ScummVM Enhancement Report

ScummAI Companion

Authors - Group 11

Andrew Clausen	21awc7@queensu.ca
Arda Utku	21au10@queensu.ca
Arianne Nantel	20aan8@queensu.ca
Daniel Dousek	22dd1@queensu.ca
Kyra Salmon	21kms22@queensu.ca
Oliver Macnaughton	20owdm@queensu.ca

Fig. 1 Potential Sprite Design

0.1 Abstract

In this report, we explore 2 ways of implementing a new feature to ScummVM. This new feature is an AI Companion that provides hints to the user either through a white box approach or a black box approach. The Black Box approach trains the AI model on countless hours of learning from expert gameplay available in the public domain on the internet. While the White Box approach modifies elements in the SCI engine, adding a new AI companion component and the hook observation component. We then explored the different stakeholders involved for each implementation method. Namely the End User, the project leadership team, the documentation team and the developers. For the Black Box approach, the developers involved are the subsystems developers, comparatively for the White Box approach, the engine division is an important stakeholder. We then discussed the important NFRs for each approach. Differentiating how they affected each stakeholder. In the end we determined the Black box method is the better implementation method.

0.2 Introduction and Overview

To enhance ScummVM, we propose the addition of an AI companion. This would constitute a small sprite, overlaid over the graphics of any game already able to be run on the ScummVM software. When toggled on in the settings menu, the AI would use its ‘knowledge’, (developed either through black box or white box training) and provide an on-screen text hint. This is not a chatbot, the AI would simply fill in pre-existing sentence scaffolds with hints, prompts, or tips relevant to the players progression at their current game state. Ideally the hints would also be provided at levels, (easy, medium, or hard), however this is not discussed in detail in this report. The primary focus of this report is covered in three sections; Proposals, Implementation and Effects. Throughout the report, we discuss the different benefits of each approach, as well as the effects either approach will have on the conceptual and concrete architecture, the stakeholders and their respective non-functional requirements (NFRs), and the gameplay.

Within *Section 1. Proposals*, we discuss the details of how the AI companion would function and the differences in training method implementations we propose. Within gameplay for the end user, these two approaches are intended to be indistinguishable. The differences lie

entirely in the training methods for the AI model and the implementations in the architecture in the low-level code. The Black Box approach trains the AI model on countless hours of learning from expert gameplay available in the public domain on the internet. In the alternative White Box approach, the modifications are integrated seamlessly with the ScummVM code base. The primary changes take place within the SCI engine. Here, within the interpreter, is where all the AI companion related scripts will be stored. In this approach, instead of being pre-trained on each individual game, the AI module will parse game scripts, searching for key words and functions. As the AI module concurrently tracks the gameplay and monitors the code internally, it will be able to generate hints based on what is required of the player in any given moment to progress their gameplay.

Section 2. Implementation details the changes required to the concrete and conceptual architecture for the implementation of the AI companion to be successfully integrated into the pre-existing ScummVM architecture. In order to add the proposed AI companion enhancement, changes will be required to be made to the architecture discussed in report 2 regardless of the implementation. The White Box approach integrates the AI directly into the existing ScummVM codebase; primarily within the SCI engine. It crawls directly through the codebase, providing a more accurate and context-sensitive hint generation system. The Black Box application is designed modularly, as its own layer. The AI Companion layer is placed inside the layered architecture between the graphics and OSystemAPI layers, allowing it to communicate up and down. The Implementation of the AI companion also has major ramifications for the external interface. As modifications will be required for the in-game settings menu for the player to toggle on and off the companion.

In *Section 3. Effects,* the stakeholders affected by the new proposals are denoted. Their key NFRs and how those will be benefited, harmed or altered as a result of the enhancement are discussed. While some are quite similar regardless of approach, many vary greatly depending on how the AI is implemented. The end user, various developer teams, the project leadership team, and the documentation team are all discussed in detail with respect to the two implementations. Additionally two sequence diagrams are presented with their corresponding use cases to further elaborate on various sequences of events required to take place in order to successfully run the new AI companion modules.

Finally *Section 4. Final Thoughts* provides an overview of complicated terms, abbreviations and some final concluding remarks about the validity of each approach in the context of ScummVM. Ultimately we have concluded that it makes most sense in the current implementation of ScummVM as a volunteer project. Primarily thanks to the modularity it introduces, which will allow developers with limited time to make the necessary bug fixes and testing without any effect on the other layers of the ScummVM code.

