OpenGL Game Engine Development

Satrajit Ghosh

May 2025

Abstract

This project introduces a custom game engine built from the ground up to support realistic cloth and ball physics in both 2D and 3D environments. The system is architected around a modular, extensible framework that integrates a constraint-based Verlet physics solver with a custom rendering pipeline. All engine components—including simulation, object management, and rendering—were implemented without relying on external game engines or physics libraries. The codebase is structured to support scalability and future enhancements, including parallel computation and agent-based control systems. Developed as a foundation for extended research, this engine also serves as a basis for future master's thesis on intelligent game development environments

1 Introduction

Physics-based simulation is a foundational pillar in modern game engines, enabling dynamic, interactive environments that respond realistically to user input and environmental forces. However, most existing engines abstract away the underlying mechanics, making it difficult to study or modify low-level behavior. This project addresses that limitation by constructing a lightweight yet extensible game engine from scratch, focused on cloth and ball physics in both 2D and 3D settings. It combines a custom-built physics engine based on Verlet integration and spring-mass systems with a programmable graphics pipeline, offering complete control over simulation fidelity and visual output.

The engine is written in C++ and uses OpenGL for rendering, but avoids the use of any high-level game frameworks such as Unity or Unreal. Instead, the system emphasizes modularity, with clearly separated components for physics computation, object representation, shader management, input handling, and rendering. This allows future integration of additional modules such as rigid-body dynamics, AI-driven agents, and parallel computation strategies.

The cloth simulation system models particles as mass points connected by spring constraints, with realistic tearing behavior governed by threshold-based rupture logic. Ball objects interact with the cloth via collision detection and force transfer, producing natural deformation and response. All visual feedback

is powered by a custom shader management system that supports textured lighting models, procedural patterns, and surface-level effects.

The goal is not only to create a simulation platform but also to lay the groundwork for research into agentic control in physically plausible environments. This engine is designed to serve as the base for a Master's thesis project, where autonomous agents will be introduced to interact with the physics system for emergent behavior modeling. The flexibility and transparency of this implementation make it suitable for educational, experimental, and real-time application use cases.

2 Literature Review

The development of a custom game engine requires an in-depth understanding of rendering pipelines, real-time physics simulation, and modular system design. Existing engines such as Unity and Unreal provide robust platforms for game development but abstract away many core systems, making them less suitable for exploring low-level design and educational insights. To address this, the present project takes inspiration from multiple foundational and contemporary sources that span academic research, open-source codebases, educational content, and active community forums.

Godot, a fully open-source and modular game engine, provides significant insights into scalable architecture, component-based design, and community-driven development practices [1]. Its extensive documentation and code organization have served as references for structuring the engine into rendering, physics, and scene subsystems. Similarly, the architecture outlined in Gregory's *Game Engine Architecture* [2] offers formal models and patterns used in production-grade engines, emphasizing separation of concerns, scheduling, and real-time simulation fidelity.

The source code of *Quake* by id Software [3] has been particularly influential in demonstrating how low-level C code can structure real-time rendering, BSP trees, and collision systems efficiently, even on constrained hardware. This historic yet enduringly relevant implementation informs the project's rendering loop and spatial partitioning schemes.

For pedagogical guidance, TheCherno's widely recognized YouTube series [4] offers a practical breakdown of graphics programming, shader management, and ECS (Entity Component System) design, contributing to the hands-on implementation of this engine. Community resources such as the r/gamedev forum [5] provide best practices for GUI frameworks like ImGui and discuss lightweight design patterns beneficial to indie engine development.

Additionally, model-driven engineering approaches in engine construction, as discussed in [6], offer insight into domain-specific language (DSL)-based tooling for maintaining modularity and scalability. While this project does not currently implement a full DSL pipeline, it is architected to allow future integration of such tooling to enhance editor automation and configuration consistency. The Verlet integration method [7], a stable and velocity-free integrator, plays

a foundational role in the physics system of this engine. Unlike explicit Euler integration, which suffers from instability under high constraint stress, Verlet integration provides a robust solution for real-time simulations where structural consistency and energy preservation are critical. Its use in cloth simulation enables iterative constraint solving without explicit velocity storage, aligning well with the architecture of mass-spring systems and making it ideal for the deformable object modules in both 2D and 3D contexts.For practical insights into the numerical differences between Euler and Verlet methods, GorillaSun's article [8] provides visual and implementation-based comparisons that guided the early validation of this engine's integrator module.

