CISC 322

A1

SCUMMVM Conceptual 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

We examine the high-level conceptual architecture of ScummVM in this report. ScummVM is an open-source piece of software that recreates old adventure game engines, and allows users to run them on various modern systems. In this report we discuss the conceptual architecture, its subsystems and concurrency among other things.

We determined that ScummVM uses the Interpreter architectural style to accomplish the task of interpreting various game engines for users to play. To that end, we have analyzed ScummVM and discovered that it utilizes 8 main components. These include the main ScummVM engine, the GUI, the outputs, the program being interpreted, the current state of the engine, the OSystem API, the backend, and common code. Every component works together to ensure smooth execution of adventure games. To further demonstrate the system, we included two use cases, which are loading a game and running a game respectively. We also examine the evolution of ScummVM and how it came to be what it is today, an overview of the concurrency present in the system, the division of responsibilities for developers, the external interfaces and the use cases. Finally, we summarize our discoveries in the conclusion, as well as listing some of the limitations we encountered and lessons we learned.

Intro/Overview

ScummVM is an open-source software created to allow users to play classic point-and-click style games. Rather than emulating the old hardware used for these games, the engine reinterprets the games engines in order for it to run smoothly on modern systems such as Windows, macOS, game consoles, etc. By continuously adding support for new platforms, this software helps in game preservation by ensuring that these games are not lost to history. This report will present an in-depth analysis of the conceptual architecture of ScummVM; Discussing the core components (architecture, concurrency, evolution, etc.) and its organisation in order to better understand this game engine.

By looking at the architecture of this engine, we can start to understand the structure of the system and how each component interacts with one another. The architecture of ScummVM is based on the interpreter style, in which the components such as the GUI, engine, OSystem API, and backend work together to interpret the games. Each of these serve a specific function, whether it's interpreting game logic, handling user input, managing system files, etc.

Moving forward through the report, the evolution of the engine is explored. ScummVM was originally developed to run LucasArts adventure games, however, ScummVM's scope has grown significantly to support a wide variety of games from a wide variety of developers. This expansion was possible due to the modular architecture approach used in the engine, allowing for the integration of new engines without compromising the core of its functionality. Overtime, the architecture has evolved to become more flexible, creating support for more games and hardware platforms.

An essential part of ScummVM's architecture would be concurrency. Rather than employing multithreading, ScummVM uses a cooperative threading model. In this way, threads voluntarily give up control to prevent complex synchronization issues. This approach helps with the system's concurrency by ensuring tasks (e.g. audio streaming, input handling, etc.) can be performed efficiently without overloading the system. In this way, ScummVM ensures smooth and consistent gameplay regardless of the platform.

The next topic of this report discusses the division of responsibilities within ScummVM's development model. ScummVM is organized in a way that allows different teams to focus on specific areas of its development. This modular approach allows for a large number of contributors to work on the development without interference or overlap; With each team responsible for a particular component of the system, helping to reduce the complexity of managing a large-scale project such as this one.

The external interfaces of ScummVM. This aspect enables seamless interaction between the different components in the software (Users, game engines, etc.). This platform design ensures that ScummVM can run on a multitude of different devices without requiring major code changes. This flexibility highlights ScummVM's portability and ease of use.

Ending off the report by highlighting two use cases from this software. They demonstrate how ScummVM loads and plays a game, showing the inner workings of the system in a digestible manner. The engine first identifies the operating system using the backends and OSystem API in order to help manage system resources. Then, it interprets various game engines with assistance from common code that provide images and audio. Then when the game is launched, the engine continuously processes various streams of information in order to update the game state accordingly throughout gameplay. The outputs are then rendered and displayed through the GUI as the game progresses.

In summary, this report highlights the architecture of ScummVM and how it interprets/runs a wide variety of adventure games across multiple platforms. It has elements of scalability and adaptability, highlighted in its evolution from a simpler emulator of LucasArts games to a more versatile system that can handle numerous game engines. Through its use of modularity and platform abstraction, ScummVM is able to bring these classic gaming experiences to modern hardware. This design allows ScummVM to remain a vital tool for preserving and playing said games.

