Assignment 1:
Conceptual Architecture of ScummVM

CISC 322/326: Software/Game Architecture

October 11th, 2024

Group 12

Trevor White - 21tw43@queenus.ca
Sophia Pagazani - 21snp8@queens.ca
Nasreen Mir - 21nsm5@queensu.ca
Aniss Hamouda - 21arh18@queensu.ca
Nicole Hernandez - 21nhm5@queensu.ca
Claire Whelan - 22cmw2@queensu.ca

Table of Contents

Abstract

This report establishes the conceptual architecture of ScummVM, a program that supports running games on a variety of platforms for which they were not originally designed by rewriting their respective engines. We examine ScummVM through its support of one specific game engine: the SCI engine. Based on information from their wikis and documentation, we established that the ScummVM program has a layered architectural style, with a combined layered and interpreter style for the SCI game engine. Included is a detailed explanation of the program's components and how the communication between ScummVM and the engine forms a layered system, as well as detail on the event loops, data flow processes, and DSLs that make up the interpreter style used by the engine. Our explanations are reinforced with two use cases—launching a game and entering a text-based command—which demonstrate how the components involved in each process combine and interact in ways consistent with our chosen architectural styles. Beyond the architectural style, we discuss our group's research process, conclusions, and lessons learned from the experience.

Introduction

ScummVM is a program that allows users to run a variety of classic adventure video games on systems for which the games were not originally designed. Although the program was originally written to support games that used the Scumm engine, hence its name, the developers soon decided to add support for other game engines as well. Over the years, as an open-source project with many contributors, ScummVM has focused on addressing portability by expanding to support a wide variety of game engines as well as systems the program can be run on.

ScummVM distinguishes itself from an emulator, which would simulate the environment the games were originally designed for; instead, the game engines are rewritten and adapted to run in a platform-independent environment, with ScummVM acting as an intermediary in communication with the user's system (ScummVM, n.d.). One of the engines supported by ScummVM is the SCI engine, which provides resources and functionality for various game elements such as event management as well as graphics, sound, and other output. The SCI engine supports a variety of adventure and text-based games (ScummVM, 2023). In our report, we describe the conceptual software architecture of both ScummVM and the SCI engine, derived from the ScummVM and SCI engine documentation as well as developer-written wiki pages. Ultimately, we establish a layered architectural style for the interaction between ScummVM and the engine and present both interpreter and layered architectural elements that form the basis for the SCI engine.

Starting in the *Architecture Overview*, we establish the different components of both ScummVM and the SCI engine, proceeding to explain how these components communicate to form a layered model of interaction between the engine and the ScummVM backend, with information transfer facilitated by APIs. We also identify an interpreter used by the SCI engine

as a controller. In the *Use Cases* section, we support our arguments with descriptions of use cases along with relevant diagrams. We discuss the system's modifiability, a central element given the open-source development model, under *How the System Evolves.* Finally, we conclude by describing some of the valuable knowledge we gained from studying ScummVM and the SCI engine's architecture under *Conclusions* and *Lessons Learned.*

Derivation Process

To understand ScummVM's architecture, our group consulted ScummVM documentation as well as the SCI engine's documentation. Half the group focused their efforts on ScummVM, and the other half on the SCI engine. We have concluded that ScummVM's architecture is layered, with the game engine and APIs being the main layers. That said, we focus specifically on how ScummVM runs the SCI engine. In that regard, our paper concludes that SCI has an interpreter-style and layered architecture. Note that we also considered the interpreter style for ScummVM, but found that it fit best under the SCI engine's framework exclusively, since the engine is responsible for maintaining the game loop. The engine's layers consist of a user-facing data layer, an interpreter layer that controls the execution of the game scripts and handles communication between the game's logic and its various subsystems, and a lower layer that contains all of the engine's other subsystems, such as graphics and audio. Our group also consulted other game engines to see how they are structured, and the layered style appears to be a common choice (IndieGameDev, 2020).

Architecture Overview

In this section, we provide a breakdown of the notable components and their interactions for both ScummVM and the SCI engine. An in-depth discussion about architectural style and how it connects the two is also included.

