1000101: ScummVM Conceptual Architecture

Jasper Nie (20jhn3@queensu.ca)

Vlad Lisitsyn (vlad.lisitsyn@queensu.ca)

Alex Ivanov (22ai11@queensu.ca)

Isaac Chun (21ihc1@queensu.ca)

Benjamin Bartman (20bb27@queensu.ca)

Sabin Pokhrel (sabin.pokhrel@queensu.ca)

CISC 322

Software Architecture Conceptual Report

October 11, 2024

Contents

1 Abstract 1

2 Introduction and Overview 1

3 Architecture 3

3.1 Components, Subsystems, and Modules 3

3.2 Architectural Style 4

4 Diagrams 5

5 External Interfaces 7

5.1 GUI 7

5.2 Game Data Files 7

5.3 Save/Load Files 7

5.4 Other System Files 8

5.5 Audio and Video Output 8

5.6 Plugin Data 8

6 Use Cases 8

6.1 Loading a Game 8

6.2 Saving and Loading Game Progress 9

6.3 Handling User Input During Gameplay 10

7 Data Dictionary 10

8 Naming Conventions 10

9 Conclusion 10

10 Lessons Learned 11

11 References 11

1 Abstract

This report provides an overview of the conceptual architecture of ScummVM, a game engine which ports games onto various platforms such as mobile devices and video game consoles while still keeping the authentic feel of the game. The architecture breakdown puts a particular focus on the subsystems and their interactions which allow for the system to function. The structure of the system is highly modular as all key components such as the OSystem, backend, game engines, common code and the GUI code. These components work together to offer flexibility and portability.

The interpreter architecture style characterizes ScummVM the best, as the system acts as a virtual machine which translates game execution files and provides a multi-platform solution for these various games. This style is crucial to the functionality of the system as it allows it to run games by simulating their native environment. Beyond this main architectural style, elements of Object Oriented (game objects and user interaction), Pub-Sub (handling events within gameplay), and Layered (organizing components) can also be seen. These work together to complement the Interpreter style of the system.

External interfaces including GUI, game data files, save/load files as well as plugin data all transmit user inputs and game information throughout the system which facilitates user interaction. Additionally, there are several key use cases in the ScummVM architecture such as reading game files, saving and loading game progress as well as handling user inputs. These all demonstrate how components of this architecture work together to support gameplay of various games in this engine. It is clear that the system has been designed taking into account testability as well as the possibility of future changes and updates through the well defined subsystems and interfaces that make it adaptable.

Overall, ScummVM's architecture is well-suited to its goal of increasing access to legacy game titles on modern systems.

2 Introduction and Overview

This report explores the conceptual architecture of ScummVM, a software platform designed to bring classic point-and-click adventure games to modern devices like Windows, macOS, Linux, and even mobile platforms. ScummVM serves as an interpreter, taking game files and scripts originally developed for older hardware and translating them into a format that modern systems can execute. The goal is to preserve the authenticity of these games while making them accessible to today's players, regardless of the platform they use. This report provides a detailed breakdown of ScummVM's architecture, focusing on its modular design and how the different components work together to provide flexibility and portability across various systems.

ScummVM is built on a modular architecture, with each subsystem dedicated to specific tasks that contribute to the software's overall functionality. The core of the system is the OSystem, which acts as a bridge between the game engines ScummVM uses and the platform it runs on. This allows ScummVM to adapt old game logic, graphics, and sound to modern operating systems. The backend ensures the platform's portability, making it possible for ScummVM to run on diverse systems, including personal computers and gaming consoles like the Nintendo Switch. The game engines within ScummVM are designed to interpret the original game logic, while the common code provides shared utilities like file compression and debugging that enhance efficiency. Finally, the GUI code offers an interface for users to interact with ScummVM, allowing them to navigate the system and manage gameplay with ease.

