2D Game Engine and Visual Level Editor

Design and Implementation Report

Callum Jones
22010031

https://github.com/calwjones/GameEngine

UFCFXK-30-3
Digital Systems Project

Word count (excluding Literature Review): 3982

UWE Bristol

Abstract

This report presents a custom 2D game engine and visual level editor built from scratch in C++17. The engine provides deterministic physics through a fixed-timestep game loop, AABB collision detection with spatial partitioning, and a three-layer architecture separating the editor, engine, and game logic. The editor, built with Dear ImGui, allows a user to create, modify, and playtest 2D platformer levels entirely through the GUI without writing any code. Features include a scene hierarchy, property inspector, entity palette, undo/redo, multi-select, grid snapping, a smooth-follow camera, multiple enemy types, collectibles with scoring, and audio support. A structured manual test plan validated the system across functional, non-functional, positive, and negative execution paths. The report evaluates the design choices against established literature and reflects on the development process.

Table of Contents

Table of Figures

1. Introduction

Commercial game engines such as Unity and Unreal Engine offer rapid development, but they hide their internals behind layers of abstraction. For a developer attempting to understand how a game engine actually works, how physics is simulated, how entities are managed in memory, and how an editor communicates with a runtime, these tools can become more of a barrier than a help. Fabian (2018) describes this as the "Abstraction Penalty": the overhead and opacity that comes from using high-level commercial engines. For an academic project focused on systems-level understanding, building from scratch is the more appropriate approach (Tan et al., 2024).

This project presents a custom 2D platformer game engine and visual level editor built from the ground up in C++17. The goal was not only to produce a working piece of software, but to demonstrate a thorough understanding of real-time systems architecture, software design patterns, and data-driven development. The end result is a tool where a user can create game levels visually, place and configure entities, test them in real-time within the editor, and save their work, all without writing any code. The engine uses SFML for windowing, rendering, and input, Dear ImGui for the editor interface, and RapidJSON for level serialisation. It targets cross-platform builds on macOS and Windows via CMake.

1.1 Aims and Objectives

These objectives were chosen to deliberately target areas beyond existing experience. Prior to this project, there was no significant experience with C++, low-level memory management, or graphics programming. Topics such as fixed-timestep simulation, collision resolution, spatial partitioning, immediate-mode GUI integration, and the command pattern for undo/redo were all entirely new. The project was scoped to be ambitious but achievable, with each objective requiring the study and application of techniques not covered in taught modules. Choosing C++ specifically was motivated by the desire to develop these skills, as it is the industry standard for systems-level game development (Gregory, 2018). The primary aim was to build a deterministic 2D engine paired with an immediate-mode GUI editor that allows for visual level creation, and in doing so, to evaluate whether established techniques from the literature review could be successfully applied to produce a functional, integrated development tool. The success of this integration is evaluated in Section 5 against the requirements identified in the literature. The core objectives were:

1. Engine Core: Implement a fixed-timestep game loop, deferred entity management, gravity and friction physics, AABB collision detection with spatial partitioning, and a smooth-follow camera.

2. Visual Editor: Build a Dear ImGui-based editor with a scene hierarchy, property inspector, entity palette, grid snapping, viewport pan and zoom, multi-select, and undo/redo.

3. Data-Driven Levels: Serialise levels to JSON so that games can be authored entirely through the editor, with save, load, and Save As workflows.

4. Stable Simulation: Guarantee frame-rate-independent physics using an accumulator-based fixed timestep at 60Hz and verify that behaviour remains identical between the editor's play mode and standalone execution.

5. Playable Demo: Validate the engine through a playable platformer with player control, patrol enemies, flying enemies, shooting enemies, and collectibles.

6. Structured Testing: Validate the system through a manual test plan covering functional correctness, edge cases, and usability, providing traceability against the objectives defined above.

2. Literature Review

When reviewing the project goals for a custom 2D game engine and visual editor, the focus areas include engine structure, programming languages, real-time systems logic, and graphical user interface (GUI) models. This review analyses technical approaches to these areas, comparing existing solutions against the development tool proposed for this project to justify architectural decisions (Gregory, 2018; McShaffry and Graham, 2013).

A central problem addressed is the "Abstraction Penalty" of high-level commercial engines. While Unity or Unreal provide rapid development, they introduce significant overhead hidden from the developer, creating a "black-box" environment where performance bottlenecks are difficult to diagnose (Fabian, 2018). These engines are often unsuitable for academic projects focused on low-level systems and memory layout (Tan et al., 2024). Instead, by building a rudimentary engine and editor from scratch, this project creates a "white-box" environment where every subsystem is tailored to specific developer needs, allowing for a deeper exploration of fundamental computer science principles (Stroustrup, 2013).

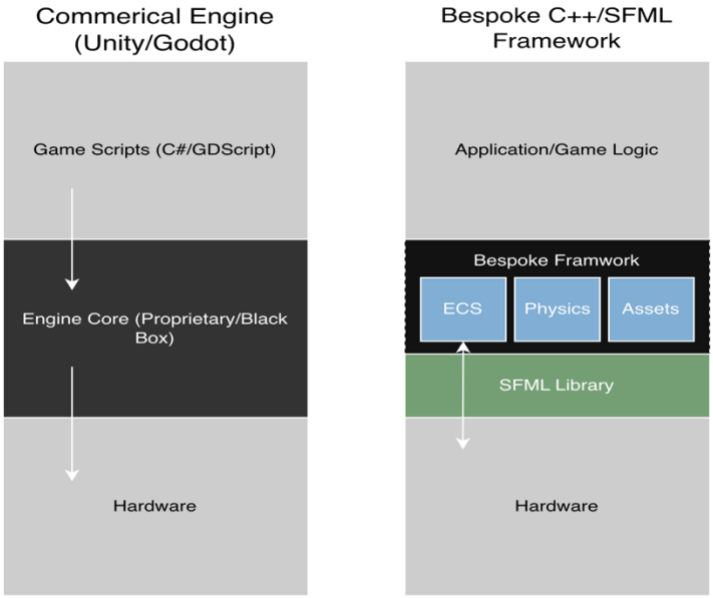

Figure 1: Visualising how a custom C++/SFML framework provides more direct access to system resources compared to commercial abstractions.

