KillSwitch: A Case for Custom Tech — Developing a High-Performance Vulkan Engine and Ecosystem from First Principles



Fig. 1. *Hell Mist*, a survival-horror rogue-like developed in the final 5 days of the 12-week cycle. The title features procedurally generated environments, dual-layer skeletal animations, and PVE gameplay, demonstrating the high development speed enabled by the engine's decoupled ECS architecture and automated Vulkan render-graph. Rigorous testing across heterogeneous hardware validates the stability of our bespoke ecosystem and its capacity for rapid, scalable game development.

In this report, we present *KillSwitch*: a high-performance game engine, with an automated render-graph, GJK-based physics solver, and dual-layer skeletal animation system, developed from first principles over a twelve-week development period. Built using the explicit Vulkan 1.4 API, *KillSwitch* is designed to maximize throughput on heterogeneous hardware, specifically targeting the iGPU gap, where monolithic engines often falter. Our work demonstrates that an engine can be rapidly developed by using a project's creative vision to focus requirements, resulting in a lightweight footprint that avoids the overhead of general-purpose frameworks. This is achieved through a modular, decoupled architecture that ensures individual systems remain extensible. By prioritising high affinity with the hardware over broad abstractions, *KillSwitch* proves that custom tech provides a superior balance of creative autonomy and long-term scalability.

CCS Concepts: • **Computing methodologies** → *Computer graphics*; • **Graphics systems and interfaces**;

Additional Key Words and Phrases: Game Engine Architecture, Frame Graphs, Vulkan API, Real Time Physics, Animation Systems, Systems Engineering.

Author's address:

1 INTRODUCTION

The modern game development landscape is characterised by an industry-wide convergence on a few monolithic engines. These frameworks represent decades of high-level engineering, and their necessity is often justified by the perceived insurmountable cost of developing custom tech for both independent and AAA studios [Purslow 2022; Rivela 2019]. However, the pursuit of a universal solution necessitates a high degree of abstraction that frequently leads to overgeneralised architecture. In practice, this manifests as an accumulation of high overhead abstractions, as the engine must maintain backwards compatibility across a sprawling feature set. This necessitates deep layers of indirection and complex state-tracking that precludes any ability to reason about the underlying hardware when programming high-speed paths later in development. The modern workaround for commercial frameworks is often to use strong industry ties to mitigate these fundamental efficiency losses

by integrating vendor-locked temporal upscaling and frame generation. These technologies mask some of the performance deficit for high-end users, but they fail to address the underlying resource contention on constrained hardware, such as integrated GPUs, where the cost of generalisation becomes a prohibitive barrier to entry. This brings us to our current situation where general-purpose engines often fail to maintain, or even reach, stable, high-frame rate performance on low-power devices [Sobchyshak et al. 2026], frequently dropping below the 60 FPS threshold for responsive interaction. Despite advancements in low-latency display technologies, even 60Hz can feel too little in modern games due to cumulative latency from complex, multi-stage graphics pipelines, temporal anti-aliasing, checkerboard rendering, triple buffering [Raaen and Petlund 2015], mouse smoothing, and desktop compositors [Dossena and Trentini 2022].

Beyond technical performance, engine choice has an often-overlooked implication that becomes entirely obvious in the final product: Because game development almost universally follows rapid, iterative cycles, a project's creative vision is inevitably shaped by the paradigms imposed by the underlying engine. This interplay between engine architecture and creative constraint can be understood through the lens of a tool's "affordance" from the field of design-engineering [Ciavola and Gershenson 2016]. Consequently, the affordance of monolithic frameworks used by thousands of titles do more than just support the creative process; they actively shape it, incentivising developers to align their design goals with the engine's pre-existing quirks and architectural biases. On the other hand, a custom engine provides the opportunity for technical and creative alignment, where the architecture is built specifically to facilitate the project's unique vision. A bespoke system can discard the overhead of substantial middleware, and prioritise high affinity hardware paths with lean, specialised workflows. As Acton [Acton 2017] found from leading the engineering behind an array of AAA titles, knowing the hardware and knowing the requirements is most important for solving the actual problem. Critically, this context is missing from general-purpose solutions.

To contextualize the architectural decisions behind *KillSwitch*, the following section reviews the technical precedents in high-performance software and explicit graphics APIs that inform this project's approach to rapidly producing a modern, scalable, extensible engine.

2 BACKGROUND RESEARCH

2.1 Seeking a Sustainable System Architecture

From the outset we knew we wanted to make a third-person horror-inspired game with beautiful rendering. Programming an entire engine and game from scratch required high development speed. This made it essential to set up a scalable overall architecture, so we could avoid being slowed down by technical debt, which can accumulate fast.

The evolution of game engine architecture has seen a shift from traditional Object-Oriented Programming (OOP) class hierarchies to modular, component-based, and data-driven designs. This transition is largely a response to the architectural decay that occurs in large-scale systems, where high inter-file connectivity propagates error-proneness and increases maintenance costs. A critical challenge in engine development is managing the accumulation of structural artifacts that cause a system to deviate from its intended design over time. To prevent this, architects often employ design rules, manifested as stable interfaces that decouple the rest of the system into independent modules [Mo et al. 2021].

In KillSwitch, this is addressed through a public/internal split for each module, a design pattern that prevents internal code being reachable from the exposed API. By ensuring that public-facing headers contain only pure C/C++ types and opaque handles, the architecture prevents the leak of low-level dependencies (such as Vulkan or platform-specific libraries) from polluting the global namespace. To ensure other team members can figure out how to use another member's code fast, it is important to minimise the number of functions and types an exposed header file for a module. Failling to do this, especially in a fast-paced, iterative development environment, will lead to only a few of the exposed features being over-used whilst the rest go underused. It also prevents team members unknowingly implementing the same thing twice, which is common when the API is too overwhelming to keep even vaguely in working memory.

The specific inspiration for our modular, minimal APIs is SDL (the Simple Direct Media Layer), which exposes a straightforward procedural C interface instead of providing users a deeply interconnected object graph of state and methods. C interfaces are also widely used due to the stability and ubiquity of the C ABI across platforms and toolchains. On the other hand, C++'s ABI is less stable due to features such as name mangling and higher-level abstractions that differ across compilers. Thus the C ABI serves as a de facto lingua franca for dynamic libraries and language bindings.

Object-Oriented architectures prioritise encapsulation through deep class hierarchies, this approach often introduces architectural rigidity and fragmented memory layouts that are suboptimal for high-throughput systems. By decoupling state (components) from logic (systems), the engine transitions from a model of individual object-persistence to one of stream-processing. This alignment with Data-Oriented Design (DOD) not only yields superior cache-line utilization but also facilitates fine-grained parallelism by minimizing data-dependency contention between threads [Ambrosio et al. 2025].

2.1.1 Deterministic Lifetimes via Region-Based Memory. A final pillar of the *KillSwitch* architecture is the rejection of granular, heap-based allocation in favour of region-based memory management, commonly referred to as *Arena Allocation*. In modern game engine development, the frequent invocation of general-purpose allocators, such as `malloc`, `new`, or `VirtualAlloc`/`mmap`, introduces significant architectural friction. These calls not only suffer from the performance penalties of page faults and heap fragmentation but also necessitate complex and often fragmented ownership strategies for their management.

The industry-standard response to this complexity is the widespread adoption of reference counting and move semantics. Which whilst automating the cleanup of small allocations, introduces extra cognitive load and run-time overhead throughout throughout the entire codebase, often resulting in non-deterministic deallocation

spikes and opaque object lifetimes. Furthermore, they fail to address the underlying hardware-level inefficiency of the general-purpose allocator itself.

In contrast, we use the arena paradigm, wherein memory is pre-allocated in large contiguous blocks. These regions are partitioned based on shared lifetimes, typically categorized as either *per-session* or *per-frame* scratch-pad memory. This approach offers three distinct advantages for high-performance engine programming:

- **O(1) Allocation and Deallocation:** Allocating from an arena is a simple pointer-increment, and freeing an entire region, such as clearing a frame's worth of transient draw calls, is a single reset operation, entirely bypassing the overhead of destructors and the C-runtime.
- **Locality of Reference:** By grouping data with similar lifetimes together in physical memory, the engine maximises cache-line utilisation and reduces Translation Lookaside Buffer (TLB) misses.
- **Architectural Simplification:** A centralised struct of global state and another per-thread state avoids the dependency hell of passing deep pointers through every function signature, a design philosophy historically validated by high-performance codebases such as the *Quake* engine [id Software 1999] and the work of Muratori [Muratori 2004] and Fleury [Fleury 2022].

This model ensures that memory management is more structured, and remains a first-class citizen of the engine's design rules, ensuring that performance remains predictable even as the complexity of the simulation and rendering increases.

2.2 Scalable Rendering Architectures for Explicit Graphics APIs

While modern explicit graphics APIs such as Vulkan are often framed as low-level, it is more accurate to describe them as driver-centric. By shifting the burden of state-tracking and synchronization from the driver to the application, manufacturers reduce driver overhead and ambiguity. However, this often results in an API design that prioritizes hardware-interop specification over developer ergonomics. We decided our games would need to run on a multitude of hardware, with access to modern features, which leads to choosing between Vulkan and Direct3D 12. Direct3D is limited to Microsoft platforms, so Vulkan was the clear winner for our use case, supporting both Windows, Linux and MacOS (through the MoltenVK compatibility layer). Some aspects of Vulkan and Direct3D 12, such as Vulkan's Pipeline State Objects (PSOs), are highly convoluted and tightly coupled with many other objects. Part of this is a generality tax paid when designing an API for heterogeneous hardware. In contrast, fully integrated solutions like Apple's Metal benefit from a narrower hardware target, allowing for abstractions that align more closely with a specific Instruction Set Architecture. This reduced generality results in an API that is simultaneously more expressive for the programmer and more efficient for the hardware.

These explicit APIs make possible many new rendering architectures with lower-overhead than ever before, but also burden the programmer with deeply complex design considerations at the get-go. There are, however, numerous ways to take that burden and use it to produce something that maximises both productivity and low-level control. The developers of SDL came up with one approach in their SDL3 GPU API, which offers a thin layer over Vulkan, Metal and Direct3D 12 [Simple DirectMedia Layer 2025] to give a far more elegant interface for the programmer. Such layers, however, are intentionally limited to a lowest common denominator feature set to remain cross-platform and clean to work with. This often entails omitting or lagging behind vendor-specific extensions and latest-spec features that are essential for cutting-edge custom engine development.

