# Conceptual Architecture of ScummVM

October 11, 2024

Queen's University
CISC322
Instructor: Prof. Bram Adams

Corey McCann 20344491 21ccm12@queensu.ca
Hayden Jenkins 20344883 21hjdm@queensu.ca
Ben Leray 202347460 21bl45@queensu.ca
Simon Nair 20297356 21scn1@queensu.ca
Jeremy Strachan 20336601 21js138@queensu.ca
Ryan Van Drunen 20331633 21rdbv@queensu.ca

# Table of Content

# 1. Abstract

This report serves to present a comprehensive view of the conceptual architecture in ScummVM. The analysis dives into its core functionality, design principles, and the relationships between different components to illustrate how they all work together to provide a nostalgic gaming experience. By examining the underlying conceptual architecture, we aim to highlight the flexibility and compatibility of ScummVM, as well as its ability to integrate various game engines into modern platforms and hardware. Additionally, the report discusses the challenges faced during development and the solutions implemented to ensure compatibility with a diverse range of classic games.

# 2. Introduction and Overview

ScummVM is an open-sourced software engine that enables users to play classic adventure and role-playing games on modern systems. Initially developed to be an emulator of the SCUMM engine, the project has since evolved over time to support an extensive library of 325 retro games. ScummVM supports iconic games such as *Broken Sword*, *Myst*, and *Blade Runner* among other titles from a wide array of revolutionary developers such as *LucasArts* and *Westwood Studios*. By preserving the flow and feel of these retro games, ScummVM ensures their legacy by making them playable on modern hardware without compromising the original experience.

Unlike most game engine emulators, ScummVM functions by translating the game's original executable file into code, allowing for games to be enhanced by modern features such as improved scaling algorithms and rendering techniques. This design choice has many advantages compared to traditional hardware emulation as it allows for ScummVM’s architecture to be modular and scalable. ScummVM operates through multiple specialized subsystems to perform platform-specific operations. This abstraction allows for multiple game engines to be run using the same application maintaining stability across a wide range of platforms. By investigating the functionality of the system, the evolution of its design, the flow of data, and the interactions of subsystems, we can better understand how ScummVM succeeds at preserving retro games.

# 3. Derivation Process

To derive the conceptual architecture of ScummVM, our team used a structured approach involving research, communication, and iterative refinement. We began by investigating various resources and splitting our efforts across the core aspects of the architecture and then worked together to align our findings into a cohesive conceptual model.

Finding documentation on the conceptual architecture of ScummVM was not an easy task. The best source of information we found was the ScummVM forum, where other

researchers shared their insights on specific game engines and detailed reports on the overall architecture of the software. These discussions provided us with an essential foundation for understanding the system's conceptual layers and the components involved.

To efficiently split the workload, we divided the architecture analysis into six main components, with each of us taking responsibility for one specific aspect: system functionality, system evolution, control, and data flow, concurrency, division of developer responsibilities, sequence diagrams, and their respective explanations. This distribution allowed each team member to deeply investigate their topic while ensuring broad coverage of the overall system architecture. Throughout the process, we maintained strong communication between team members whose topics overlapped, such as system functionality, control and data flow, concurrency, and diagrams, to maintain consistency across different parts of the report.

After completing our individual portions, we came together to combine our work into a single report. During this integration, we encountered some minor issues related to differing levels of detail across our components and subsystems. To address these concerns, we organized a meeting where we discussed and agreed on a unified set of core components that effectively conveyed everyone's contributions while maintaining a consistent level of abstraction throughout the document.

# 4. Architecture

## 4.1 Architecture Styles

Due to the size, scope, and purpose of ScummVM, a combination of architectural styles was required to conceptualize the project.

### 4.1.1 Layered Style

The developers used a layered style approach to organize the structure of the application into 4 distinct layers (Figure 1). The top layer is the User Interface Layer which handles tasks such as the GUI, application settings, and the game launcher. The Core Engine Layer handles tasks such as resource management, the virtual machine, and the kernel. Beneath the Core Engine Layer lies the Platform Abstraction Layer which deals with platform-specific operations making the application cross-platform compatible. Lastly, the Sub-System Layer deals with the components such as the game's audio and graphics, file management, and I/O subsystem.

