



BACHELOR THESIS

AI-Driven Game Engine Development using Gemini 3.0 Pro

Can Prompting Lead to a Fully Functional Game Engine?

Author

Malak Mohsen

angestrebter akademischer Grad / aspired academic degree

Bachelor of Science (BSc)

Wien, 2025 / Vienna, 2025

Studienkennzahl lt. Studienblatt /

Studyprogramm id according to studysheet : UA 033 521

Fachrichtung / Specialisation:

Informatik

Betreuer / Supervisor:

Univ.-Prof. Dipl.-Ing.

Dr. Helmut Hlavacs

Abstract

This thesis explores the extent to which Large Language Models (LLMs), specifically Gemini 3.0 Pro, can independently generate a functional, modular game engine architecture when integrated with existing low-level rendering libraries like Raylib. Utilising an iterative prompt-based methodology, the research documents the development of a C++ engine capable of supporting both 2D and 3D environments. The study details the creation of core subsystems—including scene management, resource caching, and input handling—while addressing the technical limitations of current AI models, such as the inability to export 3D geometry and audio files. To overcome these constraints, a workflow was developed where the LLM authored Python scripts to programmatically synthesise the necessary .obj and .wav assets. The resulting engine was validated through the deployment of two distinct games: the 3D labyrinth game Pac-Witch and the 2D shooter GhostInvaders. A user study involving 12 participants confirmed the engine's technical stability and performance, with 75% of testers rating the programming quality of the 3D implementation at 4 or 5 stars. Ultimately, the project demonstrates that LLMs can serve as effective architectural partners, shifting the developer's role from traditional coding to the curation of AI-generated content and laying the groundwork for future "wrapper-less" AI-driven software creation.

Kurzfassung

Diese Bachelorarbeit untersucht das Potenzial von Large Language Models (LLMs), insbesondere von Gemini 3.0 Pro, für die autonome Entwicklung einer funktionalen, modularen Game-Engine-Architektur unter Einbindung der Rendering-Bibliothek Raylib. Mittels einer iterativen, prompt-basierten Methodik dokumentiert die vorliegende Forschung die Entwicklung einer C++-Engine, die sowohl 2D- als auch 3D-Umgebungen unterstützt. Die Arbeit beschreibt detailliert die Erstellung zentraler Subsysteme – einschließlich Szenenmanagement, Ressourcen-Caching und Eingabeverarbeitung – und thematisiert dabei die technischen Grenzen aktueller KI-Modelle, wie etwa die fehlende Fähigkeit zum Export von 3D-Geometrien und Audiodateien. Um diese Einschränkungen zu überwinden, wurde ein Workflow etabliert, bei dem das LLM Python-Skripte verfasst, um die benötigten .obj- und .wav-Assets anhand eines Programms zu generieren. Die daraus resultierende Engine wurde durch die Implementierung zweier unterschiedlicher Spiele validiert: das 3D-Labyrinth-Spiel Pac-Witch und der 2D-Shooter GhostInvaders. Eine Benutzerstudie mit 12 Teilnehmern bestätigt die technische Stabilität und Leistungsfähigkeit des Frameworks, wobei 75% der Tester die Programmierqualität der 3D-Implementierung mit 4 oder 5 Sternen bewerten. Letztendlich demonstriert das Projekt, dass LLMs als effektive Architekturpartner fungieren können, wodurch sich die Aufgabe des Entwicklers vom traditionellen Programmieren hin zur Verwaltung KI-generierter Inhalte verschiebt und der Grundstein für eine zukünftige, autonome Softwareentwicklung gelegt wird.

Contents

List of Figures	iv
1 Introduction	1
2 Related Work	3
3 Technical Foundations	6
3.1 Gemini 3.0 Pro	6
3.2 Raylib	7
3.3 Dear ImGui	9
3.4 rImGui	10
4 Creating a Game Engine with LLMs	12
4.1 System Architecture and 3D Implementation	12
4.2 Functional Expansion: 2D Framework	16
4.3 Implementation Case Studies: Game Projects	18
5 Evaluation	23
6 Conclusion	29

List of Figures

4.1	Initial prompt defining the requirements for the engine's InputManager class.	14
4.2	Gemini 3.0 Pro response outlining the "Dictionary" class architecture.	14
4.3	The DebugScene: a dedicated testing environment for the 3D engine subsystems. . .	15
4.4	Segment of the Gemini response identifying the missing components required for 2D support.	16
4.5	Test2DScene: a testing scene used as a diagnostic environment for the 2D engine framework.	17
4.6	Python-based workaround suggested by Gemini 3.0 Pro to circumvent file-export limitations.	18
4.7	Visual demonstration of the Pac-Witch 3D game environment.	19
4.8	Visual demonstration of the GhostInvaders 2D game environment, illustrating the first of two gameplay waves.	20
4.9	Initial prompt specifying requirements for the HTML-based background removal utility.	21
4.10	Functional HTML utility generated by Gemini 3.0 Pro for background removal. . . .	22
5.1	Pie chart illustrating user preference data collected from the initial questionnaire. . .	23
5.2	Participant ratings regarding the overall "fun factor" of the Pac-Witch 3D environment.	24
5.3	Survey results assessing the perceived programming quality, including smoothness and the absence of bugs, for Pac-Witch.	24
5.4	Quantitative survey data illustrating the perceived entertainment value of the GhostInvaders prototype.	25
5.5	User study data illustrating the reported technical reliability of the 2D implementation.	26
5.6	Comparative participant ratings for the visual, auditory, and mechanical aspects of the Pac-Witch game.	27
5.7	Comparative participant ratings for the visual, auditory, and mechanical aspects of the GhostInvaders game.	27
5.8	Participant ratings assessing the overall quality of multimodal assets (audio, geometry (3D & 2D), and textures) generated by Gemini 3.0 Pro.	28

1 Introduction

The rapid development of Artificial Intelligence (AI) technologies in recent years has sparked a growing interest and driven innovation across many industries. The video game industry, a well-known consumer of new technologies, has become one of the primary fields that uses and benefits from AI. Integrating AI into game development is a revolutionary change that fundamentally alters the creative process, the production methods and the overall experience of interactive entertainment [1].

A key driver of this revolutionary change is the recent development and rapid advancement of Large Language Models (LLMs). LLMs can perform "in-context learning", meaning they can learn a new task from a few examples in the prompt without needing their core programming to be changed [2]. They are also able to follow complex, multi-step instructions and use advanced reasoning, like "Chain-of-Thought", which breaks down a solution step by step. LLMs understand code structure and meaning across many programming languages [3]. This enables them to write working code from a simple natural-language description or to explain any given code. This suggests a future in which humans focus on high-level design while AI handles detailed implementation. Building a game engine —a very complex task— is the ultimate test of this new approach [4].

To translate this theoretical potential into a practical experiment, this project will develop a game engine from scratch by using Gemini 3.0 Pro, Google's latest version of their AI model ¹ [5], and an iterative prompt-based method. The engine is kept as simple as possible. Furthermore, the AI will need to develop games to run on the engine to test its functionality. Building a renderer from scratch is not the goal for this research project; therefore, an already existing external low-level renderer, Raylib, will be used.

This research aims to explore how far Gemini 3.0 Pro can autonomously build a game engine, with minimal human intervention, unless it is absolutely necessary. The goal is to evaluate the generated code based on functionality, quality and correctness. As a final result, there should be a running game engine with a minimum of two playable minigames that run on the created engine. Furthermore, Gemini 3.0 Pro will be tested on its capability to generate all the necessary game assets, such as .obj files, textures, and music, which are critical aspects of a game's aesthetic.

RQ1: To what extent can Large Language Models (LLMs) independently generate a functional, modular game engine architecture when integrated with existing low-level rendering libraries like Raylib?

H1: Large Language Models, specifically Gemini 3.0 Pro, are capable of generating a functionally complete game engine and its associated assets through iterative prompting, such that the resulting

¹As of November 2025

system can support real-time interactive minigames.

To investigate this research hypothesis, the following is a breakdown of the thesis structure. Chapter Two provides a comprehensive overview of the current landscape of Large Language Models (LLMs), focusing on their evolving role within the gaming industry. Chapter Three establishes the technical foundations of the research, beginning with an in-depth analysis of Gemini 3.0 Pro, its capabilities, prompting strategies, and inherent limitations. This chapter further details the core engine components, explaining the functionalities of Raylib and its integration into the project. It also explores the Dear ImGui framework for interface management and the role of rImGui as the critical bridge enabling these systems to interact within the engine. The practical implementation and development of the game engine are detailed in Chapter Four. Subsequently, Chapter Five provides a formal evaluation of the engine's performance and utility by conducting a user study with 12 participants. Finally, Chapter Six concludes the thesis by summarising key insights, addressing the research hypothesis, and suggesting avenues for future work.

2 Related Work

This chapter lays the fundamental groundwork for this research by tracing the evolution of AI from a simple helper to a central force for creation and innovation. The field of software engineering is undergoing a fundamental transformation. This is driven by better and more popular Generative AI (GenAI), especially LLMs, which can now automate, speed up and find new solutions to complex tasks that used to require human creativity and skill [6][7][8]. Fueling this transformation was a rapid advancement in the capabilities of LLMs.

Between 2024 and 2025, LLMs evolved from simple text generators into sophisticated systems capable of deliberate, multi-step reasoning and autonomous action through agentic frameworks² [10]. A huge breakthrough was AI's capability to handle different input types such as text, image, audio, and other file formats [11]. The constant advancements of AI will lead to deep integration into core industrial workflows, which promise a productivity gain.

This promised integration is already well underway; a Google Cloud Study from 2025 confirmed that 90% of developers are already using AI in their workflows [12]. This trend of deep integration into core industrial workflows is particularly evident in the game development sector. Here, using GenAI is no longer a futuristic concept for making games; it's a current reality that's rapidly changing the industry. An Analysis of Steam's AI disclosure policy reveals that visual asset generation is the most common use case, cited in approximately 60% of all games that report using generative AI [13]. This widespread adoption is driven by the clear potential of GenAI to solve the industry's biggest problems, like high development costs and the constant need for innovation. Reports and studies agree that the main benefits of these tools are boosting efficiency and creativity [14].

However, the industry is confronting limitations, including factual hallucinations, inherent data biases absorbed from their training data and critical security vulnerabilities like prompt injection³ [15][2]. A further concern is that AI's reasoning relies on statistical matching patterns rather than logical deduction, which causes them to fail in complex or new situations [3]. Furthermore, there are technical challenges that remain, such as "catastrophic forgetting", where models tend to lose old information when they are fine-tuned. Another issue is its poor generalisation across different tasks [2]. Additionally, AI models tend to underperform in non-English languages and lack knowledge on topics relevant to low-income countries, which raises concerns about global equity and accessibility [16]. As a result, research is now shifting from just scaling AI's power to ensuring it is reliable, safe and trustworthy, which means there is a growing focus on developing more robust evaluation methods and addressing the global equity and fairness concerns [17].

²Agentic Frameworks are foundational structures for developing autonomous systems. It is a blueprint outlining how AI components should work together to achieve a specific goal [9].

³Prompt injection happens when a model is tricked by using malicious inputs (bad prompt) that are specifically designed to bypass safety checks and trick the model into producing harmful content.

While these general limitations require careful consideration, GenAI is nonetheless set to revolutionise the creation of interactive entertainment. This is a major shift, not just a minor improvement, and it will fundamentally change the way the digital worlds are designed and built [18]. Game development, a uniquely complex field within software engineering, is proving to be a great area for applying these new AI technologies. For decades, the conversation about AI in gaming has centred on its in-game functions, such as controlling NPC behaviour, dynamically adjusting difficulty and generating content [19]. These technologies are creating a more dynamic, immersive and visually spectacular virtual world by moving beyond pre-scripted events to generate intelligent behaviours, breathtaking graphics and personalised content in real-time [20].

Industry leaders are already shipping these advanced in-game capabilities. NVIDIA has developed its Avatar Cloud Engine (ACE), which is being used to create autonomous NPCs that can perform complex strategic tasks. This has been demonstrated by the NARAKA: BLADEPOINT MOBILE, which has AI-powered teammates. Additionally, NVIDIA’s project Half-Life 2 RTX utilises the RTX Remix platform to remaster classic games with full ray tracing and DLSS 4⁴ technology. This highlights AI’s power in modernising graphics [22].

