CISC 322
A2

SCUMMVM Concrete Architecture

Chamos:

Anes Numanovic 21an88@queensu.ca 20353836
Vasilije Petrovic 21vmp8@queensu.ca 20357533
Jacob Cruz 21jtc9@queensu.ca 20349571
Oliver Ren 21opr1@queensu.ca 20336697
Mattias Khan 22mhk@queensu.ca 20359237
Kevin Jiang 21kj31@queensu.ca 20353263

Table Of Contents

Abstract

In this report, we examine the concrete architecture of ScummVM, an open source software that can run old adventure games. As well as a discussion of the software's concrete architecture, we also did a deep dive on the OSystem API component.

Our analysis shows that the system still indeed follows an interpreter style like we predicted from the conceptual architecture. Despite our conceptual architecture being relatively similar, we decided to get rid of four components for various reasons: the user, the outputs, the program being executed and the current state of the engine. No new subsystems were added to architecture, but some interactions were changed. We also took a look at the OSystem API subsystem, which contain six key subsystems and some miscellaneous components

To discover the discrepancies between our conceptual and concrete model, we used the software reflexion framework, and laid out what we discovered in this report. Additionally, we include our derivation process and go over the two use cases from the last report to see what changed. Finally, we summarize with our conclusions as well as limitations and lessons learned.

Introduction

To recap, ScummVM is an open source software that allows users to play old point-and-click adventure games. It was originally designed to play LucasArts adventure games that used its namesake SCUMM system, which stands for Script Creation Utility for Maniac Mansion. ScummVM reinterprets the game engine used in these old games for systems including modern computers and game systems, instead of simply emulating these games. Since the software's launch in 2001, it has helped preserve many old titles that would've been lost to the sands of time.

Building on top of our previous report on ScummVM's conceptual architecture, this report dives deep into its concrete architecture. The major differences between our conceptual and concrete architecture was the exclusion of four components. The user and output component were removed due to being external elements, and the program being executed and current state of engine components were removed due to being part of the ScummVM engine subsystem. On the other hand, no subsystems were added and the common code, GUI, backends, ScummVM Engine and OSystem API subsystems were kept.

We also did a deep dive on the OSystem API subsystem. This subsystem has 6 main submodules, those being for core components, input, graphics, audio, timer, and file system and storage, as well as miscellaneous components. We also applied the software reflexion framework again to determine the differences between the conceptual and concrete architecture of the OSystem API.

Additionally, we refined our use-case diagrams on how ScummVM loads games, and how it plays games. Finally, we note the limitations we encountered and major lessons we learned.

Derivation Process

We derived our concrete architecture from a large variety of sources. We used the github repository, the ScummVM wiki documentation, and common game engine architectural knowledge to derive a list of ScummVM subsystems. Then we needed to analyze ScummVM's raw file structure to

complete our concrete architecture. We downloaded the source code of ScummVM, and then opened it and analyzed it using the program SciTools Understand. This program displays all the files of the source code, and all the dependencies they have with other files. We amended our list of subsystems based off of the files and directories shown by Understand. We then needed to find the dependencies between subsystems of this system. To do this we first mapped every directory/file to its appropriate subsystem. From then we converted every source code dependency of the system to an architectural dependency between the subsystems the directory/files were mapped to, and added these dependencies to our concrete architecture. Singular dependencies shown in the concrete architecture can represent numerous code level dependencies. After this process we have completed our concrete architecture.

High-Level Architecture

After deriving the concrete architecture using the Understand tool, we can conclude that the architecture style is still interpreter due to its flexible nature, that being it allows you to play games on several different platforms. This is further reinforced by components and the relationships between them. Game-specific engines are able to interpret and execute game scripts by translating commands into system actions in real-time. The OSystem API and platform-specific backends provide cross-platform functionality, while the common code supplies the essential resources for the game interpretation. The GUI ties it all together by acting as a control interface, allowing the user to run the software with ease.

In terms of the subsystems, some components did not map into the concrete architecture due to the broadness and interaction with the software. For example, the subsystems missing from the conceptual architecture were: The user, the outputs, the program being executed, and the current state of the engine. The user and outputs were removed since they are more of an external element of the software; They are not a true element of the software itself. The program being executed and the current state of the engine were removed since their components fit better within the ScummVM engine subsystem.

