Developing Procedural Generation Tools for Video Game Audio Designers

by

Christopher Wratt

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Engineering
in Electronics and Computer Systems Engineering
Victoria University of Wellington

Abstract

In video games, audio is often a vital element in the creation of immersive gaming experiences. One set of techniques that are particularly well suited to attaining this immersion are procedural audio techniques. These techniques enable enhanced immersion through supporting close synchronisation between player and game state in ways that are difficult to achieve with other game audio techniques. While this is the case, there is a lack of GUI and script-based tools that support the use of such techniques. This thesis explores this lack, and documents the development of two new video game tools for the creation of procedurally generated audio.

The first of these tools is a Musical Instrument Digital Interface (MIDI) library that supports the playback and real-time manipulation of MIDI files in the Unity game engine. The tool achieves real-time procedural audio, yet fails to meet required levels of time accuracy and is only a partial success. The second tool developed is a plugin hosting application that enables the use of the popular audio plugin format, VST2, in the Unity game engine. The tool succeeds in achieving VST2 effect plugin loading and, at the time of the completion of this thesis, is the only tool capable of embedding such plugins into applications developed in a major game engine. This will be of significant benefit to game developers who wish to achieve a high degree of immersivity in the music and sound design in their games.

Acknowledgments

Firstly I would like to thank my two supervisors, Dale Carnegie and Jim Murphy. Thank you both for your open mindedness, flexibility, and your high-speed editing skills which have kept me going throughout this project.

Thank you to my parents David and Clare and my brother James for putting up with all of my shenanigans and for always being there for me, even when I act like a diva.

Thank you to the Victoria University of Wellington Scholarships Office for your financial support. Without it this project would never have been possible.

Thank you to the International Game Developers Association (IGDA) who supplied scholarships that allowed me to attend Melbourne International Games Week and San Francisco Games Developers Conference (GDC) while undertaking this thesis. Many of the contacts I made at these conferences will be friends for the rest of my life.

Thank you to the international game development community. Your guidance via Slack, Facebook, Twitter, and in person has contributed vastly in the creation of this document. Special thanks to community members Aaron McLeran, Rich Vreeland, Peter Curry, and Charlie Huguenard for your technical support in interactive game audio over the past two years.

Thank you to Marika, Deet, Maddison, Hana, Katie, and Plague for being wonderful, kind friends who have helped me to remain human-resembling throughout this project. You are all amazing people whom I love.

Finally, thank you to my friends Jos and Piupiu for sharing with me your love for chiptune and game music. Your friendship and knowledge created the spark that started this research.

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

1.1 Motivation

In video games, audio plays a vital role in immersing listeners in game experiences. In order to achieve this immersion, a variety of techniques and tools is utilised by game audio developers, yet the ongoing development of such technologies and tools is comparatively under-resourced. This lack of resources and attention is likely due to a relative over-emphasis on visuals, which have been persistently prioritised since the inception of the medium (hence the name 'video' games). This disparity is easy to recognise when game development studios that employ hundreds of staff regularly only employ one or two audio specialists, who are then responsible for the creation of music and sound effects in all of the company's games.

This lack of emphasis on game audio has led to a scarcity of tools, which can make game audio development a difficult medium to work within. In order to work comfortably, many game audio developers rely on the use of pre-existing audio tools from tool-rich mediums such as film audio and recording studio production. While such tools are well suited to the creation of audio that is identical on every playthrough (known as linear audio) they generally lack support for the creation of non-linear audio that is different on each playthrough. Video games, through the presence of player interaction, are a non-linear medium and, while currently under-utilised in contemporary games, non-linear audio holds the potential to greatly enrich player immersion.

One of the most flexible and powerful ways to develop non-linear audio in a digital

environment is through a technique called procedural audio, and the development of procedural audio tools for use in video games is the focus of this thesis. In his book "An Introduction to Procedural Audio and its Application in Computer Games", Andy Farnell, a major figure in procedural sound design in video games, describes procedural audio as "non-linear, often synthetic sound, created in real time according to a set of programmatic rules and live input" [1]. In this thesis we expand this definition to include the use of audio effects in a procedural manner, such as the automation of effect parameters in real-time. The lack of tools that are capable of achieving procedural audio is regularly acknowledged by game audio developers. One such developer is Rich Vreeland, one of the world's top game audio developers[1]. In email correspondence with Vreeland in early 2017, he bemoaned the lack of procedural audio tools in game audio and hoped for the creation of tools for "runtime sound manipulation using waveforms and samples, various types of synthesis (subtractive, FM, granular), and an effects chain with an easy way to build out your own modules and build musical patterns" [2]. Over the last three years, we have regularly heard similar complaints of a lack of procedural audio tools by game audio developers at conferences, game jams, and meet-ups.

In spite of this lack of accessible tools, the utilisation of procedural audio in video games has existed since the first audio was implemented in digital video games in the 1970s. In spite of this, support for the use of procedural audio techniques in games has historically required the expertise of a specialised game audio programmer with knowledge of digital signal processing (DSP) techniques. The reliance on such specialists severely limits access to procedural audio techniques in games, particularly in the creation of independently developed video games.

This thesis works toward addressing this gap through the creation of new procedural game audio tools. Evaluative criteria for the success of such tools is detailed in the following section.

1.2 Research Goal and Evaluative Criteria

The primary goal of this thesis is to create tools that allow game audio developers, without specialised audio programming expertise, to create procedural audio

[1]Vreeland has composed the music for a variety of major game titles including Hyper Light Drifter, Fez, and Mini Metro

content in video games. To accomplish this goal, the game audio tools developed must be able to achieve procedural audio as defined in section 1.1. Expanding on this definition, criteria for the achievement of procedural audio in video game development are listed below:

1. Procedural audio in video games should organise audio material or effect parameters to some extent via the use of algorithmic and non-linear techniques. Music created should be different on a note-by-note or a structural level each time that it is heard, and sound design events should not identically repeat. This enables audio to achieve a high level of variation.

2. Audio created procedurally in video games should be affected by player input, and should not function as a closed system with no inputs from other elements of the game engine. One of the primary advantages of using procedural audio is its ability to interface with in-game events in a way that is more flexible than other game audio techniques. In order to achieve this, utilising player-input is vital.

3. Procedural audio generally utilises one or both of real-time audio synthesis and real-time audio effects. This requires the use of tools that are capable of DSP, and is further expanded in section 3.1 of this thesis which explores the technical requirements of such tools. Utilising DSP affords a high level of flexibility of auditory timbre results and broadens variation potential in procedural game audio.

Alongside being able to achieve the criteria listed above, a successful procedural game audio tool should utilise either a graphical user interface (GUI) or a documented application programming interface (API) that is accessible to game audio developers who may not possess specialised audio programming expertise. The tool should also be able to be used in one or more modern game engines that are used widely by the game development community.

1.3 Thesis Structure

The first chapter of this thesis has discussed the lack of accessible procedural audio technology for video game developers and has outlined evaluative criteria required for the successful creation of procedural game audio tools. Chapter 2 explores related works relevant to our research. In section 2.1 we explore interactive

audio techniques commonly utilised in game audio. We follow this in section 2.2 with an exploration of the history of computer-based algorithmic composition. Section 2.3 explores digital audio synthesis and effects and looks at ways in which the development of digital audio technology has influenced game audio. Finally, section 2.4 explores existing game audio tools and presents an overview of the current state of the field.

In Chapter 3 we undertake preliminary procedural audio experiments across a variety of game audio environments. The chapter consists of four sections. Section 3.1 sets out technical requirements of a successful procedural game audio system, against which to test tools throughout the rest of this thesis. In section 3.2 we undertake a series of tests in popular modern game audio development environments in order to understand their suitability as platforms for procedural audio tools development. Section 3.3 looks at historical game audio environments that support procedural audio. It then documents the creation of new tools in these environments and explores design patterns that made procedural audio achievable in the past. Finally, in section 3.4 we develop a procedural audio system in the visual audio programming language Pure Data (Pd) which brings together techniques explored throughout Chapter 2 and Chapter 3.

In Chapter 4, two new tools for the creation of procedural audio in video games are presented. Section 4.1 explores the creation of a musical instrument digital interface (MIDI) file reader and procedural MIDI organisation library for the Unity Game Engine [3]. Section 4.2 documents the development of a VST2 plugin host for Unity Engine, which enables the use of a variety of industry-standard audio tools within Unity Game Engine.

Chapter 5 concludes this document and begins with a summary of our research achievements and of the thesis as a whole in section 5.1. Section 5.2 discusses future work that could be undertaken in order to support advancements made in this thesis. Finally, section 5.3 reviews the thesis achievements.

Chapter 2

Related Works

This chapter explores related research that is relevant to the development of game audio tools for procedural audio creation. It provides contextual insight into the current state of game audio tools for procedural audio development and creates a foundation of knowledge from which to understand the new tools and methodologies introduced throughout the rest of this thesis. This chapter begins with a survey of significant modern techniques in interactive game audio development in section 2.1, before exploring the related fields of algorithmic composition in section 2.2, and digital audio synthesis, effects, and organisation in section 2.3. Section 2.4 describes the historical and current state of game audio tools development and discusses the functionality of a number of modern game audio tools for implementing game audio. Finally, section 2.5 provides a brief summary of the chapter.

When discussing game audio tools, it is important to understand the context in which these tools are used. Modern game development is generally categorised as either Triple-A or independent (Indie) development, and game audio tool usage is often different in each of these contexts. A Triple-A game refers to work created by a large team of game developers (typically 50 or more) that is funded by external publishers. Prominent examples of Triple-A games include the Assassins Creed™ games by Ubisoft Games, a series of 20 games by a company that employs over 20,000 people, and the Grand Theft Auto series by Rockstar Games. Triple-A game development companies generally create a number of their own proprietary tools for game development (known as in-house tools) and employ game tool developers across a variety of specialisations.

The term Indie refers to individual game developers, or small companies, that make

games. Due to their limited size, Indie game studios rarely hire specialised tools developers, and often rely on pre-existing tools that are freely or commercially available for use in game development.

Another set of definitions that are useful when discussing game audio development are the different roles that exist in the field. We define the core roles in game audio development as:

- **Game Composer:** Game composers are responsible for the creation of musical assets for video games. Game Composers specialise in the use of music organisation and notation software such as Sibelius, Finale, and MIDI editing software to compose musical material. They use Digital Audio Workstations (DAWs) to mix together musical audio material and to apply audio effects to such material.
- **Sound Designer:** Sound Designers create all non-musical audio assets in video games. This includes the creation of foley[1], ambiences, specialised design of object sounds, and voice-over. Sound Designers specialise in making and manipulating recordings through the use of DAWs.
- **Audio Implementer:** Audio Implementers take audio assets created by Game Composers and Sound Designers and use game audio tools to add those sounds to video games. Audio Implementers specialise in scripting audio events based on gameplay, and often have knowledge of common scripting languages for game development such as Lua and C#. Audio Implementers also specialise in the use of game audio middleware (software that assists them in the process of designing audio behaviour in games).
- **Audio Programmer:** Audio Programmers are responsible for the creation of game audio tools and for the development of new audio behaviours not supported in the scripted behaviours of game engines and middleware. Audio Programmers specialise in developing complex audio behaviours in game engines and work to extend those engines.

With an understanding of the core terminology in game audio development, and an outline of the chapter's structure presented, we have a foundation from which to describe related works relevant to this thesis.

[1]Foley is the creation of everyday sound effects related to human movement such as footsteps

2.1 Interactive Game Audio Development

While procedural audio tools development for games is the core subject of this thesis, a broader exploration of interactive audio techniques for game audio is an important step towards understanding how procedural audio fits within the larger field of game audio. Video games are an interactive medium and the audio that is used within them is therefore inherently interactive. There are many different styles of interactivity in game audio development and this section seeks to discuss and categorise these styles. The interactive audio techniques discussed here help us to understand the current state of the art of video game audio. When paired with subsequent discussions of algorithmic composition and digital audio synthesis and effects, they provide a strong technical and aesthetic background that informs our discussion of game audio tools throughout this thesis.

2.1.1 Looping Music

A common approach to game composition is the use of musical material that can be repeated seamlessly. The technique of looping music became popular in the 1980s in the soundtracks of games such as Super Mario Bros [4] and Donkey Kong for NES [5]. Looping music was necessary in many early video games due to the small amounts of available processing, memory, and storage resources on early game hardware for storing musical data and organisational code. This meant that long, complex compositions were impractical, and interactive audio behaviour was significantly limited by the hardware. Short looping musical segments meant that games could have music playing at all times. Through the creation of a variety of different loops, composers of the time could cater to different styles and moods required in different contexts.

In digital interactive systems such as a video games, looping audio is a useful technique, even when larger RAM and storage space is available. Through using looping sections of audio that can be interchanged based on player choices in a game, users define their own pace of change in the music rather than the music being linearly written to approximate expected player behaviour. In a modern game audio development context, loopable audio is generally developed in a DAW and then exported for use in a game. An example of the use of a DAW, in this case Reaper (a widely used DAW in the creation of game audio assets), to loop musical phrases can be seen in figure 2.1. In the figure, audio data is selected

through use of the white bar at the top of the transport window. Looping of selected material is then activated through toggling the green 'loop enable' button, seen in the bottom left-hand section of the screen. Audio is often looped in a DAW to audition for timing or waveform phase inconsistencies before being exported for use in a game.

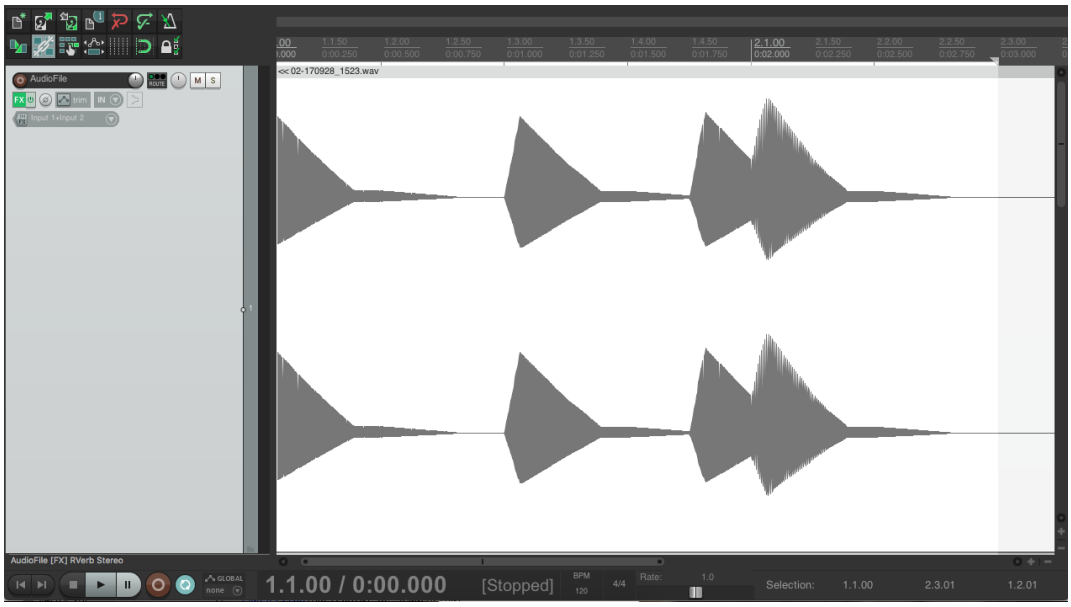

Figure 2.1: Looped audio in the DAW Reaper

Looped audio is a powerful and popular technique in game audio, but overuse of the technique can lead to ear fatigue in the listener and is a common reason for players muting audio in games. In order to combat problems with looped audio, game audio creators have developed a number of techniques to reduce the overuse of audio materials and to introduce greater degrees of audio interactivity into their games. These techniques are explored throughout the rest of this section.

2.1.2 Vertical Re-Mixing

A technique that has become popular in game composition since the 1990s is vertical re-mixing of game audio. In the book "Writing Interactive Music for Video Games", Michael Sweet defines vertical re-mixing as an "interactive composition technique in which layers of music are added or taken away to create levels of intensity and emotion" [6]. When employing vertical re-mixing, different layers of the composition are faded in and out based on events that occur in the game. Often vertically re-mixed phrases are looped sections, and through cross-fading

between these loops, game composers and audio implementers can create a variety of musical variations utilising only a small number of layers. A significant early use of vertical re-mixing in game composition can be found in the 1992 LucasArts game Monkey Island 2 [7]. Monkey Island 2 uses vertical re-mixing to build musical tension as the player explores a swamp environment early in the game. A screenshot taken from the swamp exploration section of the game can be seen in figure 2.2. The use of vertical re-mixed and looped phrases allows game players to indirectly control the rate of musical change through their actions. Even when the players' exploration of the game environment is erratic or unexpected, music systems that closely follow the behaviour of the player can be developed with the use of vertical re-mixing.

Figure 2.2: Screenshot from Monkey Island 2: Le Chuck's Revenge. The swamp exploration section pictured is widely recognised for its utilisation of vertical remixing

Vertical re-mixing is a powerful technique for game audio development, but problems occur when chord changes are introduced. When all musical material is in a single musical key, re-mixing materials is possible at any time, but when chord changes are introduced the technique often needs to be expanded and extra logic is required. A common solution to this problem is the combination of vertical remixing with a technique called horizontal re-sequencing.

2.1.3 Horizontal Re-Sequencing

Horizontal re-sequencing is the creation of musical segments that can be joined together linearly, as opposed the layered approach of vertical re-mixing. Horizontal

re-sequencing allows for chord changes determined in real-time based on game state, and is often employed with looped sections of audio that shift based on player interaction. Horizontal re-sequencing is most flexible when composers and audio implementers create different points at which music can shift from one segment to another. These shifts often occur at important musical moments such as the end of a bar, on a quarter note pulse, or at a significant chord change. Many game composers include small transitional composed sections alongside longer linear or looping sections, which allows for smoother movement between musical materials.

Horizontal re-sequencing is often used alongside vertical re-mixing, and when combined they form a useful way to support the emotion of narrative arcs in games. The combination of vertical remixing and horizontal re-sequencing is common in many modern games such as Journey [8] and Fallout: New Vegas [9]. With the combination of vertical remixing and horizontal re-sequencing, game composers can create music that functions similarly to a standard Hollywood film score, where music closely follows changes in mood on screen. This style of game audio composition works well in cinematic games, but is less useful in situations that require strong relationships between individual musical notes and gameplay. Horizontal re-sequencing may also run into problems when unexpected player behaviour occurs. For example, players failing to move from area to area of an open-world environment at an expected rate can lead to looping sequences of audio that never move to the next horizontal section and can quickly cause auditory fatigue in listeners. In these situations, procedural audio techniques are often a more suitable choice as they don't rely on pre-created audio material, and can achieve near-infinite variation possibilities.

2.1.4 Procedural Audio

The goal of this thesis is to expand the tools available for creators of procedural audio in video games, and it is therefore important to define and explore procedural audio. A definition of procedural audio can be found in section 1.2 of this thesis which describes the way in which procedural audio techniques bring together audio synthesis, audio effects, and the real-time organisation of audio. When applied to video games, procedural audio allows for careful synchronisation between in-game action and game audio. At its best, procedural game audio functions like a composer or sound designer creating audio in real-time based on the players' interaction with a game.

Procedural audio generation systems that can achieve audio synthesis, audio effects, and real-time organisation of musical material are uncommon in video game audio. However, a number of games that utilise one or more procedural audio techniques exist. An early example of the use of procedural audio techniques in a video game can be found in the soundtrack of 1984 LucasArts game Ball Blazer [10]. Ball Blazer is a two player sports game for the Atari 2600 console that uses randomised 'stochastic' note choices and real-time audio synthesis techniques to create melodic variation and tension in its soundtrack. The 1990s saw an expansion in the popularity of rhythm and music games that made use of procedural techniques in their composition. In the 1996 game PaRappa the Rapper [11], the player presses keys in time with a rhythmic pulse in order to have PaRappa (the lead character) rap in time. While PaRappa the Rapper uses pre-recorded sequences of audio, its use of player interaction to control musical creation in real-time leads to non-linear audio that is partially procedural. Five years after the release of PaRappa the Rapper, game developer Tetsuya Mizaguchi released the psychedelic rhythmic shooter game Rez [12]. Similar to PaRappa, Rez turns live rhythmic player input into musical material, yet Rez uses a more detailed approach to melodic generation than PaRappa the Rapper. Rez creates a sound for every player interaction within its world and each sound is quantised (locked to a rhythmic grid) and triggers an audio sample. Figure 2.3 shows an image of Rez gameplay. A significant example of a fully procedural composition in a video game occurs in the 2008 game Spore [13]. Spore makes use of both algorithmic composition techniques and real-time audio synthesis and effects, and will be explored in depth in sections 2.3 and 2.4 in our discussion of algorithmic composition techniques and game audio tools.

Procedural generation can also be applied to non-musical audio in games. The use of procedural algorithms for the creation of sound effects can be found in the 2013 game Grand Theft Auto Five [14], where many of the collision sounds between objects in the game world are synthesised in real-time. Collision sounds are a popular area of application for procedural sound design as data about the collisions often exists in memory as part of the in-game physics system and can be sonified. Procedural audio generation is also well suited to the creation of ambiences in games, where recognisable repetition could lead to a break in the suspension of disbelief. Standard practice in game audio ambience creation involves looping long ambience files that are changed in different environments, which requires significant memory and storage space. A further exploration of the use of procedural models for ambiences and other sound effects can be found

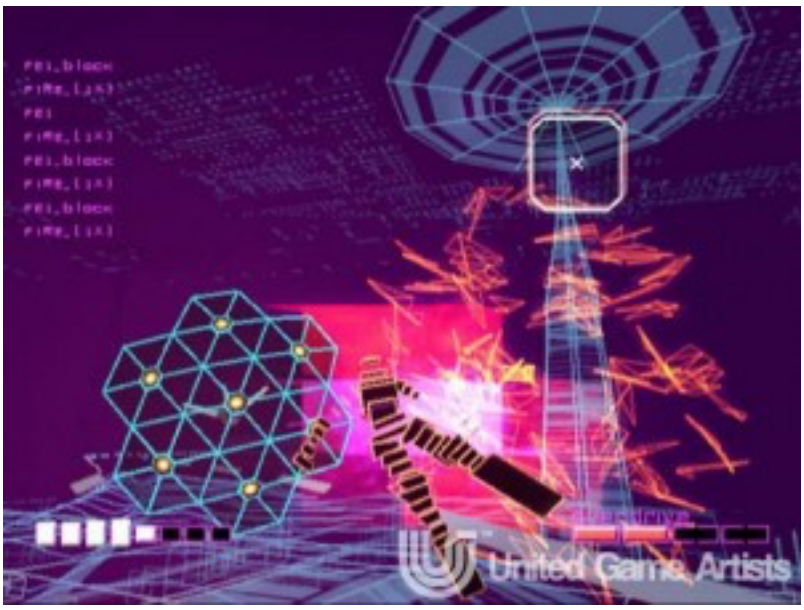

Figure 2.3: Game play in Rez, the character at the centre of the screen destroys objects in time with a metronome

in Andy Farnell's book Designing Sound [1]. While the procedural generation of audio has been utilised throughout the history of video game audio, it is often limited to use in Triple-A games (such as Spore, Grand Theft Auto, and Parapa the Rapper). Companies that create these games have access to specialised procedural audio tools developers, yet few tools for procedural audio creation are available for use outside of a Triple-A context.

2.2 Digital Audio Synthesis, Effects, and Organisation

A variety of technological advancements in digital audio synthesis, effects, and organisation has contributed significantly to the field of game audio tools development and to the techniques available to game audio developers. We need to explore these technologies in order to understand the current state of game audio technology, and to understand technical possibilities in the creation of procedural audio tools for video games. Early experiments in digital audio synthesis were undertaken by Max Mathews at Bell Lab. In 1957 Mathews developed MUSIC, which was the first widely used music synthesis software. Further significant developments in audio synthesis occurred in the early 80s at the Institut de Recherche et Coordination Acoustique/Musique (IRCAM) in Paris, where Miller Puckett developed the

software Max. Max allowed composers to write interactive computer music and the software continues to be a widely used tool in interactive digital audio. Puckett later developed a second program called Pure Data (Pd) based on developments from Max. Pd has been used in a small selection of game soundtracks and its use will be explored in our discussion of game audio tools in section 2.3.

Today, much digital audio is created in DAWs. DAWs bring together a vast array of digital audio organisational protocols in ways that allow audio creators to design audio for a large variety of purposes. In game audio development, composers and sound designers work with DAWs and are fluent in the knowledge of the techniques and organisational protocols utilised by DAWs. Modern popular DAWs include Pro Tools (popular in film and music mixing), Ableton Live (popular for music production and DJing), and Reaper (popular in academia and game audio). DAWs have become the standardised way in which game audio composers and sound designers create audio content. Later in this thesis, the DAW Reaper provides us with a number of significant benchmarks for audio latency, protocol support, and usability, to test our own audio tools against.

2.2.1 Digital Audio Synthesis

As this thesis aims to develop tools for procedural audio in video games, a survey of the history of audio synthesis, which is a vital technique in the procedural generation of audio, is an important task. Digital audio synthesis refers to the use of digital signal processing techniques to algorithmically generate audio. In digital audio synthesis, signals are produced in code before being translated into analogue signals via a digital to analogue convertor. They are then transduced into audio via one or more speakers. Audio must be synthesised at a sample rate sufficient to create alias-free signals that are within the human hearing range (20 Hz to 20,000 Hz). The modern standardised sample rate for digital audio is 44100 samples per second for CD and 48000 samples per second for film, which allows for the re-creation of signals up to the Nyquist frequencies of 22.05 kHz and 24 kHz respectively. Another major parameter in the encoding of digital audio is the bit-depth at which the audio is sampled. Standard bit depths have changed significantly throughout the history of audio synthesis, but many modern digital audio pipelines support 16-bit and 24-bit floating point and integer-based audio storage and playback.

Early audio in video games was created solely with the use of audio synthesis.

While earlier examples of digital audio in video games exist, the eight-bit video game consoles and personal computers of the 1980s provided us with the first significant use of audio synthesis to create sound design and music in video games. Early personal gaming devices such as the Atari 2600 computer and the Nintendo Entertainment System (NES) used audio synthesis processors that were separate from their main central processing units. On these early devices, sound was synthesised with an internal bit-depth from one to eight bits. Due to limitations in the hardware at the time, many signals were not band-limited and aliasing artefacts were unavoidable, creating the 'crunchy' and lo-fidelity timbre of early game audio soundtracks. Further discussion of the retro game audio technology can be found in section 3.3 of this thesis, and in the work of Australian academic and chiptune producer Sebastian Tomczak [15].

Figure 2.4: Nintendo Entertainment System or NES was a popular early video game console and utilised eight bit audio synthesis

The mid 1980s saw a significant technological advancement in audio synthesis technology with the commercial release of synthesisers that made use of phase modulation and frequency modulation synthesis (FM). Frequency and phase modulation synthesis refers to the modulation one waveform's frequency or phase by that of another waveform. When the modulation frequency is within the human audible range, it moves from creating a perceivable pitch change over time at frequencies under 20 Hz to creating new tonal material. This modulated signal displays harmonic behaviour difficult to produce with the wavetable synthesis popular in the 1970s and 1980s. FM synthesis was heavily adopted by the video games industry,

which led to a significant change in the sound of video game audio throughout the late 1980s and into the 1990s.

