CONCEPTUAL ARCHITECTURE OF SCUMMVM

CISC 322/326 Assignment 1 Report

11 October 2024

Team MAWLOK

Lance Lei (20lh10@queensu.ca)

Michael Marchello (21mgm13@queensu.ca)

Owen Meima (21owm1@queensu.ca)

Kevin Panchalingam (15kp20@queensu.ca)

Andrew Zhang (21az75@queensu.ca)

Zhuweinuo Zheng (21zz28@queensu.ca)

ABSTRACT

This report will explore the open source software, known as ScummVM, and explain its functionality and high-level conceptual architecture.

In the conceptual architecture, we will conclude that this software is designed using a layered architectural style, with a hierarchy of several components working together to allow users to play 90s to 2000s games on the platform of their choice. These components will be further explored in four layers: The Backend Layer, the Common Layer, the Engine Layer, and the Virtual Machine Layer. We will determine these specific components and how the data flows between them using two sequence diagrams, each reflecting a use case based on the actions of the user. We will also discuss the rationale behind our derivation and what lessons we learned while studying the software.

Our report will also outline our derivation process in forming the conceptual architecture and how the evolution of ScummVM is characterized by its ongoing adaptation and growth as evidenced by its frequent updates and expanding compatibility. We discuss this modular design and how it influences the division of responsibilities among developers.

Finally, this report will list any terminology we believe will be necessary to understand in a data dictionary and present any noteworthy limitations that the team encountered in our research and writing of this report.

TABLE OF CONTENTS

INTRODUCTION

With the progress of science and technology, the gaming industry has developed rapidly in recent years. Nowadays, the performance and functions of game engines, game consoles, and computers are more advanced and mature. However, problems have also risen because of this advancement. Many classic games that were developed in the 90s and early 2000s used old game engines and can only be run on specific game consoles, which meant that many games could not be run on today's mainstream PCs. In this report, we analyze the conceptual architecture of the ScummVM software. ScummVM allows users to play classic games that were previously platform specific, on any supported platform. ScummVM was released on October 9, 2001**,** and the program was originally designed to run LucasArts' SCUMM games, but now supports many other games. This includes graphical point-and-click adventures games, text adventures games, and role-playing games. ScummVM is open source in the Github community and attracts users from all over the world. With the help of a growing number of volunteers in this community, ScummVM supports a growing library of games. This report will discuss in detail the functionality, evolution, data flow, concurrency and division of responsibilities among participating developers of ScummVM.

CONCEPTUAL OVERVIEW

ScummVM possesses a *layered* architectural style. The software system contains numerous components which may be grouped by similar purpose, scope, and functionality. These groups can be organized into a hierarchy, since each will only see interaction with at most two others. For example, the backend layer (which facilitates communication between platform-dependent and platform-independent components) only ever interacts directly with the “common layer” situated above it. Regardless of procedure, if the backend layer wished to send data to any of the game engines, it would first need to communicate with a certain component found in the “common layer”. A deeper explanation of each layer (and their subcomponents) can be found in the next section.

It should be indicated that in the diagram below (shown in *Fig I*) uses arrows specifically to represent the flow of data across various elements of the software architecture. Each subcomponent of each layer is in some manner responsible for transferring various information, triggers, or files to others within its reach. Additionally, it is important to note that each layer is also *purely logical*. All functionalities and components *required* for the interpretation and execution of the user’s game files are contained within a single (local) installation entirely on one single device.

The following image provides a brief overview of the interconnections between the various architectural subcomponents, each to be described at a later point in this report.

