PROCEDURAL AUDIO GENERATION AND ITS PRACTICALITY IN GAMES

GDEV60001 GAMES DEVELOPMENT PROJECT

Daniel Neumann M015290M

Supervisor: Dr David White

Second Assessor: Peter Cooper

Contents

Glossary

ADSR – Attack Decay Release Sustain

CPU – Central Processing Unit

CD-ROM – Compact Disc Read Only Memory

dB – Decibels

DAW – Digital Audio Workstation

DFT – Discrete Fourier Transform

FFT – Fast Fourier Transform

GB – Gigabytes

Hz – Hertz

KB – Kilobytes

MIDI – Musical Instrument Digital Interface

MIMD – Multiple Instruction Multiple Data

ms - Milliseconds

RAM – Random Access Memory

RTTTL – Ring Tone Text Transfer Language

s - Seconds

State – Representation of all notes played at a given point

SIMD – Single Instruction Multiple Data

V – Volts

3D – Three Dimensional

List of Figures

Abstract

Audio and music are common themes in everyday life. From the sounds of the world itself to accompaniment art forms such as movies, video games, and songs, audio is everywhere. It provides consumers the ability to experience the world around them in unique and interesting ways and can evoke feelings and emotions based on the audio they hear. Procedural Content Generation is a field of computer science which creates random content algorithmically based on the parameters given to the program. It is widely used across many areas of video game development, from generating entire worlds to a single blade of grass. Many early games used procedural generation to produce their audio (Horowitz and Looney, 2014). Still, this method has fallen out of favour compared to those used in other forms of audio production (Böttcher, 2013). Therefore, any procedurally generated audio must stand up in quality to other methods while being inexpensive to produce. An investigation into using Markov Chains to generate audio procedurally was conducted, evaluating its feasibility using these metrics to determine whether it could be an effective method for aiding the development process or for use alongside computer games. The results show it as an effective method for mimicking existing, simplistic audio and computationally inexpensive compared to other resource-intensive areas, but it fails to improve, adapt, or replicate more advanced audio samples. An insight into the hurdles procedural audio generation must overcome, along with proposed methods for overcoming them, is provided.

Introduction

Game development most often uses pre-designed audio for later use in the game. These audio files can be replicated across several areas of a game and are often cleverly mixed to respond dynamically as the player interacts with the virtual world. The problem with this is that the possible audio a player may hear during their gameplay is fixed. No matter how much shuffling of similar sound effects is performed, they cannot truly disguise the underlying audio placed in the game. Additionally, storing these audio files consumes valuable space in the game's files. With ever-increasing size requirements for other development areas, minimising the memory cost of audio is important. This can be seen in modern games like BG3 (*Baldur's Gate 3,* 2023), requiring 150GB of storage, which is over 15% of the capacity for modern hardware in a single game (*PlayStation 5, 2020)*. Finally, this process of creating audio is costly, whether it means paying employees to create, mix and integrate audio or outsourcing this at an external rate. This money could be spent on other important areas of game development if a more cost-effective method to create high-quality music is developed.

The proposed purpose of procedurally generating audio in game creation is to create non-rigid audio that doesn't need to be pre-designed. A lot of the music in the game SPORE was procedurally generated, as they knew players would spend hours in crucial segments of the game, namely the 'creature creator' screen (Jolly & McLeran, 2008). With a standard approach to audio development, the player would listen to the same looping tune repeated endlessly as they spent hours creating their creature. To combat this, the audio engineers used procedural audio to prevent the player from getting annoyed by the same repeating track.

Procedural audio can also help audio engineers save time and money on a project. For example, an engineer might have to create several unique yet similar sound effects for every action in the game, such as a 'footstep' or a 'gunshot', once again to prevent the player from growing tired of hearing the same sound (Böttcher, 2013). The issue here is that the player may have several hundred actions they can perform, each requiring several unique sound effects. The time and cost of creating all these only grows with the project size and scope. Procedural audio would reduce or eliminate the need for an engineer to spend time creating all these unique sound effects.

Finally, the associated memory cost of storing the files also reduces with procedural audio, as it doesn't need to be stored if it can be generated dynamically at runtime. This, however, comes at the cost of more program runtime resources, which can be considered more valuable than file size, so methods of generation should aim to be inexpensive in this manner.

Aims and Objectives

Sound is a wide and diverse topic with many facets of exploration. A deep dive into a single instrument or piece of music could be a dissertation in itself. Therefore, the aim will be to focus on the technical aspects of sound creation instead, providing a foundation for further comprehensive research into areas such as music theory or instruments to be built upon it.

There are many methods for procedurally generating audio, each with its own unique approach, positives, and negatives. However, before diving into procedural generation of sound, it is necessary to understand how sound itself is created. The goal will therefore be to break down the key components of sound so they can be effectively handled and used to inform analysis of possible procedural generation methods.

The methods will be evaluated to determine their feasibility for use in a real-time game application. Or alternatively, if they can be used to aid the development process of a game.

An artefact using both the fundamentals and a selected method of procedural audio generation will be built, which will help inform the conclusion on whether procedural audio is a lucrative area for future exploration or if it has no practical application in games.

Literature Review

Fundamentals of sound:

To generate audio, it is critical to understand what computer audio is and how a computer turns numerical data into sound. Understanding this provides the building blocks for transforming randomly generated noise into recognisable, distinct sounds, such as a piano piece.

Sounds can be characterised as vibrations in a medium (Towel, 2022). These vibrations or waves travel through the air to reach our ears, where they are interpreted as sound. The nature of how humans interpret sound is outside the scope of this paper, but the attributes that make up a sound wave are not. If the pressure of the incoming wave has a consistent repeating nature over a given time interval, the sound has a periodic waveform; if there is no pattern, it is called noise (Roads, 2023).

These periodic waves have several key attributes that determine how they sound; otherwise, all sound would be awfully monotone. These three key aspects can be broken down into wavelength, amplitude, and phase.

1. Amplitude

The maximum change in air pressure from the average air pressure is also known as the amplitude of the wave and largely determines how loud the sound is (Talbot-Smith 2002). However, "loudness" itself is an abstract concept; what is loud to one person may not be loud to another. Decibels are the unit in which humans measure the volume of sound. They are a ratio representing a logarithmic scale with respect to air pressure (Vasudevan et al., 2023). Despite this being a ratio, the commonly accepted use of decibels perceives 0 dB as near-silence; however, because it is a logarithmic scale, it will never truly be silent, and hence all measurements of decibels from here are in reference to this volume. Decibels, amplitude and the human ear all line up rather differently, however. A doubling in amplitude for a wave doesn't mean a doubling in decibels or a doubling in perceived loudness for a person. Instead, a doubling in amplitude represents an approximately 6 dB increase or a 4 times increase in sound pressure level. Humans perceive a doubling in loudness at around a 10 dB increase. To convert from a change in pressure or amplitude to a change in decibels, the following formula is used.

$$\Delta db = 20 \log\left(\frac{p}{p_0}\right)$$

Substituting in our 10 dB as our desired change in volume, we get that $\frac{p}{p_0} = 10^{\frac{1}{2}}$ or 3.16 to three significant figures. This means that a 3.16-fold increase in pressure results in a perceived doubling in volume to the human ear. A conversation between humans sits at 60 dB or 64 times louder than near silence (The Open University), and the threshold for pain at 120 dB, around 8192 times louder than near silence (Newell, 2017). It is important to remember that, in this case, 130 dB is relative to near silence; when the baseline is a conversation's volume at 60 dB, it only takes a 70 dB increase to reach the pain threshold.

When generating music, therefore, the amplitude of any waves created should follow these measurements to ensure that they do not damage the hearing of anyone listening to the created sounds.

2. Frequency

The second aspect of the sound wave, the wavelength, is a measurement of the distance between two consecutive peaks or troughs of the wave and is closely related to the perceived pitch of the wave. The time taken for a wave to complete one wavelength is also known as the wave's cycle or period (Roads, 2023). Using this period allows the wave's fundamental frequency to be calculated.

$$f_0 = \frac{1}{T}$$

Where f_0 is the fundamental frequency and T is the period of the wave. Frequency is measured in hertz and represents the number of peaks that would pass through a given point over a second of the wave's travel (Talbot-Smith 2002). As sound travels at a constant speed of around ~343 ms^{-1}, the way in which the frequency of the wave increases and decreases is directly related to the period of the wave.

These varying frequencies have been mapped to musical notes as heard today. For example, the lowest musical frequency, c_0, has a frequency of 16.35Hz (Songstuff, 2025). This provides a relationship between frequency and what we hear. It should be noted that the pitch of a given frequency is not something that can be measured since it changes based on a person's perception of it, while frequency is a quantifiable measurement. Continuing the relationship between frequency and notes, waves form harmonics at integer multiples of the base frequency. The second harmonic, or the doubling of the base frequency, is known in music as an octave. Meaning to go from a c_0 to a c_1 on an instrument is as simple as doubling the frequency of the wave, or 16.35Hz to 32.7Hz. Following this pattern, the jump from a c_1 to a c_2 doubles the frequency again, from 32.7Hz to 65.4Hz. The twelve notes in an octave, then, are split into twelve equal logarithmic steps following the formula:

$$f \times 2^{\frac{n}{12}}$$

Where f is the frequency of the start of the octave and n is the number of semitones desired to step through (Wolfe, 2005). For example, going from a c_0 to a d_0, or a two-semitone jump, is 16.35Hz multiplied by $2^{\frac{2}{12}}$, which comes out at 18.35Hz.

In the generation of music, it is very important to know which notes are being played at which frequencies, as this helps abstract away from combinations of frequencies and instead helps attribute these frequencies to combinations of common musical notes, which are well studied.

MIDI note to frequency conversion chart

MIDI Note	Frequency	MIDI Note	Frequency	MIDI Note	Frequency
Octave -1		Octave 0		Octave 1	
C 0	8.1757989156	12	16.3515978313	24	32.7031956626
Db 1	8.6619572180	13	17.3239144361	25	34.6478288721
D 2	9.1770239974	14	18.3540479948	26	36.7080959897
Eb 3	9.7227182413	15	19.4454364826	27	38.8908729653
E 4	10.3008611535	16	20.6017223071	28	41.2034446141
F 5	10.9133822323	17	21.8267644646	29	43.6535289291
Gb 6	11.5623257097	18	23.1246514195	30	46.2493028390
G 7	12.2498573744	19	24.4997147489	31	48.9994294977
Ab 8	12.9782717994	20	25.9565435987	32	51.9130871975
A 9	13.7500000000	21	27.5000000000	33	55.0000000000
Bb 10	14.5676175474	22	29.1352350949	34	58.2704701898
B 11	15.4338531643	23	30.8677063285	35	61.7354126570
Octave 2		Octave 3		Octave 4	
C 36	65.4063913251	48	130.8127826503	60	261.6255653006
Db 37	69.2956577442	49	138.5913154884	61	277.1826309769
D 38	73.4161919794	50	146.8323839587	62	293.6647679174
Eb 39	77.7817459305	51	155.5634918610	63	311.1269837221
E 40	82.4068892282	52	164.8137784564	64	329.6275569129
F 41	87.3070578583	53	174.6141157165	65	349.2282314330
Gb 42	92.4986056779	54	184.9972113558	66	369.9944227116
G 43	97.9988589954	55	195.9977179909	67	391.9954359817
Ab 44	103.8261743950	56	207.6523487900	68	415.3046975799
A 45	110.0000000000	57	220.0000000000	69	440.0000000000
Bb 46	116.5409403795	58	233.0818807590	70	466.1637615181
B 47	123.4708253140	59	246.9416506281	71	493.8833012561
Octave 5		Octave 6		Octave 7	
C 72	523.2511306012	84	1046.5022612024	96	2093.0045224048
Db 73	554.3652619537	85	1108.7305239075	97	2217.4610478150
D 74	587.3295358348	86	1174.6590716696	98	2349.3181433393
Eb 75	622.2539674442	87	1244.5079348883	99	2489.0158697766
E 76	659.2551138257	88	1318.5102276515	100	2637.0204553030
F 77	698.4564628660	89	1396.9129257320	101	2793.8258514640
Gb 78	739.9888454233	90	1479.9776908465	102	2959.9553816931
G 79	783.9908719635	91	1567.9817439270	103	3135.9634878540
Ab 80	830.6093951599	92	1661.2187903198	104	3322.4375806396
A 81	880.0000000000	93	1760.0000000000	105	3520.0000000000
Bb 82	932.3275230362	94	1864.6550460724	106	3729.3100921447
B 83	987.7666025122	95	1975.5332050245	107	3951.0664100490
Octave 8		Octave 9			
C 108	4186.0090448096	120	8372.0180896192		
Db 109	4434.9220956300	121	8869.8441912599		
D 110	4698.6362866785	122	9397.2725733570		
Eb 111	4978.0317395533	123	9956.0634791066		
E 112	5274.0409106059	124	10548.0818212118		
F 113	5587.6517029281	125	11175.3034058561		
Gb 114	5919.9107633862	126	11839.8215267723		
G 115	6271.9269757080	127	12543.8539514160		
Ab 116	6644.8751612791				
A 117	7040.0000000000				
Bb 118	7458.6201842894				
B 119	7902.1328200980				

Figure 1 MIDI notes to the corresponding frequencies (Scott, 2011)

Visualising this wave helps cement the concepts, whereby you can plot a graph of air pressure against time.

