Can audio middleware be utilised to develop practical and reusable audio systems?

Commercial Games Development MSc

Creative Technology Dissertation

Samuel Scott – 17019429

Supervisor: Louca Coles

Submission Date: 07/09/2023

Keywords: Applied computing; Sound and music computing; Procedural audio; Spatial audio; Dialogue systems; Human-centred design

Abstract

While audio middleware has become a staple for audio implementation within video games, the expectations of consumers are rapidly increasing. Furthermore, while the resources available to gaming platforms grow, as do the capabilities available to sound designers. While there is an array of literature available relating to sound design, the same quantity of literature is not typically available for technical sound design, especially when considering complex audio systems. Therefore, this research paper aims to identify the purpose sound plays in video games and to investigate both novel and industry approaches to audio implementation for procedural audio, spatial audio, and dialogue implementation. Findings are utilised to inform development decisions on a set of audio tools that aim to extend the capabilities available to sound designers when implementing audio into video games developed in *Unity* (Unity Technologies, 2023) with *FMOD Studio* (Firelight Technologies, 2023). This paper identifies how the biggest area for further research is with dialogue implementation. As although there have been numerous innovations with third-party plug-ins available in FMOD, especially spatial audio, dialogue development has not seen the same extent of interest from commercial audio software companies, despite the recent popularity of large language models.

Acknowledgements

I would like to express gratitude to my mentor, Louca Coles, for his continual guidance and support during this dissertation.

Additionally, I would like to thank Luke Reed for not only imparting some of his game audio industry knowledge on me throughout this dissertation but also for inspiring me to become a technical sound designer.

I would also like to express my appreciation to Chris Nash, who provided me with the opportunity to partake in his procedural game music internship using the Manhattan tracker software. This internship was not only a great experience but also enabled me to partake in an aspect of procedural game audio I had not explored previously.

Furthermore, I would like to thank all the lecturers on the master's course for their guidance throughout the course. In addition, I would also like to thank Nikolaos Ersotelos for providing insightful information on academic writing for this module as well as others.

Finally, I would like to thank my partner, Dora Gerhardt, who has supported me through the ups and downs. Thank you for your unconditional support throughout the degree!

Contents

1. Introduction 8

2. Literature review 9
   2.1. What role does audio have in videogames? 9
   2.2. What is audio middleware? 10
   2.3. Procedural audio 11
      2.3.1. Procedural audio in video games 11
      2.3.2. Adapting linear audio with effect parameterisation 12
   2.4. Sound Spatialisation 12
      2.4.1. Spatialised audio in video games 12
      2.4.2. Sound simulation in virtual environments 13
   Figure 1. Wave Equation (Zotter and Frank, 2019). 13
   Figure 2. Inverse-square law (Patel, 2020). 14
   Figure 3. Sabine Equation (Abdullah et al., 2020). 15
   2.5. Dialogue 16
      2.5.1. Dialogue in video games 16
      2.5.2. Branching dialogue systems 17
   Figure 4. High level diagram of a branching dialogue systems architecture (Stevens, 2015). 18

3. Methodology 20
   3.1. Introduction 20
   3.2. Supporting procedural sound design through effect parameterisation 20
   Figure 5. Function as part of the static utility toolset to support different types of procedural audio. 21
   Figure 6. Example of one-shot parameter call utilising the one-shot system. 21
   Figure 7. Studio emitter script provided by the FMOD for Unity API. 21
   Figure 8. Utility script that allows users of the system to handle FMOD event parameter functionality. 22
   3.3. Audio occlusion and obstruction system 22
      3.3.1. Emulating propagation with linecasts 22
   Figure 9. Occlusion system being used to emulate a small sound size. 23

Figure 10. Occlusion system being used to emulate a larger sound size, with a higher sound occlusion widening setting. 23
3.3.2. Calculating the third vertex in an orthogonal triangle 24
Figure 11. Illustration of vertex points in an orthogonal triangle (Croft and Davison, 2018). 24
Figure 12. Calculations used for occlusion widening. 24
Figure 13. Algorithm implemented to calculate the sound widening vector for the occlusion system. 25
3.3.3. Occluding or obstructing the sound source 25
Figure 14. Inspector view of the Spatialised audio script, provided with the Occlusion system. 26
Figure 15. Automation curve for the low-pass cut-off value, automated by the occlusion parameter. 26
3.4. Dialogue system 27
3.4.1. Introduction 27
3.4.2. XML Application 28
Figure 16. Functionality provided by the XML application. 28
Figure 17. Example of RTPC approach to authoring a large collection of sounds that can interact with game data (Somberg, 2017). 29
Figure 18. An XML file generated through the application. 30
3.4.3. Deserializing XML files into Unity 31
Figure 19. Data structure for the dialogue system. 31
Figure 20. Function that validates an XML condition value and parses it into its respective type. 32
Figure 21. Function developed to detect condition type and set the Enum value accordingly. 33
3.4.4. Dialogue Entity 33
Figure 22. Example of Dialogue Entity object during runtime, being used in the player response sequence mode. 34
Figure 23. Debugging gizmos drawn for Dialogue Entity systems utilising trigger enter
and exit trigger types. 35
Figure 24. Further debugging features provided through the system. 35
3.4.5. Game data resolver 35
Figure 25. Game data resolver, responsible for querying game data and allowing game designers to and entries to the game data hashtable. 36

Figure 26. Function implemented to utilise the game data resolver to query a dialogue condition. 36
3.4.6. Node-based player response branching 37
Figure 27. Player response data structure. 37
Figure 28. Player response example utilised in the test scene provided with the package. 38
3.4.7. Supporting modular game design with the observer pattern 39
Figure 29. Architecture for the observer-based dialogue system. 40

4. Project management 41
4.1. Time management 41
4.2. Adaptations 41
4.2.1. One-shot and parameter system 41
4.2.2. Occlusion 41
4.2.3. Dialogue 42

5. Results 43
5.1. Evaluation 43
5.2. Integration testing 43
5.2.1. One-shot handling and parameters system 44
Figure 30. Integration scene created to test the One-shot and parameter handling system. Utilises the Easy FPS (Haberle, 2016) package. 44
5.2.2. Occlusion system 45
Figure 31. Integration scene created to test the occlusion system. 45
5.2.3. Dialogue System 45
Figure 32. Integration scene created to test the dialogue system. 46

6. Further work 48
6.1. Introduction 48
6.2. Performance testing 48
6.3. User testing 49
Figure 33. Calculating error percentage through user testing (Badget et al., 2011) 49

7. Conclusion 50

Bibliography 51

Appendices 59
Appendix 1. Original Gantt chart submitted with proposal. 59
Appendix 2. Adapted Gantt chart for the one-shot and parameter system, originally intended to be the footstep and gunshot system. 60
Appendix 3. Adapted Gantt chart for the occlusion system, originally intended to be the 3D and looping audio system. 61
Appendix 4. Adapted Gantt chart for the dialogue system. 62
Appendix 5. Performance testing for dialogue integration scene. Snapshot for components is taken at the highest performance spike that occurred during the scene. 63
Appendix 6. Link to one-shot and parameter system prototype video. 64
Appendix 7. Link to occlusion system prototype video. 64
Appendix 8. Link to dialogue system prototype video. 64
Appendix 9. Link to dialogue XML application prototype video 64
Appendix 10. Link to the project's git hub repository 64

1. Introduction

The field of sound design in video games has undergone significant advancements since its inception, both in the capabilities available to sound designers as well as the methodologies utilised to implement audio (Sinclair, 2020). However, the rapid and significant growth in the resources accessible to gaming platforms has unquestionably led to a shift in the expectations around modern game audio (Beig, 2019). Despite this, there is a sparse amount of literature available for supporting the advancements in technical sound design when compared to contemporary techniques for designing sound (Wratt, 2018).

Therefore, the objective of this study is to determine the significance of audio in video games and to discern the prevailing expectations for audio implementation in contemporary video game development. Additionally, this research attempts to identify and analyse the strategies employed to fulfil these expectations for three critical areas of audio: procedural sound design, spatialised audio, and dialogue. Subsequent findings will be utilised to ascertain the specific attributes the audio middleware *FMOD Studio* (FMOD) (Firelight Technologies, 2023) provides to sound designers and technical sound designers, relating to these areas of audio implementation. In addition, a critical evaluation of these attributes will be conducted to determine potential areas for improvement or expansion within the middleware.

Established findings will be used to synthesis a series of audio implementation tools that aim to improve on some of the identified areas for improvement within FMOD. Overall, while audio middleware has become a fundamental part of game audio (McIntosh et al., 2022), it is important to review some recent innovations in game audio implementation. In addition, it is necessary to critically review whether audio middleware can be utilised to make practical audio systems.

Therefore, this paper aims to identify both industry and novel approaches to various types of audio implementation to support an investigation into features provided by FMOD. These findings will be utilised to inform design decisions on providing systems that can enhance or extend functionality provided by the middleware.
This paper aims to contribute to the field of technical sound design through two methods. Firstly, through the research findings established within the literature review of this paper. Secondly, through a technical contribution, being a set of audio tools for the FMOD middleware, that sound designers and technical sound designers can use inside of *Unity* (Unity Technologies, 2022).

2. Literature Review

2.1. What role does audio have in videogames?

The perception of sound is significant in how humans comprehend the surrounding environment and according to studies is the secondly most used sense, following vision (Guillen et al., 2021). However, according to Kenwright (2023), sound is frequently overlooked as a crucial component in various interactive settings, including video games. Nevertheless, sound continues to hold significant importance in the realm of video games and is frequently employed as a means for delivering feedback and enhancing player engagement (Andersen et al., 2021). Hence, the use of audio in video games serves as a prevalent means of conveying various types of information. Furthermore, Kenwright (2023) suggests that music and sound have the capability to evoke reactions, memories, and emotions.

Therefore, video games frequently employ a range of audio, which can be divided into three categories: background music, ambient sounds, and sound effects (Guillen et al., 2021). Firstly, Andersen et al. (2021) stated that background music in video games should be able to adapt and effectively convey the unfolding of events. In addition, background music is created with the intent of enhancing the atmosphere and aesthetic that has been established within a game. Secondly, ambient audio is frequently diegetic and utilises environmental and locational sounds to immerse the player (Sinclair, 2020). Finally, sound effects are often short and specific, reflecting the player's interaction within a scene (Andersen et al., 2021).

At the same time, Kenwright (2023) highlighted how game audio could be further categorised by how it interacts with a player, as either passive or active sound. A passive sound is something contained in the background and may influence mood, while an active sound engages awareness. Through applying this categorisation of game audio, passive sounds are typically comprised of ambient sounds and background music, while active sound would be comprised of sound effects, typically one-shots.

Similarly, audio within a video game could be regarded to serve a purpose such as informing or immersing the player (Guillen et al., 2021). Informative audio is frequently used to provide the player with auditory queues pertaining to the gaming environment. Andersen et al. (2021) stated that informative audio needs to provide a stimulus that can convey distance, direction, and dimension. Additionally, immersive audio engages the player and often supports worldbuilding inside a game by reflecting the player's surroundings (McIntosh et al., 2022).

Overall, the significance of sound in video games has been recognised as an essential element that is often necessary for communicating an array of information. According to Beig (2019), there is a positive correlation between the advancement of hardware capabilities and the expected standards from consumers. Consequently, with the increasing needs of modern game development, there is a corresponding increase in the complexities associated with designing systems that fulfil these requirements. As a result of this, the utilisation of audio middleware is a commonly employed approach for the integration of audio within the context of video games.

2.2. What is audio middleware?

Audio middleware refers to a type of third-party software that is positioned between the game engine and the audio hardware of a gaming device. According to Firat et al. (2022), this toolset offers sound designers the capability to create audio events and perform real-time audio mixing.

However, as stated by Davari and Verbrugge (2021), the computing cost of simulating acoustical effects can be significant due to the intricate nature of sound interaction within a real-world context. Moreover, it is typical for audio development to receive fewer resources as opposed to other areas of game design (Sinclair, 2020). Hence, audio middleware commonly fulfils the role of facilitating efficient approaches to audio adaptation and the integration of acoustical effects, which are typically provided through an Application Programmers Interface (API).
However, as the capabilities of middleware have advanced, similarly have the techniques for optimising hardware access for engineers. This is achieved through applications that can facilitate communication with a game engine to provide adaptive audio authoring and playback (McIntosh et al., 2022).

While audio middleware is known to enhance sound designers' implementation capabilities, it is critical to identify commercial and novel approaches to modern audio implementation and determine whether audio middleware can provide the functionality to support recent advancements in audio implementation.

2.3. Procedural audio

2.3.1. Procedural audio in video games

As defined by Sinclair (2020), procedural audio generation refers to the algorithmic synthesis of audio. Therefore, procedural audio provides distinct benefits in comparison to pre-recorded audio. According to Clark (2021), the computational burdens associated with generating procedural sound are comparatively smaller than iterating through pre-recorded audio data. However, Farnell (2010) describes procedural audio as a type of sound that possesses the ability to be dynamically adapted and modified in response to various events. Therefore, the use of procedural generation extends to linear sound, allowing for the adaptation of sample data through the utilisation of effect-based parametrisation.

While computationally beneficial, there is a sparsity of tools available for algorithmic audio generation, when compared to adaptable audio effects (Wratt, 2018). Nevertheless, the feasibility of computational sound generation is indeed viable. In fact, audio synthesis was a prevalent technique employed for game audio during the earlier years of video game development. A notable example is *Ballblazer* (Lucasfilm Games, 1985), which utilised the Atari 2600's sound chip to procedurally synthesise sound (Wratt, 2018). Additionally, FMOD offers a Software Development Kit (SDK) for plug-in development, enabling developers to construct their own digital synthesisers and effect plug-ins. This can be utilised to interact with data provided from the game engine to generate procedural audio (Firelight Technologies, 2023).

Furthermore, tracker software was commonly utilised in early game systems, such as the Atari ST (Yalvaç, 2019), to compose music for video games. However, with modern video game development music is typically pre-recorded (Plut and Pasquier, 2020). Despite this, it is worth noting that Unity continues to provide support for tracker modules to facilitate the deployment of music generated programmatically (Unity Technologies, 2023). Therefore, recent innovations in tracker software for game audio implementation have also been made, as is the case with the audio software *Manhattan* (Nash Audio, 2022). The tracker software has been equipped with Unity support and possesses the capability of generating procedural music (Nash, 2020). This facilitates interactions within Unity that adapt different parameters and functions provided by the tracker software.

While several methods have been identified for procedurally generating non-linear audio, it could be argued that adapting linear audio with effect parameterisation is a more time-efficient approach to adaptive audio (McIntosh et al., 2022).

2.3.2. Adapting linear audio with effect parameterisation

An audio effect can be described as a digital signal process that is applied to sample data to change specific characteristics of the signal (Clark, 2021). Audio effects have become a common method for adapting audio within video games and have proven an effective method at expanding the timbre characteristics of a sound (Rieger and Zingler, 2023). Furthermore, with audio middleware being able to communicate with the game engine, audio effects provided by the middleware can receive game data to adapt a parameter.

The *FMOD Studio API* (Firelight Technologies, 2023) provides three parameter types for audio designers. These parameters are continuous, discrete, and labelled. The first is for floating point data, the second uses integer data, and the last uses string data. Overall, these parameter types provide a breadth of granularity over parameter control by either conveying fine decimal values or more generic state and integer values. These parameter types enable the insertion of game data into the automation of an effect parameter, resulting in the adaptation of a sound.

Other audio middleware also provides similar parameter functionality to that highlighted with FMOD. Therefore, the method of implementing adaptive audio based on real-time data, is frequently employed in video games (Wratt, 2018).

2.4. Sound Spatialisation

2.4.1. Spatialised audio in video games

As highlighted by Broderick et al. (2019, cited by Andersen et al. 2021), spatial audio in games enhances the immersion a player can experience within a virtual environment. Furthermore, spatial audio allows for the sonification of game mechanics, allowing audio to act as an independent actor, providing environmental information to the player relating to a respective mechanic.

Despite complex spatial audio solutions being present within high-end audio simulations, the same extent of spatial audio capabilities is not found within all forms of video game development. Beig et al. (2019) state that this is due to the computational costs of acoustic modelling, alongside the complexities associated with modelling a human listener. Therefore, generally video games do not accurately emulate the acoustical properties of the locations they represent. Nonetheless, they frequently employ principles of audio acoustics to generate a theoretical approximation of the acoustics present in a virtual environment (Rieger and Zingler, 2023).

However, with recent developments in Virtual Reality (VR), advanced spatial audio techniques, such as ambisonics and binaural audio have seen recent use in the

development of spatial audio tools for VR applications (Çamci, 2019). A notable example is *Resonance Audio* (Google VR, 2022), which is a plug-in compatible with FMOD. It supports third-order ambisonics decoding, while utilising Head-Related Transfer Functions (HRTFs) to increase the accuracy of spectral effects, such as interaural-level differences and interaural-time differences (Anemüller et al., 2023).

Overall, while there are numerous third-party tools that can support some of the modern spatial audio advancements within game development, the scope of this paper is to create a toolset utilising *FMOD Studio* (Firelight Technologies, 2023) and the built-in spatialiser. Therefore, further research within this chapter will be focused on approaches to implementing positional audio, as well as simulating some of the acoustical phenomena that can affect sound propagating through a space.

2.4.2. Sound simulation in virtual environments

As highlighted by Funkhouser et al. (2004, cited by Davari and Verbrugge 2021), sound simulation could be categorised as either wave-based solutions, or geometrical propagation-path estimations.

The wave-based approach utilises the wave equation, which is a second-order partial differential equation. It can be used to describe the propagation of oscillations at a fixed speed. This is highlighted in Figure 1., with ‘v’ representing velocity, ‘t’ representing time, ‘y’ representing amplitude, and ‘x’ representing position. Furthermore, there are derivatives of the equation utilised for waves of different dimensions (Zotter and Frank, 2019). However, a wave-based approach to sound simulation typically solves the wave equation by utilising a partitioning scheme and provides a high level of accuracy in audio simulations (Davari and Verbrugge, 2021). Despite this, wave-based approaches are computationally expensive and therefore not typically utilised in video game development (Beig et al., 2019).

$$\frac{1}{v^2}\frac{\partial^2 y}{\partial t^2} = \frac{\partial^2 y}{\partial x^2},$$

Figure 1. Wave Equation (Zotter and Frank, 2019)

As such, commercial applications will typically employ hybrid approaches that utilise both wave-equation and geometrical propagation-based estimations (Dalenback, 2014, cited by Firat et al. 2022). Namely geometrical acoustics, which utilises raytracing to emulate the propagation of sound (Pisha, 2020), while typically introducing principles from a wave-based approach (Firat et al., 2022). Additionally, complex audio simulations often incorporate environmental effects into the propagation of sound (Beig et al., 2019).