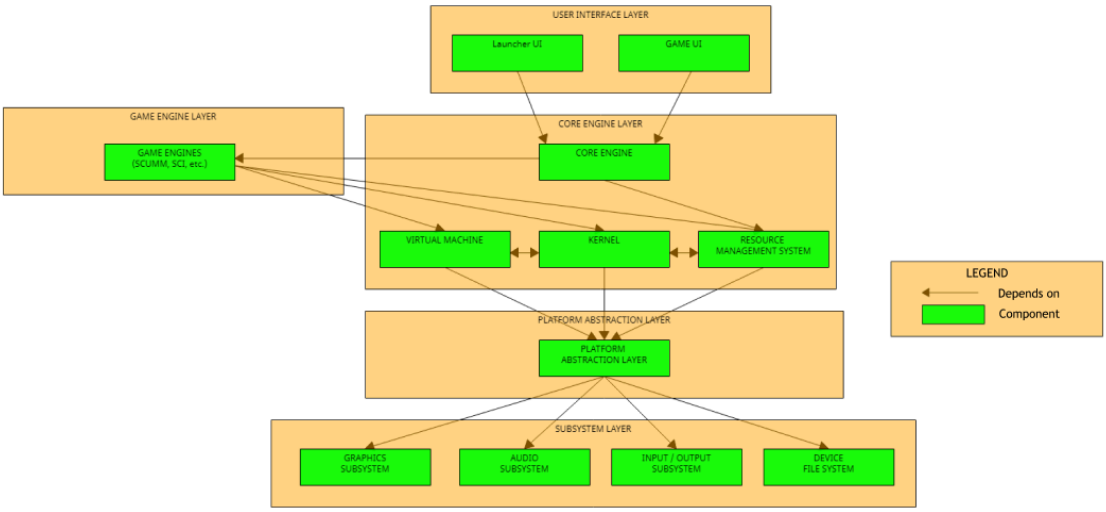


Figure 1: A box and line diagram of the conceptual architecture of ScummVM

### 4.1.2 Object-Oriented Style

An Object-Oriented Style was used to emphasize modularity, encapsulation, and polymorphism. The modularity of ScummVM allowed for functions such as sound handling and input management to be shared across different engines, simplifying maintainability and reducing duplicated code. Encapsulation was used to make adding new games and engines easier as each game engine was independent of others within the architecture. Polymorphism was used so that different game engines could implement their methods while maintaining a common interface, making the application easier to maintain and update.

### 4.1.3 Interpreter Style

Interpreter-Style architecture is also present as it enables the software to emulate different game engines without simulating the hardware underneath. The virtual machine built into the ScummVM interprets the scripting languages of old adventure games, translating the game's high-level scripts into actions, graphics, audio, and game state changes. This style allows ScummVM to simulate the logic of retro game engines on virtually any hardware and OS.

## 4.2 System Functionality

ScummVM is designed to run classic adventure games across multiple platforms. Its primary function is to replace the executable files of these games while using the original game assets, like graphics, sound, and game data. Instead of emulating old hardware, it translates the original game logic into modern code that can run across various platforms, such as Windows, macOS, Linux, iOS, Android, and even hand-held devices and consoles, such as Nintendo DS and 3DS, and various Playstation consoles.

The system is broken down into 7 interacting parts; resource management, virtual machine, game engines, graphics subsystem, audio subsystem, user interface subsystem, and platform abstraction layer. This section focuses on each component, their responsibilities, and how they interact with other components.

### 4.2.1 Core Engine

The core engine is the main orchestrator for the system and contains and manages the RMS and VM subsystems. This is the foundational layer, interfacing directly with the operating system, and offering services used by individual game engines through the resource management system.

### 4.2.2 Resource Management System

The resource management system is part of the core engine and is responsible for managing game resources like file I/O, graphics rendering, sound playback, input handling, and memory management. It interacts with the platform abstraction layer to provide abstraction for accessing different file formats.

### 4.2.3 Virtual Machine

The virtual machine is the heart of each engine where the game logic runs. It is responsible for interpreting game scripts, managing game states, and executing the game's logic. This is one of ScummVM's unique traits. It uses virtual machines to translate the original game logic into modern code, rather than emulating the hardware it was made for.

