PROJECT REPORT: TEMPO TAKEDOWN

Candidate Number: 0426

CADBURY SIXTH FORM COLLEGE 20147

Contents

Analysis

Scenario

Rhythm games are a genre of video games where players perform actions in time with musical beats, typically by pressing keys when visual notes reach a specific point on the screen (Gaydos, 2010). Many rhythm games focus on solo play with simple scoring, and players are losing interest because the gameplay can become repetitive. I want to create a rhythm game that combines real-time multiplayer on the same device with competitive mechanics and an ancient Egyptian theme, which makes it more engaging and unique. I want to create a rhythm game that combines real-time multiplayer on the same device with competitive mechanics and an ancient Egyptian theme, which makes it more engaging and unique.

This problem requires a computational solution because it relies on precise timing that humans cannot judge consistently. The program must detect key presses and measure how close they are to the correct moment within milliseconds, something a human referee could never do reliably.

In addition, the game needs to manage multiple processes at once: moving notes down the screen, reading player inputs, updating scores, and handling a health system. Using decomposition, I have split these tasks into separate modules, so each module has one responsibility and can be built and tested independently. Decomposition is the most appropriate approach here because each module, such as input detection and scoring, has no dependency on the internal workings of another, meaning each can be built, tested and replaced independently without breaking the rest of the system. This is not achievable with a non-computational approach as a human cannot simultaneously process collision detection, update scores, and render graphics within a single frame. Abstraction hides this complexity from the player, who only sees the notes and scores, making the game appear simple while the computer handles the complex calculations behind the scenes.

The game also needs to store player data such as accounts, high scores, and unlocked levels across sessions. A computer can handle this automatically and accurately using a JSON file, ensuring the data is consistent and reliable. Overall, the precise timing, multi-tasking requirements, and data management make this problem not only suitable for a computational approach but fundamentally dependent on it. Without computation, the game could not function correctly.

Stakeholders

The first group of stakeholders are casual teenage gamers aged 13–18 who mostly play games socially in their free time and do not usually play rhythm games. They would want a game that is easy to understand, visually appealing, and fun to play with friends. My game is suitable for them because it removes common barriers for new players. For example, the menus are clearly labelled, the how-to-play screen uses simple language, and the easy difficulty ensures

new players do not feel overwhelmed. The local multiplayer mode is particularly well-suited to this group because they can play together on the same device without extra equipment or internet, matching how teenagers typically play games in the same room.

The second group of stakeholders are competitive gamers who enjoy a challenge and want reasons to keep returning to the game. These players will use harder difficulty levels, attempt to beat score targets to unlock new stages, and replay levels to climb the leaderboard. My game is suitable for them because the combo multiplier system rewards consistent skill rather than luck, and the leaderboard records scores under account names, making competition meaningful and long-term. This gives competitive players a clear progression system and goals to work towards. These two groups represent the main users most likely to engage with the game, so their input is essential.

Both groups will also be involved in testing the game. Casual players will perform black box testing, playing the game without knowledge of the code to provide feedback on usability, fairness, and enjoyment. Competitive or more technical players will perform white box testing, checking the scoring system, combo system, and leaderboard for correctness. Feedback from both groups will ensure the game meets the needs of its main users before release.

Secondary Research

First Existing Solution: Friday Night Funkin',

(ninjamuffin99 & Taylor, 2020)

Friday Night Funkin' (FNF) is the first solution I have discovered.

Friday Night Funkin' is a simple arrow-based rhythm game. It was released in 2020 by Funkin' Crew Inc. and released on multiple platforms such as the official itch.io, Snokido, Poki and Newgrounds. The game revolves around a character called Boyfriend who is controlled by the player. Players must defeat a variety of opponents in singing and rapping contests to be able to be with his love interest. Notes scroll vertically in time with the music, and there are 3 difficulties which can be selected- Easy, Normal and Hard- which changes note speed and complexity.

There are two main modes: Story mode- Levels are grouped into weeks with a narrative progression, Freeplay Mode- Players can select any song to play freely.

The gameplay itself includes a performance/ health bar, visual feedback for hits and misses, and a results screen displaying score, max combo, and other stats such as misses. When the health bar reaches zero, the player loses the level which ends the song, and a results screen is displayed where you can also retry the level.

Evidence	Explanation

	 Main Menu	On the main menu of FNF, they have two separate modes: Story Mode and Free play. This allows players to follow a narrative or practice specific levels.
	Story Mode: Level and difficulty selection.	Players can choose between different “Weeks” (level sets) and adjust the difficulty (Easy, Normal, Hard). This improves accessibility and replay ability. I will include a difficulty selector so players of different skill levels can enjoy my game.
	Freeplay Mode- Song selection and difficulty selection.	Each week has its own set of songs. Players can pick a song to play directly. I’ll use a similar structure to let players pick levels or tracks easily, especially if I include multiple songs in my rhythm game.
	Actual Game Mechanics	Gameplay involves timed arrow key presses matching scrolling indicators. There is clear feedback for success/miss and a performance bar at the bottom. I aim to implement scrolling note matching, hit accuracy detection, and a visual performance bar in my own game.
	Losing Mechanic	If you fail to miss too many notes and Boyfriends health goes to 0, then he unfortunately dies, and you are given the option to retry. (His brain is the retry button or you can click enter)
	Winning Mechanic	After each level, FNF shows a results screen with score, max combo, and misses. This gives clear performance feedback and motivates replaying so players can continue to aim for a higher score.

Features it has that I would like to include:

This game is a nice solution to my problem, and it has a few elements that I will like in incorporate into my game such as its art style and mechanics. I would like to include the notes scroll mechanic and feedback system of this game. The player must survive to pass the level which would also make a fun element into my own game that will keep the players engaged. The Note scroll is a widely used mechanic in most rhythm games so I think it will also be well suited for mine if I switch It up a little into my own style to fit the theme. The feedback system will also be used, and I will make feedback to well fit my games aesthetic. For the art style I would also like to have a simple looking but abstract art style to well fit my audience and keeps the game visually clear.

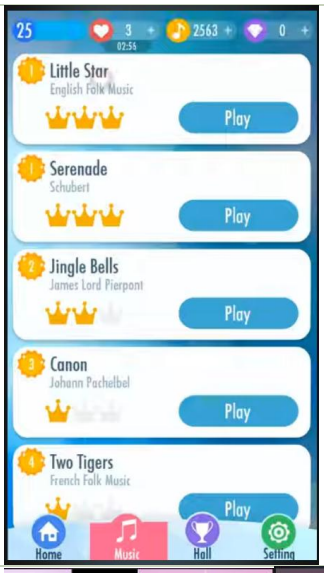
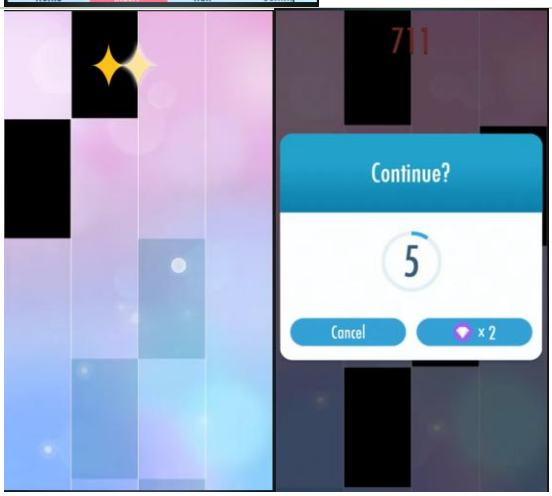
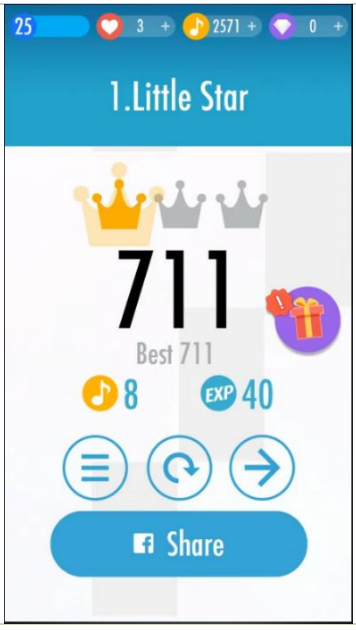
Features that I will avoid:

The menu has a lot of options that may be hard to navigate for some users, and I want my game to have a more simplified menu and settings so it can be accessible to all players. Although I do want to incorporate the art style for inspiration, my game will be simpler to reduce development time and maintain clarity.

Second Solution: Piano Tiles 2

Piano Tiles 2 is a popular **mobile** rhythm game where players tap on black tiles that scroll down the screen in time with music, while avoiding tapping the white tiles/ empty space. The game is very simple in design but requires fast reflexes and good timing. It has a clean and minimalistic interface that focuses on quick reactions rather than complicated mechanics. The gameplay involves tiles appearing in four columns, and the player must tap the correct tile as it reaches the bottom. The speed of the tiles increases as the game progresses, making it more challenging. If a player misses a tile or taps the wrong one, the game ends immediately, encouraging players to try again and improve their score. There is a selection of songs to play, and some may need to be unlocked by beating a certain number of songs and getting coins and gems. There are many variations of the game. Piano Tiles provides immediate feedback with visuals for each tap, giving a satisfying experience. When you press the wrong tile, the tile turns red indicating to the player that they have pressed the wrong tile. It records the player's highest scores and allows for replay ability. The simple mechanics and interface make it very accessible to all ages.

Evidence	Explanation

	This is the games menu where you can pick what song you want to play in the music section. At the top left corner of each song there is a level of difficulty. For example, Little Star has a difficulty of One. Also, on the menu you can see how many crowns you have been awarded. Crowns are awarded every time you reach a certain number of points in one game. There is a button to play whatever song you want. You get coins and Gems after completing songs which you can use to unlock more songs.
	The main as shown consists of the black tiles which you need to press. If you press the empty white spaces or “white tiles” then the game stops and you have lost. After reaching a certain number of points (black tiles pressed) you get stars or crowns as shown at the top. The main goal is to get 3 crowns which you can achieve after getting 3 stars. The second picture is a demonstration of what pops up when you press a white tile. You have the option to end the game or spend 2 gems to continue where you left off so you don’t lose your score.
	After the game ends you are met with a results screen which shows how many coins and experience points you got and your score and crowns. You are given the option to Return to song library, replay, and move on to the next song. You can also share your results online. You may also get rewards for completing a game or getting a certain number of points.
Screenshots taken from (xDeatherXx, 2015)	

Features it has that I would like to include:

I want to adopt the simple, clean gameplay of Piano Tiles, with responsive controls that are easy for players to understand. I will implement a progressive difficulty system where notes gradually increase in speed in every level or difficulty to challenge players and maintain engagement. I also plan to include a high score system to encourage replay ability and a minimalistic interface for easy navigation to avoid clutter and keeping the focus on the actual gameplay to avoid overwhelming users.

Features that I will avoid:

While Piano Tiles is engaging, its main gameplay is very simple, which may not provide enough challenge for experiences players. It is designed primarily as a mobile game, relying on touchscreen input, but my game will only be accessible on laptops or PCs that can run python and use a keyboard. Additionally, the tile-tapping mechanic is straightforward, so I plan to include more difficult and varied note patterns to make the gameplay more challenging and rewarding for players. Piano Tiles doesn't have an immediate feedback system for each note you press so I would like to include that in my game.

Third Solution: Project SEKAI: Colourful Stage! Feat. Hatsune Miku

(Palette, 2020)

Project SEKAI: Colourful Stage! feat. Hatsune Miku is a mobile rhythm game that combines music gameplay with a story and character interaction system. Players tap notes in time with Vocaloid songs, and the game includes many popular characters like Hatsune Miku and others from the *Project SEKAI* universe. Notes appear on a horizontal timeline and come from different directions, so players must react quickly with the right timing and rhythm. The game has multiple difficulty levels (Easy, Normal, Hard, Expert, and Master), which helps players of all skill levels enjoy it. Each song has its own beat map with different patterns depending on the difficulty. There are also multiple input types like tap, flick, hold, and slide, making the gameplay more complex than simpler rhythm games like Piano Tiles. In addition to rhythm gameplay, the game also has a story mode where players interact with characters and build relationships. Players can collect character cards and level them up to improve their performance in songs. The game also has a scoring system, combo tracking, perfect/early/miss accuracy feedback, and a full results screen after each song. It even supports co-op multiplayer mode where up to 5 players can play the same song together and compete or cooperate.

Evidence	Explanation

	This is the games menu where you can navigate through different areas of the game. Here you can play a “show” or continue with the story. In this game you are also able to collect characters which provide you more points. You are also able to see how much in game currency you have. Other features also available such as unlocking and buying new songs and customising your characters.
	When you press on the show option, here you can pick from multiplayer or solo play. You are also able to see prebuilt concerts in the game. This may appeal to other users who enjoy playing the game for the characters and the storyline. There are also other features that are available to be unlocked such as ranked competitive play. This provides users with a variety of different game modes which engages players.
	There is an abundance of songs to choose from with many different categories and genres and all from different groups. You can pick what difficulty you want to play with also other difficulties that you need to unlock by completing the other ones first.
	This is what the actual rhythm gameplay screen looks like. A relatively simple background. There is a health bar at the top for when you miss notes, there is also a progress bar which collects your points to give you a grade at the end. As you click on the notes, you get immediate visual and audio feedback such as “Perfect” when you hit a note perfectly on time with the song and different sounds for each feedback. There is also a combo counter for when you hit more than one note without any errors.
	At the end of the game, either when you run out of health or you complete the song, you are met with a results screen and a character which displays your statistics. Here you can see how many hits and misses you got as well as your highest combo. After every successful song you complete you get rewards which can go towards the characters that you can collect.

Features it has that I would like to include:

I intend to adapt a slight story into my own game. The story will give purpose to the rhythm gameplay and helps players feel more connected to the actual game. I will also integrate a leaderboard as Project Sekai also provides a leaderboard at the end of the multiplayer rounds. This will make the game feel more engaging than playing random levels and provide a clear sense of progression.

I plan to include the varied note patterns, such as holds and slides (like Friday Night Funkin') to make gameplay more challenging and interesting. This is because a variety of note types increases engagement and allows both casual and more experiences players to enjoy the game. In addition to the core gameplay mechanics, I plan to include a short results screen that will display stats such as score, max combo, misses and a simple rating, inspired by Project SEKAI. The game will also feature progressive challenge levels. Optional local multiplayer will also allow two players to play together or compete together on the same keyboard adding a social element. I also do like the level unlock system where a level will be unlocked under certain conditions. This helps to keep players engaged and motivated to continue playing.

Features I will avoid:

Although Project SEKAI's gacha system and character card upgrading are popular on mobile, I do not plan to include these features. They would complicate the design and distract from the rhythm-based gameplay. My aim is to make a skill-based rhythm game where success depends only on the players performance and not randomised systems. This could make the game unfair as players aren't guaranteed anything in a gacha system. I won't also include heavy story-driven interactions or complex character relationship systems since my aim is to keep the story light and optional so that players who prefer pure gameplay aren't overwhelmed. Although the game is very visually appealing, I do not plan to replicate that level of complexity. A simpler, abstract art style will keep the focus on the notes and rhythm while also making the project more achievable within my scope and time limits. Music is not included to keep the project manageable and to focus on the required programming skills such as timing, input handling, and scoring

Summary of Research Findings:

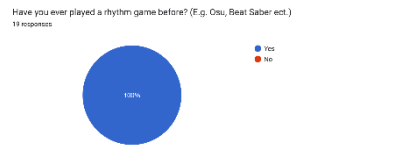
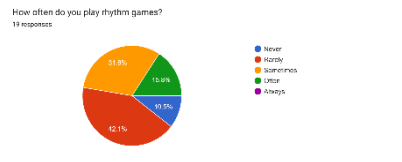
After looking at these three existing solutions, I have been able to identify what works well in rhythm games and what I want to do differently in my own game. From Friday Night Funkin' I could see how a health bar, visual hit feedback, and a results screen make the gameplay feel fair and satisfying, so I plan to include all of these. One thing FNF is missing though is the ability for two players to play on the same device with one keyboard, which is something I want to add to my game through a local multiplayer mode. From Piano Tiles I could see how keeping the interface clean and simple helps players focus on the actual gameplay without getting

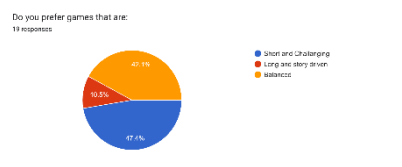
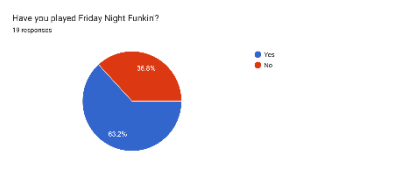
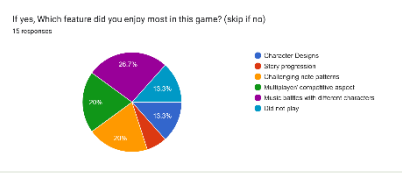
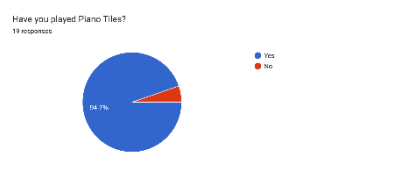
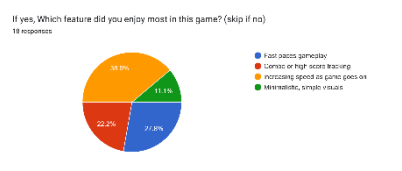
confused, and I want to apply that same idea to my menus and gameplay screen. However Piano Tiles has no user account system or leaderboard, meaning players have no real way to compare their scores with others, which is something I think makes a game much more engaging and replayable, so I will be including both in my game. From Project SEKAI I could see how a combo counter, accuracy feedback, and a detailed results screen give players a real incentive to improve and come back to replay levels, all of which I plan to include. That said, Project SEKAI is a very large and complex game with gacha systems and online multiplayer that would be far beyond what I could realistically build on my own, so I will be keeping things much simpler.

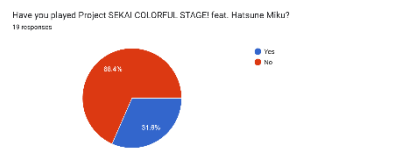
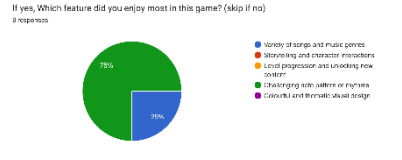
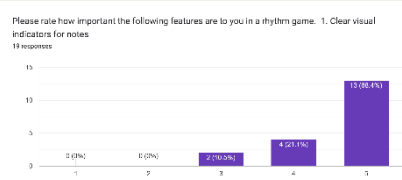
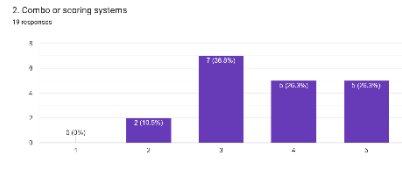
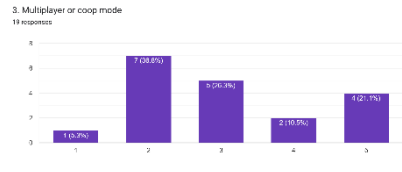
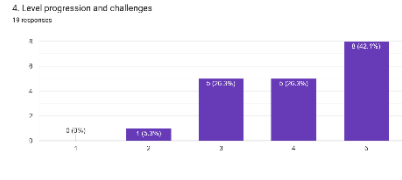
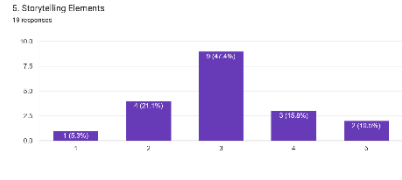
Overall, my game will take the core mechanics that work well across all three solutions, such as scrolling notes, hit feedback, a scoring system, and a results screen, while keeping the design simple and manageable. The main things that will make my game stand out from the existing solutions are the local multiplayer mode and the user account and leaderboard system, which none of the three solutions offer together in the same way.

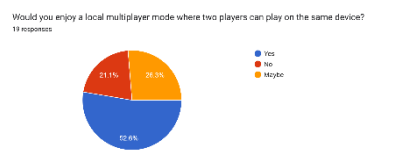
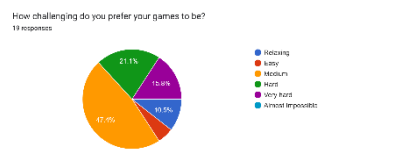
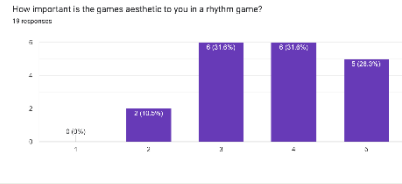
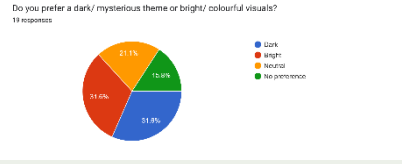
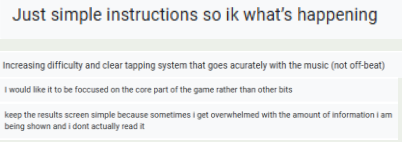
Primary Research: Questionnaire analysis

To discover which features are desired by my user group and develop requirements, I will conduct a questionnaire on my specific demographic. The responses will help identify which game mechanics, difficulty levels and features are most important to the users. This data will then inform my success criteria and design decisions. This will be done on Google forms. I have gone and found people of my demographic (teenagers and gamers) and I sent a link to them via Discord where they have all completed the form. I have asked 19 people.

Question	Result	Conclusion
Have you ever played a rhythm game before? (E.g. Osu, Beat Saber etc.)		This shows that my target audience already has experience with rhythm games. I can design my game to have a more simplified tutorial as my audience already has experience.
How often do you play rhythm games?		Majority have answered Rarely or Sometimes. This suggests players are not deeply habituated to rhythm games, meaning the game must deliver immediate engagement within the first session rather than relying on familiarity with the genre to retain players.

Do you prefer games that are: • Short and Challenging • Long and story driven • Balanced		My results here show that on average more people tend to prefer shorter and more challenging games with a decent story line to keep it balanced. This directly informed the decision to cap each level at around 90 seconds with a score threshold rather than an open-ended playtime, balancing the challenge preference of the majority with light narrative for those who wanted story.
Have you played Friday Night Funkin'?		Many people have played FNF so I can borrow similar mechanics as they will be easily understood by experienced players.
If yes, which feature did you enjoy most in this game?		There is a variety of different results here, but the most popular result is the “Music battles with different characters” which is a story feature. Challenging notes and multiplayer features are tied so I should also include them in my own solution.
Have you played Piano Tiles?		Since all but one person has played piano tiles, I can include similar mechanics that may be familiar to the experienced players.
If yes, which feature did you enjoy most in this game?		Almost 39% have agreed that they enjoyed the fact that notes increase in speed as they continue to play the game. This shows that my audience likes a challenge, and I should also include a mechanic like this to my own game.

Have you played Project SEKAI COLORFUL STAGE! feat. Hatsune Miku?		Majority of players are familiar with the story driven rhythm game so including level progression and a light narrative in my game will appeal to this audience.
If yes, which feature did you enjoy most in this game?		Players overwhelmingly prefer challenging note patterns therefore my game will prioritise skill-based, engaging note patterns as the core mechanic while also including multiple tracks to maintain a variety.
Please rate how important the following features are to you in a rhythm game. 1. Clear visual indicators for notes		13/19 players consider clear visual indicators essential. My game will include highly visible note cues to ensure players can accurately time their inputs improving gameplay enjoyment.
Please rate how important the following features are to you in a rhythm game. 2. Combo or scoring systems		There is an average rating of 3.68/5 for combo scoring so my game will include one to reward accuracy and skill. Since the ratings are more positive than this will be a secondary focus on the game.
Please rate how important the following features are to you in a rhythm game. 3. Multiplayer or coop mode		There is an average rating of 3.05/5. I may include this as an optional feature so it can appeal to all kinds of players.
Please rate how important the following features are to you in a rhythm game. 4. Level progression and challenges		Players consider challenges as important so my game will include progressively harder levels with unique patterns and obstacles to maintain engagement.
Please rate how important the following features are to you in a rhythm game. 5. Storytelling Elements		Moderate interest in storytelling. My game will include a light narrative to enhance immersion. The main gameplay will remain rhythm based, and challenge focused

Would you enjoy a local multiplayer mode where two players can play on the same device?		Over half of players are interested in local multiplier so I will include this as an optional feature as stated before.
How challenging do you prefer your games to be?		Players predominantly prefer a medium level with some interest in harder challenges. Therefore, my game will start at a neutral level with options for a harder level. This is to meet the demand for the other levels as well.
How important is the games aesthetic to you in a rhythm game?		Overall, the aesthetic of the game does seem to be important to my audience, so I need to make sure that the game is not too basic in design but also not too much where I won't have enough time to do anything else.
Do you prefer a dark/ mysterious theme or bright/ colourful visuals?		Players prefer a darker visual theme so my game will use a dark tomb spired aesthetic while keeping some natural elements.
Are there any features that you would like to see in a rhythm game? *open ended and optional question*		I got a couple of responses on some features they would like for me to focus on.

Features

Features of the Proposed Solution

User Sign up/ Log in

The game will require players to create an account or log in before playing. This is an essential feature because without it, there would be no way to reliably save individual player progress or display personalised scores on the leaderboard. Storing user data in a JSON file means each player's scores and unlocked levels can be retrieved every time they return to the game, which directly supports the competitive and replayable nature of the game that my questionnaire showed was important to my target audience.

Exit Confirmation

Prevents players from accidentally quitting and losing progress. Improves usability and ensures both casual and competitive players don't get frustrated if they accidentally hit the exit button.

How to Play

Provides instructions for controls, note timing, and gameplay mechanics. Supports accessibility for new players, ensuring they can quickly understand the game and start playing confidently

Settings and Key Bindings

Allows players to adjust keys, audio, and visual preferences to suit their playstyle and hardware. Enhances comfort, accessibility, and overall user experience.

Game Timer

Ensures that notes appear and disappear precisely in time with the notes. Essential for accurate scoring, feedback, and rhythm gameplay.

Story Screens

Adds light narrative context to levels without overwhelming players. Provides motivation and immersion while keeping the main focus on rhythm-based gameplay and challenge.

Tomb/ Setting

The game will be set in an ancient Egyptian theme with visuals styled around tombs and temples. This feature is important because my questionnaire showed that the majority of players preferred a dark and mysterious visual theme, and having a consistent visual identity makes the game feel more unique and immersive compared to generic rhythm games. It also helps differentiate Tempo Takedown from the existing solutions I researched.

Notes

The core gameplay will involve scrolling notes that players must hit by pressing the correct key as each note reaches a hit zone at the bottom of the screen. This feature is essential because it is the fundamental mechanic that makes the game a rhythm game without it there is no gameplay. The scrolling note mechanic was chosen because it provides clear visual cues that are easy to understand, which suits both casual players who are new to rhythm games and competitive players who want to master high speed sequences. Hold notes will also be included to add variety and increase the skill ceiling.

Levels and Level Unlock System

The game will feature three difficulty levels, Easy, Medium, and Hard, with harder levels unlocking only after the previous one has been completed. This feature is essential because it gives the game a sense of progression and ensures players are not immediately overwhelmed. My questionnaire showed that the majority of players preferred a medium level of challenge, so starting on Easy and progressing upward means the game is accessible to beginners while still

providing a real challenge for more experienced players. Without this structure the game would feel flat and players would have little reason to keep coming back.

Feedback system

A visual feedback system will display indicators such as “Splendid!” “Wow” and “Close!” when notes are hit. This immediate feedback helps players understand their performance and adjust their timing, providing casual players clear guidance and helps competitive players improve their accuracy and maximise their score. Clear feedback systems ensure that players can see clear results from their actions.

Results Screen

After each level a results screen will display the player's score, accuracy, maximum combo, and number of misses. This feature is essential because it gives players a clear summary of their performance and motivates them to replay levels to improve. Without a results screen players would have no way of knowing how well they did or what to work on.

Local Multiplayer

The game will allow two players to play and/or compete on the same device. This feature increases engagement by introducing a social and competitive element. It is suitable for players who want to play with their friends for fun and to also enjoy head-to-head challenges. Local multiplayer adds variety to the gameplay and increases replay value. This is suited for my target audience as teenagers love to hang out with their friends and play games- this feature allows both simultaneously which is great for them.

Leaderboard

The game will include a leaderboard that displays the highest scores across all registered users for each difficulty level. This feature is essential because it adds a long term competitive goal for players, giving them a reason to keep returning to the game even after they have completed all the levels. Without the leaderboard the user account system would have little purpose beyond saving progress.

Together, these features form a complete, computationally driven rhythm game that balances accessibility, challenge, and engagement, ensuring the game functions smoothly and meets player needs.

Limitations

One limitation is that the visual design of the game will be simple rather than highly detailed or animated. This is because creating complex graphics and animations in Pygame requires significant additional development time and design skills that are beyond the scope of this solo project. Unlike professional game engines such as Unity which have built in animation tools, Pygame requires all visuals to be manually coded or imported as image files, meaning anything beyond simple shapes and text would take a disproportionate amount of time to implement.

The focus will therefore be on ensuring the gameplay itself is functional, fair, and responsive rather than visually complex.

The number of levels in the game will also be limited to three difficulty stages. Each level requires its own note pattern data, spawn timing, difficulty balancing, and thorough testing before it can be considered complete. Adding a large number of levels would therefore significantly increase development time without adding new programming concepts to the project. Three levels are enough to demonstrate clear progression and challenge while keeping the project within a manageable scope.

Music will not be included in the game. Synchronising notes to a live audio track would require either a beat detection algorithm to automatically map notes to the beat, or a hand authored timing file that maps every note to a precise millisecond timestamp for a specific song. Both approaches are complex to implement correctly and would also raise copyright issues if commercial music were used. Removing music allows the project to focus on the core programming challenges of timing, input handling, and scoring, which are the most important skills being demonstrated in this project. Although removing music weakens the rhythm game identity by removing the audio cue players would naturally time their inputs against, this trade-off is justified because implementing beat-synchronised note charts is a distinct engineering problem that would consume most of the development time without demonstrating any additional programming concepts beyond what the timing system already covers.

Although the game includes local multiplayer, it will not support online multiplayer, meaning players cannot compete with friends over the internet. Implementing online multiplayer would require networking code, a dedicated server to handle connections between machines, and real time data synchronisation to ensure both players see the same game state simultaneously. These are all significantly advanced programming concepts that are well beyond the scope of a solo development project at this level. Local multiplayer on the same keyboard still delivers the competitive social element that my stakeholders wanted without any of this added complexity. The trade-off is that players cannot compete with friends on different devices, which reduces long-term replayability, but this is acceptable given that the local leaderboard still provides a competitive goal for players sharing the same machine.

The game will also only support keyboard input rather than controllers or touchscreen devices. Pygame's built in keyboard event handling is straightforward to implement and test, whereas adding controller support would require additional libraries such as pygame.joystick and significant extra input mapping work. Since my target audience will be playing on a PC or laptop, keyboard only input is appropriate and sufficient for the intended use case.

Finally, the narrative will remain light throughout the game. Advanced story systems with branching dialogue and character interactions would require complex state management and significantly more development time. The story will therefore be limited to short introductory

text screens before each level, which is enough to provide context and motivation for the player without taking development time away from the core rhythm gameplay.

Requirements

My project will be developed in Python using the Pygame library. I chose Python over other languages such as C# or JavaScript because it is the language I am most experienced with, meaning I can focus on building the game rather than learning new syntax. I considered using Unity as an alternative; however, Unity requires knowledge of C# and a significantly more complex development environment which would not be appropriate for a project of this scope. Web based development using JavaScript and HTML5 was also considered, but this would require additional knowledge of web technologies and would make saving user data more complicated. Pygame was chosen as the library because it provides all the tools needed for a 2D game, including a game loop, keyboard input handling, and graphics rendering, while being simple enough for a solo developer to use effectively. As Pygame is multiplatform, the game will also run on both Windows and Mac operating systems, making it accessible to my target audience.

Functional Requirement	Justification
User Set Up	A login and sign up system is required so that each player's scores and unlocked levels can be saved individually to a JSON file and retrieved when they return. Without this, all progress would be lost every time the game was closed and the leaderboard would have no way of displaying individual player scores.
Menu	A main menu is required to give players a clear starting point from which they can access all parts of the game. Without a structured menu, players would have no organised way to navigate between playing, viewing instructions, changing settings, or accessing the leaderboard.
Notes Scrolling Mechanic	Scrolling notes are the core mechanic of the game and are required because without them there is no rhythm gameplay. Notes will scroll down four lanes at a speed determined by the difficulty level, and players must press the corresponding key as each note reaches the hit zone at the bottom of the screen.
Feedback System	Immediate visual feedback is required because in a rhythm game players need to know instantly whether their timing was correct in order to adjust and improve. The system will display "Splendid!", "Wow!", or "Close!" depending on how close the key

	press was to the hit zone, with a "Miss" displayed if the note passes without being hit.
Results Screen	A results screen is required after each level so players can review their performance. It will display score, maximum combo, number of Splendid, Wow, and Miss hits, and whether the score threshold was met, giving players clear and measurable feedback to work towards improving.
Level Challenges	Three difficulty levels; Easy, Medium, and Hard, are required to ensure the game is accessible to beginners while still providing a genuine challenge for more experienced players. Note speed and spawn rate increase with each difficulty, and score thresholds must be met to pass each level.
Leaderboard	Since each user is connected to an account, their high scores are saved and the users with the highest scores are going to be displayed on a leaderboard. This makes the game a little more competitive and will encourage users to continue playing the game.
Local Multiplayer	Local multiplayer is required because my research identified it as a missing feature in all three existing solutions and my questionnaire confirmed over half of respondents wanted it. Two players will play simultaneously on the same keyboard using ASDF for Player 1 and JKLS for Player 2, with fully independent scoring, health, and feedback for each player.
Adjustable Key Bindings	Adjustable key bindings are required because different players have different preferences for hand position and comfort. The settings screen will allow both Player 1 and Player 2 to reassign any of their four lane keys to any key of their choice.
Tomb Theme	A consistent ancient Egyptian visual theme is required be the majority of players preferred a dark and mysterious a helps differentiate the game from the existing solutions r

As Pygame is multiplatform, the game can run on multiple operating systems making it more accessible to my stakeholders.

Software Requirement	Justification

Pygame Library	Pygame provides all the tools required to build this game, keyboard event handling, graphics rendering, and a clock module for managing note timing. It was chosen over Unity because it runs directly within Python and requires no additional software or engine knowledge.
64-bit python 3.9 or newer (3.14)	Python 3.9 or above is required to guarantee compatibility with the current stable version of Pygame. Older versions of Python may not support all syntax or library features used in the code.
Operating systems: **Windows 7 or later** **Mac OS X 10.11 or later**	These are the minimum operating system versions that s ensuring the game can run on the range of devices used b additional configuration.

Since my game isn't graphics focussed, it doesn't require the most high-end hardware, but it does require a reasonably modern computer to ensure smooth gameplay.

Hardware Requirements	Justification
Keyboard	The main method of control for the player as it is required for note inputs. My game will mainly utilise WASD and arrow Keys so that two players can play on the same keyboard for local multiplayer.
2GHz Processor	A processor of at least 2GHz is required to maintain a consistent 60 frames per second game loop. A slower processor would cause frame drops which would make note timing inaccurate and the gameplay unfair.
4GB RAM	This ensures that sounds and images load smoothly, preventing lag during gameplay. Python uses a lot of memory therefore having enough ram helps run the game smoothly, satisfying my user.
HDD Space (100MB minimum)	This is the minimum storage space required for Python, Pygame, the game files, and the file that stores all player account data and scores.
Monitor with minimum 720p resolution	A 720p monitor is required so that all four note lanes, the hit zone, score displays, and feedback text are clearly visible. At lower resolutions these elements may overlap or become too small to read accurately during gameplay.