Selecting a programming language requires balancing simple syntax against hardware control. Managed languages like Python or Java offer rapid prototyping but rely on non-deterministic garbage collection and virtual machine overhead, causing unpredictable lag in real-time simulations where millisecond precision is required (Meyers, 2014; Horntvedt and Åkesson, 2019). C++ is the industry standard for systems-focused projects due to its support for fixed object lifetimes, minimal overhead, and precise control over memory layout (Stroustrup, 2018; Gregory, 2018). Manual memory management and pointer arithmetic are critical learning outcomes for this study due to the use of C++ (Blow, 2004). By using C++, the engine can utilise contiguous memory allocation to minimise "Cache Misses": a common bottleneck in 2D systems where hundreds of objects must be updated every frame (Shirley and Ashikhmin, 2020). This hardware-conscious approach prioritises execution speed over developer convenience, essential for high entity counts.

Similarly, selecting a multimedia library and GUI model requires significant consideration. While SDL is a common C++ choice, SFML (Simple and Fast Multimedia Library) is selected for its modern, object-oriented interface, handling windowing and input "boilerplate" without imposing a rigid engine structure (SFML Development Team, 2018; Savchenko, 2000). To bridge the gap between a simple program and a complex engine, the system includes a visual editor built with Dear ImGui. The Immediate-Mode Graphical User Interface (IMGUI) approach is chosen for its low state-management overhead, contrasting with the heavier complexity of older "retained-mode" systems (Gregory, 2018). Unlike traditional GUIs that maintain permanent object lists, ImGui rebuilds the interface every frame from the game's current state. This simplifies the connection between engine data and visual panels, allowing for a rapid "edit-and-play" workflow where properties (like position or physics settings) are modified directly within the window (Kelly, 2012; Cornut, 2024).

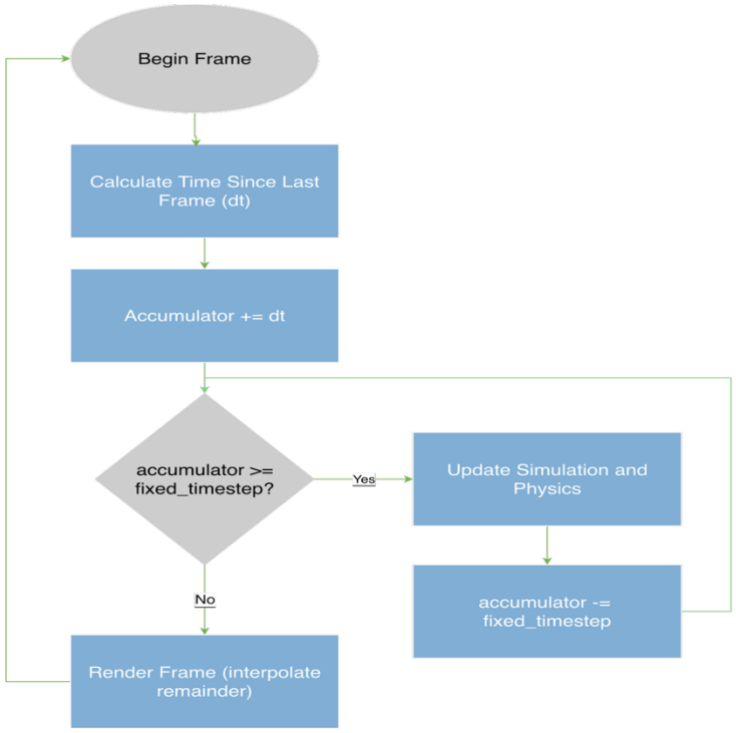

Figure 2: Diagram of the "Fixed Update, Variable Render" pattern, highlighting the separation of simulation and rendering.

The game loop serves as the core of the engine (Mileff, 2023). A common dilemma is whether to use a variable or fixed timestep. Variable timesteps are simpler to build but, as Fiedler (2004) demonstrates, lead to unstable physics. If the frame rate drops, objects can "tunnel" through solid walls, missing collision checks entirely. To ensure consistency across hardware, this framework uses a fixed timestep with an accumulator (Fiedler, 2004; Millington, 2010). This ensures the underlying physics and logic always update at a constant rate, even if rendering speeds vary. This method is technically superior for collision accuracy and provides a stable foundation for the visual editor to pause and step through simulation states without losing precision (Mileff, 2023).

Traditional game development relies on deep inheritance hierarchies (OOP), which Nystrom (2014) notes leads to "brittle" code where base class changes can break hundreds of subclasses. While modern research suggests the Entity-Component-System (ECS) pattern to maximise speed (Harris, 2022; Fabian, 2018), this framework uses a modular OOP approach centred around a dedicated EntityManager. This decision is based on the need for tight integration with the visual editor's Scene and Properties panels. A structured class setup allows the ImGui-based editor to easily modify entity attributes through a predictable interface, which is significantly more complex in a pure ECS model where data is decoupled from identity (Masiukiewicz et al., 2019). This flexible OOP model achieves a balance between C++ speed and the clear logic required for a solo developer to build complex editor features (Freitas et al., 2012; Hall, 2014).

