Concrete Architecture of ScummVM

Authors

Nick Axani 21nra@queensu.ca
Trajan Brown 21tjnb@queensu.ca
Aaron Chen aaron.chen@queensu.ca
Gwendolyn Hagan 21gwh4@queensu.ca
Jack Hickey jack.hickey@queensu.ca
James Kivenko 21jk90@queensu.ca

Table of Contents

Abstract

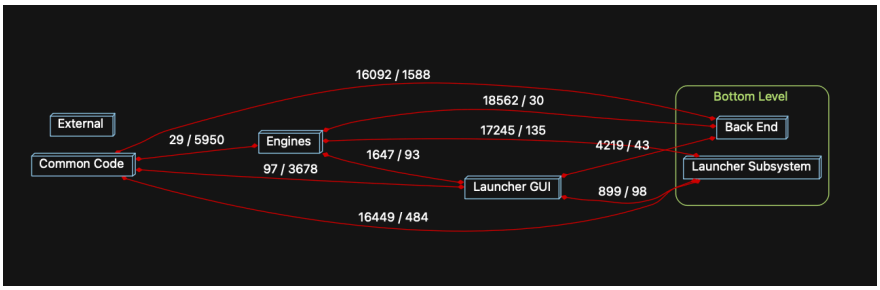
This report provides a detailed analysis of the concrete architecture of the ScummVM project, focusing on the Back End, Launcher, and Engine subsystems. Building on the conceptual architecture proposed in the previous report, it compares the conceptual design of ScummVM with the actual implementation. This report points out key divergences and optimizations that were considered in the course of development. The main conclusion is that while ScummVM indeed retains many features characteristic of a layered architecture, there are many useful shortcuts within the program that improve its performance and maintainability. For example, the Launcher GUI bypasses the Launcher Code to directly interact with the Back End, simplifying the flow of communication between the graphical interface and OS-level resources.

This report also points out the Common Code as a centralized utility layer, including that heavy dependencies exist in a number of subsystems: namely, the Engines and the Launcher GUI. This means that the Game Data Subsystem, a unique layer we originally envisioned, is integrated into both the Common Code and Back End. This centralization improves resource management and avoids code duplication. Furthermore, the SCI Engine, our subsystem of choice, depends on the shared resources provided by the Common Code, making the whole system more integrated than we initially assumed.

Using dependency analysis, we demonstrate that the concrete architecture of ScummVM is not a strict layered system but rather a practical, hybrid structure that balances flexibility, performance, and maintainability. This flexibility allows ScummVM to run a wide range of classic games on numerous platforms while minimizing unnecessary dependencies between subsystems. The results provide insight into the design decisions in ScummVM and the challenges associated with developing a cross-platform emulator of game engines with minimal overhead.

Introduction

This report will cover a concrete software architecture analysis of ScummVM, based on our previous conceptual analysis. This report provides a reflexion analysis comparing the concrete and conceptual architecture we recovered, and explores each of the top-level subsystems of the application. As well as establishing the general responsibilities and interactions between the different subsystems, this report will also detail the degree to which each subsystem is dependent on the others, and how the application has been designed to maintain high performance and flexibility. The concrete architecture of ScummVM is largely in line with the conceptual architecture from our previous report, emphasizing the assignment of specific roles to different subsystems while minimizing dependencies between them. This is characteristic of a layered architectural style, allowing ScummVM to be a versatile application which can run many different games on various hardware systems. The main difference between the concrete architecture and conceptual architecture is that while the concrete architecture still emphasizes the same inter-layer dependencies as the conceptual architecture, it also consists of multiple dependencies which skip over the conceptual layer order. The most notable of these "rogue" dependencies are between the Engine layer and the Launcher GUI, which bypass the Launcher Subsystem to allow for direct communication.



The other main difference between the conceptual and concrete architecture is an island of files which do not interact with the rest of the application. These files consist of markdown files with information about the application, licenses for externally developed libraries incorporated into ScummVM, and a folder dedicated to "devtools", which contains files used to help ScummVM developers working on the code. None of these files have any dependencies, and all are irrelevant to the standard use cases of the application (i.e. they are independent of running ScummVM in order to play a video game). For all these reasons, our analysis does not delve into these files, which in Understand have been grouped into the "External" folder.

