CISC 322/326 Assignment 1: Report
ScummVM: Conceptual Architecture Analysis
October 8, 2024

Group 10

Xinyu Liu(19xl70@queensu.ca)

Chuyi Qin(21ciq@queensu.ca)

Yuzhe Lin(21yl64@queensu.ca)

Yuran Chen(22yc15@queensu.ca)

Fangjie Qiu(21fq2@queensu.ca)

Siyuan Qiu(21SQ15@queensu.ca)

Abstract

This document provides a comprehensive analysis of the architecture, development and operational strategy of ScummVM, which was originally developed to support games created by LucasArts using the SCUMM tool, and has evolved into a cross-platform virtual machine. By reimplementing previous game engines to be compatible with modern operating systems, users are able to play classic adventure games.ScummVM uses a layered architecture consisting of a front-end virtual machine, a mid-end containing APIs and cross-platform code, and a platform-specific back-end supported by common code and a user-friendly GUI.ScummVM utilizes concatenation to simulate multi-threading, to ScummVM utilizes concurrency to emulate multi-threading, enables responsive I/O and rendering, and employs an event-driven design to improve performance. User input flows through the GUI, is processed by the interpreter, generic code, and the game engine, and is ultimately executed on the platform-specific backend. The modular architecture supports cross-platform compatibility, allowing independent development of the engine, UI, and file system, while also ensuring coordinated development of audio, video, and input modules. The modular nature of ScummVM allows for independent development of the game engine, UI, and filesystem, while interdependent modules such as audio, video, and input processing require synchronized cross-team collaboration.

Introduction and Overview

“ScummVM” which stands for Script Creation Utility for Maniac Mansion Virtual Machine is a set of game engine re-implementations, allowing games to run independently by original game scripts in various modern platforms. Originally designed to play LucasArts adventure games built with the SCUMM system, it now also enables users to enjoy a wide range of classic point-and-click adventure games. This paper analyzes the conceptual architecture of "ScummVM," which can be divided into seven sections:

- Derivation process
- Interacting parts
- Software development stages
- Concurrency
- Control and Data flow
- Use Case
- Lessons learned

In the derivation process section, we focus on the system architecture styles that break the “ScummVM” into each interacting part.Through the research, the interpreter and tiered architecture are two main suitable architectures that we will discuss. “ScummVM” uses an interpreter to re-implement the original game engine; interpret specific scripting languages of classic games and achieve high portability and compatibility. Also, its tiered architecture separates the system into distinct frontends, middle-ends and backends. Each tier of the architecture is responsible for specific tasks that ensure the added games run correctly on modern hardware.

In the interacting parts section, we discuss the five main components of the codebase structure that compose the tiered architecture of ScummVM. By allocating OSystem API and common code, Game engines perform their functions without worrying about the potential

platform specifics. OSystem API serves as a bridge between game engine and OSystem which defines available features a game can use, such as drawing on screen or receiving keyboard and mouse events. Backends are sort of platform dependent code and implements the APIs in various platforms. Common Code provides shared functionalities to promote code reuse and consistency. The GUI, a graphical user interface, offers the game launcher and options dialog for users to manage and configure their gaming experience.

In the software development stage, we have observed a progressive improvement in software performance with each upgrade. First, more and more games and game engines are supported so that users have more and more choices. Secondly, the constant updating of the GUI increases the usability of the game controls. Third, software is starting to utilize the CPU to solve the latency problem when rendering games, and performance has made a huge leap forward.

ScummVm's concurrency has been achieved through multithreading and coroutines. Both techniques contribute to enhancing ScummVM's user experience and performance. The implications of coroutines allow ScummVM to do effective input handling and rendering without overhead from traditional true multithreading.

The data flow shows how ScummVM analyzes user input and how data interacts in three layers.

In the use case part, the structure of ScummVM shows how it interacts with users. Users can access features such as file management, and archiving through a well-designed graphical user interface (GUI). Other components also play a vital role in the software, for example, the launcher can simplify navigation with game listings, search tools, and interactive buttons.

Derivation Process

In the beginning of the investigation, we study the documentation and wiki of the project to attempt to understand its conceptual structure. As a brief introduction, "ScummVM" was originally developed for many of the famous LucasArts games that were created using a utility called SCUMM (Script Creation Utility for Maniac Mansion). While the "VM" stands for Virtual Machine. It allows users to play certain classic graphic point-and -click adventure games, text adventure games, and RGPGs by uploading the game data files. It replaces the executable files of games in which the adjusted files allow game re-implements on most of the modern devices and operating systems such as Android, ios and PlayStation etc. Based on the initial learning of "ScummVM", when we discuss the possible architecture styles, several assumptions have been proposed.