High Level Conceptual Architecture

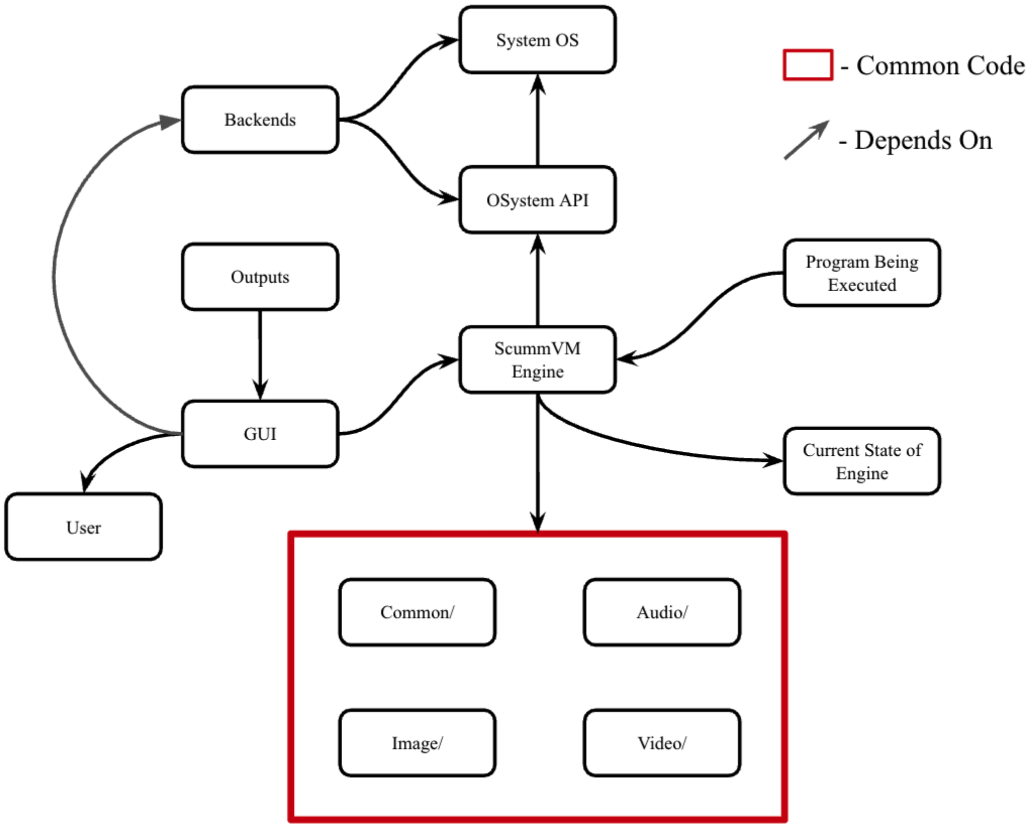
To derive our high level conceptual architecture, we first needed to do some research just browsing the provided pages along with some we found on our own so we could get a good understanding of what the program is and fundamentally how it works. After this, we used a combination of the wiki page, along with an analysis of some of the code from the official ScummVM Github page to come up with our components and understand some of their interactions and dependencies. Our components consist of: ScummVM Engine, GUI, Common Code which is split up into Image, Video, Common, and Audio, Backends, OSystem API, Program being executed, Current State of Engine, Outputs, and System OS. To deepen our understanding of the overall architecture, we reviewed the documentation for each high-level component to analyze the interdependencies and examine the control flow throughout the system.

After all of this we decided that the most logical architectural style to follow would be the interpreter style. With each of the components representing a key part of the total system. Each of the lines shows a dependency that one component has on another. Each of the components has a different function, varying from GUI and output being what the user would see, to the ScummVM Engine for interpreting and executing scripts, to the System OS which is the operating system of the hardware running the program.

Subsystems

ScummVM Engine

This component sits at the centre; It's responsible for interpreting and executing game-specific scripts for the game engine. It handles all the various incoming information from the current state of the engine, the OSystem API, as well as the common code in order to create



the graphical adventure game that is being interpreted; Creating and animating the characters, processing the user input, etc.

Current State of Execution Engine

Represents the states of the game, including memory and execution context; It is updated continuously throughout the gameplay. I.e. keeping track of the current progress of the game ***

Program Being Interpreted

This component can be looked at as the game data; it handles game files being processed by the execution machine. This component acts as the blueprint by passing along the instructions that guide how the engine will behave and which operations are performed. ***

OSystem API

Serves to abstract the rendering of graphics, receiving of input, and managing system resources from the game engine. In order to work on the game engine, interacting with the OSystem API is a must in order to understand what functionality is available. However, in order to port ScummVM to a new platform, the backend code will be the area of focus.