With the rise in popularity of red-book audio (an early CD audio format) and compact disk technology in the mid to late 1990s, audio synthesis lost popularity in game audio in favour of pre-recorded audio file play-back, yet the use of synthesis techniques in musical production remains common place outside of a video game context. Digital audio workstations utilise audio synthesis tools to allow composers to create synthesis across a huge diversity of timbres, and research in synthesis techniques such as the use of spectral models to realistically emulate instruments continues to occur [16]. A recent resurgence of interest in procedural audio has led to an increase in the use of audio synthesis in video games. Recent titles such as Fract Osc [17], Grand Theft Auto Five [14], and the upcoming VR title Lambchild VR utilise real-time audio synthesis.

2.2.2 Digital Audio Effects

Digital audio effects are audio DSP processes applied to pre-existing audio waveforms to change some aspect of the signal. The research and development of digital audio effect technology closely resembles that of digital audio synthesis, yet the application of real-time audio effects in games has been significantly different from that of synthesis. In early game audio, effect implementation was often too resource heavy for the hardware of the time, and its use was severely limited.

Since that time, while digital audio synthesis use has declined in modern game audio, the use of real-time digital audio effects has become an increasingly popular technique. One example of an audio effect processing tool in video games can be seen on the Sony Playstation 1, released in 1994, which included a digital reverberation effect implemented on its sound processing unit. Modern game audio middleware and game engines often support the use of real-time audio effects, and their use can be found in a number of modern video games (for example, Call of Duty Modern Warfare 2 which uses lowpass filtering [18], Half Life 2 which uses different reverb algorithm coefficients in each room [19], and the game 140 which uses bit-depth and sample rate reduction [20]). Through the rise in popularity of the use of audio effects in games, a number of new tools have been created such as tools used in the games listed above, yet audio effect tools development for video games lags significantly behind the effects tools available in DAWs. This lack has been lamented repeatedly in the video games industry by influential game audio

developers such as Disasterpeace [2] and has been discussed on the influential Game Audio Podcast [21].

The use of real-time digital audio effects in video game audio is an important technique for use in procedural audio, and audio effects are well suited to expanding the timbre possibilities of synthesised audio. The use of digital audio effects has also become a key feature of modern audio production in DAWs. As mentioned in Chapter 1, without support for a wide variety of digital audio effects, it is unlikely that procedural audio tools would be used in any meaningful way by modern video game audio developers.

2.2.3 Audio Organisation Protocols

Alongside the development of digital audio synthesis and effects for audio creation, a number of audio organisation protocols have been developed in order to facilitate the creation of audio in a digital environment. The most popular of these organisation protocols for digital audio is the MIDI protocol. The MIDI protocol was introduced to the digital audio market in 1982 by Roland, a synthesiser and audio hardware company, and quickly became the industry standard way to communicate digital audio event and parameter data. MIDI is a powerful communication protocol for music and is also a lightweight protocol (MIDI files are often less than one kilobyte in size). MIDI was used extensively in game audio alongside FM synthesis in the late 1980s and early 1990s. The use of MIDI in game audio applications declined with the introduction of CD audio, but MIDI is still a popular musical organisational standard used by composers when writing music in DAWs and musical notation software. A modern resurgence of interest in the use of MIDI in game audio is currently taking place, with recent games such as Peggle Blast Two [22] and the upcoming Lambchild VR finding ways to integrate MIDI into their audio workflow. The modern interest in MIDI has aligned with a surge in games using procedural audio techniques, and MIDI is functioning as a powerful tool for organising procedural audio due to its flexible nature and low storage requirements. While MIDI usage for procedural audio is becoming more popular, available tools for the integration of MIDI into a modern game audio workflow are severely limited. This lack of tools has limited the number of modern video games that utilise MIDI. A further discussion of MIDI implementation in video games occurs in our discussion of game audio tools in section 2.4.

The creation of audio organisation protocols for the utilisation of audio effects and

audio synthesis in a digital audio environment has become an important part of the digital audio ecosystem. A modular approach to digital audio synthesis and effect tools has developed, in which plugins are developed for use in digital audio workstations. A collection of file formats for the creation of synthesis and effect plugins for DAWs has been developed, the most significant being:

- **AudioUnits:** These work exclusively on Apple devices running OSX or iOS operating systems and make use of the Apple core audio and core MIDI frameworks.
- **RTAS and AAX:** Formats exclusive to the DAW Pro Tools. AAX is the current Pro Tools standard and has become a popular audio plugin format for the creation of audio tools for film and television audio.
- **VST:** VST (Virtual Studio Technology) is a plugin format created by Steinberg Audio. They are available cross platform for Linux, OSX, and Windows and are commonly used across most major DAWs.

While these standardised protocols exist for use in DAWs, there is no standardisation of plugin formats in game audio, and attempts to address this issue have never been fully realised. (The Fabric audio system includes a partial VST implementation, but its use is extremely limited and it cannot currently be compiled for use in game builds.)

Digital audio effects, synthesis, and organisational protocols have significantly affected the way in which game audio has developed. Audio synthesis via FM, wavetable synthesis, and MIDI musical organisation, were common in early games, yet the field changed throughout the 1990s due to advances in digital audio technology and audio effects and the playback of audio files rose in popularity. While many techniques from digital audio synthesis, effects, and organisation have significant historical usage in game audio, the current popular game audio tools often lack support for a number of significant digital audio techniques.

2.3 Algorithmic Composition

Due to its non-deterministic nature, algorithmic composition is an important field to explore when discussing procedural audio tools for video game development. Music theorist A. Alpern defines algorithmic music as the "process of using some formal process to make music with minimal human intervention" [23]. As

algorithmic composition on computers grew as a field through the 20th century, distinct compositional approaches became apparent. These distinctions led to a categorisation of algorithmic musical generation by Maurer in 1999 which we utilise in this thesis [24]. Maurer's work splits algorithmic composition into three main categories: stochastic composition, rules-based composition, and artificial intelligence (AI) in composition. The following sections describe each technique and explore their game audio applications.

2.3.1 Stochastic Composition

Stochastic composition is the use of random processes for the generation of musical material. This includes simple non-weighted random processes, and extends to more complex examples using techniques such as Markov models. Early non-digital examples of stochastic composition include Mozart's 1787 Musikalisches Wurfelspiel (Dice Game), a work that used dice rolls to determine the organisation of a number of short musical phrases. Another example can be found in composer John Cage's 1968 work Reunion. The work uses an electronically altered chess board upon which the players' chess moves trigger sounds that form the composition. In video games, the previously mentioned 1984 game Ball Blazer is an example of a game making use of a stochastic, non-weighted algorithm to create music.

Figure 2.5: John Cage's Reunion being played on an electronically altered chess board

Extended forms of stochastic composition include the use of Markov models to generate music. Markov models have been used extensively in the realisation of algorithmic composition and were a core technique used in the first piece of music

ever created on a computer: Hiller and Isaacson's 1957 Illiac Suite. The technique was explored by Iannis Xenakis in his book Formalised Music [25], where he wrote about the use of Markov models to determine the frequency, duration, and intensity of musical notes. Further theoretical development of Markov models for musical organisation occurred in 1987 when Polansky, Rosenboom, and Burk described a musical organisational structure using Markov models that included both a fine detailed model to determine individual note choices, and a higher-level Markov model to determine changes in musical form [26]. More recently the combination of Hidden Markov models with large databases of composition as input data has been utilised to reproduce new compositions in the style of Bach chorales [27] and jazz improvisations [28].

While the use of Markov models in video game audio is an uncommon technique, an example can be found in this thesis's author's 2016 video game Swim Swim Swim [29]. The game uses a second order Markov model to determine melodic pitch choices, and is weighted through the analysis of an input composition. The video game Spore [13] also utilises Markov models alongside rules-based techniques, and will be explored further in the following section. An image of gameplay from Spore can be seen in figure 2.6.

Markov models are highly suited to the composition of music, particularly in the creation of melodic material. This is due to their ability to closely mimic style in linear phrases, while still achieving a high level of variability (depending on the order of the Markov model used and the amount of training data used). They are commonly more musically flexible than rules-based systems, and more consistent in their creation of aesthetically sensible musical results than self-teaching AI models.

2.3.2 Rules-Based (or Formal Grammar) Composition

Rules-based and formal grammar algorithmic composition is the use of rules, usually derived from music theory, in order to compose music. Prominent historical examples include a program written by William Schottstaedt in 1989 that generates polyphonic composition based on the counterpoint music of Palestrina. The program uses 75 music theory rules to construct compositions. Mathis Lothe's 1999 paper, "Knowledge Based Automatic Composition and Variation of Melodies for Minuets in Early Classical Style", explores the application of formal grammar composition in order to create minuets in the style of Mozart [30]. Lothe's system

makes use of rules that determine the composition at a micro note by note level and at a macro level that determines musical structure. Often the musical output of rules-based composition can be difficult to distinguish from that of Markov model-based composition [31], and Loth's use of micro and macro approaches to musical composition closely resembles those proposed by Polansky, Rosenboom, and Burk [26] for application in Markov model composition.

A prominent example of the use of rules-based algorithmic composition in a video game is Maxis' 2008 life simulation game Spore, which makes extensive use of both rules-based and Markov model techniques [13]. The lead audio programmer from Spore, Aaron McLeran, used rules derived from jazz theory and counterpoint in order to create chord changes and to generate melodies. McLeran also employed Markov techniques through the use of "tuned probability tables using traditional music theory, including understandings of harmonic progression, melody construction, rhythmic construction, and form construction" to generate the audio for Spore [32]. Spore's music is also synthesised in real-time, thus fulfilling both A. Alpern's criteria of algorithmic composition [23] and A. Farnell's criteria of procedural audio [1], and is a significant work in the history of game audio development.

Figure 2.6: Spore gameplay example from 'microbe' section of game. Audio in this section changes based on the evolution of the playable creature

2.3.3 Artificial Intelligence in Composition

AI in algorithmic composition refers to models that can create their own formal rules or probabilistic weightings, rather than relying on pre-determined systems. While we are unaware of examples of AI composition techniques being applied in the context of video game audio, examples exist within the field of academic algorithmic composition. David Cope's Experiments in Musical Intelligence software (EMI) makes use of its "output to compose new examples of music in the style of the music in its database without replicating any of those pieces exactly" [33]. While EMI is similar to Markov-based systems that make use of the analysis of a database of compositions, Cope's system can also take into account its own output in order to form musical weightings in the future and can therefore, to a certain extent, shape its own musical language.

Of the algorithmic composition categories introduced in this chapter, AI methods of composition have the highest potential for variation through their ability to 'learn' and change as systems over time. While this is the case, they are generally harder to develop than rules-based or stochastic systems. Also the learning part of their system, if poorly developed, can quickly create music that would be innappropriate for use in a game development context. For this reason, a large amount of research into game play and AI composition would likely be required before AI composition techniques could be successfully utilised in a professionally released video game.

2.3.4 Summary

Each of the algorithmic techniques for composition explored in this section has the potential to be applicable in game audio applications, yet much of the technology required to enable the use of such techniques does not yet exist. Of the three approaches explored, rules-based composition is the most strictly controllable, and is therefore the most musically 'safe' algorithm presented. On the other hand, AI models, while being capable of a great deal of musical variation, are potentially difficult to use in a game audio context due to their unpredictability. The use of stochastic models to generate musical material fills a niche between rules-based and AI systems. While the use of stochastic Markov models creates rule-like systems, their rules are generally developed through analysis of training material rather than via strict enforcement by a composer. This leads to a higher degree of flexibility than

rules based systems with less potential for unexpected results than an AI-based approach.

2.4 Audio Tools Development for Video Games

Video games are a diverse art form that brings together a number of disciplines including programming, visual art, narrative design, sound design, and composition. In order to facilitate this inter-disciplinary communication, a huge variety of pipelines, tools, and standards have been created. This section will primarily discuss tool-based solutions that address workflow problems in game audio. In early game audio development during the 1970s and 1980s, programmers were often the only people that worked directly with game code, and artists would deliver assets to be transcribed into the engine by the programmers. Composer Yuki Takenouchi describes a historical process in game audio where he would write music and then hand it over to a programmer to transcribe it into code, to approximate the composers' output [34]. This approach had the disadvantage that composers and sound designers did not have direct control over how their audio would sound. Since that time, game audio tools developers have worked to create audio pipelines that reduce reliance on programming when working with audio in video games.

As audio technology improved over the last 40 years, so did game audio workflows and with the introduction of FM synthesis in video game consoles in the late 1980s came a number of game audio tools. A significant new tool was iMuse from LucasArts. Created in 1992, iMuse brought together a MIDI-based approach to musical organisation with FM synthesis, and was a versatile music creation tool for interactive audio creation. iMuse allowed for vertical re-mixing, horizontal re-sequencing, and transitionary organisation of musical material written in MIDI, and far surpassed the game audio engines at the time in terms of its flexibility and features [34]. A key part of the iMuse system was that music could be composed by a non-programmer and then integrated by an audio implementer with knowledge of SCRUMM (LucasArts' game development programming environment).

The 1990s also saw the arrival of a new concept in game tools development: the 'game engine'. Henry Lowood traces the first use of the term 'game engine' to the development of the 1993 game DOOM [35] and its development tool the DOOM Engine [36]. The DOOM team's innovation lay in their approach to organising game making tools. The DOOM Engine developers separated the core code

and technical implementation of the engine from the art, music, and other non-programmatic assets, thus creating a game development environment that both programmers and non-programmers could work in together. The term 'game engine' and the team's approach to separating programming from assets in video game organisation has gained significant popularity since its inception in DOOM, and has become a standard practice in video game tool development. Significant game development tools that use this organisational structure include the Quake Engine, Unreal Engine, and Unity Engine, three of the most widely-used game engines in video game development.

Figure 2.7: Image taken from DOOM's gameplay, a game created with DOOM Engine

The 2000's saw a number of new tools and APIs for game audio. Key developments at the start of the decade included the release of a powerful C++ API for audio: Open Audio Library (Open AL). Open AL is a multi-platform hardware abstraction layer API that became the major foundation for much of the audio tools development and video game audio engines of the 2000s. Open AL was used in historically significant games including 4 Unreal Tournament titles, Bioshock 1 [37], and in Maxis' Jedi Knight games. While Open AL is still being used in some modern video game titles, new tools and workflows have taken precedence in game audio development over the last decade.

The launch of the Apple App Store in 2008 led to a radical change in the video game market as games for mobile phones became increasingly popular and numerous. In

the same year the Danish company Unity Technologies released Unity iPhone, a version of Unity Engine 2 that supported targeting for iOS devices. Unity quickly became the dominant game engine for mobile phone development, and it continues to be one of the most popular game engines available. Unity Engine has also become popular in the creation of non-iPhone games and, as of 2018, has multi-platform build support for over 25 platforms. Like Doom Engine, Unity separates code and in-game assets. The software achieves this through the use of a GUI-based 'editor' and a C# scripting environment called Mono Develop. The Unity editor allows users to create folder hierarchies for game assets to be used in game development. The editor also includes two viewing panes, one that allows the game developer to place game assets in a 2D or 3D environment, and one that shows the scene as it will be viewed in-game. Figure 2.8 demonstrates one of the default layouts of the Unity Editor, with the scene editor used for arranging objects shown in the top left, the gameplay view pane in the bottom left, and the right-hand section of the screen dedicated to file system access and game asset management. Unity Engine's audio system is written using the FMOD C++ library, which is a game audio API for the development of interactive audio in video games.

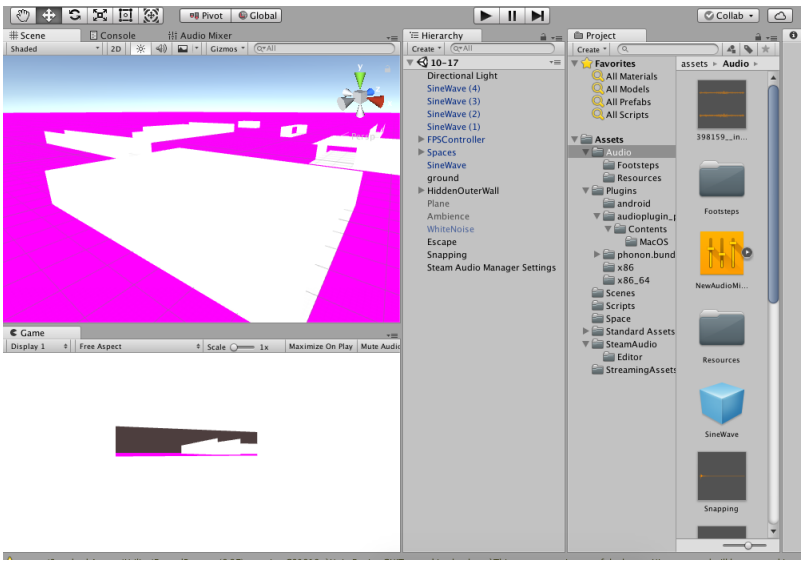

Figure 2.8: Unity Engine editor with gameplay window in bottom left, scene editor in top left, and folder hierarchy on the right

Another significant modern game engine is Unreal Engine, which has been in continuous development since its initial release in 1998. Unreal Engine is similar to Unity in its use of an organisational editor abstracted from its code, but rather than using C#, Unreal Engine uses a mixture of C++ and its own custom developed visual programming language: Blueprints. Unreal and Unity game engines are

significant development platforms for new audio tools, and a detailed discussion of their audio systems can be found in Chapter 3 of this thesis.

FMOD, alongside being a powerful C++ audio library, is utilised in the software FMOD Studio, a popular game audio middleware environment. Game audio middleware, as previously discussed, is software that assists video game audio designers in the process of designing audio behaviour in games. Middleware generally supports the use of many of the interactive game audio techniques discussed in section 2.1, alongside supporting a variety of audio effects. FMOD Studio can be integrated into games developed in a number of video game engines including the Unity and Unreal engines, and effectively replaces the in-engine audio tools and audio rendering pipelines of a game engine. Another significant audio middleware tool is Audio Kinetic's Wwise. Similar to FMOD, Wwise can be used as a C++ library, or as a full middleware solution with a graphical interface. Wwise includes a number of similar features to FMOD, and an in-depth discussion of the technical differences between audio in Wwise, FMOD, Unity, and Unreal will be covered in the following chapter of this thesis. These four game audio environments dominate game audio tools and it is rare to find current discussions of modern audio middleware and game engines that do not focus heavily on at least one of these four technologies [38] [39].

An important feature of Unreal, Unity, Wwise, and FMOD is that they are all extendable through the creation of new game audio tools. Similar to the use of VSTs and other plugin formats in DAWs, Wwise and FMOD each have a plugin development format for the development of new audio effects and synthesis tools. Unity and Unreal also allow users to extend upon their audio engines in a variety of ways. Unreal Engine is open-source and developers can change any part of the Unreal audio system to suit their game, but the engine developers offer significant support for the creation of audio plugins and tools with a mixture of blueprints and C++ [40]. Unity, on the other hand, is not open source, yet developers can create new plugins and audio behaviour in C#, or through the creation of dynamically loaded libraries that can be written in a variety of programming languages. Popular examples of tools developed using engine and middleware plugin interfaces include the McDSP plugins for Wwise, Google's resonance audio (3d audio library) which runs in all four environments, and G-Audio, an audio sequencing tool for Unity Engine. Another significant game audio tool is Tazman Audio's Fabric. Fabric is a Unity-only audio tool and supports many of the techniques available through Wwise and FMOD. Fabric blurs the line between game audio middleware

and a plugin tool as it is Unity-specific and utilises a number of features from the pre-existing Unity audio and editor systems while also adding new features. Although all of the audio tools discussed here have support for plugins for audio effects, there is no standardised format for these plugins. Developers of tools that work across multiple game engines and audio middleware systems must create different versions of their tools for each software. Also, of the major game audio middleware and game engine solutions listed, only Wwise currently supports the use of MIDI. This is a significant problem for developers of procedural audio for video games as new tools and audio organisation protocols must be learned or developed each time that a new game engine or audio middleware is used. This often constitutes an unreasonable amount of work and may reduce the likelihood of game audio developers turning to procedural audio as a solution.

Another important technology in game audio tools development is the use of audio software from non-game contexts in the real-time audio pipelines of games. An example of this is the use of the music visual coding language Pd in video games. Pd, as first discussed in section 2.2, is a powerful tool for the development of algorithmic music and audio synthesis and was developed by Miller Puckett after his creation of Max. Pd first became accessible for use in game development with the release of LibPd, which is a C tool that allows Pd patches to be run without their GUI on any device that can run native C and C++ code. LibPd has gained success in mobile applications, and has been used in a small selection of video games including Phosfiend System's music game Fract Osc [17]. While LibPd has become a popular and important piece of software for video game audio, problems with its CPU usage and with latency have limited its use in video game audio [41]. The 2008 game Spore [13], as discussed in sections 2.1.2 and 2.3.2, also made use of Pd as part of its real-time audio tool chain. This enabled the exploration of a number of algorithmic techniques that benefit from Pd's rapid DSP development paradigm.

As of late 2016, Pd implementation in video games has become significantly easier and more efficient through the efforts of UK company Enzian Audio and their tool Heavy [42]. Heavy is an online tool that allows users to upload and convert Pd patches into optimised native C or C++. Enzian have also created a number of tools to enable users to embed their Pd patches as plugins into different pieces of software on a range of platforms. Examples include the creation of VST2 plugins for use in DAWs, a Unity convertor that makes Pd patches appear in Unity's editor, and a Wwise plugin implementation. The technical aspects of Pd programming and

its use in game audio tools will be further explored in section 3.4 in the following chapter. While a range of game audio tools currently exist, their supported features are often far surpassed by many tools available for linear audio creation in DAWs. Many of the standard tools and formats used in DAWs cannot easily be used in game audio applications, and this thesis works to address this shortcoming.

With a survey of related works complete, we draw from research undertaken in this chapter in order to develop technical criteria for use in the development of game audio tools in the following chapter. We then prototype procedural audio systems with the use of software and techniques introduced throughout this chapter.

Chapter 3

Preliminary Development

This chapter examines methods for achieving procedural audio in a range of game development environments through the use of a rapid prototyping design paradigm. The information presented in this chapter is used to inform the creation of the new tools documented in Chapter 4 and to ensure that planned developments have not been achieved by other software. Throughout this chapter, a collection of applications across various game audio environments is developed. These applications are generally small tests to determine whether software meets certain technical baselines. On occasion, more extensive applications are developed with the goal of understanding nuanced software behaviour.

Section 3.1 of this chapter defines a set of design criteria and technical baselines against which to test procedural audio tools for game development. Section 3.2 explores the audio capabilities of significant modern game audio middleware and game engines and documents their suitability in a procedural audio setting. Section 3.3 expands the exploration into retro game audio technology, and looks at ways in which audio synthesis and audio organisational protocols utilised in retro game audio technology can inform modern procedural game audio systems. Finally, section 3.4 explores the use of the audio programming environment Pure Data in video games, and looks at ways in which it can be used to bridge the gap between retro and modern game audio technologies. Through each of these sections, methodologies for the creation of procedural audio tools are developed, and a map of tools and techniques for procedural audio development is established.

3.1 Technical Requirements

Through our exploration of audio tools development and related fields, a lack of audio tools and standards for use in procedural game audio has become apparent. In Chapter 1, we identified a selection of evaluative criteria for procedural audio tools that must be met in order for us to successfully meet research requirements. In this section, we extend upon our initial criteria and develop a set of technical evaluative criteria against which to test procedural game audio tools. In order for such tools to be considered technically successful in the context of this thesis, they should:

1. Use standardised audio organisational protocols from DAWs, such as MIDI, wherever possible. This ensures that tools are accessible to audio designers with DAW experience, and reduces development time whenever pre-existing protocols are utilised.

2. Achieve a level of time accuracy that is consistent with professional audio applications. Audio signal time inaccuracy should ideally not exceed 5 ms (220.5 samples at a sample rate of 44100[1]), the minimum audible threshold for percussive time inaccuracy. Signal timing should never exceed the maximum human threshold for audible non-percussive time inaccuracy of 30 ms (1323 samples). Both threshold values are taken from Helmut Haas's dissertation on psychoacoustics from 1949, a formative text on human auditory perception [43].

3. Be capable of either real-time DSP, or of interfacing with tools that support real-time DSP. This is an essential requirement of a procedural audio system as signal processing is required in both effect processing and in audio synthesis.

In our exploration of game audio technology in the rest of this chapter, we examine software in relation to both the technical guidelines above and evaluative criteria from Chapter 1. While in many circumstances understanding whether software tested meets evaluative criteria is possible through prototyping and the use of documentation, the quantitative testing of audio timing inconsistencies of each system required the development of a custom testing methodology. The need for this methodology became necessary after casual listening tests indicated

[1]Note that we use the terms 'sample' or 'samples' to refer to floating point values between -1.0 and +1.0 that represent signal amplitude. In the context of this thesis, digital audio signals are always discretely sampled at 44100 samples per second (the industry standard audio sample rate) unless otherwise stated

the existence of timing inaccuracies across a number of significant game audio tools.

In the developed test, clicks are sequenced at 120 beats per minute (BPM) in each piece of software tested (120 BPM is the default tempo used by most DAWs). All generated audio utilises the industry standard sample rate of 44100 samples per second. In each tested application, we attempt to place an amplitude +1.0 pulse at sample positions zero and 22050 of each second (thus achieving 120 BPM) over a five second duration. The resulting audio is recorded into the DAW Reaper via the audio routing software SoundFlower [44], and is edited so that the first generated impulse signal occurs at sample position zero. Audio is then rendered and imported into a custom testing application developed in the audio programming language ChucK, which checks for time-inaccuracies in the generated audio files. ChucK is chosen as the analysis environment as it is a 'strongly timed' programming language that supports audio analysis at a sample level [45].

The ChucK time inconsistency test, which can be seen in its entirety in listing 3.1, uses an amplitude threshold of 0.7 (a value chosen to filter out any noise or re-sampling artifacts present), set in line 2 and tested against in line 13, to test for time inaccuracies via checking the distance between impulse signals. The code updates at the sample rate, as seen in line 21 of the listing, and when a click event occurs, the code prints the number of samples of inaccuracy (if any) that are present. If the mean time inaccuracy over five tests ever exceeds 220.5 samples (5 ms), then the source application does not meet ideal time accuracy requirements. If the mean time inaccuracy ever exceeds 1323 samples (30 ms), then the application is unusable in the development of procedural audio (as defined by our criteria in section 3.1). If the system presents no timing inaccuracies at a sample level (i.e. 44100^{th} of a second) in any tests, then we consider the system to be 'sample-accurate', which is the highest degree of time accuracy possible in a discrete digital audio system.

With technical requirements in place and a test for sample accuracy developed, we go on to document our tests of game audio environments.

