A level OCR Computer Science Project
H446-03

Jachym Tolar
Candidate Number: 5175
Beaumont School : 17511

Contents

Analysis

Problem description and overview

A rhythm game is a subgenre of action games with a music-based element that challenges a players timing and sense of rhythm [1]. For my project, I propose developing a rhythm game where players will have the option of choosing a song from a selection menu and then have the objective of hitting keys in time to the beat of the music and gain score along the way. Objects will appear on the screen in time with the song playing and approach a hit area, corresponding to the beat of the music. If the correct key is clicked when the note is in the hit area of the screen, the player's combo throughout the song is maintained and score increases. Success in the game is measured by how close to the beat the player hits the notes, and if a note is missed (key pressed too late, too early, or not at all) the player loses out on score. The player fails if too many notes are missed in a row. When the player gets to the end of the song, they will be awarded a rank based on their score, accuracy, and consistency.

My aim will be to create a rhythm game with 2 lanes that notes can approach from, a reduction in complexity of the popular mobile rhythm game "piano tiles" [2] except the notes approach from the side, more similar to arcade drumming rhythm games such as "Taiko no Tatsujin: Drum 'n' Fun!" [3]. In line with tropes of the genre's popular games, my solution will be 2 dimensional, and objects will be moving on a background towards a fixed player area. Different types of notes will appear during a song, single notes will require a quick press and long notes require the user to hold down a key until the note is over.

Songs will be added to the game from a midi file, a file format that standardizes digital music communication and stores information about individual tracks, the bpm and length. By previously composing a midi file with drum notes on top of a song's beats in a music producing software, the midi file containing drumbeats can be interpreted in the game as notes. Through this method, stages in the game are stored as modular files that can be read from and modified and displayed as needed. The player will not need to interact with these files as the game will have a stage (song) selection menu.

Rhythm games have often demonstrated uses for rhythmic motor and function training. In a 2017 study [4] examining the potential uses of current rhythm games on the market for retraining motor and cognitive functions in patients with neurodevelopmental disorders (such as Parkinson' and ADHD), the medical effectiveness of various peripheral and output methods was tested. Whilst the study found little potential for current rhythm games to treat patients, if made more precise the foundations of rhythm games can be built to track improvements and help synchronize actions and to rehabilitate co-ordination in patients. Even if rhythm games had no tangible effects on the brain, tempo, and rhythmical understanding, on an individual level, can be improved over time. The study found that rhythm games improved general well-being and rhythmic skills were found linking to general cognitive abilities such as literacy and memory.

Stakeholders

Whilst the games industry grows and rhythm games gain potential, each game's target audience ends up being limited to the player's taste in music. Simply if the player is uninterested by the music, they will be uninterested in the game, which is why rhythm games resolve to target either a popular genre of music (e.g. "muse dash"), incorporate music choice into the story ("rhythm heaven"), or nurture a community of players and have custom user made maps – giving a multitude of variety ("osu!"). For my solution, I will resolve this by proposing genre and conducting interviews, to decide what music my stakeholders would enjoy playing along to.

Specifically, the target audience for my game will be younger rhythm game players, focusing on the lower end of the difficulty spectrum and making the game accessible to inexperienced gamers or people looking to get

into rhythm games. My aim will be to encourage new players with no need for learning techniques or dealing with complicated settings. To do this, I'd make the difficulty increase slowly from song to song and give options to lower difficultly, as well as keeping the UI as minimal and intuitive as possible.

My target age range would be from 10-18 as this is the range of players that would be most interested in a simplistic, colourful rhythm game with an easy to intermediate difficulty. There won't be any violence or frightening images so a PEGI 3 rating would be the most appropriate. Colourful backgrounds and glowing effects when notes are hit correctly will add to the overall appeal of the game to a younger generation, and the effects could be toned down for a more relaxing game.

Ideally this would look like a student who doesn't have much time in the day for listening to music yet wants to take some time to focus on their favourite songs. Perhaps someone who's attention drifts away when listening to music in isolation and would like extended moments to get to know the percussion behind a melody and gain the sense of interacting with the song. A student who's new to rhythm games might not have the time to get into difficult games and might only have a laptop to play games on. A low intensity keyboard game would be best for this user, with features such as geometric appealing aesthetics, low system requirements, various songs to select from, and customisable difficulty settings.

To keep players playing I will provide a variety of difficulty and song choice. The player can gain a sense of achievement from progressing through the stages and setting high scores. In a rhythm game it also important for the music to be synced precisely to the gameplay at all times or the player can be left unsatisfied or frustrated.

Other possible stakeholder groups to consider could be students training to improving their rhythmic timing in a more fun and relaxing way than traditional practise. A two-input key design means the game could be played by people with physical impairments such as those whose movements are restricted to only one hand.

Computational Methods

My game will use the following computational methods, each of which can be applied to my problem to produce an efficient solution, and therefore coding a game is a suitable solution to the problem.

Abstraction and visualisation

Abstraction and visualisation are the removing of unnecessary detail to simplify a problem, and communicating a solution through representations of data.

The key features of my game will be abstracted, in order for the game to be understandable and to provide a platform for the users to interact with it.

- Hit objects in the game will be represented by shapes
- And appear seemingly outside of the game
- Health in the game will be represented by a bar emptying or filling
- The game will be played on a 2D background in 2D space
- Timing of when the notes are hit is a represented by when they are at the hit area, the player can either "feel" the beat or watch when the notes approach
- the exact timing of when a note is hit is represented by the score given to the player
- the players final performance will be given as a single rank (e.g., A, B, C)
- Song length, bpm and number of beats will be hidden

Thinking ahead

Thinking ahead is the process of thorough planning of your code to ensure an efficient outcome, by considering inputs outputs and preconditions.

Inputs considered

- Player inputs a timed key press
- previous time elapsed as an input, the relative position of the song and its measure
- When loading a stage, the correct song file needs to be specified

Outputs considered

- Function that updates the current song position should output a time in seconds and count how many beats have passed in the song

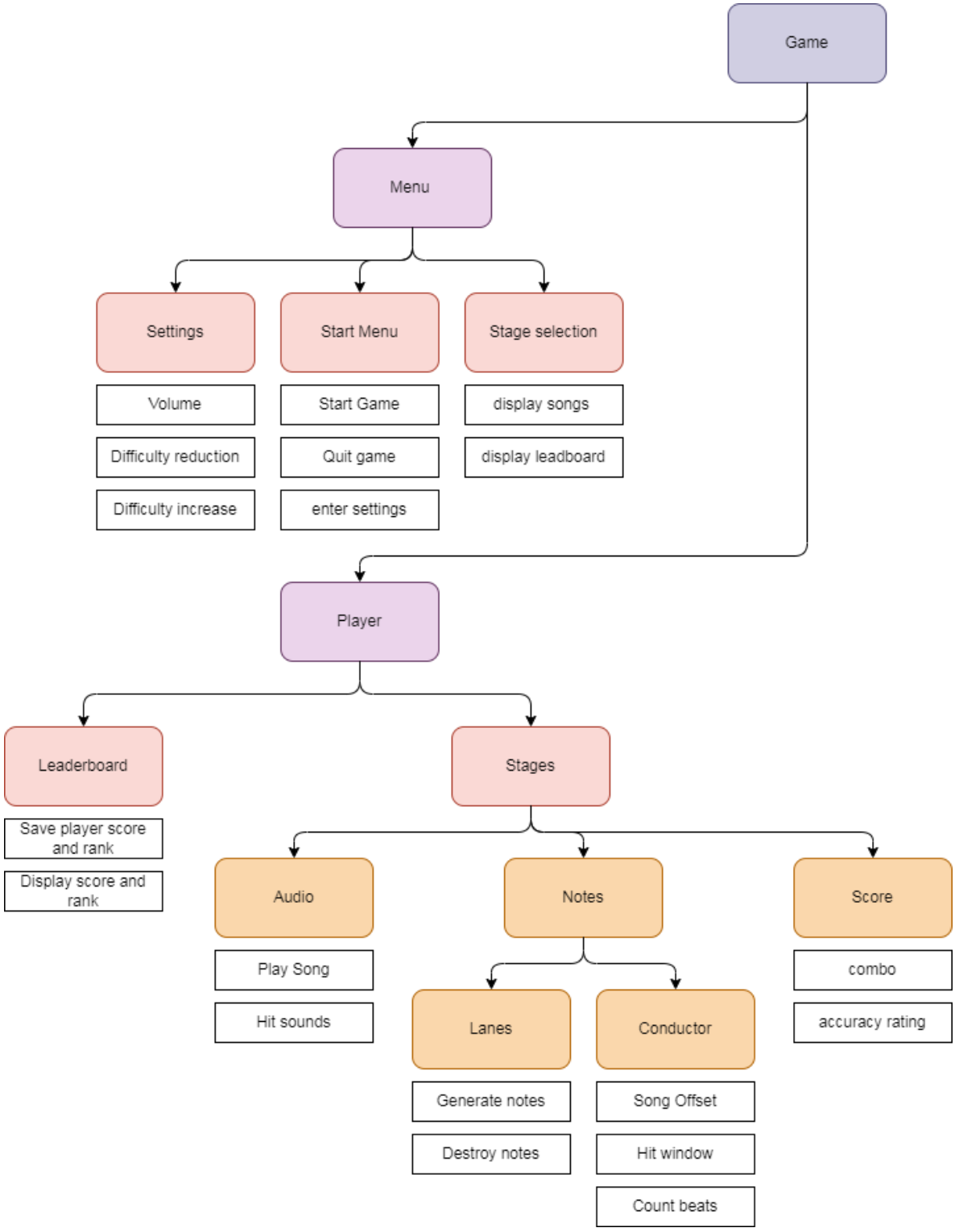
Preconditions considered

- Notes from a song will need be loaded before playing the song, I will be doing this by parsing a JSON file containing the data into a variable and reading from that, this also reduces processing power needed as the file does not have to be loaded multiple times

Thinking procedurally

Thinking procedurally is problem decomposition into smaller, underlying sub problems.

From within the main game loop, the program can be split into individual and self-contained subroutines. Hierarchy chart illustrates how the game will be broken down from the highest level (game) to the lowest.


Game
Menu
Settings
Start Menu
Stage selection
Volume
Difficulty reduction
Difficulty increase
Start Game
Quit game
enter settings
display songs
display leadboard
Player
Leaderboard
Save player score and rank
Display score and rank
Stages
Audio
Play Song
Hit sounds
Notes
Lanes
Generate notes
Destroy notes
Conductor
Song Offset
Hit window
Count beats
Score
combo
accuracy rating

Thinking Logically

Thinking logically is planning the outputs decisions of a program - an evaluation of data to give a conclusion.

Several decisions will be made by the program whilst running, changing how the game behaves.

- Program flow will be determined by the main game loop, which will run once the user enters the main menu and will contain all the functionality such as loading stages, scores and settings. Procedures inside the game loop will continue running until the player exits.
- Game logic is run once the player has selected a stage and ends once the song is over, returning to the main game loop.
- Ranking on the leader board is calculated by user performance at the end of a stage.

Certain conditions must be met in order for the program to progress.

- Users cannot progress past the tutorial without achieving a passing score on the song. By considering a basis argument of a user's current skill level a conclusion can be made as to which levels are too hard for a user.
- The game must check that the song files are installed and in the root folder before loading a stage

Other programming reasoning such as Boolean logic can be used to understand the state of the game.

- If the user is pressing a key and a note is on the screen, it can be deduced that the user is attempting to hit the note, and the program should react accordingly.

Thinking Concurrently

Thinking concurrently is the application of concurrent processing – giving tasks concurrent time slices of processor time, giving the illusion of completing more than one task at a time.

In order to perform efficiently, my project will take advantage of multicore recourses. The game loop will be running on a single thread however more threads could be added for interrelated tasks, causing fewer bottlenecks and reducing idle time. During run time, many processes will be executed concurrently to give the illusion of happening at once.

- Song playing through the entirely of the level whilst
    - Hit sounds play
    - Objects move in unison
    - Score updates
    - health updates
    - accuracy updates
    - background effects happen
- Taking two inputs at once without dropping either and registering them as valid hits on the same beat – this allows for two notes to come at the user at once

Research: Existing Solutions

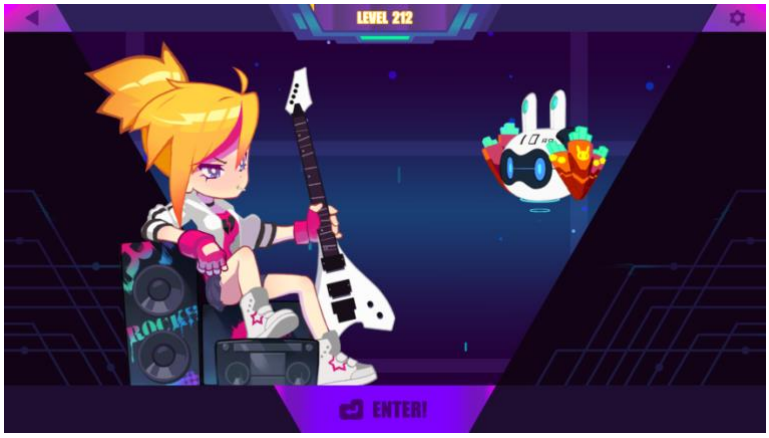
I chose to research 2 games: "Muse Dash", "Osu!Mania", as each of them has features I would like to incorporate into my solution. Both are widespread and appeal to similar stakeholders as I have identified and all of them use side scrolling hit objects, that require a precise key press for the player to succeed. Screenshots are grouped into the features they showcase in the game.

Game 1: Muse Dash

Muse Dash (2018) is a rhythm game developed by "PeroPeroGames" released for iOS, Android, Windows, Mac OS and the Nintendo Switch [5]. It features animated gameplay set in a futuristic world, with horizontal scrolling gameplay, to pop like melodies. I chose this game to research for its similar visual style and game feel to my initial idea, and features I find valuable in rhythm games.

Title screen

The main game can be accessed from the title screen – the title screen is the first screen the user sees when opening the game.



Settings menu can be accessed from the title screen

Song selection screen can be accessed from the title screen.

The game's graphics are colourful, playful and modern looking with hue changing backgrounds and 3D sci-fi elements mixed with an animated style. This makes the game appealing to a large audience, and creates a sense of continuity between to song selection screen and the in game background.

Song selection

Using background graphics keeps the player engaged and the style of animation for the main character appeals to the target audience, who enjoy this style of music.

Settings menu can be accessed from the song selection screen.

Music can be filtered from the selection screen, allowing the user to sort by song name, artist, difficult and favourite music. This allows for users to navigate the music library efficiently.

Each song is represented by a picture icon; songs can be changed with the left and right arrow keys or by scrolling.

A menu such as this one can be represented with a 2-dimensional array and pointer. Moving the pointer through the array represents moving back and forth in the menu one song at a time.

Places on the leader board are ranked globally by score. Accuracy and username are also shown. Having a leader board of some kind in the game would let users see their improvement.

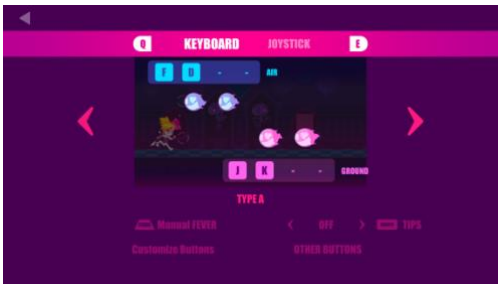
Leader board and difficulty selection

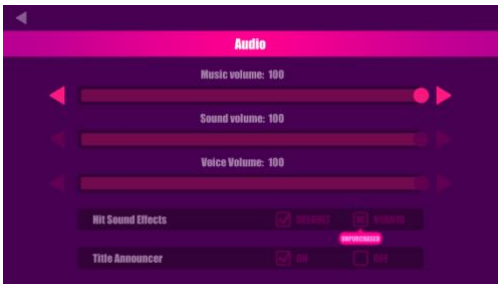
To play a song, the user has to enter a leader board screen and choose a difficulty, represented by a number and different colours.

The user's high score is shown with the rank achieved and highest combo.

However having a separate menu for the leader board can be a disadvantage as it doesn't show the user's high score straight away and can be harder to locate in the menu.

Controls and audio settings





Key binds can be adjusted in the settings menu to allow for any combination of buttons to control the game with. Here a simple button layout is shown which neatly explains positioning.

Audio sliders are split up into master, sound and voice volumes. In my game, these would be master volume, song volume, and effects volume, the option to disable hit sound effects.

Customisable settings are important for accessibility of different user's needs. These settings should also be remembered and be saved into a file to be loaded again on the next play through.

Level loading, paused and failed

As the level loads text prompting the user that the song is starting soon appears. Animations such as the character running onto stage play.

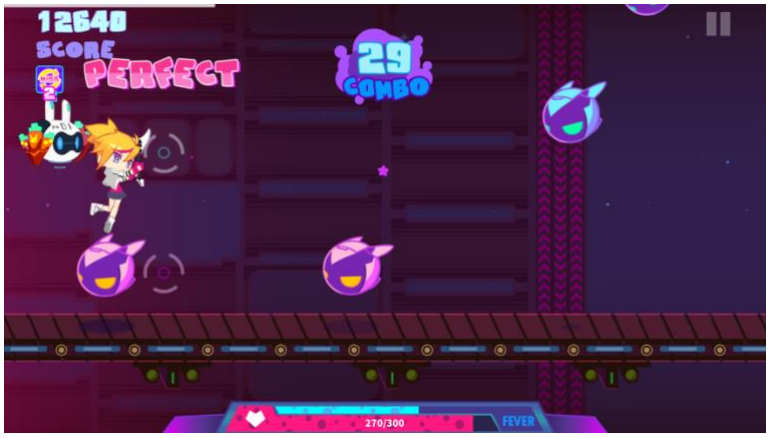
The pause menu has three icons: return, retry and resume. During this screen the rest of the game dims and all motion stop, in a state ready to play from the same place again.

This screen is displayed if the health bar reaches zero and signals user has "lost" and has to try again. From here the same options as the pause menu appear, except without the resume button. This keeps the player from continuously struggling on a song that's too difficult.

Game screen

On the left, total score for the song is tracked, which the player accumulated by hitting the notes to the beat of the song.

The Background contains simple animations and scrolls as the level goes on.

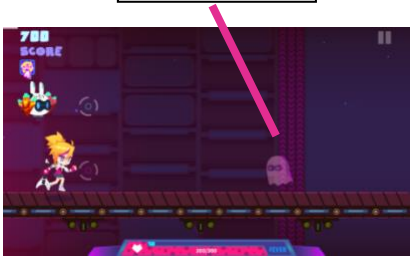


Notes appear on the right in one of two lanes (top and bottom) and move towards the player. This system is one I would like to incorporate into my game as it allows for 2 paths and splits the players focus, whilst only using 2 keys. Muse Dash effectively uses this mechanic of having two sets of notes to hit at once.

When the note is in the circular hit area next to the player, this indicates the player needs to hit the key. This either registers the hit as a perfect, great, pass or a miss. The note is destroyed after being hit.

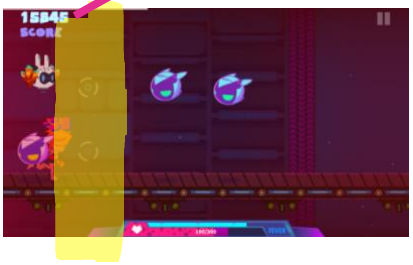
The bottom of the screen contains a fever meter which increases score, I find this addition unnecessary because combo already increases score in proportion to how well the player does. The health meter indicates how many misses long until the player fails the level. I like how the UI is placed in the game with the order of importance being combo, then score and then health. Additionally, the heath bar is good indicator of if a song is too hard for a user, meaning they can see if likely to fail the song or not in a very visual and intuitive way.

Note types



Ghost notes appear periodically and fade into transparency before reaching the hit area. This mechanic means the player has to relay on timing their hit with the music rather than to the location of the note. I would like to add ghost notes as a difficulty option, so players can turn them on and off, with a boost in score for using them.

Hit area



Once notes pass a certain distance beyond the hit area, the player can no longer hit the note and it does damage to the player, signalling a miss and lowering their health. The timing window must be carefully calculated to allow enough time before the notes disappear, especially with faster moving notes, which will inevitably spend less time in the hit area.

Note types

Long notes represent chord and require a continuous input throughout. The timing on long notes is measured when hit and again when released. If the note is released combo is lost but whilst being held combo increases. This note also helps fill down time in songs and keeps the level varied.

Timing windows

Timing windows are a measure of how hard or easy it is to hit a note. In muse dash there are two types of hits that affect the score and feel of the game. The 'perfect' hit occurs when a note is pressed within 100ms of when the beat happens. This gives 50ms either side to be too early or too late. For a lower score, the user can hit the beat too late or early, giving a 'good' rating, for this timing the user has up to 130ms either side. The area for this lesser hit is 1.6x as big as the perfect hit. This feature is important to let users know if they are on time to the music, and gives clear feedback to let the user improve. By making the 'good' area larger than the perfect, the user is rewarding more for getting the more precise timing.

A final screen at the end of the song is shown with the player's score (points gained by hitting notes on time which are multiplied by combo), players accuracy, how many notes are hit on time as a percentage out of 100.

Stage complete

The Rank (C, B, A, S or SS), which awarded based on an algorithm factoring in score and combo, is displayed in the centre as a summary statistic.

The player is then given the option to navigate back to the song selection menu or try the song again.

If the players score is a personal best it gets saved as a high score.

Other noticeable features that improve the game flow and make a more enjoyable game, which would be beneficial to consider for my own solution:

- Options are presented with the use of consistent symbols throughout the game, helping the player to navigate between screens using dedicated inputs.
- When playing a song, the same backgrounds and character animations are used, enforcing a visual story.
- Under different conditions, such as when a different character is selected, score acts differently, increasing at different rates, at the cost of difficulty increasing, for example having less life
- At the end of each level, the same screen is shown, only with a different character if selected and the stats from the most recent run. The use of an individual, re-useable module that shows an end screen with updated text and graphics is important for continuity between levels and for keeping in the main game loop, as the player will have to return to the level select screen

Overall the game's design is catered towards younger audiences, and has minimal feature with a stronger focus on graphics and variation in character selection and score multipliers.

Game 1: Osu!

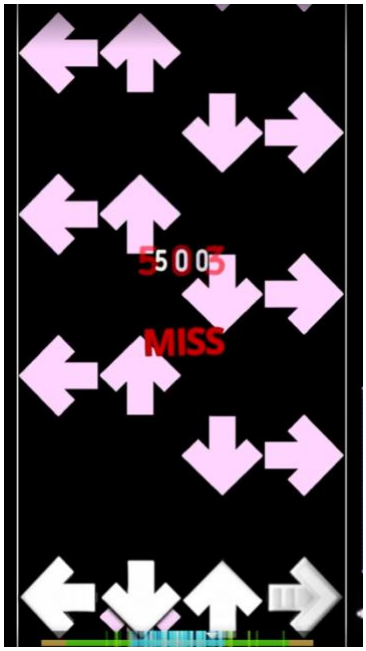
Osu!mania (2007), is a free to play rhythm game released by Dean Herbert and was written using the .NET framework for Windows, MacOS and Linux [6]. The game features scrolling note gameplay, where notes flow down from the top of the screen and must be hit in time with the music similarly to *Muse Dash* but with 4 keys needing to be pressed instead of only 2. The game has a variety of song choice with popular sub culture hits and EDM songs, but the key feature of the game is the ability for users to customise stages, and download user made stages. These are stored in specific song files and read on start up.

Gameplay screen

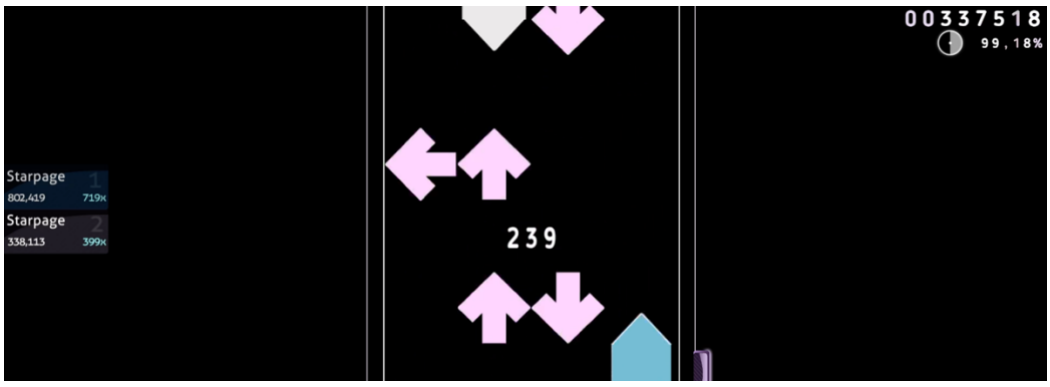
Gameplay is set on a plain black backdrop and is centred in the middle of the screen with notes approaching from the top in one of four lanes. Each note represented by an arrow and aligns with the arrow symbol below it.

The combo counter and accuracy text are updated after every note is finished and are displayed small in the centre of the screen. Miss show the timing for a note to be hit has elapsed.

White solid arrows at the bottom are indicators that tell the player when they should be pressing, and only interact with the game to light up when a key I pressed.



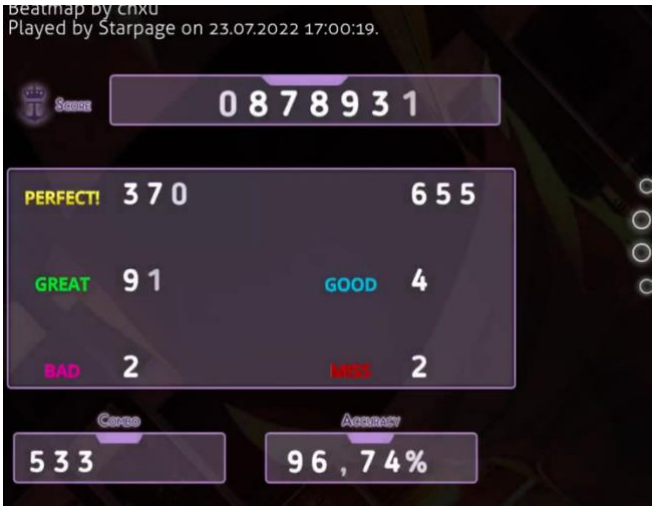
These features let the user know how they are doing and are vital parts of the UI, as they make the game user friendly. Some features such as the accuracy meter at the bottom of the screen (that denotes how early or late each note was hit.) I find unnecessary, since it appeals to higher levels player who use those statistics.



When Full screened “Osu!mania”, has high scores displayed to the left and score and overall accuracy displayed to the right. Whilst this minimal style would not suit my project, the graphics provide enough statistics to the users. I find the style in “Muse Dash” more suited to my concept.

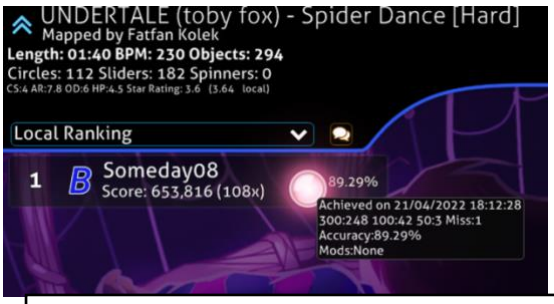
Song Complete screen

The end screen for each stage is formatted the same, like in “Muse Dash”, except with extra statistics about which accuracy scores for each individual note hit, including: Miss, Bad, Good, Great, and Perfect. This decomposes exactly how the user did and is appropriate for advanced users.

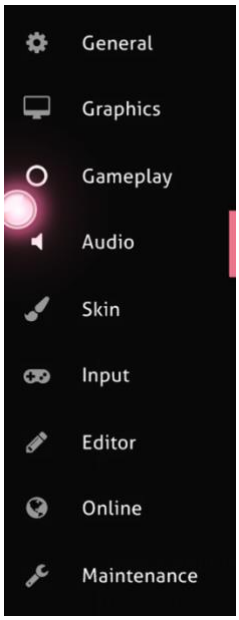


Besides these statistics the screen complete screen has an evaluation of the user’s rank, and the screen has basic graphic boxes to tell the user where the key features are. This part of the game is similar to the stage complete screen in “Muse Dash”, which shows why this screen is vital to include, as it concatenates all the important statistics into one message of how the player performed. Adding a colourful background will help all players feel encouraged and engaged.

Song Selection Screen



Near the top of the screen selection screen, the high scores are formatted in a drop down box, listed as ranking out of the total scores achieved. The leader board updates as the user selects new songs in the song selection screen. This implementation of high scores reduces the number of screens needed and therefore the user does not have to navigate through screens to find them. A greater focus is also placed on high scores, which re-iterates the need for a score system in game to track the player's progress



In contrast to "Muse Dash the settings menu has several pages, one dedicated to each variable of the game that the use would be able to adjust.

Volume and adjustment settings are controlled with a slider, limited from 0 to 10, where 0 is muted and 10 is full volume. The volume has also been split into master, music and effects which allow for a higher range of user customisation. When the slider is moved a value comes up above it to let the user know the number. Shortcuts can also be set to adjust volumes with keys however I find the settings menu is better solution – additional keys can be hit accidently.

Overall, "Osu!mania" has many features that I will include in my game. Most of which can be developed in a similar way to "Muse Dash" as the two share features. Important aspects of game design which have been re-iterated through my look at "Osu!Mania" are:

- A live updating score counter is necessary for a successful rhythm game – the gameplay focuses on building score up to gain a sense of progress, rather than unlocking more stages or any other kind of progression. This makes the game more re-playable, and improvement driven.
- Combo – combo in osu! Has a greater emphasis than score.
- Several slider options for volume – finer control than just one slider.
- Timing judgements and combo placed centrally – of the game HUD these two are possibly most important.
- Hit effects - target the hit input area rather the notes to make the users inputs feel more interactive
- Stage complete screen – condensing data into only the important information for the user to see. Score accuracy, and combo. Have repeatedly shown to be the most important statistics, whilst a rank is seen as less necessary.
- Highscores saved in a leader board – leader boards are a common feature for multiplayer games, and let a user save more than one of their scores. The scale of the two solutions I chose to research comes in here and for a game with a much smaller player base, a leader board would not be as necessary.

Research: Interviews

To gain an understanding of what the identified stakeholders would like to see from my game, I interviewed 2 people between the ages of 10-18. Choosing one person lower in the age range and one person higher in the age range gave me a more complete view of which basic features I should include. Furthermore, I chose people familiar with rhythm games to give more specific feedback.

Both have agreed to playtest the game and give feedback after each prototype iteration. As each of the interviewees fall into different age groups, I can best target my game to the full age range.

Participants

- Magda, age 12, female
- Farhan, age 17, male

I formed my questions beforehand based on what features I observed in my game research and from my problem overview, first focusing on how the rhythm game would be played, then the music choices, then the difficultly and then asking about other features.

Questions

Have you played any rhythm games before?
Farhan: Yes, I've played "Piano Tiles", "Muse Dash", "Dance Dance Revolution", and "Beat star".
Magda: Yes, "project SEKAI", "Muse dash", and "Drum Along festival".

What input devices do you play on?
Farhan: Mainly a keyboard (on a windows PC) but also a touchscreen (on an android phone), a dance dance revolution mat, and a controller.
Magda: Keyboard (on a windows PC) and touchscreen (on an iOS device).

How many fingers to you typically use to play?
Farhan: 2 thumbs for piano tiles (on mobile), or middle and index for horizontal games (on PC). For me, I just use whichever notes the game wants me to use, I'm not too bothered about which fingers I use specifically.
Magda: 4 fingers for muse dash, and 2 for project SEKAI. I prefer using 2 when possible.

Would you prefer a horizontal or vertical layout for my rhythm game?
Farhan: Horizontal for sure.
Magda: Horizontal, it's more fun to see the notes across the whole screen.

How much does music influence weather you want to play a game?
Farhan: extremely, I am not interested if the songs are bad. I often search out games with immersive .soundtracks.
Magda: Very much, if a game has good songs, I will keep coming back to it.

How important is song variety to you?
Farhan: Yeah, I like some song variety.
Magda: Not a lot, what genre the songs are matters more, I prefer playing just one genre that I enjoy a lot than many different genres.

What genres of music would you like to see?
Farhan: I'd like to see hyper pop, EDM and pop as those are fun genres to play in rhythm games. I think a rap song would be cool too as you don't usually see those in rhythm games.
Magda: Pop in rhythm games is the most fun to play for me, otherwise I don't mind too much. Lyric less songs can be good but can get boring.

How many songs do you think should be in the game?
Farhan: around 5 is more than enough for me.
Magda: More than 1, Ideally 3 songs to choose from. Repeating songs over and over again to reach a personal best is an exciting part of rhythm games so I don't mind there only being a few songs.

Would you prefer fewer longer songs or more short sections of songs?
Farhan: Fewer longer songs
Magda: Fewer longer songs for sure

Do you have an ideal song length?
Farhan: 2 - 3 minutes but shorter works as well.

Magda: Nothing overly short or cut off abruptly as these ruins the songs for the player. They should be under 5 minutes too.

Do you think there should be a tutorial song in the game?
Farhan: Yeah, I'd like a short tutorial to introduce how to play, I can see new players being discouraged without teaching the very basics first.
Magda: Yes definitely.

How difficult do you think the game should be?
Farhan: Quite difficult but I think there should be easier difficulties to choose from, I'd rather be challenged than bored with easy songs.
Magda: Expert level or above, difficult enough to grind. The levels should be do-able but only just.

Should you be able to fail a song in the game?
Farhan: yes, although not too easily.
Magda: yes.

How important are difficulty mods to you (changing the difficulty of the songs in game)?
Farhan: I think having a difficultly increase or reduction feature can make a song more fun to replay.
Magda: I tend to stick to the normal difficulty, but beginners might want an option to lower the difficulty. When playing by yourself, difficulty is fun when its overly high, so having a set difficulty wouldn't be too bad.

How interested are you in seeing how well you did after a song?
Farhan: I like seeing how bad or well I've done. If possible then having accuracy, combo and a rank at the end of each song.
Magda: Super interested. For a challenging game, seeing where you failed is motivation to keep playing. Even if the song is hard, wanting to improve your own scores is part of the fun. I'd like to see my score and rank and maybe accuracy as well.

Would you be interested in having customisable settings (e.g. note approach speed)?
Farhan: I don't particularly mind, but it would be a nice feature to have and make the game feel more professional and finer tuned.
Magda: Yes, very much, I play with low note speed settings compared to others so having that option is important to me. Without that I find the game unfair.

Would you like to be able to pause the game during a song?
Farhan: No, it's not necessary.
Magda: It would be a nice addition but its ok to not have it, especially if the songs aren't too long. I don't usually use that feature.

Any other features you would like to see?
Farhan: Maybe a feature to reduce difficulty for beginner players.
Magda: If there is time then maybe a random chance to unlock items or characters after playing the game. Also a setting to play the song automatically to impress your friends. And a place to see your high scores.

Interview feedback

From my interviews I have identified that my target audience would like a horizontal side scrolling game on the more difficult side, with song choice being very important to the overall appeal of the game. Score, rank, accuracy, and health are all features that were wanted, along with longer songs, with some variety.

Whilst interview information is invaluable, the sample size I have obtained is still small and feasibility of their answers must be taken into consideration. For this reason, additional features such as unlockable items or variable note approach speed settings may not be possible, as they would be too time consuming and too specific for most audiences.

It was useful to note that features such as a pause menu and changing key binds were not wanted, and so including these would be not improve the game.

To keep within my original stakeholder group, I will veer my game towards being more difficult and then introduce options to reduce the difficulty such as slowing the notes down, as discussed in the interview. Additionally, having a range of difficulty across the different songs would also solve this issue.

As for song choice, I will be sticking to the genre of video game music, as this is something that can be enjoyed by many audiences, however within that genre, I can choose pop songs, EDM songs, rap songs and songs with a focus on lyrics.

Features of the proposed solution

These are features that are essential to the game, along with the reasons I chose to implement them. Any evidence collected in previous research is mentioned in reference.

Feature	Description	Justification	Reference
Start screen	On opening the game a simple screen prompting the user to enter, exit or go into settings of the game as buttons. Should have the title of the game displayed in a large font, and simple graphics for the background	Allows for entering the game, exiting, and entering settings as well as providing a starting point for the script.	From Game 1 analysis
Song selection screen	A menu where the player can see all the songs, should have basic information about the songs including difficultly and should have a clear link to the leader board with a preview of the high score as the user hovers over the song.	Allows for navigation between stages and presents the user with all the options in one place. A younger audience will be overwhelmed if this screen is too crowded so having minimal information about the stages creates useful abstraction.	From Game 1 analysis
Tutorial stage	The first stage the player should enter. Unlocks other stages of the game and pops up short instructions during the song on how to play	A beginner target audience might not understand the game mechanics immediately.	From Interview
Settings menu	A menu for changing music, effects, and overall volumes. Settings should carry over once the game is closed and re-opened.	Having an option to lower volume provides accessibility to people with different sound output devices. As the game is a rhythm game the balance of sounds is vital to the gameplay.	From game 1 analysis
Game screen	Game screen with a background, hit area, approach lanes, and a HUD (heads-up display). This is where all the stages will be played, and the game screen needs to be able to support all the different songs – including displaying oncoming notes	The game screen should have a colourful animated background, appealing to my target audience and the game will play horizontally from right to left as identified previously, in order to create an engaging game.	From Interview, game 1 analysis and game 2 analysis.

	and audio and visual effects from various sources.		
Song complete screen	After a song has been finished and if the user has not failed, the screen should show all the statistics, including score, combo, accuracy, and rank. The user should then be prompted to return to the song selection screen.	Giving the player a message of how they performed is on a colourful background will help all players feel encouraged.	From game 1 analysis and interview.
Animated background	Background playing in conjunction with the game, either moving or repeating a simple effect in time with the beat of the music.	Background should be engaging for newer audiences but should not distract from the gameplay.	From game 1 analysis.
Hit area	A rectangle outlining where the notes should be hit, positioned on the left of the screen. Once notes go too far out of this area they are destroyed, and the player gets a miss. The hit area will be composed of several sub sections expanding outwards, allowing for timing leniency to get a late or early hit, depending on the difficulty set.	This gives the player a visual point of reference for where to hit the notes and makes the game intuitive and fun so the player does not have to guess when to it the note.	From game 1 and 2 analysis.
Object lanes	Two areas where the notes will come from, both horizontal and moving from left to right. One lane will be at the top of the screen and one at the bottom. The player will effectively move between them as they hit the notes.	I chose to include two lanes because it simplifies the inputs to just two keys that need to be pressed whilst also having more variety than a single row of notes.	From game 1 analysis
Hit objects	Circular objects to be hit by the player. Spawned from the same position in one of two lanes and approaching the hit area at a constant speed. Player inputs a press to destroy the notes and get score and combo from this, whilst also being judged on their timing. Hit objects are abstracted notes of the song, generated from a dictionary.	Hit objects are the core mechanic of the game, and so objects should be big enough to be clearly seen and should also have some sort of appealing outline to stand out from the background. Making these move at constant speed allows for synchronisation with the game time and music.	From game 1 and 2 analysis.
Hit input	The player will have two keys that they can press to register a hit. These will correspond to the top and bottom lanes. The game responds to a key press with an hit effect and hit sound to tell the player a key has been pressed.	Input keys should be in intuitive places to make them easy and comfortable to input. From the information I gathered in the interview, they should be where the middle and index rest.	From the interview.
Hit sounds	Hit sounds will occur whenever a hit input is detected and will tell the player they have inputted it successfully. In form of a drum sound effect.	These tell the player they have inputted a press successfully and gives a sense of immersion because it allows for freedom during empty parts of the song and creates a scene they are a part of. Additionally, these can help with developing motor	From game 1 and 2 analysis.

		functions and rhythmic senses in the player by providing a link to the real life skill of playing an instrument, making the game more functional.	
Hit effects	Hit effects will occur whenever a hit input is detected and will tell the player they have inputted it successfully. To do this, the respective hit area will glow for the duration of the key press.	Like hit sounds, hit effects are important to let the player know that their inputs are being registered correctly.	From game 1 and 2 analysis.
Score display	Part of the HUD – a constantly incrementing counter with the player's score, which is gained through higher combo and hitting notes, with a reduction in score if the hit is miss timed.	Score will be displayed small in the corner as whilst it is important information for the player it is not important to the game play. Scores are needed for higher skilled players who want to improve their place on the leader board.	From game 1 and 2 analysis.
Combo display	Part of the HUD – A counter incrementing after each successful note hit, resetting to zero if a note is missed. Used for calculating combo.	Like the score display, combo lets the player know how long ago the last note they missed was and how well they are doing.	From game 1 and 2 analysis.
Health display	Part of the HUD – a bar that depletes if the user misses too many notes. Each miss takes a percentage off the health bar, meaning too many misses causes the player to hit zero health and fail the map.	Having a mechanic to stop the player from playing levels above their skill level can encourage them to try easier songs rather than ending up frustrated on song and can also give a sense of achievement when they pass a song.	From game 1 analysis.
Accuracy display	Part of the HUD – Accuracy text shows up under the score as a constantly updating percentage (out of 100%). 100% accuracy being fully perfect timing and 0% being every note was missed.	Accuracy statistics were requested from the interviewees and are an integral part of rhythm games, letting the player know how close to the actual timing of the song they are in a single statistic.	From game 1 and 2 analysis and the interview.
Timing indicators	After each note, text displays depending on how early or late the user's press was. Either a miss, good or perfect is displayed.	Gives feedback and guidance to the user's timings, as well as telling them how to adjust on incoming notes. Timing indicators should be accurate and not too strict to be used effectively.	From game 1 analysis.
Song failed display	After depleting the health bar the user fails the stage and is given a clear, failed message. All game logics stops and fades away. From here they either get the option to retry or go back to the menu. Text should appear in the centre of the screen for most visibility and readability.	Gives feedback to the user and prevents them from playing stage far above their capabilities.	From the interview and game 1 analysis.

Limitations

Due to limitations, there are some features that whilst improving the overall quality of the game, might not be possible to implement. Because time to code the project is limited and I will only have access to free open-source libraries, these features will only be considered if the most important parts are complete.

The game's scope will also be restricted by how many songs I include. As part of the essential features, I will have 3 separate and uniquely mapped stages which are unlocked by the tutorial. This will limit the size of the game and make the project more manageable. The features bellow are simply extras that I would like to add but because of constraints they are harder to justify.

Accessibility features should be considered for users that can't play with a mouse or certain areas of the keyboard, especially since the game will be targeting those training motor functions and timing. The game's priority will be on creating an enjoyable experience over a scientific training tool, and the game will not claim to make any improvements to rhythmic ability.

Feature	**Description**	**Justification**	**Reference**
Leader board	Holds information about a user's score, highest combo, rank, and accuracy. Displays separately for each stage in descending order of scores. A more time friendly approach would be having a single high score for the player to see. The leader board should carry over when the game is closed and re-opened.	A popular rhythm game feature that allows players to see their best scores and provides a target to aim towards. The leader board should be a part of the song selection screen for easy viewing. Players can aim to beat their previous high scores. Whilst this would add a competitive edge to my game, it's not vital to the function.	From the interview and game 1 analysis.
Long beat objects	In game objects that act similar to hit objects but lasting for longer. The player must hold the key down for the duration of the note to get the full score.	This feature comes from game 1 analysis one and add variety and interest to the stages, and further develops the players rhythmic senses. This is not necessary to the game.	From game 1 analysis.
Hidden beat objects	Just like regular hit objects, expect they fade to transparency after appearing, letting the player know they are there but not exactly where they are.	Adding these types of notes would add a lot of variety to maps, however it is by no means a necessary addition. Hidden beat objects force the player to rely on the beat of the song rather than a visual representation of the song's beats.	From game 1 analysis
Difficulty settings	Changes health and hit area to increase or reduce difficulty. By making the timings windows smaller, the difficulty of the game can be artificially changed. More health would make stages easier to pass	In the interview, some people specified that more challenging stages are important to them in a rhythm game. To remedy this, a difficulty reduction would let newer players enjoy the full game without being bound by failing a stage or hitting difficult note timings.	From interview and game 2 analysis.
Note approach speed setting	In the settings menu, players could the speed at which notes approach the hit area. How much time you have to hit the note would be compensated for if this setting was changed so that it only affects how fast you want the notes to appear.	This was requested in the interview and is seen in game 2 and whilst would help more experienced players feel more comfortable with the game, newer players are unlikely to need that high of a level of freedom.	From the interview and game 2 analysis

Rank display	Part of the song complete screen – assigns the player a rank based on score after a song is complete. Ranks going from C to B to A to S to S+	This would sum up the statistics in a simple and understandable way, however, would be another potentially unnecessary metric to add to the game.	From game 1 analysis and interview.

Hardware and Software Requirements

Development

I will be using Unity 2021.3, an extensive development engine popularly used for 2D and 3D games, and mac OS as my operating system. All my development specifications are based on those two prerequisites.

Minimum hardware requirements for running Unity have come from the Unity documentation [7]. Hardware should be capable of running the unity editor and the unity player.

- CPU with X64 architecture or SSE2 instruction set support
- Intel or AMD GPU capable of using Metal API
- 12GB of secondary storage
- keyboard and mouse input
- monitor and speaker output

Minimum software requirements have come from the Unity documentation [7] and the architecture needed to run unity has come from the scripting Unity documentation [8]. These are necessary for coding in c#.

- .Net Platform
- Mono (I used 6.12.0 for mac OS)
- IL2CPP
- Mac OS 10.13+
- XCode
- Unity
- Apple supported drivers

Running the game

Requirements to run the game will be lower than the requirements to code it, estimated from games that will be around the same size as mine.

Estimated minimum hardware requirements based on hardware requirements given by geometry dash [9], which is not a graphics-intensive game.

- 1GB of RAM
- Integrated graphics GPU
- single core 2 Gigahertz CPU
- 100MB of storage
- Keyboard, mouse for input
- Monitor and speakers for output

Minimum software requirements. Unity has templates to compile the game to MacOS, Linux and windows and so the operating system is not a limiting factor in playing the game.

- Windows 7, MacOS X 10.13 or Linux

Success Criteria

For the problem solution I have identified, these are the features that will appeal best to my target audience and will make a fun, engaging and stimulating rhythm game.

My success criteria will be split into priority levels based on which features should be implemented first.

- Top – Essential features for the game to function at a basic level. The game cannot work without these.
- High – Necessary features that complete the game.
- Medium – Extra features that add functionality and depth.
- Low – Unnecessary features, mostly for aesthetic appeal or for added interest.

