CISC 322/326: Conceptual 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 breakdown and analysis of the conceptual architecture of ScummVM, an open-source program that allows users across many platforms to play retro point-and-click adventure games from the 80s and 90s. Through the examination of official wikis, developer blogs, and the ScummVM repository, we determined that ScummVM uses a layered architectural style and can be organized into 5 main subcomponents: A platform-specific backend, ScummVM's base and main loop, a GUI for the launcher and options, game engines, and the common library. The control flow between these components supports the layered architectural style, as control moves between the backend and OSystem API, then to the ScummVM base, and then to the launcher GUI and engines. One of the game engines, the SCI engine is discussed in further detail. We determined this engine has a object-oriented architectural style, with game resources being represented as objects interacting with scripts and the graphics system. Data is able to capable of flowing freely between objects in the SCI engine due to the nature of its architecture. ScummVM has multiple instances of concurrency, primarily within the backends. The audio system may also run separate threads depending on the platform, and the engines can utilize some of the multi-threading capabilities of the backends. The most significant release versions of ScummVM are listed and described. The report contains two use cases with sequence diagrams that demonstrate the relations between components in certain scenarios; one for adding a game to the game library, and one for loading a save file during runtime. Finally, the roles and responsibilities of developers and contributes are discussed.

I. Introduction

As the years go by and computer systems rapidly evolve, the old games of many people's childhoods become inaccessible, but not forgotten. ScummVM is a program that allows people to play some of the classic graphical point-and-click adventure games and RPGs from the 80s and 90s on an extensive variety of platforms that the games were originally never designed for. The only requirements are that users already have the data files for the games they wish to play. One key factor that makes ScummVM special is that it is not an emulator, it is a complete rewrite of the games' executables.

ScummVM was originally created in 2001 to simulate 2 LucasArts adventure games that used the SCUMM engine (hence the name) on a Linux machine. As the news of this project spread, many other developers began to join to create support for other games, including those on other engines. Today, over 325 games are supported across roughly 21 platforms.

As a result of ScummVM being open source and its active development community, there is a wealth of documentation and resources available to gain a strong understanding of the system's inner workings and software architecture. The open source nature also allows analysis of the system's source code to further our team's apprehensions.

The purpose of this report is to identify and document ScummVM's high-level architecture via examination of the system's documentation online and within the repository. We discuss in detail our processes for deriving information about ScummVM, the system's overarching architecture, each of the system's subcomponents and how they interact, global control flow, instances of concurrency, ScummVM's architectural evolution over the years, and developer responsibilities. Diagrams are used throughout the report to illustrate how components are related and interact, and near the end we provide use cases of the system along with sequence diagrams. An important detail to note is that when discussing ScummVM's game engines in depth, we specifically focus on the SCI engine on top of some general information about all the engines. This is done because each engine has a unique structure.

II. Derivation Process

To get an understanding of ScummVM and the SCI engine, we started with the ScummVM official wiki's developer pages where the majority of documentation can be found. We found that ScummVM was initially released in late 2001 and is still being further developed by adding support for more games or implementing game engines. The scale of the ScummVM project is so large that other similar projects such as FreeSCI (now used in the SCI engine implementation) have merged with it over the years.

The majority of our knowledge on the architecture of ScummVM and the SCI engine came from the wiki, which specifies how the code base is structured into components and what they do. But there isn't as much info about interactions between components, so we found developer blogs that contain explanations of some subsystems and what role they have in the entire system. What stood out the most were presentations given by developers who had been invited as guest speakers at open-source conferences (Google's Summer of Code and linux.conf.au), which explain the relations between subsystems as well as other developer processes such as collaboration and logistical problems they faced.

The last place we looked for information was in the ScummVM GitHub repository, where we found more resources for developers, release notes, and other resources to give us some insight. We avoided looking at any source code or file structures so as not to mistake the actual (concrete) implementation for the conceptual architecture.

There isn't much in the way of alternative processes, but the process would have been much smoother if we had discovered the developer blogs sooner as they had a lot of information that wasn't mentioned anywhere else that gave us extremely valuable insight into the architecture, which would have saved lots of time. We also wanted to use the FreeSCI project's documentation since the implementation that ScummVM uses is based off of it, however the websites and documentation were taken down after FreeSCI merged with ScummVM.

