Conceptual 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

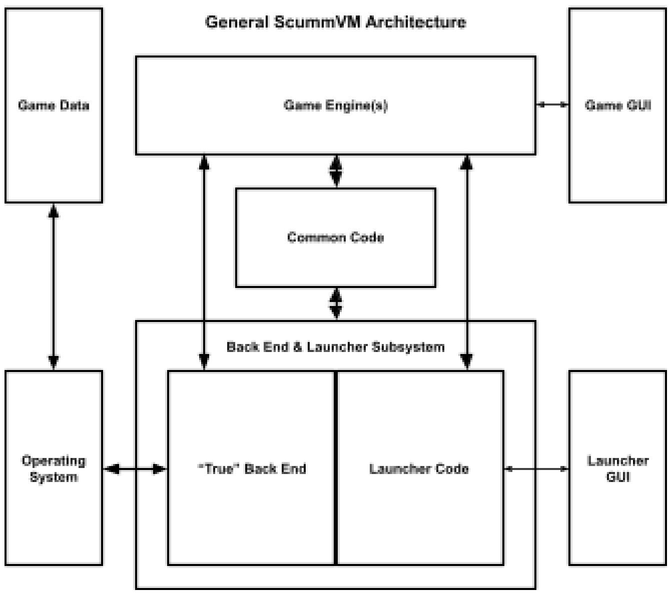
This report will cover a conceptual software architecture analysis of ScummVM, an open-source application that enables classic point-and-click games from developers to run on modern platforms by replacing original executables. This report will provide an introduction to the application and a preliminary, broad understanding of its software architecture before delving deeper. The report will then further explore the subsystems of the large systems that are identified in the introduction of this report (see Figure 1). Each of these descriptions will briefly touch on various related topics, such as interactions between components of the subsystems in question and their relevance to concurrency, adaptability, and performance. It will also provide a simplified description of their function within the ScummVM software. Next, the report will further examine the interactions of these three major components identified through the lens of several key concepts of software architecture. Following this, this report will demonstrate a few of the use cases ScummVM users will most commonly experience. Overall, key findings emphasize that ScummVM's layered architecture supports high performance, even on resource-constrained devices, by streamlining data access and reducing system load. The report

explores how ScummVM's modular architecture is not only suited to running classic games but also adaptable for future expansion as a means to preserve legacy software.

Introduction

ScummVM is an application which allows the user to run and play classic LucasArts video games, as well as video games from other classic game companies from the late 90s and early 2000s, such as Sierra. ScummVM is a recreation of the various proprietary scripting languages and engines, with a standard framework to allow them to run on various different platforms. ScummVM can be broken apart into two main parts: the game engines, and the rest of the systems required to support them (as well as the launcher). The further subdivision of systems lends ScummVM to being represented with a layered conceptual architecture, with the general structure being shown on the right (see figure 1).

Figure 1



Although there is not as strong a division of different layers as often found in layered architectures, it is still an effective way of conceptualizing the program. There are a few main reasons why:

- Components have different interchangeable identities (from a functional perspective)
    - There are multiple different versions of the "true" back end of ScummVM, one for each platform or operating system.
    - There are multiple different engines, all of which have the same contract with the other components (except the Game GUI, which is usually unique to the engine)
- Components are the divisions between sections of functionality that communicate less. These processes between components include:
    - The back end facilitates communication with the operating system
    - The launcher code passes control back and forth with the game engine to start and stop running a game
    - The common code provides tools and resources to the engine and launcher

By dividing the architecture into these modular layers, ScummVM has more flexibility for different components to change independently throughout development. This also makes it an easier project to get involved with as an open source project. As an example, developers concerned with porting ScummVM to a new system do not need to concern themself with areas of code other than the "true" back end, as each layer is largely distinct from each other. Similarly, developers of a new engine for ScummVM only need to define the implementation of the engine, not the entire application. Documentation needs to focus on explaining the purpose of each subsystem, and how they are supposed to interface with other subsystems.