N.O	Feature	Description	Reference + Justification	Priority
1	Start screen	On opening the game, a simple screen prompting the user to enter, exit or go into settings of the game as buttons. Should have the title of the game displayed in a large font, and simple graphics for the background	Game 1 analysis showed this was a needed starting point for the player.	top
2	Song selection screen	A menu where the player can see all the songs, should have basic information about the songs including difficultly and name. Should have a clear link to the leader board with a preview of the high score as the user hovers over the song.	Game 1 analysis showed that songs should be navigated to with a linear list and song details should be abstracted away.	top
3	Tutorial stage	The first stage the player should enter. Unlocks other stages of the game and pops up short instructions during the song on how to play.	The Interview covered that new players should have a brief introduction to the game.	high
4	High score	The player's best score for a stage. The high score should carry over when the game is closed and re-opened.	The interview showed that a competitive feature is wanted.	medium
5	Leader board	Holds information about a user's score, highest combo, rank, and accuracy for a stage. Displays separately for each stage in descending order of scores. The leader board should carry over when the game is closed and re-opened.	An added level of complexion to high scores; the games 1 and 2 analysis showed that a separate display for player scores was useful but cannot be justified to be important for a small game.	low
6	Settings menu	A menu for changing sound settings and difficulty settings. Settings should carry over once the game is closed and re-opened.	Games 1 and 2 analyses showed a menu for changing settings was needed	high
7	Sound settings	Sliders to control music volume, effects volume, and overall volumes. Sliders set volume percentages from 0 - 100%	Games 1 and 2 analyses showed that all three volumes settings were necessary for various output devices.	high

8	Note approach speed setting	In the settings menu, players could the speed at which notes approach the hit area. How much time you have to hit the note would be compensated for if this setting was changed so that it only affects how fast you want the notes to appear.	The interview and game 2 analysis showed that whilst this would make the game more accessible, but due to limitations may not be needed.	low
9	Difficulty settings	Changes health and hit area to increase or reduce difficulty. More health would make stages easier to pass and larger timing windows would make notes easier to hit. The reverse would apply for a difficulty increase.	The interview and game 2 analysis showed that this feature is not necessary but would help very new and very skilled players	low
10	Game screen	Game screen with a background, hit area, approach lanes, and a HUD (heads-up display). This is where all the stages will be played, and the game screen needs to be able to support all the different songs – including displaying oncoming notes and audio and visual effects from various sources.	Games 1 and 2 analyses showed that this was a necessary environment for the game.	top
11	Song complete screen	After a song has been finished and if the user has not failed, the screen should show all the statistics, including score, combo, accuracy, and rank. The user should then be prompted to return to the song selection screen.	Games 1 and 2 analyses showed that this was the best way to conclude a song and the interview confirmed having statistics was useful.	high
12	Multiple stage designs (levels)	In the stage selection screen, the user should be able to choose from at least 3 stages. These stages should have different note arrangements and should vary in. song choice difficulty. The length should be at least 2 minutes for an effective and enjoyable stage.	Stakeholder feedback showed that having multiple stages to choose from make the game more replay able.	high
13	Hit area	A rectangle outlining where the notes should be hit, positioned on the left of the screen. Once notes go too far out of this area they are destroyed, and the player gets a miss. The hit area will be composed of several sub sections expanding outwards, allowing for timing leniency to get a late or early hit, depending on the difficulty set.	Games 1 and 2 analyses showed that this make the game feel more natural to play.	top
14	2 object lanes	Two areas where the notes will come from, both horizontal and moving from left to right. One lane will be at the top of the screen and one at the bottom. The player will effectively move between them as they hit the notes.	Game 1 analysis and the interviewed showed that using 2 input keys and visually dividing the screen created an effective way to hit notes.	top
15	Hit objects	Circular objects to be hit by the player. Spawned from the same position in one of two lanes and approaching the hit area at a	Hit objects are essential parts of the game screen, from games 1	top

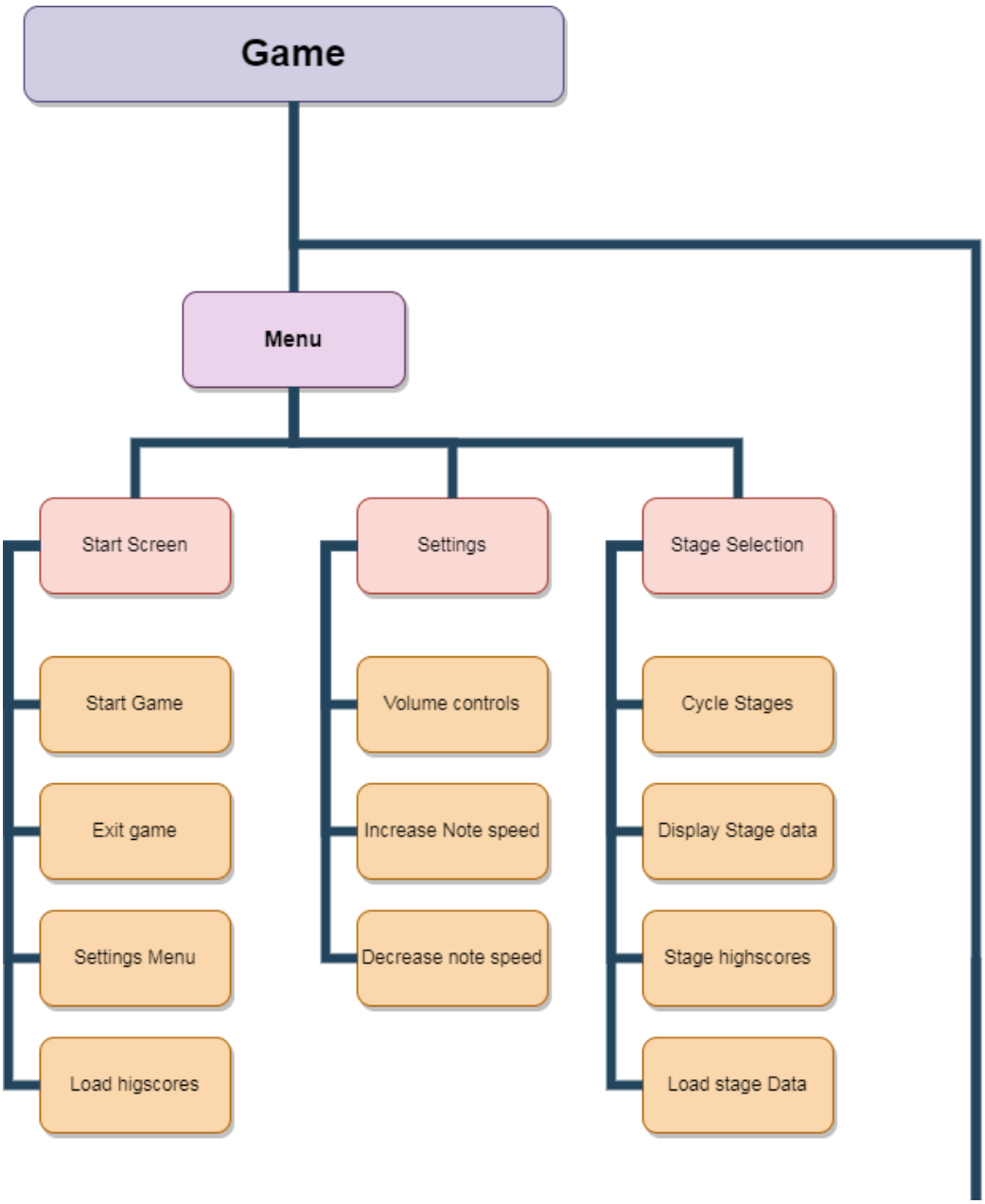
		constant speed. Player inputs a press to destroy the notes and get score and combo from this, whilst also being judged on their timing. Hit objects are abstracted notes of the song, generated from a dictionary.	and 2 analyses.	
16	Long beat objects	In game objects that act similar to hit objects but lasting for longer. The player must hold the key down for the duration of the note to get the full score.	Game 1 analysis showed that this is a dynamic edition that can be reused in various stages but is complex and not needed.	low
17	Hit input	The player will have two keys that they can press to register a hit. These will correspond to the top and bottom lanes and are used to hit approaching objects.	The interview and games 1 analysis showed that a 2 key input method would be most suited to my project and let the user input hits freely.	top
18	Hit sounds	Hit sounds will occur whenever a hit input is detected and will tell the player they have inputted it successfully.	games 1 analysis showed the game responding to a key press helped the player's co-ordination.	high
19	Hit effects	Hit effects will occur whenever a hit input is detected and will tell the player they have inputted it successfully. To do this, the respective hit area will glow for the duration of the key press.	games 1 and 2 analyses showed the game responding to a key press was a useful feature and improved graphics	medium
20	Animated background	Background playing in conjunction with the game, either moving or repeating a simple effect in time with the beat of the music.	games 1 and 2 analyses showed that a moving background was not necessary, and a simple graphic would not subtract from the game.	low
21	Score	A live counter with the player's points, which are gained through higher combo and hitting notes, with a reduction in score if the hit is miss timed.	The interview along with both game analyses showed that seeing how well the player has performed is necessary for a rhythm game.	top
22	Combo	A counter incrementing after each successful note hit, resetting to zero if a note is missed. Used for calculating combo.	The interview along with both game analyses showed that this feature is always needed in a rhythm game.	top
23	Health bar	A bar that depletes if the user misses too many notes. Each miss takes a percentage off the health bar, meaning too many misses causes the player to hit zero health and fail the map.	Game 1 analysis showed that a health bar was an effective way to measure if a song was too challenging for a player.	medium

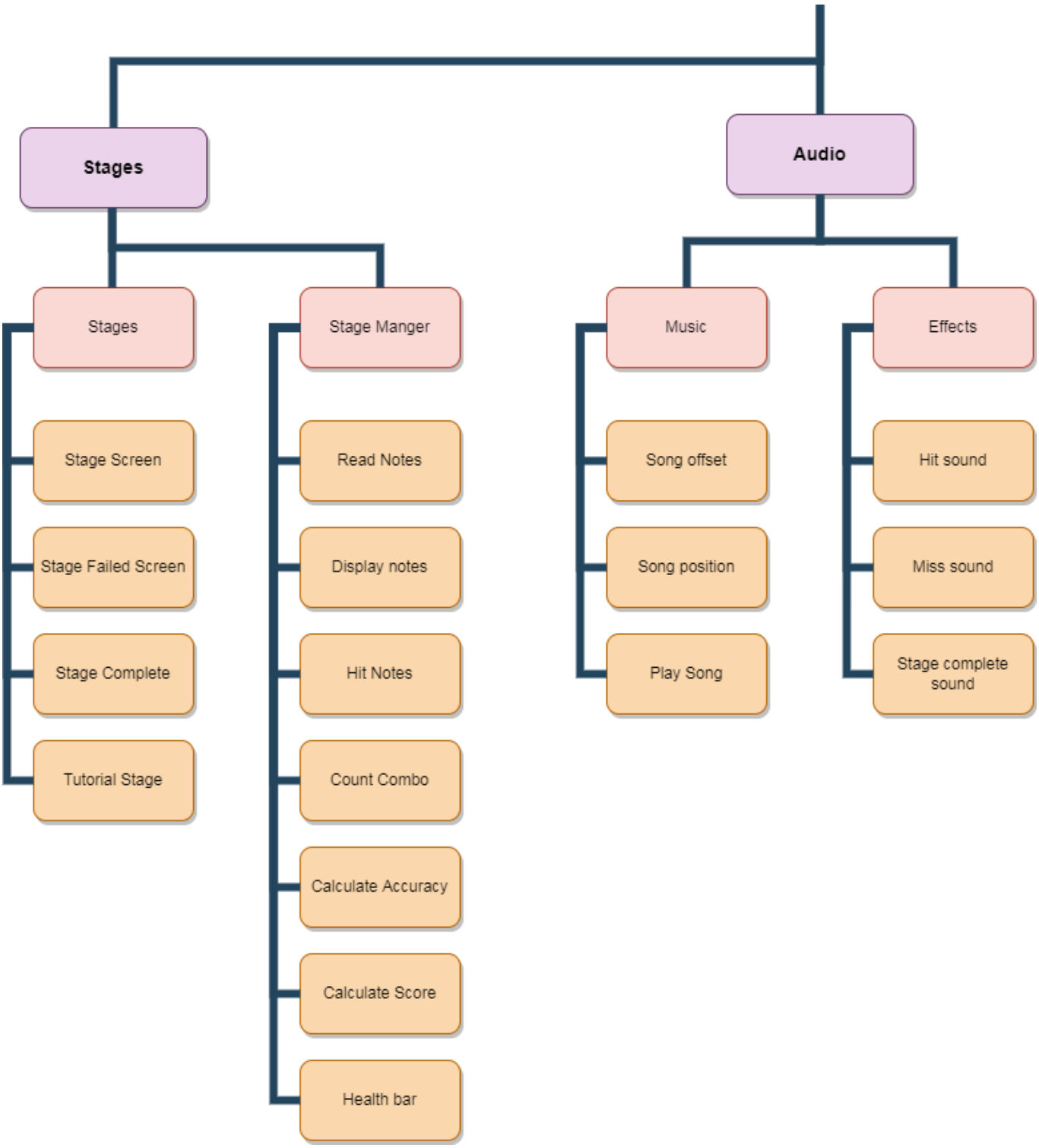
24	Song failed display	After depleting the health bar, the user fails the stage and is given a clear, failed message. All game logics stops and fades away. From here they either get the option to retry or go back to the menu. Text should appear in the centre of the screen for most readability.	The interview and game 1 analysis showed that failing a song was a important feature that helps keep the user focused.	medium
25	Accuracy	Accuracy text shows up under the score as a constantly updating percentage (out of 100%). 100% accuracy being fully perfect timing and 0% being every note was missed.	games 1 and 2 analyses, along with the interview showed that letting the user know their statistics is important	medium
26	Rank	Part of the song complete screen – assigns the player a rank based on score after a song is complete. Ranks going from C to B to A to S to S+	games 1 and 2 analyses showed that whilst not necessary, this adds a lot of interest to the game.	low
27	Timing indicators	After each note, text displays depending on how early or late the user's press was. Either a miss, good or perfect is displayed.	games 1 and 2 analyses showed that this was a key feature for the game to feel responsive and for the player to know how well they are playing.	high

Design

Systems Overview

For my Rhythm Game, I need to build a robust system with all the intended functionality from my success criteria. For this, I will have several systems working together, importantly the Menu, the Stages and the Audio systems.





The decomposition top-down diagram above describes the most important systems and their key functionality, for a fully moving and complete game. The game cannot be implemented in this way however, because certain separated areas need to work together, such as the hit sound and hit notes. Functionality needs to abstracted into objects that can interact with each other.

Solution Structure

My Solution will be composed of several public classes that will interact with each other in an object-oriented system. Compared to procedural paradigms this method allows me to closely model the relationship between objects and their data and allows for lanes and note classes to be reused to make several objects.

Object Oriented Programming encourages code integrity in this way, to code a more concise, maintainable, and modular solution, all of these features are necessary for expanding on the game and adding additional features and Stages without re-writing any functions.

The encapsulation of methods and attributes that OOP allows through private variables is useful for introducing data security and validation. At several stages private methods and attributes make it harder to accidently change data and introduce bugs to the game.

Because my game can be solved by computational methods, an object oriented approach would work, and because of its advantages, I have decided to create my game in this way, and have classes that model the data needed to run the game.

I plan to split the system in the following classes:

- Game
- Stage Manager
- Conductor
- Note
- Lane
- Score Manager

Classes

These are the classes for my solution. Each class will have its own role in interacting with other classes and its main roles are explained below.

All classes inherit from Unity's *MonoBehaviour* which I have not included here.

\+ Stands for a public attribute/method
– Stands for a private attribute/method

- **Game –** responsible for initializing the main game loop and reading and writing user settings, changing user settings, and displaying the menu and buttons for navigating the:
    - start screen
    - settings screen

Game
+ userVolume : float + userNoteSpeed : float + userFilePath : string
+ LoadStageSelect () : void + LoadSettingsMenu () : void + ExitGame () : void + StartGame () : void + GetUserSettings() : void

- **StageManager –** responsible for reading stage data from JSON files. Responsible for displaying the menu and buttons for navigating the:
    - stage selection screen
    - stage complete screen
    - stage failed screen

StageManager
+ StageID : int - StageName : string - StageArtist : string + StageHighscore : int
+ LoadStageComplete () : void + LoadStageFailed () : void + LoadStageMenu () : void + SelectStage (int stageID) : void + ReadStageData (int stageID) : void

- **Conductor –** responsible for playing each song, retrieving stage data from MIDI files, and keeping track of song and note timings.

Conductor
+ filePath : string - noteTravelTime : float - noteSpawnPositionX : float - noteTapPosition : float - noteDespawnPosition : float - songDelay : float + currentSongTime : double + lanes[ ] : Array<Object>
+ ReadMidi (filePath) : MidiType + GetDataFromMidi (MidiType) : array + StartStage (songDelay) : void + GetAudioSourceTime (AudioSourceType) : double

- **Note –** responsible for showing notes on screen as they appear and handling sprites for notes.

Note
- timeInstantiated : double - timeSinceInstantiated : double - relativePosition : float
+ SetTimeInstantiated (timeInstantiated) : double + RenderSprite () : void + UpdatePosition () : void + SelfDestruct() : void

- **Lane –** responsible for spawning notes, keeping track of both lanes, and handling all in stage user inputs. {The lane syncs notes to the song in real time}

Lane
- noteRestriction : string + input : string + notes[ ] : List<Objects> + timeStamps[ ] : List<double> - inputIndex : int - spawnIndex : int
+ SetTimeStamps (Note[] array) : array + SpawnNotes (var note) : void + CheckHit (double audioTime, marginOfError, timeStamp) : void + CheckMiss (double audioTime, marginOfError, timeStamp) : void - RegisterMiss () : void - RegisterMiss() : void

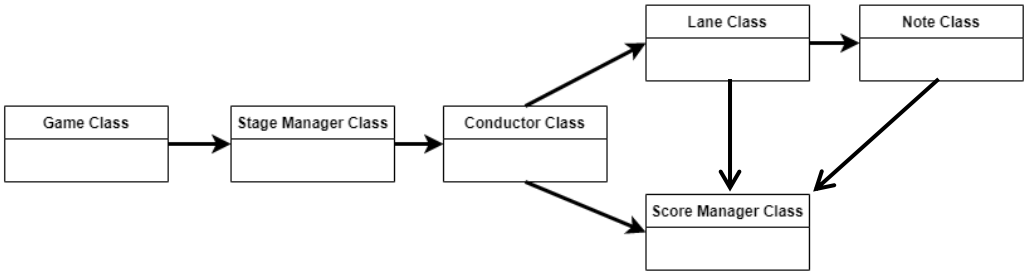
- **Score Manager –** responsible for responding to note hits and misses, including sound effects, calculating scores and updating the Stage UI. Responsible for reading and writing high score and stage data to and from JSON files.

ScoreManager
+ combo : int + score : int + rank : string + accuracy : float + health : int - comboText : textObject - scoreText : textObject

+ UpdateHit (string rating) : void
+ UpdateMiss () : void
+ UpdateUI () : void
+ CalculateScore () : int
+ CalculateRank () : string
+ CalculateAccuracy () :float
+ UpdateHealth () : int

How the structure forms a solution

Whilst no classes inherit from each other, classes will call each other to form an ordered solution. This means that the solution will act procedurally but still have modular functionality. Computational methods that are used in the final solution are encapsulated in this structural approach. The order of the solution will be defined by these hierarchical links between the classes:



Algorithms

My solution will include algorithms from the classes above. For this section I will encapsulate these into individual algorithms, and group their functionality, not necessarily corresponding with the solution structure, but closer to the system overview.

Game Class

The game class is responsible for initializing the main game loop and reading and writing user settings, responsible for displaying the menu and buttons for navigating the screens.

Start function as pseudocode

The start function should occur as soon as the game is launched and should display graphics for the start menu whilst loading all prefabs that will be used in the game to minimize user waiting time. Responsible for displaying the UI, and calling get

```
GetUserSettings (filePath)
```

And when corresponding buttons are clicked calling :

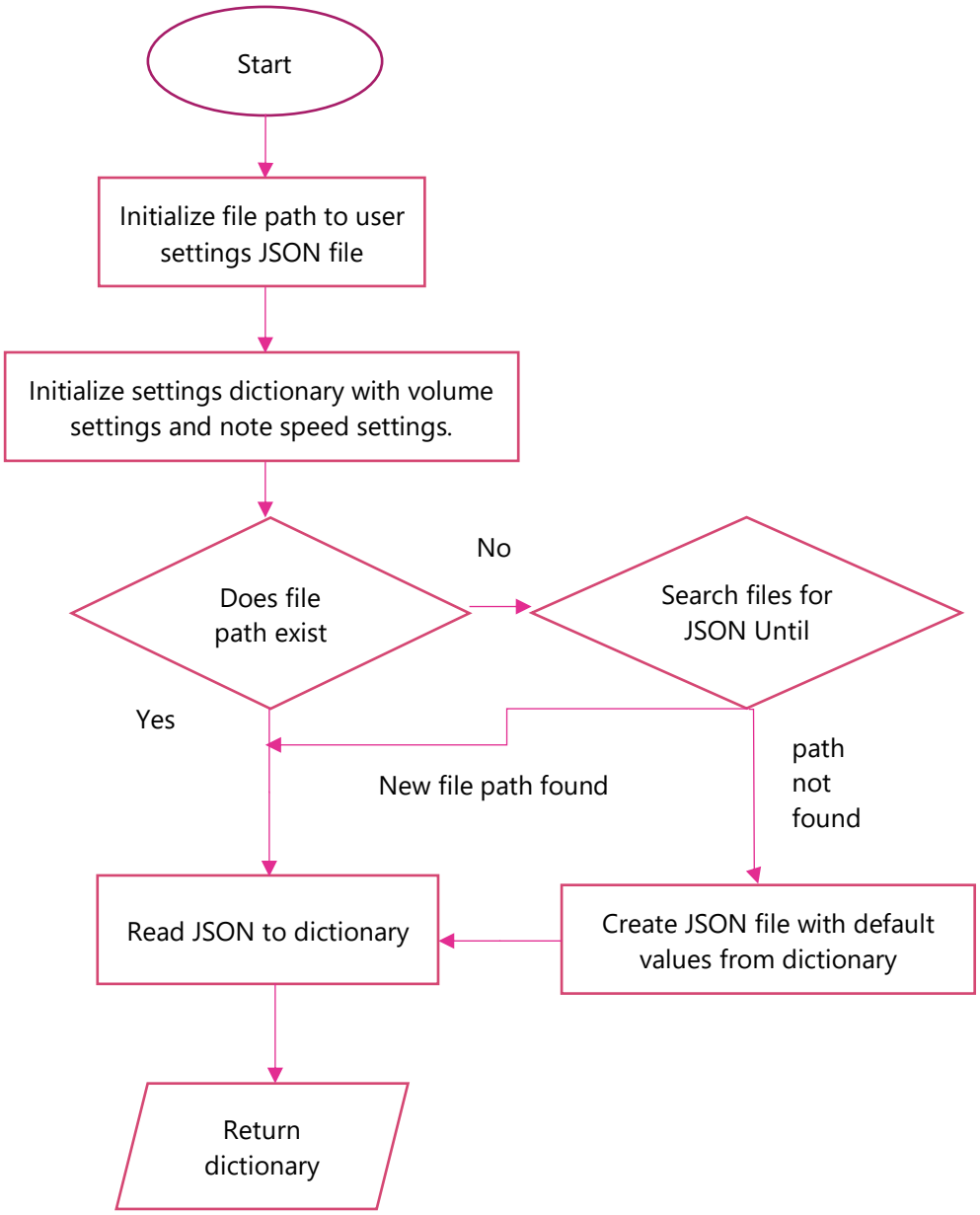
```
LoadStageSelect ()
LoadSettingsMenu ()
ExitGame ()
```

The scene should change to the corresponding screen. They should include scripts for button functionality and style.

```
procedure Start()

    GetUserSettings (filePath)

    If button1.isPressed then
        LoadStageSelect ()

    Else if button2.isPressed then
        LoadSettingsMenu ()

    Else if button2.isPressed then
        ExitGame ()
    endif

endprocedure
```

Get user settings algorithm as a Flowchart

User settings should also be read from a JSON file and parsed, start game will initialize the main game loop. The file path needs to be validated to be read correctly, and if not, we can use the system to find the JSON file or write an empty one. This algorithm ensures that a settings file always exists, even if it doesn't, hence why the algorithm always return a value.



Stage Manager Class

On being called by the start game screen, the stage manager class has to load the stage menu, which contains all the stages and UI for the user to scroll through them and select one, this is the main hub for the levels (stages).

Load stage menu algorithm as pseudocode

This algorithm gets all stage data for each song and displays it in a menu. Stages must be a list in order to be set as however many stages there are. The algorithm uses a dictionary structure, as the read stage data function inherits from the get user settings function and also returns a dictionary, this time containing name, length of the song and icon used.

The Stages list is then returned and the stage data is used to display the stage. A display stage function will show the designs of the stage icons on the screen and let the user click on them. They will act as buttons to be pressed similarly to the game class (start menu).

```
Function LoadStageMenu (int stageCount)

	DictionaryType stages[]
	Dictionary = {int stageID : string filepath, string name, float length,
UnityPrefabType icon}

	For num = 0 to stageCount

		Dictionary stage = ReadStageData(num)
		List stages[num] = stage

	return Stages

EndFunction
```

Select Stage Algorithm as pseudocode

The stage manager class then waits for the user to input an enter to proceed to the selected stage. It does this with a simple if statement to check for both a keypress and if the stage is currently selected by the user.

This algorithm then instances a new conductor class which instances all other classes and enters the main game scene, and begins to play the stage. The conductor is instanced with the filepath of the stage MIDI so that is can be read as described in the other algorithms.

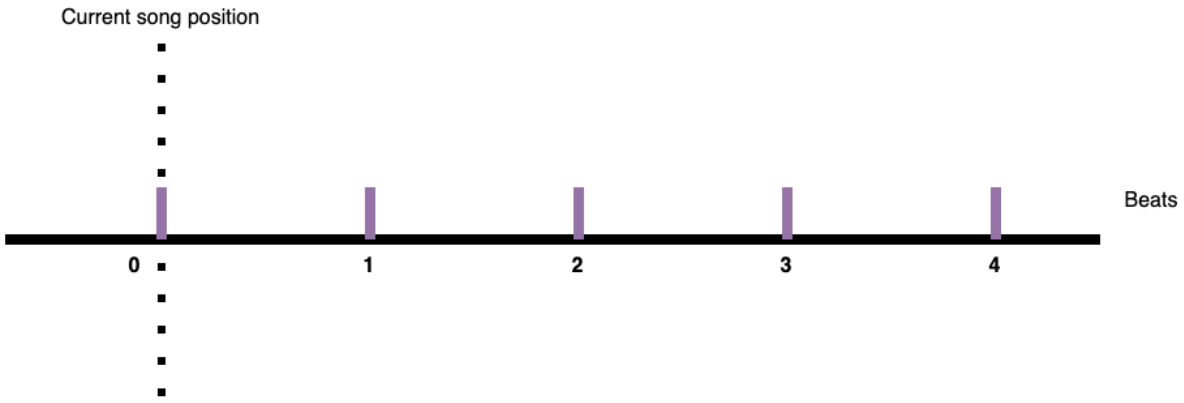
```
Procedure SelectStage (bool isStageSelected, string filepath)

	If keypress == “enter” AND isStageSelected == TRUE then

		conductor = New Conductor(filepath)

	endif

endprocedure
```

Conductor Class

The conductor begins with reading midi data and populating the lane class. It then keeps time, by updating the current song time with the function

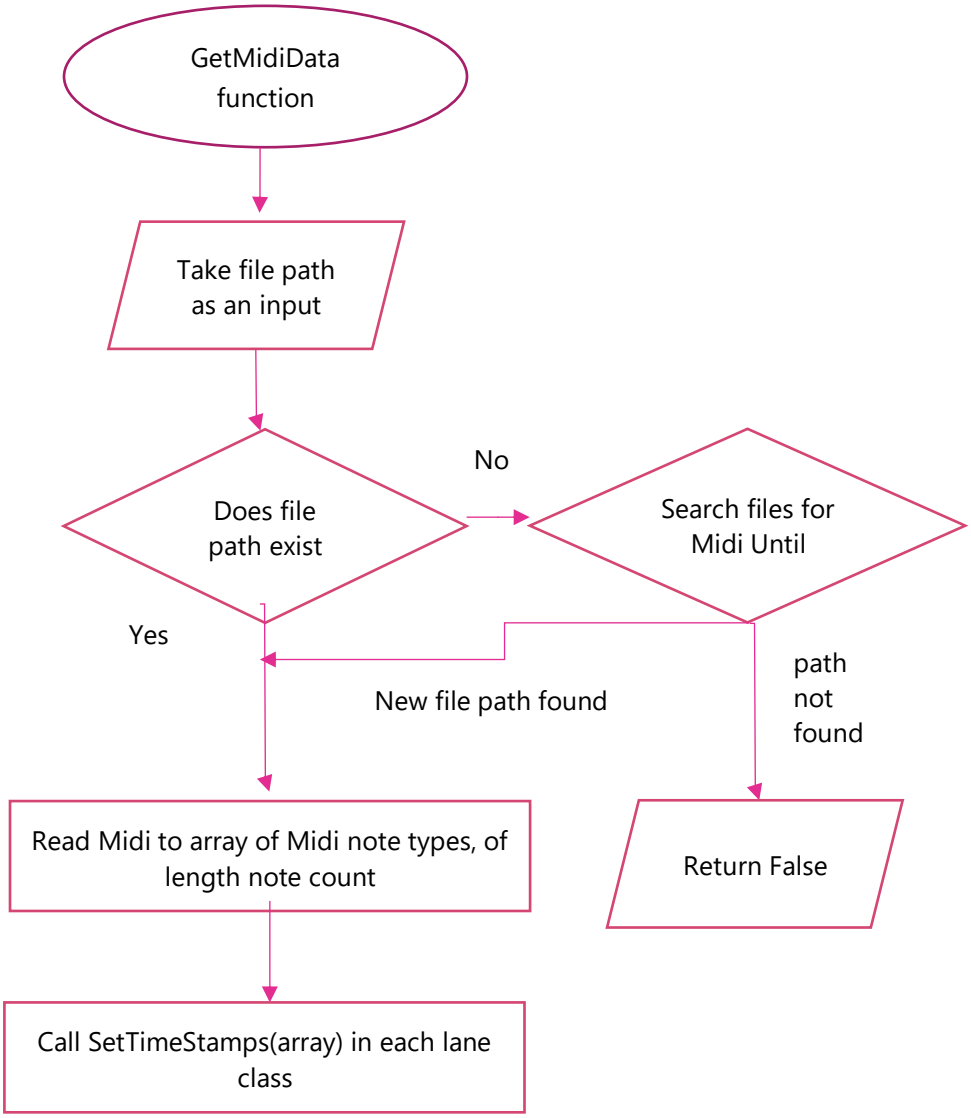
```
GetAudioSourceTime (AudioSourceType)
```

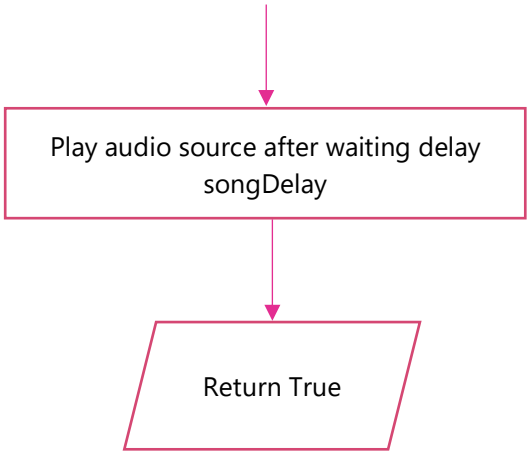
This can then be called by other classes with a static instance of the class to keep track of the stage progress and to spawn notes. The time of the song needs to precise within ~2ms, and need to update approximately every frame. The advantage of using time over beats is the compatibility with the MIDI file format and the ability to then convert the time to beats if needed. The songs position will move along as notes spawn.



Get Data from MIDI file

Midi data should be read from a MIDI file using the DryWetMidi Library, as a DryWetMidi Note type into an array, to then be used by the Lane class to populate it with Notes. The file path input needs to be validated to be read correctly, and if not, the MIDI file should not be overwritten. Audio source should only be played after all of algorithms have run, so when audio is played there is no time after waiting for the game to load.





Note Class

A note is called to represent each individual note during a stage. This means a note is a called at a specific time, called the time instantiated, which is set to the current audio time as soon as it is called. The note completes its lifecycle by moving from its spawn position to its ending position.

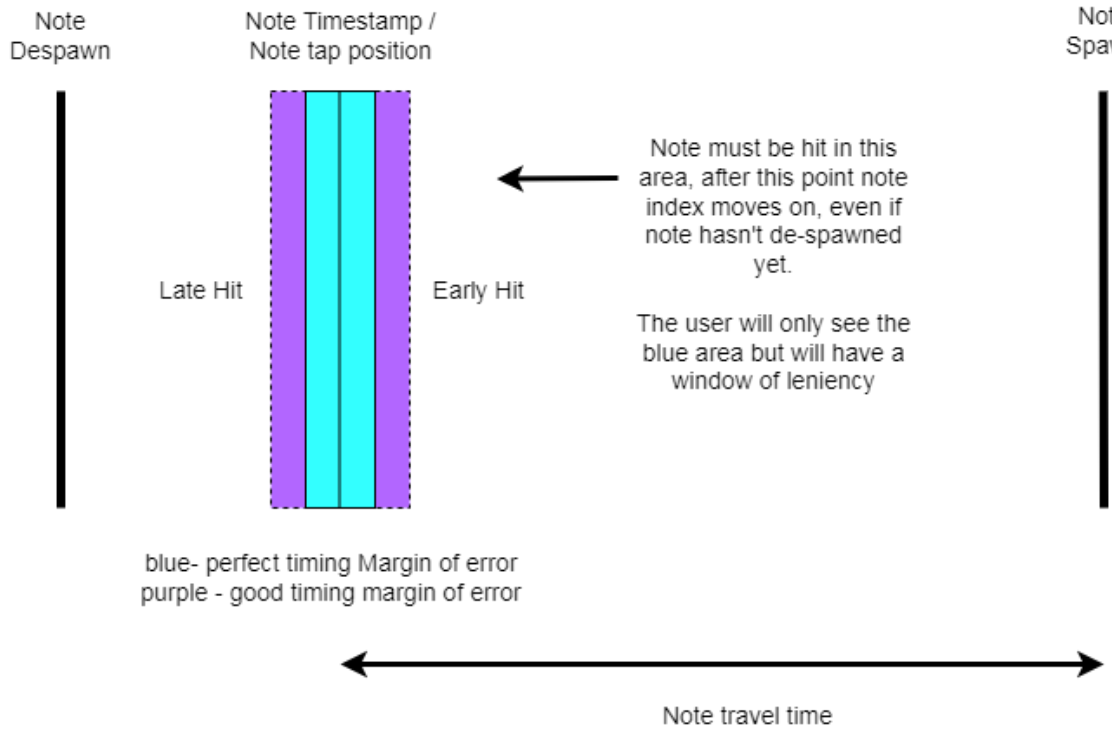
On start function as pseudocode

A note should call this function as soon as it is spawned. Time instantiated with be used in future calculations, and stores the time the note was called. Render sprite assigns the Note sprite to it and draws it on the screen with Unity's handler.

```
Function onStart()

	RenderSprite ()
	SetTimeInstantiated ()

		timeInstantiated = GetAudioSourceTime()

	endFunction

endFunction
```

Update note position as pseudocode

The diagram of a note's position over time is shown here:



A note will move from its spawn to its de-spawn in a set time, known as the note travel time. This time will be interpreted as how fast the note travels. The lower the time the faster the position of the note changes. This will be set initially by the conductor but can be changed in user settings in the game class.

The first variable gives a time difference from its birth to the current time and so this is used as a meter of how far along the algorithm is. The relativePosition is a value from 1 to 0, 0 being at note spawn and 1 being at note despawn. The note is then moved to the total distance needed to travel divided by the *relativePosition*.

This is done by the *moveToPosition()* built in function, which linearly interpolates between the two values, with *relativePosition* as the ratio between them.

The algorithm takes as inputs, the time the note was created, the position of the note's spawn and the position of the notes de-spawn. These are constants defined in the conductor class. The algorithm is then called every frame to create a jitter free motion (the algorithm is iterated over and over again as long as the relative position isn't out of its bounds).

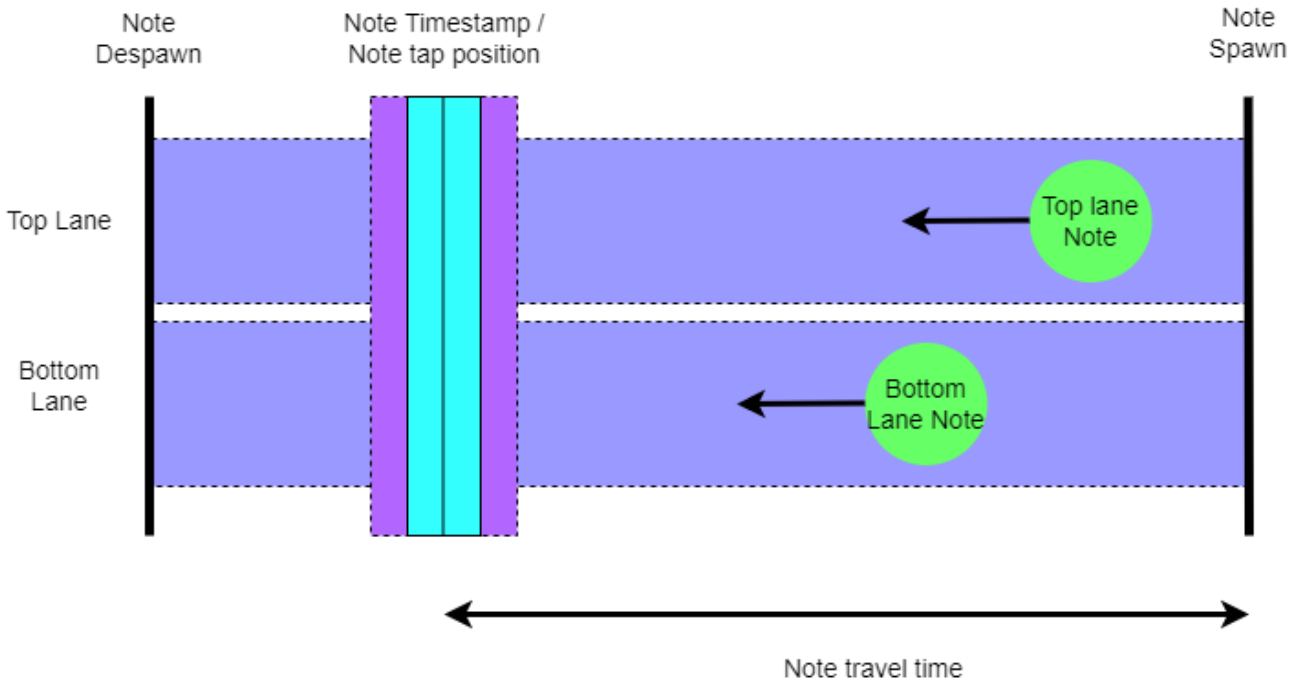
```
Procedure UpdatePosition (float timeInstantiated, float noteSpawnX, float NoteDespawnX,
float noteTravelTime)

        relativePosition = 0

        While relativePosition < 1 then
```

```
            TimeSinceInstantiated = GetAudioSourceTime() - timeInstantiated
            relativePosition = TimeSinceInstantiated / noteTravelTime

            moveToPosition(noteSpawnX, noteDespawnX, relativePosition)

      endWhile

      If relativePosition > 1 then
            SelfDestruct()
      endif

endProcedure
```

Lane Class

Two lane classes will be called once a stage is entered. The lane class is responsible for holding all the notes and handling them once they can be hit.

A diagram of the two lanes as seen from the 2D camera is shown here:



Set note time stamps algorithm as pseudocode

This function takes data read from the MIDI file and returns a formatted list of times for when a note should reach the hit area (its time in the song). An array to hold the time of the notes is initialized and then iterates over the inputted array and adds each item to the new array. The Array must be sorted from shortest time to the longest, so that the notes can be indexed and spawned in that order.

In between adding the notes it checks if the Note matches the Lane's specific note type. For example, the array contains information about C notes, D notes and F notes. If the notes mapped to the top lane were D notes then only the D notes should be added to that lanes specific array.

The function:

ConvertTimeToSeconds()

Is included in the DryWetMidi Library and converts the metric time from the Note's property to time in seconds, so that it can be used in further calculations.

```
Function SetTimeStamps(noteType array[], noteType noteRestriction)

	Float timeStamps[]

	For note in array

		If note = noteRestriction then

			Float noteSeconds = ConvertTimeToSeconds(note)
			timeStamps.add(noteSeconds)

		endif

	Next note

	Return timeStamps

endfunction
```

Spawn Notes algorithm as pseudocode

Spawn Notes takes, an index of spawn notes as a parameter by reference, the list of note arrival times in the timestamps list and the time between the notes spawn and when its timestamp hits (note travel time). Spawn index counts the total number of notes spawned and is initialized as 0.

The algorithm begins with creating a new static list that will hold all the note objects that are spawned. The list type is important so notes can be added and subtracted as they are added and disappear.

Using a while loop, the algorithm loops as long as the Lane instance exists, and the spawn Index is less than the total amount of notes that need to be spawned. Then it checks if the current song time is more than or equal to the spawn time of the Note (adjusted with note travel time because the note needs to spawn earlier than it is hit). If yes then a note class can be instantiated and the note added to the list, and the notes assigned time property for use in later calculations can be assigned here.

The spawn index increments and the algorithm loops.

```
Procedure SpawnNotes (float timestamps[], float noteTravelTime)

	ObjectType notes[]
	Int SpawnIndex = 0
```

```
	While spawnIndex < timeStamps.length then

		If GetAudioSourceTime() >= timeStamps[spawnIndex] – noteTravelTime then

			Var note = new Note()
			Notes.add(note)
			Note.assignedTime = timestamps[spawnIndex]
			SpawnIndex = SpawnIndex + 1

		Endif

	Endwhile

endprocedure
```

Check for hits and misses algorithm as pseudocode

CheckInputs takes, an index of spawn notes as a parameter by reference, the margin of error for timing a hit in seconds, the list of note arrival times in the timestamps list, an input to check to compare to the user input, and a list of the Note objects being spawned.

Input index counts the total number of registered inputs and is initialized as 0 in the function. Using a while loop, the algorithm loops as long as the Lane instance exists, and the input Index is less than the total amount of notes that need to be spawned. Then it checks if the key pressed is the correct keypress for that lane, initialized when the Lane class is created. For example, a left arrow key press. Then the distance between the current audio time and the perfect note audio time is compared to the allowance area (marginOfError).

The hit update function can then be called to handle the hit. The note can then be destroyed, and input index is incremented. This is so that the function can continue onto the next note.

If the key is pressed outside of this area, the note was hit early and therefore missed. This registers as a miss but does not destroy the note . This lets the user try again to hit the Note, when it is closer, whilst being fair.

If the note passes the margin of error after the time is it supposed to be hit, and no input has been received, a miss is registered. The note can no longer be hit and so is written off as a miss, and the update miss function is called to handle it. Notes don't have to be destroyed because they are destroyed automatically at their destroy position. In this case the input index should be incremented anyway so as to move onto the next note and so that it corresponds to the noteIndex.

```
Procedure CheckInputs (int &NoteIndex, float marginOfError, float timestamps[], string
input, ObjectType notes[])

	int InputIndex = 0

	While inputIndex < timeStamps.length then

		If keyPressed == input then

			If Abs(GetAudioSourceTime() – timeStamps[inputIndex]) <
marginOfError then
```

```
                        ScoreManager.UpdateHit(noteIndex)
                        Destroy(notes[inputIndex])
                        inputIndex = inputIndex + 1

                    else then

                        UpdateMiss ("early", noteIndex)

                    Endif
                endIf

                if (timeStamps[inputIndex] + marginOfError) <= audioTime then

                    ScoreManager.UpdateMiss ("late", noteIndex)
                    inputIndex = inputIndex + 1

                endif

            endwhile

endprocedure
```

Score Manager Class

The score manager deals with each hit and miss of a note. At the start of each stage all variables used are initialized to 0 and health is initialized to 10.

In each function the Update UI algorithm will take the new text and update the UI. It will be called in each function to update with the new value, that takes as a parameter all the elements that could have changed, converts them to strings and displays them as texts in the GUI.

Update hit algorithm as pseudocode

When a Note is hit the following function is called

```
Procedure UpdateHit (int noteIndex)

    Accuracy = CalculateAccuracy (accuracy, True, noteIndex)
    Combo = CalculateCombo (True)
    Score = CalculateScore (combo, score)
    PlaySound (hit_sfx)
    Display (rating)

endProcedure
```

This plays a hit sound effect and displays the accuracy rating on screen as a text. The function also calls the functions for calculating new accuracy, score, and combo values. Score calculations needs to be after combo calculations because the score takes combo as a parameter.

Update miss algorithm as pseudocode

When a Note is missed the following function is called

```
Procedure UpdateMiss (string rating, int noteIndex)

	Accuracy = CalculateAccuracy (accuracy, False, noteIndex,)
	Combo = CalculateCombo (False)
	UpdateHealth ()
	PlaySound (miss_sfx)
	Display (“Miss”)

endProcedure
```

This plays a miss sound effect and displays the string “miss” on screen as a text. The function also calls the functions for accuracy and combo. The score function does not have to be called as score does not increase or decrease on a note miss. Instead, the update health function is called to take health away from the player.

