Conceptual Architecture of the ScummVM Engine

Authors
Harry George - 20327293
Deniz Aydin - 20299521
Ben Hilderman - 20374738
Riley Spavor - 20218837
Kurtis Marinos - 20336452
Jacob Skiba - 20361187

Table of Contents

1.0. Abstract

This report presents our analysis of the conceptual architecture of ScummVM, including a breakdown of its major components, an evaluation of subsystem dependencies, and two sequence diagrams that illustrate critical interactions within the system. ScummVM comprises several essential modules, each playing a crucial role in meeting the functional and non-functional requirements of the system, such as supporting multiple game engines, providing cross-platform compatibility, and enabling seamless interaction between various subsystems. Its primary purpose is to provide a platform that allows users to run classic adventure and role-playing games across numerous modern and legacy operating systems by replacing the original game executables rather than emulating them.

2.0. Introduction

2.1. ScummVM

ScummVM (Script Creation Utility for Maniac Mansion Virtual Machine) is an open-source project that enables players to run classic adventure and role-playing games on modern platforms without emulation. This report analyzes ScummVM's conceptual architecture, focusing on its innovative approach to game preservation and cross-platform compatibility.

ScummVM's primary goal is to provide a unified interface for playing a diverse range of classic games from various developers and eras, effectively preserving gaming history while improving upon the originals with features like enhanced graphics scaling and modern control schemes.

At its core, ScummVM employs a modular architecture that separates game-specific logic from platform-specific implementations. The system comprises a Core subsystem, individual Game Engines, Platform Ports, and Support Modules. This design allows ScummVM to support over 250 games across more than 50 different game engines while maintaining a consistent user experience across a wide array of platforms, from desktop computers to mobile devices and game consoles.

This report will examine ScummVM's system architecture, including major components and their interactions, control and data flow, concurrency management, and external interfaces. We will provide visual representations through diagrams, discuss key use cases, and present a data dictionary of important terms. The report concludes with an analysis of ScummVM's architectural strengths and limitations, as well as insights gained from studying this complex, modular system.

Through this analysis, we aim to provide a comprehensive understanding of how ScummVM's architectural design enables its remarkable versatility and longevity in the field of game preservation and cross-platform development.

2.2 Derivations

To derive the conceptual architecture of ScummVM, we primarily relied on available documentation and resources provided by the project, including the ScummVM documentation[1], the Developer Central wiki[2], and relevant discussions in the ScummVM forums[3]. By analyzing the system's structure as described in these resources, we were able to outline the key components and their interactions.

Our approach focused on understanding the high-level architecture by studying how ScummVM's **Core subsystem**, various **Game Engines**, **Platform Ports**, and **Support Modules** function together. This helped us identify major dependencies and interaction points without delving into implementation-specific details, ensuring a conceptual rather than concrete architectural analysis.

We used the information from these sources to construct **high-level diagrams** representing the relationships between subsystems, such as the Core's coordination with game engines and platform-specific interfaces. These diagrams reflect a modular architecture that supports the system's extensibility and cross-platform functionality.

To visualize the system's behavior, we created sequence diagrams illustrating key use cases like game loading and saving. These were based on the architectural principles discussed in the provided documentation, ensuring that our conceptual model aligns with ScummVM’s operational flow.

3.0. System Architecture

3.1. Overall Structure

ScummVM’s architecture follows a **layered architecture style**, which enhances portability, modularity, and maintainability[1]. It is built to support multiple game engines, allowing ScummVM to run a wide variety of classic adventure games across platforms[1]. The layered architecture is combined with elements of **interpreter architecture**, where each game engine interprets game-specific scripts, enabling platform-independent execution.

By abstracting system-specific operations (like graphics rendering, file I/O, and input handling) and using a consistent interface, ScummVM achieves a high degree of portability across platforms like Windows, macOS, Linux, and mobile devices.

