Assignment 2:
Concrete Architecture of ScummVM

CISC 322/326: Software/Game Architecture

November 18th, 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

Having established a conceptual architecture for ScummVM and its implementation of the SCI engine in our previous paper, we put the code into Understand and sorted the files based on our original components. Our goal was to revise our conceptual architecture and establish a concrete architecture. In this paper, we first review our conceptual architecture and then perform reflexion analysis on the discrepancies, identifying both high-level divergences across the whole system as well as low-level discrepancies within the SCI engine. We also find it necessary in places to create new components or slightly redefine previously established components to account for code that did not fit into our original conceptual architecture. Following this analysis, we establish our concrete architecture based on revisions to the layered elements of our conceptual architecture and rework our two use cases to demonstrate how the program's control navigates this architecture with specific function calls. In addition to architectural analysis, we describe our investigation process and reflect on the lessons we learned throughout this project.

Introduction

ScummVM is an open-source software platform that allows users to run classic point-and-click adventure games on modern computer systems. It was originally designed to support LucasArts games created with the SCUMM engine, but ScummVM has since expanded to support titles from other developers and engines, including Sierra's SCI engines. ScummVM works by re-implementing the game engines, so users only need the original game files, without requiring the original hardware. It supports many operating systems, including Windows, Mac, and Linux, and has been adapted for mobile devices. The platform is maintained by a community of developers who continuously add support for more games and engines. For retro gaming enthusiasts, ScummVM is a crucial tool in preserving gaming history.

This report aims to provide an educational overview of ScummVM's concrete architecture, along with a detailed view into the architecture of the SCI engine subsystem. We start with a review of the conceptual architecture established in our previous report. Next, we go through how we derived our concrete architecture using Understand. Then, we dive into the concrete architecture and discuss ScummVM's high-level architecture, the architecture of the SCI engine subsystem, and why some parts of our conceptual and concrete architecture differ. Two use cases and sequence diagrams are provided for reference and support of our arguments.

Review of Conceptual Architecture

Our conceptual architecture is organized into 3 tiers: a Frontend tier, a Middleware tier, and a Backend tier. Within the Frontend tier, we had another 3 tiers for the SCI engine: Data tier, Control tier, and Functionality tier.