1. Components

1.1. ScummVM

1.1.1 User Interface

The user interface is a frontend component that bridges communication between the user and ScummVM. The user interacts with the program through inputs such as keyboard, mouse, or other controller inputs, which prompt the program to execute code. This subsystem also receives the messages from the output API so that the user has a visual of ScummVM.

1.1.2 Common API

The common API component is a collection of many functions designed to support other components and the engine. It is responsible for communicating with the engine interpreter to make save files. It will also receive player input and pass it to the engine. Additionally, it facilitates communication between the engine and the system OS. The common API is also responsible for detecting games. These are just a few important examples of its responsibilities. It contains many other functions that serve to support the other components in completing their tasks and avoiding direct interaction with the underlying system, which reduces the burden of environmental knowledge on these components (ScummVM API Documentation, n.d.).

1.1.3 Output API

The output API component is a collection of APIs dedicated to sound and graphics. It receives instruction from various subsystems in the engine and sends instruction to the hardware. It has a variety of decoders for various image formats that are received from the engine's graphics subsystem. It can also process graphical instructions from the graphics subsystem and render them on screen. The sound APIs receive instruction from the engine's sound subsystem and process that data. They process MIDI files, MIDI drivers, and sound mixing. They also communicate with the hardware to output sound (ScummVM API Documentation, n.d.).

1.1.4 Backends

The backends, also called ports, allow ScummVM to run on multiple platforms and OS. They do this by using the APIs to talk to "OSystem" classes that abstract the operating system. This allows ScummVM to be highly portable, running on platforms such as phones, gaming devices, and more, not just your normal Windows laptop (ScummVM Wiki, 2023).

1.1.5 Database for Game Files

The database component is used to store the files for other components. For example, audio, graphic files, plugging, saves, networking files for the backend, GUI, and more. It is the lowest level of the software architecture, so it interacts more with the hardware than the user (ScummVM API Documentation, n.d.).

1.2. SCI Engine

1.2.1 Graphics Subsystem

The graphic subsystem manages all ports, windows, cursor resources, view resources, font resources, and pic resources. These last four work together to generate graphics in SCI, all of which are drawn on three "maps": the visual map, the priority map (depth of the screen), and the control map (contains special information). It also facilitates the use of palettes, which have their own set of kernel calls they can make. Additionally, it handles the visibility, positioning,

and behaviour of windows, controls, and dialogues, ensuring a cohesive user interface (Skovlund, 2009).

1.2.2 Sound Subsystem

The main purpose of the sound subsystem is to manage different sound and music resources played throughout the game. As such, it can be thought of as a controller/stereo for these sounds. MIDI is the main way these sounds are formatted with sound via simple MIDI files and music composed of a series of MIDI events. The sound devices used can be categorized into four types: MIDI synths, non-MIDI synths, beepers (low-quality sound that is universally supported), and wave devices (Iyengar, 2010).

1.2.3 Window Manager

The window manager is a type of "invisible" port (a graphics data structure) that contains all other windows, who each have their own ports as well. It is one of three types of ports automatically controlled by SCI, with the other two being the menu port and the picture port. Using values from the window manager port, SCI draws the sub-windows backgrounds'. Additionally, windows are an extension of the port structure, but all ports can be accessed through them, and instantiating new ports is not possible, so a new window must be instantiated instead. The picture port draws the current game scene, and the menu port, accurately named, draws the menu (Skovlund, 2009).

1.2.4 Device Drivers

There are a number of devices that SCI interacts with. These include keyboards, mice, joysticks, and graphics drivers. The first three receive input, which is processed by their drivers. These produce events that are then managed by the event handler (Reichenbach & Skovlund, 2018). As for the graphics driver, it is used to produce all visuals shown for a game.

1.2.5 Heap and Hunk

The SCI heap is used for memory allocation. The heap starts with 200 four-byte entries, each of which points to "hunk" memory. The heap also contains memory handles that identify the holes in the heap. The memory allocation function runs an algorithm to find sufficient space and completes the process by splitting data. Exceptions can occur, which often results in returning 0, signaling a failure. The hunk is essentially dynamically allocated memory that is accessed through the heap (ScummVM Wiki, 2009). Since these components are used in memory allocation, this is where the current game state is stored during run-time.