Using AI in games remains crucial. However, LLMs have opened a new frontier. LLMs let us now use AI as a core creation tool, not just a game component. This is a critical distinction, as it separates old in-game AI from modern AI’s disruptive potential to co-develop or even autonomously build the game engines themselves. Game engines, the core software frameworks upon which games are built, are no longer passive platforms. They are actively integrating AI tools to make them a central part of the development experience. Unity has taken a significant step in this direction by combining its various AI initiatives into a single Unity AI suite. Unity AI functions as a contextual assistant directly within the Unity Editor. Using prompting, developers can generate new C# scripts, create placeholder assets, debug console errors and automate complex scene setup tasks. This deep integration eliminates the need to switch between multiple applications and integrates AI capabilities directly into the primary workspace [23]. Even gaming studios are developing AI tools to streamline their workflows. Ubisoft La Forge created Ghostwriter. Ghostwriter is an in-house AI tool that assists scriptwriters by generating first drafts of ambient NPC dialogue [24]. In 2023, Electronic Arts (EA) explored various strategies to enhance game testing, as manual testing had become impractical. EA improved game testing by using AI to move beyond traditional scripted bots, by exploring reinforcement learning to increase test coverage [25].

This potential, however, raises fundamental questions about the practical limits of AI in software creation, specifically whether a complete playable game can be developed using only natural-language prompts [26]. A foundational demonstration of this capability is Project Genie, developed

⁴DLSS 4 is NVIDIA’s latest generation of Deep Learning Super Sampling technology, which uses an AI-powered vision transformer model to enhance image quality and performance [21].

by DeepMind. Genie is a generative model that can create playable endless 2D platformer games from a single image prompt. By learning from videos on the internet, the AI demonstrates the ability to generate interactive environments from scratch, which is a core function of a game engine [27].

Building on this area of research, the 2025 Paper, AI-Driven Web Game Development with Gemini 2.5 Pro, investigated creating a multi-platform web-based game without using traditional game engines like Unity or Godot. The conclusion was that creating a playable game using prompts alone is only possible through a structured, iterative process. The findings provide an empirical basis for the current shift, where the developer's role transitions into a curator of AI-generated content. The study also offered a quantitative insight into where LLMs excel and where they struggle [28]. Further addressing this challenge, researchers from Microsoft developed a framework, called Model as a Game (MaaG), to address key challenges in generative games. The environment of the game is created frame-by-frame by neural networks instead of a traditional graphics engine. The research focuses on improving the logical and spatial consistency of AI-generated game worlds. This research represents a significant step toward making AI-driven engines more robust and coherent [29]. Along these lines, the paper "Multi-Actor Generative AI as a Game Engine" suggests the concept of having AI act like a Game Master in a tabletop role-playing game, which means the AI is responsible for generating the environment and the story based on player actions. This approach is suited for creating dynamic, multi-agent simulations and interactive narratives [30].

As a final illustration, the project that comes closest to realising a fully AI-driven game engine is GameNGen, a project by Google AI Research, which is the first game engine powered entirely by a neural model. Instead of relying on traditional, manually coded software systems for rendering and game logic, GameNGen uses a diffusion model trained on recorded gameplay sessions. It generates the next frame of a game based on a sequence of past frames and player actions, effectively simulating the classic game DOOM in real-time on a single TPU ⁵ [32].

⁵Tensor Processing Unit, a custom AI accelerator chip developed by Google [31]

3 Technical Foundations

The development of this project relies on a modern, integrated technology stack that balances high-level reasoning with low-level performance. At the core is Gemini 3.0 Pro, utilised as an "agentic" partner for architectural planning, code generation, asset generation and complex problem-solving. The engine's graphical foundation is built upon Raylib, a code-centric library that provides a transparent and powerful framework for 2D and 3D rendering. To facilitate real-time debugging and user interaction, Dear ImGui is integrated as the primary immediate-mode interface, connected to the engine via the rImGui bridge. Together, these tools form a cohesive pipeline that allows for rapid prototyping, robust input management, and a seamless transition between development tools and active gameplay.

3.1 Gemini 3.0 Pro

Gemini 3.0 Pro offers users a "state-of-the-art" reasoning engine designed to handle complex, multi-step tasks across native multimodal inputs, including text, images, audio, video, and PDFs [33][34][35]. It distinguishes itself with advanced "agentic" capabilities that allow it to function as an autonomous partner capable of planning, executing, and verifying workflows—such as browsing the web, using tools, or managing long-horizon tasks—rather than merely predicting text [36][37][38]. For developers and creators, the model provides a massive 1 million token context window, enabling the analysis of entire codebases or extensive documents, supported by new "vibe coding" features that can translate natural language ideas into fully interactive applications [39][40][36][37]. Additionally, it introduces configurable "Deep Think" capabilities that are controlled via "thinking levels" which allow the model to allocate extra compute time for solving complex mathematical or scientific problems with greater depth and nuance [37][41][42].

To effectively use Gemini 3.0 Pro, users must adapt their prompting strategy to be brief and direct, as the model prioritises logic over the verbosity or persuasion often used with previous LLMs [42][43]. When working with the model's massive 1 million token context window—which can ingest entire code repositories, hours of video and audio, or hundreds of PDF pages which are processed as images to preserve layout—it is best practice to place specific instructions at the very end of the prompt to ensure they anchor the model's reasoning [44][45]. This multimodal capability is enhanced by features like "Gemini Canvas" and "Generative UI," which allow the model to move beyond static text generation to "vibe coding," where natural language prompts are instantly converted into interactive applications, dynamic dashboards, or visual simulations directly within the interface [34][40][37].

Despite its advancements in reasoning and multimodal processing, Gemini 3.0 Pro has several documented limitations and technical constraints. While excellent for scaffolding, it can hallucinate

non-existent library methods [46]. It also has specific limitations in media creation and editing. In image editing via the Gemini 3.0 Pro Image model, users report struggles with maintaining character consistency between the input and the generated result, and the model often renders small text or long paragraphs blurrily or adds text to an image that was not wanted or needed in the first place. It also tends to confuse location instruction, such as left vs. right [47]. Regarding audio generation, Gemini 3.0 Pro accepts audio files as input for analysis but cannot natively generate music files [33]. Finally, the model cannot directly export .obj files or 3D meshes; because its output is limited to text and 2D images, it can only assist by generating code, such as Python scripts, required to build 3D assets externally, rather than producing the geometry files itself [35].

3.2 Raylib

Raylib is an open-source, cross-platform library purposefully designed for developing graphical applications and video games [48] [49]. Originally conceived as a pedagogical tool, it was designed for game development students with artistic backgrounds and no prior programming experience [50][51]. Written in plain C, the library is highly modular and prioritises accessibility by passing data by value rather than using pointers in its primary functions [52]. Raylib distinguishes itself by being entirely self-contained; all necessary components for image decoding and audio handling are embedded directly within the source [53] [54]. This code-centric approach ensures that game logic remains transparent to the developer throughout the process [48]. The library offers a comprehensive suite of features for both 2D and 3D graphics, including procedural geometry, skeletal animation, VR stereo rendering, and custom shader support. These capabilities are bolstered by dedicated modules for audio (raudio), mathematics (raymath), and a specialised OpenGL abstraction layer (rlgl). Raylib supports an extensive range of platforms, from desktop environments like Windows, Linux, and macOS to mobile, web (HTML5), and consoles such as the PS4 [51] [49] [54]. While natively written in C, a robust community has produced over 70 language bindings, enabling its use in C++, Python, Rust, and C# [51] [49]. Ultimately, utilising raylib for software development is comparable to using precision manual instruments instead of complex automated systems. While this requires a more intensive foundational implementation, it grants the developer total control and a profound understanding of the software's internal architectural interactions [48].

Beyond its high-level design principles, Raylib provides a granular set of tools that translate this "manual precision" approach into practical rendering and management capabilities. Raylib offers a robust 2D rendering system that supports both standard geometric primitives—such as lines, circles, and rectangles—and complex splines, including linear, basis, Catmull-Rom, and Bézier. A dedicated Camera2D system further extends these capabilities with integrated pan, rotate, and zoom controls [52]. For 3D operations, the library implements a comprehensive pipeline encompassing procedural geometry generation for shapes like cubes, spheres, and heightmaps. It also

provides skeletal animation compatibility for GLTF ⁶, IQM ⁷, and M3D ⁸formats utilising GPU skinning to efficiently offload vertex transformation logic from the CPU [58]. The library further enables advanced CPU-side image manipulation—including cropping, resizing, and tinting—prior to GPU uploading, and supports a variety of compressed texture formats. Its flexible material system facilitates physically based rendering (PBR) alongside diffuse and normal mapping, while custom GLSL shader support allows for both model-specific effects and full-screen post-processing. Beyond visuals, Raylib includes a multimedia audio module (raudio). Powered by the miniaudio backend, it supports multi-channel mixing and various formats such as WAV and MP3 with options for memory-loading or disk-streaming. Mathematical operations are handled by raymath, which provides specialised functions for Vector, Matrix, and Quaternion math, including C++ operator overloads for streamlined syntax. Performance is optimised via rgl, an OpenGL abstraction layer that unifies multiple OpenGL versions through an internal batching system; this system accumulates vertices to minimise draw calls and boost performance for dynamic objects [54]. For environments lacking GPU drivers, the rlsw module offers a CPU-only implementation of an OpenGL 1.1-style API [53]. Input management is equally comprehensive, covering keyboards, mice, gamepads, and touch screens, with a built-in event recording system for testing and cinematics. Finally, the ecosystem is rounded out by raygui for interface tools, physac for 2D physics, and rres for compressed, encrypted resource packaging [54].

In this project, Raylib functions as the primary engine framework, providing the core rendering pipeline and hardware abstraction required for both 2D and 3D gameplay. The application lifecycle is anchored by `InitWindow()`, `InitAudioDevice()`, and the `WindowShouldClose()` polling loop within `Application.cpp`, which orchestrates the transition between the launcher and active scenes. For 3D rendering in `Pac-Witch` and `DebugScene`, the project utilises the 3D Pipeline to load complex models via `LoadModel()` and `LoadModelFromMesh()`, applying custom Blinn-Phong GLSL shaders to handle advanced lighting and specular effects. These scenes leverage Raylib’s mathematical module (raymath) for vector subtraction and normalisation to drive camera movement and entity logic, alongside the `Camera3D` system for perspective projection. The project’s 2D capabilities, demonstrated in `GhostInvaders` and `Test2DScene`, rely on the 2D rendering system to draw animated sprites via `DrawTexturePro()` and `DrawTextureEx()`, while the `Camera2D` structure manages world-to-screen coordinate translations for mouse picking. CPU-side image manipulation is used in `Test2DScene` to procedurally generate spritesheets using `GenImageColor()` and `ImageDrawRectangle()` before uploading them to VRAM with `LoadTextureFromImage()`. Furthermore, the project implements a robust audio system using the raudio module, employing `PlayMusicStream()` for background tracks in `Pac-Witch` and `PlaySound()` for one-shot effects like pumpkin

⁶Graphics Library Transmission Format (WebGL) is a standard file format for 3D scenes and models [55].

⁷Inter-Quake Model is a binary file format for 3D models with skeletal animation [56].

⁸Model 3D (M3D) is a versatile, open-source 3D model format [57].

collection. Input management is handled globally, mapping physical keys to game actions via `IsKeyDown()` and `IsKeyPressed()`. At the same time, the `rlgl` layer is directly accessed in `ShadowMap.cpp` to manually manage texture slots and enable depth components for shadow mapping.

3.3 Dear ImGui

Dear ImGui is a lightweight C++ library designed to maximise productivity by minimising state synchronisation [59]. Unlike "Retained Mode" systems (e.g., Qt or WPF) that rely on persistent object trees, Dear ImGui defines and renders UI elements dynamically during every frame based on real-time application data [60][61]. Dear ImGui is primarily used by programmers to build internal development tools, rather than consumer-facing applications [59][62]. Widely recognised for its utility in game development, Dear ImGui is the primary choice for building internal engine tools—such as real-time loggers, profilers, level editors, and "cheat menus"—which allow developers to inspect and manipulate a program's internal state directly. High-profile implementations of these debugging suites include those used in *Cyberpunk 2077*, *Grand Theft Auto VI*, and various entries in the *Assassin's Creed* and *Call of Duty* series [62][63] [60]. Its portable, renderer-independent design makes Dear ImGui an ideal solution for real-time 3D applications, game consoles, and embedded systems where standard OS-level interface features are unavailable. Central to the Immediate Mode GUI (IMGUI) paradigm is the philosophy of State Ownership, where the application remains the "single source of truth"; unlike traditional frameworks, the UI does not store data locally but reflects it directly through Frame-by-Frame Execution. By procedurally defining the interface during every iteration—such as calling `ImGui::Button("Save")` within a conditional statement—the library shifts control back to the core logic. To ensure platform flexibility, high-level commands are transformed into optimised vertex buffers and command lists, allowing the host application to render the result using graphics APIs like OpenGL, DirectX, or Vulkan [59][64][60][61].

