Conceptual Architectures of ScummVM and the SCI Game Engine

Authors:
Frankie Cao (frankie.c@queensu.ca)
Gavin Lacy (gavin.lacy@queensu.ca)
Aidan Tulk (21art8@queensu.ca)
Michal Wrobel (24lt12@queensu.ca)
Ayman Zaher (20az27@queensu.ca)
Selena Zou (20sz88@queensu.ca)

11 October 2024

1. Abstract

ScummVM is an open-source project, hosted on GitHub, that was first released in October 2001 by computer science student Ludvig Strigeus [1]. As a project, it seeks to provide users with functionality to run certain classic point-and-click adventure and role-playing video games on a variety of platforms, including systems for which the games were never designed [2]. However, unlike many other systems, which achieve cross-platform support via OS emulation (e.g., DOSBox for the DOS family of operating systems), ScummVM is not an emulator. Instead, given the original data files (graphics, audio, game scripts, etc.) for a supported video game, ScummVM attempts to fully rewrite the game's engine in C++. By replacing the original executables with ScummVM executables, cross-platform support is achieved. Other advantages included in running games through ScummVM are improved graphics, smaller file sizes, and sometimes better save-game systems.

In general, the ScummVM platform takes a structured approach to the division of labour between components. That is, the frontend (e.g., GUI and game engines themselves) are served by a group of common APIs, which in turn are served by the platform-specific interface with the operating system. In view of this hierarchical organization of services, ScummVM employs a layered conceptual architecture. The system is divided into five major subsystems (GUI, game engines, common APIs, backends, and the OSystem API), which are logically partitioned into three layers. The most interesting subsystem is the backends component, which implements the OSystem API for each operating system that ScummVM supports. It is this component which achieves true cross-platform compatibility. In this paper, we examine each subsystem in more detail, as well as the external interfaces and two major use cases of the ScummVM program.

In addition to the ScummVM platform itself, we also examine the SCI (Sierra's Creative Interpreter) game engine in particular. The SCI engine runs on top of the ScummVM platform to execute a subset of supported games. Due to the SCI engine's employment of a p-machine (i.e., virtual CPU) and interpreter to achieve portability, we choose an interpreter-style architecture to represent this subsystem.

2. Introduction and Overview

Named for the SCUMM (Script Creation Utility for Maniac Mansion) game engine which preceded it, ScummVM is a software system supporting cross-platform compatibility for classic point-and-click adventure and role-play computer games. Released in 2001, it supports a library of over 325 games from well-known studios, like LucasArts and Sierra On-Line. Supported games include *Blade Runner, Monkey Island, Broken Sword,* etc. ScummVM provides support for these games on a wide variety of computer and mobile operating systems, including Windows, Linux, macOS, iOS, and Android [3]. A full list of supported games and platforms can be found in the ScummVM documentation.

ScummVM assumes that any game executables that users may upload have been legally purchased [4]. Once given these original executable files, the system then completely rewrites these files in C++ and replaces the originals so that they are now compatible with the operating system of the user's machine. The original graphics, audio, and game scripts files are still required to play the game itself. Thus, in this sense, ScummVM itself is not an emulator. Rather than emulating any specific operating system, the platform's backends layer contains

platform-specific code for each supported OS, to be executed in order to make higher-level game engines compatible with the user's OS at runtime. As with any approach to software development, there are advantages and disadvantages to this. Advantages include portability (an obvious one), improved graphics due to the use of newer filters during rewriting, re-encoding audio files into popular modern formats, and improved systems for saving game progress. As a result of the rewriting of executable files, some bugs in the original executables may also be fixed. Also, because ScummVM is not an emulator, it also circumvents the issue of requiring excessive CPU and memory resources, as the games run directly through ScummVM and not through an emulated platform. However, one notable disadvantage is that *because* the engine of each ScummVM game is rewritten from scratch, new bugs may be introduced, even as old ones are fixed [2].

In order to run ScummVM, users are able to install the program from the ScummVM documentation, either by using an installer, or by installing manually. Once installed, users are able to upload their game data files to the ScummVM Launcher via the GUI. For the most part, ScummVM is designed to read game files from their original floppy discs or CDs; however, some games, such as select LucasArts games, are available in a digital form which the platform also supports [4], [5]. Users are able to launch games either via the Launcher in the GUI, or from the command line.

As a specific example of a game engine running on ScummVM, we examine the SCI engine. SCI, or the Sierra Creative Interpreter, is a stack-based virtual machine providing functionality for game features such as displaying graphics, playing audio, handling input and hard disk read/writes, and some complex arithmetic and logic. Generally, the SCI engine places game data in one of nine categories: script, vocab, patch, sound, cursor, view, pic, font, and text data [6]. Games supported by the SCI engine include *King's Quest* and the *Quest for Glory* series.

