CISC 322 Assignment 2

Concrete Architecture of ScummVM

Group Website

November 16th, 2024

Authors (Group #25):

Ahmad Tahir | 20354540 | 21amt25@queensu.ca

Elill Mathivannan | 20342676 | 21em73@queensu.ca

Elias Frigui | 20347215 | 21ef32@queensu.ca

Henry Xiu | 20338400 | 21hx14@queensu.ca

Amaan Javed | 20347533 | amaan.javed@queensu.ca

Momin Alvi | 20338306 | momin.alvi@queensu.ca

1.0 Abstract

This report presents a detailed analysis of the concrete architecture of ScummVM, a software system designed to enable the cross-platform execution of classic graphical adventure and role-playing games. Building on the conceptual architecture developed in Assignment 1, this study investigates ScummVM's layered design, comprising the Presentation Layer, Interface Layer, and Game Engine/VM Layer, each responsible for critical aspects such as user interaction, platform abstraction, and game-specific logic. The derivation of the concrete architecture involved mapping source code to these layers using tools like SciTools Understand, which provided insights into the relationships and dependencies between subsystems. This process uncovered both anticipated and unexpected architectural patterns and informed the system's refined structure.

We used the reflexion model to compare the concrete architecture to the conceptual model, revealing several consistencies, such as the accurate representation of subsystems and the preservation of modularity within the Common Utilities layer. However, significant divergences were also identified, including unintended couplings, over-reliance on specific components like the Graphics and Audio subsystems, and missing dependencies, such as the absence of planned connections between the Game Engine Layer and the Common Subsystem. These discrepancies indicate areas of architectural drift that compromise modularity and maintainability.

Additionally, this report includes sequence diagrams that illustrate key use cases, such as system initialization and sound playback, to demonstrate the structured interactions between subsystems. These diagrams highlight the orchestration role of the VMSubsystem and its collaboration with other subsystems to achieve critical functionalities. The findings underscore the need for architectural refinements to address identified inefficiencies and align the implementation more closely with the original design objectives. This report provides developers and architects with actionable recommendations to improve ScummVM's scalability, adaptability, and long-term maintainability.

2.0 Introduction and Overview

This report documents the concrete architecture of ScummVM, a software system designed to enable the execution of classic graphical adventure and role-playing games on modern platforms. ScummVM achieves this by reimplementing the original game engines, ensuring compatibility across various operating systems and hardware. The architecture reflects a layered design to maintain modularity, scalability, and cross-platform adaptability. Building on the conceptual architecture outlined in Assignment 1, this report provides a comprehensive analysis of the system's concrete architecture, its derivation, and the alignment between the conceptual and concrete implementations.

The purpose of this report is to explore the structural and functional design of ScummVM in detail. By deriving the concrete architecture from the conceptual model, the report evaluates the strengths and weaknesses of the system's implementation. The study aims to identify key consistencies, divergences, and absences between the two architectures, providing insights into areas where the design aligns with or deviates from the original architectural vision. Additionally, this report seeks to understand the architectural styles and patterns used in

ScummVM, highlighting their role in achieving the system’s goals of modularity and extensibility.

To derive the concrete architecture, the team employed SciTools Understand to analyze ScummVM’s source code and extract dependency graphs. This tool facilitated the visualization of relationships between subsystems, revealing both expected and unexpected architectural connections. The findings were organized into three primary layers consistent with the conceptual architecture: the Presentation Layer, the Interface Layer, and the Game Engine/VM Layer. Each layer was further divided into subsystems, including the Graphics, Audio, Input, and Common Utilities subsystems, which were mapped to the source code structure.

The report is organized into several sections, each addressing critical aspects of the system's architecture. The derivation process section outlines the methodology used to transition from the conceptual model to the concrete architecture, describing the steps taken to map source code directories to architectural subsystems. The conceptual architecture section revisits the system’s high-level design, emphasizing the use of architectural styles such as Layered Architecture, Interpreter Pattern, and Publish-Subscribe Model. The concrete architecture section provides a detailed analysis of the concrete architecture, highlighting the organization and interactions of subsystems. The reflexion analysis compares the conceptual and concrete architectures, identifying key consistencies, discrepancies, and their implications for system maintainability. Finally, the report presents use cases supported by sequence diagrams to demonstrate critical interactions between subsystems, such as game initialization and sound playback.

The reflexion analysis revealed significant findings. While the core modular structure was preserved, divergences were noted, including unintended dependencies and excessive reliance on specific subsystems. For instance, the Game Engine Layer exhibited dependencies on the Audio and Input subsystems that were not part of the conceptual design, reducing modularity and increasing complexity. The Graphics Subsystem also displayed a higher-than-expected number of interactions with the VM Subsystem, suggesting potential performance bottlenecks. Additionally, missing dependencies, such as the planned connection between the Game Engine Layer and the Common Utilities subsystem, highlighted areas where architectural drift occurred during implementation.

These findings underscore the importance of continuous architectural evaluation and refinement. By addressing the identified divergences and inefficiencies, ScummVM can better align its implementation with the original design goals. Key recommendations include reducing subsystem coupling, restoring missing dependencies, and optimizing subsystem interactions to enhance performance and maintainability.

In conclusion, this report provides a thorough examination of ScummVM’s concrete architecture, offering insights into its strengths, limitations, and opportunities for improvement. By documenting the derivation process, analyzing subsystem interactions, and comparing the conceptual and concrete architectures, the report aims to serve as a valuable resource for developers and architects working to maintain and enhance ScummVM’s modular, scalable, and cross-platform capabilities.