The inverse-square law is typically employed to calculate the attenuation of sound as it propagates through a virtual environment (Beig et al., 2019), as is the case for FMOD's default attenuation mode (Firelight Technologies, 2023). The propagation of a point source and how it is attenuated is highlighted in Figure 2. and illustrates how sound that is double the distance from its source will be one-fourth of the sound source's intensity.

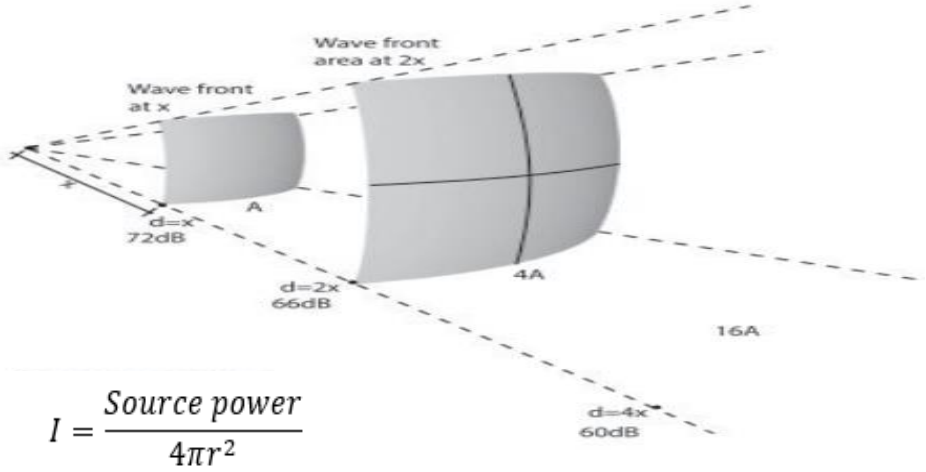


Figure 2. Inverse-square law (Patel, 2020)

However, while the inverse-square law can be used to predict distance attenuation, there are two assumptions made. Firstly, the inverse-square law assumes the source is of a point source, being a sound that radiates sound equally in all directions (Patel, 2020). Secondly, the inverse-square law assumes that the environment has free field conditions, being an environment where sound travels freely in all directions and is not impeded or encounters any reflective surfaces (Patel, 2020).

Furthermore, as sound propagates, it can interact with the environment in numerous ways, with some notable phenomena including reflection, diffraction, transmission, and absorption (Beig et al., 2019). By utilising geometrical acoustics, reflection and diffraction can be modelled as the rays interact with different geometry within the virtual environment. When a sound wave interacts with an obstacle, sound waves of greater acoustical size to the obstacles will diffract around the obstacle. The corners of the obstacle act as focal points for the diffraction process, creating a secondary wavefront of the same frequency but with a lower intensity (Firat et al., 2022). While this phenomenon is typically implemented within games empirically (Wratt, 2022), there have been more accurate approaches to calculating the properties of a sound when diffraction occurs.

There are two commonly used models for geometrical acoustical based diffraction: the Uniform Theory of Diffraction (UTD) (Kouyoumjian and Pathak, 1974, cited by Pisha 2020) and the Biot-Tolstoy-Medwin model (BTM) (Biot and Tolstoy (1957), Medwin (1981) and Svensson, Fred and Vanderkooy (1999), cited by Pisha 2020). Both models can be

described as edge diffraction models, which filter the sound on a path that circumvents one or more edges of a scene's geometry. According to Antani et al. (2010, cited by Pisha 2020), the UTD model is characterised by lower computational costs and reduced precision. Conversely, the BTM model offers more accuracy but requires greater resources, making it less suitable for commercial applications.

Consequently, geometrical acoustics has seen practical use within the games industry and is utilised in various plug-ins compatible with FMOD. A notable example being the *Oculus Audio Spatializer Plug-in for FMOD* (Meta Platforms, 2022), which utilises stochastic raytracing to compute the propagation of sound (Broderick et al., 2018).

Furthermore, there are methods utilised to increase the accuracy of other acoustical audio effects, as is the case with reverberation. Through utilising impulse responses to capture the reverberation characteristics of a room, the response can be applied to a convolution reverberation effect to emulate these characteristics in a digital realm (Rieger and Zingler, 2023). Additionally, the FMOD middleware contains a convolution reverberation effect that allows users to input their own recorded impulse responses (Firelight Technologies, 2023). Moreover, acoustical formulas such as Sabine's Equation, as shown in Figure 3., can utilise room dimensions and the acoustical properties of materials to calculate reverberation more accurately (Firat et al., 2022). *Resonance Audio* (Google VR, 2022) utilise this technique to determine reverberation for their spatial audio tool, which enables users to specify the absorptive characteristics of materials within a scene (Anemüller et al., 2023).

Sabine Equation

$$T_{RT60} = \frac{0.161 V_m}{S\,\bar{a}} = \frac{0.049\ V_f}{S\,\bar{a}}$$

T_{RT60} = RT 60 Reverberation Time, [seconds]
V_f = Volume of Room, [cu ft]
V_m = Volume of Room, [cu m]
S = Total Surface Area of Room, [sq ft] or [sq m]
$\bar{a}$ = Average Absorption Coefficient of Room Surfaces
$S\,\bar{a}$ = Total Absorption, [Sabins]

Figure 3. Sabine Equation (Abdullah et al., 2020)

Overall, while introducing the theory of sound within a physical environment, it is highlighted how there are many acoustical principles that combine to affect the properties of a sound as it travels through a space. Furthermore, the detailed acoustical phenomena and effects are not exhaustive and there are further principles that take precedence as sound interacts with the properties of a real-world environment. However, while there are numerous tools available that can simulate these principles, it could be argued that an acoustical environment achieved empirically could be more practical than acoustically modelling an

environment, especially in the video games industry which spans various platforms with differentiating device specifications and limitations (Beig et al., 2019).

2.5. Dialogue

2.5.1. Dialogue in video games

Dialogue can be vital in driving the illusion that a player's actions have an impact on the environment in which they are immersed (Taylor-Giles, 2022). Moreover, the design approach employed for a dialogue system can determine its ability to offer diverse narrative functionalities. Therefore, in addition to the requirement of initiating dialogue, further considerations may also be required to control the interaction between the dialogue system and other elements of a game, such as actor animations.

As highlighted by Stevens (2015), dialogue in video games is an area that has seen significant improvements within recent years. Despite this, creating a dialogue system that can cater to the vast amount of data required to handle how and when dialogue is triggered poses a significant challenge (Marx, 2021). As a result, in low-budget projects, many dialogue systems can exhibit a lack of precision when the narrative burden is placed on them (Bostan, 2020).

While dialogue systems can be developed to provide various degrees of functionality, as highlighted by Taylor-Giles (2022), the type of dialogues that a system could trigger could be categorised into one of three defining categories: informative, character, or ambient. As indicated by Taylor-Giles (2022), informative dialogue prioritises conveying key information, as opposed to mood or emotion. Unlike informative dialogues, character dialogues prioritise conveying emotion from the spoken character, as described by Murray (1997, cited by Taylor-Giles 2022). The final categorisation of dialogue types is ambient dialogues, which convey elements of an environment to the player and is critical for establishing the location that the player is immersed in (Marx, 2021).

Furthermore, there are numerous techniques to how dialogue assets are triggered by the game engine. As highlighted by Zdanowicz and Bambrick (2020), stitching is a method often employed by game studios to create more interactive dialogue sequences. Stitching can be described as the process of where different dialogue responses are associated through scripting by the game engine at runtime. Stitching can be used to sequence phrases which are recorded as whole lines, such as in *God of War Ragnarök* (Santa Monica Studios, 2022). However, stitching can also be used as a memory efficient method to pair short phrases together to create a common phrase which may have a small difference, such as a team name. This method of stitching short phrases together is often employed by Electronic Arts

in their sports games, such as *FIFA 23* (Electronic Arts Vancouver and Electronic Arts Romania, 2022).

Additionally, localisation is a further concept that a dialogue system must consider if a game is being shipped to multiple countries with varying languages (Zdanowicz and Bambrick, 2020). As stated by Bostan (2020), dialogue is a huge process and often requires a breadth of skill sets, such as audio recording and voice acting. Due to this, dialogue systems in video games will generally need to be able to provide a means of allowing for the localisation of dialogue by being able to replace all lines in a game to those recorded in the specified language (Stevens, 2015). Therefore, markup languages, such as Extensible Markup Language (XML), are often employed to maintain a vast collection of dialogue and its relating information (Somberg, 2017).

A more novel approach to dialogue implementation is utilising Natural Language Generation (NLG). While not widely adopted within the games industry (Ni et al., 2023), it has been utilised by research institutes for game development. A notable work being the Talk of the Town system (Ryan et al., 2016). This methodology for dialogue implementation uses a vast dataset of dialogues and employs computer generativity for interpreting user inputs to generate unique responses through the model provided.

In addition, Artificial Intelligence (AI) driven Large Language Models (LLMs) are becoming increasingly popular, being estimated to reach a market size of $9.4 billion by 2024 (Ni et al., 2023). An example of an LLM is *GPT-3.5* (OpenAI, 2022), which utilises Natural Language Processing (NLP) to provide a foundation for machine learning based language processing (Trivedi et al., 2023). Further iterations of the LLM have also been extended to support extended modalities, such as images, as seen with *GPT-4* (OpenAI, 2023).
However, as both NLG and LLMs are outside the scope of this paper, further research will be comprised of methodologies used for developing rule-based dialogue systems (Ni et al., 2023).

2.5.2. Branching dialogue systems

As highlighted by Zdanowicz and Bambrick (2020), a branching dialogue system is required to provide the player with an array of dialogue choices and progress the conversation depending on the players choice. To achieve this an eloquent framework is required to allow the game to access data pertaining to non-player characters and use these variables to dictate the flow of dialogue for a conversation (Marx, 2021). A dialogue system common to many Role-Playing Games (RPGs) is a branching system.

A branching dialogue system could be further categorised, depending on the provided functionality: system-centred, developer-centred, and player-centred.

As stated by Klein (1997, cited by Taylor-Giles 2022), a system-centred design is a method that favours gameplay over developer or player convenience. An example would be *The Elder Scrolls V: Skyrim – Anniversary Edition* (Bethesda Game Studios, 2021), where you can leave a non-narrative critical dialogue sequence at any time.
Another approach to branching dialogue is a developer-centred design. As indicated by Goldfarb and Kondratova (2004, cited by Taylor-Giles 2022), it favours the developer over player comfort, such as having a blank dialogue response option that serves no purpose but to progress the conversation.
Player-centred design is an approach that prioritises player immersion and narrative (Norman and Draper 1986, cited by Taylor-Giles 2022). The primary focuses of this type of branching system are conversational architecture and narrative direction.

To help visualise the interactions of a branching dialogue system, Figure 4. highlights a high-level overview of the flow of conversation for a choice-based branching dialogue system. While Figure 4. emphasises how a player's response drives the next node of a branching dialogue, the diagram does not show how certain responses or NPC dialogues may be condition dependent, requiring knowledge of in game data to be executable (Stevens, 2015).

The node like behaviour shown in Figure 4. can be described as a behaviour tree and has become a common methodology utilised for implementing branching dialogue in video games (Belle et al., 2019). As described by Champandard (2008, cited by Belle et al. 2019), behaviour trees act as a sequential graph that is composed of nodes behaving in an acyclic fashion. A behaviour tree is executed procedurally with different nodes being connected to other nodes.

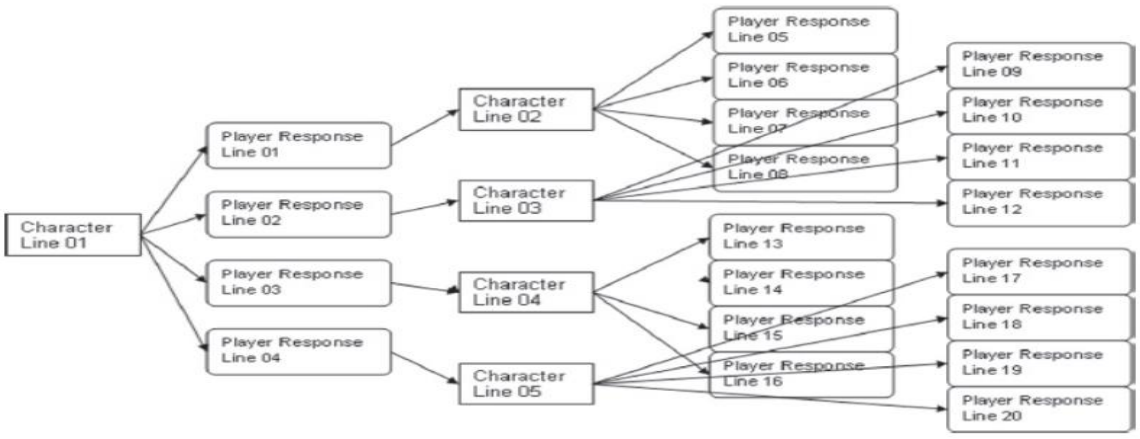


Figure 4. High level diagram of a branching dialogue systems architecture (Stevens, 2015).

While the procedural execution of a behaviour tree is shown in Figure 4., the acyclic behaviour is not highlighted. This is because acyclic is used to describe a graph with no cycle and therefore no repeated vertices (nodes); however, these exclude the start and end nodes. Therefore, a branching dialogue system would typically exit at the end node and re-enter at the start node, representing a link between these two node types (Belle et al., 2019). This branching methodology to implementing a dialogue system has become a common technique in game development due to its scalability and ability to handle as many responses as required (Subagyo et al., 2016).

3. Methodology

3.1. Introduction

The literature review for this paper focused on three topics of audio: procedural audio, spatial audio, and dialogue systems. Therefore, three independent systems have been developed for each area, with design decisions informed by findings established within the literature review.

For procedural audio, it was identified in Section 2.3.2. that utilising game data to automate audio effects is a commonly utilised method for adaptive audio (Wratt, 2018). Therefore, a series of utility scripts are developed that allow audio implementers an abstracted global method of utilising FMOD parameters to create both adaptive one-shot and looping audio events.

Section 2.3.1. highlighted how there have been recent advancements in spatial audio concerning third-party plug-ins that support more advanced audio simulation capabilities. However, the FMOD default spatialiser does not support audio occlusion or obstruction, which are found in recent audio innovations, such as *Resonance Audio* (Google VR, 2022). Therefore, a system is developed that simulates these acoustical phenomena.

Finally, Section 2.5.1. identified that rule-based dialogue systems are still the primary method utilised within the games industry. However, they are typically required to provide subtitle functionality and various trigger methods while allowing developers to invoke separate systems through the dialogue system. Furthermore, with genres such as RPGs, player response functionality may also be required. Therefore, a dialogue system is developed that attempts to provide all the aforementioned features.

3.2. Supporting procedural sound design through effect parameterisation

While FMOD provides various parameter types that can be used to automate effect parameters, an implementation method must be devised to create an appropriate pipeline for setting FMOD parameters within the game engine. Furthermore, FMOD parameters only support primitive data types. Therefore, using generics to further encapsulate the FMOD API, which relates to parameter handling, into a concise series of globally accessible functions could benefit technical implementers.

As illustrated in Figure 5., generics are utilised in the one-shot system to allow a more concise implementation that enables one function to handle all parameter types. As described by Microsoft (2023), generics allow for classes and methods to be implemented that defer the specification of the type until the class or method is instantiated or called from another part of the program. Therefore, by utilising generics, the user of the system can provide any of the supported data types, without needing to consider which function to call, as illustrated in Figure 6.

```
public static void PlayOneShotWithParameters<T>(FMODUnity.EventReference fmodEvent, Transform transformToAttachTo, params (string name, T value)[] parameters)
{
    FMOD.Studio.EventInstance instance = FMODUnity.RuntimeManager.CreateInstance(fmodEvent);

    foreach (var (name, value) in parameters)
    {

        //If param value is of type string, set as labeled param
        if (value.GetType() == typeof(string))
        {
            instance.setParameterByNameWithLabel(name, value.ToString());
        }

        //If param value is of type float or int, set param as continous or discrete
        else if (value.GetType() == typeof(float) || value.GetType() == typeof(int))
        {
            dynamic paramVal = value; //Use dynaic type so it can be int or float, without needing another if statement
            instance.setParameterByName(name, paramVal);
        }
    }

    //Check if position has been given to attach event to that position and make 3D
    if (transformToAttachTo != null)
    {
        FMODUnity.RuntimeManager.AttachInstanceToGameObject(instance, transformToAttachTo);
    }

    instance.start();
    instance.release();

}
```

Figure 5. Function as part of the static utility toolset to support different types of procedural audio.

