# CISC 322/326: Concrete Architecture of ScummVM

Ryan Jacobson
21rkj6@queensu.ca
20356136

Evan Kreutzwiser
21ejk12@queensu.ca
20352619

Reid Stobo
reid.stobo@queensu.ca
10186536

Jacob McMullen
jacob.w.mcmullen@queensu.ca
20341079

Kashan Rauf
22kr11@queensu.ca
20369391

Mike Stefan
21mgs11@queensu.ca
20342569

## Contents

***Abstract*— This report is a detailed analysis of the concrete architecture of ScummVM, with reference to our previous report on ScummVM's conceptual architecture. We derive the concrete architecture via careful examination of ScummVM's source code and with the use of Understand, a tool designed to analyze the architectures of software systems by providing advanced diagrams of dependencies and functions calls between subsystems. Similarly to the conceptual architecture, we determined the concrete architectural style to be layered, with the primary components being the Backends, Engines, Common Library, Launcher GUI, and the Games. The internal subsystems of the primary components are briefly discussed, although the SCI Engine is explored more thoroughly further into the report. Explanations for the discrepancies between the conceptual and concrete architectures are provided for both ScummVM as a whole and the SCI Engine. Two sequence diagrams of use cases are also presented and discussed; one for adding a game to the library and the other for loading a save file at runtime.

## I. Introduction

In the previous report about ScummVM's conceptual architecture, we discussed the system's high-level architecture, subcomponents, control flow, instances of concurrency, evolution, and developer responsibilities. Essentially all of the data we used to construct the conceptual architecture and the other details about ScummVM was gathered from online documentation. The source code was mostly avoided with the exception of using the git history to determine ScummVM's evolution.

This report will explore the concrete architecture of ScummVM. We carefully examine the source code found in ScummVM's open-source repository and use a software architecture analysis tool called Understand that is capable of providing useful diagrams of component dependencies. As with the conceptual architecture, we found the concrete architectural style to be layered with the same primary components: the Backend, Engines, Common Library, Launcher GUI, and the Games. However, some discrepancies still exist between the two architectures. We discuss these differences in a reflexion analysis of the subsystems and their interactions. Furthermore, an in-depth explanation of the concrete architecture for one of the subsystems, the SCI Engine, is provided along with another reflexion analysis for its divergences from its own conceptual architecture. Two use cases with detailed sequence diagrams are explored near the end of the report, and various other diagrams are used throughout to illustrate interactions between components.

## II. Derivation Process

Our derivation process for the concrete architecture was much simpler then for the conceptual due to the use of the Understand software by SCITools. After extracting the provided build of ScummVM and placing opening it in Understand the software made it easy to map directories and files to the components found in our conceptual architecture. Our starting components were the engine, the backend, the GUI and the common code library. Following the ScummVM wiki, we were able to easily find the directories where each of these components were located. After organizing our initial architecture layout, the tools within understand allowed us to immediately generate an interactive dependency graph to work off of.

Following this, our team discussed which components should and should not be included in our architecture. Despite not appearing as separate components in our conceptual architecture, the ScummVM wiki had explicitly mentioned Base and the OSystem API as major components of the project. After looking into the source code and analyzing its dependencies, we decided to make Base a subcomponent of our concrete architecture which would be grouped with GUI to form the Launcher. This was because both are responsible for handling the launching games by communicating with the engine. For the OSystem API we decided to leave it as a sub component of the common directory in the common library. While it an important file, it would be more accurate to describe it being accessed by the backend from within the common library.

After this process, there were still two directories left in the project unaccounted for: the math and graphics directories. The math directory fit nicely into the common library, since its tools could be effectively utilized by any new engine or backend added to the system. However, many of us debated about where graphics should be placed. Despite assumptions that it was a common library subcomponent, investigations into the source code revealed that it only contained files directly related to OS and macOS graphics specifically. So, we decided to group the graphics directory with the backend component. Finally, we removed the Kyra and Plumber engines from the engine component so that any unnecessary dependencies wouldn't appear in our graph.

While Understand made the process incredibly straightforward, there are some shortcomings with this method. Most of our architecture was formatted heavily around what appeared in our conceptual architecture. Components which were missing from our conceptual architecture or were poorly documented required increased investigation on our end. Additionally, Understand maps every single dependency found in the project, no matter the importance. That means that we couldn't rely that every listed dependency in the generated dependency graph was representative of the the system as a whole.

## III. Final Concrete Architecture & Subsystems

The final concrete architectural style of ScummVM is layered and it consists of 5 primary subsystems, being the Common Library, the Launcher GUI/Options Menu, the Backends, the Game Engines, and the Games. Notably, the concrete architecture is very similar to the conceptual architecture, as both have the same layered style and both have the same primary components. The lowest layer is still the backend, which remains the only subsystem that interfaces with the operating system and is the backbone of ScummVM's portability capabilities. The backend uses the OSystem API to interface with the rest of ScummVM, primarily the GUI Launcher and Engines, as discussed in the conceptual architecture. The next layer is comprised of The Launcher GUI and the Engines. The Launcher GUI is just a system launcher and configuration menu, while the game engines are portable re-implementations of the game's main executables. As before, each game engine is implemented differently, and often have their own separate architectures. The highest layer is the Game, which is ran within the environment provided by the game Engine. The Common Library is a large collection of common data types and utility functions that are utilized by the Backends, GUI Launcher, and Engines.