First of all, we can figure out interpreter architecture as one of the ScummVM's architecture styles. There are several reasons to explain.

- Re-implementation of Game Engines: "ScummVM" re-implements the original game engine like SCI, plumber in order to interpret the game data file and scripts from the original games.

- Support for Multiple Languages and Platforms: The backends of "ScummVM" are the ports which contain platform specific code. It implements the OSystem APIs for various platforms in which the core interpreter code is compiled to interpret the game scripts across all the platforms.

- Script interpretation: Many classic adventure games are built using scripting language that are specific to the game engine. For instance, “Scumm” as a game engine for LucasArts games uses scripts to define the interactions, animations and dialogues. On this basis, “ScummVM” effectively simulates how the game engine executes and interprets these scripts in real-time.

Meanwhile, the other significant architecture style of “ScummVM” is tiered architecture. ScummVM is a three-tiered software. The frontends are a sort of VMs that each is responsible for running a subset of the supported games bytecode and handle game specific logistics like pathfinding, rending, effects, and build upon the middle-end. The middle end contains three major compositions, an API for graphics, sound and other available game features; implementation of the API using cross-platform code; common code for multiple game engines. The backends are the ports which implement the rest of APIs and port specific features for various platforms.

Interacting parts

The tiered architecture constructed by code base structure consists of five essential components.

1. OSystem API:

The OSystem API is the intermediary between the engine and the underlying operating system in ScummVM. It is essentially an abstraction tier that provides the game engines supported by ScummVM with a unified interface to interact with the underlying operating system. As such, the OSystem API interfaces directly with the engine, as well as with the operating system. We speculate that it may be implemented in a tiered architecture.

With the game engine:

The OSystem API provides a unified set of abstract interfaces for the various game engines in ScummVM (e.g. SCUMM, AGI, SCI, etc.), allowing the engines to handle platform-independent tasks such as capturing user input (keyboard, mouse, etc.), managing screen drawing (image rendering), playing audio, and file system manipulation (e.g., reading, saving progress of the game).

With the operating system:

The implementation of the OSystem API depends on the operating system on which “ScummVM” is running.The implementation tier of the OSystem API is called the backend code, and it provides implementations depending on the platform or operating system. For example, on a Windows platform, the OSystem API may use DirectX or WinAPI for graphics rendering, audio output and input management.

2. Backends:

The backends are re-implementations of the Osystem API for various platforms. The code in backends ports the software to different platforms and allows games to run on multiple operating systems without modifying the game engines themselves but simply switching the backends. The source code for backends is in the backends/ directory.

3. Game engines:

The SCI game engine handles user input, game logic, image rendering, audio playback, and interaction with the underlying system, but it does not interact directly with the operating system. It interacts with the underlying platform through an abstraction tier provided by “ScummVM”, and mainly calls various underlying functions such as audio, graphics rendering and input processing through the OSystem API. Therefore, its implementation needs to be highly modular, flexible and performant. We speculate that it may be implemented in a tiered architecture.

with the OSystem API:

The SCI engine calls the OSystem API to obtain system services without having to care about how the underlying operating system is implemented. For example, the SCI Engine uses the OSystem API to render game screens, capture user input, and play game sounds.

with Common code:

The SCI engine uses the common tools and codecs in Common code to process audio, image, and video data. In this way, the SCI Engine does not have to re-implement these functions but can call these function blocks through the Common code's unified interface.

4. Common code:

The common code in “ScummVM” refers to shared utilities and libraries that are used by some of the components of the system. Instead of writing their own version of code for different game engines and backends, the common code is reusable and can be accessed by any part of the system. For example, if several game engines are working, each of them needs to handle tasks like managing data, reading files, playing audio or images. Rather than every engine creates its own code to implement functionalities, common code provides a centralized, reusable set of tools that all engines can use which saves time and effort and ensures consistency across the system.

5. GUI:

The Graphical User Interface (GUI) is a digital interface in which a user interacts with a game launcher and options dialog. The launcher allows users to add or remove games. The options dialog is the actual game setting where users adjust their graphics, keymaps, audio, volume and file paths of the games just added. The GUI ensures users have a consistent and user-friendly way to interact with the system. The GUI code is found in the gui/ Directory.

