Contents

Analysis

Introduction

I want to undergo the development of an adaptation that is targeted to Windows but not specifically exclusively to Windows, of Chris Danford's 2001 arcade style rhythm/dance game: "StepMania". StepMania can be referred to as certain sub-genre of rhythm game(s), commonly referred to as Vertical Scrolling Rhythm Game (VSRG). In StepMania's gameplay, the difficulty selection system of maps (maps can be interchangeably referred to as charts or "beatmaps") has no difficulty calculation system and it is up to the map creator to determine and input the map difficulty itself during the map creation (I will discuss the features of map creation in my essential features later). This is problematic as the difficulty is subjective and prone to human error which can lead to incorrect perceptions of a map's difficulty. My adaptation intends to add a difficulty calculation system and to improve the problem of chart difficulty calculation of most modern VSRGs through an algorithm to determine arrow patterns prone to being miscalculated and introduce a reduction/promotion system of the difficulty calculation based on these factors. I will do this whilst maintaining the regular core aspects of gameplay. This will be beneficial as it will add accurate difficulty calculation and allow players to indicate their true ability in their gameplay without their scoreboard performance being inflated or deflated. This will also prevent mis-ranked players from taking advantage of maps with miscalculated difficulties (this is often referred to as "farming").

Stakeholders

As VSRGs have grown in popularity since their birth in 1998, there is a varying age range and demographic of players who play them. For my game there will be two main audience members (stakeholders).

Early VSRGs from the late 1990s and early 2000s such as StepMania and Konami's "Dance Dance Revolution" (DDR) (more examples will be included in research), typically have a much older age range since most players were in their teens, i.e. 13-18 years old, when they first played them during the era they were released and have either continued to play them as they have got older or have returned to playing them when they have matured. This means the age group of i.e., 21-30+ years old, will be included as part of my Stakeholders.

The reason being is that the much older generations return to VSRGs due to feelings of reminiscence and/or the nostalgia of the gameplay. Another factor is that during the 1990s and early 2000s, most VSRGs like Konami's "Beatmania" Series were only playable in arcade dance machines and the computers themselves that were required to run these games were not a common household item due the expensive of owning a computer. Often the older age range continue to play the much older VSRGs as a hobby or a way to pass time and shy away from long hours of play. This is due to older VSRGs not integrating online features and therefore not having competitive scoreboard rankings of the much younger modern VSRGs. This is also since the older age group have less free time due to increased responsibilities such as working for a job and providing for their families (more on this later).

However more modern VSRGs such as osu!mania that were released in 2010 and onwards (osu!mania was released in 2007), have a much younger age range as since most players were born the time they were released and have picked them up as they became teenagers, during the 2020s modern era. This means the teenage age group of i.e., 12-18 years old will also be part of my Stakeholders. From the teenage perspective, there is more free time due to decreased responsibility. This means the younger teenage demographic is more likely to play VSRGs for longer periods. This is evident in the fact that many content creators, a famous and popular example being "jkzu123" on the platform "YouTube" (Channel link may not be feasible, however at the time of writing this, the link was active: https://www.youtube.com/channel/UCeL9uQhZ8WsXFxGvQCDzrYw), have arisen in number and gained a large audience due to consistently having the time to play and upload content on VSRGs. jkzu23 even stated he began playing VSRGs "since I was 12 years old" which further solidifies my belief. Furthermore, due to the teenage population being more socially interactive due to things such as education and increased time, teenagers are more likely to share VSRGs with their peers, thus leading to an increase in the younger audience playing them. All this evidence suggests that there is more of an audience for the younger generation of VSRG players.

Therefore, I intend to adapt my game to be inclusive for both younger and older audiences. My adaptation will try to cover all the age ranges of VSRGs and appeal to both audiences. This includes the much older age range of 21-30+s of older VSRGs from the 1990s and 2000s and the much younger age range of 12-21 from 2010s and onwards. I will do this by including the nostalgic and retrospective arcade aesthetic of the much older VSRGs i.e., Dance Dance Revolution but also adding the modern-day functionality and mechanics that are appealing to the younger audience that play VSRGs i.e., osu!mania. I also intend to keep my focus on the fact that older VSRGs typically have features (as mentioned and will discuss later) that the much larger younger generation consider to be "flaws." This is due

to modern technology improving from the time of the older generation and VSRGs themselves improving also.

To give an insight into the benefit of my solution, I will represent and outline real-life stakeholders from the stated audiences and demographics and explain the certain features that will benefit them and features that will fit their needs as shown in the list below. I will also interview the stakeholders and include the results of their perspective on VSRGs:

- Nathaniel Binuhe is a peer in my computer science class, and he picked up playing the modern VSRG: osu!mania during the pandemic and has continued to play them ever since. He enjoys playing the game and often plays it in his spare time (as mentioned earlier that the younger generation have more spare time). He represents the much younger audience that play VSRGs in their teens. However, he also states that the "grading (difficulty calculation of charts) system" of osu!mania and other VSRGs is "inconsistent" and is not satisfied (as mentioned earlier). Therefore, my solution of an adaptation using an improved algorithm to create a better difficulty calculation system for charts will benefit him and the rest of the younger audience as part of my stakeholders as a consistent and accurate difficulty calculation system will cause less dissatisfaction and improve their experience playing the VSRG.
- Maria Sheehy is representative of the upper bound of the older generation as she recalls playing the original VSRGs such as Dance Dance Revolution from the 1990s. The most notable aspect of it is that she played them when she was 28 years old when they first came out, meaning that from the time they came out, to the time of writing this, she has significantly aged and mostly reminiscent of the game. Maria also stated, "I used to play those games with my daughter" and "my daughter still plays those games with her kids as well." This gives an insight into the point that the upper bound of my game audience will be 30+ years old. Maria also states that she only plays VSRGs like Dance Dance Revolution, "when we go the arcade" and "which is a couple of times a year." This further signifies my earlier claim of the older generation only playing these games to "live in the past" and reminisce and do not play these games often for prolonged periods albeit to the younger generation. This is due to Maria having more responsibilities such as caring for her family (as mentioned earlier). Therefore, it will benefit the older generation of my stakeholders if I retain the aspects of the arcade style graphics to provoke feelings of nostalgia in my audience. However, a major drawback to my adaptation is that it will not be able to support dance pads (more on this lay) which is fundamentally one of the core

reasons of feelings of nostalgia, therefore keeping the style of the arcade style user interfaces.

- Jean Obungu also grew up playing rhythms when she was a teenager and thoroughly enjoyed them. She recalled that you had to “hit the notes to the beat really fast.” She also recalls that she played it with her older brother and said, “I remember having a lot of fun.” Since then, she has now aged and relinquished from playing the game and feels “nostalgic” about them and further iterates “it was fun at that time.” She further represents the older generation of the stakeholders who are reminiscent of the nostalgic 2001 arcade-style graphics from early VSRGs. Therefore, keeping the aspects of the original Stepmania/Dance Dance Revolution game screen and menu interfaces will benefit the audience of the older generation and fit their needs.
- Samuel Smedley is also a fellow peer of mine who I personally played VSRGs with during the pandemic era of 2021-2022. He initially picked up the popular VSRG “Friday Night Funkin’” at its peak of popularity in late 2021 and advanced onto osu!mania as he was further interested in playing VSRGs. Samuel Smedley recalls finding osu!mania “hard but really fun” and said he “likes the cool graphics and skins” (Which I will further explain in detail in the similar systems section of research later). He is representative of the younger teenage generation who picked up VSRGS in their spare time and enjoyed the much more modern implementation of VSRGs. Because of this it will be beneficial for my adaptation to retain the modern useability aspects of rhythm games such as Quaver and osu!mania (which I will point out in the research section).
- Louis White is also representative of the younger generation who has had experience playing arcade style games, as well as experience playing the VSRG: Harmonix’s “Fortnite Festival” and is interested in rhythm games and contemporary games in general. Louis represents the more casual group of the younger audience who play for shorter periods of time. When asked what Louis’s intention of playing games was. Louis responded with “Just to pass time, whenever I have free time.” This further shows my point of the younger generation playing due to increased free time.
- Harrisson Jarvis has had experience in playing rhythm games such as Beat Saber and Harmonix’s “Fortnite Festival” (The same game studio that made the VSRG Guitar Hero). Harrison is representative of the younger generation that finds rhythm games very exciting. Harrison even stated that he found rhythm games like Beat Saber, “very immersive.”
- Hamish Lindsay has also had experience playing VSRGs such as Friday Night Funkin.’ He stated it was “fun and challenging” and had experience playing it when

it first came out. However, he pointed out that there are certain aspects that he did not like about the game and even other aspects of VSRGs. Hamish represents the VSRG players who have had experience with VSRGs and recognized their flaws.

Research

Ever since their origination in 1987, dance/rhythm games, modern examples being Ubisoft's "Just Dance" to Harmonix's Guitar Hero (more on this in my research section) have been prominent in today's modern game industry. A VSRG is a type of rhythm game in which the user must yield input and "hit" icons (commonly in form of arrows) "on time to the beat," on par with a set of stationary arrows (commonly referred to receptors), synchronous to music playing in the background.

In VSRGs, sequences of arrows (interchangeably referred to as notes) can be arranged to form patterns. These sequences of arrows are reflective of the music's tone and beats per minute (BPM). For example, a fast-paced song's "beat drop" will have multiple arrows sequences that must be hit quickly, whilst a slow based song's break will have fewer arrows and are hit slowly. As the arrows scroll, there are normally a stationary set of target arrows. Once the arrows meet the stationary set of arrows, the user must hit the corresponding arrow via their input device. As music progresses, arrow sequences may vary in pattern and build up to form maps of arrows. As a user plays a map, the user is evaluated on their accuracy of how on time they hit the note coordinated with the beat of music. Each time a note is hit, it is given a judgment of accuracy based on how accurately they hit the note, or if they even hit the note at all. The user is then given a performance "grade" based on the average accuracy of how on time they hit the notes. This forms the basis of most VSRGs. Different songs can have different maps. Within these maps the arrow sequences can vary, some being harder to hit than others. A song can have multiple maps with varying arrow sequences. This forms maps with their own corresponding difficulties. As VSRGs have progressed and allowed multiple songs with maps with multiple different difficulties, these difficulties are given a value and ranked quantitively, e.g., Difficulty 1, 5, 11, or qualitatively. easy, medium, hard, expert.

Various research resources of VSRGs, of which two examples being on Dean Herbert's "mania" game mode of his game "osu!" (Referred to as "osu!mania") and on Swan's "Quaver" (I will discuss and analyze modern examples of modern VSRGs in my research in more detail later), have become online and have increasingly large online communities. This has led to VSRGs becoming competitive with scoreboard rankings based on performance and the difficulty of maps achieved. As a result, typically in modern VSRG's such as osu!mania, the value the difficulties calculated is based on the average density of

the sequences arrows in a map. This means the quantity of arrows that are to be hit in a certain amount of time (typically per second).

However, this approach is problematic as it has caused incorrect estimates of the chart difficulty. Incorrect estimations of chart difficulty can make maps seem harder or easier than they are. This has led to inflated or deflated scoreboard ranking; Thus, players being mis-ranked on online competitive scoreboards. Often mis-ranked players take advantage of maps with mis-calculated difficulties by specifically playing those maps only. This further inflates their scoreboard ranking, giving an incorrect measure of their ability. This leads to dissatisfaction and anger in the online communities for VSRGs and even leads to players quitting the VSRGs they play. (Evidence of this will be covered later)

Therefore, this is why it is a sensible approach to develop my adaptation to be better suited for map difficulty calculation by introducing a

VSRGs and rhythm games are currently amassing a large amount of traction and revenue during the modern era of gaming due to their intricate action-based game style. This can be shown according to Wikipedia's analysis (which can be found by in the follow link: https://en.wikipedia.org/wiki/Friday_Night_Funkin%27) of Ninjamuffin99's popular VSRG "Friday Night Funkin'", revealing that the game's Kickstarter funding was able to "reach[ed] its goal of $60,000 within hours." and in the end the "Kickstarter ultimately raised over $2 million". This then led to an article report from IGN (which can also be found in the following link: https://www.ign.com/articles/kickstarter-record-number-games-2021) stating, "that Friday Night Funkin': was one of the most funded Kickstarter projects of 2021". Another example of the popularity and market revenue of VSRGs can be seen in another Wikipedia analysis of Dance Dance Revolution (https://en.wikipedia.org/wiki/Dance_Dance_Revolution) states that "In 2004, Dance Dance Revolution became an official sporting event in Norway" and that through recent years Konami (the brand behind the game) dedicated "their own competitive tournament, [:] the Konami Arcade Championship"" allowing "different regions around the world to sign up and play in specific online events to earn a spot in the grand finals, typically held in Tokyo, Japan." The article also states that this led to countries like "Korea, Taiwan, and other Asian countries" were "allowed to enter," then in "February 11, 2017", competitors from the "United States were eligible" and in 2020, "eligibility for players in Australia and New Zealand" was available. The article then states that competitors have moved on to win the "global tournament." This signifies the international outreach of VSRGs alike and considering that StepMania is based off Dance Dance Revolution itself, a modern-day adaptation will undeniably have the capability of amassing popularity. Furthermore, the article states that "In March 2023, the first ever *upbeat* tournament was held

at Round1 in Denver, Colorado" with a "$10,000 prize pool". This further shows the resolution of popularity and revenue that VSRGs have the potential to create. This further justifies my reason for making an adaptation.

A list of the similar systems that I will research, some already mentioned, include Friday Night Funkin,' osu!mania, Konami's Dance Dance Revolution, Roxor Game's "In The Groove," and Guitar Hero. I have chosen these systems as all these games still retain the core functionality of a VSRG, but each has their own characteristic and customized "spin." Despite this, each has their fundamental flaws within their games. To provide evidence of what features of my adaptation are to be prioritized and what features are to be ignored. To this I gained insight from the interview results of my stakeholders and described their thoughts on the different similar systems.

Before I delve into the research of similar systems, it is fundamental to highlight the core aspects of StepMania and the reasons for a newer adaptation.

StepMania

StepMania consists of basic VSRG gameplay and is tailored to both the arcade and keyboard users. In the gameplay there are customizable settings, navigation systems to play maps and an area to play songs. The game features multiple game modes that are selectable and multiple modes to play with more than one play cooperatively.

To gain insight into the experience of the gameplay, I interviewed Hamish Lindsay. Hamish played a song on StepMania and immediately noticed a difference in gameplay. Hamish stated, "The game(StepMania) felt "unresponsive" and "I feel like there should be more visuals to audio feedback". He then noted that "It is a lot different to the newer games (VSRGs) I played." This is because StepMania's gameplay was developed before the onset of modern VSRGs and as a result, contrasts with the modern features included in newer VSRGs. This shows evidence of the immediate differences between older and modern day VSRGs and how much older VSRGs may be considered to have fundamental flaws to their gameplay.

Of these fundamental flaws is the map editor system. As mentioned earlier, although there the system allows for users to create customized maps, the difficulty calculation of the map is entirely up to the map editor/creator to input in themselves which leads to problems mentioned earlier of incorrect estimations of the difficulty. Figure 1 shows a screenshot of what entering the difficulty of a map scenario would look like.

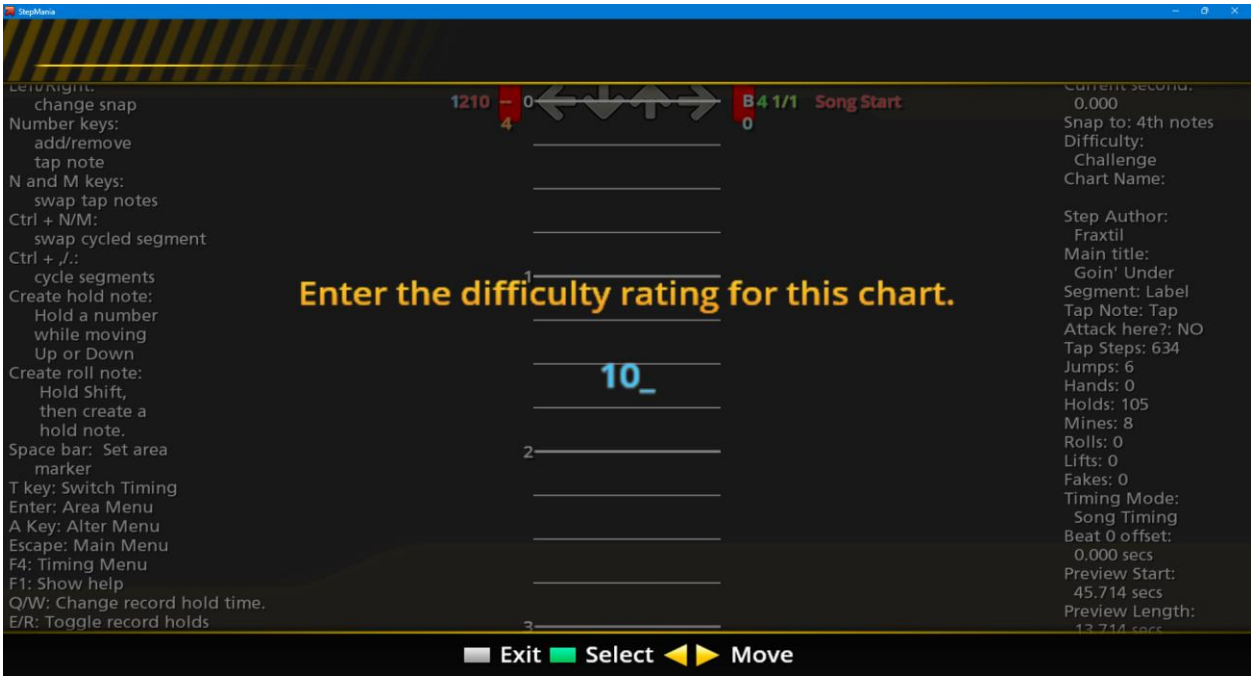


Figure 1 – StepMania's difficulty calculation system requiring the user to input the map difficulty.

Furthermore, I showed Hamish the system of map editing/map creation and asked him the following question:

- What do you think about the map editing system?

Hamish responded by saying "It can lead to human error" and "alongside human error, there is also bias" "A lot of experienced players are more likely to play a higher rated map, for example rated 10 rather than 1, which leads map creators to inflate the difficulty to play them." This shows the fundamental flaws of being able to input the difficulty and clearly identifies the issues of bias and misjudgment. The is due to every map creator has had different learning curves whilst playing the game, therefore perception of difficulty is relative; Somone else's perception of being difficult may be different to another. This suggests that in my newer adaptation of StepMania there must be constant, quantitative measures to determine difficulty to avoid misjudgment and bias.

Osu!mania

As mentioned earlier, Dean Herbert's osu!mania is a popular online VSRG with a competitive scoreboard ranking. This is evident in Osu!'s official country rankings website (https://osu.ppy.sh/rankings/osu/country) that show, at the time of writing this, that there are currently 981,025 active users in the United States alone and a total of 24,202,202 total registered users, as seen on their home page (https://osu.ppy.sh/). On top of this there is a specific scoreboard ranking for osu!mania as seen in the osu!mania rankings page (https://osu.ppy.sh/rankings/mania/performance), showing that the higher your

performance score is, the higher your scoreboard ranking. Thus, giving osu!mania the most competitive nature of modern VSRGs. This is a crucial piece of information as the score of your performance, or as it is termed in Osu! as “performance points” (commonly abbreviated as “pp”) you gain is proportional to the difficulty of the charts you play. This is evidently shown through Osu!’s official FAQ (https://osu.ppy.sh/wiki/en/FAQ#scoring) on performance points, which states “The easiest way to improve it is to score high on difficult songs and playing more songs.” Often higher-ranking players must continually play difficult maps to maintain their status due to this very reason. Furthermore, in Osu!’s Wiki[1], it describes a “weightage system” to “prevent the rapid and repeated gaining of lower pp scores” on “easy beatmaps” by “reducing the amount of pp that is gained.” This means that a percentage of pp is lost the lower the rank the map is, meaning a pp score that ranks second place in best plays will be lose Further suggesting that the only way to increase your “pp” is to play maps continually and progressively with increasing difficulty.

1 https://osu.ppy.sh/wiki/en/Performance_points/Weighting_system

All this evidence suggests that the difficulty of the maps a user plays is a key factor. Therefore, the calculation system to determine the difficulty is substantial.

Before the map difficulty must be calculated, the map must first be made in the map editor system. Osu!mania contains a section of the game in which audio files can be dragged into the game. Upon doing this, a screen allowing the user to enter the song details and customize the metadata of the song appears. Afterwards the user is taken to an area where they can place notes and navigate through the different sections of the song in a customizable manner. Afterwards, once the user is satisfied with the way they have arranged the notes for song, they can exit the map creation system.

This means that my adaptation must contain a system to create and adapt maps in real-time. My adaptation should also include a map selection system to select maps to play and an audio file management system. Much like osu!mania’s system, my adaptation must also contain the ability to edit maps metadata and background images. Furthermore, Osu!mania’s map creation system does not contain an autosave system meaning you must save maps manually. Therefore, my adaptation may add a system to automatically save maps whilst editing them. In osu!mania, once the map has been saved (often manually), The difficulty of the sections of the map is calculated.

As mentioned earlier, osu!mania uses density system to do this. The vertical column with pink and white rectangles indicates the sequences of arrows that are within the song as it scrolls. The vertical bar with yellow bars indicates the density of the patterns (notes per second). The taller the bars the denser the pattern. If the bar turns pink this signifies an extremely dense

sequence of arrows.

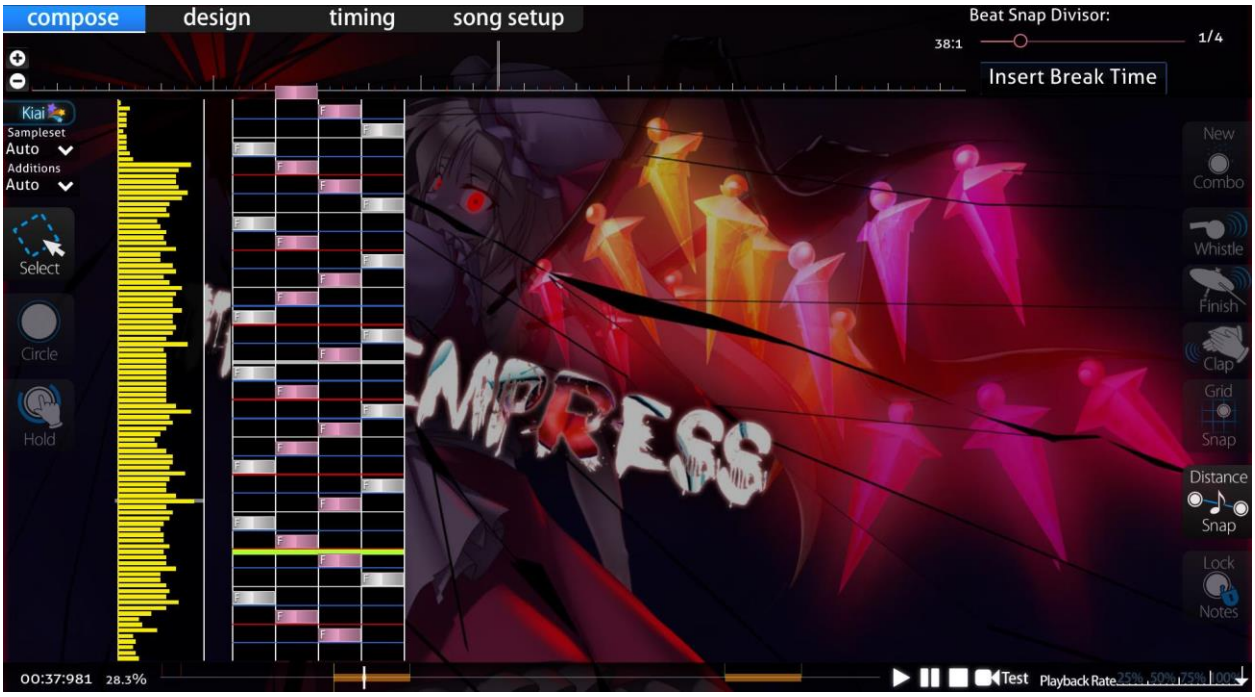


Figure 1 - An osu!mania beatmap in map editing. This reveals the inner workings of the map and the density of the notes. Yellow bars show density.

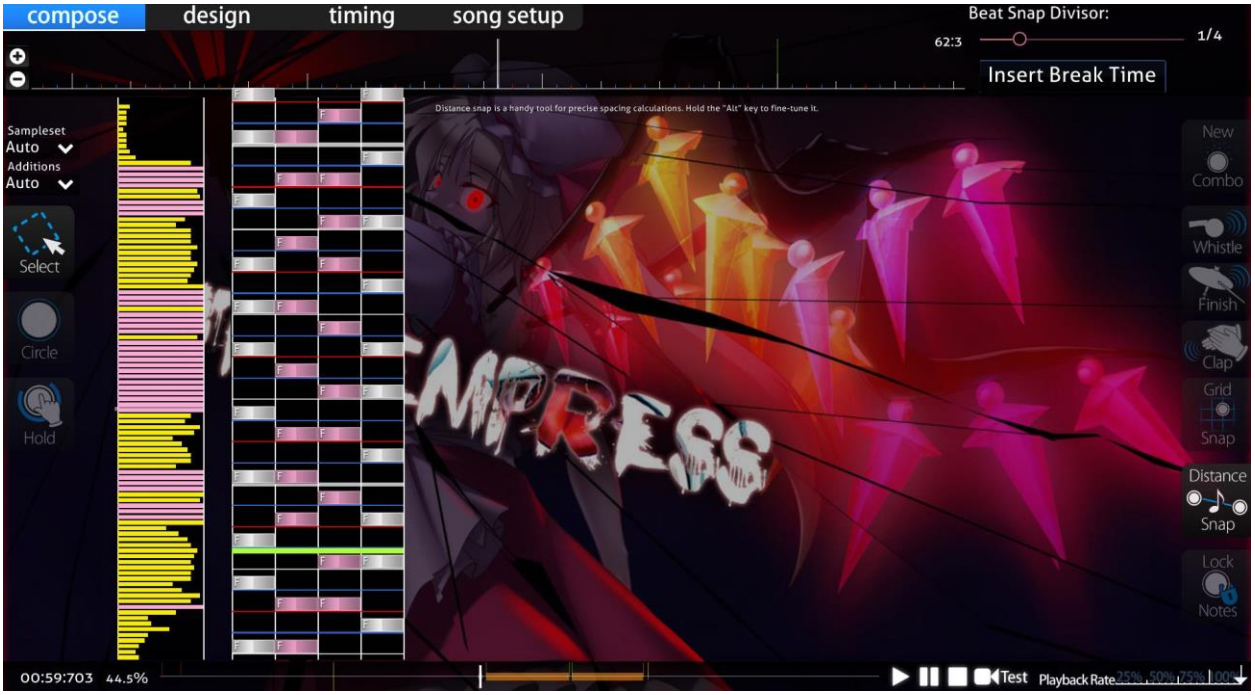


Figure 2 - The same beatmap but a higher difficulty. The pink bars show the extemely dense nature of patterns

Figure 3 shows the nature of maps with high difficulty requiring more dense patterns for longer periods of time. This is shown as most of the map is “pink” with extremely dense patterns.

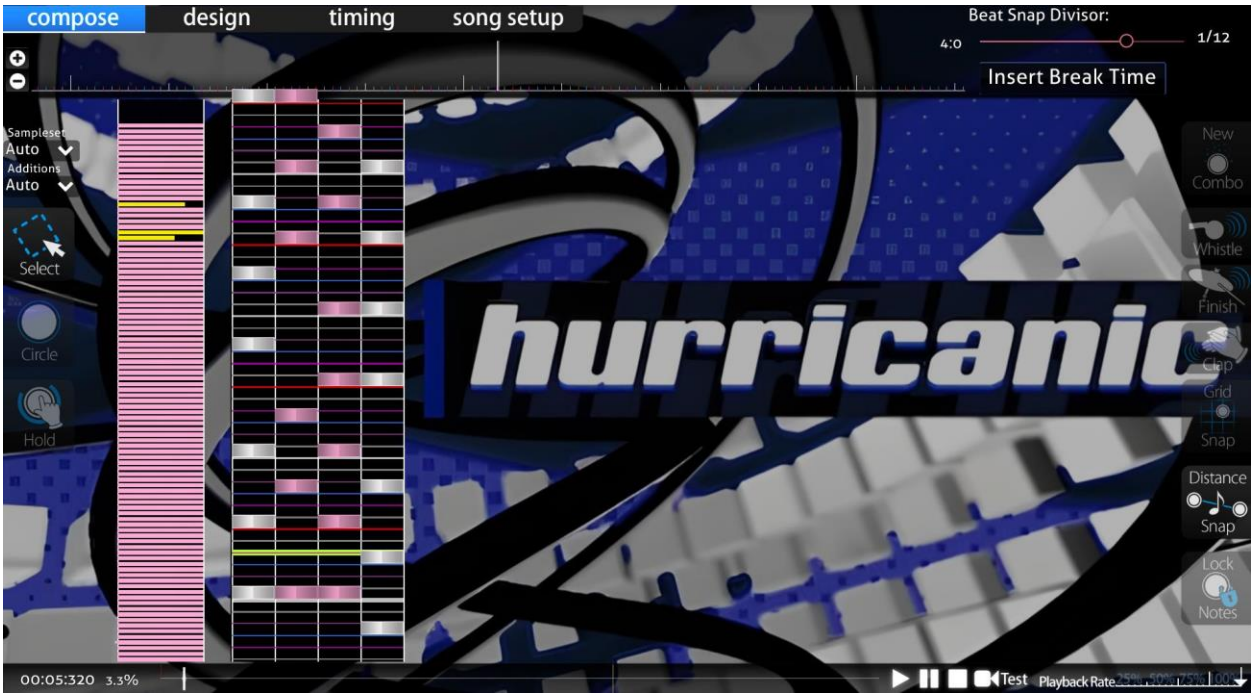


Figure 3 - The pink density graphs of beatmap with 5-star difficulty rating in osu!mania.

There are flaws to this system. This is evident in the fact that Osu! have tried to improve their difficulty calculation system by releasing an update to the difficulty calculation as seen in this official article ：https://osu.ppy.sh/home/news/2022-10-09-changes-to-osu-mania-sr-and-pp. This already gives an insight into the issues that are faced with difficulty calculating. Despite these changes, the difficulty calculation flaws have remained and are still prominent to this day.

To describe the problem with this system and show evidence of the issue of difficulty calculation, I interviewed Nathaniel Binuhe (as mentioned earlier). Nathaniel agreed to play a couple different beatmaps on osu!mania to discuss the problem of incorrect difficulty calculations. To start with, Nathaniel played a song that rated 3.88 stars but with a high BPM of 270 and with technical patterns (see earlier sections). The star rating, BPM, artist name, source (record label), beatmap creator, overall difficulty, long note count (will discuss this later), key count, etc. Can all be seen in the top left corner of the map

selection as shown in Figure 4 and 5.

