CISC 322
A2: Concrete Architecture report
November 18, 2024

Concrete Architecture of ScummVM

Group4:
Lorem Ipsum

Authors:

Dima Zaichenko (22bh17@queensu.ca)

Joshua Yarussky (21jy71@queensu.ca)

Roman Rodchenkov (21rr69@queensu.ca)

Mohammed Sarhat(19ms16@queensu.ca)

Charles Letellier(21cl106@queensu.ca)

Patrick Favret(21phf1@queensu.ca)

Table of Contents

Abstract

This report examines ScummVM's layered and interpreter style architecture, focusing on its modularity, adaptability, and subsystem interactions. Using tools like Understand along with examining source code and documentation, we performed a reflexion analysis where we derived ScummVM's concrete architecture and compared it with the initial conceptual model established earlier. The analysis uncovered previously unrecognized top-level components, such as the Graphics subsystem, highlighting its critical role in rendering and enhancing visual output. Furthermore, numerous unexpected dependencies between subsystems were identified and their rationality examined, providing a more accurate representation of ScummVM's architecture. Key insights include the effectiveness of modularity in supporting multiple game engines and platforms and the necessity of iterative refinement in architectural design. The study emphasizes aligning conceptual and concrete architectures through a careful reflexion analysis to highlight unexpected divergences, absences and converges in order to get a highly detailed visual of the scummVM system.

Introduction

The purpose of this report is to analyze the architecture of ScummVM, an open-source project that enables users to play classic adventure games on modern platforms. This analysis involves looking at ScummVM's components, dependencies, and design patterns to gain insight into its structure.

ScummVM, which stands for "Script Creation Utility for Maniac Mansion Virtual Machine," is designed to reimplement original game engines in C++. By doing so, it ensures compatibility with a wide array of classic games developed by companies like LucasArts, Sierra, and others. With its support for multiple game engines and platforms, ScummVM showcases an exceptional balance of adaptability and performance, making it a great way to enjoy retro games on modern-day hardware.

Analyzing ScummVM's architecture offers valuable lessons in managing complexity while supporting a diverse range of features. By examining its subsystems and dependencies, this report highlights the project's ability to remain flexible and scalable, even as new engines and functionalities are introduced.

The primary objectives of this report are as follows:

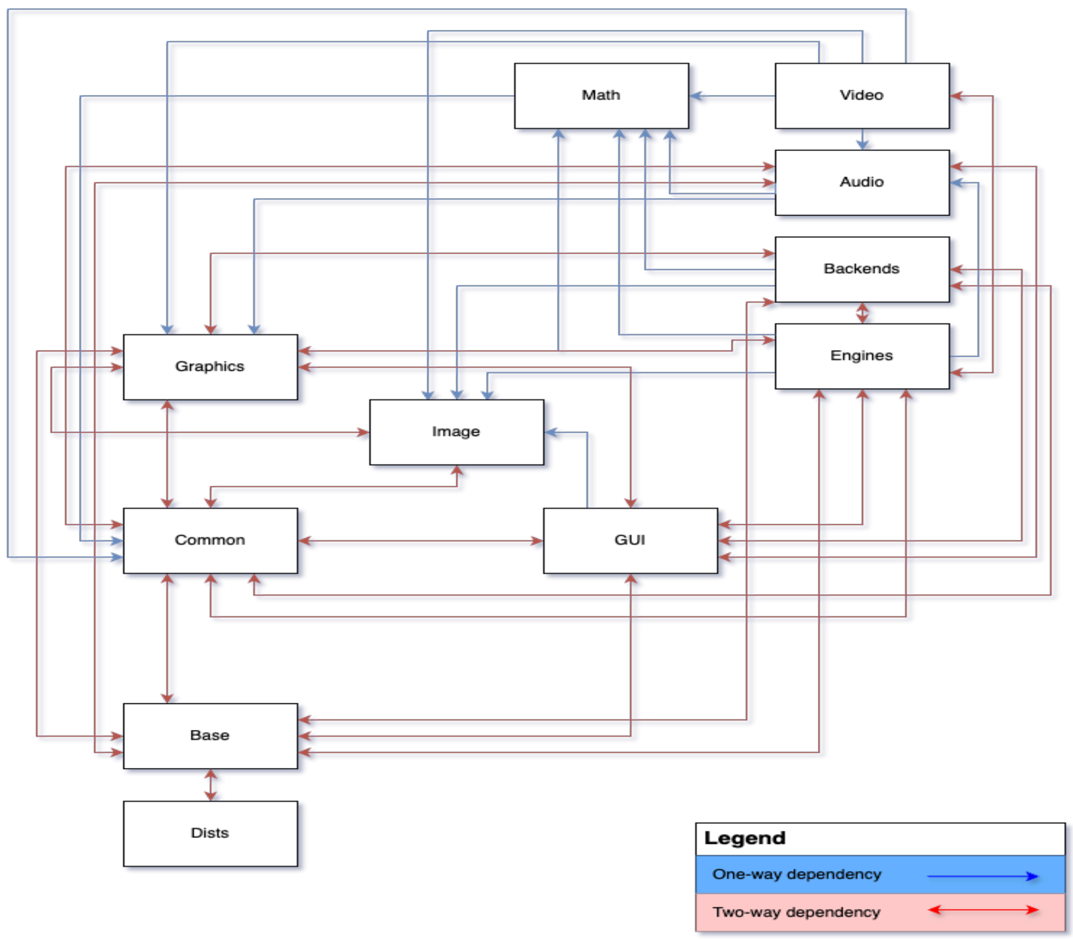
- Derive ScummVM’s concrete architecture: Understand the system’s actual structure by analyzing its codebase and comparing it to the initial conceptual model.
- Identify new subsystems and dependencies: Highlight previously unrecognized elements and connections that are crucial for system functionality.
- Perform reflexion analysis: Evaluate how well the conceptual architecture aligns with the implemented design, identifying any discrepancies.