1.2.6 Event Manager

The event manager plays a fundamental role in the SCI engine, acting as a middleman for events. It receives events from various sources and responds to them by activating the necessary subsystems, calling the required kernel functions or APIs, or relaying them to the interpreter. For

instance, typical events include movement-, keyboard-, and mouse-type events produced by device drivers. The event manager polls keyboards according to the timer tick. Joysticks are only polled when requested by the game script. The mouse is special in that it polls itself. This said, there exist more specialized event types recognized by the event manager as well, including those produced by the parser (Skovlund, 2009).

1.2.7 Parser

In addition to classic *point-and-click* style games, the SCI engine supports *text-based* games in which the user is prompted to enter a command to choose their next action (SCI Companion, n.d.). To achieve this, one of its important components is a text parser. This parser receives a string from the event manager, taken from user input, and compares it against some predefined vocabulary text resources to match unknown words to synonyms. The ultimate aim is to determine whether the entered command is valid. It uses a special kind of event in this process, and if the string is successfully parsed, this event is returned to the event handler to commit the action (Skovlund, 2010).

1.2.8 Animation Subsystem

An animation subsystem is provided by the SCI engine to manipulate the graphical objects—views—created by the graphics subsystem. Through its two primary subcomponents, it enables animation cycles. A "mover" is responsible for translating the view to a target point, and a "cycler" handles rotating out the image displayed (Reichenbach & Skovlund, 2018).

1.2.9 Game Code

The game code represents the game-specific files written for a given game. These are things such as objects within the game and scripts defining their behavior. While not inherent to the SCI engine, the engine requires a "game object" encapsulating all these elements at runtime (Skovlund, 2009). The goal of the engine, after all, is to execute these elements in a coherent manner to form the experience of a game.

1.2.10 Interpreter

The interpreter is the core of SCI. It is what allows game code to be executed using the panoply of subsystems provided. When a game is launched, the interpreter undergoes an initialization process that activates all of these. By providing a timer subcomponent to act as the game clock, it is also responsible for controlling the logic of the game loop. Other responsibilities of the interpreter include saving game state and data to the heap/hunk and starting game execution (Skovlund, 2009). In brief, it is the main point of control of the SCI engine, communicating back and forth with the majority of the other components.

2. Architectural style

The primary architectural style for ScummVM is *layered*, as seen in Figure 1. Through analysis of the documentation available, we separated its components into three individual tiers. At the top, we have our *frontend tier*. This includes game-specific interactions such as rendering and extensions of the middle tier's components. It also includes a collection of VM's for game engines. The *middleware tier* interfaces the top layer with the bottom layer but also does the bulk of the work in APIs for graphics, sound, and other related components. In the lowest tier—the *backend tier*—we see a lot of components that deal with the backend. It contains the ports, platform-specific code, game files, and implements anything that is missing due to the lack of the game executable.

After sifting through a lot of documentation, this layered-style architecture seems to be the best fit for ScummVM because of the way the system works. Since it creates an environment for other virtual machines to run, there obviously needs to be a separate section that handles those. From there, the rest of the pieces fall into place as there is already a preexisting separation of the backend from the rest of the system. To extend this idea, we can see SCI sits in that upper layer with dozens of other game engines that have been recreated to run in ScummVM. Despite being a part of this layered architecture, SCI has its own individual styles as well.

SCI has two clear styles, which we identified through our research. The first is a *layered style*, and the second is an *interpreter style*, both working together. In regards to the former, we can identify three tiers for SCI, also illustrated in Figure 1. The first layer, the *data tier*, deals with all the main resources in SCI, game code objects, and the device drivers that directly handle all of those lower-level and backend aspects. All of these are then managed by the next section, the *control tier*. This contains only two main components, the interpreter and the event manager, both of equal importance as they communicate all the processed data and scripts to the top layer of this architecture. The *functionality tier* deals with everything else not yet categorized. It all has to do with the visuals and functioning of the game being run, which includes: the graphic subsystem, the sound subsystem, the window manager, and others. These layers primarily only communicate with their neighbours, meaning the controller layer is the intermediary between the two others.