In this report, we discuss plausible candidates for the conceptual architectures of ScummVM and the SCI engine; ScummVM and the SCI engine are considered separately. Due to the hierarchical organization of services and responsibilities in the ScummVM platform, we conclude that it adopts a layered architecture. In order to demonstrate the layered nature of the ScummVM architecture, we further examine the five major subsystems of the platform, which form the salient layers: the OSystem API layer, the backends layer, the common APIs layer, the game engines, and the finally the GUI layer. By contrast, given the SCI engine's emphasis on portability and its use of a p-machine and interpreter, we conclude that it utilizes an interpreter architecture. For both the ScummVM platform and the SCI game engine, we also examine alternative candidate architectures and the reasons for which they were ultimately rejected. In the interest of relating both systems and their architectures to everyday users, we further discuss the external interfaces of the ScummVM system, as well as a few representative use cases, along with diagrams to facilitate understanding.

3. Architecture

3.1 Overview

Our analysis of conceptual architectures is largely concerned with the ScummVM platform itself and the SCI engine, which is a game engine that runs on top of ScummVM and supports games like King's Quest, Quest for Glory, etc. ScummVM and the SCI engine will be

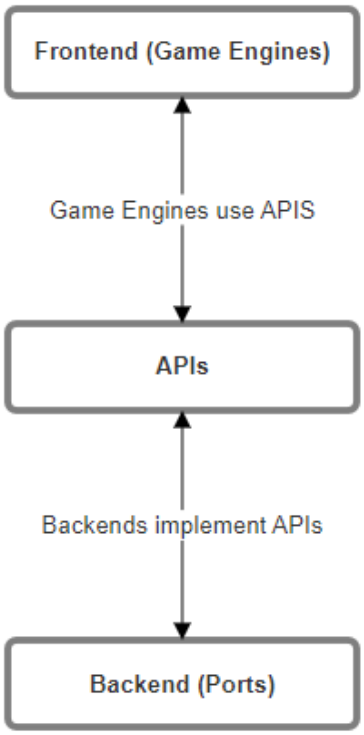
considered separately with respect to conceptual architectures, and the subsystems of the ScummVM system, with particular focus on the SCI engine as an example, will be discussed.

3.2 Conceptual Architecture Styles

3.2.1 ScummVM

The conceptual architecture of the ScummVM platform, like many other game engines, is best described as a layered architecture. A layered architecture is suitable for applications in which required services may be partitioned into a hierarchy of distinct classes [7]. For this reason, adventure game engine architectures commonly make use of layered architectures [8]. In the case of ScummVM, three overarching layers of components are implemented (from highest to lowest level) [9]:

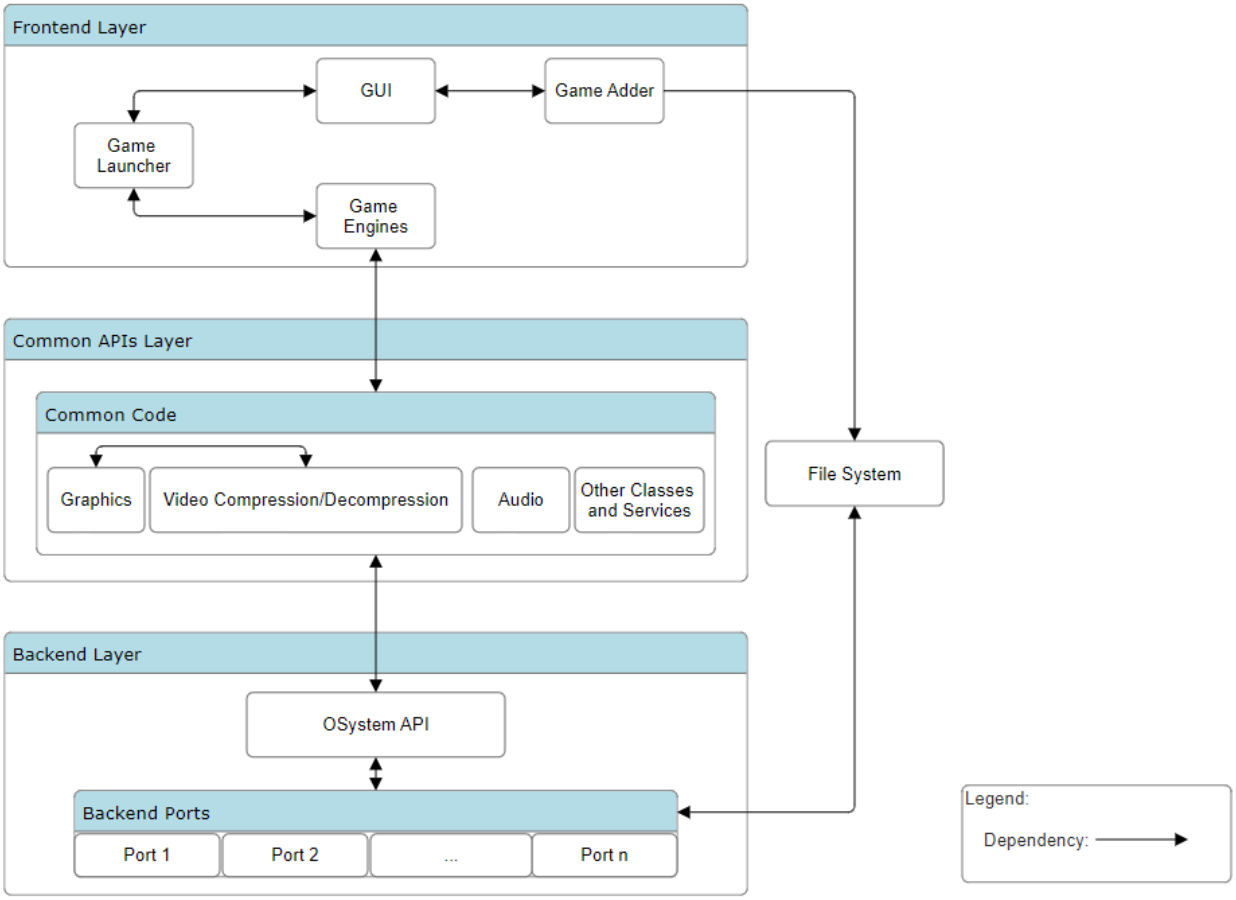
1. Frontends (or game engines), which are VMs that support a subset of ScummVM-supported games and their game-specific logistics;
2. Common APIs used by game engines for elements like graphics, audio, events and timing, written in cross-platform code; and
3. Backends (or ports), which implement the platform-specific code which underlies the common APIs.



