CISC 322/326
Assignment 1
Conceptual Architecture of ScummVM and SCI Engine
Friday, October 11, 2024

Group 21: We Love Josh

Nathan Chow 21nwc1@queensu.ca
Ben Jacoby 22bj@queensu.ca
Savannah Han 21sh113@queensu.ca
Amanda Li 22ayl@queensu.ca
Kristen Lee 21kl52@queensu.ca
Cynthia Wang 21cyw4@queensu.ca

Table of Contents

Abstract **3**
Introduction and Overview **3**
Derivation Process 3
Architecture **4**
Subsystems 5
OSystem API 5
Backends 5
Common 5
ScummVM GUI 6
Graphics 6
Sound 6
Event Manager 6
Resources 6
SCI VM 7
Changes in the System 7
Control Flow 7
Division of Responsibilities 8
External Interfaces **9**
Use Cases **9**
Use Case #1: Adding a Game 10
Use Case #2: Playing a Game 10
Use Case #3: Saving a Game 11
Conclusions **11**
Lessons Learned **12**
Data Dictionary **13**
Naming Conventions **13**
References **14**

Abstract

ScummVM—Script Creation Utility for Maniac Mansion Virtual Machine—is an open-source video game program initially developed by Ludvig Strigeus on September 17, 2001 [1]. ScummVM allows for games to be run on a variety of OS systems such as Windows, macOS, Atari, MorphOS and Linux as well as some mobile systems such as iOS and Android [2]. Originally, ScummVM was made with the intention of supporting many older disk operating system (DOS) based games published by American video game licensor, LucasArts, but overtime would support a greater variety of text-based or point-and-click graphic adventure video games developed by other video game studios such as Revolution Software, Westwood Studios, and Cyan, Inc [2].

The SCI engine—Sierra Creative Interpreter engine—is a game engine that runs on top of ScummVM [3]. The SCI engine was used for games such as King's Quest IV published by the video game company, Sierra [4].

Based on ScummVM's use of an interpreter to run game scripts, our team had concluded that ScummVM, and in turn the SCI engine, likely use a combination of the interpreter architectural style with elements of the layered architectural style.

Introduction and Overview

In this report, our team discusses the conceptual architecture of ScummVM. ScummVM is an open-source video game program that allows users to run a variety of games, often older games originally produced by LucasArts, as long as the relevant game's data files are available to access [5]. This function is achieved via rewriting the original game's executables.

Based on this information, we determined that ScummVM, and in addition, the SCI engine, follows the interpreter and layered architectural styles. The interpreter architectural style consists of an execution machine, the program being interpreted, and memories for both of these states. ScummVM follows the interpreter architectural style as SCI interprets the original game's data files—those that handle sound, graphics and game logic—and then attempts to rewrite these files so they can be used on modern operating systems.

As shown in Figure 1 below, which contains the conceptual architecture for ScummVM and SCI engine, the manner in which the determined subsystems—OSystem API, Backends, Common, ScummVM GUI, Graphics, Sound, Event Manager, and Resources—interact with each other follows a layered architectural style. There are four layers we have determined: the game layer containing the Graphics, Sound, and Event Manager subsystems, the game engine layer containing SCI VM and Resources, the hardware independent layer containing OSystem API and ScummVM GUI, and the hardware compatibility layer containing Backends.

Derivation Process

Our research and analysis was largely based on information provided by the official documentation for ScummVM and the SCI engine, as well as a mixture of other primary and secondary sources such as the main source code repository for ScummVM on Github, and the

ScummVM Wiki. Based on the information provided, our team began outlining the subsystems for ScummVM such as the OSystem API, the Backends, the engines which include SCI, and the GUI. Outlining the major subsystems enabled us to create an initial draft of a box-and-arrow diagram, which showed that the interactions between the subsystems can be viewed in a layered manner. Based on the purpose of ScummVM, which is to essentially enable portability for older games through its use of the SCI engine's interpreter, our team was also able to conclude that ScummVM and the SCI engine follow an interpreter architectural style. We also identified and created sequential diagrams for the following major use cases: adding a game to ScummVM, playing a game in ScummVM, and saving a game in ScummVM. With our subsystems identified and diagrams completed, our team concluded that ScummVM and SCI follow an interpreter architectural style with some elements of the layered architectural style.