At the heart of ScummVM's architecture is the Interpreter Style, a model often used in systems that need to run code designed for other hardware or environments. In ScummVM's case, it acts as a virtual machine that interprets old game files, simulating the environment in which the games were originally meant to run. This makes the Interpreter Style a perfect fit

for the platform's mission. In addition to this core style, other architectural patterns are also at play. Object-Oriented principles are used to manage game objects and user interactions, while Publish-Subscribe is employed to handle in-game events such as sound effects triggered by player actions. ScummVM also incorporates elements of Layered Architecture, where different system components are organized hierarchically, interacting in a structured manner to maintain system stability.
To better understand how ScummVM operates, the report examines a few essential use cases that highlight the interaction between various subsystems:

- Reading and Launching Game Files:
  - The user selects a game from the GUI.
  - The UI Manager sends a request to the Plugin Manager, which identifies the correct game engine from the Game Engine Database.
  - The File I/O Manager reads the game files from local storage and initializes the game using the appropriate game engine.
  - The game engine collaborates with the Graphics and Audio subsystems to render the game and produce sound.
  - Subsystems Involved: UI Manager, Plugin Manager, Game Engine, File I/O Manager, Peripherals Detection, Graphics, Audio.
- Saving and Loading Game Progress:
  - During gameplay, the user selects an option to save or load a game.
  - The UI Manager relays the user's choice to the Game Engine, which requests the File I/O Manager to either save the current game state or retrieve a saved state from local storage.
  - Subsystems Involved: UI Manager, Game Engine, File I/O Manager.
- Handling User Input:
  - The user interacts with the game using input devices like a keyboard or game controller.
  - The Input Handler captures the input and sends it to the Game Engine, which interprets the input and updates the game state accordingly.
  - The game state changes trigger updates to the Graphics Renderer and possibly the Audio Manager.
  - Subsystems Involved: Input Handler, Game Engine, Graphics Renderer, Audio Manager.
- External Interfaces:
  - Graphical User Interface (GUI): Facilitates user interaction with ScummVM, providing controls for starting games, adjusting settings, and accessing system features.
  - Game Data Files: Contains game logic, scripts, audio, and graphics files. These files are processed by ScummVM to render the game and simulate the original gameplay.
  - Save/Load Files: Manages player progress, system preferences, and user settings, allowing games to be saved and resumed at any point.
  - System Files: Logs system errors, performance warnings, and internal data, which are helpful for debugging and optimizing ScummVM.
  - Plugin Data: Game-specific engine plugins are dynamically loaded to interpret the game logic for different titles.

- Performance-Critical Areas:
  - Modularity: ScummVM's modular architecture allows for easy integration of new features and support for new game engines. The decoupling of components ensures that new platforms or features can be added without major refactoring.
- Challenges:
  - While ScummVM supports a wide range of platforms and games, it is currently limited in the number of game engines it supports, with a strong focus on SCUMM-based games.
  - Potential compatibility issues with modern audio and graphical rendering can arise due to the need to preserve the original atmosphere of the games.
- Conclusion:
  - ScummVM's architecture is well-suited for its mission of bringing classic games to modern platforms. Its Interpreter architecture ensures that it can run games originally designed for other hardware, while its modular subsystems make it adaptable and easy to extend. The system's layered and object-oriented patterns add flexibility, allowing it to evolve while preserving the core experience of the games it supports.

3 Architecture

3.1 Components, Subsystems, and Modules

ScummVM operates using a series of subsystems which each handle dedicated tasks. The main subsystems that ScummVM consists of include: OSystem, backends, game engines, common code, and GUI code. The core of these systems is the OSystem which acts as a bridge between the machine ScummVM is running on and the game engines ScummVM uses to run games. Some of the things the OSystem is responsible for include transmitting user inputs, overlaying GUI elements on top of the game graphics, handling the VM's internal clock, managing threads, and outputting the game's audio and graphics. The OSystem allows ScummVM to implement modern computing practices that the original engines did not support such as multithreading and outputting to modern codecs. Through the OSystem API developers and modders can easily interface with and create extensions of the OSystem which is important when creating backends. The backends allow ScummVM to be portable by containing code that ports ScummVM to run on an extensive list of supported systems such as Windows, Mac, and game consoles like the Nintendo Switch. By default, ScummVM contains a Simple DirectMedia Layer backend which enables it to run on almost all common computers whether they are running Windows, Mac or Linux. Backends are implemented as subclasses of the OSystem which allows users to create and share backends for systems that are not officially supported. If a backend sees enough use it is officially incorporated into the main ScummVM codebase. The game engines are what ScummVM uses to run games, these engines interpret the scripts, graphics and audio for their compatible games. ScummVM supports a large variety of game engines and ScummVM's extensive API allows users to add new game engines if they are not supported yet. Game engines are added to ScummVM by implementing them as subclasses of the Engine superclass. The Engine superclass enables subclasses to handle game events, interpret video and audio, access ScummVM's multithreading features, and access system files for saving and loading game states. The common code is an umbrella term for a codebase which provides utility to ScummVM's other subsystems. The common code subsystem provides shared utilities for all other parts of ScummVM. It covers tasks like file compression, debugging, configuration file parsing, and language conversion among others. The common code codebase's functions are updated as required and ScummVM's API makes these utilities easy for developers to use in extending or optimizing engines.

