CONCRETE ARCHITECTURE OF SCUMMVM

CISC 322/326 A2 Report

18 November 2024

Team MAWLOK

Lance Lei (20lh10@queensu.ca)

Michael Marchello (21mgm13@queensu.ca)

Owen Meima (21owm1@queensu.ca)

Kevin Panchalingam (15kp20@queensu.ca)

Andrew Zhang (21az75@queensu.ca)

Zhuweinuo Zheng (21zz28@queensu.ca)

TABLE OF CONTENTS

Abstract	2
Introduction	3
Review of Conceptual Architecture	3
Overview of Concrete Architecture	4
New and Altered Subsystems	6
Reflexion Analysis of New Dependencies	6
Second Level Analysis of SCI Engine	9
Sequence Diagrams	11
Derivation Process	12
Lessons Learned	13
Concluding Statements	13
Important Terms	14
References	15

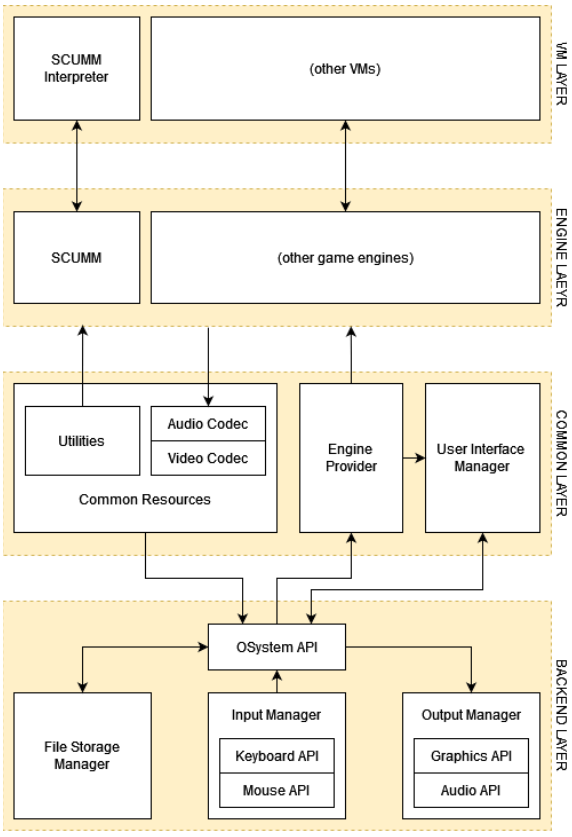
ABSTRACT

This report will explore the open source software, known as ScummVM, and explain its functionality and low-level concrete architecture. Using SciTools Understand, we transitioned from a high-level conceptual architecture to a detailed concrete architecture, identifying key divergences and absences. These insights allowed us to revise to a three-layered architecture, merging the Engine and Common Layers into a new Middle Layer. A focused analysis of the SCI Engine subsystem revealed its dependency structure and optimizations for 2D point-and-click games. This investigation highlighted the complexities of large software systems and the value of tools and documentation in understanding and refining their architectures. Using our new knowledge of the software, we created two sequence diagrams that detailed two use cases. We also documented the things we learned while determining the concrete architecture.

INTRODUCTION

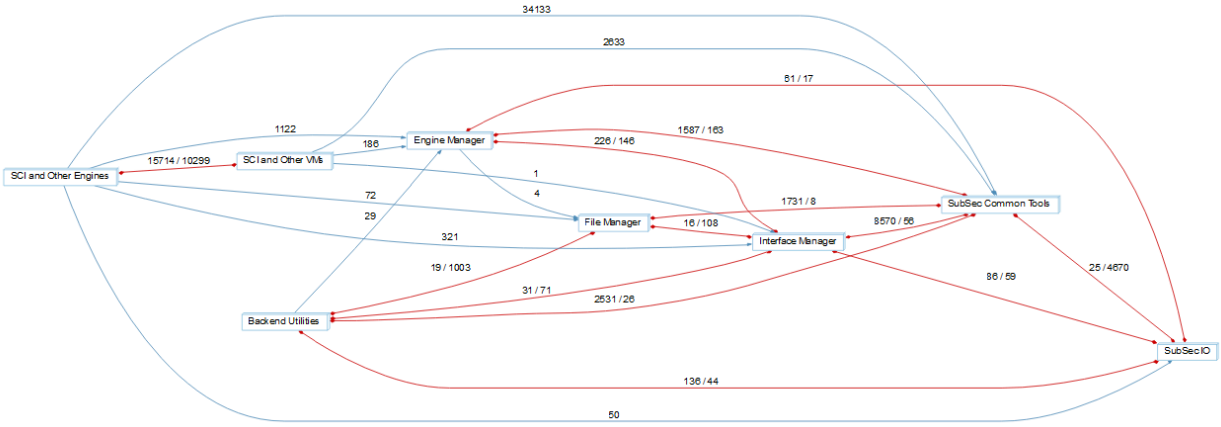
In our A1 deliverable of the project, our objective was to gather a high-level understanding of ScummVM. As previously determined, ScummVM allows users to play classic games that were previously platform specific, on any supported platform. We proposed our high-level understanding of ScummVM through a conceptual architecture of the system, along with its architectural styles, components, and dependencies between components. Although much of our conceptual understanding seemed reasonable, we now had access to SciTools Understand software that enabled us to do an in-depth analysis into the ScummVM source code. This leads us into the A2 deliverable of our project, in gathering a more detailed, concrete understanding of the ScummVM software, and an opportunity to investigate how the determined concrete architecture varies from our conceptual understanding. We will discuss our findings during this investigation process, and propose a revised conceptual architecture in this next phase.