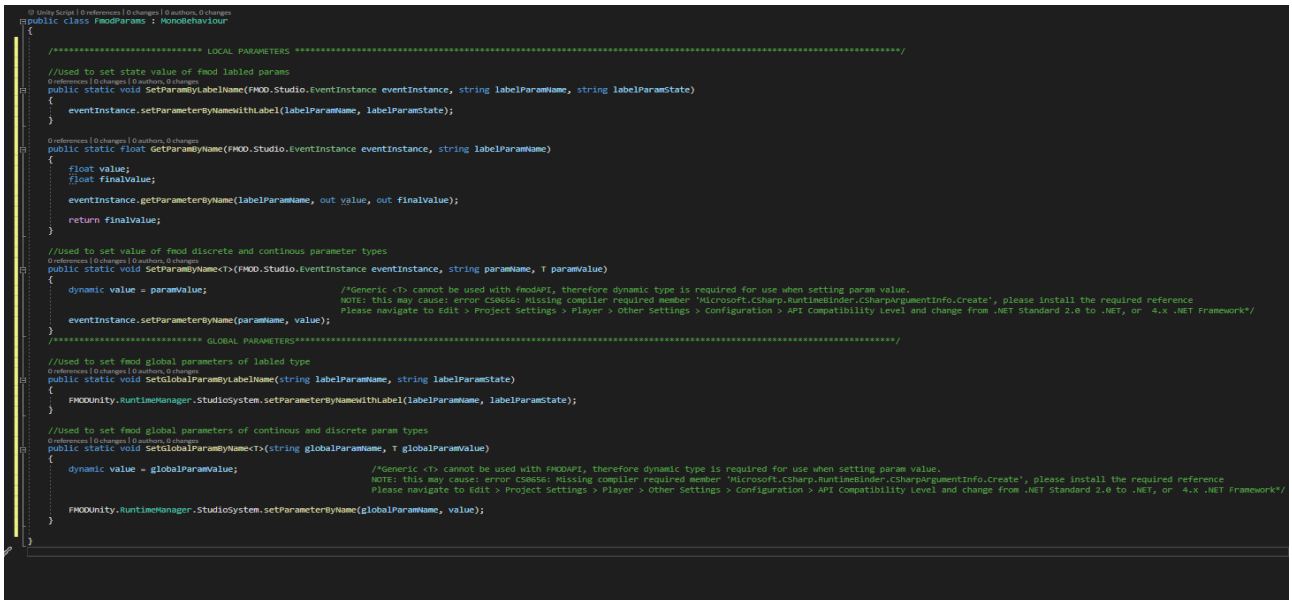
```
AudioPlayback.PlayOneShotWithParameters(FmodEventReferences.Instance.gunShotSFX, null, ("Surface", terrainDetector.DetermineShellTerrain().ToString()));
```

Figure 6 Example of one-shot parameter call utilising the one-shot system.

As highlighted by McIntosh et al. (2022), while music is often non-diegetic, repetitive music can quickly become tiresome. Therefore, providing sound designers with a means of adapting music and other looping audio could facilitate a method to overcome this. While FMOD does provide a means of triggering looping audio without needing to program the functionality, this method limits how adaptable looping events can be. Additionally, it only supports stopping and starting a script, as shown in Figure 7.

Figure 7. Studio emitter script provided by the FMOD for Unity API.

Moreover, an FMOD parameter can either be local or global. Therefore, the utility scripts provide the ability to adapt both these parameter types. Furthermore, the system can obtain parameter values to provide further functionality to the system, as highlighted in Figure 8. Additionally, references to the FMOD event can be obtained manually by creating an instance or through the FMOD event emitter script, provided as part of the FMOD API.



```
public class FmodParams : MonoBehaviour
{

    /*************************** LOCAL PARAMETERS ***************************************************************************************************************/

    //Used to set state value of fmod labled params
    public static void SetParamByLabelName(FMOD.Studio.EventInstance eventInstance, string labelParamName, string labelParamState)
    {
        eventInstance.setParameterByNameWithLabel(labelParamName, labelParamState);
    }

    public static float GetParamByName(FMOD.Studio.EventInstance eventInstance, string labelParamName)
    {
        float value;
        float finalvalue;

        eventInstance.getParameterByName(labelParamName, out value, out finalvalue);

        return finalvalue;
    }

    //Used to set value of fmod discrete and continous parameter types
    public static void SetParamByName<T>(FMOD.Studio.EventInstance eventInstance, string paramName, T paramValue)
    {
        dynamic value = paramValue;                         /*Generic <T> cannot be used with fmodAPI, therefore dynamic type is required for use when setting param value.
                                                            NOTE: this may cause: error CS0656: Missing compiler required member 'Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo.Create', please install the required reference
                                                            Please navigate to Edit > Project Settings > Player > Other Settings > Configuration > API Compatibility Level and change from .NET Standard 2.0 to .NET, or  4.x .NET Framework*/
        eventInstance.setParameterByName(paramName, value);
    }
    /*************************** GLOBAL PARAMETERS***************************************************************************************************************/

    //Used to set fmod global parameters of labled type
    public static void SetGlobalParamByLabelName(string labelParamName, string labelParamState)
    {
        FMODUnity.RuntimeManager.StudioSystem.setParameterByNameWithLabel(labelParamName, labelParamState);
    }

    //Used to set fmod global parameters of continous and discrete param types
    public static void SetGlobalParamByName<T>(string globalParamName, T globalParamValue)
    {
        dynamic value = globalParamValue;                       /*Generic <T> cannot be used with FMODAPI, therefore dynamic type is required for use when setting param value.
                                                                NOTE: this may cause: error CS0656: Missing compiler required member 'Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo.Create', please install the required reference
                                                                Please navigate to Edit > Project Settings > Player > Other Settings > Configuration > API Compatibility Level and change from .NET Standard 2.0 to .NET, or  4.x .NET Framework*/

        FMODUnity.RuntimeManager.StudioSystem.setParameterByName(globalParamName, value);
    }
}
```

Figure 8. Utility script that allows users of the system to handle FMOD event parameter functionality.

3.3. Audio Obstruction and Occlusion system

FMOD provides a series of acoustical effects that can be applied to audio to emulate acoustical phenomena, such as the doppler effect and reverberation. However, the FMOD built-in spatialiser is unable to calculate when soundwaves encounter an obstacle, which is required to implement other acoustical effects highlighted in Section 2.4.2. Therefore, an audio obstruction and occlusion system is developed to extend the capabilities provided by the FMOD spatialiser.

3.3.1. Emulating propagation with linecasts

As highlighted by Firat et al. (2022), integrating raytracing to emulate the direction a soundwave is travelling and map its propagation is a method that has been utilised within the games industry. However, while there are various types of raytracing, the technology utilised for this system is raycasts, which can indicate what obstacles a sound wave encounters between its source and the listener (Unity Technologies, 2023). This method can be used to detect when a player is behind an obstacle that is in the direct path of an audio emitter, to then apply occlusion.

The type of raycast utilised for the development of this occlusion system is a linecast. As described by Unity Technologies (2023), a linecast can detect if there are any colliders between the specified start and end point of the linecast. Furthermore, linecasts can ignore layers and provide a means of dictating which layer types to detect a collision with. This is shown in Figure 9., as gizmos are used to draw lines while in the editor. This aims to enhance implementers debugging capabilities when implementing the tool. Rays that do not intersect with an obstacle between the emitter and listener are indicated as a green line, as shown in Figure 10., while rays which encounter an obstacle on their path will be indicated as red lines.

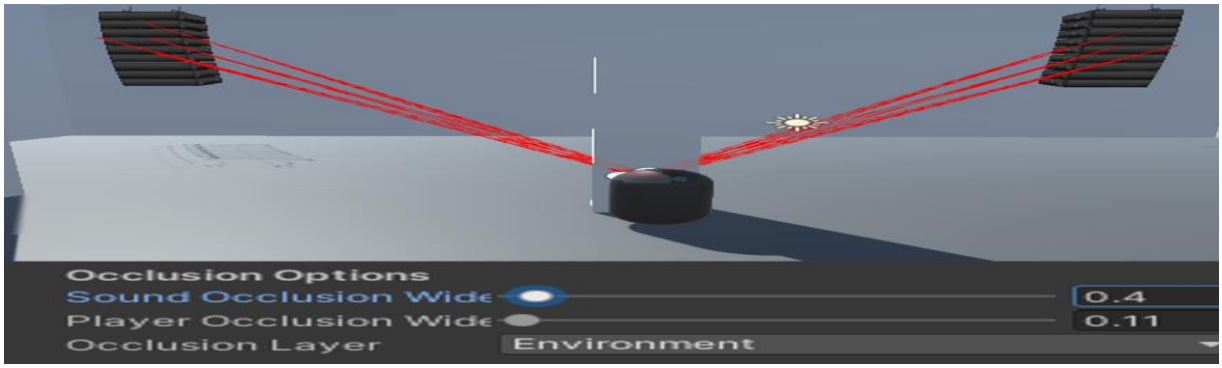


Figure 9. Occlusion system being used to emulate a small sound size.

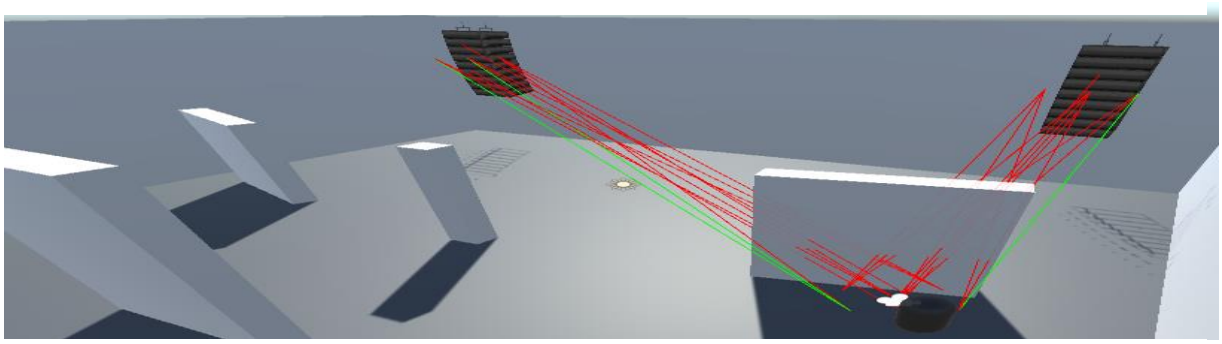


Figure 10. Occlusion system being used to emulate a larger sound size, with a higher sound occlusion widening setting.

However, sound emitting within the real world is either of a point source or line source. Point sources radiate sound equally in all directions, while line sources radiate sound cylindrically in a one-dimensional line (Beig et al., 2019). While this system behaves more like a line source, as the nature of raycasts is that of a straight line, it should be noted that that the player is the end destination supplied to generate the lines. Therefore, if the player moves out of the direct line of the sound sources' direction the lines will still follow the player, and this behaviour is not how a line source acts within the real world. Nevertheless, according to the findings of Davari and Verbrugge (2021), while there are computational methods available to render sound more accurately within a virtual environment, the prevalent approach is to employ audio effects to manipulate the acoustic characteristics empirically. This approach is preferred due to faster implementation and less resource dependency (Wratt, 2018).

Furthermore, another factor to take into consideration is the size of a sound and whether a sound is partially obstructed or fully occluded. Therefore, sound size is an aspect that is implemented into this system in the form of the 'Sound Occlusion Widening' parameter (Figure 9.). An indication of a smaller sound size is shown in Figure 9, while a larger sound size is shown in Figure 10. This parameter attempts to emulate sounds of different sizes, with a higher 'Sound Occlusion Widening' value increasing the total width that the linecasts are radiated in.

3.3.2. Calculating the third vertex in an orthogonal triangle

Subsequently, to provide the widening functionality, a formula for calculating the third vertex in an orthogonal triangle was employed. While the start and end positions of the rays can be automatically calculated, this would create a narrow cast with no width. To support the ability to change the sound size, the equation illustrated in Figure 12. was utilised to calculate the third vertex. The 'Sound Occlusion Widening' parameter, as shown in Figure 9., is the key variable responsible for calculating the third vertex of the triangle, which is vertex '(x, y)' as shown in Figure 11. The vertex marked '(x1, y1)' represents the sound source where the sound originates, while the vertex marked '(x2, y2)' represents the player position, and the distance between the player and the sound source is indicated by the line labelled 'n'. However, the '(x, y)' vertex is used to represent the maximum width that the rays will be cast in, dictated by the designer set parameter for sound occlusion widening. Furthermore, the distance between the sound source and this position is indicated by the line labelled 'm'.

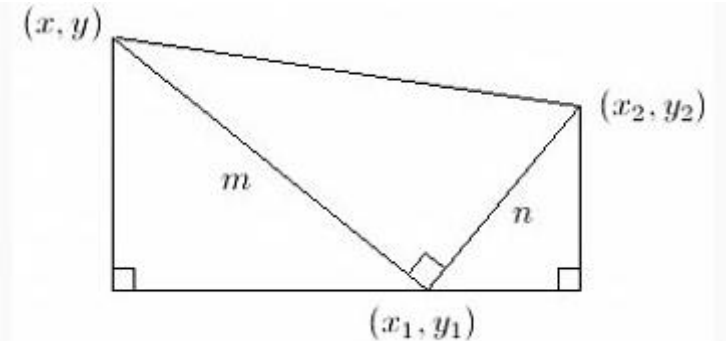


Figure 11. Illustration of vertex points in an orthogonal triangle (Croft and Davison, 2018).

Calculating y: $y = y_1 \pm \frac{m}{n}(x_1 - x_2)$

Calculating x: $x = x_1 \pm \frac{m}{n}(y_2 - y_1)$

Calculating (x, y) = ($x_1 \pm \frac{m(y_2 - y_1)}{n}$, $y_1 \pm \frac{m(x_1 - x_2)}{n}$)

Figure 12. Calculations used for occlusion widening.

As highlighted in Figure 11., the sound source '(x1, y1)' and the player position '(x2, y2)', are both already calculated by the linecasts. Therefore, the only vertex that requires a separate implementation is the '(x, y)' vertex position, which is the greatest width the lines can be cast in.

While in Figure 11. the emitter vertex is labelled '(x1, y1)', for the implementation of the occlusion system, the 'y' axis has been replaced with the 'z' axis (Figure 13.) for the calculations done within Unity at runtime, provided by the occlusion script. As shown in Figure 13., the 'CalculateXZVector' function was implemented to calculate the widening vector. This function requires four arguments to be able to calculate this vector. The 'a' variable represents the vector of the sound source, while the 'b' variable represents the player vector, and 'm' is the widening factor set by the designer within the Unity inspector for the script. The final argument is the Boolean 'absolute', which is used to dictate whether to subtract or plus with the ± operator.

```
private Vector3 CalculateXZVector(Vector3 a, Vector3 b, float m, bool absolute)
{
    float x;
    float z;
    float n = Vector3.Distance(new Vector3(a.x, 0f, a.z), new Vector3(b.x, 0f, b.z));
    float mn = (m / n);
    if (absolute)
    {
        x = a.x + (mn * (a.z - b.z));
        z = a.z - (mn * (a.x - b.x));
    }
    else
    {
        x = a.x - (mn * (a.z - b.z));
        z = a.z + (mn * (a.x - b.x));
    }
    return new Vector3(x, a.y, z);
}
```

Figure 13. Algorithm implemented to calculate the sound widening vector for the occlusion system.

3.3.3. Occluding or obstructing the sound source

With a methodology for calculating the widening vector for the occlusion system, the calculation of the occlusion can be applied. However, as highlighted by Anemüller et al. (2023), it is important to note that if all sound waves comprised within a sound are blocked by a scene's geometry from reaching the player, then the sound is occluded. While if only a partial amount of sound waves are obstructed, then it can be described as obstruction. To achieve this, a procedural methodology is used to adapt the cut-off of a low pass filter added as an effect to the audio event inside of FMOD. Any audio event employing the occlusion system will utilise a total of eleven linecasts to calculate how much the audio should be filtered, to emulate occlusion or obstruction. As highlighted in Section 2.3.2., FMOD provides various in-built parameter types that can be utilised to feed dynamic data to apply automation to effect parameters.

For this system, a parameter type that is of a continuous or floating type is required to be made inside of FMOD. For the tutorial scene, provided with the package, a continuous parameter was created inside of FMOD called ‘Occlusion’. While this is an obvious name for the parameter, any name can be used and that name will be entered into the ‘Parameter Name’ variable within the inspector by the audio implementer, as highlighted in Figure 14.

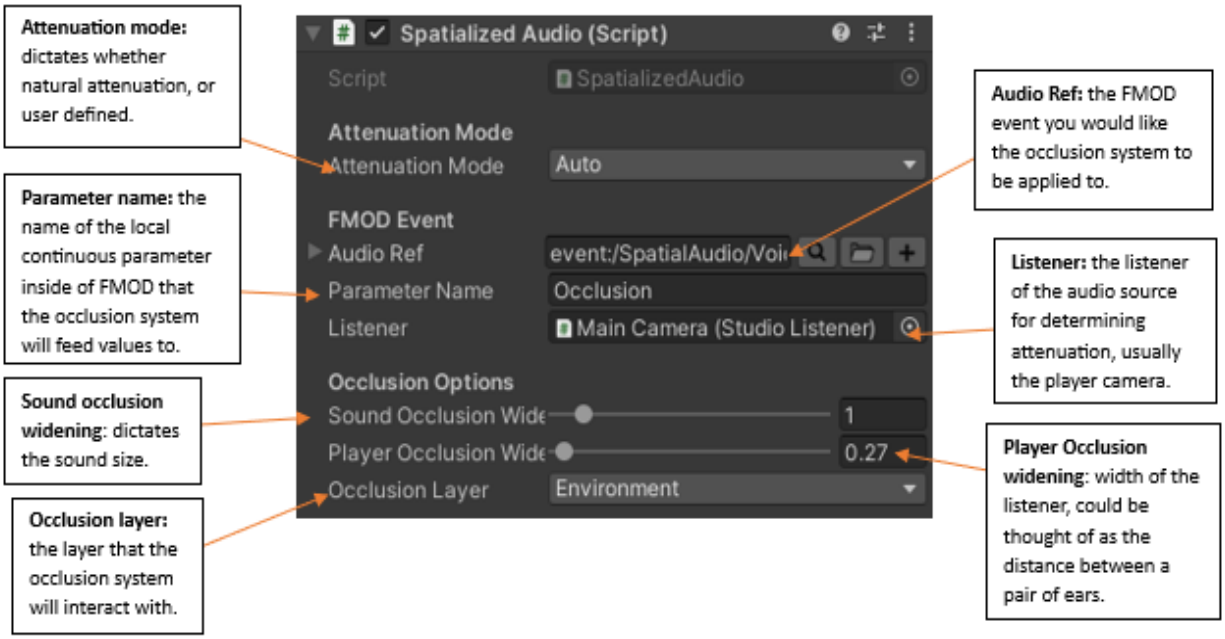


Figure 14. Inspector view of the Spatialised Audio script, provided with the Occlusion system.

With this occlusion calculation in place, it can be used to adapt the cut-off of the low-pass filter within FMOD. The low-pass cut-off attenuation curve used in the example provided as part of the package is shown in Figure 15.