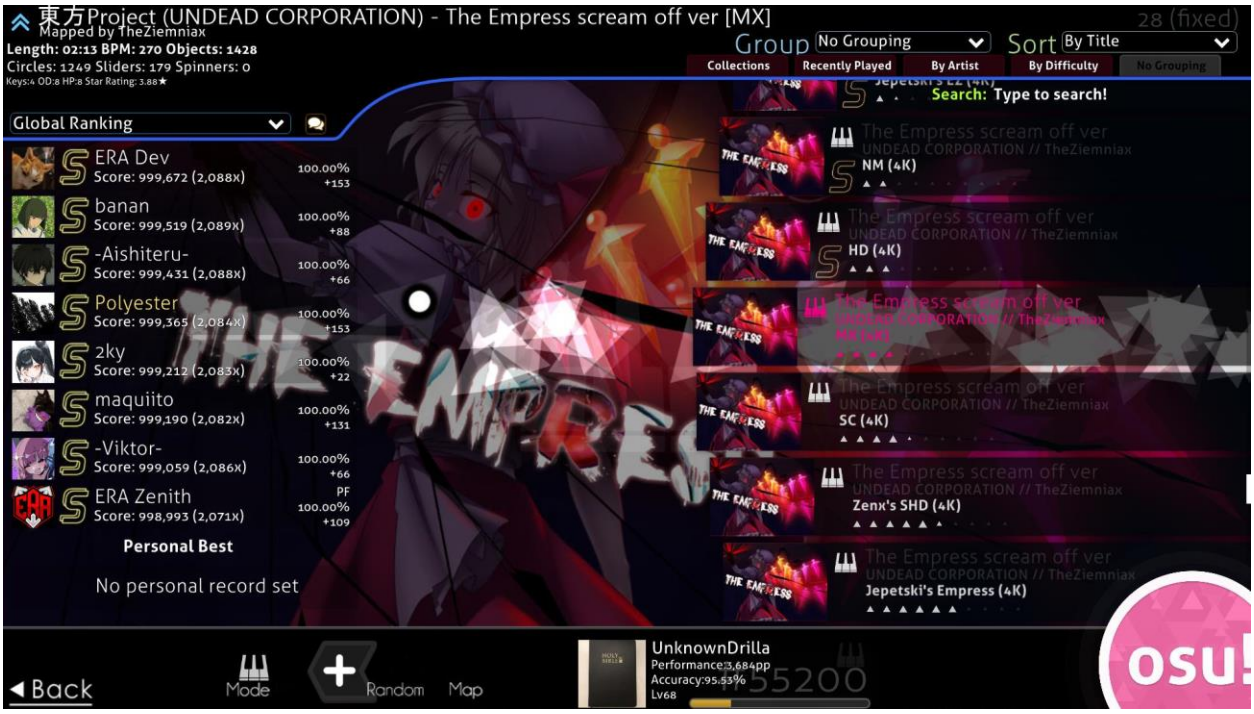


Figure 4 - Map selection screen of osu!mania. Top right shows details and difficulty calculation.

Figure 5 - A zoomed in photo of the beatmap details located in the top right corner of the map selection.

Nathaniel then played a song rated 4.27 stars with a lower BPM of 140 and full of patterns that are prone to being miscalculated. He played the first map and could not complete it. He stated it was “much harder” because of the “higher bpm” and the “insanely fast and hard to read technical patterns.” He also stated that “the first map should be harder (have higher difficulty calculation) than the second map,” even stating that the first map “should not be 3.88 stars.” This signifies the misleading nature of osu!mania’s difficulty calculation and the need for improvement on difficulty calculation. On the other hand, the second map, although having a burst of technical patterns that are prone to being miscalculated, Nathaniel stated, “it is much easier” and “slower due to the lower BPM.” He was surprised stating “the second map is harder is than first map,” further stating that “It should not be harder than the first map.” Overall, Nathaniel stated that the main issue with the game is the “misleading ratings.” Despite the fundamental flaw of the difficulty calculation,

Nathaniel still stated that the game is “still fun to play” and commends the “nice modern graphics” and the ability to “add skins.”

A discussion on Osu!’s official GitHub (https://github.com/ppy/osu/discussions/12980) can be found to show a user named “BrokenGale’s” disclosure of the need for “osu!mania star difficulty improvements”, proposing a new and “augmented star rating for mania”.

Thus, it is evident to see the potential issues of mis-judged rankings that can arise, especially if there is a malformed difficulty calculation system to determine the difficulty of maps. Therefore, It is justifiable to identify my adaptation’s approach of a new difficulty calculation system.

Friday Night Funkin’

Ninjamuffin99’s Friday Night Funkin’ is one the most popular and prevalent VSRGs of the modern-day era. It is widely renowned for its unique cartoon style graphics and its upbeat and catchy video game style music. This can be shown in two reviews on Metacritic (https://www.metacritic.com/game/friday-night-funkin/) by a user under the alias of “izack” states that the game has “banger songs” and “the sprites are fire”. EndlessFlame928 also states that “The mods are keeping the game alive” and “the songs are a banger”. This shows that the more modern and younger audience prefers a more modern, retrospective style of interface. This contrasts with older VSRGs such as Dance Dance Revolution and Stepmania which include a more perspective 3D style of interface. To get an insight into the modern graphics that are common in VSRGs I interviewed Samuel Smedley by asking him a series of questions:

- Do you like cartoon art and graphics of the game?
- Do you like music and character sprites?
- Did you think that the difficulty of maps was harder/easier than they should be?
- Did you feel like it lacked better difficulty calculation/assumption of charts?

Samuel Smedley then responded, “I like [the] cartoon art because it has a unique style that’s extremely vibrant and exciting.” This shows the evidence of the younger generation having more of an appeal for more modern and smooth graphics. Samuel also disclosed his taste for the music of the game by stating “The music is very high energy which fits the setting of each “map”.”

Therefore, it is evident that an approach of keeping the more modern useability and functionality of the user interface but at the same time maintaining the older arcade style 3D renditions of older VSRGs is an identified and justifiable approach to my adaptation. 3D re

Samuel also responded with criticism of “The character sprites I think lacked variety in how it was the same for every map and situation which left more to be desired.” Samuel’s criticism justifies the need for an ability to integrate customized design in the creation of maps. This will allow variation and not leave users with

Alongside the graphics, Samuel responded by stating “the very high energy… doesn’t always translate to the charts as the notes were not always synchronized.” This adds evidence to my previous point of VSRGs sharing a widespread problem of difficulty judgement and as Nathaniel mentioned earlier, “inconsistent grading.” Samuel also expressed “For a beginner player, the difficulty is a good standard but quite quickly after a few hours the hardest difficulty is too easy.” This led Samuel into suggesting “There should be a setting which progressively makes the encounters more difficult” . This suggestion shows the effect of underestimation of difficulty calculation and further signifies why a quantitative measure of difficulty calculation system must be approached. This is evident in the difficulty selection screen for songs as shown in figure 6. The difficulty system has no calculation and is based upon the game developers and mod developers to determine the difficulty. Another factor is that there is no quantitative measure. As mentioned earlier. the difficulty system of Friday Night Funkin’ is based on qualitative measurements e.g. “Hard”, “Medium” and “Easy”, that are relative to perspective and somewhat abstract. These measurements are prone to being misjudged generally and sometimes may give an incorrect indication of the map. Thus, it leads to inconsistencies. This means that my adaptation must implement a quantitative representation of the difficulty.

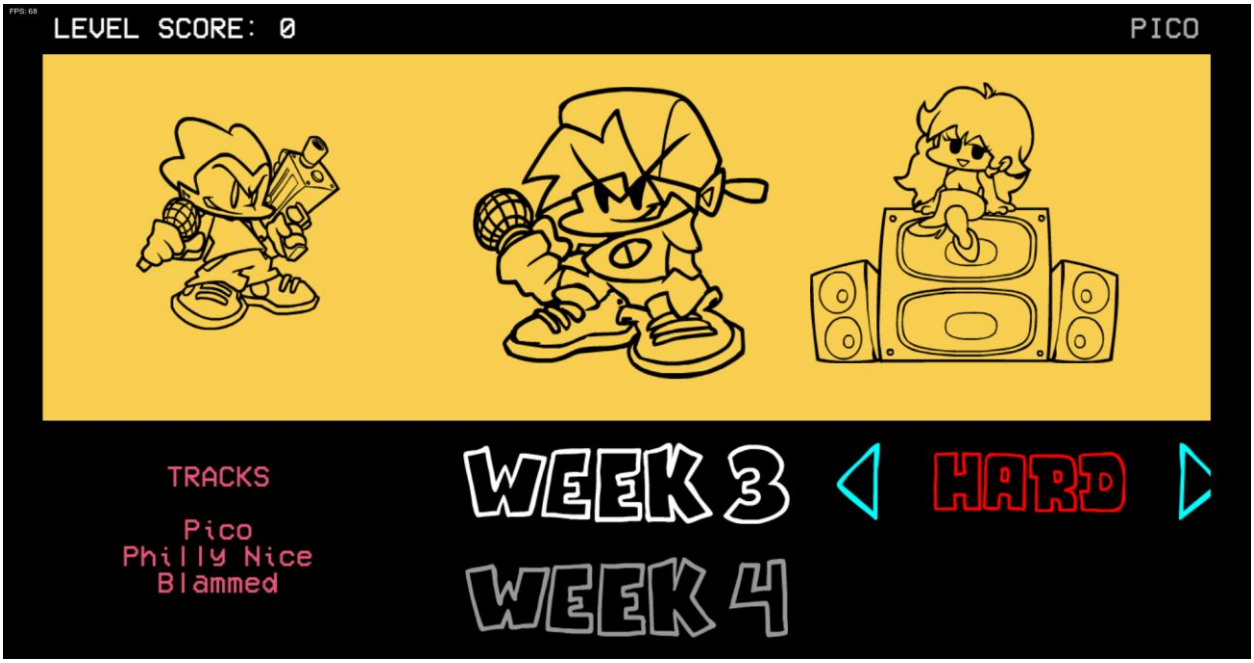


Figure 6 - Abstract difficulty calculation system of Friday Night Funkin'

Dance Dance Revolution

The significance of the success of DDR can be seen in Wikipedia's "Similar Games" section in their article about DDR (https://en.wikipedia.org/wiki/Dance_Dance_Revolution), stating "Due to the success of the Dance Dance Revolution franchise, many other games with similar or identical gameplay have been created." This evidence shows that the onset of multiple VSRGs is a result of inspiration from DDR and therefore the basis of VSRGs after DDR took inspiration from it. This is also evident as the article mentions "Fan-made versions of DDR have also been created... The most popular of these is StepMania." This indicates that core aspects of early VSRGs such as StepMania originated from Dance Dance Revolution as they were either fangames or spin-offs of the game.

As a result of my interview with Maria Sheehy, stated that she is "reminiscent" of the game and particularly the art style of the older VSRGs. Figure 7 shows an insight into the style of the graphics on Dance Dance Revolution. It consists of less cartoon stylized graphics as seen in Friday Night Funkin' and more solid 3D text and a realistic perspective. Therefore, for my adaptation to maintain the older arcade style of older VSRGs, 3D aspects of text and design shall be an implementation of my adaptation's user interface. To implement these 3D aspects, I must program and utilize a 3D rendering system into my game engine that will allow rendering of 3D and allow transformations in the z-plane.

As well as the user interface, DDR is notoriously known for its dance machines that are required to play the game. However, I do not intend to add this functionality as the younger audience of my stakeholders do not wish to have this functionality and prefer simpler forms of playing VSRGs. This is evident in my interview with Louis White. Although Louis stated, “I enjoy going to the arcade,” he feels “not everyone has access to an arcade in their local area” and that “the cost of going to an arcade may not be worth it.” He then stated, “playing at home with a computer is much easier and saves time.” To further elaborate on Louis’ point, an article from Kineticist[1] states that the average cost of a DDR machine is “$3,000-$20,000+” and a new machine is upwards of “$9,000-$20,000+”. This shows if the younger generation would want to play for longer durations personally, they would have to deal with the grievous expenses of buying a dance machine. This is evidence to suggest that integrating features of dance machine hardware into my game would not benefit my stakeholders’ needs.

1 https://www.kineticist.com/post/buy-a-ddr-arcade-machine#:~:text=TL%3BDR%20on%20how%20much%20a%20DDR%20machine%20will,a%20game%20room%20retailer%20like%20Game%20Room%20Guys.

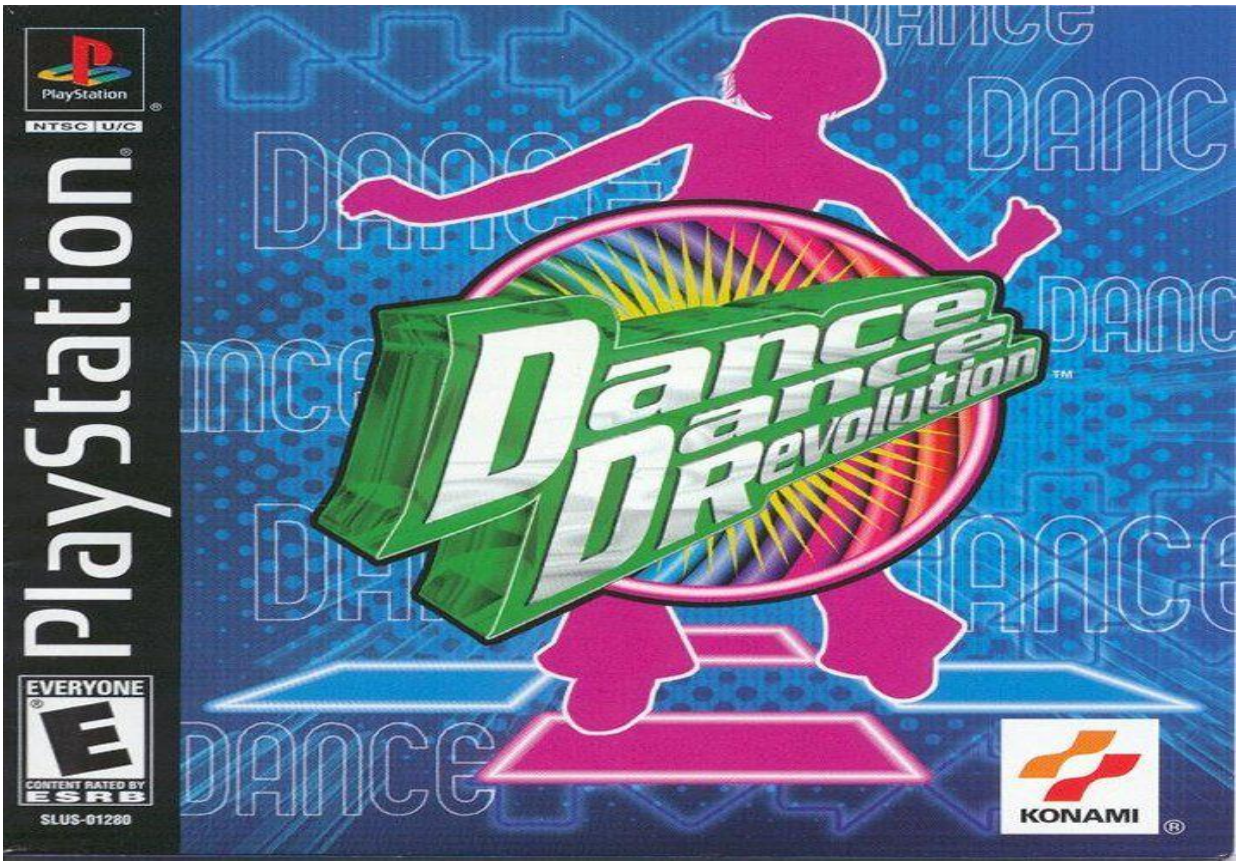


Figure 7 - PlayStation Release of Dance Dance Revolution's game cover.

In The Groove

To delve deeper into the graphical design of older VSRGs, I researched Roxor Game's In The Groove, commonly known as ITG on online forums. ITG is another PlayStation 2 clone of Dance Dance Revolution and was released shortly after. ITG's use of 3D aspects in their logo design, gameplay and menu give the game its nostalgic acquisition of the general style of arcade games of the early 2000s. I believe this is thoroughly shown thrown ITG's PlayStation 2 game cover, which involves a 3D representation of the "arrow" notes (commonly found in the main gameplay of Dance Dance Revolution and other VSRGs) at a different perspective angle and with a 3D depth to its design. This really amplifies the early 2000s feel of the game and gives it a nice touch that is distinguishable from other modern VSRG's such as Friday Night Funkin'. Therefore, in my adaptation's system, including forms of perspective shifts, as seen in the game cover of ITG and Dance Dance Revolution alike, will add to the general feel of mid 2000s VSRGs. These perspectives shifts may be included in micro interactions and parts of user interface that require animation and movement.

Harrison Jarvis has experience with the immersive 3D aspects of rhythm games such as Beat Saber. When asked about his take on the 3D immersion and perspective of ITG. Harrison stated that "It is not too crazy" and "I like the simple design of the logo and game as it makes the game style and gameplay more fluid." He stated, "I like the retro style of ITG and "It shows you what it is from the cover… you can tell it's a dance game." He concludes with "The "big arrow" reminds of the classic dance games of the arcades." Harrison's perspective further suggests that the simple retrospective 3D designs that use clear features relating to VSRGs i.e. arrows, people dancing, musical terms, etc. are what make arcade VSRGs so iconic and appealing. This is especially true for the older audiences as such distinct features are what causes feelings of reminiscence.

Figure 8 - PlayStation 2 Release of In the Groove's game cover.

Another aspect of ITG's style is its main gameplay's arrangement of arrows using different depths in the interface as well as effects of outlining, shadows and in game lighting, that

further enhances the retrospective feel of an early 2000s VSRG.

Figure 9 - In The Groove's main gameplay.

To achieve the desired 3D aspects and designs of older VSRGs, my adaptation must contain a 3D environment to render and perform the different perspectives shifts Therefore, I must use different computational 3D rendering techniques such as model and view matrices and matrix transformations with the use of perspective projection. This should be done programmatically, and the 3D aspects shall come from coding the transformations instead of it entirely being based on design.

Computational Methods

Thinking abstractly

Areas of my adaptation will need abstraction to make the solution to complex problems clearer and easier to solve. Using abstraction will help in focusing on what a part of my code does and its function (what it will output), rather than how it works. This is through hiding the actual implementation of functions after writing them and then reusing them within larger sub-routines.

These abstract functions will then help in piecing together solutions without needing to care so much about how each individual function works. This will benefit my process of development as solutions and methods shall be compounded and based only on the relevant information required for the specific problem. This means that during solving

problems, I will only require knowledge of what data needs to be input and what function is needed to process the data that gives an expected outcome.

This will benefit during debugging problems as I will have only the data, I need to solve the problem and do not have to dissect other irrelevant information. It will also greatly speed up the process of writing code itself and allows a focus on conceptualization of solutions to problems and other computational methods.

Abstraction will also benefit in areas such as making program code more readable and maintainable through the reuse of abstracted sub-routines. It will also benefit the users themselves as they will not need to understand the complicated processors of my adaptation's inner workings. Examples of how abstraction is used in my program include :

1) The use of external libraries, such as glm, to perform complex mathematical operations in accordance with development. I will need to use mathematical constructs such as:
    a) matrix transformations
    b) matrix multiplications
    c) vector addition
    d) vector subtraction
    e) matrix addition
    f) matrix subtraction
    g) multiplying matrices by a scaler
    h) multiplying matrices by a vector
    i) use of trigonometric functions
    j) compute the magnitude of a vector

All functions provided by the library will hide the unnecessary details of how to compute these mathematical concepts and purely focus on performing their intended tasks. Another factor is that library functions have been optimized and thoroughly tested by their developers. This will especially benefit debugging as it will indicate that the error is mostly likely to do with my own source code and how I used the function and not the function itself. Furthermore, the library functions are less likely to have large computational complexity. For example, in a standard matrix multiplication algorithm, to multiply two $n \times n$ matrices, it will require n^3 new multiplications of scalars and $n^3 - n^2$ new additions to compute its product. This means a standard matrix multiplication algorithm has an average time asymptotic (time) complexity of $O(n^3)$. However better matrix multiplication algorithms that are likely to be included in the library functions, have a lower asymptotic complexity of $O(n^{2.3751552})$, thus saving time and being less intensive on the CPU/GPU.

2) Navigation of the user interface - the user does not need to know the details of how the menu navigation system works, just the ability to know what inputs are required to move from one section of the game to another. I will require functions that integrate user interface controls that abstract the process of drawing/rendering triangles through the graphics pipeline and loading of textures/images. These include:
    a) Functions/subroutines that abstract the updating of uniforms variables within shaders and GLSL files to be part of the 3D rendering process.
    b) Have function/subroutines embedded within rendering/drawing GUI classes to abstract the view/projection matrix and remove the need to attribute each matrix transformation when updating source code.
    c) Have reusable sub-routines to simplify and abstract the process of drawing GUI elements. For example, the entire process of rendering a triangle, texturing it, converting it from local space coordinates to view space coordinates and then implementing accessibility to the texturized quad made of triangles to form a "GUI element", can all be simplified into one draw function e.g. `drawGUI()`
3) The use of external window rendering libraries such as SDL2 will abstract the process of rendering a window on the screen. These libraries will simplify window creation and hide all the details that happen in the process of forming a window. Instead, I will only focus on a singular window creation function and the aspects of my window. SDL2 will also provide an OpenGL context for me to render in. Rendering in the OpenGL context is the focus of my adaptation as it is where every element of the user interface shall be rendered. Window libraries, such as SDL2, will abstract features such as
    a) Providing an OpenGL context – The library does not require me to implement me a 3D rendering context/environment myself
4) The use of an OpenGL toolkit/API such as glad that will abstract the process of implementing the specification functions/subroutine calls to the drivers that the graphics card supports. OpenGL is only a standard/specification and there are many different versions of OpenGL drivers, therefore the location functions/subroutines to use OpenGL is not known at compile-time. This means it is up to the user to retrieve the location of the specification functions/subroutines (typically in pointers) and store them for later use. This process is cumbersome as you may need to retrieve the location for every function that has not been declared. However, these toolkits/APIs have abstracted the need to do this. Examples of the abstractions due to toolkits/APIs:
    a) Not having to focus on the entire process of retrieving the graphics specification function/subroutine location within the drivers and instead focus on the actual use of the subroutine. For example, the subroutine: `glGenBuffers()`, would not have to be manually retrieved and instead I can focus on the use of it. This eliminates the unnecessary process of having to retrieve it beforehand.

b) The use of the toolkit/API means that the subroutine calls are updated and thoroughly tested. This provides access to the most up-to-date function/subroutine calls within the drivers and allows me to not worry about the function itself and instead focus my adaptation's code.

Thinking Ahead

Before designing my solution, I must think about the sub-routines required to form a working GUI navigation system as the majority of my adaptation's functionality and accessibility shall be through a GUI. Furthermore, I must think about being able to integrate the 3D aspects within a 2D GUI, to provide a multi-scene perspective. For this, I must ensure that my navigation system is fully responsive to mouse clicks on GUI elements and button presses such as WASD and the Left/Right/Up/Down arrows keys. I must also be able to integrate both functions at the same time to allow customization of keyboard input which is essential for allowing the modern functionality and usability of the adaptation.

I must think about the different reusable classes needed in forming a GUI system that is maintainable, reusable and quick to form new sub classes such as different menu screens and game states like the editing/creation maps section. Features of the GUI class/system include:

1. Universal functions that are inherited in all derived classes are required to render the GUIs onto the screen. These functions will be reusable all throughout the GUI system and will simplify the process of making an interactable GUI. These functions include:
    a. A draw function to draw the GUI element on screen
        i. Within this function it will have:
            1. Position of GUI
                1) Converting local space coordinates to screen space (NDC)
            2. Rendition of triangles and quads for the element
            3. Width and height of GUI element
            4. Percentage of fill of the texture
            5. Texture filtering type (more on this later)
        ii. Within this function a function to load images for the texture
    b. A rotation function that integrates matrix transformations to be able to rotate the GUI without rewriting code
    c. A scale function that integrates the matrix transformations to scale GUIs accordingly

    d. A function to adjust the color, hue and alpha values of the textures and GUI elements without re-texturization.
    e. A function to draw untextured polygons and fill them with color. This will benefit in making some UI elements for menus
2. An animation derived sub-class that packages and abstracts the transformations into more feasible general subroutine. These include:
    a. Flip the GUI element vertically and horizontally and on its axis
    b. Animate these transformations as transitions by implementing them as animation functions
    c. A movement/transform animation that integrates a matrix transformation with iteration to give a "moving" animation on the GUI element. This will be useful while implementing gameplay.
3. Accessibility within the GUI system to ensure the usability of the GUI elements once it's been created. This includes:
    a. Checking if mouse corresponds to the range of the coordinates of the GUIs "size"
        i. Converting from local space coordinates to view space coordinates
            1. Vector and matrix multiplication
        ii. Collision detection algorithm

I must also think about the audio aspects of my game. To do this I must reuse a core audio class. This will also involve outputting audio through the user's sound system accurately and synchronously to the different aspects of gameplay. Features of audio and timing can include:

1) Starting and stopping the map's audio whilst playing a map on time to the input
2) Adding a preview of the map's soundtrack during map selection#
3) Ability to upload audio files for map creation
4) Ability to load different audio files for maps during the map selection in real time
5) Ability to preview sections of audio files for a short period of time during map selection.

Alongside audio, I must pair this with input timing at the same time. Audio and timing go hand in hand and are crucial aspects to the main gameplay. I must be able to utilize an input timing class that is able to correctly time the users input and give a valid response to their timing in real-time without any inconsistencies. This is so that users can receive accurate judgment based on their timing when hitting notes in gameplay. For the timing I must:

1) Form a system that will not have inconsistent input registration and be responsive to the different timings of input. This will be evident during the main VSRG gameplay.
    a. The system must be able to handle multiple keyboard input simultaneously and expect accurate output.
    b. Must be able to calculate the timing of the input to an accurate degree and compare the timings of the data within a map. This will involve real-time processing and will be used during the gameplay when analyzing hit time results. This will be crucial in determining hit accuracy (as discussed earlier).
    c. The system must be able to give the timing to a suitable degree of accuracy and have a universal accuracy gauge to determine the timing.
    d. An average timing system to determine the g

As well as classes and subroutines, I must think about the different data structures that I will use to implement the different aspects of my game. The data structures must allow me to access data efficiently and store large amounts of data without affecting performance and main memory. Some of these data structures include:

1. Arrays – Needed to store vertex buffer data and objects to the data of the coordinates and vertices of the triangles that will represent interfaces and icons
2. Vectors/Lists/Dynamic arrays – During the process of storing the map files and comparing them to the input timing
3. Hash Tables – For providing fast access to maps whilst searching for them in

Thinking Procedurally

To form a stepwise approach to my problem, I must break my problem into more manageable sub-programs. Each sub-program is broken down into their own retrospective elements to the solution of the problems. Examples of the sub-programs include:

1) Enhanced difficulty calculation system to determine note patterns prone to being miscalculated
    a. Data processing algorithms and pattern recognition algorithms to determine certain patterns within map files.
        i. Multiples that contribute to difficulty calculation algorithms such as strain, readability, playability and speed, not just density alone.
            1. Process and read map data
                a. Have a suitable data format for processing e.g. text files.

i. Organize that data to be processed in a fashion such as:
    1. Note timings
    2. Column placement
    3. Beat snap

b. Have predetermined data to signify certain patterns e.g. four continuous notes in the same column are considered as a “Jackhammer” pattern (This is the terminology used to describe repetitive notes in one column in VSRGs) and therefore is a strenuous pattern and should be a factor to increase the difficulty of the map.
    i. Count notes in adjacent and previous columns
    ii. Count notes in adjacent and previous rows
    iii. Measure notes timings
    iv. Count quantity of notes in each time frame
        1. Determine average quantity of notes throughout entire time frame (mean)

2. Calculate the difficulty based on the data
    a. Add discrete factors such as readability and playability of the pattern and multiply it by continuous data such as strain and speed to give a final calculation
        i. Calculate averages
        ii. Calculate the magnitude of continuous intervals of data
    b. Add a deduction system for patterns that are prone to inflate difficulty
        i. A tier/ranking/hierarchy class system of all the patterns and their different presidencies of playability.
        ii. Contribute the averages of all these factors to prevent short bursts of difficulty spikes overinflating maps.

b. Processing of difficulty efficiently and quickly to factor in large quantities of maps.
    i. Efficient use of data structures
        1. Arrays
        2. Hash tables

3. Linked list

ii. Efficient use of searching and sorting algorithms

2) Adding 3D renditions to the adaptation's user interface and gameplay
    a. Use of perspective projection and matrix transformations
        i. Use of model, view and projection matrix in perspective projection
            1. Constructing an identity matrix (A $n \times n$ matrix filled with 1s diagonally)
            2. Applying matrix multiplication using the math library functions
                a. Apply multiplication to model, view and projection matrix
        ii. Apply to the model, view and projection matrix uniforms within shaders
        iii. Allow multiple renditions of this process on different elements for reusability.
    b. Integrate this into the GUI elements
        i. Have the updating of the projection matrix as part of the GUI class process.
            1. Potential inheritance or class association of the GUI and the graphics pipeline class

Thinking Logically

I will require logical thinking in the difficulty calculation system of my adaptation, especially during the forming of algorithms when determining patterns. I will also need logical thinking for handling mathematical concepts and constructs throughout the code. Examples of logical thinking in the development of my adaptation include:

1) Determining the note patterns based on conditions such as position. My algorithms must be able to determine and factor in the positioning of the notes in their respective columns and rows.
   Here is just one example of a note pattern that can be determined using logical thinking. These are not all the examples however it does show the logical thinking involved with determining the algorithms.
   a) The "trill" note pattern
      i) The use of if-else statements or switch/case chains to check If notes are alternating between columns on adjacent rows
         (1) "Trills", where the first note begins on column *n with a column length of l*, where *n=1, l=4*
            (a) If the $n+1^{th}$ note from note n, on the $n+1^{th}$ column is on the $n+1^{th}$ row.

(i) Check if note *n+2* is in the same column as note n and repeat checking of condition (i) until note n+k is not in the same column as note n.

ii) To do this I must use while loops to count the notes until the condition is broken.

iii) Once the condition is broken the count of the loop is I must set the strain factor to the count of the notes and total time taken of the map, *t,* from note *n,* to note n+*k*.

(1) The use of multiplication

2) Determining whether the user is clicking/interacting with the user interface

a) The use of if else statements in coordinate calculations to determine if they are overlapping

i) Determine whether the user's mouse is colliding with the GUI element

(1) Check if the mouse coordinates lie in range with the GUI element's "box size"

(a) Calculating the difference between the GUI element's center and the mouse position.

(i) Integrate magnitude checking using if statements

(ii) Translating local space coordinates to screen space coordinates

1. Multiply local space coordinates by model matrix to get world space coordinates
2. Multiply world space coordinates by view matrix to get view space coordinates
3. Multiply view space coordinates by projection matrix clip space coordinates
4. Pass in the result into the uniform variables in the shader files
5. Lasty set the position of the coordinate within the shader files

(iii) Determining the difference between the GUI element's top left and the center of the element by finding the midpoint of the coordinates.

3) The use of switch/case statements to determine whether the inputs are within the different ranges of accuracy measurement. An example of this can include:

a) Checking if the timing lies within millisecond intervals

i) For example, If the judgment window to achieve "Flawless" hit accuracy is 22.5 milliseconds and the input timing is 16ms then use an if statement to compare then

(1) Use switch/case chains to compare if the input accuracy is equal to the judgment in real time.

(a) Use of an array or hash table data structure to provide instant access to timing data to minimize delay

Backtracking

I will need backtracking for various parts of my algorithms in where I will need to sequentially go back to previous the note *n* from note *n+k* to determine the duration of the and pattern sequence and the time taken to hit the sequence.

I will also need backtracking to go back over the user's gameplay data in their gameplay replay to determine an accurate "grade" based on the average count of accuracy measurements.

Heuristics

I will need Heuristics for forming new algorithms that are like each other. These algorithms will generally have the same format except the position of the columns are shifted. For example, the VSRG note pattern commonly referred to as a "jumptrill" is much like the note pattern "trill" except it consists of notes alternating in two adjacent columns instead of one adjacent column (An example of this would be two notes in the first and second column and on the next row, two notes on the third and fourth columns). The algorithm to determine whether the note pattern is a "jumptrill" will be very similar to the algorithm to determine whether the note is a "trill." Therefore, taking a heuristic approach of going back and reusing the structure of the "trill" note pattern algorithm shall be beneficial to approaching the structure of the algorithm.

Divide and Conquer

I could potentially use divide and conquer in the process of searching during map selection. However, the use of data structures like hash tables may not require me to search for maps using divide and conquer due to their constant access time of O(1) and their ability to be accessed via a key.