3.2 Procedural Audio Techniques in Game Audio Environments

This section explores the creation of procedural audio in modern, popular game audio development environments. Through the exploration of game audio tools in Chapter 2, four pieces of software emerged as forerunners in popularity, multi-platform support, and flexibility in game audio creation. These are the game engines Unity and Unreal Engine, and the audio middleware environments FMOD Studio and Wwise. In our exploration of these environments, we document whether they meet or fail the technical requirements listed in section 3.1, and we examine the viability of developing audio tools to support procedural audio synthesis in each environment. In order to test their capabilities, we create small prototype applications in each environment.

```
1  "/foo.wav" => string waveFile;
2  0.7 => float gateThreshold;
3  SndBuf sound => Gain g => dac;
4  0 => int numSamples;
5  0 => int numMs;
6  0 => int sampAim;
7  me.dir() + waveFile=>sound.read;
8  0=>sound.pos;
9
10 while(1)
11 {
12         //if audio exceeds gate threshold then print position
13         if(Std.fabs(g.last()) > gateThreshold)
14         {
15             //samples since system start minus sample aim
16                 <<< numSamples - sampAim >>>;
17                 sampAim + 22050 => sampAim;
18         }
19         //sample level update
20         1::samp => now;
21         numSamples++;
22 }
```

Listing 3.1: ChucK program for testing sample accuracy

3.2.1 FMOD Studio

As discussed in our exploration of game audio tools on Chapter 2, FMOD is a C++ library which is utilised by the audio middleware FMOD Studio. Here we focus on the middleware aspect of FMOD, and look to the C++ library when extended features are explored. FMOD Studio was commercially released in 2013, and is a GUI-based middleware application that can be used to organise audio content for video games. FMOD's GUI is similar to that of a DAW, but many of the tools and features available in FMOD Studio differ from those commonly found in DAWs. While, like a DAW, FMOD Studio can be used to organise audio clips on a timeline and to apply effects and automation to signals and parameters, FMOD does not support audio editing of waveforms and does not support MIDI. On the other hand, FMOD Studio includes a number of features that are not found in a DAW. While a DAW fundamentally allows us to render audio and MIDI files, FMOD Studio's main design paradigm features the creation of dynamic libraries as plugins. These dynamic libraries bundle audio files alongside logic and audio effect code in a way that can be used by a number of game engines. Once FMOD Studio files are included in a game project, the game engine can make calls to the compiled FMOD Studio project. Communication between a game engine and FMOD Studio occurs through the use of real-time Parameter Changes (RTPCs). RTPCs are parameters created in FMOD Studio that represent the game state of the parent game engine. Using these RTPC values, FMOD Studio can set up behaviours to be applied in game. For example, an RTPC called 'health' could be created which is fed values by the game engine representing the health of the player. In FMOD Studio, the RTPC could then, for example, be attached to a lowpass filter's cutoff frequency. As the player character's health approaches zero, a lowpass filter is manipulated programatically, thus giving an audible indicator of game state. The use of RTPCs is central to the design of FMOD Studio, and can be applied in a variety of game audio contexts. Common uses include determining the volume of vertically re-mixed layers of audio, triggering changes of looped audio with horizontal re-sequencing, and layering ambience files based on game state.

To test the accuracy of FMOD Studio's timing system, we sequenced click wavefiles in the FMOD Studio editor at 120 BPM and recorded them with our ChucK time-accuracy test. Our results show FMOD Studios output as sample accurate to the 44100^{th} of a second, making it a highly suitable environment for working with strongly timed audio content. The test can be seen in figure 3.1, which shows a looped 500 ms audio file with a click at sample position 0. FMOD Studio does not

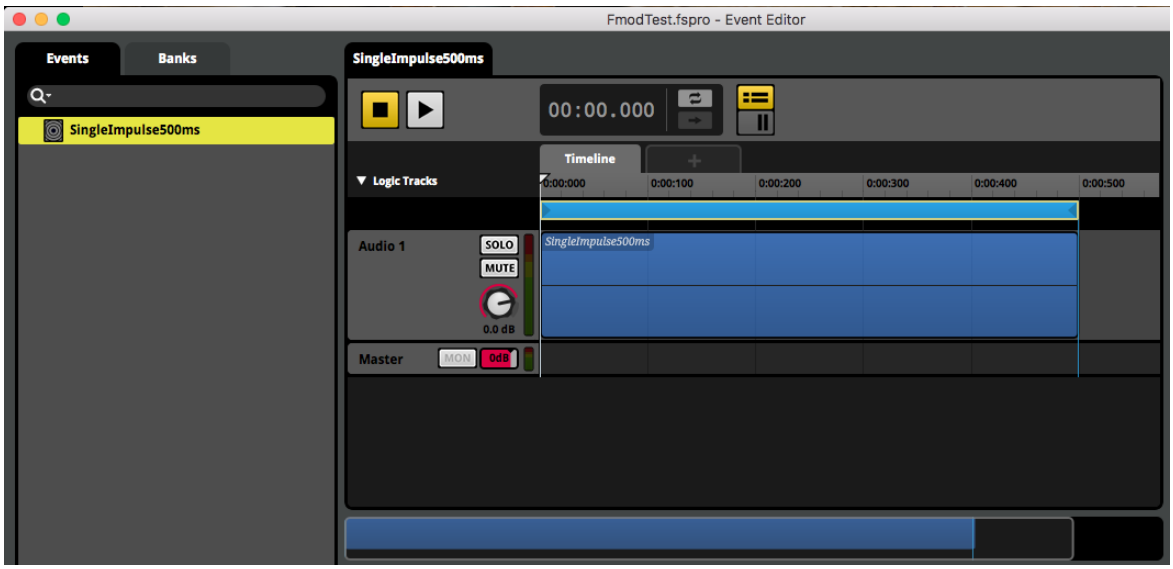

Figure 3.1: FMOD audio sequencing timing test. Note that a one-sample click is present at the start of the looped waveform but is not visible due to the resolution of the GUI

support the play back of MIDI files or the use of audio synthesisers. While this is the case, the FMOD C++ library supports the loading and playback of pre-rendered MIDI files, and these can be sequenced in time with the FMOD low-level timing system utilised by FMOD Studio. In order to achieve this behaviour, a MIDI file is loaded as a stream into the low level FMOD API, and is then scheduled with the FMOD call `System::playSound`.

In order for this function to play MIDI files, a Soundfont Two (SF2) file can be loaded or audio can be routed via the MIDI output of the application to a seperate system. While these features are supported by FMOD's C++ API, documentation of their use is scarce and the use of FMOD for the creation of a procedural MIDI system would likely require significantly more development time than is available to us in this thesis. After extensive online searching, it appears that FMOD's MIDI file reading capabilities have not been used in a modern video game, and are a legacy feature that is not under active development. The use of real-time MIDI sequencing without the use of pre-rendered MIDI files is equally undocumented and rarely used. Alongside FMOD's minimal level of support for MIDI, neither FMOD Studio nor FMOD C++ API support Open Sound Control messages.

FMOD Studio does have support for audio effects and the FMOD Studio download comes with 23 different audio plugins including a limiter, pitch shifter, and binaural spatialiser. Custom audio effects can be created as DSP objects via the FMOD C++ API and can be loaded into FMOD Studio. Full customised GUI creation is not

possible within FMOD Studio, but pre-existing GUI elements can be organised to create plugin interfaces with limited flexibility. Crucially, FMOD Studio does not support standardised audio plugin formats such as VST and Audio Unit, and there are no external tools at this stage that support standardised plugin format use in FMOD.

While FMOD Studio is a powerful and widely used environment for the design of interactive game audio, its lack of GUI-level MIDI file playback, lack of audio synthesis support, and inability to load standard audio plugin formats, make it an ineffectual environment for the type of procedural audio development pursued in this thesis. The development of procedural audio tools for FMOD Studio would require large-scale development of new tools for MIDI, synthesis, and plugin support, and would likely require access to the FMOD Studio source code, which is currently closed-source. FMOD Studio is therefore an unsuitable environment for the creation of procedural audio tools without the application of time and resources that lie outside the scope of this thesis.

3.2.2 Wwise

Wwise, like FMOD Studio, is a GUI-based audio middleware software which is coupled with a C++ programming API. While FMOD Studio strongly resembles a DAW in its layout, Wwise uses a different approach to GUI design, and utilises a large volume of windows and smaller tool interfaces that function in a modular way. The Wwise editor is more feature-rich than FMOD Studio, and, due to its extensive features, Wwise has become the standard game audio middleware for many Triple-A game development studios. Wwise editor projects are compiled to run as a plugin for game engines, and allow for the creation of elaborate interactive audio system design for video games. Wwise makes use of RTPC variables to interface between game engine and audio middleware, and the workflow is very similar to that used by FMOD Studio. In order to test the time accuracy of Wwise we use the looping audio tick test outlined in section 3.1. Wwise passes our test for time accuracy and also proved to be sample-accurate to the 44100^{th} of a second, making it a suitable timing environment for the procedural audio development pursued in this thesis.

Wwise goes beyond FMOD Studio in its support for MIDI file playback, and the inclusion of a synthesiser plugin called Synth One. While at first glance Synth One, paired with Wwise, fulfills many of the design criteria for a procedural audio

environment outlined in section 3.1, there are a number of major limitations in its use which we document below. Synth One, seen in figure 3.2, has two channels that support pitched waveform synthesis, and a channel for noise synthesis. The pitched waveform channels support the creation of sine, triangle, square, and sawtooth waveforms, and the square wave's duty cycle can be edited to create pulse wave signals. The pitched channels can be transposed, allowing the use of simple two-voice additive synthesis. The Noise channel's only parameter is volume control. The synthesiser does not have built-in filter capabilities for subtractive synthesis, but can be used in tandem with Wwise's built-in filter plugins to create subtractive synthesis. In a similar way, amplitude LFOs are not supported by the synthesiser, but automated volume modulation can be achieved in Wwise as an audio effect on the synth audio channel. Synth One includes an 'FM' parameter that supports the use of frequency modulation to the summed output of the three synth channels. The FM parameter is set by the user, but does not support changes in frequency modulation speed as a ratio of the note played (a common feature in commercial FM synthesisers). Synth One uses a four state attack-decay-sustain-release (ADSR) envelope for controlling volume with a fixed attack, decay, and release time and fixed sustain amplitude, and these parameters can be set by the user.

While Synth One is a significant audio plugin in audio middleware, it lacks many of the features of modern commercial software synthesisers used in DAWs. Its ability to synthesise a limited number of audio signals and its lack of built-in filters or filter enveloping means that Synth One is limited to playing audio that resembles 8-bit music and sound effects used in the game consoles of the 1980s. While the FM parameter allows for some audio timbre creation beyond the capabilities of these early game consoles, the lack of ratio-based control of frequency modulation and the lack of internal parameter enveloping in the synth means that it is not well equipped for emulating later FM audio technology such as the Sound Blaster FM audio card and other standard MIDI chips of the late 1980s and early 1990s. Due to the limitations in Wwise's Synth One tool, we consider it to be ineffective in attaining an acceptable level of features for modern audio synthesis, thus making it unsuited to the creation of procedural audio in a modern game context.

Wwise supports the playback of MIDI files, which can be scheduled to play using their interactive music editor window. Wwise does not include a built-in MIDI editor and MIDI files must be pre-rendered before being imported into Wwise. Because MIDI files cannot be changed in-editor, the algorithmic organisation of procedural MIDI in Wwise's editor is not possible. In discussions with Wwise's

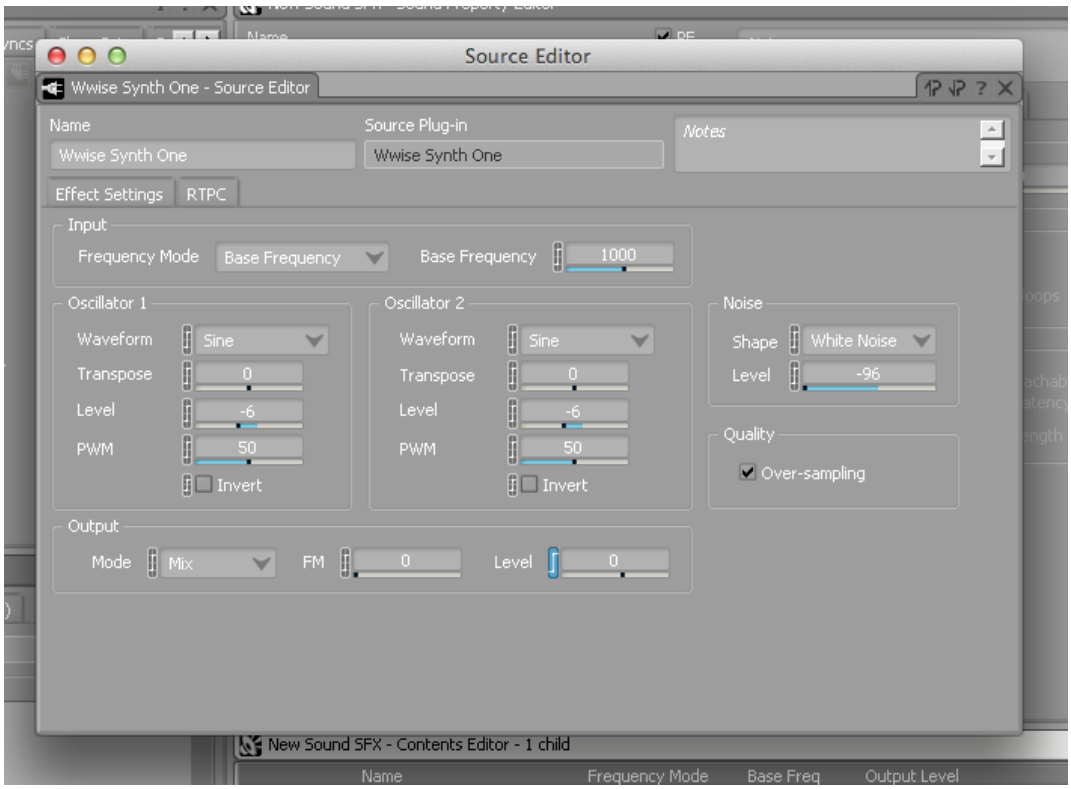

Figure 3.2: Wwise's Synth One Synthesis Plugin

audio programmers, they mentioned that it would be possible to edit MIDI files programatically using the Wwise C++ library, but they were not aware of any projects actively taking this approach to creating audio using Wwise [46]. Like FMOD Studio, Wwise does not include meaningful support for procedural audio organisation. Vertical re-mixing, horizontal re-sequencing, and real-time mixing of audio content are the true strengths of both pieces of software. In order to develop tools for procedural audio generation in Wwise, a system to edit MIDI files in real-time would need to exist. We chose not to pursue the development of such a system as the Wwise programmers did not know of a clear way to achieve real-time MIDI editing in Wwise. Also, in our experiments with the Wwise C++ plugin development API, we found it to be poorly documented and discovered few examples of its use online, which promised to result in a restrictively challenging development cycle.

3.2.3 Unity Engine

Unity Engine, as introduced in Chapter 2, is a widely used game engine and is popular in the creation of mobile and indie games. While game developers working

with Unity Engine generally use a GUI-based editor window to organise game assets, much of Unity's main features require the use of the scripting language C#. In Unity, C# is often used as an abstraction layer to call C++ functions, and this is the case for Unity's audio system which makes use of a highly abstracted implementation of the FMOD C++ audio API. While Unity Engine development often takes place in C#, the Unity Engine C# library supports the creation of GUI objects that can be viewed in the editor, and audio tools development for Unity Engine often makes use of GUI development libraries to enable rapid development of tools for non-programmers working in Unity.

Much C# programming undertaken in Unity Engine is written for use on the main processing thread, which executes a function at the application framerate called `Update()`. While the user can specify a target framerate (or use the default 60 frames per second), the final resulting framerate will fluctuate based on CPU usage. Audio programming in Unity is generally done in the `Update()` function, and uses a number of C# functions and classes that make calls to the FMOD C++ library. The FMOD code is not directly accessible to Unity audio programmers, and is present only as compiled binaries. Unity Engine supports the playback of audio files via the C# function `AudioSource.Play()`. The `AudioSource.Play()` function can only be called on the main thread (as opposed to the audio thread), and is tied to the framerate. This can lead to a number of problems when accurate timing of audio events is required. A frame rate of 60 frames per second (60 FPS) means that there is a 16.6 ms delay between frames, but this delay can change based on the CPU usage of a Unity application. Using the `Update()` function to drive a periodic audio signal leads to significant audible inaccuracy, as seen in the graph in figure 3.3 which details mean time inaccuracies generated by the code in listing 3.2 across five test iterations based on our test in section 3.1. The code presented sets an inter-event delay time of half a second in line 3. The update function in lines 14 to 20 calls at the application's framerate and is used to check if the current time exceeds the next scheduled event. We also check whether the next scheduled tick event will trigger before the next frame in line 16 through the use of `Time.deltaTime` which returns the length of the previous frame in seconds and is used here to estimate the length of the next frame. When the condition is met, the audio sample loaded into `audioSource` is played in line 18 and a new event is scheduled half a second after the previous event in line 19.

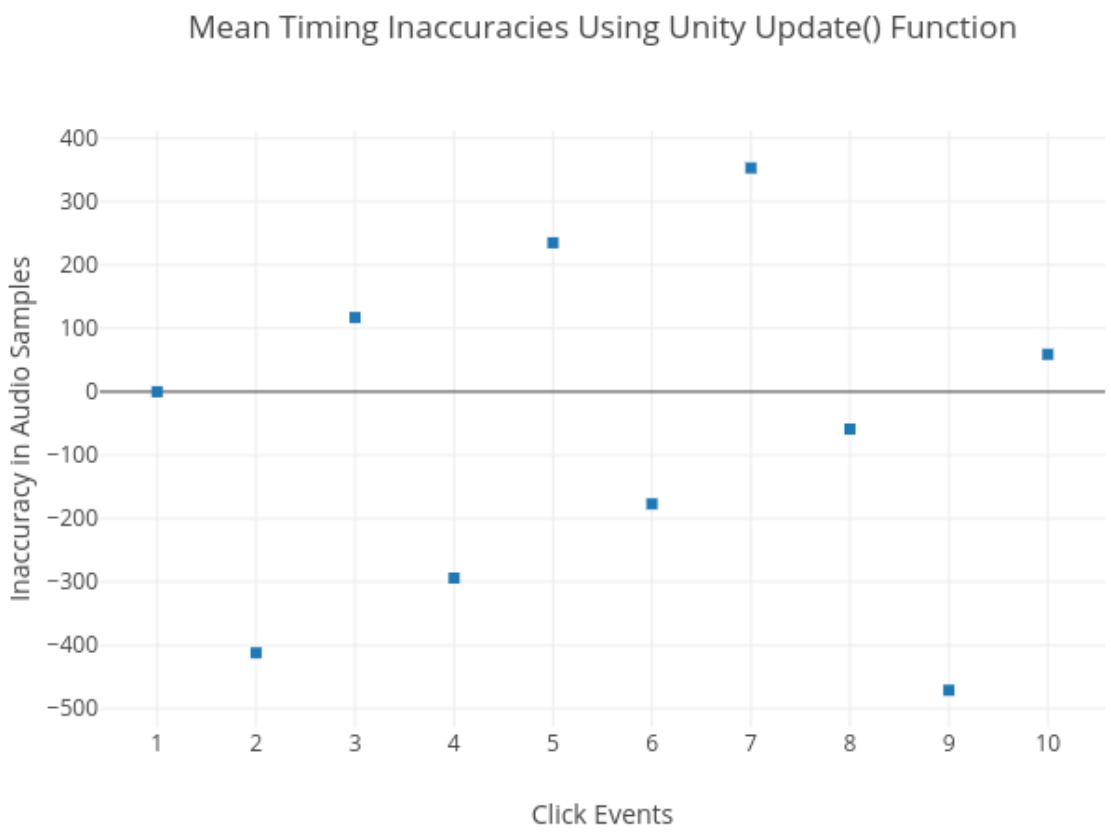

Figure 3.3: Mean sample inaccuracy from expected event timing using Unity Engine's `AudioSource.Play()` method when tested via the ChucK time-accuracy test in section 3.1. Mean time inaccuracy regularly exceeds +-220.5 samples (5 ms), the human threshold for perceiving percussive time inaccuracy, but never exceeds the maximum acceptable threshold for non-percussive signals of 30 ms (1323 samples). Note that standard deviation is not shown as there was no deviation present. Also note that the y-axis is measured in 'audio samples': discrete audio amplitude measurements that are accurate to the 44100^{th} of a second (as introduced in section 3.1)

```
1  AudioSource audioSource;
2  private double audioPlayTime;
3  private double timeOffset = 0.5;
4
5  void Awake() {
6      audioSource = GetComponent<AudioSource>();
7  }
8
9  void Start() {
10     audioPlayTime = AudioSettings.dspTime;
11     Application.targetFramerate = 60;
12 }
13
14 void Update() {
15     if(AudioSettings.dspTime >= audioPlayTime ||
16        AudioSettings.dspTime + (Time.deltaTime / 2) >= audioPlayTime)
17     {
18         audioSource.Play();
19         audioPlayTime += timeOffset;
20     }
21 }
```

Listing 3.2: Unity code for 120 BPM metronome audio using framerate Update() function

The results in figure 3.3 visibly alternate between being inaccurate in the positive and negative direction as they fall on either side of the quantised framerate before wrapping around on the ninth tick in every play-through. The results show the existence of timing inaccuracy in relation to sample accurate timing of up to 400 samples (around 10 ms). They also show an inter-event time inaccuracy of up to 600 samples (around 15 ms) existing when the number of samples between events at horizontal graph positions two and three are counted, and 600 samples is therefore the audible maximum time inaccuracy of the Unity test application. Utilisation of Unity's `AudioSource.Play()` for the creation of procedural audio is therefore only feasible when non-percussive signals are utilised, as its level of time inaccuracy is under Haas's threshold for audible time inaccuracy in such signals (as defined in section 3.1).

Due to the amount of time inaccuracy present in the preceding test, we look to other methods for achieving ideal time accuracy for procedural audio generation in Unity.

```
1  private double startTime;
2  private const double timeOffset = 0.5;
3  private double nextScheduledTime = timeOffset;
4
5  void Start() {
6      startTime = AudioSettings.dspTime;
7      audioSource.PlayScheduled(startTime);
8  }
9
10 void Update() {
11     //if the current audio system time is within two frames of the next
12     //event then schedule the event.
13     if(AudioSettings.dspTime > (startTime + nextScheduledTime +
14     (timeOffset - Time.deltaTime * 2)))
15     {
16         nextScheduledTime += timeOffset;
17         audioSource.PlayScheduled(startTime + nextScheduledTime);
18     }
19 }
```

Listing 3.3: Unity code for 120 BPM metronome audio using PlayScheduled(...) function

Unity also includes the function `AudioSource.PlayScheduled(...)`, which allows for scheduling of audio events at moments between visual frames. The code in listing 3.3 is used to generate a 120 BPM audio signal. It updates offsets in a similar fashion to the program in listing 3.2, yet rather than using the `AudioSource.Play()` function, it instead uses `PlayScheduled(...)` (on line 17 of listing 3.3).

This code generates a completely sample-accurate result in all five sample accuracy tests, and is therefore a viable way in which to schedule audio events in Unity. Issues with the use of `PlayScheduled(...)` include the fact that it must be called at least one visual frame before the audio event playback, as seen in lines 13 and 14 of the code listing, in order for the function's event to occur. The function can also only schedule one event at a time, and attempts to schedule multiple events with a single audio source component will lead to the audio not being played. While the use of `PlayScheduled(...)` allows for highly accurate audio timing in Unity, there is no built-in metronome or music-focused timing system in Unity. Developers utilising the function must therefore develop their own timing system that accounts for framerate latency, a difficult programming task that would require an audio developer who fills the role of a game audio programmer.

The final way to schedule audio in Unity in C# is through the use of an audio callback from hardware with the function `OnAudioFilterRead(...)`. The use of audio callbacks from hardware has become a standard way to implement sample accurate audio in a digital environment, and requires a hardware audio clock that buffers audio data, to be output at periodic intervals. The use of such a callback allows us to write highly accurate and low-latency audio code. Each time that the callback occurs, audio data in `OnAudioFilterRead(...)` is sent (via FMOD) to the audio hardware for output. This creates a small delay between the calling of the function and audio playback as the system must be at least one buffer behind the output in order for the computer hardware to accurately output the audio. Usual buffer sizes are 512 or 1024 samples per channel. The code in listing 3.4 demonstrates the use of `OnAudioFilterRead(...)` to generate an impulse signal at 120 BPM. The program counts elapsed samples and, if the number of elapsed samples exceeds 500 ms in line 12 of the listing, an audio signal is synthesised in lines 15 and 18.

When tested with our ChucK time-accuracy program, the system in listing 3.4 is sample accurate and therefore appropriate for accurately timed procedural audio development. While this is the case, the use of audio callback sample-level programming is an advanced programming technique, which is not accessible to many developers. Also, the function uses more CPU resources than `AudioSource.Play()` or `AudioSource.PlayScheduled()`. This extra CPU usage is due to the way in which the audio source component makes use of Unity's voice allocation system, which controls the number of active streams of audio data being processed at any one time. Voice streams each use a certain amount of CPU, and Unity generally automatically removes each stream when no audio is being played through it. Audio created via `OnAudioFilterRead(...)` is not organised by the Unity automatic voice allocation system, and therefore a single instance of `OnAudioFilterRead(...)` will use up a system audio voice for as long as the function runs. This can use significantly more system resources than the main thread audio functions previously discussed. `OnAudioFilterRead(...)` is also implemented in C# which is a managed programming language in which memory management is automated. This introduces a memory allocation pipeline and thread organisation system that is not well suited to real-time audio synthesis. The non-suitability of C# memory and thread management for sample level audio programming will be explored further in chapter four in a discussion of audio programming paradigms in C, C++, and C#.

```
1  private long sampsElapsed = 0;
2  long sr = 0;
3
4  void Start() {
5      sr = AudioSettings.outputSampleRate;
6  }
7
8  void OnAudioFilterRead(float[] data, int channels)
9  {
10     for(int i = 0; i < data.Length; i += channels)
11     {
12         sampsElapsed = sampsElapsed \% (sr / 2);
13         if(sampsElapsed == 0)
14         {
15             data[i] = 1.0f;
16             //if stereo output then copy left channel
17             //data to right channel
18             if(channels == 2) data[i + 1] = 1.0f;
19         }
20     sampsElapsed++;
21     }
22 }
```

Listing 3.4: Unity code for 120 BPM metronome audio using OnAudioFilterRead(...) function

As of July 2018, Unity Engine does not include built-in support for audio synthesis. In spite of this, the function `OnAudioFilterRead(...)` allows the creation of filters and audio synthesis at a sample level with the use of custom digital signal processing code. The Unity documentation for the `OnAudioFilterRead(...)` function includes code to implement a metronome with the use of a sinewave and an attack-decay envelope and our testing code for `OnAudioFilterRead(...)` in listing 3.4 demonstrates its use to synthesise basic impulse signals. While these techniques are possible, `OnAudioFilterRead(...)` is often not a suitable tool for efficient real-time audio manipulation due to voice allocation problems and difficulties with achieving safe audio memory management in C#.

While Unity Engine does not support audio synthesis in any meaningful way, it does support the use of audio effects through its audio mixer. The Unity Engine audio mixer allows for a number of standard audio effects, such as chorus, reverberation, delay, and basic high and low-pass filtering, to be applied to audio signals in Unity. The audio mixer can also be extended through the creation of

C++ plugins, via Unity's Native Audio SDK. The SDK is a downloadable C++ template for the creation of audio plugins to be implemented in the audio mixer and can be used to create synthesis effects and audio effects. The Native Audio SDK includes a C++ audio callback function that allows for sample accurate audio effect and synthesis programming with minimal latency. Problems with the Native Audio SDK include its poor documentation and a lack of significant usage by the community, leading to few skilled users being available to give advice on its usage. It also lacks meaningful support for MIDI, Open Sound Control, or standard audio plugin formats. Finally the SDK can only send floating point data to and from Unity, and does not support sending string or byte data which is a problem when text and raw data (such as parameters names and raw MIDI messages) need to be sent between languages. Instead of utilising the C++ SDK for audio effect and synthesis programming in Unity, we recommend the use of `OnAudioFilterRead(...)` inside of Unity's C# as explored earlier in this section, which supports C# access to audio DSP callbacks. The `OnAudioFilterRead(...)` function is made particularly powerful when combined with inter-programming language techniques, and its use will be further explored in Chapter 4 of this thesis.