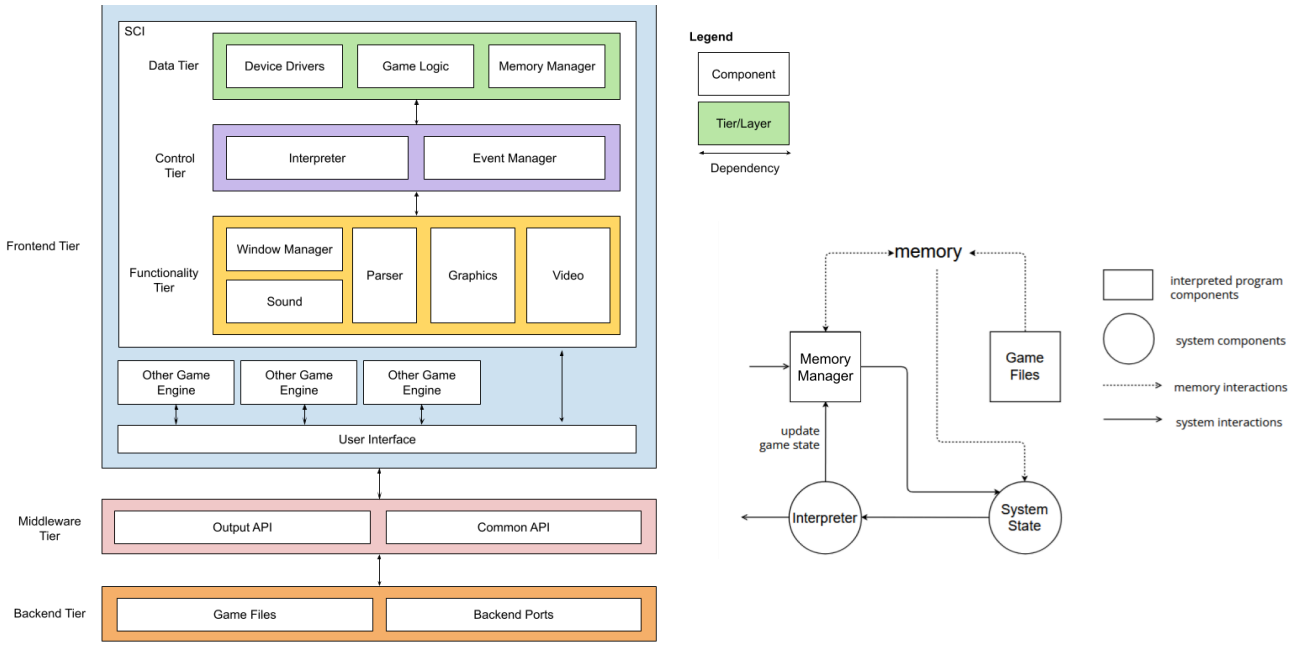

Figure 1: *Revised Conceptual Architecture and Interpreter Flow Diagram*

In the Data tier, the Device Driver manages hardware interactions like input and output, the Memory Manager handles memory allocation, and Game Logic contains the functions that create the gameplay mechanics. In the Control tier, the Interpreter manages the game loop, and the Event Manager handles user inputs and in-game events. The Functionality tier handles the essential functions that support gameplay. For instance, sound, graphical displays in windows, and video. The Middleware tier to handle communication from the game engine to the Backend and provide various APIs for the engines. Finally, the Backend tier contains the Game Files and Ports, which support various system configurations.

Furthermore, we renamed Game Code, Heap and Hunk, and Animation Subsystem. After reviewing the code files, we found that the game code focused more on logic, such as player movement and object classes; thus, we decided Game Logic would better reflect its purpose. Heap and Hunk was renamed because it was too abstract, but “Memory Manager” is a more tangible classification for anything that relates to memory. The Animation component was renamed to Video after observing that the animation-related methods were closely integrated with Graphics, with only those handling pre-rendered videos remaining separate.

Derivation Process

For our group to develop a concrete architecture and then perform a reflexion analysis, we first had to get and parse through the source code of ScummVM. As a group, we loaded the files and code into Understand. We used our conceptual architecture as a starting architecture and then began to put folders and files into each component based on its name, file content, comments in the code, looking at what the code was doing, and the API documentation that

ScummVM provides. We discussed ideas as a group, and if we were unsure of the placement of a file, we jotted it down to reassess again later.

The derivation of ScummVM's concrete architecture was easier than SCI's since we found fewer divergences and files being pre-sorted into their respective components. With SCI's concrete architecture, we had to add new components to have the architecture fit fully with the source code. To investigate the divergences, we used the Understand visualization tools to see which files were creating unexpected dependencies. Then we discussed if it had something to do with how we sorted the files, if it was an oversight in our conceptual architecture, or if it was due to how ScummVM and SCI were constructed.

Reflexion Analysis

Using Understand, we generated dependency graphs, which we have simplified in Figure 2 for better readability. In the following sections, we investigate the divergences between the layers of ScummVM, the 2nd-level issues within SCI itself, and discuss a more appropriate box-and-line diagram for the concrete architecture.