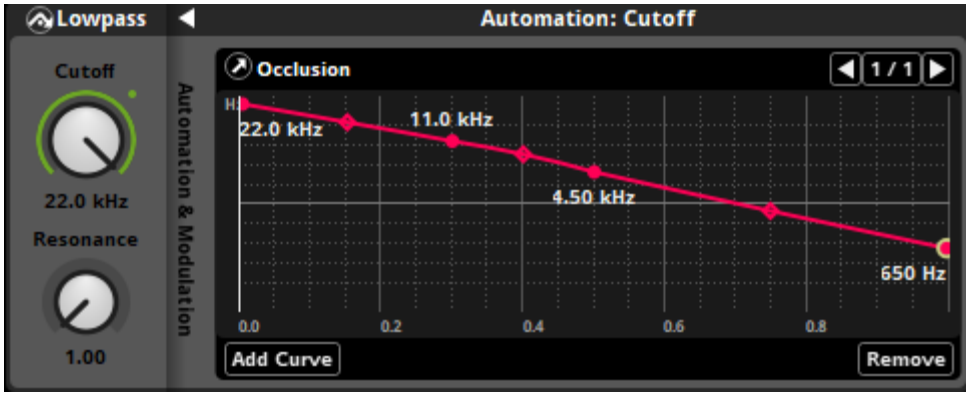


Figure 15. Automation curve for the low-pass cut-off value, automated by the occlusion parameter.

As shown in Figure 15., the parameter values range from zero to one. To calculate the occlusion value inside of FMOD at a given point, the amount of linecasts reaching the player is divided by the total number of linecasts provided by the system. The sum of this calculation will range between zero and one. Zero indicates that all linecasts are reaching the player, while a value of one indicates that no linecasts are reaching the player as they are all being obstructed. The occlusion parameter, created inside FMOD, can also be used to adapt other effect parameters. In the example provided with the package, the occlusion parameter is also used to automate the volume of an audio event by utilising the occlusion system.

3.4. Dialogue System

3.4.1. Introduction

As highlighted in Section 2.5.2. dialogue systems can be made to a variety of specifications, boasting different capabilities depending on how they were implemented. Furthermore, dialogue systems can be as simple as triggering a dialogue like a sound effect. However, as the quantity of dialogues increase, the maintainability of a system of this architecture decreases. Somberg (2017) indicated that one of the most powerful tools available to a technical sound designer are markup languages. A common markup language utilised for audio implementation is XML. As described by Somberg (2017), XML is a crucial aspect of any technical audio implementers toolset and can be utilised to create data that can be deserialised by a wide variety of system in numerous programming languages. Therefore, creating a system that can trigger dialogue based off XML file information will be a key aim for implementing this dialogue system.

Overall, the aim for developing this dialogue system is to provide sound designers with a method of easily creating a large quantity of XML files that can be integrated into Unity to handle how and when dialogue should be triggered. Furthermore, as highlighted by Andersen et al. (2021), immersion has become a vital factor in modern audio implementation. Therefore, dialogue XML files created by this system should be able to respond to various types of game data and utilise this to dictate if a certain line should be triggerable at a given context. Bostan (2020) stated that branching dialogue has become a common feature for dialogue systems, especially in the RPGs. Therefore, player choice-based branching will be another key factor in the development of this dialogue system. While the dialogue system's aim is to be a tool primarily for sound designers and technical sound designers, Belle et al. (2019) highlighted how dialogue systems can also be a means of driving other game events, such as starting a quest when finishing a quest-related dialogue.

Therefore, the architecture of the dialogue system should support the modularity of decoupled game design architecture and provide a means for executing other game functionality, separate from the dialogue system, as defined by the game designers.

3.4.2. XML Application

While it has been discussed that markup languages are a powerful tool for any technical sound designer, this paper aims to create a series of tools that can be used by both aspiring and more experienced sound designers. Therefore, requiring users of the system to learn XML in the context that it will be used within this system adds a barrier to entry and could limit the audience. Hence, an XML application has been developed to ensure that files utilised with this system match the schema of the XML deserialisation system developed within Unity. This XML application is a standalone application built in C#, employing the *Windows Form Framework* (Microsoft, 2023) to provide some generic user interface objects, such as buttons and text boxes. However, further functionality was added to the application to increase its practicality when being used as a tool. This is detailed in Figure 16.

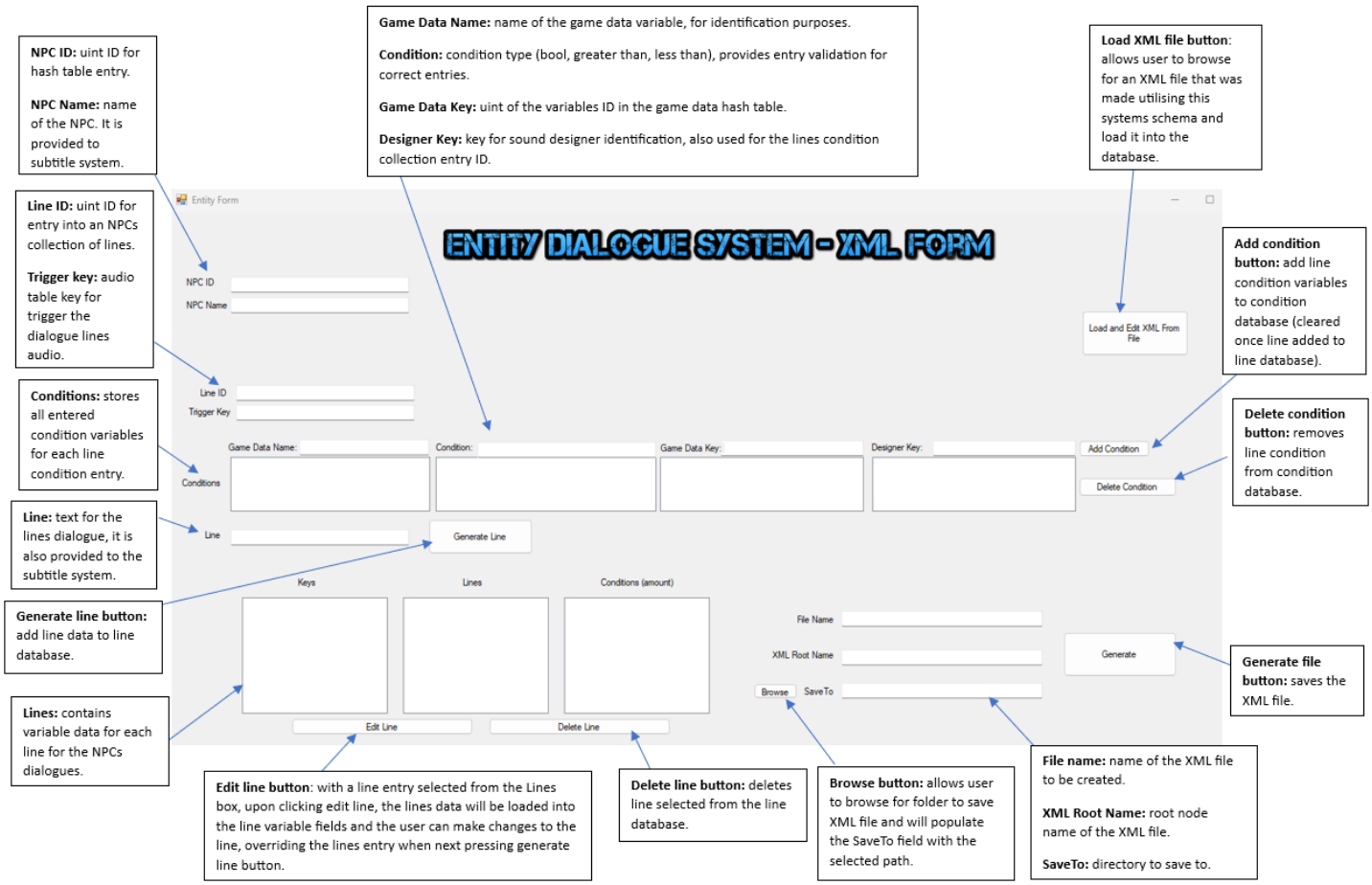


Figure 16. Functionality provided by the XML application.

Furthermore, it should be noted that users of the system are not required to use this XML application, and if they have the required knowledge for utilising XML, they can write their own files or use third-party software, such as *Microsoft Excel* (Microsoft, 2021). However, utilising the application made as part of this system will ensure that there can be no authoring errors when writing the XML files. Therefore, files generated with this system are guaranteed to be properly deserialised within Unity. The XML application allows users to fill in all fields that are part of the deserialisation system within Unity. It also supports an unlimited number of line conditions and provides exception checking to ensure entries are legal. Additionally, the XML application boasts further functionality, such as being able to edit already inputted lines and overwrite them, as well as providing a browser to facilitate saving generated files to the streaming assets of a Unity project. The application also supports the loading of pre-existing files into its database and allows users to edit a document from file.

The architecture for this system was designed to support a method for triggering audio based upon parameter conditions, also known as Real Time Parameter Controls (RTPCs) (Somberg, 2017). RTPCs are utilised to be able to interact with a database and contain unique identifiers, so that each entry into a database can be individually identified. This methodology is commonly used for complex interactions between a sound and game data (Somberg, 2017). Game developers can add keys to their own game data database, while sound designers can author their own parameters with unique identifiers that are paired with a corresponding game data key and are stored within a separate database. The result is two systems that can be maintained independently while being able to interact with each other through unique identifiers, as highlighted in Figure 17.

TABLE 17.3 Authoring of GameDataID and RTPC-ID Pairs

GameData-RTPC	GD-RTPC ID	RTPC ID	GameData ID
HealthRatio	3001	2001	1002
Velocity	3002	2002	1004
AngularVelocity	3003	2003	1001

TABLE 17.4 Authoring of GameData Subscription of Two Sounds

Sound Name	GD-RTPC ID
LaserSword	3003 (AngularVelocity)
ShieldOrb	3001 (HealthRatio)
	3002 (Velocity)

Figure 17. Example of RTPC approach to authoring a large collection of sounds that can interact with game data (Somberg, 2017)

While the development of this type of data handling was inspired by RTPCs methodology, the field names implemented into the XML system differ and are named to represent different elements of the dialogue system. To aid with dialogue handling, FMOD provides audio tables, which are a method to trigger a sound that is separate from the FMOD sound banks using a trigger key (Firelight Technologies, 2023). Therefore, as shown in Figure 16. the field of the system responsible for triggering the dialogue from the audio is the line's

'Trigger Key'. The identifier for a database entry is the 'NPC ID' (Figure 16.), while the 'NPC Name' value is just to increase readability when looking at the XML file and to be used as data in the subtitle system. A line entry for an NPC can only have one 'Line ID', 'Trigger Key' and 'Line', however a line can have an unlimited number of conditions that are used to determine whether the dialogue should be triggered at a given point. There are currently four condition types supported: greater than, less than, true, and false. Inputs are checked to ensure that entries work with the deserialisation system developed within Unity. An example of an XML file generated through this system is shown in Figure 18.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<Entity ID="900187456" Name="Dorothy">
  <Lines>
    <Line id="10683">
      <Key>Doro-Pst</Key>
      <LineText>Pssst... Over here!</LineText>
      <Conditions>
        <Condition id="901050">
          <TriggerCondition>True</TriggerCondition>
          <GameDataName>QuestAccepted</GameDataName>
          <GameDataKey>2</GameDataKey>
        </Condition>
        <Condition id="101">
          <TriggerCondition>False</TriggerCondition>
          <GameDataName>DoraSpokenTo</GameDataName>
          <GameDataKey>7</GameDataKey>
        </Condition>
      </Conditions>
    </Line>
    <Line id="80901">
      <Key>Doro-OverHere</Key>
      <LineText>Yes you, over here!</LineText>
      <Conditions>
        <Condition id="9010">
          <TriggerCondition>True</TriggerCondition>
          <GameDataName>QuestAccepted</GameDataName>
          <GameDataKey>2</GameDataKey>
        </Condition>
        <Condition id="100230">
          <TriggerCondition>False</TriggerCondition>
          <GameDataName>DoraSpokenTo</GameDataName>
          <GameDataKey>7</GameDataKey>
        </Condition>
      </Conditions>
    </Line>
    <Line id="12345901">
      <Key>Doro-WorthIt</Key>
      <LineText>Over here, it will be worth your time!</LineText>
      <Conditions>
        <Condition id="309018">
          <TriggerCondition>True</TriggerCondition>
          <GameDataName>QuestAccepted</GameDataName>
          <GameDataKey>2</GameDataKey>
        </Condition>
        <Condition id="10023">
          <TriggerCondition>False</TriggerCondition>
          <GameDataName>DoraSpokenTo</GameDataName>
          <GameDataKey>7</GameDataKey>
        </Condition>
      </Conditions>
    </Line>
  </Lines>
</Entity>
```


Figure 18. An XML file generated through the application.

Overall, the XML applications provides numerous functionalities, while providing an accessible way of creating XML files for the dialogue system within Unity. However, the database entries are comprised of multiple objects. Therefore, knowledge of this data is required to understand the workings of the system and how it is deserialised within Unity.

3.4.3. Deserializing XML files into Unity

With an XML file structure implemented, a deserialisation method could be developed within Unity to deserialise files into data objects. This could then be sorted into a dialogue hashtable, with its identifier being the ‘Entity ID’ from the XML file (Figure 18.).

The data structures developed were utilised as a container for XML data, which is shown in Figure 19., highlighting how numerous objects combine into a single ‘Entity’ object. This methodology was employed to give greater organisational capabilities to the system and to provide collections that could present line data and specific condition data. This data is accessed through the ‘conditions’ dictionary, which has its identifier value set from the condition’s ‘Designer Key’ from the XML application, shown in Figure 16. Deserialisation of an XML file is handled on an object-to-object basis, with each instance of a ‘Dialogue Entity’ script providing a user selected XML file path to a static file handling class. This static file handling class utilises the *LINQ XML Library* (Microsoft, 2021) to deserialise data into the object containers shown in Figure 19. This returns an ‘Entity’ object containing all sub-objects and data relating to a NPC’s entry into the dialogue hashtable.

```
16 references | samsquanchone, 3 days ago | 1 author, 4 changes
public class Line
{
    public uint lineID;
    public string key;
    public string line;
    public Dictionary<uint, Condition> conditions;

    1 reference | samsquanchone, 3 days ago | 1 author, 3 changes
    public Line()
    {
        conditions = new Dictionary<uint, Condition>();
    }

}

//Used to define condition info for each line so we can store it in the hash table
6 references | samsquanchone, 3 days ago | 1 author, 3 changes
public class Condition
{
    public string gameDataName;
    public ConditionalLogicType triggerCondition;
    public dynamic conditionValue;
    public uint gameDataKey;
}

//Used to store all sub-objects of an NPCs dialogue info into one object that can be stored into the dialogue hastable
[Serializable]
17 references | samsquanchone, 3 days ago | 1 author, 3 changes
public class Entity
{
    public uint id;
    public string name;
    public Dictionary<uint, Line> lines;

    1 reference | samsquanchone, 3 days ago | 1 author, 3 changes
    public Entity(string _id, string _name)
    {
        id = uint.Parse(_id);
        name = _name;

        lines = new Dictionary<uint, Line>();
    }
}
No issues found
```

Figure 19. Data structure for the dialogue system.

Although XML files could be deserialised into data objects, all data provided from the XML files are automatically deserialised into strings. However, this dialogue system aims to be able to handle varying parameter types; therefore, condition values would need to be validated for their specific data type and then be parsed into that type. Figure 19. highlights the application of the dynamic data type used with the 'conditionValue' variable. As described by Microsoft (2023) a dynamic type acts like a static type but avoids static type checking and assumes that the type supports any operation. Therefore, the condition can be trimmed and then attempted to be parsed into a data type supported by the dialogue system; this is done once on deserialisation. This functionality is handled in a static string validation script and utilises generics to be able to return any data type from the function, as shown in Figure 20.

```
public static T ConvertStringToDataType<T>(string value)
{

    if (int.TryParse(value, out int intVal))
    {
        return (T)(object)intVal;
    }
    if (float.TryParse(value, out float floatVal))
    {
        return (T)(object)floatVal;
    }
    if (bool.TryParse(value, out bool boolVal))
    {
        return (T)(object)boolVal;
    }
    else
    {
        //Debug Log that data type is not supported
        Debug.Log("Data type not supported! Please use a supported data type: float, int and string");
        return (T)(object)value;
    }
}
```

Figure 20. Function that validates an XML condition value and parses it into its respective type.

Furthermore, as the dialogue system supports condition types, further validation is required to associate a condition with an Enum representation of its detected condition type. This Enum value is shown in Figure 19. and is the 'triggerCondition' variable within the 'Condition' object; like the data parsing, this is done on deserialisation of a file. Therefore, once a condition's trigger type is associated with an Enum value, the condition can be queried against game data conditions, and apart from whether the condition is met or not, it requires no further checks. This static method is handled in a static class and serves the function of trimming the string before the non-parsed condition value is passed to the parsing function shown in Figure 20. This process identifies the condition type within the provided string and then trims the string accordingly to just be a value, instead of a condition with a value. The process implemented within this function is highlighted in Figure 21.