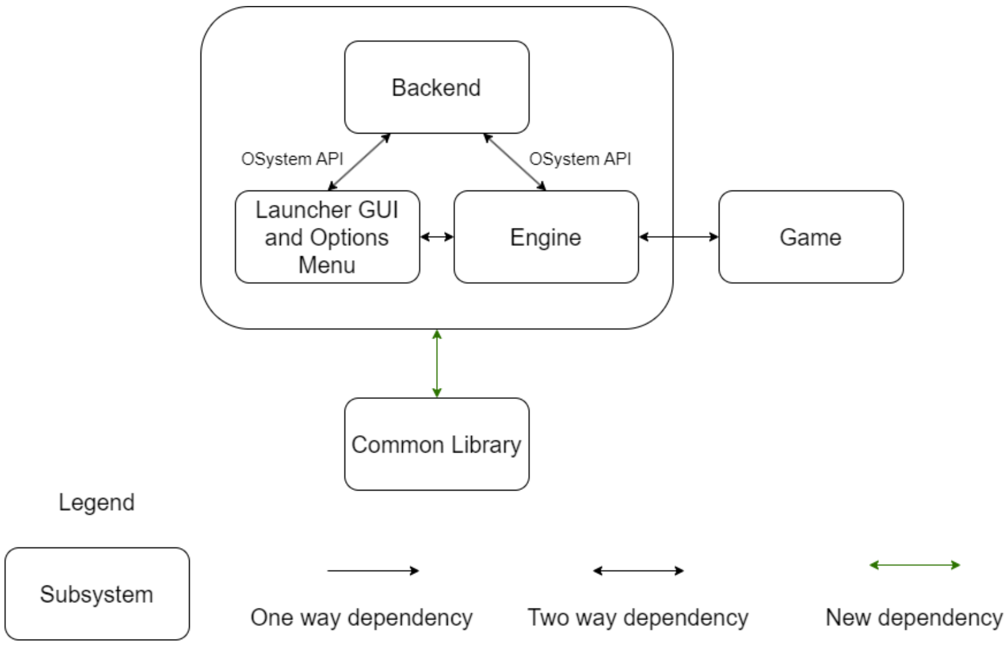

Figure 1: The concrete dependency diagram of ScummVM with only top level subsystems

The most significant changes are the new two way dependencies between the Common Library and the Engines, Backends, and Launcher GUI. In the conceptual architecture, the Engines, Backends, and Launcher GUI each only had a one way dependency on the Common Library. These discrepances will be discussed in a later section.

As shown above in Figure 1, all top level subsystems are dependent on each other, with the exception of the game subsystem, which is only dependent on the engines. As with the conceptual architecture, the OSystem API still serves as an interface between the backend layer and the lower engine and Launcher GUI layers. Two of the top level subsystems of the architecture also have their own subsystems. The first is the Launcher GUI which can be split into the GUI and the Base. The second is the Common Library which can be split into Common, Graphics, Math, Audio, Video, and Image subsystems. (Individual engines and backends can also contain their own subsystems, but they differ from each other and don't all need to be individually explored for the purposes of this report.) These lower level subsystems show more precisely how the components of ScummVM interact with each other. A more elaborate (and difficult to follow) diagram of the subsystem dependencies can be seen below in Figure 2.

