CISC 322

Assignment 1
ScummVM Conceptual Architecture

Brandon Kim	21bmkk@queensu.ca
Ewan Byrne	21emb10@queensu.ca
Owen Sawler	21ors3@queensu.ca
John Li	21jl263@queensu.ca
Gavin Yan	21gly@queensu.ca
Norah Jurdjevic	20nrtj@queensu.ca

Table of Contents

Abstract

ScummVM is an open source program which gives users the ability to run and play classic adventure and role-playing games using only the game's data files. The program rewrites the game's executables, thereby allowing users to play these games on systems for which they were not originally designed. In an effort to gain a stronger understanding of conceptual architecture, we aimed to work backwards from the existing documentation for ScummVM to determine its architectural style. We conducted research independently using the available documentation, which informed our initial ideas. We then finalized a conceptual architecture through group discussion and presentation of our findings. Together, we concluded that ScummVM uses a combination of layered and interpreter architectural styles. The modular nature of ScummVM's structure facilitates the maintenance of supported systems and assists the implementation of new ones. This report discusses these architectural features in detail, as well as major subsystems, relevant use cases, the evolution of ScummVM's systems and how these features might be divided into tasks within a development team.

Introduction

In an era marked by rapid technological advancement, the preservation of classic video game titles proposes a significant challenge. As video games transitioned from a niche pastime to a global phenomenon, many classic video game titles risk obsolescence due to compatibility issues with more modern hardware and newer operating systems. In an effort of preservation, ScummVM, also known as the Script Creation Utility for Maniac Mansion Virtual Machine, was developed as a solution to this problem, providing a platform to enable users to run older games on contemporary hardware. Initially designed to support LucasArts adventure games built on the SCUMM engine, ScummVM has since expanded to accommodate a wide range of other classic engines such as Plumbers, AGI, SCI, and many others, providing preservation and playability for an extensive library of iconic titles.

This report aims to systematically provide a comprehensive analysis of ScummVM's conceptual architecture, elucidating the essential components and mechanisms that facilitate the execution of retro video games on modern systems. The first section will detail the derivation process, outlining our team methodology for research on ScummVM, developing an implicit architecture model, and how we divided responsibilities throughout the writing process. Following this, we will take a look at the conceptual architecture of ScummVM, focusing on examining the major subsystems involved, including the OSystem API, backend, and the game engines themselves. Each subsystem will be analyzed in terms of their individual characteristics and interactions, providing a comprehensive overview on ScummVM's overall structure and illustrate how these components interact to achieve seamless game execution. To contextualize the conceptual architecture, this report will further investigate relevant use cases in an effort to demonstrate its practical application with other components. Next, we will trace ScummVM's development from its creation to its current state, illuminating key milestones and advancements that made ScummVM for what it is today. We will also analyze the division of responsibilities among developers to shed

light on the collaborative nature of the project and how it facilitates ongoing maintenance and enhancement. Lastly, to ensure clarity, this report will include a data dictionary of essential terms and concepts related to ScummVM, along with a section for naming conventions to improve code readability and maintainability.

By addressing these components and various relations between subsystems, this report aims to provide a thorough understanding of ScummVMs conceptual structure, functionality, and evolution. We anticipate that this research will offer valuable insights into ScummVM's architecture, equipping readers with a deeper understanding of its implementation and potential pathways for future development.

