ScummVM Games and User-Visible Metadata: An Enhancement Proposal

Authors:
Frankie Cao (frankie.c@queensu.ca)
Gavin Lacy (gavin.lacy@queensu.ca)
Aidan Tulk (21art8@queensu.ca)
Michal Wrobel (24lt12@queensu.ca)
Ayman Zaher (20az27@queensu.ca)
Selena Zou (20sz88@queensu.ca)

2 December 2024

Table of Contents

Appendix

1. Abstract

As an open-source project, ScummVM provides its users with the ability to play classic video games of the point-and-click adventure or role-playing varieties on various platforms; namely, platforms for which these games were never developed [1].

In our A1 report, we proposed a layered architectural style for the ScummVM system and an interpreter style for its commonly used SCI engine. Through analysis using the Understand 6.5 tool and ScummVM's GitHub repository, we concluded that these architectural styles hold true in the concrete architectures of the systems, with some modifications to components and their dependencies. These changes are discussed in detail in our A2 report, but to summarize, two-way dependencies exist between almost every combinatorial pair of architectural components, with strong one-way leanings whose overall flow supports the layered style of ScummVM. That is, some components can communicate with others that (in a traditional layered architecture) they should not be able to, likely in an effort to improve responsiveness and performance for games.

In this A3 report, we look at implementing an improvement to ScummVM: the ability to display the default audio/graphics engine and other information for the running game. Using an SEI SAAM analysis [2], we discuss two possible methods of implementation: one where the information is displayed in the settings pane, and another where a GUI overlay displays the details over the game while the user plays. We discuss how this enhancement will not require substantial changes to the architecture, and how its implementation will affect key non-functional requirements and stakeholders. We conclude that our proposed enhancement is best implemented as information displayed in the settings pane, due to it being easier to develop, test, and use.

2. Introduction and Overview

The lifecycle of a software architecture often starts out with simple diagrams and plans, before gradually – through enhancements and hack fixes – drifting further and further from its conceptual architecture. Although ScummVM's change was not so drastic, the conceptual layered architecture still displayed many unexpected dependencies.

Enhancing the architecture of a system is an easy way of improving a long-lived system like ScummVM. These enhancements do not majorly change the architecture of a system, but are made to improve system functionality. ScummVM has seen multiple enhancements over the years, from simple quality-of-life improvements to the addition of new engines to increase the number of games that can be played [3].

After investigating ScummVM's documentation and codebase, we propose our own enhancement. Following a current feature request ticket [4], we propose that ScummVM users may benefit from being able to view certain metadata about the game, such as: the audio/graphics engines being used, current framerate, filesystem location of the game, etc. This lets the user receive an in-depth view of the inner workings and performance of ScummVM, allowing them to improve their settings or observe how the program functions.

We consider two options for implementation. In the first, the information is displayed in the settings pane of the Game Launcher. In the second, the information is displayed in a GUI overlay during gameplay. Both options and their effects on the ScummVM architecture are analyzed for their pros and cons. We conclude that the second option is superior, as it involves less disruption to the overall codebase, the Engines subsystem, and user satisfaction.

3. Architecture

3.1 Overview

We present an SEI SAAM architectural analysis [2] of our enhancement, which is to display information about the default audio/graphics engines and other game information to the user. We consider the following two options for an implementation of our enhancement:

1. Updating the Game Launcher in the GUI to display the names of the default audio/graphics engines and other information, not just "<default>".
2. Adding an in-game overlay to display the default audio/graphics engines and other information during gameplay.

For both options, we discuss the perspectives of key stakeholders in the system, the potential impacts on ScummVM and the SCI engine, and assess the risks of the implementation.

3.2 SEI SAAM architectural analysis preliminaries

3.2.1 Stakeholders

The primary stakeholders of the ScummVM system are the system and game developers (including maintainers), testers (both in-house and public), end users, and various legal and licencing stakeholders (e.g., the original developers of games supported by ScummVM). Since the proposed enhancement does not deal with adding or enhancing any specific games or engines, and ScummVM already has licences for the use of any audio or graphics engines necessary for gameplay, the role of legal stakeholders is minimal. Therefore, our discussion centres primarily on the effects of this enhancement on the ScummVM developers, the testers, and the end users.

3.2.2 Nonfunctional requirements (NFRs)

As ScummVM is by nature an interactive system, performance (specifically, low user latency) is a central concern to any enhancement of the system. Due to its advertisement as a highly portable system, portability is also a major concern; however, because our enhancement is localized to the GUI and engines components, and the portability of these is abstracted away by calls to the common utility classes and backends [5], we may conclude that portability is not an immediately-relevant concern.