Backend

The only part of the codebase that is dependent on the platform. The modular category of the backend code works by distributing the OSystem API implementation into many different modules, the most notable one being the GraphicsManager which handles all the graphical information being sent to the GUI. Code is shared and the SDL library is used in these modular backends.

Common Code

Provides utility functions and codec handling to support game interpretation. Houses the files for images, videos, audios and other common data required in the interpretation of the ScummVM engine

GUI

Handles user inputs while providing a visual output to the user in the form of the game display, engine menu, game selection, etc. Allowing the user to interact with the visuals of the engine. I.e. encompassing everything outside of the gameplay itself; the interface the user sees before and after a game.

Outputs

While the GUI component handles the visual aspect, the output component is focused on rendering the game content itself. This component manages how the game visuals are drawn during gameplay as well as providing the audio output.

Evolution of the System

ScummVM has evolved significantly since its initial release in 2001. Originally developed to run LucasArts adventure games, such as *Maniac Mansion* and *Monkey Island*, ScummVM was designed to emulate the SCUMM engine used by LucasArts to script their adventure games. The initial architecture of ScummVM was focused on reverse-engineering the

SCUMM engine, allowing the software to interpret SCUMM based game scripts and enable users to play these games on modern hardware at the time without the original engine.

Over time, the scope of ScummVM expanded beyond LucasArts titles. The system's architecture was modified to be more modular, supporting additional game engines, such as Sierra's AGI/SCI engines, Revolution Software's Virtual Theatre, among others. This shift required a more generalized architecture that could support multiple game engines, each with its own set of features. The core of ScummVM's architecture became more abstract, with a unified API layer allowing the integration of different game engines while maintaining a consistent interface for game interpretation, input handling, and resource management.

As mentioned, a key aspect of ScummVM's evolution is its modularity. The software system is structured around the concept of plugins, where each game engine is essentially a separate module that can be loaded and unloaded as needed. This architecture supports the easy addition of new engines over time, without affecting the stability of the system or requiring significant changes to its core. The abstraction layers within the architecture, particularly around file handling, input processing, and graphics rendering, allowed ScummVM to adapt to different platforms and hardware environments, which has enabled the project to support a wide array of devices from desktops to mobile platforms.

With portability being a significant technical focus in the evolution of ScummVM's architecture, the development team adopted platform-independent libraries and coding practices to ensure that the software could be compiled and run on various operating systems, such as Windows, macOS, Linux, iOS, and Android. The use of SDL (Simple DirectMedia Layer) as the underlying multimedia library has been crucial in achieving this portability. SDL provides a standardized API for handling input, output, and graphics across different platforms, simplifying the process of adapting ScummVM to new devices without having to rewrite core components.

Memory management and performance optimization also became important considerations as the scope of the project increased. ScummVM had to balance the complexity of running multiple engines while ensuring efficient use of system resources. The development team introduced multiple performance improvements over time, optimizing both CPU and memory usage. Some of these include game engine code optimization and refactoring, implementation of more efficient rendering and graphics algorithms, and improved multi-core and multi-threading efficiency. These advancements allowed ScummVM to run more resource-intensive new games, as well as perform on low-power devices like handheld consoles and smartphones. The system's efficient architecture allowed it to emulate classic games while maintaining a lower overhead.

In summary, ScummVM has evolved from a specialized emulator for SCUMM-based games into a broad system capable of running various game engines across multiple platforms. Its architecture reflects this evolution, with modularity, abstraction, and portability as central improvements. In addition, by focusing on extensibility and platform independence, ScummVM has been able to grow its capabilities while maintaining a stable core system that can adapt to new challenges and platforms. The result is a flexible, robust, and efficient system that serves as a bridge between classic adventure games and modern hardware.

Concurrency in ScummVM

Traditional multi-threading isn't used in the game's logic due to the core engine operating primarily on a single threaded architecture however, this doesn't mean there is not extensive use of concurrency in the system. **The virtual machine is a simple byte-code machine with built-in cooperative threading**; up to 25 threads can be run simultaneously but only one thread may execute at one time. Each thread gets 16 words of local storage. Handling elements like audio streaming, UI interactions or background resource loading in separate threads helps in maintaining consistent playback/gameplay without affecting the main execution flow. This approach to concurrency greatly simplifies synchronization since threads yield control voluntarily. This allows for less complex synchronization mechanisms, reducing overhead and greater cross-platform portability since it can operate on both single and multi core systems.

