Adaptive Audio and Visuals in Games Through Player Input (Game: Sea of Songs)

Ryan Crinnion

N00221108

Supervisor: Timm Jeschawitz

Second Reader: Joachim Pietsch

Year 4 2025/26

DL836 BSc (Hons) in Creative Computing

Abstract

This document explores the development of creating an interactive 3D environment where the accompanying music reacts to the user's input. To carry out this project, a short game made in the Unity 6 game engine was developed, integrating FMOD to handle the adaptive audio. The purpose of the game is to contain a short gameplay experience in which the player can explore the world while influencing the music and sound effects that are playing, and the game responds to this by changing the visuals and triggering gameplay events. The steps involved in the development of the system involved researching adaptive audio in media, creating original music and graphics for the game, creating a game environment and integrating the audio into the game using the FMOD audio engine. Testing was carried out throughout design and implementation.

The research, design and implementation, testing methods and results are discussed in this paper. The result of this project was a short gameplay experience where the player collects various tracks and expands upon them by further exploring the world and adding to the music.

Acknowledgements

Firstly, I would like to thank my family for their support through my journey through creative computing. This project is meant to represent the culmination of that journey, and I could not have made it to here without them.

I would also like to thank my supervisor Timm Jeschawitz, who went above and beyond with giving me advice and resources and matching my genuine excitement for this idea from the moment we first met. He helped me through the inevitable slumps of developing a large project and kept me focused on the goal.

I'd also like to thank my secondary reader Joachim, who provided advice on the game development aspects of the project and ran the game development module when I was in third year where I learned how to use Unity to create games.

I would like to thank all the friends I made during my time in IADT, as we all helped each other cross this finish line and finish strong on what they made an incredible 4 years.

Finally, I would like to thank my wonderful girlfriend and classmate, Emma, who kept me grounded since this endeavour began. Your hard-working spirit was one of my biggest motivators to get here, and I genuinely could not have done this without you. Thank you.

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Course Director.

WARNING: Take care when discarding program listings lest they be copied by someone else, which may well bring you under suspicion. Do not to leave copies of your own files on a hard disk where they can be accessed by other. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Creative Computing (Hons) course handbook. Please read carefully and sign the declaration below

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.

DECLARATION:

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

Student : Ryan Crinnion

Signed

Failure to complete and submit this form may lead to an investigation into your work.

Table of Contents

Table of Figures

1 Introduction

This paper covers the research, design, implementation, testing and project management for the application created for the 4th year major project of IADT creative computing.

1.1 Overall Aim

The overall aim of this project is to develop a short 3D game experience in which the player has an effect on the music and soundscape of the game through their input. This will be developed using Unity 6, with the FMOD audio engine being integrated to control the audio.

1.2 Application area

The application made during the course of this project falls under the entertainment sector, as a game development project. Video games exist in a place where programming, music and sound, and visual art as well as many other creative media meet, which combine to create an interactive experience. Game developers use the medium to tell interesting stories, or to create interesting scenarios using gameplay and mechanics. This project aims to explore this game development process, particularly with how game developers create audio and visual effects. The project serves to investigate development from an approach of music first.

1.3 Technologies

This section details the technologies chosen to develop this project. Comparisons with other technologies and justifications are made in Chapter 3.5 – Core Technologies.

1.3.1 Game Development Technologies

The game engine of choice for this project is **Unity 6**. Programming was done in C#, written in **Visual Studio Code** as the IDE of choice.

1.3.2 Audio Production and Editing Technologies

FMOD Studio was used to manage the FMOD banks and all audio in the game. **Ableton Live** was used to produce the music used in the game. All music in this game is written and recorded by me. **Audacity** was used to edit and create sound effects.

1.3.3 Project Management Technologies

Miro was used as an ongoing project management board to keep track of development as well as communicate progress with the supervisor of the project. **Trello** was used to decompose and itemize game development features and manage development in the form of a Kanban board. **GitHub** was used for version control of the game. Chapter 7 further details project management.

1.4 Project management

This project was split into 7 two-week sprints, with each sprint having different goals and reviews to evaluate the progress made during the sprint. The Miro board shows what was done during each sprint, as notes were created of what work was done during the sprint on it. The Trello board was split into the cards “Late to-do”, “Bugs”, “To Do”, “Doing” and “Done”. Tasks would be created and moved over to the right towards done as progress was made on the tasks. Trello was great for breaking down large problems into smaller ones and managing what to work on next.

1.5 Requirements

1.5.1 Functional Requirements

The functional requirements of the project outline what exactly the application should perform. From this definition, the functional requirements of this application include a gameplay system which the player has control over movement and music. The audio should affect the visuals presented from the game. As the application is defined as a game, there should be some form of *goal*, as often ludology defines games as rule-based systems with goal-oriented tasks. Costikyan (1994) defined the game as "an interactive structure that requires players to struggle toward goals".

1.5.2 Non-Functional Requirements

Non-functional requirements outline *how* the application should perform, detailing how to take the bare minimum foundation of the application to the intended vision. Thus, it can be said that the non-functional requirements of this application include a defined art style, and music to accompany the gameplay. The controls should also be intuitive and satisfying, and the game should provide the player with good feedback while playing the game.

1.6 Design

As the application is a game made in the Unity game engine, it is important the correct practices are applied to the hierarchal nature of Unity's workflow. Research into good practices was conducted to ensure that efficient and clean game object structures and asset management were standard. Clean management of the game development environment not only leads to smoother development and debugging but also results in good optimization for the game's performance.

The game also requires many visual and audio assets to create the gameplay environment that suits the application. The music used in the game is all original music written and composed by the author. All 3D models and most textures are original works.

Chapter 4 further details the design stage of the project.

1.7 Implementation

The design and implementation phases during the development of the game occurred simultaneously. Given the volatile nature of game development, code and logic would often need to be changed or rewritten "on the fly". The implementation chapter deals with applying the work outlined in the design chapter to the game through logic and code and includes code snippets to give context to how things work. This chapter details the choices for the logic and explaining in detail how systems work such as the water and buoyancy, save, and inventory systems. It also gives a detailed review of each sprint and the work carried out in each.

1.8 Testing

This chapter covers the functional and user testing done for the project. It includes feedback given from testers and how that may have been implemented in the application to iterate on the work.

2 Research

2.1 Introduction

Video games are games played on common electronic devices (Video Games: Overview | EBSCO, 2020). These devices include PCs, consoles, phones, etc. Input is taken from the player, which allows them to interact with the digital world. Video games are intrinsically an interactive medium, as the player and the game react to each other as the game is played. Video games cover a wide range of

genres, with different levels of demanded skill and engagement. They are a medium that can be enjoyed by all, and as of recent years they continue to develop a strong footing in mainstream media.

The video game industry has gone on to become a multibillion-dollar field of business. As production costs and budgets increase, as well as the rapid development of new electronics, video games have gone from simple entertainment experiences to large scale projects, acting as a culmination of programming, visual art, narratives, music, advertisement, acting, cultural significance, and more. With the technology that is now accessible for video game development, developers are able to create unique experiences that are only available through video games.

This focus of this literature review is on the different ways in which video game developers can create unique experiences only available within said medium. The review will highlight and investigate existing practices that are utilized by developers, acting as common ground between games. It will also examine video game music in particular, analysing how developers bridge the gap between programming and music.

This literature review should provide sufficient research on video game development practices that encourage interactivity from the player. It should also establish what practices entice the player to continue playing, by creating a positive feedback loop. This review will reach these conclusions using research to reach firm conclusions on the prior topics.

2.2 Immersion in Video Games

2.2.1 Defining Immersion

Immersion is becoming deeply involved with something (*Definition of IMMERSION*, 2019). In the context of games, immersion is used to define the feeling that you are a part of the world that you are experiencing (*What Is Immersion : Immersion Definition | Unity*, 2025). It is a key factor in video games given that they are an interactive medium, and research suggests that the immersion a player experiences in tandem with the gameplay experience contributes to the overall flow and enjoyment of the game.

Arguably, immersion can also be considered unimportant and in some cases useless depending on the player. Because everyone is different, immersion will hit everyone differently. It's somewhat subjective, as it is with everything that gives someone a unique experience. However, research has been conducted on how creating immersive experiences in games can contribute a positive effect on the player's experience with the game (Örtqvist & Liljedahl, 2010).

Due to its subjective nature it's difficult to quantify immersion, as even asking a player if they are feeling immersed can break the concentration that creates that immersion. It can be difficult to define immersion in video games as well, given how many factors are at play to create the experience, especially in modern games as they grow bigger and more complex. A player can't "force" themself to be immersed, as it's a state one must slip into naturally.

Thus, it can be said that immersion is a subtle but powerful force that games can create by focusing on aspects that will contribute to the overall immersion of the experience. It's something that is easy to notice when it's wrong, but difficult to when it's right. This speaks to the illusion that immersion creates by blurring the lines between the player's reality, and the game's.

2.2.2 Immersion and its Contribution to "Flow"

Flow is a state of concentration that exists in a similar realm to immersion. Flow is described as an "optimal experience" by Mihály Csíkszentmihályi (2009). He further elaborates on flow being a state where one's consciousness intensifies and self-consciousness disappears, and that in this state is when one is performing their best (Csíkszentmihályi, 2009, 2016). This "optimal experience" is what game developers strive to create for their players.

Several studies have been carried out on why exact people play video games, yielding a variety of results. Notably, many describe the emotional benefits of gaming.
One study (Russoniello, O'Brien, & Parks, 2009) shows evidence that casually playing video games a few times a week in preferred games can reduce symptoms of clinical depression. "Flow" is described as a key factor of this positive emotional experience, as it allows players to gain a high sense of control in an intrinsically motivating activity. These flow experiences have been linked to positive emotions, eliciting higher self-esteem and less anxiety. As flow is described as an "optimal experience", it can be said that the developers of games are attempting to create a positive feedback loop, encouraging engagement and participation from the player, and the player devotes their time and effort into playing the experience that the developers created.

Immersion is a heavy contributor to this flow state. Immersion itself is not necessary to achieve the flow state, but it's an important factor to consider when making a good video game. Immersion doesn't require the best graphics, or sound design, but rather a positive feedback loop of rewards and emotions that the game provides that will make it easier for the player to lose their sense of self-consciousness.

2.2.3 Immersive Factors in Games

Immersive factors in games are extremely subjective, as in asking friends and colleagues their opinion on what makes a game immersive, as well as researching forums online; every answer varied on what specifically immersed them in a game. These factors include narratives, characters, world, user interface, controls, and how the game teaches you its mechanics. Many also said that graphics and visuals of the game itself are not prime factors for a flow state.

The narrative of a game does not necessarily have to come from the narrative of the story of a game. For example, some games have clear and defined narratives that take the player on a journey and experience a story, such as Silent Hill 2, Undertale, and many more with defined characters and story beats, possibly with multiple endings based on the decisions that the player made throughout the game. The narratives present in these games are praised for their writing and depth, often with players relating heavily to the themes and characters explored in these stories.

Another type of narrative present in games is a meta-narrative that the player creates with their experience with the game. Some games, such as Balatro and Mario Kart, do not have a story embedded within the game itself. The narrative produced by these games comes from the player's experience with them. Each time you play the game, you get a unique experience with your own conflict, climax, and resolution.

Regardless of the experience the game offers to the player, there is a common trait between them, being a goal. Costikyan (1994) defined the game as "an interactive structure that requires players to struggle toward goals". He also elaborates on the interactive elements of games. Games are interactive by nature, but their interactivity goes deeper than other mediums. Films and music are interactive, as you still need to actively watch them and start them by buying a ticket, opening an app, or inserting a disc into a drive. The same can be said for games, but the decision-making required for this interactivity goes deeper. When playing a game, each decision is informed by the

state of the game. Some options may be better than others, and that might change over time. You may require resources to make decisions, and those decisions lead to a goal. The player inserts themselves into this world, as a character or as a power to make decisions, to reach the goal in the narrative (Leblanc, 2019). This insertion is a powerful immersive factor that games possess, and it brings out the "flow state" previously discussed out of players, even subconsciously. It's clear why many would consider this sense of narrative as an immersive factor of games.

2.2.4 Feedback to the Player

"Juiciness" is a term used to describe audio and visual positive feedback in video games. A study (Kao, 2020) explores various levels of juiciness within the same game; from none, medium, high, and extreme. The study of 3018 participants found that no feedback and extreme feedback lead to lower playtime and lower enjoyment of the game, due to under- and overstimulation. The study also suggests that too little or too much feedback can remove intrinsic motivation from the player, so they want to stop playing the game.

Antanasov (2013) investigates juiciness as well, and how feedback to the player is important in all aspects as video games are multi-sensory experiences. It has a focus on aesthetics. It highlights many examples of feedback systems in games, their design, their function to the player and how their initial conception may have come about.

Juiciness can be supplied to the player on both an audio and visual level, and some other ways like controller rumble. In particular, many games use audio cues to give you feedback on how well you are doing a particular task in the game. As an example, the game Overwatch plays a light tick noise when you hit an enemy player with damage. In this game, damage to enemy players' heads does double damage, so when you hit a headshot, the game plays a satisfying "ding" sound to signify you hit a headshot. The sound both serves to let you know you hit a headshot while also giving the player a small dopamine hit by receiving the headshot sound. The game rewards you for landing a headshot by dealing double damage and your brain makes the connection that hearing that sound means you are playing well.

2.3 Music – in Life and Games

Media uses music to convey emotions such as joy, sadness, anger, etc. It is a tool that can be used to tell the audience how to feel.

2.3.1 Emotional Response to Music

Studies suggest that music evokes genuine emotional response from the listener. Lundqvist, Carlsson & Hilmersson, Juslin (2009) conducted a study to investigate the emotional response to music. The experiment investigated how 32 musically untrained individuals responded to different kinds of music, both in their perceived emotion and physiology. The results yielded a coherent change in the individuals' expression and perceived emotion. The study showed that when listening to happy music, the muscle region responsible for smiling became more active over listening to sad music. The subjects of the experiment also self-reported their emotional response to the music, reporting feeling happiness when listening to happy music and sadness when listening to sad music.

Humans have a physical response in the brain to music. The limbic system in the brain, which is linked to emotion and memory, and importantly understanding the emotion from the inflections in speech, is activated through listening to music (Pfizer, 2024). There is a direct link that can be drawn between listening to music and having an emotional response much like you'd have an emotional response by being told something happy, sad, or aggravating by someone.

2.3.2 Music in Media

Video games use music to create emotion and memorability in important scenes, action sequences, encounters, character introductions, and many other scenarios in game. Many games will use leitmotifs for important places and characters. According to Heckmann (2020), leitmotifs are a musical theme that composers can use to signify a moment's importance to a character, idea, or situation. An example of a leitmotif in movies is The Imperial March from Star Wars. Whenever Darth Vader walks on-screen, you will hear the first few notes of his iconic theme play. This solidifies The Imperial March as his theme song, and by extension the theme song of the emperor. The overbearing French horns playing the dark and militaristic melody convey his character, and through repetition of playing it each time we see Vader on-screen, we associate it with him.

Leitmotifs are a powerful tool for composers and storytellers because it can add to the presence of a character, a location, or an idea. Some instances of a particular leitmotif may be played in a light and playful tone, whereas later it may come back in a dark and menacing tone if something tragic has happened to a place or character.

Murphy (2019) discusses the usage of leitmotifs in the context of video game stories, showing examples of how leitmotifs are used to associate key moments of the story with characters based on the leitmotifs being played during those moments. He also discusses how leitmotifs can be changed to signify changes in characters and the story. Leitmotifs are a powerful way of providing emotion and context to a reader without necessarily using visuals, or in this case the screen, to do so.

2.3.3 Dynamic Music in Video Games

Dynamic and adaptive music in video games are similar concepts that involve the music seemingly reacting to what is happening in the game. There is a slight difference between dynamic and adaptive, but both can be used interchangeably. For this review, "adaptive music" is specifically when the music reacts to the player's input, while "dynamic music" is used to describe music that is non-linear.

These techniques in composition are used to create more impactful moments in games, as it allows the developers and composers to create music that will perfectly sync up with the game. Some games change the music while you are on low health. The Pokémon (Nintendo, Game Freak, & Creatures Inc, 1998) series and Street Fighter (Capcom, 1987) are well-known for using this idea in its gameplay. Other games will increase the energy the better you are performing in-game. Devil May Cry 5 (Capcom, 2019) is a celebrated example of this practice. There are many examples of dynamic and adaptive music being executed in games, as it's a simple but powerful technique to increase immersion and memorability of a game and its soundtrack.

Sorrenti (2025) lists some of the capabilities of adaptive audio that developers have access to with tools like Wwise (for Unreal) and FMOD (for Unity). Some of the examples they list include dynamic audio transitions and triggering audio events through gameplay actions.

2.4 Modular Music

Vagnini (1998) developed a theory on open-source musical composition, where one composer or more can overlap several compositions to create a new piece of music. This is done by adding or removing individual compositions, called "modules", from the piece depending on the context of the moment.

Modular music is a technique that can be applied to create adaptive music in video games. When a composer for a game's soundtrack is creating a song, they will then divide it into different parts;

such as drums, bass, lead, etc and then layer those parts to react to what is happening in-game . Tarro57 (2023) made a YouTube video explaining this concept further and how it is applied specifically to video games. For example, a constant drumbeat could be playing, and then when your character does an attack it might play a chord on the guitar. This type of modular music is called “vertical sequencing”. There is also “horizontal sequencing”, which is when you divide entire sections of a piece with all instruments to make the soundtrack fit perfectly with the moment. If something exciting such as an explosion is about to happen in-game, but it requires the player to move to a certain position first, a low-energy droning may loop until the player reaches the trigger for the explosion, so the music will always sync with the event in-game.

2.4.1 Diegetic Music and Sound

Diegetic music, or source music, is music that originates from within the world of fiction, so that both the characters in the fiction as well as us as the audience hear the music (The Art of Scoring - PC Feature at IGN, 2025). This is a concept that exists in all media, which includes video games. An example of diegetic music would be if a character in a level is playing an instrument, and you can hear the music that they are playing. It is music that exists within the world, and it is implied that the characters of the game can hear this music and interact with it too. Something as simple as having an in-game music player that a character must interact with instead of it being done through a menu can add charm to your world, leading your players to grow more attached to the world and its characters (The Art of Scoring - PC Feature at IGN, 2025).

Altenstedt and Willig (2025) explore the soundtrack and sound design of the video game Outer Wilds. It is a game that utilises diegetic and modular soundtrack design as a prime example of the possibilities the medium offers. For example, there are different musicians placed around the physical planets of the game’s simulated solar system, with one musician on each one. The character can hear them playing through a radio, and if they point the radio out into space while 2 or more planets are aligned both can be heard playing different parts of the same piece of music.

There are many other examples of games that utilise diegetic music. The Fallout and Grand Theft Auto series both have in-game radio stations that play real licensed music on them, along with containing advertisements and shows that play between songs. In a couple of The Legend of Zelda games, namely Ocarina of Time and The Wind Waker, the player controls instruments that affect the world when played, such as changing the weather or time of day. Guitar Hero is a game where the player plays along with a virtual band using a physical guitar controller, and if they are missing notes the music will change and remove their instrument.