Finally, recent research in differentiable rendering and real-time simulation pipelines, such as the survey in [9], underscores the potential for AI-driven extensions. Although the current system focuses on deterministic physics and procedural logic, the architecture is open to future agentic AI integration, aligning with trends in real-time adaptive content.

3 System Architecture

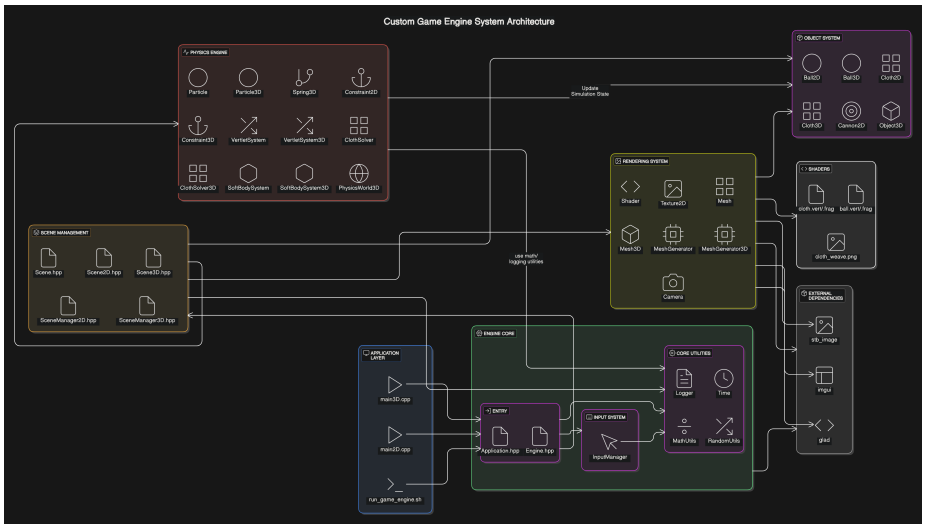

Figure 1: Custom Game Engine System Architecture

This game engine is organized into modular subsystems, each handling a core responsibility such as physics simulation, rendering, scene management, or input handling. The full system structure is visualized in Figure 1.

The **Physics Engine** implements a constraint-based Verlet integration system for simulating soft-body dynamics in both 2D and 3D. Particles are connected by `Constraint2D` or `Spring3D` structures, forming cloths or deformable bodies governed by positional correction iterations. The solvers—`VerletSystem2D`,

ClothSolver3D, and SoftBodySystem3D—manage time integration and constraint satisfaction. All physics updates are centralized under the PhysicsWorld3D, which tracks active physical entities and orchestrates per-frame evolution.

The **Object System** interfaces directly with the physics layer. Game entities like Ball2D, Cloth2D, and Cannon2D contain both physical representations (particles and constraints) and visual components (meshes). Objects define their own update() and render() methods and are responsible for interactions such as ball-cloth collisions and tear logic.

Rendering is performed by the **Rendering System**, using OpenGL and a custom shader abstraction layer. Objects pass geometry (via Mesh or Mesh3D) to the GPU. Shaders are compiled and bound at runtime, including specialized fragment shaders for cloth and ball materials. Meshes are procedurally generated using MeshGenerator classes, which produce vertex arrays including positions, normals, and UVs.

The **Scene Management** subsystem handles lifecycle, update, and render orchestration for all game objects. Scene2D and Scene3D contain registries of active objects, and SceneManager2D / SceneManager3D handle the update and draw calls per frame. This abstraction allows the application logic (main2D.cpp, main3D.cpp) to remain decoupled from object-specific implementations.

Core runtime control is provided by the **Engine Core**, which includes Application.hpp, Engine.hpp, and the main event loop. Input is processed through InputManager, which links user events to gameplay logic such as cannon rotation or projectile firing. Utility modules include logging (Logger), timing (Time), and helper math functions (MathUtils, RandomUtils).