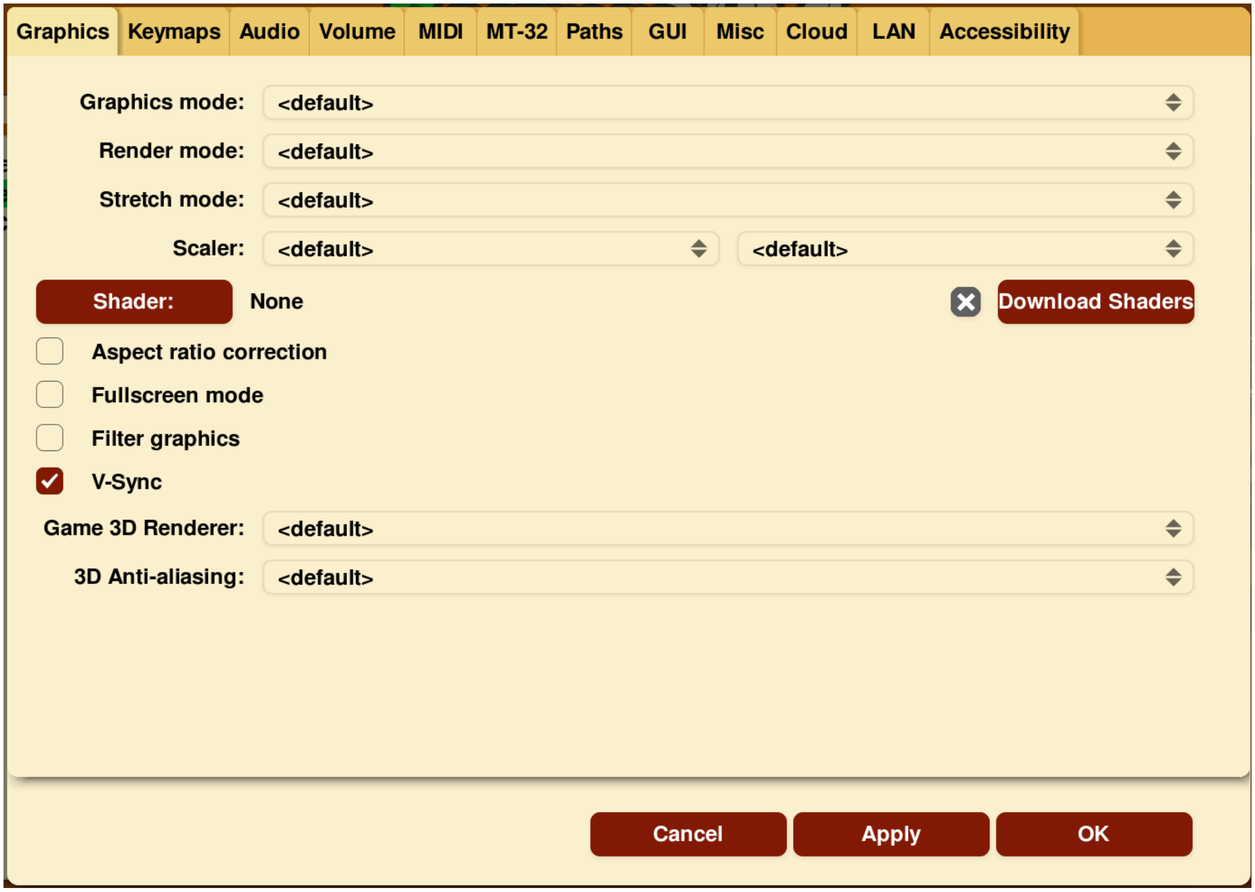
Thus we have established that the central concern for all stakeholders is performance. For each group of stakeholders, we also identify additional NFRs important to them.

1. *Developers.* As ScummVM developers are often also the maintainers of their own code, these stakeholders are naturally concerned with the maintainability and reusability of their enhancement code.
2. *Testers.* For testers, especially robustness testers, reliability is a concern.
3. *End users.* The primary concern for end users, apart from performance, is usability – i.e., user-friendliness of the information presentation. As our enhancement derives directly from a user feature request, usability must carry considerable weight in our nonfunctional concerns.

In the rest of this report, we will primarily focus our efforts on the NFRs of performance and usability, and to a lesser extent, maintainability and reliability.

3.3 Option 1: Enhancing the game launcher

Currently, in the settings pane, the user is only displayed the text “<default>”, with no further information available (examples below). Our proposed enhancement would replace this uninformative “<default>” text with the name of the actual default audio/graphics engine.