2.5 Summary

This literature review has examined some practices used to enhance player experience and immersion. It also discusses music and its effects when used in media, and how music can be applied in video games to enhance the player experience. It highlights dynamic sound and music in games and how this particular technique can be used to combine game design and sound design. In summation, the application of dynamic and adaptive audio games is shown to be an interesting marriage between the game design aspect of creating a game, with player feedback and enjoyment in mind, as well as the sound design aspect, particularly with music and the emotional and narrative connotations that it can carry. These elements were explored in the final chapter, which highlighted use cases of dynamic audio and how they are applied in games, giving diegetic and “juicy” feedback to the player.

3 Requirements

3.1 Introduction

Title: Adaptive Audio and Visuals in Games Through Player Input

Overview: This project is a short game experience that contains a deep relationship between input, music, and visual feedback. Its purpose is to explore the effects of having these three elements of game design so interwoven.. The intention of developing this project is to investigate if having music react to player input or the game changing depending on the music adds to the perceived enjoyment for the player. The game has a nautical setting, in which the player must sail a boat to collect musicians from different islands, and they will contribute to the music playing in the background.

Context: Game Design, Game Development, Interaction Design/UX, Sound Design

3.2 Requirements gathering

Overall Aim: The aim of this project is to develop a game where the music reacts to the player, and the game reacts to the music.

Specific Objectives:

- Integrate adaptive audio software (FMOD) into Unity to support cueing in different layers of music i.e. playing specific instruments at specific times, manipulating audio of specific instruments with effects
- Create a system that interprets player input that changes the music being played.
- Create a buoyancy system to emulate ocean waves and boat movements
- Create a system that creates visual feedback based on the music being played i.e. particles on beat with the music, visual effects that fit with what is currently being played
- Write pieces of music to fit the game and record individual tracks to be layered in-game

3.3 Requirements modelling

3.3.1 Backlog of features

Feature	Description	Priority
Layered Music Tracks	Recording pieces of music with each individual instrument being exported as its own file. This is for the FMOD integration, so that Unity can control the volume and effects of each instrument in relation to the state of the game.	HIGH
Audio Reactive Visuals	In order to realise the project, it requires visuals in-game that react to the music i.e. particles in sync with the beat, environmental factors changing based on which instruments are being played, etc.	HIGH
Boat/Wind Controls	The game requires snappy boat controls that feel satisfying for the player, as it is the main interface for	HIGH

	interaction with the world of the game. V ious ways of controlling the boat are being tested, as well as fine tuning drag and inertia with the water. The player controls the wind that the boat sails on. Controls for both keyboard and controller players.	
Player Control Over Music	The player needs to have some form of control over the music being played in the game. This is done by collecting musicians, and changing what is being played with the controller while you are sailing.	HIGH
Visual and Audio Feedback	The game should provide the player with visual and audio feedback for when they are sailing well (i.e. wind particles, sound effects, controller vibration)	HIGH
Boat Model	Simple stylized sailboat model.	HIGH
Environmental Assets	Islands and buoys	HIGH
Visual UI	Visual UI for things like wind direction, player input, current weather, etc.	HIGH
Advanced Water Shader	A water shader that includes wave simulation (on the GPU), sea foam motion and edge detection for objects contacting water	MID
Character Art and Assets	Simple models or drawings of the musicians, possibly brief voice acting to explore FMOD's abilities more	MID
Water and Wind Physics	Buoyancy simulation for boat, wind draft simulation that affects the speed of boat.	MID
Cel-Shading Shader	Cel-shading stylized shader for game's art style	MID
Weather	Weather effects in-game. Clear sunny weather to dark foggy storms with changes in environment backgrounds and lighting, water colour, and ambience	MID
Narrative Conclusion	Some sort of brief narrative conclusion for the player to tie the game's goal together. Bare minimum detail	LOW
Controller Vibration Feedback	Vibration in controller for feedback to reflect what is happening in-game	LOW
Wind Drafts	Wind drafts that give the player a speed boost if they sail along them	LOW

3.4 Feasibility study

3.4.1 Summary

The feasibility study provides an insight into the technical, economic, and operational achievability of this project. It is intended to lay out all of the challenges and their solutions to ensure that the project is complete and delivered within the module's timeframe.

3.4.2 Description of Project

The project that is to be developed during the module's timeframe is a vertical slice of a game in which the player controls the music through playing the game, and the game's state responds to the state of the music. Player input -> affects game as well as music -> music affects visuals i.e. particles, colours, etc.

3.4.3 Technical Considerations

Technical considerations reviews the technologies being used as well as the work involved with those technologies. The main technologies being used to develop this project are:

- **Unity**, as the game engine of choice (C#)
- **FMOD**, as the audio engine that integrates with Unity
- **Ableton**, as the digital audio workstation (DAW) for recording and producing music
- **Blender**, as the 3D modelling software for the limited modelling this project contains

The priority of work to be done in each software also falls in this order.
Unity will be handling everything to do with the game, from code to visuals and shaders. FMOD handles audio by building "banks" that are imported into Unity, from which Unity can control the volume and effects on each individual instrument. FMOD will also be handling sound effects, as it has completely replaced Unity's default audio manager in the project.
Ableton will simply be used for recording music and mixing it appropriately. All the instruments and plug-ins required for this project have already been sourced.

Blender is for simple models such as the sailboat, islands, and buoys.

A prototype was developed before the official start of the project to gather a sense of the workflow and learning involved in developing the game, which was pivotal to establishing confidence to be able to carry out this project. This project was one that demanded competence in several technologies, so it was essential that a foundation of knowledge was established in each technology.

3.4.4 Economic Considerations

All software in the technical considerations are free, using the personal plan for Unity, FMOD studio and the free tier of the FMOD engine, Ableton Lite previously acquired through buying an audio interface, and Blender which is free and open source.

3.4.5 Operational Considerations

The deadline for the final project is Friday, May 1st, 2026, which is 14 weeks from the submission date of this proposal. Over the course of the 14 weeks, development will be carried out through a series of sprints and regularly coordinate with the project's supervisor to ensure development is on pace and running smoothly. The plan is for the first half to be finishing the research and learning of the technologies, as well as creating a strong foundation for the project to be built on and getting into developing as soon as possible. The second half will be finishing the development, testing, and polishing the project down.

3.4.6 Conclusion

Given the considerations above and the experience and confidence with the technologies being used, the completion of development during the allotted time is very achievable. A strong vision of what the game will look, feel, and sound like has been established.

3.5 Core Technologies

3.5.1 Unity

Unity is a game engine with a free tier for personal use, which is perfect for a small-scale project like this, though it is certainly a powerful option for game developers as even AAA studios use Unity as their game engine of choice. Unity also goes hand in hand with FMOD, having support for direction integration with FMOD through an official plug-in from the Unity Asset Store. Unity also has a powerful visual scripting tool for creating shaders in the Shader Graph. This allows the developer to create their own shaders to be used in-game without having to learn GLSL on top of what is required for this project. Unity's scripts are written in C#, which means a level of competence in programming in this language needs to be established.

The other options that could have been used for the game engine were Unreal Engine and Godot. Unreal Engine is primarily used by AAA studios to create large-scale games, though it certainly has been used to create small indie games as well. Unreal has a different audio engine that it integrates with, being Wwise, however many consider Wwise to be the more complicated software to use for game development (Sanyshyn, 2024). That might appeal to other projects more but given the scope of this project Unreal and Wwise would be not only overkill but also a lot of extra learning as C++ would be a new language to learn.

Godot is another strong alternative; however, its 3D rendering capabilities are not quite as good as Unity's (Manasa Jayasri, 2024), and it lacks the integration with FMOD that Unity has. Still, an extensive knowledge about game development and game engines in general working with Godot win the past, so applying these practices in Unity will be useful.

3.5.2 FMOD

FMOD, through research and prototyping, is the best option for an audio engine for this project. FMOD enables the developer to bridge the gap between game development and sound design. All sound files for the game (sound effects and music) are imported into the project in FMOD studio, where the developer can set parameters that they can manipulate in code through Unity, such as volume, reverb, delay, etc. Because FMOD has a Unity plug-in, whenever the developer builds the FMOD project, it will automatically import and update within Unity. This speeds up the workflow from making audio to getting it into the game and setting it up correctly by a great degree.

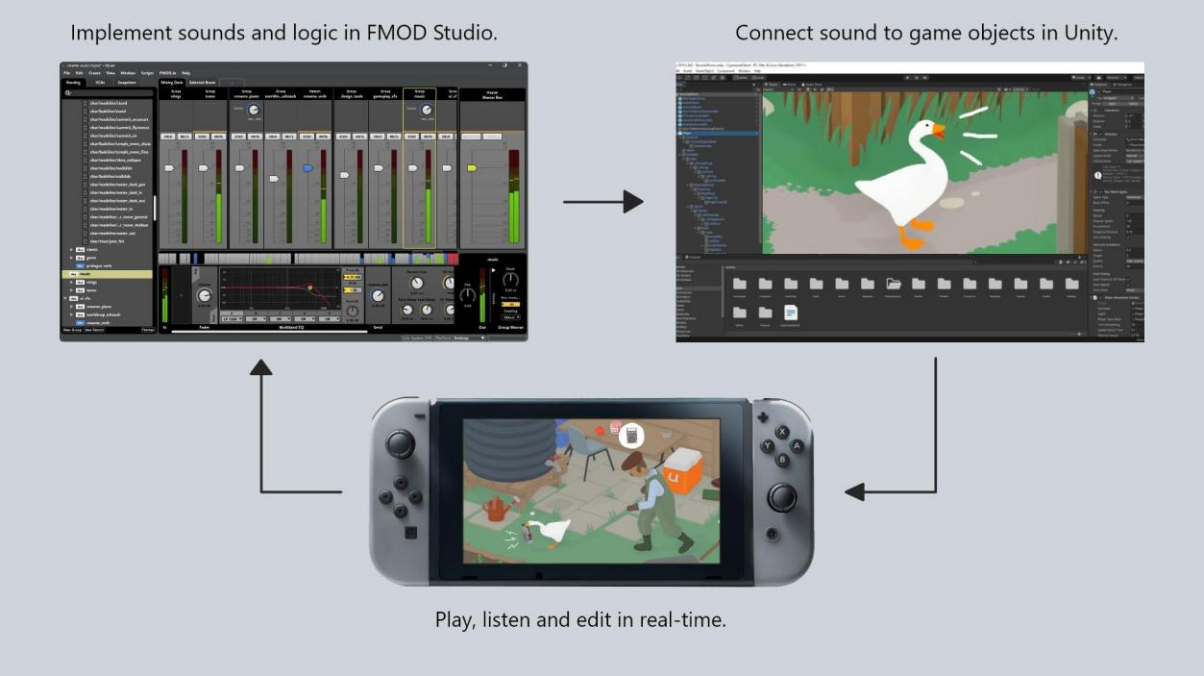

Figure 1 FMOD Studio to Unity - Image from FMOD.com

Alternative options include the previously mentioned Wwise, however that software has a lot more depth and a higher learning curve, and it is often used in tandem with Unreal Engine rather than Unity. It is interesting to see what other options are available and how they can each be used for the specific needs with this project.

3.5.3 Ableton

Ableton was the digital audio workstation (DAW) of choice because it's what the developer owns already and is most comfortable with. All the gear and plug-ins are already set-up within Ableton as well, so it's really the best option. The version of Ableton being used, Ableton 11 Lite, only lets the musician record up to 8 tracks.

Other options for DAWs include software like FL Studio, Reaper, or LMMS, but these are software that are either paid or new software that would be unfamiliar to use.

3.5.4 Blender

Blender is the 3D modelling software of choice, and it's what will be using for modelling, animation and UV unwrapping. Blender is a free and open-source program that is used for modelling, texturing and animation. FireAlpaca is also free software and is used for digital art creation that will be used for creating textures.

Blender being as powerful as it is while also being free and open source, as well as the developer's own confidence in the program make it the best option for this project.

3.6 Conclusion

This chapter serves the highlight the requirements for the project, including the technologies and the approach needed to complete the project within the given time limit. The core idea is to create a tightly interconnected system where player actions influence the music and in turn the music affects the visuals. The game is set in a nautical environment where players sail between islands, collecting musicians which contribute to the soundtrack.

The feasibility study concludes that the project is achievable within the 14-week time limit. Technically, it relies on Unity for development, FMOD for audio integration, Ableton for music production, and Blender for asset creation. All tools are accessible at no cost, minimizing economic constraints. Development will follow an iterative sprint-based approach, with early focus on research and prototyping, followed by implementation and polish.

4 Design

4.1 Introduction

This chapter deals with the design of the project. It describes the work that went into planning the application and what it needs to do to be successfully complete its intent. It also serves to justify the design decisions made.

4.1.1 Game Overview

The game created for this project is a short sandbox experience in which the player can control the audio and visual effects of the game through their input. The game reacts to the music that the player decides to use, which changes how the environment looks and reacts to the player. The game was created using the Unity game engine, with a specific focus on audio being handled by the FMOD audio engine integrated into Unity. It is only available to be played on PC, though multiple builds have been created to support Windows and Linux operating systems. The game can be played with keyboard and mouse or with a controller. It also has a save/load system so that the player can close the game and return later with their progress saved so they can continue playing where they left off.

The game offers a unique experience in interacting with music and visuals in video games and offers a strong foundation for incredible expansion on the idea.

4.1.2 Core Mechanics

The core mechanics of the game involve the movement and track selection. The player can move north and south with W and S (or the left stick on controller), and west and east with J and L (or the right stick.) This decision is elaborated on in the decisions section later in this chapter. The player can also switch tracks by pulling up a “selection wheel” with Control or R2. This lets the player select the base track that is playing. From there they player must collect the rest of the piece of music to complete the song.

4.1.3 Game Loop

The game loop involves the short-term goal of collecting pieces of a track to change the music, and the long-term goal of collecting all the music in the game to complete the game.

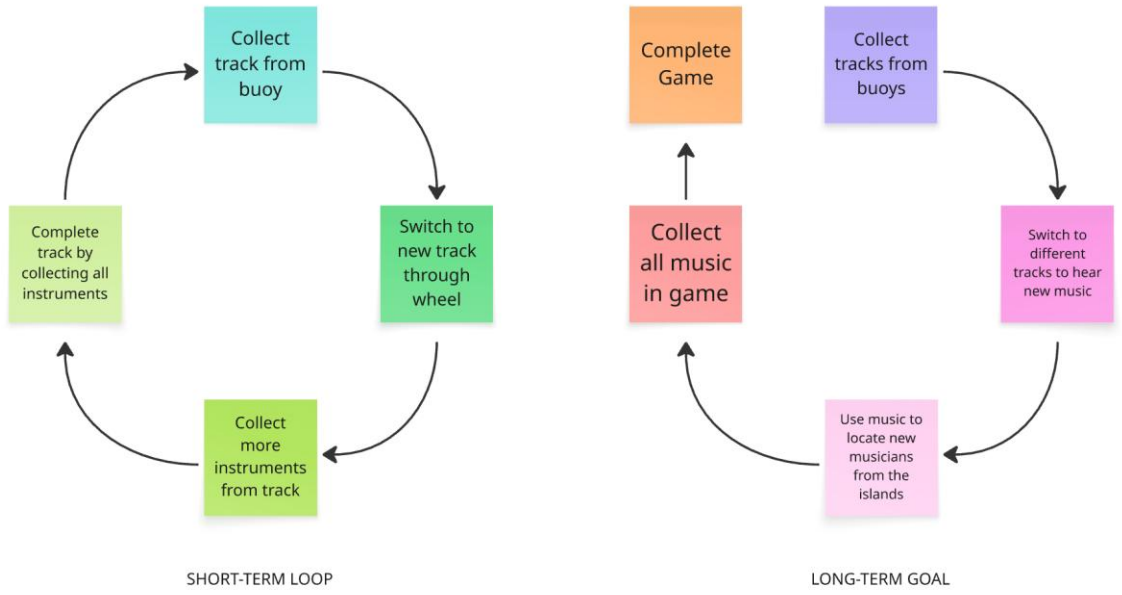

Figure 2 Gameplay Loop Diagram

Players are intrinsically motivated to complete tracks to hear the full song and see the different visuals offered by the game. Long-term, they are extrinsically motivated to complete all tracks by being given this task as an explicit goal.

4.1.4 Level Design

The level the players are given is a vast open ocean that's dotted with buoys and islands that serve a functional purpose. The **buoys** offer the player the "base" of a new track, which unlocks the song to be selected from the wheel in the UI. This will unlock one instrument of a piece of music and change the visuals of the environment. The **islands** give the player more instruments from the selected track, and each one is assigned a specific instrument from a specific song. For example, you can't get the instrument from a different song while you are playing a particular song. The islands have unique visual differences depending on the different songs and have particle effects to indicate that the island is currently "resonating" and has not been collected yet.

When you start a new game, the player is locked into a small area confined by a border that is removed once the player completes a short tutorial. This tutorial is to teach the player how to play the game and what their goal is. It's important for a game to teach a player how and why they need carry out a goal to get the player into the action as quickly as possible, and the first few minutes of the game are very important for this reason. The tutorial was designed to encompass the entire game experience within a single small area, so that by the end the player knows what to expect and how to fully finish the game.