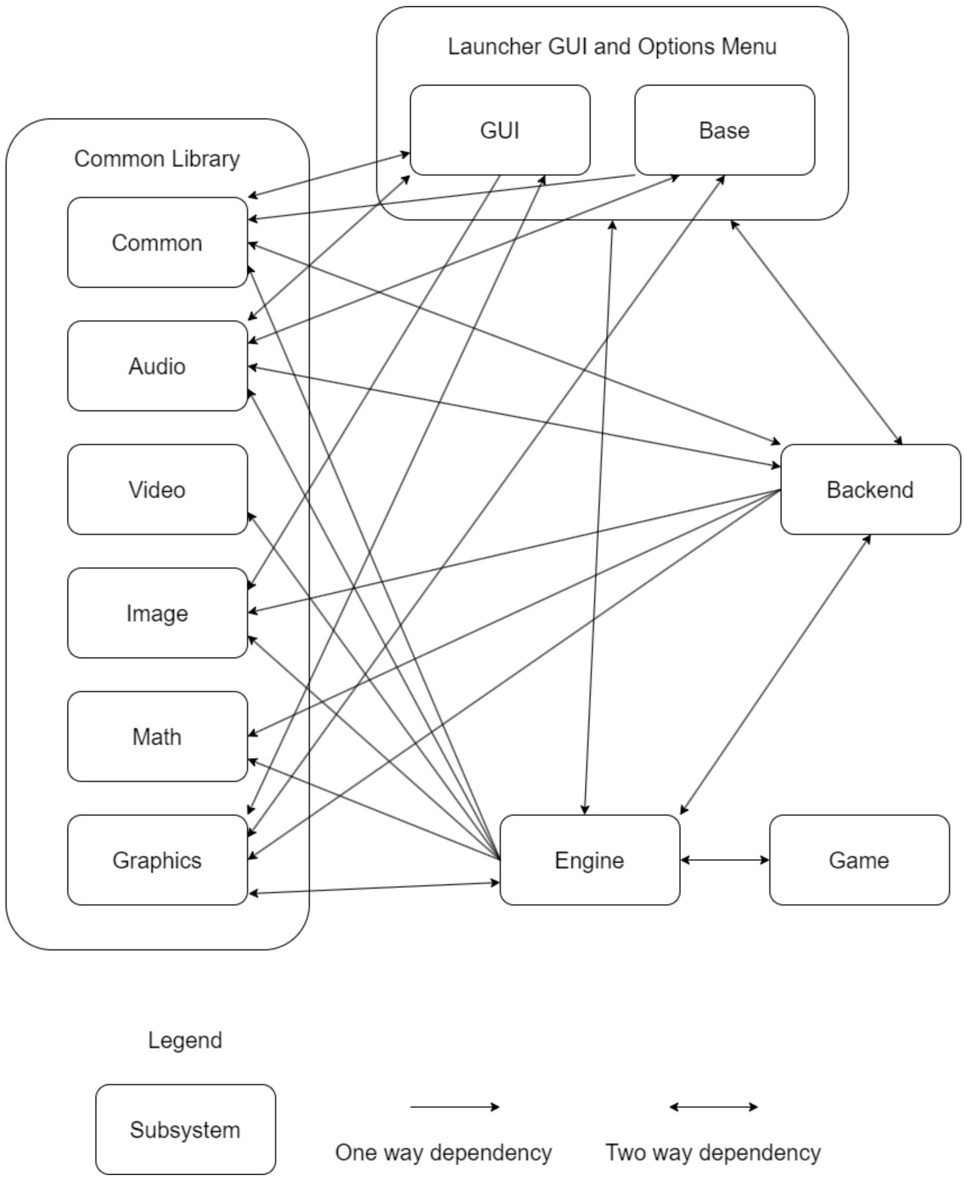

Figure 2: The concrete dependency diagram of ScummVM with all levels of subsystems

The details of the Backends, Engines, and Games remain mostly unchanged. A directory called /gui is the bulk of the Launcher GUI, and functions as the primary interface for the user while accessing settings and launching games. The Base is handles command line arguments and the main loop of ScummVM. Most of the components of the Common Library serve simple and self explanatory purposes. Math provides a large variety of mathematical methods and objects. Audio provides audio encoding/decoding and processing functionalities for games. Video provides video processing and decoding functionalities and handles subtitles. Image provides image decoders and supports several types of image file extensions. Graphics provides rendering capabilities, font management, frame rate handling, and image scaling. Common provides many different types functionalities that would commonly be found in a languages standard library.

## IV. High-Level Reflexion Analysis

In this section, we discuss the divergent dependencies found in ScummVM's source code and explain the reason that they appear in the concrete architecture, we also determine whether or not these changes are justified. Let the expression A ⟶ B mean that component A depends on component B.