Finally, the GUI subsystem is the user-facing part of ScummVM, providing the graphical interface for selecting games, adjusting settings, and accessing other menus. ScummVM's GUI uses XML for arranging and displaying visual elements which allows for easy modding. The GUI supports loading different user created themes which enables the GUI to evolve alongside modern GUI sensibilities. This modular structure allows ScummVM to support a wide range of platforms, extend functionality, and evolve while remaining user-friendly and developer-friendly.

Component	Description
ScummVM Core Engine	Interprets scripts of original games, reads data, and executes game logic and progression. Calls on necessary components.
Virtual Machine Layer	Executes the original game scripts that control game logic, character control, events, and dialogue.
Graphics	Renders video game objects, backgrounds, and UI. Supports scaling and various resolutions across platforms.
Audio	Interprets different audio formats from original CDs and enables local storage of audio for diskless play.
Input Handler	Translates inputs from mouse, keyboard, or controllers to in-game actions, adapting original game controls for multiple devices.
User Interface	Enables interaction with ScummVM via controls that reflect user input on screen.
Save/Load	Manages game save files independently from original games, offering flexible saving and loading options.
Debugging and Error Handling	Manages errors and debugging during gameplay.
Platform Abstraction Layer	Supports gameplay on various platforms (Windows, Linux, macOS, Android, iOS).
Rendering/Plugin Manager	Supports multiple rendering engines and retrieves required game engines from the database.
File I/O Handler	Reads from local files.
Game Engine Database	Contains game engines to load based on game file requirements.
Peripherals Detection Module	Detects system and I/O devices (OS, keyboard, mouse, touchscreen, etc.) and adjusts input handling.

Table 1: ScummVM Software Architecture Components

3.2 Architectural Style

The architecture style that best characterizes ScummVM and its various parts is the Interpreter Style. The Interpreter Style, commonly seen in virtual machines and scripting languages, is defined as a style for software where the most appropriate language or hardware to execute a program may not be available. The main components of software under the interpreter style include a main state machine component as the execution engine, plus multiple components for the current state of the execution engine, the current program being interpreted, and the state of the program being interpreted. The connectors within an interpreter program are for procedure calls and direct memory access. ScummVM fits this description very well as its primary function is to allow users that have the necessary data files for a game the ability to play on platforms not native to the game. The most appropriate hardware, being the game's native platform, that is needed to execute a program, the game itself, is not available, so ScummVM acts as the solution. The components of ScummVM that would map best to the general components seen in the Interpreter Style is the game engine subsystem as the execution engine and main state machine component, the program being interpreted as the game and data files provided by the user plus the translated game execution files written in Scumm, and the current state of the program being executed being the OSystem.. Some alternative styles that can be seen within ScummVM include Object-Oriented, Pub-Sub, and Layered. The Object-Oriented style is common in game development, where objects can be the current user, items in the game, and visual effects, but ScummVM is primarily a translation service, not simply storing and protecting data. Pub-Sub is a pattern seen within ScummVM, as when certain actions are done within a game, an example being picking up an item, that event is announced and an audio component can subscribe to that specific event and output some audio as a result. However, this is only for the resulting gameplay that ScummVM provides, and is not a great representation of the game execution file translation. The Layered Style is based on hierarchy and each layer interacting with each other as a client and service provider. Layers in ScummVM could be the graphical interface, event management, and audio management, with an example of communication between layers being the event management component being provided a service from the audio management component whenever an event action has been done and needs an audio to be accompanied with it. For other architectural styles, such as repository, client-server, and peer-to-peer, these have very little relation to how ScummVM operates, and therefore aren't notable styles seen in the software.