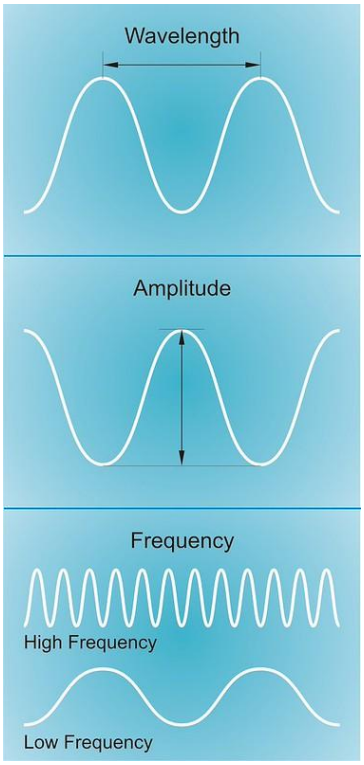

Figure 2 Visual representation of waveform attributes (Nichols, 2011)

3. Phase

The final attribute of a wave, phase, describes how far through a wave's period it is. The point where a wave begins is called its initial phase (Roads, 2023). For example, a sine wave's initial phase would be its starting amplitude or 0, it then climbs to 1 before returning to 0, then repeats in the negative and so on. A cosine wave starts at 1 and follows the same increases and decreases as the sine wave. Therefore, if the sine wave is displaced by $\frac{\pi}{2}$ or 90°, it becomes identical to the cosine wave. The difference between these two waves, despite having the same amplitude and frequency, lies in their phases.

Another key concept related to phase is the interference caused by multiple overlapping waves. The overlapping of two waves is called superposition. This superposition is covered in detail by Aguilar et al. (2010), exploring the changes in a wave's phase and frequency upon superposition.

The most relevant part of this research is how the amplitude of the resultant wave changes. This is found by summing the displacements at each point of the wave. Two waves that are in phase, meaning that their peaks and troughs are aligned, will create constructive interference (Elkashef, 2023). Using the earlier principle, it follows that the resultant wave will have the combined amplitude of the two waves, and so the amplitude will increase. Destructive interference happens when the waves are 180° out of phase, meaning that when one wave peaks, the other wave troughs. Once again, using the earlier principle, this interference will result in a decrease in the amplitude of the wave.

Waves often do not have the same period in musical combinations, meaning their superposition will form a combination of constructive points where high or low points on both waves coincide and destructive points where high and low points oppose.

Understanding this interference is important when generating audio, since unique sounds can be created by interesting patterns of wave interference. Regarding the statement on amplitude, it should be noted that superimposing too many constructive waves can produce very loud results that may damage hearing. To prevent this, a method should be used to reduce the amplitude of the resultant wave based on the waves that factor into its superposition.

Digitising Sound:

1. Sampling:

Now that the fundamentals of sound waves have been established, the next logical step is to understand how the computer interprets these waves. Computers have no ears; they can't hear sound or sense waves in the manner that people do. They can, however, interpret fluctuations in voltage, as Roads (2023) explains. A microphone picks up incoming sound waves and converts them to a corresponding voltage. The computer can then read these voltage values and store them as best it possibly can. From here, when needed, the voltage values can be converted back into sound waves for playback to the human ear via speakers. This sounds simple, but it obfuscates the complexity of how these voltage values are converted into readable data.

Each time the computer stores the value of the wave is called a sample. A computer can only sample values discretely; however, a wave is a continuous medium with constantly changing values. This means that, at best, a computer can only recreate an approximation of the wave. This comes with its fair share of issues.

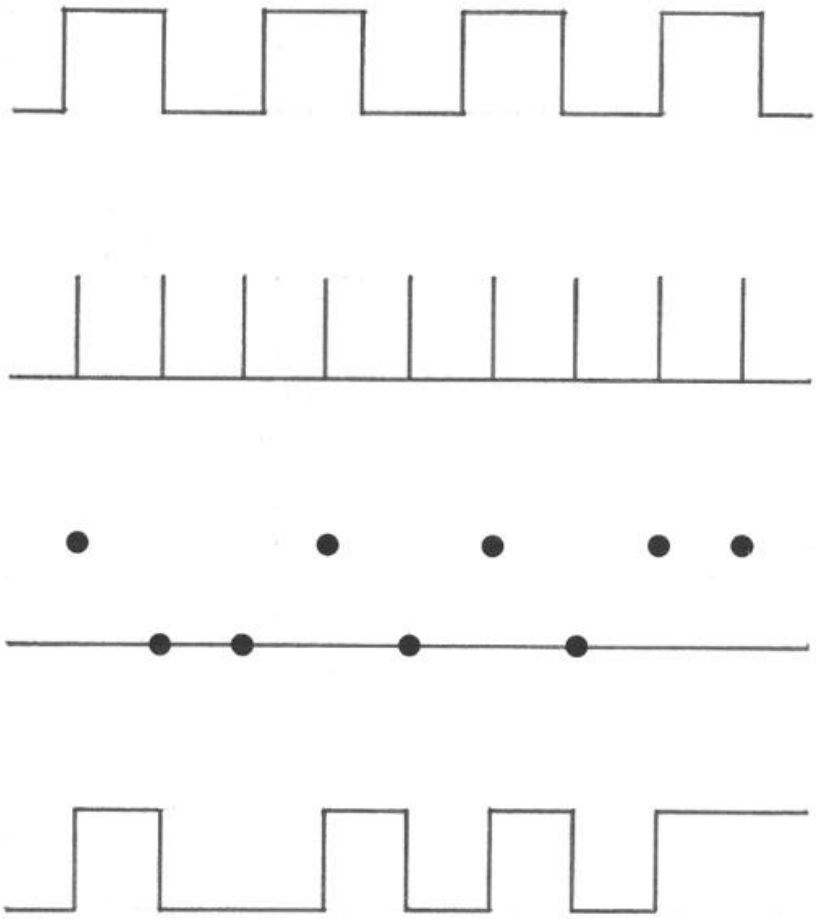

Figure 3 Sampling Issues

The Figure above highlights one of these issues. The top level shows the initial wave, and the one below it represents every time the computer takes a sample of the wave. By interpreting the samples (the dots), the computer would recreate the wave as shown at the bottom. This is very different from the original wave and will sound distinctly different.

The solution to this is quite simple, and is stated by Nyquist's Theorem that the sampling rate for any given continuous wave must be greater than twice the highest frequency of the wave (Shanon, 1949). This essentially means that two samples are required per period of the wave for recording at least the peak and trough of it. The range of human hearing is from 20Hz to 20,000Hz (Oxenham, 2017). This means that to capture the full range of human hearing for any wave, no matter its frequency, a sampling rate of at least twice the upper limit of human hearing is required to capture all audible sounds a human can hear.

2. Real-Time troubles

The widely used sampling rate is 44,100Hz, as it satisfies Nyquist's requirement, and while this isn't a problem for simply recording or replaying audio, it is quite a challenge in generative audio. Doulmer (2021) highlights them. When playing back audio, the computer's sound card expects 44,100 samples per second. If it, for any reason, does not get these samples, unexpected data will be sent to the sound card, and an audible anomaly that can sound very jarring to a listener will occur. This means when generating sound in real time, there is a strict deadline for when the generated samples must be ready to be sent to the sound card.

Another problem with this arises, though. Sending that many samples to the sound card one at a time is incredibly expensive for the computer. This means the samples produced must be batched together so as to not overwhelm the computer. This can introduce latency in the audio, as blocks of samples are written to be sent to the sound card, anticipating what the future audio might be. This means that if the program needs to react to stimuli, whether from a user or from whatever source is playing the music, there will be a delay equal to the number of samples in the sample blocks that have already been queued to send to the sound card. Therefore, if real-time stimulus is desired as a possible consideration for the generated audio, it becomes necessary to balance the number of samples sent with how long the latency is.

3. Quantisation:

The sampling rate is not the only trouble when it comes to digitising audio. Once again, due to the discrete nature of a computer, there is only a discrete range at which the value of the wave at any given point can be recorded (Roads, 2023). This is called quantisation. For example, if the wave's voltage reads 63.7 V but the computer only stores voltages as integers, some accuracy to the original sound will be lost as the quantised value will be 63 V instead. The only way to combat this is to increase the number of bits the computer can assign to each sample. Every bit assigned will help preserve the audio's original sound, but it will also drastically increase the memory size required to store it. For example, storing 1 minute of audio at a sampling rate of 44,100Hz and assigning 8 bits per sample produces a file size of 2646 KB; increasing this to 10 bits per sample produces a file size of 3307 KB. This is a 25% increase for only an additional 2 bits. It is important to consider how much accuracy can be lost from the true audio signal to maintain a reasonably sized file. Other methods, such as filtering out high frequencies, which are inaudible to humans, are effective at reducing file size without losing relevant audible data.

Sound Synthesis:

Creating or attempting to mimic sounds is generally called sound synthesis. It uses the aforementioned properties of waves in unique ways to create unique sounds. Often, oscillators are the primary source of making the base sound due to their periodic nature.

Filters, envelopes and frequency modulators are commonly used to modify these waveforms, altering the fundamental properties of the wave itself. There are also several methods for combining the waves produced and modified.

Sound creation:

Oscillators are generally what produce a waveform itself. For example, a simple sine-wave oscillator can control the frequency and amplitude of the outputted sine wave due to its periodic nature. These oscillators provide the foundation on which the rest of the tools operate, creating more unique sounds rather than having everything be the same few waves an oscillator can produce.

Waveform Modification:

1. Filters

Filters work by attenuating certain frequencies of a waveform. Which frequencies these are can be determined by the user (Roginska, 2018). Some generalised terms help explain how a filter works. The passband determines what frequencies should be allowed through the filter, while the stopband works to the contrary, the frequencies that aren't allowed through. The point from where the passband becomes the stopband is known as the cutoff frequency. Ideally, after the cutoff frequency, all frequencies beyond this point would have an amplitude of 0 and be fully

attenuated from the wave. This isn't realistically possible, and so the frequencies become more attenuated as they get deeper into the stopband. This is called the filter's slope. Filters are a powerful way to modify a waveform and can be dynamically adjusted to produce interesting effects. For example, a low-pass filter only allows low frequencies to pass through the filter (Kadam, 2023); this can make the audio sound as if it is being played in another nearby room. So, to get closer to this theoretical room, the filter's cutoff frequency can be increased proportionally as the room approaches.

2. Envelopes

Envelopes determine a set way to handle how the waveform changes over time (Rose, 2015). This is an attempt to mimic how real instruments sound and is often broken into four separate categories. The attack is the buildup of the sound as it initially starts, going from zero to some arbitrary peak. The decay follows, coming down from the peak. This decays until it reaches the sustain amplitude, after a given time, at which point the amplitude is held at this constant value, often for the rest of the sound's uptime. Finally, once the sound should stop, the release time slowly decreases the amplitude back to zero, or its starting level. The envelope for a piano key would be very different to that for a bell, for example, a piano key will keep playing for as long as you hold the note, whereas a bell will fade out quickly unless struck again.

3. Frequency Modulators

Frequency modulators change the frequency of an initial waveform, commonly known as the carrier, in this case, by using another waveform known as the modulator (Russ, 2012). This is different from synthesis, which combines waveforms; this isn't a combination but rather a modification. This means that the initial waveform oscillates at the modulator's frequency while still outputting values centred on the initial waveform.

Waveform Combination:

1. Additive Synthesis

Roads (2023) describes additive synthesis as the process of summing multiple waveforms to produce a new one, detailing that this type of synthesis was among the first types of synthesis, due to it being a naturally occurring process, i.e. playing multiple notes to create a chord, yet it is still popular to this day. It heavily relies on the use of the Fourier series, which can describe any periodic function (waveform) in terms of the addition of sine and cosine waves. For example, the common sawtooth wave is simply the addition of an infinite number of sine waves with increasing frequency in multiples of the base and a lowering amplitude in fractions of the base. Despite this being a very powerful technique, it can get complicated when working with many waves, and so many people prefer simpler methods.

2. Multiple Wave Table Synthesis

Wave table synthesis is the method of generating a wave by sampling from a lookup table of a preset waveform (Pirkle, 2021). This makes creating the wave very fast, but limits you to the frequencies inside the original waveform. The pitch of the sound can be adjusted by changing the speed at which the wavetable is played, for example, skipping or repeating lookups. Multiple wave table synthesis gets around this by using multiple waves to interpolate between, allowing for changes in tonality.

3. Wave Terrain Synthesis

Wave terrain synthesis works by analysing a multidimensional space to create a wave. For example, a given 3D space will be split into several sections, which will then be traversed by a

given trajectory. As it reaches each section, it can examine its x and y positions in the theoretical or physically modelled 3D space to determine the wave's function, with the z-value corresponding to the amplitude at that point in the function (Roma, 2023). It offers a unique way to produce sounds, though without a very tailored terrain, musical or even simple pleasant sounds can be hard to form from it.

4. Granular Synthesis

Granular synthesis produces sounds by taking grains (an acoustic event of a nearly imperceptible duration) and combining thousands of them to build a complete waveform, much like a digital image is really made up of thousands of colour grains or pixels (Russ, 2012). Each grain can act as its own unique waveform, modifiable with synthesis techniques like envelopes and filters. This helps create a rich sound environment, as it can make very textured sounds.

5. Subtractive Synthesis

Subtractive synthesis works by taking a very rich source of sound, such as white noise, and processing it with filters to remove certain frequencies and boost others. This works well for combining sounds that don't initially seem very common, but by limiting them to specific frequencies, you can find harmonic similarities that blend nicely (Creasey, 2017).

6. Wave shaping Synthesis

The idea of wave shaping is to distort a given wave using a distortion function. This function maps the original values onto different values but always produces the same result for a given input, meaning the distortion is constant in this case (Roads, 2023).

