CISC 322/326 - Software Architecture
Assignment #2: Report

Concrete Architecture Analysis of ScummVM

TC++
Evan Cook - 21egc10@queensu.ca
Hugh Tuckwell - 21halt@queensu.ca
Will MacInnis - william.macinnis@queensu.ca
Liv Stewart - olivia.stewart@queensu.ca
Jay Wu - 21jjw12@queensu.ca
Nate Rinsma - 21nr6@queensu.ca

1. Abstract

This report shows a comprehensive breakdown of the concrete architecture of ScummVM, explaining the system's components, their responsibilities, and interactions. Initially, we used seven components in our conceptual architecture - Engines, Audio, Graphics, Input Handling, Game State Management, Launcher, and OSystem API - and have now redefined our understanding, reducing to the following six components within our concrete architecture: Engines, Audio, Graphics, Launcher, OSystem API, and Common. By using the Understand tool to better comprehend the architecture and its dependencies, we have adjusted our components to better reflect our discoveries.

Throughout the report, each component's role is clarified, with Engines managing game emulation, Audio handling sound replication, graphics controlling visual output and interface, Launcher offering user interface and settings control, OSystem API dealing with OS interactions, and Common supporting shared facilities between most components. We have further examined the SCI subsystem by detailing its specific components and identified unexpected dependencies in our analysis.

Our reflexion analysis compares conceptual architecture with concrete architecture, looking for divergences, such as new dependencies between components, and including rationales for these architectural evolutions. The findings are shown through updated dependency diagrams and use case diagrams, as well as going through the rationale for these changes.

2. Introduction and Overview

ScummVM is a versatile emulator that runs a wide range of different adventure game engines. To emulate classic games accurately, ScummVM has many key components that have their own set of specific responsibilities.

This second report breaks down the ScummVM architecture into its components, explaining their roles and interactions, and shows how the overall structure supports the system's goals, performance, and future adaptability.

We will now look at the concrete architecture of ScummVM. Previously, in our report on the conceptual architecture of ScummVM, we listed that the seven main components were: Engines (including SCI), Audio, Graphics, Input Handling, Game State Management, Launcher, and OSystem API[2]. However, we have come to a different conclusion after using Understand to observe the dependencies between the files. We now believe that the concrete architecture includes six main components: Engines, Audio, Graphics, Launcher, OSystem API, and Common. These components closely work together to recreate the experience of playing classic adventure games on a variety of systems. In the new variation of the Engines component, we observe that it is a collection of recreated game engines, such as SCI, Kyra, and Plumber, which run various adventure games by interpreting the game data files. In the new variation of the Audio component, we observe that it consists of drivers, decoders, etc. with built-in MIDI support that enables authentic sound replication on modern systems. In the new variation of the Graphics component, we observe that it handles visual output 2d game rendering, 3d game rendering, and the GUI which all have various scaling methods to adapt older games to modern screens. In the new variation of the Launcher component, we observe that it provides a graphical interface for users to load, remove, and launch games while it also allows game-specific and general setting adjustments. In the new variation of the OSystem API, we observe that it offers an interface between the operating system and ScummVM which enables features like graphics display and input handling across many different platforms. Since we have a new component which is classified as Common. We observe that it contains many shared utilities, libraries, and files which are used across all components, including math functions and other resources.

Now that we have had a better look at the ScummVM architecture, we will also be delving deeper into the SCI architecture to get a better understanding of what makes up the SCI engine and how it works. The SCI subsystem's Common component is responsible for containing utility files like events, debugging, and general utilities which are all used by every other component. The SCI subsystem's Engine component is responsible for running games by emulating a virtual machine called PMachine[1]. The SCI subsystem's Sound component is responsible for handling all audio-related functions like decoders and drivers which have dependencies between most other components. The SCI subsystem's Graphics component is responsible for managing all visual elements, including animation, cursors, fonts, and video decoding, with dependencies on all other components. The SCI subsystem's Resource Manager component is responsible for decompressing and patching various resource files, including audio, video, and game logic. The SCI subsystem's Parser component is responsible for parsing text-based user input using grammar and vocabulary files to guide game actions.

As such, we have also included an updated dependency diagram in this report. The updated diagram shows the dependencies that the identified 6 main components share. We will compare the conceptual architecture to the concrete architecture to find any absences and divergences between the two. To properly compare the two architectures, we have revised our old conceptual architecture to match the components of the concrete architecture so that we can have a proper look at the dependencies without getting confused as to which dependencies would show convergences within the old model.

We have also briefly provided the conceptual and concrete architecture of the SCI engine subsystem. This includes showing the dependencies between the 6 components of SCI. From this, we can discern what architectural styles we believe the systems to be. For ScummVM, we still believe that it follows a hybrid of Layered and Interpreter styles. Now including the subsystem SCI, we believe that it is Repository style.

