Harmonising the Senses: An Audiovisual Tool for Interactive Fourier Synthesis

Ben Evans

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

This report documents the design and implementation of an interactive Fourier synthesis tool. This tool allows the user to change the amplitude, fundamental frequency and harmonic composition of waves, and gives audible and visual feedback as these parameters are changed. Crucially, it not only allows the user to also change the relative amplitudes of each harmonic, but also experiment with the phase differences between these harmonics in a pseudo-continuous fashion. It appears to be the only tool of its kind that gives the user this level of control over harmonic phase difference with audible feedback.

The report also provides the relevant background information for the tool, including the physics of sound waves, the harmonic spectrum, and how different timbres of sound are created digitally. It formalises this through a description of Fourier theory, from analysis to synthesis, discusses relevant related work, and provides an overview of the Unity game engine, which is used for the implementation of the tool. The report also provides an evaluation of the tool's strengths and weaknesses, supported by a user evaluation study and comparison to related work, and potential improvements and extensions to be implemented in future builds. The report concludes by describing some of the phenomena that can be observed by using the tool and how to recreate these, and also other areas to be explored within this field of study.

Research Ethics Approval

This project obtained approval from the Informatics Research Ethics committee.
Ethics application number: 184640
Date when approval was obtained: 2023-03-07
The participants' information sheet and consent form are included at appendices A and B, respectively.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Ben Evans*)

Acknowledgements

Thanks to my dissertation supervisor, Dr. John Longley, who has guided me through the creation of this project, and facilitated my own research and creativity throughout.

Thanks also to my parents, who have always supported by academic work and are a constant source of motivation.

Table of Contents

Chapter 1

Introduction

A sound is a wave of energy that propagates through a medium. We hear sounds when these waves vibrate the particles in the air, varying the pressure, and these vibrations reach our ears. Our outer ears detect these waves, and pass them onto our inner ears, which convert them into electrical signals which can be processed and interpreted by our brain. Sound has been used to create music and audio experiences for thousands of years, and as technology has advanced, our ability to analyse, synthesise and manipulate sound into different forms has expanded greatly. This is primarily due to our increased understanding of harmonics and overtones. The ability to manipulate and synthesise sounds, by customising the harmonic configurations, allows for the creation of a wide range of different timbres and textures to be produced. However, limited control is often provided regarding the phase difference of each harmonic. While adjusting the phase of a single-frequency wave does not have any audible effect, and indeed the human ear is much less sensitive to phase differences within sound than it is to relative amplitude differences, we will discover that there most certainly is an effect to be heard (Schroeder, 1975). The ability to manipulate these phase values in a pseudo-continuous fashion, as well as the relative amplitudes of each harmonic, allows for an even greater range of creativity when developing a sound, and also opportunity for experimentation and analysis of the effect of different phase configurations on the shape and sound of the resulting wave.

This report documents the design and implementation of an interactive Fourier synthesis tool that provides precisely this level of control over harmonic phase difference, and that appears to be the first to do so in this way. The tool also allows different numbers of overtones to be used, and continuous control over the relative harmonic amplitudes, fundamental frequency and total amplitude of a wave. The tool provides visual and audible feedback as these parameters are changed, allowing the user to explore the effects of parameter manipulation in real-time. The tool was implemented using the Unity game engine, and the report provides an overview of this technology and its suitability for this type of application. The report also provides an evaluation of the tool's strengths and weaknesses, supported by a user evaluation study and comparison to related work. The report begins by providing relevant background information on the physics of sound waves and how these are represented digitally. It then introduces the harmonic spectrum, and how it is used to create different timbres of sound in an analogue and digital context. This is formalized through a description of Fourier theory, from analysis to synthesis. The report then discusses relevant related work in the field of sound synthesis, including previous efforts to provide control over harmonic phase difference. Finally, the report concludes by describing some of the phenomena that can be observed by using the tool and how to recreate these. It also identifies potential areas for future research and development, including extensions to the tool's functionality and exploring the use of this level of control over harmonic phase in other areas of sound synthesis.

Chapter 2

Background

Sound is something with which we are all very familiar, and experience daily, but what is sound on a physical level? How do we measure and analyse sounds, and what can be done with the information that we gain from this analysis? This chapter aims to describe sound and its various components from a mathematical and analytical perspective, and to explain how adjusting these components translates to what we hear as an observer. First, I will give some background on what sound is on a basic level.

2.1 Reintroducing Sound

Sound waves travel and propagate as *longitudinal* waves, where the particles of the medium through which it is travelling oscillate in the same plane as the wave direction. They can also be displayed in a transverse form, however. This is because while sound may travel longitudinally as a function of *distance*, we can also represent it as a function of *time*. Imagine that rather than tracking the displacement of multiple particles in the direction of the wave, we instead track the displacement of an arbitrary *single* particle in that same direction, and place this on the vertical axis as opposed to the horizontal. By tracking this individual particle's displacement over time, we receive the transverse wave that corresponds to the original longitudinal sound wave. This graphical view of each particle's displacement over time due to a sound wave is called the sound's *waveform*. Analysing a sound's transverse waveform is particularly useful to obtain information about its qualities and parameters, rather than its spatial behaviour and trajectory, and so this is the primary way that sound will be visualised during this project.

Note that we consider the 'source' point shown in Figure 2.1 to have a displacement value of 0. Values above this line are positive and those below the line are negative. The positive portion of a wave's repeating waveform is called the *compression*, and the negative portion is called the *rarefaction*. The points where the displacement values are 0 (i.e. at the source value) are called the *nodes* of the wave, and the peaks and troughs of the wave are the wave's *anti-nodes*. These terms are particularly helpful to describe the shape of a waveform textually.

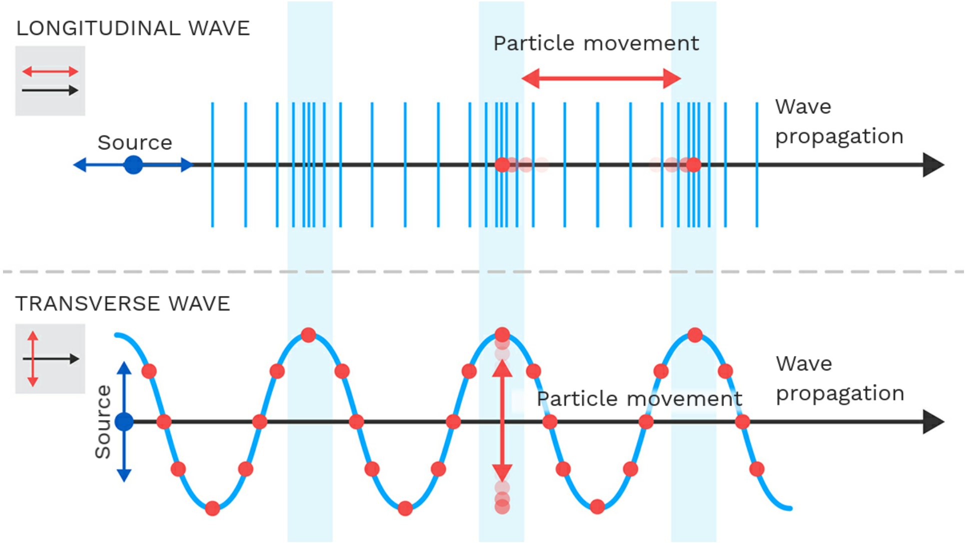


Figure 2.1: Diagram showing the difference between longitudinal and transverse waves (Labster Theory, 2022)

2.1.1 Wave Parameters

Although naturally occurring sounds are never completely smooth or perfectly repetitive, to help define the parameters of sound waves, let us consider a periodic sinusoidal wave (or *sine wave*) as our model sound wave. While sine waves are a mathematical function that can exclusively be produced digitally, we will discover that they are actually a key building block to create more complex and realistic sounds.

2.1.1.1 Frequency and Amplitude

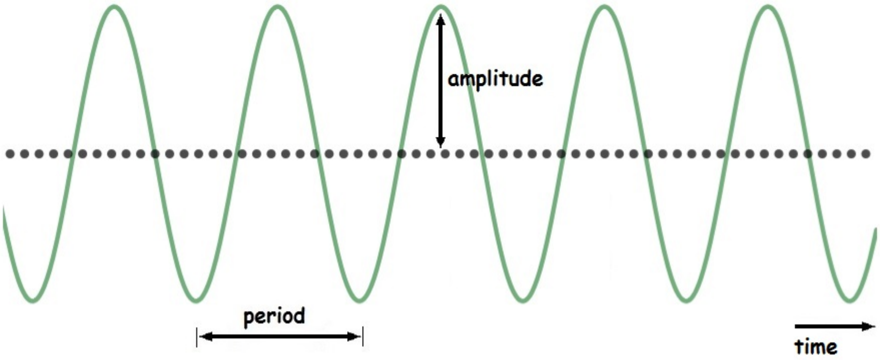


Figure 2.2: Diagram showing a sinusoidal waveform, with definitions of the period and amplitude (Abbott, 2020)

We call the time taken for a wave to go through one full cycle of its repeating waveform the wave's *period*, usually denoted as T. The wave's *frequency* is then defined as $1/T$Hz, or intuitively the number of periods that occur within one second. A wave with a constant frequency over a period is time is said to be *periodic* over that time frame. The *amplitude* of a wave is then defined as the absolute value of the maximum displacement of each particle (i.e. the displacement at the anti-nodes). These two basic parameters of a wave, frequency and amplitude, correspond to the perceived pitch (high or low) and volume (loud or quiet) of a sound, respectively.

2.1.1.2 Phase

The other main parameter we will consider with regards to an individual sound wave is phase. Phase is a cycling parameter of periodic waves that intuitively represents the ‘progress’ through a wave’s period, starting from 0 and cycling back to 0 at the completion of the period. Phase is different from the other two parameters discussed so far, because even when describing a periodic wave, the phase is non-constant. It is a many-to-one mapping of time values to displacement values, where $\phi(t) = \phi(t+T)$ for any t. The point at which a period *starts*, or where $\phi(t) = 0$, is in general an arbitrary point. For a sine wave, however, it is well defined as the node before the compression of the period.

When describing a sine wave, the *initial phase*, represented by the symbol φ, is particularly important. This is the phase value at the origin of the wave, and can be thought of as the offset from the defined start of the period. Explicitly, $\phi(0) = \varphi$. The initial phase is subtracted from the input value t of a sine function to obtain the displacement value at a specific time t.

Phase values are measured in angular units, such as degrees or radians. For consistency throughout this project, I will refer to phase values in radians except where stated otherwise.

The term ‘phase’ is used quite interchangeably in academic literature to refer to initial phase, absolute phase and phase difference, and so it can often be difficult to follow exactly to what is being referred. So, I will try to be as explicit as possible at each point throughout this report, using the symbols above.

2.1.1.3 Generic Sound Wave Formula

While our phase values are measured in radians, we want to measure time in units of one second, rather than 2π seconds, and equally we want our frequency to represent the number of periods per second, rather than the number of radians per second. So, at any time t, the displacement values for a sine wave with a frequency of f, amplitude of 1, and $\varphi = 0$ will have the formula $sin(2\pi ft)$, not simply $sin(ft)$. We can use linear [0-1] values as our phase range if we also multiply φ by 2π, and this is easier to handle when developing software regarding phase as it reduces the number of times an irrational number needs to be approximated and used in calculations. So, our generic formula to obtain the displacement value at time t for a sine wave with frequency f, amplitude A, and initial phase φ (in linear values) is as follows:

$$displacement = A \cdot sin(2\pi(ft - \varphi))$$

2.1.2 Superposition and Interference

Until this point we have only considered the case of a single sound, but when multiple sounds, and thus multiple waves are involved, how do they interact, and how does this affect what we hear?

When two waves meet at a point in space, they interfere with one another, creating a new resultant wave called their *superposition*. The principle of superposition states that the displacement at the point where two or more waves meet is equal to the sum of the displacements of the individual waves (reference). It is easiest to first illustrate this effect with the example of two identical waves. Let's use our basic sine wave for now.

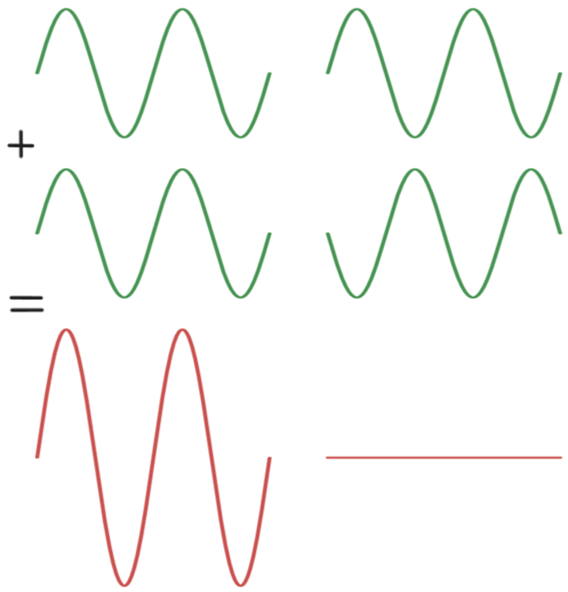


Figure 2.3: Total constructive and total destructive interference

As seen in Figure 2.3, when two peaks of a wave meet at the same point, their addition creates a total displacement value that is larger than each individual sound, and similarly when two troughs line up, the same effect is reached in the negative direction. When a peak of one wave and the trough of another meet, however, the displacement level of the two waves cancel out, and the result is a displacement of 0. The superposition of two waves causing a greater resultant displacement is called *constructive* interference, and that which results in a displacement of 0 is called *destructive* interference (reference).