We considered some alternatives for the conceptual architecture, such as putting Common inside the Hardware Independent Layer and adding an Engines subsystem in between OSystem API and SCI VM. However, since Common contains utility classes and resources used by most other subsystems, and since it doesn't depend on anything else, we thought it was best to separate Common from the layered architecture so that subsystems from all layers can use it. We also ended up scrapping the Engines subsystem because having both Engines and SCI VM would be redundant.

Architecture

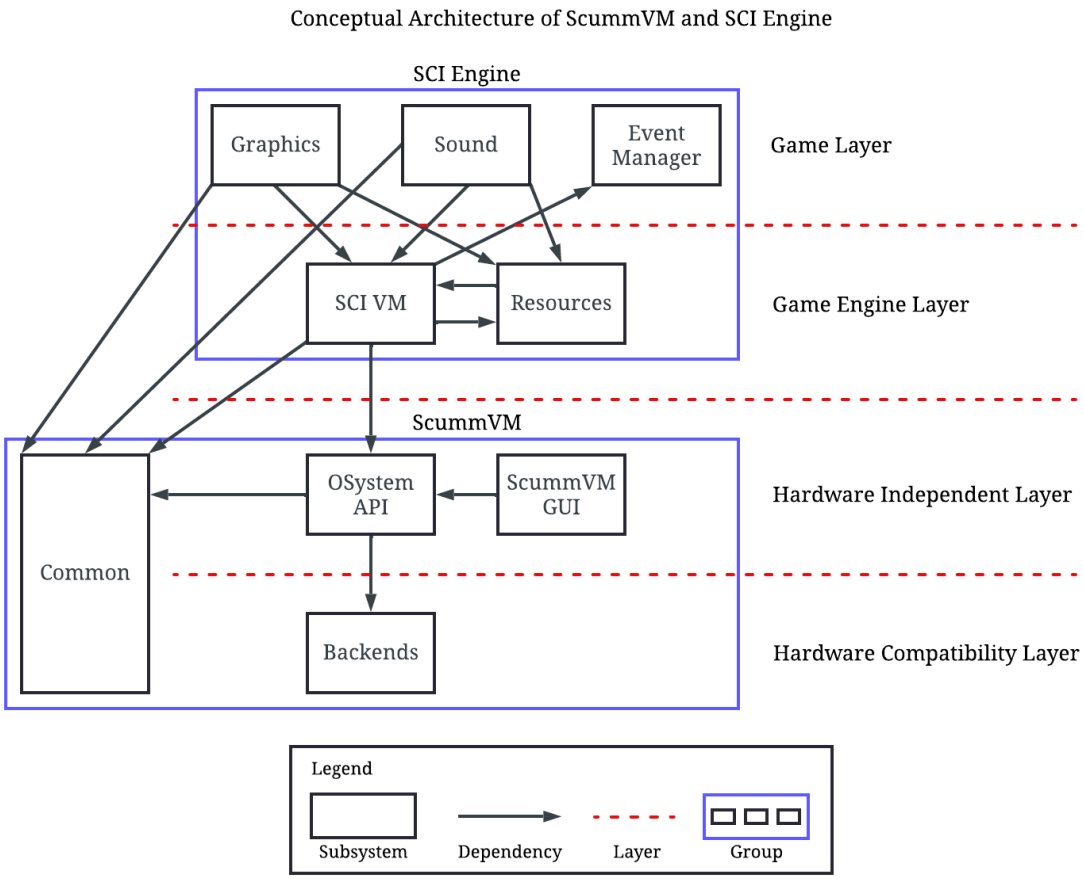

Figure 1. Proposed conceptual architecture of ScummVM and SCI engine

Above is our proposed conceptual architecture of ScummVM running SCI engine. The architecture style is a hybrid of the layered and interpreter styles. There are 2 groupings of subsystems: Common, OSystem API, ScummVM GUI, and Backends make up ScummVM while Graphics, Sound, Event Manager, SCI VM, and Resources make up SCI engine.

The interpreter style portion of the architecture is created in the SCI engine through the relationship between Resources and SCI VM. SCI VM contains an interpreter for reading and executing game scripts which are given to it by the Resources subsystem. This results in an interpreter style relationship between the two subsystems in the high-level architecture.