3.0 Derivation Process

The derivation process for ScummVM's concrete architecture began with a thorough review of the conceptual architecture developed in Assignment 1. This initial step ensured a solid foundation for understanding the system's key components and their relationships. The conceptual architecture focused on three primary architectural styles: Layered, Interpreter, and Publish-Subscribe. These styles provided a high-level structure for ScummVM, dividing the system into distinct layers and establishing the flow of control and data between them.

To transition from the conceptual to the concrete architecture, we used the dependency file provided on OnQ. This file was analyzed using SciTools Understand, a tool designed to visualize code dependencies and subsystem interactions. By importing the file into Understand, we generated a detailed dependency graph that revealed the relationships between top-level modules. This step helped us identify both expected and unexpected connections between components, as well as any missing dependencies from our conceptual architecture.

Next, the system was organized into three primary layers to align with the conceptual architecture: the Presentation Layer, the Interface Layer, and the Game Engine/VM Layer. The Presentation Layer encompasses user-facing functionalities such as the graphical user interface (GUI), input handling, rendering, and audio. The Interface Layer acts as a bridge between platform-specific implementations and game engines, ensuring compatibility and communication between various components. The Game Engine/VM Layer is responsible for managing game-specific logic and reimplementing game engines to work with modern systems. Each of these layers was further refined by grouping related subsystems, such as the Rendering Subsystem, Audio Subsystem, and Input Handling Subsystem, under their respective layers.

Throughout this process, several iterations were conducted to ensure the accuracy of the concrete architecture. Any discrepancies between the conceptual and concrete architectures were noted for further analysis. For example, new dependencies were discovered in the concrete architecture, such as additional interactions between subsystems that were not anticipated in the conceptual model. These findings highlight the importance of using tools like Understand to visualize and verify architectural decisions.

By following this methodical approach, we successfully derived ScummVM’s concrete architecture, which provides a detailed and accurate representation of the system's structure and its subsystems' interactions.

4.0 Conceptual Architecture

ScummVM's architecture enables cross-platform execution of classic graphical adventure and role-playing games by replacing original game executables with its reimplementations. This approach ensures compatibility with modern hardware and operating systems, reviving legacy games that would otherwise be inaccessible. Its modular design emphasizes scalability, flexibility, and maintainability, allowing seamless integration of new features, platforms, and game engines while preserving the integrity of the original gaming experience.

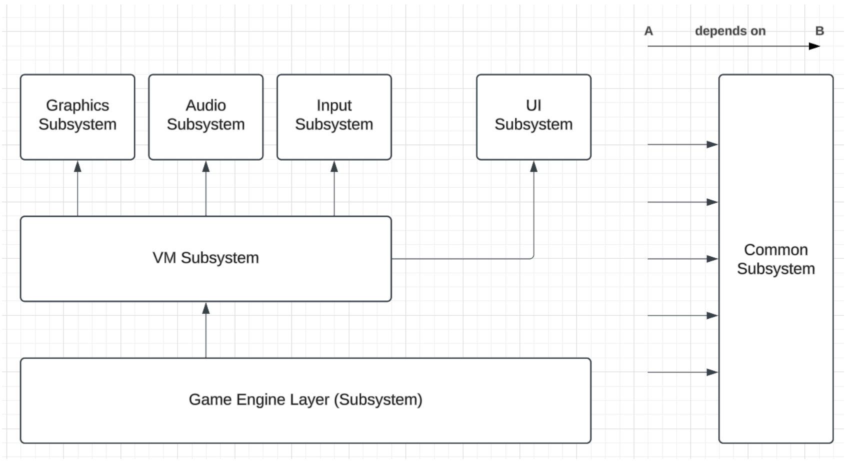

Figure 1: ScummVM Conceptual Architecture and Subsystem Interactions

The architecture is structured into three primary layers. The Presentation Layer manages user-facing functionalities such as graphical rendering, audio playback, input processing, and user interface interactions. It includes subsystems like Graphics, Audio, Input, and UI, ensuring smooth communication between the user and the system. The Interface Layer acts as an abstraction layer, bridging platform-specific implementations and game engines. By providing standardized interfaces for rendering, audio, input, and virtual machine operations, it simplifies platform-specific complexities and ensures consistent functionality across diverse environments. Finally, the Game Engine/VM Layer handles core game logic, supporting a variety of game engines, such as SCUMM and SCI, tailored to run on ScummVM's framework.

This layered architecture ensures modularity, adaptability, and ease of maintenance, making it straightforward to add features, support new games, and remain compatible with evolving technologies. Together, these layers deliver reliable gameplay experiences and fulfill ScummVM's mission to preserve and make classic games accessible to modern audiences.

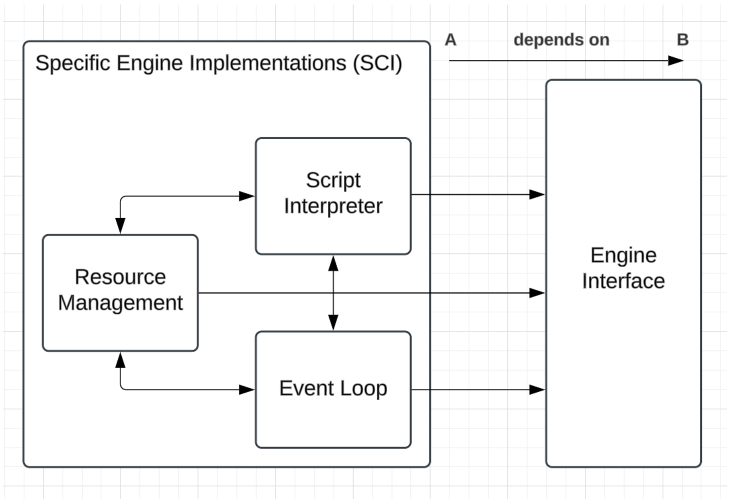

Figure 2: Detailed Architecture of the Game Engine Subsystem in ScummVM

This diagram highlights the SCI engine within the Game Engine Subsystem, focusing on key components like the Script Interpreter, Resource Management, and Event Loop. These elements work together to execute game scripts, manage assets, and handle in-game events. The Engine Interface connects these components to ScummVM's broader architecture, ensuring compatibility with the Presentation and Interface Layers.