Graphics mode: <default>

Render mode: <default>

Stretch mode: <default>

Scaler: <default> <default>

Shader: None

□ Aspect ratio correction

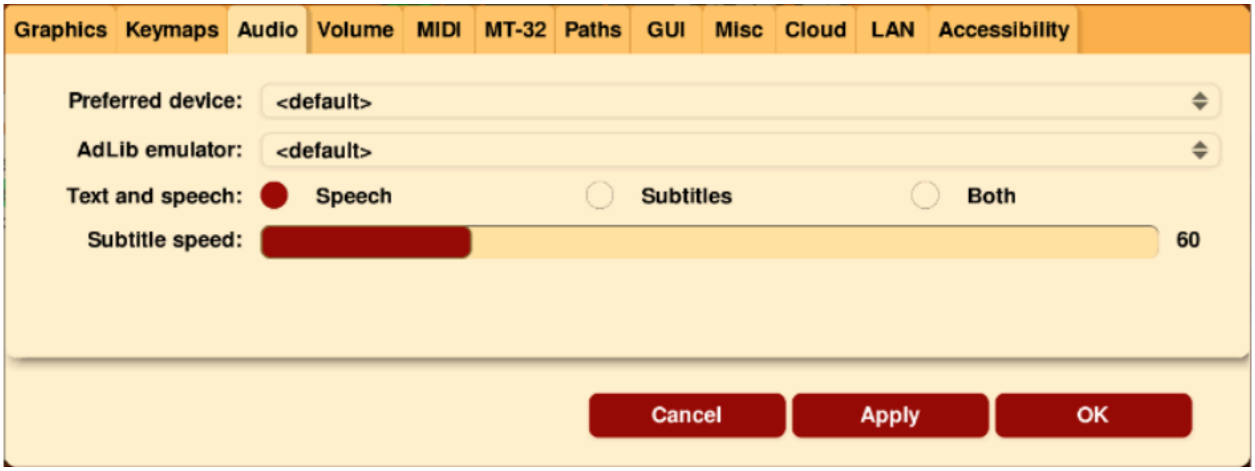
□ Fullscreen mode

□ Filter graphics

☑ V-Sync

Game 3D Renderer: <default>

3D Anti-aliasing: <default>



Preferred device: <default>

AdLib emulator: <default>

Text and speech: ● Speech ○ Subtitles ○ Both

Subtitle speed: 60

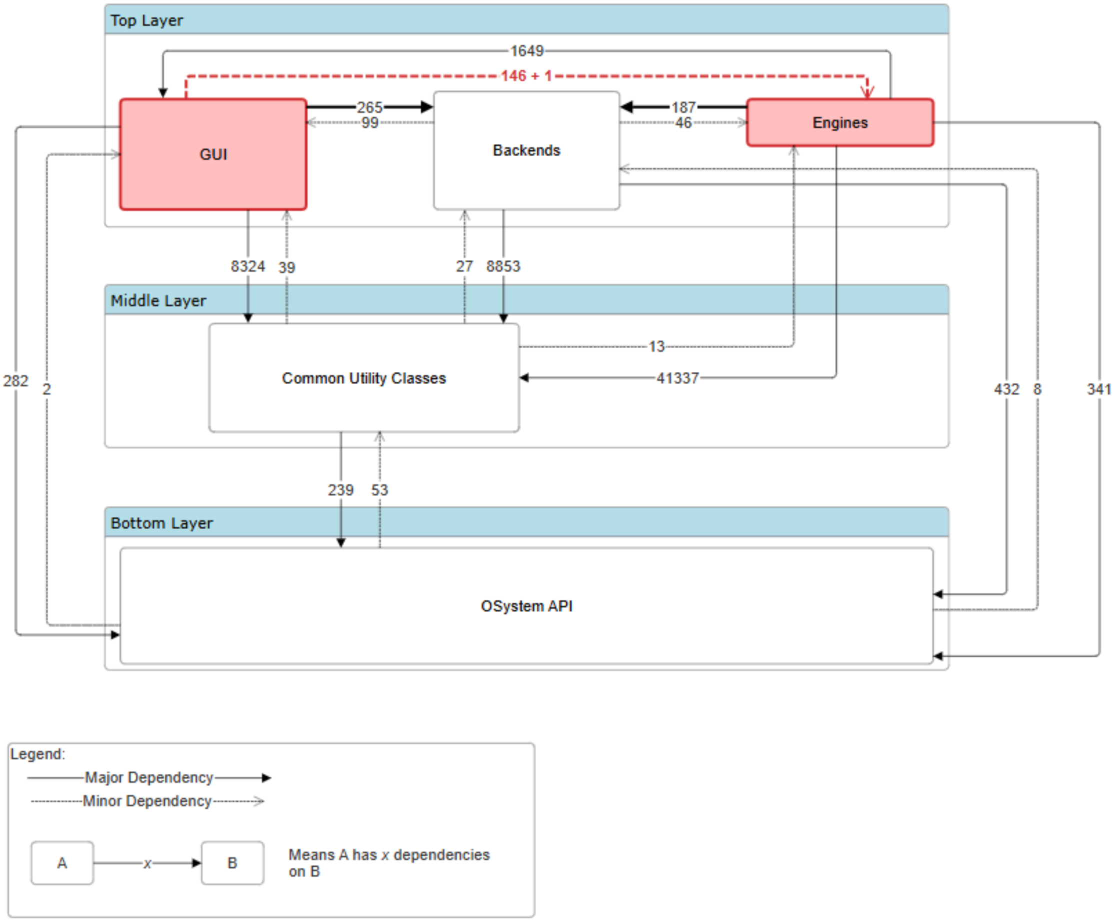
3.3.1 Effects on ScummVM

As only the Game Launcher would display the audio/graphics information, this approach to implementing our enhancement would be limited to the GUI. Specifically, the files and directories most directly implicated would be the scummvm/gui/ directory, and the gui-manager.cpp and launcher.cpp files (where the overall application is launched and the Game Launcher pane displayed, respectively) [5]. Additionally, because each individual game engine must define which audio and graphics engine(s) it supports, as well as which engines are used as default, any new queries the GUI may have to make should be directed to the engines component. As this is the only potential new dependency, and the GUI and engines components already reside in the same layer, no changes to the high-level architecture are necessary.

