

## Proximity Mesh and Combinatorial Explosion: AEROPVS06

In the vsTASKER simulation platform, the "**Proximity Mesh and Combinatorial Explosion**" demonstration is a high-performance optimization project designed to handle the **computational strain of multi-agent interactions**.

When simulating thousands of independent entities, calculating the spatial relationships between every single pair creates a computational bottleneck known as a **combinatorial explosion** ( $O(N^2)$  complexity). To bypass this limitation, vsTASKER employs a specialized **proximity mesh spatial-partitioning algorithm** to achieve real-time simulation speeds.

---

### The Problem: Combinatorial Explosion

In standard multi-agent systems—such as drone swarms, particle systems, or military battlefield simulations—entities frequently need to know what or who is near them (e.g., for collision avoidance, target acquisition, or behavior alignment).

- **The  $O(N^2)$  Trap:** If a simulation contains **4,000 particles**, a naive brute-force distance check requires every particle to scan all other 3,999 particles. This results in roughly **16 million calculations per frame**.
- **The Consequence:** The CPU quickly becomes overwhelmed, causing the simulation frame rate to drop drastically and rendering real-time execution impossible.

### The Solution: Proximity Mesh Partitioning

To resolve this, vsTASKER integrates a dynamic spatial grid or **proximity mesh**. This technique completely eliminates redundant calculations through a few key steps:

- **Grid Subdivision:** The 2D or 3D simulation space is segmented into a structured grid of localized "cells" or buckets.
- **Localized Registration:** Every entity continuously registers its coordinates strictly to the specific grid cell it currently occupies.

- **Bounded Searching:** When an entity checks its surroundings, it **only calculates distances to neighbors inside its own cell** and immediately adjacent cells.

## Performance Results in vsTASKER

By confining distance checks to local mesh cells, vsTASKER drops the algorithmic complexity from an unmanageable  $O(N^2)$  down to an efficient **near  $O(N)$  linear complexity**.

According to the official VirtualSim Portfolio, this architectural optimization enables vsTASKER to effortlessly simulate the complex, **Brownian clumping behavior of thousands of active particles in real-time at a locked 30Hz** frame rate. It provides the backend foundation required for other dense vsTASKER demos, such as tracking flocks of 3,000 independent Boids or stress-testing up to 20,000 simultaneous tactical entities.

---

If you are developing a project in vsTASKER, let me know:

- Are you designing a **drone swarm, particle system, or tactical military scenario**?
- What is the **target number of entities** you need to support?
- Do you need help implementing **custom proximity logic or Boids rules** within the vsTASKER environment?

In vsTASKER, **Proximity Mesh and Combinatorial Explosion** refers to a high-performance simulation showcase developed by VirtualSim to demonstrate how the software circumvents the computational bottlenecks of large-scale, real-time particle and entity simulations.

Specifically, the project simulates the **real-time clumping behavior of thousands of particles at a steady 30Hz refresh rate**, resolving the extreme computational demand that typically crashes real-time engines.

## The Core Problem: Combinatorial Explosion

In standard multi-entity simulations (such as Brownian movement, drone swarms, or boids), every particle needs to check its distance relative to every other particle to determine interactions like clumping, attraction, or collision.

- **The Math Bottleneck:** Testing interactions for  $N$  particles requires an  $O(N^2)$  algorithmic complexity.
- **The Explosion:** If you simulate 1,000 particles, the system must process  $1,000 \times 1,000 = 1,000,000$  checks per frame. If you scale to 10,000 particles, that explodes to **100 million checks per frame**.
- **The Result:** At standard real-time execution speeds (like 30Hz), this "combinatorial explosion" starves the CPU, lagging or crashing the simulation.

### The vsTASKER Solution: The Proximity Mesh

To bypass the  $O(N^2)$  limit, vsTASKER employs a specialized **proximity mesh** spatial partitioning technique.

- **Spatial Indexing:** The simulation space is divided into a 2D or 3D grid (the mesh). Particles are dynamically indexed into specific grid cells based on their coordinates.
- **Localized Calculation:** Instead of checking every entity on the map, a particle only calculates interactions with neighbors sharing its immediate grid cell or adjacent cells.
- **Efficiency Drop:** This effectively drops the computational complexity toward  $O(N)$ , allowing the simulation engine to completely ignore distant particles that have zero immediate physical impact on each other.