Section 1. Proposals

For our ScummVM enhancement proposal, we decided to make an AI companion, which can provide in-game hints or assistance and guide players through tricky situations. If toggled on It would be overlaid on any game supported by the new feature. We have determined two ideal strategies for this implementation; firstly a ‘black box’ reinforcement learning model, and secondly a ‘white box’ supervised learning based on the code. Both these implementations would

appear very similar at a high level and effectively identical to the user during gameplay. Where they differ is the way in which the AI is trained in 'understanding' the game, so as to provide the most helpful hints possible to the end user (the player).

1.1 Black Box Approach

The theory behind this approach is that it would use reinforcement learning-style training for the AI model. This is completed by training the model by 'watching' millions of hours of expert gameplay and learning through this. Ideally this would be done using the many hours of pre-existing game play content already available for free on the internet. Hiring game players for this learning algorithm is not feasible for a project like ScummVM. In this situation, the goal-states used to train the AI would be generated dynamically through contextual clues during playthrough reviews. Through this training the AI would learn each game to a degree of accuracy in which it would know exactly how to progress the player's game state at any point during a playthrough. Reinforcement learning allows the model to adaptively learn the optimal suggestion-making from receiving feedback (ideally watching the player and determining how quickly its advice helps them achieve the next goal), therefore knowing how to suggest hints to the player when requested. This style of training is long and potentially tedious. It is also prone to human-error, as the AI only learns the game as 'correctly' as the players it is being trained by. However, as discussed in the *Implementation* section, a large benefit to this style of AI training is how it can be implemented modularly as its own layer into the pre-existing ScummVM architecture with minimal disruption to the thoroughly tested pre-existing code base.

1.2 White Box Approach

In order to implement the second learning algorithm for the AI, there are several key steps. First, we need to modify the SCI interpreter to include an AI module where most of our AI scripts will be stored. We also need to modify existing files like sci.cpp and Event.cpp to include and reference the model. Since most of ScummVM and the SCI engine is written in c++ that will be our main language for the AI scripts to ensure consistency. The AI module will be able to parse the compiled script resources used by the SCI games to break down and find key words and functions. Conveniently, ScummVM already has scripts for reading these files, which we can use. From here a decompiler is created that translates SCI game scripts into an Abstract Syntax Tree (AST). By converting the decompiled code into an AST, we can represent the game script structures, like its functions, classes and variables. This will make it easier for the AI to analyze the game. We will also need to modify scripts to include the AI Companion and Hook observations components. The AI Companion is responsible for the AI Logic and operations and will have access to game files to generate hints, The Hook Observation component will be used by the AI to monitor the current game state and events to make sure the gameplay is unmodified. The existing layers that need to be modified are Game engine layer and GUI/Graphics layer. The Game engine layer needs to implement the decompiler and AST generator scripts while the GUI and Graphics layer will be slightly modified to include the settings of the AI and display the AI on the screen. These hooks will allow the AI module to track the game state by seeing global and local variables, player interactions, and object properties. For example, this would be inventory items, players locations, or triggered events. The AI will be able to see when these methods are called or accessed, and when variables are modified, so that the AI is running concurrently with the current game state.

Using the new AI modules and modified existing layers, the AI will analyze the AST combined with the hooks. The AI will understand the game's current state and potential game paths. With the tools provided, it can determine which actions are available or necessary for the player to take. The AI will be able to identify important methods and scripts, what they are connected to in the game's story/puzzle solution or quest completion, all depending on the type of game it is. It will recognize which files and functions are dependent on others and which items are required to unlock new areas, this enables the AI to determine the player's needs.

1.3 Both Implementations - Final Goal

Now that the AI understands the current game state and future steps, the AI needs to provide hints that will be helpful and relevant to the player. For example, if the player needs to find a key for a door, the AI can drop a hint to explore a specific location where the key is found. To convert the AI analysis into player-friendly language we can use template-based generation, where we create predefined sentences and basic structures and the AI fills in the gaps with key words. This way we aren't spending our time and limited resources making a chat bot, we are simply allowing the AI to determine which keywords to add into pre-defined sentences. Additionally, the sprite used for the AI-companion, when toggled on, will be rendered to match the resolution and art style of each game, to better blend in with the game.