For smaller-scale or solo projects, SDL3-GPU represents a highly attractive middle ground; its procedural design is architecturally sound and far more representative of modern GPU pipelines than legacy frameworks like OpenGL. Whilst OpenGL 4.5 remains accessible, its state-machine architecture is a relic of the fixed-function pipelines of the 1990s, making it increasingly difficult to map to the asynchronous, compute-heavy nature of modern graphics hardware.

To avoid encountering any limitations with SDL or older APIs, we stayed with Vulkan for our project. The API has also improved a lot for the application programming since its release, with features like dynamic rendering, GPU pointers, partially bound descriptors, and non-uniform image array indexing. Vulkan 1.4 specifically, from 2024, is brand-new but well supported, and we choose it to experimentation with architectural patterns that were not possible before. These features significantly simplify GPU resource management compared to earlier Vulkan workflows, reducing descriptor management overhead and enabling new styles of framegraph design; shown in Section 3.

Our renderer needed to support multi-pass materials (outlined characters), many geometry passes for shadows, and chained post-processing. Supporting these features over the lifetime of a project requires a rendering architecture that remains programmable and extensible, allowing evolving artistic requirements to be integrated without building a monolithic renderer tied to the requirements of a single game. Programming render-pass code directly on top of Vulkan can work for isolated research, but does not provide the scalability or extensibility needed for a production engine which may be actively updated and maintained for many years across multiple titles. As resource counts and pass complexity accumulate, integrating further passes, optimising execution order, or adding features that span all your render-passes or pipeline objects, is a productivity killer. Finding the cause of a validation layer error or GPU hang within a tightly coupled web of Vulkan code is a rite-of-passage monstrous time-sink that a more loosely coupled system is designed to mitigate. Engine developers routinely run into these problems, so it is no surprise that industry standard architectural solutions have emerged that do not trade performance for simplicity, in fact, they are closer to being the "intended usage" of explicit APIs to maximize hardware occupancy in non-trivial projects (although no rendering project is really trivial these days [Aaltonen 2025]).

2.2.1 *The Combinatorial Explosion of Pipeline State.*

The primary architectural challenge in Vulkan is the Pipeline State Object (PSO). Unlike legacy APIs (OpenGL) where state is a mutable global machine, Vulkan PSOs are immutable and monolithic, encapsulating

everything from shader stages and vertex layouts to blend modes and depth-stencil tests. While this allows the driver to pre-compile and optimise the full state up-front, it introduces two critical problems for the engine architect:

1. State Duplication: Minor variations in a material (e.g., toggling a stencil test or changing a blend mode) require an entirely new, distinct PSO.

2. Combinatorial Explosion: In a multi-pass renderer, the number of potential PSO permutations across different shaders, vertex formats, and render-target configurations quickly becomes unmanageable for manual tracking.

2.2.2 Functional FrameGraph Architectures. To manage the complexity of multi-pass execution, modern engines have shifted toward a functional, graph-based abstraction. Pioneered by Yuriy O'Donnell for the Frostbite engine, the FrameGraph (or render-graph) treats the frame not as a sequence of imperative commands, but as a directed acyclic graph (DAG) of dependencies [O'Donnell 2017].

In this paradigm, individual render-passes are decoupled from resource management. By declaring inputs and outputs for each pass, the engine can algorithmically deduce resource lifetimes and insert the necessary synchronisation barriers. This abstraction is critical for explicit APIs, as it allows the engine to optimize the global execution of render-passes, performing tasks such as automated barrier batching, without requiring the developer to manually reason about transient sub-resource states. Then, algorithms like topological sort can be applied to optimise execution, and render-targets with no dependencies between each other can be secretly reused via resource aliasing (different resources backed by the same buffer) which reduces the memory footprint.

There are two main variations of a framegraph architecture, static and dynamic. The static graph is compiled once (or occasionally, e.g. scene change) and can spend a lot of computation optimising execution and memory footprint. Static graphs are usually responsible for resource allocation, which they can do optimally. Some implementations even heavily borrow ideas from compilers, by optimising the graph in Single Static Assignment (SSA) form, introducing LLVM-esque concepts like phi-nodes, which ends up algorithmically reinventing the concept of ping-pong buffers [Dolp 2025].

The alternative, the dynamic framegraph, is compiled once per frame, incredibly cheaply. At most, for in depth implementations, it would perform simple sorting and analysis of the dependency graph, which is incredibly cheap, unless your framegraph is gigantic, in which case, good luck rendering it in real-time, optimised or not. The motivation for this form of graph, is that it solves all the architectural problems of an unmanaged Vulkan renderer, decouples your individual render-passes, and automates synchronisation in bulk.

The dynamic framegraph was chosen for this project, as even the most basic implementation is a workflow multiplier, which pays dividends for our game, which needed a highly stylised, renderer with a multiple geometry passes and chained post-processing. As well, at any point in development, multiqueue dispatch and topological sorting can be added to the underlying framegraph implementation without changing any of the renderpass code.

2.2.3 Decoupling State via Pipeline Hashing. While the framegraph manages the high-level flow of data, the combinatorial explosion of PSOs mentioned previously requires a specialised management strategy at the draw-call level. Industry-standard practice dictates a shift away from static, front-loaded PSO creation in favor of a cached pipeline state model.

Rather than hard-coding every possible pipeline variation, engines utilise a descriptor-based hashing system. This involves creating a bit-packed representation of the current state, including vertex input layouts, rasterization settings, and shader handles. Then, using a non-cryptographic hash function to generate a unique identifier. This identifier allows the engine to treat the PSO as a transient, lazily-instantiated resource. By checking this hash against a global cache, the renderer can automate the mismatch between a project's dynamic material requirements and Vulkan's requirement for immutable, pre-compiled state.

This architectural pattern is not merely a convenience; it is a prerequisite for state-sorting. By organizing the frame's draw calls by their pipeline hash, an engine can minimise expensive driver state changes, which is a vital optimization for maximizing throughput on constrained hardware architectures like integrated GPUs.

2.3 Data-Driven Asset and Prefab Architectures

Modern game engines mostly rely on data-driven workflows in which scene data is separated from runtime behaviour. Instead of hard-coding gameplay objects directly in source code, engines usually serialize object hierarchies, rendering data, and gameplay metadata into asset formats which can then be used to construct runtime entities. This architecture is available to developers through the concept of "prefabs", reusable scene objects that can be instantiated multiple times throughout a project.

While the term "prefab" is used by modern game engines such as Unity, the concept isn't new and originates from the broader need to decouple content creation pipelines from runtime systems. Gregory [Gregory 2018] describes this as the separation between authored asset data and live simulation objects, allowing designers to iterate on gameplay structures without directly modifying engine code. This workflow is especially important in larger scale projects, where rebuilding gameplay objects manually introduces maintenance overhead and slows down content production.

Data-driven instantiation pipelines are very important in ECS architectures. Gameplay objects are typically reconstructed dynamically by attaching components based on metadata in that asset rather than their object types. This allows gameplay building to remain flexible throughout development.

glTF is an asset format designed for efficient transmission and runtime loading of 3D scenes and models based on JSON [Khronos 2024]. It encodes geometry, materials, textures, skeletal structures, and animation data in a structured format that is both compact and easy to parse. In this engine, glTF provides the foundation for our asset system with its scene graph representation and support for embedding custom metadata, making conversion into ECS entities during instantiation simple.

3 METHOD

3.1 Renderer

Here we describe a dynamic framegraph architecture that combines a simple resource registry with modern Vulkan features to provide shaders easy, efficient access to data, whilst synchronising read-write targets automatically. Afterwards, we describe our drawcall system for pipeline hashing and multi-pass materials. Finally we outline how our clustered forward renderer was implemented on top of the framegraph, along with the how and why behind CPU-side light assignment, and a metric to saturate a fixed shadow-map budget with relevant spotlights.

3.1.1 Context. It is important to first understand where a renderer fits into an engine. All other systems of a game, like input, simulation, and audio, are happy to interact with each-other without caring about how they are rendered except for enabling various material properties and global rendering parameters. From this perspective, the renderer only needs to provide the rest of the program a very simple interface to convert the internal game world into data for the renderer.

To enable this loose coupling of the renderer with the gameplay code, we treat the renderer as a stateless machine that takes renderables and lights via PushRenderable() and PushLight() commands, and camera information passed to the DrawFrame() function at the end of the main loop.

Because games are dynamic, this paradigm makes sense, and allows things to change freely between frames, but there is inevitably a lot of data, mainly vertex buffers and textures, that are not only static, but also infeasible to pass through to the renderer each frame. This leads to adding one more phase to the renderer, that being an asset loading phase which happens once on scene change. This architecture can be optimised under the hood at any point for things like caching common resources between scenes, without changing gameplay code, which means can purposely push back even obvious optimisations until we actually encounter them as a bottleneck, which ensured we actually ended up with a product to showcase our engine with.

A single loading period on scene change is perfect for our types of games, since our worlds can fit into memory all at once (*HellMist* uses ~200MB of memory). This means we do not need to design around streaming data off disk, so there is no need for the overhead of designing around the allowing new assets or LODs to be uploaded during gameplay. By being precise around requirements in this way, the system can be simple and extensible. Trying to future proof the implementation for streaming resources is irrelevant, because even though they are similar systems in the abstract since they handle loading resources into VRAM, the efficient and sustainable implementation of both is entirely different, all the way down to how GPU buffers are uploaded to, so this is not a place to attempt to generalise.

3.1.2 The Resource Registry and Low-Level Abstraction. The client gameplay simulation pushes 'Renderables' to be drawn, which is nothing more than a transform attached to some metadata we call a "mesh prefab". The exposed mesh prefab is simply a material type along with opaque handles into a resource registry that is internal to the renderer, preventing low-level graphics state from leaking into the high-level game logic. To support skinned skeletal animations, the Renderable struct also incorporates a restricted pointer to an array of bone transformation matrices corresponding to the current skeletal pose. The renderer copies this data sequentially into a persistently mapped host-visible staging buffer during the frame data collection phase.