Furthermore, we have also provided new use case diagrams of ScummVM based on the concrete architecture. These use cases are visualized to reveal how the different components would interact with each other sequentially given different tasks. The first use case shows the sequence of actions the system takes to process user input when they play the game which results in updated outputs of the game due to the player's input. Subsequently, the second case shows the sequence of actions taken to launch a game on its respective engine to allow the user to play it.

As mentioned previously, we found many unexpected dependencies between the components when we examined and reviewed the concrete architecture. Although we have not identified any absences, we did identify 13 divergences of dependencies the concrete architecture had that we did not connect in the conceptual architecture. We also looked closer at the SCI engine to see the dependencies. Compared to our conceptual architecture, again we found no absences but we did find that there were 10 divergences in the concrete architecture. We will discuss further in detail about what the divergences are and why they exist in the concrete architecture.

Finally, we will conclude the report with a summary of our key findings of ScummVM based on the concrete architecture we observed by using Understand and our own understanding. This will be shortly followed by any future proposals we have for ScummVM. To wrap up the report, we will reflect on the lessons we learned while conducting our research on the newfound information about the concrete architecture of ScummVM.

3. Architecture

3.1 Architecture Overview

The concrete architecture of ScummVM follows a layered architecture, the same as the conceptual architecture. This is still most evident through the existence of the OSystem API component, which acts as a layer of abstraction between the other components and the operating system. The counterpoint to the layered architecture could be that there are dependencies between almost all the components so how can it be layered, however, the main layer is the OSystem API between the operating system, as it is the only component to really interact with that, so it makes sense that most of the components interact but can still be layered.

3.2 Components

There are six different components in the concrete architecture of ScummVM, detailed below.

Engines

The engines component is a collection of recreated adventure game engines that ScummVM uses to run adventure games. For our concrete architecture, we looked at SCI, Kyra, and Plumber, although ScummVM as a whole supports many more different engines than that. The game engine component is responsible for interpreting the data files of the game being run. The engine makes use of the audio component and graphics component to turn the interpreted data into something tangible that the user can see and hear.

Audio

The audio component of ScummVM consists of several different parts like drivers, decoder(s), and more. ScummVM includes built-in support for MIDI, allowing users to emulate or use a physical synth such as the Roland MT-32. This means that games played on modern systems are still able to sound exactly like they did on the systems that they were originally released for. The audio component is responsible for loading the appropriate audio asset as specified by the resource manager, mixing it, and initiating playback at the right time given events taking place in the game world.

Graphics

The graphics component of ScummVM is responsible for managing the visual output presented to the player. This includes rendering both two-dimensional and three-dimensional game worlds, but also the GUI that the player will see. One key task of the graphics component is scaling. Since ScummVM runs games originally made for different displays, ScummVM includes many different methods for scaling the image up to larger sizes to better fit modern screens. There are even different rendering modes included, allowing ScummVM to emulate the looks of different systems; for a few older games which could be played on more than one system.

Launcher

The launcher is responsible for providing a graphical interface to the user, as well as managing engines. Within the launcher, users can load games from their system, remove games, launch games, and edit game-specific settings as well as ScummVM settings. It communicates with the platform abstraction component to access game files, as well as to load the game programs and corresponding engine into memory when being run.

OSystem API

The OSystem API component acts as a layer between the operating system for a given machine and ScummVM itself. The API specifies various features a game can use. These could be things like displaying graphics on a screen, receiving input, etc. The API is then implemented for various operating systems so that the games and engines can run on any platform that the OSystem API has been implemented for. The OSystem API interacts with the input handler when receiving input from the operating system, as well as with the launcher when it needs to access files and other operating system resources. It also receives calls from the audio and graphical components and passes on these calls to the operating system.

Common

The Common component is a collection of various utilities, libraries, and files that are used by all of ScummVM's various components for several different tasks. An example of this is a folder simply called "Math", which contains a number of different functions used by different components. Notably, this is the component that now fills the job of the Input Handling component from the conceptual architecture. This means that Common is now the component that handles the user's input from their device of choice.

3.3 Reflexion Analysis

As we were deriving our concrete architecture, we realized that some of the components in our conceptual architecture didn't quite line up. As such, we have modified our conceptual architecture to align with the components in the concrete architecture. We've done this by moving game state management into the engines component, and the input handling into a new common component, which is a component for any functionality that provides utility to other components, such as math functions. Provided are the two architecture diagrams to reference for the following reflexion analysis.