Derivation Process

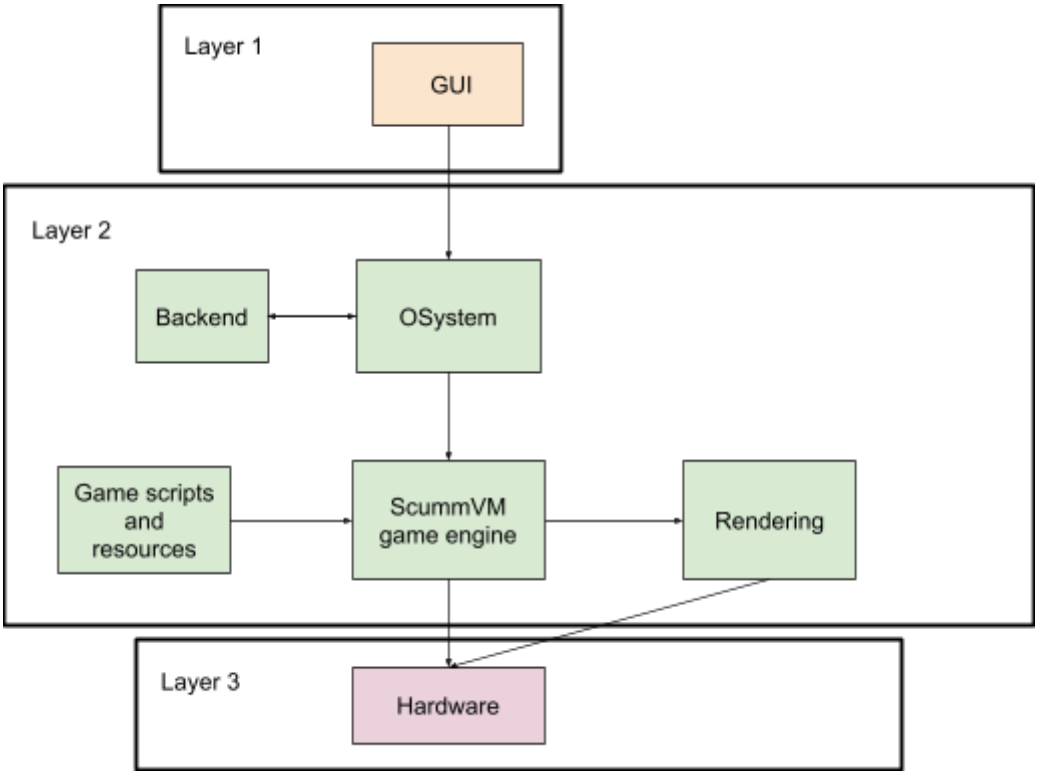
We decided that the best approach to determine an architecture that accurately represents ScummVM would be to do independent research and then meet up as a group to compare and contrast our findings. We independently examined documentation for ScummVM and forums, while noting any potential architectural styles and subsystems. We then held a meeting to discuss our findings and compare notes. We also installed ScummVM and some games in order to experiment and familiarize ourselves with the software, which was helpful in ironing out some details that weren't abundantly clear from the documentation. The consensus from the discussion was that there were two relevant architectural styles present in ScummVM: interpreter style and layered style. No alternative styles were discussed since we were all in agreement that these two were the only ones of relevance.

Once the relevant architectural styles were determined, we divided up the work into parts and assigned each member of the group a set of parts to focus on. Some parts were done entirely by one person, while other parts were done by 2-3 other people. Discussions about the different parts were had, which allowed for multiple opinions regarding each part to still be present. This enabled us to efficiently determine the rest of the architecture, which included mostly diagrams, in a cohesive manner.

We determined that some sections of the documentation were of greater importance to determining ScummVM's architecture. These were the sections relating to how to use ScummVM from the ScummVM documentation, and the "Developer Central" section from the ScummVM wiki. The former was especially useful in determining use cases and sequence diagrams, while the latter was useful in determining the relevant components and the way that they interact in the system.

Architecture

Conceptual Architecture



The conceptual architecture of ScummVM fits well within both the layered architectural style and the interpreter architectural style. The provided diagram helps to visualize how these two styles interplay to support the emulation of games.

Layered Architectural Style

The layered architecture divides the system into distinct layers, where each layer builds upon the services of the one below it. This structure supports modularity, separation of concerns, and ease of maintainability. In ScummVM's case, we can observe a clear separation of concerns into three distinct layers:

User Interface Layer

The topmost layer represents the user interface, consisting of the GUI and User Input. This layer abstracts away the complexity of the underlying game engine and hardware, ensuring that the user interacts with the system in an intuitive and platform-independent manner. The user interacts with ScummVM through this GUI, which collects inputs like clicks or keystrokes and passes them down to lower-levels for interpretation.

Game Engine Layer

This intermediate layer handles the game-specific logic and game engine functionalities. The ScummVM OSystem component plays a central role, receiving inputs from the GUI, requesting the OS-specific backend for certain tasks, and communicating that to the game engine. As the core of the system, the game engine interacts with both the game's resources and the underlying hardware, as well as requesting graphics to be rendered.

Hardware Abstraction

The lowest layer interacts directly with the hardware. This layer handles the execution at the system level and retrieves the data required for execution. The hardware includes the physical components that run the game, while the game scripts provide the instructions that the ScummVM engine interprets.