The report is organized into several key sections to ensure a thorough exploration of ScummVM’s architecture. It begins with the derivation process, where tools like SciTools Understand are used to uncover subsystem interactions. This is followed by an overview of the concrete architecture, updates to the initial conceptual model, and a discussion of newly identified subsystems and dependencies. The second-level subsystem analysis delves into a specific component to provide a deeper understanding. Finally, the report presents use cases, highlights limitations and lessons learned, and concludes with insights into ScummVM’s architectural principles and areas for potential improvement.

Derivation Process

To determine the concrete architecture of ScummVM, we began by analyzing the program with the Understand tool, which allowed us to easily visualize the connections and dependencies among the top-level components. While this tool provided a useful overview of the system layout, we needed to go deeper to understand the underlying connections and their purposes. To achieve this, we employed the git blame command to identify the origins of specific dependencies and reviewed the commit messages to understand the reasoning behind certain dependencies. This approach helped us follow how the concrete architecture was formed. Additionally, we examined individual subsystems to understand their design and functionality; for example, a closer look at the graphics subsystem revealed essential components like blit, which interacts with the math component to facilitate image scaling and positioning. These steps collectively enabled us to gain a clear and comprehensive understanding of ScummVM’s concrete architecture.

Concrete Architecture



The concrete architecture consists of eleven major components at the top level of the architecture which are as follows: Graphics, GUI, Common, Engines, Backends, Base, Dists, Math, Audio, Image and Video. Additionally there is an Icons component at the top level, but it simply contains two icons with no functionality and was therefore excluded in the diagram above. This section provides a brief overview of the top-level components and examines the architectural style of ScummVM.

Overview of Top Level Components:

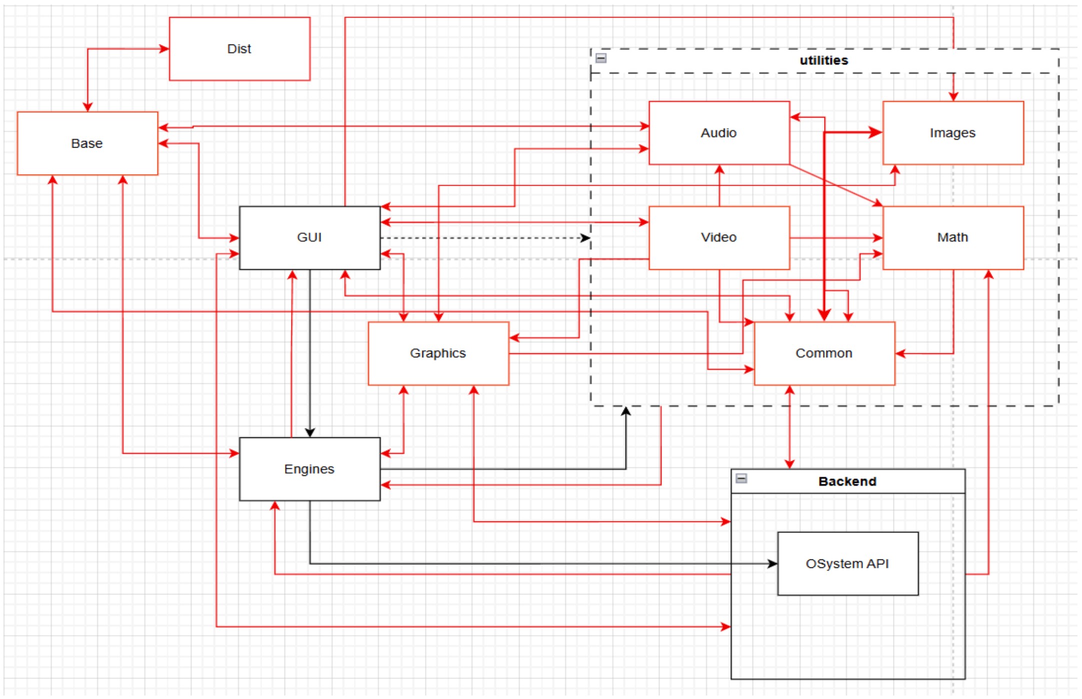
Graphics: Handles the rendering and display of the game.
GUI: Manages user interaction such as input within the game they are playing and ScummVM as a whole.
Common: Shared utilities used by many components for services such as serialization.
Engines: Engines that handle game specific scripts and gameplay logic for supported games.
Backends: Interacts with the underlying platform or system to handle file I/O and other operating system specific operations allowing for classic games to be played on modern systems.