On the CPU side, a resource handle resolves to an explicit index within a centralised registry, which is implemented as a pre-allocated, contiguous array of uniform resource metadata structs. Within this architecture, a resource uniquely identifies either a VkBuffer or a VkImage. To optimise and simplify allocation tracking, resources are structurally aggregated into three catagories based on their bulk lifetimes:

(1) **On Startup:** Persistent resources instantiated at engine startup and held until application termination.
(2) **Window-Dependent Lifetimes:** Resources tied directly to the windowing context and viewport parameters, undergoing automated re-creation during swapchain resizing events.
(3) **Scene Dependent Lifetimes:** Resources allocated during scene initialisation. On scene change all of these are purged.

The framegraph execution does not concern these lifetimes, so the implementation of it is simplified by putting each lifetime into the same contiguous registry. It being contiguous simplifies how resources are managed by the user of the framegraph system even further. Figure 2 shows how these distinct lifetimes fit into the shared registry, without gaps being introduced into the registry.

Physical memory backing for these resources is managed via the AMD Vulkan Memory Allocator (VMA) library, with the centralized registry retaining the underlying allocation handles. External platform resources, specifically the native swapchain images managed by the presentation layer, work can also be imported into the registry, all the framegraph needs to do is be hands off on the destruction of these resources.

By organizing resources into sequential, contiguous lifetime arrays, the engine automates batch recreation and deallocation. This data-oriented layout eliminates the need for reference-counted smart pointers or runtime tracking overhead, yielding a highly parallelisable and predictable framework that scales linearly as the complexity of the global frame-graph grows.

3.1.3 The Dynamic Frame-Graph and Automated Synchronisation. The primary architectural justification for implementing a centralised resource registry is the virtualisation of GPU resource state tracking. By coupling state tracking directly with a resource's metadata, the engine eliminates manual, error-prone synchronisation scheduling. Utilising the nicer VK_KHR_synchronization2 extension, the registry encapsulates the exact state required for hardware hazard resolution: the current access mask (VkAccessFlags2), the execution pipeline stage (VkPipelineStageFlags2), the active queue family index (facilitating cross-queue family transfers and presentation handoffs), and the current image layout (VkImageLayout) reserved exclusively for image resources. This state metadata is evaluated continuously to inject optimal, automated vkCmdPipelineBarrier2 dependency commands during frame execution.

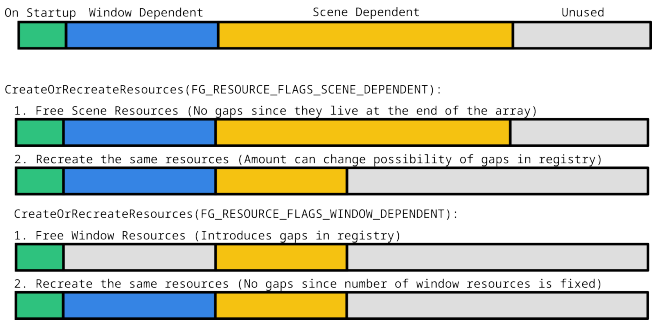

Fig. 2. A single array of resource metadata is all the framegraph cares about, but 3 resource lifetimes are used for GPU memory management. The figure shows how a scene change, followed by changing the window size does not introduce gaps in the registry. This means no fine grained resource management is required.

The underlying data structure of the frame-graph is a flat, contiguous array of discrete render-pass descriptors. Each pass declares its dependencies through explicit inputs (read/sampled resources) and outputs (render target attachments), although some outputs are read from too, specifically the depth and stencil buffers in depth and stencil testing. Both inputs and outputs are stored as arrays of resource usage structs. A resource usage states how a render-pass uses that resource, which is used to automate pipeline barriers. It contains the resource ID, paired with its target access mask and pipeline execution stage. Furthermore, the engine uses VK_KHR_dynamic_rendering natively, bypassing the rigid API associated with traditional VkRenderPass and VkFramebuffer creation. Consequently, an output resource usage directly embeds the execution parameters required for dynamic attachments, specifically the attachment load and store actions (VkAttachmentLoadOp, VkAttachmentStoreOp) and its potential clear colour value. MSAA resolves are also automated, where resource aliasing is used to automate the differences between MSAA on and MSAA off framegraphs.

By intentionally aligning the frame-graph structs with the native Vulkan data structures rather than imposing a heavy, idiomatic abstraction layer, the engine achieves a zero-cost abstraction. This design pattern ensures there is zero loss of hardware feature fidelity, maximizes data-cache locality during graph parsing, and provides a sustainable architectural foundation capable of supporting algorithmic graph compilation and topological execution sorting in the future.

3.1.4 Just-In-Time Pipeline State Caching via Bit-Packed Keys. To bridge the high-level, imperative execution of individual render-passes with Vulkan's requirement for immutable state configuration, the engine implements a Just-In-Time (JIT) pipeline caching architecture. Each render-pass descriptor encapsulates an execute– callback function pointer, allowing application-layer code to register arbitrary drawing commands dynamically. To mitigate the combinatorial explosion of hardware state changes at the draw-call boundary, the engine shifts away from heavy string or structure hashing in favour of a unique, 64-bit perfect bit-packed PipelineKey union.

The structure of this key explicitly packs both the structural intent and fixed-function state variables required by the underlying driver specification into an 8-byte primitive:

- **Architectural Core State:** Stores the pipeline type (pipeline_type : graphics, compute, or future ray-tracing pipelines), shader_id, and target frame-graph pass indexing (pass_idx).
- **Geometry and Vertex Layouts:** Dictates input assembly layout states (vertex_type), identifying structural variations across static geometries, skinned character models, or dynamic blend shapes.
- **Fixed-Function Hardware State:** Embeds immutable configuration fields directly into discrete bitfields, including depth-stencil state (depth_test , depth_write, depth_op, and stencil_mode), primitive rasterization state (cull_mode, polygon_mode, and winding order front_face), multisample anti-aliasing resolve properties (msaa_samples), and programmatic hardware blending flags (blend_mode).

During the scene-graph parsing phase, the engine constructs this 64-bit key for each draw command. The resulting draw-calls are subsequently sorted based on the scalar value of this key. This sorting mechanism ensures optimal batching, minimizing expensive hardware state-machine context switches and driver-level pipeline binds before any draws are pushed to the active command buffer.

During draw-call dispatch, the scalar key value is evaluated via the PK_GetOrCreatePipeline entry point. Because the key is pre-packed into a single uint64_t primitive, it bypasses the processing overhead and collision risks associated with traditional non-cryptographic hashing algorithms. The key serves as a direct lookup index within a flat, fast-path cache map structure (PipelineEntry). If a corresponding compiled VkPipeline handle exists within the registry, it is returned instantly; otherwise, the driver performs an on-the-fly lazy instantiation of the immutable state object, storing the result within the cache to eliminate subsequent pipeline compilation latency.

3.2 Physics

The engine's physics module is custom made from scratch. We chose to build one from the ground up instead of adopting an external library like Jolt Physics [Rouwe 2021] mainly because we estimated integrating one into our engine would take around the same time. By building our own physics engine, not only we get the benefits of learning the state-of-the-art algorithms, but it allows us to fully adapt it to the needs of any desired game. That is what happened in our case: we had to add custom mobility for `Character` rigid bodies in stairs and slopes because our game design included going up stairs to go to the next level.

3.2.1 Internal details. Our engine supports five types of rigid bodies, basic types being: static which are immovable, dynamic which are moved by external forces and kinematic which are exclusively moved by velocities; and two additional types: trigger for non-mesh static rigid bodies with different event firing, and character for a mix between a dynamic and kinematic body with additional logic for different contact resolution and movement logic.

The `PhysicsWorld` class contains all the data inside the physics (which can be accessed via specific `Handles`) and acts as the orchestrator and simulator of all the underlying systems. The RigidBodies

and Shapes (which are used by rigid bodies and queries) are stored in a Slot Pool, each Slot containing the data and a boolean that indicates if that index is occupied. The main reason of choosing a Pool over any other data structure was to have constant access, insertion and deletion but without moving the data, because underlying systems rely on references and pointers to the Pool's memory, and if its data were to change places, the pointers would have undetermined value.

The main function in the PhysicsWorld is a fixed-time step update at a 60 fps target, with the following structure:

- Apply external forces to our rigid bodies.
- Damp and integrate their velocities and positions using semi-implicit Euler.
- Detect possible collisions in a *Broadphase* check using the rigid bodies' AABBs (Axis-Aligned-Bounding-Boxes).
- Confirm those collisions and create contacts in a *Narrow-phase* check, using the Gilbert-Johnson-Keerthi algorithm [Gilbert et al. 1988], coupled with the Expanding Polytope Algorithm [Bergen 2001] (GJK+EPA).
- Resolve the contact using mass-proportional separation and impulse resolution, and apply extra logic for Characters.
- Emit Enter/Stay/Exit collision and trigger events.

The `PhysicsWorld` provides important queries that are important for other modules on the engine as well as game systems, which are the following three: ray-casting, shape-casting (using an adaptation of the Conservative Advancement algorithm [Zhang 2014]) and shape-intersecting.

3.2.2 External communication with the rest of engine. The physics engine was built decoupled from our ECS architecture to serve as an integrable physics library for any type of engine architecture. Due to this, a `PhysicsManager` class was added to translate between `Handles` and `EntityIDs`, and also to sync the entities' state in and out of the physics engine (e.g. the `Transform` component of the entities). The internal query formats from the Physics World had to also be translated to external queries. The manager was also in charge of emitting the Physics World's collision events that allow other systems to optionally subscribe to.

3.3 Animation

As with the physics module, the engine's skeletal animation system was engineered entirely from scratch, with the core deliverable being the ability to process raw glTF/glb data into a complete layered animation controller, aimed specifically at character animation.

The core evaluation loop, controlled by an `Animation_Update` function, starts by iterating through all entities with `C_AnimatedMesh` components and increments layer times, scaled by delta-time and a `PlaybackSpeed` multiplier. Keyframe windows are identified via a linear search of the glTF channel inputs and utilised by calculating an `InterpolationFactor` between them. This functionality then enables the parsed transforms of each keyframe to be interpolated with the same factor through use of `glm::mix` and `glm::slerp`, yielding the final per-bone transforms. To achieve seamless state transitions (e.g. from Walking to Running), layer-based blending is implemented. If a layer is blending, the system simultaneously samples the previous and current animations, utilising the same two `glm` functions and a clamped `blendTime` factor to obtain a resulting pose.