Sound Recreation:

Fourier Series & Transform

Joseph Fourier developed his Fourier series in the 1800s for a matter seemingly unrelated to sound. The similarity between them lies in Fourier's attempt to represent the changes in the heat distribution between metal rods graphically using basic trigonometric functions. Waves. He proposed that any wave equation can be broken down into a summation of sine and cosine waves. This is the Fourier Series, where p is the period of the function. (Howell, 2016)

$$f(t) = A_0 + \sum_{k=1}^{\infty} a_k \cos\left(\frac{2\pi k}{p}t\right) + \sum_{k=1}^{\infty} b_k \sin\left(\frac{2\pi k}{p}t\right)$$

Knowing that any wave function can be represented in this manner is useful, but the real question is what comes from this knowledge. Given the Fourier series, each of the individual sine and cosine waves that make up the greater wave function can be determined, allowing for the possible breakdown and reconstruction of the wave function. This can be done by the following functions, where k is a positive integer.

$$A_0 = \frac{1}{p}\int_{t=0}^{p} f(t)dt$$

$$a_k = \frac{2}{p}\int_{t=0}^{p} f(t)\cos\left(\frac{2\pi k}{p}t\right)dt$$

$$b_k = \frac{2}{p}\int_{t=0}^{p} f(t)\sin\left(\frac{2\pi k}{p}t\right)dt$$

Now knowing which trigonometric functions build the wave function, it can be easily reconstructed, which is very useful for exploring and recreating audio, especially musical instruments (Alhodairy & Beitalmal, 2024). As discussed, additive synthesis also works by the principle of summing sine waves to create a more detailed sound. Using the Fourier Series, any sound can be effectively recreated to a high degree of accuracy by simply combining these waves and experimenting with the alteration of their properties. The cosine half of the series is not an issue, since a key aspect of both the sine and cosine waves is that they have the same waveform and are simply out of phase with each other. Accounting for this phase shift turns a cosine wave into a sine wave.

Fourier's contribution to audio doesn't end here, however, as he also laid the groundwork for the Fourier Transform. The Fourier Transform is a mathematical technique that transforms a function from the time domain, i.e. a graph of amplitude over time, to the frequency domain, i.e. a graph of amplitude over frequency. This allows for the examination of all frequencies in a given wave, which is very useful for audio processing. Allowing for techniques such as identifying select frequencies to remove with filtering or boost with synthesis. This technique is more commonly known as spectral analysis (Khodzhaev, 2022). To go from a time domain wave to a frequency domain wave, one would use the following integral, where w is the angular velocity of the wave $\frac{1}{p}$.

$$f(t) = \int_{-\infty}^{\infty} F(w) e^{2\pi i w t} dw$$

Additionally, to go from a wave in the frequency domain back into a time domain wave, the inverse Fourier transform should be used

$$F(w) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i w t} dt$$

This is just the basics of the Fourier transform; in practice, for computer programs, improvements such as DFT (Discrete Fourier Transform), which allows the Fourier transform to work on a discrete-time signal rather than a continuous one, replacing the integral with a summation, are required. Other improvements, such as FFT (Fast Fourier Transform), which reduces the high time complexity for the DFT original function at $O(n^2)$ down to a function with a $\frac{n}{2} O(n \log_2(n))$ time complexity (Burrus, 2012).

Methods in Algorithmic Composition:

Generative Grammars

A formal grammar is the structured description of a language. By using the symbols defined in the language, the grammar specifies valid terms within the language and the rules which determine valid constructions of the terms (Roads & Wieneke, 1979). Grammars provide a means to structure music into discrete and distinct facets. Using this basic structure, different kinds of music can be interpreted from it. However, categorising music according to strict rules can lose the unique quality of music to be a unique expression of sound, as the grammar instead defines hard limits on how the music should proceed. Similarly to this, generative grammars can be very limiting in terms of genre or other social aspects of music, as what rules may work for a certain kind of composition may be completely irrelevant or counterintuitive in

other compositions. Building a grammar for a particular type of music can be very effective and replicate sounds similar to the desired effect (Holtzman, 1981). Still, it doesn't allow much room for adaptation or for the generation of emergent sounds. Probabilistic grammars, which have a defined variety of outcomes from grammatic rules, provide more emergent sounds but are still very self-similar unless using a large variety of alternate resolutions of rules.

Markov Models

A Markov chain is a mathematical system that transitions from one state to another. These transitions occur stochastically and depend solely upon the conditions of the current state, not previous ones (Ibe, 2013). An easy example of this is flipping a coin. The coin has two discrete states, heads or tails, each with an equal weighted probability. The transition between these states is flipping the coin. Using this Markov chain would provide a sequence of heads and tails in a stochastic order. Markov Chains can hence be constructed by analysing pre-existing music and identifying the transitions between notes, octaves, instruments, etc. This can be used to produce a sequence of notes where the original music is a possible outcome, but also new emergent music based on the original note progression is also a possibility. As music often has many possible outcomes, which notes are played, and how long a note is played for (Miranda, 2001), this can quickly cause a Markov Chain to spiral into a very large and unwieldy graph. A proposed solution to this is to run several Markov Chains in parallel to handle this. For example, a chain for pitch and a chain for duration. This allows for easier management of the chain but may lose some of the original coherence as aspects of the sound are split from their respective characteristics. Because Markov Chains don't consider previous states, their generation struggles to produce a consistent musical context and is often better suited to ambience over a structured piece of music. This can be somewhat mitigated by not looking at the music on a per-note level, but rather at a per-phrase level (Jan, 2022), or by keeping track of previous notes to help inform the next ones.

Some generative methods work better for some types of instruments than others. For example, drums often play a supporting role in a piece of music, keeping a consistent beat throughout the piece. Methods to generate these drums, such as Markov Chains, could lead to a very vocal drum set in a piece of music where a steady rhythm is preferred; a generative grammar for the drums could work better, limiting the drums to a structured rhythm instead.

Chaos and Self-similarity

Chaos theory is the study of randomness in complex nonlinear systems. It breaks down these complex systems into patterns and deterministic equations, which is known as self-similarity (Nierhaus, 2011). This means that these systems are essentially the same as any given choice of the system, only scaled to be magnitudes larger than the single choice. Given the possibility of exploring all choices of the system simultaneously over several iterations, it reveals that these seemingly random decisions have more order to them than what may be seen from the perspective of a single choice. For generative purposes, this plays into the idea that each part of music can be broken into an extremely large but finite state, simply stating what instruments are being played at that time, the lowest scale in the self-similarity hierarchy (Miranda, 2001). Hence, given infinite time, any and all songs that can be played will be played by the system. In practicality, throwing random numbers at the computer, hoping it produces something spectacular, is unfeasible for one lifetime.

Genetic Algorithms:

Genetic algorithms work on the theory of natural selection. The idea that species exhibit varying properties despite being the same species. These adaptations either help or hinder the subject's ability to survive and reproduce offspring with the same new adaptation as the parent, as well as adaptations of their own. This method enforces the gradual improvement of the subjects up to a certain standard. For computers, the attributes of data can be analysed through the use of a fitness function to determine how well the subject's parameters hold up to the desired objective (Affenzeller, 2009). The challenge when adapting this fitness function to suit a musical process is how to determine the traits of the subjects and how to determine if the changes have improved it or not. Describing a suitable fitness function is crucial to this approach, as individually assessing the subjects by hand defeats the purpose of genetic algorithms, drastically reducing the speed at which new generations with more favourable traits can be produced. Additionally, genetic algorithms ideally converge towards a single solution; however, in music, there can be multiple satisfactory outcomes from the generation, but genetic algorithms may discard these in favour of what they consider a preferable arrangement. Overall, genetic algorithms can be extremely powerful at generating well-sounding, unique music; however, they are extremely easy to get wrong, and a lot of time can be wasted trying to do it well (Tzimeas & Mangina, 2009). As well as their long computation time, they are also very resource-intensive, as populations can have hundreds to thousands of candidates that need to be simulated, likely making them unfeasible for real-time applications, but a powerful consideration in a non-runtime environment.

Cellular Automata:

At its core, cellular automata is a model for simulating complex systems using rules applied to a grid of cells. Like how wave terrain synthesis broke pieces of terrain into a way to describe a wave function, any cell in the cellular automata defines its state and future states by its position in its respective grid, its current state, and the neighbours around it. Once the grid has been defined, it is up to the developer to decide how to map these cellular automata to musical meaning. For example, each axis could define a property of the music, such as pitch or instrument. Then, assigning the different states of a cell to some property, which could be as simple as on/off. Finally, defining how transitions between states occur, which should be based on some degree of musical theory (Miranda & Shaji, 2023). As a result, cellular automata are very flexible in their musical expression, producing unique and often good leaping points into a musical composition. However, due to its very unstructured nature, it is often ineffective at producing consistent melodies for longer periods of music.

Neural Networks:

Neural networks are a facet of machine learning that aim to solve problems by reproducing the way neurons fire in the brain to process data, with each neuron influencing and firing the neurons it's connected to, where eventually the most prominent path of fired neurons is the answer to a given problem. Each neuron in a neural network holds a state related to the data. The initial set or layer of neurons, also known as the input, can hold all possible states the data can be in. The output holds all the possible states that would be expected upon processing the input. Between the input and output layers are hidden layers that can have a variable number of neurons based on the problem being solved. Each layer of the network informs how the next layer calculates its states (Massaron, 2019). Similarly to Markov chains, expanding a neural network can become very memory-intensive as each increase in state or layer drastically increases the number of values to store and compute. However, unlike Markov Chains, the

calculations can be condensed into matrix calculations, which allows for easy parallelisation and the use of SIMD or MIMD operations. In this sense, content can be procedurally generated, as given an initial state, whether that be random or pre-defined, the neural network will return what it has found to be the next most likely state, in a similar vein to a Markov chain, only with enhanced learning abilities. Alternatively, or in conjunction with a generator, a neural network can be used to evaluate how good a generated state is, discarding states that have been generated badly and reinforcing ones that have been well generated due to its ability to accurately recognise patterns, not dissimilar to how genetic algorithms use a fitness function to determine the rating for a subject in a population (Liu et al., 2021). A drawback of neural networks is that they require training. Much like the human brain, neural networks work via pattern recognition, but take comparatively longer than a human brain to make these connections. To this end, a large amount of manually labelled data with the correct outcome must be provided to the neural network so that, over many iterations, it can determine the correct output given a widely varying input by altering how each layer connects to the next. Various methods to reduce the time required for a neural network to learn, such as backpropagation, can be used, but it remains a time-consuming and computationally heavy process. Even if the neural network is producing the correct outcome for a given set of data, slightly altering the data or expanding the scope of the data it is given can cause it to falter, and it must relearn given the new parameters. Additionally, the network may be giving the correct answer based on wrong assumptions or connections it has made. Enforcing these incorrect assumptions may cause the network to take longer to converge on the solution or force it to require retraining with new boundaries that fit the correct assumptions. Also, finding sufficient data or creating enough data that is accurately labelled for the desired outcome can be hard and/or costly. Neural networks, therefore, are a powerful tool for creating procedural audio and can accurately and effectively produce consistent-sounding audio based on the given input space (Miranda, 2001). However, the time and cost of creating a good neural network, as well as the narrowness of a network in such a wide field of audio, may leave this as a viable option only to those who have a large budget in time and money.

Procedural Audio in Games:

Computer games didn't always have audio to accompany them. The following information is based on Horowitz and Looney (2014). In the late 1960s and early 1970s, as games were beginning to take off with the popularity of arcades, the first games started introducing simple beeps and boops alongside the gameplay. These beeps and boops were created on the hardware using basic oscillators, much as they are today; in other words, they were all procedurally generated. Games back then didn't have the hardware to facilitate scores of music on them, and so generating audio on the fly was necessary. For reference as to the hardware specifications, one of the first sound cards used in a games console was the Atari 2600's TIA (Television Interface Adaptor), which had two channels at a single bit for monophonic audio. Pong, while not the first game ever made, was the first to utilise audio as feedback to the player and drew crowds because of it. It wasn't until 1991, with the Super Famicon, that consoles with the ability to pre-mix audio were seen, which could be played back in the game. With only a few megabytes of data per cartridge, however, the audio budget was still very limited. The utilisation of CD-ROMs offered the first real possibility of pre-storing soundtracks and sound effects for games; from there, the hardware only improved, and generating the audio live became less and less favourable.

This shift makes sense, given the hardware and knowledge of the time, recordings of instruments that had been digitised would have sounded significantly better than the simple oscillators that were available. Creating sound inside the game itself was also not a very intuitive task, and rarely did developers have a crossover in knowledge between knowing the electrical engineering or programming skills required to make hardware create sound, and the know-how of music theory to create good-sounding music.

Current hardware allows for much better integration of these aspects though, with middleware software like DAWs (Digital Audio Workstations) that are dedicated to audio synthesis methods and allow the creation of "digital instruments" that can be redistributed and used in any project, and with the universal standard of the MIDI format for audio, which allows for easy editing of audio and instrument substitutions, among other things. These allow a composer to create music for a game without the need to ever communicate with the hardware the game is running on (Ciesla, 2022). The question, therefore, is why, given all these tools to create audio on the fly, has it not once again become a popular method for creating audio in games?

This, along with a variety of questions in the same vein, has been asked to industry-standard professionals in interviews by Böttcher (2013). The highlights from this outline that the sound quality of procedural music often does not stand up to standard recorded audio, and it's very hard to make it do so. It is also explained that the tooling for procedural methods is lacking for producing audio in large commercial projects. This is explained by a lack of communication between game studios and academic research being done into developing these tools, as games often don't reveal the software, processes, and systems behind their development until they've become obsolete. Academics also don't often turn to game studios when considering how to inform their research and design tools. Procedural tools created by industry are also uncommon, as the expected outcome of building these tools for their game would naturally be an increase in sales, either equal to or exceeding the amount spent researching and developing them, and this is not an outcome that can be guaranteed. Instead, the best commercial benefit for a game's sound production is to have a famous composer or artist contributing to the sound in the game, to garner popularity. The general ideation for the current best use of procedural audio is to mix pre-recorded and procedural audio, whether this mix is done by generating pre-recorded samples or leaving certain areas of a game that only require lower-quality audio to be handled procedurally, such as repetitive sound effects.