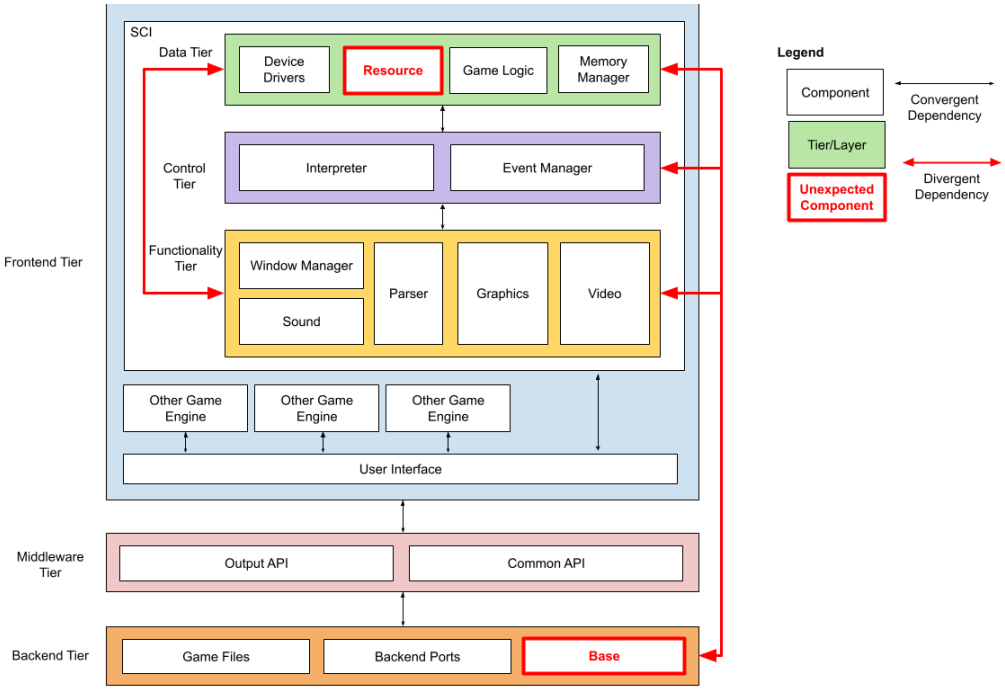

Figure 2: *Unexpected Dependencies Simplified*

1. Top-Level Reflexion Analysis

One area of concern we encountered when examining ScummVM's interaction with the SCI engine in Understand was that there were two-way dependencies between the components making up ScummVM and those making up the SCI engine. These dependencies from ScummVM to the engine are largely because when sorting code in Understand, we put

non-SCI-specific engine code from the ScummVM engines folder into our SCI components for the expediency of our model, since we are only looking at the implementation of the one game engine with ScummVM. We further investigated the following unexpected dependencies between the SCI engine and ScummVM components.

1.1 Interpreter ↔ Backend Ports

As part of the Control tier of SCI, the interpreter was one component we had not predicted interacting directly with the Backend tier. The bulk of the references to backend functions are in the class metaengine.cpp, which manages actions prompted by keyboard input and relies directly on backend functions to take this input. All other uses of backend functions by the interpreter are for the process of saving the game state.

The more unexpected relation was the Backends Ports' dependency on SCI interpreter code, specifically by backend code related to events and graphics. Files such as `default-events.cpp` and `opengl-graphics.cpp` use functions from the not-SCI-specific `engine.cpp` file, which manages the functions for each engine and which the backend ports reference for information on whichever engine is running. For example, `openglsdl-graphics3d.cpp` calls the function `hasFeature()`, which checks based on the game engine in question whether the ports can override certain values, in this case resolution when displaying graphics. It also manages functions for pausing the engines and handling autosaves, which are called by `engine.cpp`.

1.2 Interpreter ↔ Base

The base was an unexpected component we added to account for the main.cpp file and other related files, such as version information files, which clearly fit in the Backend tier of ScummVM but did not relate to specific plugins. The interpreter interacts with abstract plugin classes contained in base code files such as `internal_plugins.h` and `plugins.h` for plugin instantiation and management. It also relies on `version.h` to access version information in order to display it to the user.

As with the backend plugins' dependency on the interpreter described above, the base relies on nonspecific engine and meta-engine code that detects instantiations of engines and provides information about them. It also relies on similar functions to detect games. These functions are used in main.cpp as well as in `plugins.cpp` for plugin instantiation and `commandLine.cpp` for inputting to the command line.

1.3 Memory Manager → Backend Ports

Like other dependencies between the SCI engine and the backends, the Memory Manager's dependency was unexpected, as we believed all interaction between the engines and the backend would go through the Middleware APIs. As with the Interpreter's dependency on the

Backend Ports, files in the memory manager use backend functions related to saving files and game states from `savefile.cpp`.

1.4 Window Manager → Base

There are two different files in Windows Manager that include one file each from Base: `version.h` and `plugins.h`. No functions are called from these files, meaning this dependency has no effect on our architecture and is negligible.

1.5 Sound → Backend Ports

The majority of Sound's dependencies on ScummVM components are through Middleware as we expected. However, the sound component relies directly on Backend Ports to retrieve playback information. A function in the Sound file `audio.cpp` accesses information from the audio player file `audiocd.h` in order to determine whether certain audio is still playing.

2. Second-Level Reflexion Analysis

In our chosen subsystem, the SCI game engine, we can see a few inconsistencies between our proposed conceptual architecture and what is actually happening in the code. Once mapped to our conceptual architecture, we noticed that components in the code were not communicating in respect to our defined layers, specifically the Data tier and the Functionality tier.