The reasons for ascribing a layered style to ScummVM vis-à-vis its concrete architecture are largely the same as those in our conceptual analysis. Minimizing dependencies between the different top-level subsystems allows for portability across platforms and adaptability with regards to which games/engines it can run. This allows developers to focus on specific components of the application without having to concern themselves greatly with other subsystems. Although the actual file structure of the application does not group files in the exact

same way as this concrete architecture, it does strongly resemble the concrete architecture, with top level directories for common code, back end files, and the engines (as well as some others). The other top level directories (such as “audio” and “math”) can be grouped with the top level subsystems of this concrete architecture.

Altogether, the concrete architecture of ScummVM is strongly reminiscent of the conceptual architecture we previously proposed, with clear attributes of a layered architectural style, with the main divergences being relatively few dependencies outside of the layer order.

Architecture

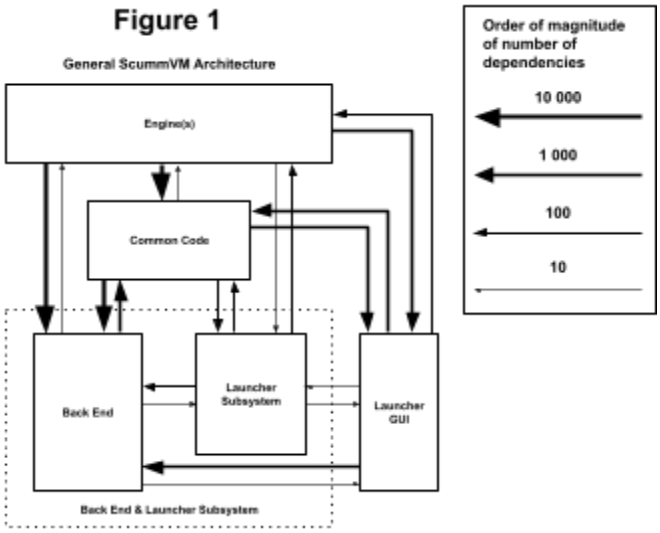
The concrete architecture of ScummVM consists of multiple distinct subsystems, each responsible for a distinct subset of functionality. These subsystems are the Bottom Level, the Common Code, the Engines, and the Launcher GUI. These approximately correspond to the layers in our conceptual architecture, the Back End and Launcher Subsystems, the Common Code, Game Engines and Data, and the Graphical User Interface, respectively. Although the conceptual architecture has similar internal divisions to our conceptual architecture, the relationship between the subsystems appears to be a more specialized and practical version of a layered structure. Instead of the linear dependency structure described by a layered structure, ScummVM’s architecture consists of components with dependencies on multiple other components. The notable attribute of its architecture is that between any two given components, a majority of the dependencies appear to travel in one direction. Each pair of components has at least three times as many dependencies in one direction than than other, with the most lopsided dependencies between a pair of components having a ratio closer to 800:1.

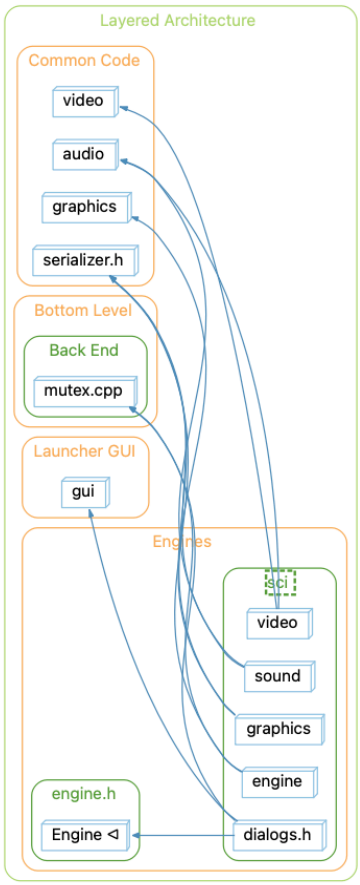
This pattern arose from attempting match files and folders to the different layers in our conceptual architecture. The goal was to match files as dictated below:

Bottom Level	The bottom level contains both the Back End and the Launcher Subsystem. The Launcher Subsystem contains the non-GUI related files which operate the main ScummVM application. The Back End contains files which interface with the operating system or other external systems. These include files for networking, files for cloud communications, log files, event handlers, and others.
Common Code	Processes and files which are useful broadly across the application, in both engines and the main ScummVM application itself. This includes audio, graphics, math files, programs for dealing with various file formats, video, header files corresponding to required functions the engines must provide, and others.
Engines	All files directly corresponding to each engine. This also included files relating to achievements, save states, and header files and abstract classes which the engines need access to.
Launcher GUI	Files related to the operation of the ScummVM main application GUI.