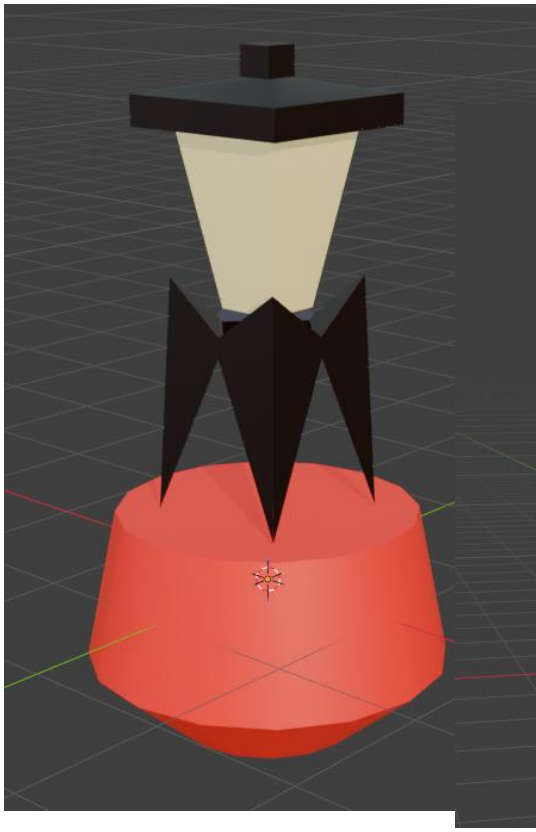

Figure 4 Buoy Model

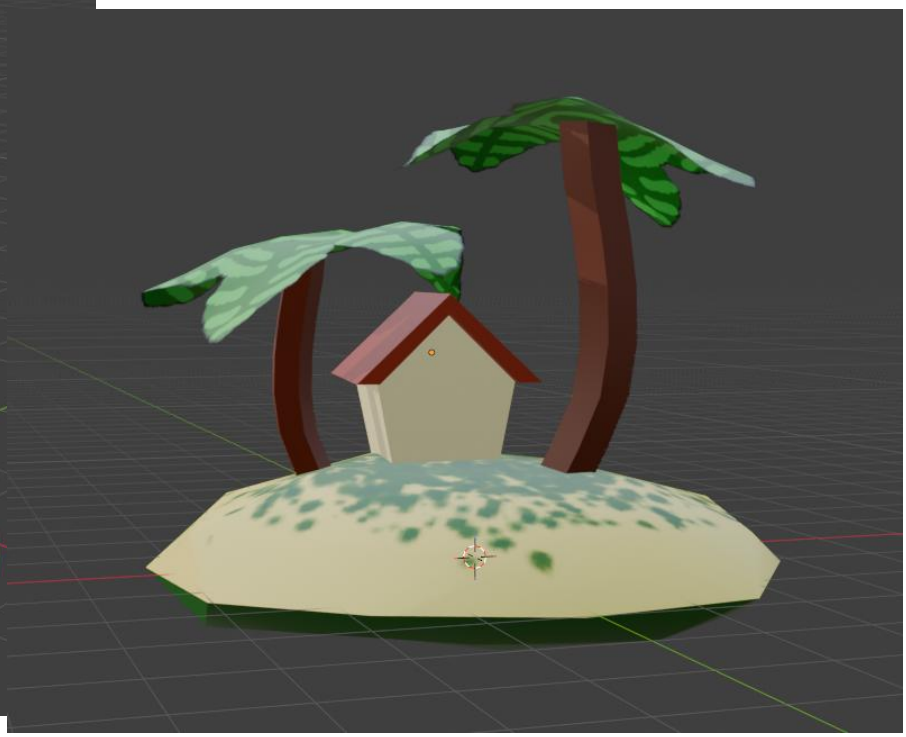

Figure 3 Island Model

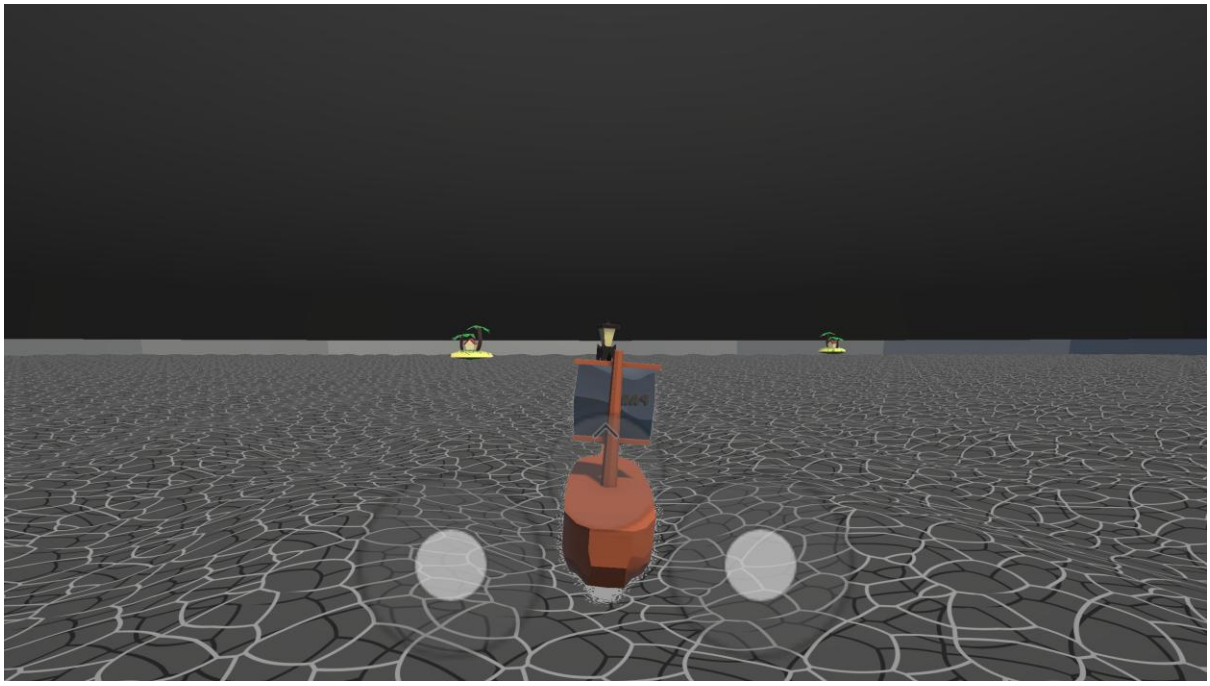

Figure 5 Tutorial Area

4.1.5 UI/UX Design

UI and UX design in games is very important, as the gameplay experience is elevated by effective, frictionless UI.

The UI for the game should be non-invasive, offering the player just the important information they need in the moment. The first element of the UI that was created was the interface for the current directional input. The two circles represent the left and right stick, and the circle in the middle with the arrow is the current direction, working like a wind vane indicating your direction in respect to the cardinal directions. It is the direction of the resultant vector of your north/south movement and east/west movement. For example, if you travel northwest it points up and left, and if you travel east, it points straight to the right.

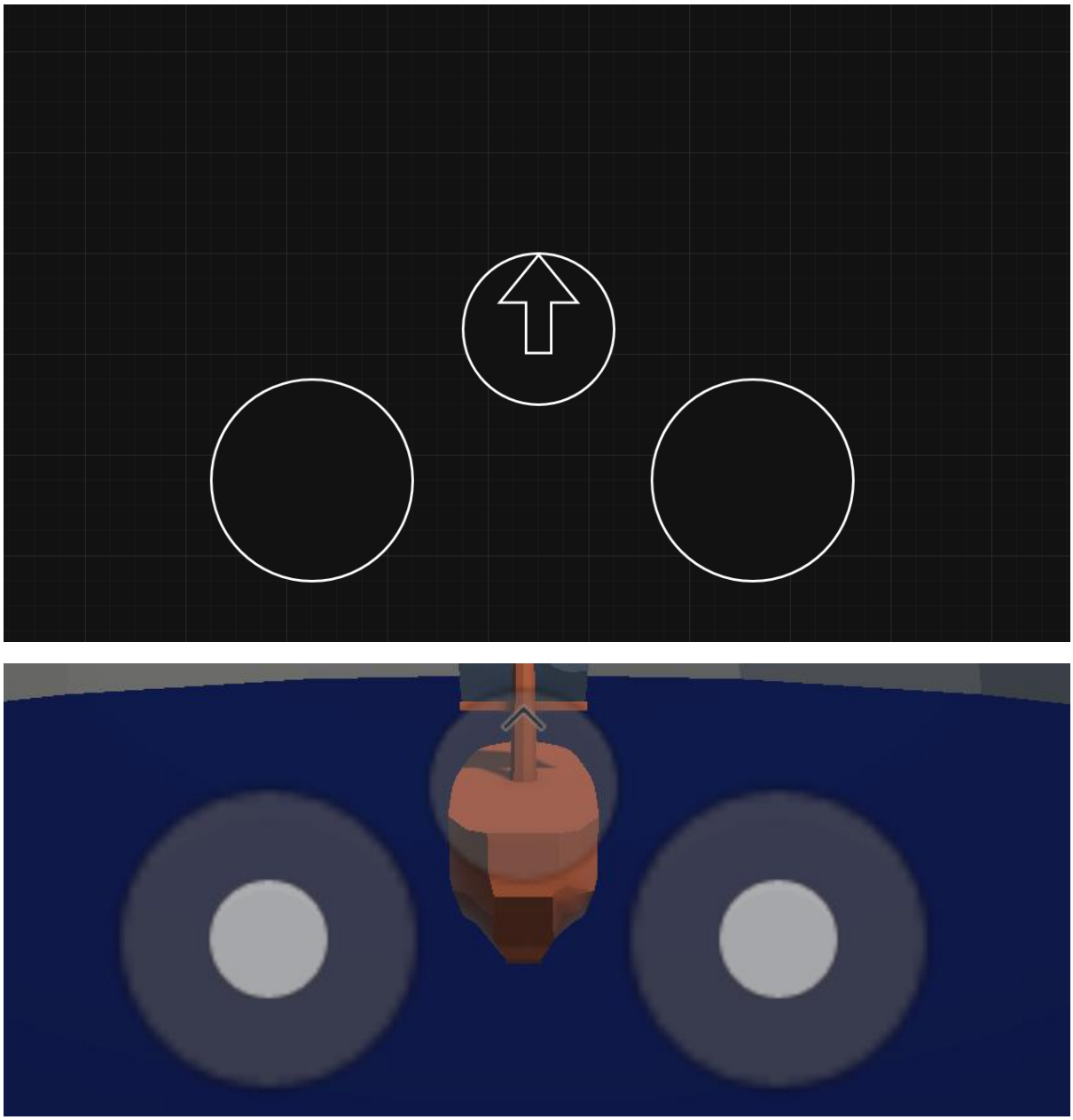

Figure 6 Navigation UI - Direction Influence and "Wind Vane"

A wind vane is an “anemoscope”, which is a device that is made to show the direction of wind (Wikipedia Contributors, 2025).

Figure 7 Wind vane showing easterly wind, By Nevit Dilmen - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1438620

The next element created was the song selection wheel, which is used to switch between tracks or turn off your music. The individual buttons are activated when you collect a piece of music from a buoy, and each are assigned their own unique colour. Next the background was drawn and the symbols for the songs in FireAlpaca, to give them their own unique flair and identity. When the player selects a song, the symbol of that song is shown on the top right of the screen to let the player know the currently selected track

Figure 9 Current selected track

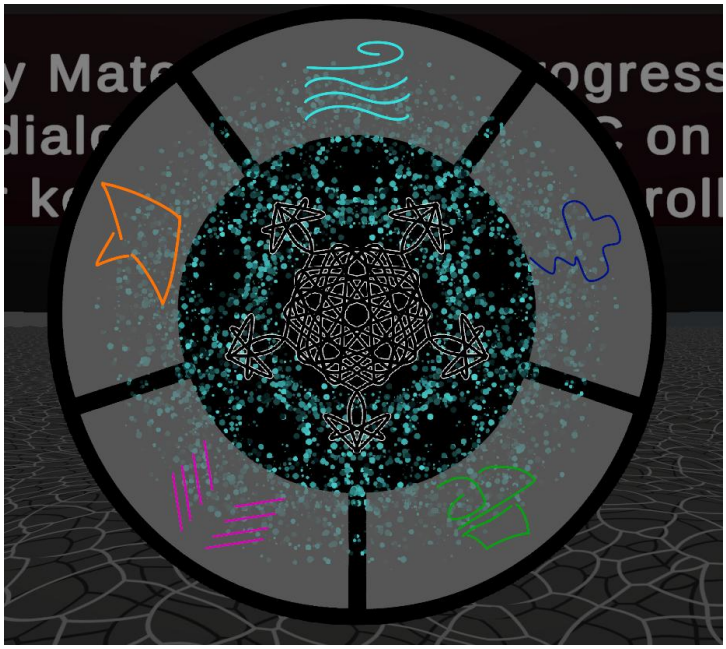

Figure 8 Song selection wheel - all songs disabled

A simple dialogue box was also created to act as a backdrop for dialogue text, increasing readability.

Figure 10 Dialogue box with dialogue text

The game also features a simple main menu and pause menu. These are functional menus designed to be as quick and easy as possible, as they shouldn't distract from the main game experience.