3.2. Major Components

The Core subsystem is a central component of ScummVM, responsible for managing key system operations such as memory allocation, file input and output, configuration settings, and ensuring platform independence[1]. It interfaces with platform-specific backend code to keep higher layers independent of hardware details. ScummVM is designed to support a variety of modular game engines, each of which interprets game scripts, processes player inputs, and manages game states. Examples include the SCUMM engine for classic adventure games, the AGI and SCI engines for titles from Sierra, and GrimE for more modern games. Each game engine is treated as a plugin, allowing for easy addition or removal without affecting other parts of the system. The platform ports ensure that ScummVM operates consistently across different operating systems such as Windows, macOS, Linux, and mobile platforms, using external libraries like SDL to handle file I/O, graphics rendering, and audio playback. Additional support modules are integrated to manage specific audio formats and graphics requirements, enhancing portability and system compatibility.

3.3. Component Breakdown

Engines (SCUMM, Sword, SCI) Each game engine is responsible for interpreting the game's logic and executing scripts[1][2]. Engines like SCUMM (LucasArts) or SCI (Sierra) include built-in mechanisms for controlling gameplay, dialog, character movement, and in-game events. They rely heavily on the Core for memory management, file I/O, and other system-level tasks.

The **common/** directory contains reusable components that are shared across the various game engines, such as utilities for file handling, encoding, memory management, and other common services that don't need to be duplicated across different engines.[1][2]

Audio ScummVM supports various audio formats and outputs through different libraries that handle sound mixing, MIDI playback, and digital sound effects. The audio module is responsible for ensuring that these formats work seamlessly across different platforms and are correctly synchronized with the game state.

Visual/Graphics (Rendering) The rendering system is abstracted through libraries like SDL, allowing ScummVM to render sprites, backgrounds, and game interfaces consistently, regardless of the platform. The graphics subsystem communicates with the game engines to display visuals based on game events and user input.

The **gui/** directory manages the user interface elements such as menus, dialog boxes, and settings screens. This subsystem interacts with both the rendering system (for displaying the UI) and the logic tier (to adjust game settings or save/load game states).

The **math/** module handles various mathematical operations required by the game engines, such as vector calculations, interpolation, and coordinate transformations, ensuring accurate game logic and movement.

The **base/** directory includes core files responsible for initializing the system, loading plugins, and managing platform-independent operations such as memory handling, input processing, and configuration management.

3.4. Control and Data Flow

The control and data flow in ScummVM are structured across its layered architecture to facilitate efficient handling of user input, data processing, game state management, and rendering. The User Interface Layer captures player inputs, such as mouse clicks or key presses, and forwards them to the Engines Layer. In this layer, the appropriate game engine interprets the input using game-specific scripts, processes the action, and updates the game state accordingly. If the game requires resources like sprites or audio, the Subsystems Layer handles these operations, retrieving the necessary assets and managing functionalities such as audio playback and graphics rendering. Once the game state is updated, it is sent back through the Engines Layer to the User Interface Layer, which renders the changes on the screen. This structure ensures that each layer has a distinct role, and the interaction between layers is streamlined to provide smooth gameplay and system responsiveness.

3.5. Concurrency and Thread Management

ScummVM primarily operates using a single-threaded model, as most older games it supports are designed to follow a linear structure. However, for modern games or ports, ScummVM may utilize asynchronous handling or threading for specific tasks. For example, background resource loading, such as audio or textures, can be handled asynchronously to prevent delays in the main game loop. Similarly, audio streaming may run on a separate thread to ensure continuous playback without disrupting gameplay, and certain concurrent input handling mechanisms are used to maintain smooth interaction. By limiting concurrency to specific areas, ScummVM maintains reliable performance even for games that were not originally designed for multi-threading. Its modular, layered architecture, coupled with an interpreter style, allows ScummVM to support a wide range of platforms while preserving the functionality of classic adventure games, ensuring portability and extensibility without requiring changes to the original game code.

3.6. Dependencies Between Subsystems

The **Game Engines** (e.g., SCUMM, Sword, SCI) are at the core of ScummVM's functionality, interpreting game logic, managing game scripts, and controlling gameplay elements such as character movement, dialogue, and in-game events. Each engine relies heavily on **Common Code** for essential system services like memory management, file I/O, and encoding. This allows the engines to remain efficient and focused on game-specific logic without needing to reimplement fundamental utilities.