The importance of phase in wave interference is best explained by considering the phase *difference* between two waves. On the left of Figure 2.3, the two waves have the same initial phase value, and so are considered to be *in phase* with one another, and the superposition wave has twice the amplitude at any point t. All interference is constructive. On the right, the two waves have a phase difference of π radians, meaning they are in *anti-phase*. All the peaks of one wave line up with a trough in the other, and the result is a displacement of 0 at any point in time. All interference is destructive. This phenomenon is intentionally used in active noise-cancellation systems, where incoming sounds are recorded, and a speaker is programmed to produce waves that are intentionally in anti-phase with the incoming waves, leading to destructive interference and thus cancellation of the unwanted sound.

2.1.3 The Digital Representation of Sound

Sound data is recorded and stored as a series of discrete displacement values, called samples. Crucially, because a microphone is essentially a single point, the collected samples do not represent each individual incoming waveform, and instead describe the resulting *superposition* of these waves.

The number of samples contained within one second of playback is known as the *sampling rate*. A higher sampling rate more accurately represents the continuous nature of the input sound, but also results in larger file sizes. If the sampling rate is too low, then the input analogue signal will not be accurately represented digitally. This does not just affect sound quality, but an insufficient sampling rate can cause aliasing, where a frequency is misrepresented as a different frequency within the audio, and some sound information can be entirely lost in the resultant data.

To demonstrate this, imagine that we are recording a constant sine wave with a frequency of 100Hz. If we were to sample at a rate of 100Hz, then every sample collected would have exactly the same value, as it would collect the displacement value at the same phase point in the wave. The recording, when played back, would therefore be completely silent. If we sample at 200Hz, however, then we will receive displacement values that alternate between negative and positive (except in the practically negligible case where our samples have a displacement value of 0). By receiving these alternating values, we guarantee that the frequency will be accurately conveyed, although the same cannot be said for the amplitude. Therefore, as formalised in the Nyquist-Shannon theorem (reference), the sampling rate must be *at least* twice that of the highest frequency present in the signal to guarantee that all present frequencies are accounted for. This minimum required rate is called the *Nyquist frequency*. The human hearing range roughly lies between 20 and 20,000Hz. As a result, the standard sampling rates are above 40,000Hz, with 44,100Hz and 48,000Hz being among the most common.

These considerations regarding sampling occur not only when measuring any form of wave, but also when creating synthetic audio. Digitally synthesised sound is stored in exactly the same way and so is subject the the same restrictions as recorded sound.

2.2 Timbre

Frequency, amplitude and phase are essential to describe sounds and their interactions. If these were the *only* relevant qualities of sound, however, then two different instruments playing at the same pitch and volume would sound exactly the same. We now need to answer the question of why this is not the case. Why does a trumpet produce a raspy sound, and why does a clarinet have a more mellow sound? Formally, what do the words ‘raspy’ and ‘mellow’ even mean, in a physical sense, when referring to sound? We use these words to describe a sound’s *timbre*. According to the American National Standards Institute: ‘Timbre is that attribute of auditory sensation in terms of which a listener can judge two sounds similarly presented and having the same loudness and pitch as being dissimilar' (ANSI, 1960). The *shape* of a sound's periodic waveform is what determines this. It is an all-important quality of sound waves, but it is arguably the most difficult to quantify and define (Howard and Angus, 2017).

2.2.1 Harmonics

The waveforms of most non-digital sounds that we hear are usually quite complex. They are especially difficult to reproduce digitally, not just because of the shape of their waveform, but also their tendency to vary over time (Howard and Angus, 2017). Figure 2.4 shows the waveform of a clarinet tone as an example.

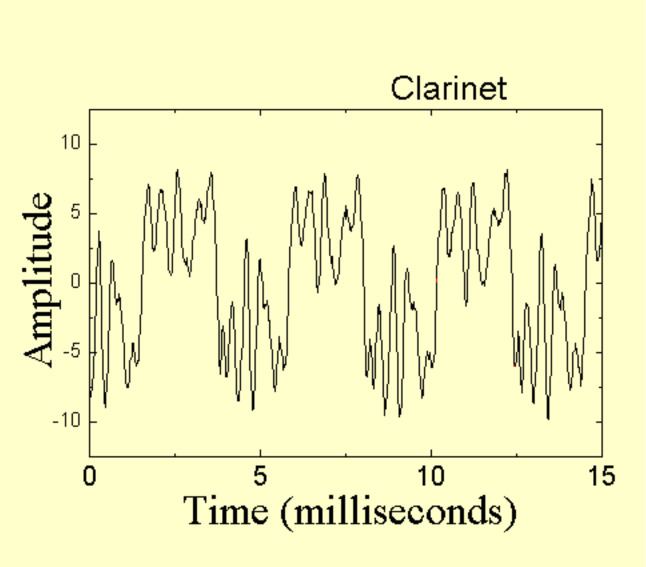


Figure 2.4: Clarinet waveform (Suits, 2023a)

Observing this waveform, one might notice that there are several peaks, and indeed there appear to be at least two different frequencies contained within this superposition. Only one note is playing played in this recording, however, and (Suits, 2023b) includes the sound that produced this waveform if you need convincing. These extra frequencies are in fact *harmonics*.

The harmonics of a frequency are the integer multiples of that frequency. Each harmonic has a harmonic number, which is simply the number by which the frequency is multiplied to obtain the frequency of the harmonic. The harmonics with a harmonic number not equal to 1 are called *overtones*, and the presence, absence and relative strength of these overtones are 'the one and only factor in sound production which conditions timbre' (Scholes, 1970). This makes sense as, intuitively, if we have a sound at a constant pitch, then the only frequencies which we *could* add to that sound that would not affect the rate at which its waveforms repeat (i.e. its base frequency), would be the integer multiples of that frequency. Psychologically, we perceive the resulting pitch as that of the lowest frequency, and the configuration of the overtones as the resulting timbre, and thus this base frequency is called the *fundamental frequency* of the sound (Howard and Angus, 2017).

We can analyse the frequency spectrum of a sound using a technique called the Fourier Transform. The Fourier Transform is a frequency decomposition transform, analogous to the process of un-mixing a bucket of paint, or separating a smoothie into its individual fruit juice components. It is a fascinating mathematical function, but a full explanation is outside of the scope of this project. Essentially, the Fourier Transform can take the superposition of a number of waves as input, and output the individual frequencies of the wave, scaled by their relative amplitudes. Part of the Fourier Transform of the clarinet sound above can be seen in Figure 2.5. Interestingly, the odd harmonics are much more prevalent in the sound than the even harmonics. This is in fact one of the main reasons that the clarinet has such an easily-identifiable timbre. As might be expected, the fundamental frequency is the most prominent harmonic.

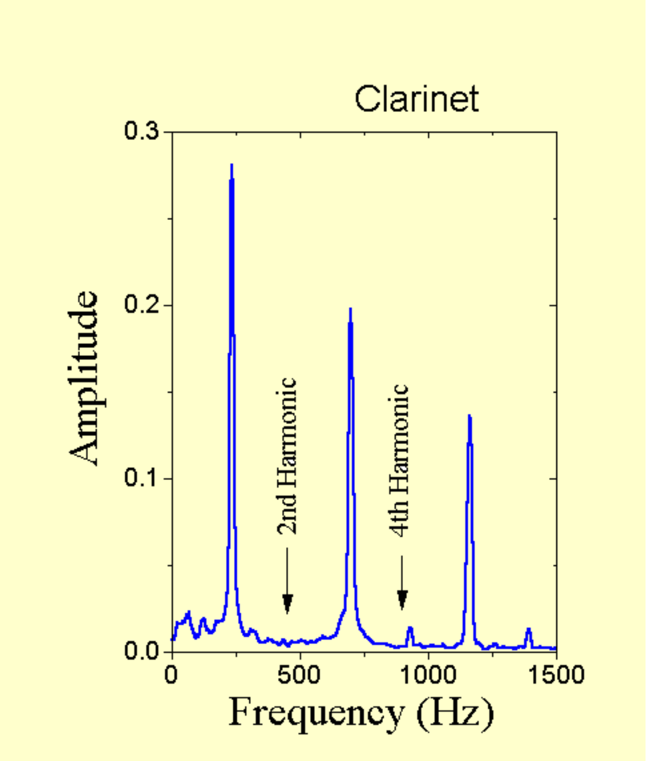


Figure 2.5: Clarinet Fourier Transform (Suits, 2023a)

2.2.2 Digital Waves

The process of creating digital sound is known as direct digital synthesis. The basic digital wave types are simple to create due to the simplicity of their functions. The common basic wave types are the sine wave, triangle wave, square wave and saw wave, and can be seen in Figure 2.6.

Saw waves and square waves have a particularly rich harmonic spectrum, giving them a bright sound, whereas triangle waves have a much duller timbre, and in fact only contain odd harmonics, like the clarinet. Sine waves do not have any overtones, and thus have the dullest of these timbres. A square wave is a type of pulse wave, which allows the maximum and minimum amplitudes to have arbitrary durations. Pulse waves

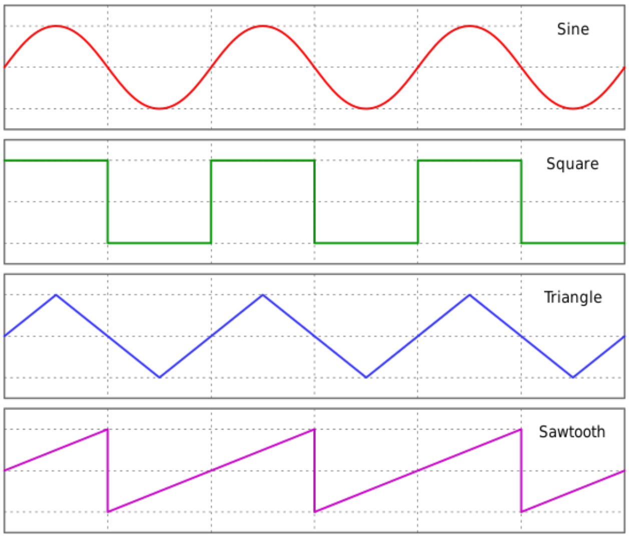


Figure 2.6: Basic Wave Types (Wikipedia contributors, 2023)

are particularly interesting, as their timbre changes depending on the defined length of each pulse. The shorter the pulse, the brighter the timbre.

Unlike in Figure 2.4, it is very difficult to see how these different timbres can be produced by the presence of overtones, as they are composed of straight lines. However, a frequency analysis of the sounds reveals that these harmonics *are* present within the waves. By combining sine waves of the harmonic frequencies in a certain way, these waves can be approximated, and this approximation technique is known as Fourier Synthesis.

2.3 Fourier Synthesis

2.3.1 Fourier Theory

Joseph Fourier arrived at the idea of the Fourier Transform only after developing the *Fourier Theory*, an idea which actually stemmed from his analysis of heat diffusion, unrelated to sound. He realised that he could get a very accurate approximation of most waveforms by adding together sine waves of different frequencies, phases and amplitudes. His theory then followed that if he combined enough sine waves together, he could precisely model *any* waveform (Howard and Angus, 2017). Unfortunately, this theory only holds under certain restrictions due to factors such as Gibbs' phenomenon (2.3.3), as formalised in Dirichliet's editions to Fourier's original paper (Dirichlet, 1829), but it *can* give a very good approximation of any wave. The waves described in 2.2.2 require an infinite summation to get the best approximation, but a good approximation can still be reached with only a few terms. These infinite summations are known as Fourier series, the study of which is called harmonic analysis.

2.3.2 Amplitude-phase Fourier Series

A Fourier series is an expansion of a periodic function in terms of an infinite sum of sine and cosine functions (Weisstein, 2023). It can, however, be represented by only using sine waves, or only using cosine waves. This is because sine and cosine waves have the same waveform, but a phase difference of π radians. The series can therefore be represented in the following form for the context of this tool:

$$s(t) \approx \sum_{n=1}^{\infty} A_n \cdot sin(2\pi(nft - \varphi_n))$$

where,

- t is a time value
- $s(t)$ is a periodic function
- n is a harmonic number
- A_n is the relative amplitude value for each harmonic,
- f is a frequency
- φ_n is the initial phase value for each harmonic, in linear values

A_n and φ_n can then be represented as functions of n for some waveforms, allowing this to be implemented in an algorithm. If the wave being approximated does not contain any even harmonics, for example, then this series is only summed over the values of n where n is an odd number.