In some cases, the organization of these layers adds middlemen to communication, e.g. when a game save is loaded, it passes from the operating system, through the middleman of the "true" back end, to the game engine. But this moderate increase in latency is worth the ease of development. Especially considering that most of the games ScummVM runs are simple point-and-click games (i.e. rather lightweight), the overall application performance is not threatened greatly by some extra latency. There are other consequences of the layered style ScummVM uses. The passing of data from component to component constrains the performance of the entire program to central layers, which means the application will only be as fast as the slowest component. Because there is a clear division between different layers, it is easier to isolate the source of any bugs, although the different layers are more likely to cause bugs due to the way they integrate with each other.

Architecture

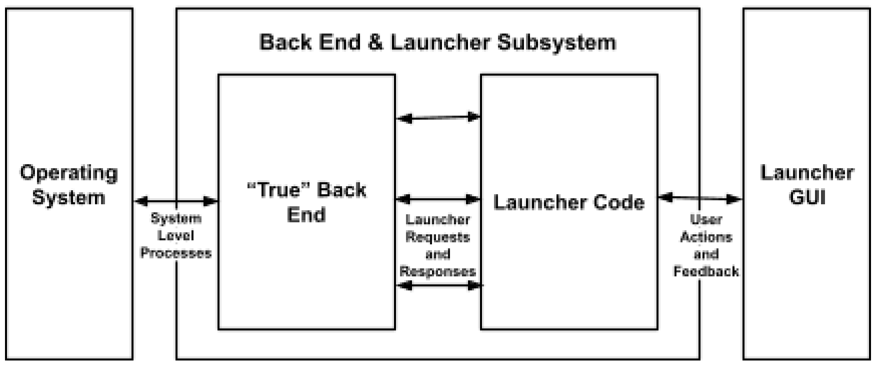
The conceptual architecture of ScummVM seems to be divided into three different layers. These are the backend, the engines, and a utility layer which provides common functionality to the engines.[1] The engines are each in control of how the user interacts with the game being run, and the ScummVM graphical user interface (GUI) allows the user to interact with the game launcher, as well as access a menu overlay while in a game. ScummVM and the engines it supports replace the original executable file(s) from a game, with the engines being able to load the original data files from the games they support.[2]

Back End and Launcher Subsystem

The Back End & Launcher subsystem is responsible for most functionality not directly related to running a game on ScummVM. There are two major divisions in the subsystem: the "true" back end and the launcher code. The "true" back end is the only component which communicates with the operating system (OS), so its implementation is different for each system it operates on. These two parts are grouped together because they are the basis for the entire application, so it makes sense to analyse them as one layer, at least conceptually.

Figure 2

Subsystem Internal Structure


Back End & Launcher Subsystem
Operating System
System Level Processes
"True" Back End
Launcher Requests and Responses
Launcher Code
User Actions and Feedback
Launcher GUI

[1] ScummVM Development Team. "Developer Central/Getting Started." *ScummVM Wiki*, wiki.scummvm.org/index.php?title=Developer_Central#Getting_started.
[2] ScummVM Development Team. "Welcome to ScummVM!" *ScummVM Documentation*, docs.scummvm.org/en/v2.8.0/.

As such, the backend and launcher components are built using a layered interpreter style. OS communicates with the back end, the back end communicates with both the OS and the launcher, the launcher communicates with the back end and the launcher GUI and the launcher GUI communicates with the back end. This creates a layered design that allows for a valuable separation and for the back end to act as an interpreter working between the rest of the program and the OS.

Subsystem Purpose: This subsystem serves as a bridge between the OS and the rest of the application, as well as the game engines. The other main component is the ScummVM launcher.

The OSystem API (Application Programming Interface) is the component responsible for communicating with the OS. It includes many functions and several data types that provide a strong layer of abstraction between the OS and the rest of the ScummVM system. It serves as a superclass to the various operating systems that the ScummVM system is capable of managing such as the Nintendo 3DS or the Android OS.