To support complex gameplay actions, such as reloading while moving, without requiring an exhaustive list of animation assets, the system features recursive bone masking. The `CreateUpperBodyLayer` function splits the hierarchy at a designated split joint (obtained by string-matching glTF node names), assigning a fractional mask of 0.5 at the split joint and 1.0 to all children of the split joint in a `boneMask` array. When blending between layers during the update loop, the stored `layerBlendDirection` smoothly increments or decrements the `boneMask` weighting between 0.0 and 1.0. The final pose is calculated by iterating through all joints and blending the obtained lower and upper body poses based on the combination of mask and weight, allowing gameplay programmers to easily perform any animation combination without snapping.

Finally, to limit the inconsistencies felt by the player, an additive rotation offset can be applied to supprt dynamic aiming directions during combat. The `SetAimingRotations` functions isolates a three-bone chain along the character's spine, and utilises input yaw and pitch values to hinge the spine to face the character in the camera direction along both x and y axes. To prevent any unnatural over-rotations or hinging, the yaw and pitch inputs are clamped within realistic human limitations, and the rotation is staggered across the three-bone chain to ensure proper distribution. These offsets are converted via `glm::angleAxis` and post-multiplied onto the existing quaternion rotations. Once these transforms are resolved, local matrices are calculated, followed by the model matrices, which are obtained by recursively traversing the bone hierarchy to compute the joint matrices, which are finally multiplied by the corresponding inverse bind matrices to generate the final vertex skinning palette for the renderer.

3.4 Entity Component System

Inspired in *seecs*, we opted for a lightweight Entity Component System with some RTTI elements for run-time type lookup. The basis of our ECS is to treat Entities as merely IDs or indexes, and Components as just data. Then Systems would require exact Component combinations and then operate on them.

Our implementation is based on a templated `SparseSet` class, which is a *Sparse* (meaning there can be empty spaces in-between active elements) paginated vector that points Entity IDs to a *Dense* array, in which all the data is stored in contiguous memory. As additional functionality, we provide another Dense vector that couples the dense index with its Entity ID. This way, we benefit from the performance of looping over a contiguous array in memory (cache hits) and from constant access, insertion and deletion of data.

The ECS is managed by the `ECS` class, which contains all the data and acts as a registry of all the different Components registered, one `SparseSet` each. The reason for using Sparse Sets is due to each entity being unique in terms of Components. The `ECS` class builds a bit-mask of every entity's registered Components that we can query at runtime to check if an entity has a Component. The class lets creating and deleting Entities, registering components, adding components under an Entity ID, removing them, getting the reference and pointers to data, and some additional functionality.

Also, to help developing the Systems, a templated `View` class parses the combination of Components provided via template and creates an array, looking up for each entity in the smallest SparseSet if the entity contains all of the desired Components (and excludes the non-desired). You can then use lambdas to loop over the data (optionally including the Entity ID) using the `View::ForEach` method. There's an alternative `View::GetPacked` method that packs that data into a vector and returns it so you can loop over it with a default *for-each*, unpacking the tupled Components.

3.5 Level Editor within Blender

A Blender plugin script was built to help importing assets with custom ECS data into our engine and also serve as a level editor. The plugin lets additional information to be added onto each Node's custom properties. Assets are then exported from (and imported into) blender using the .glb (or .gltf with .bin) format, and the Node's custom properties are laid out in a json, containing the format that we specified. The plugin adds custom buttons for export and import that add logic to translate custom values between blender and our coordinate system.

The custom properties that we can add to a Node are either the Components that we want to directly add to in the ECS (like a *RigidBodyComponent*), or combinations of components under a different name (like *Player* or *Zombie*), which are then translated into multiple Components at asset instantiation inside our engine, whose values are defaulted or optionally defined via the game state.

3.6 Asset System and Prefab Instantiation

While the Blender plugin is responsible for serialising ECS metadata into the glTF extras fields, a runtime pipeline was required to deserialize this data and build gameplay entities inside the engine. This pipeline is divided into two stages: low-level asset loading through the custom asset system followed by ECS prefab instantiation through the Scene class.

3.6.1 Binary Asset Loading. To support deterministic runtime performance and easy use of memory, the engine firstly loads assets rather than treating glTF as a runtime data source. glTF files are parsed using `cgltf`, a lightweight, zero-dependency glTF parser that reads from the file structure without imposing any internal scene representation.

During loading, `assetsys` transforms the hierarchical glTF representation into a single contiguous `Asset` structure custom made for the engine. This structure flattens all scene resources into tightly packed arrays of meshes, materials, textures, skeletons, animations, and node descriptors. This design stops constantly looking for pointers between dynamically allocated objects and ensures that all scene traversal operations are made on cached memory.

It also decouples the asset structure from gameplay metadata. While glTF nodes may contain extras fields, these are not interpreted during parsing. Instead, they are copied as raw JSON strings in the corresponding `Node.extras_json` field. This ensures that the asset layer remains fully responsible for the structuring and loading of all 3D data, while all gameplay components stored in the glTF files are deferred to the Scene class and into other core systems.

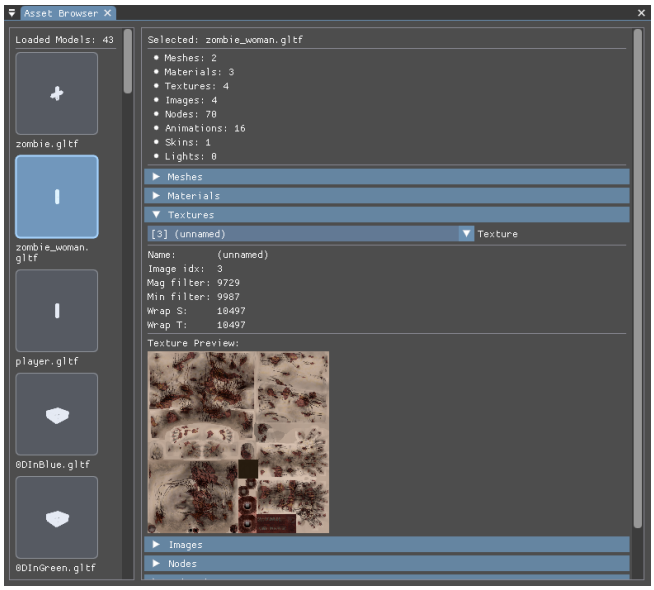

Fig. 3. Debug UI view of the asset browser showing all loaded assets and the data structures in each one.

3.6.2 Prefab Instantiation. Runtime entity construction is handled by the `Scene::InstantiatePrefab` pipeline, which connects the asset representation and the ECS. The system gets the flattened `Asset` structure produced by `assetsys` and builds entities from all the respective nodes in the asset's hierarchy .

Instantiation happens by recursively traversing the node array and treating each node as a possible entity. For each node, the system evaluates structural data such as transform hierarchy, primitives, and light descriptors, and translates them into ECS components attached to an allocated entity ID.

Renderable nodes are configured directly with specific rendering components such as `C_StaticMesh`, `C_AnimatedMesh`, or `C_Light`. These components reference shared asset data rather than owning it. That means that geometry and material data are decoupled from entity instances and can be reused across multiple prefabs without duplication.

Transform construction happens during instantiation by resolving the parent and child relationships inside the node hierarchy, ensuring consistency with the original space defined in Blender.

When a node contains an `extras_json` component, the system deserialises it using RapidJSON into separate components. These components are then resolved against the ECS registry and added dynamically to the entity. This allows prefab data to determine the ECS composition without the need for hardcoded entity definitions, making the instantiation of prefabs fully reliant on data only.

3.6.3 Design. This separation between asset loading and prefab instantiation is the boundary between data representation and runtime interpretation, decoupling the asset side completely from the scene and prefab instantiating side. The asset system is responsible only for deterministic parsing and the construction of a contiguous memory layout, while all gameplay semantics are passed to the

scene class. This key design decision was to avoid interpreting ECS metadata during the asset loading phase also allowing for future work on asynchronous asset loading and prefab instantiation.

3.7 Level Generation

To support the infinite wave-based core gameplay loop planned, a procedural generation system was implemeted as a separate module. This system utilises an infinite growth adaptation of the Wave Function Collapse (WFC) algorithm [Gumin 2016]. The module dynamically constructs a 2-bit adjacency socket system during asset initialization by parsing the string-based node names (e.g. "Doorway_N"), instead of attempting structural analysis through model geometry. The WFC solver utilises this data to calculate Shannon Entropy on a grid, starting from a few seed cells and branching outward organically to generate logical level layouts. Additionally, themes are propagated in clusters through alteration of cell choice weighting, preventing noisy, convoluted levels with many transitions and instead opting for a biome-like structure. Because constraint satisfaction algorithms can be CPU-intensive, the generation logic is wrapped in an asynchronous pipeline with `std::async`, allowing the game to keep frame rates stable while background threads calculate new grids.

Importantly, this generator does not rely on a custom rendering or physics pipeline, and was instead engineered to interact directly with the Entity Component System and Prefab architecture. The grid output `LevelFloor` acts as a spatial blueprint for the ECS to instantiate colliders and spawn entities, aided by the Blender level editor and prefab system, where components can simply be assigned to models. After grid collapse on a background thread, the engine iterates through the grid and invokes the `scene->InstantiatePrefab()` pipeline, spawning the specified room assets in their designated locations, and assigning corresponding `C_Transform` and collider components. By routing the procedural generation this way, the generalised Prefab system allows for both fully bespoke, manually-crafted levels and procedurally constucted environments using the exact same underlying architecture.

3.8 Audio

The engine includes a lightweight audio system designed to integrate closely with the ECS architecture and gameplay layer while remaining decoupled from other engine systems. The system was implemented using a small public C API backed internally by a private C++ implementation using the miniaudio backend and SDL runtime integration. This architecture allows gameplay systems to trigger audio events through handles and helper functions without exposing playback logic to the rest of the engine.

Audio clips are divided into three categories: sound effects (SFX), ambient audio, and soundtrack audio. This separation allows different layers of the game mix to be independently controlled during runtime. Final playback volume is set through a mix of master volume, category volume, clip volume, and per-instance playback volume. This proved particularly useful during debugging because environmental ambience and music could be adjusted independently from gameplay feedback such as weapon fire, zombie sounds, and UI interactions.

A major design goal was supporting gameplay sounds overlapping with each other reliably under normal game conditions. Instead of treating loaded clips as directly playable objects, the engine separates reusable clip data from active playback voices. This enables repeated one-shot effects such as footsteps, gunshots, impacts, and zombie attacks to overlap correctly without interrupting previous playback instances. Active one-shot voices are updated and cleaned automatically during runtime, while looping audio such as ambience and soundtrack tracks remain persistent until explicitly stopped or replaced.