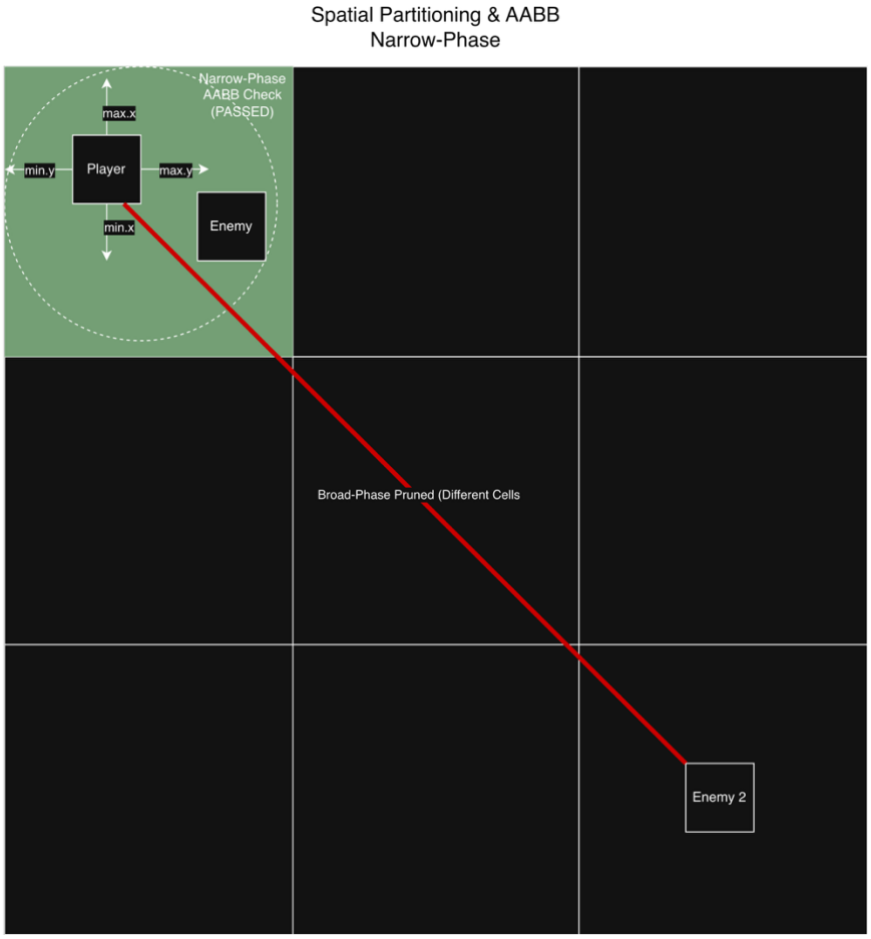

Figure 3: Visual representation of how the engine handles spatial partitioning to optimise collision detection.

Physics simulation in 2D engines requires a pragmatic balance between fidelity and performance. This framework uses a kinematic solver based on Axis-Aligned Bounding Box (AABB) detection (Ericson, 2005). AABB is efficient and suited for tile-based environments, allowing for fast intersection checks without complex math (Mileff, 2023). To improve performance, a spatial partitioning grid is used to ensure the engine only performs collision checks between nearby objects (Ericson, 2005; Millington, 2010). This optimisation is vital for maintaining a stable update frequency as scene complexity increases.

Finally, a data-driven approach is adopted to ensure the engine remains flexible. Storing level layouts and entity attributes in external JSON files enables the visual editor to save and load scene data in real-time without recompilation (White, 2020). Using a high-performance parser like RapidJSON allows the academic focus to remain on the design of the data schema (Tencent, 2016). This is supported by a centralised Resource Manager utilising smart pointers (std::shared_ptr and std::unique_ptr) to ensure textures and fonts are only loaded once. This manages resource ownership and ensures correct deallocation, preventing memory leaks and duplication in resource-heavy environments (McShaffry and Graham, 2013).

This literature review has evaluated the core parts of the proposed custom game engine and editor. The selection of C++ (an OOP language) and SFML provides a balance of control and efficiency, while Dear ImGui elevates the project from a framework to a functional development tool. The use of a fixed timestep ensures that the engine is stable while the AABB physics solver provides collision detection suitable for tile based environments. Finally, the JSON based data driven approach ensures even more flexibility. Overall, all of these steps were specifically selected based on the literature to provide a well functioning custom 2D game engine and visual editor.

3. Design

As established in the literature review (see Section 2), the reviewed sources identified the need for deterministic simulation, scalable collision detection, an immediate-mode GUI for editor tooling, a modular entity model, and data-driven level storage. This section describes how those requirements shaped the overall design of the system.

3.1 Architecture

The system follows a strict three-layer architecture. The top layer is the Editor, which owns the application window, all GUI panels, and the top-level event loop. The middle layer is the Engine, which provides all core systems: a game loop, entity management, physics, collision detection, rendering, input handling, game state management, a camera, audio, and level loading. The bottom layer is the Game, which defines the concrete entity types (player, enemies, collectibles, projectiles) that build on the engine's abstractions.

The critical design constraint is that dependencies only flow downward, shown in figure 4. The engine has no knowledge of the editor, and the game layer has no knowledge of the editor. This means the editor can be removed entirely and the engine and game will still compile and run. It also means new entity types can be added to the game layer without modifying any engine or editor code, provided they register themselves with a central entity factory. This separation was influenced by established software engineering principles around modularity and separation of concerns (Gamma et al., 1995; Gregory, 2018).