Some games make use of procedural methods, but don't rely on a fully procedural soundtrack, such as No Man's Sky. In an interview with Paul Weir, audio director for the game by Mongeau (2017), Weir described how the sounds of the environment, namely the creatures and the fauna, are procedurally generated and described that this system was very efficient with low average CPU usage. Despite the positives of this system, he states that procedural audio should only be used where traditional sound design cannot reasonably be used.

Some games have experimented with using procedural methods for larger sections of the game. An example of this is SPORE, explained by Jolly and McLeran (2008), where the procedural part of the soundtrack was largely worked on by Brian Eno. A majority of the content in SPORE is procedurally generated, so it made sense to create a soundtrack that was procedurally generated to match whatever state the game ended up in. This was done using something called the "shuffler", which would play different musical instruments at regular musical intervals based on factors from the game. The well-documented area of this is in the game's "creature creation" state, where a player will spend a large amount of time making the

creature they want. The procedural music here helps the player avoid getting tired of hearing the same soundtrack on an endless loop for this time. The game still uses some more traditional methods of audio in the game, with pre-recorded sounds or music for important key moments in the game.

The key takeaway from this is that the use of procedural music is better suited to non-critical pieces of music than to critical ones. Due to its random nature, there's no guarantee that a piece of generated audio will reproduce the desired effect of a pre-made piece of audio.

Research Methodologies

Generation Method Selection

To test how feasible procedural generation methods are for assisting games, they should be tested and compared against desired metrics. Among the possible methods, each has its upsides and downsides.

Generative grammars require detailed insight into music theory, which isn't an area of focus in the paper, and so shouldn't be used.

Chaos is too random to the point that it can be excluded without any need for testing, as it would take far too long, in or out of a real-time process, to be useful.

Genetic algorithms, while powerful, require a strong fitness function to determine if a piece created is of a satisfactory standard. This function may change drastically depending on the genre of music, what instruments are used in the piece, and, once again, music theory, so it's off the table.

Cellular automata, while could be an interesting method to analyse to see whether musical patterns could be found under the right conditions, requires more research into that area. Its initial nature is too random to generate consistent audio without additional boundaries.

Neural networks might provide the most powerful and, hence, produce the highest-quality audio out of all the methods, which is a very strong argument. However, the prerequisite necessity of training data and the issue, similar to that of genetic algorithms, that two very different pieces of audio can be of high quality, but the network may only understand one of them as such, are large enough catches to prevent this method from being used.

Finally, there is the use of Markov chains. The main downside to these is their lack of structure, making them quite random, and with a large possible number of outcomes for a state, it creates a large graph of state changes. Additionally, it requires pre-existing data to examine to generate self-similar audio; the required data is much less than methods like neural networks, though, often being only a single song or a set of similar songs. Unlike the other investigated generation methods, it is not so random that the original theming of the music can get lost or never be recovered, and it is not limited to any single genre of music, as it works based on transitions between notes, the core building blocks of any musical piece. Due to these quick transitions, it should also be fast enough for use within a real-time application.

For these reasons, Markov chains will be the selected procedural generation method for the artefact.

Artefact Considerations

It is critical that the artefact is lightweight so that it doesn't consume a lot of the computer's resources. This is a requirement, as the artefact will not be viable in a real-time application, especially one such as games, which already require a large amount of memory and CPU otherwise. This necessity does not encompass the entire scope of the artefact, as the construction of the Markov chain can be done before real-time-intensive operations must begin. Such as during the boot-up of a game or even before the game has even started. This is because only the transition weightings and their storage are strictly necessary for generation in a runtime environment. This assumes the Markov chain does not need to dynamically change its transition weightings based on shifting factors in the program it is running in or alongside. Another necessity of runtime environments is time. At a standard sample rate, a computer will need at least 44,100 amplitudes of generated sound data a second. If this budget is not met, there will certainly be large auditory artefacts where audio randomly stops, or worse, continues at a single, possibly uncomfortable amplitude until it receives the samples it needs. This shows why it is important for the artefact to generate this audio in a timely manner.

Additionally, working with the knowledge of why games currently don't use procedural audio, the artefact should be easily usable for those without knowledge of audio engineering or programming. It should also be compatible with common audio storage file formats used in digital audio processing, modification and playback. This is so that, if the generated audio needs to be modified in some way using industry-standard audio engineering software, it can be easily transferred from the artefact to other pieces of software. Providing the basis of an easy-to-use and widely transferable procedural audio tool.

Testing expectations

Based on these decisions, an artefact will be created that tests the limits and possibilities of Markov chains as a method of procedurally generating audio. The testing methods will be used to obtain quantitative data to determine the quality of the audio produced.

This includes identifying auditory artefacts, such as drastic pitch changes, volumes that are far too loud or quiet compared to other notes in the audio, and other common audio artefacts, such as blips when a note is ended abruptly, forcing a large change in amplitude. These artefacts should be able to be seen by analysing the waveform of the generated audio.

Using a spectrogram, it is possible to identify extremely high-frequency sounds that are sharp and could hurt the ear, or that are beyond human hearing, which are wasted data. It can also show the "richness" of the audio, the idea of how detailed each note in the audio is. A single sine wave would not be a rich note. Lots of layered sine waves that occur at different amplitudes and different frequencies would be a rich note. Ideally, this richness would span the hearable frequency spectrum, from 0 Hz to the limit of human hearing at around 20,000 Hz, as high-quality sound does.

Tests will also be performed to monitor the artefact's strain on a computer's CPU and memory, with each area of the artefact tested separately and combined to give a clear idea of what parts of the process are costly and what parts are cheap. Similarly, tests measuring the time taken to complete all sections will be performed, aiming to hit the required 44,100 samples a second for real-time applications.

The artefact will also be judged on its accessibility for use alongside other common audio engineering programs, encapsulating the file types it can read into the process, and the file types it outputs as results.

The data gathered from these quantitative tests will be used to determine the feasibility of procedurally generating audio in a real-time application such as a game. These values will be compared graphically to provide succinct, easily distinguishable results.

Artefact Technicals

The artefact will be built in the C++ language because it is highly performant and allows for direct memory manipulation. This is useful for optimising the process to be as efficient as possible (Madhav, 2018). For these reasons, C++ is the most popular language used for creating games or game engines; even Epic Games' (2019) Unreal Engine is written in C++. This means that writing the artefact in C++ would ensure that the process can be easily integrated into real-time game projects without any additional steps needing to be taken. The source code can be directly transferred into an existing project.

As the artefact aims to be transferable code, it does not necessitate a visual interface, as this interface would only clutter the codebase with unnecessary logic that must be stripped when transferring it to other programs.

The artefact aims to still be easy to use, so at a high level, a user will be able to change which piece of music they want to generate from with a simple change of file name, and change the length of the piece of music they want to generate with a simple change of an easily accessible number. This means that for the more technically proficient people, they can rip apart the codebase using the pieces that they want, discarding the rest, such as adding or changing instruments, and for the less technically proficient users, they can simply change a file path and number to get the strong results they are looking for.

As the focus of the artefact is not on file parsing, the project will make use of the midifile open-source library (Sapp, 2024). This library will be used for reading MIDI files and extracting the events in the piece. MIDI files are a common file format for pieces of music that are easily editable, where the events in them can identify the timing, duration, and type of notes played in the file.

Results and Findings

Deliverables from the Artefact

The artefact supports two file types. The more standard and widely usable format of MIDI and a custom RTTTL format. MIDI will work with any .mid file, and the file path concatenated with the .mid extension. This must be loaded into the sound generator before it can generate audio using the data gleaned from it. The only caveat is that MIDI supports many instruments, each with its own unique voice; only a few from a range of the major available instrument categories have been implemented. Many of the other unique instruments are played as the few implemented instruments instead. Some very unique sounds of MIDI, such as "FX 6 (goblins)", have no instrument associated with them and so will not be included in any generated piece of music unless an instrument accommodating this MIDI voice is added to the code. The RTTTL format is mostly redundant compared to the widespread use of MIDI files; however, it provides a way for someone to create their own simpler music file that the program can read and generate from. It

works by using three characters that define a note. The first being the note played from C-B, in music terms, including sharp notes or S indicating silence, a break in the music. The second character is the duration, which can be a duration of a full, half, quarter, eighth, sixteenth or one thirty-second note represented by 1, 2, 4, 8, 6, and 3, respectively. The final character denotes the octave the note is played in, ranging from 0 to 8. Multiple notes can be played simultaneously by adding a '+' between the notes to be played in unison. As for the sequencing, a change in the note being played should be signified by a ',' between notes. Finally, the file should start with a single number indicating the instrument that plays these notes. This number is based on the MIDI instrument list and has the same limitations as described for MIDI instruments.

From either of these file formats, the data within them is processed into a Markov chain per instrument, allowing multiple instruments to be played simultaneously. The transition weightings between states are all taken from the data gathered in the provided file.

Finally, using the calculated transition weights, the notes are reconstructed and assembled into an output .wav file that is self-similar in nature to the original file using the Markov chain's procedural methods. The use of .wav as an output was chosen due to it being a widely used file format in games for uncompressed audio data, which is easily transferable to other programs. The user can specify how many generated notes they would like the output to contain. This file can be transferred and listened to as the user sees fit.

Identifying Auditory Artefacts

The following tests were performed with the Audacity software (Mazzoni & Dannenberg, 2000). Audacity is a free, open-source audio editing software that allows a user to view the waveform of a piece of audio and, additionally, transform the waveform into a spectrogram to analyse the frequencies in the audio over the course of the piece. The point of these tests is to identify any auditory artefacts in the audio and compare the generated audio to the audio it was generated from.

Amplitude Waveforms Analysis

The key points to look for in analysing the waveform are whether there are consistently high amplitudes. As the amplitude only goes between -1 and 1, and combining notes with additive synthesis combines the amplitudes of these waves at any given moment, this combination will likely yield an amplitude greater than 1 or less than -1. The audio generation should handle this to prevent the audio from becoming an unchanging, loud, and possibly damaging piece to listen to. Additionally, any discontinuous points in the graph, where the amplitude jumps from one frequency to a very different one, can cause auditory blips that undermine the quality of the audio. Finally, the signal-to-noise ratio throughout the piece. This is unwanted noise that can be present in audio, such as background talking or generation that has continued after the note has ended.

Figure 4.1 below shows the waveform generated for a piece, 100 events long (approximately 1 minute and 15 seconds), using a single instrument, with amplitude on the vertical axis and time on the horizontal axis. It uses a MIDI recreation of Minecraft (Rosendfeld, 2011).

Figure 4.1 Generated piece waveform

Figure 4.2 below shows the waveform of a single note event in the above piece that lasts around a second.

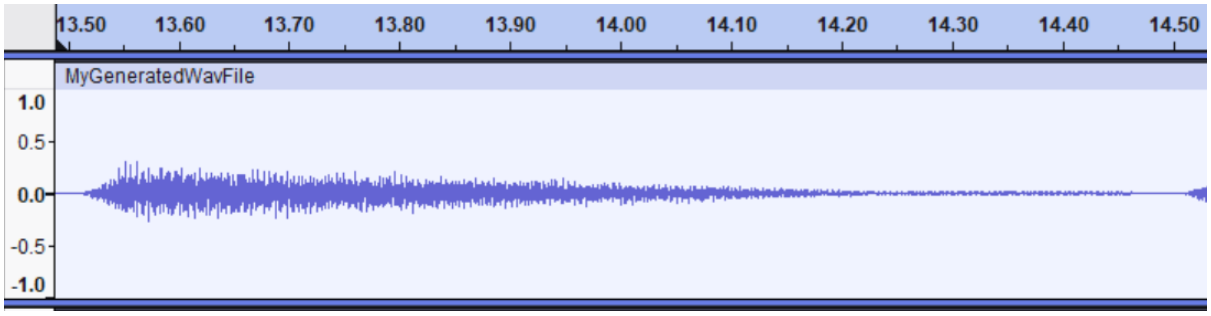

Figure 4.2 Generated piece note waveform

Figure 4.3 below shows the entire waveform of the piece, fed into the program, used for generation.

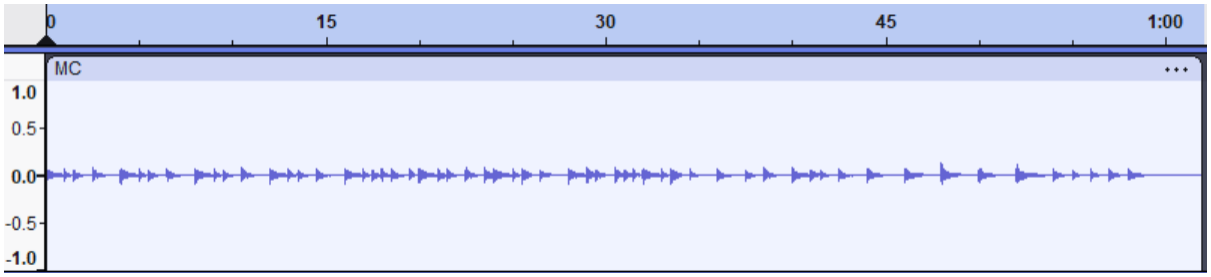

Figure 4.3 Original piece waveform

Figure 4.4 below shows the waveform of a single note in the original piece that lasts approximately one second.

