CISC 322/326 Assignment 2
Concrete architecture report

Nov. 18, 2024

Group name: '; DROP TABLE groups; –

Xavier Awadalla - 21xpa@queensu.ca

Christopher Gil - 21cagi@queensu.ca

Aaron Rivest - 21alr18@queensu.ca

Craig Tylman - 21crt6@queensu.ca

Sam Tylman - 21sdt6@queensu.ca

Felix Xing - 21fx2@queensu.ca

Document goals

The goals for this document are as follows:

- Recap ScummVM's overall conceptual architecture as discussed in A1 deliverable.
- Show ScummVM's top level concrete architecture.
- Highlight and explain the divergences for ScummVM's top level architecture.
- Briefly explain the SCI subcomponent
- Create and explain SCI's internal conceptual architecture.
- Show SCI's internal concrete architecture
- Highlight and explain the divergences for SCI's internal architecture.
- Demonstrate the functionality of SCI and its interactions with the rest of the system through 2 sequence diagrams.

1.0 Introduction + Abstract

1.1 Introduction to ScummVM's top level Architecture

From our analysis, the overall architecture of ScummVM follows a Layered style. It is organized in roughly 3 different such layers: Upper, OS (Operating System) Abstraction and OS Specific.

For the most part, users will only interact with the GUI (Graphical user interface) component of ScummVM, which in fact forms the Upper Layer in combination with the Audio and Engine components.

We have also found that unlike a strictly layered software, ScummVM's dependencies between its components are rather chaotic, so the hierarchy of the layers has been difficult to fully organize. For example, its Audio component is part of the Upper layer, but it was implemented using the behavior of a component within the OS Abstraction layer.

The OS Abstraction layer itself hides details of different APIs (Application Programming interfaces) of the operating system from layers above it. The heart of this layer are the OS Abstraction (Common) and the Backend component. OS abstraction encapsulates details like threading, timers, file operations and so on. The lowest OS Specific layer contains code like entry points and ASM (Assembly) to ensure ScummVM can run on different platforms.

We have also discovered a component within the OS Abstraction layer through our analysis of the concrete architecture. This Backend component manages libraries for tasks including networking, audio mixing, keymapping, saves and video encoding.

The introduction of a new component into our conceptual architecture creates brand new unexpected dependencies, as this new component is a dependency for every other component, save for the Engines component.

Other existing dependencies were found to diverge from our conceptual architecture. The Audio component has a mutual dependency with the GUI component for easy audio

modifications in the GUI's settings menu and for audio to play when an error message occurs (rendered by the GUI). The audio component contains an old dependency on the graphics component, due to an older method of emulating the sound of an old synthesizer, which has since changed - rendering the dependency completely unnecessary.

Before we dive into the details, we can already be confident in our conclusion that the concrete architecture has maintained the layered style we proposed in our conceptual architecture.

1.2 Introduction SCI internal architecture

Inside of ScummVM's Engines component, there are several sub components, each representing a different game engine that ScummVM has support for. One of these engines is the SCI (Sierra Creative Interpreter) engine.

Introduction to SCI's conceptual architecture:

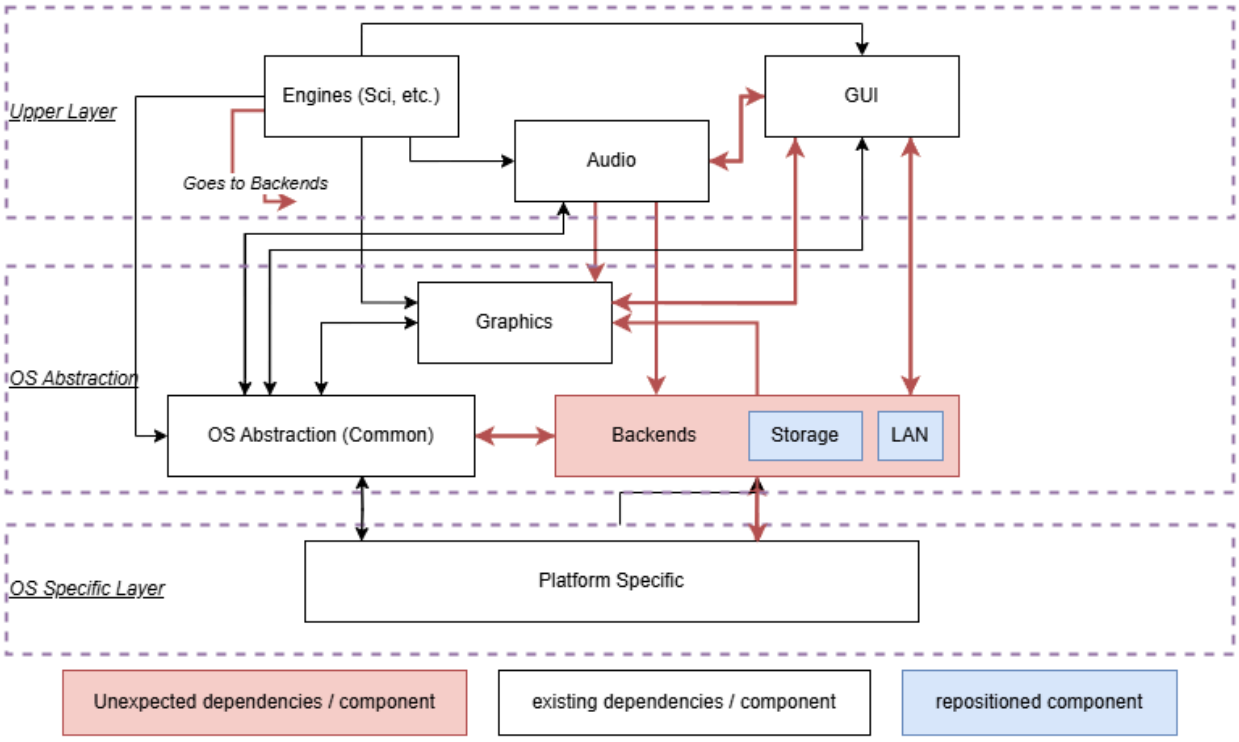
As a game engine for point and click adventure games, SCI should contain the following components:

- **Graphics,** split into 2 components: A live graphics component to deal with in game graphics that can depend on player actions, and a video component to decode and play pre recorded cutscenes.
- **Audio handler** for playing all sounds including music, sound effects and more.
- **Control parser** to take player inputs and turn them into actions within the game.
- **Detection** for ScummVM's top architecture to tell SCI which of its games is being played.
- **Engine** to deal with all of the features and functionalities within the game.
- **Resource manager** to organize, prepare and give access to game assets.

The expectation is that these components dependencies will somewhat resemble a repository architecture style (even though there is not really a repository) with the engine component in the middle having a dependency with all of the other components. Meanwhile, the other components will not need dependencies with each other, with the exception of the resource component which should have additional dependencies with all components that relate to game assets (e.g. audio and graphics components).

In reality, many components unexpectedly depend on one another, mainly to avoid extra complications when trying to implement certain use cases. For example, the graphics component depends on audio in order to modify audio playback when rendering. However, there are also a small number of cases where a divergence is made for the sake of one or two oddly specific contexts, all of which could have been easily avoided.

2.0 Overall system architecture:



The architecture of ScummVM follows a layered structure, with distinct top-level subsystems that facilitate clear interaction between user-facing and platform-specific components.

Engines

- **Usage**: The Engines component contains the core game logic and scripts for each supported game engine within ScummVM (e.g., SCI for Sierra games). Each engine interprets game-specific instructions, processes events, and handles the logic required for gameplay. This component is the foundation for running different games and is responsible for interpreting the original game code to produce the intended gameplay experience within ScummVM.

- **Interactions**: It communicates with the *Audio* component for sound playback and the *GUI* to display interface elements. It also communicates with the *Graphics* component for rendering game elements specific to each game's engine. It also communicates back and forth with the GUI component for settings changes. This component also interacts with OS Abstraction for game saves and internet connectivity. Engines essentially serve as the heart of game execution, feeding data to other components to create an interactive experience for the user.

Audio

- **Usage**: The Audio component manages all audio-related functionality within ScummVM. This includes playing sound effects, background music, and handling volume control. It translates audio commands from Engines into audible outputs, allowing for immersive sound experiences in games.

- **Interactions**: Audio communicates with *Engines* for specific game-related sounds and with *GUI* for synchronizing sound with error messages and for modification in the Settings menu. Audio also communicates with Graphics in one file for the MT-32 Roland synthesizer, leading to a rather strange divergence. The Audio component relies on the Backend for low-level platform-specific functionality, for tasks such as managing audio drivers and utilizing libraries to compile sound. Additionally, it connects with *OS Abstraction* for access to platform-specific audio resources, ensuring compatibility across different systems.

GUI (Graphical User Interface)

- **Usage**: The GUI provides the user interface layer, allowing players to interact with ScummVM, select games and configure settings. It is responsible for rendering menus, icons, text, and other graphical elements required for navigation and interaction.