### 4.2.4 Game Engines

Each supported game, or group of games, runs on a specific game engine. Some examples of these engines are SCI, Kyra, SCUMM, and Wintermute. Each engine implements specific script interpreters, custom resource handling, and game-specific logic. The game engine layer represents any of the supported game engines that are used.

### 4.2.5 Graphics Subsystem

The graphics subsystem is responsible for rendering the game's graphics, handling different graphic formats, and managing animations. It also provides scaling and filtering options. All these responsibilities are platform-specific. The game engine calls upon the graphics subsystem to render frames. The platform abstraction layer allows ScummVM to support a wide range of graphical styles and resolutions across many different platforms.

#### 4.2.6 Audio Subsystem

The audio subsystem is responsible for managing sound effects and music playback, such as MIDI or digital audio. The subsystem supports various audio formats and systems, with unique scripts for each. The game engine requests audio services from this subsystem to play sounds, from MIDI music to digitized voices.

#### 4.2.7 User Interface Subsystem

The user interface subsystem is responsible for the main launcher interface, in-game GUI elements, as well as save and load functionality, and global options management. Developers integrate ImGui, a popular user interface library for C++, into each game engine for the use of UIs.

#### 4.2.8 Platform Abstraction Layer

The platform abstraction layer provides consistent API endpoints across different platforms, and handles platform-specific implementations. It's also responsible for handling file I/O, audio output, graphics rendering, and input handling. This layer ensures consistent behavior across all platforms.

#### 4.2.9 Kernel

The kernel is responsible for providing low-level functions that scripts rely on to interact with the game's system. It works with the RMS to handle fundamental operations such as rendering graphics, playing sounds, managing memory, and loading resources.

#### 4.2.10 Device File System

The device file system is located in the user's local machine and is accessed by the PAL and is responsible for file management and the organization of all games and their respective files and assets.

#### 4.2.11 I/O Subsystem

The I/O subsystem, or Input/Output subsystem, is responsible for any I/O heavy operations, such as handling input from the user. The PAL is dependent on the I/O subsystem component.

#### 4.2.12 Component Interaction

For the generic use of ScummVM, the key components interact in the following way. To start, the user opens the graphical launcher interface from where they will select a function like starting a game or adding a game. The core engine acts as the orchestrator for the whole system. For the case of running a game, it will initialize the appropriate game-specific engine for the

selected game. The game engine then loads the required graphics, audio, and other resources from the Resource Management System. The Virtual Machine then begins executing the game scripts. While the game is running, an event system is used to handle user input and different game events throughout the system. The graphic and audio subsystems when called render output based on the game logic executed by the specific game engine on the VM or by event calls. Throughout this process, the Platform Abstraction Layer ensures consistent handling of input and other platform-specific features that are dependent on the I/O or Device File system. The request flow of resources on the local computer is managed, executed, and controlled by the RMS, which is triggered through script execution in the kernel.

In summary, ScummVM's architecture is a balance of modularity, abstraction, and extensibility, where individual game engines plug into a common framework, allowing each part of the system to focus on specific tasks without overlapping or duplicating responsibilities. This system allows new game engines to be created without modifying core components.

## 4.3 System Evolution

The origins of ScummVM are traced back to the early 2000s when a Swedish computer science student named Ludvig Strigeus started a personal project to run *Monkey Island 2: LeChuck's Revenge* on his modern computer. Originally, the primary focus of the project was to reverse-engineer SCUMM-based games so that they were playable on different platforms. ScummVM has undergone significant conceptual architecture evolution since its inception and has expanded over the years to support multiple game engines and classic games that are no longer officially supported or accessible on modern platforms. The history of ScummVM can be divided into 4 main sections.

### 4.3.1 Initial Architecture (2001 - 2003)

The project was initially developed to emulate the SCUMM game engine used by LucasArts to create their adventure games. Since the original architecture was tightly tied to the SCUMM data structure, only games running on the SCUMM engine could be emulated and interpreted. ScummVM had the following core components at its inception:

Core Engine Support: SCUMM engine support with a focus on running LucasArts games.

Basic Features: Early versions were focused on engine functionality rather than cross-platform support or GUI improvements.

### 4.3.2 Expansion to multiple engines (2003 - 2004)

Due to the open-source nature of the software, the project quickly gained popularity among the gaming community. Other developers contributed and support for additional game engines was added. Leading to a shift in architecture:

Engine Integration: ScummVM was redesigned to support non-SCUMM engines, requiring flexible emulation layers. The architecture became more modular which allowed the addition of more complex engines and the ability to interpret additional game formats.