```
public static class StringValidation
{
    const string lessThan = "<";
    const string greaterThan = ">";

    1 reference | samsquanchone, 24 days ago | 1 author, 1 change
    public static ConditionalLogicType GetConditionLogicType(string conditionString, out string trimmedString)
    {
        ConditionalLogicType conditon = ConditionalLogicType.Unknown;

        if (conditionString.Contains(lessThan))
        {
            conditon = ConditionalLogicType.LessThan;
            conditionString = conditionString.Trim('<');
        }
        if (conditionString.Contains(greaterThan))
        {
            conditon = ConditionalLogicType.GreaterThan;
            conditionString = conditionString.Trim('>');
        }

        //Try parse bool
        if (bool.TryParse(conditionString, out bool result))
        {
            //If bool val true set to condition true type and vice versa
            if (result)
            {
                conditon = ConditionalLogicType.True;
            }
            else
            {
                conditon = ConditionalLogicType.False;
            }
        }
        else
        {
            trimmedString = conditionString;
        }

        trimmedString = conditionString;

        return conditon;
    }
```

Figure 21. Function developed to detect condition type and set the Enum value accordingly.

The current deserialisation system has achieved the capability to successfully extract all data from an XML file and effectively convert string values into their corresponding data types. Consequently, it becomes feasible to evaluate conditions specified in a dialogue file against the relevant data values stored in the game data database. This facilitates the development of a system that can compare dialogue conditions with dynamic game data values.

3.4.4. Dialogue Entity

As highlighted in Section 2.5.1., dialogue systems may be responsible for triggering code that is outside the scope of a dialogue system. However, each actor utilising a dialogue system may require different levels of control over the dialogue system. Therefore, the ‘Dialogue Entity’ script was developed to not only allow users to select an XML file to deserialise, but also to provide an interface within the inspector that provides full control over different modes and functions provided with the system, as well as providing a means of invoking separate systems. An annotated diagram overviewing all elements of the ‘Dialogue Entity’ system is provided in Figure 22.

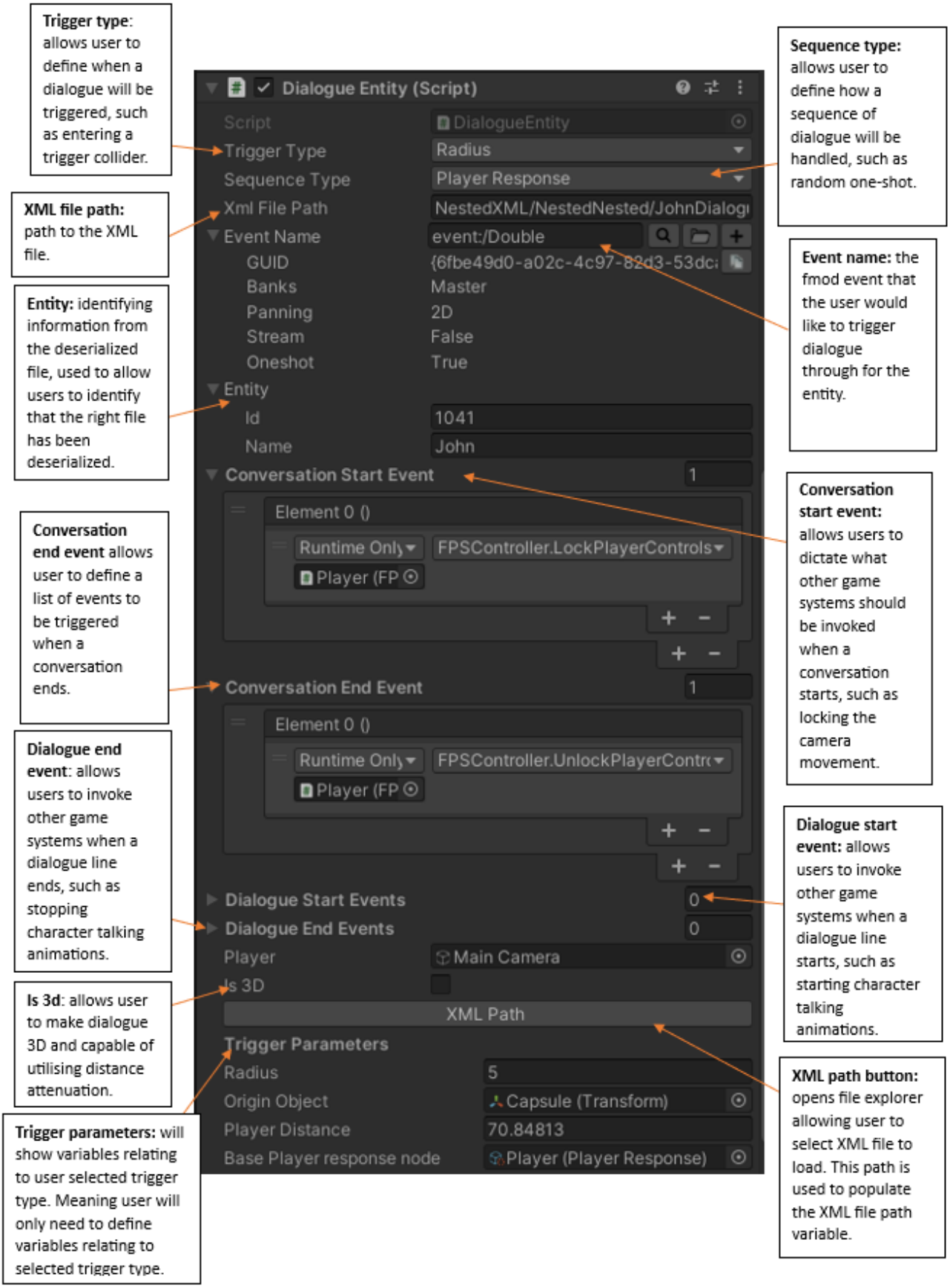


Figure 22. Example of 'Dialogue Entity' object during runtime, being used in the player response sequence mode.

Figure 22. highlights how there are numerous components comprised within the ‘Dialogue Entity’ system. However, it should be noted that an instance of this system represents a single XML file for an actor. Therefore, a single actor can have multiple instances of this system, allowing users to dictate the behaviour of different dialogue sequences. Currently the system supports four different trigger types: trigger exit, trigger enter, radius and collision. As presented in Figure 22., the trigger parameters section of the system is where the user will dictate the variables for the selected trigger type (Figure 24.), which can provide debug gizmos (Figure 23.).

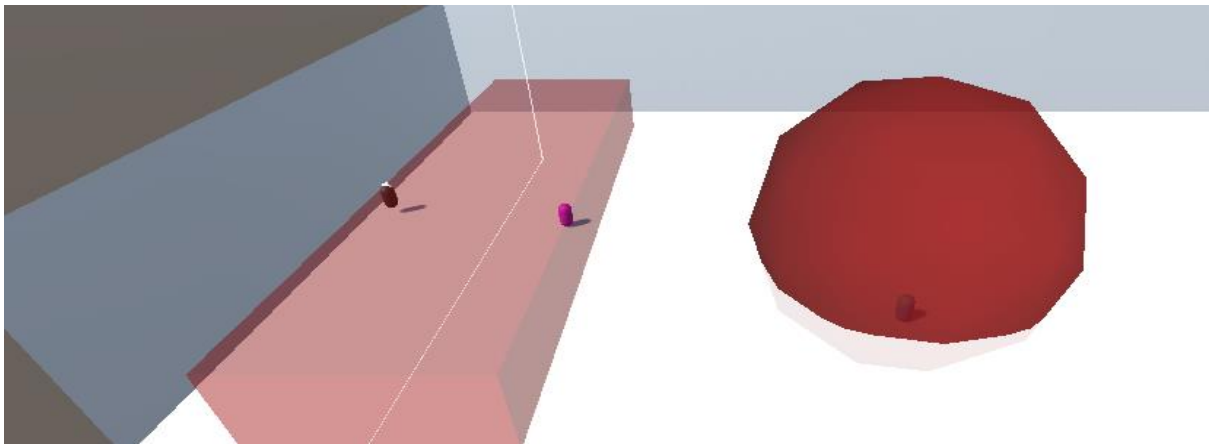


Figure 23. Debugging gizmos drawn for Entity systems utilising trigger enter and exit trigger types.

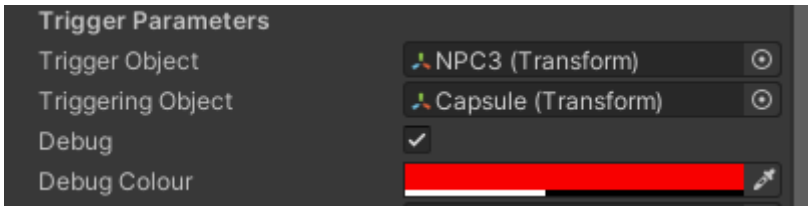


Figure 24. Further debugging features provided through the system.

3.4.5. Game data resolver

For the dialogue system to be capable of querying its conditions against game data values, located inside the game data hashtable, a system was required to allow for the dialogue system to be able to compare game data values against a line’s trigger conditions. This was achieved by developing a game data resolver, which provides game designers with the ability to add and update game data in the database and provides a means for triggering conditions. While the rest of the dialogue system can be handled entirely from the inspector, the entries into the game data hashtable are required to be handled by the game developers of the project that the dialogue system is being used in. However, once entries have been added, no further considerations are required from a programming perspective, as querying and condition checking for dialogue lines are automatically handled in the background. The main functionality of the game data resolver is illustrated in Figure 25.

```
public void SetGameDataVariable<T>(uint key, in T inValue)
{
    GameDataValue value = (GameDataValue)gameDataHashTable.hashTable[key];

    value.m_current = inValue;
    value.m_set = true;

}

1 reference | samsquanchone, 29 days ago | 1 author, 1 change
public GameDataReturnType QuerryGameData<T>(uint key, out T outValue)
{
    //Create an instance of gamedata value so we can cast the hash table value to it
    GameDataValue value = (GameDataValue)gameDataHashTable.hashTable[key];

    if (value != null)
    {

        if (value.m_set)
        {
            outValue = value.m_current;

            return GameDataReturnType.GameDataOk;
        }

        else
        {
            outValue = value.m_default;
            return GameDataReturnType.GameDataDefault;
        }
    }
    else
    {
        //Return default of the passed type, as we cant assume the type as it is generic
        outValue = default(T);
        return GameDataReturnType.GameDataUnknown;
    }
}
```

Figure 25. Game data resolver, responsible for querying game data and allowing game designers to and entries to the game-data hashtable.

Like the 'Condition' object (Figure 19.), the 'GameDataValue' object (Figure 25.), that is stored as an entry in the game data hashtable, utilises a dynamic type for its condition, allowing generics to be provided to the resolver. This class is implemented utilising the singleton design pattern, meaning that there can only be one instance of the resolver per scene (Nystrom, 2014). However, the class is globally accessible within the scene and game designers can utilise the singleton instance to access methods within the class. This implementation methodology means that game data is handled on a scene-to-scene basis, instead of having a single hashtable for all game data within the entire game. With the game resolver implemented, dialogue lines can now have their conditions queried against game data, as shown in Figure 26. Furthermore, with the parsing system implemented (Figure 21.), the condition's Enum type is utilised in a switch case to determine how to compare game data.

```
4 references | samsquanchone, 4 days ago | 1 author, 1 change
public bool CheckDialogueCondition<T>(T diaCondition)
{

    /*Player response data is Scriptable object driven, while the NPC XML system is driven by files and the data is contained in more complex collections, which can't be shown within the inspector.
     Generic object casting used to determine type and provide respective conditions to the game condition query, without needing to repeat this function for a seperate data structure */
    dynamic condition = diaCondition;
    if (typeof(T).Equals(typeof(Condition)))
    {
        condition = (Condition)(object)diaCondition;
    }
    else if (typeof(T).Equals(typeof(NodeCondition)))
    {
        condition = (NodeCondition)(object)diaCondition;
    }

    GameDataReturnType dataResolveType = GameDataResolver.Instance.QuerryGameData<dynamic>(condition.gameDataKey, out dynamic gameDataVal);

    switch (condition.triggerCondition)
    {
        case ConditionalLogicType.GreaterThan:
            if (gameDataVal > condition.conditionValue) { return true; }
            break;

        case ConditionalLogicType.LessThan:
            if (gameDataVal < condition.conditionValue) { return true; }
            break;

        case ConditionalLogicType.True:
            if (gameDataVal) { return true; }
            break;

        case ConditionalLogicType.False:
            if (!gameDataVal) { return true; }
            break;

    }

    return false;
}
```

Figure 26. Function implemented to utilise the game data resolver to query a dialogue condition.

3.4.6. Node-based player response branching

With a system in place to handle the deserialisation of XML file data into a dialogue database, dialogue is triggerable through passing identifications to a hashtable entry and retrieving the line's trigger key from a 'Line' object (Figure 19.). Furthermore, with a condition validation system implemented (Figure 21.), the condition object structure could be used to provide developers with the ability to define conditions for the responses. However, a condition object for manual definition would also need an Enum for the condition type, removing the need for calculating the condition provided by a string. By utilising systems and objects already implemented, a system could be developed that allows users to define interactions with the dialogue database in the form of player responses. The data structure created to allow users of the system to implement player responses is shown in Figure 27.

```
public class PlayerResponse : ScriptableObject
{
    [Tooltip ("Name of the Player, will be provided to the subtitle system")]
    public string playerName;

    [Tooltip("Name of the node, to aid in identification, no technical purpose")]
    public string responseNodeName;

    [Tooltip("Dialogue hashtable ID for the NPC you would like the player to interact with")]
    public uint npcID;

    [Tooltip("Player responses you would like to author (Note: CURRENT MAXIMUM OF 6 RESPONSES SUPPORTED)")]
    public List<PlayerResponseData> playerResponses;

    [Tooltip("The FMOD event that will be utilised for triggering dialogue through the programmer instrument)")]
    public FMODUnity.EventReference fmodEvent;

    [Tooltip("the next node that you would like to transition to when its condition is met")]
    public PlayerResponse transitionTo;

    [Tooltip("Condition to transition the node")]
    public NodeCondition tranistonCondition;

}

[System.Serializable]
5 references | samsquanchone, 4 days ago | 1 author, 1 change
public class PlayerResponseData
{
    [Tooltip("Text for the player response, will be provided to the subtitle system")]
    public string responseText;

    [Tooltip("List of conditions that will need to be met before a response will be shown (Note: Currently conditions are only of an OR logic type and do not support AND types)")]
    public List<NodeCondition> condition;

    [Tooltip("ID of the desired NPC line that you would like this response to trigger from its hash table entry")]
    public uint npcLineID;

    [Tooltip("Response node you would like the response to branch to. Note: Upon exiting the conversation, the node will return to its base node (Unless transition condition has been met)")]
    public PlayerResponse transitionNode;

    [Tooltip("Will exit conversation automatically if enabled")]
    public bool isExitNode;

    [Tooltip("Allows you to invoke seperate scripts when response triggered. Note: must be prefab'd behaviour due to the nature of scriptable objects")]
    public List<UnityEvent> eventsList; //Use this to be able to give quest ect ect
    [HideInInspector] public bool conditionsTrue = false; //Used for determining whether conditions are met for a response so the UI can show it or not
}


//Would just use the Condition object, however it contains data types that are not shown in inspector by default
[System.Serializable]
5 references | samsquanchone, 4 days ago | 1 author, 1 change
public class NodeCondition
{
    [Tooltip("Name of the game data, for human identification purposes only")]
    public string gameDataConditionName;

    [Tooltip("Conditional logic that you would like the condition to utilise. Note: do not use unkown type, this is for internal system purposes and not authoring!")]
    public ConditionalLogicType triggerCondition;

    [Tooltip("Value to parse: int,float, bool. Note: please populate even if using bool")]
    public string conditionToParse;
    public dynamic conditionValue;