2.3.3 Gibbs' Phenomenon

Figure 2.7 shows an approximation of a square wave using a Fourier series with 10, 20 and 30 terms used. As we can see, the deviation from the target function decreases as the number of terms used increases. However, we can notice that the approximation is much more accurate in the centre of each pulse, as opposed to at the edges, where the approximation overshoots its target, and both the x and y values of the peaks at these points are offset from the desired values. The height of the approximated peaks also seem to increase, rather than decrease as the number of terms used increases. This is an inherent problem when attempting to approximate a discontinuous function with a series of continuous sine waves, and is referred to as Gibbs' phenomenon. When summed over an infinite series, the x values at which these peaks occur do eventually converge to the same as those for the corners of the square wave. However, the overshooting still remains for the y values when an infinite number of terms is used (Gottlieb and Shu, 1997). These are sometimes, rather comically, referred to as 'Gibbs' ears', or 'bat ears' (SEG Contributors, 2018).

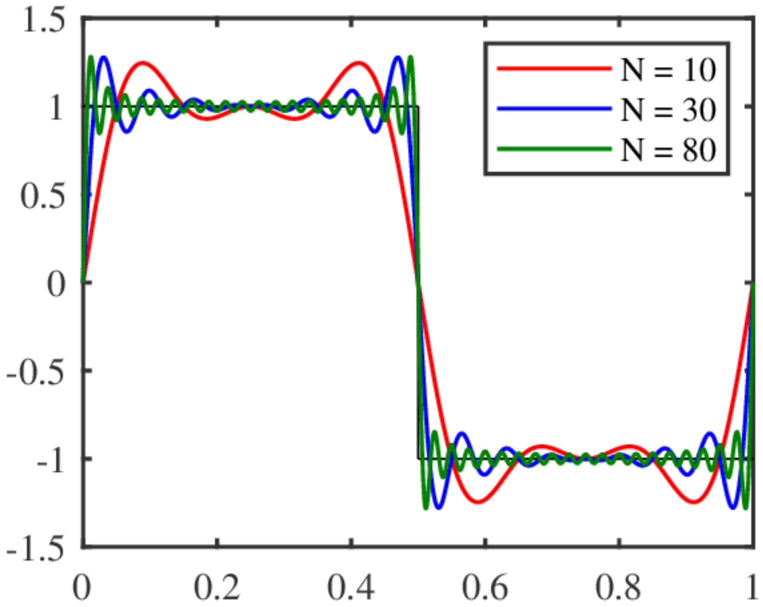


Figure 2.7: Fourier approximation of square wave by Ma and Sun (2019), where the black graph is the square wave function, and N is the number of Fourier terms used for the approximation

2.4 Related Work

There are several pieces of software that are related to this field of study. Broadly, most digital synthesisers use many of the principles of harmonics, timbre and Fourier synthesis discussed so far, and there are far too many to reference here. The specific link that I want to make and show in my tool, however, is the relationship between the physical parameters of sound waves, how resulting timbres are built using Fourier synthesis, and the resulting aural effect, those that demonstrate that particularly well are mentioned here.

2.4.1 Fourier Series Applet (Falstad, 2015)

This is an applet written in Java that shows Fourier approximations of the waves described in 2.2.2. It visually displays the coefficients used within the sine-cosine form of the Fourier series to approximate the functions, and when the ‘Mag/Phase View’ toggle is checked, it also shows the values used for the amplitude-phase form of the series. I think this tool is particularly good for showing Gibbs’ phenomenon when different numbers of terms are used, and supports a very large number of harmonics.

2.4.2 Fourier Synthesiser (Ruiz, 2017)

This is an audiovisual synthesiser written in HTML5, CSS and JavaScript. It allows the user to adjust the fundamental frequency, amplitude, and harmonic amplitude and phase values. It also provides several preset harmonic configurations, including those for the waves described in 2.2.2 that build up a wave using Fourier synthesis, up to a maximum of 16 harmonics. I discovered this tool very late in the process of writing this report, having already completed by implementation. It bears many similarities to the tool which I have implemented, and as such I have added a section to my evaluation chapter which compares the strengths and weaknesses of both tools (5.3).

2.5 Unity

I chose to implement my tool for this project in the Unity game engine (reference). This is a piece of software that allows developers to create both 3D and 2D systems and games, with code written in C# (reference). Unity structures its systems into scenes. This is useful in game development to create multiple different levels, menus and environments within a game, while still being able to access the same C# scripts that are shared between all scenes. For my tool, however, I will only need to use one scene. I chose Unity because, due to its focus on being a game development tool, it is specifically catered towards *real-time systems*, and has a good base UI system which allows the developer to customise each scene and add their own functionality to the various objects within it. It also has build and compilation options for many different platforms, and creates standalone applications that can access a good proportion of the available system's resources. I will now give a high-level description of the components of a Unity scene.

2.5.1 GameObjects

The objects within each scene in Unity are referred to as GameObjects. GameObjects have a name, a set of customisable components, and optionally can have a set of children, which are other GameObjects pointed to by the parent. They have some similarity to classes in an object-oriented language, in that they function as containers for related functionalities. Importantly, GameObjects can be stored as prefabs, which function as template GameObjects that can be customised and overridden within the scene, and used as attributes within C# classes. The only differentiating factor between GameObjects are its components.

2.5.2 Components

GameObjects have a set of components. The only compulsory component of a GameObject is a transform, which stores and controls the position and orientation of the GameObject within the scene. Other examples of components are materials and renderers, which determine the shape of the object, how light interacts with the it and how it is perceived by the scene camera, which itself is a GameObject with a camera component. Perhaps most importantly, GameObjects can have C# script components, which are used to add, remove and modify components from the GameObject, and can also create, destroy and modify other GameObjects in the scene. While other components determine a GameObjects static properties, C# scripts determine a GameObject's behaviour over time.

2.5.3 UI System

Unity provides a set of generic 2D UI GameObjects as standard, such as sliders, buttons, input fields and dropdown menus. The logic for each of these elements is attached to the GameObject itself as a component, and script components can be added to the GameObjects to customise the functionality. All UI GameObjects are instantiated as a child of the Canvas GameObject, which means these objects' positions are in terms of their location on the screen, rather than their location within the scene. Specifically, UI elements can be anchored to the corners or edges of the screen, and their relative position is then decided relative to these anchor points. This is useful to ensure that the UI can still fit correctly onto the screen, even when the tool is run on different displays with different sizes and aspect ratios.

2.5.4 MonoBehaviour Class

C# scripts that can be attached to a GameObject as a component must inherit from Unity's provided `MonoBehaviour` class. This class provides several optional functions that can be implemented, but I will only describe the ones that are relevant to the tool I have created:

- The *Start* method is run when the GameObject is instantiated, and is used to set up the attributes and methods of the class.
- The *Update* method is then run every frame. The code contained in this class is what determines the GameObject's behaviour over time, and so is where the main logic of a `MonoBehaviour` class is usually contained.
- The `OnAudioFilterRead` function is called at a rate that depends on the audio sampling rate (2.1.3) and the running platform, but it is usually around 50 times a second, or every 20ms. This function takes two parameters: An array of floats that are data values, and an integer that represents the number of output channels (either mono or stereo). The purpose of this function is to buffer a chunk of audio for output. For the tool I have implemented, the sample rate is 44100Hz, and on my machine it buffers 1024 values per channel, giving the data array a total size of 2048. This means the function is called every 23.22ms. The buffer used for `OnAudioFilterRead` is in interleaved format, meaning that the buffer takes values in the form `LRLRLR...`, where the first `L` value is played at the same time as the first `R` value, but in the left and right channel respectively.

Chapter 3

Design

I will now detail the process of planning and designing an interactive Fourier synthesiser in Unity (from this point forward referred to as ‘the tool’).

3.1 Display and Controls

3.1.1 Inputs and Outputs

The tool essentially has one main output: a waveform. This waveform should be displayed, and the corresponding sound should be produced by the tool. There are then several parameters that the user controls. Firstly, there are the parameters that concern the entire waveform: the fundamental frequency, the amplitude of the wave and its phase. As the phase of the entire wave makes no difference to the sound, there is no need to give the user control over this. Secondly, there are the controls that concern the harmonics of the wave. The user is able to control how many harmonics are present in the wave, the relative amplitude of these harmonics, and their phase. In the process of developing an early prototype that simply generated a wave and allowed the user to adjust the frequency, I realised that some way for the user to zoom in and see the individual periods of the wave would be desirable. So, the user can also control the zoom level of the wave. Finally, the importance of the digital waves described in (cross-reference) to Fourier synthesis motivated me to implement various preset configurations for the harmonics, with the user able to switch between these presets and not only see the Fourier approximation of the preset waveforms and the parameters that decide this shape, but also hear the timbre of these presets. The user may also want to adjust the weighting of certain harmonics within these presets, and therefore desire a way to reset harmonics to their preset position. So, the user also has the ability to do this.

The majority of the user input is taken via the mouse, rather than the keyboard, although some controls do have keyboard-activated alternatives. Generally, the average user will find using the mouse easier than purely keyboard-controlled tools, as they do not have to remember which key corresponds to each action, and they can physically see where they need to click. However, there is an extra degree of precision to be gained by using keyboard inputs. I have been reluctant to allow too much keyboard input for this tool as this would require several input fields, which are not particularly aesthetically pleasing from a UI design perspective, and this also requires the user to know exactly what the numbers they are inputting mean. By using abstractions of the underlying values such as sliders, I believe the design of the tool can be cleaner and less confusing to the average user. The ease of using the controls is something about which I enquired in the user study for this project, and will be discussed in 5.

3.1.2 Continuous Parameters

The majority of these inputs consist of the user adjusting *continuous parameters*, often bounded within a certain range. As the process of inputting specific values can be quite tedious and unappealing as a user, and can clutter a UI, this was something I wanted to avoid where possible. The most common solution for taking continuous input values is by using a slider, and several of these are present in my submission implementation.

The exception to this is for the control of phase, for which I implemented a rotatable ‘spinner’ instead. Due to phase being a cycling parameter (2.1.1), I thought it would be more intuitive to implement phase in a circular fashion, where the user can continue increasing the value for as long as they like, and the spinner will continue turning, with the values looping in the underlying implementation. As sine and cosine are defined in terms of periodic movements around the unit circle (reference), where the phase represents the angle between the x-axis and the line from the origin to the current function value, this is also a somewhat mathematically accurate way to represent phase, although in the tool I have set the point where phase is 0 to be in the upwards direction. For the purposes of the tool, however, I think this is more intuitive as the upward direction tends to represent the default value, or 0 value of a rotatable knob.

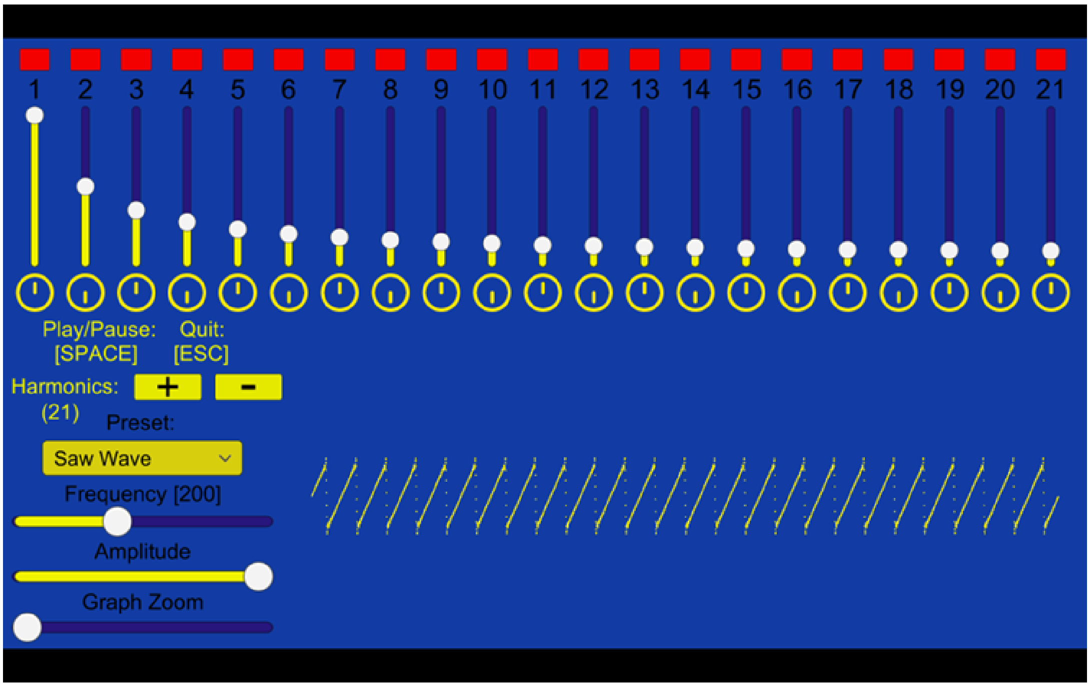
The two most common phase values used, especially in common Fourier series, are 0 and π. This is because alternating these values has the same effect as multiplying the amplitude by -1, and shifting the phase by π is thus sometimes referred to as *phase inversion* or *phase reversal*. I realised that the user may struggle to set the spinner to these values exactly, and the user may desire functionality to do this easily. Therefore, the user can double click each spinner to set it to these values. The details of how this is implemented are provided in 4.4.3.

