3D Brownian Motion Simulation: Mathematical Foundations and Implementation

Andrea Simone Costa

October 11, 2025

1 Introduction

This project extends the classical 2D Brownian motion simulation to three dimensions, providing a more realistic representation of particle diffusion in physical systems. While the fundamental principles remain the same, the 3D implementation introduces additional mathematical complexity in random walk generation, collision detection, and visualization. The simulation manages 20000 particles performing random walks in a bounded 3D domain, with optimized spatial grid collision detection and interactive Three.js visualization.

2 Mathematical Background

2.1 Random Walk in 3D Space

Unlike the 2D case where random directions are easily parameterized by a single angle, 3D random walks require careful treatment to ensure uniform directional distribution. The simulation uses spherical coordinates with two angles:

- $\theta \in [0, \pi]$: polar angle (angle from the positive z-axis)
- $\phi \in [0, 2\pi)$: azimuthal angle (angle in the xy-plane from the positive x-axis)

To generate uniformly distributed random directions on the unit sphere, we cannot simply sample θ and ϕ uniformly. Instead, we use:

$$\theta = \arccos(1 - 2u), \quad u \sim \mathcal{U}[0, 1] \tag{1}$$

$$\phi = 2\pi v, \quad v \sim \mathcal{U}[0, 1] \tag{2}$$

This ensures that the solid angle element $d\Omega = \sin\theta \, d\theta \, d\phi$ is uniformly distributed. The displacement at each step is then:

$$\begin{pmatrix} \Delta x \\ \Delta y \\ \Delta z \end{pmatrix} = \Delta r \begin{pmatrix} \sin\theta\cos\phi \\ \sin\theta\sin\phi \\ \cos\theta \end{pmatrix} \tag{3}$$

where Δr is the fixed step size.

2.2 Mean Squared Displacement in 3D

The Mean Squared Displacement for 3D motion is calculated as:

$$\text{MSD}(t) = \frac{1}{N}\sum_{i=1}^{N}\left[(x_i(t) - x_i(0))^2 + (y_i(t) - y_i(0))^2 + (z_i(t) - z_i(0))^2\right] \tag{4}$$

For ideal 3D Brownian motion in an unbounded domain, the MSD grows linearly with time according to:

$$\text{MSD}(t) = 6Dt \tag{5}$$

where D is the diffusion coefficient. Note the factor of 6 (compared to 4 in 2D and 2 in 1D), reflecting the additional spatial dimension.

2.3 Plateau Behavior in Bounded Domains

In a bounded 3D domain with characteristic length L, the MSD reaches different plateau values depending on initialization:

- **Center initialization** (radial distribution): $\text{MSD}_\infty \approx L^2/4$
- **Random initialization** (uniform distribution): $\text{MSD}_\infty \approx L^2/2$

These values differ from the 2D case ($L^2/6$ and $L^2/3$ respectively) due to the geometric properties of 3D space.

To ensure uniform radial distribution when initializing particles near the center, we use:

$$r = r_{\max}\sqrt[3]{u}, \quad u \sim \mathcal{U}[0,1] \tag{6}$$

This accounts for the fact that spherical shells of larger radius contain more volume ($V \propto r^3$).

2.4 Collision Detection in 3D

The Euclidean distance between two particles in 3D is:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{7}$$

A collision occurs when $d < r_1 + r_2$, where r_1 and r_2 are the particle radii. Upon collision, the simulation implements a simplified bounce where the particle moves along the unit vector pointing away from the collision:

$$\hat{\mathbf{n}} = \frac{\mathbf{r}_1 - \mathbf{r}_2}{|\mathbf{r}_1 - \mathbf{r}_2|} \tag{8}$$

The new displacement becomes $\Delta\mathbf{r}_{\text{new}} = \Delta r\,\hat{\mathbf{n}}$, effectively bouncing the particle away from the collision site.