While Dear ImGui is engineered for high performance and memory efficiency, its design involves specific trade-offs: the procedural rebuilding of the UI every frame imposes a constant, predictable CPU workload. Its primary strength lies in its ease of integration, requiring only a few platform-agnostic C++ files to provide a comprehensive suite of widgets—such as sliders, trees, and tables—ideal for complex tooling. However, when compared to consumer-grade frameworks, it lacks native support for internationalisation (e.g., right-to-left text), accessibility features for screen readers, and advanced anti-aliasing, reinforcing its role as a specialised tool for internal developer applications rather than general-purpose software[65][66][59][61].

Dear ImGui's functionality is built upon a procedural architecture that requires UI elements to be defined dynamically during every frame of the application loop [59][60]. The system's lifecycle is governed by Core Lifecycle and Context Functions, where `CreateContext()` initialises the library, `NewFrame()` begins the frame description, and `Render()` finalises optimised vertex buffers for the

graphics API. Spatial organisation is managed through Window and Container Functions, utilising "Begin/End" pairs such as `Begin()` and `End()` for floating windows. These containers are populated by a rich library of Interactive Widgets, ranging from standard buttons and text inputs to specialised tools like `DragFloat` sliders and sophisticated `ColorPicker4` editors. Because the paradigm is immediate, Layout and Cursor Control are handled by moving a virtual cursor using functions like `SameLine()` to align items horizontally or `BeginGroup()` to treat multiple widgets as a single unit. For more complex interfaces, the library offers Specialised Features, including nested menu bars, modal dialogues, and a dedicated drag-and-drop API, as well as hierarchical `TreeNode` structures and data visualisation tools like `PlotLines`. To assist developers in mastering these features, the library includes Debugging and Utility Tools such as `ShowMetricsWindow()` for performance tracking and the indispensable `ShowDemoWindow()`, which provides a live, interactive reference for nearly every capability within the API [65][59].

In this project, the Dear ImGui API is utilised to construct a real-time debugging suite and dynamic gameplay overlays that bridge the engine's internal logic with a visual interface. Spatial organisation is managed through Window and Container Functions, using `ImGui::Begin()` and `ImGui::End()` to create persistent "Scoreboard" and "Settings" windows, as well as scene-specific panels. Within these containers, Interactive Widgets provide direct manipulation of engine parameters: `ImGui::SliderFloat()` and `ImGui::Checkbox()` are used for real-time audio mixing and muting, while `ImGui::DragFloat3()` allows for the live repositioning of 3D light sources and camera coordinates. The library's Layout and Cursor Control functions, such as `ImGui::SameLine()`, `ImGui::Separator()`, and `ImGui::Spacing()`, are employed to organise complex player statistics and hierarchical structures like `ImGui::TreeNode()`, which categorises individual light properties. For gameplay feedback, the project leverages Specialised Features like `ImGui::ProgressBar` to visualise boss health, player health and pumpkin collection progress, and `ImGui::TextColored()` to provide visual warnings—such as flashing red timers when a power-up is nearing expiration. To handle global engine state, Menu Bar Functions like `ImGui::BeginMainMenuBar()` provide a unified "Exit to Menu" interface for scene management. Finally, the project implements Input Handling integration by querying `ImGui::GetIO().WantCaptureKeyboard` and `WantCaptureMouse`, ensuring that game actions are suppressed when the user is interacting with these procedural UI elements.

3.4 rImGui

`rImGui` is an open-source library that serves as an integration layer (backend), allowing developers to use Dear ImGui, an immediate mode graphical user interface library, within Raylib applications [67][68]. It bridges the two libraries by handling the translation of ImGui's draw commands and input handling into Raylib's rendering and event systems [69]. The primary purpose of `rImGui` is to enable the creation of complex GUIs—such as debug tools, level editors, or application inter-

faces—inside Raylib’s real-time render loop. While Raylib is designed for video game programming and multimedia, adding `rImGui` allows for the rapid development of tooling interfaces without needing to build custom widgets from scratch. `ImGui` is a C++ library, so `rImGui` is written in C++ to create the backend integration. However, the API is also designed to be usable with pure C code. The library does not have specific dependencies other than Raylib and `ImGui` itself. It includes Premake scripts for development but can be integrated into projects using CMake or by directly adding the source files [69][68].

It is designed to fit directly into the standard Raylib game loop. A typical implementation involves initialising the library with `rImGuiSetup()`, and then "sandwiching" `ImGui` calls between `rImGuiBegin()` and `rImGuiEnd()` inside the main drawing loop [68]. The library provides helper functions to render Raylib textures within `ImGui` contexts. Functions like `rImGuiImage` and `rImGuiImageButton` allow Raylib `Texture2D` objects to be used as UI elements [70]. A known issue with `rImGui` is that it does not automatically respect the `WantCaptureMouse` and `WantCaptureKeyboard` flags used by `ImGui`. This means that clicks inside an `ImGui` window may "bleed through" and be registered as clicks in the underlying Raylib game world unless the developer manually implements checks to prevent this [71].

In this project, `rImGui` serves as the vital bridge between the Raylib rendering engine and the Dear `ImGui` interface. The engine lifecycle is managed in `Application.cpp`, where `rImGuiSetup(true)` is called during construction to initialise the backend with support for custom fonts and emoji glyphs. To ensure a clean exit and prevent memory leaks, `rImGuiShutdown()` is explicitly called within the `Application` destructor. During the core execution loop in `Application::Run()`, the library’s "sandwich" pattern is utilised: `rImGuiBegin()` and `rImGuiEnd()` encapsulate the UI rendering pass, allowing the `m_currentScene` to draw its specific `OnGui()` elements and the universal "Exit to Menu" bar over the 3D or 2D game world. Furthermore, the project addresses the "input bleed-through" limitation of the integration. Within the `InputManager.cpp`, the engine queries the `ImGui` IO state via `WantCaptureKeyboard` and `WantCaptureMouse` to drive the `IsInputBlocked()` and `IsMouseCapturedByUI()` safety gates. This ensures that game actions, such as player movement or camera panning, are suppressed when the user is interacting with the `rImGui`-rendered overlays. This specific integration allows scene-specific `OnGui()` methods—such as those found in `GhostInvaders.cpp`, `PacWitch.cpp`, and `DebugScene.cpp`—to be rendered as a seamless overlay on top of the 2D and 3D game worlds. Finally, by wrapping the `m_currentScene` logic within these `rImGui` frame boundaries, the engine provides a persistent UI environment that remains stable even as the `Application` switches between different scene instances.

4 Creating a Game Engine with LLMs

This chapter details the project's implementation phase, specifically the development of a game engine using a Large Language Model (LLM) in conjunction with Raylib, ImGui, and rImGui. The engine was developed using an iterative process, where functionality was refined through a series of targeted prompts. Initial prompts established the foundational requirements for a C++ engine, while subsequent iterations defined the core architecture, including a continuous execution cycle for both logic and rendering. Central to this architecture is the Application class, which manages window context, audio devices, and subsystem memory allocation. The engine's lifecycle follows a structured sequence: Input Polling (capturing hardware events), the Update Phase (calculating logic via delta time), the Render Phase (drawing to the back buffer), and a final Teardown for safe resource deallocation. To ensure frame-rate independence, the engine utilises delta time for all logic, while simultaneously tracking total elapsed time and performance metrics (FPS). The strategy of the iterative prompting was that with each newly created class, Gemini 3.0 Pro was given all newly generated files again, to avoid the "instant-forgetting" that occurred. Should a class be modified or expanded, the updated source file was resubmitted to the LLM to ensure the model remained apprised of the latest changes. As the number of files increased, providing the entire codebase became increasingly cumbersome; consequently, I opted to submit only the .cpp files to the LLM. This strategy was highly effective for the creation of the engine itself. Furthermore, with each new aspect of the engine that was to be created, a new chat was opened to avoid biased answers from Gemini 3.0 Pro.

4.1 System Architecture and 3D Implementation

To begin, the implementation recommended by Gemini 3.0 Pro commences with the creation of the Main.cpp file; this serves as the programme's primary entry point, instantiating the engine and calling its Run() method. The Application class serves as the central orchestrator of the engine, managing the high-level lifecycle of the software. By utilising a pointer-based scene management system, the class decouples the core engine subsystems—such as input handling, resource caching, and camera control—from specific game logic. This architecture allows for fluid transitions between a "Launcher" state and various polymorphic "Scene" states. To support this polymorphism, an abstract Scene base class was implemented as the blueprint for all game states. This interface defines a standard execution contract through pure virtual methods: Init() for resource setup, Update() for logic, and Draw() for rendering. By utilising this common interface, the application can execute complex game logic while remaining agnostic to the specific implementation details of a scene—be it a 3D sandbox or a 2D menu. Furthermore, the class centralises the integration of the graphical backend (Raylib) and the immediate-mode user interface (Dear ImGui), ensuring that hardware resources are initialised, updated, and deallocated in a thread-safe and predictable

manner.

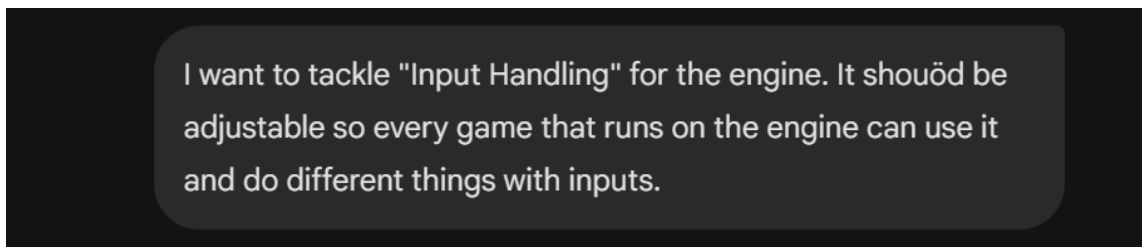
The subsequent stage of the implementation process involved determining the most effective method for loading resources into the engine to be utilised as game assets. The implementation of the ResourceManager transitioned the engine from direct disk access to a centralised caching architecture. Initially developed to manage 2D textures through a "Find or Load" strategy, the class utilises `std::unordered_map` to associate file paths with unique hardware handles, effectively preventing redundant memory allocation. This foundation was subsequently extended to accommodate diverse asset types, including 3D models and audio. By implementing specialised logic for each type—such as distinguishably handling static audio buffers versus streamed music tracks, which ensured optimal memory performance and guaranteed teardown of all resources through its automated `UnloadAll` routine.

Following the development of the ResourceManager class, a 3D camera was required to test its functionality; this necessitated the implementation and instantiation of the CameraManager. This class provided a high-level abstraction of the 3D viewing frustum, decoupling the complexities of vector-based navigation from the core application loop. By encapsulating the Camera3D structure, the manager facilitates a "camera rig" approach where movement is calculated relative to the current view orientation. Specifically, the implementation utilises vector normalisation and cross-product math to enable planar movement, ensuring that camera translation remains parallel to the world's X-Z plane. This design allows for a versatile navigation system that supports both perspective zooming and synchronised translation of the camera's eye and target coordinates, providing a stable foundation for 3D world interaction. To maintain visual stability and prevent accidental camera drift, the CameraManager employs a restricted input model. By moving away from a traditional "Free-Look" system—which often results in unintended rotation when the user interacts with UI elements—the engine utilises a fixed-target rig. Camera movement is only permitted through explicit key-presses (arrow keys) or deliberate scroll-wheel input. Furthermore, the integration with the Application layer ensures that during menu-heavy states, such as the Launcher or Debug overlays, mouse inputs are intercepted by the UI context, effectively disabling camera transformations and ensuring a seamless transition between world interaction and interface navigation. This was implemented after testing the ResourceManager and the CameraManager in a debugging environment called `DebugScene`.

Throughout various prompts, Gemini 3.0 Pro repeatedly suggested a class for managing time; however, a technical glitch caused the model to recurse over the point, much like a scratched record. Nevertheless this class proved to be useful as a foundational component of the engine architecture, serving as the global synchroniser for all frame-rate independent logic. By abstracting Raylib's native timing functions—specifically `GetFrameTime()` and `GetTime()`—into a dedicated static utility class, the engine ensures that movements, physics, and animations remain consistent

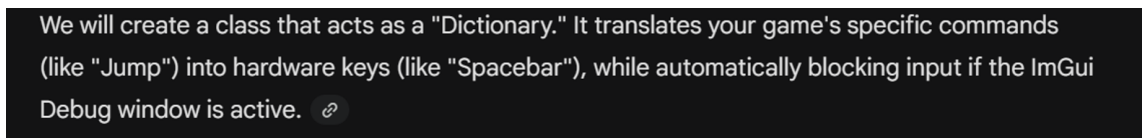
regardless of hardware performance or fluctuations in frames per second (FPS) by synchronising these values at the beginning of each iteration in the Application loop.