Each layer is then subdivided into components, each of which is responsible for a specific purpose. According to the ScummVM Developer Central documentation [10], there exist five main components. These are:

1. The OSystem API, which defines the supported features available to a game (e.g., keyboard-interrupt events). This component belongs to the backend layer.

2. The backends, which implement the OSystem API for different platforms. This component also belongs to the backend layer.
3. The common code for various utility classes (e.g., containers) and services like graphics, audio, and video compression/decompression available to game engines. This component belongs to the common API layer.
4. The game engines themselves, each of which supports a specific subset of games. This component belongs to the frontend layer.
5. The GUI, which provides a graphical user interface for downloading and launching games. This component also belongs to the frontend layer.

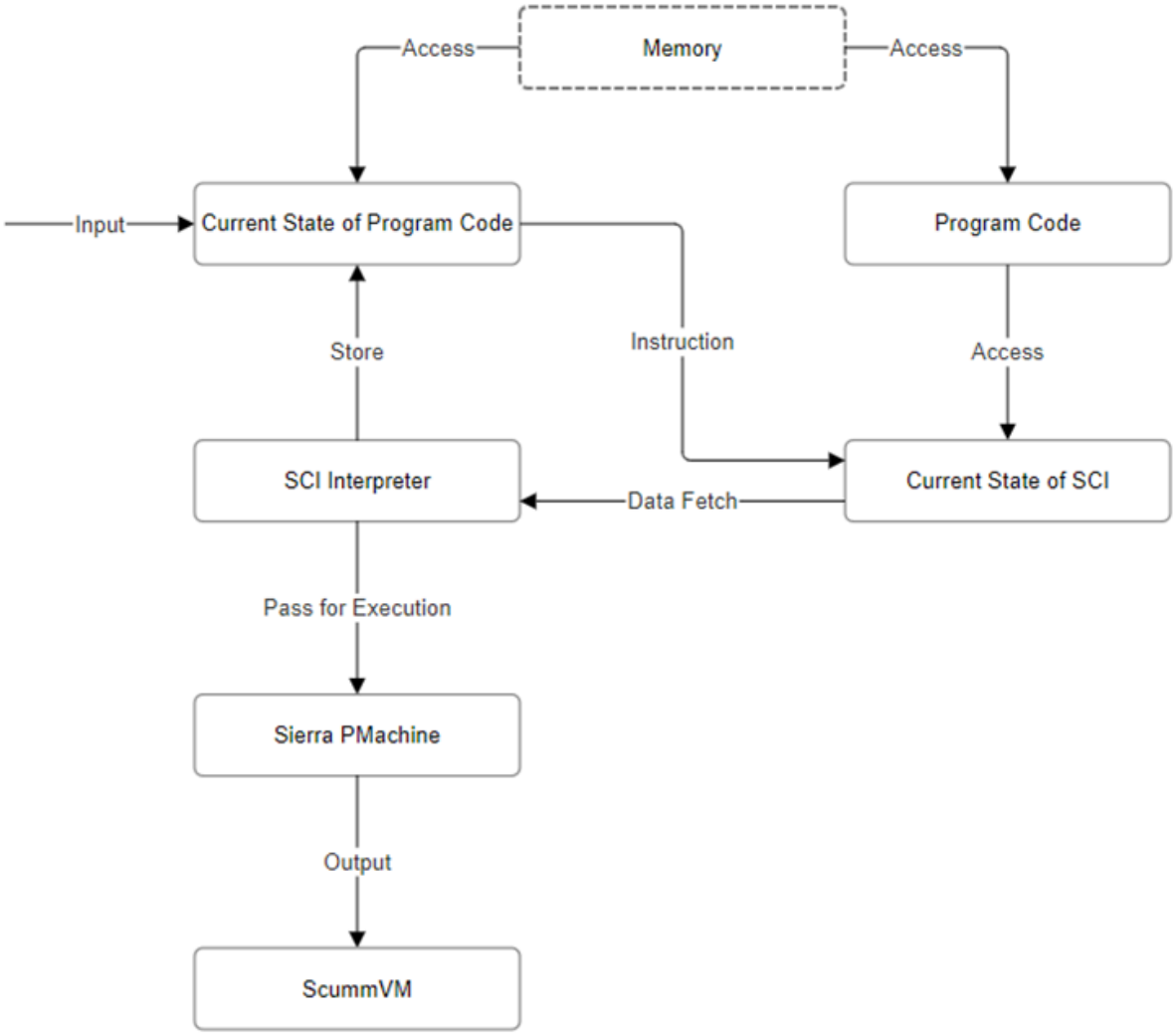


In adopting a layered architecture, ScummVM separates the concerns of portability (addressed by the backend layer) and adding support for new games and game engines (addressed by the frontend layer). Thus developers are free to choose a subset of the system with which to concern themselves, rather than having to consider the complete system.

3.2.2 The SCI game engine

The conceptual architecture of the SCI game engine uses an interpreter style. In general, interpreter-style architectures are ideal for applications in which "the most appropriate machine for executing the solution is not directly available" [7]. In the case of the SCI engine, the ScummVM Wiki [11] describes it as a "p-machine-style virtual machine for executing platform-independent, object-oriented code." There are two

overarching components: the Sierra Creative Interpreter (SCI) itself, and the Sierra PMachine. The Sierra PMachine is a virtual CPU that executes SCI programs; the design of this virtual CPU is specialized for object-oriented code [12]. Similar to the Java language architecture, the SCI first interprets SCI program code, then passes it on to the Sierra PMachine (i.e., the execution engine) to execute in its virtual environment. In this way, the SCI engine is able to simulate execution on hardware designed for object-oriented programming, and also facilitates portability of the game engine across a variety of platforms — i.e., the main purpose of ScummVM.



The SCI game engine utilizes this architecture to process nine different resource types allowing games to easily play sounds, display graphics, accept input, perform file operations, and perform any of the 125 basic opt codes provided by the Sierra PMachine [13].