Section 2. Implementation

2.1 Effects on Concrete Architecture

In assignment 2, we concluded that ScummVM is best described as a layered architecture style. This layered architecture consists of 5 layers (GUI → Graphics → OSystem API → Engine → Game Files), with bilateral dependencies between each layer. However, one key divergence we noticed is that not only did the layers have bidirectional dependencies between adjacent layers, but they also had bidirectional dependencies with every component, and therefore every layer. This differs a little from what a traditional layered architecture style might be as each layer is interacting with more than just its adjacent layers. Upon adding the required files and components for our enhancement, there are noticeable changes that need to be made to each layer.

2.2 White Box Implementation

The white box approach involves integrating the AI companion deeply into the ScummVM architecture by modifying existing layers. The white box approach allows the AI to access and analyze the game's internal structures directly. This method provides a more accurate and context-sensitive hint system, as the AI can understand the game's logic, state, and possible future actions. To implement this approach effectively, we need to adjust two key layers within the SCI Engine. By modifying these layers, we enable the AI companion to interact with the game engine, access necessary game state information, and provide real-time assistance to the player without altering the original gameplay mechanics.

The OSystem API layer handles platform-specific interactions such as input and output interactions from the hardware, it ensures communications between the hardware and the Game Engine. For example, many of the games on ScummVM were 2D adventure games, which only rely on the keyboard arrow keys and mouse interactions. With our new AI companion, we need to handle additional interactions such as requesting hints, toggling the AI Companion, and

adjusting the hint level. These interactions will be through new button clicks such as binding a letter to toggle the AI or an onscreen button that must be clicked by the mouse to ask for a hint. To ensure the new interactions work properly we need to modify the existing OSSystem API in the SCI engine to recognize and process these new inputs. This will ensure the OSystem API layer works with our new AI Companion. The existing OSystem API has a bidirectional dependency on the Engine layer for keybindings, this will not be changed as we are only focused on adding additional keybindings for the SCI Engine.

The engine layer is responsible for handling the core functionality of the game which includes managing the game states, processing player actions, and ensuring that gameplay logic remains intact. With the introduction of the AI companion, this layer will be extended to support the AI's operations without compromising the game's integrity. A key change to this layer is allowing the AI to access the game state with read-only permissions. The AI will only be able to observe and not modify the game state such as variables and events allowing the AI to generate hints based on the current game state. This ensures the integrity of the gameplay while enabling the AI to generate helpful hints based on the player's progress. The engine layer has bidirectional dependencies with both the GUI layer and the OSystem API layer. The dependency on the OSystem API layer allows the engine to know when the player presses a key to request a hint from the AI companion. Similarly, the engine layer depends on the GUI layer to display the AI generated hints; The game needs to fetch a hint from the engine and present it to the player through the GUI.

2.3 Black Box Implementation

The best way to implement the Black Box AI companion would be to build it as its own layer within the pre-existing ScummVM layered architecture. This will allow the implementation to remain modular, making it easier to toggle. The AI Companion layer would slot into the pre-existing layers we discussed in report 2, in the ScummVM architecture between the Graphics and OSystemAPI layers. When it is not actively being used, all dependencies to and from the AI Companion layer can be 'killed', therefore not altering any pre-existing architecture whenever the AI companion is toggled 'off'. This is especially helpful because it means that when the AI is not in use, the code base is completely unaltered and therefore will run as it always has and will not need to be re-tested. Its modularity will also allow for it to be tested independently.

Within the AI Companion layer, there would exist two new modules to make it function. Firstly, there is the Suggestion Generation Module. This module is responsible for mapping the AI suggestions into human-readable hints in the form of on-screen text. This module will communicate down to the OSystemAPI layer to handle real-time contextual data. Things such as the game states, inventory items, and possible player actions will be communicated back and forth here. This module will be what allows the AI to comprehend what is currently happening in the game play and what will need to happen next for the player to be successful in their current endeavor. After it is initialized and given proper computer resources, the API connections between the AI and GUI and Graphics layers must be established, which also depends on the system configuration and is thus platform-dependent. After these connections are set up, the AI does not need to interact with lower-level layers, as it functions simply by receiving frame information from the Graphics layer API and outputting textual hints through the GUI.
This is where the training of the AI model is put to use.

Secondly, there is the Rendering Module. This module is in place in order to handle everything that is required between the Graphics layer and the AI Companion layer. This includes the new AI companions specific settings in the in-game settings menu, the sprite overlay, and the text boxes. This module also communicates with OSystemAPI to ensure that the text boxes and sprite are generated in a way that aligns with the art style and resolution of the specific game for a more immersive experience. In the context of the Black Box approach, our AI companion will not interfere with the output of the graphics layer. Some potential challenges in our AI accessing graphical data from this layer include the avoidance of latency while visual data is captured for AI processing, managing to leave the rendering pipeline unmodified. A potential solution to this issue might be to utilize asynchronous processing methods for frame analysis, such that even if there is a bit of latency in producing a contextual clue, the fluidity of the game is not negatively affected.

