CISC 322/326

Assignment 2

Concrete Architecture of ScummVM and SCI Engine

Monday, November 18, 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

In this report, we recover the concrete architecture of ScummVM and SCI Engine. We use the Understand tool and GitHub to analyze dependencies between subsystems and uncover the rationale behind them. The concrete architecture, like the conceptual architecture, follows a hybrid layered and interpreter style, with the primary structure consisting of four layers. We find there are many more dependencies between subsystems than we initially expected, leading to a more complicated and interconnected architecture. We analyze the rationale for these divergences, and perform a reflexion analysis to thoroughly compare our conceptual architecture with the concrete one. With our understanding of the system's control flow and concrete architecture, we analyze some use cases and perform a detailed analysis of the Sound subsystem's conceptual and concrete architecture. Finally, we reflect on the lessons we learned along the way.

Introduction

ScummVM's development involves many developers contributing independently to various subsystems, with limited standardization. The project being open-source allows for transparency and for developers to easily keep the software up-to-date amongst other perks. However, it also leads to a lot of complications; individual developers have few barriers stopping them from creating dependencies, many of which may be unjustified and add unnecessary complexity to the system [1]. All in all, this leads to a system where every major subsystem depends on nearly every other major subsystem, with little unified documentation explaining why this is. As such, understanding these connections is important for making necessary changes to the system.

With a well-researched concrete architecture, we can determine why a dependency exists, what the effects of adding or removing a certain dependency would be, or how a system might be simplified or extended. This knowledge is especially relevant because many engines, notably SCI Engine, sit atop of ScummVM. Thus, errors in ScummVM's system can have cascading effects on many other systems. A well-functioning code base for ScummVM is critical to ensure engines can have an easier time building atop it. Similarly for SCI Engine, understanding its concrete architecture is crucial for being able to rationalize and build on the system.

To ensure that our work is accurate, it is crucial to compare our conceptual and concrete architectures and analyze where differences (absences and divergences) exist. We must also analyze and attempt to justify these differences to determine if they are vital enough to be included in the conceptual architecture, or if they can be safely excluded. This process involves performing a reflexion analysis to critically analyze our architectures. Using a concrete architecture and a firm understanding of the source code, analyzing specific use cases can be done more accurately than with just a conceptual architecture. This involves tracing the source code to infer the control flow of the system.

With motivation in hand, we can explore how we arrived at our concrete architecture, the relationships in the concrete architecture itself, a reflexion analysis comparing it with the conceptual architecture, our chosen use cases, and the lessons we learned.

Derivation Process

In order to create our conceptual architecture, we analyzed the ScummVM source code using SciTools Understand. Before actually using Understand to map files to architectural subsystems, each group member took a look at the source code for SCI Engine and ScummVM from the ScummVM Github repository to get an idea of how they were implemented [2]. Once everyone had taken a look at the source code we held a meeting where we used Understand and started mapping the source code to our concrete architecture. We started with mapping the obvious files to their appropriate subsystems from our conceptual architecture, then we moved onto the files that we were unsure about. Some of these files were surprising as they included things that we did not expect from our conceptual architecture. For example, we found a collection of engine files which contained utility code for game engines. We initially considered creating a new subsystem in the concrete architecture—an Engine subsystem—but after more discussion we decided to place these files in the Common subsystem as it matched that subsystem's role as a utility code provider.

After the concrete architecture was done, we found many divergences between our conceptual and concrete architecture. In order to investigate these divergences we used Understand to view what files were causing these divergences. For each divergence we looked at the lines in the source code that caused them, and used the blame margins in Understand to view the commits that introduced the unexpected dependencies. For some of them we were lucky and the commit message gave a clear indication as to why the dependency was introduced. On the other hand, for some, the commit message was not useful and we instead had to read and analyze the code in order to understand what the purpose of the dependency was. This process was slow and difficult and we were faced with some tough decisions during it; for example, we considered changing our architecture to be object-oriented instead of layered due to the seemingly chaotic structure of dependencies between subsystems that we did not expect. However, the object-oriented style is more about encapsulating and protecting information within subsystems, and this was not the case here as a lot of calls seemed to be bypassing the more structured and longer way to access the data they needed, like directly calling the backends instead of using the OSystem API. Ultimately we decided to keep our layered architecture as it made the most sense, especially since several divergences were caused by only a few code level dependencies.