Our architecture consists of 4 distinct layers: the hardware compatibility layer, the hardware independent layer, the game engine layer, and the game layer. The hardware compatibility layer handles platform specific issues to make ScummVM platform independent. The hardware independent layer is the hardware independent platform that the engine "sits" on top. The game engine layer implements the backend of the game engine handling logic and file storage, which in this case is the SCI engine. Finally, the game layer consists of components that interact with the user, like drawing graphics, playing sounds and reading user input. Aside from the four layers, there is also the special subsystem Common, which is not part of any layer and has a collection of utility classes and functions for use by other subsystems in multiple different layers.

Subsystems

OSystem API

OSystem API is used to keep ScummVM platform independent and protects the GUI and game engines from needing to access the OS directly, instead acting as an intermediary. It does this by providing an API for communicating with the backends which handle the platform specific tasks. The OSystem API provides various functions that define what an engine or game can do within ScummVM; for example, drawing graphics to the screen and detecting mouse and keyboard input. OSystem API receives requests from the ScummVM GUI as well as the game engine running on top of it to pass along to the backends, and sends a response back once the request is finished. This component could cause major bottleneck issues in the system should it slow down, as nearly all data flows through OSystem API.

Backends

The Backends component consists of monolithic backends, which is a large unified architectural system that couples various smaller services [6], and modular backends, which extend and add additional functionality to other existing modules. In the context of the ScummVM system, the Backends implement the OSystem API for various platforms.

Common

The Common component refers to "common code", which consists of various utility classes and relevant image, video, audio and game engines.

ScummVM GUI

The ScummVM Graphical User Interface (GUI) component consists of elements such as the options dialog and game launcher. It provides the user with a way to easily interact with the system.

Graphics

The Graphics subsystem is part of SCI and is used to draw graphics to the screen as the user plays a game. It uses four different types of image resources to create graphics: Pic, View, Font, and Cursor for backgrounds, images, text, and the mouse pointer respectively. Graphics uses three different maps to display these resources: the visual map for displaying the actual visuals the player will see, the priority map for storing information about the depth of the screen, and the control map which stores other special information. Visuals are drawn using a data structure called a port (different from ports in networking). Each port has an origin (coordinates) and size; they also store information about graphical elements within them (like cursor position and font). Ports act as subsections of the screen where graphics can be drawn, and images are drawn into a port using coordinates relative to the current port.

Sound

The Sound component is part of the SCI engine. There are two sound resources that SCI uses: sound and patch resources. Sound resources contain music data stored in a modification of the MIDI standard. Patch resources contain mapping information related to the instruments used in the sound resources.

Event Manager

The Event Manager reads user events by receiving events from device drivers and sends them to SCI VM for further processing. The inputs that Event Manager can read include: mouse input, keyboard button presses, and joystick input if applicable. Specifically, it can detect 5 different events indicated by their hex codes: 0x00 (Null event), 0x01 (Mouse button event), 0x02 (Mouse button release event), 0x04 (Keyboard event), and 0x40 (Movement event). Each event is used for a different function. The null event is an empty event generated when a script needs to receive an event but there is no event to pass into the script. The mouse button and mouse button release events are used to read mouse inputs (button presses) as well as the mouse position. The keyboard event is read every tick (at 60hz) and detects the specific key that was pressed. Finally, the movement event is used to read input from the joystick; however, if requested, keyboard events can also be converted into movement events.

Resources

The Resources subsystem can store and retrieve files (resources) for the SCI engine. There are two ways resources are stored in SCI: resource files, and external patch files. External patch files are uncompressed and easy to read, and as a result they take precedence over resource files. In contrast, resource files can be compressed and may need to be uncompressed in order to be read.

Additionally, there are 4 types of resources: graphics resources, sound resources, logic resources, and text resources. These resources are used by the Graphics, Sound, and SCI VM components.

SCI VM

SCI VM—Sierra Creative Interpreter—is a stack-based virtual machine. It couples various extended functions responsible for graphics, sound, and input. It also contains functions for reading and writing data from the hard disk, as well as handling logic and arithmetic. SCI VM contains the interpreter, which is responsible for reading scripts.