The audio system also supports spatial audio behaviour through transforms and positional attenuation settings. Gameplay entities and the camera system update the world-space positions, while attenuation remains inside the audio backend. This allows environmental and enemy sounds to contribute to gameplay awareness without needing to have gameplay code directly communicating to the underlying audio library.

3.9 Input system

The input system abstracts low-level SDL input events into game actions. Instead of letting gameplay systems read keyboard keys, mouse buttons, or gamepad buttons directly, the engine defines a fixed list of actions. Each action can have multiple binds from not only keyboard, also mouse button, gamepad button and gamepad axes. Therefore the gameplay system has less dependency on the specific input device.

3.9.1 Action Mapping. All actions are defined in a list, which is used to generate `InputAction` enum and action names. The input system first read the key binding `json` file, or create a default key binding `json` file if there is no file available. Player changes on the binding can be saved and loaded. The system categorized each action's input value and detect the state of input:
`Input_IsActionPressed`, `Input_IsActionJustPressed`, `Input_GetActionValue`.

3.9.2 In game rebinding. The input system supports rebinding through the in-game options menu. When the UI waits for a new binding, the input system captures the next keyboard, mouse, or gamepad input and stores it as a pending binding. The UI then applies it to the selected action and saves the result to the key binding JSON file.

3.10 User Interface

The UI system is split into two layers: the engine UI and in game UI. Both layers are implemented with Dear ImGui library and rendered in the same Vulkan ImGui frame. The renderer shares the same ImGui context and draws the engine UI first, and then calls the game side callbacks to draw the in game UI. This allowed the renderer, game, input system, and camera system to share the same UI frame.

The engine UI is used for runtime development tools. A docking branch of ImGui is used to provide a dockable layout for each development tool. These tools can read and edit the live runtime data. The in game UI provides a player facing menu and HUD, and handles multiple events to show the player-facing information. Its state is also used by the main loop to block gameplay input and

scene updates when menus, skill selection, or the Engine UI are active. The art style of in game UI is achieved with custom fonts and loading 2D image by `stb_image` loader. This makes the UI system part of both the game presentation layer and the engine debugging workflow.

4 EVALUATION

This chapter evaluates the effectiveness and performance of the core systems developed for the project. Since the engine was designed around scalability, modularity, and high-performance, the evaluation focuses on architectural and runtime metrics. The tests were carried out through regular in-game runtime behaviour to reflect user experience.

4.1 Asset Loading and ECS Instantiation Performances

The objective of this evaluation is to determine whether the chosen ECS architecture and prefab pipeline can scale to complex scenes without introducing excessive loading times. To evaluate this SDL high-resolution timers were positioned strategically in the engine. Measurements were collected during a cold-start execution of a level-generation sequence containing a static level geometry, animated characters, entities with physics, and gameplay objects. The world build pipeline was separated into three stages. These stages were chosen because they represent the three main bottlenecks in scene construction:

- Asset loading and deserialisation
- Prefab instantiation and component construction
- Renderer scene submission

All three stages will expose what is the biggest cost to initialise the first scene build once the game demo is executing.

4.1.1 Asset Loading. Three different categories of assets were tested: **Characters**, **Level prefab** and **Weapon**.

Characters had animated meshes, skins, and animation channels.

Level prefabs are composed of a full room with a floor, walls and furniture. These components are considered as entities and all have static meshes with corresponding colliders for each individual entity. The levels also include entities for the lights in the scene.

Weapon is a single entity with 5 nodes and simply a static mesh.

Asset Type	Asset Name	Nodes	Load Time (ms)
Level prefab	1O0DOut	75	~10
Level prefab	2O1DEastOut	54	~11
Level prefab	4O0DOut	13	~5
Character	zombie	71	~280
Character	zombie_woman	70	~860
Character	player	38	~15
Weapon	colt	5	~4

Table 1. Asset loading performance across representative prefab types.

Overall, the results indicate that asset loading cost is primarily due to animation complexity rather than node count or entity count, with static level assets remaining consistently quick to load and predictable. This animation cost is due to the amount of parsing necessary to read and interpret all the animation channels and key frames with joint transforms through the animation's node hierarchy.

4.1.2 Prefab Instantiation. Prefab instantiation was evaluated by measuring the time taken to convert loaded glTF node hierarchies into ECS entities and fully construct their associated components. This stage includes transform creation, component deserialisation, physics body generation, mesh binding, and animation initialisation.

Asset Type	Asset Name	Nodes	Entities	Time (ms)
Level prefab	1O0DOut	75	55	~1.6
Level prefab	2O1DEastOut	54	38	~1.1
Level prefab	4O0DOut	13	10	~0.21
Character	zombie	71	1	~0.4
Character	zombie_woman	70	2	~0.5
Weapon	colt	5	1	~0.09

Table 2. Prefab instantiation performance across representative asset types.

Character prefabs, such as zombies, are instantiated significantly faster than level prefabs, typically 3-4 ×, due to the lower number of ECS entities and integration of components into each entity. This holds true even when a zombie has more nodes in the asset than a room.

4.1.3 Renderer Submission Cost. It quantifies how expensive it is to take ECS data and feed it into the renderer each scene build. `Renderer_ChangeScene()` took **0.22 ms** in total to submit the first frame.

The total measured world build time for the generated gameplay scene was approximately **1649.24 ms**.

Pipeline Stage	Time (ms)
Asset Loading (Total)	1549.09
Prefab Instantiation (Total)	18.25
Renderer Scene Submission	0.22
Other (Scene setup / linking / gameplay init)	81.68
World Build Total	**1649.24**

Table 3. Breakdown of world build pipeline timing across major engine stages.

The results show that renderer submission itself is negligible compared to the surrounding scene construction and linking operations. Asset Loading takes the overwhelming majority of the startup time, but those assets only ever have to be loaded once while the other two stages happen multiple times during runtime.

4.2 CPU Frame Time Profiling

Evaluation of the mean CPU per-frame time was also performed, aimed at assessing the scalability, stability and performance bottlenecks of our implementation under varying loads. Given the current wave-based structure of our game, Hell Mist, testing was performed under progressively increasing waves of enemies, increasing the number of animated meshes, physics bodies and renderable meshes.

An in-engine, header-only CPU profiler was utilised to capture exact execution times across different systems.

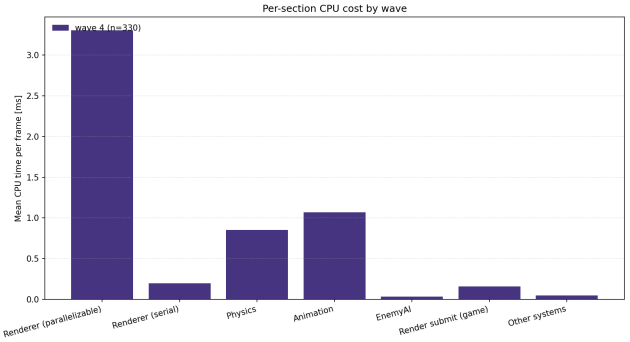

Fig. 4. Mean per-frame CPU time (Wave 4, n=330) captured on a low-end latop without a dedicated GPU (11th Gen Intel Core i7-1185G7 with 8-cores, Intel Iris Xe Graphics (TGL GT2), 16GB RAM) in Release mode at 1080p.

To establish a baseline performance profile on low-specification hardware, an initial CPU frame-time capture was executed on a laptop lacking a dedicated GPU (Figure 4). This test successfully isolated the three major CPU time-sinks in the engine: the parallelisable section of the rendering pipeline, skeletal animation, and physics solvers, which, when combined, consume over 90 of computation time. Identifying these core architectural costs justified a thorough evaluation focused on the CPU scaling of various systems.

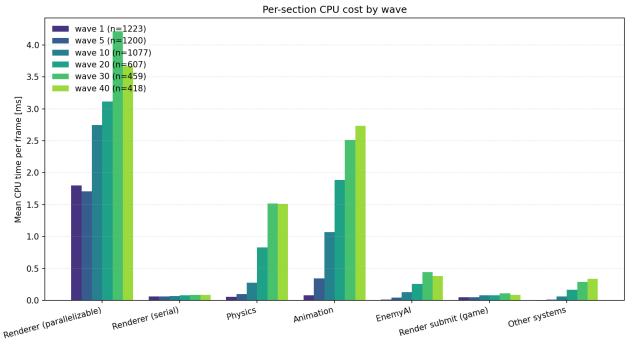

Fig. 5. Aggregated mean CPU execution times across progressing numbers of enemy waves, demonstrating system scalability under large entity counts, captured on a laptop with a dedicated GPU (12th Gen Intel Core i5-12500H with 12 cores, NVIDIA GeForce RTX 3050 Laptop GPU) in release mode at 1080p.

To evaluate system scalability, mean CPU times were averaged across an increasing number of waves (1-40). While the relative CPU frame time contributions of each section remained stable throughout the test, there are some notable points for individual sections, especially on larger waves. The parallelizable renderer remains most expensive and appears to flatten at the higher waves, however wave 30 presents anomalous and peaks higher than wave 40. This presents in a few other systems as well, never occurring on repeat profiling suggesting a testing error. Conversely, the Animation system exhibits more linear scaling, growing from near-zero to ~2.8ms by Wave 40, but still appears to plateau on slightly. Combined with the physics system, these three modules demand nearly 10ms of CPU processing time per frame at later waves. Waves 20, 30 and 40 ran at 60, 45 and 42 FPS, requiring 16.6ms, 22.2ms and 23,8ms respectively, highlighting the CPU scaling as a primary bottleneck when increasing entity counts.

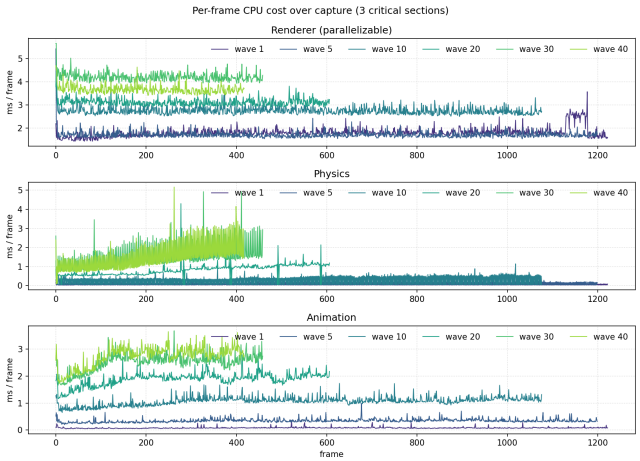

Fig. 6. Per-frame CPU cost variance across the three critical systems, highlighting performance instability and stutters at large entity counts, captured on the same system as figure 5.