4 Diagrams

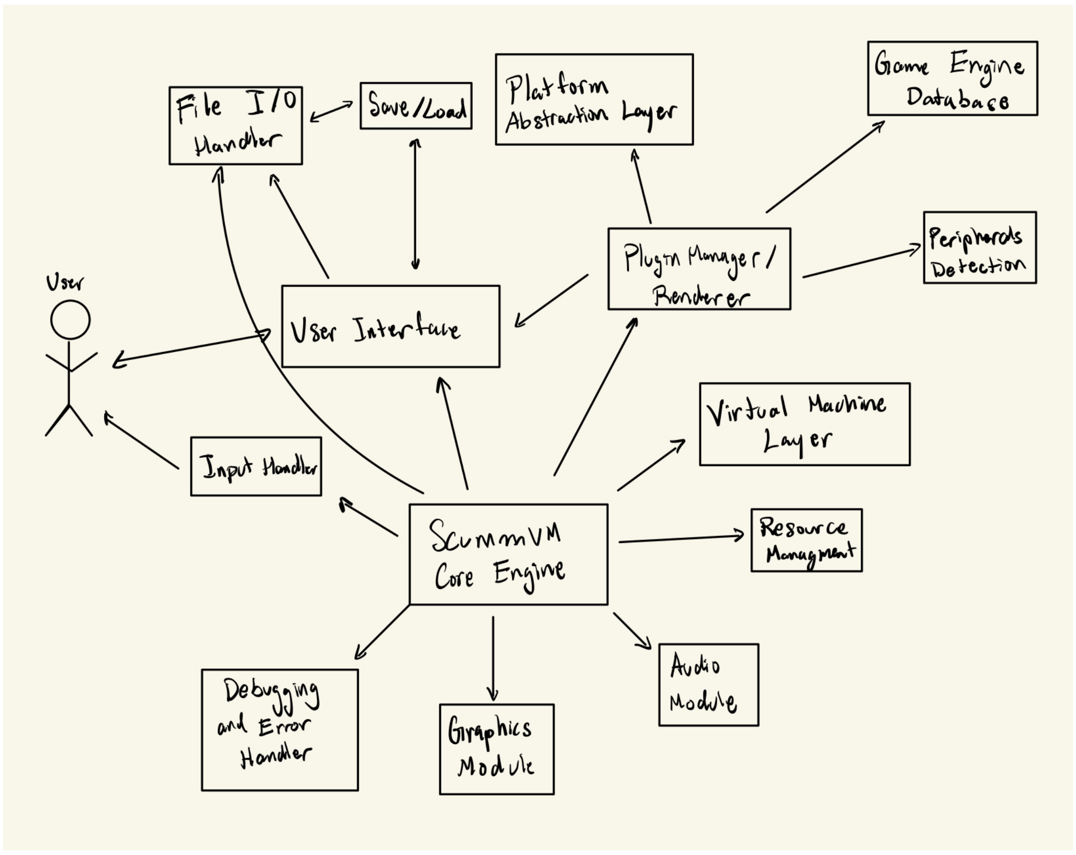

Figure 1: Dependency Diagram

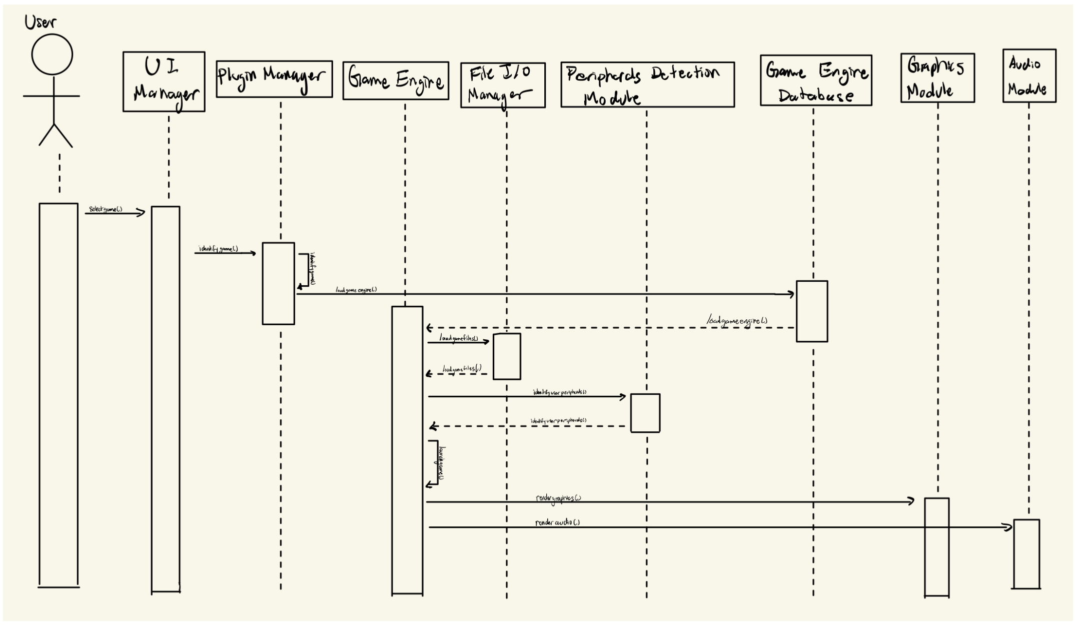