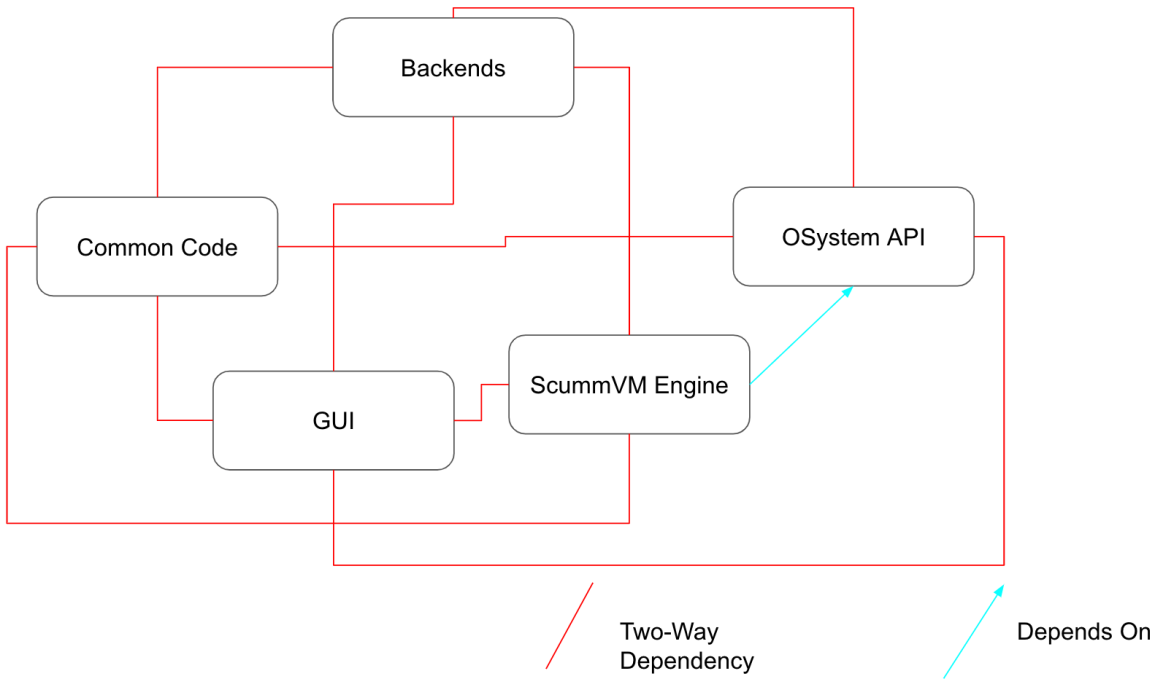


Figure 1: Concrete Architecture

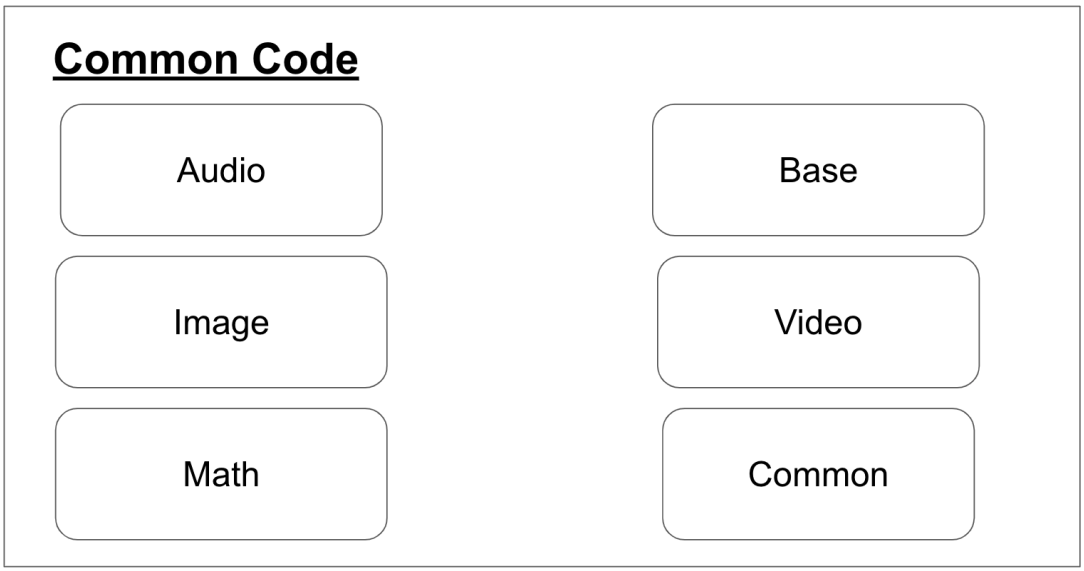


Figure 2: Common Code Subsystems

With the changes out of the way, we can dive into the subsystems, their functionality, and their dependencies within the system.

ScummVM Engine

The ScummVM Engine component works to interpret the commands or scripts pertaining to each game's data files. Each game engine (e.g. Sci, plumbers, kyra) acts as an interpreter for these games, translating game specific commands and logic for the ScummVM system. The engines read, parse, and execute the game data in real-time, reflecting the interpreter style of architecture; It allows ScummVM to emulate these classic games without modifying the game data files.

Taking a more detailed look at the **Sci engine**, we can see it is suited for the interpretation of classic Sierra adventure games. It works to abstract the platform-specific functionality, allowing the games meant for older systems to be able to run on more modern systems. Looking at the file directory of this specific engine, we can see some of the files responsible for the core functionality of the engine.

- *parser/* handles the parsing of the script files, translating it into instructions for the engine to execute.
- *resource/* handles game assets such as sprites, sounds, and text.
- *graphics/* and *video/* handle the rendering of the game's visuals.
- *sound/* handles the audio aspects of the game emulation
- *event.cpp* and *event.h* handle events in the subsystem; Most likely user inputs and mapping them to in-game actions.
- *console.cpp* and *console.h* is likely to be a developer console for debugging and testing.
- *detection.cpp* and *detection_tables.h* help to identify the loaded game and determine which game scripts and resources to load.

GUI

The GUI provides the method for which the user is able to interact with the ScummVM engine. It works with the interpreter to allow the user to control certain aspects of the engine such as loading and quitting games, changing the settings, and controls for the game.

Backends

The backends are a platform-dependent code that works to implement the OSystem API for various systems. This subsystem allows the game engines' commands to run on multiple platforms by standardizing the relationship between these commands and the system's resources.