2.1 Resource ↔ Video, Graphics, and Sound

Our first discrepancy within SCI is between the Resource component and Video, Graphics, and Sound. These are self-explanatory since the resources needed to produce the relevant images, noises, and videos are stored and managed in the resource component (objects and functions like `resourceManager`, `resourceID`, and `findResource`). For the Video and Sound components, these relate directly to a bunch of decompression/decoding algorithms that unpack the resources needed to produce output. They are both dependent on Resource. The edge between Resource and Graphics is bidirectional, with the dependent edge (Graphics → Resource) being essentially the same as Video and Sound. The other direction differs slightly but is still grounded in logic. Many `enum` objects are created in the graphics system with the use of "`helpers`" from Resource which essentially facilitates creating new types.

2.2 Resource ↔ Parser

Much like our rationale for 2.1, the Parser relies on Resource because it manages different resource types that the component uses. The Parser uses Resource as a manager with an emphasis on the file `vocabulary.cpp`, which contains a data type for various words and vocabulary. Resource depends on the Parser by means of a `vocabulary selector`, which

manages how and which words are parsed through text-based input. As with the earlier section, this relationship is necessary, and our architecture has been adjusted accordingly.

2.3 Memory Manager ↔ Video, Graphics, and Sound

Another main component that had unexpected dependencies is Memory Manager. Much like Resource, these are logical, and we have made all relevant changes to accommodate the discrepancies. Video has a one-way dependency on the Memory Manager, which deals with segment managers, allocation bitmaps, and more. Essentially, as well for the dependencies from the Graphics and Sound components to Memory Manager, it is handing out memory and related resources to the requestees. For Graphics and Sound, there are many `saveState` calls, `engineState` calls, segment managers, and calls to determine data types, all of which are a necessary form of communication. Memory Manager also depends on Graphics and Sound, with the latter being concerned with a `GFX` class that, while created in Graphics, handles a lot of stuff with bitmap, cursor, palette, and more that Memory Manager relies on. For its dependency on sound, we are seeing the `audioPlayer` manage saving audio states, settings, and other functions that Memory Manager utilizes to save game and engine states.

2.4 Memory Manager ↔ Parser

Parser depends on Memory Manager to get engine states, get game objects, utilize debuggers and segment managers, as well as get kernel states. All of this is used to start the Parser, keep track of other events that may affect it, and effectively allow it to access memory. The other direction of this dependency has the Memory Manager using formatting functions and vocabulary lists to save important information to the heap/hunk and game state. This direct communication is essential for quick processing of memory and, as such, has highlighted areas of change in our existing architecture.

2.5 Game Logic ↔ Graphics and Sound

For both of these divergences, we are seeing two-way dependencies. Starting with Sound, its dependency on Game Logic has only one `#include` statement for `_object`, while it is depended on by `kScript.cpp` for getting specific sound resources and locking them for concurrency. Graphics dependency on Game Logic is more of a catch-all, with `scriptPatcher`, workarounds, selector flags, and a few other miscellaneous files being used to handle animation and frames. Game Logic's dependency on Graphics makes a bit more sense, with essential rendering, color, and screen objects being needed for the pathing and movement scripts since those actions need to be visually represented to the user.

2.6 Game Logic ↔ Parser

Parser utilizes only one function from Game Logic, `getSynonyms`, which is self-explanatory and helps the Parser interpret user input. In the other direction, we see Game Logic utilizing essential functions from Parser, like `kSaid` and `ResultWordList`, to most likely handle what results from user input. We are also seeing debuggers and workarounds being used here for quality control.

2.7 Window Manager → Resource

This dependency is unidirectional (Resource depended on by Window Manager) and only has a single file discrepancy, which renders it insignificant. In this file, it seems no specific function is called, and just an `#include` statement for `resource.h` is found in `kmenu.cpp`. This has no significant effect on our architecture and seems to be something added with no meaning by the developers.

2.8 Window Manager → Memory Manager

This dependency is small in number of files but still significant. Window Manager utilizes functions and classes from Memory Manager to find the types of specific data, to cast data from one type to another, to get engine state, and to manage segments, most of which seems to make displaying all information considerably easier. These all are consistent with our understanding of the Window Manager, and, as such, contribute to the reasons for changes within our architecture.

2.9 Window Manager → Game Logic

The Window Manager subsystem utilizes only a single object from Game Logic—the `achievementManager`—which, as the name suggests, handles achievements. This is used to display achievement progress to the user, as the Window Manager handles the main graphical user display for SCI. We have moved Window Manager in our concrete architecture to be more reflective on the actual communication it has with essential components, because while it is small, it does play a significant role.