While average means for different systems present the general scalability, they miss any frame-pacing variability that can cause clearly visible gameplay stutter. Therefore, a frame-time stability graph was recorded shown in figure 6 to track precise execution costs per frame over a 10 second period. The Renderer has tight bands and a consistent performance throughout testing, and the Animation system performs similarly, with slightly larger peaks in frame time and an upward slope at the start, possibly due to the in-game enemies moving to attack the player, blending into multiple different animations. The Physics module at first glance appears unstable, however this is a constantly alternating pattern of higher and lower execution times, particularly visible when costs spike above 1ms. This structured variance is an artifact of the physics engine's settings: the fixed-time step update function accumulates the delta-time and runs twice on one frame, leaving the next update with only one needed step. This pattern appears repeatedly due to the game's delta-time being 1.5x larger than the fixed-time step duration, accumulating half of the duration each update.

5 DISCUSSION

5.1 Scene Building Performance

The results indicate that the primary bottleneck during scene building is asset loading, particularly the processing of rigged character assets with several animations. Despite making up a small proportion of the entities in the scene, animated characters such as zombies

account for a disproportionately large portion of loading time due to the cost of parsing skeletal hierarchies, animation channels, and keyframe data through several animations. This shows that animation deserialisation is the most costly stage of the startup scene building.

In contrast, the prefab instantiation stage shows great scalability. Even in scenes with over 50 entities per prefab, total instantiation time remains low in comparison to asset loading. This indicates that the ECS construction model does not introduce major overhead, and that using several components is appropriate for real use cases. In particular, RigidBody creation, transform setup, and animation component initialisation scale almost linearly with entity count but remain negligible in the total scene building time.

Renderer scene submission does not produce any substantial overhead in comparison to the entire workload. The cost of uploading ECS data structures into the GPU through the rendering system is still under 1ms, suggesting that the rendering interface is not costly for scene construction. This is an important result, as it confirms that rendering abstraction does not introduce unexpected startup overhead.

The remaining "other" category accounts for a non-trivial portion of total world build time. This includes system linking, gameplay initialisation, physics world configuration, and runtime attachment of components (e.g. weapon sockets).

Overall, the results demonstrate that the system scales well in terms of ECS construction and rendering submission, but that asset loading remains the biggest performance constraint during cold start.

5.2 Scene Per-Frame Performance

The evaluation data reveals an that the performance bottlenecks in a monolithic game loop rarely exist in isolation, triggering performance degradations across multiple systems. The linear scaling of the animation system was a large part of the bottleneck, most likely due to the multiple pose interpolations calculated for every entity currently active in the scene. This overarching strain on the CPU forces it into the delta-time accumulation seen in figure 6, so any optimisation of the physics Narrowphase would not stop this stutter. Close analysis of the animation pipeline demonstrates a major flaw in a component-based system, as performing many complex spherical interpolations across fragmented memory blocks leads to cache misses and limited performance. Future iterations can solve this by limiting the animated entities to only those rendered to the screen, or by migrating to a data-oriented design to maximise cache coherency.

5.3 Multi-threaded support

One of the drawbacks of having built all of the core CPU systems from scratch is the needed effort to allow support for multi-threading, which was not included in the time budget of the project. In the results we can evaluate that a single thread is not enough to handle the sizeable amount of entities being rendered in high waves inside the game: a rough approximation of the number is almost a thousand animated entities being updated every frame without animation-culling optimizations. Support for multi-threading in modern engines is a must, as increasing the amount of cores has been an increasing trend in the manufacturing of modern CPUs the past few decades.

6 CONCLUSION

This project presented the design and implementation of KillSwitch, a custom Vulkan-based game engine focused on scalable rendering, ECS architecture, and specialised workflows for retro-PSX style survival horror games. The engine integrates rendering, physics, animation, audio, prefabs, and gameplay systems focusing on modularity. It shows a practical example of being able to create an engine from scratch in the span of a few months without needing to solely rely on external libraries.

The engine is capable of swiftly building third person action games with its easy blender pipeline, and its performance is good for being single-threaded. Also, the engine being fully composed of de-coupled modules, it allows for large reusability of its components, whether it is swapping them or using them in another engine.

6.1 Limitations

The main limitation the engine currently has is its lack for multi-threaded support, which causes most of our CPU intensive tasks to become a bottleneck when the amount of entities being simulated reach a certain amount.

The evaluation also demonstrated that in the Asset-to-Entity pipeline asset loading was the most substantial part of the startup bottleneck. It added the biggest cost, while ECS entity construction and renderer submission remained negligible.

6.2 Future Work

There are still several limitations and opportunities for future work. Asset loading currently occurs synchronously during startup and would benefit from asynchronous streaming and background loading systems. Future iterations could also improve the engine with networking and broader platform compatibility.

Multi-threading support needs to be added to leverage all the capabilities of modern CPUs, and to avoid the animation and subsequent physics bottlenecks in games.

The physics engine also needs to be improved, adding an acceleration structure to be able to scale to larger world games where there is a great amount of entities being simulated.

ACKNOWLEDGMENTS

The authors would like to thank Prof. Hamish Carr and Dr. Rafael Kuffner dos Anjos for their supervision and help throughout the project.

REFERENCES

Sebastian Aaltonen. 2025. No Graphics API. https://www.sebastianaaltonen.com/blog/no-graphics-api. Accessed: 2026-05-15.

Mike Acton. 2017. *Solving the Right Problems for Engine Programmers - Mike Acton (TGC 2017).* https://youtu.be/4B00hV3wmMY?si=nQDjfBuVL4EinOGP YouTube video, accessed May 15, 2026.

Antonio Ambrosio, Daniele De Vinco, Francesco Foglia, Carmine Spagnuolo, and Vittorio Scarano. 2025. The Impact of ECS Logic on Parallel Performance in Agent-based Model Simulations. In *Proceedings of the 3rd Special Track on Big Data and High-Performance Computing (BigHPC 2025), co-located with the 4th Italian Conference on*

Big Data and Data Science (ITADATA 2025) (CEUR Workshop Proceedings, Vol. 4124). CEUR-WS.org, Turin, Italy, 1–10. https://ceur-ws.org/Vol-4124/paper43.pdf

Gino Bergen. 2001. Proximity queries and penetration depth computation on 3d game objects. (01 2001).

B. T. Ciavola and J. K. Gershenson. 2016. Affordance theory for engineering design. *Research in Engineering Design* 27 (2016), 251–263. https://doi.org/10.1007/s00163-016-0216-5

Reiner Dolp. 2025. Practical Global Optimization and Analysis of Render Graphs. https://youtu.be/v9LaTFLhP38. [T51] Video Presentation.

Federico Dossena and Andrea Trentini. 2022. OpenLDAT—A system for the measurement of display latency metrics. *Journal of the Society for Information Display* 30, 8 (2022), 599–608. https://doi.org/10.1002/jsid.1104 arXiv:https://sid.onlinelibrary.wiley.com/doi/pdf/10.1002/jsid.1104

Ryan Fleury. 2022. Untangling Lifetimes: The Arena Allocator. *Digital Grove* (23 Sept. 2022). https://www.dgtlgrove.com/p/untangling-lifetimes-the-arena-allocator Making performant dynamic manual memory management in C feel almost like garbage collection.

E.G. Gilbert, D.W. Johnson, and S.S. Keerthi. 1988. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation* 4, 2 (1988), 193–203. https://doi.org/10.1109/56.2083

Jason Gregory. 2018. *Game Engine Architecture* (3 ed.). CRC Press.

Maxim Gumin. 2016. *Wave Function Collapse Algorithm.* https://github.com/mxgmn/WaveFunctionCollapse

id Software. 1999. Quake GPL Source Release. https://github.com/id-Software/Quake. GitHub repository, accessed 2026-05-15.

Khronos. 2024. glTF 2.0 Specification. https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html.

Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. 2021. Architecture Anti-Patterns: Automatically Detectable Violations of Design Principles. 47, 5 (1 May 2021), 1008–1028. https://doi.org/10.1109/TSE.2019.2910856 Publisher Copyright: © 1976-2012 IEEE..

Casey Muratori. 2004. Designing and Evaluating Reusable Components (2004). https://caseymuratori.com/blog_0024. API Design lecture originally presented at GameTech 2004.

Yuriy O'Donnell. 2017. FrameGraph: Extensible Rendering Architecture in Frostbite. GDC Vault. https://gdcvault.com/play/1024612/FrameGraph-Extensible-Rendering-Architecture-in GDC 2017 Talk.

Matt Purslow. 2022. *The Witcher 4: CD Projekt Explains Why It Switched to Unreal Engine 5.* https://www.ign.com/articles/the-witcher-4-cd-projekt-red-explains-unreal-engine-5-switch Updated: Apr 5, 2022 10:58pm BST.

Kjetil Raaen and Andreas Petlund. 2015. How much delay is there really in current games? *Proceedings of the 6th ACM Multimedia Systems Conference, MMSys 2015* (03 2015), 89–92. https://doi.org/10.1145/2713168.2713188

Andrés Rivela. 2019. Why you should(n't) build your own game engine. In *ACM SIGGRAPH 2019 Talks* (Los Angeles, California) *(SIGGRAPH '19)*. Association for Computing Machinery, New York, NY, USA, Article 32, 2 pages. https://doi.org/10.1145/3306307.3328180

Jorrit Rouwe. 2021. JoltPhysics. https://github.com/jrouwe/JoltPhysics [Online; accessed 20-Feb-2026].

Simple DirectMedia Layer. 2025. SDL3/CategoryGPU. https://wiki.libsdl.org/SDL3/CategoryGPU. Accessed: 2026-05-15.