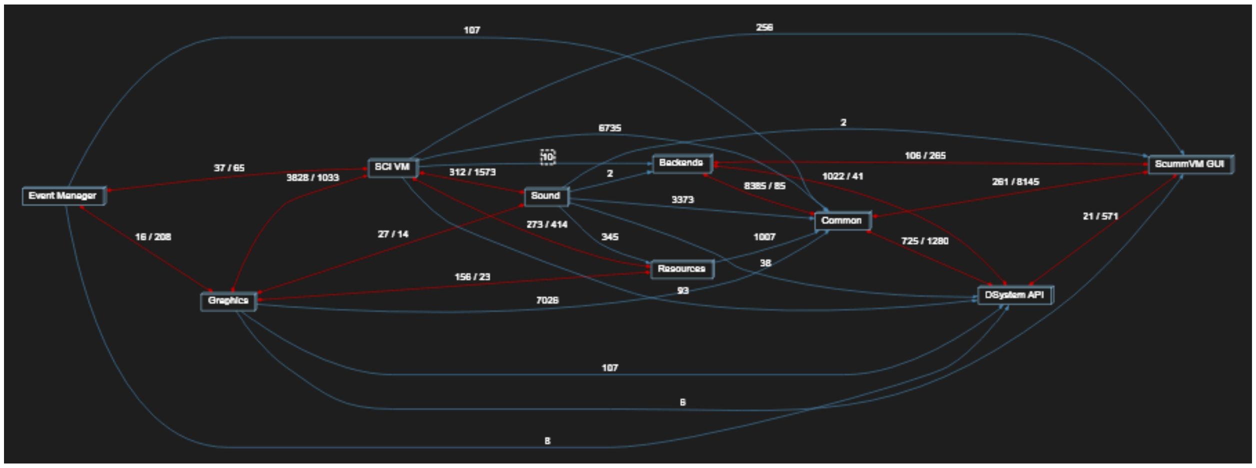

Figure 1: Understand architecture

Conceptual Architecture

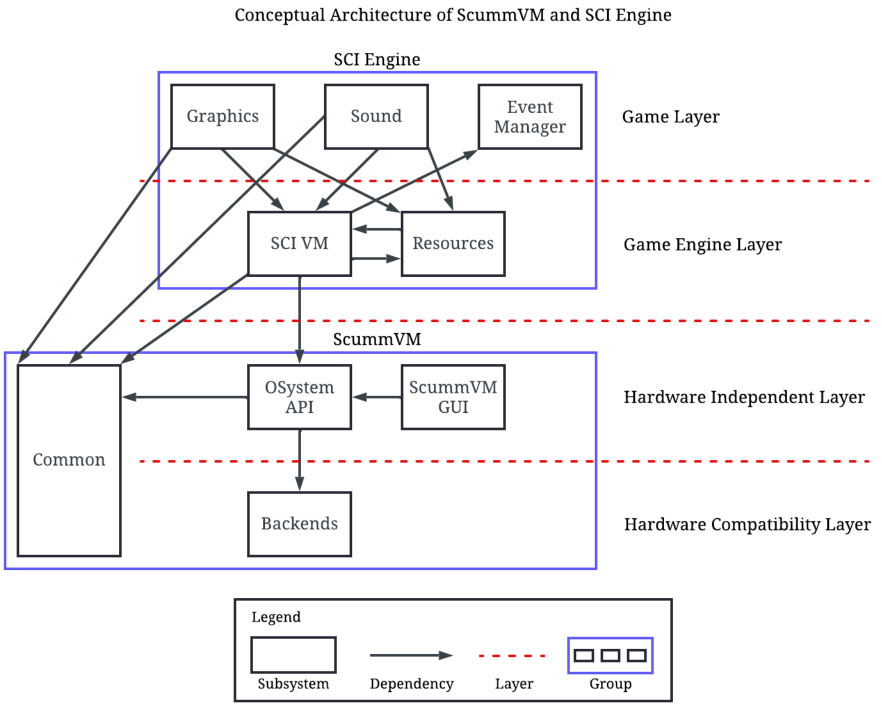

Figure 2: Conceptual architecture of ScummVM and SCI Engine

Our conceptual architecture remains unchanged from our A1 report.