Changes in the System

Since ScummVM's initial 0.0.1 release on October 8, 2001, the system has gone through a variety of changes to add new features, ports, games, and game engines, and to make improvements to its codebase so that the system can be more easily maintained.

The first major change was implemented in release 0.2.0, where the core engine of the system was rewritten and support for the first non-SCUMM game was added. New ports and platforms were also added, including MorphOS, Macintosh, and Solaris. As ScummVM continued to be developed, more features were added to make the system more user-friendly, such as launcher dialog and new in-game GUI in release 0.3.0b. As well, ScummVM continued to gain code improvements such as new saving and loading code in 0.4.0 and autosaving for all supported game engines in 2.2.0 [7].

Many ports were added throughout the course of its lifetime, such as Playstation 2 in 0.8.0, Nintendo DS in 0.9.1, iPhone in 0.11.0, Android in 1.2.0, Playstation 3 in 1.4.0, Nintendo 3DS in 1.8.1, and Nintendo Switch in 2.1.0. This allowed users to access ScummVM more easily on a variety of platforms and devices. Additionally, different game engines were gradually added, such as the SAGA and Gob engines in 0.8.0, the kyra engine in 0.9.0, the Cinematique evo 1, Sierra AGI, and Parallaction in 0.10.0, and SCI in 1.2.0, which was a major SCI-related release that added support for over 35 Sierra adventure games [7].

Other than the major releases described, there were multiple other releases with minor changes such as bug fixes, improved music, improved UI, language translations, and miscellaneous improvements and optimizations.

The system underwent major code rewrites during its early stages in order to be easier to maintain in the future, and has had plenty of bug fixes since then. Because of this, we expect that ScummVM will be easier to maintain in the future, especially since it already supports many platforms and game engines whose code can be reused. ScummVM is now on version 2.8.0, and we expect there to continue to be more games, game engines, and ports added as new devices are being used by the general public.

Control Flow

When the user's operating system needs to be accessed or communicated with, the flow of control moves downwards. The subsystems that the user interacts with in the Game Layer, such

as the Event Manager which detects the user's inputs, pass data to the SCI VM subsystem which handles logic for the SCI game engine. If resource files are needed, data is passed to Resources, otherwise data flow travels to OSystem API in order to contact the Backends which can interact with the OS.

To return information from the OS to the user, control flows back upwards, through the Backends and OSystem API, and then to the game engineer interpreter, which would be SCI VM in this case. SCI VM then decides what to do with the information and passes control to the appropriate subsystem, such as to Sound to play sound, or Graphics to display visuals.

Nearly all data goes through the OSystem API subsystem since it shields the game engine and the ScummVM GUI from the OS, so anytime the ScummVM launcher or the SCI game engine needs to pass or receive information from the OS, data will flow through OSystem API.

No mention of concurrency could be found in the ScummVM documentation, however, we presume that concurrency is used in some areas so that subsystems can run in parallel. Since the SCI VM subsystem runs on top of OSystem API, which runs on top of Backends, we assume that concurrency is used so that Backends can run at the same time as SCI VM. Additionally, in order to display graphics, play sound, and handle user input at the same time while the user is playing a game, concurrency is used in the game engines so that Graphics, Sound, and Event Manager can run in parallel.

Division of Responsibilities

ScummVM is open-source, so any developer around the world can contribute to its development if they wish by making a pull request on the ScummVM Github repository. To start, interested parties can read through the ScummVM documentation detailing how to contribute, which subsystems they can work on, and which tasks are a priority on the main TODO list. Developers can choose any area to work on, whether it's backend or adding support for new game engines, as long as they follow the guidelines on how to properly format their code and make a pull request. Contributors to ScummVM are volunteers, and thus do not get a salary [8].

Each team member is responsible for their own area, such as a certain game engine, and they are free to add code in that area as long as the general guidelines are followed. However, if they want to contribute to other areas, they need to talk to the developers in charge of said areas. The exceptions are the OSystem and Common subsystems; code for those two areas must be discussed with the team leads before implementation [8].

Although anyone can work on whichever area they want, everyone needs to adhere to the proper pull request process, follow the coding conventions, and ensure their code is compilable and working. If a developer does not follow the proper conventions or pushes problematic code, issues can occur in the codebase. To prevent this, each pull request goes through a review process lasting two weeks where any member can voice their concerns, and if there are no issues after those two weeks, the code can be successfully merged. This practice ensures quality code in the ScummVM repository [8].