Calculate Accuracy algorithm in pseudocode

This algorithm returns a percentage accuracy and takes the previous accuracy, if the last note was hit or not and the total amount of notes spawned. To do this the algorithm reverses the percentage, and gets the total notes hit correctly. Note index is zero indexed and so 1 needs to be added, but then 1 needs to be subtracted again to consider accuracy before current hit.

```
Function CalculateAccuracy (float accuracy, bool hit, int noteIndex )

	Int notesHit = accuracy * (noteIndex)

	If hit == True then
		NotesHit  = NotesHit + 1

	Else then
		NotesHit = NotesHit

	Endif

	Accuracy = NotesHit / (noteIndex + 1)
	UpdateUI.accuracy(accuracy)
	Return accuracy

endFunction
```

Calculate combo algorithm as pseudocode

This algorithm returns the combo as an integer, which is the previous combo plus 1 unless the note was missed, in which case the combo is reset to zero. Combo is always an integer

```
Function CalculateCombo (int combo, bool hit)

	If hit == True then
		Combo = combo + 1

	Else then
		Combo = 0

	UpdateUI.combo(combo)
```

```
        Return combo

endfunction
```

Calculate score algorithm as pseudocode
This algorithm returns the score, which is the previous score plus 10 times the combo unless the note was missed, in which case the score does not change. The score never changes on a miss, so it is never called. The function doesn't need to take bool hit as a parameter.

```
Int CalculateScore (int combo, int score)

        Score = score + combo*10
        UpdateUI.score(score)
        Return score

End fucntion
```

Update Heath algorithm in pseudocode
Update health is another algorithm that does not require a Boolean for hit, because it already is only called when a note is missed. Therefore, it only needs to check if health is below or equal to 0. If it is, then it calls the stage manager to handle the screen change. In most cases the health decreases by a constant.

```
Function UpdateHeath(float health)

        If health <= 0 then
                StageManager.StageFailed ()

        Else then
                Health = health - 0.5

        Endif

        UpdateUI.health(health)
        Return health

endfunction
```

Iterative Tests

In order to create a robust and bug-free game I will perform tests during each stage of development. These tests will be white box tests, and so will be done will all the knowledge of the code I have written and will focus on the most important functions described out in the algorithms section. In each development section, the following test will be done and the results justified.

There are three main types of tests that I will be using:

Valid – This test type considers normal data. Normal data is any value that is expected. A valid test should always lead to a result where the normal data is accepted.

Boundary – This test type considers ambiguous data, or data that is on the boundary of a range. This test may or may depending on the context, and is only applicable when the test data is continuous.

Invalid – This test type considers erroneous data. An invalid test should not be accepted, and the data rejected. For this test, an error message would be a suitable output. Many invalid cases will be handed by validation, which and the user will not have many options to access to these inputs.

To complete the tests, the test data should be manipulated in the way described in the tests. Specific values will be names when the tests are used in development.

Each test is justified by the solution described in the solution overview and structure, and the test can be followed along in algorithms.

Game class

This class goes on top of all the other classes and the functionality of this class depends on other classes so it should be tested after all the others.

N.O	Description	Test data	Test type	Expected result	Justification
1.1	Check if Music volume changes as settings are increased / decreased.	userVolume, using slider	valid	If increasing volume slider, display number increases, userVolume variable increases and game audio gets noticeably louder. Opposite would happen if slider is decreased.	Volume slider should correspond with volume values, in order to change in game volume.
1.2	Check if the note speed changes as settings are increased / decreased.	userNoteSpeed, using slider	valid	If increasing note speed slider, display number increases, userNoteSpeed variable increases and the in stage note speed increases. Opposite would happen if slider is decreased.	Note speed slider should correspond with user note speed values, in order to change the difficulty.
1.3	Check if user settings are read from JSON file.	userFilePath = valid file path	valid	User values are read from JSON file under file path given.	Tests the get user settings algorithm. Must always return user settings, or user settings aren't carried over
1.4	Check if user settings are read from JSON file when the file exists, but the file path is incorrect.	userFilePath = invalid file path	invalid	Invalid file path is rejected and valid file path is located and read from.	Tests the get user settings algorithm. Must handle cases where the files have been moved.
1.5	Check if user settings are read from JSON file when the file	userFilePath = empty file path	boundary	No file path is found, so new file is created with empty values, and	Tests the get user settings algorithm. Must handle the

	does not exist, and there is no file path.			file path is saved.	initial case when user file path does not exist (When the game is first played)
1.6	Check if stage selection load when button 1 is pressed.	Is button1 pressed	valid	Stage select screen is loaded and the stageManager class is called	Tests the start algorithm. There needs to get the main area of the game (the stages).
1.7	Check if the settings menu loads when button 2 is pressed.	Is button2 pressed	valid	Settings Menu is loaded and sliders for values appear.	Tests the start algorithm. There needs to be a way to access the settings.
1.8	Check if the game close when button 3 is pressed.	Is button3 pressed	valid	The open instance of the game terminates	Tests the start algorithm. There needs to be a way to close the game.
1.9	Check if user settings are saved	Slider value changed	valid	Current settings should be saved to a new file, and the old file deleted.	Tests the set user settings algorithm. Needed for data persistence.
1.10	Test background	Game loaded	valid	Background is loaded and plays repeatedly in a loop.	Animated background should be loaded immediately and play smoothly.

Stage Manager class

Like the Game class, these tests should be done after the other classes, as the stage manager class goes on top.

N.O	Description	Test data	Test type	Expected result	Justification
2.1	Try selecting a stage from a list of stage	StageID, mouse, arrow keys,	valid	Stage should be clearly selected from a list when hovered over or moved to with arrow keys. User should be able to navigate to any stage.	To see more information about a stage, it must first be possible to select it.
2.2	Check if Stage information is displayed.	Stage ID, Stage Name, Stage Highscore	valid	Stage information is displayed in a panel besides the selected stage.	Whilst the stages can be selected, the same type of information should always be shown. This lets the user know more

					about a stage before entering it, almost like entering another, smaller but more detailed menu.
2.3	Try interacting with stages	StageID, mouse pressed on button, enter pressed	valid	Stage should transition to the game screen if pressed, and the game should play the song and notes from the stage that has been clicked.	Tests the select stage algorithm. Necessary to be able to precede with the game.
2.4	Return to main menu	Return button pressed.	valid	Moves from stage selection screen to main menu screen.	User must be able to return to main menu to exit game or enter settings.
2.5	Check if high scores are read from JSON file.	userFilePath = valid file path	valid	User values are read from JSON file under file path given.	Tests the get high scores algorithm. Must always return high scores, or user settings aren't carried over
2.6	Check if high scores are read from JSON file when the file exists, but the file path is incorrect.	userFilePath = invalid file path	invalid	Invalid file path is rejected and valid file path is located and read from.	Tests the get high scores algorithm. Must handle cases where the files have been moved.
2.7	Check if high scores are read from JSON file when the file does not exist, and there is no file path.	userFilePath = empty file path	boundary	No file path is found, so new file is created with empty values, and file path is saved.	Tests the get high scores algorithm. Must handle the initial case when user file path does not exist (When the game is first played)
2.8	Check if high scores are saved	New high score set	valid	Current high scores should be saved to a new file, and the old file deleted.	Tests the set high scores algorithm. Needed for data persistence.

Conductor class

The conductor class contains the building blocks for the game and so should be tested more frequently and before the other classes.

N.O	Description	Test data	Test type	Expected result	Justification
3.1	Check if stage data is read from Midi file.	filePath = valid file path	valid	Stage values are read from the Midi, into a game readable format,	Tests the get Midi data algorithm. Must always return stage

				and the stage can be loaded, returns True	data, or else stage can't be loaded.
3.2	Check if stage data is read from Midi file when the file exists, but the file path is incorrect.	FilePath = invalid file path	invalid	Stage Midi file not found and the same name format as input is searched for. If found, file path is saved and read, returns True.	Tests the get Midi data algorithm. Must handle cases where the files have been moved.
3.3	Check if stage data is read from Midi file when the file does not exist, and there is no file path.	userFilePath = empty file path	boundary	Stage Midi file not found and the same name format as input is searched for. If not found, return false, and stage is not loaded.	Tests the get Midi data algorithm. In this scenario, there should never be an initial boundary case.
3.4	Check audio source time at the start, end and at regular intervals.	Audio source time	Valid, boundary	Audio source time should always give the exact position in the song. Once the song is finished, the time should not be outputted.	Tests the get audio source time function to ensure time is kept properly and notes can spawn in sync. Even miniscule time delays should be accounted for.
3.5	Check user has the ability to hit the note in the area defined by tapX.	tapX	valid	Tapping when a note is inside TapX should always result in a hit and the note should be counted and destroyed.	Tap X will be constant but a tapX will need to be decided alongside its area.
3.6	Check notes travel from spawnX to TapX in set time. Vary noteTravelTime, check if notes can be hit with the same accuracy.	spawnX, tapX, noteTravelTime	valid	As note travel speed is decreased, notes travel faster, area to hit notes increases, up to an upper pre-defined limit	The user should be able to choose their note travel time as a difficulty setting, and so all values of this should be tested, and needs to scale properly, and without affecting note spawning.
3.7	Try positive song delay values to test the audio source start time	songDelay = positive	Valid	Audio source should wait the length of the delay before starting.	Conductor is responsible for syncing the audio to the notes generated by the midifile. Song may start later.

Note class

Note class should test the behaviour of individual notes, and notes in a stage collectively.

N.O	Description	Test data	Test type	Expected result	Justification
4.1	Instantiate note from an array and compare with spawn time from time stamp array.	timeInstantiated, timeStamps	valid	TimeInstanciated is the same as time stamp time and note is instantiated at the correct time.	The time instantiated variables will be used in key calculates for the notes and so should be accurate, and the note should be spawning at the right time.
4.2	Check timeSinceInstantiated updates with each frame	timeSinceInstantiated	valid	Time increases, with the instance the note was created being 0	timeSinceInstantiated is used in calculations in the rest of the UpdatePosition function
4.3	Test note movement by monitoring RelativePosition and varying EndX (EndX is typically constant)	RelativePosition, EndX	invalid	The further EndX, the faster the notes should move. Constant EndX should be decided.	Tests update position function. RelativePosition determines note interpolation and so can if behaves correctly, movement will be correct.
4.4	Test note movement by monitoring RelativePosition and varying noteSpeed	RelativePosition, noteTravelTime	valid	As noteTravelTime decreases, notes will travel faster, and vice versa. Note positions updates linearly.	NoteTravelTime will be variable in the game and so ensuring notes behave correctly for all ranges of the variable will be vital to the game.
4.5	Test note movement by monitoring RelativePosition and varying spawn X (spawnX is typically constant)	RelativePosition, SpawnX	invalid	Note positions updates linearly with each spawn X position.	Tests update position function. RelativePosition determines note interpolation and so can if behaves correctly, movement will be correct.
4.6	Input relative position values of 0, 1 and 2 inside the UpdatePosition function for a single Note.	Relative position = 0, 1, 2	boundary	While loop entered for relative position 0, 1. While loop exited for relative position 2 and note self-destructs	Prevents a logical error where the while loop has no end condition and the loops infinitely.

Lane class

Of the two-Lane classes that exist for each stage, test each individually. Player input refers to the input key for hitting a note in the corresponding lane.

N.O	Description	Test data	Test type	Expected result	Justification
5.1	Test noteRestrictions against note Arrays, for each combination test if note is added to its corresponding lane.	noteRestriction, noteList	Valid, Invalid	If *noteRestriction* is the same as pitch class of the note from the Note List, then the note's time should be added to the note List.	Used to sort the Note list into the two lanes.
5.2	Check if timestamps array is populated with note times corresponding to various note List inputs.	Note List, timeStamps	valid	Note list is fully converted to time stamps for each lane.	Tests the set Timestamps function. Notes must be made from note list, notes should all be added accurately.
5.3	Check if all notes from timestamps list are spawned in the lane.	SpawnIndex, timeStamps	valid	First note through to end notes appear on screen, and at the end of the stage *spawnIndex* = *timeStamps* length	All notes must be spawned in order for the stage to be complete.
5.4	Vary note travel time to test note spawn position. Vary between 0.1 and 10, note travel time cannot be less than or equal to 0.	NoteTime = 0.5, 1, 2, 4, Note spawn position in playback editor, timeStamps	valid	Notes should spawn in the same place every time no matter the note time.	Note time will be varied in the game, and so spawn time must change for the hit area to remain constant.
5.5	Input a 'perfect' hit exactly as the measure of the score increments (when a beat occurs in the song).	Player input,	valid	Note is destroyed exactly on the hit indicator, perfect score received, hit method called.	The base case for the gameplay relies on the perfect hit being timed correctly. Platers should be able to hit the note exactly on the mark and get the highest score.
5.6	Input a delayed or early 'perfect' hit, by the margin of error	Player input, marginOfError	boundary	Note is destroyed around the hit indicator, perfect score received, hit method called.	Tests the boundary case of receiving a perfect hit timing. Perfect hit timing should be lenient enough to where this case is common.
5.7	Input an early or late hit in the 'good' area	Player input, marginOfError	valid	Note is destroyed further from the hit indicator and not	Tests the size area of the margin of error and if the user can

				exactly on the beat. good score is received, hit method called.	hit a note in that time. This ensures the notes can be hit outside of the perfect area
5.8	Input a very early or very late hit to get a miss hit.	Player input, marginOfError	invalid	Note is destroyed but registers as a miss, player receives a miss, and the miss method is called.	Tests features to prevent repeated pressed by creating an area where very late or early hits are punished.
5.8	Miss a note by pressing no inputs until it moves outside of miss area	Player input, marginOfError	valid	Note continues moving until the end of the screen and is destroyed, player receives a miss and the miss method is called.	It is vital that a note can be missed in the game.
5.9	Test hit effects	Player input	valid	Hit effects happen instantaneously on the correct player input.	Tests hit effects to ensure user interaction works correctly.

Score manager class

Score manager can be checked with the UI given at the end of each stage and from save files.

N.O	Description	Test data	Test type	Expected result	Justification
6.1	High score is recognised and saved	Score > high score	valid	A new high score is set and sent to the stage manager class to save	Tests sending data between classes. Data must be saved to files for a high score to be set.
6.2	Check combo value if note was successfully hit.	Combo, input hit	Valid	Combo increments	Testing calculateCombo function. Case for gaining combo is a valid hit.
6.3	Check combo value if note was missed.	Combo, input miss	Invalid	Combo resets to 0	Testing calculateCombo function. Alternative case of 6.2, and so needs to be tested or else combo can't be lost.
6.4	Check score value if note was successfully hit.	Score, input hit	Valid	Score increments	Testing CalculateScore function. Case for score incrementing is

					a valid hit.
6.5	Check score value if note was missed.	Score, input miss	Invalid	Score does not change	Testing CalculateScore function. Alternative case of 6.4, and so needs to be tested.
6.6	Check accuracy value if note was successfully hit.	Accuracy, input hit	Valid	Accuracy percentage re-calculated, and increases (or stays the same if 100%)	Testing CalculateAccuracy function. Case for increasing accuracy is a valid hit.
6.7	Check accuracy value if note was missed.	Accuracy, input miss	Invalid	Accuracy percentage re-calculated, and decreases	Testing CalculateAccuracy function. Alternative case of 6.6 and so needs to be tested or else accuracy can't decrease
6.8	Check health value if note was successfully hit.	Health, input hit	Valid	Health does not change	Testing UpdateHealth function. Case for keep health is a valid hit.
6.9	Check health value if note was missed.	Health, input miss	Invalid	Health decreases, if health below zero then stage failed, and failed screen shows.	Testing UpdateHealth function. Alternative case of 6.6 and so needs to be tested or the stage can't be failed
6.10	Check if sound effects play after note hits and misses .	Player input, AudioSource	valid	A hit should be followed by the hit sound effects and the same for a miss	Miss and sound effects must happen in succession to the events. Without them there is no confirmation to the user.
6.11	Test game state in song failed screen	Hit input, health = 0	invalid	User should not be able to interact with the stage, all game functionalities should be paused.	After a stage has been failed the user should restart from the stage selection screen to ensure fairness and difficulty.
6.12	Test song failed screen functionality	Health = 0, return clicked	valid	Button press returns the user to the stage selection menu.	Stage failed screen should let the user try again.
6.13	Test Stage complete screen	All notes passed, health >0	valid	Accuracy, Score, and highest combo should all be displayed correctly, and button	Allows performance statistics to be seen and time for the user to return back to the

				navigates back to stage selection screen.	menu.
6.14	Try setting invalid score	Score > maximum	invalid	If score is over maximum score, score should not be allowed	Tests validation of score at the end of the stage before score can be saved as high score.

Features – Variables and Data Structures

Variables and data Structures are locations in memory that are assigned values over the course of the program, whereas files are read only once and accessed from secondary storage (and written to many times)

Variable types

These are the uses and types of the variables I will need in the program:

- Integers – used for indexes and exact values
- Floats – used for compatibility and to save memory
- Doubles – used for times and large number handling
- String – used for text displays
- Booleans – used for marking game states.
- Constants – used for variable that won't change during run time. These will be clearly defined at the start of the class to be used throughout the program.
- Time – most time variables will be converted to seconds and so stored in a double type for ease of use.
- Object instances – used to call methods
- Pointers – used for data structures
- Unity specific variables – used for running game code

All variables that I will be using along with their type and level of protection have been identified in each class in the Solution Structure section.

File types

Variables do not persist after the game is closed and so any data that needs to be stored permanently and in non-volatile storage needs to be saved to secondary storage.

Data I will be reading and writing in this way is explained in the Game, score manager and stage manager classes but includes, reading note data from midi files, reading and writing user preferences set in settings. And reading and writing user high scores. I have used the following file types:

MIDI

A file format that standardizes digital music communication and stores information about individual tracks, the bpm and length. By previously composing a midi file with drum notes on top of a song's beats in a music producing software, the midi file containing drumbeats can be interpreted by the DryWetMidi Library as notes. Through this method, stages in the game are stored as modular files that can be read from and modified and displayed as needed. The player will not need to interact with these files as the game will have a stage (song) selection menu, and the script will only need to interact with them through reading.

JSON

JSON is a file type that stores simple data structures and objects in JavaScript Object Notation (JSON) format, which is a standard data interchange format. JSON files are lightweight, text-based, human-readable, and can

be edited using a text editor and can be encoded and decoded with unity's JSON serialization. They are an effective tool for sending data between the application (game) and the server (the storage disk).

These can be read and written to a class with Unity's JSON utility function [10]

I will be using JSON in the Game class and the Score manager class to save user data and to read user data and stage data.

OGG
OGG is a lossy compression audio file format that has higher quality than MP3s. I will use these to playback the song for each stage and for all the sound effects in the game. Lossy compression is the most suitable to use because it reduces the overall file size of the game. This file type will only be read, and audio playback will be handled by unity objects, and so no decoding is needed.

Data Structures
Data structures are necessary for handling data without using excess variables. Unity has many of these data structures built so with functions, so I have not had to use pointers in my solution. I have used the following data structures in my solution:

Array
An array in an ordered, static, of elements of a single type. In my solution arrays will store the lanes, as there is always a constant number of lane objects, which are the same type.

Using arrays will let me index these elements and access them for calculations in order.

List
A list is a dynamic, set of elements which can occur more than once, and be of more than one data type. They are like 1D arrays in that they are zero indexed and are accessed in the same way. I will use lists to store all note time stamps for spawning objects.

This is because they will have to be defined as empty first and then be populated after the note are read from the MIDI and so an array cannot be used.

Queue
Queues handle linear lists of data. They use a first in first out (FIFO) data structure. Whilst my solution does not explicitly use any queues, the way note objects spawn is using a queue like structure (implemented using a list). Notes are taken from the list in the same order that they were added.
This is done so that notes spawn in the order from first to last and not the other way around.

Dictionary
A dictionary is an abstract data type that defines an unordered collection of data as a set of key-value pairs. Each key, which must be unique, has an associated value. A dictionary behaves similarly to a database and will be used to read and write user settings and stage data as described in the JSON section. Dictionary can easy be parsed between JSON files and classes, and this makes them useful in storing unique types of data together. For example, a list would not be able to store an *int* and a *string* without another class definition.

Features – Validation
Before variables are used for processing they should be validated, using one or more types of checks (range check, length check, limit check, presence check, etc). This prevents any run time errors that may be caused from manipulating inaccurate variables. Data should be valid and accurate at every stage of the game and can

be kept consistent with each playthrough using data validation. Very little input validation needs to be made because the game has few user inputs and so most validation will be precautionary.

Validation done can be done in setter functions in classes, with conditionals during the game logic or, by catching errors using error logging similar to try expecting in python.

Variable	Type	Validation	Time of Validation	Justification
health	float	Check if less than starting health and more than 0. If below 0 should be set to zero.	Whenever health is updated and at the start and end of each stage.	Health should never be negative or be higher than initial health, or else health bar fails.
score	int	Check if 0	At the start of each stage.	If *score* is not 0 at the start of the stage the user will not be able to play the level properly. Score should never have a decimal calculation preformed on it.
score	int	Check if less than upper bound	At the end of each stage.	*Score* will have an upper bound that prevents the user from getting over the maximum score on the leader board. Although this shouldn't happen if score is validated at the start of the stage, it is still a useful check. Needs to be recalculated at the end of each stage using the total number of beat object, and factoring in combo, when every note is hit perfectly.
rank	string	Check if present	At the end of each stage.	Rank should be calculated at the end of each stage before the stage complete screen is loaded, or else the display text fails.
accuracy	float	Check if between 0 and 100	Whenever accuracy is updated, and at the start and end of each stage.	Accuracy is a percentage and so should only be between 0 and 100. Accuracy should start at 0 for each stage or else the user will not be able to play the level properly.
filePath	string	Check if path exists.	Before reading user data and song data,	If *filePath* does not exist, error will be thrown and file cannot be opened. Furthermore file paths can change from system to system for user's playing the game on different operating systems, or the user could change the locations of files, meaning that *filePath* should be checked with every launch

				of the game.
noteArray	float	Check if populated.	Before notes are spawned	Note array gets populated at the start of each stage and so should not be empty. Game needs values of notes to spawn to be played.
relativePosition	float	Check if between 0 and 1.	During relative position calculation	Used for linear interpolation between to notes spawn and end, error will be thrown if value is out of range.
songTime	double	Check if between 0 and long length in seconds.	Song	Song time is used to sync notes and so should never go out of bounds. Stages should end once song time reaches the end.

User useability

My game with have several graphical elements preceding the gameplay, all in a colourful, playful, and futuristic theme. In my research I found that the screens would be needed for the user interface would be the start screen, the settings screen, the song selection screen, the gameplay screen, and the post song screen.

Font

The font I will be using is called DF POP Mix by DynaComware. The same font is used in several rhythm games including "Project DIVA" [11]. The font is modern and has a pleasing geometric look. The font would be used for the rhythm game title, the main combo and score texts and any other bold areas. The font is very readable, despite its rounded corners.

A lighter counterpart font would be used to compliment the main font. For areas such as the buttons, the song names and high scores I would use the font Aller – a sans serif font family created by Dalton Maag in 2008 for the Danish School of Media and Journalism. Below I have Aller Light.

These two fonts will pair together to immerse the users. From my interview and game solutions analysis, it was clear the fonts should be modern, readable and match the feel of the fast paced, EDM music.

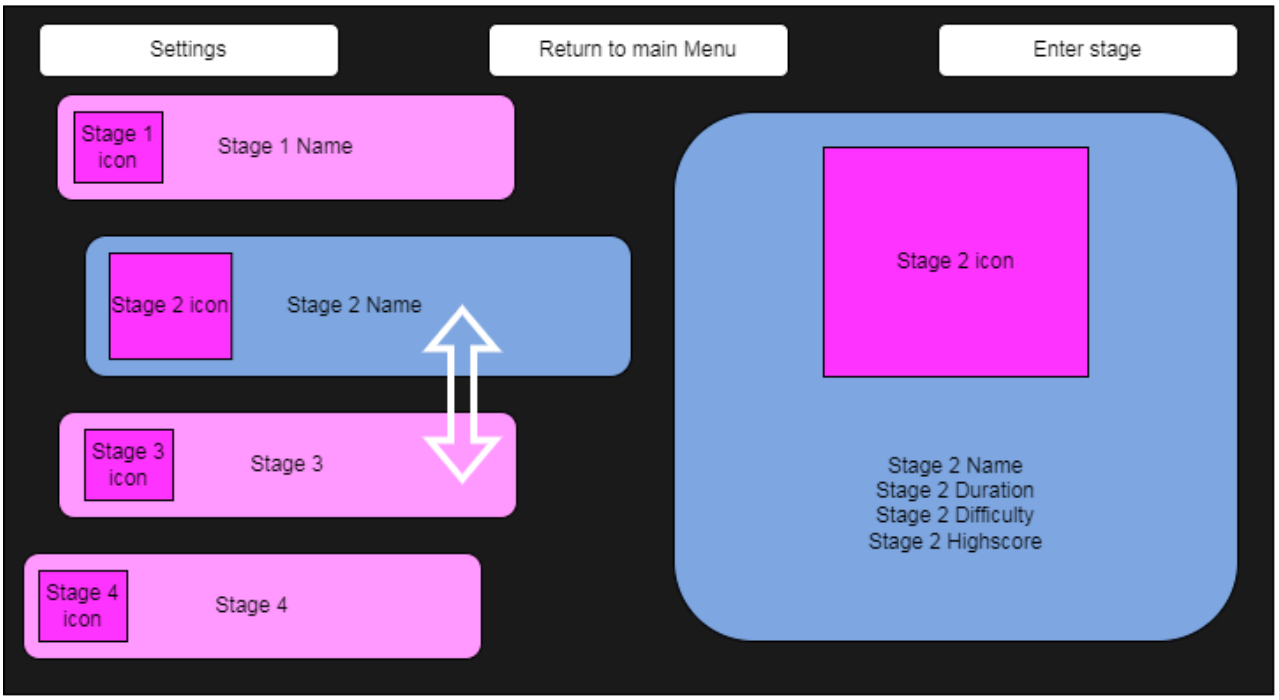
Main screen
This will be the first screen the user sees when they open the game and the last screen they see before they exit the game, and show should give an impression of the overall game. This scree will have the least functionality, but some criteria identified in the interview and game analysis are:

- Stylized Game Title
- Large buttons, separated from the background – Enter stage selection, exit game, enter settings menu
- Simple background

The setting menu should lead on from the main screen and have similar button layouts. The black background will be replaced with a simple / animated background, to be decided later in development.

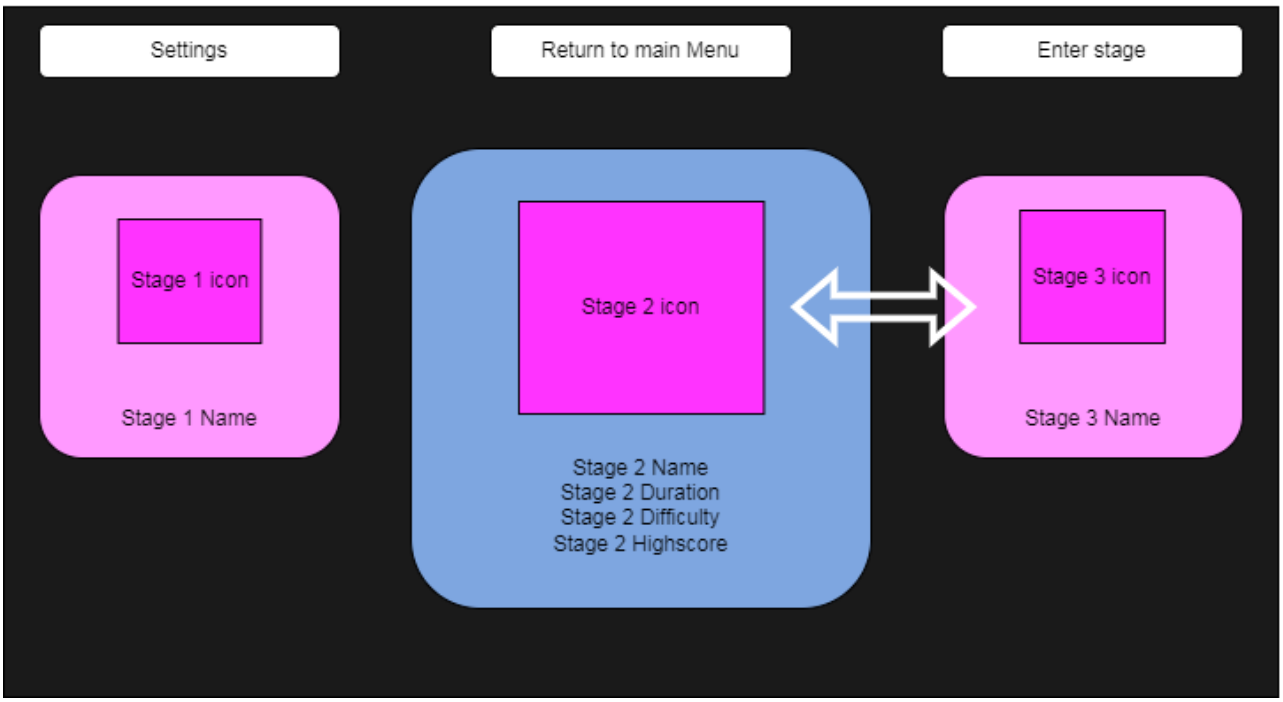
Song selection screen
Other than the game play screen, this is the most important screen, as the user will be interacting with all the all of the types of songs here and it will act as a hub for all the stages. Examples of an effective song selection screen can be found in "Muse Dash" and "Osu!". Both of these take a different approach to song selection. The first is a side scrolling, selection menu, that moves left and right, highlighting the current stage fully. The second option, used in "Osu!", shows all the stages in a downwards list and shows less of the stage on the right.

My vertical scrolling design is shown here:



The advantages of this version are its compactness and clear information of each song. This design is also used in similar rhythm games on the market and would let the user see all of the songs in the game at once. And so limits the time searching for a song.

The second horizontal design is shown below:



The horizontal song select screen is used in "Muse Dash", and when playtesting and researching the game I found this menu was easier to navigate and use, however it was much harder to find the stage I wanted. My game will only have 3- 4 stages so choosing this design would not be a problem, however I prefer the visual layout of the first design.

User accessibility

In order to build a game enjoyable for everyone, and since my stake holder is someone who is looking for a new experience, they may not know what they are entering. Additionally, users with disabilities who are looking to train their rhythmic sense and co-ordination should be able to play the game without having to overcome any obstacles standing in their way.

Photosensitivity

My game will have rapidly moving objects and at times rapid lights flashing from the effects of the notes being hit. This could seriously affect users with epilepsy. One method of making my game accessible to everyone would to implement photosensitive settings. These would turn off any particle or light effects. Another method would be to give an epilepsy warning to users before any stage with quick moving objects.

Lastly, I will take preventative measures to avoid the background or the light changing colours too quickly. Since this is hard to quantify, I will have to decide this during development. A test I can do to ensure this and to inform a photosensitive friendly solution would be to time the duration between each time an affect plays.

Colour blindness

My game will have sufficient contrast to be able to be played by colour blind people and so I will not develop a colour-blind setting.

Control scheme

My game will be using the most common input method for a rhythm, a keyboard, although a user's own input method can be used as long as it can be set to equivalent Unicode inputs.

The game will be comprised of screen to navigate before reaching the gameplay. These will be reached via typical methods of mouse inputs, but the user can also reach the game screen with additional controls. These will use arrow keys, the left hit button for exit or back, and the right hit button will be used as a confirm or enter button.

As for the gameplay, the control scheme will be the quintessential rhythm game key layout. F and J keys allow for comfortable gameplay with both hands. For one-handed users, this would commonly be changed to Z and X for left-handed or DOT, SLASH for right-handed. Using just 2 inputs is not only easier for beginners but also improves user accessibility.

To provide utmost accessibility, all of these inputs would be able to be changed in the settings menu. Since my game aims to be minimal and so as disability friendly as possible, the two game play keys should be able to be remapped to any two inputs, so that users can attach their own keyboards or controllers to the device. An example of this would a user who uses a large mouse as an input device and so would want to change the game inputs to match to the device.

Planning Post Development Tests

A final test should be done on the product before completion. This test should be done to check the functionality of the game and compare the outcome produced to the success criteria. I have identified **metrics** that measure how much of the feature is complete. To test these metrics I will select **test data** in the form of user inputs, setting variables and stakeholder feedback if the metric is not objective. Finally the test data can be compared against the metric to see if the test was successful, and this is quantified in the **acceptable test result** column.

N.o	Feature	Metric to measure success of test	Test data	Acceptable test result	Justification
1	Start screen	A simple screen should display the title of the game and menu options to progress. Should be visually appealing and be in the style of the game.	Stakeholder feedback, Click buttons to change between screens	Pressing buttons transitions between screens. User can exit game.	Provides a starting point for the player.
2	Song selection screen	Display a menu where the player can see all the songs should have basic information about the songs including difficultly and name. Should have a clear link to the leader board with a preview of the high score as the user hovers over the song.	Stakeholder feedback, Use arrow keys to change between selected stage, Click button to enter stage	Stage can be selected. Information about screen is shown. User can enter stage selected.	User can choose a song and enter the stage
3	Tutorial stage	Should be the first stage the player should enter. Unlocks other stages of the game and pops up short instructions during the song on how to play.	Stakeholder feedback, enter tutorial from stage selection	Game mechanics explained. Progression after completion.	New players should have a brief introduction to the game.
4	High score	Save the player's best score for a stage. The high score should carry over when the game is closed and re-opened. Should be non-volatile	Set high score values, Re-launch game	Same values displayed correctly when game is re-launched.	A competitive feature to encourage improvement
5	Leader board	Holds information about a user's score, highest combo, rank, and accuracy for a stage. Displays separately for each stage in descending order of scores. The leader board should carry over when the game is closed and re-opened.	Set multiple score values, Re-launch game	Scores displayed correctly and in order.	An added level of complexity to high scores; A separate display for player scores.
6	Settings menu	A menu for changing sound settings and difficulty settings. Settings should carry over once the game is closed and re-opened.	Stakeholder feedback, Click buttons to change between screens, Change settings values with slider Re-launch game	Settings values update globally across the whole game. Settings values correct when game is re-launched.	A menu for changing settings lets users accessed their saved settings.
7	Sound settings	Sliders to control music volume, effects volume, and overall volumes. Sliders set volume percentages from 0 - 100%	Input volume percentages from the settings menu, test unity audio play volume	Sliders change audio levels including music and sound effects volumes.	Necessary for user using different output devices.

			level variable		
8	Note approach speed setting	In the settings menu, players could the speed at which notes approach the hit area. How much time you have to hit the note would be compensated for if this setting was changed so that it only affects how fast you want the notes to appear.	Input note approach speed value from the settings menu, play a stage and repeat tests	Sliders change note approach speeds. Notes approach at a variable rate.	Additional settings make the game more accessible and in line with stakeholder needs.
9	Difficulty settings	Changes health and hit area to increase or reduce difficulty. More health would make stages easier to pass and larger timing windows would make notes easier to hit. The reverse would apply for a difficulty increase.	Input difficulty float from the settings menu, test hit area	Difficulty float affects hit area proportionally.	Not necessary, but would help very new and very skilled players
10	Game screen	Game screen with a background, hit area, approach lanes, and a HUD (heads-up display). This is where all the stages will be played, and the game screen needs to be able to support all the different songs – including displaying oncoming notes and audio and visual effects from various sources.	Stakeholder feedback, Visibility of elements, Update frequency.	Contains UI that updates as the user plays the stage.	The environment in which the game takes place.
11	Song complete screen	After a song has been finished and if the user has not failed, the screen should show all the statistics, including score, combo, accuracy, and rank. The user should then be prompted to return to the song selection screen.	Stakeholder feedback, input out of bounds variables: score, combo, accuracy, and rank Can be reached from completing song only, input enter key to exit.	Positive reception about style, displayed immediately after stage, correctly displayed all variables, key press exits screen to selection screen	Concludes a stage and lets user see their results.
12	Multiple stages	In the stage selection screen, the user should be able to choose from at least 3 stages. These stages should have different note arrangements and should vary in. song choice difficulty. The length should be at least 2 minutes for an effective and enjoyable	Stake holder feedback, enter each stage from the stage selection screen and play through to the end.	Stages are enjoyable to play. All notes spawn correctly, and statistics are shown accurate.	All stages that are developed for the game should be tested to ensure that they can be completed.

		stage.			
13	Hit area	A rectangle outlining where the notes should be hit, positioned on the left of the screen. Once notes go too far out of this area they are destroyed, and the player gets a miss. The hit area will be composed of several sub sections expanding outwards, allowing for timing leniency to get a late or early hit, depending on the difficulty set.	Stakeholder feedback, change leniency variables, hit notes on time, hit notes in leniency area, hit notes outside of leniency area,	Positive reception about timing difficulty, leniency changes hit area size, notes can be hit in each zone, with different scores gained	Needed to show the player where to hit the notes
14	2 object lanes	Two areas where the notes will come from, both horizontal and moving from left to right. One lane will be at the top of the screen and one at the bottom. The player will effectively move between them as they hit the notes.	Stakeholder feedback, Input note array with a note each second, from time = 0 to end time,	Positive reception about lane size, input note, note array fully displayed in lanes	Creates 2 separate areas for the player to hit the notes, needed to display the notes.
15	Hit objects (Notes)	Circular objects to be hit by the player. Spawned from the same position in one of two lanes and approaching the hit area at a constant speed. Player inputs a press to destroy the notes and get score and combo from this, whilst also being judged on their timing. Hit objects are abstracted notes of the song, generated from a dictionary.	track note sizes, object spawn location, object destroy location, object hit response, object miss response, object, score, input note hit	Hit objects spawn according to the set stage, score changed with hit/miss, input note hit on time always destroys note.	Hit objects are essential parts of the game, and are needed for the user to play the game.
16	Long beat objects	In game objects that act similar to hit objects but lasting for longer. The player must hold the key down for the duration of the note to get the full score.	Input key for entire duration, input key late, input key early, end key input late, end key input early	Results give perfect score, late gives less score, early gives a miss, end late gives a miss, end input gives less score.	Allows for more dynamic stages.
17	Hit input	The player will have two keys that they can press to register a hit. These will correspond to the top and bottom lanes and are used to hit approaching objects.	Input upper hit, input lower hit inside and out of stage	Both respond with hit accuracy, when in stage. Outside of stage no score changes, inside score changes with	Needed so that the user can hit the notes on the beat.

				notes hit	
18	Hit sounds	Hit sounds will occur whenever a hit input is detected and will tell the player they have inputted it successfully.	Input hit on note	Sound plays	Game should respond to the user's inputs with audio outputs.
19	Hit effects	Hit effects will occur whenever a hit input is detected and will tell the player they have inputted it successfully. To do this, the respective hit area will glow for the duration of the key press.	Input hit on note	Effects play on lane position	Note objects should respond to hit inputs to make to make the user feel more involved with the music.
20	Animated background	Background playing in conjunction with the game, either moving or repeating a simple effect in time with the beat of the music.	Stakeholder feedback, Enter stage, exit stage	Background appears during gameplay, stage selection and main menu, without being too intense.	A background would make the game more interesting for my stakeholders
21	Score	A live counter with the player's points, which are gained through higher combo and hitting notes, with a reduction in score if the hit is miss timed.	Enter stage, input all hits, input all misses	Score updates with any hit inputs and misses	A player's performance should be measured with a numerical value.
22	Combo	A counter incrementing after each successful note hit, resetting to zero if a note is missed. Used for calculating combo.	Enter each stage, alternating hits, and misses	Combo updates with hits and misses	A combo should let the user know how many notes they have hit in a row, in order to give score multiplier.
23	Health bar	A bar that depletes if the user misses too many notes. Each miss takes a percentage off the health bar, meaning too many misses causes the player to hit zero health and fail the map.	Enter each stage, input score, input miss	Health updates with any hit inputs and misses	Needed as a measure for the player to see if the stage was too challenging.
24	Song failed display	After depleting the health bar, the user fails the stage and is given a clear, failed message. All game logics stops and fades away. From here they either get the option to retry or go back to the menu. Text should	Stakeholder feedback, Enter stage, input miss, let health go to 0.	Game transitions to this display whenever health goes to zero.	Important feature to visual show a song has been failed, and to encourage the user to try again.

		appear in the centre of the screen for most readability. All gameplay stops, including music.			
25	Accuracy	Accuracy text shows up under the score as a constantly updating percentage (out of 100%). 100% accuracy being fully perfect timing and 0% being every note was missed.	Enter each stage, alternating hits, and misses	Accuracy updates with hits and misses.	Should provide additional and accurate information to the user.
26	Rank	Part of the song complete screen – assigns the player a rank based on score after a song is complete. Ranks going from C to B to A to S to S+	Enter each stage, alternating hits, and misses	Rank calculation displayed accurately.	Adds a lot of interest to the game, and encapsulates score and accuracy.
27	Timing indicators	After each note, text displays depending on how early or late the user's press was. Either a miss, good or perfect is displayed.	Enter each stage, input late hit, input early hit	Timing indicators appear clearly after each hit.	Important for the game to feel responsive and for the player to know how well they are playing.

Development

This section is my development diary of how I created the final project. Each decision will be justified and I will test the prototypes as I go.

Requirements for development

The iterations will build on the next and so they should be as modular, well documented and have infrastructure for the next, so that more functionalities can be added on top.

Iterations have equal priority, with iterations up the iteration 3 being required to complete the game. Iteration 4 will have less priority then 1-3 and will be worked on once the others are complete to add extra features if there is time.

Iteration 1	Iteration 2	Iteration 3	Iteration 4
Create midi file asset for first stage, Reading notes from the Midi file into game, Displaying notes in time with music, Registering inputs, Counting and displaying score, combo and hit accuracy, Displaying timing indicators, Playing hit sound effects, Playing miss sound effects	Stage selection menu functionality, Displaying stages in selection menu, Displaying stage information in selection menu, Adding a health bar metric, Stage failed screen, Stage finished screen,	Creating additional midi stages, Adding main game menu, Controlling user settings, Reading and writing user settings, Reading and writing user high scores, Displaying user high scores	Backgrounds, Button sprites, Note sprites hit effects, miss effects, Long hold notes, Hidden notes Tutorial stage

Languages and Libraries

My project will be coded in Unity's native framework, .NET. Under this, I will be using C-SHARP (C#) which is a versatile and efficient language developed by Microsoft and is used primarily for application development. The object-oriented properties of the language allow me implement to my solution as a system of objects, with each script being its own class with attributes and methods.

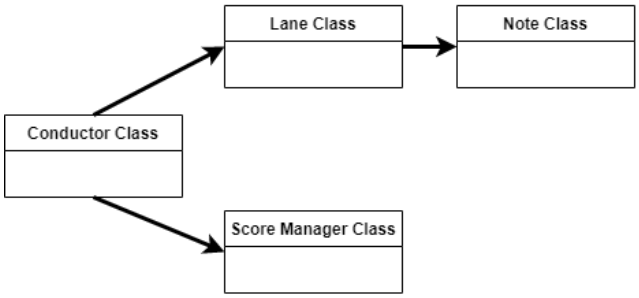
Using C# also allows me to use a .NET library for reading notes from midi files, and processing notes on the music scale. The DryWetMidi library [12] is used for processing and manipulating MiDi files and can convert MiDi notes to high level objects, which I will use to read data for the stages in my game.

Learning Unity

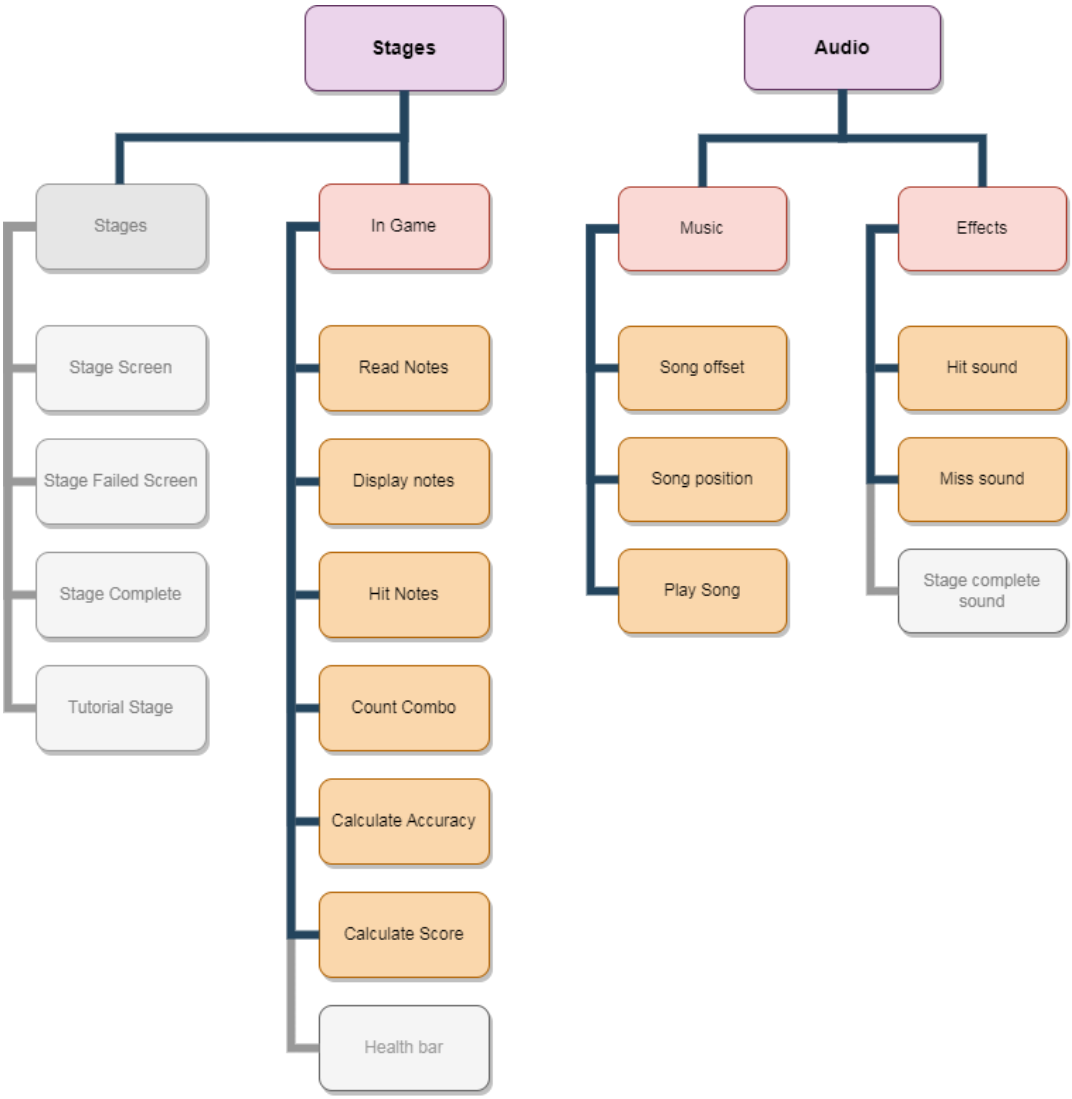
Before coding I had to get familiar with the Unity engine. I used YouTube tutorials to learn the fundamentals of the software and language. [13] [14] [15] [16] [17] [18] [19]

Iteration 1

This iteration will cover the basic template of each gameplay stage, and all the most significant features outlined in analysis. Each of the features shown in the diagram are part of the highest priority success criteria. To achieve the success criteria, I will code the main functionality of the following 4 classes.