The Game Engine/VM Layer processes core game logic and includes the Virtual Machine (VM) Subsystem, which interprets game-specific scripts and coordinates subsystems such as graphics, audio, and input. It supports reimplementations of engines like SCUMM and SCI, leveraging standardized interfaces for seamless communication with the Interface Layer.

ScummVM also uses a Common Subsystem for shared resources, including an event manager for handling game events, a file system for managing saves, and a memory manager for efficient resource allocation. These utilities ensure consistent, efficient operations across all games.

The SCI engine’s modular architecture isolates game-specific implementations from ScummVM’s core, simplifying the integration of new engines. Its design employs architectural styles like Layered Architecture for separation of concerns, the Interpreter Pattern for processing game scripts, and the Publish-Subscribe Model for real-time event handling.

In summary, ScummVM's architecture prioritizes modularity, scalability, and cross-platform compatibility. This approach supports a wide range of classic games and ensures adaptability to modern platforms.

5.0 Concrete Architecture

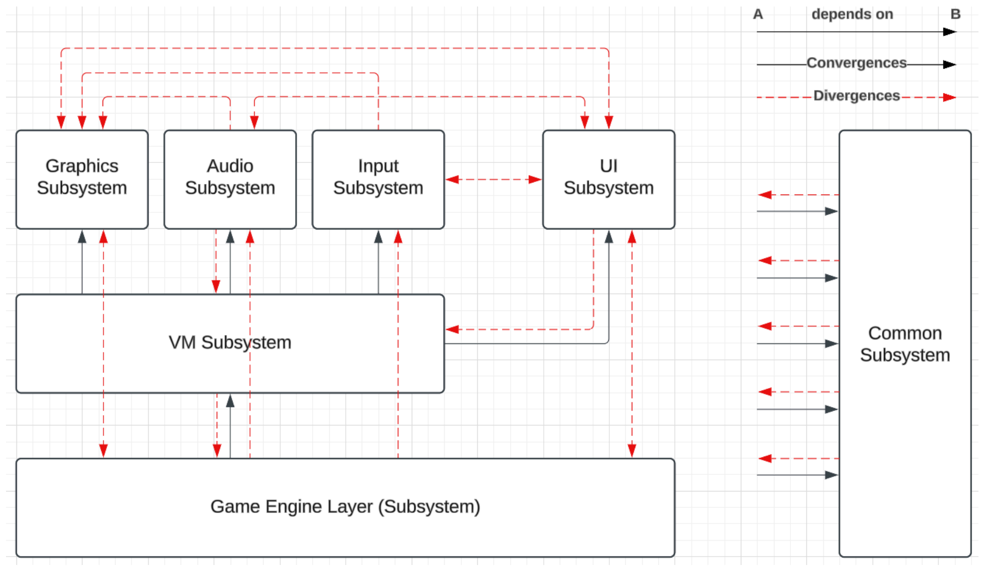

Figure 3: ScummVM Concrete Architecture and Subsystem Interactions

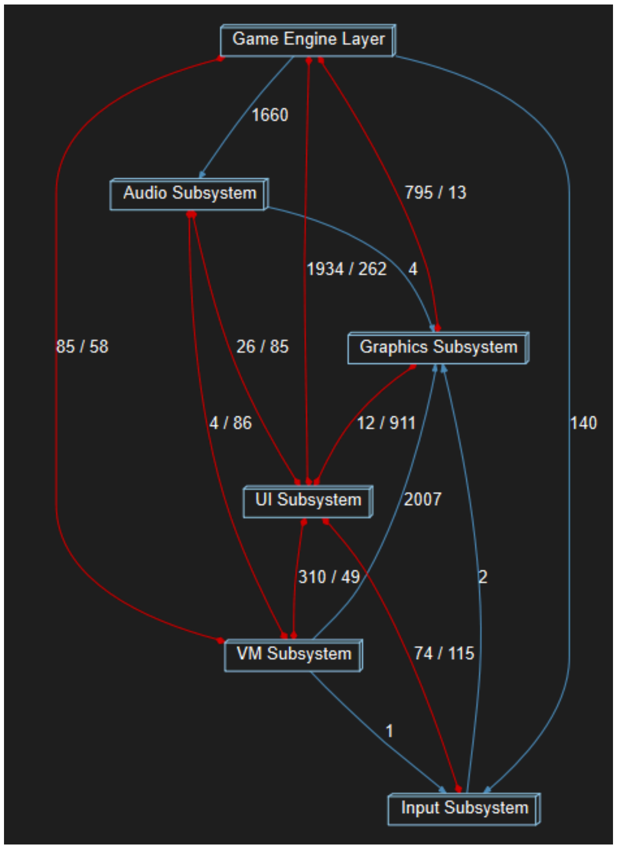

Figure 4: ScummVM Concrete Architecture and Subsystem Interactions (from Understand)

Derivation

- Many of our subsystems such as Audio, UI, and Graphics, and GEL could be immediately mapped to one of the directories in the source code.
- Within the backends/ directory, most of the files and folders could be mapped to the VM subsystem, but the specific directories on input handling: such as backends/keymapper/ and backends/vkeybd/ were mapped to the Input subsystem, with the remaining files in backends/ mapped to the VM subsystem
- Finally, the rest of the directories such as base/, common/, dists/, icons/, image/, math/, and video/ could be mapped to the Common Utilities subsystem.