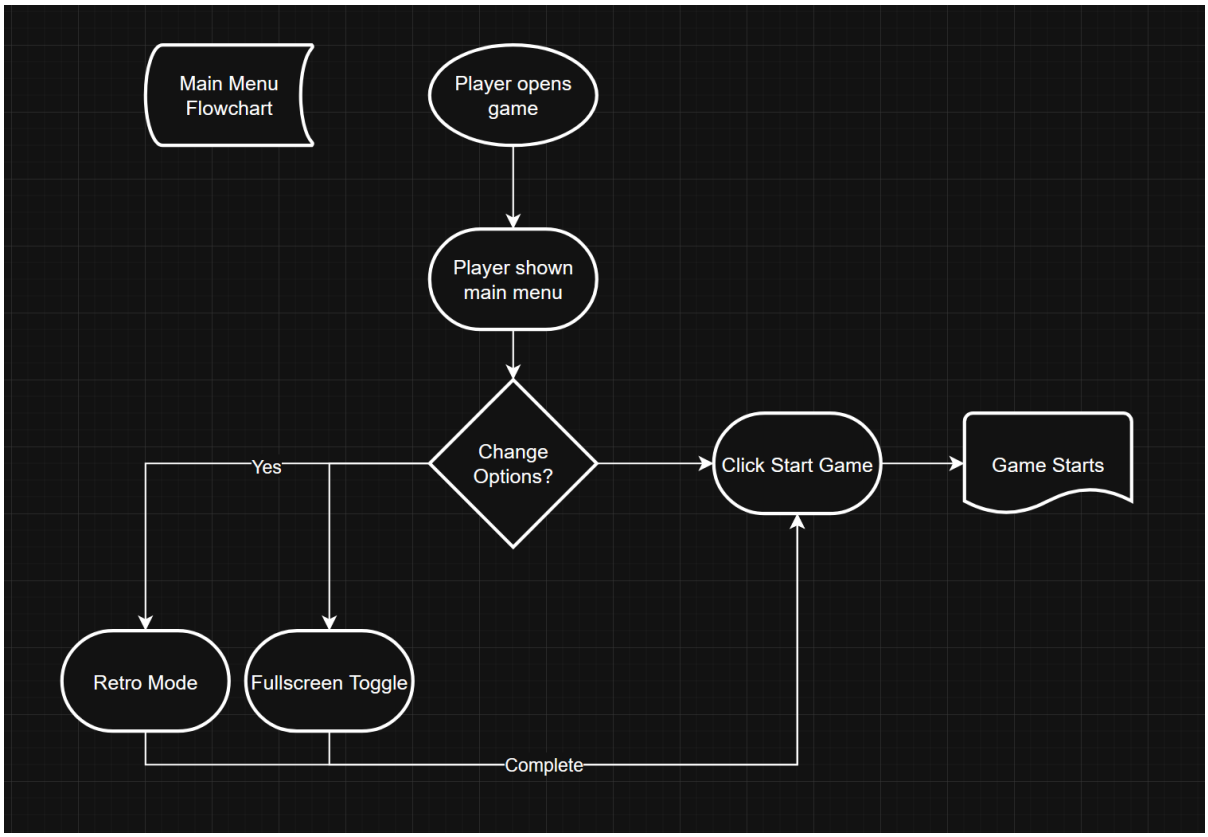

Figure 11 Main menu flowchart

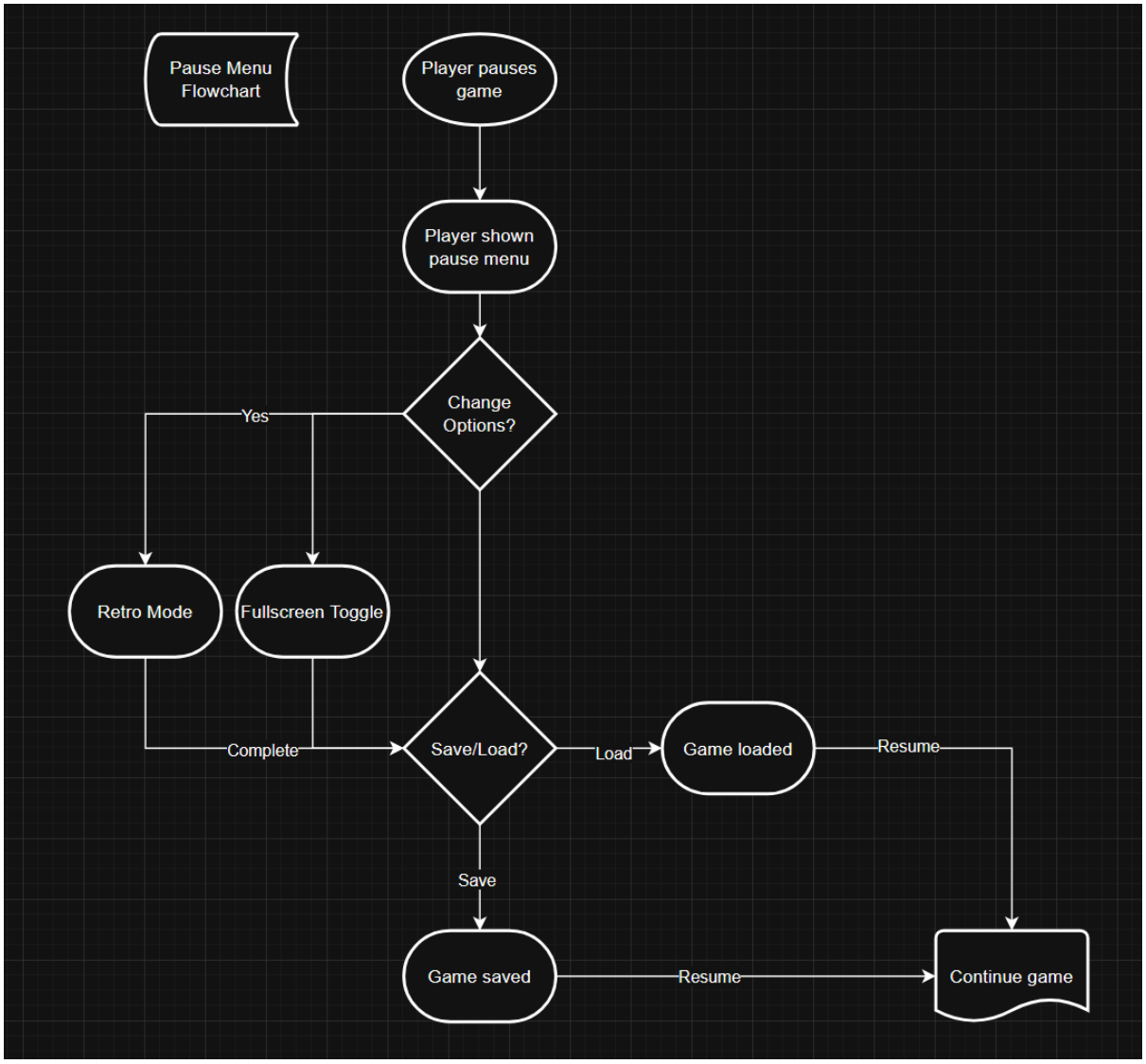

Figure 12 Pause menu flowchart

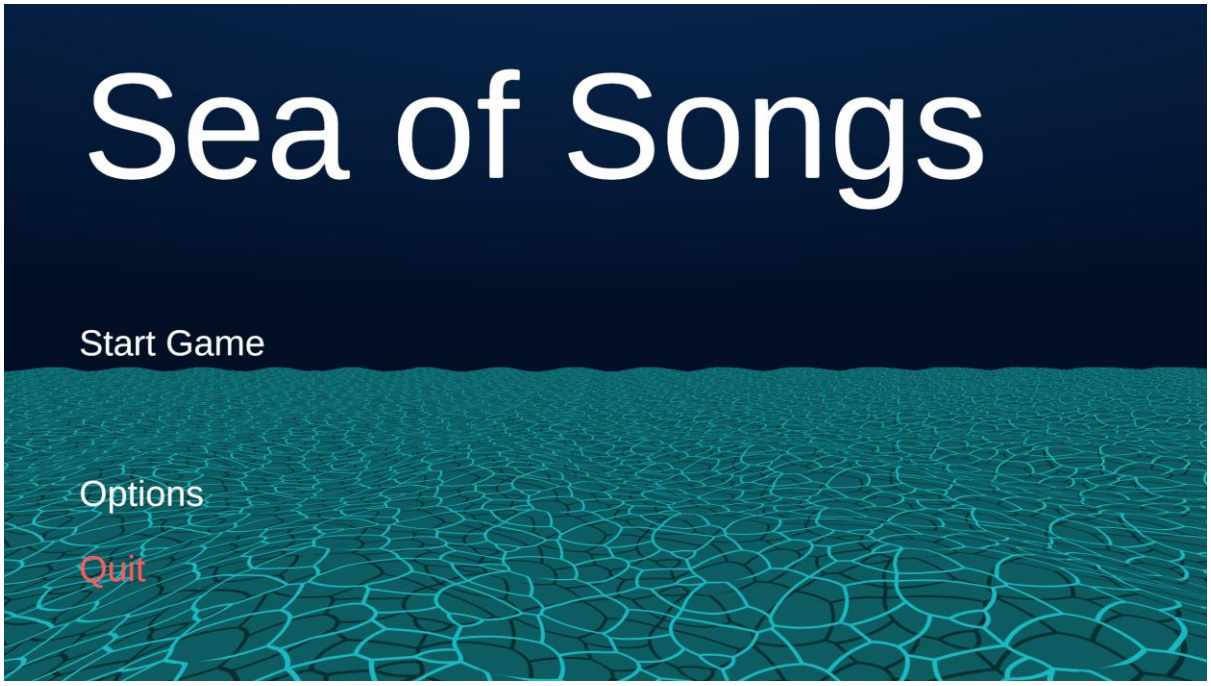

Figure 13 Main menu in-game

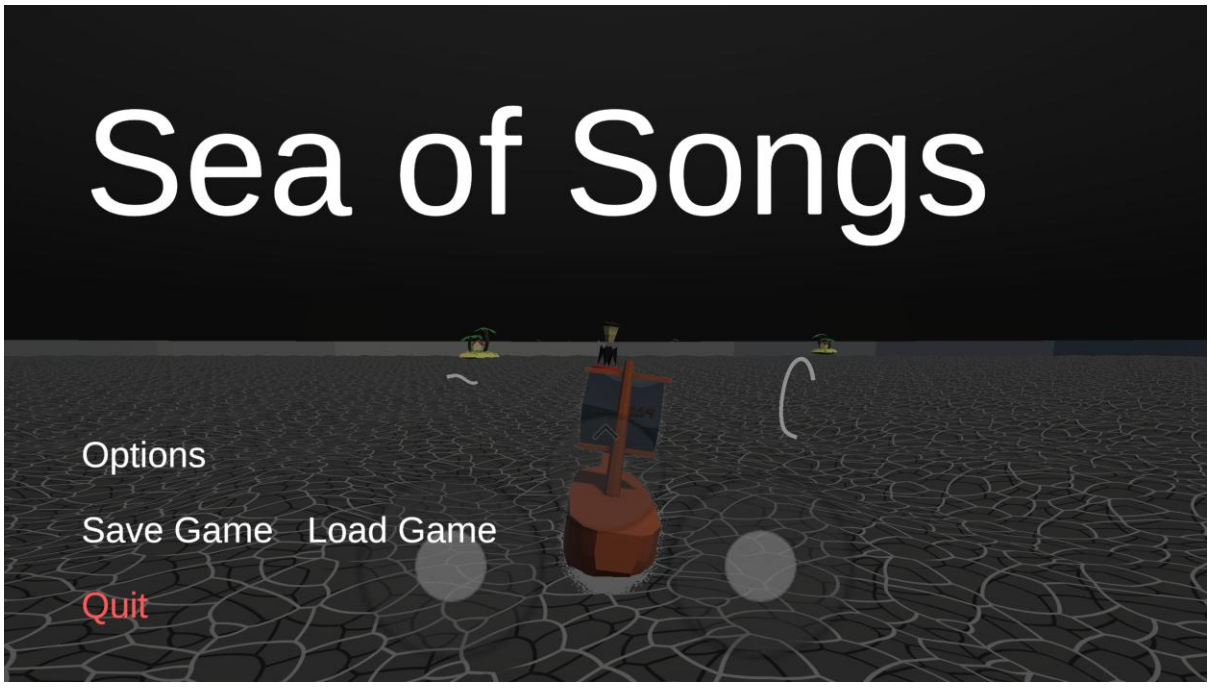

Figure 14 Pause menu in-game

4.1.6 Art and Audio Direction

The art direction of the game follows a low-poly aesthetic with simple graphics, as not only is it an effective strong art direction popular in indie game development, but it requires a lot less work than realism for example. Because the assets and visuals are low-poly and simple, they are easy and quick to make, which allows for a seamless workflow for creating and importing assets.

Audio was important for the game given that music is a key focal point of the experience. For the music, 5 tracks were created, composing of 3 to 4 instrumental layers, and each had their own

soundscape identity. The 5 songs created are: Ocean Waves, a low tempo chill groove for the tutorial; Deep Sea Rock, a high tempo metal loop; Flying Dutchman, a synth heavy spooky track; Thousand Pound Heartbeat, an energetic drum and bass track and Chase, a light hearted track which uses strings and acoustics to move along. Each song has its own associated colour which changes how the environment looks in-game.

4.1.7 Technical Design

The game was made with the Unity game engine. This decision was justified by the game engine being the best option for this task, as explained in the requirements chapter. The game was designed to be made reusing as much as possible, and decomposing bigger problems into smaller ones. Thus, there was a strong focus on creating prefabs and custom assets for the game. An example for this is the entire player system is a prefab, which contains prefabs for all the managers for things like audio and inventory, a prefab for the actual player model itself, and more. Custom asset menus were created for quick editing of assets in the Unity inspector. A dynamic dialogue system was made this way so that the dialogue can be triggered effectively and managed within Unity easily. These concepts will be expanded upon in the implementation chapter.

4.1.8 Design Decisions

When making a game, the designer must make thousands of small decisions that affect the overall product of their work. Each decision refines the idea further and attempts to create an enjoyable gameplay experience that aligns with the creator's vision. The game features unorthodox controls as an interesting control layout for controlling the boat was desired, as moving a boat around is different to moving a person around with all the physics and such involved. There was a desire to make something that felt unfamiliar but not uncomfortable, and thankfully through user testing this proved to be a good balance between these two.

4.2 Conclusion

This chapter outline the design intentions and choices made for the game. The design and implementation stages for this project happened at the same time, so this chapter is strongly linked to the implementation chapter.
Having design and implementation happen at the same time was useful as it led to an agile and volatile development cycle where features and UI could be tweaked on the fly.

5 Implementation

5.1 Introduction

This chapter describes the implementation for the application. This includes the development of the application, and the work conducted in each sprint. It contains extensive technical explanation and code snippets.

Please note all C# scripts have been attached in the appendix to avoid filling this chapter with images of code.

5.2 Development environment

Visual Studio Code was the IDE used to create the C# scripts for this project. VS code is an IDE developed by Microsoft, which has many useful features for developing games with Unity such as extensions and integrated GitHub version control support. Three extensions that were particularly useful in the development of this project were the official Microsoft *C#* and *Unity* extensions, as well as *Unity Code Snippets* by Kleber Silva. These extensions ensured a smooth workflow, as often with

Unity development you are tabbing in and out of Unity and Code constantly. Thus, it was essential that iteration and debugging was able to be as quick and easy as possible.

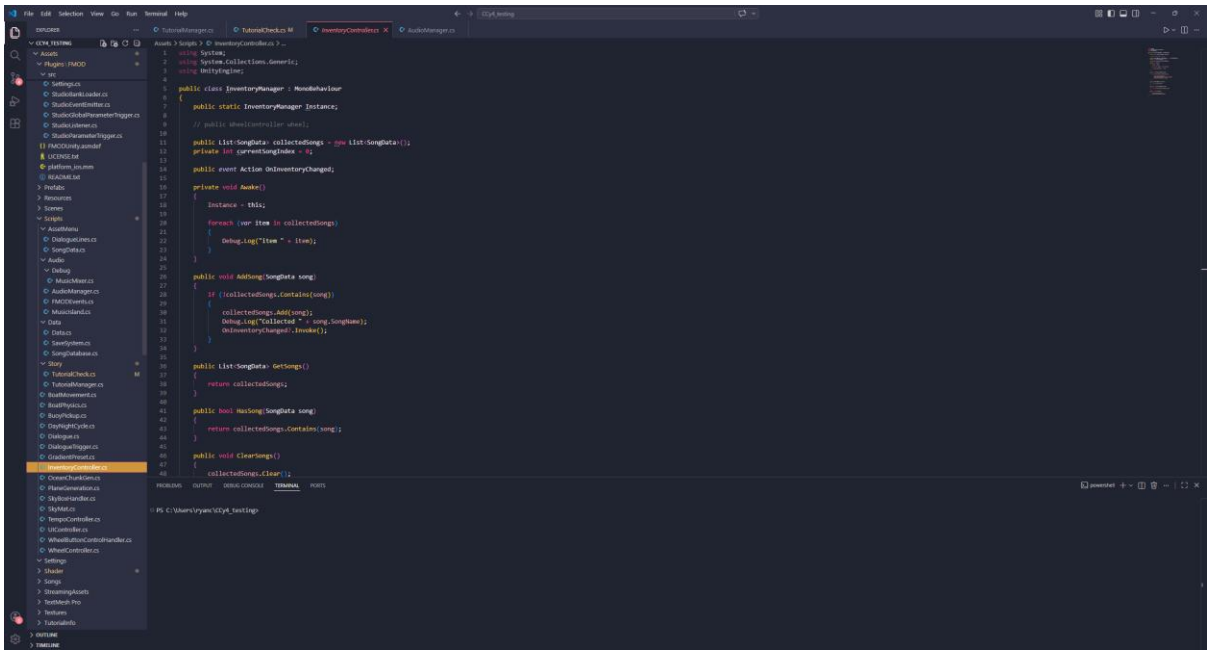

Figure 15 Screenshot of development environment in Visual Studio Code

In terms of version control, GitHub and particularly GitHub desktop were used. GitHub desktop was chosen over using the GitHub CLI as it's what was used for game development previously. GitHub desktop is very intuitive to use, and aside from creating a couple of branches from files corrupting and having to revert to early commits, 95% of the time it was used just to write commits and push to GitHub.

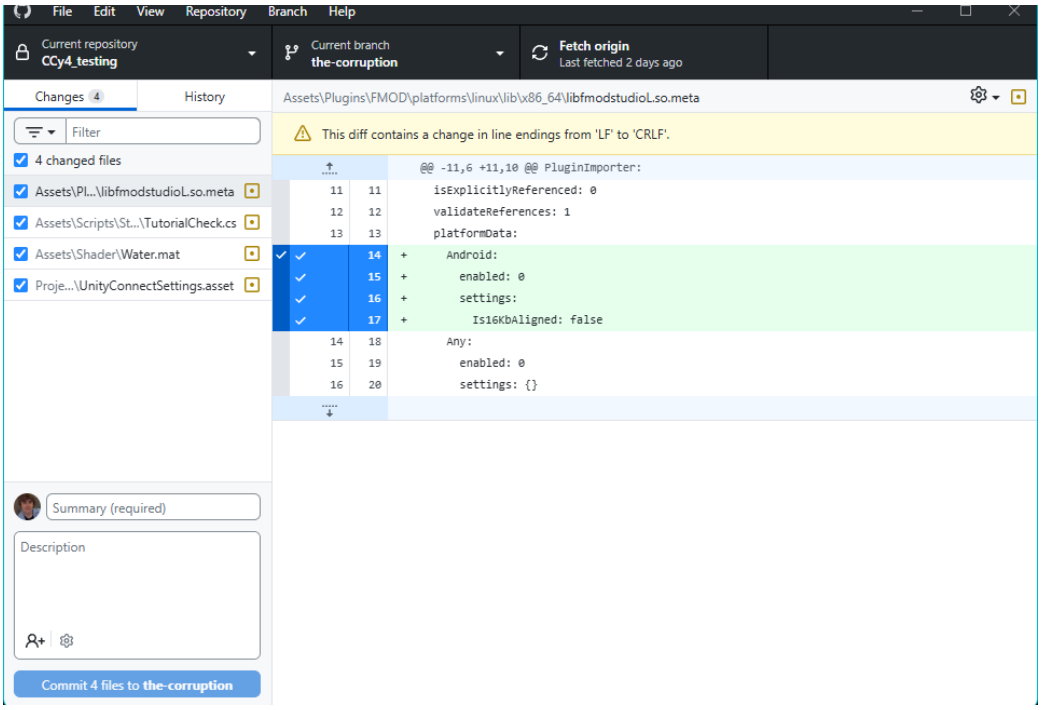

Figure 16 GitHub Desktop

5.2.1 Core Systems

This section describes various important systems that the game uses, such as player movement, physics handling, ocean simulation, shaders, and audio handling.

At a very high level there is a prefab called root which contains many empty game objects for things such as the player system, the canvas in which the UI is a skybox handler, and different managers for things such as audio, UI, inventory. Inside the player system, there is a plane for which the ocean is simulated on and the player object.

The plane that is used for the ocean is parented to the player system, which is what moves around when the player moves. When the game starts, the ocean script generates a 2D plane that contains many vertices. These are manipulated on the GPU to create the ocean waves simulation though a custom shader. This plane is generated through a script that specifically generates a plane that contains a specified number of vertices. To briefly describe ocean simulation, it's important that the visuals of the ocean are simulated on the GPU and the physics of the ocean is simulated on the CPU. The reason why is because if the game's code physically moved each vertex of the ocean on the CPU, that would result in a lot of slowdowns as it is way too much for the CPU to handle. Because the GPU is able to handle many mathematical equations at once, it's great for the purpose of manipulating a load of vertices at once. To accomplish this simulation of the ocean, a custom shader was created that not only handles the texture and material of the ocean but also handles the movement of the ocean.

This shader is divided into two parts. One handles the visual material (fragment shader), and the other handles the position of the vertices (vertex shader). The vertex shader output is fed to the fragment shader to draw pixels to the screen. For handling the vertices, the shader takes the boat position into account and creates two different sine waves based on the boat position. One sine wave is positive and the other is negative. This creates an interesting, realistic ocean wave pattern.

The two sine waves are each then multiplied by a unique choppiness variable, one for the X&Z directions, the other for just X. The two of them are then added to create one sin wave. This singular sine wave is then applied to the Y of a Vector3 variable, which is then added to the negative of the object space of the boat object. This makes it so that the vertices' position variable moves backwards when the boat moves forwards even though the vertices of the plane themselves don't move, giving the illusion that you are actually physically moving along the ocean when in reality the plane stays in place under the parent *Player* object. This makes it so that we can infinitely simulate the ocean for very cheap.

The other part of this shader which handles the visual material is quite basic. It takes 2 separate images that are black and white that have a cartoony ocean-like pattern on them and shifts them slightly while multiplying them with a water colour, light foam color and dark foam colour. This by default is blue, but because the water colour is a variable and light foam and dark foam are affected by the water colour, this can be used to create custom colours for the water depending on the environment. This is how when you switch songs that the water can turn colors such as orange, green, purple and the others.

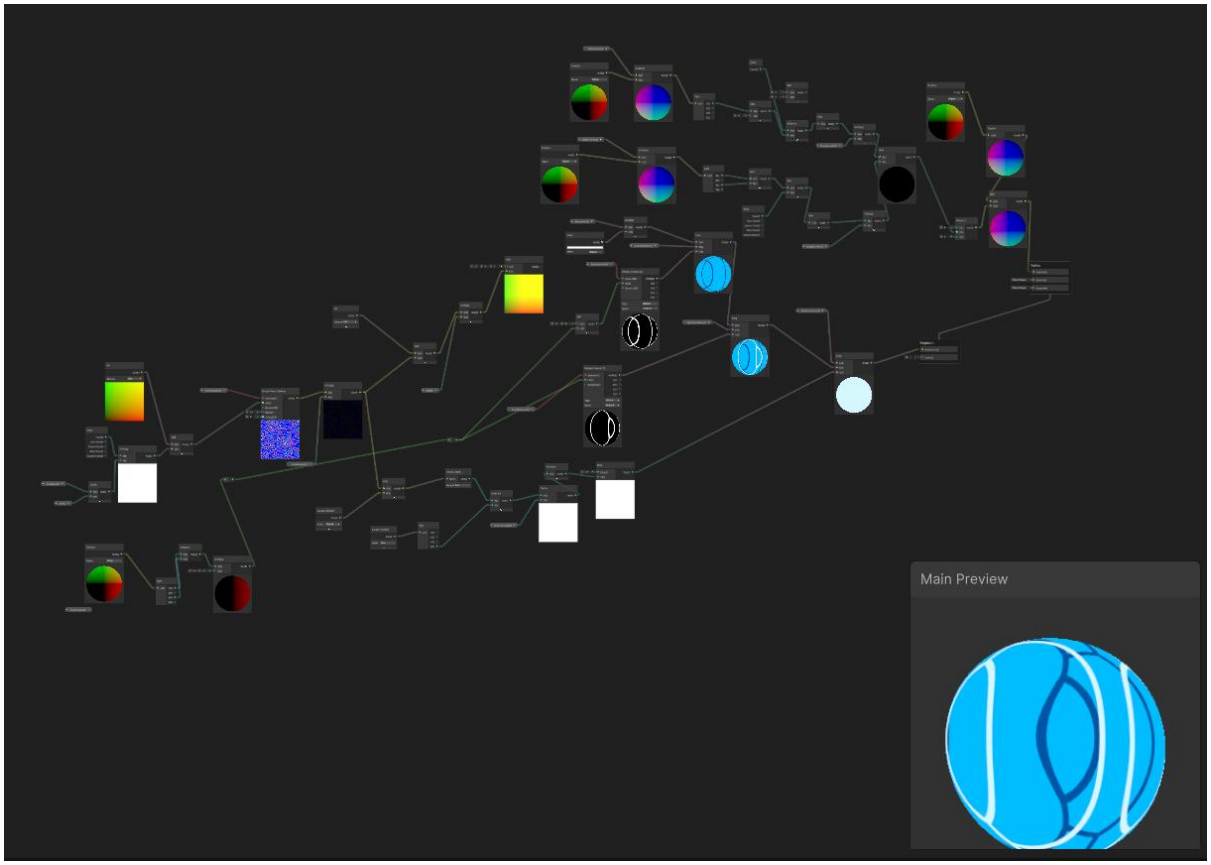

Figure 17 The entire water shader in one image

The second component of the player system is the visual of the player. This contains the physical object of the player, the camera which is attached to the player, and a render camera which is used for an optional setting to use retro style graphics. Inside the player object there is the boat visual. And then there's also an empty game object called floaters. This contains 3 different empty game objects that contain a script for simulating the buoyancy of a boat. This is how when the player is in the world that they can see the boat is moving or floating along the water, like how a boat would in real life. The waves that the floaters follow are the same formula as the one found in the shader, which is how the boat looks like it is floating on the water. The floaters script takes a rigidbody, which is attached to the boat object in this case, and then you can set things such as the depth before submerged, the volume of the floater. The water drags, water angular drag, and then also you must assign the actual water plane which is the one that is attached to the player system. The player movement system consists of a movement script and then a *character controller* Unity component, which isn't used for controlling the character, but for handling collisions. With the player movement script, you can set the jump force, the boat rigidbody, the move speed, acceleration and turn speed. There are also wind particles attached to the player system, which were created using the Unity particle animator.