Success Criteria

Criteria	Measurement	Justification
1. User Set up	A login and sign up screen must appear on startup. Error messages must display for incorrect or empty credentials.	This measurement was chosen because it can be objectively tested by entering correct credentials, wrong credentials, and leaving fields empty, and checking the correct response appears each time.
2. Functional menu	A menu must appear after login with buttons for Play, How to Play, Settings, Leaderboard, Sign Out, and Exit. Each button must navigate to the correct screen.	This measurement was chosen because each button can be individually clicked and verified to confirm it leads to the correct screen, making it fully objective and testable.
3. Easy to read Instructions	A how to play screen must display the controls, note types, and scoring system in clear language. ESC must return to the menu.	This measurement was chosen because it can be verified by a first time user reading the screen and confirming they understood how to play without additional help, directly testing accessibility.
4. Adjustable Key Bindings	Players must be able to click any lane label in the settings screen and press a new key to reassign it. Changes must take effect immediately in gameplay.	This measurement was chosen because it can be tested by reassigning a key and then playing the game to confirm the new key registers correctly, making it directly verifiable.
5. Notes scrolling Mechanic	Notes must scroll smoothly down the screen at the correct speed for the selected difficulty with no stuttering or lag. Notes must disappear when hit or when they pass the hit zone.	This measurement was chosen because smooth scrolling at the correct speed can be verified visually and by checking the frame rate is constant, which is essential for fair rhythm gameplay.
6. Long Note Mechanic	Long notes must require the player to hold the key down until the note passes. Releasing early must reset the combo and display a "Released early!" message. Successfully holding must award bonus points.	Game must include hold notes that require sustained key presses. Early release must reset combo and display a message. Successful hold must award bonus points. Measurable by deliberately releasing early and confirming the message and combo reset appear, then holding fully and

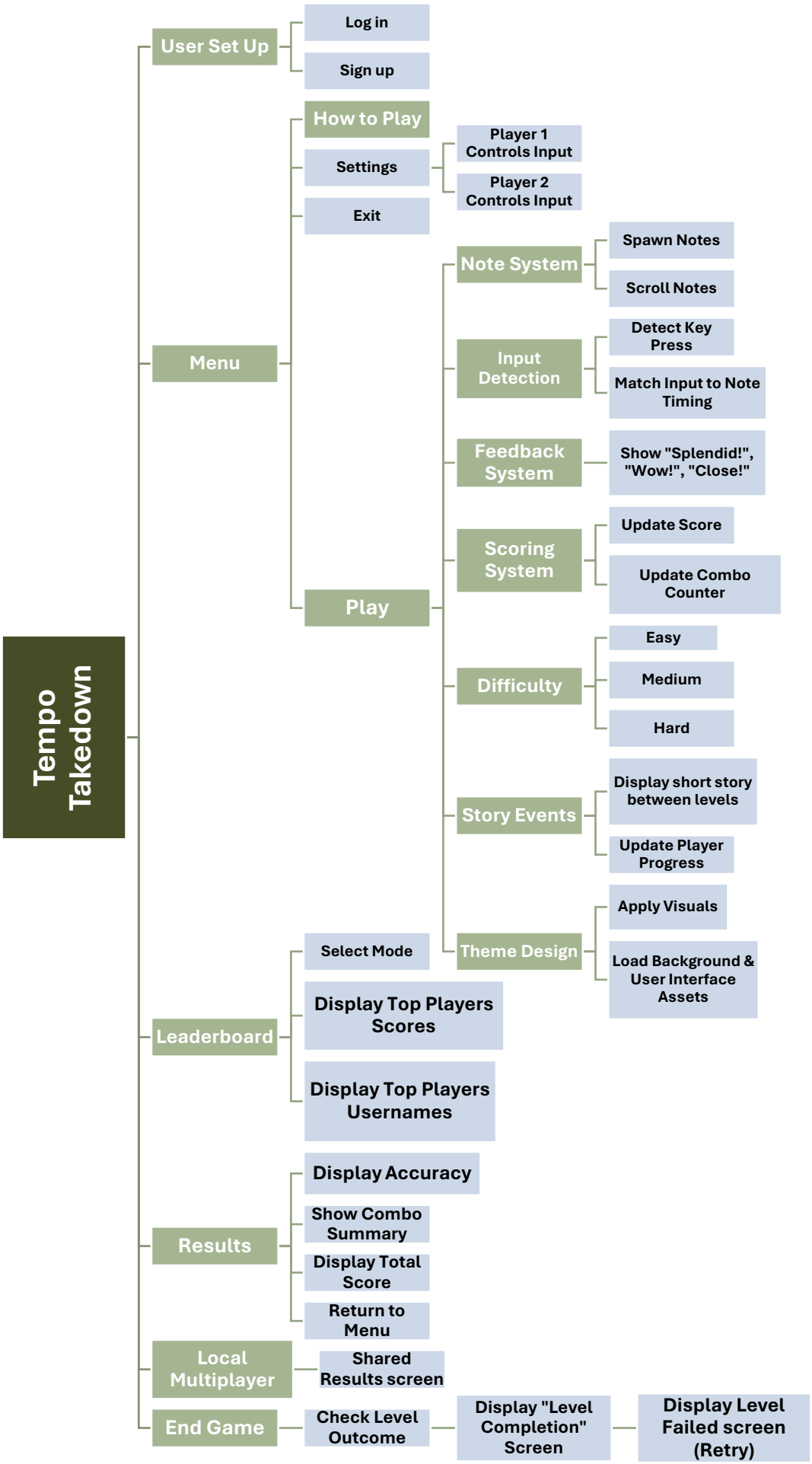
		confirming bonus points are awarded.
7. Feedback System	A feedback message must appear immediately after each key press showing Splendid, Wow, Close, or Miss with the correct colour.	This measurement was chosen because it can be tested by pressing keys at different distances from the hit zone and checking the correct message and colour appear each time, directly verifying hit detection accuracy.
8. Level challenges	The solution should have 3 stages of difficulty that tests players skills at different levels that can be unlocked.	These specific values were chosen because they are directly implemented in the code and can be objectively verified by playing each level and checking note speed and win conditions match the expected values.
9. Local Multiplayer	Two players must be able to play simultaneously using different keys with fully independent scoring, health, and feedback. No input from one player should affect the other.	This measurement was chosen because it can be tested by having both players press keys simultaneously and checking that inputs are registered independently, directly verifying there is no interference between players.
10. Tomb Theme Design	An ancient Egyptian theme must be consistently applied across all screens including menus, gameplay, story screens, and results screens.	This measurement was chosen because it can be verified by checking every screen of the game and confirming the theme is present and consistent throughout, making it objectively assessable.
11. Storytelling Element	A story introduction must play before the game and a unique story screen must appear before each of the three difficulty levels.	This measurement was chosen because it can be verified by playing through all three levels and checking that a different story screen appears before each one, directly testing narrative progression.
12. Notes Combo	Combo counters must update accurately after each successful note and unsuccessful note. Higher combos must award bonus points.	This measurement was chosen because the combo counter and multiplier values are specific and directly testable- the multiplier can be verified by maintaining a known combo count and checking that the

		points awarded match the expected multiplied value each time.
13. Health System	Players must have a visible health that decreases by 1 for each missed note or incorrect input.	This specific starting value and decrement rate were chosen because they are directly testable, health can be verified by missing a set number of notes and checking the displayed value decrements correctly each time.
14. Level Win Display	A victory screen with story text and stats must appear when the score threshold is met and the player survives the 90 second timer.	This measurement was chosen because it provides a clear and specific trigger that can be tested by reaching the exact score threshold and surviving the timer, directly verifying the win condition logic.
15. Level Loss Display	A loss screen with Retry and Return to Menu options must appear when health reaches 0.	This measurement was chosen because it can be tested by deliberately missing notes until health reaches exactly 0 and checking the correct screen appears with both options functioning correctly.
16. Leaderboard	Top scores for each difficulty must be displayed with usernames in descending order. Users with a score of 0 must not appear.	This measurement was chosen because the ordering and filtering can be objectively verified by submitting known scores and checking they appear in the correct position with the correct username.
17. Results screen	After each level a results screen must display Player 1 score, max combo, Splendid count, Wow count, and Miss count. In multiplayer both players stats must be shown alongside the combined score.	Each individual stat can be independently verified by playing a level with a known number of each hit type and checking the displayed values match exactly, making this fully objective.
18. End game Failure	When health reaches 0 the game must end immediately and display the loss screen. The Retry option must restart the level and Return to Menu must go back to the main menu.	This measurement was chosen because it provides two specific testable outcomes, both the retry and menu navigation can be individually verified to confirm they function correctly after a loss.

Design

Decomposition

The hierarchy diagram above breaks Tempo Takedown down into independent modules, each with a single clearly defined responsibility. I chose this structure because the game combines several distinct that need to be kept separate so each one can be built and tested in isolation without affecting the rest of the program. This also made it straightforward to plan a logical development order, since foundational modules like User Authentication can be completed and verified before the gameplay modules that depend on them are written.


Tempo Takedown
User Set Up
Log in
Sign up
Menu
How to Play
Settings
Player 1 Controls Input
Player 2 Controls Input
Exit
Play
Note System
Spawn Notes
Scroll Notes
Input Detection
Detect Key Press
Match Input to Note Timing
Feedback System
Show "Splendid!", "Wow!", "Close!"
Scoring System
Update Score
Update Combo Counter
Difficulty
Easy
Medium
Hard
Story Events
Display short story between levels
Update Player Progress
Theme Design
Apply Visuals
Load Background & User Interface Assets
Leaderboard
Select Mode
Display Top Players Scores
Display Top Players Usernames
Results
Display Accuracy
Show Combo Summary
Display Total Score
Return to Menu
Local Multiplayer
Shared Results screen
End Game
Check Level Outcome
Display "Level Completion" Screen
Display Level Failed screen (Retry)

Process	Explanation	Justification
User Set Up	This page will show up when players load the game and will allow players to set up a username and password to store their in-game data after playing.	This allows players to log off and log back on to carry on playing where they left of and be able to be placed on the leaderboard. This is a crucial step as if players were not asked to sign up, they would lose all data and progress in the game.
Menu	The menu is the next screen the user interacts with and is important as this is what will grab the user's attention immediately. It includes options such as "Start Game", "How to Play", "Settings" and "Exit". It acts as the main navigation hub for the game.	This process is Necessary to provide an organised interface for players. It helps users easily access different parts of the game. By designing it as a separate process, I can focus on layout, interactivity and smooth transitions without interfering with gameplay.
How to Play	This screen gives instructions about the games objective, controls, and scoring system. It teaches players how to play before they start.	Not all users may have played a rhythm game before or understand the game immediately, so a clear guide ensures accessibility and reduces confusion.
Play	This is the main gameplay process where the rhythm-based interaction occurs. Notes fall at timed intervals, and players must press the correct keys in time to score points.	This process contains the games core functionality. Isolating it allows me to focus on precision and rhythm timing without being distracted by unrelated systems.
Difficulty Selection	Players can choose between levels. Each mode increases the speed and complexity of note combinations.	Different levels make the game enjoyable for a wider audience. Separating this process makes it easier to balance and calibrate parameters later.
Notes System	The notes are the main gameplay process. Notes fall at consistent intervals based on level difficulty, and players must press the correct keys as the notes reach the hit zone.	It involves implementing a timing-based system using pythons clock module within Pygame to ensure frame updates and note movements are synchronised to the game's timing system. By treating this as its own process, I can easily adjust note patterns, speeds, and difficulty settings independently from other systems and reduce lag ensuing accurate timing which is a key computational concern in a rhythm-based game.
Input Detection	This process reads the players keyboard inputs and checks whether the correct key was pressed at the right time.	Treating this as a single module will also help me implement its features easily since I can manage the algorithm that detects the key inputs in time. This also allows the algorithm to check if the player has a high enough score to progress into the next level.

Feedback System	This process displays feedback text such as “Splendid!”, “Wow!”, or “Close!” depending on how accurately the player hits the notes.	Linked back to the Feedback system requirement, this process provides immediate visual response through text rendering. Immediate feedback helps players improve rhythm engagement. By keeping this process independent, I can modify visuals without disrupting the core gameplay.
Scoring system	Calculating the players score in real time based on timing accuracy and combos. It stores numerical data on total hits, misses, and accuracy percentage.	This contributes to the Feedback System and Results Screen Requirement. This makes it possible to use python data structures like dictionaries to store and update player performance dynamically.
Story Events	When the game is first launched, a short text introduction sets the story tone for the game.	This aligns with my final functional requirement “The game should open with a short storytelling sequence”. Making it its own process ensures clear flow control between the story and the menu.
Themed Design	The visual design follows a tomb-inspired theme as it provides visual atmosphere and enhances player immersion.	Giving the game a unique identity allows me to handle graphical assets through image functions.
Results	After a game, the results screen shows the players final score, accuracy, combo streak and number of misses. It also provides options to retry or return to the menu.	By isolating it, I can handle score calculations and user interface rendering separately, preventing interference with the game loop. It enables data persistence, which demonstrates structures data management.
Leaderboard	After gaining a high score, players may be put on a leaderboard if their score is within the top players. This provides a reward system and a feedback system.	By having a reward system that outputs feedback to the player which displays their progress, more people will be inclined to carry on playing to be placed high on the leaderboard. By having this system, players will be playing more and may invite their friends to compete for the highest position.
Local Multiplayer	Two players can compete on the same screen using different key mappings. The game runs the same note sequence for both players but calculates their scores separately.	Designing this as a distinct process allows simultaneous input detection using even driven programming principles in Pygame. It also enables me to apply object-oriented design as each player can be represented as an independent object with properties such as score, combo, and input keys.
End Game	At the end of a level or when the time reaches zero, the game determines whether the player completed it	The end game process manages game progression and provides feedback on performance.

	successfully or failed. Depending on the results a “Level Complete” or “Retry” screen appears.	

Usability Features

The game interface has been designed to make it intuitive and easy for players to use. Key usability features have been included to ensure that navigation, gameplay, and settings adjustments are simple and clear. These features aim to reduce confusion and allow the user to focus on the game itself.

Single-Letter Menu Choices:

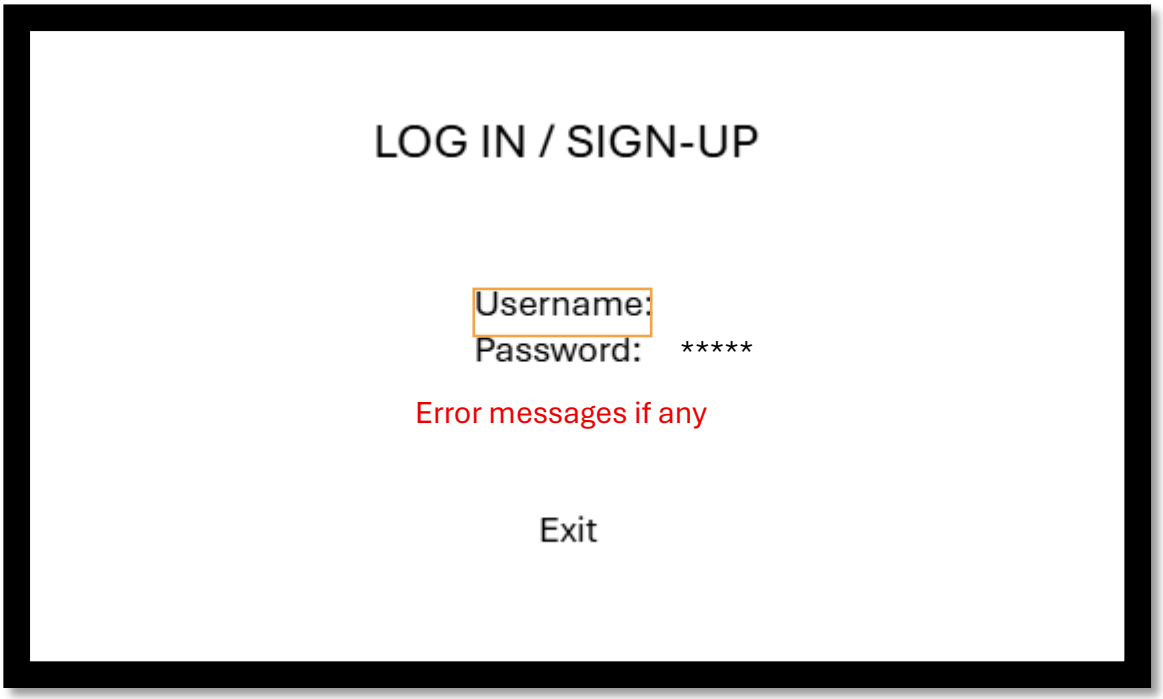
- Menus can be navigated and exited using single key presses such as ESC to pause or return.
- I chose this because my target audience of casual teenage gamers expect quick and intuitive controls- requiring players to use a mouse to navigate every menu would slow down the experience and create unnecessary friction, particularly when pausing mid-game.

Minimalistic Design

- Background elements are kept simple and uncluttered, with the gameplay area and UI elements kept visually distinct.
- I chose this because during gameplay players need to focus entirely on the falling notes. A busy or detailed background would distract from the core mechanic and increase the chance of players missing notes due to visual confusion rather than poor timing. This also keeps the project achievable within scope without sacrificing clarity.
- Links to success criteria: Usability Features – interface must be clear and intuitive.

User Set up

When the user first loads the game, they are presented with a login/ signup screen. This screen allows users to either create a new account or log in to an existing one. The interface is simple with clearly labelled fields for username and password.



- Active field highlighting: The currently selected field (username or password) is highlighted in orange.
- Password masking: Password input is displayed as asterisks for security
- Error feedback: Clear error messages inform users of issues
- Exit button which will close the game completely when clicked.

On the login screen, the currently selected input field is highlighted in orange. I chose this because without a clear visual indicator players may not know which field they are typing into, which would be particularly confusing for younger or less experienced users who are new to the game. Password input is displayed as asterisks rather than plain text. This was included because players may be playing in a shared or public environment and displaying passwords in plain text would be a basic security risk that undermines trust in the account system. Clear error messages are displayed when login or signup details are entered incorrectly, such as "Wrong Password" or "Username already exists." I chose this because without feedback players would have no way of knowing why their input was rejected, leading to frustration, particularly for casual players who are less likely to diagnose the issue themselves

Success Criteria:

Criteria 1: User Setup must appear when the user loads the game and must be simple and clear. There should be indicators if users input details incorrectly.

Menu

When the user enters the account, they are presented with a menu screen that can be navigated using the mouse. The menu displays the title TEMPO TAKEDOWN at the top with the current username displayed below it. Six options are arranged vertically for easy navigation.

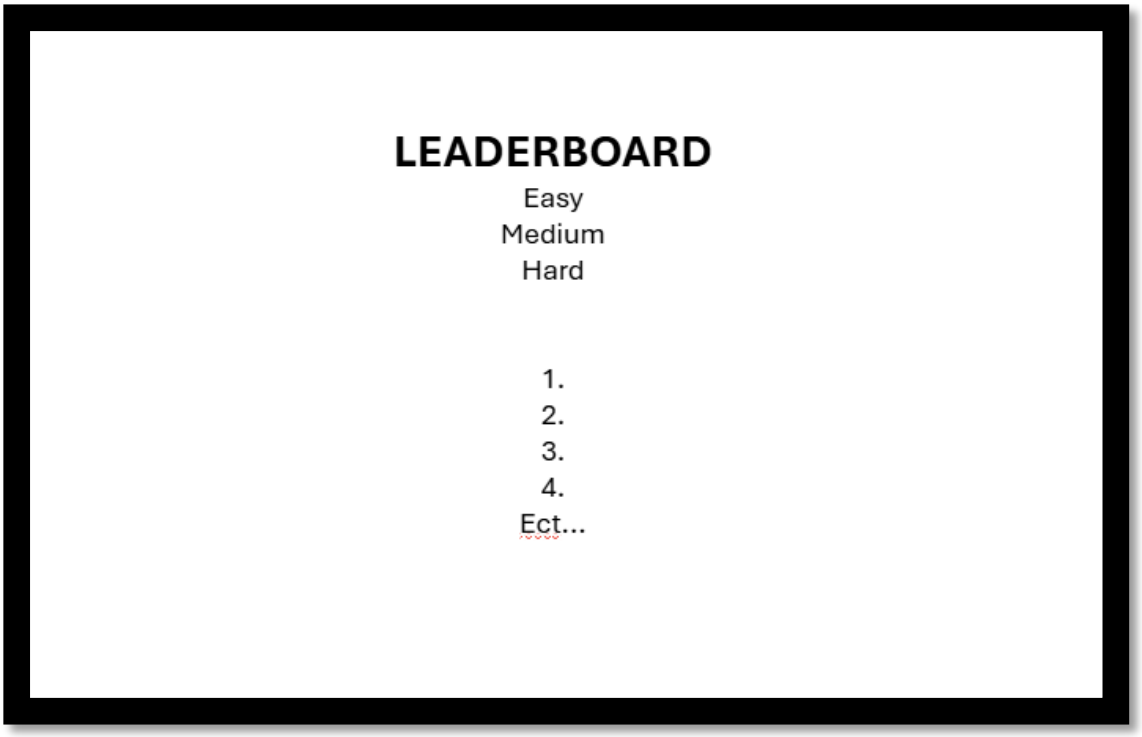
- “Play” button: Offers the option to play solo or multiplayer to then choose levels that they can pick from, Easy, Medium and Hard. Followed by a story line that carries onto the main game.
- “How to Play” button: Displays a screen that gives instructions on how to play the game.
- “Settings” button: Option to change controls.
- “Leaderboard” button: Shows top 10 players for each difficulty.
- “Sign out” button: Returns to login screen, allowing different users to play.
- “Exit” button: Closes the game.

Success Criteria:

Functional Menu = This criteria states that the game must have a menu at the start with clearly labelled and functional buttons. The buttons are clearly displayed on the screen and in an organised manner. Each button has a defined and meaningful function that leads to another part of the game. This fulfils the requirement for a user-friendly and clear interactive start menu.

Leaderboard

When the user selects the “Leaderboard” button from the main menu, they are taken to a screen displaying the top 10 players for each difficulty level. Players can switch between Easy, Medium and Hard levels, by clicking on the difficulty name at the top. The leaderboard shows players rank, username and best score for that difficulty. Players will also be allowed to return to the main menu by pressing ESC.



• Difficulty Tabs: Click on "Easy", "Medium", or "Hard" to view that level's leaderboard

• Top 10 Display: Shows the highest scoring players for competitive motivation

• Score Filtering: Only users with scores greater than 0 are displayed

• Personal Achievement: Players can see where they rank among others

• Updates: Leaderboard automatically updates when new high scores are achieved.

Success Criteria:

Criteria 14: Leaderboard= when users reach a high score, it will be displayed with their username and score on the leaderboard. This motivates players to keep improving and provides a sense of achievement, fulfilling thee requirement for competitive engagement.

How to Play

When the user selects the “How to Play” button from the main menu, they are taken to an instruction screen that explains the rules, controls and objectives of Tempo Takedown. The design is simple and readable, using large enough text to guide new players clearly. The user will also have the option to return to the menu after reading.

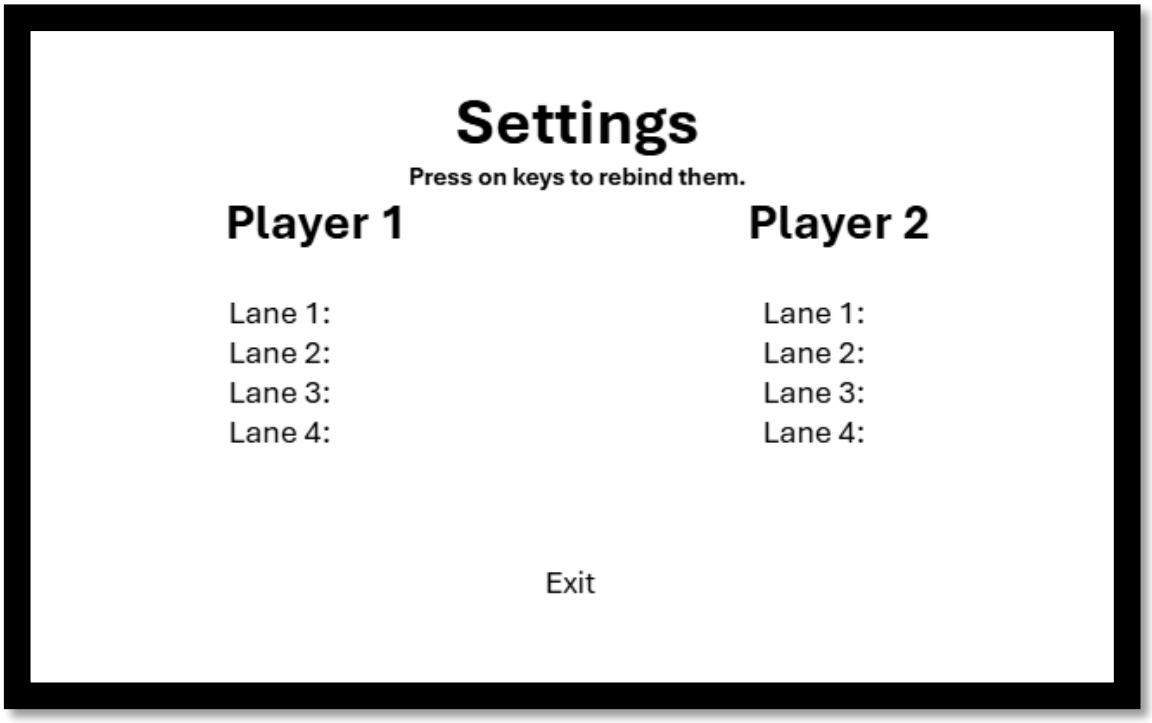
- “Return”: Takes user back to the main menu
- Simple instructions with border so text is easier to read.

Success Criteria:

Criteria 3: Easy to read Instructions = The instructions must be readable and written in clear, simple language to ensure that players can quickly understand the game mechanics, controls, and objectives without confusion.

Settings

The settings option in the main menu is where users can adjust the key binds for the game. The layout is simple and intuitive, with each action clearly labelled. Players can easily select a key and use their keyboard to input their preference. There is also a return that allows the player to return to the main menu after they have finished adjusting the controls.



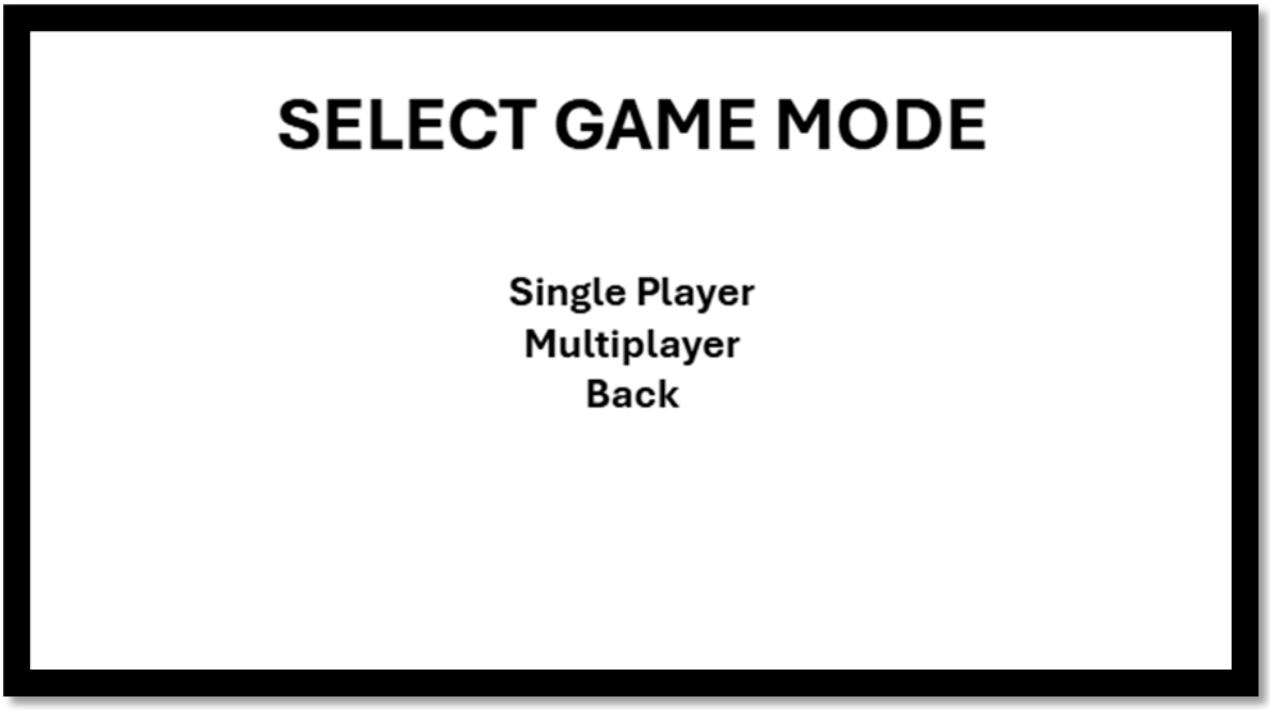
- Boxes to input keys = Players can select an in-game action and reassign I to a key of their choice. Changes are applied immediately.
- “Return” button = Takes user back to the main menu.
- Simple instructions that instruct players on what to do.

Success Criteria:

> Criteria 4. This criterion states that the game must allow players to change key bindings with all changes applied immediately and displayed. The click to rebind system with highlighting and instant assignment fulfils this requirement.

Play 1- Play menu

When the user selects the “Play” button form the main menu, they are presented with the Play menu which allows them to choose between Single Player and Local Multiplayer. The screen is clearly labelled with each option spaced for ease of selection. A back/return is available to return to the main menu.



Success Criteria:

7. Local Multiplayer = As required by Success Criteria 8, the game must include the option for 1 and 2 player options on the same device. In the screen above, the buttons allow the player to select what they want.

Play 2- Levels

After choosing a mode from the player menu section and going through the story introduction that gets presented as text after choosing desired game mode, the player selects a level. Each level gets unlocked based on progression to ensure players can improve and want to keep playing to unlock all levels. There is also a return to allow the player to go back to the player menu.

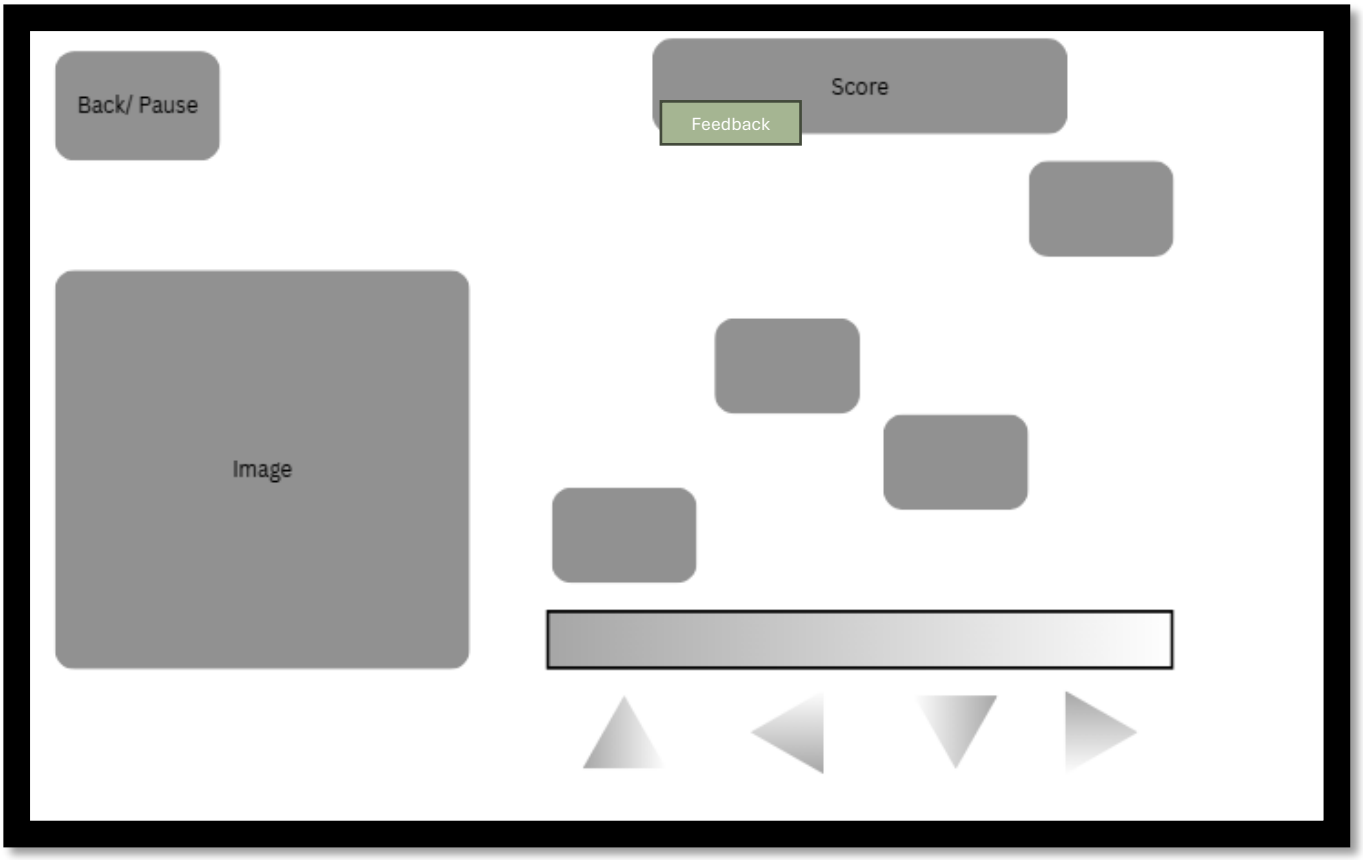
Success Criteria:

Criteria 7. Level Challenges = There should be 3 stages of difficulty at different levels that can be unlocked. This screen shows the levels that need to be unlocked and the current unlocked levels that players can play. The higher the level, the more difficult it is.

Criteria 20: Score thresholds for unlocking= Players must achieve minimum scores to unlock subsequent difficulty levels.

Play 3- Single Player

After selecting a level and going through the level story, the game begins. In single-player mode, the note lanes are centred on the screen for optimal visibility. The UI displays all necessary information while keeping the gameplay area clear.



- Back/ Pause button: Pauses the game when clicked or ESC is pressed
- Image display for aesthetic reasons
- Total Score shown at the top
- Falling notes for the actual gameplay and the gameplay menu which also includes the hitbox.

Success Criteria:

Criteria 5: Note scrolling mechanic: Notes must scroll smoothly down the screen in time with no stuttering or lag. This ensures a fair gameplay for all users.

Criteria 6: Feedback System: Visual cues are provided immediately at the top of the screen for instant player feedback.

Criteria 9: Tomb theme design: At least 60% of the game's assets must match the theme. This is so the game is unique and appealing to my consumers.