Unity Engine does not include support for popular audio organisation protocols such as MIDI, OSC, or standard audio plugin formats. In spite of this limited built-in support, each of these features can be achieved to a limited degree through the use of external tools. In the early stages of this thesis, there were two tools that supported the use of MIDI in Unity; MIDIJack, a Github project by Keijiro [47], and Tazman Audio's Fabric [48]. MIDIJack is a Unity plugin that supports input from MIDI controllers and input from internal MIDI busses on OSX and Windows computers. Fabric is a more robust solution that is both Unity plugin and audio middleware. Fabric supports the playback of MIDI files, and can use MIDI messages to trigger the playback of audio files and to transpose them, thus creating a pitched sampler. Fabric neither supports in-tool MIDI editing nor editing MIDI files at play time, and is therefore not appropriate for the procedural generation of audio via MIDI. Fabric also includes a VST plugin loader that can be used on Windows systems. The tool can import VST 2 plugins into Unity, but does not support MIDI input, and can be used in the Unity editor, but not in applications due to problems with its build code. During the early months of this thesis research, OSC messages could be utilised in Unity via a GitHub library called UnityOSC, created by jorgegarcia [49]. While external tools for MIDI, OSC, and audio plugin loading exist for Unity, none presents a complete solution for the creation of

procedurally generated audio in Unity. In early 2018, a number of new audio tools for Unity have become available. We will cover these in the conclusion section of the thesis as they were introduced too late to contribute toward our research in Chapter 4.

While Unity Engine fails to present any GUI-based tools for procedural audio synthesis in video games, it does present a high degree of flexibility and extendability as a game audio development environment. Due to the existence of Unity's C# rapid GUI development tools, its ability to access audio callbacks via `OnAudioFilterRead(...)`, and the support for C++ extensions, Unity has significant potential as a platform for procedural audio development, perhaps only rivaled by Unreal Engine in its flexibility and extendability for developing new game audio tools.

3.2.4 Unreal Engine

Like Unity Engine, Epic Games' Unreal Engine is a popular game engine and is frequently used in the creation of Triple-A, Indie, and Virtual Reality games. Due to its widespread use and long history, Unreal Engine has detailed documentation and a large online community of developers. Unreal Engine is open source and is written in C++, yet much programming in Unreal Engine is done via Epic's Blueprints visual scripting environment. As a scripting environment, Blueprints is powerful way to enable people without strong programming backgrounds to create scripts in the engine.

Like Unity Engine, much of the scripting that takes place in Unreal Engine runs on a thread that updates at the framerate. Blueprints scripts run via this method, which means that accurate timing is difficult to achieve with the use of Blueprints alone. The graph in figure 3.4 shows means and standard deviations of timing inaccuracies in Unreal's Blueprints over five iterations of testing via the audio timing tool detailed earlier in this chapter. Figures 3.5 and 3.6 show the Blueprints visual scripts used in the test. The first of these figures shows a blueprints function that has a wave file containing a tick signal loaded into it. This function is called by the script in figure 3.6, which uses the Blueprints Macro 'Set Timer by Function Name' pictured in order to call the function every 0.5 seconds. While the graph in figure 3.4 evidently shows that scheduling audio in Unreal using this built-in loop function is not a feasible way to achieve ideal time-accurate scheduling of percussive signals in Unreal Engine, it does present some interesting and unexpected results. In

the graph presented, the tick event at tick five has a mean time inaccuracy of 0 over five tests. On further inspection, it was observed that, while the test returns unacceptable levels of time inaccuracy with a high degree of standard deviation in results, all timing inaccuracies are either a positive or negative multiple of 441 (10 ms), which is a 100^{th} of the sample rate of 44100. We therefore assume that Unreal Engine's Blueprints' 'Play Sound 2D' object quantises all audio output to the 100^{th} of a second, which, at 10 ms, is an undesired amount of inaccuracy for procedural audio if percussive signals are to be utilised.

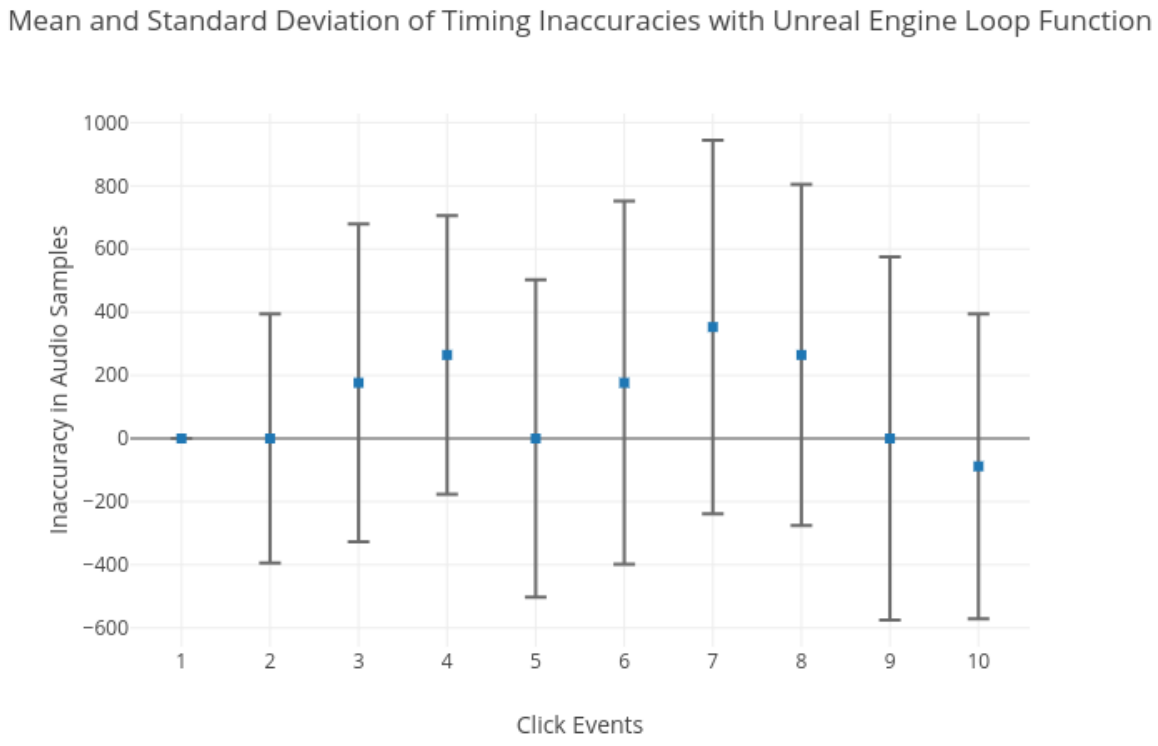

Figure 3.4: Mean and standard deviation of sample inaccuracy using Unreal Engine's 'Set Timer by Function Name' method when tested via the ChucK time-accuracy test in section 3.1

The time inaccuracy found in our testing shows that the use of Blueprints for procedural audio organisation is not a viable solution when accurate timing is required. Initially we attempted to create an audio thread update function for blueprints to remedy blueprints' timing problems, but after conversations with Epic's then-recently-appointed lead audio programmer, Aaron McLeran in early 2017, we decided to abandon the idea [50]. McLeran explained that Blueprints is built to work at the framerate update speed, and an attempt to add an audio-rate

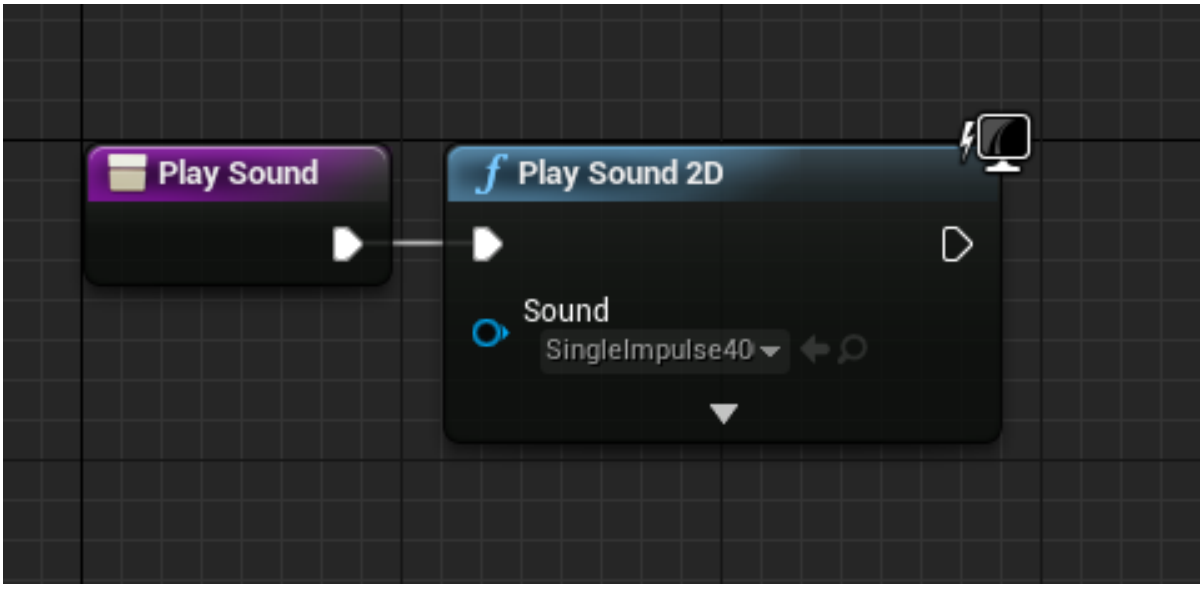

Figure 3.5: A function called PlaySound in Unreal's Blueprints that plays a 'tick' audio file when called

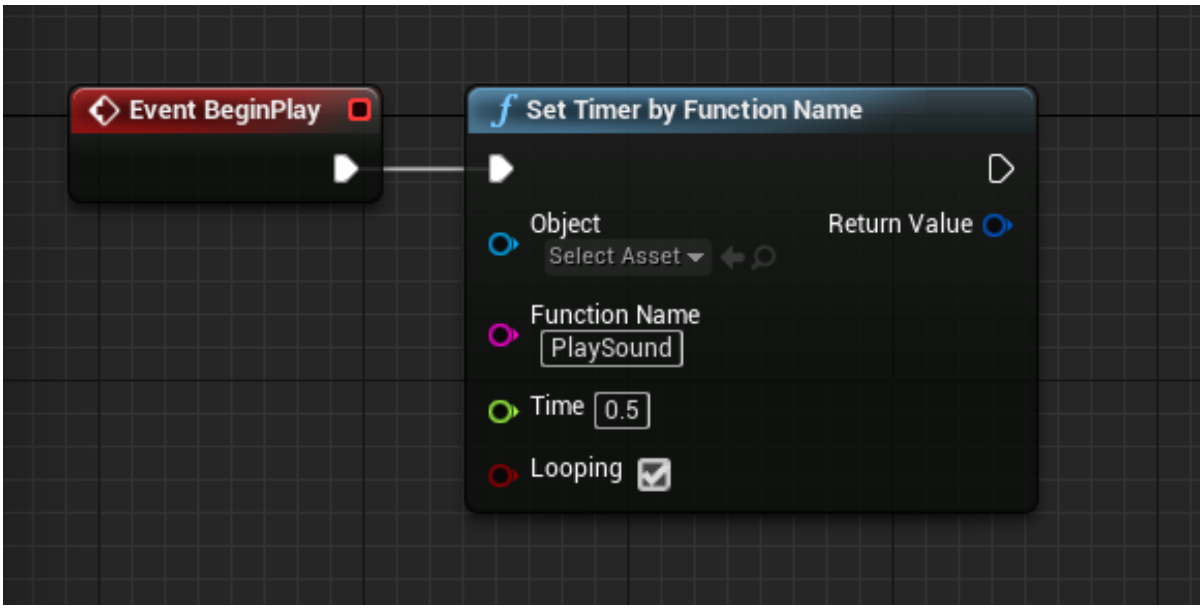

Figure 3.6: A macro in Unreal's Blueprints that repeatedly calls the PlaySound function every 0.5 seconds

update function would need to be carried out by the internal audio team at Epic. McLeran's team has been steadily adding new features to the Unreal audio engine throughout the period of this thesis, and Unreal's audio system is significantly more suited to procedural audio development now in mid-2018 than it was in early 2017.

While audio synthesis was not supported by the Unreal Engine main release branch in early 2017, since that time audio synthesis capabilities have been added [51]. Likewise, accurately timed audio was not possible without significant C++ development at the time of this investigation, yet it is now supported by the engine [52]. At this stage MIDI, OSC, and standard audio plugin formats are not supported in Unreal Engine. Unreal supports the use of audio effects through its Audio Cues system and events created via Audio Cues can be scheduled to play from Blueprints.

While Unreal Engine was not appropriate for procedural audio development in early 2017, throughout the period of this thesis Unreal's audio system underwent large-scale changes, and Unreal Engine is now a powerful environment for procedural audio development. Due to these constant changes, Unreal was not a viable choice for new procedural audio tools development in early 2017, as any tools about to be created for this thesis in Unreal could have potentially been superseded by those created by the audio team at Epic. While the development of procedural audio tools by the Unreal audio team had the potential to supersede our own developments, we firmly believe that having a diverse range of tools for procedural audio across multiple development environments is preferable to having no suitable tools, as was the case at the start of our thesis period.

3.2.5 Section Summary

While each of the game engines and audio middleware environments presented in this section has many features for game audio development, none had meaningful support for procedural audio in March of 2017. Middleware solutions FMOD Studio and Wwise both support sample accurate audio, but are let down by inflexible GUIs and limitations in their extendability. Unity and Unreal allow for larger degrees of extendability, but have fewer audio features than the middleware surveyed. Unity and Unreal Engines also require varying amounts of programming knowledge on the part of their users, which makes them largely inaccessible to non-programmers. All four tools surveyed, which are currently the most popular game

audio development environments, lack meaningful support for audio synthesis, and all but Wwise's editor and Fabric for Unity lack support for GUI-based MIDI file playback.

3.3 Exploration of Early Game Audio Systems

While section 3.2 shows that today's popular game audio environments lack meaningful support for composer, sound designer, and audio implementer-accessible procedural audio, a number of game audio development tools and workflows popular in the 1980s and 1990s had significant support for techniques that are useful in the creation of procedural audio. This section explores two such retro audio workflows in an attempt to gain insight into successful design paradigms for procedural audio systems. While the technology explored in this section is no longer usable in modern game audio development, we feel that it is important to study early approaches to game audio, as there are no modern procedural audio tools for games that meet our design criteria. Our exploration of retro game audio tools and workflows is accomplished through the development of two systems for procedural audio generation with the use of historical technology. In Section 3.3.1 we document an audio system created to run on an NES console, a popular video game console of the 1980s. In Section 3.3.2 we document a MIDI-based game audio system, which emulates audio technology used in many 1990s video games. The strengths and weaknesses of both approaches and their applicability in modern game audio tools development is the focus of this section.

3.3.1 NES

This section details the development of a procedural audio system on NES, with the goal of understanding approaches to the design of retro synthesis and timing systems, and to apply knowledge gained in the creation of new game audio tools later in this thesis. In conformation with historical workflows, we used a composition and transcription method, as discussed in section 2.4, to write music for the NES. In order to create music for the NES, we first wrote a short composition using the software LSDJ. LSDJ runs on the Nintendo Gameboy console, and is synthesis and sequencing 'tracker' software (an audio sequencer organised via a vertical scrolling layout). While NES and Gameboy make use of different audio

synthesis hardware (with the NES using a programmable sound generator (PSG) RP2A03 chip and Gameboy using its main CPU to process audio), the audio output and limitations to polyphony on each game console are closely related. Each is capable of the playback of only four sounds at once, can use basic sampling, and work primarily with basic waveform synthesis. In spite of these similarities, the NES audio chip has a dedicated triangle wave channel that does not exist on the Gameboy.

While the transcription could have been taken from music written in a DAW with MIDI notation, we felt that writing the initial audio in LSDJ had a number of advantages. The similarities of the NES and Gameboy audio systems meant that we could explore extended timbre generation in LSDJ with the knowledge that such effects were highly likely to be achievable on the NES. Also the timing subdivision limitations of LSDJ are the same as those imposed by NES, therefore rhythmic decisions made in LSDJ could be easily ported to NES.

Upon the completion of the LSDJ composition, an audio engine capable of re-creating the composition on an NES console was developed. While much game development for NES in the 1980s was accomplished using assembly language, the game audio system presented here was created with the use of the NES C library CC65. CC65 functioned as a programming abstraction layer, and enabled rapid application development on NES, without the use of hardware-specific assembly programming. In our exploration of the NES audio capabilities, we were particularly interested in exploring how synthesised audio events could be organised to work alongside gameplay. We were also interested in understanding how audio synthesis was accomplished, and whether the design and organisation utilised in the NES for audio synthesis could be applied in the development of the game audio tools later in this thesis.

In order to gain knowledge of early approaches to audio synthesis in video games, with the aim of applying them in modern game audio environments, we explored ways in which audio synthesis was achieved on the NES console. Audio synthesis on the NES required significant knowledge of the NES register layout and eight bit operation codes. The code in listing 3.5 demonstrates the synthesis of a square-wave signal in CC65. While the code presented utilises a number of low-level programming techniques such as bitwise operations and direct pointer access, it also uses abstraction in significant ways. An example of the use of abstraction can be found in line 21 of listing 3.5 in which the note and envelope of the signal are set via an eight-bit hexadecimal operation code.

```
1  //pulse width setting
2  const uchar halfPulse 0x80;
3  //frequency set across two bytes. Grouped for readability
4  const uint16_t Ab2 0xE013;
5  //amplitude set to maximum
6  const uchar maxAmp 0x0F;
7  //envelope set in first nibble of byte
8  const uchar shortEnvelope 0x10;
9  enum {SQUARE_ONE, SQUARE_TWO};
10
11 void playSquare(uint16 note, uchar pulseWidth,
12      uchar amplitude, uchar channel, uchar envelope)
13 {
14     //set offset for pointer to register location
15     channel = channel << 2;
16     //set volume and pulse-width (NES pointer locations are uint16_t type
           )
17     *((uchar*)(0x4000 + channel)) = pulseWidth | amplitude;
18     //set frequency
19     *((uchar*)(0x4002 + channel)) = (uchar)note;
20     //set octave and envelope
21     *((uchar*)(0x4003 + channel)) = envelope + (uchar)(note << 8);
22 }
23
24 //example of function call
25 playSquare(Ab2, halfPulse, maxAmp, SQUARE_ONE, shortEnvelope);
```

Listing 3.5: Square wave synthesis code on NES using CC65 library

The NES PSG chip has a variety of envelope lengths which can be set using the first nibble at register locations responsible for determining the note envelope and an example of its use can be seen in line 21 of the code listing. Amplitude can be set in a similar way and the use of enveloping and amplitude functions means that audio can change volume with a high degree of accuracy over time, without the need for sample-level amplitude specification via code. The choice to use abstraction of audio amplitude over time via presets, rather than user specification of amplitude over time, reduces the parameter-space of the system, but also makes it significantly easier to use. In our own tool development, determining the level of user control over parameters of audio synthesis is a significant design concern that will be further explored in Chapter 4. It is also important to note that all audio synthesis on the NES is done via operation codes that control an audio synthesis chip separate from the main NES CPU. This is a significantly different way of working with audio

than that used in modern game platforms, where audio programming is commonly written on the main CPU, and makes use of multi-threading in order to avoid thread blocking problems with other game processes. This multi-threaded audio system architecture will be further explored in Chapter 4.

Once functions to abstract audio synthesis had been put into place, we moved to exploring the accuracy of audio timing on the NES. This was a significant area of interest (as previously mentioned), as audio timing is an important part of our design criteria, and exploring ways in which it was historically achieved and used is useful in our own development of new time-accurate systems. On NES, the main framerate update function that runs the game logic for NES games has a high degree of accuracy, and so long as tempos used are a clean division of the NES framerate then accurately timed audio is possible. The NES hardware does not support an audio sample rate of 44100 Hz, so our single impulse audio file could not be used to test the system. Also, the NES framerate of 60.0988 frames per second meant that a 120 BPM pulse could not be accurately created. We also found that periodically scheduled and programatticaly identical noise, triangle, and pulse-wave signals have a different starting phase each time that they were scheduled, which made it impossible to use gated signals to test the system for sample accuracy. Due to these limitations and the difficulty of programming on NES, we could not prove conclusively whether NES audio is sample accurate, yet in an informal listening test we could not perceive audible timing fluctuations. It is also important to note that the NES framerate is calculated as a division of the CPU clock speed, and is therefore significantly more accurate than framerates used in modern game environments, such as Unity and Unreal Engine, which do not rely on strict hardware clock timing. Due to the lack of audible time fluctuations and to the CPU-framerate dependency, we therefore infer that the NES audio system achieves our design criteria of time accuracy.

While the system achieved our expectation for time accuracy, it was not able to achieve other design criteria from our list in section 3.1. Standard audio organisation protocols such as MIDI are not inherently supported on the NES. The development of MIDI standardised organisation would have been useful in the organisation of compositional material for our NES application, yet we found it to be ultimately unnecessary due to strict constraints on storage and RAM, and limited octave, rhythmic quantisation, channels, and amplitude possibilities. To accomplish the organisation of audio events over time, we developed a sequencing and looping system. Our audio system made use of a semiquaver tick as its smallest

musical subdivision, which was a standard constraint in game audio systems of the 1980s. Musical organisation was accomplished through the use of arrays of notes for each musical phrase. Each array of notes was 16 semiquavers long, and constituted one musical bar in 4/4 time. Notes were organised by pitch, amplitude, and playback position, as seen in listing 3.6, which shows the programmatic structure of an eight note phrase. The first element in each event description is the pitch to be played, the second is the amplitude, and the third is the semiquaver upon which to play the note.

```
1 uchar bossaBassA[][3] = {
2     {C3, 0x0F, 0x00}, {E3, 0x0A, 0x03},
3     {G3, 0x0F, 0x04}, {E3, 0x0A, 0x06},
4     {C3, 0x0F, 0x08}, {E3, 0x0A, 0x0B},
5     {G3, 0x0F, 0x0C}, {E3, 0x0A, 0x0E}
6 };
```

Listing 3.6: Organisation of musical data on NES with the use of 2D array

To organise each one-bar musical phrase, a higher level system of arrays was utilised, with each array responsible for organising eight bars of audio. Finally, a top level array organised these eight bar sections into songs. This use of three levels of musical organisation for sequenced audio mimics the organisation used in LSDJ, and is a standard approach to audio organisation in a 'tracker' style. With these arrays in place, we could swap out musical phrase choices based on game state changes in real-time, thus achieving limited procedural audio on the NES. A block diagram outlining the layout of the overall NES audio system can be seen in figure 3.7. The diagram presented shows that the game engines' state is used to control the algorithmic composition of phrases, which are then synced to a metronome and synthesised.

The creation of an audio system on the NES console offered insight into the early workings of interactive audio systems, yet many of these discoveries are not applicable in modern game audio environments. The NES's use of a PSG chip for audio synthesis, rather than synthesis via the main CPU, means that the synth design used on the NES is not directly applicable to an integrated modern system. In spite of this, some of the audio synthesis functionality such as the use of abstracted enveloping and parametric amplitude methods on the NES presents applicable design patterns for use in modern game audio systems. Also the NES's framerate-based update function is significantly more time accurate than modern game engines due to the way in which the framerate is scheduled as a direct

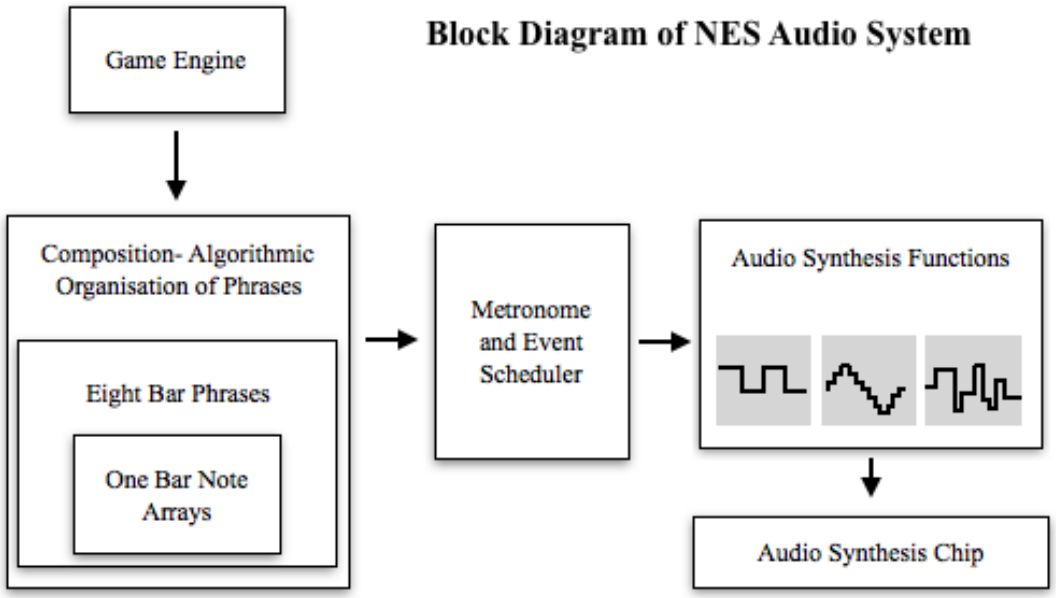

Figure 3.7: Block diagram of data flow in NES audio system

division of the NES CPU's clock. Finally, due to limitations in musical subdivision imposed by the reliance on programming audio via a framerate update function, and due to significant resource scarcity, the implementation of a robust audio sequencing system, such as implemention MIDI file playback, was not feasible on the NES. On the other hand, the sequencing system that we created with the use of different layers of musical organisation proved to be a flexible and robust organisation system for audio sequencing, and presented a number of useful design paradigms that are employed later in this thesis.

3.3.2 1990s-Style MIDI System

In a further exploration of retro approaches to procedural audio in video games, a game audio system for the sequencing of MIDI was created. MIDI is a widely used protocol for the organisation of procedural audio, and this section will present an overview of the MIDI protocol and explore its suitability for use in procedural audio applications through the creation and evaluation of a MIDI-based game audio system. The system created utilises the built-in MIDI synthesiser on a modern desktop computer. MIDI is routed to this synthesiser from a custom-developed MIDI sequencer, and this way of separating synthesiser and sequencer mimics the

way in which computer audio systems of the 1990s worked with audio in games such as Monkey Island 2: Le Chuck's Revenge [7] via an external synthesis system. The creation of the 1990s-style MIDI audio system presented provides insight into musical organisation possibilities of MIDI-based game audio, and also exposes a number of design limitations in the MIDI protocol which inform the creation of procedural game audio tools in the following chapters of this thesis.