- **Interactions**: GUI communicates with the *Engines* for displaying game-specific graphics and interactions, with *Audio* for coordinating sound effects with UI events, and with *Graphics* to render the GUI itself and management of the Graphics in the settings. GUI also interacts with *Backend* for low-level display functionality, which may vary depending on the underlying platform. Additionally, GUI interacts with OS Abstraction for support for interactions that require platform specific libraries, such as file access (saving/loading) or GUI elements (Mac OS menu bar).

Graphics

- **Usage**: The Graphics component handles rendering of all visual elements within ScummVM. It translates graphical commands into platform-compatible renderings, ensuring visuals are displayed correctly across different devices and operating systems.

- **Interactions**: Graphics connects with *GUI* and *Engines* to render both user interface fonts and game visuals. It relies on the *Backend* for lower-level graphics operations, such as drawing to the screen and handling platform-specific rendering requirements. Additionally, Graphics receives input from *Audio* for a specific device driver. Graphics also communicates with the platform specific layer for direct hardware access and access to other rendering libraries.

OS Abstraction (Common)

- **Usage**: This component provides an abstraction layer that standardizes access to OS-specific resources, isolating the upper components from the platform-specific code.

- **Interactions**: OS Abstraction interfaces with *Audio*, *Graphics*, and *Backend*, offering standardized access to resources. It ensures that ScummVM's components operate seamlessly across various platforms by handling platform-specific details. OS Abstraction communicates with the platform specific component for access to low-level OS features.

Backend

- **Usage**: The Backend component provides ScummVM with libraries, most importantly SDL. These libraries include managing video encoding, text to speech, and other tasks. Backend ensures that ScummVM can render sound and visuals in a standardized fashion for every game type and manages components such as *Storage* (for file access) and *LAN* (for network connectivity), allowing ScummVM to use these resources in a cross-platform way.

- **Interactions**: Backend communicates with *Graphics* to assist with platform-specific rendering tasks and may interface with *GUI* for display-related functions. It also connects with the *OS Abstraction* to provide platform specific codes. Backend's integration with the platform-specific layer allows ScummVM to leverage unique system capabilities while maintaining consistency across devices.

Platform Specific Layer

- **Usage**: This layer contains platform dependent codes to support the execution of ScummVM, like entry points, ASM, and abstractions for non-standard operating systems like Linux.

- **Interactions**: The Platform Specific Layer interacts with *Backend* and *OS Abstraction*, which act as intermediaries, ensuring that the upper-layer components can access platform-specific resources in a standardized way. This layer's services are accessed indirectly by ScummVM's components through Backend and OS Abstraction to maintain cross-platform compatibility.

Comparison between Backends and OS Abstraction

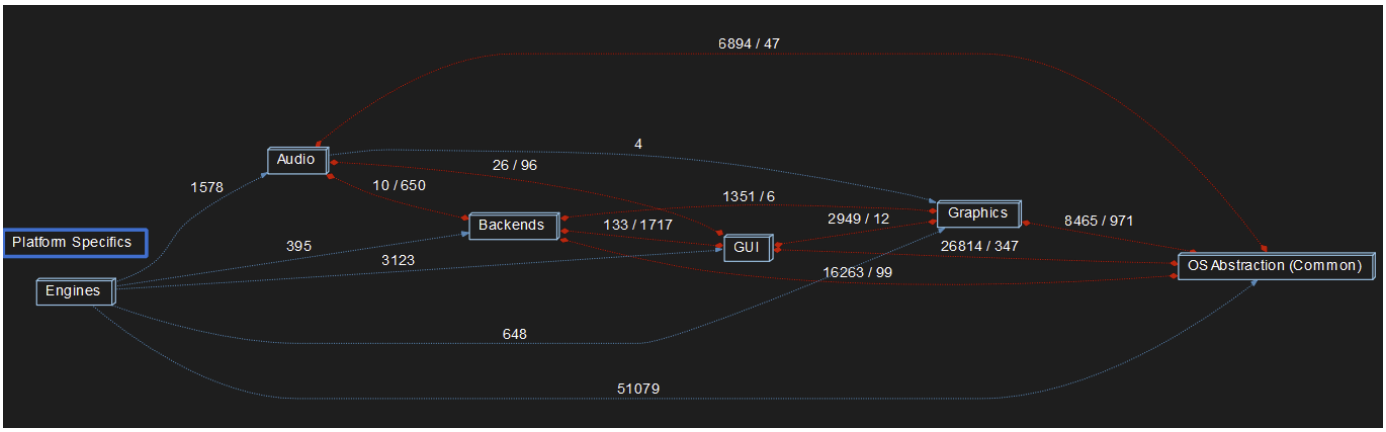
The Backends and OS Abstraction components are quite similar to each other. However, they do serve different purposes. The introduction of the new Backends component is to decouple the already existing Backend Layer / OS Abstraction component into 2 parts.