3.2 User Interface

Ensuring that the tool works well and correctly is obviously very important, but I wanted the user experience to be pleasant as well. This involved planning how the user was going to control each element of the system, where to place these controls, and where to display the output. The final system design is shown in Figure 3.1.

3.2.1 Harmonic Controls

The harmonic controls are displayed at the top of the screen, consisting of:



Harmonics: (21)

Preset: Saw Wave

Frequency [200]

Figure 3.1: UI Design of the Tool

- A reset button, that reverts the amplitude and phase to the values determined by the current preset (3.2.2). This is an important element, as the user may wish to set the wave to a certain preset and then use this as a starting point to customise the wave and experiment with the results of their adjustments. Without a reset button, however, once the parameters of a harmonic have been adjusted, it is very difficult to ensure that the parameters are returned to the preset position. The only way to guarantee this would be to change the preset to another, and back again, which would have the side effect of also resetting all the parameters of every other harmonic as well. Thus, this justifies the need for an individual harmonic reset functionality. I also considered using this button as an on/off switch, but as this can be done by setting the relative amplitude of the harmonic to 0, it seemed this would be repeated functionality, and so the reset button would be more useful for the user.

- A label that shows the number of the harmonic that is being adjusted.

- A vertical slider that displays and allows the user to control the relative amplitude of the corresponding harmonic.

- A phase ‘spinner’ (3.1.2) that controls the phase of the corresponding harmonic.

To make the controls seem more responsive, each element changes colour when clicked. For example, the slider handles darken slightly while being clicked and held, as do the reset buttons, and the phase spinners change to red colour while being turned.

There are also ‘+’ and ‘-’ buttons that add or remove one harmonic, respectively, and a total harmonic count on the left of the screen in brackets. This is because the tool supports the ability to add more than 21 harmonics, so a total count allows the user to keep track of these extra harmonics.

3.2.2 Presets

There are several preset Fourier series from which the user can select, which approximate some defined waveforms. These can be selected from the preset dropdown menu, shown on the left of the screen. The current preset selected is shown on the label. When this is clicked, the dropdown menu unravels, as seen in Figure 3.2. The user can select a new preset by clicking the corresponding name. The current preset is also indicated with a tick symbol. Once the user has selected a new preset, the dropdown menu automatically collapses.

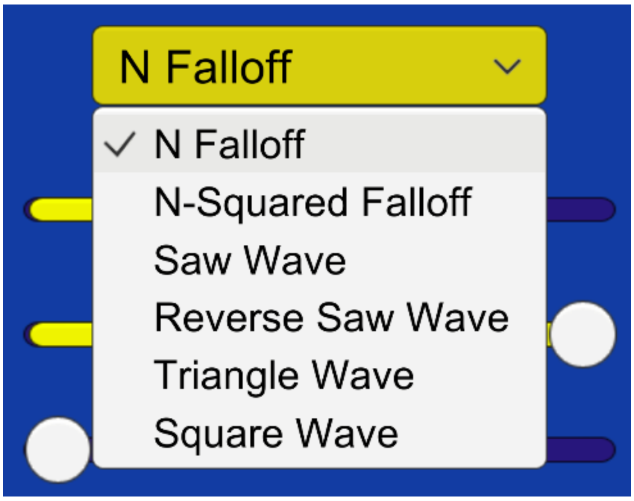


Figure 3.2: Preset Dropdown Menu

3.2.3 Wave Controls

The fundamental components of the wave output can be controlled by adjusting the sliders shown in the bottom left of the screen. These control the fundamental frequency of the wave, the amplitude of the whole wave and the zoom level of the displayed waveform. The value of the frequency is also displayed next to the frequency slider.

3.2.4 Waveform Graph

The output waveform is displayed at the bottom right of the screen. When the user increases the value of the zoom slider (3.2.3), the horizontal distance between each point on the graph is increased, so the user can more clearly see the shape of the waveform over the individual periods of the graph.

3.2.5 Colour and Clarity

I have chosen a general colour scheme of blue and yellow for the tool. These colours stand out from each other which makes the controls easy to see. I have aimed to make the colour-scheme of the tool colour-blind friendly, and so have tried not to use combinations of red and green, which those with the most common form of colour blindness can have difficulty distinguising. There is a less common form of colour-blindness called blue-yellow colour blindness, however despite its name this has more of an impact on one's ability to distinguish blue/green combinations and red/yellow combinations. While there are both yellow and red elements to the system, namely the sliders and reset buttons, it is not essential that these two elements are distinguished by colour, as they should still appear distinct from each other due to the positioning of the controls.

Despite the number of different controls that are available to the user in the tool, I have made an effort to stop the screen becoming too cluttered with numbers, values and labels. I have also tried to ensure that each element is big enough on the screen, so that it is readable. This also has an impact on the precision level of the sliders. The larger the slider, the greater degree of the precision that the user has over the tool. Equally, if the sliders become too large, then they dominate the screen, and this reduces the amount of space that the other elements of the system can take up. I have taken all these factors into account and tried to produce a UI design that balances the above concerns well. I will evaluate the effectiveness of the design using a user study and my own reflections in Chapter 5.

3.3 Priorities

Beyond the user interface, there are some design principles and priorities that I have considered when designing the structure and details of the implementation of the tool. These can be prioritised as follows:

- **Accuracy**

  Perhaps the most important priority for developing a tool of this nature, especially with its potentially educational uses, is that the audio and the visuals are always accurate to each other. Concretely, this means that when the user adjusts the wave parameters, the waveform should accurately reflect the changes and equally, for any waveform that can be generated, the parameters should be accurate such that the user can recreate that wave by setting the parameters to the same values in future. The way that I have ensured this is the case in the tool i have developed is by ensuring that each parameter is stored once, and passed around the system by *reference*, as opposed to by value. This includes the values that describe the waveform itself. The displacement values for the wave are stored in a list. These values are not only used to calculate the position of each point in the displayed waveform, but are also passed to the the audio output. This ensures that the wave the user is seeing always corresponds to the wave that is displayed and the displayed parameter values.

- **Responsiveness**

  My main criticism of the system described in 2.4.2 is that in some cases, the user has to adjust the parameters, and then press a button to make the system take the parameter adjustments into account and calculate the new output. It is a priority to me that I avoid this, and make sure that the tool responds to the user adjusting parameters in real-time. This is because almost all the relevant parameters in the tool are continuous, and I think one of the most interesting things to see is the wave gradually changing from one shape to another as a parameter is adjusted, and how this affects the sound. I have ensured this by again making sure the user can directly control the values that are used to calculate the wave, and any change in these values triggers a recalculation of the wave. The aim is thus to have a tool where the responsiveness feels as close to an analogue synthesiser as possible.

- **Efficiency**

  While efficiency should be a priority in the development and design of any system, due to the real-time quality of the tool, inefficient calculations will reduce responsiveness and accuracy, and negatively impact the overall user experience. The main bottlenecks for the tool will be in relation to recalculating the values that make up the entire wave. I will give specific details on how I have attempted to reduce the computational power required to run this tool in 4.

- **Ease of Use**

  A final priority in designing this tool is ensuring that it is not too complex or unpleasant to operate. The theory behind Fourier synthesis can be quite technical and confusing for someone who is unfamiliar with the relevant background study. Furthermore, without knowledge of what each wave parameter actually controls, the tool could be quite an intimidating introduction to this field. Ideally, the tool should be simple enough to use such that even with limited knowledge, the user can gain some understanding of what each control is doing to the wave, both physically and in its aural effect. In order to achieve this, I have tried to make the user interface uncluttered, and with as few specific numerical values as possible, to allow smooth control of the different wave parameters, and to make the audio as smooth and unbroken as possible, with few artifacts, so as to not be off-putting to the user. Ensuring that the audio was accurate, while not off-putting to the user, ended up being a rather difficult and time-consuming task, and I will discuss the details of the efforts I have made regarding this in Chapter 4.

Chapter 4

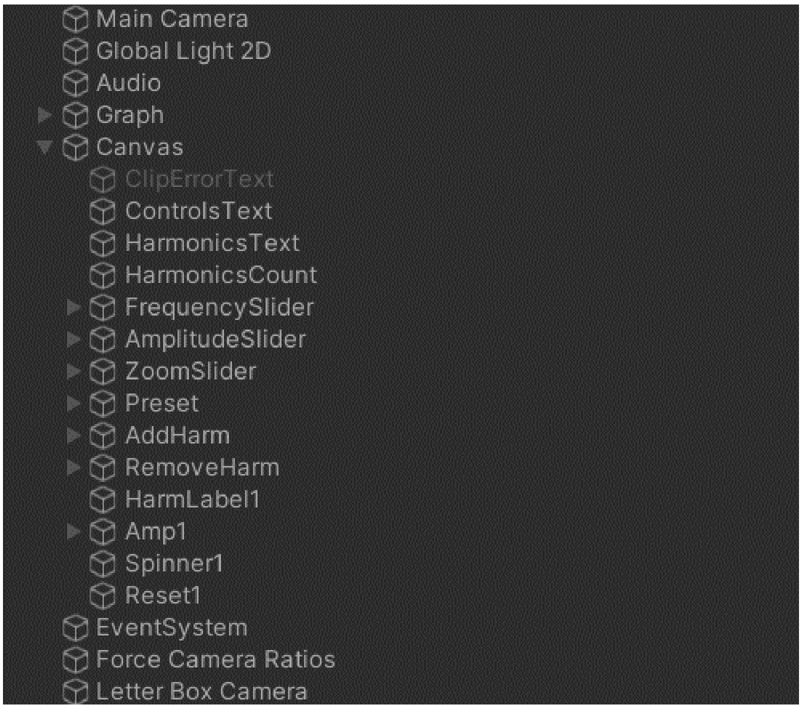
Implementation

In this chapter I will give a detailed view of how I have used the framework provided by Unity to create an interactive Fourier Synthesiser. It includes sections on how the audio is generated, buffered and looped, how this is displayed graphically, and how the various UI controls are implemented. To give a high-level view of how the system is constructed, I have provided the names and structure of the different GameObjects in the tool below. While Unity can lend itself quite well to object-oriented programming for large games, I have used minimal object-oriented concepts for this system, so the GameObject Structure is far more informative than a traditional class diagram in this case.

4.1 GameObject Structure

Figure 4.1 below shows the GameObject structure of the scene for the tool:

- The Main Camera and Global Light are the standard 2D Camera and Light provided for 2D Unity scenes.
- The Audio GameObject has an AudioSource component and the AudioGeneration script (4.2) attached. This is the main control script that takes input from the UI components and also sends a message to them when their value needs to be updated.
- The Graph GameObject controls the drawing and redrawing of the graph, by determining the position of each of the graph points via the WaveGraph script (4.3). Each of the graph points is a circle GameObject that is a child of the Graph GameObject.
- The Canvas (as described in 2.5.3), controls the appearance and positioning of all the input UI elements via the UI script (4.4). Most importantly, this includes the dynamic creation and destruction of the harmonic controls.
- The various sliders, text elements and buttons are then the children of the canvas. Some of these have their own children which represent the different subcomponents of the object, and some also have their own scripts attached that



- Main Camera
- Global Light 2D
- Audio
- Graph
- Canvas
  - ClipErrorText
  - ControlsText
  - HarmonicsText
  - HarmonicsCount
  - FrequencySlider
  - AmplitudeSlider
  - ZoomSlider
  - Preset
  - AddHarm
  - RemoveHarm
  - HarmLabel1
  - Amp1
  - Spinner1
  - Reset1
- EventSystem
- Force Camera Ratios
- Letter Box Camera

Figure 4.1: GameObject Structure and Hierarchy

relay their value back to the Audio GameObject. For example, the sliders have a child GameObject for the handle, the area in which the handle can move, the background, fill area and label.

- The EventSystem is used to detect the behaviour of the mouse on the screen, and when the user clicks or gives keyboard input, and relates this information to developer-defined functions.
- Force Camera Ratios and Letter Box Camera are both used to ensure that the tool runs at an appropriate aspect ratio, and adds black bars at the top and bottom of the screen when this is necessary. Letter Box Camera is a Unity asset available at the Unity Asset Store: (reference)

I will now describe the functionality of each of the scripts in the system, and the GameObject to which they relate. Each of these scripts can be inspected at the following git repository in the sub-directory “Assets/Scripts”: `https://github.com/MajesticBevans/dissertation`.