Concrete Architecture

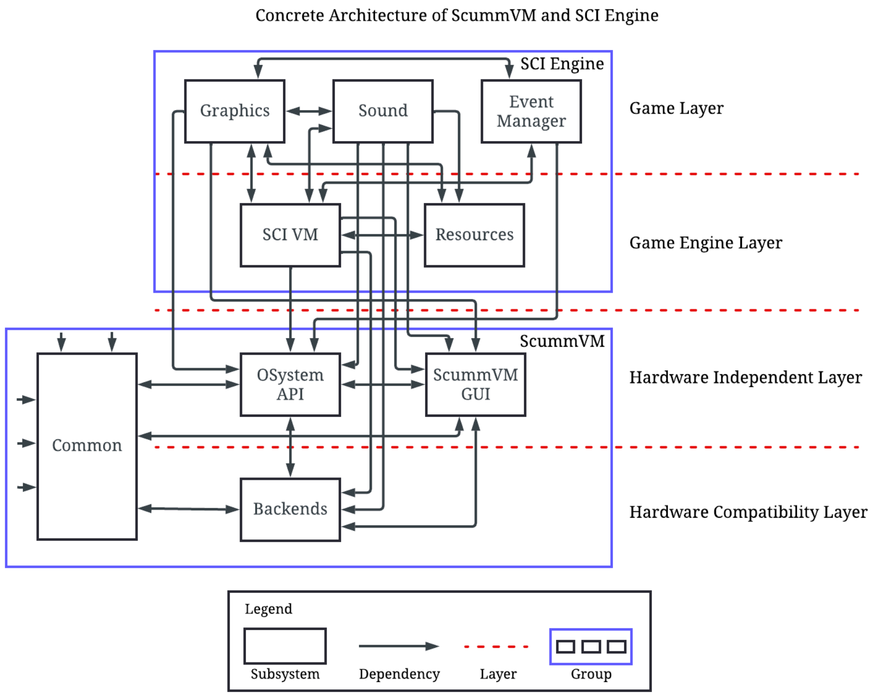

Figure 3: Proposed concrete architecture of ScummVM and SCI Engine

The architectural styles—interpreter and layered architectural style—remain the same between the conceptual and concrete architecture. Our team had discovered that some dependencies between subsystems bypass some of the layers in our concrete architecture; however, we believe the layered architectural style still remains accurate as often this bypassing was simply for the sake of speed and efficiency, and other dependencies still adhere to a layered hierarchy. The concrete architecture also contains the same modules as the conceptual architecture, though some new dependencies and interactions between these subsystems were introduced. These divergences from our proposed conceptual architecture are explained in-depth in the reflexion analysis portion of our report. We also gained some new insight into the function of each module upon closely examining the GitHub repository for the ScummVM source code and via use of the Understand tool. These insights are also explained in the following subsystems' descriptions.

OSystem API

The OSystem API acts as an intermediary component between the Backends and the rest of the system by providing an interface for communicating with the Backends, such that other components like the GUI and game engines do not need to access the OS directly. It also provides functions that define what an engine or a game can do within ScummVM like handling user input events, drawing graphics on a video surface, or controlling audio playback. We altered this subsystem slightly from our A1 report, as it now includes the entry point for the system on startup and the main loop of the program. Nearly all data flows through OSystem API, which is why all other subsystems except Resources depend on it. As well, OSystem API depends on Backends, Common, and ScummVM GUI.

Backends

The Backends consists of several coupled smaller services and implements the OSystem API for various platforms as described in our A1 report. It has a bidirectional dependency with OSystem API, Common, and ScummVM GUI. Additionally, SCI VM and Sound depend on it.

Common

The Common component refers to "common code" and it provides various utility classes to the rest of the system as described in our A1 report. For example, Common contains useful classes that support audio, game engines, graphics and images, arithmetic, and video functionalities. All subsystems depend on Common given that the utility classes provided by it are used by every other subsystem. Common depends on OSystem API, Backends, and ScummVM GUI.

ScummVM GUI

The ScummVM Graphical User Interface (GUI) provides the elements for the game launcher and options dialog, which the user can interact with. Our team had discovered that the global main menu is accessible while the user is playing a game instead of only prior to or after a game is played. This discovery is why, contrary to our original conceptual architecture, more components depend on the ScummVM GUI than we initially thought such as Graphics, Sound, and SCI VM. The ScummVM GUI has two-way dependency with OSystem API, Backends, and Common.

Graphics

The Graphics component is part of the SCI engine. It contains graphics and video functions for drawing graphics to the screen as a user plays a SCI Engine game. The Graphics component has a two-way dependency to Sound, Event Manager, Resources, and SCI VM. We also discovered that the Graphics component depends on OSystem API and ScummVM GUI.

Sound

The Sound component is part of the SCI Engine. It contains sound functions that control the volume of the audio and when the audio plays or stops for when a user is playing a SCI Engine game. The Sound component has a two-way dependency to Graphics and SCI VM, and a one-way dependency to Resources and Common. The Sound component also directly accesses OSystem API, ScummVM GUI, and Backends for speed and efficiency.

Event Manager

The Event Manager is part of the SCI Engine. It handles user input from their keyboard, mouse, or joystick for when the user is playing a SCI Engine game. The Event Manager has a two-way dependency with Graphics and SCI VM. We also discovered that it depends on OSystem API and Common.

Resources

The Resources component is part of the SCI Engine. It stores and retrieves files for the SCI Engine. Resources has a two-way dependency with SCI VM and a one-way dependency from Sound to Resources as expected from our conceptual architecture. Additionally, we found that the one-way dependency from Graphics to Resources that we proposed in our conceptual architecture is actually two-way, and that Resources depends on Common.