Since games designed to run utilizing the SCI engine were intended to be loaded into the system using a series of floppy disks, the game data needed to be split up into multiple chunks. To accomplish this the SCI engine loads game data from nine different resource types: scripts, vocab, patch, sound, cursor, view, pic, font and text. Script resource files contain the code required to run the game, including the entry point into the game, lookup tables for required resources, event handlers, and any other code required to run the game. The SCI engine initially loads the entry point script from the first floppy disk and utilizes the resource lookup table included to identify which floppy disks need to be inserted further to load all of the data required by the game. During runtime of the game, the scripts are utilized by the engine to determine when to play sounds, show graphics, accept input, etc. and what resources to utilize in order to do so. Sound data and sound configuration are loaded into the engine utilizing the patch and sound resource files and are controlled by scripts to play sounds as needed. Graphical elements of the game are stored in cursor, view, pic and font resource files and serve a different purpose. Cursor files contain images to utilize for the cursor, view files contain sets of images to be used in the game, pic files contain background images, and font files contain fonts to be used when displaying text. Text resource files include text displayed throughout the runtime of the game and are displayed using the fonts included in font resource files. Finally, vocab files are utilized to configure the SCI engine's parser and debugger to ensure it can parse the included scripts correctly and provide the correct debug information for the current environment [13].

3.3 System Evolution

As mentioned before, ScummVM was initially released in 2001 by developers Ludvig Strigeus and Vincent Hamm. Initially designed only to implement a system player for the SCUMM game engine on different platforms, it first supported the classic games *Monkey Island 2* and *Indiana Jones and the Fate of Atlantis*. Shortly afterward, the project grew to support additional games and engines (for example, the SCI engine examined in this paper). By 2002, the ScummVM development team had shifted the platform's code from C to C++ (possible because C++ is in many ways a superset of C) and added a GUI to improve the experience for non-technical users. Since then, over 300 games have been added to ScummVM support; in particular, those developed by LucasArts and Sierra On-Line (necessitating the implementation of the AGI and SCI engines) were in high demand [1].