Oleksandra Sobchyshak, Santiago Berrezueta-Guzman, and Stefan Wagner. 2026. Pushing the boundaries of immersion and storytelling: A technical review of Unreal Engine. *Displays* 91 (Jan. 2026), 103268. https://doi.org/10.1016/j.displa.2025.103268

XY Zhang. 2014. *Reliable and Fast Conservative Advancement for Physically Realistic Rigid Body Simulation.* Springer Singapore, Singapore, 105–120. https://doi.org/10.1007/978-981-4560-32-0_7

A PROJECT MANAGEMENT

A.1 Supervisor and Theme

The moment we were assigned to a group we debated about the game theme. There was common ground, as half the members wanted a horror theme, and we all wanted to have some style of third person combat. At week two we agreed to delve into a third-person horror-style action game with adventure and rogue-like elements, rendering using a PBR model with stylish shaders on characters.

We were assigned Prof. Hamish Carr for this project. We set meetings every Friday morning with our supervisor with the purpose of presenting our weekly progress and receiving feedback, and to get guidance on on-going and future tasks. Hamish was very helpful and insistent on the proper ways of organising, managing and tracking the project's progress, ensuring we had a proper view of all the alternatives to the choices we made, and to have a clear critical path in our project estimation.

A.2 Group organization

For the development of the project, we adopted an Agile methodology: Scrum. We decided to do two-week sprints, setting up milestones to achieve and alternatives if we did not. We also planned a Minimum Viable Product (MVP) to have a player with a third person camera attached, moving and colliding with a hand-made map imported from our blender pipeline, optionally having enemies that were able to combat with the player, also having audio. Then we had a big list of improvements we could add to the MVP depending on the remaining time left.

Once the ideas and the MVP were set, we decided to split up the workload in tasks, and individually assign them depending on each member's strengths and interests, unless a task was determined to need more than one member. This way tasks were roughly evenly distributed or re-distributed depending on members not reaching the desired deadlines. With a group of six people, this was beneficial for us as we could parallelize tasks from the beginning without stepping on each other and making the sure critical path did not slow us down. We ended up grouping certain tasks when there were blocks or delays, like the blender plugin or the engine animation system.

The code structure was organized and planned out since the beginning of the project, and it was laid out immediately on GitHub, which is the version control we used. We first planned on using a trunk-based approach (iterative and incremental programming), but in the beginning for big separate tasks, we ended up using feature-based branching, where we all finished our set of tasks before pushing to main. Later, we switched to trunk-based development once our MVP was reached, where we then incrementally pushed small changes directly to the main branch, reducing the risk of bringing branch-breaking changes.

Along with the weekly meetings with our supervisor, we planned to hold short daily in-person meetings, where we would talk about the task we were currently implementing, and warn if we were stuck or needed help. We initially followed those daily meetings, but they diverged into a single weekly meeting during the middle of the semester, towards Easter break.

However, this was not our primary or only method of communication, as we had set up a Discord server from the beginning, where we had all our planning and ideas. We used daily communication and planning there, and we integrated a Discord bot to show the progress of every commit submitted to the GitHub repository. This way, we could keep track of every member's task regardless of availability.

Having this method of communication helped us move from the Scrum methodology to Iterative Development, which is what we fully ended up doing after all our core engine technologies were developed. That greatly increased our productivity in the last few weeks, where we managed to implement a lot of added features to our final game.

A.3 Planning and execution

Initial Planning. At the top of figure 7 you can appreciate the initial Gantt chart that was made in week 3. The reason for the late creation of this gantt chart was due to the knowledge and tips Hamish gave us over the span of the first two week to make a proper one. As stated before, the initial tasks were all split individually in parallel depending on our members' interests and strengths.

Progression until the MVP. The actual progress of the project can be seen in the bottom Gantt chart at figure 7. Though roughly accurate, it does not truly represent the amount of work each member has put into the project.

At the time of the creation of the initial Gantt chart, the build system was already done by Liam so each member could work on their own initial task. This initial part of the project was very slow as some of the members had no knowledge of how to tackle the start of the project or their new tasks. We still had discussions at this point on how to develop most of the tasks, what the game should be, the engine structure, etc. Almost all the tasks were elongated by at least one week more than expected due to these reasons, and until they were done we could not see much progress.

The project started taking shape after this first group of tasks. Three weeks left for the initial MVP we still had to pipeline assets from blender, to the asset system, create entities and then render them. While this process was being done, there was a second separation of tasks, mostly individually except the animation system, which was tackled by two members.

During the initial period and this one, some of our members fell ill during a long period of time, slowing down their tasks, but we could still see progress from many of the other tasks. We had assets being rendered, and the renderer was showing good progress. The physics engine, which was part of our critical path, was taking a lot more time than expected and the MVP could not be reached in time due to that. The rest of the systems continued to progress as intended, with delays in the audio system.

At week two after the initial MVP deadline, the physics engine was mostly finished, so our critical path was freed. From that point on, almost all the other systems were done, with still amazing extra progress on the renderer. With three weeks left before the deadline, we reached our MVP, which was having a movable and collide-able character in an imported scene with an attached camera through the world.

Progression until Poster Day. Once the MVP was reached, we still had to polish some of the modules and we still needed to have level generation, final assets, and have to do all the gameplay systems to have a proper game done for Poster Day. Our members continued to fall ill and we had to apply for an extension, which is reflected in the Gantt charts in figure 7. Even with an extension, we had three weeks left, and at this point, almost all the members were showing great efforts and progress through this last push.

A week before Poster Day, our engine was almost finished, and we had to Game-Jam our way into having a playable game and to have the poster done for the session. We put a lot of effort into the poster and it turned out amazing. Now only the game, video and report was needed to be finished.

After the Poster Day session, we still had four days until the deadline. All focus was on the report and the showcase video on this very final push.

Differences between the Gantt charts. The main differences between the Gantt charts can be seen in some of the elongated tasks, as well as the distribution within them. Initially, we had some tasks planned for the same team of members, but they ended up doing different tasks individually.

The apparent reason for the sizeable delay in most of the affected tasks was due to the continuous illness of the members developing said tasks. That delay propagated to some of the other tasks, delaying them by about a week. Communication was not an issue, as there was almost daily communication throughout the project, and most of the delays were noticed almost immediately. We still re-planned and accounted for the delays, concluding that we could still finalise our project in time. Also, the separation of one of the teams into individual tasks was mainly due to the difference in interests and the difference in schedule between the two members.

It can be concluded that the initial Gantt chart was fairly well estimated, it just did not account for the amount of time some members could fall ill consecutively, and also did not take into account the bugs and small refactors to be done after integrating all the engine modules together, which can be seen in the final Gantt chart. Still, measures were taken towards the unforeseen, and the final Gantt chart did not diverge far off the initial one.

B SELF-REFLECTION AND SOFT SKILLS ACQUIRED.

Considering our project execution as a whole, the team reached achieved significant triumphs, although not without a challenging management experience. This section critically reflects on our project management, team dynamics, and adaptability, showcasing the professional-level soft skills acquired that will help the future transition into the software engineering industry.

B.1 Planning, Scope, and Time allocation

In the initial stages of the project, the team formulated a plan via a Gantt chart, critical path analysis, and dependency mapping. However, constructing a correct set of tasks and allocating accurate time estimates for each proved difficult. All members had extensive experience in software development with similar levels of proficiency, however none had experience building a game engine from scratch with some areas going beyond the taught material. This resulted

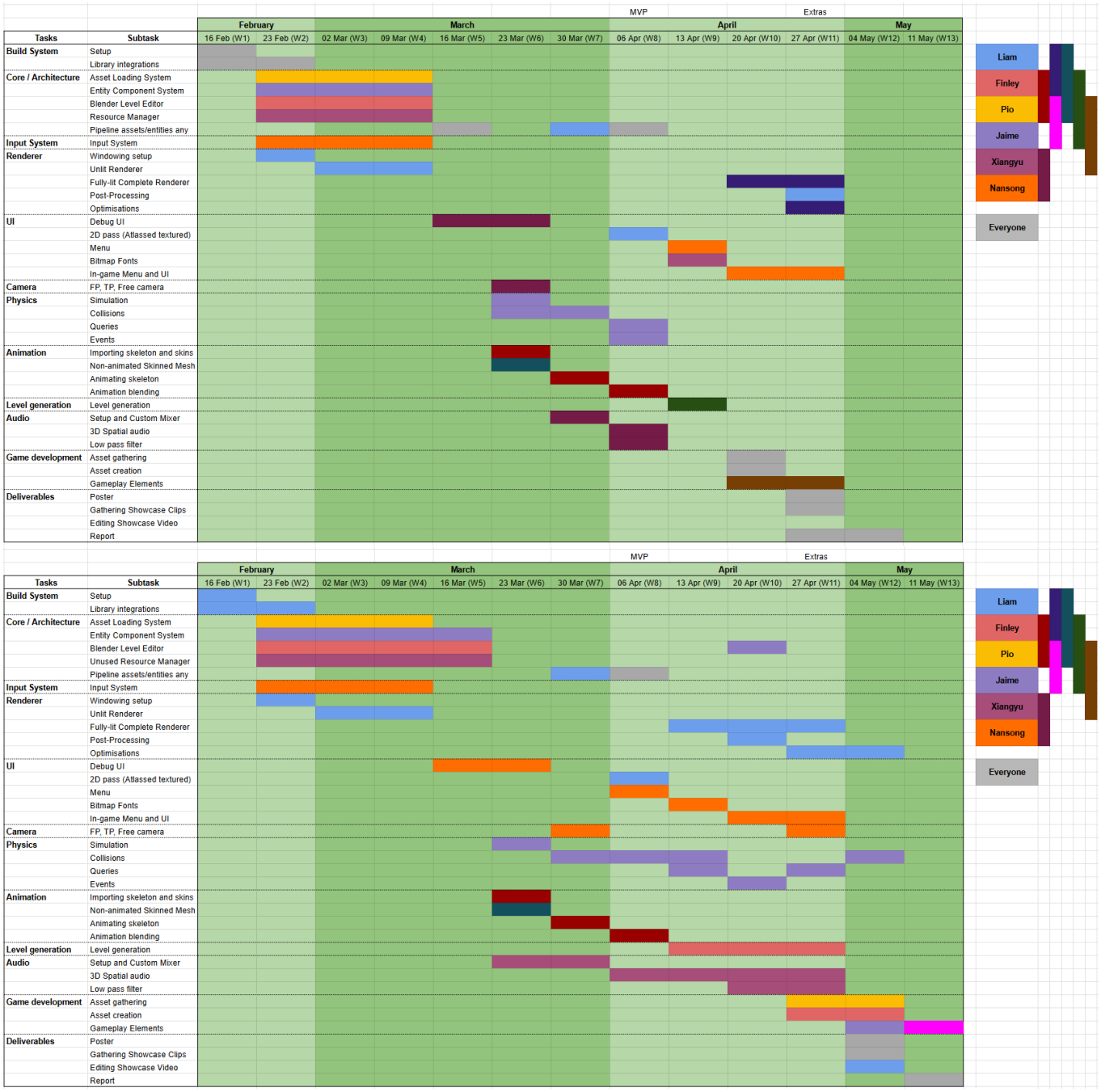

Fig. 7. Top figure: Initial Gantt chart. Bottom figure: actual progress (approximate) through the progress of the project.

in uncertainty regarding what systems specifically a game engine should contain, how these should interact together, and how time and module functionality should be prioritised. Consequently, our initial time allocations were optimistic, and task completion hopes were ambitious.