    [Tooltip("Game data hashtable ID for checking a condition with its respective game data value")]
    public uint gameDataKey;
}
```

Figure 27. Player response data structure.

Although the 'NodeCondition' object highlighted in Figure 27. differs slightly from the 'Condition' object (Figure 19.) created for the base dialogue system, all variables apart from the 'triggerCondition' are identical. Therefore, generics could be employed to reuse the condition validation system by identifying the type passed to the system and then casting the generic into that type (Figure 21.). The data structure showed in Figure 27. is integrated to create a player response *Scriptable Object* (Unity Technologies, 2023) that developers can utilise to create separate instances and define each instance's object variables. This player response system also allows for the responses to have conditions, meaning they will not show as a dialogue option until the conditions have been met. Furthermore, responses are capable of invoking events when its respective response is triggered, allowing the system to be utilised to trigger behaviour separate from the dialogue system, such as a quest system. All functionality provided by the player response system is highlighted in Figure 28.

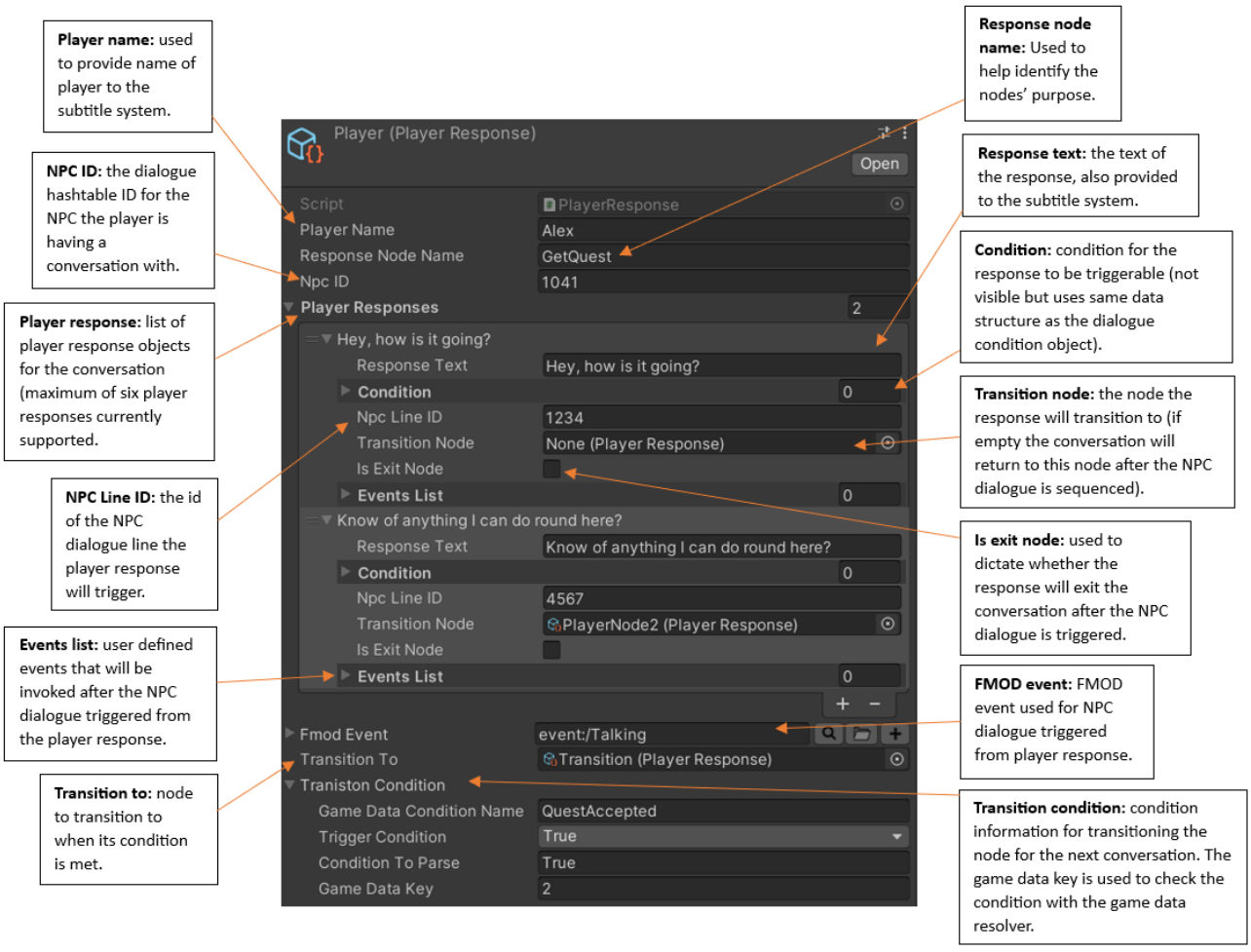


Figure 28. Player response example utilised in the test scene provided with the package.

3.4.7. Supporting modular game design with the observer pattern

Dialogue systems can become increasingly complex as the breadth of functionality they provide increases (Taylor-Giles 2022). Furthermore, when a dialogue system become the narrative driving force for a video game, it can also be required to support other game systems and provide a means of decoupled interaction with various game systems (Stevens, 2015). Therefore, through developing this dialogue system, a primary focus was on supporting the modularity required for interacting with any given game system while operating independently from it. As such, the observer pattern was put into practice.

As described by Sarcar (2020), the observer pattern enables developers to define a subscription mechanism to notify observers of a subject when something of interest happens. For this system the dialogue manager acts as the subject, being responsible for a variety of key functionality comprised within the dialogue system. In addition, the dialogue system has numerous observers, comprised of the dialogue manager, the player response user interface, subtitle manager, and the dialogue entities. When something of interest occurs within the dialogue manager, it notifies its observers and provides data key to that event, namely the dialogue state and the sequence type.

Consequently, each component can delegate behaviour independently inside their own system, without knowledge of other components. Furthermore, this architecture is utilised to provide developers with the ability to invoke independent events at key stages of a conversation through a ‘Dialogue Entity’ instance. However, the subject also provides the instance identification of the ‘Dialogue Entity’ script currently being interacted with to avoid all instances being invoked. The architecture for this implementation is illustrated in Figure 29.

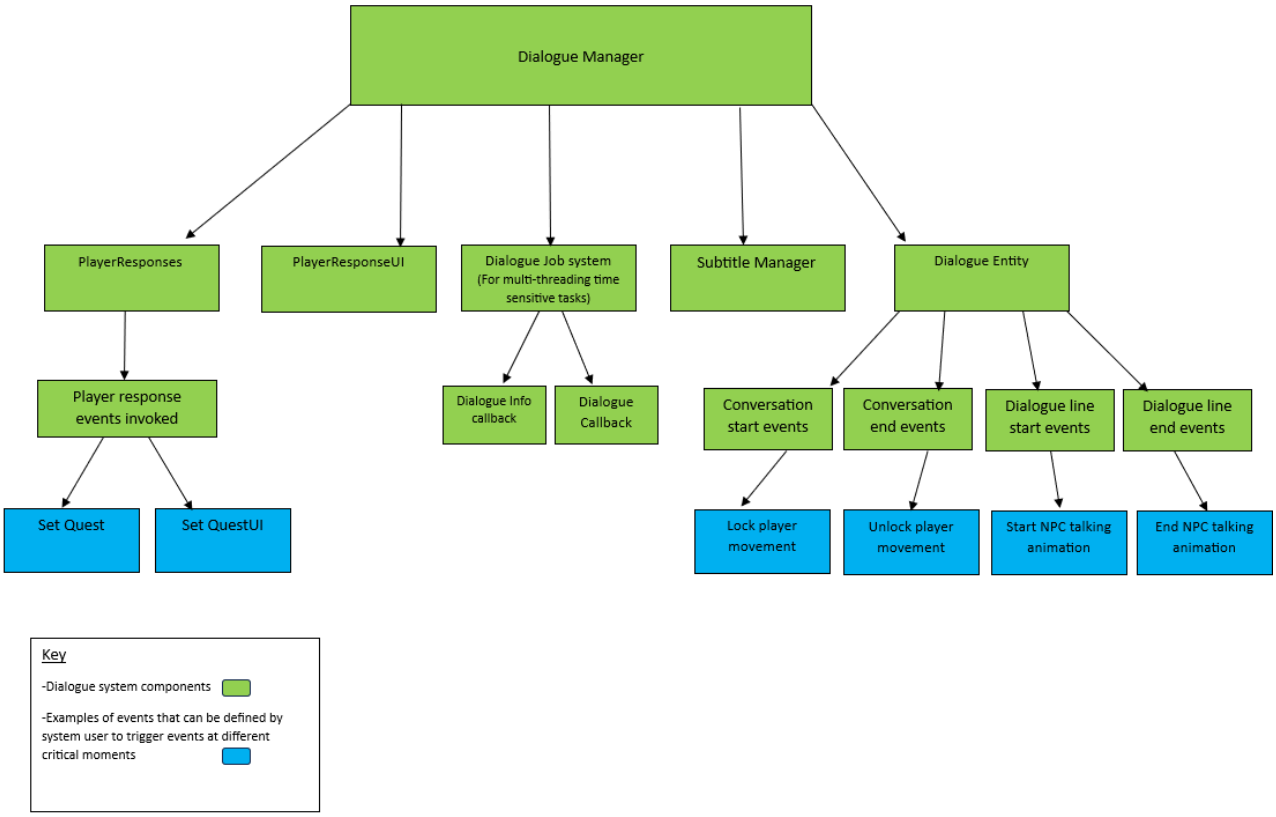


Figure 29. Architecture for the observer-based dialogue system.

4. Project management

4.1. Time management

As highlighted by Appendix 1-4. the time management for the production of this paper's artefact had to be adjusted as the development process advanced. This is particularly evident in the context of the dialogue system. As indicated by Appendix 1., it was planned for the dialogue development to commence at a later stage in the project cycle. However, both research and early implementation have demonstrated the potential for the dialogue system to become a highly labour-intensive and intricate system (Somberg, 2017). As such, the prioritisation of time management was modified to allocate most of the time into the creation of the dialogue system. Furthermore, the proposal had originally intended to be focused on design patterns. However, the research undertaken for this paper has been revised to primarily focus on the techniques for integrating audio to determine the appropriate design patterns to be employed during the production of the artefact, as indicated by Appendix, 2., 3. and 4.

4.2. Adaptations

4.2.1. One-shot and parameter system (Appendix 2.)

The initial plan for this system was to develop a footstep and gunshot system. However, as the literature review for this section progressed, it was identified how effect parameterisation is a common method for creating adaptive audio for these types of audio events. Furthermore, there are numerous factors that will require adaptations for a standalone footstep and gunshot system. An example being the *Unity Input System* (Unity Technologies, 2023), which would require changes to the input method for these systems. Therefore, it was decided that it would be more prudent to create a system that supports adaptable audio without users needing to reference the FMOD API in unrelated scripts.

4.2.2. Occlusion system (Appendix 3.)

While this system had originally been planned to provide a more generalised 3D and looping audio system, it was identified how FMOD does provide event emitters for 3D audio, as shown in Figure 7. However, FMOD does not support audio occlusion with its default spatialiser (Firelight Technologies, 2023). Therefore, the decision was made to focus on developing an audio occlusion system to extend the capabilities provided by the FMOD spatialiser.

4.2.3. Dialogue system (Appendix 4.)

The development of the dialogue system remained consistent with the objectives outlined in the proposal for this artefact. However, significant modifications had to be made to the time-plan of this system to accommodate the adaptations required. Furthermore, when implementing the observer-based architecture, adjustments had to be made to implement a script identification system. The purpose of this measure is to ensure that only the specific instance of an ‘Dialogue Entity’ script that is being interacted with has its behaviour activated.

As the dialogue system was adapted to involve a substantial amount of iterative research, it became the most time-consuming section within the whole project. This is reflected by the contrast of time management from the original time-plan (Appendix 1.) and the adapted one (Appendix 4.).

5. Results

5.1. Evaluation

By conducting research to investigate the role of audio in video games and understanding the expectations of contemporary game development, this study was able to develop a set of audio tools based on the findings. The resulting artefact from this research consists of three different systems. Firstly, a series of static classes that aim to enhance sound designers' capabilities of creating adaptive audio. Secondly, an audio occlusion system that aims to extend the capabilities provided by the FMOD default spatialiser. Finally, a dialogue system that allows for the handling of dialogue through XML files as well as providing other functionality within Unity, such as a player response system. However, although these systems have been constructed based on research findings and are hypothesised to enhance or extend the capabilities offered by the FMOD middleware, it is necessary to conduct testing. By utilising testing to analyse the feasibility of the developed systems, areas for improvements can be highlighted for further development.

5.2. Integration testing

There are many types of methods utilised to test software and typically a fully commercialised software will have been scrutinised by numerous different types of testing (TanLi et al., 2021). However, a commonly used method is integration testing. As highlighted by Badget et al. (2011), integration testing involves the testing of separate components to establish how they function as a combined entity. Furthermore, the aim of integration testing is to identify any erroneous behaviour that occurs due to these modules interacting with each other.

Therefore, integration testing was the primary mechanism utilised for quantifying the success of the developed systems. To support the integration testing of the systems, a unique environment was developed for each system in the form of Unity scenes. This was done to ensure that these systems work when integrated into an environment containing separate, non-related systems. Furthermore, the integration testing applied to these systems were done utilising a sub-type of integration testing, called big-bang testing. Big-bang testing involves testing all modules at the same time and can easily identify errors in such systems (Brar and Kaur, 2015).

5.2.1. One-shot handling and parameters system

For the one-shot with parameters and looping parameters system, a test scene was created that utilised a free-to-use Unity asset store package *Easy FPS* (Haberle, 2016). This test scene utilised the First-Person Shooting (FPS) scripts and assets provided with the package. However, the Unity audio found within the FPS package was replaced with audio created in FMOD, as the original audio had no modulation or adaptability. Hence, the implementation of adaptive sound was achieved by exploiting the one-shot parameter handling feature offered by the developed system. One instance of adaptable audio generated by the developed system is the sound resulting from shell casing impacts in the test environment, shown in Figure 30. This audio can be modified based on the characteristics of the surface upon which footsteps are made, indicated by different coloured tiles, hence adjusting the degree of attenuation applied.

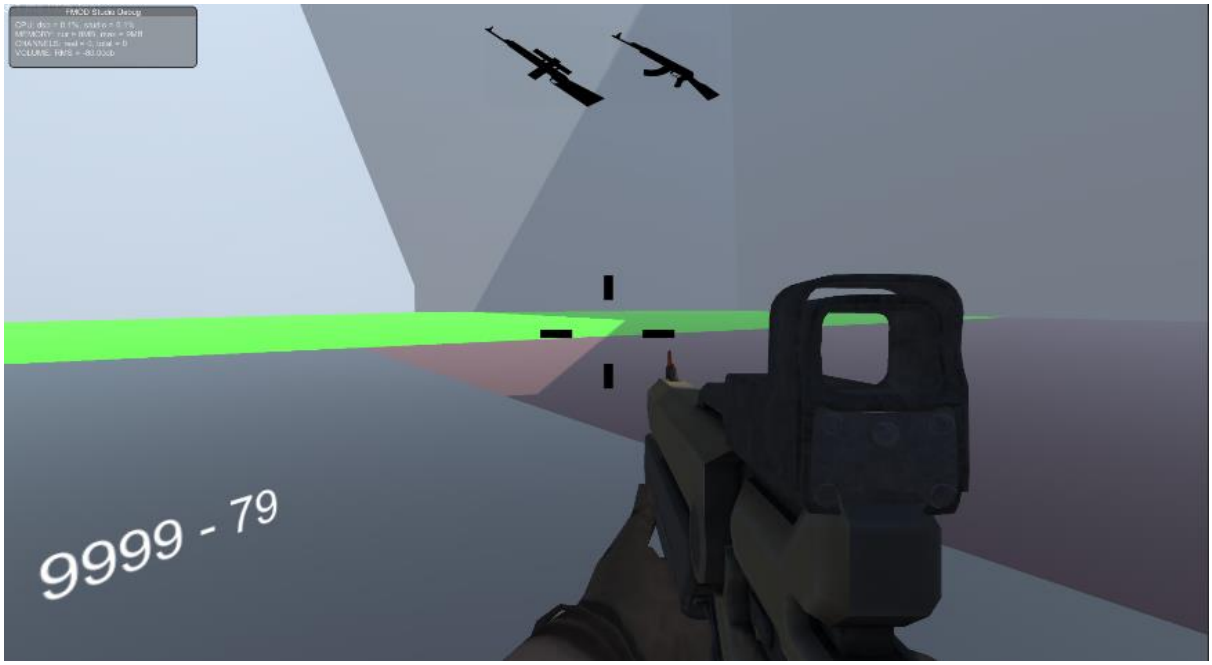


Figure 30. Integration scene created to test One-shot and parameter handling system. Utilises the Easy FPS (Haberle, 2016) package.

While the integration of this system into other game systems, provided by *Easy FPS* (Haberle, 2016) Unity package, was possible and indicated no erroneous behaviour with separate modules, a consideration for further work was identified.
References to FMOD events are still required for this system and would need to be handled by the users of this package; this could encourage people to reference audio events in non-related scripts. Therefore, providing a system for people to practically handle a collection of audio references could be a beneficial improvement to the system.

5.2.2. Occlusion system

The occlusion test scene utilised the first-person character controller script provided by Unity to provide a means of navigation around the scene. However, the rest of the scene contained prototyping and utilised no assets, apart from a pair of loudspeakers Filmbox (FBX) files. The aim of this test was to highlight two practical uses of the system and to evaluate how they interact with each other and the geometry of a scene. Within the scene are two distinct sound sources: an audio source pair emulating a large sound in the form of a set of loudspeakers, and ambient chatter in a separate room emulating a small sound size. This implementation is highlighted in Figure 31. and while the occlusion system provides an empirical method for implementing occlusion, the linecasts provided through the system do not behave as sound does in the real word. Therefore, utilising third-party plug-ins that provide more advanced approaches, such as raytracing-based diffraction, would provide higher levels of accuracy.

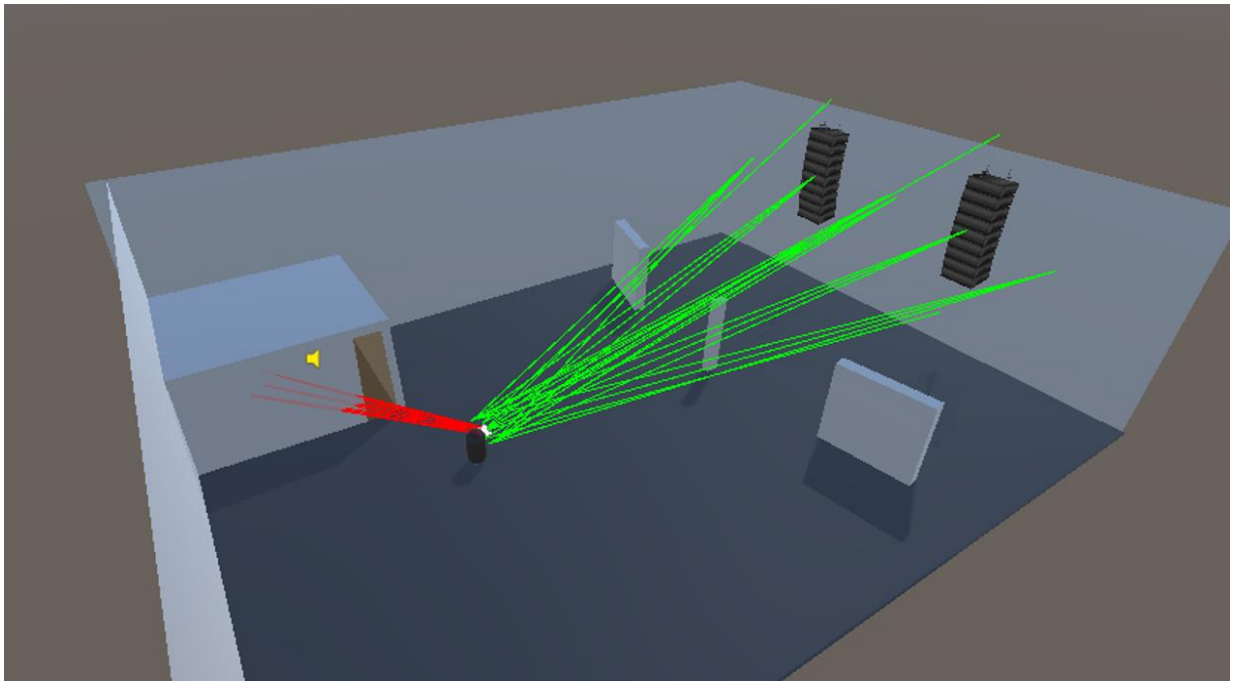


Figure 31. Integration scene created to test the occlusion system.

5.2.3. Dialogue System

Within the dialogue test scene (Figure 32.) there are three separate NPCs who utilise multiple instances of the ‘Dialogue Entity’ system script. This is to test how multiple instances interact with each other when contained within a single NPC. Furthermore, the test scene implements every trigger case provided by the system and highlights how the system can adapt based upon data within a game. The scene consists of numerous interaction types and provides an example on how it can be integrated into other game systems. For the test scene, this is achieved through providing the player with a quest upon accepting it from the

NPC named John, with the event system being used to trigger behaviour separate from the system.

A notable example of the adaptability of the system is highlighted in how the Dorothy NPC will not shout at you from a distance until you accept the quest. Upon doing so, new player responses and nodes unlock, in the form of asking what the NPC wants, as shown in Figure 32. While integration testing into other systems was successful for the dialogue system, it did highlight considerations for further work.

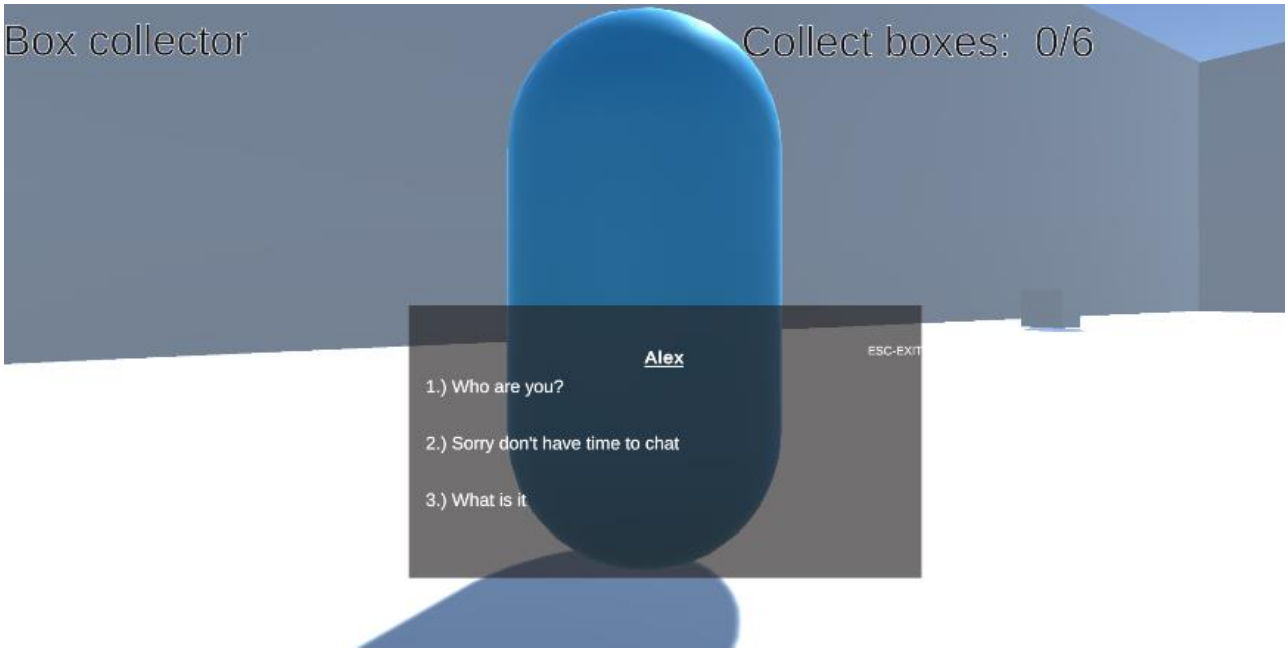


Figure 32. Integration scene created to test the dialogue system.

Firstly, while the player response system does allow for custom input bindings for each player response, the default Unity input system is used. Therefore, if a game was utilising the dialogue system and the new Unity input system, the response bindings would need to be updated for the new input system.

Secondly, the implementation for calculating the length of an audio clip to dictate the duration that subtitles appear on screen for are not always accurate. This length calculation is provided by the FMOD low-level API and uses an algorithm to estimate the length of a file, having around an 80% success rate (Firelight Technologies, 2023). Therefore, allowing users to define audio lengths from the XML system could provide a substantial improvement to this system.

