1000101: ScummVM Concrete 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

November 18, 2024

Contents

1 Abstract

This report provides an overview of the concrete architecture and a reflexion analysis of ScummVM, a game engine that ports games onto various non-native platforms by interpreting the game files provided by the user into their own scripting language SCUMM. The concrete architecture analysis documents the top-level concrete architecture of the nine major subsystems of ScummVM alongside an in-depth analysis on the interactions these subsystems have with each other. The derivation process for the architecture was done through analysis of source code, visualizations of dependencies using the Understand software, and architectural style considerations. The report contains a reflexion analysis on the high level architecture of ScummVM as well as reflexion analysis of the graphics subsystem. These reflexion analysis map the concrete and conceptual subsystems of ScummVM, recording the convergences, divergences, and absences of these subsystems. The subsystem reflexion analysis looks deeper into the graphics subsystem and how it operates within the ScummVM exosystem, including the libraries utilized and the dependencies it has with other subsystems. Diagrams included in the report consist of the dependency diagram of ScummVM's subsystems alongside sequence diagrams of use cases and explanations of these processes in the software. In conclusion, this report offers a comprehensive view of ScummVM's concrete architecture, subsystem interactions, reflexion analysis, and key functional processes.

2 Introduction and Overview

ScummVM is a game engine that allows users that possess game files of certain classical graphical adventure and role-playing games to play them on platforms that these games were not originally built for. ScummVM will replace the executables for these games through their scripting language SCUMM. An understanding of the software's subsystems and interactions is valuable to be able to further preserve the history of games and could be applied to create softwares that would do the same for games built on other engines. ScummVM follows a modular architecture with subsystems dedicated to specific tasks that each contribute to the overall functionality of the software. These subsystems include the engine, graphics, audio, input handler, GUI, backend, core/common code, and plugins. These subsystems each have functionalities and are made of components that work together. Many subsystems like the engine, audio, and graphics have interactions to send rendering and audio playback requests, which are necessary for the software. The derivation process of the concrete architecture breakdown included an analysis of ScummVM's source code, a visualization of dependencies between these subsystems using the Understand software, and a consideration of the architectural styles that ScummVM follows most. A reflexion analysis was done on the high level architecture of the software, where conceptual subsystems were mapped to the concrete subsystems found during the concrete architecture analysis. These mappings were then noted as whether they were convergences, divergences, or absences. An example of a divergence in the mapping include the graphics module in the conceptual architecture being split into the graphics and video subsystems in the conceptual architecture, with the divergence improving the modularity of the system. The impact of these divergences and absences are further developed, where benefits like increased modularity of the software and drawbacks like increased complexity are discussed. A reflexion analysis was also done on the graphics subsystem of ScummVM. The responsibilities of the subsystem are described alongside the libraries and utilities used in those processes. Dependencies of the subsystem are further discussed with examples of the interactions the subsystem has with other subsystems in the software. A reflection is done on our previous description of the graphics subsystem from the conceptual architecture report with a better explanation of the subsystem's functionality and responsibilities within the software as a result of the reflexion analysis. Diagrams created for this report include a dependency diagram of the subsystems of ScummVM, as well as sequence diagrams linked to specific use cases outlined in the report. The use cases of

rendering a scene during gameplay and handling audio during gameplay are further explored with sequence diagrams and an outlining of the user, components involved, preconditions, flow of events, and postconditions. Overall, the report covers the concrete architecture analysis of ScummVM, reflexion analysis for the high level architecture and graphics subsystem, and relevant use cases and sequence diagrams for functions within the software.