The AI Companion layer will be initialized when the remainder of the ScummVM system is loaded up for gameplay. However, it will only be active if toggled on, as mentioned above, the codebase will function as previously intended if the companion is active. When toggled on, the dependencies described above would become active and the AI Companion layer would begin to function as its own layer, fully immersed within the ScummVM architecture. he Black Box implementation of the companion AI does not need to interact with any layers lower than our OSystemAPI, since the Black Box approach is not given direct access to the in-engine game state or any goal states that might be defined in game files. We believe this to be beneficial to the feasibility of the Black Box approach in terms of ease of integration and minimizing necessary system modification.

For the Black Box approach, our AI companion will have to interact with the GUI layer in such a way that does not lower performance in terms of frame speed or impair the visual presentation by adding too much clutter. GUI components will need to be added in a game that can display the icon representing our AI companion as well as the context-generated clues. In order to implement this approach we would have to define API's between the GUI and AI companion such that hints can be loaded and formatted cleanly. To reiterate, the Black Box approach does not involve the AI directly changing GUI components, only supplying them with information to display. This means that if the API connection is set up well, incoming textual prompts from the AI should not drastically decrease the performance of the GUI component.

2.4 External Interface

GUI Layer:

In this context the GUI will be responsible for the in-game settings menu, providing the player with controls to enable, disable, and change the level of hints that the AI companion gives, displaying the text boxes containing the hints, and the generation of the sprite showing the user that the AI companion is toggled 'on'. Hints would also be levelled, depending on the amount of help the player desires. Easy hints would be very direct and simple to understand, whereas hard hints might be more vague and still require the player to problem-solve.

After receiving a hint it must be displayed on the screen. To not impact the gameplay this must be rendered dynamically and not modify the GUI Layer of the game, this can be done with

the assistance of the Graphics layer. This is another bidirectional dependency that appears upon adding the AI companion, since the graphics layer must be dynamically communicating with the GUI layer at all times while the player is interacting with the AI companion.
The files that will be modified in the sci engine will be `gui-manager.cpp` where we need to add how the file will handle the initialization of the AI. `Options.cpp`, where we will expand the existing settings menu to include settings for the AI helper, `scummvm.ini` will be modified to store the settings the user set for the AI. For the AI user interface itself, we can create an `ai-helper.cpp` to have its own UI files where we can manage its messages and interactions.2

Graphics Layer

Our AI companion may interact with the Graphics layer in order to acquire visual data, such as rendered frames or screenshots in order to deduce the in-game state as well as possible goal states. In order to do so, an API connection between the AI and graphics layers must be initialized so the AI can receive screen information.

The graphics layer is responsible for rendering all visual elements of the game, which includes the AI companion and input/output text boxes. All of those features will appear as an overlay that doesn't interfere with the existing gameplay elements. These new visual components will need to communicate with the rendering backend to manage platform-specific adjustments, such as scaling or positioning on different screen sizes or resolutions. This is another bidirectional dependency as anytime anything needs to be rendered the graphics layer needs to gather the platform specific information from the OSystem API layer to ensure that everything is rendered properly and to the systems specifications.

Section 3. Effects

3.1 Identifying the Major Stakeholders of the Proposed Enhancement:

End-Users (Players)

Players would be the ones whose experience would be affected by the addition of the AI companion. The success of the AI companion feature depends on whether there is a sufficient population of players who would want or use such a feature. Accordingly, the AI companion will be able to be toggled on and off for those who want to experience the game in its original format, without assistance. A high degree of quality assurance testing would be required to develop the balance of the AI, ensuring that it is able to assist the player without spoiling the game.

Project Leadership Team

The project leadership team would want to preserve the nostalgia of the games available on their platform. A hands-on audit/discussion should occur for every game for which the AI is implemented, as this is a novel feature that, when toggled on, would affect a user's gameplay experience.

Documentation Team

The inclusion of the Documentation team as a stakeholder would also be important for both approaches. This division would want to ensure the AI is simple and easily documentable to allow for easy maintenance of this new feature. Documentation would also allow other users to add and improve this new feature.