Common Code

Common code consists of 6 sub-components, these being Audio, Image, Video, Base, Math, and Common. Every other component has a dependency on common code because it provides critical utilities, data structures, and shared functions that are essential for the system to operate. Essentially, common code acts as the backbone of the system which allows all other components to function cohesively. This shared reliance eliminates redundancy and ensures consistency across the codebase.

Audio:

The Audio sub-component is responsible for handling all audio-related operations such as: Decoding and playing sound effects, managing audio mixing, controlling audio settings like pitch, volume, and balance, and providing utilities for audio synchronizations.

Image:

The Image sub-component is responsible for handling graphical operations such as decoding and loading image files, manipulating sprites and textures for in-game visuals, converting and optimizing image data, and supporting features like color palettes and transparency.

Video:

The Video sub-component handles video playback and related tasks, such as decoding and playing cutscenes or in-game movies, synchronizing video playback with audio, supporting various video file formats, and handling scaling, aspect ratio and other video transformations.

Base:

The Base sub-component holds the foundational classes, configuration managers, and initialization code that bootstrap the system. Configuration management refers to handling user settings and preferences. Initialization means setting up the core components and ensuring dependencies are loaded in the correct order.

Math:

The Math component provides mathematical utilities and algorithms used throughout the ScummVM system. These are essential for transformations and calculations, data manipulation with utility functions for arrays, vectors, and matrices, and game logic, handling numerical operations required by game engines.

Common:

The Common sub-component provides general purpose utilities, such as Core data structures, platform independent abstractions for file I/O, memory management, and debugging, and shared algorithms across many components such as compression and string manipulation.