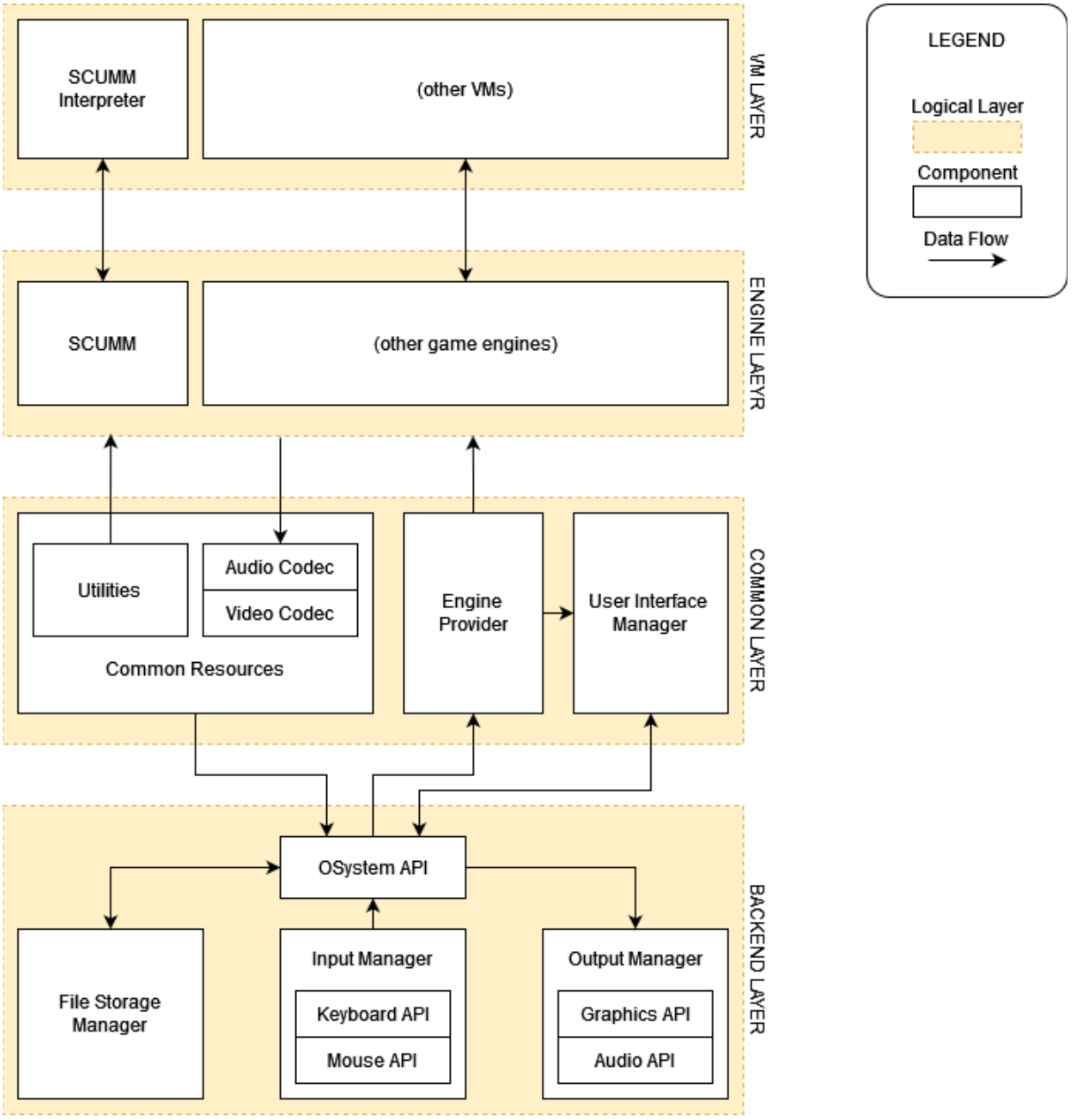


***Fig I** : Box-and-line diagram illustrating ScummVM's layered architecture.*

SYSTEM COMPONENTS

THE ENGINE LAYER

In order for nearly all video games to properly execute (on any machine), they often make use of game *engine* software. This type of software is, essentially, an 'engine' (hence the name) powering the progression of a game's logic and rendering across a large timeline of various user inputs. A user will interact with a game via continuous keyboard, controller, or mouse inputs. An *engine* is a tool used by the running device in order to accurately display a continuous graphical output, in response to the user's inputs.

The "engine layer" is, in essence, an array containing each of the game engines which ScummVM supports (despite being originally named after the SCUMM engine, ScummVM now supports a plethora of other engines). Each engine is ultimately its own component within this layer, as ScummVM contains modern re-implementations of them capable of running on present-day machines and operating systems. In *Fig I*, the only explicitly referenced game engine component is SCUMM, though in actuality, there would be numerous game engines.