Together, these should allow for the following features (the coloured blocks will be programmed in).



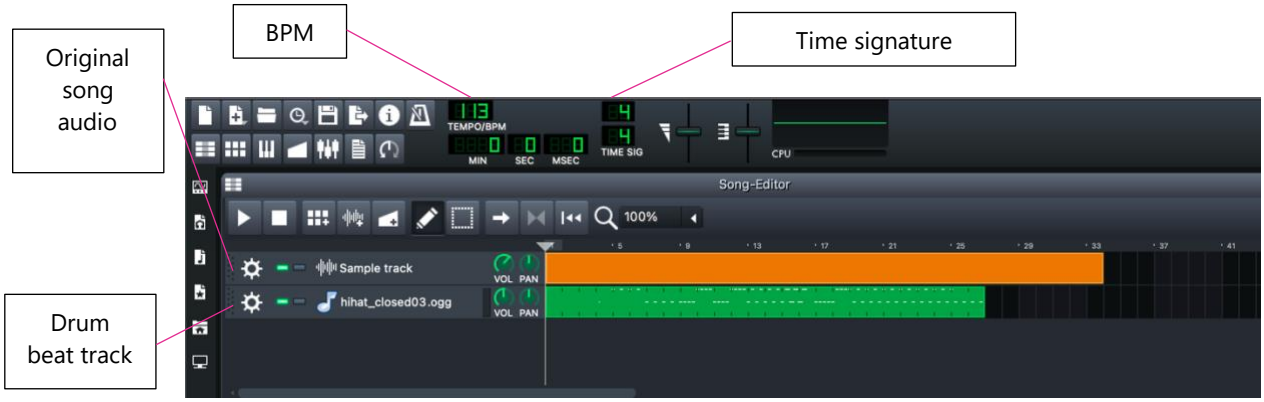
Creating Assets - Feature 1.1

Before I could program my game, I first had to create data that could be manipulated. The game's logic will be based around dealing with this data, as it is what the user will be interacting with and is the main premise for the game.

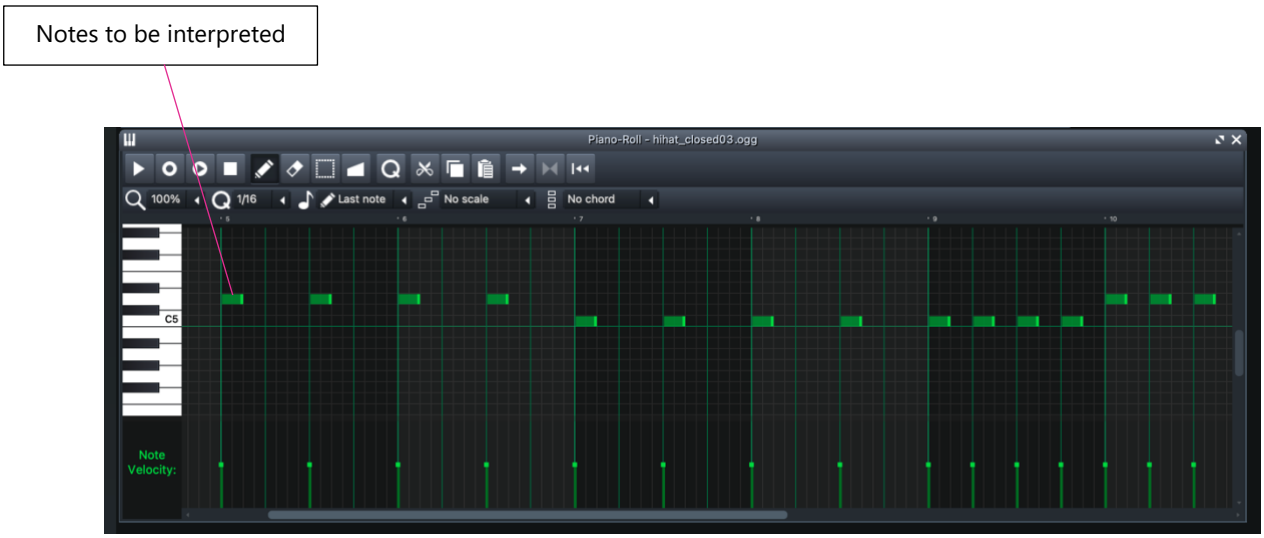
One solution would have been stored the notes in game by either detecting key areas of the song mathematically from the song file and generating notes by interpreting wavelengths, but this solution would not be practical, and would produce less purposeful, as there would be no way to edit the stages. I chose to use MIDI files to store this stage data and use predetermined stage layouts to reduce run time calculations.

MIDI is a file format that standardizes digital music communication and stores information about individual tracks that compose the song, the bpm and length. Rather than being made of a single continuous audio source, Midi files are like Photoshop files in that they store the layers that make up the song. Applying this file format, the song audio itself would not do anything, but it can be used as a spreadsheet for holding the stage data. Through this method, stages in the game are stored as modular files that can be read from, modified, and displayed as needed. By previously composing a midi file with drum notes in time with a song's beats in a music producing software, the midi file can be interpreted by the DryWetMidi Library as the start time and pitch of the notes (from the chromatic scale). The player will not need to interact with these files as the game will have a stage selection menu, and the script will only need to interact with them when loading the stages (reading the file). Because of its utility in editing, representing and storing notes, I will be using the MIDI file format.

I will be making these MIDI stages using LMMS for macOS – A digital audio workstation where sounds can be synthesised and arranged on a MIDI keyboard. Since each stage runs on top of a song, I started by importing the song to use as a guide, however only the synthesised track will be saved in the midi file. For the song, the bpm and time signature need to be set so that notes can be synthesises to the correct tempo.



In a separate layer I made a drum beat track, using a simple high hat hit, played on a midi keyboard. Only two notes were needed, the C note and the D note, representing the top and bottom lanes. Setting the song's bpm, I could place notes at up to 1/16 beat intervals. The drumbeats aligned with the indicated lines. For my first stage, I placed the notes at regular intervals making the stage easy to test and debug and easy for play testers to try. I designed the stage with new rhythm game players in mind. Making sure the track is the full length of the song, I saved just the individual track as a MIDI file into the Streaming Assets folder of my game project.



Whilst there Is no need method for testing the asset, it will be added to the project and tested as a part of it.

Reading Notes - Feature 1.2

This feature should allow any midi file to read to an array of notes. This is justified as the basis for the game and will be used to import generated levels.

To begin, I initialised the *conductor* class with public variables. In general, the public static variables will be assigned later in the class and the public variables are assigned values at runtime, set using unity's editor before the game is run.

```
9   public class Conductor : MonoBehaviour
10  {
        18 references
11      public static Conductor Instance;
        3 references
12      public static MidiFile midiFile;
        1 reference
13      public Lane[] lanes;
        3 references
```

lanes will reference the two-lane classes

The *Start* method is a built-in unity function that executes before the first frame that the conductor class runs. From here the *readFromFile* function is called, so that it is the first function that executes. Whilst doing this, the if statement checks if the file can be found. If an incorrect file name is passed to the class, then an error is returned.

Note on returning errors: Since the game will be distributed for players to play on their own devices, they won't be able to see any prints or errors logged. Everything in the unity development engine will stay inside the engine and so these prints wouldn't be displayed to the user. They are still useful however for debugging so I will keep them in until the final version of the game, found at the end of project. As a replacement for these, errors, if there is no way to fix them, the game will crash. I will need to do plenty of tests to make sure this doesn't happen.

```
// Start is called before the first frame update
0 references
void Start()
{
    Instance = this;

    if (!ReadFromFile()){
        print("Stage data cannot be loaded: file not found");
    }
}
```

The ReadFromFile function returns a Boolean value, corresponding to if a file was found or not. This algorithm is adapted slightly from the originally proposed one. Instead of searching for the file path, the file path searching is handled by unity's *Appliction.steamingAssestPath* class attribute that returns a consistent file path for any device the game is running on. When testing this function, I will be checking this works when the application is built. If found, the midi file can be opened as the correct file format, and the next function is called. If not found, then the function returns false. This is justified because the *fileName* needs to be validated so that

```
// File name specified beforehand by the stageManager class
1 reference
private bool ReadFromFile()
{
    // check if midi file exists in the streaming assets folder
    if (File.Exists(Application.streamingAssetsPath + "/" + fileName + ".midi")){
        // read from midi file
        midiFile = MidiFile.Read(Application.streamingAssetsPath + "/" + fileName + ".midi");
        GetDataFromMidi();
        return true;
    }
    // if midi file does not exist
    else{
        print("File not found");
        return false;
    }
}
```

The *GetDataFromMidi* function uses public variables from the conductor class defined previously, such as the *midiFile*. The notes array is copied to the *array* array, which is defined as the same type and length. Then a copy of each array is passed to each of the two lane classes using a *foreach* loop. Lastly the *audioSource* is played after waiting an adjustable length of time that helps sync to the note times. *audioSource* contains the song that the notes are synced to.

```
// Using the Dry Wet Midi library
1 reference
public void GetDataFromMidi()
{
    // notes read and copied to array
    var notes = midiFile.GetNotes();
    var array = new Melanchall.DryWetMidi.Interaction.Note[notes.Count];
    notes.CopyTo(array, 0);

    // passes array to lane classes
    foreach (var lane in lanes) lane.SetTimeStamps(array);

    // calls start song after waiting for songDelay seconds
    Invoke(nameof(StartSong), songDelayInSeconds);
}

1 reference
public void StartSong()
{
    audioSource.Play();
}
```

Notes are then finally saved into an array in the lane class called *timeStamps* using the *SetTimeStamps* algorithm.

The *SetTimeStamps* algorithm is in the Lane class, set as public to be accessed from the conductor. This is because the *timeStamps* lists where the notes will be indexed are initialised in the Lane class and are spawned from the Lane class. *timeStamps* only holds the notes for a single lane. Each lane class will need to filter which notes it saves and this is done by comparing each note to the *noteRestriction* variable, which is set at runtime for the class. Matching notes type from the read midi file then have their times taken from the input list and these are then assigned to the lanes own *TimeStamps* list after being converted to metric time (hours, minutes, seconds, milliseconds), and then to just seconds. I needed *TimeStamps* to be in seconds to be able to compare them to the output from the GetAudioSourceTime() function.

```
public void SetTimeStamps(Melanchall.DryWetMidi.Interaction.Note[] array)
{
    foreach (var note in array)
    {
        // filters notes
        if (note.NoteName == noteRestriction)
        {
            // converts to metric time
            var metricTimeSpan = TimeConverter.ConvertTo<MetricTimeSpan>(note.Time, Conductor.midiFile.GetTempoMap());

            // convert to seconds
            var secondsTimeSpan = (double)metricTimeSpan.Minutes * 60f + metricTimeSpan.Seconds + (double)metricTimeSpan.Milliseconds / 1000f;

            // adds to timeStamps list
            timeStamps.Add(secondsTimeSpan);
        }
    }
}
```

Testing feature 1.2

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
3.1	Check if stage data is read from Midi file.	filePath = valid midi file in streaming assets	Stage values are read from the Midi, into a game readable format, and the stage can be loaded, returns True	Figure 1.2.1	Pass
3.2	Check if stage data is read from Midi file when the file exists, but the file path is incorrect.	FilePath = valid midi file name, streaming assets not specified	Stage Midi file not found and the same name format as input is searched for. If found, file path is saved and read, returns True.	Figure 1.2.1	Pass Note: Unity has a class attribute for always locating the streaming assets path on any device, so this always works.
3.3	Check if stage data is read from Midi file when the file does not exist, and there is no file path.	FilePath = empty file path	Stage Midi file not found and the same name format as input is searched for. If not found, return false, and stage is not loaded.	Figure 1.2.2	Pass

5.1	Test noteRestrictions against note Arrays, for each combination test if note is added to its corresponding lane.	*noteRestriction*, note List	If *noteRestriction* is the same as pitch class of the note from the Note List, then the note's time should be added to the note List.	Figure 1.2.1	Pass
5.2	Check if timestamps array is populated with note times corresponding to various note List inputs.	Note List, *timeStamps*	Note list is fully converted to time stamps for each lane.	Figure 1.2.1	Failed – The list for Lane 1 populated incorrectly

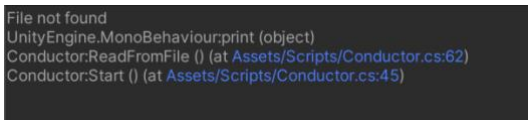


Figure 1.2.2 – File not found and stage is not loaded. This is important for catching any errors later in development.

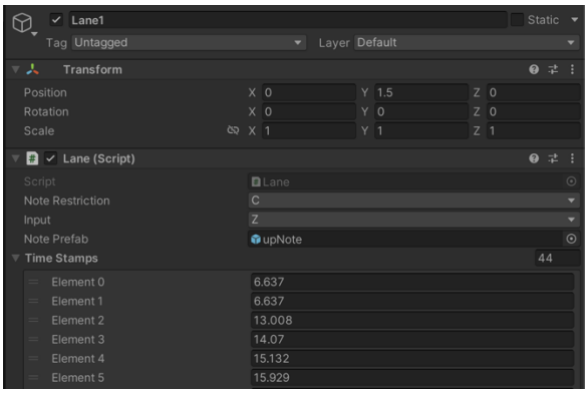


Figure 1.2.1 – When a valid file path is inputted, and the note restriction was correctly set to a 'C', list was populated with every top lane notes Elements 0 and 1 in the *timeStamps* list are the same after the midi file was read.

Test 5.2 Results

In test 5.2, I saw that the first element in the list had duplicated. This error would cause two notes to spawn at once in. The main issue is that any note value can be added to the list, no matter what its timing position was.

A possible fix to this could be snapping each note as it is added to either a beat or a half beat. Since beats can be calculated as a metric in terms of beats per second using the song's tempo, I could add this to be displayed and at this stage in development, it was early enough for me to add a metronome display, that could count beats:

```
public float songBeat;
1 reference
public float bpm;
2 references
public float secondsPerBeat
{
    get
    {
        return 60f / bpm;
    }
}
```

However, this method would be unnecessarily complicated, and although would validate all timings, there were more efficient methods of fixing the error. More importantly, this method was still limited to timing accurately if the song was already synced to its beats.

```
    void Update()
    {
        Instance.beatFrame = false;
        // counts beats during the song
        if (((GetAudioSourceTime() - Instance.songDelayInSeconds) / Instance.secondsPerBeat) - Instance.songBeat > 1){
            Instance.beatFrame = true;
            Instance.songBeat = Convert.ToInt32((GetAudioSourceTime() - Instance.songDelayInSeconds) / Instance.secondsPerBeat);
            beatText.text = Instance.songBeat.ToString();
        }
    }
}
```

I added an extra validation rule in the *SetTimeStamps* method by creating an if loop that checks if the value is already in the list. If not, then the time span is added, but if the item is a duplicate then it is rejected. By adding validation to this part of the code, the criteria that the game should have notes that spawn in every time a level is loaded is met, and the user won't experience an invisible note being missed, as was happening.

```
            // adds to list if not a duplicate
            if (!(timeStamps.Contains(secondsTimeSpan))){
                timeStamps.Add(secondsTimeSpan);
            }
```

Changed code after test

5.2	Check if timestamps array is populated with note times corresponding to various note List inputs.	Note List, *timeStamps*	Note list is fully converted to time stamps for each lane, without any duplicates	Figure 1.2.3	Pass

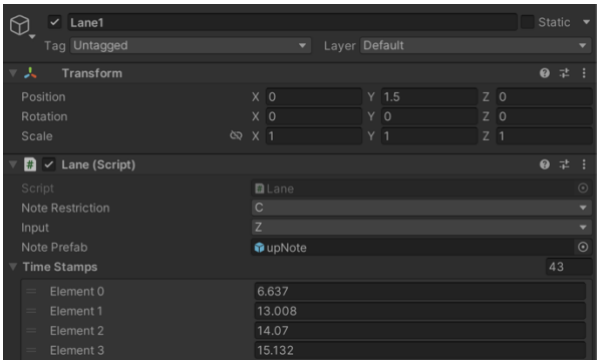
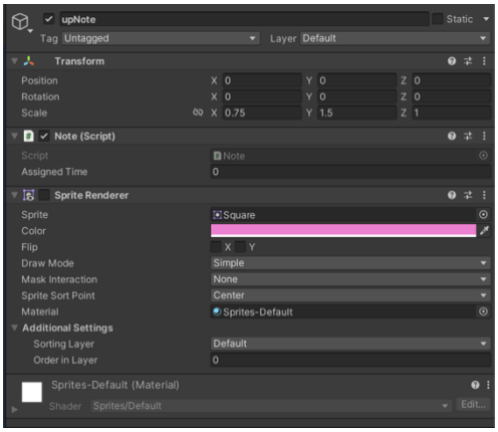
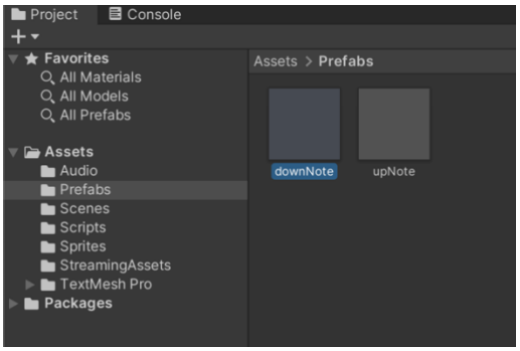


Figure 1.2.3 – The list contains no duplicates.

Spawning Notes - Feature 1.3

Once notes have been read and saved, they can be displayed in time with the audio. Notes are generated by calling the Note class, and using a note prefab. A prefab in Unity is pre-complete game object, acting like a re-useable template. New instances of the prefab can be created in the scene, and all the copies remain in sync. Later the rectangle prefabs will be replaced with drawn assets that will make the game more appealing and user friendly. In the images below I have created a separate prefab for both types of notes - for the top and bottom lanes.





These note prefabs will be set to spawn on the far-right hand side of the scene, just out of the players vision.

Notes are spawned in the Lane class by considering a *spawnIndex*. The *spawnIndex* is a variable that initialises at zero at the start of the stage, and will increment by one after each note is spawned. This is an implementation of a list, where the spawnIndex acts as a pointer. Using this, we can check for the time the note needs to appear by comparing the current song time to the time of the next note that needs to be spawned in the *timeStamps* array, and subtract the time it would take for the note to reach the hit position.

This effectively allows the note spawns to move left and right, so long as the same variable for the distance is used throughout the program, to avoid the time de-syncing with the distance. Crucially, this allows for an extra variable to be added, song, input delay.

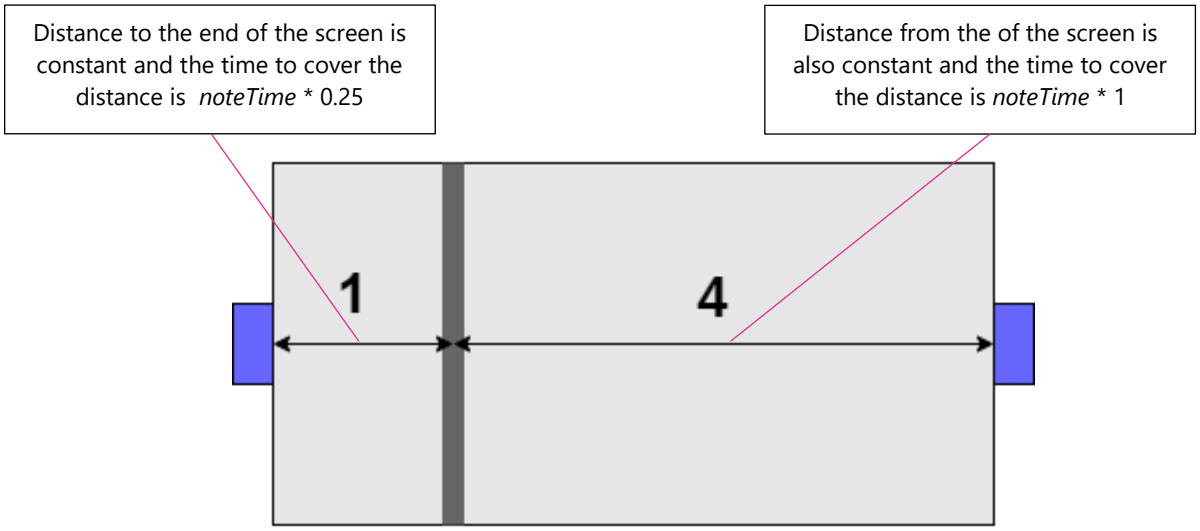
```
public class Lane : MonoBehaviour
{
    public Melanchall.DryWetMidi.MusicTheory.NoteName noteRestriction;
    public KeyCode input;
    public GameObject notePrefab;
    List<Note> notes = new List<Note>();
    public List<double> timeStamps = new List<double>();

    int spawnIndex = 0;
    int inputIndex = 0;
```

```
// Update is called once per frame
void Update()
{
    // spawn notes
    if (spawnIndex < timeStamps.Count)
    {
        if (Conductor.GetAudioSourceTime() >= timeStamps[spawnIndex] - Conductor.Instance.noteTime)
        {
            var note = Instantiate(notePrefab, transform);
            notes.Add(note.GetComponent<Note>());
            note.GetComponent<Note>().assignedTime = (float)timeStamps[spawnIndex];
            spawnIndex++;
        }
    }
```

A new Note object is instantiated and the time it was instantiated is saved. This is useful because the note position will be calculated as a measure of how much time has passed since the note appeared. *T* is a ratio between how long the note has been on the screen and how long its lifespan is. The life span is calculated from the *noteTime,* which determines the time from the spawn to the hit area, so this is multiplied by 1.25 to cover the rest of the screen.

Note spawn and respawn just outside of the screen. The 1:4 ratio allows enough time for the user to react to the notes as they approach from the right. The total travel time is

1 noteTime* + 0.25 * *noteTime*
= 1.25 * *noteTime*.



Since t is a ratio, we know the note has finished its movement if the value goes above 1, so the note can be deleted. Then every frame the game updates, the game linearly transforms the vector position of the object, between the universal spawn X position and despawn X position.

This method was needed because the notes had to move smoothly, and continuously, and the speed had to be adjustable. From the tests, it is clear that this was successful as *noteTime* can be changed to adjust the speed, because distance is constant, so speed scales with time.

```
public class Note : MonoBehaviour
{
    double timeInstantiated;
    public float assignedTime;

    void Start()
    {
        timeInstantiated = Conductor.GetAudioSourceTime();
    }

    // Update is called once per frame
    void Update()
    {
        double timeSinceInstantiated = Conductor.GetAudioSourceTime() - timeInstantiated;
        float t = (float)(timeSinceInstantiated / (Conductor.Instance.noteTime * 1.25));    // noteTime * 1.25 represents total time to travel from start to end

        GetComponent<SpriteRenderer>().enabled = true;

        if (t > 1)
        {
            Destroy(gameObject);
        }
        else
        {
            transform.localPosition = Vector3.Lerp(Vector3.right * Conductor.Instance.noteSpawnX, Vector3.right * Conductor.Instance.noteDespawnX, t);
            GetComponent<SpriteRenderer>().enabled = true;
        }
    }
}
```

Validation

The *noteTime* variable needs to be validated in the conductor class as it controls logic for spawning notes in the Note class, and will later be able to be changed by the user from the Conductor class. To validate the variable, the upper and lower bounds are checked in the start function of the class, so that the variable is validate before other code executes. To avoid divisions by 0 and having the notes move too fast, the lowest value will be set to 0.1 and the highest and slowest value to 5.

```
    // Start is called before the first frame update
    0 references
    void Start()
    {
        // validate NoteTime
        if (noteTime < 0.1 || noteTime > 5)
        {
            print("Invalid NoteTime");
            noteTime = 1; // reset to default value
        }
```

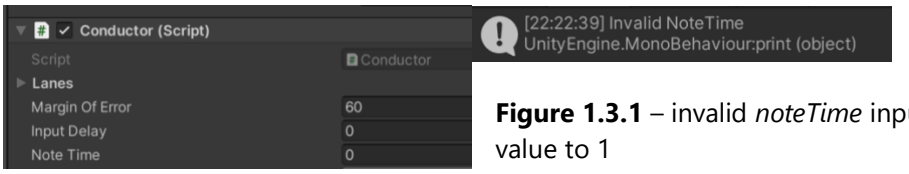


Figure 1.3.1 – invalid *noteTime* input results in an error and sets value to 1

Testing Feature 1.3

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
5.3	Check if all notes from timestamps list are spawned in the lane.	SpawnIndex, timeStamps	First note through to the end note appear on screen, and at the end of the stage *spawnIndex* = *timeStamps* length	Figure 1.3.2	Pass
5.4	Vary note travel time to test note spawn position.	NoteTime = 0.5, 1.5, 5, Note spawn	Notes should spawn in the same place every time no matter the note time.	Figure 1.3.1, Figure 1.3.2	Pass

	Vary between 0.1 and 5, note travel time cannot be less than 0.1	position in playback editor, timeStamps			
4.1	Instantiate note from an array and compare with spawn time from time stamp array.	timeInstantiated, timeStamps	TimeInstanciated is the same as time stamp time and note is instantiated at the correct time.	n/a	Pass
4.2	Check timeSinceInstantiated updates with each frame	timeSinceInstantiated	Time increases, with the instance the note was created being 0 seconds.	n/a	Pass
4.3	Test note movement by monitoring RelativePosition and varying EndX (EndX is typically constant)	RelativePosition, EndX	The further EndX, the faster the notes should move. Constant EndX should be decided.	Figure 1.3.4	Pass- *endX* set to -10, just outside of the camera's view.
4.4	Test note movement by monitoring RelativePosition and varying noteSpeed	RelativePosition, noteTravelTime	As noteTravelTime decreases, notes will travel faster, and vice versa. Note positions updates linearly.	Figure 1.3.3	Pass
4.5	Test note movement by monitoring RelativePosition and varying spawn X (spawnX is typically constant)	RelativePosition, SpawnX	Note positions updates linearly with each spawn X position.	Figure 1.3.4	Pass - *spawnX* set to 10, just outside of the camera's view.
4.6	Input relative position values of 0, 1 and 2 inside the UpdatePosition function for a single Note.	Relative position = 0, 1, 2	While loop entered for relative position 0, 1. While loop exited for relative position 2 and note self-destructs	n/a	Pass

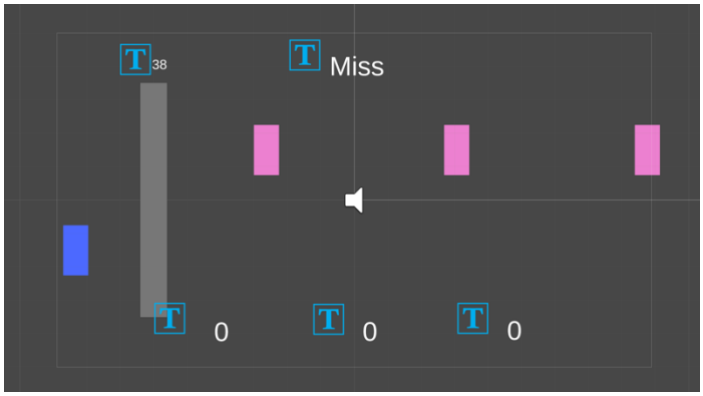


Figure 1.3.2 – Notes spawning at *spawnX* and ending at *endX*

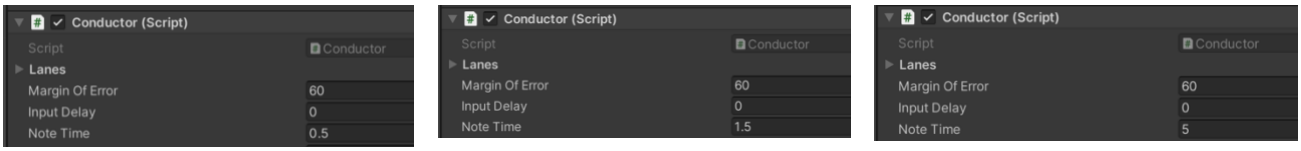


Figure 1.3.3 – Varying *noteTime* within bounds resulted in constant spawn and end positions, and relative position behaving linearly.

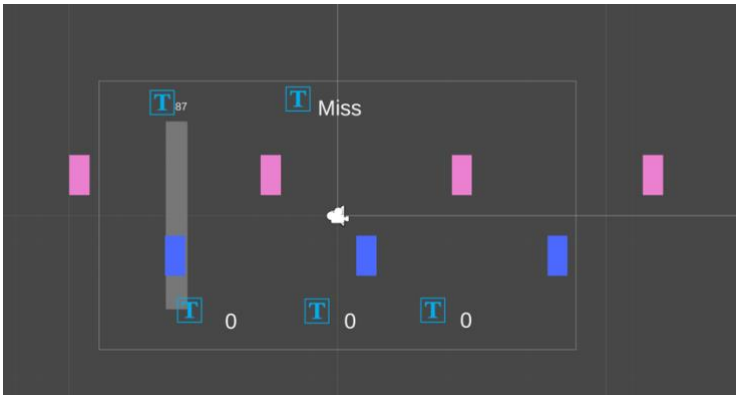


Figure 1.3.4 – Varying *spawnX* and *endX* within bounds resulted in a change in speed, but with notes moving at correct speeds.

Registering Inputs - Feature 1.4

One function of the lane class is handling user inputs. Because both lanes are instances of the same class, each lane object handles user inputs individual, each considering at most 1 key press a frame. The keyboard presses are received by *input.GetKeyDown(key)*. The code shown is inside the class's built in update function, which is called once per frame.

Before the game checks for inputs, the lane checks the *inputIndex*. Which holds how many notes have been either hit or missed. *inputIndex* is compared to length of the notes list for that lane, so that once the stage is finished, the user cannot input any more presses and the stage ends.

.

```
// register inputs
if (inputIndex < timeStamps.Count)
{
    double timeStamp = timeStamps[inputIndex]; // hit time of incoming note
    double marginOfError = Conductor.Instance.marginOfError;
    double audioTime = Conductor.GetAudioSourceTime() - (Conductor.Instance.inputDelay / 1000.0); // input delay in ms
```

On every frame that the *input* key for the lane is pressed, an input registers and the object enters an if loop, where the input is dealt with. The key that the user uses for each lane can then be set at run time. I set the top lane to be the Z key and the bottom lane to the X key.

There are several places a note could be when the input key is pressed so it is important to check all of them individually. The position of the note is calculated by the *audioTime* variable, which has the input delay constant subtracted from it, moving the notes position left or right accordingly. This can be visualised as a flat, horizontal line scrolling right, like an audio track would. The notes are numbers on this line, representing the

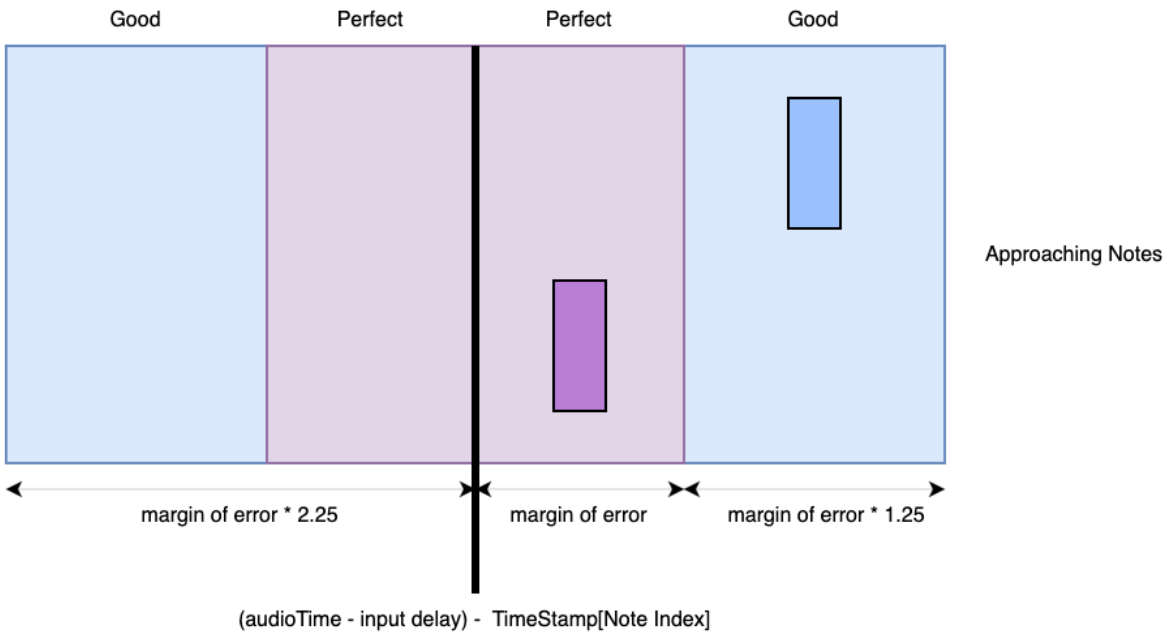
beats of the song. For the note to be in the right position to be hit, it has to scroll far enough to line up with a marker, but not scroll past it.

The width of the area is the *marginoferror* variable, and the distance from the hit marker (the timestamp of the current note, gotten from the *timeStamps* list using the *inputIndex)* is obtained using *Math.Abs(audioTime - timeStamp).* The user has 2 times the margin of error of time to hit the note perfectly. Values of the constant *marginOfError,* and its scale factors for the good and miss regions will affect difficultly, and my values chosen will be justified in testing and with stakeholder feedback.

During development, I found that using a *marginOfError* of 50ms worked well as a timing for the game. Overall, combined with widths for the perfect good and miss timings this gives the player:

Perfect: 1x *marginOfError* = 50ms, for a total of 100ms of leniency
Good: 1.25x *marginOfError* = 62.5ms, for a total of 125ms of leniency, on top of the perfect timing
Miss: 0.5x *marginOfError* = 25ms, = 50ms of catching leniency

These values worked well to create an accessible rhythm game with enough difficulty.



If the note is within the 'perfect' margin of error, the note is destroyed immediately, input index increments and the user receive a perfect hit, to be calculated later. This frees up the user to hit the next note on the consecutive frame. If the note is within the 'good' margin of error, the same happens except the user receives a good hit. The note can also be in the miss area. The miss area was added to prevent notes being hit by pressing keys randomly and rapidly, by catching any inputs that come within the actual note hit windows.

```
// on a key press
if (Input.GetKeyDown(input))
{
    // conditions to hit a perfect
    if (Math.Abs(audioTime - timeStamp) < marginOfError)
    {
        Hit(0);
        print($"Hit {inputIndex} note with PERFECT timing and {audioTime - timeStamp} delay");
        Destroy(notes[inputIndex].gameObject);
        inputIndex++;
    }

    // outside of perfect range (good range)
    else if (Math.Abs(audioTime - timeStamp) < marginOfError*1.5)
    {
        Hit(1);
        print($"Hit {inputIndex} note with GOOD timing and {audioTime - timeStamp} delay");
        Destroy(notes[inputIndex].gameObject);
        inputIndex++;
    }

    //outside of good range (miss range)
    else if (Math.Abs(audioTime - timeStamp) < marginOfError*2)
    {
        Miss();
        print($"Missed {inputIndex} note with {audioTime - timeStamp} delay");
        Destroy(notes[inputIndex].gameObject);
        inputIndex++;
    }

}
```

The user may also not press any inputs and miss a note completely. The note exits the latest possible hit area, and so the note is immediately missed. If the note is outside the miss area then the input is ignored. This is so notes far enough away cannot be interacted with. This means that notes will disappear off the screen and despawn themselves. The boundary for this happening is

```
// outside of miss range and no input key pressed
if (timeStamp + marginOfError*2 <= audioTime)
{
    Miss();
    print($"Missed {inputIndex} note: {audioTime - timeStamp} delay");
    inputIndex++;
}
```

Successful hits and misses call a private method that encapsulates all the action that should happen on the events. In both cases the score manager class is called. In the hit method, the accuracy rating is passed in as an argument and will control the score give to the player. The rating is passed as an integer, either a 0 or a 1.

```
    // calls scoreManager
    private void Hit(int timingRating)
    {
        ScoreManager.Hit(timingRating);
    }

    // calls scoreManager
    private void Miss()
    {
        ScoreManager.Miss();
    }
}
```

Testing feature 1.4

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
5.5	Input a 'perfect' hit exactly as the measure of the score increments (when a beat occurs in the song).	Player input, marginOfError	Note is destroyed exactly on the hit indicator, perfect score received, hit method called.	Figure 1.4.1	Pass
5.6	Input a delayed or early 'perfect' hit, by the margin of error	Player input, marginOfError	Note is destroyed around the hit indicator, perfect score received, hit method called.	Figure 1.4.2	Pass
5.7	Input an early or late hit in the 'good' area	Player input, marginOfError	Note is destroyed further from the hit indicator and not exactly on the beat. good score is received, hit method called.	Figure 1.4.3	Pass
5.8	Input a very early or very late hit to get a miss hit.	Player input, marginOfError	Note is destroyed but registers as a miss, player receives a miss, and the miss method is called.	Figure 1.4.4	Pass
5.9	Miss a note by pressing no inputs until it moves outside of miss area	Player input, marginOfError	Note continues moving until the end of the screen and is destroyed, player receives a miss and the miss method is called.	Figure 1.4.5	Pass

Figure 1.4.1 - hitting on the change of beats gives a perfect hit, delay between the time of user input and the note's time stamp in seconds is roughly equal to 0ms

Figure 1.4.2 – Hitting early just within the margin of error timing gives a perfect hit, delay between the time of user input and the note's time stamp in seconds is roughly equal to 50ms, and margin of error was set to 50.

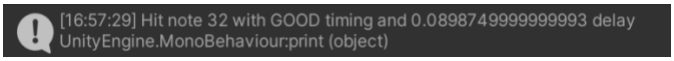


Figure 1.4.3 – Hitting late after the margin of error timing gives a GOOD hit, delay between the time of user input and the note's time stamp in seconds is roughly equal to 90ms, and margin of error was set to 50.

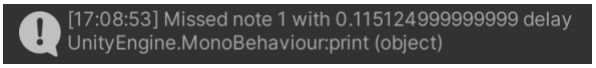


Figure 1.4.4 – Hitting just late after the 2.25x margin of error timing gives a miss, delay between the time of user input and the note's time stamp in seconds is roughly equal to 115ms, and margin of error *2.25 was equal to 112.5.

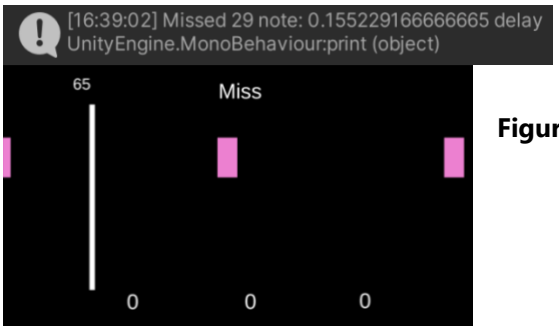


Figure 1.4.5 – Note is missed if no input is pressed

Counting Score - Feature 1.5

Another important part of the game, outlined in the success criteria is a measure of score. To code the score into the game, I created a separate class, *ScoreManager*. The class would keep track of any variables relating to the users performance, and to any calculations needed.

```
public class ScoreManager : MonoBehaviour
{
    public static ScoreManager Instance;
    public AudioSource hitSFX;
    public AudioSource missSFX;
    public TMPro.TextMeshPro comboText;
    public TMPro.TextMeshPro scoreText;
    public TMPro.TextMeshPro timingText;
    public TMPro.TextMeshPro accuracyText;
```

I used unity text variables to display the text for the UI elements and the hit ratings as they were appearing. The variables are declared are runtime.

```
void Start()
{
    Instance = this;
    combo = 0;
    score = 0;
    accuracy = 100;
    notesPassed = 0;
    notesPerfect = 0;
    notesGood = 0;
    notesMissed = 0;
}
```

At the start of each stage, when an instance of a score manager will be called, variables are initialized at zero. Combo, score and accuracy will be calculated and displayed, whilst *notesPassed*, *notesGood*, *NotesMissed* and *NotesPerfect* keep track of the users exact inputs.

After each hit, the scores manger is called. The *timingRating* argument, determines how much score is given to the user. For a Perfect, hit 200 points are assigned and for a good hit 100 points.

```
36      public static void Hit(int timingRating)
37      {
38          combo += 1;
39          notesPassed += 1;
40
41          if (timingRating == 0){
42              notesPerfect += 1;
43              timing = "Perfect";
44              score += combo*30;
45          }
46
47          else if (timingRating == 1){
48              notesGood += 1;
49              timing = "Good";
50              score += combo*15;
51          }
52
```

Also inside the hit method, the accuracy is calculated using the users perfect, good and missed hits. Out of all total hits, perfect hits give the most accuracy, whilst a good hit gives 66.66%. Two thirds is used here to have the accuracy be proportional to the hit area. The good timing is 50% larger than the perfect area, dividing the original area by this gives a ratio of 1:2, making the good hit worth proportionality less than the perfect hit

Because accuracy is a float and needs to be displayed, it is rounded to 2 decimal places. Since accuracy is re-calculated each time, this doesn't result in any rounding inaccuracies, as it is only used in the UI.

```
// method for calculating accuracy, should be proprtianl to hit areas
accuracy = (notesPerfect + notesGood*0.66666) / notesPassed * 100;
accuracy = Math.Round(accuracy, 2);
print($"Accuracy: {accuracy}");
```

```
60      public static void Miss()
61      {
62          combo = 0;
63          notesPassed += 1;
64          notesMissed += 1;
65          timing = "Miss";
66
```

The miss method responds to any misses by resetting the combo variable to 0 and displaying miss text.

Testing feature 1.5

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
6.2	Check combo value if note was successfully hit.	Combo, input hit	Combo increments	Figure 1.5.1	Pass
6.3	Check combo value if note was missed.	Combo, input miss	Combo resets to 0	Figure 1.5.2	Pass
6.4	Check score value if note was successfully hit.	Score, input hit	Score increments	Figure 1.5.1	Pass
6.5	Check score value if note was missed.	Score, input miss	Score does not change	Figure 1.5.2	Pass
6.6	Check accuracy value if note was	Accuracy, input hit	Accuracy percentage re-calculated, and increases	Figure 1.5.1	Pass – when no notes were

	successfully hit.		(or stays the same if 100%)		missed at all – accuracy remained at 100% until the end of the stage.
6.7	Check accuracy value if note was missed.	Accuracy, input miss	Accuracy percentage re-calculated, and decreases	Figure 1.5.2, Figure 1.5.3	Fail – accuracy value does not update after a missed note

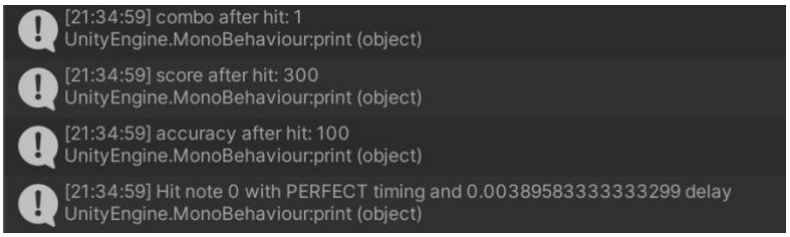


Figure 1.5.1 – Score, combo and accuracy variables behave as expected after a hit

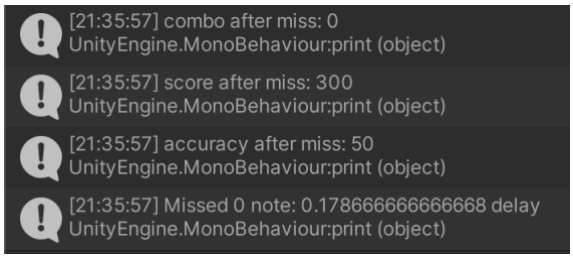
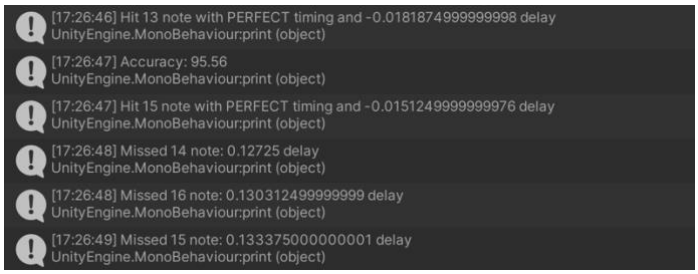


Figure 1.5.2 – Score, combo and accuracy variables behave as expected after a miss

Figure 1.5.3 – Display accuracy unchanged after miss

Test 6.7 Results

From test 6.7, I identified the issue as being the accuracy value not updating not frequently enough. Debugging the error, I set the accuracy to print just before the note hit or miss information would print. In the debug log screenshot, the perfect hit gives a new accuracy of 95.56%, however after that, 3 notes are missed and there are no new accuracy values. The result is an accuracy that only updates after a successful hit, but is still stored correctly, after a miss, since the line of code for recording a passed note is still keeping track of misses.



Accuracy not printing after miss

```
public static void Hit(int timingRating)
{
    combo += 1;
    notesPassed += 1;

    if (timingRating == 0){ // for a perfect
        notesPerfect += 1;
        timing = "Perfect";
        score += combo*300; // increase score
    }

    else if (timingRating == 1){ // for a good
        notesGood += 1;
        timing = "Good";
        score += combo*200; // increase score
    }

    Instance.calculateAcc();
    Instance.hitSFX.Play();
}

public static void Miss()
{
    combo = 0; // reset combo
    notesPassed += 1;
    notesMissed += 1;
    timing = "Miss";

    Instance.calculateAcc();
    Instance.missSFX.Play();
}

// method for calculating accuracy proprtianlly to hit areas
private void calculateAcc()
{
    accuracy = (notesPerfect + notesGood*0.66666666) / notesPassed * 100;
    accuracy = Math.Round(accuracy, 2);
}
```

To fix the issue, a new function for calculating accuracy was added in the *ScoreManager* class. Moving the accuracy calculation into this function, made it possible to call it both from the miss and hit method, and because the function uses public variables from the class, this fixed the error by updated the accuracy every frame an action happens. This solution also result in more maintainable code.