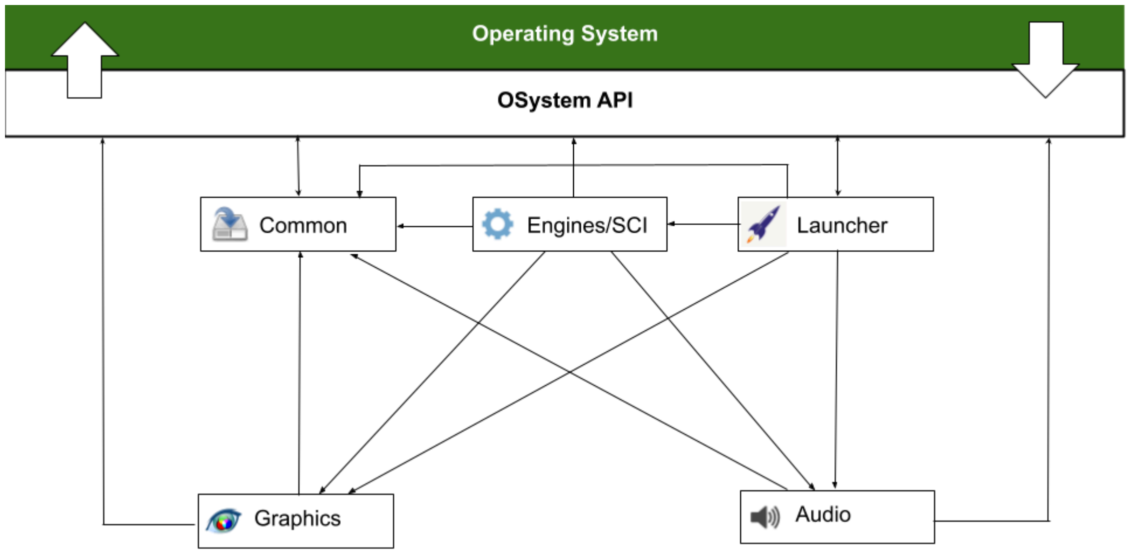

Figure 1: Updated ScummVM Conceptual Architecture

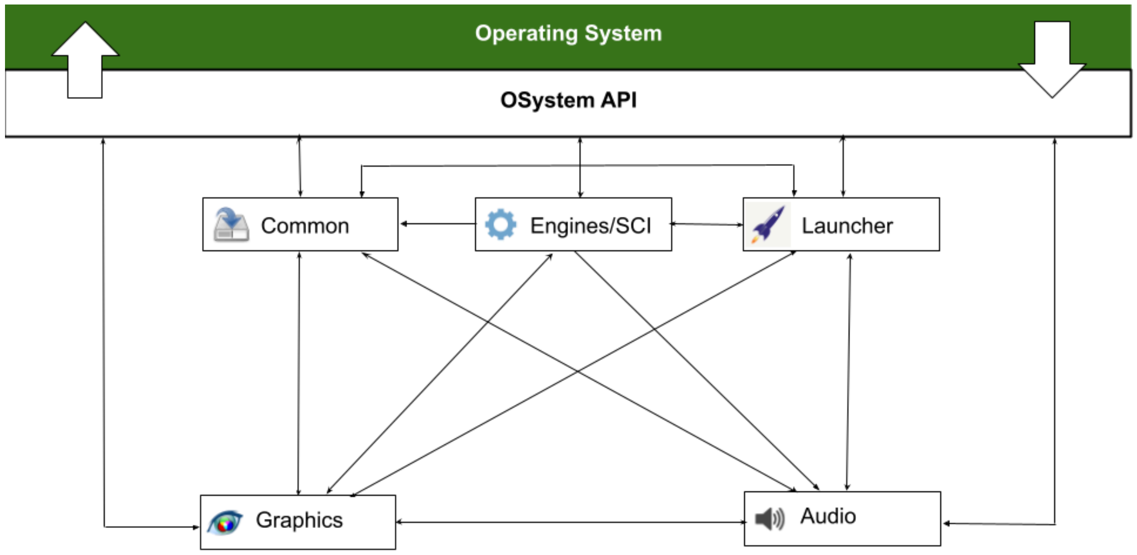

Figure 2: ScummVM Concrete Architecture

Comparing the two architectures, we can see they both follow a layered architectural style. The main differences between our conceptual architecture and our concrete architecture are the several missing dependencies that we didn't foresee in our conceptual architecture, which will be discussed now.