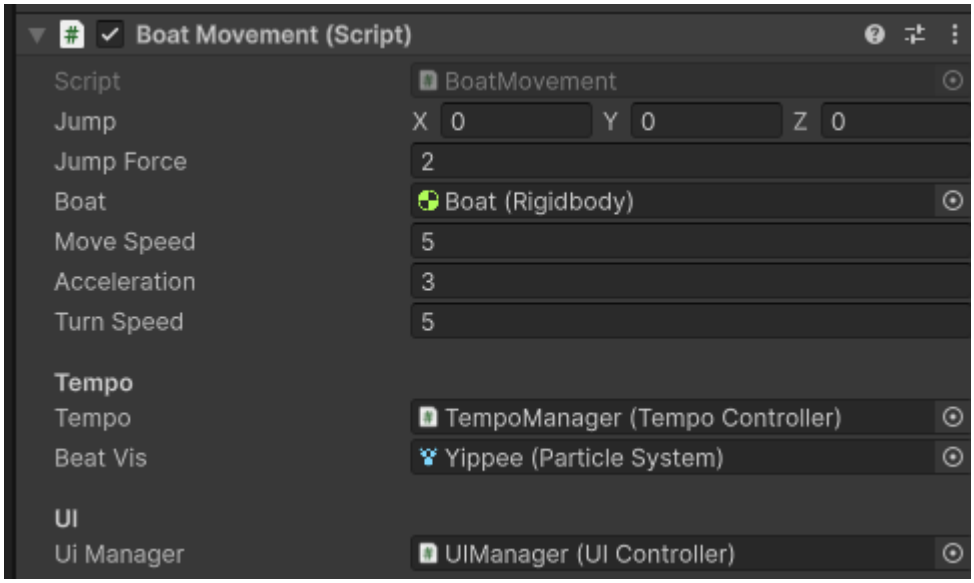

Figure 18 Boat Movement MonoBehaviour script in Unity inspector

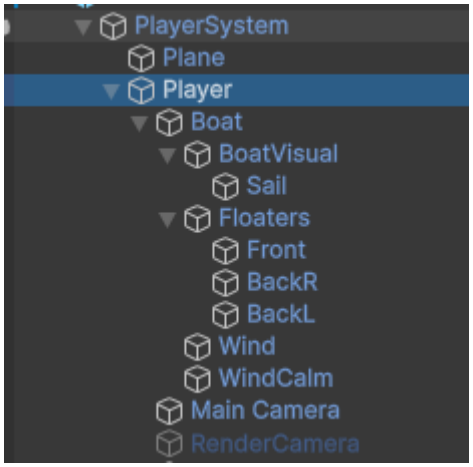

Figure 19 Player object hierarchy

The render camera used in the player system is used for a retro style visual effect. The player can choose between using the regular main camera or the render camera for this retro effect. The retro effect is achieved by using a render texture that is put on the top of the canvas (behind all the UI), and the output of the render camera is sent to this render texture. The render texture is rendered at a smaller resolution which creates a retro style pixelated effect.

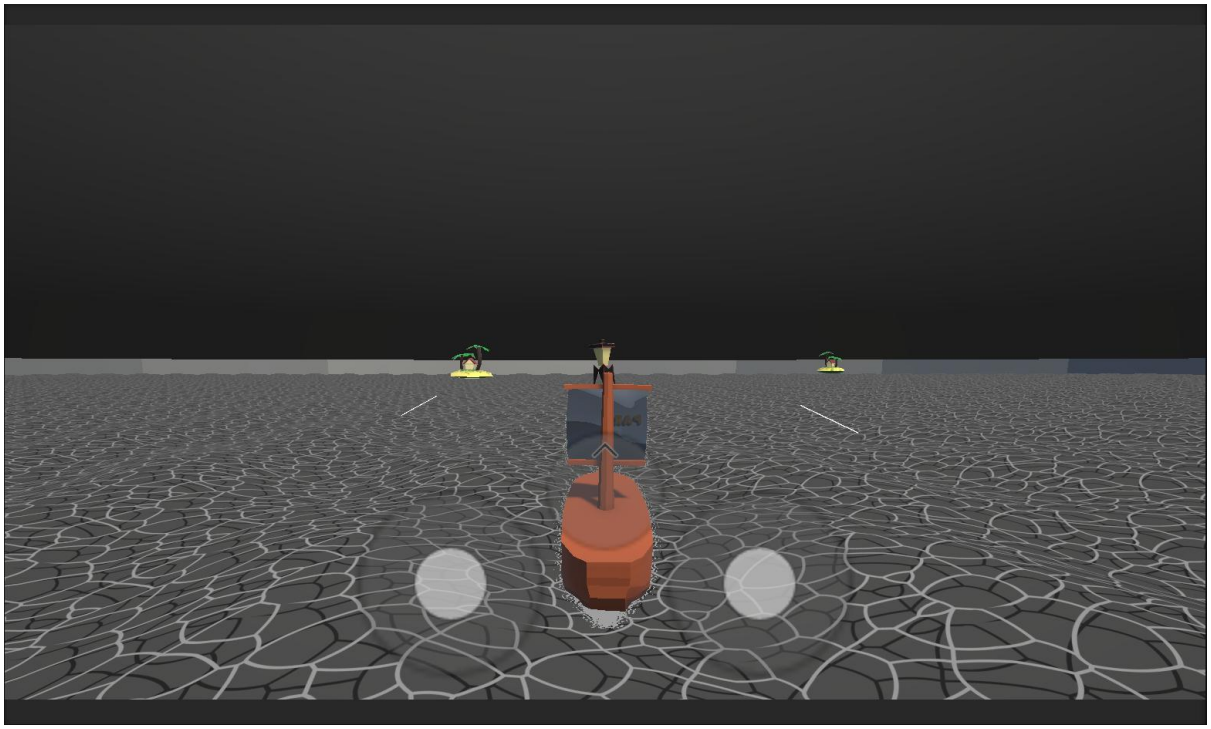

Figure 20 Tutorial are with normal settings

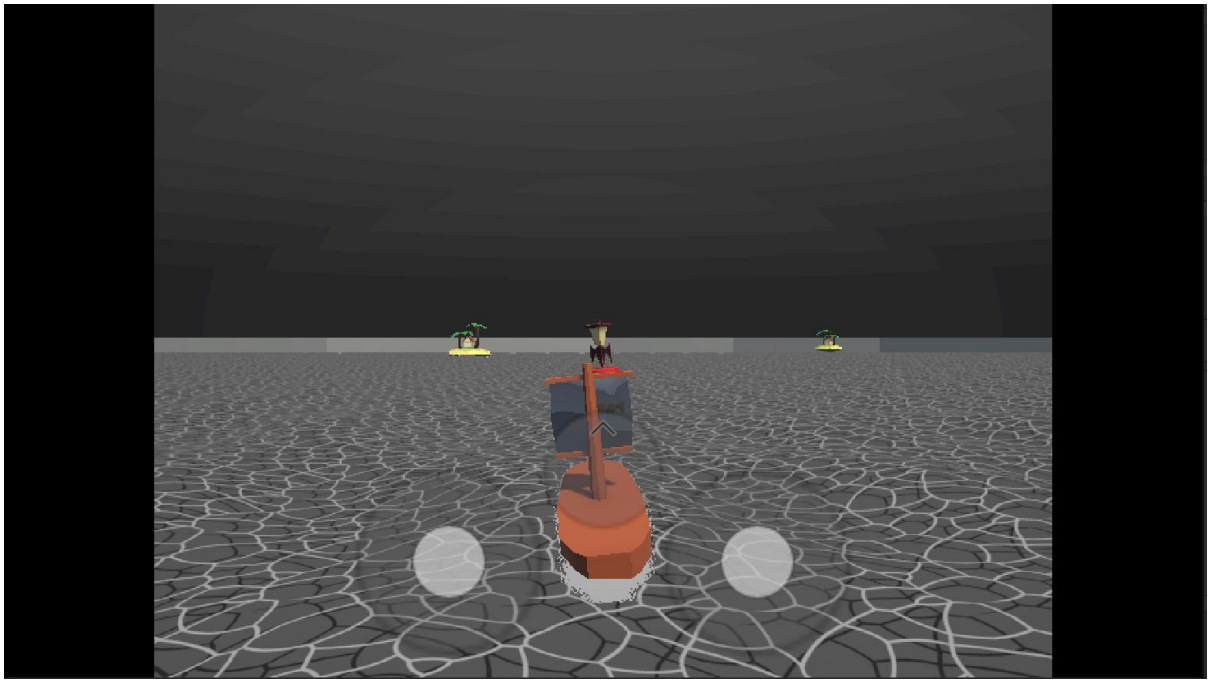

Figure 21 Tutorial area in "retro mode"

The sky box of the world is handled by an empty game object that contains a skybox handler script. This handles the material and the time of day in the world. The time of day by default is 2 minutes which can be changed in the inspector. There are two arrays attached to this component which are Sky gradients and Water gradients. These are populated with custom gradient objects that were created specifically for this game. When the player selects at different music track in the game, the current index of the skybox handler changes depending on the index of the music track selected of the audio manager singleton. The elements of the sky gradients of water gradients are linked to the song selection, and so the sky material changes to the appropriate gradient preset.

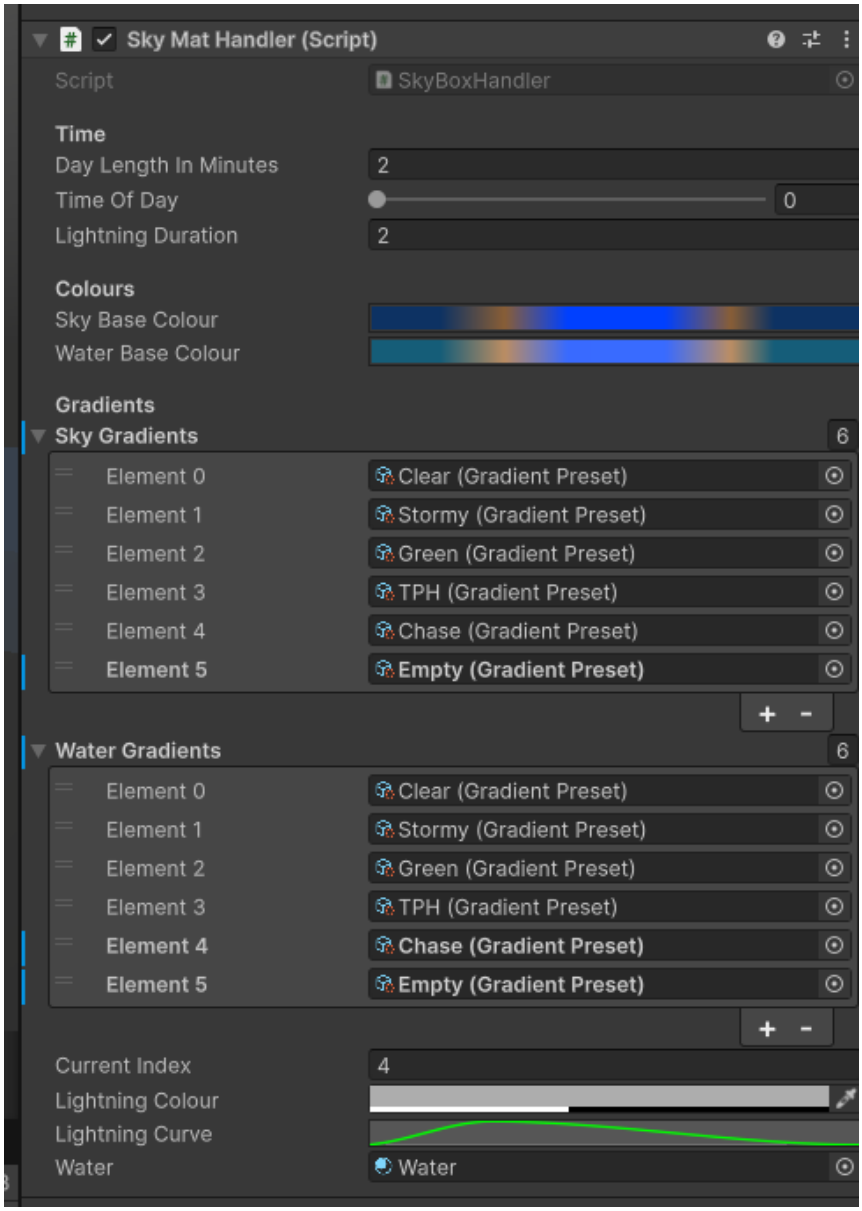

Figure 22 Skybox Handler in inspector

The canvas element contains various objects such as the background for the retro rendering, the movement UI, the dialogue box, the song selection, and the selected track UI. The movement away just shows the current direction in which you're moving and which directions you push the left and right stick or your keyboard depending on which way you're playing. The dialogue box is just a static box that is turned on and off depending on if there is dialogue currently in action. The text of the box is also changed by a dynamic dialogue system when needed.

There are three parts to the dialogue system. There is a dialogue handler which is a Singleton. Dialogue trigger which can be set to different components and then a dialogue object which is made in the asset menu. The dialogue object was a custom asset that was created for this game so that the developer could easily create dialogue when needed. The dialogue object is an array of strings. Dialogue objects are attached to dialogue trigger objects. These are invisible colliders that when triggered will trigger a dialogue event with the attached dialogue. Two are used in the tutorial.

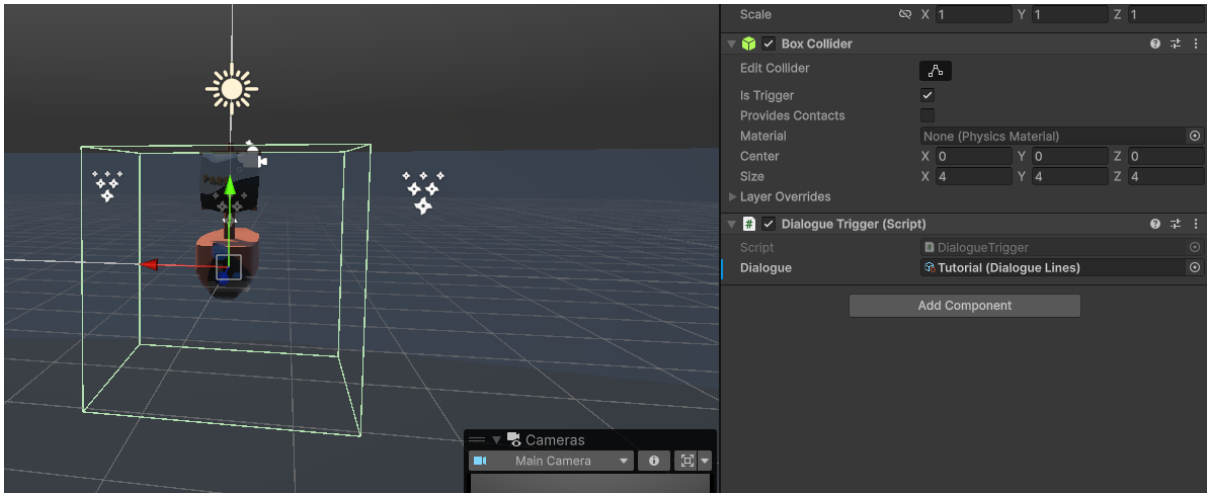

Figure 23 Dialogue trigger in inspector

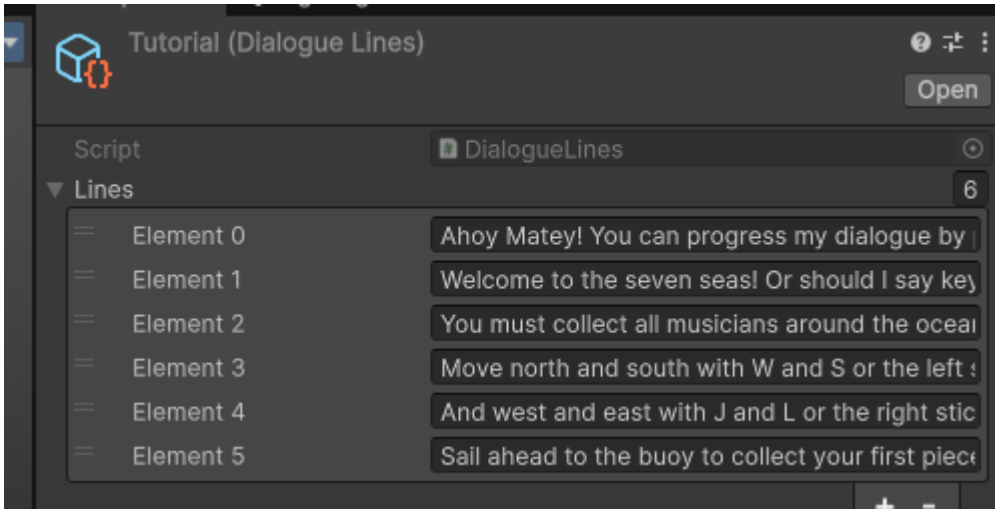

Figure 24 Dialogue scriptable object in inspector

The selection wheel is the final component of the UI. It's a drawn circle with five buttons attached to it. Each button is assigned to a different song and when the player collects the song in the game, the button becomes interactable. When the player clicks on the song, it changes the music in game to that song. The animations of the wheel were created in the Unity animation editor.

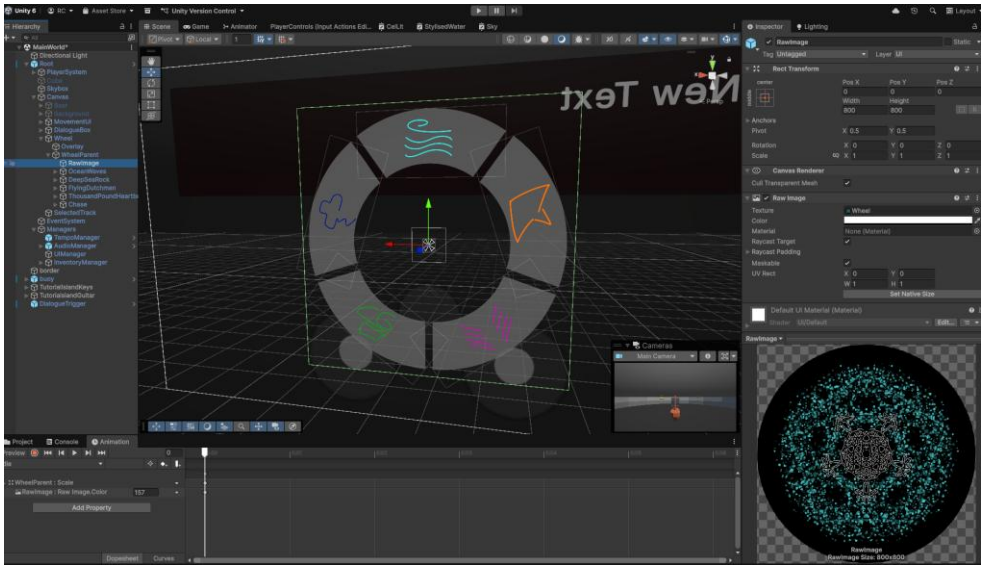

Figure 25 Selection wheel in scene view

The final component of the root object is the managers. This contains objects for managing tempo, audio, UI and inventory. The tempo manager is used to manage the tempo of the current song, as originally there was a mechanic to play along with the beat of the song. When the player jumps in time with the song, sparkles come off the boat. Next is the audio manager which contains another object under it for FMOD events specifically. The audio manager is where all the FMOD related code goes. It also is a singleton. The FMOD events object contains just references to the music and UI and sound effects, also acting as a singleton. The reason why singletons are used for these objects is so that the game can access the scripts from anywhere without having to reference the audio manager directly. The audio manager script handles things such as individual instrument volume, setting the tempo, changing the skybox handler, as well as handling the current track playing.