3.2 Stakeholders That Differ Between the White Box and the Black Box Approach:
Black Box Approach

The developers are a branch of major stakeholders, as they would be responsible for implementing the feature. As discussed in our first report, the developers are split into several teams to delegate responsibilities. This branch would most affect the subsystems developers, as they would be required to maintain the new AI Companion layer required for the Black Box approach. Accordingly, the feature needs to have a scope which is viable within the limits of the resources and expectations of the project. This includes the developer availability, which could result in other features having to be pushed back, how much time the feature would take to implement, whether the feature fits within the specifications and expectations for ScummVM itself (whether an "external" feature such as an AI companion should be a first party integrated feature of ScummVM), and ensuring that the codebase for the new feature has a sufficient degree of modifiability and maintainability, as AI is a rapidly updating field, and new games are constantly added to ScummVM's list of compatible games. The infrastructure for the AI will be set up in a way that it can be automatically trained when a new game is supported by ScummVM. Additionally, the developers would utilize various testing methods such as regression testing to ensure that all previous functionality is unaffected by the new changes.

3.3 White Box approach

Comparatively to the Black Box approach, another important branch of stakeholders for the White Box approach would be the ScummVm Engine division. This new feature would have to be implemented in the codebase for the SCI Engine itself. The Engine division would want to ensure the game remains functional, whilst also ensuring the AI remains compatible with any future game additions. This differs from the Black Box approach as the component would be added to the codebase for the engine, rather than it becoming its own layer in the scummVm architecture. The engine developers would also want to ensure that their additions are compatible with the pre-existing testing tools such as the buildbot.

3.4 Identifying, for each stakeholder, the most important non-functional requirements (NFRs) regarding the enhancement:

3.5 Black Box Approach:

<u>End-User</u>

For the players, one of the most important NFRs would be to ensure that the AI companion does not dramatically decrease the performance of the game. One advantage of ScummVM's target use case is that most titles are older games which can run very efficiently on new hardware barring driver compatibility issues. As such, we would set an NFR performance target of no more than a 5% frame rate decrease while the AI companion is toggled on. Another important NFR for players is usability; Access and use of the AI should be intuitive to not distract the player from their game. A default "generate hint" button would be set for each game, when clicked the AI would spend a short amount of time processing the current game state, and provide a hint to progress. This "single click to generate hint" functionality would ensure the feature is intuitive and does not distract from the game.

<u>Developers</u>

Four crucial NFR's of the new feature for the developers include modifiability, maintainability, portability, and security. The new additions need to be highly modifiable as AI is a constantly changing and updating field, so modifications to the AI would be necessary throughout its lifetime. New games and platforms are constantly added to ScummVM, because of this, the infrastructure of the new feature will support automated training of the AI so that it can be implemented to new games with minimal hands-on human effort. The code needs to also be highly maintainable to allow for the mentioned modifiability. This means having periodic testing phases. Once per day is standard in the industry during deployment, to ensure any new modifications have not affected system functionality or uptime. Portability is another important NFR, as per the functionality of ScummVM, the AI needs to work on a large variety of platforms. Finally, security should also be considered. The AI would be recording the player's screen, actions and game states in real-time, as such, screen recordings should remain local and use industry-standard encryption methods to ensure player data security.

Project Leadership Team

As it pertains to ScummVM's project leadership team, important non-functional requirements include reusability, portability, and accuracy. The AI companion must be able to function cross-platform in order to maintain the portability requirements of ScummVM, which is crucial in keeping ScummVMs large user base. Reusability is a crucial NFR since the AI companion must be able to give hints within the context of many different games. Thus, the companion must be reusable within as many game contexts as possible. Ideally, the companion would interpret or read the game and goal states (depending on the approach) just from watching the game be played or having direct access to the engine game states. The amount of games that the AI companion is functional within is in direct correlation to its overall utility in the context of ScummVM. Finally, the accuracy of the AI companions hints defines its utility in in-game contexts. Functional divergences such as hallucinations will lead to user confusion and distrust in the AI companion and its developers, possibly lowering confidence in the system as a whole. Therefore it is crucial that the hints given by our AI companion are accurate in order to maintain user confidence in the system.

Documentation Team:

The main two NFR that the documentation team would be concerned with is the usability and the Interoperability of the AI Companion. Both these NFRs ensure that the documentation for the code and the instructions on how to use it is simple. Good documentation would ensure the continued success of ScummVM as an open source project that the community can collaborate on.