REVIEW OF CONCEPTUAL ARCHITECTURE

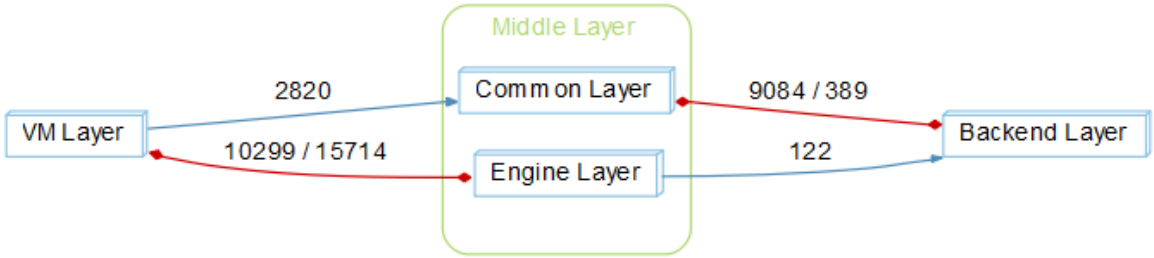


The previously designed conceptual architecture for ScummVM consisted of four distinct layers, each with a particular set of responsibilities, and each communicating solely within adjacent layers. The layers were as follows: the backend layer (containing all platform-specific code), the engine layer (containing an array of ScummVM engines), the common layer (containing various utilities and helper functions for the backend, engines, and GUI), and lastly the virtual machine layer (containing tools for the engines to translate old game bytecode into more readable, up-to-date scripts). Within each layer, there exists a variety of distinct components that suit a unique purpose and/or process a unique type of data. For example, each component within the engine layer is a unique game engine implementation capable of powering a certain set of playable games. Importantly, all arrows in our original conceptual diagrams relate specifically to *data flow* (rather than 'dependency'). The conceptual architecture focused entirely on where various data/files types were being processed, and did not focus on *all* dependency structures needed to completely fulfill the data flow.