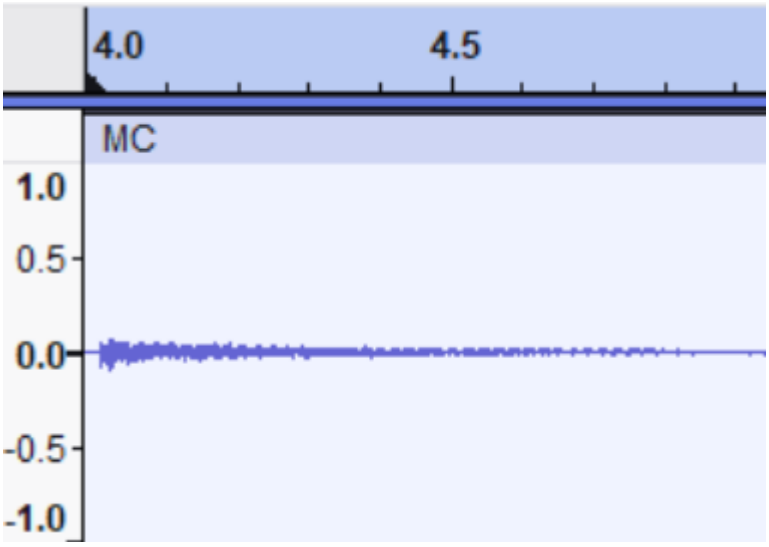

Figure 4.4 Original piece note waveform

Examining the generated waveform of the whole piece, there are no egregiously high amplitudes that would be an uncomfortable listening experience; the highest this piece gets is around an amplitude of 0.4. Looking at the generated piece as a whole, the amplitude does seem to jump from a very small amplitude to a large one very quickly; however, on closer examination of a single note, it does show a gradual build-up into the note, even if it is fairly quick compared to the duration of the rest of the note. This is consistent across all the notes in the piece. Looking at the whole piece and on a note-by-note basis, there is no amplitude between one note starting and another note ending, meaning there is no unwanted noise occurring at points or throughout the piece. This is again consistent across each note.

Comparing the generated waveform to the original waveform shows that the generated waveform outputs a higher amplitude than the original waveform, but they could be made similar by either boosting the volume of the original or lowering the volume of the generated waveform. Other than that, the waveforms look very similar to each other.

Figure 5.1 below shows the waveforms for a song with several instruments that are generated to play simultaneously with each other over 1 minute and 45 seconds of audio. The generator supports all the instruments in the song. The generated piece is on top, and the original is on the bottom. The audio used is a MIDI recreation of Battle 1 (Uematsu, 1991).

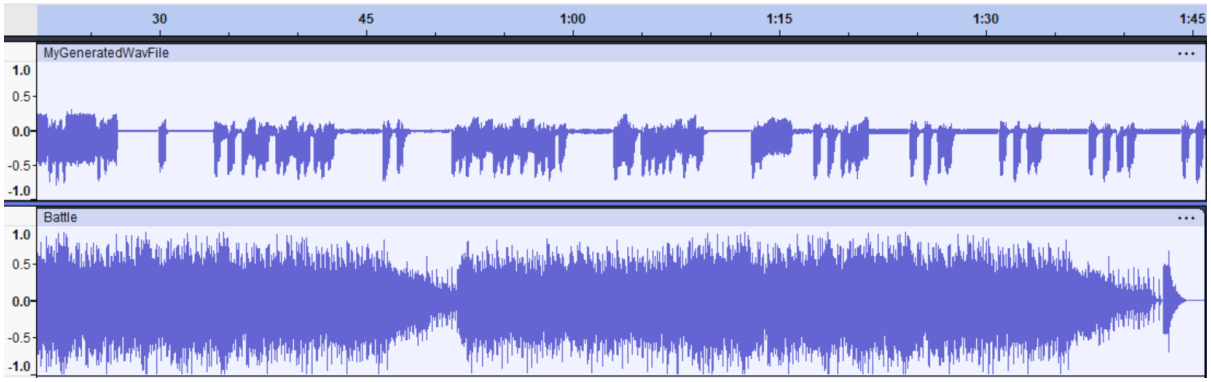

Figure 5.1 Comparison of generated and original piece waveforms

Figure 5.2 below shows a closer view of the waveforms spanning 2 seconds.

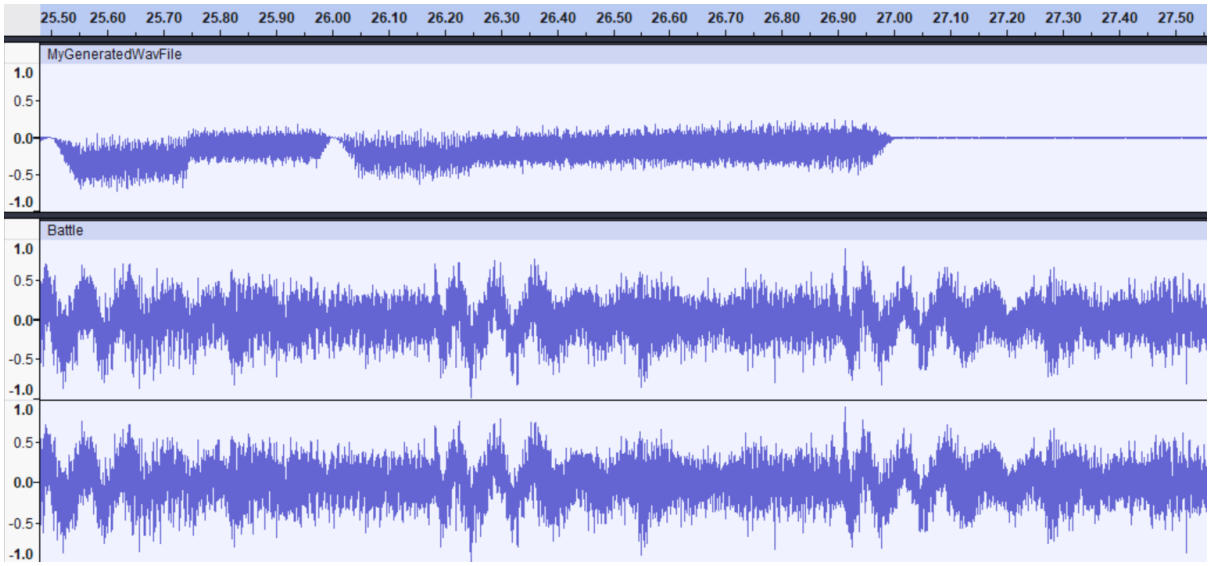

Figure 5.2 Close up comparison of waveforms

Examining the generated waveform shows many sharp negative amplitudes, although these have some buildup into them; repeated sharp changes in amplitudes do not sound good. Additionally, there are lots of empty spaces in the generated waveform where clearly no note is being played; however, the amplitude is not at 0 dB, leading to non-musical noise being produced in this time, which greatly detracts from the audio quality.

Comparing the waveforms shows an immediate difference between the two. The original waveform is rich with audio, with rarely anything being close to a break, while the generated waveform has lots of empty space. The original waveform has a large amplitude, but this amplitude is consistently smoother over the lifetime of individual notes and the lifetime of the song when compared with the generated waveform.

Spectral Analysis

The key points to look out for with spectral analysis are: frequencies above the range of human hearing, as this is useless data being stored; frequencies that jump around like the amplitude of the wave, as these would be frequencies in the wrong octave and would sound very dissonant; finally, the richness of the frequencies in each note, where a rich note would stretch and cover most of the hearing spectrum, while a non-rich note would be more localised around its key frequencies.

Figure 6.1 below is the spectrogram for the same piece as the waveform analysis in Figure 4.1, with frequency in Hertz on the vertical axis and time on the horizontal axis. The colour of the band represents how high the amplitude of the given frequency is, with lighter colours indicating a greater amplitude and darker colours indicating a lower amplitude.

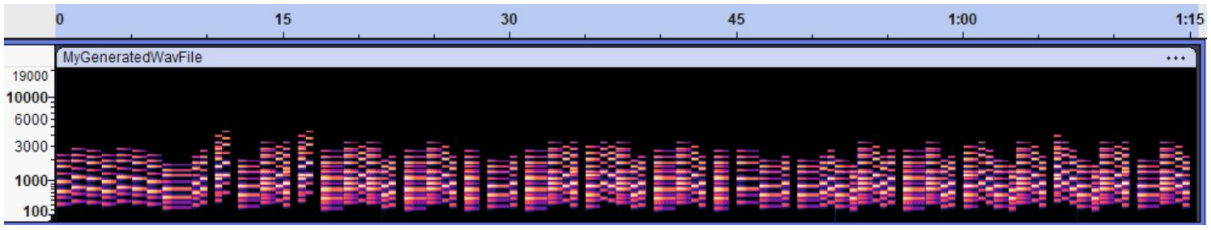

Figure 6.1 Spectrogram view of Figure 4.1

Figure 6.2 below shows the spectrogram of the original piece in Figure 4.2.

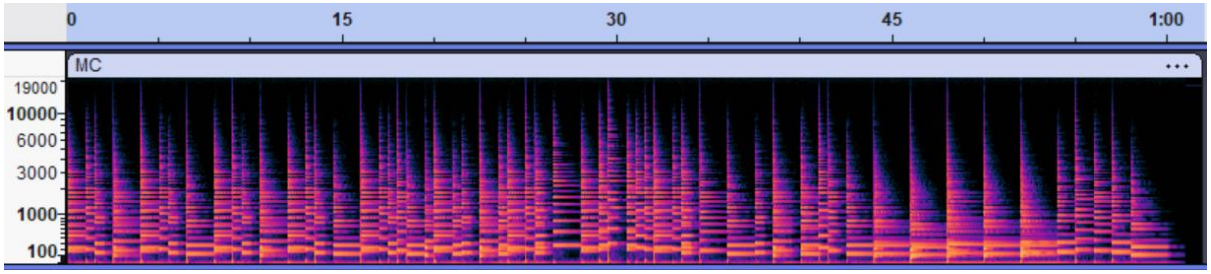

Figure 6.2 Spectrogram view of Figure 4.2

From examination of the generated waveform (Figure 6.1), it does not include any frequencies outside of the hearing range and therefore does not store any unimportant information. The frequencies also follow a smooth progression and don't jump around drastically. Some amount of silence precedes the largest jumps, so they will not be comparatively dissonant to the previous notes. The big difference between the graphs is how rich they are. The generated frequencies do not stretch very far from their fundamental frequency and across the rest of the available spectrum, instead being a lot more localised. The original piece (Figure 6.2), on the other hand, spans a much wider range over the spectrum than the generated counterpart, using the full range of frequencies and thus producing a very rich sound. Even within the present frequencies of the generated audio, there are significant gaps between closely connected frequencies, where this is not the case for the original audio source. Meaning the generated audio is much less rich than the original after undergoing the generative process. To a keen ear, this difference will be substantial, and the original will be preferable.

Figure 7 below shows the spectrograms for the piece with many layered instruments in Figure 5.1. The generated piece is on top, and the original is on the bottom.

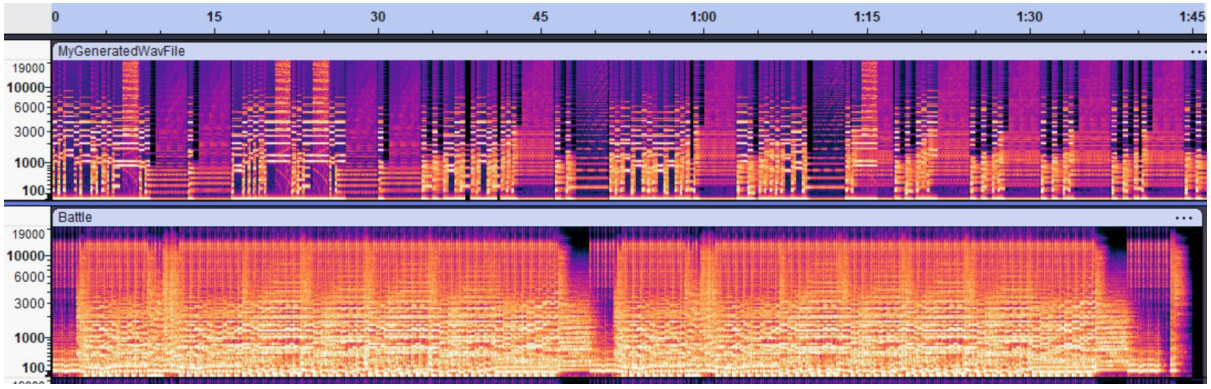

Figure 7 Spectrogram view of Figure 5.1

Unlike before, the generated waveform (top) spans a much wider spectrum of frequencies. However, these do not appear to lose amplitude much as they reach the higher frequencies, meaning the file likely contains frequencies outside the natural hearing range and sharp, unpleasant notes. The jumps between fundamental frequencies are also quite significant, which sounds dissonant. Compared to the original waveform (bottom), however, it is still very lacking in richness. There is a stark contrast between the two, which is very noticeable to a listener.

Runtime Resource Usage

Identifying how costly the generative process is on the computer is important to understanding if this method would be feasible in a real-time environment. The time budget for generation is at least 44,100 samples per second to keep up with real-time playback. The memory budget must also be relatively small, as other aspects of modern games are memory-intensive, and a game with too high a memory usage will be impossible to run on computers without the hardware to meet this memory requirement. This may cause the game to lose sales.

Figure 8 Below shows the process memory and CPU usage, as collected by Microsoft's Visual Studio (2022), for parsing the MIDI file, constructing the Markov chain, and generating an audio piece with 100 events in it. Based on previous examples, 100 events last, on average, around 1 minute and 15 seconds.

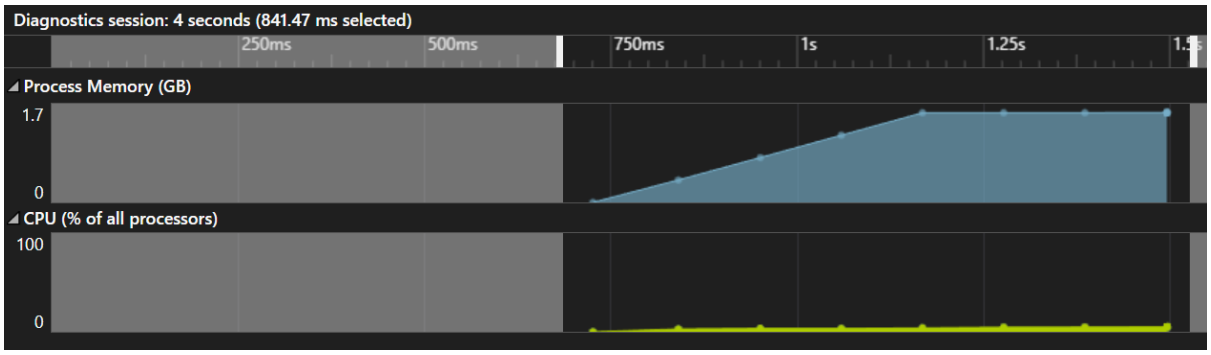

Figure 8 Visual Studio's performance profiler, diagnostic results

Figure 8 shows the program's total lifetime at approximately 800ms and the total memory used by the process at approximately 1.7GB. CPU usage does not exceed 4%.