3.6 White Box approach:

End-User

For the players, like for the Black Box approach, performance would be very important. The addition of the AI should not diminish the gameplay as this could deter users from using ScummVM. However, due to having to go through the code, the hints would take longer to generate and send to the user. Another important NFR for the user would be Accuracy. For this new feature to be successful the hints should be accurate, allowing players to advance when stuck. If the hints were not true to where in the game the user was, it would deter players from using this new feature. Finally, the last important NFR for the player would be usability. The AI

companions should be easy to use. This would allow players to easily get accustomed with this new feature, ensuring the players keep playing

Developers

4 major NFRs would concern the ScummVM engine division team: modifiability, maintainability, portability, and Interoperability. Developers would want the code for the AI to be easily modifiable. This would ensure the code can easily be updated when needed. This comes in handy for an AI as this technology is relatively new and thus constantly changing. The code being Modifiable allows for the AI to be easily kept up to date with the ever-evolving technology. This new feature should also be highly manageable. For the engine division, this means that there should be systems in place that allow testing of the AI feature. In this case, this could be a similar system to the buildbot system described in A1. This would ensure that if any changes are made or any additions are committed to the code base, the AI would not break. Furthermore, the engine division would have to ensure that this new layer is portable from platform to platform. For the feature to be implemented effectively it would have to be able to support all the platforms available on ScummVM. Lastly, developers would be concerned with Interoperability. This means that this new feature would be easily integrated with the larger and more complex ScummVM architecture. In the White Box approach, this would require more work as the implementation for this feature requires changes to the sci engine itself, which could lead to problems down the line.

Project Leadership Team

The project leadership team, for a white box approach, would be concerned with 3 major NFRs, manageability, portability and accuracy. Manageability would be greatly important to the project leadership team. They would want to ensure there are tools in place to ensure this new feature can be tested alongside the rest of the code base. This would allow for quick identification of issues. For the white box approach, the project leadership team would want to ensure that the code can be tested with the Buildbot method. This is ideal as it is highly incorporated with the SCI engine and thus should be tested alongside it to ensure all the components work together as intended. They would also want to ensure that the AI companion would be portable. This would ensure that this new feature can be implemented with every platform ScummVM supports, conserving the goal of ScummVM. Finally, The project leadership team would be concerned about accuracy. The accuracy of the hints would be one of the main factors affecting the use of this new feature. If the hints aren't accurate then it could deter players from using it. For the white box approach, the hints should be accurate as the hints are derived straight from the code. This reduces the chance of the player's game state being mistaken for another.

Documentation Team:

Like for the Black Box approach, the main two NFR that the documentation team would be concerned with is the usability and the Interoperability of the AI Companion. Both these NFRs ensure that the documentation for the code and the instructions on how to use it is simple. Good documentation would ensure the continued success of ScummVM as an open source project that the community can collaborate on.