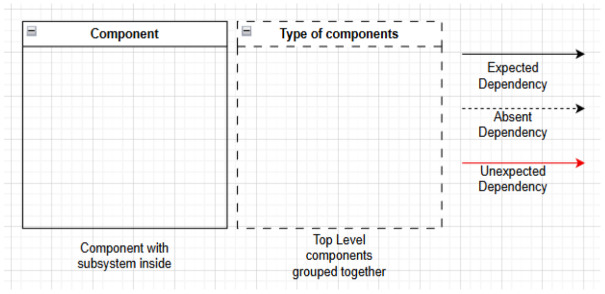
Base: Provides core functionality for the startup and running of ScummVM with files such as main.cpp.
Dists: Contains files to assist in the distribution of ScummVM across different platforms.
Math: Utilities for mathematical operations, frequently used by other components such as audio and video for mathematical operations such as vector and matrix calculations among others.
Audio: Provides MIDI support and other audio features such as music and sound effects.
Image: Contains image codecs, handles image processing and supports common file types such as PNG, JPEG and GIF.
Video: Similar to images but for videos, encodes and decodes various standard video formats such as AVI and MKV.

Architectural Style:

The architectural style of ScummVM is a combination of layered and interpreter-style architecture. The first key architectural style is layered architecture where each layer has their own responsibilities and functionalities that are abstracted away from each other. For example, the GUI, a component on the top layer processes user input, and sends it to the Engine to be interpreted. From here the engine may utilize utilities from the Common component or send it to the Backend for an platform-specific operation. The second key architectural style of ScummVM is the interpreter-style architecture which is at the core of ScummVM's functionality. ScummVM, using game-specific engines, acts as an interpreter for each supported game. This interpreter-style approach enables ScummVM to run a wide range of classic games on modern systems with minimal additional code, as only the engines need to be written specifically for each game.

Updated Conceptual Architecture





Initial Conceptual Architecture:

Our initial Conceptual architecture only had 5 components; GUI, Game Engines, OsystemAPI, Backend and Utilities. These components were set up in a layered style where GUI was the top layer, Game Engines and OsystemAPI were the middle layer, and backend was the bottom layer.

Modifications Made:

Additional components were added as the system is a lot more complex than initially described. The new components are: Base, Dist, and Graphics. Additionally, the “Utilities” component is a lot less centralized than the conceptual architecture made them out to be, as in reality, there is not one central utilities folder. Instead, there are a number of sub-components that make up utilities in the concrete architecture, and we have updated the concrete architecture to emphasize that the utilities component has several sub-components, which are: audio, images, video, math, and common. For more information about these new components/subsystems, consult the “new subsystems” section below.

Due to all these new components, there are a lot more dependencies in the new conceptual architecture, many of which were not present in our initial conceptual architecture. These divergences are represented with red arrows. For convergences, we have represented them with black arrows, and absences are noted with dotted arrows. More information about new dependencies can be found in the “new dependencies” section below.