Adaptability: The OSystem API defines a standard set of functionalities that the ScummVM back end for a system should contain. This allows most of the components of ScummVM to be shared across operating systems, with only the back end needing to be rewritten for new systems. The OSystem API also implements several public abstract data types that can be used to further shorthand many other implementations. This makes it significantly easier to implement features of this type as they are made to work across the system broadly rather than in specific cases.

Evolution: The ScummVM team has built a game engine called ‘testbed’ whose purpose is to test the performance of a new backend. It is built to test all the standard functionality the back end is required to provide to a ScummVM engine.[3] This makes it easier to test new back ends ports of the application through the course of development. Similarly, public data types implemented within the OSystem API make it significantly easier to add new features that are similar to the ones already implemented. This provides many opportunities for evolution and growth within the system.

Performance-Critical Aspects: The critical responsibility of this layer is to perform system calls on behalf of the entire application. It is important that these requests be managed promptly to prevent any delays for the game engine. These important actions include loading an engine, saving a game, and loading a game save. Although it may be more efficient to make these system calls individually rather than through the additional layer of abstraction that this system provides, the OSystem API is providing a valuable service here. Due to the number of different operating systems supported by ScummVM, this provides an advantage when porting ScummVM to new systems.

Control: The back end is always available to facilitate communication with the OS. The launcher, on the other hand, generally passes control back and forth with the game engine. The exception is the pause menu overlay, which merely suspends the game engine temporarily.

[3] ScummVM Development Team. “HOWTO-Backends/Implementing Subsystems.” *ScummVM Wiki*, wiki.scummvm.org/index.php?title=HOWTO-Backends#Implementing_subsystems.

1. The Launcher

The ScummVM Launcher is the application's main menu. It provides access to the user's games, and provides the ability to launch games, or even directly open save files from games. There are options to add new games, edit metadata about owned games, group games based on their qualities and remove games. The user can also access global settings, which affect the way every game runs. The user accesses this functionality through a GUI, but the code that underlies the launcher can be conceptually grouped with the rest of the back end, which controls coordinating OS level processes with the middle end and the game engines. The Launcher can also be considered to include the global main menu (GMM) overlay, which provides some similar options as the Launcher menu, as well as the options to save the game, resume the game, get help about the controls, and exit to launcher.[4]

2. Back End

As mentioned above, the back end connects the application to the OS it is running on, so as such, that makes it platform specific. More specifically, it provides functionality defined in the OSystem API, which is a standard set of actions that both games and the application should have access to (i.e. OSystem defines the signature and contract of the action, the back end defines the implementation) . The implementation of the back end on each system defines how the OS helps provide this functionality to the application.[5] Through the Launcher GUI, the user can carry out multiple actions which require system level operations, mainly saving/loading game saves and changing local/global settings.

3. OSystem

OSystem defines multiple subsystems with actions the backend must be able to fulfill. The ScummVM wiki page suggests these as the major subsystems the backend is obligated by OSystem to implement on most platforms:

- Audio
- Events
- File system
- Graphics
- Mutex + Timers
- Plugins

Some of these subsystems correspond to services provided by the common code layer. This is because the subsystems here often help the common code provide those services, by communicating with the OS for them. There are also other categories of OSystem subsystems which different backend elements may fall into, depending on the specific platform.[6]

OSystem serves as a level of abstraction between the OS and the rest of ScummVM, partially to put particular management on the operations that involve system calls, and partially

[4] ScummVM Development Team. "The Launcher." *ScummVM Documentation*, scumm-thedocs.readthedocs.io/en/latest/using/launcher.html.
[5] ScummVM Development Team. "OSYSTEM Class Reference." *ScummVM API Documentation*, doxygen.scummvm.org/d2/d38/class_o_system.html.
[6] ScummVM Development Team. "Scummvm/Backends/Saves at Master." *GitHub*, github.com/scummvm/scummvm/tree/master/backends/saves.

to allow for functionality between the many different operating systems that ScummVM is intended to utilize. OSystem provides a useful abstraction of the different functionalities of the various operating systems that ScummVM supports and this allows ScummVM to easily access its OS without the additional overhead of determining the correct system calls to make individually, creating a much more adaptable system overall.