ScummVM continues to expand its support for new games and engines. This is made possible by the selection of a layered conceptual architecture; by definition, layered architectures lend themselves well to enhancements and additions of new components (e.g., game engines in the frontend layer) because each layer generally interacts with only the one directly above and below it, and so changes to one layer impact at most two others rather than the whole system [7]. In the case specifically of the SCI engine, evolution and enhancement are also facilitated by the choice of an interpreter architecture—the game engine is already there, providing backend support and portability; developers have only to add game-specific support for each new game. This task, although nontrivial, is much simplified from having to implement a game engine from scratch.

3.4 Concurrency

Little evidence of concurrency is available in the ScummVM documentation. The games ScummVM aims to support are older, usually from the 1980s-1990s. Thus, these games were written for sequential execution on a single processor. As a general rule, game engines, especially older ones, tend to be optimized for single-thread performance, because most processing runs in a single chain of dependencies. As a matter of fact, the ScummVM main loop mimics concurrency in that it updates the state of the game at regular time intervals (quanta), during which it breaks down large, long-running tasks (e.g., graphics, audio, input processing) into small pieces which are interleaved in order to maintain synchronized progress [10].

3.5 Subsystems

The ScummVM platform has five major subsystems, which are grouped into three layers.

3.5.1 The OSystem API

The OSystem API defines the available features and services a game engine can use. It shields the higher levels (utility classes, game engines, and GUI) from platform-dependent specifics and allows for game engine development abstracted from individual platform concerns [10].

3.5.2 The backends

The backends represent the actual implementation of the OSystem API for each platform that ScummVM supports. It is the only layer of the ScummVM architecture that is dependent on hardware. The platform-specific backends are generally implemented in one of two ways: monolithic or modular design [10]. While monolithic backends are self-contained and independent from other applications, modular backends split the OSystem API implementation into multiple modules and use the Simple DirectMedia Layer (SDL) library to implement low-level access to audio, keyboard, mouse, joystick, and graphics hardware [14].

3.5.3 Common utility classes

The common code provides utility classes for higher-level game engines, as well as services for audio, image, and video compression/decompression [10]. This layer provides a general representation of the class hierarchy and the relationship between classes.

3.5.4 Game engines

In general, each game engine runs on top of the overall ScummVM architecture and supports a certain subset of games. This section will focus in particular on the SCI game engine.

The SCI game engine was added to ScummVM with the release of v1.2.0 in 2009, and currently allows playing many of the games originally designed for the SCI game engine in the 1980s and 1990s. In order to do so, the game engine was rewritten to make use of ScummVM's OSystem API, allowing it to run on any platform ScummVM supports. The ScummVM SCI engine loads and plays games in the same way the original SCI engine did (as described in Section 3.2.2), allowing it to be fully compatible with games

developed for the original engine. The original SCI engine was developed to run on MS-DOS and Windows, and utilized the API provided by the platforms to perform interactions with hardware such as playing sounds, displaying graphics and performing file operations [15]. The ScummVM SCI engine, however, utilizes the OSystem API included in ScummVM to perform the same interactions with hardware. By rewriting the SCI engine in this way, games originally developed for the SCI engine can be played on any platform for which ScummVM has a current or planned future backend implementation.

3.5.5 The GUI

The GUI provides an interactive interface for the user to add games, select options, and launch games.

3.6 Alternative Architectures Considered

For ScummVM, we initially also considered an interpreter-style architecture, primarily because of the keywords in the ScummVM general description [16]: "ScummVM… [allows] you to play games on systems for which they were never designed." The implication that ScummVM may simulate non-implemented hardware in order to execute certain games led us to investigate the interpreter possibility. However (although we eventually adopted it for the SCI game engine), we rejected this architecture style for the ScummVM platform because of additional information published on the Developer Central page about the program's code and services structure. This description indicates a strong hierarchical (i.e., layered) organization of services and responsibilities, with the OSystem API at the lowest layer interfacing with the user's operating system, the GUI allowing for user interaction at the highest layer, and common utility code and game engines in between. Additionally, we found no mentions of an interpreter and execution engine, which are common indications of an interpreter-style architecture. Therefore we concluded that ScummVM uses a layered conceptual architecture.

For the SCI game engine in particular, we briefly considered an implicit invocation (i.e., publish-subscribe) architecture. Instinctively, one thinks of the SCI game engine as a sort of plug-in that runs on top of the overall ScummVM system and does not necessarily know the platform-specific implementation details of ScummVM. Given that a game engine must also be able to signal, receive, and process events, we studied the documentation for hints of a pub-sub style. However, pub-sub architectures possess one disadvantage: the processing of broadcasted events may not occur synchronously, or in a specific order [7]. In the case of a game engine, it is possible that components raising events expect a very specific order of responses from other components in order to continue the game in a sequence that makes sense to the user. This, combined with the discussion of the SCI interpreter and Sierra PMachine in the official documentation, led us to reject a pub-sub architecture and adopt an interpreter-style architecture for the SCI engine.