Testing the memory usage over 50 attempts is consistent with the 1.7GB usage throughout, as shown in Figure 8.

Using custom timers, the time taken can be broken down into the program's constituent parts.

Figure 9 below shows the time, in microseconds, taken for each of the major operations in the program.

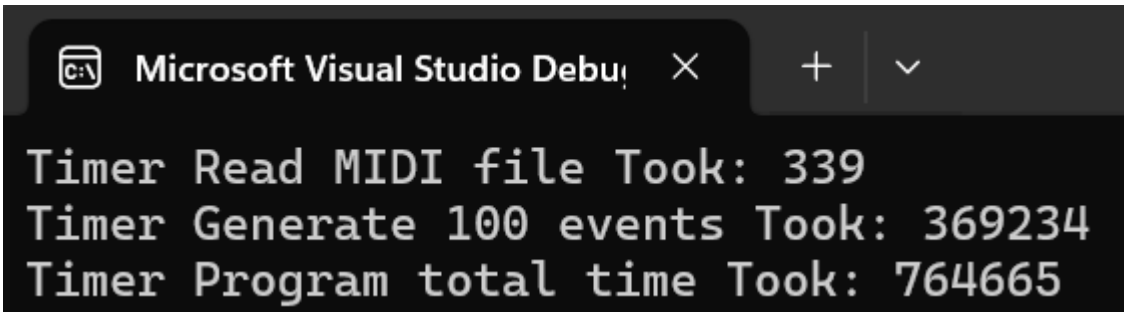

Figure 9 Program operation timings in microseconds

Figure 9 shows that parsing the MIDI file is extremely quick at around 340 microseconds (0.00034 seconds). The generation took 369,234 microseconds (0.369234 seconds), which is significantly longer. The big question is where the rest of the time came from, as these two times don't add up to the final time.

Figure 10 below shows the percentage of time the CPU spent in each function.

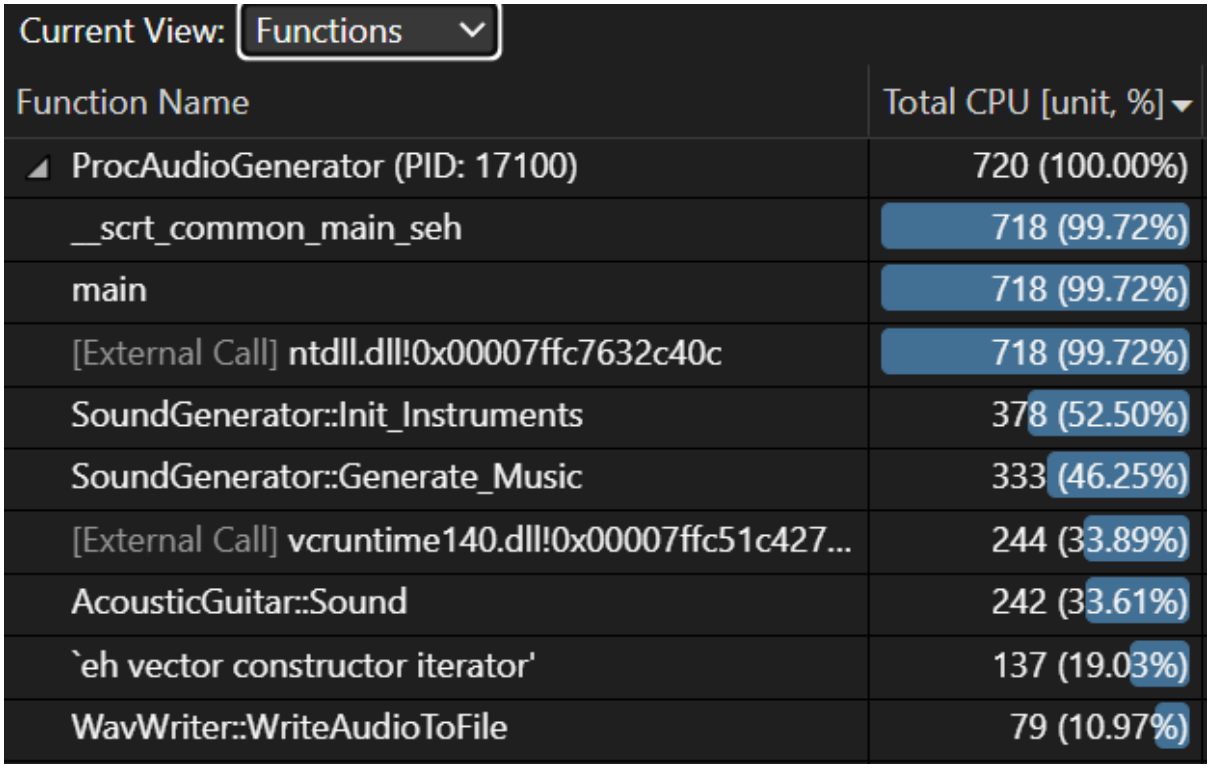

Figure 10 Visual Studio's performance profiler, function view

The WavWriter::WriteAudioToFile and AcousticGuitar::Sound are both encapsulated inside the generate music function. This means the remaining time is spent inside Init_Instruments.

Taking a look at that function in Figure 11 below

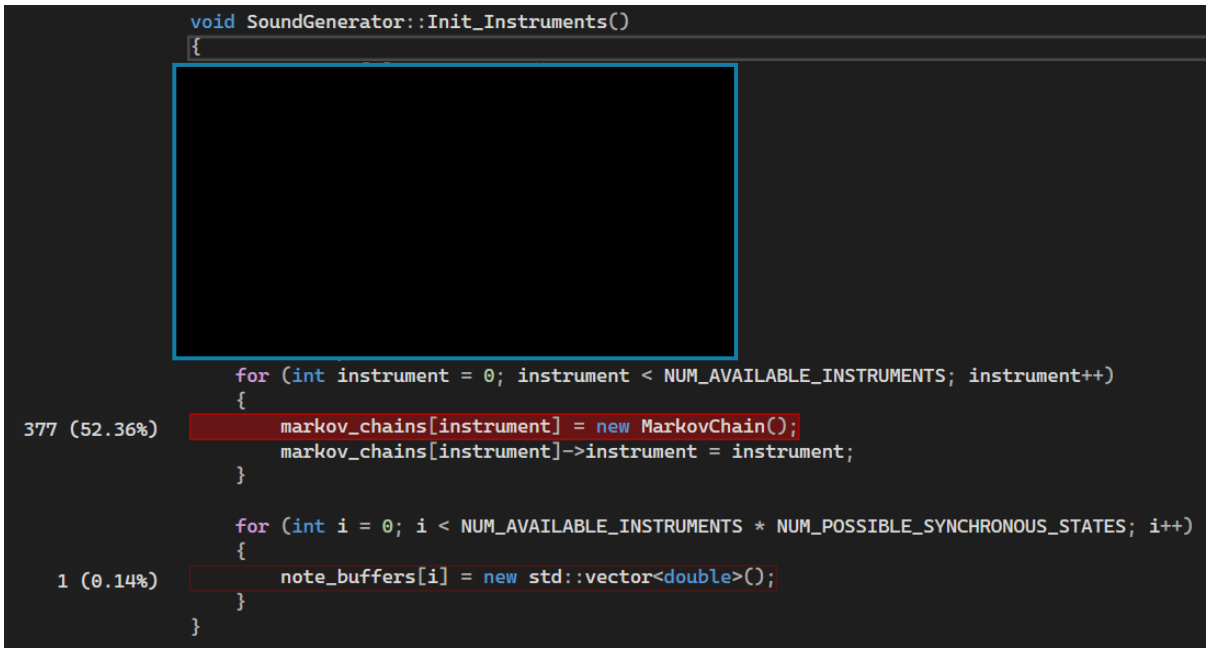

Figure 11 Visual Studio's performance profiler, hot path breakdown

Figure 11 shows that essentially all of the time spent in the function, 52.36% out of the total function time of 52.50%, seen in Figure 10, is used for allocating the memory for the Markov chain for each of the available instruments.

Figure 12.1 below shows the times taken to populate the Markov chains for a song with many instruments and events.

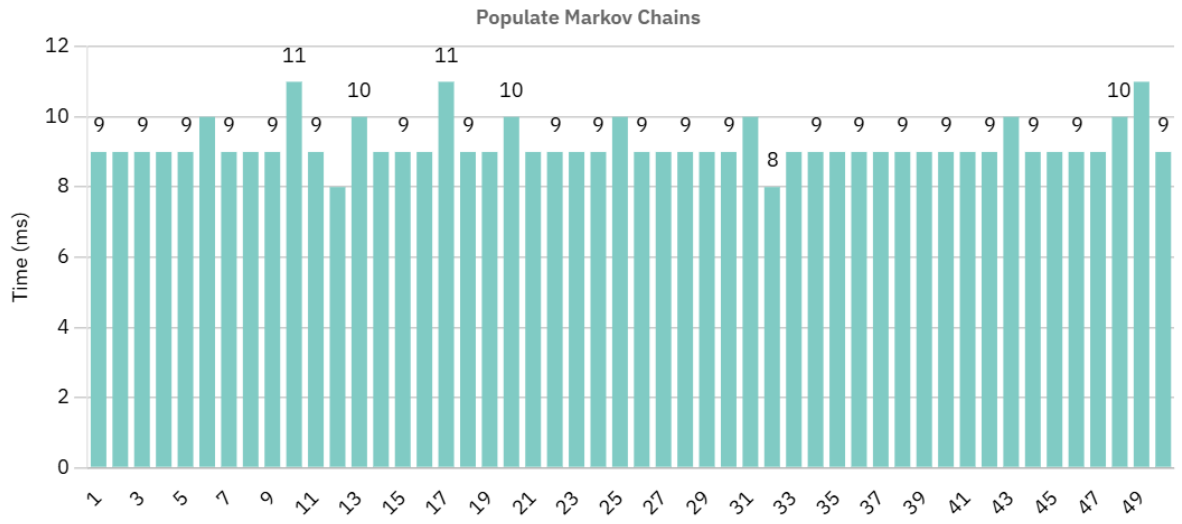

Figure 12.1 Graph showing 50 entries, time taken to populate Markov chains in milliseconds

Figure 12.1 shows that allocating the memory for the Markov chain is the expensive part of setting up the Markov chain, not the populating, with an average time to read of 9ms. Compared to the rest of the time not accounted for, which leaves 386ms of memory allocation. This means the time taken to populate the chain is 2.3% of the time to allocate, ~43 times faster.

Figure 12.2 below shows the time taken to generate the 100 events tested 50 times.

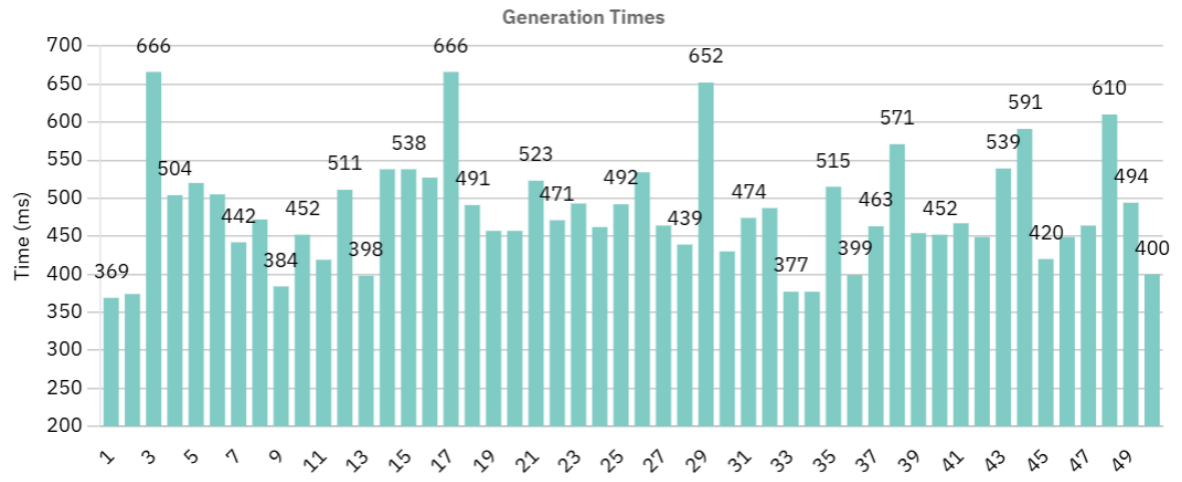

Figure 12.2 Graph showing 50 entries, time taken to generate audio in milliseconds

Figure 12.2 shows that the longest time taken was 666ms, and the shortest was 369ms, with an average of 482ms. This is largely consistent with a 0.2s difference between the worst and best times.

Figure 12.3 below shows the total program times tested 50 times.