The **Common Subsystem** provides reusable services shared by all game engines. These utilities handle tasks like file handling, memory management, and encoding, which are essential for the engines to function effectively. By sharing these resources, ScummVM avoids duplicating efforts across different game engines, ensuring consistency and reducing development complexity.

The **Audio Subsystem** plays a crucial role in supporting different audio formats and outputs, managing sound mixing, MIDI playback, and digital sound effects. This subsystem ensures that audio is properly synchronized with the game state and works seamlessly across platforms. It relies on the game engines to trigger audio events and uses the underlying platform-specific libraries to deliver audio output to the user.

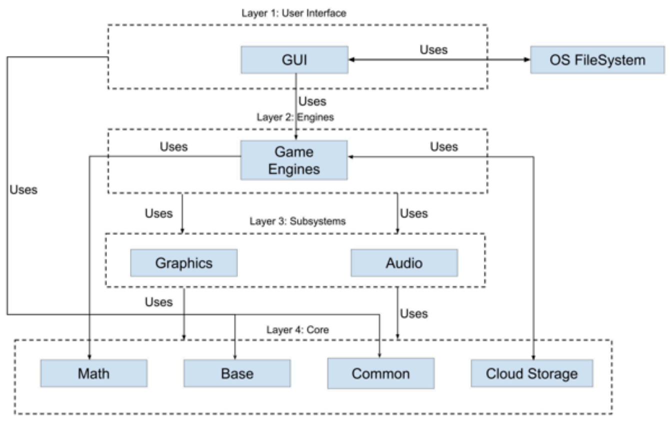
The **Graphics Subsystem** (Rendering) is responsible for rendering the game's visuals, including sprites, backgrounds, and interfaces. This subsystem relies on abstraction layers like SDL to ensure consistent rendering across different platforms. It communicates directly with the game engines to display visuals in response to game logic and user input. The graphics subsystem also interacts with **Common Code** for utilities like image processing and with the **Math Subsystem** to perform complex operations such as transformations and scaling.

The **GUI Subsystem** manages the user interface elements, such as menus, dialog boxes, and settings screens. It interacts with the **Graphics Subsystem** to display these UI elements and with the game engines to adjust settings, save and load game states, and provide user feedback. The **GUI Subsystem** is critical for providing an intuitive and consistent user experience across games and platforms.

The **Math Subsystem** provides essential mathematical functions for the game engines, including vector calculations, interpolation, and coordinate transformations. These operations are critical for accurate game logic, physics, and character movement. The math module interacts closely with the game engines and **Graphics Subsystem**, ensuring that visual elements are rendered correctly and game objects behave according to their designed logic.

Finally, the **Base Subsystem** is responsible for initializing the system, loading necessary plugins, and managing platform-independent operations like memory handling and input processing. This subsystem forms the foundation upon which all other components rely. It interacts with every other subsystem—ensuring that engines, audio, graphics, and user interfaces can be loaded, executed, and managed efficiently across platforms.

4.0. Diagrams



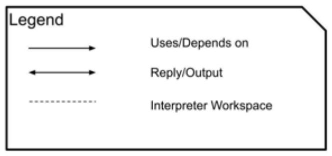


Diagram 1: This diagram shows the structure of ScummVM, organized into layers. At the top, the GUI interacts with the OS FileSystem. The Game Engines use Graphics, Audio, and other subsystems, which rely on core components like Math, Base, and Cloud Storage to function smoothly.

5.0. External Interfaces

The external interfaces of ScummVM define the points of interaction between ScummVM and various external components, such as operating systems, input devices, file systems, and external libraries. This section will outline how these interfaces are used to support ScummVM's operations across different platforms, game engines, and use cases.

User Input Interfaces

ScummVM's user interface captures inputs from a variety of devices, such as keyboards, mice, and game controllers. This is crucial for user navigation through menus, game controls, and configuration settings. These inputs are processed through the **Core Subsystem**, which translates hardware signals into meaningful commands that the active game engine can interpret.