To the right is a figure which shows the relative magnitudes of the amounts of dependencies between each pair of components. This was measured by taking the logarithm of the number of dependencies in each direction, and scaling the arrows to a proportional pixel size. For example, there are on the order of 10 000 dependencies from the Engines to the Back End, and only on the order of 10 dependencies the other way. This figure helps to illustrate the lopsidedness of dependencies between some different subsystems. It's clear that some elements of the layered style from our conceptual architecture are present here. With the components of the concrete architecture organized in diagram form to resemble the layers of our conceptual architecture, it is clear that there is a bias for dependencies moving "downward" through the components, and that the subsystems corresponding to layers from our conceptual architecture have more dependencies to/from them overall.

Figure 1





Top-Level Subsystem Studied: SCI Engine

Within ScummVM, the SCI engine is modularly designed, with primary subsystems dedicated to specific functions essential to SCI games. Much like the overall structure of ScummVM, the SCI engine is composed of distinct systems that each serve a particular function. Key subsystems include the Virtual Machine, Resource Loader, Audio Handler, Graphics Handler, User Interface, and Game Logic Core. These components operate with a degree of dependency between them to utilize game data, process commands, and manage gameplay, allowing the SCI engine to support various SCI titles within a single framework that handles the core functions, while each game title retains its unique assets and scripts.

The architecture features directed dependencies, typically flowing from higher-level components to lower-level ones. Similar to ScummVM's conceptual architecture, this design establishes a hierarchy that enables consistent performance across SCI games by minimizing excessive inter-subsystem calls. For example, the Virtual Machine relies heavily on the Resource Loader for asset retrieval, while the Game Logic Core uses the Virtual Machine to interpret SCI scripts. This dependency ladder enables efficient debugging and maintenance, allowing subsystems to operate independently and make one-way calls.

SCI Engine Subsystems

- **Virtual Machine:** Integral to SCI operations, it translates compiled SCI scripts into gameplay, managing game states and coordinating with the Resource Loader for necessary assets.
- **Resource Loader:** Responsible for using game resources such as character models, sprites, sound files, and background images. This separation reduces the load times and data dependencies, ensuring optimal performance.
- **Audio and Graphics Handlers:** These units manage audio and visual elements based on game requirements, using ScummVM's shared graphics and audio subsystems for broad device compatibility.
- **User Interface:** This subsystem manages in-game menus, inventories, and player interactions. Its design integrates seamlessly with ScummVM's interface standards, allowing for consistent user experience across games.
- **Game Logic Core:** This subsystem holds game-specific scripts and mechanics for individual SCI titles, allowing unique game logic within the SCI framework without changing the broader engine operations.

Reflexion Analysis

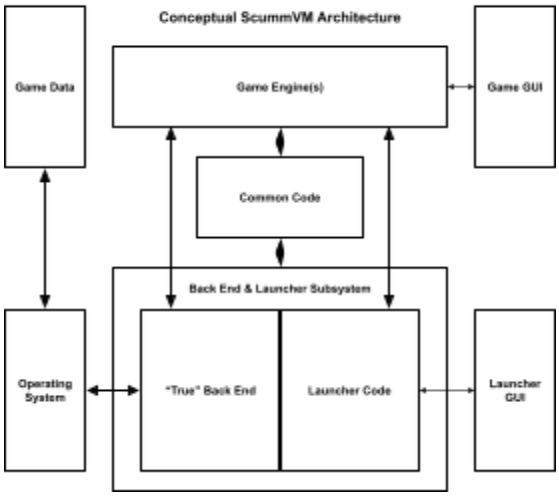
Back End and Launcher Subsystem

Upon further examination of the back end system, we find that the subsystem largely resembles our proposed conceptual architecture.