Game Data Abstraction: ScummVM decoupled game logic from game assets, which provided better compatibility with different game file formats, reduced the memory footprint, and simplified maintenance.

#### 4.3.3 GUI and Portability Improvements (2005-2009)

A push to make the application more user-friendly meant that ScummVM's launcher became simpler, allowing users to launch their games from a single application regardless of the game engine. This further separated the game engine from the user interface.

GUI Overhaul: The addition of a robust GUI improved the user experience, allowing for easier navigation and saved game settings.

Multi-Platform Support: ScummVM added support for more operating systems including Windows, MacOS, Linux, iOS, and Android.

#### 4.3.4 Engine Expansion and Modularization (2010 - Present)

ScummVM's architecture became more sophisticated as it added a wider range of games and platforms. Improvements on graphics, sound quality, and performance were added to make the games run smoother and more efficiently:

Improved Graphics: Advanced rendering tools, memory management systems, and improved scaling algorithms were added to enhance the visuals of classic games.

Modern OS Integration: ScummVM's architecture evolved to take advantage of modern operating system features, such as multi-threading and 64-bit support.

Further Engine Abstraction: Audio, graphics, and input handling were further abstracted to allow for different game engines to share common entities, thus reducing redundancy. This made ScummVM more efficient, portable, and scalable.

The conceptual architecture of ScummVM has shifted from a SCUMM specific emulator to a highly modular interpreter capable of supporting multiple engines and a variety of platforms. Over the last 2 decades, through consistent updates and improvements, the project has expanded its technical capabilities while remaining faithful to the preservation of retro games.

## 4.4 Control and Data Flow

In this section, we will be discussing the control and data flow among the components of the system. There are two important segments that we want to focus on for this part. What happens during the initialization of the program and what happens during the execution of a game?

First, we want to focus on the control flow of the system during the initialization of the program. The first part of the control flow is the initialization of the core engine. The core engine contains and manages the RMS and also interfaces directly with the operating system. This is all needed to prepare the environment for running the games. Next, the core engine processes command-line arguments, determining how the application should be launched. This gives the user the ability to start a game directly or specify some additional options. Next, and one of the more important parts of the control flow is the actual GUI and Launcher initialization. If a game is not specified directly via the command line then a graphical launcher is presented. The launcher is what allows the users to select which game to play. The launcher also acts as the main control interface until a game is started. The next part of the control flow is the loading of a game engine. Once a game is chosen, the corresponding game engine is loaded. The core engine decides which engine should be used based on the selected game. Once the correct engine is identified, it is initialized and the engine takes over control of the system. Finally, the last part of this control flow is what I am going to call the "loop for launcher and game execution". If a user has successfully selected a game and the game engine is initialized then control is passed to the game engine and the game execution loop becomes responsible for managing the actual gameplay which we will get to later. If a user decides to quit their game and return to the launcher, control flow is returned to the launcher and the system will remain there until the user selects a new game.

Next, we will discuss the data flow among the components during the initialization process. First, configuration details are read and loaded by the RMS to provide game settings and system configuration. This includes preferences for graphics audio and other game settings. Next, the launcher or command line provides the data required for the core engine to select the correct game engine. This flow of data ensures the proper association between user choices and internal engine setup. Finally, the user input from the launcher is passed from the GUI component to the core system, triggering the engine loading process.

Now we'll discuss the control and data flow during game execution. To do this we will take a high-level look at the Sci engine to give us a better idea of the control and data flow. Initially, the engine sets up the environment for the game to run. It does this by initializing the core components, such as graphics, audio, and scripts. The game loop is then run continuously and is responsible for processing user inputs, executing scripts, and updating graphics and audio. The game loop also manages events and updates on user actions such as mouse clicks. These events are fed through the platform abstraction layer and queued for the game engine to process and determine the proper scripts for the virtual machine to execute. The virtual machine then runs the appropriate game scripts based on the game state, which determines what actions need to be performed in the game world. The game world is divided into discrete "rooms", each representing a different area or scene in the game. Rooms are used to manage game logic, objects, events, and scripts specific to each location. When the player moves between rooms, the

game state updates to reflect the current room, and the corresponding room scripts are loaded and executed by the virtual machine. This allows everything to be kept organized and based on the player's current location. Script execution is not possible without the kernel, which provides low-level functions that scripts rely on to interact with the game's systems. The kernel, along with the RMS, handles fundamental operations such as rendering graphics, playing sounds, managing memory, and loading resources. Additionally, the Platform Abstraction Layer ensures that these operations are performed consistently across different hardware platforms, abstracting platform-specific details. This abstraction is what allows the game scripts to focus on high-level game logic rather than low-level implementation, which is executed using the built-in routines provided by the kernel.