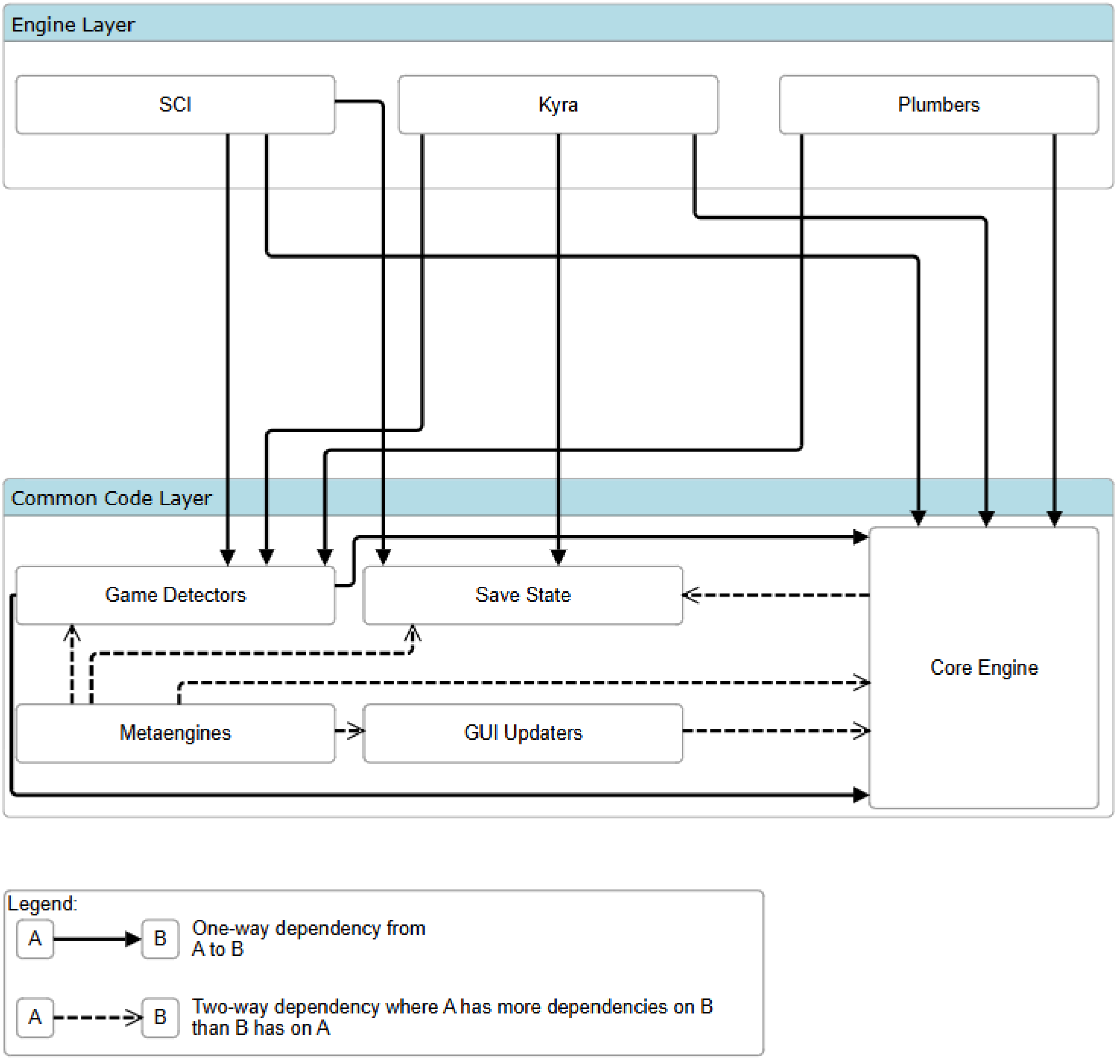
The low-level details of querying and storing the audio/graphics information are beyond the scope of this paper. However, we may outline two likely scenarios:

1. The GUI component may already query and store the audio/graphics information in a local data structure. In this case, the only code change necessary is to process this data structure to display the desired information to the GUI pane.
2. The audio/graphics information may require a dynamic query to the engines component to retrieve its defined default engines. In this case, the necessary code change is to add a query from the GUI to the engines component, then output the retrieved information to the console.

Although scenario (1) is obviously simpler, both possibilities are quite lightweight. The diagram below describes scenario (2).



3.3.2 Effects on the SCI engine



As this change would only affect the global graphics and audio settings GUI, the SCI engine implementation will not need to be modified. The SCI engine, like all engines in ScummVM, can already query the options selected in the global graphics and audio GUI menus and alter its behaviour accordingly. Since this change would only be adding extra information in the graphics and audio settings GUI interface, there would be no change in the data the SCI engine would receive from querying the options selected. Additionally, since the SCI engine will not need any modifications and the performance of the backend code it utilizes via the OSystem API will not be impacted, the performance of the SCI engine will also not be impacted by this change.

3.3.3 Risks

The risks of this option are minimal. The primary failure case occurs either when the GUI query to the engines component fails to return valid data, or the retrieved data (i.e., audio/graphics information) fails to be displayed correctly in the Game Launcher.

Both of these potential failures can be mitigated by proper error checking in both the query and display code, and the display of either "<default>" or an error message in the console. As a failure leads only to a display error in the GUI, before any game is launched, it should not interfere with gameplay and therefore is unlikely to cause a user-perceived critical error.

Additionally, because the query and display occur in the Game Launcher, prior to launching the game itself, performance hits are not only unlikely (a single function call in C++ executes in negligible time), but would not affect game performance. Thus we conclude that this option carries minimal risk to the system and to user satisfaction.

