Concrete Architecture of the ScummVM Engine

Presentation URL: https://youtu.be/yOR1JKfF7p8

Group 18

Harry George (Leader) - High level architecture diagram, Reflexion analysis

Deniz Aydin (Presenter) - Top-Level Architecture, Reflexion Analysis, Lessons Learned, Presenter

Kurtis Marinos (Presenter) - Data dictionary, Naming conventions, External Interfaces, Presenter

Ben Hilderman - Sequence Diagram, Detailed Subsystem Diagram

Riley Spavor - Top-Level Architecture

Jacob Skiba - Top-Level Architecture, subsystem analysis

Overview

ScummVM Architecture Overview

- Purpose: Enables playing classic adventure games by interpreting original game engines.
- Evolution:
  - Initially designed for SCUMM-based games (Script Creation Utility for Maniac Mansion).
  - Now supports a wide range of engines, preserving retro games on modern systems.
- Conceptual vs. Concrete Architecture:
  - Conceptual: Proposed a layered structure with modular interactions.
  - Concrete: Implements event-driven communication and the interpreter pattern for game-specific scripts.
- Key Features:
  - Game engines act as interpreters for scripts, managing visuals, audio, and inputs.
  - Subsystems interact dynamically to deliver a cohesive gaming experience.
- Architectural Analysis:
  - Reflexion analysis highlights discrepancies between conceptual and concrete designs.
  - Trade-offs made for scalability and cross-platform support.
- Recommendations:
  - Enhance scalability and adaptability while preserving the goal of retro game preservation.

Derivations

Deriving ScummVM's Concrete Architecture

- **Approach:**
  - **Analyzed the GitHub repository for code structure and subsystem roles (e.g., Engines, GUI, Common).**
  - **Used the Understand tool for static analysis of dependencies, complexity, and performance-critical components.**
  - **Consulted community forums and documentation for context and validation of findings.**
- **Key Insights:**
  - **Dependency diagrams clarified interactions between subsystems like Graphics, Audio, and Common.**
  - **SCI engine identified as a key component, with specific focus on its integration and functionality.**
- **Challenges:**
  - **Managing the complexity of subsystem interactions within a large codebase.**
  - **Required cross-referencing multiple resources to resolve ambiguities and validate assumptions.**
- **Outcome:**
  - **Combined analysis provided a detailed view of ScummVM's concrete architecture.**
  - **Established a strong foundation for reflexion analysis and deeper architectural evaluation.**

Top-Level Architecture

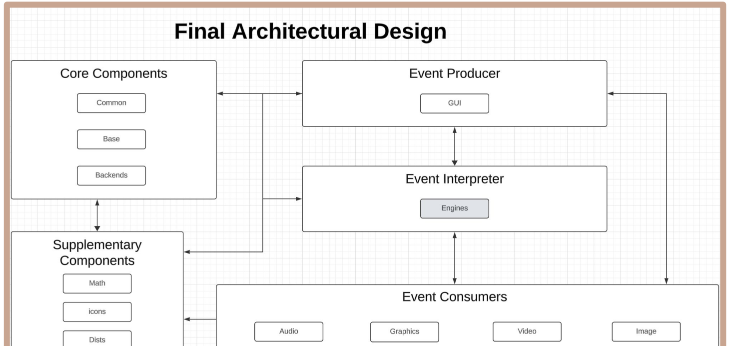
Architecture Style:

- **Event-driven model with interpreter design.**
- **Decentralized structure with dynamic subsystem interactions.**
- **Supports over 300 games while ensuring cross-platform compatibility.**

- **Key Subsystems and Roles:**
  - **Engines: Interprets game scripts, manages game states, and coordinates Graphics, Audio, and GUI.**
  - **Common: Provides shared utilities, ensuring consistency across subsystems.**
  - **GUI: Manages menus and user inputs, interacting with Engines for user-driven events.**
  - **Graphics: Renders visuals based on game-specific instructions.**
  - **Audio: Handles music, effects, and dialogue playback.**
  - **Image: Processes graphical assets for use in Graphics.**
  - **Math: Performs calculations for Engines and Graphics (e.g., scaling, transformations).**
  - **Backends: Abstracts platform-specific functionality for seamless multi-platform operation.**
- **Event-Driven Communication:**
  - **User actions (e.g., inputs via GUI) trigger events processed by Engines.**
  - **Engines dispatch tasks to subsystems like Graphics and Audio for execution.**
  - **Bidirectional communication enables real-time updates for game states, visuals, and audio.**

Top-Level Architecture Cont.