3 Implementation

3.1 Spatial Grid in 3D

The spatial grid optimization is even more critical in 3D than in 2D. Without optimization, collision detection requires checking all $\binom{N}{2} \approx N^2/2$ particle pairs. For $N = 20000$, this means approximately 200 million distance calculations per time step, which is computationally infeasible.

The simulation divides the 3D domain into cubic cells of size $2r$ (twice the particle radius). Each particle is assigned to its containing cell, and collision checks are restricted to particles in the same cell or the 26 neighboring cells (a $3 \times 3 \times 3$ cube minus the center).

The spatial grid is implemented as a hash map:

```
Map<"cellX,cellY,cellZ", Particle[]>
```

For evenly distributed particles, the average number of particles per cell is $k = N/C$, where C is the total number of cells. Each particle checks approximately $27k$ other particles instead of N, reducing complexity from $O(N^2)$ to $O(N \times k)$.

For the implemented parameters ($N = 20000$, domain size 150^3, cell size 2), this provides approximately a 1000-fold speedup.

3.2 Dynamic Grid Updates

A critical implementation detail distinguishes the 3D version from the 2D version: the spatial grid must be dynamically updated after each particle movement. Unlike the 2D implementation which rebuilds the grid at each time step, the 3D version updates the grid incrementally as particles move:

1. Store the particle's old position $(x_{\text{old}}, y_{\text{old}}, z_{\text{old}})$
2. Calculate new position after random walk and collision detection
3. Remove particle from old cell's list
4. Add particle to new cell's list (creating the cell entry if necessary)

This incremental update strategy is essential for performance with 20000 particles, as rebuilding the entire grid 60 times per second would be prohibitively expensive.

3.3 Three.js Visualization

The 3D visualization uses the Three.js library with GPU-accelerated particle rendering:

- **Points geometry**: All particles rendered as a single `THREE.Points` object for GPU efficiency
- **BufferGeometry**: Particle positions stored in `Float32Array` for fast updates
- **OrbitControls**: Interactive camera allowing rotation, pan, and zoom
- **Wireframe boundary box**: Visual guide showing the simulation domain
- **Axes helper**: Small viewport showing coordinate axes orientation

The camera is automatically positioned to frame the entire simulation domain based on the field of view and domain dimensions.

3.4 Boundary Conditions

Reflective boundaries are implemented independently for each coordinate:

$$\text{if } x > x_{\max}: \quad x_{\text{new}} = x_{\max} - (x - x_{\max}) \tag{9}$$

Similarly for $x < x_{\min}$ and for the y and z coordinates. This ensures particles remain within the bounded domain while preserving the distance traveled (in the reflected direction).

4 Software Architecture

4.1 System Overview

The 3D implementation extends the 2D architecture with Three.js for GPU-accelerated rendering:

```
index.ts (Entry Point)
    |
    v
BrownianModel (AgentScript Model3D)
    |-- turtles.create() -> 20000 particles in 3D
    |-- step() -> Random walk + collision detection
    |
    +-- collisions.ts -> Spatial grid (3x3x3 cells)
    |
    +-- msd.ts (MSDChart) -> Chart.js (same as 2D)
    |
    +-- simulation.ts -> Three.js WebGL rendering
```

4.2 Core Modules

BrownianModel (`brownianModel.ts`): Extends AgentScript's `Model3D` for 3D space. Uses `turtles.create()` to instantiate 20000 particles with (x, y, z) positions. The `step()` method performs spherical coordinate sampling for uniform 3D random walks.

Spatial Grid 3D (`collisions.ts`): Hash map with keys `"x,y,z"`. Each particle checks 27 neighboring cells ($3 \times 3 \times 3$ cube). Implements incremental updates: when particle moves, remove from old cell and add to new cell without rebuilding entire grid.

Three.js Renderer (`simulation.ts`): Creates `THREE.Scene`, `THREE.PerspectiveCamera`, and `THREE.WebGLRenderer`. Particles rendered as `THREE.Points` with `BufferGeometry` storing positions in `Float32Array`. Uses `OrbitControls` for interactive camera.