3.7 Sequence Diagrams:

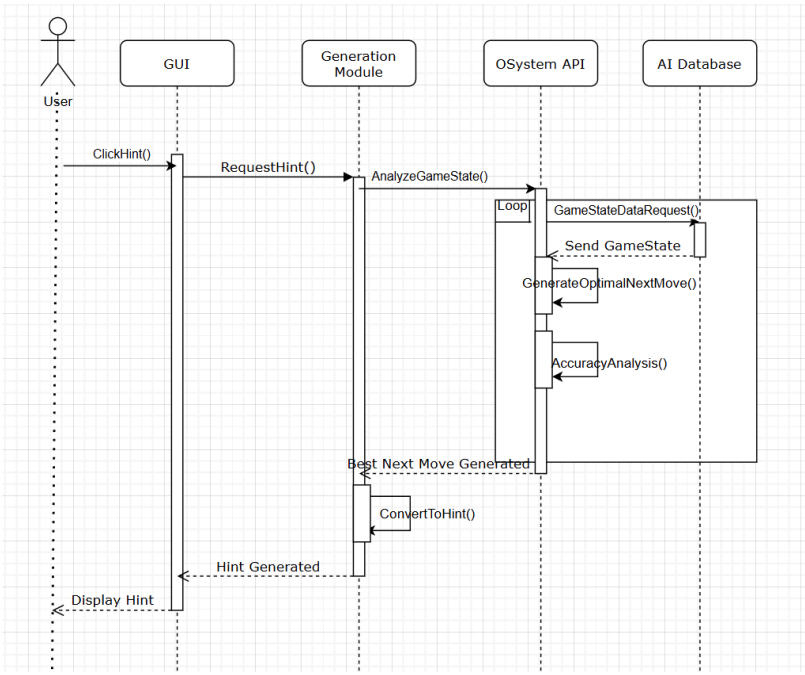

Fig. 2 Sequence diagram showcasing Use case 1

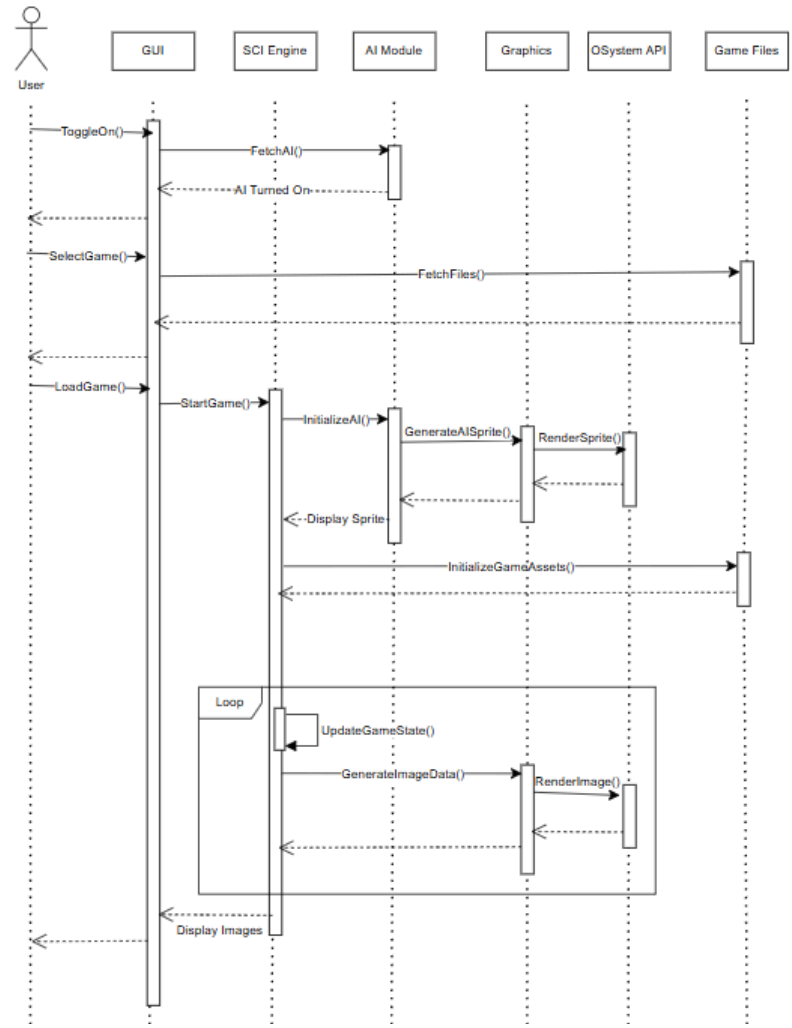

Fig. 3 Sequence diagram showcasing Use case 2

3.8 Use Cases

Use Case 1: User clicks the "Hint" button which generates a hint to help the player progress their GameState

1. [User → GUI] User clicks "Hint" button to trigger a hint request through the ClickHint() function
2. [GUI → Generation Module] The GUI forwards the hint request to the Generation Module and triggers the RequestHint() function
3. [Generation Module → OSystem API] Calls AnalyseGameState() which begins the main logic loop of generating a hint within the OSystem API
4. [OSystem API → AI Database] The OSystem API sends a GameStateDataRequest() to the AI Database in order to compare the current GameState to previous GameStates in similar situations from its database in order to begin generating a best possible next move.
5. [OSystem API → Generation Module] The Osystem API runs GenerateOptimalNextMove() which uses the current GameState and previous GameStates in order to deduce what the next best move would be. The OSystemAPI then

runs AccuracyAnalysis() which reviews the generated move, determining if it is sufficiently accurate in improving the player's GameState. If the generated move passes the AccuracyAnalysis, it is returned to the Generation Module.