Breakdown of Initial Conceptual Architecture

In our proposed conceptual architecture (see **Figure 2**), we suggested that the back end and launcher subsystem could be divided into largely four subsystems that communicated only with adjacent subsystems. These were: the OS (Operating System), the "true" back end, the launcher code, and the launcher GUI, in that order. Although the OS is not a part of ScummVM itself, it was included to further elaborate on the purpose of the back end. We had understood the back end as being the interface between the OS, the game engine(s) and the code of the launcher. The back end was responsible for all of the system level processes, ensuring that all of the most vital and important procedures were isolated from the rest of the system and could be contained to one section of the code. Like its name suggests, we understood the launcher code subsystem to be responsible for the functionality of the launcher. Finally, we had included the launcher GUI as a distinct subsystem from the launcher code.

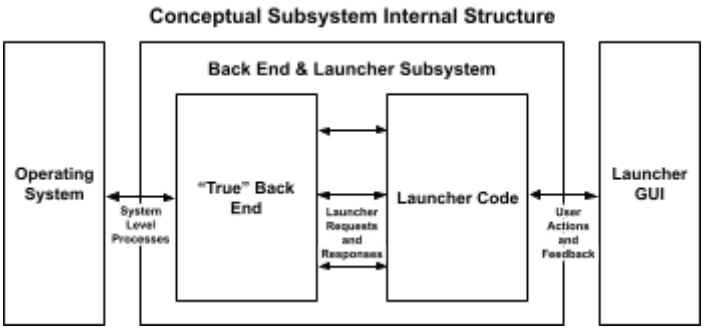
Figure 2



Outside of those interconnected subsystems, we had conceived of the game data being accessed by the OS, with ScummVM itself receiving game data via the "true" back end. Of course, with the entire purpose of ScummVM being to recreate older game engines for modern

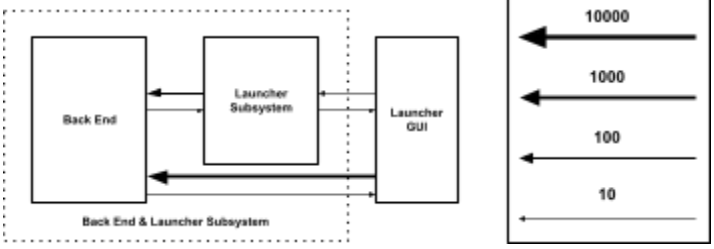
devices, we had also included a subsystem to encapsulate the game engines managed and contained within ScummVM. With particular relevance to the back end and launcher subsystems, we had considered the game engine subsystem to rely on the launcher code subsystem and heavily on the back end subsystem, due to its position as an interface between ScummVM and the OS. The common code, also, was thought to have a similar reliance on the back end.

Divergences and Absences in the Back End and Launcher Subsystems



Upon examining the source code and the concrete architecture (see **Figure 3** below) we have determined that it is largely consistent with our conceptual architecture. We've found no reason to fundamentally change out any of the components described in our conceptual architecture as relating to the back end and launcher subsystems. As expected, every single other component relies most heavily on the back end, which is entirely consistent with our conception of the back end as the primary interface between the software and the OS and its system level processes. Similarly consistent with our conceptual architecture, the strongest dependencies regarding the back end are the engine and common code subsystems, which rely heavily on the back end. In particular, subsystems including rendering and graphical information tend to have the strongest reliance on the back end.

Figure 3

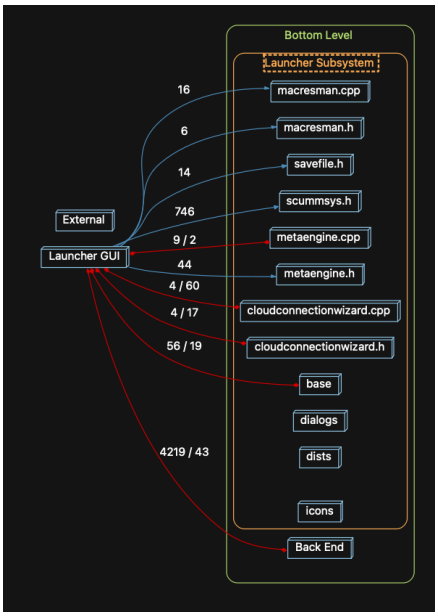


The most significant divergence from our initial conceptual architecture was the dependencies between the launcher GUI and the back end. Instead of communicating with the back end by passing through the launcher subsystem, the launcher GUI often bypasses the launcher subsystem to rely directly on the back end for communication with the OS. In retrospect, this makes a lot of sense as much of the system's reliance on the back end and OS is due to its implementations of graphical processing. This highlights a consistent theme that we have encountered in our building of the concrete architecture from the bones of the source code and our previous conceptualization architecture, that being the natural dissolving of a strict layer order.

