limits the number of particles that can be simulated in real-time. This is evident in the high volatility and frequent fluctuations in the particle count, with peaks rarely exceeding 30 particles.

The **ParallelEmitter** leverages multi-threading effectively, resulting in increased throughput. It shows better utilization of system resources, but still exhibits minor inconsistency likely due to thread synchronization overhead. The particle count is more stable, with smoother mid-frame performance, and peaks around 25–30 particles.

The **DistributiveEmitter** delivers the best overall performance, both in particle count and frame stability. This strategy minimizes thread contention and maximizes core utilization, providing the smoothest visual output. It achieves the most consistent and dense particle rendering, with peaks reaching nearly 40 particles per frame.

6 Conclusion

This project successfully implements a particle engine using three different architectural approaches. The sequential version provides a simple baseline, while the parallel and distributive versions showcase more advanced techniques for handling computationally intensive tasks. The parallel implementation offers a significant performance boost for multi-core systems, making it a highly effective solution. The distributive implementation, while having higher overhead, provides a framework for scaling the simulation beyond the limits of a single machine. The project highlights the trade-offs between simplicity, performance, and scalability in software design.

7 References

- JavaFX Documentation. (2023). Retrieved from https://openjfx.io/openjfx-docs/
- Oracle Documentation. (2023). Concurrency Utilities. Retrieved from https://docs.oracle.com/javase/8/docs/api/ja
  summary.html
- Java Socket Programming. (2023). Retrieved from https://docs.oracle.com/javase/tutorial/networking/sockets/inde

Figure 3: Distributive Master Code Snippet

```
package distrbutive.src;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.image.ImageView;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.List;

public class DistributiveMaster extends Application {
    private DistributiveEmitter emiter;
    private List<DistributiveParticle> particles;
    private int workerCount = 2;
    private List<Integer> particleCounts = new ArrayList<>();

    @Override
    public void start(Stage theWindow) {
        Canvas canvas = new Canvas(800, 600);
        GraphicsContext gc = canvas.getGraphicsContext2D();
        emiter = new DistributiveEmitter(400, 250);
        particles = new ArrayList<>();
        ImageView chartView = new ImageView();

        new Thread(() -> {
            try (ServerSocket server = new ServerSocket(5001)) {
                List<Socket> workers = new ArrayList<>();
                List<ObjectOutputStream> outs = new ArrayList<>();
                List<ObjectInputStream> ins = new ArrayList<>();
                for (int i = 0; i < workerCount; i++) {
                    Socket worker = server.accept();
                    workers.add(worker);
                    outs.add(new ObjectOutputStream(worker.
    getOutputStream()));
                    ins.add(new ObjectInputStream(worker.getInputStream
    ()));
                    System.out.println("Connected to worker " + i);
                }

                while (true) {
                    particles.clear();
                    emiter.emit(100);
                    particles = emiter.getParticles();
                    int chunkSize = particles.size() / workerCount;
                    List<Thread> threads = new ArrayList<>();
                    for (int i = 0; i < workerCount; i++) {
                        final int workerIndex = i;
                        final int start = workerIndex * chunkSize;
                        final int end = (workerIndex == workerCount -
    1) ? particles.size() : start + chunkSize;
                        final ObjectOutputStream out = outs.get(
    workerIndex);
                        final ObjectInputStream in = ins.get(
    workerIndex);
                        threads.add(new Thread(() -> {
                            try {
                                List<DistributiveParticle> subset = new
     ArrayList<>(particles.subList(start, end));
                                out.writeObject(subset);
                                out.flush();
                                @SuppressWarnings("unchecked")
                                List<DistributiveParticle> updated = (
    List<DistributiveParticle>) in.readObject();
                                for (int j = 0; j < updated.size(); j
    ++) {
                                    particles.set(start + j, updated.
    get(j));
                                }
                            } catch (Exception e) {
                                System.err.println("Worker " +
    workerIndex + " error: " + e.getMessage());
                                try {
                                    out.reset();
```