The user inputs interface primarily leverages libraries like **SDL** [2], which abstracts platform-specific details, ensuring compatibility across different operating systems. This design allows ScummVM to maintain a consistent experience, regardless of the hardware used by the player.[1]

File System Interfaces

ScummVM depends on file system interfaces to load game assets, save game states, and manage configuration files. Each game engine within ScummVM interacts with the file system to retrieve necessary assets like graphics, audio, and script files.

The file system interface is handled by the **Data Tier**, which abstracts the complexities of platform-specific file systems, ensuring that file paths and directory structures are properly managed. Configuration files, which store user settings, are also accessed and updated through this interface, allowing ScummVM to remember user preferences across sessions.[1]

Graphics and Audio Interfaces

The audio and graphics interfaces are integral to providing an immersive experience for players. ScummVM utilizes libraries such as **SDL**, **OpenGL**, and **ALSA** to render graphics and play audio. These libraries enable ScummVM to support various resolutions, aspect ratios, and audio formats, preserving the original gaming experience.

The **Presentation Tier** manages the interaction with these external libraries, sending rendering commands to SDL for graphical output and delegating audio commands to ALSA or PulseAudio, depending on the platform. This modular approach allows ScummVM to remain flexible and support a wide range of devices without modifying its internal logic.[2]

Game Engine Interfaces

Each game engine acts as an interpreter for a specific set of games, managing the execution of game scripts and controlling game flow. The interaction between ScummVM's **Logic Tier** and game engines is a crucial external interface, as game engines must interface with the **Core Subsystem** for services like file I/O, memory management, and state management.

These interfaces are standardized, ensuring that any supported game engine can plug into ScummVM's architecture and utilize shared services provided by the Core. This design enables the addition of new game engines with minimal changes to the overall architecture.[1]

Platform Port Interfaces

ScummVM includes platform-specific backends for handling low-level operations such as rendering, input management, and audio playback. These backends act as adapters, translating platform-specific calls (e.g., DirectSound for Windows or PulseAudio for Linux) into a standard interface that ScummVM can utilize.[1]

The **Platform Ports** enable ScummVM to achieve cross-platform compatibility, ensuring consistent performance and functionality across various devices, from desktop PCs to handheld consoles.

Network Interfaces (Optional)

While ScummVM primarily operates as a standalone application, it interfaces with the internet for certain features, such as downloading game assets, checking for updates, or providing links to documentation. These network interfaces are primarily used when ScummVM connects to its official website or a remote game repository.

ScummVM's website (scummvm.org) serves as a central point for users to download the application, access supported game files, and view documentation. The **Core Subsystem** manages these HTTP/FTP interactions using libraries such as **libcurl**. These network requests enable ScummVM to fetch additional game files, perform updates, and communicate with online resources.

The network interface is optional and can be disabled or customized based on user preferences. This flexibility ensures that ScummVM remains lightweight and can function entirely offline if desired, preserving its portability and ease of use.

6.0. Use Cases

6.1. Use Case 1

In this use case, the process of downloading and installing ScummVM involves several key components of the web apps system architecture. The user begins by navigating to the ScummVM website through a web browser, which acts as the entry point for accessing the downloadable installer. The ScummVM website component handles HTTP requests and provides the installer file, which is then saved to the local file system by the browser. When the user executes the installer, the ScummVM Core component is activated, setting up the necessary environment and configuring the software on the local system. After installation, the user can navigate to the Game Repository through the web browser, where the ScummVM Core interacts with the repository to request and display the list of supported games. Upon selecting a game, ScummVM sends a request to the Game Repository for the corresponding game files, retrieves them over the network, and saves them to the local file system. This sequence showcases ScummVM's modular architecture, highlighting the interaction between the user interface, web components, file system, and external game repository to facilitate cross-platform gameplay.

