CISC 322 Assignment 2
ScummVM's Concrete Architecture

Group 5 Authors:
Piero Amendola
Kai Dalen
Sohan Kolla
Stephan Leznikov
Jackson Reid
Harrison Ye

Table of Contents

Abstract

This report delves into the concrete architecture of ScummVM, an open-source software designed to run classic adventure games. Using the Understand tool, we derived and analyzed the concrete architecture, identifying core components such as the ScummVM Engine, GUI, Backends, OSystem API, and Common Code. The study compares the conceptual and concrete architectures, revealing critical discrepancies that emerged from practical implementation challenges. Additionally, a detailed examination of the OSystem API highlights its pivotal role in abstracting platform dependencies. Use cases for loading and playing games further illustrate the system's flexibility. Our findings affirm that ScummVM's architecture employs a modular and interpreter-style approach, ensuring scalability, maintainability, and compatibility with diverse platforms.

Architecture derivation

To derive the architecture for ScummVM, we used the Understand tool by SciTools, which allowed us to map the existing folders to parts of the conceptual architecture. From the ScummVM documentation, we identified five main components: the OSystem API, the backends, the game engines, the common utility code, and the GUI. Initially, we used folder names and their apparent purpose (e.g., the backends/ folder for the backends) to perform a first-stage mapping of the files into these components.

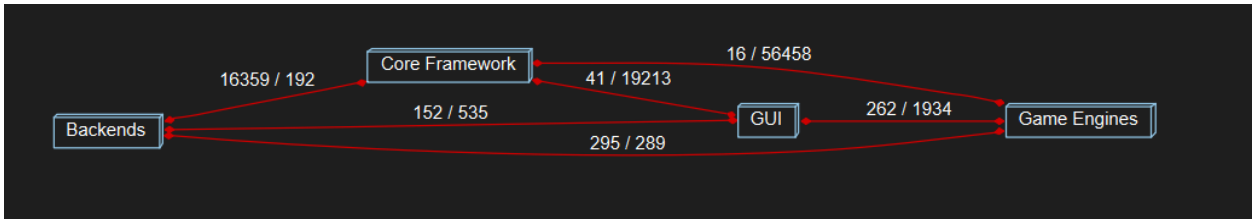
After the first-stage mapping, we generated a dependency graph to visualize the relationships between these components. We noticed several unexpected dependencies between components not anticipated in the conceptual architecture. We followed through the files, their dependencies, and their roles. We adjusted our mapping by manually analyzing the purpose of each folder and reassigning them to different subsystems that better aligned with their functionality if needed.

After this adjustment stage, we generated another dependency graph to verify that the concrete dependencies aligned with the conceptual architecture. While many of the unexpected dependencies were eliminated through this process, some remained due to unforeseen circumstances. For example, some common utility files, such as audio/ or image/ code, were found to interact heavily with both the backends and the game engines, which resulted in unavoidable dependencies.

Finally, to match our conceptual architecture better, we grouped the common utility code and the OSystem API as the "Core Framework". This led us to the following final dependency graphs. Note that the dependencies still vary from the conceptual architecture, some subsystems communicate in the files that don't in the conceptual architecture, some dependencies are mutual when they should be one way, and so on. This will be expanded upon in the dependency changes/reflexion analysis section.

Dependency graph before grouping into Core Framework:

After grouping:



Concrete Architecture

Top-level concrete subsystems

Doc

The docs component is responsible for all the documentation and setup for the project. This component has no dependencies and mainly creates a page for the documentation.

Audio

This subsystem handles all the sound effects, music, and speech generation for all the supported games. This can handle various file types and provide outputs compatible with multiple devices. This component depends on dev tools, math, GUI, common, base, and backends.

Base

This component is a core framework that lays the foundation for ScummVM. The component manages the application life cycle, engine integration, and subsystem coordination. Additionally, the component is responsible for subsystem coordination, file and resource management, configuration, and abstracting the platform to other operating systems. This component depends on audio, backends, common, dists, GUI, graphics, dev tools, and engines.