As for interpreter style, which can be viewed in more detail in Figure 2, SCI has two domain-specific languages (DSLs). The first of these is the SCI Programming Language, which is an object-oriented language with a Lisp-like syntax and is compiled into machine code for the p-machine used by the interpreter, sci.exe (Stephenson, 2016). The second is the Script Programming Language, which has the same characteristics as the former but is compiled into pseudocode and mainly used to create new scripts (Stephenson, 2015). As an interpreter-style architecture, four of SCI's components make up its state machine. First, the execution engine, which is the main game loop; in other words, the interpreter. Second, the current state of the system, which is communicated to the interpreter by the timer component, which produces 60 ticks per second for the game clock (Skovlund, 2009). Next, the program being interpreted, which for SCI is the game files for any of the games created for the engine, such as King's Quest

IV. Finally, we have the current state of the program being interpreted, meaning the current game state. This will be communicated to the interpreter through a number of different outputs that the game creates and is saved to the heap/hunk. For example, in terms of graphics, a “rect” structure is the basis for passing a screen position to the interpreter. The interpreter can directly access these elements, but communication can also be conducted through procedure calls.

In summary, ScummVM and SCI both have layered architectural styles, made clear by the prominent hierarchy between components, and with the addition of interpreter style to SCI due to the use of DSLs and the game loop.