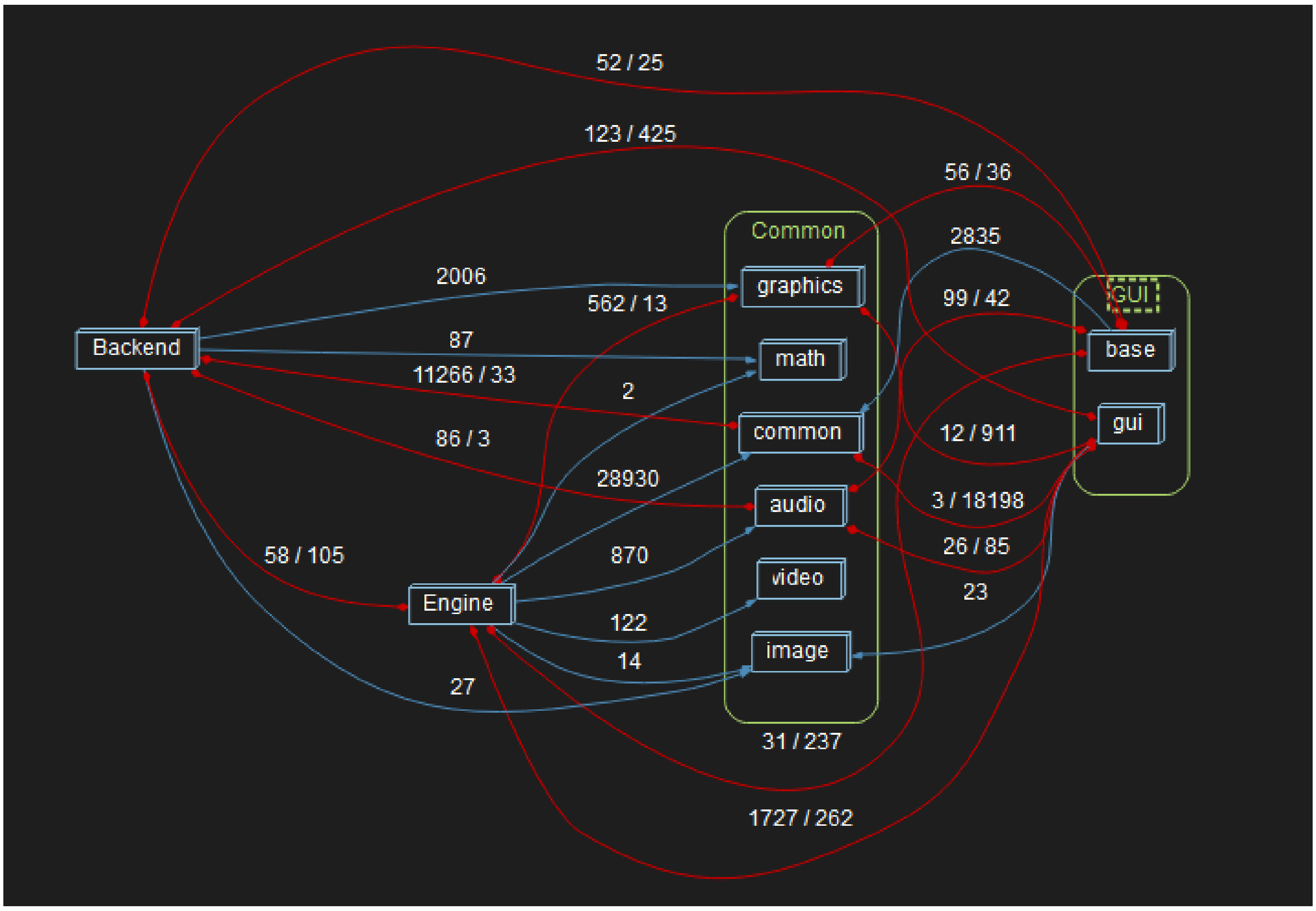

Figure 3: The dependency diagram generated with Understand

*A. Common ⟶ Engine*

We discovered that Common has 13 dependencies on Engine, while Engine has 30,483 dependencies on Common, making the dependency relation between them two-way. After investigating the source of these dependencies it was discovered that they all stem from the same functionality. The files in Common: graphics/macgui/macwindowmanager.cpp and .h rely on the Engine component for functionality related to pausing.

Even though the dependency itself is justified, the divergent dependency relation should not exist. Code from the macgui folder is related to platform-specific code for macOS devices, which should be part of the backends component as per the developer wiki [1]. As such, these dependencies should be refactored to stem from the Backend component instead of Common by moving the files, this would consequently resolve the divergent relation since Backend and Engine are already dependent on each-other.

*B. Common ⟶ Launcher (formerly GUI)*

Similar to the above dependencies, we found that Common has 196 dependencies on the Launcher component, leading to another unexpected two-way dependency relation. The majority of these dependencies (155 to be exact) are on the “base” subcomponent of Launcher, 153 dependencies are related to plugins for audio and graphics settings, while the other 2 are for version information that happens to be in the directory. The 41 remaining dependencies to the “gui” subcomponent of Launcher are for UI themes, and outputting messages about detected audio devices and drivers to the user.

The base subcomponent doesn’t interact with the gui subcomponent very much, but for the same reason we made them part of the same top-level component, we have determined that this dependency shouldn’t be changed or refactored as it would be counterintuitive to move the functionalities from either component to another.

### *C. Common → Backend*

We found that Common had 36 dependencies on the Backend component, making this another divergent relation that goes two-ways instead of just one. Common depends on various different modules found in Backend: An abstract class for managing audio on CDs called AudioCDManager, DLC (downloadable content) management functions, file management, the keymapper module which maps inputs to actions and events, and a timer used by OSystem.

The Backend component is described as a component for platform-specific backends to interact with the system through the implementation of the OSystem API by the documentation and our conceptual architecture report. However, most of these modules that Common depends on don’t fall into this category despite the component they are a part of:

- Common has a dedicated audio subcomponent that the AudioCDManager class would be better suited to.
- DLC management is related to games rather than platforms and is better suited to be in the same place as the modules related to managing save files (the Common component).
- By exploring what the keymapper and timer modules depend on are are depended on by, we found that they were extremely dependent on the Common component and were barely used in Backend.

The only exception to this is the modules related to file management, as file systems differ by platform. Similar to the previous divergence, we find that these modules should be refactored to be a part of Common except for the file management modules, as this change would reduce the number of dependencies that this divergent relation has.

## V. SCI Engine Architecture & Subsystems

The final concrete architecture of the SCI Engine is object-oriented, consisting of six primary subsystems responsible for various behaviour: Engine, Graphics, Sound, Resource Management, Parser, and Video.

The Engine subsystem is responsible for the bulk of the functionality for the Sierra Creative Interpreter (SCI), a stack-based virtual machine. It interprets and executes Sierra Script, is responsible for event handling, game logic and state management, and does so by managing and calling upon the other subsystems as required. The Graphics subsystem is responsible for the visuals of SCI games, managing animations, displaying of images and sprites on the screen, handling colour palettes and limited video rendering. The Sound subsystem manages audio, handling game audio and music playback and MIDI parsing which determines how such sound is triggered in the engine; the Audio subsystem also contains dedicated second-level subsystems for drivers and decoding specific audio formats used for the SCI Engine.