While the ScummVM software is still certainly a layered architecture, it is not quite as layered as we had originally thought. There is no interface between much of the system and the back end (which interfaces directly with the OS), everything communicates directly with the back end in some way. However, this is consistent with the ideas discussed in lecture where, over time, additional shortcuts are added between layers for the convenience of developers and for the overall betterment of the system.

The next most significant divergence from our initial conceptual architecture was the relationship between the launcher GUI and the rest of the code base. Previously, the launcher

GUI subsystem had been thought of as entirely independent from the rest of the system except for its sole reliance on the launcher subsystem for obvious reasons. However, we have found that in our concrete architecture, the launcher GUI subsystem has additional dependency relationships with both the common code and the game engine subsystems, the latter having far fewer than the former. The common code, as will be described in the following section, is thought to contain the remainder of the GUI's code base as well as much of the additional graphical processing that is not otherwise handled by the back end. With such similarities, it is understandable that the launcher GUI subsystem would rely on the common code and vice versa.

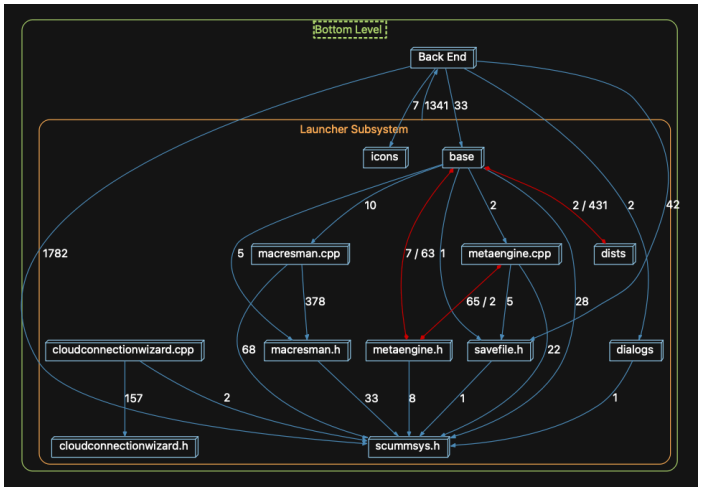


A new dependency relationship has also been formed between the launcher GUI and the game engine subsystems, with the game engine subsystem relying more heavily on the launcher GUI than the reverse. With the benefit of hindsight, we can also easily rationalize the relationship between the launcher GUI and game engine subsystems. Firstly, the game engine relies on the launcher GUI for reasons similar to the launcher GUI's dependency on the common code, both systems interface heavily with the back end and the code relating to the graphical processing of the ScummVM software as a whole. However, the launcher GUI has very little reliance on the game engines because the majority of its functionality interfaces directly with related components previously identified.

The only identifiable absence relating to the back end subsystem that we have identified is the lack of a separate game data subsystem. This game data component was not carried forward to our concrete architecture and all of its proposed functionalities have been incorporated into other components (namely, the common code and back end subsystems). This is unsurprising, as it was more of an abstraction of how the game data was stored, and was not a fully fledged subsystem in our conceptual analysis.

Further Examination of the Concrete Back End and Launcher Subsystems

Beneath the back end subsystem we find the vast majority of the processes and functionalities that rely on external interfaces such as the OS and external file systems. Broadly, the former functions as the OSystem that was identified in our conceptual architecture, handling much of the subsystems that interface the rest of the ScummVM software with the OS. Additionally, another major function of the back end is to abstract platform specific processes and functions to enable portability among devices. Among those that work as an interface between the OS and the ScummVM system are:



- The sections of the audio system which interface directly with the OS
- Process scheduling so as to avoid race conditions
- The interface with the OS and the system that handles the management of game save files
- The sections that manage ScummVM's interactions with the taskbar of the OS
- A virtual keyboard
- Several other features