3. Concrete Architecture

Through our reflexion analysis, it became clear that the conceptual architecture designed for ScummVM and the SCI engine was not a good fit. Some unanticipated components became evident and divergent dependencies defied the original purpose of the layering, leading to a reevaluation of the layers and architectural style.

First, the most evident failure of the conceptual architecture is that two components had to be added to properly map all the source files, as seen in Figure 2. These are the Resource and Base subsystems and resulted in many extra dependencies. The Resource component belongs to

SCI, while Base is a part of ScummVM. Initially, we imagined that specific resource instances would be handled independently. For example, vocabulary data would belong to the Parser, and audio data would belong to the Sound component. However, inspection of the code revealed the existence of a `Resource` class used to handle and identify every type of data with a single method of interface. Thus, resources could not be easily distributed among multiple components, and a new component was needed. As for the Base component, it was added to the Backend tier and contains files that act as the main control point for ScummVM—similar to how the Interpreter handles the control within SCI. This is a vital functionality that we overlooked in our conceptual architecture, assuming it was handled within the User Interface or Backend Ports, which was not the case.

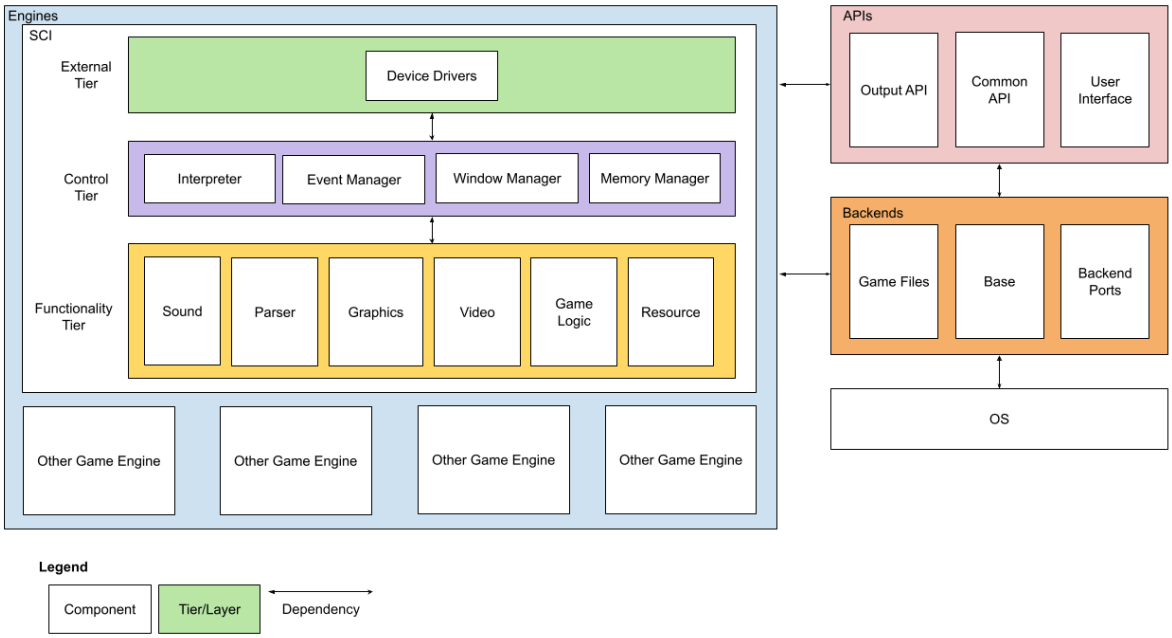

Figure 3: *Final Concrete Architecture*

Moreover, the investigation into the divergent dependencies elucidated some fundamental contradictions in the architecture of ScummVM. Primarily, the Backend tier of ScummVM communicates directly with the engine in the Frontend tier and vice versa, rather than funneling requests through the Middleware tier. Instead, those two layers work alongside the SCI engine, as shown in our concrete architecture in Figure 3. From a technical standpoint, this decision is logical. By relaxing the rigidity of the layered style, the system gains the ability to support higher-performance operations required in real-time systems, such as when playing a game. All the same, there remain clear groupings of components with similar purposes—Engines, APIs, and Backends—and the Backend still acts as a layer of abstraction for the system's operating system. In that sense, ScummVM retains a partially layered style.