### Key Features of the Demonstration

As highlighted in VirtualSim's portfolio and tutorials, this specific implementation showcases several core strengths of the software:

- **High Performance:** Stabilises the simulation of thousands of interacting, clumped particles at a consistent **30Hz** real-time frequency using OpenGL rendering.

- **Native C++ Execution:** Because vsTASKER operates by translating visual logic diagrams into highly optimized, compiled **native C++ code**, it avoids the overhead lag found in interpreted simulation software.
- **Behavior Testing:** The framework allows users to quickly modify environmental rules (like changing flocking or Brownian physics) within the Scenario Editor and immediately observe how mass entities react in real time.

---

If you want to dig deeper into implementing these optimization techniques, let me know if you would like to:

- See the **C++ code logic structure** vsTASKER uses for grid-based partitioning.
- Learn how this same mesh logic scales up to stress testing 20,000 military or drone entities.
- Explore how to integrate this behavior with 3D visualization engines like **Presagis VegaPrime**.

## Proximity Mesh and Combinatorial Explosion

by VirtualSim

When having a particle's behavior depending on surrounding particles, using a proximity mesh is one way to reduce the combinatorial explosion of the relationships. With only 5 particles evolving onto an area, asking each of them to compute the distance to the others requires  $(5 \times 4)$  distance computations:  $n \times (n-1)$ . Complexity is  $O(n^2)$ ; not dramatic but still.

With 1000 particles, it requires almost 1 million distance computations; and such, many times per second. With an engine kernel running at 30Hz, it may needs roughly 10 millions distance computations per second. Only to check if two particles are close enough to activate a behavior. That never works in real time.

The first optimization consists of avoiding to recompute same distances. When particle A computes the distance to particle B, B should not redo it. This would divide the number of

computation by 2 but, the storage of the result (and its retrieval) would become a problem.

The second optimization is to use a Proximity Mesh which divides the gaming area into a grid. Each cell of the grid lists all particles belonging to it. At a low rate frequency, the Mesh checks that each moving particle is still inside the cell's perimeter. If found outside, the Mesh simply uses the coordinates of the particle to find the cell it belongs to, and moves the particle into its new cell.

Each particle now just has to make distance computations with the particles of the same list, plus those nearby, in case a particle is getting close to the cell borders.

In all cases, performances are drastically improved and real time (30Hz frame rate) can be reached, even with 30000 particles!

To speed up even more the simulation, the Mesh component can run on its own thread, on another CPU core. The engine kernel and the Mesh process then run in parallel.

### **Behavior of the Particles**

In this demo, each particle is moving with random speed and direction. At random time, both speed and direction are changed, iteratively. The obtained result looks like a Brownian movement.

At start, each particle is a newborn. After couple of seconds, each one may clump with any other according to rules explained below. The colors burst at the beginning (big bang) comes from the triggering of the rules, at the same time, for almost all of the particles.