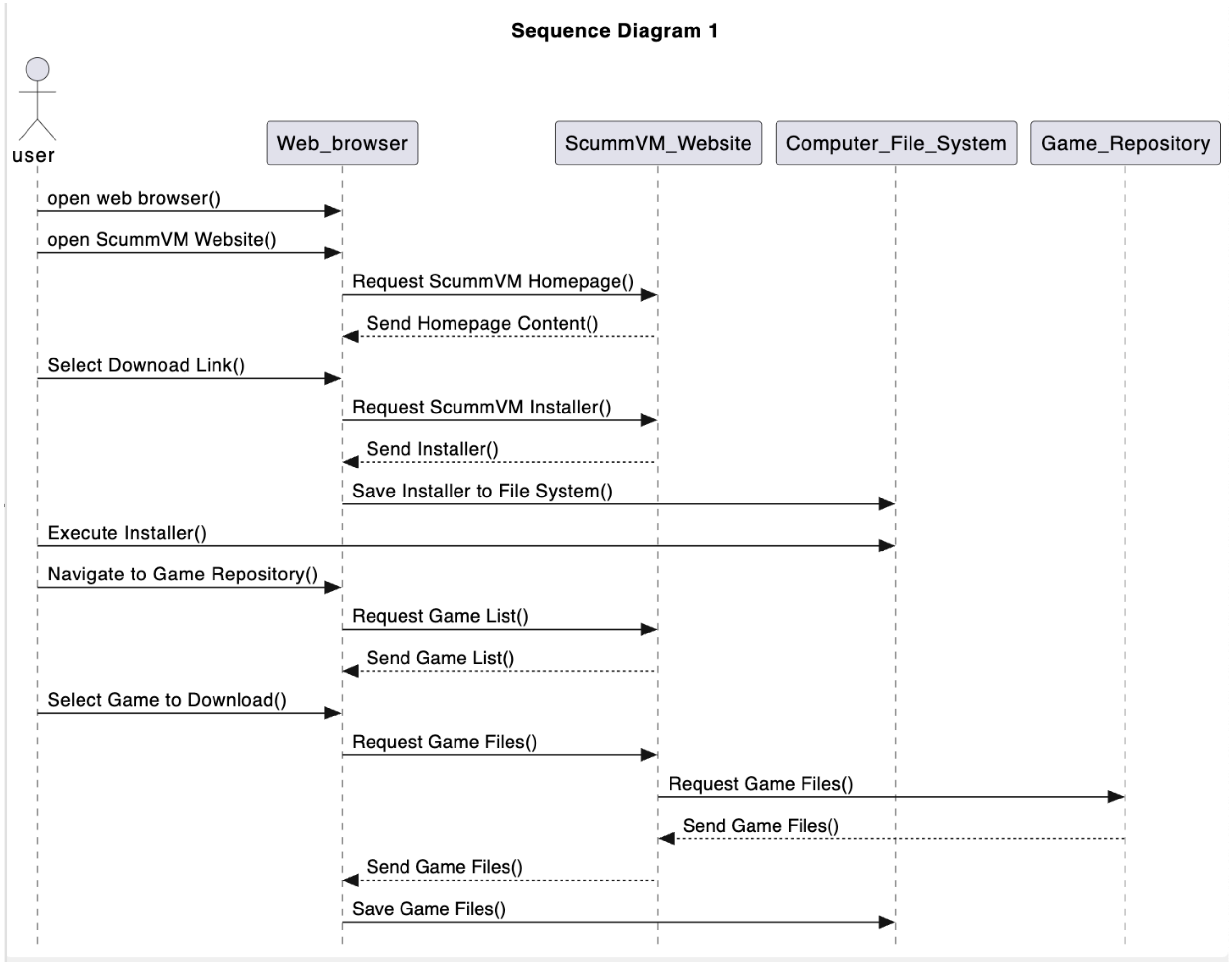
