SUBSTRATE FIRST

Architecting a Unified Game Engine for Simulation and Emergence

> "Game development should be a pipeline, not a puppet show."

INTRODUCTION: BUILDING BENEATH THE SURFACE

This article is a companion to A Simulated Truth, which explored the vision behind a simulation-first world. Here, I present the foundation of that vision: the substrate, the architecture, the systems. What does it mean to design from the ground up, treating geometry, logic, AI, and physics as consumers of the same data? How can I, a solo developer, accomplish this with a manageable codebase and scalable fidelity?

This is a technical postmortem of a world that hasn't launched yet — but is already being engineered as if it will.

WHY SUBSTRATE MATTERS

The substrate is the deepest layer of the engine: the raw structure of space. Instead of treating geometry as an asset and collision as an afterthought, the substrate defines what exists. It must represent both solid and empty space, be accessible by CPU and GPU, and support all interaction modalities with a single set of semantics.

I chose signed distance fields (SDFs) as the basis for this layer. An SDF encodes the distance from any point in 3D space to the nearest surface. A value of zero lies on the surface. Negative values are inside, and positive values are outside. This means that any point in the game world can be probed — not just for whether something exists there, but for how far away it is and in what direction.

This transforms geometry from a list of surfaces into a field — a medium. And when the substrate is a field, everything else can speak the same language: AI, physics, rendering, logic.

GRIDS AND FIELDS IN DIALOGUE

To support building, editing, and logical reasoning, I layer voxel grids on top of the SDF substrate. These grids are player- or system-editable structures that act as high-level symbolic representations of intended form. Each chunk of grid data exists temporarily in memory as a compact array of material IDs and flags, and is compiled into a volumetric field on demand.

During compilation, grids are translated into unioned primitives in the SDF. If a section is static, its grid representation can be discarded after conversion to save memory. If it's interactive, the grid remains hot, and the SDF becomes a kind of dynamic cache — queried, rebuilt, or diffed as necessary.

This hybrid model — symbolic grid feeding into a physical field — means that building with blocks becomes an act of world-shaping at the simulation level, not just a cosmetic one. You're not placing models. You're placing matter.

MEMORY AND REPRESENTATION CHOICES

In early prototypes, I explored dense float grids for SDF storage. These are intuitive but expensive, especially when scaling up to large or sparse volumes. I migrated toward sparse representations — hashed dictionaries of voxel positions and values — and eventually began developing a sparse voxel octree implementation. Each has tradeoffs in update speed, sampling performance, and integration with GPU-side processes.

Right now, my system supports multiple SDF representations depending on distance and interaction scope: dense fields for high-frequency, local interactions; sparse trees for mid-range; and analytic or procedural shells for far distance.

This multi-resolution strategy keeps performance in check while preserving the benefits of unified simulation.

ECS AND DATA-ORIENTED STRUCTURE

The engine follows a data-oriented architecture inspired by ECS principles. Entities are simple integer IDs. Components are flat arrays, typed and shaped for memory coherence. Systems operate on filtered batches, touching only the data they need.

There are no inheritance trees, no method dispatches. Position, orientation, velocity, mesh reference — all of it lives in dense, type-specific arrays. Behavior becomes a function. Identity is a row number. Logic is explicit and centralized, not fragmented across hundreds of subclass overrides.

This structure lets me simulate thousands of entities without architectural bloat, and makes debugging more of a process of data inspection than of tracking polymorphic call chains.

RENDERING FROM A SHARED SOURCE

Rendering starts with a unified model-matrix instancing pipeline. Meshes are streamed to the GPU as compact arrays. Instances reference their transform and optional material parameters. For dense structures or far-field geometry, mesh generation is deferred or replaced with raymarched shells.

Frustum culling is handled by compute shaders that operate on entity bounding volumes. This keeps CPU-GPU traffic minimal. For performance-heavy scenes, level-of-detail scaling happens at the data level — distant regions may be represented by downsampled SDFs, or even pure mathematical shells.

By making rendering just another system that reads from the substrate, I eliminate redundancy. If something exists, it can be drawn. If it can be drawn, it can be interacted with. There are no invisible ghosts or redundant colliders.

PHYSICS AND EMERGENT BEHAVIOR

Physics in the engine is SDF-aware. Instead of mesh colliders, objects query the substrate directly. Broadphase detection uses bounding box overlaps. Narrowphase resolution uses gradient sampling — essentially, asking the SDF what direction to push objects to resolve penetration.

This makes deformation and destruction seamless. If a section of the field is damaged, removed, or replaced, all systems immediately adapt. There’s no need to sync a separate collision map or notify a physics engine of changes. The field is the engine.

This also enables emergent interactions. A blocked path can be dug through. A collapsed floor can be detected and rerouted around. AI agents react not to tags or zones, but to the space itself.

AI PERCEPTION AND SYMBOLIC INFERENCE

My AI systems are designed to simulate perception, not omniscience. They query the world via raycasts and field samples, resolving visible shapes into symbolic entities. These are then stored in local memory — with positions, descriptors, and inferred relationships.

When an AI sees a tall, spindly, reddish object, it might log it as "a sharp red pillar". If the player labels it as dangerous, the AI associates that shape with danger. Its understanding is grounded in what it sees, not what I’ve told it.

This allows learning, improvisation, and narrative emergence. And again, all of it flows from the same substrate. Visual appearance, physical shape, and behavioral risk all derive from one coherent field.

CONCLUSION: ONE SOURCE, MANY SYSTEMS

The engine I’m building is a conversation between systems, and the common tongue is the substrate.

When all simulation layers read from the same data — when logic, physics, AI, and rendering share the same truth — the world stops being a script and starts being a place.

This isn’t just a technical improvement. It’s a philosophical one. A commitment to building from the inside out.

And when the player walks through that world, the door doesn’t open because someone wrote a line of code. It opens because the simulation says it should.

That’s the kind of game I want to make.

About the Author

Cameron Bains is a self-taught game developer, mechanical engineer, and simulation designer based in Calgary, Alberta. After years working in environmental compliance, process engineering, and AI tool development, he shifted focus to explore the intersections of simulation, emergent gameplay, and artificial intelligence. He is the solo creator of *Astral Trail*, a voxel-based space simulation game built from a custom engine designed around unified data substrates and volumetric physics.

Cameron writes about the future of game engines, data-oriented design, and the creative edge where engineering meets play. He believes the next revolution in interactive worlds will be born not from better textures, but from better foundations.

When he's not coding, he's probably planting something, fixing something, or trying to explain to his dog what raymarching is.