External Interfaces

ScummVM relies on several external interfaces to function effectively, allowing it to communicate with the user, the operating system, and other key components. These interfaces are crucial in ensuring that ScummVM will be able to support seamless gaming for a variety of platforms.

The Graphical User Interface (GUI) is one of the most prominent external interfaces with which users will interact. It provides a GUI whereby the user can add games, start the games, change their settings, and save or load game states. This GUI interfaces with other subsystems like the OSystem API to actually get things done. For example, the GUI may interact with OSystem API to gain file access or to detect input from input devices like keyboards and mice. It also communicates with the SCI VM and Resources subsystems for the execution of games and the handling of game files and saved states.

Another critical external interface is the Operating System's File System through the ScummVM Backends subsystem. Since ScummVM needs to run on a number of different platforms, it also has to interface with the OS to take care of tasks such as browsing for files, loading game data, and accessing save files. Likewise, the OSystem API provides an abstraction layer between the GUI and file system, thereby allowing ScummVM to load game data and save game progress onto systems like Windows, macOS, and Linux.

ScummVM interacts with the game data files contributed by the user. These files are required in loading the game and include game assets such as scripts, sound, and graphics. The Resources subsystem will load and interpret such files while the SCI VM executes them. Save files, which maintain the player's progress, constitute another type of game data that ScummVM reaches through this interface.

Finally, ScummVM needs to interact with some input devices, such as a mouse, keyboard or touchscreen depending on the platform. Through these input devices, the user can both control the game and interact with the ScummVM GUI. The Event Manager subsystem processes input from these devices, and sends relevant commands via the OSystem API to either the game engine or to other subsystems.

In short, these external interfaces enable ScummVM to interface with the user, the OS, game data, and input devices to ensure flexibility and usability while playing games on every platform [9].

Use Cases

ScummVM and its available game engines are most commonly used to play text-based or point-and-click graphic adventure games. Thus, we propose three use cases to demonstrate the use of and the interactions between the major subsystems of our proposed conceptual architecture.

Use Case #1: Adding a Game

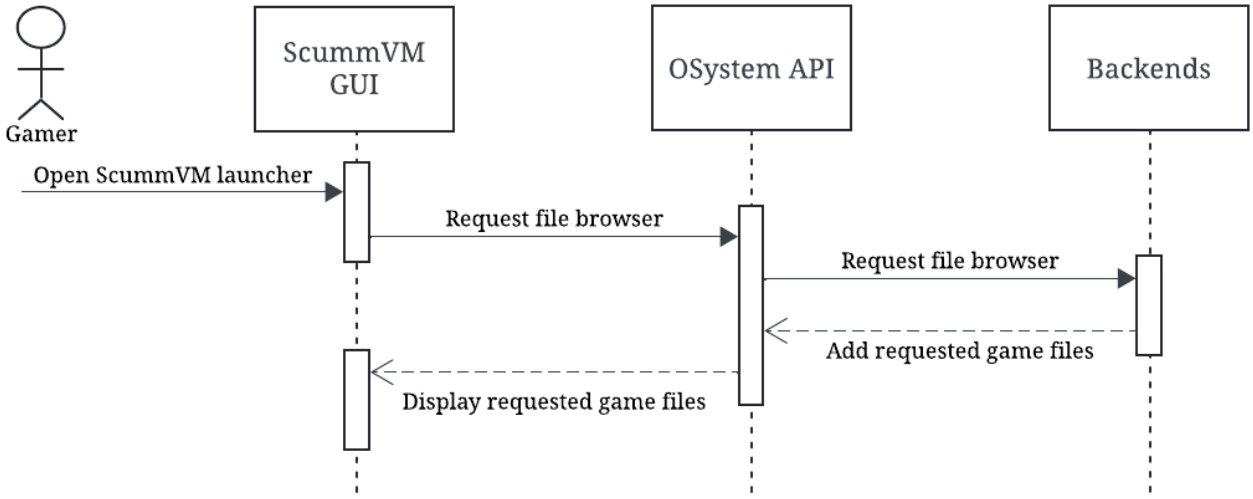

Figure 2. Sequence diagram for use case of adding a game

The first use case involves the user adding a game into their ScummVM launcher. This use case was chosen because in order to play the intended game, the game must first be added into ScummVM using existing game files.