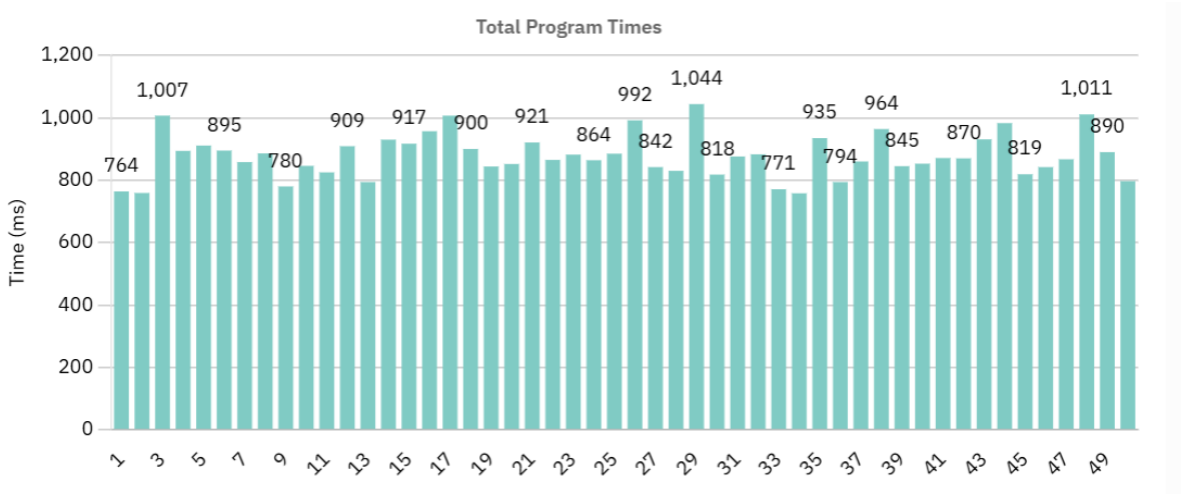

Figure 12.3 Graph showing 50 entries, total program lifetime in milliseconds

Figure 12.3 shows that the longest time taken was 1,044ms and the shortest time taken was 759ms, with an average time of 878ms. This is also largely consistent with a 0.3s difference between the worst and best times.

Figure 12.4 below shows the times taken to get all relevant events from the MIDI file to build the Markov chain.

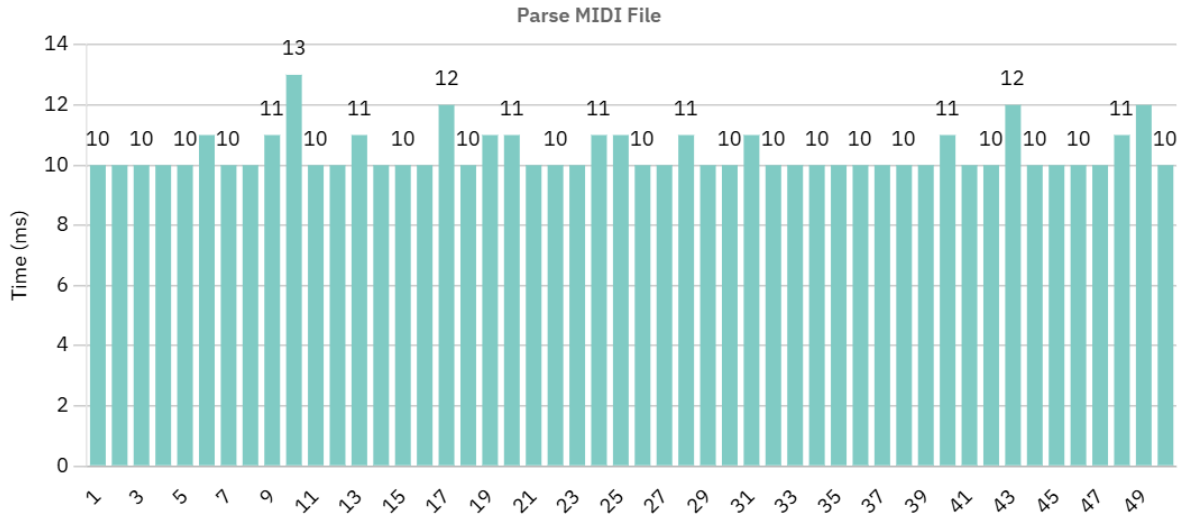

Figure 12.4 Graph showing 50 entries, time taken to parse a MIDI file

Figure 12.4 shows a fairly consistent average of 10ms.

Discussion and Analysis

Single Instrument Generation

The single-instrument generation using the Markov chain performed well, quite accurately matching the original waveform in both note waveforms and frequencies. This is likely due to the linearity of the piece used for generation, with long notes and no additional instruments to accompany these notes. The generation was consistent throughout the generated piece, with no noticeable auditory artefacts.

The major difference between the two pieces of audio, aside from the ordering of the notes, was the richness of frequencies played per note. This, however, is not a shortcoming of the generation method but rather a shortcoming in the implementation of the available instruments the generator could use to produce sounds. This is because these instruments weren't created with the intention of being life-like and realistic, but rather work by identifying the key frequencies that make up an instrument's sound and mimicking only those specific frequencies. The implementation also includes some frequency modulation synthesis, though it is limited to modulation by another sine wave, and accommodates ADSR envelopes to help shape the voice of the instruments. Because the instrument's quality was not highly valued in the creation of the project, the implementation lacks several tools commonly present in instrument synthesis, such as frequency filters.

With some more effort put into this area, the instruments could reach a similar standard to those used in the original pieces. A good way to do this programmatically is to use the Fourier series, as it is an effective method of replicating the sound of instruments (Alhodairy & Beitalmal, 2024). This suggests that this method of generation, when used in conjunction with linear songs and a more advanced implementation of some instruments, is an effective method of generating self-similar audio. This aligns well with the way procedural audio has been used in games previously. Much like in No Man's Sky, to generate a one-time sound effect for the creatures in the game (Mongeau, 2017), or in SPORE as a simplistic background sound that is unrepetitive to keep the audio from becoming stale (Jolly and McLeran, 2008). These are very linear pieces of audio that aren't the main focus of audio in the game, but act as an accompaniment to the main audio sources.

A drawback of this Markov chain method of generation was discovered upon listening to the generated pieces. As the chain can only ever make a transition between states based on the original song, after some time of listening to an extended piece of audio, components of the song that are linear, meaning they have few or no other possible transitions, can be identified and become stale regardless of the random nature of the chain. In the extreme case where an audio piece only has a single possible transition at any given point in the piece, e.g. C -> E -> G, the Markov chain can only ever produce the original sequence of notes unendingly. To the opposite effect, where single states have many possible transitions, a state explosion, the audio can lose its structured nature and generate a piece that seems completely random and nothing like the original piece.

Overall, this suggests that single-instrument Markov chains could be feasibly used in games for a piece of audio that is not meant to be the focus of the user, assuming that richer instruments than those currently available in the program are implemented alongside them.

Multiple Instrument Generation

The implementation of Markov chain generation for multiple-instrument audio pieces performed poorly. The generated audio looked very dissimilar to the original waveform in both amplitude and the shaping of individual notes at any given moment.

There were also significant auditory artefacts, such as sharp increases in amplitude. This is likely due to several instruments combining to produce a strong initial amplitude when first played. Additionally, when instruments with shorter lifespans, e.g., a drum hit, are phased out, the waveform drops significantly, creating a very dissonant sound. Another identifiable artefact is that quiet but audible noise was present in the generation. The audible noise can once again

be associated with the instrument generation, as this is a side effect of using a sustained saw wave to generate an instrument's notes.

The reason for long periods where only this sustained saw wave was present, however, is likely a limitation of the Markov chain itself. Each instrument can pick the next note to be played, but as these must be played simultaneously with other instruments, it becomes difficult to know when to choose the next note. The provided implementation works on a single note at a time for all instruments, meaning all instruments must wait until the longest note on an instrument at the shared start time finishes. This works when notes are all short and have consistent lifetimes across instruments, but in cases like these, where one instrument clearly keeps a long-sustained note while other instruments are meant to stop and start multiple times over this lifetime, it doesn't work well.

A proposed solution to this problem would be to keep generating the other instruments with consecutive notes, up to the length of the longest note at the starting point. However, this again runs into issues with Markov chains and their random nature. Choosing the notes at a single point in time ensures that these notes were meant to be played together, as they were a valid combination used in the song originally. Choosing additional notes that lack context for which notes are still playing can cause additional auditory artefacts and easily lose the very rhythmic nature that is crucial to most music pieces, as notes no longer occur at set intervals within the piece and in context with one another.

Similar to single-instrument generation, the spectrograph revealed that the generated audio was far less rich than that of the original piece and seemed to include frequencies far beyond the range of hearing, which is pointless data being stored. This is likely due to the implementation of specific instruments with a wide range of frequencies. In ordinary synthesis, the high-end frequencies that won't be heard would be filtered out, but this is not present in the instruments in the artefact.

Overall, without some core changes to the Markov chain model's inner workings, it seems unfeasible to expect that multi-instrument audio pieces would be effective as main or background music, given the significant auditory artefacts caused in the generation process.

Program performance

Temporal footprint

The minimum requirement for the program to be usable in a real-time application is at least 44,100 generated audio samples per second. The graph for generation time, Figure 12.2, shows that the program generates 100 events of varying lengths at an average time of 482ms. From this, around 75 seconds of audio were generated. That makes 3,307,500 samples in 482ms, and approximately 6,862,033 samples per second. This easily meets the requirement, at almost 156 times faster than the required speed. Even in the worst-case situation at 666ms, and for argument's sake, only 30 seconds of audio. This is still 1,323,000 samples in 666ms and 1,986,486 samples in a second, which is 45 times faster than required.

It cannot be guaranteed that the Markov chain has been pre-created or that its transition states aren't being updated in real time with information from the program. To that end, the time taken to both parse the desired file, which could be substituted for obtaining the relevant data to build the chain upon, and the building of the chain itself adds an extra 19ms. This brings the average time to complete the generation to 6,601,796 samples per second, or 150 times faster than

required, bringing the worst-case generation time to 1,931,386 samples per second or 43 times faster than required.

The caveat is that the memory allocation for the Markov chain is a significant part of the program's lifetime. This is a mitigatable cost; as once the memory has been allocated the first time, that memory can then be reused when remaking the Markov chain with new data. Assuming this is a one-time cost, with an average time for the entire program of 878ms, the initial generation produces 3,767,084 samples per second, which is still 71 times faster than required. Again, in the worst-case scenario, at 1,044ms for only 30 seconds of audio, there are 1,267,241 samples per second, which is still 28 times faster than required. This means, even assuming this allocation must be done every time, the generation is still more than quick enough to generate the necessary samples to ensure the audio output does not contain any auditory artefacts caused by insufficient samples.

This clearly shows that the shortcomings of a Markov chain are not its temporal components, as it more than satisfies the minimum requirements for generating audio in a real-time application. This also shows that some of the initially perceived slower methods of generation may be more feasible than initially thought. For example, although genetic algorithms need time to adapt to find the best possible next note, given the example here, there is plenty of spare time available to perform those adaptations.

Memory Footprint

There is no hard limit on how much memory is too much for the program to use, as that is up to the developer to decide this factor; however, current hardware often sits in the range of 8GB-32GB of RAM. Many games take somewhere between 4GB and 7GB as their minimum requirements, leaving the user with RAM for other processing the device may need to perform (Salas, 2024).

With a consistent usage of 1.7GB for the process due to many thousands of states a note can take, this seems an unfeasible amount of memory usage to be run alongside a game.

With some memory optimisations, such as custom allocation, storing the data in an external file to be read back in, or some form of compression, memory usage could be greatly reduced. Without testing these methods, though, it cannot be determined whether they would reduce the memory usage enough to make this a viable option. However, this may only be a shortcoming of Markov chains, as they store all the possible states a given state can transition to, along with the weightings of transitioning to them. Given that a state can have many instruments playing many notes simultaneously, this becomes an unreasonable amount of memory usage. Other methods, like neural networks, only need to store all possible state variations once, and so intrinsically have a lower memory cost than Markov chains.

Conclusion

Determining whether procedural audio methods can feasibly assist in game development has been investigated. Structuring noise to give it meaning is a challenging feat, even with an understanding of audio synthesis. Convincing a computer to generate those audio pieces is even more challenging. A computer program doesn't intuitively understand what sounds good like a human can; it can't generate anything that it has not been explicitly instructed that it can generate. Procedural methods attempt to solve this issue by making choices close enough to random that emergent outcomes occur. Fine-tuning these outcomes to a high-quality standard

requires significant processing, especially when the data being worked with is as complex as audio. If these methods can be used for high-quality audio production, it would help streamline the audio development process in games by providing a simpler workflow for developers (Böttcher, 2013). The fundamentals of how to create and structure noise to alchemise quality audio were explored, identifying the key components of waveforms and synthesis.

From there, possible procedural generation methods were explored. Finding the time budget for real-time generation to be more lenient than what was initially thought. Handcrafting each sample at 44,100 samples per second seems like a daunting task, but when a single note can encapsulate those samples and more, and knowing a computer can readily handle those operations, it becomes evident that higher complexity generation methods, or more complex sound synthesis methods, should be feasible for real-time applications. However, memory is a significant constraint that must be considered when choosing a generation method. This is because any point in the audio's lifetime can have several instruments, each playing several notes of varying durations and frequencies. The memory cost of quantising and storing all this information is quite high. This was an unforeseen shortcoming in the creation of the artefact, which should be considered in future research into the topic. This may make other possible generation methods more favourable, as many of them require less memory than the large graph of transitions from every possible state to every other state needed by a Markov chain. More complete tunes were mainly explored using the generation method, but this does not fully encapsulate all the possible types of audio in games. Audio, like sound effects, is a large part of games that the artefact was not designed for or did not attempt to generate. It is possible that Markov chains would be better suited to creating this style of fire-and-forget audio, perhaps even dividing a sound effect so deeply that it chooses individual amplitudes to create a unique yet similar-sounding sound effect. This would require more in-depth analysis than is provided to support this statement. Additionally, this paper was limited in that it did not explore the deep topic of musical theory, which is a key aspect of several possible procedural generation methods. With a knowledge of musical theory, other promising generation methods can be explored as to their feasibility in game development.