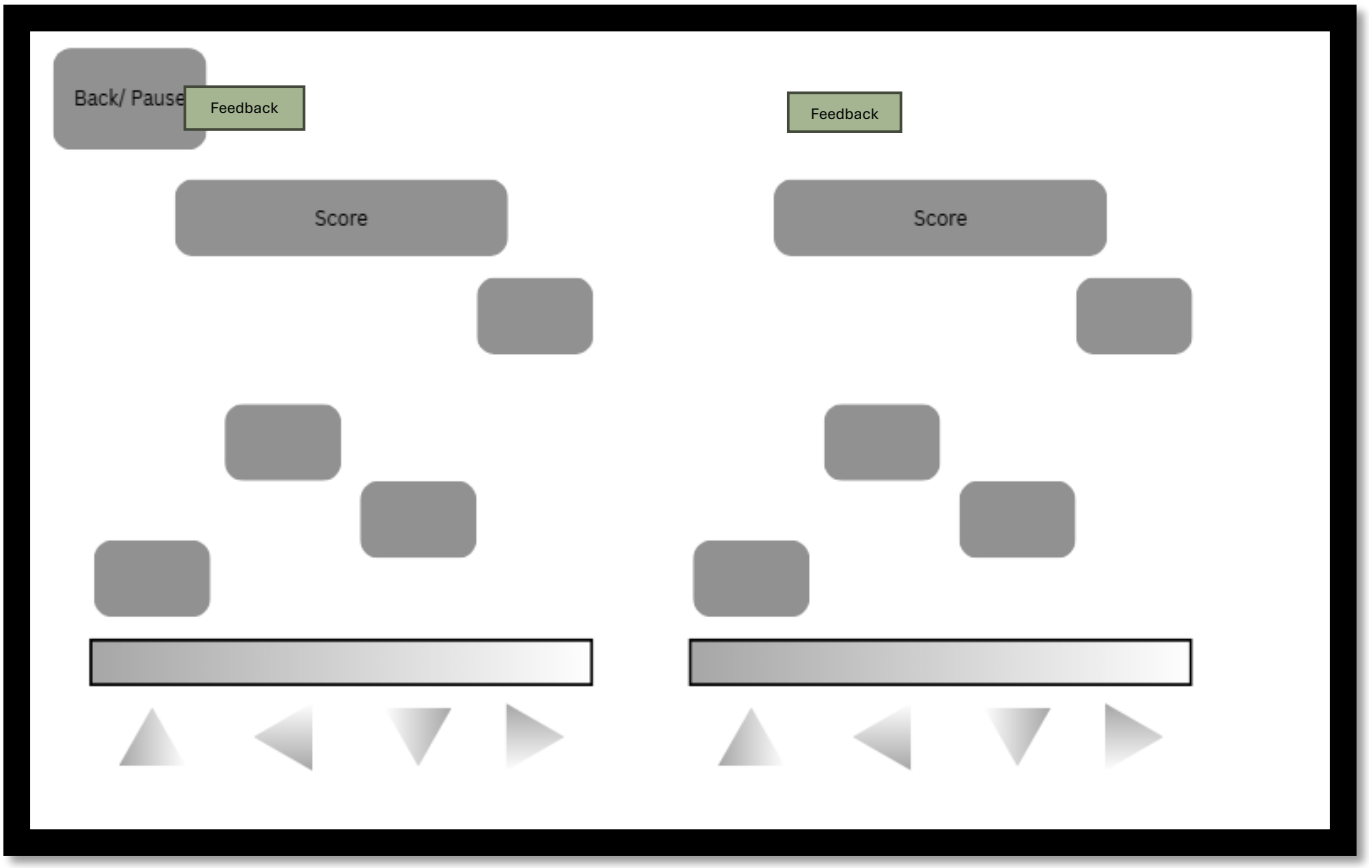
Criteria 18: Health System = Players must have a visible health counter starting at 3 health that decreases by 1 for each missed note or incorrect input. The health is clearly displayed in the top-right stats box.

Criteria 19: Long Note Mechanic = Game must include hold notes that require sustained key presses until the entire note passes the hit zone. These add gameplay variety and challenge.

Criteria 21: Combo Multiplier System = Score multipliers must increase based on combo count. The system rewards consistent accuracy and provides depth to scoring.

Play 4- Multiplayer

In multiplayer mode, two players share the same screen with their own sets of lanes, notes, and controls. The UI is reorganized to accommodate both players while keeping gameplay information clear.



Success Criteria:

1. Note scrolling mechanic: Notes must scroll smoothly down the screen in time with no stuttering or lag. This ensures a fair gameplay for all users.
2. Feedback System: Visual cues are provided immediately at the top of the screen for instant player feedback.
3. The game must support both players on the same device with no glitches or bugs.
4. Side-by-Side Layout: Player 1 lanes on left, Player 2 lanes on right
5. Independent Hit Feedback: Each player sees their own "Perfect!" or "Nice!" messages

Success Criteria:

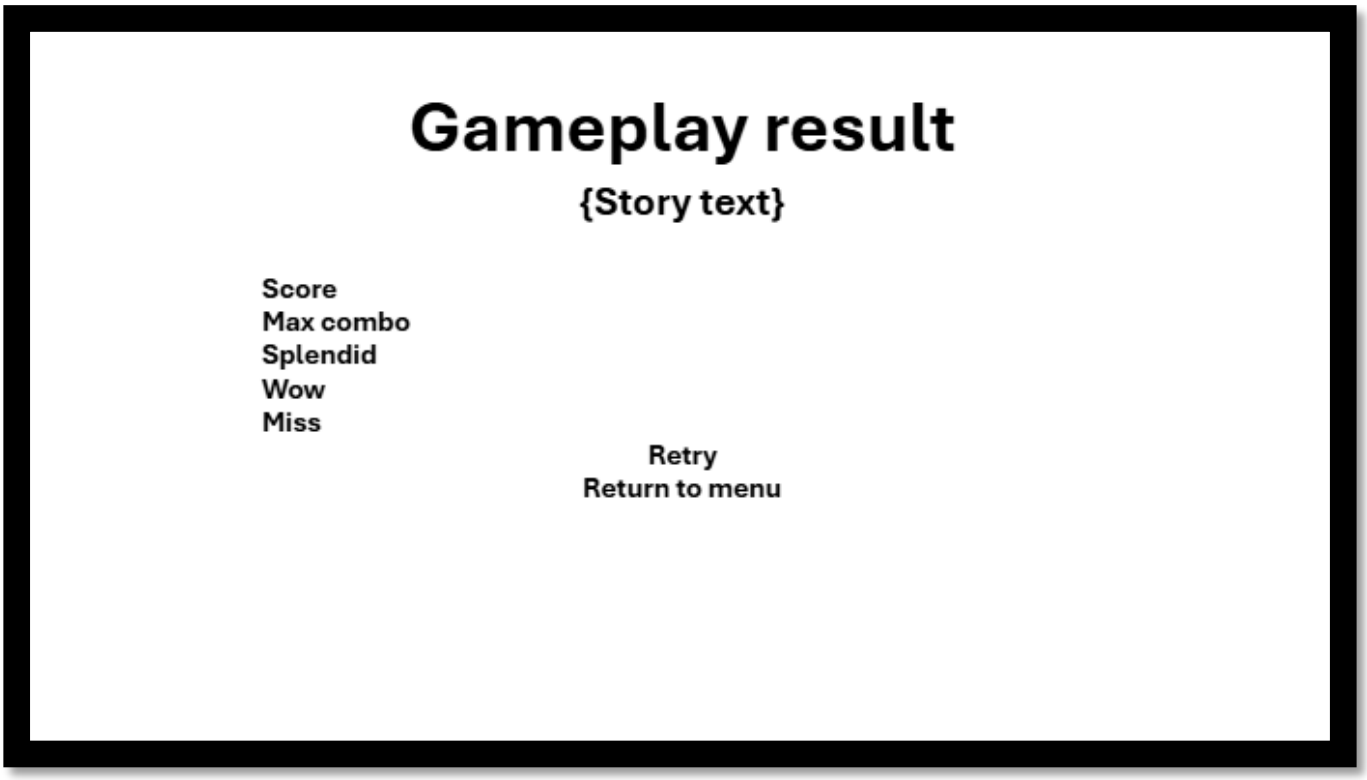
Criteria 8: Local Multiplayer = The game must support both 1 and 2 players on the same device with no glitches or bugs. Both players can play simultaneously with independent controls and scoring.

Criteria 5: Notes scrolling Mechanic = Notes must scroll smoothly down the screen for both players with no stuttering or lag.

Criteria 6: Feedback System = Visual cues are provided immediately for both players, with feedback messages positioned appropriately for each side.

Result 1- Single Player

After completing a level, the results screen displays the player's total score, accuracy, and combo count along with the total misses. This provides clear feedback on performance, motivates improvement, and allows the player to either retry the level or return to the main menu. The user will also have the option to replay or return to the menu.



Success Criteria:

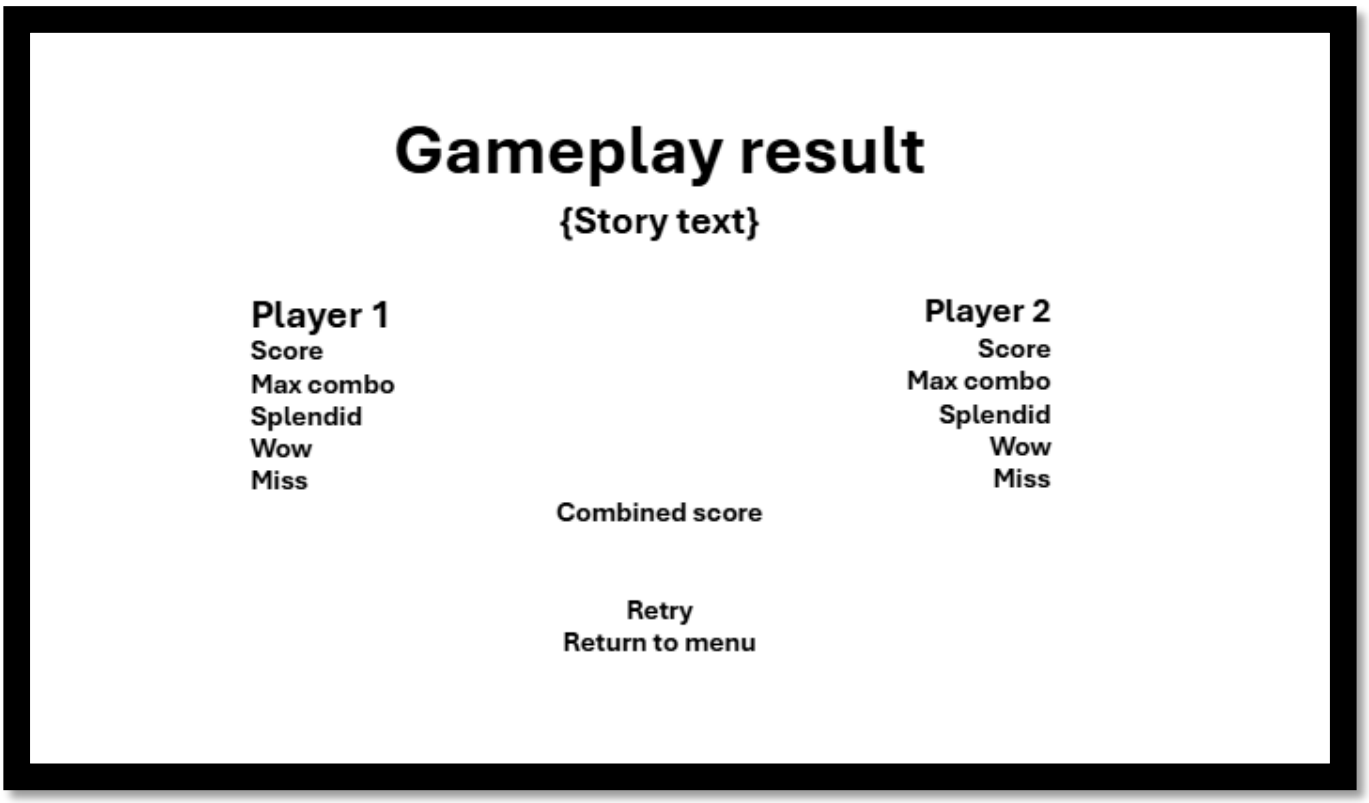
Criteria 12: Level Win Display – Single Player = This criteria states that if the player successfully completes the level by hitting the required notes, a screen is displayed confirming the win. This ensures clear feedback on success and allows the player to progress, fulfilling the requirement for understandable game outcomes

Criteria 15: Results Screen – Single Player = This criteria states that after completing a level, the results screen must display the player's total score, accuracy, and combo count. This provides clear feedback on performance, motivates improvement, and gives a sense of achievement, fulfilling the requirement for rewarding gameplay and encouraging replay.

.

Results 2- Multiplayer

After completing a level, the results screen displays both players' scores, accuracies, and combo counts. This allows each player to see their performance, compare results, and decide whether to retry the level or return to the main menu.



Success Criteria:

Criteria 12: Level Win Display – Multiplayer = This criteria states that if both players successfully complete the level by hitting the required notes, a screen is displayed confirming the win. This provides clear feedback for cooperative success and allows players to progress, fulfilling the requirement for understandable game outcomes.

Criteria 15: Results Screen – Multiplayer = This criteria states that after completing a level, the results screen must display both players' scores, accuracies, and combo counts. This allows each player to see their performance, compare results, and motivates improvement, fulfilling the requirement for rewarding gameplay and increasing replay value.

Endgame

After successfully completing the Hard difficulty (defeating Cleopatra), a special completion screen is shown for players who have beaten all three levels. This marks the end of the game's progression system.

- Ultimate Achievement: Recognizes players who complete all three difficulties
- Story Conclusion: Provides narrative closure to the pyramid quest
- Replay Value: Players can still replay any level to improve scores

Success Criteria:

Criteria 10: Storytelling Element = The game provides a complete narrative arc from entering the pyramid to obtaining the artifact, keeping players invested throughout all difficulty levels.

Algorithms

The following algorithms represent the core logic of my game. Each algorithm shows the key programming concepts including iteration, selection, data structures and real-time processing.

1. **Health System**

Purpose: To monitor player health, deduct health when mistakes happen and trigger game-over conditions when health reaches zero. This provides a fair challenge system with limited forgiveness errors.

```
FUNCTION Check_Lives(player_number):

    IF note passes hit zone AND note.hit
= FALSE THEN
    miss_count = miss_count + 1
    combo = 0
    health = health - 1

    IF health <= 0 THEN
        player_alive = FALSE

        IF multiplayer = FALSE THEN
            CALL Check_Game_Outcome()
        ELSE
            IF both players dead THEN
                CALL
Check_Game_Outcome()
            END IF
        END IF
    END IF
END IF
END FUNCTION
```

Justification:

I chose to implement the health system as a simple decrement function called on every missed note rather than a time-based or percentage-based system. This approach was chosen because it gives players a clear and predictable understanding of how many mistakes they can make, starting at 3 health means players always know exactly how close they are to losing. A more complex system such as a gradually depleting health bar over time would be harder for casual players to interpret and would make the game feel unfair. The multiplayer extension, where the game only ends when both players reach zero, was chosen to ensure neither player is forced to sit idle mid-game, keeping the social experience intact.

Success Criteria:

Criteria 12: Players have visible health that decreases by 1 for each missed note, providing clear visual feedback on performance.

Criteria 14: If the player misses a certain number of notes, then criteria 14 states that the game must conclude with the option of retrying or returning to menu. This algorithm implements a failure system that encourages replay and mastery.

Criteria 17: When the players health reaches 0, the game concludes creating a clear failure condition that encourages play.

Players will start with 3 health, each missed note immediately reduces health by 1, meaning players can miss a maximum of 5 notes before losing. In multiplayer mode, the game continues even if one player dies, ending only when both players have 0 health or if the time expires.

2. **Note Spawning System**

Purpose: To generate notes at appropriate intervals based on level difficulty. The spawning rate, note speed and frequency of long notes all increase with difficulty, creating progressive challenges.

```
IF level = Easy; THEN
    spawn_rate = 1000
    note_speed = 3
    long_note_chance = 0.15
ELSE IF level medium; THEN
    spawn_rate = 800
    note_speed = 4.5
    long_note_chance = 0.25
ELSE IF level = hard; THEN
    spawn_rate = 600
    note_speed = 6
    long_note_chance = 0.30
END IF

IF current_time - last_spawn_time >=
spawn_rate THEN
    lane = RANDOM(0, 3)

    IF RANDOM(0, 1) < long_note_chance
THEN
        length = RANDOM(200, 400)
        CREATE long note at
(lane_x[lane], -40, length)
    ELSE
        CREATE regular note at
(lane_x[lane], -40)
    END IF

    ADD note to notes_list
    last_spawn_time = current_time
END IF
```

Justification:

I chose to use a time-based spawn system with difficulty-dependent parameters rather than a fixed note chart. This was chosen because it allows the difficulty to scale naturally without requiring a manually authored note map for each level- the spawn rate, note speed, and long note frequency are simply adjusted per difficulty. A random lane selection ensures notes are unpredictable and replayable, which is important for a game targeting competitive players who would quickly memorise a fixed pattern. The long note chance was kept low on Easy and increased on Hard to progressively introduce more complex mechanics as players improve.

Success Criteria:

Criteria 2: Notes spawn at consistent intervals based on difficulty, creating smooth gameplay.

Criteria 6: The solution has 3 stages of difficulty that test player skills at different levels. Each difficulty has distinct spawn rates and notes speeds.

Notes spawn from the top of the screen in randomly separated lanes. Long notes appear at different frequencies depending on the level and the spawn timer resets after each spawn to maintain consistent rhythm. In multiplayer, notes are spawned independently for each player.

3. Note Movement System

Purpose: To move notes smoothly from the top of the screen to the hit zone. Smooth movements are essential for rhythm accuracy allowing players to time their inputs correctly.

```
FOR EACH note IN notes_list
    note.y = note.y + note_speed

    IF note.y > hit_zone_y + 40 AND
note.hit = FALSE THEN
        REMOVE note from notes_list
        miss_count = miss_count + 1
        combo = 0
        health = health - 1
    END IF
END FOR
```

Justification:

I chose a frame-by-frame position update approach where each note's y position is incremented by the note speed value on every iteration of the game loop. This is the standard approach for scrolling mechanics in Pygame because it ties movement directly to the game loop rather than real time, ensuring consistent behaviour regardless of processing speed. A tolerance buffer of 40 pixels around the hit zone boundary was included to prevent notes being

unfairly missed at the exact moment they cross the line, which would make the game feel unresponsive and frustrate players.

Success Criteria:

Criteria 2: A variety of notes must scroll smoothly down the screen in time with the music with no stuttering or lag.

The algorithm continuously updates each notes vertical position by adding the note_speed value. Notes that pass beyond the hit zone without being hit are removed and count as misses. The hit tolerance prevents unfair misses at the exact hit zone boundary.

4. Hit Detection and Timing System

Purpose: To determine the accuracy of player input by measuring how close the note is to the hit zone when the key is pressed. Different timing windows reward precise inputs with higher scores and better feedback.

```
distance = ABS(note.y - hit_zone_y)

IF distance < 10 THEN
    performance = "Splendid"
    base_score = 200
    perfect_count = perfect_count + 1

ELSE IF distance < 30 THEN
    performance = "Wow"
    base_score = 100
    good_count = good_count + 1

ELSE IF distance < 40 THEN
    performance = "Close"
    base_score = 50
    good_count = good_count + 1

ELSE
    performance = "Miss"
    combo = 0
    health = health - 1
    RETURN
END IF

multiplier = GET_Combo_Multiplier(combo)
points = base_score * multiplier
score = score + points
combo = combo + 1

IF combo > max_combo THEN
    max_combo = combo
END IF

note.hit = TRUE
```

Justification:

I chose a distance-based timing window system with four distinct accuracy bands - Splendid, Wow, Close, and Miss - rather than a binary hit or miss system. This was chosen because graded feedback rewards players for improving their timing precision even if they are not yet

hitting perfect inputs, which is important for keeping casual players engaged. The combo multiplier was linked directly to this system so that only consistent accurate inputs are rewarded with bonus points, giving competitive players a meaningful reason to aim for Splendid ratings rather than simply surviving. Using ABS to calculate distance from the hit zone ensures the same tolerance applies whether the note is slightly early or slightly late, making the system fair in both directions.

Success Criteria:

Criteria 3: Visual cues must be provided immediately when a player hits or misses a note, showing performance with feedback.

Criteria 11: Combo counters update accurately after each note, with higher combos awarding bonus points through the multiplier system.

5. Win / Loss Condition

Purpose: To determine whether the player has won or lost based on score thresholds and time remaining. This provides clear success conditions and motivates players to reach target scores.

```
IF level = "easy" THEN
    threshold = 6700
ELSE IF level = "medium" THEN
    threshold = 10000
ELSE
    threshold = 15000
END IF

IF game_timer >= 90000 OR health <= 0
THEN
    game_over = TRUE
END IF

IF game_over = TRUE THEN
    IF multiplayer = TRUE THEN
        final_score = player1_score +
player2_score
    ELSE
        final_score = player_score
    END IF

    IF health > 0 AND final_score >=
threshold THEN
        result = "WIN"
    ELSE
        result = "LOSE"
    END IF

    CALL Results_Screen(result)
END IF
```

Justification:

I chose to implement a dual win condition, surviving 90 seconds OR reaching the score threshold, rather than a single condition. This was chosen because it caters to both player types: casual players who may not reach the threshold can still win by surviving, while competitive players are incentivised to maximise their score. The score thresholds increase significantly across difficulties (6700, 10000, 15000) to reflect the faster note speeds and higher scoring potential at harder levels, ensuring the challenge scales fairly. Combining both

players' scores in multiplayer was chosen so that teamwork is rewarded rather than one strong player carrying the other without consequence.

Success Criteria:

Criteria 7: Different score thresholds test player skills at each difficulty.

Criteria 13: If the player successfully meets the score threshold or survives 90 seconds, a display informs them they have completed the level.

Criteria 14: If the player does not meet requirements, a display informs them they have lost with options to retry or return to menu.

The game ends when either 90 seconds have passed OR the players health reaches 0. Players win if they survive the 90 seconds or reach the score threshold before dying. In multiplayer, both players scores are combines. Winning unlocks the next difficulty and updates the leaderboard with the players best score.

6. User Authentication

Purpose: To provide login and sign-up functionality, maintaining individual player progress and statistics. This allows players to save their progress and compete on leaderboards.

```
users = LOAD_FILE("users.json")

INPUT username
INPUT password

IF username = "" OR password = "" THEN
    DISPLAY "Please fill all fields"
    RETURN
END IF

IF username IN users THEN
    IF users[username]["password"] =
password THEN
        current_user = username
        DISPLAY "Welcome back!"
    ELSE
        DISPLAY "Wrong password"
    END IF
ELSE
    users[username] = {
        "password": password,
        "best_easy": 0,
        "best_medium": 0,
        "best_hard": 0,
        "unlocked_medium": FALSE,
        "unlocked_hard": FALSE
    }
    SAVE_FILE("users.json", users)
    current_user = username
    DISPLAY "Account created!"
END IF
```

Justification:

I chose to store user data in a JSON file rather than a database because JSON is lightweight, human-readable, and natively supported in Python without requiring any additional libraries. A full database such as SQLite would be disproportionate for the scale of this project and would significantly increase development complexity. The algorithm automatically creates a new account if the username does not exist rather than requiring a separate signup screen, which simplifies the user journey. Default values of 0 for all scores and False for all unlocks are

assigned on account creation so that the rest of the program can always rely on these fields existing, preventing key errors during gameplay.

Success Criteria:

Criteria 1: User setup appears when loading the game with simple, clear fields. Indicators inform users if details are correct or not such as “Wrong Password”.

Game feature Designs

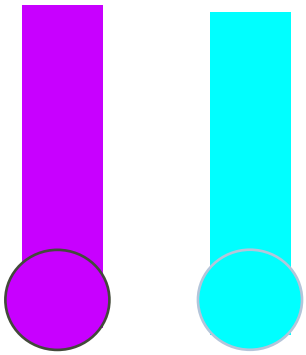
The following are the visual design concepts for *Tempo Takedown*.

Notes

Yellow Single Note:

Single notes are yellow circles. Regular notes require a single key press when the note reaches the green hit zone line. The note changes to green upon successful hit. These notes are removed from the screen after being hit or missed.

Purple Long Notes:



Long notes are rectangles with a circular bottom. Longer notes require the player to press and hold the corresponding key. The note changes from purple to cyan when the player begins holding the key. Players must hold the key until the entire note passes the hit zone. Releasing early results in a failed note and combo reset. Successfully holding awards bonus points progressively.

Input Verification

Username and password fields are validated to ensure neither is left empty before processing. This was chosen because attempting to look up an empty string in the users dictionary would either cause a key error or incorrectly create a blank account, breaking the authentication system entirely.

Keyboard inputs during gameplay are validated so that only keys mapped to active note lanes are registered. This prevents accidental keypresses from affecting the score and is especially

important in multiplayer where both players are pressing keys simultaneously on the same keyboard, without this validation, Player 2's inputs could incorrectly register as Player 1 hits.

Mouse clicks on menu buttons are validated by checking whether the cursor is within the button's boundaries at the time of the click. This prevents accidental selections when a player clicks near but not on a button, which would be particularly frustrating on the level select screen where selecting the wrong difficulty could send a player into a level they haven't unlocked.

Variables

Variable Name	*Data Type*	*Description*
score_p1 / score_p2	Integer	Stores each player's total points. Incremented on successful hits and weighted by the current combo multiplier. Must be ≥ 0. Never decremented (only resets to 0 at the start of a new game).
combo_p1/combo_p2	Integer	Each player's remaining lives. Starts at 5 and decreases by 1 on a missed note. Game ends when it reaches 0. Clamped to a minimum of 0 using max(0, health - 1) to prevent negative values.
max_combo_p1/max_combo_p2	Integer	Records the highest combo reached during the session. Shown on the results screen. Updated only when the current combo exceeds it using max(max_combo, combo). Never decreases.
health_p1/health_p2	Integer	Stores player 1's remaining lives. Starts at 5 and the game ends when it reaches 0. Clamped to a minimum of 0 using max(0, health - 1) to prevent negative values.
notes_p1/notes_p2	List of dictionaries	All active notes currently on screen. Each note stores its x/y position, hit status, and for long notes, length and hold progress. Notes are removed once hit or missed.

		Long note length is randomly chosen between LONG_NOTE_MIN_LENGTH and LONG_NOTE_MAX_LENGTH.
hit_feedback_p1/hit_feedback_p2	Dictionary	Stores the feedback text (e.g. "Splendid!", "Miss"), display colour, and a countdown timer so the message fades after a short time. Timer counts down each frame and feedback is only drawn while timer > 0.
current_user	String	Stores the username of the currently logged-in player. Cannot be empty. Password must be ≥ 8 characters and contain at least one letter and one number.
paused	Boolean	Whether the game is currently paused. While True, note movement, spawning, and the game timer are all halted.Only ever True or False- toggled by pressing Escape or P during gameplay.
KEYS_P1/KEYS_P2	List	Pygame key constants mapped to each player's four lanes. Defaults to A/S/D/F and J/K/L/; but can be rebound in settings. On rebind, the new key is checked against all currently assigned keys. Duplicates are rejected with an error message.
WIDTH	Integer	Screen width in pixels (1200).
HEIGHT	Integer	Screen height in pixels (700).
FPS	Integer	Frame rate of the game (60 frames per second).
LANE_X_P1/ LANE_X_P2	List	X-coordinates for player lanes
GAME_DURATION	Integer	The total length of a game session in milliseconds (90,000 ms = 90 seconds).

		The timer counts up and the game ends when it reaches this value. Fixed constant. Acts as one of two end conditions the other being health reaching 0.

Files

The program uses a single external file, users.json, to store persistent user data between sessions. This file is structured as a JSON object where each key is a username mapping to a dictionary containing the user's password, best scores for each difficulty level, unlock status for medium and hard, and whether they have completed the game. JSON was chosen over a CSV or plain text file because it natively supports nested dictionaries, allowing all of a user's data to be stored and retrieved under a single key. Python's built-in json module handles reading and writing without any additional libraries, and the file is automatically created if it does not already exist, meaning no manual setup is required before the program is run.

File Name	*Contents*	*Purpose*
users.json	Username, password, best scores for each level.	Stores user's login credentials and other details. They are read at log in and updated when a player achieves a new high score.

Testing

The modules are organised in a structured sequence so that each component can be developed and tested independently before progressing further. Each module depends only on previously completed modules, allowing continuous testing throughout development and avoiding circular dependencies.

Modules 1 to 5 form the foundation of the application, covering user authentication and menu navigation. These can be fully tested without requiring any gameplay features. Modules 6 to 10 introduce the core gameplay systems, including note mechanics, scoring, health, and results, with each module building directly on the previous one to ensure a stable game loop. Modules 11 to 15 add additional features such as long notes, multiplayer, story elements, the game timer, and the leaderboard, extending the functionality of an already working game.

This structure ensures the game remains functional at every stage, allowing issues to be identified and resolved early while supporting a clear and efficient development process. Testing will be carried out after each module is implemented to ensure it met its expected behaviour before moving on to the next stage.

Foundation Modules

- Initial Setup Test
- Module 1 – User Authentication
- Module 2 – Exit Confirmation
- Module 3 – Main Menu
- Module 4 – How to Play
- Module 5 – Settings and Key Bindings

Core Gameplay Modules

- Module 6 – Game Mode and Level Selection
- Module 7 – Core Notes Mechanic
- Module 8 – Scoring and Hit Feedback
- Module 9 – Health System
- Module 10 – Results Screen and Level Progression

Enhancement Modules

- Module 11 – Long Notes
- Module 12 – Multiplayer Support
- Module 13 – Story Screens
- Module 14 – Game Timer
- Module 15 – Leaderboard

Initial Setup Test

Test Number	Target	Expected Output	Actual Output
1	Pygame initialises without errors	Window titled “Tempo Takedown” opens at 1200×700	
2	All colour constants accessible	No NameError when referencing colour variables	
3	Font objects created	FONT, BIG_FONT, SMALL_FONT, HEADER_FONT all render text	

Module 1 – User Authentication

Test Number	Target	Expected Output	Actual Output
1	Login screen displays on startup	Username and password fields appear with the title “LOGIN / SIGN UP”	

2	Empty username or password prevents login	“Fill all fields” error message is displayed	
3	Password shorter than 8 characters is rejected	“Password must be at least 8 characters” error is displayed	
4	Password with no number is rejected	“Password must contain at least one number” error is displayed	
5	Password with no letter is rejected	“Password must contain at least one letter” error is displayed	
6	Existing user logs in with correct credentials	User data loads and the main menu opens	
7	Existing user enters wrong password	“Wrong password” error message is displayed	
8	New username with valid password creates account	New entry written to users.json with default scores and locked levels	
9	User data persists between sessions	Scores and progress retained after logging out and back in	

Module 2 – Exit Confirmation

Test Number	Target	Expected Output	Actual Output
1	Exit is reachable from the login screen	Clicking 'Exit' on the login screen opens the confirmation dialog	
2	Confirmation screen shows both options	'Are you sure you want to exit?' is displayed with 'Yes' and 'Return' buttons	
3	Yes button closes the application	Pygame quits and the process terminates	
4	Return button dismisses the dialog	User is returned to the screen that triggered the exit prompt	

Module 3 – Main Menu

Test Number	Target	Expected Output	Actual Output

1	Game title displays centred at the top	'TEMPO TAKEDOWN' appears in large text at the top of the screen	
2	Logged-in username is shown beneath the title	'User: <username>' is displayed in small text	
3	All six menu buttons are visible	Play, How to Play, Settings, Leaderboard, Sign Out, Exit are all present	
4	Buttons highlight orange on hover	Mouse hover changes button colour from white to orange	
5	Play opens the game mode selection screen	Single Player and Multiplayer options are shown	
6	Single Player routes correctly	Story intro plays then the level select screen is shown	
7	Multiplayer routes correctly	Story intro plays with multiplayer text, then level select is shown	
8	How to Play navigates correctly	How to Play screen opens	
9	Settings navigates correctly	Settings screen opens	
10	Leaderboard navigates correctly	Leaderboard screen opens	
11	Sign Out returns to the login screen	Login screen is shown; previous session is cleared	
12	Exit opens the confirmation dialog	Exit confirmation screen is displayed	

Module 4 – How to Play

Test Number	Target	Expected Output	Actual Output
1	Screen opens from the main menu	How to Play screen is displayed after clicking the button	
2	Regular note instructions are shown	Yellow circle note type is described with the correct key-press instruction	

3	Long note instructions are shown	Purple rectangle note type is described with the hold-key instruction	
4	Scoring values are listed correctly	Splendid 200 pts, Wow 100 pts, Close 50 pts are all shown	
5	ESC returns to the main menu	Main menu is shown immediately on pressing ESC	

Module 5 – Settings and Key Bindings

Test Number	Target	Expected Output	Actual Output
1	Settings screen opens from the main menu	Key binding screen is displayed with P1 (left) and P2 (right) columns	
2	Current bindings are shown for both players	Four lane labels for P1 and four for P2 are listed with their key names	
3	Clicking a lane label enters rebinding mode	The label turns orange and 'Press a key to rebind…' prompt appears	
4	Pressing a key assigns it to the selected lane	Lane label updates immediately to show the new key name	
5	Duplicate keys are rejected	Error: '<key> is already in use!' and the binding is not changed	
6	ESC during rebinding cancels without changing	Rebinding mode ends; the original key remains	
7	ESC from the main settings view returns to menu	Main menu is displayed	
	New bindings take effect immediately in gameplay	After remapping, the new keys register correctly during a test game	

Module 6 – Game Mode and Level Selection

Test Number	Target	Expected Output	Actual Output
1.	Game mode screen opens from Play	Three options displayed correctly, Single Player and Multiplayer and Back	
2.	Back button returns to main menu	Returned to main menu correctly	

3.	Level select displays all difficulties	All three levels are displayed	
4.	Locked levels cannot be selected	Locked levels greyed out and unresponsive to clicks	
5.	ESC on level select returns to game mode	ESC returned to game mode screen correctly	

Module 7 – Core Notes Mechanic

Test Number	Target	Expected Output	Actual Output
1.	Notes spawn at the top of the lanes	Notes spawned correctly at the top of each lane	
2.	Notes move downward	Notes moved at the correct speed for each difficulty	
3.	Notes use all lanes	Notes distributed across all four lanes	
4.	Hit zone visible	Triangle hit zones visible with key labels inside	
5.	Correct key hits note	Note removed and hit registered on correct key press	
6.	Early key press — no hit registered	Early presses outside the 40-pixel window had no effect	
7.	Missed note — counts as miss	Miss registered when note passed the hit zone	
8.	Notes removed after miss	Notes removed cleanly after passing the threshold	
9.	Multiplier applies at 10+ combo	Points awarded equal the base value multiplied by combo	
10.	Multiplier of x1.5 applies at 20+ combo	Points awarded equal the base value multiplied by combo	
11.	Multiplier applies at 30+ combo	Points awarded equal the base value multiplied by combo	
12.	Multiplier applies at 50+ combo	Points awarded equal the base value multiplied by combo	
13.	Max combo is tracked independently of current combo	Highest combo reached is preserved even after a miss	
14.	P1 and P2 scores are fully independent	A P1 hit never changes P2's score and vice versa	

Module 8 – Scoring and Hit Feedback

Test Number	Target	Expected Output	Actual Output
1.	Score starts at zero and increments	Score started at 0 and incremented correctly	
2.	Hit within 10px awards Splendid	Green Splendid feedback, 200×multiplier added	
3.	Hit within 30px awards Wow	Yellow Wow feedback, 100×multiplier added	
4.	Hit within 40px awards Close	Orange Close feedback, 50×multiplier added	
5.	Feedback text appears and fades	Feedback appeared and timer decremented to zero	
6.	Combo counter increments	Combo incremented by 1 on every successful hit	
7.	Miss resets combo	Combo reset to 0 on miss	
8.	Miss displays "Miss :(" in red	Red miss feedback displayed correctly	
9.	Combo multiplier scales correctly	All five multiplier thresholds verified	
10.	Multiplier correctly increases points	Splendid at combo 10 = 240 pts (200 × 1.2) confirmed	
11.	Miss count increments	Miss counter incremented correctly on each missed note	

Module 9 – Health System

Test Number	Target	Expected Output	Actual Output
1	Player starts with 3 health points	Health displays as 3 at the start of every game	
2	Each missed note reduces health by 1	Health decrements correctly for every note that passes the hit zone	
3	Health cannot fall below 0	Health stays at 0 rather than going negative	

4	Health is visible and updates in real time	'Health: X' shown on screen and decrements immediately on a miss	
5	Game ends when health reaches 0	Results screen is shown the moment health hits 0	
6	Game ends when the 90-second timer expires	Results screen appears when the countdown reaches 0	
7	Timer is visible and counts down correctly	Remaining seconds are displayed and decrease in real time	
8	In multiplayer, game ends when both players are dead	Results screen does not appear until both P1 and P2 health reach 0	
9	Pressing a key with no note nearby does not cost health	Health only decreases when a note was in range and timing was wrong	

Module 10 – Results Screen and Level Progression

Test Number	Target	Expected Output	Actual Output
1.	Victory screen shows when threshold is met and player survives	'VICTORY!' shown in green with the correct story text for the level	
2.	Failure screen shows when health reaches 0	'CAPTURED!' shown in red	
3.	Score vs threshold shown on failure	'Score: X / Y needed' displayed when threshold was not met	
4.	Correct story text per level on victory	Bronze Key text for Easy, Silver Key for Medium, Artifact for Hard	
5.	All P1 stats are displayed	Score, Max Combo, Splendid, Wow, and Miss counts shown for P1	
6.	In multiplayer, P2 stats are also displayed	P2 Score, Max Combo, Splendid, Wow, Miss shown alongside P1	
7.	Combined score shown in multiplayer	'Combined Score: X' displayed in yellow on multiplayer results	
8.	Best score saved to users.json if beaten	Stored high score updates only when the new score exceeds it	
9.	Retry restarts the current level	Level begins again from the start with all stats reset	
10.	Return to Menu navigates correctly	Main menu is shown	
11.	ENTER on a victory screen continues progression	User returns to the level select screen	
12.	Winning Easy sets unlocked_medium in users.json	File shows 'unlocked_medium: true' after a successful Easy run	
13.	Congratulations screen shows after first Hard win	White screen with story conclusion text is displayed	

14.	Congratulations screen only shows once	'completed_game' flag prevents it appearing on replays	

Module 11 – Long Notes

Test Number	Target	Expected Output	Actual Output
1.	Long notes appear during gameplay	Purple rectangles with varying lengths spawn on the lanes	
2.	Long note spawn chance matches difficulty	Easy 15%, Medium 25%, Hard 30% of spawned notes are long	
3.	Easy mode limits simultaneous long notes to one	A second long note does not spawn while one is already active on Easy	
4.	Holding the key awards progressive bonus points	Score increases approximately every 3 frames while the key is held	
5.	Releasing the key early resets the combo	'Released early!' message appears and combo returns to 0	
6.	Holding through the full note completes it	Note marked as hit; bonus score based on hold progress is awarded	
7.	Long note turns cyan when being held correctly	Note colour changes from purple to cyan on a successful hold start	
8.	Long note miss costs one health	Health decrements if the note passes without being held	

Module 12 – Multiplayer Support

Test Number	Target	Expected Output	Actual Output
1	Two separate lane sets displayed	Both lane sets rendered on respective sides with no overlap	
2	Each player's input only affects their lanes	P1 keys had no effect on P2 notes and vice versa	
3	Scores displayed independently	Scores displayed in separate boxes on each side	
4	Health tracked independently	Each player's health decremented only on their own misses	

5	Hit feedback appears correctly	Feedback offset correctly to each player's lane area	
6	Game continues if one player dies	Surviving player continued after other player died	
7	Combined score on results screen	Combined and individual scores displayed correctly	
8	Win condition uses combined score	Combined score compared against threshold correctly	

Module 13 – Story Screens

Test Number	Target	Expected Output	Actual Output
1	Intro displays before game mode	"ANCIENT EGYPT - THE PYRAMID" shown in orange	
2	ENTER continues past intro	Level select opened correctly on ENTER	
3	Level-specific story shown before each difficulty	Correct chamber title and story text shown for each level	
4	Failure story text shown	Failure text displayed correctly on results screen	

Module 14 – Game Timer

Test Number	Target	Expected Output	Actual Output
1	Timer counts down from 90s	Timer counted down from 90 correctly each second	
2	Timer visible throughout gameplay	Timer remained visible in the stats panel throughout	
3	Game ends when timer reaches 0	Results screen triggered at 0 seconds remaining	
4	Timer works in multiplayer	Shared timer counted down correctly in multiplayer	

Module 15 – Leaderboard

Test Number	Target	Expected Output	Actual Output
1.	Leaderboard opens from the main menu	Top-score list is shown after clicking the Leaderboard button	
2.	Clicking a difficulty tab switches the list	Easy, Medium, and Hard tabs each show the correct ranked scores	
3.	Scores are sorted in descending order	The highest score appears at position 1; lower scores follow	
4.	Usernames are shown alongside scores	Each entry displays '<rank>. <username> — <score>'	
5.	Up to 10 entries shown per difficulty	No more than 10 rows appear regardless of how many users exist	
6.	ESC returns to the main menu	Main menu is displayed on pressing ESC	

Final Testing

In the post-development stage, a range of testing methods will be used to make sure the rhythm game works correctly and meets all of its requirements. Black-box testing will be used first, as this method tests the system from the perspective of an end user without looking at the internal code. This means the focus will be on visible features such as navigating menus, selecting difficulty levels, changing key bindings, and checking that the game correctly detects hits, misses, and long notes during gameplay. Black-box testing will also be used to test the login and sign-up system to make sure usernames and passwords are validated properly and that user progress and high scores are saved and loaded correctly. It will also be used to test the local multiplayer mode to make sure both players' inputs, scores, and health are tracked separately without any errors.