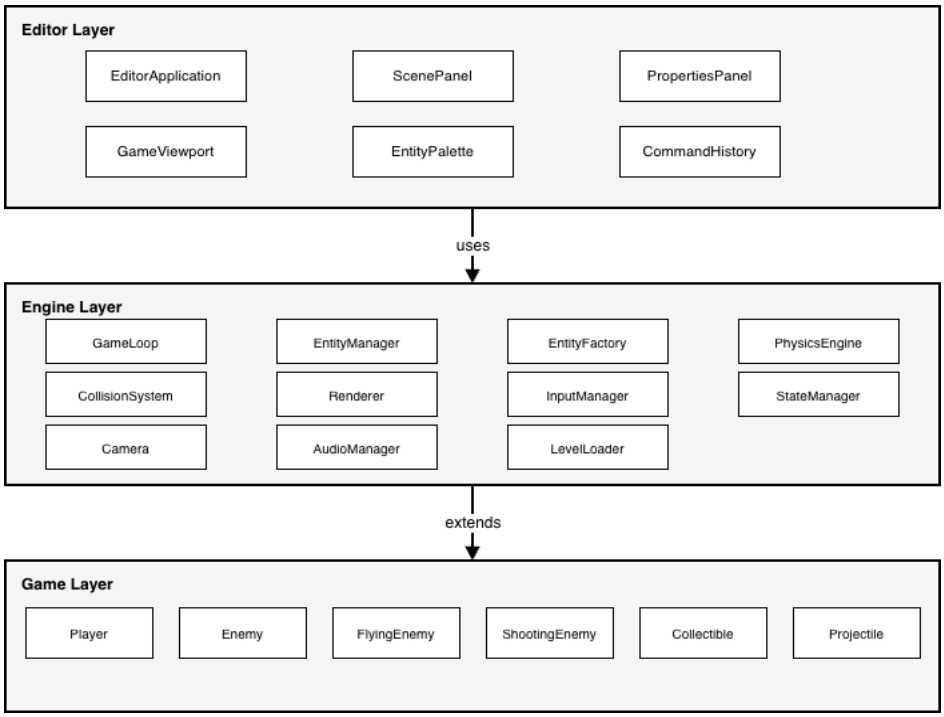

Figure 4: Three-Layer System Architecture Diagram. The downward flow of dependencies ensures the Engine layer remains decoupled from the ImGui Editor interface.

3.2 System Design

The engine needs to provide a simulation that behaves identically regardless of hardware performance. As discussed in Section 2, a fixed-timestep approach with an accumulator achieves this (Fiedler, 2004). The collision system needs to detect and resolve overlaps between axis-aligned rectangles, and must scale beyond small entity counts without significant performance loss. As identified in the literature review, spatial partitioning addresses this by reducing the number of pairwise checks (Ericson, 2005). Entity lifecycle management must be safe during iteration, meaning entities can be created or removed mid-frame without corrupting the update loop. This requires a deferred approach where changes are queued and only applied once the current frame's processing is complete.

Several alternative approaches were considered and rejected during the design stage. An Entity-Component-System (ECS) architecture was evaluated based on the literature review (see Section 2), but was rejected because the editor's property inspector and undo/redo system both require a direct, stateful mapping to entity objects. In an ECS model, where data is distributed across component arrays and decoupled from identity, implementing a property panel that displays and edits a single selected entity becomes significantly more complex (Masiukiewicz et al., 2019). Similarly, the command pattern for undo/redo relies on capturing and restoring named entity state, which maps naturally to an object with fields but poorly to a set of loosely associated components. The inheritance-based model was therefore a deliberate design choice driven by the editor requirements, not simply a default. For collision, brute-force O(n squared) pairwise checking was considered sufficient for early development but was designed from the outset to be replaceable with spatial partitioning once entity counts grew, following the principle of starting simple and validating correctness before adding optimisation complexity.

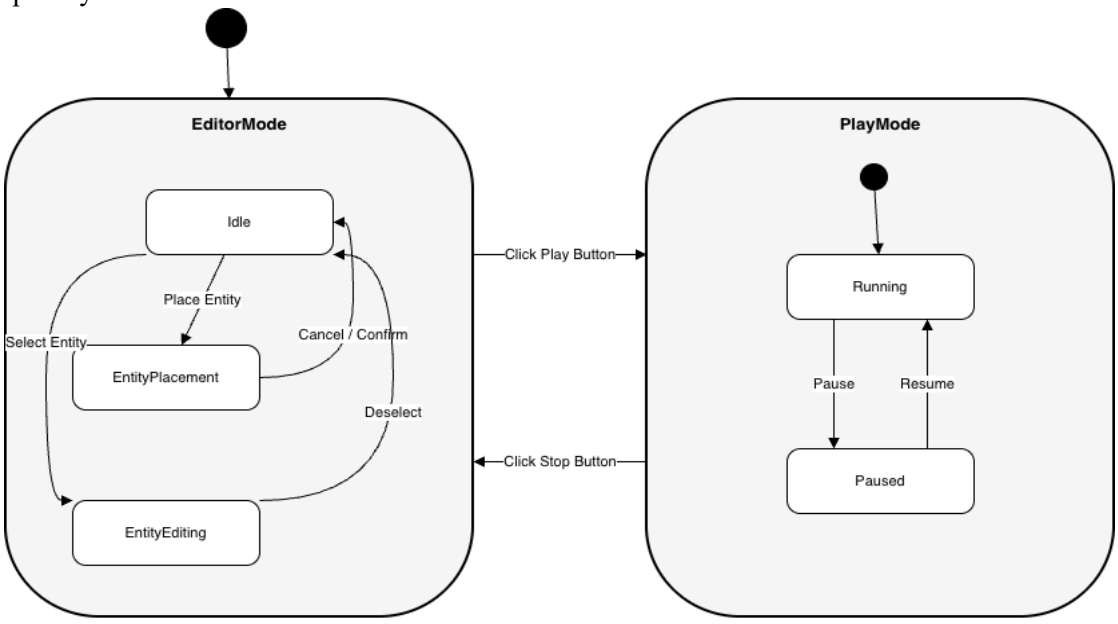

Figure 5: UML State Machine Diagram of Application Modes. Detailing the execution states and transition triggers between the level editor and the active game viewport

The editor needs to embed a live game view within its interface, provide panels for inspecting and editing entity properties, and support visual manipulation of entities in the viewport (selecting, moving, and multi-selecting). It also needs to support reversible editing through an undo/redo system, illustrated in figure 5. The design for this follows the command pattern (Gamma et al., 1995), where every edit creates a command object that knows how to reverse itself. A key design challenge is handling deleted entities: when an entity is removed, the undo system must be able to restore it later, which requires removing entities from the active list without destroying them.