To facilitate interactivity, the engine required a robust input management system. This enables the capturing of keyboard and mouse events, allowing the player to control scene assets and execute game commands. Figure 4.1 illustrates the prompt provided to the LLM to generate the InputManager class. As shown in Figure 4.2, Gemini advocates a "dictionary-based" strategy; this maps abstract game actions to physical hardware keys, thereby facilitating seamless input rebinding across various game scenes. Furthermore, the class implements a critical "Safety Gate" mechanism that resolves input conflicts between the game world and the Dear ImGui interface. By monitoring UI capture states, the InputManager suppresses game-level inputs during active UI interaction, thereby preventing unintended character actions and ensuring a professional, stable user experience during debugging and menu navigation.

A screenshot of a text prompt on a dark background. The text is white and reads: "I want to tackle 'Input Handling' for the engine. It should be adjustable so every game that runs on the engine can use it and do different things with inputs." The text is enclosed in a light gray rounded rectangle.

I want to tackle "Input Handling" for the engine. It should be adjustable so every game that runs on the engine can use it and do different things with inputs.

Figure 4.1: Initial prompt defining the requirements for the engine's InputManager class.

A screenshot of a text response on a dark background. The text is white and reads: "We will create a class that acts as a 'Dictionary.' It translates your game's specific commands (like 'Jump') into hardware keys (like 'Spacebar'), while automatically blocking input if the ImGui Debug window is active." There is a small blue icon at the end of the text.


We will create a class that acts as a "Dictionary." It translates your game's specific commands (like "Jump") into hardware keys (like "Spacebar"), while automatically blocking input if the ImGui Debug window is active. 

Figure 4.2: Gemini 3.0 Pro response outlining the "Dictionary" class architecture.

The next step toward achieving visual realism within the engine was the implementation of shading and dynamic light sources. My primary goal was to ensure the engine could switch between different shader types fluidly, which led to Gemini suggesting the architectural decision to treat shaders as standard engine assets. In alignment with the goal of decoupling rendering from the core engine, the ResourceManager was extended, incorporating the LoadShaderAsset method to manage shader resources independently. This function accepts GLSL vertex and fragment shader files and integrates them directly into the active Scene, facilitating a highly modular approach to visual fidelity. In addition to the C++ logic, Gemini 3.0 Pro produced a suite of specialised shader files designed to support various illumination models, such as Gouraud, Phong, and Blinn-Phong

shading. To manage these assets effectively, the engine employs a unique key-generation strategy within the resource cache; by concatenating the vertex and fragment file paths, it ensures that shader programmes are compiled only once and reused efficiently. A dedicated light system was developed to bridge the gap between C++ CPU data and the GLSL GPU shader, facilitating the effective management of physical light sources. The core of this system is a modular Light struct that encapsulates properties such as type (Point or Directional), position, colour, and intensity. A critical feature of this implementation is the use of cached uniform locations; by identifying and storing the GPU memory addresses of light properties at startup, the engine avoids expensive string-based lookups during the main game loop. The system supports an array of multiple light sources, which are updated every frame by the active Scene to provide real-time illumination. To complement these shading models, a robust shadow mapping system was necessary to handle spatial depth and light occlusion. This was implemented through a dedicated ShadowMap class designed to encapsulate the complexity of the Shadow Buffer (RenderTexture), the Light Camera, and the necessary coordinate transformations.

During the development phase, a dedicated testing scene was established to evaluate each new class; the results were then reported back to Gemini 3.0 Pro to determine which features required further adaptation or inclusion. This class served as a visual feedback to the code that was provided by the LLM. The DebugScene acted as the engine's primary diagnostic environment, facilitating the validation of complex 3D math and subsystem integration. The visual configuration of the scene is shown in Figure 4.3. This environment served as a vital testbed for 3D picking and input rebinding, ensuring the integrity of the "Safety Gate" separating the UI from the game world.

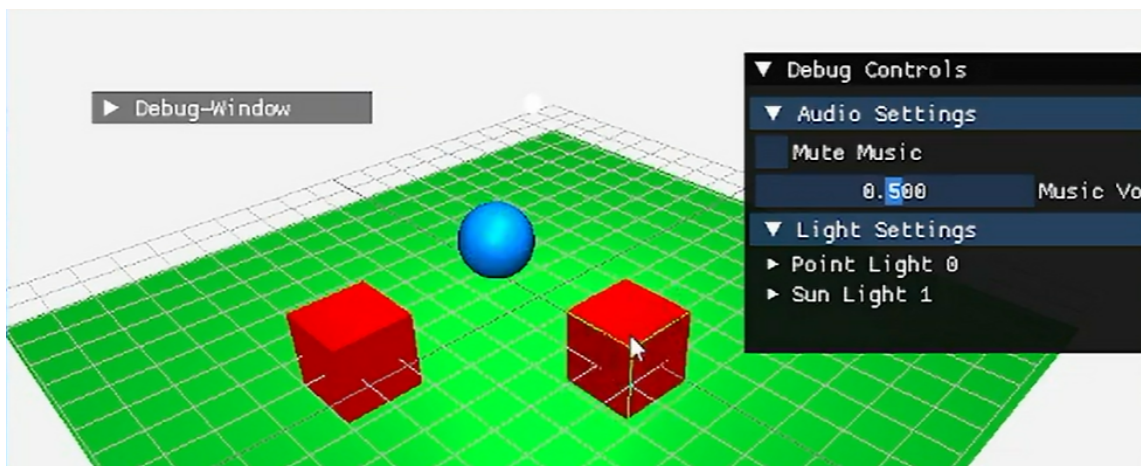


Figure 4.3: The DebugScene: a dedicated testing environment for the 3D engine subsystems.

4.2 Functional Expansion: 2D Framework

Once the 3D engine was confirmed as fully operational, the decision was made to extend its capabilities to support 2D game development. To facilitate 2D development, I requested a gap analysis from Gemini 3.0 Pro. Figure 4.4 depicts the portion of the response outlining the missing components, specifically the need to extend the camera system to support Camera2D objects and to adapt the light and shadow logic for a 2D environment.

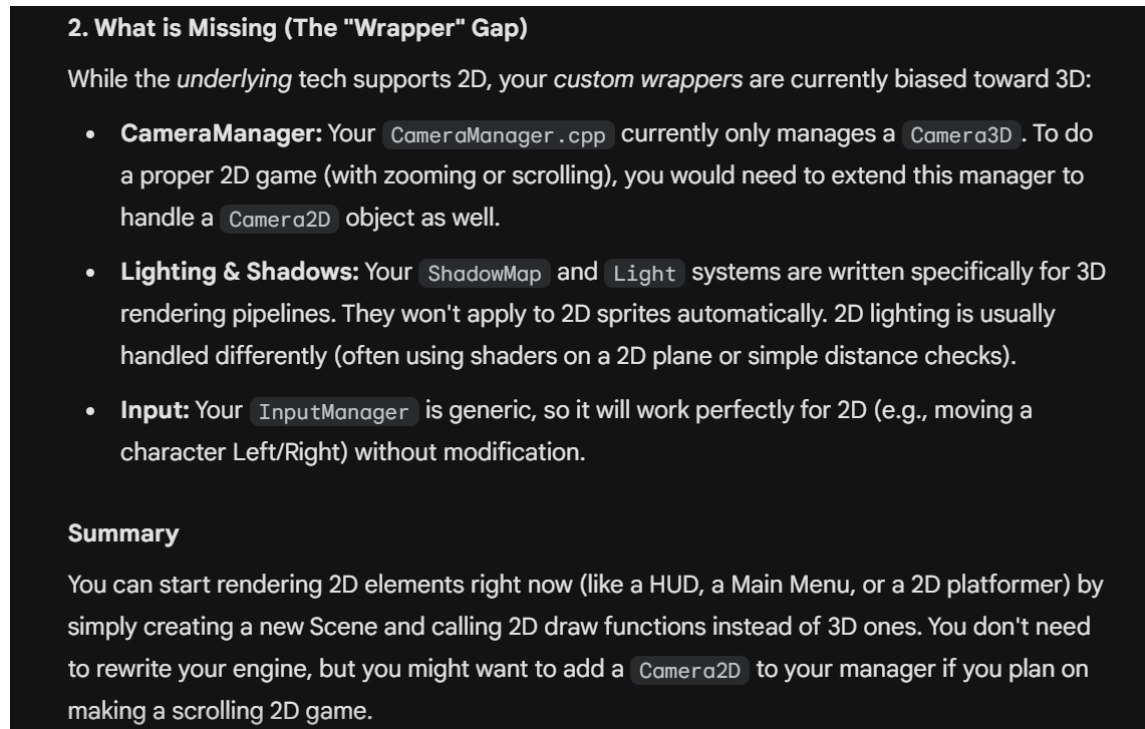


Figure 4.4: Segment of the Gemini response identifying the missing components required for 2D support.

To implement these requirements, a dedicated `Camera2DManager` class was developed to wrap Raylib's native 2D camera structure and provide high-level control over the orthographic viewport. This manager mirrors the architecture of the 3D system, utilising a similar update logic to handle frame-rate-independent panning and zooming. Specifically, the implementation captures mouse-wheel input to adjust the zoom factor—clamped between $0.1x$ and $5.0x$ to avoid visual artefacts—while mapping arrow key inputs to translate the camera target within world space.

The subsequent logical phase in the engine's development, as recommended by Gemini 3.0 Pro, was the implementation of a dedicated sprite animation system. This component was essential;

whilst the engine could already render static textures via the `ResourceManager`, it lacked the functionality to manage the dynamic, frame-based movement required for expressive 2D gameplay. The `SpriteAnimation` class was designed to manage a "source rectangle" that traverses a spritesheet, isolating the texture into individual frames and cycling through them based on the engine's delta time. This system was necessary to provide visual feedback for character states—such as idle or walking states—by simply shifting the vertical or horizontal coordinates of the texture "window".

To verify the logic behind these frame calculations, the system underwent initial validation within a `Test2DScene`. This secondary technical sandbox served as the 2D equivalent to the original `DebugScene`, providing a controlled environment to validate procedural texture generation, 2D coordinate translation, and mixed-media rendering. A key feature of this environment was the `GenerateFakeSpriteSheet()` method; this demonstrated CPU-to-GPU memory transfer by programmatically "painting" coloured squares onto an image and uploading them to VRAM. By creating these test assets without external file dependencies, the engine's core rendering capabilities were proven in isolation. Furthermore, the scene was used to validate the `GetScreenToWorld2D()` function, ensuring that mouse clicks were accurately translated into world coordinates regardless of camera zoom or panning—a requirement critical for precise 2D "picking" and interaction logic. The visual configuration and various debug elements of this environment are illustrated in Figure 4.5, demonstrating the scene's utility as a comprehensive development testbed.

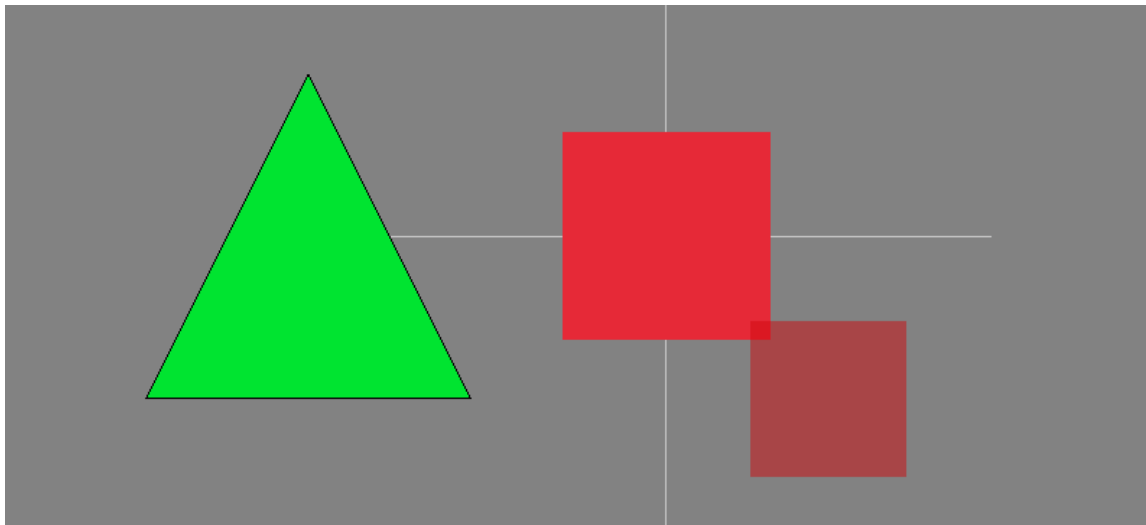


Figure 4.5: `Test2DScene`: a testing scene used as a diagnostic environment for the 2D engine framework.