Common Code and Graphical User Interface

The common code in ScummVM acts as an important intermediary layer that serves as a bridge between the game engines and the back end. It ensures reusability for the code across different engines, which enables a higher level of efficiency and maintainability. The common code is spread across a large variety of differing modules, each providing specific functionality that can be utilized by all game engines, some primary ones are as follows[7]:

File System Management: The common code includes APIs for I/O operations, allowing engines to access game data easily. For instance, it simplifies file handling within engines and games by allowing files within a ZIP archive to be accessed directly through function calls.

Data Structures: Common data structures such as lists, hashmaps, and queues are implemented within the common code, providing a standardized way to manage data across differing engines.

Media Handling: APIs for audio, image, and video processing are included within the common code ensuring that all engines can handle audiovisual content.

Algorithm Templates: The common code provides some templates for algorithms used to manipulate data, which can be utilized by various engines for tasks such as searching, sifting, sorting and moving.

Debugging Tools: Functions for logging and debugging are also within the common code, this aids developers in diagnosing issues throughout their development processes.

Compression and Decompression Tools: Varying APIs relating to compression of files to formats or decompression of files from Stuffit, ARJ or ZIP, which is integral for the File System Management.

Memory Management: General API for managing memory efficiency, storage and capacity for the engines.

Minor Tools: A great variety of API are in place to allow for easier development process in specific tasks such as a timer, random number generator or game language management.

[7] ScummVM Development Team. "Common API." *ScummVM API Documentation*, doxygen.scummvm.org/db/d78/group__common.html.

Graphics: Provides a unified interface for rendering graphics, abstracting the underlying format differences between games. It does this by taking the graphic format from the engine and converting it for the backend to use with the underlying platform's graphics API.

Audio Management: Abstracts sound and music playback, supporting different audio formats and systems. Like the graphics this converts the sound file from the engine for the backend to use with the underlying platform's audio API.

Input Handling: manages user input (keyboard, mouse, game controllers) and provides a consistent interface across different platforms. This moves in the reverse of graphics and audio taking information from the hardware and the OS to the input handling where it converts it for the engine to understand and call the correct events.

Save/Load: The system serializes the game state and writes it to a save file. When loading, it reads and deserializes the save file to restore the game state. The save/load system interacts with the platform's file system through the backend, abstracting platform-specific differences in how save files are stored and retrieved.

Tests: The common code contains unit testing for functionality in individual components.

Virtual Machine Layer: The environment where the game engines are interpreted for the hardware. When it does this it standardizes how the game logic is executed on different platforms.

The GUI (Graphical User Interface) is the interface that allows users to interact with the software. In ScummVM this is used to add, launch, and interact with games. This includes an initial interface to add games, remove games, start games, and load save files. As well as the ability to change settings for graphics, audio, keymaps, the GUI itself and more. Once a game is launched the GUI is the game interface, allowing the user to interact with the game on their specific platform.

Game Engines and Data

The Game Engine and Data subsystem in ScummVM enables classic point-and-click games to run on modern systems through modular engine re-implementations. This layer contains the core functionality of game engines, as well as the individual engines tailored to interpret specific game formats (e.g., SCUMM, AGI, SCI) and the components which allow them to interface with the game files. The main responsibility of the game engine layer is reading the game files from memory, handling the game logic, getting and processing user input during a game, playing audio from the game, saving and loading game saves, and providing debug tools for developers[8].