OSystem API

OSystem API serves the purpose of being an abstraction layer between ScummVM and the underlying operating system of the user. Its main role is to provide an interface for accessing platform dependent resources such as: Graphics and rendering, Input management, Sound and Audio, File access, etc. OSystem API has dependencies with almost all other components within the system. It has a dependency with Common Code because common utility functions rely on the OSystem API to access platform specific operations like file access and memory management. The reason ScummVM Engine depends on OSystem API is due to the engine needing to interact with the hardware for rendering graphics, managing inputs, and accessing sound and file services. The GUI depends on the OSystem API for rendering interface elements like menus or buttons, as well as processing input events like mouse clicks or keyboard shortcuts. Finally, the dependencies between Backends and OSystem API are because the OSystem API must interact with backend-specific implementations to support different platforms. For example it might invoke a platform specific graphics library.

Reflexion Analysis

After deriving our concrete architecture, there were a few discrepancies between the concrete architecture (Figure 1) and the conceptual architecture (Figure 3). We performed a reflexion analysis to investigate these discrepancies; Below we have outlined the analysis, what each divergence was and why it occurred.

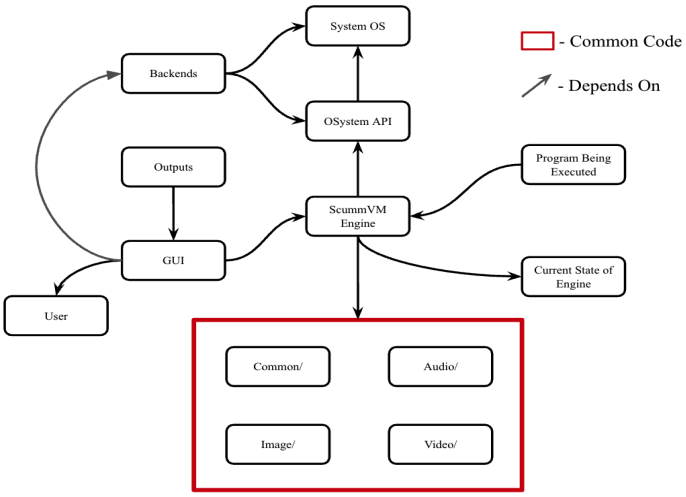


Figure 3: Conceptual Architecture

ScummVM Engine

The ScummVM Engine was found to have two-way dependencies between every subsystem in the concrete except for OSystem API in which it had a one-way dependence. These dependencies were not directly shown in our conceptual architecture; We did not have a link between backends and the ScummVM engine, instead it was linked through the OSystem API. The backends help the engine to work

on multiple platforms with a level of abstraction. The ScummVM engine also absorbed the ‘program being executed’ and ‘current state of engine’ subsystems, which is somewhat reflected in our conceptual architecture as the only relationships the two have are to the ScummVM engine.

GUI

The GUI had one new dependency, that being a two-way dependency between the OSystem API and itself. The GUI uses the OSystem API to be able to generate graphical imagery, and handle user input in a platform-independent manner. It allows the GUI to focus on these aspects while the OSystem API abstracts the complexities of working with differing operating systems and/or hardware.

Backends

The backend subsystem had a new dependency on the ScummVM engine that was not seen in the conceptual architecture. This was due to the OSystem API being seen as a sort of middle-man in the process; The OSystem API would act as an abstraction layer between these two components. After analysis however, this was shown to be true to a point however there were still some direct dependencies between the ScummVM engine and the backends, such as with certain more simplistic information that did not require abstraction.

Common Code

The common code was shown to have dependencies between all subsystems in our concrete architecture. This was not seen in the conceptual architecture as the common code seemed more closely related to the ScummVM engine itself, however after analysis, we can see that the common code contains more file systems that previously thought, namely with the addition of ‘math\’ and ‘base\’, which further helps to support the concept that all the subsystems rely on common code in the overall system.

OSystem API

Compared to the concrete architecture, the OSystem API is missing its dependency to the GUI in the conceptual architecture. This was missed due to the idea that the GUI was solely used by the ScummVM engine, however as stated prior in the GUI section of the reflexion, the GUI must interact with the OSystem API in order to abstract the differences in using different platforms to run the ScummVM engine.