Additionally, while the XML system does provide a basis to handle a vast number of dialogues, further considerations are required to support localisation features.

Finally, conditions within the dialogue system only support the ‘OR’ conditional operator type and not ‘AND’. Therefore, providing both conditional operator types could provide users with even greater control over dialogue triggering.

6. Further work

6.1. Introduction

While three overall systems were developed for different types of game audio, the dialogue system would have the most scope for further work. This is highlighted with the one-shot and parameter systems, as the scope is as large as the *FMOD Studio API* (Firelight Technologies, 2023) for parameter handling, which is provided through the developed series of static classes. Furthermore, while FMOD does not support occlusion with its default spatialiser, there are numerous third-party tools compatible with FMOD that can provide more accuracy than the system provided with this artefact, with a notable example being *Resonance Audio* (Google VR, 2022). Additionally, it was identified that FMOD provides audio tables to aid in triggering dialogue files, separate from a project's sound banks. However, all other functionality for a dialogue system is required to be implemented by a developer. Furthermore, many aspects developed for the dialogue system are independent of the FMOD API and only the dialogue calls themselves utilise FMOD to trigger the audio. Therefore, additional development could consist of providing a means to use the dialogue system with other audio middleware, such as *Wwise* (AudioKinetic, 2022).

6.2. Performance testing

As highlighted by Badget et al. (2011), performance testing aims to evaluate a system's performance under a realistic workload, this is used to quantify a system's stability and identify any potential bottlenecks. While integration testing was utilised to quantify the success of the dialogue system and its capabilities to interact with non-related game systems, additional testing is a key consideration for developing a foundation for further work. Although the *Unity Profiler* (Unity Technologies, 2023) was utilised to gain an estimation of the systems performance (Appendix 5.), it was done in the scope of a prototype scene and is not indicative of what would be found in a real game.

Consequently, to better gauge the overall performance of the dialogue system under more realistic conditions, a series of performance-based scalability tests would be employed to inform future developments. Scalability testing would be conducted by increasing the volume of data handled by the system and the resources used by non-related systems. To achieve this, the dialogue system would be implemented into an actual game that utilises animations, visual effects, and other systems that will utilise system resources.

Furthermore, this type of performance testing could also be used to identify parts of the system that could hinder the overall scalability when being implemented into an actual game environment. Therefore, this type of testing could not only identify areas of technical improvement but also areas that may aid in supporting scalability with the system.

6.3. User testing

User testing is a further method that would be utilised to identify areas for future development. As highlighted by Badget et al. (2011), user testing aims to identify design floors and software mistakes in a system. However, when utilising testers to quantify the usability of a system, it is important to consider the testing pool so that it is sufficient to obtain indicative results. Hence, as a basis for quantifying usability, the formula provided in Figure 33. would be applied to gain a percentage of errors found based upon the testing size. Additionally, the methodology utilised to gain data from user testing is an important consideration, as such a usability questionnaire would be created.

$$E = 100 \times (1 - (1 - L)^{\wedge} n)$$

where: E = percent of errors found
n = number of testers
L = percent of usability problems found by a tester

Figure 33. Calculating error percentage through user testing (Badget et al., 2011).

This questionnaire would mostly take the form of a Likert-scale with response values from one to seven (Joshi, 2015), with an aim of obtaining quantifiable data for analysis. However, some questions would elicit free-form comments, allowing for more descriptive suggestions in areas for improvement as well as noting any bugs. While it is important to have a sufficient testing pool to formulate conclusions from the testing (TanLi et al., 2021), having a targeted testing pool comprised of sound designers and technical sound designers is a crucial consideration for user testing. Therefore, by identifying which additions to the system would most benefit the target audience, more informed considerations could be made to establish a basis for further development.

7. Conclusion

Overall, this paper achieved its aim of developing a series of audio tools in line with what was declared in the proposal. However, as highlighted in Section 4.2.3., production of the dialogue system became the most cumbersome system in the development cycle. This observation is evident in the quantity of components associated with this particular system as compared to the other systems discussed in the context of this paper's methodology.

Furthermore, it has been recognised that with the increase of system resources available, there has been a corresponding rise in the range of tools accessible to sound designers (Andersen et al., 2021). This is particularly the case with spatial audio, as highlighted by recent innovations in VR game audio, which have been facilitated by the integration of third-party plug-ins, such as the *Oculus Audio Spatializer Plug-in for FMOD* (Meta Platforms, 2022).

However, despite the toolsets provided through middleware, as the complexities of developing audio solutions that satisfy increasing consumer demands rises, as does the necessity to develop independent systems to extend the functionality provided by middleware. This is highlighted with the development of the dialogue system. While *FMOD Studio* (Firelight Technologies, 2023) provides an approach to facilitate the activation of dialogue via audio tables, an additional system is necessary to effectively manage a large volume of dialogue lines and their corresponding trigger keys.

In conclusion, to fulfil the demands of contemporary audio implementation, it is often necessary to employ custom audio solutions that enhance the capabilities offered by middleware. Therefore, while middleware can be utilised to develop practical audio systems, the role of a technical sound designer is of vital importance.

Bibliography

Abdullah, R., Ismail, S. and Dzulkefli, N. (2020) Potential acoustic treatment analysis using Sabine formula in unoccupied classroom. *Journal of Physics: Conference Series* [online]. 1529 (2), pp. 1–7. [Accessed 10 July 2023].

Andersen, F., King, C.L. and Gunawan, A.A. (2021) Audio influence on game atmosphere during various game events. In: Budiharto, W., ed. *ICCSCI5: The 5th International Conference on Computer Science and Computational Intelligence* [online]. Virtual conference, 19-20 November 2020. ScienceDirect, 179, pp. 222–231. Available from: https://www.sciencedirect.com/science/article/pii/S1877050921000016 [Accessed 23 April 2023].

Andersen, J., Miccini, R., Serafin, S. and Spagnol, S. (2021) Evaluation of Individualized HRTFs in a 3D Shooter Game. In: Lamberto, T., ed. *2021 Immersive and 3D Audio: from Architecture to Automotive (I3DA)* [online]. Bologna, Italy, 8-10 September 2021. Piscataway, New Jersey, USA, pp. 49–58. Available from: https://ieeexplore.ieee.org/document/9610934 [Accessed 2 June 2023].

Anemüller, C. A., Adami and Herre, J. (2023) Efficient Binaural Rendering of Spatially Extended Sound Sources. *Journal of the Audio Engineering Society* [online]. 71 (5), 281–292. [Accessed 5 June 2023].

Audiokinetic (2022) *Wwise* (Version 2022.1.7) [computer program]. Available from: https://www.audiokinetic.com/en/download/ [Accessed 25 August 2023].

Badgett, T., Myers, G. and Sandler, C. (2011) Usability testing. *The Art of Software Testing.* 3rd Edition [online]. Wiley, pp. 143–156. [Accessed 2 August 2023].

Bala, P., Masu, R., Nisi, V. and Nunes, N. (2019) "When the Elephant Trumps": A Comparative Study on Spatial Audio for Orientation in 360º Videos. In: Brewster, S. and Fitzpatrick, G., ed. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)* [online]. Glasgow, Scotland, UK, 4-9 May 2019. New York, NY, USA: Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/3290605.3300925 [Accessed 5 June 2023].

Beig, M., Kapralos, B., Collins, K. and Mirza-Babaei, P. (2019) An introduction to spatial sound rendering in virtual environments and games. *The Computer Games Journal* [online]. 8 (3–4), pp. 199–214. [Accessed 23 March 2023].

Belle, S., Gittens, C. and Graham, N. (2019) Programming with Affect: How Behaviour Trees and a Lightweight Cognitive Architecture Enable the Development of Non-Player Characters with Emotions. In: IEEE Staff., ed. *2019 IEEE Games, Entertainment, Media Conference (GEM)* [online]. Yale University, CT, USA, 19-22 June 2019. Piscataway, New Jersey, USA: IEEE. Available from: https://ieeexplore.ieee.org/document/8811542 [Accessed 25 May 2023].

Bethesda Game Studios (2021) *The Elder Scrolls V: Skyrim – Anniversary Edition*. [Video game]. Bethesda Softworks.

Bostan, B. (2020) Game Design and Player Experience. *Game User Experience and Player-Centered Design*. Cham: Springer International Publishing AG, pp. 253-327. [Accessed 25 May 2023].

Brar, H. K. and Kaur, P. J. (2015) *Differentiating Integration Testing and unit testing*. In: Hoda, M.N., ed. *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)* [online]. New Delhi, India, 11-13 March 2015. Piscataway, New Jersey, USA: IEEE. pp. 796-798. Available from: https://ieeexplore.ieee.org/document/7100358 [Accessed 5 August 2023].

Broderick, J., Duggan, J. and Redfern, S. (2018) The Importance of Spatial Audio in Modern Games and Virtual Environments. In: IEEE Staff., ed. *2018 IEEE Games, Entertainment, Media Conference (GEM)* [online]. Galway, Ireland 15-17 August 2018. Piscataway, New Jersey, USA: IEEE, pp. 1–9. Available from: https://ieeexplore.ieee.org/document/8516445 [Accessed 5 June 2023].

Çamci, A. (2019) Some Considerations on Creativity Support for VR Audio. In: IEEE Staff., ed. *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)* [online]. Osaka, Japan, 23-27 March 2019. Piscataway, New Jersey, USA: IEEE, pp. 1500-1502. Available from: https://ieeexplore.ieee.org/document/8798210 [Accessed 20 May 2023].

Clark, C. A. (2021). Gameplay as Discrete Form: Leveraging Procedural Audio for Greater Adaptability in Video Game Music. *University of California Doctorate submission* [online]. University of California, Riverside. [Accessed 10 April 2023].

Croft, A. and Davison, R. (2018) Vectors. *Mathematics for Engineers* [online]. 5th ed. United Kingdom: Pearson Education, Limited, pp. 669-739. [Accessed 10 June 2023].

Davari, N. & Verbrugge, C. (2021) Efficient 2D Sound Propagation in Video Games. In: Thue, D. and Ware, S. G., ed. *Seventeenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'21)* [online]. Virtual conference, 11-15 October 2021. New York, NY, USA: Association for Computing Machinery, 17 (1), 132–139. Available from: https://dl.acm.org/doi/10.5555/3505520.3505537 [Accessed 5 April 2023].

Electronic Arts Vancouver and Electronic Arts Romania (2022) *FIFA 2023*. [Video game]. Electronic Arts.

Farnell, A. (2010) Game Audio. *Designing sound* [online]. Cambridge, MA: MIT Press, pp. 230–237. [Accessed 25 May 2023].

Fırat, H. B., Maffei, L. and Masullo, M. (2022) 3D sound spatialization with game engines: the virtual acoustics performance of a game engine and a middleware for interactive audio design. *Virtual reality: the journal of the Virtual Reality Society* [online]. 26 (2), pp. 539–558. [Accessed 20 April 2023].

Firelight Technologies (2023) *FMOD Studio* (Version 2.01.23) [computer program]. Available from: https://www.fmod.com/download [Accessed 26 August 2023].

Firelight Technologies (2023) *3D Sounds – white papers*. Available at: https://www.fmod.com/docs/2.02/api/white-papers-3d-sounds.html [Accessed 2 July 2023].

Firelight Technologies (2023) *DSP Plugin API – white papers*. Available at: https://www.fmod.com/docs/2.01/api/white-papers-dsp-plugin-api.html [Accessed 2 July 2023].

Firelight Technologies (2023) *Dialogue and Localization.* Available at: https://www.fmod.com/docs/2.01/studio/dialogue-and-localization.html [Accessed 2 July 2023].

Firelight Technologies (2023) *Effects reference.* Available at: https://www.fmod.com/docs/2.00/api/effects-reference.html [Accessed 2 July 2023].

Firelight Technologies (2023) *Core API Reference*. Available at: https://www.fmod.com/docs/2.02/api/core-api-common.html#fmod_3d_inverserolloff [Accessed 2 July 2023].

Google VR (2022) *Resonance Audio* (Version 1.2.1) [computer program]. Available from: https://github.com/resonance-audio/resonance-audio-fmod-sdk [Accessed 15 August 2023].

Guillen, G., Jylhä, H. and Hassan, L. (2021) The Role Sound Plays in Games: A Thematic Literature Study on Immersion, Inclusivity and Accessibility in Game Sound Research. In: Lugmayr, A., et al., ed*. Mindtrek '21: Academic Mindtrek 2021* [online]. Tampere, Finland 1-3 June 2023. New York, NY, USA: Association for Computing Machinery, pp. 12-20. Available from: https://dl.acm.org/doi/10.1145/3464327.3464365 [Accessed 23 March 2023].