One part that specifically deals with the operating systems itself (OS Abstraction) and the other part contains Middlewares (external libraries or codes that focus on very specific features like codecs for media).

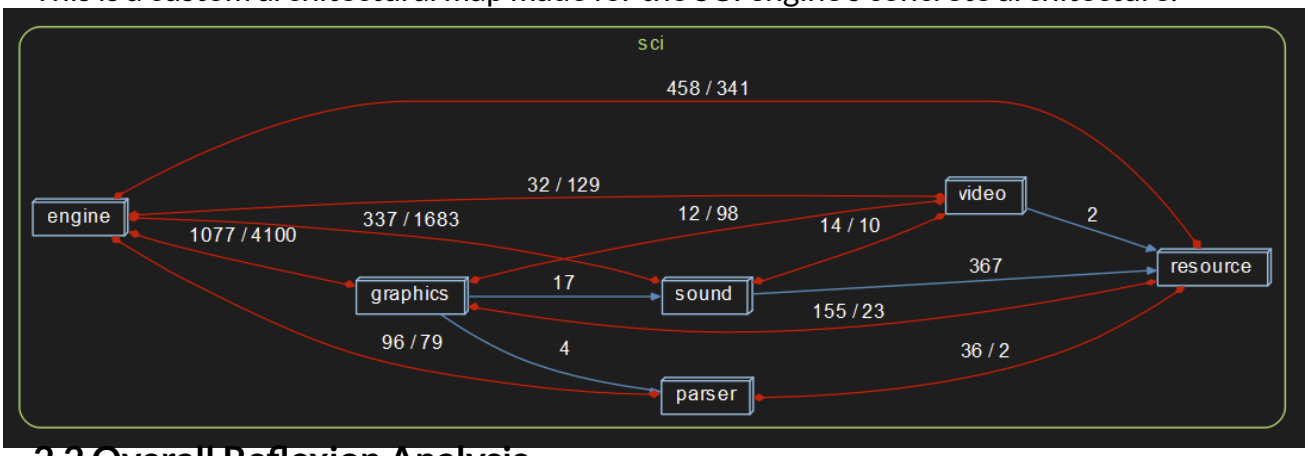
2.1 Overall Architecture Derivation Process

ScummVM & the SCI engine's (2nd level subsystem) concrete architecture were derived using the Understand tool. By making custom architectures in the Architecture Browser, we could create 'nodes' that align with each subcomponent derived in our conceptual architecture. By adding folders into the nodes for the architecture aligned with its subcomponent, we can create a graph that accurately displays the concrete architecture of ScummVM. By comparing this with our initial conceptual architecture, we can define which interactions and subcomponents are unexpected, and which may be positioned differently from our original conceptual architecture.

This is a custom architecture map made for the ScummVM concrete architecture.



This is a custom architectural map made for the SCI engine's concrete architecture.



2.2 Overall Reflexion Analysis

The aforementioned overall system architecture diagram contains the overall concrete architecture for ScummVM, with divergences and other changes highlighted in pink and blue, respectively.

Although there are several discrepancies from the original conceptual architecture, they are done in good faith in order to improve the software's performance and streamline its development.

Divergence: Audio ⇔ GUI

In our original conceptual architecture, Audio and GUI did not depend on each other. This was done under the assumption that the audio engine was reserved for gameplay, meaning that the GUI did not need to directly link up to it. However, closer analysis of the ScummVM codebase reveals many cases where it is reasonable for both modules to depend on each other.

For example, ScummVM uses the GUI component to render error messages, so when something goes wrong with the audio engine it will directly display a modal via the GUI's MessageDialog class. The audio engine will also be used to play a sound effect for certain errors, if applicable.

The GUI's options menu (options.cpp) also directly links to the Audio component in order to easily get and set audio-related options (mainly MIDI-related settings) without needing anything in between them.

Divergence: Audio → Graphics

There is a single call from Audio to Graphics present in ScummVM's handling of the Roland MT-32 MIDI synthesizer. Four classes from the Graphics component are imported (font, fontman, surface, pixelformat), but they **do not appear to be used for anything.** This is likely a mistake from one of the developers. Git blame reveals that these dependencies were added 20 years ago, but since then the codebase has changed so they are no longer used in any remaining methods.