4.2 Audio Generation and Wave Operations

As mentioned above, the AudioGeneration script is the main control script of the tool. It directs when the different parts of the tool are updated, when the waveform is updated, and passes the resultant data to the audio buffer via OnAudioFilterRead (2.5.4).

4.2.1 Constants and Variables

The wave values that are used for both the graph and audio-output are stored in a list of floats called `currWave`. The frequency, amplitude and linear initial phase of the wave are also stored as floats. The parameters concerning the harmonics of the wave are stored in a dictionary called `harmonics`, where the key is the harmonic number, and the value is a tuple of the amplitude and phase values. Within this script, I have also defined an enumeration type (.NET contributors, 2023) for the different wave presets, with the currently selected preset stored as `currPreset`. An integer `timeIndex` is also stored which keeps track of the current progress through the buffer. By dividing `timeIndex` by the sample rate, we can obtain a *t* value that can be used in variations of the generic wave formula detailed in 2.1.1.3.

The majority of the functionality in this script is triggered when one of the above values changes. To detect the changes, copies of the variables, with the prefix `prev-`, are also stored. In the regularly called functions, the current value is checked against the previous value and if they differ, an operation is run and a Boolean is set. Storing both the previous values and the Booleans allows some computational optimisation, such as ensuring that operations are not unnecessarily run multiple times per frame, allowing communication between separate threads such as the audio and visuals threads (4.2.2), and using the ratio between the current and previous values to optimise wave recalculations.

`currWave` is recalculated regularly to remove any inaccuracies that may arise due to floating-point arithmetic issues. This simply involves a counter, that is incremented on every frame where the graph is *not* redrawn, and a maximum counter value, that is stored as a constant set to 60 in the submission build. This ensures that the wave recalculation only occurs during a frame where no other operations are performed, and at at its maximum rate will run every 60 frames, so as to not overload the CPU.

There are also two buffer sizes stored: `graphBufferSize`, which represents the number of values passed to the `WaveGraph` class to be displayed, and `audioBufferSize`, which stores the size of the buffer through which the audio output loops.`graphBufferSize` is set to the value of `SAMPLE_RATE / MINIMUM_FREQUENCY`. This is because the buffer must be of a size such that it can contain the number of samples needed to complete one period of a wave. Setting `graphBufferSize` in this way guarantees this, and that the buffer is as small as it can be without losing any accuracy. Hence, this is an example of where I have tried to balance both efficiency and accuracy.

`graphBufferSize` remains constant during execution, but `audioBufferSize` must be dynamically updated. These two buffer sizes are needed to prevent displacement jumps as the buffer loops, which is particularly relevant when changing frequency, described in 4.2.4.3.

4.2.2 OnAudioFilterRead and Update

The reason that not all operations can be triggered from the same point is because of some multi-threading that Unity performs. Recall that there are two functions that can be called regularly in `MonoBehaviour` scripts (2.5.4): One is `Update()`, that is called every frame, and the other is `OnAudioFilterRead(float[] data, int channels)`, which is called about 50 times a second when a new chunk of audio is buffered. The visuals and audio of Unity run on separate threads, which means that `OnAudioFilterRead` cannot affect the visuals, and `Update` cannot affect the audio. So, when the user changes the frequency of the wave, the audio should respond as soon as possible (i.e. the next time `OnAudioBufferRead` is called), to uphold the responsiveness of the tool, but the graph and slider cannot be updated in the same function. Therefore, a Boolean is set so that when `Update` is called, the graph is redrawn to reflect the changes in the audio. To maintain smooth audio, I have minimised the computation in `OnAudioFilterRead` to avoid pops and gaps.

4.2.3 Generating a Wave

Recall that our sound wave formula with linear phase values is as follows (2.1.1.3):

$$displacement = A \cdot sin(2\pi(ft - \phi))$$

This can be extended to create the algorithm that generates a wave with a given harmonic configuration, as shown in the following pseudocode:

```
for (i = 0, i < wave.Count, i++)
{
    value = 0

    for (key in harmonics)
    {
        harmPhase = (phase + harmonics[key].phase) % 1f

        value += amplitude * harmonics[key].amplitude *
        sin(2 * pi * key * ((i * frequency / SAMPLE_RATE) - harmPhase))
    }

    displacement[i] = value;
}
```

For each value in the wave list, the displacement value for each harmonic is added to create the superposition of the individual waves. These values are then passed to `OnAudioFilterRead`, where `timeIndex` is also incremented on each loop. When `timeIndex` reaches the value of `audioBufferSize`, it is reset to 0.

Because the phase value for each harmonic is multiplied by the harmonic number, the values cycle from $0-1/n$ for each harmonic number, rather than $0-1$. For m harmonics, and a wave of n samples, this algorithm runs in $O(mn)$ time.

4.2.4 Changing Frequency

Changing the frequency of the wave is probably the most intensive operation in the tool. The task is to join a wave of a specified frequency to another wave of an arbitrary frequency at a given point through the waveform, shown in Figure 4.2. Perhaps surprisingly, I could not find many references to this problem, or indeed a common solution that is used. So, I developed my own method for this tool.

4.2.4.1 The Problem

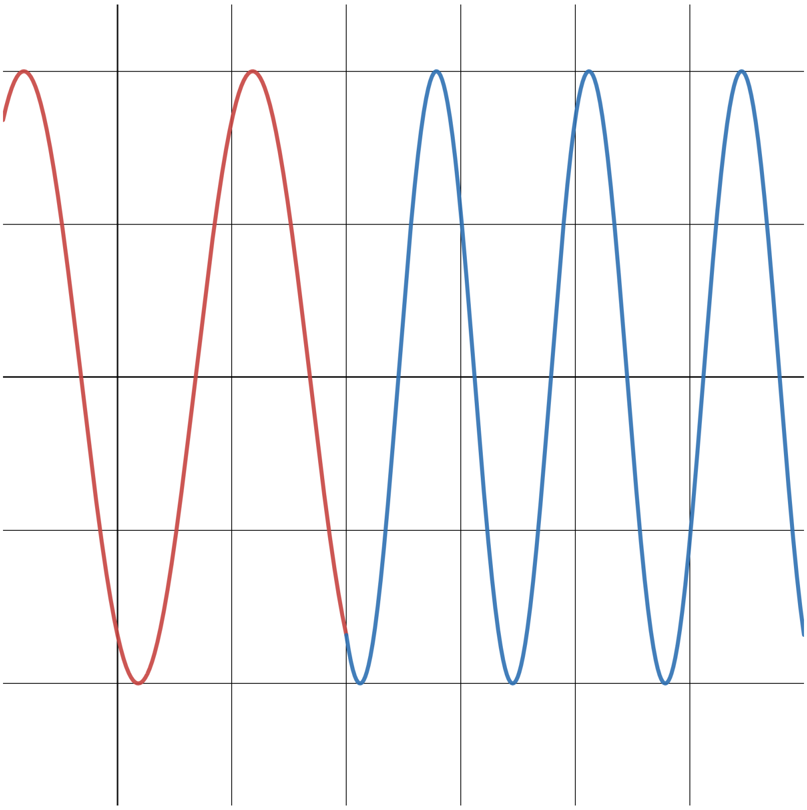


Figure 4.2: Joining two sine waves of differing frequency

Due to the design of the system, the user can only change one parameter at a time. This allows me to assume that I will only have to connect two waves of the same amplitude and harmonic composition, which simplifies the problem from joining up two completely arbitrary waves. This is because the displacement values for both waves are equal when their phase values are equal. As both waveforms have the same shape, they will also certainly be curving in the same direction at the same phase value, which means there won't be any unexpected kinks in the resultant wave. If we generate the wave with the new frequency at the appropriate *starting phase*, such that the phase value at the point where the frequency is changed is equal for both waves, then they will smoothly blend together. When I initially encountered this task, I only aimed to develop a solution to connect two basic sine waves, believing I may need to extend this at a later date to account for different harmonic compositions. However, for the reasons described above, no modifications were needed. Concretely, given two frequencies f_1, f_2, an amplitude A, phase value ϕ_1 and a point at which the frequency is changed t, the task is to find a value ϕ_2 that satisfies the following equation:

$$A \cdot sin(2\pi(f_1 t - \phi_1)) = A \cdot sin(2\pi(f_2 t - \phi_2))$$

4.2.4.2 The Solution

The frequency change is detected in `OnAudioFilterRead`, by comparing `frequency` with `prevFrequency`. If these are not equal, and the wave is currently playing, then the function `BlendSine` is executed. `BlendSine` takes the previous frequency, current frequency, harmonics dictionary, current starting phase (i.e. the phase value at $t = 0$), time index and current waveform as parameters.

To do this, the key value to obtain is the phase value of wave 1 at time t, as this will equal the phase value of wave 2 at time t, and from this we can determine the starting phase value for wave 2. We can do this by first obtaining our t value, `timeIndex / SAMPLE_RATE`, and then multiplying this value by the frequency. This effectively expresses t in units of *periods*, rather than *seconds*. As I have used linear phase values throughout the tool, by then subtracting the floor of this value (the greatest integer less than or equal to the value), the linear phase value at t can be obtained.

Now that we have the phase value, we need to write the new wave into the buffer. In a previous implementation, I did this by writing the new values into the remaining slots of the buffer, and then filling in the start of the buffer that contained the previous wave on the next frame. However, this required keeping track of the point at which the frequency was changed, two separate functions for writing the start and end of the buffer, and caused problems when multiple frequency changes occurred in a short space of time. So, I changed my implementation to rewrite the entire buffer when the user changed the frequency. Rather than writing wave 2 into the buffer starting from `timeIndex + 1`, I instead subtract the phase value at t from the current starting phase value at $t = 0$, to obtain a new starting phase value for $t = 0$, and rewrite the entire buffer from the beginning, returning the new value for `currPhase`. Once the value has returned, I simply reset the `timeIndex` to 0. When `OnAudioFilterRead` is next called, rather than continuing to read from where it stopped in its last iteration, it will begin reading the values from the beginning of the buffer which will smoothly follow on from the last read values.

At a first glance, it may seem like this algorithm might only work for the simple case of two sine wave, but the algorithm will work regardless of the number of harmonics present, and their parameters, because of the use of `currPhase`. Indeed, changing frequency is the only purpose for which a `currPhase` variable is needed. It may be confusing to someone observing the code why there is a phase value for the first harmonic (the fundamental) and also a phase value for the entire wave, as it might appear that these represent the same thing. The reason for these two values is that the phase value for the whole wave is used as a reference point, which is changed each time the frequency is updated. If I had implemented frequency changing in a way that did not affect the starting phase value for the wave, then I could assume it is always 0 and only use the phase values for each harmonic. As it is, the first harmonic's phase value represents its phase *difference* between the first harmonic and the `currPhase`.

[If can, add more justification for method along with alternative options such as waiting until the end of the buffer. Maybe another subsection called alternate solutions, and take sentences from above and append]

4.2.4.3 Updating the Buffer Size

There is one final issue to address with regards to changing the frequency. When `timeIndex` reaches the end of the buffer, it is reset to 0, and so the buffer loops. This achieves a smooth connection between the end and start of the buffer when the `bufferSize` equals the sample rate, and the frequency is an integer. However, the user is able to change the frequency to non-integer values. When a non-integer frequency is selected, the end and start values of the buffer will not be continuous in the waveform, and so the buffer will not loop correctly and a 'pop' will be produced as the displacement from the final value to the first. To prevent this, two different buffer sizes are used: the *graph* buffer size and the *audio* buffer size. Each time the frequency is adjusted by the user, the audio buffer must be resized to prevent a displacement jump at the buffer loop point. The audio buffer size is calculated as follows:

1. We can obtain the number of complete periods that will fit into the buffer by taking the floor of `frequency * graphBufferSize / SAMPLE_RATE`.
2. This period count is then multiplied by the number of samples per period: `SAMPLE_RATE / frequency`. This gives the number of samples within the number of complete periods that fit in the buffer. This will give the maximum buffer size where the final value will follow onto the first.
3. Finally, the floor of this value is taken to ensure it is an integer, as the buffer size cannot be a non-integer value (we cannot have a fraction of a sample).

4.2.5 Changing Harmonics

Changing amplitude is perhaps the least complex operation in the tool, and so does not warrant an entire section in this report. By maintaining a record of the previous amplitude, the ratio between the current and previous amplitude is calculated, and this ratio is multiplied onto all points of the wave. What does require some explanation, however, is the way I used my dictionary of harmonic parameters to update the wave values.