Each game engine in this layer is responsible for handling the logic of the games it is designed for. It does this by interfacing with the *common code* layer. The common code layer

[8] ScummVM Development Team. "Engine Class Reference." *ScummVM API Documentation*, doxygen.scummvm.org/d8/d9b/class_engine.html.

contains systems for reading game files, playing audio, and getting input from the operating system. By structuring it this way, the game engine layer does not need to provide system-specific functionality[9]. Instead, it makes use of the common code which interfaces with system-specific backends in order to interact with the operating system to accomplish these tasks. This way, game engine developers only need to focus on implementing code that is required to be part of the game engine, and can safely abstract away details of the operating system that might vary from one environment to another. Another advantage of this common code for the engine is reusability. The code in the common code layer is tried and tested, so developers can reduce their workload and program size by using efficient code that's already been written.[10]

The engine also needs to save and load game states. This is handled by its *data controller*, which makes use of the backend layer and the common code, meaning it does not need to handle any specifics of opening and handling files. The data controller is primarily responsible for saving and loading game states, although it also stores other information such as which "slot" a save is in, the play time on a given save, and a short description detailing the saved game[11].

Subsystem Purpose: Each engine acts as an independent module designed to run games by interpreting original scripts, managing save/load features, and processing media directly from game data files.

Core Services Interaction: Engines rely on Core Services for standardized APIs in rendering, sound, and input. This approach isolates game logic from the underlying platform, allowing engines to focus on game-specific tasks without duplicating essential functions.

Platform Abstraction: Because the architecture separates the game engine components from the other components which handle OS interaction and complicated programming tasks, the task of working on the game engine layer is greatly simplified. This allows game developers to focus on game-specific tasks without duplicating essential functions. This has the additional benefit of making the game engine layer cross-platform and OS-agnostic, allowing game engines to easily be run on new systems without code modifications.

Performance-Critical Aspects: ScummVM's game engine subsystem ensures smooth data access, media playback, and save/load processes. ScummVM maintains efficient resource use and consistent performance across a range of devices, especially for those with limited processing power.

Concurrency: ScummVM's architecture is primarily single-threaded, avoiding the complexity of concurrency but prioritizing smooth control flow between layers. Engines invoke Core Services sequentially to maintain predictable performance across platforms.

[9] ScummVM Development Team. "Developer Central/First Steps." *ScummVM Wiki*, wiki.scummvm.org/index.php?title=Developer_Central#First_steps.
[10] ScummVM Development Team. "Developer Central/Pull Request Approval Process." *ScummVM Wiki*, wiki.scummvm.org/index.php?title=Developer_Central#Pull_Request_approval_process.
[11] ScummVM Development Team. "SaveStateDescriptor Class Reference." *ScummVM API Documentation*, doxygen.scummvm.org/d1/d2c/class_save_state_descriptor.html.

The game engines also communicate with the *backend* layer directly in order to perform tasks which are less common and are not implemented in the common code layer[12]. This allows programmers to complete complex tasks by making use of simple-to-use APIs,

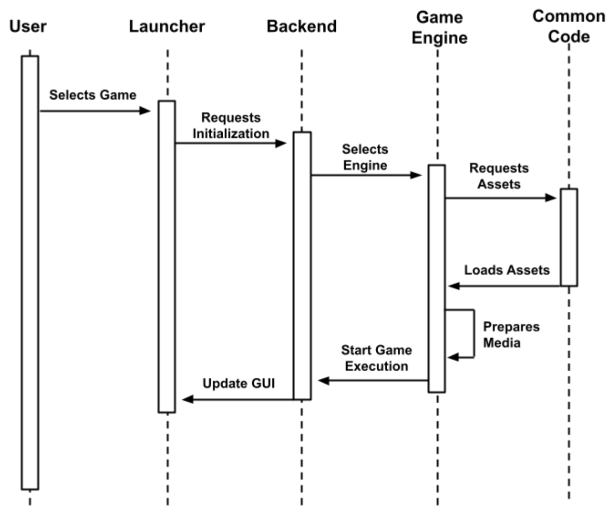
Control, Concurrency & Major Interactions

Control: The control mechanism within ScummVM is fundamentally structured around a main event loop, which serves as the central coordinator for the system's operations. Upon launching a game, this loop initiates the corresponding game engine, which subsequently manages its own state. The backend of ScummVM is responsible for processing player input and relaying this information to the currently active game engine. Additionally, it handles the rendering of graphics and the playback of sound, directing these outputs according to the requirements of the respective game engine.