4.3 Implementation Case Studies: Game Projects

Following the development phase, I identified two unique game projects to serve as a comprehensive stress test for the engine's 2D and 3D capabilities. To maintain consistency in the development workflow, Gemini 3.0 Pro was utilised to produce the necessary music, models, and textures. An immediate challenge arose as Gemini 3.0 Pro lacked the capability to generate .obj or .mtl files directly. However, as illustrated in Figure 4.6, the model provided a workaround by producing a Python script designed to automatically generate these files upon execution. A similar constraint was encountered regarding audio assets, as the model could not directly export MP3 or WAV files. This was again overcome by using Gemini 3.0 Pro to develop Python scripts that synthesised the necessary audio data programmatically.

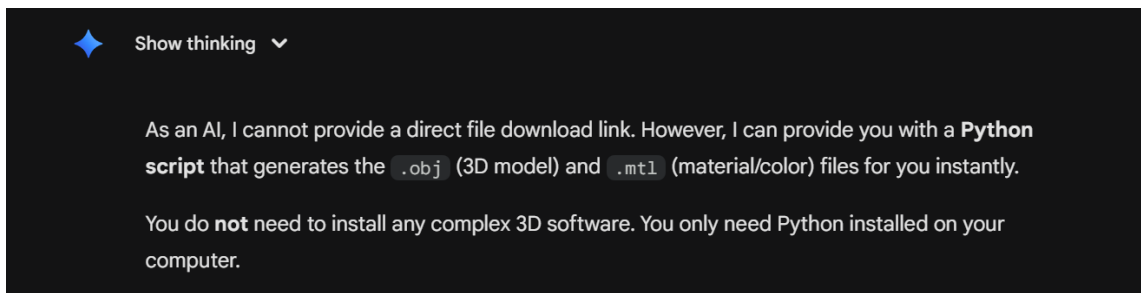


Figure 4.6: Python-based workaround suggested by Gemini 3.0 Pro to circumvent file-export limitations.

Pac-Witch was developed as the engine's first practical application, designed to stress-test core capabilities such as scene management and resource handling while benchmarking the robustness of the input subsystems. The game utilises a modular class structure inheriting from a base Scene component. This 3D arcade style labyrinth game utilises the engine's scene management and input subsystems to orchestrate grid-based player movement through a custom-defined `m_levelMap`. This string-based grid is parsed by the engine to translate abstract characters into physical 3D world space coordinates using calculated offsets and a fixed tile size of 8.0 units. In this graveyard-themed labyrinth, the user plays as Pac-Witch, whose objective is to clear the level by collecting all pumpkins scattered throughout the map. The challenge intensifies as four ghosts patrol the area, utilising line-of-sight detection to spot and chase the player; contact results in the loss of a life and a reset to the starting position. Strategic depth is introduced through "big pumpkin" power-ups, which grant ten seconds of invulnerability, causing ghosts to flee and allowing the player to safely capture them. Upon being captured, ghosts are returned to their designated spawn area at the edge of the level. The project rigorously tests the rendering pipeline by implementing Blinn-Phong shading, directional shadow mapping, and dynamic point lighting within the engine's varied

model contexts. Furthermore, the game validates the audio system’s flexibility through a custom "ducking" implementation that prioritises sound effects over background music during gameplay events. Player movement is facilitated via the WASD keys, providing intuitive navigation through the game world. Uniquely, the entire codebase and 3D assets were generated via Gemini 3.0 Pro. As shown in Figure 4.7, the final game environment demonstrates the successful synthesis of the engine’s 3D rendering capabilities.



Figure 4.7: Visual demonstration of the Pac-Witch 3D game environment.

To facilitate real-time performance tuning and evaluation, the project implements a comprehensive ImGui-based control suite. This interface provides the player with granular control over the environment, including real-time manipulation of light colour and position, as well as the ability to manually override camera coordinates and "Look At" targets. Additionally, the UI serves as a central audio console, allowing for independent volume adjustments and muting of music and sound effects. In addition to the configuration tools, the interface features a dedicated "Scoreboard" window, as seen in Figure 4.7, that provides real-time telemetry and gameplay statistics. This module tracks the player’s remaining lives—represented visually by heart icons—and monitors the progression of the "Power Up" state through a dynamic, colour-coded timer bar that flashes as the effect nears expiration. Furthermore, the tab maintains an active tally of the "Pumpkin Hunt," displaying the total number of items collected versus those remaining to provide the player with a clear metric of level completion.

Complementing the 3D stress test, GhostInvaders was developed to rigorously validate the engine’s 2D rendering and logic subsystems. Inspired by the arcade classic Space Invaders, this fixed-shooter

tasks the player with controlling a witch at the bottom of the screen, launching pumpkin projectiles to defend against a descending formation of 20 ghosts. Once the wave of ghosts is cleared, the state transitions to a boss battle against a Devil entity. Horizontal movement is mapped to the A and D keys, while the Spacebar is utilised to trigger the primary firing mechanism. Technically, this project validates the `SpriteAnimation` class for handling frame-based character states—including idle, attacking, and dying animations—and utilises the `Camera2DManager` to handle viewport scaling and coordinate mapping. Furthermore, the game serves as a benchmark for the engine's 2D collision logic, managing high-frequency interactions between multiple projectile types—pumpkins, chains, and flames—simultaneously. As with the previous game, all sprite textures were generated using Gemini 3.0 Pro nanobanana feature. Figure 4.8 illustrates the initial wave of the game, showcasing the starting formation and enemy layout.



Figure 4.8: Visual demonstration of the GhostInvaders 2D game environment, illustrating the first of two gameplay waves.

The gameplay logic is underpinned by a dynamic difficulty scaling system within the ghost formation. As the number of active ghosts decreases, the engine calculates a linear progression that increases the march speed from a base of 120.0 to a maximum of 400.0 units, intensifying the challenge as the player nears the end of the wave. This phase also tests the AI's ability to align attacks with player positioning; ghosts only trigger their "Chain" projectiles when the witch is within a specific horizontal threshold. Upon transitioning to the `WAVE_BOSS` state, the engine shifts to a more complex behaviour tree where the Devil entity actively monitors the player's projectiles. The boss utilises a "Dodge" mechanic, calculating the trajectory of incoming pumpkins to perform evasive manoeuvres.

To ensure a high degree of player control and system transparency, GhostInvaders incorporates a multi-window ImGui interface. As shown in Figure 4.8, the "Witch" tab provides essential gameplay telemetry, tracking remaining lives via a visual heart count system and displaying the current score. Parallel to this, an "Enemy" status window dynamically updates according to the game state: it monitors the remaining ghost count via a purple progress bar during the initial wave, before transitioning to a comprehensive boss tracker. In the final encounter, this interface displays the boss's remaining lives using heart icons alongside a dedicated health bar for precise tracking. This UI suite is rounded out by a "Settings" pane, granting the user manual control over the Camera2D properties—such as zoom level and target offsets—and a granular audio mixer to independently balance the background music against the localised sound effects for throwing, hits, and death.

Despite the project's successes, significant limitations were observed within the AI-assisted asset pipeline, particularly concerning the reliability of the image generation process. A recurring issue involved the model's tendency to embed unsolicited text into visual assets. Furthermore, the system frequently failed to acknowledge these additions when queried for refinement. Additionally, the model demonstrated a lack of contextual persistence, occasionally providing irrelevant imagery that bore no relation to established visual themes or previous prompts within the same session. The most prominent technical hurdle occurred during the generation of 2D sprites using the "Nano Banana" feature. While the Gemini 3.0 Pro model consistently reported that it had successfully rendered PNG files with transparent backgrounds, the resulting assets remained intact with solid backdrops. This necessitated manual intervention to achieve the transparency required for the game engine's rendering layers.

To streamline this resolution, a custom HTML-based utility was synthesised within the model's Canvas mode, designed specifically to remove white backgrounds from sprite sheets. This workflow, adapted from methodologies documented by the YouTuber *AsapGuide*, proved essential in restoring the intended transparency and ensuring visual cohesion across the GhostInvaders scenes [72]. Figure 4.9 illustrates the prompt engineering used to generate this tool, while Figure 4.10 displays the functional application utilised to rectify the asset hallucinations.

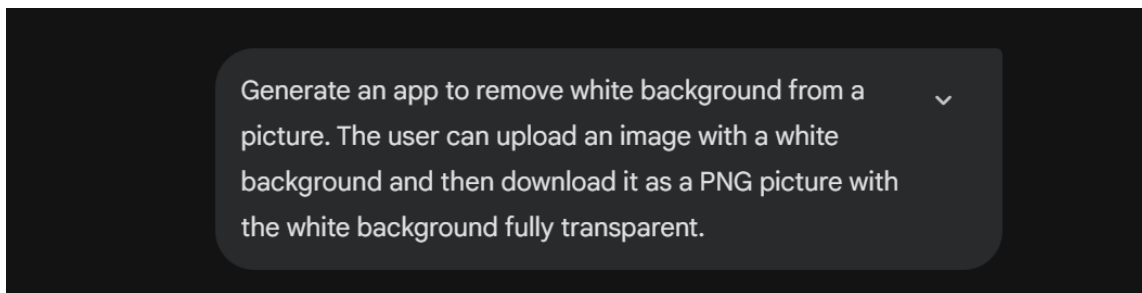


Figure 4.9: Initial prompt specifying requirements for the HTML-based background removal utility.

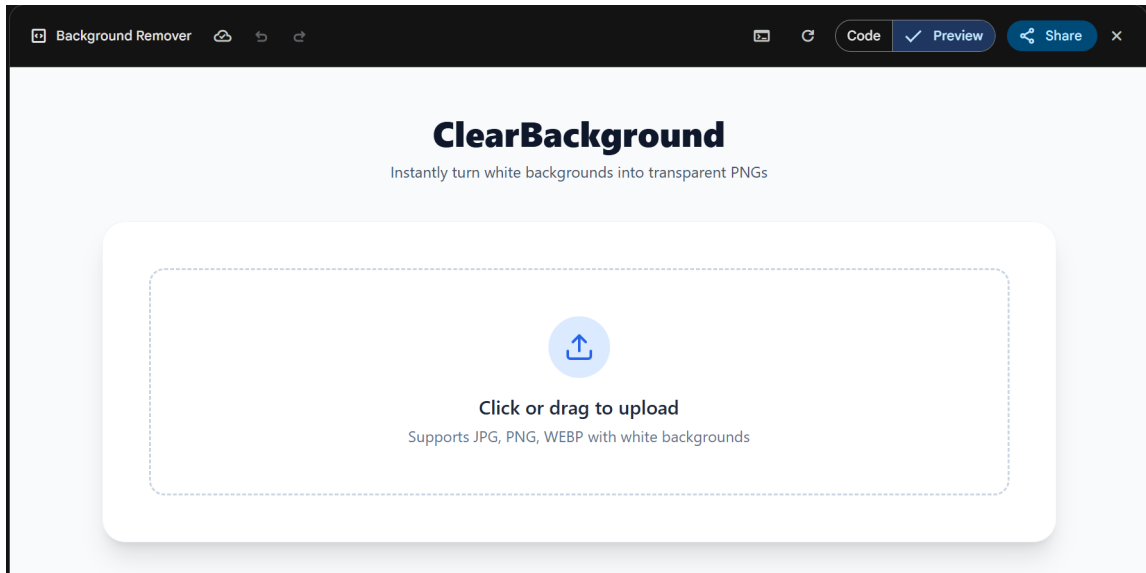


Figure 4.10: Functional HTML utility generated by Gemini 3.0 Pro for background removal.

5 Evaluation

To evaluate the effectiveness of the engine and the games produced, a user study was conducted involving twelve participants. Each individual was tasked with playtesting both Pac-Witch and GhostInvaders before completing a structured questionnaire. This survey consisted of eight questions designed to gauge various aspects of the user experience, requiring participants to rate specific gameplay and technical elements on a predefined scale.

Participants were first asked to select their preferred game. Interestingly, the feedback was evenly divided, with six participants opting for Pac-Witch (3D) and six for GhostInvaders (2D) as shown in Figure 5.1. This symmetry suggests that the core engine features—such as input responsiveness and resource management—scale effectively across different graphical perspectives, resulting in a balanced user experience regardless of the project’s dimensional complexity.

Which game did you like more?

12 responses

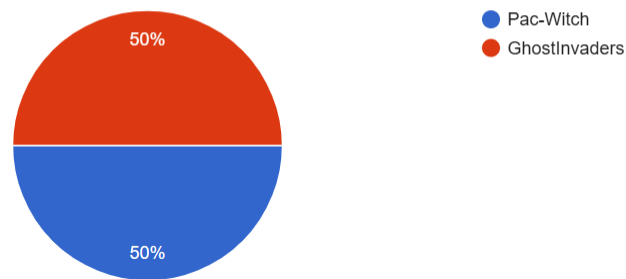


Figure 5.1: Pie chart illustrating user preference data collected from the initial questionnaire.

