SIMULATED TRUTH: HOW UNIFIED PHYSICS COULD REDEFINE THE FUTURE OF GAME DEVELOPMENT

By Cameron Bains

"The next great simulation game won't come from a studio. It'll come from a garage."

Not because solo developers have better tools. But because they're not burdened by legacy systems, cross-departmental entanglement, or the pressure to release another monetizable loop before quarter's end. Because in one room, under one roof, a single person can build not just a game—but a new kind of engine. One that unifies rendering, physics, and simulation into a single coherent system. One that doesn't fake the world—it models it.

That's what I'm trying to do.

I'm building a game called *Astral Trail*, and with it, a custom voxel-based engine that could one day power a revolution in simulation-driven games. But before I tell you about where it's going, I want to explain where it comes from—not just for me, but for games, for engineering, and for the strange and beautiful intersection between the two.

A HISTORY OF SIMULATION: FROM ENGINEERING TO ENTERTAINMENT

Before games, there were models. Simulations were born not for fun, but for survival: predicting fluid flow in heat exchangers, modeling load across aircraft wings, estimating thrust tolerances in jet engines. In the 1960s and '70s, engineers pioneered early finite element methods (FEM) and computational fluid dynamics (CFD) on room-sized mainframes, solving problems that couldn't be reduced to clean math.

This spirit—the idea that the world could be abstracted and reasoned about through computational systems—slowly bled into the entertainment space. When *SimCity* launched in 1989, it was marketed not as a game, but as a "software toy." No win conditions. No explosions. Just systems. You built roads. You zoned land. You watched as a simulation slowly unfolded, reacting not to your goals but to the rules of its own internal logic.

I didn't play it when it came out. I was born a year later. But a few years down the line, I discovered *SimCity 2000* on a babysitter's old desktop in a basement that smelled like dust and laundry. It had a CRT monitor the size of a microwave and a hum that never stopped. The forgotten Titan abandoned to be a kids toy by adults who did not understand the impression it left with me.

I didn't understand what a simulation was. But I knew I could put a power plant somewhere and watch things grow around it. I knew that traffic moved, that fires started, and that if you clicked the wrong tool, an earthquake might wipe out half your work. The game didn't tell stories—it *became* them.

That was the first time I realized that play didn't have to be reactive. It could be exploratory. A kind of thinking. A kind of design. Even if I didn't have the language for it yet, that was the moment I started to fall in love with systems.

As I grew, so did the complexity of simulation in games. *Microsoft Flight Simulator* became a towering milestone in procedural fidelity. *Farming Simulator*, with its agricultural specificity, struck a chord by refusing to simplify the world. And *Kerbal Space Program*, perhaps more than any other, taught a generation that physics—not graphics—could be the most immersive feature a game has.

But behind every great sim was a compromise. *KSP* modeled thrust and torque beautifully—but its terrain was static. *Flight Simulator* captured wind, lift, and drag—but rarely heat or mechanical fatigue. *EVE Online* simulated economy and risk at planetary scale—but not one ship in it could tip over or dent a hull.

These systems weren't shallow –- quite the contrary, in fact. However, they were just *siloed*. Every game simulated something—but rarely everything. And even in games that aspired to depth, the engine architectures were built for flexibility, not fidelity.

THE PROBLEM WITH MODERN GAME ENGINES

Most commercial engines—Unreal, Unity, Godot—are miracles of abstraction. They can render a film, host a mobile game, or run a first-person shooter. But when it comes to building a simulation that doesn't cheat, the architecture gets in the way.

Visuals are rendered from meshes. Physics operates on convex hulls or colliders. Gameplay logic manipulates higher-level objects through scripts. Each layer speaks its own language, often translating the same object three times. Want your spaceship to deform when it's hit, look burned after reentry, and generate less lift as it heats up? Congratulations—you now need to synchronize your mesh, physics body, shader material, and logic layer. Manually.

This is what I call a fractured substrate. It works—but only up to a point. Beyond that, it collapses under its own complexity.

But what if all those systems were built on the *same foundation*? What if simulation wasn't something you *added* to a game—but the thing the game was made of?

THE ELEGANCE OF SDFS

The first time I encountered signed distance fields (SDFs), it wasn't in a physics engine. It was in Valve's text rendering system, where they used them to anti-alias fonts in *Team Fortress 2*. That's right—one of the most quietly powerful spatial representations in graphics was first

introduced to gaming so your ammo count could look crisp at any scale.

SDFs encode how far you are from the surface of an object, and in which direction. That's it. But from that one function, you can derive collisions, lighting normals, raymarchable geometry, and field-based interactions. It's a unified description of space, and one that plays beautifully with GPU compute pipelines.

Games like *Dreams* and *Claybook* have flirted with this architecture—using SDFs to represent deformable material and blend shapes in ways meshes can't. But these are still rare examples, and they don't push the envelope into physically meaningful simulation. That's where I see the next leap happening.

A GAME ENGINE AS A PHYSICS SYSTEM

When I started building my own engine, I wasn't trying to write a renderer. I was trying to build a world where physics was *honest*. A voxel cracked under pressure because the material couldn't hold. A panel heated and warped based on angle, velocity, and exposure. Objects didn't just break; they *failed*—structurally, thermally, dynamically.

To make that happen, I needed an engine that didn't separate visual representation from physical behavior. One where the shape of a thing *was* the basis of its simulation. That's why I use SDFs. It's why I built an ECS from scratch, and why everything in the world—chunks, terrain, actors, even space—is just data flowing through parallel systems.

And it works. Right now, my engine can render and update millions of individually transformed voxels in real time. It performs GPU-side frustum culling. It batches simulation across layers, from terrain deformation to light diffusion. And it does all of this without treating simulation as a plugin.

There is no "physics engine." The engine *is* the physics.

SIMULATION AS STORY

This isn't just about performance. It's about meaning. When simulation depth rises, systems start to tell their own stories.

The first time my test rocket broke up on ascent—not due to a script, but because the drag torque exceeded the material threshold of a joint—I felt something I hadn't felt in a long time: surprise. Not just at the failure, but at the fact that *I hadn't seen it coming*. The system had outplayed me.

And that, I think, is where simulation is going. Not just toward visual realism. Not just toward better destruction or heat maps or squishy terrain. But toward games that feel *alive* because they are built from the same kinds of interlocking principles that govern real systems.

It's a tall order. But I think it's within reach. Especially for solo developers—because they don't have to compromise.

They don't have to negotiate architecture with teams of dozens. They don't have to worry about supporting every genre. They can build engines that do one thing—and do it right. That's how *Minecraft* happened. That's how *Dwarf Fortress* happened. And maybe, if I'm lucky, it's how *Astral Trail* will happen too.

THE ROAD AHEAD

My goal isn't to out-code Unreal. It's to build something *complete in spirit*. A simulation-native engine that can power a new kind of game—one where systems don't mimic the world. They *are* the world.

It won't be easy. But I believe it's the next frontier for game development. A future where solo developers and small teams don't just make clever mechanics—they make engines that speak physics, live systems, and spatial truth from the ground up.

And when they do, we'll stop calling them hobby projects.

We'll start calling them revolutions.

About the Author
Cameron Bains is a mechanical engineer-in-training and solo game developer based in Calgary, Alberta. After working in emissions modeling, industrial data systems, and AI tooling, he began building *Astral Trail*, a physics-rich voxel game with a custom simulation engine. His development draws from a lifelong love of games like *SimCity 2000*, *Microsoft Flight Simulator*, *Kerbal Space Program*, and *Half-Life 2*, and is grounded in a belief that the best games don't just look real—they behave real.