We chose to use the interactive programming environment Processing [53] and the programming language Java for our MIDI music system. Java is well suited to the rapid prototyping of interactive applications due to its large online community and extendability as a language. Processing was chosen for the creation of our MIDI system due to its user friendly programming interface and large online support as a platform for visual and audio experimentation. We utilised an extended version of Java, JavaX, which includes a MIDI library that supports the creation and playback of MIDI files.

Audio synthesis in our 1990s-style MIDI game audio system is accomplished through the use of the built-in synthesiser on modern Windows or OSX computers. On current OSX systems, the built-in synthesiser is the DLS-Synthesiser, which utilises MIDI input and an audio sampling format called Soundfonts in order to generate audio. Modern Windows computers use the Microsoft GS Wavetable Synth, which was created by Roland in 1991 and licensed to Windows in 1996, and currently maintains the same Soundfont files of the original licensed synthesiser. Soundfont files were a popular way to organise audio sample banks in game audio of the 1990s, and their use in our application mimics the technology of the era. The Soundfont synthesisers on Windows and OSX computers both make use of MIDI input, and their sound banks and channels align with the 1991 General MIDI (GM) standard, which assigns specific instrument tones to certain ranges of MIDI messages.

In standard audio application programming, event scheduling with the use of MIDI is generally accomplished in one of two ways: event scheduling for real-time applications in which events are scheduled by an external or user-created timing system, or event scheduling with the use of the MIDI file format, in which musical event data is stored for playback. Our MIDI game audio system uses MIDI organised with the MIDI file format, as opposed to using a custom MIDI clocking system. MIDI, when used simply to describe event data without the full MIDI file format, is expressed as three bytes of data. While these bytes can express a number of musical parameters, a common usage is for the description of the start and end

of a musical note event. `[0x90, 60, 127]` is an example of a 'note on' message in which the first byte describes a 'note on' event, the second describes the note pitch: 60 or note C3 in 12-tone equal temperament (12tet), and the third is the amplitude, in this case the maximum amplitude of 127. `[0x80, 60, 0]` is an example of a 'note off' event, which is structured similarly to the 'note on' event but uses a first byte of 0x80 and an amplitude of 0 in byte three to turn the note off.

With these two simple types of MIDI message a wide variety of musical expression can be achieved. The MIDI file format utilises the organisation of musical events in a similar way, but adds timing data to organise the playback of MIDI events. This timing data can be expressed in pulses per quarter note (PPQ), which allows for accurate time division based on a metronome pulse, or via the Society of Motion Picture and Television Engineers format (SMPTE), an organisational data format which uses standardised time-code data, and is not dependant on a metronome.

Through the use of MIDI file playback in JavaX's MIDI library, we were able to import MIDI files created in a DAW directly into Processing, and to schedule their playback in a way that mimics audio workflows used by the developers of Monkey Island 2 with IMuse [7]. In our system, audio is organised into sequences that are played with the Sequencer.Start() method and a vertical remixing system is utilised to organise musical material via adding multiple musical layers to each sequence. An important feature of our system was the creation of procedurally generated audio content system in the form of a simplistic improvising instrument voice. The instrument made use of a first order Markov model with weightings assigned to notes of a blues scale, and was used to test the viability of MIDI file usage in the development of procedural audio content. In order to facilitate the editing of MIDI files, we implemented an 'add note' function. The function can be seen in listing 3.7, and allowed us to specify the pitch, length, channel, and volume of notes as parameter data (see lines 1 and 2 of the listing), and to add them to the existing MIDI files via locking their PPQ to the existing clock (lines 8 and 9).

```
1  private void addNote(Track track, int startTick,
2                int tickLength, int key, int velocity)
3  {
4      ShortMessage on = new ShortMessage();
5      on.setMessage(ShortMessage.NOTE_ON, 2, key, velocity);
6      ShortMessage off = new ShortMessage();
7      off.setMessage(ShortMessage.NOTE_OFF, 2, key, velocity);
8      track.add(new MIDIEvent(on, startTick));
9      track.add(new MIDIEvent(off, startTick + tickLength));
10 }
```

Listing 3.7: Playing MIDI notes in processing with the JavaX library

While the approach to MIDI track editing shown in listing 3.7 allowed the ordering of procedural MIDI events, the track of 'improvised' messages had to be pre-rendered before it could be played. This is due to the structure of PPQ timing in MIDI files, which uses timing data of previous notes in order to determine the timing of current events, and therefore the use of standard MIDI files with PPQ timing is not suitable for low-latency real-time MIDI editing. In order to deal with this limitation, we developed a double buffered system with MIDI tracks that allowed us to procedurally organise MIDI messages. While the system allows for the procedural organisation of MIDI data, it also relies on a large amount of latency (up to eight bars of 4/4 timing) in order to function. This is due to limitations of PPQ timing of MIDI files which means that MIDI tracks that are currently playing cannot be accurately edited.

Our double-buffer system used two empty MIDI tracks, which are looped back to back. We then schedule MIDI notes with the code in listing 3.7 in order to fill the track that is not currently playing, thus creating procedural audio content. Every time the track switches to the alternate track, all MIDI data contained in the recently played MIDI track is deleted.

Like the NES audio system documented in section 3.3, the 1990s-style audio engine presented here could not be accurately tested via our ChucK time accuracy program. This is because it is not possible to load the single pulse wavefile into the built-in synthesiser on OSC or Windows. Also, the Soundfont synthesisers used on Windows and OSX computers use sample randomisation of repeated samples which meant that we could not count the samples between consecutive identical signals or accurately gate signals and measure sample timing between peaks. In spite of these shortcomings, informal listening tests showed no audible

jitter, and due to the widespread use of the JavaX MIDI library in interactive audio applications and lack of documented timing problems online, we assume that the system is accurate to within Haas's 5 ms threshold for human audible time inaccuracy [43].

The 1990s-style MIDI audio system created met a number of our design criteria: the system utilised real-time audio synthesis, uses the standard audio protocols of General MIDI and the MIDI file format, and is assumed to meet time accuracy requirements. On the other hand, a number of significant design criteria could not be met by the tool. While Processing is a popular environment for creative coding, it does not contain many of the standard features of a game engine, and is not a widely used tool for game development. Audio synthesis in real-time can be achieved, but it utilises outdated methods that rely on built-in system MIDI synthesisers that sound 'old fashioned' in a modern game audio environment. Also, real-time changes in procedural audio content were not possible due to the structure of MIDI files. We were highly aware of these limitations throughout our development process, and therefore use the system created only as a prototype for exploration of MIDI in video games from which to draw on later in this thesis.

3.4 Pure Data

With our exploration of historical and current applications of audio synthesis and organisational protocols in game audio complete, we moved on to prototype a game audio system for procedural audio that could be used in a modern game audio environment. In order to undertake this development, the visual programming language Pure Data (Pd) was used. While Pd has been used in the development of game audio in the past, with titles such as Spore [13] and Fract Osc [17] using Pd as their main audio development environment, Pd was never designed to be a game audio tool. With the 2016 release of Enzian Audio's Heavy, a library for converting Pd patches into C for embedding in interactive applications [42], Pd has become suitable for use in modern video game development and can now be embedded as a native plugin into game development environments including Unity and Wwise. Pd is powerful due to its suitability for rapid prototyping of procedural audio behaviours and signal processing. It also has broad online and in-application documentation which makes it an easy to learn tool.

Due to Pd's flexibility in the creation of procedural audio, it was utilised to

implement elements from our NES and Processing prototypes in a way that was usable in modern game audio. The Pd tool created partially emulated the NES audio system's audio synthesis capabilities and used MIDI to organise note playback, two techniques that are difficult to accomplish in modern game engines and middleware environments. We chose to emulate these early game audio systems in order to explore forgotten game audio paradigms, and also to enable users to create retro-style game audio, which is becoming a popular game audio aesthetic with modern game titles such as Shovel Knight [54] and Hyper Light Drifter [55] utilising retro 'chiptune' style soundtracks.

In order to create a modern system in Pd that integrated strengths from both contemporary and retro approaches to game audio synthesis, we created a real-time wavetable synthesiser and used a number of wavetables sampled from an NES console. Pulse waves at four pulse widths, a triangle wave, and noise signals, all sampled from an NES, were utilised. In order to create different pitches using the waveforms, we utilised an interpolating wave table look-up technique which allowed us to keep a consistent sample rate of 44100 samples per second while reading the waveform wavetables at different speeds. While this approach was successful for the transposition of notes based on their fundamental frequencies, notes that are synthesised at frequencies higher than the fundamental frequency of the sampled wave used in our wavetable caused aliasing artifacts as their higher harmonics exceeded the Nyquist frequency of our system. A standard technique to solve this problem is the utilisation of a collection of single-period waveforms, sampled at different frequencies, that can be changed between depending on the note to be synthesised. When low notes are played, then harmonically dense waveforms sampled at low frequencies are utilised, when higher notes are played, then samples recorded at higher frequencies with less dense harmonic spectra are used, therefore removing aliasing artifacts. While this approach would have yielded the best results for our system, our Pd synth was a prototype, and the creation of alias-free signals was out of scope in our prototype development phase. On the other hand, realisations of aliasing problems in wavetable-based synthesis informed our explorations of real-time audio synthesis in tool developments documented in Chapter 4. The Pd GUI controlling audio synthesis can be seen in figure 3.8, which includes GUI elements for controlling the lowpass filtering, wavetables, low frequency oscillators, and enveloping of generated signals.

Alongside implementing basic synthesis features from the NES such as attack-release envelopes, variable pulse width signals, a triangle wave, and noise syn-

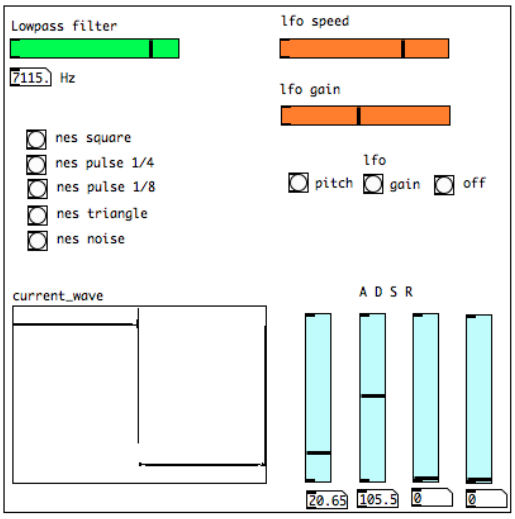

Figure 3.8: Synthesis GUI for Pd patch

thesis, features that extended the NES design based on common features of modern synthesisers are included. The addition of a lowpass filter allowed us to use subtractive synthesis techniques to shape the tone of the synthesiser. The addition of low frequency oscillators for pitch and frequency allowed parameter modulation and through expanding the modulation speed range available our plugin can achieve frequency modulation (FM) effects and amplitude modulation (AM) effects. We expanded the attack-release two state envelope system from the NES by adding sustain and decay parameters, thus implementing an ADSR envelope. The patch uses a fixed sustain duration set to 0.2 seconds as scheduling note off-type events was not practical with our GUI layout, but a more robust sequencing environment would allow for more control over note lengths.

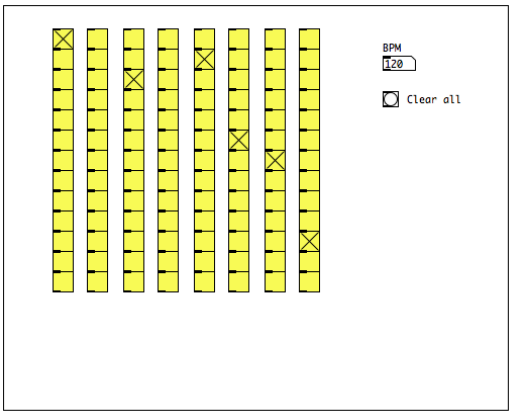

Figure 3.9: Sequencer section of Pd patch

Another significant feature of our Pd synthesiser was the use of accurately timed

musical events that could be sequenced and looped with a high degree of rhythmic accuracy. We used an event scheduling interface which controls audio in sixteenth notes, and allowed for notes to be scheduled via a custom GUI, seen in figure 3.9, with a MIDI link to the synthesiser section of the patch. The GUI pictured is an eight by twelve matrix of toggle boxes with the vertical axis corresponding to pitch (in 12 tone equal temperament from middle C in semitones). Horizontal spacing represents time (in semiquavers), and the material sequenced via the GUI is looped continuously in the style of a musical sequencer. Rather than utilising the MIDI file protocol, we implemented our own real-time MIDI organisation that bypasses problems documented in section 3.4 with real-time scheduling of MIDI events using MIDI files. Through testing this Pd procedural audio tool with our ChucK timing program, we found that our Pd application exhibits a small amount of sample inaccuracy, but the amount is below the human threshold for audible latency of 5 ms (220.5 samples), as can be seen in the graph in figure 3.10. The graph, which plots mean time inconsistencies across five iterations of testing, shows that while there is time inaccuracy, it complies to a predictable pattern of always being slightly before the correct time (thus the negative values), and is all under the human audible threshold of time inaccuracy. Control rate data in Pd is run every 64 audio samples and the data shown conforms to expectations associated with such a system. With that said, the accuracy of of the Pd timing system created will slowly move away from correct timing over time, and if played alongside a sample-accurate 120 BPM signal, would eventually become out of sync with the signal. This could be a problem if our system were to be used alongside other audio clocks, and demonstrates a design flaw in the standard Pd audio timing design pattern.

The code in figure 3.11 was used by our system to organise audio events, and the metro object pictured, which drives audio timing, was the standard timing tool used in Pd development. While the metro object presented timing inaccuracies that could become significant problems over long periods of time due to drift, Pd is capable of sample accurate audio as documented by Eric Lyon in their paper "A Sample Accurate Triggering System for Pd and Max/MSP" [56]. Lyon's method is applicable in the creation of time accurate event scheduling for use in Pd as a standalone application, but it requires the use of an externally created Pd object which is not supported for use in Heavy, and therefore is not usable in a modern game audio context.

While our Pd tool met all four of our design criteria documented in section 3.1 to varying degrees, it also presented a fundamental problem that led us to pursue

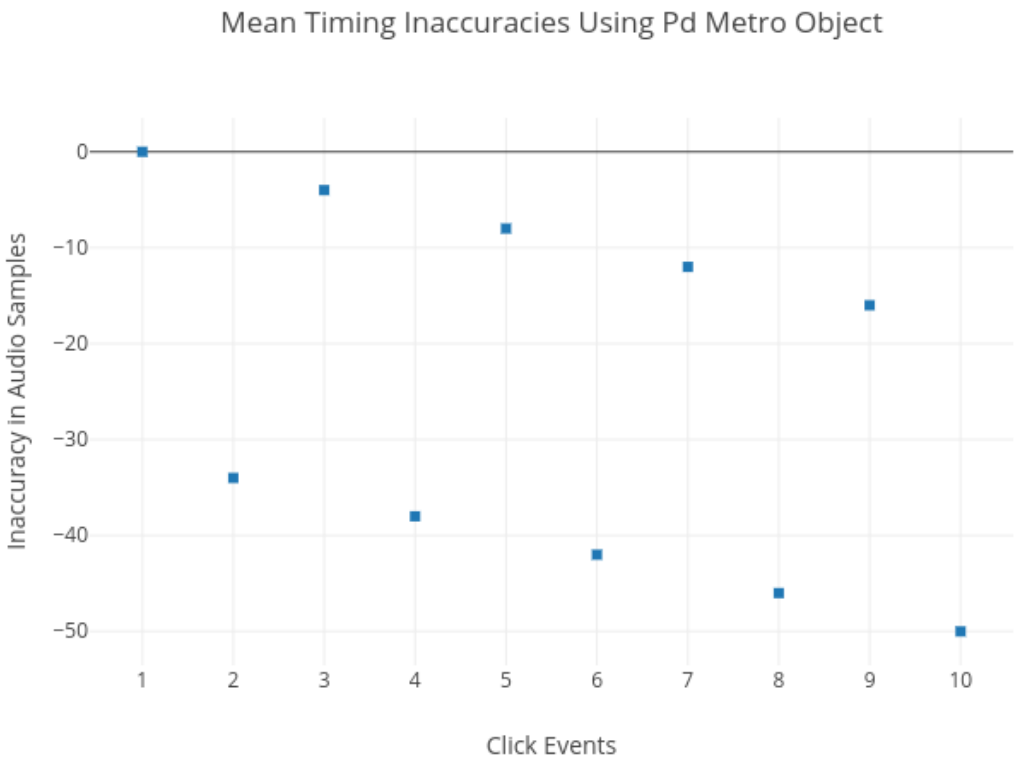

Figure 3.10: Mean audio timing inaccuracies in Pd procedural audio system using metro object. This data is generated using the test in section 3.1 and shows a consistent negative drift present in the Pd metronome object utilised. Note that standard deviation is not shown as there was no deviation present

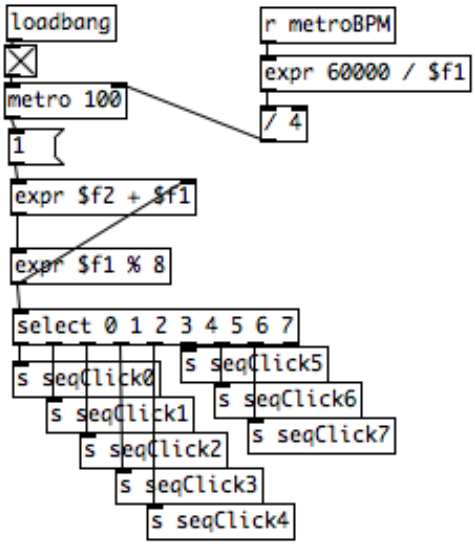

Figure 3.11: Metronome that drives sequencer timing in our Pd prototype

different software solutions later in this thesis. Pd's use of a graphical programming interface made it very difficult to store large amounts of data. This includes the storage of large transition matrices for Markov models, a useful technique for creating algorithmic musical material as explored in Chapter 2 of this thesis. Pd's lack of control over memory allocation, in comparison to languages such as C++, C, and C#, means that optimisation of resource intensive procedural audio techniques including higher order Markov models and neural networks is inappropriate in Pd. While the traditional approach to solving these problems for a Pd user would have been to create a custom Pd object with the use of the Pd C object development SDK, this was not possible in our application as the game-oriented Heavy library used does not support the compilation of custom Pd externals. Another approach would have been to embed resource-heavy procedural audio processes in the Game Engine running the Pd patch, and then send data to and from Pd via MIDI or OSC. This would likely have been a powerful procedural audio workflow, but once such a workflow had been created, solutions that bypass Pd would quickly have become preferable to the use of Pd. For example, Pd's timing problems documented in figure 3.10, which shows a steady drift away from correct metronomic timing, could be bypassed with the creation of a sample accurate metronome and audio organisation system in a programming language such as C, C#, or C++. If Audio timing and algorithmic logic were moved into the game engine, Pd would only be responsible for audio synthesis. While Pd's audio synthesis is robust and easy-to-use, we predict that the quantity of inter-language communication that would be required in order to use such a system would have quickly outweighed the benefits

that Pd brings.

3.5 Chapter Summary

This chapter has examined methods for achieving procedural audio in a range of game development environments. Through our exploration of modern game audio development environments, we found a distinct lack of support for important features of a procedural audio system. In order to understand possible ways to address these issues, we looked to retro game audio workflows of the 1980s and 1990s, and became aware of a number of useful design paradigms for use in procedural game audio tools creation. We went on to apply these findings to the creation of a synthesiser and sequencer using the visual programming language Pd. Our Pd patch fulfilled our design criteria, but failed to prove a robust environment for more fundamental aspects of our research goals in terms of data organisation. A summary of our findings can be seen in figure 3.12, which shows that at the time of this investigation none of the game audio environments explored in this chapter filled the gap in the field of procedural game audio tools. In order to address this lack of support, the following chapter presents the development of two custom tools for the creation of procedural game audio.

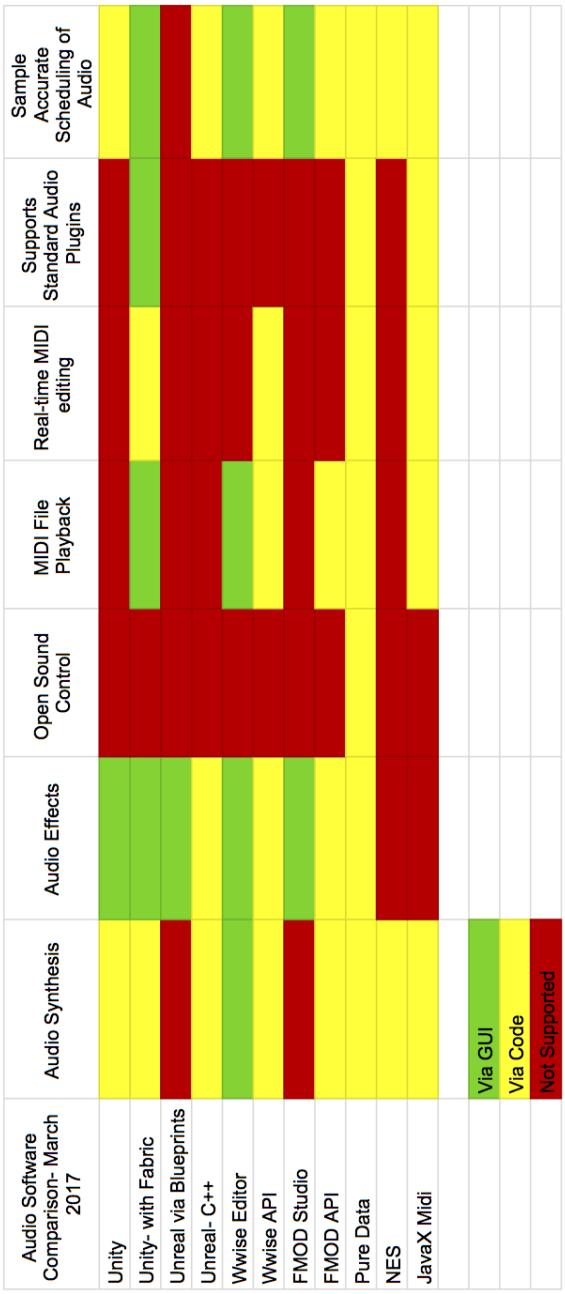

Figure 3.12: A feature comparison of software examined in this chapter

Chapter 4

Implementation

In the preceding three chapters, we have documented the lack of procedural audio tools available to modern game audio developers. In order to address this deficit, this chapter explores the development of two new tools for the creation of procedural game audio. In section 4.1 we discuss the development of a new MIDI tool for procedural audio development in Unity Engine which has the ability to interface with existing audio systems on modern personal computers. In section 4.2 we focus on audio effects in procedural game audio and create a plugin hosting tool that allows VST2 plugins, a commonly used audio effect plugin format in DAWs, to be used in Unity. All game audio tools documented in this chapter are made solely for use with Unity Engine, which is chosen as our development platform as discussed in section 3.2.3. The tools have been released as open source code and are hosted on github for public download [57][58].

In order to evaluate the tools developed in this chapter, two sets of evaluative criteria are used. First, the set of high-level design criteria introduced in Chapter 1 are utilised. In summary, these criteria require that the tools developed are able to create procedurally generated audio and that these tools be accessible for use via a GUI or scripting API. The second set of evaluative criteria (detailed in section 3.1) require that the audio tools developed achieve professional-level timing accuracy, achieve real-time DSP, and use existing audio protocols when possible.

4.1 Tool 1: Unity MIDI System

The first tool created to support the procedural generation of audio is a library for procedural MIDI manipulation in Unity. As discussed throughout the previous chapters of this thesis, MIDI is a highly appropriate format for use in procedural audio development, yet support for MIDI in modern game audio environments is limited. In Chapter 3, two existing tools that utilise MIDI in Unity are introduced: MIDIJack and Fabric Audio, yet neither tool allows for the algorithmic ordering of MIDI messages in real-time, a core technique in procedural audio. In order to fill this gap, a new Unity MIDI tool is created. The discussion of this Unity MIDI tool is documented over the next four sections which each outline design concerns that arise in its development. Section 4.1.1 provides a broad overview of the tool, section 4.1.2 discusses approaches to inter-programming language communication utilised in the tool, section 4.1.3 compares the system created to existing MIDI tools, and section 4.1.4 evaluates the tool's success.

In order to meet general and technical design criteria, the MIDI tool created should have the following technical attributes:

1. The tool should be able to read and play back standard MIDI files without error at run-time.
2. The tool should be able to re-order MIDI messages in either a MIDI file or as individual events at run-time without errors occurring, thus enabling the tool to be utilised in the creation of algorithmic composition.
3. The tool should be able to schedule individual MIDI messages and MIDI tracks for time-accurate play back (as defined in section 3.1) using Unity's existing audio timing system.
4. MIDI output by the tool should be routable to the software MIDI ports of a modern Windows computer for external synthesis.

4.1.1 Overview of Tool

The MIDI library described in this section is developed to support the play back, re-ordering, and dynamic creation of MIDI data in Unity Engine. The system is developed across both C# and C programming languages and is split into two main sections: a section that controls MIDI parsing and organisation, and a section that

controls the output of MIDI data. The diagram in figure 4.1 presents an outline of the tool's structure. As seen in the figure, a mixture of static and non-static classes make up the C# section of the tool, with the static classes organising MIDI output and general utilities and the non-static classes storing and organising MIDI data. All blocks in the figure represent C# classes with the exception of the block titled `portmidi.dll`. This block represents a dynamic library which exposes a variety of functions from the C MIDI I/O library Portmidi [59] for use in Unity.