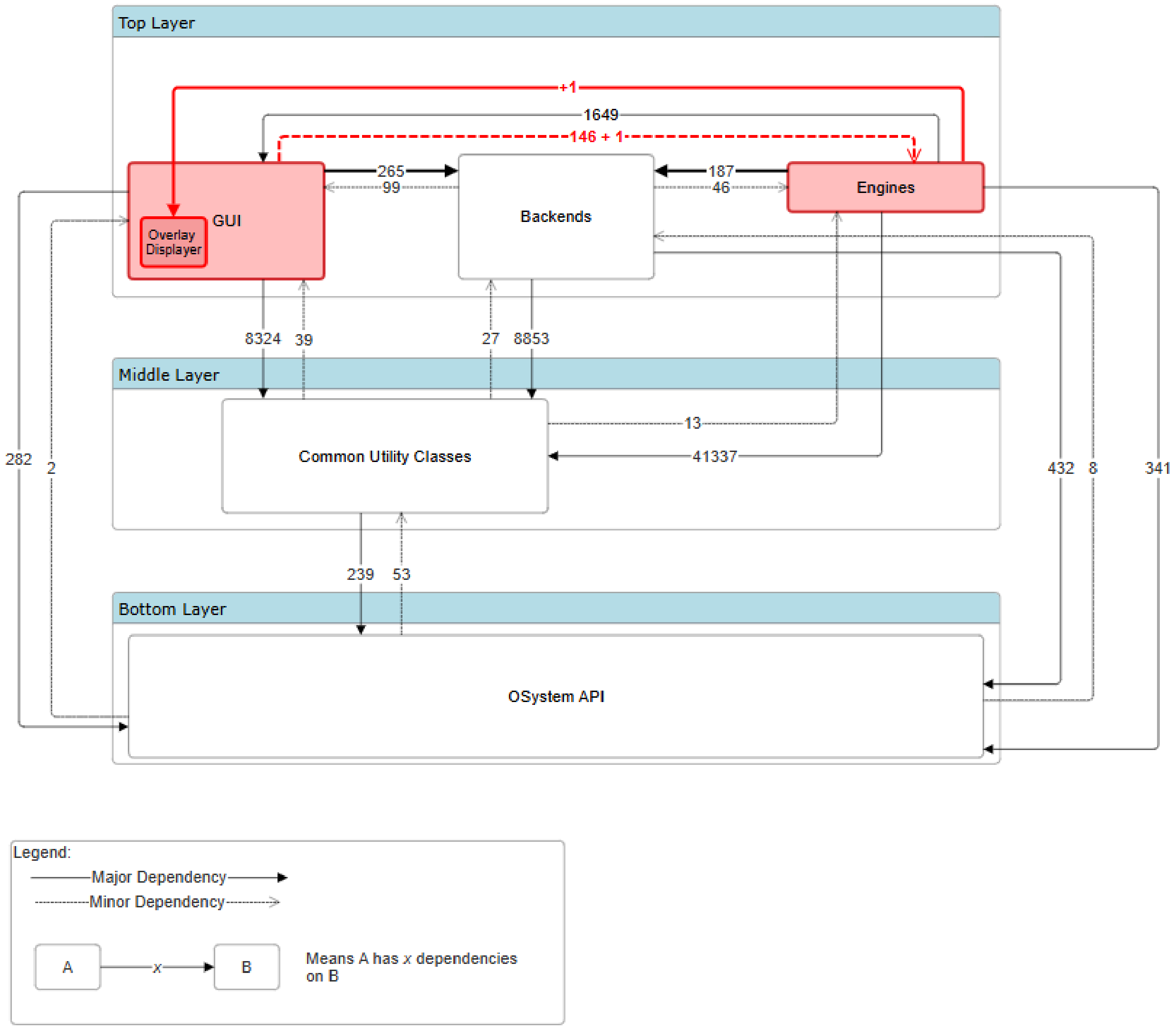
3.4 Option 2: Adding an in-game overlay

This option is significantly more far-reaching than the previous one. In this option, an in-game overlay would display the same information about the audio and graphics engines during gameplay somewhere on the user's screen. To avoid modifying core engine graphics code (which is highly risky due to its use by each individual game engine), we propose the creation of an additional global GUI subcomponent that is always visible during gameplay (e.g., in a corner of the user's window) and does not pause the game engine. Of course, this requires a higher degree of GUI-engines integration than is currently present. However, this approach provides a mechanism that can be further expanded in the future to display engine/game specific information to the user during gameplay.

3.4.1 Effects on ScummVM

In this implementation, the global GUI component would be displayed on top of the game engine graphics in the same window. In order to retrieve and display the appropriate audio/graphics data, the GUI would still need to query the current game engine. This, depending on whether the GUI already saves the desired audio/graphics information or not, would potentially require a new function call to the engines component. Additionally, each game engine would be required to call the GUI to display the overlay component upon game start-up. However, because these two components (GUI and engines) both reside in the top layer of the system architecture and therefore can communicate easily, no changes to our high-level architecture are required.

With respect to the specifics of implementation, this option would require the creation of a new, global GUI subcomponent, whose express purpose is to display the audio/graphics data. This component would be called upon game engine start-up, and it would in turn call the game engine to obtain the necessary audio/graphics data. Once again, the main directory implicated is the scummvm/gui/ directory [5]. A new module, named "in_game_overlay" (for example), may be the best way to encapsulate this new subcomponent. The work required to develop and test this change is likely not trivial. The diagram on the next page describes the new dependencies needed.



3.4.2 Effects on the SCI engine

ScummVM already allows users to open a global menu during the gameplay of any engine/game. To do so the user can press CTRL+F5 by default or set a custom keybind to pause the engine/game and open the global menu providing options like going back to the launcher, exiting and changing some gameplay settings. ScummVM utilizes the GUI component to create and display the global menu whenever the keybind is pressed and utilizes the base engine code to pause the engine. For the implementation of the in-game overlay a similar approach can be utilized to display a global GUI overlay component always on top of any running engine/game.

Following this approach to displaying the new global GUI overlay component, the SCI engine will not require any modifications to allow the display of the component as it will be displayed globally. The SCI engine, like all engines in ScummVM, is solely responsible for rendering the desired game and plays little role in displaying GUI components such as the global menu and the proposed in-game overlay in this option.

Additionally, since the GUI component will be global, the SCI engine will not need to expose any new data or functions as the information required by the in-game overlay can already be retrieved.