5.1 Concrete Architecture of the Game Engine Layer

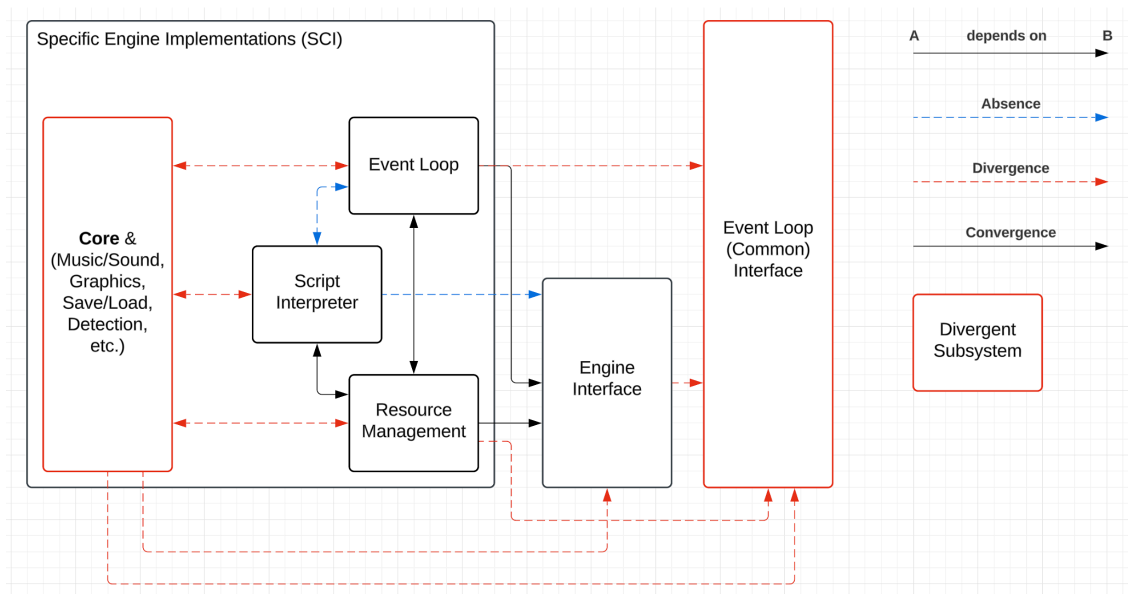
Specific Engine Implementations (SCI)
Core & (Music/Sound, Graphics, Save/Load, Detection, etc.)
Event Loop
Script Interpreter
Resource Management
Engine Interface
Event Loop (Common) Interface
A
depends on
B
Absence
Divergence
Convergence
Divergent Subsystem

Figure 5: ScummVM Game Engine Concrete Architecture and Subsystem Interactions

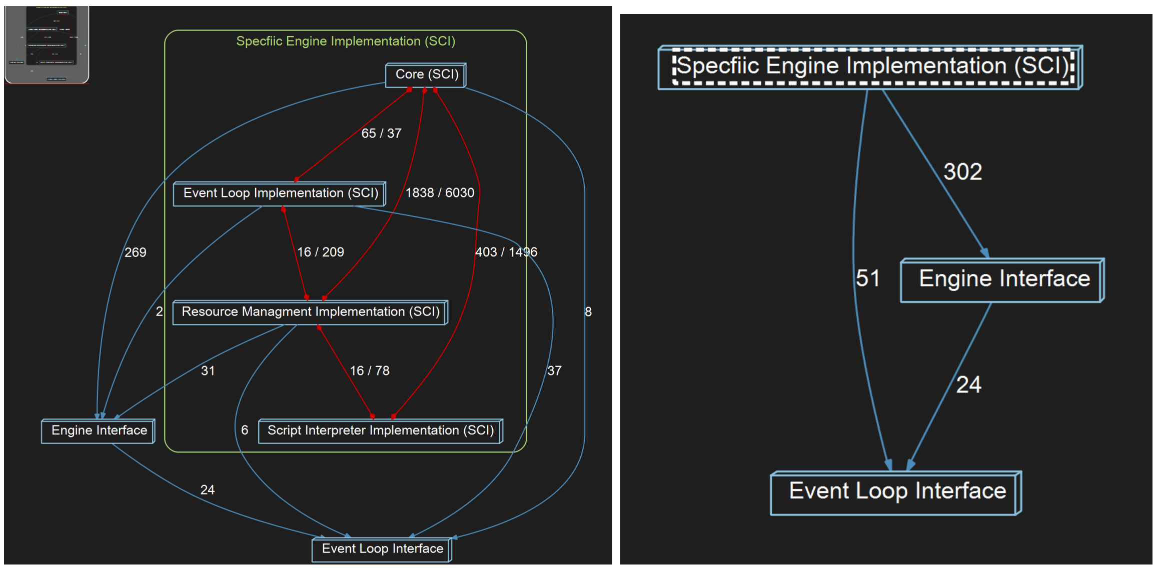
Specfiic Engine Implementation (SCI)
Core (SCI)
65 / 37
Event Loop Implementation (SCI)
1838 / 6030
269
16 / 209
403 / 1496
2
Resource Managment Implementation (SCI)
8
31
16 / 78
37
Engine Interface
6
Script Interpreter Implementation (SCI)
24
Event Loop Interface
Specfiic Engine Implementation (SCI)
302
51
Engine Interface
24
Event Loop Interface