The Resource Manager handles the loading and handling of game resources such as images and audio files and decompressing game data and assets. The Parser subsystem defines the syntax and structure of the Sierra script language, handling parsing of game scripts as demanded by calls from the Engine subsystem. Finally there is a minor but dedicated Video subsystem for handling animation files referred to as VMD and RBT (Robot) videos), a more resource-intensive video container introduced in later Sierra games. This subsystem interacts with the Engine and Graphics systems accordingly.

Certain SCI subsystems depend on their relevant doppelgangers in the ScummVM common library, borrowing an amount of general functionality that is shared with other engines in ScummVM. This includes the Graphics, Sound and Video systems. This borrowing of functionality from the common library is routed predominantly through the Engine subsystem, but is sometimes achieved through direct calls. As should be the case, all dependencies between the SCI Engine and the rest of ScummVM are one-way: There are no classes outside of the SCI Engine that depend on any classes found within the engine.

The SCI Engine subsystem also depends on ScummVM's base Engine and MetaEngine classes to interface with the rest of ScummVM using the OSystem API. It is important to note that, departing from our conceptual architecture, engines in the backend such as SCI do not exclusively use OSystem to interact with the rest of Scumm; many cases exist where interacts are made through other elements in the the common library.

## VI. SCI Engine Reflexion

Performing a reflexion analysis on the SCI Engine highlighted several key differences between the conceptual and concrete architectures.

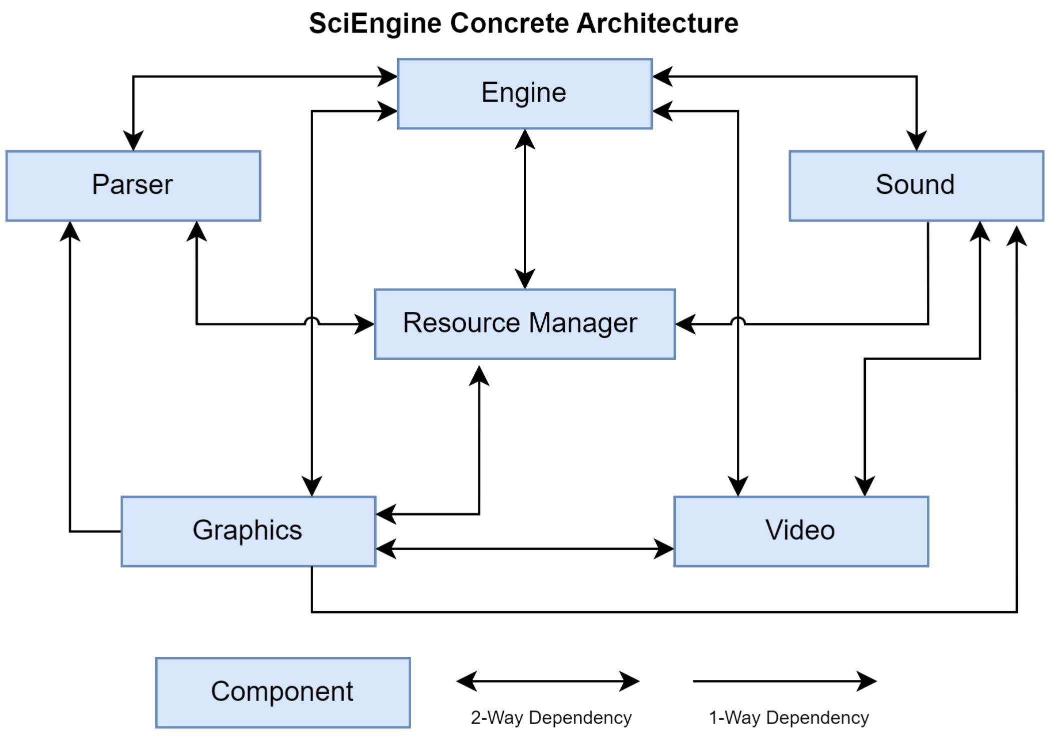

Figure 4: SCI Engine Concrete Architecture

The conceptual architecture has four divergences and one absence, as shown in the concrete architecture depicted in Figure 4. The absence can be found with Resource Manager $\longrightarrow$ Sound. There is still a dependency relationship between the Sound and Resource Manager system but is only one way in the concrete architecture. This absence can be explained through a fundamental difference between how sound resources are handled and how other resources are handled internally. Other systems, like the Parser or Graphics, must send the Resource Manager some level of configuration information. In the case of Graphics, it is the desired number of colours. This information will dictate which version of the resources the Manager will use. This relationship does not exist with the Sound subsystem because the differences handled at the resource level for other systems can be handled at the driver level for sound. All games for the SCI Engine will store their audio in the same format, rather than different resources for different hardware configurations, as they do for graphics.[1] The hardware and implementation details of actually playing this sound is handled at the driver level, with that being extensible to maintain portability. Since the audio system is unique in this way compared to all other systems in SCI Engine, it explains why the other systems require this dependency relationship to be two-way, while Sound only requires a single direction.