The second question evaluated the "fun factor" of Pac-Witch, the engine’s primary 3D project. As illustrated in Figure 5.2, the game received an impressive average rating of 4.08 out of 5 Stars. With 83.3% of participants awarding a score of 4 or higher, the data suggests that the core 3D gameplay loop—supported by the engine’s real-time collision detection and scene management—was highly engaging. This positive reception indicates that the technical integration of 3D assets and input handling successfully translated into a compelling user experience. This high level of engagement is further supported by the data in Figure 5.3, which assesses the technical quality of the implementation. When asked to rate how well-programmed the game felt—considering smoothness, performance, and a lack of bugs—75% of testers provided a rating of 4 or 5. Crucially, no participant reported a score below 3, indicating a high level of technical stability. These results suggest that the engine’s 3D subsystems, particularly the rendering pipeline and collision logic, provided a

sufficiently robust foundation to maintain immersion without the interruption of technical glitches.

Rate the overall fun factor of Pac-Witch.

12 responses

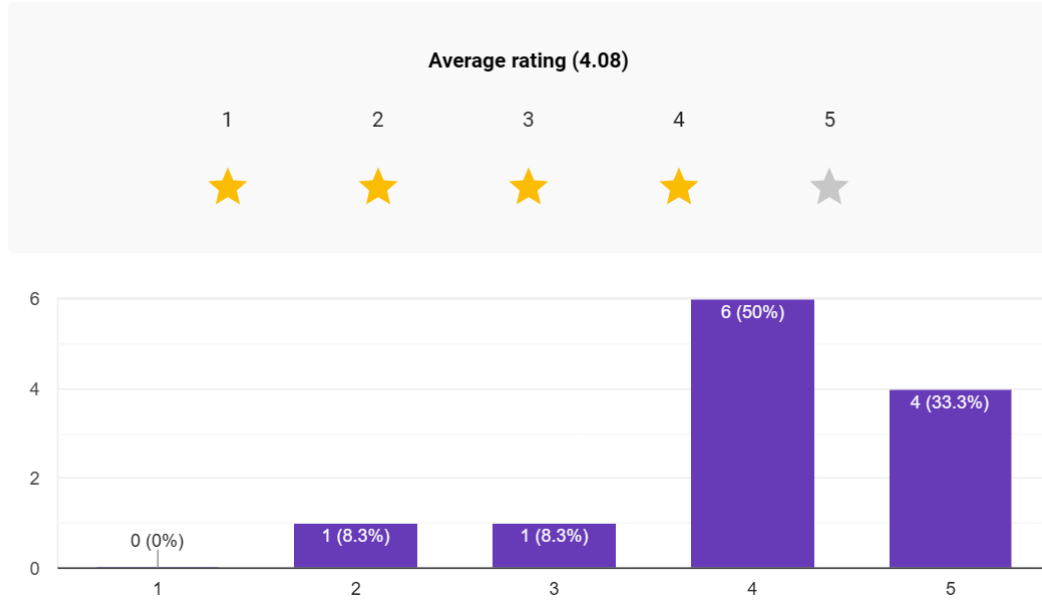


Figure 5.2: Participant ratings regarding the overall "fun factor" of the Pac-Witch 3D environment.

How well programmed was Pac-Witch? (Smoothness, lack of bugs, performance)

12 responses

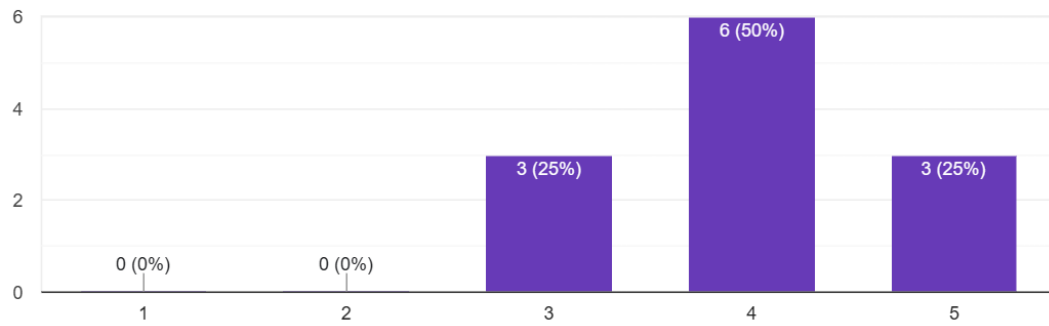


Figure 5.3: Survey results assessing the perceived programming quality, including smoothness and the absence of bugs, for Pac-Witch.

Following the assessment of the 3D environment, the evaluation shifts to the engine's 2D performance through GhostInvaders. The subsequent questions examine the user experience and technical execution of this game, providing a comparative look at how the 2D subsystems were received by the participants. As illustrated in Figure 5.4, the game achieved an average "fun factor" rating of 3.67. While the distribution was more varied than Pac-Witch, 50% of participants still awarded a high score of 4 or 5. This suggests that the 2D gameplay loop—driven by the engine's sprite animation and orthographic camera systems—was successful, albeit with room for further mechanical refinement. Furthermore, the technical execution of the 2D subsystems is further illustrated in Figure 5.5, which focuses on the programming quality of GhostInvaders. The results demonstrate a high degree of stability, with 66.6% of participants awarding a score of 4 or higher for smoothness and the absence of bugs. This confirms that the engine's 2D rendering pipeline and collision logic provided a robust and reliable user experience.

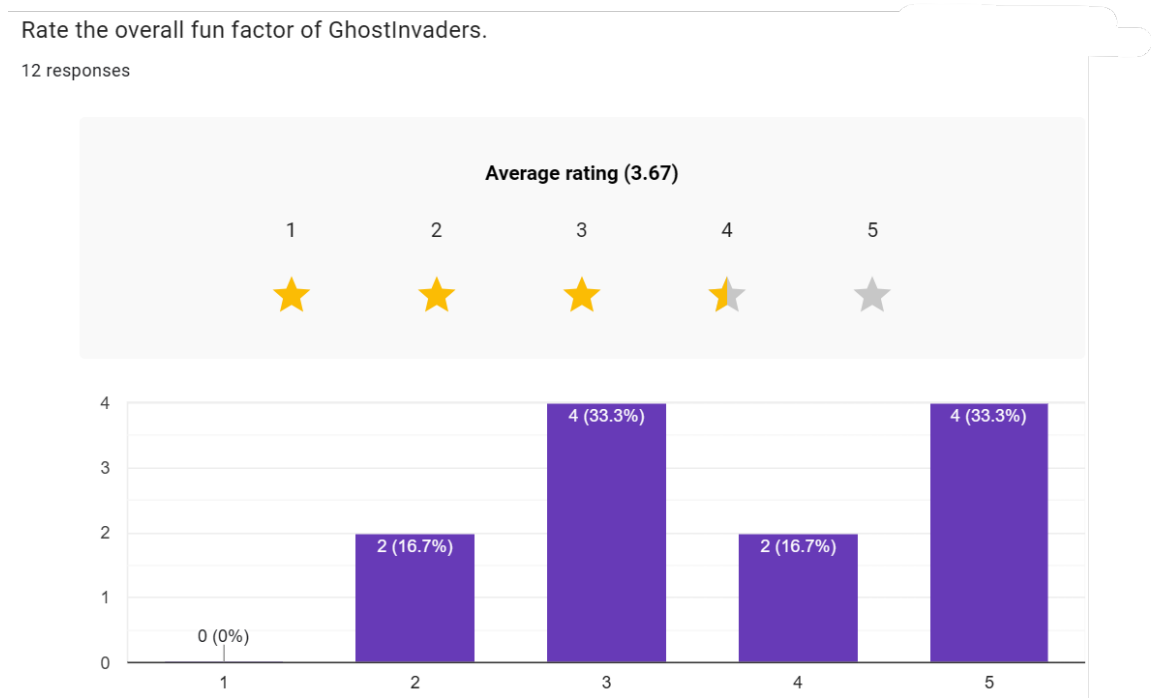


Figure 5.4: Quantitative survey data illustrating the perceived entertainment value of the GhostInvaders prototype.

How well programmed was GhostInvaders? (Smoothness, lack of bugs, performance)

12 responses

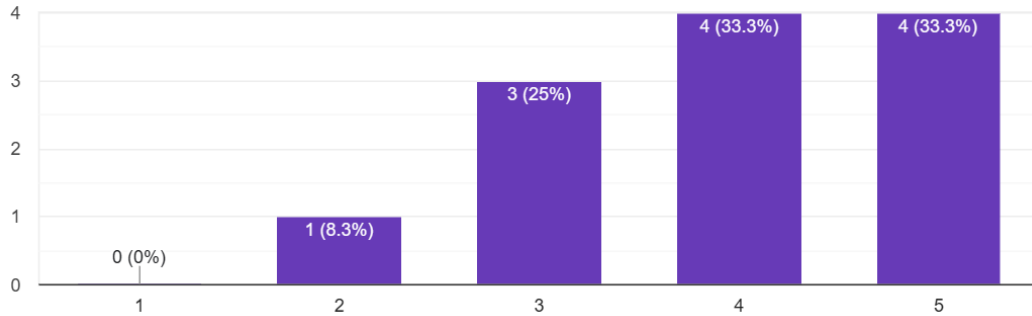


Figure 5.5: User study data illustrating the reported technical reliability of the 2D implementation.

A comparative analysis of the specific qualitative metrics for Pac-Witch (Figure 5.6) and GhostInvaders (Figure 5.7) reveals consistent performance across both dimensions. In terms of "Graphics and Visuals", both games saw their highest concentration of scores in the "Good" (3) to "Excellent" (4) range. This validates the engine's ability to render 3D meshes and 2D sprites effectively, confirming that the shader pipelines and texture mapping systems functioned as intended. The "Sound Effects and Music" followed a parallel trend, with "Good" (3) being the most frequent rating for both games. This suggests that the generated audio assets met participant expectations for clarity and atmosphere, justifying the scripted approach to asset production. Regarding "Gameplay Mechanics", Pac-Witch exhibited a more polarised response, with a higher number of "Fair" (2) and an equal number of "Outstanding" (5) and "Good" (3) ratings. This variance likely stems from the added complexity of 3D spatial navigation, which some users found intuitive while others required more fine-tuned sensitivity. Conversely, GhostInvaders maintained a more centralised "Good" (3) average. This indicates that the 2D mechanics provided a consistently stable and well-understood experience, benefiting from the inherent simplicity of the 2D plane. Overall, the data demonstrates that while both games were technically sound, the 3D environment of Pac-Witch provoked a more varied emotional response from players, whereas the 2D framework of GhostInvaders offered a reliable baseline for the engine's functionality. The high frequency of positive ratings across both games serves as a robust validation of the engine's cross-dimensional architecture, proving its reliability under diverse graphical loads. Furthermore, the lack of significant technical grievances indicates that the custom subsystems—from input handling to real-time rendering—achieved a commendable level of optimisation. Ultimately, the successful deployment of these two distinct projects demonstrates that the engine provides a versatile and stable foundation for rapid game development using

AI-assisted asset pipelines.

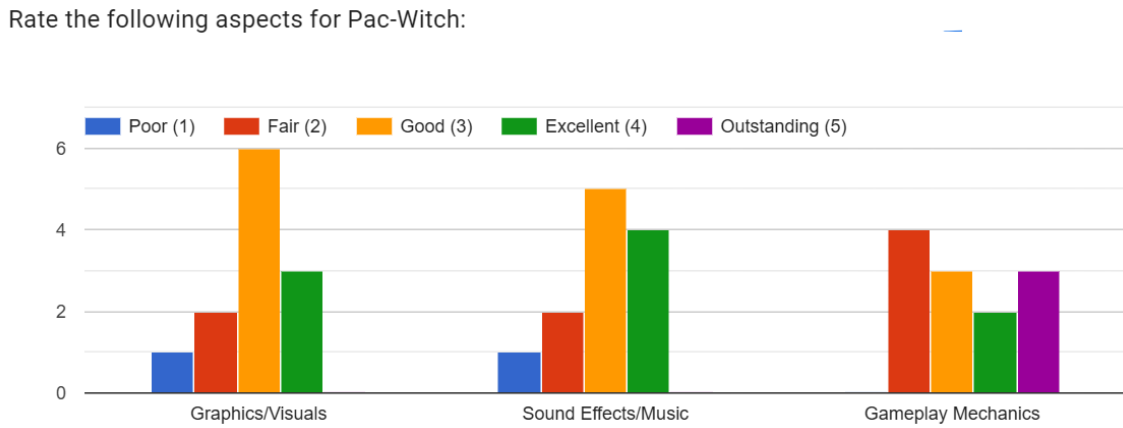


Figure 5.6: Comparative participant ratings for the visual, auditory, and mechanical aspects of the Pac-Witch game.