Improved code with calculateAccuracy() method

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
6.7	Check accuracy value if note was missed.	Accuracy, input miss	Accuracy percentage re-calculated, and decreases	Figure 1.5.5	Pass

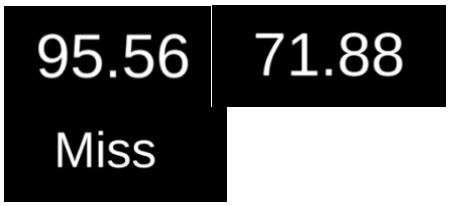


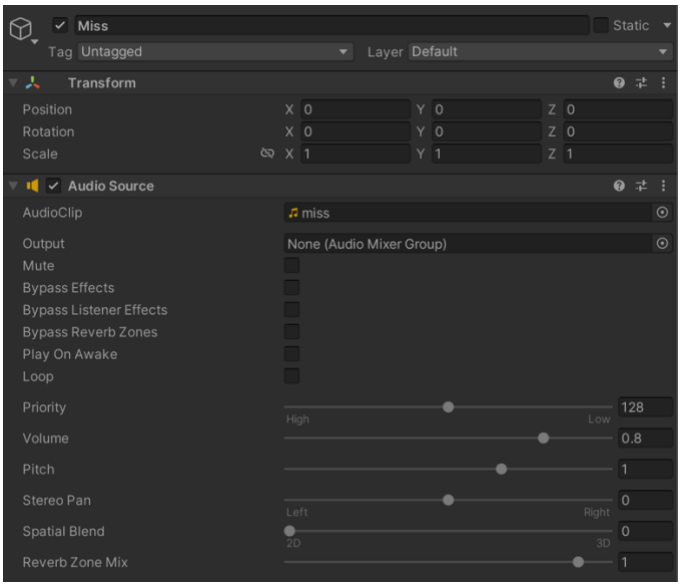
Figure 1.5.5 Accuracy updates after miss

Sound effects - Feature 1.6

When choosing sound effects, I made sure to refer to the original success criteria of the project, to choose a sound effect that would be engaging to a user and give them an audible response to the gameplay on screen,

as well as fitting with the stakeholders preferred style. One for hitting a note and another sound effect for missing.

Despite being inside the score manger class, both audio sources are separate unity game objects, and can be controlled in the scene individually, as shown in the unity editor for the miss sound audio player.



Implementing the sound effects was done with unity's audio source variables. The variables act as constants and have the file path of the sound effects file set at the start of the score manager class.

```
public static void Miss()
{
    combo = 0;
    notesPassed += 1;
    notesMissed += 1;
    timing = "Miss";

    Instance.missSFX.Play();
}
```

Sound effect sources are played with the *Play()* function.

Sound effect volume can is adjusted by accessing the .*volume* attribute of the game object, and so I created an extra variable to control this, initializing it with a float value of 0.8, being the equivalent to 80%. Future iterations will build on this setting.

```
public class ScoreManager : MonoBehaviour
{
    public static ScoreManager Instance;
    public AudioSource hitSFX;
    public AudioSource missSFX;

    public float volumeSFX;
```

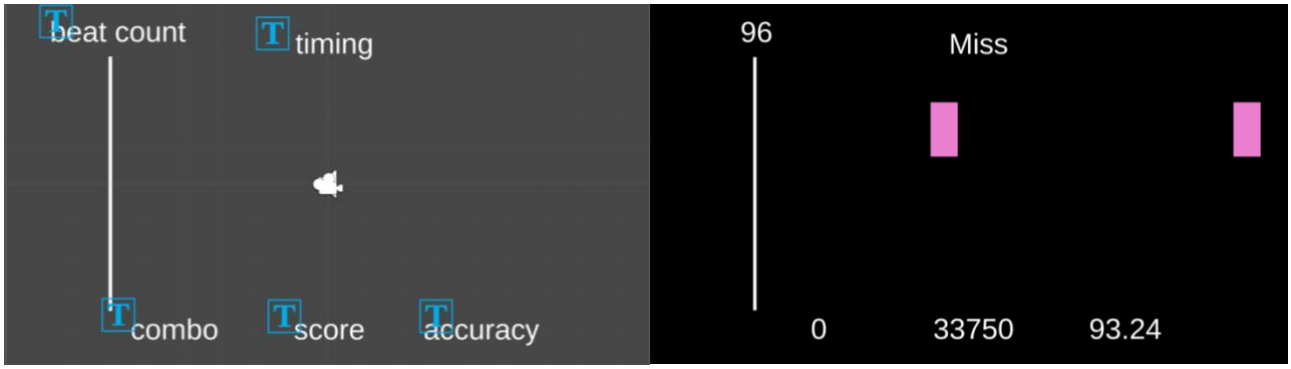
volumeSFX will control both effects, in a single variable

Testing Feature 1.5

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
6.10	Check if sound effects plays after note hit	Player input, AudioSource	A hit should be followed by the hit sound effect.	n/a	Pass – expected result

6.10	Check if sound effects plays after note misses.	Player input, AudioSource	A miss should be followed by the miss sound effect.	n/a	Pass – expected result

Iteration 1 Evaluation



Above I have shown the game after the first prototype, as seen in unity to the left, and as seen by the player during runtime to the right. After running the game, the stage immediately begins and the user can hit the incoming notes. Each of the text variables updates in real time for the player to see their statistics.

Iteration 1 has accomplished the following success criteria, and each feature implemented is justified:

- First stage created - Stage is developed in a rhythm game fashion, with gameplay objects following beats in a song.
- Hit objects created – note objects spawned from the midi file containing the first stage. Notes spawn in accordance with beat timings.
- 2 object lanes - notes are distinguished between two lane objects, which reference two separate lists of notes for users to hit.
- Hit area – user can interact with the notes by tapping keys, and there is a visual reference for when to hit the notes.
- Timing bands – How accurate the hit was can be seen from the timing band the hit was given. Perfect being the most accurate, good being slightly late or early and miss being too far off.
- Score – Score increments in accordance to the accuracy band of the hit.
- Accuracy – User can see how close to the beat their hits are.
- Combo – The number of notes hit in a row
- Hit and miss effects – sound effects respond to user inputs

The following success criteria are not fully complete / features have not been fully developed/ need improvement in the next iteration:

- UI displays
    - Layout of combo, score, accuracy and timing
    - Statistics fonts
- Sprites
    - Note sprites
    - Hit line sprite

Stake holder feedback

Each of my two stake holders played the complete stage of the game and I gathered information on the current features in the prototype. Since the prototype was laying the groundwork for the game, I mostly asked the user about the basic features of the game, and less about subject elements, that will be improved and evaluated later in the project?

Does the timing for hitting a note feel fair?
Farhan: In my opinion the timing is strict but not impossible. All the notes were synced to the beat, but some of the miss timings were very close.
Magda: Yes, hitting the notes feels fun and like a challenge, I think the timing is good as it is. I knew whenever I missed a note that I had hit too late or early.

Should the ability to miss a note by hitting too early should be in the game?
Farhan: Yes, although maybe make the 'good' area bigger so that the player doesn't miss as much or make it harder to miss slightly. At times the miss area is too big.
Magda: I think the feature should be in the game. It avoids people spamming

Is the perfect and good timing large enough?
Farhan: Yes, it is lenient.
Magda: I like the perfect timing; it doesn't feel too hard, and the player is rewarded for hitting on the beat. The good timing also feels good.

What do you think of the sound effects?
Farhan: Being able to change the volume would be nice, otherwise they are perfect.
Magda: good, but maybe there could be a separate sound effect for a good hit. I like how the miss sounds are discouraging.

Where do you think the position of the hit line should be?
Farhan: I think its fine where it is, however it would be better if it was thicker, to better indicate where to hit the notes.
Magda: I like where it is at the moment.

What do you think of the current stage?
Farhan: The sections are quite repetitive, but fun to play.
Magda: good for beginners and music is very suitable, would be a good tutorial stage.

How do you feel about the area for hitting a note at higher note speed being consistent?
Farhan: It's a useful feature for users who want to change the note speed they play at.
Magda: It's good; it ensures consistency across stages too.

How much score should the user get per hit?
Farhan: If the good area was made bigger, then more score should be awarded for it too.
Magda: Maybe increase the score to be in the hundreds instead of in the tens.

Are the timing indicators clear?
Farhan: Yes, although they could be a bit clearer.
Magda: If the colour or font was different to the rest of the screen I think that would be better.

Any other feedback?
Farhan: Nope, that's all for this episode, thanks for listening and I'll see you all on the next iteration.
Magda: The beat counter would be better off hidden from the user so that they can't use it to know when the beat changes.

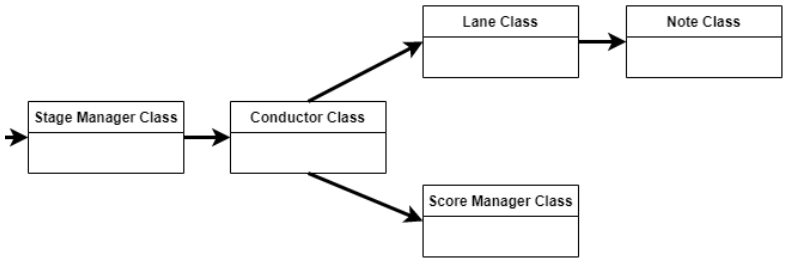
From my stakeholder feedback, the first prototype is a solid foundation to the project. The elements of the game successfully worked together and both play testers could negative the game easily; no bugs stopped the game from functioning properly. The timings and displays of the notes work as intended, and the stakeholders agreed that iteration 1 was on the right track.

From the feedback the key areas that need improvement is the design of UI elements, which will be done in a later iteration, as well as user settings. Small changes such as these will be simple to implement, however will make a big difference to user experience so I will implement these in iteration two. The main ideas communicated were:

- Remove beat counter
- Make hit bar thicker
- Re-arrange UI elements to make them more readable
- Change hit indicator element to a different font / colour
- Make hit indicators last a short period instead of staying on the screen indefinitely
- Increase scores from 30 and 15 to 300 and 200

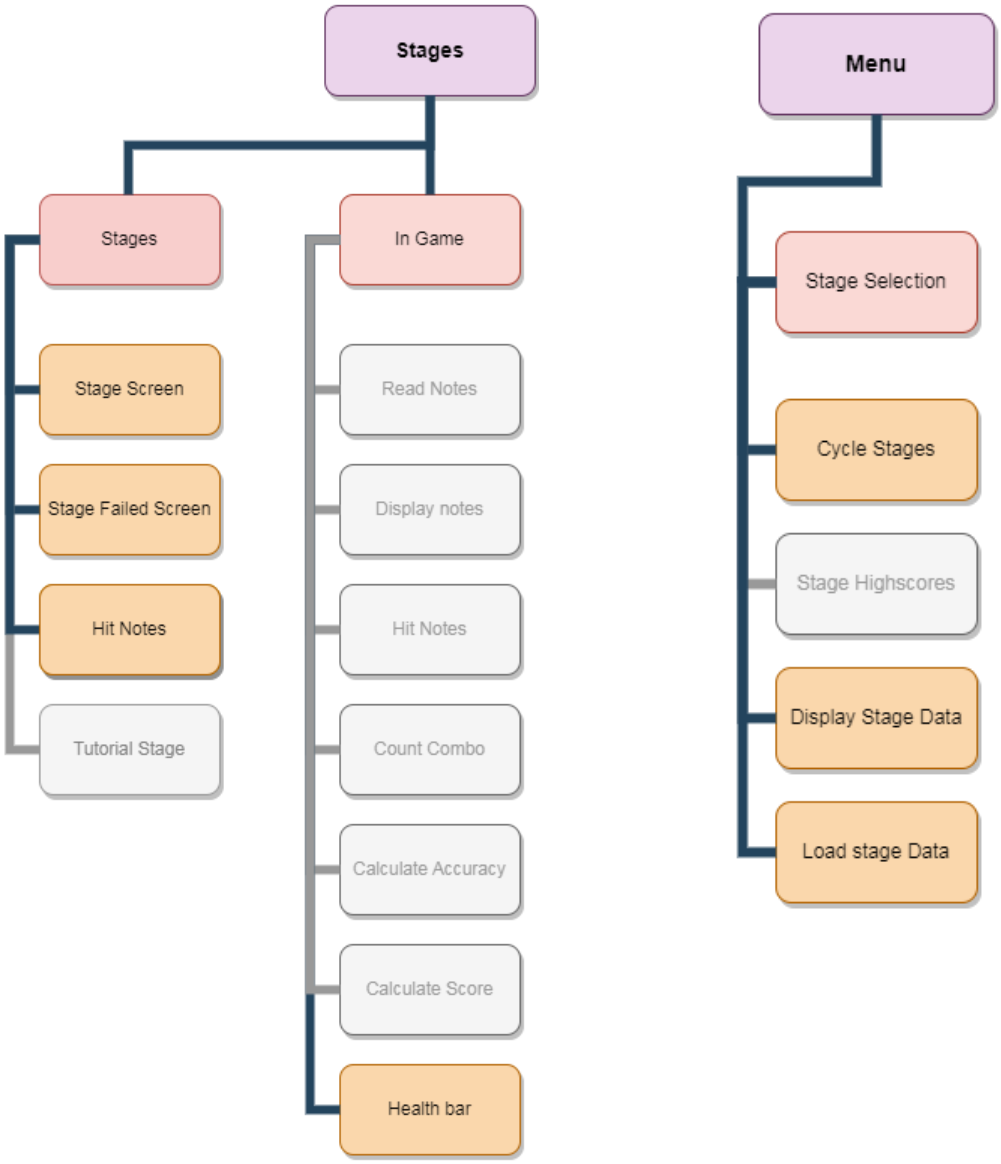
Iteration 2

Iteration 2 will build on iteration 1 by adding an additional class, and adding functionality to the previous classes, as well as responding to feedback from iteration 1.



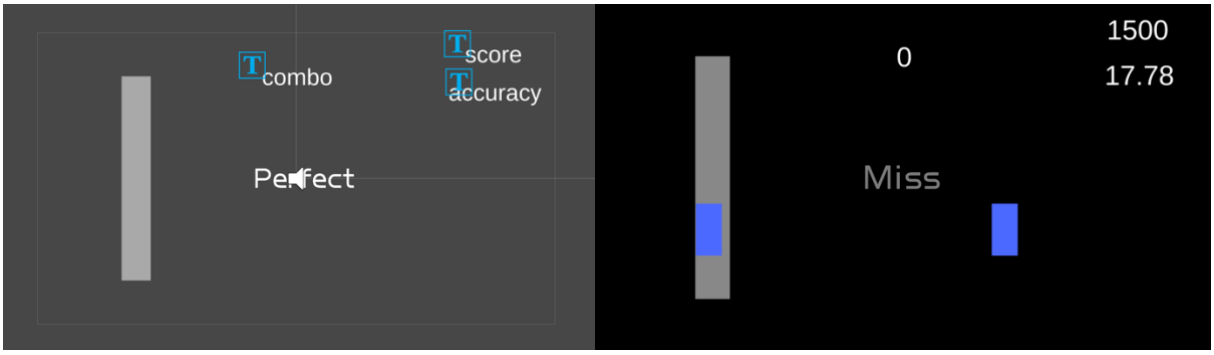
The coloured in features will be implemented. Here the Menu part of the chart is an abstraction for a new scene that will be added to the game. The new scene in Unity will hold all the game objects that handle logic for the stage selection screen, and so will need to be fully built in this iteration to handling moving between the scenes, moving between stage objects and displaying stage data.

Additionally, the stage screen will be updated with a health bar, letting the user fail, and when a stage is complete; the users results will be displayed.



Feedback Results - Feature 2.1

Applying the feedback received for prototype 1, I made changes to the layout of the stage, and added fonts for the timing indicator UI statistics to make them stand out.



An additional feature I added was making the centra indicator text fade out. A new float variable, *transparency*, where 0 represents complete transparency of the text and 1 means the text is fully visible. *Transparency* is set to 1 as soon as a timing indicator is set to appear. Then every frame until the text has disappeared, the transparency value is decreased with a non-linear calculation, dampened by the Unity variable *Time.deltaTime* to account for varying frame rates. When the value reaches 0, it stays as 0 until another hit or miss is called. This was accomplished by checking the method every frame.

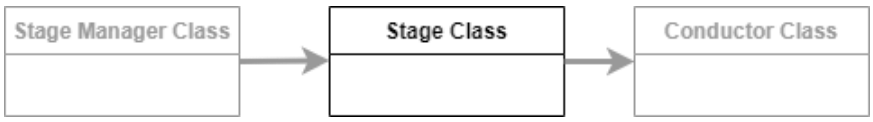
```
public void fadeText()
{
    if (transparency > 0)
    {
        transparency = Mathf.Sqrt((transparency*transparency) - Time.deltaTime*2); //fades in 1/2 seconds
        timingText.color = new Color(1f,1f,1f,transparency);
    }
    else{
        transparency = 0;
    }
}
```

Stage select - Feature 2.2

The first problem I ran into in this development section was having a method for storing data about the stages. The data about the stage, including the midi file and the audio file, would have to pass through to the next scene where the conductor and other classes are instituted. In my original design, these were stored in a dictionary; however storing all the data together is overly complicated. In order to avoid problems cause by one class controlling everything on the screen, I have decided to add an extra class between the stage manager and the conductor.

There are several reasons for splitting Stage manager into Stage class and stage manager class.

- The first is that it makes for a solution that can better be handed by object-oriented techniques. Each stage can be instanced from a single class, and have unique properties,
- Data from each stage instance can then be controlled by the stage manager, and the relationship between them and the menu can be handled properly.
- The stage selections will be moving, and will have the most user interactions, whilst the rest of the screen will be stationary and so it makes sense to separate these two aspects.



The new stage class will hold the all information about the Stage. All this data will be displayed in the stage selection menu. Reading stage data will be done in iteration 3.

Stage
+ ID : int + MidiFile : string + AudioFile : audio object + Title : string + BPM : float + Length : string + Artist : string + Difficulty : string + AudioDelay : float + highscore : int

+ SelectStage(stageID) : void
+ EnterStage(midiFile, AudioFile, AudioDelay) : void
+ LoadStageData (stageID) : dictionary

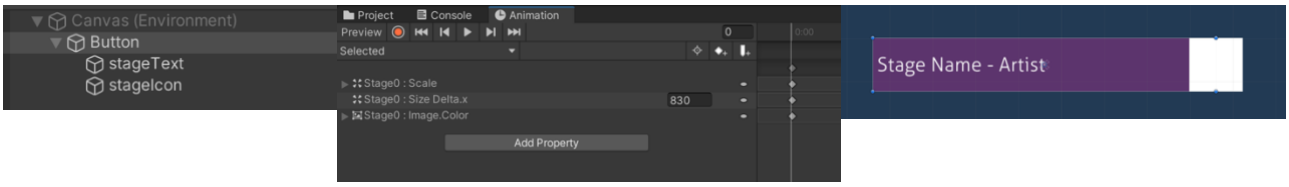
Assets > Scripts > Stage.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;
using TMPro;
using UnityEngine.EventSystems;

public class Stage : MonoBehaviour, ISelectHandler // required for OnSelect()
{
    public TMPro.TextMeshProUGUI stageText;
    public Image stageIconSmall;
    public int stageID;
    public string midiFile;
    public AudioClip audioFile;
    public Sprite icon;
    public float bpm;
    public string length;
    public string title;
    public string artist;
    public string difficulty;
    public int highscore;
    public float audioDelay;

    void Start()
    {
        // button style initialized
        stageText.text = title.ToString() + " - " + artist.ToString();
        stageIconSmall.sprite = icon;

        // load highscores
    }
    // button pressed
    public void OnEnter()
    {
        Conductor.fileName = midiFile;
        Conductor.songDelay = audioDelay;
        Conductor.bpm = bpm;
        Conductor.currentSong = audioFile;
        ScoreManager.highscore = highscore;
        SceneManager.LoadScene("MainStage");
    }

    // button hovered over
    public void OnSelect(BaseEventData eventData)
    {
        StageManager.Instance.UpdateSidePanel(stageID);
    }
}
```

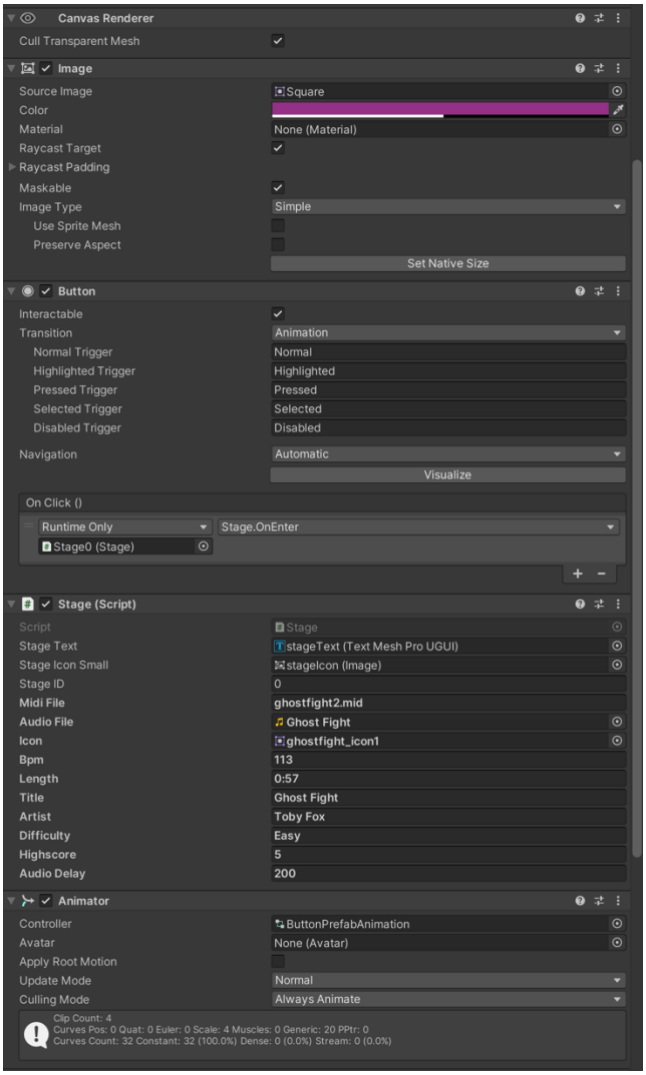
To implement the stage class, I set each variable as public and in the unity editor set values that would be used throughout the game for the first stage. The stage class does not need much functionality as it will only need to respond to button interaction, and its primary purpose is hold data. The class encapsulate all data about a stage, however other classes such as the stage manager will need to use data from the variables in the class which made it necessary to set them as public.

In the C# implementation of the class, entering the stage transitions the scene and passes variables to the conductor class. From here the game can play out the stage, and all the values that make the map unique are passed over.

Additionally, selecting a stage to see information about it is handled by unity's *ISelecthandler* and then by the *StageManager*.



Next I created a universal prefab for the stage class. The idea behind the design was that it would simultaneously function as a button, making assigning data to the button much easier.

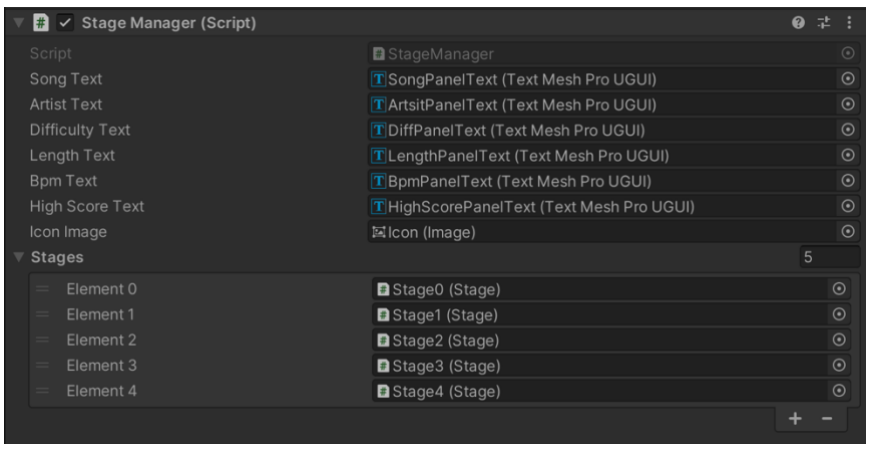


I instanced Game Objects for each of the stages, and displayed them as buttons, adding functionality for them to be pressed. As shown, each button instances a Stage class and the variables can be set for the specific stage. When the button is clicked, *OnEnter()* is called and the scene is transitioned to the scene developed in iteration 1. Because the button uses the method from its own class instance, it can pass unique values to the next scene specific to the stage, whilst keeping the buttons re-usable.

Adding animations for the selected and default state made it clear for the user to see how they were interacting with the buttons, and this was done inside the unity editor by exporting the animation as a *.controller* file type.

To hold these buttons, I made a new Scene in the unity project that would act as the stage selection screen. Unity's scene management let me transition between the two scenes to let the user enter a stage.

Since the stage Manager is a singleton class, to know what stage its currently dealing with, it keeps an array of references to every stage. I added the stages in order in the array. Arrays are a mutable data type and so more stages can be added. Any stage can be accessed through an index in the array. The *StageManger* only needs to know the *stage ID* of the stage to accesses any of its attributes.



Stage Manager controls the data panel holds references to each stage in the *Stages* array.

Assets > Scripts > StageManager.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;
using TMPro;

public class StageManager : MonoBehaviour
{
    //public static int stageSelected; //can be changed from outside script
    public static StageManager Instance;

    public TMPro.TextMeshProUGUI songText;
    public TMPro.TextMeshProUGUI artistText;
    public TMPro.TextMeshProUGUI difficultyText;
    public TMPro.TextMeshProUGUI lengthText;
    public TMPro.TextMeshProUGUI bpmText;
    public TMPro.TextMeshProUGUI highScoreText;
    public Image iconImage;

    public Stage[] stages; // array of game objects

    void Start()
    {
        Instance = this;
        UpdateSidePanel(0); // first stage is automatically selected

    }
```

The Stage Manager itself has one main method – updating the side panel. The side panel should update whenever a stage is selected, and the stage class this function using its ID. All values are fetched from the instances and displayed as text, formatted for the player to read.

```
public void UpdateSidePanel(int stageSelected)
{
    songText.text = stages[stageSelected].title;
    artistText.text = stages[stageSelected].artist;
    difficultyText.text = stages[stageSelected].difficulty;
    lengthText.text = "Length: " + stages[stageSelected].length;
    bpmText.text = "BPM: " + stages[stageSelected].bpm.ToString();
    highScoreText.text = "High Score: " + stages[stageSelected].highscore.ToString();

    iconImage.sprite = stages[stageSelected].icon;
}
}
```

The results are displayed to the user as a list of buttons, and a side UI panel. Icons for the stages were taken from the song's cover.

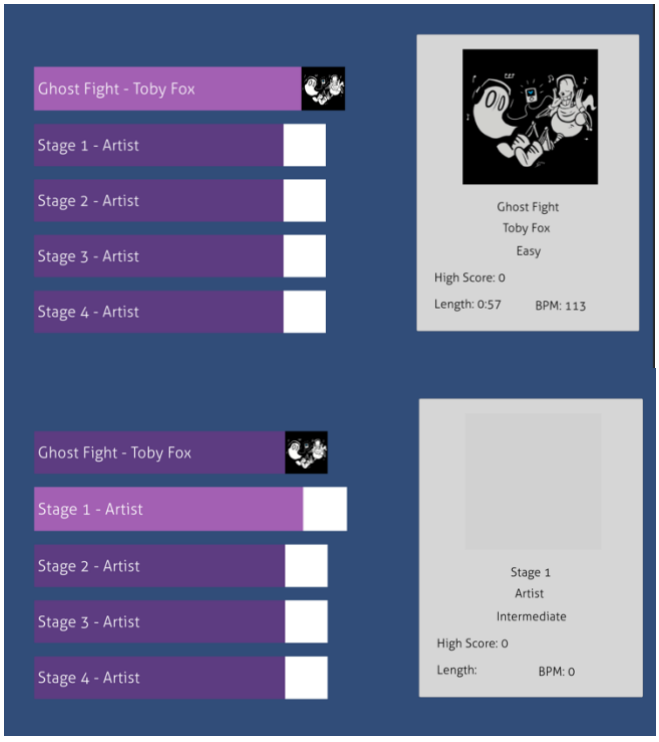


Figure 2.2.1 – Populated list of buttons can be selected with arrow keys. Buttons indicate selected state and panel on the right-side updates with stage information.

Testing Feature 2.3

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
2.1	Try selecting a stage from a list of stage	StageID, mouse, arrow keys,	Stage should be clearly selected from a list when hovered over or moved to with arrow keys. User should be able to navigate to any stage.	Figure 2.2.1	Pass

2.2	Check if Stage information is displayed.	Stage ID, Name, Highscore, Artist, Icon, BPM and Length	Stage information is displayed in a panel besides the selected stage.	Figure 2.2.1	Pass
2.3	Try interacting with stages	StageID, mouse pressed on button, enter pressed	Stage should transition to the game screen if pressed, and the game should play the song and notes from the stage that has been clicked.	Figure 2.2.2	Pass

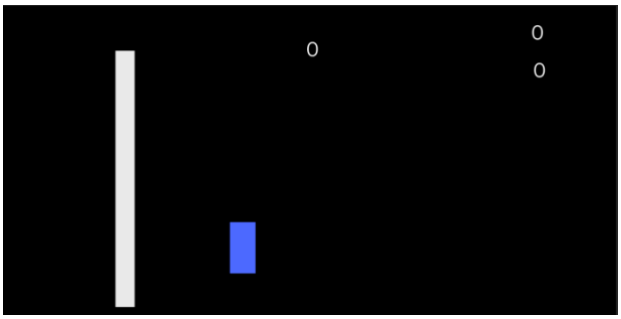


Figure 2.2.2 – Clicking through to the stage, correctly loads in the stage data from the midi file. It will be useful to repeat this test on other stage later on in development.

Health bar - Feature 2.3

To create the health bar feature, I first created a health variable to keep track of health throughout the stage. Inside the *scoreManager* class, the health current float keeps track of exact health at any moment, and health max is a constant to set the maximum health. Initially, I set the max health constant to 250 at the start of every stage to avoid users going over or under the maximum.

```
public Image healthBar; // set to mask
public Image healthBarParent; // set to parent
public float healthMax;
static float healthCurrent;
```

The two image variable types are used to display the health bar UI elements

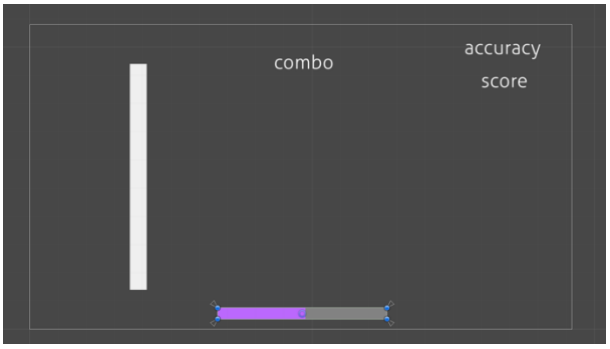
```
public static void Miss()
{
    combo = 0; // reset combo
    notesPassed += 1;
    notesMissed += 1;
    healthCurrent -= 30; // take away health
```

Health only needs to be updated every miss, since health does not change during a hit. User looses 30 health for every miss.

The health bar on screen is updated using two images, one which acts as the background and one that changes width based on the health. Width of the health bar is calculated as a percentage of the full length of the bar. Validating the health so that it cannot go below, means that the health bar cannot have a negative number as its width, which would cause an error. If the health bar is empty the stage is failed.

```
// update healthbar
if (healthCurrent > 0)
{
    healthBar.fillAmount = healthCurrent / healthMax;
}
// stage failed
else if (!failedControl)
{
    StageFailed();
}
```

The fill amount property of the health bar is a float between 0 and 1, representing a percentage of the total width.



Health bar at the bottom of the screen shows the user visually how much health left they have.

Testing Feature 2.3

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
6.8	Check health value if note was successfully hit.	Health, input hit	Health and health bar do not change,	Figure 2.3.1	Pass
6.9.1	Check health value if note was missed.	Health, input miss	Health and health bar decrease.	Figure 2.3.2	Pass
6.9.2	Test conditions for failing a stage; when health is zero.	Health, input miss.	If health below zero, then stage is failed, and health bar is empty.	Figure 2.3.3	Failed – health bar did not reach zero and health was negative.

Figure 2.3.1 – full health

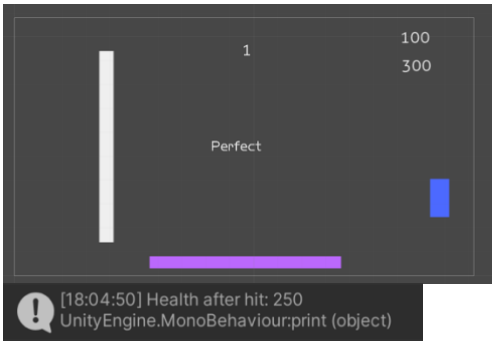


Figure 2.3.2 – drained health

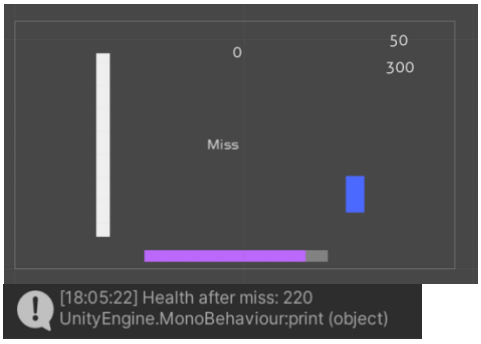
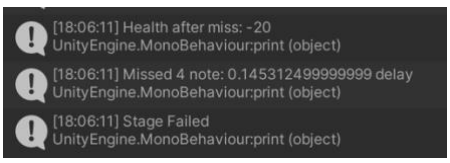


Figure 2.3.3 - Health bar did not reach zero and health went into negatives.



Test 6.9.2 Results

To fix the health going below 0, I added validation code to immediately set the health to 0 if it becomes negative. By validating the health as it's set, the game does not have to deal with negative values later, and so the later code can be updated too. Instead of the health bar

simply stopping if it encountered a negative value, the bar can be updated without any checks, and the game only needs to check for 0 to fail the stage.

```
// take away health, unless less than 0
healthCurrent -= 30;
if (healthCurrent < 0)
{
    healthCurrent = 0;
}
```

```
// stage failed
if (healthCurrent == 0 && !failedControl)
{
    print("Stage Failed");
    StageFailed();
}
```

Updated code with validation

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
6.9.2	Test conditions for failing a stage; when health is zero.	Health, input miss.	If health below zero, then stage is failed, and health bar is empty.	Figure 2.3.4	Pass

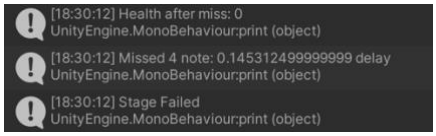


Figure 2.3.4 – Health goes from 10 to 0 and then stays at zero after continuously missing. Health bar correctly goes to 0.

Stage failed - Feature 2.4

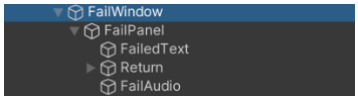
The stage failed screen will be done through a game design feature know as a modal window. Windows of this type are used for dialog boxes or pause menus and are a useful reference for my rhythm game's stage failed screen. Creating a pop-up style screen will keep the user more emersed than if they were taken out of the game entirely.

Logic is handled inside the *scoremanger* class where the health variable is stored. As soon the health variable reaches 0, the *StageFailed* function is called once. *FailedControl* is a public static Boolean variable that tells other aspects of the game to stop when it is set to true:

- Notes stop spawning
- Notes stop moving
- Inputs stop registering

```
// when stage is failed, pause all events and show fail window
public void StageFailed()
{
    failedControl = true;
    failWindow.gameObject.SetActive(failedControl);
    Instance.failSFX.Play();
    Conductor.Instance.StopSong();
}
```

The music stops and a sound effect is also played.



On top, a stage failed modal window is shown. The fail Window is comprised of an image going across the canvas, text, and a return button to take the user back to the stage selection menu. Initially it is hidden and so the *Active* property needs to be set to true.

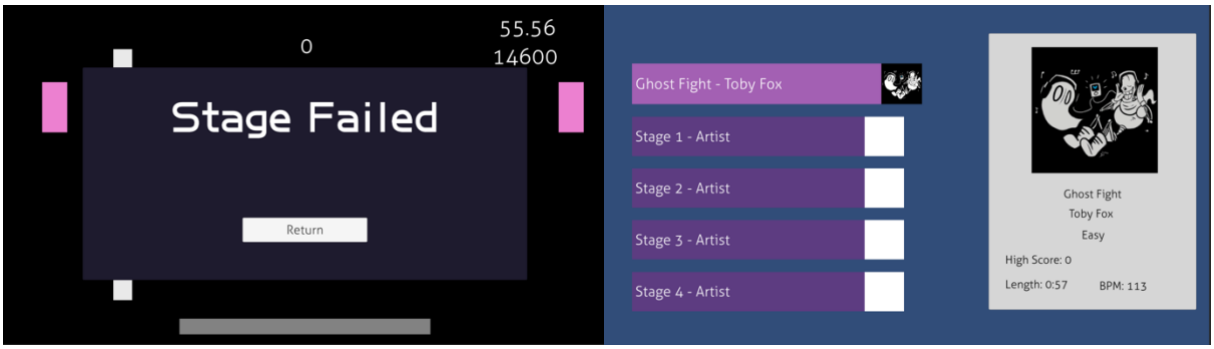


Figure 2.4.1 – Stage failed screen shows when health is empty. Notes stop and can't be hit. Return returns to stage selection screen

Testing Feature 2.4

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
6.9	Check health value if note was missed.	Health, input miss	Health decreases, if health below zero then stage failed, and failed screen shows.	Figure 2.4.1	Pass
6.11	Test game state in song failed screen	Hit input, health = 0	User should not be able to interact with the stage, all game functionalities should be paused.	Figure 2.4.1	Pass
6.12	Test song failed screen functionality	Health = 0, return clicked	Button press returns the user to the stage selection menu.	Figure 2.4.1	Pass

Stage Complete Screen – Feature 2.5

Development for the stage complete window is similar to development of the stage failed window. After each hit and miss, the number of notes passed is compared to the total number of note objects in the stage. If all the notes have occurred, a co-routine is then called instead of a method.

```
//checks if stage is complete
if (notesPassed == Instance.allNotes)
{
    Instance.StartCoroutine(Instance.stageComplete());
}

// use IEnumerator to be able to wait
IEnumerator stageComplete()
{
    // wait for 3 seconds
    yield return new WaitForSecondsRealtime(3);

    healthBarParent.gameObject.SetActive(false);

    // maxScore is the highest possible score the user can achieve
    int maxScore = 0;
    for (int i = 1; i <= Instance.allNotes; i++)
    {
        maxScore += i*300;
    }
    if (score > maxScore)  // score cannot be greater than maxScore
    {
        score = maxScore;
    }

    // update text before displaying
    comboEndText.text = "Combo: " + maxCombo.ToString();
    scoreEndText.text = "Score: " + score.ToString();
    accuracyEndText.text = "Accuracy: " + accuracy.ToString();
    perfectEndText.text = "Perfects: " + notesPerfect.ToString();
    goodEndText.text = "Goods: " + notesGood.ToString();
    missEndText.text = "Misses: " + notesMissed.ToString();

    // save highscore data

    winWindow.gameObject.SetActive(true);
}
```

allNotes is calculated by adding the total number of notes in the *timestamp* arrays in both lane classes.

StageComplete() Co-routine is called to be able to wait for 3 seconds before updating the text on the screen and showing the window.

A for loop in the middle validates the score. Max score is calculated by simulating the user hitting perfects throughout the entire stage, with *i* acting as the combo multiplier. If the score is found to be too high, it is set to

its maximum value, preventing the user from getting an impossible score. The validation is done before saving the *highscore*, which is important for maintaining data integrity.

Window is displayed as a background image covering the entire screen, with text variables on top. I added this in the same way as stage failed window in the unity editor with parented text game objects on top of a background.

Figure 2.5.1 – Statistics displayed on screen and return button functions correctly.

Testing Feature 2.5

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
6.13	Test Stage complete screen.	All notes passed, health >0	Accuracy, Score, and highest combo should all be displayed correctly, and button navigates back to stage selection screen.	Figure 2.5.1	Pass
6.14	Try setting invalid score.	Score > maximum	If score is over maximum score, score should not be allowed.	Figure 2.5.2	Pass – score was validated

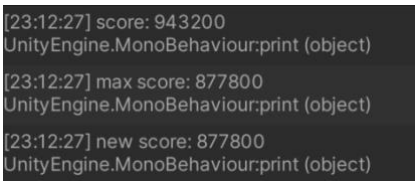


Figure 2.5.2 – score reset to maximum

Iteration 2 Evaluation

Since iteration 2 was mainly adding UI navigation, the progress made has mostly been to make the game more accessible and scalable. All the features are compatible with all future stages.

Iteration 2 has accomplished the following success criteria:

- Stage selection menu – Stages are displayed in a separate menu and user can navigate between them freely.
- Displaying stage information in selection menu – Stage selection leads to information about the stage being displayed in a separate panel.

- Adding a health bar metric – Health bar shown at the bottom of the screen displays how much health the user.
- Stage failed screen – Game can be failed if health drops to zero and all actions stop. User has to return to stage selection menu, adding difficulty to the game.
- Stage finished screen - Game can be failed if health drops to zero and all actions stop. User has to return to stage selection menu, adds a motive to play the game.

The following success criteria are not fully complete / features have not been fully developed/ need improvement in the next iteration:

- New Stages to replace placeholders – planned in iteration 3
- Stage selection screen custom sprites -– planned in iteration 4
- Background – planned in iteration 4

Stake holder feedback

Each of my two stake holders entered the game from the stage selection menu and played the stage through to either to end to the stage complete screen and then went back and failed the stage. Since prototype 2 was mainly in preparation for adding more stages, the stakeholders critiqued this prototype based on how useable and accurate the game was, and tested interactions between the UI.

Is the stage selection menu simple to navigate?

Farhan: Yes, I like the arrows keys feature to move between the stages
Magda: It's useful to have all the stages on the same page to not have to scroll between them

How important is displaying pieces of information about a stage to you?

Farhan: It's important that the song title, artist and difficulty are shown, but the bmp and length aren't as important to me.
Magda: Yeah, the information is valuable.

Does failing a song feel too easy or hard?

Farhan: Maybe make the health variable between stages, so the user has more lives for harder stages.
Magda: In my opinion is should be a little harder, especially for harder stages.

Does the stage failed screen have a smooth transition? Is it too abrupt?

Farhan: It should be as abrupt as it is.
Magda: The fail sound effects help the transition to the failed screen.

Is the stage complete screen easy to understand?

Farhan: yes, the text is well laid out.
Magda: The information is clear, and the information about how many types of each hit is valuable.

Any other feedback?

Farhan: No
Magda: Adding a background if there is time and adding more stages to fill the empty spaces.

From my stakeholder feedback, the second prototype adds structure to the game, letting the user navigate successful and play the stages successful. Changes in this iteration were mostly for user experience and so more feedback on them can be given in final testing. No bugs stopped the game from functioning properly.

Some features requested from the stakeholders were already planned in future iterations such as backgrounds, additional stages, and new stage button sprites. Stake holder's main requests and feedback to the current features were:

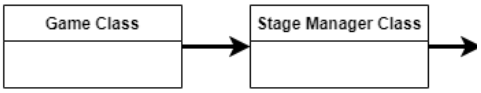
- Increase health

- Make health variable between stages

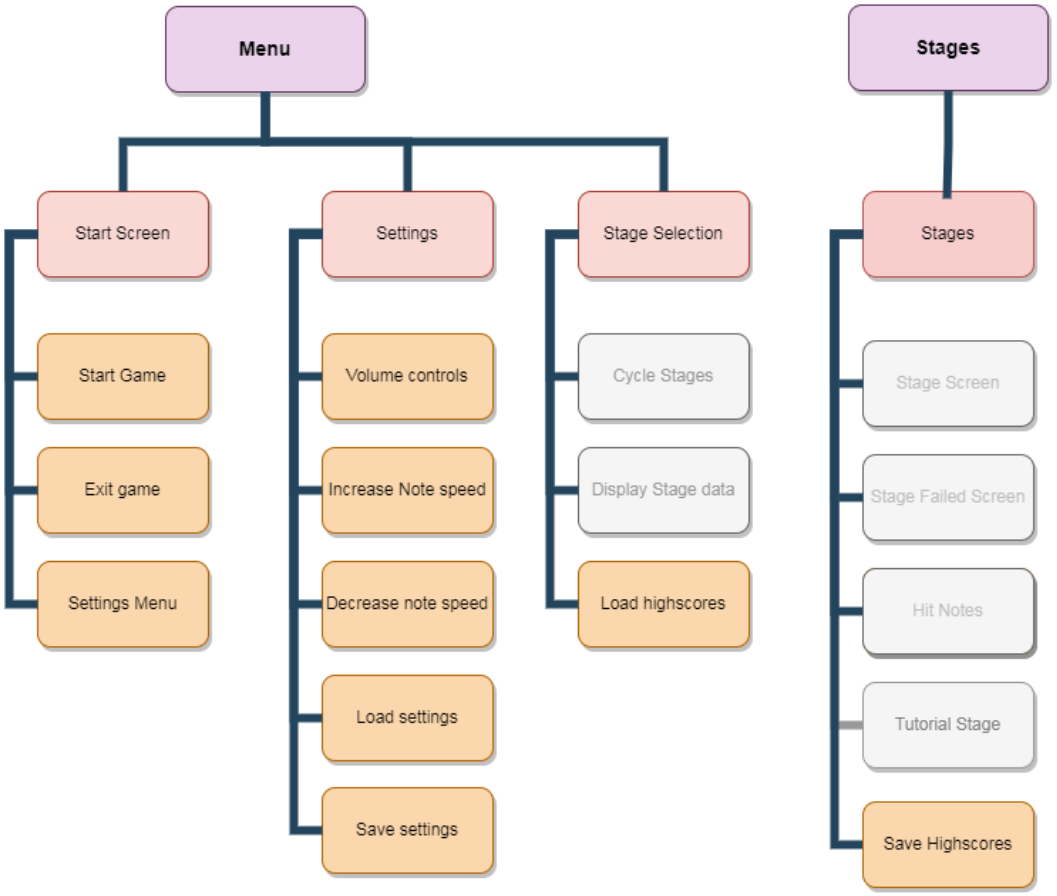
Making health inherent to each stage would allow for separate health values and variables, letting me control the difficulty of the stage better. However, this would be confusing for the user as they wouldn't be able to keep track of health between stages as well. One solution would be to let the user gain health from combo or simply to increase health for every stage because variable health would be used to make hard stage easier anyway. I chose this solution implement, and health was changed from 250 to 610, allowing for up to 20 notes missed before failing, since each note missed subtracts 30 health.