Software development stages(evolution)

Work on ScummVM started in September 2001 by computer science student Ludvig Strigeus. In order to write his own adventure game, he wanted to understand the mechanics of existing game engines, specifically working to create a way to play *Monkey Island 2* on his Linux machine. Around the same time, Vincent Hamm was also looking to implement a SCUMM system player, and although he made more in-depth research into understanding how the SCUMM engine worked, he found that Strigeus had gone further, and the two co-produced the project. In October of the same year, ScummVM released their initial version, Version 0.0.1, which marked the official launch of the project and made it possible for players to play classic adventure games on modern systems.

Version 0.2.0: The core engine underwent a significant rewrite, enhancing performance and stability. A new in-game GUI with options like volume control was added, and the auto-save feature was introduced to improve user experience. More command-line options and configuration files were made available. In addition, support for non-SCUMM engine games debuted in *Simon the Sorcerer*, a major milestone.

Version 0.6.0: This release added several important engines for the first time, including *Broken Sword 1* and *Broken Sword 2*. It also introduced support for Flight of the Amazon Queen, and SCUMM V1 games like *Maniac Mansion* and *Zak McKracken*. Support for *Full Throttle* was added as well. Other improvements included enhanced subtitle options, new HQ2x and HQ3x scalers, and major sound code rewrites for better performance. The update also improved the iMUSE system for more accurate MIDI and AdLib GM emulation .

Version 0.8.0: This version introduced two new engines: the SAGA engine, which support games like *I Have No Mouth* and *I Must Scream* and *Inherit the Earth*, and the Gob engine, which support the *Goblins series*. This update mainly focused on expanding compatibility, adding new ports for PlayStation 2, PlayStation Portable (PSP), AmigaOS 4, and EPOC/SymbianOS.

Version 1.5.0: The update supported 11 new games, including *Backyard Baseball 2003*, *Blue Force*, and *Darby the Dragon*. Besides, it enhanced the gaming experience with significant improvements, such as updating the MT-32 emulation code to the latest Munt project snapshot, which led to better emulation. The predictive dialog look and various GUI were also improved.

Version 2.1.0: This release added support for 16 new games, including *Blade Runner*, which featured complex 3D environments. The update also brought significant optimizations to the gaming experience, such as enabling cloud support services for the first time, adding text-to-speech conversion, and improving GUI rendering and overall performance.

Version 2.5: This version marked the 20th-anniversary release of the project, bringing several significant updates. It expanded ScummVM's library by officially adding support for all AGS v2.5+ games and, following the 2.5.1 update, merged ResidualVM into ScummVM, integrating engine implementations for both 2D and 3D games. Numerous ports were optimized for better performance, and HiDPI support was introduced to enhance visuals on high-resolution devices. Additionally, Discord Rich Presence was integrated, allowing players to share their game status easily.

Version 2.8.0: This latest release introduced support for more games, focusing on CPU graphics acceleration, enhanced rendering, and performance optimizations.

With over 20 years of continuous development, ScummVM has evolved into a powerful cross-platform game engine emulator, enabling users to relive classic adventure games on modern systems. Beyond emulation, ScummVM continuously enhances the gaming experience by increasing resolutions, adding multi-language subtitles, and more.

ScummVM's development has made a crucial contribution to preserving gaming history and cultural heritage, allowing new generations of players to experience these timeless classics. Its maintainability and regular updates ensure compatibility with modern operating systems and hardware, overcoming challenges like compatibility issues and graphical enhancements. By supporting a vast library of games and constantly expanding its capabilities, ScummVM

proves that in the world of classic adventure games, the only limit is imagination, not technology.

Concurrency

Concurrency is a vital part of the software system's conceptual structure. ScummVm, the typical classical point-and-click adventure game, has applied concurrency in several fields, such as multithreading and Asynchronous I/O. Concurrency plays an important role in enhancing performance and user experiences. This part mainly focuses on analyzing the concurrency achieved by multithreading simulation and how concurrency delves in other areas.

Multithreading simulation:

Concurrency through multithreading can benefit the ScummVm system. Concurrency allows parallelism, and multithreading archives concurrency by allowing different threads to be executed on different cores at the same time. However, in real life scenarios, it is impossible to always let tasks run in parallel. Therefore, the ScummVM codespace uses Coroutines to simulate multithreading to improve the performance in input handling and rendering and to improve the user's experience.