External libraries such as stb_image (texture loading), glad (OpenGL loader), and imgui (debug overlays) are integrated at the engine level, but sandboxed to avoid dependency leakage into object or physics layers.

This architecture isolates responsibilities cleanly while maintaining efficient data flow across modules, making it suitable for real-time physical interaction, shader experimentation, and future AI-driven extensions.

4 Methodology

This section details the simulation, interaction, and rendering methodology implemented in the custom-built game engine. The system operates on modular principles, with distinct subsystems for physics simulation, object dynamics, and visual rendering, all orchestrated via a unified update and render loop.

4.1 Cloth Simulation

The cloth simulation system is based on a Verlet integration approach applied to a grid of interconnected particles. Each particle represents a mass point, and spring constraints define their pairwise distances to preserve local cloth structure. Constraints are categorized into structural, shear, and bend types to mimic the mechanical behavior of real-world fabrics.

In the 3D system, simulation is handled by the `ClothSolver3D` module, which iteratively applies gravitational acceleration and resolves constraints through positional correction. The cloth grid is generated using `generateClothMesh()` from `MeshGenerator3D`, producing vertex and triangle data suitable for GPU rendering. Boundary conditions such as pinned top corners allow the cloth to hang and deform under load.

For 2D, the simulation is structurally similar but dimensionally specialized. Cloth lives in the XY plane and is handled by `Cloth2D`, `Particle`, and `Constraint2D` classes. The same Verlet-based solver logic is adapted to 2D vectors, preserving simulation stability and interactivity. Constraints are resolved using iterative positional correction, and the cloth is rendered either as a wireframe or filled mesh using basic OpenGL 2D primitives. Pinning and tearing logic are also supported, allowing dynamic response to collisions.

4.2 Ball Simulation

Ball dynamics are governed by Newtonian mechanics applied to spherical rigid bodies. Each `Ball3D` object encapsulates position, velocity, mass, radius, and a force accumulator. External forces such as gravity and user-applied impulses are integrated per frame, updating the position and velocity accordingly.

In 3D, ground-plane collisions are handled through restitution-based resolution. Upon penetration, the vertical component of velocity is inverted and scaled. The update loop is lightweight and decoupled, allowing parallel ball-object integration.

For 2D, `Ball2D` objects behave similarly but within the XY plane. Position updates and force integration follow the same mechanics, and collisions are handled with radius-based distance checks. Balls are visualized using circular mesh generators or flat shaders, maintaining visual consistency with the simplified geometry.

4.3 Ball-Cloth Interaction

Interaction between balls and the cloth mesh is handled within the `PhysicsWorld3D` and `PhysicsWorld2D` modules. In 3D, ball positions are tested against cloth mesh particles using broad-phase checks and, if intersected, repulsive impulses are applied based on penetration depth and local surface normals. This simulates realistic momentum exchange and localized deformation.

In 2D, collision checks are distance-based. For each cloth particle, the engine computes the Euclidean distance to the ball center, and if it falls below a contact threshold, a restoring force is applied. Additionally, cloth constraints that experience excessive stress during collision are marked for removal, enabling real-time cloth tearing.

4.4 Rendering and Shader Pipeline

Rendering in the 3D engine is implemented through a programmable OpenGL pipeline that separates object geometry, material properties, and GPU operations. Game objects such as cloth and ball instances encapsulate structured mesh data—positions, normals, and UVs—uploaded to GPU memory via the `Mesh3D` layer using vertex and index buffers.

Shader compilation and linkage are managed by the `Shader` class. Each object type is associated with a fragment shader tailored to its visual characteristics: cloth uses a textured shader with normal-aware lighting, while balls employ a simplified Phong shader with uniform material constants. Both rely on a shared vertex shader that transforms mesh geometry into clip space using the camera's view-projection matrix.

For 2D rendering, the system leverages a parallel shader pipeline with reduced complexity. Meshes are rendered using flat-shaded geometry or basic textures. The 2D `MeshGenerator` classes output vertex data for circular or grid-based objects, and rendering is executed via orthographic projection. Uniforms such as color, scale, and transformation matrices are dynamically bound using utility methods from the `Shader` class, allowing seamless runtime control.

At runtime, mesh attributes are streamed through the vertex shader, and the appropriate fragment shader computes final pixel colors using lighting inputs and textures where applicable. The rendered output is written to the framebuffer in a unified draw call.