III. Architectural Overview

ScummVM primarily follows a layered architectural style, with most engines taking an interpreter or object oriented form. The layers are the backend, game engines and the GUI, and the games themselves. Each layer has a well defined interface between the higher and lower layers. ScummVM relies heavily on the reuse principles of a good layered system, with some of their main requirements, including portability and the ability to support a wide variety of games, being implemented through this architecture. By changing a single layer, ScummVM can be built for many different systems or many different types of games.

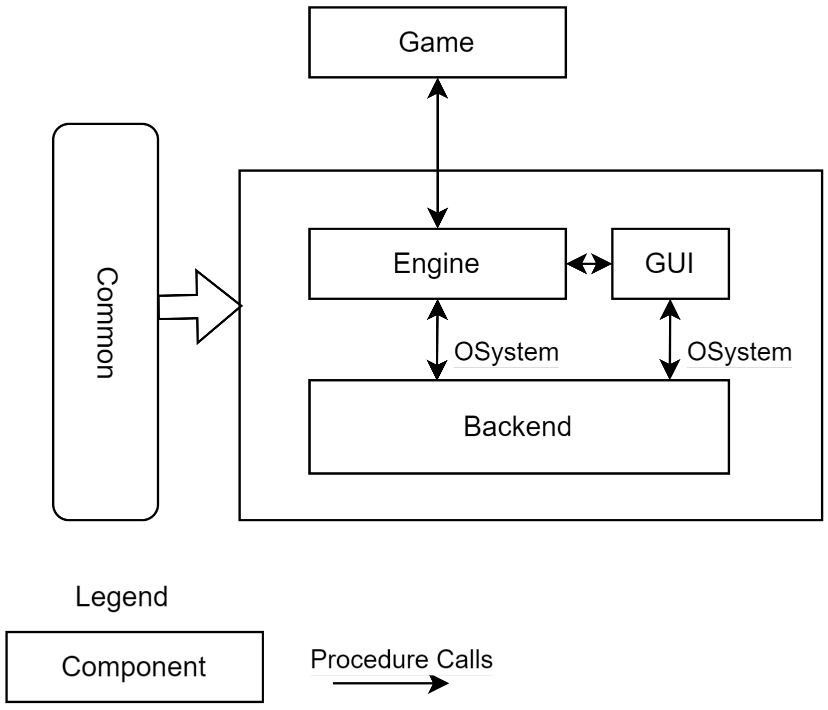


Figure 1: The layered architecture of ScummVM (without engine-specific details)

Starting at the lowest layer, we have the backend. Tasked with interfacing with the user's operating system (OS), this layer is the only portion of the system which is OS dependent and will be used to implement the common interface which the following layer can build off of. That interface is called the OSystem API, which is the only interface the rest of the system is guaranteed will exist on any platform. This layer achieves one of the primary goals of the software, which is to be portable to any platform people want to use. To port ScummVM to a new platform, all that has to be done is to create the backend which implements the OSystem API for the new platform and the program will work as intended. This makes use of one of the main advantages of this architecture, being the software reuse, so long as the backend interface is the same, the rest of the system can be perfectly reused on any platform.

The next layer is the GUI and Engine. The GUI component is simply the system launcher/configuration menu. A user will make use of the GUI to select and launch their game, which will call the engine launch procedure. The user can switch back into the GUI menu at any point by pausing engine execution to change settings or quit the game. The engine itself is slightly different from the other components found in the architecture. The exact implementation, and even the architecture of it varies drastically depending on the game/system being recreated. A typical implementation of this component will follow an interpreter architecture. It will take the game script and execute its bytecode using the engine execution state-machine. The engine is a complete re-implementation of the running game's main executable, rebuilding its engine from the ground up using the OSystem API to run the game on any supported system.

The final layer of the architecture is the game itself. The game data is read by the engine, used as the execution target of the engine state-machine, and save data is written back to the game save files. The engine provides an environment for the game to run which is nearly identical to the context the game was originally developed for. One remaining section to comment on is the common component. This is a collection of utility functions and common data types to be used while writing the engine, GUI and backend.