Backends

Backends are part of the backends layer, making the app portable to various OSs. This component holds domain-specific details needed to allow the platform to be compatible. This component depends on audio, GUI, base, common, dev tools, test, graphics, engines, math, and image.

Dists

Dists is outside the core framework for the engine and provides ScummVM with the resources and configurations needed for building and distributing the system. This module supports each platform's packaging, deployment, and adjustments, including libraries and dependencies. This module depends on common and base, mainly used during distribution and building.

Common

Common is part of the core framework layer, which helps the other components complete their tasks. This component is responsible for handling shared resources, specifically classes and functions such as data structures and computation, and supporting portability by providing operations that have platform-independent values, such as file I/O. Additionally, this component is responsible for resource management, event handling such as mouse inputs, error handling, logging and reporting problems. This component depends on backends, base, audio, GUI, image, graphics, dev tools, test, and engines.

Dev tools

This is outside the runtime system as it assists developers by holding developer-specific tools for development, testing, and maintenance. The end-users do not use this component; instead, it is strictly to support developers. This component depends on the base, standard, backends, engines, graphics, and GUI.

Engines

The engine component is part of the game engine layer, which will heavily depend on the core framework layer for functionality. This component is responsible for holding the engine that powers different groups of games. Each engine is responsible for interpreting game logic, assets, and scripts. This component allows access to a variety of games typically hosted by a variety of different engines. This component depends on dev tools, common, base, ends, GUI, graphics, video, math, image, and audio.

Video

This component is part of the core framework layer, as all the different engines use it for video-related tasks. This component is responsible for video playback and rendering sequences such as pre-rendered animations and cutscenes. Allowing the game to have storytelling aspects such as cutscenes, intros, and highlight moments. It is essential that this module supports

different video formats and accurately plays back sequences of videos. This component depends on engines, graphics, math, common, backends, audio, dev tools, and images.

Graphics

Graphics are part of the core framework layer and depend on providing all engines with rendering functionality. This component renders visuals such as game environments, user interfaces, and animations, ensuring the game correctly displays everything. Graphics depend on dev tools, backends, engines, math, image, GUI, base, and standard.

Math

The math component is part of the core framework layer, which provides utility to other components. This component offers mathematical functions and structures commonly used by the graphics and game physics. This component only depends on backends.