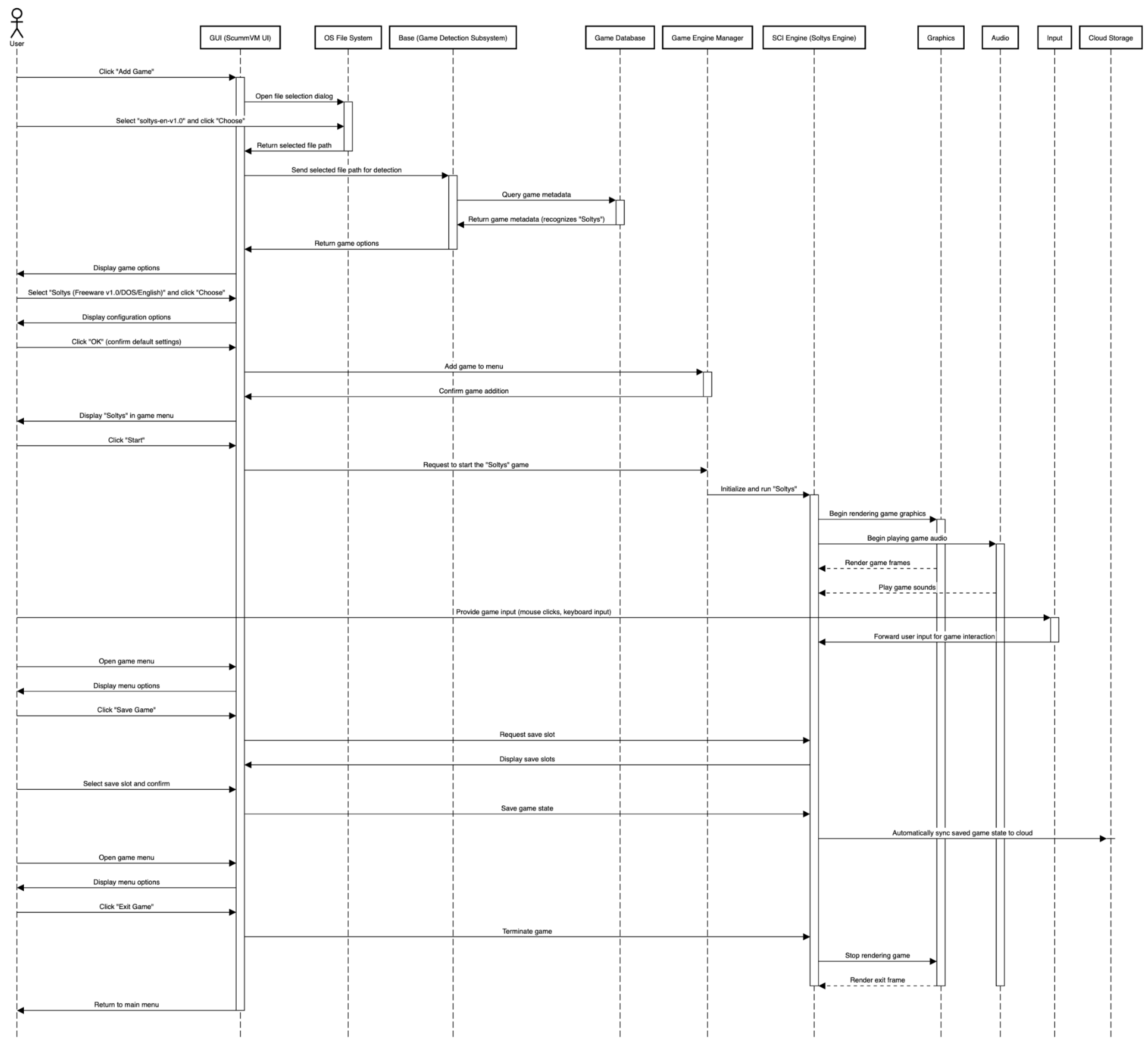