3 Top-Level Concrete Subsystems

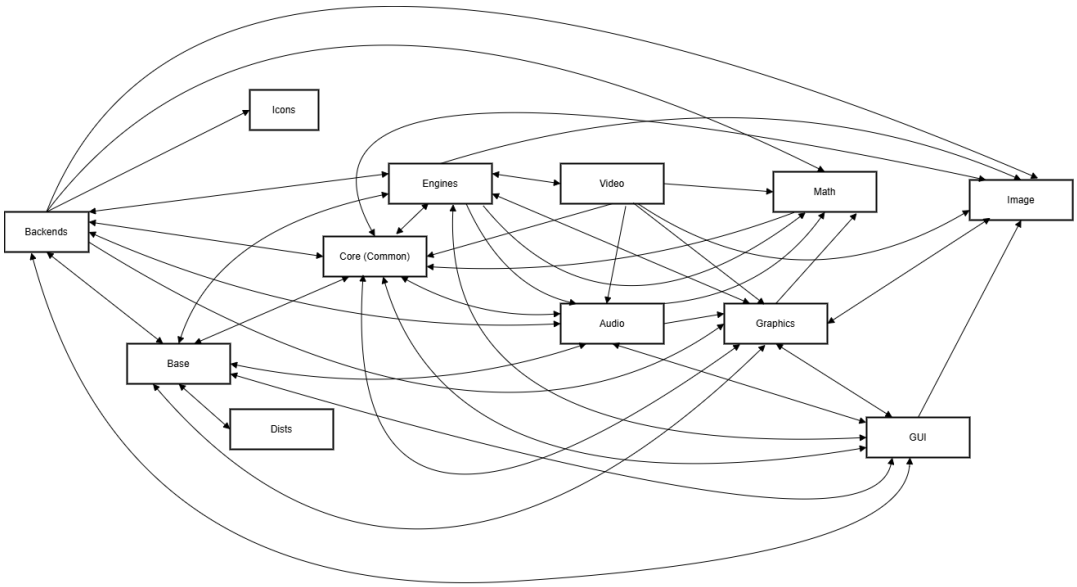

Figure 1: Concrete Architecture for ScummVM

3.1 Engines Subsystem

Functionality: Implements game-specific logic and behaviors, tailored to individual games or game families (e.g., SCI, Kyra). Ensures accurate execution of supported games.
Interaction: Interfaces with Graphics, Audio, Video, Image, and Core to execute logic and facilitate gameplay.
Components:

- Game logic modules (e.g., `engine.cpp`, `metaengine.cpp`).
- Handlers for save states, dialogs, and scripting logic.

3.2 Graphics Subsystem

Functionality: Handles rendering of sprites, backgrounds, videos, and user interfaces. Ensures compatibility across platforms while supporting high-quality visuals.
Interaction: Works with Engines for rendering gameplay visuals, Video for multimedia playback, Image for asset management, and GUI for interface overlays.
Components:

- Core rendering and graphical transformation modules.
- Interfaces for managing overlays and visual effects in the GUI.

3.3 Audio Subsystem

Functionality: Manages decoding, processing, and playback of sounds, music, and dialogues. Ensures synchronization between audio and video elements for a seamless user experience.
Interaction: Processes audio playback requests from the Engines, Video, and GUI subsystems to deliver immersive sound effects, background music, and dialogue.
Components:

- Decoding and playback modules for formats such as wave and MIDI.
- Synchronization utilities to maintain precise coordination between audio and visual elements.

3.4 GUI Subsystem

Functionality: Provides the user interface, including menus, game selection screens, and configuration options. Integrates input-handling functionality to process user interactions.
Interaction: Relays user commands to Engines and Backends. Relying on Graphics and Image, it renders visual interface elements.
Components:

- Menu components and dialogs for user interaction.
- A theme engine for customizable visual styles.
- Input event handlers for processing user actions.

3.5 Video Subsystem

Functionality: Responsible for multimedia content playback, such as cutscenes and animations. Ensures smooth integration with audio and graphics.
Interaction: Coordinates with Graphics for rendering video frames and Audio for synchronized playback.
Components:

- Video decoding libraries supporting multiple codecs.
- Synchronization modules for real-time multimedia playback.

3.6 Math Subsystem

Functionality: Provides utilities for complex computations, including transformations, physics, and other game logic calculations.
Interaction: Supports Graphics for rendering transformations, Engines for physics computations, and Core for general mathematical operations.
Components:

- Libraries for matrix operations, vector transformations, and geometry calculations.