The game layer needs a player entity with keyboard movement and jumping, multiple enemy types with varied AI behaviours (patrolling, flying, and shooting), collectible trigger entities, and projectile entities. Each entity type must handle its own behaviour through polymorphic methods, so that the engine and editor do not contain type-specific logic. An entity factory maps type identifiers to constructors so that entities can be created from data without hardcoding (see figure 6).

Level data must be stored externally so that levels can be saved and loaded by the editor without recompilation. Each entity record needs to store its core properties (position, size, type, colour, physics flags) as well as type-specific properties (patrol speed, fire rate, amplitude, point values) in an extensible format.

The JSON data format was designed around two layers of information. The base layer stores properties common to all entities: name, type identifier, position, size, velocity, colour, and physics flags (static, gravity, trigger). A second layer stores type-specific properties (patrol speed, fire rate, oscillation amplitude, point values) as a generic key-value map within each entity record. This two-layer approach means new entity types can define their own serialisable properties without changing the JSON schema or the level loader. The design also requires that entity subclasses implement their own serialisation methods, keeping the level loader decoupled from specific entity types.

Figure 6: UML Class Diagram of the Entity Subsystem. Illustrating the inheritance hierarchy of game entities and their relationship with the EntityManager.

4. Implementation and Testing

4.1 Technology Choices

Based on the design in Section 3, the system was implemented in C++17 using SFML for windowing and rendering, Dear ImGui (via ImGui-SFML) for the editor interface, RapidJSON for JSON level serialisation, and CMake for cross-platform builds on macOS and Windows. The rationale for each of these selections is discussed in the literature review (see Section 2).

4.2 Engine

The game loop implements a fixed timestep of 60 Hz following Fiedler's (2004) accumulator methodology. Each frame, real elapsed time is added to an accumulator, and physics steps execute at exactly 1/60th of a second until the accumulator is drained. A ceiling cap of 0.25 seconds prevents the "spiral of death" where a lag spike causes runaway catch-up computation. The editor's play mode uses the same accumulator loop, ensuring identical gameplay in both contexts.

Collision uses AABB with minimum-penetration-axis resolution (Ericson, 2005). Initially, collision checking was brute-force O(n squared), which worked with a small number of entities but caused performance issues at higher counts (see Test 19 in Section 4.4). A uniform grid spatial partitioning system was implemented to address this, subdividing the world into 128x128 pixel cells and only testing entities that share a cell. In testing (see Test 19 in Section 4.4), brute-force collision caused visible frame-time spikes at approximately 150 entities, while the spatial grid maintained a stable 60 Hz update rate, as observed through the engine's frame-rate counter, with over 300 entities in the scene. Building the simpler version first made it possible to verify that the resolution logic was correct before adding the complexity of spatial indexing.

Physics uses Euler integration with an exponential friction decay model (velocity *= exp(-friction * dt)), which is frame-rate-independent and never overshoots zero. Ground friction is ten times stronger than air friction, producing responsive ground movement and floaty air control. The entity manager uses a deferred add/remove pattern, where changes during the update loop are queued and applied once the loop completes, preventing iterator invalidation.

4.3 Editor and Gameplay

The editor is the primary way a user interacts with the engine. It provides a scene panel listing all entities with type icons and search filtering, a live game viewport in the centre, a properties panel for editing the selected entity, and a palette with predefined templates for quick entity creation, as seen in figure 7.

Embedding a live game view inside the ImGui interface required a render-to-texture approach. The engine renders to an sf::RenderTexture, which ImGui displays as an image. An invisible button widget captures mouse input on top of the image, enabling click-to-select, drag-to-move, and multi-select. The viewport supports panning, zooming (0.25x to 4x), and grid snapping at configurable sizes.

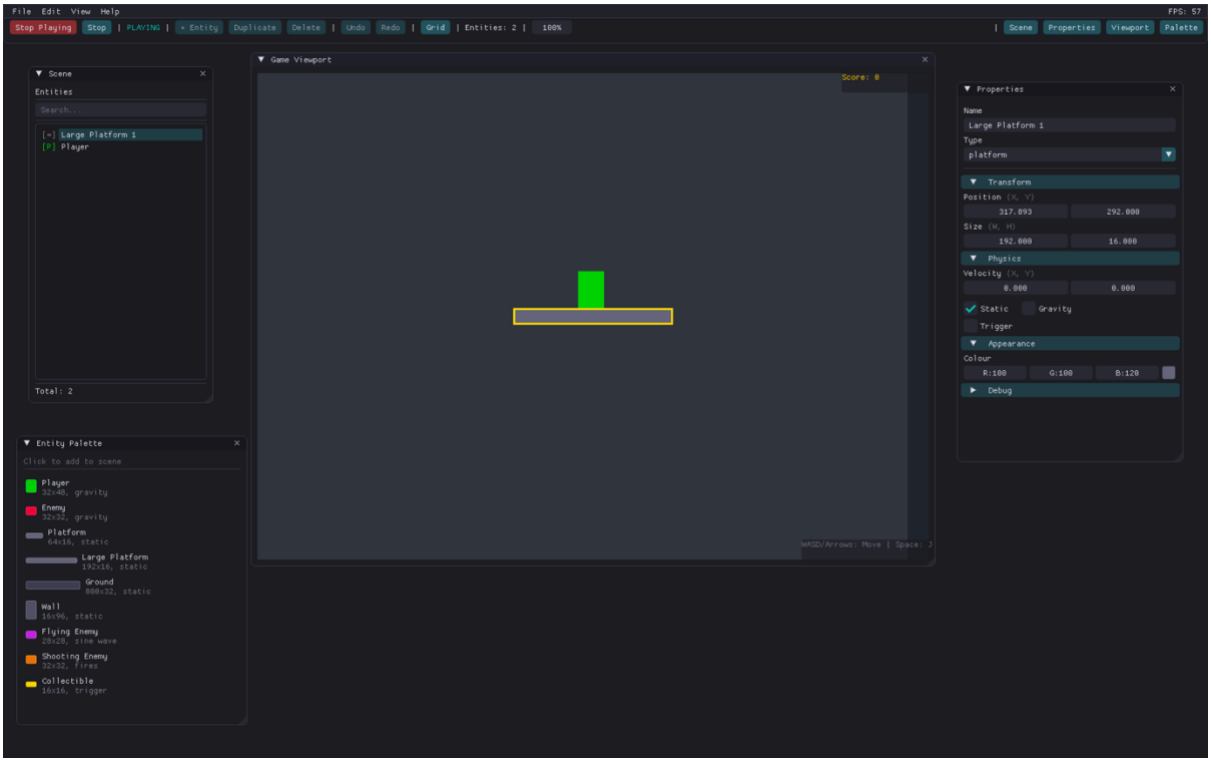

Figure 7: Screenshot of the Game Engine with a demo level open, and an entity selected with its properties showing