By analyzing Coroutines API's document, we can see that control is the critical characteristic of Coroutines, and that feature prevents blocking behavior. The coroutine part also chooses single-threaded context to reduce the context-switching overhead and other risks that may be caused by synchronization. Additionally, there is the CoroutineScheduler, which can decide whether each Coroutine runs, yields, or resumes tasks according to its state. The most significant difference between Coroutine and actual multithreading is that Coroutines run on a single thread and achieve parallelism by cooperative multitasking; in contrast, real multithreading operates on multiple different cores.

Coroutines have improved the user's experience in several fields. First, they guarantee efficient input handling; I/O operations like waiting for user input or loading resources would not impact ScummVm execution since I/O operations can be managed by Coroutines asynchronously. When a coroutine waits for user input, CoroutineSchedule can control other coroutines to continue processing tasks like updating the game state or playing sound effects. Therefore, the ScummVM users would not get the stacked game and would have a seamless experience. Second, Coroutines enable smooth graphical rendering in ScummVM since enough Coroutines cooperate with each other to achieve fluid visual experiences. For example, one coroutine is responsible for updating the main character's position, and Another one is responsible for handling background scenery change. Thirdly, the cooperative mechanism in Coroutines reduces the need for control flow in the game, improving responsiveness somehow. Since Coroutines simulate multithreading, whenever a player interacts with an object, the corresponding coroutine can generate appropriate responses(like playing an animation or displaying dialogue) as soon as possible. Thus, ScummVm can provide more engaging and dynamic gameplay experiences.

Other areas:

ScummVM employs an event-driven architecture alongside coroutines to maintain the concurrency and avoid blocking the main execution flow. When an event occurs, the event-driver architecture will assign a specific coroutine thread to handle the task. From the script of scummVM fame, we can see the developer determines the flow of the program by events such as clicks or critical presses from graphical user interfaces(GUI). There are defined callbacks for event response in the ScummVm script. Both event-driven architecture and coroutines contribute to concurrency in ScummVm since they enable ScummVm to handle multiple events in parallel. ScummVM users get a smooth, immersive game experience since the game performance gets enhanced.

Maintaining concurrency in the resource management field is also very important. The developer chooses to use concurrent access to support the concurrency. Large amounts of data are scattered across various folders; simultaneous access to files improves speed and performance by allowing multiple files to be read or written simultaneously. This technique would be beneficial whenever the ScummVM needs to load significant assets like high-resolution images or long audio files. Concurrent access improves game responsiveness by enabling faster loading times. In an ordinary game system, the game needs to process different operations sequentially, and this mechanism always causes significant wait time. In contrast, concurrent access reduces the significant delays and load time since there is no need for all sets(e.g., sound effects, images, etc.) to be retrieved.

ScummVM supports multiplayer features and requires uploading online content a lot of the time. Therefore, concurrency is critical for managing network requests and responses. Developers also need to address the issue of ScummVM games getting frozen when ScummVM handles network requests. In the network area, asynchronous I/O operation ensures gameplay remains fluid by running net-wrok-related operations concurrently.

Control and Data flow

This diagram shows the process that occurs when a user inputs content through the GUI. Initially, the input is directed to the front end, which comprises a series of VMs (Virtual Machines) within the ScummVM front ends. From there, it is processed by a script interpreter that manages game-specific logistics. Next, the data transitions to the middle end, which includes common code and cross-platform code, in the cross-platform code which implements part of the OSystem API. On the other hand, the common code can provide image, video, and audio codecs to game engines. And then the game engine can use the previously provided codecs to output from the speaker and monitor. Finally, the data reaches the back end, where the ports implement the remaining API.

In summary, the diagram outlines user input through a GUI, implemented in three parts, processed by a virtual machine and script interpreter to multimedia processing and finally API implementation.

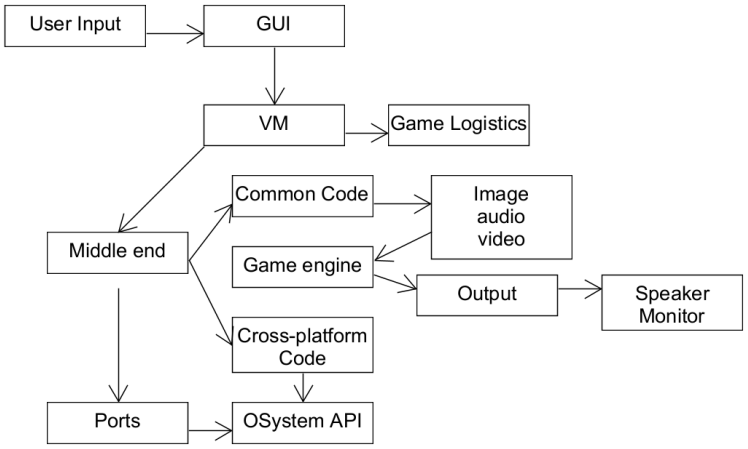

Figure 1: Box and arrow diagram

Use Case

This pre-model provides an overview of how the ScummVM system interacts with its two main functions (developers and users), and provides an overview of each function and its available features.