Iteration 3

Iteration 3 will add the final screen to the game and include all the functionality for the menus. I will also be adding the ability to save user setting and high score to non-volatile storage and loading the data when the game is re-launched.



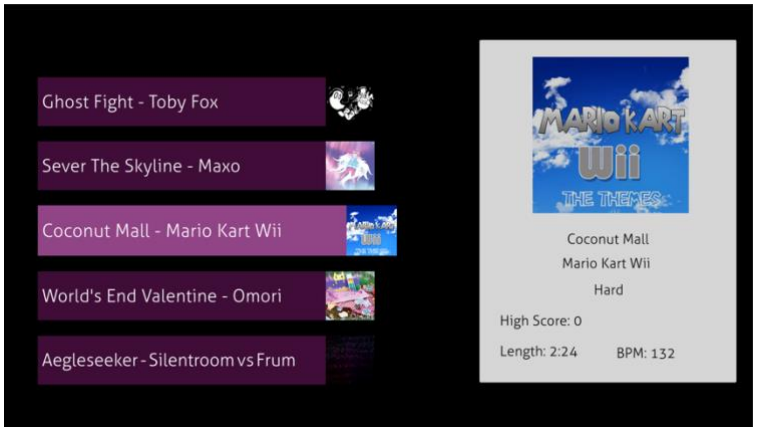
The menu part of the chart will be split into just 2 scenes instead of 3, the stage section scene that was developed in the previous iteration and the start screen, controlled by the game class, that will be called first when the game is launched. The settings menu can be implemented into the start screen, with all the functionality shown below. Some of the saving will need to be done from inside the stage, since that's where the high scores are calculated.



Additional stages – Feature 3.1

To make the game enjoyable for the users the game needed more stages to play, with a variety in songs and speeds. I used the techniques described in feature 1.1, using the LMMS software to create 4 additional midi files. The stages last about 2 minutes and have between 200 and 400 notes. To sync the note timings to the audio file, the correct bpm was set and the audio was started earlier or later in game using the song delay variable. I marked the first beat in every song and adjusted the song position until the note spawn lined up with the exact bet in the song.

Stages are ordered by difficulty from easiest to hardest and each stage has a difficulty rating in the panel. This is important so that a user starts at the first and easiest stage, acting as a tutorial, introducing the user to how the game is played. The user can then progress through the stages as desired. Limiting the game to only 5 stages makes the stages easier to search for, whilst still providing skill progression.



Main menu – Feature 3.2

The main menu Is the first and last screen of the game the user will see, and links to all other screens. I developed the *game* class, in a new scene in the game, containing a canvas to hold UI elements, a text title, a start button, quit button and a background. Highlighted buttons show selection and when the start button is pressed or enter is pressed whilst it is selected the stage selection scene is loaded. This completes the class hierarchy, of the game class linking to the rest. The quit button simply calls *Application.Quit()* to exit the game.

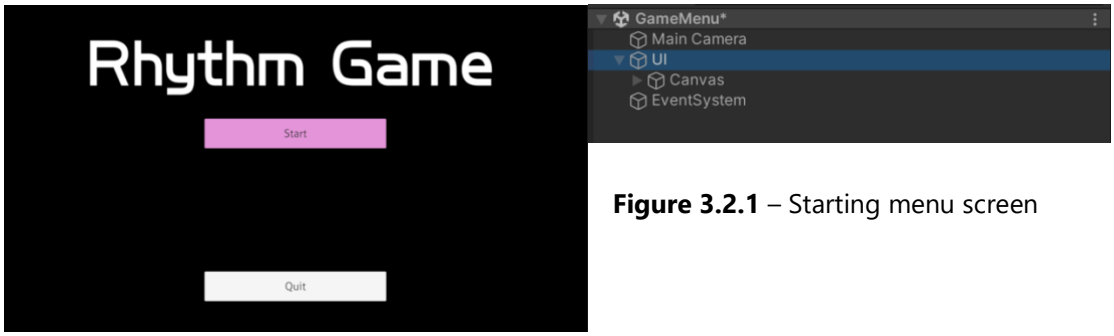


Figure 3.2.1 – Starting menu screen

In the stage selection screen, a return button was added in the top left corner, and its navigation was set to load the *game menu* scene from the stage manager class.

Figure 3.2.2 – Stage select screen shown after start button is pressed. Return button transitions back to start menu.

User Settings

Implementing user settings into the main menu screen was done in the same UI area to avoid further complicating the UI with more pop ups. Settings are encapsulated by 3 sliders, one for music volume, one for sound effect volume and one for note approach speed. Each slider has boundaries set to act as validation

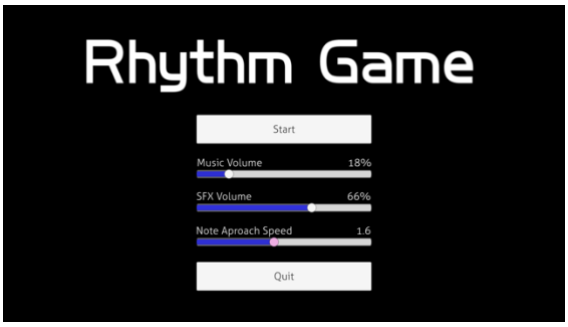
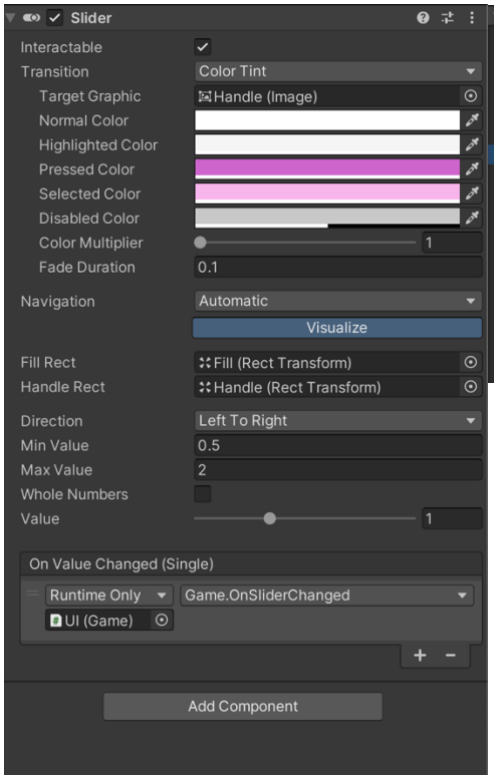
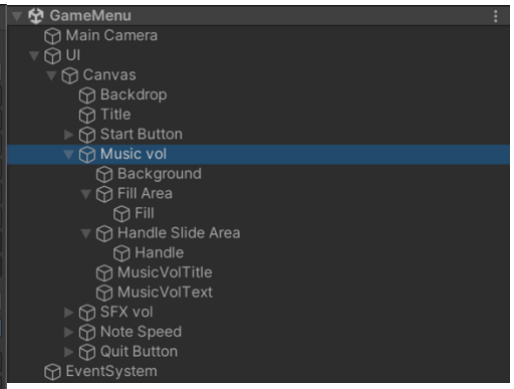


Figure 3.2.3 - User can drag sliders to change settings.

Sliders all have the same *onsliderchanged* method attached from the game class. The sliders are composed of the slider mechanics, handled by unity, the static title text on the left, and the variable text showing the slider value.





Slider values have set boundaries for additional validation.

Each slider is validated using the on slider changed method, to ensure settings are correct before the user can continue in the game, Music volume and sound effect volume are both floats that must be between 0 and 1, and if not in that range are set to 0.5. The note speed slider was reversed, so that the lower end represents a slower speed and the upper end a higher speed instead of the other way round, making the number more meaningful to the user. To adjust for this, the slider value is reversed in the code and saved to a speed variable, and that new variable is then validated to be between 0.5 and 2. At the end of each validation, the text value is updated with a rounded value.

```
29      // called once slider is moved
30      public void OnSliderChanged()
31      {
32          // validate music volume slider
33          if (musicSlider.value < 0 || musicSlider.value > 1)
34          {
35              print("Invalid music volume");
36              musicSlider.value = 0.5f; // reset to default value
37          }
38          //round to nearest whole number, convert to percentage and display
39          musicVolText.text = (Math.Round(musicSlider.value* 100, 0)).ToString() + "%";
40
41          // validate sfx volume slider
42          if (sfxSlider.value < 0 || sfxSlider.value > 1)
43          {
44              print("Invalid sound effects volume");
45              sfxSlider.value = 0.5f; // reset to default value
46          }
47          //round to nearest whole number, convert to percentage and display
48          SFXVolText.text = (Math.Round(sfxSlider.value * 100, 0)).ToString() + "%";
49
50          // convert slider value to float between 0.5 and 2
51          speed = 2.5f - speedSlider.value;
52
53          // validate new speed
54          if (speed < 0.5 || speed > 2)
55          {
56              print("Invalid note speed");
57              speed = 1f; // speed needs to be reset and not slider.
58          }
59          // display slider value and not actua value for human readability.
60          NoteSpeedText.text = (Math.Round(speedSlider.value, 2)).ToString();
61
```

Music Volume Validation

Sound effect validation

Note speed validation

```
65
66      public void OnStartClicked()
67      {
68          // pass values to conductor and score manager classes
69          Conductor.noteTime = speed;
70          Conductor.volumeMusic = musicSlider.value;
71          ScoreManager.volumeSFX = sfxSlider.value;
72
73          SceneManager.LoadScene("StageSelect");
74      }
75
```

Lastly, the start button sets static variables from unloaded scenes to the user settings saved by sliders. The next scene is then loaded.

Music and sound effect values are used to set the volume of the music and sound effect sources. In unity these are set using floats from 0 to 1, with 1 being full volume. In the Start() method inside the score manager, the sound effect volumes are initialised.

```
Instance.hitSFX.volume = volumeSFX;
Instance.missSFX.volume = volumeSFX;
Instance.failSFX.volume = volumeSFX;
```

Testing Feature 3.2

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
1.1.1	Check if Music volume changes as settings are increased / decreased.	*musicVolume* , using slider	If increasing volume slider, display number increases, *musicVolume* variable increases and game audio gets noticeably louder. Opposite would happen if slider were decreased.	Figure 3.2.4 Figure 3.2.5	Pass
1.1.2	Check if Sound effects volume changes as settings are increased / decreased.	*sfxVolume*, using slider	If increasing volume slider, display number increases, *sfxVolume* variable increases and game audio gets noticeably louder. Opposite would happen if slider were decreased.	Figure 3.2.4 Figure 3.2.5	Pass
1.2	Check if the note speed changes as settings are increased / decreased.	*userNoteSpeed*, using slider	If increasing note speed slider, display number increases, *userNoteSpeed* variable increases and the in stage note speed increases. Opposite would happen if slider were decreased.	Figure 3.2.4 Figure 3.2.5	Pass
1.6	Check if stage selection load when button 1 is pressed.	Is button1 pressed	Stage select screen is loaded and the stage Manager class is called.	Figure 3.2.1 Figure 3.2.2	Pass
1.7	Check if the settings menu loads when button 2 is pressed.	Is button2 pressed	Settings Menu is loaded and sliders for values appear.	n/a	No longer valid due to changes to the settings menu being integrated into the main menu.
1.8	Check if the game close when button 3 is pressed.	Is button3 pressed	The open instance of the game terminates.	n/a	Pass – game quit successfully
2.3	Return to main menu	Return button pressed.	Moves from stage selection screen to main menu screen.	Figure 3.2.1 Figure 3.2.2	Pass

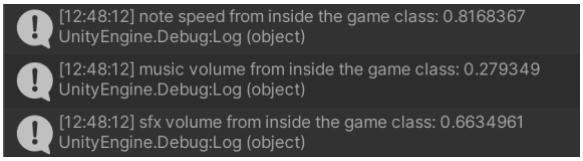


Figure 3.2.4 – Values of sliders as set in the start menu

3.2.5 – Values of sliders from after the start has been pressed. Volumes and note speed correctly.

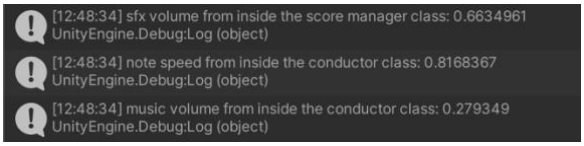


Figure button change

Saving and loading settings – Feature 3.3

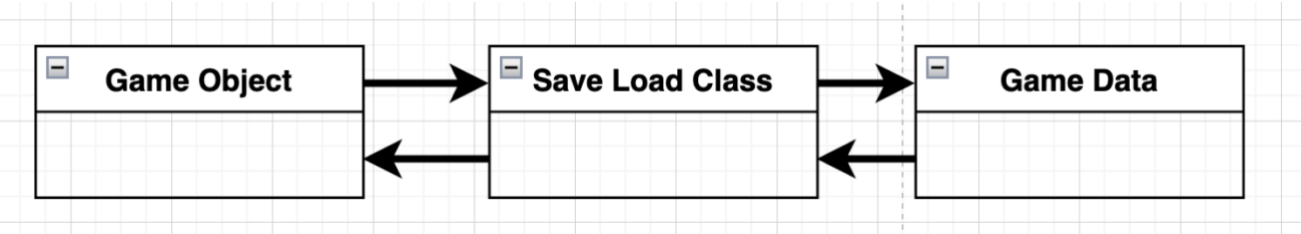
The three sliders containing user settings need to be saved permanently and be loaded when the game is opened again. Initially I planned data persistence to be done using Json files but there were a few issues with this method. Firstly, whilst Json has the advantage of being human readable and formattable, it would not be as secure as using an encrypted method. Furthermore, serialization in unity using encrypted methods is better documented and supported. For these reasons I decided to use a binary formatted instead of Json one and to save data to binary .bin files. Validation done in the previous class will be applied before saving any data.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[System.Serializable]
public class GameData
{
    public float musicVolume;
    public float sfxVolume;
    public float noteSpeed;

    // constructor method
    public GameData(Game game)
    {
        this.musicVolume = game.musicSlider.value;
        this.sfxVolume = game.sfxSlider.value;
        this.noteSpeed = game.speedSlider.value;
    }
}
```

To do this, the data had to meet two preconditions. The first being that it had to be serializable. This meant that whilst a custom struct could be created, the data would be best held contained as a series of variables or arrays. Secondly, the data had to be in its own class, and that class had to be marked as serializable. I created a kind of template class that wouldn't be placed in the scene and would only handle data. It would represent the format in which the binary data would be saved.

The constructor method in the class lets the class be used as a data structure for handling the save data.

Saving and loading itself will be done from a separate class called the *SaveSystem*. Together the classes work with any game object that has data that needs to be saved. In this case, the Game class has data from sliders that needs to be saved. The data is passed into the save load class, which passes it to the Data class to format It correctly (explained above) and then the save load class (explained below) does the saving. The same would happen for loading data, moving in the opposite direction. The game object requests data, and data in the form of the Game Data class is returned.



When used correctly the save system never need to be initialized, and only the methods are used. The save system initialises a file, which acts as a constant. When saving or loading, Unity's persistent data path is combined with the file to the location of the file, even if the file doesn't exist yet. The physical data from the game menu class is passed to the save method to be formatted as a *GameData* object. The binary formater then serializes the data into a file at the file path, overwriting any old data. Similarly for loading, a path is created, except in this case the path needs to be checked. If the file path is successfully found, then data can be read in the same format as it was saved - a *GameData* object. This data can be returned as an object of that type. If the file was not found then data cannot be loaded and a null game object is returned, which needs to be checked for in the class that requests loading data.

```
using UnityEngine;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

// static classes cannot be instantiated
public static class SaveSystem
{
    private static string gameDataPath = "RhythmGameData.bin";

    // saves user game data to file
    public static void SaveGameData(Game game)
    {
        string path = Path.Combine(Application.persistentDataPath, gameDataPath);
        // converts data to a serilazable object
        GameData data = new GameData(game);

        // write to new file
        BinaryFormatter formatter = new BinaryFormatter();
        FileStream stream = new FileStream(path, FileMode.Create);
        formatter.Serialize(stream, data);
        stream.Close();
    }

    // reads file and returns serializable game data object
    public static GameData LoadGameData()
    {
        string path = Path.Combine(Application.persistentDataPath, gameDataPath);

        //check for file
        if (File.Exists(path))
        {
            Debug.Log("game data file found");

            // read from file
            BinaryFormatter formatter = new BinaryFormatter();
            FileStream stream = new FileStream(path, FileMode.Open);
            GameData data = formatter.Deserialize(stream) as GameData; // cast type
            stream.Close();
            return data;

        }
        else
        {
            Debug.Log("game data file not found");
            return null;
        }
    }

```

```
86
87      public void Save()
88      {
89          //passes a referance to the game class
90          SaveSystem.SaveGameData(this);
91      }
92
93      public void Load()
94      {
95          // gets game data from file
96          GameData data = SaveSystem.LoadGameData();
97
98          if (data != null)
99          {
100             musicSlider.value = data.musicVolume;
101             sfxSlider.value = data.sfxVolume;
102             speedSlider.value = data.noteSpeed;
103         }
104         else
105         {
106             Debug.Log("No save made");
107             Save();
108         }
109     }
110 }
```

Leading back to the Game class, I added a function for saving to be called whenever the user quits the game or enters the stage selection menu. To save, a reference to the game object is used and to load, the game data object's values are gotten, as long as the load was successful.

Load() is called whenever the game class is called, and the game class is called at the start of the game and when the scene transitions back to the main menu so settings can be reloaded. The settings are validated because the on-slider method is also called as a part of loading slider values.

Testing Feature 3.3

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
1.3	Check if user settings are read from JSON file.	*userFilePath* = valid file path	User values are read from JSON file under file path given.	Figure 3.3.1	Pass
1.4	Check if user settings are read from JSON file when the file exists, but the file path is incorrect.	*userFilePath* = invalid file path	Invalid file path is rejected, and valid file path is located and read from.	n/a	Pass – since file path is constant with unity's persistent data path, the valid path is always located. If the file path were to be changed, data would just be saved in that new file.
1.5	Check if user settings are read from JSON file when the file does	*userFilePath* = empty file path	No file path is found, so new file is created with empty values, and file path is saved.	Figure 3.3.2	Pass

	not exist, and there is no file path.				
1.9	Check if user settings are saved	Slider value changed	Current settings should be saved to a new file, and the old file deleted.	Figure 3.3.1	Pass

Figure 3.3.1 – initial settings are overridden with saved settings as soon as game is loaded. Settings can be changed and are loaded again when game is closed and re-opened.

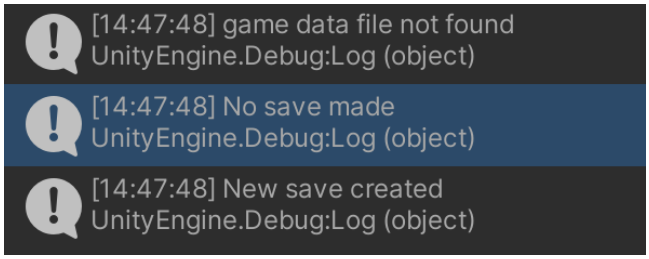


Figure 3.3.2 – When there was no file to load, no data was loaded and a save was made with the default values for all three user settings.

High scores – 3.4

Previously, the high score field shown for each stage was set to a constant during runtime. To add high scores, I first had to check for a high score at the end of each completed stage in the score manager class. This was only done after score was validated.

```
// set new highscore
if (score > highscore)
{
    highscore = score;
    //saves highscore from stageManager class
    StageManager.setHighscore(currentStage, highscore);
}
```

Figure 3.4.1 –The *stage ID* and *high score* are passed to the *stage manager* from the *score manager* class, which checks for a high score after validating the score.

Inside the stage manager class a new was made to hold the highscores. Because scores were defined as *integers,* list must be of type int as well. The list would hold all the stages highscores in one place at any time. Whenever the stage manager class is called, all the high scores from the stages are added to the list. These highscores can then be serialized and saved.

```
public List<int> highScores = new List<int>(); // list of highscores
```

```
// stage highscores addded to list
foreach (var stage in stages)
{
    highScores.Add(stage.highscore);
}
```

```
public static void setHighscore(int stageID, int highscore)
{
    StageManager.Instance.highScores[stageID] = highscore;
    Debug.Log(StageManager.Instance.highScores[0]);
    Debug.Log(StageManager.Instance.highScores[1]);

    // save highscores
    StageManager.Instance.Save();
}

public void Save()
{
    // passes a referance to the stage manager class
    SaveSystem.SaveHighScoreData(StageManager.Instance);
}
```

The highscores are saved using the list format from the class using the *setHighscore()* method. First the list is updated with the new high score from the score manager class. Using this method allows all highscores to be saved at once.

Again, the save system is used with a instance of the data to be serialized and sav

Exactly like with the *GameData* class, a serializable class was made, except this time using an array data type, rather than using a dictionary which cannot be serialized. Each index of the array corresponds to a stage ID, and so all of the stage highscores can be saved at once. To set each value of the array, a for loop is used.

```
[System.Serializable]
public class ScoreData
{
    public int[] highScores  = new int[5];

    public ScoreData(StageManager stageManager)
    {
        for (int i = 0 ; i < 5; i++)
        {
            this.highScores[i] = stageManager.highScores[i];
        }
    }
}
```

The Save system was also extended with two new methods, one for saving data of type *ScoreData*, and one for loading data of *ScoreData*. Overall, the system for highscores works the same as the one for user settings. For purposes of data integrity, it is vital that the data is only loaded for each individual stage and that the stage manager then references that data. The received data after the load is validated with a null check, preventing highscores being overwritten with empty values.

```
public void Load()
{
    // gets game data from file
    ScoreData data = SaveSystem.LoadHighScoreData();

    if (data != null)
    {
        // only loads data matching stage ID
        highscore = data.highScores[stageID];
    }
    else
    {
        Debug.Log("No high score save made");
    }
```

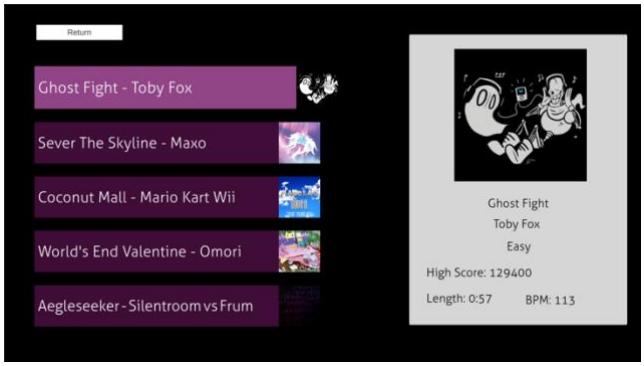
Highscores are loaded by each stage individually whenever the scene is loaded, and so each stage only uses the high score corresponding to its ID. If the ScoreData object is empty, no high score is loaded. In this case the default high score value remains, which is set to 0 at run time.

Testing Feature 3.4

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
2.5	Check if high scores are read from JSON file.	userFilePath = valid file path	User values are read from JSON file under file path given.	Figure 3.4.3	Failed – Some highscore are not read to the stage manager
2.6	Check if high scores are read from JSON file when the file exists, but the file path is incorrect.	userFilePath = invalid file path	Invalid file path is rejected and valid file path is located and read from.	n/a	Pass – since file path is constant with unity's persistent data path, the valid path is always located. If the file path were to be changed, data would just be saved in that new file.
2.7	Check if high scores are read from JSON file when the file does not exist, and there is no file path.	userFilePath = empty file path	No file path is found, so new file is created with empty values, and file path is saved.	Figure 3.4.3	pass
2.8	Check if high scores are saved	High score set	Current high scores should be saved to a new file, and the old file deleted.	Figure 3.4.2	pass
6.1	High score is recognised and saved	Score > high score	A new high score is set and sent to the stage manager class to save.	Figure 3.4.1	pass

```
jachym@192 ~ % cd '/Users/jachym/Library/Application Support/DefaultCompany/Rhyt
hm Game'
jachym@192 Rhythm Game % ls
RhythmGameData.bin      ScoreData.bin
```

Figure 3.4.2 – Checking the persistent data path given by unity shows the two data files exist – high scores and user settings have been saved.



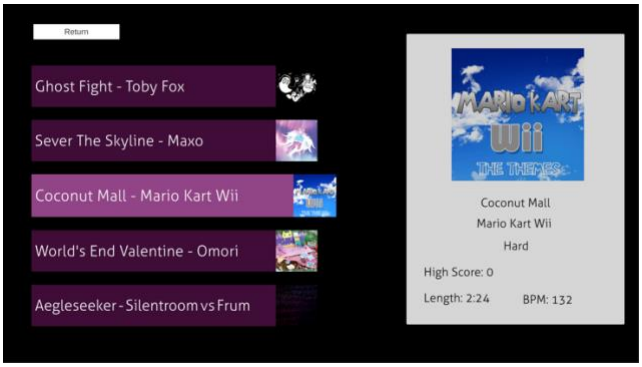
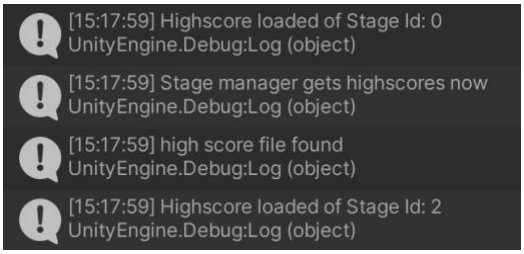


Figure – 3.4.3 – Only some high scores are loaded, others are still set to the default value of 0. In the figure, the left had side shows an incorrectly loaded high score and the right-hand side shows a correctly loaded high score.

Test 2.5 Results

Checking the values loaded from the files shows that the scores in the files for stage 2, which was not loaded properly, had values that could have been loaded. Saving and loading were both done correctly but the error occurred when the stage manager was fetching the values.



Printing the order of being loaded for the stages shows that the stages do not finish loaded before the stage manager fetches the high scores.

Some stages are loaded after the stage manager, leading to the default highs scores being fetched instead of the newly loaded ones. This led to 0 being displayed during stage selection instead of the high score set previously.

```
IEnumerator ReadHighscores()
{
    // wait for 0.1 seconds
    yield return new WaitForSecondsRealtime(0.1f);

    Debug.Log("Stage maanger reading scores");

    // stage highscores addded to list
    foreach (var stage in stages)
    {
        highScores.Add(stage.highscore);
    }
}
```

To fix the error, I added a delay to the stage manager using a coroutine that waits 0.1 seconds before reading the highscores, giving the stages time to load. The new code is shown to the left.

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
2.5	Check if high scores are read from JSON file.	userFilePath = valid file path	User values are read from JSON file under file path given.	Figure 3.4.3	Pass

Figure 3.4.3 – Stage high score loaded correctly

Iteration 3 Evaluation

The main goal of iteration 3 was to implement the full class system, adding a game class for the main menu and adding data persistence for user settings and high scores.

Iteration 3 has accomplished the following success criteria:

- Main menu screen– Main menu screen displays when the game is launched and transitions back and forth between the stage selection screen and other screens. Introduces the user to the game with a title.
- Settings options in the main menu – Ability to change music volume, sound effect volume and note approach speed with sliders.
- Additional stages – User has a choice of 5 stages to selection from in the stage selection menu- each with uniquely themed music and difficulty.
- Save user settings – Settings are carried over between game and automatically saved and loaded.
- Save user high scores – high scores are saved whenever a new high score is set and loaded in the stage selection menu.

The following success criteria are not fully complete and need improvement in

- Main menu background – planned in iteration 4
- Button sprites – planned in iteration 4
- Slider sprites – planned in iteration 4

Stake holder Feedback

Both of my stakeholders entered the game from the start menu, entered the stage selection, played a stage until completion, setting a high score, returning to the stage selection menu, and adjusting settings. In both play tests, the settings and high scores loaded and saved correctly.

How easy is the main menu to navigate?

Farhan: Very easy

Magda: The buttons are placed in order of importance and the sliders are also positioned nicely. The sliders could stand out a little more.

Are the sliders clearly labelled?

Farhan: Yes, especially the note approach speed display.

Magda: I think most players would find them clear.

Is the return button in the stage selection menu clear?

Farhan: It's a little small for me.

Magda: Its clear.

Should highlighting a stage (rolling over it with the mouse) also display its information?

Farhan: Although some rhythm games disregard the mouse completely, I like the addition control having a cursor gives. It should have the same functionality as arrow key selection, especially for users who aren't used to those controls.
Magda: Yes.

How important is knowing whether you achieved a high score to you?
Farhan: Very important.
Magda: It is important, but not as important as setting a best combo.

Any other feedback?
Farhan: Possibly adding a best combo and highest possible combo display, as said by Magda.
Magda: In my opinion saving the users best combo along with the high score would be an interesting feature. Aiming for the highest combo is a fun part of rhythm games for me and would encourage the player to try and get the maximum combo.

From my stakeholder feedback, the third prototype added all the functionality wanted from the game, making it fully operational, with no major changes needed. Changes in this iteration were mostly for user experience and so more feedback on them can be given in final testing. No bugs stopped the game from functioning properly.

Some features requested from the stakeholders were already planned in future iterations such as backgrounds, slider and button sprites and score display sprites. Stake holder's main requests and feedback to the current features were:

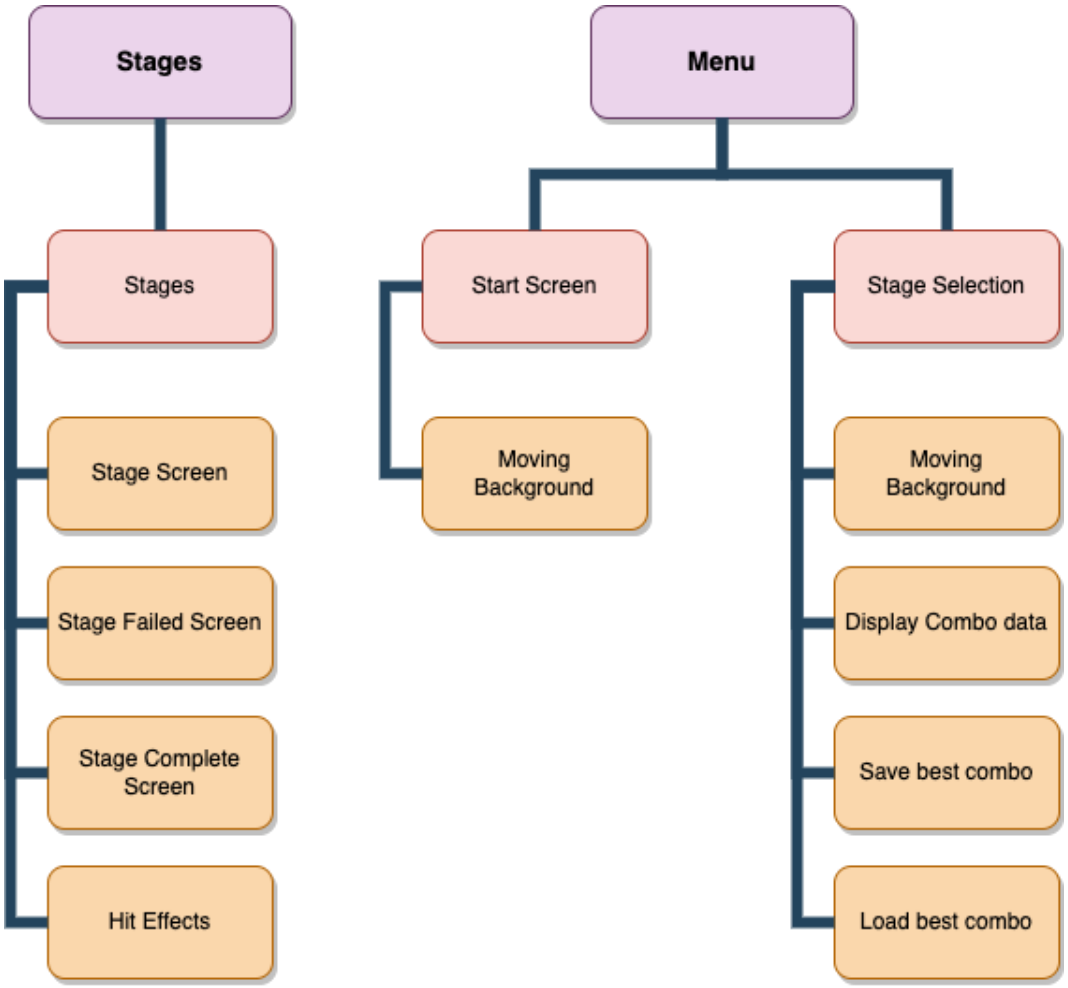
- Show best combo along with high score
- Show highest possible combo
- Save best combo along with score
- Display message for getting a new high score
- Make hovering over a button with the mouse select it as well.

I took these suggestions forward into iteration 4.

Iteration 4

Iteration 4 will add the graphical elements to the game. The aim of this iteration will be to make the game look and feel lively, exciting, and flashing, marketed towards a younger audience and emphasising the games modern choice of music. The visuals should be simple, geometric, and colourful, fitting into the types of games researched. The gameplay itself is fast and so the visuals need to match that energy, without being too intense so that newer rhythm game players can still enjoy the experience.

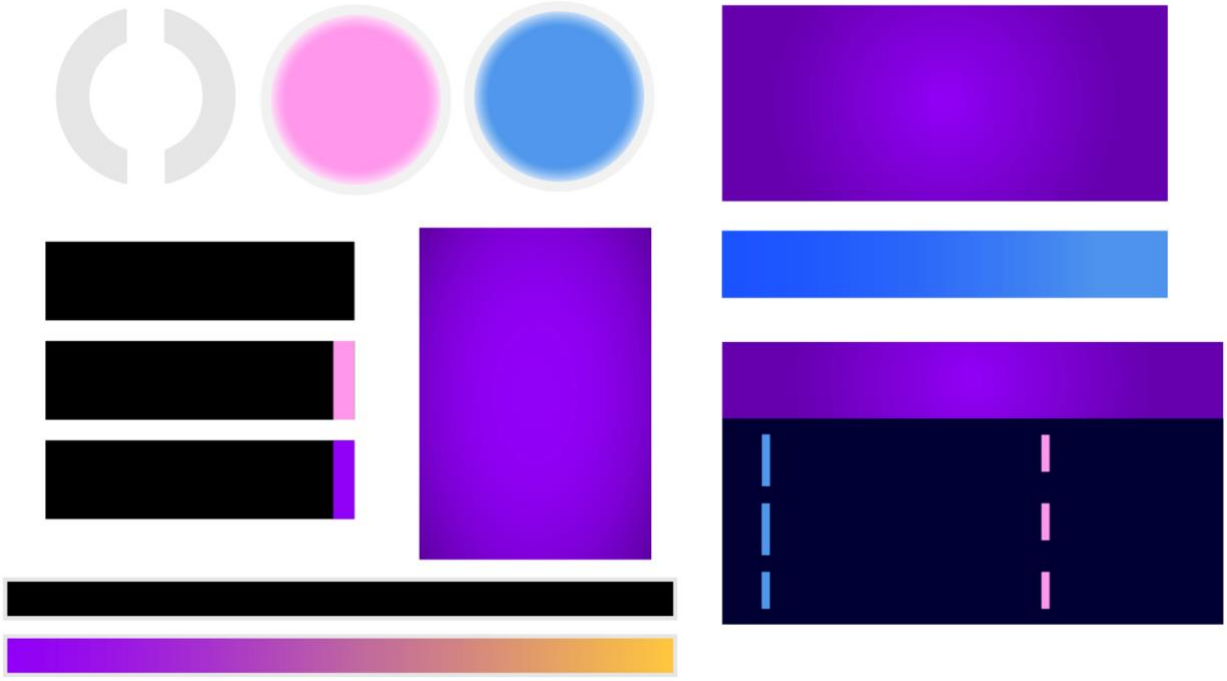
Graphics will be made for the following areas of the game. In the stage selection menu and save system, compatibility will be added for saving and displaying combo, as discussed in the iteration 3 evaluation.



Creating Sprites – feature 4.1

Creating sprites for key objects in my game in photoshop, I could then apply then as images to the game. Together, these would make the game look and feel complete and be enticing for the player. A priority for creating game sprites was establishing a vibrant colour scheme that would tie the separate screens together. The game including all of the following sprites:

- Hit indicator
- Top and bottom notes
- Stage Failed window
- 3 menu button state (normal, selected and pressed)
- Stage data panel
- Stage selection button
- Stage Complete screen
- Empty health bar
- Full health bar



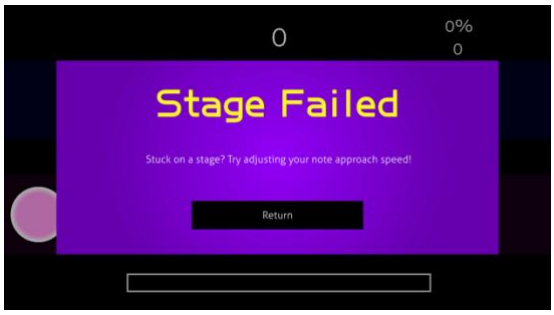


Figure 4.1.1 – Stage failed shown on top of main stage screen, with empty health bar sprite, unselected button sprite and the bottom lane sprite. Adding a tip to the stage failed screen also encourages the user to try again.

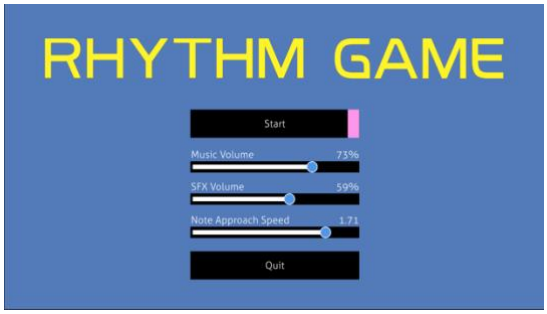


Figure 4.1.2 – Menu screen displays the title and shows selected button sprite, along with changed slider designs.

Sprites will be tested in the evaluation.

Saving Combo – feature 4.2

From feedback in iteration 3, I added best combo statistics to the save system. With every high score set, the game should also save the highest combo the user had when setting the score. This is done by rhythm games such as 'osu!', where high score contains other details about the play, such as the date and time, and accuracy. A new high score overrides the previous high score, and the combo is simply a statistic to go along with it.

In the stage class, 2 new variables were added: *bestCombo* and *maxCombo*. *maxCombo* is a constant that holds the total number of notes for the stage, calculated previously in the game. A list for best combos was added in

the score data class and in the stage manager, working based of indexes exactly like with high scores. The stage can then read the best combo value exactly like with highscores.

```
[System.Serializable]
public class ScoreData
{
    public int[] highScores  = new int[5];
    public int[] highCombos  = new int[5];

    public ScoreData(StageManager stageManager)
    {
        for (int i = 0 ; i < 5; i++)
        {
            this.highScores[i] = stageManager.highScores[i];
            this.highCombos[i] = stageManager.highCombos[i];
        }
    }
}
```

```
public Stage[] stages; // array of game objects
public List<int> highScores = new List<int>(); // list of highscores
public List<int> highCombos = new List<int>(); // list of highcombos
```

```
// only loads data matching stage ID
highscore = data.highScores[stageID];
highcombo = data.highCombos[stageID];
```

Figure 4.2.1 - Another feature requested in the feedback from iteration 3 was a message telling the user a new high score was set. The user can then see how much they beat the high score was beaten by with the high score statistic, which isn't updated with the new high score.

Figure 4.2.2 – The best combo data is displayed in the stage data panel, along with the total number of notes in the stage. This lets the user put their progress in context.

Testing Feature 4.2

The conditions for saving a combo and high score are the same, which made it best to repeat tests from feature 3.4. Tests check that combo and high score are both still saved correctly.

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
2.5	Check if high scores are read from JSON file.	userFilePath = valid file path	User values are read from JSON file under file path given.	Figure 4.2.2	Pass
2.6	Check if high scores	userFilePath	Invalid file path is	n/a	Pass – since file

	are read from JSON file when the file exists, but the file path is incorrect.	= invalid file path	rejected, and valid file path is located and read from.		path is constant with unity's persistent data path, the valid path is always located. If the file path were to be changed, data would just be saved in that new file.
2.7	Check if high scores are read from JSON file when the file does not exist, and there is no file path.	userFilePath = empty file path	No file path is found, so new file is created with empty values, and file path is saved.	Figure 4.2.2	pass
2.8	Check if high scores are saved	High score set	Current high scores should be saved to a new file, and the old file deleted.	Figure 4.2.2	pass
6.1	High score is recognised and saved	Score > high score	A new high score is set and sent to the stage manager class to save.	Figure 4.2.1	pass

Hit effects – feature 4.3

Whilst hit effects were originally planned to be more intense, I decide to keep them subtle, to let the user know they were hitting the notes but not to make the game overly flashy – going for a simple geometric style. Including hit effects, the game has 3 responses to hitting notes: hit sound effects, hit indicators lighting up and notes being destroyed.

To implement the feature, I created 3 new variables inside the *Lane* class. Whenever the input key was pressed, before checking for hitting a note, a float corresponding to the sprites alpha, *hitAreaAlpha*, value is set to 1, meaning a full alpha value of 255.

```
// hit area effects starts at full alpha brightness
hitAreaAlpha = 1f;
t = 1f;
```

Next, the value was interpolated towards the constant float variable *grey,* which was set to 0.5 brightness at run time. The interpolation is done with another variable *t*. This float between 1 and 0 works the same at the interpolation done by notes when changing positions, except using Unity's *SmoothStep()*, non-linear interpolation function. The result is a sprite that glows brightly when the lane key is pressed and fades to grey over less than a second.

```
// whilst image isnt at the grey constant, decrease alpha
if (hitAreaAlpha != grey)
{
    fadeHitArea();
}
```

```
public void fadeHitArea()
{
    // using unity's non linnear interpolation
    t -= Time.deltaTime/3;
    hitAreaAlpha = Mathf.SmoothStep(grey, hitAreaAlpha, t);

    // change alpha after interpolatinng
    laneHitArea.color = new Color(1f,1f,1f,hitAreaAlpha);
}
```

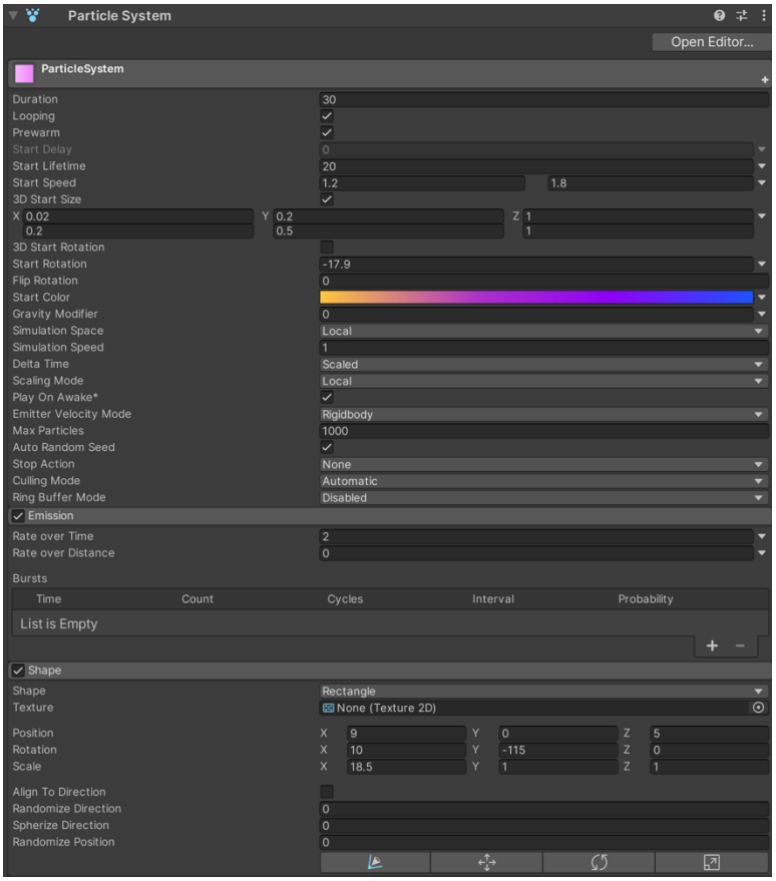
Figure 4.3.1 – Top hit indicator lights up on pressing the top lane input key.

Testing Feature 4.3

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
5.9	Test hit effects	Player input	Hit effects happen instantaneously on the correct player input.	Figure 4.3.1	Pass

Background – feature 4.4

An animated background adds a lot of character and value to a game, making it feel more alive. To create the background I used Unity's particle system, a powerful system for animating the same sprite to create copies and move in repeated loop with random variance [20].



I set up the game object as shown, using a simple rectangle sprite as the base, and adding a random chance to the width, the length, and the speed. Then I positioned the particles to come from the top right corner, all going in the same direction.

Over a time of 30 seconds, the particles spawn as colours from the gradient. Result is a colourful, light streak effect.

Lastly, a random starting state was set for the particle system inside the game and stage manager classes. With the code below.