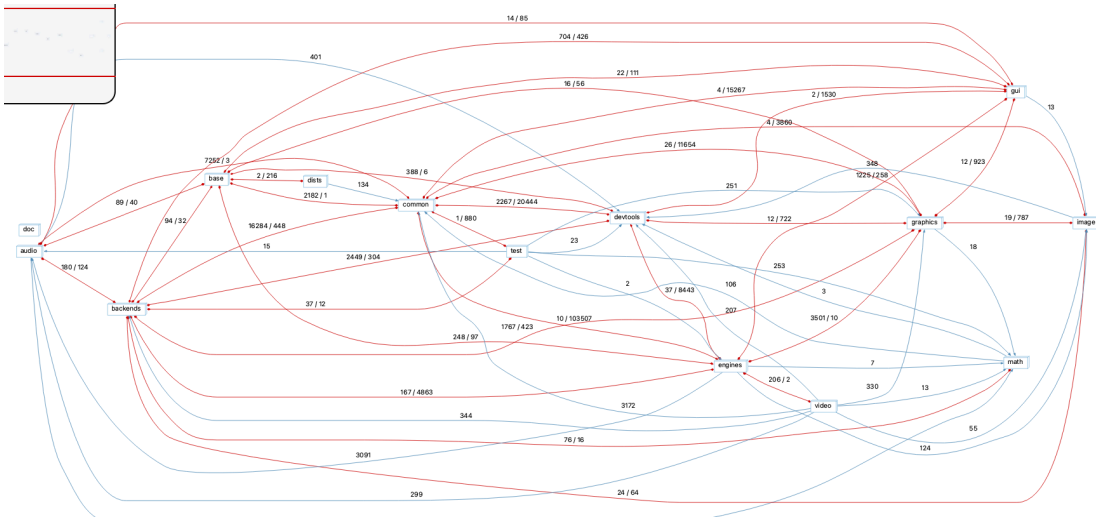
Gui

This component is part of the top layer, enabling human interaction with the system through overlays, settings, and menus. This component bridges the gap between the user and the functionality of the ScummVM, ensuring a seamless user experience. Gui depends on graphics, engines, tools, common, base, backends, audio, and image.

Image

The image component is in the core framework layer, providing utility for other components, specifically for all game engines. This component handles, loads, decodes, and renders image files. This is often used to create the backgrounds, icons, or dynamic graphics, ensuring reproducibility on any OS. This component is dependent on graphics, common, and backends.

Dependency Graph



Main Architectural Styles

ScummVM's architecture exemplifies a combination of layered, modular, and interpreter architectural styles, with a strong emphasis on component-based design. This approach facilitates maintainability, extensibility, and compatibility with a wide variety of platforms and game engines. The layered architecture is particularly evident in the way ScummVM separates platform-specific concerns, core functionality, and game-specific logic, enabling seamless interaction between these elements while maintaining clear boundaries. At the foundation of ScummVM's architecture is the Platform Abstraction Layer, encapsulated in the Backends subsystem. This layer ensures portability by abstracting operating system–specific details, enabling ScummVM to run consistently across numerous platforms. By isolating platform-specific concerns, this layer provides a stable interface for higher layers, shielding the rest of the system from variations in hardware and operating systems.

Above the abstraction layer lies the Core Layer, implemented in the Base subsystem. This layer is responsible for coordinating interactions between subsystems, managing the application lifecycle, and integrating game engines. It provides the scaffolding that supports the modularity of ScummVM, ensuring that game engines and subsystems can function cohesively. The Base subsystem also abstracts platform-specific details for upper layers, further reinforcing the layered design and promoting code reuse. The Game Engine Subsystem, central to ScummVM, illustrates the modular and interpreter styles. It houses over 60 independently developed game engines, each designed as a self-contained module that replicates the behaviour of a specific original game engine. This modular design ensures that new game engines can be added without disrupting existing functionality, promoting scalability. Additionally, each game engine acts as an interpreter, executing game-specific scripts written in languages unique to the original games. These interpreters process commands, manage game states, and control logic, rendering ScummVM highly flexible in supporting a diverse array of games.

In addition to modularity and layering, ScummVM employs a component-based approach to manage functionality such as audio, graphics, and input handling. Subsystems like Audio and Graphics are distinct components with well-defined interfaces, enabling other parts of the system, such as game engines, to interact with them consistently. This separation of concerns simplifies development and makes it easier to address platform-specific or engine-specific needs. By combining layered, modular, interpreter, and component-based architectural styles, ScummVM achieves a balance between flexibility and robustness. The layered architecture ensures portability and separation of concerns, while the modular and component-based designs allow the system to scale efficiently and support a wide array of games and platforms. The interpreter style further enhances ScummVM's adaptability, enabling it to emulate diverse game engines faithfully. Together, these design choices have been instrumental in ScummVM's success as a widely adopted tool for preserving and playing classic games.

Game Engine Subsystem

The game engine subsystem is one of the most critical parts of the ScummVM architecture. It is made up of 64 different game engines that are at the core of ScummVM's ability to emulate classic adventure games. Each game engine is a modular interpreter specifically made to replicate the original engine on new devices. This is what gives ScummVM the modularity to house a vast library of over 300 games, while at the same time maintaining the ability to add new engines to play more classic games.