Since this software is designed to be run on any hardware, the ScummVM developers could not guarantee the existence of the C/C++ Standard library. Instead of relying on a standard library, the common code implements many of the standard functions and more in a completely portable form.

IV. ScummVM Subcomponents

I Platform Backend & OSystem API

As mentioned previously, the backend serves the role of making ScummVM more portable by abstracting away any reference to the OS. The backend is the only component in the whole system which interacts with and processes information related to the OS. The backend is a broad collection of implementations of the OSystem API for each OS that ScummVM has been ported to. Each backend generates a subclass of the greater OSystem class, which is the parent class for each platform's implementation of their platform-specific OSystem class. This platform-specific OSystem class defines OSystem API implementations specific to the given OS platform, as well as drawing in reusable API implementations from the common library. The platform-specific OSystem class defines the parameters of the OSystem API passed to the rest of the system, which includes API access to various OS resources and OS constraints which limit ScummVM functionality. Only one of these backend implementations is chosen at compile time, with the remaining backend implementations being excluded.

II ScummVM Base & Main Loop

The ScummVM base serves as an in-between for the engine or GUI and the backend. This subcomponent wasn't included in the architectural overview because its effect on the system is limited in scope. After the Backend is initialized it calls the base to run its main loop which alternates between running the engine directly and running the GUI. ScummVM can be used by either running games directly through the command line or through the GUI. The main loop within the base facilitates that functionality by allowing users to bypass the GUI altogether.

Additionally, the base serves to facilitate communication between the engine and the backend. When the main loop starts it ensures that the backend has been initialized correctly before collecting its implementation of the OSystem API and its limitations. The limitations are fed to the engine and the main loop then facilitates OSystem API communication between the engine and the backend.

III Launcher GUI & Options Menu

The launcher GUI is the primary way that most ScummVM users will interact with the system as a whole. The GUI serves as a frontend platform to manage launching games as well as other meta processes like managing game saves, keymappings and achievements. These meta processes are displayed through the GUI and are sourced from an engine's MetaEngine, which will be discussed more in the engine section. The GUI also operates a lightweight global options menu which is accessible at all times a game is running. This options menu allows users to preform meta engine actions while a game is running. These actions include saving or loading saves for the currently running game, modifying the games adjustable settings, returning to the launcher, and quitting the service altogether.