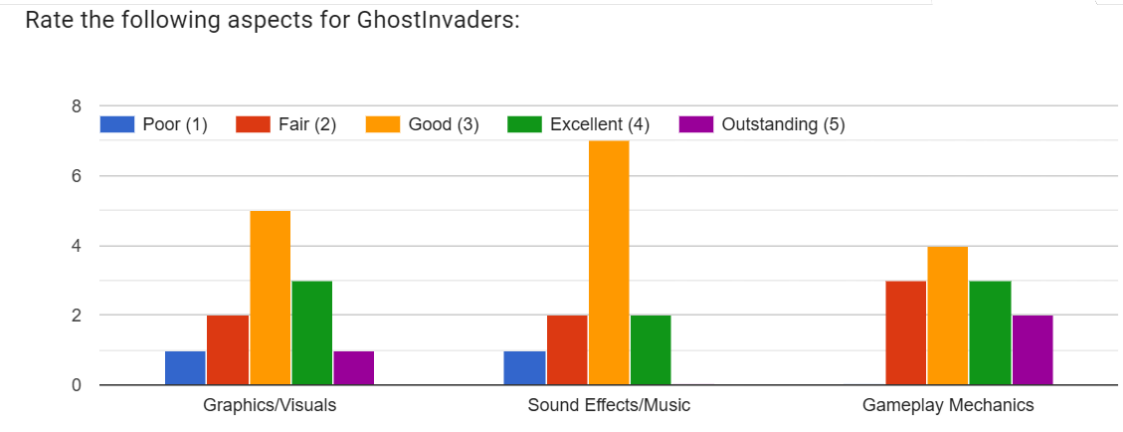


Figure 5.7: Comparative participant ratings for the visual, auditory, and mechanical aspects of the GhostInvaders game.

The final question presented to the 12 participants aimed to evaluate the overall quality of the multimedia assets produced through Gemini 3.0 Pro, including the music, sound effects, 3D objects, and textures used across both games. As shown in Figure 5.8, the results were overwhelmingly positive; 50% of respondents rated the asset quality as "Good" (3), while the remaining 50% awarded it even higher scores of 4 or 5. Significantly, no participant provided a rating below 3, indicating that the assets generated—despite being produced programmatically via Python scripts—met or

exceeded the expected standards for a prototype environment. This data suggests that utilising Gemini 3.0 Pro for asset generation is a viable methodology for indie developers or engine testers looking to achieve visual and auditory cohesion without a dedicated art team.

How would you rate the overall quality of the assets (Music, Soundeffects, objects, textures and images) generated by Gemini 3.0 Pro?

12 responses

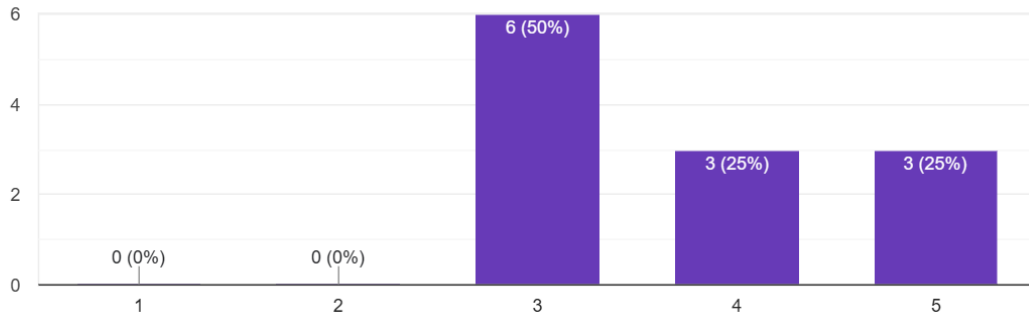


Figure 5.8: Participant ratings assessing the overall quality of multimodal assets (audio, geometry (3D & 2D), and textures) generated by Gemini 3.0 Pro.

6 Conclusion

This project has successfully demonstrated that Gemini 3.0 Pro is capable of independently generating a functional, modular game engine architecture when integrated with existing low-level rendering libraries like Raylib. Through an iterative prompting methodology, the project translated high-level natural language concepts into a robust C++ framework, validating the hypothesis that modern LLMs can act as autonomous architectural partners in complex software engineering tasks. The development and deployment of Pac-Witch and GhostInvaders served as a vital stress test, proving that the engine provides a stable foundation for both 2D and 3D interactive environments. Furthermore, the project successfully navigated the current limitations of LLMs—such as the inability to natively export 3D meshes or audio files—by utilising the model to author programmatic workarounds in Python.

The success of this simple engine raises significant questions regarding the future of software development and the potential for a "wrapper-less" architecture. While this project relied on Raylib for rendering and Dear ImGui for interface management, the rapid advancement of AI reasoning suggests a shift toward total system autonomy. A critical next step in this research is to investigate whether an AI can generate a renderer from scratch, effectively removing the dependency on external libraries. While Raylib currently provides the essential OpenGL abstraction required for hardware communication, there is significant potential for future models to directly author the Vulkan or DirectX boilerplate required for GPU communication. Similarly, the reliance on ImGui for tooling could eventually be superseded by Generative UI, where the AI procedurally designs and injects project-specific debugging suites without the need for fixed C++ libraries. Ultimately, this project confirms that the developer's role is transitioning into a curator of AI-generated content. The journey from a natural-language prompt to a fully functional game engine is no longer a theoretical possibility but a practical reality, laying the groundwork for a new era of AI-driven software creation.

References

- [1] X. Chen, “Research on artificial intelligence-assisted game design and development,” in *Proceedings of the 2nd International Conference on Data Science and Engineering (ICDSE 2025)*, SCITEPRESS – Science and Technology Publications, Lda., 2025, pp. 679–683, ISBN: 978-989-758-765-8. Accessed: Oct. 23, 2025. [Online]. Available: <https://www.scitepress.org/publishedPapers/2025/137044/pdf/index.html>.
- [2] S. Johnson and D. Hyland-Wood, *A primer on large language models and their limitations*, Dec. 2024. DOI: 10.48550/arXiv.2412.04503. arXiv: 2412.04503. Accessed: Nov. 2, 2025. [Online]. Available: <https://arxiv.org/html/2412.04503v1>.
- [3] A. Brennen, *The ultimate guide to LLM reasoning (2025)*, Kili Technology Blog, Feb. 2025. Accessed: Nov. 2, 2025. [Online]. Available: <https://kili-technology.com/large-language-models-llms/llm-reasoning-guide>.
- [4] S. Minaee, T. Mikolov, and N. Nikzad, “Large Language Models: A Survey,” *arXiv preprint arXiv:2402.06196v3*, 2025. arXiv: 2402.06196v3 [cs.CL]. Accessed: Oct. 25, 2025. [Online]. Available: <https://arxiv.org/html/2402.06196v3>.
- [5] K. Kavukcuoglu, *Gemini 2.5: Our most intelligent ai model*, The Keyword (Google Blog), Mar. 2025. Accessed: Oct. 26, 2025. [Online]. Available: <https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/#gemini-2-5-pro>.
- [6] O. Peña-Cáceres, E. Garay-Silupú, D. Aguilar-Chuquizuta, and H. Silva-Marchan, “Research trends and networks in self-explaining autonomous systems: A bibliometric study,” *Tech Science Press: Computers, Materials and Continua*, vol. 84, no. 2, pp. 2151–2188, 2025. DOI: 10.32604/cmc.2025.065149. Accessed: Oct. 26, 2025. [Online]. Available: <https://www.techscience.com/cmc/v84n2/62902/html>.
- [7] R. He, J. Cao, and T. Tan, “Generative artificial intelligence: A historical perspective,” *National Science Review*, vol. 12, no. 5, May 2025. DOI: 10.1093/nsr/nwaf050. Accessed: Oct. 31, 2025. [Online]. Available: <https://doi.org/10.1093/nsr/nwaf050>.
- [8] M. Esposito et al., *Generative ai for software architecture. applications, challenges, and future directions*, version 2, 2025. DOI: 10.48550/arXiv.2503.13310. arXiv: 2503.13310. Accessed: Oct. 31, 2025. [Online]. Available: <https://arxiv.org/abs/2503.13310v2>.
- [9] A. Brennen, *Agentic frameworks: The complete guide to the systems used in building autonomous agents*, Moveworks Blog, Feb. 2025. Accessed: Nov. 2, 2025. [Online]. Available: <https://www.moveworks.com/us/en/resources/blog/what-is-agentic-framework>.
- [10] Z.-Z. Li et al., *From system 1 to system 2: A survey of reasoning large language models*, Feb. 2025. DOI: 10.48550/arXiv.2502.17419. arXiv: 2502.17419. Accessed: Nov. 2, 2025. [Online]. Available: <https://arxiv.org/abs/2502.17419>.

- [11] Google, *Upload & analyze files in gemini apps - android - google help*, Google Help Center. Accessed: Nov. 2, 2025. [Online]. Available: <https://support.google.com/gemini/answer/14903178?hl=en&co=GENIE.Platform%3DAndroid>.
- [12] Google Cloud, *90% of games developers already using ai in workflows, according to new google cloud research*, Press Release, Aug. 2025. Accessed: Oct. 31, 2025. [Online]. Available: <https://www.googlecloudpresscorner.com/2025-08-18-90-of-Games-Developers-Already-Using-AI-in-Workflows,-According-to-New-Google-Cloud-Research>.
- [13] I. Lambe, *The NEW surprising number of Steam games that use GenAI*, Totally Human Media Blog, Jul. 2025. Accessed: Nov. 2, 2025. [Online]. Available: <https://www.totallyhuman.io/blog/the-surprising-new-number-of-genai-games-on-steam>.
- [14] A. Ternar, A. Denisova, J. o M. Cunha, A. Kultima, and C. Guckelsberger, *Generative ai in game development: A qualitative research synthesis*, Sep. 2025. arXiv: 2509.11898 [cs.HC]. Accessed: Oct. 31, 2025. [Online]. Available: <https://arxiv.org/abs/2509.11898v1>.
- [15] A. Chaudhary, *Top llm trends 2025: What's the future of llms*, Turing.com Blog, May 2025. Accessed: Nov. 2, 2025. [Online]. Available: <https://www.turing.com/resources/top-llm-trends>.
- [16] H. Chen et al., “Large language models and global health equity: A roadmap for equitable adoption in LMICs,” *The Lancet Regional Health - Western Pacific*, vol. 63, p. 101707, Oct. 2025. DOI: 10.1016/j.lanwpc.2025.101707. Accessed: Nov. 2, 2025. [Online]. Available: <https://pmc.ncbi.nlm.nih.gov/articles/PMC12556221/>.
- [17] A. Kostikova, Z. Wang, D. Bajri, O. Pütz, B. Paaßen, and S. Eger, *Llms: A data-driven survey of evolving research on limitations of large language models*, May 2025. DOI: 10.48550/arXiv.2505.19240. arXiv: 2505.19240. Accessed: Nov. 2, 2025. [Online]. Available: <https://arxiv.org/abs/2505.19240>.
- [18] A. D. Saei, M. Sharbaf, and S. Kolahdouz-Rahimi, “Large language models for game development: A mapping study on automated code generation,” in *First Workshop on Large Language Models For Generative Software Engineering (LLM4SE 2025)*, Koblenz, Germany, Jun. 2025. Accessed: Oct. 31, 2025. [Online]. Available: https://pure.roehampton.ac.uk/portal/files/31032276/STAF_2025_paper_54-3.pdf.
- [19] J. Ratican and J. Huston, “Adaptive worlds: Generative AI in game design and future of gaming, and interactive media,” *ISRG Journal of Arts, Humanities and Social Sciences*, vol. 2, no. 5, pp. 284–290, Sep. 2024, ISSN: 2583-7672. DOI: 10.5281/zenodo.13894497. Accessed: Oct. 31, 2025. [Online]. Available: <https://digitalcommons.lindenwood.edu/cgi/viewcontent.cgi?article=1693&context=faculty-research-papers>.
- [20] R. Lopez Mendez, *Generative ai in game development*, Arm Community Blog, Mar. 2025. Accessed: Nov. 2, 2025. [Online]. Available: <https://developer.arm.com/community/arm->