The undo/redo system follows the command pattern described in Section 3.2. Every edit creates a Command object, and batch operations such as multi-entity deletion are grouped into a CompoundCommand for single-step undo. The entity manager's detachEntity() method removes an entity without deleting it, handing ownership to the command for later restoration. The editor also tracks unsaved changes with a dirty flag, and play/stop controls snapshot and restore all entity state so that playtesting is non-destructive.

The player moves at 250 px/s with a jump force of -450 px/s. Three enemy types were implemented: patrol enemies that walk back and forth, flying enemies following a path with per-entity phase offsets, and shooting enemies that fire projectiles at regular intervals. Contact with any enemy respawns the player. Collectibles increment the score on overlap and are detached rather than deleted to preserve the undo history. Each entity subclass handles its own behaviour through polymorphic update() methods and persists its specific properties through JSON serialisation. An EntityFactory maps type strings to constructors so that the level loader, palette, and duplicate functions all create the correct subclass. This factory pattern was not part of the original design; initially, entity creation used type-string checks scattered across the editor. The refactoring to a centralised factory was one of the more significant improvements made during development. An audio manager wraps SFML's audio module for sound effects, with conditional compilation for systems without audio support. Four demo levels validate different engine features (figure 8).

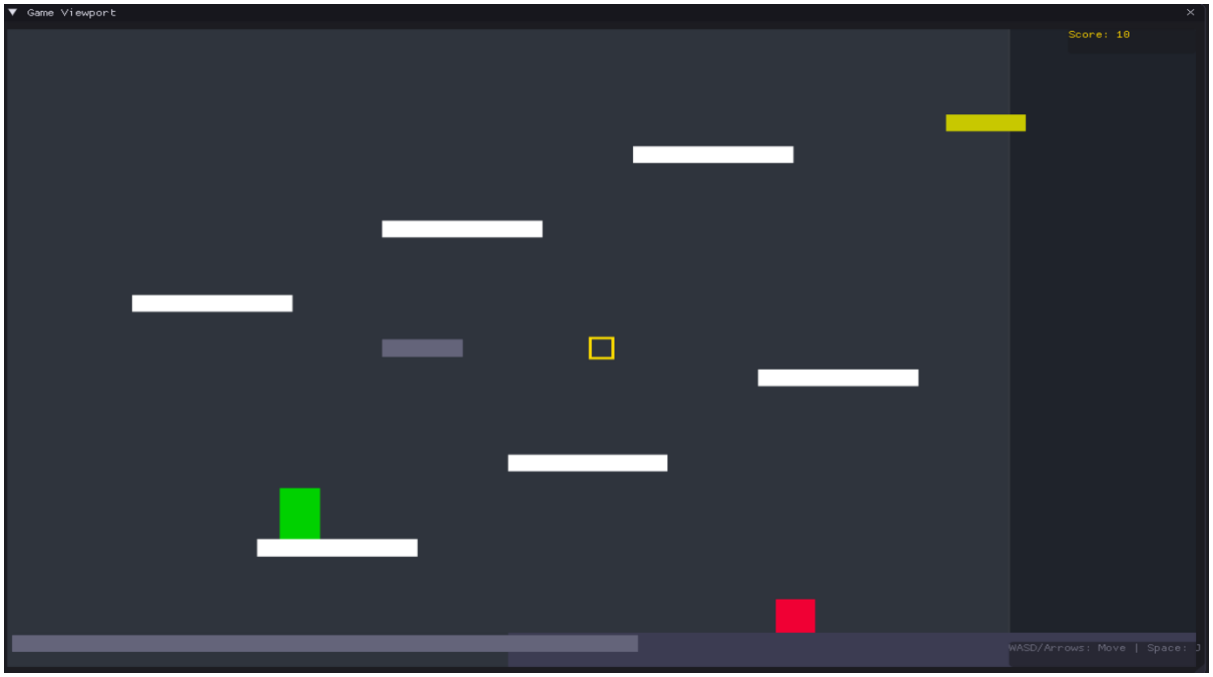

Figure 8: A demo level showing the player (green) an enemy (red) and a collectible (yellow) and the score, from interacting with the collectible (top right corner)

4.4 Testing

To evaluate the success of the project, a structured manual test plan was produced covering functional, non-functional, positive, and negative execution paths. Each test case is linked to a specific area of the system for traceability. The following table presents the results. Where a test initially failed, the issue was fixed and the test re-run. Re-test rows are shown with a shaded background.

Of the 24 tests executed, 19 passed on the first attempt. The 5 failures revealed genuine issues: missing JSON type validation, uncapped terminal velocity, individual rather than compound undo for batch operations, incomplete dirty flag coverage, and synchronised enemy phases. Each issue was fixed and re-tested successfully. Collectively, the 24 tests provide coverage across all six objectives stated in Section 1.1.

The testing process was valuable beyond simply catching bugs. The performance test (ID 19) validated the spatial partitioning implementation under load. The compound undo test (ID 22) improved the multi-entity workflow considerably. The testing also directly influenced the implementation: when brute-force collision was first tested at scale (during early development, before the formal test plan), the performance issues led directly to implementing spatial partitioning, which became one of the project's strongest technical features.