- **Interpreter Role:**
  - **Engines serve as interpreters for game-specific scripts.**
  - **Replicates original game engine behavior while maintaining modern hardware compatibility.**
- **Control and Data Flow:**
  - **Control Flow: Originates from Engines, managing dynamic subsystem interactions.**
  - **Data Flow:**
    - **Event Queues: Dispatch user inputs and game logic triggers.**
    - **Shared Resources: Managed by Common for runtime states and configurations.**
- **Advantages:**
  - **Flexible and scalable for diverse games and platforms.**
  - **Prioritizes dynamic adaptability over rigid hierarchical structures.**



Dependency Analysis

Dynamic and Decentralized Design:

- Event-driven and interpreter-based architecture emphasizes interconnected subsystems.
- Contrasts with the hierarchical conceptual model, prioritizing scalability and cross-platform functionality.

Insights from the Dependency Diagram:

- **Engines as the Central Orchestrator**:
  - Interprets game scripts and drives interactions with Graphics, Audio, and GUI.
  - Relies heavily on Common for shared resources like memory allocation, file handling, and configuration.
- **Common as the Shared Backbone**:
  - Connects all components, providing utilities and abstractions for consistent subsystem interactions.
  - Facilitates extensibility and reduces redundancy.

Dependency Analysis Cont.

- **Subsystem Specialization**:
  - **Graphics and Audio**: Handle rendering and sound playback based on requests from Engines.
  - **Backends**: Abstract platform-specific functionality for cross-platform operation.
- **Bidirectional Dependencies**:
  - GUI and Graphics exhibit real-time feedback (e.g., GUI input updates visuals, Graphics informs GUI state).

Control and Data Flow:

- **Control Flow**:
  - Event Processing: GUI captures inputs, Engines interpret events, and subsystems respond.
  - Real-Time Updates: Bidirectional communication ensures dynamic state and visual/audio updates.
- **Data Flow**:
  - Event Queues: Manages asynchronous event processing.
  - Shared Resources: Centralized in Common for consistent and efficient access.

Performance-Critical Relationships:

- **Engines and Graphics**:
  - Frequent communication ensures smooth rendering of visuals. Delays can lead to visual lag.
- **Engines and Audio**:
  - Precise synchronization ensures immersive gameplay.
- **Backends**:
  - Optimized for platform-specific operations like file I/O and input handling.

Subsystem Breakdown: The Sci Engine

Subsystem Overview:

- The SCI Engine emulates Sierra's Creative Interpreter for classic Sierra games.
- Interacts with key ScummVM subsystems:
  - **Graphics**: Renders scenes and animations.
  - **Audio**: Manages sound effects and music.
  - **GUI**: Handles user inputs and integrates gameplay.
  - **Common**: Provides shared utilities and resources.

Internal Architecture:

- **engine.cpp / engine.h**: Core execution loop, linking game logic to subsystems.
- **metaengine.cpp / metaengine.h**: Detects and configures the correct game engine.
- **dialogs.cpp / dialogs.h**: Manages in-game dialogs and overlays.
- **saves.cpp / savestate.h**: Handles save/load functionalities.
- **advancedDetector.cpp / advancedDetector.h**: Verifies game files for compatibility.

Interactions and Dependencies:

- **Engine & Meta-engine**: Engine manages execution, meta-engine handles configuration.
- **Dialogs & GUI**: Dialogs integrate with GUI for in-game interfaces.
- **Saves & Common**: Save-state functionality relies on Common for file handling.
- **Advanced Detector & Game Initialization**: Ensures correct game files for loading.

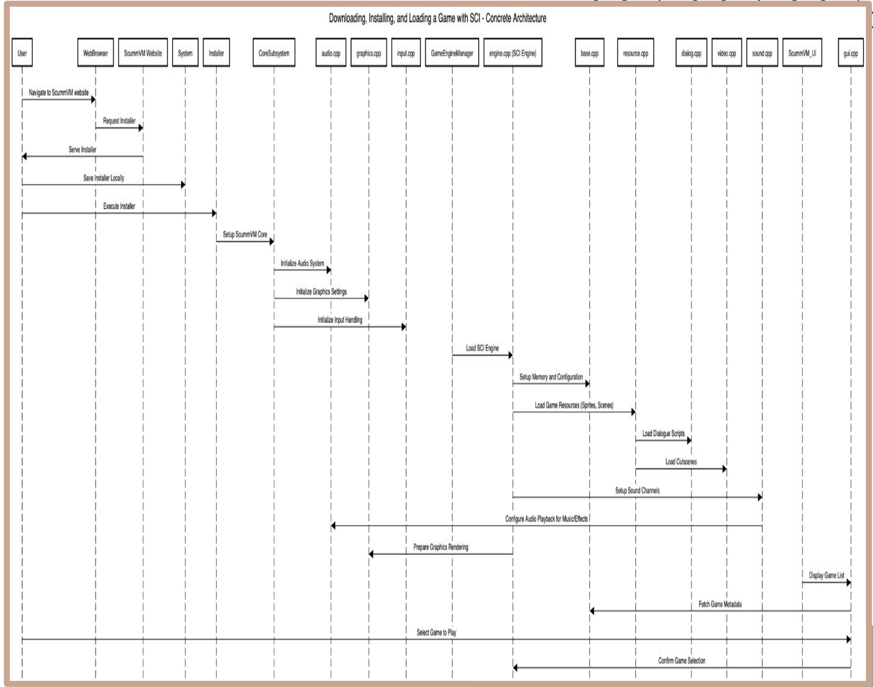
Concurrency and Performance:

- **Rendering**: Offloaded to Graphics to avoid delays.
- **Audio**: Runs on a separate thread for continuous playback.
- **Save-state**: Uses asynchronous operations to minimize gameplay impact.

Use-Case I

Downloading and Installing ScummVM

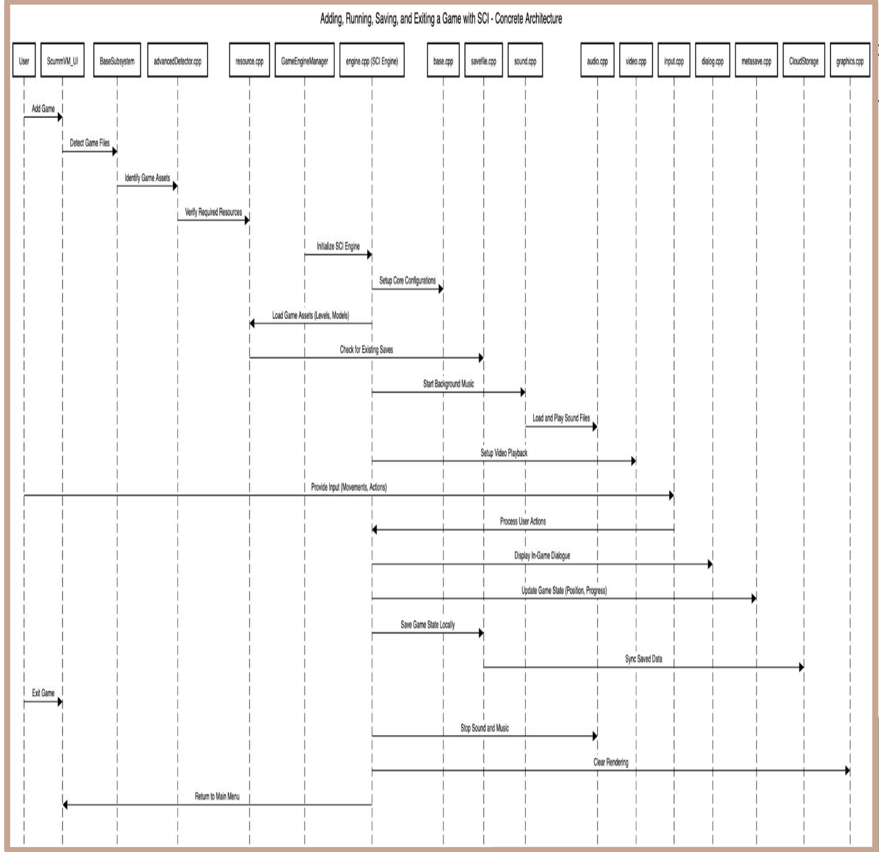
1. **Installation and Initialization**:
    - User downloads and installs ScummVM.
    - The launcher initializes the Core subsystem, loading GUI, Graphics, and Audio modules.
2. **Game Selection and Validation**:
    - User selects a game via GUI, triggering an event to the Engines subsystem.
    - The Engines subsystem validates game files through advancedDetector.cpp.
3. **Game Initialization**:
    - SCI Engine initializes game assets.
    - Triggers Graphics for rendering and Audio for sound playback.
4. **Gameplay**:
    - SCI Engine interprets game logic.
    - Dynamically coordinates subsystem responses for gameplay.