Visualization

I will use Visualization during the map creation process. As mentioned earlier, many VSRGs such as StepMania and osu!mania use a visual representation of the notes to allow the editing of the map file. Despite the data itself being in text file, the sequences of notes and their patterns are represented as adjacent icons in a horizontal row of vertical columns. The user can then place or delete notes by clicking on them. The user can also adjust the beat time of the maps and make the visual representation of the gaps between the notes smaller. This thoroughly helps in the map editing/creation process as it abstracts/simplifies the need to edit and place the map data in a file one note at a time.

Another key concept of visualization is based within the concept of VSRGs themselves. VSRGs and rhythm games alike typically do not give a judgment on whether the visual

artifacts are overlapping but give it based on the accuracy of the user's input timing . In cases like StepMania and other VSRGs, notes scrolling towards the receptors give a visual indication of the time to press the key. This is because the notes being rendered on the screen begin at the top of the screen and scroll towards the receptors in a certain amount of time. This means the time taken for the note to begin at the top of the screen and scroll vertically until it visually appears like it overlaps with the receptors, will coincide with the actual timing value that is within the file of the chart; The note acts like a visual cue for the correct time to press the note. In essence, if the user were to input when the note is perfectly in the center of the receptor visually, the user's input timing will the equal to the exact time of "the beat" (based on the current time of "the beat" in the song). This means that the visual aspects of VSRGs are merely just a visual representation of the interval of time that the user must input (via their input device) within.

Complex Calculations

My adaptation shall require multiple complex calculations, many of which are provided by external libraries. Many of these calculations have been abstracted and therefore do not need to be manually written. However, they must be implemented to allow the essential aspects of gameplay, user accessibility and interface. Most of these calculations have been abstracted by the external math library. These calculations include:

1) Calculating the timing offset to draw the arrow based on the users scroll speed
    a) Calculate the vertical magnitude of the distance from the center of the receptor to the top of the column by subtraction. This is to calculate the time by dividing this distance by the user's note velocity.
        i) Transform local space coordinates to view space coordinates
            (1) Multiply local space coordinates by 4x4 matrices and multiply view, model and projection matrices together. Most of these f
                (a) 4x4 Matrix Multiplication:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} =$$

$$\begin{bmatrix} a + 5b + 9c + 13d & 2a + 6b + 10c + 14d & 3a + 7b + 11c + 15d & 4a + 8b + 12c + 16d \\ e + 5f + 9g + 13h & 2e + 6f + 10g + 14h & 3e + 7f + 11g + 15h & 4e + 8f + 12g + 16h \\ i + 5j + 9k + 13l & 2i + 6j + 10k + 14l & 3i + 7j + 11k + 15l & 4i + 8j + 12k + 16l \\ m + 5n + 9o + 13p & 2m + 6n + 10o + 14p & 3m + 7n + 11o + 15p & 4m + 8n + 12o + 16p \end{bmatrix}$$

                (b) Matrix by vector multiplication (for local space coordinates)

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{bmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ mx + ny + oz + pw \end{bmatrix}$$

b) Divide it by the users scroll speed to get the time in milliseconds

c) Display the notes at the top of the screen and let it scroll vertically at the velocity of the users scroll speed. By the time it scrolls to the center of the receptor, the timing of this event would be the same as the time data in the map file data.

2) Adding 3D perspective rotations for GUI elements integrated within perspective projection

a) Matrix rotation

i) Rotation around the X-axis:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{bmatrix} x \\ cos\theta \cdot y - sin\theta \cdot z \\ sin\theta \cdot y + cos\theta \cdot z \\ 1 \end{bmatrix}$$

ii) Rotation around the Y-axis:

$$\begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{bmatrix} cos\theta \cdot x + sin\theta \cdot z \\ y \\ -sin\theta \cdot x + cos\theta \cdot z \\ 1 \end{bmatrix}$$

iii) Rotation around the Z-axis:

$$\begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{bmatrix} cos\theta \cdot x - sin\theta \cdot y \\ sin\theta \cdot x + cos\theta \cdot y \\ z \\ 1 \end{bmatrix}$$

3) Adding translations to GUI elements to give them aspects of animation

a) Translating a vector coordinate into local view space to a new coordinate on the local space

i) Vector by matrix multiplication (see above)

Real Time Processing

I will require real time processing for the main rendering process in the graphics pipeline. Programming the graphics pipeline is the process of turning the raw coordinate and texture data that is currently held in the GPU buffers to the on-screen pixels that are seen on the display monitor. The graphics pipeline must process data and render it onto the window's framebuffer in real-time. Problems with this process happening in real-time will cause graphical artifacts such as tearing and freezing and affect the user's visual perception of gameplay.

The graphics pipeline is essential as it is fundamental for rendering anything on display. The process of the graphics pipeline must be implemented in my source code programmatically (by writing code). In my development, this process is as follows:

1) Vertex Specification/Generation – Specifying the vertex positions and data in local space coordinates on the GPU.
2) Vertex shading – The programmable part of the graphics pipeline in which we can program what to do with the vertex data. What we program will then be executed on each vertex data. Processes like converting to local to view space coordinates occur here.
3) Primitive assembly – Assembling the vertices to form triangles which in turn make up the basis of the shapes seen on screen
4) Rasterization – The process of determining which pixels to be represented based on factors such as depth testing
5) Fragment shading – Another programmable part of the graphics pipeline that determines the final fragment (color) of each pixel to be displayed on screen.

This process must happen all in real-time, typically before each frame is rendered for my adaptation to have its core functionality.

User Requirements

For my adaptation and for most VSRGs in general, the user does not require any foreknowledge or background information requirements on how to play VSRGs. The user can simply engage with the adaptation/VSRG for the first time and be able to experience gameplay. This is due to the VSRGs game mechanics and interfaces being mostly intuitive and indications on the mechanics of gameplay. However, some knowledge and experience on playing VSRGs would be recommended. This can include general knowledge on how to set up key binds for gameplay and general experience with map editing/creation in VSRGs.

The only feasible user requirements that the user needs are a laptop/desktop with a functioning keyboard and mouse and monitor to play the game. However, certain factors such as a keyboard with a high polling rate (1000Hz+) and a mechanical keyboard (preferably with red switches) shall greatly enhance user experience. These factors are not a requirement.

The user will be able to run the adaptation on any form of operating system if the OpenGL drivers are supported within them. Most computer operating systems such as Windows, MacOS and most Linux kernels have their graphics card manufacturers integrate OpenGL drivers into their systems. This means that use will have a wide range of systems to play my adaptation on. Despite my adaptation conforming to multiple operating systems, my

primary operating system will be tailored towards Windows due to it being the operating system that I am using to create the game.

Essential Features

There are multiple factors that are essential to my adaptation. The factors all include different features and their justification.

Feature:

1. A working user interface with different menu screens for the different game modes
    a. A main menu
        i. Working 3D interface and micro interactions
        ii. Section button to allow access to map editing/creation
        iii. Section into gameplay settings
    b. A map selection menu
        i. Displaying map metadata
            1. Song name
            2. Map difficulties
            3. Map BPM
            4. Thumbnail
            5. Map length
            6. Map artist
        ii. Ability to scroll and select each song

Justification:

A working GUI menu is an essential part of gameplay as it is the gateway to a user playing a map and experiencing the main gameplay. If there was no map selection, the user would not be able to decide which maps they can play. This means issues such as playing maps not tailored to their skill level can arise. Therefore, having the choice of maps of different difficulties is an essential feature

Feature:

2. A working map creation and editing system.
    a. Uploading audio files system
    b. Allowing input of metadata for map file
    c. Previewing of song sections during map editing
        i. Ability to inspect the song in real-time
        ii. Automatically set a preview section that plays during the selection of the map

d. A visualized map editing system consisting of four vertical columns that can be navigated through each section of the map
    i. The deletion and placement of notes
        1. Placement of long notes
            a. Algorithm to determine the release of hold note
                i. Account to beat timings
                ii. Ensure the key release is judged
                iii. Ensure key press is judged
    ii. Beat snap divisors
        1. Divisors that to the 4th beat up to 32nd beat
            a. Zoom in / scale editing preview factor for sections with densely populated notes
e. Map saving system
    i. Save map data to text file
    ii. Auto save
        1. Update map in short intervals

Justification:

As mentioned earlier, for the difficulty of a map to be calculated, the map must first be created. Therefore, a system to create and play maps is an essential feature of the adaptation. If there was no visual map system, then maps would need to be created manually via inputting data into a text file. This would mean a very cumbersome process as the timing and column data will need to be input for each note. This would potentially cause errors and difficulties as large quantities of text are data are prone to mistakes.

Feature:

3. The standard working VSRG gameplay – Once a map is selected to play with its desired difficulty, the gameplay shall commence. This consists of a stationary receptor at the bottom of the screen and notes that will scroll from the top of the screen to the bottom of the screen. All of this will happen typically to the "beat" of the map's music.
    a. Notes to be drawn outside the maximum viewport y-value to ensure notes do not just "appear" and seem like they are scrolling in from out of the screen.
    b. Notes to scroll from outside screen to bottom of receptors
        i. Position to be updated every frame
            1. Position updated by distance between the notes starting position and receptor, all divided the user's scroll speed
                a. Scroll speed system
                    i. Accessed through main menu game settings

ii. Numerical value between

Justification:
Without the fully adept, core gameplay mechanics of a VSRG, my adaptation will not have succeeded in amending the problem of difficulty calculation. This is because if the main gameplay system does not function fully intended then the difficulty calculated for the map will be a misrepresentation of the user's true ability as the user's true ability is not truly accounted for. Therefore, the functionality of the gameplay must be working to the correct standard.

Feature :

4. An improved difficulty calculation system to accurately and gauge map difficulty.
    a. Algorithms to process the map data and determine the note patterns within maps
        i. Pattern detection system
            1. Conditions for certain patterns
    b. A system to open the map files
        i. System must integrate the difficulty calculation in real-time after map saving
            1. Map saving system
    c. Different Factors to attribute to difficulty calculation
        i. Speed
        ii. Strain
        iii. Readability
        iv. Playability

Justification:

This is the main feature of my adaptation. As mentioned in research, a new and improved difficulty calculation system will prevent the map difficulty of maps with note patterns prone to being miscalculated from being overestimated/underestimated. I conclude in research that the difficulty should not be calculated based on density alone and other factors such as strain and the maps BPM should be considered into calculation. This way an accurate estimate to map difficulty shall be achieved.

Hardware and Software Requirements

For my adaptation, I will be using the latest version of OpenGL (4.6) as of now. However, most features of OpenGL 4.6 are backwards compatible with version 3.3. Therefore, The minimum hardware and software requirements must be suited to run at least OpenGL 3.3. The minimum hardware and software requirements run my adaptation include:

1. Windows (OpenGL 3.3+)
    a. CPU: Any dual-core based processor e.g. Intel core 2 duo E6850 SLA9U
    b. RAM: 1 GB or more
    c. OS: Windows XP SP2 or later
    d. Graphics Card: Must support OpenGL 3.0; typically requires a card from NVIDIA GeForce 8 series or AMD Radeon HD 2000 series or later.
2. macOS
    a. CPU: Intel-based Mac processor
    b. RAM: 2 GB
    c. OS: macOS 10.6 or later
    d. Graphics Card: Supports OpenGL 3.0, typically found in NVIDIA GeForce 8600 or newer, or ATI Radeon HD series.
3. Linux
    a. CPU: Dual-core processor e.g. Intel core 2 duo E6850 SLA9U
    b. RAM: 1 GB
    c. OS: Kernel 2.6 or later
    d. Graphics Card: Supports OpenGL 3.0; NVIDIA GeForce 8 series or AMD Radeon HD 2000 series or later.

Regardless of the operating system and its version, the main requirement for the user is the appropriate drivers installed for your graphics card to utilize OpenGL. This means any system with the essential OpenGL drivers can be able to utilize my adaptation. Proprietary drivers (like NVIDIA or AMD) often provide better support than open-source drivers for this reason.

Limitations

The main limitation to my adaptation is not integrating compatibility for dance machine support. As mentioned in my research earlier, it would not benefit my stakeholders to include dance machine support due to the expensive of owning a machine and that most of my audience prefer to engage with VSRGs on their own personal home device instead of commuting to an arcade to access a dance machine. This is also due to the limitation of not having direct access to a local arcade or branch that owns dance machines.

Another fundamental limitation to my adaptation is it only consists of four keys of input during main gameplay. Many VSRGs such as Konami's Beatmania series have more than four keys for input, allowing users to hit notes on 5 or more columns. This is another feature found in osu!mania. osu!mania allows users to create and edit beatmaps that allow different amounts of keys. The key count of the maps ranges from as low as a single key of input to as high as ten keys of input. This means that you can play a one key map

using a single finger and a ten key map with ten fingers. The reason for this limitation is that the original DDR and StepMania did not allow more than four keys, so to keep the aspects of my adaptation like StepMania, it is necessary to maintain strictly four keys of input only.

Success Criteria

For my adaptation to be successful, there are a set of essential criteria that by the end of development, my adaptation must have met. These criteria include:

- The user interface must be fully working with access to the different game sections via clickable buttons
- The user interface must also allow keyboard navigation with the use of left/right/up/down and enter input keys
- The user interface must contain a main menu screen with the logo of the game and the ability to use input to navigate to gameplay
- The main menu must contain a background with aspects such as underlap based on different z-coordinates (layering)
- The main menu must contain a text button in which the user can click to indicate progression in map selection i.e. a play button
- The interaction of the start button must successfully transition you into the map selection screen
- The map selection must contain a list of maps ordered and grouped by artist or difficulty
- The map selection system must allow scrolling through each map via mouse wheel or input left/right keys
- The map selection system must display a thumbnail of the maps song cover/ thumbnail image
- The map selection system must display a moving background as the underlay/background for the map selection
- The map selection system must preview a preview of the song in the background of the current selected map
- The map selection should display the maps, metadata and song information within a specified GUI section beside the list of maps
- Each map should have the ability to have more than one difficulty
- The map selection system should allow the changing of the maps different difficulties
- The map selection system should allow the pressing of enter to signify the progression into gameplay

- The map should have standard VSRG gameplay - notes that scroll from out the top of the screen to stationary receptors
- The gameplay must accurately start audio in synchronization of beginning gameplay
- The gameplay must allow the player to hit notes once they are in the timing interval (near) the receptors and give an on-time judgement of accuracy without delay
- The gameplay must keep track of the average accuracy of the user and give an end “grade” based on performance
- The gameplay must save the “grade” as a score inside an external file and
- The user must display their average accuracy as a percentage overall along with their “grade”
- The user must allow the exiting of the “grade” screen and can repeat the cycle of selecting a map and entering gameplay
- The user must have the ability to go back to the main menu via back button GUI
- The user must have the ability to enter map editing/creation via a GUI
- The map editing screen must first have a list of all the current maps
- The map editing screen must have a section to upload an audio file and commence map creation
- The map editing must have columns in which the song can be previewed, and notes can be placed
- The map editing system must have a feature to set a thumbnail/song cover for the map
- The map editing system must have a feature to save maps and update their data in real time
- The map editing system must store the map data in an external folder with an external file in a representable and readable data format e.g. a text file.
- The map editing system must allow returning to the menu screen from saving a map
- The adaptation GUI system must have quick interoperability and interchangeability of each section of game i.e. map selection to map creation to gameplay, etc. Without error
- The menu screen must a GUI to allow access to customization of gameplay i.e. a “settings” button
- The settings section must allow changing of scroll speed and changing of key binds
- The settings section must allow the changing of note/receptor size
- The settings section must allow the changing of scroll direction
- The user must successfully be able to exit for the game by pressing the “x” on the window
- The user can also exit the game via the menu through an “exit” GUI.

Design

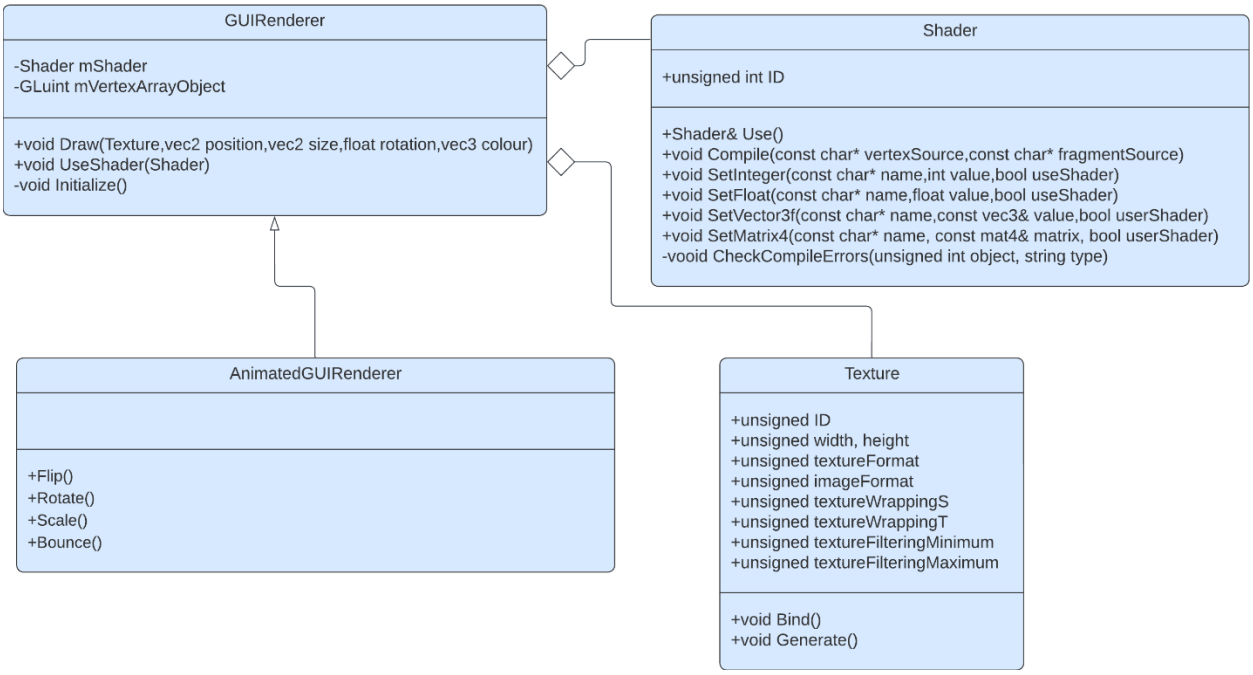
To initiate the design of my adaptation, I first must begin with the design decomposition of the components of my game engine that must be formed first when setting up my adaptation. These components will be responsible for allowing my adaptation to have the core features of its system. These features include the ability to create a window, render user interface, handle input and manage audio. With these components, the larger parts of my adaptation that will be accessible by the user e.g. Chart Editing System can be formed.

GUI Renderer Class

As mentioned in research, my system will require a GUI system that will simplify the process of rendering UI onto the game window. The GUIRenderer class will act as that "GUI system" and will be the base class in which other classes such as the "AnimatedGUIRenderer" class (more on this later) will be derived from. However, the GUIRenderer class itself is an aggregation of two other sub-components that are required to render any kind of interface when working with OpenGL. These two aggregations are a shader class and texture class and will be discussed in detail later. The GUIRenderer takes these sub-components and combines their features to allow the use of an abstracted and reusable "Draw()" function that will allow quick rendering of user interface. This will allow the real time rendering of interfaces such as buttons and textures for GUIs within my adaptation. For example, the GUIRenderer class will allow both the game logo and the background in adaptation's start menu to be render in just two calls of the "Draw()" function. This links back to the research in which having a GUIRenderer class allows abstraction of the somewhat cumbersome process of loading textures, generating the texture data, binding the data to a texture bind spot and using the specified shader and prevention of this process being repeated for each instance of user interface in source code.

Class Diagram

The class diagram of the GUIRenderer shows the composition of the class and its main procedures and variables. The class diagram also shows the aggregated Shader and Texture classes and their functions and procedures and the derived AnimatedGUIRenderer class.



Shader Object

The GUIRenderer class will consist of a Shader object and an unsigned integer that is a numerical identifier to a vertex array object. Earlier in the research, discussion about real-time processing in graphics pipeline mentioned that shaders are programmable and are responsible for processing vertex data so that OpenGL can display them as pixels on the viewport. The shader object in the GUIRenderer class specifies which set of vertex and fragment shader files are being used to process the vertex data (that is in a vertex buffer in which an attribute pointer in the VAO points to). This means the GUIRenderer class will have the ability to be instanced with different types of shaders. This means that vertex data can be processed in different ways at the same time which allows different ways of pixels to be displayed on the screen. For example, an instance of GUIRenderer has its shader variable as a shader that processes the data by moving the texture coordinates on each frame, whilst another instance of GUIRenderer uses a shader that keeps the vertex coordinates static. In this example, the first instance will display an animated “moving” user interface whilst the second instance will display a static use interface. This is crucially important for a system like my adaptation as the rendering of static and animated interfaces will be used frequently in areas such as main gameplay e.g. for the rendering of scrolling notes but the rendering of a background image remaining static.

Vertex Array Object

The vertex array object is responsible for disambiguating the different types of data that are stored within the vertex buffer. In the case of my adaptation, the different types of data

within the vertex buffer objects include the texture coordinates (the coordinates of the texturized pixel) and actual vertex coordinates for a quad. These types of data will be stored as an attribute that the vertex array object points to. The reason for having a vertex array object is due to OpenGL 3.3 requiring at least one vertex array object that points to one vertex buffer object for any rendering to be done. Another reason for using a vertex array object is the key feature of OpenGL allowing the binding and unbinding vertex array objects once they are set. What this means is that once the attributes of the vertex array are set, they do not have to be set again for each time the user wants to use the vertex array object.

Initialize() procedure

This key feature ties in with the use of the "Initialize()" procedure for the GUIRenderer class. The Initialize() procedure is responsible for initializing the vertices required to form a quad and the texture coordinates. Once this data has been initialized, it is bound to the vertex attribute object. This means that the same vertices will be used to draw every user interface. This is important as the process of rendering a texture will become standardized which leaves no ambiguity and thus less room for errors .

UseShader() procedure

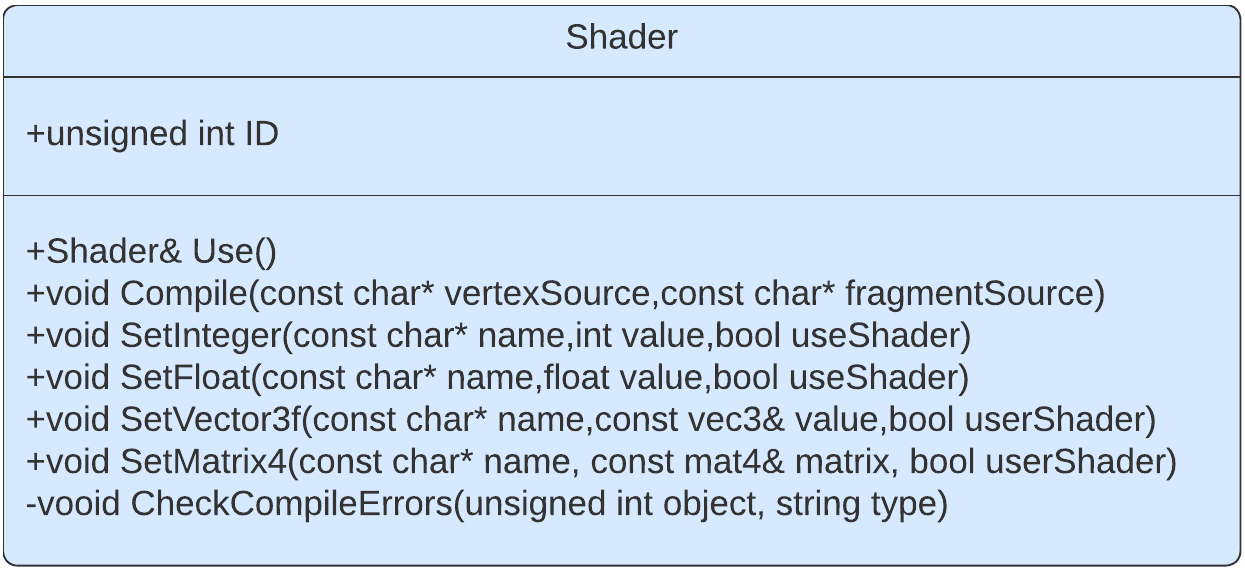
The "UseShader()" procedure will allow the changing the currently in use shader object to another shader object. This will be useful in circumstances where rendering a different user interface requires a different type of shader. For example, if one interface requires rendering of a GUI with 3D elements and another interface only requires rendering of a 2D GUI, then there will be two different shader files requiring perspective projection and orthographic projection respectively. In this case, the use of the "UseShader()" will allow the dynamic rendering of both 2D and 3D objects in my adaptation.

Shader Class

As shown in the class diagram above, the shader class is used to encapsulate the process of loading and using shader files. The shader class will take the paths of the fragment and vertex shaders as input, read the file's source code, compile the source code and create the shaders. Once the shaders are created, an OpenGL shader program is made, and the pair of shaders are attached to the program. Afterwards, I can use the shader program to process vertex data program by calling it my code when processing vertex data.

Class Diagram

This is the class diagram of shader in independence.



ID

The shader class contains an unsigned integer variable called “ID” to represent the shaders numerical identifier. Each time OpenGL compiles and creates a shader program, OpenGL stores its reference as an unsigned integer. The ID variable stores a copy of the reference to allow specific use of the shader program for any instance of the shader class.

Use() procedure

The Use() procedure in the shader class will be used for quickly changing the shader program to the program of the current shader object via it’s “ID”. This is useful for allowing multiple scene rendering (2D and 3D) and for rendering in different formats concurrently.

Pseudocode for Use() procedure

The Use() procedure will only have the function of switching the currently in use shader program. The key feature is with this function is that each shader class has an “ID” which is the numerical identifier for the shader. This means that calling the Use() function will switch the program to the instanced shader class program.

```
public procedure Use()
    glUseProgram(this.ID)
endprocedure
```

checkCompileErrors()

A function part of the Shader class that will check for any compilation errors when compiling the source code from a shader file. If any error is found, then the error is logged with a description of the error and the location of the error. The checkCompileErrors() has parameters for the shader ID and the type of compile error that Is being checked.

Compile() procedure

The Compile() procedure is responsible for taking in the sources code of fragment and vertex shaders, reading the source and compiling the source into shaders. Once the shaders are made, they are attached to an OpenGL shader program.

Pseudocode for Compile() procedure

The Compile() pseudocode will consist of creating the shaders using OpenGL's built in functions, attaching the source code of the shaders then following the procedure to compile and link the shaders. A key feature of the Compile() pseudocode is the deletion of the shaders once they have been created. This is because the actual shader files themselves are not needed once the OpenGL shader program has been compiled and created.

```
public procedure Compile(vertexShaderSource, fragmentShaderSource)
    // The vertex shader and the fragment shader source code will
    be passed in as a string and will be compiled for each shader
    respectively


    // Create and compile the vertex shader
    vertexShader = glCreateShader(GL_VERTEX_SHADER)
    glShaderSource(vertexShader, vertexShaderSource:ByRef)


    glCompileShader(vertexShader)
    checkCompileErrors(vertexShader)


    // Create and compile the fragment shader
    fragmentShader = glCreateShader(GL_FRAGMENT_SHADER)
    glShaderSource(fragmentShader, fragmentShaderSource:ByRef)
```

```
        glCompileShader(fragmentShader)
        checkCompileErrors(fragmentShader)

        // Create an empty shader program
        this.ID = glCreateProgram()
        // Attach the shaders to the shader program
        glAttachShader(this.ID, vertexShader)
        glAttachShader(this.ID, fragmentShader)

        // Link the shaders once they have been compiled and attached
        glLinkProgram(this.ID)

        // Once the programs have been made, the shaders can be
        discarded to free up memory
        glDeleteShader(vertexShader)
        glDeleteShader(fragmentShader)
endprocedure
```

Uniform Variable Setters

Within the shader class, the SetInteger(), SetFloat(), SetVector3f, SetMatrix4f() procedures will be responsible for updating the uniform variables inside the shader. This means that variables that have been set in the shader to take in some form of data, e.g. an integer variable to update the time in milliseconds, can be dynamically changed to any form from within the source via these functions. These functions will be responsible for changing the variables required for updating the matrices for the GUI elements. An example use could be dynamically changing the position of an element by performing a matrix transformation on the model matrix and setting the model uniform to the result of the matrix transformation. This will cause the position of the rendered GUI element to be shifted.

Pseudocode for Uniform Variable Setters

The pseudocode for the uniform variable setters will have the option to identify the variable based on its name using the built in OpenGL glGetUniformLocation function. This is to

allow easy access to the variable in real time without the need to call the uniform location finder function. The variable setters also give the option to use the shader when setting the variable for ease of access when switching between shaders.

```
// Set Float uniform variable
public procedure SetFloat(name, float value, useShader)
    if useShader then
        this.Use()
    glUniform1f(glGetUniformLocation(this.ID, name), value)
    endif
endprocedure
// Set Integer/Sample2D uniform vairable
public procedure SetInteger(name, value, useShader)
   if useShader then
        this.Use()
    glUniform1i(glGetUniformLocation(this.ID, name), value)
   endif
endprocedure
// Set Vector3 uniform vairable
public procedure SetVector3f(name, value:ByRef, useShader)
    if useShader then
        this->Use()
    glUniform3f(glGetUniformLocation(this.ID, name), value.x,
value.y, value.z)
    endif
endprocedure
```

Test Data

This table indicates the test data I will use for testing the shader class during the iterative development of my adaptation.

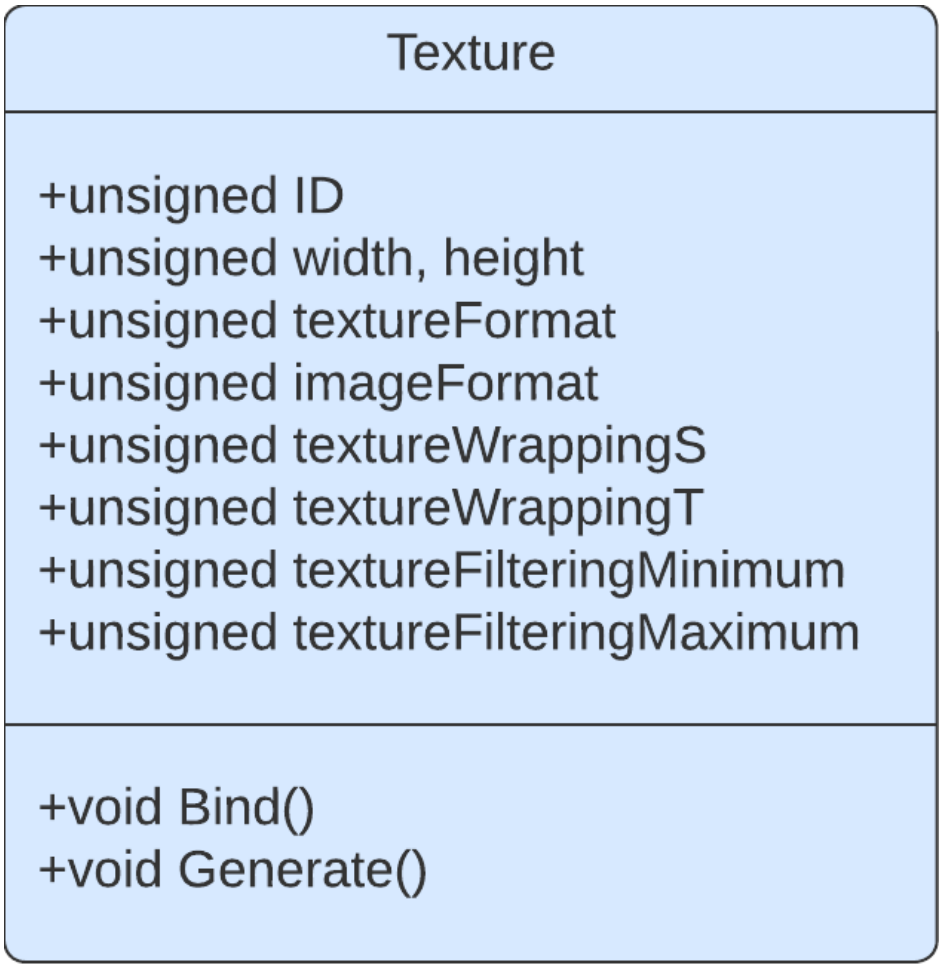
Test	How I will test the data	Justification For Test Data	Expected result
Compile() procedure	Pass in valid fragment and vertex shader source code with no syntax errors or logic errors	This is valid data to show that all OpenGL programs require at least one fragment and vertex shader for any rendering to occur	No shader compilation or syntax errors shown on the console and correctly rendered graphics on viewport
Compile() procedure	Pass in fragment and vertex source code with no syntax errors but with logic errors	This is boundary test data to suggest that shader source code must logically be correct and follow the correctly OpenGL procedures	No compilation or syntax errors shown on the console, however incorrectly rendered graphics or completely black screen on viewport
Compile() procedure	Pass in fragment and vertex source code with syntax errors	This is an invalid test data to test the error logging mechanisms in code	Immediate console errors that indicate the syntax error in source code alongside completely black screen on viewport.