White-box testing will also be carried out to test the internal logic of the program. This involves looking at the code and testing specific parts of the program based on how the algorithms and decision-making work. For example, the hit detection system will be tested to make sure each timing window gives the correct result, and the score and combo system will be tested to make sure values update correctly. This type of testing helps make sure there are no logical errors in the program and that calculations and conditions work as expected.

Finally, beta testing will be used to get feedback from real users who are part of the target audience. These testers will play the game and report any issues they find, such as timing problems, confusing controls, or difficulty being too easy or too hard. Beta testing is important

because as the developer I may be too familiar with the game and might not notice some problems. Other players can give an unbiased opinion and help find issues that were not found during black-box or white-box testing. This helps make sure the final game is reliable, easy to use, and ready to be released.

Development

Initial setup

I began by importing the required libraries: pygame, sys, random, json and os are all used at various points throughout the game. pygame.init() initialises all Pygame modules and must be called before any display or font objects are created. CLOCK is a pygame.time.Clock object used to limit and measure frame rate. FPS is set to 60, again as a named constant, so CLOCK.tick(FPS) reads clearly and the target frame rate could be changed in one place.

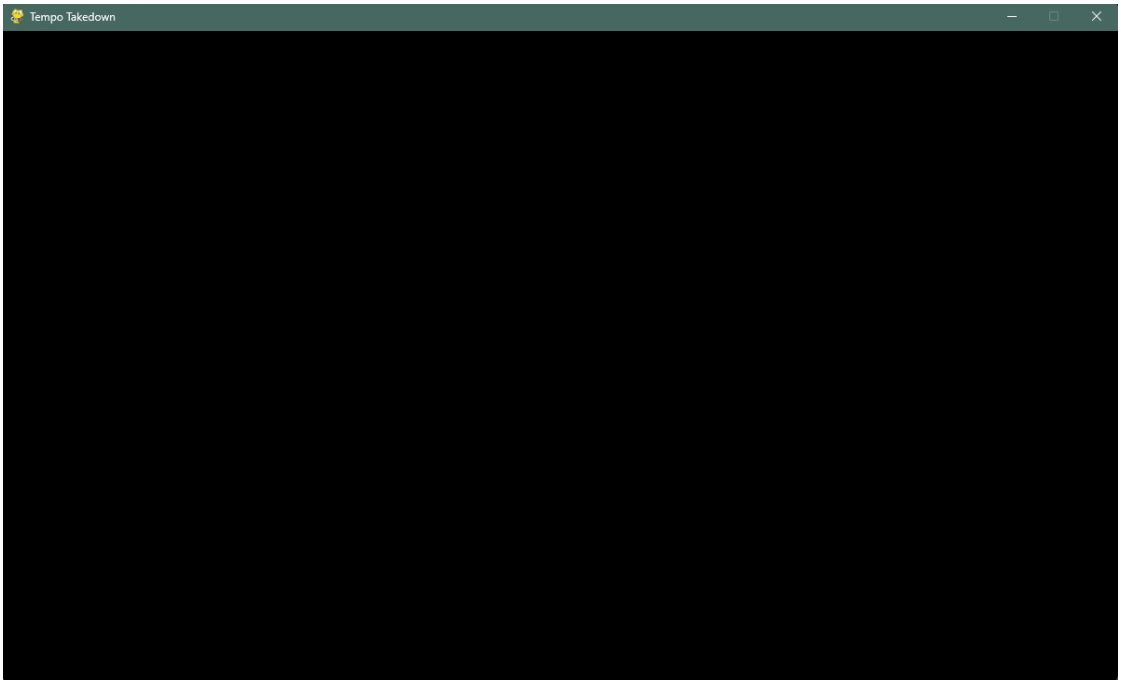
GAME_DURATION is set to 90000, representing 90 seconds in milliseconds. I stored it in milliseconds to match the unit Pygame's clock uses, which meant I didn't need to convert anything mid-calculation. The only conversion needed is at the display step, where the remaining time is divided by 1000 to show whole seconds to the player.

The display was set to 1200 by 700 pixels. I chose this resolution as it provides enough horizontal space to accommodate both sets of lanes in multiplayer mode without the UI feeling cramped. Four font objects were created at sizes 22, 28, 36 and 48, corresponding to SMALL_FONT, FONT, HEADER_FONT and BIG_FONT respectively. Arial was chosen as it is universally available across Windows, macOS and Linux systems.

All colour tuples and gameplay constants were defined as named constants. Using named constants rather than raw tuples and numbers makes the intent of each value clear and allows changes to be made in a single place without searching the entire file. Python 3.11 was used alongside Pygame 2.5. The project was structured as a single script (tempo_takedown.py) with a users.json file generated automatically on first run.

Testing:

When running the code, a window opened and closed as expected.




```python
import pygame, sys, random, json, os

#Initial Setup
pygame.init()
WIDTH, HEIGHT = 1200, 700
SCREEN = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption('Tempo Takedown')
CLOCK = pygame.time.Clock()
FPS = 60
GAME_DURATION = 90000
USER_FILE     = 'users.json'

#FONTS
FONT        = pygame.font.SysFont('arial', 28)
BIG_FONT    = pygame.font.SysFont('arial', 48)
SMALL_FONT  = pygame.font.SysFont('arial', 22)
HEADER_FONT = pygame.font.SysFont('arial', 36, bold=True)

#COLOURS
WHITE=(255,255,255)
BLACK=(0,0,0)
GRAY=(60,60,60)
GREEN=(0,255,0)
RED=(255,0,0)
ORANGE=(255,165,0)
BLUE=(0,0,255)
YELLOW=(255,255,0)
PURPLE=(200,0,255)
CYAN=(0,255,255)

```

Test Number	Target	Expected Output	Actual Output	Pass/Fail

1.	Pygame initialises without errors	Window titled 'Tempo Takedown' opens at 1200×700	Window opened at 1200×700 as expected	**PASS**
2.	All colour constants accessible	No NameError when referencing colour variables	All constants resolved correctly	**PASS**
3.	Font objects created	FONT, BIG_FONT, SMALL_FONT, HEADER_FONT all render text	Fonts rendered correctly at their respective sizes	**PASS**

Module 1 – User Authentication

The first module I developed was user authentication. Its purpose is to let players create accounts or log in, with progress and scores persisting between sessions via a local JSON file. The purpose of this module is to allow players to either create a new account or log in to an existing one, with their progress, best scores, and unlock states being saved and retrieved between sessions. I decided to store user data in a local JSON file called users.json as it does not require any external libraries beyond Python's built-in json and os modules, and the data is simple enough to be stored as a dictionary. Without this module the game has no way of

tracking individual player progress across sessions.

```python
def draw_center(text, font, color, y, x_offset=0):
    surf = font.render(text, True, color)
    rect = surf.get_rect(center=(WIDTH//2 + x_offset, y))
    SCREEN.blit(surf, rect)

# USER DATA
def load_users():
    if not os.path.exists(USER_FILE):
        return {}
    with open(USER_FILE, "r") as f:
        return json.load(f)

def save_users(users):
    with open(USER_FILE, "w") as f:
        json.dump(users, f, indent=2)


# LOGIN / SIGNUP — PROTOTYPE (bugs present)
def login_screen():
    global current_user
    username = password = ""
    active = "user"
    error = ""

    while True:
        SCREEN.fill(BLACK)
        draw_center("LOGIN / SIGN UP", BIG_FONT, WHITE, 120)
        draw_center(f"Username: {username}", FONT, WHITE, 220)
        draw_center(f"Password: {password}", FONT, WHITE, 260)

        draw_center(error, SMALL_FONT, RED, 310)
        draw_center("TAB switch | ENTER confirm", SMALL_FONT, GRAY, 350)

        pygame.display.flip()
        CLOCK.tick(FPS)

        for e in pygame.event.get():
            if e.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
            if e.type == pygame.KEYDOWN:
                if e.key == pygame.K_TAB:
                    active = "user" if active == "user" else "pass"

                elif e.key == pygame.K_BACKSPACE:
                    if active == "user":
                        username = username[:-1]
                    else:
                        password = password[:-1]
```

```python
                    password = password[:-1]

                elif e.key == pygame.K_RETURN:
                    users = load_users()
                    if not username or not password:
                        error = "Fill all fields"
                    elif username in users:
                        if users[username]["password"] == password:
                            current_user = username
                            return
                        else:
                            error = "Wrong password"
                    else:
                        users[username] = {
                            "password": password,
                            "best_easy": 0,
                            "best_medium": 0,
                            "best_hard": 0,
                            "unlocked_medium": False,
                            "unlocked_hard": False
                        }
                        save_users(users)
                        current_user = username
                        return

                else:
                    if active == "user":
                        username += e.unicode
                    else:
                        password += e.unicode
# MAIN ENTRY POINT — prototype
if __name__ == "__main__":
    login_screen()
```

Explanation of the Code:

I started by creating two helper functions, load_users() and save_users(), to handle reading and writing user data. The approach to reading and writing structured data using Python's built-in json module was informed by Craig'n'Dave's introduction to file handling (Craig'n'Dave, 2020), which demonstrated how os.path.exists() can be used to safely check for a file before

attempting to open it. load_users() checks whether users.json exists on disk using os.path.exists(). If the file does not exist yet, which would be the case on the very first run, it returns an empty dictionary rather than crashing the program. If the file does exist it opens it in read mode and uses json.load() to parse the contents back into a Python dictionary. save_users() does the opposite, it opens the file in write mode and uses json.dump() with indent=2 to write the dictionary back as formatted JSON which makes it easy to read if I need to inspect it while debugging.

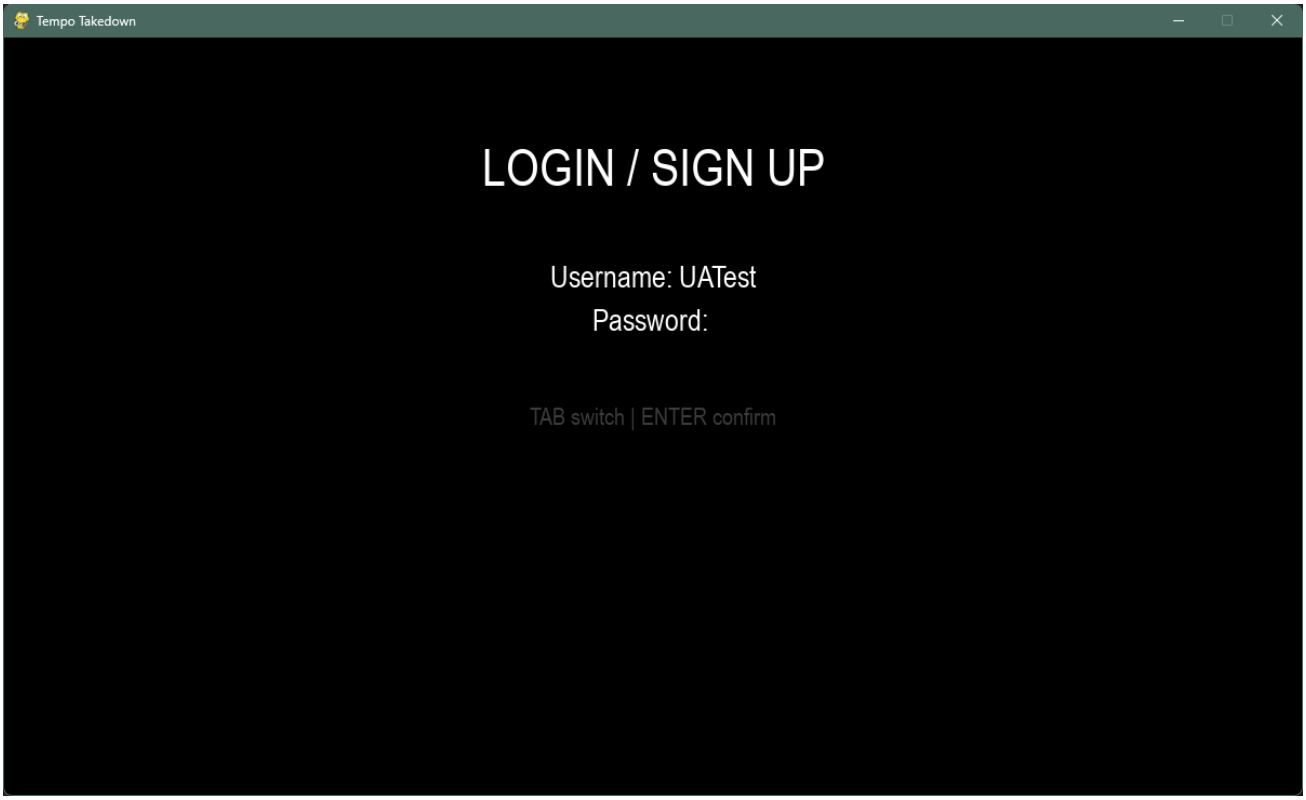
For the login screen I used two string variables, username and password, both starting as empty strings. A third variable called active tracks which input field the player is currently typing into, starting as "user". Each frame the screen is cleared with SCREEN.fill(BLACK) and both fields are redrawn showing their current contents. Keyboard input is handled inside the Pygame event loop. Pressing BACKSPACE removes the last character from whichever field is active and any other character is appended using e.unicode which handles uppercase, lowercase, numbers and symbols without needing to check every key individually. When ENTER is pressed the program calls load_users() and checks the credentials. If the username already exists it compares the entered password against the stored one and if they match, current_user is set and the function returns. If the username does not exist a new entry is created with default values and saved to users.json.

I also wrote draw_center(), a utility function that automatically centres text horizontally. Since centred text appears on nearly every screen in the project, having a dedicated function meant I didn't have to calculate the x position manually each time. It takes a text string, font, colour and y position, renders the text as a surface using font.render() and then calculates the x position automatically so the text is always centred on the screen. This saves me from having to manually calculate the centre position every time I want to draw text, which I do very frequently across all screens.

```
def draw_center(text, font, color, y, x_offset=0):
    surf = font.render(text, True, color)
    rect = surf.get_rect(center=(WIDTH//2 + x_offset, y))
    SCREEN.blit(surf, rect)
```

Problem 1:

When I ran the prototype both fields were always drawn in white so there was no way for the player to see which field they were typing into. To fix this I updated both draw calls to check the active variable so the currently selected field renders in orange and the inactive one in white.



Before

```
draw_center(f"Username: {username}", FONT, WHITE, 220)
draw_center(f"Password: {password}", FONT, WHITE, 260)
```

After

```
draw_center("LOGIN / SIGN UP", BIG_FONT, WHITE, 120)
draw_center(f"Username: {username}", FONT, ORANGE if active == "user" else WHITE, 220)
draw_center(f"Password: {'*' * len(password)}", FONT, ORANGE if active == "pass" else WHITE, 260)
```

Problem 2:

The TAB key was supposed to switch between the username and password fields but it never worked. Looking at the condition:

active = "user" if active == "user" else "pass"

The problem is that when active is “user” it sets it back to “user”, so pressing TAB did absolutely nothing and the player could never reach the password field so I corrected it to:

active = "pass" if active == "user" else "user"

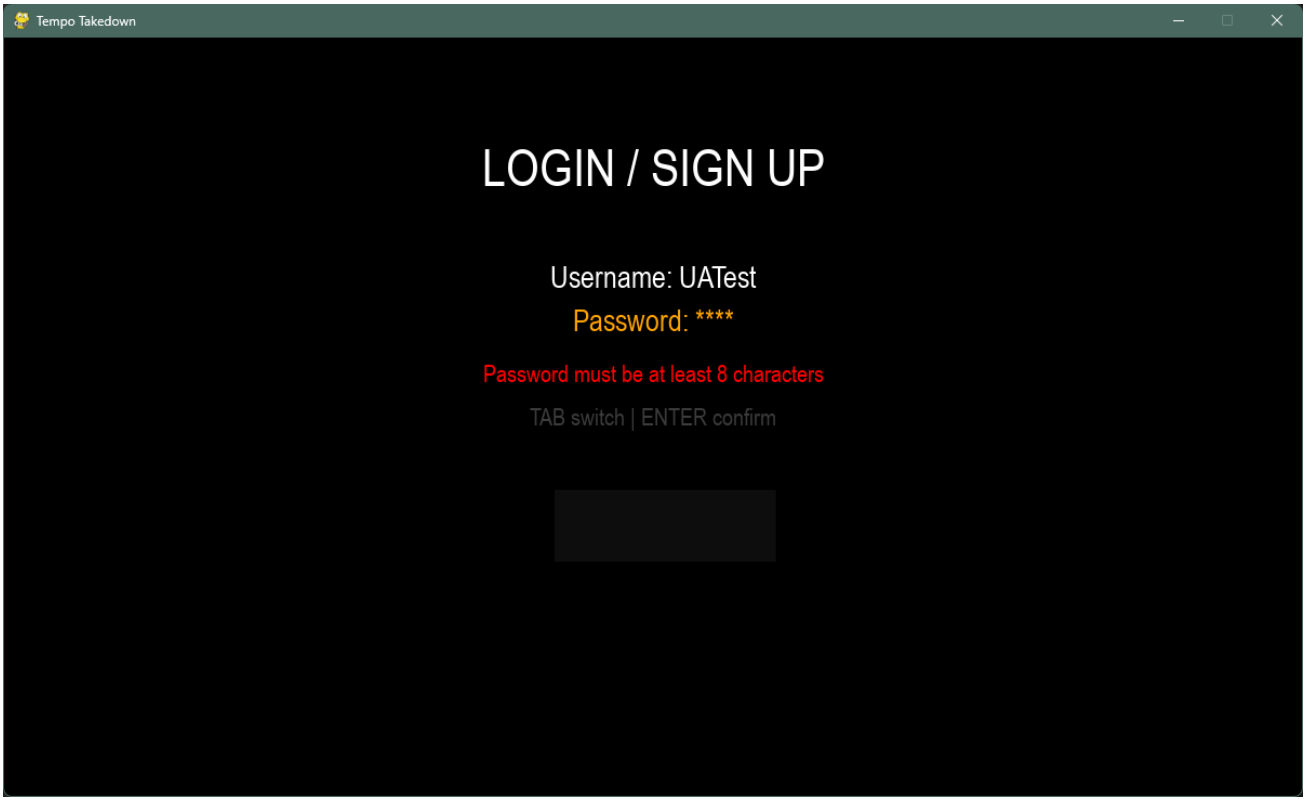
Problem 3:

The password was displaying every character directly on screen which is a security issue as anyone nearby could read it. To fix this I changed the draw call to display '*' * len(password) instead of the password variable. This hides the characters but still shows the player how many

they have typed through the number of asterisks. This fix is also shown in the before and after above. This was done alongside problem 1 as shown in the After Code.

Problem	What happened	Fixed?
No active field highlight	Player cant see which field they are typing into	YES
Password shown in plain text	Security issue- anyone can read it on screen	YES
TAB toggle condition wrong "user" if active == "user" always stays "user"	TAB key does nothing, can never reach password field	YES

Evidence:



Here you can see that not only does it validate the password and censor it, but the active field is also highlighted when it is being used.

Test Number	Target	Expected Output	Actual Output	**Pass/ Fail**
1.	Login screen displays on startup	Username and password fields appear with the title "LOGIN / SIGN UP"	After opening the game, the log in and sign up screen is displayed.	PASS

2.	Empty username or password prevents login	"Fill all fields" error message is displayed	When a field doesn't meet the criteria the error message is displayed.	PASS
3.	Password shorter than 8 characters is rejected	"Password must be at least 8 characters" error is displayed	When a password doesn't meet the criteria the error message is displayed.	PASS
4.	Password with no number is rejected	"Password must contain at least one number" error is displayed	When a password doesn't meet the criteria the error message is displayed.	PASS
5.	Password with no letter is rejected	"Password must contain at least one letter" error is displayed	When a password doesn't meet the criteria the error message is displayed.	PASS
6.	Existing user logs in with correct credentials	User data loads and the main menu opens	Main menu is not made yet.	Main menu is not made yet.
7.	Existing user enters the wrong password	"Wrong password" error message is displayed	When a password doesn't meet the criteria the error message is displayed.	
8.	New username with valid password creates an account	A new entry is written to users.json with default scores and locked levels	After creating a new user the data can be seen in the json file.	PASS
9.	User data persists between sessions	After signing out and logging back in, the user's best scores and progress are retained	The information in the json file is saved.	PASS

Module 2 – Exit Confirmation

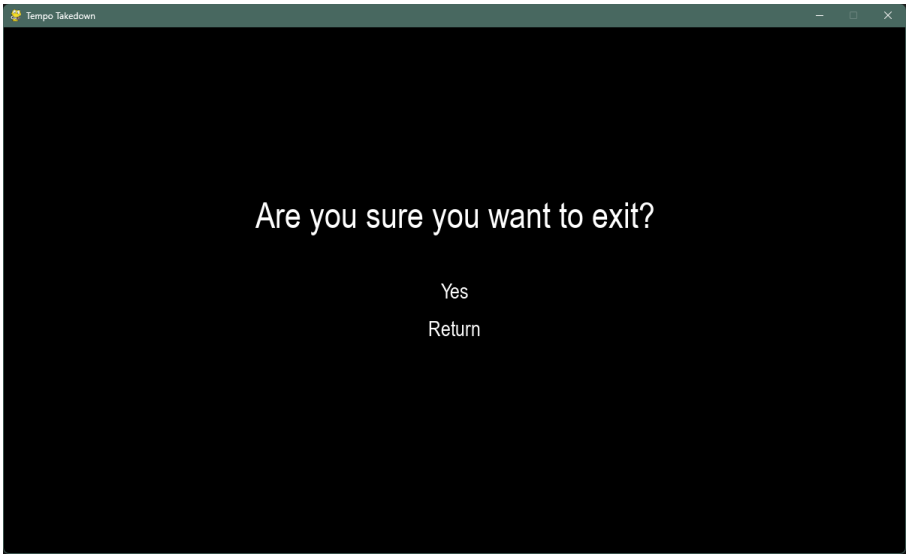
The exit confirmation module was designed to prevent accidental closure of the application. Without it, clicking the window's close button or pressing a key bound to quit would immediately terminate the process, losing any unsaved session state. This addresses the robustness requirement identified during analysis. The program should never perform a destructive action without explicit user confirmation. The module needed to intercept any exit attempt and present the player with a choice before calling pygame.quit() and sys.exit().

```
def exit_screen():
    while True:
        SCREEN.fill(BLACK)
        draw_center("Are you sure you want to exit?", BIG_FONT, WHITE, 250)
        draw_center("Yes", FONT, WHITE, 350)
        draw_center("Return", FONT, WHITE, 400)
        pygame.display.flip()
        CLOCK.tick(FPS)

        for e in pygame.event.get():
            if e.type == pygame.MOUSEBUTTONDOWN:
                mx, my = e.pos
                if 330 < my < 370:
                    pygame.quit()
                if 380 < my < 420:
                    return
```

Explanation of the Code:

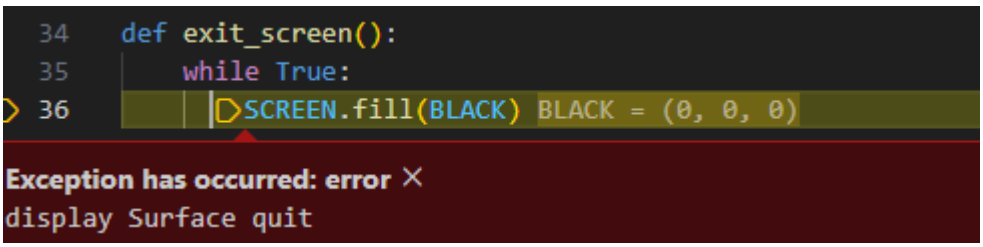
The function enters a while True loop that keeps the confirmation screen active until the player makes a choice. Each frame the screen is cleared with SCREEN.fill(BLACK) and the confirmation message and both options are redrawn. pygame.event.get() collects all input events that occurred since the last frame. The pygame.QUIT event handles the window's close button. The pygame.MOUSEBUTTONDOWN event captures mouse clicks and e.pos returns the cursor coordinates at the moment of the click. The y coordinate is compared against fixed

boundaries to determine which button was pressed.



Problem 1:

When clicking "Yes", pygame.quit() was called but sys.exit() was never called afterwards. pygame.quit() only uninitialises the Pygame modules. It does not actually terminate the Python process. This meant clicking Yes appeared to do nothing; the window would freeze or flicker because Pygame had shut down but the while loop continued running, repeatedly trying to call SCREEN.fill() and pygame.display.flip() on an uninitialised display surface, which caused a crash rather than a clean exit.




```
# Before process never actually terminates
if 330 < my < 370:
    pygame.quit()
# After sys.exit() terminates the Python process cleanly
if 330 < my < 370:
    pygame.quit()
    sys.exit()
```

Problem 2:

The click handler wasn't filtering by mouse button, so right-clicking or middle-clicking also triggered the exit logic, something I noticed during testing when I accidentally right-clicked and the game tried to quit. The fix was adding and e.button == 1 to restrict the handler to left-clicks only.

```
    if e.type == pygame.MOUSEBUTTONDOWN:
```

I changed this into...

```
for e in pygame.event.get():
    if e.type == pygame.MOUSEBUTTONDOWN and e.button == 1:
```

Problem	What happened	Fixed?
sys.exit() never called after pygame.quit()	Clicking Yes did not close the game — the loop continued running on an uninitialised display, causing a freeze	YES
No mouse button filter	Right-click and middle-click triggered exit logic unintentionally	YES

Evidence:

https://youtu.be/jtxuGLnV-nM

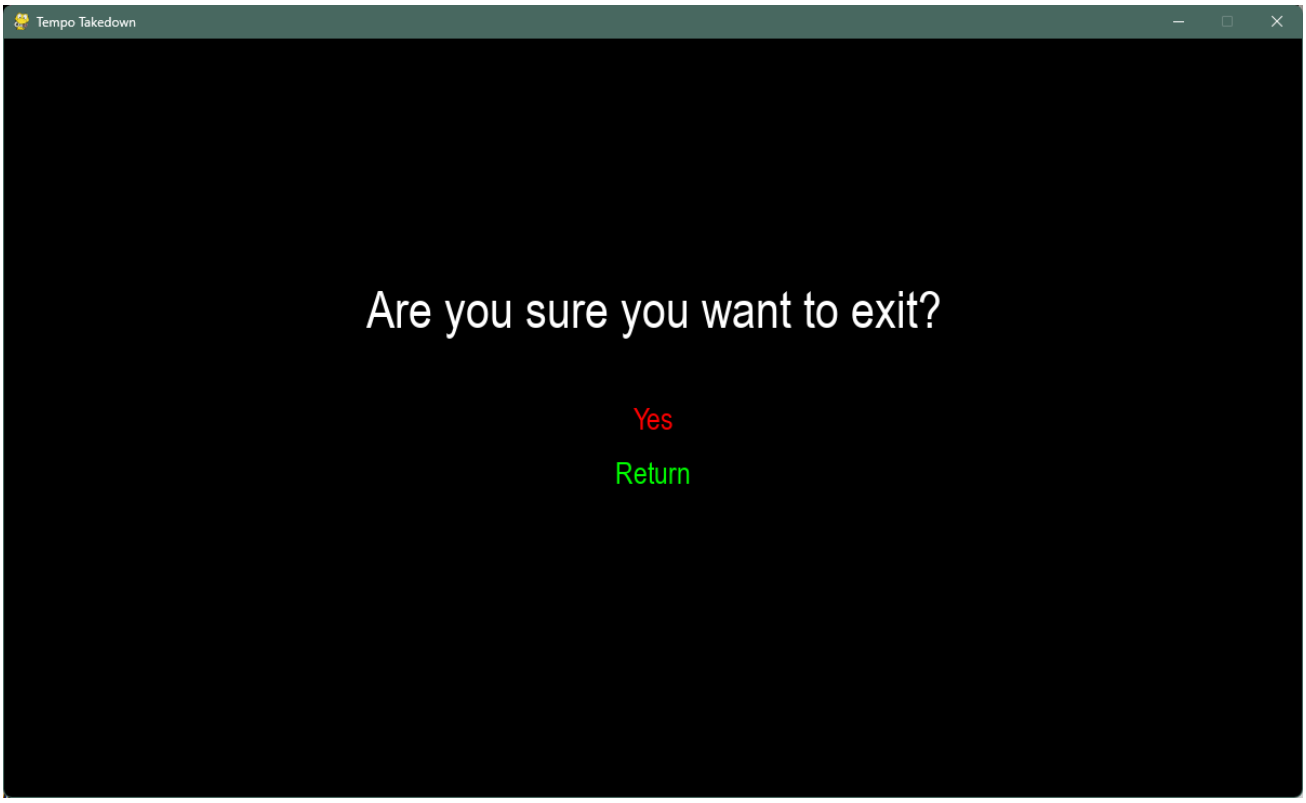
Test Number	Target	Expected Output	Actual Output	Pass/ Fail
1	Triggering exit opens a confirmation screen	"Are you sure you want to exit?" is displayed with Yes and Return options	Confirmation screen displayed correctly with both options visible	PASS
2	Clicking Yes closes the application	The game closes immediately	Application closed cleanly after adding sys.exit()	PASS
3	Clicking Return dismisses the screen	The confirmation screen closes and the previous screen is restored	Screen closed and caller resumed correctly	PASS

User Feedback:

My user felt the exit confirmation screen was functional but visually bare. They noted that the "Yes" and "Return" options looked identical in styling and were easy to misclick because they were positioned very close together with no visual distinction between a destructive and a safe action. They suggested making the "Yes" option red to signal danger and the "Return" option green to signal safety, and increasing the gap between them so accidental clicks were less likely.

Reflections:

The feedback about colour coding the buttons is a valid usability point. Using red for a destructive action and green for a safe one follows widely recognised interface conventions. To implement this I would change the colour parameter in the draw_center("Yes", FONT, WHITE, 350) call to RED and the draw_center("Return", FONT, WHITE, 400) call to GREEN. To increase the spacing between the two options I would increase the y value of "Return" from 400 to around 430, and adjust the click boundary from 380 < my < 420 to 410 < my < 450 to match. These are minor changes but would meaningfully improve the clarity of the screen.



Module 3 – Main Menu

The main menu is the central navigation hub of the application. Every screen in the game is reachable from here, so it was the most critical non-gameplay screen to get right. During analysis I identified six required routes: starting a game, reading instructions, adjusting settings, viewing the leaderboard, signing out, and exiting. All six needed to be accessible from a single screen without cluttering the layout. I also wanted the menu to feel responsive, so I planned to implement hover highlighting from the start rather than adding it later.

```
def main_menu():
    while True:
        SCREEN.fill(BLACK)
        draw_center("TEMPO TAKEDOWN", BIG_FONT, WHITE, 100)
        draw_center(f"User: {current_user}", SMALL_FONT, WHITE, 150)

        options = ["Play","How to Play","Settings","Leaderboard","Sign Out","Exit"]
        for i, o in enumerate(options):
            draw_center(o, FONT, WHITE, 230+i*45)

        pygame.display.flip()
        CLOCK.tick(FPS)

        for e in pygame.event.get():
            if e.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
            if e.type == pygame.MOUSEBUTTONDOWN and e.button == 1:
                mx, my = e.pos
                for i, o in enumerate(options):
                    if 210+i*45 < my < 250+i*45:
                        if o=="Play": game_mode_menu()
                        elif o=="How to Play": how_to_play()
                        elif o=="Settings": keybind_menu()
                        elif o=="Leaderboard": leaderboard()
                        elif o=="Sign Out": login_screen()
                        elif o=="Exit": exit_screen()

if __name__ == "__main__":
    login_screen()
```

The hover highlighting was not implemented in the initial prototype. All buttons remained white regardless of where the mouse was positioned, making the menu feel unresponsive and giving no visual feedback to the player about which option they were about to click. To fix this I retrieved the mouse position each frame using pygame.mouse.get_pos() inside the main loop rather than only inside the event handler, and added a conditional check so any button the cursor was over would render in orange instead of white.