OVERVIEW OF CONCRETE ARCHITECTURE

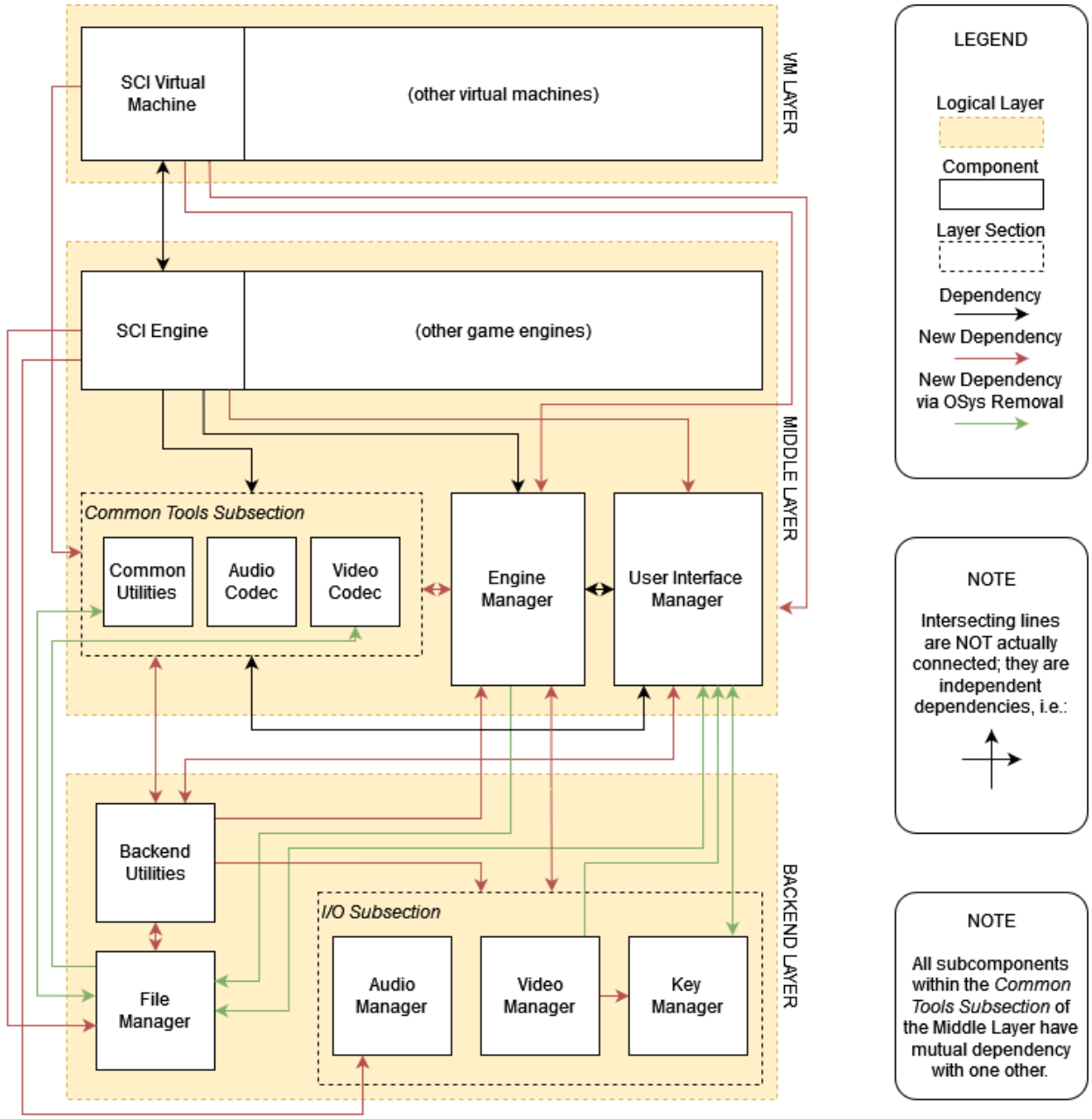


The concrete architecture presented in this report contains a variety of key differences from the previously derived conceptual architecture. Firstly, the prior ‘engine’ and ‘common’ layers have been merged into a unified ‘middle layer’, turning the architecture into a three-layered one. This merge was, in part, the result of discovering a vast number of helper functions within the old ‘common layer’ existing as dependencies of the various top-layer virtual machines. In addition, the engine layer turned out to utilize various functions contained within the backend source code. The new three-layer architecture is reinforced by the fact that the VM layer contains no dependencies with the backend whatsoever, and vice versa.



The concrete architecture also features a complete removal of the OSystemAPI, which had been a vital component in our conceptual architecture. This removal was performed following the realization that the OSystemAPI was not a literal ‘shield’ between the backend and above layers, but rather, a superclass of the entire backend layer itself. In other words, the OSystemAPI encapsulates a number of backend function *declarations* that indicate what tools the game engines have at their disposal. Then, each platform-specific (i.e. ‘Windows’, ‘iOS’, etc) variant of the ScummVM backend layer will contain the actual *implementations* of these functions. Thus, the game engines may have dependencies on the platform-specific backend, without themselves being platform-specific.

Lastly, the concrete architecture features certain rearrangements/renaming of components from the conceptual architecture. For the sake of clarity, the ‘Common Resources’ component from the conceptual design has been turned into a ‘common tools subsection’ with separate utility and codec components. The conceptual ‘engine provider’ is now called ‘engine manager’ to reflect newly-discovered functionality. Due to them possessing similar dependencies, the SCI engine and VM have been merged with the ‘other engines’ and ‘other VMs’ boxes (also, since our provided source code featured the SCI engine instead of SCUMM, the latter is no longer referenced). Finally, the backend layer has been significantly altered to reflect its actual source code structure, and these changes are further described below.



NEW AND ALTERED SUBSYSTEMS

In the conceptual architecture, there is an ‘engine provider’ box that is supposed to check if a game data file is valid for one of the above game engines. The new diagram has this replaced with ‘engine manager’. It contains all the code in the source code ‘engines’ folder that is not in any *specific* engine folder (i.e. files directly in the ‘engines’ folder). This code does what the engine provider does but also contains various functionality common to all engines (such as saving your progress, storing achievements, etc).