3.7 Image Subsystem

Functionality: Focuses on managing image assets, including textures, sprites, and other visual resources. Enhances modularity and facilitates the reuse of image-related operations.
Interaction: Supports Graphics for rendering, GUI for displaying visual assets, and Core for centralized resource handling.
Components:

- Libraries for image decoding and manipulation (e.g., PNG, BMP).
- Utilities for scaling, transforming, and color management.

3.8 Backends Subsystem

Functionality: Abstracts platform-specific operations, enabling ScummVM to run seamlessly on diverse systems like Windows, macOS, Linux, and Android.
Interaction: Provides platform-specific APIs to the Graphics, Audio, and GUI subsystems for hardware interaction.
Components:

- SDL integrations and device drivers for hardware abstraction.

3.9 Core (Common) Subsystem

Functionality: Centralizes shared utilities like memory management, file I/O, debugging, and resource handling. It also manages essential plugin functionality for dynamic engine integration.
Interaction: Serves as the backbone, providing foundational support to all other subsystems, ensuring consistent data handling and operations.
Components:

- Debugging tools, threading utilities, and configuration managers.
- Libraries for file I/O, resource management, and mathematical computations.

3.10 Base Subsystem

Functionality: Manages plugin-related functionality and provides lower-level file I/O support. Ensures efficient plugin handling and acts as a bridge for shared operations across subsystems.
Interaction: Extends file I/O functionalities in collaboration with the Core subsystem and supports dynamic plugin integration for rendering and gameplay.
Components:

- Utilities for plugin management and dynamic loading.
- File I/O handlers and modules for rendering integration.

3.11 Dists Subsystem

Functionality: Manages platform-specific configurations, such as installers and libraries. This subsystem enables smooth adaptation of ScummVM for different operating systems.
Interaction: Collaborates with the Base subsystem to adapt ScummVM for distribution across diverse platforms.
Components:

- Tools for generating platform-specific builds and configurations.

3.12 Icons Subsystem

Functionality: Manages graphical icons used across the GUI and gameplay, including toolbar icons, status indicators, and in-game symbols. Enhances user experience by providing visual cues and interaction elements.
Interaction: Collaborates with the GUI subsystem to render interface icons and with the Graphics subsystem for scaling and transformation. Works with the Core subsystem for resource management and loading.
Components:

- Icon rendering utilities for displaying scalable vector and raster icons.
- Asset loaders for handling icon files in various formats (e.g., PNG, SVG).
- Transformation tools for resizing, rotation, and color adjustments.

3.13 Subsystem Interactions

- Engines ↔ Core: Engines use Core for debugging, mathematical operations, and resource handling.
- Engines ↔ Graphics, Audio, Video, Image, Math: Engines communicate with these subsystems for rendering, sound, multimedia, and computational tasks.
- Core ↔ Base: Extends file I/O functionalities provided by Base and integrates plugin management.
- Graphics ↔ GUI, Image, Math: Graphics processes visual assets, interface rendering, and mathematical transformations.
- GUI ↔ Backends, Core: Relies on Backends for device compatibility and Core for configuration management.
- Video ↔ Graphics, Audio: Coordinates with Graphics for video rendering and Audio for synchronized playback.
- Math ↔ Core: Core incorporates Math libraries for computational tasks.
- Dists ↔ Base: Manages platform-specific distributions through Base.

3.14 Derivation Process

Source Code Analysis

Subsystems were identified by analyzing the directory structure (e.g., engines, graphics, audio, video, math, image, base) and key source files (e.g., engine.cpp, graphics.cpp). Additional components, such as Math and Image, were included to reflect their utility in computations and resource handling.

Dependency Visualization

The dependency graph provided clear insights into the interactions between subsystems. For instance, Engines heavily rely on Core, Graphics, and Audio, while Core serves as the backbone for all other subsystems.