Use-Case II

Use Case: Game Addition, Gameplay, and Save/Exit

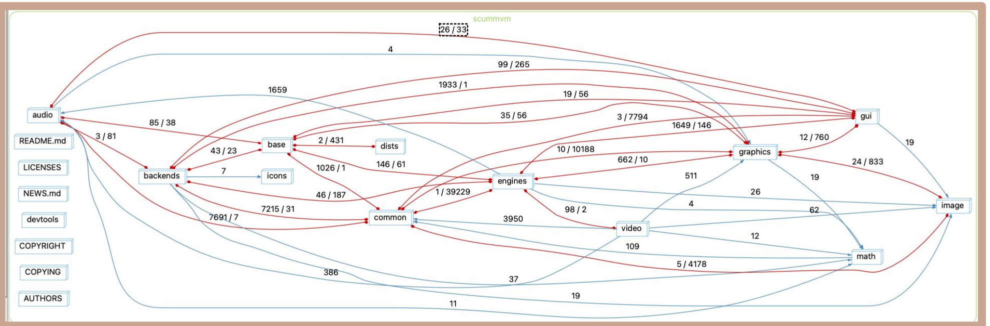
- **Game Addition**:
  - User adds a game through the GUI, triggering the SCI Engine to validate game files using advancedDetector.cpp.
- **Gameplay**:
  - SCI Engine processes real-time user inputs via GUI, updating game state.
  - Triggers subsystem actions like rendering visuals (Graphics) and playing sound (Audio).
- **Saving Progress**:
  - SCI Engine uses saves.cpp to serialize the game state.
  - Asynchronous file handling ensures minimal gameplay interruption.
- **Exiting**:
  - SCI Engine clears resources and stops audio playback.
  - Control is returned to the GUI.



Reflexion Analysis

High-Level Comparison:

- **Alignment with Conceptual Architecture**:
  - **Modular Design**: Separation of subsystems like Engines, Graphics, Audio, and GUI remains consistent.
  - **Engines Subsystem**: Central role in interpreting game scripts and interacting with other subsystems as envisioned.
  - **Common Subsystem**: Provides shared resources and utilities, maintaining consistency and reducing redundancy.



Concrete Architecture Diagram

- **Notable Divergences**:
  - **Architecture Style**:
    - Conceptual model: Layered, hierarchical design.
    - Concrete model: Decentralized, event-driven system with dynamic interactions.
  - **Control Flow**:
    - Conceptual model: Assumed hierarchical control flow.
    - Concrete model: Distributed control, with Engines orchestrating real-time interactions.
  - **Concurrency**:
    - Conceptual model: Primarily single-threaded.
    - Concrete model: Incorporates concurrency for performance optimization (audio playback, save-state handling).
  - **Subsystem Coupling**:
    - Conceptual model: Loosely coupled components.
    - Concrete model: Tighter coupling (e.g., dialogs.cpp and engine.cpp), driven by game-specific needs

External Interfaces

Interpreter-Based Design:

- **Supports diverse platforms through platform-specific backends (e.g., posix-main.cpp for Linux, macosx-main.cpp for macOS).**

Graphical User Interface (GUI):

- **Built on a customizable Theme Engine (gui/themes/) using the STX format.**
- **Manages resolution-dependent layouts and assets for consistent visuals.**

Input Handling:

- **Keyboard and mouse input are abstracted for uniform interactivity across platforms.**

Event-Driven System:

- **Manages multiple game engines, with each engine interpreting game files via Engine::run().**

Theme Creation Tool:

- scummtheme.py compiles GUI themes into zip formats or embeds them directly in ScummVM.

Cloud and Networking:

- Progressively adding cloud-based save synchronization and networking features.

Launcher and Libraries:

- The Launcher uses libraries like TinyGL and SDL for rendering, input handling, and providing a unified interface for game selection.

Lessons Learned

Subsystem Responsibilities: Clear role separation (e.g., Engines for game logic, Backends for platform tasks) supports flexibility for new games and platforms.

Concept vs. Implementation: Divergences between conceptual and concrete architectures reflect real-world constraints, like added dependencies and concurrency in the SCI engine.

Tools and Manual Analysis: Combining tools like the Understand tool with manual inspection ensures a deeper understanding of complex systems.

Documentation's Value: Developer forums and official docs are essential for clarifying design decisions and complex dependencies in large systems.

Challenges with Legacy Systems: Legacy constraints in the SCI engine highlight the need for careful integration when adding modern features.

Team Collaboration: Effective communication and shared tools are key to addressing challenges and ensuring cohesive analysis.

Recommendations: Clear roles, iterative design refinement, and leveraging tools and documentation are crucial for future projects involving complex or legacy systems.

Conclusion

Shift in Architecture: Transition from a conceptual layered model to an event-driven, interpreter-based design.

Engines Subsystem: Central orchestrator, interacting with Graphics, Audio, and GUI.

Common Subsystem: Provides shared resources for consistency across subsystems.

Prioritization: Focus on scalability and performance across diverse platforms.

Challenges: Complexity in managing dependencies and concurrency, especially with audio and save-state handling.

Balance: Successful integration of modularity, extensibility, and pragmatic software engineering principles.

Thank You

References

1. *ScummVM Documentation*. ScummVM, https://docs.scummvm.org.
2. *Developer Central*. ScummVM Wiki, https://wiki.scummvm.org/index.php?title=Developer_Central.
3. *ScummVM Forums*. ScummVM, 27 Mar. 2009, https://forums.scummvm.org/viewtopic.php?t=7886.