Discrepancies

As stated previously in the high-level architecture section, some of the subsystems found in the conceptual architecture did not translate to the concrete. This is due to several reasons, primarily because of the lack of need for them within the concrete. The outputs, system OS, and user are left out for obvious reasons, that being they do not have an internal role in the software, but are more by-products in the case of the output, and an external factor in the case of the user and system OS. In terms of the program being executed and current state of the engine, they are not separate entities when looking at the actual structure of ScummVM. They exist in the concept to help illustrate the process of the software however, do not exist in the actual software since they more closely fall under one subsystem, that being the ScummVM engine.

Low-Level Architecture: OSystem API

To dive further into the ScummVM architecture, we will take a step deeper into the codebase and analyze one of the core components of ScummVM defined in our high level architecture; the OSystem API. This subsystem is located in the source code at **scummvm/common/system.h** (C header file). Breaking down and analyzing the system.h file allowed us to gain a strong understanding of what this subsystem is responsible for, and how it carries out these responsibilities. Our team has broken down this

system.h file into multiple subsystems and components in order to better understand the functionality of the system. This interface serves as an interface for the system to interact with ScummVM backends. In particular, a backend provides: A video surface for ScummVM to draw in, methods to create timers, methods to handle user input events, control audio CD playback, and sound output.

Subsystems and Components of OSystem API

1. Core

Components:

OSystem Class:
Acts as the central interface for managing all subsystems and components, providing lifecycle management for subsystems such as audio, graphics, timers, and file management.

Engine Initialization and Lifecycle Management:
Handles backend initialization (initBackend) and shutdown tasks (destroy).
Provides methods for engine-specific initialization (engineInit) and deinitialization (engineDone).
Tracks backend initialization state (backendInitialized).

Task Management:
Enum-based task tracking (e.g., cloud downloads, local server activities).
Notifies backend about task start (taskStarted) and completion (taskFinished).

Dependencies:

The core is not dependent on other subsystems of the OSystem API; however, it is important to note that almost all other subsystems are dependent on the core for management.

2. Graphics

Components:

Virtual Screen Management:
Manages the virtual screen dimensions (initSize) and pixel format.
Handles screen updates (updateScreen, lockScreen) and shaking effects for certain game styles.

Graphics Modes:
Lists supported graphics modes (getSupportedGraphicsModes) and enables mode switching (setGraphicsMode).

Overlay Layer:
Implements overlays for GUI elements. Handles overlay-specific rendering methods (showOverlay, hideOverlay, and copyRectToOverlay).

Cursor Management:
Provides low-level APIs for cursor visibility (showMouse), positioning (warpMouse), and appearance (setMouseCursor).

Dependencies:

The graphics subsystem relies on the core component OSystem class for management of its operations. It is also dependent on the timer subsystem for synchronized updates such as screen refresh and animations.

3. Audio

Components:

Mixer Management:
Provides access to the audio mixer (getMixer) for handling sound playback.

Audio CD Management:
Offers interfaces for managing audio CDs (getAudioCDManager).

Dependencies:

Relies on core OSystem class to provide an entry point for accessing audio-related components. Dependent on the timer subsystem for synchronization of playback.

4. Input

Components:

Event Handling:
Event manager for processing user inputs (getEventManager).

Keymapper Integration:
Provides default bindings and platform-specific keymaps (getGlobalKeymaps and getKeymapperDefaultBindings).

Dependencies:

Though not explicitly shown in system.h, input functionalities in backend may rely on TimerManager to coordinate with input events for timed interactions.

5. Timer

Components:

Timer Manager:
Offers timing-related functionalities through a timer manager (getTimerManager).

Dependencies:

Not dependent on other subsystems but is dependent for Audio, Input, and Graphics.

6. Storage

Components:

Save File Management:
Interfaces for managing save files (getSavefileManager).

Config Management:

Manages reading and writing of configuration files through streams (createConfigReadStream, createConfigWriteStream).

Filesystem Factory:
Provides a factory interface for interacting with the underlying file system (getFilesystemFactory).

Dependencies:

Dependent on core for management and initialization of file-related components.