OSystem API: There are three components that the OSystem API component depends on that aren't in our conceptual architecture, the graphics, the engines, and the audio. In the conceptual architecture, we didn't think that the OSystem API needed the graphics component at all. Eventually, one of the programmers added a connection between the two, such that a file in the backends folder relating to OSystem API required a font manager from the graphics component. The rationale stated for this was so that a correctly sized rectangle could be determined in order that the text displayed on the screen would not run out of the box. This was necessary due to the variety of different systems that ScummVM runs on. As for engines, a programmer added a requirement for a function that gets date and time in the "savegame.cpp" file located within the sci/engine folder within the ScummVM project in Understand. The reasoning given for this seems to be that they wanted to be able to use save files directly from the ScummVM launcher, and this needed a separate method of obtaining metadata for the save compared to an earlier way of getting it. There are lots of dependencies from OSystem API to the audio component. This is because the OSystem API component is also responsible for implementing the backends for various operating systems. This means that the OSystem API also needs to handle audio in ways specific to certain operating systems. As such, the OSystem API depends on various functions from the audio component to properly handle audio for each implemented operating system. An example of this is backends/midi/windows.cpp depending on the audio component. Looking back into its commit history, the earliest commit is commit 7d3a423df706 from 2002 where the Windows MIDI driver gets separated into its own file (backends/midi/windows.cpp), and it's been dependent on the audio component since its creation. This dependency seems justified as the OSystem API needs the various audio-related functions from audio so it can implement its backends, and it was likely planned from the start and was an oversight in our conceptual architecture.

Common: There are three components that the common component depends on that aren't in our conceptual architecture, the graphics, the audio, and the launcher. Looking into this with Understand, we can see that common depends on each of these components in the forbidden.h file. Looking into this more, we can see that what actually happens is that "forbiden.h" checks if various tokens are defined, like FORBIDDEN_SYMBOL_EXCEPTION_printf, and if it is, it redefines the associated function, in this case, printf, so it causes a compiler error in the file that FORBIDDEN_SYMBOL_EXCEPTION_printf is defined in. As such, it is more of a dependency from the other components to the common component. The other case for graphics is the use of macros defined in the file "avi_decoder.cpp" from graphics. As for the dependency on the launcher component, other than the forbidden tokens, common has one dependency on the launcher from "random.cpp". This dependency was added in commit f59512c47ea2 in 2013, with the commit message "RECORDER: Implement Events Recorder". Looking at the commit, we can see that the file already included EventRecorder.h, however, the file was moved from the common component to the GUI folder under the launcher component. In doing so, a dependency is created from common to launcher. The other change made in the commit was changing how the seed for "random.cpp" is generated, from using system time to using getRandomSeed from EventRecorder.h. Although we aren't experts on random number generation, we'd like to believe that there are other ways to include seed generation without adding a whole dependency on another component. As such, we believe that this dependency should be reviewed by the ScummVM team and see if it's really necessary and justified.

Graphics: There are three components that the graphics component depends on that aren't in our conceptual architecture, the engine, the launcher, and the audio. The dependency from graphics to engines exists because of the "3do_decoder.cpp" file. This file enables the playback of movies/cinematics of games ported from the 3DO Interactive Multiplayer console initially intended for playing a movie from a SHERLOCK game. This file implements logic to handle video and audio data of various chunk types. It then processes video data using the Cinepak codec and processes audio data using ADP4 and SDX2 which are 3DO-specific codecs. The introduction of this file dependency was created to allow the emulation of 3DO games with FMV sequences. The dependency from graphics to engines exists because of the VectorRenderer.h and VectorRendererSpec.cpp files. "VectorRendererSpec.cpp" was originally a part of the VectorRenderer.cpp file but the Vector Renderer interface was then split from the custom vector renderers in the commit b53220d2 by Vicent Marti on August 31, 2008. "VectorRender.h" and VectorRenderer.cpp were implemented as base API which provided line drawing for graphics. Since VectorRendererSpec.cpp was split off to provide line drawing for the Launcher interface, it would also need a duplicate file of "VectorRender.h". As for graphics depending on audio, in the beginning, we thought of audio and graphics as two different entities. After doing some digging with Understand, the graphics component heavily relies on audio which can be seen in the decoders under /video. The audio module has been used by the graphics module since the early days of ScummVM when it was named "sound". You can see this in commit 42ab839dd6c8 by Max Horn.

Engines: There is only one component that the Engines component depends on that isn't in our conceptual architecture, the launcher. Looking into it with Understand, we can see that the Engines component actually heavily relies on the launcher component. This is because the Engines component utilizes the launcher for things like loading saves, as well as some debugging functions.

Audio: There are two components that the audio component depends on that aren't in our conceptual architecture; launcher and graphics. After doing some digging, we discovered that the Audio component relies on the Launcher component to ensure that audio settings and behaviours are consistently applied and responsive to user interactions in the ScummVM interface. Because the launcher acts as a central point for configuring global settings, including audio preferences, the audio component must access these settings through the launcher component to ensure that all games adhere to the user-defined audio preferences. Looking into the dependency on graphics, there are only four references to the graphics component from within audio, which makes sense as we expected none and is very little. Looking further, we can see that all the references to graphics are made from one file, audio/softsynth/mt32.cpp. Using git, it was found that the dependency was added in 2004 (or earlier, as git was invented in 2005, so that commit was probably manually changed). Looking at the commit from 2004, the commit message says "Added graphical representation of initialization process. This is quite hacky", among other unrelated things. This commit also adds the lines that include graphics/surface.h and graphics/font.h, which introduces the dependency on graphics. Another commit a year later then added graphics/fontman.h, as this commit added a new font manager to graphics, and “mt32.cpp” includes “font.h” so “fontman.h” was also added. Finally, in 2011, the file was edited to also include graphics/pixelformat.h. Since the dependency was present from the start, as well as the fact that it was never changed and instead slightly added onto, this is likely a planned dependency. It’s also possible that this wasn’t a planned dependency, but have decided to allow an exception for this one file to access the graphics component.