Divergence: Graphics → GUI

There are certain cases in which the graphics module calls the GUI, despite our initial assumption that this would only work the other way around. Most notably, the engine's vector renderer makes several calls to the GUI's theme engine in order to customize shading, overlays, text alignment, and other finer details.

Divergence: Backend → GUI

While the GUI frequently calls to the backend, there are also several GUI features and classes that are directly used by the backend. The GUI's modal and dialogue systems are once again used to display messages directly to the client, in the event of an error or other warning that the client must be aware of.

The software's event recorder system (which records and plays back gameplay to help test engines) is also defined under GUI, with the backend connecting to it for recording purposes.

Divergence: The New Backend Component

ScummVM's concrete backend ended up being much more 'central' than initially anticipated, with nearly all components connecting to it in some way. Many core functions, variables, and enumerations are stored here, to be used by other classes throughout the software. The backend component actually represents multiple different unique backends, and is more commonly referred to as "backends" in the software rather than a single one. Cloud storage, text to speech, midi output, networking, and much more are all handled here.

3.0 Sci Internal Architecture:

3.1 Conceptual View:

SCI (Sierra Creative Interpreter) is one of the major game engines that ScummVM has support for. Knowing this, the conceptual architecture should contain all of the essential components for a game engine that runs point and click adventure games:

Components:

Engine Component:

The engine component is the most crucial component for the game engine. The engine is responsible for all of the functionalities in the game. This includes camera movement scripts, collision scripts, mapping controls to actions in the game and all other features that the game engine might have. Additionally, the game engine acts as the central controller for all of the other components in the game engine and will therefore be somehow dependent on all the other major components in the game engine.

Resource Manager Component:

The resource manager component deals with managing all of the assets for a given game in SCI. This includes functions like loading, organizing and managing access to resources. This includes assets like audio files, sprites, shaders scripts etc. Additionally, this would be the component to hardcode fix any bugs within specific assets for SCI games.

Graphics Component:

The graphics component deals with drawing **real time** graphics to the screen. This includes sprites, shaders and any other visuals that are drawn to the screen during gameplay.

Video Component:

The Video component deals exclusively with pre-recorded videos. The video component would take video files from the SCI games and decode them so that they can then be played during specific parts of the game as cutscenes or introductions.

Sounds Component:

The sound component is responsible for playing all sounds including dialogue, music, sound effects, ambient noise and more. The sound component will have a decoder component to decode sound files. Additionally, it will contain a drivers component to actually play the sounds.

Control Parser Component:

The control parser component is the most simple major component in the SCI game engine. This component takes user input and turns it into actions in the game. Keep in mind that the control parser needs to interact with the 'common' component in ScummVM's top-level architecture to account for any key bind changes that might have been made in the main settings tab (something that is outside of SCI's scope).

Detection Component:

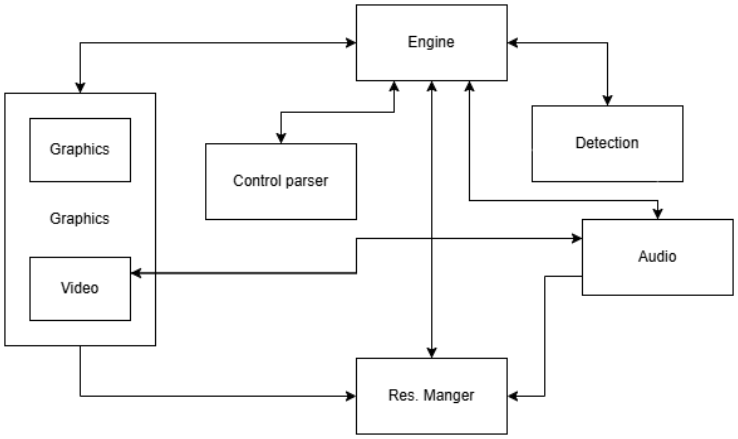
Detects which game is being played.

Excluded Components:

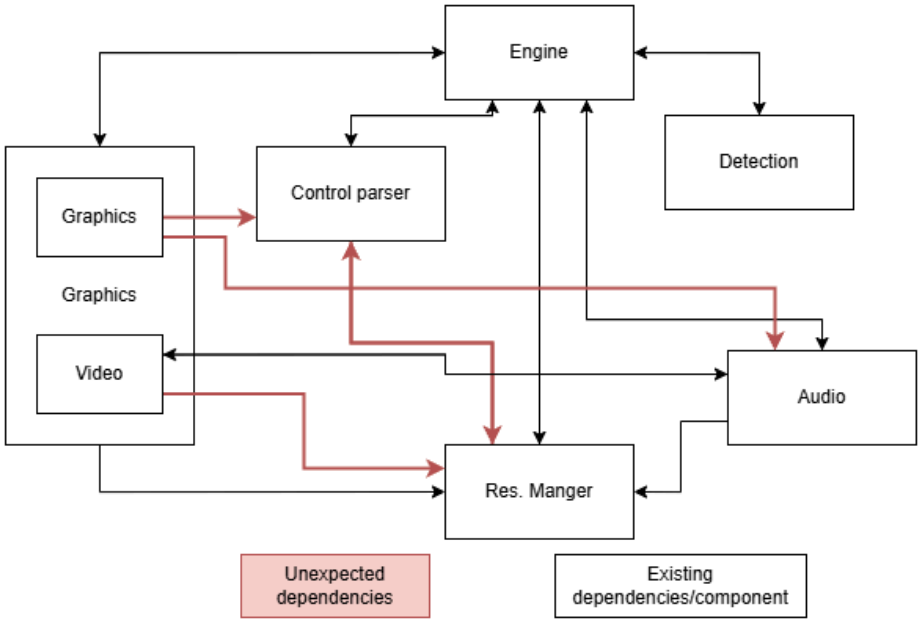
Most game engines would contain some sort of component inside the game engine to deal with simulated physics. However, SCI would not have this as it only deals with simple adventure games that do not really contain any simulated physics.

Component interactions:

Architectural style: In a way, ScummVM's architectural style should resemble a sort of repository style with the engine component in the middle having dependencies with all of the other components. Meanwhile, most of the other components don't really need to interact with each other very much. The main exception to this would be the resource manager component which should have dependencies with all of the components that relate to assets (e.g. sound, graphics, video, engine). Additionally, there should be a dependency between the audio and the video component as cutscenes are probably stored in a file type the includes audio built into it:



3.2 Sci Concrete Architecture and Reflexion Analysis



Divergence: Graphics → Control Parser

Similar to the above, the GfxControls16 class in the graphics component makes a single call to the control parser for vocabulary-related reasons - more specifically the checkAltInput method. It appears to be used for a text input field.

The Sci menu class also calls the parser's said() method, as well as its SAID_NO_MATCH constant - more vocabulary related features with little documentation.

Divergence: Video → Resource Manager

The "robot video" decoder uses a class called DecompressorLZS, defined in the resource manager. Its unpacking method is called once.

Divergence: Graphics → Audio

Sci's graphics component appears to have an unexpected dependency for the audio component, for a few fairly simple cases. For example, many methods in the GfxMenu class contain an optional argument to pause all sound playback, which is immediately handled by the audio engine.

Additionally, the graphics engine's video player connects to the audio mixer in order to correctly decode all sound.

Divergence: Control Parser ⇔ Resource Manager

Surprisingly, the resource manager is frequently called in the parser's vocabulary class - which is used to reference and look up words similar to a dictionary. All vocabulary data appears to be stored in the resource manager, while the control parser is responsible for looking words up.

There is also one single call from the resource manager to the control parser. When initializing specific games made by Sierra Entertainment, the resource manager accesses a single constant (VOCAB_RESOURCE_SELECTORS = 997) defined in the parser. This dependency likely could have been avoided with better structuring.