Interpreter Architectural Style

ScummVM's interpreter architectural style is characterized by its clear process of translating high-level game scripts into low-level actions that the system can execute. In an interpreter architecture, the key component is the interpreter itself, which dynamically reads, parses, and executes instructions at runtime, rather than compiling them into machine code beforehand. In the case of ScummVM, the game engine acts as this interpreter. It processes the game's original scripts and resources, which are typically written in domain-specific languages or data formats for adventure games. These scripts dictate the game's logic, flow, and responses to player actions, and the engine interprets them step by step, turning abstract game logic into concrete operations.

The game scripts themselves are not executable code but need to be interpreted in real time. The ScummVM engine reads these scripts and determines what action to take—whether to display a specific image, play a sound, or respond to player input. For instance, when a game script calls for an object to be rendered on the screen, the game engine interprets this command and communicates with the rendering component to generate the appropriate graphics. This real-time translation of instructions is a defining feature of the interpreter architectural style.

The engine only interprets the game's logic and defers other tasks, like rendering or OS-specific tasks, to dedicated components such as the renderer or the backend. This ensures that the interpreter focuses purely on executing game scripts, while leaving low-level operations to other parts of the system. This separation of concerns is a core characteristic of the interpreter architectural pattern: the interpreter handles high-level logic, while other subsystems manage detailed system interactions. By maintaining this clear division, ScummVM uses the interpreter architectural style to enable cross-platform compatibility for various games.

Major Subsystems

Game Engines

The Game Engines subsystem is one of the most crucial components of ScummVM. Each game engine is designed to replicate the functionality of the original engines used by the games that ScummVM can run. This is what enables ScummVM to run these titles on modern systems. Each game engine interprets the original game-specific data files such as scripts, graphics, audio, and other assets, and executes the game logic defined in these files. The engines allow ScummVM to support a wide array of games from engines like SCUMM, SCI, and many more.

The modular subsystem architecture of ScummVM's game engines ensures each game engine can operate independently of each other, allowing them to run games built on different technologies without conflicts. At the same time, the engines interact seamlessly with common subsystems like graphics rendering, audio processor, input handling, and file I/O. This in turn ensures that ScummVM can support a broad range of games while maintaining high flexibility and modularity.

Each game engine is specialized and responsible for the unique requirements of the games it supports. An example is the SCI engine, which is responsible for interpreting Sierra's complex point and click adventure games, handling scripts, puzzles, inventory management, and game interactions. Meanwhile the SCUMM engine is focused on parsing the scripting language in LucasArts games. Each engine interprets the scripts - translating the game's original code and resources into instructions a modern system can understand and process. This effectively decouples the game from the hardware it was originally designed for and allows for ScummVM to run them on different hardware.

The game engines also manage the interaction with external game assets such as sound effects, music, animations, and dialog scripts. These assets are usually stored in proprietary formats specific to each game engine, and so the engine needs to decode and process them in real-time for the game to be presented as intended.

The modular structure of ScummVM and the game engines subsystem means that new game engines can be integrated over time. This is key to ScummVM's evolvability, as developers are able to contribute new engines for currently unsupported games without disrupting existing functionality. This architecture allows for easy testing and debugging when integrating new engines - and ensures flexibility while also enhancing the scalability of ScummVM.

OSystem API

The OSystem API is responsible for defining which available features a game can use. To do this, the OSystem retrieves information from the backend as requested, and returns it to the frontend. This allows game frontends to access backend methods and information without knowing the specific location.

The OSystem defines the graphical features of a game to ensure that the game frontend is able to implement everything it needs efficiently. The game graphics are dealt with as three separate layers: game graphics, overlay graphics and the mouse. Game graphics are responsible for the appearance and sizing of the virtual screen as well as resizing to the real screen. The OSystem retrieves supported graphics modes and pixel formats for the game, which are then used to determine the optimal way to set up the display hardware. The overlay is responsible for the graphics which contain menu information, anything you might expect to see when the game is paused. The mouse layer contains a low-level implementation for the in-game mouse. The properties of the mouse are fetched from backend definitions.

The OSystem also handles timing, hardware inputs, keybindings and thread management. Like the graphical features, the requirements and restrictions of these features are defined in the platform's backend, while the OSystem is only retrieving the information. It serves as a bridge between the platform implementation in the backend and the frontend of the game. The OSystem does not store any information or perform any operations directly, it simply retrieves the information. As a result, the OSystem without a backend to reference does nothing. This improves ScummVM's evolvability as introducing a new platform or backend is easily facilitated through the implementation of an OSystem subclass. This allows for new backends to be implemented without affecting existing functionality.

