Analysis of ScummVM's Architectural Design

Video Link: https://youtu.be/k67KWGPIHEA

Arda Utku and Oliver Macnaughton

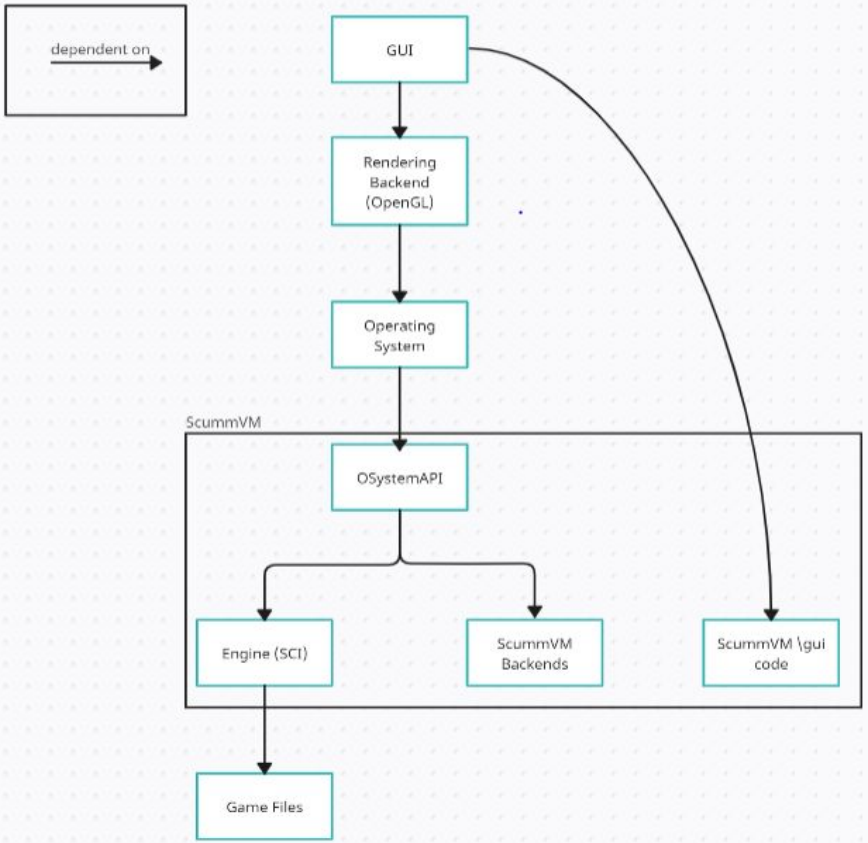
Team Member:

- Arianne Nantel (Team Lead, Participated in research, wrote Abstract and conclusion, use case and edited the report)
- Kyra Salmon (Participated in research, wrote Introduction, use case and edited the report and made reference list)
- Andrew Clawson (Participated in research wrote about roles and divisions in the report)
- Daniel Dousek (Participated in the research, wrote about how developers work together)
- Arda Utku (Presentation, participated in research and wrote about concurrency)
- Oliver Macnaughton (Presentation, participated in research and wrote about the architecture style and external interfaces)

Presentation Overview

- ScummVM architectural style
- SCI Engine
- Non-functional requirements
- Developers and Development Process
- Concurrency

ScummVM



Presentation Layer

GUI

Seen as the ScummVM launcher or running game. This layer is responsible for the user interface, and allows the user to interact with the application. The presentation layer is coordinated by the operating system, with possible help from rendering or graphics libraries such as OpenGL or SDL.

Operating System

MacOS, Windows, etc.

Manages operating system resources such as window management and input handling (resources needed by presentation layer.) Implements platform-specific tasks sent by the Abstraction layer, such as rendering updates to state in GUI. Outputs commands and resources for rendering and presenting information.

Abstraction Layer

OSystem API

Allows communication between SCI engine and operating systems, handles platform specific commands, essentially converting output of logic layer into instructions that can be interpreted by the operating system. OSystem API is dependent upon ScummVM "backends", the code that instantiates the API for a specific OS, allowing the game engine to receive input from and send commands back up to the operating system.