This system enables modular rendering logic per object type while maintaining performance through efficient buffer binding and GPU execution. The full data flow—object to mesh to shader—is illustrated in Figure 2.

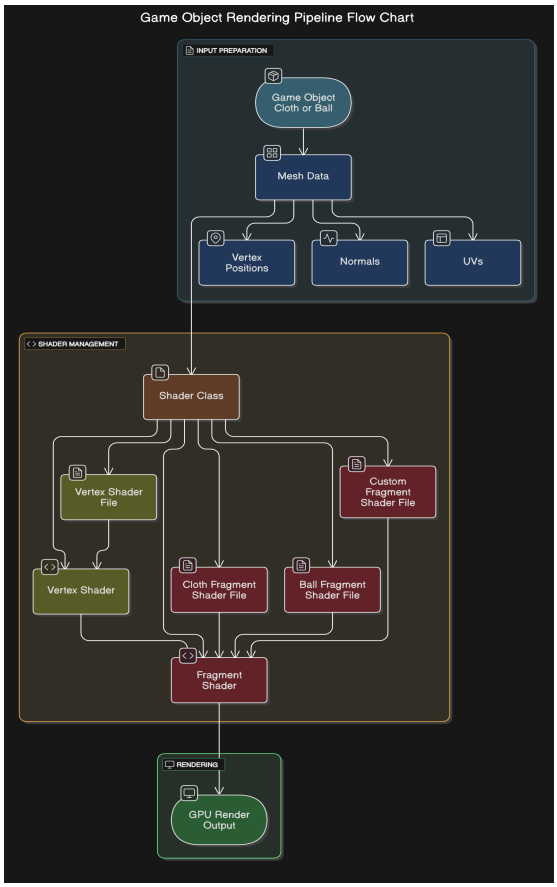

Figure 2: Game Object Rendering Pipeline Flow Chart showing how mesh data, shaders, and rendering stages interact.

5 Results

The current implementation demonstrates real-time physics-based rendering in both 2D and 3D contexts. In the 3D simulation, shown in Figure 3, the cloth mesh responds dynamically to environmental forces and object collisions. The deformation and shading effects are computed using a mass-spring system paired with fragment shader-based lighting, validating the correctness of the simulation and rendering pipeline.

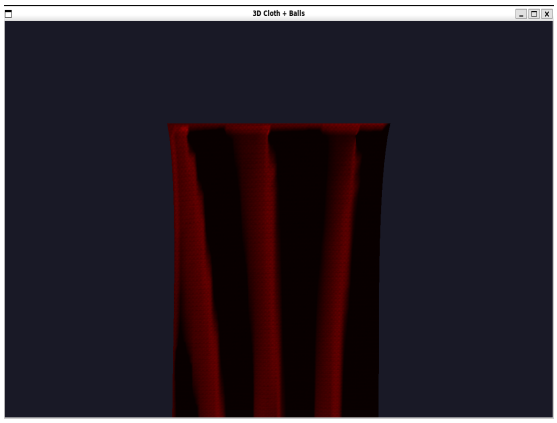

Figure 3: Real-time 3D cloth simulation demonstrating dynamic response to gravity and lighting.

For verification in a simplified setting, a 2D cloth simulation was developed. As shown in Figure 4, the cloth responds to projectile collisions using a grid of particles and spring constraints. This simulation validates constraint satisfaction and force propagation mechanisms in the engine.

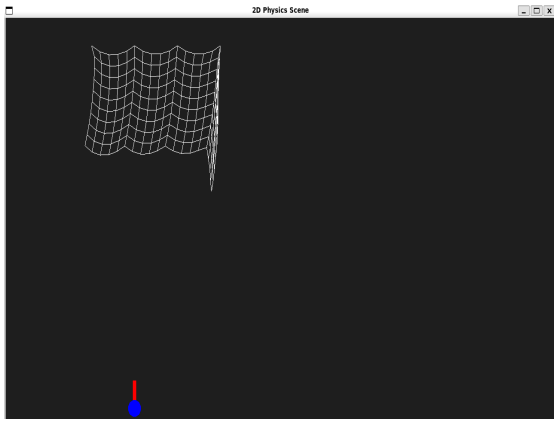

Figure 4: 2D cloth simulation with spring-mass particle system interacting with falling ball.