New Subsystems

In this section, we identify and discuss subsystems that were initially unrecognized in our conceptual architecture but have proven crucial after analyzing the actual ScummVM codebase. First and foremost, the Graphics subsystem manages all rendering tasks within ScummVM, handling image processing, scaling, and visual representation. Simply put, this subsystem is essential for rendering both game visuals and the GUI, ensuring a smooth user experience. Although it was initially overlooked during the conceptual design phase, further analysis of the code revealed that this subsystem plays a critical role in the operation of ScummVM. Including the Graphics subsystem in our model provides a more accurate representation of ScummVM's architecture, as it emphasizes the importance of this component for visual output and user interaction. Conversely, the Dist subsystem contains files and scripts that assist in distributing ScummVM across various platforms. It is responsible for ensuring that the application can be packaged and deployed efficiently through the accommodation of different operating systems and hardware configurations. Simply put, the Dist subsystem plays a crucial role in managing platform-specific configuration settings and packaging tools, which allows for a smooth experience for users across platforms such as Windows, macOS, and Linux. On the other hand, the Base subsystem provides core functionalities for the initialization and execution of ScummVM. Simply put, it acts as the foundation upon which other subsystems operate. The Base subsystem interacts with components like Common for configuration management and Engines for managing the lifecycle of game execution.

New Dependencies

Common ↔ Backend:

Description: Backends connect to common through the following files: files.cpp, systems.cpp, systems.h, fs.cpp and fs.h files in the common subsystem. Common connects to Backends through the timer, dlc, keymapper, audiocd and fs subsystems.

Impact: This two-way dependency is crucial for both the backends and common modules. It enables backends to access essential components from common, while common can leverage functionalities provided by backends. For example, the connection between fs-factory.h in backends and file.cpp in common allows common to manage file handling seamlessly. This setup ensures that as files are created or accessed in the system, they are stored, maintained, and managed effectively. The dependency provides a standardized interface across platforms, facilitating a consistent and efficient way for ScummVM to interact with files and directories regardless of the operating environment.

Common ↔ GUI:

Description: GUI Relies on the common component to pull assets from the system and keep the GUI formatted correctly, the common component also relies on the GUI in order for common and the rest of the system (through common) to get user inputs/data.

Impact: Essential for proper display of game content and having the proper formatting for example, the GUI pulls word translations from common if the user chooses to play in another language.

Common ↔ Base:

Description: There is a bidirectional dependency between Common and Base. Base relies on Common for fundamental utilities like configuration management and system-specific operations, while Common depends on Base for FORBIDDEN_SYMBOL_EXCEPTION_printf definition.

Impact: This relationship is crucial for the initialization and overall operation of ScummVM, ensuring that core services are available system-wide and facilitating seamless interaction between different subsystems.

Images ↔ Graphics:

Description: Images works with Graphics to render images onto the screen. Images handles loading and decoding of image files, while Graphics provides the tools to display these images and manage rendering contexts.

Impact: This dependency is vital for displaying visual content within games and the GUI, ensuring high-quality visual output and accurate representation of graphics.

Audio ↔ Base:

Description: There is an interaction between Audio and Base for initialization, configuration, and accessing global settings.

Impact: This dependency ensures that audio functionalities are correctly initialized and managed within ScummVM, adapting audio behavior based on user settings and system configurations.

Audio ↔ GUI:

Description: GUI uses the Audio subsystem to provide auditory feedback within the user interface, triggering sound effects for actions like button clicks or alerts. It also opens the pop-ups to configure audio.

Impact: This dependency enhances user interaction by adding an auditory dimension to the GUI, making the interface more engaging and intuitive. Allows to control audio.

Video ↔ GUI:

Description: Video and GUI work together to render ScummVM's user interface onto the display.

Impact: This dependency ensures that the GUI is properly displayed on the user's screen as intended.

Video → Graphics:

Description: Video relies on Graphics to render video frames onto the display and for decoding.

Impact: This dependency is critical for presenting video content within games or the GUI, ensuring videos are displayed correctly and maintaining visual quality.

Backend → Math:

Description: Backend (openGL in particular) depends on Math for performing low-level mathematical operations required for various platform-specific functionalities.

Impact: This dependency supports precise computations essential for accurate system behavior, contributing to smooth operation across different hardware and operating systems.

Backend → Engines:

Description: The Backend and Engines subsystems have a bidirectional dependency. Engines use OSystemApi from Backends. Backends utilize the engines' debugger and perform some checks to configure the engine (for example it checks if the game can't adapt to any resolution, then renders it to a framebuffer so it can be scaled to fill the available space). It also provides common implementations for event handling.

Impact: This bidirectional dependency is crucial for the seamless operation of games within ScummVM.

Backend ↔ GUI:

Description: GUI relies on backends for networking and cloud. Backend gets information from GUI about the debugger and may use it to display some pop ups or send an event.