4. External Interfaces

4.1 Game Files

The files of the games are not included with the installation of ScummVM [16]. The user must have the data files from the game on hand to be able to use them. This includes practically everything about the game, which includes the media files and sound files, localization files,

configuration files and more. The media files are the various assets used in the game like the character sprites, background textures, anything visual involved with the game engine. This is the same for the sound files but for audio instead of visual assets. The localization files can be the various text files with multiple languages so that the game can run. The final one is the configuration files, which are used by the engine to actually run the game.

4.2 Cloud services

Users are allowed to connect ScummVM with a cloud based storage system to perform both syncing of saved games, and downloading of games from the cloud [17]. ScummVM allows the user to carry their saved games across the cloud to multiple systems, allowing users to play their games across a large variety of devices. It also allows the user to download game files from the cloud, meaning that game files are not stuck to one device.

4.3 Local Web Server

If the cloud is inaccessible to the user, or if the cloud is simply not preferred, then the user can use a local web server to perform many of the same functions [18]. The local web server can transfer save data between devices, and can also allow users to download games from a different device.

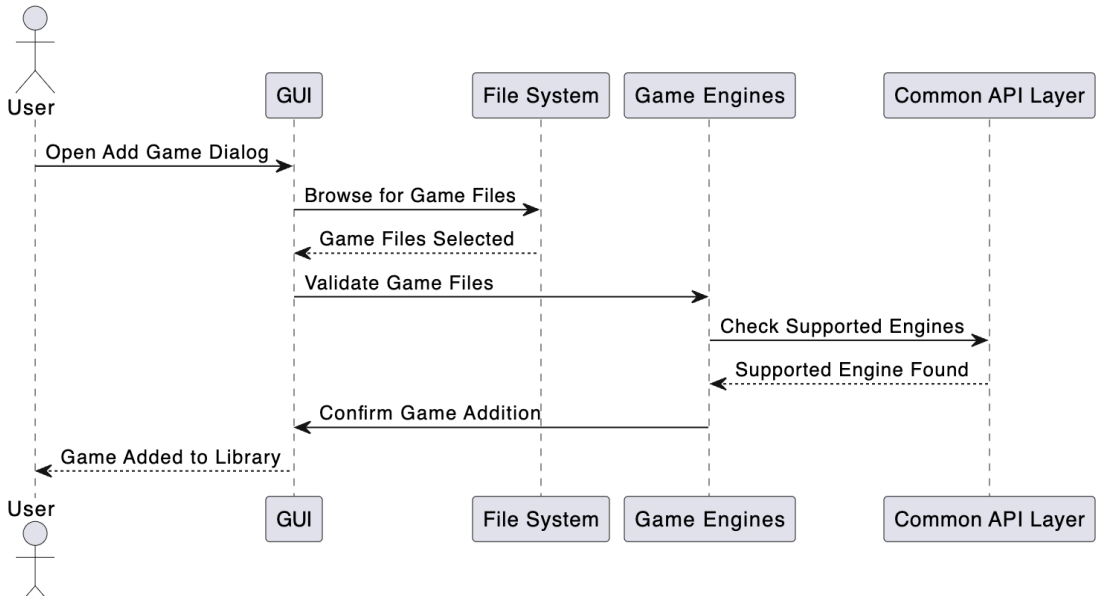
5. Use Cases

5.1 Adding A Game

The most obvious use case of ScummVM is to play a game, but before this is possible the game needs to be added to the virtual machine. To do this the user must interact with components such as the GUI, File Systems, Game Engines, and Common API Layer. According to the conceptual architecture which we contrived, these interactions are facilitated through procedure calls, which is consistent with how a layered architectural style behaves.

Firstly, to initiate this process the user must interact with the frontend GUI and navigate their computer's file system to select the game they wish to add. Once this is complete, the game files of the selected game are retrieved by the File System component of the conceptual architecture. The system then needs to ensure that there is a supporting engine which the game is compatible with and this is done by the Common API and Game Engine layer. If the game files are valid and a supported engine is found then the game will be added to the library of the user.