Diagram 2 illustrates a sequence of user interactions with the web browser, which communicates with the ScummVM website, computer file system, and game repository. This is shown by the arrows with solid lines. The ScummVM website, and the game repository both communicate back information as shown by the dotted line arrows.

6.2. Use Case 2

In this use case, the user adds and interacts with the game "Soltys" within the ScummVM application. The user begins by clicking "Add Game" in the ScummVM UI, triggering the Base (Game Detection Subsystem) to check the selected files for known game data. The Base returns the metadata for the game "Soltys" to the user. After the user confirms the game options and settings, the Game Engine Manager adds "Soltys" to the ScummVM game menu. The user then clicks "Start" to launch the game, which initializes the SCI Engine. This engine runs the game and interacts with the Graphics and Audio subsystems to display graphics and play sounds. During gameplay, the user can open the in-game menu to load or save the game state. When the user clicks "Save Game," the game state is saved locally by the SCI Engine and automatically synced to the cloud via the Cloud Storage subsystem. After interacting with the game, the user can open the in-game menu and select "Exit Game." This action stops the SCI Engine, halts the graphics rendering and audio playback, and returns the user to the ScummVM main menu.

Sequence Diagram 2: Add, Run, Save, and Exit "Soltys" in ScummVM (with Automatic Cloud Sync)



User
GUI (ScummVM UI)
OS File System
Base (Game Detection Subsystem)
Game Database
Game Engine Manager
SCI Engine (Soltys Engine)
Graphics
Audio
Input
Cloud Storage
Click "Add Game"
Open file selection dialog
Select "soltys-en-v1.0" and click "Choose"
Return selected file path
Send selected file path for detection
Query game metadata
Return game metadata (recognizes "Soltys")
Return game options
Display game options
Select "Soltys (Freeware v1.0/DOS/English)" and click "Choose"
Display configuration options
Click "OK" (confirm default settings)
Add game to menu
Confirm game addition
Display "Soltys" in game menu
Click "Start"
Request to start the "Soltys" game
Initialize and run "Soltys"
Begin rendering game graphics
Begin playing game audio
Render game frames
Play game sounds
Provide game input (mouse clicks, keyboard input)
Forward user input for game interaction
Open game menu
Display menu options
Click "Save Game"
Request save slot
Display save slots
Select save slot and confirm
Save game state
Automatically sync saved game state to cloud
Open game menu
Display menu options
Click "Exit Game"
Terminate game
Stop rendering game
Render exit frame
Return to main menu

Diagram 3 illustrates the sequence of events where the user adds, launches, interacts with, saves, and exits the game "Soltys." The User, depicted as a stick figure, interacts directly with the ScummVM UI. The diagram shows solid arrows representing direct interactions between the user and the GUI (ScummVM UI). It also depicts communications with the Base (Game Detection Subsystem), Game Engine Manager, and SCI Engine to manage the game's lifecycle. Dotted arrows represent the continuous rendering of graphics and playing of audio while the game is running, with the SCI Engine managing the game's interaction with the Graphics and Audio subsystems.

7.0. Data Dictionary

Engine	An “engine” is the program that is responsible for executing the foundational or crucial tasks for the program. In the case of ScummVM, an engine is responsible for organizing the functions together in order to run the designated program.[1]
Graphics	Graphics are responsible for rendering the visuals of the game ScummVM is running. This includes the drawings (sprites) and background imagery, as well as the screen resolution, scaling, and aspect ratios depending on the user’s device. In the case of ScummVM, it is responsible for ensuring the graphics of older games are able to be properly executed on modern displays.[1]
Input	Inputs are user interactions that are through an external device. Inputs are the software data, or information it receives, that produces an output, typically as movement or producing textual information for the engine to interpret within the game. In the context of ScummVM, the inputs are typically from the mouse, keyboard, or game controller. The quality and accuracy of this data directly influence the software's output and overall performance.[1]
Sound	The sound component is responsible for playback of any audio the original game may have produced. This includes music, sound effects, or any dialogue from in-game characters.[1]

Save/Load	The saving module is responsible for saving the game's current save file so it can be loaded back later. This includes any progress the user may have completed within the game's original storyline, or any in-game objects and/or achievements the user has completed. This is done so the user does not have to start from the beginning every time they run the game.
Backend	The backend of the game is ensuring that the specifics of the game engine are accessible to different platforms, by handling any platform-specific operations. For example, ensuring the inputs, graphics, and audio are performed the same on both Windows Operating Systems and Linux; Windows possibly requiring **DirectSound** to provide audio, while Linux may require **PulseAudio[2]**
"Game Engine"	A "game engine" in ScummVM refers to the code that is responsible for interpreting and running a specific game. Each game has its own game engine. For example, *"The 7th Guest"*, is a supported game on the ScummVM, which was originally run using the **GROOVIE game engine.[2]**
"Virtual Machine"	A virtual machine is software that emulates a physical computer or processor, so that programs specific to that emulation can be run. For the context of ScummVM, the SCUMM Virtual Machine runs the SCUMM scripts written by LucasArts developers, allowing for the games to be run on modern technology.