Figure 6: ScummVM Game Engine Concrete Architecture and Interactions (from Understand)

Derivation

- We began to derive the concrete architecture of the Game Engine Layer subsystem by simplifying the problem down and focusing on a specific engine implementation (SCI)
- Looking at SCI engine we found 4 major subcomponents to it: Event Loop, Script Interpreter, Resource Management, and Core:

- In the Event Loop subsystem are the implementations of the newly added Common Event Loop Interface; these implementations (event.cpp and event.h) were in the root directory of engines/sci/
- In the Script Interpreter subsystem are the engines/sci/parser/ directory which contained grammar and vocabulary rules for the SCI specific scripts, as well as several other script specific C++ code files and header files
- In the Resource Management subsystem are the engines/sci/graphics/, engines/sci/resource/, engines/sci/sound/, and engines/sci/video/ directories, which contains the logic for handling various different kinds of resources used by the SCI engine.
- Finally, in the Core subsystem which contains everything else in SCI, is the actual implementation of the engine interface found in the engines/ directory

5.2 Reflexion Analysis for High-Level Architecture

The reflexion analysis for the high-level architecture highlights key convergences, divergences, and absences between the conceptual and concrete architectures. Convergences are evident in the consistent inclusion of core subsystems, such as the Game Engine Layer, VM Subsystem, Graphics Subsystem, Audio Subsystem, Input Subsystem, UI Subsystem, and the Common Subsystem. Additionally, dependencies such as the one between the Graphics Subsystem and the VM Subsystem align with the conceptual design, demonstrating that the core rendering logic was correctly delegated to the VM layer. Furthermore, the connections from the Graphics, Audio, Input, and UI Subsystems to the Common Subsystem are maintained, reflecting proper modularity for shared functionality, as initially planned.

However, the reflexion analysis reveals several divergences that deviate from the conceptual model, indicating significant architectural drift during implementation. A notable example is the Game Engine Layer, which exhibits unintended dependencies on multiple subsystems. In the concrete diagram, the Game Engine Layer → Input Subsystem dependency is present but was not part of the original conceptual design. This unexpected coupling suggests that input-related logic has been integrated into the game engine, which reduces modularity. Additionally, the Game Engine Layer makes 1660 calls to the Audio Subsystem, a dependency that is not explicitly modeled in the conceptual diagram. This over-reliance on the Audio Subsystem may reflect inefficiencies or ad hoc additions during development.

The UI Subsystem also shows significant divergences, as it directly interacts with all other subsystems in the concrete architecture. This is particularly evident in its dependency on the Graphics Subsystem, which is absent from the conceptual model but results in 2007 calls in the concrete implementation. This divergence indicates a tight coupling between rendering and UI-related tasks, which was not anticipated in the original design. Additionally, the concrete model shows that the UI Subsystem interacts with the Audio Subsystem, further complicating its role and increasing interdependence.

The Graphics Subsystem demonstrates another major divergence. While its dependency on the VM Subsystem is expected, the concrete architecture reveals an excessive number of interactions, with 1934 actual calls compared to 262 planned calls. This over-reliance on the VM Subsystem suggests that the Graphics Subsystem is offloading far more functionality than originally intended, potentially leading to performance bottlenecks. Furthermore, the Graphics Subsystem → UI Subsystem interaction, which was not modeled in the conceptual architecture, represents an unintended dependency that reduces the independence of the Graphics Subsystem.

In terms of absences, the most significant is the missing interaction between the Game Engine Layer and the Common Subsystem. The conceptual model planned for the Game Engine Layer to utilize shared functionalities provided by the Common Subsystem, but this connection is absent in the concrete implementation. Similarly, the Input Subsystem → Common Subsystem dependency is missing, indicating that input-related shared utilities may have been handled differently in the implementation, potentially reducing the overall modularity.

In conclusion, while the high-level architecture maintains the presence of key subsystems and some planned dependencies, numerous divergences and absences reduce its alignment with the conceptual design. The presence of unintended dependencies, excessive reliance on specific subsystems, and missing modular components such as the Common Subsystem highlight areas where the implementation has drifted from the conceptual architecture. These issues should be addressed to improve the maintainability, scalability, and adherence to the original architectural vision.

5.3 Reflexion Analysis for Chosen Second-Level Subsystem: Graphics Subsystem

The Specific Engine Implementation (SCI) subsystem was chosen for detailed reflexion analysis due to its critical role in handling core game logic and its significant deviations between the conceptual and concrete diagrams. This subsystem includes the Core, Event Loop Implementation, Script Interpreter Implementation, and Resource Management Implementation, each contributing to game processing through dependencies on the Engine Interface and other internal components.

The convergences in this subsystem align with the conceptual design's intended modularity. The Event Loop Implementation, for instance, depends on both the Script Interpreter and the Engine Interface in both models, reflecting its role in coordinating events with script logic and system-level engine operations. Similarly, the Resource Management Implementation and Script Interpreter maintain their planned dependencies on the Engine Interface, facilitating proper interaction with system-wide functionalities. These convergences indicate that the subsystem largely adheres to the high-level modular architecture's goals for clear separation of responsibilities.