7. Miscellaneous

Components:

Dialogs and Messaging:
Displays messages on the screen (displayMessageOnOSD) and provides dialog management.

Clipboard Management:
Implements clipboard operations (getTextFromClipboard, setTextInClipboard).

Taskbar and Updates (Optional):
Manages taskbar-related features (getTaskbarManager) and updates (getUpdateManager).

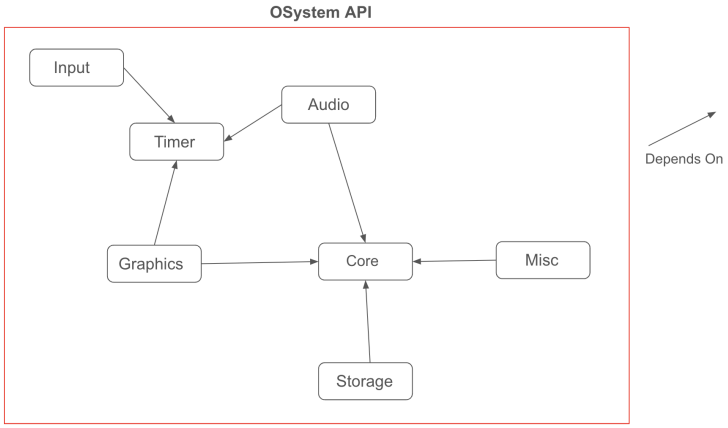
DLC Store:
Integrates a DLC store for downloadable content (getDLCStore).

Localization:
Retrieves the system language (getSystemLanguage).

Dependencies:

Dependent on core OSystem class as it is the access point for these components.

Concrete Architecture for Low-Level Subsystem



Reflexion Analysis: OSystem API Concrete vs. Conceptual Architecture

In order to conduct a reflexion analysis, we compared our concrete architecture of the OSystem API subsystem to its conceptual architecture described previously in Assignment 1. The following is a summary of noted discrepancies and the rationale behind them:

Overall, our conceptual architecture simplifies the OSystem API's role, focusing on its bridging functionality, while the concrete architecture portrays it as a central hub managing multiple subsystems and advanced functionalities. Essentially, the conceptual architecture abstracts away implementation details to highlight its integrative role, whereas the concrete architecture provides an exact detailed breakdown. The degree of detail to which our conceptual architecture is described is much lower than its concrete architecture, with all functionality under the OSystem API being grouped without differentiating between subsystems such as graphics, timer, and audio. At the conceptual level, these lower subsystems were seen as implementation details less relevant to the high-level conceptual design.

The lack of OSystem API's subsystems in the conceptual architecture posed another discrepancy in regards to subsystem interdependencies. While the conceptual architecture has no mention of them, the concrete architecture highlights interdependencies, such as graphics relying on timers (for synchronized updates), and audio synchronization with the timer subsystem. The rationale for this discrepancy is the same as the previous one, since key interactions and dependencies present in OSystem API were overlooked along with its subsystems in the conceptual architecture.

Advanced features were also omitted from our conceptual architecture, while they are present in the concrete architecture as taskbar management, DLC integration, clipboard operations, and localization to name a few. Each of these features is grouped under the miscellaneous module in our concrete architecture. This discrepancy falls in parallel with the others, as OSystem API is only generally explained in the conceptual architecture versus the in depth technical concrete architecture.

Use Cases

Diagram for starting game:

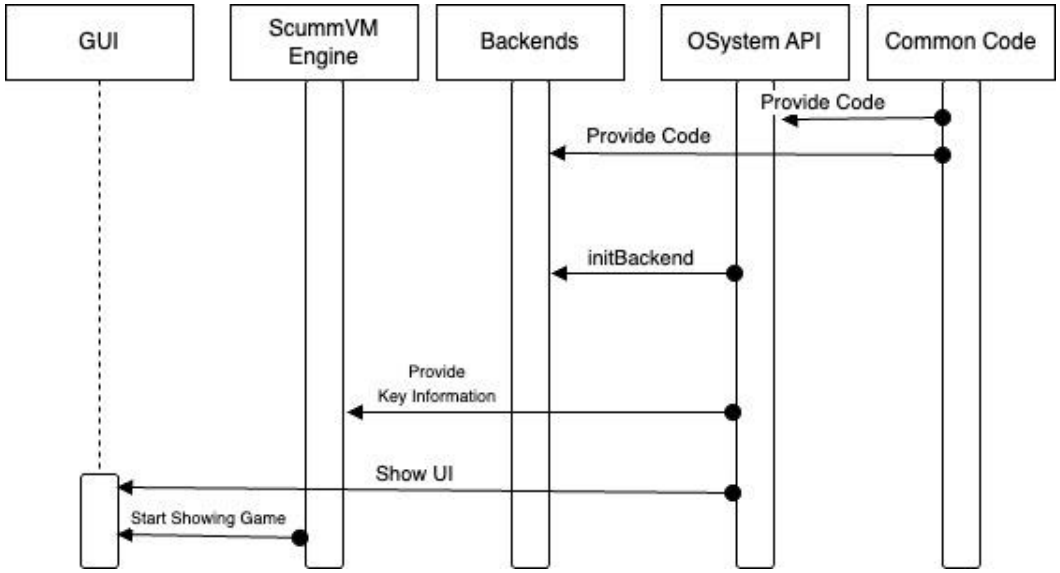
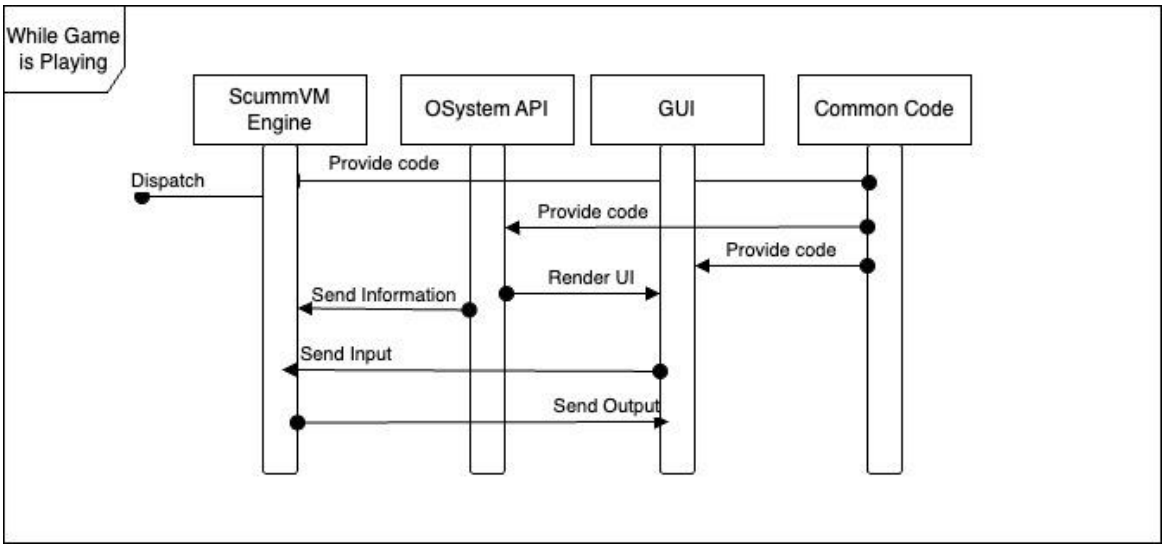


Diagram for playing game:



The first updated use case diagram shows how ScummVM loads a game. First, the common code provides important code to multiple components, including the OSystem API and the backends. This serves as a backbone to the system. Next, the OSystem API initializes the backend. The backend implements the API for the various systems ScummVM can run on. The OSystem API also helps provide key information for the engine like graphics and inputs. The API also helps the GUI render the interface elements. Finally, the engine renders the game's visuals.

The second updated use case diagram shows how ScummVM plays a game. First, the common code provides important code to the GUI, OSystem API and ScummVM engine components. The OSystem API helps the GUI render interface elements, as well as processing inputs like mouse clicks. The OSystem API also sends important information to the engine like providing graphics rendering, input management and sound and file services. The GUI also sends inputs to the engine, and the GUI also displays output from the engine. All these events keep happening while the game is running.