Backends

The backends are responsible for the implementation of the OSystem API. Each backend supports at least one platform, with some providing support for multiple platforms. These backends are subclass implementations of the OSystem class defined in the OSystem API. Each backend is responsible for providing methods to create timers and handle user input, controls for CD playback and other sounds, and a video surface which ScummVM can draw in. The backends are organized into a number of directories, with each directory representing one component of the backend. These contain platform-specific implementations for the features required from the component. Combined, these categories provide all the features required of the backend by the OSystem API.

The OSystem subclass implementation for the backend is stored in the platform directory. This includes information about hardware inputs such as keymapping and other user input processing. The implementation also contains the relevant graphics information, such as the supported graphics modes, overlays and mouse information. This directory also holds the assets, if any, required by the platform. Much of this information is referenced by the OSystem API to determine the specific features of a game.

There are individual directories responsible for the maintenance and implementation of several other features such as CD sound playback, dialogs, DLC, cloud-based saving, event handling, file system management, detailed graphics information, keymapping and hardware inputs, logging, thread management, plugins, saves, text-to-speech functionality, timers, platform updates and a virtual keyboard. These directories contain information and methods accessed by the OSystem API on behalf of the game's frontend. This backend structure allows for the straightforward addition of any currently unsupported platforms to ScummVM, as developers are able to interact with only those directories relevant to their implementation without any cost to performance or functionality.

Concurrency

ScummVM does not rely heavily on concurrency. It was designed to emulate classic games, many of which were designed for single-threaded environments, so multithreading would not be beneficial for running the games. Some elements of ScummVM's audio processing may involve concurrency, such as handling audio streaming in separate threads, or

audio buffering. However, the extent to which ScummVM utilizes multithreading does not extend very far beyond this.

Evolution

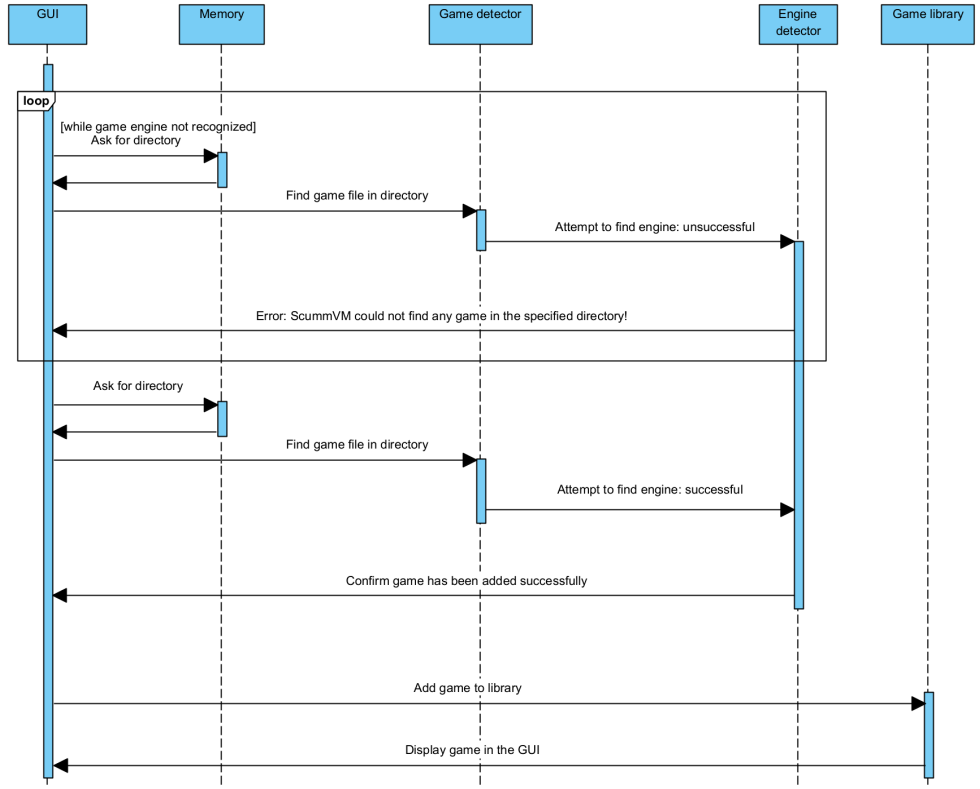
ScummVM originally started as a project to run games built using the SCUMM engine, which LucasArts had developed for their graphic adventure games. The SCUMM engine was first used in 1987 for the iconic *Maniac Mansion* and later became the foundation for many other LucasArts' titles, such as *The Secret of Monkey Island, Day of the Tentacle,* and *Max Hit the Road*. SCUMM itself was designed as a general-purpose engine for creating classic point-and-click adventure games. The engine separated game data and logic from the underlying platform specific code, and by using a separate interpreter they were able to process game files that contained artwork, dialogue, and logic while simultaneously leaving the game data untouched. This modular approach allowed games to be ported across different systems, laying down a foundation for the interpreter-based approach which ScummVM would later adopt as LucasArts retired the original SCUMM engine. So, as these games aged and became more obsolete, newer operating systems and hardware were developed, becoming the norm for most modern systems. As a result, many players encountered difficulties running their original SCUMM-based games on modern systems.