Texture Class

The texture class will be responsible for storing the characteristics that will allow OpenGL to take the path of an image, generate a texture and set a bound OpenGL texture bind slot.

Class Diagram

Below is the class diagram of texture in independence



ID

Much like the shader class, each texture object has an unsigned integer used to represent the texture's numerical identifier. In OpenGL the maximum number of textures that can be bound simultaneously is 16, therefore having numerical identifiers to indicate which texture is bound will allow interchangeability between textures. Furthermore, each texture class stores their "ID" meaning the need to recall the numerical identifier is completely abstracted as the class can call its own "ID" via the "this->ID" pointer dereference.

Texture Format and Image Format

The textureFormat and imageFormat variables within the Texture class will be used to determine whether the texture being generated will require alpha values or not. This will be used for loading transparent images files such as the .PNG file format to be generated as textures.

Texture Wrapping

The textureWrappingS (X-axis) and textureWrappingT (Y-Axis) are the variables that determine the wrapping direction of the texture. This is how texture will be presented if it is a repeating texture or if the texture coordinates get shifted outside the displayed vertices.

Texture Filtering

As discussed in the research, the texture filtering must be determined when setting the texture properties. Texture filtering is the process in which OpenGL must figure out the texture pixel to map the texture coordinate to. This is because texture coordinates can be any floating-point number and are not precisely on the pixels themselves. Thus, OpenGL must find a way to map the texture pixel correctly via a filtering method. There are different kinds of texture filtering and each one can be applied when scaling the image resolution up or down. Thus, there is a need for both a textureFilteringMin and textureFilteringMax variable.

Generate() function

The Generate() function within the texture will carry out the process of taking in image data, setting the OpenGL texture parameters to the image properties and finally binding the texture to an OpenGL bind slot.

Pseudocode Texture Constructor

The constructor for the Texture class will be used to initiate the properties of the texture. The declared values will be the default values; however, they can be set to any value when the texture object is being instanced.

```
// Set default values for when instancing a texture object

class Texure

     public procedure new Texture

         Width = 0

         Height = 0

         iamgeFormat = GL_RGB

         textureFormat = GL_RGB

         textureWrappingS = GL_REPEAT

         textureWrappingT = GL_REPEAT

         textureFilteringMin = GL_LINEAR
```

```
        texturingFiltergMax = GL_LINEAR
        glGenTextures(1, &this->ID)
    endprocedure
endclass
```

Pseudocode For Generate() Function

The Generate() function will be responsible for setting the OpenGL texture state variables and calling the bind texture function calls.

```
public procedure Generate(width, height, data)
    this.Width = width
    this.Height = height

    // Create Texture

    glBindTexture(GL_TEXTURE_2D, this.ID)

    // Set texture image format

    glTexImage2D(this.textureFormat, width, height, 0,
this.imageFormat)

    // Set Texture wrap and filter modes

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
this.textureWrappingS)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
this.textureWrappingT)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
this.textureFilteringMin)
```

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
this.textureFilteringMax)


    // Unbind texture

    glBindTexture(GL_TEXTURE_2D, 0)

endprocedure
```

Test Data

Test	How I will test	Justification For Test Data	Expected result
Generate() function	Pass in correct image data parameters including correct width and height of image, the correct image format and the actual image data.	This is valid data to suggest that the image data extraction library has successfully extracted image data to be texturized	The quad being rendered using the texture should be fully texturized correctly with no defects
Generate() function	Pass in correct width, height and image data however the image formatting is incorrect. For example, the image has alpha values, but the image formatting is not set to check them	This is boundary test data to show the errors that will occur when trying to load different image types without considering the image format when texturizing. E.g. loading a PNG image as a texture.	The quad being rendered will be texturized but the texture will have visual defects such as tearing, image warping or the image being vertically flipped.
Generate() function	Pass in incorrect image data all together and/or image formatting.	This is invalid data to show that texturization is highly dependent on whether the image data conversion function converts the image correctly.	The quad being rendered will not be texturized at all and will only show the fragment of the quad and potential runtime errors will output on the console.

Draw() procedure

The “Draw()”procedure (for the GUIRenderer class) will consist of the actual texture object that is used in reference when drawing and the transformations that can be applied to change the final output of the interface to be drawn. These transformations include translation, scaling, rotation and an additional parameter to transform the texture coordinate color of the texture. This is another important feature as the process of setting matrix variable and continually updating them in my code for each case of

Pseudocode For Draw() procedure

The Draw() function will take into consideration the parameters that have been passed into the function when drawing and will pass these parameters into the appropriate matrix arithmetic functions. The main key feature of the Draw() function is that the order of transformations is in reverse order. This is because matrix multiplication is non-commutative and is performed right from left.

The mat4 model variable is responsible for the initial identity matrix that does not have any transformations applied to it. For scaling, the model variable is first translated by half of its size. This is because the origin of the texture will be from the top left. To maintain the centered position of the texture must be scaled, translated by half its size, have any other transformations performed and then scaling transformation is reversed.

Another key feature of the Draw() procedure is the use of default parameter variables. This is because the user may not always want to input parameters for all variables.

Once the transformations have been performed and sent to the shader, the texture is bound, and the texturized quad can be drawn onto the viewport.

```
public procedure Draw(Texture, position = (0,0,0), size =
(10,10,10),rotate = 0, colour = (1,0,1.0,1.0))


    // Create identity matrix


        model = mat4(1.0f)


    // Apply any transformations to identity matrix
```

```
        model = translate(model, vec3(position, 0.0f))



        model = translate(model, vec3(0.5f * size.x, 0.5f *
size.y, 0.0f))
        model = rotate(model, radians(rotate), vec3(0.0f, 0.0f,
1.0f))
        model = translate(model, vec3(-0.5f * size.x, -0.5f *
size.y, 0.0f));
        model = scale(model, vec3(size, 1.0f))


        this.shader.SetMatrix4("model", model)
        this.shader.SetVector3f("GUIColor", colour)


        // Bind texture to texture bind slot
        texture.Bind()


        // Set currently in use vertex array object to the
class's vertex array object


        glBindVertexArray(this.vertexArrayObject)


        // Draw the triangles to form the texturized quads


        glDrawArrays(GL_TRIANGLES, 0, 6)
        glBindVertexArray(0)
endprocedure
```

Test Data

Test data	How I will test the data	Justification For Test Data	Expected result
Draw() procedure	Pass texture object and correct position, size, rotation and color value.	This is valid test data to show the process of drawing successfully rendering an image on screen	The texturized quad showing the GUI image being rendered at the correct position, at the correct size, and color value
Draw() procedure	Pass in texture object but not all parameters are inputted. For example, only passing in the size and position of the texture without inputting the color or rotation.	This is also valid test data due to the use of default variable declarations within the parameters. This means that if the user does not input anything for the function parameters, then it will resort to the default value stored in the function declaration. This is justified because during development, most textures will remain static with size and position. These default values will typically not affect the image in any manner.	The texturized quad being rendered at the correct position and size with no other resultant features. For example, the default value for position is (0,0,0) meaning if there is no position value entered then the position of the quad will be in the top left corner of the screen.
Draw() procedure	Pass in the texture object with the size at (0,0,0) and the position this maximum size of the viewport	This is boundary test data as GUI elements being rendered must be within the limits of viewport otherwise the element's pixels will be clipped (not rendered on screen)	The texturized quad's dimensions will be the size of the viewport's boundaries.

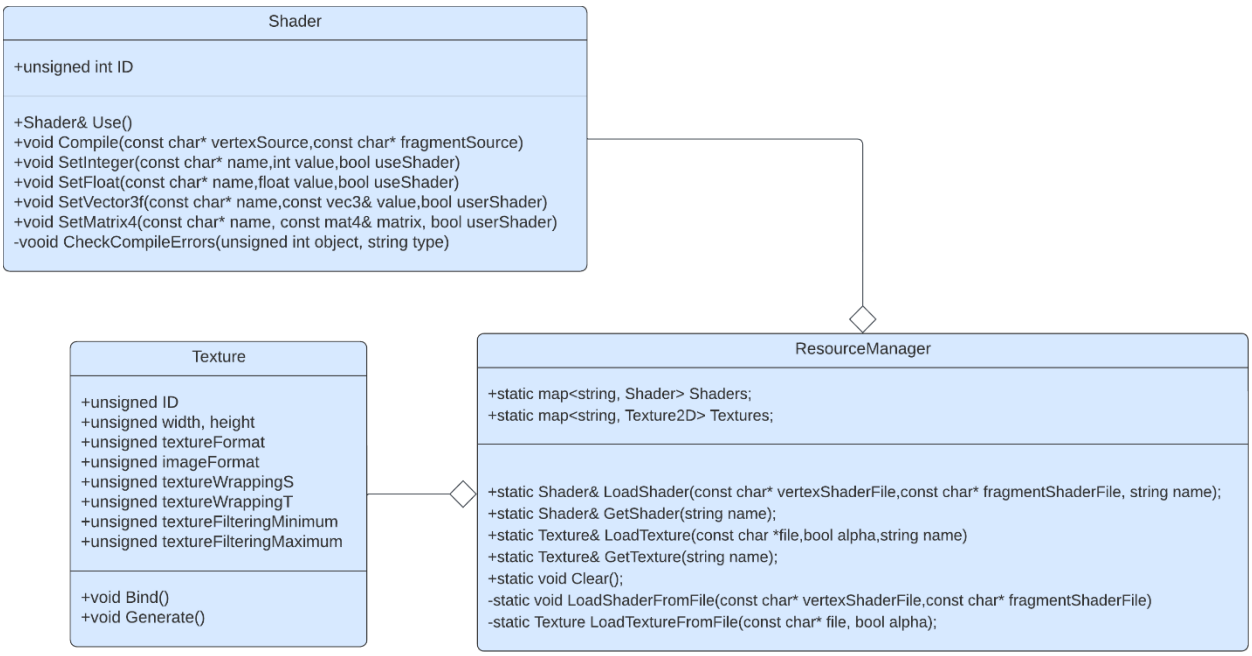
Draw() procedure	Pass in the texture with no input parameters	This is boundary test data as the texture will immediately resort to its default values	The texture will be rendered according to the value of the default variables. In this case it would render in the top left of the screen and with a size of 10x10 pixels.
Draw() Procedure	Pass in the texture with a position and size that is greater than the viewport	This is boundary test data as the data is still valid, however it may not be intentional	The texture will be rendered correctly; However, it will be oversized and parts of the texture that are outside the viewport will be clipped.

Resource Manager Class

The Resource Manager class will be responsible for all the associated asset management. This includes loading shaders from .GLSL files and loading textures from image paths. The resource manager will greatly abstract the process of converting external resources to be useable by OpenGL by aggregating the Shader and Texture classes. The resource manager will not be responsible for the actual conversion of the assets, only the loading of paths of assets to be used by the Shader and Texture Class. However, the resource manager will make use of the Shader and Texture Class by calling the functions that convert the shader paths into shaders and image files into textures within the resource manager function calls. This greatly simplifies and abstracts the process of loading assets as the use of the class will allow the paths to be input into the function petameters and ensure direct conversion into the assets with no further need inducing any other function calls.

Class Diagram

The class diagram for the Resource Manager shows the use of the Shader and Texture class as aggregated data types that will be used in asset management and the associated data structures used for these objects.



Data Structures used in Resource Manager

Within the Resource Manager class, I will use data structures that will be useful for allowing access to the instanced shader and texture objects.

Hash Tables

The Resource Manager class includes the use of the C++ map<> data structure which is a hash table. These hash tables alongside the other variables will be static variables (In C++ and other OOP languages, static refers variables that last the whole length of runtime). This is because shaders and textures will need to be accessed constantly throughout the program as it runs. I have chosen the use of a hash table to store the shaders and textures due to hash tables having an access time of O(1) and elements being accessible with a key identifier. For example, using the LoadShader() function for loading a shader that uses perspective projection for 3D rendering, the user can store the shader's name parameter in the function as "3D renderer". This means there will not be a need to remember the numerical identifiers for the actual shaders themselves but instead a much more readable and intuitive description of them. As well as this, the use of hash tables is justified due to the need to access the shaders in real-time. As discussed in research, shaders are used in the graphics pipeline. If there are delays loading shaders, then the process of rendering may be delayed and cause dropped frames and unintentional visual delay.

LoadShaderFromFile() function

The LoadShaderFromFile() function will be responsible for retrieving the paths of the shader files and instancing shader objects within the function. As mentioned, the

instanced shader objects themselves will be responsible for compiling and creating the OpenGL shader programs via their compile functions.

Pseudocode for LoadShaderFromFile()

The pseudocode for the LoadShaderFromFile() function first begins by loading the contents of shader file from the shader path as a string. Afterwards the function instances the shader objects. This is important as this allows the shader Objects' compile functions to be called. Finally, the function uses the shader objects to compile the string contents. This shows how the resource manager abstracts the process of loading shaders down to only requiring the paths the shader. Once the shader has been created, the object instanced is stored in the shader's hash table to be accessed.

```
public function loadShaderFromFile(vertexShaderPath,
fragmentShaderPath)



    // Open files

    vertexShaderFile = openRead(vertexShaderPath)
    fragmentShaderFile = openRead(fragmentShaderPath)

    // Read file's buffer contents into streams

    vertexShaderString = myFile.readLine()
    fragmentShaderString = myFile.readLine()

    // Close file handlers

    vertexShaderFile.close()
    fragmentShaderFile.close()
```

```
    // Instance the shader object(s)
   shader = new Shader

    // Compile the shader source code

    shader.Compile(vertexShaderString, fragmentShaderString)

    // Return the shader object

    return shader;
endfunction
```

LoadTextureFromFile() function

The LoadTextureFromFile() will follow the same procedure as in the LoadShaderFromFile() function except it will include the additional steps required to convert and image into texture data and pass that data to be generated.

Pseudocode for LoadTextureFromFile()

The function will include a Boolean parameter to determine if the image path's file format contains alpha values e.g. A .PNG file with a transparent background. If there are alpha values, then the texture's image format attributes are updated correctly.

```
public function loadTextureFromFile(file, alpha)
    // Create texture object
    Texture texture

    // Consider the issue of alpha values
    if alpha then
        texture.imageFormat = GL_RGBA
```

```
        texture.textureFormat = GL_RGBA
    endif

    // Load image data using library function
   data = stbi_load(file, width:ByRef, height:ByRef, nrChannels,
0)

    // Issue a console error in case of problems loading texture
data
    if data != true then
      print("Failed to load texture from”, file)
    endif

    // Generate texture using texture object function

    texture.Generate(data)
    return texture
endprocedure
```

LoadShader() Function

The LoadShader() function will be responsible for appending shader objects in the hash table. This shows the aggregated link to the GUIRenderer class. This is because to allow different forms of rendering, there will be a need to pass different forms of shaders into the GUIRenderer instance. Thus, the need for loading the different shader files is justified.

Pseudocode for LoadShader()

The LoadShader() pseudocode consists of using the loadShaderFromFile() function to load the shader files from the path and then storing them within the hash table

```
public function LoadShader(vertexShaderPath, fragmentShaderPath,
name)
```

```
    Shaders[name] = loadShaderFromFile(vertexShaderPath,
fragmentShaderPath)
    return Shaders[name]
endfunction
```

LoadTexture() Function

The LoadTexture() functon will be responsible for returning the loaded textures that have been loaded from their image paths.

Pseudocode for LoadTexture()

The LoadTexure() pseudocode consists of using the loadTextureFromFile() function to load the generated textures from image the path and then storing them within the hash table.

```
public function LoadTexture(file, alpha, name)
    Textures[name] = loadTextureFromFile(file, alpha)
    return Textures[name]
endfunction
```

Test Data

Test Data	How I will test the data	Justification of the data	Expected result
loadShadeFromFile()	Passing in a correct path to both a vertex and fragment shader	This is valid test data to show the successful loading of a shader source file	The shader object will be instanced and will proceed to be compiled via the Shader class Compile() function
loadShaderFromFile	Passing the same path for the vertex and fragment shader	This is boundary data to show that for a shader object to be correctly instanced, there must be both a separate fragment and vertex shader.	The shader objected will be instanced and compiled however there will possible runtime errors due to the ambiguation of uniform variables
loadShaderFromFile()	Passing in a path to a shader that does not exist or in incorrect	This is invalid test data to show that for shaders to be correctly loaded	There will be immediate console runtime errors

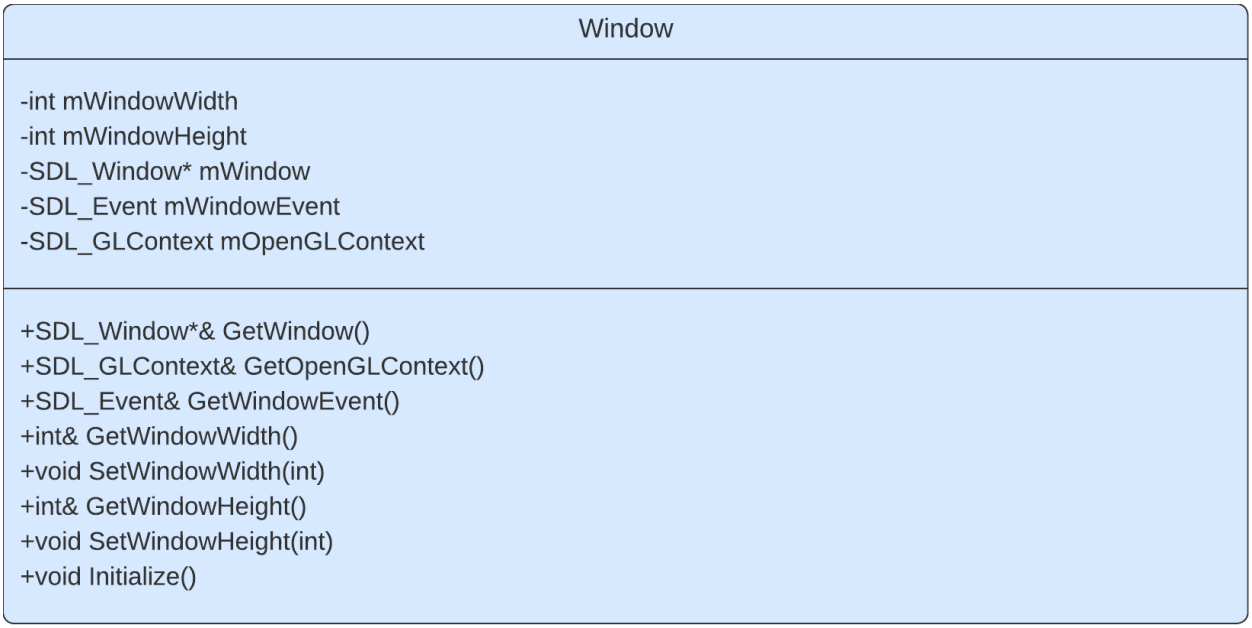
		then the path must be led to a correct shader.	
loadTextureFromFile()	Passing in a correct path an image file that doesn't contain alpha values e.g. JPEG	This is valid test data due to show the consideration of alpha values. This is justified to test allowance interoperability between different image formats	The image will be texturized however any aspects of transparency will be replaced with white pixels
loadTextureFromFile()	Passing in a path that does contain alpha values e.g. PNG	This is valid test data to show the acceptance of images with alpha values	The image will be texturized, and any aspects of transparency will be fully transparent.

Window Class

As discussed in the research, the window class will make use of the SDL2 external window library and its functions to set the properties of the game window and initialize a working game window. The class will hold properties for the window such as its dimensions, the references to the OpenGL context, the pointer to the window itself and a reference to the window events.

Class Diagram

The class diagram of the Window class shows that window class itself is the most independent class. This is because it does not require any other aggregations to set up a window. It only takes SDL2's library functions and organises them into the system required for my adaptation.



Getters and Setters

The GetWindow(), GetOpenGLContext(), GetWindowWidth(), SetWindowWidth(), GetWindowHeight() and SetWindowHeight() functions and procedures will all be responsible for returning and setting the references of the desired variables in the window class. The purpose of this is to maintain the concept of encapsulation and provide clean and authorized access to the member variables within the class.

Initialize() procedure

The initialize procedure will be called when initiating a game window. It is where the window creation happens and it will be responsible for setting OpenGL states vairables. It will also be responsible for checking if all libraries required for my adaptation are functioning.

Pseudocode For Initialize() procedure

The pseudocode for the Initialize) procedure will contain the necessary code to create a window and create an OpenGL context, using SDL2. Furthermore, the pseudocode shows the error checking procedures that will take place in case of any errors during initialization.

```
public procedure Initialize()
      // Initialize SDL

    if SDL_Init(SDL_INIT_VIDEO) < 0 then
```

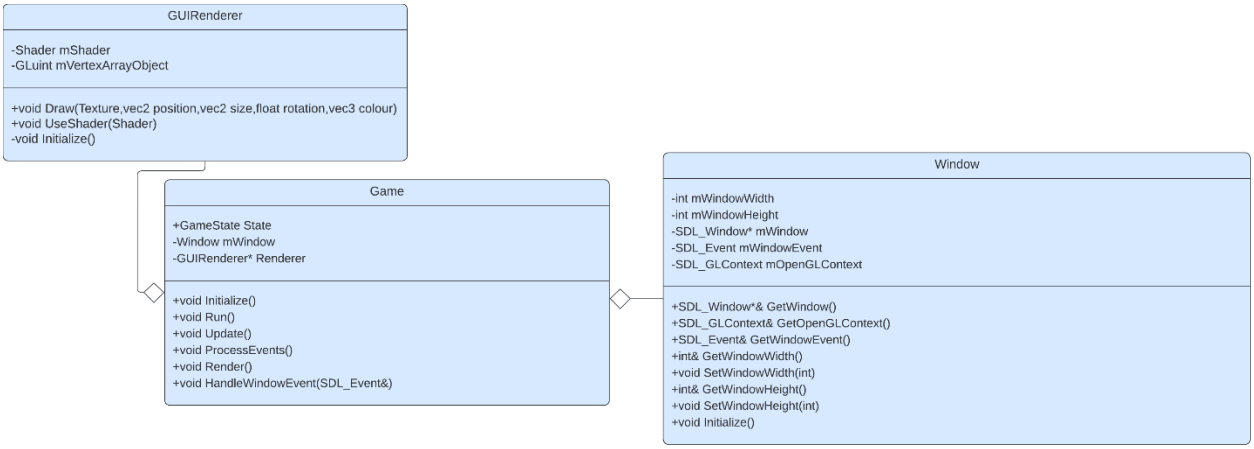
```
        SDL_LogCritical(SDL_GetError())
    endif


      // Create Window


    mWindow = SDL_CreateWindow("Game", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, mWindowWidth, mWindowHeight,
SDL_WINDOW_OPENGL)
    if mWindow == nullptr then
        SDL_LogCritical(SDL_GetError());
        SDL_Quit()
    endif



     // Create OpenGL context
     // Check for errors


    mOpenGLContext = SDL_GL_CreateContext(mWindow)
    if mOpenGLContext == nullptr then
        SDL_LogCritical(SDL_GetError())
    endif
endprocedure
```

Game Class

The game class will act as an uber class that aggregates the GUIRenderer and Window class. This class will intend to organize the game code whilst also decoupling all the window code from the game code. The game class will also hold functions for processing input and handling window events. It will also contain the “main loop” that makes use of the GUI Renderer and is where all rendering activities happen. Furthermore, the class will also store an enumeration of the current game state and update all necessary functions.

The game class will also heavily utilize the resource manager class for loading the assets required to be used for the game

Class Diagram



Initialize() procedure

The Initialize() procedure for the game class will be responsible for setting up projection matrices, initializing shaders, initializing textures and setting uniform variables. Once these variables are initialized, they will not need to be initialized again (excluding textures). This step of initialization links back to the thinking ahead section of my research. This is because once I implement, I do not need to write a separate system to reproduce more features in my adaptation. This modularity saves time, improves code readability and will reduce the chance of logical and syntax errors.

Initialize() pseudocode

The first key feature of the Initialize() procedure is the use of the orthographic projection matrix. Throughout my research, I discussed the need to use perspective projection for the 3D aspects of my adaptation. As well as that, I will have a majority of 2D aspects such as the menu GUIs. Due to orthographic projection keeping the distance properties of every model from the camera constant, there is no perspective. Thus, everything in orthographic projection remains 2D which is especially useful for rendering models that are strictly meant to be kept in 2D.

The next key feature of the Initialize() is the utilization of the Resource Manager functions to greatly simplify and abstract the process of making new shaders from shader files and loading textures form image files. This is shown as the process of how loading assets works is hidden away. This creates a style of “black box” coding which allows me focus on what the function does and the intention of the use of the function.

Once the shaders have been loaded, the GUI Renderer is instanced, and the game engine is ready to be used correctly.

```
public procedure Initialize()


    // Set up orthographic projection matrix


    mat4 orthographicProjection = ortho(0.0f,
mWindow.GetWindowWidth(),mWindow.GetWindowHeight(), 0.0f, -1.0f,
1.0f)


    // load shaders


    ResourceManager::LoadShader("shaders/[YOUR_VERTEX_SHADER_PATH_
HERE" "shaders/[YOUR_FRAGMENT_SHADER_PATH_HERE],"[SHADER NAME]");


    // Set uniform variables within shaders


    ResourceManager::GetShader(“[SHADER
NAME]).Use().SetMatrix4("projection", orthographicProjection);
    ResourceManager::GetShader(“[SHADER
NAME]).Use().SetInteger("image", 0)


    // Instance GUIRenderer


    Renderer = new GUIRenderer(ResourceManager::GetShader("[SHADER
NAME"));
    // load textures
    ResourceManager::LoadTexture(“assets/[PATH_TO_YOUR_IMAGE.PNG]”
, true, “[NAME TO IDENTIFY YOUR TEXTURE]”)
```