I Game Engine

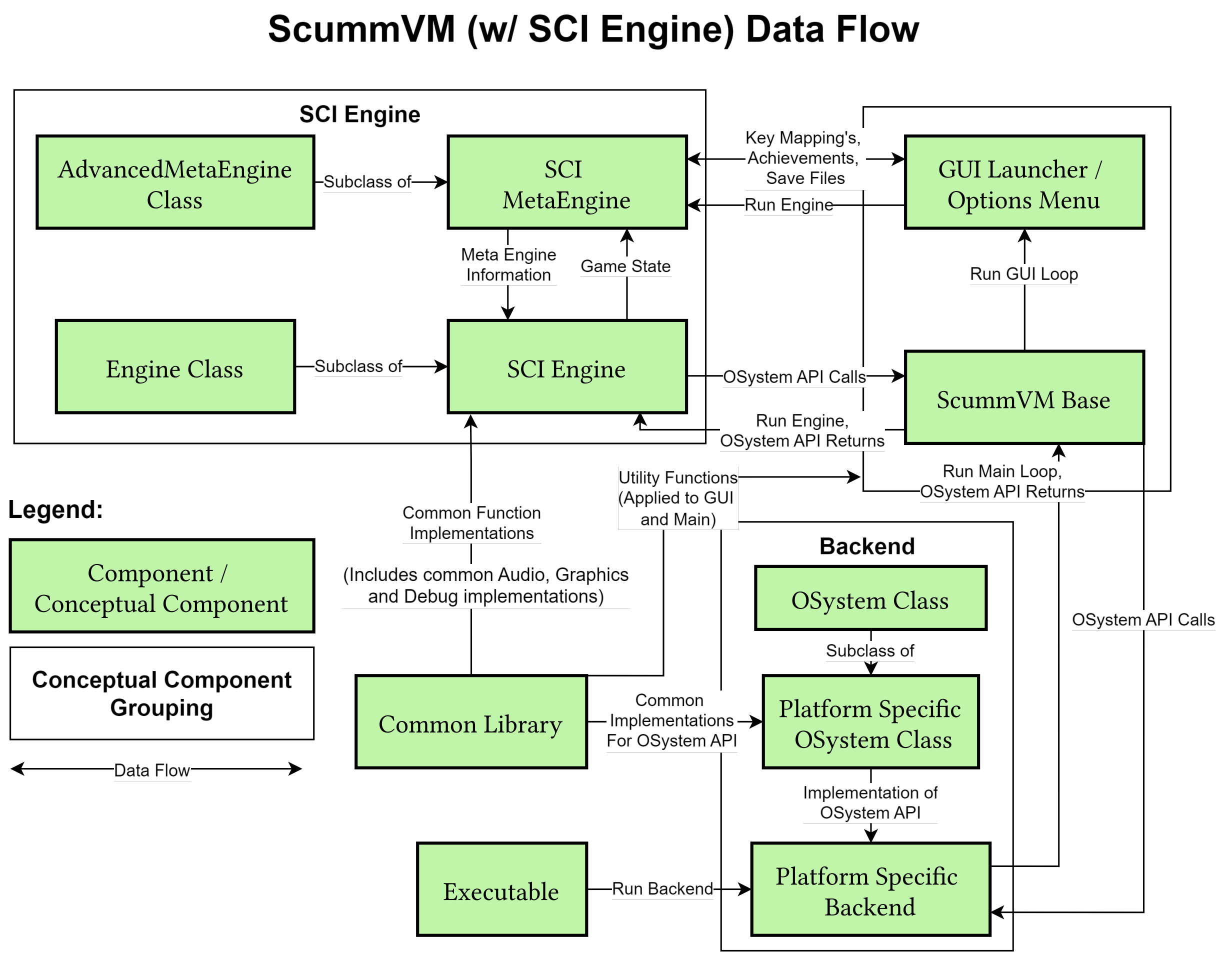
Within ScummVM are the implementations of numerous different engines. Each engine is uniquely developed to replicate a classic game engine, and provide an environment for the game's code to run. The engines are designed to closely mimic their respective original system's behavior with a much higher standard for portability. How individual engines are designed varies, but there are some standards shared amongst all engines. All engines must follow a simular structure via implementations of a couple classes which serves as the interface between the engine and the rest of ScummVM. An "Engine" subclass is implemented which allows the engine to properly interact with the OSystem API, and an "AdvancedMetaEngine" subclass is implemented to handle any data outside of the running game and provide communication with the launcher GUI.

The SCI engine uses an object-oriented architectural style. Game resources (sprites, fonts, videos, sounds, etc) are represented as distinct objects that scripts and the graphics system interact with, and scripts encapsulate their own variables and execution contexts within the engine. The engine itself consists of various singletons which neatly divide responsibility and orchestrate the script interpreter, memory allocation, graphics, and audio. These components manage the objects that make up the game and serve as a gateway to the OSystem API.

II Common Library

The common library is the abstract term for a collection of shared utility classes and codecs which are utilized across the project. These functions are written to be highly reusable, and are used in all components across the project. Depending on the platform some implementations of functions of the OSystem class draw from the common library. There are specialized media codecs for audio, image, and video processing which can be used by any of the engines to help with their implementations. The common library also includes a collection of tests for the utility classes.

V. Data and Control Flow



ScummVM (w/ SCI Engine) Data Flow
SCI Engine
AdvancedMetaEngine Class
Subclass of
SCI MetaEngine
Key Mapping's, Achievements, Save Files
GUI Launcher / Options Menu
Run Engine
Meta Engine Information
Game State
Run GUI Loop
Engine Class
Subclass of
SCI Engine
OSystem API Calls
ScummVM Base
Run Engine, OSystem API Returns
Utility Functions (Applied to GUI and Main)
Run Main Loop, OSystem API Returns
Legend:
Common Function Implementations
(Includes common Audio, Graphics and Debug implementations)
Component / Conceptual Component
Conceptual Component Grouping
Data Flow
Backend
OSystem Class
Subclass of
OSystem API Calls
Common Library
Common Implementations For OSystem API
Platform Specific OSystem Class
Implementation of OSystem API
Executable
Run Backend
Platform Specific Backend