Impact: This dependency ensures consistent interface behavior across platforms, providing a uniform user experience and managing device-specific features.

Backend ↔ Graphics:

Description: Backend interacts with Graphics to handle low-level rendering operations. Graphics uses Backends APIENTRY definition.

Impact: This dependency is crucial and enables ScummVM to utilize the full potential of underlying hardware, improving rendering performance and visual quality.

Engines ↔ Graphics:

Description: Engines interact with Graphics to render game visuals. Graphics only use Engines when a user is playing on Mac. From the code I assume it uses some workarounds in case the rendering fails.

Impact: This bidirectional dependency is crucial for ensuring seamless visual output and stable gameplay.

Engines ↔ Base:

Description: Engine uses Base for some global variables like scumm vm version. Base may interact with Engines to manage the lifecycle of game execution.

Impact: This dependency ensures that game engines are properly integrated into ScummVM, facilitating smooth transitions between different games and consistent handling of core functionalities

Engines → GUI:

Description: Engines depend on GUI to display engine-specific interfaces, such as in-game menus or dialogue boxes or to monitor states.

Impact: This dependency allows games to present interactive elements to the player, enhancing gameplay by providing necessary interfaces

Graphics → Math:

Description: Graphics depends on Math for calculations related to rendering, such as coordinate transformations and scaling

Impact: This dependency ensures that graphical computations are precise and efficient, improving visual fidelity

Graphics ↔ GUI:

Description: Graphics and GUI have a bidirectional dependency; GUI uses Graphics to render interface elements, such as animations, while Graphics may receive rendering instructions from GUI

Impact: This dependency is vital for displaying the user interface correctly, ensuring GUI components are rendered accurately

GUI ↔ Base:

Description: GUI interacts with Base for initialization, event handling, and accessing global configurations

Impact: This dependency integrates the GUI into ScummVM's core functionality, enhancing responsiveness and user interaction

GUI → Images:

Description: GUI depends on Images to load and process graphical assets used in the user interface

Impact: This dependency enables the GUI to display visual elements effectively, enhancing the aesthetic appeal and usability of the interface

Base ↔ Dist:

Description: Use each other for configurations.

Inter Utility Dependencies

Description: all the components in the utility box have some dependency to one another this allows them to quickly support the whole system through each other

Impact: Allows all utility components to quickly get the data they need from other components to be helpful and efficient when supporting the rest of the system

Second-Level Subsystem

The scummvm/graphics component contains 6 sub-components responsible for handling different aspects of graphics, text, and UI rendering. Below are the expected dependencies and an estimation of each sub-components' intended use.

Blit -> scummvm/gui
Responsible for bit-block transfers of pixel data from one area of memory to another, often used in rendering images onto the screen.

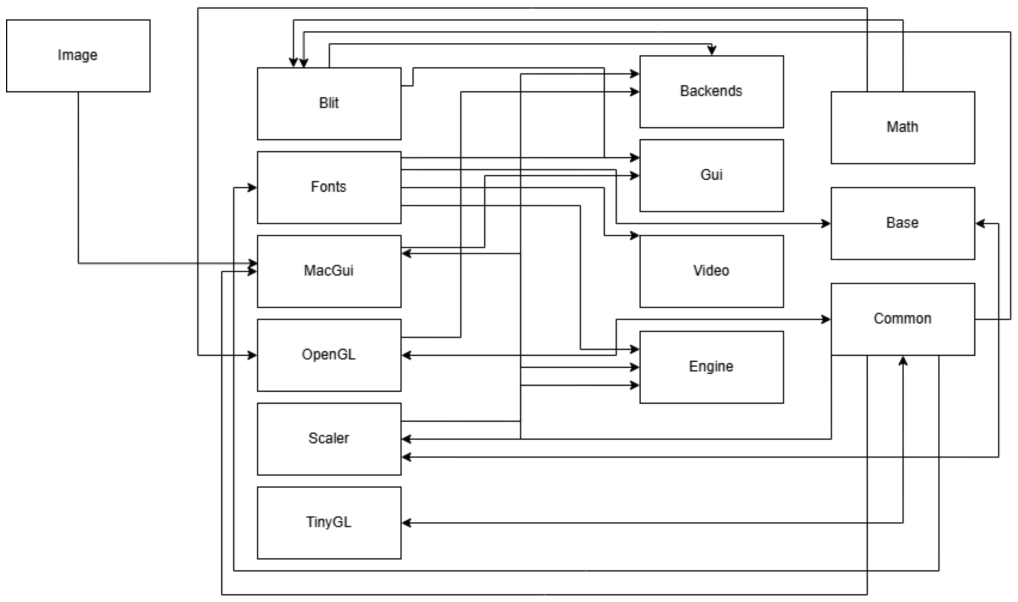
Fonts -> scummvm/engines
-> scummvm/gui
-> scummvm/video
Manages text rendering within ScummVM. It provides font loading, text formatting, and rendering services to various parts of the system, including game engines, the GUI, and video playback modules.