Each engine within this layer (i.e. SCUMM) obtains access to the various utilities within the "common resources" component, as well as its codecs for building output data. In addition, they each have access to the VM layer (described below) for interpreting the locally-saved game bytecode provided by the user. Ultimately, each of these engines works to produce continuous productions of graphical/audio output towards the back-end (via the "common layer") concurrently with both accepting user input and calculating (via the VMs) the necessary outputs in reaction to that input. For example, for a game with animated features, the engine would need to repeatedly trigger graphical rendering requests while simultaneously detecting user actions such as camera movement or menu opening.

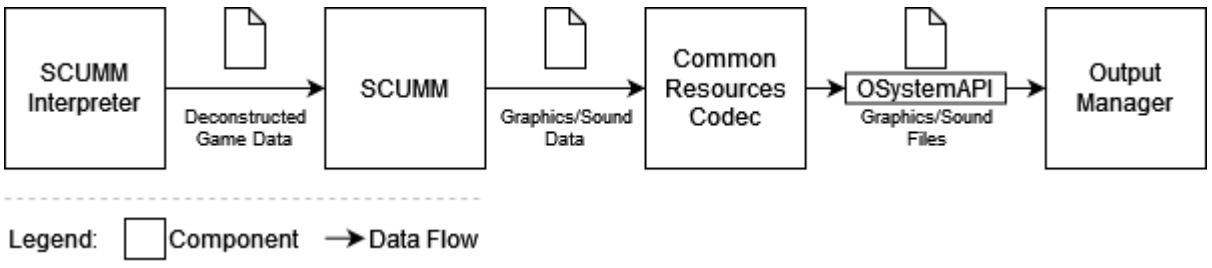


***Fig II** : Data from raw game code being translated into a relevant output format.*

The diagrams shown in *Fig II* describes an example process of raw game data being fed into the game engine, and then being sent towards the backend output component via the codec. At each stage of this process, the software works to turn the logical executions (described in the game's original code) one step closer towards a user-friendly output format.

VM (INTERPRETER) LAYER

The data files initially inputted by the user into their game library are archaic, and cannot be simply executed on a modern computer/device. These files consist of raw bytecode that, although readable to some older (outdated) operating system, must now be 'interpreted' by some component capable of deciphering its original functionality. Note that this does *not* imply that ScummVM is primarily an 'interpreter style' architecture. 'Interpret' in the context of this layer has more to do with the 'translation' of the data files.

Each engine stored within the engine layer obtains a respective 'virtual machine' (VM) which helps to translate raw game bytecode of the games native to that engine. For example, the SCUMM engine will make use of a designated SCUMM virtual machine. This specific VM will feed the engine logical data (relevant to the progression/state of the played game at that specific point in time) in a more "readable" format. The engine may then have the opportunity to work with the lower layers in the facilitation of graphical/audio output and the recognition of user input.

COMMON LAYER

The common layer can only be loosely defined, as it is the only layer whose subcomponents do not share a single, unifying purpose. Rather, the common layer contains all components separate from the backend (i.e. non-platform dependent) that are not specifically a game engine or virtual machine.

I. **Common Resources**
   Within the architecture of ScummVM, there exist numerous distinct game engines utilized in the execution of imported games. Between these various engines, there will be input and output data streams that are similar to each of them. Components that help to hold and transfer this data are contained within a subcomponent of the common layer called "common resources". The common resources component has (at least) two subsections itself (Utilities and Codec). The utilities subsection is generally abstract, and encapsulates various functions and data structures for the game engines (in the higher layer) to utilize. Secondly, the codec subsection has the specific purpose of constructing readable media data (i.e. images, video, or audio) to be sent towards the backend for output. The codecs within the "common resources" component can be seen as a tool used by the game engines in building valid output data for the user.