4. SCI Subsystem

4.1 Architecture Style

In our conceptual architecture analysis of ScummVM, we stated that the conceptual architecture of SCI is an interpreter style. While we believe this to be true, the interpreter style part of SCI is more of its own subsystem in the overall ScummVM SCI subsystem. We believe the architectural style of the SCI subsystem in ScummVM to be a repository architecture. With this architecture, the engine component is the main component, which uses all the other components to perform various tasks, such as rendering graphics or outputting audio. This is because the engine component is responsible for running the game, and all the other components act on the game being run, so the engine acts as the repository of data for the other components to act on.

4.2 Components

Common: Similar to the ScummVM architecture, the SCI subsystem also has a common component of its own. The conceptual view of this component is to contain utility functions that can be useful to all the other components. As such, it is depended on by every other component. In a more concrete view, this component contains files with various useful functionalities to other components, such as a util file, an event handling file, and a debug file. These files don’t really share a common theme like files in other components, other than being useful or required by many other components. This component also contains “sci.cpp”, which seems to be the main file of the subsystem that ties everything together. The common component also depends on everything, which are all divergent dependencies in the conceptual architecture, but most or all of them are likely due to the main file (sci.cpp) being included in the common component.

Engine: The engine component is the component responsible for running the games. Going back to report 1, the engine component is responsible for emulating the PMachine, which is a virtual machine that runs the games written for the SCI engine. This is done through the “vm.cpp” file found in the engine component. The engine component also contains various other files related to running the games such as a “savegame.cpp” file and movement and pathing files called “kmovement.cpp” and “kpathing.cpp”. The component depends on all the other components, as was expected with the conceptual architecture.

Sound: The sound component is pretty self-explanatory, it is responsible for all things relating to sound for the SCI subsystem. It contains various decoders, drivers for audio systems, and various other functionalities related to sound. It depends on all the other components except for the parser. The dependencies on common and resource manager are expected, as common contains various useful utilities, and the resource manager is responsible for handling resources such as patching audio files. The unexpected dependencies on the graphics and engine components will be looked into later.

Graphics: Similar to the sound component, the graphics component is also quite self-explanatory, as it is responsible for all things related to graphics, whether that be picture-related or video-related. The component contains many files related to rendering graphics on screen, such as files for animation, rendering the cursor, and displaying fonts, as well as some video decoders. The graphics component has dependencies with every other subcomponent. The dependencies documented in the conceptual architecture are sound, as sound is needed for videos, resource manager for managing and patching graphical resources, and common for various utilities. The unexpected dependencies are the engine component, and even more interesting the parser component, which will both be looked into later.

Resource Manager: The resource manager is a relatively small but important component. It is responsible for decompressing as well as patching various kinds of resource files, whether that's audio, video, or game logic files. Conceptually, it should only have a dependency on the common component, as the resource manager is more of a tool for the other components to use, however in the concrete architecture, it depends on everything except sound, and all of these divergences will be investigated.

Parser: The last component is the parser. The parser is responsible for parsing user input, particularly for games with text-based input. The parser contains grammar and vocabulary files, which are used to parse user input so the engine can determine what to do based on what the player inputs. As such, the conceptual architecture has just the common component as a dependency for the parser, however in the concrete architecture, it also depends on the engine and resource manager components, which will be looked into.

4.3 SCI Reflexion Analysis

We only briefly talked about the architecture of SCI in our previous report, however, we mentioned that we believed the architecture to be an interpreter style. We still believe this to be true, but in ScummVM, the SCI subsystem is more than just the SCI engine, and so our conceptual architecture of SCI from our first report is more of an overview of the engine component of the SCI subsystem in ScummVM. As such, we have created a new conceptual architecture for the SCI subsystem, based on the concrete components. Using these components, we created dependencies where we thought made sense to have them given the use cases we thought up. Going forward, that will be the conceptual architecture used for reflexion analysis with the concrete architecture.