Figure 2: The Data Flow Diagram of ScummVM (with minor expansion of sci-engine-specific details)

The data and control flow of the system is best described by returning to the concept of ScummVM as a layered system. Control starts in the backend, where the OSystem class is defined for a given system. Control over background OS processes and resources never leaves the backend, and instead the OSystem API is used to facilitate communication about the OS between the backend and all other systems. Control of non-backend services is then passed to the ScummVM base, which in turn passes control off to either an engine or the launcher GUI. If the GUI is run, it takes in meta data from each MetaEngine the user has interacted with before to grab info related to save data, achievements and key mappings. When a game is run, the GUI will pass along control to that game's respective engine. The data flow within each engine is unique to their specific implementation.

Within the object-oriented architecture of the SCI engine, game resource data flows very freely between components such as the script contexts, the memory manager, and audio/video systems. Data in the form of resource objects reaches the OSystem API solely through the graphics and audio components, and the interpreter engine is the gateway for inbound input and launcher GUI events.

VI. Concurrency

There are several instances of concurrency present within ScummVM, although there are limitations to which subcomponents are capable of multi-threading. First, it should be noted that the majority of ScummVM's components run together on a single thread. Additional threads (if any) are introduced by the backend or platform-specific audio libraries. Depending on the platform, the common audio system may use multi-threading for audio mixers, allowing any game on any engine to have an instance of concurrency. Otherwise, if the platform does not support the required audio libraries, the audio mixers will instead run on the main thread. The backends utilize multi-threading to handle timers and event queues; some individual backends may also have separate use cases for multi-threading. Many (though not all) of the engines have varying degrees of concurrency. Some of these engines have callbacks that utilize the multi-threading timers and event queues within the backends. The quantity of these callbacks in each engine and their functionalities differ. Due to portability issues (originating from the initial design of the engines and their respective games for single core systems), the engines themselves are unable to spawn new threads. However, some of them define their own threads as objects in order to simulate cooperative multi-threading and manage multiple scripts in a concurrent fashion while still running sequentially.

VII. Evolution

Since its initial version back in November 10, 2001, ScummVM has developed an extensive release history and is constantly being updated. As of October, 2024, ScummVM is on version 2.8.1, has over 150,000 commits have been made to the master branch of the repository, and roughly 10-20 commits are made per day. Every few months a new official release version is posted in the change log that summarizes the various (more important) changes that were made since the previous version. The majority of changes in each version typically consists of support for new games and various bug fixes and feature additions for a mix of ports, engines, and games. Occasionally, new ports and engines are introduced or existing ones undergo extensive rewrites. There have been many releases of ScummVM over the years, so the rest of this section will cover some of the versions with the most substantial changes to the system's overall architecture.

The first major release of ScummVM was version 0.1.0, released on January 13, 2002, although there was very little documentation written at this time. The only engine was the SCUMM engine and only 5 games were supported. There was very little abstraction around the engine or backend and it could only run on Windows and Linux systems.

A few months later on April 14, 2002, version 0.2.0 was released. The SCUMM engine was completely rewritten, auto-save was added, non-SCUMM games were added, and many new platforms were added including MorphOS, Macintosh, Dreamcast, and Solari, just to name a few. This release of ScummVM introduced the OSystem API for better abstraction to the backends, followed by version 0.3.0b adding better abstraction for the engines by separating their various parts into libraries.

The next major releases to have significant changes were versions 0.6.0 and 0.7.0 released on March 14, 2004 and December 24, 2004, respectively. Version 0.6.0 had many games added and a new engine, along with the SCUMM engine having many big fixes and improvements. More importantly, the audio code was rewritten for flexibility and performance and improvements were made for plugin capabilities. Version 0.7.0 had a partial rewrite of the backend.

0.11.0 released on January 15, 2008, had the internal access to files reworked, gained improvements to the was input was handled internally, and had ARM assembly routines added in the audio mixer and SCUMM video playback.

Version 0.13.0 released on February 28, 2009, had a complete rewrite for the GUI renderer and GUI configuration.