The UI manager is used to change the UI for related to movements in the canvas.

The inventory controller is an instance for the songs that the player is collected. The controller has functions for adding a song to their inventory, getting the songs in their current inventory, checking if the player has songs, and clearing their songs. The Inventory Manager object in Unity also contains a save system script. This is for managing the player's save. The inventory manager also uses a custom SongData asset. The inventory is a C# list of these SongData assets.

The save system is very simple. Upon awake, the script creates a singleton. It also sets the save data path as a variable. There is a function to save the game which saves the current inventory of the player as well as their position in the world into a JSON file. The function to load the game reads this file and applies the inventory and transformation to the player in-game.

Figure 26 Managers hierarchy

5.2.2 Asset Integration

The assets of the game are organised in relevant folders depending on what the asset type is. Examples include animations, dialogue, materials, models, scenes, scripts, and more. Some of these folders are further organised through folders inside them, for example scripts are broken down into smaller folders depending on where the script is for.

As previously mentioned, some custom asset types were created such as dialogue lines, SongData and gradient presets. This means that you can right click in the Unity browser and create a new asset of those types specifically. This ensured a smooth workflow and allowed the developer to reuse some assets where possible to save on time and storage.

```
using UnityEngine;

[CreateAssetMenu(fileName = "SongData", menuName = "Songs/Song Item")]
Unity Script
public class SongData : ScriptableObject
{
    Unity Serialized Field
    public string SongName;
    Unity Serialized Field
    public int TrackID;

    Unity Serialized Field
    public AudioManager.Instrument[] instruments;
}
```

Figure 27 SongData scriptable object code

As this project has extensive FMOD integration, it's important to note that FMOD audio files are not directly within the Unity asset browser but are instead built with the FMOD Studio programme. Everything related to audio is edited within FMOD and then once that work is done you click build in FMOD which the build is linked in Unity and will refresh upon a new build.

When a song was created using Ableton, each individual track of the song was exported as an individual **WAV** file. Each track would then be put together in FMOD under one song so that the instruments could be manipulated individually.

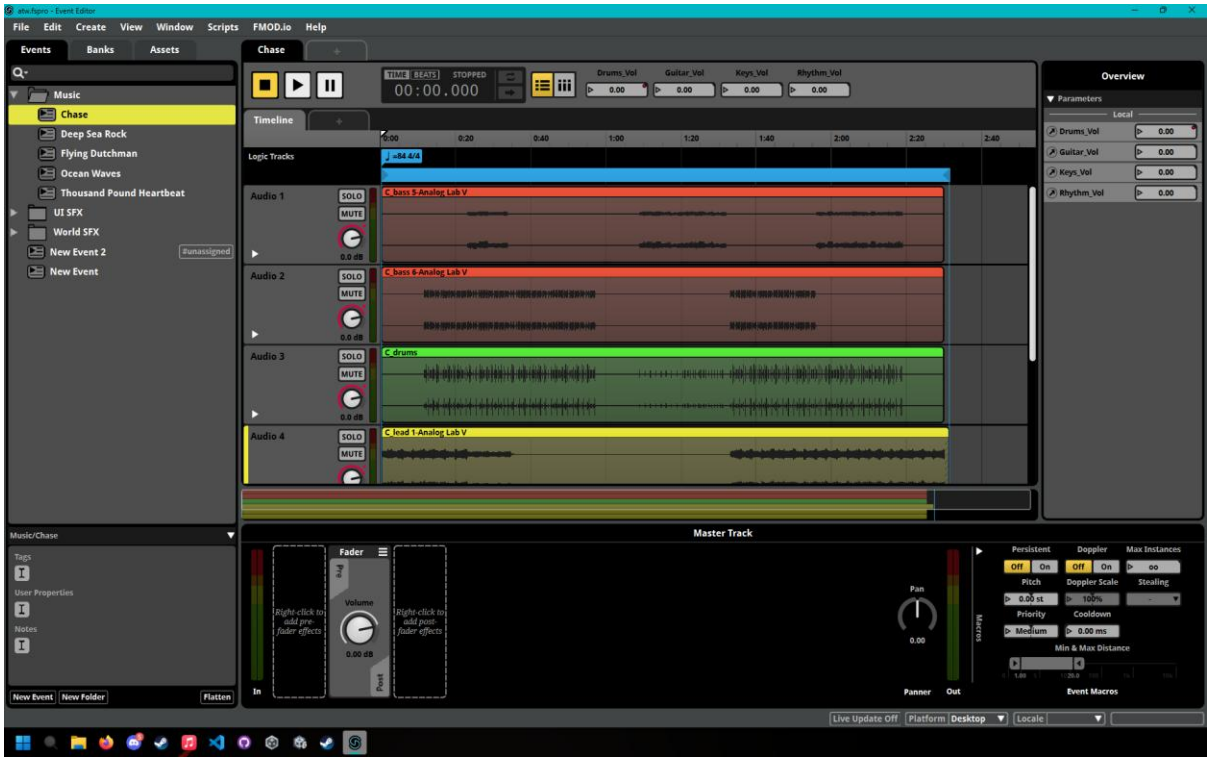

Figure 28 FMOD studio

Some interesting assets of note are the models which are imported into the game. The models and their animations were created in Blender. They are exported from Blender in the **FBX** format. They are then imported into Unity with the textures baked into the model data. This ensures that when the model is imported it the texture is correctly UV unwrapped and imported properly.

5.2.3 Performance and Optimisation

As the game is supposed to be a simple experience with low-poly graphics, it was important to ensure that good performance and optimization practises were utilised in the development of the game. Singletons were used as much as possible to ensure that there was no repetition of instances of scripts where appropriate. Game objects were also compartmentalised to ensure that components could be reused where possible. To ensure that the game was performing and optimised as well as desired, it was ensured that the game maintained a steady framerate for its performance.

5.2.4 Debugging

Unity's console log by default logs any errors, critical or noncritical. Which was great for debugging purposes. Any problems from code that was behaving unexpected were solved using debug log in Unity. This was the quickest solution to debugging as it was very easy to implement and remove once debugging was complete. Most of the debugging process involved searching the Unity 6 documentation or online for niche bugs that stemmed from FMOD interacting with particular parts of the code such as the inventory or save system.

5.2.5 Challenges and Solutions

There were many challenges faced while developing this game. These challenges these challenges may require creative solutions and thinking outside the box. For example, one problem that was faced early on was creating an infinitely large ocean. The first approach taken to creating this was

creating some sort of coordinate system that placed ocean planes depending on where the player was in the world. When they moved away from these plains, they'd be culled and removed from the game. As they move through the world, more would be placed towards the direction that they are moving. After an amount of work was completed on this idea, it proved it was not feasible for the current setup, so a change in approach was needed. The next idea was to create one plane that was attached to the player and then using a shader to make to give the plane and the illusion that it is infinitely large. The plane itself is 200 metres by 200 metres. Using the water shader, each vertex in water plane could be manipulated on an individual level based on its location within the world, and this approach proved to be very effective.

The problem now was that the water would move in the same direction as the player so while the waves were moving appropriately the texture of the water looked strange the solution to this was to get the object space of the player and negate it so that when the player moves in one direction the ocean moves in the other. With the solution the water now looks realistic, and when objects border on the water the waves stay where they should be in relation to the movement of the player.

Another problem that was faced was that the player would sometimes collide with an object and the camera would continue to move with the mesh didn't. Due to the way that the player was set up, the collider on the player was colliding with objects, yet the camera kept moving as it was parented to the object above. The solution to this was to wrap the entire player system in a rigidbody under the character controller, as that was easy to set up with collisions. This now ensured that when the player collided with an object that everything would stay still and not break.

5.3 Sprint 1

5.3.1 Goal

The goal of Sprint One was to set up a development environment in Unity research. Relevant topics for the game such as manipulating shaders with code and general shadergraph knowledge.

5.3.2 Item 1

The first item of this sprint was boat movement.

The first thing completed in this sprint was to implement moving around a small cube. This was to test the movement system on simulated water physics. As the movement system was refined, a simple boat was modelled that could replace the placeholder cube. Because the boat was bigger than the cube, the floater values need to be adjusted appropriately. The boat needed more than one floater as just one was not enough for the boat to balance. To bind the controls to the movement script, the Unity Action Map Editor is used to create actions.

The next thing to work on for this item was getting the boat to rotate the correct direction the player is moving. The boat rotates using a rigidbody move rotation function that has a quaternion slerp using the boat rotation, the target rotation being the resultant vector that the boat is currently moving and the turn speed multiplied by fixed delta time.

5.3.3 Item 2

The next item of this sprint was individual instrument volume control.

As previously mentioned, all audio handling is done through the Audio Manager scripts linked with the FMOD API. FMOD is directly integrated with Unity using the Unity FMOD extension. This lets us call upon the FMOD API directly in the C# audio manager script.

The audio manager's script is a singleton instance, so they can use it anywhere in the game. In the audio manager scripts 2 enums are defined. One for the instruments and the other is for the songs. On awake it sets up the singleton instance. There are various functions in the audio Manager script such as initialise music, set instrument volume, set enum value, set reverb, play music, stop music and track control. During this Sprint, the main concern was setting the instruments' volume levels and initializing music, as at this point only one song had been created. Under the AudioManager Game object in Unity, there is an FMOD Events game object. This game object is where all the songs are referenced. It's also a singleton that can be accessed anywhere if needed.

FMOD parameters are created for each individual instrument that could be used for each song. The code in the audio manager accesses the parameters and sets the audio volume depending on what the game wants. To debug this, visual UI was created to test the audio changing. With this UI the developer was able to set the individual instruments volume level while playing the game.

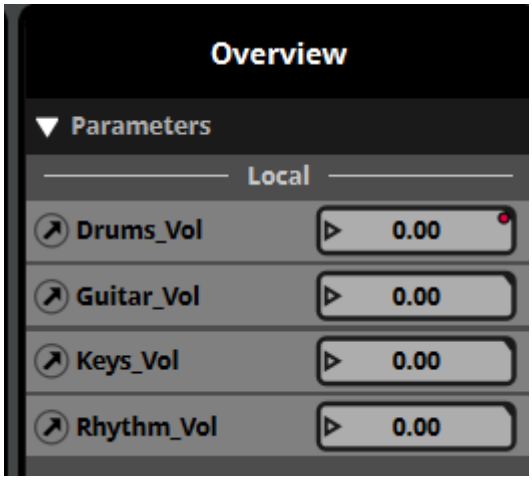

Figure 29 Parameters viewed in FMOD Studio

5.4 Sprint 2

5.4.1 Goal

The goal for Sprint, too, was to research and create visuals that change depending on the music. This included the environment, the skybox and the water.

5.4.2 Item 1 – Skybox Shader

Item 1 of this Sprint was to create a skybox shader that I could use to fully customise the skybox. This skybox shader needed to be accessed through code to change things such as colour and certain effects like lightning. The water shader also needs to change in respect to the lightbox shader. This meant that the colour would synchronise between the skybox and the water.

I started with creating a shader in Unity using the shadergraph. The shader starts with creating a new UV that goes into a swizzle function to mask the Y vector. This creates a simple gradient going into Y axis from white to black. I then remap the gradient vector from -1 to 1, to 0 to 1. This effectively means that we're left with the top half of the gradient vector. I then create a grey to white gradient node and plug it into a sample gradient using the remaps gradient as the Y vector. At

this point you could set the result to the fragment base colour of the shader, resulting in the sky box being grey at the bottom and white at the top. As the shader needs to change colour depending on the current song I I create a base colour variable that is plugged into a multiplier with this gradient that sent them to the fragment. This then means that you could set the base colour to anything you want and it will come out as a nice gradient with the colour on top, and a slightly darker version of the colour at the bottom. Finally, right before I put it into the fragment of the shader, I create a lightning variable for a lightning coroutine used later.

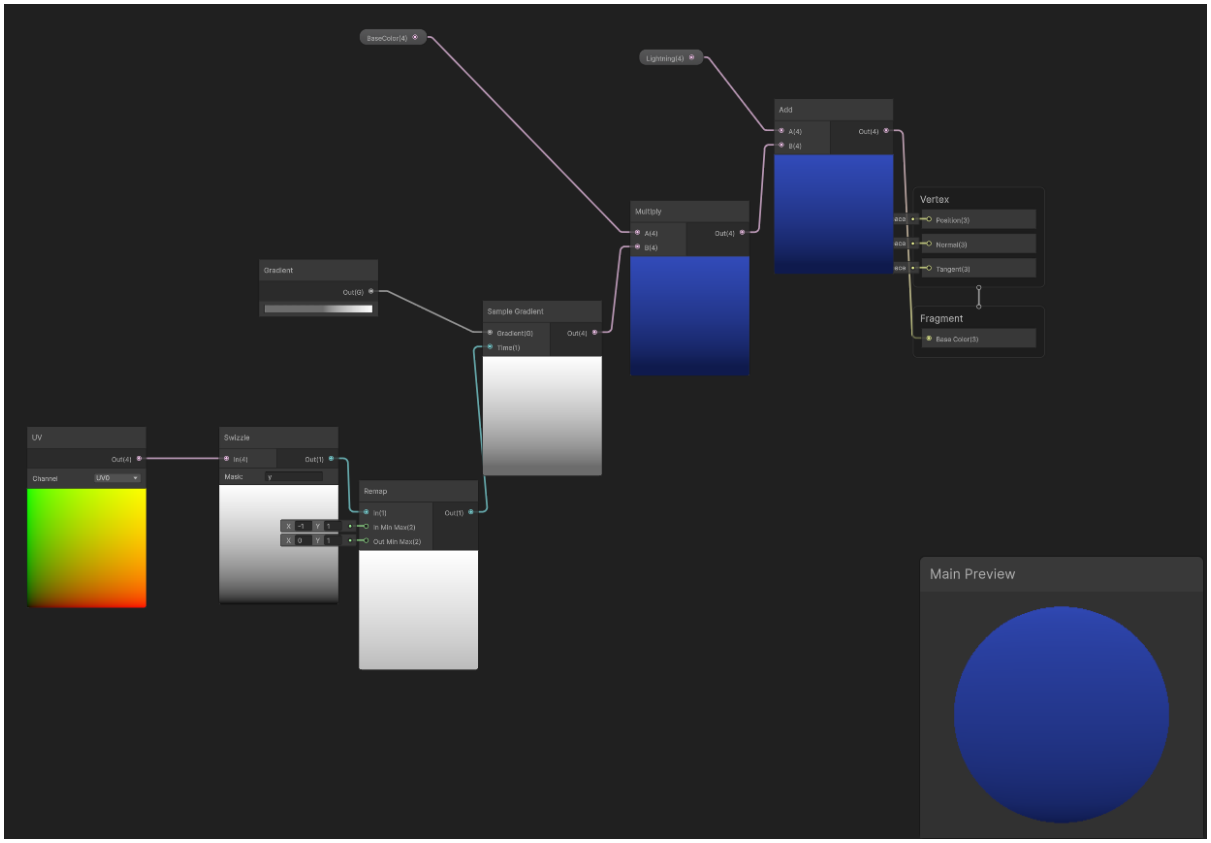

Figure 30 Skybox material in shadergraph

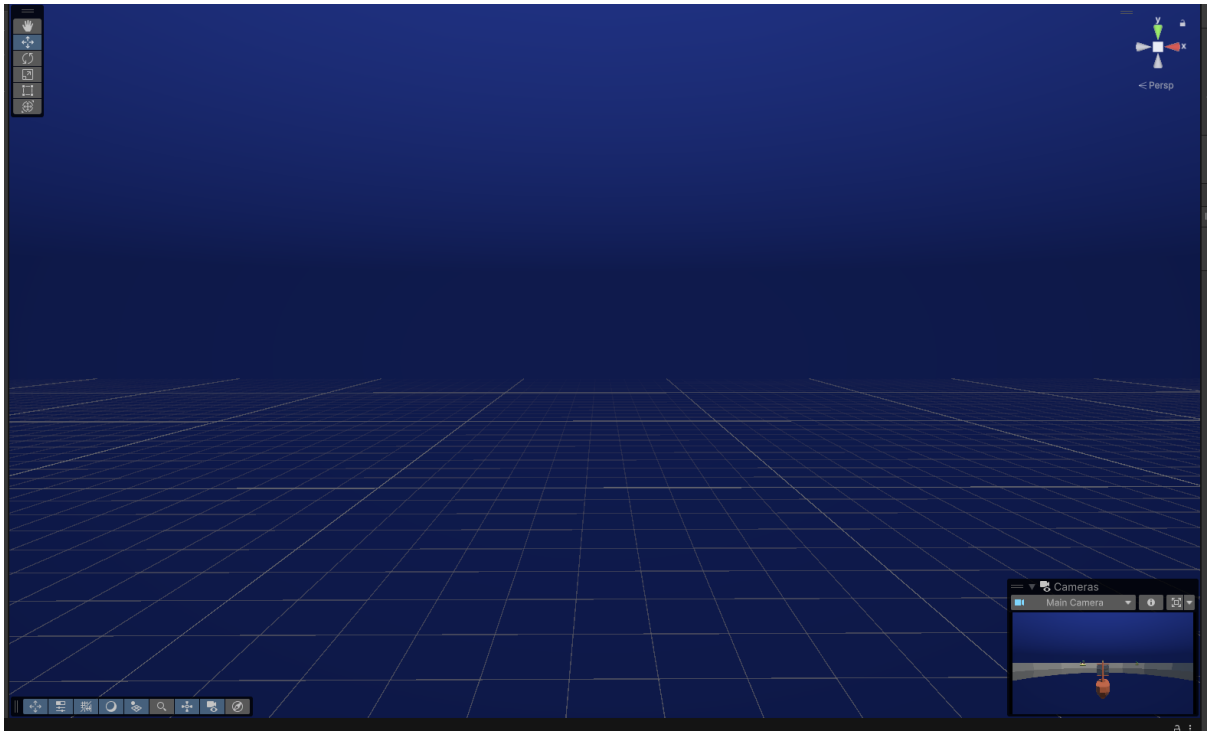

Figure 31 Skybox in scene view

After the shader was set up, a skybox handler C# script was created. This sky box handler handles things such as colour changes, lightning effects, and the time of day. The script has a lot of references, but it's quite simple. At the start it sets the date duration and gets the rendering sky box. It then starts a coroutine for lightning if lightning is needed. It will then update the time of day based on the amount of time passed, as well as update shaders. The way it updates the shaders is by setting the sky box base colour, the water base colour, the water light foam colour in the water dark foam colour. This means that as the day progresses, the sky box will change depending on the gradient selected in the array of gradients. The colour sampled from the gradient moves from left to right as the day progresses. The script also handles a lightning coroutine that I referred to earlier talking about the shader. Certain songs have a lightning boolean. While this boolean is true, the lightning coroutine will start, waiting a random number of seconds before doing a random number of lightning strikes and starting again.