Concurrency: ScummVM mainly adopts a single-threaded model to maintain concurrency, meaning that all game logic executes sequentially within the main event loop. However, there are notable exceptions to this approach. For example, audio playback is executed in a separate thread to prevent interruptions in the game's functionality. Moreover, certain I/O operations, such as the loading of game assets, can be performed asynchronously. This capability allows the game to continue operating smoothly while waiting for resources to load, thereby enhancing the overall user experience.

Major Interactions: The primary interactions within ScummVM occur between the core and the various game engines. The core functions as an intermediary, effectively managing inputs and outputs for each engine. Game engines request resources from the resource manager, which is integrated within the core system. This design ensures that the engines remain abstracted from the complexities of asset loading, as they simply need to specify their requirements and rely on the core to facilitate the process. Overall, this architecture fosters a streamlined and efficient integration of components, minimizing potential complications and enhancing operational cohesion.

Use Cases



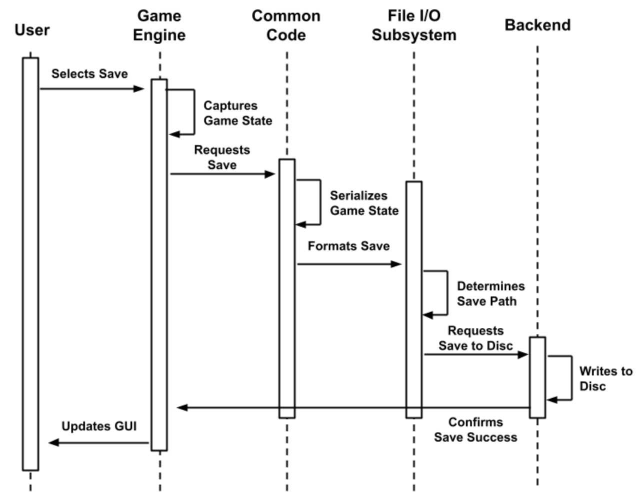
Use Case 1: Game Launching

In the game launching process, the user interacts with the ScummVM launcher's graphical interface to select a game for play. The launcher then communicates with the backend via the OSystem API to request the necessary resources specific to that game. The backend initializes the appropriate game engine based on the game's format and manages system calls

[12] ScummVM Development Team. "Developer Central/Code Base Structure." *ScummVM Wiki*, wiki.scummvm.org/index.php?title=Developer_Central#Code_base_structure.

to load assets such as graphics and audio. Common code facilitates file I/O operations to access the game data and prepares media assets for playback. Once everything is loaded, the game engine begins executing game logic, responding to user inputs through the input handling subsystem, while the rendering and audio subsystems deliver the visual and auditory experience.

Use Case 2: Saving Game State



When a user opts to save their progress, they pause the game and select the save option from the in-game menu. The game engine captures the current game state, including the player's position and inventory, and prepares to serialize this information. The common code's save/load system manages the serialization process and interacts with the file I/O management subsystem to determine the appropriate path and format for saving the file. The backend then handles the system call to write the serialized data to disk using the OSystem API, ensuring compatibility across different operating systems. After the save operation is completed, the game engine updates the user interface to confirm that the save was successful, providing immediate feedback to the player.

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[13]

Conclusions

ScummVM uses a combination layered / interpreter architecture style. The layered nature of ScummVM is shown by the division of its major components: the back end and launcher, the common code, and the game engines, which are the semi-detached layers of the architecture. In other ways ScummVM resembles an interpreter architecture style, as it functionally interprets game data files in the same way that the original game executable would have. Major subsystems have multiple main responsibilities and obligations to other components, allowing ScummVM to reuse some of the same software across different operating systems, and use the same back-end with different game engines. Data largely tends to flow along this order of layers: operating

[13] ScummVM Development Team. "Handling Game Files." *ScummVM Documentation*, docs.scummvm.org/en/v2.8.0/use_scummvm/game_files.html.

system ↔ back end and launcher ↔ (common code) ↔ game engine. As most of what ScummVM runs are lightweight games from the early internet age, it has been relatively straightforward to determine what factors affect concurrency, control, and data flow.