3.4.3 Risks

Although many risks are eliminated by avoiding the modification of core engine code, the risks that remain are significant. As mentioned previously, development work is nontrivial. In particular, a large – possibly impractical – scope of testing is required, as each game engine must be tested (i.e., have a game played) separately to ensure that the new GUI overlay displays properly and does not interfere with the game engine's actions.

Furthermore, the processing power necessary to continuously render the GUI overlay *during* gameplay may cause significant performance hits. Although this overlay is static in nature, unlike the dynamic gameplay which consumes most processing power, it still requires processing and rendering resources. In an interactive system like ScummVM, which operates under real-time constraints, such a nonessential resource usage may lead to unacceptable user latencies, which in turn would negatively affect user satisfaction.

Finally, because the overlay would be displayed constantly during gameplay, the usability of the system might suffer. Players might find that the overlay restricts their field of vision in the game, or that it generally interferes with gameplay. Somewhat ironically, the larger and more detailed this overlay and the information it presents, the more likely user satisfaction is to suffer.

3.5 Test plans

ScummVM maintains a test suite of functional, nonfunctional (e.g., performance testing), and operational test cases [6]. For either proposed implementation, this test suite should be run to ensure that our enhancement does not cause a failure in some existing part of the software and maintains performance requirements. However, since these tests are automated and generally limited to utility code, additional test cases are required to guarantee performance and usability.

For Option 1, the scope of the change is small. Thus, the necessary test case is to open the Game Launcher, select a game, open its settings pane, and check that the correct default audio/graphics engines are displayed, rather than the old "<default>" text. This should be repeated for several games, although not all games need to be tested. For completeness, a brief playthrough of the games may be conducted to ensure that no in-game functionality was affected.

For Option 2, the scope of the change is much larger. In addition to the automated test cases, at least one game from each game engine should be launched and a playthrough conducted, in order to ensure that the audio/graphics information is displayed correctly, and additionally that no preexisting in-game functionality is affected. Compared to Option 1, where the main concern was the Game Launcher display and not the effects on the gameplay itself, the overhead of testing Option 2 is obviously much greater.

For either option, user acceptance testing should also be conducted to ensure user satisfaction.

3.6 Comparison of options and conclusion

In our SAAM analysis so far, we have detailed two distinct options for implementing our enhancement and discussed the impacts and risks of both. In the first option, implementation is straightforward – development work is relatively lightweight, testing is possible via a combination of automated and manual tests, and usability and end-user satisfaction are unlikely to suffer, due to the isolated nature of the change. By contrast, in the second option, implementation requires a new module, resulting in nontrivial development, testing, and maintenance work; the change is far-reaching, possibly negatively affecting the usability of the gameplay window as well as the overall gameplay performance; and the testing overhead is massive.

In view of these observations, our choice is clear. Option 1 is easier for developers to implement and testers to test. And because no core functionality changes (in fact, the audio/graphics details can only be accessed on a user's own initiative in the Game Launcher), the risk of offending end users is negligible. Therefore, we evaluate Option 1 as being the better choice for our enhancement.

4. External Interfaces

Overall, the external interfaces involved with ScummVM would not be majorly affected by the new overlay, as the overlay only provides details as to the statistics of the game. No enhancements are made to these components as these are not part of the ScummVM system. However, there are slight mentions of these systems within the new overlay that should be mentioned.

4.1 Game Files

The statistics will display the file path where the game files are located. So while the game library has not changed, the details of the location will now be properly communicated to the user.

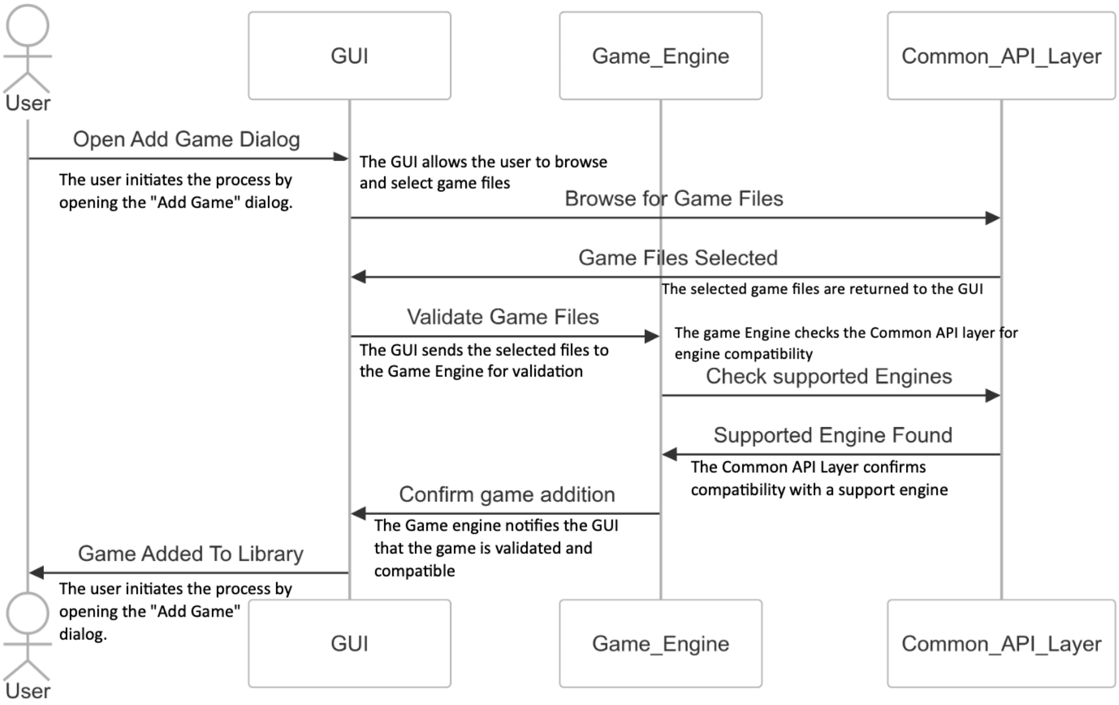
4.2 Cloud Services and Local Web Server

If the game's save data was found in either a cloud server or from a local web server, these details will be displayed.