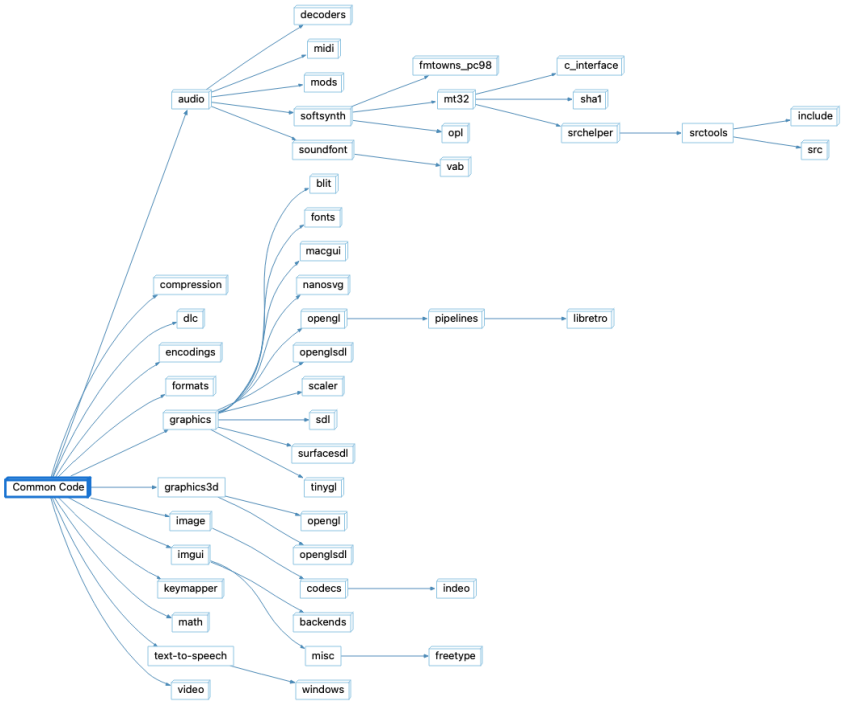
Similarly, there are the items that have been grouped under the back end because they operate in conjunction with other external interfaces that ScummVM utilizes. Broadly, this includes items such as:

- The portion of ScummVM dedicated to their networking and their internet interactions for both file sharing and multiplayer
- Those that interface with cloud-based file sharing systems such as the Google Drive and Dropbox
- The plugins that ScummVM utilizes
- Several other features

Finally, the concrete implementation of the launcher subsystem includes the *icons* and *base* folders. *Icons* contains the image icon of the ScummVM software and *base* contains the back end functionality of the launcher, particularly relating to recognizing plug-ins, games and its functionality with the command line of the OS. Additionally, the launcher subsystem features several loose files unattributed to any particular directory. These files are responsible for managing shaders, information about completed engines and basic detection for plug-ins and engines, and they are one cause for the dependency relationship between the engine and launcher subsystems.

Common Code

The common code dependencies show that the architecture style is not strictly layered, since there are plenty of codependencies. However, they are heavily weighted in one direction making it similar to a layered structure. The common code in ScummVM holds a lot of utility to be accessed by the engine and launcher, while it itself depends on the backend. Subfolders within the common code include, common, video, audio, graphics, image, GUI and math.