Logic Layer

SCI Engine

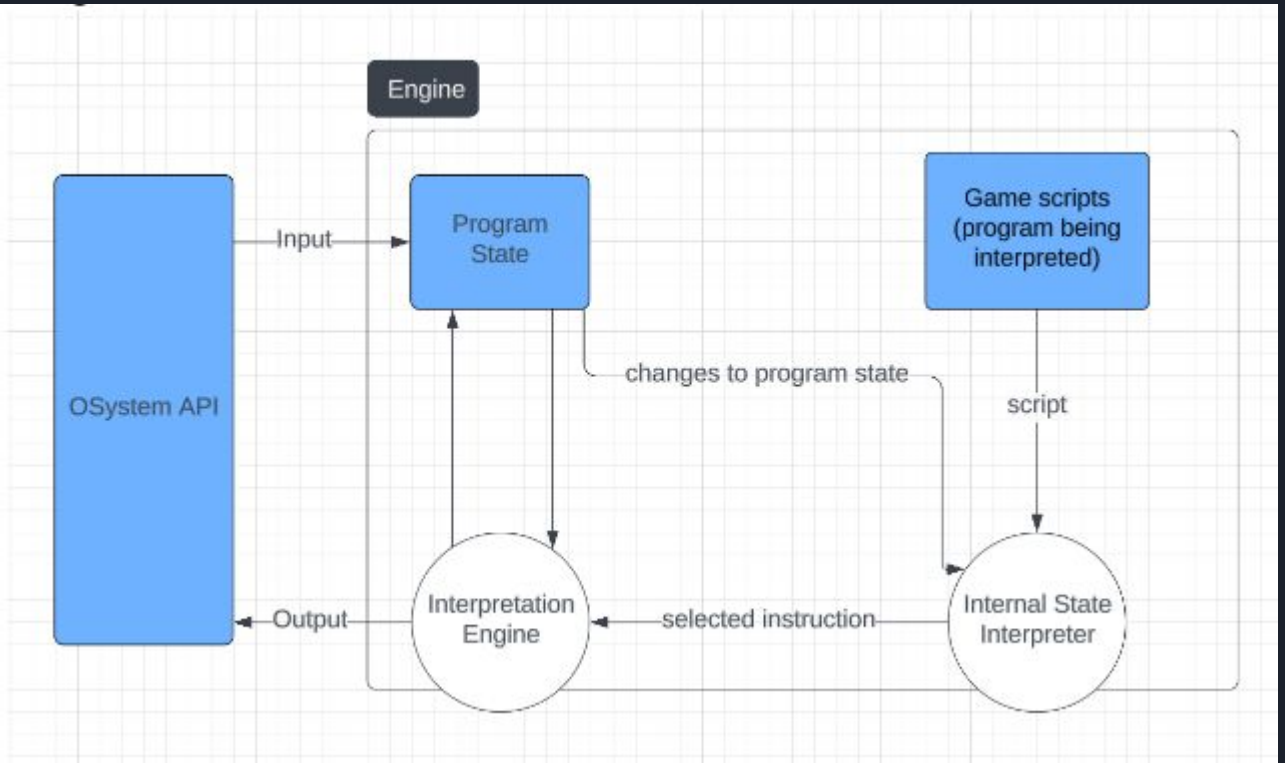
Game engine, responsible for interpretation of game files, processing of game logic, tracking of game state, handling events (button press, mouse click, etc.). Essentially controls the behavior of the game and how it runs. Sends output (new game state) to OSystem API to be represented in GUI. Receives input from user-loaded game files and OSystem API calls.

Data Layer

Game Files from Local System

Game files, scripts, animations, sprites, graphics, all specific to the game. The game files are user provided, and ScummVM must be given access to the location of the users local game files in order to load them into the engine. The data layer represent

SCI Engine



- SCI in interpreter style
- SCI engine resources: graphics, audio, logic, text
- Timer 60Hz

Non Functional Requirements

Portability

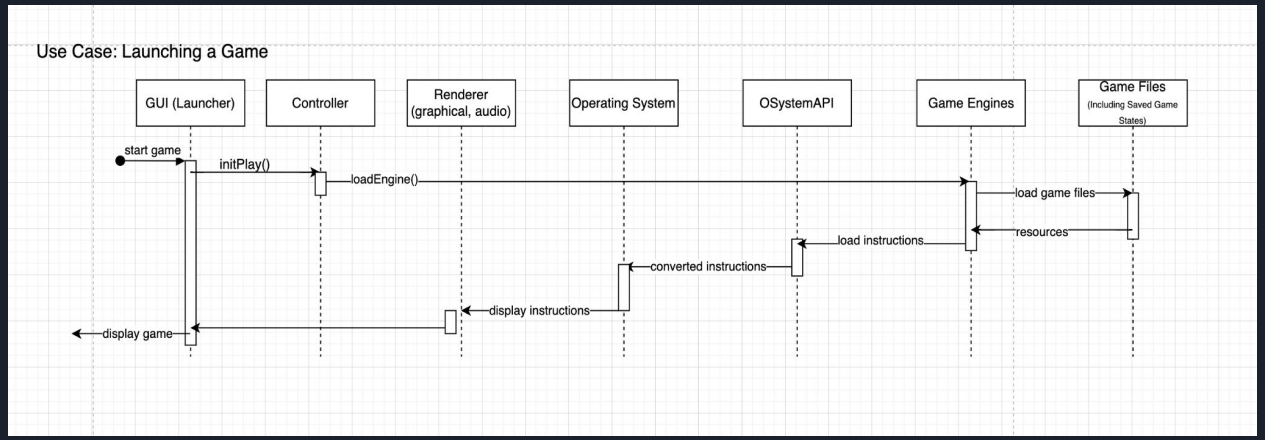
- More platforms = more users
- ScummVM backends code facilitates implementation of OSystem API