## 4.5 Concurrency

ScummVM focuses on single-threaded execution to maintain support for older classic games that were designed to run on single-threaded platforms. However, some parts of the system use concurrency to improve performance and user experience.

The audio subsystem uses concurrency, running on a separate thread handling multiple audio streams and sound effects separate from the main game loop. This ensures smooth and uninterrupted audio playback even when complex and intensive tasks are being executed, such as graphics rendering or game logic processing. Using a separate thread ensures that sound remains continuous and stutter-free throughout gameplay.

Similarly, file I/O operations can use concurrency when loading larger game assets such as textures, sprites, and audio files. Many games supported by ScummVM are large, and their assets cannot be all loaded and stored in memory at once. Therefore, game engines dynamically load and unload assets during gameplay. ScummVM uses concurrency to manage this process, allowing assets to be fetched and loaded into memory in the background without blocking the main game loop. If assets were fetched and loaded in the main game loop it would lead to paused or interrupted gameplay as I/O operations are quite slow. However, the specifics of game asset retrieval and the use of concurrency is different between the different supported game engines.

User input is also another component where concurrency is used. ScummVM allows user inputs (such as mouse clicks and keyboard actions) to be detected and processed simultaneously. Using a separate thread to capture and process input allows the player's actions to be handled and reflected in near real-time even if the main game loop is busy. Having a separate thread processing events from the event queue is key to preventing player input from being missed or delayed and provides a responsive user experience that is normally only found in new modern game engines.

While ScummVM incorporates true concurrency in areas like audio and input, the original SCUMM system (which ScummVM emulates) used simulated concurrency through time-slicing. SCUMM simulates concurrency through time-slicing, executing multiple scripts (up to 20) “simultaneously” by sequentially executing each script until it reaches a defined break, at which point the next script is executed. This process is executed in a loop within the main game loop, giving the illusion of parallel execution, but is only sequentially executing different scripts one at a time. This time-slicing behavior allows for complex seemingly concurrent behaviors throughout the game.

## 4.6 Division of Responsibilities

When considering the multifaceted nature of ScummVM, the division of responsibilities among developers in the context of conceptual architecture proves complex. Many implications must be taken into account to effectively achieve a product that can seamlessly integrate with a wide variety of platforms for the extensive library of games. With ScummVM being now released to the public for over 23 years, the approach towards development has shifted from a timelined release to an open-source project available on the version control platform GitHub.

ScummVM takes on a two-role approach toward product development, with team members working on engine development or backend as outlined in their development guidelines. Each member is responsible for their area of development. This division of roles fosters an environment where members can focus on their area of expertise while maintaining best practices independently, such as continuous implementation, code commits, and collaboration for extended work. Documentation is strongly outlined to maintain a level of organization, as the program contains over 6 million lines of code. Should a developer proceed with a large change, appropriate file documentation is required and must be submitted following the protocols in place.

The team members focusing on engine development have a large emphasis on modularity as ScummVM’s architecture is intended to support multiple game engines. The program's modular design approach towards the various supported engines helps separate and establish clear teams to undertake particular engines so that they can be supported. The creation of specialized teams allows members to understand the different aspects and mechanics to implement each engine and hone in on their area of specialization. Engine development is expected to be done semi-independently while the guidelines detail how to “hook” it into the platform.

The team members on the backend will focus more on platform dependency as it pertains more to operating systems. Similar to engine development, a member's work is largely self-focused with only the approach to initial development differentiating. ScummVM already

supports a large collection of platforms through the SDL backend, however, new development has been focused on adding support for new currently unsupported systems.