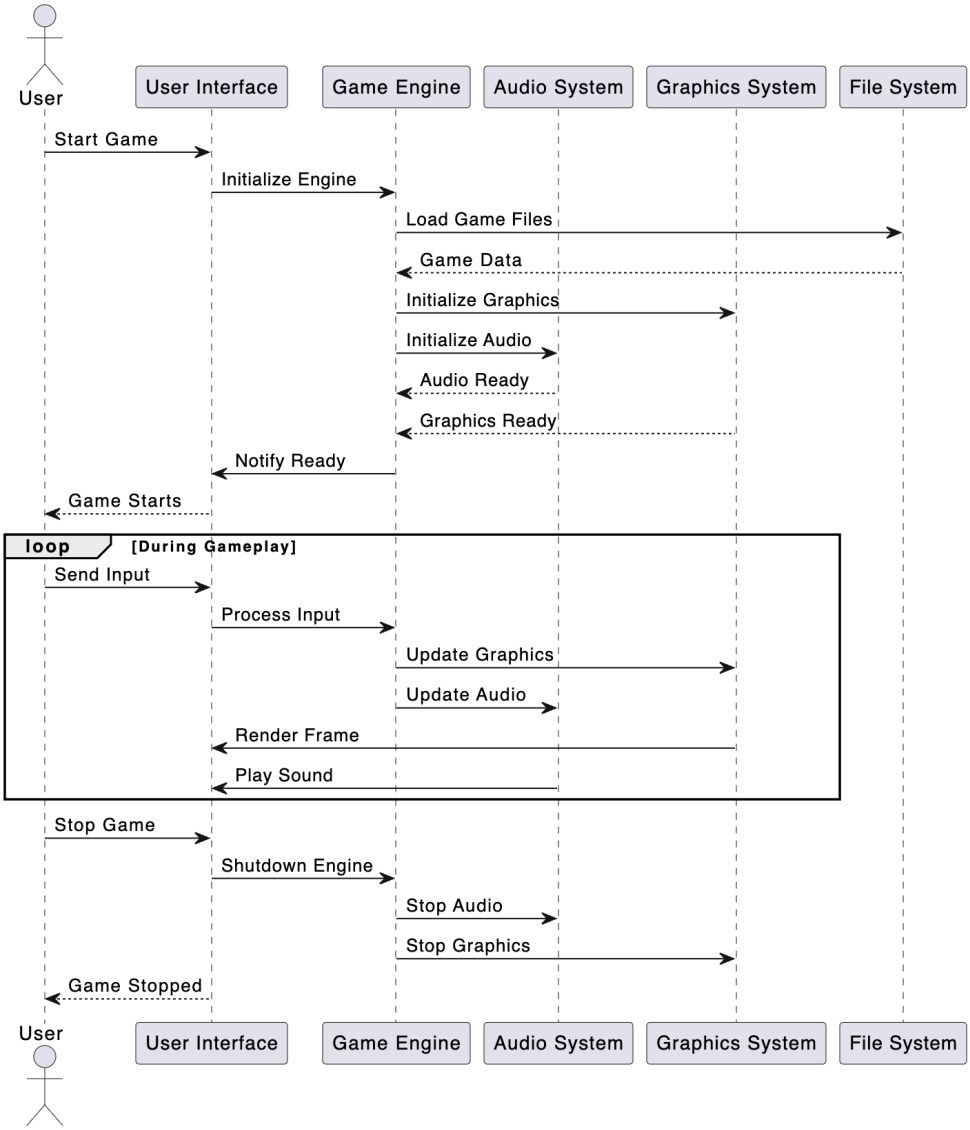
This is a barebones abstraction of the process of adding a game that is just meant to illustrate the process. There are other considerations that the system has to make when retrieving the game files and adding the game to the user's library. For example, depending on the operating system, the methodology for file retrieval may differ but the idea is fundamentally the same.



5.2 Playing A Game

After a game is added, the most important use case of ScummVM can be performed; playing the game. The process of playing the game from an architectural standpoint is more complicated. First, the user needs to initialize the game by calling the game engine through the User Interface. The game engine then interacts with components within the API layer to initialize the game data, depending on the engine it should also be able enhance the source audio and graphics.

The user plays the game through a user interface which continuously calls the graphics and audio systems depending on the input and the current state of the game. This is obviously a generalization but illustrates how the frontend and API layer interact in order to emulate these games. After execution of the game is completed, the system shuts down the engine and related systems to free processor resources.



6. Conclusions

From the available documentation on ScummVM and general adventure game engine architectures, we conclude that ScummVM adopts a layered architecture. Significantly, this choice of architecture enables the abstraction of ScummVM's higher-level game engines and GUI (and even, to an extent, the common API layer) away from the platform-specific implementation details of the backends and OSystem API layers. Thus, the burden of achieving portability (addressed in the backends and OSystem API layers) is effectively separated from that of implementing support for new games or game engines (addressed in the game engine and GUI layers). This separation of concerns facilitates system evolution, as well as enhancement.

We also conclude that the SCI engine uses an interpreter-style architecture. Our research here is somewhat hampered by the fact that most SCI documentation refers to an older version (SCI0 instead of SCI1), but nevertheless the purpose of the SCI engine, combined with its use of a p-machine and interpreter, constitute sufficient evidence to establish its interpreter style. This

choice of conceptual architecture greatly facilitates portability, which is, by definition, a core concern of ScummVM systems and subsystems like SCI.