ID	Area	Description	Expected Output	Result
1	Level I/O	Load valid JSON level file.	All entities render in viewport and Scene Panel with correct names.	**Pass**
2	Level I/O	File > Save As with a new filename.	JSON file created. Window title updates. Dirty flag clears.	**Pass**
3	Level I/O	Save a level, close and reopen it.	All entity properties match exactly between original and reloaded.	**Pass**
4	Level I/O	Open JSON with {"entities": []}.	Level loads with empty viewport and Scene Panel. No crash.	**Pass**
5	Entities	Add a Player from the Entity Palette.	Player appears in viewport. Scene Panel updates. Properties show defaults.	**Pass**
6	Entities	Right-click entity in Scene Panel, click Delete.	Entity removed. Ctrl+Z restores it with all properties.	**Pass**
7	Properties	Change an Enemy's type dropdown to Player.	Size, colour, and physics flags update to new type defaults.	**Pass**
8	Physics	Drop a Player onto a platform (play mode).	Player falls, lands on surface, stops. isOnGround = Yes.	**Pass**
9	Physics	Move Player horizontally, release input mid-air, land.	Slow deceleration in air (low friction), rapid on ground (high friction).	**Pass**
10	Undo/Redo	Select a platform, drag it +50px, press Ctrl+Z.	Platform returns to its exact original position.	**Pass**
11	Selection	Click overlapping Player and Platform.	Player (drawn on top) is selected. Properties Panel shows Player.	**Pass**
12	Viewport	Scroll zoom in and out to extremes.	Zoom clamps between 25% and 400%. No artefacts.	**Pass**
13	Viewport	Press G while in edit mode.	Grid overlay toggles. Snap activates/deactivates.	**Pass**
14	Gameplay	Run Player through a Collectible (play mode).	Collectible disappears. Score increments. Player not blocked.	**Pass**
15	Gameplay	Run Player off the bottom of the level.	Player respawns at starting position. Velocity resets. Score preserved.	**Pass**
16	Usability	Type "Coin" into Scene Panel search with 50+ entities.	List filters to matching names. X button clears the filter.	**Pass**
17	Usability	Right-click Scene Panel during play mode.	Duplicate, Delete, Move Up, Move Down are greyed out.	**Pass**
18	Audio	Build without SFML audio support. Run the application.	Application runs normally. No crashes. All non-audio features work.	**Pass**
19	Performance	Duplicate entities to 250+. Press Play.	Editor and simulation remain responsive.	**Pass**
20	Level I/O	Load JSON with malformed values (string where float expected).	Error in status bar. No crash.	**Fail**
20a	Level I/O	Fix: Added type guards (IsNumber, IsBool) to all JSON reads. Malformed values use entity defaults.	Re-test: Malformed JSON loads without crash. Invalid fields fall back to defaults.	**Pass**
21	Physics	Position Player at Y: -5000, press Play.	Fall speed caps at 800 px/s.	**Fail**
21a	Physics	Fix: Gravity was not clamping after accumulation. Added velocity.y = min(velocity.y, MAX_FALL).	Re-test: Velocity caps correctly at 800 px/s regardless of fall distance.	**Pass**
22	Undo/Redo	Ctrl+Click 5 entities, drag +100px, Ctrl+Z, Ctrl+Shift+Z.	All 5 return to offset simultaneously.	**Fail**
22a	Undo/Redo	Fix: Multi-entity drags created individual MoveCommands. Wrapped in CompoundCommand.	Re-test: Single Ctrl+Z reverts all 5 at once. Positions preserved.	**Pass**
23	Usability	Make an edit. Try to close the window.	"Unsaved Changes" modal with Save, Discard, Cancel.	**Fail**
23a	Usability	Fix: Dirty flag was not set by undo/redo or Edit menu actions. Added markDirty() to all paths.	Re-test: Modal triggers correctly for all edit types.	**Pass**
24	Gameplay	Place 3 Flying Enemies at same Y. Press Play.	Each oscillates with a different phase. Not in unison.	**Fail**
24a	Gameplay	Fix: All flying enemies shared the same sine phase. Added per-entity offsets (1.2 rad apart).	Re-test: Three enemies bob with distinct, desynchronised patterns.	**Pass**

5. Project Evaluation

5.1 Strengths

The three-layer architecture proved to be robust under iteration. Over the course of development, the editor layer underwent significant changes (addition of undo/redo, multi-select, entity palette, play/stop state management) while the engine layer required no modifications to accommodate them. This validated the design constraint that dependencies flow downward only. The engine compiles and runs with no ImGui dependencies, confirming that the separation of concerns described in Section 3.1 was maintained throughout development.

Implementing the fixed-timestep game loop correctly from the start saved considerable time later in development. Physics behaved consistently regardless of frame rate, and because the editor's play mode uses the same accumulator loop as the standalone engine, gameplay was identical in both contexts. This eliminated an entire category of bugs where behaviour might differ between editing and playtesting. The editor achieved a complete authoring workflow. A user can create levels from templates, snap entities to a grid, undo mistakes, playtest with camera tracking and audio feedback, and stop to revert everything back to its original state. This kind of edit-play-iterate loop is what makes commercial tools productive, and achieving it in a solo academic project was a meaningful result.

5.2 Limitations

Although the entity system supports a texturePath field and the renderer can draw sprites, the editor has no UI for assigning textures. All entities currently render as coloured rectangles, which limits the visual quality of the demo. A texture browser in the properties panel would address this and is noted as further work.

Entity behaviours such as patrol bounds and sine wave frequency are persisted through JSON, but the editor does not yet expose them as editable fields in the properties panel. Level bounds are hardcoded to 800x600 rather than being configurable per level. There are no win or lose conditions, and no level-to-level progression.

On a design level, the inheritance-based entity model served the editor's property inspector well, but scaling to a larger entity taxonomy would require a new subclass, serialisation methods, and factory registration for each type, whereas a component-based approach would allow new behaviours through composition alone. The testing strategy also relies entirely on manual execution, meaning there is no automated regression safety net for future refactoring.