```
// start partice loop partway through
particleSystem1.Simulate(UnityEngine.Random.Range(0f, 30f));
particleSystem1.Play();
```

The entire particle system was placed on top of a simple blue background created using gradients in Adobe Photoshop.

Figure 4.4.1 – Random moving background generated.

Testing Feature 4.4

N.O	Description	Test data	Expected result	Evidence	Pass / Fail
1.10	Test background	Game loaded	Background is loaded and plays repeatedly in a loop.	Figure 4.4.1	Pass

Iteration 4 Evaluation

The final iteration managed to most of the scope of the project estimated in requirements for development.

Iteration 4 has accomplished the following success criteria:

- Animated background
- Hit effects
- Menu buttons
- An overall theme

The major lacking features that were not implemented were:

- Miss effects – Like hit effects, miss effects would have emphasised missing a note, making the player more aware of their mistakes. This feature was vaguely defined in the design stage, making it difficult to implement, and due to time and stakeholder feedback from previous iterations, I decided the miss effects would not be added. The miss sound effect along with the miss text would still let the user know when a note was missed.
- Long hold notes – Long hold notes were a feature planned to add interest to the stages, adding another style of gameplay to the stages, by storing a start and end position instead of just a time, and distinguishing between normal, single tap notes. The feature was ambitious and as the game developed it became clear that the gameplay in itself was complex enough to add interest without needing long hold notes. Due to time restriction and the addition of stages in iteration 3, it would not have been feasible to go back and re make stages with added hold notes, as well as re-designing the how notes work entirely. Notes would have to different between the two times, and new sprites would have needed to be added for them.
- Hidden notes – Due to time limitations hidden notes were not added.

- A tutorial stage – Difficulty ratings to replace a tutorial stage. Game features no progression and players can play difficulties at wish. New players will still be encouraged to try easier difficulties if they fail harder ones and more experienced players can go straight to harder difficulties.

Stake holder Feedback
For the final iteration, I let both my stake holder play the game freely. Both of my stakeholders entered the game from the start menu, changed settings, navigated through the menus, and played through each stage until completion or failing and moving onto the next.

Is the UI consistent between the different screens?
Farhan: Yes especially with the return buttons being the same.
Magda: The main menu buttons and stage selections are consistent.

Is the UI easy to navigate?
Farhan: Yes it's easy to follow.
Magda: I like the controls.

Is the heads-up display whilst playing clearly laid out?
Farhan: Definitely.
Magda: The size of elements works well together.

Does the background suit the style of the game?
Farhan: I would say so.
Magda: Its very geometric and minimal like the rest of the game so yes.

Are you more interested in playing the game now that the graphics have been updated?
Farhan: Yes, they add a lot to game feel.
Magda: Also keeping the game play screen minimal doesn't affect the gameplay.

Is the max combo text clear?
Farhan: Yes.
Magda: Yes.

Do the hit effects make the game feel more responsive?
Farhan: They are quite subtle so it can still feel like the game is not that responsive.
Magda: Yes, the game feels fairer.

How important is a tutorial feature?
Farhan: Quite important, it would have been nice to have some text to tell the user which keys to press to play the game. It was confusing at first to figure out which keys to use.
Magda: It would be nice but it's not important to me.

Any other feedback?
Farhan: Maybe a pause menu to be able to return back to the stage selection menu from a stage.
Magda: I agree.

Skate holders gave feedback mostly on features from stage 4, and at times on the game as a whole. The sprites functioned properly in throughout the game and were all integrated into the design. Feedback was positive on the whole, with the background having a larger impact on the game feel than I first thought it would. The other aesthetic changes also helped package the game. The last iteration was successful at adding a polished feel to the game.

As pointed out by the stake holders there were a few final touches that could be added to improve the game, other than the features that I couldn't add because of time limitations. The main critiques were the lack of a tutorial and a pause feature:

- Adding tutorial text to the main screen – This would be a simple change as it would only require adding a small amount of text, but it would help new players know which keys to use. Instructional text would help guide new players. No new code was needed for this change.
- Making the hit effects more noticeable – This would make the game feel more responsive but would also distract from the gameplay. There could also be risks for photosensitive player and so I chose not to make this change.
- Adding a pause menu – A pause menu was not highly requested and so adding one would not be justified. However, an easy solution was to add a single return button. No additional code was needed since logic for returning was already done by the score manager and so the user could simply exit the stage instead of waiting for their health to run out.

Text added to the top right corner of the game menu and button added to the left corner of the stage screen.

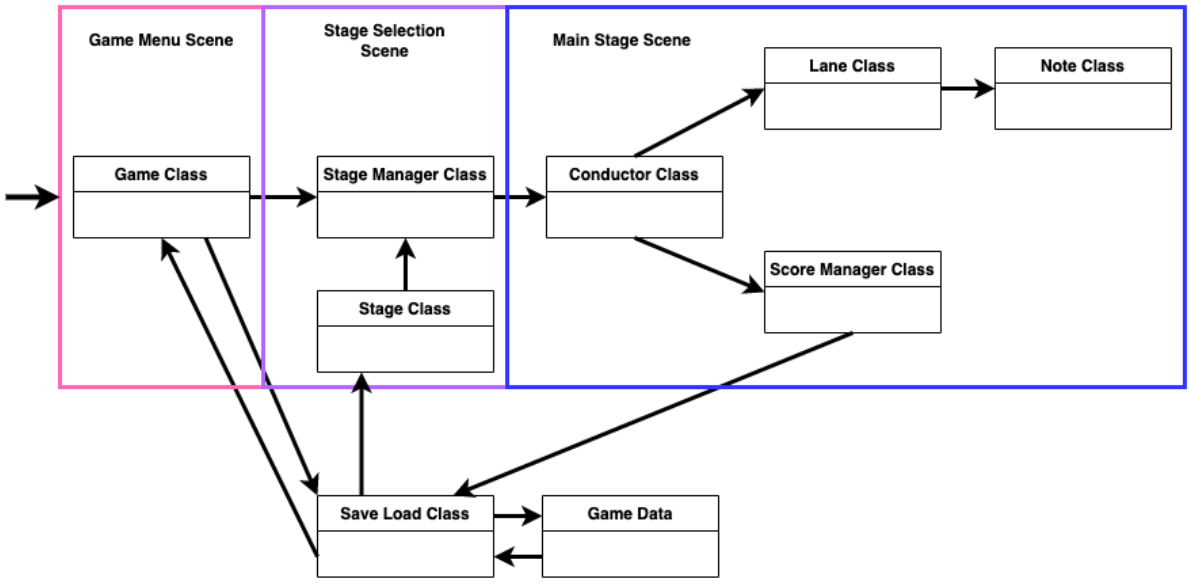
Evaluation

Across the evaluation, the Rhythm Game will be evaluated in different way ways. The main one being testing the features developed to see if they work properly and stating technical limitations. Next stake holder feedback will provide opinions from the target demographic, and an unbiased look at the project. The last is a retrospective review of the success criteria, comparing aims that were set at the start of the project with what has been developed, evaluating the final software.

Development Review

Looking back at the 4 iterative development stages, my goals established in analysis stayed mostly constant. Algorithms were the thing most heavily changed, since I found better ways for methods to work during development. The core mechanics developed in iteration 1 mostly stayed the same, and using MIDI ended up being a good choice and worked as need. No major issues came up in development, however finding small bugs was a tricky and tedious process, since I was learning the language as I went.

The choice of using C# and Unity were beneficial and allowed me to develop an object orientated solution. I could have better chose what each class did, as some classes were unorganised and needed messy solutions to link up. Making more use of private variable would have helped make the game more maintainable.

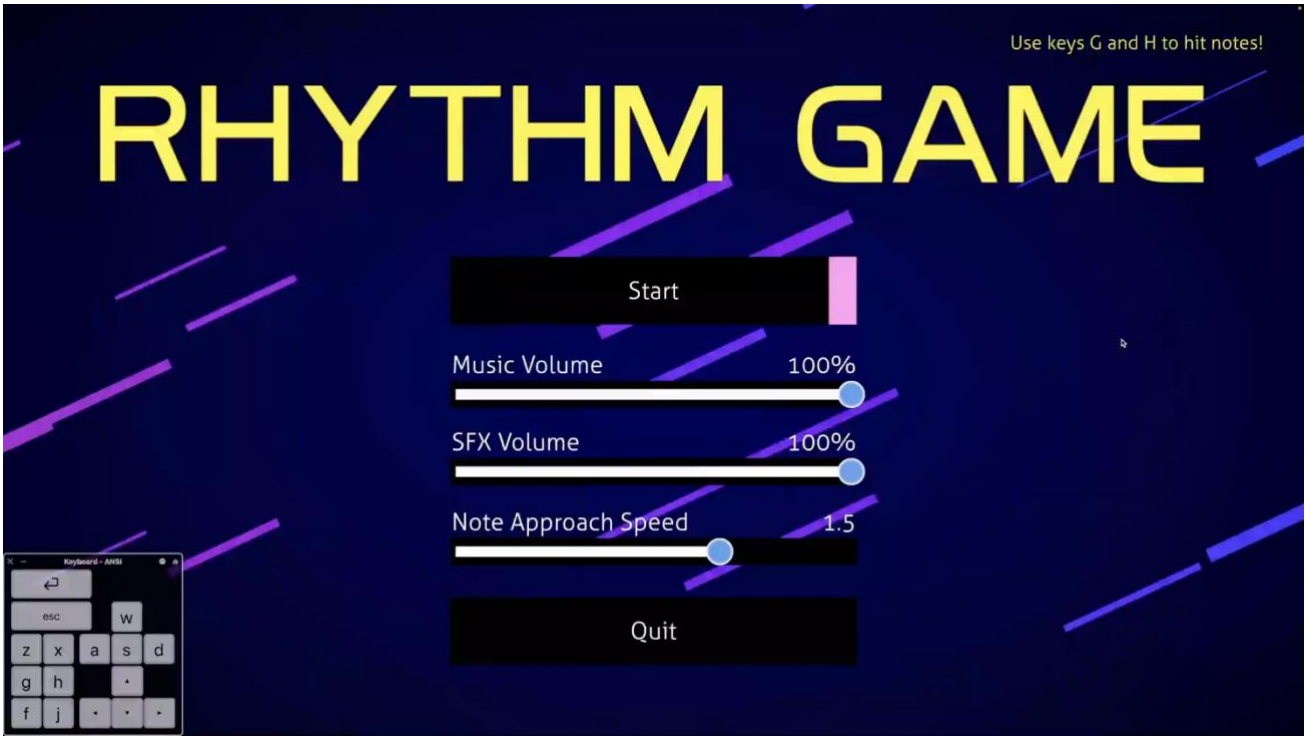


Future game maintenance would be possible because of the modular nature of the MIDI stages. For example, if stakeholders requested more stages, an update could be added with a scrolling list of stages, in place of the 5 current stages.

A limitation of the game I found after development was its storage size. When compiled for mac OS the total space need is 180MB, making the game more difficult to distribute because of how much space it took up on the disk, meaning it would be inaccessible to people with limited storage. If there was more time to develop the game further, finding a solution to this – such as optimising code – would be a high priority. Comparing to the estimated hardware and software requirements, playing the game would require more than 100MB of storage and likely more RAM and processing power than estimated at first.

Project showcase

The video below showcases the final version of the project and provides evidence for the post development tests below. The game was opened for the first time, and no previous save data had been created. In the bottom corner of the video there is an input display for the keyboard, showing all of the keys used. Video (with sound) below or at: https://youtu.be/JYiDZ_TiDus

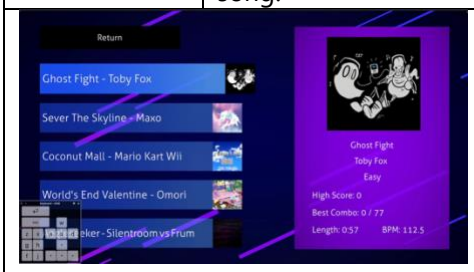


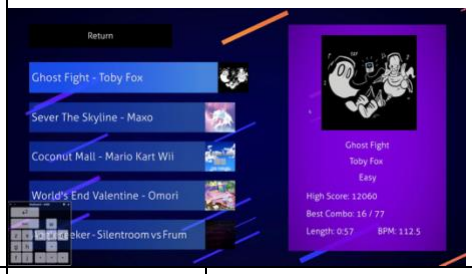
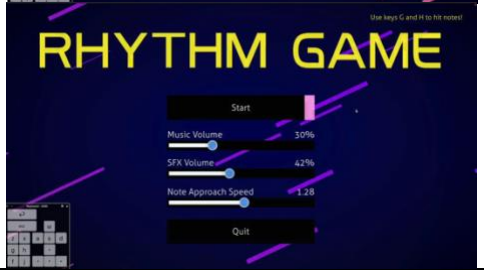
The web version can be played online at: https://flowerbath.itch.io/rhythm-game

Post Development Tests

In my post development tests, I have explained the functions of each feature and why they work or don't work successfully. The evaluation of each feature has been left to the evaluating success criteria section. All evidence is based on the video evidence above, and time stamps have been referenced.

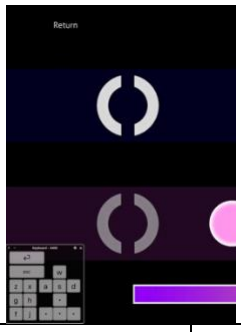
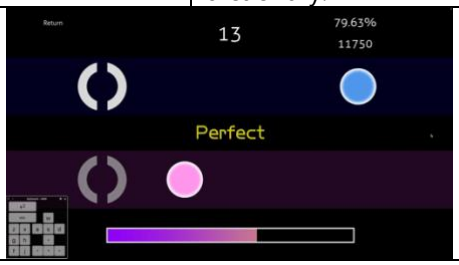
N.o	Feature	Metric to measure success of test	Test data	Acceptable test result	Justification
1	Start screen	A simple screen should display the title of the game and menu options to progress. Should be visually appealing and be in the style of the game.	Stakeholder feedback, Click buttons to change between screens	Pressing buttons transitions between screens. User can exit game.	Provides a starting point for the player.
	Evidence			**Pass/fail**	
	 Evidence at 0:03			Pass – Start screen is the first screen shown.	
2	Song	Display a menu where the	Stakeholder	Stage can be	User can

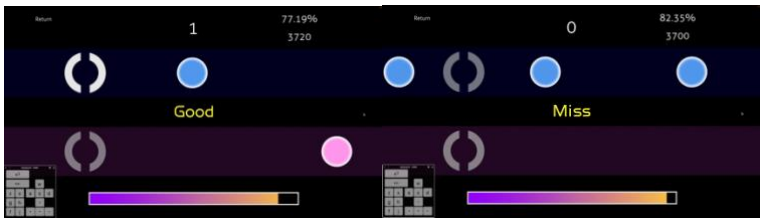
	selection screen	player can see all the songs should have basic information about the songs including difficultly and name. Should have a clear link to the leader board with a preview of the high score as the user hovers over the song.	feedback, Use arrow keys to change between selected stage, Click button to enter stage.	selected. Information about screen is shown. User can enter stage selected.	choose a song and enter the stage.
	 Evidence at 0:23 – use can use a mouse, w and s, or up and down arrow keys to navigate between the stages. To enter a stage enter or the mouse button can be pressed. Length, name, artist, bpm, high score and best combo are all displayed when a stage is selected.			Pass	
3	Tutorial stage	Should be the first stage the player should enter. Unlocks other stages of the game and pops up short instructions during the song on how to play.	Stakeholder feedback, enter tutorial from stage selection	Game mechanics explained. Progression after completion.	New players should have a brief introduction to the game.
	 Evidence at 0:03 – Tutorial text in the top right corner Evidence at 0:23 – Stage selection screen has stage difficulty			Fail – Whilst no tutorial stage was developed, there are two features to help new users, the tutorial text in the start menu and the stage difficulty in the stage selection menu.	
4	High score	Save the player's best score for a stage. The high score should carry over when the game is closed and re-opened. Should be non-volatile	Set high score values, Re-launch game	Same values displayed correctly when game is re-launched.	A competitive feature to encourage improvement

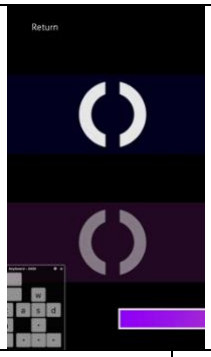
	 Evidence at 1:28 – high score set  Evidence at 2:11 – high score seen in the menu after game was closed and re-opened			Pass – Highscores can be repeatedly set. The score and combo are saved and can be viewed in the stage selection menu.	
5	Leader board	Holds information about a user's score, highest combo, rank, and accuracy for a stage. Displays separately for each stage in descending order of scores. The leader board should carry over when the game is closed and re-opened.	Set multiple score values, Re-launch game	Scores displayed correctly and in order.	An added level of complexity to high scores; A separate display for player scores.
	n/a			Fail – leader board not developed. Only a single high score is saved.	
6	Settings menu	A menu for changing sound settings and difficulty settings. Settings should carry over once the game is closed and re-opened.	Stakeholder feedback, Click buttons to change between screens, Change settings values with slider Re-launch game	Settings values update globally across the whole game. Settings values correct when game is re-launched.	A menu for changing settings lets users accessed their saved settings.
	  Evidence at 0:10 – All sliders can be dragged around and changed by arrow keys Evidence at 1:35 to 1:45 – After closing and re-opening the game the same settings are displayed			Pass – Settings menu integrated into main menu.	

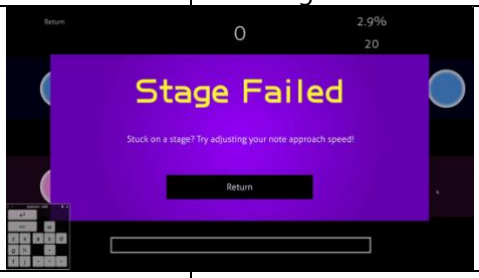
7	Sound settings	Sliders to control music volume, effects volume, and overall volumes. Sliders set volume percentages from 0 - 100%	Input volume percentages from the settings menu, test unity audio play volume level variable	Sliders change audio levels including music and sound effects volumes.	Necessary for users using different output devices.
	At 2:00 to 2:20 - the sound settings are changed and when a stage is entered, the volume changes.			Pass – full range of control on music and sound effects. All sound effects in the game change volume.	
8	Note approach speed setting	In the settings menu, players could the speed at which notes approach the hit area. How much time you have to hit the note would be compensated for if this setting was changed so that it only affects how fast you want the notes to appear.	Input note approach speed value from the settings menu, play a stage and repeat tests	Sliders change note approach speeds. Notes approach at a variable rate.	Additional settings make the game more accessible and in line with stakeholder needs.
		Evidence at 2:35 – 3:15 note speed changed from low value to high value and note approach speed changes in the game.		pass	
9	Difficulty settings	Changes health and hit area to increase or reduce difficulty. More health would make stages easier to pass and larger timing windows would make notes easier to hit. The reverse would apply for a difficulty increase.	Input difficulty float from the settings menu, test hit area	Difficulty float affects hit area proportionally.	Not necessary, but would help very new and very skilled players
	n/a			Fail - Difficulty settings were not developed, instead stages were designed with varying difficulty.	
10	Game screen	Game screen with a background, hit area, approach lanes, and a HUD (heads-up display). This is where all the stages will be played, and the game screen needs to be able to support all the different songs – including displaying oncoming notes and audio and visual effects from various sources.	Stakeholder feedback, Visibility of elements, Update frequency.	Contains UI that updates as the user plays the stage.	The environment in which the game takes place.

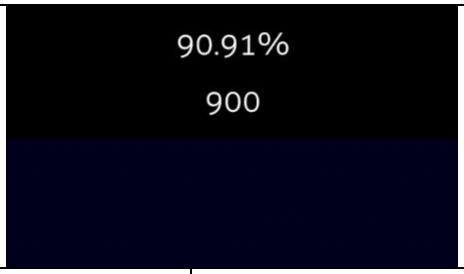
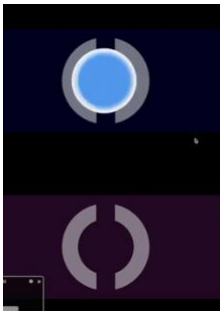
		Evidence at 0:28 to 1:22. UI elements update instantly as the stage is played. UI is consistent between stages (evidence at 2:15)		Pass	
11	Song complete screen	After a song has been finished and if the user has not failed, the screen should show all the statistics, including score, combo, accuracy, and rank. The user should then be prompted to return to the song selection screen.	Stakeholder feedback, input out of bounds variables: score, combo, accuracy, and rank Can be reached from completing song only, input enter key to exit.	Positive reception about style, displayed immediately after stage, correctly displayed all variables, key press exits screen to selection screen	Concludes a stage and lets user see their results.
		Evidence at 1:25. Stage complete screen shows on completion,		Pass	
12	Multiple stages	In the stage selection screen, the user should be able to choose from at least 3 stages. These stages should have different note arrangements and should vary in. song choice difficulty. The length should be at least 2 minutes for an effective and enjoyable stage.	Stake holder feedback, enter each stage from the stage selection screen and play through to the end.	Stages are enjoyable to play. All notes spawn correctly, and statistics are shown accurate.	All stages that are developed for the game should be tested to ensure that they can be completed.
		Evidence at 0:30, 2:30 and 3:05. To keep the video brief not all stages were shown to their full extent.		Pass – in total 5 full length stages were developed.	
13	Hit area	A rectangle outlining where the notes should be hit, positioned on the left of the screen. Once notes go too far out of this area they are destroyed, and the player gets a miss. The hit area will	Stakeholder feedback, change leniency variables, hit notes on time, hit notes in leniency area, hit notes outside of	Positive reception about timing difficulty, leniency changes hit area size, notes can be hit in each	Needed to show the player where to hit the notes

		be composed of several sub sections expanding outwards, allowing for timing leniency to get a late or early hit, depending on the difficulty set.	leniency area,	zone, with different scores gained	
		Evidence at 0:28 to 1:22. and at 3:14 to 3:30. Notes hit inside the hit area are always registered, hit area size adjusts with varying note approach speeds.		pass	
14	2 object lanes	Two areas where the notes will come from, both horizontal and moving from left to right. One lane will be at the top of the screen and one at the bottom. The player will effectively move between them as they hit the notes.	Stakeholder feedback, Input note array with a note each second, from time = 0 to end time,	Positive reception about lane size, input note, note array fully displayed in lanes	Creates 2 separate areas for the player to hit the notes, needed to display the notes.
		Evidence at 0:28. Notes spawn in both object lanes throughout the whole stage		pass	
15	Hit objects (Notes)	Circular objects to be hit by the player. Spawned from the same position in one of two lanes and approaching the hit area at a constant speed. Player inputs a press to destroy the notes and get score and combo from this, whilst also being judged on their timing. Hit objects are abstracted notes of the song, generated from a dictionary.	track note sizes, object spawn location, object destroy location, object hit response, object miss response, object, score, input note hit.	Hit objects spawn according to the set stage, score changed with hit/miss, input note hit on time always destroys note.	Hit objects are essential parts of the game and are needed for the user to play the game.
		Evidence at 0:28. Objects move smoothly and are destroyed when reaching the end or being hit.		Pass	

16	Long beat objects	In game objects that act similar to hit objects but lasting for longer. The player must hold the key down for the duration of the note to get the full score.	Input key for entire duration, input key late, input key early, end key input late, end key input early	Results give perfect score, late gives less score, early gives a miss, end late gives a miss, end input gives less score.	Allows for more dynamic stages.
	n/a			Feature not implemented. Explained in success criteria evaluation.	
17	Hit input	The player will have two keys that they can press to register a hit. These will correspond to the top and bottom lanes and are used to hit approaching objects.	Input upper hit, input lower hit inside and out of stage	Both respond with hit accuracy, when in stage. Outside of stage no score changes, inside score changes with notes hit	Needed so that the user can hit the notes on the beat.
	 Evidence at 0:28 to 1:22. Notes hit late early and on time.			Pass – multiple different key binds for accessibility. Valid and invalid cases tests. When no input made (invalid) note was missed)	
18	Hit sounds	Hit sounds will occur whenever a hit input is detected and will tell the player they have inputted it successfully.	Input hit on note	Sound plays	Game should respond to the user's inputs with audio outputs.
	Evidence at 0:28 to 1:22. Hit sounds are played on a 'good' or 'perfect' hit			Pass	
19	Hit effects	Hit effects will occur whenever a hit input is detected and will tell the player they have inputted it successfully. To do this, the respective hit area will glow for the duration of the key press.	Input hit on note	Effects play on lane position	Note objects should respond to hit inputs to make to make the user feel more involved with the music.

		Evidence at 0:28 to 1:22. Hit effects happen on the correct lane.		Pass – hit effects were changed to take effect whenever the user presses a key instead of just acting on the note.	
20	Animated background	Background playing in conjunction with the game, either moving or repeating a simple effect in time with the beat of the music.	Stakeholder feedback, Enter stage, exit stage	Background appears during gameplay, stage selection and main menu, without being too intense.	A background would make the game more interesting for my stakeholders
		Evidence at 1:50 to 2:10 – Background animates on enter game and entering stage selection. Background restarts when scene reloaded.		Pass – background changed to be on main menu and stage selection screen instead of the game screen.	
21	Score	A live counter with the player's points, which are gained through higher combo and hitting notes, with a reduction in score if the hit is miss timed.	Enter stage, input all hits, input all misses	Score updates with any hit inputs and misses	A player's performance should be measured with a numerical value.
		Evidence at 0:28 to 1:22. Score updates after each hit. Less score is given for 'goods' and no score given for misses.		Pass	
22	Combo	A counter incrementing after each successful note hit, resetting to zero if a note is missed. Used for calculating combo.	Enter each stage, alternating hits, and misses	Combo updates with hits and misses	A combo should let the user know how many notes they have hit in a row, in order to give score multiplier.

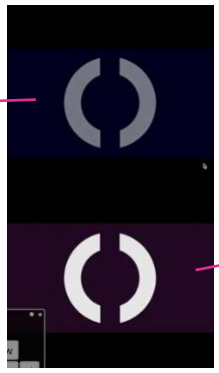
			Evidence at 0:28 to 1:22. Combo increments by 1 with each hit. Combo resets to 0 with a miss.	Pass	
23	Health bar	A bar that depletes if the user misses too many notes. Each miss takes a percentage off the health bar, meaning too many misses causes the player to hit zero health and fail the map.	Enter each stage, input score, input miss	Health updates with any hit inputs and misses	Needed as a measure for the player to see if the stage was too challenging.
			Evidence at 2:12 to 2:35 Health decreases are each miss is detected. Health unchanged after hit input.	Pass – Health is validated after each miss and at the start of the level.	
24	Song failed display	After depleting the health bar, the user fails the stage and is given a clear, failed message. All game logics stops and fades away. From here they either get the option to retry or go back to the menu. Text should appear in the centre of the screen for most readability. All gameplay stops, including music.	Stakeholder feedback, Enter stage, input miss, let health go to 0.	Game transitions to this display whenever health goes to zero.	Important feature to visual show a song has been failed, and to encourage the user to try again.
			evidence at 2:35 to 2:45 – When health hits 0 stage failed sound effect plays and stage failed windows shows.	Pass – user can exit by pressing return key (or escape on the keyboard)	
25	Accuracy	Accuracy text shows up under the score as a constantly updating percentage (out of 100%). 100% accuracy being fully perfect timing and 0% being every note was missed.	Enter each stage, alternating hits, and misses	Accuracy updates with hits and misses.	Should provide additional and accurate information to the user.

			Evidence at 3:20 to 3:44. As hitting or missing notes varies, so does accuracy. Accuracy also falters with 'good' timings	Pass	
26	Rank	Part of the song complete screen – assigns the player a rank based on score after a song is complete. Ranks going from C to B to A to S to S+	Enter each stage, alternating hits, and misses	Rank calculation displayed accurately.	Adds a lot of interest to the game and encapsulates score and accuracy.
	n/a			Feature was not developed.	
27	Timing indicators	After each note, text displays depending on how early or late the user's press was. Either a miss, good or perfect is displayed.	Enter each stage, input late hit, input early hit.	Timing indicators appear clearly after each hit.	Important for the game to feel responsive and for the player to know how well they are playing.
	  Evidence at 3:20 to 3:44 - When the note is timed exactly to the beat, a perfect judgement is given. When it is just early or late, good is given and too early or late, a miss.			Pass – Timing indicators are given for all notes and are always within a set time frame, making the feature robust.	

Evaluating Accessibility Features

My primary concern when developing the game was for users who can have seizures or discomfort caused by sensory overload, known as photosensitivity or photosensitive epilepsy. Although developing in the way would prevent harm from affecting these users, I also hoped that it would make the game more enjoyable and cause less eye strain for all users. Strobing (bright flashes), in my game, occur in one area of the game, when playing a stage, and are a result of the hit effects feature, where the hit area lights up on a key press. I had no metric to measure what threshold would be dangerous, and so I chose to keep this effect as subtle as possible. The brightness is contained to a single area of the screen, and the effect doesn't fade all the way to black. Additionally, the effect is set to fade out slowly, over about a second, to prevent the effect from being too intense. When testing the feature, I made sure that rapid hits on top of each other wouldn't stack the effect and cause a strobe, instead the animation resets and the colour change is less noticeable.

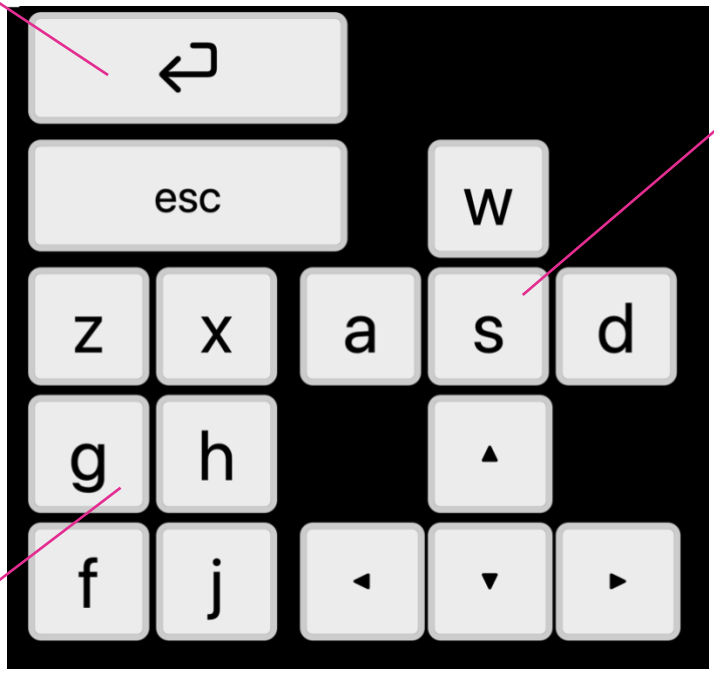
Default sprite colour is dimmed to a grey.



After an input the sprite fades from white back to grey.

Another limitation of the project accessibility feature that I was not able to develop was a re bind key setting to let players change the controls of the game. I felt that whilst this was important, I would have been limited by my Unity knowledge to implement a method to change your control scheme for all possible controllers / input methods. Instead, my game aims to let users customise their play style with 3 set of 2 keys that can be used. As stated in the analysis, the primary appeal of the game is still that it needs only 2 keys to play the game (the mouse or other keys need to be used to control the UI, but unlike the gameplay this can be done at the user's own pace). In total the following key binds were developed:

Escape and enter are both shortcuts for progressing or backtracking through the game. With these and the arrow keys, it is possible to navigate the game completely without a mouse, which is crucial for users who are unable to use a mouse.



'WASD' and arrow keys are used for menu navigating, with both being interchangeable for the other, and both having the ability to control sliders as well as select buttons. I chose these two as both are standard control in many other video games.

Z,X ; G,H ; and F, J are all pairs of inputs that can be used for the top and bottom lanes when playing the stages, giving the user options for controls.

Evaluating Usability

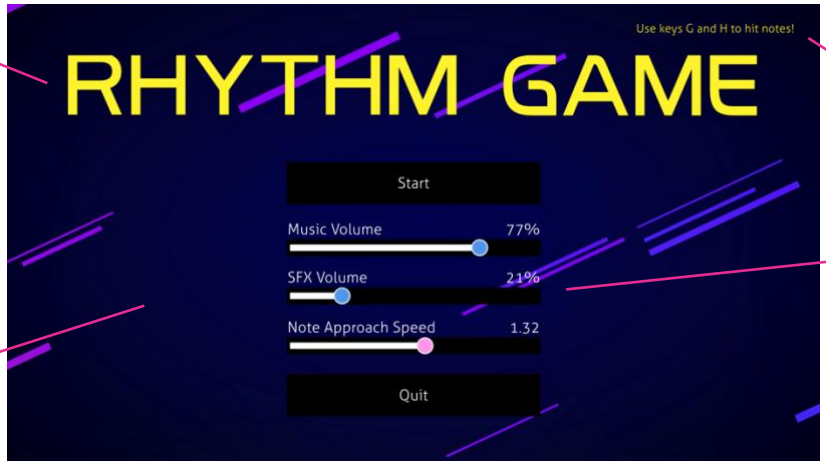
From all the tests conducted so far, there were no glitches or bugs in the game, and the game ran smoothly without any issues.

Stake holder feedback

The final testing of the game would be done by stakeholders. I let stake holders play the game for as long as they wanted and then asked for feedback, first in the form of questions to my two main stake holders and then inviting more participants to give overall, more general feedback. The wider audience consisted of my original two stake holder, plus an additional 3 people I invited to test the game.

The testers were asked to consider each element based on how well it fit the theme of the game, how successful it targeted their demographic (all testers were a part of my target demographic) and how engaging or interesting it was to them. This included how users interacted with the game, their persona needs and any other thoughts.

Main menu (feature 1)



Game title is highlighted in yellow and uses a bold and modern typeface.

Animated background is colourful, appealing to a younger audience.

Tutorial text is also yellow, making it clearly visible.

Settings sliders are centrally positioned for ease of use.

Farhan: The menu buttons are very easy to use and intuitive and I had no issues reading any text. Having the background being animated is a great feature too. I'd say it's a very universal starting screen and can appeal to any player, young or old.
Magda: The design is clean looking; I like the background and the different button state sprites to show a button is highlighted, it's really useful to see what button I am about to press. For a younger target audience this would be appealing.

The main menu successfully introduces the user to the game and provides a starting point. Whilst nothing an about the game is revealed, the payer can infer from the tutorial text and title how the game will play out. The stakeholders agreed that the design was easy to use and well-constructed. The active sprites for the buttons and slider also helped the players a lot when navigating the menu. Most importantly the title screen is simple and lets the user continue to the game.

Sound settings (feature 6)

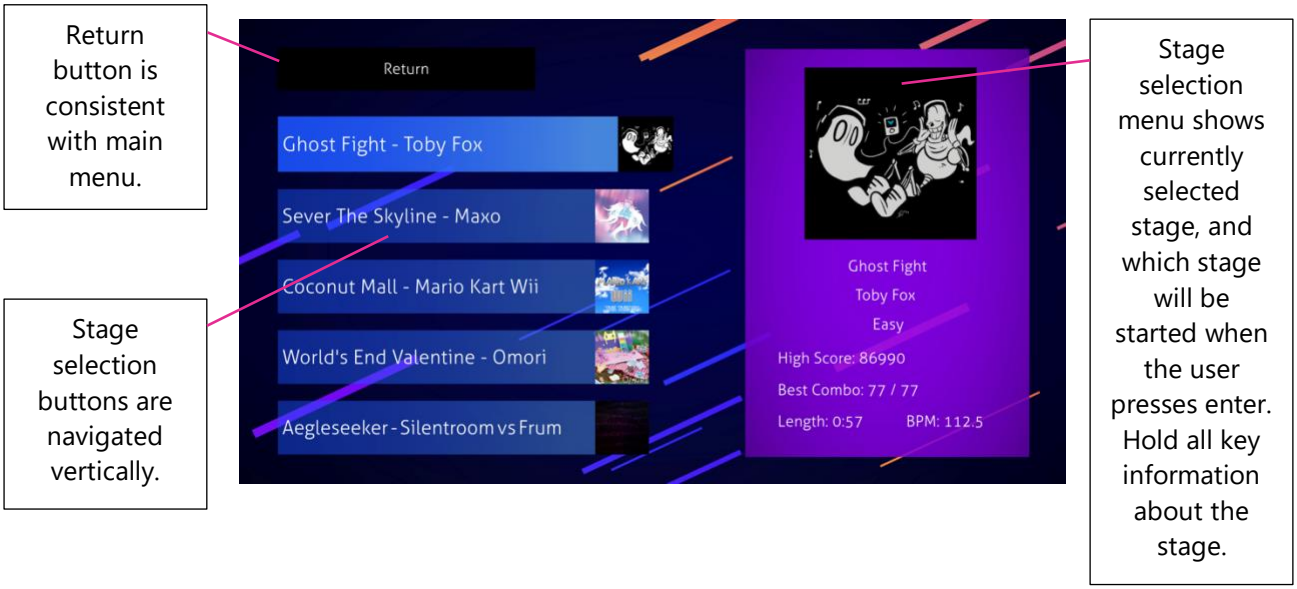
Farhan: very useful, intuitive.
Magda: I didn't need to use them.

Note approach speed (feature 8)

Farhan: Useful for easy stages and less useful for harder stages. Would have been good if there was a tutorial explaining it.

Magda: This feature was highly requested and is definitely a must have.

Stage selection screen (feature 2)



Farhan: Stage selection buttons are very easy to use and intuitive. The stage selection info panel thumbnail is perfect for quickly seeing which stage will be entered next. It is very clear and readable. The plain designs don't subtract from the experience – the panels are modern and can be understood by any audience.

Magda: I like how the background is also on this screen, it adds a lot to the atmosphere. Navigating between stages is super easy with arrow keys and the sound effect makes it obvious. My only suggestion would be that song selection could be saved so the same stage is selected when entering the screen, for example to retry a song faster.

The stage selection screen successfully provides a menu for the user to view all the songs and see all their high scores in one place. The stages come ordered in difficulty and all information about the stage is abstracted into the info panel. The thumbnail next to each stage makes them recognisable for the future. From stake holder feedback the screen has all the basic functionality needed, is clear and easy to navigate.

High scores (feature 4)

Farhan: The game is very replay able because of high scores.

Magda: high scores are not that important to me but combos help me track my performance. Good feature, I like being able to save my best scores and combos.

Difficulty selection (feature 9)

Farhan: Good difficulty range between the 5 stages.

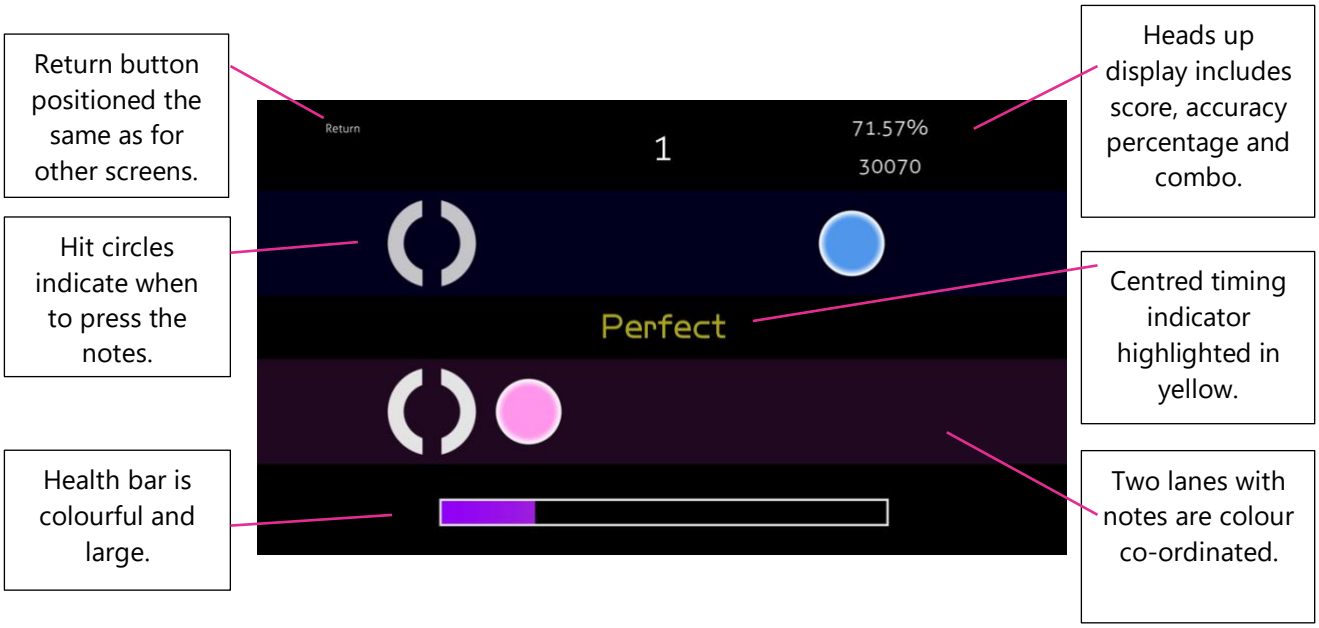
Magda: Excellent difficulty range .

Stage design (feature 12)

Farhan: Pretty good. The stage length is not too important to me. The song choices are very recognisable - helps indoctrinate new players.

Magda: All of the stage designs are great. The last one has awkward note placements at times.

Game screen (feature 10)



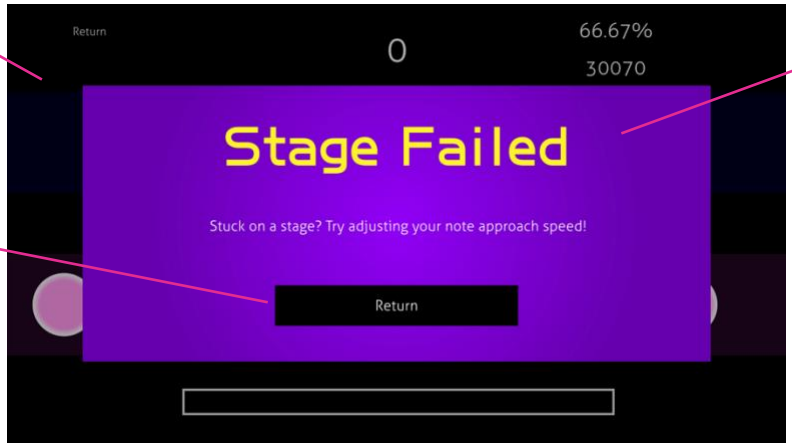
Farhan: The return button is a nice feature encase you accidently enter a stage without meaning to. Otherwise, all the elements work well together, and everything can be seen easily at a glance.
Magda: I think the game screen is very nice, it works well with the two colours, and fits in consistently with the colour scheme. The notes aren't too bright, and the white outline makes them standout enough. Hit circles lighting up is a really good features for orienting yourself as your player, for example if forgot which keys you need to press. Health bar position is really clear and can be. Everything else in the HUD is also easily visible, although the return button is a little small.

The game screen abstracts away all the information needed for the player to know how well they are doing into a heads-up display. There are only 3 numbers to keep track of for the user, with combo being g the clear most important one. Next the health bar abstracts the health variable away. A player is unlikely to want to know exactly how much health they have left and so health is only shown visually. The Overall the layout was suited to the stakeholders needs, with only small changes requested.

Sync to music (Feature 17)
Farhan: Flawed, some button presses didn't register. When they did the hits were accurate.
Magda: I like it, I didn't notice any problems. The game is highly responsive.

Timings (Feature 27)
Farhan: The timings for perfects and goods are nice.
Magda: The timings for perfects and goods are excellent. I like how the game distinguished between perfects, good and misses with different accuracy and score, but combo is inclusive of goods.

Stage failed (feature 24)



Dimmed and paused background.

Return button takes user back to stage selection screen.

Stage failed window appears with fail sound effect when health hits 0.

Farhan: The Stage failed display UI could be better, but the simple design works well with the look of the game and hopefully the user won't see this screen much. The encouraging tip is a nice addition.
Magda: There is nothing wrong with it, it's good, although a little plain. I like how you can still see the paused game behind it. The navigation works perfectly in this interface.

The stage failed screen successfully informs of the user of when they have run out of heath without completely taking them out of the game. This is thanks to the background dim and the stage failed sound effect that makes the transition less abrupt. Stakeholders commented on the simplicity of the UI, which could be improved to be made more interesting. The navigation worked as intended and the stage selection screen can be returned to easily. The added tip in the middle makes the screen more user friendly.

Sound effects (Feature 18)
Farhan: The stage failed sound effect isn't super noticeable. The other sound effects are good and make the game environment more responsive.
Magda: Sound of notes being hit is satisfying. Also, the stage selection sound is fun.

Stage complete (feature 11)

Return button takes user back to stage selection screen.

Informs the user of a new high score if set.

Combos and scores are compared to past bests.

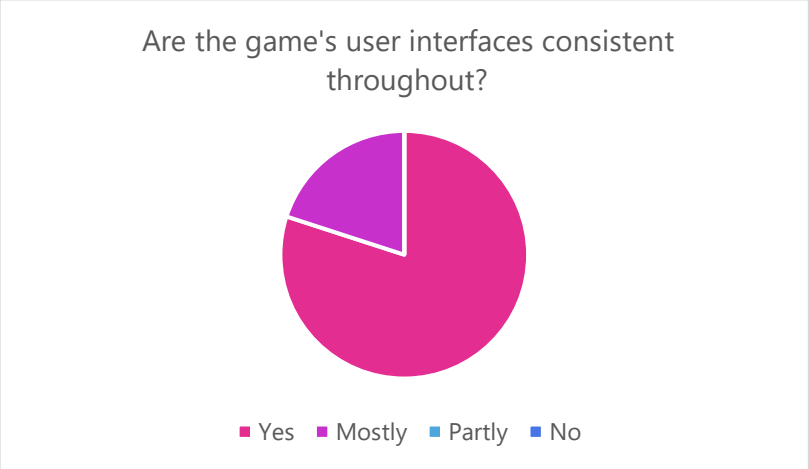
Stage complete shows 2 seconds after all notes have finished spawning.

Statistics for timings give a more detailed summary of performance.

Farhan: Stage complete info panel is very good at conveying information but is lacking uniqueness with the colours. Ease of use is unequivocally perfect, especially with the return button being bound to the escape key.
Magda: I like the stage complete screen. It has all the necessary information and is important for seeing how well the player has done. The yellow on top of the purple works well for the colours.

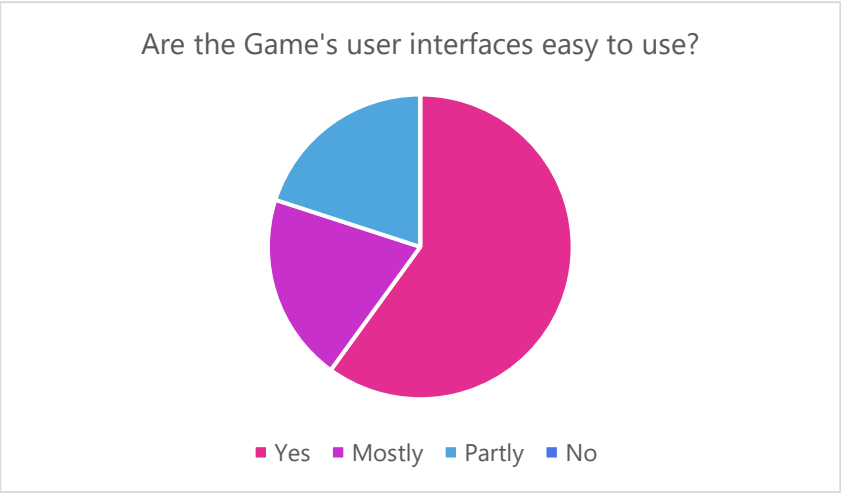
The stage complete screen appears at the end of the level. Play testers thought the transition to the stage complete screen was good and that all the text was very readable. The layout was easy to understand, and the colours separated the elements well. Features such as the text to let the user know they achieved a high score and the statistics to compare to their previous best were also a good addition. Stakeholders agreed that only valuable information was shown, which is vital for experience rhythm games to see where to improve, whilst not overwhelming new players.

User interface consistency
Farhan: Mostly consistent, the return buttons are the same, but some parts are not as consistent as could be.
Magda: Yes, colour scheme and fonts are consistent between screens.
Wider audience:



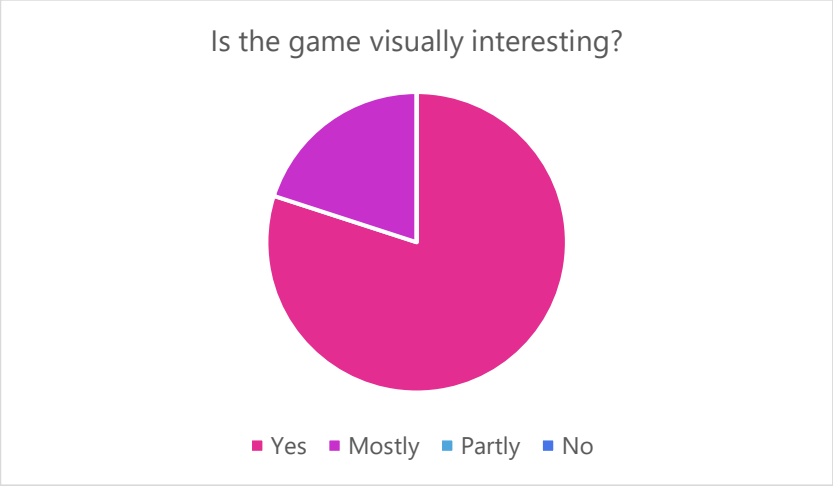
Interfaces are on the whole consistent, and all together push the game's theme and look. The modern, geometric, and vibrant look between the interfaces emphasises the fast-paced nature of the game and targets the 12-18 demographic that would enjoy the drum based and electronic music chosen fro the project. The game is established as rhythm game in this way.

Is the UI easy to use
Farhan: Yes it's easy to follow. I like the controls.
Magda: Yeah, even though there is no tutorial, controls are easy to figure out if you have played video games before.
Wider audience:



Stakeholder feedback showed that adding the escape key was a useful feature for navigating the menus, as well as the addition of the arrows keys and enter button. Play testers did not have trouble using or discovering the controls, other than some players not discovering the use of arrow keys in the stage selection menu. Limiting the game to only keyboard inputs instead of mouse would have made this clearer. Other users enjoyed the versatility of choosing between mouse and keyboard. Animations for selected buttons also greatly helped the UI feel more intuitive. A flaw in the UI was that in the stage selection screen information panel, the text on the side was not explained, so new rhythm game players were unsure what best combo and bpm meant, as well as the artist / song name / difficulty not being clearly labelled.

Is the game visually interesting?
Farhan: Yes, although the colours are not super important to me, the game is eye catching and colourful. It looks addictive.
Magda: Yes. The fonts are very aesthetically pleasing and fit the style very well.
Wider audience:



From my stake holder feedback the game successfully targets the 12-18 age group, people looking to get into rhythm games and veteran rhythm game players. As some stake holders pointed out the visual aspect of the game are not the most important feature. Still, the animated background and engaging colour scheme contributed a big part to the game's appeal to people looking to get into rhythm games.

Lastly, what improvements could have been made, and what would you want from the project in the future?
Farhan: As a single player rhythm game there's not much else that could be added other than more types of notes and more stages to play. Possibly exploring other methods of timing beats to the song, such as generating them algorithmically from analysing music or displaying them differently.
Magda: Yeah, I'd also add that expanding the game to have leader boards stored on a data base remotely with a login system so that players from anywhere can complete for scores would be pretty neat, but all of those suggestions are really far reaching. Some small things that could be added are sound effects for all the menu buttons, menu background music and more stage options.

Evaluating Success Criteria

To evaluate the success criteria, all the previous tests will be looked at, as well as the previous section where stake holders gave feedback about each feature and the overall style of the game.

The completion column is colour coded according to how much of the feature was completed.

- Fully completed
- Sufficiently completed – feature changed slightly in some way
- Partially completed – feature changed dramatically in some way
- Incomplete

The priority column has been left in the original colour to show how much of the stage actually completed, compared with how much I expected to complete. Where the left and right colours match priority was met.

N.O	Feature	Description	Reference + Justification	Priority
1	Start screen	On opening the game, a simple screen prompting the user to enter, exit or go into settings of the game as buttons. Should have the title of the game displayed in a large font, and simple graphics for the background.	Game 1 analysis showed this was a needed starting point for the player.	Top
	Completion	**Evidence**	**Evaluation**	
	Fully completed	Evidence in test 1 and stake holder feedback.	The start screen successfully introduces the user to the game and provides a starting point. Whilst nothing an about the game is revealed, the payer can infer from the tutorial text and title how the game will play out. The stakeholders agreed that the design was easy to use and well-constructed. The active sprites for the buttons and slider also helped the players a lot when navigating the menu. Most importantly the title screen is simple and lets the user continue to the game.	
2	Song selection screen	A menu where the player can see all the songs, should have basic information about the songs including difficultly and name. Should have a clear link to the leader board with a preview of the high score as the user hovers over the song.	Game 1 analysis showed that songs should be navigated to with a linear list and song details should be abstracted away.	top
	Fully completed	Evidence in test 2 and stake holder feedback.	The stage selection screen successfully provides a menu for the user to view all the songs and see all their highscores in one place. The stages come ordered in difficulty and all information about the stage is abstracted into the info panel. The thumbnail next to each stage makes them recognisable for the future. From stake holder feedback the screen has all the basic functionality needed, is clear and easy to navigate.	
3	Tutorial stage	The first stage the player should enter. Unlocks other stages of the game and pops up short instructions during the song on how to play.	The Interview covered that new players should have a brief introduction to the game.	High – priority not met
	Partially completed	Evidence in test 3 and stake holder	No tutorial stage was added and the game features no progression and players can play stages in any order. Instead,	

		feedback.	difficulty ratings are displayed on each stage and tutorial text is added to the start screen, which is all that users need to start playing the game. In my opinion this is more successful than a tutorial stage would be because it doesn't affect experienced players ability to enjoy the game.		
4	High score	The player's best score for a stage. The high score should carry over when the game is closed and re-opened.		The interview showed that a competitive feature is wanted.	Medium – priority exceeded
	Fully completed	Evidence in test 4 and stake holder feedback.	Highscores add a lot of replay value to the game, letting the user replay stages with the aim of improving their high score or best combo. Stakeholders of all skill levels found saving high scores to be a useful feature. For user training their music ability, the feature is a crucial metric.		
5	Leader board	Holds information about a user's score, highest combo, rank, and accuracy for a stage. Displays separately for each stage in descending order of scores. The leader board should carry over when the game is closed and re-opened.		An added level of complexion to high scores; the games 1 and 2 analysis showed that a separate display for player scores was useful but cannot be justified to be important for a small game.	low
	Incomplete	n/a	No leader board feature was added due to time restraints and the development decision to focus the game for single player users.		
6	Settings menu	A menu for changing sound settings and difficulty settings. Settings should carry over once the game is closed and re-opened.		Games 1 and 2 analyses showed a menu for changing settings was needed	high
	Sufficiently complete	Evidence in test 6 and stake holder feedback.	The settings menu was developed into the main menu. From stake holder feedback the settings menu was intuitive, and the sliders were easy to use.		
7	Sound settings	Sliders to control music volume, effects volume, and overall volumes. Sliders set volume percentages from 0 - 100%		Games 1 and 2 analyses showed that all three volumes settings were necessary for various output devices.	High - priority exceeded
	Fully completed	Evidence in test 7 and stake holder feedback.	The sound settings – separated into music and effects – are a universal feature in game and add customisability to the game. Having the setting be saved when the game is closed and re-opened is important for players who don't have much time and want to be able to start playing the game instantly.		
8	Note approach speed setting	In the settings menu, players could the speed at which notes approach the hit area. How much time you have to hit the note would be compensated for if this setting was changed so that it only affects how fast you want the notes to appear.		The interview and game 2 analysis showed that whilst this would make the game more accessible, but due to limitations may not be	Low - priority exceeded

				needed.	
	Fully completed	Evidence in test 8 and stake holder feedback.	Note approach speed settings were originally planned to have a greater range, however, to be accessible to new players, the setting was validated between two similar values, with enough range for experienced player. The user is encouraged to experiment with the setting in the failed screen.		
9	Difficulty settings	Changes health and hit area to increase or reduce difficulty. More health would make stages easier to pass and larger timing windows would make notes easier to hit. The reverse would apply for a difficulty increase.		The interview and game 2 analysis showed that this feature is not necessary but would help very new and very skilled players.	low
	Incomplete	n/a	Difficulty settings were not developed; the feature is not common in rhythm games and could deter new players, however making the difficulty setting unlockable as players progress would have overcome this limitation.		
10	Game screen	Game screen with a background, hit area, approach lanes, and a HUD (heads-up display). This is where all the stages will be played, and the game screen needs to be able to support all the different songs – including displaying oncoming notes and audio and visual effects from various sources.		Games 1 and 2 analyses showed that this was a necessary environment for the game.	top
	Fully completed	Evidence in test 10 and stake holder feedback.	The game screen abstracts away all the information needed for the player to know how well they are doing into a heads-up display. There are only 3 numbers to keep track of for the user, with combo being g the clear most important one. Next the health bar abstracts the health variable away. A player is unlikely to want to know exactly how much health they have left and so health is only shown visually. The Overall the layout was suited to the stakeholders needs, with only small changes requested.		
11	Song complete screen	After a song has been finished and if the user has not failed, the screen should show all the statistics, including score, combo, accuracy, and rank. The user should then be prompted to return to the song selection screen.		Games 1 and 2 analyses showed that this was the best way to conclude a song and the interview confirmed having statistics was useful.	High - priority exceeded
	Fully completed	Evidence in test 11 and stake holder feedback.	The stage complete screen appears at the end of the level. Play testers thought the transition to the stage complete screen was good and that all the text was very readable. The layout was easy to understand, and the colours separated the elements well. Features such as the text to let the user know they achieved a high score and the statistics to compare to their previous best were also a good addition. Stakeholders agreed that only valuable information was shown, which is vital for experience rhythm games to see where to improve, whilst not overwhelming new players.		
12	Multiple stage designs	In the stage selection screen, the user should be able to choose from at least 3 stages.		Stakeholder feedback showed that having	High - priority

	(levels)	These stages should have different note arrangements and should vary in. song choice difficulty. The length should be at least 2 minutes for an effective and enjoyable stage.	multiple stages to choose from make the game more replay able.	exceeded
	Fully completed	Evidence in test 12 and stake holder feedback.	Five stages were completed in total, two more than planned originally. This gave the game a greater range of difficulty options to choose from, and more song choice. Stake holder feedback showed that the stages were interesting and uniquely designed, making the game fun to play. The choice of songs was the same as planned – songs from popular videos games, with a focus on drums since the game mechanics works on timing hits to the beats of the song. The stages were designed successfully.	
13	Hit area	A rectangle outlining where the notes should be hit, positioned on the left of the screen. Once notes go too far out of this area they are destroyed, and the player gets a miss. The hit area will be composed of several sub sections expanding outwards, allowing for timing leniency to get a late or early hit, depending on the difficulty set.	Games 1 and 2 analyses showed that this makes the game feel more natural to play.	top
	Fully completed	Evidence in test 13 and stake holder feedback.	The hit area was developed into. Two hit circles or indicators that act as guides for the user to hit the notes. The timings for when to hit the notes are centred around these hit areas, making them vital for the gameplay. Stake holders agreed that the functionality for the hit area was useful and fair, however that the design could be improved.	
14	2 object lanes	Two areas where the notes will come from, both horizontal and moving from left to right. One lane will be at the top of the screen and one at the bottom. The player will effectively move between them as they hit the notes.	Game 1 analysis and the interviewed showed that using 2 input keys and visually dividing the screen created an effective way to hit notes.	top
	Fully completed	Evidence in test 14.	As proposed, the game was designed specifically for playing with 2 keyboard inputs. Whilst more were added later in development for accessibility, the two object lanes add interest to the stages developed and allow for horizontal scrolling music beat notes.	
15	Hit objects	Circular objects to be hit by the player. Spawned from the same position in one of two lanes and approaching the hit area at a constant speed. Player inputs a press to destroy the notes and get score and combo from this, whilst also being judged on their timing. Hit objects are abstracted notes of	Hit objects are essential parts of the game screen, from games 1 and 2 analyses.	top

		the song, generated from a dictionary.			
	Fully completed	Evidence in test 15 and stake holder feedback.	Hit objects are needed for the player to know when to hit the notes. These are spawned from the right side of the screen and disappear on the left. From stake holder feedback, the notes are clearly visible and always appear in sync with the music. Notes correspond to beat, but are not algorithmically generated, instead are placed designedly with MIDI files. This makes the feature successful at creating interesting and modular stages, that can be then algorithmically synced to the music.		
16	Long beat objects	In game objects that act similar to hit objects but lasting for longer. The player must hold the key down for the duration of the note to get the full score.		Game 1 analysis showed that this is a dynamic edition to add variation that can be reused in various stages. The feature is complex and not needed.	Medium – priority not met
	Incomplete	n/a	The original proposal for this feature was ambitious and as the game developed it became clear that the gameplay in itself was complex enough to add interest without needing long hold notes. Due to time restriction, long hold notes were not developed, however if the feature was developed in the future, the feature would be added in as described in Iteration 4 Evaluation.		
17	Hit input	The player will have two keys that they can press to register a hit. These will correspond to the top and bottom lanes and are used to hit approaching objects.		The interview and games 1 analysis showed that a 2 key input method would be most suited to my project and let the user input hits freely.	top
	Fully completed	Evidence in test 17 and stake holder feedback.	The game was designed specifically for playing with 2 keyboard inputs and more were added later in development for accessibility. Stakeholder feedback showed that the hit inputs were responsive and that notes were always destroyed on time. Hit inputs successfully and reliably determine the outcome of the user's performance in a stage.		
18	Hit sounds	Hit sounds will occur whenever a hit input is detected and will tell the player they have inputted it successfully.		games 1 analysis showed the game responding to a key press helped the player's co-ordination.	High - priority exceeded
	Fully completed	Evidence in test 18 and stake holder feedback.	Sound effects in the game successful add a sense of control and impact to the user's inputs. My rhythm game is made to replicate a drum beat for players to match the rhythm two, and so adding a drum sound effect makes the game more immersive. Additional sound effects such as miss, and button sound effects were also added. According to stakeholder feedback, the sound effects add to the overall game feel.		

19	Hit effects	Hit effects will occur whenever a hit input is detected and will tell the player they have inputted it successfully. To do this, the respective hit area will glow for the duration of the key press.		games 1 and 2 analyses showed the game responding to a key press was a useful feature and improved graphics	Medium - priority exceeded
	Fully completed	Evidence in test 19 and stake holder feedback.	From stake holder feedback, the hit effects have a positive impact on the game, despite being toned down in development (see evaluating accessibility features). The glow of the hit area as a part of the hit effects is a successful indicator to the user to know when a key has been pressed.		
20	Animated background	Background playing in conjunction with the game, either moving or repeating a simple effect in time with the beat of the music.		games 1 and 2 analyses showed that a moving background was not necessary, and a simple graphic would not subtract from the game.	Low - priority exceeded
	Fully completed	Evidence in test 20 and stake holder feedback.	The background's purpose to make the game look and feel more modern, geometric, and colourful, using Unity's random particle generation all succeeded in making the game's ambient state feel alive and interesting. It was repeatedly mentioned in stake holder feedback as helping tie the menu screen and the stage selection screen together.		
21	Score	A live counter with the player's points, which are gained through higher combo and hitting notes, with a reduction in score if the hit is miss timed.		The interview along with both game analyses showed that seeing how well the player has performed is necessary for a rhythm game.	top
	Fully completed	Evidence in test 21.	The score successfully keeps track of the user's progress in the stage, is readable and can be used to set high score and measure performance.		
22	Combo	A counter incrementing after each successful note hit, resetting to zero if a note is missed. Used for calculating combo.		The interview along with both game analyses showed that this feature is always needed in a rhythm game.	top
	Fully completed	Evidence in test 22.	The combo successfully keeps track of the user's progress in the stage, is readable and can be used to set best combos and measure performance.		
23	Health bar	A bar that depletes if the user misses too many notes. Each miss takes a percentage off the health bar, meaning too many misses causes the player to hit zero health and fail the map.		Game 1 analysis showed that a health bar was an effective way to measure if a song was too challenging for a player.	Medium - priority exceeded
	Fully completed	Evidence in test 23 and stake holder feedback.	The health bar successful show the user how much health they have left at any time when playing a stage. The bar abstracts the health variable away, showing only a rough visual percentage		

			This it because A player is unlikely to want to know exactly how much health they have left. From stake holder feedback, being able to fail a stage raises the stakes of the game.		
24	Song failed display	After depleting the health bar, the user fails the stage and is given a clear, failed message. All game logics stops and fades away. From here they either get the option to retry or go back to the menu. Text should appear in the centre of the screen for most readability.		The interview and game 1 analysis showed that failing a song was a important feature that helps keep the user focused.	Medium - priority exceeded
	Fully completed	Evidence in test 24 and stake holder feedback.	The stage failed display successfully informs of the user of when they have run out of heath without completely taking them out of the game. This is thanks to the background dim and the stage failed sound effect that makes the transition less abrupt. Stakeholders commented on the simplicity of the UI, which could be improved to be made more interesting. The navigation worked as intended and the stage selection screen can be returned to easily. The added tip in the middle makes the screen more user friendly.		
25	Accuracy	Accuracy text shows up under the score as a constantly updating percentage (out of 100%). 100% accuracy being fully perfect timing and 0% being every note was missed.		games 1 and 2 analyses, along with the interview showed that letting the user know their statistics is important.	Medium - priority exceeded
	Fully completed	Evidence in test 25	The accuracy successfully keeps track of the user's timing accuracy in the stage, is readable and can be used to measure performance.		
26	Rank	Part of the song complete screen – assigns the player a rank based on score after a song is complete. Ranks going from C to B to A to S to S+		games 1 and 2 analyses showed that whilst not necessary, this adds a lot of interest to the game.	low
	Incomplete	n/a	Due to time limitations rank was not developed.		
27	Timing indicators	After each note, text displays depending on how early or late the user's press was. Either a miss, good or perfect is displayed.		games 1 and 2 analyses showed that this was a key feature for the game to feel responsive and for the player to know how well they are playing.	high-priority exceeded
	Fully completed	Evidence in test 27 and stake holder feedback.	Timing indicators successfully let the user know how close their hit input was to the beat in the song. Stake holder feedback shows that the timing indicators are fair and that the judgements of perfect, good and miss, help the player know how well they are doing. A limitation of these timing indicators is that the user doesn't know if they were too late or too early compared to the song beat, however the simplicity of the design also appeals to newer players who can easily understand the timings.		

If there were no limitations on the scope of this project, future versions would attempt to target add server-side saved leader boards and experiment will different procedural note generation, so that users could add their own songs. Limitations for the first idea would be that a permanent solution for saving files would be needed, and since this project was a one off and made for single player use, saving scores in the cloud would not be a reliable long-term solution. Generating notes procedurally based on recognising beats in the music was discussed previously in the project, and if added, would be limited by stages needed to be saved after being generated, since procedural generation gives random results, as well as needing a user interface for players to add their own songs.

Overall, the scope of the project was estimated roughly correctly, as most of the features that I planned to include were included. 4 out of 27 success criteria were not completed, 2 of the 27 priorities were not met, and 10 of the 27 priorities were exceeded. In the end a rhythm game accessible to new and experienced players of varying ages was developed, with repeatable and re-playable stages, and a robust and accurate gameplay loop based on features from other 2 key games, such as “Muse Dash”, “Osu!Taiko” and “Taiko no Tatsujin: Drum ‘n’ Fun!”. Stake holders who enjoyed these games enjoyed the similar features in my project, and the game was understandable to new player aiming to improve their rhythm sense.

Appendix

Code

All code and project files can also be viewed at https://github.com/JachymT/RhythmGame

Conductor.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Melanchall.DryWetMidi.Core;
using Melanchall.DryWetMidi.Interaction;
using System.IO;
using System;
using TMPro;

public class Conductor : MonoBehaviour
{
    public static Conductor Instance;
    public static MidiFile midiFile;

    public Lane[] lanes; // array
    public static float songDelay; // in milliseconds
    public double marginOfError; // in milliseconds
    public float inputDelay; // in milliseconds
    public static string fileName;
    public static float noteTime; // travel time aka aproach rate
    public float noteSpawnX;
```

```
public float noteTapX;
public float noteDespawnX;
public AudioSource audioSource;
public static AudioClip currentSong;
public static float volumeMusic;
public float songBeat;
public static float bpm;
private float secondsPerBeat
{
    get
    {
        return 60f / bpm;
    }
}
public bool beatFrame;
private float audioStartTime;

// Start is called before the first frame update
void Start()
{
    Instance = this;

    if (!ReadFromFile()){
        print("Stage data cannot be loaded: file not found");
    }
}

// File name specified beforehand by the stageManager class
private bool ReadFromFile()
{
    // check if midi file exists in the streaming assets folder
    if (File.Exists(Application.streamingAssetsPath + "/" + fileName)){
        // read from midi file
        midiFile = MidiFile.Read(Application.streamingAssetsPath + "/" + fileName);
        GetDataFromMidi();
        return true;
    }
    // if midi file does not exist
    else{
        print("File not found");
        return false;
    }
}

// Using the Dry Wet Midi library
```

```
public void GetDataFromMidi()
{
    // notes read and copied to array
    var notes = midiFile.GetNotes();
    var array = new Melanchall.DryWetMidi.Interaction.Note[notes.Count];
    notes.CopyTo(array, 0);

    // passes array to lane classes
    foreach (var lane in lanes)
        lane.SetTimeStamps(array);

    // calls start song after waiting for songDelay miliseconds
    StartSong();
}

public void StartSong()
{
    audioStartTime = (float)AudioSettings.dspTime;
    audioSource.volume = volumeMusic;
    audioSource.clip = currentSong;
    audioSource.Play();
}

public void StopSong()
{
    audioSource.Stop();
}


public static double GetAudioSourceTime()
{
    // replacesd time sampling method with dspTime to allow for build in song delay
    return (double)(AudioSettings.dspTime - Instance.audioStartTime - songDelay/1000);
}
```