MacGUI -> scummvm/gui
Handles the graphical user interface elements specific to Mac-based game engines. It leverages common GUI widgets and interacts with engine-specific graphics modules.

OpenGL -> scummvm/backends
Provides support for rendering graphics using the OpenGL API. It serves as a bridge between ScummVM's abstract graphics operations and the hardware-accelerated functions provided by the host system's GPU, enhancing performance for graphics-intensive tasks.

Scaler -> scummvm/backend
Responsible for resizing and scaling graphical content. It interfaces with the backend to access platform-specific scaling algorithms.

TinyGL <-> scummvm/common
A lightweight OpenGL subset used within ScummVM to provide basic 3D graphics capabilities where full OpenGL support may not be available.



Upon performing Reflexion Analysis, we discover the following unexpected dependencies within the subsystem:

Common -> Blit
Provides utility functions or shared resources that the Blit component uses for rendering operations, such as image manipulation utilities or memory management.

Math -> Blit
Provides mathematical functions, possibly for graphics transformations or calculations necessary for positioning and scaling images during rendering

Fonts <-> scummvm/base
Accesses core resources for font rendering and configuration settings, such as font size, color, or localization, ensuring consistent text rendering across different games.

MacGUI <-> scummvm/engines/sci/graphics
Interacts with the sci/graphics engine to render graphics specific to the Mac interface or support game-specific Mac UI elements in titles that use the SCI engine.

Common -> MacGUI
Provides shared resources or utilities to MacGUI, potentially including UI elements, cross-platform abstractions, or helper functions used for rendering Mac-style graphics.

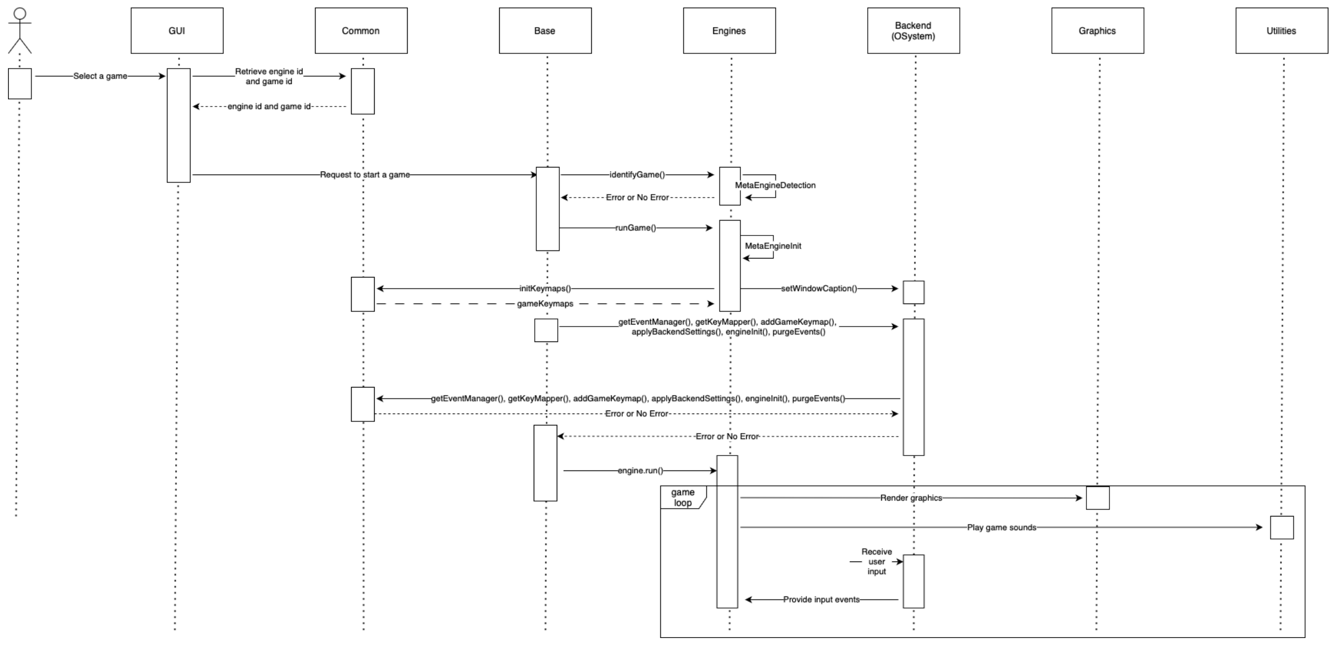
Image -> MacGUI
Supports MacGUI by providing image resources or functions for displaying Mac-specific graphical elements, such as icons or UI images.