In order to add a game, the user first opens the ScummVM launcher, which uses the ScummVM GUI subsystem. The user then requests to add a game, which prompts a file browser to open. In this use case, the user's OS file browser is used, so the subsystems should eventually interact with the OS. The request for the file browser goes through the OSystem API first, since it acts as an intermediate between the ScummVM GUI and the Backends, and then it reaches the Backends which can interact with the user's specific OS. Once this is done, the files are sent back to be displayed on the ScummVM GUI, and the user can choose the game files to add into ScummVM.

Use Case #2: Playing a Game

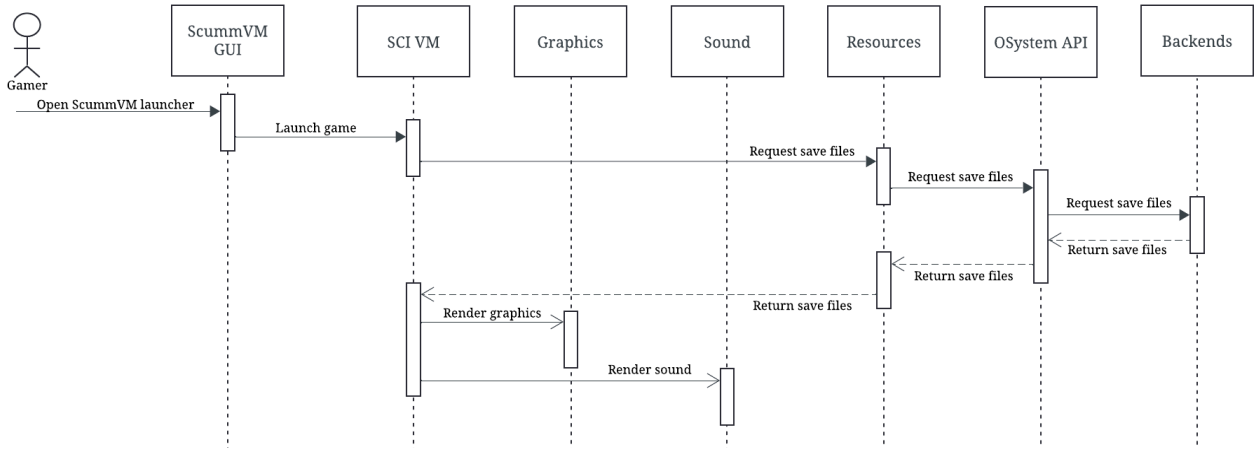

Figure 3. Sequence diagram for use case of playing a game

The second use case involves the user launching a game with the intention of playing it. This use case was chosen because the intended use for ScummVM is for users to play their desired games inside the program and matching game engine.

To play a game, the user first opens the ScummVM launcher much like the previous use case, which uses the ScummVM GUI subsystem. In this use case, the user is playing a game that uses the SCI game engine. Thus, the SCI VM subsystem is used. The game's save files are requested, which is handled by the Resources subsystem, and in order to access the user's OS file browser, the OSystem API subsystem and then the Backends subsystem are contacted. After returning the save files, the SCI VM subsystem can request that the graphics and sound for the game are rendered using the Graphics and Sound subsystems. Afterwards, the game is ready to play.