II. Engine Provider

When the user wishes to add new collections of game bytecode to their library, a subcomponent of the common layer must be given the task of verifying these files to be of a valid game. That is, there must be some element capable of detecting if the given files make up a game designed for any of the engines within the engine layer. If the given files do, in fact, fit this requirement, then the "engine provider" has the opportunity to send confirmation to the interface manager (described below), allowing the user to proceed with their game installation (and play).

III. User Interface Manager

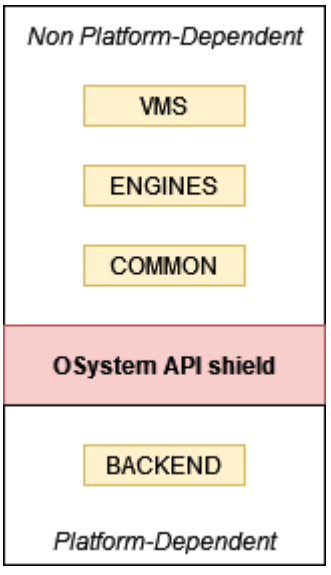
This component encapsulates all the GUI functionality on the ScummVM main library menu. Any additional user interface logic across the ScummVM front-end software, that are *not* directly part of a running video game, would be located within this component. Thus, one could view the user interface manager as an 'engine' for the base ScummVM application itself, facilitating the various button clicks and menu selections the user may trigger.

BACKEND LAYER

The backend layer of ScummVM contains all platform-dependent components and is used for close communication with the user's operating system. Thus, it may be assumed that various deployments of ScummVM (i.e for Windows, Mac, etc) will contain some differentiation in their backend code. However, on the conceptual level, regardless of OS, the backend serves a common collection of purposes, detailed by the subcomponents below.

I. OSystem API

The OSystem API is the single most fundamental element of the backend component, as it acts as a hard shield between all platform-dependent and platform-independent components. All elements of the common layer and above will need to pass through (in other words, utilize) this API in order to send or receive data from the backend. The user's operating system acts as the first step in detecting user input, and the final step in displaying graphical/audio output. Thus, whenever a mouseclick/keystroke is made, an on-screen image is displayed, or a file directory location is accessed, the OSystem API will come into use. All backend components listed below have some interaction with this API.



II. File Storage Manager

Whenever a non-backend component needs to access a file (somewhere within the user's system directory), the file storage manager component must be used. This component is the sole middle-man between the user's device hard drive file system and the rest of the ScummVM software. It may also be used to force open an external, OS-specific file selection software (i.e. "File Explorer" for Windows), in the process of, for example, the user selecting game files to add to their library. In general, the file storage manager receives file-related requests from the OSystem API, and sends back file data at some later point.

III. Input/Output Managers

The input manager works to detect mouse clicks, keystrokes, and any other form of raw user input that the operating system may detect at any point in time. From here, these detectors may then be sent towards the OSystem API (and thus higher layers). For example, the input manager could merely detect a mouse click at a specific screen position, whereas the higher-layer engines would then determine which logic to execute based on that click (i.e. render a new location, open a menu, etc.). Any graphical or auditory output from the higher-layer components will then flow (back through the Osystem API) down to the output manager, which directly utilizes the operating system's speakers and displays accordingly. Both the input and output manager involve one-way data flow, hence their naming.

REFLECTIONS

DERIVATION AND LEARNING PROCESS

We began our project by first conducting research into the wikis, QA forums, and official discord of ScummVM in order to determine how the ScummVM software worked. This involved understanding how games played using game emulators differed from ScummVM. The key difference was that game emulators run games by executing the executable files of games, and ScummVM runs games by executing the data files of a game. This difference took us on a path towards trying to understand how ScummVM is able to recreate the game experience using only a game's data files. The Developer Central wiki page provided a great stepping stone in our understanding of ScummVM's code base by identifying five main components: The OSystem API, the backends, the game engines, Common code, and GUI code. Simple google searches helped us gain general understanding of the high-level function of each component, but determining the interactions between components required a lot of trial and error, and brainstorming.
To start the brainstorming phase, we arranged the five main components identified and tried to determine how we can trigger an event from the user interface towards the ultimate goal of playing a ScummVM supported game. This proved difficult, so we then attempted to draw a sequence diagram in hopes of some guidance. This led us to installing ScummVM software and downloading a freeware game to play, to gain an understanding of the steps the user interface goes through in generating game play. Using the software directly proved very helpful as we were able to work out sequence flows of selecting and playing a game. Once we were able to identify steps and attach various thoughtful components such as File Manager to previously described components such as OSystem API, we then started to gain an understanding of how the components worked together and the direction of flow of data from the moment the user selects a game within ScummVM, to the game experience.