OpenGL <-> scummvm/common
OpenGL interacts with scummvm/common to access cross-platform abstractions or shared resources that facilitate rendering on different hardware, improving compatibility across systems.

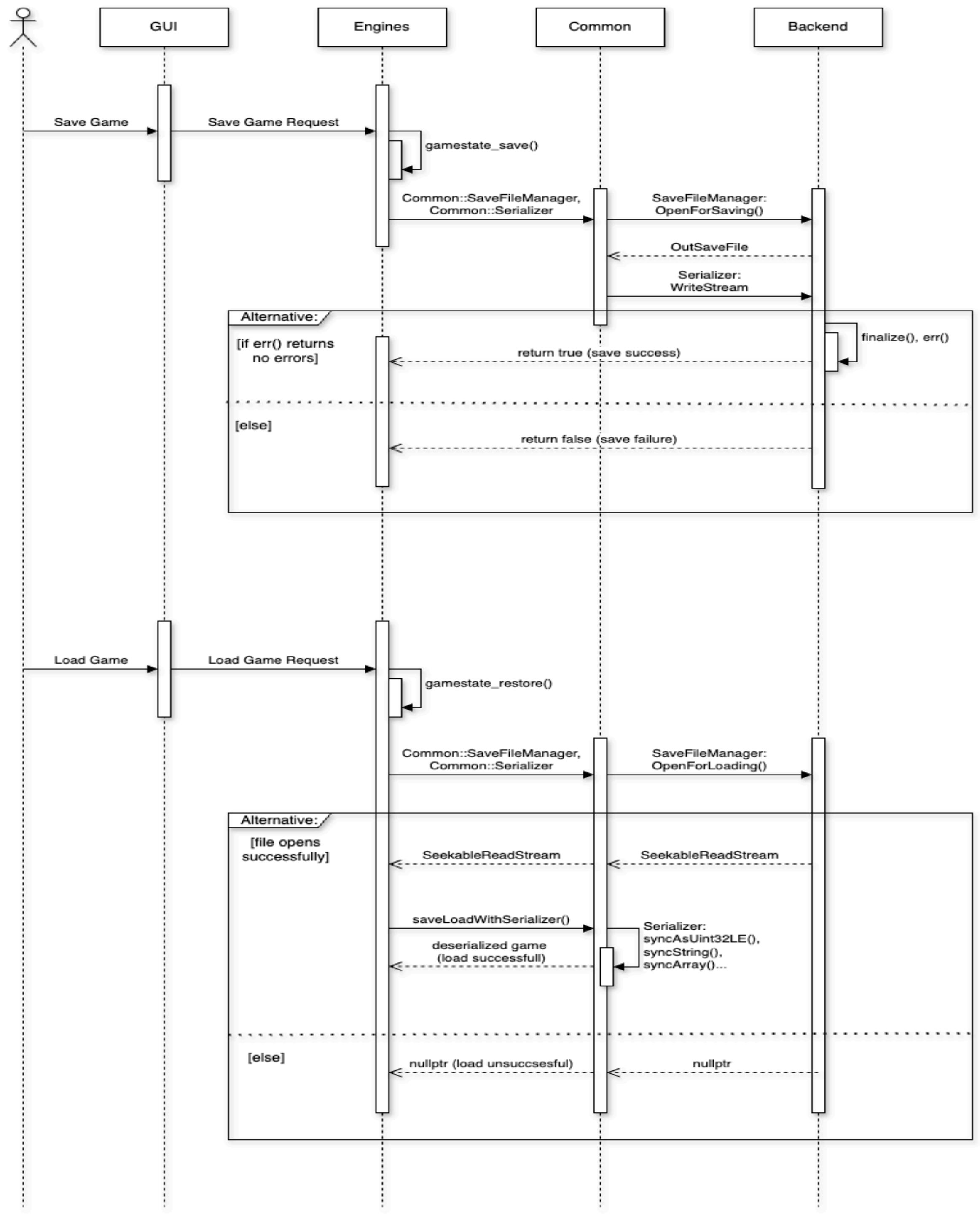
Scaler <-> scummvm/base
<-> scummvm/engines
Relies on scummvm/base for core settings or utilities related to scaling, such as screen resolution configurations or scaling preferences, ensuring proper display on various screen sizes and interacts with scummvm/engines to perform engine-specific scaling adjustments, accommodating unique graphics requirements or resolutions needed for different game engines.