Figure 2: Use Case #1 - Loading a Game

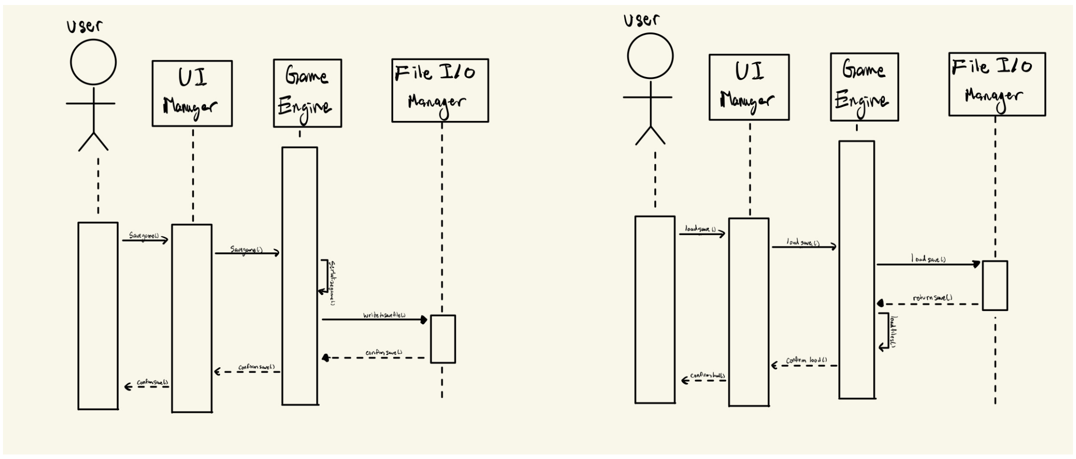

Figure 3: Use Case #2 - Saving(Left) and Loading a Game(Right)