As for the divergences, There is a two-way relationship between the Video and Sound subsystems which was not found in our architecture. This dependency was added to allow the SCI Engine to support more games. While most games for this engine used the Sierra SEQ encoding for their video data, there were a select few games, including RAMA and Lighthouse, which used a different video encoding format. This format, called Robot, is a specialized format that compresses the video data along with the audio data in a single encoding. This form has several benefits over the encoder used by the rest of the games which use SCI Engine. The normal encoding relies on the Graphics and Engine system to ensure that the audio and visuals are synchronized. While that system works, the Robot encoding ensures that they will always synchronized by compressing the data together, and having the frame itself contain the audio information. [2] While it could be possible to have the data go through Graphics or Engine systems, those would both require a lot of overhead, while implementing this in the manner that they did creates a concise and efficient method for decoding this format. This dependency allows the SCI Engine to support games which, without it, would be impossible to play.

Another divergence can be seen between the Graphics and Sound subsystems. The primary sources for these divergences can be traced back to patches implemented to allow for a more efficient display of specific elements. Some of these calls include stopping all other audio output before beginning a video playback. Other examples can be found in specific elements used by some games which directly make use of audio resources as part of the element. The aforementioned examples could both have been implemented in ways which are supported by our conceptual architecture. Another important case with this divergence is found in the menu code. The user drop-down menus are implemented in the graphics system, which requires a connection between that system and Sound for the menus to use sounds. Since they do utilize Sounds, this dependency is mandatory for their implementation of the drop-down user menus. The final divergence can be found between the Parser and Graphics. Like the above-mentioned dependency between Sound and Graphics, this dependency is an implementation detail of the user drop-down menu. There are a few cases where the menu must make use of the Parser, and as such, this dependency is another consequence of the menu system being implemented in Graphics.

## VII. Use Cases

### I Use Case: Adding a Game to the Library

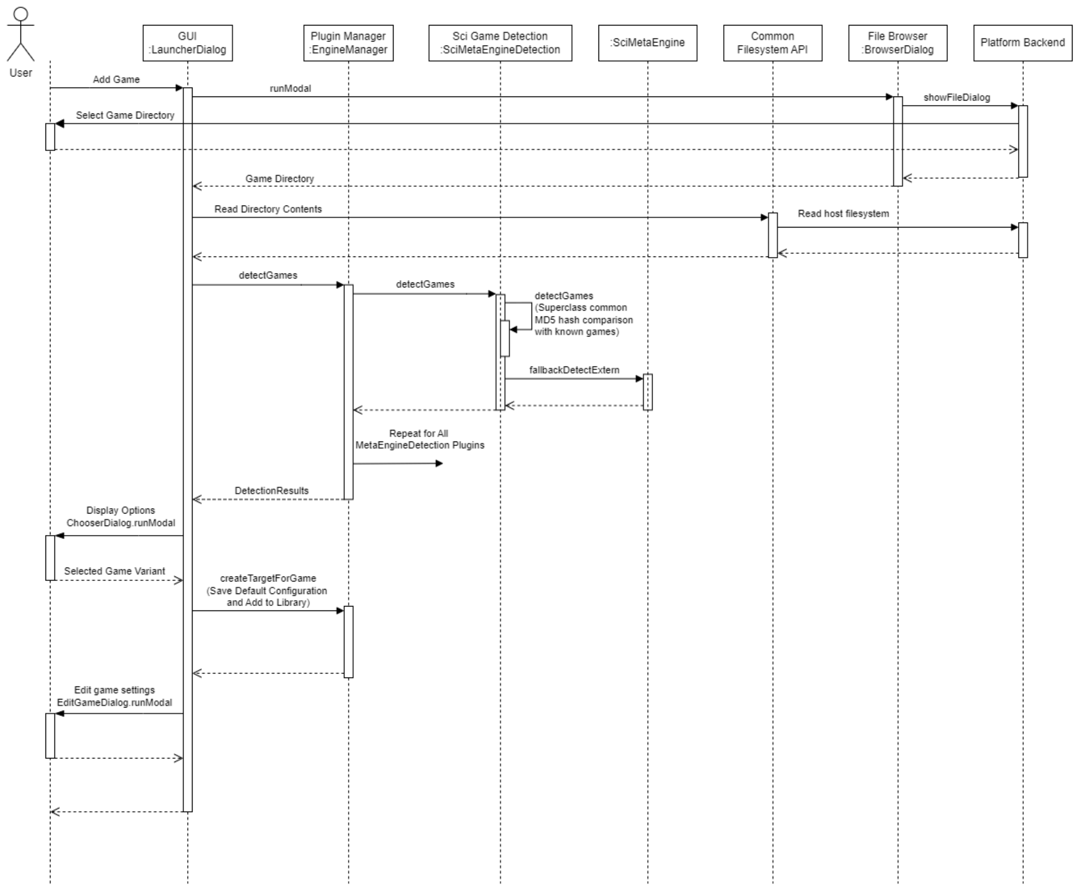
User
GUI :LauncherDialog
Plugin Manager :EngineManager
Sci Game Detection :SciMetaEngineDetection
:SciMetaEngine
Common Filesystem API
File Browser :BrowserDialog
Platform Backend
Add Game
runModal
showFileDialog
Select Game Directory
Game Directory
Read Directory Contents
Read host filesystem
detectGames
detectGames
detectGames (Superclass common MD5 hash comparison with known games)
fallbackDetectExtern
Repeat for All MetaEngineDetection Plugins
DetectionResults
Display Options ChooserDialog.runModal
Selected Game Variant
createTargetForGame (Save Default Configuration and Add to Library)
Edit game settings EditGameDialog.runModal