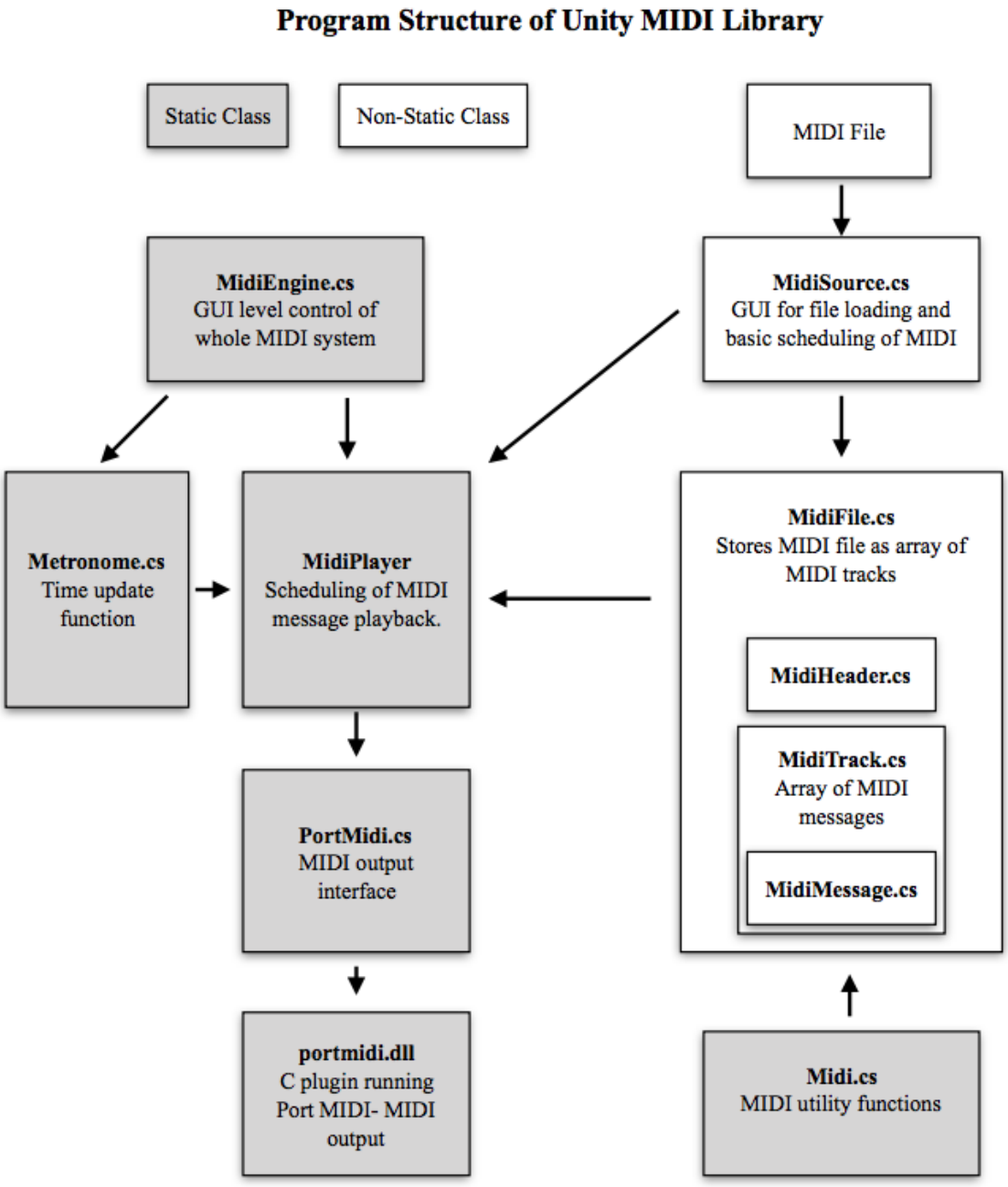

Figure 4.1: Block diagram and dataflow of Unity MIDI system

While our tool supports a number of methods of MIDI manipulation, a core feature is its ability to play back pre-rendered MIDI files. In order to support the play back of such files, we developed a MIDI file reader. In our file reader, MIDI files are loaded into Unity via a loading component that is displayed in the Unity Editor. Our custom-developed GUI for the component is shown in figure 4.2. The GUI includes a field for loading MIDI files, a collection of toggles for controlling muting and playback time, and two horizontal sliders that allow the setting of volume and pitch offsets.

As MIDI files are loaded, we split them into individual MIDI track objects, each consisting of MIDI message objects. Each MIDI file also includes a single MIDI header to store meta-data about the MIDI file. MIDI headers are 14 bytes long and store data outlining tempo, PPQ or SMPTE timing division, and MIDI file type. Following a 14-byte header, the rest of a MIDI file consists of MIDI tracks. MIDI files can have one or more MIDI tracks and each MIDI track consists of a short header followed by MIDI event messages such as note on and note off messages alongside meta events such as tempo changes and lyrics.

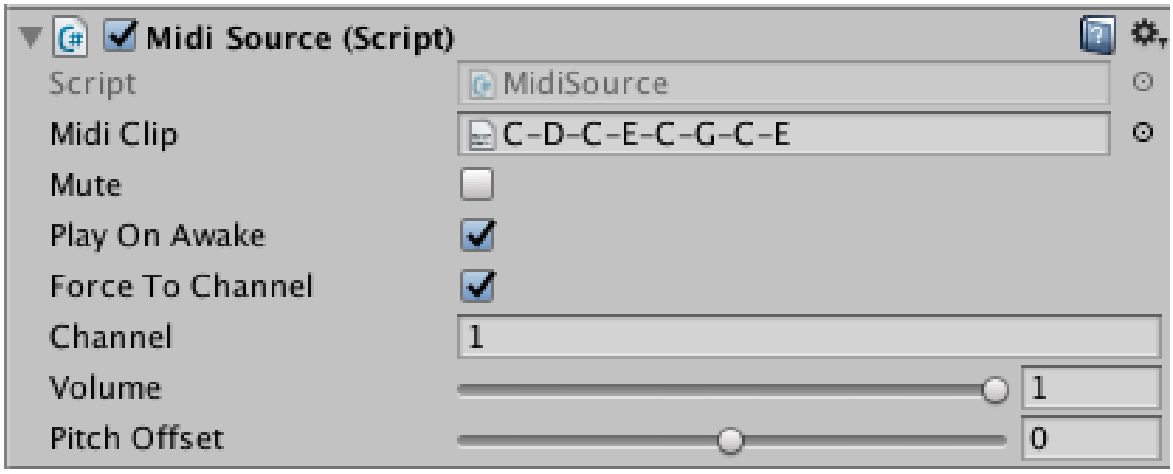

Figure 4.2: MIDI Source GUI in Unity that can load MIDI files and mirrors many of the parameters of the Unity AudioSource GUI component for wave file play back

We store MIDI header data in a `MidiFile` object, which dynamically allocates memory for each MIDI track based on metadata from track and file headers. Individual MIDI tracks in a MIDI file are then stored via our `MidiTrack` class, with each track containing a linked list of MIDI MidiMessage objects extracted from the MIDI file coupled with meta-data from the MIDI track's header.

One important feature of our MIDI file reading system is the way in which MIDI

message timestamps are stored. Our MIDI timestamps are stored as PPQ ticks since the start of a MIDI file, which differs from the approach taken by the standard MIDI file format. In the standard MIDI file format, MIDI event timestamps store the amount of time in PPQ ticks until the next MIDIMessage message will occur. This relative storage of MIDI timestamps based on inter-event timing means that the MIDI file format can store timestamps very efficiently (the protocol utilises the least number of bytes possible for every timestamp and often requires only one or two bytes to store each value). While this is an efficient system, it can lead to problems when real-time manipulation of MIDI data occurs. In such a system, individual MIDI events cannot be scheduled to play unless the time until the following note is known. This limits the feasibility of updating MIDI files in real time, and makes traditional MIDI files a poor host for real-time procedural audio. This is the case in video games, where a close synchronisation between audio and visual content rules out the viability of large latencies between MIDI message writing and play back.

The use of absolute timestamps in our MIDI library means that MIDI tracks can be written and read in real-time. Our absolute timestamps are stored as 32-bit integers, larger than the relative timestamps used in the MIDI file protocol, but more suited to our utility. This increase in size is not a problem, as modern personal computers have exponentially more RAM and storage space than the 1980s digital systems MIDI files were designed for. Our MIDI library is made to play modern MIDI files which are made up of basic MIDI 'note on' and 'note off' messages, and MIDI header data. The tool does not support the reading of MIDI files that utilise extended MIDI features such as lyric reading and tempo changes. We choose not to support these features as they have rarely been used in professional DAW and MIDI editing contexts since the late 1990s and require significant amounts of development time to support.

The standard MIDI protocol utilises three types of MIDI file- type-zero, type-one, and type-two MIDI. Our system supports the loading of type-zero MIDI files, which contain a maximum of only one MIDI track, and type-one MIDI files, which can contain up to 127 MIDI tracks. We do not support the play back of type-two MIDI files in our tool due to their relative rarity in modern applications. Alongside the play back of pre-rendered MIDI files, the MIDI library presented also provides a number of functions for procedurally editing MIDI files in real-time and for scheduling MIDI events for play back. These features will be explored in section 4.1.3.

Once MIDI files and procedurally scheduled MIDI messages have been stored by our tool, they can be played back via the `MidiPlayer` class. The `MidiPlayer` class takes an input of a `MidiMessage` or `MidiTrack` object, and plays it back at a user-specified time. The `MidiPlayer` class requires a BPM timing setting and MIDI output specification, which are set via the `MidiEngine` class's GUI, seen in figure 4.3.

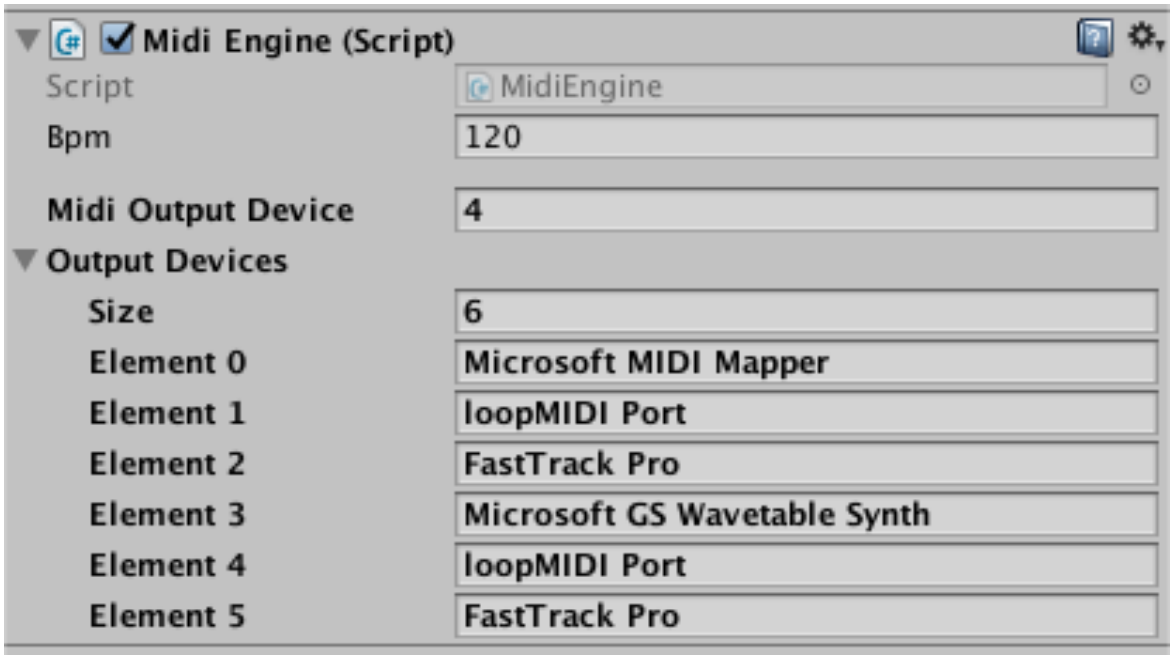

Figure 4.3: MIDI Engine GUI interface showing the MIDI output choices available on the Windows desktop computer used in development

In the `MidiPlayer` class, `MidiMessage` objects are added to a dynamically sized array, which orders events by timestamp in relation to application start. MIDI message timestamps are checked against the current system time and if an event's timestamp is within two visual frames of the current time, it is sent via the C# `Portmidi` class to C for output[1].

The C dynamic library used to output MIDI consists of an edited version of the Portmidi C library. We add a number of functions to Portmidi in order to support data transfer between C# and Unity and for debugging purposes. A closer look at our edited version of Portmidi and the way in which data is sent between C and C# is explored in the following section.

[1]We send MIDI messages to C at the last possible moment because Portmidi has a maximum MIDI event buffer size of 127 events

4.1.2 Inter-Programming Language Communication

We utilise the C library Portmidi to enable MIDI output on Windows computers from our application. Portmidi is chosen due to its multi-platform support and wide use as an open source MIDI I/O library. An alternative multi-platform MIDI I/O library exists called RTMidi, but we chose to use Portmidi as it is more widely used than RTMidi. Inter-programming language communication is used to send MIDI event and timestamp data from C# to the Portmidi library via the C# library `System.Runtime.InteropServices`, which supports the static import of functions from the compiled Portmidi C dynamic library into C#. In C#, functions can be imported via the code in listing 4.1, which uses the dll name and function name to locate the function in the C file.

```
1 [dllImport("Portmidi", EntryPoint = "midiEvent")]
2 public static extern void midiEvent(int status, int mess1, int mess2, int
      delay);
```

Listing 4.1: C# export code for functions to C

The code in listing 4.2 shows the registering of the function from listing 4.1 as a C function via dll export, and is used here to expose the function in lines four to ten which schedules MIDI events.

```
1  __declspec(dllexport) void midiEvent(int status, int mess1, int mess2,
       int delay)
2
3  PmEvent event;
4  void midiEvent(int status, int mess1, int mess2, int delay)
5  {
6      timestamp = TIME_PROC(TIME_INFO); //current time
7      event.timestamp = timestamp + delay;
8      event.message = Pm_Message(status, mess, mess2);
9      Pm_Write(event);
10 }
```

Listing 4.2: dll function exporting from C of the midiEvent function for scheduling MIDI event play back

Once functions have been declared exportable in C and imported into C#, they can be utilised in Unity Engine. A problem with the calling of C functions from within C# is that data sent from C# may not be readable in C and vice versa. Basic types such as integers, floats, characters, and doubles can be sent as function

parameters or return values without issue, but the sending of non-basic data types such as arrays and pointers is more difficult. In order to send such data, C#'s interopServices' `IntPtr` object can be utilised to handle raw pointers in C# and a collection of functions for extracting data from de-referenced pointers exists in C#. We utilise the `IntPtr` object when sending text arrays for setting MIDI output channels from C to C# via the code in listing 4.3, which shows the C# and C functions used to transfer the data. Line 20 of the listing demonstrates the use of a function from the `Marshal` class which is used to extract string data from the raw pointer. The `Marshal` class is part of C#'s `interopServices`library and includes a number of member functions useful when using pointer data in C#, a language that traditionally does not support the use of raw pointers.

```
1  //C:
2  __declspec(dllexport) char* printOutputDevice(int index);
3
4  char* printOutputDevice(int index)
5  {
6      //read device information by device index
7      const PmDeviceInfo* info = Pm_GetDeviceInfo(index);
8      return info->name;
9  }
10
11 //C#
12 [dllImport("Portmidi", EntryPoint = "printOutputDevice")]
13 public static extern IntPtr printOutputDevice(int index);
14
15 numDevices = Portmidi.Pm_CountDevices();
16 outputDevices = new string[numDevices];
17 for(int i = 0; i < numDevices; i++)
18 {
19     //convert raw pointer to Ansi-formatted string
20     outputDevices[i] = Marshal.PtrToStringAnsi(Portmidi.printOutputDevice
           (i));
21 }
```

Listing 4.3: Sending text data from C to C# via the use of raw pointers and functions via C#'s interopServices library

While the use of inter-language communication in Unity to run C dlls is a powerful way to leverage behaviours not available in C#, it also has disadvantages. The Unity editor does not support hot reloading of dynamic libraries, and a full Unity Engine re-boot is required each time a dynamic library file is loaded. This leads

to a slow development cycle and proved to be a significant factor in the speed of our tool development. Also, debugging native C and C++ code from C# is not supported in Unity and any memory leaks or run-time bugs in the native code will cause application crashes, rather than being caught by Unity's debugger. In order to support debugging and to speed up development time, a custom Unity debugger for native code has been developed in the course of this research. The debugger uses the pattern for inter-language text data sending introduced in listing 4.3 in order to send debug messages from C to C#. When the C dll is run in debug mode, functions that have the potential for failure are checked for errors, and if such an error occurs the dll ceases activity and an error message is sent to Unity. We highly recommend the creation of such a debug tool for all developers undertaking Unity dll development due to its ability to greatly speed up development.

4.1.3 Comparison to Existing C# MIDI Tools

While C# does not include a built-in MIDI library, the play back of MIDI on personal computers with C# has previously been achieved. Sadly, existing tools do not not achieve a number of the features required of a successful procedural audio system according to our design criteria and this section outlines ways in which our tool differs from existing MIDI solutions in C#. Two tools for the manipulation and output of MIDI files in C# exist: Github user obiwanjacobi's library midi.net [60] and Leslie Sanford's C# MIDI Toolkit [61]. midi.net supports the play back of MIDI files through the use of platform invoke calls to native code, which work solely on Windows machines. The library gives users access to Windows' system-level MIDI output. MIDI Toolkit also supports the loading of MIDI files on Windows systems and supports output to Windows MIDI outputs via connection to the Windows Multimedia MIDI output stream API. While both systems achieve MIDI file play back, there are a number of factors that led us to creating a custom solution, rather than utilising a pre-existing one.

One way in which our system differs from existing C# MIDI libraries is the way in which MIDI timestamp data is organised. While existing MIDI organisation tools utilise relative PPQ or SMPTE timing to store event timestamps, we store timestamps as PPQ since track start, as discussed in section 4.1.1. This means that MIDI messages can be scheduled for play back in real-time, rather than requiring MIDI tracks to be pre-rendered, making our system well suited to the low-latency procedural generation of audio. A third advantageous feature of our library is the

way in which it closely interfaces with Unity. The timing system of our tool is written solely for use with Unity Engine and makes use of Unity's `AudioSettings.dspTime` timer, a highly accurate timer which is explored in section 3.2 of this thesis. The use of Unity's timing system in our tool means MIDI events can be accurately timed to coincide with Unity system events. This differs from the timing systems of existing C# MIDI tools, which are applicable in non-Unity applications, but lack the advantages that Unity timing gives when working alongside existing Unity systems.

Another feature of our system that differs from existing C# solutions is its potential for multi-platform support. Through the use of Portmidi, our system has significantly higher potential for multi-platform support than either of the existing C# MIDI libraries which are limited to Windows computers. This is a useful feature as Unity is a multi-platform environment and limiting the tool to Windows significantly reduces its usability. That being said, due to time constraints in our development process, we currently only supply a Windows build of our dynamic library for download. This is because we develop and test the tool on Windows, yet the creation of dynamic library files for unix-based systems would require little effort as much of the cross platform code is already in place.

`AudioSettings.dspTime` can be used in our system to directly set MIDI messages to play at specified times via the function `MidiPlayer.PlayScheduled(MidiMessage, dspTime)` which takes a MIDI message input and a Unity dsp time in seconds (as a double precision floating pointer number) in order to schedule MIDI events. This pattern emulates Unity's `AudioSource.PlayScheduled()` function, which schedules audio file play back and is documented in our discussion of Unity Engine's audio timing in section 3.2. This similarity in syntax is utilised in order to enable Unity users with programming knowledge to quickly access our library without learning new design patterns. We also implement a `MidiPlayer.Play(MidiMessage)` function, which mimics Unity's `AudioSource.Play()` function, and causes a MIDI message to play at function call. Through the availability of these two functions and the use of absolute timestamps in our MIDI file storage (allowing audio implementers to edit MIDI files in real-time), our tool supports a level of procedural audio that cannot be achieved by existing Unity MIDI tools, and nor can it be achieved by existing C# MIDI libraries.

4.1.4 Section Results

In this section, we test whether the developed MIDI library meets the evaluative criteria detailed in chapters 1 and 3 of this thesis in order to determine the tool's suitability for the creation of procedural game audio. The MIDI library's ability to meet evaluative criteria can often be understood through our preceding documentation, but in some situations further testing is required.

As described in section 4.1.1, the tool has both a GUI and a programming API, both of which are based on the style of Unity's own scripting and GUI design, thus meeting a core design criteria outlined in Chapter 1. In order to understand whether this tool meets the second core evaluative criteria of this thesis, support for procedural audio, the tool should meet the technical criteria from Chapter 3 and contain the technical attributes introduced in section 4.1.

The tool presented makes widespread use of the MIDI protocol, and therefore achieves the first of our three technical evaluative criteria. In order to understand the extent to which the tool achieves support for MIDI, we test the tool's ability to accurately play back MIDI files later in this section. The second technical evaluative criteria requires support for time-accurate audio. This is linked to technical requirement 2 from the start of this chapter and once again requires testing, which will be carried out later in this section. The final technical evaluative criteria from Chapter 3, support for audio synthesis or audio effect processing, cannot be met directly by our MIDI tool as it has no DSP functionality. While this is the case, the ability of our Unity MIDI library to output MIDI messages via software MIDI ports, as explored in section 4.1.1, means that audio synthesis and effects can be accomplished through the use of external tools. This includes the modular use of DAWs and hardware synthesisers for synthesis and effect processing, and it is therefore unnecessary for our tool to internally support DSP.

Finally, as explored in section 4.1.3, our tool is capable of procedurally editing MIDI track data and can schedule MIDI events using a collection of custom C# functions, thus achieving technical attribute 2 from section 4.1.1. While these functions are implemented, they rely on the same mechanisms that enable MIDI file play back and the time accurate output of MIDI, and therefore their degree of success will be best understood once the outcome of the tests carried out in the remainder of this section are known.

In order to test the tool's ability to play MIDI files, we create three MIDI files using

Reaper that utilise different aspects of the MIDI standard. We load each file into our Unity MIDI player and then route output MIDI data back into Reaper (with the use of MIDI Loop, a MIDI routing solution for Windows [62]), where the MIDI data is recorded in real-time and compared to the original MIDI files. The test is used to check that note channel data, amplitude, and pitch data is accurately read and output by the tool, and to see whether MIDI data is time-accurate once processed by the tool. We do not test for our tool's ability to read MIDI files that include meta-data within MIDI tracks, as these are not supported for play back by our tool (see section 4.1.1).

The first of three MIDI files used to test our system's MIDI file play back is a one octave ascending C Major scale starting on middle C in crotchets. A C Major scale is chosen because it is the most widely used 12 tet scale, consisting of the white notes on a keyboard from middle C up an octave. Crotchets are chosen as they make time-accuracy testing easy in a second-based system, as each note should be exactly half a second from the previous note at 120 BPM. In our test MIDI file, amplitude data starts at a maximum amplitude of 0x7F on the first note and reduces amplitude by 18 every subsequent note, ending at an amplitude 0x01 on C5. All MIDI data is set to MIDI channel 1 and the file is stored as a type-zero MIDI file, meaning that only one MIDI track can be present. The second MIDI test is identical to the first test, except a type-one MIDI file is used rather than a type-0. The final test utilises more complex MIDI features and consists of a one bar phrase from the Kyrie Eleison section of Bach's Mass in B Minor. The type-one MIDI file includes musical data split across five MIDI tracks corresponding to vocal parts in the mass, each with an assigned channel. While the original phrase is written at a single amplitude, we adjust the amplitude of each track to test for problems occurring in play back.

The outcome of all three MIDI file tests can be seen in figure 4.1. The figure shows that our tool accurately reproduces MIDI note, amplitude, channel, and track data, but is not sample accurate (to the 44100^{th} of a second) in any of our tests. In order to collect this data we compare the output Unity MIDI files against the original MIDI in Reaper and check each note against the original MIDI file. MIDI parsed through our Unity MIDI system is not accurate to the sample, yet the threshold for audible time inaccuracies is 5 ms (220.5 samples) for percussive signals and is 30 ms (1323 samples) for non-percussive signals [43] meaning that exact sample accuracy is not necessarily required. We therefore undertake a further test in order to understand whether timing inaccuracies of our MIDI tool are under one or both of Haas' thresholds. To test time inaccuracy, we use the test introduced in Chapter

3 of this thesis which uses a ChucK timing test program that analyses five iterations of impulse signals synthesised at 120 BPM in order to test for time inaccuracies. We use the MIDI output from MIDI file test 1, which is four seconds of MIDI notes each spaced 500 ms apart, and measures the mean and standard deviation of time inaccuracy in the signal. Results of the testing can be seen in figure 4.4 and show that our Unity MIDI system has a potential for time inaccuracy of up to 630 samples (14 ms). It is important to note that the maximum time inaccuracy between events is higher than 630 samples because signals can be positively and negatively displaced, therefore a potential audible inaccuracy of up to 1260 samples (28 ms) can be inferred. This does not meet our ideal time inaccuracy threshold of 5 ms (or 220 samples at a sample rate of 44.1kHz), but is below our higher threshold for non-percussive signal inaccuracy of 30 ms (1323 samples) and therefore passes our minimum requirement for time accuracy.

	Channel Errors	Track Errors	Pitch Errors	Amplitude Errors	Sample Jitter (excluding first note)
Test 1	0/8	0/8	0/8	0/8	7/7
Test 2	0/8	0/8	0/8	0/8	7/7
Test 3	0/15	0/15	0/15	0/15	14/14

Table 4.1: MIDI file read errors once files have been read and output by our Unity MIDI library. The data shows that time inaccuracies are the only issues in the system's output

In order to understand the source of the inaccuracy, we run a number of debugging checks. At first we collect timestamp info as audio is scheduled in Unity and check whether MIDI timestamps are being incorrectly calculated. All timestamp data used internally in our Unity plugin is stored as PPQ values (in integers) and is translated into seconds at note schedule time (in double precision floats). When checked, both types of timestamp proved to be accurate to the highest degree possible (a 960^{th} of a crotchet in PPQ and a 64 bit double precision floating point number in seconds). We then test the MIDI timestamp data once it is translated into milliseconds by Portmidi. At this stage, timestamp data loses accuracy as it is translated into millisecond delays in integers, but this rounding error cannot account for the up to 16 ms of time inaccuracy measured. In order to check the timing of MIDI message scheduling times from Portmidi, we route Portmidi timestamp data from C to the Unity debugger. The results show that Portmidi's timing function `timestamp = TIME_PROC(TIME_INFO)` is not outputting

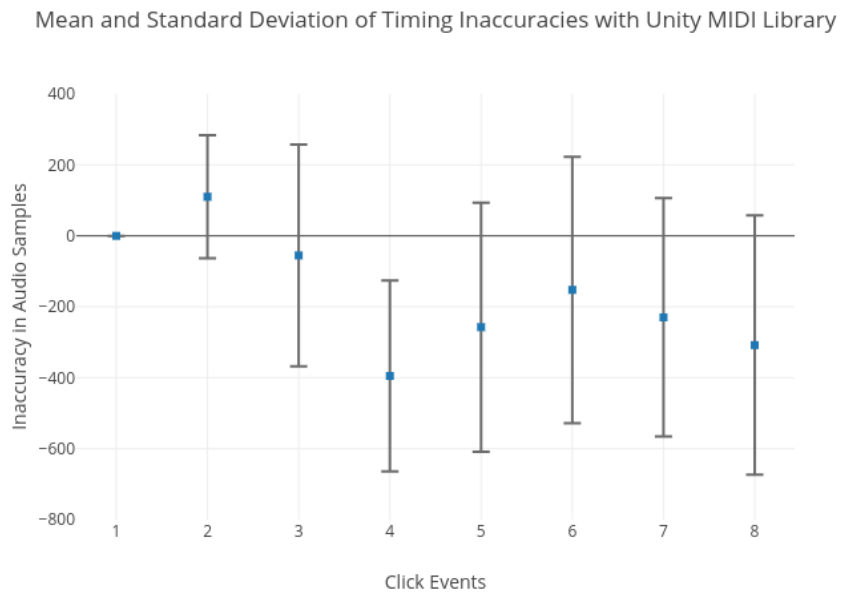

Figure 4.4: Mean and standard deviation of audio timing inaccuracies of the Unity MIDI library when tested with a C major scale over five iterations. The vertical axis shows the degree to which audio signal timing differs from accurate timing in samples

accurate results and is responsible for the time inaccuracy present. While the location of the time inaccuracy can be pinpointed, fixing it requires work that is not within the time scope of this thesis's development. However, an outline of potential solutions is outlined below. In order to fix the problem, a custom C timer could be created that uses `timestamp = TIME_PROC(TIME_INFO)` as its start time, but with all timestamps beyond system startup determined by a custom high accuracy timer. This would remove reliance on the timing function that is the source of the timing problems experienced. Secondly, a custom updating thread in C running the MIDI system would help to improve performance and timing accuracy. Such a thread would reduce the danger of code blocking and data races occurring on the Unity thread, which is likely to be contributing to the time inaccuracy experienced. An alternative possibility would be to use RTMidi instead of Portmidi. RTMidi and Portmidi fulfill a similar role as multi-platform MIDI I/O libraries, but RTMidi is written in C++ rather than in C, and may have a timing system that simplifies high accuracy scheduling of MIDI events.