Thread states:

State	Meaning
RUNNING	the thread is currently executing.
PENDED	the thread was executing, but has spawned another thread. When the child thread is descheduled, the pended thread will start executing again.
DELAYED	the thread is not executing, and is waiting for a timer to expire.
FROZEN	the thread is suspended and will not execute until it has been thawed again.

*table from ScummVM docs.

A thread will run once every frame, unless it is delayed or frozen. Any code may start a new thread at any point. (Sometimes this is done automatically.) The new thread will immediately run until it is descheduled, at which point execution continues in the parent thread. A thread may spawn another thread at any point. When this happens, the parent thread moves into the PENDED state and stops executing. The child thread starts executing immediately. When the child thread is descheduled, the parent moves back into the RUNNING state and continues on immediately after the instruction that created the thread. PENDING threads may be nested up to 15 times.

Resource Sharing

Up to 32768 bits of global storage are available. These global storage areas are shared among all threads, allowing for communication between different parts of the game. In the Version 6 machine, the stack is shared among all threads. Array resources are also shared across all threads. In Version 6, arrays can be defined on the fly, allowing this array storage to be used as a dynamic heap. The thread states are also stored on the game engine and can be affected by different threads. It's important to note that while these resources are shared, the cooperative threading model used in ScummVM ensures that only one thread is actively executing at any given time. This design helps mitigate many traditional concurrency issues such as race conditions, deadlocks or context switching overhead while still allowing for a form of multitasking within the game engine.

Scripts

Each script has the opportunity to run once per frame (with some exceptions) sharing execution time. These scripts run on a per-frame basis in order to maintain consistent game timing. There is (usually) a main script for the whole game/room. On top of this main script there exists ephemeral scripts. For example, a script that checks where the user clicks in a room may run continuously until the user exits that room. These ephemeral scripts are only used once and then stop. These scripts execute **concurrently** with the main script execution.

Conclusion

In conclusion, the concurrency model in ScummVM is primarily based on cooperative threading, which simplifies synchronization by allowing threads to voluntarily yield control. This model, combined with shared global resources and per-frame script execution, allows for effective multitasking without the complications and complexity of traditional multithreading. The modularity of ScummVM's architecture, where game engines operate independently, allows it to handle multiple game titles from different platforms efficiently. Despite the simplicity of the virtual machine and cooperative threading, the design supports concurrent operations like audio streaming, UI handling, and background resource loading, ensuring gameplay remains smooth throughout execution, across various platforms.

Division of Responsibilities for Developers:

ScummVM is an open-source project, meaning that the project has been completed by a high number of volunteers contributing smaller pieces of code. According to the copyright file and credits page, several hundred developers have worked on ScummVM and the project is still garnering new developers. Moreover these developers are geographically dispersed and work remotely through a remote repository hosted on Github. These traits have the potential to be problematic. The sheer amount of developers and the lack of in-person organization and communication may lead to developers interfering with each other. To account for this open-source development model, ScummVM's architecture was structured in a way that can support a large number of developers working on the project simultaneously.

ScummVM is made up of a large number of small modules. This trait is beneficial for distributing responsibilities among a large number of developers. The ScummVM developers are organized into sub-teams, and each of these sub-teams is responsible for implementing one of these modules. This team style organization prevents the potential risk of developers getting in each other's way that may otherwise plague an open-source project. The structure of the game engines component exemplifies ScummVMs modularized architecture, and how developer responsibility is distributed. The game engines component is made up of many smaller individual engine modules that correspond to specific game engines. In total, ScummVM has over 50 engine modules, which are completed by sub-teams ranging in size from 1-41 member(s) depending on the complexity of the engine. This module sub-team style of responsibility allocation is seen throughout the entire ScummVM project.

ScummVM's engines were the system component that took by far the highest number of developers to complete due to the immense number of engines that needed to be implemented. The backends also required a large amount of developers due to the amount of hardware that needed to be accounted for. It is implied that the Osystem API and GUI components took few developers to complete, as they are more standard components that don't contain numerous submodules.