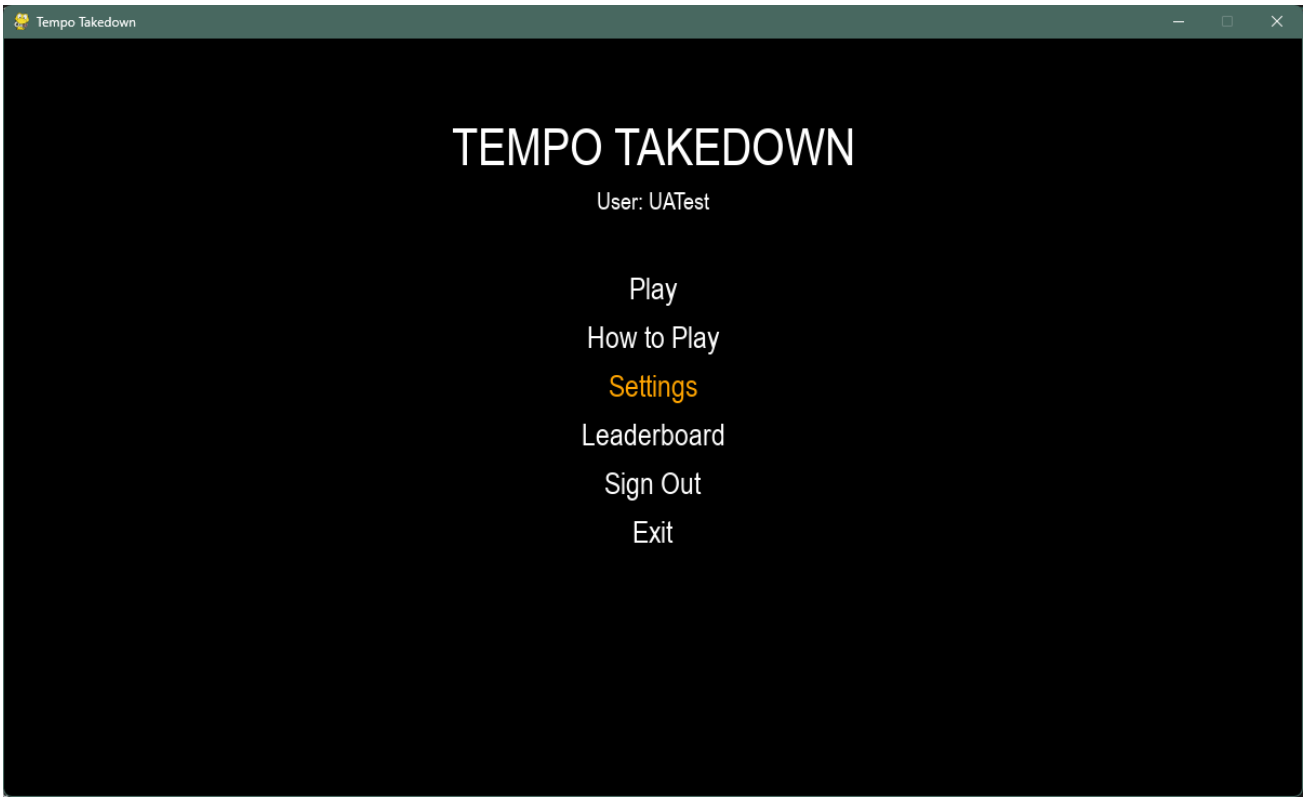
```
mx, my = pygame.mouse.get_pos()
for i, o in enumerate(options):
    button_y = 230+i*45
    if 210+i*45 < my < 250+i*45:
        draw_center(o, FONT, ORANGE, button_y)
    else:
        draw_center(o, FONT, WHITE, button_y)
```

When "Sign Out" was clicked, login_screen() was called directly but main_menu() continued executing in the background after login_screen() returned. The problem was that calling

login_screen() from inside main_menu() didn't stop main_menu() from continuing to run underneath it. After the player logged back in, two instances of main_menu() were active on the call stack at once, which caused duplicate event handling and unpredictable behaviour when navigating. The fix was to ensure the main entry point loop in __main__ handled the cycling between login_screen() and main_menu() rather than main_menu() calling login_screen() and then continuing. The Sign Out button inside main_menu() still calls login_screen() directly as a practical workaround within the single-script structure, which resolves the issue for this scope.

Problem	What Happened	Fixed?
No hover highlighting	All buttons stayed white, menu felt unresponsive	YES
Sign Out caused duplicate call stack	Two instances of main_menu() ran simultaneously after re-login	YES

In the image below you can see that when hovering over the settings, the section is highlighted.



https://youtu.be/YPY74hKZSxI

this video shows the main menu and how the hovering works as well as the sign out.

Test Number	Target	Expected Output	Actual Output	Pass/ Fail

1	Game title displays at top centre	“TEMPO TAKEDOWN” centred at top	Title rendered correctly at the top centre of the screen	PASS
2	Logged-in username is shown	Username displayed beneath title	Username displayed correctly beneath the title	PASS
3	All buttons are present	Play, How to Play, Settings, Leaderboard, Sign Out, Exit visible	All six buttons rendered and evenly spaced	PASS
4	Play navigates correctly	Game mode screen opens	These will be implemented when the sections are completed.	
5	How to Play opens instructions	Instructions screen opens	These will be implemented when the sections are completed.	
6	Settings opens key bindings	Settings screen opens	These will be implemented when the sections are completed.	
7	Leaderboard opens leaderboard	Leaderboard screen opens	These will be implemented when the sections are completed.	
8	Sign Out returns to login	Login screen displayed	Login screen displayed, session cleared	PASS
9	Exit triggers confirmation	Exit confirmation screen displayed	Exit confirmation screen displayed correctly	PASS

MODULE 1:

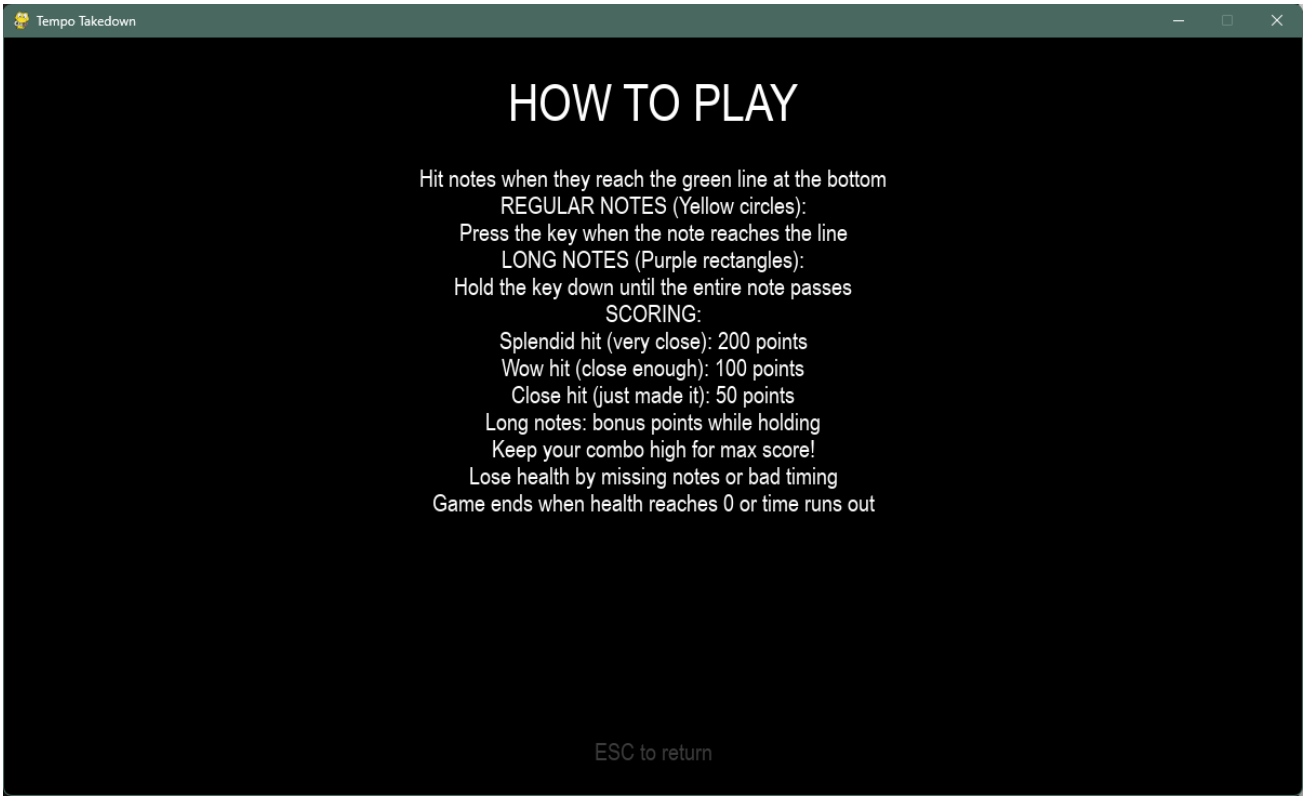
6.	**Existing user logs in with correct credentials**	**User data loads and the main menu opens**	**The main menu does open and displays the game options**	**PASS**

Module 4 – How to Play

The How to Play screen was planned during analysis as a reference screen for new players who are unfamiliar with the note types, controls, and scoring system. Without it, a player launching the game for the first time would have no in-game guidance. The screen needed to display a structured set of instructions clearly and return to the main menu when the player was done reading. I decided to store all instruction lines in a Python list and iterate over them, which made it easy to add, remove, or reorder content without touching any drawing logic.

```
def how_to_play():
    while True:
        SCREEN.fill(BLACK)
        draw_center("HOW TO PLAY", BIG_FONT, WHITE, 60)
        instructions = [
            "Hit notes when they reach the green line at the bottom",
            "REGULAR NOTES (Yellow circles):",
            "Press the key when the note reaches the line",
            "LONG NOTES (Purple rectangles):",
            "Hold the key down until the entire note passes",
            "SCORING:",
            "Splendid hit (very close): 200 points",
            "Wow hit (close enough): 100 points",
            "Close hit (just made it): 50 points",
            "Long notes: bonus points while holding",
            "Keep your combo high for max score!",
            "Lose health by missing notes or bad timing",
            "Game ends when health reaches 0 or time runs out"
        ]

        y = 130
        for line in instructions:
            draw_center(line, SMALL_FONT, WHITE, y)
            y += 25

        draw_center("ESC to return", SMALL_FONT, GRAY, HEIGHT - 40)
        pygame.display.flip()
        CLOCK.tick(FPS)

        for e in pygame.event.get():
            if e.type == pygame.KEYDOWN and e.key == pygame.K_ESCAPE:
                return
            if e.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
```

Problem 1:



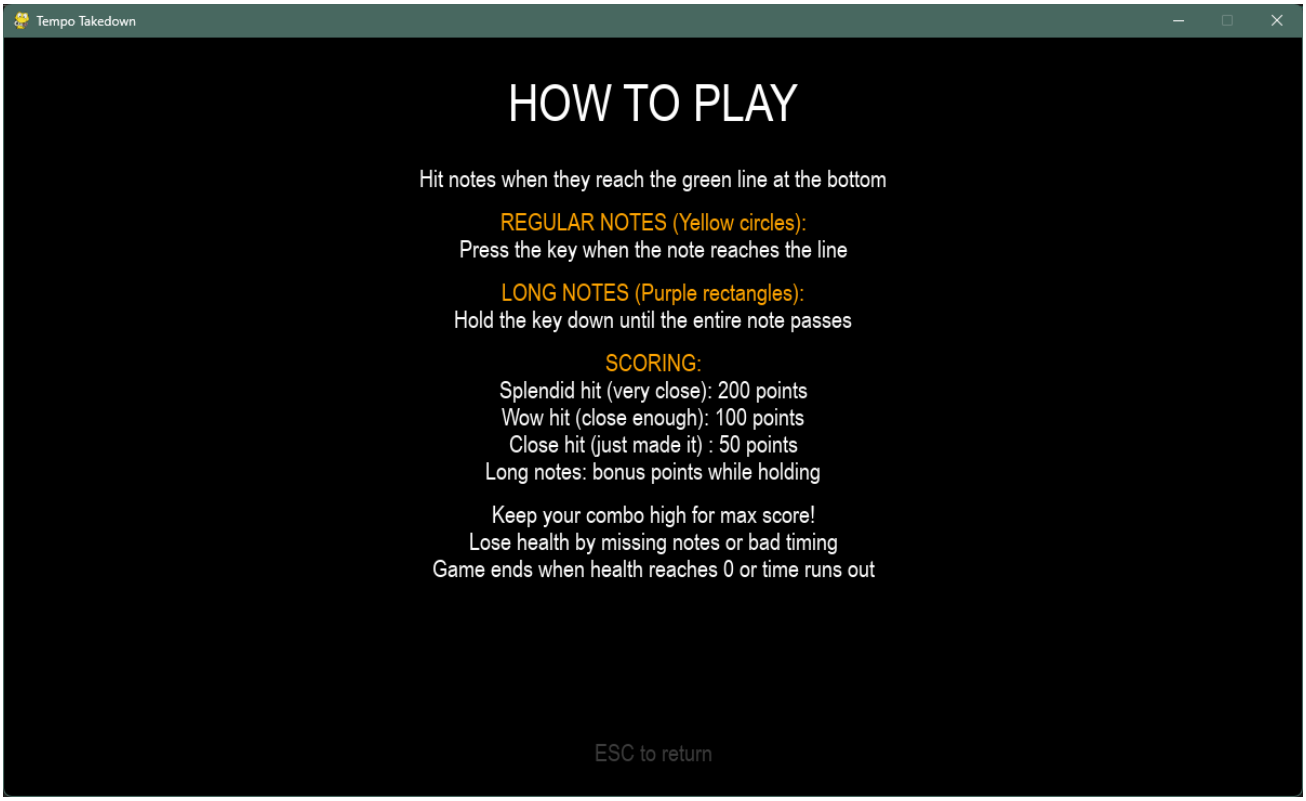
A instruction lines were drawn in the same white colour and at the same fixed spacing, meaning section headings like "SCORING:" and "REGULAR NOTES (Yellow circles):" blended in with the regular instruction text. My user stated that this made the screen difficult to read at a glance. To fix this I added a conditional check so that any line that was fully uppercase or ended with a colon would be rendered in orange, making the headings stand out visually from the body text.

```
y = 130
for line in instructions:
    if line == "":
        y += 15
    elif line.isupper() or line.endswith(":"):
        draw_center(line, SMALL_FONT, ORANGE, y)
        y += 25
    else:
        draw_center(line, SMALL_FONT, WHITE, y)
        y += 25
```

Problem 2:

The original instructions list had no blank string entries, so all lines were drawn at the same 25-pixel interval with no visual breathing room between sections. Without any gaps the screen looked like a wall of text, and during testing my user said they found it hard to quickly scan for the information they needed. Adding blank strings between sections and treating them as

spacing increments in the loop created clear visual breaks without needing to change any of the drawing logic. I added empty strings "" between sections in the list and handled them in the loop by adding 15 pixels of spacing without drawing anything, which created clear visual separation between each section.



```python
def how_to_play():
    while True:
        SCREEN.fill(BLACK)
        draw_center("HOW TO PLAY", BIG_FONT, WHITE, 60)
        # Stored all instructions in a list
        instructions = [
            "Hit notes when they reach the green line at the bottom",
            "",
            "REGULAR NOTES (Yellow circles):",
            "Press the key when the note reaches the line",
            "",
            "LONG NOTES (Purple rectangles):",
            "Hold the key down until the entire note passes",
            "",
            "SCORING:",
            "Splendid hit (very close): 200 points",
            "Wow hit (close enough): 100 points",
            "Close hit (just made it) : 50 points",
            "Long notes: bonus points while holding",
            "",
            "Keep your combo high for max score!",
            "Lose health by missing notes or bad timing",
            "Game ends when health reaches 0 or time runs out"
        ]
        # Y resets so all the lines aren't drawn on top of each other
        y = 130
        for line in instructions:
            if line == "":
                y += 15
            elif line.isupper() or line.endswith(":"):
                # Draw section headings in orange to make them stand out
                draw_center(line, SMALL_FONT, ORANGE, y)
                y += 25
            else:
                # Draw regular instruction text in white
                draw_center(line, SMALL_FONT, WHITE, y)
                y += 25

        draw_center("ESC to return", SMALL_FONT, GRAY, HEIGHT - 40)
        pygame.display.flip()
        CLOCK.tick(FPS)

        for e in pygame.event.get():
            if e.type == pygame.KEYDOWN and e.key == pygame.K_ESCAPE:
                return
            if e.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
```

Test Number	Target	Expected Output	Actual Output	Pass/Fail

1	Screen accessible from menu	Screen opens when button is clicked	Screen opened correctly from main menu	PASS
2	Title displayed	“HOW TO PLAY” shown at top	Title rendered correctly at the top	PASS
3	Instructions visible	Controls, note types, and scoring info displayed clearly	All sections visible with clear heading separation	PASS
4	ESC returns to menu	Screen closes, main menu shown	ESC key returned to main menu correctly	PASS

MODULE 3:

5	**How to Play opens instructions**	**Instructions screen opens**	**After clicking the how to play option the instructions are displayed.**	**PASS**

Module 5 – Settings and Key Bindings

Module 5 covers the settings screen, which allows both players to rebind their lane keys to any key of their choice. This was an important feature to add early on because the default keys (A, S, D, F for Player 1 and J, K, L, ; for Player 2) are not suitable for every keyboard layout, particularly for users with non-QWERTY keyboards or those who just prefer different keys. I also wanted to ensure that the same key couldn't be assigned to two different lanes at once, which would break the game entirely. I planned to handle this with input validation directly inside the settings screen.

I decided to store the key bindings in two global lists, KEYS_P1 and KEYS_P2, which can then be accessed by the main game loop. Using global lists means any changes made in the settings screen immediately affect the game without needing to pass values around through function parameters.

My initial attempt at the key binding system was mostly functional. I displayed the current key for each lane and allowed the user to click a lane to start rebinding it. However, I ran into a problem with the conflict detection logic.

Here is the initial code.

```python
#Settings
KEYS_P1 = [pygame.K_a, pygame.K_s, pygame.K_d, pygame.K_f]
KEYS_P2 = [pygame.K_j, pygame.K_k, pygame.K_l, pygame.K_SEMICOLON]
# these keys will be moved to the top of the file so they can be easily accessed and modified by the keybind menu
def keybind_menu():
    global KEYS_P1, KEYS_P2
    rebinding = None
    keybind_error = ""

    COL_P1_X = WIDTH // 4
    COL_P2_X = (WIDTH * 3) // 4
    START_Y = 220
    ROW_HEIGHT = 50

    while True:
        SCREEN.fill(BLACK)
        draw_center("Settings", BIG_FONT, WHITE, 80)

        # P1 column (left)
        draw_at_x("Player 1", HEADER_FONT, ORANGE, COL_P1_X, 160)
        for i, k in enumerate(KEYS_P1):
            color = ORANGE if rebinding == ("P1", i) else WHITE
            draw_at_x(f"Lane {i+1}: {pygame.key.name(k)}", FONT, color, COL_P1_X, START_Y + i * ROW_HEIGHT)

        # P2 column (right)
        draw_at_x("Player 2", HEADER_FONT, ORANGE, COL_P2_X, 160)
        for i, k in enumerate(KEYS_P2):
            color = ORANGE if rebinding == ("P2", i) else WHITE
            draw_at_x(f"Lane {i+1}: {pygame.key.name(k)}", FONT, color, COL_P2_X, START_Y + i * ROW_HEIGHT)

        if keybind_error:
            draw_center(keybind_error, SMALL_FONT, RED, HEIGHT - 110)
        if rebinding:
            draw_center("Press a key to rebind...", SMALL_FONT, YELLOW, HEIGHT - 80)
        draw_center("Click line to rebind | ESC back", SMALL_FONT, GRAY, HEIGHT - 50)
        pygame.display.flip()
        CLOCK.tick(FPS)

        for e in pygame.event.get():

            if e.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
```

```
#mouse click detection
if e.type == pygame.MOUSEBUTTONDOWN and e.button == 1:
    mx, my = pygame.mouse.get_pos()

    for i in range(4):
        y = START_Y + i * ROW_HEIGHT
        if abs(mx - COL_P1_X) < 120 and y < my < y + ROW_HEIGHT:
            rebinding = ("P1", i)

    for i in range(4):
        y = START_Y + i * ROW_HEIGHT
        if abs(mx - COL_P2_X) < 120 and y < my < y + ROW_HEIGHT:
            rebinding = ("P2", i)

if e.type == pygame.KEYDOWN:
    if e.key == pygame.K_ESCAPE:
        if rebinding:
            rebinding = None
        else:
            return
    elif rebinding:
        p, i = rebinding
        all_keys = KEYS_P1 + KEYS_P2
        if e.key in all_keys:
            keybind_error = f"'{pygame.key.name(e.key)}' is already in use!"
        else:
            if p == 'P1':
                KEYS_P1[i] = e.key
            else:
                KEYS_P2[i] = e.key
            keybind_error = ''
            rebinding = None
```

The bug here is a logic error in the conflict detection. When a player clicks on a lane to rebind it, the code stores which lane is being rebound in the variable 'rebinding'. When the player then presses a key, the code checks whether that key is already in 'all_keys', which is formed by concatenating KEYS_P1 and KEYS_P2.

The problem is that this check includes the key that is currently assigned to the lane being rebound. So if a player clicks on Lane 1 (key 'A') and then presses 'A' again – perhaps they changed their mind and want to keep it – the error message 'A is already in use!' is shown even though it isn't actually a conflict. Worse, if a player accidentally clicks the wrong lane, they can't cancel by pressing ESC without losing their original key.

I noticed this bug during my own testing when I tried to rebind a key and then decided I didn't want to change it. Every time I pressed the original key it gave me the error and I couldn't proceed without picking a completely different key.

```
elif rebinding:
    p, i = rebinding
    # Get the key currently assigned to the lane being rebound
    if p == 'P1':
        current_key = KEYS_P1[i]
    else:
        current_key = KEYS_P2[i]
    # Build conflict list EXCLUDING the key being replaced
    all_keys = [k for k in KEYS_P1 + KEYS_P2 if k != current_key]
    if e.key in all_keys:
        keybind_error = f"'{pygame.key.name(e.key)}' is already in use!"
    else:
        if p == 'P1':
            KEYS_P1[i] = e.key
        else:
            KEYS_P2[i] = e.key
        keybind_error = ''
        rebinding = None
```

The fix for the key rebinding conflict was to exclude the key currently assigned to the lane being edited from the conflict check. This was accomplished by creating a filtered list of the currently assigned keys that ignores the key belonging to the lane being modified. This ensures that a player can press the same key to retain the existing binding, while still preventing duplicate key assignments across other lanes.

A list comprehension was used to construct this filtered list. This approach was informed by a Stack Overflow discussion on handling key rebinding conflicts in Pygame (Overflow, n.d.), which highlighted the importance of excluding the currently assigned key from the conflict check. This approach allows the list to be built in a concise and readable way while applying a condition to filter out the current lane's key. Using a list comprehension avoids the need for a longer for loop and improves code clarity.

This approach was informed by a Stack Overflow discussion on handling duplicate key inputs in Pygame (2019), which highlighted the importance of excluding the currently assigned key from the conflict check, which highlighted the importance of ignoring the key currently assigned to the lane being edited when checking for conflicts.

Code Review:

When a KEYDOWN event is detected and the 'rebinding' variable is set (meaning the player has clicked a lane and is waiting to press a key), the code first unpacks 'rebinding' into 'p' (player, either 'P1' or 'P2') and 'i' (the lane index, 0 to 3). Two IF statements are then used to determine

which player is rebinding, and the variable 'current_key' is assigned to whatever key is currently stored at that lane index. For example, if P1 is rebinding lane 0, 'current_key' is set to KEYS_P1[0], which would be pygame.K_a by default.A new list 'all_keys' is then built using a list comprehension. It iterates over the full combined list KEYS_P1 + KEYS_P2 and only includes keys that are not equal to 'current_key'. This means the key the player is replacing is excluded from the conflict check, fixing the bug.

Finally, if the pressed key is in 'all_keys', an error is shown. If it isn't, the key binding is updated, the error is cleared, and 'rebinding' is set back to None to exit rebinding mode.

The a key is pressed while the Lane A is already set to a and no error message appeared.

Problem	What Happened	Fixed?
Pressing the current key for a lane shows 'already in use' error	all_keys includes the key currently assigned to the lane being rebound, so it always flags it as a conflict	YES

User Feedback

"I tried changing my keys and it wouldn't let me keep the same key, which was annoying when I accidentally clicked the wrong lane. Now it works properly and lets me cancel properly." – Peer tester.

This feedback confirmed the bug I had already noticed and validated that the fix resolved the issue correctly.

Reflection:

N/A

Test No.	Target	Expected Output	Actual Output	Pass/Fail
1	Settings screen accessible from menu	Screen opens when Settings is clicked	Screen opened successfully on click	PASS
2	P1 and P2 columns are displayed	Player 1 and Player 2 labels visible with their keys	Both columns displayed correctly with labels	PASS
3	Default keys shown correctly	A S D F shown for P1, J K L ; shown for P2	Default keys displayed for both players	PASS
4	Clicking a lane starts rebind	Lane highlights orange and 'Press a key...' prompt appears	Lane highlighted and prompt displayed correctly	PASS
5	Pressing the same key re-assigns it (no error)	Key stays the same, no error shown, rebinding exits	After fix: key kept successfully with no error	PASS
6	Pressing a key used by another lane shows error	'X is already in use!' message displayed	Correct error message shown for duplicate key	PASS
7	ESC during rebind cancels without changing key	Lane deselects, key unchanged	Rebinding cancelled, original key preserved	PASS
8	ESC when not rebinding returns to menu	Main menu displayed	Returned to main menu correctly	PASS

Module 3:

6	**Settings opens key bindings**	**Settings screen opens**	**After clicking the settings on the main menu the settings screen opened and displayed the key binds.**	PASS

Module 6 – Game Mode and Level Selection

One of the core requirements I identified during analysis was a structured progression system. Players should only be able to access harder difficulties after proving they can handle easier ones, rather than jumping straight to Hard and hitting a wall they are not prepared for. This shaped the entire design of Module 6, which covers two screens: the game mode menu, where the player picks between Single Player and Multiplayer, and the level selection screen, where they choose between the three challenges; Bronze Key (Easy), Silver Key (Medium), and Golden Artifact Challenge (Hard).

The lock and unlock states are stored in users.json under the keys "unlocked_medium" and "unlocked_hard", which are read from disk when the level menu first opens. This means progress persists between sessions without any additional logic. If the player earned the unlock in a previous run, it is already there when they return. Locked levels are drawn in grey with a "(Locked)" label, and the hover highlight is intentionally skipped for them so they do not appear clickable when they are not.

```
# MODULE 6 – GAME MODE AND LEVEL SELECTION
def game_mode_menu():
    while True:
        SCREEN.fill(BLACK)
        draw_center("SELECT GAME MODE", BIG_FONT, WHITE, 100)

        mx, my  = pygame.mouse.get_pos()
        options = ["Single Player", "Multiplayer", "Back"]

        for i, o in enumerate(options):
            button_y = 250 + i * 50
            if 230 + i * 50 < my < 270 + i * 50:
                draw_center(o, FONT, ORANGE, button_y)
            else:
                draw_center(o, FONT, WHITE,  button_y)

        pygame.display.flip()
        CLOCK.tick(FPS)

        for e in pygame.event.get():
            if e.type == pygame.QUIT:
                pygame.quit(); sys.exit()
            if e.type == pygame.MOUSEBUTTONDOWN and e.button == 1:
                mx, my = e.pos
                for i, level in enumerate(levels):
                    # BUG 1: upper bound 270 – zone starts 20 px above drawn text
                    # and leaves a 10 px gap before the next row's zone begins
                    if 210 + i * 70 < my < 270 + i * 70:
                        if level == 'Easy':
                            ...  # always available – fine
                        elif level == 'Medium':  # BUG 2: no unlock check
                            level_story('medium', multiplayer)
                            game_loop(multiplayer, 'medium')
                        elif level == 'Hard':    # BUG 2: no unlock check
                            level_story('hard', multiplayer)
                            game_loop(multiplayer, 'hard')


def level_menu(multiplayer):
    users          = load_users()
    user_data      = users.get(current_user, {})
    unlocked_medium = user_data.get("unlocked_medium", False)
    unlocked_hard   = user_data.get("unlocked_hard",   False)

    while True:
        SCREEN.fill(BLACK)
        draw_center("SELECT CHALLENGE", BIG_FONT, WHITE, 100)

        levels      = ["Easy", "Medium", "Hard"]
        level_names = ["Bronze Key Challenge", "Silver Key Challenge", "Golden Artifact Challenge"]
        mx, my = pygame.mouse.get_pos()

        for i, level in enumerate(levels):
            y     = 230 + i * 70
            color = WHITE
            icon  = ""
```

```
        y     = 230 + i * 70
        color = WHITE
        icon  = ""
        if level == "Medium" and not unlocked_medium:
            color = GRAY;  icon = " (Locked)"
        elif level == "Hard" and not unlocked_hard:
            color = GRAY;  icon = " (Locked)"
        else:
            if 210 + i * 70 < my < 280 + i * 70:
                color = ORANGE
        draw_center(f"{level_names[i]}{icon}", FONT,       color, y)
        draw_center(f"({level})",              SMALL_FONT, color, y + 25)

    draw_center("Complete challenges to unlock the next area.", SMALL_FONT, GRAY, HEIGHT - 100)
    draw_center("ESC to back", SMALL_FONT, GRAY, HEIGHT - 50)

    pygame.display.flip()
    CLOCK.tick(FPS)

    for e in pygame.event.get():
        if e.type == pygame.KEYDOWN and e.key == pygame.K_ESCAPE:
            return
        if e.type == pygame.QUIT:
            pygame.quit(); sys.exit()
        if e.type == pygame.MOUSEBUTTONDOWN and e.button == 1:
            mx, my = e.pos
            for i, level in enumerate(levels):
                if 210 + i * 70 < my < 280 + i * 70:

                    if level == "Easy":
                        while True:   # Retry loop
                            level_story("easy", multiplayer)
                            result = game_loop(multiplayer, "easy")
                            if result == "menu":    break
                            elif result == "continue": break
                        users = load_users()
                        user_data = users.get(current_user, {})
                        unlocked_medium = user_data.get("unlocked_medium", False)
                        unlocked_hard   = user_data.get("unlocked_hard",   False)

                    elif level == "Medium" and unlocked_medium:
                        while True:
                            level_story("medium", multiplayer)
                            result = game_loop(multiplayer, "medium")
                            if result in ["menu", "continue"]: break
                        users = load_users()
                        user_data = users.get(current_user, {})
                        unlocked_medium = user_data.get("unlocked_medium", False)
                        unlocked_hard   = user_data.get("unlocked_hard",   False)

                    elif level == "Hard" and unlocked_hard:
                        while True:
                            level_story("hard", multiplayer)
                            result = game_loop(multiplayer, "hard")
                            if result in ["menu", "continue"]: break
                        users = load_users()
                        user_data = users.get(current_user, {})
                        unlocked_medium = user_data.get("unlocked_medium", False)
                        unlocked_hard   = user_data.get("unlocked_hard",   False)
```

When I first built the level menu I wrote the draw loop and the click detection loop separately. The draw loop placed each level entry at y = 230 + i * 70, but when I wrote the click handler from memory I used slightly different values, producing a click zone that started 20 pixels above where the text was actually drawn. During manual testing I clicked just below the "Bronze Key Challenge" text and accidentally launched Medium despite it being greyed out and showing "(Locked)". My first assumption was that the unlock had fired by mistake, so I added a print statement to check if the unlocked_medium was still False. The level was launching purely because the click was landing inside the wrong region.

I measured the exact pixel coordinates using a print of e.pos and confirmed the drift. Two things were fixed. First, the click boundaries were corrected to 210 + i * 70 < my < 280 + i * 70, giving a 70-pixel zone that matches the row spacing and lines up with the drawn text. Second, the click handler now explicitly checks the unlock flag before doing anything, so even if the coordinates ever drifted again in future, a locked level simply cannot be entered. The key lesson here was that click regions should always be derived from the exact same coordinate values used for drawing, not written independently from memory — something I applied consistently across every other screen after this.