While the degree of time inaccuracy present in our tool means that it does not meet ideal technical specifications, the tool achieves all other design criteria. The tool is therefore a partial success, but requires further iteration if it is to be useful in

professional game audio applications which utilise percussive signals. Work to improve the tool's time-accuracy is underway and will be explored in the future works section of this thesis.

4.2 Tool 2: Unity VST System

The second tool created is a VST2 plugin effect host for Unity Engine. VST2 is the most popular audio plugin format in DAWs, yet there is no support for its use in any of the major game audio development environments. This is due to licensing concerns that make it difficult to embed commercial VST plugin code into video games. The audio plugin host presented here, and all plugins used to test the tool, are open source and freely available, thus avoiding commercialisation issues that have led to the lack of support for VST2 plugins in game engines. If successful, the VST2 plugin host presented will make hundreds of audio effect plugins available for use in Unity and will vastly expand the procedural audio effect potential of the engine. We discuss the development of the Unity VST2 plugin host over four sections: section 4.2.1 presents an overview of the tool and outlines its technical features and components. Section 4.2.2 explores audio programming in C and C++ and looks at ways that we apply audio programming design patterns in the development of our tool. Section 4.3.3 explores extended inter-application communication techniques utilised in our plugin host, with a particular focus on optimising the tool's performance, a major hurdle in the development of the plugin host. Finally, section 4.3.4 evaluates the tool's success in relation to the design criteria defined in section 3.1.

The VST2 plugin tool created supports the loading of VST2 effect plugins, but does not support VST1 or VST3 plugins. VST1 is a deprecated plugin format that is not supported by modern DAWs and is therefore not a concern in our development. On the other hand, while VST3 succeeds VST2 as a format, the quantity of plugins that utilise the VST2 standard far surpasses those using the VST3 format, and we therefore choose to support VST2 plugins with a future goal of expanding the tool to support other plugin formats. The tool is also currently only runable on Windows computers. While multi-platform support would be a powerful feature of our tool, it is not within the scope of this thesis. This is particularly the case in the creation of an OSX compatible dynamic library which would require writing platform specific C++ on a Macintosh computer via Xcode, a programming environment that we

are currently unfamiliar with. The tool presented also only supports audio effect plugins, but not audio synthesis plugins, as support for synthesis proved to be out of scope of our project.

In addition to achieving the evaluative criteria from chapters 1 and 3 of this thesis, in order to be successful, the plugin host developed must be able to load VST2 effect plugins in a DAW-like fashion. DAWs represent the main use-case for VST2 plugins and we use the popular DAW Reaper as a benchmark against which to test our tool later in this chapter.

4.2.1 Overview of Tool

In order to understand the development process of our Unity tool, it is important to understand the function of a VST2 effect and that of an effect host. VST2 plugin effects take audio signals as an input, apply signal processing effects, and then output processed audio. In order to support this process in an executable application, plugin hosting code is required to load plugins into memory and to route audio between plugins and the application. Alongside audio signals, VST2 effect plugins also utilise changeable parameter data which is used to control signal processing algorithms. A VST2 plugin host is responsible for querying loaded plugins in order to understand these parameters and exposes them for use by the executing application.

The tool developed here allows users to load VST2 plugin effects into Unity Engine and to route audio from the engine through one or more audio effect plugins and back to Unity. The process of loading and routing audio through VST2 plugins cannot be accomplished with Unity's C# scripting language alone, and requires the use of a C++ VST2 library called the VST2 SDK. The VST2 SDK, released by Steinberg, the creator of the VST plugin format, includes a number of functions useful when working with VST2 effect plugins. In order to access these functions, we develop a dynamic library for Unity using C++ that wraps key VST2 SDK functions and makes them accessible to Unity's C#.

The C++ Unity dynamic library created loads and unloads VST2 plugins from memory, handles audio input and output to the plugins, and is responsible for communicating parameter changes to the loaded plugins. The library also includes a number of exported dll functions that are used to communicate directly with Unity's C#. The development of the C++ section of the VST2 plugin host requires

a significant amount of work and this process is documented across two sections of this chapter. Section 4.2.2 explores the development of audio software in C and C++ and its application in our plugin host. Section 4.2.3 explores the process of interfacing audio and parameter data between C++ and C# .

The C# section of the plugin host tool, visualised as the three grey boxes in figure 4.6, is responsible for all interfacing with the tool's users and includes a GUI that can be viewed in the Unity Engine editor. The figure also shows the separation between C++ and C# classes and broadly outlines dataflow in the tool.

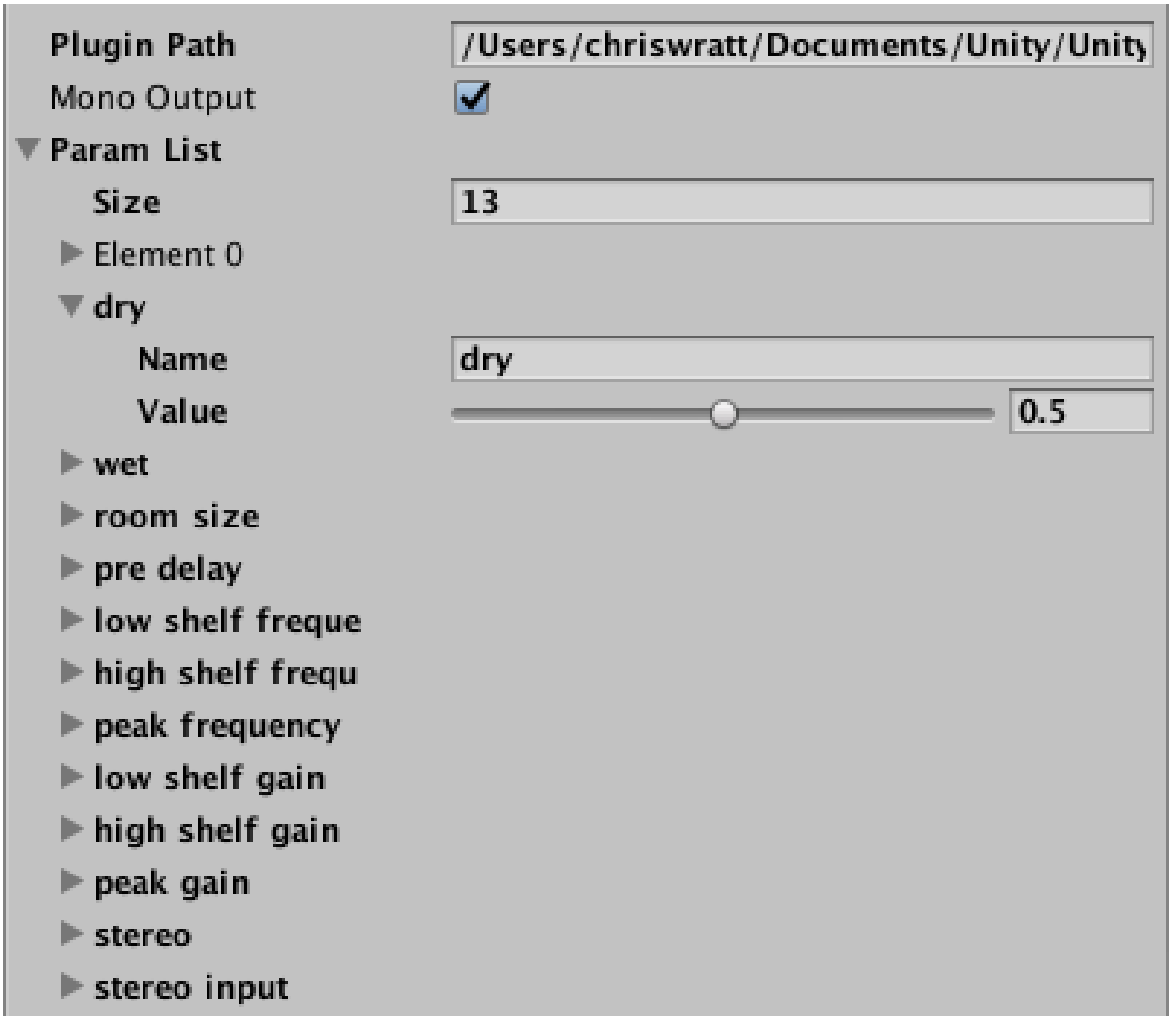

Figure 4.5: VST effect GUI in Unity showing plugin parameter and path settings for a loaded reverb plugin

This new Unity C# GUI controls the loading of VST2 effect plugins and the setting of parameters and can be seen in figure 4.5. The GUI allows users to specify the path of a plugin file to be loaded and presents a list of parameters which allow users to sculpt the tone of the plugin's output. The GUI is generated

by the C# class `VSTEffect`, and is loaded as a component onto a Unity object containing an audio source component, thus allowing the effecting of audio files played in engine. Through utilising the engine's built-in GUI development library, the interface presented uses a highly standardised design style and should be familiar to a regular Unity user. We utilise the engine's audio callback function `OnAudioFilterRead(...)`, as introduced in section 3.2.3, in order to effect audio produced in Unity. `OnAudioFilterRead(...)` alongside being a function applicable in the synthesis of audio signals, can also be used to effect existing audio in the engine such as that created by an audio source component.

In order to achieve audio effect processing via `OnAudioFilterRead(...)` audio and parameter data are sent from Unity to the C++ plugin host and then on to the VST2 plugin. Audio is then routed back to the Unity callback function from C++ for output by Unity. In order to send large quantities of audio and parameter data between Unity and C++, we utilise inter-programming language techniques that go beyond those explored in our MIDI tool in section 4.1. Section 4.2.3 outlines these extended techniques.

The C++ plugin host developed also includes a class responsible for debugging via Unity's console debugger, which we take directly from the MIDI library explored earlier in this chapter. The use of this debugger greatly expedites the development of our tool, and utilising such a tool is highly recommended to others undertaking C or C++ plugin development for Unity Engine.

4.2.2 Audio Programming in C and C++

C++ is chosen as the main language for the development of the tool for a number of reasons. C++, alongside the C programming language, is a widely used audio programming language and many significant libraries for audio development run in either C++ or C. This list includes the VST2 SDK, Unity's underlying audio system code (FMOD), and the Windows audio output system upon which all of our development is built. C and C++ have become popular for audio programming due to their ability to give users a high degree of control over memory and thread management, and do not automate memory management in the way that languages such as Java and C# do.

The strict control over memory and threading systems is highly useful in audio programming as it allows users to optimally fulfill a number of requirements in

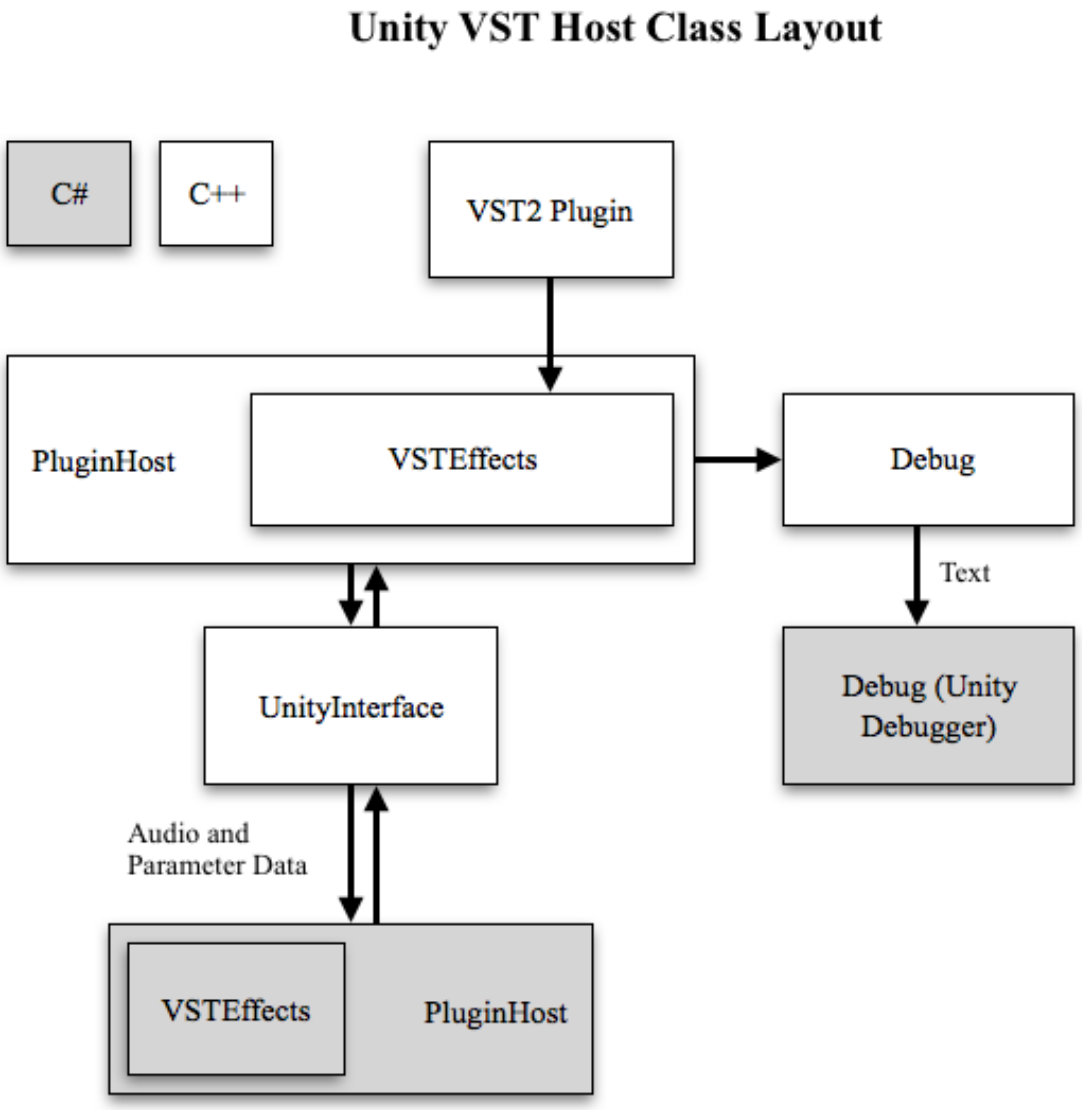

Figure 4.6: Block diagram of Unity VST host showing major C# and C++ classes and inter-language dataflow

audio programming, which are often difficult to achieve in higher level languages. One example is C and C++'s ability to avoid memory allocation in audio hardware callbacks by initialising all memory in other threads without the danger of it being auto-deallocated. Audio hardware callbacks, which are a widely adopted program design paradigm in audio programming, constitute a hard-real-time system and memory allocation cannot be used in such a system. This is because memory allocation uses memory locking techniques such as mutexes, a mechanism for locking and unlocking memory while it is being read and written to, which create blocking code and introduce a high degree of risk of buffer under-run which can in turn cause audible artifacts.

Another example of an audio programming problem that can be avoided through the use of C++, and, with more difficulty, C, is that of slow and un-safe inter-thread communication. When data is sent between threads in C and C++, a traditional way to avoid problems with data being read by one thread while it is being written to by another (thus causing undefined behaviour) is to use mutexes. As explained in the previous paragraph, mutexing in an audio hardware callback in a real-time application is not a safe technique. Instead of mutexing, the C++ standard library supplies functionality for lock-less inter-thread data transfer through the type `std::atomic<type>`. Functions such as `std::atomic.exchange(...)` allow the swapping of simple data types stored by an atomic object within a single cycle of the CPU, thus removing the need for thread locking to prevent read-write overlap. Applications of atomic operations within C are also possible, but are more difficult to achieve as there is no atomic type in the C standard library and external tools are required. In our tool, we use the parameter setter and getter code presented by the VST SDK and therefore avoid writing our own atomic code, but an awareness of its need leads to our avoidance of directly accessing data utilised from other threads within our audio callback.

We choose to use C++ rather than C for the development of this tool because the VST2 SDK is written in C++, and it is easier to interface with it from C++ than from C. C++ is also chosen because of its support for the use of dynamic memory allocation classes such as `std::vector`. These classes make the storage of large numbers of VST plugins within our host easy to manage, whereas such a process would be more difficult to achieve in C where the labour-intensive development of a custom dynamic memory system would likely be required.

In summary, our choice to use C++ for the bulk of the coding in our application is due to C++'s wide use as a standard programming language for audio program-

ming and due to its suitability for real-time system design where high performance is required. C++ features such as user-managed memory, use of atomic types to avoid mutexes, and avoidance of memory allocation at run-time make it a highly suitable environment for audio programming.

4.2.3 VST-Host Architecture

The hosting of VST2 plugins on Windows is a sparsely documented process and much of our development relied upon reading and interpreting the VST2 SDK source code. Due to this scarcity of documentation, we outline the core requirements of a VST2 host below, and explore how we meet these requirements in our tool. The key attributes of a plugin host are its ability to load VST files into memory, its ability to handle operation codes sent to the VST2 plugin, and its ability to deal with audio and MIDI interfacing with the plugin.

In order to dynamically load a plugin, the C++ `HMODULE LoadLibrary(wstring)` function is used to create a handle to load the plugin into memory. Once this is achieved, `GetProcAddress(HMODULE, string)` is used to find the memory address of the plugin's main function so that the dll file can be run. Once the plugin is loaded into memory (as an `AEffect` object), and the dll's `main()` function has been called, operation codes are sent and received from the loaded plugin via the `AEffect->dispatcher(...)` call, a heavily overloaded function that can take a range of parameters. One parameter type that is shared between all overloaded versions of the dispatcher function is a string variable input for the operation code identifier, which corresponds to an enumerated list of operation commands and can be found in the source code for the `AEffect` class in the VST2 SDK. Important examples of operation codes used in our plugin host include `effGetParamName`, which retrieves parameter names from the plugin, and `effSetSampleRate`, which sets the plugin's sample rate.

Audio processing via the VST2 SDK utilises an audio callback function to be called by an external process. We trigger the plugin audio callback events from Unity and include the audio DSP call in `OnAudioFilterRead(...)` functions which are synced to the Unity master audio callback. When `OnAudioFilterRead()` is used to call VST audio callbacks, audio data is routed from Unity via an exported function called `processBuffer(...)`. The `processBuffer(...)` function in turn calls an audio processing function in our `VSTEffect` objects that wraps the VST2 SDK audio processing functions. In order for a VST2 host to process audio in VST2 plugins

the VST2 SDK function `AEffect->processReplacing(...)` is utilised which calls the core audio signal processing function of the VST2 in use. The `processReplacing(...)` function requires a pointer to a 2D audio buffer for input and one for output. This I/O structure is different to that used by Unity's `OnAudioFilterRead(...)` and requires a de-interleaver and interleaver for stereo signals within our audio callback function in C++.

Alongside processing audio, VST2 plugins are often used to process MIDI data. Like VST2 parameter processing, VST2 MIDI processing utilises operation codes for its core functions via the plugin's dispatcher. The dispatcher function for scheduling events takes an input of an array of MIDI events that must be stored using the `VSTEvents type`, an array of `VSTEvent` objects. The code in listing 4.4 demonstrates the creation of a note on message, which is packaged into a `VSTEvents` object. The listing also demonstrates the use of `VSTMidiEvent.deltaFrames`, which allows the offsetting of MIDI data within the DSP callback in which the events are processed, thus supporting sample accurate MIDI messages. While we can achieve sample accurate MIDI play back using the techniques explored above, our final released plugin does not include VST2 instrument file or MIDI support. This is due to memory access issues that currently occur when we load VST2 plugins that utilise MIDI input, and a crash-free implementation of VST2 MIDI input has proved to be outside of the time constraints of this thesis.

```
1  VstEvents    midiOutEvents;
2  VstMidiEvent midiEvent;
3  memset(&midiOutEvents, 0, sizeof(midiEvent));
4
5  midiEvent.type = kVstMidiType;
6  midiEvent.midiData[0] = 0x91; //Note on, channel 1
7  midiEvent.midiData[1] = 60; //Middle C
8  midiEvent.midiData[2] = 80; //Amplitude 80
9  //send to VstEvents struct
10 midiOutEvents.events[0] = (VstEvent*)&midiEvent;
11 midiEvent.deltaFrames = 0; //Time offset in samples
```

Listing 4.4: VSTEvent code that generates a MIDI note on message and stores it in a VSTEvents object

4.2.4 Extended Inter-Programming Language Communication

In order to support the high volume of audio data being sent between C++ and C# in audio callback functions, we have invested significant effort into the optimisation of inter-application array and pointer transfer. Initially we utilised the C# interopServices function `Marshal.Copy(...)`, which converts a raw pointer (`intPtr` in C#) into an array of floating point numbers, and allows float arrays created in C++ to be 'unpacked' in a format that can be read by C#. The same programming structure was used to send audio data from C# to C++ in section 4.1 and the complete C# code for sending audio via this method can be seen in listing 4.5. In the listing, lines six and seven call the C++ audio processing callback function, and the surrounding code prepares the data for inter-language transfer.

While the use of the marshalling technique seen in listing 4.5 was successful in transferring audio data between C# and C++, it used large amounts of system CPU resources. When tested with a basic 2 channel delay plugin it required around 35% of the CPU allotted to audio by Unity (on a 2011 Macbook Pro with an Intel Core i5 processor). This is a high CPU usage, especially in game development where intensive visual processing alongside audio processing is common. The use of the `Marshal.Copy(...)` functions in lines four and nine of listing 4.5 required significant CPU usage because all audio data sent through is copied into newly allocated memory, thus causing a large volume of memory allocations and de-allocations for every audio buffer processed. This use of memory allocation on the audio thread caused mutexes to occur, a technique that is avoided in audio programming as discussed in the previous section of this chapter.

Due to the unsuitability of `Marshal.Copy(...)` for the sending of large volumes of real-time audio data between applications, we looked for alternative solutions. A major problem with sending data between C# and C++ is that the C# garbage collector could call while data is being read by C++, thus causing the program to crash. One solution is the copying of data from the garbage collected C# into an intermediary un-managed data type as achieved by `Marshal.Copy(...)`. Another solution is to 'lock' data in a way that stops the garbage collector from de-allocating it. C# supports this behaviour through the interopServices class `GCHandle`, which allows the creation of handles to the garbage collector and the pinning of data, thus moving it from managed to un-managed code. We utilised the `GCHandle` to

send audio data between C# and C++ with GC pinning in line three of listing 4.6. The code presented pins data before sending it to C++ via the `processBuffer(...)` function in line four. GCHandle pinned data is cast to a raw pointer for transfer and then cast back to a GCHandle in line seven. While the use of garbage collector locking appeared to be a solution to our CPU problem, it was quickly determined that it required memory allocation and de-allocation that once again contributed to heavy CPU usage and mutex locking. The code in listing 4.6, when used to run the same stereo delay plugin loaded with the `Marshal.Copy(...)` method examined earlier, once again used 35% of audio CPU, a clearly unacceptable amount of CPU.

```
1  IntPtr inputArrayAsVoidPointer = new IntPtr(Void*);
2  void OnAudioFilterRead(float[] data, int channels)
3  {
4      Marshal.Copy(data, 0, inputArrayAsVoidPointer,
5          pluginHost.blockSize * channels);
6      IntPtr outputVoidPtr = HostdllCpp.processFxAudio(
7          thisVSTIndex, inputArrayAsVoidPointer,
8          pluginHost.blockSize, channels);
9      Marshal.Copy(outputVoidPtr, data, 0,
10         pluginHost.blockSize * channels);
11 }
```

Listing 4.5: The marshalling of audio data with IntPtr's in C# as used in the initial development on our VST2 host tool

```
1  void OnAudioFilterRead(float[] data, int channels)
2  {
3      GCHandle audioHandle = GCHandle.Alloc(data, GCHandleType.Pinned);
4      IntPtr outputVoidPtr = HostdllCpp.processFxAudio(
5          thisVSTIndex, GCHandle.ToIntPtr(audioHandle),
6          pluginHost.blockSize, channels);
7      GCHandle gch = GCHandle.FromIntPtr(outputVoidPtr);
8      data = (float[])gch.Target;
9      audioHandle.Free();
10 }
```

Listing 4.6: Use of garbage collector calls to avoid memory de-allocation while inter-programming language data transfer is occurring in C#

Through informal discussions with audio programmers at GDC in 2018, a third way of sending audio data between C# and C++ was trialled. The new approach

can be seen in listing 4.7 and uses the keywords `[In, Out]` in the external function definition of the float array parameter in lines one and two. The use of these keywords changes the float parameter from being passed by value to being passed by reference, and when such a parameter is passed to C++ it can be read and written to as if it were a reference to an object in C++, thus avoiding the need to cast data to and from the `IntPtr` type via marshaling or copying. While initially we were skeptical as this approach does not lock the garbage collector or copy data into a safe un-managed location, we have observed no audible artifacts or crashes caused by unexprected de-allocations. The technique was also introduced to us by professional game industry audio programmers and we therefore act under the assumption that the C# compiler is aware that data is in use and does not de-allocate it. The use of this technique reduces our CPU usage from the 35% of the Unity audio CPU budget for loading a simple stereo delay plugin to 0.8%. This shows that the `[In, Out]` method used is a highly appropriate C# method for inter-application data transfer of audio data due to its ability to send data between applications without copying of data by value occurring.

```
1 public static extern void processBuffer(int vstIndex,
2     [In, Out] float[] audioThrough, long numFrames,
3     int numChannels);
4
5 void OnAudioFilterRead(float[] data, int channels)
6 {
7     HostdllCpp.processBuffer(thisVSTIndex, data,
8         pluggoHost.blockSize, channels);
9 }
```

Listing 4.7: Callback using `In, Out` keywords to send and receive data in C# from C++

4.2.5 Section Results

In order to understand whether the tool presented meets the evaluative criteria of a successful procedural audio system, testing is undertaken. We test audio output across different plugins loaded in Unity against the same plugins loaded in Reaper. This is done in order to understand whether our tool adheres to the VST2 plugin standard and, by extension, whether our tool meets our technical evaluative criteria of supporting a standard audio protocol. We test our VST host by loading two different plugins into the tool and compare audio output against that achieved by

Reaper by routing audio from Unity to Reaper through the use of audio routing tool Virtual Audio Cable [63]. We utilise a five second stereo audio file consisting of clicks at 120 BPM which pans linearly from left to right over the five second period. The click file is used to simplify analysis of panning and sample-level differences in signals as the click samples are easy to observe on visual debugging tools and allows us to quickly diagnose any errors that may occur. The loading of plugins in our tests occurs in a 'built' .exe Unity application, but informal tests show that identical results can be achieved in Unity's edit mode.

Our first test utilises a custom-built delay plugin created in Juce, which we developed with this test in mind. The plugin is developed so that no randomisation of internal parameters occurs, therefore allowing a sample-level analysis of audio data. The delay plugin uses stereo input with a 50% mix parameter and 50% feedback parameter. Figure 4.7 demonstrates processing clicks that are played at 120 BPM and pan from the left channel to the right channel linearly over five seconds as processed by the delay plugin in Unity, and as processed by Reaper. Even from a low resolution visual representation of the signals, we can see that the signals are not identical: signals created in Reaper are always positive and never go below amplitude zero, yet Unity signals regularly drop below amplitude zero. In order to understand this disparity, a sample level visual representation of the waves is shown in figure 4.8. In this figure we can see that the Unity output smears transients that were initially one sample by up to four samples.