Use Cases (Sequence Diagrams)

Use Case 1: Starting Game



Use Case 2: Saving and Loading Game:


GUI
Engines
Common
Backend
Save Game
Save Game Request
gamestate_save()
Common::SaveFileManager, Common::Serializer
SaveFileManager: OpenForSaving()
OutSaveFile
Serializer: WriteStream
Alternative:
[if err() returns no errors]
finalize(), err()
return true (save success)
[else]
return false (save failure)
Load Game
Load Game Request
gamestate_restore()
Common::SaveFileManager, Common::Serializer
SaveFileManager: OpenForLoading()
Alternative:
[file opens successfully]
SeekableReadStream
SeekableReadStream
saveLoadWithSerializer()
deserialized game (load successfull)
Serializer: syncAsUint32LE(), syncString(), syncArray()...
[else]
nullptr (load unsuccsesful)
nullptr

Limitations and Lessons Learned

Throughout this project, we used SciTools Understand to explore ScummVM's layered architecture. At its core, ScummVM separates responsibilities into distinct layers to ensure scalability and adaptability. The core layer handles shared functionality like memory management and file I/O, while the engine layer provides modular support for individual game engines, allowing them to operate independently. The backend layer abstracts platform-specific details, such as rendering, input, and audio, enabling ScummVM to run seamlessly across multiple platforms like Windows, macOS, and Android. This architecture ensures maintainability and flexibility, making it possible for ScummVM to support a wide variety of games while evolving to meet modern platform requirements.

Completing this analysis provided us with valuable insights into the distinctions between conceptual and concrete architectures. We learned how to conduct reflexion analysis effectively, allowing us to identify actual dependencies and compare them with our expected design.

Furthermore, to deepen our understanding of ScummVM's architecture, we examined the source code to verify subsystem interactions and dependencies. This process improved our skills in analyzing and interpreting complex codebases, especially those we didn't originally develop.

Conclusion

This report set out to derive the concrete architecture of ScummVM, compare it with the original conceptual model, and conduct a reflexion analysis to identify discrepancies. We utilized tools such as SciTools Understand for codebase analysis and conducted an in-depth exploration of subsystem interactions. Our analysis uncovered significant components that were not included in the original conceptual architecture, such as the Graphics subsystem. These discoveries led to substantial updates, including a deeper understanding of dependencies and the need for subsystem integration to accurately reflect ScummVM's structure.

The primary takeaway from this analysis is the emphasis on modularity and abstraction in the software's design. ScummVM's capacity to support multiple game engines and platforms is largely dependent on these architectural principles, which allow it to remain robust, flexible, and adaptable. Moreover, enhancing documentation would be beneficial for new contributors, aiding them in understanding the system's architecture and making meaningful contributions.

The reflexion analysis yielded valuable insights into aligning the conceptual architecture with its practical implementation. This process highlighted the importance of iterative design and detailed code analysis to maintain architectural accuracy and system reliability.

Data Dictionary

Term	Definition	Type
ScummVM	An open source software that reimplements game engines to allow classic adventure games to run on modern platforms.	Software
OpenGL	An API for rendering 2D and 3D vector graphics.	API
TinyGL	A lightweight subset of OpenGL used on systems without full OpenGL support.	Library
OSystem API	A layer within the system architecture that deals with platform-specific details. Allows game engines to run on multiple platforms without issues.	Subsystem

Naming Conventions

ScummVM	Script Creation Utility for Maniac Mansion Virtual Machine
I/O	Input and Output
GUI	Graphical User Interface
OS	Operating System
API	Application Programming Interface
OpenGL	Open Graphics Library
TinyGL	Lightweight Subset of OpenGL

References

E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Boston, MA, USA: Addison-Wesley, 1994.

ScummVM, "ScummVM GitHub Repository," [Online]. Available: https://github.com/scummvm/scummvm. [Accessed: Nov. 15, 2024].

ScummVM Documentation: ScummVM, "ScummVM Official Documentation," [Online]. Available: https://docs.scummvm.org/. [Accessed: Nov. 15, 2024].

Sci-Tools Understand: Sci-Tools, "Understand," [Online]. Available: https://scitools.com/. [Accessed: Nov. 15, 2024].