The backend layer features a new subcomponent named ‘backend utilities’. This box contains either backend-related helper functions for the rest of the codebase, or contains some other functionality non-related to any of the other four backend subcomponents. By contrast, ‘common utilities’ (called simply ‘utilities’ in the conceptual) contains helper functions and miscellaneous files that are not directly located in the ‘backends’ source code folder (and thus are not platform-dependent). Additionally, in the original conceptual diagram, the backend features the two components ‘input manager’ and ‘output manager’, which themselves contain APIs for keyboard/mouse/video/audio. This design has been shifted in the concrete architecture, as now there only exist the ‘video’, ‘audio’, and ‘key’ manager components within a ‘I/O subsection’. The dissolution of a separate mouse component was done due to the backend code not containing a distinct section for mouse input; this functionality was rather included alongside the rest of the keybindings.

The removal of the OSystemAPI produced a number of ‘new’ dependency lines, coloured in green on our concrete diagram. These lines do not represent new functionality, as they merely represent the same arrows that previously existed (in the conceptual) between the backend layer and above, just without the OSystemAPI as a middle-step.

REFLEXION ANALYSIS OF NEW DEPENDENCIES

Engine Manager ↔ Common Tools Subsection
For this divergence, most of the dependencies are between common utilities and engine manager. In our conceptual report, the Engine Manager (used to be called Engine Provider) is considered to have the ability of detecting if the given files make up a game designed for any of the engines within the engine layer. In Understand, it is shown that the encapsulated various functions and data structures in Common Utilities do not only serve for the engines but also serve the Engine Manager for detection.

Sci Engine → File Manager
This new dependency connects savefile.cpp with the engine.cpp and metaengine.cpp in the engine manager. These 2 .cpp files use ‘savefile.cpp’ to save the game state. The conceptual box-and-line diagram was focusing on the functions mentioned in the use cases, so necessary data flow for saving progress a file for a game is not shown on the conceptual diagram, which leads to the divergence here.

Sci Engine → User Interface Manager
The majority of the dependencies between them is from ‘console.cpp’ to ‘debugger.cpp’. This means they want players to use the command when playing a game. Also, in our conceptual architecture the user interface is defined to only serve for the ScummVM software itself without connecting to the game engines. However, there are many functions about printing messages to the players in UI Manager, thus, the SCI engine uses them to show messages to the players.

Engine Manager ↔ I/O Subsection
For those with the key manager, most dependencies are calling from method ‘initKeymaps()’ in metaengine.cpp. initKeymaps() define multiple keymaps which may be turned on or off in the engine code. The dependencies with the video manager relate to the ‘Openglsdl’ library, used in part for scaling games to desired resolutions. Finally, the dependencies with the audio manager involve references in ‘engine.cpp’ to ‘audiocd.h’, which helps to validate audio files.

SCI Engine → Audio Manager
There are only 2 dependencies for this divergence. The audio.cpp in SCI Engine calls the “getStatus().duration” method in ‘audiocd.h’ in Audio Manager to check if the sample is done playing. When making conceptual architecture, the checking situation is not considered since it is too detailed.

Virtual Machines → User Interface Manager
There is only 1 dependency for this divergence. The vm.cpp calls an ‘onframe()’ method in debugger.cpp to have a temporary execution handler. The SCI Virtual Machine was expected to have all functions it needs inside it when making conceptual architecture, so didn’t consider the situation of it using helping functions from User Interface Manager.

Virtual Machines → Engine Manager
In this divergence, ‘shouldquit()’ and ‘_system’ are 2 most used functions. The former returns the MetaEngine instance used by the VM’s respective engine, and the latter creates the instance of that engine. The Virtual Machines were expected to have all functions it needs inside them, but now it shows that it needs helping functions about engines from Engine Manager.

Virtual Machines → Common Tools
For this divergence, most dependencies are made because the SCI Virtual Machine calls a ‘typedef’ in common utilities. SCI Virtual Machine was defined to translate old game bytecode into more readable, up-to-date scripts and supposed to have all functions it needed inside the component, but in reality it calls ‘uint16’, ‘int16’, etc to create the correct type of variables. This helps in making the code portable and reducing platform-specific compilation issues.

Video manager → Key manager
In the ‘SdlGraphicsManager’ module file, which calls ‘sdl-graphics.cpp’ there is a ‘getKeymap()’ function returning a pointer to an object for key mappings. For example, if the system supports full screen mode, a new action for toggling fullscreen is added to the keymap.