ScummVM offers the user a variety of features. First, users can operate the software through the graphical user interface. The GUI then transmits input commands to the launcher, which mobilizes various functions in the software, such as searching for games, browsing the game library, and changing and applying settings. Users can also manage game files by adding, deleting and managing archives. In addition, when users encounter problems with the software, they can use the help function to find solutions.

The developer's main responsibilities include adding and maintaining the game engines, checking whether added games are supported, managing cross-platform code to ensure that ScummVM runs on different operating systems and hardware, developing ports to extend compatibility, and resolving issues raised by players.

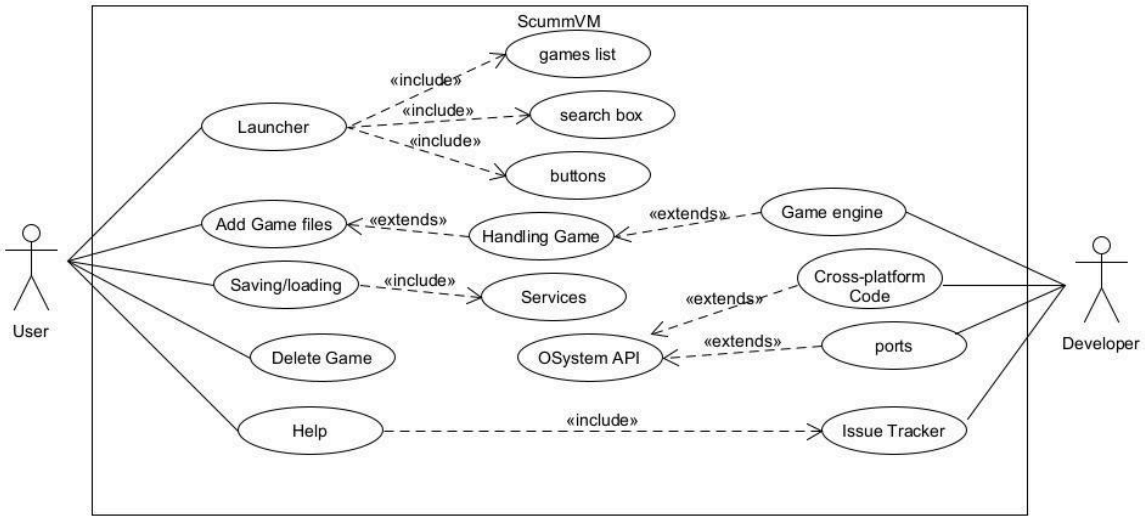

Figure 2: Use case

Use case 1: Playing games

This diagram shows how to launch, manage, and play games in ScummVM. After launching ScummVM using the GUI, users can change the settings. If the users make any changes, the system will apply them; if not, the default settings take effect. Next, the users launch the game by exploring the game library through the launcher. Using the search box, users will look for the game and determine whether it is included in the list of games. If the game is found, the system will match the game with the proper engine. If the engine match is successful, the system will load the necessary data from the HFS/HFS+ storage, including the game body and game archive. After loading the game's files and setups, the engine performs a number of decodes and conversions before displaying the visuals and audio on the GUI. Users return to the launcher to try again or choose a different game if the engine is unable to match, and an error message appears on the GUI. Additionally, the player can end the game whenever they want. The system saves users' progress, stops the engine, and releases all resources when the user decides to exit the game using the GUI. The user will be directed back to the GUI to take further action if the system is unable to locate the game in the list of games and displays the message ", (game not found)". Ultimately, the user can end the session by closing the application, which ends the ScummVM launcher and the game.