A dependency graph highlighted key interactions. For instance:

- Engines: Relies heavily on Core, Graphics, and Audio.
- Core: Serves as the backbone, supporting all subsystems.
- Math and Image: Provide utility support to Graphics and Engines.

Architectural Style

A modular architecture ensures clear boundaries and responsibilities for each subsystem. The client-server style is reflected in the Core subsystem, which mediates interactions between Engines and Backends.

4 Reflection Analysis of the High Level Architecture

Mapping Table: Conceptual to Concrete Architecture

Conceptual Subsystem	Concrete Module	Mapping
ScummVM Core Engine	Base	Convergence
Virtual Machine Layer	Engines	Convergence
Graphics	Graphics, Video	Divergence
Audio	Audio, Video	Divergence
Input Handler	GUI	Divergence
User Interface	GUI	Convergence
Save/Load	Common	Divergence
Debugging and Error Handling	Common	Divergence
Platform Abstraction Layer	Backends	Convergence
Rendering/Plugin Manager	Rendering, Base	Divergence
File I/O Handler	Common, Base	Divergence
Game Database Engine	Common	Divergence
Peripherals Detection	Backends	Convergence
—	Icons	Absence
—	Image	Absence
—	Dists	Absence
—	Math	Absence

Divergences

Graphics ↔ Graphics, Video
The Graphics module in the conceptual architecture maps to both the Graphics module (for object rendering) and to Video (for video playback). This split improves the modularity of the system, allowing real-time rendering as well as video playback to be simultaneously optimized. This is in line with ScummVM's mission of supporting a variety of games on their engine.

Audio ↔ Audio, Video
There are some audio functionalities such as synchronization that should belong in the Video module. Video playback usually requires highly accurate synchronization with audio, so having these functions in a separate module allows for smoother playback and can reduce latency.

Input Handler ↔ GUI
The Input Handler module in the conceptual architecture is integrated directly into the GUI instead of being a separate module within the concrete architecture. This is done because ScummVM has relatively simple input requirements, meaning consolidating this module into GUI will reduce the system's complexity without adding problems to it.

Save/Load ↔ Common
The Save/Load module is merged into the Common module of the concrete architecture. This is done because the ScummVM design has a preference for centralized utility modules. This means that instead of having various utilities as their own modules, they would all be placed under one.

Debugging and Error Handling ↔ Common
Debugging and Error Handling is another utility function that can be merged into the Common module. This merge supports consistent logging and error reporting across all of ScummVM's subsystems.

Rendering/Plugin Manager ↔ Rendering, Base
This conceptual module is split into two modules: Rendering, and Base (stores all plugin-related functions). This is done so that the rendering pipelines in the system remain efficient while still allowing plugins to be dynamically managed without any interference.

File I/O Handler ↔ Common, Base
I/O functionalities span across both the Common and Base modules of the concrete architecture. The overlap in these is caused by the historical code evolution of the system. File I/O functions were originally implemented in Base but were later extended into the Common module to add functionality to the system.

Game Database Engine ↔ Common
The Game Database Engine is another module integrated into Common. This is done because the engine's modular game support highly benefits from having centralized access to game metadata and configurations. This is much more easily done when this information is available in the Common module.

Absences

No Mapping for Icons in the Conceptual Architecture
The conceptual architecture focuses on core functional subsystems such as audio and input handling. Icons are more of a necessity that becomes apparent when handling simple tasks such as status indicators, toolbar icons, and others. Adding this module in the concrete architecture improves the maintainability of the system by isolating these specific resources but was not a major part of the conceptual architecture, which is why it was overlooked.

No Mapping for Dists in the Conceptual Architecture
This module is absent in the conceptual architecture since it is not part of the core functionalities of the system. However, as we can see, it is much needed in the concrete design. Dists handles platform-specific configurations such as installers, libraries, and package management. These are all essential tasks that may not have been considered when designing the conceptual architecture.