SCI VM

The SCI VM (Sierra Creative Interpreter Virtual Machine) component is an interpreter that enables SCI Engine games to be playable as described in our A1 report. It contains all the other necessary classes for the SCI Engine such as extended functions for graphics, sound, and input, reading and writing data from the hard disk, as well as handling logic and arithmetic. The SCI VM component has a two-way dependency with the Graphics, Sound, Event Manager, and Resources component, and it depends on OSystem API as expected. We discovered it also depends on ScummVM GUI and is able to bypass OSystem API to access Backends directly.

Reflexion Analysis

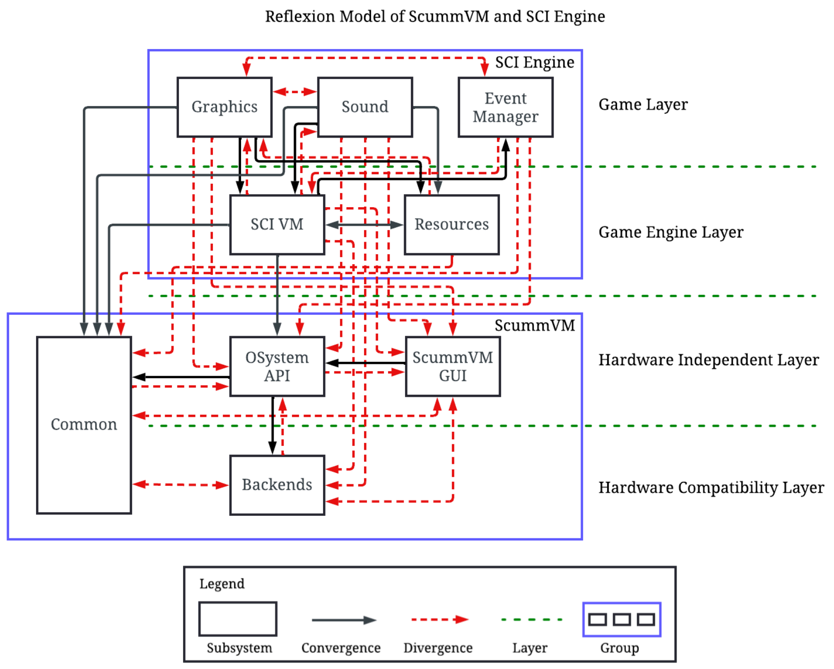

Figure 4: Reflexion model of ScummVM and SCI Engine

The above diagram depicts our reflexion model. Our team discovered no absences, as all the dependencies present in our conceptual architecture are also present in the concrete architecture, which therefore makes them convergences. We also discovered several divergences that are explained below in the following divergence descriptions.

OSystem API

As previously mentioned, nearly all data flows through OSystem API since it contains the main loop and functions that define what a game or engine can do. However, we expected the modules in the Game Engine and Game Layer to go through the interpreter SCI VM since SCI VM depends on OSystem API, but it turns out that these subsystems can bypass SCI VM and access functions from OSystem API directly for the purpose of efficiency and minimizing performance issues.

There are 4 divergences we will discuss in this section which involve another subsystems depending on OSystem API: **Backends → OSystem API**, **Graphics → OSystem API**, **Sound → OSystem API**, and **Event Manager → OSystem API**. Backends turns out to depend on or implement functions provided by the OSystemAPI; as examples, virtual-mouse.cpp from Backends uses getOverlayHeight, getOverlayWidth, getWidth, and getHeight from the OSystem API for adjusting cursor speed according to virtual screen resolution, and remap-widget.cpp uses getEventManager for accessing the event object. The Graphics component depends on the OSystem API for color functions, and information of the necessary size for game graphics. The dependency from Sound to OSystem API was added to get the audio manager from ScummVM, the sound mixer for drivers, and timing functions, like tracking time since the application has started. The dependency from Event Manager to OSystem API was added so that Event Manager can call the event manager from OSystem API, which is used to detect the user input (like keyboard and mouse events).

Backends

There are 2 unexpected dependencies to Backends: **Sound → Backends** and **SCI VM → Backends**. Based on our concrete architecture, we expected these subsystems to access OSystem API since OSystem API depends on Backends, but we discovered that SCI VM and Sound can bypass OSystem API to directly access Backends for speed.

The dependency from Sound to Backends was added so that Sound could directly access a value in Backends to check the duration of audio samples. Although it uses OSystem API calls to get a struct from Backends, it directly accesses the value in Backends. SCI VM directly calls Backends to help create savestates (save games). This is because OSystem API does not provide a mechanism to write saves to the filesystem like we expected in our conceptual architecture; instead, it provides a function that will get a file that is open to have a save written to. The rest has to be implemented by the engine authors with help from other subsystems like Common. Here, Backends is used to finalize saves as well as detect if there was an error writing the save to a file.