However, there are several divergences that reveal implementation-specific adjustments or inefficiencies. Notably, the concrete diagram shows a significant over-reliance on the Core (SCI), particularly from the Event Loop Implementation, with 1838 actual calls compared to 6030 expected calls. This over-reliance suggests that the Event Loop is heavily interacting with the Core for tasks that might have been better distributed among other components, potentially reducing modularity and efficiency. Another unexpected dependency is observed between the Resource Management Implementation and the Script Interpreter Implementation, as reflected by the red arrows. This interaction was not planned in the conceptual architecture, suggesting a tighter coupling between script execution and resource allocation than originally envisioned.

The absences in the concrete architecture highlight areas where planned dependencies are missing or incomplete. For example, while the conceptual design includes dependencies from the Resource Management Implementation to the Core for centralized logic, this connection is either absent or underutilized in the concrete implementation. Additionally, the Engine Interface shows fewer interactions with the Core than planned, which could indicate a deviation in the intended architecture for handling centralized system operations.

Overall, while the SCI subsystem retains its key responsibilities and maintains alignment in several areas, the divergences, particularly the over-reliance on the Core and unexpected interactions between components, suggest inefficiencies and reduced modularity. Addressing these issues would enhance the alignment between the conceptual and concrete designs, ensuring better scalability and maintainability for the system.

6.0 Use Cases

This section presents two sequence diagrams illustrating key system use cases: game initialization and sound loading/playback. The first diagram details the sequential activation of subsystems during startup, including graphics, audio, and event handling. The second shows how sound resources are loaded into memory and played, highlighting coordination between audio components. These diagrams demonstrate the system's modular, structured design, ensuring reliability and efficient subsystem communication.

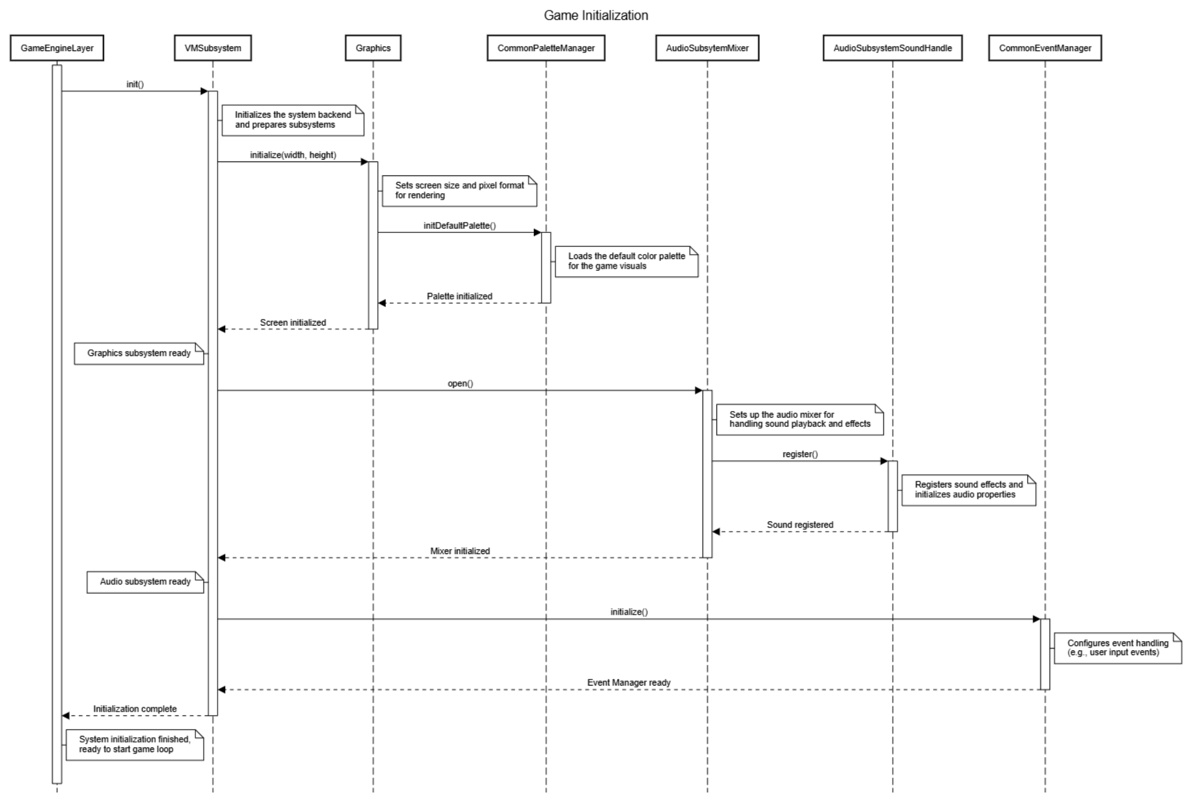
Game Initialization
GameEngineLayer
VMSubsystem
Graphics
CommonPaletteManager
AudioSubsytemMixer
AudioSubsystemSoundHandle
CommonEventManager
init()
Initializes the system backend and prepares subsystems
initialize(width, height)
Sets screen size and pixel format for rendering
initDefaultPalette()
Loads the default color palette for the game visuals
Palette initialized
Screen initialized
Graphics subsystem ready
open()
Sets up the audio mixer for handling sound playback and effects
register()
Registers sound effects and initializes audio properties
Sound registered
Mixer initialized
Audio subsystem ready
initialize()
Configures event handling (e.g., user input events)
Event Manager ready
Initialization complete
System initialization finished, ready to start game loop

Figure 7: Game Initialization Sequence Diagram

This sequence diagram illustrates the structured process of game initialization, beginning with the GameEngineLayer invoking the init() method on the VMSubsystem, which orchestrates the initialization of essential subsystems—graphics, audio, and event handling. The process starts with the initialization of the Graphics Subsystem, where the VMSubsystem calls initialize(width, height) on Graphics to set up the screen resolution and pixel format. Graphics then interacts with the CommonPaletteManager to load the default color palette via initDefaultPalette(). Once completed, Graphics confirms its readiness with a "Screen initialized" response.