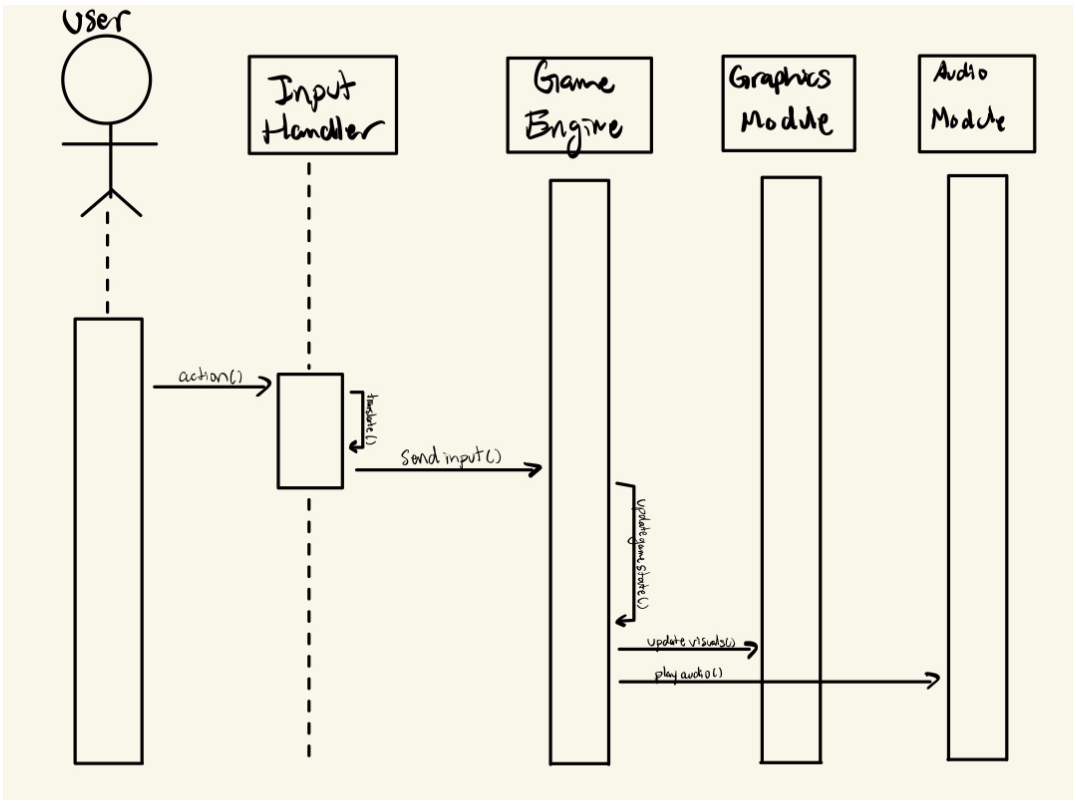

Figure 4: Use Case #3 - Handling User Input

5 External Interfaces

5.1 GUI

The GUI transmits user inputs through input devices into the system. Different examples of this are commands for specific actions or events in-game (interacting with objects, triggering dialogue, etc.), navigation commands which include starting a game and loading save files, and finally settings configuration such as user controls, volume and screen resolution. The GUI is also responsible for transmitting information found within the game from the system to the user. Examples of this are in-game character movement, interactions, dialogue, menus and in game statistics (health bar, score, inventory).

5.2 Game Data Files

The game data files contain external information about the games being run on ScummVM and are transmitted to the system in order to be played. The first key part of this is the game logic and scripts. This information which includes game events, certain triggers and conditions as well as story progression is transmitted into the system so it can be processed. There is also all of the multimedia assets used in game including character sprites, audio, backgrounds and in-game animations. These files also contain data from resource bundles which the system can load dynamically during gameplay.

5.3 Save/Load Files

These files allow in-game save data to be stored and retrieved whenever a game is loaded into the ScummVM engine. Saved data can include player progress, in-game variables as well as timestamped data for exactly when the save file was created. These save files are also responsible for retaining the user set preferences and system settings for that specific game, and having them show up as they were previously set the next time the user loads that specific save file.

5.4 Other System Files

There are other miscellaneous files containing system logs and debugging which transfer error logs, system issues, warnings and details about the internal state of the engine out of the system so that ScummVM developers can assess the performance of their engine and fix any problems through additional development.

5.5 Audio and Video Output

Any audio generated by the system is played back to the user. This includes game music, character dialogue, and sound effects which are transmitted in real time. There are also the graphics output which are transmitted from the system to the user's display device and allows for the video component of the game to function.

5.6 Plugin Data

There is a corresponding game engine plugin that is loaded dynamically for each specific game. This plugin contains information about the game and its logic to the ScummVM Core Engine.

6 Use Cases

6.1 Loading a Game

The user selects a game to load from ScummVM's interface, initiating the process that identifies and loads the appropriate game engine and assets.
Actors:

- User: Initiates the process by selecting a game to load.
- Components Involved:
    - ∗ UI Manager
    - ∗ Plugin Manager
    - ∗ Game Engine
    - ∗ File I/O Manager
    - ∗ Peripherals Detection Module
    - ∗ Game Engine Database
    - ∗ Graphics Module
    - ∗ Audio Module

Preconditions:

- The user has installed ScummVM and has the necessary game data files on their local storage.
- ScummVM supports the game engine required for the selected game.

Flow of Events:

1. User selects a game from the UI.
2. UI Manager sends a request to the Plugin Manager to identify the game.
3. The Plugin Manager checks the file system to validate the presence of necessary game files and determine which game engine is required.
4. The Plugin Manager loads the appropriate game engine from the Game Engine Database.
5. File I/O Manager reads the game files from local storage, including scripts, sprites, and audio data.
6. Peripherals Detection Module identifies the user's input devices (e.g., keyboard, mouse, game controller) and configures them for interaction with the game.
7. Game Engine initializes and loads the game using the provided data files.
8. Game Engine calls on the Graphics Renderer to initialize and render game visuals, and the Audio Manager to initialize sound.

Post Conditions:

- The game is loaded and displayed on the screen, ready for the user to play.