No Mapping for Image in the Conceptual Architecture
In this case, it was likely assumed that Image would be handled via the Graphics or Core Engine subsystems. Separating it into its own subsystem, however, enhances reusability and modularity of image handling. This is essential since image handling is important in multiple areas of this game engine, such as game textures, assets, and UI.

No Mapping for Math in the Conceptual Architecture
Mathematical operations are foundational for all computational tasks in the system but do not really have a place in the conceptual architecture. These operations were likely assumed to already exist in the various subsystems where they are used. Math is centralized in the concrete architecture, which improves consistency and reusability of these mathematical operations across the system.

Impact of Divergences and Absences

Divergences such as splitting Graphics into Graphics and Video improve the modularity of the system. Functions such as video playback and rendering being separated allow developers to further enhance each function independently. Centralizing utilities into Math and

Common has a similar effect because of the consistency that is added across engines when performing calculations or other functions.

There is also practical implementation through the absences seen in the concrete architecture. Having a dedicated Icons module reflects real-world UI requirements that were not seen in the conceptual architecture. This can also be seen through Dists, since this module addresses distribution and packaging, which is critical for the system. Even though this is not a core function, it is key in being able to have ScummVM on multiple operating systems.

One downside of the divergences found is that the complexity of the system could potentially increase in the concrete architecture. This is because of overlapping responsibilities seen in, for example, File I/O, which is found both in Common and Base. This can make it more challenging for developers to identify where issues in the code logic reside, which makes debugging or updating the code much more strenuous.

Conclusion

Many of the differences seen between ScummVM's concrete and conceptual architecture are because of the practical implementation needs that are overlooked in the conceptual design of the system. The divergences and absences improve modularity and real-life functionality while also adding some slight complexity to the system.

5 Subsystem Explanation and Architecture Reflexion

This subsystem analysis will be focused on the graphics subsystem. This subsystem is responsible for handling image rendering, containing graphics libraries, code for basic image operations, fonts and text related functions, platform specific functions, and other miscellaneous necessary code. ScummVM primarily utilizes the OpenGL graphics library which enables ScummVM to send needed commands to the system's graphics processing unit to handle 2D and 3D vector rendering. ScummVM also contains the TinyGL graphics library which is a lightweight, portable software based graphics library mimicking several key functions of OpenGL. This library is used for rendering simple graphics but is not used for rendering complicated scenes as software based rendering is slower than it's hardware counterpart. Aside from graphics libraries, the graphics subsystem also handles a few GUI elements, namely the mouse cursor and fonts. The graphics subsystem can modify the size and look of the system cursor which can enable it to look different when in certain menus or when playing games. It also contains all the fonts that ScummVM uses as well as an API that displays text and manages fonts. The bulk of the code in the graphics subsystem is dedicated to performing basic image operations. This includes code for scaling and transforming images, swapping between colour spaces, pixel formatting, and managing colour palettes. These basic functions are necessary additions to the subsystem and are used very often. Finally, the graphics subsystem contains mac-specific code for managing the GUI. Mac systems require a proprietary implementation of basic GUI functions such as displaying windows, managing the cursor and fonts, and displaying text. ScummVM is designed to be able to run on as many system configurations as possible so including mac-specific code is a necessity, as the mac platform is very popular.
The graphics subsystem is one of the most important subsystems of ScummVM and as such it depends on and is depended on by many other subsystems. The subsystem depends on the base, backends, common, engines, image, GUI and math subsystems for performing many of its functions and in turn is depended on by almost all of them as well as the audio and video subsystems. The graphics subsystem depends on the base subsystem for reading configurations and plugins as well as using functions and variables from the main.cpp file which is the