## 4.7 Sequence Diagrams and Explanations

### 4.7.1 Game Startup

The game startup process in ScummVM begins when a user selects a game from the launcher interface. The Core Engine first initiates a detection process, analyzing the selected game to identify the appropriate game engine needed. During this phase, the system examines file signatures, directory structures, and known file patterns to accurately determine the game type. Once identified, the Core Engine loads the corresponding game engine plugin, which is responsible for handling game-specific behaviors and requirements.

After successful engine selection, the Resource Manager catalogs all available game resources, creating an internal mapping of resource identifiers to actual game assets. This process involves examining various file types, from graphics and audio assets to game scripts and data files. Critical resources needed for immediate gameplay are preloaded into memory to ensure smooth game initiation, with other game assets getting loaded during runtime as needed. The Resource Manager works through the Platform Abstraction Layer, which ensures that file access remains consistent across different operating systems and hardware configurations.

Simultaneously, the Graphics and Audio subsystems undergo initialization. The Graphics System determines and prepares various graphic modes that might be needed during gameplay. The Audio System initializes the necessary sound drivers and prepares audio channels for both music and sound effects.

The final stage of startup involves the Game Engine initializing the game state and preparing the Virtual Machine for script execution. The engine creates a fresh game state, setting up initial variables and conditions as specified by the game's design. The Virtual Machine, which is responsible for executing game scripts, prepares its script context and typically runs a startup script that initializes the game world. This might involve loading the initial room, placing characters in their starting positions, and initiating any beginning cutscenes or dialogues.

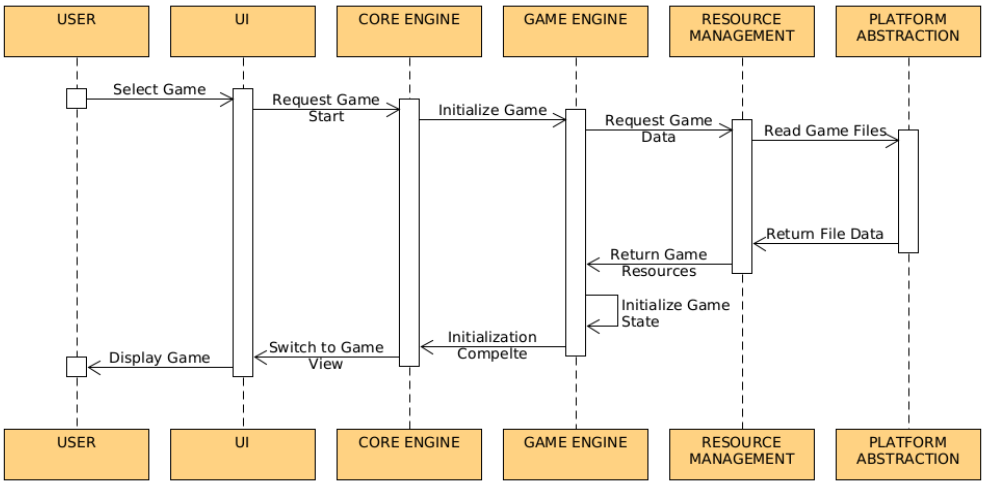


Figure 2: Sequence Diagram for the use case of a user launching a game.

### 4.7.2 Save Game

Upon receiving a save request, the Game Engine initiates a sequence of operations to capture the entire game state. The first step involves pausing the Virtual Machine to ensure that no scripts can modify the game state during the save process, guaranteeing data consistency.

Once the game is paused, the Game Engine begins the meticulous process of state collection. This involves gathering various types of data: the current values of all game variables, the state of every object in the game world, inventory contents, character positions, and any other game-specific data that needs to be preserved. The Virtual Machine also plays a crucial role here, as it must serialize its current execution state, including any scripts that are mid-execution and the current script call stack.

The Resource Manager is then involved, identifying all the resources that are currently in use or might be needed to correctly restore the game state later. This could include references to background images, sprite sheets, or audio files. The manager creates a comprehensive dependency list that will be stored alongside the game state, ensuring that when the game is loaded later, all necessary resources can be correctly reidentified and reloaded at a later time.

All of this collected data is then prepared for serialization. The system may employ various optimization techniques at this stage, such as data compression to reduce file size or delta encoding to only save changes from a known baseline state.