The game engine subsystem serves as the interface between the GUI and low-level OSystem API and the Backends. It is responsible for interpreting game scripts, managing a game's state, and handling game-specific logic. The modular design of ScummVM and its game engine subsystem allows new engines to be developed without interfering with the functionality of existing ones. This is key for ScummVM's scalability and maintainability, as new game engines are frequently added to the project.

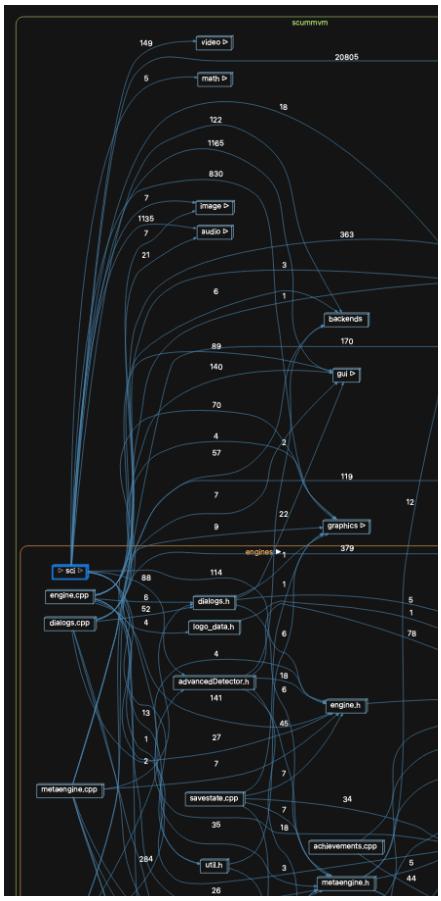
In addition to the responsibilities of the game engine mentioned above, here is the complete list of what the game engine subsystem does:

1. Interpret game scripts: Each engine processes game-specific script languages to execute game logic, interactions, and animations.
2. Managing game state: Engines track the game state, including player position, inventory, completed puzzles, etc.
3. Rendering graphics: Engines convert the game logic into rendering instructions. These are sent to the graphics subsystem using the OSystem API.
4. Playing Audio: Engines send instructions to the audio system, which outputs sound effects and music.
5. Handling Input: Engines process user input and trigger consequential in-game actions.

The game engine includes several subcomponents that work together to deliver functionality. The first is the script interpreter, which processes the game-specific scripting language describing how the game plays. The script interpreter also acts as a state machine that transitions between game states based on user actions or events. The renderer takes the processed game script and converts it into graphical instructions. It then interfaces with the graphics subsystem using the OSystem API. The audio handler sends all sound instructions to the audio subsystem for playback and ensures synchronization between the visual and audio elements. The game engine's input processor maps user input to in-game actions and handles game-specific input logic. The resource manager manages all game assets that it retrieves using the filesystem utility.

The game engine subsystem has several benefits and challenges. The benefits include modularity, modifiability, and testability. It is modular because each game engine is designed as an independent module, allowing for easy addition of new engines. It is modifiable due to the developer's ability to implement new engines without changing core components. It is very testable because each engine can be tested in isolation. The challenges are the complexity of

game logic and performance considerations. Game logic complexity has to do with recreating the behaviour of old game engines. This is a challenge because of the necessary careful interpretation of the original design. Performance is a challenge because old games are designed for much less powerful systems, and there can be issues with smooth gameplay when using new more powerful systems.



Dependency Diagram of the Sci Game Engine

Dependency Changes/Reflexion Analysis

GUI ↔ Backends

In the conceptual architecture, the Graphical User Interface (GUI) and Backends (platform-dependent code) do not have any direct dependencies on each other but in the concrete architecture, there is a mutual dependency between these two components. The GUI depends on

Backends to obtain platform-specific information, this includes accommodating various display sizes and elements, which are necessary for a proper graphics display. On the other hand, various backends in the Backends component rely on GUI components to handle user interactions related to platform-specific configurations (e.g., audio, input, image, video, etc.. devices for the platform used for the current game). From the conceptual to the concrete, the mutual dependency was put in to provide a multi-platform user experience, allowing the GUI and the specific backend to work with whatever platform is necessary to run the current game.