5.4.3 Item 2 – Music and Skybox Interaction

The next item of this print was to get the music to change the sky box.

Changing the skybox colour is handled by the audio manager track control function. Inside this function is a switch statement. The switch uses the track in AM established earlier in the audio manager. When the selected track changes, it'll switch the music to the selected song, change the skybox index which changes the sky box colour and the water colour, and then change the tempo.

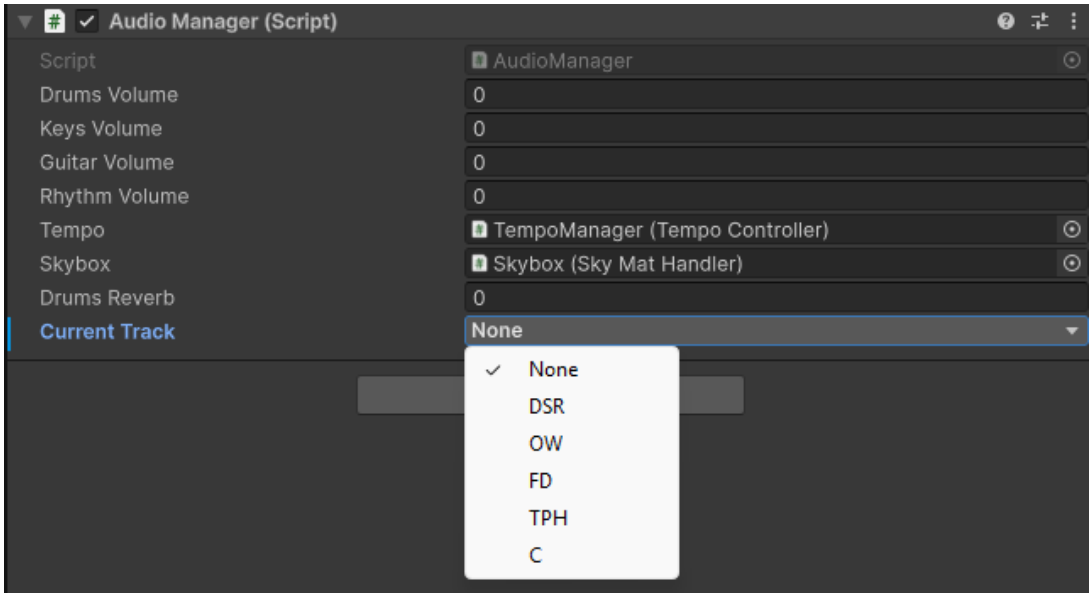

Figure 32 Track selection for audio manager in inspector

```
public void TrackControl(Track track)
{
    switch (track)
    {
        case Track.None:
            StopMusic();
            skybox.currentIndex = 5;
            tempo.ResetTempo(92f);
            break;

        case Track.DSR:
            PlayMusic(FMODEvents.instance.rock);
            skybox.currentIndex = 1;
            tempo.ResetTempo(160f);
            break;

        case Track.OW:
            PlayMusic(FMODEvents.instance.chill);
            skybox.currentIndex = 0;
            tempo.ResetTempo(92f);
            break;

        case Track.FD:
            PlayMusic(FMODEvents.instance.alien);
            skybox.currentIndex = 2;
            tempo.ResetTempo(90f);
            break;

        case Track.TPH:
            PlayMusic(FMODEvents.instance.drive);
```

Figure 33 Track control switch case

5.5 Sprint 3

5.5.1 Goal

The goal of this sprint was to create a tempo manager for the game.

5.5.2 Item 1 – Tempo Manager

The first item of this sprint was a tempo manager script.

The idea with the tempo manager script was to add some interactivity to the music. The player would have to input a button in sync with the beat of the current music to keep the music in time and tune though this mechanic wasn't used in the final product. That this mechanic didn't make it to the final product to the game, there's still an input for the tempo manager. Particles will come off the boat if you press the button in time with the music. The way it worked was by the way it worked was by getting the current music instance from the audio manager. And using maths to get the song position in seconds and create a beat distance float depending on the BPM of the song. While the song was on beat, a short window is created while the song is on beat that lets the player use an input to call an on-beat function. If the button is pressed during this on beat function, particles are emitted from the boat. Originally this was going to be something like EQ filter manipulation but that didn't make it to the final product.

5.5.3 Item 2 – Wind Particles

Item 2 of this sprint was wind particles.

To add some flair to the boat, wind particles were attached to the player. These were made using the Unity particle system and animated with the various options available in the particle system. The particles change speed and size over lifetime. The changes are created by creating a graph in the particle system for each vector.

5.6 Sprint 4

5.6.1 Goal

The goal of sprint 4 was to work on the UI of the game.

5.6.2 Item 1 – Movement UI

The first item of this sprint was the movement UI. Movement UI was created so the player could see the influence that their input has on the direction that they're moving. 2 circles were created that each represented a joystick or set of keys, placed at the bottom centre of the screen. When creating UI in Unity, it's important that you set the canvas to scale with screen size so that the placement of the objects is consistent on every screen resolution. The circle on the left represents the left stick or the W and S keys, and the circle on the right represents the right stick or the J and L keys. During testing, players found the circle backgrounds confusing as to why the input UI were circles when the smaller circle could only move on one axis.

After these two circles was set up, the idea to create a wind vane like UI was implemented in the centre of the screen that would show the resultant vector of the player’s input. This would indicate the player’s movement direction in respect to the cardinal directions.

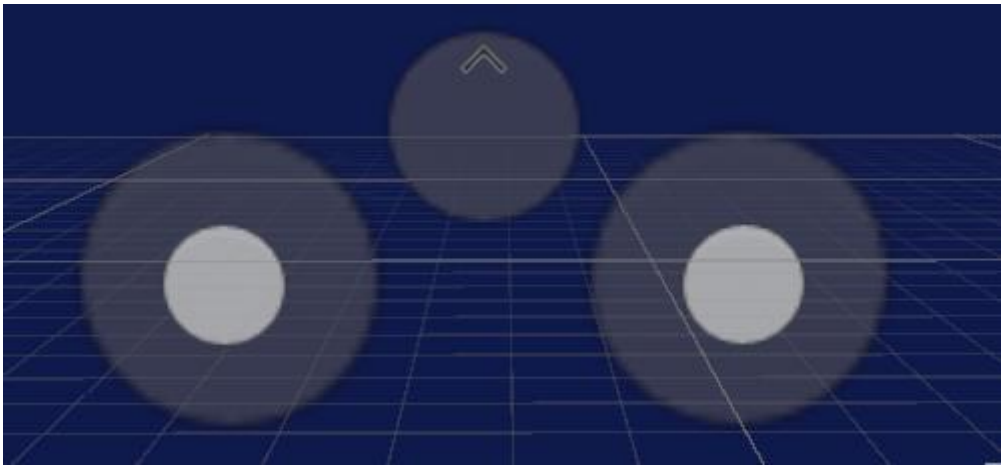

Figure 34 Navigation UI in scene view

5.6.3 Item 2 – Song selection UI

The next item of this sprint was song selection UI.

A UI was needed for the player to select the current track. The first idea implanted for this was a radial menu UI. A wheel-based radial UI is very common in video games, especially in recent years. Popularised by the Grand Theft Auto series for its use in selecting the player's weapon, it became known as the weapon wheel.

In Unity, the wheel is composed of many parts. At the top of the wheel object, there is a black overlay that goes over the screen behind the wheel to add some contrast to the UI. There is then a wheel parent that has five buttons attached to it. When the parent presses the button to activate the wheel UI sets itself to active turning on the wheel parent, the overlay and the buttons. By default, when you start the game, the buttons are set to be not interactable. This is because when you start a new game, your inventory is empty. When you go up to a Boy in Game, you collect a song. When you open the wheel again, the song is available to select. Each song has its own unique colour, so each button was handmade to have its own colours assigned. Using the Unity Animation Editor, several animations were created for the wheel, such as opening the wheel, selecting buttons, deselecting buttons, and closing the wheel.

Interactions with the wheel are handled by two scripts, a wheel controller script and a wheel button controller script. The wheel controller script is set to the parent. This script is linked to the Audio Manager track control function so that it can switch the music depending on what the player selects in the wheel. It also controls which buttons are interactable or not. The wheel button controller handles selection, deselection and hovering for the buttons. Send. It also has a set interactable function, which is what the wheel controller uses to set the buttons as interactable.

```
    public void Selected()
    {
        selected = true;
        WheelController.trackID = ID;
    }

    public void Deselected()
    {
        selected = false;
        WheelController.trackID = -1;
    }

    public void HoverEnter()
    {
        anim.SetBool("Hover", true);

    }

    public void HoverExit()
    {
        anim.SetBool("Hover", false);

    }

    public void SetInteractable(bool state)
    {
        selected = false;

        GetComponent<UnityEngine.UI.Button>().interactable = state;
    }
}
```

```
public class WheelController : MonoBehaviour
{
    public Image selectedItem;

    public Sprite noImage;
    public static int trackID = -1;
    private AudioManager music;

    public WeaponWheelButtonController[] buttons;
    public InventoryManager inv;

    int lastTrackID = -1;

    private void Start() {
        music = AudioManager.instance;
        // subscribe to event in inventory manager
        inv.OnInventoryChanged += PopulateWheel;
        // run once on start
        PopulateWheel();
    }

    void Update()
    {
        if (trackID > 0)
        {
            selectedItem.enabled = true;
        }
```

5.7 Sprint 5

5.7.1 Goal

The goal of sprint 5 was to create some objectives for the player, as well as add new music to the game.

5.7.2 Item 1 – Buoys

The first item of this Sprint was a buoy with animations. The buoy was modelled and animated in Blender. This was imported into the game and give it collision and floaters as well as the script the boat also has to give it buoyancy on the water. The visual model of the buoy is quite abstract as a distinct appearance was desired.

5.7.3 Item 2 – New Music

The next item of this print was new music. During this print, four songs were added to the game. These are all original songs. Along with this, a song inventory script and a song database were created inside the game. These scripts are used for tracking the inventory of the player to know which songs they have collected in the game. A custom SongData asset object was created to better handle inventory tracking.

A song data scriptable object was created. It has a string *SongName* and an int *TrackID*, as well as an array of AudioManager instruments *instruments*.

The inventory manager script starts by setting up a static instance. It also creates a C# list of SongData asset objects. On awake, it sets up its own instance as a singleton. It also contains functions for adding a song, getting a song checking if it has a song and clearing the songs. The function to add a song takes by taking in a SongData object asset. It then cheques if the inventory already contains that song. If it doesn't, then it will add the collected song to the list. And then invoke an OnInventoryChange() event for the wheel selector script to repopulate itself with the correct songs the player has collected.

The song database scripts is just a Singleton instance, with a list of all the songs in the game. It also contains a function to get a song by ID as some scripts need to get songs by the ID.

5.7.4 Item 3 – Save Data

The third item of this print was a save data implementation.

For the game, a save data function was created so that the player could leave the game, resume where they left off, keeping their position in the world and their inventory. Save data is a public class that is set with a list of collected songs and three floats being the players XY&Z coordinates. There's also a Save System C# script that is a singleton instance. On a week it sets up a safe path, which is the persistent data path of the. Games Storage. It saves into a file called *data.json*. For the save, it gets all the songs from the inventory and then also saves the players position into a vector3. The save data class is then written into a json file.

5.8 Sprint 6

5.8.1 Goal

The goal of sprint 6 was to create a playable tutorial.

5.8.2 Item 1 – Islands

The first item of this sprint was to add an island. The island was modelled in Blender. The island was added to the game and given a trigger collider to add to your inventory when you collide with it.

A border for the tutorial area was created, which vanishes when the player finishes the tutorial, letting them into the open world

5.8.3 Item 2 – Dialogue

The second item of this sprint was a dialogue system. The tutorial needed dialogue to explain the game to the player.

The dialogue system is designed to be as reusable as possible. Initially the dialogue was hardcoded, but that was not going to work as dialogue was needed more than once. So, it was decided to try and design dynamic system that could be reused. This system was explained previously in the implementation chapter. During making the dialogue system various sounds were sampled through user testing to see what participants thought of the sound. A few iterations were need as several tests showed it was considered unpleasant or didn't fit the tone. Even after picking one final one, it was decided that the sound would be changed again.

5.8.4 Item 3 – Tutorial

The last item of this sprint was the tutorial itself. The tutorial is a short, scripted section of the game that teaches the player had to play the game. It's a brief section about two minutes long. There is dialogue to guide the player through the tutorial. This dialogue teaches the player the controls and what they're doing in the world. When the player reaches the end of the tutorial, the border surrounding the area drops so that they're free to explore the open world.

The tutorial script is very basic. There are colliders around the world that when touched, trigger a function that adds to a number in a script handling the tutorial. When this number reaches a certain value, which is the number of checks to pass the tutorial, the tutorial is considered complete. It also has a boolean that is saved so that the game knows the tutorial has been completed by the player if they decide to quit and come back later.

5.9 Sprint 7

5.9.1 Goal

The goal of sprint 7 was to create a main menu and finish the game.

5.9.2 Item 1 – Main Menu

The first item of this print was a main menu.

The main menu is the first thing that the player will see when they open the game. It needs to have a start game button, an options button and a quit button. The main menu script contains all the functions for all the buttons on the main menu. These include NewGame(), Options(), OptionsBack() which is used to get out of the options page, and CloseGame().

```
public class MainMenu : MonoBehaviour
{
    public GameObject mm;
    public GameObject options;

    public void NewGame()
    {
        Debug.Log("the clicker");
        SceneManager.LoadScene("MainWorld", LoadSceneMode.Single);
    }

    public void Options()
    {
        mm.SetActive(false);
        options.SetActive(true);
    }

    public void OptionsBack()
    {
        options.SetActive(false);
        mm.SetActive(true);
    }

    public void CloseGame()
    {
        Application.Quit();
        Debug.Log("this is where it would quit in the build lol");
    }
}
```

Figure 35 Main menu script

When the player clicks on the options button, they are taken to the options page, and the main menu is closed. The options include a retro mode toggle and a fullscreen toggle. There is code here to support volume controls for sound effects and music, but these were not fully implemented for the submission of the project and will be demonstrated at the project demo. There is a “back” button at the bottom of the options page which saves the player’s settings using Unity’s built-in PlayerPrefs class that stores data between game sessions. An OptionsData class was created to store the player’s settings in a C# class.

A settings manager creates a new OptionsData class that handles saving this class into the PlayerPrefs class. This manager also loads the PlayerPrefs into the OptionsData when the player starts the game so their settings are applied when the application is opened.

```
public void SaveSettings()
{
    string json = JsonUtility.ToJson(settings);
    PlayerPrefs.SetString("settings", json);
    PlayerPrefs.Save();
}

public void LoadSettings()
{
    if (PlayerPrefs.HasKey("settings"))
    {
        string json = PlayerPrefs.GetString("settings");
        settings = JsonUtility.FromJson<OptionsData>(json);
    }
    else
    {
        settings = new OptionsData();
    }

    ApplySettings();
}

public void ApplySettings()
{
    Screen.fullScreen = settings.fullscreen;

    RetroCheck?.Invoke();
}
```

Figure 36 Options manager script

A pause menu accessible while playing the game was also created. This is accessed when the player presses escape or the start button on their controller. It contains the options to save and load the game.

5.9.3 Item 2 – Reworked inventory and save system

During this sprint the music inventory system was reworked, so that the player's progress would be saved between sessions down to exactly which instruments they had collected. It was also important to rework this system as there were some unexpected behaviours when the player collected multiple tracks such as some instruments never being collected and some instruments never muting or changing volume.

In the inventory, each song is linked to a dictionary which contains the songData and a hash set that contains instrument enum options from the audio manager. When a musician is collected, the musician is added to this dictionary inside the hash set. This creates a key value pair of the song being the key and the value being the hash set of instruments collected, similar to an array.

Because JSON cannot save dictionaries, this dictionary is changed into a C# class instance with an int for a songID and a List that takes in the hash set to create a class instance of the key value pair. These class instances are what are saved in the save data.

```
[System.Serializable]
public class SaveData
{
    public List<int> collectedSongIDs = new List<int>();
    public List<SongProgressData> songProgress = new List<SongProgressData>();
    public bool tutorialComplete;

    public float playerX;
    public float playerY;
    public float playerZ;
}

[System.Serializable]
public class SongProgressData
{
    public int songID;
    public List<AudioManager.Instrument> collectedInstruments = new List<AudioManager.Instrument>();
}
```

Figure 37 New save data classes

When the player loads the game, all of this data is turned back into the correct data types and set to the player's current inventory, effectively applying all the progress they made from the last session to the current session.

```
public void LoadGame(Transform player)
{
    if (!File.Exists(savePath))
    {
        Debug.Log("No save file!");
        return;
    }

    string json = File.ReadAllText(savePath);
    SaveData data = JsonUtility.FromJson<SaveData>(json);

    // Clear songs (if going to old save)
    InventoryManager.Instance.ClearSongs();

    // Load songs
    foreach (int id in data.collectedSongIDs)
    {
        SongData song = SongDatabase.Instance.GetSongByID(id);
        if (song != null)
            InventoryManager.Instance.AddSong(song);
    }

    // Load song progress
    foreach (var progressData in data.songProgress)
    {
        SongData song = SongDatabase.Instance.GetSongByID(progressData.songID);
        if (song == null) continue;

        foreach (var instrument in progressData.collectedInstruments)
        {
            InventoryManager.Instance.AddMusician(song, instrument);
        }
    }
```

Figure 38 New loading function including inventory rework

The save data now also contains a boolean value that changes depending on if the player has completed the tutorial or not.

```
    // Skip tutorial is done
    if(data.tutorialComplete)
        TutorialManager.Instance.OnTutorialComplete();
```

Figure 39 Tutorial skip boolean

5.9.4 Item 3 – Final level placement

The last item of this sprint was lay out the final level of the game. This included putting the islands and buoys into place in an order that felt natural for the player, ensuring that steady progress was made while also encouraging exploration. The level was changed a couple times before the final build due to user testing, and a final layout was settled on after a couple rounds of testing.