Lane.cs

```
using Melanchall.DryWetMidi.Interaction;
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Lane : MonoBehaviour
{
```

```
    public Melanchall.DryWetMidi.MusicTheory.NoteName noteRestriction;
    public KeyCode input1;
    public KeyCode input2;
    public KeyCode input3;
    List<Note> notes = new List<Note>();
    public List<double> timeStamps = new List<double>();

    public GameObject notePrefab;

    public SpriteRenderer laneHitArea;

    private float hitAreaAlpha;
    private float grey = 0.5f; // alpha value constant
    private float t; // for interpolation

    int spawnIndex = 0;
    int inputIndex = 0;

    void Start()
    {
        // between the two lanes, sets allNotes to the numebr of total notes
        ScoreManager.Instance.allNotes = ScoreManager.Instance.allNotes + timeStamps.Count;

        // set hit area colour
        laneHitArea.color = new Color(1f,1f,1f,grey);
    }

    public void SetTimeStamps(Melanchall.DryWetMidi.Interaction.Note[] array)
    {
        foreach (var note in array)
        {
            // filters notes
            if (note.NoteName == noteRestriction)
            {
                // converts to metric time
                var metricTimeSpan = TimeConverter.ConvertTo<MetricTimeSpan>(note.Time, Conductor.midiFile.GetTempoMap());

                // convert to seconds
                var secondsTimeSpan = (double)metricTimeSpan.Minutes * 60f + metricTimeSpan.Seconds +
(double)metricTimeSpan.Milliseconds / 1000f;

                // adds to timeStamps list if not duplicate
                if(!(timeStamps.Contains(secondsTimeSpan))){
                    timeStamps.Add(secondsTimeSpan);
                }
```

```
        }
    }
}

// Update is called once per frame
void Update()
{
    // spawn notes if there are notes left to spawn and time is correct
    if (spawnIndex < timeStamps.Count && !ScoreManager.Instance.failedControl)
    {
        if (Conductor.GetAudioSourceTime() >= timeStamps[spawnIndex] - Conductor.noteTime)
        {
            // spawn note
            var note = Instantiate(notePrefab, transform);
            notes.Add(note.GetComponent<Note>());

            // calculate how late note is spawning to adjust stating value of t
            double timeDelay = Conductor.GetAudioSourceTime() - (timeStamps[spawnIndex] - Conductor.noteTime);
            note.GetComponent<Note>().spawnDelay = timeDelay / (Conductor.noteTime * 1.25);

            // assign time
            note.GetComponent<Note>().assignedTime = (float)timeStamps[spawnIndex];
            spawnIndex++;
        }
    }

    // register inputs
    if (inputIndex < timeStamps.Count && !ScoreManager.Instance.failedControl)
    {
        double timeStamp = timeStamps[inputIndex]; // hit time of incoming note
        double marginOfError = (Conductor.Instance.marginOfError) / 1000; // convert marginOfError to seconds
        double audioTime = Conductor.GetAudioSourceTime() - (Conductor.Instance.inputDelay / 1000.0f); // convert input delay to
seconds

        // Input.GetKeyDown(input) || Conductor.Instance.beatFrame == true
        // Input.GetKeyDown(input) || Math.Abs(timeStamp - audioTime) < 0.01
        if (Input.GetKeyDown(input1) || Input.GetKeyDown(input2) || Input.GetKeyDown(input3))
        {
            // hit area effects starts at full alpha brightness
            hitAreaAlpha = 1f;
            t = 1f;

            // conditions to hit a perfect
            if (Math.Abs(audioTime - timeStamp) < marginOfError)
            {
```

```
                Hit(0);
                //print($"Hit note {inputIndex} with PERFECT timing and {audioTime - timeStamp} delay");
                Destroy(notes[inputIndex].gameObject);
                inputIndex++;
            }

            else if (Math.Abs(audioTime - timeStamp) < (marginOfError*2.25)) //muse dash uses *2.6 and a 100ms margin of error,
            {
                Hit(1);
                //print($"Hit note {inputIndex} with GOOD timing and {audioTime - timeStamp} delay");
                Destroy(notes[inputIndex].gameObject);
                inputIndex++;
            }

            //outside of good range (miss range)
            else if (Math.Abs(audioTime - timeStamp) < marginOfError*2.75)
            {
                Miss();
                //print($"Missed note {inputIndex} with {audioTime - timeStamp} delay");
                Destroy(notes[inputIndex].gameObject);
                inputIndex++;
            }
        }

        // outside of miss range and no input key pressed
        if (timeStamp + marginOfError*2.75 <= audioTime)
        {
            Miss();
            //print($"Missed {inputIndex} note: {audioTime - timeStamp} delay");
            inputIndex++;
        }
    }

    // whilst image isnt at the grey constant, decrease alpha
    if (hitAreaAlpha != grey)
    {
        fadeHitArea();
    }
}

public void fadeHitArea()
{
    // using unity's non linnear interpolation
    t -= Time.deltaTime/3;
    hitAreaAlpha = Mathf.SmoothStep(grey, hitAreaAlpha, t);
```

```
        // change alpha after interpolatinng
        laneHitArea.color = new Color(1f,1f,1f,hitAreaAlpha);
    }

    // calls scoreManager
    private void Hit(int timingRating)
    {
        ScoreManager.Hit(timingRating);
    }

    // calls scoreManager
    private void Miss()
    {
        ScoreManager.Miss();
    }
}
```

Note.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Note : MonoBehaviour
{
    private double timeInstantiated;
    public float assignedTime;
    public double spawnDelay;

    void Start()
    {
        // spawn delay moves note left according to how late it was spawned
        timeInstantiated = Conductor.GetAudioSourceTime() - spawnDelay;
    }

    // Update is called once per frame
    void Update()
    {
        double timeSinceInstantiated = Conductor.GetAudioSourceTime() - timeInstantiated;
        float t = (float)(timeSinceInstantiated / (Conductor.noteTime * 1.25));    // noteTime * 1.25 represents total time to travel from start to
end

        if (!ScoreManager.Instance.failedControl)
        {
```

```
            if (t > 1)
            {
                Destroy(gameObject);
            }
            else
            {
                transform.localPosition = Vector3.Lerp(Vector3.right * Conductor.Instance.noteSpawnX, Vector3.right *
Conductor.Instance.noteDespawnX, t);
                GetComponent<SpriteRenderer>().enabled = true;
            }
        }
    }
}
```

ScoreManager.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;
using TMPro;

public class ScoreManager : MonoBehaviour
{
    public static ScoreManager Instance;
    public AudioSource hitSFX;
    public AudioSource missSFX;
    public AudioSource failSFX;

    // attributes from other classes
    public static float volumeSFX;
    public int allNotes;
    public static int highscore;
    public static int currentStage;

    // in game UI
    public TMPro.TextMeshPro comboText;
    public TMPro.TextMeshPro scoreText;
    public TMPro.TextMeshPro accuracyText;

    public SpriteRenderer timingText;
    public Sprite[] indicatorImgs; // perfect, good, miss
    static float transparency;
```

```
public Image healthBar; // set to mask
public Image healthBarParent; // set to parent
public float healthMax;
static float healthCurrent;

// post game UI
public Transform failWindow;
public Transform winWindow;
public bool failedControl;

public TMPro.TextMeshProUGUI comboEndText;
public TMPro.TextMeshProUGUI scoreEndText;
public TMPro.TextMeshProUGUI accuracyEndText;
public TMPro.TextMeshProUGUI perfectEndText;
public TMPro.TextMeshProUGUI goodEndText;
public TMPro.TextMeshProUGUI missEndText;
public TMPro.TextMeshProUGUI highscoreEndText;
public TMPro.TextMeshProUGUI highscoreEndNotif;
public TMPro.TextMeshProUGUI highestComboText;

// Performance stats
static int combo;
static int maxCombo;
static int score;
static double accuracy;
static int notesPassed;
static int notesPerfect;
static int notesGood;
static int notesMissed;

void Start()
{
    Instance = this;
    combo = 0;
    maxCombo = 0;
    score = 0;
    accuracy = 100;
    notesPassed = 0;
    notesPerfect = 0;
    notesGood = 0;
    notesMissed = 0;
    transparency = 0;
    healthCurrent = healthMax;
    failedControl = false;
    allNotes = 0;
```

```
    Instance.hitSFX.volume = volumeSFX;
    Instance.missSFX.volume = volumeSFX;
    Instance.failSFX.volume = volumeSFX;
    failWindow.gameObject.SetActive(failedControl);
    winWindow.gameObject.SetActive(false);
    healthBarParent.gameObject.SetActive(true);

}

public static void Hit(int timingRating)
{
    // update combo and max combo
    combo += 1;
    if (combo > maxCombo)
    {
        maxCombo = combo;
    }

    notesPassed += 1;
    transparency = 1;
    Instance.timingText.color = new Color(1f,244f/255f,44f/255f,transparency);

    if (timingRating == 0){ // for a perfect hit
        notesPerfect += 1;
        score += combo *30; // increase score
    }

    else if (timingRating == 1){ // for a good hit
        notesGood += 1;
        score += combo*20; // increase score
    }

    Instance.CalculateAcc();
    Instance.hitSFX.Play();
    Instance.timingText.sprite = Instance.indicatorImgs[timingRating];

    //checks if stage is complete
    if (notesPassed == Instance.allNotes)
    {
        Instance.StartCoroutine(Instance.StageComplete());
    }
}

public static void Miss()
```

```
{
    combo = 0; // reset combo
    notesPassed += 1;
    notesMissed += 1;

    // take away health, unless less than 0
    healthCurrent -= 30;
    if (healthCurrent < 0)
    {
        healthCurrent = 0;
    }

    transparency = 1;
    Instance.timingText.color = new Color(1f,244f/255f,44f/255f,transparency);

    Instance.CalculateAcc();
    Instance.missSFX.Play();
    Instance.timingText.sprite = Instance.indicatorImgs[2];

    //checks if stage is complete
    if (notesPassed == Instance.allNotes)
    {
        Instance.StartCoroutine(Instance.StageComplete());
    }
}

// method for calculating accuracy proprtianlly to hit areas
private void CalculateAcc()
{
    accuracy = (notesPerfect + notesGood*0.66666666) / notesPassed * 100;
    accuracy = Math.Round(accuracy, 2);
}

private void Update()
{
    // keybinds for return button
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        OnReturn();
    }

    // update HUD
    comboText.text = combo.ToString();
    scoreText.text = score.ToString();
    // add 0s to accuracy text
```

```
        if (accuracy.ToString().Length == 4)
        {
            accuracyText.text = accuracy.ToString() + "0%";
        }
        else
        {
            accuracyText.text = accuracy.ToString() + "%";
        }

        // update healthbar
        healthBar.fillAmount = healthCurrent / healthMax;

        // stage failed
        if (healthCurrent == 0 && !failedControl)
        {
            StageFailed();
        }

        FadeText();
    }

    // when stage is failed, pause all events and show fail window
    public void StageFailed()
    {
        failedControl = true;
        failWindow.gameObject.SetActive(failedControl);
        Instance.failSFX.Play();
        Conductor.Instance.StopSong();
    }

    // use IEnumerator to be able to wait
    IEnumerator StageComplete()
    {
        // wait for 3 seconds
        yield return new WaitForSecondsRealtime(3);

        // camera is rendered first, canvas second (canvas goes ontop)
        healthBarParent.gameObject.SetActive(false);

        // maxScore is the highest possible score the user can achieve
        int maxScore = 0;
        for (int i = 1; i <= Instance.allNotes; i++)
        {
            maxScore += i*30;
        }
```

```
    if (score > maxScore)  // score cannot be greater than maxScore
    {
        score = maxScore;
    }

    // update text before displaying
    comboEndText.text = "Best Combo: " + maxCombo.ToString();
    scoreEndText.text = "Score: " + score.ToString();
    highscoreEndText.text = "High Score: " + highscore.ToString();
    accuracyEndText.text = "Accuracy: " + accuracy.ToString() + "%";
    perfectEndText.text = "Perfects: " + notesPerfect.ToString();
    goodEndText.text = "Goods: " + notesGood.ToString();
    missEndText.text = "Misses: " + notesMissed.ToString();
    highestComboText.text = "Highest Possible Combo: " + allNotes.ToString();
    highscoreEndNotif.text = " "; // empty text

    // set new highscore
    if (score > highscore)
    {
        highscore = score;
        // saves highscore from stageManager class
        StageManager.setHighscore(currentStage, highscore, maxCombo);
        // show message to user
        highscoreEndNotif.text = "New High Score!";
    }

    winWindow.gameObject.SetActive(true);
}

// fades out timing indicator text with exponential interpolation
public void FadeText()
{
    if (transparency > 0)
    {
        // animation takes 1/constant after delta time, in this case 1/2 seconds
        transparency = Mathf.Pow((transparency*transparency*transparency) - Time.deltaTime*2, 1f/3f);
        timingText.color = new Color(1f,244f/255f,44f/255f,transparency); // changes alpha value
    }
    else{
        transparency = 0;
    }
}

public void OnReturn()
{
```

```
        SceneManager.LoadScene("StageSelect");
    }
}
```

Stage.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;
using TMPro;
using UnityEngine.EventSystems;

// inheritance required required for OnSelect() and OnPointerEnter()
public class Stage : MonoBehaviour, ISelectHandler, IPointerEnterHandler
{
    public TMPro.TextMeshProUGUI stageText;
    public Image stageIconSmall;
    public int stageID;
    public string midiFile;
    public AudioClip audioFile;
    public Sprite icon;
    public float bpm;
    public string length;
    public string title;
    public string artist;
    public string difficulty;
    public int highscore;
    public int highcombo;
    public int maxCombo;
    public float audioDelay;

    void Start()
    {
        // button style initialized
        stageText.text = title.ToString() + " - " + artist.ToString();
        stageIconSmall.sprite = icon;

        Load();
    }

    public void Load()
    {
        // gets game data from file
        ScoreData data = SaveSystem.LoadHighScoreData();
```

```
        if (data != null)
        {
            // only loads data matching stage ID
            highscore = data.highScores[stageID];
            highcombo = data.highCombos[stageID];
        }
        else
        {
            Debug.Log("No high score save made");
        }
    }

    // button pressed
    public void OnEnter()
    {
        Conductor.fileName = midiFile;
        Conductor.songDelay = audioDelay;
        Conductor.bpm = bpm;
        Conductor.currentSong = audioFile;
        ScoreManager.highscore = highscore;
        ScoreManager.currentStage = stageID;
        SceneManager.LoadScene("MainStage");
    }

    // button selected
    public void OnSelect(BaseEventData eventData)
    {
        StageManager.Instance.UpdateSidePanel(stageID);
    }

    // button hovered over
    public void OnPointerEnter(PointerEventData eventData)
    {
        StageManager.Instance.UpdateSidePanel(stageID);
    }
}
```

StageManager.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;
using TMPro;
```

```
public class StageManager : MonoBehaviour
{
    public static StageManager Instance;

    public TMPro.TextMeshProUGUI songText;
    public TMPro.TextMeshProUGUI artistText;
    public TMPro.TextMeshProUGUI difficultyText;
    public TMPro.TextMeshProUGUI lengthText;
    public TMPro.TextMeshProUGUI bpmText;
    public TMPro.TextMeshProUGUI highScoreText;
    public TMPro.TextMeshProUGUI highComboText;
    public Image iconImage;

    public ParticleSystem particleSystem2;
    public AudioSource buttonSound;

    public Stage[] stages; // array of game objects
    public List<int> highScores = new List<int>(); // list of highscores
    public List<int> highCombos = new List<int>(); // list of highcombos

    void Start()
    {
        Instance = this;
        UpdateSidePanel(0); // first stage is automatically selected

        // start partice loop partway through
        particleSystem2.Simulate(UnityEngine.Random.Range(0f, 30f));
        particleSystem2.Play();

        buttonSound.volume = ScoreManager.volumeSFX;

        // use co-routine to wait for stages to fully load
        Instance.StartCoroutine(Instance.ReadHighscores());
    }

    void Update()
    {
        // keybinds for return button
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            OnReturnClicked();
        }
    }
```

```
    IEnumerator ReadHighscores()
    {
        // wait for 0.1 seconds
        yield return new WaitForSecondsRealtime(0.1f);

        // stage highscores addded to list
        foreach (var stage in stages)
        {
            highScores.Add(stage.highscore);
            highCombos.Add(stage.highcombo);
        }
    }

    public void OnReturnClicked()
    {
        SceneManager.LoadScene("GameMenu");
    }

    public void UpdateSidePanel(int stageSelected)
    {
        buttonSound.Play(); // play sfx

        songText.text = stages[stageSelected].title;
        artistText.text = stages[stageSelected].artist;
        difficultyText.text = stages[stageSelected].difficulty;
        lengthText.text = "Length: " + stages[stageSelected].length;
        bpmText.text = "BPM: " + stages[stageSelected].bpm.ToString();
        highScoreText.text = "High Score: " + stages[stageSelected].highscore.ToString();
        highComboText.text = "Best Combo: " + stages[stageSelected].highcombo.ToString() + " / " +
stages[stageSelected].maxCombo.ToString();

        iconImage.sprite = stages[stageSelected].icon;
    }

    public static void setHighscore(int stageID, int highscore, int highcombo)
    {
        // set highscore
        StageManager.Instance.highScores[stageID] = highscore;
        StageManager.Instance.highCombos[stageID] = highcombo;

        // save all highscores
        StageManager.Instance.Save();
    }

    public void Save()
```

```
    {
        // passes a referance to the stage manager class
        SaveSystem.SaveHighScoreData(StageManager.Instance);
    }
}
```

Game.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;
using TMPro;

public class Game : MonoBehaviour
{
    public Slider musicSlider; // between 0 and 1
    public Slider sfxSlider; // between 0 and 1
    public Slider speedSlider; // between 0.5 and 2
    public float speed;

    public TMPro.TextMeshProUGUI musicVolText;
    public TMPro.TextMeshProUGUI SFXVolText;
    public TMPro.TextMeshProUGUI NoteSpeedText;

    public ParticleSystem particleSystem1;

    void Start()
    {
        Application.targetFrameRate = 300;

        // start partice loop partway through
        particleSystem1.Simulate(UnityEngine.Random.Range(0f, 30f));
        particleSystem1.Play();

        // loads user settings
        // on slider changed method is automatically called and settings are validated.
        Load();
    }

    // called once slider is moved
    public void OnSliderChanged()
    {
        // validate music volume slider
```

```
    if (musicSlider.value < 0 || musicSlider.value > 1)
    {
        // Invalid music volume, reset to default value
        musicSlider.value = 0.5f;
    }
    //round to nearest whole number, convert to percentage and display
    musicVolText.text = (Math.Round(musicSlider.value* 100, 0)).ToString() + "%";

    // validate sfx volume slider
    if (sfxSlider.value < 0 || sfxSlider.value > 1)
    {
        // Invalid sound effects volume, reset to default value
        sfxSlider.value = 0.5f;
    }
    //round to nearest whole number, convert to percentage and display
    SFXVolText.text = (Math.Round(sfxSlider.value * 100, 0)).ToString() + "%";

    // convert slider value to float between 0.5 and 2
    speed = 2.5f - speedSlider.value;

    // validate new speed
    if (speed < 0.5 || speed > 2)
    {
        // Invalid note speed, speed needs to be reset and not slider.
        speed = 1f;
    }
    // display slider value and not actua value for human readability.
    NoteSpeedText.text = (Math.Round(speedSlider.value, 2)).ToString();
}

public void OnStartClicked()
{
    // save user settings
    Save();

    // pass values to conductor and score manager classes
    Conductor.noteTime = speed;
    Conductor.volumeMusic = musicSlider.value;
    ScoreManager.volumeSFX = sfxSlider.value;

    SceneManager.LoadScene("StageSelect");
}

public void OnExitClicked()
{
```

```
        // save user settings before quiting
        Save();
        Application.Quit();
    }

    public void Save()
    {
        //passes a referance to the game class
        SaveSystem.SaveGameData(this);
    }

    public void Load()
    {
        // gets game data from file
        GameData data = SaveSystem.LoadGameData();

        if (data != null)
        {
            musicSlider.value = data.musicVolume;
            sfxSlider.value = data.sfxVolume;
            speedSlider.value = data.noteSpeed;
        }
        else
        {
            Debug.Log("No save made");
            Save();
        }
    }
}
```

SaveData.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[System.Serializable]
public class GameData
{
    public float musicVolume;
    public float sfxVolume;
    public float noteSpeed;

    // constructor method
    public GameData(Game game)
    {
```

```
        this.musicVolume = game.musicSlider.value;
        this.sfxVolume = game.sfxSlider.value;
        this.noteSpeed = game.speedSlider.value;
    }
}

[System.Serializable]
public class ScoreData
{
    public int[] highScores  = new int[5];
    public int[] highCombos  = new int[5];

    public ScoreData(StageManager stageManager)
    {
        for (int i = 0 ; i < 5; i++)
        {
            this.highScores[i] = stageManager.highScores[i];
            this.highCombos[i] = stageManager.highCombos[i];
        }
    }
}
```

SaveSystem.cs

```
using UnityEngine;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

// static classes cannot be instantiated
public static class SaveSystem
{
    private static string gameDataPath = "RhythmGameData.bin";
    private static string highScoreDataPath = "ScoreData.bin";

    // saves user game data to file
    public static void SaveGameData(Game game)
    {
        string path = Path.Combine(Application.persistentDataPath, gameDataPath);

        // converts data to a serilazable object
        GameData data = new GameData(game);

        // write to new file
        BinaryFormatter formatter = new BinaryFormatter();
        FileStream stream = new FileStream(path, FileMode.Create);
```

```
    formatter.Serialize(stream, data);
    stream.Close();
}

// reads file and returns serializable game data object
public static GameData LoadGameData()
{
    string path = Path.Combine(Application.persistentDataPath, gameDataPath);

    //check for file
    if (File.Exists(path))
    {
        Debug.Log("game data file found");

        // read from file
        BinaryFormatter formatter = new BinaryFormatter();
        FileStream stream = new FileStream(path, FileMode.Open);
        GameData data = formatter.Deserialize(stream) as GameData; // cast type
        stream.Close();
        return data;

    }
    else
    {
        Debug.Log("game data file not found");
        return null;
    }
}

// saves user highscore data to file
public static void SaveHighScoreData(StageManager stageManager)
{
    string path = Path.Combine(Application.persistentDataPath, highScoreDataPath);
    // converts data to a serilazable object
    ScoreData data = new ScoreData(stageManager);

    // write to new file
    BinaryFormatter formatter = new BinaryFormatter();
    FileStream stream = new FileStream(path, FileMode.Create);
    formatter.Serialize(stream, data);
    stream.Close();
}

// reads file and returns serializable highscore data object
public static ScoreData LoadHighScoreData()
```

```
    {
        string path = Path.Combine(Application.persistentDataPath, highScoreDataPath);

        //check for file
        if (File.Exists(path))
        {
            // read from file
            BinaryFormatter formatter = new BinaryFormatter();
            FileStream stream = new FileStream(path, FileMode.Open);
            ScoreData data = formatter.Deserialize(stream) as ScoreData; // cast type
            stream.Close();
            return data;

        }
        else
        {
            Debug.Log("high score file not found");
            return null;
        }
    }
}
```

Bibliography

[1] https://en.wikipedia.org/wiki/Rhythm_game

[2] https://en.wikipedia.org/wiki/Piano_Tiles

[3] https://en.wikipedia.org/wiki/Taiko_no_Tatsujin:_Drum_%27n%27_Fun!

[4] https://www.frontiersin.org/articles/10.3389/fnhum.2017.00273/full

[5] https://en.wikipedia.org/wiki/Muse_Dash

[6] https://en.wikipedia.org/wiki/Osu!

[7] https://docs.unity3d.com/Manual/system-requirements.html#player

[8] https://docs.unity3d.com/Manual/overview-of-dot-net-in-unity.html

[9] https://store.steampowered.com/app/322170/Geometry_Dash/

[10] https://videlais.com/2021/02/25/using-jsonutility-in-unity-to-save-and-load-game-data/

[11] https://en.wikipedia.org/wiki/Hatsune_Miku:_Project_DIVA

[12] https://github.com/melanchall/drywetmidi

[13] https://youtu.be/XOjd_qU2Ido

[14] https://youtu.be/klvndhw5LGY

[15] https://youtu.be/VHFJgQraVUs

[16] https://youtu.be/ev0HsmgLScg

[17] https://youtu.be/YAHFnF2MRsE

[18] https://youtu.be/6G2NbfadwRA

[19] https://youtu.be/YAHFnF2MRsE

[20] https://docs.unity3d.com/ScriptReference/ParticleSystem.html

Additional sources on rhythm game development

https://www.gamedeveloper.com/audio/coding-to-the-beat---under-the-hood-of-a-rhythm-game-in-unity

https://replit.com/talk/learn/Introduction-to-Making-a-Rhythm-Game/21414

https://www.theverge.com/22417808/unbeatable-rhythm-doctor-accessible-games