5.3 Reflection

One of the most important lessons from this project was the complexity introduced by the undo/redo system. The command pattern itself was straightforward to implement, but it created memory lifecycle challenges that were not anticipated. Commands that hold pointers to entities become dangling if those entities are freed by a separate code path, such as loading a new level. The solution was to always clear the command history before clearing entities, and to use detachEntity() to transfer ownership to the command rather than deleting. This required careful auditing of every code path that

touches entity lifetime. In future work, replacing raw pointers with unique entity IDs would avoid this class of problem entirely.

Another important lesson was the value of separating gameplay logic from the editor. Initially, player input handling, enemy AI, and collision responses were implemented directly in the editor's update function using type-string checks. This worked but became difficult to maintain. Refactoring this logic into polymorphic update() methods on the entity subclasses, with an EntityFactory to create the correct types, made the codebase significantly cleaner and easier to extend.

The structured test plan was more useful than initially expected. The compound undo test revealed a genuine usability problem with multi-entity editing, and the performance testing during development directly led to implementing spatial partitioning. Producing the test plan earlier would have been beneficial, as it could have guided implementation rather than only validating it at the end.

Beyond the technical lessons, the project also highlighted the importance of planning and iterative design. The initial approach was to build each system fully before moving to the next, but in practice, earlier systems frequently needed revisiting once later systems revealed new requirements. The EntityFactory refactoring is one example; another is the editor's play/stop system, which was originally a simple state toggle but had to be significantly reworked once undo/redo and collectible detachment were added. This experience reinforced the value of designing systems to be revisitable rather than treating the first implementation as final, and of accepting that some rework is a normal part of development rather than a sign of poor planning.

6. Further Work and Conclusion

6.1 Further Work

Two areas of further work have been identified. First, refactoring the entity model toward an Entity-Component-System (ECS) architecture would improve data cache locality and allow more flexible behaviour composition, which becomes increasingly important as the number of entity types grows (Nystrom, 2014; Harris, 2022). Second, expanding the editor's property panels to include a texture browser would fully integrate visual art assets into the authoring pipeline, replacing the current rendering.

6.2 Conclusion

The project achieved its stated aims (see Section 1.1). The engine provides deterministic fixed-timestep physics, spatial partitioning for collision, and a clean three-layer architecture with one-directional dependencies. The editor provides a complete level authoring workflow with undo/redo, multi-select, grid snapping, non-destructive playtesting, and JSON serialisation with subclass property persistence. A playable platformer with multiple enemy types, collectibles, camera tracking, and audio runs across four demo levels.

The structured test plan confirmed correctness and, where tests failed, directly guided improvements that strengthened the final system. All design decisions were grounded in established literature, and while there is clear scope for further development, particularly around visual assets, configurable level bounds, and scripting, the project demonstrates a thorough understanding of the principles that underpin modern game development tools.

References / Bibliography

Blow, J. (2004) 'Game Development: Harder Than It Looks', Queue, 1(10), pp. 28-37.

Cornut, O. (2024) Dear ImGui. [Online] Available from: https://github.com/ocornut/imgui [Accessed 15 November 2025].

Ericson, C. (2005) Real-Time Collision Detection. San Francisco: Morgan Kaufmann.

Fabian, R. (2018) Data-Oriented Design. [e-book] Available from: http://www.dataorienteddesign.com/dodbook/ [Accessed 15 November 2025].

Fiedler, G. (2004) Fix Your Timestep! [Online] Available from: https://gafferongames.com/post/fix_your_timestep/ [Accessed 25 November 2025].

Freitas, L. G. de et al. (2012) Gear2D: an extensible component-based game engine. Proceedings of FDG 2012. pp. 81-88.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) Design Patterns. Boston: Addison-Wesley.

Gregory, J. (2018) Game Engine Architecture. 3rd edn. Boca Raton: CRC Press.

Hall, M. (2014) Entity Component System Game Engine Design. [Online].

Harris, S. M. (2022) Implementation and Analysis of the ECS Architecture. Undergraduate Thesis. University of Portsmouth.

Horntvedt, M. and Akesson, A. (2019) Java, Python and Javascript, a comparison. [Online].

Kelly, C. (2012) Programming 2D Games. Boca Raton: CRC Press.

Masiukiewicz, D., Masiukiewicz, D. and Smolka, J. (2019) Research of an ECS architectural pattern. Journal of Computer Sciences Institute. 13, pp. 349-353.

McShaffry, M. and Graham, D. (2013) Game Coding Complete. 4th edn. Boston: Cengage.

Meyers, S. (2014) Effective Modern C++. Sebastopol: O'Reilly Media.

Mileff, P. (2023) Game loop: the heart of the game engine. Production Systems and Information Engineering. 11(3), pp. 45-56.

Millington, I. (2010) Game Physics Engine Development. 2nd edn. Burlington: Morgan Kaufmann.

Nystrom, R. (2014) Game Programming Patterns. New York: Genever Benning.

Savchenko, A. (2000) Dragonfly: a simple game engine for education. Worcester: WPI.

SFML Development Team (2018) SFML 2.5.1 Documentation. [Online] Available from: https://www.sfml-dev.org/documentation/2.5.1/ [Accessed 2 January 2026].

Shirley, P. and Ashikhmin, M. (2020) Fundamentals of Computer Graphics. 5th edn. CRC Press.

Stroustrup, B. (2013) The C++ Programming Language. 4th edn. New York: Pearson.

Stroustrup, B. (2018) A Tour of C++. 2nd edn. Boston: Addison-Wesley.

Tan, W., Go, K. and Lee, W. (2024) Constructing a game engine: a proposed course. Entertainment Computing. 48.

Tencent (2016) RapidJSON. [Online] Available from: https://rapidjson.org/ [Accessed 12 January 2026].

White, W. (2020) Data-Driven Design. [Lecture] CS4152, Cornell University.