Figure 5: Sequence diagram depicting the process of adding a new game to the library

ScummVM eases the process of adding new games to the library by implementing a system that can automatically detect which game you are adding from the game files alone, and even make educated guesses about the default settings for games it does not recognize. This saves non-technically inclined users the trouble of editing configuration files and determining how to set up the game they want to play.

The process of adding a game begins with the user selecting the directory containing the game files. ScummVM reads the directory contents so it can hand off the game files to the game engines' detection code. Engines and their game detection routines are tracked using a plugin system that allows engines to be dynamically linked and loaded. The `EngineManager` portion of this plugin system keeps a record of all the included game engines, and the `MetaEngineDetection` plugins that provide game recognizing functionality. The launcher queries the engine manager for a complete listing of detection plugins and asks each one to check whether the files match any games they recognize. All of the detection plugins subclass the `AdvancedMetaEngineDetection` class, which contains common functionality to calculate MD5 hashes of the game file contents and compare them against records of known supported games. Afterward the MD5 hash comparison the engines may also implement "fallback" detection which perform an engine-specific check for the files that may indicate a game is an unknown game or variant that runs on the engine. The SCI game detector implements this by checking for files such with names such as "resource.cfg", "resource.map", "resmap.000" or "Data1", which are either required or very common in games made with the SCI Engine. The SCI fallback detection also reads the "resource.cfg" file to make a guess about the games properties such as the interpreter version and game language, and creates a default configuration for the game. The results of each engines' checks are collated and presented to the user in a custom `ChooserDialog` that prompts them to select which variant of the game they intended to add from a list of potential matches. Their choice is relayed back to the `EngineManager`, which saves the default configuration data paired with the selected variant, and the settings dialog is opened for the game to allow the user a chance to customize or correct the settings if necessary.

Although much goes on the behind the scenes during the game setup process, the user is kept unaware of the complexity and multitude of checks. The only input required on their end before they can start playing is providing a directory and selecting the correct variant from a short list, narrowed down from potentially hundreds of games. The process also highlights the effort taken to reduce by developers to abstract common use cases and prevent code duplication across the many engines included in ScummVM.