Version 1.6.0 released on May 31, 2013, had the video decoder system rewritten. Then in version 1.7.0 released on July 21, 2014, the build system was rewritten to be more modular.

After 1.7.0, there were no significant changes or rewrites that affected the architecture of ScummVM. This is to be expected at this point due to the system's age and size; almost all major architectural decisions/changes are made early into the development cycle. Several engines and backends were rewritten past (and before) this point but they do not alter ScummVM as a whole. It is still possible that architectural changes may be made in the future, however, they will require careful consideration and extensive work to be done, especially if they affect how the subcomponents interact.

VIII. Use Cases

I Use Case: Adding a Game to the Library

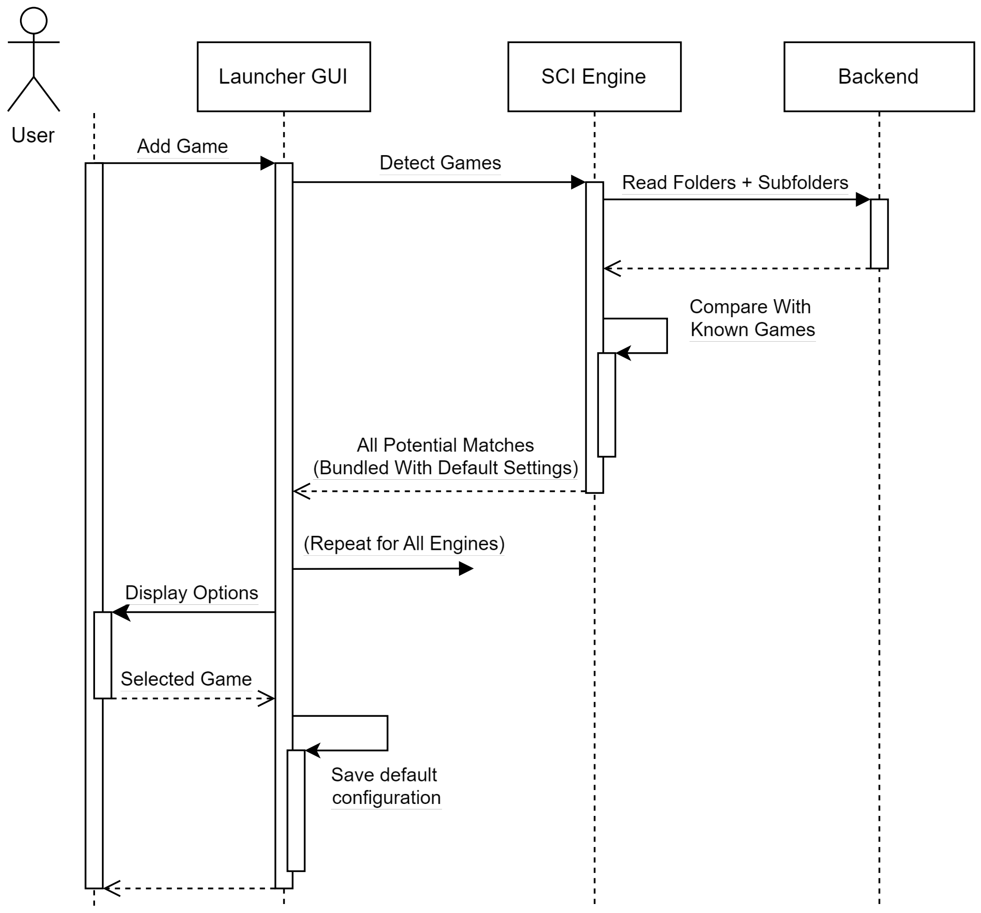


Figure 3: Sequence diagram depicting the process of recognizing supported games when adding to the library

When a user wishes to add a new game to their ScummVM library, they specify the directory containing the game files. The Launcher then calls upon the game detection routines of every included game engine, allowing each engine to compare the game files to a list of supported games. The engines are free to go about this however they choose using the platform backend to investigate the included files. The engines are built to be platform agnostic, so any interaction with the filesystem is performed through the OSystem API. It isn't always possible to determine the exact game from the files alone without creating false negatives for unknown variants of supported games, so the launcher collates possibilities reported by every engine and presents the user with a dialog to select the game they intended to add. The selected game is set up in the library with default configurations supplied by the engine, and the game is ready to launch. This system relies on the principle that very few new games will be created for engines old enough to be replicated in ScummVM, and the majority have already been documented. Automatically detecting the game in this fashion allows users unfamiliar with the internals of games and the relevant configuration settings to play just as easily as enthusiasts.