Common

There are several divergences involving Common but most of them exist for similar reasons. The divergences involving other subsystems depending on Common, which are **Backends ↔ Common**, **ScummVM GUI ↔ Common**, **Event Manager → Common**, and **Resources → Common**, are all a result of our group underestimating how much the utility code in Common would be needed by other subsystems. All these divergences exist because the subsystems need some sort of helper function from Common.

There are 3 other divergences which are ones involving subsystems that Common relies on, these are: **Common ↔ OSystem API**, and **Common ↔ Backends** and **Common ↔ ScummVM GUI**. Common depends on Backends because it needs some helper functions to perform hardware independent actions, like accessing the computer's file system and keymapping. Common depends on OSystem API because it needs to use the API to call many functions to implement the common code; for example, mutex.h includes system.h (OSystem API) to create another API for managing mutex locks. Finally, Common depends on ScummVM GUI primarily to display warning and error messages in ScummVM and to get data from the GUI like getting the name of a recording.

ScummVM GUI

There are more subsystems that access ScummVM GUI than we originally thought, due to the global main menu being accessible while playing a game, instead of the launcher being used only before or after playing a game.

There are 4 new divergences involving a subsystem depending on ScummVM GUI, mainly to have the ability to display messages on screen using the GUI. The **OSystem API → ScummVM GUI** dependency exists to get GUI related information like the available themes and launchers so this information can be displayed if the user requests it. The **Sound → ScummVM GUI** dependency is for displaying an error message when sound files or audio drivers are missing, while the **Graphics → ScummVM GUI** dependency exists to display message boxes to the screen. The **SCI VM → ScummVM GUI** dependency is not only for SCI VM to be able to display messages, but also to use the ScummVM GUI debugger for SCI's console.

There is 1 new two-way dependency, which is **Backends ↔ ScummVM GUI**. Backends depends on ScummVM GUI for user input so that the user can perform actions such as selecting a target folder to be downloaded. ScummVM GUI depends on Backends to use functions for graphics, audio, and running dialogue.

SCI VM

Many of the one-way dependencies to SCI VM in our conceptual architecture turned out to be two way dependencies. This is so that the modules in the Game Layer can easily communicate with SCI VM since it is the main logic handler of the SCI Engine. There are 2 new dependencies of SCI VM to another subsystem, **SCI VM → Graphics** and **SCI VM → Sound**, and 1 new dependency of another subsystem to SCI VM, **Event Manager → SCI VM**.

SCI VM calls functions in Graphics in order to deal with the creation and destruction of graphical elements in SCI. Although Graphics is responsible for drawing objects, SCI VM handles the management of those objects, like calling a function from Graphics to create a certain type of object or calling a destructor to remove the object and free up memory. SCI VM uses Sound in order to perform certain initialization and management tasks. For example, when the run() function is called in SCI VM, an instance of SoundCommandParser is created in order to initialize the sound system. Finally, the dependency from Event Manager to SCI VM was added to handle mouse positioning in certain graphics formats by getting the version of SCI being run and applying changes accordingly. Event Manager also limits how often SCI VM checks if it should quit by sending a signal to SCI VM telling the engine to quit if needed.

Graphics

The last 3 unexpected dependencies are between subsystems in the SCI Engine group, which were added so that the subsystems can all communicate with each other to update the visuals, audio, and save files accordingly while the user plays the game. The **Sound ↔ Graphics** dependency coordinates visual and audio elements to maintain synchronization. When a player's action triggers a visual event (like a character's movement or an item pickup), the Graphics component cues the Sound component to play the matching audio. This two-way interaction allows visuals and sound to adjust in tandem, providing immersive, cohesive feedback that enriches the game's multimedia experience. The **Event Manager ↔ Graphics** dependency exists to handle real-time user interactions that affect game visuals. The Event Manager processes inputs (like keyboard and mouse actions) and relays these commands to the Graphics component, which updates the display accordingly. This feedback loop ensures that visual responses to player inputs are smooth and immediate, enhancing the gameplay experience. Finally, the **Resources → Graphics** dependency was added since Resources uses an instance of ViewType from Graphics in order to determine the type of a specific graphical resource; for example, it determines whether a graphic resource is an EGA or VGA resource.

Sound Subsystem

For our analysis of a ScummVM and SCI Engine subsystem, we chose to investigate the Sound subsystem. This subsystem is used to control the sound and music when playing a SCI game.