Response Time

- Overloading GPU, CPU
- codecs

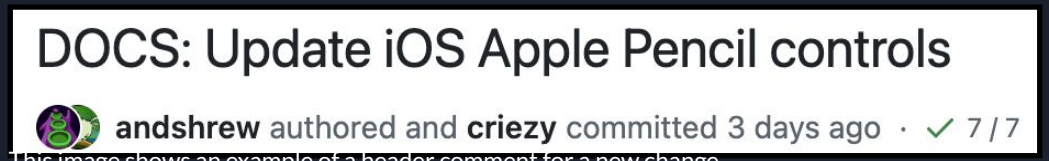
Sequence Diagram

Use Case: User Starts a game, loading in files from their computer to the game engine



Developers and Development Process

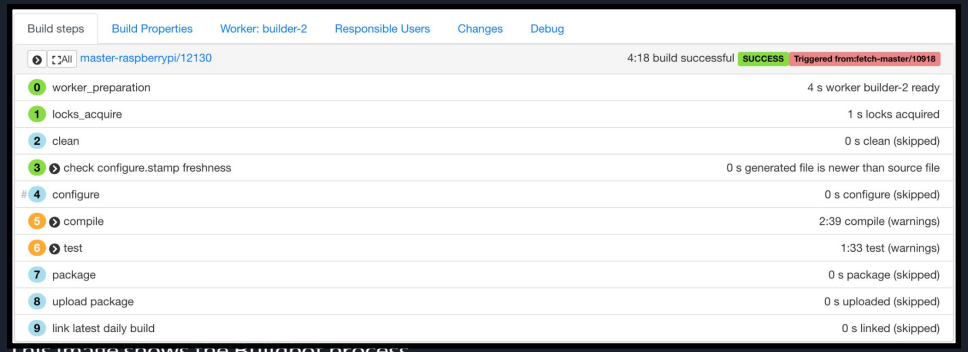
- Each developer given full ownership over their area of expertise
- Collaboration between teams and developers for significant changes to ensure codebase stability
- Coding standards followed to maintain consistency in the project
- Communication through Discord, Github, and a mailing list
- Developers can become part of a team by demonstrating their skills on relevant ongoing issues



This image shows an example of a header comment for a new change

Developers and Development Process: Buildbot

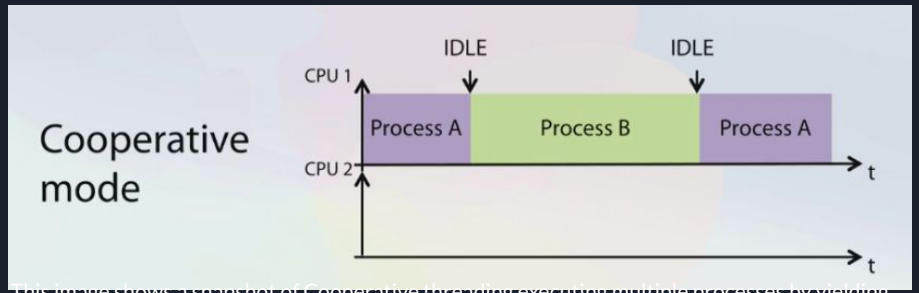
- Buildbot system is a vital tool that helps developers maintain the integrity of the code base
- Automatically compiles and tests the updated code to identify any potential issues or breakages
- Allows for continuous integrations where testing happens frequently, adapting dynamically



This image shows the Buildbot process

Concurrency Within ScummVM

- Concurrency covers the methodologies used to have a program perform multiple tasks simultaneously
- Cooperative threading used for secondary tasks such audio processing, networking, and resources loading and decompression
- Multiple threads execute simultaneously within a shared environment
- Reduces the complexity of synchronization between scripts
- Easier to implement within the virtual machine



This image shows a snapshot of Cooperative threading executing multiple processes by yielding control to one another over fixed time intervals

Concurrency Within ScummVM: Audio Processing

- Audio processing executed on a separate thread to the main execution of the program
- SDL (Simple DirectMedia Layer) deals with mixing audio channels and buffering audio data
- Processed audio data is then sent to the operating system's own audio output API
- MIDI protocol to deal with high-level audio management such as device output control

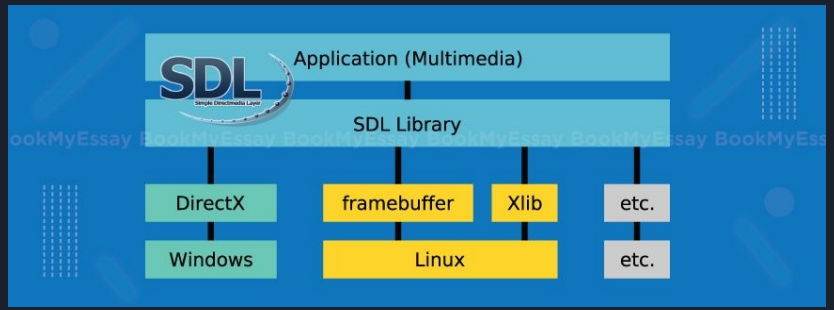


Diagram showing SDL Layers

Conclusions

Key takeaways:

- ScummVM is a layered architecture style
- Development process is open source and involves a high-degree of individual responsibility over code
- Portability is dependent upon ScummVM backends
- Simple forms of concurrency implemented for secondary tasks such as audio processing

Lessons Learned:

- Collaboration and communication
- Task Sharing

Thank You for Listening