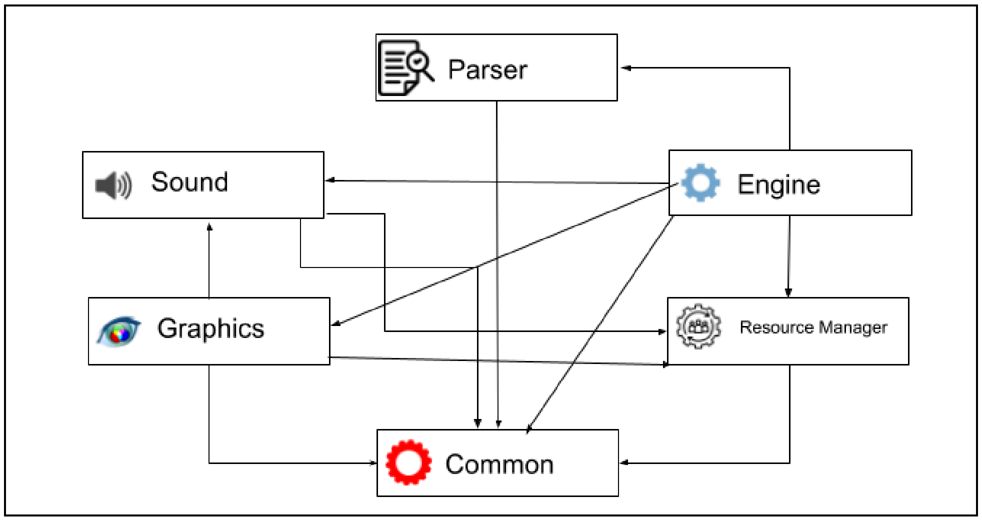

Figure 3: Conceptual Architecture of SCI Subsystem

Comparing the overview of our conceptual architecture to our concrete architecture, we can see that there are quite a few divergences in the concrete architecture from the conceptual architecture. In order to understand the differences in the two architectures better, we will investigate each divergence on a case-by-case basis.

Common: The biggest divergence you may notice is common depending on no components in our conceptual diagram, while it depends on every component in the concrete diagram. After looking into this divergence, it becomes clear why it depends on every component. The common component contains the file "sci.cpp", which is the main file of the SCI subsystem. As such, it is required to use all the different components to fulfill its role.

Graphics: There are two components that the graphics component depends on that aren't in our conceptual architecture, the engine and the parser. The graphics component is heavily reliant on the engine component. After looking into it with Understand, it becomes clear why. The engine component defines a lot of data types to be used throughout the SCI subsystem by other components. It does this as everything that will be used by the SCI engine needs to be compatible with the engine, and needs to fit the limitations of the engine, which these data types enforce. As such, the graphics component also uses these data types to ensure that it is compatible with the engine. Another example of graphics being dependent on the engine component is that certain files, such as "cursor.cpp", which renders the cursor, rely on the EngineState, which is another way that graphics are dependent on the engine component outside of datatypes. With this knowledge, it's very likely that this was a planned dependency and is justified. Graphics depends on the parser component in two files, the first one being graphics/controls16.cpp. This file calls checkAltInputs twice from parser/vocabulary.cpp. Based on the comments, it seems like it's doing this to make sure there is space for a new character to be displayed in an input field. Looking at the commit history, this dependency was added in 2010, so it was not present from the start. This use case seems like something that should be handled by the engine component or common component, and should not be a direct call from graphics to parser. The other dependency is from graphics/menu.cpp. From what we can tell, this dependency comes from the menu using text input as a means of selection. This was added in commit d1dc586aa39c from 2009. This also seems like something that should be handled by the engine or common component and not called directly by the graphics component, as the graphics component shouldn't be dealing with input handling directly. It's also possible that this was a decision made by the original SCI team and the ScummVM team is trying to keep it as accurate as possible. Since neither of these dependencies existed from the start, it is likely that this is an unintended dependency and should probably be looked into by the ScummVM team, as there are likely better ways to handle these two use cases.

Parser: There are two components that the parser component depends on that aren't in our conceptual architecture, the engine and the resource manager. The engine dependency can be explained with the same reasoning as why the graphics component is dependent on it; the parser also uses multiple types defined by the engine component, as well as functions related to these types, and also the engine state. The parser is dependent on the resource manager component, as the parser needs to load the vocabulary file for the game being played, which it uses functions from the resource manager component to do. The file that relies on the resource manager the most is "vocabulary.cpp", and the earliest commit for that file is df149e1509d9, from 2010, which separates the parser into its own files. From this, we can see that the parser has always relied on the resource manager. The more interesting thing about this commit, however, is that the parser code wasn't originally separated, meaning that this component likely wasn't planned from the start, which makes the divergences related to the parser component make a lot more sense. Due to the heavy use of resource manager in the parser, and the fact that it makes sense for the resource manager to load the games vocabulary for the parser, dependency makes sense and shouldn't be an issue.