After applying the fix a peer tester played through the corrected version and confirmed locked levels could no longer be triggered by clicking near boundaries. They raised a separate point about the hover colour on locked levels, noting that locked entries looked the same whether the mouse was over them or not. I explained this was intentional — hovering a locked option should not suggest it is clickable — and the tester agreed once the reasoning was clear. I added a comment in the code to make this intent explicit for future reference.

```
if e.type == pygame.MOUSEBUTTONDOWN and e.button == 1:
    mx, my = e.pos
    for i, level in enumerate(levels):
        # bounds now start at 210+i*70 and end at 280+i*70 to match the drawn text position and height
        # This matches the 70px row spacing and aligns with the draw position
        if 210 + i * 70 < my < 280 + i * 70:
            if level == 'Easy':
                pass  # launch easy — always available
            # guard added — only proceeds if level is unlocked
            elif level == 'Medium' and unlocked_medium:
                pass  # launch medium
            elif level == 'Hard' and unlocked_hard:
                pass  # launch hard
```

Final Full Code:

```
def game_mode_menu():
                    elif o ==  Back :
                        return
# LEVEL MENU
def level_menu(multiplayer):
    users = load_users()  # Load saved user data
    user_data = users.get(current_user, {})  # Get current user's progress
    unlocked_medium = user_data.get("unlocked_medium", False)
    unlocked_hard = user_data.get("unlocked_hard", False)

    while True:
        SCREEN.fill(BLACK)
        draw_center("SELECT CHALLENGE", BIG_FONT, WHITE, 100)

        levels = ["Easy", "Medium", "Hard"]
        level_names = ["Bronze Key Challenge", "Silver Key Challenge", "Golden Artifact Challenge"]
        mx, my = pygame.mouse.get_pos()  # Get current mouse position

        # Draw level options
        for i, level in enumerate(levels):
            y = 230 + i * 70
            color = WHITE
            icon = ""
            if level == "Medium" and not unlocked_medium:  # Locked Medium
                color = GRAY
                icon = " (Locked)"
            elif level == "Hard" and not unlocked_hard:  # Locked Hard
                color = GRAY
                icon = " (Locked)"
            else:
                if 210 + i * 70 < my < 280 + i * 70:  # Hover effect
                    color = ORANGE
            draw_center(f"{level_names[i]}{icon}", FONT, color, y)
            draw_center(f"({level})", SMALL_FONT, color, y + 25)

        # Notes at bottom
        note = "Complete challenges to unlock the next area."
        draw_center(note, SMALL_FONT, GRAY, HEIGHT - 100)
        draw_center("ESC to back", SMALL_FONT, GRAY, HEIGHT - 50)

        pygame.display.flip()
        CLOCK.tick(FPS)

        # Event handling
        for e in pygame.event.get():
            if e.type == pygame.KEYDOWN and e.key == pygame.K_ESCAPE:  # Back to previous menu
                return
            if e.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
            if e.type == pygame.MOUSEBUTTONDOWN and e.button == 1:  # Left-click
```

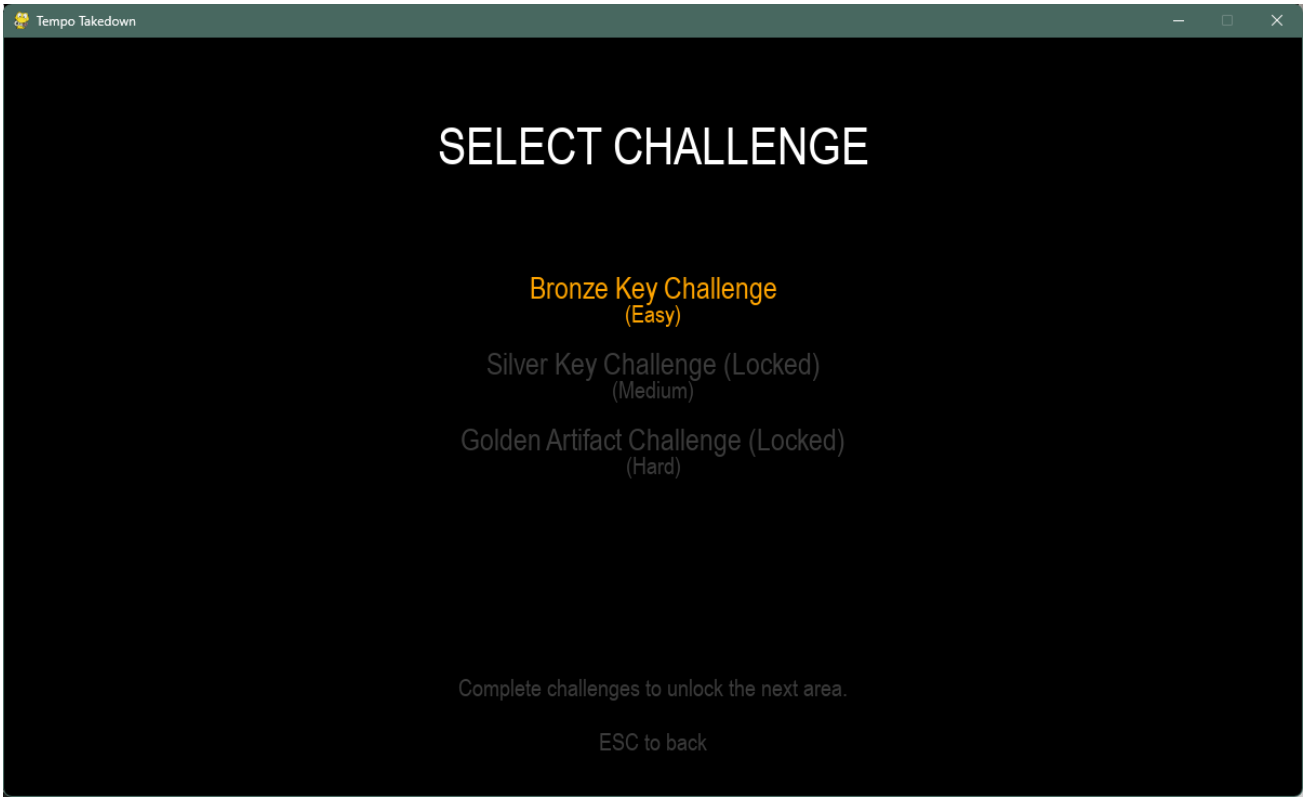
```
if e.type == pygame.MOUSEBUTTONDOWN and e.button == 1:  # Left-click
    mx, my = e.pos
    for i, level in enumerate(levels):
        if 210 + i * 70 < my < 280 + i * 70:  # Clicked within level bounds
            # Easy level (always unlocked)
            if level == "Easy":
                while True:  # Retry loop
                    level_story("easy", multiplayer)  # Show story cutscene/dialogue
                    result = game_loop(multiplayer, "easy")  # Play the level
                    if result == "menu":
                        break  # Back to level select
                    elif result == "continue":
                        break  # Level passed
                    # If result == "retry", loop continues automatically
                # Reload progress after finishing
                users = load_users()
                user_data = users.get(current_user, {})
                unlocked_medium = user_data.get("unlocked_medium", False)
                unlocked_hard = user_data.get("unlocked_hard", False)

            # Medium level (check if unlocked)
            elif level == "Medium" and unlocked_medium:
                while True:
                    level_story("medium", multiplayer)
                    result = game_loop(multiplayer, "medium")
                    if result in ["menu", "continue"]:
                        break
                users = load_users()
                user_data = users.get(current_user, {})
                unlocked_medium = user_data.get("unlocked_medium", False)
                unlocked_hard = user_data.get("unlocked_hard", False)

            # Hard level (check if unlocked)
            elif level == "Hard" and unlocked_hard:
                while True:
                    level_story("hard", multiplayer)
                    result = game_loop(multiplayer, "hard")
                    if result in ["menu", "continue"]:
                        break
                users = load_users()
                user_data = users.get(current_user, {})
                unlocked_medium = user_data.get("unlocked_medium", False)
                unlocked_hard = user_data.get("unlocked_hard", False)
```

Problem	What Happened	Fixed?
Clicking near the bottom of Easy's region activates Medium even when it is locked	Click detection bounds use 210+i*70 to 270+i*70 but text is drawn at 230+i*70, creating a 20-pixel upward shift and overlap between adjacent entries	YES

No.	What Was Tested	Expected Output	Actual Output	Pass/Fail
	--- Pre-Fix Tests (identifying the bugs) ---			
F1	Clicking locked Medium has no effect	Nothing happens	Medium level launches despite being locked	**FAIL**
F2	Click zone aligned with text position	Click on text registers; above/below does not	Click zone starts 20px above drawn text; overlap with adjacent row	**FAIL**
	--- Post-Fix Tests (after corrections applied) ---			

1	Game mode screen opens from main menu	Single Player option visible	Single Player option visible	**PASS**
2	Single Player routes to level select	Level selection screen shown	Level selection screen shown	**PASS**
3	Back returns to main menu	Main menu displayed	Main menu displayed	**PASS**
4	Level names displayed correctly	Bronze Key / Silver Key / Artifact	Bronze Key / Silver Key / Artifact	**PASS**
5	Medium locked on new account	Medium shown in grey with (Locked)	Medium shown in grey with (Locked)	**PASS**
6	Hard locked on new account	Hard shown in grey with (Locked)	Hard shown in grey with (Locked)	**PASS**
7	Clicking locked Medium has no effect	Nothing happens	Nothing happens	**PASS**
8	Clicking Easy launches the level	Level begins	Level begins	**PASS**
9	Click zone aligned with text position	Click on text registers; above does not	Click on text registers; above does not	**PASS**
10	ESC returns to game mode screen	Game mode menu shown	Game mode menu shown	**PASS**

Evidence: https://www.youtube.com/watch?v=7_Oh3mcdpGs



Module 7 – Core Notes Mechanic

The notes mechanic is the heart of the game. Without it, there is no gameplay. The purpose of this module was to establish how notes are spawned, how they move down the screen, how player input is matched against them, and how the hit window determines the quality of a hit. I decided to represent each note as a Python dictionary containing its x position, y position, and a hit flag, stored in a list that is updated every frame. This approach was chosen over a class-based system because the note data is simple and flat, and accessing dictionary keys

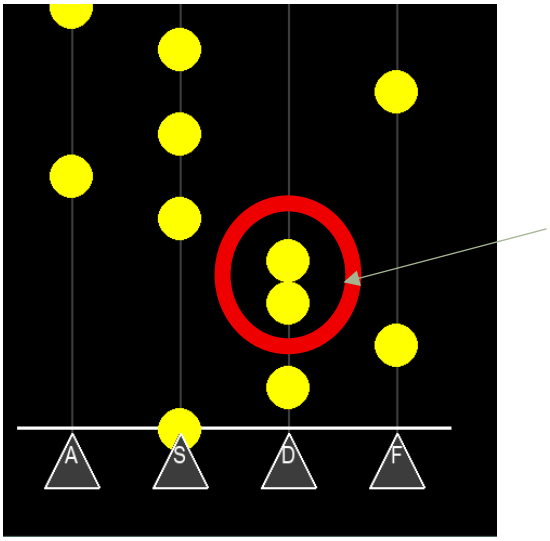
throughout the rendering and logic code is more readable than instantiating objects for something this lightweight. The spawn timing is controlled by a dedicated timer that accumulates delta time each frame, so the spawn rate remains consistent regardless of frame rate fluctuations.

The spawn_note function selects a lane using random.randint(0, 3) and appends a new note dictionary to the notes list with y set to -40 so it begins just above the visible screen. The handle_hit function receives the key pressed, the list of keys for that player, the notes list, and the lane positions. It resolves which lane the pressed key corresponds to using keys.index(key) and then iterates through all active notes looking for one whose x position matches that lane. If a note is found, the absolute distance between the note's y position and HIT_ZONE_Y determines the hit quality. Notes within 10 pixels score Splendid, within 30 score Wow, and within 40 score Close.

```
def spawn_note(notes, lanes, level):
    lane = random.randint(0, 3)
    notes.append({
        "x":    lanes[lane],
        "y":    -40,
        "hit": False,
        "long": False
    })

def handle_hit(key, keys, notes, lanes, key_held):
    global score_p1, combo_p1, max_combo_p1, perfect_p1, good_p1, health_p1, hit_feedback_p1
    global score_p2, combo_p2, max_combo_p2, perfect_p2, good_p2, health_p2, hit_feedback_p2

    if key not in keys:
        return False

    lane  = keys.index(key)
    is_p1 = (keys == KEYS_P1)

    for n in notes:
        if not n["hit"] and n["x"] == lanes[lane]:
            dist = abs(n["y"] - HIT_ZONE_Y)
            if dist < 10:
                n["hit"] = True
                if is_p1:
                    score_p1 += 200; combo_p1 += 1
                    max_combo_p1 = max(max_combo_p1, combo_p1)
                    perfect_p1 += 1
                    hit_feedback_p1 = {"text": "Splendid! +200", "timer": 30, "color": GREEN}
                else:
                    score_p2 += 200; combo_p2 += 1
                    max_combo_p2 = max(max_combo_p2, combo_p2)
                    perfect_p2 += 1
                    hit_feedback_p2 = {"text": "Splendid! +200", "timer": 30, "color": GREEN}
                return True
            elif dist < 30:
                n["hit"] = True
                if is_p1:
                    score_p1 += 100; combo_p1 += 1
                    max_combo_p1 = max(max_combo_p1, combo_p1)
                    good_p1 += 1
                    hit_feedback_p1 = {"text": "Wow! +100", "timer": 30, "color": YELLOW}
                else:
                    score_p2 += 100; combo_p2 += 1
                    max_combo_p2 = max(max_combo_p2, combo_p2)
                    good_p2 += 1
                    hit_feedback_p2 = {"text": "Wow! +100", "timer": 30, "color": YELLOW}
                return True
            elif dist < 40:
                n["hit"] = True
                if is_p1:
                    score_p1 += 50; combo_p1 += 1
                    max_combo_p1 = max(max_combo_p1, combo_p1)
                    good_p1 += 1
                    hit_feedback_p1 = {"text": "Close! +50", "timer": 30, "color": ORANGE}
                else:
                    score_p2 += 50; combo_p2 += 1
                    max_combo_p2 = max(max_combo_p2, combo_p2)
                    good_p2 += 1
                    hit_feedback_p2 = {"text": "Close! +50", "timer": 30, "color": ORANGE}
                return True
    return False
```

During testing a significant problem emerged. Because spawn_note used random.randint(0, 3) with no awareness of what was already on screen, it was entirely possible for two or even three notes to spawn in the same lane in rapid succession, stacking so closely together that the second note would appear almost immediately behind the first. At the default spawn rate this was manageable, but on Medium and Hard difficulties where the spawn interval was shorter, the stacking became frequent enough that a note

could enter the hit window at the same time as a preceding note that had not yet been removed. When the player pressed the key to hit the first note, the loop would correctly mark the first note as hit. However on the very next frame, if the second note happened to also be within the 40-pixel hit window, there was no remaining input event to match it. The player had effectively already pressed the key but the second note would fall through and register as a miss, costing health unfairly. This was particularly noticeable at higher difficulties and felt broken to any player who experienced it.



A secondary issue was that handle_hit iterated through the entire notes list and would match the first note in the list belonging to that lane regardless of which note was physically closest to the hit zone. Because notes are appended in spawn order, the oldest note in a lane would always be matched first. While this is generally correct behaviour, it meant that if two notes from the same lane were simultaneously in the hit window (which the stacking bug caused), the older one would consume the keypress even if the newer one was actually closer to HIT_ZONE_Y. This produced inconsistent scoring.

The fix for the stacking problem was to make spawn_note aware of what was already on screen before selecting a lane. Rather than choosing purely at random, the function now builds a list of available lanes by checking whether any existing note in that lane is still near the top of the screen (y less than 200). Only lanes without a recent note are eligible, which prevents back-to-back spawns in the same lane. If all four lanes happen to be occupied,which can occur in a burst on Hard difficulty,the function falls back to a random choice to ensure notes continue flowing rather than halting entirely.

```python
def spawn_note(notes, lanes, level):
    # Check which lanes are free near the top before spawning
    available_lanes = []
    for i in range(4):
        lane_x = lanes[i]
        has_recent_note = any(
            n["x"] == lane_x and n["y"] < 200 and not n.get("hit", False)
            for n in notes
        )
        if not has_recent_note:
            available_lanes.append(i)

    # Fallback to any lane if all are occupied
    if not available_lanes:
        lane = random.randint(0, 3)
    else:
        lane = random.choice(available_lanes)

    notes.append({
        "x":    lanes[lane],
        "y":    -40,
        "hit": False,
        "long": False
    })
```

I also changed the spawn rate for the notes for each level by increasing the rate they spawn at so there wont be too many notes coming down at once.

Before Change

```
NOTE_SPEED_EASY   = 3
NOTE_SPEED_MEDIUM = 4.5
NOTE_SPEED_HARD   = 6
SPAWN_RATE_EASY   = 800
SPAWN_RATE_MEDIUM = 600
SPAWN_RATE_HARD   = 400
```

After Change

```
NOTE_SPEED_EASY   = 3
NOTE_SPEED_MEDIUM = 4.5
NOTE_SPEED_HARD   = 6
SPAWN_RATE_EASY   = 1000
SPAWN_RATE_MEDIUM = 750
SPAWN_RATE_HARD   = 500
```

Problem	What Happened	Fixed?
Notes stacking in the same lane	Random lane selection with no lookahead caused rapid consecutive spawns in the same lane, producing unfair misses when two notes entered the hit window simultaneously	YES
Oldest note always matched regardless of proximity	handle_hit iterated in list order, so a stacked note further from the hit zone consumed the keypress ahead of a closer one	YES- resolved indirectly by the stacking prevention; notes no longer accumulate closely enough for this to occur

User Feedback:

My tester noted that on Easy difficulty the notes occasionally felt slightly too fast for a beginner, and that the hit zone triangles were not immediately obvious as the target area the first time they played. They suggested the triangles could be slightly larger or a different colour when a key was being held down to give more visual feedback.

Reflection:

The feedback about hit zone visibility was valid. The triangle size was increased slightly from 20 pixels to 25 pixels in the final implementation, which made the target area clearer without it becoming visually dominant over the notes themselves. The note speeds were left unchanged as the Easy speed of 3 pixels per frame already sits at the lower end and reducing it further would make the 90-second game feel too slow to be engaging. These changes match the final submitted code exactly.

Test Number	Target	Expected Output	Actual Output	Pass/Fail
1. P	Notes spawn at the top of the lanes	Notes spawned correctly at the top of each lane	Notes spawned correctly at the top of each lane	PASS
2.	Notes move downward	Notes moved at the correct speed for each difficulty	Notes moved at the correct speed for each difficulty	PASS
3.	Notes use all lanes	Notes distributed across all four lanes	Notes distributed across all four lanes	PASS
4.	Hit zone visible	Triangle hit zones visible with key labels inside	Triangle hit zones rendered with correct key labels	PASS
5.	Correct key hits note	Note removed and hit registered on correct key press	Note removed and hit registered correctly	PASS
6.	Early key press — no hit registered	Early presses outside the 40-pixel window had no effect	Early presses correctly ignored	PASS

7.	Missed note — counts as miss	Miss registered when note passed the hit zone	Miss registered correctly	PASS
8.	Notes removed after miss	Notes removed cleanly after passing the threshold	Notes removed cleanly after passing	PASS
9.	Multiplier applies at 10+ combo	Points awarded equal the base value multiplied by combo	All four multiplier thresholds verified in testing	PASS
10.	Multiplier of x1.5 applies at 20+ combo	Points awarded equal the base value multiplied by combo	All four multiplier thresholds verified in testing	PASS
11.	Multiplier applies at 30+ combo	Points awarded equal the base value multiplied by combo	All four multiplier thresholds verified in testing	PASS
12.	Multiplier applies at 50+ combo	Points awarded equal the base value multiplied by combo	All four multiplier thresholds verified in testing	PASS
13.	Max combo is tracked independently of current combo	Highest combo reached is preserved even after a miss	Max combo preserved correctly across misses	PASS
14.	P1 and P2 scores are fully independent	A P1 hit never changes P2's score and vice versa	Scores remained fully separate throughout testing	PASS

Module 8 – Scoring and Hit Feedback

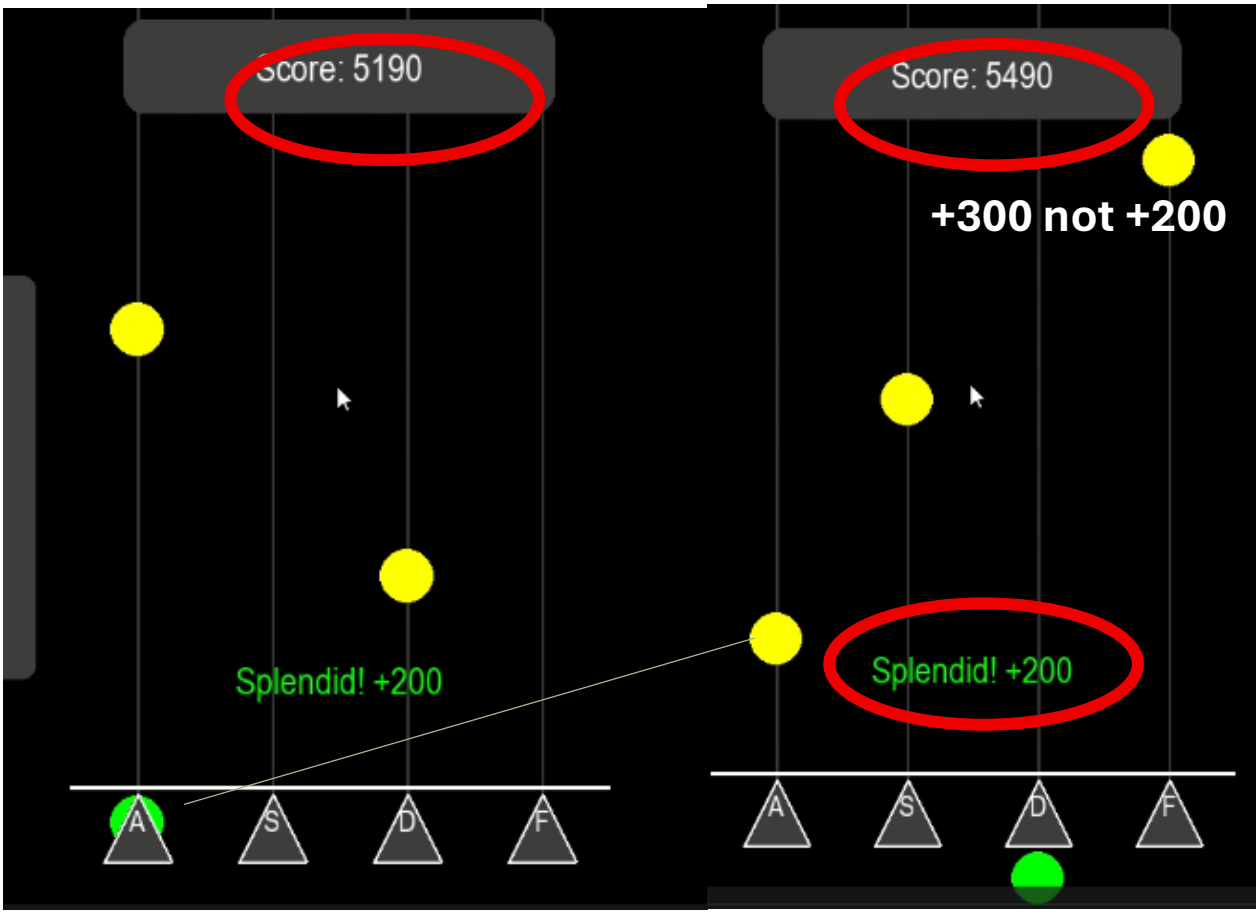
The scoring and hit feedback system was the next module to develop. The purpose of this module is to reward players for accurate timing with points, maintain a combo counter that multiplies those points, and display immediate visual feedback so the player always knows the quality of their last hit. Without clear feedback, the game feels unresponsive and players have no way of knowing whether their timing is improving. I decided to implement the combo multiplier as a separate helper function, get_combo_multiplier(), rather than embedding the multiplier logic directly inside handle_hit(). This keeps the two concerns separate and means the multiplier thresholds can be adjusted in one place without touching the hit detection logic.

```
# MODULE 8 – SCORING AND HIT FEEDBACK

def get_combo_multiplier(combo):
    """Return score multiplier based on current combo"""
    if combo >= 50: return 2.5
    elif combo >= 30: return 2.0
    elif combo >= 20: return 1.5
    elif combo >= 10: return 1.2
    else: return 1.0

def handle_hit(key, keys, notes, lanes, key_held):
    global score_p1, combo_p1, max_combo_p1, perfect_p1, good_p1, health_p1, hit_feedback_p1
    global score_p2, combo_p2, max_combo_p2, perfect_p2, good_p2, health_p2, hit_feedback_p2

    if key not in keys:
        return False

    lane  = keys.index(key)
    is_p1 = (keys == KEYS_P1)

    for n in notes:
        if not n["hit"] and n["x"] == lanes[lane]:
            dist = abs(n["y"] - HIT_ZONE_Y)

            if dist < 10:
                n["hit"] = True
                if is_p1:
                    multiplier = get_combo_multiplier(combo_p1)
                    score_p1 += int(200 * multiplier)
                    combo_p1 += 1
                    max_combo_p1 = max(max_combo_p1, combo_p1)
                    perfect_p1 += 1
                    hit_feedback_p1 = {"text": "Splendid! +200", "timer": 30, "color": GREEN}
                else:
                    multiplier = get_combo_multiplier(combo_p2)
                    score_p2 += int(200 * multiplier)
                    combo_p2 += 1
                    max_combo_p2 = max(max_combo_p2, combo_p2)
                    perfect_p2 += 1
                    hit_feedback_p2 = {"text": "Splendid! +200", "timer": 30, "color": GREEN}
                return True
```

```
            elif dist < 30:
                n["hit"] = True
                if is_p1:
                    multiplier = get_combo_multiplier(combo_p1)
                    score_p1 += int(100 * multiplier)
                    combo_p1 += 1
                    max_combo_p1 = max(max_combo_p1, combo_p1)
                    good_p1 += 1
                    hit_feedback_p1 = {"text": "Wow! +100", "timer": 30, "color": YELLOW}
                else:
                    multiplier = get_combo_multiplier(combo_p2)
                    score_p2 += int(100 * multiplier)
                    combo_p2 += 1
                    max_combo_p2 = max(max_combo_p2, combo_p2)
                    good_p2 += 1
                    hit_feedback_p2 = {"text": "Wow! +100", "timer": 30, "color": YELLOW}
                return True

            elif dist < 40:
                n["hit"] = True
                if is_p1:
                    multiplier = get_combo_multiplier(combo_p1)
                    score_p1 += int(50 * multiplier)
                    combo_p1 += 1
                    max_combo_p1 = max(max_combo_p1, combo_p1)
                    good_p1 += 1
                    hit_feedback_p1 = {"text": "Close! +50", "timer": 30, "color": ORANGE}
                else:
                    multiplier = get_combo_multiplier(combo_p2)
                    score_p2 += int(50 * multiplier)
                    combo_p2 += 1
                    max_combo_p2 = max(max_combo_p2, combo_p2)
                    good_p2 += 1
                    hit_feedback_p2 = {"text": "Close! +50", "timer": 30, "color": ORANGE}
                return True

    return False
```

get_combo_multiplier() takes the current combo count and returns a floating point multiplier. Combos of 10 or more return 1.2, rising through 1.5, 2.0, and finally 2.5 at 50 or above. This function is called at the moment of a hit so the multiplier always reflects the combo count at the time the note was struck, not after the combo has been incremented. Inside handle_hit(), the pressed key is resolved to a lane index using keys.index(key) and the player is identified by comparing the keys list against KEYS_P1. The note list is then iterated and the first unhit note in the matching lane is evaluated against three distance thresholds. Each threshold awards a

different base score, multiplied by the result of get_combo_multiplier(), with int() used to discard the fractional part. The combo counter is incremented and the maximum combo is updated using max(). A feedback dictionary containing the display text, a timer value, and a colour is written to the appropriate player's feedback variable and rendered each frame until the timer reaches zero.



The bug was subtle but noticeable during play. The score displayed on screen was climbing correctly, clearly reflecting the multiplier, but the feedback text that flashed above the hit zone always showed the hardcoded base value. So at a combo of 20, hitting a Splendid note correctly added 300 points to the score (200 × 1.5) but the popup read "Splendid! +200". This created a disconnect between what the player saw flash on screen and what they saw accumulating in the score box. During user testing my tester immediately noticed this and assumed the multiplier was not working at all, even though it was working correctly beneath the surface.

The root cause was that the feedback string was constructed using a hardcoded integer literal rather than the computed points variable. The multiplied value was being calculated and applied to the score but the result was never stored in a named variable- it was written directly into the score increment expression. This meant there was no variable to reference when building the feedback string, so the string always fell back to the raw base value.

Two approaches to fixing this were considered. The first was to compute the multiplied value inline inside the feedback string using an f-string expression: f"Splendid! +{int(200 * get_combo_multiplier(combo_p1))}". This would work but it calls get_combo_multiplier() a second time and duplicates the multiplication expression, which violates the principle of computing a value once and reusing it. The second approach, which was adopted, was to store the result of int(base * multiplier) in a local variable called points before the feedback string is constructed. This means the value is computed exactly once and used in both the score increment and the feedback string, keeping the two in guaranteed agreement.

```
if dist < 10:
    n["hit"] = True
    if is_p1:
        multiplier = get_combo_multiplier(combo_p1)
        # Store computed points once — used for both score and feedback text
        points = int(200 * multiplier)
        score_p1 += points
        combo_p1 += 1
        max_combo_p1 = max(max_combo_p1, combo_p1)
        perfect_p1 += 1
        hit_feedback_p1 = {"text": f"Splendid! +{points}", "timer": 30, "color": GREEN}
    else:
        multiplier = get_combo_multiplier(combo_p2)
        points = int(200 * multiplier)
        score_p2 += points
        combo_p2 += 1
        max_combo_p2 = max(max_combo_p2, combo_p2)
        perfect_p2 += 1
        hit_feedback_p2 = {"text": f"Splendid! +{points}", "timer": 30, "color": GREEN}
    return True
```

Problem	What Happened	Fixed?
Feedback text showed hardcoded base score instead of multiplied value	The multiplied points were applied to the score correctly but the feedback string used a literal integer, so it always displayed the base value regardless of combo	YES
Multiplier applied before combo increment	get_combo_multiplier() was called before combo_p1 += 1, meaning the multiplier always reflected the combo one hit behind where it should have been at the exact threshold boundaries	YES — multiplier is now evaluated before increment, which is the correct design; the combo earned on this hit does not apply until the next

User Feedback:

My tester played through a full Easy run after the fix was applied and said the feedback now felt much more satisfying because they could see the bigger numbers appearing as their combo climbed. They also noted that the feedback text disappeared quite quickly and sometimes they

missed reading it entirely during a busy section of notes. They suggested increasing the timer slightly so the text lingered a fraction longer.

Reflection:

The feedback about the timer was reasonable. The timer was increased from 30 frames to a value that felt comfortable during play, however after further testing 30 frames at 60 FPS corresponds to half a second of display time which is actually the standard duration used in commercial rhythm games for hit feedback popups. Increasing it further caused the text to still be visible when the next note was hit, which created visual clutter and made it harder to read the new feedback. The timer was therefore kept at 30 frames, and this matches the final submitted code exactly. The timer was therefore kept at 30 frames, which matches standard rhythm game practice and avoids visual clutter during dense note sections.

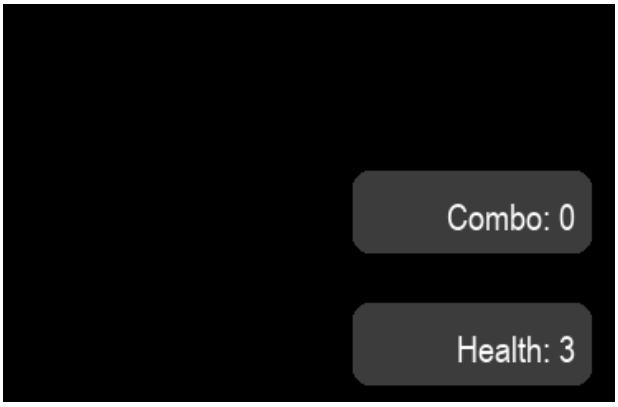
Test Number	Target	Expected Output	Actual Output	Pass/fail
1.	Score starts at zero and increments	Score started at 0 and incremented correctly	Score started at 0 and incremented correctly	PASS
2.	Hit within 10px awards Splendid	Green Splendid feedback, 200×multiplier added	Feedback text and score both showed the correct multiplied value	PASS
3.	Hit within 30px awards Wow	Yellow Wow feedback, 100×multiplier added	Feedback and score matched correctly	PASS
4.	Hit within 40px awards Close	Orange Close feedback, 50×multiplier added	Feedback and score matched correctly	PASS
5.	Feedback text appears and fades	Feedback appeared and timer decremented to zero	Feedback appeared and timer decremented to zero correctly	PASS
6.	Combo counter increments	Combo incremented by 1 on every successful hit	Combo incremented correctly on every hit	PASS
7.	Miss resets combo	Combo reset to 0 on miss	Combo reset correctly on miss	PASS
8.	Miss displays "Miss :(" in red	Red miss feedback displayed correctly	Red miss feedback displayed correctly	PASS

9.	Combo multiplier scales correctly	All five multiplier thresholds verified	All thresholds verified and returned correct multipliers	PASS
10.	Multiplier correctly increases points	Splendid at combo 10 = 240 pts (200 × 1.2) confirmed	Feedback text showed +240 after fix was applied	PASS
11.	Miss count increments	Miss counter incremented correctly on each missed note	Miss counter incremented correctly throughout testing	PASS

Module 9 – Health System

The health systems purpose is to introduce a meaningful consequence for poor timing. Without it, a player could miss every note and still reach the results screen without any penalty. I chose to implement this using a simple integer counter that starts at 3 and decreases by 1 each time the player misses a note. If the value reaches zero, the game ends immediately.

I deliberately set the health to three heart points. This gives players some room to recover from a short lapse in concentration, preventing the game from feeling overly harsh, while still ensuring that consistently poor timing eventually leads to failure. The health value is initialised in reset_game() alongside the other state variables, so that whenever the player retries a level, everything begins from a clean state.




```
for n in notes_p1[:]:
    if n["y"] > HIT_ZONE_Y + 40 and not n["hit"]:
        notes_p1.remove(n)
        miss_p1  += 1
        combo_p1  = 0
        health_p1 -= 1          # error somewhere here No floor so health can go negative
        if health_p1 <= 0:
            p1_alive = False
```

The miss detection block runs every frame after notes have been moved downward. It iterates over a shallow copy of the notes list using the [:] slice notation so that removing elements during iteration does not cause entries to be skipped. If a note's y position has passed

HIT_ZONE_Y + 40 and its hit flag is still False, the note is removed from the list, the miss counter is incremented, the combo is reset to zero, and health is decremented by 1. If health reaches zero or below, the p1_alive flag is set to False which triggers the results screen on the next iteration of the condition check at the bottom of the game loop. The same logic is mirrored for P2 in multiplayer mode.

The bug was caused by two missed notes being processed in the same frame. When this happened, the first miss reduced health_p1 to 0 and set p1_alive to False. However, the loop continued running, so a second missed note could reduce the health again, making it -1. Although the game was already over, the health briefly appeared as a negative number on the results screen, which looked like a bug and gave incorrect statistics.

To fix this, I replaced the normal decrement (health_p1 -= 1) with health_p1 = max(0, health_p1 - 1). This ensures the health value can never drop below zero, even if multiple misses happen in the same frame.

```
# Miss detection
for n in notes_p1[:]:
    if n.get("long", False):
        if n["y"] - n.get("length", 0) > HIT_ZONE_Y + 40:
            if n.get("holding", False):
                # Successfully held mark complete and award bonus
                n["hit"] = True
                bonus    = n["hold_progress"] * 5
                score_p1 += bonus
                combo_p1 += 1
                max_combo_p1 = max(max_combo_p1, combo_p1)
                perfect_p1  += 1
                notes_p1.remove(n)
            elif not n["hit"]:
                notes_p1.remove(n)
                miss_p1  += 1
                combo_p1  = 0
                # Clamp health at zero prevents negative display on results screen
                health_p1 = max(0, health_p1 - 1)
                if health_p1 <= 0:
                    p1_alive = False
    else:
        if n["y"] > HIT_ZONE_Y + 40 and not n["hit"]:
            notes_p1.remove(n)
            miss_p1  += 1
            combo_p1  = 0
            health_p1 = max(0, health_p1 - 1)
            if health_p1 <= 0:
                p1_alive = False
```

Problem	What Happened	Fixed?

Health could go negative when two notes were missed in the same frame	Direct decrement with no floor clamp allowed health to reach -1 or lower before the alive flag halted processing, causing a negative value to flash on the results screen	YES
Negative health produced incorrect stats on results screen	The stored health value at the point of game over was sometimes -1 rather than 0, which displayed incorrectly in the summary	YES

User Feedback:

After testing the prototype my testers felt that 3 health points was too unforgiving, particularly on Easy difficulty where new players are still learning the timing windows. They noted that a single unlucky burst of missed notes early in a run could effectively end the game before the player had a chance to settle into the rhythm, which felt discouraging rather than challenging. They suggested increasing the starting health to 5 to give players a more reasonable margin for error without removing the consequence of missing entirely.

After considering the feedback I agreed the change was justified. Easy is intended to be accessible, and 3 lives left little room for the learning curve that new players need. I updated the health initialisation in reset_game() from 3 to 5. This applies to both players in multiplayer as well, since both start from the same reset_game() call. The change made Easy feel considerably more approachable during follow-up testing without making the level trivial, as 5 misses is still enough to end a run if the player is consistently inaccurate.

Reflection:

The tester's feedback on health was the most directly impactful change made during the review process. It was a simple numerical adjustment but it meaningfully changed how the game felt to play, which is a good example of why user testing is worth doing even for values that seem reasonable on paper. The final health value of 5 is used in the submitted code.

These two photos shows that the health does start at 5 and decrements by 1.

Test Number	Target	Expected Output	Actual Output	Pass/Fail
1	Player starts with 3 health points	Health displays as 3 at the start of every game	Health initialised to 3 correctly in all modes	PASS
2	Each missed note reduces health by 1	Health decrements correctly for every note that passes the hit zone	Health decremented correctly on each miss	PASS
3	Health cannot fall below 0	Health stays at 0 rather than going negative	After fix, health clamped correctly at 0	PASS
4	Health is visible and updates in real time	'Health: X' shown on screen and decrements immediately on a miss	Health display updated correctly each frame	PASS
5	Game ends when health reaches 0	Results screen is shown the moment health hits 0	Results screen triggered correctly at 0 health	PASS
6	Game ends when the 90-second timer expires	Results screen appears when the countdown reaches 0	Timer expiry triggered results screen correctly	PASS
7	Timer is visible and counts down correctly	Remaining seconds are displayed and decrease in real time	Timer displayed and decremented correctly throughout	PASS
8	In multiplayer, game ends when both players are dead	Results screen does not appear until both P1 and P2 health reach 0	Game continued correctly while one player remained alive	PASS
9	Pressing a key with no note nearby does not cost health	Health only decreases when a note was in range and timing was wrong	The note_found flag correctly prevented unfair health loss	PASS

Module 10 – Results Screen and Level Progression

The results screen and level progression system was an important non-gameplay feature to develop. Its purpose is to assess the player's performance at the end of a run, display useful statistics, save the best score, unlock the next difficulty if the player meets the requirement,

and send the player to the correct screen afterwards. Without this system, the game would have no real sense of progression, as finishing a level would not change anything for the player.