II Use Case: Loading a Save File at Runtime

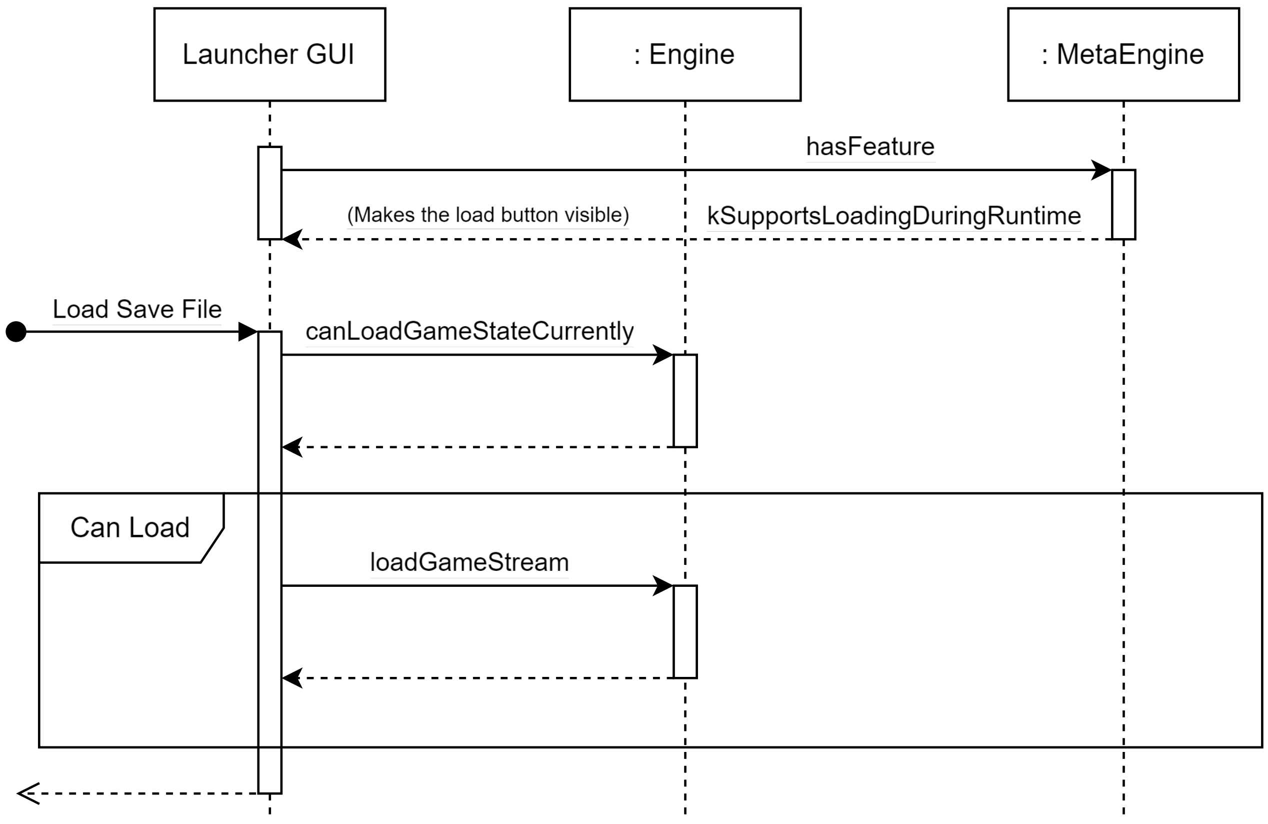

Launcher GUI
: Engine
: MetaEngine
hasFeature
(Makes the load button visible)
kSupportsLoadingDuringRuntime
Load Save File
canLoadGameStateCurrently
Can Load
loadGameStream

Figure 4: Seqeuence diagram showing ScummVM checking for engine support before loading