Future directions for research include other game engines (for example, the also-popular Adventure Game Interpreter developed by Sierra On-Line) and further exploration of the ScummVM common API layer, particularly the degree to which it is shielded from the platform-specific details of the backends. Like most software projects, ScummVM and the SCI engine are under- rather than over-documented. Thus, future avenues for research on these two topics lie largely in examining the codebase—which we will do in our next report, which will discuss the systems' concrete architectures.

Appendix

7. Data Dictionary

Term	**Definition**
Game engine	A system that provides libraries and APIs to be used by developers to ease the creation of computer-based games.
Emulator	A software system that allows a computer to behave similarly to another one.
Port	A software system that has been converted or translated to run on a system it was not originally designed for.
P-Machine	A virtual machine designed to execute portable code regardless of the host platform.
Interpreter	A software system that directly executes instructions that have not been compiled for the platform.

8. Naming Conventions

Abbreviated Term	**Full Form**
VM	Virtual machine
API	Application programming interface
GUI	Graphical user interface
SCI	Sierra Creative Interpreter (occasionally also “Script Code Interpreter” in older SCI docs)
AGI	Adventure Game Interpreter

9. Lessons Learned

Apart from looking directly at the ScummVM source code (which is more relevant to researching the concrete rather than conceptual architecture), we suggest that downloading and running ScummVM ourselves would have given us more opportunity to familiarize ourselves with the structure of the program from a user perspective. Also, generally speaking, more practice investigating and identifying conceptual architectures from project documentation would have been useful in the research stage of this report.

References

[1] R. Moss. “Maniac Tentacle Mindbenders: How ScummVM’s unpaid coders kept adventure gaming alive.” Ars Technica. Accessed 11 October 2024. [Online]. Available: https://arstechnica.com/gaming/2012/01/maniac-tentacle-mindbenders-of-atlantis-how-scummvm-kept-adventure-gaming-alive/.

[2] “About ScummVM.” ScummVM Wiki. Accessed 11 October 2024. [Online]. Available: https://wiki.scummvm.org/index.php?title=About.

[3] “What is ScummVM?” ScummVM. Accessed 11 October 2024. [Online]. Available: https://www.scummvm.org/.

[4] “Adding and playing a game.” ScummVM Documentation. Accessed 11 October 2024. [Online]. Available: https://docs.scummvm.org/en/latest/use_scummvm/add_play_games.html.

[5] “Where to get the games.” ScummVM Wiki. Accessed 11 October 2024. [Online]. Available: https://wiki.scummvm.org/index.php?title=Where_to_get_the_games.

[6] “SCI specifications - SCI Wiki.” SierraHelp. Accessed 11 October 2024. [Online]. Available: http://sciwiki.sierrahelp.com/index.php/SCI_Specifications.

[7] A. E. Hassan and B. Adams. (2024). Module 03: Architecture Styles [PowerPoint slides]. Available: https://onq.queensu.ca/d2l/le/content/959322/viewContent/5711378/View.

[8] M. Uurloon. “Design of a point and click adventure game engine.” Groebelsloot.com. Accessed 11 October 2024. [Online]. Available: https://www.groebelsloot.com/2015/12/01/design-of-a-point-and-click-adventure-game-engine/.

[9] zorbid. “Programming a new game.” ScummVM Forums. Accessed 11 October 2024. [Online]. Available: https://forums.scummvm.org/viewtopic.php?t=7886.

[10] “Developer Central.” ScummVM Wiki. Accessed 11 October 2024. [Online]. Available: https://wiki.scummvm.org/index.php?title=Developer_Central.

[11] “The Sierra PMachine.” ScummVM Wiki. Accessed 11 October 2024. [Online]. Available: https://wiki.scummvm.org/index.php?title=SCI/Specifications/SCI_virtual_machine/The_Sierra_PMachine.

[12] “SCI.” ScummVM Wiki. Accessed 11 October 2024. [Online]. Available: https://wiki.scummvm.org/index.php/SCI.

[13] “SCI specifications.” SierraHelp. Accessed 11 October 2024. [Online]. Available: http://sciwiki.sierrahelp.com/index.php/SCI_Specifications.

[14] “About SDL.” libsdl.org. Accessed 11 October 2024. [Online]. Available: https://www.libsdl.org/index.php.

[15] “Sierra Creative Interpreter.” SierraHelp. Accessed 11 October 2024. [Online]. Available: http://sciwiki.sierrahelp.com/index.php/Sierra_Creative_Interpreter.

[16] “What is ScummVM?” ScummVM. Accessed 11 October 2024. [Online]. Available: https://www.scummvm.org/.

[17] “Connecting a cloud service” ScummVM. Accessed 11 October 2024. [Online]. Available: https://docs.scummvm.org/en/v2.8.0/use_scummvm/connect_cloud.html.

[18] “Using the local web server” ScummVM. Accessed 11 October 2024. [Online]. Available: https://docs.scummvm.org/en/v2.8.0/use_scummvm/LAN.html.