A similar phenomenon is seen within the second-level component SCI in Figure 2. The Functionality tier depends directly on the Data tier. The same is true for the opposite direction. Here, the issue stems from our initial definitions of the layers. The renamed Game Logic and

Memory Manager no longer fit the exclusive "data" role. For example, Game Logic also determines pathing and movement. As a result, we redefined each tier (Figure 3). The Game Logic and Resource components have been transferred to the Functionality tier. Likewise, the Window Manager and Memory Manager, both controllers in nature, belong more appropriately to the Control tier. This leaves only the Device Drivers in the top layer, which is refactored to the External tier. No files were found belonging to the Device Drivers; we suppose that they are external to the engine. With this new organization, the divergences are now expected.

Finally, in terms of architectural styles, ScummVM and SCI are both layered, as just described. This provides an organization to their components and interactions. However, the SCI engine primarily uses an interpreter style, hence its Interpreter component. This is the main way in which it supports the game loop, allowing it to track and process the game's state over time. The concrete architecture has not revealed any changes to this fact. Additionally, SCI appears to make use of a publish-subscribe style as well with its Event Manager. The Event Manager subscribes to the external Device Drivers to learn of input events such as mouse movement and publishes resulting events for the Interpreter to process.

Use Cases

In this section, we reconsider the non-trivial use cases discussed in the previous report in light of our findings regarding the concrete architecture of ScummVM and the SCI engine.