Lessons Learned

From creating this architecture based on the ScummVM software, we have learned how complex even a relatively simple program can be. Despite this difficulty, we found the assignment very rewarding by allowing us the opportunity to better consider the ways in which we think about the design of software architecture. Particularly, the benefits of abstraction have been a valuable lesson to us. Examining the architecture of ScummVM has directly shown the value of abstraction in large systems by making generalization a part of its efficiency. Additionally, this assignment has also convinced us of the value of conceptual architecture before you begin building a large-scale software project. ScummVM seems so simple on the surface but requires far more specificity than one would initially expect from a task as simple as ". . . [replacing] the executables shipped with the games . . ."[14].

However, writing this report was not without its issues. During this assignment, many of us learned that we did not understand what we had previously thought we had understood about software architecture. This forced several of us to take more time with the materials provided by this class and ultimately come through the project with a better understanding of software architecture as a topic and practice. Of course, there are always bound to be issues within groups but ultimately, our group has been rather cooperative with each other. Here is to hoping it stays that way. Our division of work in the group roughly corresponds to different layers of the architecture, which simplified the progress of researching and writing our report. A modular division of group labor allowed each of us to work largely independently.

Overall, we consider this project a success as we have learned much about both the specific architecture and structure of ScummVM but also of the topic as a whole.

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 piece of software intended to provide all the functionality needed to run a game file.

I/O: Short for Input/Output. I/O is a shorthand for all information passing between two agents in a system.

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.

[14] ScummVM Development Team. "Home." *ScummVM*, www.scummvm.org/.

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. “Developer Central.” *ScummVM Wiki*, wiki.scummvm.org/index.php?title=Developer_Central. Accessed 9 October, 2024.

ScummVM Development Team. “HOWTO-Backends.” *ScummVM Wiki*, wiki.scummvm.org/index.php?title=HOWTO-Backends. Accessed 10 October, 2024.

ScummVM Development Team. “Welcome to ScummVM!” *ScummVM Documentation*, docs.scummvm.org/en/v2.8.0/. Accessed 8 October, 2024.

ScummVM Development Team. “Handling Game Files.” *ScummVM Documentation*, docs.scummvm.org/en/v2.8.0/use_scummvm/game_files.html. Accessed 9 October, 2024.

ScummVM Development Team. “The Launcher.” *ScummVM Documentation*, scumm-thedocs.readthedocs.io/en/latest/using/launcher.html. Accessed 10 October, 2024.

ScummVM Development Team. “OSYSTEM Class Reference.” *ScummVM API Documentation*, doxygen.scummvm.org/d2/d38/class_o_system.html. Accessed 10 October, 2024.

ScummVM Development Team. “Common API.” *ScummVM API Documentation*, doxygen.scummvm.org/db/d78/group__common.html. Accessed 10 October, 2024.

ScummVM Development Team. “Engine Class Reference.” *ScummVM API Documentation*, doxygen.scummvm.org/d8/d9b/class_engine.html. Accessed 9 October, 2024.

ScummVM Development Team. “SaveStateDescriptor Class Reference.” *ScummVM API Documentation*, doxygen.scummvm.org/d1/d2c/class_save_state_descriptor.html. Accessed 9 October, 2024.

ScummVM Development Team. “Scummvm/Backends/Saves at Master.” *GitHub*, github.com/scummvm/scummvm/tree/master/backends/saves. Accessed 8 October, 2024.

ScummVM Development Team. “Home.” *ScummVM*, www.scummvm.org/. Accessed 8 October, 2024.