In 2001, the creator of ScummVM, Ludvig Strigeus launched a project to emulate the SCUMM engine. This would allow players to run their old LucasArts games on modern platforms. However, Strigeus' goal was unique. Rather than trying to build a traditional emulator for modern systems, he instead chose to rebuild the SCUMM interpreter. Much like the original, this system allowed players to play the original games by reading their data files and reinterpreting them for newer systems, effectively bypassing the need to emulate the hardware itself, and the original game's executable in turn.

Over time, ScummVM evolved into a platform capable of supporting multiple game engines. It became an umbrella project for adventure game preservation, adding support for engineers like Sierra's AGI and SCI, as well as Revolution Software's Virtual Theater which was used in *Beneath a Steel Sky and Broken Sword*. The project continued to evolve, allowing users to easily contribute with their own game engine implementations. Later, ScummVM was ported to a variety of different platforms (IOS, Nintendo Systems, Playstation 3, ect). Overtime, ScummVM has become a vast ecosystem of preservation, supporting over 70 different game engines and hundreds of classic titles. Due to its flexible design and open source nature, ScummVM has become a testament to interpreter-based architecture all with a purpose for keeping these beloved games alive on today's modern systems.

Use Cases

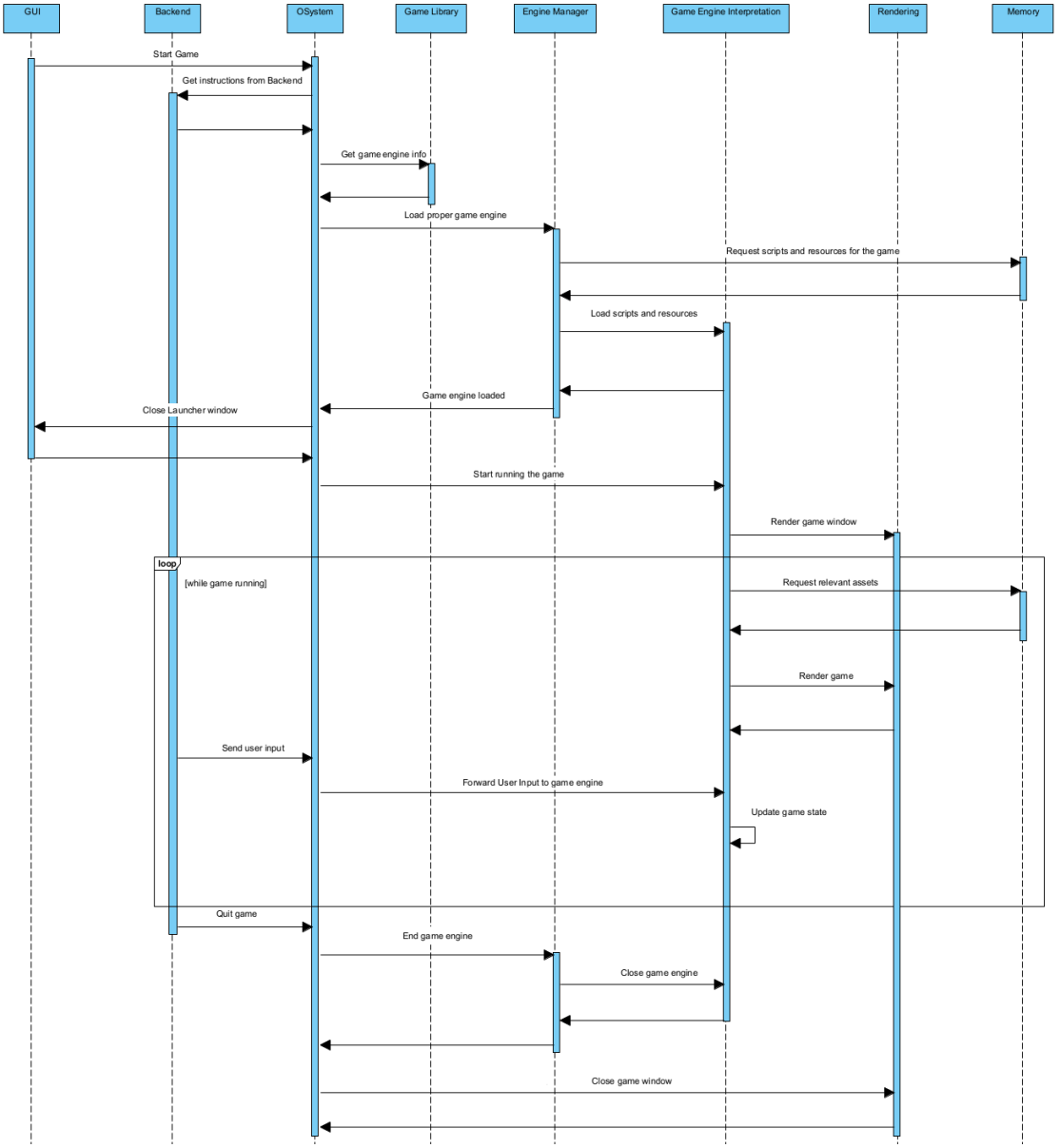
Use Case 1: Adding a Game to ScummVM