Next, the Audio Subsystem is initialized. The VMSubsystem calls the open() method on the AudioSubsystemMixer, which prepares the audio mixer for handling sound playback and effects. The AudioSubsystemMixer registers sound properties using the register() method on the AudioSubsystemSoundHandle and confirms completion with a "Mixer initialized" response.

The final step involves setting up the Event Management System. The VMSubsystem invokes the initialize() method on the CommonEventManager, which configures mechanisms for handling user inputs and in-game events. Once ready, it sends an "Event Manager ready" response back to the VMSubsystem.

With all subsystems successfully initialized, the VMSubsystem signals completion to the GameEngineLayer with an "Initialization complete" message. At this point, the GameEngineLayer notes that initialization is finished, and the system is ready to start the game's main loop. This diagram emphasizes the VMSubsystem's critical role in coordinating subsystem initialization, ensuring dependencies are managed, and enabling smooth gameplay startup.

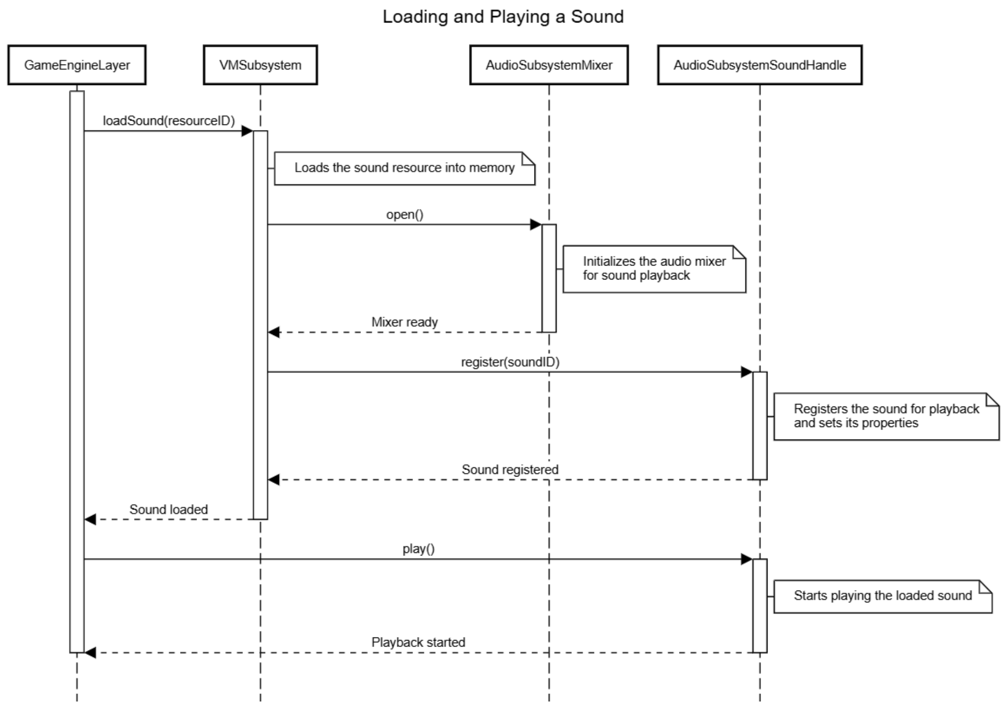
Loading and Playing a Sound
GameEngineLayer
VMSubsystem
AudioSubsystemMixer
AudioSubsystemSoundHandle
loadSound(resourceID)
Loads the sound resource into memory
open()
Initializes the audio mixer for sound playback
Mixer ready
register(soundID)
Registers the sound for playback and sets its properties
Sound registered
Sound loaded
play()
Starts playing the loaded sound
Playback started

Figure 8: Load and Play Sounds

This sequence diagram illustrates the process of loading and playing a sound in the game system. The interaction begins with the GameEngineLayer invoking the loadSound(resourceID) method on the VMSubsystem, which acts as the coordinator for audio operations. The VMSubsystem handles the request by ensuring the sound resource is loaded into memory and ready for playback. To enable sound playback, the VMSubsystem calls the open() method on the AudioSubsystemMixer, which initializes the audio mixer. Once the mixer is prepared, the AudioSubsystemMixer returns a "Mixer ready" acknowledgment, signaling that the audio subsystem is ready to process sound.

Next, the VMSubsystem proceeds to register the sound by invoking the register(soundID) method on the AudioSubsystemSoundHandle. The AudioSubsystemSoundHandle registers the sound for playback, setting the necessary properties such as volume and playback configuration. Upon successful registration, the AudioSubsystemSoundHandle sends a "Sound registered" acknowledgment to the VMSubsystem, which then notifies the GameEngineLayer with a "Sound loaded" message, indicating that the sound is fully loaded and ready for use.

Finally, the GameEngineLayer initiates sound playback by invoking the play() method on the AudioSubsystemSoundHandle. This method starts playing the loaded sound, and a "Playback started" confirmation is sent back to the GameEngineLayer. This sequence demonstrates a clear and systematic flow of interactions between components, where the VMSubsystem orchestrates the collaboration between the AudioSubsystemMixer and the AudioSubsystemSoundHandle. Each step ensures that the subsystems are ready and confirms successful task completion before proceeding to the next. This structured process ensures smooth sound loading and playback, minimizing the risk of errors or delays during execution.