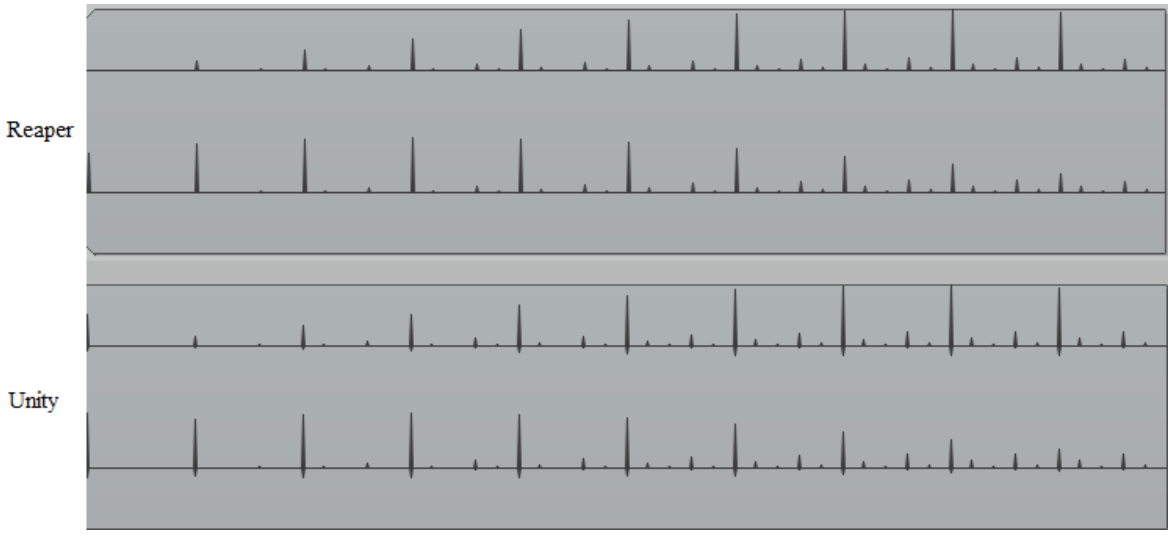

Figure 4.7: Audio output from a delay plugin loaded in Reaper (top) and loaded in Unity (bottom). Note that signals are identical except for transient blurring in the Unity wave file

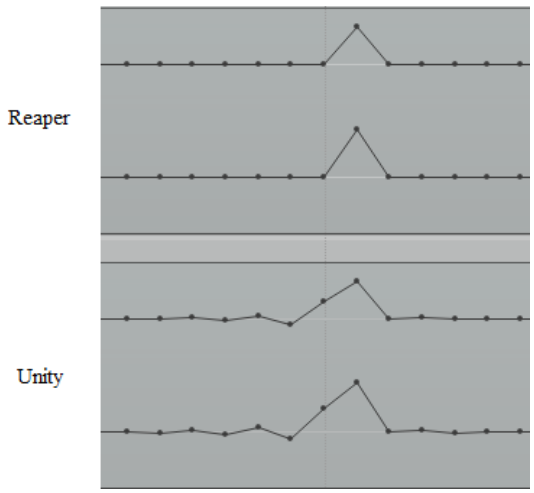

Figure 4.8: Audio output at a sample level from a delay plugin loaded in Reaper (top) and loaded in Unity (bottom). The Unity signal shows smearing around the impulse event

There are three potential sources of the signal smearing: our Unity VST host, Unity's audio output system, or the inter-application signal routing software Virtual Audio Cable. If the signal smearing is occurring in Unity's output or in the inter-application routing, then it is out of our control, however if it is occurring in our software, then it must be further explored and may signal the existence of an error in our code. In order to understand the source of the signal smearing, we analyse an un-effected version of the audio file used in the previous test that is routed from Unity back to Reaper and we compare it to the original signal in Reaper. Figures 4.9 and 4.10 show the result: smearing of audio signals is occurring without our VST host included in the signal chain. It is also important to note that the signal smearing seen is not audible in any of the tests undertaken. With these results we suspect that there may be a re-sampling step occurring in Unity or in Virtual Audio Cable which would account for the presence of signal smearing as shown in figures 4.8 and 4.10. Because there is no ability to access the low level code of either Unity's audio output or Virtual Audio Cable, the source of the signal smearing cannot be ascertained beyond the fact that it is not our tool causing it. It is therefore concluded that the plugin host tool is accurate to a sample level and meets our required technical criteria for time-accuracy.

It is also important to note that signal smearing will have been present in our tests throughout Chapter 3 when testing Unity's audio output. While this is the case, the gate utilised in our ChucK program effectively avoided issues caused by the signal smearing.

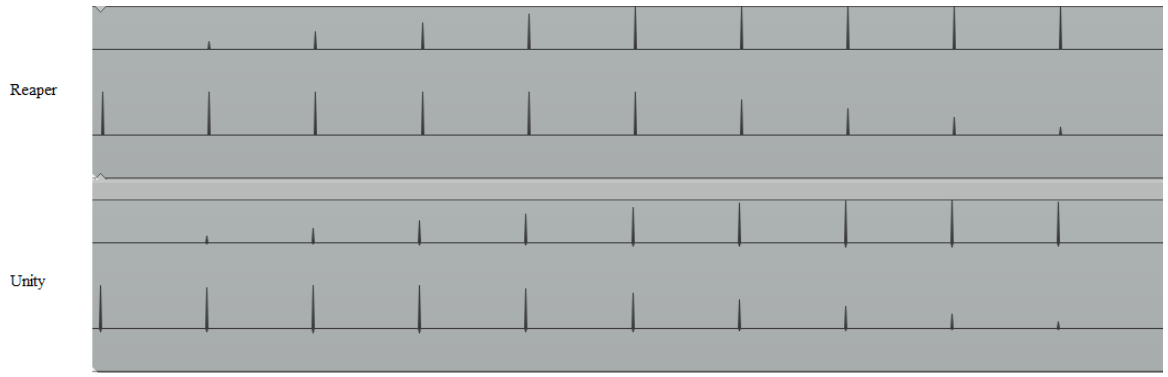

Figure 4.9: Un-effected audio output from Reaper and Unity playing a panning five second file of clicks. We can see that signal smearing is still present in the Unity waveform

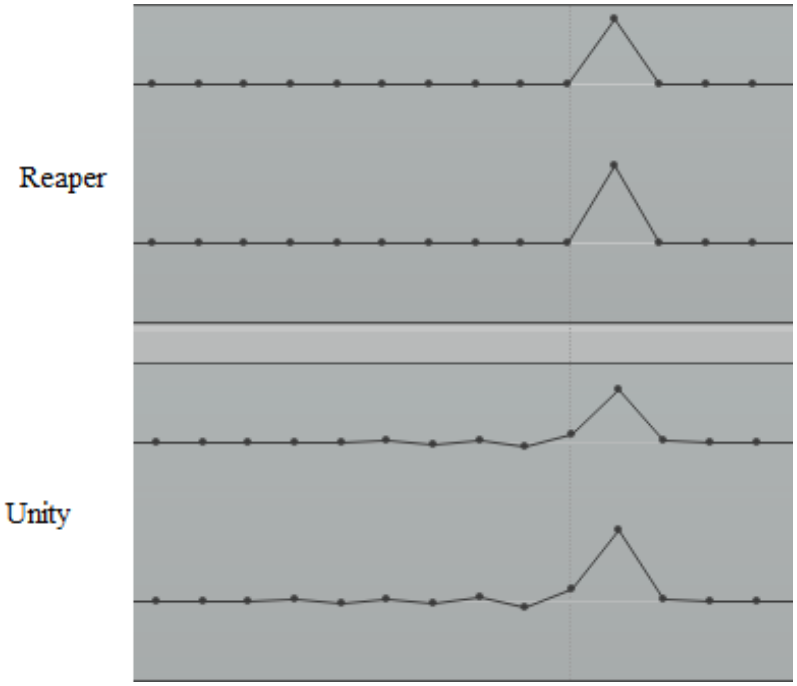

Figure 4.10: Un-effected audio output from Reaper and Unity at a sample level on the sixth impulse in figure 4.9 showing signal smearing similar to that seen in figure 4.8

The second plugin test uses a reverb plugin called TAL-Reverb-2. We use TAL-Reverb-2 because it is free and open source and is therefore a plugin that can be used by a game developer in an application of our plugin host. By analysing TAL-Reverb-2's source code, we can see that it uses five comb filter delay lines and six longer all-pass delay lines. Each delay line is lowpass and highpass filtered and the coefficients of both filters are modulated in real-time. The length of each delay line is slightly randomised and delayed signals are therefore not identical on each play back. Due to the modulation of filter coefficients and delay line lengths, the plugin does not produce identical results on subsequent uses. This means that sample level analysis, such as that used in the delay plugin test, is not possible. The test presented here is used to check if the plugin is correctly loaded and whether audio signals are audibly similar (taking into account modulation) across Unity and Reaper. We set the plugin parameters to 100% wet reverb signal with no pre-delay so that signals all start at the first impulse, and a short 200 ms reverb tail so that reverb signals do not significantly overlap. Figure 4.11 shows four waveforms (we ran the test twice in each software to give an indication of the level of randomisation present).

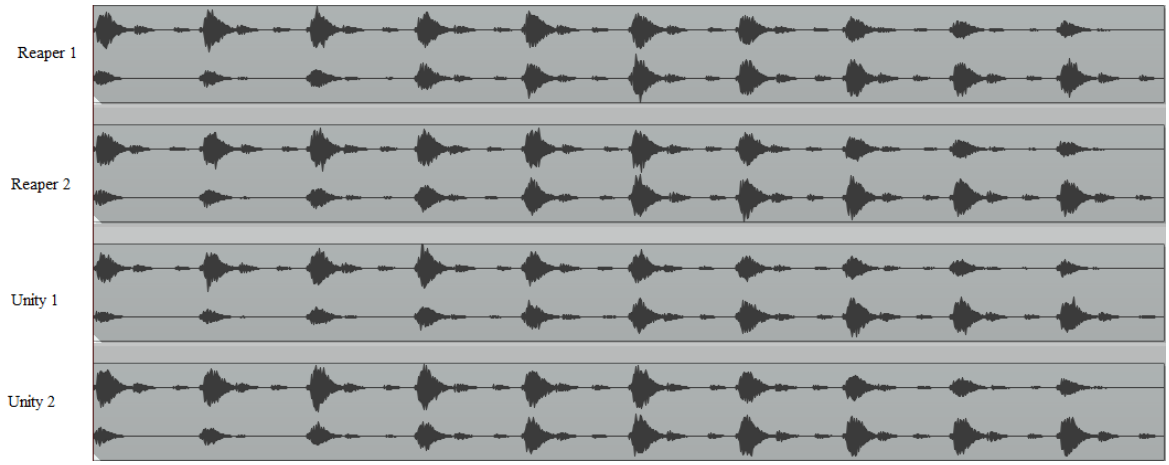

Figure 4.11: Comparison of five second 120 BPM ticks processed through TAL-Reverb-2 in Reaper vrs Unity. Each test is shown twice due to randomisation inside of plugin to give idea of error threshold. The test does not show significant differences in output between applications

The results of loading the reverb plugin into Reaper and Unity show no significant differences beyond those expected due to internal parameter randomisation. The output also presented no audible differences beyond slight tonal changes which can be accounted for by randomised coefficients inside of the plugin. The loading of the reverb plugin proves the ability of our tool to accurately set plugin parameters and

to load a resource heavy plugin.

Between the two tests presented here, we show that our tool can accurately load VST2 effect plugins, thus achieving our first technical evaluative criteria: support for a standardised audio protocol in a game audio setting. We also show that our plugin is sample accurate. The ability to load VST plugins and build them into executable applications is a major step forward in procedural audio tool development for game developers, and is an accomplishment that has not been achieved by other game developers before this time. As the tool presented meets all evaluative criteria for this thesis, it can therefore be considered to be complete success.

4.3 Section Summary

This chapter has presented two new game audio tools that expand Unity Engine's capability to achieve procedural audio.

The MIDI tool presented, through its support for real-time manipulation of MIDI files and MIDI output, allows for procedural organisation of MIDI data, which was not previously supported by any of the four main game audio environments. While it succeeds in this achievement, the tool does not meet ideal thresholds for time accuracy in percussive signals sequencing and can therefore only be considered a partial success.

The VST2 host developed adds new possibilities in the procedural application of audio effects in video games. The tool makes VST2 effect plugins accessible in Unity Engine applications and in doing so vastly expands the potential effect tools available to Unity developers creating games for Windows. The tool is successful in achieving all required criteria and is a significant step forward in procedural game audio technology.

Chapter 5

Conclusion

To conclude this thesis, this chapter summarises the research conducted and evaluates its success. Future work to extend the MIDI and VST tools presented in Chapter 4 is also discussed.

5.1 Summary

In this thesis, we have documented and explored the lack of accessible procedural game audio tools. In Chapter 1, we introduced the gap in the field and presented initial evaluative criteria against which to test tools developed later in the thesis. Chapter 2 outlined related fields of research that informed our development process. We explored the related fields of algorithmic composition, interactive techniques in game audio, and game audio tools development.

Chapter 3 presented experiments undertaken across a range of significant game audio environments. In section 3.1, a set of technical criteria for evaluating game audio tools was developed. The following three sections analysed a wide variety of game audio software and explored ways in which existing game audio solutions achieve or fail to achieve successful procedural audio. Section 3.2 explored the four most popular game audio development environments; Wwise, Fmod Studio, Unreal Engine, and Unity Engine, and evaluated their suitability in the procedural generation of audio. The section concluded that none of the four environments have support for procedural audio development, yet Unity Engine stood out as a potential host for such a system due to its flexibility and extendability as a tool. Section 3.3 looked at the utilisation of procedural audio in retro game audio

technology. We chose to undertake this analysis in order to explore ways in which historical applications of procedural audio in video games could be applied in modern game audio environments. In section 3.4, we applied lessons learned throughout the preceding chapter to the development of a procedural synthesiser and sequencer created in Pd. While our Pd tool achieved a number of the evaluative criteria introduced in sections 1.1 and 3.1 of the thesis, limitations in Pd's data storage and video game-applicability led us to look to other solutions for procedural game audio tools development.

In Chapter 4, we developed two custom tools for procedural audio development in Unity Engine. The tools developed bypassed many of the software constrictions uncovered in Chapter 3, yet in order to do so, much of our development required the use of rarely-used programming libraries and techniques that are severely under-documented.

Section 4.1 explored the development of a MIDI library for MIDI file playback, procedural organisation of MIDI data, and for MIDI output on Windows computers. While the system developed successfully achieved procedural audio and utilised a GUI and simple programming API, it failed to achieve an optimal level of time accuracy. It is therefore inappropriate if highly percussive signal sequencing is required. We uncovered the source of this time inaccuracy, but were unable to solve it. Nevertheless, the tool was a partial success and the timing error appears to be solvable, albeit with development time that is outside of the scope of this thesis.

Section 4.2 documented the development of a VST2 effect plugin host for Unity Engine. The tool developed enables the use of a wide variety of pre-existing tools for the procedural application of audio effects in Unity Engine via a custom-developed GUI. The Unity VST host tool successfully meets all of our design criteria and is a significant contribution to the development of a successful modular procedural game audio workflow in Unity engine.

When viewed together, our developed Unity MIDI library and Unity VST Host achieve all technical and general evaluative criteria of this thesis, other than the MIDI library's documented audio timing issue. The tools signify an important advancement toward achieving accessible procedural audio in modern video games and an exploration of their current impact in the game audio community will be explored in section 5.3.

5.2 Future Works

In order to further support procedural audio development in video games, we are continuing to develop the game audio tools presented in Chapter 4 of this thesis. Firstly, our most pressing future goal is to reduce the MIDI systems' time inaccuracies to be under the human threshold for audible time inaccuracies. An outline of the work required to achieve this is included in the final remarks of section 4.1, which calls for the development of a more robust timing system that interfaces with Unity via inter-thread communication.

Secondly, we would like to expand the MIDI tool developed so that it can be used across a wider variety of platforms. Currently the tool only supports Windows development, but much of the Portmidi and Unity source used is written for use across multiple platforms. Achieving Macintosh, mobile, and Linux builds of the tool would significantly widen its potential user-base.

The VST2 plugin hosting tool presented in section 4.2 achieved the majority of our technical requirements, yet there are a number of ways in which it can be improved upon after the completion of this thesis. The ability to load the graphical user interfaces of loaded plugins, rather than interfacing directly with them through parameters via our custom GUI, would make the tool more accessible to audio developers with DAW skills. Also, support for VST3 and Audio Unit plugin formats would greatly expand the possible plugins that can be utilised by our plugin host. Finally, supporting the loading of VST2 instrument plugins and enabling MIDI input to our plugins would be an important achievement for our tool.

If a MIDI-enabled version of our plugin host were to be paired with our MIDI library (once time inaccuracy issues are fixed), this would constitute a massive achievement in procedural game audio and would easily make the Unity Engine the most powerful modern game engine for use in the procedural development of audio. In the meantime, the tools developed in this thesis are significant advancements in procedural game audio and are an important step toward achieving this larger goal.

5.3 Final Remarks

This thesis has outlined gaps in the field of procedural game audio development and has undertaken work to fill these gaps. While our development does not achieve a fully realised procedural game audio solution, our tools have moved the field forward in a number of significant ways. Our achievement of VST2 plugin effect loading in Unity has received recognition by the game audio community, and a tweet documenting its release was publicly liked and shared on twitter by a number of important figures in game audio including a senior sound designer at Ubisoft and Damian Kasbour, who is perhaps the leading game audio implementer in the world [64]. Chris Wratt, the author of this thesis, has been offered a paid position to continue the research by two internationally recognised game companies: Strange Band Audio in Sydney and Indie Darling in Los Angeles. They have also been offered a position by the interactive art space Meow Wolf in Santa Fe, New Mexico, to create a procedural audio system based on developments made in this thesis. The VST2 plugin host developed has picked up followers on GitHub and is actively being used by Maxime Barruet, a French audiologist working for Maitre Audio (a major audiology company in France). Barruet is using the tool to load plugins that correct speaker frequency response for audiology testing of children's hearing.

The attention shown to our work by the international game audio community reiterates the importance of our tools and affirms that our research is an important step forward in modern game audio. We are delighted to see our tools being used by audio developers to support the creation of immersive experiences in their games, and we look forward to continuing to work with the international game audio community in the future.

References

[1] A. Farnell, *Designing Sound*. MIT Press, 2010.

[2] R. Vreeland, "Personal correspondence concerning lack of game audio tools available to independent game developers." 2017.

[3] Unity Technologies, "Unity Game Engine." [Website]. https://unity3d.com/ [Accessed: 2018-06-11].

[4] Nintendo Entertainment, "Super Mario Bros." [Video Game (NES)], 1985.

[5] Nintendo Entertainment, "Donkey Kong." [Video Game (NES)], 1983.

[6] M. Sweet, *Writing Interactive Music for Video Games*. Addison-Wesley, 2015.

[7] Lucas Arts Games, "Monkey Island 2." [Video Game (Multi-Platform)], 1991.

[8] thatgamecompany, "Journey." [Video Game (Playstation 3)], 2012.

[9] Obsidian Games, "Fallout: New Vegas." [Video Game (Multi-Platform)], 2010.

[10] Lucas Film Games, "Ball Blazer." [Video Game (Atari 8-bit Family)], 1984.

[11] NanaOn-Sha Games, "PaRappa the Rapper." [Video Game (Play Station 1)], 1996.

[12] United Game Artists, "Rez." [Video Game (PS2)], 2001.

[13] Maxis Games, "Spore." [Video Game (Multi-Platform)], 2008.

[14] Rockstar Games, "Grand Theft Auto Five." [Video Game (Multi-Platform)], 2013.

[15] S. Tomczak, "LittleScale: Chiptune Website." [Website]. http://chiptech.milkcrate.com.au/ [Accessed: 2018-05-10].

[16] P. Cook, *Real Sound Synthesis for Interactive Applications*. CRC Publishing, 2002.

[17] Phosfiend Systems, "Fract Osc." [Video Game (Windows and OSX)], 2014.

[18] Infinity Ward, "Call of Duty: Modern Warfare 2."
[Video Game (Multi-Platform)], 2009 .

[19] Valve Corporation, "Half Life Two." [Video Game (Multi-Platform)], 2004.

[20] Abstraction Games, "140." [Video Game (Windows and OSX)], 2013.

[21] D. Kastbauer and A. Woldhek, "Game Audio Podcast." [Podcast].
http://www.gameaudiopodcast.com/ [Accessed 12/5/2018].

[22] PopCap Games, "Peggle Blast 2." [Video Game (Mobile)], 2014.

[23] A. Alpern, *Techniques for Algorithmic Composition of Music*. Hampshire College, 1995.

[24] J. Maurer, "The History of Algorithmic Composition." [Web Article], 1999.
https://ccrma.stanford.edu/ blackrse/algorithm.html [Accessed 2018-5-4].

[25] I. Xenakis, *Formalized Music: Thought and Mathematics in Composition* . Pendragon Press, 1971.

[26] L. Polansky D. Rosenboom and P. Burk, "Hierarchical Music Specification Language," *Leonard Music Journal*, 1991.

[27] M. Allan and C. Williams, "Harmonising Chorales by Probabilistic Inference. Advances in Neural Information Processing Systems," *Advances in Neural Information Processing Systems (NIPS*, 2005.

[28] J. Gillick, "A Clustering Algorithm for Recombinant Jazz Improvisation," Master's thesis, Wesleyan University, 2009.

[29] C. Wratt and K. Moen, "Swim Swim Swim." [Downloadable Video Game (Windows and OSX)], 2016.
https://chriswratt.itch.io/swimswimswim [Accessed 12/5/2018].

[30] M. Lothe, "Knowledge Based Automatic Composition and Variation of Melodies for Minuets in Early Classical Style," *Annual Conference on Artificial Intelligence*, 1999.

[31] G. Papadopoulos and G.Wiggins, "AI methods for algorithmic composition: A survey, a critical view and future prospects," *In Proceedings of the Symposium on Musical Creativity*, 1999.

[32] A. McLeran, "Personal correspondence concerning algorithmic composition techniques in Spore." May 2018.

[33] C. Garcia, "Algorithmic Music- David Cope and EMI." [Web Article], 2015. http://www.computerhistory.org/atchm/algorithmic-music-david-cope-and-emi/ [Accessed 12/5/2018].

[34] Director: K. Collins, "Beep." [Online Documentary], 2016. http://gamessound.com/ [Accessed: 2018-05-10].

[35] id Software, "Doom." [Video Game (DOS)], 1993.

[36] H. Lowood, "Game Engines and Game History." [Online Article], 2014. https://www.kinephanos.ca/Revue_files/2014-Lowood.pdf [Accessed 2/5/2018].

[37] 2K, "Bioshock." [Video Game (Multi-Platform)], 2007.

[38] B. Nil, "Learning Audio Middleware Online: Where to Start?." [Web Article], 2015. http://designingsound.org/2015/01/26/learning-audio-middleware-online-where-to-start/ [Accessed 12/5/2018].

[39] S.Horowitz and S Looney, "Masterclass: Using Game Audio Middleware." [Web Article], 2014. https://www.emusician.com/how-to/masterclass-using-game-audio-middleware [Accessed 12/5/2018].

[40] A. McLeran, "The Future of Audio in Unreal Engine." [Game Developers Conference (GDC) Presentation], 2017.

[41] P. Leonard, "Using Pure Data as a Game Audio Engine," *Audio Engineering Society Journal*, 2015.

[42] Enzian Audio, "Enzian Audio's Heavy." [Website]. https://enzienaudio.com/ [Accessed: 2018-05-11].

[43] H. Haas, *Uber den Einfluss eines Einfachechos auf die Horsamkeit von Sprache*. PhD thesis, University of Gottingen, 1949.

[44] A. Hudek, "Soundflower." [Github Repository]. https://github.com/akhudek/Soundflower [Accessed 12/6/2018].

[45] G. Wang, *The ChucK Audio Programming Language: A Strongly-timed and On-the-fly Environ/mentality*. PhD thesis, Princeton University, 2008.

[46] Audio Kinetic Staff, "Personal correspondence concerning MIDI in Wwise with Audio Kinetic programmers at GDC 2018." 2018.

[47] Keijiro, "MidiJack." [GitHub Repository]. https://github.com/keijiro/MidiJack [Accessed 2/5/2018].

[48] Tazman Audio, "Fabric." [Downloadable Software]. http://www.tazman-audio.co.uk/ [Accessed 2/5/2018].

[49] J. Garcia, "UnityOSC." [GitHub Repository]. https://github.com/jorgegarcia/UnityOSC [Accessed 2/5/2018].

[50] A. McLeran, "Personal correspondence concerning Blueprints audio development." March 2017.

[51] A. McLeran, "The Future of Audio in Unreal Engine." [Youtube Video]. https://www.youtube.com/watch?v=ErejaBCicds [Accessed 7/6/2018].

[52] D. Reynolds, "Building a Music System in Blueprints." [Youtube Video]. https://www.youtube.com/watch?v=yce2t85MJD8feature=youtu.be [Accessed 2/6/2018].

[53] The Processing Foundation, "Processing." [Website]. https://processing.org/ [Accessed 12/6/2018].

[54] Yatcht Club Games, "Shovel Knight." [Video Game (Multi-Platform)], 2014.

[55] Heart Machine, "Hyper Light Drifter." [Video Game (Multi-Platform)], 2016.

[56] E. Lyon, "A Sample Accurate Triggering System for Pd and Max/MSP." [Website Article], 2006. http://disis.music.vt.edu/eric/LyonPapers/SampleAccurate-Lyon-ICMC2006.pdf [Accessed: 2018-05-10].

[57] C. Wratt, "Unity VST Host." [Github Repository], 2018. https://github.com/Chris-TopherW/UnityVSTHost [Accessed 12/5/2018].

[58] C. Wratt, "Unity MIDI Library." [Github Repository], 2018. https://github.com/Chris-TopherW/UnityMidiPlayer [Accessed 12/5/2018].

[59] Haochuan, Pstieber and RDB, "Portmidi." [Website]. https://sourceforge.net/p/portmedia/wiki/portmidi/ [Accessed: 2018-07-10].

[60] obiwanjacobi, "midi.net." [Github Repository].
https://github.com/obiwanjacobi/midi.net [Accessed: 2018-06-11].

[61] Leslie Sanford, "C# MIDI Toolkit." [Downloadable C# Library].
https://www.codeproject.com/Articles/6228/C-MIDI-Toolkit
[Accessed: 2018-06-11].

[62] Tobias Erichsen, "Loop MIDI." [Website].
https://www.tobias-erichsen.de/software/loopmidi.html
[Accessed: 2018-06-12].

[63] E. Muzychenko, "Virtual Audio Cable." [Website].
http://software.muzychenko.net/eng/vac.htm
[Accessed: 2018-05-10].

[64] C. Wratt, "Tweet About Game Audio Tools." [Tweet].
https://twitter.com/ASCIIVacation/status/990812742192840704
[Accessed: 2018-05-10].