Backend Utilities → Engine Manager
The backend may need to communicate with the engine manager in order to trigger the starting/halting of the ongoing running engine. Since tasks such as saving/logging are fundamentally file (and thus backend) based, the backend will require these kinds of connection points with the engine manager.

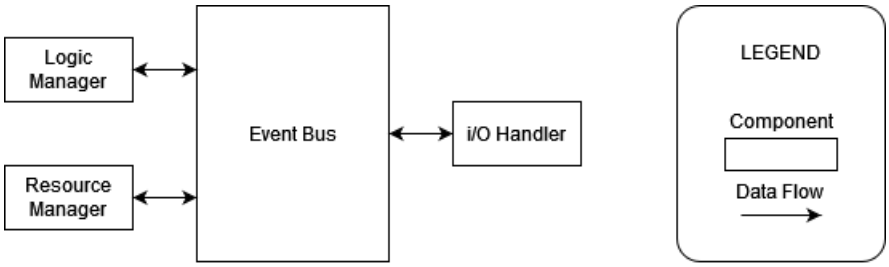
File Manager ↔ Backend Utilities
The backend file manager uses a plethora of helper functions related to cloud storage. In this case, the backend utilities contain a variety of remote data saving capabilities which the file manager seeks to use. Likewise, the vast majority of these dependencies go outward from the file manager.

Interface Manager ↔ Backend Utilities
Like above, the backend utilities component has a variety of functions related to cloud storage. The interface manager contains the code for the main ScummVM GUI, which itself contains options for cloud storage/connection.

Backend Utilities ↔ Common Tools
Both of these components contain many utility and helper functions for other classes/components throughout ScummVM. There is significant intertwining between the code of each one (i.e. the majority of files in both components have some dependency with some in the other). Recall, the only reason these components are split is since the functions inside backend util are those implemented as a part of the unique, platform-specific backend implementation.

SECOND LEVEL ANALYSIS OF SCI ENGINE

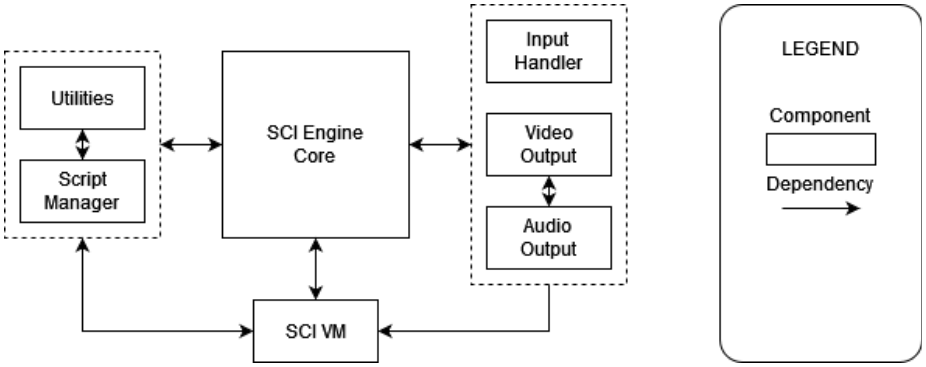
SCI Conceptual Architecture



For this report, the subsystem of the SCI engine was chosen for deeper investigation. Part of the rationale for this decision was that our previous investigations of the high-level conceptual architecture focused heavily on interactions with the backend- and utility-related components. By looking into the functionality of the SCI engine specifically, we may increase our understanding of higher-layer code implementation.

The initial conceptual architecture for the SCI engine was derived from our general understanding of the core functionality of game engines (such as Unity, etc). Various external sources were read to affirm our notions of what core components made up an engine. In general, our conceptual architecture for the SCI engine involves four simple subcomponents in a publish-subscribe structure, acting out the fundamental functions of a typical game engine. This included an event bus to/from which user input/output could be sent. Based on the nature of the incoming input, the event bus would consult two subsystems related to logic (i.e. coded game functionality) and resources (i.e. stored/cached image or audio files) in order to produce the necessary responses a user would expect. For example, if a user clicked a left-arrow key, the event bus would receive a message from the I/O handler, and then consult the logic manager on the necessary response. The logic handler would then indicate back to the event bus that the user must be shown their character moved to the left, and the I/O handler would fulfill this task.