Sound: There are two components that the sound component depends on that isn't in our conceptual architecture, the engine and graphics. The sound component depends on the engine for the same reason as the previous two components, as well as using helper functions like useAltWinGMSound() which controls whether the engine should use an alternative MIDI sound configuration. The sound component depends on the graphics for one file called "robot_decoder.h" which defines the implementation details for decoding and rendering Robot Video Format used in SCI games. It provides functionality for decoding, rendering, and handling audio/video synchronization of Robot video files. This implementation was added in 2016 by Colin Snover to update the version of Robot Decoder they were using.

Resource Manager: There are three components that the resource manager depends on that aren't in our conceptual architecture, the engine, the parser, and the graphics. The Resource Manager depends on the Engine component for a few files which are a collection of files for patches for various script bugs or inconsistencies in SCI. Instead of fixing the actual bugs that are causing the issues, the developers decided to create workarounds. All bugs for which these workarounds are based should be fixed, and these files deleted. These workarounds add unnecessary bloat and complexity to this project. Resources depend on Parser for one file named “vocabulary.h” which is responsible for handling and parsing vocabulary resources for SCI games. This is essential for parsing player input. Vocabulary resources (VOCAB) in SCI games store information like game-specific words, verbs, and selectors, which are essential for game interactions and script processing. The file supports various SCI versions and VOCAB management differs across these versions. The resource manager depends on the graphics component for the file “helpers.h”. The graphics helper “helpers.h” has been relied on for quite a long time dating back to 2010 when it was added to breakup “resource.h”, a file that defines ViewType for SCI to use. This dependency allows for SCI to avoid duplication of the ViewType definition. This helper file is the only file that the Resource Manager depends on under the Graphics component.

5. Use Cases

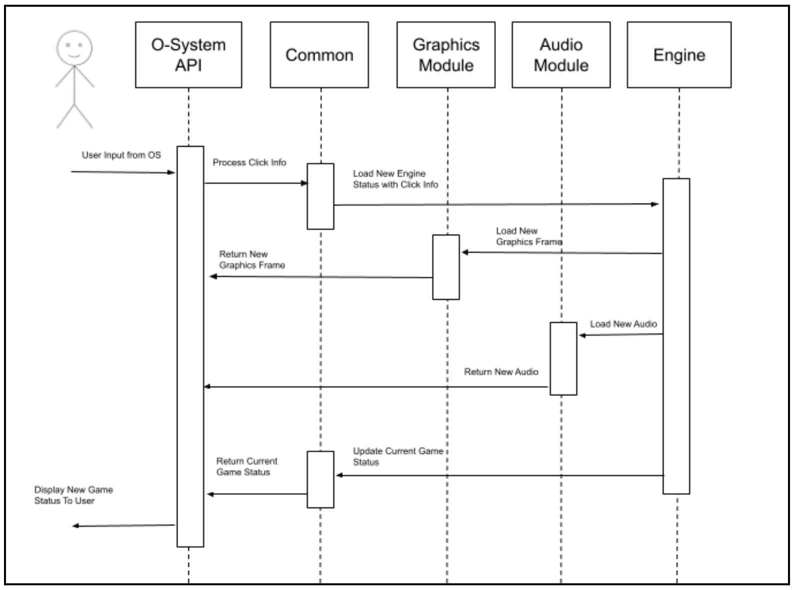
O-System API
Common
Graphics Module
Audio Module
Engine
User Input from OS
Process Click Info
Load New Engine Status with Click Info
Load New Graphics Frame
Return New Graphics Frame
Load New Audio
Return New Audio
Update Current Game Status
Return Current Game Status
Display New Game Status To User

Figure 4: Use case 1: ScummVM loads the required components to register a click in a point-and-click game.

The first use case assumes that ScumVM has already booted a point-and-click game from the initial ScummVM launcher and the game is currently being played by the user. The diagram starts with the user clicking their mouse. That click is registered by the host OS and passed off to the OSystem API. The OSystem API uses the Common component to send the click information to the Engine component. The Engine component then decides logically what should happen in the game, and then sends the new Audio and Graphics data to their respective components. Then those components hand off the newly generated audio and graphics data to the OSystem API to display to the user. Once the in-game logic is computed by the Engine component it is sent back to Common and then off to the OSystem API to update the game status to the user.