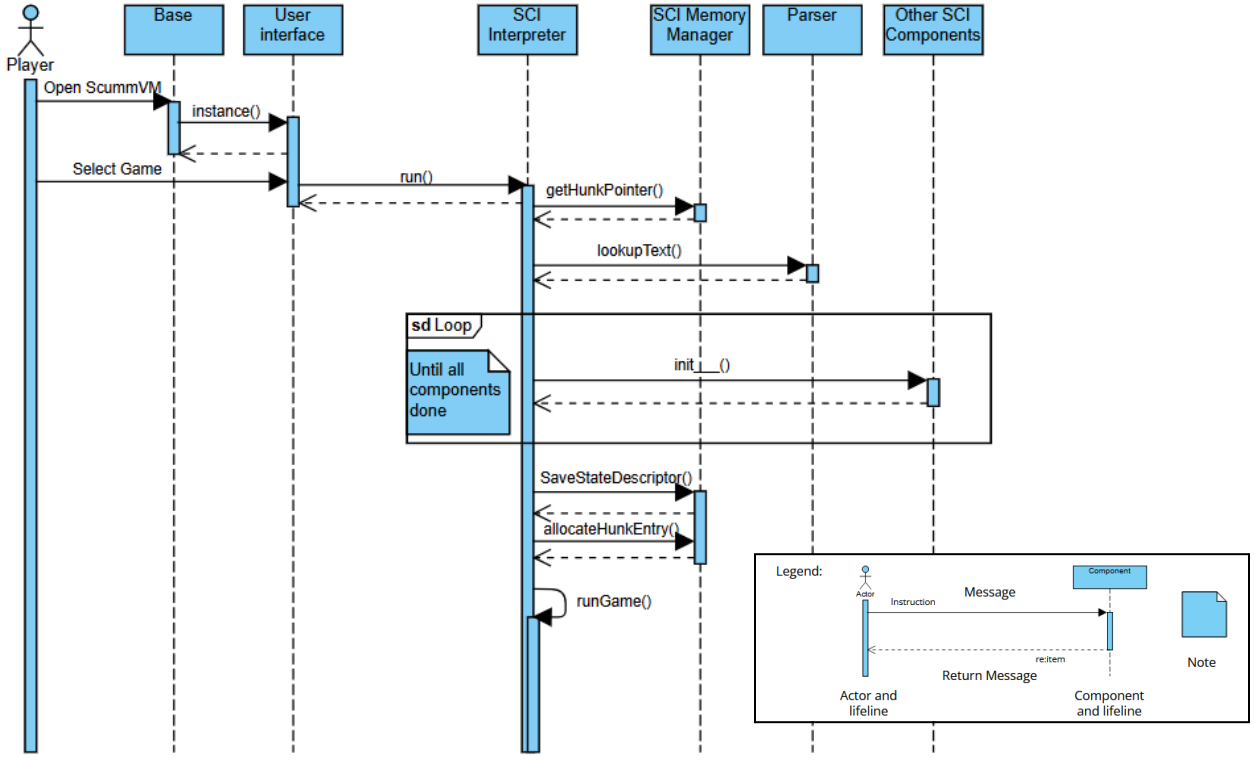

Figure 4: *Sequence Diagram for Launching a Game*

1. Launching a Game

The sequence diagram for this use case is presented in Figure 4. When launching a ScummVM, the user must open the program. This launches ScummVM from the Base component. The Base then calls `instance()` to initialize the User Interface. The user will then select the game they want to play from the User Interface menu. This will call the `run()` function in the Interpreter. The Interpreter will then initialize the memory segments by calling `getHunkPointer()` and call the Parser to go through the config files with `lookupText()`. The Interpreter will then initialize the drivers specified in the config file, as well as the various remaining components in the SCI engine using their respective `init` functions. Once done, the Interpreter will call the Memory Manager to save the current state of the machine using `SaveStateDescriptor()` and allocate additional space in the memory "hunk" using `allocateHunkEntry()`. When everything is complete, it will begin the execution of the game by calling a recursive `runGame()` function.

2. Moving With a Text Command

Now, consider the use case wherein a player successfully inputs a text-based prompt to move their in-game character, illustrated in Figure 5. First, the Interpreter prompts the Event Manager according to the clock using `getSciEvent()`. The Event Manager then continuously polls the keyboard for input characters via calls to `kGetEvent()`. The Parser provides a method named `getVocabulary()` to compare the input string to vocabulary resources. These are pulled from the Resource via `loadParseWords()`, then analyzed with functions such as `parseNodes()`. In this case, we assume input is a match and the Event Manager returns control to the Interpreter.

Next, since a move command was issued, the Interpreter calls `kDoBresen()`. This implements Bresenham's line generation algorithm to determine all the pixels in the movement path (Pradhan, 2024). It is provided by the Game Logic subsystem, which also pulls the target game object from memory by calling the Memory Manager. After this, control is again relinquished to the interpreter along with the path and game object information.

The final step is to call the Graphics subsystem to render the movement on screen. It fetches the view space using `getView()`, provided by the Window Manager. Finally, looping until the movement is done, it uses various functions to update the screen, such as `loadPict()` from the ScummVM Output API and its internal `updateScreen()` method.