For adding a new harmonic, the values of the wave with the harmonic frequency, amplitude and phase can simply be added to `currWave`, and similarly when removing a harmonic they can be subtracted. For adjusting the amplitude of one of the harmonics, I initially subtracted the values of the harmonic wave with the previous amplitude from the wave, and then added the values of the wave with the new amplitude. However, this can be done more efficiently by just adding the wave with amplitude equal to the difference between the current and previous values, and this is what I have done in my implementation. This is not the case, however, for harmonic phase values. The wave with the previous phase value must first be removed, and then the wave with the new phase value added.

These operations are all of order $O(m)$, where m represents the number of samples in the graph buffer, which allows them to be quite efficient compared to the frequency changing and wave generation algorithms, and therefore smooth-sounding and responsive in the implementation. Before implementing, I wondered whether these operations may be a source of ‘popping’ in the audio playback, but due to the continuous nature by which the user must change these variables, the displacement value difference between the buffers before and after the variable changes are minimal, providing the user does not adjust the parameters particularly rapidly.

4.2.6 Changing Preset

When the selected preset is changed, the values stored in the `harmonics` and `prevHarmonics` dictionaries are both updated by the function `handlePreset`. This function takes a harmonic number as input, and returns a relative amplitude and phase value based on the Fourier series of the type of wave that corresponds to the selected preset. The function only changes one harmonic at a time to accommodate the functionality of the reset buttons, which set a single harmonic's parameters to their preset value. When the user changes the preset via the dropdown menu, `handlePreset` is called for each currently added harmonic.

Implementing `handlePreset` required translating each Fourier series from a mathematical formula into code. The function uses a switch statement based on the new preset for efficiency over an if statement, where each case assigns an amplitude and phase value to be returned.

After the new values for each of the harmonics have been set, the wave is recalculated using the algorithm shown in 4.2.3. As it is almost certain that each harmonic in the wave will need to be adjusted, there is no reason to use the $O(m)$ algorithms described in 4.2.5, as this would involve an unnecessary number of function calls. Both `harmonics` and `prevHarmonics` are updated with the same values to prevent the tool from then adjusting the wave further, and allows future changes to be detected accurately.

4.3 Graph

In the `Start()` function, `AudioGeneration` instructs the `WaveGraph` class to instantiate a number of points equal the the `graphBufferSize`. These are spaced evenly in the x direction, and their spacing is also determined by `graphBufferSize`. The larger the buffer, the smaller the x-distance between each point. In the `Update()` function, i.e. each frame, the current user-defined zoom value is checked against the previous zoom value, and if they differ, the graph is redrawn with a new x-distance between each point. In order for this to work, a local copy of the values that were last sent to this class by `AudioGeneration` is stored.

The algorithm for redrawing the graph is shown below:

```
void draw(List values)
{
    currValues = values

    for (i = 0, i < values.Count, i++)
    {
        x = startX + (i - values.Count/2) * zoom
        if (x < startX - graph_bounds or x > startX + graph_bounds)
        {
            graphPoints[i].SetActive(false)
        }
        else
        {
            graphPoints[i].position = (x, startY + values[i]*HEIGHT)
            points[i].SetActive(true)
        }
    }
}
```

This algorithm takes the incoming values (taken from `currWave`) and changes the position of each graph point to a scaled version of these values. The x values are scaled by the `zoom` variable, and the y values are scaled by a constant called `HEIGHT`. The if statement in this function ensures that when the user zooms in, the graph points to the left or right of the defined graph bounds are set to inactive, which means that they are not shown to the user in Unity. Their new y position is also not computed for efficiency.

4.4 User Interface Implementation

4.4.1 UI Script

The majority of the UI behaviour is directed by the the `UI` script that is attached to the Canvas GameObject, of which each interactive UI element is a child.

The main purpose of this script is to handle the dynamic addition and removal of the harmonic control UI elements, and this functionality is contained within a public function called `RedrawSliders`, which takes a `harmonics` dictionary as its sole parameter. The script stores a dictionary for each type of UI element used to control the harmonics of the system: The reset buttons, harmonic labels, amplitude sliders and phase spinners. When `RedrawSliders` is called, the keys of the `harmonics` dictionary are compared with the keys of the dictionary of amplitude sliders. For each key that is present in the parameter dictionary, but not in the slider dictionary, a new set of UI elements are instantiated and set up at a constant offset along the x-axis. This is done up to a maximum slider count, which is 21 in the submission build. Similarly, for each slider key that does not appear in the parameter dictionary, the elements with that key are destroyed. Finally, if the keys for the two dictionaries are the same, then the values for each UI element are checked and updated if necessary to those passed to the function.

By having prefabs (2.5) for each element as attributes of the script, the computation needed to set up these elements is reduced. Only their position, value and behaviour needs to be set. As the sliders and buttons are standard Unity UI elements, their behaviour can be set using Unity provided functions: `OnValueChanged` for sliders and `onClick` for buttons. Developer-created functions can be added as ‘listeners’, and so are executed when a particular event occurs. Specific parameters can also be set for when these functions execute. The functionality for this is handled by Unity's event system, and this is the reason it appears in the GameObject hierarchy (4.1). These listeners are set as follows:

```
amp_slider.onValueChanged.AddListener
(delegate {ChangeAmpSliderValue(harmNumber);});

reset_button.onClick.AddListener
(delegate {ResetValues(harmNumber);});
```

4.4.2 Sliders

The sliders that are not dynamically created are set up with their own `MonoBehaviour` script. The start function is used to set up the slider's parameters and also associate it with its label. For example, the frequency slider has its minimum and maximum value set to the corresponding values stored in `AudioGeneration` (4.2), its label attached and set to the starting frequency value, and its `OnValueChanged` function delegated to a function that updates `AudioGeneration`'s frequency variable and the label text to match. The amplitude and graph zoom sliders have a much more minimal script that simply updates the corresponding values in `AudioGeneration` and `WaveGraph`.

4.4.3 Phase Spinner

Unity's provided slider was very useful to represent most of the continuous parameters, but I wanted the phase adjustment to be more intuitive, and importantly, loop in the same way that phase does. It therefore made sense to create a new UI element with its own custom script that was circular. I used a sprite from a package available on the Unity Asset store (Hippo, 2022), that is suggested to be used as a loading icon, and applied a custom script to give it the functionality that I wanted. In terms of the UI, this is the main innovation of the project. The phase spinner template is stored as a prefab, with a custom script called `Spinner` (inherited from `MonoBehaviour`) as one of its components. When this prefab is instantiated, its image sprite, colour, relevant harmonic number and current rotation value are set up in the `Start` function. There are then 5 things that this script must do:

- Detect if the spinner is clicked
- Detect if the spinner is double-clicked
- If a double click is detected, then set the value to 0 if not currently 0, or to 0.5 if the value is currently 0.
- If a single click is detected, track the user's mouse movements while the they are holding down their mouse.
- Reflect these movements in the rotation of the spinner, and return the corresponding phase value back to the `AudioGeneration` script to update the graph and audio

I created a Boolean function isPointerOverSpinner to detect whether the user's mouse is hovering over the spinner. This function uses Unity's event system for detection. Metadata about the user's mouse pointer can be copied and stored in a class called PointerEventData. The pointer's position can then be used as the origin of a ray cast, by calling the Event System's RaycastAll function, which returns GameObjects as its results. The pointer is not guaranteed to only be over one GameObject at a time, so the results are returned as a list. isPointerOverSpinner then searches through the list of results, and returns true if the spinner GameObject is present in the results.

In Update, isPointerOverSpinner is checked every frame in which the spinner is not already being rotated. If the user then clicks, the current time is recorded. The function then checks if the current time is less than the sum of a variable called initial_click_time (set to 0 in Start) and a constant DOUBLE_CLICK_DELAY, which is set to half a second. If this is the case, then a double click has occurred. If the current value is 0, then it is set to $1/(2n)$, where n is the harmonic number, otherwise, the value is set to 0.

If a single click is detected, however, then initial_click_time is set to the current time (in case a second click occurs before DOUBLE_CLICK_DELAY, the current value, x-axis position of the mouse pointer, and rotation of the spinner are stored as starting values, the colour of the spinner is changed and Boolean phasing is set. While phasing is set, the difference between the mouse pointer's current x-axis position and the starting x-axis position is scaled by a constant. As Unity stores rotation values in degrees, the modulo of this scaled value is taken over 360 to give a corresponding rotation value, which is added to the starting rotation value of the spinner. Finally, the scaled value is added to the current phase value and returned to audioGeneration to recalculate the wave. If the user releases their click while phasing is set, then it is returned to false and the colour of the spinner is returned to its default colour.

Chapter 5

User Evaluation, Self-Review and Extensions

5.1 User Study

To evaluate the user-experience when using the tool, I created a user study, for which I received 8 responses. The user study aims to obtain feedback about all aspects of the system, but especially regarding the UI design of the system. This is so this can be improved in future to allow a wider range of users, specifically with regard to their level of knowledge in this field, to use the tool effectively and gain some understanding of Fourier synthesis and sound.

The study was sent out as a Google Forms link (Appendix D), with a download link for the different builds of the tool for each common PC platform (Windows, MacOS, and Linux). For the study, each participant downloaded the tool onto their machine, and was instructed to spend some time using it (at least 10 minutes was recommended) and complete a task to test their understanding. An annotated image of the UI at the time was used to explain the controls, which can be viewed at Appendix C. The particpant information sheet and consent form are also available at Appendix A and B, respectively. The task was described as follows:

1. Select the n-squared falloff preset.
2. Add harmonics until there are 21 total.
3. Increase the amplitude of the second harmonic so it is at its maximum.
4. Increase the amplitude of the third harmonic so it is about halfway to its maximum.
5. Invert the phase of all of the even harmonics by setting the value to pi (see controls below if this isn't clear).
6. Remove the harmonics one-by-one, starting at harmonic 1, by setting the harmonic amplitude to 0, and listen to the timbre of the sound as it changes. If you have done the previous steps correctly, you should still be able to hear the original pitch, even though it is not actually being generated by the synthesiser.

7. See how many harmonics you can remove before you can no longer detect the original pitch (remember this number for the questionnaire!).
8. Finally, re-add the harmonics in reverse order by hitting the reset button above them, until the harmonic spectrum is restored.

I received 8 responses to the study. While this is not a large number of responses, and perhaps too few to draw any statistically significant conclusions, I have still been able to gauge what did and did not appeal to the participants in general, and this should allow me to make future changes to improve the general user experience for the tool. I will now breakdown the results of the study:

5.1.1 Results

- **On a scale of 1-5, where 1 implies very difficult and 5 implies very easy, how easy/intuitive did you find controlling the synthesiser?**

  In general, people seemed to find controlling the synthesiser intuitive, with 62.5% of participants answering 4, 25% answering 5 and 12.5% answering 2. This is good news, as despite users not necessarily having particularly strong background knowledge in this area, they were still able to understand how to control the tool.

- **On a scale of 1-5, where 1 implies you hated the look, and 5 implies you loved the look, how do you feel about the look of the system? (e.g. the colours, placement of different controls, design of the controls etc.)**

  The results here were more mixed, with 37.5% of participants answering 3, and the same proportion answering 4, and then 25% answering 2.

- **Were there any control features you particularly liked, or found worked very well?**

  The two most common responses here were in regards to the responsiveness of the sliders, the clarity of the graph and use of the zoom feature. Participants seemed to appreciate the ability to zoom into the graph to gain a finer granularity of detail, but also to zoom out and put each waveform into perspective of the wave. The word ‘smooth’ appeared in multiple responses and participant 3 appreciated that the graph was ‘quick to increase and decrease’ when the harmonic amplitude sliders were used. Participant 7 also appreciated the dropdown menu to change preset, and noted that using the arrow keys to add and remove harmonics ‘felt better than using the buttons’, although they also noted that using the right and left arrow made more sense to then than using up and down.

- **Were there any control features you particularly disliked, or found didn’t work well?**

  The most common response to this was with regards to the phase spinners. 3 participants commented that changing the phase of individual harmonics did not feel very intuitive, and ‘took some getting used to’. One participant specifically mentioned inverting the phase as not being intuitive. The only other comment was that the reset buttons did not ‘look great’

- **Were there any elements of the synthesiser that you couldn't control that you would have liked to? Or any features you would like more *precise* control over?**

  There were 3 things highlighted by participants in this section:

  1. One participant desired some numerical values for the harmonic parameters
  2. One participant wanted the ability to key in a specific frequency
  3. Two participants mentioned that they would like some extra precision on the sliders, as it could be difficult to precisely control the amplitude of the harmonics when they were at low levels. One of these specifically mentioned that 'cursor control can be difficult on a laptop', and suggested some buttons may be helpful for finer control.

  Some other comments were made in response to other questions that I think belong in this section. One participant noted that they would like some functionality to be able to control the amplitude of all the harmonics at once. I believe they are referring to a sort of scaling feature. Another participant noted that a simple button to mute each harmonic would be useful, and potentially more useful than the reset button.

- **On a scale of 1-5, where 1 means you could not do it and 5 means it was very easy, how easy was it to complete the task described in the instructions?**

  Generally, participants seemed to complete the given task easily, with 75% answering with a 4 or 5. No participants were unable to complete the task.