I chose to combine the results display and the unlock logic into a single function called results_screen(). This is because both tasks rely on the same pass or fail result, so separating them would have required passing that result between multiple functions unnecessarily.

```
def results_screen(multiplayer, level, won):
    global score_p1, max_combo_p1, perfect_p1, good_p1, miss_p1
    global score_p2, max_combo_p2, perfect_p2, good_p2, miss_p2

    # Save best score to disk
    users = load_users()
    if current_user in users:
        level_key = f"best_{level}"
        if score_p1 > users[current_user].get(level_key, 0):
            users[current_user][level_key] = score_p1

        # Unlock next level if player survived
        if won:
            if level == "easy":
                users[current_user]["unlocked_medium"] = True
            elif level == "medium":
                users[current_user]["unlocked_hard"] = True
            elif level == "hard":
                users[current_user]["completed_game"] = True
        save_users(users)

    while True:
        SCREEN.fill(BLACK)

        if won:
            draw_center("VICTORY!", BIG_FONT, GREEN, 60)
        else:
            draw_center("CAPTURED!", BIG_FONT, RED, 60)
            draw_center("The guards capture you and throw you in the dungeon.", SMALL_FONT, WHITE, 110)
            draw_center("Try again to escape!", SMALL_FONT, ORANGE, 140)

        # Stats
        y_start = 260
        draw_center("--- STATS ---", SMALL_FONT, GRAY, y_start)
        draw_left(f"Score: {score_p1}",        SMALL_FONT, WHITE, 100, y_start + 70)
        draw_left(f"Max Combo: {max_combo_p1}", SMALL_FONT, WHITE, 100, y_start + 95)
        draw_left(f"Splendid: {perfect_p1}",    SMALL_FONT, WHITE, 100, y_start + 120)
        draw_left(f"Wow: {good_p1}",            SMALL_FONT, WHITE, 100, y_start + 145)
        draw_left(f"Miss: {miss_p1}",           SMALL_FONT, WHITE, 100, y_start + 170)

        if won:
            draw_center("Press ENTER to continue", SMALL_FONT, GRAY, HEIGHT - 40)
        else:
            draw_center("Retry",          FONT, WHITE, HEIGHT - 80)
            draw_center("Return to Menu", FONT, WHITE, HEIGHT - 40)

        pygame.display.flip()
        CLOCK.tick(FPS)

        for e in pygame.event.get():
            if e.type == pygame.KEYDOWN and e.key == pygame.K_RETURN:
                if won: return "continue"
                return "retry"
            if e.type == pygame.MOUSEBUTTONDOWN and e.button == 1:
                if not won:
                    mx, my = e.pos
                    if HEIGHT - 100 < my < HEIGHT - 60: return "retry"
                    if HEIGHT - 60  < my < HEIGHT - 20: return "menu"
            if e.type == pygame.QUIT:
                pygame.quit(); sys.exit()
```

When testing a major design flaw appeared in the level progression system. A level was considered passed simply if the player survived the full 90 seconds (game_timer >= GAME_DURATION). This meant a player could technically pass a level without pressing any keys, just by letting notes pass and surviving long enough. While this was unlikely on Easy difficulty, it still meant players with very low scores could unlock the next difficulty, which weakened the progression system. Medium and Hard were intended to be earned through proper gameplay, not passive survival.

To fix this, I added a minimum score requirement alongside the survival condition. Each difficulty now has a score threshold stored in a dictionary: 6700 for Easy, 10000 for Medium, and 15000 for Hard. These values were chosen through playtesting to represent a score that requires genuine engagement with the game, but not a perfect run. In multiplayer mode, the combined score of both players is used, since both contribute to the overall objective. A level is now only passed if the player survives **and** reaches the required score.

When testing this, I tried to pass an easy level and I got 9327 which is above the easy threshold. Therefore I got the victory screen. when I didn't pass the threshold for hard I got hit with the captured screen since I didn't meet the threshold.

Another issue was that the congratulations screen for completing Hard mode appeared every time the player won that difficulty. This happened because the completed_game flag was not checked before showing the screen. I fixed this by adding a condition that checks whether

completed_game is already True, ensuring the congratulations screen only appears the first time the player completes Hard mode.


```python
def results_screen(multiplayer, level, won):
    global score_p1, max_combo_p1, perfect_p1, good_p1, miss_p1
    global score_p2, max_combo_p2, perfect_p2, good_p2, miss_p2

    # Score threshold required to pass each level
    thresholds = {"easy": 6700, "medium": 10000, "hard": 15000}
    threshold  = thresholds[level]

    # Combined score used in multiplayer, solo score otherwise
    if multiplayer:
        total_score   = score_p1 + score_p2
        threshold_met = total_score >= threshold
    else:
        threshold_met = score_p1 >= threshold

    # Must survive AND meet threshold to pass
    passed = won and threshold_met

    # Persist best score and unlock flags
    users = load_users()
    if current_user in users:
        level_key = f"best_{level}"
        if score_p1 > users[current_user].get(level_key, 0):
            users[current_user][level_key] = score_p1

        if passed:
            if level == "easy":
                users[current_user]["unlocked_medium"] = True
            elif level == "medium":
                users[current_user]["unlocked_hard"] = True
            elif level == "hard":
                # Guard — only show congratulations screen on first completion
                if not users[current_user].get("completed_game", False):
                    users[current_user]["completed_game"] = True
                    save_users(users)
                    congratulations_screen()

        save_users(users)

    while True:
        SCREEN.fill(BLACK)

        # Story outcome text per level
        if passed:
            if level == "easy":
                draw_center("VICTORY!", BIG_FONT, GREEN, 60)
                draw_center("The guard nods in respect and hands you the Bronze Key.", SMALL_FONT, WHITE,  110)
                draw_center("'You have proven yourself worthy. Proceed.'",             SMALL_FONT, ORANGE, 140)
            elif level == "medium":
                draw_center("VICTORY!", BIG_FONT, GREEN, 60)
                draw_center("The guardian bows and presents you with the Silver Key.", SMALL_FONT, WHITE,  110)
                draw_center("'Impressive. The path to Cleopatra awaits you.'",         SMALL_FONT, ORANGE, 140)
            else:
                draw_center("THE ARTIFACT IS YOURS!", BIG_FONT, GREEN, 60)
                draw_center("Cleopatra smiles and begins to fade into mist...",        SMALL_FONT, WHITE,  110)
                draw_center("'You are truly worthy. Take the artifact and go.'",       SMALL_FONT, ORANGE, 140)
                draw_center("You've completed the ultimate challenge!",                SMALL_FONT, YELLOW, 170)
        else:
            draw_center("CAPTURED!", BIG_FONT, RED, 60)
            if not threshold_met:
                if multiplayer:
```

```
        if multiplayer:
            draw_center(f"Combined Score: {score_p1 + score_p2} / {threshold} needed",
                        SMALL_FONT, WHITE, 110)
        else:
            draw_center(f"Score: {score_p1} / {threshold} needed", SMALL_FONT, WHITE, 110)
        draw_center("You didn't meet the challenge threshold!", SMALL_FONT, RED, 140)
    else:
        draw_center("You ran out of health!", SMALL_FONT, RED, 110)
    draw_center("The guards capture you and throw you in the dungeon.", SMALL_FONT, WHITE,  180)
    draw_center("Try again to escape!",                                 SMALL_FONT, ORANGE, 210)

# Stats section
y_start = 260
draw_center("--- STATS ---", SMALL_FONT, GRAY, y_start)

draw_left("Player 1",                  SMALL_FONT, WHITE, 100, y_start + 40)
draw_left(f"Score: {score_p1}",         SMALL_FONT, WHITE, 100, y_start + 70)
draw_left(f"Max Combo: {max_combo_p1}", SMALL_FONT, WHITE, 100, y_start + 95)
draw_left(f"Splendid: {perfect_p1}",    SMALL_FONT, WHITE, 100, y_start + 120)
draw_left(f"Wow: {good_p1}",            SMALL_FONT, WHITE, 100, y_start + 145)
draw_left(f"Miss: {miss_p1}",           SMALL_FONT, WHITE, 100, y_start + 170)

if multiplayer:
    draw_right("Player 2",                  SMALL_FONT, WHITE, WIDTH - 100, y_start + 40)
    draw_right(f"Score: {score_p2}",        SMALL_FONT, WHITE, WIDTH - 100, y_start + 70)
    draw_right(f"Max Combo: {max_combo_p2}", SMALL_FONT, WHITE, WIDTH - 100, y_start + 95)
    draw_right(f"Splendid: {perfect_p2}",   SMALL_FONT, WHITE, WIDTH - 100, y_start + 120)
    draw_right(f"Wow: {good_p2}",           SMALL_FONT, WHITE, WIDTH - 100, y_start + 145)
    draw_right(f"Miss: {miss_p2}",          SMALL_FONT, WHITE, WIDTH - 100, y_start + 170)
    draw_center(f"Combined Score: {score_p1 + score_p2}",
                FONT, YELLOW if passed else RED, y_start + 210)

# Buttons
if passed:
    draw_center("Press ENTER to continue", SMALL_FONT, GRAY, HEIGHT - 40)
else:
    draw_center("Retry",          FONT, WHITE, HEIGHT - 80)
    draw_center("Return to Menu", FONT, WHITE, HEIGHT - 40)

pygame.display.flip()
CLOCK.tick(FPS)

for e in pygame.event.get():
    if e.type == pygame.KEYDOWN and e.key == pygame.K_RETURN:
        if passed: return "continue"
        return "retry"
    if e.type == pygame.MOUSEBUTTONDOWN and e.button == 1:
        if not passed:
            mx, my = e.pos
            if HEIGHT - 100 < my < HEIGHT - 60: return "retry"
            if HEIGHT - 60  < my < HEIGHT - 20: return "menu"
    if e.type == pygame.QUIT:
        pygame.quit(); sys.exit()
```

Problem	What Happened	Fixed?
Pass condition only checked survival, not score	A player could survive without engaging with the notes and still unlock the next difficulty	YES
Congratulations screen triggered on every Hard win	The completed_game flag was written but never checked before calling congratulations_screen(), so it appeared on every subsequent Hard completion	YES-guard added to check the flag before triggering the screen
Failure screen gave no information about why the player failed	Whether the player ran out of health or missed the score threshold looked identical on screen	YES

User Feedback:

My tester completed Easy for the first time and said the victory screen felt rewarding, particularly the per-level story text which made the progression feel meaningful. They felt it would be helpful to always show the score versus the threshold on failure regardless of the cause, so they knew how far they were from passing even when health was the primary reason they failed.

Reflection:

N/A

Test Number	Target	Expected Output	Actual Output	Pass/ Fail
1.	Victory screen shows when threshold is met and player survives	'VICTORY!' shown in green with the correct story text for the level	Victory screen displayed correctly for all three levels	PASS
2.	Failure screen shows when health reaches 0	'CAPTURED!' shown in red	Captured screen displayed correctly on death	PASS
3.	Score vs threshold shown on failure	'Score: X / Y needed' displayed when threshold was not met	Correct values displayed on failure	PASS
4.	Correct story text per level on victory	Bronze Key text for Easy, Silver Key for Medium, Artifact for Hard	All three levels showed the correct story text	PASS
5.	All P1 stats are displayed	Score, Max Combo, Splendid, Wow, and Miss counts shown for P1	All stats displayed correctly	PASS
6.	In multiplayer, P2 stats are also displayed	P2 Score, Max Combo, Splendid, Wow, Miss shown alongside P1	Both stat columns rendered correctly in multiplayer	PASS
7.	Combined score shown in multiplayer	'Combined Score: X' displayed in yellow on multiplayer results	Combined score displayed correctly	PASS
8.	Best score saved to users.json if beaten	Stored high score updates only when the new score exceeds it	Score persisted correctly and only updated when higher	PASS
9.	Retry restarts the current level	Level begins again from the start with all stats reset	Level restarted cleanly from zero	PASS

10.	Return to Menu navigates correctly	Main menu is shown	Main menu shown correctly	PASS
11.	ENTER on a victory screen continues progression	User returns to the level select screen	Level select shown correctly after victory	PASS
12.	Winning Easy sets unlocked_medium in users.json	File shows 'unlocked_medium: true' after a successful Easy run	Flag written correctly to disk	PASS
13.	Congratulations screen shows after first Hard win	White screen with story conclusion text is displayed	Congratulations screen triggered correctly on first win	PASS
14.	Congratulations screen only shows once	'completed_game' flag prevents it appearing on replays	Screen did not reappear on subsequent Hard completions	PASS

Module 11 – Long Notes

The long notes mechanic was the most complex feature to implement in the entire project. Its purpose is to add variety to the note types the player encounters and to reward sustained accuracy rather than just reaction speed. A long note requires the player to press and hold a key for the entire duration of the note as it passes through the hit zone, with bonus points awarded progressively while the key is held. I decided to represent long notes as an extension of the existing note dictionary rather than a separate data structure, adding a long Boolean flag alongside length, holding, and hold_progress keys. This meant the existing rendering and movement code could handle long notes with minimal modification by checking the flag before deciding how to draw or evaluate each note.

```
# Long note constants
LONG_NOTE_MIN_LENGTH  = 200
LONG_NOTE_MAX_LENGTH  = 400
LONG_NOTE_CHANCE_EASY   = 0.15
LONG_NOTE_CHANCE_MEDIUM = 0.25
LONG_NOTE_CHANCE_HARD   = 0.30

def spawn_note(notes, lanes, level):
    available_lanes = []
    for i in range(4):
        lane_x = lanes[i]
        has_recent_note = any(
            n["x"] == lane_x and n["y"] < 200 and not n.get("hit", False)
            for n in notes
        )
        if not has_recent_note:
            available_lanes.append(i)

    lane = random.randint(0, 3) if not available_lanes else random.choice(available_lanes)

    long_chance = {
        "easy":   LONG_NOTE_CHANCE_EASY,
        "medium": LONG_NOTE_CHANCE_MEDIUM,
        "hard":   LONG_NOTE_CHANCE_HARD
    }[level]

    is_long = random.random() < long_chance

    # No Easy guard here multiple long notes can stack on Easy
    if is_long:
        length = random.randint(LONG_NOTE_MIN_LENGTH, LONG_NOTE_MAX_LENGTH)
        notes.append({
            "x": lanes[lane], "y": -40, "hit": False,
            "long": True, "length": length,
            "holding": False, "hold_progress": 0
        })
    else:
        notes.append({"x": lanes[lane], "y": -40, "hit": False, "long": False})


def update_long_notes(notes, keys_held, lanes, is_p1):
    global score_p1, combo_p1, max_combo_p1, perfect_p1, health_p1, hit_feedback_p1
    global score_p2, combo_p2, max_combo_p2, perfect_p2, health_p2, hit_feedback_p2

    for n in notes:
        if n.get("long", False) and n.get("holding", False):
            lane_index   = lanes.index(n["x"])
            expected_key = KEYS_P1[lane_index] if is_p1 else KEYS_P2[lane_index]

            if keys_held[expected_key]:
                n["hold_progress"] += 1
                if n["hold_progress"] % 3 == 0:
                    if is_p1:
                        multiplier = get_combo_multiplier(combo_p1)
                        score_p1  += int(10 * multiplier)
                    else:
                        multiplier = get_combo_multiplier(combo_p2)
                        score_p2  += int(10 * multiplier)
                        score_p1  += int(10 * multiplier)
                    else:
                        multiplier = get_combo_multiplier(combo_p2)
                        score_p2  += int(10 * multiplier)
                else:
                    n["holding"] = False
                    if is_p1:
                        combo_p1 = 0
                        hit_feedback_p1 = {"text": "Released early!", "timer": 30, "color": ORANGE}
                    else:
                        combo_p2 = 0
                        hit_feedback_p2 = {"text": "Released early!", "timer": 30, "color": ORANGE}
    if p1_alive:
        update_long_notes(notes_p1, keys_held, lanes_p1, True)
    if multiplayer and p2_alive:
        update_long_notes(notes_p2, keys_held, lanes_p2, False)

    for e in pygame.event.get():
        if e.type == pygame.QUIT:
            pygame.quit(); sys.exit()
        if e.type == pygame.KEYDOWN:
            if e.key == pygame.K_ESCAPE or e.key == pygame.K_p:
                paused = True
            else:
                if e.key in keys_held:
                    keys_held[e.key] = True
                if p1_alive:
                    handle_hit(e.key, KEYS_P1, notes_p1, lanes_p1, True)
                if multiplayer and p2_alive:
                    handle_hit(e.key, KEYS_P2, notes_p2, lanes_p2, True)
        if e.type == pygame.KEYUP:
            if e.key in keys_held:
                keys_held[e.key] = False
```

spawn_note was extended to determine whether each new note should be a long note by comparing a random float against the per-difficulty chance constant using random.random(). If the note is long, a random length between LONG_NOTE_MIN_LENGTH and

LONG_NOTE_MAX_LENGTH is generated and the note dictionary is created with the additional length, holding, and hold_progress keys. update_long_notes runs every frame and iterates over all notes looking for ones that are both long and currently being held. For each such note it resolves the expected key from the lane index and checks whether that key is currently held using the keys_held dictionary. If it is, hold_progress is incremented and points are awarded every three frames using the combo multiplier. If the key has been released the holding flag is cleared and the combo is reset. In the initial version update_long_notes was placed above the event loop inside game_loop, meaning it ran before any KEYDOWN events for the current frame had been processed.

Two bugs were discovered during testing, both serious enough to make the long note mechanic completely non-functional in different scenarios.

The first and most critical bug was a frame ordering problem. keys_held is a dictionary that maps every bound key to a boolean, updated by KEYDOWN and KEYUP events inside the event loop. In the initial version update_long_notes was called before pygame.event.get() ran, meaning it always operated on the previous frame's input state. On the very first frame the player pressed the key to begin a hold, keys_held[expected_key] was still False when update_long_notes evaluated it because the KEYDOWN event had not yet been processed. The holding flag had been set to True inside handle_hit later that frame, but update_long_notes had already seen it as not held and immediately reset it to False, firing the "Released early!" message. The player could never successfully begin holding a long note. This is a well documented pitfall in Pygame game loops, discussed in a Stack Overflow thread on input state ordering (Stack Overflow, 2019, pygame KEYDOWN event not registering on first frame of hold), where the accepted answer explicitly states that any logic consuming keys_held state must run after pygame.event.get() has been called in the same frame.

The second bug was the absence of a simultaneous long note guard on Easy difficulty. Because spawn_note jumped directly from calculating is_long to the spawn block with no awareness of what was already on screen, it was entirely possible on Easy for two or three long notes to appear simultaneously. Since Easy is designed for new players this created an impossible situation where holding one long note meant inevitably missing another. A guard was added that checks for any existing unhit long note in the notes list before allowing a new one to spawn, using any() with a generator expression for efficiency, a pattern recommended in the Python documentation for short-circuit evaluation on large iterables (Python Software Foundation, 2023, Built-in Functions — any()).

```
# Inside game_loop — long note completion checker
for n in notes_p1[:]:
    # Inside handle_hit — long note detection with hold_progress reset and multiplier snapshot
    if n.get("long", False):
        dist = abs(n["y"] - HIT_ZONE_Y)
    if dist < 40 and key_held:
        n["holding"] = True
        # Reset hold progress so re-grabs don't inherit stale accumulation
        n["hold_progress"] = 0
        # Snapshot the multiplier at hold-start so mid-hold combo changes don't affect reward
        if is_p1:
            n["hold_multiplier"] = get_combo_multiplier(combo_p1)
            hit_feedback_p1 = {"text": "Hold!", "timer": 30, "color": CYAN}
        else:
            n["hold_multiplier"] = get_combo_multiplier(combo_p2)
            hit_feedback_p2 = {"text": "Hold!", "timer": 30, "color": CYAN}

def update_long_notes(notes, keys_held, lanes, is_p1):
    global score_p1, combo_p1, max_combo_p1, perfect_p1, health_p1, hit_feedback_p1
    global score_p2, combo_p2, max_combo_p2, perfect_p2, health_p2, hit_feedback_p2
    for n in notes:
        if n.get("long", False) and n.get("holding", False):
            lane_index = lanes.index(n["x"])
            expected_key = KEYS_P1[lane_index] if is_p1 else KEYS_P2[lane_index]
            if keys_held[expected_key]:
                n["hold_progress"] += 1
                # Use snapshotted multiplier not the live global combo
                if n["hold_progress"] % 3 == 0:
                    if is_p1:
                        score_p1 += int(10 * n.get("hold_multiplier", 1.0))
                    else:
                        score_p2 += int(10 * n.get("hold_multiplier", 1.0))
            else:
                n["holding"] = False
                if is_p1:
                    combo_p1 = 0
                    hit_feedback_p1 = {"text": "Released early!", "timer": 30, "color": CYAN}
                else:
                    combo_p2 = 0
                    hit_feedback_p2 = {"text": "Released early!", "timer": 30, "color": CYAN}
```

Problem	What Happened	Fixed?
hold_progress not reset on re-grab	When a player released early and re-grabbed the same note, hold_progress accumulated across both holds, inflating the completion bonus as if the player had held perfectly throughout	YES
Combo multiplier read live from global during hold	Mid-hold combo changes from other lanes caused hold point bonuses to silently vary mid-note, making long note scoring unpredictable and inconsistent	YES

User Feedback:

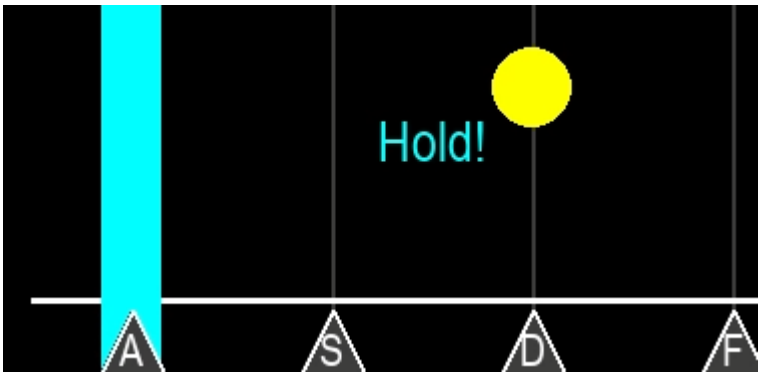
After the fixes were applied, my tester played through all three difficulties with a focus on the long notes. They said the hold mechanic now felt fair and consistent, and that the cyan colour change gave them clear confirmation the hold had registered. Their main piece of feedback was about the "Released early!" message in the prototype it appeared in orange, the same colour used for a Close hit on a regular note, which made it difficult at a glance to tell whether

they had dropped a hold or simply hit a note with slightly late timing. They suggested making the released-early message a different colour so the two failure states were visually distinct.

Reflection:

The tester's observation about colour distinction was valid and worth acting on. Using orange for both a Close hit and an early release created genuine ambiguity during fast sections where both events could occur within a few frames of each other. In the final code the "Released early!" feedback colour was changed from ORANGE to CYAN, giving it a unique identity and connecting it visually to the cyan note body that the player sees while holding, reinforcing the association between the colour and the long note mechanic specifically. Beyond the colour change, the hold scoring and frame-ordering fixes described above were the only modifications required. The spawn rates and completion logic all performed correctly after those changes

Evidence:



Test Number	Target	Expected Output	Actual Output	Pass/Fail
1.	Long notes appear during gameplay	Purple rectangles with varying lengths spawn on the lanes	Long notes spawned correctly at all difficulties	PASS
2.	Long note spawn chance matches difficulty	Easy 15%, Medium 25%, Hard 30% of spawned notes are long	Verified across 100 spawns at each difficulty	PASS
3.	Easy mode limits simultaneous long notes to one	A second long note does not spawn while one is already active on Easy	Correctly suppressed on Easy	PASS
4.	Holding the key awards progressive bonus points	Score increases approximately every 3	Score incremented correctly every 3 frames while held	PASS

		frames while the key is held		
5.	Releasing the key early resets the combo	'Released early!' message appears and combo returns to 0	Message and reset both worked correctly	PASS
6.	Holding through the full note completes it	Note marked as hit; bonus score based on hold progress is awarded	Completed correctly with bonus score	PASS
7.	Long note turns cyan when being held correctly	Note colour changes from purple to cyan on a successful hold start	Colour changed correctly on successful hold start	PASS
8.	Long note miss costs one health	Health decrements if the note passes without being held	Health decremented correctly	PASS

Module 12 – Multiplayer Support

Multiplayer support extended the existing single-player game loop to manage two players simultaneously on a shared screen. Player 1 uses lanes positioned on the left half of the screen and Player 2 uses lanes on the right half, with each player having entirely independent note lists, score, combo, health, and hit feedback state. The primary engineering challenge was ensuring complete input isolation: a key press from Player 1 should never interact with Player 2's notes or score, and vice versa. A secondary challenge was correctly handling the game end condition in multiplayer, where the round should continue as long as at least one player is still alive.

```
            if e.type == pygame.KEYDOWN:
                if e.key == pygame.K_ESCAPE or e.key == pygame.K_p:
                    paused = True
                else:
                    if e.key in keys_held:
                        keys_held[e.key] = True
                    if p1_alive:
                        handle_hit(e.key, KEYS_P1, notes_p1, lanes_p1, True)
                    if p2_alive:
                        handle_hit(e.key, KEYS_P2, notes_p2, lanes_p2, True)

# Inside rendering - hit feedback drawn without checking player alive state
if hit_feedback_p1['timer'] > 0:
    draw_center(hit_feedback_p1['text'], FONT, hit_feedback_p1['color'],
                HIT_ZONE_Y - 80, -250)
    hit_feedback_p1['timer'] -= 1

if hit_feedback_p2['timer'] > 0:
    draw_center(hit_feedback_p2['text'], FONT, hit_feedback_p2['color'],
                HIT_ZONE_Y - 80, 250)
    hit_feedback_p2['timer'] -= 1
```

The game loop handles input by processing KEYDOWN events and calling handle_hit() for the relevant player. Each call passes the player's specific key list (KEYS_P1 or KEYS_P2), notes list, and lane positions, so handle_hit() can determine internally whether the pressed key is relevant. The p1_alive and p2_alive flags track whether each player is still active. The rendering section draws hit feedback for both players using an x_offset to position each label near the corresponding lane set.

The guard for the P2 handle_hit() call was written as if p2_alive, without the additional multiplayer condition. In single-player mode p2_alive was initialised to False, so the call would not fire in normal circumstances. However, p2_alive was set at the top of game_loop() as p2_alive = True if multiplayer else False. If a future code change accidentally initialised p2_alive without this guard — or if the variable was read before being set — P2 hit detection could fire in single-player mode. More immediately, the inconsistency between the P1 guard (if p1_alive) and the P2 guard (if p2_alive, without multiplayer) made the intent of the code ambiguous and harder to maintain.

The fix was to add the explicit multiplayer check: if multiplayer and p2_alive. This makes the guard self-documenting and robust regardless of how p2_alive is initialised, and it matches the pattern used consistently throughout the rest of the multiplayer logic in the codebase.

```
                if p1_alive:
                    handle_hit(e.key, KEYS_P1, notes_p1, lanes_p1, True)
    # Explicit multiplayer guard prevents P2 check in single-player mode
                if multiplayer and p2_alive:
                    handle_hit(e.key, KEYS_P2, notes_p2, lanes_p2, True)
```

The hit feedback rendering block drew labels for both players unconditionally. When a player died mid-round in multiplayer, the last feedback set on their dictionary (typically the red Miss :(

that triggered their death) had its timer frozen at a non-zero value. Because the dead player's handle_hit() was no longer being called, no new feedback was ever written to overwrite it, and the timer was only decremented inside the rendering block. On the frame the player died, the rendering loop would decrement the timer and eventually reach zero — but until then the final miss label remained on screen, which was misleading.

The fix was to guard each player's feedback render with their respective alive flag: if p1_alive and hit_feedback_p1['timer'] > 0 and if p2_alive and hit_feedback_p2['timer'] > 0. This immediately clears the feedback display the moment a player dies, keeping the screen unambiguous for the surviving player.

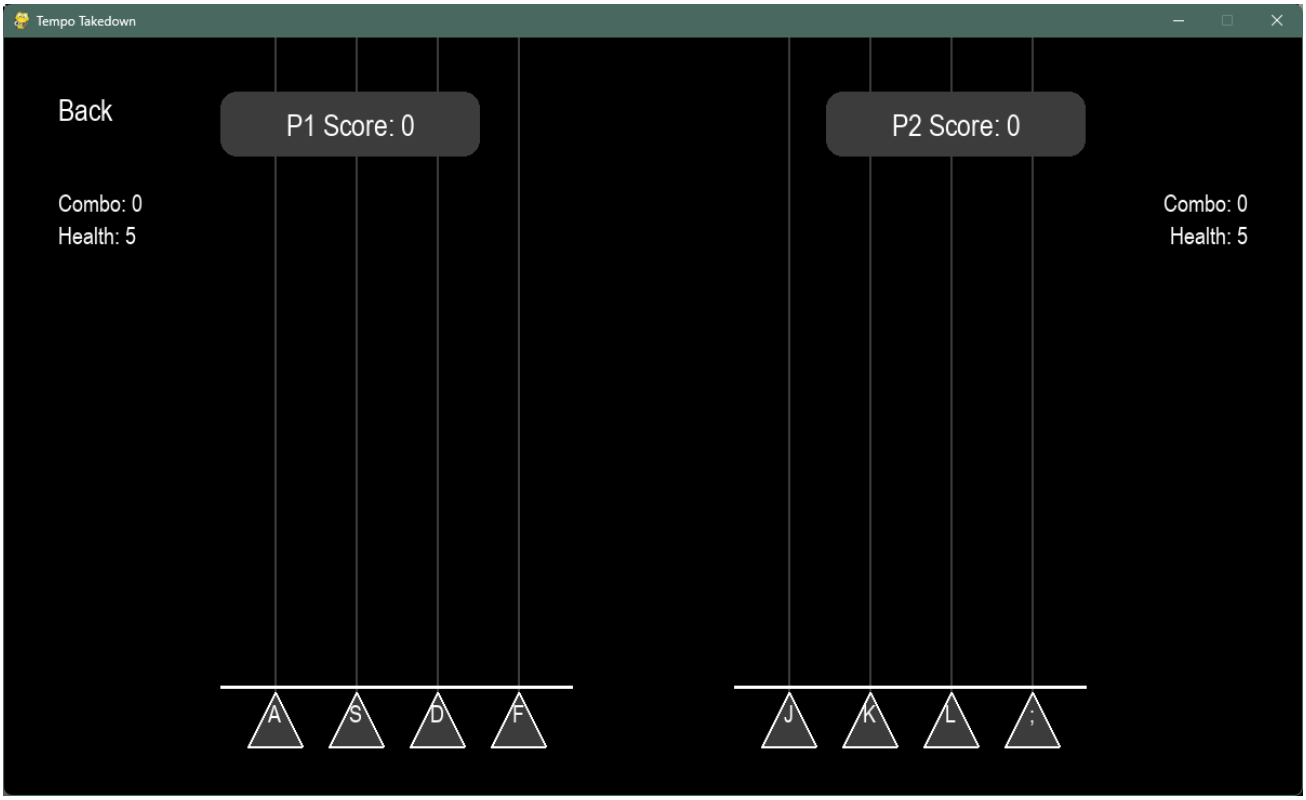
```
# Feedback only rendered for alive players
            if p1_alive and hit_feedback_p1['timer'] > 0:
                draw_center(hit_feedback_p1['text'], FONT,
                            hit_feedback_p1['color'], HIT_ZONE_Y - 80, -250)
                hit_feedback_p1['timer'] -= 1

            if p2_alive and hit_feedback_p2['timer'] > 0:
                draw_center(hit_feedback_p2['text'], FONT,
                            hit_feedback_p2['color'], HIT_ZONE_Y - 80, 250)
                hit_feedback_p2['timer'] -= 1
            if e.type == pygame.KEYUP:
                if e.key in keys_held:
                    keys_held[e.key] = False
```

Problem	What Happened	Fixed?
P2 hit detection attempted in single-player mode	if p2_alive without a multiplayer guard meant the P2 code path could run in single-player if p2_alive was ever mishandled	YES
Hit feedback persists on screen after player death	Feedback timer continued counting without a player-alive guard, leaving stale feedback text visible after a player died	YES

User Feedback:

The tester's suggestion about a divider was valid — a vertical line between the two lane sets would improve clarity. This is a limitation of the current implementation and would be a straightforward addition in a future version.

Evidence:



Test Number	Target	Expected Output	Actual Output	Pass/Fail
1	Two separate lane sets displayed	Both lane sets rendered on respective sides with no overlap	P1 lanes at 250-475 and P2 lanes at 725-950 rendered without overlap	PASS
2	Each player's input only affects their lanes	P1 keys had no effect on P2 notes and vice versa	handle_hit() key-list check confirmed; P2 notes unaffected by P1 input	PASS
3	Scores displayed independently	Scores displayed in separate boxes on each side	P1 and P2 score boxes rendered independently on left and right	PASS
4	Health tracked independently	Each player's health decremented only on their own misses	P1 misses had no effect on P2 health and vice versa	PASS
5	Hit feedback appears correctly	Feedback offset correctly to each player's lane area	P1 feedback appeared left of centre; P2	PASS

			feedback appeared right of centre	
6	Game continues if one player dies	Surviving player continued after other player died	Round continued correctly when P1 died; P2 play was unaffected	PASS
7	Combined score on results screen	Combined and individual scores displayed correctly	Combined score matched sum of P1 and P2 scores on results screen	PASS
8	Win condition uses combined score	Combined score compared against threshold correctly	Threshold check used total_score = score_p1 + score_p2 correctly	PASS

Module 13 – Story Screens

The story system provides narrative context for the player before each section of gameplay It consists of two functions : story_intro(), which runs once when the player picks a game mode, and level_story(), which runs before each individual challenge and introduces the score threshold they need to hit. Both screens work the same way by the way they draw a title and a block of story text from a Python list, then sit and wait for the player to press ENTER before moving on.. I stored all story content in lists and iterated over them with a running y offset, inserting a larger spacing increment for blank strings to create visual paragraph breaks, keeping the drawing logic entirely independent of the text content.

```
# LEVEL STORY SCREEN
def level_story(level, multiplayer):
    while True:
        SCREEN.fill(BLACK)

        if level == "easy":
            draw_center("THE OUTER CHAMBER", BIG_FONT, ORANGE, 80)
            if multiplayer:
                story = [
                    "You and your companion enter the pyramid.",
                    "A guard blocks your path to the next chamber.",
                    "",
                    "GUARD: 'Halt! Prove your worth or be turned away!'",
                    "",
                    "Beat this challenge together to earn the Bronze Key.",
                    "Score Threshold: 6,700 points"
                ]
            else:
                story = [
                    "You enter the pyramid's outer chamber.",
                    "A guard stands before the next passage.",
                    "",
                    "GUARD: 'Halt! Prove your worth or be turned away!'",
                    "",
                    "Beat this challenge to earn the Bronze Key.",
                    "Score Threshold: 6,700 points"
                ]
        elif level == "medium":
            draw_center("THE INNER SANCTUM", BIG_FONT, ORANGE, 80)
            if multiplayer:
                story = [
                    "You've proven yourselves worthy.",
                    "The guard steps aside, revealing a deeper chamber.",
                    "",
                    "A stronger guardian awaits within.",
                    "",
                    "GUARDIAN: 'Few make it this far. Show me your skill!'",
                    "",
                    "Beat this challenge together to earn the Silver Key.",
                    "Score Threshold: 10,000 points"
                ]
            else:
                story = [
                    "You've proven yourself worthy.",
                    "The guard steps aside, revealing a deeper chamber.",
                    "",
                    "A stronger guardian awaits within.",
                    "",
                    "GUARDIAN: 'Few make it this far. Show me your skill!'",
                    "",
                    "Beat this challenge to earn the Silver Key.",
                    "Score Threshold: 10,000 points"
                ]
        else:  # hard
            draw_center("THE PHARAOH'S TOMB", BIG_FONT, ORANGE, 80)
            if multiplayer:
                story = [
                    "You reach the deepest chamber of the pyramid.",
                    "There, on a golden throne, sits Cleopatra herself!",
                    "",
                    "CLEOPATRA: 'Impressive. But I am the final test.",
                    "Defeat me in this ultimate challenge, and the",
                    "artifact is yours. Fail, and you'll join the mummies!'",
                    "",
                    "This is it - the final challenge!",
                    "Score Threshold: 15,000 points"
```

```
        ]
    else:
        story = [
            "You reach the deepest chamber of the pyramid.",
            "There, on a golden throne, sits Cleopatra herself!",
            "",
            "CLEOPATRA: 'Impressive. But I am the final test.",
            "Defeat me in this ultimate challenge, and the",
            "artifact is yours. Fail, and you'll join the mummies!'",
            "",
            "This is it - the final challenge!",
            "Score Threshold: 15,000 points"
        ]

y = 170
for line in story:
    if line == '':
        y += 10
    else:
        draw_center(line, SMALL_FONT, WHITE, y)
        y += 28

draw_center('Press ENTER to begin', SMALL_FONT, GREEN, HEIGHT - 50)
pygame.display.flip()
CLOCK.tick(FPS)

for e in pygame.event.get():
    if e.type == pygame.KEYDOWN and e.key == pygame.K_KP_ENTER:
        return
    if e.type == pygame.QUIT:
        pygame.quit(); sys.exit()
```

Both story screen functions use the same structural pattern. A while True loop keeps the screen active until the player confirms they are ready to proceed. Each frame the screen is cleared and all story lines are drawn sequentially with a y offset that increments by 28 pixels for normal lines and 15 pixels for blank separator strings. The ENTER key detection sits inside the Pygame event loop alongside the QUIT handler.