Clumping and scattering

### **Game Plan**

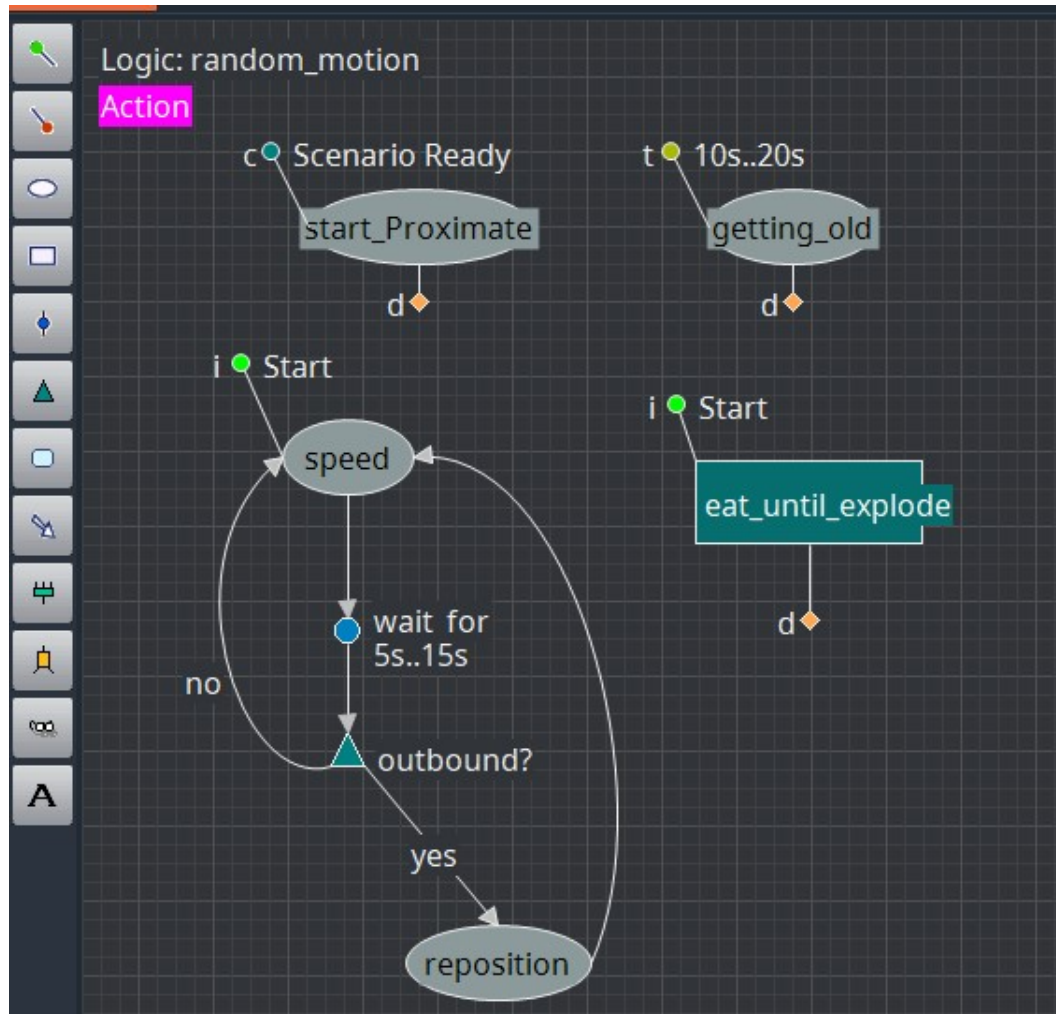
Each particle (called A) does the following at every cycle (30hz):

- A moves at random speed and direction, for a random time, then repeat until clumped;
- if A is a newborn, it draws lines to all particles of the same color around;
- if A is not a newborn, it finds the closest particle B of the same color around, which is also not a newborn:

- If the distance to B minus A+B radius, falls below a given value:
  - if A is bigger than B, B clumps to A, otherwise, A clumps to B;
  - if A and B are of the same size, a dice throw determines who clumps to who;
  - the size (radius) of the clumped one increases by 1/3 the size of the clumping.
- if A radius gets above an obese value, color changes to fuchsia;
- if A radius gets above a maximum value, A explodes and releases all clumped particles; they all become newborns and revert to their initial size;
- After a certain idle time, if A is still clumped, it explodes automatically.

The Mesh component does the following at maximum speed, on another thread:

- Processes all cells of the mesh grid:
  - For each particle A of a cell list:
    - if A position falls outside the cell perimeter, remove A from the list and use position (x,y) to find the cell A belongs to; add A to the new cell list.



Logic given to each particle (called Entity in vsTASKER)

## Conclusion

Multi-threading is not the perfect solution to reduce the complexity of an algorithm but it helps. If the thread is running on another core, the result is good; if all share the same core, the multi-threading effect is not measurable.

The graphics (OpenGL) becomes also a high CPU consumer when the number of particles and connecting lines increases, mostly at high refresh rate (30Hz and more for the IG).

Increasing the number of threads (one per grid cell for example) is not helping at all; the CPU management of the threads suppresses all the benefits. The main vantage of a thread is to parallelize a process which cannot be (easily) time sliced from the code. When it can be done, embedding it into the real-time engine is a very good solution (which frees from the hassle of using semaphores).

A Proximity Mesh process is very easy to time slice from within the code, with a complexity of  $O(n)$ .

| OS & HARDWARE SUPPORT | INDUSTRY APPLICATION |
|-----------------------|----------------------|
|                       |                      |

Note: \*\* Specifications are copied from manufacturer data. Please contact before buying.