Game Engines ↔ Common Utilities (Under the Core Framework)
The conceptual architecture does include that the Game Engines component depends on the Core Framework/Common Utilities component but the concrete architecture shows that the Core Framework/Common Utilities component also depends on the Game Engines component which is an unexpected mutual dependency instead of a one-way dependency. This is caused due to the case where the common utilities libraries are updated to accommodate for when new games are added to ScummVM and specific needs have to be met to support the games. This causes files in the common utilities libraries to be used for specific game engines rather than general usage.

Backends ↔ OSystem API (Under the Core Framework)
The Backends component is meant to depend on the OSystem API for platform abstraction and hardware interaction to shield the game engines and GUI code from the actual platform the software is running on in the conceptual architecture but the concrete architecture shows a mutual dependency, the OSystem API also depends on the Backends. This dependency was most likely introduced because the OSystem API needs to call platform-specific features during runtime, the Backends component is filled with code that implements the OSystem API for various platforms so it makes sense that the OSystem API might need to call a function for a specific platform.

GUI ↔ Common Utilities (Under the Core Framework)
Again, the conceptual architecture does not plan for any dependency between the Common Utilities component and the GUI but the concrete architecture shows that they have a mutual dependency. The Common Utilities (CU) component (e.g., image, video) depends on the GUI for displaying previews and debugging utilities, while the GUI depends on the CU component for loading resources such as icons or fonts. The mutual dependency was probably a way for developers to improve ease of development and ease of debugging by using the GUI to visualize resources during development.

GUI ↔ Game Engines
Finally, the conceptual architecture assumes that the GUI only interacts with the Common Utilities and OSystem API components (Both are under the Core Framework) without any direct dependency on the Game Engines component but the dependency graph shows that the GUI

occasionally communicates directly with Game Engines for some functionality such as rendering specific game states or providing in-game debug interfaces. The cause for this dependency is a little broader and less specific, it can range from just needing overlays for performance within games (e.g., an FPS counter) or even to help developers debug game-specific features (engine-dependent settings) by visualizing the process.

Use Cases

The following writing will show how ScummVM operates during two use cases. For simplicity, calls to subsystems related to resource management (e.g., loading game assets, audio management) are abstracted, as these operations occur in the background while handling the core game logic and rendering.