This sequence diagram shows how ScummVM is able to add a game to the game library to be played. The GUI represents the launcher interface that the user interacts with. The Memory refers to any type of memory, in this case the storage. This is accessed through either the ScummVM file browser or the operating system's file browser, depending on the user's global ScummVM settings. The Game detector searches through files to identify games that are playable in ScummVM. The Engine detector attempts to find a game engine interpreted in ScummVM that can run the game. The Game library holds the list of games that the user has installed in ScummVM.

To start, the user selects the "Add Game" button on the GUI, which requests a directory from the Memory. The GUI then sends the directory to the game detector, which searches through the directory to find a file containing a game. This file is sent to the engine detector, which attempts to identify the engine that the game is running on. If the engine detector can't find a supported engine, a message will be sent to the GUI to communicate to the user that there is no valid game in the directory. The GUI will then send another request to the Memory, allowing the user to select another directory to search for a game. Otherwise, the engine detector will send a message to the GUI confirming that the game has been recognized, and the GUI will display a message saying that the game has been added. The engine detector will then send a message to the game library telling it to add the game to the library. The game library will then tell the GUI to display the game in the launcher.

Use Case 2: Playing a Game Through ScummVM



The second sequence diagram is for playing a game. This is a process that includes starting the game, followed by gameplay and eventually quitting the game. Note that this sequence diagram does not take into account all the possible actions that could be taken by the user during gameplay, such as pausing, saving or loading the game state, and exiting to the launcher.