5. Use Cases

5.1 Changing the Settings

Since the use case of adding a game would not change, for the first use case we will explore what will happen when changing the settings. This use case is very similar to adding a game in the sense that it interacts with most of the same components. When a user is inside ScummVM to change the system, they interact with the GUI – much like adding a game – but instead of interacting with the file subcomponents inside the Common API layer, they interact with the configuration file when you change a setting. Looking at the concrete architecture, the specific file to change settings is the "scummvm.ini" file.

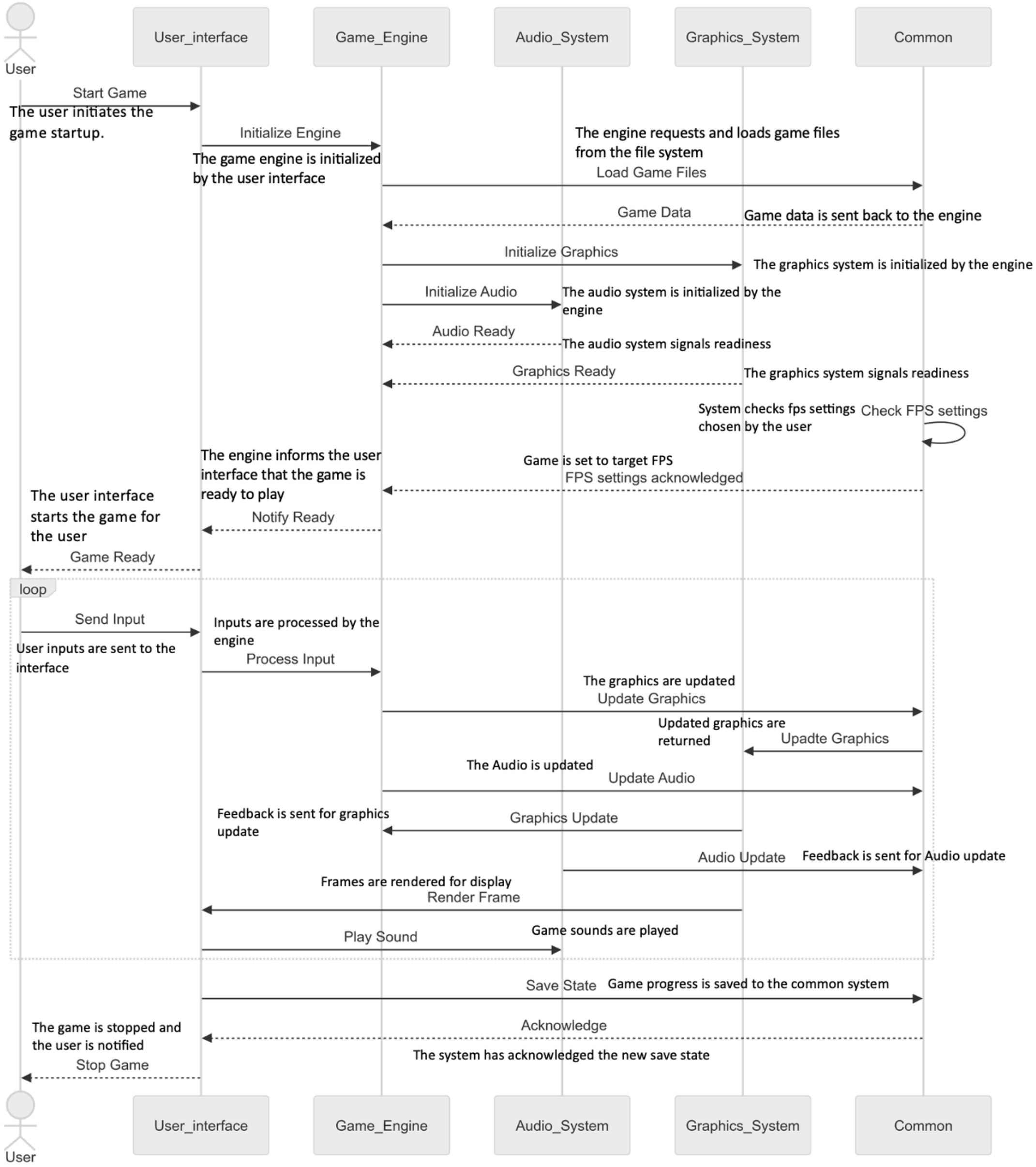

User
GUI
Game_Engine
Common_API_Layer
Open Add Game Dialog
The user initiates the process by opening the "Add Game" dialog.
The GUI allows the user to browse and select game files
Browse for Game Files
Game Files Selected
The selected game files are returned to the GUI
Validate Game Files
The GUI sends the selected files to the Game Engine for validation
The game Engine checks the Common API layer for engine compatibility
Check supported Engines
Supported Engine Found
The Common API Layer confirms compatibility with a support engine
Confirm game addition
The Game engine notifies the GUI that the game is validated and compatible
Game Added To Library
The user initiates the process by opening the "Add Game" dialog.
User
GUI
Game_Engine
Common_API_Layer

5.2 Playing a game

For the use case of playing a game, this change can be easily implemented from an architectural standpoint by including a minor addition to the common component of the system where it checks for the settings. This highlights the benefit of just enhancing the game launcher because most of the sub-component interactions will be within their own components and there will be little to no changes to the external dependencies.