Moreover, there are many guidelines that all developers are responsible for following in order to maintain code quality and communication. Following coding conventions is important to maintain program portability. Code formatting guidelines are important for improving readability, which will benefit other developers working the code. Git committing guidelines help maintain clear communication. Additionally, there are numerous crucial ground rules. For example, developers must make sure their code compiles before committing it, should ask other devs before making changes to their code, and must consult team leaders before making changes to crucial components including the Osystem API and common code.

External Interfaces

The external interfaces of ScummVM are designed to provide a smooth interaction between the user, the underlying game engines, and the platform on which it runs. From a technical standpoint, these interfaces can be broadly categorized into user interfaces, platform interfaces, file system interfaces, input device interfaces, audio and video output interfaces, and external tool interfaces.

User Interface

The user interface is the primary way users interact with ScummVM. It is designed to be lightweight and straightforward, allowing users to launch and configure games without extensive technical knowledge. The UI provides a unified interface capable of running different game engines to launch any game supported by ScummVM. It also includes configuration menus for game-specific settings such as graphics, sound, and control schemes, which are consistent across platforms. This abstraction is a crucial part of ScummVM's conceptual architecture, as it separates the internal functions of game engines from the player's experience, ensuring ease of use while maintaining flexibility for future extensions.

Platform Interfaces

ScummVM's platform interfaces ensure compatibility with a wide range of operating systems and hardware, including Windows, macOS, Linux, iOS, and Android. This is achieved through platform abstraction layers, which use libraries like SDL to handle input, audio, video, and file operations. The system leverages SDL to manage multimedia and device-specific operations, ensuring that ScummVM can run games across different devices with minimal changes to its core architecture. The platform interfaces abstract the differences between these systems, allowing the game engines to interact with hardware in a consistent manner regardless of the underlying platform.

File System Interfaces

The file system interfaces in ScummVM play a critical role in managing game data, ensuring that the system can access and handle game assets from various formats and storage locations. ScummVM interacts with the file system to load game resources such as scripts, audio files, and graphics, typically extracted from the original game files or installed game directories. The architecture abstracts the file system, allowing ScummVM to work with different file formats and storage mechanisms, such as local files, CD-ROMs, and even compressed archives like zip files. This interface is designed to be flexible and portable, adapting to various file systems across different platforms. ScummVM also includes support for virtual file systems that simulate a unified file access layer, ensuring consistent access to resources regardless of the underlying storage type.

Input Device Interfaces

Input device interfaces allow ScummVM to handle various input devices such as keyboards, mice, game controllers, and touchscreen. Through abstraction of input device handling, consistent interaction across different platforms is possible. Various modes of input, including mouse and keyboard clicks, touch gestures, and even game controller inputs, are all supported. It's important to note that game controller support is handled through standardized APIs, ensuring that ScummVM's input system remains flexible.

Audio/Video Output Interfaces

These interfaces are responsible for managing sound and graphical rendering. The system abstracts audio mixing, MIDI playback, and video rendering through libraries like SDL and other platform-specific APIs. This allows ScummVM to reproduce audio and video content accurately across various devices. The system's video output interface ensures proper rendering of in-game visuals, adjusting resolutions and handling video scaling to suit different hardware configurations. These interfaces maintain the integrity of the original gaming experience while adapting to modern display and sound technologies.

External Tool Interfaces

ScummVM's external tool interfaces facilitate interaction with external utilities and development tools. These interfaces allow for operations like save file importing and exporting, resource extraction, and debugging without altering the core system. For example, users can use external tools to convert save files from different platforms, and developers can access built-in debugging tools to examine game behaviour and engine performance. This external tool support provides additional extensibility and enhances ScummVM's flexibility, allowing advanced users and developers to interact with the system beyond its standard game-playing functionalities.