There are several optional engine features game engines can implement that add additional functionality to the launcher GUI for that engine's games. One of those features is the ability to load save files at runtime using the launcher GUI instead of reopening the target game or navigating in-game menus. Before the relevant buttons can be shown to the user ScummVM must consult the active engine's MetaEngine for the kSupportsLoadingDuringRuntime feature. The running engine can also hint to the system whether the game is in an appropriate state to save or be overwritten with another state, and the GUI can use this information to provide a visual cue to the user when loading is not possible. If the conditions to load a game state are met, the game engine will be provided an open handle to the file when the user clicks the load button and selects a save file, and the engine will execute its internal logic for swapping out the game state.

IX. Implications for Division of Responsibilities

ScummVM currently has more than 500 total contributors (past and present). To manage them all and ensure the development process is as smooth as possible, developers are assigned roles based on their responsibilities. These roles include the project leader and co-leads who oversee overall development and ensure the rules and conventions are followed and also give new developers permission to push to the GitHub repository. Apart from those two main roles, there are also engine developers who maintain and implement game engines, porters who implement support for more platforms and fix compatibility issues, and contributors who support the project in other ways such as working on the website.

The division of responsibilities is outlined in ScummVM's onboarding page and is very simple. Each developer has full ownership of the area they are working on—whether it's an engine, support for a game, or a backend — in which they can push changes freely. If a developer wants to make changes to another section, they only need to get permission from the developer in charge or make a pull request. To make changes to the OSystem API or common code, developers must consult the leads and porters.

Dividing responsibilities is extremely important to streamline the development process, this is why all developers have to follow some strict rules for writing and committing code regardless of responsibilities or ownership. For consistency, readability, and code cleanliness, developers must adhere to strict formatting conventions, there are also certain code conventions enforced due to compatibility issues that come from using certain features of the C++ language which would harm portability.

X. Conclusion and Lessons Learned

In this report we have explored the conceptual architecture of ScummVM through the examination of documentation available on the internet. The architectural style of ScummVM was identified to be layered, and the primary subcomponents consist of the platform backend and OSystem API, the ScummVM base and main loop, the launcher GUI and options menu, the game engines, and the common library. We have discussed the data and control flow between the subcomponents along with the system's instances of concurrency within the audio system, backends, and engines. The most significant releases regarding ScummVM's architecture were listed. Then finally, we provided use case examples with comprehensive diagrams and descriptions followed by an explanation of the responsibilities of developers.

Throughout the creation of this report, our group learned two major lessons: how to manage time better and communicate effectively as a group, especially in regard to assigning responsibilities to group members. There were times when we were forced to improvise certain sections of this report that could have been avoided had we established who was doing which sections earlier and communicated misunderstandings sooner. We now have the insight to avoid these issues in future reports.

References

[1] “Developer Central.” [Online]. Available: https://wiki.scummvm.org/index.php?title=Developer_Central

[2] “Team Onboarding.” [Online]. Available: https://wiki.scummvm.org/index.php?title=Team_Onboarding

[3] “HOWTO-Backends.” [Online]. Available: https://wiki.scummvm.org/index.php?title=HOWTO-Backends

[4] “Auto Detection.” [Online]. Available: https://wiki.scummvm.org/index.php?title=Auto_detection

[5] “HOWTO-Engines.” [Online]. Available: https://wiki.scummvm.org/index.php?title=HOWTO-Engines

[6] “Scumm Virtual Machine.” [Online]. Available: https://wiki.scummvm.org/index.php?title=SCUMM/Virtual_Machine#Threads

[7] ScummVM, “Scummvm/scummvm: ScummVM Main Repository.” [Online]. Available: https://github.com/scummvm/scummvm/tree/master

[8] J. "Ender" Brown, “ScummVM Retrospective: linux.conf.au 2017.” [Online]. Available: https://www.youtube.com/watch?v=QihSN7VCrB0

[9] ScummVM, “ScummVM Developer Blogs.” [Online]. Available: https://planet.scummvm.org/

[10] “ScummVM - Wikipedia.” [Online]. Available: https://en.wikipedia.org/wiki/ScummVM

[11] “SCI Wiki.” [Online]. Available: http://sciwiki.sierrahelp.com/