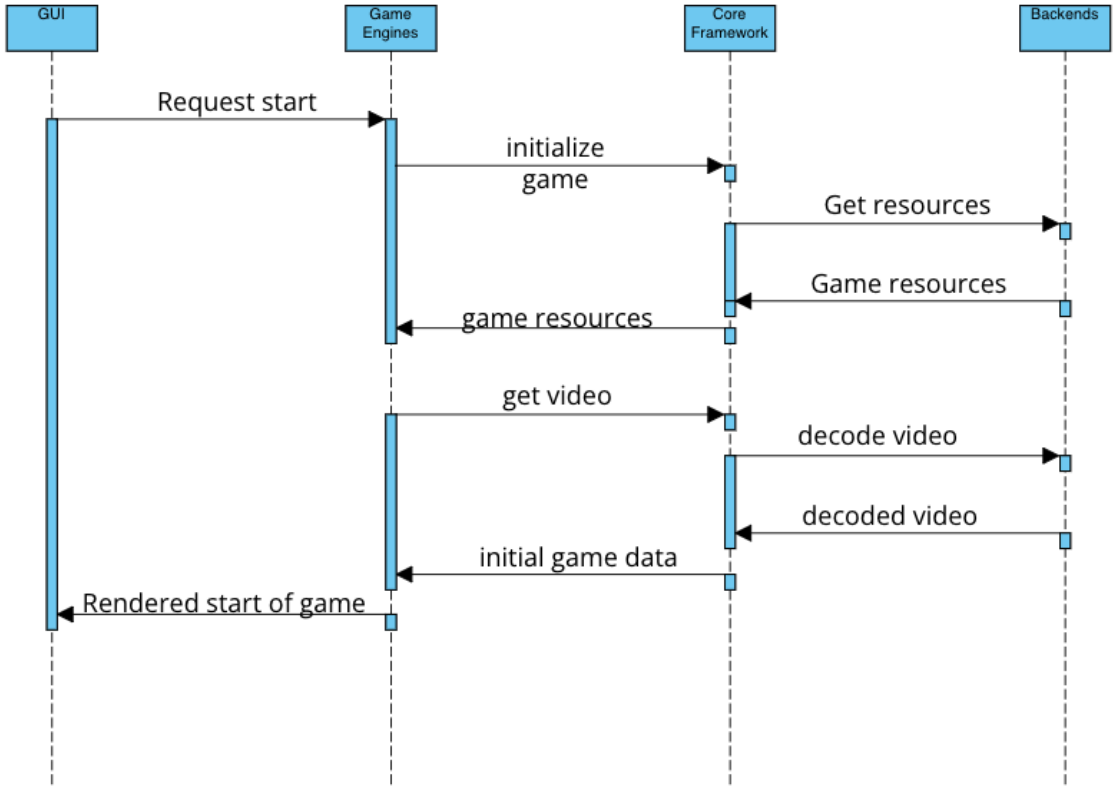
Use Case 1: User selects and initializes a game in ScummVM with video sequence support

This use case begins when the user launches ScummVM and selects a game. The process starts with the user navigating to the game folder and choosing the game's executable. Upon selection, the user interface sends a request to the ScummVM core to initialize the game, including support for video sequences, which are an important part of the gameplay experience.

Once the game is selected, the ScummVM core determines the appropriate engine for the game, which, in this case, we will be talking about the SCI engine. The core then initializes the game by loading the necessary resources, such as graphics, audio, and game scripts. These resources are retrieved asynchronously, meaning that while the game data loads in the background, the system remains responsive to user interactions. Part of the resource loading includes the video sequence files, which are managed by the SEQDecoder. This decoder is responsible for loading and initializing video tracks that will later be used to display video sequences during gameplay. As these sequences are loaded, the game also reads and prepares the palette, which defines the colours used in the video. The correct palette ensures that each frame of the video sequence is displayed with accurate colour mapping, making sure that the visuals are consistent with the game's design.

Once the game resources, including the video sequences, are loaded, the game's main window is rendered. The game logic is initialized, and the player is presented with the first screen, whether it's the main menu or the game's introduction. The game is now ready to be played, and user input is processed, allowing the player to interact with the game and trigger video sequences when needed.