7.0 Data Dictionary

- **Audio Subsystem:** Handles sound playback, including music, effects, and voiceovers.
- **Common Subsystem:** Provides shared utilities like event management, file handling, and memory allocation.
- **Core (SCI):** Implements the central logic for the SCI game engine, including script interpretation.
- **Game Engine Layer:** Processes core game logic and coordinates subsystems like input, graphics, and audio.
- **Graphics Subsystem:** Manages rendering of visuals, sprites, and UI elements.
- **Input Subsystem:** Handles input from devices (keyboard, mouse, controllers) and maps actions.
- **Virtual Machine (VM) Subsystem:** Orchestrates interactions between game logic and other subsystems.

8.0 Naming Conventions

Abbreviations:

- **VM**: Stands for *Virtual Machine*. Refers to the subsystem responsible for interpreting game logic and orchestrating interactions between layers.

- **SCI**: Refers to the *Script Creation Utility for Maniac Mansion* engine used in certain games, implemented as a subsystem in ScummVM.
- **GEL**: Abbreviation for *Game Engine Layer*, which processes core game logic and integrates subsystems.
- **UI**: Stands for *User Interface*, referring to components or subsystems related to user interactions.

Code Components:

- **Classes**: Use PascalCase (e.g., AudioMixer, GraphicsRenderer) to indicate distinct objects or modules.
- **Methods**: Use camelCase (e.g., initializeGraphics(), loadSoundResource()) to describe actions or operations.
- **Constants**: Use ALL_CAPS with underscores (e.g., DEFAULT_VOLUME, MAX_SCREEN_WIDTH) for global or system-wide fixed values.

9.0 Conclusions

This report provided an in-depth analysis of ScummVM's concrete architecture, highlighting its strengths, limitations, and alignment with the conceptual model. Key findings indicate that while the system successfully maintains modularity in its high-level structure and adheres to many design principles, there are significant areas where architectural drift has occurred. Notable divergences include unintended subsystem dependencies, such as the Game Engine Layer's reliance on the Audio and Input subsystems, and excessive interactions between components like the Graphics Subsystem and the VM Subsystem. These issues compromise modularity, scalability, and maintainability, which are core goals of ScummVM's architecture.

The comparison of the conceptual and concrete architectures revealed both consistent and missing dependencies. Consistencies, such as the clear separation of the Presentation, Interface, and Game Engine/VM layers, showcase the robustness of the layered design. However, missing connections, like the planned interaction between the Game Engine Layer and the Common Subsystem, highlight gaps that reduce the overall efficiency and functionality of the system. Additionally, unintended couplings, such as those within the Graphics and UI subsystems, suggest the need for better adherence to design principles during implementation.

To address these findings, several proposals for future directions are recommended. First, ScummVM's architecture would benefit from regular architectural reviews and iterative updates to align the concrete implementation with the original design. Automating dependency checks and integrating architectural constraints into the development process could help maintain modularity and reduce unintended couplings. Second, optimizing subsystem interactions, such as reducing over-reliance on the VM Subsystem, would enhance performance and scalability. Finally, addressing missing dependencies and refining the roles of subsystems, particularly in the Game Engine Layer, would improve maintainability and adherence to the original conceptual vision.

In conclusion, while ScummVM demonstrates a strong foundation in modularity and scalability, there is significant room for improvement in aligning its implementation with its design goals.

By addressing identified inefficiencies, strengthening adherence to architectural principles, and proactively refining subsystem interactions, ScummVM can continue to evolve as a robust, maintainable, and scalable system for preserving and modernizing classic games.

10.0 Lessons Learned

Analyzing ScummVM's architecture offered valuable insights into working with complex software systems and highlighted areas for improvement in our approach. One key lesson was the importance of early familiarization with tools like SciTools Understand. While the tool was critical for deriving the concrete architecture, we underestimated its learning curve. Spending more time upfront mastering its features would have streamlined the analysis and improved our efficiency. Reconciling the conceptual and concrete architectures further emphasized the need for iterative reviews to identify architectural drift and enforce modularity. The tight coupling and unintended dependencies revealed in the concrete implementation underscored the challenge of maintaining modularity during development, highlighting the need for strict design guidelines and automated checks to ensure alignment with the conceptual model.

We also learned the critical importance of clear and comprehensive documentation. The lack of detailed documentation in some parts of ScummVM's codebase made it challenging to fully understand subsystem interactions, delaying our analysis and increasing the risk of misinterpretation. In future projects, maintaining thorough documentation will be a priority to improve understanding and system maintainability. Additionally, more effective role assignment among team members could have improved our workflow. Assigning specific subsystems to individuals earlier would have reduced redundancy and enhanced overall efficiency.

Lastly, the divergences between the conceptual and concrete architectures illustrated the practical trade-offs made during development. Many unintended dependencies likely stemmed from performance optimizations or feature additions, highlighting the balance between adhering to design principles and meeting real-world constraints. Explicitly documenting these trade-offs ensures they are deliberate and well-justified. These lessons in tool mastery, iterative refinement, modularity, documentation, and team collaboration will guide us in future projects, helping us navigate similar challenges with greater preparation and insight.

References

Bowman, Ivan T., and Richard C. Holt. *Linux as a Case Study: Its Extracted Software Architecture*. University of Waterloo, 1999.

VM, Scumm. "Howto-Engines." *ScummVM*, 9 June 2023, wiki.scummvm.org/index.php/HOWTO-Engines.

VM, Scumm. "ScummVM API Documentation: Scummvm API Reference." *ScummVM API Documentation: ScummVM API Reference*, doxygen.scummvm.org/. Accessed 18 Nov. 2024.