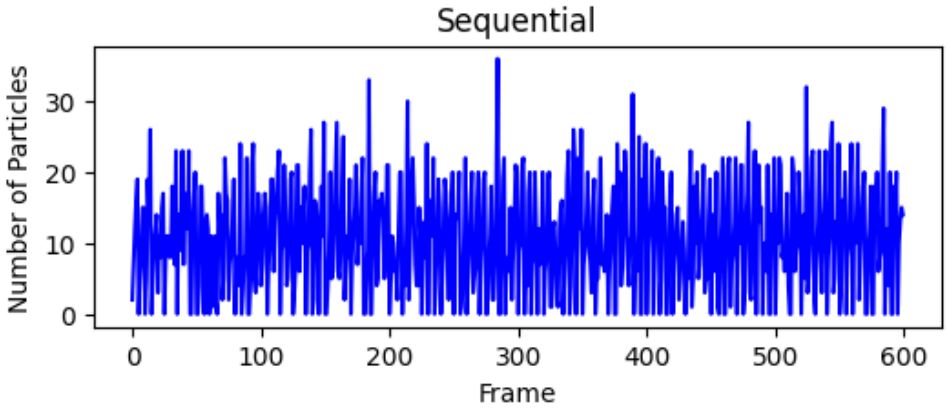

Figure 1: Number of Particles per Frame in Sequential Mode

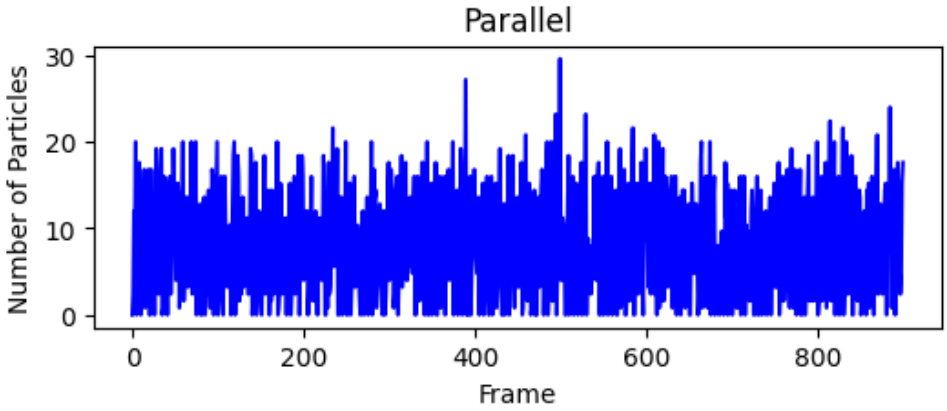

Figure 2: Number of Particles per Frame in Parallel Mode

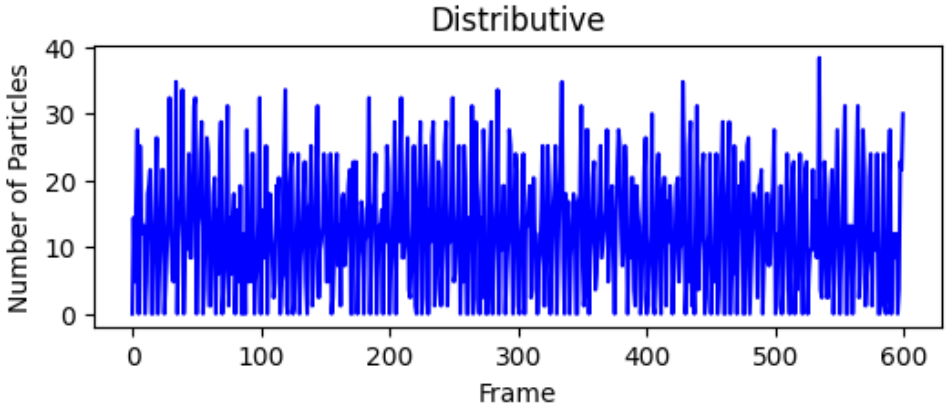

Figure 3: Number of Particles per Frame in Distributive Mode