Finally, the actual writing of the save file is handled by the Platform Abstraction Layer. This layer ensures that the save operation is as safe as possible, often employing techniques like atomic writes to prevent data corruption in case of system crashes or power failures during the

save process. The save file is typically written to a temporary location first and then renamed to its final filename only after a successful write, providing an additional layer of safety.

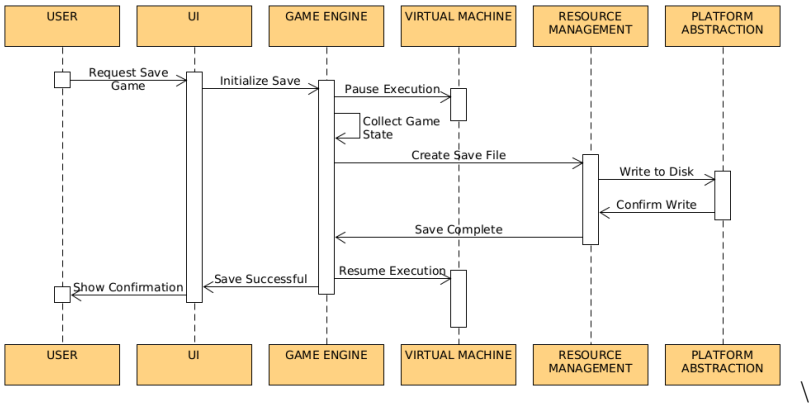


Figure 3: Sequence Diagram for the use case of a user saving a game.

### 4.7.3 Player Interaction

When a player performs an action, such as clicking the mouse or pressing a key, the Platform Layer first captures this raw input. It immediately translates these hardware-specific inputs into events that can be uniformly processed by the rest of the system. These standardized events are then queued for processing.

The Core Engine regularly processes this event queue as it is on a separate thread, forwarding events to the active Game Engine for handling. The Game Engine, being game-specific, understands the context of these events within the game world. It determines what type of response is required for each player action, often by mapping events to specific script triggers. This mapping might be direct, like clicking on an object triggering its interaction script, or more complex, like a combination of inputs triggering a special game event.

Once the appropriate response is determined, the Virtual Machine comes into play, executing the necessary game scripts. The VM executes these scripts instruction by instruction, with each instruction potentially affecting different parts of the game world or triggering other events that will be processed. It’s also possible that the VM will execute additional scripts as a side effect of the initial execution.

As the game state changes due to script execution, various subsystems are called upon to provide feedback to the player. The Graphics System updates the visual state of the game, which might involve redrawing backgrounds, animating sprites, or displaying new UI elements. Simultaneously, the Audio System handles any sound effects or music changes that result from the player's action. All these updates are carefully synchronized to provide seamless and immediate feedback to the player's input.

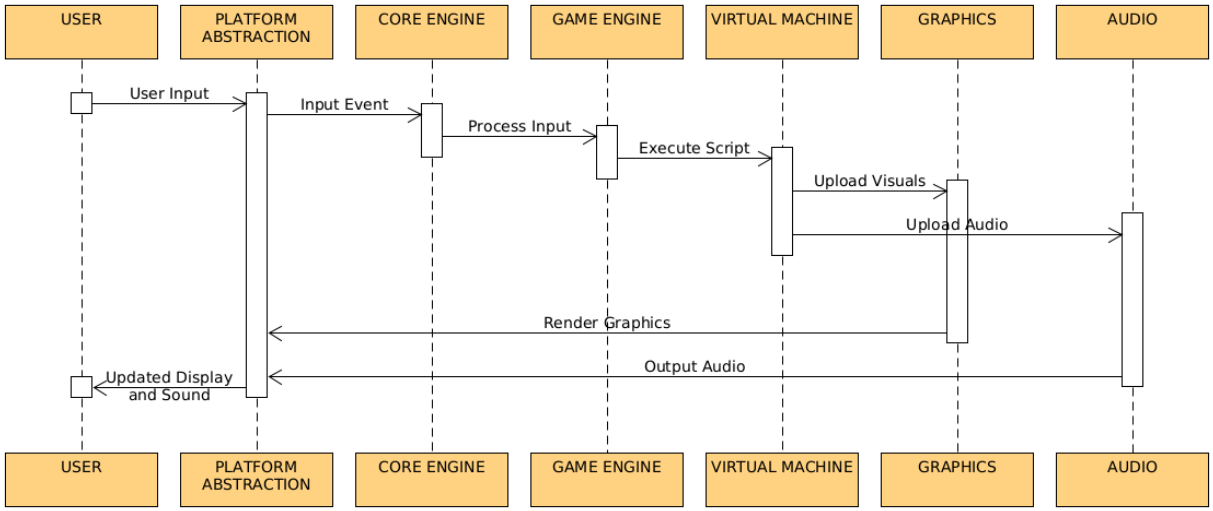


Figure 4: Sequence Diagram for the use case of a user interacting with a game.