- community-blogs/b/mobile-graphics-and-gaming-blog/posts/generative-ai-game-development.
- [21] NVIDIA, *DLSS 4 technology*, Webpage, 2025. Accessed: Nov. 2, 2025. [Online]. Available: <https://www.nvidia.com/en-us/geforce/technologies/dlss/>.
- [22] NVIDIA, *Game developer conference (GDC) 2025*, Webpage, 2025. Accessed: Nov. 2, 2025. [Online]. Available: <https://www.nvidia.com/en-us/events/gdc/>.
- [23] Unity Technologies, *Unity AI: AI game development tools & RT3D software*, Webpage, 2025. Accessed: Nov. 2, 2025. [Online]. Available: <https://unity.com/products/ai>.
- [24] R. Barth, *The convergence of AI and creativity: Introducing Ghostwriter*, Ubisoft News, Mar. 2023. Accessed: Nov. 2, 2025. [Online]. Available: <https://news.ubisoft.com/en-us/article/7Cm07zbBGy4Xm16WgYi25d/the-convergence-of-ai-and-creativity-introducing-ghostwriter>.
- [25] SEED, *SEED applies ML research to the growing demands of AAA game testing*, Electronic Arts (SEED News), Aug. 2023. Accessed: Nov. 2, 2025. [Online]. Available: <https://www.ea.com/seed/news/seed-ml-research-aaa-game-testing>.
- [26] K. G. Nalbant and M. E. Yarar, “AI-powered game development: Intelligent systems for future gaming experiences,” *ResearchGate*, Jul. 2025, Preprint. DOI: 10.21203/rs.3.rs-7177087/v1. Accessed: Oct. 31, 2025. [Online]. Available: https://www.researchgate.net/publication/396236019_AI-Powered_Game_Development_Intelligent_Systems_for_Future_Gaming_Experiences.
- [27] C. Hu, Y. Zhao, Z. Wang, H. Du, and J. Liu, *Games for artificial intelligence research: A review and perspectives*, Apr. 2023. DOI: 10.48550/arXiv.2304.13269. arXiv: 2304.13269 [cs.AI]. [Online]. Available: <https://arxiv.org/html/2304.13269v4>.
- [28] E. Popp and H. Hlavacs, “AI-Driven Web Game Development with Gemini 2.5 Pro,” in *Proceedings of the International Conference on Information Integration and Web-based Applications & Services (iiWAS 2025)*, Matsue, Japan, Dec. 8–10, 2025.
- [29] Microsoft Research, *MaaG: A new framework for consistent AI-generated games*, Microsoft Research Blog, Jun. 2025. Accessed: Nov. 2, 2025. [Online]. Available: <https://www.microsoft.com/en-us/research/articles/maag-a-new-framework-for-consistent-ai-generated-games/>.
- [30] A. S. Vezhnevets et al., *Multi-actor generative artificial intelligence as a game engine*, Jul. 2025. DOI: 10.48550/arXiv.2507.08892. arXiv: 2507.08892. Accessed: Nov. 2, 2025. [Online]. Available: <https://arxiv.org/pdf/2507.08892>.
- [31] Google Cloud, *Tensor processing units (TPUs)*, Webpage, 2025. Accessed: Nov. 2, 2025. [Online]. Available: <https://cloud.google.com/tpu>.

- [32] D. Valevski, Y. Leviathan, M. Arar, and S. Fruchter, *Diffusion models are real-time game engines*, Aug. 2024. DOI: 10.48550/arXiv.2408.14837. arXiv: 2408.14837. Accessed: Nov. 2, 2025. [Online]. Available: <https://arxiv.org/pdf/2408.14837>.
- [33] A. Dev, *Gemini 3 pro: A comprehensive analysis of google's advanced multimodal ai*, 2025. Accessed: Jan. 13, 2026. [Online]. Available: <https://atoms.dev/insights/gemini-3-pro-a-comprehensive-analysis-of-googles-advanced-multimodal-ai/58ca5df7013541208505f466bbb9172a>.
- [34] Google DeepMind. "Gemini: The most capable and general ai model yet," Accessed: Jan. 13, 2026. [Online]. Available: <https://deepmind.google/models/gemini/pro/>.
- [35] Google DeepMind, "Gemini 3 pro external model card," Google, Tech. Rep., Dec. 2025. Accessed: Jan. 13, 2026. [Online]. Available: <https://deepmind.google/models/model-cards/gemini-3-pro>.
- [36] L. Kilpatrick. "Start building with gemini 3," Google, Accessed: Jan. 13, 2026. [Online]. Available: <https://blog.google/innovation-and-ai/technology/developers-tools/gemini-3-developers/>.
- [37] S. Pichai, D. Hassabis, and K. Koray. "A new era of intelligence with gemini 3," Accessed: Jan. 13, 2026. [Online]. Available: <https://blog.google/products-and-platforms/products/gemini/gemini-3/>.
- [38] A. Aryal, *Google's gemini 3.0 changes: What's new, what's improved, and why it matter*, Nov. 2025. Accessed: Jan. 13, 2026. [Online]. Available: <https://www.blankboard.studio/originals/blog/googles-gemini-3-0-whats-new-whats-improved-and-why-it-matter>.
- [39] APIYI, *Deep dive into gemini 3 pro preview: 7 major technical innovations and api integration guide for 2025's most powerful gemini model*, Nov. 2025. Accessed: Jan. 13, 2026. [Online]. Available: <https://help.apiyi.com/gemini-3-pro-preview-2025-ultimate-guide-en.html>.
- [40] Metana Editorial, "Gemini 3 vs. gemini 2.5: What are the main differences?" *Metana Blog*, Jan. 2026. Accessed: Jan. 13, 2026. [Online]. Available: <https://metana.io/blog/gemini-3-vs-gemini-2-5/>.
- [41] Vertu Lifestyle Editorial, "Gemini 3.0 vs gemini 2.5 pro: Google sets new performance standards in 2025," *Vertu Blog*, Nov. 2025. Accessed: Jan. 13, 2026. [Online]. Available: <https://vertu.com/lifestyle/gemini-3-0-vs-gemini-2-5-pro-google-sets-new-performance-standards-in-2025/>.
- [42] Google. "Gemini 3 developer guide," Accessed: Jan. 13, 2026. [Online]. Available: <https://ai.google.dev/gemini-api/docs/gemini-3>.

- [43] G. A. D. Community. “Gemini 3 not adhering to system prompts,” Accessed: Jan. 13, 2026. [Online]. Available: <https://discuss.ai.google.dev/t/gemini-3-not-adhering-to-system-prompts/110320>.
- [44] P. Schmid. “New gemini api updates for gemini 3,” Accessed: Jan. 13, 2026. [Online]. Available: <https://developers.googleblog.com/new-gemini-api-updates-for-gemini-3/>.
- [45] Google Cloud, “Gemini 3 pro | generative ai on vertex ai,” Google Cloud Documentation, Tech. Rep., Nov. 2025. Accessed: Jan. 13, 2026. [Online]. Available: <https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/3-pro>.
- [46] Camille, *Gemini 3 limitations: What it can't do (honest review)*, Nov. 2025. Accessed: Jan. 13, 2026. [Online]. Available: <https://skywork.ai/blog/llm/gemini-3-limitations/>.
- [47] r/GeminiAI Community, *Testing gemini 3.0 pro's actual context window in production: Results and needle-in-a-haystack analysis*, Nov. 2025. Accessed: Jan. 13, 2026. [Online]. Available: https://www.reddit.com/r/GeminiAI/comments/1q6viir/testing_gemini_30_pros_actual_context_window_in/.
- [48] G. Martinez, *4 reasons I chose Raylib over other game engines*, Atomic Spin, Dec. 2025. Accessed: Jan. 9, 2026. [Online]. Available: <https://spin.atomicobject.com/4-reasons-i-chose-raylib/>.
- [49] R. Santamaria, *Raylib - a simple and easy-to-use library to enjoy videogames programming*, 2026. Accessed: Jan. 9, 2026. [Online]. Available: <https://www.raylib.com/>.
- [50] JDBC, *Making games with raylib library as senior developer*, DEV Community, Dec. 2023. Accessed: Jan. 9, 2026. [Online]. Available: <https://dev.to/jdbcdev/raylib-library-for-video-games-programming-as-senior-developer-56jo>.
- [51] Wikipedia contributors, *Raylib — Wikipedia, the free encyclopedia*, 2025. Accessed: Jan. 9, 2026. [Online]. Available: <https://en.wikipedia.org/wiki/Raylib>.
- [52] R. Santamaria, *Raylib v5.5*, Nov. 2024. Accessed: Jan. 9, 2026. [Online]. Available: <https://github.com/raysan5/raylib/releases>.
- [53] GameFromScratch, *Raylib new software renderer*, GameFromScratch.com, Oct. 2025. Accessed: Jan. 10, 2026. [Online]. Available: <https://gamefromscratch.com/raylib-new-software-renderer/>.
- [54] R. Santamaria, *Raylib: A simple and easy-to-use library to enjoy videogames programming*, GitHub repository, 2026. Accessed: Jan. 10, 2026. [Online]. Available: <https://github.com/raysan5/raylib>.
- [55] Wikipedia contributors, *Gltf — Wikipedia, the free encyclopedia*, 2025. Accessed: Jan. 10, 2026. [Online]. Available: <https://en.wikipedia.org/wiki/GlTF>.
- [56] L. Salzman, *Inter-quake model (IQM)*, GitHub repository, 2026. Accessed: Jan. 10, 2026. [Online]. Available: <https://github.com/lsalzman/iqm>.

- [57] bzt, *Model 3d (M3D) format specification*, GitHub repository, 2026. Accessed: Jan. 10, 2026. [Online]. Available: <https://bztsrc.gitlab.io/model3d/#:~:text=Welcome%20to%20Model%203D!,%22%20height=%22300%22%3E>.
- [58] R. Santamaria, *Raylib FAQ — Frequently Asked Questions*, GitHub repository, 2026. Accessed: Jan. 10, 2026. [Online]. Available: <https://github.com/raysan5/raylib/blob/master/FAQ.md>.
- [59] O. Cornut, *Dear imgui: Bloat-free graphical user interface for c++ with minimal dependencies*, version 1.91.7, 2026. Accessed: Jan. 12, 2026. [Online]. Available: <https://github.com/ocornut/imgui>.
- [60] D. Rodriguez. “An introduction to the dear ImGui library,” Conan Blog, Accessed: Jan. 12, 2026. [Online]. Available: <https://blog.conan.io/2019/06/26/An-introduction-to-the-Dear-ImGui-library.html>.
- [61] C. Quinn. “Immediate UI vs retained UI,” Accessed: Jan. 12, 2026. [Online]. Available: <https://collquinn.gitlab.io/portfolio/my-article.html>.
- [62] Various Authors. “Why use dear ImGui for game engines if it’s not for complex apps?” Reddit, Accessed: Jan. 12, 2026. [Online]. Available: https://www.reddit.com/r/cpp/comments/1erq2vx/why_use_dear_imgui_for_game_engines_if_its_not/.
- [63] O. Cornut and Dear ImGui Contributors, *Software using Dear ImGui*, <https://github.com/ocornut/imgui/wiki/Software-using-dear-ImGui>, 2026. Accessed: Jan. 12, 2026.
- [64] O. Cornut. “About the IMGUI paradigm,” GitHub Wiki, Accessed: Jan. 12, 2026. [Online]. Available: <https://github.com/ocornut/imgui/wiki/About-the-IMGUI-paradigm>.
- [65] O. Cornut and L. Maggi. “Dear ImGui end-user API reference,” Accessed: Jan. 12, 2026. [Online]. Available: <https://pixtur.github.io/mkdocs-for-ImGui/site/api-ImGui/ImGui--Dear-ImGui-end-user/>.
- [66] O. Cornut and Skia Project Authors. “Dear ImGui documentation (skia mirror),” Google Source, Accessed: Jan. 12, 2026. [Online]. Available: <https://skia.googlesource.com/external/github.com/ocornut/imgui+/refs/heads/features/fnv1a/docs>.
- [67] Lecrapouille, ocornut, and JeffM2501. “Add raylib backend? · issue #6933 · ocornut/imgui,” GitHub, Accessed: Jan. 13, 2026. [Online]. Available: <https://github.com/ocornut/imgui/issues/6933>.
- [68] H. de Ruiter. “Building a cross-platform c++ gui app with cmake, raylib, and dear imgui,” Kea Sigma Delta, Accessed: Jan. 13, 2026. [Online]. Available: <https://keasigmadelta.com/blog/building-a-cross-platform-c-gui-app-with-cmake-raylib-and-dear-ImGui/>.
- [69] JeffM2501 et al., *Rlimgui: A raylib integration with dear imgui*, <https://github.com/raylib-extras/rImGui>, 2024. Accessed: Jan. 13, 2026.
- [70] JeffM2501 et al., *Rlimgui-cs: A raylib-cs integration with dear imgui*, <https://github.com/raylib-extras/rImGui-cs>, 2025. Accessed: Jan. 13, 2026.

- [71] A. V. Cintora and JeffM2501. “Rlingui doesn’t respect wantcapturemouse/wantcapturekeyboard · issue #63 · raylib-extras/rllmGui,” GitHub, Accessed: Jan. 13, 2026. [Online]. Available: <https://github.com/raylib-extras/rllmGui/issues/63>.
- [72] ASAPGuide, *How to remove background in gemini ai and download as png*, YouTube video, 04:41, May 2025. Accessed: Dec. 18, 2025. [Online]. Available: https://www.youtube.com/watch?v=uBMcmvJ_qRk.