SCI Concrete Architecture



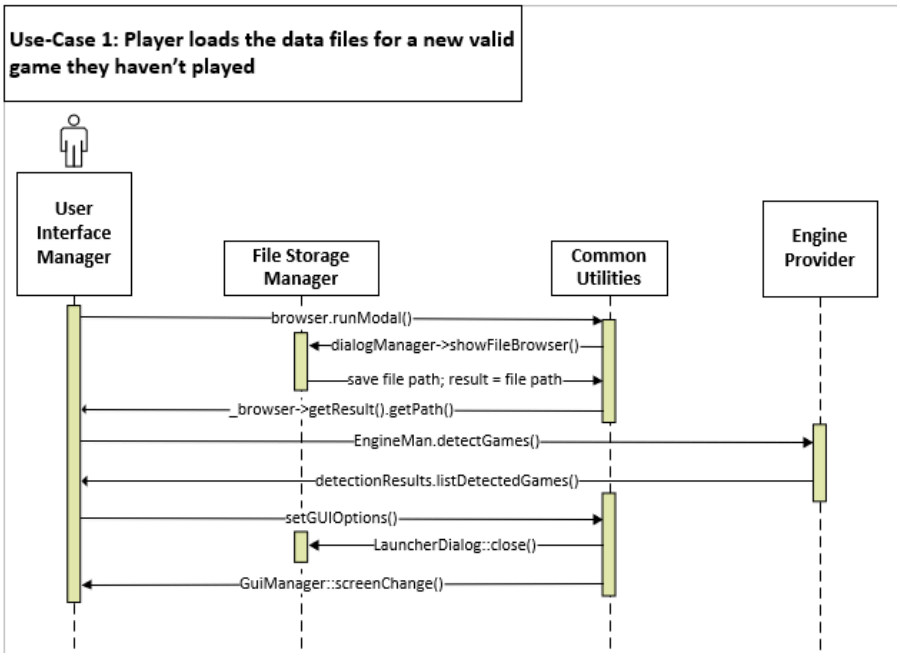
In developing the concrete architecture for the SCI engine, the source code directory structure was investigated and organized into relevant subcomponents. These subcomponents were renamed and/or expanded (from the conceptual) to reflect their more-specific purpose. The SCI engine core contains an event driver in addition to code facilitating the use of surrounding components. The I/O section has been expanded into three specific elements for input, video, and audio. Most importantly, a specific reference to the SCI virtual machine has been placed. Though this component is technically a part of the VM layer overall, its source code files are located alongside the rest of the engine code. The SCI VM interacts with the entire engine system, including the script manager (which helps process 'translated' game logic') and utilities (helper functions and resource caches).

SCI Reflexion Analysis

The central (core) component of the engine architecture remained connected to all other components surrounding it. However, the addition of the SCI VM as an important related element to the engine brings in a number of new dependencies. The script manager ultimately contains the core classes and functions utilized in the game implementation the VM reads from the bytecode. The process of this data flow requires a mutual dependency between these two elements. Additionally, the VM makes use of helper functions and visual resource management from the utilities component. Finally, the input and output components make use of the VM code due to various optimization functions found within it. Each of these dependencies are divergences from the initial engine architecture due to the VM component not initially being included. The VM's interaction with the various engine components was crucial in understanding the data flow surrounding the engine.

Extra dependencies non-related to the VM were also found between the utilities and script manager components. The majority of these dependencies are outward of the script manager, which needs to take use of certain functions in managing visual game assets as they are used in the game logic/progression. Finally, the video and audio managers were found to contain mutual dependencies following the expansion of the conceptual I/O component. The video/audio components use various functions from each other to ensure proper synchronization between visual output and relevant game sounds.