Use Case 1 Diagram



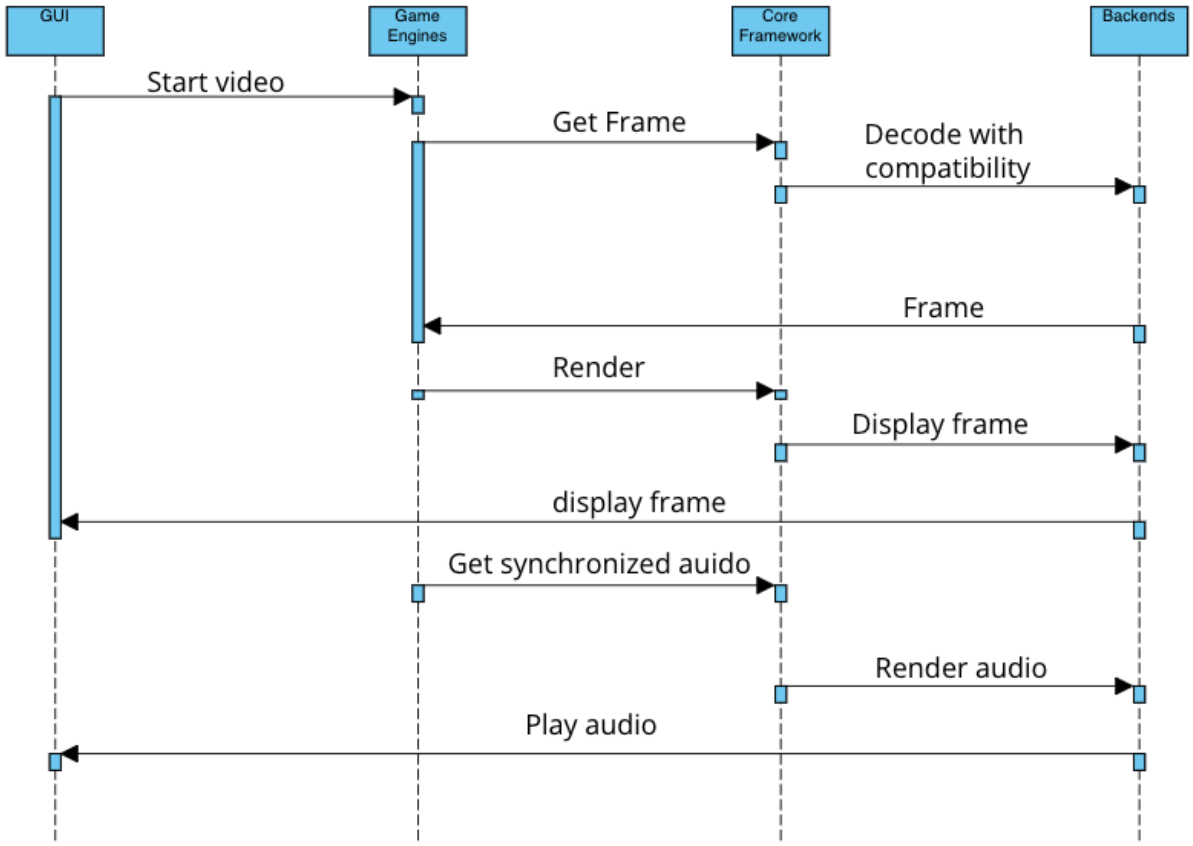
Use Case 2: ScummVM renders video frames and manages audio during gameplay

This use case focuses on the continuous rendering of video frames and the management of synchronized audio during gameplay. Once the game has started, the video sequence begins to play, with ScummVM decoding and displaying frames as they are read from the game files. This happens seamlessly in the background, with the SEQDecoder responsible for handling the video frames and making sure that the visuals are shown in the right order. Each video sequence consists of a series of frames, and the decoder is capable of handling frames, which are compressed to save memory. As the game progresses, the video is decoded and rendered, with the palette updated dynamically to show the changes in the sequence, making sure the video looks correct. This dynamic palette management is crucial for games that feature colour shifts or special visual effects. At the same time, the game's audio is managed and played by the AudioPlayer. Background music, sound effects, and dialogue are streamed in real-time, with the audio synchronized to the video. As each frame of the video is decoded and displayed, the corresponding audio is played to match the action on the screen. This synchronization ensures that the game's visuals and sound remain aligned, providing a seamless experience for the player.

ScummVM's game loop ensures that everything happens in sync. While the video sequence is rendered and audio is played, the system continues to process user input. The video

playback doesn't interrupt the gameplay loop but instead flows alongside it, adding to the immersion. As the video sequence continues, ScummVM's subsystems handle all the necessary tasks: decoding frames, updating the palette, managing audio, and synchronizing everything so that the player experiences smooth, continuous gameplay. After the video sequence is completed, ScummVM returns control to the game, ready to process the next user interaction or event.

Use case 2 diagram



Lessons Learned

Working with ScummVM has been a fun experience, but it also presented some limitations and challenges that required us to adapt our approach and learn important lessons.