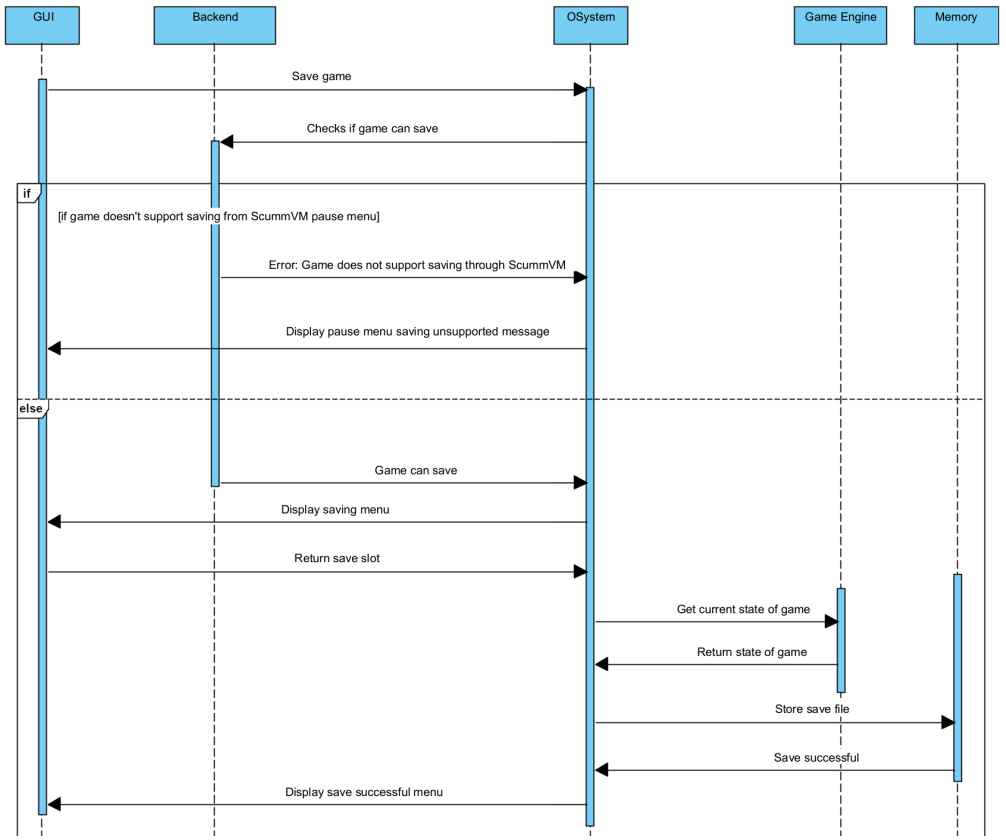
In this sequence diagram, the GUI represents the interface for the launcher and the pause menu. The Backend, OSystem and Game engine interpretation are as described in the subsystems section. These three components deal primarily with functions relating to the game being run, while the GUI handles the non-game related menus provided by ScummVM. The Rendering component handles rendering the game. The Engine manager deals with loading the appropriate game engine and shutting down the game engine. The Game library and Memory are as described in the first use case.

When the user selects a game to play in the GUI, the GUI will immediately start the OSystem component. The OSystem will then get instructions regarding how to start the game from the backend corresponding to the user's device. The OSystem will request the game engine information of the game to be started from the game library, and will then tell the Engine manager to load the proper game engine. The Engine manager gets the scripts and assets for the game from the memory, and then loads them into the Game engine interpretation. It then tells the OSystem that the game engine has been loaded. At this point, the launcher is no longer needed, so the OSystem tells the GUI to close the launcher window. It then tells the Game engine to start running the game. The Game engine tells the Rendering component to render the game. This completes the starting a game portion of the diagram.

From the perspective of the computer, a game can be thought of as a sequence of operations that occurs on each frame, at least for the graphical aspects. We can represent this cycle in the sequence diagram. On each cycle, the game engine asks the Memory, in this case RAM, for any relevant assets that need to be loaded, then tells the rendering component to render the next frame. The Backend sends any user input to the OSystem, which forwards it to the game engine, and the game engine updates the state of the game accordingly.

The last part of the sequence diagram handles quitting the game. When the backend detects that the user would like to quit the game, it sends a message to the OSystem telling it to stop operations. The OSystem then tells the game engine to shut down, then tells the renderer to close the game window.

Use Case 3: Saving a Game



This third sequence diagram is for saving the game. This assumes that the game is currently paused, since the saving option in ScummVM is only accessible from the pause menu. All the components here are as described in the previous sequence diagrams.

Some games in ScummVM cannot be saved from ScummVM's pause menu, and must instead be saved from an in-game save menu. This sequence diagram focuses on saving through ScummVM's pause menu. This menu is handled by the GUI, and the only component that is allowed to communicate with the backend is OSystem, so the GUI must send a message to the OSystem asking whether the game can be saved in the pause menu. The OSystem forwards this message to the backend. If the game cannot be saved through ScummVM's pause menu, the backend communicates this to the OSystem. The OSystem then replies to the GUI, telling it to display a message indicating that the user must save from the in-game menu. If ScummVM does support saving from its pause menu for the game being played, the OSystem sends a message to the GUI, telling it to display the game saving menu for the player to select. Once the player has chosen a save slot, the OSystem sends a request to the Game Engine, asking for the current state of the game. Once this is returned, the OSystem sends a message to the Memory, in this case the storage, which will ask the user where they want to save the game. Once the user has chosen a location, a message is returned to the OSystem to let it know that the game has been saved successfully. Finally, the OSystem sends a message to the GUI, to display that the game has been saved to the user.