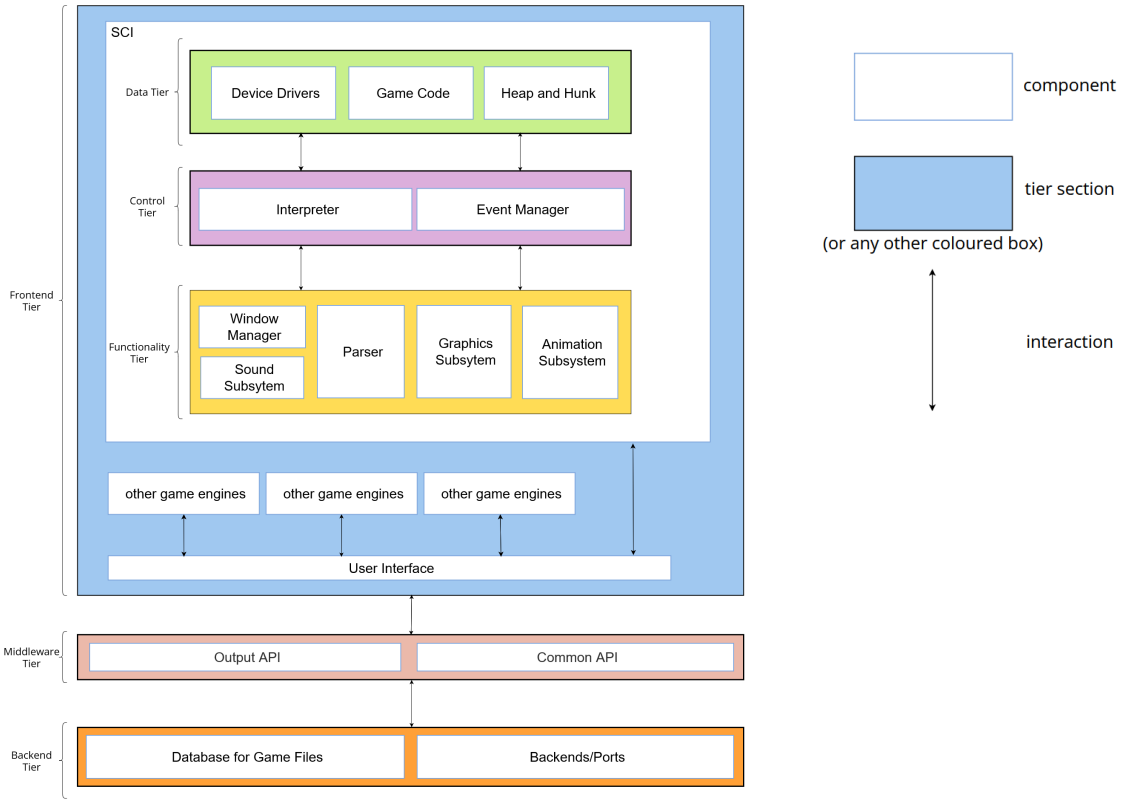

Figure 1: Box and Line Diagram for the System Architecture

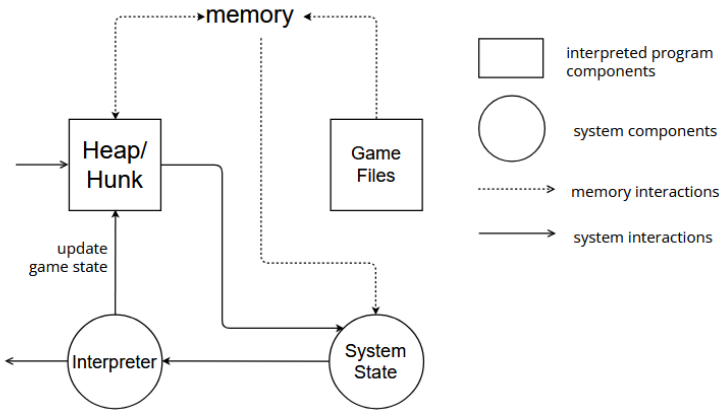

Figure 2: Interpreter Flow Diagram

External Interface

ScummVM primarily acts as an intermediary between the game engines—which have been rewritten to be platform-independent—and the system ScummVM is being run on (ScummVM, n.d.). To facilitate the transfer of data to and from an engine, ScummVM provides a wide variety of APIs related to event handling, audio and visual output, and interaction with the filesystem, among many other features. These APIs transport data from the engines to ScummVM's backend, which is able to communicate with the user's operating system. The SCI engine supports a variety of inputs, such as mouse clicks for point-and-click games and text input for text-based RPGs. In many cases, a game will take input, which thereafter requires the SCI engine to access source files or communicate with audio output, among others. Here the engine will use an API to communicate with these elements (ScummVM, n.d.).

Information such as user inputs and display updates is transmitted between the GUI and the engine. The GUI communicates state changes (ex. inventory updates, dialogue selections) to the engine, which processes these to alter the game state. The SCI engine reads and interprets game files containing assets and script information. It retrieves game resources like sprites, audio clips, and script functions from resource files on demand during gameplay. These resources are transmitted internally and parsed by the engine to display graphics, play sounds, and handle user interactions.

ScummVM also has its own non-game-specific files such as configuration files, plugins, translation files, and a global save directory, which are mainly stored in the user directory where the ScummVM application is located (ScummVM, n.d.). The ScummVM launcher interface facilitates interaction with the user's file system, allowing users to add games to ScummVM from their local filesystem as long as they are supported, as well as save and load previous game states. Save and load operations involve writing and retrieving game state information (ex. player position, inventory items) to and from save files (ScummVM, n.d.).

Use Cases

1. Launching a Game

Launching a game starts from the ScummVM menu, which will show a list of all available games. The player must select the game and click launch. When this happens, the user interface will pass this command to the common APIs. These will pass a message to start the game to the interpreter of the SCI engine. This begins its initialization process. It will initialize the heap and hunk components. It will also request the config file and parse through it. After loading the drivers in the config file, the interpreter will go through the other components of the SCI engine and initialize them. It then allocates space on the heap, gets a pointer to the game object, and starts running the game (Skuvland, 2009). This control and data flow is visualized in Figure 3.

Moreover, observe that the interpreter remains active once the game begins. In essence, it could theoretically be represented as another actor; it remains alive to trigger interactions according to the clock and act as the main controller for the game engine.