Conclusion

In conclusion, to account for its highly flexible nature, ScummVM's concrete architecture follows an interpreter architectural style. Moreover, the architecture is modular, backend agnostic, and features flexible initiation. The system is made up of numerous unique subsystems, including the Engine, GUI, Backends, OSystemAPI, and the Common Code (Made up of Audio, Base, Image, Video, Math, and Common). These components all play a unique role in this system, and are heavily reliant on each other. ScummVM's subsystems all have numerous two way dependencies with other subsystems. We compared our concrete architecture with our conceptual architecture with reflexion analysis and found divergences in all subsystems. We then took a deeper look at the OSystemAPI subsystem to find out how it functions and the purpose it serves. We found that the OSystemAPI could be broken down into numerous subsystems with their own subcomponents. These subcomponents include the Core Components, Input, Audio, Graphics, Timer, File System and Storage, and Miscellaneous. These components also share many dependencies with each other and with other ScummVM subsystems. We also performed reflexion analysis on the OSystemAPI subsystem, and found numerous divergences. We then studied how the system would handle the use cases of starting a game and playing a game. We reached our conclusions by using SciTools Understand to study the files, folders, and dependencies found in ScummVM's source code.

Limitations and Lessons Learned

We came across some difficulties when creating the concrete architecture for ScummVM. We faced some difficulties when mapping files to their proper subsystems. Some architectural components related to the interpreter architectural style such as the program being executed and the current state of the engine could not be mapped to our source code, and thus could not be portrayed in our concrete architecture. Additionally, non code related components such as the user could not be portrayed for similar reasons. All ScummVM game engines besides SCI, Kyra, and Plumber, were not considered when creating our concrete architecture. This is as ScummVM's engine count is massive, and considering all of them would slow down our derivation process greatly due to the increased memory usage for programs such as Understand.

Over the course of recovering ScummVM's concrete architecture we learned many things - both relating to software architecture and group work in general. One thing we learned as a group is how to better incorporate technology into our workflows and communications. Due to our busy schedules we found that it was becoming harder and harder to coordinate in-person meetings during tutorial timeslots. To remedy this, we substituted in person meetings with online meetings using web based communication apps such as Discord. The convenience of this allowed us to schedule meetups more easily and with more flexibility. Moreover, we learned how to use the SciTools Understand application. This app played a pivotal role in allowing us to better understand the complex structure of ScummVM's source code. By using this technology, we were able to more easily navigate through ScummVM's large codebase and get a deeper understanding of the system.

Data Dictionary

- **ScummVM Engine**: The central component responsible for interpreting and executing game-specific scripts for platforms.
- **Current State of Execution Engine**: Keeps track of the current state of the game; Continuously updates the state of the game throughout the gameplay.
- **Program Being Interpreted**: Represents the game data, acts as a sort of blueprint with instructions for the engine to use during the game execution.
- **OSystem API**: Creates a layer of abstraction for handling system tasks such as graphics rendering, receiving inputs, and managing resources.
- **Backend**: Works as a modular section of the codebase that handles platform-specific aspects of the engine.
- **Common Code**: Contains utility functions and codec handling for images, audio, video, and other game content.
- **GUI**: Manages user inputs and displays outputs; creates a visual interface for the engine before and after game selection.
- **Outputs**: Similar to the GUI but is solely responsible for rendering the visual and audio content of the game during gameplay.
- **Script**: The game logic that runs every frame to ensure consistent gameplay. Falls into two categories: main scripts (i.e. for the entire game) or ephemeral (i.e. short-term actions like user interactions).

Naming Conventions

- **AGI**: Adventure Game Interpreter
- **API**: Application Programming Interface
- **GUI**: Graphical User Interface
- **MIDI**: Musical Instrument Digital Interface
- **OS**: Operating System
- **SCI**: Sierra's Creative Interpreter
- **SCUMM**: Script Creation Utility for Maniac Mansion
- **SDL**: Simple DirectMedia Layer
- **UI**: User Interface
- **OSystem API:** Operating System Application Programming Interface

References

https://www.scummvm.org/
https://wiki.scummvm.org/index.php?title=Developer_Central
https://doxygen.scummvm.org/index.html
https://github.com/scummvm/scummvm