4.0 Sequence diagrams

4.1 Selecting and starting a game in ScummVM

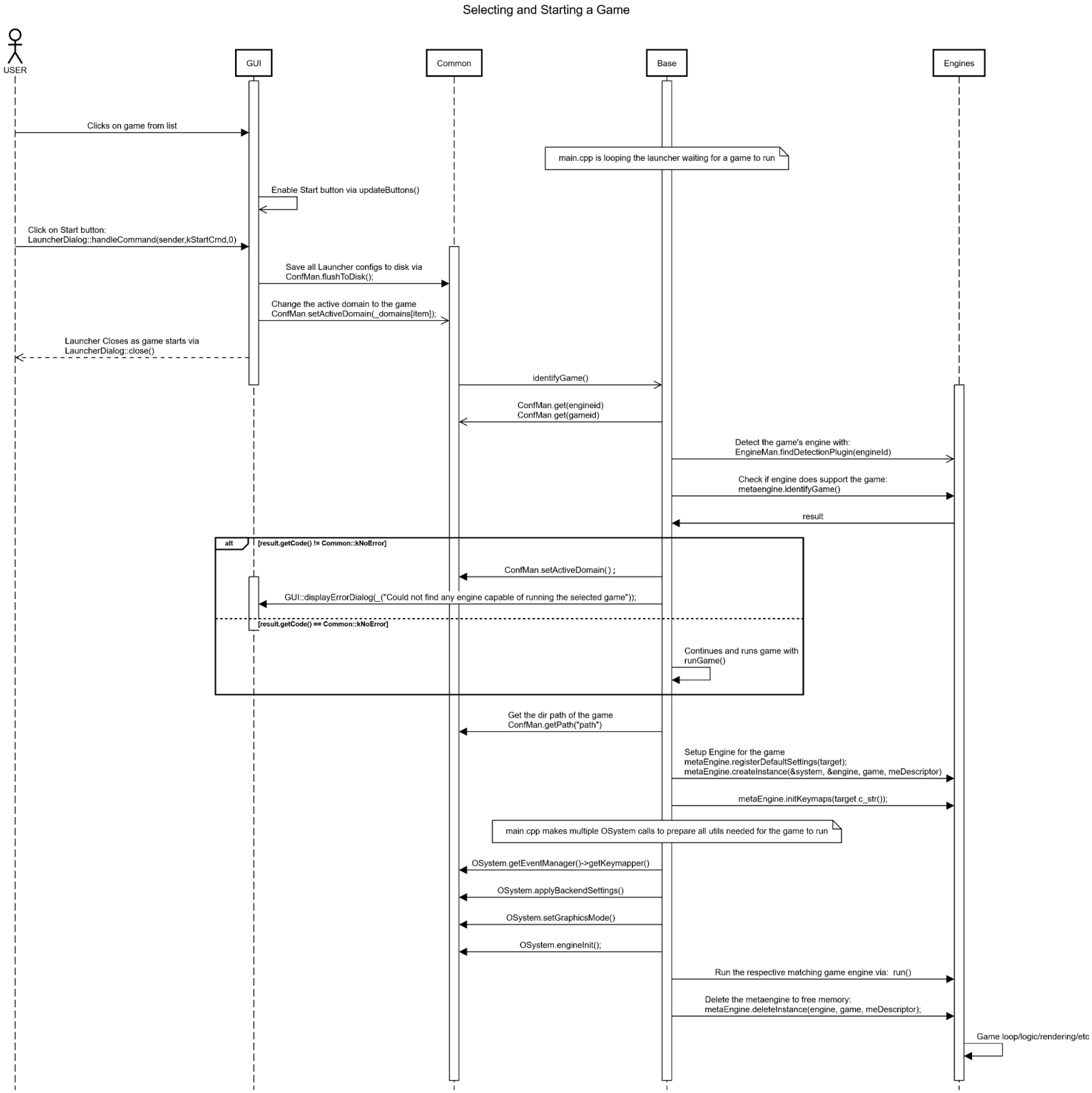
The diagram below (next page) illustrates the very general use case of "The user selects and starts a game".

The user clicks on the game from the game list widget, which enables the Start, Edit, and Remove buttons. The User then clicks on the Start button. The GUI saves all Launcher settings/configs to a save file and changes the active domain to be the game's ID.