## I Use Case: Loading a Save File at Runtime

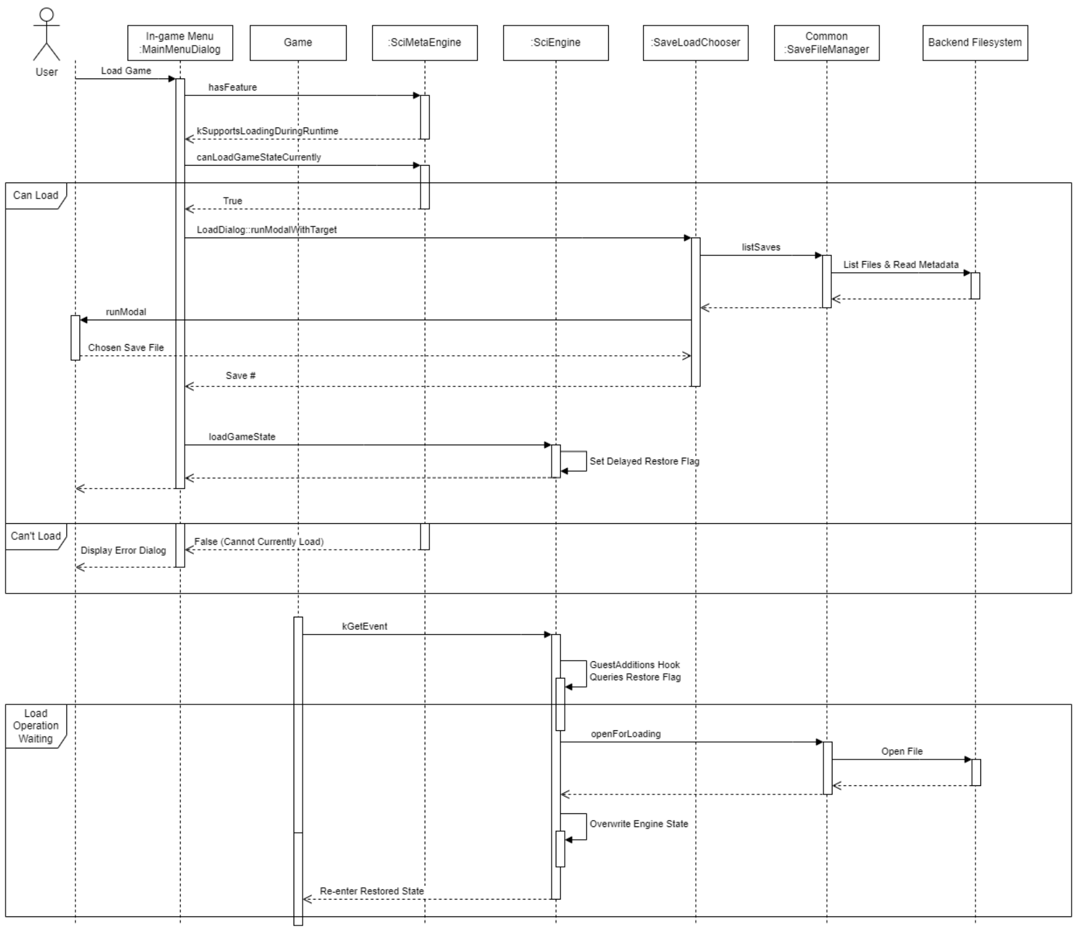

Figure 6: Sequence diagram demonstrating the process of selecting and a save file during runtime, and loading it within the SCI Engine

One of the advantages offered by ScummVM's unique approach to emulation is that the software has the opportunity to expand upon the features offered by the original engines. One of these extra features is the ability to load save files and restore a previous state while the game is running, even if the original game did not provide a way to do this. It also allows the user to switch between multiple concurrent save files in games which only supported a single save slot.

Loading a file at runtime is done through the ScummVM in-game menu, named `MainMenuDialog` within the codebase. The menu first checks the `MetaEngine` of the currently running game to see if it supports runtime loading, and whether the engine is in a state where it would allow overwriting the running game (e.g. In certain menus or cutscenes saving/loading would be prohibited). The user is prompted to select a save file from the GUI's `SaveLoadChooser` dialog, which consults the common `SaveFileManager` that all engines use to manage save files on a per-game basis. This shared code wraps backend filesystem access to abstract the file naming and management portions of the process such that the engines only concern themselves with interpreting the save data. Once the save file is selected the engine is requested to load it with `loadGameState`, which is the point in the process where engine-specific implementations diverge.

The SCI Engine chooses to defer the loading with a flag informing itself to switch at its earliest convenience. The game continues to run as if the user had not initiated the load, until one of the running game scripts invokes the `kGetEvent` function from within the interpreter. This function is normally used to poll the engine for input events that have occurred since the last call, but the internal logic contains a hook for the engine's `GuestAdditions`; a place where many of the ScummVM enhancements on top of the original engine are consolidated. Noticing the flag, the guest additions refer back to the `SaveFileManager` to open the save state. Interestingly, the saving and loading process makes use of a common combined serializer/deserializer (omitted from the sequence diagram for brevity) that utilizes the same code paths for both saving and loading! The save file contains a directly serialized copy of the engine state, which is read directly into the engine state variables, replacing the running game with the one in the save file. When control is returned to the game, it returns from the very same call to `kGetEvent` that gave `GuestAdditions` the opportunity to save the game previously, completely unaware what just occurred.

This type of situation is what makes the dependency between the engine and the game components bidirectional in the concrete architecture: certain behaviors of some engines like SCI require that a game is running and calling certain functions every so often to allow their hooks and logic to run.

## VIII. Lessons Learned

The most valuable lesson this project taught us was the virtue of project scheduling. When dividing up responsibilities for this project, the components of the report were split into 3 main assignments: research & report writing, presentation & script preparation, and recording & editing. The script and presentations were ment to be adapted from the well researched writings of the report. However, some members encountered unexpected delays that impacted initial research. This delayed not only our completion of the report, but also had a knock on effect on every other activity in the project. Since the recordings relied on this script and the script relied on the report, these three activities formed a critical path that had all been delayed due to these events. Since the recordings had to be made a fixed time before the due date so that they could be edited, the total slack in our project was now severely limited. What we have learned from this situation is to introduce more slack into our project pipeline. We'll need to ensure that the deadlines we set for ourselves are spaced far enough apart to minimize the impact of any unforeseeable delays.

## IX. Conclusions

Through this report we have explored the process and rationale we used to arrive at our concrete architecture for ScummVM and the SCI Engine. We have strengthened our understanding of ScummVM's layered architectural style and its top-level components. We have showcased all the ways in which our concrete architecture diverges from our conceptual architecture, alongside the reasons for those differences. We have described the intricacies of the object oriented style of the SCI Engine component, including its absences and divergences from its original concrete architecture. Finally, following the data flow through the source code, we have demonstrated two substantial use cases via sequence diagrams.

## References

[1] "SCI/Specifications/Introduction." [Online]. Available: https://wiki.scummvm.org/index.php?title=SCI/Specifications/Introduction

[2] "ScummVm." [Online]. Available: https://github.com/scummvm/scummvm/blob/master/engines/sci/video/robot_decoder.h

[3] "Developer Central." [Online]. Available: https://wiki.scummvm.org/index.php?title=Developer_Central