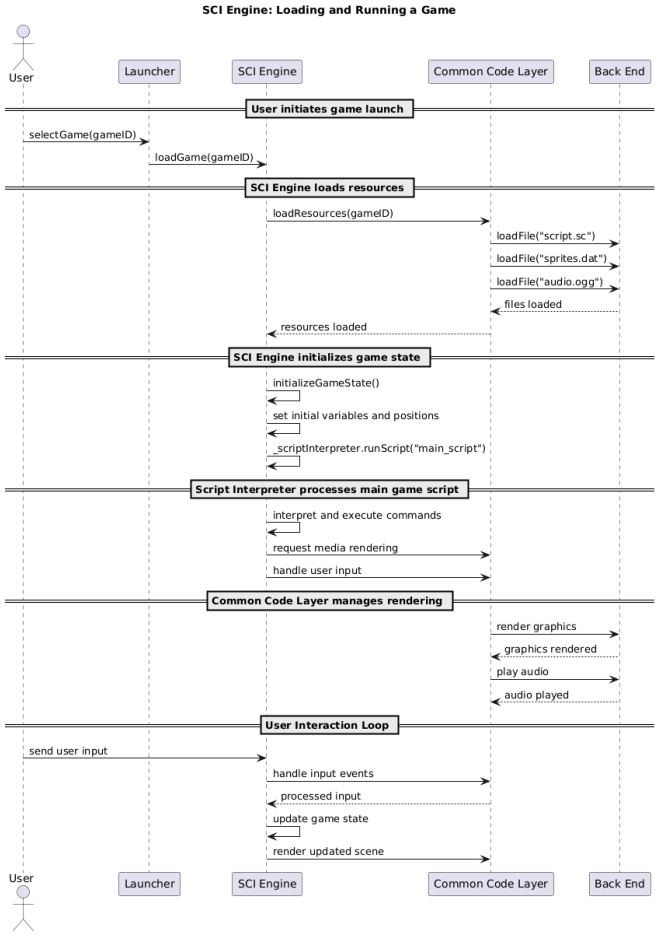
The engine depends heavily on the common, video, graphics, audio, and GUI subfolders within the common code. This is because the engine

is a higher layer that will frequently call functions that can be compartmentalized into common code, essentially calling from the layer below it.

The backend acts as a low-level layer that abstracts platform-specific details, providing a bridge between ScummVM's core modules and the underlying hardware or operating system. By offloading platform-dependent tasks to the backend, the common code remains relatively platform-agnostic, enhancing portability across different devices. Despite this structure, the direct dependencies between modules like GUI and audio show that certain functionalities are intertwined to support a seamless user experience. The dependencies of the common code on to the backend are a way for the common code to centralize the inputs and information to be interpreted for the specific platform being used.

Engines

The SCI (Sierra's Creative Interpreter) Engine in ScummVM is responsible for interpreting and running game logic specific to Sierra's classic titles. As a core module, it processes game scripts, manages game states, and handles user interactions, allowing these older games to be played seamlessly on modern platforms. Contrary to the more isolated structure we visualized in our conceptual architecture, the SCI Engine is integrated with both the *common code* layer and the *back end* subsystem to handle essential functionalities like rendering, media processing, and input handling.



The SCI Engine's structure includes several key components that enable it to replicate the behaviour of the original Sierra engine. There is a script interpreter, which reads and executes game commands from game data files. Using the Interpreter pattern, this interpreter processes commands in real-time, converting them into actions such as character movement, scene changes, and object interactions, mimicking the original gameplay logic. Another crucial component is the resource manager, which oversees the loading of assets such as sprites, audio files, and text. By using ScummVM's common code APIs for file handling and decoding, the resource manager ensures that assets are processed consistently across platforms, allowing the SCI Engine to run smoothly without platform-specific modifications.

Overall, the reflexion analysis of the SCI Engine shows that the engine is very interconnected with shared resources, unlike what the conceptual architecture suggested. In practice, the SCI Engine draws heavily on shared methods in common code for functions like asset management and input processing. This is in line with ScummVM's focus on creating

uniform methods that can be reused across multiple game engines, which enhances maintainability within each engine. Rather than functioning as a standalone module, centralization allows ScummVM to minimize duplicated code while supporting the needs of unique engines.

External Interfaces

The following is a list of the external interfaces that were identified within the ScummVM software:

- Operating System
- Launcher GUI & Menu Dialog
- Game Engine GUI(s)
- External File Systems (e.g. Google Drive, OneDrive)
    - In order to load/save game data
- LAN / other networks for file sharing

Conclusions

We have concluded that while minor changes to our conceptual architecture have been required since we began our conceptual architecture, the vast majority of our initial attempts were successful and consistent with our thoughts moving forward. As such, the influence of our conceptual architecture is very clear in our concrete architecture.

Similarly, we have concluded that our initial conception of the ScummVM system as a layered architecture style remains applicable to our current concrete architecture. However, the implications of the layered structure of our conceptual architecture have changed. Previously, we had thought that the system was much more layered, particularly with respect to the back end. The back end communicated with the OS directly, interfacing between itself and the rest of the system and especially the launcher subsystem. Several components relied on the back end only through their reliance on another component that relied on the back end. In our concrete architecture, however, each of the top-level subsystems relies directly on the back end, bypassing much of the original artificially layered structure we had previously conceived. Though some of the layered structure we had previously considered in our conceptual architecture for ScummVM has been stripped in our new concrete architecture, the back end still serves its purpose as a layer between the software and OS. Similarly, other components rely far more heavily on the back end than the reverse, as do the game engine components in the common code component. This further suggests a strong layer style within the ScummVM system. As such, we are comfortable continuing with our understanding of the ScummVM system as a layered architecture.