6.2 Saving and Loading Game Progress

The user saves or loads game progress during gameplay, involving interactions with the game engine and file system to preserve or restore game state.
Actors:

- User: Initiates the save or load operation during gameplay.
- Components Involved:
    - ∗ UI Manager
    - ∗ Game Engine
    - ∗ File I/O Manager

Preconditions:

- The game is already loaded into the game engine.
- The user is playing a game and has progressed to a point where they want to save or load their game state.

Flow of Events:

1. User opens the game menu via the UI and selects "Save Game" or "Load Game."
2. UI Manager sends the user's selection to the Game Engine.
3. Game Engine requests the File I/O Manager to either write the current game state (save) or read a previously saved game state (load) from local storage.
4. For saving, File I/O Manager serializes the game state and writes it to a save file.
5. For loading, File I/O Manager retrieves the saved game state and passes it back to the Game Engine.
6. Game Engine either updates the current game state (load) or confirms the save operation (save).

Post Conditions:

- The game is saved, or a previously saved game is successfully loaded, allowing the user to resume from the saved point.

6.3 Handling User Input During Gameplay

The user interacts with the game through various input devices, which are processed to update game visuals and audio in real-time.
Actors:

- User: Interacts with the game using input devices (keyboard, mouse, controller).
- Components Involved:
  - ∗ Input Handler
  - ∗ Game Engine
  - ∗ Graphics Renderer
  - ∗ Audio Manager

Preconditions:

- The user is actively playing a game in ScummVM.

Flow of Events:

1. User performs an action (e.g., presses a key, clicks a mouse).
2. Input Handler translates user action into input the game engine can understand.
3. Input Handler captures the input and sends it to the Game Engine.
4. Game Engine interprets the input and updates the game state accordingly (e.g., moves a character, performs an action).
5. The Game Engine calls on the Graphics Renderer to update the visuals on the screen based on the new game state.
6. If necessary, the Game Engine calls on the Audio Manager to play the corresponding sound effect.

Post Conditions:

- The user's input is reflected in the game, and the graphics and audio respond accordingly.

7 Data Dictionary

8 Naming Conventions

UI: User Interface
Pub-Sub : Publish/Subscribe
GUI: Graphical User Interface

9 Conclusion

The conceptual architecture of ScummVM includes its main components and subsystems, made up of the OSystem, backend, game engine, common code, and GUI code. The OSystem is the core of the software, acting as the bridge between the user's machine and the game engine ScummVM uses. Examples of interactions between these subsystems are the backend using the common code libraries to perform tasks like audio and video decoding, the GUI using the OSystem to handle user input, and the OSystem and backend working together to allow users to run games on almost all common computers. The architectural style that best fits ScummVM is the Interpreter Style, as the software acts as a solution for users to play games even if the most appropriate hardware, being the native game console of these games, is not readily available. Other architectural styles, such as Layered, Publish-Subscribe, and Object-Oriented appear to a lesser extent throughout the ScummVM software. ScummVM is a highly modular and portable software, as it supports multiple platforms and all components of the system work together based on the specific game files provided by the user and the user's platform. Examples of the current architectural limitations with ScummVM include the limitation of game engines the software supports, as ScummVM supports mainly SCUMM scripts, and possible compatibility issues with graphics and audio. However, these issues are partly left intentional due to the developers' intentions of the software maintaining the original game atmosphere for their desired game engines. If ScummVM were to scale into a much bigger game execution script creator, the increase of game engines supported by the software would be the first start. Overall, the conceptual architecture of ScummVM is well suited to solve the problems that it's meant for.

10 Lessons Learned

Some noteworthy lessons that we learned as a group doing this first report were managing workloads and time management between each member of the group. For many of us, this was the first time during our Queen's schooling that we had a class that required us to be working in groups. From this, we had a learning period about how we were going to efficiently work together to finish the assignment, whether that was splitting the report evenly between ourselves, or regularly meeting in-person to work on it together. Being able to quickly and efficiently communicate with each other on the expectations each group member has in order to do effective work for these reports is something we will adapt and improve for upcoming assignments. Our time management skills were definitely tested too, as we underestimated the time commitment each of us were going to have to make in order to complete the report, resulting in a more stressful period of working than what it should've been, and that is definitely something to take note of and adapt for the future.

11 References

https://www.scummvm.org/
https://wiki.scummvm.org/index.php?title=Developer_Central