central file of ScummVM. In turn, the base subsystem depends on the graphics subsystem for reading version info and info about plugin compatibility. The graphics subsystem depends on the backends subsystem for reading info about the current system as it needs to know what kind of system ScummVM is running on to properly execute rendering functions. Like the main subsystem, the backends subsystem does not depend much on the graphics subsystem, only depending on it for certain platform specific GUI functions. The graphics subsystem depends on the engines subsystem for obtaining info about the engine, but due to using universal graphics libraries it does not depend on the engine much. However, the engines subsystem depends heavily on the graphics subsystem with the engine needing to use the graphics subsystem for displaying necessary windows and graphics. The image and graphics subsystems are expectedly very intertwined with the graphics subsystem depending on the image subsystem for info and functions related to .png and .bmp files which are common image formats. In turn, the image subsystem depends on the graphics subsystem for rendering images. The graphics subsystem only depends on the GUI subsystem for the ThemeEngine which dictates how windows and images are layered and displayed. However, the GUI subsystem depends heavily on the graphics subsystem as it needs the graphics subsystem for displaying the GUI itself. The last subsystem that the graphics subsystem depends on is the math subsystem, this is a one way dependence. Rendering vectors involves lots of complicated math so the graphics subsystem depends on the math subsystem for functions related to matrix operations, vector operations, and degree to radian conversions. This marks the end of the graphics subsystem's dependencies, with the following subsystems having a one way dependency on the graphics subsystem. The audio subsystem depends on the graphics subsystem for reading info related to fonts, pixel formatting, and surfaces. Finally, the video subsystem depends on the graphics subsystem for displaying videos in the wide array of available codecs.

In our initial report, we briefly mentioned the graphics subsystem as a component with the following description: "Renders video game objects, backgrounds, and UI. Supports scaling and various resolutions across platforms." This description accurately describes several of the key functions and interactions that have been outlined for the graphics subsystem. The graphics subsystem does render game objects and backgrounds and the GUI. It also supports scaling and various resolutions. However, this explanation neglects to mention several of the key functions and dependencies of the graphics subsystem. The subsystem also handles operations relating to text display, and drawing windows, which are very important functions that were not mentioned. These functions were likely not mentioned because we had incorrectly assumed that the GUI subsystem would have the capability to display the windows and fonts itself. The current implementation is better than our assumption as compartmentalizing all the rendering into its own subsystem allows for easier understanding of the system and in turn easier debugging. We had also highlighted the interpreter architectural style as the best fit for ScummVM. In this style, the graphics subsystem would be located inside the execution engine as it remains static regardless of the game or game engine. In this way, the interpreter architectural style properly accommodates the graphics subsystem and it's functions. Finally, our dependency diagram only contained a single one way dependence from the core engine to the graphics subsystem. This diagram was based off of the assumption that since graphics rendering is generally done with universal libraries, the graphics subsystem would not have dependencies of its own and would only be depended on by the core engine when a graphic is needed. This concept was quite far from how the subsystem is implemented in actuality. The graphics subsystem has a lot of dependencies of its own with it needed to import functions and variables for math, info about the engine, and info about the platform. It also is depended on by many subsystems which call it to handle their graphics. As stated, we had assumed that it would all be sent to the core engine beforehand so our diagram was lacking the dependencies. Overall, our concepts for how the graphics subsystem worked were quite far detached from how it is implemented in reality.