8.0. Naming Conventions

iMUSE	Interactive Music Streaming Engine or Interactive Music and Sound Effects [3]
INSANE	INteractive Streaming ANimation Engine[3]

LEC	LucasArts Entertainment Company[3]
SAGA	Scripts for Animated Graphic Adventures[3]
SCUMM	Script Creation Utility for Maniac Mansion[3]
VM	Virtual Machine
SPU	SCUMM Presentation Utility[3]
OS	Operating System

9.0. Conclusions

Our analysis of ScummVM's conceptual architecture reveals a sophisticated system that effectively achieves its goals of game preservation and cross-platform compatibility. The project's success stems from its modular design, which separates core functionality, game engines, and platform-specific code. This approach, combined with well-defined abstraction layers like the OSystem API, enables ScummVM to support a diverse range of games across numerous platforms while maintaining a consistent user experience.

The plugin architecture for game engines stands out as a particularly innovative feature, allowing for the continuous expansion of supported titles without compromising the integrity of the core system. This extensibility, coupled with standardized interfaces for audio, graphics, and input handling, ensures that ScummVM can adapt to new challenges and opportunities in the realm of classic game preservation.

ScummVM's architecture demonstrates remarkable strengths in portability and performance. By replacing original executables rather than emulating them, the system often delivers superior performance compared to the original games, while also enabling play on a wide array of modern devices. This approach not only preserves gaming history but often enhances it, making classic titles more accessible and enjoyable for contemporary audiences.

However, the complexity of ScummVM's highly modular structure presents certain challenges. New contributors may face a steeper learning curve, and the maintenance overhead required to support numerous platforms and game engines is substantial. Additionally, ensuring consistent behavior across all combinations of game engines and platforms remains an ongoing challenge for the development team.

Despite these limitations, ScummVM's architectural design stands as a testament to the power of modular, well-abstracted software design in creating flexible, long-lasting systems. Its success in preserving and rejuvenating classic games offers valuable lessons for other software projects aiming for similar goals of compatibility and longevity. As ScummVM continues to evolve, its robust architectural foundation promises to support the preservation and enjoyment of classic games for years to come.

10.0. Lessons learned

Our analysis of ScummVM's architecture provided valuable insights into large-scale software design, bridging the gap between our theoretical knowledge and real-world application. The complexity of ScummVM's modular system initially seemed daunting, but it ultimately demonstrated the power of well-structured design in achieving flexibility and maintainability.

We gained a deeper appreciation for abstraction layers, particularly how the OSystem API enables cross-platform compatibility. This practical application of interface design principles, familiar from our object-oriented programming studies, showcased their crucial role in large-scale systems. The balance ScummVM strikes between supporting diverse games and maintaining performance highlighted the importance of efficient design in flexible systems, a consideration we hadn't fully grasped before.

Examining ScummVM's architecture allowed us to see design patterns in action, moving beyond textbook examples to understand their practical benefits and trade-offs. The challenge of comprehending how ScummVM accommodates older games gave us insight into the complexities of working with legacy software, an aspect of software engineering we hadn't previously encountered.

For future projects, we've learned the value of starting with a high-level overview before delving into specific components. Leveraging existing documentation proved crucial, as did the iterative process of creating and refining diagrams to visualize the system's structure. Our group discussions were instrumental in unraveling complex architectural aspects, reinforcing the importance of collaborative problem-solving in software analysis.

Connecting theoretical concepts to their practical implementation enhanced our understanding significantly. We also realized the importance of considering historical context when analyzing systems that interface with older software, providing valuable insights into architectural decisions.

This experience has deepened our appreciation for software architecture, illustrating how theoretical concepts translate into real-world systems. It has given us a new perspective on the challenges and considerations involved in designing large-scale, adaptable software systems, preparing us for the complexities we'll face in our future careers as software engineers.

11.0. References

1. *ScummVM Documentation*. ScummVM, https://docs.scummvm.org.
2. *Developer Central*. ScummVM Wiki, https://wiki.scummvm.org/index.php?title=Developer_Central.
3. *ScummVM Forums*. ScummVM, 27 Mar. 2009, https://forums.scummvm.org/viewtopic.php?t=7886.