Rather than strictly adhering to flawed plan, we adapted by embracing a highly iterative, Agile approach, continually re-evaluating the scope and downscaling non-essential features. This helped ensure a playable product by the demo session, and taught us that

effective planning in the workforce requires non-optimistic, realistic estimates, as well as deep research into the domain prior to establishing commitments or scope.

B.2 Adaptability to Unforeseen Circumstances

A constant issue throughout project development was unexpected illness among members. Initially, our contingency planning consisted of only leaving buffer time between tasks or at the end of sprints, however this proved inefficient, forcing us to adapt our task management and altering the group dynamics. This highlighted the importance of contingency planning for all scenarios, possibly with the idea of creating multiple scopes to ensure that, if many members were incapacitated, the scope of the project could be altered to be less ambitious to ensure a final product was completed. Additionally, after early advice from our supervisor, we made damage mitigation plans for when one member struggled or fell ill by utilising peer-programming, documentation and clear comments.

This experience taught us that in industry environments, projects are frequently affected by staff absence, illness or other unexpected setbacks. Learning to redistribute work and re-prioritise tasks is critical to ensuring a finished product, and rigorous writing of documentation provides a clear introduction for anyone inheriting another's task unexpectedly.

B.3 Communication Challenges

Our initial communication strategy had in-person or online group meetings every weekday, to stay up-to-date with each others progress and discuss the next steps. Although this plan was strictly followed for some time, it did not last for the whole development cycle due to schedule changes. Furthermore, the group faced real-world workforce challenges through language barriers with non-native English speakers, preventing them from being able to fully understand or contribute in many in-person meetings. Similarly, one member relocated throughout the Easter period to a drastically different timezone, raising scheduling issues. Relying only on synchronised verbal meetings alienated the members who relied on translation software, so we realised that effective communication depends heavily on reliable communication methods instead of solely on technical knowledge, leading us to utilise robust, asynchronous communication software.

From these issues, we learned to create detailed pull-requests and commit messages, structure communication channels for each task, and track task completion through a modifiable Gantt chart. With remote work becoming increasingly common, long-range meetings or communications are more prevalent, so extra steps must be taken to enforce strict schedules that all parties can adhere to.

B.4 Utilisation of Supervisor Meetings

For the first half of the project, weekly meetings with our project supervisor were arranged to present current achievements, give a broad perspective on ideas and decisions, and communicate any alterations to the plan or scope. Initially, this time was not used properly and meetings lacked a structured agenda built from questions the team had, resulting in broad progress updates that failed to communicate our current situation to the supervisor. After reflection on this and strict instruction from the supervisor, we adapted the approach to ensure presentation materials were properly prepared, and any technical questions were answered.

By shifting our efforts towards clarity of preparation and relaying a broader understanding, we successfully demonstrated our constructed plan and current progress to the supervisor, enabling him to give us feedback and validate any architecture design decisions we made. In an industry scenario, we learned that pitching an idea and plan for a project requires the ability to clearly present or refactor documents so they are easily understandable with less context or in a shorter time frame.

B.5 Individual Reflections

Finley Byrne: Having never been a member of a group project with this scale before, project management was definitely challenging, especially since the supervisor assigned both the lead and secretary roles to me. At the start, I was quite overwhelmed and struggled to make decisions confidently, clearly recount meeting notes, or communicate deadlines well to the team. Later on however, by consistently attending all group and supervisor meetings, organizing communication between team members, and constructing a solid development plan together with the team, I gained more confidence in my leadership skills and learned a lot about delegating responsibility and organisation.

Nansong Yue: This was the first time I participated in such a well-organized group project to develop a game engine. In the early stages of development, I proposed many ideas. However, during the implementation phase, I lacked sufficient motivation and proactivity to discuss the implementation details due to my limited theoretical knowledge. Thanks to the support of the other team members and our supervisor, I gradually discovered my role within the group through continuous communication and close collaboration with them. This experience has taught me how to address a balance between ideation and realization, while also providing me with valuable insights into team-based software development.

Xiangyu Liu: During this project, I realised that large-scale engine and graphics projects require not only technical implementation skills, but also effective communication and collaboration abilities.One of my biggest weaknesses during the project was my lack of experience with collaborative workflows such as Git and large-scale team integration. Earlier in development, I focused too much on completing my own systems and underestimated the importance of communication and coordination within the team.I also found that communicating technical ideas clearly in English meetings was more difficult for me than I originally expected, especially during discussions with the supervisor and team members. Despite these difficulties, I genuinely tried my best to contribute to the project. Through implementing the audio and resource-management systems, I gained valuable experience in engine architecture, runtime systems, and gameplay integration inside a larger collaborative codebase. Overall, this project taught me a lot about teamwork, communication, and sustainable engine development. If I continued similar projects in the future, I would place much greater emphasis on earlier communication, collaborative workflows, and more consistent integration with the rest of the team.

Liam Swarbrick:

I had an array of personal research goals I wanted to investigate with this project, specifically around programming a sustainable Vulkan rendering architecture. My initial expectation was that I would want end up with a web of technical debt that I couldn't reuse. However, surprisingly, it turned out to be incredibly solid, something I can use to develop further projects. Our team's goals aligned nicely, and it didn't seem like anyone was unhappy with the direction our game ending up going. I for one think we ended up with something especially good, especially given the time frame, and given how half the team was dealing with illness throughout most of the development). In terms of my role in the team, I was most familiar with how to approach huge unstructured projects, to turn the teams initial shopping list of all the game engine features listed in one of the game engine books into a cohesive architecture, that can scale to real titles. I also led the high level solutions of many systems, mostly knowing what the simple alternative always is, and how they things together. Our team had high communication, but very only a few times did we work together in person, mostly during the game development at the end of the project. I knew the programming paradigms I use are not in the business code cannon taught in university, so I made sure to align with peoples existing programming patters in their parts of the code. For the renderer, I wrote in a very restricted language subset of C++ compatible C code, as my usual work is written in procedural, imperative C code, and I knew the renderer would be something I may like to continue for a long while, so wanted a C-port to be as simple as replacing glm. The project was a good opportunity to see how far you can get in a short timespan, when working with smart people.

Pio Cañas Toimil: From working first time in a big group project with high goals set from the very beggining I quickly learned that constant communication is essential. You don't want to overstep or affect anyone's work negatively and more importantly you don't want anyone in the group working on the same thing simultaneously because that would literally be a waste of time and brain power.

I needed to make sure everyone was on the same page and that we were all collectively working towards the same goal otherwise the purpose gets lost along the way. I understood where I fit in the team helping everyone individually where they needed so that no one got stuck. I also would always make sure any of my final decision were in line with everyone else's idea of the project. For example with assets I would ask what the group thought before I implemented.

Adaptability was really important, I got ill for a few days, many other group members did too. So quick adaption to different workflows for the time being were necessary. Adaptability in the sense of what work I was assigned also played an important role, sometimes I was working on a system with another group member and if I needed help with a section I would ask for it and I would also be proactive to help when it was needed by someone else.

Essentially, everyone needs to work together and if the whole group is on the same page the work flows smoothly.

Jaime Gine Paraiso: This is my first time working in such a sizeable project in a university environment. I entered with high expectations, as I already knew some of my group-mates were very smart and consistent workers. From work experience, I already knew communication is key and that we should check-in on everybody at least once a day.

I was excited and motivated seeing how almost everyone was showing a lot of interest from the start. Our visions aligned, and we were having a great time discussing all of the ideas and architecture for our game and engine.

Once the first division of tasks started, I felt useless. I had very poor management of the rest of the modules that were happening at the same time, and had high stress due to the early deadlines for the MSc Project module. It might have been due to stress or other circumstances, but I fell ill an unimaginable amount of times during this semester. It made me not focus properly on any of the modules, and made my project tasks take almost double the amount of time that was estimated.

Still, I put in good effort after our first round of tasks was done and I started feeling well back again. I feel like we had very good communication within each other and we were having a great time. The management was good and I have learnt that consistency is key. Also, I now know I need to work on my time and task management a lot more. It was a very enjoyable experience and it has been an amazing journey.

C HOW ETHICAL ISSUES ARE ADDRESSED

The project does not process personal or sensitive data. Ethical considerations primarily related to the use of third-party assets, all of which were sourced under permissive or royalty-free licences for academic use. In addition, all third-party software dependencies are used in accordance to their open-source licenses, with no changes or redistribution outside of the project.

D SUMMARY OF CONTRIBUTIONS

Liam Swarbrick: The renderer, build system and architecture for all systems (except physics and ecs, which was Jaime), stylised graphics for the game, game-design stuff, deep knowledge on the full engine-game pipeline, report introduction, background (except 2.3), and renderer sections.

Finley Byrne: Initial Blender Plugin, Layered Animation System, Procedural Level Generation, Enemy Wave Spawning/Balancing, Level Asset Creation, Poster, Showcase Video, Report.

Pio Cañas Toimil: Asset System, Initial Animation System Structure, Asset Gathering, Character Rigging and Blender Setup, Movement and Animation integration for characters, Scene architecture for multiple prefabs and specifically animated meshes, Player and Enemy components, Poster, Showcase Video, Report.

Jaime Gine Paraiso: Physics engine, Entity Component System, Blender plugin refactor, Core utilities, Events, Gameplay systems architecture, Gameplay Systems: { Player, EnemyAI, Movement, Combat }, Poster, Showcase Video, Report.

Nansong Yue: In game input support, Engine debug UI, In game UI/HUD, In game camera mechanism, UI art asset and audio asset acquisition, Audio system, Report evaluation script, Poster.

Xiangyu Liu: Resource Manager, Audio System, Audio asset acquisition, Poster, Report.

Received 16 May 2026