Use Case #3: Saving a Game

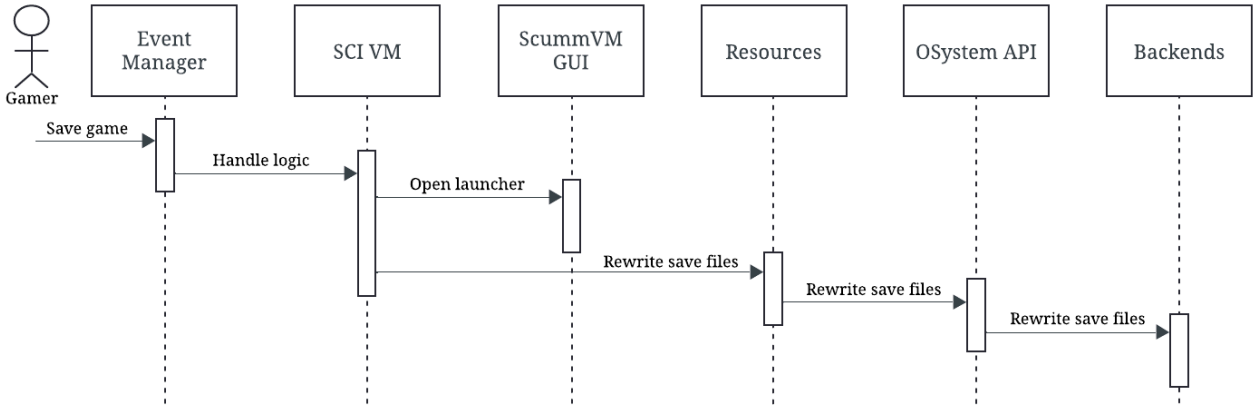

Figure 4. Sequence diagram for use case of saving a game

The third use case involves the user saving their game. Although games are autosaved every five minutes, the user may choose to make a save after they are finished playing.

To save a game, the Event Manager subsystem detects when the user inputs the shortcut programmed to make a save, which is either Ctrl + F5 or Ctrl + fn + F5 on macOS. The input is forwarded to the SCI VM subsystem, which handles the logic of what to do. First, the ScummVM GUI is opened, since saves are made in the global ScummVM launcher. Then, the Resources subsystem is called to manage the save files, which contacts the OSystem API subsystem and then the Backends subsystem in order to rewrite the save files in the user's OS file system.

Conclusions

In conclusion, ScummVM is a convenient and useful open-source program for making many older DOS-based games accessible on a variety of modern OS systems. It achieves this function by following the interpreter architectural style with elements of the layered architectural style. This combination of architectural styles increases flexibility in implementation as additional

subsystems can be added to the overall architecture without affecting existing subsystems substantially. It also enables portability of the original game to different devices.

As shown in our research and analysis, ScummVM achieves its functionality through rewriting the game files of the original game. Our team also determined that the SCI engine contains many important components for the system of ScummVM, which includes the interpreter, Graphics, Sound, and Event Manager components, and that SCI VM acts like a central logic handler by managing the tasks of all of these components. Identifying the relationships between components greatly helped us formulate our proposed conceptual architecture for ScummVM and SCI engine.

Lessons Learned

While formulating this report, our group ran into some issues when designing the conceptual architecture ScummVM and the SCI engine. We found it difficult to nail down a conceptual architecture using the provided ScummVM documentation alone, since common game architecture subsystems such as graphics, audio, and game logic seemed to be missing. Seeking out additional documentation on the SCI engine aided us in rounding out our subsystems since it included Graphics, Sound, an Event Manager, a Resources subsystem that acted as a database, and a SCI VM subsystem that acted as the central logic handler. However, we were still unsure of the dependencies between all the subsystems, so we turned to the source code documented in the ScummVM GitHub repository in order to figure out which subsystems depend on each other. Finally, after thorough group discussion, we were able to settle on the subsystems to include in our conceptual architecture, the dependencies between the subsystems, and which layers to put each subsystem in.

Overall, our group worked well together, and we frequently communicated in our group chat and distributed the work according to each member's strengths. We found that meeting every week at the scheduled tutorial time on Mondays was very helpful, since we didn't need to go out of our way to figure out a separate time and place to meet. Additionally, there was a TA at every tutorial, and on one occasion our group's assigned TA was at the tutorial answering our questions. What ultimately led to the success of our group was discussing deadlines right at the beginning of the project so that it was clear to every member what they needed to work on and when. We set soft deadlines for the report to be finished before the actual due date so that we would have enough time to make the presentation and polish our work.

An area we could improve on is discussing the architecture ahead of time. While discussing the deadlines at the beginning of the project, we also assigned specific sections of the report for certain members to work on. This led to separate people working on the diagrams, the introduction and conclusion, or the main architecture, when in reality the conceptual architecture should have been discussed and designed by each member before writing the report. Figuring out the architecture style, subsystems, and use cases as a full group before writing the report would have lessened confusion and knowledge gaps. Additionally, for this stage of our project, we separated writers from presenters so that writers would only work on the report and presenters would only work on the presentation. Going forward, each member should help with both deliverables so that everyone is on the same page, since there were a couple of occasions where the presenters needed to ask the report writers for clarification. Implementing these

improvements will help strengthen our understanding of the course material and lead to a more cohesive group project.