Once we had our sequence diagrams, we then needed to understand where the components we identified in these diagrams fit in with the previously mentioned five main components used in ScummVM's code base. This was a sub-component to a main component match-making effort (such as matching the File Storage Manager to the backends), while remaining within the following understood constraints, "The OSystem API shields game engines and GUI code from the actual platform the software is running on. The backends code is the only part of the code base that is platform dependent." At this point, the sequence diagram gave us the details of interactions between each component. As soon as we had our box and line diagram complete, the layers within the diagram became apparent, thus the architectural style for ScummVM was determined to be layered. Alternative architecture styles such as pipe and filter were discussed, but we concluded that every input is consumed in its entirety before output is generated thus no filtering is assumed.

SYSTEM EVOLUTION

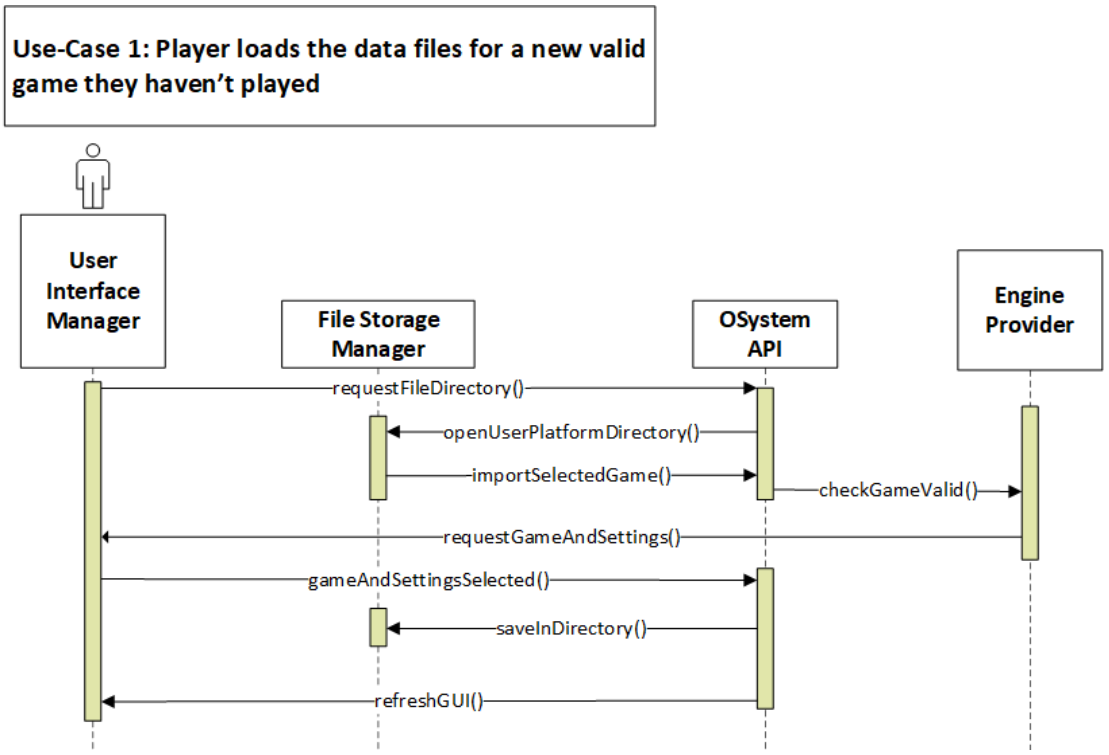
Based on the content from the Github release notes, the ScummVM game library is constantly changing. With the list of compatible ScummVM games and related game engines growing, the demand for rapid bug fixes and modifiability becomes apparent. We thus inferred that individual game engines were independent, or in other words they could be added, edited, and removed with ease due their limited dependency structure. The Developer Central wiki page referred us to the ScummVM API documentation which helped solidify our inference as it clearly shows game engines separated by classes reflecting a modular structure.