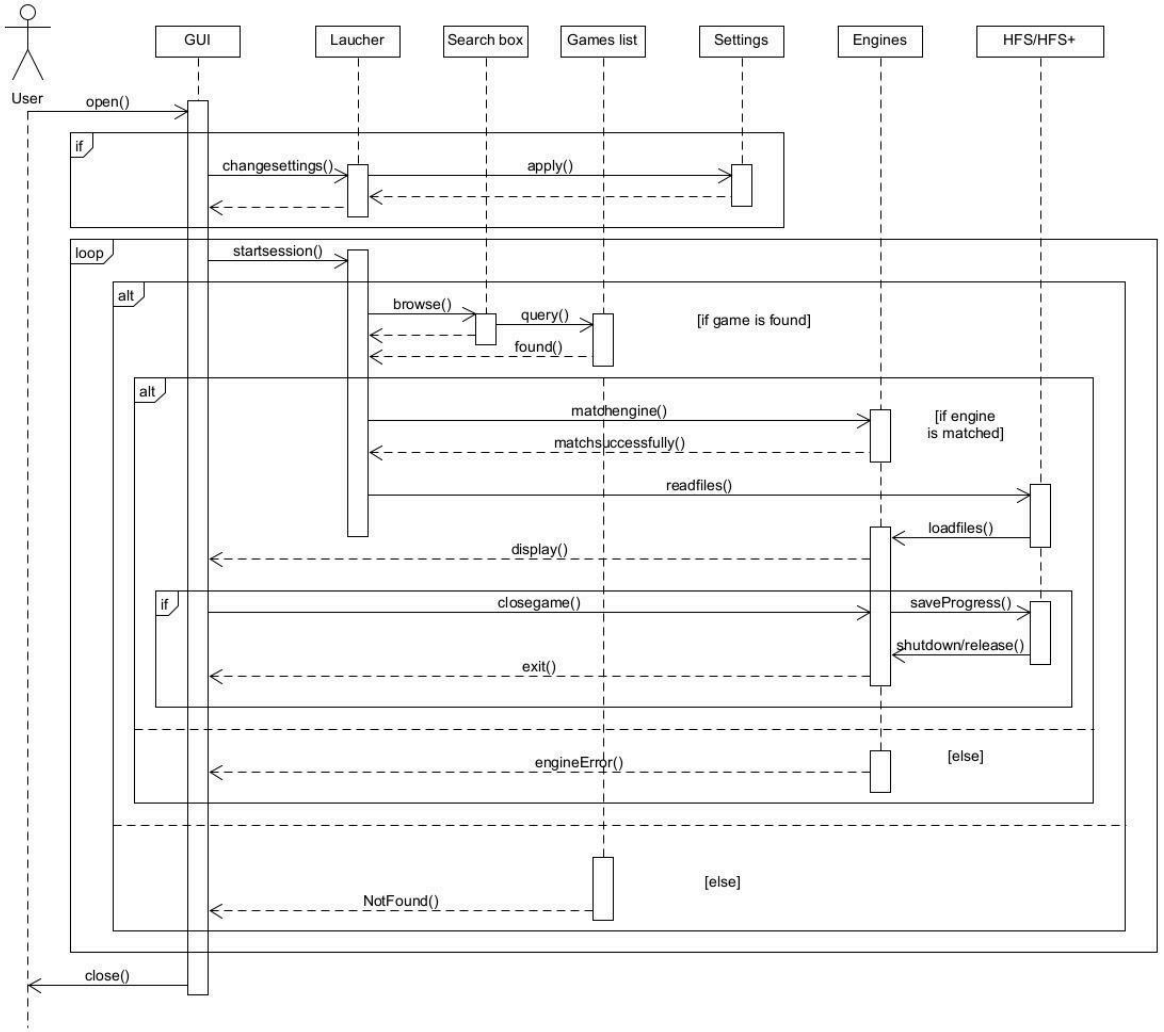

Figure 3: Sequence diagram of use case 1

Use case 2: Memory function

This diagram depicts the memory related cases and help in ScummVM. When a user opens ScummVM, they can add new game files in the launcher through the GUI. The uploaded game will first confirm, its engine is compatible with the virtual machine. The game is then added to the game list and its name is displayed on the GUI. At the same time, the game file is saved to HFS/HFS+ and services. This process can be repeated and users can also add multiple games at the same time. If the user wants to delete a game, the name will be removed from the game list, but this will not affect the content of the game stored in HFS/HFS+ and the services. After starting a game, the system will automatically save progress every five minutes. If the user wishes to save the progress themselves, they can do this through the GUI by opening the menu in the launcher ,and then clicking the save button to save the game progress to the save page. All saves will be saved to the cloud service or local storage, and will be displayed in the user's launcher. When the user wants to load an save, they can similarly use the GUI to open the menu in the launcher, click Load, go to the save page where they can select an archive and load the corresponding progress, and the system will find the corresponding progress from the services to restore it. If users choose to delete the save, they can also complete the operation in the save page to delete the corresponding data in services. At the same time, if problems are encountered during the process, the user can open the menu and select Help for support. Finally the user exits the game and closes the launcher.