Joshi, A., Kale, S., Chandel, S. and Kumar Pal, D. (2015) Likert scale: Explored and explained. *British Journal of Applied Science & Technology* [online], 7 (4), pp. 396–403. [Accessed 10 August 2023].

Kenwright, B. (2023) There's more to sound than meets the ear: Sound in interactive environments. *IEEE Computer Graphics and Applications* [online]. 40 (4), pp. 62–70. [Accessed 23 March 2023]

Lucasfilm Games (1985) *Ballblazer.* [Video game]. Atari Corporation, Epyx, Activision and Famicom.

Marx, C. (2021). Story-driven games and branching dialogue. *Writing for Animation, Comics, and Games* [online]. 2nd ed. United States: CRC Press, pp. 229-237. [Accessed 5 June 2023].

McIntosh, T., Weinel, J. and Cunningham, S. (2022) Lundheim: Exploring Affective Audio Techniques in an Action-Adventure Video Game. In: Anon, ed. *Proceedings of the 17th International Audio Mostly Conference (AM '22).* St. Pölten, Austria 6-9 September 2023. New York, NY, USA: Association for Computing Machinery, pp. 155–158. [Accessed 26 April 2023].

Meta Platforms (2022) *Oculus Audio Spatializer Plug-in for FMOD* (Version 47.0.0) [computer program]. Available from: https://developer.oculus.com/downloads/package/oculus-spatializer-fmod/ [Accessed 20 August 2023].

Microsoft (2021) *Microsoft Excel* (Version 16.0.13901.20400) [computer program]. Available from: https://www.microsoft.com/en-gb/microsoft-365/excel [Accessed 10 July 2023].

Microsoft (2021) *Overview - LINQ TO XML, Overview - LINQ to XML*. Available from: https://learn.microsoft.com/en-us/dotnet/standard/linq/linq-xml-overview [Accessed 30 May 2023].

Microsoft (2023) *Desktop Guide (Windows Form .Net)*. Available from: https://learn.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-7.0 [Accessed 23 August 2023].

Microsoft (2023) *Generics in .NET*. Available from: https://learn.microsoft.com/en-us/dotnet/standard/generics/ [Accessed 20 April 2023].

Microsoft (2023) *Using type dynamic*. Available from: https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/interop/using-type-dynamic [Accessed 26 May 2023].

Nash Audio (2022) Manhattan (Version 2.0) [computer software]. Available from: https://nash.audio/manhattan/ [Accessed 20 July 2023].

Nash, C. (2020) Crowd-driven Music: Interactive and Generative Approaches using Machine Vision and Manhattan. *New Interfaces for Musical Expression* [online]. [Accessed 6 July 2023].

Ni, J., Young, T., Pandelea, V., Xue, F. and Cambria, E. (2023). Recent advances in deep learning based dialogue systems: A systematic survey. *Artificial intelligence review* [online]. *56* (4), pp. 3055-3155 [online]. [Accessed 15 July 2023].

Nystrom, R. (2014) Singleton. Game Programming Patterns [online]. Genever: Benning, pp. 1–9. [Accessed 28 June 2023].

Oliver, J. and García Zapirain, B. (2021) *Software Tools for Teaching and Learning Sound Synthesis.* In: Chova, L. G., Martínez, A. L. and Torres, I. C., ed. *15th International Technology, Education and Development Conference (INTED2021)*. Virtual conference, 8-9 March 2021. IATED, pp. 1724-1729. Available from: https://library.iated.org/view/OLIVER2021SOF [Accessed 20 May 2023].

OpenAI (2022) *GPT-3.5* [Application Programming Interface]. Available from: https://openai.com/gpt-4 [Accessed 10 August 2023].

OpenAI (2023) *GPT-4* [Application Programming Interface]. Available from: https://platform.openai.com/docs/models/gpt-3-5 [Accessed 10 August 2023].

Patel, R. (2020) Acoustic terminology. *Architectural acoustics a guide to integrated thinking* [online]. Newcastle upon Tyne: RIBA Publishing, pp. 29-66. [Accessed 20 May 2023].

Pisha, L., Atre, S., Burnett, J. and Yadegari, S. (2020) Approximate diffraction modelling for real-time sound propagation simulation. *The Journal of the Acoustical Society of America* [online]. 148 (4), pp. 1922–1933. [Accessed 2 June 2023].

Plut, C. and Pasquier, P. (2020) Generative music in video games: State of the art, challenges, and prospects. *Entertainment computing* [online]. 33 (1). [Accessed 10 June 2023].

Rieger, M. and Zingler, D. (2023) 3D audio: The next step in realism for gaming: Benefits and pitfalls of 3D audio in games from a creative content and technical implementation point of View. *Games: Research and Practice*. 1 (1), pp. 1–3. [Accessed 26 May 2023].

Ryan, J., Mateas, M. and Wardrip-Fruin, N. (2016) Characters who speak their minds: Dialogue generation in Talk of the Town. In Sturtevant, N. and Magerko, B., ed. *Proceedings of the Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'16)* [online]. Burlingame, California, USA 8-12 October 2016. New York, NY, USA: Association for Computing Machinery, 12 (1), pp. 204-210. Available from: https://dl.acm.org/doi/10.5555/3505288.3505318 [Accessed 10 July 2023].

Ryan, J.O., Mateas, M. and Wardrip-Fruin, N. (2016) A Lightweight Videogame Dialogue Manager. In: Cole, T. and Salisbury, J. H., ed*. DiGRA/FDG '16 - Proceedings of the First International Joint Conference of DiGRA and FDG* [online]. Dundee, Scotland, UK 1-6 August 2016. Finland: Digital Games Research Association, 13 (2). Available at: http://www.digra.org/digital-library/publications/a-lightweight-videogame-dialogue-manager/. [Accessed 6 February 2023].

Santa Monica Studio (2022) *God of War Ragnarök*. [Video game]. Sony Interactive Entertainment.

Sarcar, V. (2020) Observer. *Design patterns in C#: a hands-on guide with real -world examples* [online]. 2nd ed. New York: Apress, pp. 269-284 [Accessed 2 June 2023].

Sinclair, J. L. (2020) Procedural Audio. *Principles of Game Audio and Sound Design, Sound Design and audio implementation for interactive and Immersive Media* [online]. New York, NY: Routledge, pp. 238–251. [Accessed 19 January 2023].

Somberg, G. (2017) Advanced topics. In: Somberg, G., ed. *Game audio programming principles and practices* [online]. Boca Raton: CRC Press, pp. 287-300 [Accessed 9 June 2023].

Stevens, R. and Raybould, D. (2015) Dialogue. *Game Audio Implementation* [online]. Routledge. pp. 219–248. [Accessed 12 June 2023].

Stevens, R. and Raybould, D. (2017) Sound sources and spatialization. *The game audio tutorial: A practical guide to sound and music for interactive games* [online]. New York, NY: Routledge, pp. 89–160. [Accessed 20 January 2023].

Subagyo, W. P. Nugroho, S. M. S., Sumpeno, S. (2016) Simulation multi behaviour NPCs in fire evacuation using emotional behaviour tree., In: Anon., ed. *2016 International Seminar on Application for Technology of Information and Communication (ISemantic)* [online]. Semarang, Indonesia 5-6 August 2016. Piscataway, New Jersey, USA: IEEE, pp. 184–190. Available from: https://ieeexplore.ieee.org/document/7873835 [Accessed 2 June 2023].

TanLi, M., Zhang, Y., Wang, Y. and Jiang, Y. (2021) Grey-Box Technique of Software Integration Testing Based on Message. *Journal of physics. Conference series* [online]. 2025 (1), pp. 12196–12203. [Accessed 25 July 2023].

Taylor-Giles, L. (2022) Systemic NPC-Side Branching in Linear AAA Video Game Stories. In: *Bostan, B., ed. Games and Narrative: Theory and Practice. International Series on Computer Entertainment and Media Technology* [online]. Springer, Cham, pp. 295-325. [Accessed 26 May 2023].

Trivedi, A., Kaur, E. K., Choudhary, C., Kunal and Barnwal, P. (2023) Should AI Technologies Replace the Human Jobs? In: Anon, ed. *2023 2nd International Conference for Innovation in Technology (INOCON)* [online]. Bangalore, India, 2023. IEEE, pp. 1-6. Available from: https://ieeexplore.ieee.org/abstract/document/10101202 [Accessed 15 July 2023].

Mario Haberle (2016) *Easy FPS* (Version 1.0) [unity package]. Available from: https://assetstore.unity.com/packages/3d/characters/humanoids/sci-fi/easy-fps-73776#releases [Accessed 20 August 2023].

Unity Technologies (2022) *Unity* (Version 2021.3.8f1) [computer program]. Available from: https://unity.com/releases/editor/archive [Accessed 20 August 2023].

Unity Technologies (2023) *Input system: Input system: 1.7.0, Input System | 1.7.0*. Available from: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.7/manual/index.html [Accessed 20 August 2023].

Unity Technologies (2023) *Profiler overview*. Available from: https://docs.unity3d.com/Manual/Profiler.html [Accessed 20 August 2023]

Unity Technologies (2023) *Physics.Linecast*. Available from: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html [Accessed 20 August 2023].

Unity Technologies (2023) *Physics.Raycast*. Available from: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html [Accessed 20 August 2023].

Unity Technologies (2023) *ScriptableObject*. Available from: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html [Accessed 20 August 2023].

Unity Technologies (2023) *Tracker modules*. Available from: https://docs.unity3d.com/Manual/TrackerModules.html [Accessed 31 July 2023].

Wratt, C. (2018) Developing procedural generation tools for video game audio designers. *Victoria University of Wellington department of Electronics and Computer Systems Engineering* [online]. [Accessed 25 May 2023].

Yalvaç, H.E. (2019) 'Creating Music inside The Box'*:* Do Chiptune's Limitations Have Aesthetic and Creative Value? *Istanbul technical university: Doctoral dissertation, Institute of Social Sciences* [online]. [Accessed 22 May].

Zdanowicz, G. and Bambrick, S. (2020) *Voice Production. The Game Audio Strategy Guide* [online]. 1st ed. Routledge. pp. 116–138. [Accessed 5 June 2023].

Zotter, F. and Frank, M. (2019) *Ambisonics A Practical 3D Audio Theory for Recording, Studio Production, Sound Reinforcement, and Virtual Reality* [online]. 1st ed. Cham: Springer Nature. [Accessed 15 April 2023].

Appendices

Appendix 1. Original Gantt chart submitted with proposal.

Task	Feb	March	April	May	June	July	Aug	Sep
Initial Research								
Review existing academic papers and studies relating to implementing audio systems within video games.								
Review existing literature relating to programming principles that could be applied to the developement of the proposed								
Pre-Development								
Review initial findings to develop system architecture plans for developing audio systems.								
Synthesise initial findings relating to softw are principles to abstract FMOD API into reusuable scripts, that can be used for the								
Development of audio								
Develop gunshot system								
Develop footstep and layer detection systems								
Develop dialogue system								
Develop event system for 3D and looping audio events								
Further research								
Use findings to make iterative improvements on already developed systems								
Use findings to develop remaining audio systems								
Dissertation paper								
Literature review								
Artifact write up								

Appendix 2. Adapted Gantt chart for the one-shot and parameter system, originally intended to be the footstep and gunshot system.

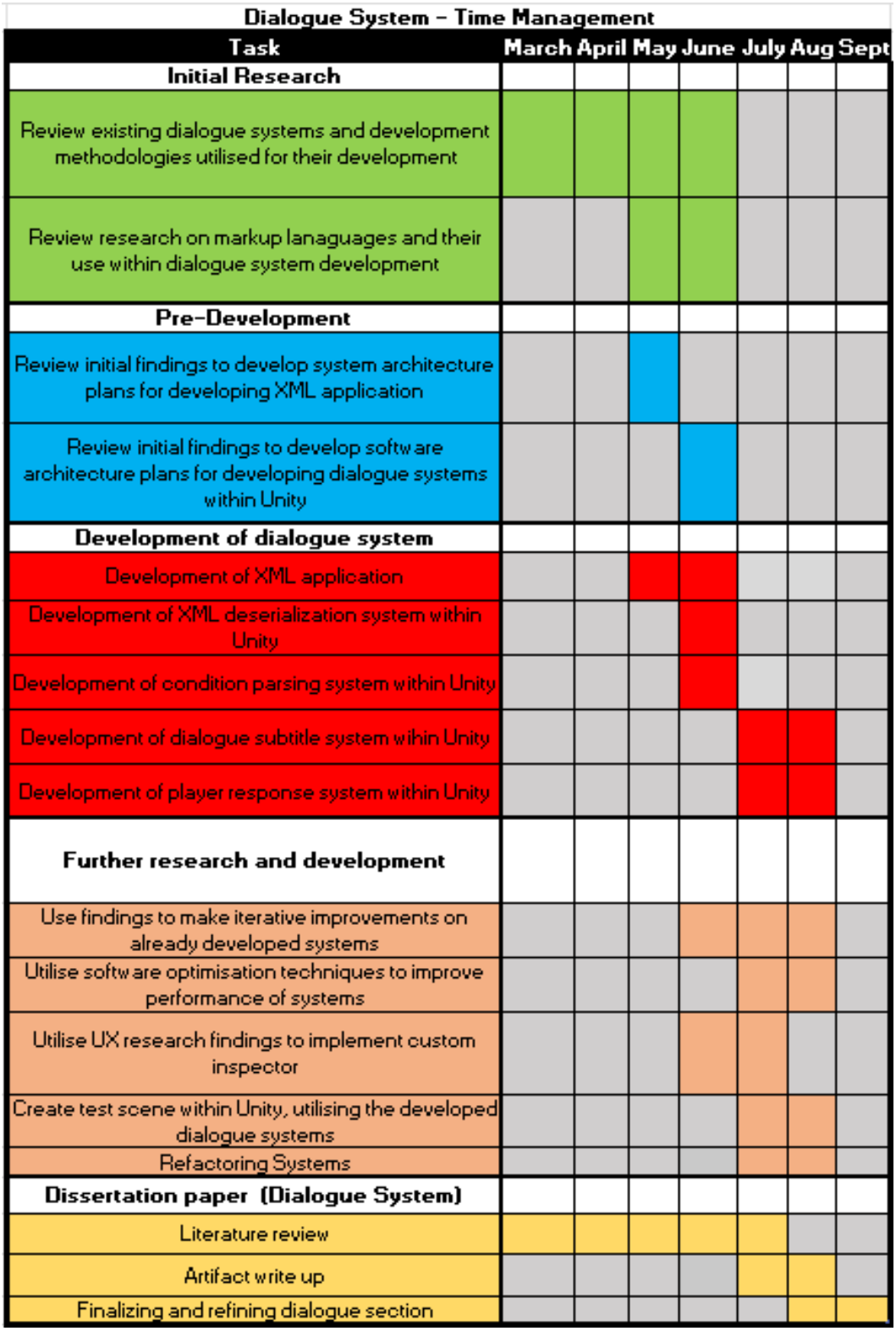
One-shot and Parameter Handing System - Time Management							
Task	**March**	**April**	**May**	**June**	**July**	**Aug**	**Sept**
Initial Research							
Review software principles utilised in SDK development							
Review parameter systems provided by the FMOD middleware							
Development of one-shot and parameter system							
Development of one-shot system							
Development of parameter handling system							
Further research and development							
Create test scene within Unity, utilising the developed occlusion system							
Refactoring Systems							
Dissertation paper (Occlusion System)							
Literature review							
Artifact write up							
Finalizing and refining one-shot and parameter section							

Appendix 3. Adapted Gantt chart for the occlusion system, originally intended to be the 3D and looping audio system, as shown in Appendix 1.

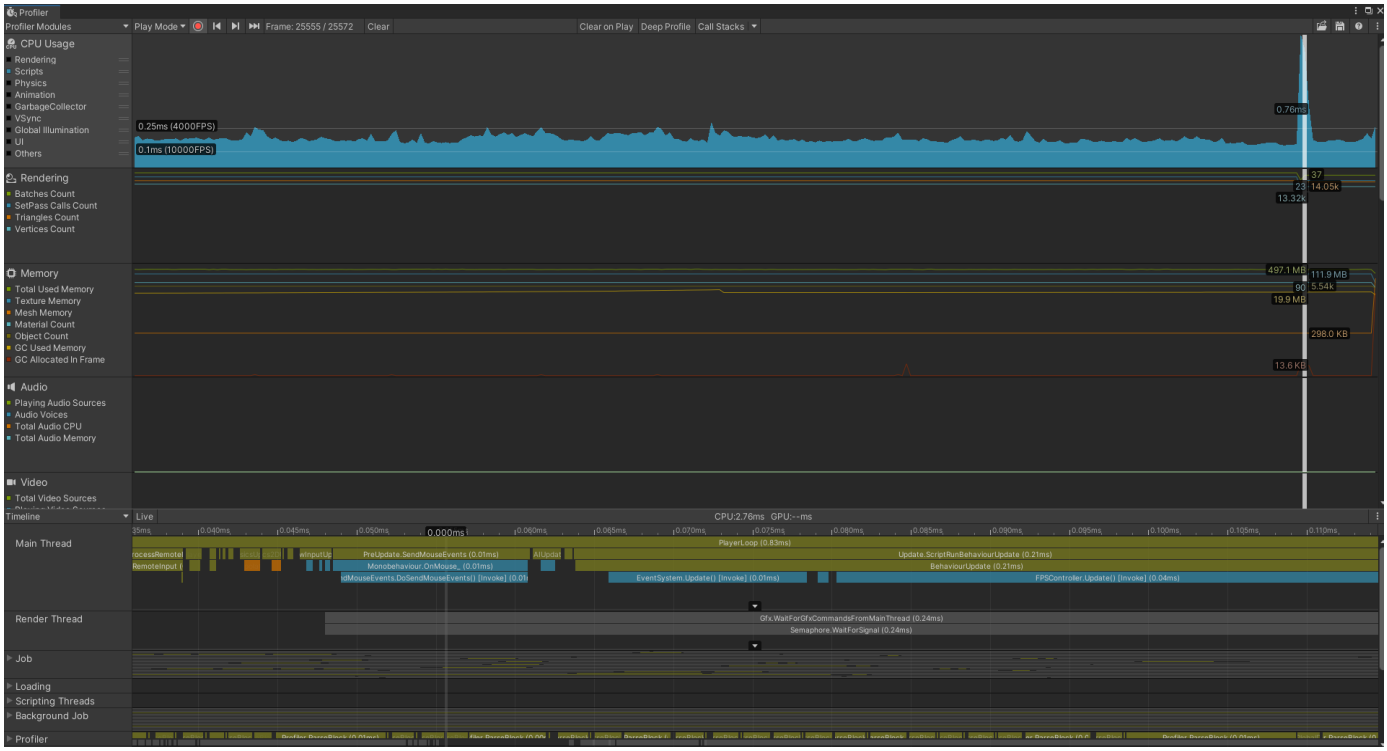
Occlusion System - Time Management

Task	March	April	May	June	July	Aug	Sept
Initial Research							
Review both industry utilisied and novel approaches for audio occlusion development							
Review 3D audio functionality provided by the FMOD middleware							
Development of occlusion system							
Development of line cast system							
Development of occlusion widening system							
Further research and development							
Use findings to make iterative improvements on already developed systems							
Utilise UX research findings to implement custom inspector							
Create test scene within Unity, utilising the developed occlusion system							
Refactoring Systems							
Dissertation paper (Occlusion System)							
Literature review							
Artifact write up							
Finalizing and refining occlusion section							

Appendix 4. Adapted Gantt chart for the dialogue system.



Appendix 5. Performance testing for the dialogue integration scene. Snapshot for components is taken at the highest performance spike that occurred during the scene.



Appendix 6. Link to one-Shot and parameter system prototype video: https://uweacuk-my.sharepoint.com/:v:/g/personal/samuel3_scott_live_uwe_ac_uk/EU_KTgTKojBEkTb0kh3HuDcBM2zmmGs1FHtYMePGBrLecw?e=d7zMfi

Appendix 7. Link to occlusion system prototype video: https://uweacuk-my.sharepoint.com/:v:/g/personal/samuel3_scott_live_uwe_ac_uk/EQ3ca7Gug3FDpzVYAV6JF3UBYAGd5EK67DveEPmv2F9R5w?e=8harEp

Appendix 8. Link to dialogue system prototype video: https://uweacuk-my.sharepoint.com/:v:/g/personal/samuel3_scott_live_uwe_ac_uk/EfDyuCYyrkJBhk5ZKbJdtlsBU_qWtD_yvwcZee4IoVFiSQ?e=a3wPRV

Appendix 9. Link to dialogue XML application prototype video: https://uweacuk-my.sharepoint.com/:v:/g/personal/samuel3_scott_live_uwe_ac_uk/EbE3gWY9qPpLhpCf7txdMBsB15lHJUY8OTVD23-xQLk_-w?e=AQSZqn

Appendix 10. Link to the project's git hub repository: https://github.com/samsquanchone/Dissertation-AudioSystems