The engine also supports texture and lighting refinement as shown in Figure 5, which highlights material-based rendering of soft bodies under directional illumination. These visual outputs confirm integration between mesh dynamics and fragment shader effects.

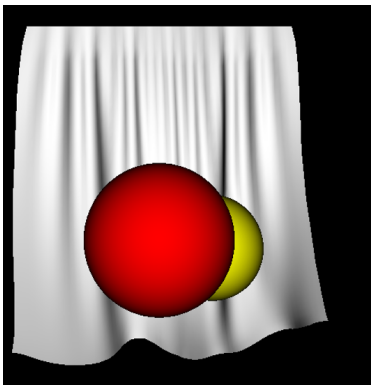

Figure 5: Textured 3D cloth simulation with dual ball interaction and lighting effects.

In terms of usability, Figure 6 shows the current minimal scene editor that enables users to spawn objects, run physics updates, and interact with cloth in real time. The toolchain is designed to be modular and easily extensible.

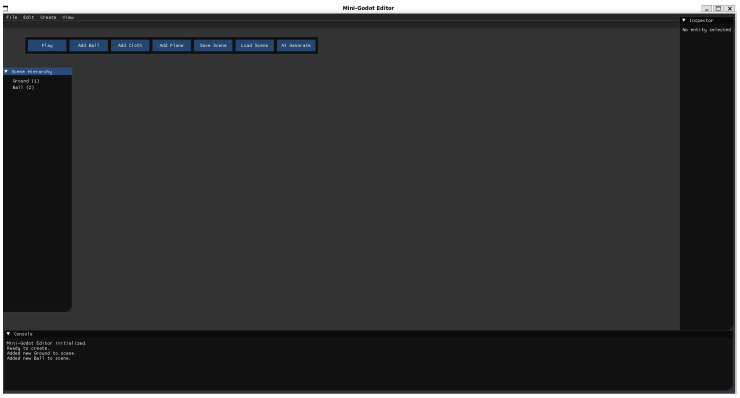

Figure 6: Current implementation of the in-house scene editor for 2D/3D simulation control.

Looking ahead, development is underway on a futuristic editor interface inspired by procedural and agentic scene generation. Figures 7, 8, and 9 display mockups and early prototypes of an AI-assisted content creation interface, enabling natural language-based scene construction, material editing, and physics configuration.

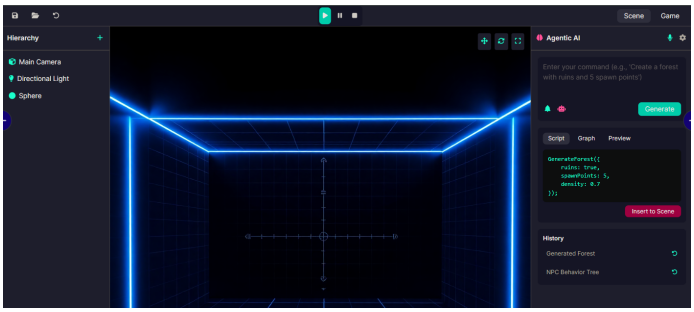

Figure 7: Future AI-driven UI design for interactive scene authoring.

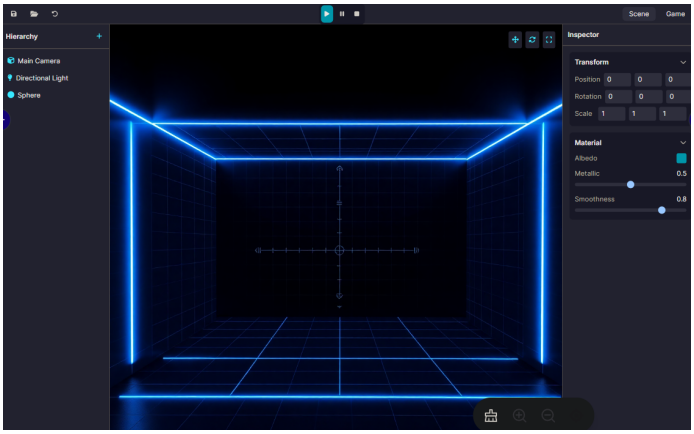

Figure 8: Visual editor mockup integrating AI-generated object placement and parameter control.