6 Diagrams

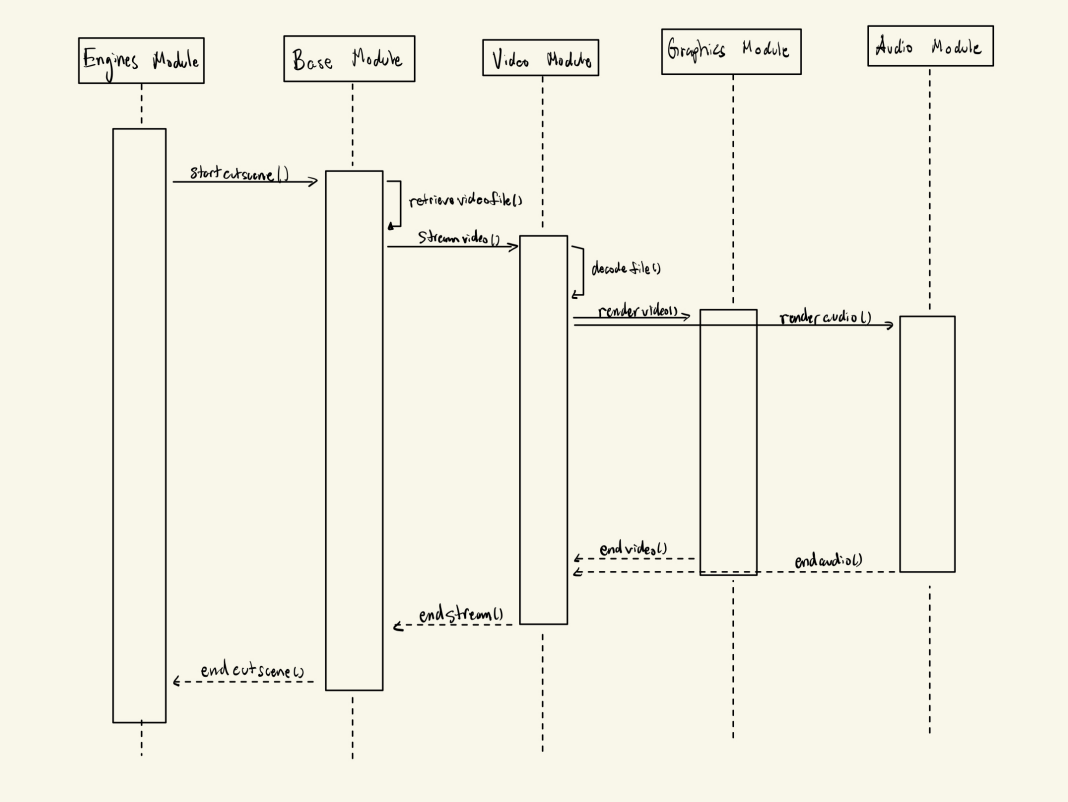

Figure 2: Use Case #1 - Playing a Cut-scene

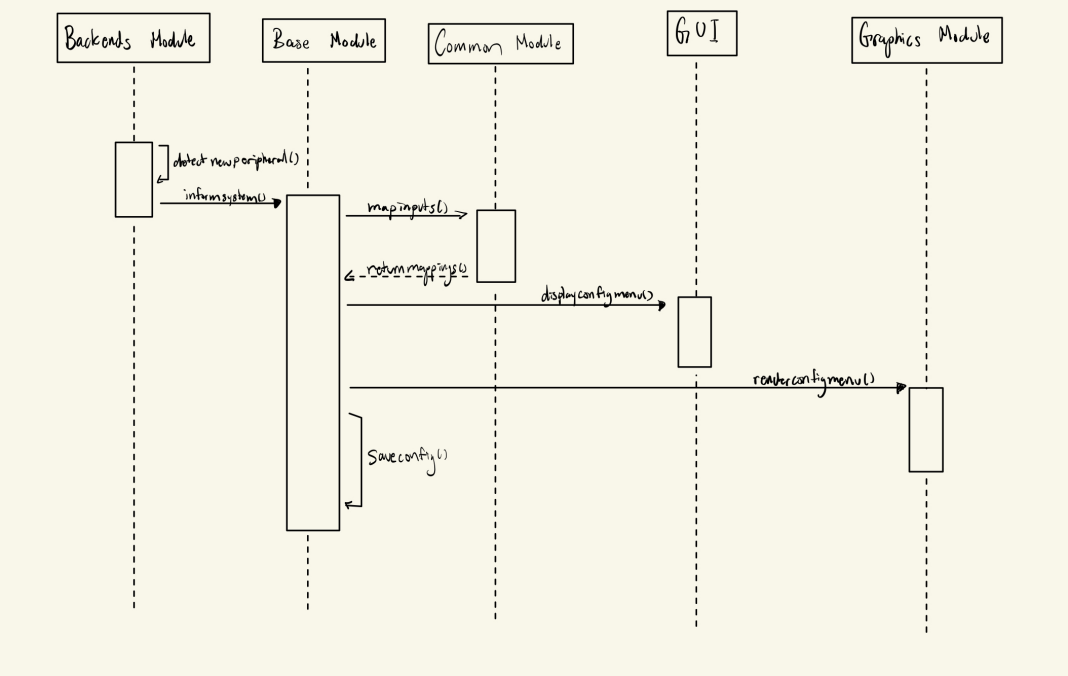

Figure 3: Use Case #2 - Detecting and Configuring Peripherals

7 Use Cases

7.1 Playing a Cut-Scene

The system sends a command to start a cut-scene.
Actors:

- Components Involved:
    * Game Engine
    * Graphics Module
    * Base Module
    * Video Module
    * Audio Module

Preconditions:

- The game is running, and the game state has been updated based on previous actions (e.g., movement, interactions).
- A cut-scene is starting.

Flow of Events:

1. The game engine sends a request to the base module to start a cut-scene.
2. The base module retrieves the video file and sends it to the video module.
3. The video module decodes the video file and simultaneously sends the frames of the video as well as the audio to the graphics module and the audio module respectively.
4. The cut-scene ends when all modules have confirmed that their tasks are complete.

Post Conditions:

- The scene is rendered on the screen and displayed to the user.

7.2 Detecting and Configuring Peripherals

The user connects a new peripheral device (e.g. a controller, mouse, trackpad, etc.).
Actors:

- Components Involved:
    * Backends Module
    * Base Module
    * Common Module
    * GUI
    * Graphics Module

Preconditions:

- The backends module detects a new peripheral connected to the user's device.
- A game that requires inputs is currently loaded into ScummVM.

Flow of Events:

1. The backends module detects a new peripheral connected to the user's device.

2. The backends module informs the base module of the new device and its properties.
3. The base module asks the common module to map the inputs of the new device to controls within the game.
4. The new configuration is displayed on the GUI.
5. The configuration menu is rendered in game as well.
6. The base module then saves the new configuration.

Post Conditions:

- The new peripheral is successfully configured to the controls of the currently loaded game.

8 Naming Conventions

UI: User Interface
Pub-Sub : Publish/Subscribe
GUI: Graphical User Interface
API: Application Programming Interface
I/O: Input/Output

9 Conclusion