```
endprocedure
```

Start Menu

For my adaptation's start menu's design, my approach is a dynamic between the old and retrospective arcade dance design theme that is across older VSRGs whilst keeping the modern functionality and useability. The general aesthetic is meant to conform to the iconic designs of older arcade dance games by using features such as arrows, musical terms and silhouettes of people in dance positions,

Game Cover (Logo)

To begin with, one of the most fundamental aspects of a VSRG is its game cover/logo design. It is often the first impression that users behold and often what they judge the expected outcome of the gameplay to consist of; It forms the initial expectation of what the game is about. For this very reason my game design must include the features of that will suggest that it is aimed at both the older and younger generation alike.

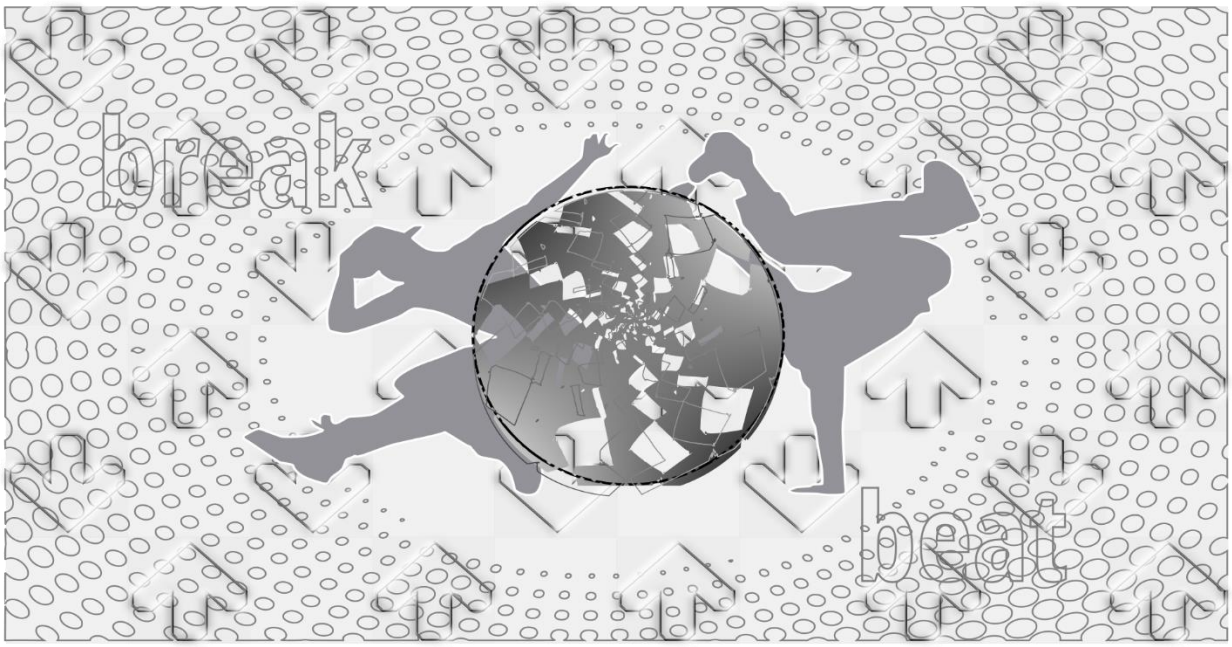
Many older VSRGs designs have a focus on bold, smooth and perspective text. This is prevalent in examples mentioned in research such as ITG and DDR. To stay inclined to the theme of older VSRGs and the general arcade aesthetic found in them, my adaptation's logo design will consist of smooth text with a horizontally tilted perspective. This design choice is intended to mimic the 3D perspective shifts commonly found in older VSRGs designs like DDR.

During the era of many old arcade games and VSRGs the graphical processing power was not advanced as the modern era. This meant the polygon count used to render 3D models and designs was much lower. Therefore, my design consists of a lower sample of curve segments in text rendering. This is to give it the same nostalgic essence found in many older VSRGs and arcade games alike (An example being SuperMario64) that will appeal to the older generation.

Smooth, bold text with perspective is a feature of VSRGs design logos that the older generation tend towards. Examples in which this is found are DDR and Beatmania. These games have their logo design text stylized with a smooth but heavy outline. This is also prevalent in many modern VSRGs such as Friday Night Funkin', too therefore adding outline to my design will appeal both to audiences. Because of this, I implemented my design with an initial white outline of the text then an additional black outline and additional perspective shift. This is to maintain the maintain the general 3D text aspects of older VSRGs.

Background

One theme that is prevalent in many rhythm games and VSRGs alike is distinctive features that make it clear to the user what the game is about. Earlier in research, the evidence I gained suggested that features like arrows, musical terms, people dancing etc. are what make it clear to a user that game is in the genre of a rhythm game. Therefore, for my design, I implemented a background design with the use of an array of alternating hollow arrows to enhance the arcade dance game style theme. I also implemented silhouettes of people dancing, a clear dotted overlay and the use of musical terms to give the 90s music game vibrant feel that is commonly found in old VSRGs. This will benefit the older stakeholder age range and further allude to the design of older VSRGs.



Furthermore, as I commence development of my adaptation, this design will be implemented as an animated moving background to add to the classic dance game aesthetic and generally give the design a fuller and vibrant appeal. My justification for this is the aim provokes the feelings of reminiscence and nostalgia within the older generation as discussed in research earlier.

User Navigation Assist

As mentioned, my design will follow the typical design attributes to older VSRGs however to benefit all my stakeholder audience age ranges, implementing modern functionality and useability is a must. Part of the functionality and useability that will benefit the audience of my younger stakeholder is clear and instructed guidance on how to navigate the user interface. Modern VSRGs such Friday Night Funkin' tell the user how to This is through keystroke indication or user input guidance, typically either by directly addressing the keys to be pressed or by showing an image or sort. To add to the modern functionality and useability of my design, I implemented a user navigation assist bar that will be situated at the bottom of the start menu. The bar will give a clear aid on which keyboard inputs are required to navigate the user interface through buttons to represent keyboard icons. My justification for this design is to generally enhance the intuitiveness of my adaptation and add to the modern functionality and useability.

Start and Exit Buttons

As well as features relating to dance games i.e. arrows, musical terms and the use of bold perspective text, the use of stylish and centered text that a user must interact with via user input to allow progression into main gameplay. This general theme of text signifies to the user that they must press a key, e.g. enter, to start the game is a common occurrence in many old arcade games and VSRGs alike. Furthermore, many older arcade games tend to leave the text without any form of encasement around the text i.e. the text has no aspects of being a “button” but purely just the text alone. Therefore, for my start menu’s user interface, I implemented this style of start and exit buttons to mimic the retrospective menu design of older VSRGs and keep on the pattern of an older design that is appealing to the older audience.

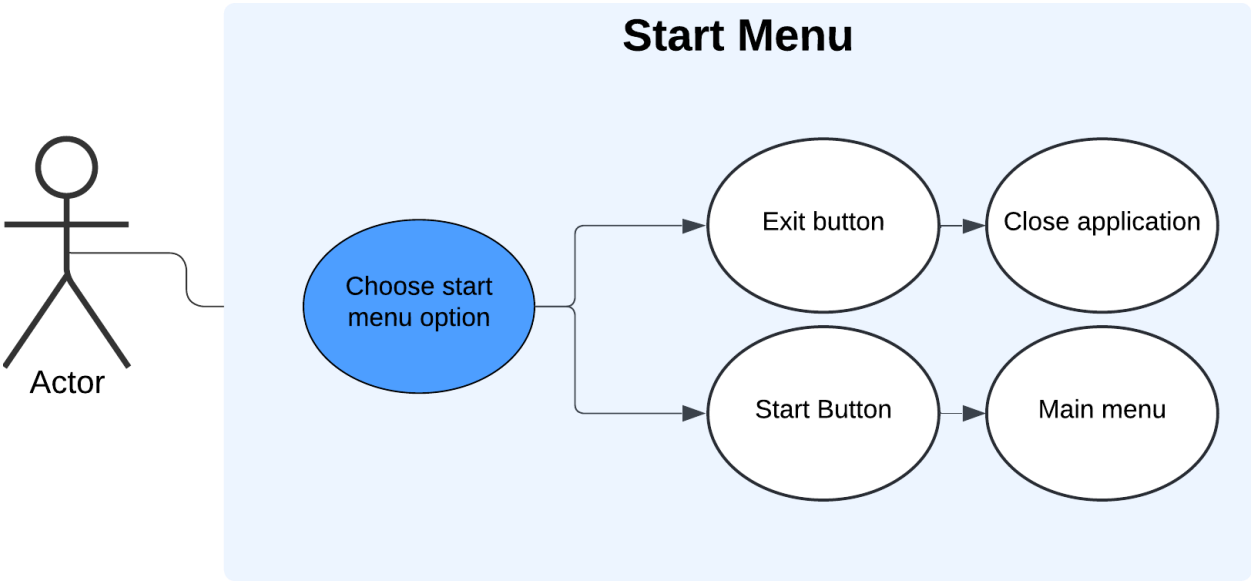
UML Use Case Diagram

The UML use case diagram below visually describes how the user will navigate into the different sections of my adaptation when first loading the game. As mentioned in the design, the start menu will consist of the buttons that will indicate to the user the expected sections of gameplay. The “start” button of the menu will lead into the adaptation’s main menu and the “exit button” will give ability to terminate the process by closing the application

Adding an “exit button” was a feature that was discussed during research and a criterion of the success criteria. Throughout my application, the user interface will emphasize the use of clearly indicated “exit” and “back” buttons to direct termination of the program and return to previous game states. My justification for this can be seen in the example of the user suddenly closing the application via the “x” button (that is built into the window management API) during chart editing process which contains a file currently in use. If the file in use was not last saved properly, the file could be at risk of corruption which may lead to potential memory and storage on the user’s computer. However, if the user were to exit via “exit” button that is built into the programs source code and clearly intended to close

the application, it will allow proper means of cleaning up the resources used for running the application. This can be through calling functions to free memory, calling destructors and automatically saving files e.g. Settings,

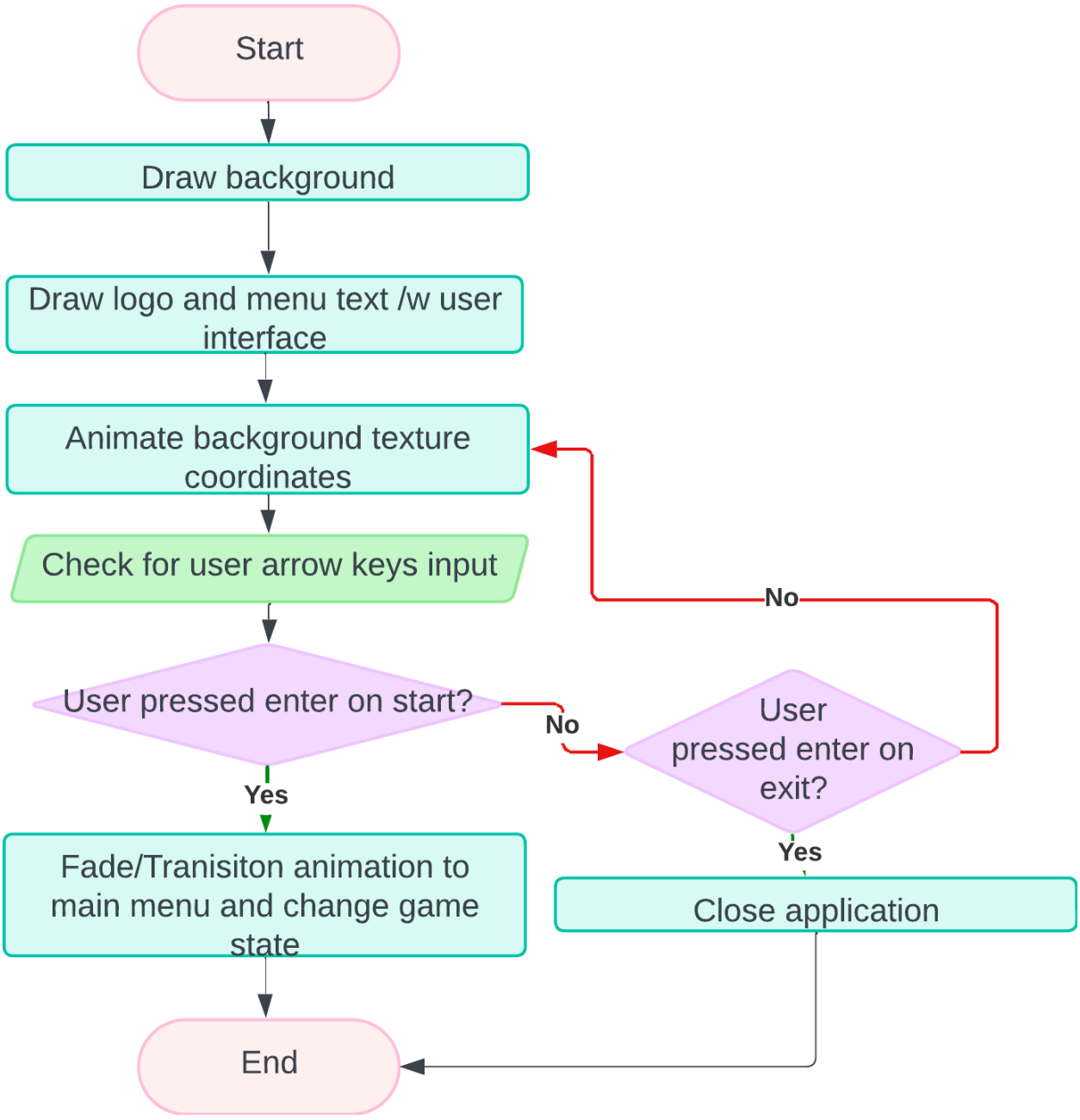
Another justification of adding an “exit button” is to further allude to modern functionality and useability that will appeal to the modern generation and increase the overall intuitiveness of my design. Increasing the intuitiveness of my design further justifies the idea of the user not requiring any foreknowledge of background information to interact with the game. This was stated in the user requirements section of my research and will be a theme that is kept throughout my adaptation.



Flowchart

The flowchart for the start menu begins with drawing the background. This is because for the positioning and layering of the user interface to be correctly organized, the bottom most object must be rendered first. Afterwards, the user interface for the main menu can be drawn on top of the background. Once this happens, the game will begin animating the background by updating the pixel coordinates of the background whilst waiting/processing user input. This process typically happens through using a .GLSL (shader) file and is implemented on the GPU (hardware accelerated). Simultaneously, the game will check for user input via the arrows keys and commence to change the game state based on the menu option selected. When this happens, the transition animation will occur. The animation consists of the entire game window quickly “fading” darker and then brightening on the new interface. In this case the interface will be the main menu screen. My

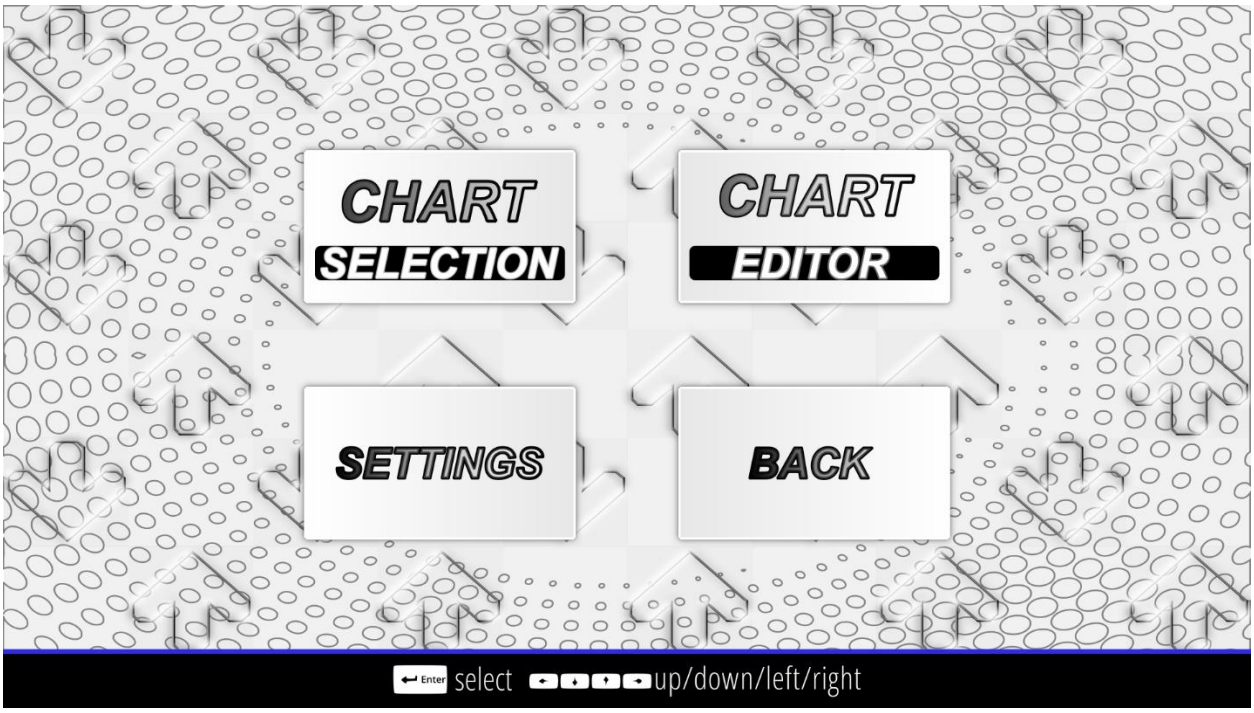
justification for this animation is to keep the design of my adaptation accessible by making it visually clear to the user that the game is transitioning into a different part of the game.



Main Menu

The start menu of my adaptation focused on implementing the desired design aspects. The main menu of my adaptation will focus on navigation and be the core system in which all

aspects of the adaptation are accessed. The menu system will allow progression into features such as gameplay, map creation and customization of gameplay settings.



User Interface

The user interface of the menu system is coordinated using left/right/up/down keys. My justification for this is to keep with pattern of the retrospective style of older arcade dance games that commonly used button input as means of navigation. Furthermore, my intention is to add micro-interaction animations such as 3D rotations around the x axis and scale hover animations to the interfaces whenever a user is selected on it. The intention behind this is to mimic the arcade menu feel that is appealing to the older generation. This is because

Although the user interface will be accessible through the arrow input keys, I intended to keep the availability using the mouse to navigate a viable option for the user. This is to keep the duality of retrospective design and useability and functionality.

As well as the user interface being accessible, as part of the older design features, I implemented a slight drop shadow around the edges of the interfaces and a linear gradient as the background color. This is justified as older VSRGs like DDR do not tend to use a singular still color but often use gradients alongside their perspective backdrops. Furthermore, the aspect of older design is added through a bold, smooth and outlined text.


CHART
SELECTION


CHART
EDITOR

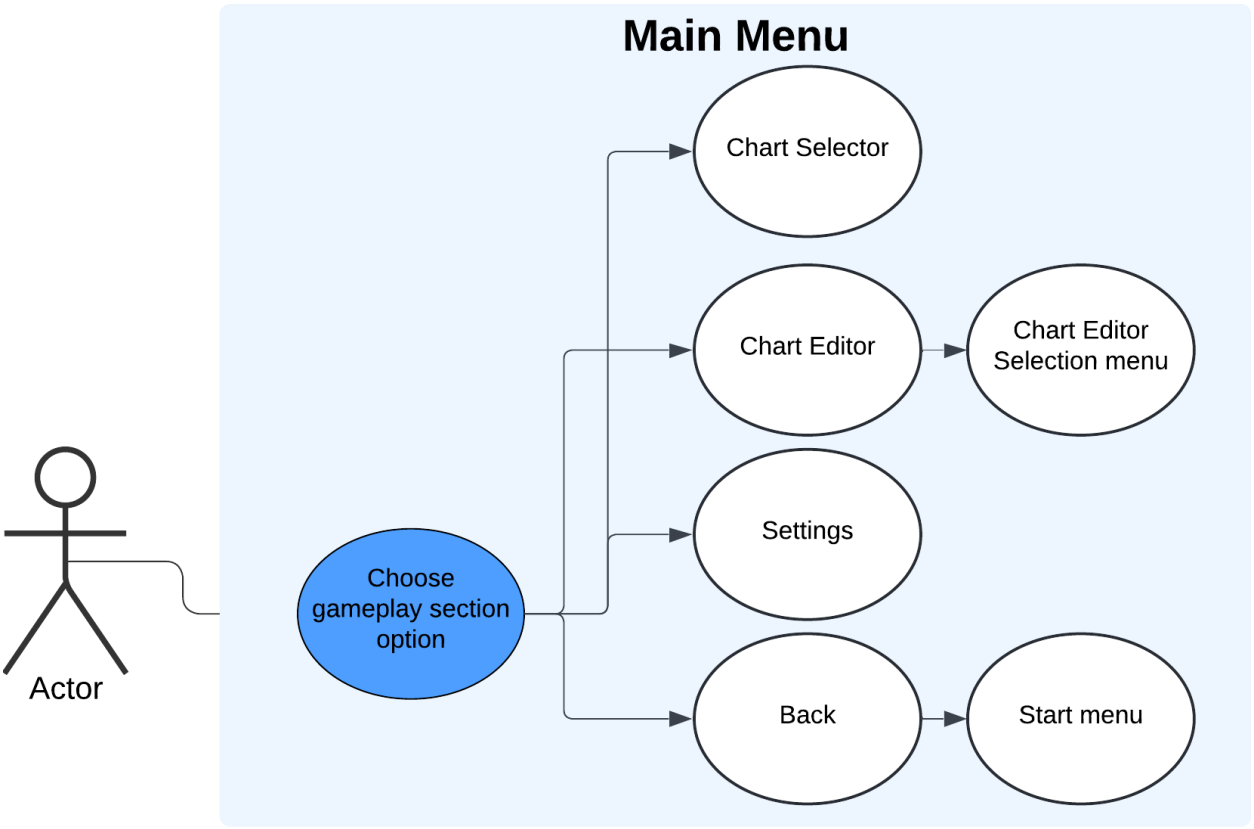

SETTINGS

Another feature that adds to functionality is the method of going back to the start menu either via a “back” button via mouse input or key input navigation. This adds to the ease of access and further enhances the intuitiveness of my adaptation.

Another aspect of useability and functionality is the user navigation assist that is tailored to the navigation of the main menu. This user navigation bar continues to indicate to the user how to maneuver around the menu, via icons indicating the keyboard keys to be pressed. This further keeps the aspects of useability and functionality that are tailored to the modern audience.

UML Use Case Diagram

The use case diagram for the main menu shows the accessibility the user has when changing between game states. This is particularly shown through the user having the ability to return to the start menu via the “back button.” As mentioned earlier in the start menu section, such “back” buttons will not only give the user more control and allow faster access to different game sections, but also allow the program to implement appropriate procedures and functions during the transition of the game state. An example of a procedure that can be implemented when a “back button” is pressed is a confirmation screen that asks the user if they would like to “go back” to the previous screen. This will give the user thinking time and generally add to the modern useability of the interface.



Flowchart

As well as the basic features of a menu user interface like input processing to allow UI navigation, the flowchart for the start menu takes into consideration the steps taken in visually animating the interface. The justification of adding these steps into the flowchart is to ensure the animations are synchronized to the user's keypress input and allow reverting to the original UI state when deselected. Furthermore, the flowchart includes the specific type of transformations such as scale and rotation, that will occur to the user interface when being animated. The justification for this inclusion is due to the animation process being the product of the mathematical process of matrix multiplication to the object's coordinates in the shader, with time as a variable. Within this process, the correct function for matrix multiplication must be used to ensure the objects coordinate are translated correctly.

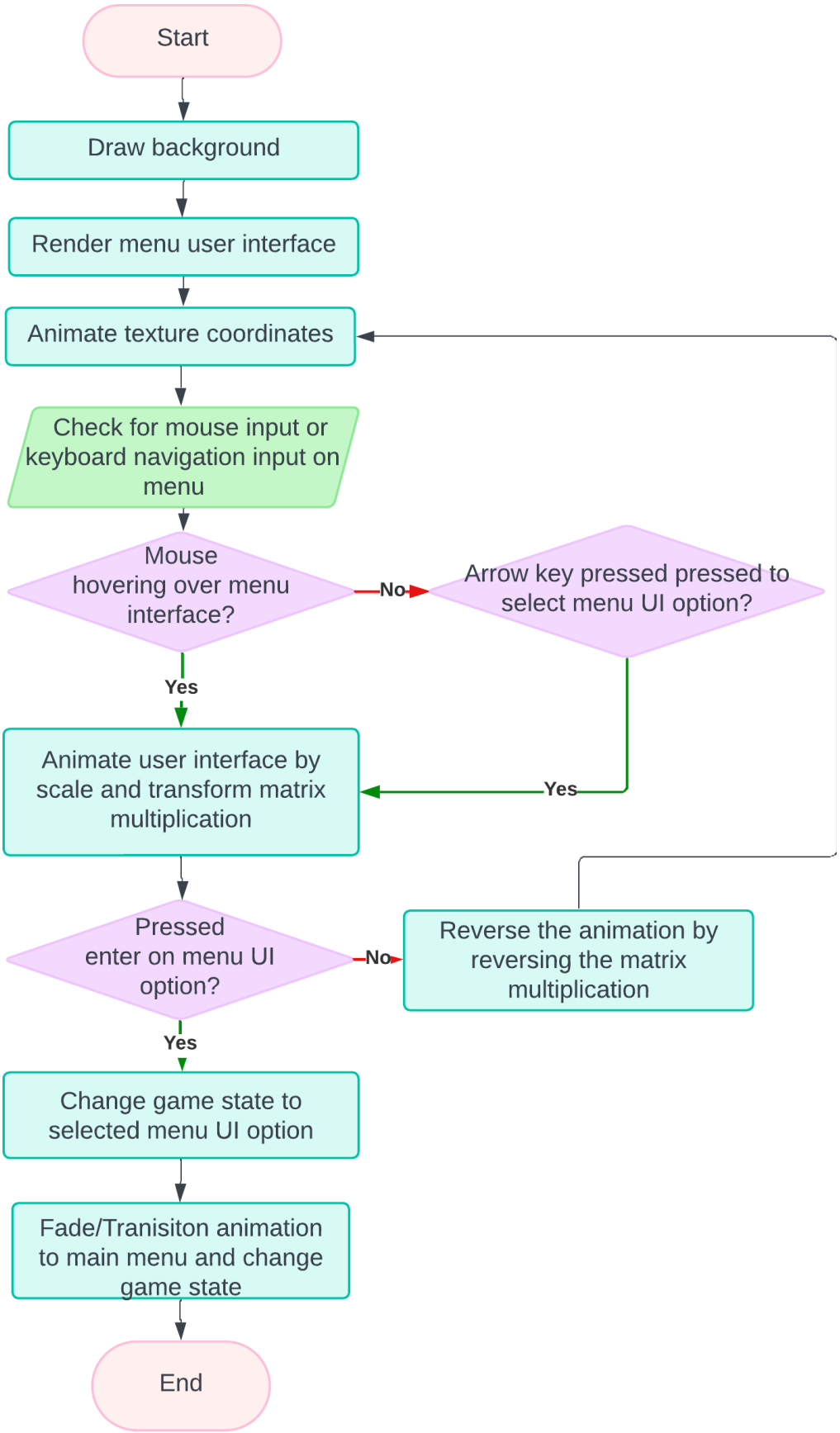

Start
Draw background
Render menu user interface
Animate texture coordinates
Check for mouse input or keyboard navigation input on menu
Mouse hovering over menu interface?
No
Arrow key pressed pressed to select menu UI option?
Yes
Animate user interface by scale and transform matrix multiplication
Yes
Pressed enter on menu UI option?
No
Reverse the animation by reversing the matrix multiplication
Yes
Change game state to selected menu UI option
Fade/Tranisiton animation to main menu and change game state
End

Chart (Map) Selection

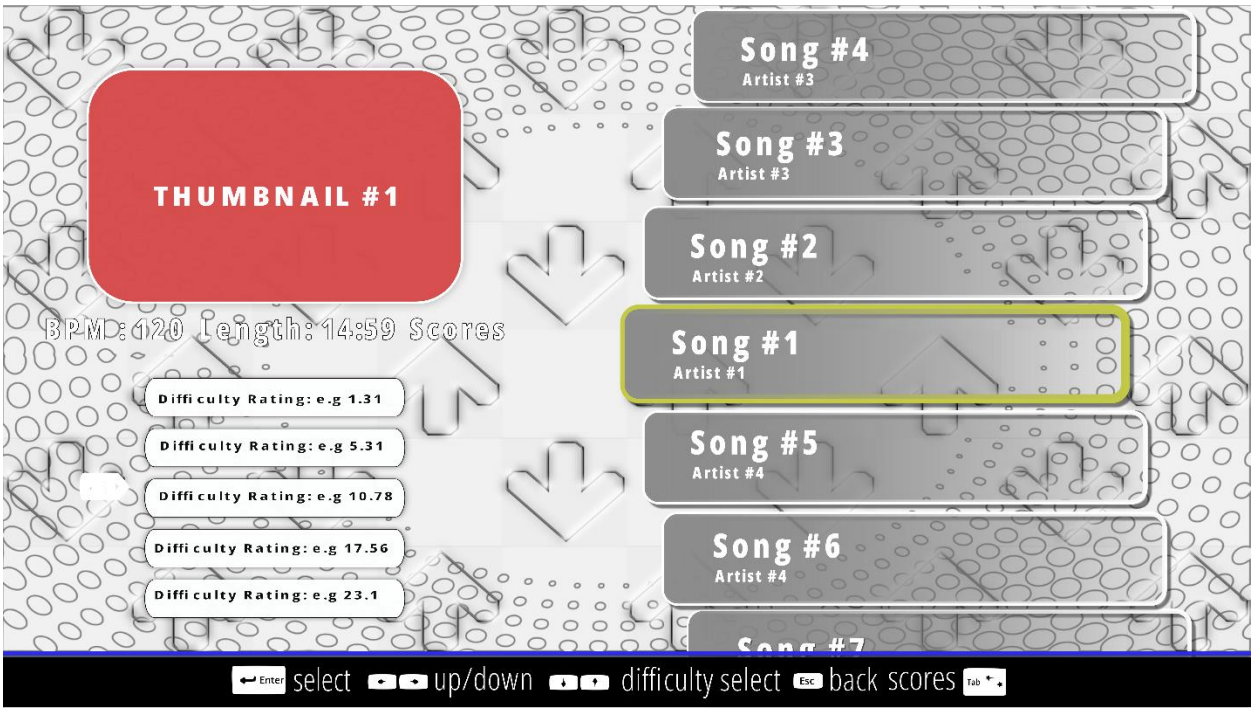
The chart selection menu of my adaptation will allow the user to select a song and commence into the main gameplay. This menu is accessed after the user selects the "chart selection" user interface. The chart selection menu will consist of a list of all the users' maps they are able to access on their device. The design remains like most VSRGs and StepMania itself. This design simplifies and heavily abstracts the actual process of loading a chart file into the game. The user does not need to see the internal process happening within the game system. The user only needs to be able to navigate the user interface and access the maps based on the information the map gives. Therefore, simplifying the process to just a few button presses. This design allows the user to focus on aspects of gameplay such as an evaluation of their skill level, whether they can complete the map, which difficulty they should select for the map and if they have the time for the length of duration of the map. All of this will benefit the user's functionality and allow them to have immersive experience.

As well as the process of loading maps being abstracted, the chart selection menu design also provides a means of ordering the charts in an organized manner. In reality, the files for maps are stored in different locations in secondary storage, but the chart selection design visualizes the charts as one continuous list. This will improve the accessibility of charts and speed up the time taken to get into the main gameplay as the user will not need to search for them in memory. This design makes the game smoother and allows fast repetition of entering gameplay, finishing gameplay and then selecting another chart to enter gameplay. The seamless transition between gameplay and chart selection will benefit the stakeholders of my younger generation as they are more tailored to spending long hours on the game and as mentioned in research. This means they can spend more of their total time practicing playing the game and take less time during navigation.

The chart selection design will also show the most crucial feature to my adaptation, The new and improved difficulty calculations. As mentioned in research, the calculation must be of a quantitative measure and give an accurate measure of the difficulty chart. The actual difficulty of the map is calculated in real-time during and after the map creation process and is displayed in both the editor and selection menu. This feature is a major part of my success criteria and therefore must be a fundamental aspect of my design. As well that, the ability to change between different map difficulties and view scores must be a feature of my design as the difficulty of maps and the scores achieved play a large part in gameplay as they are the main indicators of progression in skill level.

The final feature of my chart selection design is the displaying of a thumbnail/background for the chart. This feature will help in differentiation of charts for when charts may have the

same name. This feature along with an audio preview of the song whilst it is playing will further improve the user experience and meet the standards of my success criteria.

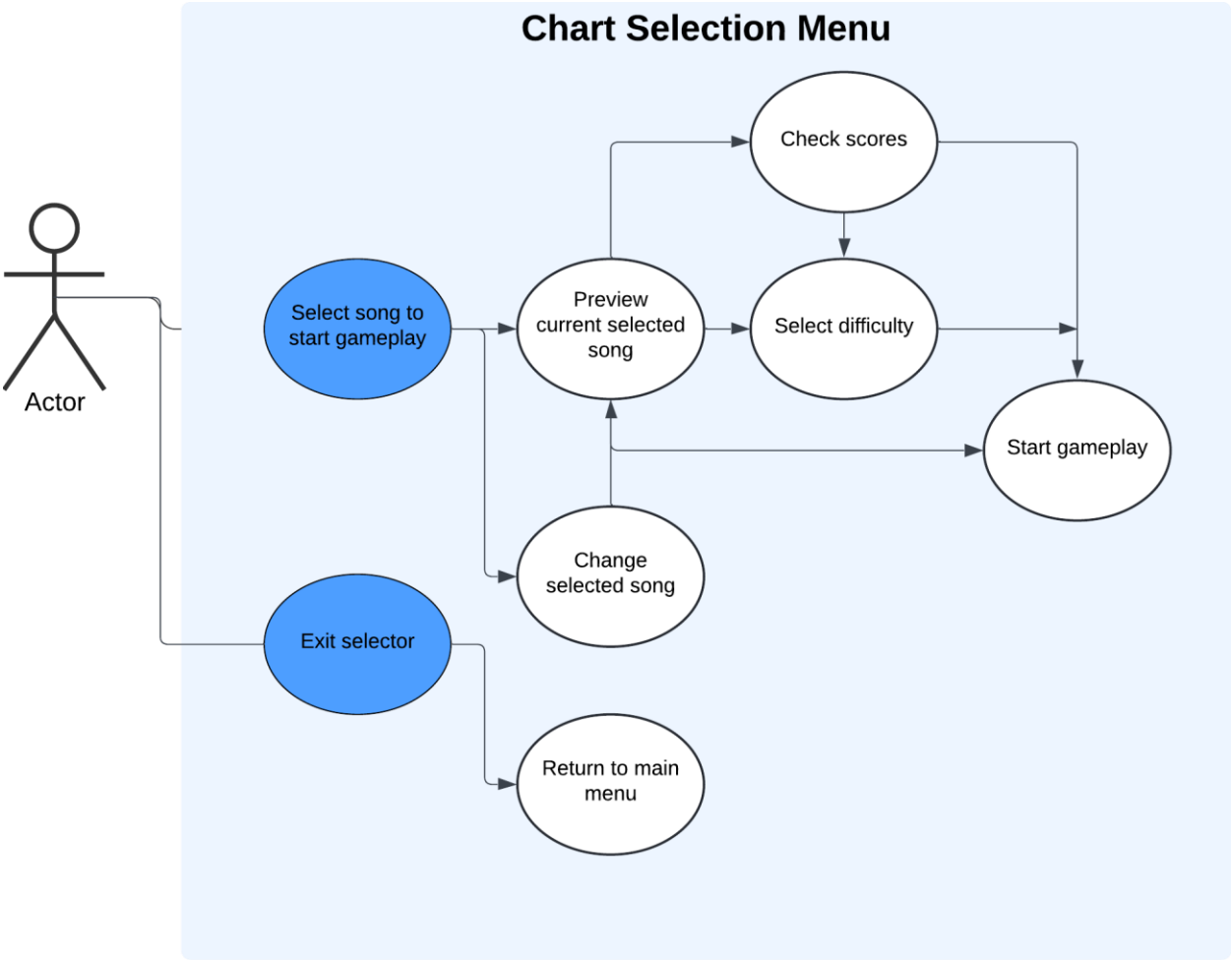


The user navigation assist bar is still present within the design to keep the aspect of usability and functionality. In this section of the game, the bar contains more input icons to tailor to the increased controls to navigate through the chart selection.

UML Use case diagram

The use case diagram for the chart selection menu displays the different and unique mechanisms, such as previewing the most recent play scores for a chart, that can be accessed within the chart selection. The diagram considers the different orders that the user may take when navigating and interacting with the mechanisms . This is shown in the diagram as I have included multiple paths from the “Preview current selected song” that stem in different directions and lead to different use cases. From the user’s perspective this means that when the user is previewing the charts, the user will have the choice of choosing the mechanics in a non-linear fashion; In this case, the user can select the difficulty of the chart via arrows keys then check the scores via the tab key or the user can first check the scores via tab then select the difficulty of the chart. This shows that the user is not restricted to using the interface in a certain order for the game to allow functionality. My justification for this feature is that it gives the user more freedom and ease of access in navigation. For example, if the user was only allowed to check their scores only when they

have selected the difficulty in that specific order then the user would need to learn that specific order that a linear fashion. This would be cumbersome for the user and potentially invoke frustration as the user may only want to check the scores of the chart and not select the difficulty but is forced to select the difficulty first regardless. This will increase the overall fluidity of the navigation system and increase the desired modern functionality of my adaptation.

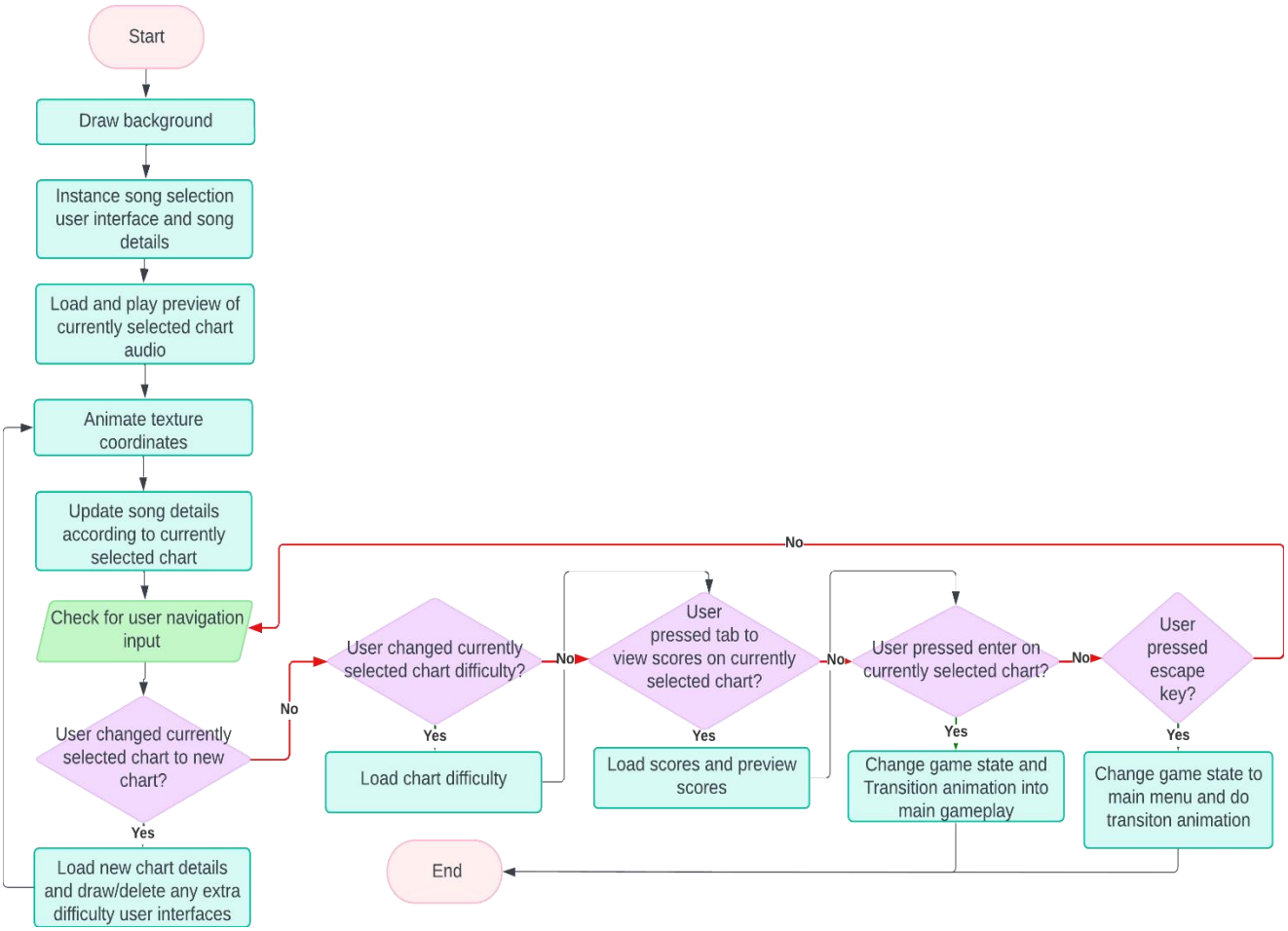


Flowchart

The flowchart for the chart selection menu contains more features than other sections of the game. Most noticeably, the algorithm to implement the allowance of non-linear access to chart selector mechanisms as mentioned in the previous paragraph. This algorithm will consist of a chain of non-nested if-statements. The non-nested nature of these if-statements means that there are no ‘else’ statements at the end of each statement. This means that the game does not need first check for the first condition to move onto the next

condition; In this case, the system does not need to check if the statement of “the user has currently selected a new chart” is false before moving onto the checking if the user has changed currently selected chart. Instead, the system checks if both statements are true or false contemporaneously, thus allowing the execution of menu mechanism based on input in any order (Which ever statement’s Boolean value is true first).

Another notable feature is the use of instancing when drawing song selection user interface. The use of instancing means that each of the GUI elements that encase the song information e.g. song name and song name, can be drawn in a single draw call. This is opposed to having multiple draw calls for each GUI element containing the song information. My justification for this is due to the GUI elements being the same texture thus not requiring a different texture to be bound to the when rendering. Only using one draw call will mean that the CPU/GPU resources will be used more efficiently and lead to an increase in performance factors like frame rate. An increase in performance will lead to smoother gameplay and increase the overall user experience. Factors like frame rate are especially important in VSRGs, as mentioned in the visualization section of research, it will give a more accurate representation of the timing interval for the user to register input in thus leading to a more engaging gameplay experience.