6. [Generation Module → GUI] First, the generation Module converts the received next best move into a human legible text prompt, conveying the hint in a way where it does not completely spoil the next move for the player but helps guide them to it. This hint is then returned and displayed through the GUI
7. [GUI → User] The generated hint is displayed for the user through the GUI overlay

Use Case 2: User toggles the AI companion on and loads a game

1. [GUI → AI Module] The user communicates its request to use the AI companion by toggling the corresponding GUI component. AI component responds with its status (on/off)
2. [GUI → Game Files] User selects desired game, triggering FetchFiles() call, retrieving associated game files.
3. [GUI → SCI Engine] The LoadGame() function initializes the SCI engine and loads initial game state
4. [SCI Engine → AI Module] the SCI engine initializes the AI within the game context through the InitializeAI() function. This process involves generating the AI sprite through GenerateAISprite() such that our icon represents an active AI companion.
5. [AI Module → Graphics] The AI sprite data is passed to the Graphics layer by the RenderSprite() function. The sprite can then be displayed, representing the completion of the Ai companion initialization process.
6. [SCI Engine → Graphics] The SCI engine initializes its graphical assets through the function InitializeGameAssets(). This is in preparation for the rendering of the in-game resources.
7. [SCI Engine] Sci Engine enters the game loop, continuously updating game state through the UpdateGameState() function. This loop ensures the game responds to user interactions and other triggered events.
8. [SCI Engine → Graphics] During the game loop, the GenerateImageData() function prepares the visual resources for rendering.
9. [Graphics → GUI] The graphics layer sends visual resources to be rendered in the GUI layer with the RenderImage() function.

Section 4. Final Thoughts

4.1 Data Dictionary:

- **Black box approach:** A method of training our AI by observing inputs, outputs, and the rendered state of the game screen.
- **Reinforcement learning model:** A machine learning technique that trains software to make decisions to achieve the most optimal results.
- **White box approach:** A method of training our AI by directly examining the program's internal code structure, logic and functionality.
- **Abstract Syntax Tree:** An abstract syntax tree is a data structure used to represent the structure of a program or code snippet.
- **Hook observations component:** A way of observing a change in the code

- **Modularity:** The practice of dividing big things into smaller pieces
- **Regression Testing:** A methodology of testing to verify that all previously developed components work as expected after adding a new feature

4.2 Naming Conventions:

- **AI:** Artificial Intelligence
- **SCI:** "Sierra Creative Interpreter" or "Script Code Interpreter"
- **AST:** Abstract Syntax tree
- **GUI:** Graphical User Interface
- **API:** Application Programming Interface
- **UI:** User Interface
- **NFR:** Non Functional Requirement

4.3 Conclusions:

In conclusion, our AI companion can be implemented in two ways. The Black Box approach is implemented through a new layer in the ScummVm architecture, the AI Companion Layer. While the White Box approach is implemented through new components in the SCI engine. After analysing both approaches using the SEI SAAM analysis method, we determined that the better implementation technique in this instance is the Black Box approach. This approach provides a modular implementation. This means that the majority of the ScummVm architecture remains unchanged compared to the White Box implementation that requires major modifications to the SCI engine. In addition, the Black Box approach does not interact with the ScummVM game files, meaning that when turned off it has no effect on the games. This ensures no performance decrease in the event of bugs. Although the accuracy could be decreased slightly, the ease of implementation and the limited interaction with the rest of the code outweigh the cons.

4.4 Lessons Learned

In terms of group communication, we learned that communicating progress and plans at a high frequency is beneficial to the coordination of the project and the members involved. Initially we recognized some inconsistencies between the features of the system stated in the report due to a lack of communication. This led to an unnecessarily intensive period of editing that might've been avoided had there been a more comprehensive conversation about each group member's research direction prior to writing the final report.

Overall, our group learned that when considering enhancements for a preexisting system, it is best to analyze the proposed implementation and weigh the adverse effects and potential hurdles the enhancement may introduce. In this sense, one must analyze the feasibility and utility of the proposed enhancement in order to determine whether or not it is worth dedicating resources towards. In this context, we analyzed the feasibility of our proposed approaches using the SEI SAAM analysis.

4.5 Bibliography

Abstract syntax tree. In *Wikipedia*. Retrieved December 2, 2024, from https://en.wikipedia.org/wiki/Abstract_syntax_tree

Black box. Retrieved December 2, 2024, from https://www.investopedia.com/terms/b/blackbox.asp

Machine learning algorithms for beginners: An easy-to-follow tutorial with Python examples. Medium. Retrieved December 2, 2024, from https://ai.plainenglish.io/machine-learning-algorithms-for-beginners-an-easy-to-follow-tutorial-with-python-examples-5bb68e464a1e

"Non Functional Requirements (NFR) – Quality Attributes" Queen's University, https://onq.queensu.ca/d2l/le/content/959322/viewContent/5711375/View. Accessed Nov 27. 2024.