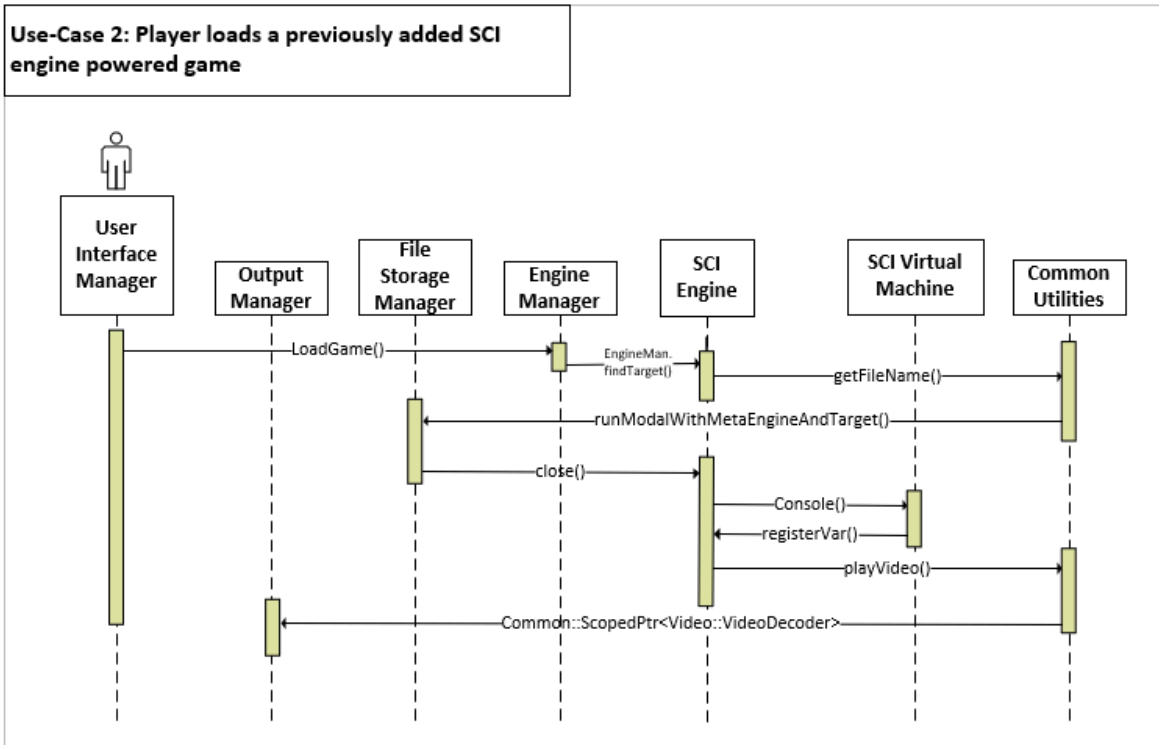
SEQUENCE DIAGRAMS



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

1. When the user selects Load Game, the User Interface Manager sends a request to the Common Utilities to open the user's file explorer by calling the runModal() function. This function opens the user's file explorer.
2. The Common Utilities displays the user's file explorer by directing a function call to the File Manager through function showFileBrowser().
3. The File Manager returns the file path of user selected game files saved in variable "result".
4. The User Interface Manager then checks with Engine Manager to determine if it's a valid game by calling getResult().getPath().
5. Since our use case only considers "valid" games, a game engine will be found, and a game or array of games will be provided to the user interface through function call detectionResults.listDetectedGames() where the user can select a game and desired game settings.
6. User selects game they want to play and preferred graphics/audio settings which triggers the Common Utilities through function call setGUIOptions() to save the selected game and settings in user's chosen file directory
7. Users can now see the saved data on the user interface and close application which triggers a call to function close(). This function saves the user's last settings selection.

8. The User Interface is refreshed



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

1. The user has triggered the “Start Game” button on the main menu so a call is made to the the engine manager through function LoadGame(), to find the engine capable of running the game
2. The Engine Manager calls on SCI Engine through the function EgineMan.findTarget(). Since our use case only looks at valid games, we only consider valid scenarios.
3. The SCI Engine then calls the Common Utilities to get the file path of the user selected game through the function getFileName().
4. Common Utilities calls the function runModalWithMetaEngineAndTarget() which gets the path of the selected file for the SCI Engine to run.
5. The SCI ENGINE calls its engine specific interpreter (the SCI Virtual Machine) through the function call to Console(). This call will help SCI Engine determine which graphics/audio/images such as game state or mouse click tracking are required for the game.
6. The SCI Virtual Machine provides the information in step 5 above through the function call to registerVar().
7. The SCI Engine then runs the game and sends video output to the user screen through the function call to Common Utilities, by function playVideo().
8. The Common Utilities sends required output to the Output Manager through function call ScopedPtr<Video::VideoDecoder>.
9. The Output Manager (part of the backend) works with the user’s operating system to provide visuals to play the game.

DERIVATION PROCESS

The derivation process to determine the concrete architecture of ScummVM was lengthy. We started by determining how SciTools Understand software worked. Once we loaded ScummVM's repository into the software, we needed to transfer our team's high-level understanding of ScummVM architecture into the Understand software before we could map this understanding to ScummVM's "concrete" files. Our next step was to create components in Understand to mirror our original conceptual architecture proposal so we could then map ScummVM's files to each component. The mapping process was tedious as we first started looking at individual files, but quickly realized that looking into each and every file individually to complete the mapping process would not be feasible. We then took the approach of mapping folders of files to components based on folder names, or API documentation, and then reviewed dependency graphs to see the impact of our mapping decisions. If the created dependencies reasonably reflected our high-level thinking, we proceeded until the mapping process was complete. If during the mapping process, we found a large number of dependencies that did not corroborate our high-level thinking, we then took a closer look at the files causing the divergences to ensure that mapped files were not tagged to the wrong components. We now had a true picture of our divergences and absences.