While analyzing the ScummVM codebase, we encountered several limitations that presented challenges in our approach. One strong limitation was related to performance issues with the tools we used for code analysis. Specifically, the "Understand" software, which we relied on to analyze dependencies and navigate the code, was slow and resource-intensive. This sluggishness often caused delays, with operations taking minutes, and at times, even causing the tool to crash. As a result, our analysis process was frequently bottlenecked, as multiple team

members were working on the same software and were slowed down by these performance issues. This experience taught us the importance of planning and assessing workflows ahead of time. We realized that simply adding more team members to tasks that were bottlenecked by performance issues would not necessarily improve the situation. Understanding the limitations of the tools at hand and managing expectations around these constraints were important lessons. In future projects, we plan to evaluate the performance of tools earlier in the process and look for ways to optimize workflows to avoid similar bottlenecks.

Additionally, ScummVM's codebase was a challenge due to its size and complexity. With support for multiple game engines and various formats, it was difficult to understand the entire system immediately. At first, we struggled with navigating through the vast amount of code, especially when trying to trace the interaction between the different subsystems such as video rendering, audio management, and resource handling. We learned that a top-down approach, starting with high-level concepts before delving into specific areas, was important for making sense of the project and avoiding getting overwhelmed.

Conclusion

The concrete architecture of ScummVM reflects a successful realization of an interpreter-style system, balancing modularity and scalability to support a wide variety of game engines. Key components such as the ScummVM Engine, GUI, Backends, OSystem API, and Common Code demonstrate the system's ability to abstract platform dependencies while ensuring seamless gameplay experiences. Reflexion analysis provided valuable insights into the alignment and discrepancies between conceptual and concrete architectures, with differences arising from implementation nuances and evolving requirements. The OSystem API, as a core subsystem, was dissected to illustrate its responsibilities, including graphics rendering, input handling, and file management. Despite challenges like complex dependency mapping and codebase navigation, the tools and methodologies applied allowed us to achieve a comprehensive understanding of the architecture. ScummVM's design ensures the preservation of classic games while maintaining adaptability to modern platforms, embodying a robust solution for game emulation and preservation.

Glossary

ScummVM Engine: The central module responsible for interpreting game-specific scripts, enabling compatibility with various platforms.

OSystem API: An abstraction layer facilitating system-level operations such as graphics rendering, input processing, and file access.

Backends: Platform-specific subsystems that implement the OSystem API to support cross-platform functionality.

Common Code: A collection of shared utilities, including functions for handling audio, video, images, and mathematical operations.

GUI (Graphical User Interface): The interface through which users interact with ScummVM, enabling game selection, configuration, and control.

Interpreter Style: A design paradigm where the system dynamically executes instructions, promoting flexibility and cross-platform adaptability.

Reflexion Analysis: A technique for comparing the conceptual and concrete architectures to identify and rationalize discrepancies.

Understand Tool: A software tool used to analyze source code dependencies, aiding in architectural mapping and analysis.

References

ScummVM. (n.d.). Features. Wikipedia. Retrieved October 11, 2024, from https://en.wikipedia.org/wiki/ScummVM#Features

ScummVM. (n.d.). ScummVM. Retrieved October 11, 2024, from https://www.scummvm.org/

ScummVM. (n.d.). ScummVM GitHub repository. GitHub. Retrieved October 11, 2024, from https://github.com/scummvm/scummvm

ScummVM. (n.d.). Developer central. ScummVM Wiki. Retrieved October 11, 2024, from https://wiki.scummvm.org/index.php?title=Developer_Central

ScummVM. (2024). Documentation v2.8.0. Retrieved October 11, 2024, from https://docs.scummvm.org/en/v2.8.0/

ScummVM. (n.d.). Doxygen source code reference. Retrieved October 11, 2024, from https://doxygen.scummvm.org/