Additionally, the majority of the divergences between our conceptual and concrete architecture are the new dependency relationships that have formed between previously unrelated components. We predicted each of the major components of the system to have dependency relationships with at most three other components. Clearly, in our concrete architecture, all top-level subsystems have a dependency relationship with each other - another divergence. Through our concrete architecture, it has become clear that ScummVM's major components rely on each other far more than previously anticipated, which remains consistent with the typical lifecycle of software architectures. As discussed throughout the report, several absences and divergences can be found between our conceptual architecture and concrete

architecture. For example, the absence of two major components (game data and game GUI) from our conceptual architecture and new diverging dependency relationships between major components. While these absences and divergences represent our own differing conceptions of the ScummVM software and its architecture, each can be rationally understood and the rationale for each have been discussed through this report.

Altogether, the concrete architecture of ScummVM is strongly reminiscent of the conceptual architecture we previously proposed, with clear attributes of a layered architectural style.

Lessons Learned

In the transition from conceptual to concrete architecture we learned a lot about the importance of architecture and architecture practices. We saw many examples of how our own conceptual architecture diverges from the concrete architecture. While this shows the importance of comparing concrete architecture to find mistakes, it also shows us how in practice a developer's need for efficiency, ease of use and functionality can cause code to vary from its original conceptual form. While layered architecture provides a strong foundation in conceptual designs, concrete implementations often differ to strict architectural styles. In ScummVM, subsystems depend on each other based on what is efficient and convenient, rather than enforcing an artificially strict layering of subsystems. While the layered architecture style we identified in our conceptual architecture provides a good foundation, the implementation of said conceptual architecture must be able to change and adapt when necessary to allow for functional, fast and simple to understand code.

We also learned how concrete architecture often reveals a greater degree of integration and interconnectedness than initially anticipated in the conceptual stage. One way this is apparent is how we observed significantly more divergences than absences. This is because, while a conceptual architecture can be great for planning and understanding, it doesn't always account for the needs that a concrete architecture has: to collaborate and share resources between subsystems. As such, certain systems in the conceptual architecture must be carried forward to the concrete architecture because they are very difficult to omit. In these cases, the system cannot function without them. Absences only exist when their subjects are carried forward in another (often lesser) form, such as being integrated into other components. In contrast, divergences are significantly more common as the concrete development process often reveals previously unnoticed gaps within the conceptual architecture that must be filled before the software can function.

Overall, we consider this project to be a success as we have learned a lot of new information about the specific architecture of ScummVM. More importantly, we have also learned about the transition from a conceptual architecture to a concrete architecture in the realm of software development, a new skill that we anticipate will be particularly valuable in our future careers within software development.

Naming Conventions / Glossary

API: Short for Application Programming Interface. An API is a way for a system to provide its functionality to other systems without allowing them full access to its internal structure.

Game Engine: A collection of software intended to provide all the functionality needed to run a game file.

Local Area Network: Often referred to as simple LAN, a local area network is a network that connects devices throughout a localized area.

Operating System: Sometimes shortened to OS, an operating system is a piece of software which runs on a given machine which allows processes to display output, access files, and interface with other hardware resources in an organised fashion.

SCI Engine: Sierra's Creative Interpreter Engine. Shortened to SCI Engine for ease of reference throughout this report and within most materials regarding the ScummVM software.

UI: User Interface. This is the part of the system that the user can see and interact with. Can be a Graphical User Interface (GUI) for visual navigation, or a Command Line Interface (CLI) for navigation using text commands.

References

ScummVM Development Team. "scummvm." ScummVM GitHub, https://github.com/scummvm/scummvm. Accessed 15 November, 2024.

echo-cowsay. "Conceptual Architecture of ScummVM", echo-cowsay. https://jdhickey.github.io/echo-cowsay/. Accessed October 31 2024.