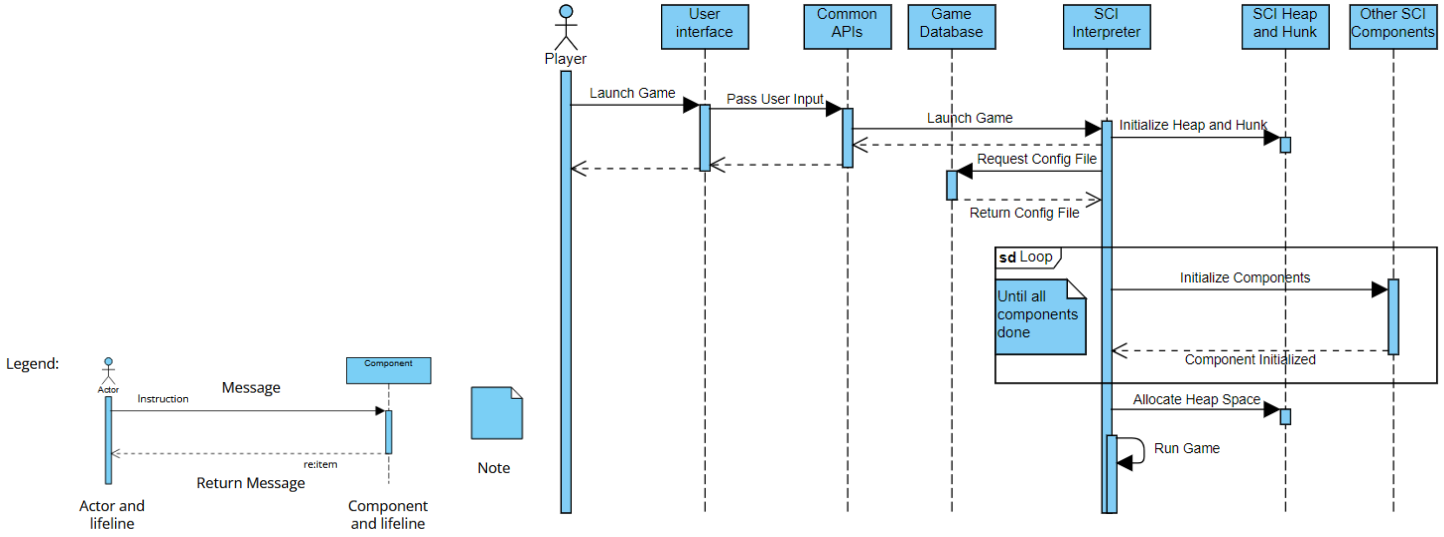

Figure 3: Sequence Diagram for Launching a Game

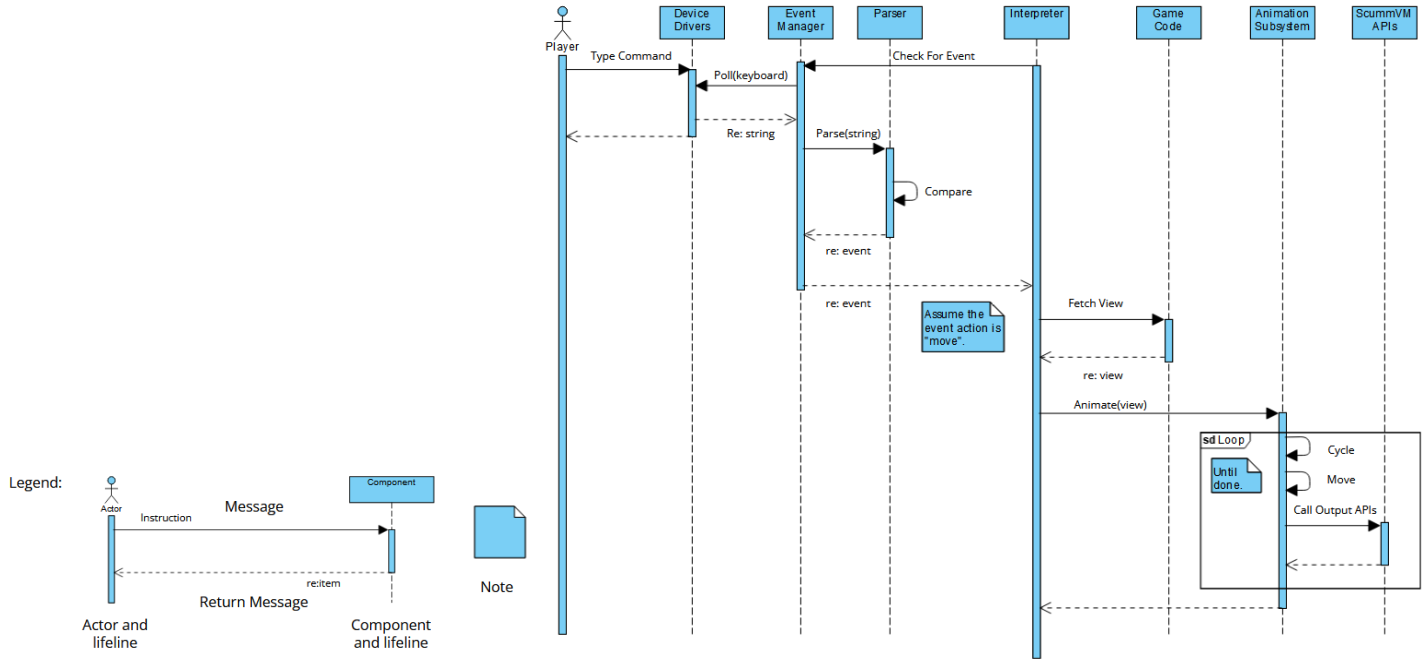

Figure 4: Sequence Diagram for Entering a Text-Based Command

2. Using a Text Command

As previously described, the SCI engine enables *text-based games*. In such a game, the process of entering a text command, manipulating it, and executing the action is a frequent use

case. Figure 4 provides a simplified visualization of a scenario in which parsing is successful and the command relates to player movement.

First, at some point in the game loop, the engine's interpreter verifies the presence of events by contacting the event manager. It polls the device driver corresponding to the keyboard. If the user is concurrently entering a command, the string is received by the event manager, who then passes it to the parser. The parser compares it to its vocabulary resources and generates an appropriate event, captured by the event manager, who then returns control to the interpreter. At this point, the interpreter requests the affected view object from the game code and dispatches the movement command to the animation subsystem. There, it uses the cycler, mover, and graphical output APIs provided by ScummVM to manipulate the image displayed on screen. This continues until the animation cycle is complete.

How the System Evolves

ScummVM is open source, meaning that the source code is public to everyone through its Github repository. Anyone is able to update the source code as long as it aligns with existing coding conventions, complies with Github commit guidelines, and the existing developers accept the changes or additions. Things that people can contribute to ScummVM include fixing bugs, working on open tasks (found on the ScummVM wiki), adding ports, plugins, new game engines or features to game engines, and more.

These developments are primarily in C++, but Python, Objective-C, and Java may be used if the changes include niche features such as dev tools, Apple OS backends, or Android backends, respectively. The affected code evidently depends on what is being changed. For example, you would touch the backend code if you wanted to port ScummVM to a new platform. However, if you were more interested in working from the game engine perspective, then you would evolve the OSystem API. To keep the maintainability aspect of ScummVM, there is a rigorous process before someone changes the code. Of course, you have to fork the repository to have your own work space. Once you are done with your updates, they must be submitted via the pull request system. Once the pull request is made, the public and team members of ScummVM are able to view and comment on the change for two weeks. If there are no objections from the ScummVM developers in that timeframe, then the new additions are merged with the existing code. BuildBot—an open source framework that is used to help automate software development like testing, integration, and deployment release—ensures that the changes do not cause code regression. The owner of the change must document the new feature by editing the documentation files on the GitHub scummvm/doc/docportal/ directory (ScummVM Wiki, 2023).