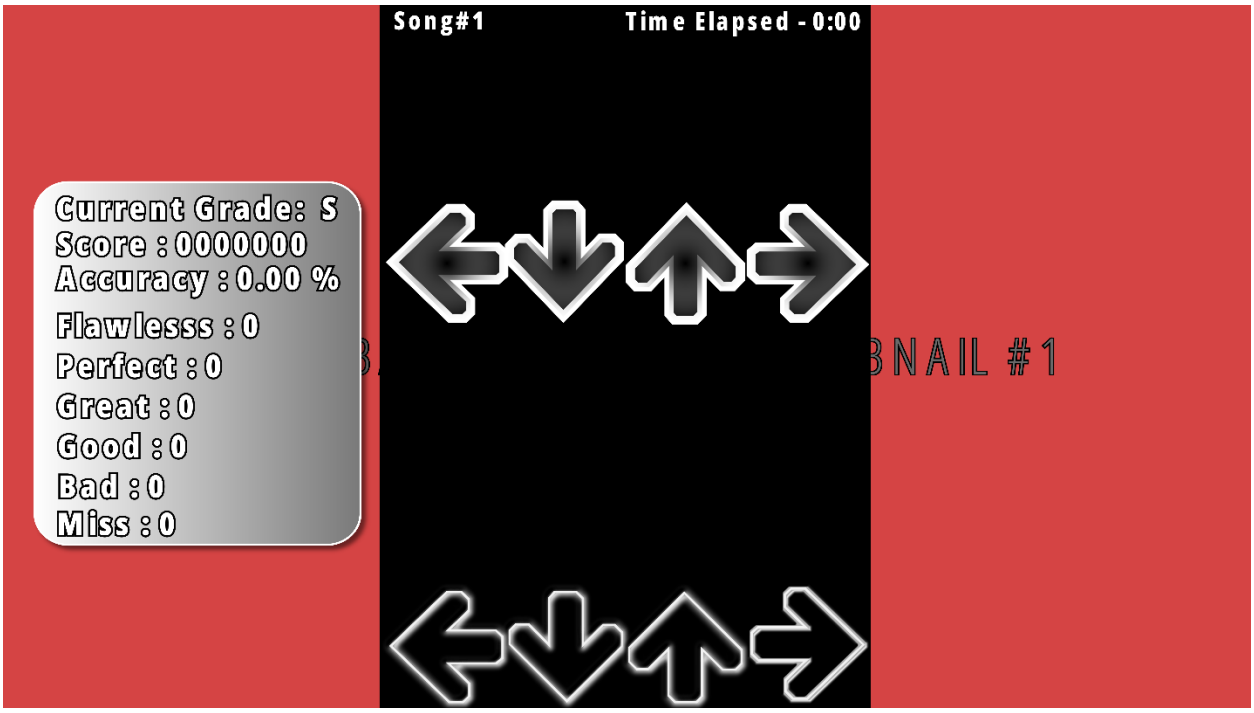


Main Gameplay

The main gameplay will consist of the standard VSRGs gameplay. The user will be able to accordingly respond to notes that are scrolling vertically and yield input based on the visual information. As well as the visual information of how close the notes are to the receptors, the user will also have audio information. There must be audio playing in the background of the main gameplay which will aid in timing the notes and set the basis of the 'rhythm' aspect of the adaptation.

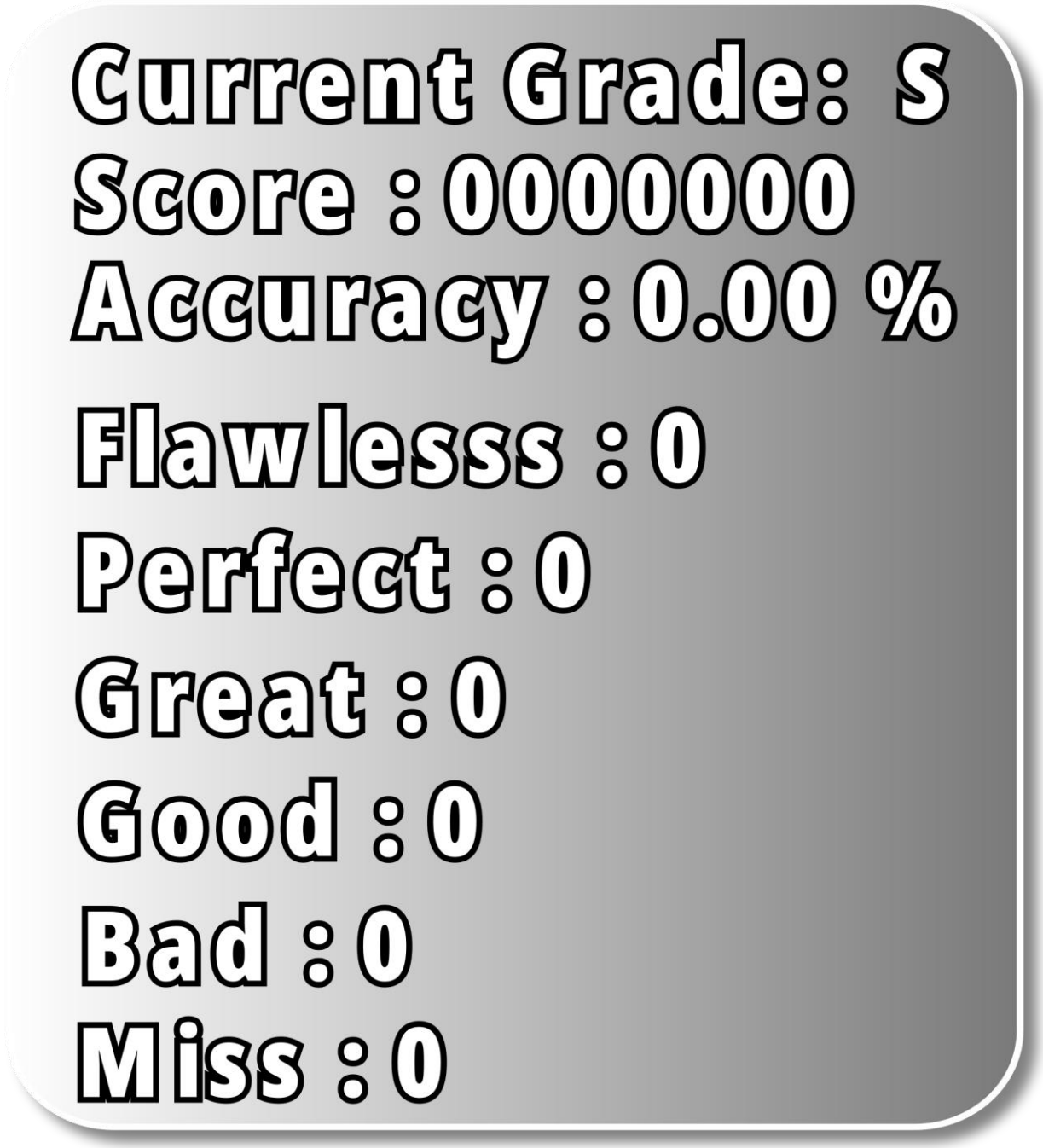
In the background of my adaptation's gameplay design, there shall also be a timing class that will process the timings in real time and give an accurate response on timing judgement. The timing class will form the basis for setting up the user's scoring, thus in turn forming the basis of the grade and accuracy system.

As well as the gameplay and timing, in the actual background of the columns in which the notes scroll, there will be a background image that the user can set in map creation. This



Gameplay Statistics

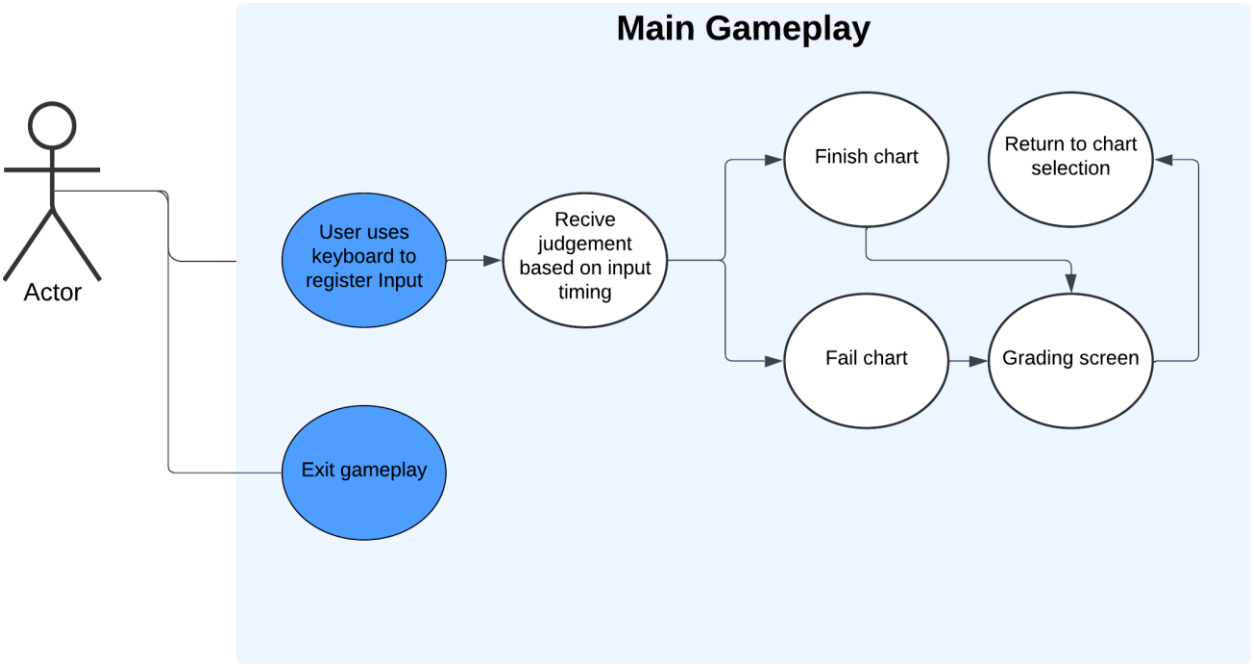
The gameplay statistics will be what is used to indicate the user's performance and the general skill level of the user on the map. The statistics will show the quantity of hit judgements they have received and the average hit accuracy throughout the map. A higher hit accuracy will contribute to a higher accuracy which will increase the score. The displaying of real time gameplay statistics will be beneficial to the user during gameplay. This can be during areas of which there are not many notes in which the user can take a glance. It will also benefit them as it will allow them to monitor their performance progress between their current gameplay and past gameplay score if they are replaying the map. The ability to monitor and track progress statistics in real time adds to the functionality and useability of the adaptation that is appealing to the modern audience of my stakeholders.



UML Use Case Diagram

The use case diagram used for my gameplay clearly considers the different pathways that can form during the user in main gameplay. My justification for this is because it is not known, upon the entering of the main gameplay, what the result of entering the main gameplay will be. For example, the user could enter the main gameplay but fail the chart

being played; Initially the user will have the choice to begin registering input upon the start of audio. From then, if the user decides to yield input, the user will continually receive judgment based on the timing of their input. However, if the user's input is greatly off time during this process, the timing judgement will be registered as a "miss". As this happens, the game health bar will decrease. If the user's health bar diminishes, then main gameplay will terminate and the grade for the user will be a "failure". However, another path the user could take is initially entering main gameplay but then exiting before the end of the chart due reasons such not being satisfied with their current performance. The user could also take the pathway of entering main gameplay and neither exiting nor failing but completing the chart all the way through. In this case the user will be awarded a different grade. Another pathway the user could take is entering main gameplay but immediately exiting the main gameplay. Some reasons for this could be mistakenly selecting the wrong chart or changing their mind on playing the chart. Finally, the user has choice to enter main gameplay, not yield any input, miss the notes and dimmish the life bar and immediately fail the chart. The consideration of different pathways increases the versatility of the gameplay and contributes to the modern useability and functionality that will benefit stakeholders as discussed in research



Flowchart

The two most prominent features of the main gameplay are the rendering of the notes based on the timing in the chart file and calculating the judgement of the input timing.

In this diagram, the process of the how the notes will be rendered has been heavily abstracted. However, it maintains the core aspects of how the note rendering will work; The time the note will be rendered will have an offset of the time by about the time taken for the note to scroll from the top of the screen to the receptors. This is dependent on scroll speed as a faster scroll speed will mean the offset time for notes to be rendered decreases due to the time taken to reach the receptor decreasing.

Note Judgement

The second prominent feature that is subsequent of the note rendering is the note judgment system by comparison of the user's note input timing. This will be done through real time reading of the chart file and the use of switch/case chains to compare the input timing in milliseconds. Below is the table for the range of input timings and their indicated judgement:

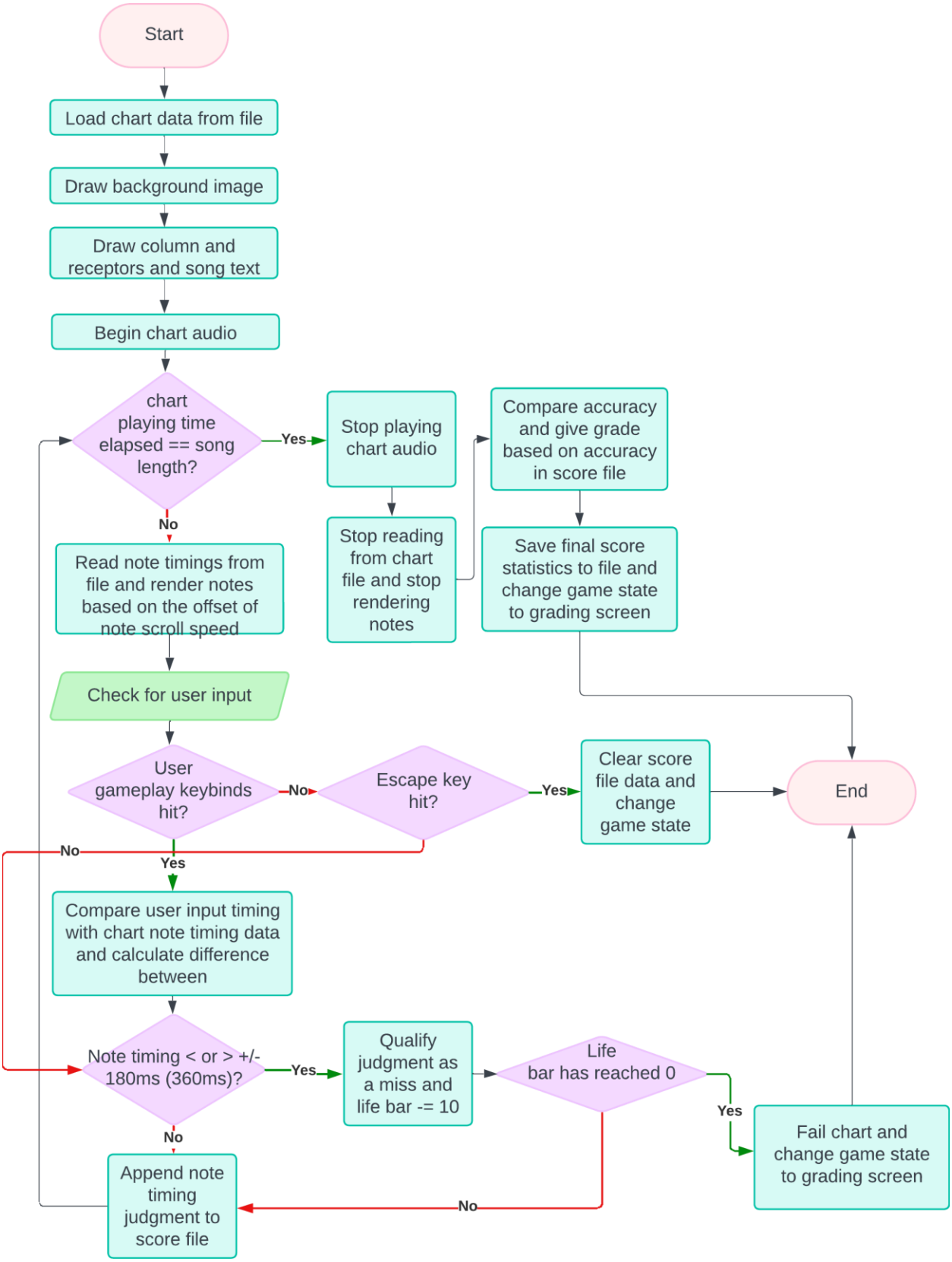
Judgement	**Input timing**
"Flawless"	Within +/- 18.9ms of real timing
"Perfect"	Within +/- 37.8ms of real timing
"Great"	Within +/- 75.6ms of real timing
"Good"	Within +/- 113.4ms of real timing
"Bad"	+/-180ms of real timing
"Miss"	>+180ms or <-180ms of real timing

The judgement system will work by comparing the difference of input time in milliseconds with the target time value that is stored within the file of the chart being played. For example, a note that is roughly at the 4 second mark for the first "beat" of the song is stored in a chart file with a time value of 4000 milliseconds (ms). In my system, a timer will begin from the start of the song audio and will be used to measure the time of the user's input. As mentioned, the note for this note timing will be visually rendered by an offset of based on the user's scroll speed. As the note scrolls, time will pass until the note is on par with its receptor. At this point the user will receive the visual queue to give keyboard input and yield input time. This time is what is compared with the time value that is stored within the chart file. In this case, let's say the user presses the keyboard and the input timing is recorded with a time of 3992ms (from the beginning of the song), the user's input timing will be -8.00ms. This input timing is then compared to the judgement range and the judgement for the input timing is given. In this case, an input time of 8ms is well in range of the "Flawless" timing and the user will be given this judgment in the score file.

Other prominent features in the flowchart include constantly checking for the escape key being inputted alongside the user's key binds and checking if the value of the health bar is

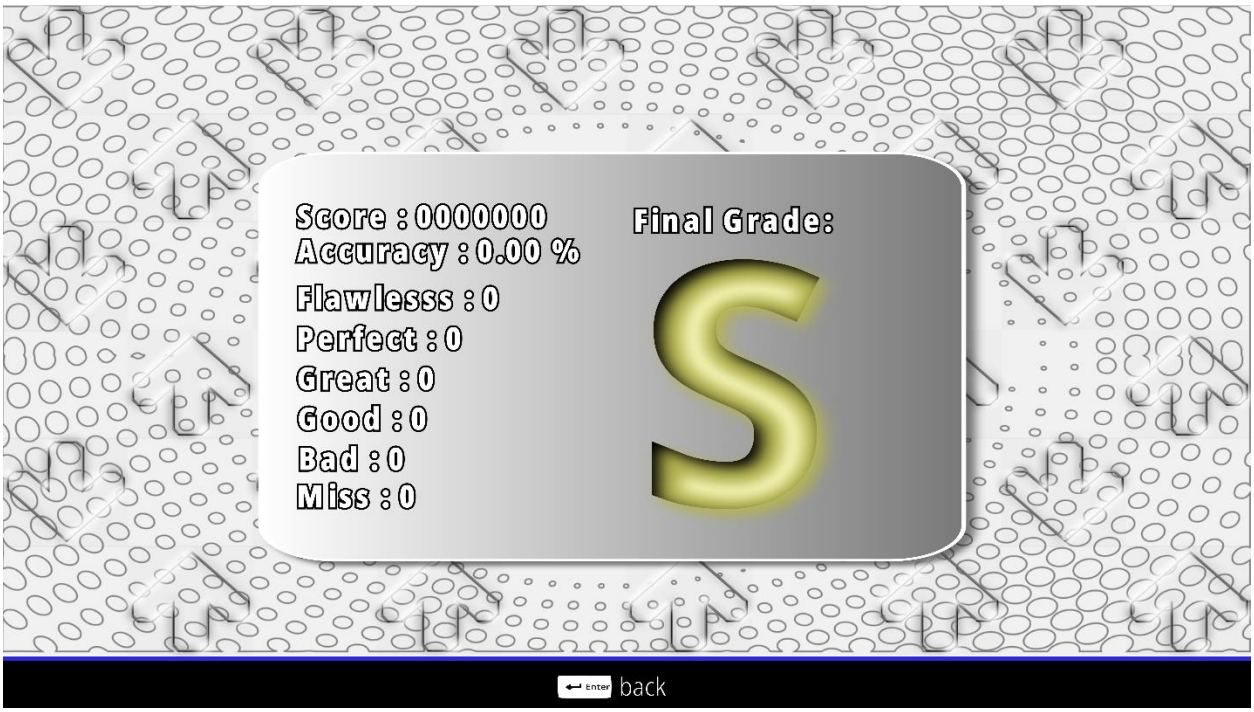
equal to 0. My justification for this is that the health bar and the escape key are the two features that can immediately alter the course of gameplay. This is because the escape key is responsible for immediately terminating the gameplay upon pressing and the health bar has the ability stop gameplay and proceed to the grade screen, once its value reaches 0. This means intended course of gameplay can be altered

The final prominent feature of the flowchart is real time reading and writing to a score file when the user receives a judgement based on their input time. This feature links back to my game statistics design where I stated the user can monitor their improvement over time, based on their accuracy and “grade” being achieved . My justification for this is that if the scores file were not stored in files within user’s secondary storage, the scores would instead be stored in the user’s primary storage (RAM). This is problematic as not only will the usage of main memory increase on every iteration but also due to the volatile nature of primary storage, the user will not be able to monitor their performance over a longer period ,e.g. the next day, without the user losing all their past score data.


Start
Load chart data from file
Draw background image
Draw column and receptors and song text
Begin chart audio
chart playing time elapsed == song length?
Yes
No
Stop playing chart audio
Stop reading from chart file and stop rendering notes
Compare accuracy and give grade based on accuracy in score file
Save final score statistics to file and change game state to grading screen
Read note timings from file and render notes based on the offset of note scroll speed
Check for user input
User gameplay keybinds hit?
No
Escape key hit?
Yes
Clear score file data and change game state
End
No
Yes
Compare user input timing with chart note timing data and calculate difference between
Note timing < or > +/- 180ms (360ms)?
Yes
Qualify judgment as a miss and life bar -= 10
Life bar has reached 0
Yes
Fail chart and change game state to grading screen
No
Append note timing judgment to score file
No

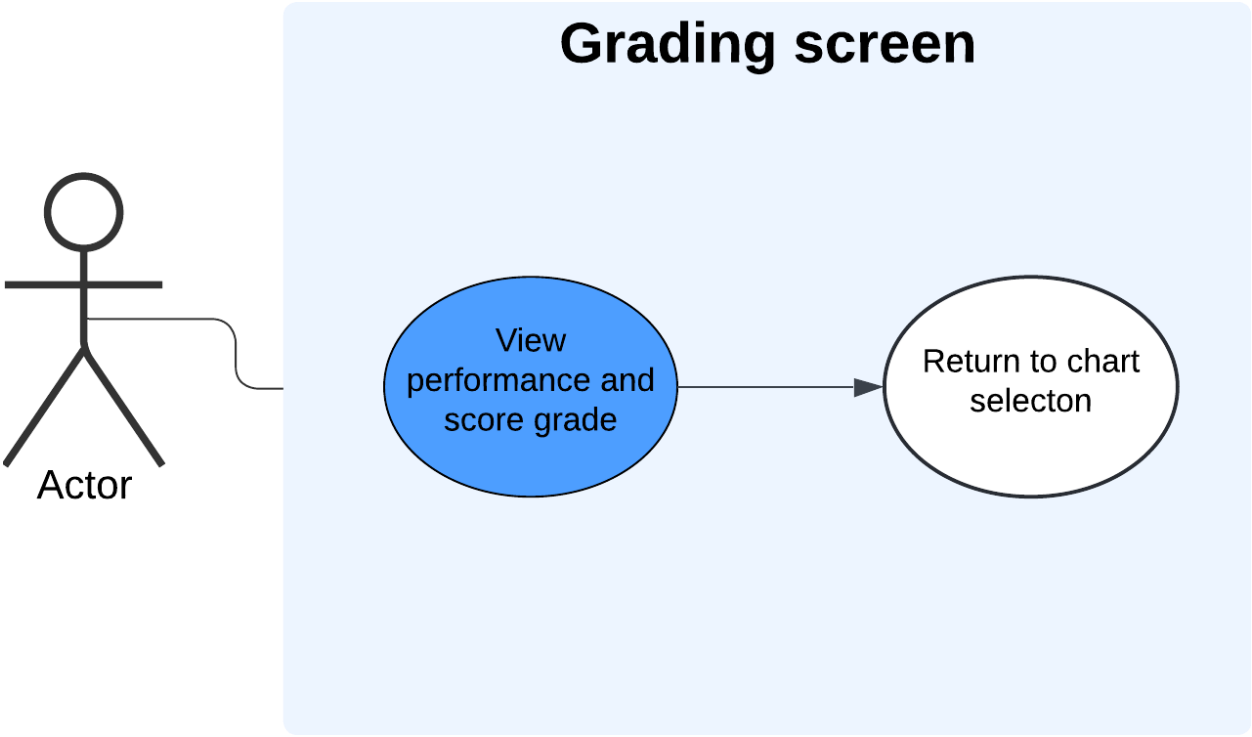
Grade Screen

Once the user has finished the map, the user will be “graded” based on their performance. The user’s grade will be accompanied by the final iteration of the gameplay statistics. This design forms modern functionality and useability of my adaptation. This is because scoring systems can be used to keep track of how the users’ scores change over time and give an indication of improvement in gameplay skill. Being able to monitor and track improvement can give the user a sense of motivation. It can also give the user an indication of what needs improvement. As mentioned earlier, maps can have different sequences of note patterns. If the user’s score is consistently below expectation for maps that are tailored to a specific note pattern, then this design can help indicate which maps the user should focus on in gameplay.



UML Use Case Diagram

The use case diagram for the grading screen shows that the user can only do a limited number of actions when on the grade screen. In this case the user is only able to preview their score and exit the grade screen to chart selection menu.



Flowchart

The flowchart for the grading screen shows the need for real time score file processing when previewing a score file. The process will consist of loading the statistics and rendering the correct text and grade. This process will happen in the background and will be abstracted from the users themselves.

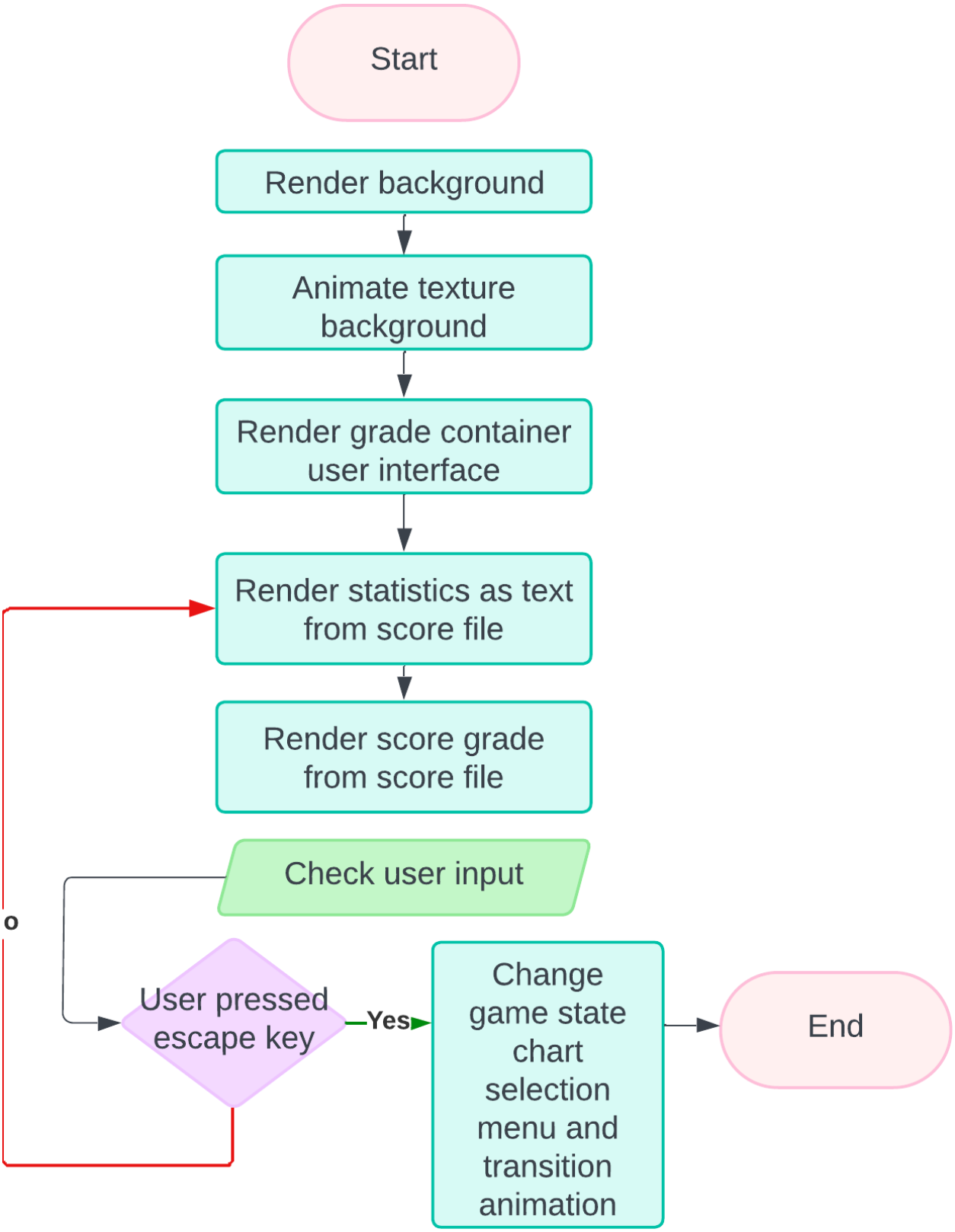
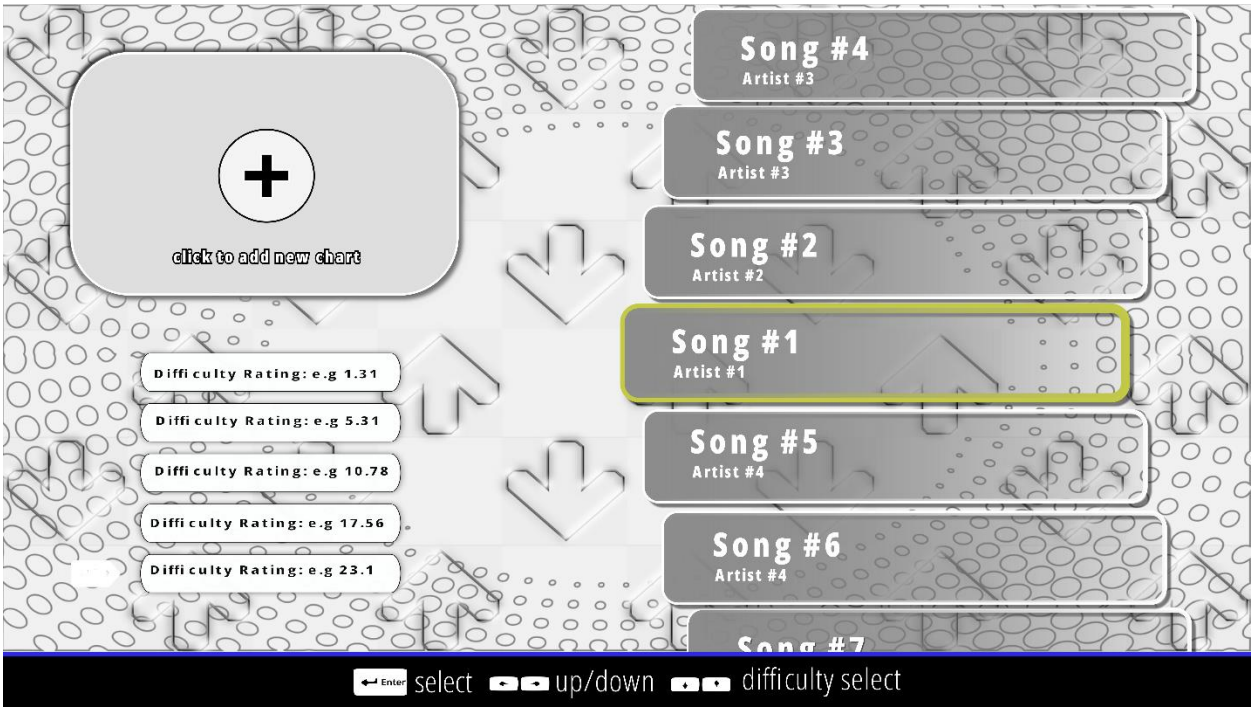
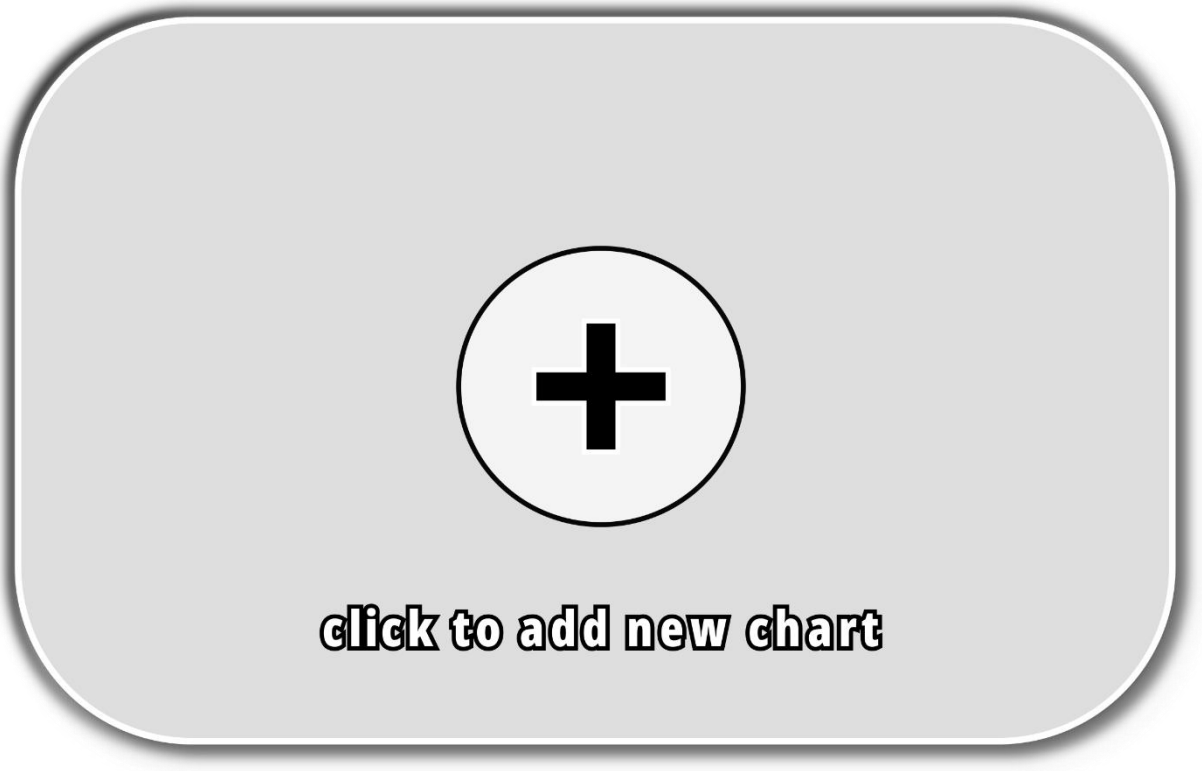

Start
Render background
Animate texture background
Render grade container user interface
Render statistics as text from score file
Render score grade from score file
Check user input
o
User pressed escape key
Yes
Change game state chart selection menu and transition animation
End

Chart Editor Selection

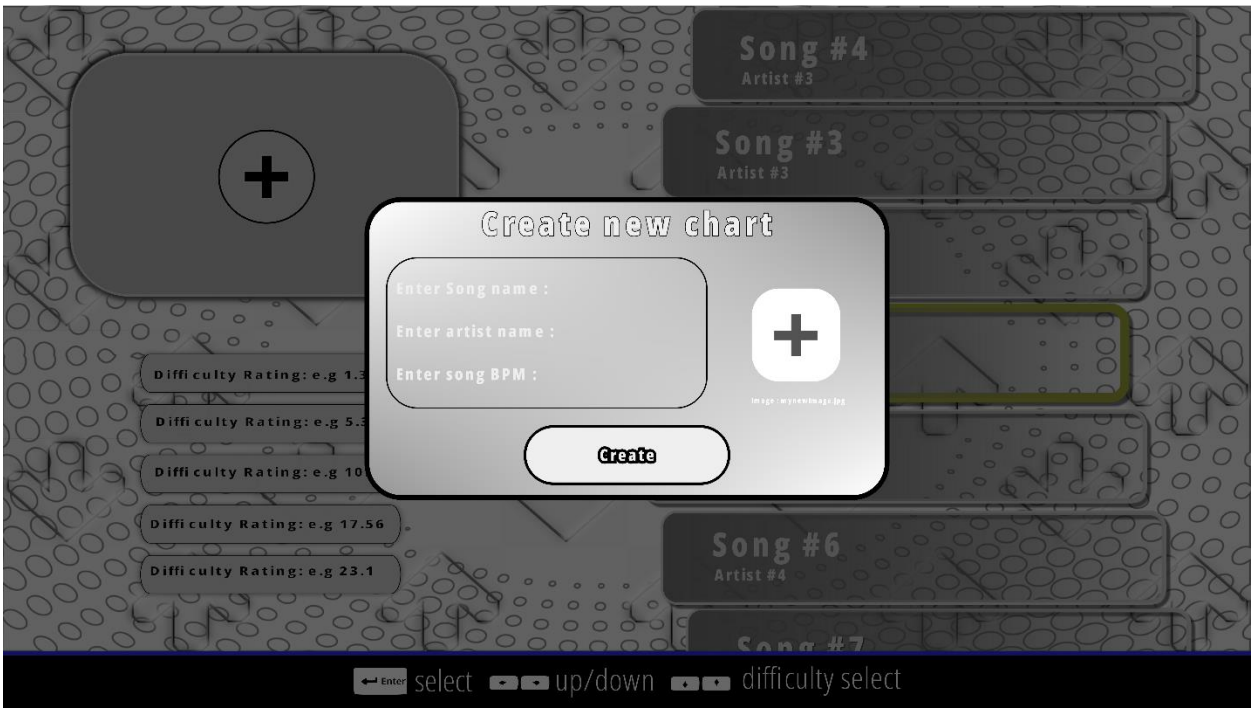
The chart editor is the second choice within the main menu and will mostly resemble aspects of the map selection menu. However, the difference between the chart selection editor and the map selection is the ability to upload audio files to initiate the creation of a new map. As mentioned in research earlier, this is an essential aspect of the adaptation's functionality as it is the gateway into which map the difficulties calculation system is exercised.



The user will be able to initiate the process of creating a new map via the "create new chart" button via mouse input. The reason for limiting it to mouse input only is to limit accidental keyboard mis-input leading to the creation of unwanted new maps.

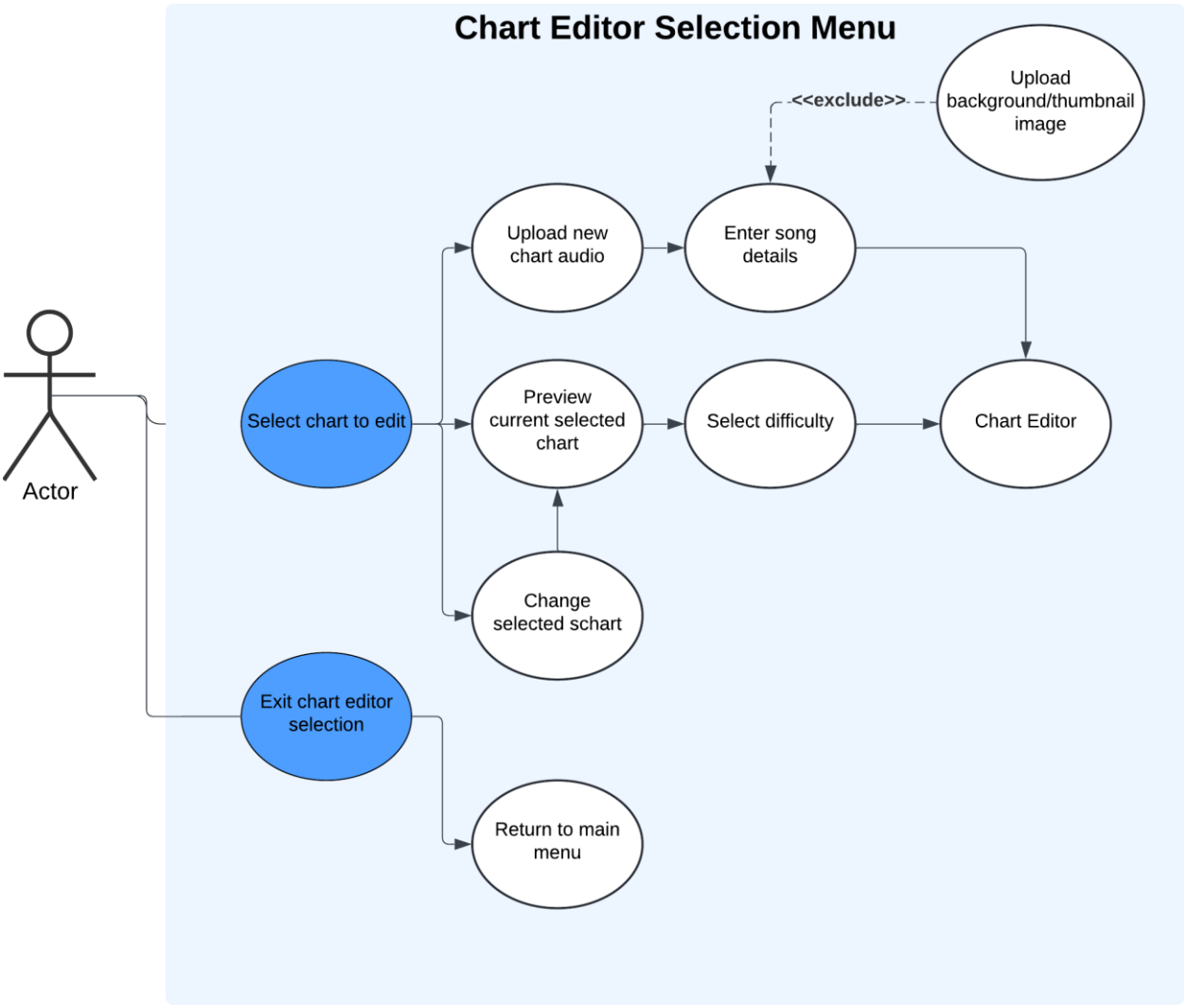


Once the user has clicked on the chart button, the user will be able to enter the song descriptions and choose the accompanying background and thumbnail images. This will form the basis for previewing the song in the map selection editor.



UML Use Case Diagram

The use case diagram for the chart editor selection menu remains most like the chart selection screen however the difference is that the section for previewing charts and their thumbnails, has been completely replaced with the ability to modify or upload new charts for songs. The chart editor system will still retain useability of being able to choose an option from the chart editor in a non-linear fashion.



Flowchart

As described in the chart selection menu, the menu system flowchart consists of more features than other sections adaptation. Despite this, the mechanisms for the chart editor selection remain the same. The difference in the chart editor selection is the consideration of the ability to upload audio files to initiate the creation of new charts or select a chart to edit. This will involve real-time file handling and the processing of new textures. This is

because the user will need to input an audio file with a chart name and upload a new image to be loaded as a texture.

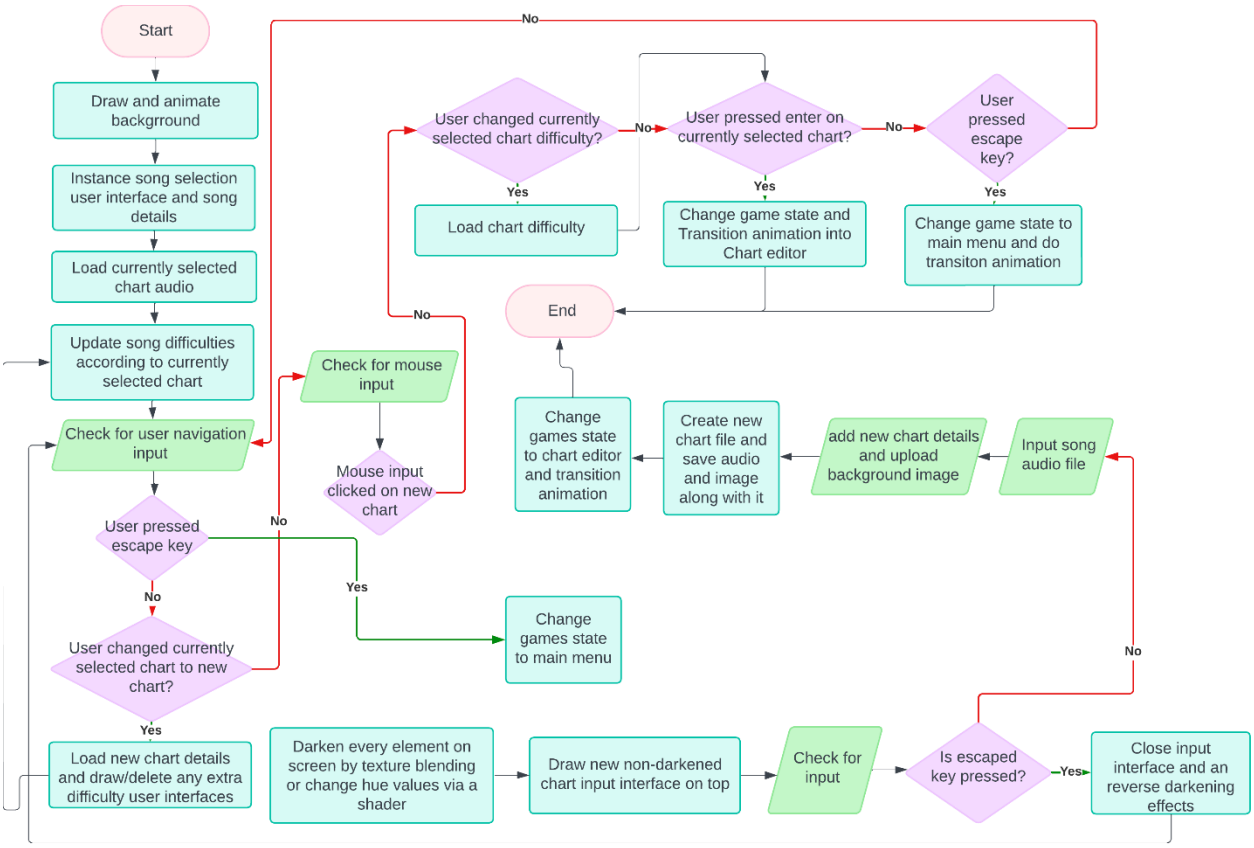


Chart Editor

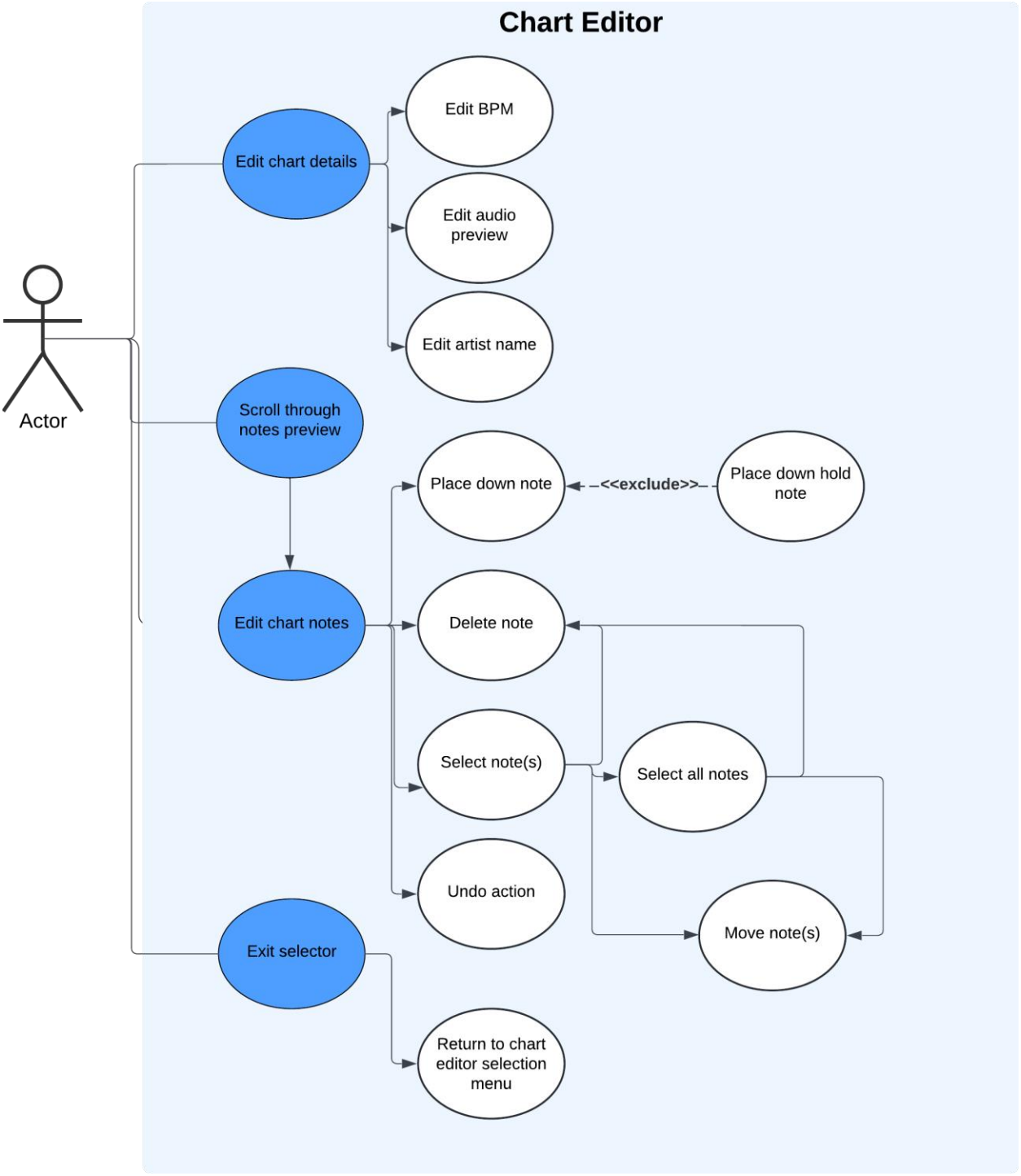
The chart editor is one of the most crucial parts of gameplay as it will be where the main aspect calculation of difficulty will take place. The chart editor will consist of a range of statistics that will be updated in real time as the user is editing a map. The statistics will include the different factors that contribute to difficulty calculation and the terminology for the different note patterns within the map. As well as this, the chart editor will provide the abstracted and visualized process of placing down notes. As mentioned earlier in research, the design of this abstracts all the background process of reading and writing to a chart file down to a few mouse inputs. This further adds to functionality the adaptation.

The process of determining the note statistics such as patterns and counts will be abstracted from the user but will be happening in the background in real time via algorithms to determine them.

As well as displaying the chart statistics, the user will be able to change the beat snapping to allow for more of a precise representation of flow of the music on the notes. The user will also be able to scroll through the length of the map via the scroll bar and preview the notes that are within that section of the chart.

UML Use Case Diagram

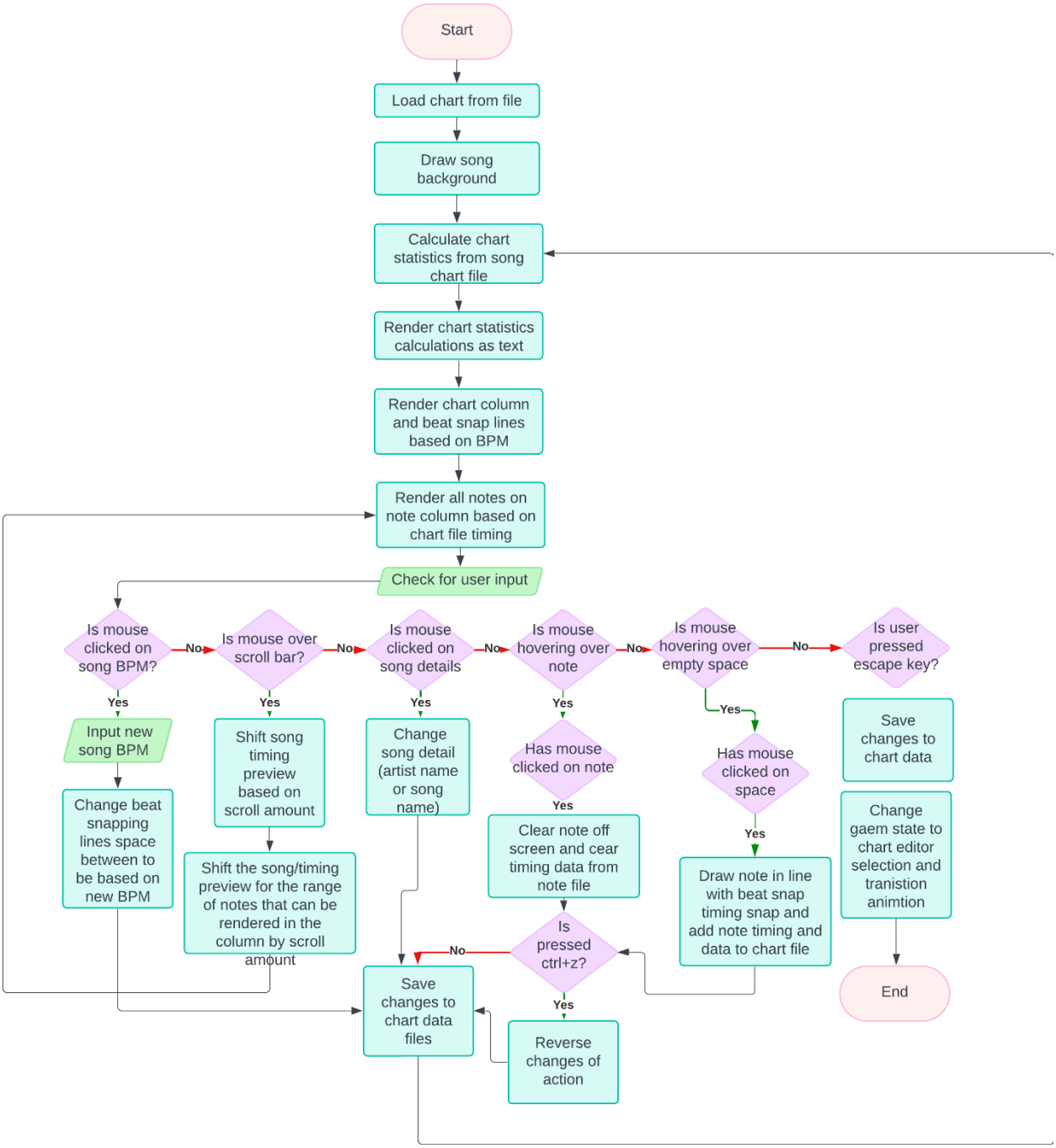
The use case diagram for chart editor is like the chart selection menu in that the user can freely decide what order they wish to operate their actions. In this case the diagram shows the different cases that can happen whilst the user is editing a chart. My justification for this is much like the main gameplay in that it is not known what the user's actions the user will be once is in the chart editor and therefore there is a need to consider the different pathways that can happen. This will be prevalent around real time chart editing where reversibility and flexibility are integrated in the usage. This can be through the user "undoing" an unwanted action or the user making a few edits to a chart without taking up too much time.



Flowchart

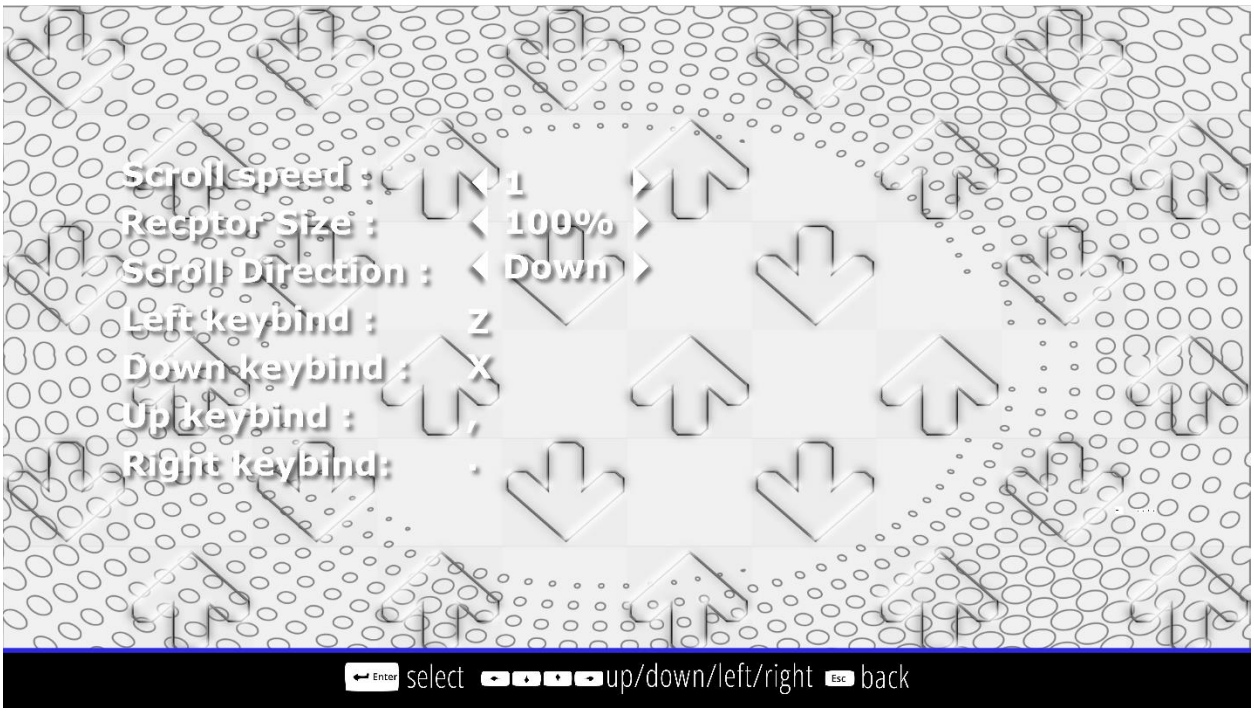
The flowchart for the chart editor follows the principles of main gameplay in that there are many different possibilities that can be taken and thus a need for contemporaneous checking of these possibilities.

Another key feature of the chart editor is the reversibility of using “control + z” key bind. This feature is included to allow efficient change of unwanted actions during chart editing. However, the use of this feature will only apply to placing notes down in the chart column. My justification for adding this feature is to increase the ease of access by not making it a requirement that the user must manually delete each note if they have mistakenly placed it down.



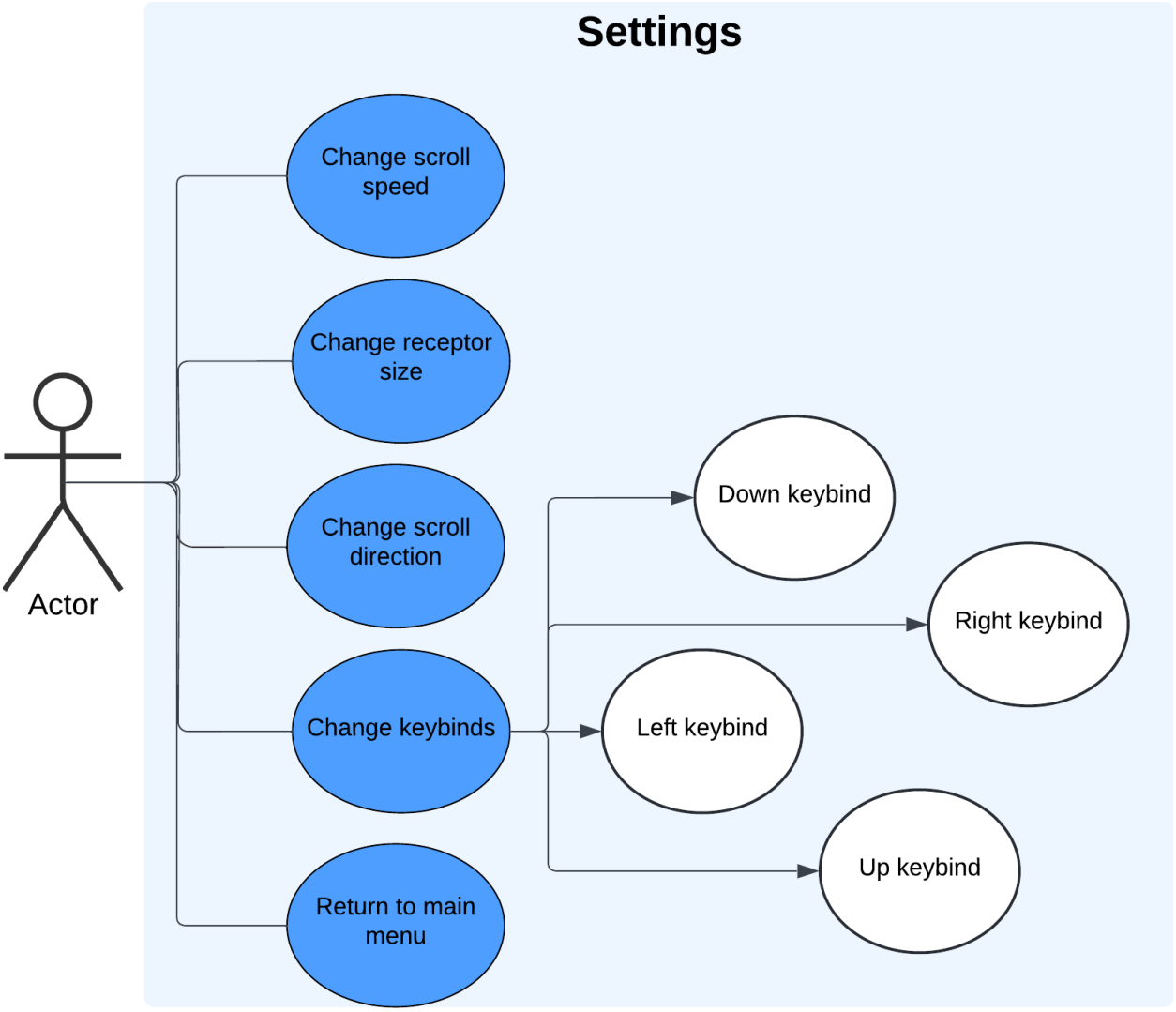
Settings

The final section of the game that use will be able to access is the settings menu. Including settings will add to the useability and functionality of the adaptation as it will allow customizability of gameplay. This will be useful for people who have a preference in the way they engage in gameplay. This is because there are a variety of different preferences that a user may use to achieve the best possible gameplay, with each preference being relative to the user. This will appeal to the modern audience who are more prone to longer hours of gameplay as it will add extensibility for changing styles as they progress. Furthermore, older VSRGs lacked this feature of customizability, therefore adding a settings section will better suit my adaptation for all age ranges of my stakeholders.



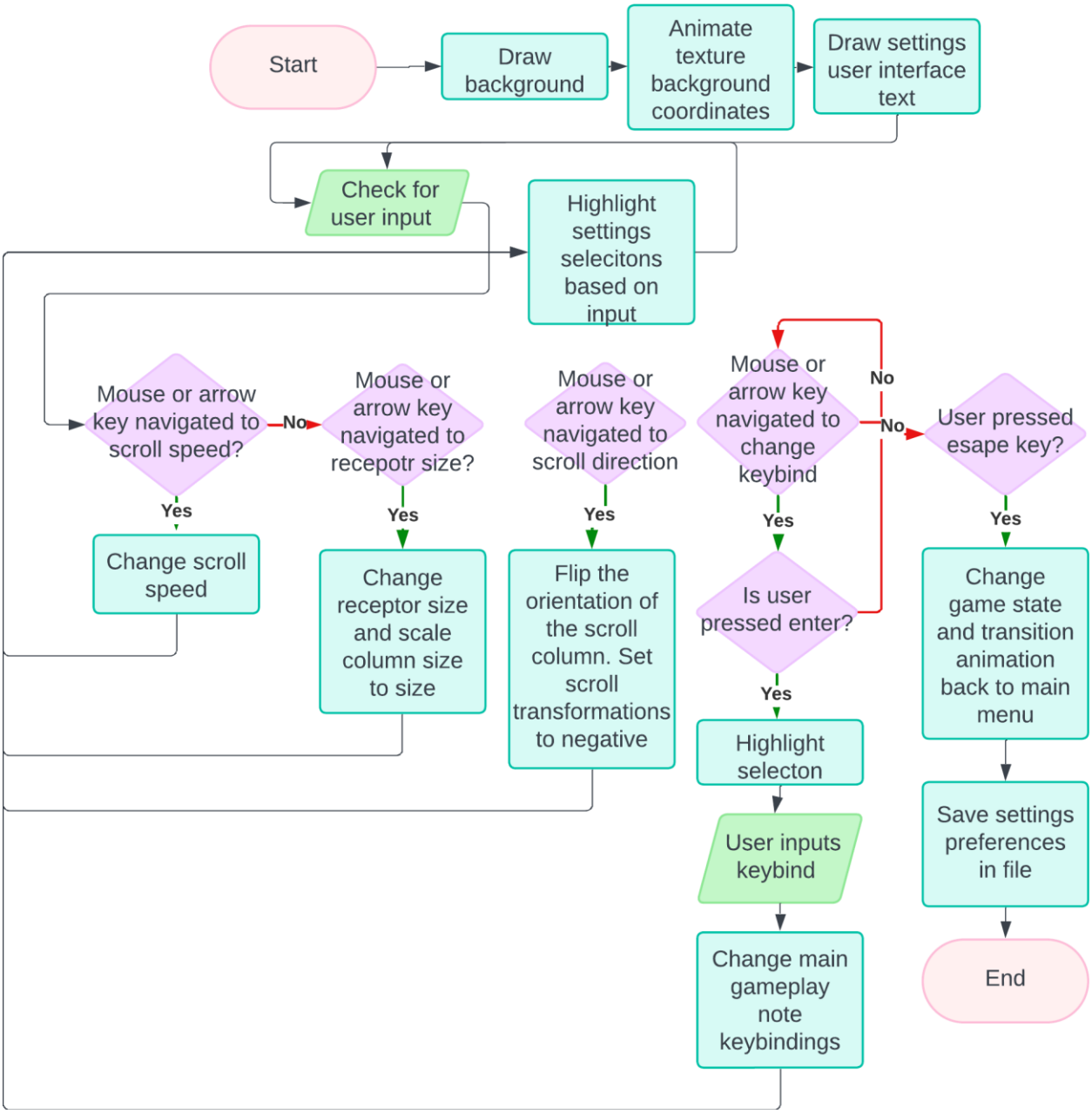
UML Use Case Diagram

The use case diagram shows different cases that the user can make when changing their settings. This continues the use of non-linear option choosing.



Flowchart

The most prominent feature of the flowchart for settings is the feature of setting key binds. This is because I will need a letter representation of each key being pressed on the keyboard. Therefore, I will need to consider the limitations of how many characters are in the font of the text being used.



Further Design Details

From the analysis up to the design, I have been taking the traditional approach to the software development life cycle. This is because I first understood the requirements by gathering information from stakeholders, then I used that data to form the scope and objectives of my adaptation. Afterwards I began to design the most logical architecture model that will conform to the objectives of the adaptation.

As I begin development and testing, I will switch to the agile (iterative) software development life cycle. This means I will begin developing my adaptation in incremental

approaches and perform intermediate testing throughout. My justification for this is because, throughout my development, I will engage in user testing for each section. As well as that, I will need to continue to design areas of my adaptation that may be required to develop my adaptation. However, I did not notice that I needed these areas until the stages of development. This means I may encounter the need to begin designing new pseudocode and class diagrams for a system that I have not considered in design specifically. This will allow me to make changes to my ideas if needed and get user feedback that could potentially improve the overall outcome of the adaptation

With the need for iterative development mentioned, it evident that I may need to change aspects of my design whilst development. Thus, for areas of my user interface design, it is also evident that these areas may be subject to change based on developmental purposes and may not completely reflect the design in the outcome. However, this does not mean that I will greatly deviate from overall architecture.

Development and Testing

Throughout development, I will use object-oriented programming and develop each section of my adaptation in a modular fashion. As discussed, in the design, the use of iterative development will permit sequential testing after each section. This will include any failed tests and the process of amended testing.

Setting Up A Window

To begin development, I will need to develop a functioning game window with an OpenGL context in which all aspects of the adaptation will happen. This will involve developing the Window class and its initiation functions which were discussed in design. It will also involve the creation of the uber game class that will be responsible for decoupling the window code from the game code.

Testing Plan

To test my program, I will first test if upon running the program a window with a black viewport appears. I will then test to see I am able to click the "x" on the top right of the window to see if the window will disappear. This will test whether the event handling functions are working correctly and whether the window closed Boolean is being updated.

Development

To begin with, I first imported the necessary SDL2 library header files that will be responsible for providing the functions for creating the window with the OpenGL context. I also included the header files for the OpenGL toolkit/API that will provide all the

subroutines required to use the OpenGL specification for rendering. I also defined the header guards for the .cpp file using the #ifndef and #define macros. This is to prevent the source code in the header file being defined multiple times at each time it included in a separate .cpp file that uses the header. This will allow the usage of the functions and subroutines in other .cpp files without needing to include the actual .cpp file itself.

src > Window.hpp > Window