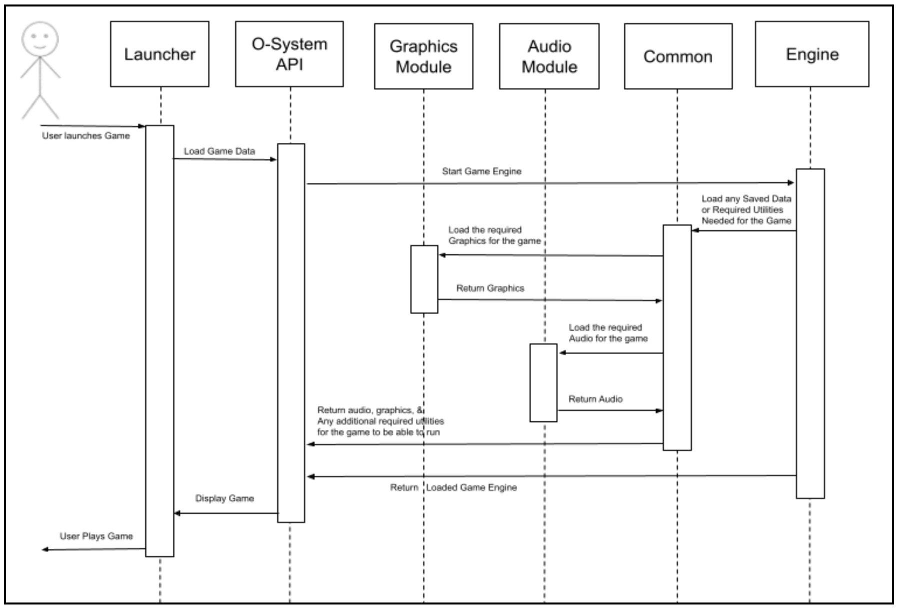
Launcher
O-System API
Graphics Module
Audio Module
Common
Engine
User launches Game
Load Game Data
Start Game Engine
Load any Saved Data or Required Utilities Needed for the Game
Load the required Graphics for the game
Return Graphics
Load the required Audio for the game
Return Audio
Return audio, graphics, & Any additional required utilities for the game to be able to run
Return Loaded Game Engine
Display Game
User Plays Game

Figure 5: Use Case 2: ScummVM loads the required components to successfully launch a point-and-click game

The second use case assumes that ScummVM is booting a point-and-click game from the initial ScummVM launcher. It starts with the user choosing a game to launch through the launcher. After the user has selected a game to play, the launcher calls the O-System API to start the conversion of the engine of the selected game as well as any required graphics, audio, or additional utilities to run and play the game. The O-System API then calls the engine to start loading the converted game engine required to play the game. Firstly, it loads the common module, which is responsible for loading all required utilities to the API, including the graphics module and the audio module. The common module loads the required graphics and audio for the game and returns them to the common module to send all required utilities to be converted into utilities that the user's operating system can understand through the O-System API. Finally, after all required utilities have been returned to the API to be converted, the loaded game engine also returns to the O-System API for it to be loaded through the API to the user as well. After the O-System API has all utilities of the game loaded successfully as well as the loaded game engine, it now displays the elements of the game to the user, who can now play the game.

6. Conclusions

Since we had already conceptualized the architecture of ScummVM in our first report, we felt pretty confident that we had a good understanding of what the concrete architecture would look like. While we were correct for the most part, there were still many differences as we observed how intertwined the components were. We saw that there were 6 key components to ScummVM which were: Engines, Audio, Graphics, Launcher, OSystem API, and Common. We also went further in-depth and examined the SCI engine architecture to find that its main components were Engine, Sound, Graphics, Resource Manager, and Parser. Subsequently, we recreated our Dependency diagrams to graphically show the dependencies between the components. We also redesigned our Use Case diagrams to fit these dependencies. Furthermore, we made a reflexion analysis for both the ScummVM and SCI architectures to determine what the divergences were and how they came to be.

Our outlook for ScummVM has not changed since the last report. We still believe that ScummVM has the potential for growth and development. If anything, we have become more fond of ScummVM and its development team. During our reflexion analysis research, we witnessed the git commits of the team members and realized how much effort they put into making ScummVM work. Overall, we have much respect for ScummVM and its team of developers and hope they can continue thriving way into the future.

7. Lessons Learned

We have learned that deriving a concrete architecture can be challenging, even with specialized tools such as Understand. Unless you have a clear understanding of how the system is intended to function it takes time to figure it all out. The naming conventions can sometimes be cryptic which can further complicate the process. We also realized the crucial role reflexion analysis plays in helping to identify the purpose of various elements and determine whether they should remain in the design. Flexibility in problem-solving is essential; a static perspective from the start can lead to challenges if the architecture later reveals itself to function differently than initially expected.

8. Naming Conventions

API: Application Programming Interface
FMV: Full-Motion Video
GUI: Graphical User Interface
MIDI: Musical Instrument Digital Interface
SCI: Script Code Interpreter/Sierra's Creative Interpreter

9. References

[1] *SCI. (n.d.). ScummVM :: Wiki*. https://wiki.scummvm.org/index.php/SCI

[2] *TCPlusPlus.org*
https://tcplusplus.org/documents/CISC%20322_326%20-%20Software%20Architecture%20Report%201.pdf

[3] *Scummvm. (n.d.). GitHub - scummvm/scummvm: ScummVM main repository. GitHub.*
https://github.com/scummvm/scummvm