Use Cases

Diagram for starting game:

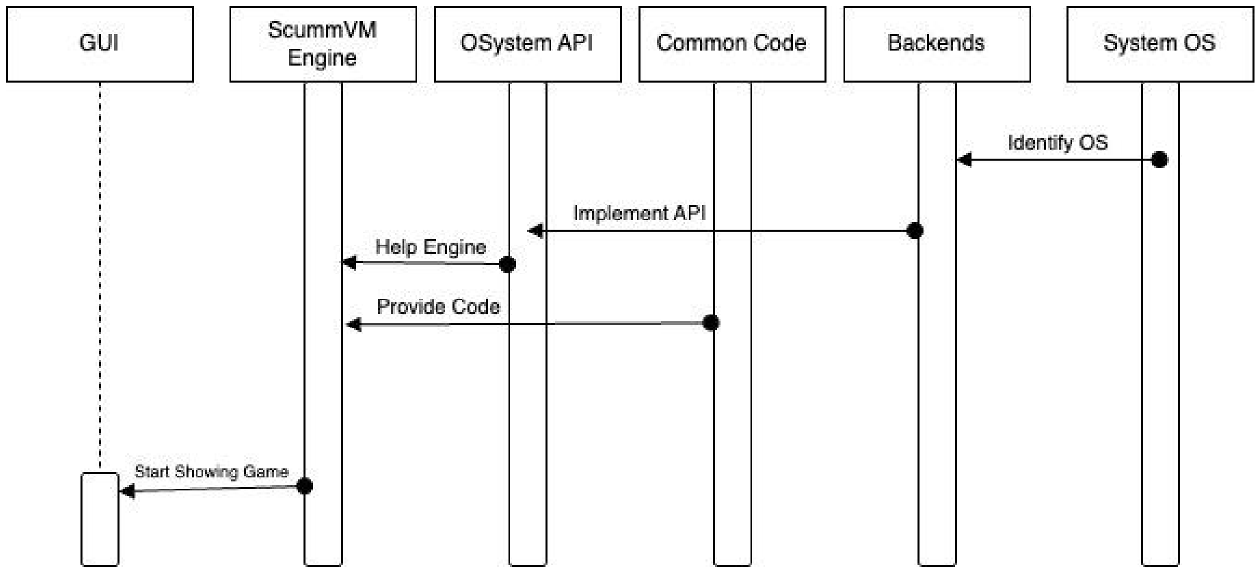
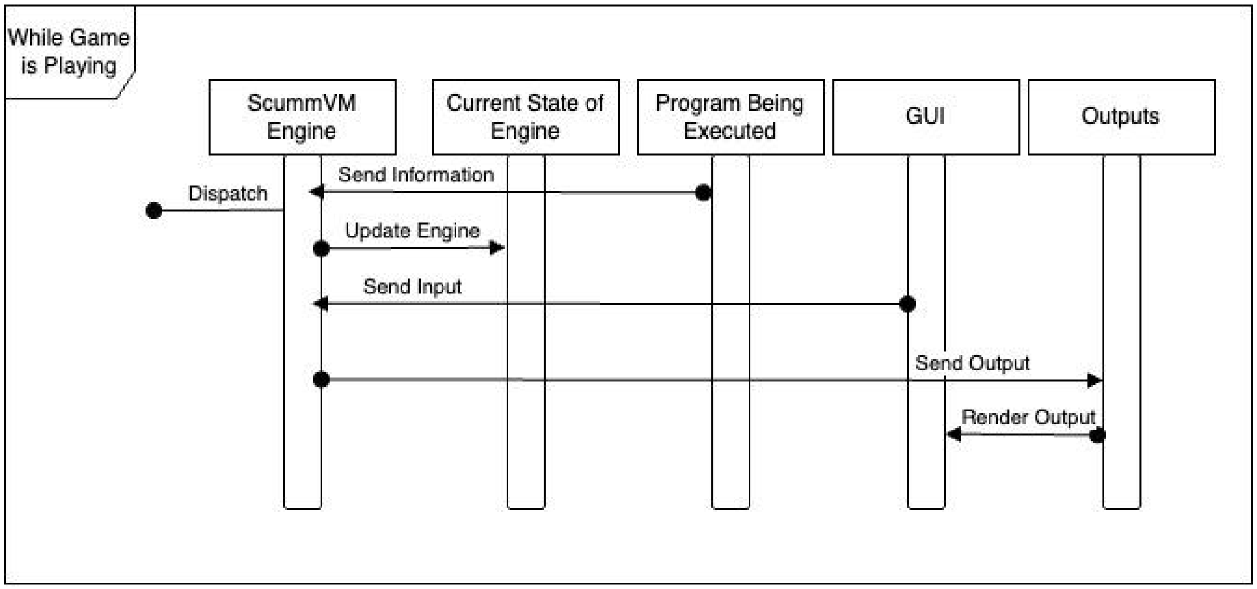


Diagram for playing game:



The following use case will show how ScummVM can load and play a game. This will help to explain the inner workings of ScummVM.
To start, ScummVM needs to determine what OS it is running on. This can range from anything like Windows and MacOS to platforms like Android and even the Playstation Portable. The backends help with this, distributing the OSystem API. The ScummVM engine can interpret various adventure game engines like its namesake SCUMM to AGI or SCI. The OSystem API helps manage system resources for the engine. The common code also helps with game interpretation, providing things like images and audio to the ScummVM engine. Combined, the OSystem API and the common code let the game boot to the GUI.