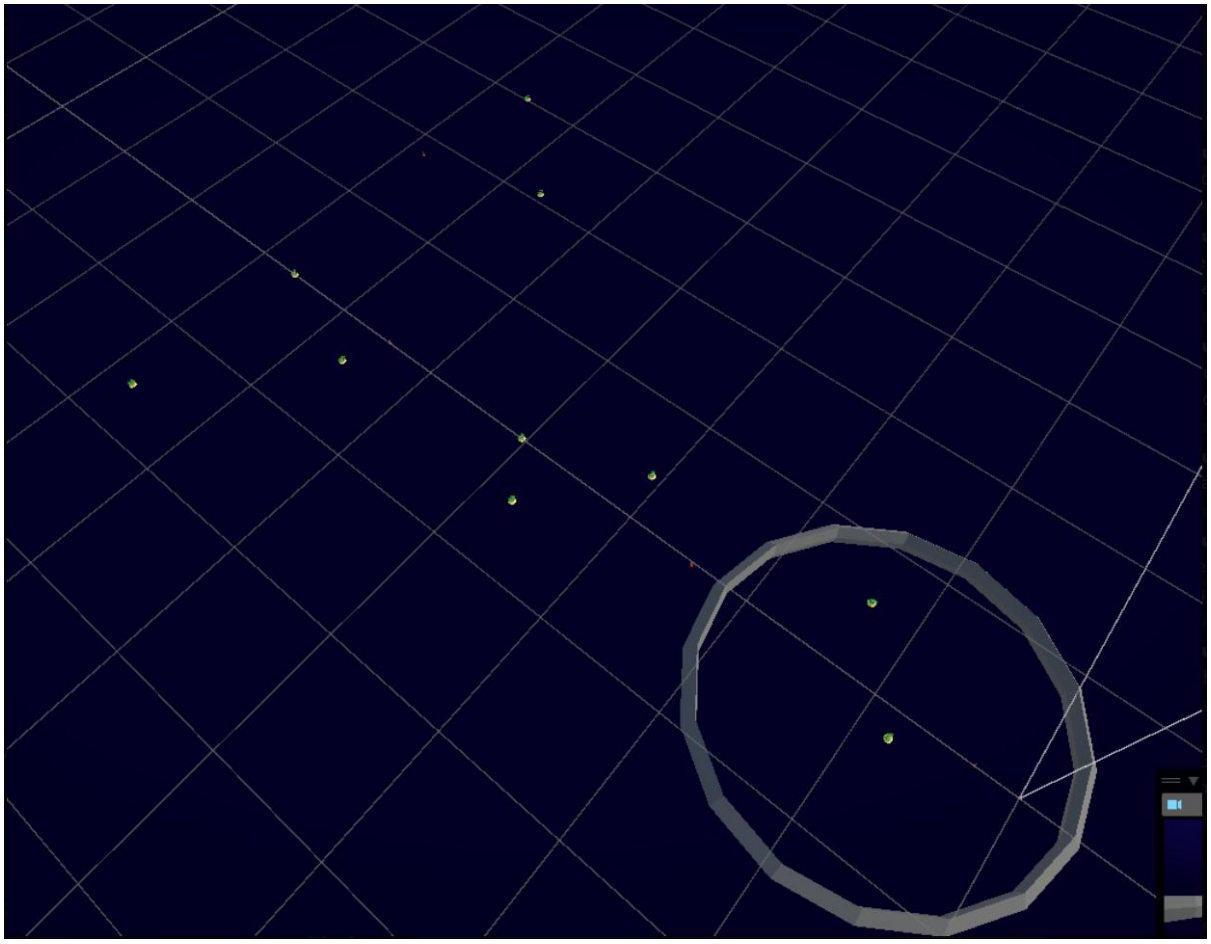

Figure 40 Final level layout

5.10 Conclusion

This chapter described the implementation phase of the game. It included a breakdown of the sprints and how things were implemented in each sprint. All scripts have been attached to the appendix in the OneDrive folder submitted, alongside the demonstration screencast.

6 Testing

6.1 Introduction

This chapter describes the testing that has been undertaken for the application. This chapter is presented in two sections:

1. Functional Testing
2. User Testing

Functional testing is a type of software testing whereby the system is tested against the functional requirements. The app is assessed by looking to see if the actual output for a given input corresponds with the expected output. The tests should be based on the requirements for the app. The results of functional testing can indicate if a piece of software is functional and working, but not if the software is easy to use.

User testing looks to see if a piece of software is easy and intuitive for the user.

6.2 Functional Testing

This section describes the functional tests which were conducted on the app. These functional tests can be categorised as:

- Navigation
- Gameplay
- Save Function

Functional testing uses a Black Box Testing technique which means that the internal logic of the system being tested is not of interest to the tester. The tester is only interested in whether the actual output agrees with the expected output.

6.2.1 Navigation

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
1	Quit button on the main menu	Click Quit	Application Closes	Application Closes	Unity Application.Quit() closes the window
2	Options button on the main menu	Click Options	Options page opens	Options Page opens	Options object activates appropriately
3	Options to main menu button	Click “back”	Main menu opens	Main menu opens	Works as expected
4	Start game from main menu	Click “Start game”	Taken to main game	Taken to main game	SceneManager works appropriately,

					loads in single mode

6.2.2 Gameplay

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
1	North/South movement	Left stick/W&S	North and South movement only	North and South movement only	Left stick controls only allow for north and south movement as intended
2	East/West movement	Right stick/J&L	East and west movement only	East and west movement only	Right stick controls only allow for east and west movement as intended
3	Opening selection wheel	R2/Control	Selection wheel opens	Selection wheel opens	Selection wheel opens and animation plays correctly

6.2.3 Save Function

Test No	Description of test case	Input	Expected Output	Actual Output	Comment
1	Save function	Save button clicked	Save created, data.json	Save created, data.json	Json save file is created in LocalLow under "DefaultCompany", containing player's position and inventory
2	Load function	Load button clicked	Game loads player's data	Game loads player's data	Json file is read and applies position to player's current Transform and loads their inventory. Also bypasses tutorial.

6.3 User Testing

6.3.1 User Testing Introduction

User testing was conducted by sending the application in it's nearly-finished state to a variety of participants. It was important to have a varied pool of participants, some with gaming experience and others with less. This variety allows participants familiar with games to point out flaws with the video game experience, and participants who are not gamers to point out the flaws in the tutorial section and general learning experience when playing the game.

6.3.2 User test cases

6.3.2.1 Case 1

Does this participant play games often: Yes

Positive Feedback:

- Good graphics and music
- Found controls intuitive, especially when playing with a controller
- Completed game without getting lost

Negative Feedback:

- No FPS (Frames per second) limit, which increased load on CPU
- Suggested increasing the length of the ocean
- World layout too simple

How this feedback was used for iteration: This feedback proved useful for a number of reasons, mainly in that the controls felt intuitive for this participant. They completed the game without

getting lost but also suggested that the world layout was too simple. With this feedback, the layout of the level was altered slightly to encourage players to explore more.

6.3.2.2 Case 2

Does this participant play games often: No

Positive Feedback:

- Good graphics and music
- Although struggled with novel controls initially, accustomed to them after a few minutes
- Enjoyed playing with the “retro mode” setting

Negative Feedback:

- Struggled with controls initially, suggested “simple controls” option
- Could not locate the last piece of music

How this feedback was used for iteration: As this participant could not locate the last piece of music, it was moved to be in a more obvious location as the application for this project is meant to be completed in one sitting. In the future, a simple controls option could certainly be added, and was something initially thought about during the implementation phase of the project.

6.4 Conclusion

This chapter outlines the testing carried out on the application, divided into functional testing and user testing.

Functional testing evaluates the app for its technical requirements, testing features in isolation to assess if a specific input leads to a desired output. The features tested with this approach include UI navigation, gameplay, and save functionality.

User testing assesses the usability and player experience by involving participants with various previous experience with games. These tests produce feedback from an end-to-end test case where the user plays through the full game experience. The feedback received for these tests include:

- Struggling with controls
- Level design too simple
- Confusing goal placement
- Performance concerns with no FPS limit

This feedback could prove useful for further development in the future, as not all feedback could be put into the game for further iteration.

7 Project Management

7.1 Introduction

This chapter describes how the project was managed and how well the student kept within the project guiderails. It shows the phases of the project, going from the project idea through the requirements gathering, the specification for the project, the design, implementation and testing phases for the project. It also discusses Trello, GitHub and project member journal as tools which assist in project management.

7.2 Scrum Methodology

Scrum methodology is a work pattern that structures work into equidistant time-based windows, known as sprints (Drumond, 2024). These sprints have defined goals and items and conclude with a review of the work done during the sprint. This review ensures that the work carried out during the sprint met the goals while also keeping the standard for the quality of the work.

Scrum methodology was used during the development of this project, as described previously in Chapter 5's Sprint detailing. It proved to be an effective approach to development, specifically in game development as it involves taking larger problems and breaking them into smaller ones.

7.3 Project Management Tools

7.3.1 Trello

Trello was one of the software tools used to manage the project. Trello is a website that uses Kanban organisation methods to track tasks and progress.

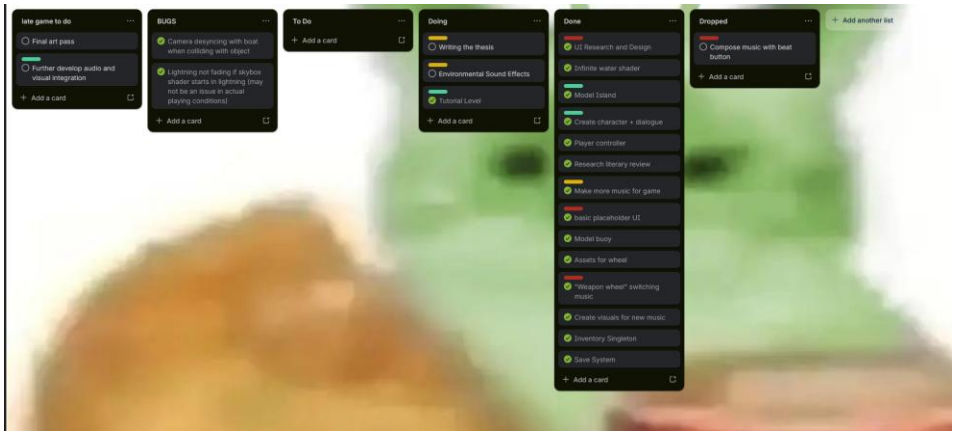

Figure 41 Trello board that was used - mid-development

Trello proved to be very useful in the development of the project. Given the simple Kanban layout, it was easy to add tasks and move them along. One aspect that proved particularly useful was putting tasks into words to break them down into smaller problems. It also was good for writing down spontaneous ideas so that they wouldn't be forgotten while working on other features.

7.3.2 GitHub

GitHub is a developer platform that allows developers to create, store and manage their code online. GitHub was used during the development of this project to manage and back-up the code and assets as well as version control.

For this project GitHub Desktop was used rather than the CLI. The reason why is because GitHub Desktop is easy to use given its user-friendly GUI interface. I always use GitHub Desktop for game development projects, as you might not necessarily be working with a terminal while developing games.

GitHub proved to be very useful during the development of the project. There were multiple instances of the files corrupting and being unrecoverable were it not for GitHub. I would regularly commit to the GitHub repository for this reason. Rolling back to a previous commit is very easy with GitHub desktop, and it also meant that I could push the GitHub repository to the GitHub classroom for submission at the end of the project.

7.3.3 Miro

Miro is an online web app that lets you create and share online whiteboards for organisation and creative thought process. A Miro board was set up at the start of the project so that Timm and I could communicate visually and journal progress. This is where I visually logged thoughts that I had along the sprints and posted any sort of tutorials or resources that I wanted to use or found particularly useful.

7.4 Reflection

7.4.1 Your views on the project

My feelings on the project are conflicted. On the one hand, I'm proud of the work that I've done. But on the other hand, I do feel like the project didn't reach its full potential. I do feel like the project. I do feel like I underestimated the workload in making a game, especially one with adaptive audio as a core mechanic. Still, I think the learning opportunities I was given during the development of this game were great. I learned a lot about visuals in games and took a keen interest in shaders during the development of the game. I think I as well didn't realise just how much goes into making a game. There are so many small parts that you need to worry about with individual scripts and game objects. It’s a lot of small moving parts that create one big cohesive thing.

7.4.2 Working with a supervisor

Timm and I met every week online to discuss the progress of the project. Timm provided useful insights into the progress and the path that the projected needed to follow. The weekly meetings proved useful to keep up my motivation with developments, as well as keeping scope under check.

7.4.3 Technical skills

Given that the project was a game development project, I learned a lot of skills that were relevant to development of games. This included programming in C# and general good practise in Unity development. It also I also learned a lot in developing. I also learned a lot about working with audio inn games. I got to further develop my skills in Blender with 3D modelling and animation. I learned a lot about shaders and the shadergraph in Unity, and I took a keen interest in how shaders work and how the GPU works. To me, that was the most interesting part of the development.

8 Conclusion

8.1 Project Conception and Goal Achievement

I started this project because I wanted to bring together 2 of my creative outlets into one project; programming and music. From the beginning I wanted to make a short game experience that integrated the music I had made in an interesting way. The initial idea was that the player controlled the wind, and that would change the direction the boat moved, and as the boat changed direction the music would change. Maybe that would have made for a more interesting game, but I changed it pretty quickly because I thought that it would be annoying for the player. So, I tried to brainstorm an idea where the player interacted with the environment to change the music instead of all of the music being dependent directly on the player's input.

Thus, the idea of having a boat that collected musicians came along. At the start of the project, my supervisor asked "Why sailing? Why can't this be a game where someone is driving around a city picking up musicians from different buildings?". At the time I didn't have an exact answer, but I know it's the idea I wanted to stick with. And through developing I realised that sailing in the vast open ocean gives a sense of freedom that one has while creating music. You can go any direction you want and choose your own path, and to me that resonated with the feeling I wanted to give the player.

This project is a short demo, but I think it has a lot of potential to be something more. So, I fully intend to continue to work on the project and open the development up to my friends as they've gained an interest as the development came along. It's exciting to think about what can be made with this idea, as all testers found the concept of the game very strong.

8.2 What was learnt

While working on this project I was introduced to a lot of novel concepts for myself, and I feel like I gained so much experience in working with all the new technologies in particular. Looking back on code that I had written at the start of project versus towards the end, it was clear how much of a better grasp I had of programming concepts, especially practices that are specific towards game development. Towards the end I was running into issues where I had been creating singletons but then creating new instances to reference anyway, so debugging those issues was a good refresher on singletons and also how all my game objects connect to each other.

I also had to learn a lot about working with audio and how that should be implemented in games. During my testing I had to go through multiple iterations of testing sounds with the dialogue before players thought that it fit the game, and that was mostly due to my poor audio mixing for sound effects and also choosing a sound that fit the game. Over the course of producing the music for the game as well I had to learn some basic mixing and mastering techniques and employ music theory to get the sound I wanted while having the music still sound good in the game. FMOD was an entirely new technology to me that I needed to research how to use, and upon reflection I think for a first attempt it was integrated pretty well using an audio manager singleton and a singleton that handled event references specifically.

8.3 How the project could be further developed

This project certainly has a lot of potential. It started off as a very ambitious project that needed to be downscaled as time went on, purely from the fact that I had not realised how much work goes into making a game. There are so many small moving parts of a game that need to work together to form a strong final application. When this project started I had a very clear vision in my head of what it would look like and how the player would interact with it and that vision may not have been realised here for this paper but it certainly has the potential to be realised one day.

Some examples of features I could think of to include would be more extensive audio manipulation and more music and a more interesting world to play in. The final world is only buoys and islands and a tutorial area, so I think having more interesting looking islands that you could explore or different parts of the sea having different levels of choppiness and weather would be very interesting.

FMOD Studio is also an incredibly powerful piece of software that lets you do so many things with your audio and this project barely scratched the surface with what can be done. For a new technology for me I am happy with its implementation but there is so much more that can be done. If I were to make this game again, I would love to find a way to incorporate things like changing EQ to eliminate the highs of the soundscape if a wave crashes into you or changing pitch if members of the band get drunk or something.

Speaking of the band members, I wanted to include actual characters and art of those characters in the game, but it just wasn't meant to be. I had initial designs and ideas for these characters but realised soon into the project that this was not going to be covered in the scope of the project.

I do think reflecting on the project that I wish I had done more, and at one point I was stressing over how much content was actually in the game, but I remembered that this project was something I undertook to learn new technologies and expand my knowledge for game development and working with audio, so I won't let the hard work I put into the project be invalidated by the fact I did not get everything I thought of back in October of 2025.

9 References

Altenstedt, I., & Willig, R. (2025). *Sounds and Emotions in Video Games: An Analysis of Outer Wilds' Sound Design* . https://doi.org/10.2139/ssrn.5667070

Coe, D. (2017). Why people play table-top role-playing games: a grounded theory of becoming as motivation. *The Qualitative Report*, *22*(11). https://doi.org/10.46743/2160-3715/2017.3071

Costikyan, G. (2005). *I Have No Words & I Must Design: Toward a Critical Vocabulary for Games*. http://www.costik.com/nowords2002.pdf

Definition of IMMERSION. (2019). Merriam-Webster.com. https://www.merriam-webster.com/dictionary/immersion

Heckmann, C. (2020, September 27). *Leitmotifs and Musical Themes Explained*. StudioBinder. https://www.studiobinder.com/blog/what-is-a-leitmotif-definition/

Leblanc, C. (2019). *Creating Games*. https://files01.core.ac.uk/download/pdf/327105102.pdf

Lundqvist, L.-O., Carlsson, F., Hilmersson, P., & Juslin, P. N. (2008). Emotional responses to music: experience, expression, and physiology. *Psychology of Music*, *37*(1), 61–90. https://doi.org/10.1177/0305735607086048

Manasa Jayasri. (2024, October 16). *Unity vs. Godot: A Game Developer's Guide*. DEV Community. https://dev.to/manasajayasri/unity-vs-godot-a-game-developers-guide-2a6o

Murphy, D. J. (2019). How Musical Leitmotifs Enhance Narration and Evoke Emotion. *Digra.org*. https://doi.org/10.26503/dl.v2019i1.1820

Örtqvist, D., & Liljedahl, M. (2010). Immersion and Gameplay Experience: A Contingency Framework. *International Journal of Computer Games Technology*, *2010*, 1–11. https://doi.org/10.1155/2010/613931

Russoniello, C. V., Fish, M., & O'Brien, K. (2013). The Efficacy of Casual Videogame Play in Reducing Clinical Depression: A Randomized Controlled Study. *Games for Health Journal*, *2*(6), 341–346. https://doi.org/10.1089/g4h.2013.0010

Sanyshyn, B. (2024, November). *Wwise or FMOD? A Guide to Choosing the Right Audio Tool for Every Game Developer*. Thegameaudioco.com. https://www.thegameaudioco.com/wwise-or-fmod-a-guide-to-choosing-the-right-audio-tool-for-every-game-developer

Tarro57. (2023, March 14). *How to Make Adaptive Video Game Music*. Youtu.be. https://youtu.be/MbqfVB22ngI?si=IgxVPX3PAQdbQ_FD

Terzioglu, Y. (2015). *Purdue e-Pubs Immersion and identity in video games*. https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1640&context=open_access_theses

The Art of Scoring - PC Feature at IGN. (2025). IGN. https://web.archive.org/web/20110924004211/http://pc.ign.com/articles/119/1193729p1.html

The Role of Sound and Music in Creating Atmosphere In Games - GAME PILL Game Studio. (2024, November 13). GAME PILL Game Studio. https://gamepill.com/the-role-of-sound-and-music-in-creating-atmosphere-in-games/

Vagnini, S. (2008). Modular Art - Stefano Vagnini. In *Youtu.be*. https://youtu.be/3brCx3wWr0I?si=0oOsApdw-rUgZ0_x

Video Games: Overview | EBSCO. (2020). EBSCO Information Services, Inc. | Www.ebsco.com. https://www.ebsco.com/research-starters/sports-and-leisure/video-games-overview

What is Immersion : Immersion Definition | Unity. (2025). Unity. https://unity.com/glossary/immersion

Wikipedia Contributors. (2025, April 2). *Anemoscope*. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/Anemoscope