In conclusion, this report was made to analyze the concrete architecture of ScummVM through its subsystems and interactions between them as well as providing a reflexion analysis to map the conceptual and architectural subsystems. Important findings for the top level concrete subsystems include the descriptions of each subsystem's functionality and components used, as well as the many interactions that each subsystem has with each other. Within the reflexion analysis of the top level architecture, the benefits and drawbacks of the many divergences and absences were discussed, an example being the divergences of the graphics subsystem were actually beneficial to the software as it resulted in increased modularity. The significance that the reflexion analysis of the top level and subsystem provides is a look into the process of building and designing software, where certain subsystems that prove necessary in the source code were either implemented in a different manner from the conceptual subsystem or in some cases were not even thought of on a conceptual level, which shows just how much change can go on through the development of software. The analysis done in this report can be utilized further down the road for other software through the insight gained from subsystem interaction understanding, where this knowledge could then be applied into other game engine softwares that share similarities to ScummVM.

10 Lessons Learned

A challenge that our group faced during the process of creating this report was becoming familiar with new software introduced to us. The Understand software was crucial for the visualization of dependencies of subsystems in the software, so there was a learning curve to being able to effectively navigate the software and make sure that we could get the information we needed from it. Coordination within our group on getting sections done felt very improved from the conceptual architecture report to now, which shows growth between all of us in trust and leadership, as we were much more effective in time management and communicating with each other on what needs to get done and by who. Our biggest takeaways from this report were to prioritize getting familiar with new software when needed, as well as believing more in each other to take initiative and show effort for the group as we saw a great improvement in that from previous and hope to continue that into the future.

11 References

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