In order to join the ScummVM development team, you must express interest in joining or have one of the leads or co-leads invite you to the team. However, to be a member, you must have previously improved ScummVM through pull requests while following the developer guidelines (ScummVM Wiki, 2023).

Conclusions

In summary, ScummVM enables portability to non-native systems for the game engines it recreates thanks to its layered-style architecture. It is composed of user interface elements, game files provided by the user, and a layer of APIs. The former facilitates launching and saving games, and the latter implements the abstraction between the engines, the operating system, and the hardware. Moreover, the SCI engine itself has a layered structure, which separates the user-written game elements from its inner workings. It strongly relies on an interpreter-style controller, acting as the main point of control between components, enabling the game loop, and tracking the state of execution. This further enables portability and increases the ability to extend the system, such as by writing new SCI-compatible games. This said, these findings are conceptual in nature as they are derived from documentation. Discriminating the concrete architecture from the source code and comparing the two is a logical next step.

Lessons Learned

Overall, uncovering the conceptual architecture of ScummVM and the SCI engine was very insightful. Through the process, we developed a deeper understanding of the interpreter-style architecture and other common styles used in video game engines. It also provided a chance to learn more about how open source projects are maintained and evolved, and a deeper look into the meaning of “API.” The task was not without its challenges, however. At times, broad documentation of components was not available in favour of extensive, detailed, or unclear specifications of the inner workings, which made deriving a full image of the system difficult. In a similar vein, there was a lack of documentation written prior to the implementation. Thus, we were essentially tasked with producing a conceptual architecture based on already uncertain resources due to themselves being reverse-engineered. Finally, our approach to research was flawed. While it made working as a group easier by dividing us into smaller teams, it ultimately hindered our ability to understand the role of ScummVM. Moving forward, we aim to look at the SCI engine and ScummVM not as two separate subsystems but as a single unified system.