Division of Responsibilities Among Developers

In the development ecosystem surrounding ScummVM, there is a clear division of responsibilities that allows classic games to be run on modern systems without requiring any input from the original game developers. Game developers, particularly those who originally created games for older platforms, do not need to make any changes or modifications to their games for them to be supported by ScummVM. This is a crucial aspect of ScummVM's value, as it acts as a bridge between older game code and contemporary hardware without burdening the original developers with reworking or adapting their codebase. The architecture and modularity of ScummVM enable this independence, allowing it to interface directly with game resources and scripts in a way that preserves the original functionality.

The responsibility of making these games compatible with ScummVM falls primarily on two teams: game engine rewriters and ScummVM developers. The game engine rewriters play a critical role by reverse engineering or rewriting the game engines used by these classic titles. Many older games were built with custom engines tailored to specific hardware or system configurations. Game engine rewriters analyze how these engines function and then recreate or adapt the necessary components to function within the ScummVM framework. This can involve interpreting how the original engine handles game logic, input, and rendering, and then rewriting these processes in a way that ScummVM can execute them seamlessly on modern platforms.

Once the game engines are rewritten, the ScummVM developers take over by ensuring that the ScummVM platform can understand and run the newly adapted engines.

This team focuses on the core functionality of ScummVM, which involves integrating these rewritten engines into ScummVM, maintaining compatibility across various systems, and ensuring the game logic is correctly interpreted. Their work includes ensuring that ScummVM's emulation environment supports these new engines, managing the interactions between the game scripts, hardware, and user input. Together, these two teams can be mostly working independently, with occasional communication on the interface.

Conclusion

Through research and analysis, we determined that ScummVM's architectural style uses a combination of both layered and interpreter styles. The layered architecture enhances modularity and maintainability through the separation of the user interface, game engine and hardware and resources into separate layers. The interpreter style allows ScummVM to support a variety of game engines by abstracting differences in game logic. The platform is composed of three primary subsystems, the game engines, the OSystem API and the backend, which interact and work together to bring ScummVM to life. Our research also discusses its evolution and limited concurrency. Through the illustration of several key use cases, the interaction between key components and overall functionality of ScummVM is further illuminated. The report provides a systematic and comprehensive analysis of ScummVM's conceptual architecture, with detailed descriptions of the key components required to execute classic games on modern systems.

ScummVM's open source structure ensures its continued evolution as users are invited to make changes driven by their needs. It provides a valuable service in a rapidly modernizing world, and will likely continue to grow in popularity as game technologies continue to advance.

Lessons Learned

While researching the conceptual architecture, we found that the summary from the ScummVM wiki of the components and their interactions was useful for developing a surface-level understanding of how the system works, as well as the code structure of the repository. This information could be a great entry point for developers looking to work on this system, and this would certainly be the case for any software, even closed-source software. This shows the importance of having a well-documented conceptual architecture.

We found while working on the project that doing research individually and then creating the base of the architecture as a group, in-person, was an effective way of maximizing the generation of ideas while minimizing confusion later on while diving deeper into the architecture and writing the report. Working independently on different sections of the report while maintaining communication with other people working on related sections was a good way of ensuring that the architecture remained cohesive, despite working remotely and individually.

Data Dictionary

Open Source: Software with freely available original source code which may be redistributed and modified.
Virtual Screen: Screen used by the game.
Real Screen: The physical screen of the device.

Naming Conventions

SCUMM – Script Creation Utility for Maniac Mansion
SCI – Sierra Creative Interpreter
I/O – Input and Output
API – Application Programming Interface
CD – Compact Disc
DLC – Downloadable Content

References

- McMurray, J. 2021. Where does ScummVM fit in?. Retrieved October 9, 2024 from https://wiki.ScummVM.org/index.php?title=About
- Newman, J. 2020. What is SCUMM?, SCUMM reference guide :: Introduction. Retrieved October 9, 2024 from https://www.scummvm.org/old/docs/specs/introduction.php
- ScummVM API Documentation. Retrieved from https://doxygen.ScummVM.org/index.html.
- ScummVM Github Repository. Retrieved from https://github.com/ScummVM/ScummVM/blob/master/README.md.
- ScummVM Wiki. 2023. Developer Central. Retrieved October 8, 2024 from https://wiki.ScummVM.org/index.php?title=Developer_Central.