```
1   /*
2
3   Window.hpp
4
5   The header file for Window.cpp which is used for initalising a window
6   and creating an OpenGL Context within the window using SDL2 and glad
7
8   */
9
10  #ifndef WINDOW_HPP
11  #define WINDOW_HPP
12
13  // Include external libaries for window initialisation
14
15  #include "glad/glad.h"
16  #include "SDL.h"
```

I then defined constants within an anonymous namespace that will be used later to determine the dimensions of the game window. I have set them as constexpr and inline. This means that all uses of the variables are evaluated at compile time and replaced with the rvalue (the actual value, not the identifier) of the variables. My justification for this is that evaluating at compile time is more efficient than evaluating at runtime and will improve performance.

```
18  namespace
19  {
20      inline constexpr int WindowWidth { 800 } ;
21      inline constexpr int WindowHeight { 600 };
22  }
```

Afterwards, I defined the procedures and member variables for the window class that were included in the window class in the design phase and included their scope within the .hpp file. This included setting the member variables to private and the procedure definitions to public. This means that the member variables of the actual window are only accessible by

the class and its procedures; There is no access to these variables by any other subroutine outside the class itself. This will preserve the encapsulation of the object's properties and keep the object-oriented style of programming.

The definitions of the procedures are only the function definitions. This means that actual source code of the function will be written and implemented in the Window.cpp file.