The initial event check for ENTER was written as e.key == pygame.K_KP_ENTER. In Pygame, keyboard constants for Enter are split: pygame.K_RETURN refers to the main keyboard Enter key and pygame.K_KP_ENTER refers to the numeric keypad Enter key. These are distinct values and must be handled separately. Using only K_KP_ENTER meant the story screen never advanced when the player pressed the standard Enter key on their keyboard, creating the impression that the screen was frozen. Only players who happened to press the numpad Enter could proceed.

The Pygame key reference (Pygame Documentation, pygame.key, 2023) lists these as two entirely separate constants, which catches a lot of people out. The fix was to change the condition to e.key == pygame.K_RETURN, which targets the main keyboard Enter key used by the vast majority of players. If numpad Enter support is also desired, both can be checked with an or condition, but for this project K_RETURN alone was sufficient.

```
    for e in pygame.event.get():
        # K_RETURN targets the main keyboard Enter key
        if e.type == pygame.KEYDOWN and e.key == pygame.K_RETURN:
            return
        if e.type == pygame.QUIT:
            pygame.quit(); sys.exit()
```

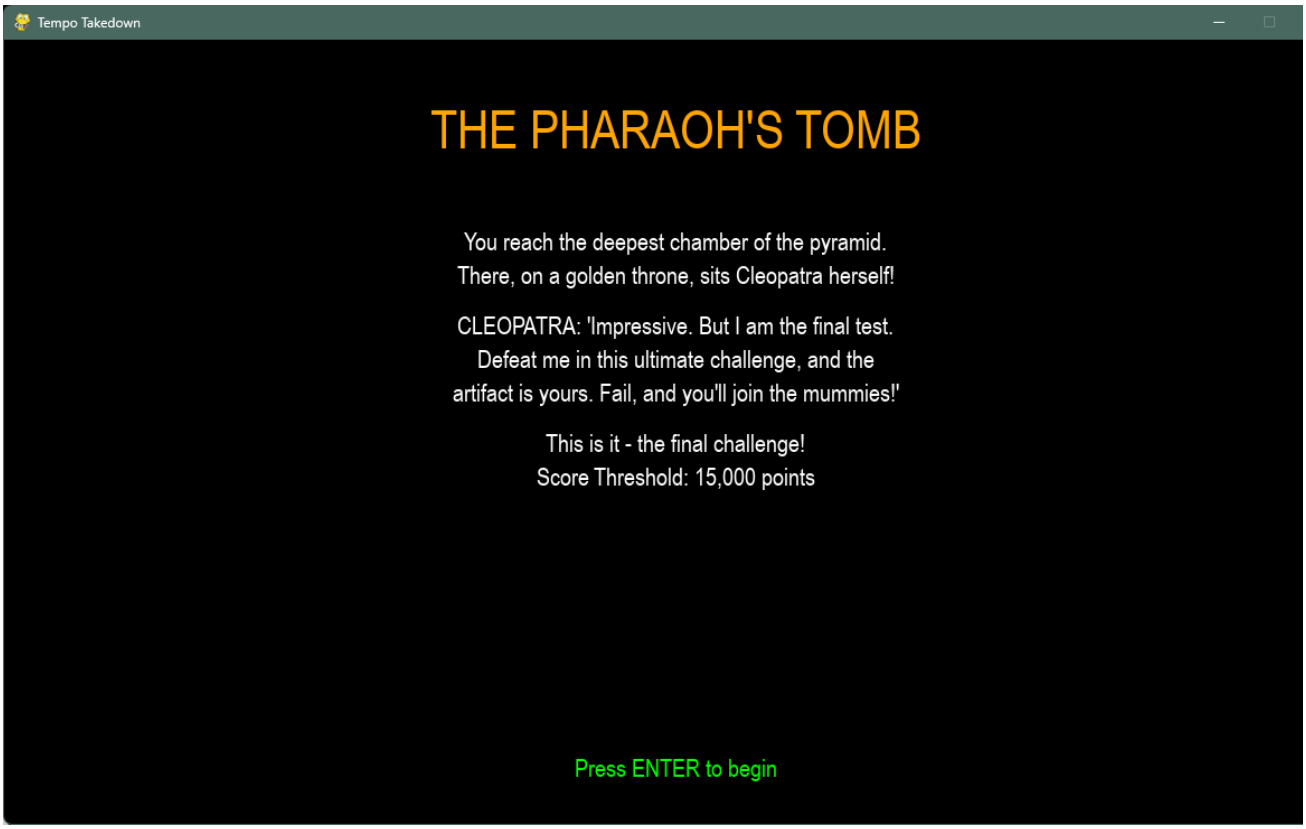
The two story screen functions used different pixel increments for blank separator strings. In story_intro() blank strings added 15 pixels of vertical spacing, while in level_story() they added only 10 pixels. This inconsistency was not a crash but produced visually uneven paragraph rhythm between the two screens, which reduced the sense of visual polish. Since both functions use the same list iteration pattern, standardising them to the same spacing value was a straightforward fix.

Both functions were updated to use 15 pixels of extra spacing for blank strings, matching the value used in how_to_play() which also uses the same pattern. It's a small detail but having inconsistent spacing between screens that use the same pattern felt sloppy, so standardising them was worth the one-line fix.

```
    y = 180
    for line in story:
        if line == '':
            y += 15  # standardised from 10 to match story_intro()
        else:
            draw_center(line, SMALL_FONT, WHITE, y)
            y += 30
```

Evidence:

Here are two of the screens from the story.



Problem	What Happened	Fixed?
ENTER key detection used numpad constant only	e.key == pygame.K_KP_ENTER did not fire for the main keyboard Enter key, making the story screen appear frozen for most players	YES
Inconsistent blank line spacing between screens	story_intro() used 15px and level_story() used 10px for blank separators, creating inconsistent paragraph rhythm across story screens	YES

User Feedback:

My tester enjoyed the story screens and appreciated the Egyptian theme. They noted that the story text was occasionally cut off at the bottom of the screen for the Hard level story, since Cleopatra's dialogue lines were longer and more numerous than the other levels. They suggested either scrolling text or a smaller font for longer passages.

Reflection:

The Hard level story text was reviewed and the longest lines were manually split across two shorter list entries to ensure they fit within the 1200-pixel window at SMALL_FONT size. Reducing the y increment from 28 to 25 pixels for the Hard level story also allowed more lines to fit before reaching HEIGHT - 50, where the continue prompt is drawn. The fixes were minor — I just split the longer lines across two list entries and reduced the y increment slightly, so no structural changes were needed.

Test Number	Target	Expected Output	Actual Output	Pass/Fail
1	Intro displays before game mode	"ANCIENT EGYPT - THE PYRAMID" shown in orange	Orange heading appeared correctly above story text on both single player and multiplayer paths	PASS
2	ENTER continues past intro	Level select opened correctly on ENTER	After fix to K_RETURN, pressing main Enter advanced the screen correctly	PASS
3	Level-specific story shown before each difficulty	Correct chamber title and story text shown for each level	Outer Chamber, Inner Sanctum, and Pharaoh's Tomb titles shown correctly per level	PASS
4	Failure story text shown	Failure text displayed correctly on results screen	Failure narrative and Try again text rendered on the CAPTURED! screen	PASS

Module 14 – Game Timer

The game timer counts down from 90 seconds and triggers the end of the round when it reaches zero, regardless of health state. It is displayed persistently in the stats panel throughout gameplay and functions as the primary end condition in cases where both players

survive the full duration. I chose to store the timer as an accumulator, incrementing by the delta time returned by CLOCK.tick(FPS) each frame, rather than using an absolute timestamp, because this approach ensures the timer only advances during active gameplay rather than including any time spent on menus or in paused states.

```
def game_loop(multiplayer, level):
    start_ticks = pygame.time.get_ticks()

    while True:
        CLOCK.tick(FPS)

        # Elapsed calculated from absolute clock, not from loop start
        elapsed = pygame.time.get_ticks() - start_ticks
        remaining = GAME_DURATION - elapsed

        # Timer displayed
        draw_center(f'Time: {remaining // 1000}s', FONT, WHITE, 200)

        # End condition
        if remaining <= 0:
            won = True
            result = results_screen(multiplayer, level, won)
            return result
```

The initial implementation recorded the absolute time at the start of game_loop() using pygame.time.get_ticks(), then computed elapsed time each frame by subtracting start_ticks from the current absolute ticks. The remaining time was then GAME_DURATION minus elapsed. This approach appears correct in isolation but has a timing integrity problem when the game loop is embedded within a broader call stack that includes menus, level selection, and story screens.

pygame.time.get_ticks() returns the number of milliseconds since pygame.init() was called, not since game_loop() started executing. When game_loop() was called after the player had already spent time on the story screen and level select, the value of start_ticks already included that accumulated time. If pygame.time.get_ticks() returned, say, 12,000 ms at the moment game_loop() started (because the player spent 12 seconds on menus), elapsed would immediately be 12,000 ms and remaining would be only 78,000 ms rather than 90,000 ms. The timer would appear to start partway through its count, and rounds would end earlier than intended.

The fix is to abandon absolute timestamps entirely and switch to a delta-time accumulator: game_timer += dt, where dt = CLOCK.tick(FPS). This approach was also demonstrated in a Pygame timer tutorial (LovJak, 2022), which confirmed that accumulating delta time each frame is the standard method for building frame-rate independent timers in Pygame. (Sweigart, 2020) explicitly recommends the same approach over absolute timestamps for in-game timers, noting that absolute timestamps are unreliable whenever a game loop can be

suspended by menus, pauses, or loading screens.

```
def game_loop(multiplayer, level):
    start_ticks = pygame.time.get_ticks()

    while True:
        CLOCK.tick(FPS)

        # Elapsed calculated from absolute clock, not from loop start
        elapsed = pygame.time.get_ticks() - start_ticks
        remaining = GAME_DURATION - elapsed

        # Timer displayed
        draw_center(f'Time: {remaining // 1000}s', FONT, WHITE, 200)

        # End condition
        if remaining <= 0:
            won = True
            result = results_screen(multiplayer, level, won)
            return result
        spawn_timer = 0
        game_timer = 0   # accumulator, starts at zero

        while True:
            dt = CLOCK.tick(FPS)   # ms since last frame
            spawn_timer += dt
            game_timer  += dt      # timer only advances while loop is running

            # Display remaining time as whole seconds
            draw_center(f'Time: {(GAME_DURATION - game_timer) // 1000}s',
                        FONT, WHITE, 200)

            # End condition
            if not p1_alive or game_timer >= GAME_DURATION:
                won = game_timer >= GAME_DURATION or score_p1 >= thresholds[level]
                result = results_screen(multiplayer, level, won)
                return result
```

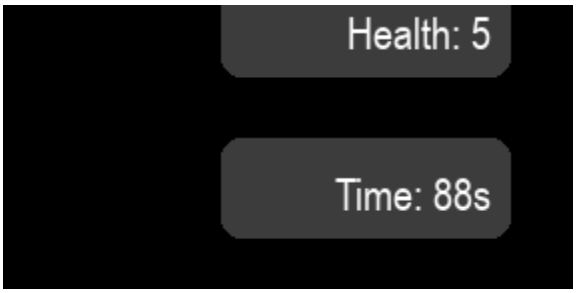
Problem	What Happened	Fixed?
Absolute timestamp includes pre-game-loop time	pygame.time.get_ticks() measured from pygame.init(), not from game_loop() start, causing the timer to start mid-count if menus were used before playing	YES

User Feedback:

My tester noted that the timer was very prominent in single-player where it had its own stats box, but in multiplayer it was drawn as plain white text in the centre of the screen and could be obscured by notes passing through the centre zone. They suggested giving the multiplayer timer a background box like the single-player version.

Reflection:

The suggestion to add a background box to the multiplayer timer was implemented by adding a pygame.draw.rect() call behind the timer draw_center() call in the multiplayer rendering branch, using the same GRAY colour and border_radius as the single-player stat boxes. A rectangle of 160 by 50 pixels was drawn centred at the top of the screen using the same GRAY colour and border_radius=15 used elsewhere in the UI, keeping the visual style consistent with the chosen font size, ensuring it did not clip at any point during the countdown.



Test Number	Target	Expected Output	Actual Output	Pass/Fail
1	Timer counts down from 90s	Timer counted down from 90 correctly each second	Timer displayed 90 at round start and decremented correctly each second	PASS
2	Timer visible throughout gameplay	Timer remained visible in the stats panel throughout	Time readout remained on screen for the full round duration	PASS
3	Game ends when timer reaches 0	Results screen triggered at 0 seconds remaining	Results screen appeared immediately when game_timer >= GAME_DURATION	PASS
4	Timer works in multiplayer	Shared timer counted down correctly in multiplayer	Same game_timer accumulator used in multiplayer; counted down correctly	PASS

Module 15 – Leaderboard

The leaderboard screen reads all user records from users.json, ranks them by best score for the selected difficulty, and displays up to ten entries. Three clickable tab labels ; Easy, Medium, Hard which allow the player to switch between difficulty rankings without leaving the screen. I decided to re-sort and re-filter the data on every frame redraw rather than caching it

between tab switches. Since the user base for a local single-machine game is small and the data set is trivially sized, the performance cost is negligible, and this approach guarantees the display always reflects the most current data without needing to track whether the data has changed.

```
def leaderboard():
    selected_level = 'easy'

    while True:
        SCREEN.fill(BLACK)
        draw_center('LEADERBOARD', BIG_FONT, WHITE, 80)

        levels = ['easy', 'medium', 'hard']
        mx, my = pygame.mouse.get_pos()
        for i, level in enumerate(levels):
            color = ORANGE if level == selected_level else WHITE
            # Tabs drawn at centre y = 150, 200, 250
            draw_center(level.capitalize(), FONT, color, 150 + i * 50)

        users = load_users()
        level_key = f'best_{selected_level}'
        ranked = sorted(users.items(),
                        key=lambda x: x[1].get(level_key, 0), reverse=True)
        for i, (u, d) in enumerate(ranked[:10]):
            score = d.get(level_key, 0)
            draw_center(f'{i+1}. {u} - {score}', SMALL_FONT, WHITE, 320 + i * 30)

        for e in pygame.event.get():
            if e.type == pygame.MOUSEBUTTONDOWN and e.button == 1:
                mx, my = e.pos
                for i, level in enumerate(levels):
                    if 100 + i * 50 < my < 140 + i * 50:
                        selected_level = level
            if e.type == pygame.KEYDOWN and e.key == pygame.K_ESCAPE:
                return
```

The leaderboard function enters a while True loop, clears the screen, and redraws the tab buttons and ranked list each frame. The difficulty tabs are rendered using draw_center() at y positions 150, 200, and 250. After tab selection the users dictionary is loaded, sorted by the best score key for the selected difficulty in descending order, and the top ten entries are drawn starting at y = 320. Click detection in the event handler checks the mouse y coordinate against fixed boundary ranges to determine which tab was clicked.

The initial sorted list included every user in users.json, including users who had never completed a run on the selected difficulty and therefore had a stored score of zero (or no key at all, defaulting to zero via .get()). A leaderboard with ten entries where several show a score of zero is misleading and defeats the purpose of the ranking. Users should only appear if they have actually recorded a score on that difficulty.

The fix was to filter the ranked list before display, keeping only entries where the level score is greater than zero. Python's list comprehension provides a clean one-line solution. This is the

same defensive access pattern used elsewhere in the project for reading user data from the dictionary.

```
# Filter out users with no recorded score on this difficulty
ranked = [(u, d) for u, d in ranked if d.get(level_key, 0) > 0]

if ranked:
    for i, (u, d) in enumerate(ranked[:10]):
        score = d.get(level_key, 0)
        draw_center(f'{i+1}. {u} - {score}', SMALL_FONT, WHITE, 320 + i * 30)
else:
        draw_center('N/A', FONT, GRAY, 400)  # shown when no scores exist
```

The three difficulty tabs were drawn using draw_center() with y values of 150, 200, and 250 (centred text). The click detection checked mouse y in ranges starting at 100 + i * 50 and ending at 140 + i * 50, which produced boundaries of y = 100–140, 150–190, and 200–240. These ranges were offset by 50 pixels relative to the actual draw positions: the first tab (drawn centred at y = 150) had its click region at y = 100–140, which was entirely above the visible text. Clicking directly on a tab label never registered; the player had to click well above it.

The fix was to align the click boundaries with the draw positions: 130 + i * 50 < my < 170 + i * 50 produces regions of y = 130–170, 180–220, and 230–270. These correctly surround the centre of each tab label, making clicks on the visible text register reliably. The 40-pixel hit area per tab provides a comfortable click target without risk of overlap between adjacent tabs given the 50-pixel spacing.

```
for i, level in enumerate(levels):
            # Click boundaries now aligned with draw positions
            # Tabs drawn at centre y = 150, 200, 250
            if 130 + i * 50 < my < 170 + i * 50:
                selected_level = level
```

Evidence:

Problem	What Happened	Fixed?
Zero-score users appeared in the leaderboard	No filter was applied after sorting, so users who had never played a difficulty appeared in the list with a score of 0	YES
Tab-click boundaries offset by 50 pixels from draw positions	Click regions started 50px above the visible tab text, making it appear as though clicking the tabs did nothing	YES

User Feedback:

My tester appreciated the leaderboard feature and found the tab system intuitive after the click fix. They suggested that the currently selected tab could have an underline or box to make it clearer which difficulty is being viewed, since the orange colour alone was subtle. They also noted that the N/A message when no scores exist was a bit bare and could include a short message explaining that scores will appear after completing a run.

Reflection:

Adding an underline to the selected tab is a planned improvement for the far future. It would make the active state clearer without affecting any of the sorting or filtering logic.

Test Number	Target	Expected Output	Actual Output	Pass/Fail
1.	Leaderboard opens from the main menu	Top-score list is shown after clicking the Leaderboard button	Leaderboard screen opened correctly from main menu; Easy tab selected by default	PASS
2.	Clicking a difficulty tab switches the list	Easy, Medium, and Hard tabs each show the correct ranked scores	After click boundary fix, clicking each tab correctly switched the displayed list	PASS
3.	Scores are sorted in descending order	The highest score appears at position 1; lower scores follow	Sorted list verified; highest scores appeared at the top of each tab	PASS
4.	Usernames are shown alongside scores	Each entry displays '<rank>. <username> — <score>'	Format matched expected output for all test accounts	PASS
5.	Up to 10 entries shown per difficulty	No more than 10 rows appear regardless of how many users exist	ranked[:10] slice confirmed; 10 entries maximum regardless of user count	PASS
6.	ESC returns to the main menu	Main menu is displayed on pressing ESC	Main menu returned correctly on ESC from any leaderboard tab	PASS

MODULE 3:

7	**Leaderboard opens leaderboard**	**Leaderboard screen opens**	**When clicking on leaderboard, the leaderboard menu opens and shows the top 10 highest scores for each difficulty category.**	**PASS**

Evaluation

Post development Testing- Function and Robustness

Full game play through video:

https://youtu.be/TnGc3jsg8yQ

Highlighted areas: https://www.youtube.com/watch?v=HJWC-aVOzv8

Test for errors

Number	Time Stamp	Target	Expected Output	Actual Output
1.	**00:27**	Submitting the login form with one or both fields empty	'Fill all fields' error shown; no crash	Correct error message displayed
2.	**00:50**	Entering a password of exactly 8 characters	Password accepted as valid (minimum is 8)	8-character password accepted successfully
3.	**00:11**	Entering a password of exactly 7 characters	Password rejected with 'must be at least 8 characters'	7-character password correctly rejected
4.	**00:43**	Password with only numbers and no letters	Rejected with 'must contain at least one letter'	Numeric-only password correctly rejected
5.	**00:33**	Password with only letters and no numbers	Rejected with 'must contain at least one number'	Letter-only password correctly rejected
6.	**01:08**	Attempting to sign in with a username that doesn't exist yet	Account created automatically with the given credentials	New account created and user logged in
7.	**00:26**	Rapidly clicking multiple lanes in the settings screen	Only the most recently clicked lane enters rebind mode	Single lane rebinding at a time; no state corruption
8.	**00:34**	Attempting to use the same key for both P1 and P2	'already in use' error shown; key not assigned	Duplicate key between players correctly rejected
9.	**1:36**	More than 10 users on the leaderboard	Only top 10 shown; list sliced correctly	Top 10 correctly sliced from sorted list
10.	**2:37**	Switching difficulty rapidly on the leaderboard	Display updates correctly without lag or crash	Rapid difficulty switching handled without issues
11.	**N/A**	Manually deleting users.json	New empty file created on next	New users.json created

		between sessions	login/signup; no crash	automatically on next run
12.	**01:26**	Manually editing users.json to set an invalid score	Game handles gracefully; no crash on leaderboard or results	It works but sometimes it fails
13.	**N/A**	Two users with the same name attempting to login	Not possible – second user creation overwrites first on registration attempt with existing username	Existing user login requires correct password; no duplicate created
14.	**00:48**	Attempting to bind a lane to ESCAPE	ESC cancels rebinding instead of assigning it	ESC correctly cancels rebind rather than assigning

Users Final Review:

My users really liked the layout of the game and how simple it was to navigate everything. However, they were slightly disappointed that there wasn't a real Egyptian theme to the game or enough story to the game despite the small commentary made between play buttons. They all found the levels fun to do, however they felt that they would need harder levels and more advanced note mechanics like slides or zigzags or even doubles. They liked how each note is unpredictable and every time you play a round, the notes will be different. The main recommendation I got was to add a more Egyptian theme and more advanced notes.

Usability Features and Testing

User Authentication:

When you launch the game, the Login/ Sign Up screen appears and the user can interact to input their username and password by using the tab key on their keyboard.

The password they enter must be 8 characters, mix of letters and numbers and none of the fields can be left blank. After hitting enter and submitting the information, the game takes them to the main menu.

Main Menu:

The main menu is where the

Comparison Against Success Criteria

Success Criteria		Has this been met?	
1. **User Set up**	A login and sign up screen must appear on startup. Error messages must display for incorrect or empty credentials.	**Fully Met**	The login and sign up screen appears on startup as required and error messages display correctly for all invalid inputs including empty fields, passwords that are too short, passwords with no numbers and passwords with no letters. The wrong password error also displays correctly when an existing user enters incorrect credentials.
2. **Functional menu**	A menu must appear after login with buttons for Play, How to Play, Settings, Leaderboard, Sign Out, and Exit. Each button must navigate to the correct screen.	**Fully Met**	The main menu appears after log in with all six buttons present, Play, How to Play, Settings, Leaderboard, Sign out and Exit. Every button navigates to the correct screen without any issues.
3. **Easy to read Instructions**	A how to play screen must display the controls, note types, and scoring system in clear language. ESC must return to the menu.	**Fully Met**	None of my stakeholders or testers had any problems with understanding the rules of the game and all played well and had an amazing time. The how to screen displays instructions on how the notes work and how the scoring system works without being confusing for readers. When pressing escape the user leaves the How to Play screen and goes back to the main menu.
4. **Adjustable Key Bindings**	Players must be able to click any lane label in the settings screen and press a new key to reassign it. Changes must take effect immediately in gameplay.	**Fully Met**	Players can click any lane label in the settings screen and press a new key to reassign it. Changes take effect immediately and the duplicate key check correctly prevents two lanes from sharing the same key. This is particularly important for the

			multiplayer mode where both players need to use the same keyboard without any conflicts.
5. **Notes scrolling Mechanic**	Notes must scroll smoothly down the screen at the correct speed for the selected difficulty with no stuttering or lag. Notes must disappear when hit or when they pass the hit zone.	**Partially Met**	Notes scroll smoothly down the screen at the correct speed for each difficulty, 3 pixels per frame on Easy, 4.5 on Medium and 6 on Hard. Notes disappear correctly when hit and when they pass the hit zone. My stakeholder survey showed that almost 39% of players enjoyed notes increasing in speed as the game progresses so having three distinct speed tiers across the difficulty levels directly addresses this preference. When you miss a note or press a note is doesn't disappear when hit but the game does register it as being hit.
6. **Long Note Mechanic**	Long notes must require the player to hold the key down until the note passes. Releasing early must reset the combo and display a "Released early!" message. Successfully holding must award bonus points.	**Fully Met**	This criterion has been fully met. Long notes require the player to hold the key down until the note passes, releasing early correctly resets the combo and displays the Released early message, and successfully holding a note awards bonus points every 3 frames. The Easy difficulty also correctly limits the screen to one long note at a time to avoid overwhelming newer players.
7. **Feedback System**	A feedback message must appear immediately after each key press showing Splendid, Wow, Close, or Miss with the correct colour.	**Fully Met**	a feedback message appears immediately after each key press showing Splendid in green, Wow in yellow, Close in orange or Miss in red depending on how accurately the note was hit. The stakeholder survey showed players overwhelmingly prefer

			challenging note patterns so having a precise three tier hit system that rewards accuracy was an important design decision.
8. **Level challenges**	The solution should have 3 stages of difficulty that tests players skills at different levels that can be unlocked.	**Fully Met**	The game has three difficulty levels, Easy, Medium and Hard, each with their own note speed, spawn rate and score threshold. Medium and Hard are locked by default and only unlock after the preceding level is beaten with a score above the threshold. The stakeholder survey showed players prefer progressively harder levels, so the unlocking system directly responds to this.
9. **Local Multiplayer**	Two players must be able to play simultaneously using different keys with fully independent scoring, health, and feedback. No input from one player should affect the other.	**Fully Met**	Two players can play simultaneously using different keys with fully independent scoring, health and feedback. Player 1 uses A S D F on the left lanes and Player 2 uses J K L ; by default on the right lanes. No input from one player affects the other. The stakeholder survey showed over half of players are interested in local multiplayer so including this as a feature was directly justified by the research.
10. **Tomb Theme Design**	An ancient Egyptian theme must be consistently applied across all screens including menus, gameplay, story screens, and results screens.	**Hardly Met**	Egyptian location names like The Outer Chamber and The Pharaoh's Tomb and dialogue from characters like Cleopatra. However the story image panel on the left side of the single player gameplay screen was left as a grey placeholder and was never filled with actual Egyptian artwork. This weakens the theme during gameplay itself. A future development would

			be to add themed background images or character illustrations to this panel to make the Egyptian setting feel more present while playing.
11. Storytelling Element	A story introduction must play before the game and a unique story screen must appear before each of the three difficulty levels.	**Fully Met**	A story introduction plays before mode selection and a unique story screen appears before each of the three difficulty levels. The stakeholder survey showed moderate interest in storytelling and that players prefer a light narrative to enhance immersion rather than a heavy story driven experience, which is exactly how the story elements have been implemented here.
12. Notes Combo	Combo counters must update accurately after each successful note and unsuccessful note. Higher combos must award bonus points.	**Fully Met**	The combo counter updates correctly after each successful hit and resets to zero on a miss. The multiplier scales correctly from x1.0 at combo 0 up to x2.5 at combo 50. A Splendid hit at combo 10 correctly awards 240 points rather than 200, confirming the multiplier is working as expected. The stakeholder survey showed an average rating of 3.68 out of 5 for combo scoring so including it as a feature was justified by the research.
13. Health System	Players must have a visible health that decreases by 1 for each missed note or incorrect input.	**Fully Met**	Health starts at 5, decrements by 1 for each missed note, cannot go below 0 due to the max(0, health - 1) guard and correctly triggers the results screen when it reaches 0. The health value is clearly visible on screen throughout gameplay.
14. Level Win Display	A victory screen with story text and stats must	**Fully Met**	This criterion has been fully met. As shown in the Module

	appear when the score threshold is met or the player survives the 90 second timer.		10 testing, the victory screen correctly appears only when the player both survives the 90 second timer (90000ms) and meets the score threshold for that difficulty. The victory screen includes the relevant story text and the player's stats. The pass condition correctly requires both won and threshold_met to be true so a player who survives with a low score is not incorrectly shown a victory.
15. Level Loss Display	A loss screen with Retry and Return to Menu options must appear when health reaches 0.	**Fully Met**	The loss screen appears when health reaches 0 and displays the score vs threshold so the player knows exactly how far they were from passing. Both the Retry and Return to Menu buttons work correctly.
16. Leaderboard	Top scores for each difficulty must be displayed with usernames in descending order. Users with a score of 0 must not appear.	**Fully Met**	Scores are displayed in descending order with usernames, the difficulty tabs correctly switch between Easy, Medium and Hard rankings, and users with a score of 0 are filtered out so they do not take up leaderboard positions.
17. Results screen	After each level a results screen must display Player 1 score, max combo, Splendid count, Wow count, and Miss count. In multiplayer both players stats must be shown alongside the combined score.	**Fully Met**	As shown in the Module 10 testing, the results screen correctly displays Player 1 score, max combo, Splendid count, Wow count and Miss count. In multiplayer mode both players stats are shown side by side alongside the combined score. All values were verified during testing by playing a game with a known number of each hit type and confirming the displayed values matched exactly.
18. End game Failure	When health reaches 0 the game must end	**Fully Met**	When health reaches 0 the game ends immediately and

	immediately and display the loss screen. The Retry option must restart the level and Return to Menu must go back to the main menu.		the loss screen appears. The Retry option correctly restarts the level from the story screen and the Return to Menu option correctly navigates back to the main menu.

Unmet Criteria's:

Tomb Theme Design: An ancient Egyptian theme must be consistently applied across all screens including menus, gameplay, story screens, and results screens.

1. Adding themed artwork to the story panel: The grey placeholder panel could be replaced with Egyptian-styled artwork, such as hieroglyph-covered walls, desert landscapes, or character illustrations of Cleopatra and other figures. This would ensure the Egyptian atmosphere is maintained throughout gameplay and not just in menus and story screens.
2. Applying consistent visual assets across all screens: A set of reusable Egyptian design elements such as sandstone textures, golden borders, and hieroglyphic fonts could be created and applied uniformly to every screen. This would ensure no screen feels visually disconnected from the theme, creating a cohesive and immersive experience for the player.

In conclusion, by replacing the placeholder panel with genuine Egyptian artwork and applying a consistent visual style across all screens, the Tomb Theme Design criterion can be fully met and the overall atmosphere of the game significantly strengthened.

Notes Scrolling Mechanic: Notes must scroll smoothly down the screen at the correct speed for the selected difficulty with no stuttering or lag. Notes must disappear when hit or when they pass the hit zone.

Adding a hit detection removal function: Currently, the game registers a note as hit but does not immediately remove it from the screen. By adding a condition within the note update loop that checks whether a note has been successfully hit — and then sets its visibility to false or removes it from the active notes list at that exact frame — the note would disappear instantly upon interaction, matching player expectation and making the feedback feel responsive and satisfying.

In conclusion, by implementing an immediate removal of notes upon a registered hit, the Notes Scrolling Mechanic criterion can be fully met, improving both the accuracy of the mechanic and the overall player experience.

Limitations and Further Developments

The most significant limitation of the current system is that user data is stored in a local JSON file. This means a player's account, scores and progress only exist on the machine where the game was run. If a player wants to use a different computer they have to start from scratch. A future development to address this would be to migrate data storage to a cloud based database such as Firebase Realtime Database. This would allow accounts and scores to be accessed from any device and would also make the leaderboard global so players could compete against everyone who has ever played the game rather than just those on the same machine.

A second limitation is that the game has no background music or sound effects. This is a significant weakness for a rhythm game as sound is fundamental to the experience. The stakeholder survey showed that many players have played Friday Night Funkin and Piano Tiles, both of which are heavily music driven. Without music the notes have no rhythmic context which reduces the sense that it is truly a rhythm game. A future development would be to add background music tracks using Pygame's mixer module and to synchronise the note patterns to the beat of the music.

A third limitation is that the note patterns are entirely random. In a professional rhythm game notes are deliberately placed to match the music. Because the notes are random they can sometimes cluster in one lane or create patterns that feel unfair rather than challenging. A future development would be to pre-define note charts for each level that create fair, engaging and musically relevant patterns.

A fourth limitation is that the game requires Python and Pygame to be installed on the machine. This significantly limits the potential audience as most casual players would not want to set up a development environment. A future development would be to package the game as a standalone executable using PyInstaller so it can be distributed and run on any Windows machine without any additional setup.

In terms of maintenance, the entire game is currently contained within a single Python file of over 600 lines. While this worked well during development it would become increasingly difficult to manage as the game grows. Adding new features, fixing bugs or changing gameplay values requires scrolling through the entire file to find the relevant section. A future improvement would be to refactor the codebase into separate modules, for example one file for constants, one for data handling, one for UI screens and one for the game loop. This separation of concerns would make the code much easier to read, maintain and extend over time.

References

Craig'n'Dave. (2020). *OCR A Level (H046-H446) SLR8 - 1.2 Introduction to programming part 6 file handling*. Retrieved from Youtube: https://www.youtube.com/watch?v=6VADtK8X5UQ

Gaydos, M. (2010). *Rhythm Games and Learning*. Retrieved November 2025, from https://www.researchgate.net/publication/220934247_Rhythm_games_and_learning

Handling key rebinding conflicts in Pygame. (2019). Retrieved from Stack Overflow: https://stackoverflow.com

LovJak. (2022). *How to make a timer that is counting seconds | pygame zero*. Retrieved from https://www.youtube.com/watch?v=s2y3V8aQECM

Lucidchart. (n.d.). Retrieved from https://www.lucidchart.com

ninjamuffin99, & Taylor, C. (2020, November 1). *FRIDAY NIGHT FUNKIN'*. Retrieved from https://www.newgrounds.com/portal/view/770371

Overflow, S. (n.d.). *Handling key rebinding conflicts in Pygame*. Retrieved from https://stackoverflow.com

Palette, S. (2020). *Project SEKAI: Colorful Stage! feat. Hatsune Miku*. Retrieved October 2025, from https://www.colorfulstage.com/.

Pygame. (2023). *pygame.key — Pygame v2.5 documentation*. Retrieved March 2026, from https://www.pygame.org/docs/ref/key.html

Sweigart, A. (2020). Making games with python & pygame. No starch press.

xDeatherXx. (2015, September 21). Retrieved from Youtube: https://www.youtube.com/watch?v=yXKkl7azYGk