University of Information Technology & Sciences

Department of
Computer Science and Engineering

Project Report

3D Particle Fountain Simulation

Course Title: Simulation & Modeling Lab
Course Code: CSE-413

Submitted To

Audity Ghosh
Lecturer of CSE Department, UITS

Submitted By

Name	: Mobarok Hossain Zobaer	Name	: Sadia Akter Tuly
Id	: 0432220005101029	Id	: 0432220005101030
Name	: Afnan		
Id	: 0432220005101015		

3D Particle Fountain Simulation

1. Introduction

This project implements a real-time 3D particle fountain simulation using Python. The system models the physics of particles ejected from a central source, subject to gravitational forces, air resistance, and boundary collisions. The simulation demonstrates fundamental concepts in computational physics, computer graphics, and numerical methods.

2. Theoretical Background

2.1 Particle Systems

A particle system is a technique in computer graphics that uses many small sprites or geometric primitives to simulate fuzzy phenomena that are difficult to reproduce with conventional renderings such as fire, smoke, water, and fountains. Each particle is treated as an independent entity with its own position, velocity, and lifespan.

2.2 Newtonian Mechanics

The simulation is grounded in classical mechanics, where each particle obeys Newton's laws of motion. Particles experience gravitational acceleration and velocity damping to simulate air resistance, creating realistic trajectories.

2.3 Numerical Integration

The system uses the Euler method for time integration—a first-order numerical procedure for solving ordinary differential equations. While simple, it provides sufficient accuracy for real-time visualization at small time steps.

3. Equations Used

3.1 Kinematic Equations

Velocity Update (with gravity):

$$v_z(t + \Delta t) = v_z(t) + g \cdot \Delta t$$

Where:

- v_z= vertical velocity component
- $g = -9.8\, m/s^2$(gravitational acceleration)
- $\Delta t = 0.1\, s$(time step)

Position Update:

$$\vec{p}(t + \Delta t) = \vec{p}(t) + \vec{v}(t) \cdot \Delta t$$

3.2 Air Resistance (Damping)

$$\vec{v}_{new} = \vec{v}_{old} \times 0.99$$

This applies to a 1% velocity reduction per frame, simulating drag forces.

3.3 Collision Response

Ground/Wall Bounce:

$$v'_{normal} = -v_{normal} \times e$$

Where $e = 0.6$is the coefficient of restitution (energy loss on impact).

3.4 Initial Velocity Distribution

Radial fountain pattern:

$$v_x = s \cdot \cos{(\theta)}, v_y = s \cdot \sin{(\theta)}, v_z \in [10,14]$$

Where:

- $\theta \in [0,2\pi]$(random angle)
- $s \in [4,7]$(random horizontal speed)

4. System Features

Feature	**Description**
Vectorized Computation	NumPy arrays enable batch processing of all particles simultaneously
Dynamic Spawning	3 new particles spawned per frame with randomized properties
Particle Lifecycle	Each particle has a lifetime of 60–100 frames before deactivation
Boundary Handling	Elastic collisions with ground (z=0) and walls (x,y = ±6)
Object Pooling	Fixed array size (100 particles) with slot reuse for memory efficiency
Visual Effects	Multi-color palette, rotating camera view, transparency effects

5. Dataset / Parameters

5.1 Simulation Parameters

Max Particles:	100
Time Step (dt):	0.1 seconds
Gravity:	-9.8 m/s^2
Air Resistance:	0.99 (per frame)
Spawn Rate:	3 particles/frame
Bounce Factor:	0.6 (coefficient of restitution)

5.2 Spatial Bounds

X-axis:	[-6, 6] units
Y-axis:	[-6, 6] units
Z-axis:	[0, 12] units
Origin:	(0, 0, 0.5) - particle spawn point

5.3 Particle Properties

Initial Horizontal Speed:	4–7 units/s (uniform random)
Initial Vertical Speed:	10–14 units/s (uniform random)
Lifetime:	60–100 frames (uniform random)
Colors:	5-color palette (random assignment)