Although Common contains a folder for sound utility classes, we are specifically focusing on the Sound module, part of SCI Engine, that is implemented specifically for SCI games.

Conceptual and Concrete Architecture

We determined the conceptual architecture of the Sound subsystem from documentation, then used the Understand tool and the ScummVM source code to get its concrete architecture and to perform reflexion analysis on our architecture [3]. This subsystem follows the Object-Oriented architectural style due to the encapsulation of the components. We determined that the concrete and conceptual architectures both have the same modules and architectural style, but the concrete architecture has 1 unexpected dependency.

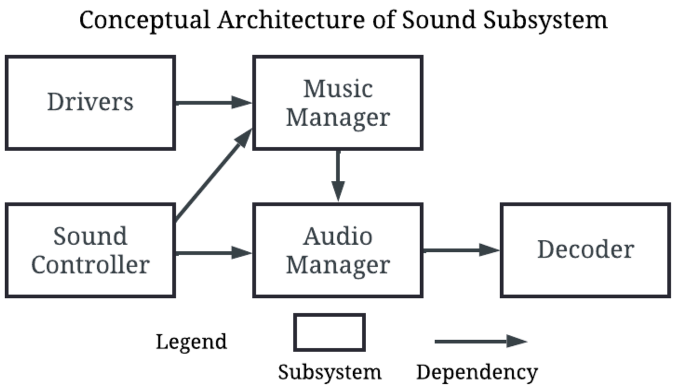

Figure 5: Conceptual architecture of Sound subsystem

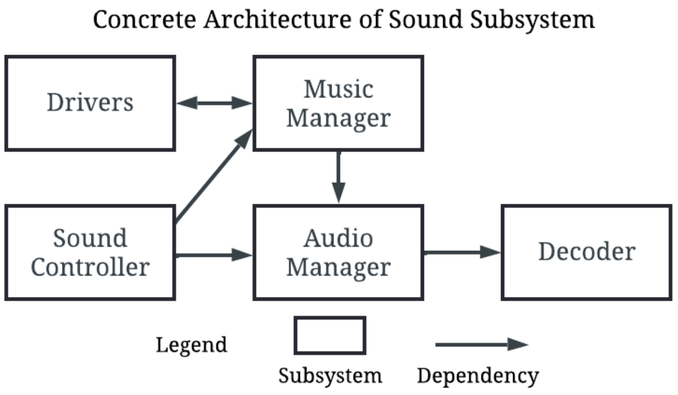

Figure 6: Concrete architecture of Sound subsystem

Drivers

The Drivers subsystem contains drivers to handle different audio formats. It also contains a MIDI Parser in order to parse MIDI events from MIDI files and process them into playable audio.

Sound Controller

The Sound Controller subsystem provides the main entry point into the Sound subsystem. It provides several functions for manipulating the organizational aspects of the Sound system, such as supporting audio-video synchronization, controlling the master volume, and adding or removing tracks from audio playlists.

Music Manager

The Music Manager subsystem is used to implement functions for manipulating tracks in the audio player. It provides functions that perform tasks like adding or removing tracks from a playlist, and pausing or resume tracks. These functions are then wrapped by the Sound Controller for external use.

Audio Manager

The Audio Manager provides master control over all audio in general. It is used to set up SCI audio channels if SCI 32 is enabled, and it provides functions that pause or resume audio, and start or stop the playing of audio. Unlike Music Manager, functions from the Audio Manager are not wrapped by another system and are directly called from outside the Sound subsystem.

Decoder

The Decoder subsystem contains decoders to decompress certain audio file types. It adds additional decoders for other file types that are not included in the default ScummVM decoders.

Reflexion Analysis

After performing reflexion analysis by comparing our conceptual architecture of the Sound subsystem with the concrete architecture, we found no absences but discovered one divergence: **Music Manager → Drivers.** Although Drivers contains sound drivers to handle different types of audio, we found that the sound drivers are actually initialized in Music Manager instead of their original files, which explains the new dependency.

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 two use cases to demonstrate the use of and the interactions between the major subsystems of our proposed concrete architecture.