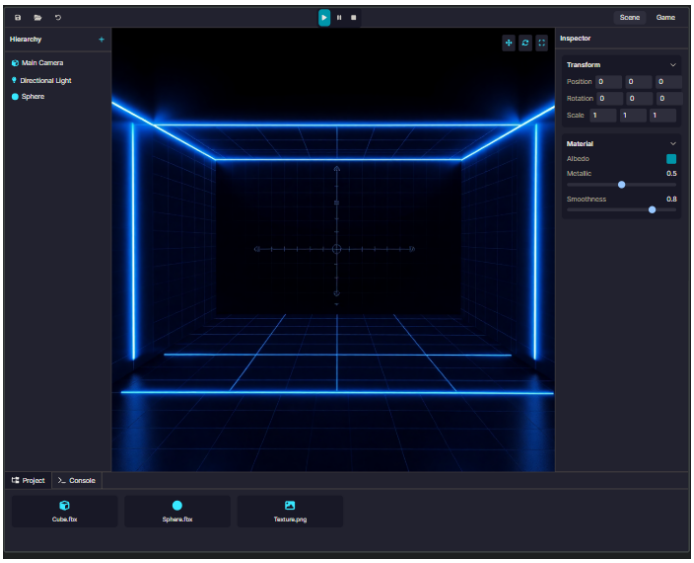

Figure 9: Interactive shader and material inspector from the futuristic editor prototype.

Supplementary Material

Additional visual results and simulation demonstrations are available at the following links:

- 3D Cloth Simulation Demo
- 2D Cloth with Tearing Interaction
- Shader-Based Ball Rendering and Cloth Demo

These results collectively affirm the engine's current capabilities and establish a foundation for future work on intelligent, user-assisted simulation environments.

6 Conclusion and Future Work

This project introduces a modular real-time engine designed to simulate and render physically accurate interactions between cloth and ball objects in both 2D and 3D. By implementing a custom Verlet-based physics system, GPU-backed mesh management, and a programmable shader pipeline entirely from scratch, the engine enables granular control over simulation and rendering. The results validate its ability to produce dynamic cloth deformation, realistic collisions, and visually coherent output under a unified architecture.

Looking ahead, this engine will evolve into a fully functional game engine equipped with real-time editing, procedural generation, and intelligent content creation. Planned extensions include advanced collision handling, volumetric soft-body simulation, physically based rendering, shadow mapping, and GPU compute acceleration. A major milestone will be the integration of an AI-assisted editor and agentic behaviors, transforming the platform into a robust sandbox for interactive simulation, game development, and embodied AI research.

References

[1] "Godot engine contributor documentation," 2024, https://docs.godotengine.org/en/stable/contributing/how_to_contribute.html.

[2] J. Gregory, *Game Engine Architecture*. CRC Press, 2021.

[3] id Software, "Quake source code (winquake)," 1997, https://github.com/id-Software/Quake/tree/master/WinQuake.

[4] TheCherno, "Game engine series on youtube," https://www.youtube.com/@TheCherno/videos.

[5] "r/gamedev: Lightweight c++ gui discussion," 2023, https://www.reddit.com/r/gamedev/comments/122wegt/lightweight_c_gui_libraryframework_for_games/.

[6] T. Mayerhofer, M. Wimmer, and G. Kappel, "Building a game engine: A tale of modern model-driven engineering," *arXiv preprint arXiv:2406.05487*, 2024.

[7] Wikipedia contributors, "Verlet integration — Wikipedia, The Free Encyclopedia," https://en.wikipedia.org/wiki/Verlet_integration, 2024, [Online; accessed 08-May-2025].

[8] GorillaSun, "Euler and verlet integration for particle physics," https://www.gorillasun.de/blog/euler-and-verlet-integration-for-particle-physics/, 2022, [Online; accessed 08-May-2025].

[9] P. Author, "A survey on differentiable simulation and rendering pipelines for games," *arXiv preprint arXiv:XXXX.XXXXX*, 2024, manually referenced from provided PDF.