6. Summary Code

6.1 Core Classes

python

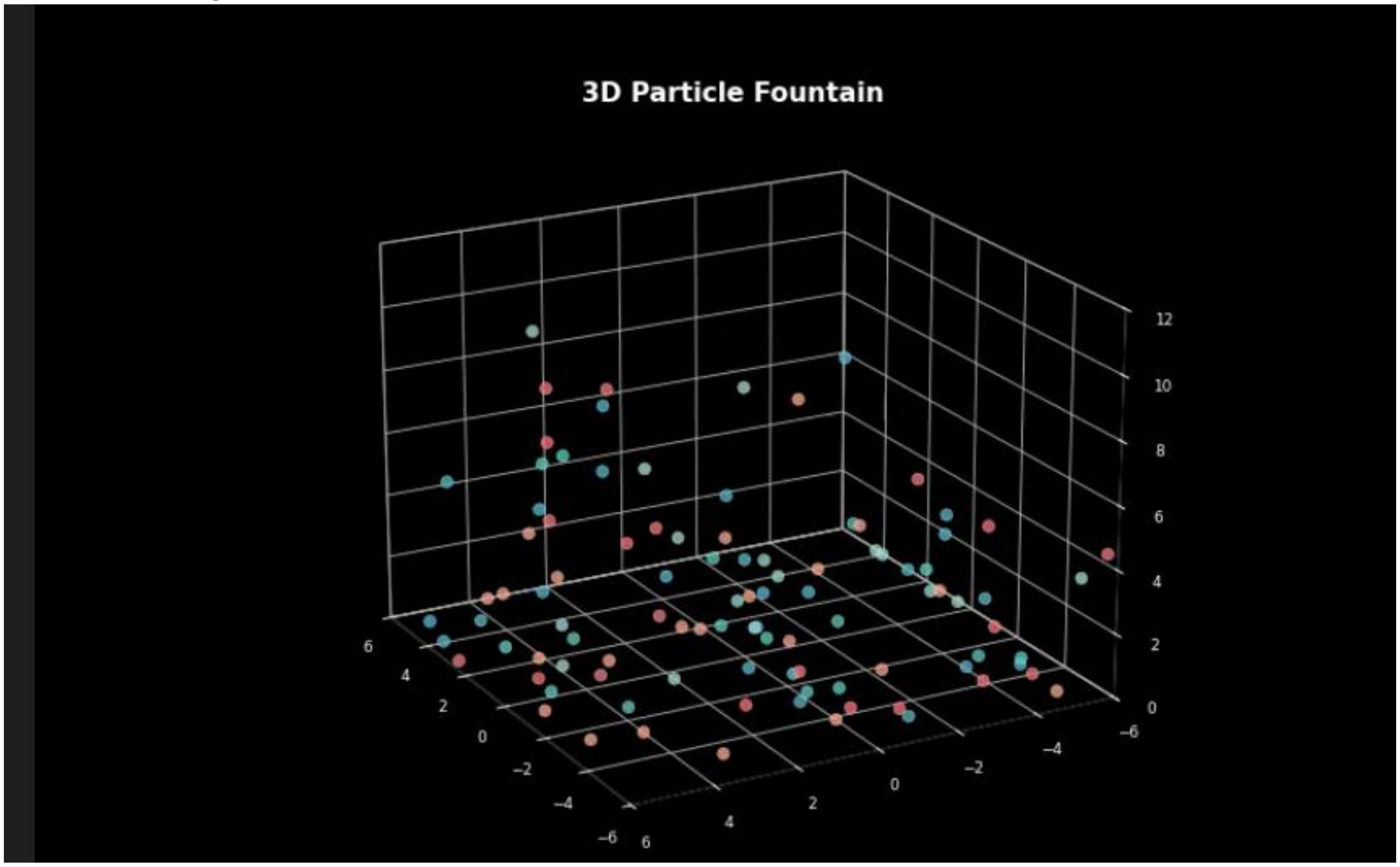
```
class FastParticleSystem:
    def __init__(self, max_particles=100):
        # Pre-allocate arrays for positions, velocities, ages
        self.positions = np.zeros((max_particles, 3))
        self.velocities = np.zeros((max_particles, 3))
        self.active = np.zeros(max_particles, dtype=bool)

    def spawn_particles(self, n):
        # Find inactive slots and initialize new particles
        # with random velocities in fountain pattern

    def update(self):
        # Apply gravity, air resistance
        # Update positions, handle collisions
        # Age particles and deactivate expired ones
```

6.2 Animation Loop

python

```
def update(frame):
    system.update()        # Physics step
    pos = system.positions[system.active]
    ax.scatter(pos[:,0], pos[:,1], pos[:,2], ...)
    ax.view_init(elev=20, azim=frame * 0.8)  # Rotate camera
```

7. Visual Output



8. Result Analysis

8.1 Performance Observations

The vectorized implementation using NumPy demonstrates significant performance advantages over naive particle-by-particle iteration. By processing all 100 particles simultaneously through array operations, the simulation achieves smooth real-time rendering at 20 FPS with minimal computational overhead.

8.2 Physical Realism

The fountain exhibits believable behavior due to the combination of several physical factors. The parabolic trajectories arise naturally from constant gravitational acceleration, while the gradual velocity decay from air resistance prevents particles from bouncing indefinitely. The coefficient of restitution of 0.6 creates visually satisfying bounces that lose energy progressively, mimicking real-world inelastic collisions.

8.3 Visual Quality

The multi-color particle palette combined with the slowly rotating camera creates an engaging visual experience. The particle density reaches a dynamic equilibrium where the spawn rate approximately matches the death rate, maintaining consistent visual fullness throughout the animation.

8.4 Limitations and Future Improvements
The current implementation uses the Euler integration method, which can introduce energy drift over long simulations. More sophisticated integrators like Verlet or RK4 could improve accuracy. Additionally, inter-particle collisions are not modeled particles pass through each other, which reduces physical realism but significantly improves performance.
8.5 Computational Efficiency
The object pooling approach (reusing particle slots rather than creating/destroying objects) eliminates memory allocation overhead during runtime. This design pattern is essential for maintaining consistent frame rates in particle-heavy simulations.

9. Conclusion
This project successfully demonstrates a physically based 3D particle fountain using fundamental principles of Newtonian mechanics and efficient numerical methods. The implementation balances visual quality with computational performance through vectorized operations and smart memory management. The simulation serves as an excellent foundation for understanding particle systems, which are widely used in game development, scientific visualization, and visual effects.

10. References

1. Reeves, W.T. (1983). "Particle Systems—A Technique for Modeling a Class of Fuzzy Objects." ACM SIGGRAPH Computer Graphics.
2. Numerical Methods: Euler Integration for ODEs
3. Matplotlib Documentation: 3D Plotting and Animation
4. NumPy Documentation: Vectorized Array Operations