Analysis of the artefact shows that Markov chains, as a method of generation, are not a catch-all method for generating high-quality audio. Simplistic tunes only composed of a single instrument are well encapsulated by the generation boundaries of a Markov chain implementation. They produce similar quality results to hand-crafted audio and could serve as a feasible method of procedural generation to assist in the production of games. They may also be feasible to be used directly in real-time games if the high memory usage can be mitigated or accepted. Classical Markov chains pale at anything more complex than basic single-instrument tunes. This is because they become riddled with auditory artefacts, quickly falling apart under the complex nature that is audio due to a chain's rigid states. As a result, generating full, rich, and complex pieces of audio in any capacity, not only within the context of games, can be considered an unfeasible process to be completed using Markov chains. Improvements to a Markov chain's methods of generation, or other procedural generation methods entirely, may be found to be more capable of creating a fuller piece of audio. Because improvements to Markov chains and alternative generation methods have not been fully explored, the answer to whether procedural methods are a feasible possibility for generating audio in games cannot be definitively answered.

A large part of why the generated audio did not match the quality compared to original pieces of music was the artefact's implementation of its available instruments. Many of these were very

simple implementations of sound synthesis techniques, lacking some key synthesis tools such as filters. On top of this, the instruments were developed with only the key frequencies of an instrument in mind. In reality, instruments harness frequencies across the entire hearing spectrum to create their unique sounds. Additionally, instruments can be played in many ways, whereas the artefact only models each instrument as having a set and singular method for generating its sound.

This initial foray into procedural methods does show consistency with the limited number of games that have tried implementing procedural methods. Whereby, the generated pieces in these games are quite simplistic in nature and are used to add to the ensemble of handmade audio already in the game, rather than attempting to take the limelight. The investigation also shows consistency with previously identifiable issues with Markov chains, such as the high memory cost and attempting to use them to model complex or non-linear data types (Hassani & Wuryandari, 2016).

With the current knowledge of the investigated procedural methods, the audio director for No Man's Sky's quote that procedural audio generation "only makes sense if it solves a problem for you that would otherwise be difficult to resolve using conventional sound design" (Mongeau, 2017), holds true. Generative audio does not currently match the quality of handmade audio. It should not be used as the focal point of audio in a game, unless there is no alternative solution, and even then, the generated audio should not be expected to have the quality or richness of handmade audio.

Recommendations

The logical jumping-off points from this paper would be to pick up where the artefact fell short, such as implementing better synthesis tools for instrument creation.

Additional areas of research should, given the efficacy of the temporal results, be dedicated to more computationally expensive or complex procedural methods, such as genetic algorithms, neural networks or improvements to the Markov chain method, as these were initially dismissed due to their high computational cost.

Context-Aware Markov Chains

Simplistic Markov chains had numerous limitations, preventing the generation of more complex and high-quality audio. Methods to improve and adapt the Markov chain model may prove that success can still be found in this generative technique. However, reducing the runtime memory usage of Markov chains is an area that should be investigated before improving the generation method.

A proposed approach to circumvent the issues with multiple-instrument generation and state explosion in both single-instrument and multi-instrument chains is to use a context-aware Markov chain (Bar et al., 2023). While a traditional Markov chain chooses its next state independently of its previous state, a context-aware Markov chain can consider other notes in the chain to inform its transitions. This means that the transitions must follow the context provided by the context window (how many previous states the current transition is influenced by). For example, in the case of a state explosion, simply taking the previous note into account would greatly reduce the possible transitions from this state. To keep things more random, this could not be a hard context lock based on the context window, and instead, only increase the weight of the transitions that would normally follow from the context window. With layered

instruments, the context window informs the current context of notes from a different instrument that are still being played, and from there, it can influence the transition to play only notes found while the other instrument's note is still being held. This means that several additional instruments can make a variable number of informed and well-sounding musical transitions during the duration of an overarching instrument's singular long note. The problem with this is the scaling complexity. Each time the context window is increased, an additional layer of computation is required to find states that match the new context window, as well as each state now needing to store which states came before it up to the width of the context window. With a window that is too large or a context selection that is too strict, the problem once again arises that the generated piece can only follow the original song directly.

Bibliography

Affenzeller, M. (2009) *Genetic Algorithms and Genetic Programming*. Boca Raton: Taylor & Francis.

Aguilar, L.M.A. et al. (2010) The principle of superposition for waves, arXiv.org. Available at: https://arxiv.org/abs/1005.3237 (Accessed: 21 November 2025).

Alhodairy, S. and Beitalmal, A. (2024) 'The applications of Fourier series harmonics in musical tones', Journal of Pure & Applied Sciences, 23(2), pp. 154–161. Available at: doi:10.51984/jopas.v23i2.3590 (Accessed: 5 January 2026).

Baldur's Gate 3 (2023) *Steam*. Available at: https://store.steampowered.com/app/1086940/Baldurs_Gate_3/ (Accessed: 16 February 2026).

Bar, A., Shapira, B. and Rokach, L. (2023) "Context Aware Markov Chains Models," Knowledge-based systems, 282. Available at: https://doi.org/10.1016/j.knosys.2023.111083.

Böttcher, N. (2013) 'Current problems and future possibilities of procedural audio in computer games', Journal of Gaming & Virtual Worlds, 5(3), pp. 215–234. doi:10.1386/jgvw.5.3.215_1.

Burrus, C.S. (2012) Fast Fourier Transforms. Place of publication not identified: OpenStax CNX.

Ciesla, R. (2022) Sound and Music for Games : The Basics of Digital Audio for Video Games. 1st ed. 2022. Berkeley, CA: Apress. Available at: https://doi.org/10.1007/978-1-4842-8661-6.

Creasey, D. (2017) "Subtractive Synthesis," in *Audio Processes*. Routledge, pp. 523–544. Available at: https://doi.org/10.4324/9781315657813-27.

Doumler, T 2021. Real-time Programming with the C++ Standard Library [PowerPoint presentation] CppCon, 24 October, Colorado.

Elkashef, O. (2023) (PDF) conditions of constructive and destructive interference of sound waves, Research Gate. Available at: https://www.researchgate.net/publication/376554576_Conditions_of_constructive_and_destructive_interference_of_sound_waves (Accessed: 21 November 2025).

Epic Games, 2019. *Unreal Engine*, Available at: https://www.unrealengine.com.

Hassani, Z. and Wuryandari, A.I. (2016) "Music generator with Markov Chain: A case study with Beatme Touchdown," in *IEEE International Conference on System Engineering and Technology (Online)*. IEEE, pp. 179–183. Available at: https://doi.org/10.1109/ICSEngT.2016.7849646.

Holtzman, S.R. (1981) "Using Generative Grammars for Music Composition," *Computer music journal*, 5(1), pp. 51–64. Available at: https://doi.org/10.2307/3679694.

Horowitz, S. and Looney, S. (2014) The essential guide to game audio : the theory and practice of sound for games. 1st edition. Burlington, Mass: Focal Press. Available at: https://doi.org/10.4324/9781315886794.

Howell, K. (2016) Principles of Fourier Analysis, 2nd Edition. 2nd edition. CRC Press.
Jolly, K. and McLeran, A. (2008) Procedural Music in SPORE, GDC Vault. Available at: https://gdcvault.com/play/323/Procedural-Music-in (Accessed: 3 January 2026).

Ibe, O.C. (2013) *Markov processes for stochastic modeling*. 2nd ed. Amsterdam, Netherlands: Elsevier.

Jan, S. (2022) *Music in Evolution and Evolution in Music*. 1st ed. Cambridge, UK: Open Book Publishers.

Kadam, A.A. *et al.* (2023) "A Compact and Ultra-low Power Low Pass Filter based on Band-to-Band Tunneling Effect," *IEEE transactions on circuits and systems. II, Express briefs*, 70(9), pp. 1–1. Available at: https://doi.org/10.1109/TCSII.2023.3273621.

Khodzhaev, Z. (2022) A Practical Guide to Spectrogram Analysis for Audio Signal Processing, arXiv.org. Available at: https://www.researchgate.net/publication/339299002_Spectrogram_-_Practical_Guide (Accessed: 10 February 2026).

Liu, J. et al. (2021) 'Deep learning for procedural content generation,' Neural computing & applications, 33(1), pp. 19–37. Available at: https://doi.org/10.1007/s00521-020-05383-8.

Newell, P. (2017) *Recording Studio Design, 4th Edition*. 4th edition. Routledge.

Nichols, K. (2011) 3 Sound Wave Terms. [Digital Image] Available at: https://www.flickr.com/photos/wessexarchaeology/6465253951/in/photostream (Accessed: 20 November 2025).

Nierhaus, G. (2011) "Chaos and self-similarity," *Нелинейная динамика*, pp. 153–175. Available at: https://doi.org/10.20537/nd1101010.

Madhav, S. (2018) Game programming in C++ : creating 3D games. 1st edition. Boston: Addison-Wesley.

Massaron, L. (2019) Deep learning. 1st edition. Hoboken, N.J: J. Wiley.

Mazzoni, D. and Dannenberg, R. (2000) Audacity. Available at: https://www.audacityteam.org/ (Accessed: 12 February 2026).

Miranda, E.R. (2001) *Composing music with computers*. 1st ed. Oxford ; Focal Press.

Miranda, E.R. and Shaji, H. (2023) "Generative Music with Partitioned Quantum Cellular Automata," *Applied sciences*, 13(4), p. 2401. Available at: https://doi.org/10.3390/app13042401.

Mongeau, A.-S. (2017) Behind the sound of 'no man's sky': A Q&A with paul weir on procedural audio, A Sound Effect. Available at: https://www.asoundeffect.com/no-mans-sky-sound-procedural-audio/ (Accessed: 11 February 2026).

Music note fundamental frequencies (2025) Songstuff. Available at: https://www.songstuff.com/recording/article/music-fundamental-frequencies (Accessed: 20 November 2025).

Oxenham, A.J. (2017) How we hear: The perception and neural coding of sound | annual reviews, Annual Reviews. Available at: https://www.annualreviews.org/content/journals/10.1146/annurev-psych-122216-011635 (Accessed: 21 November 2025).

Pirkle, W.C (2021) *Designing Software Synthesizer Plugins in C++, 2nd Edition*. Focal Press.

PlayStation®5 console - 1 TB (2020) Sony. Available at: https://direct.playstation.com/en-us/buy-consoles/playstation5-console-1-tb (Accessed: 16 February 2026).

Roads, C. and Wieneke, P. (1979) Grammars as representations for Music, Computer Music Journal, 3(1), pp. 48–55. doi:10.2307/3679756.

Roads, C. (2023) The computer music tutorial. Second edition. Cambridge, Massachusetts: The MIT Press.

Roginska, A. (2018) *Immersive sound : the art and science of binaural and multi-channel audio*. 1st edition. Edited by A. Roginska and P. Geluso. New York: Routledge.

Roma, G. (2023) "Agent-Based Music Live Coding: Sonic adventures in 2D," *Organised sound : an international journal of music technology*, 28(2), pp. 231–240. Available at: https://doi.org/10.1017/S1355771823000274.

Rose, J. (2015) *Producing great sound for film and video : expert tips from preproduction to final mix*. 4th ed. Burlington, Massachusetts ; Focal Press.

Rosenfeld, D (2011) 'Minecraft', Minecraft - Volume Alpha. [Spotify] C418.

Russ, M (2012) *Sound Synthesis and Sampling, 3rd Edition*. Routledge.

Salas, M. (2024) Ram usage in games (53 games tested), Laptop Study - Find the cheapest & best laptop. Available at: https://laptopstudy.com/ram-usage-games/ (Accessed: 13 February 2026).

Sapp, S.S. (2024) midifile. Github. Available at: https://github.com/craigsapp/midifile (Accessed: 2 February 2026).

Scott, C. (2011) *Pure Data Module , VTechWorks repository*. Available at: https://vtechworks.lib.vt.edu/server/api/core/bitstreams/b4dbd0bf-c126-4d13-a055-0fe88019168b/content (Accessed: 21 November 2026).

Shannon, C.E. (1949) Communication in the Presence of Noise. Proceedings of the IRE, 37(1), pp.10–21. Available at: https://doi.org/10.1109/jrproc.1949.232969.

Talbot-Smith, M. (2002) *Sound engineering explained*. 2nd ed. Oxford ; Focal Press. Available at: https://doi.org/10.4324/9780080498171.

The Open University. Sound for music technology: An introduction, Open Learning. Available at: https://www.open.edu/openlearn/science-maths-technology/engineering-technology/sound-music-technology-an-introduction/content-section-6.1 (Accessed: 20 November 2025).

Towell, G. (2022) The Physics of Music: Waves, Beats & Frequencies, Sciencing. Available at: https://www.sciencing.com/the-physics-of-music-waves-beats-frequencies-13722354/ (Accessed: 20 November 2025).

Tzimeas, D. and Mangina, E. (2009) "Dynamic Techniques for Genetic Algorithm-Based Music Systems," *Computer music journal*, 33(3), pp. 45–60. Available at: https://doi.org/10.1162/comj.2009.33.3.45.

Uematsu, N (1991) 'Battle 1', FINAL FANTASY IV Original Soundtrack. [Spotify] Square Enix.

Vasudevan , H., Zolghadri , S. and Makarem, M.A. (2023) 'Introduction to oil, gas, and petrochemical industries: importance to the current world', in Crises in Oil, Gas and Petrochemical Industries Volume 1: Disasters and Environmental Challenges. Amsterdam, Netherlands: Elsevier, pp. 25–46.

Visual Studio 2022 (2021) Microsoft.

Wolfe, J. (2005) *Note names, MIDI numbers and frequencies*, *Basics in Music Acoustics*. Available at: https://newt.phys.unsw.edu.au/jw/notes.html (Accessed: 16 February 2026).