- **If you were able to complete the task, how many harmonics did you remove before you were no longer able to hear the original frequency?**

  The results here ranged vastly, from 1 to 16 harmonics removed. As explained in ???, I expected a range of results here, but not quite to this extent, especially with such a small sample size. Both the mean and median values were approximately 10.

- **Did you learn anything while using the system? Were there any behaviours that you didn't expect?**

  Participants seemed surprised by some of the interesting wave shapes that could be created, especially when adjusting the phase values. One participant specifically noted that they didn't know how sensitive sound is to harmonic phase differences.

- **Do you have any other feedback about any aspect of the system? (e.g. likes, dislikes, features you would like to see, performance comments etc.)**

  There were several useful comments left here. Two participants noted that the colours should be of a softer palette that complement each other better, that the black text on the blue background was difficult to distinguish, and it was also noted that bounding boxes on the UI elements would be useful. Two participants also responded that a more detailed introduction, with some more explanation of the purpose of the tool would be useful for the study, potentially with some extra detailed tasks for them to familiarise themselves with the tool. One participant also noted that an option to run the tool in windowed mode, rather than fullscreen, would be appreciated. Another comment was also made about the precision level of the sliders.

5.2 Discussion and Self-Review

Overall, I am happy with the feedback I have obtained. I think it highlights some useful points, which can be used to improve upon the current implementation for future releases. I am glad that, in general, the participants seemed to find controlling the tool an intuitive process, and no one had particular trouble getting the tool to do what they wanted it to do. There were also no comments with regards to audio ‘popping’, or the audio quality at all. In fact, the participants seemed to appreciate the smooth, responsive nature of the tool, which was a priority for me during development. However, there were several points raised that highlight some drawbacks of the tool, and some improvements that I would like to make for future releases.

5.2.1 Positives

Participants seemed to appreciate the smoothness and responsiveness of the tool, which I can attribute to the efficiency of the algorithms used, especially those that control the changing of harmonic parameters, and also the choice of Unity for its real-time system strength. Another element that the participants appreciated was the wave graph, and the ability to zoom in and out which was mentioned several times. I think allowing the user to see both the individual periodic waveforms of the graph, and how these waveforms repeat in the overall context of the wave was a good decision from which users benefitted. There were also positive comments about the use of a dropdown menu for preset selection, and the alternative keyboard inputs for certain controls. I think the keyboard could be utilised more in future builds, and this could be used to satisfy some of the participants’ desire for more precise control over certain parameters as well. I also received no negative comments about the layout of the UI elements, although I have since moved the instructions for how to play/pause the audio and exit the tool to the bottom left corner for aesthetic purposes.

5.2.2 Improvements

A few participants gave negative feedback about the use of the phase spinners. As this is a custom UI element, and one that may be quite different to controls that the participants have used before, adjusting values in this way may not have come as naturally the participants as the other controls of the system.

In a previous version of my implementation, sliders were used to adjust the phase values. This achieved the continuous effect, but I was not satisfied with this method of control for phase, as the values did not cycle in the way that is accurate to what they represent. The sliders also took up far more space than the phase spinners do, resulting in a more cluttered UI. Another alternative input method, which is used in the synthesiser developed by M. Ruiz, is to use text input to set the phase value. While this method does display the exact value of the phase, it is impossible to continuously increase the phase value, and to see and hear the difference over time. I am keen to maintain the smoothness that this phase input method provides, as this is something the participants of the study highlighted that they appreciated. I would be interested to know if, had the participants also tried these alternative input methods, their feedback on the phase spinners would be more positive.

I think that the way the mouse movements are translated to the movement of the spinners may be part of the reason the participants did not find the phase control intuitive. In the current implementation, the horizontal movement of the mouse is translated to the rotational movement of the phase spinner. It is potentially more intuitive to instead have the user *rotate* the mouse for this. However, it can be quite uncomfortable and difficult to move the mouse in a circular motion, both when using a track-pad and a dedicated mouse, and I think users may lose some precision compared to the horizontal input method. I think that the best solution could be to instead make the spinner point in the direction of the mouse, relative to the centre of the spinner. This would then not necessarily require any rotational mouse movement, and could be implemented with a simple vector calculation, rather than tracking the circular movement of the mouse pointer. I would be interested to hear the participants feedback on different implementations of this in future builds.

I have also thought about how best to add more numerical values to the tool without cluttering the UI. One of the cleanest solutions to this may be to add a text box that appears next to the mouse cursor when it is hovering over a UI element and when changing a value. I would also like to allow the user to input a specific frequency, although this is a much simpler addition to the tool.

The participants also stated that the precision level when using the sliders to adjust the amplitude of harmonics was insufficient, especially at low levels. One way to address this would be to simply make the sliders bigger, but this may cause them to dominate the screen and negatively affect the overall UI design. An alternative way to improve this may be to have the slider affect the value of the *square root* of the amplitude, rather than the absolute value. This will come with an added computation cost, but it means that the user will gain more precision over the value change at the lower end of the slider, which is specifically where more precision is needed. Again, this is something with which I will experiment in future builds.

Regarding the comments on the colours and look of the tool, I will try out different colour schemes. I think I perhaps focused too hard on making sure each control stood out from the background, which resulted in the colours of the tool seeming quite garish and heavy. I think lightening the shades of blue and yellow for the background and sliders will make a big difference here, and I will rethink the red colour of the reset buttons. Lightening the background colour will also make the black text more prominent. I have attempted to make the tool run in windowed mode by default, but despite setting this in the Unity build settings, it does not seem to have worked. However, I am sure that with some further setting tweaks and research this can be achieved, and my efforts to allow the tool to run at different aspect ratios should help in this area.

Finally, concerning the quality of the user study itself, I will spend more time making sure that more relevant knowledge is available to the participants of future studies. I did not want to make the study too long or time-consuming, but also adding some more questions would have been beneficial in hindsight. For example, a question which asks about each participant's confidence in their knowledge of this area, perhaps on a scale of 1-5, would have been useful to compare to their later answers. I also would add some more specific questions about what exactly participants found difficult to control, such as input method, size of UI element and precision level, as this would allow me to focus on more specific improvements.

5.3 Comparison with M. Ruiz's Fourier Synthesiser

Due to the similarities between my tool and Michael Ruiz's Fourier Synthesiser, a comparison of the two implementations, along with a discussion of their strengths and weaknesses, will be a useful form of evaluation.

- Ruiz's synthesiser supports 16 adjustable harmonics, whereas my tool supports 21 adjustable harmonics, and allows the user to add more which cannot be adjusted.
- Ruiz's synthesiser uses text input boxes for phase values in degrees, where mine uses the phase spinners. Ruiz's synthesiser also allows text input for frequency values and harmonic amplitude values, where mine does not.
- Ruiz's synthesiser uses buttons for preset selection, where mine uses a dropdown menu.
- My tool has reset buttons for each of the harmonics, which return the amplitude and phase values to their preset position. Ruiz has a 'reset amplitudes' button, which appears to have the same functionality as the sine wave preset button.
- Ruiz's graph is normalised. I.e. it has a constant height, regardless of the number of harmonics and their amplitudes. It does not have any zoom functionality, however, which my tool does.
- As stated in 2.4.2, Ruiz's synthesiser runs in an HTML5 browser. It uses JavaScript for the functionality, and HTML and CSS for the UI positioning and appearance, whereas my implementation is written in C#, using the Unity Engine. Specifically, Ruiz uses JavaScript AudioContext oscillators, and has an individual oscillator for each harmonic. This differs from my implementation, in which I only generate one wave that is the superposition of the present harmonics. The use of the word oscillator leads me to believe that this is perhaps because of a particularly small buffer size used, or potentially no buffer at all. Unfortunately, the implementation of the AudioContext class is not publicly available so I cannot check this.
- When the harmonic phase values are changed on Ruiz's synthesiser, the graph updates accordingly, but the audio is not affected, whereas in my tool changing phase values has an audible effect.

5.3.1 Strengths and Weaknesses of Both Tools

The main strength of Ruiz's tool is its smoothness and efficiency. There is almost no audio popping and it is very responsive, especially when changing frequency. It also provides the values of each parameter to the user in text, and allows these values to be changed with keyboard input, which the participants of the user study mentioned would be a welcome addition to the tool. I think normalising the graph is a good idea as well, and implementing this in my tool would allow me to increase the size of some of the UI elements, as I would not have to consider the possibility of the wave graph growing in size. Finally, the convenience of having a tool that runs in a browser is particularly beneficial, as this requires no download or installation process, and is easily available on many modern platforms, including mobile devices.

The main advantage of my tool is undoubtedly that changing the phase values of the harmonics has an audible effect, as well as graphical. The AudioContext oscillators do not support phase changes, and so this cannot be done with Ruiz's method of implementation. This makes the user interface of Ruiz's synthesiser somewhat misleading, as the displayed waveform does not accurately represent the sound being played when the phase value of any of the harmonics is non-zero. This includes triangle wave preset values, and means that the timbre of a triangle wave is not actually produced when this button is clicked. I also believe that the ability to change the phase continuously is another advantage, and not just for the audible effect. Using numerical input as the only control of the phase values feels quite clunky. Also, seeing the shape of the wave gradually change as the phase of different harmonics are adjusted is one of my favourite aspects of the tool I have implemented.

While I think the look of Ruiz's UI is in general superior to that of my tool, I do prefer the use of a dropdown menu to change preset as opposed to several buttons. My tool also supports a larger number of harmonics, and so more configuration.

Overall, I think both tools have their benefits and drawbacks, and the fact that some elements of the two interfaces are very similar, such as the vertical amplitude sliders and ability to select different presets, confirms to me that these are somewhat intuitive choices.

Chapter 6

Conclusions

6.1 Observable Phenomena

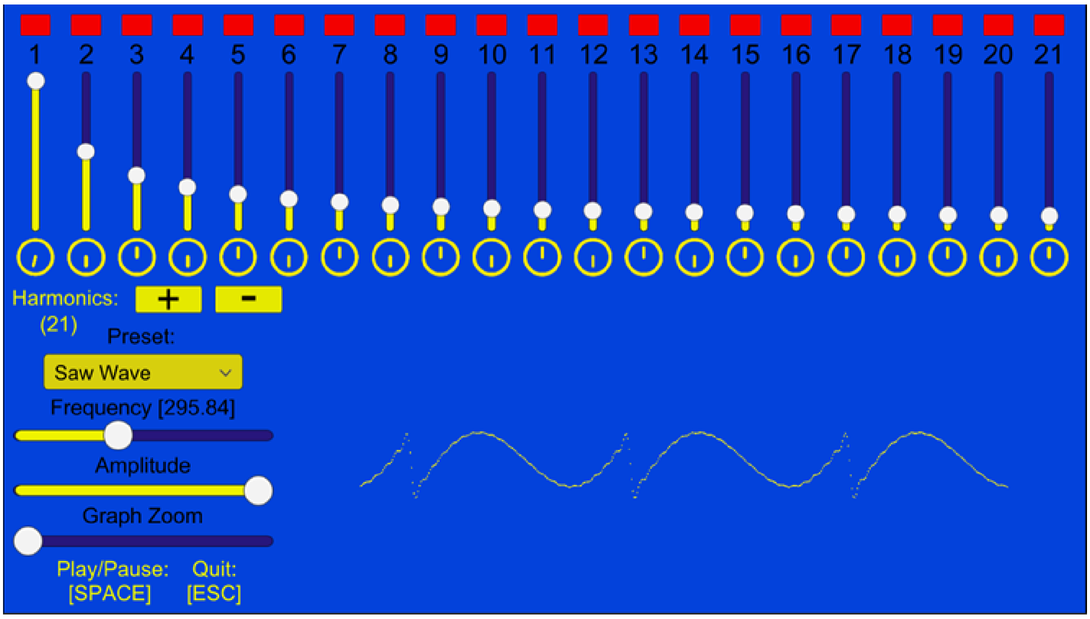
Creating this tool has allowed me to observe some expected and unexpected effects when changing parameters, including:

- **The Missing/Phantom Fundamental** - Used to test my user study, this is a psychological phenomenon that occurs when an observer listens to a sound with a number of overtones, and can still hear the fundamental frequency even when that base frequency is removed. As seen in the results of my user study, this effect can be extended to removing up to 16 harmonics, while still hearing the fundamental sound.

- **The Effect of Phase** - While manipulating the amplitude and phase values of harmonics, it is quite easy to hear the effect that is being caused. Increasing the amplitude of certain harmonics, even only by a small amount, allows one to identify the pitch of the harmonic that is being adjusted, and hear how it changes. With phase, although the effect is lessened, when rotating the phase spinners, one can hear as the interference between the different harmonics changes from constructive to destructive, and the most prominent harmonics change as the phase parameters are manipulated. This interference can also be seen in the displayed wave graph, and so it is sometimes clear exactly which peaks are being removed or emphasised. As a musician, I am fascinated by this, and cannot help but wonder about the potential artistic applications of a synthesiser that varies phase values over time to bring out different harmonics of a sound.