IMPLICATIONS FOR DEVELOPERS

ScummVM system structure has four layers, a Backend layer, an Engine layer, an VM layer, and a Common layer. The Backend layer has four components which are File Storage Manager, Input Manager, Output Manager, and the OSystem API. The common layer has three components which are Common resources, the Engine Provider, and the User Interface Manager. In the Engine and VM layers there are a series of engines and a series of interpreters.

Each layer in our architecture establishes the implications for division of responsibility among participating developers. The Backend layer needs developers having more skills in operating systems to abstract platform/OS-specific features. The Common layer needs developers that understand both user interface implementation and game engine interpretation, along with programming ability to incorporate common functionality across the ScummVM system. The Engine and VM layers require developers to have in-depth understanding in game logic, graphics, audio etc required to execute games based on their data files, but also have knowledge surrounding interactions with interpreters in the VM layer.

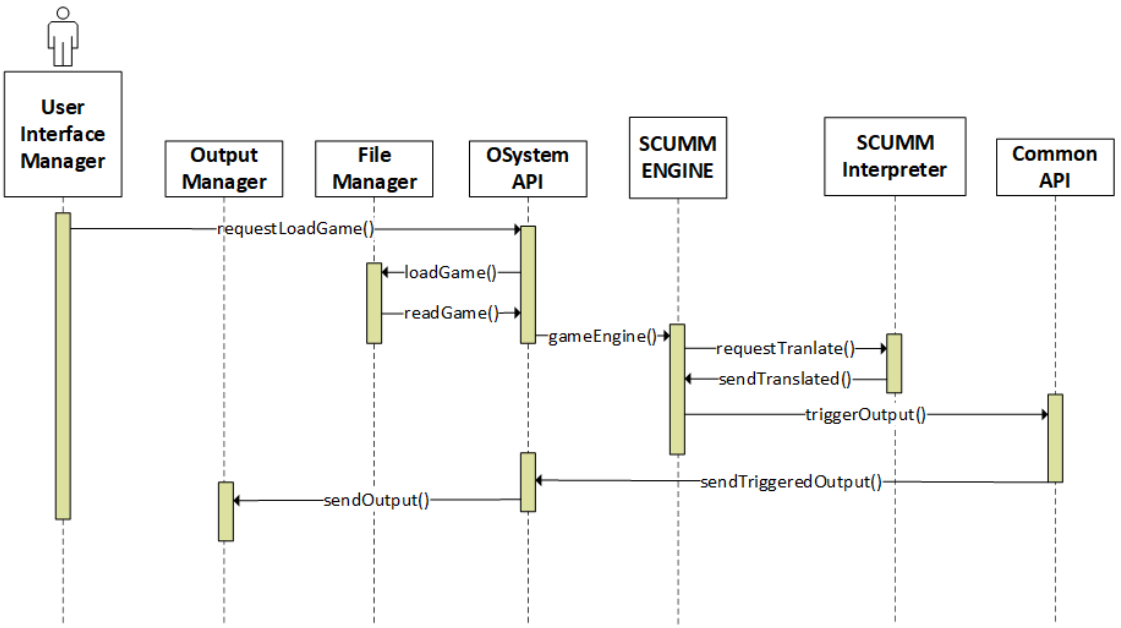
SEQUENCE DIAGRAMS



The first use-case describes the process of a user adding a new "valid" game to their library. The ScummVM interface, upon installation and first run, displays a blank library alongside a series of options for adding and modifying game files copied from the user's local storage. In this use-case, the user has triggered the "Add Game" button on the main menu, and has a collection of valid (importable) game data files to import from somewhere on their device. The following procedures will then be executed in the given order:

1. The user interface manager sends a message to the OSystem API requesting to add a game from platform's file directory
2. The OSystem sends a request to the File Storage Manager (which sits in the backend) to open the user's platform directory.
3. The user selects the folder containing the original game data files from their file explorer (which is independent of the ScummVM software).
4. OSystem API imports the user selected game files and checks if a game engine(s) exists in the Engine Provider.
5. Since our use case only considers "valid" games, a game engine will be found, and a game or list of games will be provided to the user interface where the user can select a game, operating system (if default OS detection is not selected), and preferred sound/graphics settings if available for the selected game.
6. User selects game they want to play and preferred graphics/audio settings which then triggers the OSystem API to save the selected game and settings in user's chosen file directory
7. User can now see the saved data on the user interface

Use-Case 2: Player loads a previously added SCUMM engine powered game



The second use-case describes the process of a user loading and rendering a game that has already been added to their library. When opening the ScummVM interface, the user will see their library listing all their previously played games. In this use-case, the user has selected a game in their library. The following procedures will then be executed in the given order:

1. The user has triggered the “Start Game” button on the main menu.
2. The OSystem API reads the user selected game from the user’s file directory and calls on the related game engine (which in this use-case is the SCUMM engine).
3. The SCUMM ENGINE calls on its engine specific interpreter (the SCUMM Interpreter) to determine which graphics/audio/images are required for the game.
4. Since the SCUMM interpreter is not in an adjacent layer to the layer containing the Common API (which stores graphics/audio/images), the interpreter sends graphics/audio/image requirements back to the game engine to relay to the Common API.
5. The game engine sends the interpreter graphics/audio/image requirements to the Common API.
6. The Common API sends required graphics/audio/images to the OSystem API to generate output on the user interface.
7. The output manager (part of the backend) works with the user’s operating system to provide visuals and sounds to play the game.

CONCLUDING STATEMENTS

ScummVM is an open source software that allows users to play 90s and 2000s games. Through our research we determined that this software is designed using a layered architectural style, consisting of a hierarchy of four layers: The Backend Layer, built for various user platforms, contains all of the platform dependent components in addition to the OSystem API; The Common Layer contains all components separate from the backend that are not specifically a game engine or virtual machine; The engine layer contains an array of each game engine currently supported by ScummVM; The Virtual Machine Layer is a collection of VMs used by individual game engines in reading old game bytecode. We determined the specific components for each layer and how the data flows between them using two sequence diagrams, each reflecting a use case based on the actions of the user. We further expanded on these use cases by describing the flow of information between each component. In our derivation process we analyzed wikis, forums, and official discords to aid us in determining the architectural style and components for the software. We expanded on components and structured them into the four layers seen above. We described the evolution of the software and how it will continue to update and add new engines and games to its library. ScummVM is very fascinating to study and is a great concept for a piece of software. Our group looks forward to seeing what it will look like in the future.

TERM DEFINITIONS

API	Stands for “Application Programming Interface”. This is a collection of functions/interfaces/etc that are used in order to facilitate communication between two distinct programs or software components.
OS	Stands for “Operating Sytem”, the foundation-level program on a machine managing the resources needed to run higher-level programs.
GUI	Stands for “Application Programming Interface”. This is a collection of functions/interfaces/etc that are used in order to facilitate communication between two distinct programs or software components.
Front-End	Used to categorize parts of a software directly visible to the user.
Back-end	Used to categorize elements of a software system that are not visible to the user, i.e. parts working “behind the scenes” (closer to the OS) in order to let the front-end elements function. The “backend layer” in this report is named as such due to its proximity to the OS.
Layered Architecture	An architectural style consisting of numerous stacked layers from top to bottom, in which each layer only has direct communication with those immediately above or below them.

REFERENCES

ScummVM API Docs. (n.d.). doxygen.scummvm.org/d9/d7b/namespace_access.html.

ScummVM Developer Central. (n.d.). wiki.scummvm.org/index.php/Developer_Central.

Uurloon, Mic. “Design of a point and click adventure game engine”. *Groebelsloot*. 1 December 2015. www.groebelsloot.com/2015/12/01/design-of-a-point-and-click-adventure-game-engine/.

SummVM GitHub Repository. (n.d.). github.com/scummvm/scummvm.