However, if we were to create a global GUI subcomponent then this would be a large architectural burden as mentioned previously in the assignment. There would need to be dependencies within the engine, GUI, and possibly other components in order to relay relevant information back and forth.

As can be concluded by the use-case diagram, both options of enhancement will benefit the user experience for the use case of playing a game. It can also be noted that the relatively simple implementation of the enhancement will cause a negligible change to throughput of the system.


User
User_interface
Game_Engine
Audio_System
Graphics_System
Common
Start Game
The user initiates the game startup.
Initialize Engine
The game engine is initialized by the user interface
The engine requests and loads game files from the file system
Load Game Files
Game Data
Game data is sent back to the engine
Initialize Graphics
The graphics system is initialized by the engine
Initialize Audio
The audio system is initialized by the engine
Audio Ready
The audio system signals readiness
Graphics Ready
The graphics system signals readiness
System checks fps settings chosen by the user
Check FPS settings
The engine informs the user interface that the game is ready to play
Game is set to target FPS
FPS settings acknowledged
The user interface starts the game for the user
Notify Ready
Game Ready
loop
Send Input
Inputs are processed by the engine
User inputs are sent to the interface
Process Input
The graphics are updated
Update Graphics
Updated graphics are returned
Upadte Graphics
The Audio is updated
Update Audio
Feedback is sent for graphics update
Graphics Update
Audio Update
Feedback is sent for Audio update
Frames are rendered for display
Render Frame
Play Sound
Game sounds are played
Save State
Game progress is saved to the common system
Acknowledge
The game is stopped and the user is notified
Stop Game
The system has acknowledged the new save state
User
User_interface
Game_Engine
Audio_System
Graphics_System
Common

6. Conclusions

In this report, we proposed two distinct ways of providing audio/graphics and settings information to the user: one which uses an existing settings pane in the GUI, and one which would implement a new overlay in the running game. From our SEI SAAM architectural analysis, we conclude that the first option minimizes development and testing overhead while guaranteeing equivalent or better fulfillment of the essential NFRs, performance and usability. We therefore recommend this option for feature implementation.

Future directions for research include an in-depth investigation into the low-level details of implementation (e.g., what changes to data structures in the GUI component are necessary). With more concrete information, we may refine and optimize our implementation recommendations.

Appendix

7. Data Dictionary

Term	**Definition**
Enhancement	Changes made to software to improve its overall quality.
SEI SAAM analysis	(Full acronym below) A structured approach to the analysis of software architectures with a focus on non-functional requirements and stakeholder involvement [2].
Audio/graphics engine	Software to control sound effects and graphics rendering in computer games.
Stakeholders	People or groups affected by a software development project. They may be internal or external to the organization.
Non-functional requirements	Specifications for how well a software system should perform; e.g., performance, usability, security, scalability, etc [7].

8. Naming Conventions

Abbreviated Term	**Full Form**
GUI	Graphical User Interface
SCI	Sierra Creative Interpreter
NFR	Non-functional requirement
SEI SAAM	Software Engineering Institute Software Architecture Analysis Method
API	Application programming interface

9. Lessons Learned

This report is limited primarily by time constraints and a lack of familiarity with a ScummVM user's perspective. As such, we learned that the best way to generate ideas for feature enhancements was to read through high-priority ScummVM bug and feature request tickets to gain an understanding of user needs. We propose that this research could have been supplemented by experimenting more with ScummVM – namely, by downloading and running games in the interface ourselves. Then we could more easily have narrowed our scope to ideas that are both plausible and attractive to users, and conducted a more thorough investigation of the low-level implementation details, which are the primary details lacking in this report.

References

[1] "What is ScummVM?" ScummVM. Accessed 2 Dec. 2024. Available: https://www.scummvm.org/.

[2] R. Kazman et al. "SAAM: A Method for Analyzing the Properties of Software Architectures." *Proceedings of 16th International Conference on Software Engineering,* Sorrento, May 1994, 81-89. Available: http://doi.org/10.1109/ICSE.1994.296768.

[3] "ScummVM History." ScummVM Wiki. Accessed 2 Dec. 2024. Available: https://wiki.scummvm.org/index.php/ScummVM_History.

[4] "Feature request: options could show *which* <default> is actually chosen for given game." ScummVM Issue Tracker. Accessed 2 Dec. 2024. Available: https://bugs.scummvm.org/ticket/13995.

[5] Sandulenko, E. et al. "scummvm/gui," Github Repository, 2024. Accessed 2 Dec. 2024. Available: https://github.com/scummvm/scummvm/blob/master/gui/.

[6] "Developer Central." ScummVm Wiki. Accessed 2 Dec. 2024. Available: https://wiki.scummvm.org/index.php?title=Developer_Central.

[7] A. E. Hassan and B. Adams. (2024). Module 02: Non-Functional Requirements (NFR) – Quality Attributes [PowerPoint slides]. Available: https://onq.queensu.ca/d2l/le/content/959322/viewContent/5711375/View.