Use Case #1: Adding a Game

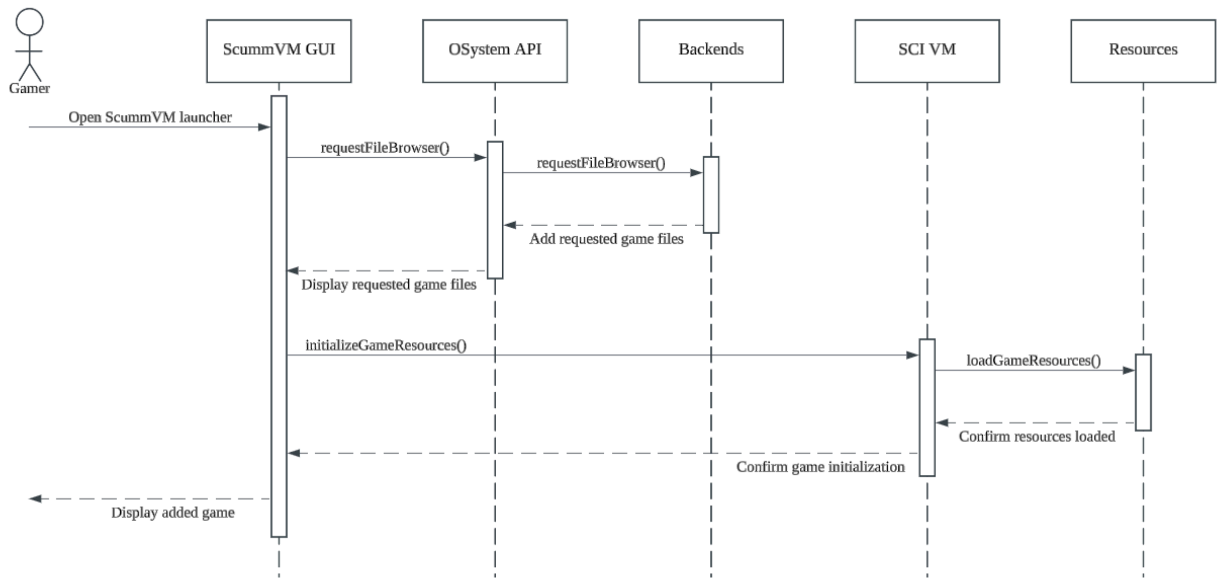

Figure 7: Sequence diagram for use case of adding a game

This use case involves adding a game to the ScummVM launcher, allowing the user to play a selected game. This interaction is essential since games must be added to ScummVM before they can be accessed through the launcher.

The process of adding a game starts when the user opens the ScummVM launcher via the ScummVM GUI. The user requests to add a game, triggering the ScummVM GUI to call requestFileBrowser() through the OSystem API. This call is forwarded to Backends, which interacts with the OS to open the file browser. After the user selects the game files, they are returned to the ScummVM GUI for display. Next, the GUI calls initializeGameResources() to start loading the game, delegating this task to the SCI VM. The SCI VM requests loadGameResources() from the Resources subsystem, which loads and confirms that all necessary resources are in place. Once confirmed, the game is successfully initialized, and the ScummVM GUI updates to display the newly added game.

Use Case #2: Saving a Game

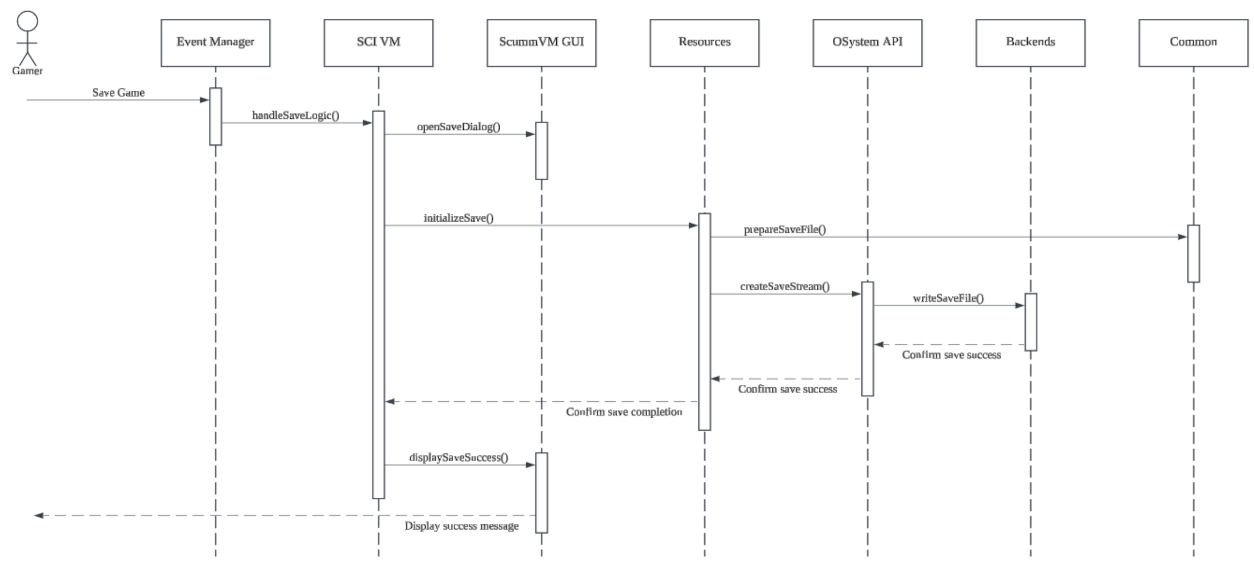

Figure 8: Sequence diagram for use case of saving a game

This 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.