# 5. External Interfaces

ScummVM interacts with several external interfaces to facilitate user interaction, data management, and system integration. The first key interface is the GUI, the GUI receives user inputs, such as mouse clicks and keystrokes, allowing users to interact with the game launcher, select game files, adjust settings, and control gameplay. These inputs are ultimately communicated to the core engine, which adds them to an event queue and processes them to execute corresponding actions. Another important external interface is within the local files and game data. ScummVM relies on user-provided game data files, including scripts, graphics, and audio, which are essential to run the games. These game files are loaded into the system and used by the core engine and virtual machine to recreate the original game experience within ScummVM. The platform abstraction layer provides an interface for interactions between the VM and the host operating system, handling tasks such as reading game data files, saving game progress, and managing configuration files. ScummVM also integrates with cloud storage services like Dropbox, OneDrive, Google Drive, and Box to enable synchronization of saved game states. These features are implemented through various APIs provided by the cloud storage providers, and allow users to download and upload game files and saved states to their cloud storage for easy game access from anywhere.

# 6. Data Dictionary

SDL: A cross-platform software development library designed to provide a hardware abstraction layer for computer multimedia hardware components

MIDI: Standard protocol used to communicate music-related data between electronic musical instruments, computers, and other audio devices

API: A set of protocols and tools that allows different software applications to communicate with each other by defining methods and data formats for interaction.

## 7. Naming Conventions

ScummVM - Script Creation Utility for Maniac Mansion Virtual Machine
SCUMM - Script Creation Utility for Maniac Mansion
RMS - Resource Management System
VM - Virtual Machine
PAL - Platform Abstraction Layer
UI - User Interface
GUI - Graphical User Interface
SDL - Simple DirectMedia Layer
I/O - Input / Output
OS - Operating System
MIDI - Musical Instrument Digital Interface
API - Application Programming Interface

## 8. Conclusions and Lessons Learned

Through multiple years of community development, ScummVM has ensured the longevity of classic games by making them playable on modern hardware without compromising on their original experience. The analysis of ScummVM's conceptual architecture has offered various insights regarding our research and development process as a team. Throughout the gathering and writing stage of the report, specific issues were encountered such as time management challenges and frequent refinement of models. The coordination and organization of tasks among group members proved crucial in meeting deadlines and expectations for the submission of work. With a limited production window, careful consideration was taken in the project's planning. Communication remained steady to ensure all parties were on the same page and up to date in their tasks. As the report progressed and new information was added, we had to frequently update and enhance our architecture model. These frequent changes introduced added complexity, as we had to ensure the seamless integration of new ideas while maintaining consistent terminology and levels of abstraction across the document.

# 9. References

*Developer central*. ScummVM. (n.d.-a). https://wiki.scummvm.org/index.php/Developer_Central
*Howto-backends*. ScummVM. (n.d.-b).
https://wiki.scummvm.org/index.php?title=HOWTO-Backends
*Howto-Engines*. ScummVM. (n.d.-c).
https://wiki.scummvm.org/index.php?title=HOWTO-Engines
Poulter, W. (1991). S.C.U.M.M. Tutorial. http://tandell.com/misc/SCUMM-Tutorial-0.1.pdf
quote58. (2022, July 28). *Conceptualizing the game engine*. quote58s GSoC blog.
https://blogs.scummvm.org/quote58/2022/07/28/conceptualizing-the-game-engine/
*ScummVM history*. ScummVM. (n.d.-d). https://wiki.scummvm.org/index.php/ScummVM_History
*Team onboarding*. ScummVM. (n.d.-e)
https://wiki.scummvm.org/index.php?title=Team_Onboarding
*Understanding the interface*. Understanding the interface - ScummVM Documentation documentation. (n.d.). https://docs.scummvm.org/en/v2.8.0/use_scummvm/the_launcher.html