Data Dictionary

Word	Definition/Description
Point-and-click game	A video game controlled uniquely by clicking on various objects using a mouse.
Ports	A key data structure in the graphics subsystem that records the state of the subsystem storing information like cursor pointer, colours, etc.
Text-based game	A video game controlled primarily by typing natural language instructions equating to actions via a text prompt.
View	A special data-type of the SCI engine implementing graphical representations for game objects.

Naming Conventions

Acronym	Definition/Description
API	Application Programming Interface
DSL	Domain Specific Language
GUI	Graphical User Interface
OS	Operating System
RPG	Role-Playing Game
SCI	Script Code Interpreter, or Sierra's Creative Interpreter (ScummVM Wiki, 2023)
ScummVM	Script Creation Utility for Maniac Mansion Virtual Machine
VM	Virtual Machine

References

Buildbot. (n.d.). *Buildbot*. https://buildbot.net/

Deckhead. (2020, January 15). *Game Engine Development: Engine Parts*. https://indiegamedev.net/2020/01/15/game-engine-development-for-the-hobby-developer-part-2-engine-parts/

Reichenbach, C. & Skovlund, L. (2018, October 25). *Views and animations in SCI*. https://wiki.scummvm.org/index.php?title=SCI/Specifications/SCI_in_action/Views_and_animation_in_SCI

SCI Companion. (n.d.). *Overview*. https://scicompanion.com/overview/

ScummVM. (n.d.). *Saving and loading a game*. https://docs.scummvm.org/en/latest/use_scummvm/save_load_games.html

ScummVM. (n.d.). *ScummVM*. https://www.scummvm.org/

ScummVM. (n.d.). *Understanding the interface*. https://docs.scummvm.org/en/v2.8.0/use_scummvm/the_launcher.html

ScummVM. (2009, September 4). *Programming a new game*. https://forums.scummvm.org/viewtopic.php?t=7886

ScummVM API Documentation. (n.d.). *API reference*. https://doxygen.scummvm.org/modules.html

ScummVM Wiki. (2023, April 17). *SCI*. https://wiki.scummvm.org/index.php?title=SCI

ScummVM Wiki. (2022, April 19). *HOWTO-Backends*. https://wiki.scummvm.org/index.php?title=HOWTO-Backends

ScummVM Wiki. (2023, August 10). *Developer Central*. https://wiki.scummvm.org/index.php?title=Developer_Central

ScummVM Wiki. (2023, August 10). *Team Onboarding*. https://wiki.scummvm.org/index.php?title=Team_Onboarding

ScummVM Wiki. (2023, February 26). *Platforms*. https://wiki.scummvm.org/index.php/Platforms

ScummVM Wiki. (2009, January 24). *Hunk and heap*. https://wiki.scummvm.org/index.php?title=SCI/FreeSCI/Kernel_hacking/Hunk_and_heap

ScummVM Wiki. (2009, January 31). *The SCI Heap*. https://wiki.scummvm.org/index.php?title=SCI/Specifications/SCI_virtual_machine/The_SCI_Heap

Skovlund, L. (2010, February 10). *Parser*. https://wiki.scummvm.org/index.php?title=SCI/Specifications/SCI_in_action/Parser

Skovlund, L. (2009, January 6). *SCI Ports*. https://wiki.scummvm.org/index.php?title=SCI/Specifications/Graphics/SCI_Ports

Skovlund, L. (2009, January 31). *Event handling in SCI*. https://wiki.scummvm.org/index.php?title=SCI/Specifications/SCI_in_action/Event_handling_in_SCI

Skovlund, L. (2009, January 31). *Interpreter initialization and the main execution loop*. https://wiki.scummvm.org/index.php?title=SCI/Specifications/SCI_virtual_machine/Interpreter_initialization_and_the_main_execution_loop