With mapping complete, we now had our first visual of the concrete architecture which contained many divergences. We proceeded to divide and conquer within the team by splitting the tasks of looking into and explaining each divergence and absence. During this process we learned that although the OSystem API acts as the middle-man between the user's platform and the ScummVM software, it is not itself a component, rather it just works in the background to relay events from the user's platform to components within the ScummVM software. Additionally, we noticed that various sources of ScummVM's documentation and forums explicitly describe ScummVM as a three-layered architecture, which varied from our four-layer structure seen in our original conceptual diagram. This was originally done for the sake of clarity in showing general data flow between parts, and not the full dependency structures needed for that flow. After investigating various divergences and absences, we then proposed a revised conceptual architecture that aligned with our findings during the investigation phase.

With a revised conceptual architecture now in hand, we made modifications to our original two sequence diagrams submitted with the A1 deliverable. The main change included removing the OSystem API and redirecting the data flow directly to components rather than through the API with understanding that the API is just running in the background to implement this data flow. This determination of redirected data flow, along with finding support for our original sequence diagram flows involved in-depth analysis of ScummVM's source code, including a thorough analysis of specific functions calls within files and within components to determine how ScummVM worked.

Before wrapping up the project, we derived the inner architecture of one subsystem, the SCI Engine. The architecture was found by researching the core components of any general game engine, and then determining (based on the file structure of the SCI source code) which components were most relevant.

Looking at ScummVM's software within the SciTools Understand software was rewarding, and really helped us get a better understanding of how complex systems can be better understood using such tools.

LESSONS LEARNED

In the process of determining the concrete architecture for this deliverable, we developed a better understanding of the SciTools Understand software. We learned how Understand can be used to get visuals on the number of dependencies between components, and also gave us a convenient way to jump between function calls to see how the user experience flows through the source code. In using the SciTools Understand software, we quickly realized the benefits of such tools in assisting the process of determining how large complex systems work. A better understanding and appreciation of proper documentation of systems was also developed throughout this A2 phase. Documentation such as ScummVM API documentation and the Developer Central Wiki Page proved a valuable reference during our reflexion analysis, and also in understanding class structures and hierarchies within the ScummVM system.

CONCLUDING STATEMENTS

ScummVM is an open source software that allows users to play 90s and 2000s games. The SciTools Understand software assisted us with the reflexion analysis of ScummVM. The analysis exposed variances between our original proposed conceptual architecture and the concrete architecture, triggering subsequent investigations. Based on this analysis, we revised our original four-layered architectural style, to a style consisting of only three layers. Our main change in this regard was merging the Engine Layer and Common Layer into one layer called the Middle Layer which supported the explanation of many divergences we were seeing between non-adjacent layers. Additionally, we obtained a better understanding of how the OSystem API works in supporting the interactions between user events and the ScummVM software. We explored the inner architecture of the SCI Engine subsystem and how this subsystem is optimized for 2D point-and-click games. Throughout this phase, our appreciation for the collaboration required in building and documenting open-source software such as ScummVM, and tools such as SciTools Understand software used to understand such systems was a truly rewarding experience.

TERM DEFINITIONS

API	Stands for "Application Programming Interface". This is a collection of functions/interfaces/etc that are used in order to facilitate communication between two distinct programs or software components.
OS	Stands for "Operating System", the foundation-level program on a machine managing the resources needed to run higher-level programs. The term "platform-specific" refers an implementation unique to a certain operating system (i.e. "ScummVM for Windows 10")
GUI	Stands for "Graphic User Interface". A directory inside the User Interface Manager component and provides a lot of helping functions that show messages on the screen like printing error messages.
Layered Architecture	An architectural style consisting of numerous stacked layers from top to bottom, in which each layer only has direct communication with those immediately above or below them.
Divergence	After mapping dependencies from real codes, divergences are the dependencies that exist in the codes but not in the conceptual architecture.

REFERENCES

Interesting Engineering Editors. "How Do Game Engines Work?". *Interesting Engineering*. 2 November 2016. https://interestingengineering.com/innovation/how-game-engines-work.

Kang, Shih-Chung & Juang, Jhih-Ren & Hung, W. "Using Game Engine for Physics-Based Simulation". *Journal of Information Technology in Construction*. 2011. 16. 3-22.

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

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

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

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