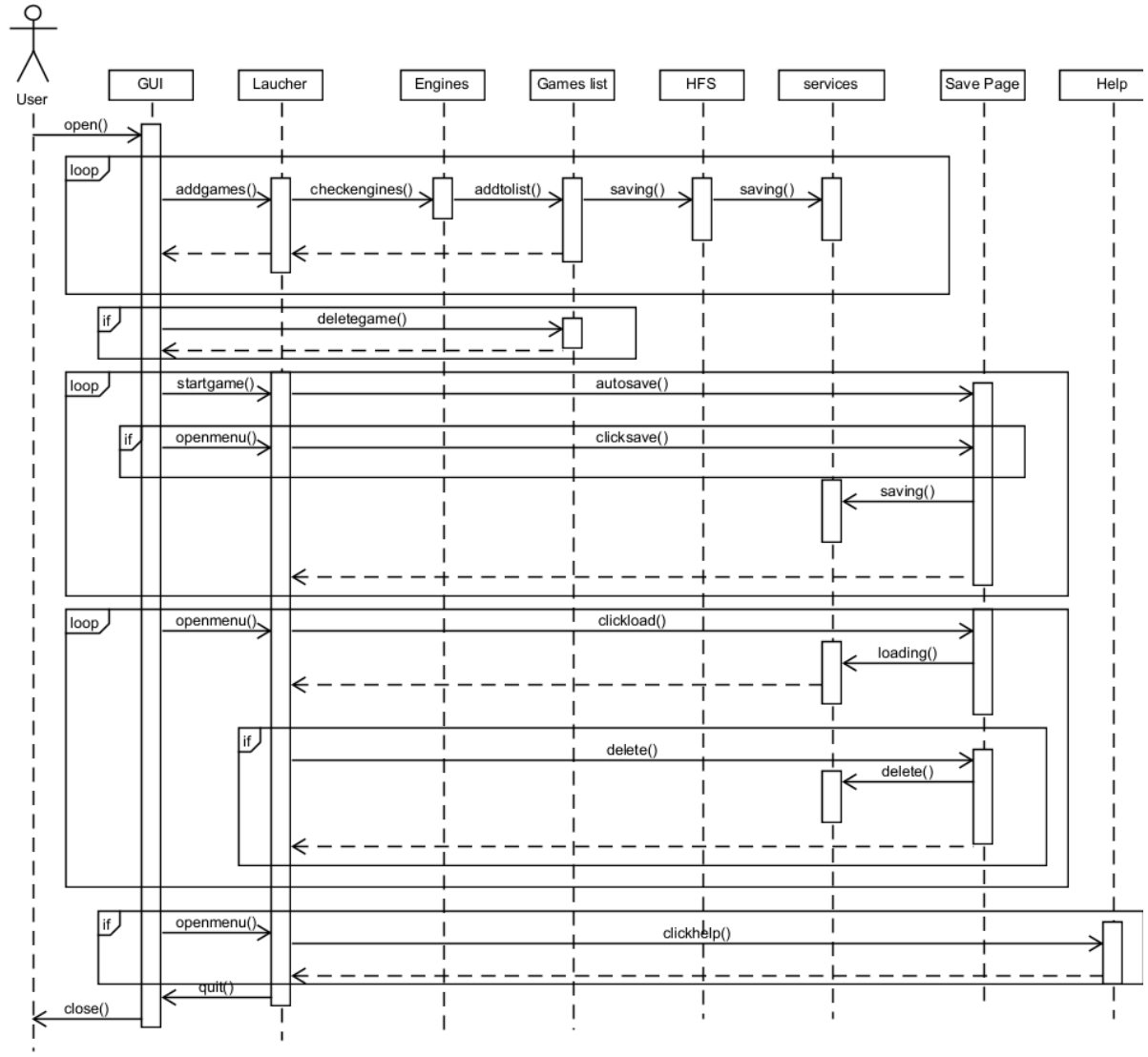

Figure 2: Sequence diagram of use case 2

What are the implications for division of responsibilities among participating developers?

Modules have no dependencies:

ScummVM supports multiple game engines (e.g. AGI, AGOS, ADL, etc.). These engines are relatively independent, allowing different developers to work on different engine modules at the same time without worrying about too many dependencies.