Data Dictionary

Conceptual Architecture: A theoretical high-level overview of the system's structure, illustrating how various components and modules are organized and interact with one another, without detailing the actual implementation.

Control Flow: The sequence of execution paths taken through the software, dictating how different modules and functions are activated during gameplay.

Game Engine: The underlying software platform that provides the necessary tools and functionalities for creating and running video games, including rendering graphics, managing physics, and processing input.

Interpreter Style Architecture: A software design pattern where code is interpreted and executed directly, line by line, at runtime, allowing for flexible updates and modifications without the need for recompilation.

Layered Style Architecture: A structural framework that organizes system components into distinct layers, where each layer has specific responsibilities and interacts with adjacent layers, enhancing modularity and ease of maintenance.

Open-source: Software that is made publicly available for anyone to use, modify, and distribute, typically fostering collaboration and community-driven development.

Subsystem: An independent component within the overall architecture that encapsulates specific functionalities, enabling collaboration with other subsystems to achieve the system's goals.

Virtual Machine: A software emulation of hardware that enables the execution of programs in an isolated environment, providing a platform for running games across different systems without modification.

Naming Conventions

Application Programming Interface (API): An Application Programming Interface that allows different software applications to communicate and interact with one another by defining a set of rules and protocols.

Graphical User Interface (GUI): A Graphical User Interface that enables users to interact with a system through visual elements such as windows, icons, and buttons, rather than text-based commands.

Operating System (OS): A system software that manages hardware and software resources on a computer, providing services for computer programs.

Script Creation Utility for Maniac Mansion Virtual Machine (ScummVM): A software that allows users to run classic point-and-click adventure games on modern hardware by emulating the original game engines.

Sierra Creative Interpreter (SCI): A game engine developed by Sierra On-Line that powers the scripting and execution of adventure games, supporting graphics, sound, and user input through an interpreter-based architecture, allowing for complex narrative-driven gameplay.

Virtual Machine (VM): A Virtual Machine, an emulation of a computer system that allows software to run in an isolated environment, simulating hardware for program execution.

References

[1] A. Staff, "Maniac Tentacle Mindbenders: How ScummVM's unpaid coders kept adventure gaming alive," *Ars Technica*, Jan. 17, 2012. https://arstechnica.com/gaming/2012/01/maniac-tentacle-mindbenders-of-atlantis-how-scummvm-kept-adventure-gaming-alive/ (accessed Oct. 07, 2024).
[2] C. H, "Welcome to ScummVM! — ScummVM Documentation documentation," *Scummvm.org*, 2020. https://docs.scummvm.org/en/v2.8.0/index.html (accessed Oct. 07, 2024).
[3] "SCI," *ScummVM :: Wiki*, 2023. https://wiki.scummvm.org/index.php/SCI (accessed Oct. 07, 2024).
[4] P. Fortier, "An IDE for Sierra's SCI engine — SCICompanion 3.0 documentation," *Scicompanion.com*, 2015. https://scicompanion.com/Documentation/ (accessed Oct. 07, 2024).
[5] C. H, "Handling game files — ScummVM Documentation documentation," *Scummvm.org*, 2020. https://docs.scummvm.org/en/v2.8.0/use_scummvm/game_files.html (accessed Oct. 07, 2024).
[6] C. Harris, "Microservices vs. monolithic architecture," *Atlassian,* https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith (accessed Oct. 05, 2024).
[7] C. H, "Release notes — ScummVM Documentation documentation," *Scummvm.org*, 2023. https://docs.scummvm.org/en/v2.8.0/help/release.html (accessed Oct. 07, 2024).
[8] "Developer Central," *ScummVM :: Wiki*, 2020. https://wiki.scummvm.org/index.php?title=Developer_Central (accessed Oct. 07, 2024).
[9] C. H, "Understanding the interface — ScummVM Documentation documentation," *Scummvm.org*, 2020. https://docs.scummvm.org/en/latest/use_scummvm/the_launcher.html (accessed Oct. 07, 2024).