When the user initiates a save request, either through a menu selection or a keyboard shortcut (e.g., Ctrl + F5 or Ctrl + fn + F5 on macOS), the Event Manager subsystem detects the input and forwards it to the SCI VM subsystem. SCI VM processes this input, handling the game's logic and initiating the save operation by calling openSaveDialog() to display the save dialog in the ScummVM GUI. Once the user selects save options, SCI VM proceeds with initializeSave(), prompting the Resources subsystem to handle the save file's data. Resources gathers all necessary information — such as the player's location, inventory, and game state — by calling prepareSaveFile(), which organizes the required assets and states for the save file. To write the data, Resources calls createSaveStream() through the OSystem API, which provides platform-independent file operations. The OSystem API forwards this request to the Backends subsystem, which performs writeSaveFile() to actually write the data to the user's file system. After the save operation completes, a confirmation is sent back from Backends to OSystem API and then to Resources, indicating that the save was successful. Resources relays this confirmation to SCI VM, which then triggers displaySaveSuccess() in the ScummVM GUI to inform the user that the game has been successfully saved. The GUI displays a success message, providing clear feedback that the save was completed.

Concurrency

Concurrency was found in Backends, Sound, Graphics, SCI VM, ScummVM GUI, and Common. Examples of concurrency used include threads, mutexes and semaphores. Audio-visual synchronization is one example of the applications of concurrency used.

Lessons Learned and Limitations

When investigating the dependencies in Understand, our group ran into some issues trying to understand the reasoning for the unexpected dependencies. Since the commit messages were vague and limited, we had to infer what the files do. Furthermore, during the start of the investigation, the number of unexpected dependencies far exceeded our expectations, leading us to doubt if we were on the right track.

Overall, our group worked well together due to our communication through Discord and our weekly meetings. We also applied the lessons we learnt from the last report by assigning everyone part of the report. Task distribution improved as a result, increasing efficiency.

Staying up to date with the course material is one of the lessons we took away. Keeping up with the course material would have given us more time to view dependencies and work on the report. Time was spent catching up and looking at some of the source code in the weeks leading up to the test. Only after test two, did we finally get to work on the report.

Conclusion

We explained how we deduced a concrete architecture for ScummVM and SCI Engine, using the Understand tool to not only see the high-level connections between subsystems, but also analyze the low-level file structure and source code. This allowed us to infer the reasoning behind various dependencies and to get a better understanding of the structure of the system.

With this research, we presented our concrete architecture. It was similar to the conceptual architecture in that it followed a hybrid layered interpreter style, with the four layers themselves being the same. However, there were many differences, namely that the concrete architecture had many more dependencies between subsystems, many of which served as exceptions to the layered structure. As such, the concrete architecture is much more complicated and less neatly put together as compared to the conceptual architecture. We explored the details of the similarities and differences between the architectures—namely the many divergences—in our reflexion analysis.

Using our concrete architecture and source code knowledge, we analyzed the Sound subsystem and explored use cases for ScummVM and SCI in a way that we could not do before with just our conceptual architecture in hand. We concluded by reflecting on the lessons we learned along the way in the creation of the architecture and the report.

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.
Concrete architecture: The actual, detailed structure of a software system, showing how the system is implemented.
Dependencies: A relation between components or resources, where one component uses functions or resources from the other.
Game engine: The underlying software platform that provides the necessary tools and functionalities for creating and running video games.
Interpreter architecture style: 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 architecture style: 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.
Object-oriented architecture style: A software design pattern where data is encapsulated in an abstract data type to identify and protect bodies of information.
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.
Understand: A software tool used to investigate the structure and dependencies of code.

Naming Conventions

API: Application Programming Interface
GUI: Graphical User Interface
ScummVM: Script Creation Utility for Maniac Mansion Virtual Machine
SCI: Sierra Creative Interpreter
VM: Virtual Machine

References

[1] "Developer Central," *ScummVM :: Wiki*, 2020. https://wiki.scummvm.org/index.php?title=Developer_Central (accessed Nov. 17, 2024).
[2] scummvm, "GitHub - scummvm/scummvm: ScummVM main repository," *GitHub*, Mar. 31, 2024. https://github.com/scummvm/scummvm (accessed Nov. 17, 2024).
[3] "SCI/Specifications/Sound," *ScummVM :: Wiki*, 2024. https://wiki.scummvm.org/index.php?title=SCI/Specifications/Sound (accessed Nov. 17, 2024).