```
23
24  class Window
25  {
26  public:
27
28      //  Getters and setters
29      SDL_Window*& GetWindow();
30      SDL_GLContext& GetOpenGLContext();
31      SDL_Event& GetWindowEvent();
32      int& GetWindowWidth();
33      bool& GetWindowClosedBoolean();
34      void SetWindowClosedBoolean(bool);
35      void SetWindowWidth(int);
36      int& GetWindowHeight();
37      void SetWindowHeight(int);
38      // Window constructor containing the width and the height of the window as parameters
39
40      Window();
41      ~Window();
42
43  private:
44
45      // Window class mememberss for window properties
46      int mWindowWidth;
47      int mWindowHeight;
48      SDL_Window* mWindow;
49      SDL_Event mWindowEvent;
50      bool mWindowClosedBoolean;
51
52      // OpenGL context
53      SDL_GLContext mOpenGLContext;
54  };
55
56  #endif
```

Afterwards, I declared the window class in the window .cpp files and began writing the source code for the definition of the Initialize function(). I first began setting the window width and window height variables within the constructor by calling the SDL_Init() function for the SDL2 library and passing in the SDL_INIT_EVERYTHING enumerator. This is to initialize the library and allow usage of its functions within my source code. With this, I added an error checking step during initialization to make sure the function has worked as intended. This is through checking if the result of the SDL_Init() function is greater than 0. If

it is not, then an error message will be logged to the console when running the adaptation and the contents of the actual error will be appended to the string on the console error using the SDL_GetError() function.

Afterwards the actual window is created using the SDL_CreateWindow() function. During this, the window position on the screen when it first loads is determined using the SDL_WINDOWPOS_CENTERED enumerators. This means that the window will be in the center of the users display when the game loads. Afterwards, the additional error check step is also implemented for the window creation except the difference is that it will check if the mWindow variable pointers to nothing. This will indicate that the result of the SDL_CreateWindow() function has not returned a window pointer due to an error.

src > Window.cpp > ...

```cpp
/*

Window.cpp

The source file for Window.cpp for initalising a window,
OpenGL states and the OpenGL context for rendering

*/

// Include libararies
#include "Window.hpp"

// Window constructor function to initialize window and its properties
Window::Window() :
 mWindowWidth { ::WindowWidth },
 mWindowHeight { ::WindowHeight }
{
    // Initialize SDL
    // Check if the initialization function was sucessful
    if (SDL_Init(SDL_INIT_EVERYTHING) < 0)
    {
        // Log a console error to show that initialization was unsucessfull
        // Append the actual error that is provided by SDL2 onto the string of the console error
        SDL_LogCritical(SDL_LOG_CATEGORY_APPLICATION, "Failed to initialize SDL! %s\n", SDL_GetError());
    }

    // Create the window and define its position, width and height properties and the type of window it is
    mWindow = SDL_CreateWindow("breakbeat", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
        mWindowWidth, mWindowHeight, SDL_WINDOW_OPENGL);

    // If the value return to the mWindow vairable is null then the window failed to be initiazlaied
    if (mWindow == nullptr)
    {
        SDL_LogCritical(SDL_LOG_CATEGORY_APPLICATION, "Failed to create window! %s\n", SDL_GetError());
        SDL_Quit();
    }

```

I then wrote the code to determine the OpenGL version that my adaptation is going to use. This was discussed during the Hardware Specifications chapter of my research. As discussed, the latest version of OpenGL 4.6 will be backwards compatible with the previous version and thus allow the different operating systems with older graphics drivers to run my adaptation.

I also set the OpenGL profile version to the core profile through the SDL_CONTEXT_PROFILE_CORE enumerator. The core profile is used for all versions of OpenGL above 3.3 due to OpenGL switching from the immediate mode profile (immediate mode refers to the older and easier method of drawing graphics) to the core profile mode (the core profile is the much more modern but harder to use method of drawing graphics). This means that the deprecated functions that use the immediate mode of OpenGL in OpenGL versions 3.2 and under will not function when using OpenGL 3.3

```
38      // Set OpenGL version to 4.6
39      SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 4);
40      SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 6);
41      SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, SDL_GL_CONTEXT_PROFILE_CORE);
42
```

After setting the OpenGL version, I then proceeded to instantiate the OpenGL context on the window using the SDL_GL_CreateContext function and add the additional error checking code that will be useful during the debugging process.

Finally, I loaded the glad OpenG toolkit library using the gladLoadGLLoader() function. This is required to allow access to OpenGL functions. I also implemented the additional error checking procedure, however this time I used the glGetError() function instead of the SDL_GetError() function. My justification for this is because the glad OpenGL toolkit library and SDL2 are separate from each other and therefore require different function calls to retrieve errors.

```
43      // Create the OpenGL context
44      mOpenGLContext = SDL_GL_CreateContext(mWindow);
45      if (mOpenGLContext == nullptr)
46      {
47          SDL_LogCritical(SDL_LOG_CATEGORY_APPLICATION, "Failed to create OpenGL context! %s\n", SDL_GetError());
48      }
49
50      // Load OpenGL functions
51      if (!gladLoadGLLoader(SDL_GL_GetProcAddress))
52      {
53          SDL_LogCritical(SDL_LOG_CATEGORY_APPLICATION, "Failed to initialize glad! %s\n", glGetError());
54      }
55  }
```

After the source code for the initialize() function was written, I proceeded to write the source code for the getters and setters in the window class according to the type of function required. This involved using the this-> keyword and the & keyword to return the variable relative to the class and to return the variable by reference respectively. My justification for returning by reference is to prevent inefficient copy initialization of the variables during the use of the function. This will save memory and optimize performance. The this-> keyword was also used to set the member variable relative to the class itself.

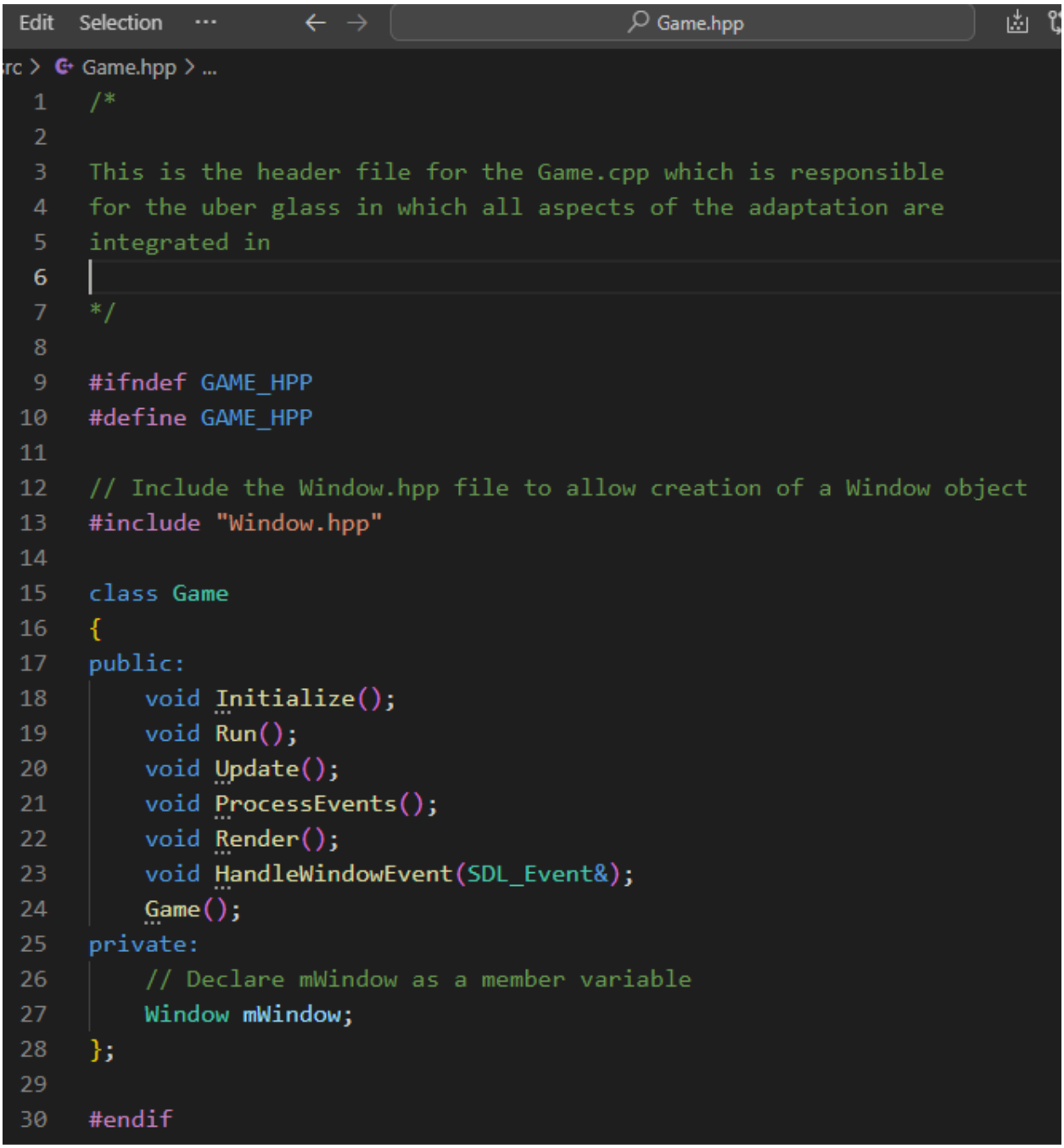
This is because there can be many instances of a class in object-oriented programming therefore there must be a form of disambiguation for the different member variables.

```
56
57  SDL_Window*& Window::GetWindow()
58  {
59      return mWindow;
60  }
61  SDL_GLContext& Window::GetOpenGLContext()
62  {
63      return this->mOpenGLContext;
64  }
65
66  int& Window::GetWindowWidth()
67  {
68      return this->mWindowWidth;
69  }
70  int& Window::GetWindowHeight()
71  {
72      return this->mWindowHeight;
73  }
74
75  SDL_Event& Window::GetWindowEvent()
76  {
77      return this->mWindowEvent;
78  }
79
80  void Window::SetWindowWidth(int width)
81  {
82      this->mWindowWidth = width;
83  }
84  void Window::SetWindowHeight(int height)
85  {
86      this->mWindowHeight = height;
87  }
88
```

Afterwards, I wrote the source code for the window destructor which deletes the OpenGL context and destroys the window once the use of the window class goes out of scope during the program.

```
88
89  Window::~Window()
90  {
91      SDL_GL_DeleteContext(mOpenGLContext);
92      SDL_DestroyWindow(mWindow);
93      SDL_Quit();
94  }
```

After creating the window class, I proceeded to write the function definitions for the game class in the game.hpp file. This involved defining the subroutines that were defined in the game class diagram, defining the class header files and declaring a private window object member within the class.



Afterwards I began writing the actual source for the game class's procedures. I first began by instantiating the window object variable with its default constructor. This means that window class construction is called as the window object is being instantiated.

```
src > Game.cpp > Game()
1   #include "Game.hpp"
2
3   Game::Game() :
4       mWindow(Window())
5   {
6   }
```

As I was about to begin writing the source code for the main game loop, I realized that for my game loop to loop continuously, it must have some form of condition to keep it looping. This condition must be decided when the user wants. Therefore, the condition for my game loop to keep running must be when the user decides to close the game window. To develop this feature, introduced a window closed Boolean in the private member variables and a getter and setter for the Boolean function. This is a feature that allows meeting of the user being able to exit the game via the “x” button success criterion.

```
27      bool& GetWindowClosedBoolean();
28      void SetWindowClosedBoolean(bool);
29      void SetWindowWidth(int);
30      int& GetWindowHeight();
31      void SetWindowHeight(int);
32      // Window constructor containing the width and the
33
34      Window();
35      ~Window();
36
37  private:
38
39      // Window class mememberS for window properties
40      int mWindowWidth;
41      int mWindowHeight;
42      SDL_Window* mWindow;
43      SDL_Event mWindowEvent;
44      bool mWindowClosedBoolean;
```

I also wrote the source code functionality of the additional window closed getter and setter

```
88
89  void Window::SetWindowClosedBoolean(bool boolean)
90  {
91      this->mWindowClosedBoolean = boolean;
92  }
93
94  bool& Window::GetWindowClosedBoolean()
95  {
96      return this->mWindowClosedBoolean;
97  }
```

For me to develop the use of an “x” button to exit the window, I must start writing the source for the ProcessEvents() source code. Within this procedure I began checking for window events using the written GetWindowEvent() function. This function returns the current SDL event that is being referenced in the mWindowEvent() variable. Once the window event has been returned, the contents of event variable are continually checked using the SDL_PollEvent and a while loop. Within this while loop, the event variable is checked using a switch statement. In this, the case of whether the event type is an SDL_QUIT is checked. This is possible as all event types in SDL2 are enumerated types. The SDL_QUIT enumerator is only called when the current window event is when the user is clicked on the “x” button on the window. Therefore, it is evident to set the mWindowClosedBoolean to true when this happens. Once this happens the main loop’s condition will be broken and the game will stop running.

```
8   void Game::ProcessEvents()
9   {
10      SDL_Event& event = mWindow.GetWindowEvent();
11      while (SDL_PollEvent(&event))
12      {
13          switch (event.type)
14          {
15              case SDL_QUIT:
16                  mWindow.SetWindowClosedBoolean(true);
17                  break;
18          }
19      }
20  }
```

After I wrote this code, I began writing the source code for the Run() procedure. I initated the main loop while loop and called the ProcessEvents() procedure within the main loop. This means the game will continually check for window events on each iteration of the game loop. My justification for this is that if the game loop did not check for window events on each iteration, then when the user clicks on the "x" button whilst the game is running, the game will would not close immediately and potentially risk of running indefinitely on the users system.

```
21
22  void Game::Run()
23  {
24      // Main loop for the game
25      while (!mWindow.GetWindowClosedBoolean())
26      {
27          // Handle input and window events
28          ProcessEvents();
29
30          // Update the window with OpenGL
31          SDL_GL_SwapWindow(mWindow.GetWindow());
32      }
33  }
34
```

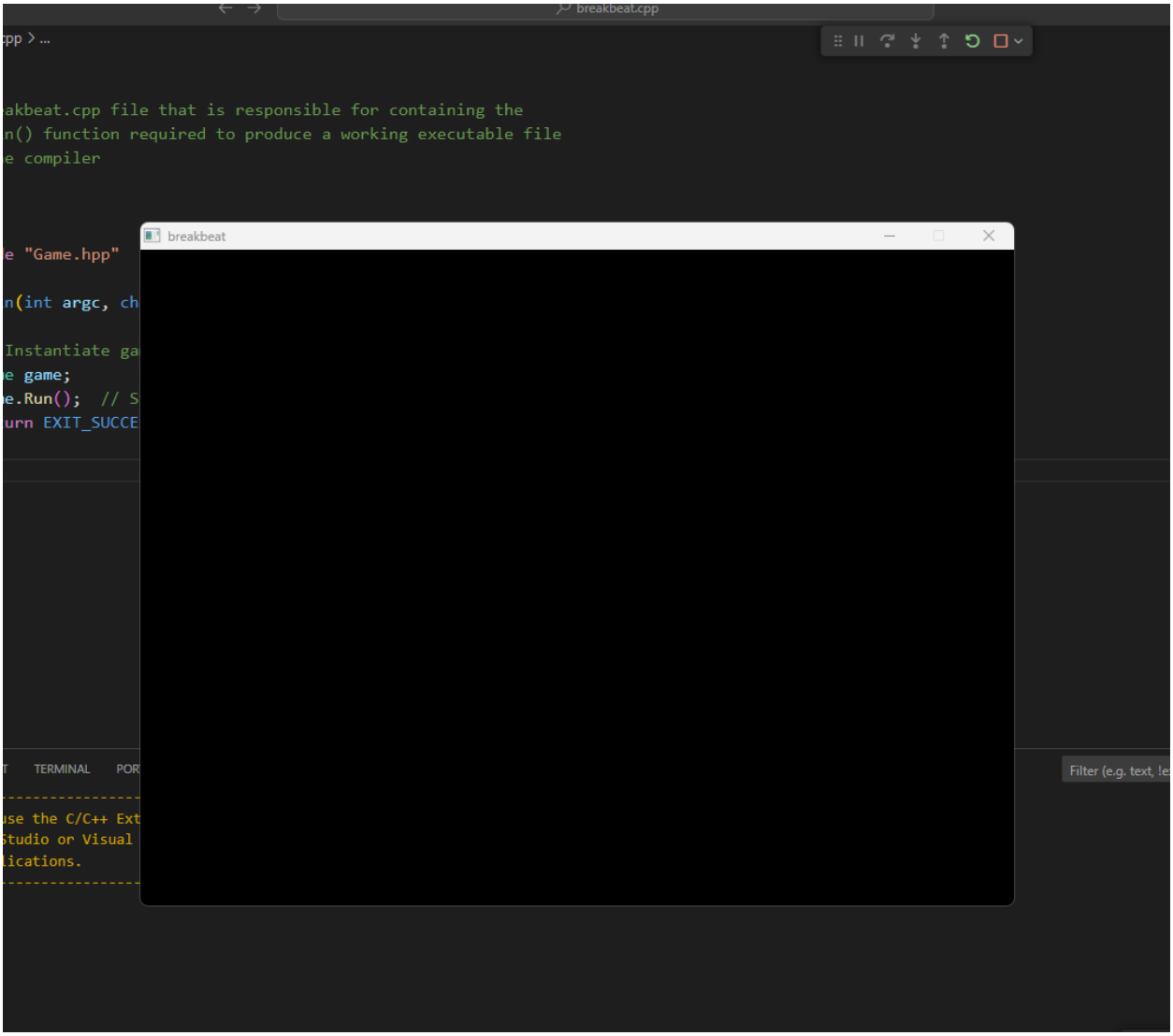
Afterwards, I then wrote the boilerplate int main() function source code that is must for all C++ programs that are being compiled. However, the difference in in the main() code is the usage of the int arc and char* argv[] as parameters in the function. My justification for this is because the SDL2 library uses a macro to define the usage of the main function. SDL2 uses #define main SDL_main within the headers for their source code. Because of this, the preprocessor will replace the int main() function with int SDL_main(). The actual int main() function is located within the SDL code. This is because the actual int main() is needed to perform initialization of SDL2 first. After the initialization has occurred, SDL2 then calls our main int main() function as SDL_main() and execution of our program that uses SDL2 begins. However, since SDL2 is a C based API, there is no function overloading meaning there can be only one prototype for the SDL_main() function. The SDL developers decided that it should be int SDL_main(int, char **).

src > breakbeat.cpp > main(int, char * [])

```cpp
/*

The breakbeat.cpp file that is responsible for containi
int main() function required to produce a working execu
from the compiler

*/

#include "Game.hpp"

int main(int argc, char* argv[])
{
    // Instantiate game object
    Game game;
    game.Run();  // Start the main game loop
    return EXIT_SUCCESS;
}

```

Testing

To commence testing, I compiled all the source files and ran the executable produced by the compiler. Upon running the program, a named window with the correct height and width properties appeared. This window was also freely movable on my display.



I was also able to click the “x” in the top left corner to test if the window event handling successfully worked and the window successfully closed itself. Thus, the success criteria of the user being able to exit the game via the “x” has been met.