The user interface is also relatively independent. User interface development can be synchronized with engine development. The user interface team can work separately from the game engine team, focusing on the user experience while maintaining the necessary interface communication with the engine developers.

File systems and data processing modules are responsible for loading, storing, and processing game resources. These modules can be developed independently, as they provide the base functionality for the upper-level engine and user interface modules.

Modules with dependencies:

Although the audio and video processing parts can be developed in parallel, they are highly coupled to the game engine, especially for certain games that require specific optimizations to the engine. Therefore, the development of these modules must be closely coordinated with the engine development team.

Input modules are tightly integrated with the user interface and the game engine and are responsible for capturing and transmitting user input. Therefore, the user interface and the engine interface must first be clearly defined.

In summary, when assigning tasks, priority must be given to identifying modules that can be developed independently, and for highly dependent modules, their dependencies must be clearly defined, and the development of dependent modules must be carried out after the completion of the core functional modules.

Cross-team collaboration:

When multiple teams develop different modules, the interfaces and communication mechanisms of each module must be defined through detailed design documents and interface protocols. Different engine modules may be handled by different teams. By defining clear API interfaces, teams can ensure low coupling and high collaboration between different modules.

The development of ScummVM relies on the unified management of the code base. Through a branching mechanism, teams can develop independently on different branches and then consolidate their work into the master branch through code reviews and merge requests. Regular integration and testing helps to identify issues early and reduces the risk of cross-team collaboration.

Teams can stay in communication with each other through regular meetings or online tools such as Zoom, email, and so on. Each team should report on its development progress on a regular basis to ensure that everyone has a clear picture of the overall progress of the project.

To ensure smooth module integration, a comprehensive automated testing framework is required to perform cross-module testing.

In summary, the above components enable effective task division and cross-team collaboration to ensure efficient and stable project development.

Lesson Learned

Interacting parts

The five components interact with each other to make the software work, and the GUI acts as a frontend that allows the user to manipulate the entire software in a clear and simple way and start playing. The game engine handles user input and a range of game runtime functions. Code provides shared utilities that can be reused between the game engine and the back end, thus simplifying the development process and reducing redundancy. OSystem APIs allow game engines to interact with the operating system without the need for platform-specific details. The backend ensures cross-platform compatibility by reimplementing the OSystem API to allow the software to be ported to different platforms.

Concurrency

The ability to handle multiple tasks concurrently is important for SCummVM game play since this game requires a lot of I/O input from the user simultaneously. Unconventionally, ScummVm chose to apply coroutines to simulate the multi threads working instead of using real multithreading. Therefore, no matter how much I/O instruction Scumm receives from users, the user experience would not be impacted because ScummVm can handle tasks concurrently.

Control and Data Flow

The ScummVM's architecture has been well developed to deliver a seamless game experience for users and provide tools for developers to maintain the ScummVM system. The three diagrams clearly illustrate the coordination between Front End(GUI), Script Interpreter, Middle End, Game Engine and Back End inside the ScummVM. The well-structured interactive system ensures that ScummVM can process a variety of I/O requirements from users and developers can easily maintain the ScummVm's systems.

Division of responsibilities

ScummVM's development follows a modular approach, allowing it to be independent of the game engines, like UI, and file systems. Modules such as audio, video, and input require closer coordination. Maintain cross-team collaboration with defined interfaces, branch-based code management, and automated testing to ensure effective task partitioning and smooth project integration.

Reference

ScummVM. (2024) Release Note. Retrieved from
https://docs.scummvm.org/en/latest/help/release.html

ScummVM. (n.d). Welcome to ScummVM! Retrieved from
https://docs.scummvm.org/en/v2.8.0/

ScummVM. (n.d). Programming a new game. Retrieved from
https://forums.scummvm.org/viewtopic.php?t=7886

ScummVM. (n.d). Coroutines support for simulating multi-threading. Retrieved from
https://doxygen.scummvm.org/d7/db8/group__common__coroutine.html

Wikipedia. (2022). SCI/SCI32 Mac Support Status, from
https://wiki.scummvm.org/index.php?title=SCI/SCI32_Mac_Support_Status

Github. (2021) engines.awk[Source code], Retrieved from
https://github.com/scummvm/scummvm/blob/master/engines.awk

Dutt, A. (2023) Implementing DLC downloading through cURL, Retrieved from
https://blogs.scummvm.org/ankushdutt/2023/07/15/dlc-curl/

Wikipedia. (2023). Developer Central, Retrieved from https://wiki.scummvm.org/index.php?title=Developer_Central