  Phase sometimes also seems to be able to lower the perceived pitch of the resulting sound of a wave, specifically when the first two or three harmonic phases are manipulated. I suspect this may be because this can cause some particularly large extra peaks before the completion of a wave period, as shown in Figure 6.1.

  Understanding exactly how phase affected timbre and sound was one of my main motivations for creating this tool, and I'm glad that its effect can be seen and heard so prominently.



Harmonics: (21)

Preset: Saw Wave

Frequency [295.84]

Figure 6.1: Configuration to create a perceived drop in pitch by manipulating phase values

- **The Effect of Changing Frequency** - After changing a number of different harmonic parameters, I find that my ear adjusts to the current configuration, and I can do quite a good job of separating the resulting sound into the different frequency components, especially with the more prominent harmonics. However, I can also remove this effect entirely simply by changing the fundamental frequency, even if only by a small amount. This seems to almost 'reset' my ears' adjustment, and I return to only perceiving the sound at its fundamental frequency, with the harmonics determining its timbre as opposed to other notes that I can hear.

I am glad that I have not only been able to create a tool that is useful for my dissertation, but also one that I am quite fascinated by, and I am continuing to still find new interesting effects and configurations.

6.2 Final Note

There are plenty of improvements and extensions I can make to this tool, and I will endeavour to keep working on improving the quality of this tool for future builds. I hope that this tool not only has some educational value, but also that users find themselves lost in the process of experimentation and creation, as I have many times. I have always found that there is not enough software that marries creative concepts with scientific ones, or displays this in a way that is useful, so I am happy that I can contribute to this overall body of work, and potentially do the same in the future.

Bibliography

Paul Falstad. Fourier series applet. https://www.falstad.com/fourier/, 2015. accessed = 12-April-2023.

Michael J. Ruiz. Fourier synthesiser. http://mjtruiz.com/ped/fourier/, 2017. accessed 25-March-2023.

M.R. Schroeder. Models of hearing. *Proceedings of the IEEE*, 63(9):1332–1350, 1975. doi: 10.1109/PROC.1975.9941.

Labster Theory. Longitudinal and transverse. https://theory.labster.com/longitudinal-transverse-waw/, 2022. accessed 17-March-2023.

David Abbott. *Understanding Sound*. Pressbooks, 2020.

ANSI. *American Standard Acoustical Terminology*. American National Standards Institute, 1960.

David M. Howard and Jamie A.S. Angus. *Acoustics and Psychoacoustics*. Routledge, 5th edition, 2017.

Bryan H. Suits. *Physics Behind Music: An Introduction*. Cambridge University Press, 2023a. doi: 10.1017/9781108953153.

Bryan H. Suits. Clarinet. https://pages.mtu.edu/~suits/clarinet.html, 2023b. accessed 12-April-2023.

Percy A. Scholes. *The Oxford Companion to Music*. Oxford University Press, 1970.

Wikipedia contributors. Square wave. https://en.wikipedia.org/wiki/Square_wave, 2023. accessed 12-April-2023.

Peter G.L. Dirichlet. Sur la convergence des séries trigonométriques qui servent à représenter une fonction arbitraire entre des limites données. *Journal für die reine und angewandte Mathematik*, 1829.

Eric W. Weisstein. Fourier series. https://mathworld.wolfram.com/FourierSeries.html, 2023. accessed 12-April-2023.

Ran Ma and Waiching Sun. Fft-based solver for higher-order and multi-phase-field fracture models applied to strongly anisotropic brittle materials and poly-crystals. *Computer Methods in Applied Mechanics and Engineering*, 2019. doi: 10.1016/j.cma.2019.112781.

David Gottlieb and Chi-Wang Shu. On the gibbs phenomenon and its resolution. *SIAM review*, 39(4):644–668, 1997.

SEG Contributors. Dictionary: Gibbs' phenomenon. `https://wiki.seg.org/wiki/Dictionary:Gibbs%E2%80%99_phenomenon`, 2018.

.NET contributors. Enumeration types (c# reference). `https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum`, 2023. accessed = 12-April-2023.

Hippo. Simple spinner (progress indicators for ui). `https://assetstore.unity.com/packages/2d/gui/icons/simple-spinner-progress-indicators-for-ui-237500`, 2022. accessed 12-April-2023.

Appendix A

Participants' Information Sheet

Participant Information Sheet

Project title:	Harmonising The Senses: An Audiovisual Tool For Interactive Fourier Synthesiss
Principal investigator:	Dr. John Longley
Researcher collecting data:	Ben Evans

This study was certified according to the Informatics Research Ethics Process, RT number 184640. Please take time to read the following information carefully. You should keep this page for your records.

Who are the researchers?

Ben Evans – Undergraduate Computer Science Student, University of Edinburgh

Dr. John Longley – Informatics Lecturer, University of Edinburgh

What is the purpose of the study?

To obtain feedback on the usability and quality of the UI design in the application built for the project. The feedback you give may be anonymously referred to in published articles, reports and presentations, and used to assess the quality of the UI design of the project, and the implementation of various other features.

Why have I been asked to take part?

There is no defined target group for the study. We have asked for your feedback on the application to gauge how a general set of users would respond to the application.

Do I have to take part?

No – participation in this study is entirely up to you. You can withdraw from the study at any time, up until April 6th, 2023, without giving a reason. Your rights will not be affected. If you wish to withdraw, contact the PI. We will keep copies of your original consent, and of your withdrawal request.

What will happen if I decide to take part?

- We will collect your feedback on some individual aspects of the system, as well as general feedback and comments regarding the system as a whole. This feedback will be used to assess the ability of the system to be used by a general user base. As there is no need to store or compare each individual's feedback with others, no personal or identifying data will be collected.
- There will be a questionnaire for you to fill out. The answers that you provide on this questionnaire will be the only information regarding you that is recorded.
- The session should take a maximum of 30 minutes, including time to read about the system, use the system and complete some tasks, and fill out the questionnaire at the end.
- Only one session will be needed, and can be completed remotely or in-person, at a time and place of your choosing before April 6th, 2023.

Are there any risks associated with taking part?

There are no significant risks associated with participation.

Are there any benefits associated with taking part?

There are no significant benefits associated with participation.

What will happen to the results of this study?

The results of this study may be summarised in published articles, reports and presentations. Quotes or key findings will be anonymized: We will remove any information that could, in our assessment, allow anyone to identify you. With your consent, information can also be used for future research. Your data may be archived for a maximum of 4 years. All potentially identifiable data will be deleted within this timeframe if it has not already been deleted as part of anonymization.

Data protection and confidentiality.

Your data will be processed in accordance with Data Protection Law. All information collected about you will be kept strictly confidential. Your data will be referred to by a unique participant number rather than by name. Your data will only be viewed by the researcher/research team (Ben Evans and Dr. John Longley).

All electronic data will be stored on a password-protected encrypted computer, on the School of Informatics' secure file servers, or on the University's secure encrypted cloud storage services (DataShare, ownCloud, or Sharepoint) and all paper records will be stored in a locked filing cabinet in the PI's office. Your consent information will be kept separately from your responses in order to minimise risk.

What are my data protection rights?

The University of Edinburgh is a Data Controller for the information you provide. You have the right to access information held about you. Your right of access can be exercised in accordance Data Protection Law. You also have other rights including rights of correction, erasure and objection. For more details, including the right to lodge a complaint with the Information Commissioner's Office, please visit www.ico.org.uk. Questions, comments and requests about your personal data can also be sent to the University Data Protection Officer at dpo@ed.ac.uk.

Who can I contact?

If you have any further questions about the study, please contact the lead researcher, Ben Evans – s1907069@ed.ac.uk .

If you wish to make a complaint about the study, please contact inf-ethics@inf.ed.ac.uk. When you contact us, please provide the study title and detail the nature of your complaint.

Updated information.

If the research project changes in any way, an updated Participant Information Sheet will be made available on http://web.inf.ed.ac.uk/infweb/research/study-updates.

Alternative formats.

To request this document in an alternative format, such as large print or on coloured paper, please contact Ben Evans – s1907069@ed.ac.uk .

General information.

For general information about how we use your data, go to: edin.ac/privacy-research

Appendix B

Participants' Consent Form

Participant number:________________________

Participant Consent Form

Project title:	Building a Fourier Synthesiser in Unity
Principal investigator (PI):	Dr. John Longley
Researcher:	Ben Evans
PI contact details:	J.R.Longley@ed.ac.uk

By participating in the study you agree that the data you provide can be used and referred to in published articles, reports and presentations, and used to assess the quality of the UI design of the project, and the implementation of various other features.

- I have read and understood the Participant Information Sheet for the above study, that I have had the opportunity to ask questions, and that any questions I had were answered to my satisfaction.
- My participation is voluntary, and that I can withdraw at any time without giving a reason. Withdrawing will not affect any of my rights.
- I consent to my anonymised data being used in academic publications and presentations.
- I understand that my anonymised data will be stored for the duration outlined in the Participant Information Sheet.

Please tick yes or no for each of these statements.

1. I allow my data to be used in future ethically approved research. □ **Yes** □ **No**

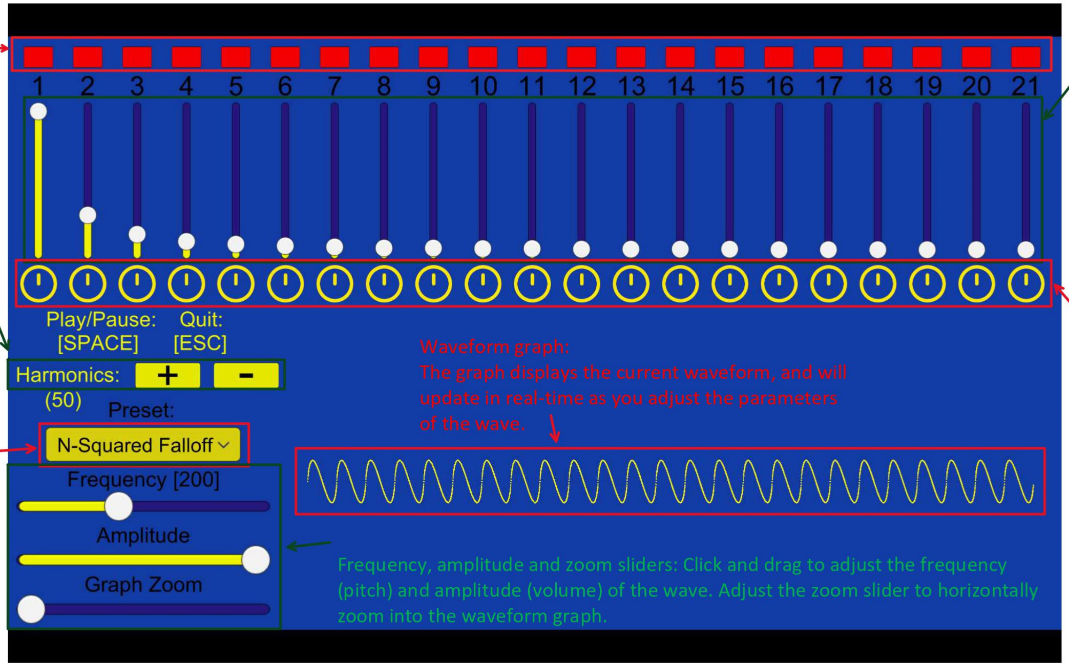
2. I agree to take part in this study. □ **Yes** □ **No**

Name of person giving consent ______________ Date ______________ Signature ______________

Name of person taking consent ______________ Date ______________ Signature ______________

Appendix C

Controls Image



Reset buttons:
Click to reset amplitude and phase values of respective harmonic to the default values determined by the current **preset**.

Harmonic add/remove buttons:
Click **+** to add a new highest harmonic and **-** to remove the current highest harmonic. You can add as many harmonics as you like, but only the first 21 will be adjustable. Alternatively, use the up and down arrow keys. The number of active harmonics is displayed in brackets.

Preset dropdown:
Click to open the dropdown menu and select a different preset. On selection of a new preset, all harmonics will update to the preset position.

Harmonic amplitude sliders:
Click and drag to adjust the relative amplitude of each harmonic (labelled above)

Phase control knobs:
Click and drag your mouse left or right to adjust the phase of each harmonic. Double click to reset to 0, or if already 0, double click to set value to **π**.

Waveform graph:
The graph displays the current waveform, and will update in real-time as you adjust the parameters of the wave.

Frequency, amplitude and zoom sliders: Click and drag to adjust the frequency (pitch) and amplitude (volume) of the wave. Adjust the zoom slider to horizontally zoom into the waveform graph.

Figure C.1: Description of controls given to participants of the user study

Appendix D

User Study

The user study can be viewed at the following link: `https://forms.gle/EtLC5YsYAcR82GFG8`