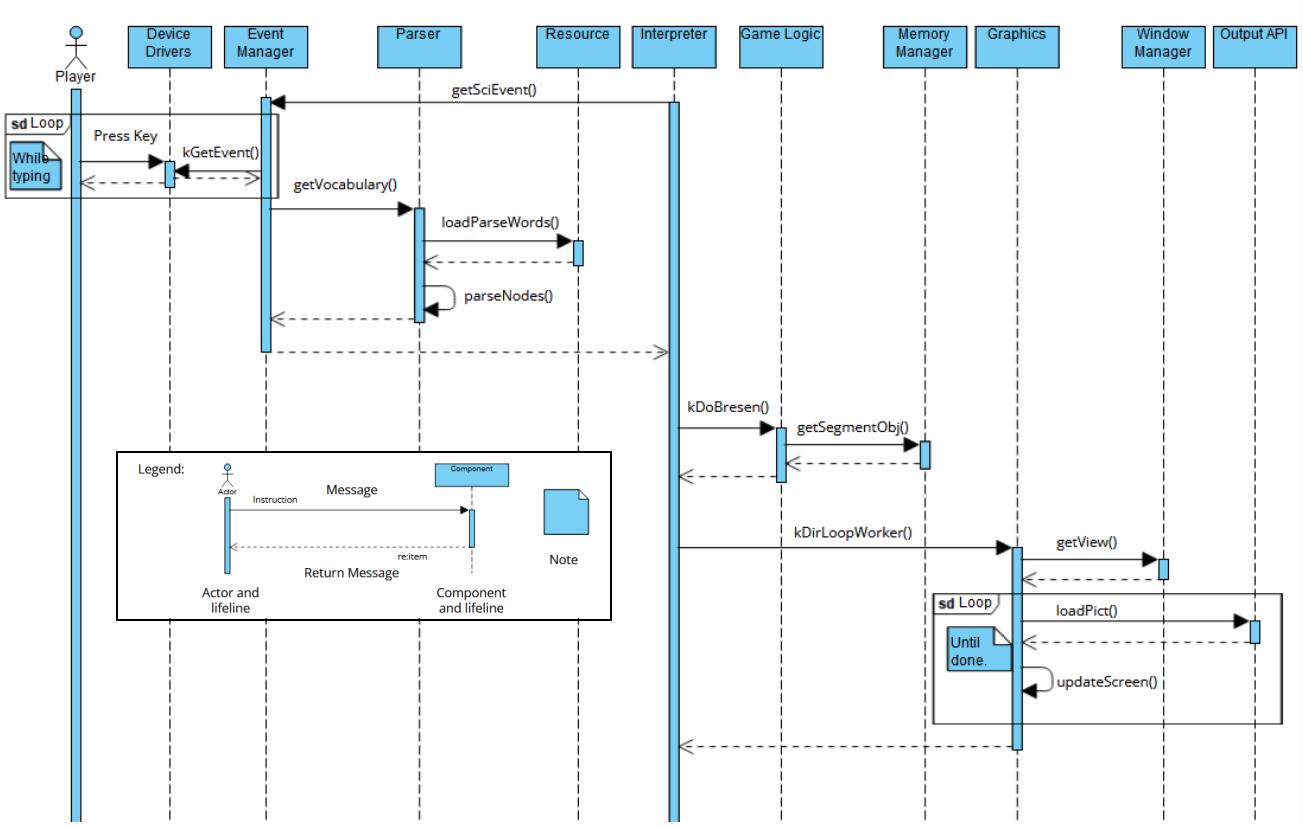

Figure 5: *Sequence Diagram for Entering a Text-Based Command*

Conclusions

By using Understand to analyze ScummVM's source code, we were able to develop a concrete architecture of the system. In this analysis, we found that all of the components in the conceptual architecture were convergent, but we also discovered divergences in the form of a Resource component in SCI and a Base module in ScummVM. We also changed the structure of the architecture. The Frontend became the Engine tier, and the User Interface was moved to the Middleware tier, becoming the API tier. The SCI engine remained as a layered architecture, but ScummVM changed. It is no longer a strict hierarchy, with a Frontend and Backend separated by a middle tier. Instead, the Engine tier is supported by the API and Backend tiers that work side by side. By analyzing the source code, we developed a stronger understanding of the system. We can use this knowledge to think of modifications that could be made to improve the system.

Lessons Learned

Through coming up with conceptual architecture and performing a reflexion analysis, we learned how important teamwork, proper documentation, and using the right tools are. Instead of splitting up the research and analysis, we decided to work on that part of the project together.

The process was easier to manage due to being able to discuss and brainstorm the concrete architecture in real time. This was important due to the lack of comments in the source code, documentation of what each file does, and proper naming of files. Due to minimal documentation, it was a challenge to understand each file's function and what component it would fit best in. This taught us how important it is to have well-documented code and naming conventions so that others can have an easier time understanding our code. It also taught us how important it is to use proper tools while analyzing software. This is because we would have had more of a challenge understanding component dependencies without using Understand. Overall, this assignment showed us how much effort it takes to understand software and that using tools and effective teamwork makes the process smoother.

Data Dictionary

Word	Definition/Description
Understand	A software used to help create concrete architecture diagrams from code and help visualize dependencies between components and files
Ports	A key data structure in the graphics subsystem that records the state of the subsystem, storing information like cursor pointer, colours, etc.
View	A special data type of the SCI engine implementing graphical representations for game objects.
Enumerators (`Enum`)	An enumeration, or `enum`, is a symbolic name for a set of values. Enumerations are treated as data types, and you can use them to create sets of constants for use with variables and properties (Dollard, 2021).

Naming Conventions

Acronym	Definition/Description
API	Application Programming Interface
SCI	Script Code Interpreter, or Sierra's Creative Interpreter (ScummVM Wiki, 2023)
ScummVM	Script Creation Utility for Maniac Mansion Virtual Machine

References

Dollard, K. (2021, September 15). *When to use an enumeration - visual basic*. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/constants-enums/when-to-use-an-enumeration

Pradhan, S. (2024, March 11). *Bresenham's Line Generation Algorithm*. GeeksForGeeks. https://www.geeksforgeeks.org/bresenhams-line-generation-algorithm/

ScummVM. (n.d). *scummvm/scummvm*. GitHub. https://github.com/scummvm/scummvm

ScummVM API documentation: Scummvm API reference. (n.d.). *ScummVM API documentation: ScummVM API reference*. https://doxygen.scummvm.org/

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