While playing the game, the program being executed component sends instructions and information to the ScummVM engine component that determines how it behaves. The engine then uses this to update the current state of the engine component, which includes the memory and current progress of the game. When the GUI receives input, like the user clicking a certain spot in the game to move to, that input is sent to the engine, which determines what to do with it. While all of this is happening, the ScummVM engine sends the output to the outputs component, which renders the output and also provides the sound and music to the user. The rendered output is then sent to the GUI component to be displayed. All of this is continuously happening while the user plays the game.

Conclusion

ScummVM's architecture is a robust, modular system designed to recreate and run classic adventure games across a wide range of platforms. Its interpreter-based design and modular components (such as the ScummVM engine, GUI, and OSystem API) allow use of various game engines. Through abstraction of platform dependencies and using common code for media handling, ScummVM maintains a high degree of portability, enabling users to play games on modern and legacy hardware. Its evolution from supporting only SCUMM-based games to a variety of modern game engines reflect its flexibility and scalability.

The development and concurrency model emphasize the system's capability. ScummVM adopted an open-source model with clear developer responsibility division and a multitasking approach. In addition, its event focused concurrency system, supported by occasional threading for tasks like audio processing, ensures optimal performance without overburdening the system. In conclusion, ScummVM's design achieves the balance of running complex game engines while maintaining compatibility across diverse platforms, showcasing its versatility as an enduring tool for preserving the classic gaming experience.

Limitations and Lessons Learned

When researching ScummVM, we encountered some difficulties. For one, ScummVM did not have highly in-depth documentation of their architecture. The public ScummVM repository listed all of the program's components and files, but does not state clearly the way the system is structured and how these components relate to each other. This made parts such as coming up with a conceptual architecture and modelling use cases difficult. To make up for this, we had to make some inferences based on how adventure game engines like ScummVM are typically structured. Additionally, since ScummVM is a very old project that is constantly updating, some of the online information about the project may be outdated. ScummVM has been in development for 23 years, and has been through many different versions. Information about older versions may not apply to the current version. Additionally, due to future updates, this conceptual architecture may also become outdated in the future.

We also learned a lot from the process of making this conceptual architecture report. One thing we learned was the importance of communication. It is important that every groupmate knows what they are supposed to be doing, and is able to share with the group any key information. Additionally, we learned about the importance of regular meetings. Meetings allowed for improved communication, and forced us to keep on track with the project. When we decided to skip meetings we found that we were getting far less work done and falling behind. Moreover, this project helped us improve our research skills. A lot of information was not laid out to us plainly, and we had to dig deep to find it. We had to navigate through the ScummVM documentation and code base to find the information we were looking for. An effective strategy that we used during this project was dividing up workload and assigning every group member a task to do. This made sure that everyone had a job to do, and gave people direction on what they needed to work on. Something that we could have done better for this project was time management. We did not have a consistent work schedule, and struggled to deal with time constraints.

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

References

Wiki Pages:
https://wiki.scummvm.org/index.php?title=Developer_Central#Code_base_structure

https://wiki.scummvm.org/index.php?mobileaction=toggle_view_desktop&title=SCUMM%2FVirtual_Machine
GitHub: https://github.com/scummvm/scummvm
SDL: https://www.libsdl.org/
Documentation: https://docs.scummvm.org/en/v2.8.0/index.html
ScummVM Forums:
https://forums.scummvm.org/viewtopic.php?t=13775
https://forums.scummvm.org/viewtopic.php?t=14380
ScummVM Credits page: https://www.scummvm.org/credits/
Dev: https://dev.to/roperzh/scumm-internals-and-syntax-for-the-sake-of-nostalgia-384j
Linux Conf Au:
https://www.youtube.com/watch?v=QihSN7VCrB0&ab_channel=linuxconfau2017-Hobart%2CAustralia