4.3 Execution Flow

Each animation frame (60 FPS):

1. **Random Walk**: For each particle, generate uniform 3D direction via $\theta = \arccos(1 - 2u)$, $\phi = 2\pi v$
2. **Collision Check**: Query $3 \times 3 \times 3$ cell neighborhood, bounce if collision detected
3. **Grid Update**: Remove from old cell (`oldCellKey`), add to new cell (`newCellKey`)
4. **Three.js Rendering**:
   - Update `BufferGeometry` positions from particle data
   - `geometry.attributes.position.needsUpdate = true`
   - `controls.update()` for camera damping
   - `renderer.render(scene, camera)`

4.4 Three.js Integration

Key Three.js setup patterns:

```
// Scene and camera
const scene = new THREE.Scene()
const camera = new THREE.PerspectiveCamera(75, aspect, 0.1, 1000)
camera.position.set(worldSize, worldSize, worldSize)

// Particle geometry
const geometry = new THREE.BufferGeometry()
const positions = new Float32Array(numParticles * 3)
geometry.setAttribute('position',
  new THREE.BufferAttribute(positions, 3))
const particles = new THREE.Points(geometry, material)

// OrbitControls
const controls = new OrbitControls(camera, renderer.domElement)
controls.enableDamping = true
```

Each frame, particle positions are copied to the `Float32Array` and the buffer attribute is marked for GPU upload.

4.5 Performance Optimizations

- **Incremental grid updates**: $O(1)$ per particle instead of $O(N)$ grid rebuild
- **GPU rendering**: `THREE.Points` renders 20000 particles in single draw call
- **BufferGeometry**: Direct memory access avoids per-particle object overhead
- **Spatial hashing**: String keys like `"5,3,-2"` enable fast cell lookup

5 Results and Discussion

The 3D simulation successfully demonstrates key features of three-dimensional Brownian motion:

1. **Linear MSD growth**: Before boundary effects become significant, the MSD grows linearly according to $\text{MSD}(t) = 6Dt$, confirming the theoretical prediction for 3D diffusion.
2. **Plateau behavior**: The MSD reaches different plateaus depending on initialization strategy, with the random initialization plateau being twice that of center initialization ($L^2/2$ vs $L^2/4$), consistent with 3D geometric expectations.
3. **Uniform directional distribution**: The spherical coordinate sampling correctly produces uniform random walk directions, avoiding the directional bias that would result from naive angle sampling.
4. **Computational efficiency**: The spatial grid optimization enables real-time simulation of 20000 particles at 60 FPS, demonstrating the critical importance of algorithmic optimization for large-scale particle systems.

5.1 Comparison with 2D

Several key differences emerge when comparing 3D and 2D Brownian motion:

- **Diffusion speed**: The MSD coefficient changes from 4D (2D) to 6D (3D), reflecting faster exploration of space in higher dimensions
- **Collision complexity**: 3D requires checking 27 cells (vs 9 in 2D), but the reduced particle density in 3D partially compensates
- **Visualization**: 3D requires sophisticated rendering (Three.js) and interactive camera controls, while 2D uses simple Canvas2D
- **Plateau values**: Different geometric factors lead to different plateau relationships (2D: $L^2/6$ and $L^2/3$; 3D: $L^2/4$ and $L^2/2$)

6 Conclusion

This 3D Brownian motion simulation successfully extends classical random walk theory to three dimensions, demonstrating both the mathematical complexity and computational challenges of higher-dimensional diffusion. The implementation balances physical realism with computational efficiency through careful algorithm design, particularly the spatial grid optimization and incremental grid updates. The interactive Three.js visualization provides intuitive understanding of 3D particle dynamics, while the MSD analysis quantitatively confirms theoretical predictions. This project serves as both an educational tool for understanding diffusion processes and a foundation for more complex simulations involving particle interactions, external forces, or non-equilibrium dynamics.