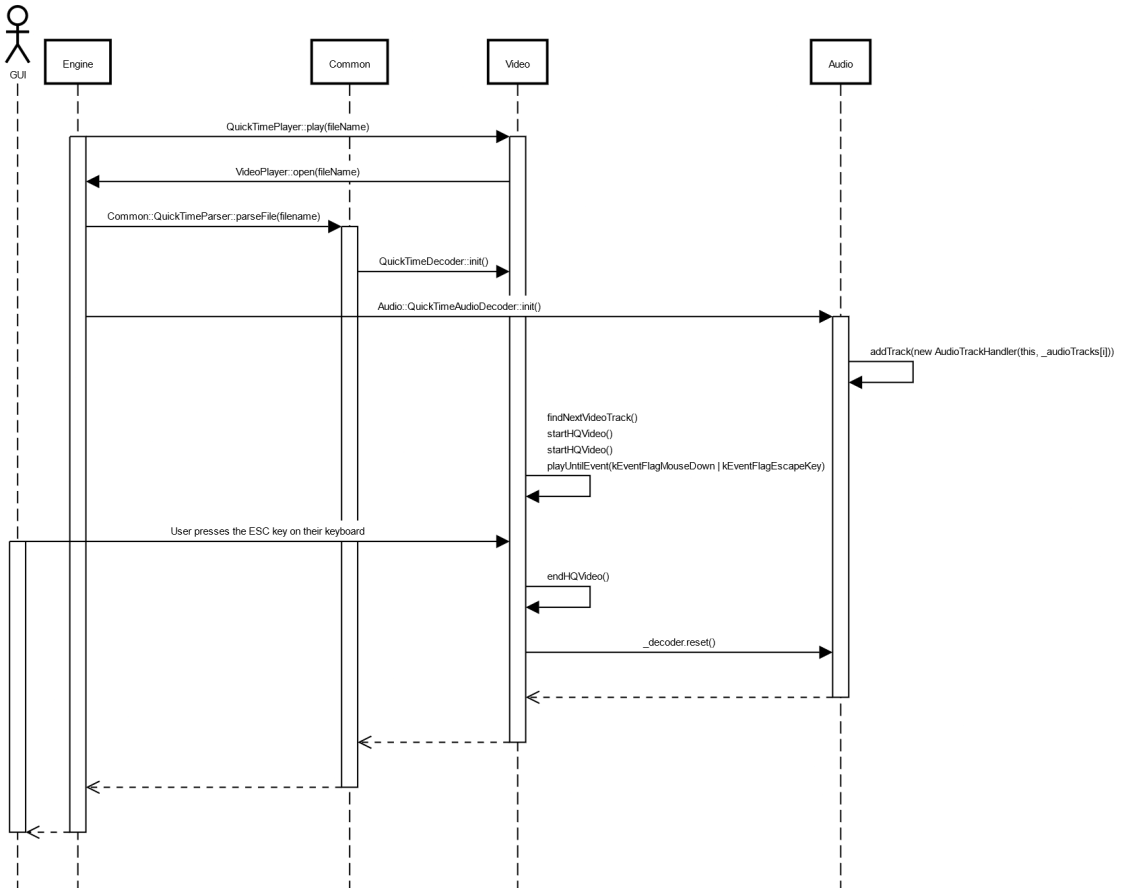
The Main method of Scummvm identifies the game and finds the game's details. It performs a second check by using EngineMan to detect if the engine exists, and then uses the engine's metaengine functions to determine if that engine supports the game. Based on the result of this check, the program either displays an error or makes a call to run the game.

After all the vital components and settings are set up, the old metaengine reference is cleared to free memory and the Engine takes over to handle all of its responsibilities.

A separate image file of this has been included in the ZIP for easier viewing if needed.



4.2 Running an introductory cutscene in ScummVM



The diagram above illustrates a simpler use case: “The user has already started a game which uses the Sci engine and the introductory cutscene plays using the Quicktime video player. The user then presses the 'escape' key on their keyboard (or clicks the left mouse button), and this exits the cutscene.” It is important to note that GUI and Common are not components within the Sci engine, where Engine, Video and Audio refer to Sci engine subcomponents.

First, the Engine sends a request to the Video component to play the video file. In order to do this, decoders for the quicktime player (both Video and Audio) need to be started and a call needs to be made to Common to parse the file given by the Video component. Afterward, Audio and Video tracks are gathered, the video is played and we await user input to end the cutscene (note that this is not the exclusive way of ending a cutscene and this could be triggered by, say, the player entering a new area in the game).

The escape key is then pressed, which signals the Video component to get rid of the Video and Audio decoder, which will return the screen back to the engine for further videos or gameplay.

5.0 Glossary + Naming Conventions

- **SCI** (Sierra Creative Interpreter): One of ScummVM's supported game engines.
- **GUI** (Graphical User Interface): A visual way for users to interact with the program.
- **MIDI**: Musical Instrument Digital Interface, a digital file type made for composing and editing electronic music and note sequences.
- **Text-to-speech**: Technology that reads aloud text in a computer
- **Cloud Storage**: Mode of computer data storage in which data is stored on a remote server as opposed to the user's local machine
- **ASM** (Assembly): Low-level programming language which directly corresponds with machine code instructions
- **API**: Application programming interface. A set of rules and protocols that allow different software applications to communicate and share information with each other
- **SDL**: Simple DirectMedia Layer. A cross-platform development library designed to help developing multimedia apps.

5.1 Conclusion + Lessons learned:

Overall, ScummVM's concrete architecture contains a lot more divergences from the conceptual architecture than initially expected. While some of them make sense (as they directly use a component's method in their intended manner), other divergences are simply caused by a single method call or variable access for an oddly specific situation. We learned that as ScummVM evolved over the past few decades, it underwent several internal changes that generally improved its architecture. However, this also means that there are a handful of dependencies and interactions that remain implemented, but are no longer necessary within the current codebase.

An improvement can be decoupling its components. Because of the long history of this project and less organized maintenance over the time, the structure of it is chaotic and there are not many documentations helping us understand it. To extract a clear structure from the existing "everyone depends on everyone else" situation; we introduced the Backends component and restructured what we have in A1 which benefits us from simply building architecture based on directory names.

5.2 Reference

"ScummVM", https://www.scummvm.org/ *Where almost all information came from*